Understanding Control Flow
with Concurrent Programming using μC++

(Draft)

Peter A. Buhr

December 28, 2006

Contents

Preface

Learning to program involves progressing through a series of basic concepts; each is fundamentally new and cannot be trivially constructed from previous ones. Programming concepts can be divided into two broad categories: data flow and control flow. While many books concentrate on data flow, i.e., data structures and objects, this book concentrates on control flow. The basic control-flow concepts are conditional and loop, subroutine and object member call/return, recursion, exception, coroutine, and concurrency; each of these concepts is fundamental because it cannot be trivially constructed from the others. The difficulty in understanding and using these concepts grows exponentially from loops to concurrency, and a programmer must understand all of these concepts to be able to solve the complete gamut of modern programming problems.

Control flow issues are extremely relevant in modern computer languages and programming styles. In addition to the basic control-flow mechanisms, virtually all new computer languages provide some form of exceptional control flow to support robust programming. As well, concurrency capabilities are appearing with increasing frequency in both new and old programming languages. The complexity issues and reliability requirements of current and future software demand a broader knowledge of control flow; this book attempts to provide that knowledge.

The book starts with looping, and works through each of the basic control-flow concepts, examining why each is fundamental and where it is useful. Time is spent on each concept according to its level of difficulty, with coroutines and concurrency receiving the most time because they should be new to the reader and because of their additional complexity. The goal is to do for control flow what is done for data structures in a data structure book, i.e., provide a foundation in fundamental control flow, show which problems require certain kinds of control flow, and which language constructs provide the different forms of control flow. In essence, a good data structure book should provide its reader with the skills to select the proper data structure from a palette of structures to efficiently solve a problem; this book attempts to provide the same skill for control flow.

The book is intended for students learning advanced control flow and concurrent programming *at the undergraduate level*. Often the only advanced control-flow students see is concurrency, and this material is usually presented as a secondary issue in a fourth year operating systems and/or database course. The amount of undergraduate time spent on concurrency is usually only 2 to 3 weeks of classes. Only if a student progresses to graduate school is there an opportunity to obtain additional material on concurrency. However, the need for advanced control flow and especially concurrency is growing at a tremendous rate. New programming methodologies are requiring new forms of control flow, and new programming languages are supporting these methodologies with new control structures, such as concurrency constructs. Also, new computers contain multiple processors and distributed systems are ubiquitous, which require these advanced programming methodologies to take full advantage of the available capabilities. Therefore, many of these advance forms of control flow are becoming basic programming skills needed by all programmers, not just graduate students working in the operating system or database disciplines.

The material in the book is presented *bottom up*: basic concepts are presented first, with more advanced concepts built slowly and methodically on previous ones. Because many of the concepts are difficult and new to most readers, extra time is spent and additional examples are presented to ensure each concept is understood before progressing to the next.

Prerequisites

The reader is expected to have the following knowledge:

- intermediate programming experience in some object-oriented programming language;
- an introduction to the programming language C++.

Furthermore, the kinds of languages discussed in this book are imperative programming languages, such as, Pascal, C, Modula-3, ML, Ada, etc. The reader is expected to have a good understanding of these kinds of programming languages.

μC++

While C++ is used as the base programming language in this book, C++ does not provide all the advanced control-flow mechanisms needed; most importantly, C++ lacks concurrency features. Bjarne Stroustrup, the creator of C++, said the following on this point:

> My conclusion at the time when I designed C++ was that no single model of concurrency would serve more than a small fraction of the user community well. I could build a single model of concurrency into C++ by providing language features that directly supported its fundamental concepts and ease its use through notational conveniences. However, if I did that I would favor a small fraction of my users over the majority. This I declined to do, and by refraining from doing so I left every form of concurrency equally badly supported by the basic C++ mechanisms. [Str96]

However, having a concrete programming language for use in both examples and assignments provides a powerful bridge between what is learned and how that knowledge is subsequently used in practice. While much of the material in this book is language independent, at some point it is necessary to express the material in a concrete form. To provide a concrete form, this book uses a dialect of C++, called μC++, as a vehicle to explain ideas and present real examples.[1] μC++ was developed in response to the need for expressive high-level language facilities to teach advanced control-flow and concurrency [BDS+92]. Basically, μC++ extends the C++ programming language [Str97] in the same way that C++ extends the C programming language. The extensions introduce new objects that augment the existing control flow facilities and provide for light-weight concurrency on uniprocessor and parallel execution on multiprocessor computers. Nevertheless, μC++ has its own design bias, short comings, and idiosyncrasies, which make it less than perfect in all situations. These problems are not specific to μC++, but occur in programming-language design, which involves many compromises to accommodate the practical aspects of the hardware and software environment in which languages execute.

This book does not cover all of the details of μC++, such as how to compile a μC++ program or how to use some of the more complex μC++ options. These details are covered in the *μC++ Annotated Reference Manual* [Buh06]. When there is a discrepancy between this book and the *μC++ Annotated Reference Manual*, the manual always takes precedence, as it reflects the most recent version of the software. Unfortunately, the current dynamic nature of software development virtually precludes any computer science textbook from being perfectly up to date with the software it discusses.

Programming Questions

Many of the programming questions at the end of the chapters are quite detailed. (Some might argue they are not detailed enough.) The reason for the detail is that there are just too many ways to write a program. Therefore, some reasonable level of guidance is necessary to know what is required for a solution and for subsequent marking of the assignment.

Acknowledgments

Greg Andrews (SR), Igor Benko (software solutions), Roy Krischer and Richard Bilson (exceptions), Ashif Harji (exceptions, automatic-signal monitors), Ric Holt (deadlock), Doug Lea (Java), Isaac Morland (Modula-3), Prabhakar Ragde (complexity). Robert Holte, Paul Kates, Caroline Kierstead, John Plaice (general discussion and proofreading).

[1] A similar approach was taken to teach concurrency using a dialect of Pascal, called Pascal-FC (Functionally Concurrent) [BD93].

Chapter 1

Introduction

The purpose of this book is to study and analyse control flow. While the book's main emphasis is concurrent programming, which is the most complex form of control flow, it is important to lay a strong foundation in all the different forms of control flow that can be used in programming. Separating control flow issues from concurrency issues allows a step-wise presentation of the aspects of concurrency, rather than presenting the issues at once and trying to deal with all their complexity. Therefore, the next three chapters do not deal with concurrency, but with advanced control flow techniques used in writing both sequential and concurrent programs, and subsequently used throughout this book.

1.1 Control Flow

The following is a list of different kinds of control flow available in modern programming languages by increasing order of complexity:

1. basic control structures;
2. routine/member call/return;
3. recursion;
4. exception;
5. coroutine;
6. concurrency.

Some of the elements in the list are programming language constructs that provide particular forms of control flow, while other elements are particular techniques that achieve advanced forms of control flow using the language constructs. The reader should be familiar with the first three items in the list. Each element is discussed briefly now, and then, in detail in subsequent chapters.

Basic Control Structures Selection and looping constructs, such as IF and WHILE statements, are the basic control flow mechanisms used in a program and taught at the beginning of any programming course. These constructs allow virtually any control flow patterns *within* a routine. Without these control structures, a computer is nothing more than a calculator, unable to make decisions or repeatedly perform operations on its own. The most general of the basic control structures is the GOTO statement, which allows arbitrary transfers of control. However, arbitrary transfers of control can result in control flow that is difficult to understand. Therefore, certain limits are often placed on the kinds of control flow allowed both in the programming language or by the programmers using the programming language.

Routine/Member Call/Return Subroutines, subprograms, functions or procedures are used to parameterize a section of code so that it can be reused without having to copy the code with different variables substituted for the parameters. Routines provide reuse of knowledge and time by allowing other programs to call the routine, substituting values from different argument variables for parameter variables in the computation of the routine. **Modularization** of programs into reusable routines is the foundation of modern software engineering. To support this capability, a programming language allows virtually any contiguous, complete block of code to be factored into a subroutine and called from anywhere in the program; conversely, any routine call can be replaceable by its corresponding routine body with appropriate substitutions of arguments for parameters. The ability to provide modularization is an important design factor when developing control structures to support software-engineering techniques.

Routine activation (invocation) introduces a complex form of control flow. When control reaches a routine call, the current execution context is temporarily suspended and its local state is saved, and the routine begins execution at its beginning along with a new set of its local variables. When the routine returns, the routine's local variables are destroyed and control is implicitly returned to the point of the routine call, reactivating any saved state. From the control perspective, the programmer does not have to know where a routine is or how to get back to the call; it just happens. In fact, there is no magic, the compiler stores the location of the routine at each call site, and as part of a routine call, the location of where to return is saved so that the routine can transfer back. In most programming languages, it is impossible to build a routine call/return using the basic control flow constructs; hence, a routine call/return is a fundamental mechanism to affect control flow.

It is common for one routine to call another, making routines one of the basic building blocks for program construction. When routines call one another, a chain of suspended routines is formed. A call adds to one end of the chain; a return removes from the same end of the chain. Therefore, the chain behaves like a stack, and it is normally implemented that way, with the local storage of each routine allocated consecutively from one end of storage for each call and removed from that same end when the routine returns.

The notion of a record (or structure) for grouping heterogeneous data types exists in most programming languages. As well, many programming languages support nested routines (C++ does not support nested routines). An **object** is a combination of these two notions: nesting a routine in a record. Both a routine and a record define a static declaration scope that can be used in exactly the same way by a routine nested in either. Therefore, allowing the definition of routines in a record follows naturally from the static declaration scope present by a record and the record instance acts like a routine activation.

However, from a control flow perspective, there is no difference in calling a routine or a member routine of an object. While the environment in which the member routine executes is different from the environment of a routine because the member routine can access an object's context as well as the static context where the object is defined, the control flow is still a basic routine call/return.

Recursion While it is common for one routine to call another, it is uncommon for a routine to call itself, either directly or indirectly, but if it does, the calls form a cycle, called **recursion**. The cycles formed by recursion are analogous to looping with control structures *with the addition that a new instance of the routine is created for each call*.

Recursion is often confusing, and there is a good reason for this: there are very few physical situations where recursion occurs, and therefore, there are no common analogies that a person can use to understand recursion. The most common examples of physical recursion are standing between two parallel mirrors or pointing a T.V. camera at the T.V. screen it is broadcasting to. In theory, there is an infinite number of copies of the original entity; in practice, the number of repeated instances drops off very quickly. Alternatively, it is straightforward to imagine looping as simply doing an operation over and over. For example, in carrying bricks from a truck to a house, a person might make repeated trips carrying one or more bricks at a time. With recursion, a simple analogy cannot be constructed. Essentially a person carries one or more bricks, clones oneself back at the truck, and the clone carries the next load of bricks; after making N trips, there are N copies of the person at the house, and then, $N - 1$ of them disappear. While both approaches are equally valid ways of moving the bricks, the latter approach is not one that would immediately suggest itself to most people.

(Interestingly, recursive data structures, such as a linked list or a tree data structure, have a similar problem. These structures are rarely specified, manipulated or implemented using direct recursive techniques; exceptions are languages where all data and/or objects are implicitly referenced via pointers, like CLU [LAB+81], Ada [Uni83], and Java [GJSB00]. In all cases, the data recursion is indirect through an explicit or implicit pointer in the data structure. While it might be argued that this is an implementation issue, the fact that it exists in virtually all programming languages indicates that direct recursive data-structures are more of a concept than a reality.)

While recursion is not intuitive, the class of problems that require building up state information during the execution of an algorithm may be trivially implemented by a recursive solution. These algorithms use the local state of a routine to implicitly build a stack data-structure that remembers both data and execution state. For example, recursive solutions to many tree-walking algorithms are usually the simplest because the implicit stack created by the routine calls is required by these algorithms and is implicitly managed by the normal routine call mechanism. By informally decomposing these algorithms into control flow and implicit data-structure, it is often possible to easily understand them and their recursive solutions.

Like all programming techniques, recursion can be abused, for example, by constructing a complex recursive solution using the implicit stack, when using a different, explicit data-structure would produce a straightforward solution. An example of this problem is the obvious recursive solution for calculating the Fibonacci sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Recursive

```
int Fib( int n ) {
    if ( n == 0 )
        return 0;
    else if ( n == 1 )
        return 1;
    else
        return Fib( n - 1 ) + Fib( n - 2 );
}
```

Looping

```
int Fib( int n ) {
    int fn = 0, fn1 = 0, fn2 = 1;
    for ( int i = 0; i < n; i += 1 ) {
        fn = fn1 + fn2;
        fn2 = fn1; fn1 = fn;
    }
    return fn;
}
```

The recursive solution (left example) builds multiple stacks to calculate each partial Fibonacci number in the sequence up to the initial argument value (exponential complexity in space and time, specifically $\Theta(\phi^n)$), while the looping solution (right example) uses only 3 variables and n interations. While excellent recursive solutions for this problem exist, a novice programmer can unknowingly generate a space and time inefficient recursive solution because of the implicit aspects of recursion. Hence, it is often essential for a programmer to understand both a control-flow technique and its underlying details to construct an efficient program.

Exception Basic control structures allow for virtually any control flow pattern *within* a routine. However, control flow *among* routines is rigidly controlled by the call/return mechanism. If routine A calls routine B, the normal way for A to affect the control flow of B is indirectly, by passing arguments to B. One common mechanism to directly affect control flow is for A to pass a routine pointer as an argument to B, which B then calls. Similarly, the only way for B to affect the control flow of A is through its return value and output parameters. One common mechanism is for B to return a flag, called a **return code**, which A can use to influence its control flow. Note that in both cases programming conventions are used to implement more sophisticated control flow patterns using only the basic control structures and call/return.

When intermediate routines exist between A and B, e.g., if A calls M and M calls B, it becomes much more difficult to use these conventions to allow A and B to influence each other's control flow. In this general case, all routine calls between A and B must be modified to pass the necessary extra arguments from A to B, and the necessary extra return values from B to A. Transferring this additional information among the intermediate levels can be tedious and error prone, resulting in significant complication.

Exceptions extend call/return semantics to reference and transfer in the "reverse" direction to normal routine calls. Instead of passing explicit routine pointers through multiple levels, routines are implicitly passed so any intermediate routine are unaffected. As well, control flow is extended to allow a direct transfer from a called routine through multiple intermediate called routines back to a specific calling routine. These additional forms of control flow support a variety of programming techniques that cannot be simulated well with basic control flow mechanisms.

One common programming technique where exception are used is dealing with error situations detected in a lower-level routine invocation, allowing the dynamic selection of a corrective routine and possibly a transfer of control to some stable recovery point in the call chain if correction is impossible. However, not all exceptions result from errors; some exceptions only indicate infrequently occurring events, which are handled outside of the normal algorithmic control sequence.

Coroutine Like exceptions, coroutines provide a new mechanism for transferring control among routines. What differentiates a coroutine from a routine is that a coroutine can suspend its execution and return to the caller *without* terminating. The caller can then resume the coroutine at a later time and it restarts from the point where it suspended, continuing with the local state that existed at the point of suspension.

While this form of control flow may seem peculiar, it handles the class of problems called finite and push-down automata (and more). As well, this flow of control is also the precursor to true tasks but without concurrency problems; hence, a coroutine allows incremental development of these properties [Yea91].

Concurrency In all the previous discussions of control flow, there has been the implicit assumption of a single point of execution that moves through the program, controlled by the values in variables and the control structures. As a consequence, a program is only ever executing at one particular location. Concurrency is the introduction of multiple points of execution into a program. As a consequence, a program now has multiple locations where execution is occurring. Reasoning, understanding, and coordinating these multiple execution points as they interact and manipulate shared data is what concurrent programming is about.

1.2 Summary

The minimal set of control flow concepts needed in modern programming languages are: basic control flow, (member) routine call/return, recursion, exception, coroutine, and concurrency. While not all languages support this set of concepts directly, it is essential to be able to recognize these basic forms, and then adapt, as best as possible, to the specific implementation language/system.

Chapter 2

Advanced Control Flow*

This chapter reviews basic control flow and then discusses advanced control flow that *cannot* easily be simulated using the basic control structures.

2.1 Basic Control Flow

Basic control structures allow for virtually any control flow pattern *within* a routine. The basic control structures are:

- selection: if-then-else, case;
- looping: while, repeat, for;

which are used to specify most of the control flow in a program. In fact, only the while-loop is necessary to construct any control flow as it can be used to mimic the others. The proof of this point can be presented informally by showing a few specific transformations and the reader can generalize from the examples. The case statement is just a series of disjuncted if-statements and the for-loop is just a while-loop with a counter so it is sufficient to show the transformations from the if-then-else and repeat-loop constructs to the while-loop:

IF	IF using WHILE	REPEAT	REPEAT using WHILE
	`flag = true;`		`x += 1; // prime loop`
`if ( a > b ) {`	`while ( a > b & flag ) {`	`do {`	`while ( x < 10 ) {`
	`    flag = false;`	`    x += 1;`	`    x += 1;`
`    b = 2;`	`    b = 2;`	`} while ( x < 10 );`	`}`
	`}`		
`} else {`	`while ( flag ) {`		
	`    flag = false;`		
`    b = 7;`	`    b = 7;`		
`}`	`}`		

In the if-then-else simulation (left example), the `flag` variable ensures the loop body executes at most once and the second while-loop does not execute if the first one does; in the repeat simulation, the priming of the while-loop with the loop body ensures it is executed at least once. However, even though the while-loop can mimic the other control structures, it is clear that programming with only a while-loop is awkward. Therefore, there is a significant difference between being able to do a job and doing the job well; unfortunately, it is hard to quantify the latter, and so it is often ignored. A good analogy is that a hammer can be used to insert screws and a screwdriver can pound nails so they are weakly interchangeable, but both tools have their own special tasks at which they excel. This notion of **weak equivalence** is common in computer science, and should be watched for because it does not take into account many other tangible and intangible factors. Weak equivalences appear repeatedly in this book.

The remainder of this chapter discusses several advanced forms of control flow: multi-exit loop and static multi-level exit. Each form allows altering the flow of control in a program in a way that is not trivially possible with the basic control structures. Again, there is a weak equivalence between the new forms of control and the while-loop,

*This chapter is a revision of the work published as “A Case for Teaching Multi-exit Loops to Beginning Programmers” in [Buh85].

but each new form provides substantial advantages in readability, expressibility, maintainability and efficiency, which justifies having specific language constructs to support them.

2.2 GOTO Statement

Conspicuous by its absence from the list of basic control structures is the GOTO statement. The reason is that most programming styles do not advocate the use of the goto, and some modern programming languages no longer provide a GOTO statement (e.g., Java). However, the much maligned goto is an essential part of control flow and exists implicitly in virtually all control structures. For example, Figure 2.1 shows the implicit GOTO statements that exist in both the if-then-else and while-loop. For the if-then-else (top), there are two gotos: the first transfers over the then-clause if the condition is false (notice the reversal of the condition with the not operator), and the second transfers over the else-clause after executing the then-clause if the condition is true. For the while-loop (bottom), there are also two gotos: the first transfers over the loop-body if the condition is false (notice the reversal of the condition with the not operator), and the second transfers from the bottom of the loop-body to before the loop condition. The key point is that programming cannot exist without the goto to transfer control. Therefore, it is necessary to understand the goto even when it is used implicitly within high-level control structures. By understanding the goto, it is possible to know when it is necessary to use it explicitly and when it is being used correctly.

Implicit Transfer	Explicit Transfer
if (C) { *// false => transfer to else* . . . *// then-clause* *// transfer after else* } **else** { . . . *// else-clause* }	**if** (! C) **goto** L1; . . . *// then-clause* **goto** L2; L1: . . . *// else-clause* L2:
while (C) { *// false => transfer after while* . . . *// loop-body* } *// transfer to start of while*	L3: **if** (! C) **goto** L4; . . . *// loop-body* **goto** L3; L4:

Figure 2.1: Implicit GOTO

Given that the goto is essential, why is it a problem? It was discovered in the initial development of programming that arbitrary transfer of control results in programs that are difficult to understand and maintain:

> For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of GO TO statements in the programs they produce. [Dij68b].

During the decade of the 1970's, much work was done to understand the effect of different restrictions on transfer of control with regard to expressibility versus understandability and maintainability. This research has the generic name **structured programming**. The conclusion was that a small set of control structures used with a particular programming style make programs easier to write and understand, as well as maintain. In essence, a set of reasonable conventions were adopted by programmers, similar to adopting a set of traffic rules for car drivers. Not all structured programming conventions are identical, just as traffic conventions vary from country to country. Nevertheless, the conventions have a large intersection, and therefore, it is only necessary to learn/adjust for any small differences.

The basis for structured programming is the work of Böhm and Jacopini [BJ66], who demonstrated that any program written with gotos can be transformed into an equivalent program that uses only IF and WHILE control structures (actually only WHILE is necessary). The transformation requires the following modifications to the original goto program:

- copying fragments of the original program, and
- introducing **flag variables** that are used only to affect control flow, and thus, do not contain data from the original program.

While these transformations are necessary when going from a GOTO program to a structured program, they are frequently required when programming with IF and WHILE even if the program is not initially written with GOTOs. However, copying code introduces problems such as maintaining consistent copies. Although this problem is mitigated by using routines, routines introduce additional declarations, calls, and execution cost, especially if many parameters are required. Similarly, introducing flag variables requires extra declarations and executable statements to set, test, and reset the flags. In both these cases, additional complexity is introduced into the program; this complexity is largely a consequence of the simple control structures in the programming language and not the implementation of a particular algorithm.

Peterson, Kasami and Tokura [PKT73] demonstrated that some copying of code and all of the flag variables required by the Böhm and Jacopini transformation can be eliminated by using a new control structure. This control structure involves an EXIT statement and requires that some of the loops in the program be labelled. The EXIT statement transfers control after the appropriately labelled control structure, exiting any intervening (nested) control structures; for example, in:

```
A: LOOP
    ...
    B: LOOP
        ...
        EXIT A
        ...
        EXIT B
        ...
    END LOOP
    ...
END LOOP
```

EXIT A transfers control to the statement after (the END LOOP of) the loop labelled A and EXIT B transfers control to the statement after the loop labelled B. Fundamentally, EXIT is a GOTO but restricted in the following ways:

- EXIT *cannot* be used to create a loop, i.e., cause a backward branch in the program, which means only looping constructs can be used to create a loop. This restriction is important to the programmer since all the situations that result in repeated execution of statements in a program are clearly delineated by the looping constructs.

- Since EXIT always transfers out of containing control structures, it cannot be used to branch into a control structure. This restriction is important for languages allowing declarations within the bodies of control structures. Branching into the middle of a control structure may not create the necessary local variables or initialize them properly.

Programs written in languages without an exit control structure often contain flag variables that are used to determine flow of control at different times and stages during execution. In fact, these flag variables are as undesirable as gotos because they detract from readability, must be continually set and reset, and often must be tested repeatedly in a number of places. A flag variable used strictly for control flow purposes and not containing information about actual data being manipulated is the variable equivalent of the GOTO statement. This is because modification of a flag variable can be buried in a loop body or even appear in remote parts of the program. Thus, exits are a valuable programming tool to simplify control flow.

Three different forms of exits are identified: one in which control leaves only the current loop control structure at possibly several different locations, called a **mid-exit** or **multi-exit loop**; one in which control leaves multiple levels but to a statically determinable level, called **static multi-level exit**; and one in which control leaves multiple levels but to a level that is determined dynamically, called **dynamic multi-level exit**. In the case of multi-exit loop, the level of exit is determined implicitly, and hence, it is unnecessary to name program levels nor to use that name to exit the loop. However, terminating multiple control structures either statically or dynamically requires additional naming. Because the multi-exit loop occurs more frequently than the other kinds of exits, and absence of names reduces program complexity, many languages have a specific construct for direct exit from a loop. However, multi-exit loop and static multi-level exit can be combined into a single construct and some languages do this.

2.3 Mid-Exit/Multi-Exit Loop

The mid-exit/multi-exit loop is a loop that can have one or more exit locations from anywhere in the body of the loop. Since a multi-exit loop exits from the body of the loop, no condition is needed at the beginning or end of the loop. Therefore, it is necessary to start an apparent infinite loop, which subsequently exits from within the loop body. Infinite loops can be created in C++ in the following ways:

```
while ( true ) {        for ( ;; ) {        do {
    . . .                   . . .               . . .
}                       }                   } while ( true );
```

In this book, the **for** version is used because it is more general; that is, it can be easily modified to have a loop index. For example, the **for** version above can be easily changed into:

```
for ( i = 0; i < 10; i += 1 ) {
    . . .
}
```

without having to change existing text in the program. If a **while** construct is used to generate an infinite loop, it must be changed to a **for** construct if a loop index is introduced. (Alternatively, the loop index could be added to the **while** construct by adding multiple statements, but this defeats the purpose of having a **for** construct and increases the chance of additional errors when adding or removing the loop index because three separate locations must be examined instead of one.) This style is used wherever possible in this book, that is, programs are written and programming language constructs are used so that new text can be inserted or existing text removed without having to change any other text in the program. This approach ensures that any errors introduced by a change are only associated with new code, never existing code. (Unfortunately, it is occasionally necessary to violate this style to make examples fit on a page of this book.)

The exit from the body of the loop is done with two statements in C++: **if** and **break**. The **if** statement checks if the loop should stop at this point, and the **break** statement terminates the loop body and continues execution after the end of the loop body. Notice that both the while and repeat loops can be implemented with a loop exit, as in:

```
// simulate WHILE               // simulate REPEAT
for ( ;; ) {                    for ( ;; ) {
  if ( i >= 10 ) break;             . . .
    . . .                         if ( i >= 10 ) break;
}                               }
```

There is one important difference, the condition of the **if** statement is reversed; the condition specifies why the loop exits instead of why the loop should continue execution. The advantage of using loop exit is the ability to change from one kind of loop to another solely by moving the exit line; no other changes are necessary. To change a while loop to a repeat loop requires changing several lines in the program, which increases the possibility of introducing errors.

A more interesting use of the loop exit is to exit from locations other than the top or bottom of the loop, for example:

```
for ( ;; ) {
    . . .
  if ( . . . ) break;     // exit from middle
    . . .
}
```

Exits of this form are used to eliminate copied code necessary with while and repeat loops. The most common example of this is reading until end of file, as in the following C++ code fragments:

```
cin >> d; // priming            for ( ;; ) {
while ( cin.eof() ) {               cin >> d;
    . . .                         if ( cin.eof() ) break;
    cin >> d;                       . . .
}                               }
```

The left example shows traditional **loop priming**, which duplicates the reading of data both before the loop and at the end of the loop body. The reason for the initial read is to check for the existence of any input data, which sets the end-of-file indicator. However, duplicated code is a software maintenance problem. The right example uses loop exit to eliminate the duplicate code, and hence, has eliminated the software maintenance problem of managing duplicate code. A similar C++ idiom for this example is:

```
while ( cin >> d ) {
    ...
}
```

but results in side-effects in the expression and precludes analysis of d without code duplication, e.g., print the status of stream cin after every read for debugging purposes, as in:

```
while ( cin >> d ) {                     for ( ;; ) {
    cout << cin.good() << endl;              cin >> d;
                                             cout << cin.good() << endl;
    ...                                    if ( cin.eof() ) break;
}                                            ...
cout << cin.good() << endl;              }
```

In the C++ idiom version (left example), printing the status of the read, i.e., did it fail or succeed, must be duplicated after the loop to print the status for end-of-file. In the mid-exit version (right example), the status can be printed or analysed simply by inserting one line of code.

It is also possible to have multiple exit points from a loop, as in:

```
for ( ;; ) {
    ...
  if ( C1 ) break;
    ...
  if ( C2 ) break;
    ...
}
```

This loop has two reasons for terminating, such as no more items to search or found the item during the search. Using a while loop forces an awkward conjunction in the condition, as in:

```
while ( ! C1 & ! C2 ) ...
```

which precludes executing code before and between the calculation of the conditional expression.

Finally, an exit can also include code that is specific to that exit, specified as:

```
for ( ;; ) {
    ...
  if ( ... ) {
        // specific code for exit
        ...     ← second level of indentation to emphasize the exit-body
        break;
    }
    ...
}
```

Here the statements of the then-clause of the exiting **if** statement are executed *before* exit from the loop. For a loop with only a single exit, any specific exit-code can be placed after the loop. However, when there are multiple exits with specific exit-code, placing all the exit codes after the loop requires additional **if** statements to determine which code should be executed based on the reason for the loop exit. In some situations, it is impossible to retest an exit condition after the loop has exited because the exit condition is transitory, e.g., reading a sensor that resets after each read. The alternative is to set a flag variable, which is tested at the end of the loop. These additional tests are eliminated by using exit code.

2.3.1 Loop Exit Criticisms

While multiple exits from a loop is an extremely powerful control flow capability, it has its detractors. Critics of loop exits often claim exits are difficult to find in the loop body. However, this criticism is simply dealt with by outdenting the exit, as in all the examples given thus far, which is the same indentation rule used for the **else** of the if-then-else:

```
// not outdented              // outdented
if (...) {                    if (...) {
    ...                           ...
    } else {                  } else {
    ...                           ...
}                             }
```

In the left example, it is difficult to locate the **else** clause because the **else** is not outdented; in the right example, the outdenting of the **else** clause makes it trivial to locate. In addition (or possibly as an alternative to outdenting), comments can be used to clearly identify loop exits.

Critics of loop exit also claim such a construct is not "structured", i.e., it does not follow the doctrine originally presented for structured programming, which is to use only **if** and **while**. However, this is a very restricted and conservative view of structured programming and does not reflect current knowledge and understanding in either programming language design or software engineering. That is, it makes no sense to summarily reject the multi-exit loop when its complexity is virtually equivalent to a while loop and yet it simplifies certain coding situation.

Finally, folding specific exit into a loop body can make it difficult to read when the exit-specific code is large. This criticism is true, and so programmers must use good judgment when considering this technique. The key point is to understand the potential techniques available to solve a problem and select the most appropriate technique for each individual case.

2.3.2 Linear Search Example

A good example that illustrates the complexity of control flow is a linear search of an array to determine if a key is present as an element of the array. In addition, if the search finds the key, the position in the array where the key is found must be available. The two common errors in this program are:

- using an invalid subscript into the array. This problem often happens when the list is full and the key is not in the list; it is easy to subscript one past the end of the list.

- off-by-one error for the index of the array element that is the same as the search key, when the key is found. This problem often happens if the loop index is advanced too early or too late with respect to stopping the search.

Here is a solution using only **if/while** and basic relational/logical operators:

```
i = -1; found = 0;
while ( i < size - 1 & ! found ) {
    i += 1;
    found = key == list[i];
}
if ( found ) {
    ... // found
} else {
    ... // not found
}
```

Why must the program be written this way? First, if the condition of the while-loop is written like this:

```
while ( i < size & key != list[i] ) {
```

a subscript error occurs when i has the value size because the bitwise logical operator & evaluates both of its operands (and C arrays are subscripted from zero). There is no way around this problem using only **if/while** constructs and basic relational/logical operators. Second, i must be incremented *before* key is compared in the loop body to ensure it has the correct value should the comparison be true and the loop stops. This results in the unusual initializing of i to -1, and the loop test working in the range -1 to $N-2$. Finally, the variable found is used solely for control flow purposes, and therefore, it can be eliminated.

Many C programmers believe there is a simple solution using only the **if** and **while** constructs (**for** is used here to simplify it even further):

```
for ( i = 0; i < size && [key != list[i]]; i += 1 ) {}
if ( i < size ) {                  ↑ only executed if i < size
    . . . // found
} else {
    . . . // not found
}
```

However, there are actually three control structures used here: **if**, **for**, and &&. Unlike the bitwise operator &, the operation && is not a traditional operator because it may not evaluate both of its operands; if the left-hand operand evaluates to false, the right-hand operand is not evaluated. By not evaluating list[i] when i has the value size, the invalid subscript problem is removed. Such partial evaluation of an expression is called **short-circuit** or **lazy** evaluation. A short-circuit evaluation is a flow-control construct and not an operator because it cannot be written in the form of a routine, as a routine-call eagerly evaluates its arguments before being invoked in most programming languages.[1] Only operators in C++ can be written in the form of a routine. Therefore, the built-in logical constructs && and || are not operators but rather control structures in the middle of an expression. Furthermore, to understand both constructs requires a basic understanding of boolean algebra: for && it necessary to understand $false \wedge ? = false$, and for || to understand $true \vee ? = true$.

Alternatively, the search can be written using a multi-exit loop:

```
for ( i = 0; ; i += 1 ) {
  if ( i == size ) break;
  if ( key == list[i] ) break;
}
if ( i < size ) {
    . . . // found
} else {
    . . . // not found
}
```

If the first-loop exit occurs, it is obvious there is no subscript error because the loop is terminated when i has the value size. If the second loop-exit occurs, i must reference the position of the element equal to the key. As well, no direct knowledge if boolean algebra is required; hence, this form of linear search can be learned very early. Finally, further optimization is possible by removing the extra test after the loop using exit code, as in:

```
for ( i = 0; ; i += 1 ) {
  if ( i == size ) {
          . . . // not found
          break;
      } // exit
  if ( key == list[i] ) {
          . . . // found
          break;
      } // exit
}
```

The main point is that a multi-exit loop can be used to construct algorithms as efficiently as other techniques like short-circuit logical operators. As well, the multi-exit form is more flexible because new code can be inserted between the exits and on exit termination; accomplishing similar additions, when other constructs are used, requires program restructuring, which introduces the potential for errors.

[1] Some programming languages, such as Algol 60 [BBG+63] and Haskell [HF92], support deferred/lazy argument evaluation, which allows the C++ built-in operators && and || to be written as routines. However, deferred/lazy argument evaluation produces a substantially different kind of routine call, and has implementation and efficiency implications. Interestingly, C++ allows the operators && and || to be overloaded with user-defined versions, but these overloaded versions *do not* display the short-circuit property. Instead, they behave like normal routines with eager evaluation of their arguments. This semantic behaviour can easily result in problems if a user is relying on the short-circuit property. For this reason, operators && and || should never be overloaded in C++. Fundamentally, C++ should not allow redefinition of operators && and || because they do not follow the same semantic behaviour as the built-in versions. Interestingly, C++ does not allow overloading of operator ?: because of the lazy evaluation of its arguments.

2.4 Static Multi-Level Exit

A multi-level exit is one in which control leaves multiple levels of *nested* control structures. For static multi-level exit, the exit point is *known* at compile time (i.e., determined statically). That is, the target of the transfer is a lexically scoped point within a routine.

As stated previously, without multi-level exit, flag variables are required. For example, if two nested loops must exit to different levels depending on certain conditions, it must be written as follows using only **if** and **while**:

```
flag1 = true;
for ( ; C1 && flag1; ) {
    flag2 = true;
    for ( ; C2 && flag2; ) {
        . . .
        if ( C3 ) {
            flag1 = flag2 = false;    // exit 2 levels
        } else {
            . . .
            if ( C4 ) {
                flag2 = false;        // exit 1 level
            } else {
                . . .
            } // if
        } // if
    } // for
} // for
```

Notice to terminate both loops, *both* flag variables must be set to false. If an additional nested loop is added between the existing two, all locations that terminate the outer loop must be found and changed to set three flag variables to false. Similarly, removing a nested loop requires removing all occurrences of its associated flag variable; hence, changes of this sort are tedious and error-prone.

Alternatively, a multi-level exit version has no flag variables and does not increase execution cost by testing flag variables; as well, nested loops can be introduced or removed without having to change existing code. The mechanism providing this flexibility is loop labelling. The previous example might be written as follows in C++:

```
for ( ; C1 ; ) {
    for ( ; C2 ; ) {
        . . .
  if ( C3 ) goto A;          // exit 2 levels
        . . .
      if ( C4 ) goto B;      // exit 1 level
        . . .
    }
    B: ;
}
A: ;
```

There are several points to note here. First, because the **break** statement exits only one level, a **goto** statement *must* be used to obtain the desired transfer of control. The **goto** transfers control to the corresponding label, terminating any control structure between the **goto** and the label. *Multi-level exit is the only reasonable use for the* **goto** *in C++, which precludes creating loops and transferring into containing control structures.* Second, labels A: and B: are necessary to denote the transfer points. Notice the simple case of multi-level exit (exiting 1 level) is just a multi-exit loop and could have been done with a **break**. However, using **break** would require changing existing code if a nested loop is added inside the inner loop containing the **break** statement. For readability, an exaggerated indentation scheme is presented, which shows exactly the level that an exit transfers to. Remember, a program is a utilitarian entity; indentation should reflect this and not be just an esthetic consideration.

Other programming languages provide better facilities for multi-level exit, similar to that proposed by Peterson et al. (see Section 2.2, p. 8). For example, both μC++ and Java provide a labelled **break** and **continue** statement. For the labelled **break**, it is possible to specify which control structure is the target for exit, as in:

C++	**μC++/Java**
<pre>for (...) { for (...) { for (...) { ... goto L1; ... ... goto L2; ... ... goto L3; ... // or break } L3: ; } L2: ; } L1: ;</pre>	<pre>L1: for (...) { L2: for (...) { L3: for (...) { ... break L1; ... ... break L2; ... ... break L3; ... // or break } } }</pre>

The innermost loop has three exit points, which cause termination of one or more of the three nested loops, respectively. (Note, a label must be attached to a statement. In the example, the null statement is used by putting a semicolon after the label.) For the labelled **continue**, it is possible to specify which control structure is the target for the next loop iteration, as in:

C++	**μC++/Java**
<pre>for (...) { for (...) { for (...) { ... goto L1; ... ... goto L2; ... ... goto L3; ... // or continue L3: ; } L2: ; } L1: ; }</pre>	<pre>L1: for (...) { L2: for (...) { L3: for (...) { ... continue L1; ... ... continue L2; ... ... continue L3; ... // or continue } } }</pre>

The innermost loop has three restart points, which cause the next loop iteration to begin, respectively. For both **break** and **continue**, the target label must be directly associated with a **for**, **while** or **do** statement; for **break**, the target label can also be associated with a **switch** or compound ({}) statement, e.g.:

```
L1: {
    ... declarations ...
    L2: switch ( ... ) {
        L3: for ( ... ) {
            ... break L1; ... // exit compound statement
            ... break L2; ... // exit switch
            ... break L3; ... // exit loop
        }
        ...
    }
    ...
}
```

While labelled **break/continue** do not require explicit use of the **goto** statement to exit multiple levels of control structure, that is only a cosmetic effect; there is always a **goto** performing the transfer.

The advantage of the labelled **break/continue** is that it allows static multi-level exits without having to use the **goto** statement and ties control flow to the target control structure rather than an arbitrary point in a program. Furthermore, the location of the label at the *beginning* of the target control structure informs the reader that complex control-flow is occurring in the body of the control structure. With **goto**, the label at the end of the control structure fails to convey this important clue early enough to the reader. Finally, using an explicit target for the transfer instead of an implicit target allows new nested loop or **switch** constructs to be added or removed without affecting other constructs.

An alternative mechanism for performing some forms of exit is to use **return** statements in a routine body, as in:

```
int rtn( ... ) {
    ... return expr1; ...
    ... return expr2; ...
}
```

These **return** statements may appear in nested control structures, and hence, cause termination of both the control structures and the routine. But since **return** terminates a routine, using it for all exit situations in a routine is impossible. In general, it is advantageous for a routine to have as few **return** statements as possible so it is easy to determine when the routine terminates. In many cases, a few nested **if** statements can eliminate multiple returns entirely; alternatively, an **if** statement with a return at the beginning of a routine can eliminate unnecessary nesting in the body of the routine. Reducing the number of **return** statements is usually more important for a routine that returns a value, i.e., a function rather than a procedure, as it reduces the number of locations to change when the variable or expression that calculates the returned value is modified. But more important than the number of returns is using well-designed logic, e.g.:

<table>
<tr><th>Poor Logic</th><th>Better Logic</th></tr>
<tr><td><pre>
int rtn(int x, int y) {
    if (x > 3) {
        x += 1;
        if (x > y) {
            x = y;
        } else {
            return x + 3;
        }
        return x + y;
    } else if (x < 3) {
        return 3;
    } else {
        return y;
}
</pre></td><td><pre>
int rtn(int x, int y) {
  if (x < 3) return 3;
  if (x == 3) return y;
    x += 1;
    if (x > y) {
        x = y + y;
    } else {
        x += 3;
    }
    return x;
}
</pre></td></tr>
</table>

As with exits, return points should be highlighted by outdenting or comments to make them easy to locate.

Finally, while multi-exit loops appear often, static multi-level exits appear infrequently. Do not expect to find many static multi-level exits in even large programs. Nevertheless, they are extremely concise and execution-time efficient when needed; use them, but do not abuse them.

2.5 Summary

There is a weak equivalence between the basic control structures and advanced control structures, like multi-exit and static multi-level exit, because simulating advanced control structures with basic ones is both awkward and inefficient in most cases. There is a weak equivalence between multi-exit and static multi-level exit because multi-exit cannot simulate static multi-level exit, but the reverse is true; i.e., multi-exit is a subset (one level exit) of static multi-level exit (multiple levels). Having separate constructs for multi-exit and static multi-level exit is largely a programmer convenience, as static multi-level exit is sufficient. The GOTO statement can be used to simulate both multi-exit and static multi-level exit if these specific control structures do not exist in a programming language. While the GOTO statement can be easily misused, resulting in control flow that is extremely difficult to follow, it can be used for very legitimate purposes, and therefore, should not be maligned or removed from programming languages unless other strongly equivalent constructs are provided. Interestingly, even more advanced forms of exit are possible when the transfer point is determined dynamically; these dynamic multi-level exits are presented next.

2.6 Questions

1. A loop exit in C++ is created with the **if** and **break** statements. Simplify the loop bodies in each of the following code fragments and explain why the simplification is better than the original:

   (a)
   ```
   for ( int i = 0;; i += 1 ) {
       if ( key == list[i] ) {
           break;
       } else {
           cout << list[i] << endl;
       }
   }
   ```

(b)
```
for ( int i = 0;; i += 1 ) {
    if ( key != list[i] ) {
        cout << list[i] << endl;
    } else {
        break;
    }
}
```

2. (a) Explain how the short-circuit *AND* (&&) and *OR* (||) work in a conditional expression.

   (b) Use De Morgan's law to convert this **while** loop into an equivalent **for** loop with an exit using a **break**.

```
while ( Values[i] != HighValue && Values[i] <= Max ) { ... }
```

3. Give two restrictions on static multi-level exit that makes it an acceptable programming language construct, i.e., it cannot be misused like a **goto**.

4. Rewrite the following C++ code fragment using **ONLY** assignment and **while** statements (no **for**, **do**, **if**, **switch**, **break**, **goto** or **throw** statements are allowed, as well, the operators &&, || or ? are not allowed) so that it preserves the exact same execution sequence. New variables may be created to accomplish the transformation.

```
if ( i > 27 ) {
    j = 10;
} else {
    k = 27;
}
for ( ;; ) {
    cin >> d;
    d += 5;
  if ( cin.eof() ) break;
    cout << "j:" << j << "i:" << i << "d:" << d << endl;
}
```

5. Rewrite the following C++ program using **ONLY** assignment and **while** statements (no **for**, **do**, **if**, **switch**, **break**, **goto** or **throw** statements are allowed, as well, the operators &&, || or ? are not allowed) so that it preserves the exact same execution sequence. New variables may be created to accomplish the transformation.

```
const int MAX = 5;

int main() {
    int num = 11;

    for ( int i = 1; i <= MAX; i += 1 ) {
        for ( int j = 1; j <= MAX; j += 1 ) {
            cout << num << "  ";
            num += 1;
        }
        cout << endl;
    }
    return 0;
}
```

6. Rewrite the following C++ program using only assignment, **one while**, and any number of **if/else** statements (no **for**, **do**, **switch**, **break**, **goto** or **throw** statements are allowed, as well, the operators &&, || or ? are not allowed) so that it preserves the exact same execution sequence. New variables may be created to accomplish the transformation.

```
int main() {
    int g, b;
    char ch;
    for ( ;; ) {                                // loop until eof
        for ( g = 0; g < 5; g += 1 ) {          // groups of 5 blocks
            for ( b = 0; b < 4; b += 1 ) {      // blocks of 4 characters
                cin >> ch;                      // read one character
      if ( cin.eof() ) goto fini;               // eof ?
                cout << ch;                     // print character
            } // for
            cout << "  ";                       // block separator
        } // for
        cout << endl;                           // group separator
    } // for
  fini: ;
    if ( g != 0 || b != 0 ) cout << endl;
} // main
```

7. Rewrite the following C++ code fragment so it performs the same control flow but removes the **for** and **goto** statements, and replaces them with **ONLY** expressions, **if/else** and **while** statements; no **for**, **do**, **switch**, **break**, **continue**, **goto**, **throw**, &&, || or ? control structures are allowed.

```
for ( int i = 0; i < 10; i += 1 ) {
    S1;
    for ( int j = 0; j < 10; j += 1 ) {
        S2;
        for ( int k = 0; k < 10; k += 1 ) {
            S3;
          if ( C1 ) goto EXIT1;
            S4;
          if ( C2 ) goto EXIT2;
            S5;
          if ( C3 ) goto EXIT3;
        } // for
        EXIT3: ;
        S6;
    } // for
    EXIT2: ;
    S7;
} // for
EXIT1: ;
```

Each C_i boolean expression can be changed within any of the S_j to eventually cause multi-level exits. You are not allowed to make direct assignments to the loop indexes to force a loop to stop, e.g., setting the loop index to 10 is not allowed. New variables may be created to accomplish the rewrite.

8. Write a linear search that looks up a key in an array of elements, and if the key does not appear in the array it is added to the end of the list; otherwise, if the key does exist the number of times it has been looked up is recorded by incrementing a counter associated with the element in the array. Use the following data structures:

```
struct elem {
    int data;     // data value examined in the search
    int occ;      // number of times data is looked up
};
```

Use a multi-exit loop with exit code in the solution and terminate the program if adding an element results in exceeding the array size.

9. Convert the following program into one that does not use **goto**:

```
int c[2] = { 1, 1 }, turn = 1;

void Dekker( int me, int other ) {
    int i = 0;
  A1:
    c[me] = 0;
  L1:
    if ( c[other] == 0 ) {
        if ( turn == me ) goto L1;
        c[me] = 1;
      B1:
        if ( turn == other ) goto B1;
        goto A1;
    }
    CS();
    turn = other;
    c[me] = 1;
    i += 1;
    if ( i <= 1000 ) goto A1;
}
```

10. The following form of control flow appears occasionally:

```
if C1 then
    S1
    if C2 then
        S2
        if C3 then
            S3
        else
            S4
        endif
    else
        S4
    endif
else
    S4
endif
```

Notice, if any of the conditions are false, the same code is executed (often printing an error message or back-tracking), resulting in code duplication. One way to deal with the code duplication is to put code S4 into a routine and call it. Alternatively, imagine fixing this duplication with a labelled **if** for use in the **else** clause, where the **else** terminates all **if** statements up to the corresponding labelled one, as in:

```
L1: if C1 then
        S1
        if C2 then
            S2
            if C3 then
                S3
    else L1 // terminates 3 if statements
        S4
    endif
```

In this example, all 3 **if** statements transfer to the same **else** clause if the conditional is false. Why is it impossible to implement this extension to the **if** statement in C?

Chapter 3

Exceptions*

Basic and advanced control structures allow for virtually any control flow pattern *within* a routine. However, control flow *among* routines is rigidly controlled by the call/return mechanism. Exceptions extend call/return semantics to reference and transfer in the "reverse" direction to normal routine calls. This chapter examines different forms of "reverse"-direction control-flow to augment routine call/return.

3.1 Exception

Advanced routine-call mechanisms are often referred to as exception handling, but this name is misleading because there is little agreement on what an exception is. Attempts have been made to define exceptions in terms of errors but an error itself is also ill-defined. Instead of struggling to define which phenomenon are or are not exceptional (abnormal), this discussion treats exception handling as a control-flow mechanism, where an exception is a component of an **exception handling mechanism** (EHM) that specifies program behaviour after an exceptional event has been detected. The control flow generated by an EHM is supposed to make certain programming tasks easier, in particular, writing robust programs. Robustness results because exceptions are active rather than passive phenomenon, forcing programs to react immediately when an exceptional event occurs. This dynamic redirection of control flow indirectly forces programmers to think about the consequences of exceptional events when designing and writing programs. While exceptions raise the total cost of program development because they require additional programmer time, there is an enormous benefit in program reliability; in particular, ensuring that errors cannot be ignored unless explicitly done so. Nevertheless, an EHM is not a panacea and only as good as the programmer using it.

The definition presented here for an **exceptional event** is an event that is (usually) known to exist but which is *ancillary* to an algorithm. Because it is ancillary, an exceptional event may be forgotten or ignored without causing a direct failure, e.g., an arithmetic overflow can occur during a computation without generating a program error, but result in erroneous results. In other situations, the exceptional event always occurs but with a low frequency, e.g., encountering end-of-file when reading data. Essentially, a programmer must decide on the level of frequency that moves an event from the algorithmic norm to an exceptional case. Once this decision is made, the mechanism to deal with the exceptional event is best moved out of the normal algorithmic code and handled separately. The mechanism to accomplish this separation is the EHM.

EHMs exist in several programming languages, e.g., ML [Mil78], CLU, Ada, Modula-3 [Nel91], Java and C++. Fundamentally, all EHMs require the following three language constructs:

1. A special kind of label, not bound statically to a particular statement, that can be used to uniquely identify an exception at runtime. Often type names are used for these labels since type names are not normally used at execution time (in contrast to variable names). A type name representing an exceptional event is an **exception type**. C++ allows *any* type to be used as an exception type; other programming languages may restrict which type names can be used as an exception type.

2. A statement to transfer control from a statically or dynamically nested control structure to an exception label in a different statically or dynamically nested control structure. C++ provides the **throw** statement.

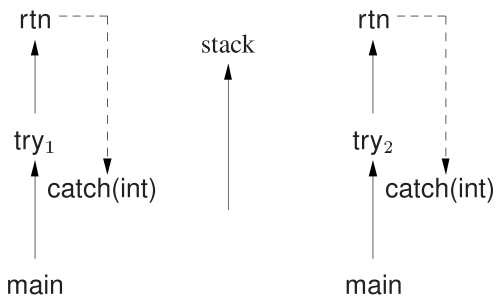
*This chapter is a revision of the work published as "Exception Handling" in [BHM02] ©2002 Elsevier Science. Portions reprinted by permission.

3. A special clause or statement to catch and handle the named exception raised from within a statically or dynamically created block. C++ provides the **try** statement with **catch** clauses.

The key property in most EHMs is that the point where control flow transfers may only be *known* at execution time (i.e., determined dynamically), and hence, an exception is sometimes referred to as a dynamic multi-level exit.

Figure 3.1 shows a simple C++ example of a dynamic multi-level exit. The **throw** statement in routine rtn may raise an exception of type **int** (i.e., the type of 0) depending on the value of a random number. When an exception is raised, control exits routine rtn because the exception is not handled in the scope of rtn. Control transfers down the call stack, to the block for the routine call to rtn. If the exception is not handled in the containing scopes of the call site, the exception continues down the stack to the next routine call. Only when the exception is handled or the bottom of the stack is reached does the search stop.

```
void rtn( int i ) {
    if ( rand() % 3 ) throw 0;
}
int main() {
    int i, j;

    try { // first
        for ( i = 0; i < 10; i += 1 ) {
            rtn( i );
        }
    } catch( int ex ) {
        cout << "first transfer" << endl;
    }
    try { // second
        for ( i = 10; i < 20; i += 1 ) {
            for ( j = 0; j < 10; j += 1 ) {
                rtn( j );
            }
        }
    } catch( int ex ) {
        cout << "second transfer" << endl;
    }
}
```



a) 1st set of calls to rtn   b) 2nd set of calls to rtn

Figure 3.1: Simple Exception

Notice that it is impossible to determine statically where the statement **throw** 0 in rtn transfers control; control may transfer to the first or second **catch** clause (or both or neither) depending on the values generated by the random-number generator. It is this fact that differentiates dynamic and static multi-level exit.

3.2 Traditional EHM

Even with the availability of modern EHMs, the most common programming techniques used to handle an exceptional event are return codes and status flags (although this is slowly changing).

return code: This technique has a routine return a correctness value on completion indicating normal or exceptional execution during the routine. For example, the C output routine, printf, normally returns an integer indicating the number of characters transmitted, or exceptionally returns a negative value if an output or encoding error occurs.

status flag: This technique has a routine set a shared (global) variable with a correctness value on completion indicating normal or exceptional execution during the routine. The value remains in the shared variable as long as it is not overwritten by another routine. For example, many UNIX library routines indicate a specific error by storing a predefined integer value (e.g., 5 means I/O error) into the global variable errno.

fix-up routine: This technique uses a number of local and/or global fix-up routines. A computation may call an appropriate fix-up routine for an exceptional event, and the fix-up routine returns a corrective result so the computation can continue. In general, the fix-up routine must be passed information about the exceptional event to compute a corrective result. This technique allows the corrective action to be determined dynamically, e.g.:

```
int fixup1( int i, int j ) { . . . }
int fixup2( int i, int j ) { . . . }
rtn( a, b, c > 3 ? fixup1 : fixup2 );  // fixup routine determined dynamically
```

The trivial case for this technique is to only pass a fix-up variable rather than a routine. However, it is rare for the fix-up result to be pre-computable, and hence, independent of the specific exceptional event.

The UNIX library routines often use a combination of return code and status flag by returning a negative or zero value to indicate an exceptional condition occurred and using the status variable, errno, to indicate the specific kind of error.

```
if ( printf(. . .) < 0 ) {      // check return code for error
    perror( "printf:" );       // errno describes specific error
    abort();                   // terminate program
}
```

C++ provides a global fix-up routine through the following global routine-pointer:

```
typedef void (*new_handler)();
```

Any fix-up routine assigned to this global pointer is called by operator **new** when a request for storage cannot be satisfied. When called, the fix-up routine must make more storage available and then return, or perform more complex exception handling. Since this particular fix-up routine does not take any parameters, changes must occur using global variables or a functor rather than a routine (see Section 3.16.2, p. 56). A similar kind of global fix-up is provided by UNIX systems for signal handling, where a signal handler deals with ancillary situations that occur infrequently, e.g.:

```
void sigIOHandler( int sig, siginfo_t *sfp, struct ucontext * cxt ) { . . . }
signal( SIGIO, sigIOHandler );
```

The call to signal registers the signal-handler (fix-up) routine sigIOHandler to be called after completion of an I/O operation, called a SIGIO event. When the handler is called, it is passed information appropriate to the particular signal and the execution context at the time of the signal. The signal handler must then perform an appropriate action to deal with the completed I/O operation before execution continues.

Unfortunately, these traditional techniques have significant drawbacks:

- Checking a return code or status flag is optional, so it can be delayed or even omitted, which can cause a cascade of errors later in the program's execution. Hence, traditional techniques are passive rather than active.

- The return code technique often encodes exceptional values among normal returned values, which artificially enlarges the range of valid values independent of the computation. Hence, changing a value representing an exceptional value into a normal return value or vice versa can result in interactions between the normal and exception-handling code, where the two cases should be independent.

- Testing and handling the return code or status flag is often performed locally, i.e., immediately after a routine call, to prevent loss of information and to ensure a problem is dealt with immediately. However, handling a problem immediately is not always the best solution. First, local handling can make a program difficult to read as each routine call results in multiple statements, obscuring the fundamental algorithm. Second, local handling can be inappropriate, especially for library routines, because a local action, e.g., terminating the program, may be inappropriate for a program calling the routine. Hence, local handling is not always desirable since lower-level code does not necessarily know the exact context and/or reasons for its invocation.

- Third, local fix-up routines increases the number of parameters, and hence the cost of each call, and may be passed through multiple levels enlarging parameter lists even when the fix-up routine is not used. Local with fix-up routines can work well, but at the cost of increasing the number of parameters, and hence the cost of each call. But for nonlocal handling, fix-up routines have a problem similar to return codes. For intermediate-level routines in a call chain, local fix-up routines must be passed through these levels enlarging parameter lists even when the fix-up routine may not be applicable. This effect results in unnecessary coupling of routine prototypes between low, intermediate and high level routines, e.g., changing the type of a fix-up routine causes the prototypes of multiple routines to change. Finally, changes may be impossible for legacy routine in a call chain requiring the pass-through of fix-up routines from a high to low level.

- The alternative to local handling is nonlocal handling, meaning the problem is handler further down the call stack, rather than at the original call site. However, to allow nonlocal handling by a calling routine, the return or status values from *all* local calls within a routine must be folded into the single return and/or status values for a routine, requiring no overlap among values and possibly significantly enlarging the set of returned values. As a result, it is difficult to generate unique return/status values, and the routine that finally handles the exception must be able to decode the value to determine the situation that generated it.

- A status flag can be overwritten before the previous condition is checked or handled. This problem is exacerbated in a concurrent environment because of sharing issues.

- Global fix-up routines, often implemented using a global routine-pointer, have identical problems with status flags. In particular, because the pointer values can change in many places to perform different fix-up actions, programmers must follow a strict convention of saving and restoring the previous routine value to ensure no local computation changes the environment for a previous one. Relying on programmers to follow conventions is always extremely error prone. Also, shared global variables do not work in concurrent contexts.

3.3 EHM Objectives

The failure of return codes and status flags as an informal (ad hoc) EHM argues strongly for a formal EHM supported by the programming language, which must:

1. alleviate multiple testing for the occurrence of rare conditions throughout the program, and at the location where the test must occur, be able to change control flow without requiring additional testing and transfers,

2. provide a mechanism to prevent an incomplete operation from continuing, and

3. be extensible to allow adding, changing and removing exceptions.

The first objective targets readability and programmability by eliminating multiple checking of return codes and flags, and removing the need to pass fix-up routines or have complex control-logic within normal code to deal with exceptional cases. The second objective provides a transfer from the exceptional event that disallows returning, directing control flow away from an operation where local information is possibly corrupted, i.e., the operation is **nonresumable**. The last objective targets extensibility, allowing change in the EHM, and these changes should have minimal effects on existing programs using them.

As well, a good EHM should strive to be orthogonal to other language features, i.e., the EHM features should be able to be used in any reasonable context without unnecessary restrictions. Nevertheless, specific implementation and optimization techniques for some language constructs can impose restrictions on constructs, particularly the EHM.

3.4 Execution Environment

The structure of the execution environment has a significant effect on an EHM, i.e., an object-oriented concurrent environment requires a more complex EHM than a non-object-oriented sequential environment. For example, objects declared in a block may have destructors, and these destructors must be executed no matter how the block ends, either by normal or exceptional block termination. As well, a control structure may have a `finally` clause that must always be executed no matter how it ends, either by normal or exceptional block termination. Hence, the EHM becomes more complicated since terminating a block containing local objects with destructors (and recursively if these objects contain local objects) or control structures with `finally` clauses must be executed. Another example is that sequential execution has only one stack; therefore, it is simple to define consistent and well-formed semantics for exceptional control flow. However, a complex execution-environment involves advanced objects (e.g., continuation, coroutine, task), each with its own separate execution stack (see Section 4.6.2, p. 82). Given multiple stacks, the EHM semantics for this environment must be more sophisticated, resulting in more complexity. For example, when an exception is raised, the current stack is traversed down to its base searching for an appropriate handler. If no handler is found, it is possible to continue propagating the exception from the current stack to another object's stack. The choice for selecting the point of continuation depends on the particular EHM strategy. Hence, the complexity and design of the execution environment significantly affects the complexity and design of its EHM.

3.5 EHM Overview

A programming entity with a stack is generically called an **execution entity**, because it has the potential to perform all basic and advanced execution independently of other execution entities. An **exception** is an instance of an exception type, and is generated by executing a language or system operation indicating an ancillary (exceptional) situation for a particular execution entity. **Raising** (**throwing**) is the special operation that creates an exception, denoting the ancillary condition the programmer cannot or does not want to handle via conventional control flow. Some exceptions cannot be raised by the programmer, e.g., only the runtime system may raise predefined exceptions such as hardware exceptions. Usually an exception may be raised at any point during a program's execution. The execution entity raising the exception is the **source execution**. The execution entity that changes control flow due to a raised exception is the **faulting execution**. A **local exception** is when an exception is raised and handled by the same execution entity, i.e, the source and faulting executions are the same. A **nonlocal exception** is when the source and faulting executions are different. A **concurrent exception** is a nonlocal exception where the source and faulting executions are executing concurrently, and it is the most advanced form of nonlocal exception. Unlike a local/nonlocal exception, raising a concurrent exception does not lead to an immediate transfer of control in the faulting execution. That is, a concurrent exception may be delayed, until the faulting execution begins propagation at some appropriate time.

Propagation directs control flow from the source execution to a handler that matches the exception in the faulting execution. A **propagation mechanism** is the set of rules used to locate and match an exception with a handler. The most common propagation mechanism gives precedence to handlers higher on the call stack, so an exception is handled by a handler closest to the block where propagation of the exception starts. Usually, operations higher on the stack are more specific while those lower on the call stack are more general. Handling an exception is often easier and more precise in a specific context than in a general context. This propagation approach also minimizes the amount of stack searching and unwinding when raising an exception, so it is efficient to implement.

A **handler** has two components:

1. A value for comparison with the raised exception to determine if the handler is qualified to deal with it.
2. An inline (nested) block of code responsible for dealing with matching a raised exception.

A chosen handler is said to have **caught** the exception when propagation transfers there; a handler may **match** with one or more exceptions. It is possible that another exception is raised or the current exception is reraised while executing the handler. A handler is said to have **handled** an exception only if the handler completes rather than raising another exception. Unlike returning from a routine, there may be multiple return mechanisms for a handler (see Section 3.8, p. 45). After an exception is handled, the blocks on the stack from the start of propagation to the handler may be removed from the stack, called **stack unwinding**.

In summary, an EHM = *exception type* + *raise* + *propagation* + *handler*, where *exception type* represents a kind of exceptional event, *raise* instantiates an exception from the exception type in the source execution and finds the faulting execution, *propagation* finds an appropriate handler for the exception in the faulting execution, and that *handler* deals with the exceptional event or raises the same or another exception.

3.5.1 Handler Association

Goodenough's seminal paper on exception handling suggests a handler, enclosed in brackets [...], can be associated with programming units as small as a subexpression and as large as a routine [Goo75, pp. 686–7], e.g.:

```
D = (A + B)[OVERFLOW: /* handle overflow */ ] + C;  /* subexpression */

DO WHILE ( ... );                                   /* statement */
    ... /* read and process */
END; [ENDFILE: /* handle end-of-file */ ]

P : PROCEDURE(...);                                 /* routine */
BEGIN;
    ...
END; [ERROR: /* handle error */ ]
```

Between these extremes is associating a handler with a language's notion of a block, i.e., the facility that combines multiple statements into a single unit, as in Ada, Modula-3, C++ and Java. While the granularity of a block is coarser than an expression, the need for fine-grained handling is usually rare. As well, having handlers, which may contain arbitrarily complex code, in the middle of an expression can be difficult to read, e.g.:

```
G = ((A + B)[OVERFLOW: ... ] - C)[UNDERFLOW: ... ] * ((D / E)[ZERODIVIDE: ... ] + F) ...;
```

In this situation, it is usually better to subdivide the expression into multiple blocks with necessary handlers. Finally, handlers in expressions or for routines may need a mechanism to return results to allow execution to continue, which requires additional constructs [Goo75, p. 690], e.g.:

```
D = (A + (B + C))[OVERFLOW: EXIT(-1);];
```

assigns -1 to D if an overflow occurs during the calculation of the expression. Because of these drawbacks, this discussion assumes handlers are only associated with language blocks, as in C++, which can simulate the other forms.

In addition, a single handler can handle several different kinds of exceptions (see Section 3.7.1, p. 37) and multiple handlers can be bound to one block. Syntactically, the set of handlers bound to a particular block is the **handler clause**, and a block with handlers becomes a **guarded block**, e.g.:

```
try {
    // guarded block
} catch( E1 ) {         // may handle multiple kinds of exceptions
    // handler1
} catch( E2 ) {         // multiple handlers for guarded block
    // handler2
}
```

A block with no handler clause is an **unguarded block**. The propagation mechanism determines the order the handler clauses bound to a guarded block are searched.

The scope of a guarded block and its handler clause varies among languages with an EHM. For example, in Ada, the handler clause is nested inside a guarded block, and hence, can access variables declared in it, while a C++ handler executes in a scope outside its guarded block making variables in the guarded block inaccessible, e.g.:

Ada	C++
VAR x : INTEGER; -- outer	**int** x; *// outer*
BEGIN	**try** {
VAR x : INTEGER; -- inner	**int** x; *// inner*
EXCEPTION WHEN Others =>	} **catch**(...) {
x := 0; -- inner x	x = 0; *// outer x*
END;	}

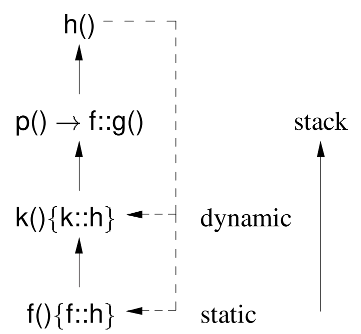
Notice that the variable x in the handler clause is different between the two languages. By moving the handler and possibly adding another nested block, the same semantics can be accomplished in either language, as long as a handler can be associated with any nested block.

3.5.2 Control-Flow Taxonomy

It is possible to categorize normal and exceptional control-flow via two general properties:

static/dynamic call: The routine/exception-type name at the call/raise is looked up statically (compile-time) or dynamically (runtime) [Cos03]. For example, in (pretend C++ supports nested routines):

```
void f() {
    void h() {}                  // f::h
    void g() {                   // f::g
        h();                     // call h ?
    }
    void k( void (*p)() ) {      // routine parameter
        void h() {}              // k::h
        p();                     // call supplied routine
    }
    k(g);                        // pass g to be called in k
}
```

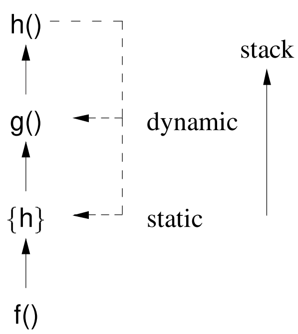


the difference between static and dynamic call occurs after g is indirectly called from k through parameter p. When g is invoked, it calls routine h; but which h? For static call, during compilation of g the name h is looked up in the lexical context, which is the h defined in f because the h in k is inaccessible by the lexical scope rules. For dynamic call, during execution of g, the name h is looked up in the dynamic context, which is the h defined in k because the call occurs inside of k through parameter p. Essentially, dynamic call searches for a routine definition with the same name as the call name closest on the stack. While dynamic call may seem unusual, it provides a form of polymorphism, allowing a routine call to vary dynamically depending on the caller, e.g., to generalize pre-compiled library routines.[1]

Interestingly, this polymorphism is subtly different from the polymorphism provided by virtual members of an object accessed by an object reference. When a virtual member is called through an object reference, it is not known statically which member is called, as it depends on the particular object referenced. However, (usually) no search is required to look up the member in the object; the member's location is known at compile time (e.g., virtual-routine vector). Hence, this case is only a partial dynamic call.[2]

static/dynamic return: After a routine/handler completes, it returns to its static (lexical) or dynamic context. For example, in (pretend C++ supports nested routines):

```
void f() {
    {                   // compound statement
        void h() {}
        void g() {
            h();        // dynamic return point
        }
        g();
    }                   // static return point
    ...
}
```



the difference between static and dynamic return occurs after h is called from g. For static return, during compilation of h the lexical context for h is remembered as the return point, which means when h returns from the call in g, control transfers to the statement after the end of the block defining h, i.e., after the compound statement. Notice, no matter where h is called (directly or indirectly), the block containing its lexical context must exist because h can only be referenced from within the block in which it is defined. Also, during the transfer to the end of the compound statement, the stack is unwound, removing the routine activations for h and then g. Hence, static return is extremely powerful because it allows terminating multiple routine activations without programmatically returning from each routine. For dynamic return, during execution of h the dynamic routine making the call is remembered as the return point, which means when h returns from the call in g, control transfers to the statement after the call. In this case, only the top routine activation on the stack is removed, and control continues after the call in the activation that is now at the top of the stack, which is the behaviour for normal routine-call in most languages.

These two control-flow properties can be combined in four ways, resulting in the following language constructs:

	call/raise	
return/handled	static	dynamic
static	1. sequel	3. termination
dynamic	2. routine	4. resumption

Table 3.1: Static/Dynamic Call/Return Constructs

Case 2 is already familiar as it is a normal routine, with static name lookup at the call and a dynamic return. The other three cases contain all the extant forms of exception handling, and each is discussed in detail. An EHM may provide all or only a subset of these three possible cases.

[1]The original Lisp [Ste84] programming language uses dynamic call, but newer versions of Lisp, e.g., Scheme [RC86] provide both static and dynamic call.

[2]Some object-oriented programming languages, e.g., Smalltalk [GR83] and Self [ABC+93], actually do a full dynamic call involving a search for the member's location in the object.

3.6 Exception Models

It is common to classify an EHM based on the call/raise property that occurs during propagation, i.e., the two columns in Table 3.1, excluding case 2 because it is normal routine call. The left column is called static propagation and the right dynamic propagation. However, because static propagation is rarely used, dynamic propagation is often known as propagation. Both propagation models are discussed and analysed. As well, it is common to classify an EHM based on the return/handled property that occurs after propagation, i.e., the two rows in Table 3.1. The top row is called terminate semantics as it is impossible to return to the point of the call/raise because the routine activations on the stack are removed during propagation. The bottom row is called resume semantics as it is possible to return to the point of the call/raise because the routine activations on the stack are not removed during propagation.

3.6.1 Static Propagation

Static propagation (case 1 in Table 3.1) is based on Tennent's sequel construct [Ten77, p. 108]. A **sequel** is a routine, including parameters but no return value, where the routine (handler) name is looked up lexically at the call site but control returns to the end of the block in which the sequel is declared (static return) rather than after the sequel call.[3] In this case, return is along the lexical structure of the program versus the routine activations on the call stack. Note, a sequel requires routine nesting within the program's lexical structure (which C++ does not have).

The original purpose of the sequel was not exception handling; it was to **modularize** static exits. Modularization is one of the fundamental tenets of software engineering allowing any contiguous code fragment to be replaced by a call to a routine containing that code, passing any appropriate local information. Conversely, any routine call can be expanded inline, rewriting call arguments into parameter variables. Interestingly, modularization of code containing a static exit is impossible without a sequel-like construct. In the left example of Figure 3.2, it is impossible to modularize any code containing a **break** statement. The reason is that the transfer points only have meaning in the routine where a **break** is located. For example, if **break** A is placed in another routine, it can no longer be a static exit because that routine can be called from other locations where a different label A is defined or there is no label A. In the right example of Figure 3.2, the sequels S1 and S2 are used to allow modularization of two code fragments containing static exits. When sequel S2 is called, control transfers to the end of the loop labelled B, terminating two nested loops, because the body of loop B contains the definition of S2. When sequel S1 is called from within M1, control transfers to the end of the loop labelled A, terminating the call to M1 and the three nested loops because the body of loop A contains the definition of S1.

```
A: for ( ;; ) {

    B: for ( ;; ) {


        C: for ( ;; ) {
            . . .
            if ( . . . ) { . . . break A; }
            . . .
            if ( . . . ) { . . . break B; }
            . . .
            if ( . . . ) { . . . break C; }
            . . .
        }
    }
}
```

```
A: for ( ;; ) {
    sequel S1( . . . ) { . . . }
    B: for ( ;; ) {
        void M1( . . . ) { . . . if ( . . . ) S1( . . . ); . . . }
        sequel S2( . . . ) { . . . }
        C: for ( ;; ) {

            M1( . . . );            // modularize

            if ( . . . ) S2( . . . );    // modularize
            . . .
            if ( . . . ) break C;
            . . .
        }
    } // S2 static return
} // S1 static return
```

Figure 3.2: Static-Exit Modularization

With respect to exception handling, static return provides a crucial feature that is unavailable with dynamic return: the ability to terminate multiple routine activations in a single transfer. The left example in Figure 3.3 shows an

[3] An alternate design for the sequel is to have two forms of return: normal and exceptional. A normal **return** is dynamic, like a routine; an exceptional return, using a raise, is static.

Monolithic Compilation	Separate Compilation
<pre>{ // compound statement sequel overflow(...) { ... } // static exit class stack { void push(int i) { ... if (...) overflow(...); ... } ... }; stack s; // create stack for (...) { ... s.push(3); ... // possible overflow ? } } // overflow transfers here</pre>	<pre>class stack { // library code int (*overflow)(int); stack(sequel of(...)) : overflow(of) {...} void push(int i) { ... if (...) overflow(...); ... } };</pre><hr><pre>{ // user code sequel my_overflow(...) {...} stack s(my_overflow); ... s.push(3); ... // possible overflow ? } // my_overflow transfers here</pre>

Figure 3.3: Sequel compilation structure

example of how a sequel can be used to provide exception handling [Knu84, Knu87]. The sequel overflow is defined along with the class stack. Inside stack::push, the sequel overflow is called if the current push operation cannot be performed because of insufficient storage for the stack (imagine an array implementation of the stack with a finite number of elements). After stack s is created, the **for** loop begins to push elements onto the stack, which could result in a stack overflow. If push calls overflow, control returns to the end of the compound statement because it contains the definition of sequel overflow. Hence, the advantage of the sequel is the handler is statically known (like static multi-level exit), and can be as efficient as a direct (**goto**) transfer. If a handler behaves like a sequel, the handling action for each raise can be determined at compile time (statically).

The disadvantage with static return for exception handling is that it only works for monolithic code, i.e., code not subdivided for separate compilation. For example, library and user code have disjoint static contexts, e.g., if stack is separately compiled in a library, the sequel call in push no longer knows the static blocks containing calls to push. To overcome this problem, a sequel can be made a parameter of stack (right example in Figure 3.3), and the stack creator provides a fix-up routine as an argument. Now when overflow is called inside stack::push, control transfers to the lexical location of the sequel passed as an argument, in this case to the end of the block containing my_overflow. However, when a sequel is passed as an argument, the handler is no longer known statically at the raise because the parameter changes with each argument value. Furthermore, adding the sequel to a routine's type prototype has the same problems as fix-up routines (see Section 3.2, p. 22) and can inhibit reuse (see Section 3.7.4, p. 40).

3.6.2 Dynamic Propagation

Dynamic propagation differs from static propagation because the handler name at the raise is looked up during execution (unlike a routine). The return semantics for dynamic propagation can be either static (like a sequel) or dynamic (like a routine), giving termination (case 3) and resumption (case 4). Termination semantics (static return) provides the important capability of terminating multiple routine activations if an exception is nonresumable; resumption semantics (dynamic return) provides the important capability of going back (returning) to a raise if an exception is resumable after a correction. The advantage of dynamic propagation is that it offers an additional form of polymorphism to generalize exception handling (see Section 3.5.2, p. 26) by searching for handlers installed in guarded blocks on the stack. As well, dynamic propagation works for separate compilation because the lexical context is not required.

The disadvantages of dynamic propagation are:

Visibility Dynamic propagation can propagate an exception through a scope where it is invisible and then back into a scope where it is visible. For example, if routine A calls B and B calls C, C can raise an exception that is unknown to B but caught by A. It has been suggested this behaviour is undesirable because a routine, i.e., B, is indirectly propagating an exception it does not know [MMG96]. Some language designers believe an exception should never be propagated into a scope where it is unknown, or if allowed, the exception should lose its identity and be converted into a general failure exception. However, there are significant reuse restrictions resulting from preventing such propagation (see Section 3.7.4, p. 40), or if the exception is converted to a general exception, the loss of specific information may preclude proper handling of the exception at a lower-level of abstraction.

Dynamic Handler Selection With dynamic propagation, the handler chosen for an exception cannot (usually) be determined statically. Hence, a programmer seldom knows statically which handler may be selected, making the program more difficult to trace and the EHM harder to use [Mac77, Knu84, YB85, MMG96]. However, when raising an exception it is rare to know which specific action is to be taken; otherwise, it is unnecessary to define the handler in a separate place, i.e., bound to a guarded block lower on the call stack. Therefore, the uncertainty of a handling action when an exception is raised is not introduced by a specific EHM but by the nature of the problem and its solution. For example, a library normally defines exception types and raises them without providing any handlers; the library client provides the specific handlers for the exception in the application. Similarly, the return code technique does not allow the library writer to know the action taken by a client. When an EHM facility is used correctly, the control flow of propagation, the handler selection, and its execution should be understandable.

Recursive Resuming With termination, the handlers in previous scopes disappear as the stack is unwound; with resumption, the stack is not unwound leaving in place handlers in previous scopes. The presence of resumption handlers in previous scopes can cause a situation called **recursive resuming**, if the handler or routines called by the handler raise the same exception it just caught, e.g.:

```
try {
    ... resume R; ...      // T{H(R)}    => try block T has handler H for exception-type R
} catch( R ) resume R;     // H(R)       => handler for R
```

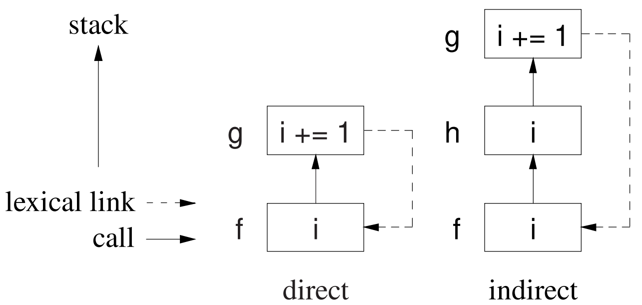
The **try** block resumes R. Handler H is invoked by the resume, and the blocks on the call stack are:

$\ldots \rightarrow$ T{H(R)} $\rightarrow$ H(R) $\quad\leftarrow$ stack top

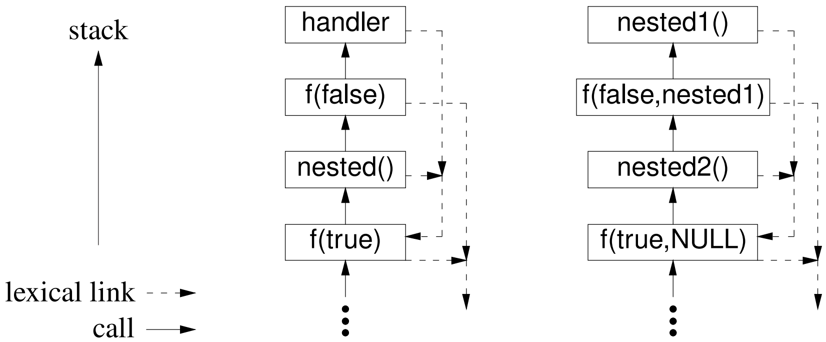
Then H resumes an exception of type R again, which finds the handler just above it at T{H(R)} and invokes handler H(R) again, and this continues until the runtime stack overflows. Recursive resuming is similar to infinite recursion, and is difficult to discover both at compile time and at runtime because of the dynamic choice of a handler. Concurrent exceptions compound the difficulty because it can cause recursive resuming where it is impossible otherwise because the concurrent exception can be delivered at any time. MacLaren briefly discusses the recursive resuming problem in the context of PL/I [Mac77, p. 101], and the problem exists in Exceptional C [Geh92] and μSystem [BMZ92]. Mesa made an attempt to solve this problem but its solution is often criticized as non-intuitive. Both the Mesa and VMS solutions to the recursive-resuming problem are presented in Section 3.12, p. 51.

Lexical Context When a handler executes, its static context must be established to access variables and routines in that context. In the Ada/C++ example at the end of Section 3.5.1, p. 25, each handler must have access to the proper instance of variable x in the specific lexical scope. If a raise occurs directly within the guarded block, it is straightforward to provide access to the proper variables because the guarded block is physically next to the handler block on the stack. However, when the raise point is separated from the guarded block by multiple blocks (stack frames), it is generally necessary for the handler to access variables in its enclosing scope using a special pointer called a **lexical (static) link**. Lexical links are a standard technique in the implementation of nested routines to give a routine access to variables from the lexical context of its definition [FL91, Section 9.5], e.g.:

```
void f() {
    int i = 10;
    void g() { i += 1; }
    void h( int i ) {
        if ( i == 0 ) g(); // indirect call
        else h( i - 1 );
    }
    g(); // direct call
    h( 0 );
}
```



In the example, the nested routine g increments the variable f::i in its lexical scope. To access f::i, g uses a lexical-link pointer to dynamically locate the lexical context of its definition, i.e., f's stack frame, because there can be




```
void f( bool x ) {
    if ( x ) {
        try {
            f( ! x );
        } catch( E ) ...;
    }
    else resume E;
}
```

```
void f( bool x, void (*r)() ) {
    void nested1() {...}
    void nested2() {
        f( ! x, nested1 );
    }
    if ( x ) nested2();
    else r();
}
```

Figure 3.4: Lexical Contexts

any number of intervening routine activations between frames g and f. Specifically, the direct call to g results in its stack frame being next to the stack frame for f. An indirect call to g occurs from within h. When h is called with 0, there is one intervening h stack frames between f and g. By increasing the argument to h it is possible to create any number of intervening h stack frames between g and f. Hence, the need for the lexical pointer from g to f to bypass intervening stack frames. Interestingly, a resumption handler behaves almost identically to a nested routine, except it is invoked as the result of a raise rather than a call; hence, the presence of resumption usually also requires the presence of lexical links. Lexical links are usually unnecessary in the case of termination as the intermediate stack frames are discarded before execution of the handler. In this case, the handler executes in the same lexical context in which it is defined. Hence, implementing resumption can have an inherent additional cost (lexical links); furthermore, languages without nested routines (e.g., C/C++) would need to add lexical links to support resumption, which can increase the cost of all routine calls. The need for lexical links is one reason why resumption exception-handling is not implemented as widely as termination.

In fact, the lexical context for resumption handlers has been cited as a source of confusion and complexity [Geh92, p. 833] [Str94, pp. 391-392]. Confusion results from unexpected values being accessed due to differences between static and dynamic contexts, and complexity from the need for lexical links. Interestingly, both these issues are also related to nested routines. The confusion between static and dynamic contexts is illustrated in Figure 3.4, which shows a resumption and nesting example that generate identical dynamic situations. The resumption handler in the left example and the call to nested1 in the right example both have a lexical context of f(**true**...), so both routines reference x with a value of **true** even though there is an instance of f(**false**...) (i.e., x is **false**) directly above them on the stack. However, this case is rare. Complexity for supporting resumption is not an issue for languages with nested routines as lexical links are already required to support nesting. For languages without nested routines, it is still possible to provide lexical links for resumption handlers, but at a potential significant cost. If lexical links are not provided, resumption handlers are still possible but must be specified separately from the guarded block, affecting readability and ease of use (see Section 3.16.2, p. 56).

The two forms of dynamic propagation, termination and resumption, are discussed in detail.

3.6.2.1 Termination

There are three termination models: nonlocal transfer, terminate, and retry [YB85, p. 218], all using dynamic call and static return.

```
label L;
void f() { goto L; }
void g() { f(); }
void h() {
      L = L1;
      f();
  L1:
      L = L2;
      g();
  L2:
}
```

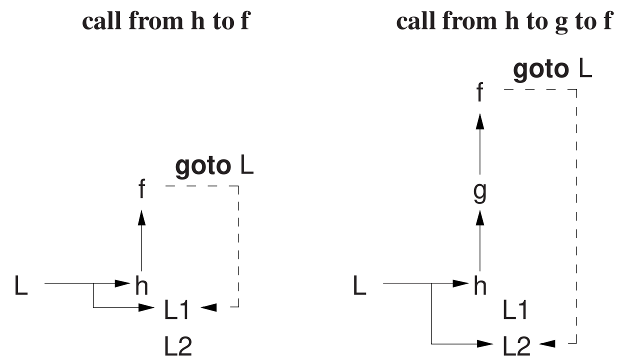


Figure 3.5: Dynamically-Scoped Transfer-Points

Nonlocal Transfer Local transfer exists in all programming languages, implicitly or explicitly (see Section 2.2, p. 8), where the transfer point is known at compile time. Dynamically-scoped transfer-points are also possible, called **nonlocal transfer**. In Figure 3.5, the label variable L contains both a pointer to a routine activation on the stack and a transfer point within the routine; therefore, a label value is not statically/lexically determined. The nonlocal transfer, **goto** L, in f becomes a two-step operation:

1. direct control flow to the specified routine activation on the stack;
2. and then go to the transfer point (label) within the routine.

A consequence of the transfer is that blocks activated between the **goto** and the label value are terminated because of stack unwinding. In the example, the first nonlocal transfer from f transfers to the label L1 in h's routine activation, terminating f's activation. The second nonlocal transfer from f transfers to the static label L2 in the activation record for h, terminating the activation records for f and g. PL/I [IBM81b] is one of a small number of languages (Beta [MMPN93], C [KR88]) supporting nonlocal transfer among dynamic blocks through the use of label variables. PL/I and Beta provide nonlocal transfer via a basic language control-flow mechanism, while C uses a library approach.

An EHM can be constructed using a nonlocal transfer mechanism by labelling code to form handlers and terminating operations using nonlocal transfers to labels in prior blocks. For example, in Figure 3.6, the PL/I program (left example) assigns to the label variable E1 a label in the scope of procedure TEST and then calls procedure F. Procedure F executes a transfer (GOTO) to the label variable, transferring out of F, through any number of additional scope levels, to the label L1 in TEST. Inside the handler, the same action is performed but the transfer point is changed to L2. Similarly, the C program (right example) uses the library routine setjmp to store the current execution context in variable E1, *which is within the scope of the call to* setjmp, setjmp returns a zero value, and a call is made to routine f. Routine f executes a transfer using the library routine longjmp to the execution-context variable, transferring out of f, through any number of additional scope levels, back within the saved scope of setjmp, which now returns a nonzero value. However, having setjmp return once for the explicit setup call and implicitly again after the raise via longjmp, makes this library approach for nonlocal transfer both obscure and confusing. Inside the handler, the same action is performed but the transfer point is changed to the second call of setjmp. The key point is that the transfer point for the GOTO or longjmp is unknown statically; it is determined by the dynamic value of the label or execution-context variable.

Unfortunately, nonlocal transfer is too general, allowing branching to almost anywhere, i.e., the structured programming problem. This lack of discipline can make programs less maintainable and error-prone [Mac77, p. 102]. More importantly, an EHM is essential for sound and efficient code generation by a compiler (as for concurrency [Buh95]). If a compiler is unaware of exception handling, e.g., setjmp/longjmp in C, it may perform code optimizations that *invalidate* the program, needing cryptic concepts like the **volatile** declaration qualifier. Because of these problems, nonlocal transfer is unacceptable as an EHM. However, nonlocal transfer is essential in an EHM, otherwise is it impossible to achieve the first two EHM objectives in Section 3.3, p. 24, i.e., alleviating explicit testing and preventing return to a nonresumable operation. Therefore, a restricted form of nonlocal transfer is necessary.

Terminate The terminate approach *restricts* nonlocal transfer similar to the way labelled break restricts local transfer: no backward branch and no branching into a block (see Section 2.2, p. 8). After dynamic propagation to locate a

PL/I	C
TEST: PROCEDURE OPTIONS(MAIN);	
DCL E1 LABEL;	jmp_buf E1;
F: PROCEDURE;	**void** f(**void**) {
GOTO E1;	longjmp(E1, 1);
END;	}
	int main() {
E1 = L1;	**if** (setjmp(E1) == 0) {
CALL F; RETURN;	f();
L1:	} **else** {
/ HANDLER 1 */*	*/* handler 1 */*
E1 = L2;	**if** (setjmp(E1) == 0) {
CALL F; RETURN;	f();
L2:	} **else** {
/ HANDLER 2 */*	*/* handler 2 */*
END;	}
	}
	}

Figure 3.6: Nonlocal transfer

handler, if the handler completes, control flow continues as if the incomplete operation in the guarded block terminated without encountering the exception, i.e., static return like a sequel. Hence, the handler acts as an alternative operation for its guarded block. This model is the most popular form of restricted nonlocal transfer, appearing in Ada, C++, ML [MT91], Modula-3 and Java [GJSB00]. Terminate is often likened to a *reverse* routine call, particularly when argument/parameters are available (see Section 3.7.3, p. 39). Essentially, the raise acts like a call and the handler acts like a routine, but control flows down the stack rather than up. Some EHMs divide terminate into one level and multiple levels [LS79, p. 547] [YB85, p. 218]. That is, control transfers from the raise to the immediate caller (one level) or from the raise to any nested caller (multiple levels). However, this distinction is artificial and largely stems from a desire to support exception lists (see Section 3.7.4, p. 40).

For example, in Figure 3.7, the Ada program (left example) defines an exception type E1 and procedure f in the scope of procedure main and then calls f. Procedure f executes a raise of the exception, transferring out of f, through any number of blocks on the stack, to the handler at the end of main. The C++ program (right example) does not declare an exception type; instead, an object type is used as the label, and the type is inferred from an object specified at the raise point; in this case, **throw** 0 implies an exception type of **int**. Routine f executes a raise (**throw**) of the exception, transferring out of f, through any number of blocks on the stack, to the handler at the end of the **try** statement in main. Notice that termination achieves the first two EHM objectives in Section 3.3, p. 24, without the drawbacks of nonlocal transfer.

Ada	C++
procedure main is	
E1 : exception;	
procedure f is begin	**void** f() {
raise E1;	**throw** 0;
end f;	}
begin	**int** main() {
	try {
f;	f();
exception	} **catch**(**int**) {
when E1 => -- handler	*// handler*
end main;	}
	}

Figure 3.7: Terminate

Interestingly, because C++ allows any type to be used as an exception type, it seems to provide additional generality, i.e., there is no special exception type in the language. However, in practice, this generality is almost never used. First, using a built-in type like **int** as an exception type is dangerous because the type has no inherent meaning for any exceptional event. That is, one library routine can raise **int** to mean one thing and another routine can raise **int** to mean another; a handler catching **int** may have no idea about the meaning of the exception. To prevent this ambiguity, programmers create specific types describing the exception, e.g., overflow, underflow, etc. Second, these specific exception types can very rarely be used in normal computations, so the sole purpose of these types is for raising unambiguous exceptions. In essence, C++ programmers ignore the generality available in the language and follow a convention of creating explicit exceptions types. Therefore, having a specific exception type in a programming language is not a restriction, and it provides additional documentation, discrimination among conventional and exception types, and provides the compiler with exact knowledge about type usage rather than having to infer it from the program.

Modula-3 defines dynamic return indirectly using static return (terminate): RETURN is defined as a raise of exception-type return−exception and each call is structured as:

```
TRY
    rtn;        (* routine call implicitly raising return-exception *)
EXCEPT
    return−exception (v) => (* optional variable for function result *)
END
```

The reason given for this transformation is to specify interactions between RETURN and exception handling statements, especially the FINALLY clause. Specifically, the conversion turns a simple dynamic return into a complex dynamic call (throw), and the actual return from the routine occurs indirectly because of the static return from the handler after the routine call in the guarded block. However, this transformation is impractical to implement as the dynamic call from the throw to start the return is more costly than a normal return. Furthermore, it confuses exception handling and routine call control-flow for catch-any (see Section 3.7.2, p. 38), e.g.:

```
TRY
    rtn;
    (* control never reaches here *)
EXCEPT
ELSE    (* catch any exception raised in the guarded block *)
    (* normal and exceptional control always goes here *)
END
```

While this transformation was used by the designers of Modula-3 to define the interaction of the RETURN statement with the EHM, there is little practical benefit in expressing dynamic return in this way.

Retry The retry model combines the terminate model with special handler semantics, i.e., restart the failed operation, creating an implicit loop in the control flow. There must be a clear beginning for the operation to be restarted. The beginning of the guarded block is usually the restart point and there is hardly any other sensible choice. The left example of Figure 3.8 shows a **retry** handler by extending the C++ exception mechanism; the example reads numbers from multiple files and sums the values. For this example, pretend that C++ raises an exception-type eof for end-of-file, and the **retry** handler completes by jumping to the start of the **try** block. The handler is supposed to remove the exceptional event in the **retry** handler so the operation can continue. In the example, it opens the next file so reading can continue until all files are read. Mesa [MMS79], Exceptional C [Geh92], and Eiffel [Mey92] provide retry semantics through a **retry** statement only available in the handler clause.

As mentioned, establishing the operation restart point is essential; reversing lines 5 and 7 in the figure generates a subtle error with respect to the exceptional but not normal execution, i.e., the sum counter is incorrectly reset on retry. This error can be difficult to discover because control flow involving propagation may occur infrequently. In addition, when multiple handlers exist in the handler clause, these handlers must use the same restart point, which may make retrying more difficult to use in some cases.

While the retry model does provide some additional facilities, it can be easily simulated with a loop and the termination model (see right example of Figure 3.8) [Geh92, p. 834]. The transformation is straightforward by nesting the guarded block in a loop with a loop-exit in the handler. In general, rewriting is superior to using retry so that all looping is the result of language looping constructs, not hidden in the EHM. Because of the above problems and that

Retry

```
char readfiles( char *files[], int N ) {
    int i = 0, sum, value;
    ifstream infile;
    infile.open( files[i] );
    sum = 0;

    try {
        infile >> value;
        sum += value; ...
    } retry( eof ) {
        i += 1;
      if ( i == N ) goto EOF;
        infile.close();
        infile.open( files[i] );
    }
  EOF: ;
}
```

Simulation

```
char readfiles( char *files[], int N ) {
    int i = 0, sum, value;
    ifstream infile;
    infile.open( files[i] );
    sum = 0;
    while ( true ) {
        try {
            infile >> value;
            sum += value; ...
        } catch( eof ) {
            i += 1;
          if ( i == N ) break;
            infile.close();
            infile.open( files[i] );
        }
    }
}
```

Figure 3.8: Retry

retry can be simulated easily with termination and looping, retry is not really needed, and hence, seldom appears in an EHM.

3.6.2.2 Resumption

All the termination models are based on nonlocal transfer to ensure control *does not* return to a nonresumable operation. In the resuming model, control flow transfers from the raise point to a handler to correct an incomplete operation, and then *back* to the raise point to continue execution. Resumption is often likened to a *normal* routine call, particularly when argument/parameters are available, as both return to the dynamic location of the call [Geh92, DG94]. However, a routine call is statically bound, whereas a resumption raise is dynamically bound.

Resumption is used in cases where fix-up and continuation are possible. For example, in large scientific applications, which run for hours or days, it is unacceptable to terminate the program for many "error" situations, such as computational problems like zero divide, overflow/underflow, etc. and logical situations like a singular matrix. Instead, these problems call a fix-up routine, which logs the problem and performs a fix-up action allowing execution to continue. While a fix-up may not result in an exact result, an approximation may suffice and many hours of computer time are salvaged.

Possibly the first programming language to introduce resumption was PL/I.[4] For example, in Figure 3.9, the PL/I-like program declares the built-in on-condition ZERODIVIDE in the scope of procedure TEST at line 1. (This example applies equally well to user defined on-conditions.) An on-condition is like a routine name that is dynamically scoped, i.e., a call to ZERODIVIDE (by the hardware on division by zero) selects the closest instance on the call stack rather than selecting an instance at compile time based on lexical scope. In detail, each on-condition has a stack, on which handler bodies are pushed, and a call to an on-condition executes the top handler (unless the condition is disabled). When an ON CONDITION statement is executed, it replaces the top element of the stack with a new handler routine. Replacement is used instead of associating resumption handlers with a block. The effect of entering a guarded block is achieved implicitly when flow of control enters a procedure or block within the lexical scope of on-condition variables by duplicating the top element of each stack. When flow of control leaves a procedure or block that duplicated the top element, the stacks are implicitly popped so any on-units set up inside a block disappear.

Stepping through the code in Figure 3.9:

1. The definition of the built-in on-condition ZERODIVIDE in the body of procedure TEST creates a stack with a system default handler. (This definition is superfluous because it already exists in the language preamble.)

[4] A common mistake about the ON CONDITION is that its semantics are termination rather than resumption. PL/I only provides termination via nonlocal **goto**, which requires programmers to construct the termination model by convention.

```
    TEST: PROCEDURE OPTIONS(MAIN);
1       DCL ZERODIVIDE CONDITION; /* default handler D */
        F: PROCEDURE;
            A = B / 0; /* call H1 */
4           ON CONDITION(ZERODIVIDE) RETURN 2; /* H2 */
            A = B / 0; /* call H2 */
5       END;

2       ON CONDITION(ZERODIVIDE) RETURN 1; /* H1*/
        A = B / 0; /* call H1 */
3       CALL F();
    END;
```

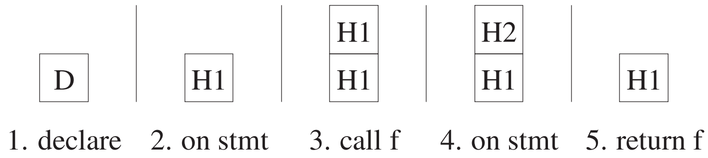


Figure 3.9: PL/I-like Resumption

2. The ON CONDITION statement for ZERODIVIDE replaces the default handler at the top of ZERODIVIDE's stack with a user handler-routine. (The PL/I mechanism for returning a result from the handler routine is simplified for this example.) Within TEST there is an implicit call to condition ZERODIVIDE via the hardware when the expression B / 0 is executed, which invokes the top handler, the handler executes, and the value 1 is returned for this erroneous expression. Therefore, A is assigned the value 1.

3. The call to procedure F duplicates the element of the ZERODIVIDE stack. At the start of F, there is an implicit call to condition ZERODIVIDE via the hardware when the expression B / 0 is executed, which performs the same handler action as before.

4. The ON CONDITION statement for ZERODIVIDE in f then replaces the handler at the top of ZERODIVIDE's stack, and this handler is signalled as before.

5. When F returns, the top (local) handler is popped from ZERODIVIDE's stack.

This facility provides polymorphism by allowing a fixed name, ZERODIVIDE, to have different handlers associated with it based on the dynamic versus the static structure of the program.

Liskov and Snyder [LS79, p. 549], and Mitchell and Stroustrup [Str94, p. 392] have argued against the resumption model but their reasons are largely anecdotal. Goodenough's resumption model is extensive and complex, and Mesa's resumption is based on this model [YB85, pp. 235-240]. However, there are only two technical issues that present difficulties with resumption: recursive resuming during propagation and accessing the resumption handler's lexical scope when the exception is caught. Both of these technical issues have reasonable solutions. Hence, while resumption has been rejected by some language designers, it is a viable and useful mechanism in an EHM. model can be as simple as dynamic routine call, which is easy to implement in languages with nested routines. However, a resumption model can be as simple as dynamic routine call, which is easy to implement in languages with nested routines. For languages without nested routines, like C++, it is still possible to construct a simple resumption model [BMZ92, Geh92, Buh06].

In summary, Figure 3.10 illustrates the different forms of dynamic propagation. The three forms of dynamic propagation (termination, retry, resumption) appear at the bottom-right and define where control continues after an exception is handled.

3.7 EHM Features

The previous discussion covers the fundamental features of an EHM. Now additional features are presented that make an EHM easier to use (see also [Goo75, KS90, BMZ92, DG94]). Some of these features can have a significant impact on the design of the EHM.

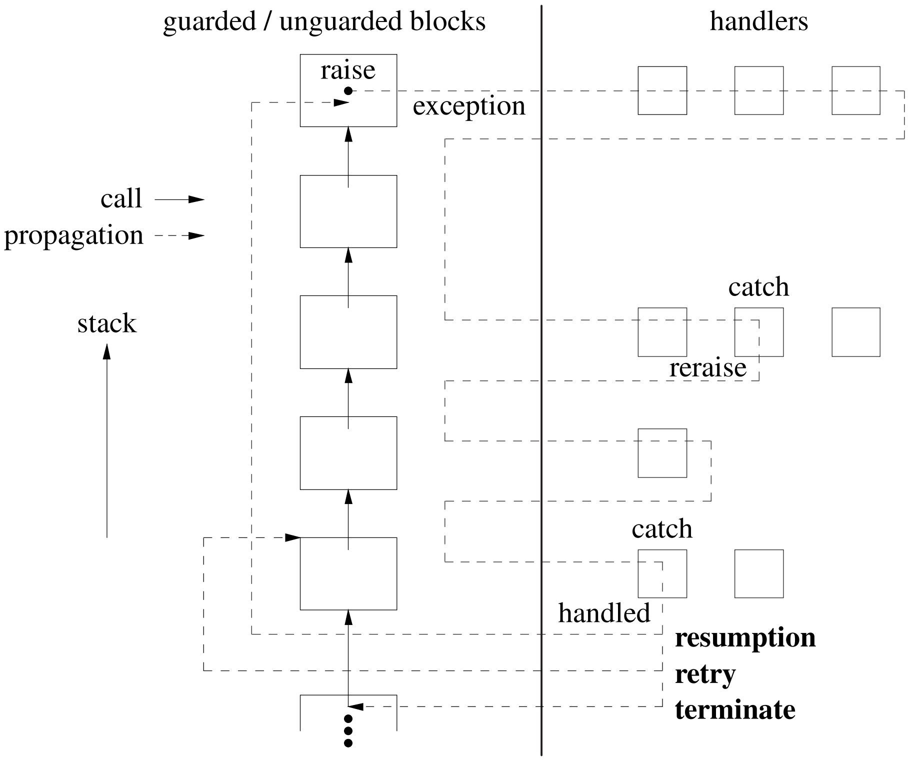
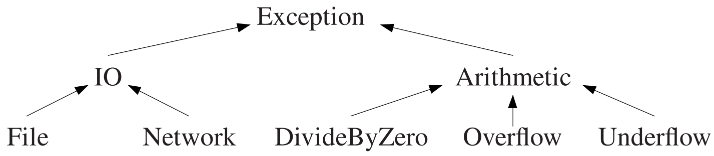


Figure 3.10: Dynamic Propagation

3.7.1 Derived Exception-Type

An exception type can be derived from another exception type, just like deriving a subclass from a class, providing a kind of polymorphism among exception types. The exception-type hierarchy that is created is also useful to organize exception types, similar to a class hierarchy in object-oriented languages, e.g.:



A programmer can then choose to handle an exception at different degrees of specificity along the hierarchy; derived exception-types support a more flexible programming style, and hence, significantly enhance an EHM. For example, higher-level code should catch general exceptions to reduce coupling to the specific implementation at the lower levels; unnecessary coupling may force changes in higher-level code when low-level code changes. A consequence of derived exception-types is that a handler may match multiple exceptions, e.g., the handler:

```
catch( Arithmetic )
```

can match all three derived exception-types: DivideByZero, Overflow, and Underflow. To handle this case, most propagation mechanisms perform a simple linear search of the handler clause for a guarded block and select the first matching handler, so the order of catch clauses in the handler clause becomes important, e.g.:

```
try {
    ...
} catch( Overflow ) {      // must appear first
    // handle overflow
} catch( Arithmetic )
    // handle other arithmetic issues
}
```

An important design question is whether to allow derivation from multiple exception types, called **multiple derivation**, which is similar to multiple inheritance of classes. While Cargill [Car90] and others argue against multiple inheritance among classes in object-oriented programming, multiple inheritance among derived exception-types is different. For example, consider the following multiple inheritance among derived exception-types [KS90, p. 19]:

```
exception network_err, file_err;
exception network_file_err : network_err, file_err; // multiple derivation
```

which derives network_file_err from network_err and file_err. While this looks reasonable, there are subtle problems:

```
try {
    ... throw network_file_err ...
} catch( network_err ) ...     // close network connection
  catch( file_err ) ...        // close file
```

If network_file_err is raised, neither of the handlers may be appropriate to handle the raised exception, but more importantly, which handler in the handler clause should be chosen because of the inheritance relationship? Executing both handlers may look legitimate, but indeed it is not. If a handler clause has a handler only for file_err, does it mean that it cannot handle network_file_err completely and should raise network_err afterwards? The example shows that handling an exception having multiple parents may be inappropriate. If an exception cannot be caught by one of its parents, the derivation becomes moot. Therefore, multiple derivation is a questionable feature for derived exception-types as it introduces significant complications into the semantics with little benefit.

3.7.2 Catch-Any and Reraise

Catch-any is a mechanism to match any exception propagating through a guarded block, e.g.:

```
try {
    ...
} catch( E ) {          // handle specific exceptions
  ...
} catch(...) {          // catch any remaining exceptions
    // general cleanup
}
```

For termination, this capability is often used as a general cleanup when a non-specific exception occurs during a guarded-block's execution. For resumption, this capability allows a guarded block to gather or generate information about control flow passing through a guarded block, but only in one direction, i.e., during the raise versus the resumption. With exception-type inheritance, catch-any can be provided by the root exception-type, e.g., **catch**(Exception) in Java. When there is no predefined exception-type hierarchy, special syntax is necessary, e.g., **catch**(...) in C++.

Java block finalization:

```
try {
    ...
} catch( E ) { ... }
  ... // other catch clauses
} finally { ... } // always executed
```

is executed on *both* normal and exceptional termination, providing additional catch-any capabilities plus dealing with the non-exceptional case. Interestingly, finalization is difficult to mimic with other language features that handle finalization, like destructors in C++. The left example, in Figure 3.11 shows how a finally clause can be used to ensure an open file is closed regardless of how the block manipulating the file ends. The simulations in the centre and right examples both have a problem preventing duplication of the cleanup code, which forces the use of a routine or class for the cleanup. Unfortunately, this approach makes accessing local variables in the block containing the **try** statement difficult from the cleanup routine or class. For systems with nested routines and classes, the references can be direct; otherwise, variables must be explicitly passed to the cleanup routine/class.

Reraise is a mechanism for aborting the current handling of a caught exception and restarting its propagation from the handler's context. Reraise is useful if a handler detects it is incapable of (completely) handling an exception and needs to propagate the same exception to a handler further down the stack. Reraise is provided by a raise statement without an exception type, e.g.:

Finalization

```
int main() {
    int f = open(...);
    try {
        // file operations

    } finally {
        close(f);
    }
}
```

Routine Simulation

```
void cleanup(int &f) {
    close(f);
}
int main() {
    int f = open(...);
    try {
        // file operations
        cleanup(f);
    } catch( ... ) {
        cleanup(f);
    }
}
```

Destructor Simulation

```
class cleanup {
    int &f;
  public:
    cleanup(int &f) : f(f) {}
    ~cleanup() { close(f); }
};
int main() {
    f = open(...);
    {
        cleanup v(f);
        // file operations
    }
}
```

Figure 3.11: Finalization

```
} catch ( E e ) {
    ... throw;    // no exception type
```

Here, the implicit exception for the reraise is the exception from the original raise.

3.7.3 Exception Parameters

Of almost equal importance to the control flow capability provided by exceptions is the ability to pass information from the raise to the handler. (This statement is also true for a routine.) Without this capability, the exact reason for raising the exception and/or where the exception was raised cannot be transferred to the exception handler. Without this information, a handler may not be able to determine the extent of the problem and if recovery is possible.

Exception parameters enable a source execution to transfer information into and out of the faulting execution's handler, just like routine parameters and results. An exception parameter can be read-only, write-only and read-write. While information could be passed through shared objects, exception parameters eliminate side-effects and locking in a concurrent environment. Ada has no parameters, C (via setjmp/longjmp) has a single integer parameter, Modula-3 and C++ have a single general parameter, ML and Mesa have multiple parameters.

Parameter specification for an exception-type depends on the form of its definition. In Mesa and Modula-3, a technique similar to routine parameters is used, as in:

```
exception E( int );        // exception definition with parameter
throw E( 7 ) ...           // integer argument supplied at raise
catch( E( int p ) ) ...    // integer parameter p received in handler
```

In C++, an object type is the exception-type and an object instance is created to contain any parameters, as in:

```
struct E {
    int i;                 // parameter
    E(int p) { i = p; }
};
void rtn(...) { ... throw E( 7 ); ... }  // object argument supplied at raise
int main() {
    try {
        ... rtn(...); ...
    } catch( E p ) {       // object parameter p received in handler
        // use p.i
    }
}
```

In all cases, it is possible to have parameters that are routines (or member routines), and these routines can perform special operations. For example, by convention or with special syntax, an argument or member routine can be used as a **default handler**, which is called if the faulting execution does not find a handler during propagation, as in:

```
void f(...) {...}
exception E( ... ) default( f ); // special syntax, default routine f
struct E { ... void default() {...}; }; // special name, default member
```

Other specialized operations are conceivable.

Finally, with derived exception-types, parameters to and results from a handler must be dealt with carefully depending on the particular language. For example, in Figure 3.12 exception-type D is derived from B with additional data fields for passing information into and out of a handler. When an exception of type D is raised and caught by a handler for exception-type B, it is being treated as a B exception-type within the handler and the additional data field, j, cannot be accessed safely without a dynamic down-cast. Consequently, if the handler returns to the raise point, some data fields in the exception may be uninitialized. A similar problem occurs if static dispatch is used instead of dynamic (both Modula-3 and C++ support both forms of dispatch). The handler matching an exception of type D with exception type B may only call members in B because of static dispatch; any overridden or new members in the actual exception are inaccessible. For termination, these problems do not exist because the handler parameters are the same or up-casts of arguments. For resumption, any result values returned from the handler to the raise point are the same or down-casts of arguments. However, the problem of down-casting is a subtyping issue, independent of the EHM, which programmers must be aware of when combining derived exception-types and exception parameters with resumption.

```
exception B { // base
    int i;
    void m() {...}
};
exception D : B { // derived
    int j;
    void m() {...}
};
void f() {
    throw D(); // derived
}
void g() {
    try {
        f();
    } catch( B b ) {
        // cannot access D::j without down-cast
        b.m(); // calls D::m or B::m ?
    }
}
```

Figure 3.12: Subtyping

3.7.4 Exception List

An **exception list**[5] is part of a routine's prototype and specifies which exception types may propagate from the routine to its caller, e.g., in Goodenough, CLU, Modula-3, Java and C++. For example, in the routine prototype:

```
int rtn() throw(E1,E2);
```

the exception list, **throw**(E1,E2), indicates that only exception types E1 and E2 may be raised to any caller of rtn. The purpose of the exception list is to make it possible to ameliorate the visibility problem associated with dynamic propagation (see Section 3.6.2, p. 29) via the following. First, the union of all exception lists of called routines defines all possible kinds of exceptions that can propagate into the caller's scope. Second, the exception list for the caller's scope restricts which exception types from its called routines can pass to the next level and which ones must be handled within the routine. Therefore, when an exception is raised, it is known statically that all blocks traversed during propagation are aware of this particular exception's type, and that the exception is ultimately handled at some point during propagation. From an implementation standpoint, the compiler checks the static call-graph of

[5]Called an **exception specification** in C++.

a program to ensure every raised exception ultimately propagates into a guarded block with an appropriate handler, except possibly for the main program, which may be able to propagate exceptions to the language's runtime system. Exception lists can be used to establish invariants about control flow, especially in situations where being interrupted by an exception causes problems. For example, in

```
int f() throw();        // throws no exceptions
int rtn() {
    try {
        // may throw E
    } catch( E ) {
        . . . f(); . . .
    }
}
```

it is essential that the clean-up code in the handler not be interrupted, otherwise the program is left in an inconsistent state. Since f's definition states it throws no exceptions, it is possible to formally reason that the clean-up code always runs to completion. In C++, this kind of reasoning is particularly important for the destructor of a class because it is illegal to throw an exception in a destructor executed during propagation.

A consequence of exception lists is that nonlocal handling requires intermediate routines in a call graph to list all exception types that may propagate through it from lower-level to higher-level routines, in which the exception type is ultimately handled. In some cases, the exception types associated with nonlocal handling have little or no meaning at the intermediate levels. So while these exception types are syntactically visible to intermediate routines, they are semantically invisible. The result is unwanted coupling between low and intermediate-level routines, i.e., a change to a lower-level routine raising an exception may cause a cascade of changes in exception lists for intermediate-level routines, where these changes may be largely irrelevant. Another consequence of exception lists is that handlers must ultimately exist for all possible exception types. This may result in the main routine of a program having a guarded block with a large or a catch-any handler clause to deal with exception types not handled lower in the call graph. Hence, there can be unwanted coupling between low and high-level routines. Some languages allow the main routine to have an exception list, allowing certain exceptions to be propagated to the language's runtime system. The assumption is that the runtime system may have a default handler for these exceptions, which performs some appropriate action.

Unfortunately, the formality of statically-checked exception-lists can become annoying to programmers, resulting in their circumventing much its benefits. For example, it is common in Java programs to see the following idioms, which negate the benefits of statically-checked exception-lists. One idiom is to have a try block with an empty handler for an exception type, as in:

```
try { wait(); } catch(InterruptedException ex) { /* empty handler */ }
```

The handler is empty because the exception occurs rarely and/or the programmer has no obvious action to perform should the exception occur; however, without the handler the program does not compile. An alternative idiom to deal with this situation is over-relaxing the routine's exception-list by specifying the root of the exception-type hierarchy, which allows any exception to propagate from a routine. In other words, the programmer is pushing the issue onto the next level simply to make progress. The conclusion is that the rigid specification and handling adherence required by statically-checked exception-lists may be too high a requirement for many programming situations, and hence, programmers find ways to relax these requirements to simplify program construction, which negates the purpose of exception lists.

An alternative approach to static checking of exception lists is dynamic checking, or a combination of static and dynamic checking. During propagation, at each routine activation, a comparison is made of the raised exception with the routine's exception-list. From a software engineering perspective, the dynamic approach is less attractive because there is now the potential for a runtime error versus a compile-time error. Unlike the static approach, which simply does not compile the program if an exception list is violated, the dynamic case results in a runtime violation and some appropriate semantics must exist for this occurrence. For example, the program can be terminated, the specific exception can be changed to a general failure exception which conceptually appears in all exception lists, or a special user supplied routine is called to take some action. Terminating the program is normally unacceptable in some programming situations. Converting the specific raised exception to the failure exception when an exception list is violated, precludes any chance of handling the specific exception-type and only complicates any recovery. The problem is exacerbated when a raised exception has an argument because the argument is lost in the conversion.

C++ supports a combination of static and dynamic checking of exception lists. Static checking occurs when overriding virtual routines and on assignment to routine pointers. For virtual routines, any routine overriding a virtual routine in any derived class may only allow exception types that are allowed by the exception list of the base class virtual routine, e.g.:

```
struct B {
    virtual void f() throw(E);
};
struct D : B {
    void f(); // static error
};
```

The definition of D::f is checked statically and fails because it allows all exception types, whereas B::f allows only exception type E. Similarly, any initialization or assignment to a routine pointer must only allow exception types that are allowed by the pointer being initialized or assigned, e.g.:

```
void (*pf1)();
void (*pf2)() throw(E);
void f() {
    pf1 = pf2; // OK: pf1 is less restrictive
    pf2 = pf1; // static error: pf2 is more restrictive
}
```

In addition, exception specifications on return types and parameter types must also match exactly. Otherwise, all verification of checked exception types occurs dynamically during propagation, so at each routine activation during propagation, a comparison is made of the raised exception type with the routine's exception-list. If the comparison fails to find the raised exception type in the exception list, the routine unexpected is called, which must not return but can raise another exception. C++ explicitly chose to perform dynamic checking over static checking to allow programmers to control the execution behaviour when an exception list is violated through the unexpected routine. Otherwise, C++ could have adopted a static checking approach.

Interestingly, static checking cannot cover all possible situations where exceptions can be raised. Many hardware-specific failures resulting in exceptions can occur at any time, making it impossible to know statically which exception types can occur within a routine. In addition, a nonlocal exception may be propagated at essentially any time, making it impossible to know statically which exception types can occur within a routine. An exception type that is allowed to propagate even when it is not in an exception list is called **unchecked**. Another way of thinking about an unchecked exception is that it appears implicitly in every exception list because it can happen in virtually all routines. Either way, an unchecked exception-type is allowed to propagate into and out of a scope where it is invisible, which is the visibility problem for dynamic propagation. Java supports unchecked exception-types by having all exception types organized into a hierarchy divided into checked and unchecked; an exception type can be derived from either branch of the hierarchy using exception-type derivation. Only checked exception-types can appear in exception lists, and hence, are statically checked. C++ does not have unchecked exception-types but rather unchecked routines, i.e., a routine without an exception list means this routine has all exception types in its exception list, and hence, any exception can pass through this routine.

While exception lists are a useful software engineering mechanism to further specify a routine's behaviour, exception lists can constrain polymorphism [Str94, p. 394], having a significant feature interaction between the EHM and a language's type system. For example, consider the simplified C++ template routine sort:

```
template<class T> void sort( T items[] ) {
    // using bool operator<( const T &a, const T &b );
}
```

using the operator routine < in its definition. There are two possibilities with respect to raising exceptions:

1. sort may directly or indirectly raise exceptions
2. operator routine < used by sort may raise exceptions

If sort raises exceptions, it must have an exception list, e.g.:

```
template<class T> void sort( T items[] ) throw(E1) {
    // using bool operator<( const T &a, const T &b );
}
```

If routine < raises exceptions, static checking requires that all of its exception types must be caught in sort or appear in sort's exception list. For languages requiring explicit specification of all required routines as part of a template definition (e.g., Ada, Haskell), it is possible to build on this facility to extend static exception-checking. For example, extending the syntax for the the template routine sort to:

```
template<class T> void sort( T items[ ] ) throw(E1)
    with bool operator<( const T &a, const T &b ) throw(E2); // required routine with exception list
```

implies all < routines used by sort can throw exception type E2, which sort must handle or include in its exception list. However, any type T with an operator routine < that has a larger exception list than what sort expects cannot be sorted, constraining reuse. A slightly more general approach is to relax the constrains on the required routines but compensate by specifying which exceptions are caught within sort, as in:

```
template<class T> void sort( T items[ ] ) throw(E1), catch(E2)
    with bool operator<( const T &a, const T &b ) throw(E2), catch(E3);
```

This prototype states that sort allows E1 to escape from it but catches exception-type E2 within its body, including those that might be generated by an required routine it might call. A similar specification is made by operator routine < with respect to exception-types E2 and E3. This additional information still allows static checking of exceptions but reduces the constraints on the prototypes for the required routines. For example, any set of required routines that raises at most exception types E1 and E2 is applicable because these exception types are either caught in sort or appear in sort's exception list. Similarly, if operator routine < can call any of other routine that raises at most exception types E2 and E3. However, reuse is still constrained because the set of exception types that can be raised by the required routines is restricted to those specified in the exception and catch lists.

A general solution is suggested but only for languages that have complete access to all source code (or its equivalent); hence, this solution is not applicable for separate-compilation systems where only partial information about a definition is available at a declaration. The approach is to extend the syntax for instantiation of a template routine by adding an exception list:

```
sort( v ) throw(E4); // instantiate with exception list
```

implying sort's exception list is extended with exception type E4 in this context, which allows a routine < raising E4 to be used. In C++, this extension could work because a template is expanded as if "inlined", so conceptually a new sort routine is generated for each instantiation; hence, there are actually multiple versions of sort each with a different prototype. This additional information at the call makes it possible for the compiler to statically determine which exception types propagate *within* sort and *into* the local context of its calls. However, this solution imposes a significant burden on definition of sort, as it can no longer make certain assumptions about which exceptions may or may not occur in particular contexts. Hence, sort must be designed to be more robust with respect to failure points in its execution to allow this approach.

Exception lists also constrain reuse for arguments of routine pointers (functional style) and/or polymorphic methods or routines (object-oriented style), e.g.:

```
int f( int (*p)(...) throw() ) {
    ... *p(...) ...
}
int g() throw(E) { throw E(); }
int h(...) {
    try {
        ... f( g ); ...
    } catch( E ) ...
}
```

```
class B {
    virtual int g() throw() {}
    int f() { ... g(); ... }
};
class D : public B {
    int g() throw(E) { throw E(); }
    int h() {
        try {
            ... f(); ...
        } catch( E ) ...
    }
};
```

The left example illustrates arguments of routine pointers, where routine h calls f passing argument g, and f calls g with the potential to raise exception type E. Routine h is clearly capable of handling the exception because it has an appropriate **try** block and is aware the version of g it passes to f may raise the exception. However, this reasonable case is precluded because the prototype of the argument routine g is less restrictive than the parameter variable p of f, and only works if the exception is passed through the intermediate routine f. Similarly, the right example illustrates object-oriented dynamic dispatch, where the derived class replaces member g, which is called from member B::f. Member routine D::h calls B::f, which calls D::g with the potential to raise exception type E. Member D::h is clearly capable of handling the exception because it has an appropriate **try** block and it created the version of g raising the exception. However, this reasonable case is precluded because the prototype of D::g is less restrictive than B::g, and only works if the exception is passed through the intermediate routine B::f. If f in the left example or B in the right example are pre-compiled in a library, there is no option to expand the prototypes to allow this reuse scenario. Nor is it reasonable to expand the prototype for every routine. In fact, doing so makes the program less robust because the prototype now covers too broad a range of exception types.

In general, programmers do not want to inhibit reuse, nor do they want to write seemingly unnecessary specifications, but they also want good software-engineering practice, such as static exception-list checking. Unfortunately, these are conflicting goals. In many cases, programmers under-constraining a routine by allowing any exception not specifically related to it to propagate out of the routine, which temporarily simplifies software development. However, this approach can be very disruptive to other developers calling this routine as now they are required to deal with even more exceptions seemingly unrelated to their code. The result is a cascade of exceptions from multiple levels, which are often deferred higher and higher in the abstraction chain. This approach is usually supported in programming languages with static exception-checking, and is commonly used by programmers. For example, in Java exception checking can be deferred by specifying the root of the exception hierarchy in an exception list, i.e., **throw**(Exception). Alternatively, programmers simply can catch and ignore exceptions they do not want to deal with. Again, in Java, it is common to see code like:

```
try { wait(); } catch( InterruptedException ex ) {};
```

in concurrent programs, where an exception of type InterruptedException is ignored because it cannot occur or there is no obvious correction should it occur. The maximum under-constraining is to allow *all* exceptions to propagate from *all* routines, which clearly simplifies programming but defeats the software engineering goal provided by static exception-checking. And at some point, in a complete program with static exception-checking, unspecified exceptions must be dealt with. Either the "main" routine of a program must catch all exceptions and do something appropriate or possibly the language preamble (implicit lexical scope in which a program is compiled) implicitly does this. In either case, if too many exceptions are deferred to this point, the purpose of static exception-list checking has been largely circumvented. Similarly, with dynamic checking, it is possible for a programmer to defer exceptions until they must be handled at the top-level of execution, again, often when the top of the stack for the "main" routine is reached.

Finally, Section 3.6.1, p. 28 mentions sequels can be passed as arguments to pre-compiled routines to perform exception handling. Similarly, some schemes for simulating resumption involve passing fix-up routines, which are called to correct a problem. However, the list of sequel or fix-up routine parameters is similar to an exception list because it makes a routine's prototype more restrictive, which constrains reuse.

3.7.5 Bound Exceptions and Conditional Handling

In Ada, an exception type declared in a generic package creates a new instance for each package instantiation, e.g.:

```
        generic package Stack is
            overflow : exception; . . .
        end Stack;
        package S1 is new Stack;    -- new overflow
        package S2 is new Stack;    -- new overflow
    begin
        . . . S1.push(. . .); . . . S2.push(. . .); . . .
    exception
        when S1.overflow => . . .   -- catch overflow for S1
        when S2.overflow => . . .   -- catch overflow for S2
```

Hence, it is possible to distinguish which stack raised the overflow without passing data to the handler. In object-oriented languages, the class is used as a unit of modularity for controlling scope and visibility. Similarly, it makes sense to associate exception types with the class that raises them, as in:

```
class file {
    exception file_err; . . .
```

However, is the exception type associated with the class or objects instantiated from it? As above, the answer affects the capabilities for catching the exception, as in:

```
file f;
try {
    ... f.read(...); ...            // may raise file_err
} catch( file.file_err ) ...        // option 1
  catch( f.file_err ) ...           // option 2
```

In option 1, only one file_err exception type exists for *all* objects created by type file. Hence, this **catch** clause deals with file_err events regardless of which file object raises it. In option 2, each file object has its own file_err exception type. Hence, this **catch** clause *only* deals with file_err events raised by object f, i.e., the handler is for an exception bound to a particular object, called a **bound exception**. This specificity prevents the handler from catching the same exception bound to a different object. Both facilities are useful but the difference between them is substantial and leads to an important robustness issue. Finally, an exception type *among* classes is simply handled by declaring the exception type outside of the classes and referencing it within the classes.

Bound exceptions cannot be trivially mimicked by other mechanisms. Deriving a new exception type for each file object (e.g., f_file_err from file_err) results in an explosion in the total number of exception types, and cannot handle dynamically allocated objects, which have no static name. Passing the associated object as an argument to the handler and checking if the argument is the bound object, as in:

```
catch( file.file_err( file *fp ) ) { // fp is passed from the raise
    if ( fp == &f ) ...     // deal only with f
    else throw              // reraise exception
```

requires programmers to follow a coding convention of reraising the exception if the bound object is inappropriate [BMZ92]. Such a coding convention is unreliable, significantly reducing robustness. In addition, mimicking becomes infeasible for derived exception-types using the termination model, as in:

```
exception B( obj );        // base exception-type
exception D( obj ) : B;    // derived exception-type
obj o1, o2;
try {
    ... throw D(...);                       |   // bound form
} catch( D(obj *o) ) {                      |   } catch( o1.D ) {
    if ( o == &o1 ) ...  // deal only with o1 |
    else throw           // reraise exception |
} catch( B(obj *o) ) {                      |   } catch( o2.B ) {
    if ( o == &o2 ) ...  // deal only with o2 |
    else throw           // reraise exception |
```

When exception type D is raised, the problem occurs when the first handler catches the derived exception-type and reraises it if the object is inappropriate. The reraise for the termination model immediately terminates the current guarded block, which precludes the handler for the base exception-type in that guarded block from being considered. Therefore, the "catch first, then reraise" approach is an incomplete substitute for bound exceptions.

Finally, it is possible to generalize the concept of the bound exception with **conditional handling** [Mok97], as in:

```
catch( E( obj &o ) ) when( o.f == 5 ) ...
```

where the when clause specifies a general conditional expression that must also be true before the handler is chosen. Conditional handling can mimic bound exceptions simply by checking if the object parameter is equal to the desired object. Also, the object in the conditional does not have to be the object containing the exception-type definition, as for bound exceptions. The problem with conditional handling is the necessity of passing the object as an argument or embedding it in the exception before it is raised. Furthermore, there is now only a coincidental connection between the exception and conditional object versus the statically nested exception type in the bound object.

3.8 Propagation Mechanisms

Propagating directs control flow of the faulting execution to a handler; the search for a handler proceeds through the blocks, guarded and unguarded, on the faulting execution's stack. Different actions may occur during the search depending on the kind of propagation, where the kinds of propagation are termination and resumption, and both forms can coexist in a single EHM.

Terminating or **throwing** propagation means control does not return to the raise point. The unwinding associated with terminating normally occurs during propagation, although this is not required; unwinding can occur when the handler is found, during the handler's execution, or on its completion. However, there is no advantage to delaying unwinding for termination, and doing so results in problems (see "Lexical Context", p. 30 and Section 3.12, p. 51) and complicates most implementations.

Resuming propagation means control returns to the point of the raise; hence, there is no stack unwinding. However, a handler may determine that control cannot return, and needs to unwind the stack, i.e., change the resume into a terminate. This capability is essential to prevent unsafe resumption, and mechanisms to accomplish it are discussed below.

Three approaches for associating terminating or resuming propagation with an exception are possible:

1. At the definition of the exception type, as in:

   ```
   terminate E1;           // specific definition
   resume E2;
   try {
         ... throw E1;     // generic raise
         ... throw E2;
   } catch( E1 ) ...       // generic handler
     catch( E2 ) ...
   ```

   Associating the propagation mechanism at exception-type definition means the raise and handler can be generic. In this approach, there is a partitioning of exception types, as in Goodenough [Goo75] with ESCAPE and NOTIFY, μSystem [BMZ92] with exceptions and interventions, and Exceptional C [Geh92] with exceptions and signals.

2. At the raise of the exception, as in:

   ```
   exception E;            // generic definition
   try {
         ... terminate E;  // specific raise
         ... resume E;
   } catch( E ) ...        // generic handler
   ```

   Associating the propagation mechanism at the raise means the exception type and handler can be generic. In this approach, an exception type can be used for either kind of propagation, i.e., exception type E can imply termination or resumption depending on the raise. The generic handler catching the exception must behave according to the propagation mechanism associated with the raise of the exception. As a result, it is almost mandatory to have a facility in the handler to determine the kind of propagation, as different actions are usually taken for each.

3. At the handler, as in:

   ```
   exception E;            // generic definition
   try {
         ... raise E;      // generic raise
   } terminate( E ) ...    // specific handler
   try {
         ... raise E;      // generic raise
   } resume( E ) ...       // specific handler
   ```

   Associating the propagation mechanism at the handler means the exception type and raise can be generic. In this approach, an exception type can be used for either kind of propagation, i.e., exception type E can imply termination or resumption depending on the handler. Note that stack unwinding must be delayed until the handler is found because that is the point when the kind of propagation is determined. However, it is ambiguous to have the two handlers appear in the same handler clause for the same exception.

```
exception E;            // generic definition
try {
    . . . raise E;      // generic raise
} terminate( E ) . . .  // ambiguous choice
  resume( E ) . . .
```

For this approach, the choice of handling model can be further delayed using an unwind statement available only in the handler to trigger stack unwinding, as in:

```
exception E;            // generic definition
try {
    . . . raise E;      // generic raise
} catch( E ) {          // generic handler
    if (. . .) {
        . . . unwind;   // => termination
    } else {
        . . .           // => resumption
    }
```

In this form, a handler implies resumption unless an unwind is executed resulting in termination. The unwind capability in VMS [KGB88, Chapter 4] and any language with nonlocal transfer can support this approach. Both forms have implications with respect to the implementation because stack unwinding must be delayed, which can have an effect on other aspects of the EHM.

Unfortunately, this approach violates the EHM objective of preventing an incomplete operation from continuing, i.e., it is impossible at the raise point to ensure control flow does not return. Hence, this approach is rejected.

If an exception type can be overloaded, i.e., be both a terminating and resuming exception type, combinations of the previous approaches are possible, as in:

Approaches 1 & 2	Approaches 2 & 3
terminate E; // *overload, specific definition* resume E; **try** { . . . terminate E; // *specific raise* . . . resume E; } **catch**(E) . . . // *generic handler*	exception E; // *generic definition* **try** { . . . terminate E; // *specific raise* . . . resume E; } terminate(E) . . . // *overload, specific handler* resume(E) . . .

In both forms, the kind of propagation mechanism for an exception type is specified at the raise and fixed during the propagation. In the left example, exception type E is overloaded at the definition and the generic handler catching the exception must behave according to the propagation mechanism associated with the exception type. As mentioned, it is almost mandatory to have a facility in the handler to determine the propagation mechanism to ensure the correct action is taken. In general, it is better software engineering to partition the handler code for each kind of propagation mechanism rather than combine it in a single handler. In the right example, the generic exception type is made specific at the raise and the overloaded handlers choose the appropriate kind. In this form, the handler code is partitioned for each kind of propagation mechanism. However, unlike the previous form, where the exception type is generic, this form allows a type to be specific, which describes how the exception type may be used in the program, i.e., terminate, resume or both.

Finally, it is possible to combine all three previous approaches, as in:

```
terminate E; // overload, specific definition
resume E;
try {
    . . . terminate E; // specific raise
    . . . resume E;
} terminate( E ) . . . // overload, specific handler
  resume( E ) . . .
```

While pedantic, the redundancy of this format helps in reading the code because the definition specifies the kind of the propagation mechanism (especially when the exception type is part of an interface). As well, it is unnecessary to have a mechanism in the handler to determine the kind of raised exception. The EHM in μSystem [BMZ92] uses all three locations to specify the propagation mechanism.

In an EHM where terminating and resuming coexist, it is possible to partially override their semantics by raising an exception within a handler, as in:

```
try {                                try {
    ... resume E1;                       ... terminate E1;
} catch( E1 ) terminate E2;          } catch( E1 ) resume E2;
```

In the left example, the terminate overrides the resuming and forces stack unwinding, starting with the stack frame of the handler (frame on the top of the stack), followed by the stack frame of the block that originally resumed the exception. In the right example, the resume cannot override the terminate because the stack frames are already unwound, so the new resume starts with the handler stack frame.

3.9 Exception Partitioning

As mentioned, associating the propagation mechanism at exception definition results in **exception partitioning** into terminating and resuming exception types. Without partitioning, i.e., generic exception-definitions, every exception type becomes **dual** as it can be raised with either form of propagation mechanism. However, an exception definition should reflect the nature of the exceptional event. For example, UNIX signals SIGBUS or SIGTERM always lead to termination of an operation, and hence, should be declared as termination-only. Indeed, having termination-only and resume-only exception types removes the mistake of using the wrong kind of raise and/or handler.

However, having a dual exception-type is also useful. While overloading an exception type allows it to be treated as a dual, few languages allow overloading of variables in a block. Alternatively, an exception type can be declared as dual. Both forms of making an exception type dual have the following advantages. First, encountering an exceptional event can lead to resuming or terminating an exception type depending on the particular context. Without dual exception-types, two different exception types must be declared, one being terminate-only and the other resume-only. These two exception types are apparently unrelated without a naming convention; using a single dual exception-type is simpler. Second, using a dual exception-type instead of resume-only for some exceptional events allows a resume exception-type to be terminated when no resumption handler is found. This effect can be achieved through a default resumption handler that raises a termination exception. Essentially terminate-only and resume-only exception types lack the flexibility of dual, and flexibility improves reusability. This observation does not imply all exception types should be dual, only that dual exception-types are useful.

3.9.1 Derived Exception Implications

With derived exception-types and partitioned exceptions, there is the issue of deriving one kind of exception type from another, e.g., terminate from resume, called **heterogeneous derivation**. If the derivation is restricted to exception types of the same kind it is called **homogeneous derivation**. Homogeneous derivation is straightforward and easy to understand. Heterogeneous derivation is complex but more flexible because it allows deriving from any kind of exception type. With heterogeneous derivation, it is possible to have all exception types in one hierarchy.

The complexity with heterogeneous derivation comes from the following derivations:

parent	terminate	resume	dual	dual	terminate	resume
	↓	↓	↓	↓	↓	↓
derived	resume	terminate	resume	terminate	dual	dual
option	1		2		3	

In option 1, the kind of exception type is different when the derived exception-type is raised and the parent is caught. If a resume-only exception type is caught by a terminate-only handler, it could unwind the stack, but that invalidates resumption at the raise point. If a terminate-only exception type is caught by a resume-only handler, it could resume the exception, but that invalidates the termination at the raise point. In option 2, problems occur when the handler for the dual exception-type attempts to perform an unwind or resume on an exception of the wrong kind, resulting in the option 1 problems. In option 3, there is neither an obvious problem nor an advantage if the dual exception-type is caught by the more specific parent. In most cases, it seems that heterogeneous derivation does not simplify programming and may confuse programmers; hence, it is a questionable feature.

3.10 Matching

In Section 3.8, p. 45, either the exception definition or the raise fixes the propagation mechanism. The propagation mechanism then finds a handler matching both the kind and exception type. However, there is no requirement the kind must match; only the exception type must match, which leads to four possible situations in an EHM:

	termination handler	resumption handler
termination	1. matching	2. unmatching
resumption	3. unmatching	4. matching

Up to now, **matching** has been assumed between handling model and propagation mechanism, i.e., termination matches with terminating and resumption with resuming. However, the other two possibilities (options 2 and 3) must be examined to determine whether there are useful semantics. In fact, this discussion parallels that for heterogeneous derivation.

In option 2, when a termination exception-type is raised, the stack is immediately unwound and the operation cannot be resumed. Therefore, a resumption handler handling a termination exception-type cannot resume the terminated operation. This semantics is misleading and difficult to understand, possibly resulting in an error long after the handler returns, because an operation raising a termination exception-type expects a handler to provide an alternative for its guarded block, and a resumption handler catching an exception expects the operation raising it to continue. Therefore, unmatching semantics for a termination exception-type is largely an unsafe feature.

In option 3, when an exception is resumed, the stack is not unwound so a termination handler has four possibilities. First, the stack is not unwound and the exception is handled with the resumption model, i.e., the termination is ignored. Second, the stack is unwound only after the handler executes to completion. Third, the stack is unwound by executing a special statement during execution of the handler (like `unwind`). Fourth, the stack is unwound after finding the termination handler but before executing it. The first option is unsafe because the termination handler does not intend to resume, and therefore, it does not correct the problem before returning to the raise point. The next two options afford no benefit as there is no advantage to delaying unwinding for termination, and doing so results in problems (see “Lexical Context”, p. 30 and Section 3.12, p. 51) and complicates most implementations. These problems can be avoided by the fourth option, which unwinds the stack before executing the handler, essentially handling the resumed exception-type as a termination exception-type. It also simplifies the task of writing a termination handler because a programmer does not have to be concerned about unwinding the stack explicitly, or any esoteric issues if the stack is unwound inside or after the termination handler. Because of its superiority over the other two options favouring termination, the last option is the best unmatching semantics for a resume exception-type (but it is still questionable).

With matching semantics, it is possible to determine what model is used to handle a raised exception (and the control flow) by knowing either how an exception is raised or which handler is chosen. Abstracting the resumption and the termination model is done in a symmetric fashion. The same cannot be said about unmatching semantics. In particular, it is impossible to tell whether a resumed exception is handled with the resumption model without knowing the handler catching it, but a termination exception is always handled with the termination model. Hence, terminating and resuming are asymmetric in unmatching semantics. Without knowing the handling model used for a resumed exception, it becomes more difficult to understand the resuming mechanism for unmatching semantics than the terminating and resuming mechanism for matching semantics. Therefore, unmatching semantics is inferior to matching and a questionable feature in an EHM.

3.11 Handler Clause Selection

The propagation mechanism determines how handler clauses are searched to locate a handler. It does not specify which handler in a handler clause is chosen if there are multiple handlers capable of catching the exception. For example, a handler clause can handle both a derived and base exception types. This section discusses issues about two orthogonal criteria — matching and specificity — for choosing a handler among those capable of handling a raised exception in a handler clause.

matching criteria (see Section 3.10) selects a handler matching with the propagation mechanism, e.g.:

```
try {
    resume E;
} terminate E ...
  resume E ... // matching
```

Matching only applies for an EHM with the two distinct propagation mechanisms and handler partitioning.

specificity criteria selects the most specific eligible handler within a handler clause using the following ordering rules:

1. The exception type is derived from another exception type (see Section 3.7.1, p. 37):

```
terminate B;
terminate D : B;
try {
    ... terminate D;
} terminate( D ) ... // more specific
  terminate( B ) ...
```

2. The exception is bound to an object rather than to a class (see Section 3.7.5, p. 44):

```
try {
    ... f.read(); ...
} terminate( f.file_err ) ... // more specific
  terminate( file.file_err ) ...
```

3. The exception is bound to the same object and derived from another exception type:

```
class foo {
    terminate B;
    terminate D : B;
    void m() { ... terminate D; }
    ...
foo f;
try {
    ... f.m();
} terminate( f.D ) ... // more specific
  terminate( f.B )...
```

In this case, it may be infeasible to tell which handler in a handler clause is more specific:

```
try {
    ...
} terminate( D ) ... // equally specific
  terminate( f.B ) ...
```

Here, there is a choice between a derived exception-type and a bound, base exception-type, which are equally specific.

Priorities must be set among these orthogonal criteria. In addition, the priority of handling a termination exception-type is orthogonal to that of a resumed one.

Dynamic propagation (see Section 3.6.2, p. 29) uses closeness, i.e., select a handler closest to the raise point on the stack, to first locate a possible set of eligible handlers in a handler clause. Given a set of eligible handlers, matching should have the highest priority, when applicable, because matching semantics is safe, consistent and comprehensible (see Section 3.10). A consequence of matching is a termination-handler hierarchy for termination exception-types and a resumption-handler hierarchy for resumed ones. With separate **handler hierarchies**, it is reasonable for an exception type to have both a default terminating and resumption handler (see Section 3.7.3, p. 39 concerning default handlers). It is still possible for a default resumption handler to override resuming (see Section 3.8, p. 45) and raise a termination exception-type in the terminating hierarchy. Overriding does not violate mandatory matching because of the explicit terminating raise in the handler. If there is no default handler in either case, the runtime system must take some appropriate action, usually terminating the execution entity.

Specificity is good, but comes after matching, e.g., if specificity is selected before matching in:

```
try {
    ...terminate D; ...     // D is derived from B
} terminate( B ) ...        // matching
  resume( D ) ...           // specificity
```

then the handler resume(D) is chosen, not that for terminate(B), which violates handler matching.

The only exception to these rules is when two handlers in the same handler clause are equally specific, requiring additional criteria to resolve the ambiguity. The most common one is the position of a handler in a handler clause, e.g., select the first equally matching handler found in the handler-clause list. Whatever this additional criteria is, it should be applied to resolve ambiguity only after using the other criteria.

3.12 Recursive Resuming

Recursive resuming is probably the only legitimate criticism against resuming propagation (see Section 3.6.2.2, p. 35), which can occur because resuming propagation does not unwind the stack. The simplest situation where recursive resuming can occur is when a handler for a resuming exception-type resumes the same exception, e.g.:

```
try {
    ... resume R; ...     // T{H(R)}    => try block T has handler H for exception-type R
} catch( R ) resume R;    // H(R)       => handler for R
```

The **try** block resumes R. Handler H is invoked by the resume, and the blocks on the call stack are:

... → T{H(R)} → H(R)   ← stack top

Then H resumes an exception of type R again, which finds the handler just above it at T{H(R)} and invokes handler H(R) again, and this continues until the runtime stack overflows. Recursive resuming is similar to infinite recursion, and is difficult to discover both at compile time and at runtime because of the dynamic choice of a handler. Concurrent resuming compounds the difficulty because it can cause recursive resuming where it is impossible otherwise because the concurrent exception can be delivered at any time.

However, not all exceptions handled by a resumption handler cause recursive resuming. Even if a resumption handler resumes the exception it handles, which guarantees activating the same resumption handler again, (infinite) recursive resuming may not happen because the handler can take a different execution path as a result of a modified execution state. Nor is it impossible to prevent recursive resuming via appropriate semantics during propagation. The mechanisms in Mesa [MMS79, p. 143] and VMS [KGB88, pp. 90-92] represent the two main approaches for solving this problem. The rest of this section looks at these two solutions.

3.12.1 Mesa Propagation

Mesa propagation prevents recursive resuming by not reusing an unhandled handler bound to a specific called block, i.e., once a handler for a block is entered it is **marked** as unhandled and not used again. The propagation mechanism always starts from the top of the stack to find an unmarked handler for a resume exception.[6] However, this unambiguous semantics is often described as confusing. The following program demonstrates how Mesa solves recursive resuming:

```
void test() {
    try {                                // T1{H1(R2)}
        try {                            // T2{H2(R1)}
            try {                        // T3{H3(R2)}
                resume R1;
            } catch( R2 ) resume R1;     // H3(R2)
        } catch( R1 ) resume R2;         // H2(R1)
    } catch( R2 ) ...                    // H1(R2)
}
```

The following stack is generated at the point when exception type R1 is resumed from the innermost **try** block:

test → T1{H1(R2)} → T2{H2(R1)} → T3{H3(R2)} → H2(R1)

[6] This semantics was determined with test programs and discussions with Michael Plass and Alan Freier at Xerox Parc.

The potential infinite recursion occurs because H2(R1) resumes an exception of type R2, and there is resumption-handler H3(R2), which resumes an exception of type R1, while resumption-handler H2(R1) is still on the stack. Hence, handler body H2(R1) invokes handler body H3(R2) and vice versa with no case to stop the recursion.

Mesa propagation prevents the infinite recursion by only marking an unhandled handler, i.e., a handler that has not returned, as ineligible (in bold), resulting in:

test → T1{H1(R2)} → T2{**H2(R1)**} → T3{H3(R2)} → H2(R1)

Now, H2(R1) resumes R2, which is handled by H3(R2):

test → T1{H1(R2)} → T2{**H2(R1)**} → T3{**H3(R2)**} → H2(R1) → H3(R2)

Therefore, when H3(R2) resumes R1 no infinite recursion occurs as the handler for R1 in T2{H2(R1)} is marked ineligible.

However, the confusion with the Mesa semantics is that there is no handler for R1, even though the nested **try** blocks appear to properly deal with this situation. In fact, looking at the static structure, a programmer might incorrectly assume there is an infinite recursion between handlers H2(R1) and H3(R2), as they resume one another. This programmer confusion results in a reticence by language designers to incorporate resuming facilities in new languages. In detail, the Mesa semantics has the following negative attributes:

- Resuming an exception type in a block and in one of its handlers can call different handlers, even though the block and its handlers are in the same lexical scope. For instance, in the above example, an exception generated in guarded block T3{H3(R2)} is handled by a handler *at* or *below* the block on the stack, H2(R1), but an exception generated in that handler body is handled by handlers dynamically *above* it on the stack, H3(R2). Clearly, lexical scoping does not reflect this semantics.
- Abstraction implies a routine should be treated as a client of routines it calls directly or indirectly, and have no access to the implementations it uses. However, if resuming from a resumption handler is a useful feature, some implementation knowledge about the handlers bound to the stack *above* it must be available to successfully understand how to make corrections, thereby violating abstraction. For example, assume guarded block T3{H3(R2)} is actually contained in library routine, f, and called from guarded block T2{H2(R1)}. It seems peculiar that after the original exception of type R1 raised in f is caught in T2{H2(R1)}, and a new exception of type R2 is raised, that this new exception is caught and handled by a handler back within f. Without knowing which and when handlers f are installed, it is is impossible to understand how control behaves.
- Finally, exception types are designed for communicating exceptional events from callee to caller. The peculiar behaviour observed in the previous point means resuming an exception inside a resumption handler is like propagating from caller to callee because of the use of handlers *above* it on the stack, making control flow very difficult to understand.

3.12.2 VMS Propagation

The VMS propagation mechanism solves the recursive resuming problem, but without the Mesa problems. Before looking at the VMS mechanism, the concept of a consequent event is defined, which helps to explain why the semantics of the VMS mechanism are desirable.

3.12.2.1 Consequent Events

Raising an exception implies an exceptional event has been encountered. A handler can catch an exception and then raise another exception if it encounters another exceptional event, resulting in a second exception. The second exception is considered a **consequent exception** of the first. More precisely, every exceptional event is an **immediate consequent event** of the most recent exception being handled in the execution (if there is one). For example, in the previous Mesa resuming example, the consequence sequence is R1, R2, and R1. Therefore, a consequent event is either the immediate consequent event of an event or the immediate consequent event of another consequent event. The consequence relation is transitive, but not reflexive. Hence, exceptions propagated when no other exceptions are being handled are the only nonconsequent events.

A concurrent exception is *not* a consequent event of other exceptions propagated in the faulting execution because the condition resulting in the exceptional event is encountered by the source execution, and in general, not related to the faulting execution. Only an exception raised *after* a concurrent event is delivered can be a consequent event of the concurrent event.

3.12.2.2 Consequential Propagation

The VMS propagation mechanism is referred to as **consequential propagation** because it builds on the notion of a consequent event to provide a solution for recursive resuming. The approach is based on the idea that if a handler cannot handle an event, it should not handle its consequent events, either. Conceptually, the propagation searches the execution stack in the normal way to find a handler, but marks as ineligible *all* handlers inspected, including the chosen handler. Marks are cleared only if an exception is handled, so any consequent event raised during handling does not see the marked handlers. Practically, all resumption handlers at each level are marked when resuming an exception; however, stack unwinding eliminates the need for marking when raising a termination exception-type. Matching (see Section 3.10, p. 49) eliminates the need to mark termination handlers because only resumption handlers catch resume exception-types. If a resumption handler overrides the propagation by raising a termination exception-type, the stack is unwound normally from the current handler frame.

How does consequential propagation make a difference? Given the previous runtime stack:

test → T1{H1(R2)} → T2{H2(R1)} → T3{H3(R2)} → H2(R1)

consequential propagation marks all handlers between the raise of R1 in T3{H3(R2)} to T2{H2(R1)} as ineligible (in bold):

test → T1{H1(R2)} → T2{**H2(R1)**} → T3{**H3(R2)**} → H2(R1)

Now, H2(R1) resumes R2, which is handled by H1(R2) instead of H3(R2).

test → T1{**H1(R2)**} → T2{**H2(R1)**} → T3{**H3(R2)**} → H2(R1) → H1(R2)

Like Mesa, recursive resuming is eliminated, but consequential propagation does not result in the confusing resumption of R1 from H3(R2). In general, consequential propagation eliminates recursive resuming because a resumption handler marked for a particular exception type cannot be called to handle its consequent events. As well, propagating a resume exception-type out of a handler does not call a handler bound to a stack frame between the handler and the handler body, which is similar to a termination exception-type propagated out of a guarded block because of stack unwinding.

Consequential propagation does not preclude all infinite recursions with respect to propagation, e.g.:

```
void test() {
    try {                       // T{H(R)}
        . . . resume R; . . .
    } catch( R ) test();        // H(R)
}
```

Here, each call to test creates a new **try** block to handle the next recursion, resulting in an infinite number of handlers:

test → T{**H(R)**} → H(R) → test → T{**H(R)**} → H(R) → test → . . .

As a result, there is always an eligible handler to catch the next exception in the recursion. Consequential propagation is not supposed to handle this situation as it is considered a programming error with respect to recursion not propagation.

Finally, consequential propagation does not affect termination propagation because marked resumption handlers are simply removed during stack unwinding. Hence, the application of consequential propagation is consistent with either terminating or resuming. As well, because of handler partitioning, a termination handler for the same exception type bound to a prior block of a resumption handler is still eligible, as in:

```
void test() {
    dual R; // terminate and resume
    try {                           // T{r(R),t(R)}
        . . . resume R; . . .
    } terminate( R ) . . .          // t(R)
      resume( R ) terminate R;      // r(R)
}
```

Here, the resume of R in the **try** block is first handled by r(R), resulting in the following call stack:

test → T{**r(R)**,t(R)} → r(R)

While r(R) is marked ineligible, the termination handler, t(R), for the same **try** block is still eligible. The handler r(R) then terminates the exception type R, and the stack is unwound starting at the frame for handler r(R) to the try block where the exception is caught by handler t(R), resulting in the following call stack:

test → t(R)

The try block is effectively gone because the scope of the handler does not include the try block (see Section 3.5.1, p. 25).

All handlers are considered unmarked when propagating nonlocal exceptions because the exception is unrelated to any existing propagation. Therefore, the propagation mechanism searches every handler on the runtime stack. Hence, a handler ineligible to handle a local exception and its consequent events can be chosen to handle a delivered nonlocal exception, reflecting the fact that a new propagation has started.

In summation, consequential propagation is better than other existing propagation mechanisms because:

- it supports terminating and resuming propagation, and the search for a handler occurs in a uniformly defined way,
- it prevents recursive resuming and handles normal and concurrent exceptions according to a sensible consequence relation among exceptions, and
- the context of a handler closely resembles its guarded block with respect to lexical location; in effect, an exception propagated out of a handler is handled as if the exception is directly propagated out of its guarded block.

3.13 μC++ EHM

The following features characterize the μC++ EHM, and differentiate it from the C++ EHM:

- μC++ supports three different kinds of exception types: throw, resume and dual; C++ only supports throw exception-types. Each kind of μC++ exception type has its own exception hierarchy. Like C++, these hierarchies are built by publicly inheriting from another exception type of the same kind, and exception parameters are encapsulated inside an exception. μC++ also extends the C++ set of predefined exception-types[7] covering μC++ exceptional runtime and I/O events.
- μC++ restricts raising of exceptions to specific exception-types; C++ allows any type to be raised.
- μC++ supports two forms of raising: throwing and resuming; C++ only supports throwing. Throw exception-types can only be thrown, resume exception-types can only be resumed, and dual exception-types can be either thrown or resumed. μC++ adopts consequential propagation mechanism so recursive resuming is eliminated, even for concurrent exceptions.
- μC++ supports two kinds of handlers: termination and resumption; C++ only supports termination handlers. Unfortunately, resumption handlers must be simulated using routines/functors due to the lack of nested routines in C++.
- μC++ supports raising of nonlocal and concurrent exceptions so that exceptions can be used to affect control flow *among* coroutines and tasks. The μC++ kernel implicitly polls for both kinds of exceptions after a context switch or unblocking. It is also possible to (hierarchically) block these kinds of exceptions when delivery would be inappropriate or erroneous.

3.14 Exception Type

While C++ allows any type to be used as an exception type, μC++ restricts exception types to those types defined by three kinds of classes: **_DualEvent**, **_ThrowEvent** and **_ResumeEvent**. An exception type has all the properties of a **class**, and its general form is:

```
_DualEvent exception-type name {  // or _ThrowEvent or _ResumeEvent
    . . .
};
```

As well, every exception type must have a public default and copy constructor.

[7] std::bad_alloc, std::bad_cast, std::bad_typeid, std::bad_exception, std::basic_ios::failure, etc.

3.14.1 Creation and Destruction

An exception is the same as a class-object with respect to creation and destruction:

```
_DualEvent D { ... };
D d;                    // local exception
_Resume d;
D *dp = new D;          // dynamic exception
_Resume *dp;
delete dp;
_Throw D();             // temporary local exception
```

3.14.2 Inherited Members

Each exception type, if not derived from another exception type, is implicitly derived from one of the following types, depending on the kind of exception type:

```
_DualEvent exception-type name : public uEHM::uDualClass ...
_ThrowEvent exception-type name : public uEHM::uThrowClass ...
_ResumeEvent exception-type name : public uEHM::uRaiseClass ...
```

where the interface for the base classes are as follows:

```
class uEHM::uDualClass {
    uDualClass( const char *const msg = "" );
    const char *const message() const;
    const uBaseCoroutine &source() const;
    const char *sourceName() const;
    virtual void defaultResume() const;
    virtual void defaultTerminate() const;
};
class uEHM::uThrowClass : private uEHM::uDualClass {
    uThrowClass( const char *const msg = "" );
    const char *const message() const;
    const uBaseCoroutine &source() const;
    virtual void defaultTerminate() const;
};
class uEHM::uRaiseClass : private uEHM::uDualClass {
    uRaiseClass( const char *const msg = "" );
    const char *const message() const;
    const uBaseCoroutine &source() const;
    virtual void defaultResume() const;
};
```

Only the base class uEHM::uDualClass is discussed as the other two exception types are subsets.

The constructor routine uDualClass has the following form:

uDualClass(**const char** *const msg = "") – creates a dual exception with specified message, which is printed in an error message if the exception is not handled. The message is not copied so its storage must persist for the duration of the exception.

The member routine message returns the string message associated with an exception. The member routine source returns the coroutine/task that raised the exception. In some cases, the coroutine or task may be deleted when the exception is caught so this reference may be undefined. The member routine sourceName returns the name of the coroutine/task that raised the exception. This name is copied from the raising coroutine/task when an exception is created so it is safe to use even if the coroutine/task is deleted.

The member routine defaultResume is implicitly called if an exception is resumed but not handled; the default action is to throw the exception, which begins the search for a termination handler from the point of the initial resume for a dual exception-type, or calls uAbort to terminate the program with the supplied message for a resume exception-type. The member routine defaultTerminate is implicitly called if an exception is thrown but not handled; the default action is to call uAbort to terminate the program with the supplied message. In both cases, a user-defined default action may be implemented by overriding the appropriate virtual member.

3.15 Raising

There are two raising mechanisms: throwing and resuming; furthermore, each kind of raising can be done nonlocally or concurrently. The kind of raising for an exception is specified by the raise statement:

```
_Throw [ throwable-exception [ _At uBaseCoroutine-id ] ] ;
_Resume [ resumable-exception [ _At uBaseCoroutine-id ] ] ;
```

If **_Throw** has no *throwable-exception*, it is a rethrow; if **_Resume** has no *resumable-exception*, it is a reraise; The optional **_At** clause allows nonlocal or concurrent exceptions to be raised at another coroutine or task. The kind of exception type (throw, resume, dual) must match with the kind of raise, i.e., **_Throw** or **_Resume**.

Exceptions in μC++ are propagated differently from C++. In C++, the **throw** statement initializes a temporary object, the type of which is determined from the static type of the operand, and propagates the temporary object. In μC++, the **_Throw** and **_Resume** statements throw the actual operand. For example:

C++	μC++
class T1 {}; class T2 : public T1 {}; void f(T1 &t) { throw t; } T2 m; f(m);	_ThrowEvent T1 {}; _ThrowEvent T2 : public T1 {}; void f(T1 &t) { _Throw t; } T2 m; f(m);

in the C++ program, routine f is passed an object of type T2 but throws an object of type T1, because the static type of the operand for throw, t, is of type T1. However, in the μC++ program, routine f is passed an object of type T2 and throws the original object of type T2. This change makes a significant difference in the organization of handlers for dealing with exceptions by allowing handlers to catch the specific rather than the general exception-type.

Note, when subclassing is used, it is better to catch an exception by reference for termination and resumption handlers. Otherwise, the exception is truncated from the actual type to the static type specified at the handler, and cannot be dynamically down-cast to the actual type. Notice, catching-exception truncation is different from propagation-exception truncation, which does not occur in μC++.

3.16 Handler

A handler catches a propagated exception and attempts to deal with the exceptional event. Each handler is in the handler clause of a guarded block. μC++ supports two kinds of handlers, termination and resumption, which match with the kind of exception type (throw, resume, dual). An unhandled exception is dealt with by an exception default-member (see Section 3.14.2).

3.16.1 Termination

A termination handler is a corrective action replacing the action of the guarded block with the handler block. In μC++, a termination handler is specified identically to that in C++: **catch** clause of a **try** statement. (The details of termination handlers can be found in a C++ textbook.) Figure 3.13 shows how C++ and μC++ use a throw exception with throwing propagation and a termination handler. The differences are using **_Throw** instead of **throw**, throwing the actual type instead of the static type, and requiring a special exception type for all exceptions.

3.16.2 Resumption

A resumption handler is a corrective action for a failing operation, or it determines correction is impossible and throws an exception. Effectively, the corrective action is an intervention in the normal computation when a problem is encountered. Unlike normal routine calls, the call to a resumption handler is dynamically bound rather than statically bound, so different corrections can occur for the same static context.

Figure 3.14, p. 58 shows the ideal syntax for specifying resumption handlers on the left, and the compromise syntax provided by μC++ on the right. On the left, the resumption handler is, in effect, a nested routine called when a propagated resume exception is caught by the handler; when the resumption handler completes, control returns back to the point of the raise. Values at the raise point can be modified directly in the handler if variables are visible in both

C++	μC++
<pre>**class** E { **public**: **int** i; E(**int** i) : i(i) {} }; **void** f() { **throw** E(3); } **int** main() { **try** { f(); } **catch**(E e) { cout << e.i << endl; **throw**; } // try }</pre>	<pre>**_ThrowEvent** E { **public**: **int** i; E(**int** i) : i(i) {} }; **void** f() { **_Throw** E(3); } **void** uMain::main() { **try** { f(); } **catch**(E e) { cout << e.i << endl; **_Throw**; } // try }</pre>

Figure 3.13: C++ versus μC++ Throwing Propagation

contexts, or indirectly through reference or pointer parameters; there is no concept of a return value from a resumption handler, as is possible with a normal routine. Unfortunately, C++ has no notion of nested routines, so it is largely impossible to achieve the ideal resumption-handler syntax.

On the right is the simulation of the ideal resumption-handler syntax. The most significant change is the movement of the resumption-handler bodies to routines h1 and H2::**operator**(), respectively. Also, the direct access of local variables x and y in the first resume handler necessitates creating a functor so that h2 can access them.

In detail, μC++ extends the **try** block to set up resumption handlers, where the resumption handler is a routine. Any number of resumption handlers can be associated with a **try** block and there are 2 different forms for specifying a resumption handler:

```
try <E1,h><E2>... {
    // statements to be guarded
} // possible catch clauses
```

The 2 forms of resumption handler are:

1. <E1,h>
2. <E2>

The first form specifies a routine/functor, h, to handle resumed exceptions of type E1. The routine/functor must take the exception type as a reference parameter:

```
void h( E1 & )                 // routine
void H::operator()( E1 & )     // functor
```

This form is the most common, particularly for resumption handlers in library code. The second form specifies resumed exceptions of type E2 are handled by an empty handler. This form eliminates having to create a handler routine with an empty routine body. Type checking is performed on the first form, to ensure a proper handler is specified to handle the designated exception type.

3.17 Inheritance

Table 3.2, p. 59 shows the forms of inheritance allowed among the different kinds of exception types. First, the case of *single* public inheritance among homogeneous kinds of exception type, i.e., base and derived type are the same kind, is supported in μC++ (major diagonal), e.g.:

Ideal Syntax

```
_ResumeEvent R1 {
  public:
    int &i; char &c;
    R1( int &i, char &c ) : i( i ), c( c ) {}
};

_ResumeEvent R2 {};

void f( int x, char y ) { _Resume R2(); }
void g( int &x, char &y ) { _Resume R1( x, y ); }










void uMain::main() {
    try {
        int x = 0;
        char y = 'a';

        g( x, y );

        try {
            f( x, y );
        } resume( R2 ) {
            x = 2; y = 'c'; // modify local variables
        } // try
        try {
            g( x, y );
        } resume( R1 ) { // ignore R1
        } // try
    } resume( R1 &r ) {
        // cannot see variables x and y
        r.i = 1; r.c = 'b'; // modify arguments
    } // try
}
```

Actual μC++ Syntax

```
_ResumeEvent R1 {
  public:
    int &i; char &c;
    R1( int &i, char &c ) : i( i ), c( c ) {}
};

_ResumeEvent R2 {};

void f( int x, char y ) { _Resume R2(); }
void g( int &x, char &y ) { _Resume R1( x, y ); }

void h1( R1 &r ) { r.i = 1; r.c = 'b'; }
struct H2 { // functor
    int &i; char &c;
    H2( int &i, char &c ) : i( i ), c( c ) {}
    void operator()( R2 & r ) { // required
        i = 2; c = 'c';
    }
};
void uMain::main() {
    try <R1,h1> {
        int x = 0;
        char y = 'a';

        g( x, y );
        H2 h2( x, y );    // bind to locals
        try <R2,h2> {
            f( x, y );


        } // try
        try <R1> { // ignore R1
            g( x, y );

        } // try



    } // try
}
```

Figure 3.14: Syntax for Resumption Handlers

```
_ThrowEvent TEbase {};
_ThrowEvent TEderived : public TEbase {}; // homogeneous public inheritance

_ResumeEvent REbase {};
_ResumeEvent REderived : public REbase {}; // homogeneous public inheritance

_DualEvent DEbase {};
_DualEvent DEderived  : public DEbase {}; // homogeneous public inheritance
```

In this situation, all implicit functionality matches between base and derived types, and therefore, there are no problems. Public derivation of exception types is for building the three exception-type hierarchies, and restricting public inheritance to only exception types enhances the distinction between the class and exception hierarchies. Single private/protected inheritance among homogeneous kinds of exception types is not supported, e.g.:

```
_ThrowEvent TEderived : private TEbase {};     // homogeneous private inheritance, not allowed
_ThrowEvent TEderived : protected TEbase {};   // homogeneous protected inheritance, not allowed
```

because each exception type must appear in one of the three exception-type hierarchies (throw, resume, dual), and hence must be a subtype of another exception type of the same kind. Neither **private** nor **protected** inheritance establishes a subtyping relationship.

base / derived	struct/class	*public only / NO multiple inheritance* throw	raise	dual
struct/class	√	X	X	X
throw	√	√	X	X
raise	√	X	√	X
dual	√	X	X	√

Table 3.2: Inheritance among Exception Types

Second, the case of *single* private/protected/public inheritance among heterogeneous kinds of type, i.e., base and derived type of different kind, is supported in μC++ only if the base kind is an ordinary class, e.g.:

```
class cbase {};                               // only class kind allowed

_ThrowEvent TEderived    : private cbase {};   // heterogeneous private inheritance
_ResumeEvent REderived   : protected cbase{};  // heterogeneous protected inheritance
_DualEvent DEderived     : public cbase {};    // heterogeneous public inheritance
```

An example for using such inheritance is different exception types using a common logging class. The ordinary class implements the logging functionality and can be reused among the different exception types.

Heterogeneous inheritance among exception types and other kinds of class, exception types, coroutine, mutex or task, are not allowed, e.g.:

```
_ThrowEvent TEbase {};

struct structDerived        : public TEbase {};  // not allowed
class classDerived          : public TEbase {};  // not allowed
_ResumeEvent REderived      : public TEbase {};  // not allowed
_Coroutine corDerived       : public TEbase {};  // not allowed
_Monitor monitorDerived : public TEbase {}; // not allowed
_Task taskDerived           : public TEbase {};  // not allowed
```

A structure/class cannot inherit from an exception type because operations defined for exception types may cause problems when accessed through a class object. This restriction does not mean exception types and non-exception-types cannot share code. Rather, shared code must be factored out as an ordinary class and then inherited by exception types and non-exception-types, e.g.:

```
class commonBase {};

class classDerived          : public commonBase {};
_ResumeEvent REderived      : public commonBase {};
```

Technically, it is possible for exception types to inherit from mutex, coroutine, and task types, but logically there does not appear to be a need. Exception types do not need mutual exclusion because a new exception is generated at each throw, so the exception is not a shared resource. For example, arithmetic overflow can be encountered by different executions but each arithmetic overflow is independent. Hence, there is no race condition for exception types. Finally, exception types do not need context switching or a thread to carry out computation. Consequently, any form of inheritance from a mutex, coroutine or task by an exception type is rejected.

Multiple inheritance is allowed for private/protected/public inheritance of exception types with **struct/class** for the same reason as single inheritance.

3.18 Predefined Exception-Types

μC++ provides a number of predefined exception-types, which are structured into the hierarchy in Figure 3.15. Notice that all predefined exception-types are dual so that they can be both resumed or thrown. Also, the predefined exception-types are divided into two major groups: kernel and I/O. The kernel exception-types are raised by the μC++ runtime kernel when problems arise using the μC++ concurrency extensions. The I/O exception-types are raised by the μC++ I/O library when problems arise using the file system. Only the kernel exception-types are discussed, as the I/O exception-types are OS specific.

3.18.1 Implicitly Enabled Exception-Types

Certain of the predefined kernel exception-types are implicitly enabled in certain contexts to ensure prompt delivery for nonlocal exceptions. The predefined exception-type uBaseCoroutine::Failure is implicitly enabled and polling is performed when a coroutine restarts after a suspend or resume. The predefined exception-type uSerial::Failure is implicitly enabled and polling is performed when a task restarts from blocking on entry to a mutex member. This situation also occurs when a task restarts after being accept blocked on an **_Accept** or a wait. The predefined exception-type uSerial::RendezvousFailure is implicitly enabled and polling is performed when an acceptor task restarts after blocking for a rendezvous to finish.

3.19 Summary

Static and dynamic name binding, and static and dynamic transfer points can be combined to form the following different language constructs:

	call/raise	
return/handled	static	dynamic
static	1. sequel	3. termination
dynamic	2. routine	4. resumption

These four constructs succinctly cover all the kinds of control flow associated with routines and exceptions. Raising, propagating and handling an exception are the three core control-flow mechanisms of an EHM. There are two useful handling models: termination and resumption. For safety, an EHM should provide matching propagation mechanisms: terminating and resuming. Handlers should be partitioned with respect to the handling models to provide better abstraction. Consequential propagation solves the recursive resuming problem and provides consistent propagation semantics with termination, making it the best choice for an EHM with resumption. Homogeneous derivation of exception types, catch-any and reraise, exception parameters, and bound/conditional handling all improve programmability and extensibility.

There is a weak equivalence between multi-exit and static multi-level, and exceptions. Suffice it to say that exceptions can only be simulated with great difficulty with simpler control structures. The simulation usually involves routines returning return-codes and complex return-code checking after routine calls, which makes the program extremely difficult to understand and maintain. But most importantly, return code checking is optional, and hence, leaves open the possibly for programmer error. On the other hand, it is fairly easy to simulate both multi-exit and static multi-level exit with dynamic multi-level exit. However, dynamic multi-level exit is usually more costly than multi-exit and static multi-level exit. This cost appears in several different forms, such as increased compilation cost, larger executable programs, and/or increased execution time. Several implementation schemes exist for zero-cost entry of the handler construct (**try** construct in C++), but this should not be confused with the total cost of implementing exceptions. Thus, there is strong justification for having both a static and dynamic multi-level exit construct in a programming language.

3.20 Questions

1. Convert the program in Figure 3.1, p. 22 from dynamic multi-level exits to one using only multi-exit and static multi-level exit. The return type of routine rtn may be changed.

2. Convert the C++ program:

```
uEHM::uDualClass
    uKernelFailure
        uSerial::Failure
            uSerial::EntryFailure
            uSerial::RendezvousFailure
            uCondition::WaitingFailure
        uBaseCoroutine::Failure
            uBaseCoroutine::UnHandledException
    uIOFailure
        uFile::Failure
            uFile::TerminateFailure
            uFile::StatusFailure
            uFileAccess::Failure
                uFileAccess::OpenFailure
                uFileAccess::CloseFailure
                uFileAccess::SeekFailure
                uFileAccess::SyncFailure
                uFileAccess::ReadFailure
                    uFileAccess::ReadTimeout
                uFileAccess::WriteFailure
                    uFileAccess::WriteTimeout
        uSocket::Failure
            uSocket::OpenFailure
            uSocket::CloseFailure
            uSocketServer::Failure
                uSocketServer::OpenFailure
                uSocketServer::CloseFailure
                uSocketServer::ReadFailure
                    uSocketServer::ReadTimeout
                uSocketServer::WriteFailure
                    uSocketServer::WriteTimeout
            uSocketAccept::Failure
                uSocketAccept::OpenFailure
                    uSocketAccept::uOpenTimeout
                uSocketAccept::CloseFailure
                uSocketAccept::ReadFailure
                    uSocketAccept::ReadTimeout
                uSocketAccept::WriteFailure
                    uSocketAccept::WriteTimeout
            uSocketClient::Failure
                uSocketClient::OpenFailure
                    uSocketClient::OpenTimeout
                uSocketClient::CloseFailure
                uSocketClient::ReadFailure
                    uSocketClient::ReadTimeout
                uSocketClient::WriteFailure
                    uSocketClient::WriteTimeout
uEHM::uThrowClass
    // no predefines
uEHM::uRaiseClass
    // no predefines
```

Figure 3.15: μC++ Predefined Exception-Type Hierarchy

```
#include <iostream>
using namespace std;

int myfixup1( int i ) { return i + 2; }
int myfixup2( int i ) { return i + 1; }

int rtn2( int p, int (*fixup)( int ) ); // forward declaration

int rtn1( int p, int (*fixup)( int ) ) {
  if ( p <= 0 ) return 0;        // base case
    if ( p % 2 ) {
        p = rtn2( p - 1, myfixup1 );
    } else {
        p = rtn1( p - 2, myfixup2 );
    }
    if ( p % 3 ) p = fixup( p );
    cout << p << "   ";
    return p + 1;
}

int rtn2( int p, int (*fixup)( int ) ) {
  if ( p <= 0 ) return 0;        // base case
    if ( p % 3 ) {
        p = rtn2( p - 2, myfixup1 );
    } else {
        p = rtn1( p - 1, myfixup2 );
    }
    if ( p % 2 ) p = fixup( p );
    cout << p << "   ";
    return p + 2;
}
int main() {
    cout << rtn2( 30, myfixup1 ) << endl;
}
```

from using local fix-up routines to one using only a single global fixup pointer, e.g.:

```
int (*fixup)( int );
```

You are allowed to make local copies of the global fixup pointer.

3. In languages without resumption exceptions, resumption can be simulated by explicitly passing handler routines as arguments (often called fix-up routines), which are called where the resumption exception is raised. Given the following μC++ program:

```
_ResumeEvent R1 {
  public:
    int &i;
    R1( int &i ) : i(i) {}
};

_ResumeEvent R2 {
  public:
    int &i;
    R2( int &i ) : i(i) {}
};

void f( int &i );
void g( int &i );
```

```
void h1( R1 &r ) {
    r.i -= 1;
    cout << "h1, i:" << r.i << endl;
    f( r.i );
}
void h2( R2 &r ) {
    r.i -= 1;
    cout << "h2, i:" << r.i << endl;
    g( r.i );
}
void h3( R2 &r ) {
    r.i -= 1;
    cout << "h3, i:" << r.i << endl;
}

void f( int &i ) {
    i -= 1;
    cout << "f, i:" << i << endl;
    if ( i > 5 ) {
        try <R1,h1> {
            if ( i < 10 ) _Resume R2(i);
            try <R2,h2> {
                g( i );
            }
        }
    } else {
        if ( i > -5 ) _Resume R2(i);
    } // if
}

void g( int &i ) {
    i -= 1;
    cout << "g, i:" << i << endl;
    if ( i > 10 ) {
        try <R2,h2> {
            f( i );
        }
        try <R2,h3> {
            f( i );
        }
        _Resume R1(i);
    }
}
void uMain::main() {
    int i = 20;
    f( i );
}
```

(a) Explain why this program *cannot* be simulated by passing handler (fix-up) routines among the routines.

(b) Construct a simulation of the program in C++ by passing handler functors rather than handler routines among the routines. (Hint: some inheritance is required.)

Chapter 4

Coroutine

As mentioned in Section 1.1, p. 3, a routine call cannot be constructed out of basic control flow constructs. Therefore, a routine call is a fundamental control flow mechanism. What characterizes a routine is that it always starts execution from the beginning (top), executes until it returns normally or abnormally (exception), and its local variables only persist for a single invocation. A routine temporarily suspends its execution when it calls another routine, which saves the local state of the caller and reactivates it when the called routine returns. But a routine cannot temporarily suspend execution and return to *its* caller. Interestingly, there exist problems where a routine needs to retain state, both data and execution location, *between* routine calls. Basically, a routine implementing these kinds of problems needs to remember something about what it was doing the last time it was called. These kinds of problems cannot be adequately implemented using normal routines. Two simple examples are presented to illustrate the kinds of situations where a routine needs to retain both data and execution state between calls.

4.1 Fibonacci Series

A series generator, like the Fibonacci series, must remember prior calculations to generate the next value in the series. To demonstrate this situation, the Fibonacci series is used and is defined as follows:

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & \text{otherwise} \end{cases}$$

producing the sequence of numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, etc. Ignoring the problem of stopping, a direct solution for producing the Fibonacci series is:

```
int fn, fn1, fn2;
fn = 0;  fn1 = fn;                      // 1st case
cout << fn << ", ";
fn = 1;  fn2 = fn1;  fn1 = fn;          // 2nd case
cout << fn << ", ";
for ( ;; ) {
    fn = fn1 + fn2;  fn2 = fn1;  fn1 = fn;   // general case
    cout << fn << ", ";
}
```

The difficulty comes in modularizing this code into a fibonacci routine, so that each time the routine is called, it produces the next Fibonacci number in the sequence. This abstraction provides a Fibonacci generator, which can be placed in a library with other series generators for use by math or statistics programmers. Notice the modularization moves the output of each series calculation from external, i.e., to a file, to internal, i.e., the return value from the generator routine. The main problem occurs when the fibonacci routine returns after each call—it forgets where it is in the series. Several possible solutions are presented to illustrate the problems in implementing the Fibonacci generator.

```
int fn, fn1, fn2, state = 1;           // global variables

int fibonacci() {
    switch (state) {
      case 1:
        fn = 0;  fn1 = fn;
        state = 2;
        break;
      case 2:
        fn = 1;  fn2 = fn1;  fn1 = fn;
        state = 3;
        break;
      case 3:
        fn = fn1 + fn2;  fn2 = fn1;  fn1 = fn;
        break;
    }
    return fn;
}
int main() {
    for ( int i = 1; i <= 10; i += 1 ) {
        cout << fibonacci() << endl;
    }
}
```

Figure 4.1: Fibonacci, Shared Variables and Explicit State

4.1.1 Routine Solution

The first solution in Figure 4.1 uses global state to remember information between routine calls. The main routine calls fibonacci to get the next number in the Fibonacci series, which is then printed. Since the routine fibonacci cannot remember information between calls, information must be remembered in global variables, which are accessed in the fibonacci routine. The global variables can be divided into two kinds: those holding data used in calculations and those used to control execution during each call (flag variables as discussed in Section 2.2, p. 8). In this case, the variables fn, fn1 and fn2 hold data while the variable, state, holds control-flow information. The purpose of the data variables is the same as in the initial, simple solution; it is the control flow variable, state, that is new and requires explanation. The definition of fibonacci has three states: two special cases to start the third general case. Each of these three states is denoted by the three values taken on by variable state. When fibonacci is called, it checks the value of state to determine what code should be executed. After a block of code is executed, state is changed to indicate the next block of code executed on a subsequent call to fibonacci. Thus, state is initialized to the starting state, and after two calls, state no longer changes because the general case can be executed from that point onwards.

While this program works, it has problems. First, the use of global variables violates the encapsulation of the generator because the variables can be accessed or changed by users. (C aficionados would move the global variables into fibonacci with storage class **static** but that does not solve the remaining problems.) Second, because there is only one set of global (or **static**) variables for fibonacci, there can only be one sequence of Fibonacci numbers generated in a program. It is possible to imagine situations where multiple sequences of Fibonacci numbers have to be simultaneously generated in a single program. Third, explicitly managing the execution control information is tedious and error-prone. For a simple program like fibonacci, the number of states is small and manageable; however, for complex programs with many states, flattening the program structure into sequential blocks of code controlled by execution-state variables can completely hide the fundamental algorithm. Structuring a program into this flattened form corresponds to having only one controlling loop (possibly for the entire program), and each time through the loop, flag variables and **if** statements specify different blocks of code to execute. In fact, any program can be flattened and any flattened program can be written with multiple loops; hence, there is a weak equivalence between both forms. Nevertheless, writing a flattened program is not recommended (unless forced to) because, in general, it obscures the algorithm, making the program difficult to understand and maintain.

```
class fibonacci {
    int fn, fn1, fn2, state;        // class variables
  public:
    fibonacci() : state(1) {}
    int next() {
        switch (state) {
          case 1:
            fn = 0;  fn1 = fn;
            state = 2;
            break;
          case 2:
            fn = 1;  fn2 = fn1;  fn1 = fn;
            state = 3;
            break;
          case 3:
            fn = fn1 + fn2;  fn2 = fn1;  fn1 = fn;
            break;
        }
        return fn;
    }
};
int main() {
    fibonacci f1, f2;
    for ( int i = 1; i <= 10; i += 1 ) {
        cout << f1.next() << " " << f2.next() << endl;
    }
}
```

Figure 4.2: Fibonacci, Explicit Execution State

4.1.2 Class Solution

The problems of global variables and allowing multiple Fibonacci generators can be handled by creating fibonacci objects from a class definition, as in Figure 4.2. (Alternatively, a structure containing the class state can be explicitly passed to the fibonacci routine.) The global variables from the routine fibonacci are made class variables in the fibonacci class. It is now possible to generate multiple Fibonacci generators by instantiating multiple instances of the class, each with its own set of variables to generate a sequence of Fibonacci numbers. The member next is invoked to obtain each number in the sequence, and it uses the class variables of the object to retain state between invocations. Note, a member routine cannot retain state between calls any more than a routine can, so some form of global variable is necessary. The main program can now create two Fibonacci generators and print out two sequences of Fibonacci numbers. Notice the class solution still requires explicit management of the execution-state information so no improvement has occurred there.

4.1.3 Coroutine Solution

To remove the execution control information requires a coroutine. The first description of a **coroutine** was:

> ... an autonomous program which communicates with adjacent modules as if they were input or output subroutines. Thus, coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program. [Con63, p. 396]

and the coroutine concept was developed in parallel by Melvin E. Conway and Joel Erdwinn. A Fibonacci coroutine-solution cannot be presented in C++, as it does not support coroutines; hence, it is necessary to switch to μC++. Only a short explanation of the μC++ coroutine is presented at this time, so as not to interrupt the Fibonacci example. A detailed description of a μC++ coroutine is presented later in Section 4.6, p. 80.

A μC++ coroutine type has all the properties of a **class**. The general form of the coroutine type is the following:

```
_Coroutine coroutine-name {
      . . .                  // implementation variables and members
      void main();           // coroutine main (distinguished member)
   public:
      . . .                  // interface members
};
```

A coroutine is an object created from a coroutine type in which execution of its distinguished member routine can be suspended and resumed; this distinguished member is named main and is called the **coroutine main**. Execution of a coroutine main may be suspended as control leaves it, only to carry on from that point when control returns at some later time. Hence, a coroutine main is not restarted at the beginning on each activation and its local variables are preserved while it is suspended. It is distinguished because it has special properties that none of the other member routines have and only the member routine called main in a μC++ coroutine has special properties. (In a normal C++ program, the routine main is a distinguished routine because the program begins execution there.)

A coroutine main has its own **execution state**, which is the state information needed to retain information while the coroutine main is suspended. An execution state is either **active** or **inactive**, depending on whether or not it is currently being executed. In practice, an execution state consists of the data items created by a coroutine, including its local data and routine activations, and a current execution location, which is initialized to a starting point (member main). The routine activations are often maintained in a contiguous stack, which constitutes the bulk of an execution state and is dynamic in size. This stack is the area where the local variables and execution location are preserved when an execution state is inactive. When flow of control transfers from one execution state to another, it is called a **context switch**, which involves saving the state of the active coroutine, making it inactive, and restoring the inactive state of the next coroutine, making it active. In general, the overhead for a context switch is 2–3 times more expensive than the overhead for a routine call (i.e., excluding execution of the code within the routine).

Normally, a coroutine is only activated indirectly through its interface members, which directly interact with the coroutine's main; hence, the visibility of main is usually **private** or **protected**. The decision to make the coroutine main **private** or **protected** depends solely on whether derived classes can reuse the coroutine main or must supply their own. This structure allows a coroutine type to have multiple public member routines to service different kinds of requests that are statically type-checked. A coroutine main cannot have parameters or return a result, but the same effect can be accomplished indirectly by passing values through the coroutine's variables, called **communication variables**, which are accessible from both the coroutine's member and main routines. Like a routine or class, a coroutine can access all the external variables of a C++ program and variables allocated in the heap. Also, any **static** member variables declared within a coroutine are shared among all instances of that coroutine type.

Figure 4.3 shows the Fibonacci coroutine and a driver routine, as well as a diagram of the basic control flow in a fibonacci coroutine. The general structure is similar to the class solution but with the following differences. The type for fibonacci is now **_Coroutine** instead of **class**. The code from the previous fibonacci::next member is now in private member fibonacci::main, which is the "distinguished member routine" (coroutine main) referred to in the coroutine definition. The program main routine, i.e., where the program begins, is the same as before, except that it is now a member of a special anonymous coroutine with type uMain (explained in more detail in Chapter 5, p. 115).

The next member makes a call to a special coroutine member, resume, which causes control to start execution at the beginning of member main the first time it is executed, and thereafter, restarts the coroutine main at its point of last suspension. In detail, the first call to next resumes execution of the coroutine main, causing fibonacci::main to start and allocate its local variables fn1 and fn2, and then the communication variable fn and the local variable fn1 are initialized. (This implicit starting of the coroutine main is indicated by the dashed line in Figure 4.3.) Now the other special coroutine member, suspend, is executed, which causes control to leave the coroutine main, without deallocating the local variables fn1 and fn2, and return from the call to resume in fibonacci::next. Whereupon, member next also returns, passing back the value in the communication variable fn, which is the first Fibonacci number. The second call to member next resumes the coroutine main, which returns from the first call to suspend, and it then performs the next calculation, suspends back to the last resume, and returns the next Fibonacci number. The third call of next resumes the coroutine main, which returns from the second suspend, enters the general loop, performs the next general calculation, and suspends back to the last resume with the next Fibonacci number. All subsequent calls to next cause resumption at the third suspend, which then performs the next general calculation, and suspends back to the last resume to return a Fibonacci number. Notice, the Fibonacci algorithm has no explicit control variables to remember where to execute, which is the point behind using a coroutine. Just as a routine knows where to return to when it is called, a coroutine

```
_Coroutine fibonacci {
    int fn;                 // communication variable
    void main() {           // distinguished member
        int fn1, fn2;       // retained between resumes
        fn = 0;  fn1 = fn;
        suspend();          // return to last resume
        fn = 1;  fn2 = fn1;  fn1 = fn;
        suspend();          // return to last resume
        for ( ;; ) {
            fn = fn1 + fn2;  fn2 = fn1;  fn1 = fn;
            suspend();      // return to last resume
        }
    }
  public:
    int next() {
        resume();           // transfer to last suspend
        return fn;
    }
};
void uMain::main() {
    fibonacci f1, f2;       // create 2 generates
    for ( int i = 1; i <= 10; i += 1 ) {
        cout << f1.next() << " "
             << f2.next() << endl;
    }
}
```

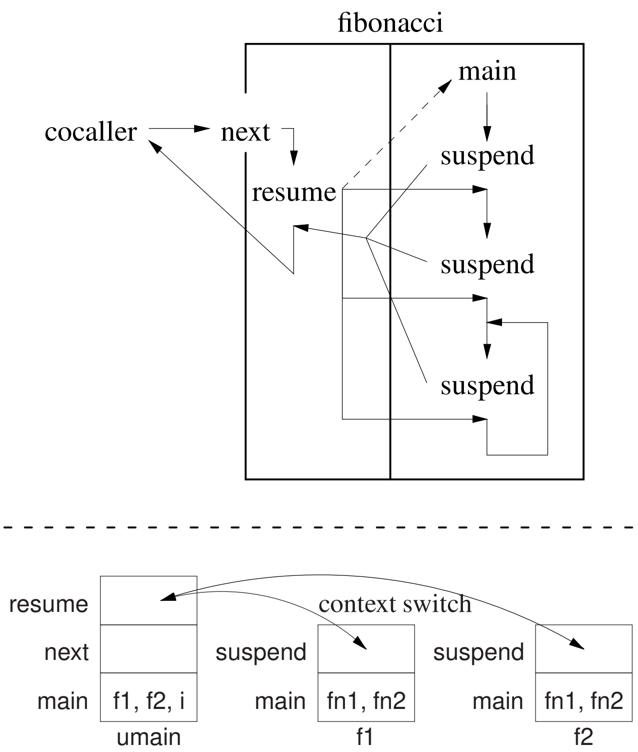


Figure 4.3: Fibonacci, No Explicit Execution State

knows where to restart when it is suspended. This capability means the information explicitly managed by execution-state variables is implicitly managed by the resume/suspend mechanism of a coroutine, allowing direct modularizing of the initial external-output solution in Section 4.1, p. 65 into an internal-output solution in the coroutine main.

There are several additional points to note about programming with coroutines. First, to communicate information between a member routine and the coroutine main requires class variables, like the variable fn. Data is copied from the member routine into the communication variable so that it can be accessed by the coroutine main, and vice versa for returning values. This additional step in communication is the price to be paid for statically type-checked communication. Second, all the variables used to store data between activations of the coroutine main should be declared in the coroutine main, e.g., variables fn1 and fn2. While these variables *could* be defined as class variables, this violates encapsulation of the coroutine main, making the variables appear as if they can be read or written by other members of the coroutine. (In other words, never make a variable more visible than is needed.) Finally, it is normal programming style to delete a coroutine that is suspended. For example, when uMain::main terminates, the two local coroutines, f1 and f2, are automatically deleted even though both are suspended in their coroutine main. Deleting a suspended coroutine is no different than deleting a normal class object filled with values.

4.2 Formatting

A formatting routine for generating specific output may need to remember the prior printing location and values to decide how the next value is formatted and printed. To demonstrate this situation, a formatting program is used that reads characters (ignoring newline characters), formats them into blocks of 4 characters, and groups the blocks of characters into groups of 5:

```
abcd  efgh  ijkl  mnop  qrst
uvwx  yzab  cdef  ghij  klmn
opqr  stuv  wxyz
```

A direct solution for producing the formatted output appears in Figure 4.4.

```
int main() {
    int g, b;
    char ch;
    cin >> noskipws;                        // turn off white space skipping
    for ( ;; ) {                            // for as many characters
        for ( g = 0; g < 5; g += 1 ) {      // groups of 5 blocks
            for ( b = 0; b < 4; b += 1 ) {  // blocks of 4 characters
                for ( ;; ) {                // for newline characters
                    cin >> ch;              // read one character
      if ( cin.eof() ) goto fini;           // eof ? multi-level exit
                  if ( ch != '\n' ) break;  // ignore newline characters
                }
                cout << ch;                 // print character
            }
            cout << "  ";                   // print block separator
        }
        cout << endl;                       // print group separator
    }
  fini: ;
    if ( g != 0 || b != 0 ) cout << endl;   // special case
}
```

Figure 4.4: Output Formatter: direct

```
class FmtLines {
    int g, b;
  public:
    void prt( char ch ) {
      if ( ch == '\n' ) return;
        cout << ch;
        b += 1;
        if ( b == 4 ) {    // blocks of 4
            cout << "  "; // block separator
            b = 0;
            g += 1;
        }
        if ( g == 5 ) {    // groups of 5
            cout << endl; // group separator
            g = 0;
        }
    }


    FmtLines() : g(0), b(0) {}


    ~FmtLines() {
        if ( g != 0 || b != 0 ) cout << endl;
    }
};
```

(a) class

```
_Coroutine FmtLines {
    char ch;
    int g, b;
    void main() {
        for ( ;; ) {        // for as many characters
            for ( g = 0; g < 5; g += 1 ) { // groups of 5
                for ( b = 0; b < 4; b += 1 ) { // blocks of 4
                    for ( ;; ) {
                        suspend();
                      if ( ch != '\n' ) break;
                    }
                    cout << ch;
                }
                cout << "  "; // block separator
            }
            cout << endl; // group separator
        }
    }
  public:
    FmtLines() {
        resume();    // eliminate priming read effect
    }
    ~FmtLines() {
        if ( g != 0 || b != 0 ) cout << endl;
    }
    void prt( char ch ) {
        FmtLines::ch = ch;
        resume();
    }
};
```

(b) coroutine

Figure 4.5: Output Formatters

As for the Fibonacci program, the difficulty comes in modularizing this code into a routine so that each time the routine is called with a character, it produces the correct formatted output. Both data and execution state need to be retained between successive calls, i.e., the values of variables b and g, and the locations within nested **for** loops. Figure 4.5(a) shows the class version that accepts single characters and formats them. The prt member of class FmtLines is an example of flattening the structure of an algorithm (the loop is in the program's main routine reading characters). There are really three nested loops in prt that have been flattened into assignments and **if** statements.

Figure 4.5(b) shows the equivalent coroutine solution. Like the Fibonacci coroutine, the code in member prt is moved into the coroutine main. However, the structure of the algorithm is now coded as two nested loops, plus an outer infinite loop to handle any number of characters. Notice the structure of this program is almost the same as the initial solution in Figure 4.4. Think about how the formatter programs would be altered to print a row of asterisks after every 10 lines of output. The program in Figure 4.5(a) must continue to flatten the control flow, while the program in Figure 4.5(b) can use the direct equivalent to a modified version of Figure 4.4

A new technique is used in the coroutine formatter to obtain the same structure as the version in Figure 4.4. Note the resume in the constructor of FmtLines. This resume allows the coroutine main to start *before* any members are called. For the formatter, all the loops are started before the coroutine main suspends back to finish construction. At this point, no character has been passed to the coroutine to be formatted. When the next resume occurs from member prt, the first character is available *after* the suspend, just like reading from a file generates the first character *after* the first input operation. This subtle change eliminates the equivalent priming read. Without this technique, the coroutine main does not start until the first resume in prt, so the coroutine has to initialize itself *and* process a character. If this priming-read effect is retained (by removing the resume from the constructor), the innermost looping code of the formatter's coroutine main must be changed to:

```
for ( ;; ) {
  if ( ch != '\n' ) break;
    suspend();
}
cout << ch;
suspend();
```

Here the suspends appear *after* processing of a character rather than *before*. If the program in Figure 4.4 is written using a priming read, then this form of the coroutine would be its equivalent. Both forms are valid and depend on the specific problem and programmer.

4.3 Writing a Coroutine

Notice for both the Fibonacci and the formatter programs, the coroutine directly expresses the structure of the original algorithm, rather than restructuring the algorithm to fit the available control structures. In fact, one of the simplest ways to write a coroutine is to first write (and test) a stand-alone program that follows the basic structure of reading data, processing the data, and writing the result. This program can be converted into a coroutine by putting the code for processing the data into the coroutine main, replacing the reads and/or writes with calls to suspend, and providing communication variables and interface members to transfer data in/out of the coroutine. The decision about which of the reads or writes to convert depends on whether the program is consuming or producing. For example, the Fibonacci program consumes nothing and produces (generates) Fibonacci numbers, so the writes are converted to suspends; the formatter program directly consumes characters and only indirectly produces output (as a side-effect), so the reads are converted to suspends.

4.3.1 Correct Coroutine Usage

In general, new users of coroutines have difficulty using the ability of the coroutine to remember execution state. As a result, unnecessary computation is performed to determine execution location or unnecessary flag variables appear containing explicit information about execution state. An example of unnecessary computation occurs in this coroutine, which is passed each digit of a 10-digit number and sums up the even and odd digits separately:

Explicit Execution State	Implicit Execution State
`for ( int i = 0; i < 10; i += 1 ) {` `    if ( i % 2 == 0 )` `        even += digit;` `    else` `        odd += digit;` `    suspend();` `}`	`for ( int i = 0; i < 5; i += 1 ) {` `    even += digit;` `    suspend();` `    odd += digit;` `    suspend();` `}`

The left code fragment uses explicit execution-state by testing the loop counter for even or odd values to determine which block of code to execute in the loop body. The right code fragment reduces the loop iterations to 5 instead of 10, and suspends twice for each iteration, once for an even and once for an odd digit. While the difference between these examples may appear trivial, the right example illustrates thinking in the implicit execution-state style versus explicit, and hence, takes full advantage of the coroutine.

An example of unnecessary flag variables occurs in this coroutine, which is based on the Fibonacci generator presented in Section 4.1.2, p. 67:

```
void main() {
    int fn1, fn2, state = 1;
    for ( ;; ) {
        switch (state) {
          case 1:
            fn = 0;  fn1 = fn;
            state = 2;
            break;
          case 2:
            fn = 1;  fn2 = fn1;  fn1 = fn;
            state = 3;
            break;
          case 3:
            fn = fn1 + fn2;  fn2 = fn1;  fn1 = fn;
            break;
        }
        suspend();
    }
}
```

This coroutine main uses explicit flag variables to control execution state and a single suspend at the end of an enclosing loop. None of the coroutine's capabilities for remembering execution state are used, and the structure of the program is lost by the loop flattening. While the presence of resume/suspend uses the coroutine main, it is necessary to do more than just *activate* the coroutine main to demonstrate an understanding of retaining data and execution state within a coroutine.

4.4 Parsing

The general concept of parsing, which is important in many problem areas, is another example of needing to retain data and execution state between calls. For example, a tokenizer routine reads characters and combines them into tokens, e.g., identifiers, numbers, punctuation, etc., and returns each token. The tokenizer may need to retain state between calls to handle multi-part tokens and tokens that are sensitive to context. Another example is a parser routine that is passed a sequence of individual characters to determine if a string of characters matches some pattern. The parser routine may also need to retain state between calls to handle complex inter-relationships among the characters of the pattern. Both of these kinds of problems are amenable to solutions using a coroutine because of its ability to retain both data and execution state between calls. Interestingly, there is a significant body of work covering different kinds of parsing; this section discusses how traditional parsing approaches relate to the coroutine.

The simplest traditional mechanism for parsing is the finite automaton (FA):

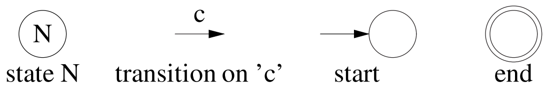
> The **finite automaton** is a mathematical model of a system, with discrete inputs and outputs. The system can be in any of a finite number of internal configurations or "states." The state of the system summa-

rizes the information concerning past inputs that is needed to determine the behavior of the system on subsequent inputs. [HU79, p. 13]

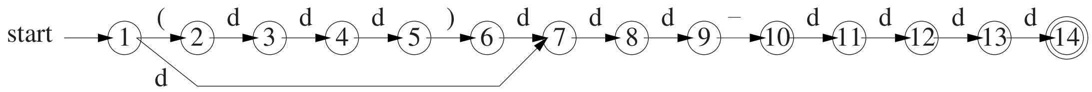
Essentially, an FA is a simple computer consisting of:

- a finite set of states,
- a set of transitions among states, labelled with characters from a fixed alphabet,
- start and end transitions.

An FA can be represented as a directed graph, called a **transition diagram**, using the following symbols:



The meaning of each graph element is illustrated using an example: the transition diagram representing the FA for a North-American phone-number is (where 'd' is any decimal digit):

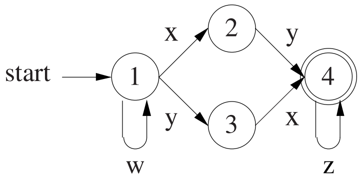


which matches the finite set of phone numbers, such as:

```
(555)123-4567
123-4567
```

The start transition does not involve a character; rather it is a pseudo-transition to begin the FA. State 1 is entered from the start transition, and if the first character is a '(', the FA transitions to state 2 or if the first character is a digit, 0–9, the FA transitions to state 7 (skipping the area code). The remaining transitions are all based on the following characters. The end state indicates a complete phone number.

In general, a state may have multiple transitions emanating from it and may cycle back to any state, including itself. For example:



matches an infinite set of strings, such as:

```
wxyz
wwyxz
wxyzzzzz
wwwwxyzzz
```

Multiple emanating transitions provide different choices for the next character, while cycles allow repetition of one or more character sequences. There can be multiple end states.

An FA with the property that for any given state and character there is exactly one unique transition is called a **deterministic FA** (DFA). Any DFA can be represented by a **transition table**. For example, the FA for the previous North-American phone-number is a DFA with the following transition table (empty boxes represent error states):

	State													
Character	1	2	3	4	5	6	7	8	9	10	11	12	13	14
(	2													
d(igit)	7	3	4	5		7	8	9		11	12	13	14	
)					6									
-									10					

If the FA transitions to an error state, it cannot accept the character, indicating the string of characters is unmatchable by the FA. As well, a transition involving a character not in the transition table is equivalent to entering an error state. It is possible to build a universal scanner for any transition table to determine if a string is recognized by a DFA:

```
state = 1;
for ( ;; ) {
    cin >> ch;                        // read characters one at a time
  if ( cin.eof() ) break;             // complete scan ? (no more characters)
    state = transTable[ch][state];    // use transition table to determine next action
  if ( state == ERROR ) break;        // invalid state ?
}
cout << (state is an end-state ? "" : "no ") << "match" << endl;
```

Tools exists to automatically create a transition table to implement an FA, e.g., UNIX lex.

If such a tool is unavailable or inappropriate, it is possible to directly convert an FA into a program. Figure 4.6 shows two direct conversions of the FA for a phone number. Figure 4.6(a) is a stand-alone program reading characters one at a time (external input) and checking if the string of characters is a phone number. Figure 4.6(b) modularizes the stand-alone program into a coroutine, where the characters of the phone number are passed one at a time to the next member (internal input), which returns the current status of the scan. Notice how the coroutine main retains its execution location and restarts there when it is resumed, e.g., when parsing groups of digits, the coroutine suspends in the middle of a **for** loop, which counts the number of digits, and restarts within the particular loop when resumed. In both cases, the handling of characters after a phone number is accepted is problem specific. The program/coroutine can continue accepting characters and ignore them; alternatively, the program/coroutine can just end. As well, certain practical parsing situation need to know the "end-of-string", which can be denoted with a special character like newline. This case can be represented in an FA by adding the "end-of-string" character and associated states.

A convenient mechanism to describe an FA in a more compact form is the **regular expression**, which has 3 basic operations (where x and y can be a character or regular expression):

- concatenation: x y meaning x *and* y,
- alternation: x | y meaning x *or* y,
- (Kleene) closure: x^* meaning 0 or more concatenations of x.

Parenthesis may be used for grouping. For example, the expression, (′x′ | (′y′ ′z′))*, specifies the infinite set of strings containing zero or more ′x′ characters or ′yz′ strings, such as:

```
x
yzyz
xxxyz
yzyzxxxyzxxx
```

Additional operations are sometimes added:

- positive closure: x^+ meaning 1 or more concatenations of x,
- enumeration: x^n meaning n concatenations of x.

These additional operators can be derived from the basic operations, although awkwardly, because regular expressions cannot directly count or remember an arbitrary amount of characters (no auxiliary variables).

The regular expression for the North-American phone-number is:

```
d = ′0′ | ′1′ | ′2′ | ′3′ | ′4′ | ′5′ | ′6′ | ′7′ | ′8′ | ′9′
phoneno = ( ′(′ d³ ′)′ d³ ′-′ d⁴ )  |  ( d³ ′-′ d⁴ )
```

Notice the regular-expression syntax is a simpler representation than the corresponding FA. The following are general guidelines for converting a regular expression to a direct program (see Figure 4.6 for examples):

- Concatenation is performed by a sequence of checks.
- Alternation is performed by a disjunction of checks using an **if** statement.
- Closure and enumeration are performed by looping.

In fact, the coroutine is more powerful than an FA, and can use mechanisms like counting and auxiliary variables to simplify recognition of strings for a DFA. Hence, it is often unnecessary and inefficient to transform every aspect of an FA directly into a coroutine, i.e., state-for-state translation can result in the poor stylistic approaches discussed in Section 4.3.1, p. 71. Often the transformation from a regular expression for an FA to a coroutine is direct and efficient. Using counting and auxiliary variables in the coroutine can further simplify aspects of the conversion.

An FA that can make nondeterministic choices for accepting input characters is called a **nondeterministic finite automaton** (NFA). For example, consider an NFA described by the regular expression (′x′ | ′y′)* ′x′ ′x′, which is a string of zero or more x or y characters ending with two consecutive x characters, e.g.:

```
enum status { GOOD, BAD };

char ch;
status stat = BAD;

fini: {
    cin >> ch;
    if ( ch == '(' ) {
        for ( i = 0; i < 3; i += 1 ) {
            cin >> ch;
            if ( ! isdigit(ch) ) { break fini; }
        } // for
        cin >> ch;
        if ( ch != ')' ) { break fini; }
        cin >> ch;
    } // if
    for ( i = 0; i < 3; i += 1 ) {
        if ( ! isdigit(ch) ) { break fini; }
        cin >> ch;
    } // for
    if ( ch != '-' ) { stat = BAD; break fini; }
    for ( i = 0; i < 4; i += 1 ) {
        cin >> ch;
        if ( ! isdigit(ch) ) { break fini; }
    } // for
    stat = GOOD;
    // more characters ?
} // block
cout << endl << "status:" << stat << endl;
```

(a) stand-alone

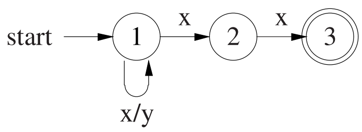
```
_Coroutine PhoneNo {
  public:
    enum status { GOOD, BAD };
  private:
    char ch;          // character passed by cocaller
    status stat;      // current status of match
    void main() {
        int i;
        stat = BAD;
        if ( ch == '(' ) {              // optional area code ?
            for ( i = 0; i < 3; i += 1 ) {  // 3 digits
                suspend();              // get digit
                if ( ! isdigit(ch) ) { return; }
            } // for
            suspend();                  // get ')'
            if ( ch != ')' ) { return; } // ')' ?
            suspend();                  // get digit
        } // if
        for ( i = 0; i < 3; i += 1 ) {  // 3 digits
            if ( ! isdigit(ch) ) { return; }
            suspend();                  // get digit or '-'
        } // for
        if ( ch != '-' ) { return; }    // '-' ?
        for ( i = 0; i < 4; i += 1 ) {  // 4 digits
            suspend();                  // get digit
            if ( ! isdigit(ch) ) { return; }
        } // for
        stat = GOOD;
        // more characters ?
    }
  public:
    status next( char c ) {
        ch = c;
        resume();
        return stat;
    }
};
```

(b) coroutine

Figure 4.6: Direct FA Conversion : Phone Number

```
xx
yxx
xxx
xyxxyyxx
```

The transition diagram and table for an NFA accepting these strings are:

NFA 

	State		
Character	1	2	3
x	1, 2	3	
y	1		

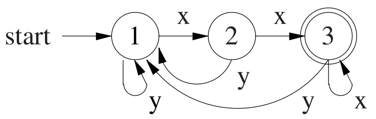
Notice in state 1, the character x can return back to state 1 or transition to state 2. Hence, there is no unique transition at this point in the NFA, which can be seen in the transition table by the two possible states for character x for state 1. Therefore, the simple approach of generating a transition table and using a universal scanner does not work. Interestingly, all NFAs can be converted by appropriate transformations into a DFA (see [HU79, Section 2.3]), and the transition table for that DFA can be used for scanning. Unlike a DFA, an NFA cannot be converted state-for-state into a coroutine because of the nondeterminism; converting the NFA to a DFA and then to a coroutine is still unlikely to be a good approach. Rather the coroutine should take advantage of counting and auxiliary variables. For example, the coroutine main to parse the above NFA could be:

```
char prev = 'y';
for ( ;; ) {
    stat = ch == 'x' && prev == 'x' ? GOOD : BAD;
    prev = ch;                    // remember last character
    suspend();                    // assume characters are x/y
}
```

Here, the coroutine uses auxiliary variables to retain the last two characters scanned to determine if both are x. Note, even though an NFA may not represent the best implementation for parsing, it is often a convenient mechanism to express and reason about certain parsing problems.

For the previous example, the complexity of the NFA and its equivalent DFA is the same, allowing the use of the DFA's transition table for scanning:

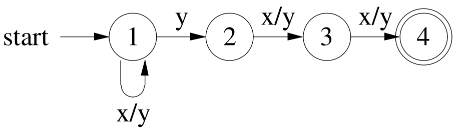
DFA



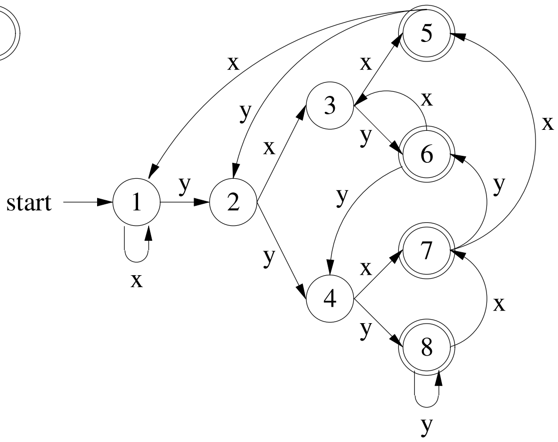
	State		
Character	1	2	3
x	2	3	3
y	1	1	1

However, for some NFA, the equivalent DFA can grow exponentially, making it impractical to use its transition table. Consider the NFA described by the regular expression (′x′ | ′y′)* ′y′ (′x′ | ′y′)$^{n-1}$, where the nth symbol from the end of the input must be a y. For $n = 3$, the NFA and its equivalent DFA are:

NFA



DFA



It is easy to see this NFA grows linearly in n. However, its DFA grows exponentially in n because it is necessary to remember the last n characters to determine whether the nth character is a y. As an automaton has no memory, it is necessary to encode each combination of the n trailing characters as a state, resulting in 2^n states. Hence, the DFA's transition table grows exponentially in n making its use for scanning strings impractical for larger values of n. As above, the coroutine can replace the NFA's nondeterminism with auxiliary variables to remember the last n characters. For example, the coroutine main to parse the above NFA could be:

```
list<char> prev;                  // or an array of size n
for ( ;; ) {
    prev.push_back( ch );         // remember character
    if ( prev.size() > n ) { prev.pop_front( ); } // only retain last n characters
    stat = prev.size() == n && prev.front() == 'y' ? GOOD : BAD;
    suspend();                    // assume characters are x/y
}
```

Here, the number of auxiliary variables is linear in n. For practical parsing problems, the number of auxiliary variables to replace an NFA's nondeterminism is at most linear in the number of states of the NFA.

There are also problems, such as matching balanced parentheses, that cannot be handled by the FA discussed so far. While it is possible to write a regular representation for a fixed n, $'('^n\ ')'^n$, or set of n:

$$'('^1\ ')'^1 \ \mid\ '('^2\ ')'^2 \ \mid\ '('^3\ ')'^3 \ \mid\ '('^4\ ')'^4 \ \mid\ \ldots$$

the general form $'('^+\ ')'^+$ does not ensure the number of opening and closing parentheses are equal. An FA can be made more powerful by allowing an auxiliary stack to remember characters; it is then called a **push-down automaton**. To match balanced parentheses, an open parenthesis is pushed onto the stack and popped off for each closing parenthesis. Once more, a coroutine can directly mimic a push-down automaton by creating an auxiliary stack. However, to match parentheses, it is sufficient to use a single auxiliary variable that counts (which a push-down automaton cannot do):

```
stat = BAD;
int cnt = 0;
for ( ;; ) {                        // scan opening parenthesis
  if ( ch == ')' ) break;           // start of closing parenthesis ?
     cnt += 1;                      // assume '('
     suspend();                     // get next character
}
for ( ;; ) {                        // scan closing parenthesis
     cnt -= 1;                      // assume ')'
  if ( cnt == 0 ) { stat = GOOD; break; }; // balanced ?
     suspend();                     // get next character
  if ( ch != ')' ) { break; };      // invalid character
}
suspend();                          // return result
// more characters ?
```

In summary, parsing is a complex class of problems for which automata are powerful tools; however, the finite and push-down automata are only simple computers, resulting in restricted programming models. Even with a more powerful computer, the ideas, concepts, and understanding provided by automata and regular expressions are applicable in many parsing contexts and present a succinct way to represent these situations with many provable properties. However, when implementing parsing problems, a programming language usually provides a less restricted programming model than those dictated by the different kinds of FA. A parsing program is allowed to have a finite number/kind of auxiliary variables and perform arbitrary arithmetic operations, which is almost equivalent to the most general model of computing, called a **Turing machine**. Hence, the actual implementation of parsing problems is not constrained by the limitations of the FA; a programmer is free to step outside of the FA programming model, producing simpler and more efficient solutions that only use the FA as a guide. Within a coroutine, a programmer can use auxiliary variables and arbitrary arithmetic, which is why the coroutine can have simpler solutions than an FA for complex parsing problems. But it is the ability of the coroutine to directly represent the execution "state" of an automaton that makes it the ideal tool for directly representing FA, and hence, programming the entire class of parsing problems.

4.5 Iterator

Another example where a routine needs to retain both data and execution state between calls is an **iterator** (or generator or cursor). In many situations, it is necessary to abstract the notion of iteration, such as when the elements of a data structure are not directly accessible. For example, a data structure may be implemented so that its elements are structured via a linked-list or a tree. Abstraction requires the data structure interface be independent of the implementation. Hence, a user cannot construct a loop that accesses any of the link fields in the data structure, precluding any form of direct traversal. Therefore, an indirect mechanism is needed to traverse the structure without exposing the internal representation. To achieve this indirect traversal requires remembering the last element of the traversal between calls to advance to the next element. In some languages, special coroutine-like constructs exist that facilitate building iterators [LAB+81, GG83, MOSS96, SPH01]. In C++, iterators are created through companion types of the basic data structure. For example, in the list data structures supplied with μC++, there are different iterator types for traversing different kinds of lists. The Standard Template Library (STL) for C++ also provides iterator types for its different data structures.

Figure 4.7, p. 79 illustrates creating and using a μC++ sequence and sequence iterator. A sequence is a doubly linked list; therefore, there are two iterators: uSeqIter iterates over the list in the forward direction and uSeqIterRev in the reverse. The member routine over resets the iterator to the head or tail of the specified sequence depending on which iterator is used. The member routine >> attempts to move the iterator's internal cursor to the next node. If the head (front) or tail (end) of the sequence has not been reached, the argument is set to the address of the next node and **true** is returned; otherwise the argument is set to NULL and **false** is returned. Because the iterator is an object, there can be multiple iterators instantiated for the same and/or different list structures, where each iterator is iterating over a different part of the same list or a completely different list.

Because an iterator can retain arbitrary information between calls, it is even possible to iterate along a list that is being deallocated, as in:

```
for ( fore.over(seq); fore >> f; ) {        // forward traversal of list
    seq.remove( f );                         // remove node during iteration
    delete f;                                // delete node storage
}
```

As in previous examples, this capability is accomplished by storing data in class variables, which are created for each instance of the object and available to each member routine of the object. Figure 4.8 shows an abridged version of the code for uSeqIter. Note that the iterator *remembers* both the list being traversed and the next list element returned by operator >> in class member variables seq and next, respectively. In effect, the iterator manages a *cursor* (next) that moves in a particular direction during the list traversal, i.e., between successive calls to operator >>. Remembering the next element to be accessed for the list traversal allows the current element to be removed and deleted without affecting the iterator's ability to continue traversing the remainder of the list. However, the control flow in operator >> is not a loop traversing the data structure but rather a sequential block of code controlled by execution-state variables. This control flow is necessary because a class type cannot retain execution state between calls. To retain both data and execution state in an iterator requires a coroutine, and a coroutine iterator makes traversing more complex data structures, like a tree data-structure, significantly simpler because execution state can be remembered [MOSS96].

Normal tree traversal algorithms use recursion to implicitly build a stack data structure, as in:

```
void printTree( Tree &t ) {                     // infixed printing of tree
    if ( t.left != NULL ) printTree( t.left );
    cout << t.data << endl;
    if ( t.right != NULL ) printTree( t.right );
}
Tree tree;
printTree( tree );
```

The recursion builds a stack down each branch of the tree, and the stack remembers both data and execution state, namely, the current vertex (node) in the tree and whether execution is traversing the left or right branch of the tree. This specific program can be generalized into a (kind of) iterator for performing any action during a traversal, as in:

```
void mapTree( Tree &t, void (*f)( Data & ) ) {  // infixed application on a tree
    if ( t.left != NULL ) mapTree( t.left );
    f( t.data );                                 // perform action on data in node
    if ( t.right != NULL ) mapTree( t.right );
}
void print( Data &d ) { cout << d << endl; }    // action
Tree tree;
mapTree( tree, print );                          // print tree
```

By passing a routine to the iterator, it is possible to perform a user specified action at each node during the traversal. This kind of iterator is often called a **mapping iterator** or **applicative iterator** because it maps or applies an action onto each node of the data structure. However, this kind of iterator is different from uSeqIter, which brings the data to the action. Bringing the data to the action (mapping iterator) is called an **internal iterator**, while bringing the action to the data is called an **external iterator** [GHJV95, p. 260]. The advantages and disadvantages of each form of iterator are discussed.

The internal iterator has difficulty dealing with the following:

- not traversing the entire data structure;
- performing different actions based on data in the node;
- the conditional for making either of the previous decisions may depend on state at the point of traversal, which is not readily available to the traverser or action.

```
struct seqNode : public uSeqable {        // inherit forward and backward link fields
    int v;
    seqNode( int v ) : v( v ) {}
};
void uMain::main() {
    uSequence<seqNode> seq;             // top pointer to list
    uSeqIter<seqNode> fore;             // forward traversal of list
    uSeqIterRev<seqNode> rev;           // backward traversal of list
    seqNode *f, *r;                     // pointer to list element

    // create list . . .

    // print list in both directions
    for ( fore.over(seq), rev.over(seq); fore >> f & rev >> r; ) {
        cout << f->v << " " << r->v << " ";
    }
    cout << endl;
    . . .
}
```

Figure 4.7: Sequence with Iterator

```
template<class T> class uSeqIter {
    const uSequence<T> *seq;            // list being traversed
    T *next;                            // cursor to element returned by >>
  public:
    uSeqIter() { seq = next = NULL; }
    void over( const uSequence<T> &s ) {
        seq = &s;                       // set list to be traversed
        next = seq.head();              // set cursor to start of list
    }
    bool operator>>( T *&tp ) {         // in/out parameter
        if ( next ) {                   // traversal completed ?
            tp = next;                  // return current location
            next = seq->succ( next );   // move and remember next location
        } else {
            tp = NULL;
        }
        return tp != NULL;              // traversal completed ?
    }
};
```

Figure 4.8: Sequence Iterator

This loop illustrates why bringing the data to the action is more general:

```
for ( int i = 0, gen.over(tree); i < 10 && gen >> d; i += 1 ) {
    if ( d->f != c ) {              // c is in the local context
        cout << *d << endl;
    } else {
        d->f += i;                  // i is in the local context
    }
}
```

The loop traverses at most the first 10 nodes of the tree, performs different actions on the nodes depending on the data in the node, and both the condition for deciding on which action to perform and the incrementing of the field f depend on state information local to the traversal context (i.e., variables c and i, respectively). While it is possible to generalize an internal iterator to handle this complex case, the generalization becomes complicated, and possibly violates abstraction and/or encapsulation.

The external iterator, i.e., bring the data to the action, has difficulty dealing with traversing portions of a data structure, i.e., only visiting relevant nodes. For example, in:

```
typeCheck( parseTree );
```

the internal iterator typeCheck may only visit portions of the parse tree containing information associated with type checking. Deciding which nodes are appropriate may be a combination of the decisions made in typeCheck and/or decisions in the parse tree itself. Because significant portions of the parse tree may not be visited during type checking, there can be a substantial performance advantage. For an external iterator, usually all of the data structure is traversed and only the appropriate nodes are used.

Therefore, there is only a weak equivalence between the two kinds of iterators; hence, both forms are needed.

4.5.1 Tree Iterator

An internal iterator (mapping iterator) is usually straightforward because it is similar in structure to a "stand-alone" program, i.e., obtaining input from the data structure and processing it directly. An external iterator is more complex because the data is not processed directly; instead it is returned for processing at the point of iteration, like the Fibonacci generator (see Section 4.1, p. 65). However, note the recursion in mapTree to implicitly retain both data and execution state during traversal. To transform mapTree to an external iterator requires a coroutine to allow recursion to traverse the tree, and yet, return control back for each node of the traversal.

Figure 4.9 shows an abridged version of the code for uTreeGen, and its basic structure is similar to uSeqIter in Figure 4.8. Like uSeqIter, uTreeGen manages a *cursor* (next) that moves up and down the branches of the tree between successive calls to operator >>. Operator >> resumes the coroutine main to get the next node in the tree. Member nextNode performs the recursive traversal of the tree, suspending back to operator >> for each node in the traversal. Therefore, the resume in operator >> normally restarts the coroutine main in one of the recursive invocations of nextNode, where it has suspended. When member nextNode finishes the traversal, it returns to the call in uTreeGen::main, which completes the traversal by setting next to NULL. Member uTreeGen::main then suspends back to operator >>, which returns a NULL data pointer and **false**; therefore, uTreeGen::main does not terminate the coroutine, but waits for the next traversal to start.

The tree iterator convincingly illustrates the need for coroutines to build advanced external iterators. Without a coroutine, the implementation is extremely awkward requiring the explicit creation and manipulation of a stack.

4.6 Coroutine Details

The following μC++ coroutine details are presented to understand subsequent examples or to help in solving questions at the end of the chapter.

4.6.1 Coroutine Creation and Destruction

A coroutine is the same as a class object with respect to creation and destruction, e.g.:

```
template<class T> _Coroutine uTreeGen {
    T *next;                                    // cursor to element returned by >>
    const Tree<T> *tree;                        // tree being traversed

    void nextNode( T *node ) {                  // infix tree traversal
        if ( node ) {                           // traversal complete ?
            nextNode( node->left );             // search left branch
            next = node;                        // remember next location
            suspend();                          // reactivate resumer
            nextNode( node->right );            // search right branch
        }
    }
    void main () {
        for ( ;; ) {                            // for each traversal
            nextNode( next );                   // start recursion
            next = NULL;                        // indicate traversal is complete
            suspend();                          // reactivate resumer
        }
    }
  public:
    uTreeGen() { tree = next = NULL; }
    void over( const Tree<T> &t ) {
        tree = &t;                              // set tree to be traversed
        next = t.top();                         // set cursor to root of tree
    }
    bool operator>>( T *&tp ) {                 // in/out parameter
        resume();                               // activate coroutine
        tp = next;                              // return current location
        return next != NULL;                    // traversal completed ?
    }
};
```

Figure 4.9: Tree Iterator

```
_Coroutine C {
    void main() . . .         // coroutine main
  public:
    void r( . . . ) . . .
};
C *cp;                        // pointer to a C coroutine
{ // start a new block
    C c, ca[3];               // local creation
    cp = new C;               // dynamic creation
    . . .
    c.r( . . . );             // call a member routine that activates the coroutine
    ca[1].r( . . . );
    cp->r( . . . );
    . . .
} // c, ca[0], ca[1] and ca[2] are deallocated
. . .
delete cp; // cp's instance is deallocated
```

When a coroutine is created, the appropriate coroutine constructor and, if there is inheritance, any base-class constructors are executed in the normal order. The stack component of the coroutine's execution-state is created and the starting point (activation point) is initialized to the coroutine's main routine visible by the inheritance scope rules from the coroutine type; however, the main routine does not start execution until the coroutine is activated by one of its member routines. The location of a coroutine's variables—in the coroutine's data area or in member routine main—depends on whether the variables must be accessed by member routines other than main. Once main is activated, it executes until it activates another coroutine or terminates. The coroutine's point of last activation may be outside of the main routine because main may have called another routine; the routine called could be local to the coroutine or in another coroutine. This issue is discussed further in several example programs.

The coroutine property of a coroutine type, i.e., its execution state, is independent of its object properties. If a coroutine is never resumed, its execution state is never used (and hence, may not be implicitly allocated); in this case, the coroutine behaves solely as an object. A coroutine terminates when its main routine terminates, and its execution state is no longer available (and hence, may be implicitly deleted). After termination, the coroutine behaves as an object, *and cannot be resumed again*. It is possible to determine if a coroutine is terminated by calling getState(), which returns one of Halt, Active, Inactive, where Halt implies terminated. Because a coroutine begins and ends as an object, calls to member routines that manipulate coroutine variables are possible at these times.

When a coroutine terminates, it activates the coroutine that caused main to *start* execution. This choice ensures that the starting sequence is a tree, i.e., there are no cycles. Control flow can move in a cycle among a group of coroutines but termination always proceeds back along the branches of the starting tree. This choice for termination does impose certain requirements on the starting order of coroutines, but it is essential to ensure that cycles can be broken at termination. This issue is discussed further in several example programs. Finally, attempting to activate a terminated coroutine is an error.

4.6.2 Inherited Members

Each coroutine type, if not derived from some other coroutine type, is implicitly derived from the coroutine type uBaseCoroutine, e.g.:

```
_Coroutine coroutine-name : public uBaseCoroutine {  // implicit inheritance
    . . .
};
```

where the interface for the base-class uBaseCoroutine is:

```
_Coroutine uBaseCoroutine {
  protected:
    void resume();
    void suspend();
  public:
    uBaseCoroutine();
    uBaseCoroutine( unsigned int stacksize );
    void verify();
    const char *setName( const char *name );
    const char *getName() const;
    enum State { Halt, Active, Inactive };
    State getState() const;
    uBaseCoroutine &starter() const;
    uBaseCoroutine &resumer() const;
};
```

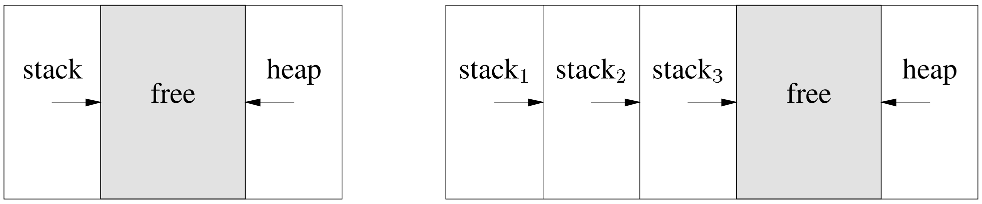
The protected member routines resume and suspend are discussed in Section 4.6.3, p. 84.

The overloaded constructor routine uBaseCoroutine has the following forms:

uBaseCoroutine() – creates a coroutine with the default stack size.

uBaseCoroutine(**unsigned int** stacksize) – creates a coroutine with the specified stack size (in bytes).

Unlike a normal program with a single execution-state, a coroutine program has multiple execution-states, and hence, multiple stacks for each coroutine's routine activations. Managing a single stack can be done simply and implicitly for normal programs, as in Figure 4.10(a). The storage is divided into three areas: the stack for routine activations, the heap for dynamically allocated variables (i.e., variables created with **new**), and the free currently unused storage. Normally, the stack and heap storage grow towards one another until they meet; at which time the program normally terminates with an error. However, managing multiple stacks is significantly more complex and may require some explicit involvement from the programmer, as in Figure 4.10(b). There is no simple technique to allow each stack to grow dynamically into the free area, as for a normal program. Any technique allowing each coroutine stack to grow dynamically requires compiler and runtime support, which is beyond what μC++ provides. Instead, in μC++, each



(a) normal (b) coroutine

Figure 4.10: Storage Allocation

coroutine stack is a fixed size and cannot grow, which requires the programmer to ensure the stack for each coroutine is large enough when the coroutine is created. If no stack size is specified, the default value is architecture dependent, but is usually at least 32K bytes of storage. While this amount is small, it is usually large enough for most coroutines unless there are large variables (e.g., a big array) declared in the coroutine main or a deep call graph for the coroutine (e.g., a call to a routine with deep recursion). A coroutine type can be designed to allow declarations of it to specify individual stack sizes by doing the following:

```
_Coroutine C {
  public:
    C() : uBaseCoroutine( 8192 ) {};       // default 8K stack
    C( unsigned int size ) : uBaseCoroutine( size ) {};  // user specified stack size
    . . .
};
C x, y( 16384 );      // x has an 8K stack, y has a 16K stack
```

μC++ attempts to check if a coroutine overflows its stack during execution, but there are a few situations it cannot detect. One of these situations is when the top of one stack overflows onto the bottom of the next coroutine's stack, arbitrarily modifying this stack. In this case, an error may not occur until the modified coroutine moves back down its stack to the point of corruption. At this point, an unusual error may occur completely unrelated to the coroutine that overflowed its stack and caused the problem. Such an error is extremely difficult to track down. One way to detect possible stack overflows is through the member verify, which checks whether the current coroutine has overflowed its stack; if it has, the program terminates. To completely ensure the stack size is never exceeded, a call to verify must be included after each set of declarations, as in the following:

```
void main() {
    . . .          // declarations
    verify();      // check for stack overflow
    . . .          // code
}
```

Thus, after a coroutine has allocated its local variables, a check is made that its stack has not overflowed. Clearly, this technique is not ideal and requires additional work for the programmer, but it does handle complex cases where the stack depth is difficult to determine and can be used to help debug possible stack overflow situations.

The member routine setName associates a name with a coroutine and returns the previous name. *The name is not copied so its storage must persist for the duration of the coroutine.* The member routine getName returns the string name associated with a coroutine. If a coroutine has not been assigned a name, getName returns the type name of the coroutine. μC++ uses the name when printing any error message, which is helpful in debugging.

The member routine getState returns the current state of a coroutine's execution, which is one of the enumerated values Halt, Active or Inactive. (When accessed from outside of a coroutine type, these enumerated values must be qualified with "uBaseCoroutine::".)

The member routine starter returns the coroutine's starter, i.e., the coroutine that performed the first resume of this coroutine (see Section 4.6.1, p. 80). The member routine resumer returns the coroutine's last resumer, i.e., the coroutine that performed the last resume of this coroutine (see Section 4.6.3).

The routine:

```
uBaseCoroutine &uThisCoroutine();
```

is used to determine the identity of the coroutine executing this routine. Because it returns a reference to the base coroutine type, uBaseCoroutine, this reference can only be used to access the public routines of type uBaseCoroutine. For example, a routine can check whether the allocation of its local variables has overflowed the stack of a coroutine that called it by performing the following:

```
int Rtn( ... ) {
    ...                      // declarations
    uThisCoroutine().verify();      // check for stack overflow
    ...                      // code
}
```

As well, printing a coroutine's address for debugging purposes must be done like this:

```
cout << "coroutine:" << &uThisCoroutine() << endl;  // notice the ampersand (&)
```

4.6.3 Coroutine Control and Communication

Control flow among coroutines is specified by the protected members resume and suspend. These members are protected to prevent one coroutine from directly resuming or suspending another coroutine, as in c.resume(). Rather, a coroutine should package these operations as in the following:

- A call to resume may appear in any member of the coroutine, but normally it is used only in the public members.
- A call to suspend may appear in any member of the coroutine, but normally it is used only in the coroutine main or non-public members called directly or indirectly from the coroutine main.

The action performed by members resume and suspend is composed of two parts. The first part inactivates the coroutine calling the member and the second part reactivates another coroutine.

1 inactive	$\rightarrow$	2 active
uThisCoroutine()	resume	this
	suspend	last resumer

Both resume and suspend inactivate the current coroutine (denoted by uThisCoroutine). The second part reactivates a difference coroutine depending on the specific member. Member resume activates the current coroutine object, i.e., the coroutine specified by the implicit **this** variable. Member suspend activates the coroutine that previously executed a call to resume for the coroutine executing the suspend, *ignoring any resumes of a coroutine to itself*. In effect, these special members cause control flow to transfer among execution states, which involves context switches.

It is important to understand that calling a coroutine's member by another coroutine does not cause a switch to the other coroutine. A switch only occurs when a resume is executed in the other coroutine's member. Therefore, printing &uThisCoroutine() in the other coroutine's member always prints the *calling* coroutine's address; printing **this** in the other coroutine's member always prints the *called* coroutine's address (which is the coroutine that resume switches to). Hence, there is a difference between who is executing and where execution is occurring.

4.7 Semi- and Full Coroutine

Coroutines can be structured in two different ways:

1. The **semi-coroutine** [Mar80, p. 4,37] structure acts asymmetrically, like non-recursive subroutines performing call and return. The first coroutine activates the second coroutine by calling one of the members in the second coroutine that performs a resume; the second coroutine implicitly reactivates the first coroutine by performing a suspend. Just as a return for a routine implicitly knows its last caller, a suspend for a coroutine implicitly knows the last coroutine that resumed it.

2. The **full coroutine** structure acts symmetrically, like recursive subroutines performing calls that involve a cycle. The first coroutine activates the second coroutine by calling one of the members of the second coroutine that performs a resume; the second coroutine activates the first coroutine by directly or indirectly calling one of

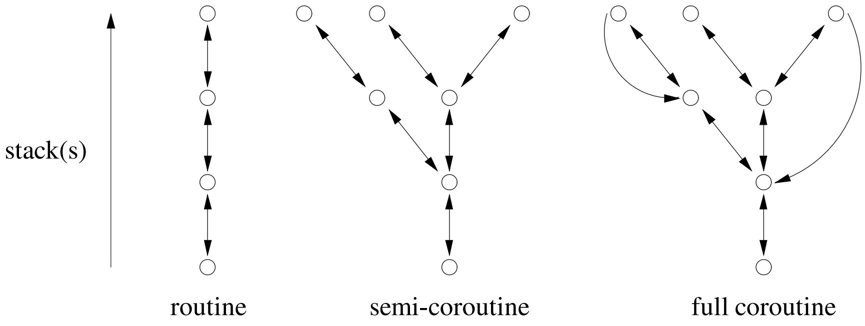


Figure 4.11: Activation Structure

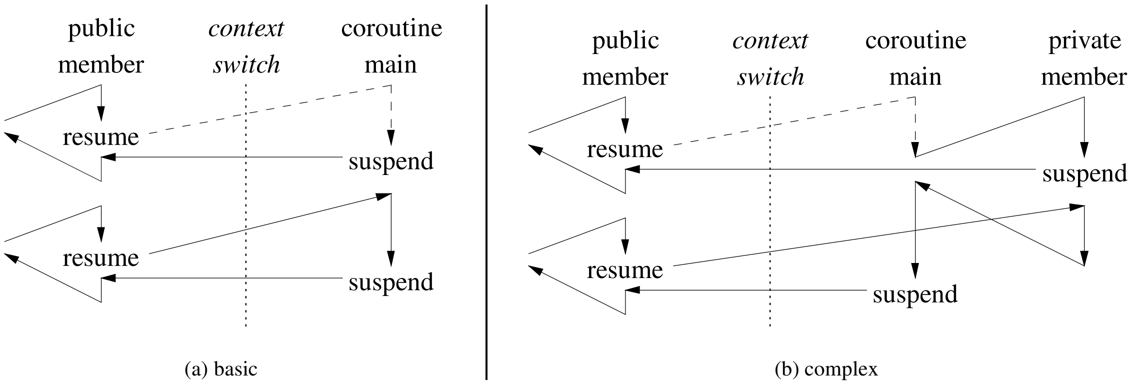


Figure 4.12: Semi-Coroutine Control Flow

> the members of the first coroutine, which then performs a resume. The difference between recursion and full-coroutining is that each invocation of a routine creates a new instance of the routine, while full-coroutining only cycles among preexisting coroutines.

Hence, a full coroutine is part of a resume cycle, while a semi-coroutine never participates in a resume cycle. A full coroutine is allowed to perform semi-coroutine operations because it subsumes the notion of the semi-coroutine; i.e., a full coroutine can use suspend to activate the member routine that activated it or resume to itself, but it must always form a resume cycle with other coroutines.

Figure 4.11 compares the general control flow available from routines, semi-coroutines and full coroutines. In the figure, vertices represent instances of routines or coroutines, and arcs represent call/return or suspend/resume or resume/resume control flow. On the left, multiple routine calls form a non-branching tree of suspended routines where call goes up the branch allocating a new vertex and return goes down the branch deallocating the current vertex. Even when routines use recursion, there is still only a non-branching tree because each new instance is created at the top of the tree. In the middle, semi-coroutines allow the tree to branch because each coroutine has its own execution state, which allows it to persist between calls and form another call/return branch. Resuming forks up a branch and suspending joins back to a branch point. On the right, full coroutines allow cycles to form, creating a network graph with arbitrary topology. Each kind of coroutine control-flow is discussed in detail.

4.7.1 Semi-Coroutine

The most common form of coroutine structure is the semi-coroutine; all the examples presented thus far are semi-coroutines. Semi-coroutines are somewhat like routines in that the resumes and suspends move to and from a coroutine like a call and return move to and from a routine. This basic flow of control is illustrated in Figure 4.12(a). At each resume, the complete state of the resumer is retained on the stack of the cocaller, including the call to the coroutine's member that executes the resume. At each suspend, the complete state of the suspender is retained on the stack of the coroutine.

```
_Coroutine fc {
    void main() {
        for ( int i = 0; i < 5; i += 1 ) {
            start();
        }
        . . . // resume() here ?
    }
  public:
    void start() { resume(); }
};
void uMain::main() {
    fc x;
    x.start();
}
```

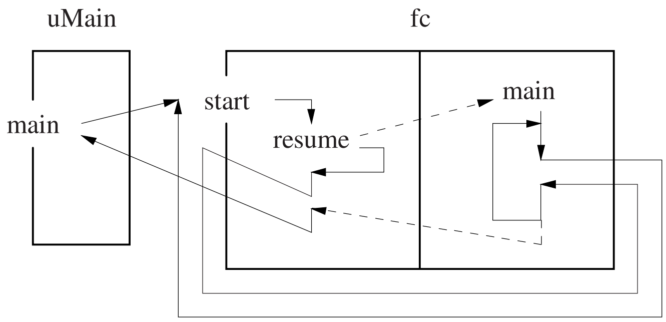


Figure 4.13: Basic Full Coroutine Control Flow

A more complex example appears in Figure 4.12(b), which shows that the coroutine main can be suspended outside of its code body. That is, the coroutine main can call a private member, for example, which then suspends the coroutine (see routine nextNode in Figure 4.9, p. 81). In this case, when the coroutine suspends, there is an activation for the private member on the coroutine's stack along with the coroutine main. When the coroutine is resumed, it restarts execution at its last suspension point, which restarts the private member with all of its local state. The number of suspended routine calls is limited only by the stack size of the coroutine main.

4.7.2 Full Coroutine

The less common form of coroutine structure is the full coroutine; it is the most complex form of control flow possible with coroutines. This form of control flow is useful in solving very complex forms of sequential control-flow; as well, understanding full-coroutine control-flow is the first step in understanding concurrent control-flow.

Full coroutines are somewhat like recursive routines in that the coroutine calls itself directly or indirectly to form a cycle. *However, unlike recursion, a new instance of the coroutine is not created for each call*; control simply flows in a cycle among the coroutines, like a loop. Figure 4.13 shows the simplest full coroutine, and a driver routine, as well as a diagram of the basic control flow between these components. Starting in member uMain::main, the full coroutine x is created and a call is made to its member start to activate it. The start member immediately resumes the coroutine main, which makes the anonymous coroutine for uMain::main inactive and starts the coroutine associated with the object of the member call, x. Control transfers to the beginning of x's main member because it is the first resume (dashed line in the diagram), and the anonymous coroutine for uMain::main is remembered as the starting coroutine for subsequent termination.

The full coroutine's main member creates local variables that are retained between subsequent activations of the coroutine. It then executes 5 calls to *its* start member; the call to start begins a cycle. This call implicitly suspends member fc::main (part of a normal routine call) and starts a new instance of start on x's execution-state. The resume in start makes x inactive and reactivates the coroutine associated with the object of the member call, x, but does not create a new instance of it. Hence, x does a context switch to itself and execution continues after the resume; effectively, the resume does nothing but costs something, i.e., the time to do the context switch. The call to start returns and the loop in fc::main performs another iteration. Follow this flow of control in the diagram of Figure 4.13. The key point is that resume restarts the current coroutine, i.e., **this**, wherever it is suspended. Since, the coroutine main can call other routines before it suspends, it may restart completely outside of its own code, as is also possible with semi-coroutines. Given this description of resume, explain what occurs if a call to resume is placed in fc::main?

When the loop finishes, two scenarios are examined: execute a suspend or return. For the first scenario, suspend starts by inactivating the coroutine that invokes it, and then activates the coroutine that most recently executed a call to resume for this coroutine. Since the coroutine that last executed a call to resume is x, the suspend would result in x context switching to itself (like a resume). While this semantics is consistent with the definition of suspend, it points out that the original resumer coroutine, uMain::main, is now lost because it was overwritten by the resume of the coroutine to itself. Now the ability to reactivate a coroutine's last resumer is useful, and this capability is lost only when a coroutine resumes itself, and the need for a coroutine to resume is neither necessary nor useful, so μC++ makes a special exception when performing a resume: a resume of a coroutine to itself does not overwrite the last resumer of the coroutine. Given this exception, the suspend makes active the anonymous coroutine for uMain::main, which continues after the resume in start, returns from the call x.start(), and uMain::main ends, with an unterminated coroutine x. For the second scenario, fc::main returns and terminates so the special termination rule applies, which reactivates the coroutine that *first* resumed (activated) this coroutine. For x, the starting coroutine was the anonymous coroutine for uMain::main, so x terminates and cannot be resumed again; then, the anonymous coroutine for uMain::main is made active, it continues the resume in start, returns from the call x.start(), and uMain::main ends, with a terminated coroutine x. When uMain::main ends, the special termination rule resumes its starter and the program subsequently terminates.

Figure 4.14 shows two coroutines that form a cycle, and a driver routine, as well as a diagram of the basic control flow among these components. This example is sufficient to illustrate how to build more complex full coroutines; there are three phases to any full coroutine program:

1. starting the cycle
2. executing the cycle
3. stopping the cycle

Starting the cycle is complicated because each coroutine needs to know at least one other coroutine to form a cycle, for example, x knows y, y knows x. The problem is the mutually recursive references, i.e., the declarations need to be:

```
fc x(y), y(x);
```

but the first declaration fails because of the *definition before use rule* of C++, that is, y is passed as an argument to x *before* it is defined. Clearly, switching the order of declaration of x and y does not help.

There are several ways to solve this problem. The mechanism presented here is to declare the first instance without a partner and pass it a partner after the other coroutines are created, as in:

```
fc x, y(x);      // only y gets a partner now
x.partner(y);    // x gets its partner after y is declared
```

The cycle is now closed by the call to x.partner, which initializes x's partner. This approach is the technique used in Figure 4.14.

(Notice the type of the class variable part is a pointer rather than a reference. A pointer is required because a reference can only be initialized when an object is created through constructor initialization. Since the constructor cannot be used to initialize all the partner values because of the mutually recursive reference, a pointer type must be used.)

Once the cycle is created, execution around the cycle may begin. Like the previous example, uMain::main starts the cycle by calling member start in one of the coroutines in the cycle; in this example, the cycle can be started in either x or y.

Follow the flow of control between the two coroutines at the bottom of Figure 4.14. Resume$_1$ in x's start member resumes the coroutine main, which makes the anonymous coroutine for uMain::main inactive and starts the coroutine associated with the object of the member call, x, but does not create a new instance of it. Control transfers to the beginning of x's main member because it is the first resume (dashed line in the diagram), and the anonymous coroutine for uMain::main is remembered as the starting coroutine for subsequent termination.

x's main member creates local variables that are retained between subsequent activations of the coroutine. It then executes 5 calls to y.start. Each call implicitly suspends member fc::main (part of a normal routine call) and starts a new instance of start on x's execution-state. Resume$_2$ in y's start member resumes the coroutine main, which makes x inactive and starts the coroutine associated with the object of the member call, y, but does not create a new instance of it. Hence, x only does a context switch to y. Control transfers to the beginning of y's main member because it is the first resume (dashed line in the diagram), and coroutine x is remembered as the starting coroutine for subsequent termination.

y's main member creates local variables that are retained between subsequent activations of the coroutine. It then executes 5 calls to x.start. Each call implicitly suspends member fc::main (part of a normal routine call) and starts a new instance of start on y's coroutine's stack. Resume$_1$ in x's start makes y inactive and reactivates the coroutine associated with the object of the member call, x, but does not create a new instance of it. Hence, y only does a context switch to x, and the cycle is now formed. Control transfers to the location of x's last inactivation after resume$_2$ in member y.start, which returns to x.main for the next call to y.start. Resume$_2$ in y's start member makes x inactive

```
_Coroutine fc {
    fc *part;
    void main() {
        for ( int i = 0; i < 5; i += 1 ) {
            part->start();
        }
    }
  public:
    fc() {};
    fc( fc &p ) : part(&p) {};
    void partner( fc &p ) { part = &p; }
    void start() { resume(); }
};
void uMain::main() {
    fc x, y(x);
    x.partner(y);         // close cycle
    x.start();            // start cycle
}
```

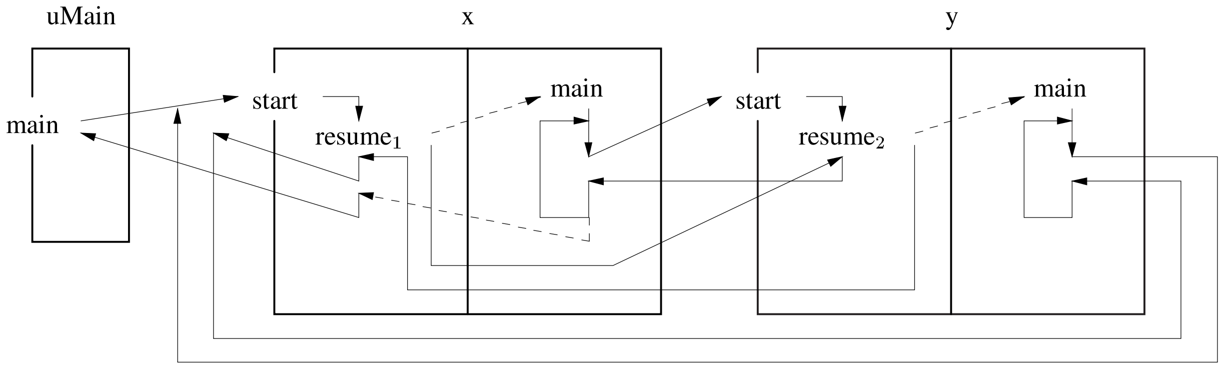


Figure 4.14: Complex Full Coroutine

and reactivates the coroutine associated with the object of the member call, y, but does not create a new instance of it. Hence, x only does a context switch to y. Control transfers to the location of y's last inactivation after resume_1 in member x.start, which returns to y.main for the next call to x.start. Trace along the flow lines of Figure 4.14 until you have found the cycle and understand why resume_1 and resume_2 transfer control to where they do. It might help to write down for each resume which routine is suspending so it is easy to see why it restarts where it does.

Stopping the cycle can sometimes be as complex as starting the cycle. In this example, x finishes first because it started first. When x's main finishes, the special termination rule applies so x resumes the anonymous coroutine for uMain::main, which first resumed x (dashed line in the diagram); uMain then terminates with a terminated coroutine x and unterminated coroutine y. As stated previously, there is nothing wrong with deallocating a coroutine that is suspended; it is no different than deallocating an object. Further examples show how to programmatically terminate *all* coroutines in a cycle.

It is important to understand this program to understand further examples. Therefore, before reading on, make sure you understand the control flow in this program.

4.8 Producer-Consumer Problem

The producer-consumer problem is a standard problem, that is, it is a problem that represents a class of problems. All problems in that class can be solved using some variation on the basic solution to the standard problem. This book discusses several standard problems and presents one or more basic solutions. One of the main skills the reader must develop is the ability to identify standard problems within a problem and adapt standard solutions to solve these problems. This approach is significantly faster than solving problems directly from first principles.

The producer-consumer problem is very simple: a producer generates objects and a consumer receives these objects. It does not matter what the objects are or what the consumer does with them, as long as the producer and consumer agree on the kind of object. When the producer is only calling the consumer there are no cycles in the control flow. If the producer calls the consumer *and* the consumer calls the producer, there is a cycle in the control flow. Both semi- and full coroutine producer and consumer solutions are discussed, which illustrate these two cases.

4.8.1 Semi-coroutine Solution

Figure 4.15 shows a semi-coroutine producer and consumer, and a driver routine, as well as a diagram of the basic control flow among these components. Starting in member uMain::main, the consumer cons is created followed by the producer prod. Since the solution is semi-coroutining, only the consumer needs to be passed to the producer so the producer can reference the consumer, making starting straightforward. Next, a call is made to the start member of the producer to start the producer coroutine. The number of values generated by the producer is passed as the argument (5 in the program).

The start member in Prod communicates the number of elements to be produced to the coroutine main by copying the parameter to a class variable, which is accessible to the member main. Next, a resume is executed, which makes the anonymous coroutine for uMain::main inactive and starts the coroutine associated with the object of the member call, prod. Control transfers to the beginning of prod's main member because it is the first resume (dashed line in the diagram), and the anonymous coroutine for uMain::main is remembered as the starting coroutine for subsequent termination.

The producer's main member creates local variables that are retained between subsequent activations of the coroutine. Then it executes N iterations of generating two random integer values between 0–99, printing the two values, calling the consumer to deliver the two values, and printing the status returned from the consumer. Normally, the values generated are more complex and the status from the consumer is examined to determine if there is a problem with the delivered values; however, neither of these points is important to understand the control flow between producer and consumer in this example.

The call from the producer to the consumer's delivery member transfers data from producer to consumer. When delivery is called, it is the producer coroutine, prod, executing the member. The values delivered by the producer are copied into communication variables in the consumer and a resume is executed. The resume makes prod inactive and reactivates the coroutine associated with the object of the member call, cons. Control transfers to the beginning of cons's main member because it is the first resume (dashed line in the diagram), and coroutine prod is remembered as the starting coroutine for subsequent termination.

The consumer's main member creates local variables that are retained between subsequent activations of the coroutine. Then it iterates until the done flag is set, printing the two values delivered, incrementing the status for returning, printing the amount it pays for the values, suspending and incrementing the amount of money for the next payment. The suspend makes cons inactive and reactivates the coroutine that last resumed it, namely prod. Control transfers to the point where prod was last made inactive and it continues, which is in member delivery after the resume.

The delivery member returns the status value to the call in prod's main member, where the status is printed. The loop then repeats calling delivery, where each call resumes the consumer coroutine. When the consumer is made active it continues after the point where it was last made inactive, which is the suspend in cons's main member.

After iterating N times, the producer calls the stop member in cons. When stop is called, it is the producer coroutine, prod, executing the member. The done flag is set to stop the consumer's execution and a resume is executed. The resume makes prod inactive and reactivates the coroutine associated with the object of the member call, cons. Control transfers after the last inactive point in cons, which is the suspend in cons's main member. The consumer prints a termination message and its main member terminates. When main terminates, the special termination rule applies, which reactivates the coroutine that *first* resumed (activated) this coroutine. For cons, the starting coroutine was prod, so cons terminates and cannot be resumed again, and prod reactivates after the resume in consumer member stop (dashed line in the diagram).

The stop member returns and prod's main member terminates. When main terminates, the special termination rule applies, which reactivates the coroutine that *first* resumed this coroutine. For prod, the starting coroutine was the anonymous coroutine for uMain::main, so prod terminates and cannot be resumed again, and the anonymous coroutine for uMain::main reactivates after the resume in start. The start member returns and uMain::main terminates, restarting the coroutine that *first* started it. Hence, this example illustrates terminating all coroutines rather than having uMain::main end with suspended coroutines.

Trace through the program and diagram to see the control flow. The dashed lines at the end of each main show

```
_Coroutine Cons {
    int p1, p2, status;         // communication
    bool done;

    void main() {
        // 1st resume starts here
        int money = 1;
        for ( ;; ) {
          if ( done ) break;
            cout << "cons receives: " <<
                p1 << ", " << p2;
            status += 1;
            cout << " and pays $" <<
                money << endl;
            suspend();  // restart delivery & stop
            money += 1;
        }
        cout << "cons stops" << endl;
    }
  public:
    Cons() : status(0), done(false) {}
    int delivery( int p1, int p2 ) {
        Cons::p1 = p1;
        Cons::p2 = p2;
        resume();           // restart main
        return status;
    }
    void stop() {
        done = true;
        resume();           // restart main
    }
};
```

```
_Coroutine Prod {
    Cons &cons;         // communication
    int N;

    void main() {
        // 1st resume starts here
        int i, p1, p2, status;
        for ( i = 1; i <= N; i += 1 ) {
            p1 = rand() % 100;
            p2 = rand() % 100;
            cout << "prod delivers: " <<
                p1 << ", " << p2 << endl;
            status = cons.delivery( p1, p2 );
            cout << "prod status: " <<
                status << endl;
        }
        cout << "prod stops" << endl;
        cons.stop();
    }
  public:
    Prod( Cons &c ) : cons(c) {}
    void start( int N ) {
        Prod::N = N;
        resume();           // restart main
    }
};

void uMain::main() {
    Cons cons;              // create consumer
    Prod prod(cons);        // create producer
    prod.start(5);          // start producer
}
```

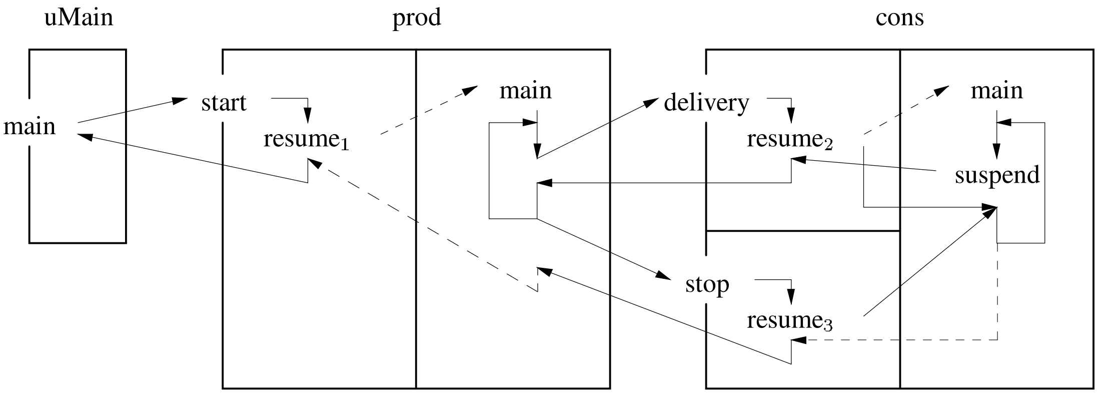


Figure 4.15: Semi-coroutine: Producer-Consumer

the implicit resume when a coroutine terminates, which go in the reverse direction to the dashed lines that started each coroutine main on the first resume. These lines go back to the coroutine that started the coroutine originally; remember, it is not the order of declaration, but who starts whom. Is the Prod::start member necessary or could the number of loop iterations be passed to Prod::Prod along with the consumer? Depending on your answer to the previous question, does Prod need to be a coroutine or could it be a class?

4.8.2 Full Coroutine Solution

Figure 4.16 shows a full coroutine producer and consumer coroutine, and a driver routine. (Actually, the code for member Cons::main must be placed after the definition of Prod because of mutual references.) Figure 4.17, p. 93 shows the basic control flow between the producer and consumer. Starting in member uMain::main, the consumer cons is created followed by the producer prod. Since the solution is full coroutining, the consumer needs to be passed to the producer so the producer can reference the consumer, and vice versa. This mutual reference is accomplished using the approach mentioned in Section 4.7.2, p. 86, that is, pass the final partner after the other coroutines are created. In this example, the partner member is combined with the start member. A call is then made to the start member of the producer to start the producer coroutine, passing both the number of values generated by the producer and the partner for prod.

The start member in Prod communicates both the number of elements to be produced and the consumer to the coroutine main by copying the parameters to class variables, which are accessible to the member main. Next, a resume is executed, which makes the anonymous coroutine for uMain::main inactive and starts the coroutine associated with the object of the member call, prod. Control transfers to the beginning of prod's main member because it is the first resume (dashed line in the diagram), and the anonymous coroutine for uMain::main is remembered as the starting coroutine for subsequent termination.

The producer's main member creates local variables that are retained between subsequent activations of the coroutine. Like the semi-coroutine version, the main member executes N iterations of generating two random integer values between 0–99, printing the two values, calling the consumer to deliver the two values, and printing the status returned from the consumer.

The call from the producer to the consumer's delivery member transfers data from producer to consumer. When delivery is called, it is the producer coroutine, prod, executing the member. The values delivered by the producer are copied into communication variables in the consumer and a resume is executed. The resume makes prod inactive and reactivates the coroutine associated with the object of the member call, cons. Control transfers to the beginning of cons's main member because it is the first resume (dashed line in the diagram), and coroutine prod is remembered as the starting coroutine for subsequent termination.

The consumer's main member creates local variables that are retained between subsequent activations of the coroutine. The main member iterates until the done flag is set, printing the two values delivered, incrementing the status for returning, printing the amount it pays for the values, calling back to the producer's payment member, printing the receipt from the producer, and incrementing the amount of money for the next payment.

It is the call from the consumer to the producer's payment member that makes this a full coroutine solution because it forms a cycle between producer and consumer. When payment is called, it is the consumer coroutine, cons, executing the member. The value delivered by the consumer is copied into a communication variable, the value is printed, and a resume is executed. The resume makes cons inactive and reactivates the coroutine associated with the object of the member call, prod. Control transfers to the point where prod was last made inactive and it continues, which is in member delivery after the resume.

The delivery member returns the status value to the call in prod's main member, where the status is printed. The loop then repeats calling delivery, where each call resumes the consumer coroutine. When the consumer is made active it continues after the point where it was last made inactive, which is the resume in payment. The consumer increments the receipt and returns the receipt value to the call in cons's main member. The loop then repeats calling payment, where each call resumes the producer coroutine.

After iterating N times, the producer calls the stop member in cons. When stop is called, it is the producer coroutine, prod, executing the member. The done flag is set to stop the consumer's execution and a resume is executed. The resume makes prod inactive and reactivates the coroutine associated with the object of the member call, cons. Control transfers after the last inactive point in cons, which is the resume in payment, which returns with the last receipt. The consumer prints the last receipt, terminates its loops because done is true, prints a termination message and its main member terminates. When main terminates, the special termination rule applies, which reactivates the

```
_Coroutine Cons {
    Prod &prod;          // communication
    int p1, p2, status;
    bool done;

    void main() {
        // 1st resume starts here
        int money = 1, receipt;
        for ( ;; ) {
          if ( done ) break;
            cout << "cons receives: " <<
                p1 << ", " << p2;
            status += 1;
            cout << " and pays $" <<
                money << endl;
            receipt = prod.payment( money );
            cout << "cons receipt #" <<
                receipt << endl;
            money += 1;
        }
        cout << "cons stops" << endl;
    }
  public:
    Cons( Prod &p ) : prod(p), done(false), status(0) {}

    int delivery( int p1, int p2 ) {
        Cons::p1 = p1;  // restart cons in
        Cons::p2 = p2;  // Cons::main 1st time
        resume();       // and afterwards cons
        return status;  // in Prod::payment
    }
    void stop() {
        done = true;
        resume();
    }
};
```

```
_Coroutine Prod {
    Cons *cons;          // communication
    int N, money, receipt;


    void main() {
        // 1st resume starts here
        int i, p1, p2, status;
        for ( i = 1; i <= N; i += 1 ) {
            p1 = rand() % 100;
            p2 = rand() % 100;
            cout << "prod delivers: " <<
                p1 << ", " << p2 << endl;
            status = cons->delivery( p1, p2 );
            cout << "prod status: " <<
                status << endl;
        }
        cons->stop();
        cout << "prod stops" << endl;
    }
  public:
    Prod() : receipt(0) {}

    int payment( int money ) {
        Prod::money = money;
        cout << "prod payment of $" <<
            money << endl;
        resume();           // restart prod
        receipt += 1;       // in Cons::delivery
        return receipt;
    }
    void start( int N, Cons &c ) {
        Prod::N = N;
        cons = &c;
        resume();
    }
};
void uMain::main() {
    Prod prod;
    Cons cons( prod );
    prod.start( 5, cons );
}
```

Figure 4.16: Full Coroutine: Producer-Consumer

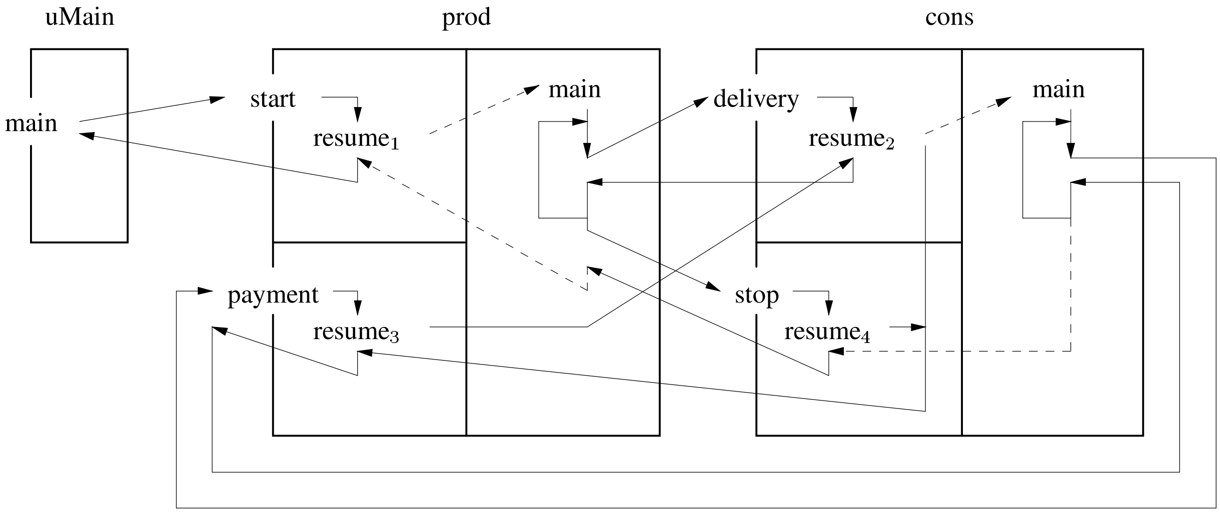


Figure 4.17: Full Coroutine: Control Flow for Producer-Consumer

coroutine that *first* resumed (activated) this coroutine. For cons, the starting coroutine was prod, so cons terminates and cannot be resumed again, and prod reactivates after the resume in stop (dashed line in the diagram).

The stop member returns and prod's main member terminates. When main terminates, the special termination rule applies, which reactivates the coroutine that *first* resumed this coroutine. For prod, the starting coroutine was the anonymous coroutine for uMain::main, so prod terminates and cannot be resumed again, and the anonymous coroutine for uMain::main reactivates after the resume in start. The start member returns and uMain::main terminates, restarting the coroutine that *first* started it. Hence, this example illustrates terminating all coroutines rather than having uMain::main end with suspended coroutines.

Trace through the starting of the two coroutines until both coroutine main routines are running. At this point, the full coroutine cycle begins. Trace through the full coroutine cycle to see how it passes both control and data between the producer and consumer. Note, the resume in member payment activates the execution state associated with Prod::main and that execution state continues in member Cons::delivery. Similarly, the resume in member delivery activates the execution state associated with Cons::main and that execution state continues in member Prod::payment. Finally, trace through the stopping sequence to see how the full coroutine cycle is broken so that the coroutines can terminate programmatically instead of deleting suspended coroutines in uMain::main.

4.9 Nonlocal Propagation

A local exception within a coroutine behaves like an exception within a routine or class, with one difference. An exception raised and not handled inside a coroutine terminates it and implicitly raises a nonlocal exception of type uBaseCoroutine::UnHandledException at the coroutine's last resumer rather than performing the default action of aborting the program. For example, in:

```
_ThrowEvent E {};

_Coroutine C {
    void main() { _Throw E(); }
  public:
    void mem() { resume(); }
};
void uMain::main() {
    C c;
    c.mem();            // first call fails
}
```

the call to c.mem resumes coroutine c, and then inside c.main an exception is raised that is not handled locally by c. When the exception of type E reaches the top of c's stack without finding an appropriate handler, coroutine c is

```
_ThrowEvent E {};

_Coroutine C {
    void main() {
        try {
            _Enable {          // allow nonlocal exceptions
                for ( int i = 0; i < 5; i += 1 ) {
                    suspend();
                }
            }
        } catch( E ) { . . . }
    }
  public:
    C() { resume(); } // prime loop
    void mem() { resume(); }
};
void uMain::main() {
    C c;
    for ( int i = 0; i < 5; i += 1 ) {
        _Throw E() _At c; // nonlocal exception pending
        c.mem(); // trigger exception
    }
}
```

Figure 4.18: Nonlocal Propagation

terminated and the nonlocal exception of type uBaseCoroutine::UnHandledException is implicitly raised at uMain, since it is c's last resumer. This semantics reflects the fact that the last resumer is most capable of understanding and reacting to a failure of the operation it just invoked. Furthermore, the last resumer is guaranteed to be restartable because it became inactive when it did the last resume. Finally, when the last resumer is restarted, the implicitly raised nonlocal exception is immediately delivered because the context switch back to it *implicitly enables* uBaseCoroutine::UnHandledException, which triggers the propagation of the exception.

A nonlocal exception can be used to affect control flow with respect to *sequential* execution *among* coroutines. That is, a source execution raises an exception at a faulting execution; propagation occurs in the faulting execution. The faulting execution polls at certain points to check for pending nonlocal-exceptions; when nonlocal exceptions are present, the oldest matching exception is propagated, i.e., First-In First-Out (FIFO) service, as if it had been raised locally at the point of the poll. Nonlocal exceptions among coroutines are possible because each coroutine has its own execution-state (stack). For example, in Figure 4.18 coroutine uMain throws a nonlocal exception at coroutine c. Since coroutine control-flow is sequential, the exception type E is not propagated immediately. In fact, the exception can only be propagated the next time coroutine c becomes active. Hence, uMain must make a call to c.mem so mem resumes c and the pending exception is propagated. If multiple nonlocal-exceptions are raised at a coroutine, the exceptions are delivered serially but only when the coroutine becomes active. Note, *nonlocal exceptions are initially turned off for a coroutine*, so handlers can be set up *before* any nonlocal exception can be propagated. Propagation of nonlocal exceptions is turned on via the **_Enable** statement (see Section 5.10.1, p. 130 for complete details). To ensure nonlocal exceptions are enabled, the constructor for C resumes C::main to execute the **_Enable** statement *before* any calls to the coroutine can occur. Since, C::main has started and is suspended in the **for** loop, the nonlocal exception thrown by uMain propagates out of the call to suspend to the catch clause for the exception of type E.

4.10 Summary

There exist a class of problems that experience major control-flow problems when the location of either the input or output is changed by modularizing the code into a routine or class, e.g., series generators, print-formatters, tokenizers, parsers, and iterators. When written as a stand-alone program, these problems can be constructed with the basic control-flow constructs; however, when modularized, the solutions require the ability to remember information between module invocations. Depending on the implementation mechanism used, both data and execution-state can be retain across calls. While it is straightforward to understand how to retain and use saved data (information) between calls, it is more difficult to appreciate the notion of retaining execution state between calls. And yet, it is the ability to preserve the execution state and its associated context between calls that is the more powerful concept, and which significantly simplifies the implementation of this class of problems. The coroutine is one of a small number of programming-language constructs that supports saving both data and execution-state across calls to its members and the coroutine main, and hence, it provides an ideal modularization mechanism for this class of problems.

Coroutines do not appear often in a program, but when needed, they are invaluable. A coroutine cannot be simulated with simpler constructs without violating encapsulation or coding explicit execution states, both of which are unacceptable tradeoffs of the weak equivalence between coroutines and routines or classes. Coroutines are divided into two kinds: semi and full. A semi-coroutine is the most common kind of coroutine. It suspend and resume to and from the coroutine main like a call and return moves to and from a routine. A full coroutine appears infrequently and in esoteric situations, like a device driver in an operating system. It forms a call cycles like a recursive routine, and therefore, is more complex to create and understand. Finally, routines and coroutines may interact freely, i.e., routines may call coroutines, which may call routines, and calls may be recursive within routines or coroutines; all combinations of interactions are possible.

4.11 Questions

1. What feature does a coroutine have that differentiates it from a normal routine?
2. What class of problems is best implemented using coroutines?
3. What is the difference between a full coroutine and a semi-coroutine?
4. What is the special termination rule for coroutines in μC++?
5. Explain the 3 basic phases in all full coroutine programs (also necessary in a semi-coroutine program, too).
6. Both the suspend and resume members are composed of two parts with respect to the actions taken in inactivating and reactivating coroutines. Explain exactly which coroutine is inactivated and which is reactivated for each special coroutine member.
7. Write a *semi-coroutine* with the following public interface (you may only add a public destructor and private members):

```
_Coroutine grammar {
    char ch;                    // character passed by cocaller
    int status;                 // current status of match: 0, 1, 2
  public:
    int next( char c ) {
        ch = c;
        resume();
        return status;
    }
};
```

   which verifies a string of characters matches the language $x_1(x_2x_3)^+x_1$, that is, a character x_1, followed by one or more pairs of characters x_2x_3, followed by the character x_1, where $x_1 \neq x_2$, e.g.:

valid strings	invalid strings
xyzx	x
ababaa	ababa
a b ba	a bba
xyyyyx	xyx
#@%@%@%#	##@#@#

   After creation, the coroutine is resumed with a series of characters from a string (one character at a time). The coroutine returns a value for each character:

- 0 means continue sending characters, may yet be a valid string of the language,
- 1 means the characters form a valid string of the language and no more characters can be sent,
- 2 means the last character resulted in a string not of the language.

After the coroutine returns a value of 1 or 2, it must terminate; sending more characters to the coroutine after this point is undefined.

Write a program grammar that checks if a string is in the above language. The shell interface to the grammar program is as follows:

```
grammar [ filename ]
```

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) If no input file name is specified, input comes from standard input. Output is sent to standard output. *For any specified command-line file, check it exists and can be opened. You may assume I/O reading and writing do not result in I/O errors.*

The program should:

- read a line from the file,
- create a grammar coroutine,
- pass characters from the input line to the coroutine one at time while the coroutine returns 0,
- print an appropriate message when the coroutine returns 1 or 2,
- terminate the coroutine,
- print out result information, and
- repeat these steps for each line in the file.

If there are any additional characters (including whitespace) on a line after the coroutine returns 1 or 2, print an appropriate warning message about the additional characters. For every non-empty input line, print the line and the string yes if the line is in the language and the string no otherwise; print an appropriate warning for an empty input line (i.e., a line containing only ´\n´). The following is some example output:

```
"xyzx" : "xyzx" yes
"ababaa" : "ababaa" yes
"a b ba" : "a b ba" yes
"xyyyyx" : "xyyyyx" yes
"#@%@%@%#" : "#@%@%@%#" yes
"xyyyyxabc" : "xyyyyx" yes -- extraneous characters "abc"
"x" : "x" no
"xab" : "xab" no
"" : Warning! Blank line.
"abab" : "abab" no
"a bba" : "a bb" no -- extraneous characters "a"
"xyx" : "xyx" no
"##@#@#" : "##" no -- extraneous characters "@#@#"
```

8. Write a *semi-coroutine* with the following public interface (you may only add a public destructor and private members):

```
_Coroutine FloatConstant {
  public:
    enum status { MORE, GOOD, BAD }; // possible status
  private:
    status stat;                       // current status of match
    char ch;                           // character passed by cocaller
    void main();                       // coroutine main
  public:
    status next( char c ) {
        ch = c;                        // communication in
        resume();                      // activate
        return stat;                   // communication out
    }
};
```

which verifies a string of characters corresponds to a C++ floating-point constant described by:

floating-constant : $sign_{opt}$ *fractional-constant* $exponent\text{-}part_{opt}$ $floating\text{-}suffix_{opt}$ |
  $sign_{opt}$ *digit-sequence exponent-part* $floating\text{-}suffix_{opt}$
fractional-constant : $digit\text{-}sequence_{opt}$ “.” *digit-sequence* | *digit-sequence* “.”
exponent-part : { “e” | “E” } $sign_{opt}$ *digit-sequence*
sign : “+” | “–”
digit-sequence : *digit* | *digit-sequence digit*
floating-suffix : “f” | “l” | “F” | “L”
digit : “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”

(Where X_{opt} means X | ϵ and ϵ means empty.) In addition, there is a maximum of 16 digits for the mantissa (non-exponent digits) and 3 digits for the characteristic (exponent digits). For example, the following are valid C/C++ floating-point constants:

```
123.456
-.099
+555.
2.7E+1
-3.555E-12
```

After creation, the coroutine is resumed with a series of characters from a string (one character at a time). The coroutine returns a status for each character:

- MORE means continue sending characters, may yet be a valid string of the language,
- GOOD means the characters form a valid floating-point constant but more characters can be sent (e.g., 1., .1)
- BAD means the last character resulted in a string that is not a floating-point constant.

After the coroutine returns a value of GOOD and the string is complete, the next character passed to the coroutine returns a value of BAD. After the coroutine returns a value of BAD, it must terminate; sending more characters to the coroutine after this point is undefined.

Write a program floatconstant that checks if a string is in a floating-point constant. The shell interface to the floatconstant program is as follows:

```
floatconstant [ infile ]
```

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) If no input file name is specified, input comes from standard input. Output is sent to standard output. *For any specified command-line file, check it exists and can be opened. You may assume I/O reading and writing do not result in I/O errors.*

The program should:

- read a line from the file,
- create a FloatConstant coroutine,
- pass characters from the input line to the coroutine one at time while the coroutine returns MORE or GOOD,
- stop passing characters when there are no more or the coroutine returns BAD,
- terminate the coroutine,
- print out result information, and
- repeat these steps for each line in the file.

Assume a *valid* floating-point constant starts at the beginning of the input line, i.e., there is no leading whitespace.

If the end of line is reached while the coroutine is returning MORE, the text is not a floating-point constant. If the end of line is reached while the coroutine is returning GOOD, the text is a floating-point constant. If the coroutine returns BAD, the text is not a floating-point constant. For every non-empty input line, print the line, how much of the line is parsed, and the string yes if it is a valid floating-point constant and the string no otherwise; print an appropriate warning for an empty input line (i.e., a line containing only '\n'). The following is some example output:

```
"+1234567890123456." : "+1234567890123456." yes
"+12.E-2" : "+12.E-2" yes
"-12.5" : "-12.5" yes
"12." : "12." yes
"-.5" : "-.5" yes
".1E+123" : ".1E+123" yes
"-12.5F" : "-12.5F" yes
"" : Warning! Blank line.
"a" : "a" no
"+." : "+." no
" 12.0" : " " no -- extraneous characters "12.0"
"12.0   " : "12.0 " no -- extraneous characters "  "
"1.2.0a" : "1.2." no -- extraneous characters "0a"
"0123456789.0123456E-0124" : "0123456789.0123456" no -- extraneous characters "E-0124"
```

See the C library function isdigit(c), which returns true if character c is a digit.

9. Write a *semi-coroutine* with the following public interface (you may only add a public destructor and private members):

```
_Coroutine Grammar {
  public:
    enum Status { MORE, GOOD, BAD }; // Send more bytes for utf-8 char, valid utf-8, invalid utf-8.
  private:
    Status stat;
    union {
        unsigned char ch;               // Character passed by cocaller.
        struct {                        // Types for first utf-8 byte.
            unsigned char ck : 1;
            unsigned char dt : 7;
        } t1;
        struct {
            unsigned char pd : 2;
            unsigned char ck : 1;
            unsigned char dt : 5;
        } t2;
```

```
            struct {
                // you figure it out
            } t3;
            struct {
                // you figure it out
            } t4;
            struct {                        // Type for extra utf-8 bytes.
                // you figure it out
            } dt;
        } utf8;
    public:
        Status next( unsigned char c ) {
            utf8.ch = c;
            resume();
            return stat;
        }
};
```

that verifies that a string of bytes is a valid UTF-8[1] character. UTF-8 allows any universal character to be represented while maintaining full backwards-compatibility with ASCII encoding, which is achieved by using a variable-length encoding.

The following table provides a summary of the Unicode value ranges in hexadecimal, and how they are represented in binary for UTF-8.

Unicode ranges	UTF-8 binary
000000-00007F	0xxxxxxx
000080-0007FF	110xxxxx 10xxxxxx
000800-00FFFF	1110xxxx 10xxxxxx 10xxxxxx
010000-10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

For example, the symbol £ is represented by Unicode U+00A3 (binary 1010 0011). Since £ falls within the range of U+0080 to U+07FF, it is encoded by the UTF-8 bit string 110xxxxx 10xxxxxx. To fit the character into the eleven bits of the UTF-8 encoding, it is padded on the left with zeroes to 000<u>10100011</u>. The UTF-8 encoding becomes 110<u>00010</u> 10<u>100011</u> when the xs are replaced with the 11-bit binary encoding, which gives the UTF-8 character encoding 0xC2A3 for symbol £.

After creation, the coroutine is resumed with a series of bytes from a string (one byte at a time). The coroutine returns a value for each character:

- MORE means continue sending bytes, may yet be a valid encoding,
- GOOD means the bytes form an encoding and no more bytes can be sent,
- BAD means the last byte resulted in an invalid encoding.

After the coroutine returns a value of GOOD or BAD, it must terminate; sending more bytes to the coroutine after this point is undefined.

Write a program utf8 that checks if a string follows the UTF-8 encoding. The shell interface to the utf8 program is as follows:

```
utf8 [ filename ]
```

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) If no input file name is specified, input comes from standard input. Output is sent to standard output. *For any specified command-line file, check it exists and can be opened. You may assume I/O reading and writing do not result in I/O errors.*

The program should:

[1] Short for *8-bit Unicode Transformation Format*.

- read a line from the file,
- create a Grammar coroutine,
- pass bytes from the input line to the coroutine one at time while the coroutine returns MORE,
- print an appropriate message when the coroutine returns GOOD or BAD,
- terminate the coroutine,
- repeat these steps for each line in the file.

For every non-empty input line, print the bytes in hexadecimal and the string yes if the line is a valid UTF-8 character and the string no, otherwise; print an appropriate warning for an empty input line (i.e., a line containing only ´\n´). If there are any additional bytes on a line after the coroutine returns GOOD or BAD, print an appropriate warning message and the additional bytes in hexadecimal. Hint: to print a character in hexadecimal use the following cast:

```
char ch = 0xff;
cout << hex << (unsigned int)(unsigned char)ch << endl;
```

The following is some example output:

```
0x23 : yes
0x23 : yes. Extra characters 0x23
0xd790 : yes
0xd7 : no
0xc2a3 : yes
 : Warning! Blank line.
0xb0 : no
0xe0e3 : no
0xe98080 : yes
0xe98080 : yes. Extra characters 0xfff8
0xe09390 : no
0xff : no. Extra characters 0x9a84
0xf09089 : no
0xf0908980 : yes
```

10. Write a *semi-coroutine* to sort a set of values into ascending order using a binary-tree insertion method. This method constructs a binary tree of the data values, which can subsequently be traversed to retrieve the values in sorted order. Construct a binary tree without balancing it, so that the values 25, 6, 9, 5, 99, 100, 101, 7 produce the tree:

```
          25
    6           99
5       9           100
    7                   101
```

By traversing the tree in infix order — go left if possible, return value, go right if possible — the values are returned in sorted order. Instead of constructing the binary tree with each vertex having two pointers and a value, build the tree using a coroutine for each vertex. Hence, each coroutine in the tree contains two other coroutines and a value. (The coroutine must be self-contained, i.e., it cannot access any global variables in the program.)

The coroutine must have the following interface (you may only add a public destructor and private members):

```
template<class T> _Coroutine Binsertsort {
    const T EndOfSet;              // Value denoting end of set.
    T value;                       // Communication: value being passed down/up the tree.
    void main();
  public:
    Binsertsort( T EndOfSet ) : EndOfSet(EndOfSet) {}
    void sort( T value ) {         // Pass value to be sorted.
        Binsertsort::value = value;
        resume();
    }
    T retrieve() {                 // Retrieve sorted value.
        resume();
        return value;
    }
};
```

For each value in the set to be sorted, the root coroutine's sort member is called with the value. The end of the set of values is signaled by passing the root coroutine the value EndOfSet; EndOfSet is initialized when the sort coroutine is created. (The EndOfSet is not considered to be part of the set of values.) When the coroutine receives the EndOfSet value, it stops sorting and prepares to return the sorted values one at a time. In retrieve-mode, each call to the root coroutine's retrieve member returns the next value of the sorted set. The last value returned by the root coroutine is EndOfSet, which denotes the end of the sorted set, and then the root coroutine terminates.

Each coroutine in the tree must be identical to the root coroutine, i.e., recursive cocalling, and it behaves as follows. When passed its first value, v, the coroutine stores it in variable pivot. Each subsequent value received by a resume is compared to pivot. If v <= pivot, a Binsertsort coroutine called less is resumed with v; if v > pivot, a Binsertsort coroutine called greater resumed with v. Each of the two coroutines, less and greater, creates two more coroutines in turn. The result is a binary tree of identical coroutines. The coroutines less and greater must not be created by calls to **new**, i.e., no dynamic allocation is necessary in this coroutine. (Note, every recursive algorithm must have a base case to stop the recursion, similar to the condition for stopping a loop. While both a missing or wrong base-case or loop-condition generate infinite execution, a recursive program quickly fails by running out of memory because each recursive call adds to the stack size, whereas an infinite loop never stops executing because it does not consume additional resources.)

When a coroutine receives a value of EndOfSet to indicate end-of-unsorted-values, it resumes its left branch (if it exists) with a value of EndOfSet, prepares to receive the sorted values from the left branch and passes them back up the tree until it receives a value of EndOfSet from that branch. The coroutine then passes up its pivot value. Next, the coroutine resumes its right branch (if it exists) with a value of EndOfSet, prepares to receive the sorted-values from the right branch and passes them back up the tree until it receives a value of EndOfSet from that branch. Finally, the coroutine passes up a value of EndOfSet to indicate end-of-sorted-values and the coroutine terminates. (Note, the coroutine does not print out the sorted values it simply passes them to its resumer.)

Remember to handle the special cases where a set of values is 0 or 1, e.g., EndOfSet being passed as the first or second value to the coroutine. These cases must be handled by the coroutine versus special cases in the main program.

The executable program is named binsertsort and has the following shell interface:

binsertsort *unsorted-file [sorted-file]*

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) Your program must sort non-negative integer values; use -1 for the EndOfSet value.

- If the unsorted input file is not specified, print an appropriate usage message and terminate. The input file contains lists of unsorted values. Each list starts with the number of values in that list. For example, the input file:

```
8 25 6 8 5 99 100 101 7
3 1 3 5
0
10 9 8 7 6 5 4 3 2 1 0
```

contains 4 lists with 8, 3, 0 and 10 values in each list. (The line breaks are for readability only; values can be separated by any white-space character and appear across any number of lines.)

Assume the first number in the input file is always present and correctly specifies the number of following values; assume all following values are correctly formed so no error checking is required on the input data.

- If no output file name is specified, use standard output. Print the original input list followed by the sorted list, as in:

```
25 6 8 5 99 100 101 7
5 6 7 8 25 99 100 101

1 3 5
1 3 5

blank line from list of length 0 (not actually printed)
blank line from list of length 0 (not actually printed)

9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9
```

for the previous input file. End each set of output with a blank line.

Print an appropriate error message and terminate the program if unable to open the given files.

11. Write a semi-coroutine that extracts messages (i.e., byte sequences of a specified format) from a byte stream. The coroutine is resumed for every byte read from the stream. When a complete message has been identified, the coroutine makes the message available to the cocaller and terminates.

A message starts with an STX character (octal value 002), followed by up to 64 bytes of data, an ETX character (octal value 003), and a 2-byte check sum (CHS). For example (all values are in octal):

002,	145,142,147, ... ,135,	003,	164,031
STX	data	ETX	CHS

Characters such as ETX may appear in the message but they will be preceded by an ESC character (octal value 033). When an ESC character is found in the message, it is ignored and the next character is read as is. Messages may be separated by arbitrary bytes; in this assignment you should assume that the bytes between messages are different from STX.

Embed the coroutine in a test program that reads a byte stream from standard input, and writes only the data portions of the identified messages on separate lines to standard output. The test program should check that each message was transmitted correctly by performing the following calculation on the data bytes of the message and comparing the result with the CHS value.

$$\sum_{i=1}^{n} byte_i \quad \text{where } n \text{ is the number of bytes in the message.}$$

Do NOT include control characters STX, ETX, ESC or the CHS bytes in this summation. The only exception to this rule is if one of the control characters is escaped in the data portion of the message. If the message was transmitted incorrectly, print an appropriate error message along with the message.

Do not attempt to print files that contain control characters (e.g., STX, ETX, ESC) directly on the line printer. Use the following command sequence to print files containing control characters:

```
od -av your-test-file | lpr regular printer options BUT NO file name
```

(Approximate size of the sort coroutine is 30-40 lines without comments.)

12. Write a *semi-coroutine* that filters a stream of text. The filter semantics are specified by command-line options.

    This program is logically divided into 3 parts, a reader, a filter (optional), and a writer. The reader and writer are implemented as single coroutines. The filter is implemented as a sequence of coroutines. The coroutines are joined together in a pipeline such that the reader is at the input end of the pipeline, the filter comprises the middle of the pipeline, and the writer is at the output end of the pipeline. Control passes from the reader to the filter to the writer, eventually returning back to the reader. All these coroutines are semi-coroutines.

    (In the following, you may not add, change or remove members from a public interface; you may add a destructor and/or private and protected members.)

    Each filter must inherit from the abstract class Filter:

```
_Coroutine Filter {
  protected:
    static const unsigned char end_filter = '\377';
    unsigned char ch;
  public:
    void put( unsigned char c ) {
        ch = c;
        resume();
    }
};
```

    which ensures each filter has a put routine that can be called to transfer a character along the pipeline.

    The reader reads characters from the specified input file and passes these characters to the first coroutine in the filter:

```
_Coroutine Reader : public Filter {
  public:
    Reader( istream &i, Filter &f );
};
```

    (No coroutine calls the put routine of the reader; all other coroutines have their put routine called.) The reader constructor is given a stream object from which it reads characters, and a filter object that it passes one character at a time from the input. The filter object must therefore be created *before* the reader object.

    The writer is passed characters from the last coroutine in the filter pipeline and writes these characters to the specified output file:

```
_Coroutine Writer : public Filter {
  public:
    Writer( ostream &o );
};
```

    All other filters have the following interface:

```
_Coroutine filter-name : public Filter {
  public:
    filter-name( Filter &f, ... );
};
```

    where "..." is any additional information needed by the filter.

    The pipeline is built by uMain from writer to reader, in reverse order to the data flow. Each newly created coroutine is passed to the constructor of its predecessor coroutine in the pipeline. Control then passes to the reader, beginning the flow of data and control from the reader through the filters to the writer. Specifically, when the reader is created at the head of the pipeline, it starts reading characters and passing them to the first filter coroutine in the pipeline (i.e., the first filter on the command line if there is one) by calling the filter's inherited put members. Each coroutine of the filter is passed characters from the previous coroutine in the pipeline, possibly performing some transformations, and possibly passing characters to the next coroutine in the pipeline. All characters are passed through the pipeline, including control characters (e.g., \n, \t). When the reader reaches end-of-file, it passes the sentinel character `'\377'` through the pipeline and then terminates. Similarly, each coroutine along the filter must pass the sentinel character through the pipeline and then terminate. The writer does not print the sentinel character. The reader coroutine can read characters one at a time or in groups, it makes no difference. The writer coroutine can write characters one at a time or in groups, it makes no difference.

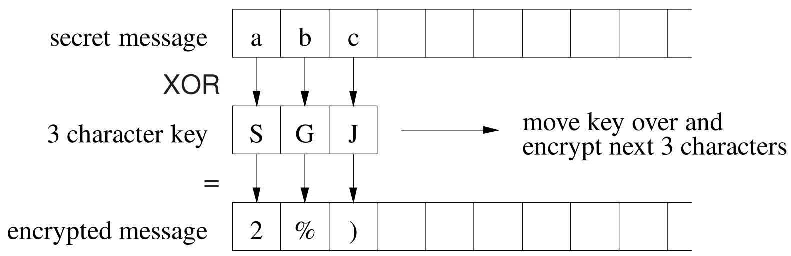
Filter options are passed to the program via command line arguments. For each filter option, create a coroutine that becomes part of the pipeline. If no filter options are specified, then the output should simply be an echo of the input from the reader to the writer. *Assume all filter options are correctly specified and from the list given below, i.e., no error checking is required on the filter options.* Hint: scan the command line left-to-right to locate and remember the position of each option, and then scan the option-position information right-to-left (reverse order) to create the filters with their specified arguments.

The filter options that must be supported are:

-w The *white space* option removes all blanks from the beginning and end of lines, and collapses multiple blank spaces within a line into a single blank space. This option does not affect other white-space characters, such as tab, `'\t'`, and newline, `'\n'`.

-s If the first non-whitespace character (space, tab, newline are whitespace) after a period, question mark, or exclamation point is a lower case letter, the *sentence* option changes the letter to upper case. In other words, this option attempts to capitalize the first letter of every sentence. You must deal with the special case of capitalizing the first letter in the pipeline. This case is special because there is no preceding punctuation character.

-e *key* The encrypt option uses *key* to encrypt the stream. Each byte in the stream is exclusive-or'ed (xor) with successive characters in the key, and the resulting byte is passed along. When the end of *key* is reached, cycle back to the beginning of *key* and continue, e.g.:



This encryption scheme is reversible, so that encrypting an encrypted file produces the original data. Assume -e and *key* are separated by spaces, and spaces cannot occur in *key*. There should be no limit on the length of *key*.

-h The *hex dump* option replaces each character in the stream with a string displaying the character's hex value. For example, the character `'a'` would be replaced with the string 61, and the character `'0'` (zero) would be replaced with the string 30. The hex values must be generated with the necessary spacing and newlines to look like the following:

```
7257 7469   2065 2061   6f63 6f72   7475 6e69
2065 6874   7461 6220   6765 6e69   3a73 2e0a
4d50 4920   760a 696f   2064 7267   6d61 616d
6d50 7473   6964 5c6e   282a
```

Spacing among groups of hex digits is one space, then three spaces, etc. Note, it is possible to convert a character to its hexadecimal value using a simple, short expression.

In addition, design, implement and document one other filter operation that might be useful.

The order in which filter options appear on the command line is significant, i.e., the left-to-right order of the filter options on the command line is the first-to-last order of the filter coroutines. As well, a filter option may appear more than once in a command. Each filter option should be constructed without regard to what any other filters do, i.e., there is no communication among filters using global variables; all information is passed using member put.

The executable program is to be named filter and has the following shell interface:

```
filter [ –filter-options ... ] [ infile [outfile] ]
```

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) If no input file name is specified, input from standard input and output to standard output. If no output file name is specified, output to standard output. *For any specified command-line file, check it exists and can be opened. You may assume I/O reading and writing do not result in I/O errors.*

NOTE: the routine names read and write are used by the UNIX operating system and **cannot** be used as routine names in your program.

13. Write a generic *binary search-tree* and *semi-coroutine* iterators to traverse this tree using an external iterator. The binary search-tree has two parts: the generic tree container and the nodes held within the container.

Each node within the tree container must inherit from the following abstract type:

```
class Treeable {
    friend class TFriend;
    Treeable *parent, *left, *right;
  public:
    Treeable() { parent = left = right = NULL; }
};
```

so it has a parent pointer, and left and right child pointers. The concrete type (T) (derived type) must provide operators:

```
T *T::operator=( T & );
bool operator==( T, T );
bool operator<( T, T );
bool operator>( T, T );
ostream &operator<<( ostream &, T & );
```

which are assignment, equality, greater and less than, and output. These operations are used in copying nodes, inserting and removing nodes from the binary search-tree, and printing the contents of a node. A node can appear in only one tree at a time.

The tree is generic in the nodes it contains:

```
template <class T> class Tree : protected TFriend {
  public:
    Tree();                      // create empty tree
    virtual ~Tree();             // delete tree nodes
    virtual T *top() const;      // return root node of the tree
    virtual void insert( T * );  // insert a node into the tree (do not copy node)
    virtual void remove( T * );  // remove a node from the tree
};
```

The constructor creates an empty tree after allocation, and the destructor deletes any tree nodes so the tree is empty for deallocation. In addition, member top returns the root of the tree, member insert adds a node to the tree (does not copy the node), and member remove removes a node from the tree. Nodes with duplicate values may appear in the tree but a node can appear in only one tree. (Please acknowledge any sources if you copy a binary search-tree algorithm.)

For encapsulation reasons, it is important that users cannot directly access the pointers in Treeable. Normally these fields would be made **private** and the Tree type would be a **friend** of Treeable. However, when Tree is a generic type, there is no single type that can be specified as the friend in Treeable because there are an infinite number of Tree types. One trick around this problem is to create an intermediate type, which is the friend, and the intermediate type supplies access to the private information via inheritance: The intermediate type TFriend:

```
class TFriend {
  protected:
    Treeable *&parent( Treeable *tp ) const {
        return tp->parent;
    }
    Treeable *&left( Treeable *tp ) const {
        return tp->left;
    }
    Treeable *&right( Treeable *tp ) const {
        return tp->right;
    }
};
```

has been used above as the friend of Treeable and its descendants have access to the private information through TFriend's member routines. In this case, the access is total, i.e., both read and write, but the technique can be applied in other cases giving only read or write access as appropriate.

To further ensure abstraction and encapsulation, a iterator is used to traverse the tree so it is unnecessary to know the internal structure of the tree. This abstract tree-iterator is used to create specific iterators for accessing the nodes of a tree:

```
template <class T> _Coroutine treeIter : protected TFriend {
  public:
    treeIter();
    treeIter( const Tree<T> & );
    virtual void over( const Tree<T> & );
    virtual bool operator>>( T *& );
};
```

The first constructor creates a iterator that works with any tree of type Tree<T>. The second constructor creates a iterator that is bound to a specific tree of type Tree<T>. Member over is used to set the tree that is traversed by the iterator (used with the first constructor). Member >> returns the next node in the traversal through its parameter and returns **true** if there are more nodes in the tree or **false** otherwise (i.e., there are no more nodes in the tree).

Create the following specific iterators for the tree:

(a) A preorder tree-iterator coroutine, which returns the tree nodes one at a time using a preorder traversal of the tree:

```
template <class T> _Coroutine preorderTreeIter : public treeIter<T> {
  public:
    preorderTreeIter();
    preorderTreeIter( const Tree<T> & );
};
```

(b) An inorder tree-iterator coroutine, which returns the tree nodes one at a time using an inorder traversal of the tree:

```
template <class T> _Coroutine inorderTreeIter : public treeIter<T> {
  public:
    inorderTreeIter();
    inorderTreeIter( const Tree<T> & );
};
```

(c) A postorder tree-iterator coroutine, which returns the tree nodes one at a time using a postorder traversal of the tree:

```
template <class T> _Coroutine postorderTreeIter : public treeIter<T> {
  public:
    postorderTreeIter();
    postorderTreeIter( const Tree<T> & );
};
```

In addition, create the following non-member routines:

(a) A non-member routine to recursively delete *all* the nodes of the tree:

```
template <class T> static void deleteTree( Tree<T> & );
```

(b) A non-member routine to recursively print *all* the nodes of the tree:

```
template <class T> static void printTree( const Tree<T> & );
```

Use inorder traversal for printing the tree.

(c) A non-member routine to recursively duplicate (copy) *all* the nodes of the tree:

```
template <class T> static void copyTree( Tree<T> &to, const Tree<T> &from );
```

The first parameter is where the copy is made and the second parameter is where the copy comes from. Delete any existing nodes in the first parameter before copying the values from the second.

(d) A non-member routine to determine whether the fringe (leaf nodes) of two trees are the same independent of the tree structure.

```
template <class T> bool sameFringe( const Tree<T> &, const Tree<T> & );
```

(e) Design another non-member routine using the tree iterators to perform some useful action on a tree or trees.

Create appropriate tests illustrating the use of the tree iterators and non-member routines, such as creating one or more trees and manipulating them.

HINT: several casts are necessary from type Treeable to T when using members left and right in TFriend. Why can these be static casts but are guaranteed safe? (Include the answer to this question in the program documentation.)

14. Write a *full coroutine* that plays the following card game. Each player takes a number of cards from a deck of cards and passes the remaining deck to the player on the left if the number of remaining cards is odd, or to the right if the number of remaining cards is even. A player must take at least one card and no more than a certain maximum. The player who takes the last cards wins.

The interface for a Player is (you may only add a public destructor and private members):

```
_Coroutine Player {
    void main();
  public:
    Player( Printer &printer, unsigned int id );
    void start( Player &lp, Player &rp );
    void play( unsigned int deck );
};
```

The constructor is passed a reference to a printer object and an identification number assigned by the main program. (Use values in the range 0 to $N-1$ for identification values.) To form the circle of players, the start routine is called for each player from uMain::main to pass a reference to the player on the left and right; the start routine also resumes the player coroutine to set uMain as its starter (needed during termination). The play routine receives the deck of cards passed among the players.

All output from the program is generated by calls to a printer class. The interface for the printer is (you may add only a public destructor and private members):

```
Players: 5    Cards: 188
P0      P1      P2      P3      P4
                4:184>* 3:181<
                1:180>* 5:175<
        5:168>  2:173<*
3:155<          4:164>  2:162>  4:158>*
5:147<                          3:152>*
1:143<                          3:144>*
3:135<                          5:138>*
1:129<                          5:130>*
4:115<  2:119<  4:121<  2:125<  2:127<*
5:105<                          5:110>*
2:98>*  5:93<                   5:100>
4:89<*                          5:84>
1:83<*                          1:82>
5:77<*                          3:74>
4:70>*  3:67<
3:64>   4:60>*  1:59<
        1:58>   2:56>*  3:53<
                3:50>*  1:49<
        5:40>   4:45<*
                4:36>*  1:35<
                3:32>*  3:29<
                5:24>*  5:19<
3:3<            5:14>   4:10>   4:6>*
                                3:0
```

```
Players: 6    Cards: 133
P0      P1      P2      P3      P4      P5
                        1:128>  4:129<*
                        3:124>  1:127<*
                        3:116>  5:119<*
                        3:110>  3:113<*
                        5:100>  5:105<*
                        1:96>   3:97<*
        3:86>   2:89<*  4:91<   1:95<
                2:84>   4:80>*  1:79<
                1:76>   2:77<*
        1:72>   2:73<*  1:75<
                4:68>*  3:65<
2:57<   4:59<   2:63<           1:52>   4:53<*
                                3:48>   1:51<*
                                3:42>   3:45<*
3:35<                                   4:38>*
3:29<                                   3:32>*
2:26>*  5:21<                           1:28>
2:19<*                                  1:18>
4:14>*  1:13<
5:8>*   1:7<
5:2>    2:0
```

Figure 4.19: Card Game: Example Output

```
class Printer {
  public:
    Printer( const unsigned int NoOfPlayers, const unsigned int NoOfCards );
    void prt( const unsigned int id, const unsigned int took );
};
```

The printer attempts to reduce output by condensing the play along a line. Figure 4.19 shows two example outputs from different runs of the program. Each column is assigned to a player, and a column entry indicates a player's current play "took:remaining<|>", indicating:

a) the number of cards taken by a player,
b) the number of cards remaining in the deck,
c) the direction the remaining deck is passed.

Player information in buffered in the printer until a play would overwrite a buffer value. At that point, the buffer is flushed (written out) displaying a line of player information. An asterisk * marks the prior buffer information for the player causing the buffered data to be flushed. That is, the * on line i appears above the value on line $i+1$ that caused the buffer to be flushed. The buffer is cleared by a flush so previously stored values do not appear when the next player flushes the buffer and an empty column is printed. All output spacing can be accomplished using the standard 8-space tabbing, *so it is unnecessary to build and store strings of text for output*.

The main program plays N games sequentially, i.e., one game after the other, where N is a command line parameter For each game, N players are created, where N is a random value in the range from 2 to 10 inclusive, and one player is passed a deck containing M cards, where M is a random value in the range from 10 to 200 inclusive (see man rand to generate random values). The player passed the deck of cards begins the game, and each player follows the simple strategy of taking C cards, where C is a random value in the range from 1 to 8 inclusive. **Do not spend time developing complex strategies to win.** At the end of each game, it is unnecessary for a player's coroutine main to terminate but ensure each player is deleted before starting the next game.

The executable program is named cardgame and has the following shell interface:

cardgame [*N*]

where *N* is the number of card games to be played and must be greater than or equal to 0. *Assume N is always a valid integer value.* If no value for *N* is specified, assume 5.

15. Write a *full coroutine* that simulates the game of Hot Potato. The game consists of an umpire and a number of players. The umpire starts a *set* by tossing the *hot* potato to a player. What makes the potato *hot* is the timer inside it. The potato is then tossed among the players until the timer goes off. A player never tosses the potato to themself. The player holding the potato when the timer goes off is eliminated. The potato is then given back to the umpire, who resets the timer in the potato, and begins the game again with the remaining players, unless only one player remains, which the umpire declares the winner.

    The potato is the item tossed around by the players and it also contains the timer that goes off after a random period of time. The interface for the Potato is (you may only add a public destructor and private members):

```
class Potato {
  public:
    Potato( unsigned int maxTicks = 10 );
    void reset( unsigned int maxTicks = 10 );
    bool countdown();
};
```

    The constructor is optionally passed the maximum number of ticks until the timer goes off. The potato chooses a random value between 1 and this maximum for the number of ticks. Member reset is called by the umpire to re-initialize the timer to reuse the potato. Member countdown is called by the players, and returns **true** if the timer has gone off and **false** otherwise. Rather than use absolute time to implement the potato's timer, make each call to countdown be one tick of the clock.

    The interface for a Player is (you may only add a public destructor and private members):

```
_Coroutine Player {
    void main();
  public:
    typedef ... PlayerList; // container type of your choice
    Player( Umpire &umpire, unsigned int Id, PlayerList &players );
    unsigned int getId();
    void toss( Potato &potato );
};
```

    The type PlayerList is an STL (standard template library) container of your choice, containing all the players still in the game. The constructor is passed the umpire, an identification number assigned by the main program, and the container of players still in the game. The member getId returns a player's identification number. If a player is not eliminated while holding the *hot* potato, then the player chooses another player, excluding itself, at random from the list of players and tosses it the potato using its toss member.

    The interface for the Umpire is (you may only add a public destructor and private members):

```
_Coroutine Umpire {
    void main();
  public:
    Umpire( Player::PlayerList &players );
    void set( unsigned int player );
};
```

    The umpire creates the potato. Its constructor is passed the container with the players still in the game. When a player determines it is eliminated, i.e., the timer went off while holding the potato, it calls set, passing its player identifier so the umpire knows the set is finished. The umpire removes the eliminated player from the list of players, but does not delete the player because it might be used to play some other game. Hence, player coroutines are created and deleted in the main program. The umpire then resets the potato and tosses it to a randomly selected player to start the next set; this toss counts with respect to the timer in the potato.

The executable program is named hotpotato and has the following shell interface:

```
hotpotato n
```

where *n* is the number of players in the game and must be between 2 and 20, inclusive. *Assume n is always a valid integer value.* The main program creates the umpire, the *n* players, with player identifiers from 0 to *n*–1, and the container holding all the players. It then starts the umpire by calling set with a player identifier of 0.

Make sure that a coroutine's public methods are used for passing information to the coroutine, but not for doing the coroutine's work.

The output should show a dynamic display of the game, i.e., the set number, each player taking their turn (Id identifies a player), and who gets eliminated for each set and the winning player. Output must be both concise and informative, e.g.:

```
5 players in the match
  POTATO will go off after 7 tosses
Set 1:   U -> 1 -> 2 -> 1 -> 3 -> 4 -> 1 -> 0 is eliminated
  POTATO will go off after 10 tosses
Set 2:   U -> 3 -> 1 -> 3 -> 1 -> 3 -> 1 -> 2 -> 3 -> 1 -> 3 is eliminated
  POTATO will go off after 1 toss
Set 3:   U -> 1 is eliminated
  POTATO will go off after 8 tosses
Set 4:   U -> 2 -> 4 -> 2 -> 4 -> 2 -> 4 -> 2 -> 4 is eliminated
2 wins the Match!
```

16. A *ring election algorithm* is used in a distributed system to ensure one process is elected as a *coordinator* if the current coordinator is no longer responding. (A coordinator can be used to implement various specialized work, such as managing deleted resources in a distributed application.) For simplicity, a process in this application is represented by a full coroutine. Assume all of the coroutines are identical, distinguishable only by their identification number (id), so there is only one type of full coroutine. An id number is used to impose an ordering upon the coroutines, and the coroutines are structured into a ring. The coroutines pass values around the ring; when a value is received, it is increment before passing it to the next coroutine (partner).

    A coroutine has a 1 in 10 chance of failing. When a coroutine determines its partner has failed, the coordinator is passed the failed coroutine, which it deletes and returns a new partner. When a coroutine determines the coordinator has failed, it starts an election by raising a resumption exception in the coroutine *after* the coordinator (i.e., the coordinator's partner). When a coroutine receives an election exception, it compares its id with the coroutine's id in the exception, and the coroutine with the larger id is placed in an exception that is raised at the next coroutine. Once the exception has completely circulated the ring, the coroutine with the highest id is assigned as the new coordinator.

    Write a *full coroutine* with following interface (you may add only a public destructor and private members):

```
_Coroutine Process {
  public:
    _ResumeEvent Election {
      public:
        Process *newCoordinator;
        Election( Process *newCoordinator ) : newCoordinator(newCoordinator) {}
    };
    static Process *coordinator;            // shared among all instances
    Process( unsigned int id );
    bool alive();                           // check for failure
    Process *remove( Process *victim );     // delete failed object and return new partner
    void start( Process *partner );         // supply partner
    int data( int value );                  // pass value and receive new one
    void vote( Process *newCoordinator );   // perform election voting
```

```
  private:
    void main() {
        Handler handler( this );          // resumption handler functor
        try <Election, handler> {         // establish resumption handler for election
            _Enable {                     // allow delivery of nonlocal exceptions
                suspend();                // establish starter for termination
                . . .                     // YOU WRITE THIS CODE
            } // _Enable
        } // try
    } // Process::main
}; // Process
```

To form the ring of players, uMain::main calls the start member for each player to pass a pointer to the coroutine's partner; the start routine also resumes the coroutine to set uMain as its starter (needed during termination). The main routine of each coroutine suspends back immediately so the next coroutine can be started. After uMain initializes the ring, it sets the coordinator pointer to a randomly selected coroutine and calls it data member with the value zero. A coroutine can query its partner via the alive method to see if it has failed, where **false** indicates the coroutine has failed. If the partner has not failed, the coroutine passes it an incremented value via the data method. After the call to data, a coroutine determines if it has failed. If a coroutine's partner has failed and it is not the coordinator, the remove method is called in the coordinator via the coordinator pointer to delete the failed coroutine and return its partner. Note, this partner may also be failed, so the check must be repeated. If a coroutine's partner has failed and it is the coordinator, an election is started by calling the vote method in the coordinator's partner. The vote method raises an Election exception in the partner and performs any other actions for exception delivery. You define the type Handler used for handling the resumption exception.

The executable program is to be named election and has the following shell interface:

```
election P
```

where P is the number of coroutines in the ring. Provide appropriate error-checking that P is provided and is a positive integer greater than 1. For output purposes, P is restricted to be less than eleven. Terminate the program with an appropriate error message if P is missing or invalid.

Output must show a dynamic display of the ring behaviour, e.g.:

```
0:0 1:1 2:2 3:3 4:4 5:5 6:6 7:7 8:8 9:9 0:10 1:11 2:12 3:13 4:14 5:15 6:16 7:17 8:18 9:19 |1|
0->2 0:20 2:21 3:22 4:23 5:24 6:25 7:26 8:27 **E** 9 9?2 9?3 9?4 9?5 9?6 9?7 9?8 **C** 9 |0|
9->2 9:28 |3|
2->4 |4|
2->5 2:29 5:30 6:31 7:32 8:33 9:34 2:35 5:36 6:37 7:38 8:39 9:40 2:41 5:42 6:43 |8|
7->9 7:44 9:45 |5|
2->6 2:46 6:47 7:48 9:49 2:50 6:51 7:52 9:53 2:54 6:55 7:56 9:57 2:58 |7|
6->9 6:59 9:60 2:61 **E** 6 6?2 **C** 6 |9|
6->2 6:62 2:63 6:64 2:65 6:66 2:67 |2|
6->6 no more partners
```

The first line shows the "id:value" for each coroutine as execution moves around the ring for a 10 process ring. At the end of the first line, coroutine 1 has failed "|1|" and coroutine 0 calls the coordinator to delete it. At the start of the second line, coroutine 0 indicates its new partner is coroutine 2 "0->2". More values are passed around the ring until coroutine 9 detects coroutine 0 has failed and it is the coordinator; hence, an election is started `**E**` by coroutine 9. The election shows the "id?id" used for each comparison around the ring, until the new coordinator `**C** 9` is selected. Note, the coroutine starting an election may not be the one elected as the new coordinator. Finally, coroutine 9 calls the coordinator to delete coroutine 0. At the start of line 3, coroutine 9 indicates its new partner is coroutine 2 "9->2", and more values are passed around the ring until the next failure. Try to keep your output similar to the above to aid in marking, but you may add some additional elements as long as each is fully explained.

17. Write a *full coroutine* called merge that works with another instance of itself to merge together two sorted arrays of integer values in ascending order, producing a third sorted array. Duplicate values may appear in the merged

arrays.

The merge coroutine has the following public interface (you may add only a public destructor and private members):

```
_Coroutine merge {
  public:
    merge();
    void start( merge *partner, int sortValues[ ], int mergedValues[ ] );
    void mergeTo( int limit, int nextPos );
};
```

The start member is used to initialize the coroutine and has the following parameters: a pointer to its partner coroutine, one of the sorted arrays to be merged, and the third array into which the merged values are placed. Assume no data value in the sorted arrays is greater than or equal to the sentinel value 99999999. Assume the sentinel value can be placed at the end of the sorted array in merge, and that mergedValues is large enough to hold the values from the two sorted arrays. Note, inserting the sentinel value 99999999 at the end of the sort values simplifies the merge algorithm **but do not copy it into mergedValues**.

The main program reads the necessary data, creates two instances of merge and then calls the start member for each. The start member saves its parameter values but does not resume the coroutine. The main program then compares the first values in each of the sorted arrays, and then calls the mergeTo member of the instance of merge that is passed the array containing the smaller value, passing it the larger of the first array values and 0, respectively. The mergeTo member resumes the coroutine. The merge that is resumed now copies from its sorted array into the merged array all values that are less than or equal to the limit. When this is complete, the current merge calls the mergeTo member of the other instance, passing the value that caused the current instance to stop merging, i.e., the first value in its array that is greater than limit) and the first empty location in mergedValues. The second merge then does exactly what the first merge did, i.e., copy values from its sorted array into mergedValues until a value is greater than limit. The two merges pass control back and forth until both have copied all their sorted values into the merged array. Remember, when a merge has completed copying all its values, it must allow the other merge to complete its copying. Depending on which merge finishes first, one of the merges may not have finished its main member when control returns to uMain::main; this is fine as that merge is deleted when uMain::main terminates. The main program then prints the result of the merge from the mergedValues array.

HINT: The merge algorithm is extremely simple; if you write a substantial number of lines of code to do the merge operation itself, something is wrong.

The executable program is named merge and has the following shell interface:

merge *sorted-infile1 sorted-infile2 [merged-outfile]*

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.)

- If the two sorted input files are not specified, print an appropriate usage message and terminate. A input file contains a list of sorted values. Each list starts with the number of values in that list. For example, the input file:

  8 -5 6 7 8 25 99 100 101

  contains a list with 8 values in it. (The placement of numbers above is for readability only; values can be separated by any white-space character and appear across any number of lines.) Both lists of sorted values from each file are to be read into arrays. You must handle an arbitrary number of values in a list. (HINT: GNU C++ allows arrays to be dynamically dimensioned.)

  Assume the first number in the input file is always present and correctly specifies the number of following values; assume all following values are correctly formed and sorted so no error checking is required on the input data.

  The UNIX jot and sort commands may be useful in preparing test data.
- If no merge output file name is specified, use standard output. The format of the output is the same as the input file so that the output of one merge can be used as the input to another.

For any specified command-line file, check it exists and can be opened. You may assume I/O reading and writing do not result in I/O errors.

NOTE: there is an existing UNIX command called `merge` so ensure you invoke your program and not the UNIX one.

Chapter 5

Concurrency

While coroutines are important in their own right, they also introduce one of the basic forms of control flow found in concurrent programming, which is, suspending execution and subsequently resuming it while retaining state information during the suspension. What differentiates concurrency from coroutining is multiple execution points in a program that behave independently of one another.

In the past, coroutines have been *associated* with concurrency:

> In essence, coroutines are concurrent processes in which process switching has been completely specified, rather than left to the discretion of the implementation. [AS83, p. 8]

However, coroutines execute with a single point of execution, and hence, coroutining is *not* concurrency. While coroutining and concurrency share the ability to retain execution state between calls, coroutining never experiences the same problems nor is there the same power as concurrency. In essence, once task switching is completely specified rather than left to the discretion of the implementation, there is no longer independent execution, and hence, neither concurrency nor processes.

This chapter begins with general information on concurrency and several definitions needed in further explanations, starting with these:

- A **thread** is an independent sequential execution path through a program. Conceptually, a thread has at one end the point of execution that is moving through the program, and the path taken by that point of execution forms a trail through the program. This analogy can be likened to a "thread of consciousness", which is the train of thoughts that lead to an idea or answer. Often a thread is illustrated by a needle with a thread trailing behind it showing where the point of the needle has been in the program. The most important property of a thread, and the aspect producing concurrency, *is scheduling execution separately and independently among threads*.

- A **process** is a program component that combines a thread and an execution state into a single programming language or operating system construct.

- A **task** is similar to a process except it is reduced along some dimension (like the difference between a boat and a ship: one is physically smaller than the other). It is often the case that a process is an operating system construct having its own memory, while a task is a programming language construct that shares a common memory with other tasks. A task is sometimes called a light-weight process or LWP, which emphasizes the reduced stature.

- **Parallel execution** is when two or more operations occur simultaneously, which can *only* occur when multiple processors (CPUs) are present. It is the threads of control associated with processes and tasks, executing on multiple processors, that results in parallel execution. Hence, without multiple processors there can be no parallel execution.

- **Concurrent execution** is any situation in which execution of multiple processes or tasks *appears* to be performed in parallel. Again, it is the threads of control associated with processes and tasks that results in concurrent execution.

μC++ provides tasks, while an operating system, such as UNIX, provides processes. Since μC++ is used in this book, the term task is used unless there is an important reason to distinguish between a process and a task. Nevertheless, all of the discussion concerning a task applies equally to a process (although the reverse is not necessarily true).

5.1 Why Write Concurrent Programs

In general, concurrent programs are more complex to write and debug than an equivalent sequential program. The main reason to write a concurrent program is to decrease the execution time of a program through parallelism. However, this is only possible when there are multiple processors. Interestingly, it may be the case that making a program concurrent *increases the cost of execution*, just like modularizing a program into multiple routines might increase execution cost because of additional routine call overhead. Therefore, for certain problems, a sequential program is appropriate; for other problems, dividing it into multiple executing tasks may be the natural way of creating a solution. What is crucial is that an algorithm expressed concurrently can immediately take advantage of any available hardware parallelism. On the other hand, a sequential program can never take advantage of available parallelism. Thus, in certain situations, it is reasonable to spend the time and effort to construct parallel algorithms and write concurrent programs to implement them in anticipation of available hardware parallelism, otherwise much of the advantage of new multiprocessor computers is unrealized.

5.2 Why Concurrency is Complex?

Concurrency is inherently complex; no amount of "magic" can hide the complexity. If you read or hear that concurrent programming can be made as simple as sequential programming, *don't believe it*! Information and complexity theory show that it is impossible for this to be true. Here are some reasons why:

understanding: While people can do several things concurrently and in parallel, the number is small because of the difficulty in managing and coordinating them. This fact is especially true when things interact with one another. Hence, reasoning about multiple streams of execution and their interactions is much more complex than sequential reasoning.

specifying: How can/should a problem be broken up so that parts of it can be solved at the same time as other parts? How and when do these parts interact or are they independent? If interaction is necessary, what information must be communicated during an interaction? Do certain interactions interfere with one another?

debugging: Concurrent operations proceed at varying speeds and in non-deterministic order, hence execution is not entirely anticipatable nor repeatable.

An example to illustrate some of these problems is moving furniture out of an apartment. In general, moving requires multiple people (parallelism) and lots of coordination and communication. If the moving truck is rented by the hour and pizza has to be purchased for all the helpers, there is incentive to maximize the parallelism with as few helpers as possible.

Understanding:

- How many helpers?

  The number of helpers depends on how many friends you have or how much money you have or both, for example, 1,2,3, ... N helpers, where N is the number of items of furniture. With N helpers, each person picks up one item and moves it, which is highly parallel. Is there any point in having more than N helpers? Yes, because some items are large and need multiple people to carry them; in fact, this is usually the reason why a single person cannot move on their own.

- How many trucks?

  Like helpers, it is conceivable to have a truck for each item, assuming each helper can also drive, but such a scheme is impractical. Another consideration is that a single large truck might be better than many small trucks, but more expensive to hire. Alternatively, a single small truck might require a large number of trips, slowing down the entire moving process. A medium truck or a couple of small trucks might be the most efficient and cost effective form of transportation.

Specifying:

- Who does what and when?

  Given a set of M helpers, who does what and when? All workers may not be equal, so there may be restrictions on certain workers with respect to certain jobs. For example, the person that drives the truck must know how to drive and have a driver's license. Heavy items should usually be moved by the strongest people to avoid damage to the item, the house/apartment, and the people carrying the item. As well, the order in which items are moved can make a difference depending on the size and shape of the truck, and who is available to carry them.

- Where are the bottlenecks?

  A bottleneck is interference resulting from interaction among tasks or the resources the tasks are using; it often imposes a maximum on the amount of possible concurrency. The term **bottleneck** refers to pouring liquid out of a bottle: a liquid cannot be poured faster than the size of the bottle's neck, regardless of how large the bottle or what is in the bottle. Some examples of bottlenecks are the door out of a room, items in front of other items, and large items. Imagine if there are N helpers, and each picks up an item and they all rush to the door at the same time. Clearly, the more helpers, the more coordination is required among them. While it is true that some helpers can work in small groups and others can work alone, there must be some overall coordination or major problems arise at the bottlenecks.

- What communication is necessary among the helpers?

  During the course of the move, communication is used as the mechanism to provide coordination. Orders are given to specify which items to take next, that some items are fragile and need special care, and that big items need several helpers working together. The amount of communication should be as brief and concise as possible; if helpers are communicating, they are probably not working.

Debugging:

- How to detect and deal with problems?

  Many problems can occur during a move. What differentiates a move from a program is that problems must be dealt with during the move. In the case of a program, problems are dealt with in the next run of the program; only rarely is a program debugged during its execution and allowed to continue. Still, in the case of moving, understanding problems in the current move can help prevent the same problems occurring in the next move. (Experience is recognizing when you have made the same mistake, twice.)

  However, many problems are move specific and timing specific. Even given the same move done multiple times, different problems can occur. For example, two people might arrive simultaneously at a doorway in one move but not in another, or the sofa is not in front of the stove in one move but it is in another, or some communication is lost in one move but not in another. In other words, no two moves are identical (even from the same location), and similarly no two runs of a concurrent program are identical. Therefore, even after carefully planning what appears to be a problem free move, a problem might appear because of a minute difference in timing at some point in the move that is apparently unrelated to the problem situation. Therefore, guaranteeing a problem free move requires understanding of the dynamics of the move over time not just a static description of the move algorithm. The same is true for concurrent programs.

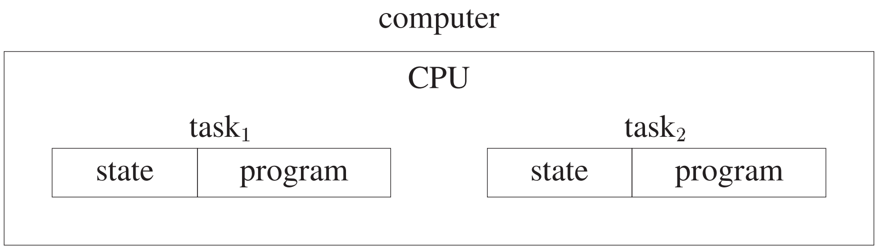
  It is these kinds of problems that make specifying concurrent algorithms, and subsequently testing and debugging them difficult and frustrating. One peculiar situation is called a "Heisenbug", named after the German physicist Werner Karl Heisenberg. For example, when a print statement is added to a failing concurrent program to determine what is wrong, the program works; when the print statement is removed, the program continues to fail. In this situation, the act of observing the concurrent system changes the timing of the program in a subtle way so the error case does not occur. In certain difficult cases, the only way to find such an error is to perform thought experiments to try to imagine the case that is failing.

As this example illustrates, managing and coordinating concurrency and parallelism is difficult but essential in certain situations. In effect, a good manager is someone who can deal with high levels of concurrency and parallelism among employees; a good concurrent programmer must also be a good manager.

5.3 Concurrent Hardware

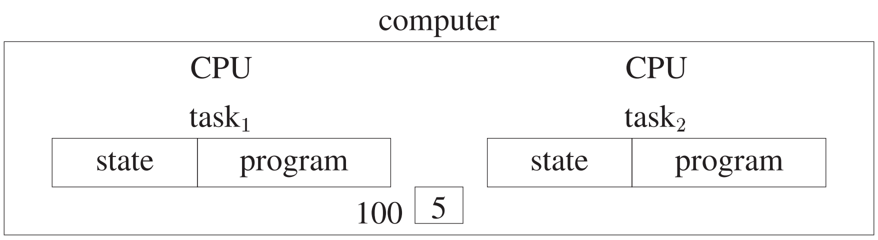
Hardware structure dictates several aspects of concurrency. While many esoteric hardware designs exist, only three major structures are examined, as they cover almost all designs.

1. Concurrent execution of tasks is possible on a computer that has only one CPU, called a **uniprocessor**.



   There can be no parallelism in this structure because there is only CPU. But parallelism can be simulated by rapidly switching the CPU back and forth among tasks at non-deterministic program locations providing concurrency. Switching involves a context switch from task to task, which saves the state of the currently executing task and restores the state of the next task so it can execute. Different names for this technique are **multitasking** for multiple tasks, **multiprocessing** for multiple processes, or **pseudo-parallelism** for either scheme. Unlike coroutining, there is no control over where or when a process or task is suspended and another resumed. In fact, it is this phenomenon that introduces all the difficulties in concurrent programs; programs must be written to work regardless of non-deterministic ordering of program execution. Switching is usually based on a timer interrupt that is independent of program execution; that is, after a period of time, called a **time-slice**, a task can be interrupted between any two instructions and control switches to another task. Hence, the execution of a task is not continuous; rather it is composed of a series of discrete time-slice intervals.

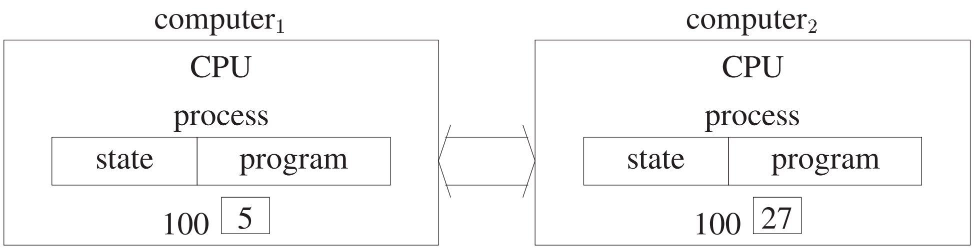
2. True parallelism is only possible when a computer has multiple CPUs, called a **multiprocessor**. In this structure, the CPUs share the same memory, called a **shared-memory multiprocessor**:



   Because memory is shared, tasks can pass both data and addresses to one another during communication. What is crucial is that an address can be dereferenced by another CPU and get the same data. For example, if one task executing on a particular CPU passes the memory address 100 to another task executing on another CPU, both tasks can dereference the address 100 and get the same memory location in the shared memory, and hence, access the same value.

   Most computers use this structure for input/output (I/O). For example, a disk controller often has its own CPU and can directly read and write memory, called **direct memory access** (**DMA**). Because there are multiple CPUs, there can be parallelism between program execution and I/O operations. Therefore, virtually all computers are in fact multiprocessors with other CPUs hidden in I/O devices. However, because these additional CPUs are not available for running user programs, these computers are uniprocessor from the user's perspective but multiprocessor from the hardware and operating system perspective.

3. Finally, different computers have separate CPUs and separate memories:



Processes or tasks that run on different computers run in parallel with each other. If there is a connection between the computers, communication among them is possible; a system of interconnected computers has non-shared memory, called a **distributed system**. While non-shared memory systems have the same concurrency issues as shared, they have the additional problem of being unable to dereference addresses passed from another computer. For example, if one task executing on one computer passes the memory address 100 to another task executing on another computer, the data at location 100 is probably different on both computers. Therefore, either all data must be passed by value among computers or addresses must be specially marked, so when dereferenced, data is fetched from the computer that the address originated from. Thus, linked data structures, like a list or tree, are very difficult to manipulate on a distributed system.

Finally, it is possible to combine the different hardware structures, such as a distributed system composed of uniprocessor and multiprocessor computers.

It is sufficient to examine the first case, which is the simplest, as it generates virtually all of the concurrency issues and problems that arise with the latter two cases. Furthermore, μC++ provides support for only the first two cases. Additional problems associated with distributed systems are often orthogonal to concurrency and are discussed in Chapter 15, p. 417.

5.4 Specifying Concurrency

There are two major approaches for specifying concurrency in a program: attempting to *discover* concurrency in an otherwise sequential program, for example, by parallelizing loops and access to data structures [BGS94], and providing concurrency through *explicit* constructs, which a programmer uses to build a concurrent program. The former approach is extremely appealing because a programmer continues to write sequential programs and they are automatically made concurrent. Furthermore, existing sequential programs, regardless of their age, can be automatically converted as well (often referred to as parallelizing "dusty decks[1]"). Unfortunately, there is a fundamental limit to how much parallelism can be found and current techniques only work on certain kinds of programs. In general, concurrency needs to be explicit to solve all the different kinds of concurrency problems. While both implicit and explicit approaches are complementary, and hence, can appear together in a single programming language, the limitations of the implicit approach require some form of explicit approach always be available to achieve maximum concurrency. Currently, μC++ only provides an explicit approach, but nothing in its design precludes adding an implicit approach as well.

Within the explicit approaches, some languages provide a single technique or paradigm (see Chapter 13, p. 359) that must be used to solve all concurrent problems. While a particular paradigm may be very good for solving certain kinds of problems, it may be awkward or preclude other kinds of solutions. Therefore, a good concurrent system must support a variety of different concurrency paradigms, while at the same time not requiring the programmer to work at too low a level. As stated before, there is no simple solution to concurrency that requires little or no work on the part of the programmer; as the amount of concurrency increases, so does the complexity to express and manage it. *There is no free lunch*!

5.5 Basic Concurrent Programming

It is now time to introduce concurrent programming, one of the most difficult forms of control flow to understand and master. Concurrent programming requires the ability to specify the following 3 mechanisms in a programming language.

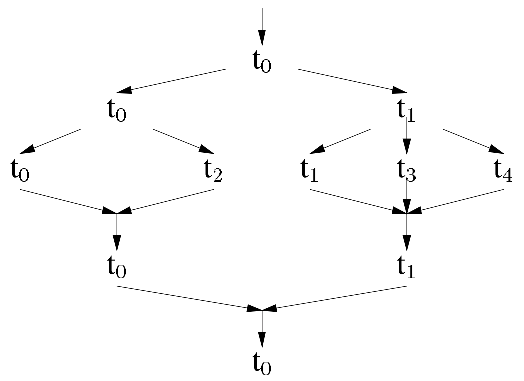
[1] The "deck" part refers to the time when programs were punched onto cards making the stack of cards for a program a "deck" of cards. The "dusty" part simply means the card deck is old enough to have gathered dust.

1. thread creation and termination;
2. thread synchronization;
3. thread communication.

Each of these mechanisms is discussed in detail.

5.5.1 Thread Graph

It is possible to visualize the creation/termination of threads graphically by a **thread graph**. The purpose of the graph is to show the lifetime of a thread. New thread creation is represented by forks in the graph, and termination is represented by joins in the graph, as in:



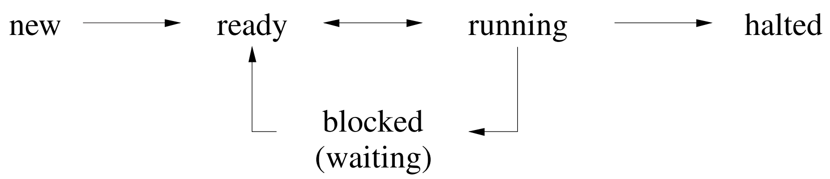
Starting at the top (root) of the graph with an initial thread for task t_0, t_0 creates a new task t_1 with its own thread. The creation is denoted by the fork in the graph with t_0 continuing execution along the left branch and t_1 starting along the right branch. Task t_0 then creates a new task, t_2, with its own thread, and both tasks proceed. Similarly, task t_1 creates two new tasks, t_3 and t_4, and all three tasks proceed. At some point in time, independent of the length of the arcs in the graph, tasks t_0 and t_2 join (synchronize), task t_2 terminates, and task t_0 proceeds. Similarly, tasks t_1, t_3 and t_4 join, tasks t_3 and t_4 terminate, and task t_1 proceeds. Finally, tasks t_0 and t_1 join, task t_1 terminates, and task t_0 proceeds.

Thread graphs are useful for understanding the basic structure of a concurrent system. Knowing the creation and termination relationships among tasks defines this basic structure. As mentioned, the arcs in the graph do not represent absolute time, only relative time with respect to creation and termination. Additional graphs are presented that display other kinds of important information in a concurrent system. While these kinds of graphs are not used extensively, either in this book or in practical concurrent programming, they can be extremely useful at this early stage of understanding.

5.5.2 Thread States

As mentioned, a thread does not execute from start to end in an uninterrupted sequence; a thread may be interrupted multiple times during its execution, possibly to share the CPU with other threads, but there may be other reasons, too. Even a sequential program very seldomly executes from start to end without interruption because most operating systems are multiprocessing. Hence, the operating system interrupts each sequential program at various points during it execution and switches to another, which in turn may be another sequential program or a concurrent program containing multiple threads, each of which may be interrupted during their execution. Because of these interruptions, a thread goes through multiple state transitions during its execution, and it is important to discuss the states and transitions briefly as they are relevant in further discussion.

At any moment in its lifetime, a thread is in one of 5 states



new – When a thread is created it enters the system in the new state. In this state, the system initializes all aspects of the thread, such as allocating storage, loading executable code, etc., before execution begins.

ready – After a thread has been initialized, it is now ready for execution when a CPU becomes available. While waiting for an available CPU, the thread is in the ready state. A ready thread is not executing, it only has the *potential* to execute.

running – When a CPU begins executing a ready thread, the thread's state changes to running and its execution state to active (see Section 4.1.3, p. 67). As long as a thread is executing, its state is running. There can only be as many running threads as there are CPUs.

blocked – If a thread has to wait for another thread or a user to type in input data, it may block itself so that other threads may execute. In these situations, a thread voluntarily gives up the CPU because it cannot make further progress at this time and its state changes to blocked. When a thread is blocked, it is not eligible for execution until it is explicitly marked as ready, which only happens when the event that the thread is waiting for has occurred.

halted – When a thread finishes execution, either successfully or unsuccessfully, it is moved to the halted state. At this time, the system cleans up all outstanding aspects of the thread, such as freeing storage, closing files, printing output, etc. Only then can the thread be deallocateed.

As mentioned, transitions of state are initiated in response to events, such as:

timer alarm – When a timer expires it indicates that it may be time to switch to another thread. Notice that the timer is running in parallel with the execution of the current thread. In this case, a thread moves directly from the running to the ready state. The thread is not blocked because it can still continue to execute if it had a CPU; it has simply been moved back to the ready state to wait its turn for access to a CPU.

I/O completion – For example, a user finishes typing and presses the "Return" or "Enter" key to complete a read from a terminal. If a thread has blocked itself waiting for an I/O completion, the subsequent completion changes the thread's state to ready. Notice that after the event's completion the thread is only ready to execute; it must always wait its turn for access to a CPU.

exception during execution – If a program exceeds some limit (CPU time, etc.) or generates an error, it is moved from the running to the halted state and the system cleans up after it (e.g., close files, free storage, etc.).

A thread may experience thousands of these transitions during its execution depending on what it is doing and how long it executes. Understanding a thread's states and the transitions among states is important in understanding concurrent execution. While an operating system/programming language may make most transitions largely invisible in sequential programs, this is not the case for concurrent programs. Therefore, a programmer has to be aware of a thread's states and transitions.

5.6 Thread Creation/Termination: START/WAIT

The first and most important aspect of concurrent programming is how to start a new thread of control. The two ingredients needed are a new thread and a location for the new thread to start execution. A common approach is to have a construct that creates a new thread and the thread's starting location is a routine, which is called by the thread. In theory, there is no reason to restrict the starting point of a thread, for example, a thread could start in the middle of an expression. In practice, there are practical considerations for the implementation that preclude such generality. (Think of restricting the starting point of a thread like restricting the target of a GOTO statement.) A routine is usually chosen as the thread starting point because it is a well defined building block in program composition and is an equally well defined unit of execution. Since any segment of code can be modularized into a routine, there is no fundamental restriction on the code a thread may execute; however, a new name has to be created and code has to be restructured into a routine.

A fictitious START statement[2] is introduced to illustrate thread creation before discussing how to accomplish the equivalent in μC++. The START statement has the form:

[2] Another common name for this statement is "fork" because the calling thread splits into two threads after executing it. However, the name "fork" already has existing meaning and connotations in UNIX for starting a process.

```
START routine-call-list
```

where START creates a new thread for each routine call in the list, and the names of the routines in the list specify the starting points for each new thread along with any arguments needed by a routine, e.g.:

```
PROGRAM p
    PROCEDURE p1(...) ...
    FUNCTION f1(...) ...
  BEGIN
    START p1(5);   // execute p1 concurrently with statement s1
    s1             // continue execution, do not wait for p1
    START f1(8);   // execute f1 concurrently with p1 and statement s2
    s2             // continue execution, do not wait for p1 or f1
```

How many threads are created after the second START? In fact, there are three: one associated with program p, the initial thread given to the program when it starts, and the two new ones running in p1 and f1, respectively.

The most important point to understand is that, once the threads are started, their execution with respect to one another is no longer under strict programmer control. After the first START, the thread associated with p or p1 may continue next or both threads may continue simultaneously, if the program is running on a multiprocessor. If only one thread is running, the other is in the ready state waiting its turn to become running. (Devise a situation where both threads are in the ready state.) After the second START, the same phenomenon is true between p's, p1's, and f1's thread. In fact, p's thread may get all the way to the end of s2 with neither p1's or f1's thread having started execution. Many other possible execution sequences may occur, where each thread takes several turns being blocked, ready and running, so that the execution of s1 and s2 by p's thread and p1 and f1 by the other two threads are interleaved many times. This apparent lack of control during execution is often disconcerting to many programmers and is one of the most difficult aspects of concurrent programming to come to grips with. In fact, much of the subsequent discussion involves controlling the order of execution.

5.6.1 Termination Synchronization

It is rare in concurrent programming to start a task and then ignore it. An example might be a task that prints out some values; however, such a thread has a side-effect independent of the program's execution. Normally, a task is working on behalf of another task, and it returns results to be used in further computations. The creator of a new thread needs to know, at the very minimum, when the computation is done, and where to obtain the result of the computation.

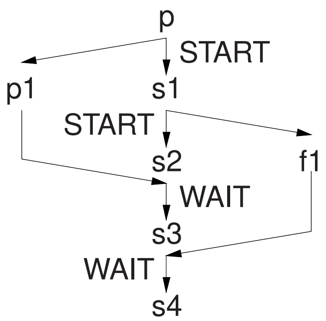
A fictitious WAIT statement is introduced to illustrate waiting for another thread to finish execution. The WAIT statement has the form:

```
WAIT expression-list
```

where WAIT blocks the executing thread until the thread(s) associated with the named routines have finished execution and terminated, for example:

```
PROGRAM p
    PROCEDURE p1(...) ...
    FUNCTION f1(...) ...
  BEGIN
    START p1(5);   // execute p1 concurrently with statement s1
    s1             // continue execution, do not wait for p1
    START f1(8);   // execute f1 concurrently with p1 and statement s2
    s2             // continue execution, do not wait for p1 or f1
    WAIT p1;       // wait for p1 to finish
    s3             // continue execution, do not wait for f1
    WAIT i := f1;  // wait for f1 to finish, retrieve result
    s4
```

which has the following thread graph:



(This call graph is slightly simplified to reduce its size by removing p's name after each task start and join along the central branch.)

Notice the names p1 and f1 appear in the WAIT statement without arguments because a routine call is now broken into two parts. The first part appears in the START, starting the call and specifying any necessary arguments, the second part appears in the WAIT, waiting for completion of the call, and where appropriate, retrieving a result. This division indicates the temporal nature of the call and that it is possible to perform other work between the two parts. In a sequential program, these two aspects are combined because the caller waits for the result. Finally, when control reaches a WAIT statement it may not have to wait at all if the specified routine(s) has already finished.

When two or more threads wait for an event to occur, and an event can be virtually anything, this is called **synchronization**. A construct such as WAIT performs **termination synchronization** because the event is the termination of the thread executing in the specified routine. Once the thread executing the WAIT is synchronized with the other thread's termination, it is possible to perform additional operations, such as fetching a result. Notice the synchronization is crucial because it is incorrect to try to obtain the result earlier than the thread's termination. The next few chapters discuss this point in more detail, and show how to synchronize and communicate during execution of two threads, not just when a thread terminates.

Finally, the astute reader has probably noticed a problem with the design of the WAIT statement. This problem is similar to that identified for coroutines: creation of multiple instances of the same coroutine (see Section 4.1.2, p. 67). With the current definition of START it is possible to write:

```
START p1(5), p1(10), f1(8), f1(16);
```

which creates new four threads, two start executing in routine p1 and two start executing in f1. However, the WAIT statement is inadequate to handle this situation, as:

```
WAIT p1, i := f1;
```

is ambiguous—which of the two threads executing in p1 or f1 is to be waited for? This deficiency is fixed in the discussion of how to achieve the equivalent of START and WAIT in μC++.

5.7 μC++ Threads

So as not to interrupt the START/WAIT example with a detailed explanation of the μC++ task at this time, a basic task definition is given followed by a general transformation of START/WAIT into μC++. More details are presented in Section 5.9, p. 127.

A μC++ task type has all the properties of a **class**. The general form of the task type is the following:

```
_Task task-name {
  private:
    . . .               // these members are not visible externally
  protected:
    . . .               // these members are visible to descendants
    void main();        // task main (distinguished member)
  public:
    . . .               // these members are visible externally
};
```

A task is an object created from a task type in which a new thread of control is started in its distinguished member routine; this distinguished member is named main and is called the **task main**. Like a coroutine main, execution of a task main may be suspended as control leaves it, only to carry on from that point when control returns at some later time; however, the mechanism to accomplish this is normally quite different and not discussed until Chapter 10, p. 317. It is distinguished because it has special properties that none of the other member routines have and only the member routine called main in a μC++ task has special properties.

Like a coroutine, a task main has its own execution state, which is the state information needed to retain information while the task main is suspended. In practice, an execution state consists of the data items created by a task, including its local data and routine activations, and a current execution location, which is initialized to a starting point (member main). However, unlike a coroutine, which waits for a thread to resume it, a new thread is implicitly created and begins execution in the task main after a task is instantiated.

Instead of allowing direct interaction with main, its visibility is usually **private** or **protected**. The decision to make the task main **private** or **protected** depends solely on whether derived classes can reuse the task main or must supply their own. Hence, a user interacts with a task indirectly through its member routines. This approach allows a task type to have multiple public member routines to service different kinds of requests that are statically type checked. A task main cannot have parameters or return a result, but the same effect can be accomplished indirectly by passing values through the coroutine's global variables, called **communication variables**, which are accessible from both the coroutine's member and main routines. Like a routine or class, a task can access all the external variables of a C++ program and variables allocated in the heap. Also, any **static** member variables declared within a task are shared among all instances of that task type.

For example, given the simple task type definition:

```
_Task T {
    void main() { ... }          // distinguished member
  public:
    T() { ... }
};
T t;      // new thread created, which starts execution in T::main
```

the declaration of t can be broken into the following parts:

```
T t;              // storage allocated
t();              // implicitly invoke appropriate constructor
START t.main();   // start new thread in task main
```

Since declarations can appear anywhere in a C++ block, a new thread can be started anywhere a START statement can be located. The new thread starts in a routine, albeit a member routine for μC++. It is true that having the starting routine embedded in a class-like definition is more complex for simple cases. However, it will be shown in Chapter 10, p. 317 that the class-like definition is critical for communication purposes. *For now, all tasks are restricted to having only public constructors and a destructor; all other member routines are* **private** *or* **protected**. Additional **public** member routines are deferred until Chapter 10, p. 317.

To transform the statement START p(4) to μC++ requires creating a task with its main member calling p(4), as in:

```
_Task T {
    void main() { p( 4 ); }
};
T t;              // create thread and execute T::main
```

A thread can also be created dynamically by allocating a task in the heap, as in:

```
T *t = new T;     // create thread and execute T::main
```

While creating threads this way may seem awkward in comparison to the START statement, it turns out that this is true only for simple examples; more complex examples (i.e., real life problems) are simpler to specify using a task.

Termination synchronization is achieved through an indirect mechanism rather than a WAIT statement. The indirect mechanism is variable deallocation. Since all tasks are created through some form of declaration, they must subsequently be deallocated, at which time their storage is freed. For tasks allocated in a block (i.e., {...}), the task is implicitly deallocated at the end of the block; for a dynamically allocated task, the task is explicitly deallocated by the **delete** statement. However, it is unreasonable for a task's destructor to deallocate storage if a thread is executing within it; otherwise, the deallocated storage might be reused for another object but the thread would continue to change it! Therefore, a block cannot end until the threads for all the tasks declared in it have finished execution, and a **delete** statement cannot continue until the thread of the deleted task finishes execution. In both cases, the thread executing the block or **delete** synchronizes with the termination of any threads being deallocated. For example, the deallocation of t can be broken into the following parts:

```
WAIT t.main;    // wait for thread to terminate
~t();           // invoke destructor
FREE t          // deallocate storage
```

It is now possible to write the START/WAIT example above in μC++:

```
int i;                      // return variable
_Task T1 {
    void main() { p1(5); }
};
_Task T2 {
    void main() { i = f1(8); }
};
void uMain::main() {
    T1 *p1 = new T1;    // start a T1 running
    ... s1 ...
    T2 *f1 = new T2;    // start a T2 running
    ... s2 ...
    delete p1;          // wait for p1 to finish
    ... s3 ...
    delete f1;          // wait for f1 to finish
    ... s4 ...
}
```

By using **new** and **delete** it is possible to obtain the same fine-grain control of thread creation and termination as with START and WAIT. However, the same cannot be said about returning a result. Is the μC++ program the same as the START/WAIT program, with regard to assignment of variable i? The answer is no. In the START/WAIT program, i is updated only after the second WAIT statement; therefore, its value is unaffected in statements s2 or s3, unless it is changed directly in that code. In the μC++ program, i can change at anytime after f1 is assigned because there is no control over the order and speed of execution of tasks. Thus, it is unsafe to use the value of i until after f1's thread has terminated. If i is used, the result of the program is non-deterministic, which means the program may produce different answers each time it is run. This problem is solved in subsequent chapters.

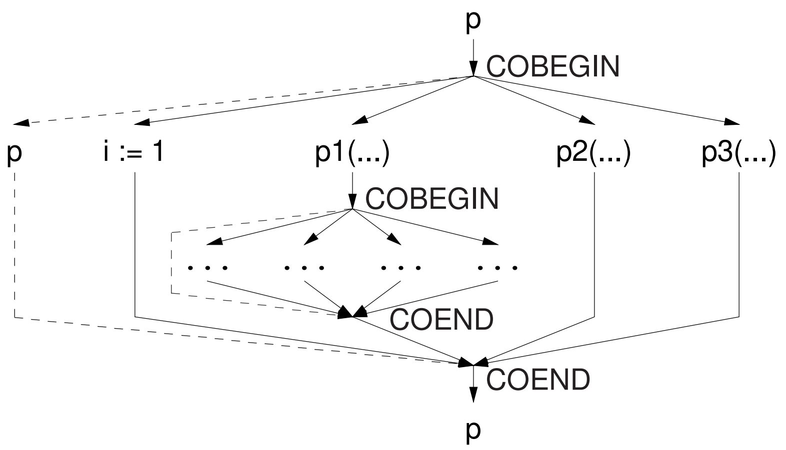
5.8 Thread Creation/Termination: COBEGIN/COEND

Another mechanism for starting threads is a fictitious COBEGIN ... COEND statement[3], where CO stands for concurrent. COBEGIN creates a block out of a statement list, just like BEGIN; the difference is that each statement is executed concurrently. That is, a new thread is created for each statement, which starts execution at the beginning of the statement and terminates at the end of the statement. Furthermore, the thread that starts the COBEGIN blocks at the COEND until all the threads created in the block have terminated, for example:

```
PROGRAM p
    PROCEDURE p1(...) ...
    PROCEDURE p2(...) ...
    PROCEDURE p3(...) ...
BEGIN
    COBEGIN   // each statement in the block starts executing concurrently
        i := 1;
        p1(5);   // order and speed of thread execution is unknown
        p2(7);
        p3(9);
    COEND     // all threads finish before control reaches here
END           // p's thread continues
```

which has the following thread graph:

[3] Another common name for this statement is PARBEGIN ... PAREND where PAR stands for parallel. However, this name is misleading because there is no parallelism when the program is run on a uniprocessor computer, only concurrency.



The fact that p's thread is implicitly blocked during the COBEGIN is illustrated in the graph by a dashed line. What can be inferred from the thread graph about routine p1?

This example can be rewritten using START and WAIT as follows:

```
PROGRAM p
    PROCEDURE p0() BEGIN  i := 1  END
    PROCEDURE p1(...) ...
    PROCEDURE p2(...) ...
    PROCEDURE p3(...) ...
BEGIN
    START p0(), p1(5), p2(7), p3(9)
    WAIT p0, p1, p2, p3
END
```

Notice, a new routine, p0, is created for the assignment statement. While this additional routine increases the complexity of the program, it is usually the case that a single assignment or even small groups of assignments do not benefit from concurrency because of concurrency overhead. That is, the cost of creating a new thread is more expensive than the cost of executing the code by the thread. For concurrency to achieve a performance benefit, the amount of work performed by a thread must be significantly greater than the administrative cost of the concurrency. This phenomenon is not specific to concurrency, but is true for managing any work.

By using the fact that a block can only continue after all threads declared in it have terminated, it is possible to write the COBEGIN/COEND example above in μC++:

```
int i;                          // return variable
_Task T1 {
    void main() { i = 1; }
};
_Task T2 {
    void main() { p1(5); }
};
_Task T3 {
    void main() { p2(7); }
};
_Task T4 {
    void main() { p3(9); }
};
void uMain::main() {
    { // COBEGIN
        T1 t1; T2 t2; T3 t3; T4 t4;
    } // COEND
}
```

The block in uMain::main declares 4 different tasks, which implicitly start execution. The thread of uMain::main then immediately attempts to deallocate the 4 tasks but must wait until the threads of these tasks have terminated. As well, routine p1 might look like this:

```
void p1(...) {
    ...
    { // COBEGIN
        T5 t5; T6 t6; T7 t7; T8 t8;
    } // COEND
    ...
}
```

The main drawback of COBEGIN is that it is difficult to create a dynamically determined number of new threads of control. For example, if the number of threads to be created is read in interactively, it is necessary to use a recursive routine to construct a dynamic number of nested scopes, as in:

```
PROGRAM p
    PROCEDURE p1(...) ...
    PROCEDURE loop( N : INTEGER ) BEGIN
        IF N ~= 0 THEN
            COBEGIN
                p1(5);          // routine to be executed concurrently
                loop( N - 1 );  // start another COBEGIN
            COEND
        END IF
    END
BEGIN
    READ( N );          // read number of threads
    loop( N );          // dynamically create threads
END
```

This effect can be accomplished more easily in μC++ by creating an array of N tasks, as in:

```
{
    T t[N];
} // wait for N tasks to terminate
```

Another drawback of COBEGIN is that it can only create a tree (more formally a lattice) of processes, where START and WAIT can create arbitrarily complex thread graphs. For example, it is impossible to transform the previous START/WAIT example directly into COBEGIN/COEND, and maintain the same amount of concurrency. (Give it a try.) Finally, concurrency is inhibited because the thread that executes the COBEGIN is blocked during its execution, and therefore, is prevented from doing other work.

5.9 Task Details

The following μC++ task details are presented now to understand subsequent examples or to help in solving questions at the end of the chapter. More μC++ task details are presented in Chapter 10, p. 317.

5.9.1 Task Creation and Destruction

A task is the same as a class with respect to creation and destruction, as in:

```
_Task T {
    void main() ...       // task main
};
T *tp;                    // pointer to a T
{ // start a new block
    T t, ta[3];           // local creation
    tp = new T;           // dynamic creation
    ...
} // wait for t, ta[0], ta[1] and ta[2] to terminate and then deallocate
...
delete tp;    // wait for tp's instance to terminate and then deallocate
```

When a task is created, the appropriate task constructor and any base-class constructors are executed in the normal order by the creating thread. The stack component of the task's execution-state is created and the starting point (activation point) is initialized to the task main routine visible by the inheritance scope rules from the task type. Then a new thread of control is created for the task, which begins execution at the main routine. From this point, the creating thread executes concurrently with the new task's thread; main executes until its thread blocks or terminates. The location of a task's variables—in the task's data area or in member routine main—depends on whether the variables must be accessed by member routines other than main.

A task terminates when its main routine terminates. When a task terminates, so does the task's thread of control and execution state. A task's destructor is invoked by the deallocating thread when the block containing the task declaration terminates or by an explicit **delete** statement for a dynamically allocated task. Because storage for a task cannot be deallocated while a thread is executing, a block cannot terminate until all tasks declared in it terminate. Similarly, deleting a task on the heap must also wait until the task being deleted has terminated.

While a task that creates another task is conceptually the parent and the created task its child, μC++ makes no implicit use of this relationship nor does it provide any facilities based on this relationship. Once a task is declared it has no special relationship with its declarer other than what results from the normal scope rules.

Like a coroutine, a task can access all the external variables of a C++ program and the heap. However, because tasks execute concurrently, there is the general problem of concurrent access to such shared variables. Furthermore, this problem also arises with **static** member variables within a task that is instantiated multiple times. Therefore, it is suggested that these kinds of references be used with extreme caution.

5.9.2 Inherited Members

Each task type, if not derived from some other task type, is implicitly derived from the task type uBaseTask, e.g.:

```
_Task task-name : public uBaseTask {
    . . .
};
```

where the interface for the base class uBaseTask is:

```
_Task uBaseTask : public uBaseCoroutine { // inherits from coroutine base type
  public:
    uBaseTask();
    uBaseTask( unsigned int stacksize );

    void yield( unsigned int times = 1 );
    enum State { Start, Ready, Running, Blocked, Terminate };
    State getState() const;
};
```

The public member routines from uBaseCoroutine are inherited and have the same functionality.

The overloaded constructor routine uBaseTask has the following forms:

uBaseTask() – creates the task with the default stack size (same as uBaseCoroutine()).

uBaseTask(**int** stackSize) – creates the task with the specified stack size (in bytes) (same as uBaseCoroutine(**int** stackSize)).

Multiple-task execution-states have the same storage management problem as for coroutines (see Section 4.6.2, p. 82). Therefore, programmers have to be vigilant to ensure there is sufficient stack space for the calls made by the thread of a task. As for coroutines, this is not usually an issue because call depth is shallow so the default stack is normally adequate. A task type can be designed to allow declarations to specify the stack size by doing the following:

```
_Task T {
  public:
    T() : uBaseTask( 8192 ) {};            // default 8K stack
    T( int s ) : uBaseTask( s ) {};        // user specified stack size
    . . .
};
T x, y( 16384 );       // x has an 8K stack, y has a 16K stack
```

The member routine yield gives up control of the CPU to another ready task the specified number of times. For example, the routine call yield(5) ignores the next 5 times the task is scheduled for execution. If there are no other ready tasks, the delaying task is simply delayed and restarted 5 times. Member yield allows a task to relinquish control when it has no current work to do or when it wants other ready tasks to execute before it performs more work. For example, when creating a large number of tasks, it may be important to ensure some of them start execution before all the tasks are created. If after the creation of several tasks the creator yields control, some created tasks then have an opportunity to begin execution before the next group of tasks is created. This facility is not a mechanism to control the exact order of execution of tasks, only to ensure some additional fairness among tasks. Finally, one task *cannot* yield another task; a task can only yield itself. A call such as:

```
x.yield(...)
```

fails, via a dynamic check, if task x is not the same as the currently executing task.

The member routine getState returns the current state of the task, which is one of the enumerated values uBaseTask::Ready, uBaseTask::Running, uBaseTask::Blocked or uBaseTask::Terminate.

The routine:

```
uBaseTask &uThisTask();
```

is used to determine the identity of the task executing this routine. Because it returns a reference to the base task type, uBaseTask, of the current task, this reference can only be used to access the public routines of type uBaseTask and uBaseCoroutine. For example, a routine can verify the stack or yield execution of the calling task by performing the following:

```
uThisTask().verify();
uThisTask().yield();
```

As well, printing a task's address for debugging purposes must be done like this:

```
cout << "task:" << &uThisTask() << endl; // notice the ampersand (&)
```

5.10 Concurrent Propagation

A local exception within a task is the same as for an exception within a routine or class. An exception raised and not handled inside a task performs the C++ default action of calling terminate, which must abort. As mentioned, a nonlocal exception between a task and a coroutine is the same as between coroutines (sequential). A concurrent exception between tasks is more complex due to the multiple threads.

A concurrent exception can be handled locally within a task, or nonlocally among coroutines, or concurrently among tasks. All concurrent exceptions are nonlocal, but nonlocal exceptions can also be sequential. Concurrent exceptions provide an additional kind of communication over a normal member call. That is, a concurrent exception can be used to force a communication when an execution state might otherwise be computing instead of accepting calls. For example, two tasks may begin searching for a key in different sets; the first task to find the key needs to inform the other task to stop searching, e.g.:

```
_Task searcher {
    searcher &partner;        // other searching task
    void main() {
        try {
            _Enable {
                ...           // implicit or explicit polling is occurring
                if ( key == ... )
                    _Throw stop() _At partner;  // inform partner search is finished
            }
        } catch( stop ) { ... }
```

Without this control-flow mechanism, both tasks have to poll for a call from the other task at regular intervals to know if the other task found the key. Concurrent exceptions handle this case and others.

When a task performs a concurrent raise, it blocks only long enough to deliver the exception to the specified task and then continues. Hence, the communication is asynchronous, whereas member-call communication is synchronous. Once an exception is delivered to a task, the runtime system propagates it at the soonest possible opportunity. If multiple concurrent-exceptions are raised at a task, the exceptions are delivered serially.

5.10.1 Enabling/Disabling Propagation

μC++ allows dynamic enabling and disabling of nonlocal exception-propagation. The constructs for controlling propagation of nonlocal exceptions are the **_Enable** and the **_Disable** blocks, e.g.:

```
_Enable <E1> <E2> ... {          _Disable <E1> <E2> ... {
    // code in enable block              // code in disable block
}                                    }
```

The arguments in angle brackets for the **_Enable** or **_Disable** block specify the exception types allowed to be propagated or postponed, respectively. Specifying no exception types is shorthand for specifying all exception types. Though a nonlocal exception being propagated may match with more than one exception type specified in the **_Enable** or **_Disable** block due to exception inheritance (see Sections 3.14.2, p. 55 and 3.17, p. 57), it is unnecessary to define a precise matching scheme because the exception type is either enabled or disabled regardless of which exception type it matches with.

_Enable and **_Disable** blocks can be nested, turning propagation on/off on entry and reestablishing the delivery state to its prior value on exit. Upon entry of a **_Enable** block, exceptions of the specified types can be propagated, even if the exception types were previously disabled. Similarly, upon entry to a **_Disable** block, exceptions of the specified types become disabled, even if the exception types were previously enabled. Upon exiting a **_Enable** or **_Disable** block, the propagation of exceptions of the specified types are restored to their state prior to entering the block.

Initially, nonlocal propagation is disabled for all exception types in a coroutine or task, so handlers can be set up before any nonlocal exceptions can be propagated, resulting in the following μC++ idiom in a coroutine or task main:

```
void main() {
    // initialization, nonlocal exceptions disabled
    try {                   // setup handlers for nonlocal exceptions
        _Enable {           // enable propagation of all nonlocal exception-types
            // rest of the code for this coroutine or task
        }                   // disable all nonlocal exception-types
    } catch ...             // catch nonlocal exceptions occurring in enable block
    // finalization, nonlocal exceptions disabled
}
```

Several of the predefined kernel exception-types are implicitly enabled in certain contexts to ensure their prompt delivery (see Section 3.18.1, p. 60).

The μC++ kernel implicitly polls for nonlocal exceptions after a coroutine/task becomes active, and at the start of a **_Enable** block after enabling nonlocal exception-propagation. If this level of polling is insufficient, explicit polling is possible by calling:

```
bool uEHM::poll();
```

For throwable exceptions, the return value from poll is not usable because a throwable exception unwinds the stack frame containing the call to poll. For resumable exceptions, poll returns **true** if a nonlocal resumable-exception was delivered and **false** otherwise. In general, explicit polling is only necessary if pre-emption is disabled, a large number of nonlocal exception-types are arriving, or timely propagation is important.

5.11 Divide-and-Conquer

Divide-and-conquer is a technique that can be applied to certain kinds of problems. These problems are characterized by the ability to subdivide the work across the data, so that the work can be performed independently on the data. In general, the work performed on each group of data is identical to the work that is performed on the data as a whole. In some cases, groups of data must be handled differently, but each group's work is identical. Taken to the extreme, each data item is processed independently, but at this stage, the administration of concurrency usually becomes greater than the cost of the work. What is important is that only termination synchronization is required to know when the work is done; the partial results can then be processed further if necessary.

A simple example of divide-and-conquer is adding up the elements of an $N \times N$ matrix. The elements can be added up by row or by column or by blocks of size $M \times M$, where $M < N$. A task can be created for each row, column or block, and work independently of the other tasks. Totalling the subtotals is performed after the tasks terminate;

```
_Task Adder {
    int *row, size, &subtotal;                      // communication

    void main() {
        subtotal = 0;
        for ( int r = 0; r < size; r += 1 ) {       // sum row of matrix
            subtotal += row[r];
        }
    }
  public:
    Adder( int row[ ], int size, int &subtotal ) : row( row ), size( size ), subtotal( subtotal ) {}
};
void uMain::main() {
    const int rows = 10, cols = 10;
    int matrix[rows][cols], subtotals[rows], total, r;
    Adder *adders[rows];

    // read matrix

    for ( r = 0; r < rows; r += 1 ) {               // create tasks
        adders[r] = new Adder( matrix[r], cols, subtotals[r] );
    }
    for ( r = 0; r < rows; r += 1 ) {               // wait for termination
        delete adders[r];
    }
    total = 0;
    for ( r = 0; r < rows; r += 1 ) {               // total all subtotals
        total += subtotals[r];
    }
    cout << total << endl;
}
```

Figure 5.1: Concurrent Summation of Matrix Elements

hence, only termination synchronization is needed. Figure 5.1 divides a matrix up into rows, and has tasks sum the rows independently.

uMain::main creates the matrix of values, and the array where the subtotals generated by the tasks are placed. Also, an array is created to hold the pointers to the tasks summing each row. No code is shown for reading the matrix, just a comment to that effect. The first loop dynamically allocates a task for each row of the matrix, passing to each task the address of the row it is to sum, the number of columns in the row, and the address in the result array to place the total for the row. The next loop attempts to delete each of the newly created tasks. It does not matter what order the Adder tasks terminate because the **delete** waits for each one to finish before deallocating its storage. Sometimes the **delete** must wait and other times it can do the deallocation immediately because the task is already finished. After the deletion loop completes, the thread of uMain has synchronized with all the Adder tasks; hence, it knows that all the row subtotals have been calculated and the subtotal values are stored in their respective locations. Then, and only then, can the subtotals be summed; the final loop then sums the subtotals to generate the total.

The task main of an Adder task is straightforward, adding up the elements of the row supplied by the constructor, and then terminating.

5.12 Synchronization and Communication During Execution

Up to now tasks only synchronize and communicate when one task terminates. There are many cases where tasks need to synchronize and communicate *during* their lifetime, e.g., in a producer/consumer situation (see Section 4.8, p. 88). This section shows that such communication can be done using the facilities discussed so far, although not very well. How to do this properly is discussed in subsequent chapters.

The program in Figure 5.2 shows unidirectional communication between two tasks, prod and cons, during their

```
int data;                                    // shared storage for communication
enum { full, empty } status = empty;         // status of data in shared storage

_Task Prod {
    int NoOfTrials;
    void main() {
        for ( int i = 1; i <= NoOfTrials; i += 1 ) {
            while ( status == full ) {};     // busy wait
            ::data = i;                      // transfer data
            status = full;                   // indicate availability of data
            // perform other work
        }
    }
  public:
    Prod( int NoOfTrials ) : NoOfTrials(NoOfTrials) {}
};
_Task Cons {
    int NoOfTrials;
    void main() {
        int data;
        for ( int i = 1; i <= NoOfTrials; i += 1 ) {
            while ( status == empty ) {};    // busy wait
            data = ::data;                   // remove data
            status = empty;                  // indicate removal of data
            // perform other work
        }
    }
  public:
    Cons( int NoOfTrials ) : NoOfTrials(NoOfTrials) {}
};
void uMain::main() {
    Prod prod(5); Cons cons(5);
}
```

Figure 5.2: Synchronization and Communication During Execution

lifetime. When the producer wants to transfer data, the data is copied into shared storage, i.e., storage visible to both tasks. However, just placing data into the shared variable is insufficient. For communication, one task has to be ready to transmit the information and the other has to be ready to receive it, *simultaneously*. Otherwise data may be transmitted when no one is receiving, or received before it is transmitted. For example, the producer can insert data into the shared storage before the previous data has been copied out by the consumer, or the consumer can copy the same data out of the shared storage multiple times before the next data is inserted by the producer.

To prevent these problems a **protocol** is established between the producer and consumer. The protocol conceptually has the producer raise a flag when data is copied into the shared storage, and the consumer drops the flag when the data is removed; thus, by appropriate checking of the flag by the producer and consumer, it is possible to safely transfer data. In the example program, when the producer wants to transmit data, it first checks if the flag is up indicating there is still data in the shared storage. If the flag is up, i.e., status is full, the producer must wait; in this case, the method for waiting is to spin in a loop, called **busy waiting**, until the other task executes. If the flag is down, the data is copied into shared storage and the flag is raised by setting the status to full. The consumer, which may have started execution before the producer, does not attempt to remove data until the flag goes up; this is implemented by busy waiting until the flag is up, i.e., status's value is *not* empty. Once the producer changes status to full the consumer exits its busy loop and copies out the data; then it drops the flag by setting status to empty. The protocol is repeated for the next transfer of data. The main problem with this approach is the busy waiting by the producer and consumer; it is never good concurrent programming style to busy wait. Subsequent chapters gradually eliminate busy waiting.

5.13 Summary

Concurrency is inherently complex because of the management of simultaneous actions. All concurrency systems must provide mechanisms to create new threads of control, synchronize execution among threads, and communicate after synchronizing. These mechanisms vary with the particular concurrency systems. In μC++, instantiating a **_Task** object implicitly creates a new thread of control and termination of this object forces synchronization, which can be used as a point for communication. While communication among tasks during their lifetime is essential for complex concurrent programs, an efficient mechanism to accomplish this is postponed until later. Temporarily, **_Task** definitions will not have public members, other than constructors and possibly a destructor, until further material is presented.

5.14 Questions

1. A single interrupt (event) occurs in a multitasking system. As a result of this interrupt, one process is moved from the running state to the ready state, and another is moved from the blocked state to the running state. Explain the likely reasons for these two state changes. What can be assumed concerning the task scheduling strategy?

2. Extend the program in Figure 5.2 to perform bidirectional communication between the tasks.

3. Type in the following program, exactly as is, compile it using the u++ command, and run the program 10 times (see the repeat command described in the manual entry for csh).

```
#include <uC++.h>
#include <iostream>
using namespace std;

int shared = 0;

_Task increment {
    void main() {
        for ( int i = 1; i <= 5000000; i += 1 ) {
            shared += 1;
        } // for
    }
  public:
    increment() {}
};

void uMain::main() {
    {
        increment t[2];
    } // wait for tasks to finish
    cout << "shared:" << shared << endl;
} // uMain::main
```

   Must all 10 runs produce the same result? What results did you actually get (include some output)? In theory, what are the smallest and largest values that could be printed out by this program? Explain your answers. (**Hint:** the obvious answer is wrong.)

4. Quicksort is one of the best sorting algorithms in terms of execution speed on randomly arranged data. It also lends itself easily to concurrent execution by partitioning the data into those greater than the pivot and those less than the pivot so each partition can be sorted independently and concurrently by another task.

   Write a concurrent quicksort with the following public interface (you may add only a public destructor and private members):

```
_Task QuickSort {
  public:
    QuickSort( int values[ ], int low, int high );
};
```

that sorts an array of non-unique integer values into ascending order. Choose the pivot as follows:

```
pivot = values[low + ( high – low ) / 2];
```

The quicksort partitions the data values as normal, but instead of recursively invoking quicksort on each partition, start a new quicksort task, to handle one partition, and continue sorting the other partition in the current quicksort task. Do not use other sorting algorithms for small partitions as some versions of quicksort do. (Please acknowledge any sources if you copy a quicksort algorithm.)

To maximize efficiency, quicksort tasks must not be created by calls to **new**, i.e., no dynamic allocation is necessary. (Hint, think of recursion as a way to add new variables to the stack.)

The executable program is named quicksort and has the following shell interface:

quicksort *unsorted-file [sorted-file]*

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.)

- If the unsorted input file is not specified, print an appropriate usage message and terminate. The input file contains lists of unsorted values. Each list starts with the number of values in that list. For example, the input file:

  ```
  8 25 6 8 –5 99 100 101 7
  3 1 –3 5
  0
  10 9 8 7 6 5 4 3 2 1 0
  ```

  contains 4 lists with 8, 3, 0 and 10 values in each list. (The line breaks are for readability only; values can be separated by any white-space character and appear across any number of lines.)

  Assume the first number in the input file is always present and correctly specifies the number of following values; assume all following values are correctly formed so no error checking is required on the input data.
- If no output file name is specified, use standard output. Print the original input list followed by the sorted list, as in:

  ```
  25 6 8 –5 99 100 101 7
  –5 6 7 8 25 99 100 101

  1 –3 5
  –3 1 5

  blank line from list of length 0 (not actually printed)
  blank line from list of length 0 (not actually printed)

  9 8 7 6 5 4 3 2 1 0
  0 1 2 3 4 5 6 7 8 9
  ```

  for the previous input file. End each set of output with a blank line.

Print an appropriate error message and terminate the program if unable to open the given files.

5. Merge sort is one of several sorting algorithms that take optimal time (to within a constant factor) to sort N items. It also lends itself to parallel execution - after a list is split into two, each half list can be sorted concurrently without interfering with tasks working on the other half list.

   (a) Write a concurrent mergesort with the following public interface (you may add only a public destructor and private members):

   ```
   _Task MergeSort {
       // Precondition: values[low..high] may have duplicates
       // Postcondition: values[low..high] are sorted in ascending order
     public:
       MergeSort( int values[], int low, int high );
   };
   ```

that sorts an array of non-unique integer values into ascending order. The mergesort partitions the data values as normal, but instead of creating new mergesort tasks to handle both partitions (call this concurrent algorithm A, for use later), start a new mergesort task, to handle one partition, and carry on with the current mergesort task sorting the other partition (call this algorithm B). Don't use other sorting algorithms for small partitions as some versions of mergesort do. (Please acknowledge any sources if you copy a mergesort algorithm.)

To encourage a simpler and more robust program design, mergesort tasks must not be created by calls to **new**, i.e., use no dynamic allocation of tasks in this program.

The executable program is named mergesort and has the following shell interface:

mergesort *unsorted-file [sorted-file]*

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.)

- If the unsorted input file is not specified, print an appropriate usage message and terminate. The input file contains lists of unsorted values. Each list starts with the number of values in that list. For example, the input file:

  ```
  8 25 6 8 –5 99 100 101 7
  3 1 –3 5
  0
  10 9 8 7 6 5 4 3 2 1 0
  ```

  contains 4 lists with 8, 3, 0 and 10 values in each list. (The line breaks are for readability only; values can be separated by any white-space character and appear across any number of lines.) Read a list of values into an array (only one list at a time should be read in). You must handle an arbitrary number of values in a list. (HINT: GNU C++ allows arrays to be dynamically dimensioned.)

  Assume the first number in the input file is always present and correctly specifies the number of following values; assume all following values are correctly formed so no error checking is required on the input data.

- If no output file name is specified, use standard output. Print the original input list followed by the sorted list, as in:

  ```
  25 6 8 –5 99 100 101 7
  –5 6 7 8 25 99 100 101

  1 –3 5
  –3 1 5

  blank line from list of length 0 (not actually printed)
  blank line from list of length 0 (not actually printed)

  9 8 7 6 5 4 3 2 1 0
  0 1 2 3 4 5 6 7 8 9
  ```

  for the previous input file.

Assume all command line information is error-free, i.e, specified input files exist and do not cause I/O errors, and specified output files do not cause I/O errors.

(b) State the ratio of the number of threads used in algorithm A to the number of threads used in algorithm B. Explain your result.

(c) State the big-O runtime order of a non-parallel mergesort algorithm. Explain the theoretical big-O runtime order of your concurrent mergesort program if tasks were running in parallel.

Chapter 6

Atomicity

In the previous chapter, the importance of thread synchronization was demonstrated, e.g., before performing certain operations like communication. Notice that synchronization is an action that occurs *among* threads. However, there are situations where a thread may access a non-concurrent object, e.g., writing to a file or adding a node to a linked-list. In both cases, synchronization is unnecessary with the object because the object is always ready to have an operation applied to it since it has no thread.

However, a complex problem arises if there are multiple threads attempting to operate on the *same* non-concurrent object simultaneously. Imagine the chaos in a file or linked-list if two threads simultaneously write into a file or add a node to a linked-list. In the case of a file, one thread might write over characters that are being written by the other thread. In the case of a linked-list, one thread might change the link fields at the same time as the other thread, and so, only one of the two nodes is added to the list or the list is left in an inconsistent state.

This problem is solved if the operation on the object is **atomic**, meaning indivisible.[1] In other words, no other process can read or modify any partial results during the operation on the object, but the operation can be interrupted for one or more periods of time during its execution. Therefore, each atomic operation executes sequentially (or serially) from the perspective of all threads. The operation that is performed atomically is called a **critical section**, and preventing simultaneous execution of a critical section by multiple threads is called **mutual exclusion**. Therefore, there are two aspects to the problem: knowing which operations must be critical sections and then providing appropriate mutual exclusion for these critical sections.

6.1 Critical Section

A critical section is always a pairing: data and code manipulating the data, where the data are usually objects in this book. It is a common mistake to consider only the block of code as the critical section. Collectively, this pairing denotes an instance of a critical section. Many instances may exist for a particular block of code with different objects, but an object can only be associated with one critical section at a time; hence there is a many-to-one, i.e., objects to code, relationship. For example, two threads may be simultaneously in the file `write` routine but writing to different files. In this case, the fact that both threads are executing the same code, possibly at the same time, does not imply a critical section. Only when both threads are in the `write` routine for the *same* file is a critical section generated.

It is non-trivial to examine a program and detect all the critical sections, particularly because of the pairing necessary to form a critical section. Essentially, a programmer must determine if and when objects are shared by multiple threads and prevent it somehow if it results in problems. One technique is to detect any sharing and serialize all access. Unfortunately, this approach significantly inhibits concurrency if threads are only reading. It is always the case that multiple threads can simultaneously read data without interfering with one another; only writing causes problems. Therefore, it is reasonable to allow multiple simultaneous readers but writers must be serialized. However, care must be taken here. For example, if multiple threads are performing a linear search, they can all be searching the list simultaneously as that only involves reading the shared list. However, if a thread inserts data into the list when the search fails to find a key, there is a problem because the list is changed. Clearly, if two threads try to simultaneously update the list, there is a problem. In fact, there can also be a problem if one thread is updating and another is only reading.

[1] From the Greek $\alpha\tau o\mu o\sigma$ meaning atom (atomic does not mean radioactive).

The reading thread might search onto the new node before data is inserted or the node is completely linked; either case might cause the reader to fail. One solution is to serialize searching of the list, but that inhibits concurrency because most of the time a key is found and no updating occurs. Therefore, it is wasteful from a concurrency perspective to serialize an operation when a write may only occur occasionally. In general, a programmer needs to minimize the amount of mutual exclusion, i.e., make critical sections as short as possible, to maximize concurrency.

However, before rushing ahead on the topic of maximizing concurrency, the fundamental question of how to provide mutual exclusion must be answered. That is, once an operation or part of an operation has been identified as a critical section how is mutual exclusion enforced?

6.2 Mutual Exclusion

The goal of mutual exclusion is to ensure only one thread is executing a block of code with respect to a particular object, i.e., a critical section. This capability is accomplished by mutual exclusion code placed before and possibly after the critical section. In general, any number of threads may want to simultaneously enter a critical section.

Mutual exclusion solutions can get complex, so it is useful to have an analogy to explain the problems. The analogy used here is a bathroom.[2] Everyone can relate to the fact that a bathroom (the code) is a shared resource that is normally a critical section with respect to each person (object) using it. Therefore, a bathroom requires mutual exclusion to enforce this, and this mutual exclusion is usually put in place at the bathroom door.

6.2.1 Mutual Exclusion Game

By casting the mutual exclusion problem in the form of a game, it makes the discussion more interesting. To preclude trivial solutions to the mutual exclusion game, such as the computer executing each thread to completion before executing another thread, some additional rules are required:

1. Only one thread can be in its critical section at a time with respect to a particular object.

   The primary requirement and the goal of the game.

2. Tasks run at arbitrary speed and in arbitrary order.

   This rule precludes solutions that require one person to start before the other, or that a person can only take a certain amount of time in the bathroom, or a person must be away from the bathroom and not try to enter for a certain amount of time. Therefore, a person can arrive at the bathroom at any time and spend an arbitrary amount of time in the bathroom.

3. If a thread is not in its critical section or the entry or exit code that controls access to the critical section, it may not prevent other threads from entering the critical section.

   This rule precludes someone from locking the bathroom and taking the key away to guarantee access on their return. Therefore, if someone is not in the bathroom, someone else may always enter the empty bathroom.

4. In selecting a thread for entry to a critical section, a selection cannot be postponed indefinitely. *Not* satisfying this rule is called **indefinite postponement** or **live-lock**.

   This rule covers the situation where two people arrive at the bathroom simultaneously and stand at the door saying “You go first”, “No, you go first”, “No, you go first!!!”, “NO, YOU GO FIRST!!!”, ... This happens all the time in real life and there is always some way to solve it (this is also true for concurrency). People are never found unconscious outside doorways, collapsed from exhaustion, from yelling “No, you go first”. Tasks on the other hand, will argue back and forth forever! Notice in this case, no one is in the bathroom; the argument is going on *outside* the bathroom. Also, because the people (threads) are arguing, they are taking up resources (CPU time) so they are active or “alive”, but indirectly locking each other out of the bathroom, hence the name live-lock.

5. There must exist a bound on the number of threads that are allowed to enter the critical section after a thread has made a request to enter it. *Not* satisfying this rule is called **starvation** or **unfairness**.

[2] The euphemism “bathroom” is not universal; other names include restroom, WC (water closet), washroom, lavatory and just toilet. Choose a euphemism appropriate to your modesty level and cultural background.

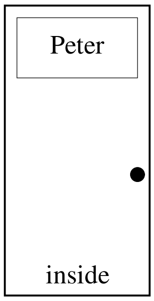
When a person arrives at the bathroom, they must be given a guarantee about how many people go in and out of the bathroom ahead of them. Notice this is not a time guarantee, such as, you can go into the bathroom in 5 minutes. It says that you can get into the bathroom after this or that person has finished. There is no control on how long people spend in the bathroom, but when they come out, there has to be a guarantee that everyone else can eventually get in. This rule does not have to be completely fair, either, as long as there is a bound. For example, the person coming out of the bathroom might be allowed to go right back in again, if they forgot their toothbrush; whereupon, they might decide to take a shower. But when they come out the second time or third time or whatever the bound number is, someone else waiting is allowed to enter the bathroom. The bound number can be a million, but there has to be a bound.

Some solutions to the mutual exclusion game ignore this rule if it can be shown that the chance for any long-term unfairness is extremely low. That is, the chance that a solution might favour one thread over the other for a long period of time is so small that in practice a thread always gets into the critical section after waiting only a short period of time (where short may be from a human perspective not a computer's perspective). Notice that short periods of unfairness are always tolerated depending on the bound, e.g., one thread might enter the bathroom 20 times in a row before another waiting thread is allowed in. The problem with these kinds of probabilistic solutions is the small chance that one thread may never get into the critical section. In important situations like the space shuttle or a nuclear reactor, a probabilistic solution is normally deemed unacceptable because it may result in catastrophic failure.

6.2.2 Self-Checking Critical Section

It turns out that it is very difficult to check that an algorithm satisfies all 5 rules to the mutual exclusion game. In most cases, failure to meet a rule is demonstrated by a counter example, which is usually difficult to construct. However, it is possible to constructively check for rule 1, which is the most important rule of the game, by constructing a self-testing bathroom.

Upon entering the bathroom, a person writes their name on a chalk board on the *inside* of the bathroom door, as in:



A person then starts performing their bathroom chores, but every so often, they look at the chalk board to see if their name is still on the door. If rule 1 is ever broken, someone else will have entered the bathroom and changed the name on the back of the door. Notice that the chalk board is essential because just looking around the bathroom is insufficient; someone could be in the shower and not see someone else enter the bathroom. If a person notices the name has changed, does that mean there is someone currently in the bathroom with them?

The following routine is an implementation of a self-checking bathroom:

```
uBaseTask *CurrTid;                          // shared, current task id in critical section

void CriticalSection() {
    ::CurrTid = &uThisTask();                // address of current task

    for ( int i = 1; i <= 100; i += 1 ) {    // delay
        // perform critical section operation
        if ( ::CurrTid != &uThisTask() ) {   // check for mutual exclusion violation
            uAbort( "interference" );
        }
    }
}
```

The shared, global variable CurrTid is the chalk board and the assignment at the beginning of CriticalSection initializes it to the address of the current task executing in the routine. A task then goes into a loop performing the critical section operation, but at the end of each iteration, the task compares its address with the one in CurrTid for any change. If there is a change, rule 1 has been violated and the program is stopped. Is it necessary to check for interference every time through the loop? For checking in the loop, when would it be a good idea to check? What is the minimum amount of checking necessary to detect a violation of the critical section, i.e., break rule 1?

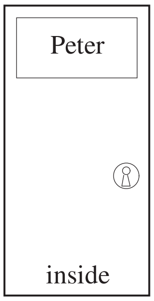
Interestingly, there is no perfect mechanism to ensure absolute segregation of the bathroom by the threads. For example, the self-checking bathroom does a check at the end of the loop but not at the end of the routine. Therefore, there is the potential for a slight overlap between someone entering and someone leaving the bathroom. However, for this critical section, no operations occur after the last loop iteration. Checking at the end of the routine reduces the size of the overlap, but the overlap remains because the thread in the critical section could be interrupted *after* the last check but *before* returning from the routine. In essence, there is always the potential for a moment in time when the back foot of someone leaving the bathroom and the front foot of someone entering are both in the bathroom. However, this does not affect the critical section because its work has been completed by the time the apparent violation occurs.

6.3 Software Solution

Is it possible to write (in your favourite programming language) some code that guarantees that a statement (or group of statements) is always serially executed with respect to a particular object? There is no help from the hardware, only the software is used to solve the problem. Basically, this code must create atomicity out of thin air; that is, in a language and on a computer that provides no atomicity, can atomicity be built from basic language and machine capabilities? The answer is yes, but demonstrating it is non-trivial. Initially, solutions for only *two* threads are examined; N-thread solutions are examined later.

6.3.1 Lock

The first solution is the obvious one—put a lock on the door of the bathroom, as in:



When someone enters the bathroom, they lock the door. While this may work for people, it does not work for computers; in fact, it does not work for people if they behave like computers. Here is a scenario where it fails. Two people walk up to the door, open the door together, enter together, and lock the door together. This scenario does not happen normally because people use their sense of sight, hearing and touch to notice there is someone with them. However, computers have none of these senses, and therefore, they cannot use solutions that rely on them.

It is still worth looking at the program that implements a lock (see Figure 6.1) to see, in detail, why it fails. The shared variable Yale is either open or closed and is initialized to open; this corresponds to the lock on the bathroom door. Routine uMain::main creates two tasks passing a lock reference to each task. Each task's constructor stores the lock reference in a class variable so it can be used by the main member, which goes in and out of the self-checking critical section (bathroom) 1000 times. Having the tasks perform 1000 consecutive entries is contrived; in general, a task goes off to perform some other operation after leaving the critical section, like getting dressed or going to work. Thus, this program should generate many more simultaneous arrives than would occur normally in a concurrent program; nevertheless, it must still handle them.

There is code before the call to routine CriticalSection, called the **entry protocol**, and after, called the **exit protocol**, which is trying to ensure mutual exclusion and the other four rules of the mutual exclusion game. The entry protocol must check if the critical section is being used and wait if it is. If it is being used, there is a busy loop that spins around constantly checking the status of the lock. During busy waiting, a thread is using CPU resources but is not

```
enum Yale { CLOSED, OPEN };

_Task PermissionLock {
    Yale &Lock;

    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            while ( Lock == CLOSED ) {}     // entry protocol
            Lock = CLOSED;
            CriticalSection();              // critical section
            Lock = OPEN;                    // exit protocol
        }
    }
  public:
    PermissionLock( Yale &Lock ) Lock( Lock ) {}
};
void uMain::main() {
    Yale Lock = OPEN;                       // shared
    PermissionLock t0( Lock ) , t1( Lock );
}
```

Figure 6.1: Lock

accomplishing any useful work. As mentioned in Section 5.12, p. 131, busy waiting should be avoided. When the lock is closed, it continues looping because the other thread must be in the critical section (entered the bathroom and locked the door); otherwise, it stops looping, closes the lock, and enters the critical section, which happens when no one is in the bathroom at all, or someone in the bathroom unlocks the door and comes out.

Here is the scenario where rule 1 is broken on a multiprocessor. Two tasks both execute the test in the entry protocol at the same time. They both see the lock is open, they both close the lock, and they both enter the critical section. On a uniprocessor, it is slightly more complex. The CPU must switch between tasks immediately after one task sees the lock is open and exits the busy wait, but before the lock is closed, i.e., between the two entry protocol statements. The other task now attempts to enter the critical section, sees the lock is open, closes the lock and enters the critical section. Eventually, the other task restarts and it does not recheck the lock because it knows it just did that. So it locks the lock (again) and enters the critical section with the other task. You might think that checking the lock for closed and starting the entry protocol again would fix the problem; however, a task does not know when it is interrupted, so it cannot know when to check again.

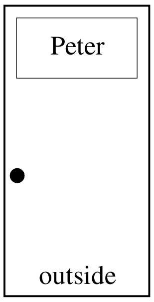
One final point is that even if this could be made to work (and it will be in Section 6.4, p. 160), it still breaks rule 5. That is, there is no bound on how long a thread has to wait before it can enter the critical section. In this specific example, there happens to be a bound of 1000 because each task only enters the critical section that many times; but, in general, no such bound exists if the loops are infinite. In theory, one task could always see the lock closed as the other task whizzes in and out of the critical section; in practice, it is highly unlikely that this would occur for very long, so this is a probabilistic solution with respect to rule 5.

6.3.2 Alternation

Since the obvious approach does not work, it is necessary to look at alternative approaches. Here is a simple approach that does not work for all rules, but does solve rule 1; the idea is to take turns entering the bathroom. This approach is implemented by putting a chalk board on the *outside* of the bathroom door on which you write your name as you leave the bathroom, as in:

```
_Task Alternation {
    int me, &Last;

    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            while ( Last == me ) {}         // entry protocol
            CriticalSection();              // critical section
            Last = me;                      // exit protocol
        }
    }
  public:
    Alternation( int me, int &Last ) : me( me ), Last( Last ) {}
};
void uMain::main() {
    int Last = rand() % 2;                  // shared, random starter
    Alternation t0( 0, Last ), t1( 1, Last );
}
```

Figure 6.2: Alternation



This chalk board is different from the one on the *inside* of the bathroom, on which you write your name when entering the bathroom for use in checking violation of rule 1. Someone cannot enter the bathroom if their name is on the outside of the door, because it is not their turn; they must wait for the other person to finish their turn in the critical section, leave the bathroom, and change the name. The algorithm begins with either person's name on the chalk board.

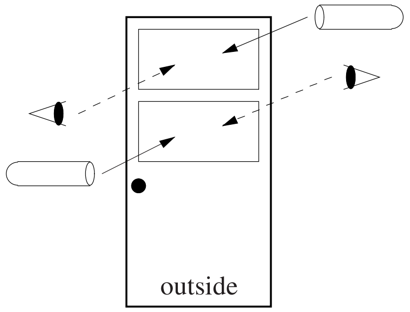
The program to implement this is in Figure 6.2. The shared variable Last, declared in uMain::main indicates which task was last in the bathroom; this corresponds to the chalk board on the outside of the bathroom door. uMain::main initializes Last to the value 0 or 1. These values are arbitrary names for the two tasks; any distinct pair of values works, e.g., task identifiers. uMain::main then creates two tasks passing the names and a reference to the blackboard on the outside of the bathroom door to each task. Each task's constructor stores its name and reference to Last in class variables so they can be used by the task main, which goes in and out of the self-checking critical section (bathroom) 1000 times. Note, one task's me variable has the value 0 and the other has the value 1 but both tasks have a reference to the shared variable Last. The entry protocol is a busy loop waiting until the value of Last is different from the task's name, which means that it was not the last task in the critical section. The exit protocol changes Last to the name of the task last in the critical section.

Now this approach actually achieves rule 1 and it does not rely on order or speed of execution (rule 2), but it does violate rule 3: if a thread is not in the critical section or the entry or exit protocol, it may not prevent other threads from entering the critical section. If one person finishes in the bathroom and goes to work, the other person can enter the bathroom at most once before the other person comes home! When the person at home leaves the bathroom, they become the last person to use the bathroom and have to wait until the person at work comes home and uses the bathroom. While waiting, the bathroom is empty, yet the algorithm is preventing them from using it. Another example of this kind of failure is if task 0 only enters the bathroom 500 times while task 1 wants to enter 1000 times. As soon as task 0 finishes, there is no partner for the other task to alternate with, and so it cannot get into the empty bathroom.

While this approach failed, it illustrates an important lesson: a thread must be able to go into the critical section multiple times in succession if the other thread does not want to go in. While direct use of alternation does not work, later algorithms come back to the idea of alternation for secondary purposes.

6.3.3 Declare Intent

The alternation solution did solve rules 1 and 2, so maybe it can be modified to handle rule 3. To solve rule 3, it is necessary to know if a person is *interested* in entering the bathroom. If one person is not interested, the other person can enter multiple times in succession satisfying rule 3. This scheme can be handled using two chalk boards on the outside of the bathroom door; each is used to indicate intent to enter and erased when leaving the bathroom. Each person is assigned one of the chalk boards (top or bottom). A person can only write on their chalk board and look at the other persons, as in:



(This approach is expressed in the picture by a piece of chalk to write with, and an eye to look at the other chalk board.) If someone wants to enter the bathroom, they first put a check mark on their chalk board indicating intent to enter and then they look at the other chalk board to see if the other person has indicated intent, i.e., put a check mark on their chalk board. If the other person has indicated intent, it implies one of two cases: the other person is in the bathroom or the other person arrived ahead of you and is just entering the bathroom (you cannot see them standing beside you). In either case, they will eventually exit the bathroom and retract their intent. Since you are watching their chalk board, you will see them exit the bathroom when they erase their chalk board.

The program to implement this is in Figure 6.3. The shared variables me and you, declared in uMain::main, correspond to the two chalk boards on the outside of the bathroom door; they are both initialized to not intending to enter the critical section. uMain::main then creates two tasks passing references to the two shared variables to each task; however, the variables are reversed for the second task. This means that task t0 can see both variables but calls one me and the other you, while task t1 sees the same two variables but with the names reversed. Each task's constructor stores these values in class variables so they can be used by the task main, which goes in and out of the self-checking critical section (bathroom) 1000 times. The entry protocol marks the intent to enter, i.e., that task's me, and starts a busy loop waiting until the other task retracts intent, which means it has exited the critical section. The exit protocol retracts intent by changing that task's me. If one task does not declare intent, the other task is free to enter the critical section multiple times in succession, solving rule 3.

While this approach satisfies rules 1–3, it fails on rule 4: in selecting a thread for entry to a critical section, the selection cannot be postponed indefinitely. The failure scenario occurs when the tasks arrive simultaneously. On a multiprocessor, both tasks indicate intent to enter at the same time, both notice the other task wants in, and both wait for the other task to retract intent. However, neither task can now retract intent because neither is making progress towards the critical section; both are spinning in the empty busy loop, mistakenly thinking the critical section is occupied. On a uniprocessor, the failure scenario is slightly more complex. The CPU must switch between tasks immediately after one task indicates intent, but before the other task's intent is checked, i.e., between the two entry protocol statements. The other task now attempts to enter the critical section, sets its intent, and detects the other task's intent is set. Both tasks now spin in the busy loop, taking turns using the CPU to check the other task's intent, which never changes, resulting in a live-lock.

6.3.4 Retract Intent

The declare intent solution did solve rules 1–3, so maybe it can be extended to handle rule 4. Notice, the problem *only* occurs on simultaneous arrival, when people say "You go first", "No, you go first", "No, you go first!!!", "NO, YOU GO FIRST!!!", ... As mentioned, live-lock problems are always fixable. People solve this problem when one person politely backs down so the other person can make progress. This notion of backing down or retracting intent can be incorporated into the previous approach by simply retracting intent in the entry protocol if the other task has its intent set. The task that backs down then waits until the other task has exited the bathroom and retracted its intent. At that

```
enum Intent { WantIn, DontWantIn };

_Task DeclIntent {
    Intent &me, &you;

    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            me = WantIn;                    // entry protocol
            while ( you == WantIn ) {}      // busy wait
            CriticalSection();              // critical section
            me = DontWantIn;                // exit protocol
        }
    }
  public:
    DeclIntent( Intent &me, Intent &you ) : me(me), you(you) {}
};
void uMain::main() {
    Intent me = DontWantIn, you = DontWantIn;  // shared
    DeclIntent t0(me, you), t1(you, me);
}
```

Figure 6.3: Declare Intent

point, the task that backed down, starts the entire entry protocol again. (This approach is like retesting the lock and trying again as suggested in Section 6.3.1, p. 140.) Notice, retracting is a very pessimistic action, because when one task sees that the other task's intent is set it is probably already in the critical section not in the entry protocol, so in most cases it is unnecessary to back down. But it is impossible to tell if a task is in the entry protocol or the critical section when its intent is set, so the simultaneous arrival case must be assumed.

The program to implement this is in Figure 6.4. The only change from the last program is the entry protocol. The entry protocol marks the intent to enter and then checks the other task's intent. If the other task's intent is set, intent is retracted and a busy loop is started waiting until the other task retracts intent. When the other task retracts its intent, which means it has exited the critical section, the entry protocol is started again. As before, if one task does not declare intent, the other task is free to enter the critical section multiple times in succession.

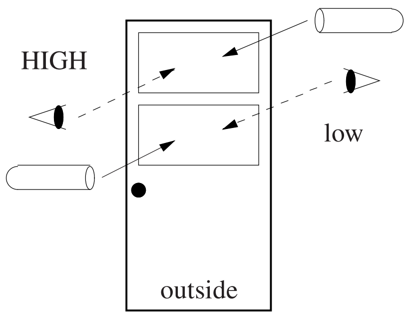
Unfortunately, rule 4 is still broken, however only with a very low probability. Again, the problem occurs if the tasks arrive simultaneously. Here is the scenario where rule 4 is broken on a multiprocessor. The two tasks both indicate intent to enter at the same time. They both notice the other task wants in and so they both retract intent. They both exit the busy loop immediately because neither task has indicated intent and so they both start the protocol over again. If the two tasks continue to execute in perfect synchronization, executing the statements of the protocol at the same time, they defer entry forever. However, the chance of staying in perfect synchronization is very low. The moment one task gets ahead of the other, it enters the critical section because the other task still has its intent retracted. On a uniprocessor, it is even more complex to produce a failure of rule 4. The CPU must switch back and forth after almost every statement in the entry protocol. Furthermore, this has to be keep in perfect synchronization to maintain the live-lock situation. The chances of this occurring in practice is extremely low; nevertheless, it fails in theory.

6.3.5 Prioritize Retract Intent

While the retract intent solution failed to provide a perfect solution to rule 4, it came probabilistically close; unfortunately, "close" only counts in the game of horseshoes. Notice the problem only occurs on simultaneous arrival; if there is some way of breaking the tie, the problem is solved. One possible approach is to assign different priorities to the people as a tie breaking mechanism. For example, at one time, it was considered proper etiquette to give a woman higher priority over a man when simultaneous arrival occurred at a doorway. However, even if tasks had gender, this only solves half the problem, but the approach can be adapted simply by giving one person high priority and the other low priority, as in:

```
enum Intent { WantIn, DontWantIn };

_Task RetractIntent {
    Intent &me, &you;

    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            for ( ;; ) {                        // entry protocol
                me = WantIn;                    // indicate intent
              if ( you == DontWantIn ) break;
                me = DontWantIn;                // retract intent
                while ( you == WantIn ){}       // busy wait
            }
            CriticalSection();                  // critical section
            me = DontWantIn;                    // exit protocol
        }
    }
  public:
    RetractIntent(Intent &me, Intent &you) : me(me), you(you) {}
};
void uMain::main() {
    Intent me = DontWantIn, you = DontWantIn;  // shared
    RetractIntent t0(me, you), t1(you, me);
}
```

Figure 6.4: Retract Intent



When simultaneous arrival occurs, the high priority person always goes ahead of the other person.

The program to implement this is in Figure 6.5. When uMain::main creates the two tasks, it now passes one the value HIGH and the other the value low, in addition to the other values. The entry protocol is now divided into two parts by the **if** statement: code for the high priority task and code for the low priority task. Alternatively, two different tasks could have been created with the high priority code in one task and the low priority code in the other. The code for the high priority task is identical to the declare intent program (Figure 6.3) and the code for the low priority task is identical to the retract intent program (Figure 6.4). Basically, the high priority task never retracts intent; if it detects the other task has indicated its intent, it just spins with its intent set until the other task retracts intent, either in the entry protocol or by exiting the critical section. The low priority task always retracts intent in the entry protocol if it detects that the high priority task wants into or is already in the critical section. The priorities ensure there is never a live-lock.

Having solved rule 4, the next problem is rule 5. Unfortunately, this approach violates rule 5: there must exist a bound on the number of other tasks that are allowed to enter the critical section after a task has made a request to enter it. In theory, the high priority task could always "barge" ahead of the low priority task by racing out of the critical section and back into it again, so the low priority task starves, i.e., never enters the critical section; in practice, this happens very rarely and so this approach is probabilistically correct for rule 5 in many situations.

```
enum Intent { WantIn, DontWantIn };
enum Priority { HIGH, low };

_Task PriorEntry {
    Intent &me, &you;
    Priority priority;

    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            if ( priority == HIGH ) {           // entry protocol
                me = WantIn;                    // high priority
                while ( you == WantIn ) {}      // busy wait
            } else {                            // low priority
                for ( ;; ) {                    // busy wait
                    me = WantIn;
                  if ( you == DontWantIn ) break;
                    me = DontWantIn;
                    while (you == WantIn) {}        // busy wait
                }
            }
            CriticalSection();                  // critical section
            me = DontWantIn;                    // exit protocol
        }
    }
  public:
    PriorEntry( Priority p, Intent &me, Intent &you ) : priority(p), me(me), you(you) {}
};
void uMain::main() {
    Intent me = DontWantIn, you = DontWantIn;  // shared
    PriorEntry t0( HIGH, me, you ), t1( low, you, me );
}
```

Figure 6.5: Prioritize Retract Intent

6.3.6 Fair Retract Intent

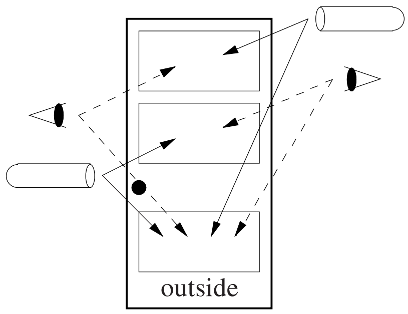
If "close" is not good enough for rule 4, it is also not good enough for rule 5. The problem with the priority scheme is that it is not fair; if it can be made fair, it would solve rule 5. One way to introduce fairness (and there are many different ways) is to alternate entry to the critical section for simultaneous arrivals. While alternation is rejected in Section 6.3.2, p. 141, it is now going to be used *but* only for the rare case of simultaneous arrival, when it is known that both threads want to enter the critical section. It is not used in the case when one thread does not want to go into the critical section, and so prevents the other from entering multiple times in succession. Two approaches to alternation on simultaneous arrival are examined.

6.3.6.1 Dekker

This scheme is implemented by adding one more chalk board on the outside of the bathroom door, which is used to indicate who was last in the bathroom, as in:

```
enum Intent { WantIn, DontWantIn };

_Task Dekker {
    Intent &me, &you, *&Last;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            for ( ;; ) {                          // entry protocol, high priority busy wait
                me = WantIn;
              if ( you == DontWantIn ) break;
                if ( Last == &me ) {
                    me = DontWantIn;
                    while ( Last == &me ){}     // low priority busy wait
                }
            }
            CriticalSection();                    // critical section
            Last = &me;                           // exit protocol
            me = DontWantIn;
        }
    }
  public:
    Dekker( Intent &me, Intent &you, Intent *&Last ) : me(me), you(you), Last(Last) {}
};
void uMain::main() {
    Intent me = DontWantIn, you = DontWantIn,  // shared
            *Last = rand() % 2 ? &me : &you;
    Dekker t0(me, you, Last), t1(you, me, Last);
}
```

Figure 6.6: Fair Retract Intent: Dekker



Whenever there is a simultaneous arrival, the person who was last in the bathroom has to wait, which seems fair. However, if there is no simultaneous arrival, i.e., the other task does not want in, the new chalk board is not even used, so a single task can make multiple entries without the other task. In fact, the first person to win the mutual exclusion game was the Dutch mathematician Theo J. Dekker, and so this approach is named after him. (The algorithm and credit for its solution are discussed at length by Edsger W. Dijkstra in [Dij65].)

The program to implement this is in Figure 6.6. (This version is a structured version of the original presented by Dijkstra. An unstructured version, similar to the original, appears in question 9 at the end of Chapter 2, p. 7.) The program is a combination of the alternation program (Figure 6.2, p. 142) and the retract intent program (Figure 6.4, p. 145). The shared variable Last indicates which task was last in the bathroom by alternately pointing at the two intent variables in uMain::main; this corresponds to the new chalk board on the outside of the bathroom door. These values are arbitrary names for the two tasks; any distinct pair of values works. uMain::main initializes Last to point to one of the intent variables. Now, the entry protocol is slightly complex. Like priority entry, there are two parts: code for the high and low priority tasks. However, which task is high or low alternates depending on which task was last in the critical section. The outer loop is where the high priority task busy waits and the inner loop is where the low priority task busy waits. The **if** statement (not loop exit) controls which task has low or high priority. The low priority loop is nested in the high priority loop because of the two steps in the exit protocol: setting Last and retracting intent. The low priority task in the nested loop of the entry protocol might detect Last having changed, exit the busy loop, but still have to wait for the other task to complete the exit protocol by retracting its intent. Think of what could go wrong if the low priority task did not wait for the other task to complete *all* the exit protocol before preceding into the critical section. In the case of simultaneous arrival, the task last in the critical section is directed to the low priority code, where it retracts intent so the other task can make progress. Hence, fairness by alternation occurs and rule 5 is satisfied.

What is the bound for Dekker's algorithm? It is one, *once the low priority task in the entry protocol sets its intent*. When the high priority task exits the critical section, it could take the low priority task an arbitrary amount of time to restart, exit the low priority busy wait, and set its intent. However, the bound begins only *after* a task states its desire to enter the critical section, whether it has just arrived or is in the entry protocol. Until then, the other task is not prevented from entering the critical section (rule 3). However, once the low priority task sets its intent and has become the high priority task because Last is no longer equal to this task, the other task cannot enter again unless it is just entering or is already in the critical section. Only if the new high priority task does not get any CPU time could it starve, but it is assumed that all tasks eventually execute with some degree of fairness (e.g., the ready queue is FIFO).

6.3.6.2 Peterson

G. L. Peterson [Pet81] designed another winning algorithm for the mutual exclusion game, but it makes an assumption about the hardware. Peterson's algorithm relies on the fact that almost all parallel computers guarantee memory write is executed atomically. That is, the hardware treats assignment like a critical section and provides mutual exclusion around it. If the hardware does not do this, simultaneous assignment to the same memory location can result in the bits being scrambled, generating an arbitrary value. In effect, this assumption of atomicity is cheating, because the purpose of the mutual exclusion game is to generate mutual exclusion, and Peterson's algorithm is building its mutual exclusion out of some that already exists. Still, it is always easier to construct general mutual exclusion out of any existing mutual exclusion.

The algorithm uses the third chalk board, but in a slightly different way, to deal with simultaneous arrivals and fairness. After a person indicates intent on their own chalk board, they *race* with the other person to put their name on the third chalk board. The person who wins the race breaks the tie and goes into the critical section, which handles live-lock problems (rule 4). Fairness (rule 5) is provided by ensuring that a person cannot win two races in a row. As with Dekker's algorithm, the third chalk board is used only for simultaneous arrivals, otherwise a person is free to enter multiple times in succession.

The program to implement this is in Figure 6.7. The program looks identical to the declare intent program (Figure 6.3, p. 144), except for the entry protocol. If both tasks arrive simultaneously, they both indicate their intent to enter and they both race to put the address of their me variable in Last. Since the assignment occurs atomically, one of the tasks has to wait while the other performs the assignment. After the race is over, how do you know which task won? That is, how does a task know which task put its me variable address in first? In fact, the loser's value is in variable Last, because it wrote over the winner's value! Now each task enters the busy loop, checking first if the other task even wants in, and if it does, who won the race. Notice this last check, if the value in Last is that task's me variable, it lost the race and has to wait. Therefore, live-lock cannot occur.

Starvation cannot occur for the following reason. If a task exits the critical section, and tries to "barge" ahead of a waiting task, it must first indicate its intent and run a race with itself in the entry protocol (the other task is still in the busy loop). Interestingly, if you run a race by yourself and you are an optimist, you win the race; if you are a pessimist, you lose the race. In this case, the task that runs the race by itself loses the race and waits in the busy loop. (Why does it lose the race?) Now the other task, which is waiting in the busy loop, suddenly notices that it has won the previous race because Last has changed, i.e., this task has now become the high priority task. Therefore, it makes progress and enters the critical section. Thus, a task cannot win two races in a row, and so the tasks alternate on a particular series of simultaneous arrivals. Since winning a race is random, whoever wins the first race starts the alternation for any series of simultaneous arrivals. Similarly for Dekker's algorithm, whoever was last in the bathroom is random for a simultaneous arrival, but that task starts the alternation for any series of simultaneous arrivals. However, Dekker's algorithm is slightly better if the new high priority task is a slow busy waiter or delayed in the ready queue. In Peterson's algorithm, the race loser does not retract intent, so if it subsequently becomes the high priority task and is slow to exit the busy loop, the other task cannot use the critical section. On the surface, this situation seems to violate rule 3, i.e., preventing access to the critical section, but the prevention is occurring *in the entry protocol*, which is

```
enum Intent { WantIn, DontWantIn };

_Task Peterson {
    Intent &me, &you, *&Last;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            me = WantIn;                    // entry protocol
            Last = &me;                     // race!
            while ( you == WantIn && Last == &me ) {}
            CriticalSection();              // critical section
            me = DontWantIn;                // exit protocol
        }
    }
  public:
    Peterson( Intent &me, Intent &you, Intent *&Last ) : me(me), you(you), Last(Last) {}
};
void uMain::main() {
    Intent me = DontWantIn, you = DontWantIn,   // shared
            Intent *Last = rand() % 2 ? &me : &you;
    Peterson t0(me, you, Last), t1(you, me, Last);
}
```

Figure 6.7: Fair Retract Intent: Peterson

allowed by rule 3.

Once again, it is important to understand that Peterson's algorithm assumes assignment is an atomic operation, while Dekker's algorithm makes no assumptions about existing atomicity.[3] In fact, Dekker's algorithm works on a machine where the bits are scrambled during simultaneous assignment. This fact is easy to see because the only assignment to the shared variable, Last, in Dekker's algorithm occurs in the exit protocol and the other task cannot make progress out of the entry protocol until both assignments in the exit protocol are finished. Therefore, there are never simultaneous assignments to a shared variable. Furthermore, even if the waiting task sees the bits changing in the shared variables, but not stabilized, the check for equality on the outer loop ensures progress cannot occur until the entire exit protocol has been executed. (The **stabilization assumption** is that the left-hand side of an assignment cannot take on the value of the right-hand side more than once during the assignment. For example, the assumption is true if the bits are assigned by zeroing the assigned variable and shifting the bits in from the expression value one at a time, from either direction.)

6.3.7 N-Thread Mutual Exclusion

It is now time to increase the complexity of the the mutual exclusion game by increasing the number of players from 2 to N. To be honest, the purpose of this section is to show that even a simple N-thread solution is too complex for most people to grasp from a correctness standpoint.

> Another possible myth is that Dekker's solution can be trivially modified to solve the n process case. The algorithms known to the author *(G. L. Peterson)* actually require major changes in form that result in entirely new algorithms, even when n is two. [Pet81, p. 116]

Therefore, only a few N-thread algorithms are examined to illustrate just how complex the problem becomes.

6.3.7.1 Prioritize Retract Intent

The first N-thread algorithm is based on the Prioritize Retract Intent program (Figure 6.5, p. 146). Because the algorithm is based on fixed priorities, it does not satisfy rule 5 (starvation), as a high-priority task can theoretically

[3] This fact is normally ignored in the literature, with the incorrect assumption that Peterson's algorithm is the same as Dekker's but simpler.

> Simpler solutions *(than Dekker's)* to the process mutual-exclusion problem have since been presented by Doran and Thomas [DT80] and Peterson [Pet81]. [SPG91, p. 192]

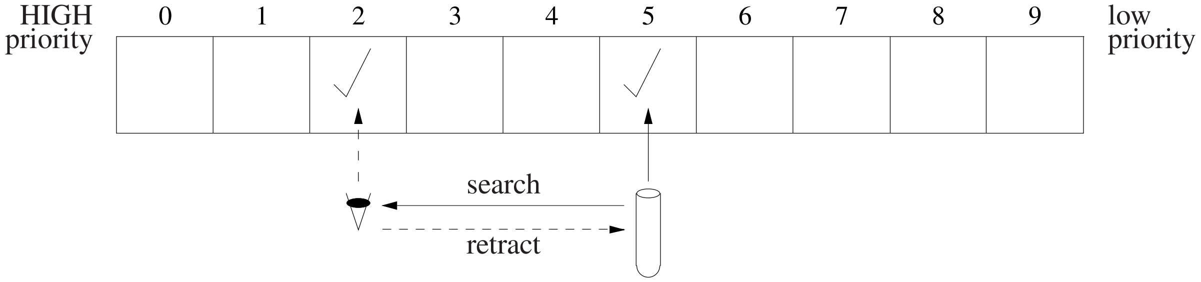
The algorithm by Doran and Thomas rewrites Dekker's algorithm so it is marginally simpler to read, but still the same algorithm. Peterson's algorithm is different and has less code than Dekker's, but it is neither as robust nor does it create atomicity out of thin air.

```
enum Intent { WantIn, DontWantIn };

_Task Worker {
    Intent *intents;
    int N, priority, i, j;

    void main() {
        for ( i = 1; i <= 1000; i += 1 ) {
            // step 1, wait for tasks with higher priority
            do {                                        // entry protocol
                intents[priority] = WantIn;
                // check if task with higher priority wants in
                for ( j = priority - 1; j >= 0; j -= 1 ) {
                  if ( intents[j] == WantIn ) {
                        intents[priority] = DontWantIn;
                        while ( intents[j] == WantIn ) {}
                        break;
                    } // exit
                }
            } while ( intents[priority] == DontWantIn );
            // step 2, wait for tasks with lower priority
            for ( j = priority + 1; j < N; j += 1 ) {
                while ( intents[j] == WantIn ) {}
            }

            CriticalSection();                          // critical section

            intents[priority] = DontWantIn;             // exit protocol
        }
    }
  public:
    Worker( Intent intents[ ], int N, int priority ) :
        intents( intents ), N( N ), priority( priority ) {}
};
void uMain::main() {
    const int NoOfTasks = 10;
    Intent intents[NoOfTasks];                          // shared
    Worker *workers[NoOfTasks];
    int i;

    for ( i = 0; i < NoOfTasks; i += 1 ) {              // initialize shared data
        intents[i] = DontWantIn;
    }
    for ( i = 0; i < NoOfTasks; i += 1 ) {
        workers[i] = new Worker( intents, NoOfTasks, i ); // create workers
    }
    for ( i = 0; i < NoOfTasks; i += 1 ) {              // terminate workers
        delete workers[i];
    }
}
```

Figure 6.8: N-Thread Mutual Exclusion: Prioritize Retract Intent

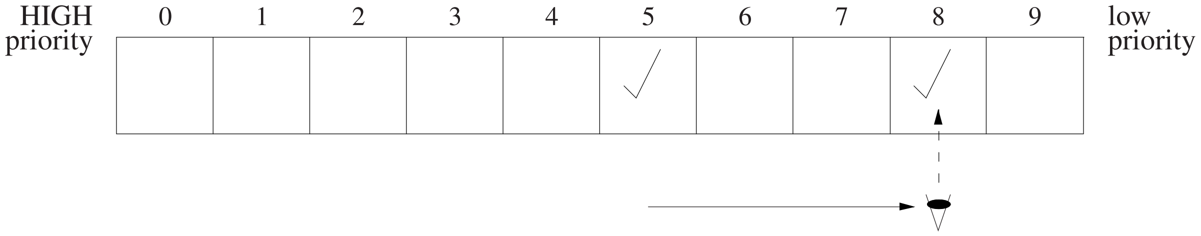
preclude a low-priority task from making progress. Nevertheless, this simple approach illustrates solving at least rules 1-4 (see program in Figure 6.8). The program uses an array of bytes (but only one bit is used), one per task, to represent the N blackboards needed on the bathroom door. This array is passed to each task along with the size of the array and a position/priority value. The position/priority indicates the position in the array of a particular task's bit and its priority level. For example, a task passed the value 4 for the position/priority can set and reset element 4 of the array and has higher priority than tasks given higher values and lower priority than tasks given lower values, i.e., low subscript values have higher priority.

The entry protocol has two steps. In the first step, a task begins by setting its intent and then linearly searching in the high priority direction to see if there is a task with higher priority that has declared its intent. If such a task is found, the searching task retracts its intent and busy waits until the higher priority task retracts its intent and begins the search over again from its position/priority location, as in:



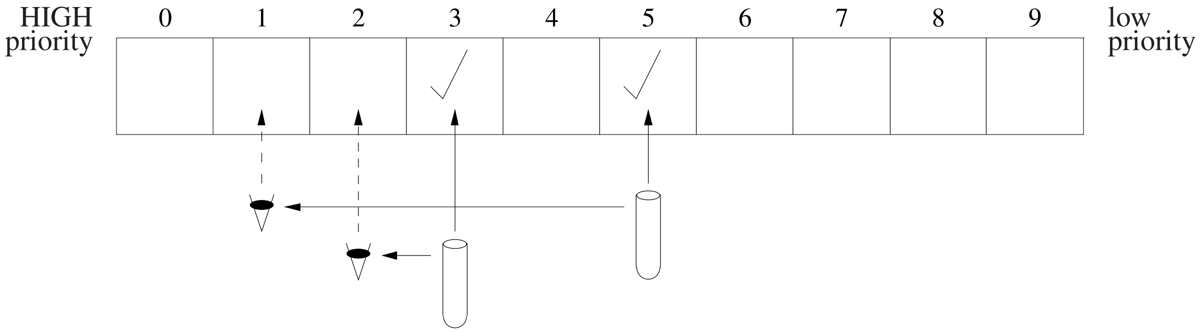
Only after a task has searched all the way up the high priority direction and found no task with higher priority intent, does it move on to the second step. In essence, a task sees a moment in time when there are no higher priority tasks ahead of it.

In the second step of the entry protocol, a task searches in the low priority direction. However, since all tasks in this direction have lower priority, the searching task never retracts its intent; it just busy waits until the low priority task retracts its intent, i.e., leaves the critical section or retracts intent in the entry protocol, and then continues along the list, as in:

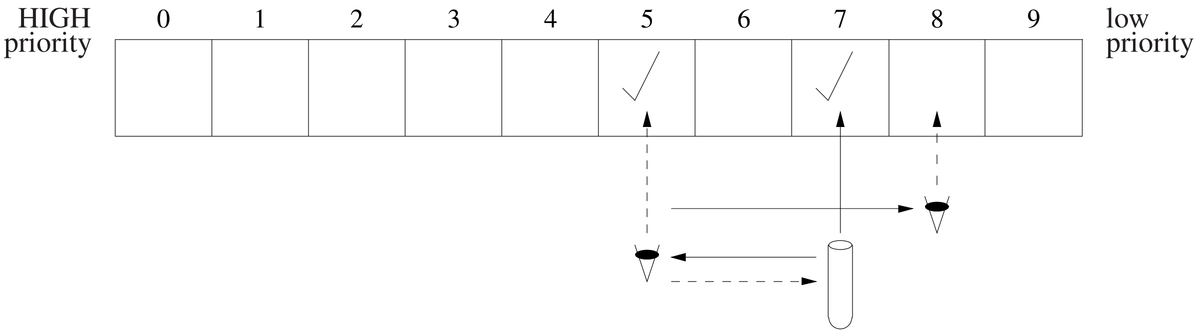


When a task reaches the end of the list, it sees no lower priority task in the critical section and so it can safely enter it.

Since the algorithm is complex, some of the problem cases need to be examined. First, what happens if a task is searching in the high priority direction and a higher priority task starts searching *after* its intent has been passed by the lower priority task, as in:



This situation is not a problem because when the higher priority task starts step 2 of the entry protocol it waits for the lower priority task to eventually exit the critical section. In essence, the lower priority task started searching *before* the higher priority task and now the higher priority task waits for it to complete. In this situation, the time of arrival affects the priority of a task. The second case is where a low priority task declares its intent *after* the higher priority task has started its search, as in:



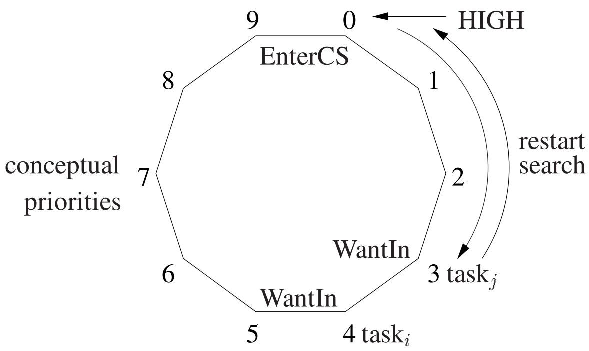
This situation is not a problem because when the lower priority task finds the high priority task's intent, it retracts its intent. Remember, the high priority task does not retract its intent when performing the low priority search. There are many other cases that need to be examined before this algorithm can be shown to satisfy rules 1-4, but a complete proof is beyond the scope of this book. Suffice it to say, that the algorithm is correct for rules 1-4.

Notice this solution uses only one bit per task, indicating the task's intent to enter the critical section. In fact, any N-thread solution using only atomic read and write, has a lower bound of one bit for each task attempting entry and requires the ability to read the bits of all the tasks attempting entry [BL80]. However, no complete N-thread solution has achieved this lower bound. In effect, dealing with rule 5 (starvation) requires additional information, which correspondingly requires additional bits to represent the information.

6.3.7.2 Eisenberg and McGuire

The first (almost) complete N-thread solution to the critical section game was by Eisenberg and McGuire [EM72] (see program in Figure 6.9). This solution is similar to the previous one with a variation on how the priorities work. Instead of having fixed priorities assigned to the array positions, the priorities slide around the array in a circle. The task currently in the critical section or last in the critical section always has the highest priority. The tasks waiting to enter, in the clockwise direction from the highest priority, have decreasing priority levels. When the task in the critical section exits, it rotates the priorities clockwise to the next task attempting entry, making it the highest priority task. If no tasks are attempting entry, the priorities are not rotated. This algorithm uses 3 states (WantIn, EnterCS, DontWantIn) to indicate intention (at least 2 bits/intent) and N intents plus an integer variable.

The entry protocol has two steps. In the first step, a task at position i begins by setting its intent and then linearly searching clockwise from position HIGH, i.e., the position of the current/last user of the critical section, to itself for any task that wants into the critical section. If such a task is found, e.g., task$_j$, that task has higher priority, so the searching task restarts the search *but without retracting its intent*, as in:

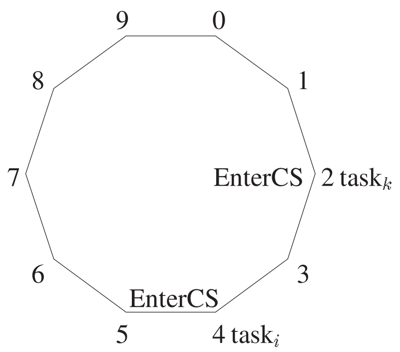


Only after a task has searched from position HIGH to its position and found no task waiting to enter does it move onto the second step. In essence, a task sees a moment in time when there are no higher priority tasks between HIGH and its position.

```
enum Intent { WantIn, EnterCS, DontWantIn };

_Task Worker {
    Intent *intents;
    int &HIGH, N, posn;

    void main() {
        int i, j, k;
        for ( i = 0; i < 1000; i += 1 ) {
            for ( ;; ) {                                    // entry protocol
                intents[posn] = WantIn;
                // step 1, wait for tasks with higher priority
                for ( j = HIGH; j != posn; )
                    if ( intents[j] != DontWantIn ) j = HIGH; // restart search
                    else j = ( j + 1 ) % N;                 // check next intent
                intents[posn] = EnterCS;
                // step 2, check if any other task finished step 1
                for ( j = 0; j < N && ( j == posn || intents[j] != EnterCS ); j += 1 ) {}
              if ( j == N && (HIGH == posn || intents[HIGH] == DontWantIn) ) break;
            }
            HIGH = posn;

            CriticalSection();

            for ( j = (HIGH + 1) % N; intents[j] == DontWantIn ; j = ( j + 1 ) % N ) {} // exit protocol
            HIGH = j;
            intents[posn] = DontWantIn;
        }
    }
  public:
    Worker( int &HIGH, Intent intents[ ], int N, int posn ) :
        HIGH( HIGH ), intents( intents ), N( N ), posn( posn ) {}
};
void uMain::main() {
    const int NoOfTasks = 10;
    Worker *workers[NoOfTasks];                             // pointer to an array of workers
    int HIGH;                                               // shared
    Intent intents[NoOfTasks];                              // shared
    int i;

    HIGH = 0;                                               // initialize shared data
    for ( i = 0; i < NoOfTasks; i += 1 ) {
        intents[i] = DontWantIn;
    }
    for ( i = 0; i < NoOfTasks; i += 1 ) {
        workers[i] = new Worker( HIGH, intents, NoOfTasks, i ); // create workers
    }
    for ( i = 0; i < NoOfTasks; i += 1 ) {                  // terminate workers
        delete workers[i];
    }
}
```

Figure 6.9: N-Thread Mutual Exclusion: Eisenberg and McGuire

In the second step of the entry protocol, a task sets its intent to a new state, EnterCS, to indicate it has completed the first step of the entry protocol. It then linearly searches for any other task (excluding itself) that might have also reached the second step of the entry protocol and have its intent set to EnterCS, as in:



Multiple EnterCS states can occur when a higher priority task arrives after its position has been examined by a lower priority task in step 1. The second step of the entry protocol succeeds if a task does not find another task with an intent of EnterCS, i.e., j == N, and either its position has been designated the highest priority by the task leaving the critical section or the task associated with the current highest priority location does not want to enter the critical section. If the appropriate conditions are true, the task rotates the priorities so that it is now the highest priority task, i.e., HIGH = posn; otherwise, the entry protocol is repeated.

In the exit protocol, the exiting task searches in priority order for any task that wants in other than itself. If a task is found, that task is made the highest priority, in affect rotating the priorities; otherwise, HIGH is reset to its previous value. Then the exiting task retracts its intent. Because HIGH is being read for assignment (versus comparison) in step 1 to restart the search, assignment must be atomic, otherwise writing HIGH could cause incorrect termination of the loop in step 1.

It is easy to see how this algorithm works if there is always a task in the critical section. Tasks busy wait in step 1 of the entry protocol for the task currently in the critical section to exit and rotate the priorities by changing HIGH to the waiting task closest to it in the clock-wise direction. Changing HIGH short-circuits the loop in step 1 of the entry protocol for the task at that position because j and posn have the value HIGH. All other tasks stay in step 1 and the newly assigned high priority task progresses through step 2 and enters the critical section. The difficult case in this algorithm is when no task is in the critical section, and HIGH is sitting at an arbitrary location when the tasks arrive, e.g., like the initial condition when the tasks all start. This situation is the reason for step 2 in the entry protocol.

When there is no task in the critical section to designate the next task to enter, the entering tasks race to become the next task to enter, and the previous value of HIGH is used to control the selection. As above, the task closest to the previous value of HIGH exits step 1 first because it finds no intent set for higher priority tasks; the other tasks wait for it. However, there can be many false starts by the waiting tasks as other tasks arrive with different priorities. For example, if tasks arrive from low to high priority, all the tasks can simultaneously get to step 2 of the algorithm, as the low priority tasks miss seeing the later arriving higher-priority tasks in step 1. This situation results in *all* of the tasks restarting the entry protocol. The arrival of a higher priority task, including the one at the current HIGH position, can force this cycle to start over again at any time. Hence, the selection process, when there is no task in the critical section and there are simultaneous arrivals, can be expensive.

The bound on the waiting time is $N - 1$ because a task may have to wait for all the other tasks to enter and leave the critical section before it can proceed. The bound is ensured because rotating the priorities in a fixed direction in the exit protocol eventually services all waiting tasks. However, arriving tasks are not serviced in FIFO order but in cyclic order. A task arriving after the current HIGH position is serviced only when tasks before the current HIGH position are serviced, even though task's after the current HIGH position may have arrived prior to task's before it. Therefore, this algorithm is an example of bounded but non-FIFO selection.

This algorithm introduces a new and crucial concept in its exit protocol: the introduction of **cooperation** between the acquiring tasks and the releasing task. In all the previous locks, the releasing task *only* modifies variables associated with itself and possibly a shared variable. All the work in detecting if a lock is open and acquiring the lock is the responsibility of the acquiring tasks with no help from the releasing task. In this solution, the releasing task must perform additional work on behalf of an acquiring task. Each acquiring task relies not only on itself to detect when the lock is released, but on the releasing task. This idea of cooperation, where the releaser does more work than just indicating it is finished, forms the basis for advanced lock design.

6.3.7.3 Bakery Algorithm

The second complete N-thread solution to the critical section game was by Lamport [Lam74] (see program in Figure 6.10). This solution introduces a new idea for achieving a bounded waiting time, which appears in different forms in subsequent programs throughout the rest of the book. The new idea is one used by people sharing a resource requiring mutual exclusion, which is to take a ticket on arrival and wait until that ticket is chosen for service. For example, in a bakery, people take a ticket on arrival and then wait for exclusive service. In fact, Lamport's algorithm is often referred to as the bakery algorithm. If the tickets are serviced in increasing order by ticket number, the people are serviced in FIFO order (in contrast to the cyclic order in the previous algorithm). However, as will be seen, Lamport's version of tickets is quite different from that used by people in a store. This algorithm uses M intention states (tickets), needing B bits (usually the size of an integer), and N intents.

The entry protocol has two steps. In the first step, task$_i$ takes a ticket and stores it at position i in the shared array `ticket`. The `ticket` array is initialized to 0, where 0 means a task has retracted its intent. However, selecting a ticket is done in an unusual manner. Instead of selecting a ticket from a machine that atomically generates monotonically increasing values, a task computes a unique ticket based on the current ticket values held by any waiting tasks. Ticket generation is accomplished by finding the maximum ticket value of the waiting tasks and creating a new ticket value one greater, which requires an $O(N)$ search (but could be less, e.g., store the tickets in a binary tree).

Unfortunately, this procedure does not ensure a unique ticket value. Two (or more) tasks can simultaneously compute the same maximum ticket value and add one to it, resulting in multiple tickets with the same value. While the chance of this problem occurring is probabilistically low, it still must be dealt with. To deal with this problem, the same technique is used as for the N-thread prioritize-retract-intent solution (see Section 6.3.7.1, p. 149), which is to assign a priority to each position in the intent array. When two tasks have the same ticket value, their position in the intent array is used to break the tie. As a result, it is possible for task$_i$ to arrive ahead of task$_j$ and begin computing its ticket but be serviced in non-FIFO order because both tasks generate identical tickets due to time-slice interrupts during the ticket computation. This anomaly arises from the non-trivial amount of time required to compute a ticket, which makes it impossible to make the ticket selection atomic.

In the second step of the entry protocol, a task linearly searches through all the waiting tasks (including itself) and waits for any task with higher priority, because it has a lower ticket value or lower subscript for equal tickets. Remember, a non-waiting task is identified by a zero ticket value. However, there is a problem if two (or more) tasks compute the same ticket value but the lower-priority task advances to the second step before the higher-priority task assigns its equal ticket-value into the ticket array, e.g., it is interrupted just before the assignment. When the lower-priority task checks the ticket array, it finds no task with an equal ticket-value, as the other task has not performed its assignment; therefore, it proceeds into the critical section. When the higher-priority task checks the ticket array, it finds a lower-priority task with the same ticket value but higher subscript so it proceeds into the critical section, violating the mutual exclusion. To solve this problem, it is crucial to delay checking a task's ticket value if it is computing a ticket. This information is provided by the `choosing` array. A task sets its position in this array before starting to compute a ticket, and resets it only after assigning its new ticket. The search in the second step is now composed of two checks. First, determine if a task is computing a ticket, and if so, wait for the ticket to be assigned. Second, check if the task wants in, and if so, is it higher priority. Now a lower-priority task knows if a higher-priority task is computing a ticket and waits for the new ticket value to appear. When the ticket value appears, it is either less, equal, or greater than the checking task's ticket, and it proceeds or waits appropriately. Finally, assignment must be atomic for the bakery algorithm because `choosing` and `ticket` values can be read by other tasks in step 2 while being written in step 1.

Hehner and Shyamasundar [HS81] simplified Lamport's algorithm by folding the `choosing` array into the ticket array (see program fragment in Figure 6.11, p. 157). In essence, two ticket values are set aside to indicate that either a task does not want into the critical section or it is computing a ticket. A non-waiting task is identified by the maximum possible ticket value, `INT_MAX`, implying lowest priority, and all elements of the ticket array are initialized to this maximum value. Computing a ticket is identified by the minimum ticket value, 0, implying highest priority.

A task starts computing its ticket by setting its ticket value to zero, which is the highest-priority ticket-value. No task can have a ticket value of zero because the maximum value found is always incremented. Notice, during the computation of a task's ticket-value, the `INT_MAX` value must be ignored. Now, when a task is searching in the second step, it cannot proceed if a task is computing a ticket because of the temporary high-priority value. Only after the new

```
_Task Worker {
    bool *choosing;
    int *ticket, N, priority;

    void main() {
        int i, j;
        for ( i = 0; i < 1000; i += 1 ) {
            // step 1, compute a ticket
            choosing[priority] = true;                 // entry protocol
            int max = 0;                               // O(N) search for largest ticket
            for ( j = 0; j < N; j += 1 )
                if ( max < ticket[j] ) max = ticket[j];
            ticket[priority] = max + 1;                // advance ticket
            choosing[priority] = false;
            // step 2, wait for ticket to be selected
            for ( j = 0; j < N; j += 1 ) {             // check other tickets
                while ( choosing[j] ) {}               // busy wait if task selecting ticket
                while ( ticket[j] != 0 &&              // busy wait if intent set and
                    ( ticket[j] < ticket[priority] ||  //  greater ticket value or subscript
                    ( ticket[j] == ticket[priority] && j < priority ) ) ) {}
            }

            CriticalSection();

            ticket[priority] = 0;                      // exit protocol
        }
    }
  public:
    Worker( bool choosing[ ], int ticket[ ], int N, int priority ) :
        choosing( choosing ), ticket( ticket ), N( N ), priority( priority ) {}
};
void uMain::main() {
    const int NoOfTasks = 10;
    Worker *workers[NoOfTasks];                        // pointer to an array of workers
    bool choosing[NoOfTasks];                          // shared
    int ticket[NoOfTasks];                             // shared
    int i;

    for ( i = 0; i < NoOfTasks; i += 1 ) {             // initialize shared data
        choosing[i] = false;
        ticket[i] = 0;
    }
    for ( i = 0; i < NoOfTasks; i += 1 ) {             // create workers
        workers[i] = new Worker( choosing, ticket, NoOfTasks, i );
    }
    for ( i = 0; i < NoOfTasks; i += 1 ) {             // terminate workers
        delete workers[i];
    }
}
```

Figure 6.10: N-Thread Mutual Exclusion: Lamport (Bakery Algorithm I)

```
// step 1, select a ticket
ticket[priority] = 0;                       // set highest priority
int max = 0;                                // O(N) search for largest ticket
for ( j = 0; j < N; j += 1 )
    if ( max < ticket[j] && ticket[j] < INT_MAX ) max = ticket[j];
ticket[priority] = max + 1;                 // advance ticket
// step 2, wait for ticket to be selected
for ( j = 0; j < N; j += 1 ) {              // check other tickets
    while ( ticket[j] < ticket[priority] ||      // busy wait
        ( ticket[j] == ticket[priority] && j < priority ) ) {}
}

CriticalSection();

ticket[priority] = INT_MAX;                 // exit protocol
```

Figure 6.11: N-Thread Mutual Exclusion: Hehner and Shyamasundar (Bakery Algorithm II)

ticket value is assigned does the searching task make progress. Finally, while busy waiting in the second step, it is unnecessary to explicitly check for non-waiting tasks because they all have a priority lower than the searching task, i.e., INT_MAX.

HIGH priority	0	1	2	3	4	5	6	7	8	9	low priority
	∞	∞	17	∞	∞	18	18	0	20	19	

There is no indefinite postponement (violation of rule 4) for the same reason as in the previous solutions. That is, the priority of the tickets, or alternatively, the priority of a task's position in the intent array ensures that on simultaneous arrival a tie is always broken. There is no starvation (violation of rule 5) because the ticket values are always increasing as long as there are waiting tasks. Hence, if a task exits the critical section and tries to rush in again ahead of other waiting tasks, it *must* generate a ticket value greater than any waiting task because it examines all their ticket values. Because no task retracts its intent in the entry protocol, this new maximal ticket value ensures the arriving task cannot enter before any of the other waiting tasks and establishes its bound with respect to any waiting tasks.

The problem with this and any other approach based on tickets is that the ticket values cannot increase indefinitely due to the finite size of the integer type, say 2^{32} values. In the worst case, a ticket value overflows back to zero (or a negative value), which results in multiple tasks entering the critical section simultaneously, i.e., a violation of the critical section. However, in these bakery algorithms, whenever the critical section becomes unused, i.e., no task is using it or waiting to use it, the maximal ticket value resets because the next task selects ticket 0. The ticket values only increase when tasks arrive and must wait for entry to the critical section. For example, this failure occurs when 2^{32} tasks arrive at the critical section while it is being used, which is highly unlikely. Even two tasks can overflow the tickets by perfectly alternating 2^{32} time so one arrives at the critical section while the other is using it; again, this scenario is equally unlikely. Therefore, these algorithms are probabilistically correct because the chance of ticket overflow is extremely small.

6.3.7.4 Tournament

The next example shows a general approach for constructing an N-thread solution for mutual exclusion from any 2-thread solution. The approach is based on the notion of a tournament, where each round of the tournament involves only two tasks and the winner of each round progresses towards the goal (critical section), as in a tennis tournament. The unusual aspect of the tournament is that the losing task at each round keeps playing and is guaranteed to eventually become the tournament winner.

Figure 6.12 illustrates the minimal binary-tree for N intents with $\lceil N/2 \rceil$ start nodes and $\lceil \lg N \rceil$ levels to implement the tournament. Note, a minimal rather than a full, balanced binary-tree is sufficient because perfect fairness is not a requirement, only a bound is necessary. Each node is a 2-thread solution for mutual exclusion, such as Dekker or Peterson. Each task is assigned to a particular start node, where it begins the mutual exclusion process (like the

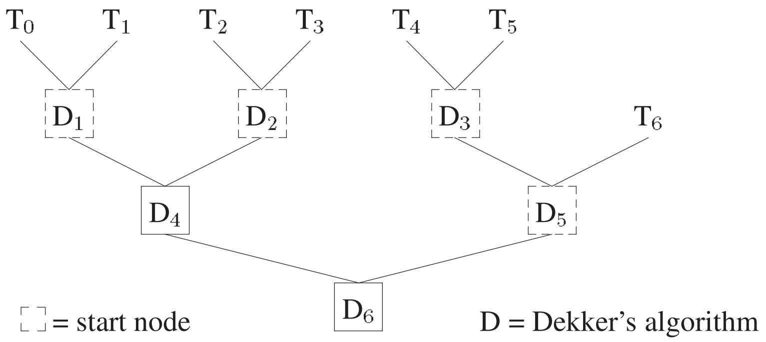


Figure 6.12: N-Thread Tournament

blackboards on the bathroom door except 2 tasks are usually assigned to each starting node). At each node, the 2-thread algorithm ensures each arriving task is guaranteed to make progress because it must guarantee a bound (rule 5); i.e., the loser at each node eventually becomes the winner and continues to the next level. Therefore, each task eventually reaches the root of the tree and enters the critical section. With a minimal binary tree, the tournament approach uses $(N - 1)M$ bits, where $(N - 1)$ is the number of tree nodes and M is the node size (e.g., if Dekker's algorithm is used at each node, the node size is the variables Last, me, you and the link fields to represent the tree).

By this point it should be clear that N-thread solutions to the critical section problem are so complex that only experts can create and validate them. This restricts what can be done by non-experts and requires a high level of trust be placed on their results. Furthermore, all N-thread algorithms require a high degree of communication and cooperation among the participants. Finally, the busy waiting in all the software solutions can be a major source of inefficiency when multiple tasks contented for a critical section. Therefore, software solutions are rejected, in general, as a way of providing mutual exclusion for a critical section. However, in certain specialized cases, a software solution may be required and may work effectively and efficiently.

The reader should not feel time has been wasted examining approaches that are subsequently rejected. In many cases, it is just as important to understand what does not work and why as it is to understand what does work and why. In fact, it is often the case that the latter is only possible after the former.

6.3.7.5 Arbiter

The final software example is a complete N-thread solution to the critical section game by using a full-time arbitrator task to control entry to the critical section (see program in Figure 6.13). However, this approach changes the definition of the problem because the N-threads no longer decide among themselves about entry to the critical section; rather, the N-threads communicate with an arbiter to know when to enter the critical section. In essence, the mutual exclusion problem among N-threads is converted into a a synchronization problem between the N-threads and the arbiter (see Section 5.12, p. 131).

In the entry protocol, a task indicates its intent to enter and waits until the arbiter indicates the start of this task's turn. The exit protocol retracts intent after exiting the critical section and waits until the arbiter indicates the finish of the task's turn. The arbiter cycles around the N intent-flags looking for a task wanting to enter. Once one is found, the arbiter indicates the start of that task's turn, and waits for the task to retract its intent indicating it has exited the critical section. The arbiter then indicates the finish of the task's turn. Because there are no simultaneous assignments to shared data among the arbiter and N tasks, atomic assignment is not required.

There is no indefinite postponement (violation of rule 4) because the arbiter never uses the critical section, it only lets a task in. There is no starvation (violation of rule 5) because the arbiter cycles through the N intent flags so there are at most $N - 1$ tasks ahead of any waiting task.

Unfortunately, this solution has a major problem. The arbiter is continuously busy waiting checking the intent flags even when there is no contention for a critical section, which compounds the inefficiency of busy waiting by contenting tasks. For M critical sections, there are M arbiters spinning constantly, which can consume a substantial amount of the CPU resource. So while the arbiter is largely impractical for software critical sections, the concept of an arbiter is useful and it appears in later approaches.

```
_Task Worker {
    bool *intent, *serving;
    int me, i;

    void main() {
        for ( i = 0; i < 1000; i += 1 ) {
            intent[me] = true;                      // entry protocol
            while ( ! serving[me] ) {}              // wait for turn to start

            CriticalSection();

            intent[me] = false;                     // exit protocol
            while ( serving[me] ) {}                // wait for turn to finish
        }
    }
  public:
    Worker( bool intent[ ], bool serving[ ], int me ) :
        intent( intent ), serving( serving ), me( me ) {}
};
_Task Arbiter {
    bool *intent, *serving, &stop;
    int NoOfTasks;

    void main() {
        int i = 0;
        for ( ;; ) {
            // cycle looking for requests => no starvation
            for ( ; ! intent[i] && ! stop; i = (i + 1) % NoOfTasks ) {} // anyone want in ?
          if ( stop ) break;
            serving[i] = true;                      // start turn for worker
            while ( intent[i] ) {}                  // wait for worker to exit CS
            serving[i] = false;                     // finish turn for worker
        }
    }
  public:
    Arbiter( bool intent[ ], bool serving[ ], bool &stop, int NoOfTasks ) :
        intent( intent ), serving( serving ), stop( stop ), NoOfTasks( NoOfTasks ) {}
};
void uMain::main() {
    const int NoOfTasks = 10;
    Worker *workers[NoOfTasks];                     // pointer to an array of workers
    bool intent[NoOfTasks], serving[NoOfTasks], stop;   // shared
    int i;

    for ( i = 0; i < NoOfTasks; i += 1 ) {          // initialize shared data
        intent[i] = serving[i] = false;
    }
    stop = false;

    Arbiter arbiter( intent, serving, stop, NoOfTasks ); // create arbiter
    for ( i = 0; i < NoOfTasks; i += 1 ) {
        workers[i] = new Worker( intent, serving, i );   // create workers
    }
    for ( i = 0; i < NoOfTasks; i += 1 ) {          // terminate workers
        delete workers[i];
    }
    stop = true;                                    // stop arbiter
}
```

Figure 6.13: N-Thread Arbiter

6.4 Hardware Solutions

Software solutions to the critical section problem cannot be rejected unless there is some alternative. The alternative is to cheat and rely on atomicity provided at the hardware level (as in Peterson's and the N-thread algorithms). At this level, it is possible to make assumptions about execution that are impossible at the software level, e.g., that certain operations are executed atomically, such as assignment, significantly reducing the need for shared information and the checking of this information required in a software solution.

As shown, atomic assignment can be used to simplify mutual exclusion. However, it is straightforward to provide more powerful atomic actions at the hardware level to significantly simplify mutual exclusion. The atomic functionality these instructions provide is the ability to perform an uninterruptable read/write cycle. That is, an instruction reads from memory, possibly performs an action, and then writes to memory guaranteeing that these operations cannot be interrupted; nor can another CPU write to the same memory location during these operations. In the latter case, the memory location is locked during the atomic instruction to other CPUs. Over the years, different computers have provided different atomic instructions. Several of these instructions are examined here, but others exist and each can be used to solve the critical section problem, albeit with varying degrees of complexity and efficiency.

6.4.1 MIPS R4000

An example of a general approach to hardware atomicity is provided on the MIPS R4000 [Mip91] computer and subsequently provided on the Alpha computer [Sit92]. Rather than provide a fixed set of atomic instructions, a general capability exists to build many different atomic operations. This novel approach is used to illustrate several instructions implemented by difference architectures. That is, a multi-line MIPS assembler program is shown that corresponds to single instructions on other architectures; while one instruction may seem better than several instructions, the latter is significantly more general.

The general approach is based on two instructions that capture the necessary requirements of an atomic read/write cycle: LL (load locked) and SC (store conditional). The LL instruction loads (reads) a value from memory into a register, but in addition, it saves the memory address from which the value is fetched. It is then possible to modify the register value through one or more instructions. When the register value is completely modified, it is stored (written) back into the original or another memory location using the SC instruction. However, the store is conditional and occurs only if no interrupt, exception, or interfering write has occurred to the memory location associated with the original LL. If any of these conditions is not met, the store does not occur. Instead, the failure is indicated by setting the register containing the value to be stored to 0.[4]

(The following MIPS assembler programs are simplified by ignoring the delay slots of branch and jump instructions. All the programs require a nop instruction after each branch and jump instruction.)

6.4.2 Test and Set

The test-and-set instruction is the most common atomic instruction. The instruction has one parameter: a pointer to a lock. It is implemented as: read of a memory location (possibly into an implicit register), write a value into the memory location (often the value 1), and return the previous value. All these actions are performed as a single atomic operation without interruption and without change by other CPUs to the memory location. It is written here as a C++ equivalent routine:

```
int testSet( int &lock ) {      // atomic execution
    int temp = lock;            // read
    lock = 1;                   // write
    return temp;                // return previous value
}
```

The equivalent assembler program written using LL and SC is:

[4] Unfortunately, the register is set to 1 on success, so the value being stored is always destroyed. Setting a condition code would have been better.

```
testSet:                    // register $4 contains pointer to lock
    ll    $2,($4)           // read and lock location
    or    $8,$2,1           // set register $8 to 1
    sc    $8,($4)           // attempt to store 1 into lock
    beq $8,$0,testSet       // retry if interference between read and write
    j     $31               // return previous value in register $2
```

Notice the busy wait around the read/write cycle, via the compare and branch, beq, back to label testSet if the store fails.

How to achieve mutual exclusion using test-and-set is answered by going back to the first attempt at a software solution: locking (see Section 6.3.1, p. 140). The problem with the locking solution is the potential for interruption between checking if the lock is open and closing the lock, e.g.:

```
while ( Lock == CLOSED ) {} // entry protocol
// interruption
Lock = CLOSED;
```

This problem can be solved with test-and-set by noticing that after the previous value is returned, the current value is always 1; therefore, the instant the previous value of the lock is returned, the lock is always closed. If the previous value is open, the lock is closed; if the previous value is closed, the lock is closed. In other words, the first task to test-and-set on an open lock reads the open value and snaps the lock closed, atomically; any other task simply closes the locks, atomically. Therefore, the entry protocol simply transforms into:

```
int Lock = OPEN;                // shared lock

while ( testSet( Lock ) == CLOSED ) {}  // entry protocol
// critical section
Lock = OPEN;                    // exit protocol
```

If the lock is open, the loop stops and the lock is set to closed. If the lock is closed, the loop busy waits until another task sets the lock to open.

Unlike a software solution, the hardware solution does not depend on any shared information other than the lock. Furthermore, this solution works even for N tasks trying to enter a critical section. These observations are very significant because there is a substantial decrease in complexity at the software level with only a marginal increase in complexity at the hardware level. Unfortunately, rule 5 is broken, as there is no bound on service; this problem is addressed in subsequent examples.

The first hardware solution to include a bound was presented by Burns [Bur78] (see Figure 6.14). Like the software N-thread solution, there is a bit assigned to each task wanting to enter the critical section; in addition, there is a key used to control the critical section resulting in $N+1$ bits. Conceptually, the task in the critical section has the key, and when it leaves, it either hangs it on a hook outside the critical section if no other task is waiting or passes it directly to a waiting task.

The entry protocol begins with a task declaring its intent to enter the critical section. It then busy waits until either:

- It acquires the entry key via the test-and-set.

  In this case, the key is hanging outside the critical section and the task just atomically picks it up. Like the simple test-and-set solution above, if one or more tasks simultaneously race to pick up the key, the test-and-set ensures only one wins, so there is no indefinite postponement. Notice, the test-and-set implicitly changes the value of the key to non-zero, so it is only necessary to explicitly reset the key (i.e., set to zero) in the program.

- Or it is told that it has been selected to enter next and is conceptually handed the key by the task leaving the critical section.

  In this case, the key is not put back on the hook as a task leaves the critical section, so no other task can access it. Cooperation is being used by the task leaving the critical section instead of just putting the key back on the hook and letting the waiting tasks race for it, like the simple test-and-set solution. It is through this cooperation that a bound is established.

Interestingly, this algorithm works if assignment is not atomic by coding the busy wait in the entry protocol to compare exact values written in the exit protocol. Hence, even if a value fluctuates during assignment, the busy loop only stops after the final value stabilizes.

```
enum Intent { WantIn, DontWantIn };

_Task Worker {
    Intent *intents;
    int &Key, N, posn;

    void main() {
        for ( int i = 0; i < 1000; i += 1 ) {
            intents[posn] = WantIn;                  // entry protocol
            // acquire key or selected to enter
            while ( testSet(Key) != 0 && intents[posn] != DontWantIn ); // busy wait

            CriticalSection();

            intents[posn] = DontWantIn;              // exit protocol
            // locate next task to enter
            int next;
            for ( next = (posn + 1) % N;             // search for waiting task
                    intents[next] != WantIn && next != posn;
                    next = (next + 1) % N ) {}
            if ( next == posn ) {                    // no waiting tasks ?
                Key = 0;                             // return key
            } else {
                intents[next] = DontWantIn;          // stop their busy waiting
            }
        }
    }
  public:
    Worker( int &Key, Intent intents[ ], int N, int posn ) :
        Key( Key ), intents( intents ), N( N ), posn( posn ) {}
};
void uMain::main() {
    const int NoOfTasks = 10;
    int Key;                                         // shared
    Intent intents[NoOfTasks];                       // shared
    Worker *workers[NoOfTasks];                      // pointer to an array of workers
    int i;

    Key = 0;                                         // initialize shared data
    for ( i = 0; i < NoOfTasks; i += 1 ) {
        intents[i] = DontWantIn;
    }
    for ( i = 0; i < NoOfTasks; i += 1 ) {
        workers[i] = new Worker( Key, intents, NoOfTasks, i ); // create workers
    }
    for ( i = 0; i < NoOfTasks; i += 1 ) {           // terminate workers
        delete workers[i];
    }
}
```

Figure 6.14: Bounded Mutual Exclusion with Test-and-Set

The exit protocol begins with a task retracting its intent to enter the critical section. The exiting task must now check to see if there are any waiting tasks. To do this, it cycles from its position around the array of intent bits until either it finds a task wanting entry to the critical section or returns back to the starting location of the search (i.e., its intent bit). If another task is found wanting entry, it is *tricked* into making progress by turning off its intent bit so it terminates its entry-protocol busy-loop (i.e., intents[posn] != DontWantIn is false). Notice, the key is not reset as it is conceptually given to this entering task; therefore, all other tasks continue to see a locked critical section. If no task is found waiting, the key is reset, which conceptually places it back on the hook outside the critical section.

There are several important points about this algorithm. First, the entry protocol spins executing the test-and-set instruction, which normally causes a problem on most multiprocessor computers. The reason is that any atomic instruction on a multiprocessor is expensive because of the need to ensure mutual exclusion to the particular memory location across all CPUs. This problem is mitigated by performing a test-and-set only after detecting an open lock, as in:

```
while ( lock == CLOSED || testSet( lock ) == CLOSED ) {}     // busy wait until OPEN
```

This approach to busy waiting only does the test-and-set when a task exits the critical section and resets the lock; otherwise, only a simple read is performed, which is cheaper than an atomic instruction.[5] Second, this hardware solution still requires one bit for each waiting task, which imposes a maximum on the number of tasks that can wait for entry into the critical section. Furthermore, the cooperation in the exit protocol is $O(N)$ as there is a linear search through the list of intent bits. Lastly, the bound on the waiting time is $N - 1$ because a task may have to wait for all the other tasks to enter and leave the critical section before it can proceed. The bound is ensured because the linear search always starts from the bit position of the last task in the critical section. Therefore, in the worst case of all tasks waiting, the linear search cycles through all the tasks. However, arriving tasks are not serviced in FIFO order because the intent bits are serviced in cycle order not arrival order. A task arriving behind the current intent bit is serviced only after tasks in front of the current intent bit, even though task's behind the current intent bit may have arrived before task's in front of it. Therefore, this algorithm is another example of bounded non-FIFO selection.

6.4.3 Fetch and Increment

The fetch-and-increment instruction performs an atomic fetch and increment of a memory location. The instruction has one parameter: a pointer to a value. It is implemented as: read of a memory location (possibly into an implicit register), increment the value, write the value into the memory location, and return the previous value. All these actions are performed as a single atomic operation without interruption and without change by other CPUs to the memory location. It is written here as a C++ equivalent routine:

```
int fetchInc( int &val ) {      // atomic execution
    int temp = val;             // read
    val += 1;                   // increment and write
    return temp;                // return previous value
}
```

The equivalent assembler program written using LL and SC is:

```
fetchInc:                       // register $4 contains pointer to val
    ll   $2,($4)                // read and lock location
    add  $8,$2,1                // set register $8 to val + 1
    sc   $8,($4)                // attempt to store new val into pointer
    beq  $8,$0,fetchInc         // retry if interference between read and write
    j    $31                    // return previous value in register $2
```

Generating mutual exclusion using atomic increment is similar to that for test-and-set, e.g:

```
int Lock = 0;                   // shared lock

while ( fetchInc( Lock ) != 0 ) {}  // entry protocol
// critical section
Lock = 0;                       // exit protocol
```

[5] There is an additional problem with this approach on multiprocessor computers. When the lock is set opened, all waiting tasks may simultaneously perform a test-and-set, which causes the hardware problem of invalidating multiple CPU caches; see [And90] for solutions to this problem.

The lock is set to 0, and all tasks wanting entry attempt to increment the lock. Because of the atomic increment, only one task makes the transition from 0 to 1. The task that makes this transition is returned the value 0, which indicates it can enter the critical section. All other tasks busy wait, atomically incrementing the lock. When the task in the critical section exits, it resets the lock to zero. This assignment should be atomic in case an increment of a partial result causes a problem. Then, one of the busy waiting tasks causes the next lock transition from 0 to 1 and enters the critical section.

There are two problems with this solution. First, like test-and-set, this solution has no bound on service. Second, the lock counter can overflow from zero back to zero during the busy waiting, which would allow another task into the critical section. The time to overflow the lock counter can range from tens of seconds to several minutes depending on the speed of the computer. As long as the time to execute the critical section is less than this amount, including the time a task might be interrupted by time slicing while in the critical section, the overflow problem can be ignored. Nevertheless, it is a risky solution, especially with the ever increasing speed of modern computers, which shortens the time to overflow, but equally shortens the time to execute the critical section.

A more complex example of how to use the atomic increment to solve mutual exclusion is now presented that solves rule 5, i.e., no starvation. The algorithm builds a true atomic ticket-dispenser, as opposed to the self-generating tickets for Lamport's algorithm, so each task simply takes a ticket and waits for its value to appear before entering the critical section. After exiting the critical section, a task conceptually throws its ticket away (or turns it back in) so it can be reused. Like the previous solutions, there is the potential for ticket overflow. However, this solution allows the tickets to overflow from positive to negative (assuming 2's complement) and back again without problem. The only restriction is that the total number of tasks waiting to use a ticket lock is less than the total number of values that can be represented by an integer, e.g., more than 2^{32} values for most 4-byte integers.

For a change, this lock solution is presented as a class to encapsulate the two variables needed by each lock, as in:

```
class ticketLock {
    unsigned int tickets, serving;
  public:
    ticketLock() : tickets( 0 ), serving( 0 ) {}
    void acquire() {                        // entry protocol
        int ticket = fetchInc( tickets );   // obtain a ticket
        while ( ticket != serving ) {}      // busy wait
    }
    void release() {                        // exit protocol
        serving += 1;
    }
};
```

The acquire member atomically acquires the current ticket and advances the ticket counter so that two tasks cannot obtain the same ticket, except in the case of more than 2^{32} tasks. Once a ticket is acquired, a task busy waits until its turn for service. The release member advances the serving counter so the next task can execute, which is a very simple form of cooperation. Notice, the release member does not use fetchInc to increment serving for the following reasons. First, only one task can be in the release member incrementing server or the mutual exclusion property has been violated. Second, the next task wanting entry cannot make progress until the increment has occurred, and even if it continues before the releasing task exits release, this does not cause a problem because there is no more work performed in release. Finally, this analysis probably requires atomic assignment, even with the inequality check in the busy loop of the entry protocol, because there may be many outstanding ticket values and the server counter could contain several of these values before stabilizing.

This algorithm has many excellent properties. There is no practical limit on the number of waiting tasks because it does not require a bit per task; only $O(1)$ bits are needed (64 bits in this implementation). Tasks are serviced in FIFO order of arrival and the cooperation to select the next task to execute in the critical section is $O(1)$. Only one atomic instruction is executed per entering task; afterwards, the busy wait only reads.

While tasks appear to be finally handled in FIFO order because of the monotonic increasing ticket values due to the atomic increment, which ensures a fair bound on service, the additional busy wait in the implementation of fetchInc using LL and SC, means that a task, in theory, could spin forever attempting to increment tickets. Is this an artifact of the implementation of atomic increment or it is a general problem? That is, if the architecture implemented a single instruction to perform an atomic increment is this problem solved? In fact, the answer is maybe. On a uniprocessor, a single atomic instruction normally precludes time slicing during its execution, so a task never waits to do the increment.

On a multiprocessor, if the architecture forms a FIFO queue of waiting tasks from different CPUs for each memory location involved in an atomic operation, the tasks have a bound on service. However, this begs the question of how the tasks are added to the hardware waiting queue, because operations on the same end of a queue by multiple tasks must be executed atomically. Therefore, there is a recursion, and the fixed point to stop the recursion is always a busy wait; thus, there is always a busy wait somewhere in a computer system, in the software or hardware or both. Most multiprocessor computers actually busy wait in the hardware until the memory location becomes unlocked. Therefore, it may be impossible to guarantee a bound in practice. However, the probability of a task starving is so low that a probabilistic solution at this level is acceptable.

> It is an amusing paradox of critical sections that to implement one, we must appeal to the existence of simpler critical section (called *wait* and *signal*). The implementation of *wait* and *signal* operations, in turn, requires the use of an *arbiter*—a hardware implementation of still simpler critical sections that guarantee exclusive access to a semaphore *(discussed in Chapter 7, p. 177)* by a single processor in a multiprocessor system. This use of nested critical sections continues at all levels of machine design until we reach the atomic level, at which nuclear states are known to be discrete and mutually exclusive. [Bri73, p. 241]

6.4.4 Compare and Swap

The compare-and-swap instruction performs an atomic compare and conditional assignment. To start with, the name of this instruction is a misnomer because it only compares and assigns; no "swapping" or interchanging of values occurs. The instruction has three parameters: a pointer to a value, a comparison value, and a new value for the conditional assignment. It is implemented as: read of a memory location (possibly into an implicit register), compare the value read with the comparison value, and if equal, assign the new value into the memory location. The result of the comparison is returned. All these actions are performed as a single atomic operation without interruption and without change by other CPUs to the memory location. It is written here as a C++ equivalent routine:

```
bool CSwap( int &val, int comp, int nval ) {      // atomic execution
    if ( val == comp ) {          // equal ? assign new value
        val = nval;               // write new value
        return true;              // indicate assignment occurred
    }
    return false;                 // indicate assignment did not occur
}
```

The equivalent assembler program written using LL and SC is:

```
CSwap:                         // register $4, $5, $6 contain val, comp, nval
    move $8,$6                 // copy nval because destroyed by SC
    ll   $2,($4)               // read and lock location
    bne  $2,$5,notequal        // equal ? try to assign new value
    sc   $8,($4)               // try to write new value
    beq  $8,$0,CSwap           // retry if interference between read and write
    li   $2,1                  // set $2 to true
    j    $31                   // return result of comparison in register $2
notequal:
    sc   $2,($4)               // write back current value
    beq  $2,$0,CSwap           // retry if interference between read and write
    li   $2,0                  // set $2 to false
    j    $31                   // return result of comparison in register $2
```

Notice the second SC instruction writes back the original value and is required to form a match for the initial LL. If the second SC is not present, another LL is performed without terminating the first one, which may cause problems on certain architectures.

Here is a simple usage of compare-and-swap to achieve mutual exclusion:

```
int Lock = OPEN;               // shared lock

while ( ! CSwap( Lock, OPEN, CLOSED ) ) {} // entry protocol
// critical section
Lock = OPEN;                   // exit protocol
```

```
struct node {
    node *next;
    bool waiting;
};
class queueLock {
    node *last;
  public:
    queueLock() : last( NULL ) {}
    void acquire( node &n ) {
        node *pred;
        n.next = NULL;
        for ( ;; ) {                        // simulate atomic fetch and store
            pred = last;
          if ( CSwap( &last, pred, &n ) ) break;
        }
        if ( pred != NULL ) {               // someone on list ?
            n.waiting = true;               // mark me as waiting
            pred->next = &n;                // add me to list of waiting tasks
            while ( n.waiting ) {}          // busy wait
        }
    }
    void release( node &n ) {
        if ( n.next != NULL ) {             // someone waiting ?
            n.next->waiting = false;        // stop their busy wait
        } else {                            // no one waiting
            if ( ! CSwap( &last, &n, NULL ) ) { // changed since last looked ?
                while ( n.next == NULL ) {} // busy wait until my node is modified
                n.next->waiting = false;    // stop their busy wait
            }
        }
    }
};
```

Figure 6.15: MCS List-Based Queuing Lock

The compare-and-swap compares the lock with OPEN, and if open, assigns it CLOSED, like the test-and-set solution. Also, like the test-and-set solution, this solution has no bound on service.

Two solutions are examined that put a bound on service. The first solution is based on the one in Section 6.4.3, p. 163 by simply constructing an atomic fetch-and-increment using compare-and-swap.

```
int fetchInc( int &val ) {
    int temp;
    for ( ;; ) {
        temp = val;             // copy val
      if ( CSwap( val, temp, temp + 1 ) ) break;
    }
    return temp;
}
```

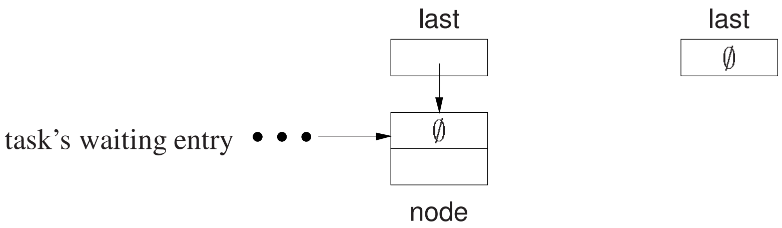
The value of the variable is copied and then an attempt is made to change the value to be one greater. The CSwap only assigns the incremented value into the variable if its value has not changed since it was copied, which can only be true if no other task performed an assignment between the copy and the CSwap. The ticketLock now works as-is using the fetchInc implemented with compare-and-swap. Notice, how this complex atomic instruction can be used to simulate simpler ones.

The second solution is more complex, using a linked-list structure to ensure FIFO ordering of tasks waiting for service. The algorithm is by John M. Mellor-Crummey and Michael L. Scott [MCS91] and is named after their initials, MCS. A slightly modified version of the algorithm appears in Figure 6.15 in the form of a class and using only compare-and-swap. (The original solution uses an atomic fetch-and-store instruction as well as compare-and-swap.)

The lock in Figure 6.15 is used as follows. Each task creates a node to be added to a list if it is necessary to wait, e.g.:

```
queueLock Lock;          // shared lock
  task_n
node n;                  // task node for waiting
Lock.acquire( n );       // entry protocol
// critical section
Lock.release( n );       // exit protocol
```

The node contains a pointer to the next waiting task, possibly null, and a boolean flag for spinning (busy waiting). A queueLock contains a class variable, last, pointing to the last node in a linked list of tasks waiting entry or it is null, as in:



The acquire member starts by initializing the next field of an acquiring task's node to NULL. Then compare-and-swap is used to add this node to the end of the list by fetching the value of the last pointer, and simultaneously storing a pointer to the node for this request into the last pointer. This operation is accomplished by copying the value of the last pointer into pred, and using compare-and-swap to atomically check if the value of last is still the same as the copy, and if so, assign the new value to last. Hence, the value in pred was the last value and the last value is now a pointer to n, which is the new last node in the list. The value of pred (previous value of last) is checked to determine if the list was empty or had waiting tasks. If the list is empty, it is possible to proceed into the critical section. Otherwise, the waiting field is set to true, the predecessor's node is modified to point at the last node, and the acquiring task busy waits on its waiting flag.

The release member checks if any task has chained itself onto the releasing task's node, which is the cooperation necessary to make servicing FIFO. If so, that chained task is about to or is already busy waiting on its waiting flag. By setting that task's waiting flag to false, it stops busy waiting and proceeds. If no task is chained onto this task's node, this task must be the only node on the list, and this task races to remove itself and mark the list null. The problem is that between checking that it is the last node and removing itself, another task can chain itself onto the list so this task's node no longer has a next field that is null. A compare-and-swap is performed to try to set the last pointer to null but only if it is still pointing to this task's node. If the two values are still equal, last is assigned NULL, and the task has successfully removed its node. If the two values are not equal, a task has linked itself onto the end of the list. However, while the new task may have set last, it may not have finished modifying this task's node to point to its node. Therefore, it is necessary to busy wait for the other task to get from the compare-and-swap in acquire to setting this node's next field. Once this is done, the other task's waiting flag can be changed so it proceeds. Notice how important the ordering of assignments is in acquire. The waiting flag must be set to true *before* setting the next field so that in release it is safe to reset the other task's waiting flag once the nodes are linked together. Finally, it is unnecessary to have a global front pointer to the list as the node associated with the task in the critical section is always at the front of the list.

6.4.5 Swap

The swap instruction performs an atomic interchange of two values in memory. The instruction has two parameters: two pointers to the values to be interchanged. It is implemented as: two reads from the memory locations (possibly into implicit registers), and two assignments of the values back into each other's memory locations. All these actions are performed as a single atomic operation without interruption and without change by other CPUs to the memory location. It is written here as a C++ equivalent routine:

```
void Swap( int &val1, int &val2 ) {    // atomic execution
    int temp = val1;           // read
    val1 = val2;               // read and write
    val2 = temp;               // write
}
```

The equivalent assembler program written using LL and SC is:

```
Swap:                          // registers $4 and $5 contain *val1, *val2
    ll   $2,($4)               // read and lock location
    ll   $3,($5)               // read and lock location
    sc   $3,($4)               // attempt to store
    sc   $2,($5)               // attempt to store
    beq  $3,$0,Swap            // retry if interference between read and write
    beq  $2,$0,Swap            // retry if interference between read and write
    j    $31
```

Notice that it is necessary to lock two memory locations to implement this atomic instruction. Unfortunately, the MIPS and Alpha computers allow only one memory location to be locked, so this assembler program cannot work. Furthermore, even the ability to lock two memory locations is insufficient because, if the first store is successful and the second store fails, information has been changed before it is correct. In this case, the memory locations cannot be locked independently, but must be tied together so that a change to one causes a failure during a store to the other. Even though an atomic swap operation cannot be implemented with LL and SC, some computers do provide it as a single atomic instruction.

Here is how a swap instruction can be used to achieve mutual exclusion:

```
int Lock = OPEN;

for ( ;; ) {                   // entry protocol
    int dummy = CLOSED;
    Swap( Lock, dummy );
  if ( dummy == OPEN ) break;
}
// critical section
Lock = OPEN;                   // exit protocol
```

A task specific memory location, dummy, is initialized with the CLOSED value for each task attempting entry to the critical section. Then a busy loop is started swapping dummy and Lock. If Lock has the value OPEN, that value is atomically swapped with dummy. After the swap, dummy has the value OPEN and Lock is CLOSED. Hence, Lock is snapped closed at the same instance that its value it copied into dummy, like in the test-and-set solution. Since each task has its own dummy variable, only one task obtains the lock. The other tasks keep swapping the value CLOSED between Lock and dummy, until a task exits the critical section and sets the Lock to OPEN. Whereupon, one of the other tasks swaps CLOSED into Lock and OPEN into its dummy. Also like the test-and-set solution, this solution has no bound on service.

Suggestions have been made to extend the power of LL and SC to deal with multiple memory locations instead of just one, e.g., adding two instructions [GC96, p. 129]:

1. LLP (load-linked-pipelined): load and link to a second address after a LL. This load is linked to the following SCP.
2. SCP (store-conditional-pipelined): Store to the specified location provided that no modifications have occurred to either of the memory cells designated by *either* of the most recent LL and LLP instructions.

It is now possible to implement an atomic swap:

```
Swap:                          // registers $4 and $5 contain *val1, *val2
    ll   $2,($4)               // read and lock location
    llp  $3,($5)               // read and lock location
    scp  $3,($4)               // attempt to store
    sc   $2,($5)               // attempt to store
    beq  $2,$0,Swap            // retry if interference between read and write
    j    $31
```

The SCP store is buffered pending a successful SC, at which time registers $2 and $3 are written atomically to locations ($5) and ($4), respectively. The number of levels of nested LLP and SCP instructions would depend on the particular hardware architecture. One level is needed for atomic swap, and levels of 3 and 4 are needed to implement some of following exotic atomic instructions.

6.4.6 Exotic Atomic Instructions

Just to show how complex hardware atomicity can be, two different exotic atomic instructions are examined:

- Both the IBM 370 [IBM81a] and Motorola MC680XX [Mot92] computers have two compare-and-swap atomic instructions, CS/CDS and CAS/CAS2, respectively, illustrated approximately by the following C routines:

```
bool CAS( node *&v, node *&c, node *nv ) {
    if ( v == c ) {                  // atomic execution
        c = nv;
        return true;
    } else {
        v = c;
        return false;
    }
}

bool CAS2( node *&v1, node *&c1, int &v2, int &c2, node *nv1, int nv2 ) {
    if ( v1 == c1 && v2 == c2 ) {    // atomic execution
        c1 = nv1; c2 = nv2;
        return true;
    } else {
        v1 = c1; v2 = c2;
        return false;
    }
}
```

(Again, neither of these instructions actually swaps values.) The CAS instruction is more powerful than the CSwap presented in Section 6.4.4, p. 165 as it not only compares two values (pointers rather than integers), and if equal, assigns a new value to the compare value but if unequal, it assigns the compare value to value. The additional assignment when unequal is useful because it returns the new value that has changed since the value was set. The CAS2 instruction is even more powerful than the CAS instruction as it compares and assigns a pair of values. Both instructions are discussed in detail through routines push and pop for adding and removing nodes from a stack.

Similar to the MCS example in Figure 6.15, p. 166, the CAS instruction is used to allow multiple tasks to simultaneously add to one end of a stack, but only one task is removing from the stack, i.e., the task exiting the critical section. This restricted form of access to the data structure is sufficient for the problem of building mutual exclusion. Now the more complex situation of multiple tasks simultaneously adding and removing from a stack is examined. For example, a set of resources, e.g., a group of printers, can have multiple tasks attempting to remove a resource from the set to use it, and subsequently, adding a resource back to the set once used. The following code fragments illustrate the basic parts of the stack:

```
struct node {
    // information about resource
    node *next; // pointer to next resource
};
struct header {
    node *top;
};
```

Using the CAS instruction is it possible to write routines to add and remove nodes to/from a stack, as in:

```
void push( header &h, node &n ) {
    node *t = h.top;                        // copy top node
    for ( ;; ) {                            // busy wait
        n.next = t;                         // link new node to top node
      if ( CAS( t, h.top, &n ) ) break;     // attempt to add new top node
    }
}

node *pop( header &h ) {
    node *t = h.top;                        // copy top node
    for ( ;; ) {                            // busy wait
      if ( t == NULL ) break;               // empty list ?
      if ( CAS( t, h.top, t->next )) break; // attempt to remove top node
    }
    return t;
}
```

In the push routine, the stack top pointer in the header, h.top, is copied to t. Within the busy loop, the stack top pointer is copied into the next field of the new node to be added to the top of the stack. Then a CAS operation is performed on the stack top pointer in the header, h.top, and the new top pointer in t. If the top pointer in the header has not changed since its value was copied into t, the address of the new node is copied into h.top, becoming the new top of the list. This check ensures the value in n.next is still correct, i.e., a node has not been added or removed between the assignment to n.next and trying to set the stack top to point to n. If the top pointer in the header has changed, the CAS copies the changed value into t, so the operation can be tried again with the new stack top pointer. The additional assignment of the CAS when the compare fails eliminates having to reset t with t = h.top in the body of the busy loop.

In the pop routine, the stack top pointer in the header, h.top, is copied to t. Within the busy loop, a check is first made for popping from an empty stack, and NULL is returned. If the stack is not empty, a CAS operation is performed on the stack top pointer in the header, h.top, and the new top pointer in t. If the top pointer in the header has not changed since its value was copied into t, the address of the second node, t->next, is copied into h.top, becoming the new top of the list. This check ensures the value in t->next is still correct, i.e., a node has not been added or removed between computing t->next while trying to set the stack top to point to t->next. If the top pointer in the header has changed, the CAS copies the changed value into t, so the operation can be tried again with the new stack top pointer. As for push, the additional assignment of the CAS when the compare fails eliminates having to reset t with t = h.top in the body of the busy loop.

The pop operation is slightly problematic because the node pointed to by t may have been removed between copying h.top to t and computing t->next, resulting in random data being generated by t->next. In general, this is not a problem because the CAS does not assign the invalid data into the stack header because h.top is unequal to t. (A problem might occur if the reference t->next addresses outside of the program's memory, resulting in an address fault.) However, there is a problem causing failure for a particular series of pops and pushes, e.g., a stack with the basic structure:

h → x → y → z

and a popping task, T_i, with its t set to node x and having extracted node y from t->next in the argument of the CAS. T_i is now time-sliced, and while T_i is blocked, node x is popped, y is popped, and x is pushed again, giving:

h → x → z

Now, when T_i unblocks and performs the CAS, it successfully removes x because it is the same header before the time-slice, but incorrectly sets h.top to its local value of node y, giving:

h → y → ???

where the next pointer of y may have any value, as it could have been added to other lists or the storage freed and reused. The stack data is now lost and the stack is left in an inconsistent state.

There is a probabilistic solution to this problem using a counter and the CAS2 instruction. The trick is to associate a counter with the header node:

```
struct header {
    node *top;
    int count;
};
```

and increment the counter every time a node is added to the list. As a result, a removing task can detect if a node has been removed and re-added to the list because the counter in the header node is increased. To make this work, the CAS2 instruction must be used to atomically read and write both fields of the header node, resulting in the following changes:

```
void push( header &h, node &n ) {
    header t;
    CAS2( t.top, h.top, t.count, h.count, h.top, h.count );
    for ( ;; ) {                                  // busy wait
        n.next = t.top;                           // link new node to first node
      if ( CAS2( t.top, h.top, t.count, h.count, &n, t.count + 1 ) ) break;
    }
}

node *pop( header &h ) {
    header t;
    CAS2( t.top, h.top, t.count, h.count, h.top, h.count );
    for ( ;; ) {                                  // busy wait
      if ( t.top == NULL ) break;                 // empty list
      if ( CAS2( t.top, h.top, t.count, h.count, t.top->next, t.count ) ) break;
    }
    return t.top;
}
```

In the push routine, the entire header is copied to t. Because the header contains two fields, normal assignment cannot be assumed to be atomic as the values may be copied in two steps; in general, structure assignment is non-atomic. Therefore, it is necessary to use a CAS2 to atomically perform the assignment. Interestingly, variable t does not need to be initialized. If the fields of t and h just happen to be equal, the previous values of h are atomically copied back into h, i.e., an idempotent operation. If the values are unequal, the additional assignment of the CAS2 when the compare fails atomically copies both fields of h to t. Within the busy loop, the stack top pointer is copied into the next field of the new node to be added to the top of the list. Then a CAS2 operation is performed on both fields of the stack header. If the stack header has not changed since its value was copied into t, the address of the new node is copied into h.top, becoming the new top of the list, and the counter is incremented. This check ensures the value in n.next is still correct, i.e., a node has not been added or removed between the assignment to n.next while trying to set the stack top to point to n. If the stack header has changed, the CAS2 copies the changed values into t, so the operation can be tried again with the new stack header. The additional assignment of the CAS2 when the compare fails eliminates having to reset t with t = h in the body of the busy loop.

In the pop routine, the entire header is copied to t using the same technique as in push. Within the busy loop, a check is first made for popping from an empty stack, and NULL is returned. If the stack is not empty, a CAS2 operation is performed on both fields of the stack header. If the stack header has not changed since its value was copied into t, the address of the second node, t.top->next, is copied into h.top, becoming the new top of the list. Because the counter value is also checked, it ensures the value in t.top->next is still correct, i.e., a node has not been added or removed between computing t.top->next and trying to set the stack top to point to t.top->next If the stack header has changed, the CAS2 copies the changed value into t, so the operation can be tried again with the new stack header. As for push, the additional assignment of the CAS2 when the compare fails eliminates having to reset t with t = h in the body of the busy loop.

The problem scenario is fixed, e.g., a stack with the basic structure:

h,3 → x → y → z

and a popping task, T_i, with its t set to node x,3 and having extracted node y from t.top->next in the argument of the CAS. T_i is now time-sliced, and while T_i is blocked, node x is popped, y is popped, and x is pushed again, giving:

h,4 → x → z

because the adding of x back onto the stack increments the counter. Now, when T_i unblocks and performs the CAS2, it is unsuccessful because its header x,3 does not match the current stack header, x,4.

As mentioned this solution is only probabilistic because the counter is finite in size (as for the ticket counter). The failure situation occurs if task T_i is time-sliced as above and then sufficient pushes occur to cause the counter to wrap around to the value stored in T_i's header *and* node x just happens to be at the top of the stack when T_i unblocks. Given a 32-bit counter, and the incredibly unlikely circumstance that must occur, it is doubtful if this failure situation could arise in practice.

Finally, none of the programs using CAS and CAS2 have a bound on the busy waiting; therefore, rule 5 is broken.

- To deal with the lack of a bound when using CAS and CAS2, and to handle more complex data structures, the VAX [VAX82] computer has instructions to atomically insert and remove a node to/from the head or tail of a circular doubly linked list, illustrated approximately by the following C routines:

```
struct links {
    links *front, *back;
}
bool INSQUE( links &entry, links &pred ) {     // atomic execution
    entry.front = pred.front; entry.back = pred;  // insert entry following pred
    pred.front.back = entry; pred.front = entry;
    return entry.front == entry.back;             // circular queue previously empty ?
}
bool REMQUE( links &entry ) {
    entry.back.front = entry.front;
    entry.front.back = entry.back.front;
    return entry.front == entry.back;             // circular queue now empty ?
}
```

  While these instructions can insert and remove anywhere in a doubly linked list, insertion and removal is restricted to the list head and tail for multiprocessor usage on the VAX.

The problem these exotic instructions attempt to deal with is interruption, e.g., time slicing, during critical operations, especially in the language runtime system and within the operating system. Imagine the following scenario. A thread enters the **new** operation to allocate storage, where the **new** operation manages free storage using a list data structure. Half way through allocating the storage, a time-slice interrupt occurs. The interrupt handler for the time-slice attempts to allocate storage using **new**; however, the **new** operation cannot be re-entered because it is currently in an inconsistent state with respect to its data structures. Protecting the **new** operation using a lock does not help in this case because the time-slice handler cannot block; it must execute to completion because another time-slice interrupt would result in recursive time-slices, which most likely results in further inconsistencies.

Often the inconsistent state can be reduced to just the manipulation of the data structures used to manage resources. If the linking and unlinking of nodes to and from these data structures can be made atomic, it is often possible to write the routine so it is both wait/lock free and interruptible. The basic approach is to prepare a node, and then with a single atomic operation, add it or remove it from a list.

Fundamentally, these exotic atomic instructions provide ever more complex hardware critical sections on which to build even more complex software critical sections, e.g., reading or writing data to/from a file, which can take thousands of instructions. Unfortunately, the more complex the atomic instruction, the longer it takes to execute it, and hence, the longer the delay between servicing interrupts.

6.5 Wait/Lock Free

Herlihy [] presents a methodology for converting ...

6.6 Roll Backward/Forward

One final approach to mutual exclusion is presented for the special case of a critical section that only manipulates a single object, i.e., a one-to-one correspondence between code and data for a critical section, or a specialized kind of critical section where mutual exclusion is required across all objects using the code. In both cases, it is possible to serialize execution of the code independent of the object being manipulated in the code. For example, the **new** and **delete** operations manage the free storage area, and manipulation of its data structure requires mutual exclusion; hence, there is a one-to-one correspondence between code and object for this critical section. Similar situations occur in the operating system, where there is a single data structure and set of routines to manage that data structure for serial resources. One approach to provide mutual exclusion is to turn off some interrupts, e.g., timer interrupts, on entry and exit to the **new** and **delete** operations, respectively, which ensures mutually exclusive execution of the code with respect to the free-storage data structure. While turning interrupts off might be efficient at the operating-system level, which has direct control of the hardware and can execute privileged instructions, turning off interrupts at the programming-language level can be impossible or expensive because it requires two calls into the operating system to perform this action on behalf of the user program.

Because of the inability or cost to turn off interrupts, an alternate approach is to check at the start of an interrupt if any execution is in one of these specialized critical sections, and if so, to force that execution out of the critical section by either undoing its execution back to the start of the critical section, called **roll backward**, or by doing the remainder of the critical section, called **roll forward**. In either case, the lock for the critical section is not released to prevent another task from acquiring it. After the roll backward/forward, there is logically no task in the critical section, so the interrupt handler can safely enter it, if necessary. When control returns to the interrupted task, it releases the lock for the critical section. To perform either roll backward or forward requires the ability to replace the normal interrupt-handler routines.

The basic approach is to replace the existing interrupt handler routine with one that begins with a check if the program is executing within the code area(s) that must execute mutually exclusively. The new interrupt-handler's check involves comparing the program counter at the time of the interrupt with a range of addresses that denote the start and end of the appropriate code areas, which assumes that the interrupt handler has access to the registers of the program at the time it is interrupted. Notice, there is no check for the object that is being manipulated by the code, only that the program is executing a particular block of code. If the program counter is not in the special code area(s), the normal interrupt handling processing is performed. If the program counter is in the special code area(s), the new interrupt handler then determines how much of the atomic code is executed by the program counter. Then it either undoes the execution of the critical section completed so far by the interrupted program (roll backward) or finishes executing what remains of the critical section (roll forward). Finally, the interrupt handler modifies the return address of the interrupted program to not return to the point of the interrupt in the critical section but to either the start or the end of the block of the critical section that is interrupted. Now the remainder of the interrupt handler can execute knowing there is no execution currently in any of the special critical sections. When the interrupted program restarts after the interrupt, it either redoes the critical section or continues after it, ultimately releasing the lock.

For simple critical sections that end with a single write to memory, like test-and-set, fetch-and-increment and compare-and-swap, it is simple to perform roll backwards because no state changes until the final write. For complex critical sections, like inserting or removing from a linked list, it may be extremely difficult or impossible to roll backwards. In general, it is always possible to roll forward. However, determining how much work has been accomplished and finishing the remaining work can be complex even for simple critical sections. Suffice it to say that performing roll backward or forward can be tricky to implement and difficult to prove correct; furthermore, it is architecture dependent. Nevertheless, once correct, it provides an extremely powerful capability as it allows construction of an arbitrarily large and complex atomic action without busy waiting.

6.7 Summary

Software solutions can generate mutual exclusion out of thin air, that is, mutual exclusion can be constructed without extending a programming language or the hardware it executes on. However, all software solutions are complex and require at least one bit of information be shared among all tasks attempting entry to a critical section. Hardware solutions generate mutual exclusion by providing atomicity at the instruction execution level by restricting interrupts and controlling access to memory. Certain atomic hardware instructions are more powerful than others. Compare-and-swap is considered a reasonably powerful, yet simple to implement, atomic instruction. The more general LL/SC instructions allow a variety of atomic operations to be built. However, the ability to only lock one address is insufficient to build operations like atomic swap or a compare-and-swap that actually *swaps*, not just assigns. There is a strong argument for a few exotic atomic instructions to simplify mutual exclusion in both programming-language runtime systems and in the operating system.

6.8 Questions

1. Explain why there is a bound on Dekker's algorithm and state the bound. That is, if one task has just completed the entry protocol (and is about to enter the critical section) and another arrives, what is the maximum number of times the first task can enter the critical section before the second task may enter. This bound prevents starvation while the critical section is being used. (Not to be confused with indefinite postponement where tasks are attempting entry and no task is using the critical section.)

2. The following solution to the mutual exclusion problem appeared in a letter (i.e., a non-refereed article) to the Communications of the ACM. Unfortunately it does not work. Explain which rule(s) of the critical-section game is broken and the pathological situation where it does not work.

```
bool intent[2] = { false, false };          // shared between tasks
int turn = 0;

_Task Hyman {
    int me, you;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            intent[me] = true;              // entry protocol
            while ( turn != me ) {
                while ( intent[you] ) {}
                turn = me;
            }
            CriticalSection();
            intent[me] = false;             // exit protocol
        }
    }
  public:
    Hyman( int me, int you ) : me(me), you(you) {}
};
void uMain::main() {
    Hyman h0( 0, 1 ), h1( 1, 0 );
}
```

3. The following is a Dekker-like solution to the mutual exclusion problem. Unfortunately it does not work. Explain which rule(s) of the critical-section game is broken and the pathological situation where it does not work. Explain why it never failed during a test of 100,000 tries.

```
enum Intent {WantIn, DontWantIn};

_Task Dekker {
    Intent &me, &you, *&Last;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            for ( ;; ) {                   // entry protocol, high priority busy wait
                me = WantIn;
              if ( you == DontWantIn ) break;
                if ( Last == &me ) {
                    me = DontWantIn;
                    while (you == WantIn){}  // low priority bust wait
                }
            }
            CriticalSection();             // critical section
            Last = &me;                    // exit protocol
            me = DontWantIn;
        }
    }
  public:
    Dekker( Intent &me, Intent &you, Intent *&Last ) : me(me), you(you), Last(Last) {}
};
void uMain::main() {
    Intent me = DontWantIn, you = DontWantIn, *Last = rand() % 2 ? &me : &you;
    Dekker t0(me, you, Last), t1(you, me, Last);
}
```

4. The following is a G. L. Peterson solution to the mutual exclusion problem. Does this solution work if lines 8 and 9 are interchanged? If the solution does not work, explain a pathological situation that results in failure of one rule of the critical section game. Be sure to clearly specify which rule is broken and why.

```
1   enum Intent {WantIn, DontWantIn};
2   Intent *Last;
3
4   _Task Peterson {
5       Intent &me, &you;
6       void main() {
7           for ( int i = 1; i <= 1000; i += 1 ) {
8               me = WantIn;                // entry protocol
9               ::Last = &me;
10              while ( you == WantIn && ::Last == &me ) {}
11              CriticalSection();          // critical section
12              me = DontWantIn;            // exit protocol
13          }
14      }
15    public:
16      Peterson( Intent &me, Intent &you ) : me(me), you(you) {}
17  };
18  void uMain::main() {
19      Intent me = DontWantIn, you = DontWantIn;
20      Peterson t0(me, you), t1(you, me);
21  }
```

5. Show informally whether the software solutions Alternation, Declare Intent, and Retract Intent do or do not satisfy rule 5, i.e., prevent starvation.

6. When write is not atomic, there are several different assumptions about what value is stored during simultaneous write to the same memory location; Section 6.3.6.2, p. 148 assumes the bits are scrambled. Cor-

man et al [CLR92, p. 690] suggest other assumptions:

- arbitrary – an arbitrary value from among those written is stored (but not scrambled);
- priority – the value written by the lowest-indexed CPU is stored;
- combining – the value stored is some combination of the values written, e.g., sum, max.

Discuss the effect of these alternate write assumptions with respect to correct execution of the Dekker and Peterson algorithms.

7. While Peterson's algorithm "cheats" by using atomic assignment, J. L. W. Kessels extended Peterson's algorithm so it no longer requires atomic assignment:

```
enum Intent { WantIn, DontWantIn };

_Task Kessels {
    int who;
    Intent &me, &you;
    int &Last1, &Last2;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            me = WantIn;                          // entry protocol
            Last1 = ( Last2 + who ) % 2;          // race
            while ( you == WantIn && Last1 == ( Last2 + who ) % 2 ) {}
            CriticalSection();                    // critical section
            me = DontWantIn;                      // exit protocol
        }
    }
  public:
    Kessels( int who, Intent &me, Intent &you, int &Last1, int &Last2 ) :
        who(who), me(me), you(you), Last1(Last1), Last2(Last2) {}
};
void uMain::main() {
    Intent me = DontWantIn, you = DontWantIn;   // shared
    int Last1, Last2;
    Kessels t0(0, me, you, Last1, Last2), t1(1, you, me, Last2, Last1);
}
```

(a) Explain, briefly, how this algorithm works. Include tables or diagrams, if useful.

(b) Explain what initial values are required for Last1 and Last2.

(c) Why is the atomic assignment assumption of Peterson not needed in this algorithm?

8. The fetch-and-increment instruction provides the old value of a counter before incrementing it. Imagine just an atomic increment instruction that does not return the old value. Is it possible to construct mutual exclusion with this instruction? If so, is it possible to construct a solution that has a bound on service?

9. Use the CAS2 instruction as defined in Section 6.4.6, p. 169 to build routines front and back, which remove a node from the front and add a node to the back of a singly-lined list, respectively.

Chapter 7

Threads and Locks

The previous two chapters (Chapters 5, p. 115 and 6, p. 137) introduce the notions of synchronization and mutual exclusion. Understanding the difference between these crucial concepts is a significant step in designing and building concurrent programs. In general, *synchronization* defines a timing relationship among tasks and *mutual exclusion* defines a restriction on access to shared resources. While both synchronization and mutual exclusion share the notion of constraining operations in time, the practical temporal differences are so profound they should always be considered as independent concepts.

Synchronization is concerned with maintaining a timing relationship, which includes actions happening at the same time, e.g. synchronized swimming, or happening at the same relative rate, e.g. synchronizing video/audio signals, or simply some action having to occur before another (precedence relationship), e.g. synchronizing manufacturing operations. Synchronization is accomplished using a variety of concurrent techniques, such as latches, barriers, rendezvous, etc., and often its main purpose is for communication of data. The key point is that synchronization always requires two or more tasks because a single task is self-synchronized, i.e., a single task does not require any concurrent techniques to interact with itself.

Simple mutual exclusion ensures only one task is in a critical section at a time. Complex mutual exclusion may allow multiple tasks in a critical section but with some "complex" restriction among tasks, such as allowing multiple readers to share a resource simultaneously but writers must be serialized. Any solution to mutual exclusion (simple or complex) must provide appropriate access restrictions, assume that order and speed of task execution is arbitrary, that tasks not in the critical section cannot prevent other tasks from entering, that tasks do not postpone indefinitely when deciding to enter the critical section, and finally, there is often some notion of fairness (guarantee) on entry so no task starves (short or long term). These rules (see Section 8.2.3.2, p. 234 for details) are significantly more complex than those for establishing a timing relationship with synchronization.

Specifically, the differences between synchronization and mutual exclusion are as follows:

1. Mutual exclusion does not require an interaction among tasks when entering a critical section. That is, when there is no simultaneous arrival at a critical section or no task is in the critical section, a task may enter the critical section without interacting with another task. For a lightly accessed critical section, a task may seldom encounter another task; nevertheless, the mutual exclusion is still required. Conversely, synchronization mandates an interaction (rendezvous) with another task before one or more tasks can make progress. Fundamentally, synchronization of a task with itself is meaningless, because a task is always synchronized with itself.

2. Mutual exclusion forbids assuming any specific timing relationship among tasks; synchronization enforces a timing relationship. If two (or more) tasks arrive simultaneously at a critical section, this is a random occurrence, not synchronization between the tasks. Therefore, just because a task blocks (spins) as part of mutual exclusion, the delay does not imply any synchronization between a delayed task(s) and the task(s) using the critical section. For mutual exclusion, such a delay is arbitrary, and therefore, cannot be used for any kind of synchronization notification or communication.

3. Mutual exclusion must not prevent entry to an inactive critical section; synchronization must prevent progress until a partner arrives.

4. Mutual exclusion always protects shared resources, while synchronization does not require any sharing.

Hence, these two concepts are neither similar nor a subset of one another. Furthermore, synchronization and mutual exclusion mistakes result in completely different kinds of errors (see Chapter 8, p. 231).

Because synchronization and mutual exclusion are so different, it is reasonable to imagine that very different mechanisms are needed to implement them. However, it is sometimes possible to use a single lock to perform both functions. Given a general software/hardware lock described in Chapter 6, p. 137, it is possible to construct mutual exclusion in the following way:

```
lock = OPEN;
      task1                      task2
...                          ...
lock entry code              lock entry code
critical section             critical section
lock exit code               lock exit code
...                          ...
```

The lock starts unlocked (open) so a task may enter the critical section, and the entry/exit code provides the necessary mutual exclusion. Now using the same lock, it may be possible to construct synchronization in the following way:

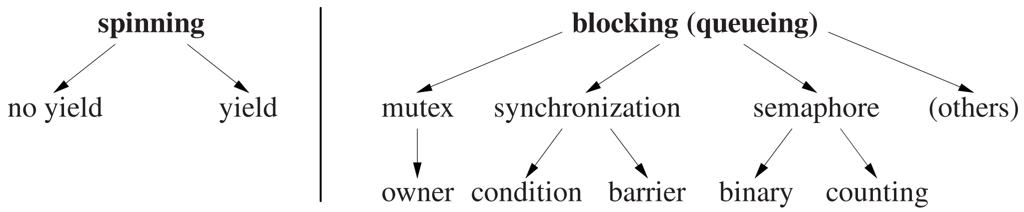
```
lock = CLOSED;
      task1                      task2
...                          ...
S1                           lock entry code
lock exit code               S2
...                          ...
```

Here, the lock starts locked (closed) instead of unlocked (opened), which seems peculiar. Also, the entry and exit code no longer bracket a critical section; instead, the protocol is asymmetric between the synchronizing tasks. The trivial case is when $task_1$ executes S1 before $task_2$ tries to execute S2, and thus, $task_1$ executes the exit code that opens the lock; when $task_2$ reaches the entry code, no waiting is necessary. In the opposite situation, $task_2$ reaches the entry code before $task_1$ gets to or finishes execution of S1. The trick here is that the lock is initially closed, therefore $task_2$ waits for the lock to be opened. Only after $task_1$ completes S1 and the lock exit-code is the lock opened so that $task_2$ can make progress. In fact, this approach is similar to the technique used in the communication program in Figure 5.12, p. 131. Thus, some locks can be used to control synchronization *and* provide mutual exclusion, allowing tasks to communicate during execution and access shared resources using a single mechanism.

This chapter focuses on abstracting the detailed, low-level mechanisms for constructing synchronization and mutual exclusion into different kinds of high-level locks, and then how to use these locks to write concurrent programs that require synchronization and mutual exclusion.

7.1 Lock Taxonomy

Locks can be divided into two general categories: spinning and blocking:



Spinning locks continuously check (busy wait) for an event to occur, implying the waiting task is solely responsible for detecting when the event happens. (While checking, a task oscillates between ready and running states due to either time slicing or yielding.) Blocking locks do not busy wait, but rather block until an event occurs, implying some *other* mechanism must unblock the waiting task when the event happens. Within each of the general categories, there are different kinds of spinning and blocking locks, which are discussed in detail.

7.2 Spin Lock

A **spin lock** is the general abstraction for a lock built from a software approach (e.g., Dekker) or hardware instruction (e.g., test-and-set). As the name suggests, a spin lock loops (spins) continuously checking for an event to occur, which is busy waiting. Unfortunately, busy waiting makes a spin lock impractical for providing *general* synchronization or mutual exclusion, for the following reason. In all the examples so far, if a task is busy waiting, it loops until:

1. a critical section becomes unlocked or an event happens,
2. or the waiting task is pre-empted at the end of its time-slice and put back on the ready queue.

On a multiprocessor, a waiting task may consume substantial amounts of CPU time spinning until another task indicates an event has occurred. A uniprocessor has the same problem, and it is exacerbated by the fact that all checking after the first check is superfluous because the event cannot occur until another task is scheduled, completes its action, and indicates the event, as there is only one CPU. Thus, any further spinning to check for the event by the waiting task during its time-slice is futile. In fact, a waiting task may experience multiple periods of futile spinning, until another task eventually indicates the occurrence of an event. To increase efficiency in the uniprocessor case, a task could explicitly terminate its time-slice and move back to the ready state after the first event check fails. For example, in any of the software solutions presented in Chapter 6, p. 137, placing a call to yield() (see Section 5.9.2, p. 128) in the body of the entry-protocol busy-loops can decrease the execution time by up to two orders of magnitude on a uniprocessor! This optimization can be adapted to the multiprocessor case by having a spinning task terminate its time-slice and move back to the ready state after N checks have failed; other tasks would then have an opportunity to execute on that CPU. However, in the multiprocessor case, this optimization could produce the opposite effect, i.e., cause the entire execution of the program to slow down. The reason is that terminating a time-slice puts the task back on the ready queue, and the length of the ready queue could be long if there are a large number of tasks. Thus, even if the multiprocessor has several CPUs, a task might still have to wait a substantial period of time to get to the front of the ready queue to run on one. Alternatively, if the task just remains spinning, it might detect the event much earlier assuming the task making the change is already scheduled on another CPU. In essence, it is a gamble by the spinning task to remain spinning to obtain quick response or yield its time-slice to better utilize the CPU overall. Some systems, like μC++ (see Chapter 12, p. 353), allow the duration of spinning to be adjusted so an application can find a balance between these conflicting goals, called an **adaptive lock**. Finally, depending on how the spin lock is implemented, it may break rule 5, i.e., no bound on service, possibly resulting in starvation of one or more tasks.

Nevertheless, the spin lock is appropriate and necessary in situations where there is no other work to do. For example, in the μC++ multiprocessor kernel, a spin lock is used to protect access to the ready queue to ensure safety when multiple CPUs are adding and removing tasks from this queue. If a CPU cannot acquire the ready-queue spin-lock, it *must* spin because there is no other work it can do until it acquires the spin lock because all work comes from ready tasks on this queue. Furthermore, a CPU cannot block because it is always running (except for the special case where the CPU powers-down to save electricity). Therefore, a spin lock is necessary in a concurrent system but not necessarily at the user level.

7.2.1 Spin Lock Details

μC++ provides a non-yielding spin lock, uSpinLock, and a yielding spin lock, uLock. Both locks are built directly from an atomic hardware instruction; the particular hardware instruction depends on the architecture on which μC++ is running. These locks are either closed (0) or opened (1), and waiting tasks compete to acquire the lock after it is released. The competition means waiting tasks can be selected in any order. In theory, arbitrary selection order could result in starvation if tasks are constantly contending for the lock and one unlucky task never acquires it; in practice, this scenario seldom occurs.

The specific lock details are:

```
class uSpinLock {
  public:
    uSpinLock();  // open
    void acquire();
    bool tryacquire();
    void release();
};
uSpinLock x, y, *z;
z = new uSpinLock();
```

```
class uLock {
  public:
    uLock( unsigned int value = 1 );  // default open
    void acquire();
    bool tryacquire();
    void release();
};
uLock x, y, *z;
z = new uLock( 0 );  // closed
```

The declarations create six lock variables, all initialized to open, except the last uLock variable, which is explicitly set closed.

uSpinLock in μC++ is non-preemptive, meaning once a task acquires the lock on a particular CPU, no other task may execute on that CPU, which allows performance optimizations and additional error checking in the μC++ kernel, where uSpinLock is used extensively. A consequence of the non-preemptive restriction is that uSpinLock can only be used for mutual exclusion because the task acquiring the lock must be the task releasing it, and hence, its constructor does not take a starting state and instances are initialized to open.

The constructor routine uLock has the following form:

uLock(**unsigned int** value = 1) – this form specifies an initialization value for the lock. Appropriate values are 0 and 1. The default value is 1.

The following discussion applies to both uSpinLock and uLock. The member routines acquire and release are used to atomically acquire and release the lock, closing and opening it, respectively. acquire acquires the lock if it is open, otherwise the calling task spins waiting until it can acquire the lock. The member routine tryacquire makes one attempt to try to acquire the lock, i.e., it does not wait. tryacquire returns **true** if the lock is acquired and **false** otherwise. This method allows a thread to check if an event has occurred without blocking when it has other work to do; the thread recheck for the event after the work is completed. release releases the lock, which allows any waiting tasks to compete to acquire the lock. Any number of releases can be performed on a lock as a release only (re)sets the lock to open (1).

It is *not* meaningful to read or to assign to a lock variable, or copy a lock variable, e.g., pass it as a value parameter. Why is this true? Copying a lock would, in essence, provides another way to open the door (like copying the key for a real lock). Hence, copying must be prevented to ensure mutual exclusion.

7.2.2 Synchronization

In synchronization, two or more tasks must execute a section of code in a particular order, e.g., a block of code S2 must be executed only after S1 has completed. In Figure 7.1(a), two tasks are created in uMain::main and each is passed a reference to an instance of uLock. As in previous synchronization examples, the lock starts closed instead of open so if task t2 reaches S2 first, it waits until the lock is released by task t1. As well, the synchronization protocol is separated between the tasks instead of surrounding a critical section.

7.2.3 Mutual Exclusion

In simple mutual exclusion, two or more tasks execute a section of code one at a time, e.g., a block of code S1 and S2 must be executed by only one task at a time. In Figure 7.1(b), two tasks are created in uMain::main and each is passed a reference to an instance of uLock. Each task uses the lock to provide mutual exclusion for two critical sections in their main routine by acquiring and releasing the lock before and after the critical section code. When one task's thread is in a critical section, the other task's thread can execute outside the critical sections but must wait to gain entry to the critical section.

Does this solution afford maximum concurrency, that is, are tasks waiting unnecessarily to enter a critical section? The answer depends on the critical sections. (Remember, a critical section is a pairing of data and code.) If the critical sections are disjoint, that is, they operate on an independent set of variables, one task can be in one critical section while the other task is in the other critical section. If the critical sections are not disjoint, i.e., they operate on one or more common variables, only one task can be in either critical section at a time. To prevent inhibiting concurrency if the critical sections are disjoint, a separate lock is needed for each critical section. In the example program, if the critical sections are independent, two locks can be created in uMain::main and passed to the tasks so maximum concurrency can be achieved.

7.3 Blocking Locks

As pointed out in Section 7.2, a spin lock has poor performance because of busy waiting. Can anything be done to remove the busy wait? The answer is (mostly) yes, but only if the task releasing the lock is willing to do some additional work. This work is the cooperation mentioned at the end of Section 6.3.7.2, p. 152. That is, the responsibility for detecting when a lock is open is not borne solely by the waiting tasks attempting to acquire the lock, but also shared with the task releasing the lock. For example, when a person leaves the bathroom, if they tell the next person waiting that the bathroom is empty, that is cooperation. What advantage does the releasing task gain in doing this extra work,

```
_Task T1 {
    uLock &lk;

    void main() {
        . . .
        S1
        lk.release();
        . . .
    }
  public:
    T1( uLock &lk ) : lk(lk) {}
};
void uMain::main() {
    uLock lock(0);
    T1 t1( lock );
    T2 t2( lock );
}
```

```
_Task T2 {
    uLock &lk;

    void main() {
        . . .
        lk.acquire();
        S2
        . . .
    }
  public:
    T2( uLock &lk ) : lk(lk) {}
};
```

(a) Synchronization

```
_Task T {
    uLock &lk;

    void main() {
        . . .
        lk.acquire();
        // critical section
        lk.release();
        . . .
        lk.acquire();
        // critical section
        lk.release();
        . . .
    }
  public:
    T( uLock &lk ) : lk(lk) {}
};
void uMain::main() {
    uLock lock;
    T t0( lock ), t1( lock );
}
```

(b) Mutual Exclusion

Figure 7.1: Spin Lock

which has no immediate benefit to it? The answer is that in the long run each task has to wait to use the bathroom, so cooperating benefits both (all) people using the bathroom. In general, each task does a little extra work so that all tasks perform better.

7.3.1 Mutex Lock

A **mutex lock** is used only for mutual exclusion, and tasks waiting on these locks block rather than spin when an event has not occurred. Restricting a lock to just mutual exclusion is usually done to help programmers clearly separate lock usage between synchronization and mutual exclusion, and possibly to allow special optimizations and checks as the lock only provides one specialized function. Mutex locks are divided into two kinds: single and multiple acquisition. Single acquisition is non-reentrant, meaning the task that acquires the lock (lock owner) cannot acquire it again, while multiple acquisition is reentrant, meaning the lock owner can acquire it multiple times, called an **owner lock**.

Single-acquisition locks have appeared in all previous examples of mutual exclusion. However, single acquisition cannot handle looping or recursion involving the same lock, e.g.:

```
void f() {
    . . .
    lock.acquire();
    . . . f();              // recursive call within critical section
    lock.release;
    . . .
}
```

The recursive call within the critical section fails after the first recursion because the lock is already acquired, which prevents further entry into the critical section by the lock owner. (This situation is called a deadlock and is discussed in Chapter 8, p. 231.) Using the bathroom analogy, this situation corresponds to a person acquiring mutual exclusion to the bathroom, and then needing to momentarily step out of the bathroom because there is no soap. Clearly, the person using the bathroom does not want to release the bathroom lock because someone could then enter the bathroom. Hence, the lock must stay locked to other tasks but allow the task that has currently acquired the lock to step out of and back into the bathroom, allowing the person to get some soap. While in theory it is possible to restructure a program to eliminate this situation (e.g., always bring extra soap into bathroom just in case there is none); in practice, the restructuring is difficult and in some case impossible if code is inaccessible (e.g., library code).

Multiple-acquisition locks solve this problem by allowing the lock owner to reenter the critical section multiple times. Using the bathroom analogy, once a person has acquired the bathroom lock, they should be able to enter and leave the bathroom as many times as they want before releasing the lock for someone else. To allow multiple acquisition, an owner lock remembers the task that has acquired the lock, and that task is allowed to acquire the lock again without blocking. The number of lock releases depends on the particular implementation of the owner lock. Some implementations may require only one release, regardless of the number of acquires, while other implementations may require the same number of releases as acquires.

Figure 7.2(a) presents an outline for the implementation of a mutex (owner) lock. The blocking aspect of a mutex lock requires two changes: a waiting tasks blocks when it cannot acquire the lock and cooperation occurs at release to unblock a waiting task, if one is present. A task blocks by linking itself onto a list, marking itself blocked, and yielding its time-slice. Since this task is not on the ready queue, it is ineligible for execution until the task releasing the lock transfers it from the blocked list to the ready queue. Because adding and removing operations of a node to/from a list are not atomic, mutex locking is necessary to protect the critical section within the implementation of this blocking mutex-lock. The only mechanisms available to provide the locking is the nonblocking spin-lock (or a wait-free mechanism, see Section 6.5, p. 172), which is used to provide the necessary mutual exclusion. It is necessary to release the spin lock before blocking otherwise a releasing task cannot acquire the lock to perform the unblocking action (cooperation). After unblocking, a task must re-acquire the spin lock if there are any additional operations needing mutual exclusion. It is necessary to use the flag variable inUse to indicate if the lock is currently acquired because there may not be any task on the blocked queue when a new task attempts to acquire the lock. To make cooperation work, the releasing task does not reset the inUse flag if it unblocks a waiting task; hence, no new task can acquire the lock because it appears to be in use. When the unblocked task restarts, it can progress without further checking because it has been granted acquisition directly from the previous unblocking task. Making the mutex lock an owner lock requires remembering which task currently has the lock acquired, and checking during a lock acquire so the lock owner does not block.

Figure 7.2(b) presents an alternative approach using a "trick" to eliminate the flag variable to indicate if the lock is acquired. After the releasing task acquires the spin lock and decides to unblock a task, *it does not release the spin lock*, which is accomplished by moving the spin-lock release into the **else** clause in member release. The releasing task may not access any of the shared data associated with the lock after restarting a blocked task because it may start execution immediately. When the blocked task restarts in member acquire, the spin-lock acquire after the yield is removed because the spin lock has not been released, so the unblocked task can safely perform any additional operations needing mutual exclusion. In effect, the mutual exclusion provided by the spin lock is transfered directly from the releasing task to the unblocking task. Here, the critical section is not bracketed by the spin lock; instead, the acquire and release are performed in separate blocks of code, similar to lock usage for synchronization.

Finally, both of the previous implementations can guarantee a bound on waiting by servicing the blocked tasks in FIFO order (e.g., by using a queue for the waiting tasks). If fairness is unnecessary, for Figure 7.2(a), the flag is always reset in release and the **if** statement in member acquire is changed to a **while** loop. For Figure 7.2(b), only the latter change is necessary. Hence, after a task unblocks it polls to see if it must block again because the lock may have been acquired by a task that has "barged" ahead of it when the spin lock was released. Without fairness, tasks can experience short and long term starvation; however, in circumstances where any task may an action, it does not matter if some wait while others work, as long as the work gets done. (The issues of barging tasks, starvation, and work sharing are discussed further in this and other chapters.)

7.3.1.1 Owner Lock Details

μC++ only provides a multiple-acquisition lock (owner lock) because it subsumes the notion of a single-acquisition mutex lock; i.e., if a task only acquires an owner lock once before a release, the lock behaves as a single-acquisition lock. The type uOwnerLock defines an owner lock:

```
class uOwnerLock {
  public:
    uOwnerLock();
    unsigned int times() const;
    uBaseTask *owner() const;
    void acquire();
    bool tryacquire();
    void release();
};
uOwnerLock x, y, *z;
z = new uOwnerLock;
```

```
class MutexLock {
      spinlock lock;              // nonblocking lock
      queue<Task> blocked;        // blocked tasks
      bool inUse;                 // resource being used ?
      Task *owner                 // optional
   public:
      MutexLock() : inUse( false ), owner( NULL ) {}
      void acquire() {
            lock.acquire();
            if ( inUse
                   && owner != thistask() ) { // (optional)
                  // add self to lock's blocked list
                  lock.release();   // release before blocking
                  // yield
                  lock.acquire();   // re-acquire lock
            }
            inUse = true;
            owner = thistask();   // set new owner (optional)
            lock.release();
      }
      void release() {
            lock.acquire();
            if ( ! blocked.empty() ) {
                  // remove task from blocked list
                  //    and make ready
            } else {
                  inUse = false;
                  owner = NULL;   // no owner (optional)
            }
            lock.release();       // always release lock
      }
};
```

(a) Implementation 1

```
class MutexLock {
      spinlock lock;              // nonblocking lock
      queue<Task> blocked;        // blocked tasks

      Task *owner                 // optional
   public:
      MutexLock() : owner( NULL ) {}
      void acquire() {
            lock.acquire();
            if ( ! blocked.empty()
                   && owner != thistask() ) { // (optional)
                  // add self to lock's blocked list
                  lock.release();   // release before blocking
                  // yield
                  // UNBLOCK HOLDING LOCK

            }
            owner = thistask();   // set new owner (optional)
            lock.release();
      }
      void release() {
            lock.acquire();
            if ( ! blocked.empty() ) {
                  // remove task from blocked list
                  //    and make ready
                  // DO NOT RELEASE LOCK
            } else {
                  owner = NULL;   // no owner (optional)
                  lock.release();   // conditionally release lock
            }
      }
};
```

(b) Implementation 2

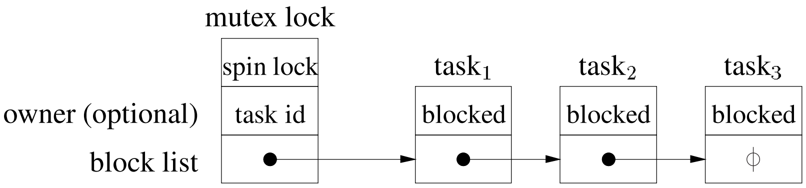


Figure 7.2: Mutex-Lock Implementation

The declarations create three owner-lock variables and initializes them to open. An owner lock is owned by the task that acquires it; all other tasks attempting to acquire the lock block until the owner releases it. The owner of an owner lock can acquire the lock multiple times, but a matching number of releases must occur or the lock remains in the owner's possession and other tasks cannot acquire it.

The member routine times returns the number of times the lock has been acquired by the lock owner. The member routine owner returns the task owning the lock or NULL if there is no owner. The member routine acquire acquires the lock if it is open, otherwise the calling task blocks until it can acquire the lock. The member routine tryacquire makes one attempt to try to acquire the lock, i.e., it does not block; the value **true** is returned if the lock is acquired and **false** otherwise. The member routine release releases the lock, and if there are waiting tasks, one is restarted; waiting tasks are released in FIFO order.

It is *not* meaningful to read or to assign to an owner lock variable, or copy an owner lock variable (e.g., pass it as a value parameter).

7.3.1.2 Mutual Exclusion

A mutex lock is used exactly the same way as a spin lock to construct mutual exclusion (see Figure 7.1(b), p. 181). The lock is passed by reference to the participating tasks, which bracket one or more critical sections by acquiring and releasing the lock. When one task's thread is in a critical section, the other task's thread can execute outside the critical sections but must wait to gain entry to the critical section. If the mutex lock is an owner lock, a lock owner may enter the critical section multiple times.

One example of owner-lock usage in μC++ is controlling I/O to a C++ stream. Because a stream may be shared by multiple tasks, characters generated by the insertion operator (<<) and/or the extraction operator >> in different tasks may be intermixed. For example, if two tasks execute the following:

```
task1 : cout << "abc " << "def " << endl;
task2 : cout << "uvw " << "xyz " << endl;
```

some of the different outputs that can appear are:

```
abc def
uvw xyz
uvw abc def
xyz
abc uvw xyz
def
uvw abc xyz def

abuvwc dexfyz
```

In fact, concurrent operations can even corrupt the internal state of the stream, resulting in failure. As a result, some form of mutual exclusion is required for concurrent stream access. A coarse-grained solution is to perform all stream operations (e.g., I/O) via a single task, providing the necessary mutual exclusion for the stream. A fine-grained solution is to have a lock for each stream, which is acquired and released around stream operations by each task.

μC++ provides a fine-grained solution where an owner lock is acquired and released indirectly by instantiating a type that is specific to the kind stream: type isacquire for input streams and type osacquire for output streams. For the duration of objects of these types on an appropriate stream, that stream's owner lock is held so I/O for that stream occurs with mutual exclusion within and across I/O operations performed on the stream. The lock acquire is performed in the object's constructor and the release is performed in the destructor. The most common usage is to create an anonymous object to lock the stream during a single cascaded I/O expression, e.g.:

```
task1 : osacquire( cout ) << "abc " << "def " << endl; // anonymous locking object
task2 : osacquire( cout ) << "uvw " << "xyz " << endl; // anonymous locking object
```

constraining the output to two different lines in any order:

```
abc def | uvw xyz
uvw xyz | abc def
```

The anonymous locking object is only deallocated after the entire cascaded I/O expression is completed, and it then implicitly releases the stream's owner lock in its destructor.

Because of the properties of an owner lock, a task can allocate multiple locking objects for a specified stream, and the stream's owner lock is only released when the topmost locking object is deallocated. Therefore, multiple I/O statements can be protected atomically using normal block structure, e.g.:

```
{   // acquire the lock for stream cout for block duration
    osacquire acq( cout ); // named stream locker
    cout << "abc";
    osacquire( cout ) << "uvw " << "xyz " << endl; // ok to acquire and release again
    cout << "def";
}   // implicitly release the lock when "acq" is deallocated
```

This handles complex cases of nesting and recursion with respect to reading or printing, e.g.:

```
int f( int i ) {
    if ( i > 0 ) osacquire( cout ) << g( i - rand() % 2 ) << endl;
    return i;
}
```

```
int g( int i ) {
    if ( i > 0 ) osacquire( cout ) << f( i - rand() % 2 ) << endl;
    return i;
}
```

Here, the mutually recursive routines both print and so they both want to acquire mutual exclusion at the start of printing. (Inserting debugging print statements often results in nested output calls.) Hence, once a task acquires the I/O owner lock for a stream, it owns the stream until it unlocks it, and it can acquire the lock as many times as is necessary after the first acquisition. Therefore, the previous example does not cause a problem even though a single task acquires and releases the I/O owner lock many times.

7.3.2 Synchronization Lock

In contrast to a mutex lock, a **synchronization lock** is simpler and used solely for synchronization. Restricting a lock to just synchronization is usually done to help programmers clearly separate lock usage between synchronization and mutual exclusion, and possibly to allow special optimizations and checks as the lock only provides one specialized function. Synchronization locks is often called a **condition lock**, with wait/signal(notify) for acquire/release. While a synchronization lock has internal state to manage tasks blocked waiting for an event, it does not retain state about the status of the event it may be associated with. That is, a synchronization lock does not know if the event has or has not occurred; its sole purpose is to block task and unblock tasks from a list, usually in FIFO order.

Like the implementation of a mutex lock, the blocking aspect of a synchronization lock requires two changes: a waiting task needs to block when the event has not occurred, and cooperation occurs at release to unblock a waiting task, if one is present. A task blocks by linking itself onto a list, marking itself blocked, and yielding its time-slice. Since this task is not on the ready queue, it is ineligible for execution until the task releasing the lock transfers it from the blocked list to the ready queue. Notice the adding and removing operations of a node to/from a list are not atomic; hence, mutex locking is necessary to protect the critical section within the implementation of a blocking synchronization-lock. Synchronization locks are divided into two kinds, external or internal locking, depending on how the mutex locking is provided. External means the synchronization lock uses an external mutex lock to protect its state, while internal means the synchronization lock uses an internal mutex lock to protect its state.

The form of the external locking for a condition lock can vary. Figure 7.3(a) presents an outline for the implementation of an unprotected condition lock, meaning the programmer must use a programming convention of bracketing all usages of the condition lock with mutual exclusion provided via a mutex lock. To accomplish this, it is necessary to pass the mutex lock to the acquire routine because this routine *must both block and release the mutex lock* otherwise a releasing task cannot acquire the (external) mutex lock to perform the unblocking action (cooperation). It is also the programmer's responsibly to use the same mutex lock for a particular usage of the condition lock. While different mutex locks can be used to protect the condition lock at different times, great care must be exercised to ensure no overlapping use of a condition lock among multiple mutex locks or using the same mutex lock for different condition locks.

Figure 7.3(b) presents an outline for the implementation of a protected condition lock, meaning the condition lock directly uses a mutex lock, which is supplied by the programmer at creation of the condition lock. This particular implementation binds a condition lock to a specific mutex lock, restricting the condition lock to a single mutex lock but reducing the chances for errors in usage; because the mutex lock is fixed, it is no longer passed as a parameter to member acquire. Since the condition lock in this case manages its own mutual exclusion, it can use any of the approaches used for the mutex lock discussed in Section 7.3.1, p. 181. The approach used in Figure 7.3(b) is the locking trick of transferring the mutual exclusion between releasing and unblocking task, but other approaches could be used.

Figure 7.4, p. 187 presents an outline for the implementation of an internal synchronization lock, which provides its own internal mutual exclusion. The additional state necessary is a lock for mutual exclusion (usually a spin lock, as in the mutex lock) and a flag to know when the event has occurred. While not absolutely required, it is often extremely important to be able to pass an external mutex lock to acquire, which is released if a task needs to wait for an event. This capability deals with the situation where a task has acquired mutual exclusion to safely look at some data using a different mutex lock but then decides it must now block for an event to occur. If the synchronization lock does not release the mutex lock, there is a race between releasing the mutex lock and blocking on the synchronization lock, e.g.:

```
mutexlock m;
synclock s;
m.acquire();            // need mutual exclusion to examine state
// examine data
if ( ... ) {            // event not occurred ?
    m.release();        // release mutex lock for event notifier
    // can be interrupted here
    s.acquire();        // race to block for event
```

```
class CondLock {

    Task *list;
  public:
    CondLock() : list( NULL ) {}

    void acquire( MutexLock &mutexlock ) {

        // add self to lock's blocked list
        mutexlock.release();
        // yield

    }
    void release() {

        if ( list != NULL ) {
            // remove task from blocked list
            // and make ready



        }
    }
};
```

(a) Unprotected

```
class CondLock {
    uOwnerLock &mlock;
    Task *list;
  public:
    CondLock( uOwnerLock &mlock ) :
        mlock( mlock ), list( NULL ) {}
    void acquire() {
        mlock.acquire();
        // add self to lock's blocked list
        mlock.release();
        // yield
        // mlock is already held on restart
    }
    void release() {
        mlock.acquire();
        if ( list != NULL ) {
            // remove task from blocked list
            // and make ready
            // do not release mlock
        } else {
            mlock.release();
        }
    }
};
```

(b) Protected

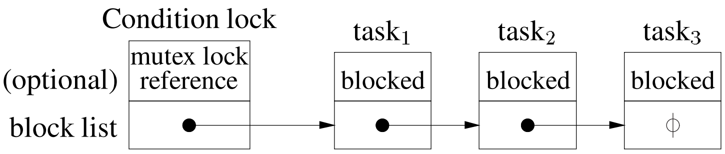


Figure 7.3: (External) Condition-Lock Implementation

If a time-slice occurs between the mutex-lock release and the synchronization-lock acquire, this task may not be on the synchronization-lock's block list when the event is signalled; hence, the cooperation fails. By having the synchronization lock release the mutex lock, e.g.:

```
s.acquire( m );
```

it is possible to ensure the blocking task is on the synchronization-lock's block list *and* the external mutex lock is released. Finally, because of the internal locking, it is possible for a release to occur without having to acquire an external mutex lock.

7.3.2.1 Condition Lock Details

The type uCondLock defines a synchronization lock:

```
class uCondLock {
  public:
    uCondLock();
    bool empty();
    void wait( uOwnerLock &lock );
    void signal();
    void broadcast();
};
uCondLock x, y, *z;
z = new uCondLock;
```

```
class SyncLock {
    spinlock mlock;             // nonblocking lock
    Task *list;                 // blocked tasks
    bool event;                 // event occurred ?
  public:
    SyncLock() : list( NULL ), event( false ) {}
    void acquire( MutexLock &userlock ) { // optional parameter
        mlock.acquire();
        if ( ! event ) {        // event not occurred ?
            // add self to task list
            userlock.release(); // release before blocking
            mlock.release();
            // yield
        } else {
            event = false;      // reset event
            mlock.release();
        }
    }
    void release() {
        mlock.acquire();
        if ( list != NULL ) {
            // remove task from blocked list and make ready
        } else {
            event = true;       // set event
            mlock.release();
        }
    }
};
```

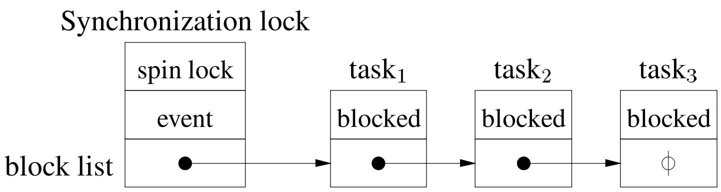


Figure 7.4: (Internal) Synchronization-Lock Implementation

(Note the name change: wait ⇒ acquire and signal ⇒ release.) The declarations create three condition locks and initialize them to closed.

The member routine empty() returns **false** if there are tasks blocked on the queue and **true** otherwise. The routines wait and signal are used to block a thread on and unblock a thread from the queue of a condition, respectively. The wait routine atomically blocks the calling task and releases the argument owner-lock; in addition, the wait routine re-acquires its argument owner-lock before returning. The signal routine checks if there is a waiting task, and if so, unblocks a waiting task from the queue of the condition lock; waiting tasks are released in FIFO order. The signal routine has no locking so mutual exclusion for its operation must be provided, usually by an owner lock. The broadcast routine is the same as the signal routine, except all waiting tasks are unblocked.

It is *not* meaningful to read or to assign to a lock variable, or copy a lock variable (e.g., pass it as a value parameter).

7.3.2.2 Synchronization

In synchronization, two or more tasks must execute a section of code in a particular order, e.g., a block of code S2 must be executed only after S1 has completed. In Figure 7.5, two tasks are created in uMain::main and each is passed a reference to an instance of uCondLock because a lock cannot be copied, i.e., passed by value. As in previous synchronization examples, the lock starts closed instead of open so if task t1 reaches S2 first, it waits until the lock is released by task t2. As well, the synchronization protocol is separated between the tasks instead of surrounding a critical section.

```
_Task T1 {
    uOwnerLock &mlk;
    uCondLock &clk;

    void main() {
        . . .
        mlk.acquire();
        clk.wait( mlk );
        S2
        . . .
    }
  public:
    T1( uOwnerLock &mlk, uCondLock &clk ) : mlk(mlk), clk(clk) {}
};
void uMain::main() {
    uOwnerLock mlk;
    uCondLock clk;
    T1 t1( mlk, clk );
    T2 t2( mlk, clk );
}
```

```
_Task T2 {
    uOwnerLock &mlk;
    uCondLock &clk;

    void main() {
        . . .
        S1
        mlk.acquire();
        clk.signal();
        mlk.release();
        . . .
    }
  public:
    T2( uOwnerLock &mlk, uCondLock &clk ) : mlk(mlk), clk(clk)
};
```

Figure 7.5: Condition Lock : Synchronization

7.3.3 Barrier

A **barrier** is used to repeatedly coordinate a group of tasks performing a concurrent operation surrounded by a series of sequential operations. Hence, a barrier is specifically for synchronization and cannot be used to build mutual exclusion. For example, three processes must execute a section of code in a particular order: S1, S2 and S3 must *all* execute before S5, S6 and S7 are executed.

```
X::main() {      Y::main() {      Z::main() {
    . . .            . . .            . . .
    S1               S2               S3
    b.block();       b.block();       b.block();
    S5               S6               S7
    . . .            . . .            . . .
}                }                }
void uMain::main() {
    uBarrier b( 3 );
    X x( b );
    Y y( b );
    Z z( b );
}
```

The barrier is initialized to control 3 tasks and passed to each task by reference (not copied). The barrier works by blocking each task at the call to block until all 3 tasks have reached their call to block on barrier b. The last task to call block detects that all the tasks have arrived at the barrier, and releases all the tasks (cooperation). Hence, all 3 tasks continue execution together after they have all arrived at the barrier. Notice, it is necessary to know in advance the total number of block operations executed before the tasks are released. Why not use termination synchronization and create new tasks for each computation? The reason is that the creation and deletion of computation tasks may be an unnecessary expense that can be eliminated by using a barrier. More importantly, it may be necessary to retain information about prior computations within the tasks.

In general, a barrier computation is composed of multiple worker tasks controlled by one or more barriers. Each barrier has the following basic structure:

1. initialization: prepare work for the computation phase
2. release: tell worker tasks (simultaneously) to perform their computation
3. computation: each worker task performs a portion of the total computation, in arbitrary order and speed

4. synchronization: the workers report completion of their computation
5. summary: compile the individual computations into a final result

Control can cycle through these steps multiple times so the worker tasks perform computations multiple times *during* their lifetime. Before a task starts its next computation, it is essential to wait for the previous computations to complete, which is why a barrier is needed. Imagine the situation where a fast task completes its second computation before a slow task has completed its first. In general, the fast task cannot begin the second computation because it is sharing resources with the slow task still performing its first computation. In this case, the tasks interfere with one another without proper synchronization.

There are several ways to construct a barrier. The obvious approach is to have a coordinator task, which performs the sequential initialization and summary work, and have worker tasks perform the concurrent computation. However, this approach results in an additional task, the coordinator, and a synchronization at both the start and end of the work, instead of just at the end [MCS91, pp. 33-34]. The reason for the additional synchronization at the start is that the coordinator must wait on the barrier along with the workers to know when the work is complete, so it can summarize and re-initialize. However, when the last worker arrives at the barrier, *all* the tasks are restarted, but only the coordinator can execute at this time to do the summary on the just completed computation, and then perform the initialization for the next computation. To prevent the worker tasks from rushing ahead and destroying the results of the previous computation and/or starting to compute again with uninitialized data, the workers must block on another barrier at the start of the computation until the coordinator also blocks on the same barrier. Only then is it known that the summary and re-initialization are complete, and all the worker tasks are ready to begin the next round of computation. Therefore, the basic structure of a worker task's main routine in this approach is:

```
for ( ;; ) {
    start.block();          // synchronize
    // compute
    end.block();            // synchronize
}
```

It is possible to eliminate both the coordinator task and the start synchronization simply by using cooperation among the worker tasks. The cooperation has the last worker synchronizing at the barrier, i.e., the task that ends the computation phase does the necessary summary work plus the initialization to begin the next computation, and finally, releases the other worker tasks. This action is done implicitly by the barrier, which knows when the last task arrives and uses that task's thread to execute some cooperation code on behalf of all the tasks. Therefore, the last task does the work, but the code to do it is part of the barrier not the task's main. Hence, the basic structure of a worker task's main routine is simplified to:

```
for ( ;; ) {
    // compute
    end.block();            // synchronize with implicit cooperation by the last task
}
```

The following discussion shows how the μC++ barrier can be used to implement this simpler and more efficient approach.

In μC++, the type uBarrier defines a barrier, and a barrier is a special kind of coroutine called a **_Cormonitor**:

```
_Cormonitor uBarrier {
  protected:
    void main() {
        for ( ;; ) {
            suspend();
        }
    }
  public:
    uBarrier( unsigned int total );
    _Nomutex unsigned int total() const;
    _Nomutex unsigned int waiters() const;
    void reset( unsigned int total );
    void block();
    virtual void last() {
        resume();
    }
};
uBarrier x(10), *y;
y = new uBarrier( 20 );
```

(The details of a **_Cormonitor** are discussed in Section 9.10, p. 282. Also, see Section 9.10.2, p. 283 for a complete implementation of uBarrier.) The declarations create two barrier variables and initialize the first to work with 10 tasks and the second to work with 20 tasks. For now, assume it is safe for a barrier coroutine to be accessed by multiple tasks, i.e., only one task is allowed in the barrier at a time.

The constructor routine uBarrier has the following form:

uBarrier(**unsigned int** total) – this form specifies the total number of tasks participating in the synchronization. Appropriate values are ≥ 0.

The member routines total and waiters return the total number of tasks participating in the synchronization and the total number of tasks currently waiting at the barrier, respectively. The member routine reset changes the total number of tasks participating in the synchronization; no tasks may be waiting in the barrier when the total is changed. block is called to synchronize with N tasks; tasks block until any N tasks have called block.

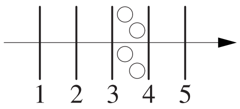
The virtual member routine last is called by the last task to synchronize at the barrier. It can be replaced by subclassing from uBarrier to provide a specific action to be executed when synchronization is complete. This capability is often used to reset a computation before releasing the tasks from the barrier to start the next computation. The default code for last is to resume the coroutine main. Like last, the coroutine main is usually replaced by subclassing to supply the code to be executed before and after tasks synchronize. The general form for a barrier main routine is:

```
void main() {
    for ( ;; ) {
        // code executed before synchronization (initialization)
        suspend();
        // code executed after synchronization (finalization)
    }
}
```

Normally, the last operation of the constructor for the subclass is a resume to the coroutine main to prime the barrier's initialization. When main suspends back to the constructor, the barrier is initialized and ready to synchronize the first set of tasks.

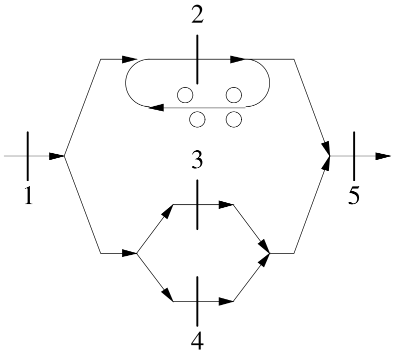
It is *not* meaningful to read or to assign to a barrier variable, or copy a barrier variable (e.g., pass it as a value parameter).

Figure 7.6(a) shows the simplest use of barriers in a pipeline of computations, segmented by barriers, where all worker tasks are in a particular segment at any time. (Each numbered vertical line in the diagram matches with a corresponding suspend in the barrier::main code.) Figure 7.6(b) shows that arbitrarily complex topologies are possible, where tasks branch down different paths or loop around a section each controlled by the barrier. Note, only one group of worker tasks can be in particular segment at a time, but multiple groups can proceed simultaneously through segments of the barrier. Hence, each group of tasks perform a series of concurrent computations requiring synchronization after each step. As in previous situations, the coroutine allows this complex series to be written in a




```
barrier::main() {
    . . .
    suspend(); // 1
    . . .
    suspend(); // 2
    . . .
    suspend(); // 3
    . . .
    suspend(); // 4
    . . .
    suspend(); // 5
    . . .
}
```

(a) Pipeline




```
barrier::main() {
    . . .
    suspend(); // 1
    . . .
    if ( . . . ) {
        . . .
        for ( . . . ) {
            . . .
            suspend(); // 2
            . . .
        }
        . . .
    } else {
        if ( . . . ) {
            . . .
            suspend(); // 3
            . . .
        } else {
            . . .
            suspend(); // 4
            . . .
        }
    }
    . . .
    suspend(); // 5
    . . .
}
```

(b) Network

Figure 7.6: Barrier Structure

form that follows directly from the algorithm, rather than flattening the code and maintaining explicit execution-state information.

Figure 7.7 shows the matrix summation example from Section 5.11, p. 130 extended to read in multiple matrices and sum the elements of each matrix. The program uses an extension of the basic uBarrier to control the synchronization of the Adder tasks for summing each matrix. The extension is accomplished by subclassing from type uBarrier, generating a new barrier type called gatekeeper, replacing the main member and adding the member eod. The constructor for gatekeeper is passed the matrix, the number of rows and columns of the matrix, and an array for the subtotals generated by the Adder tasks. The member eod is called by the Adder tasks to learn when there are no more matrices to sum, and therefore, when they can terminate. The coroutine main of gatekeeper is the control point of the barrier. It is a simple loop with a one-stage pipeline, where the initialization reads a matrix, then the computation suspends back to all Adder tasks, which sum their respective rows, finishing with the summation and printing of the row subtotals when all tasks reach the barrier.

An Adder task computes the sum of a row in the matrix. The constructor for Adder is passed a row of the matrix, the number of columns in the row, the location to place the subtotal, and a gatekeeper barrier on which to synchronize. The task main of Adder loops while the barrier indicates there is a matrix to sum, computes the sum of its row of the matrix, and blocks on the barrier. If it is the last adder to arrive at the barrier, instead of blocking, the default member last in subclass uBarrier is called implicitly, which resumes the coroutine main of the gatekeeper. In general, the gatekeeper's coroutine main restarts at the first suspend in the middle of the outer **for** loop, sums the subtotals and prints the total, loops back to read the next matrix, and if data is present, suspends back to the last worker task, which then releases the other worker tasks to start the cycle again. (Notice, the use of the μC++ labelled break to implement a multi-level exit.) If no data is present, the outer loop is terminated, the done flag is set to indicate completion of the barrier, and control suspends back to the last adder task, causing the release of the other worker tasks, which immediately terminate their task main. (Notice, running off the end of the barrier's coroutine main would be incorrect because control would transfer to the barrier's starter task not the last resumer.)

```
#include <uBarrier.h>

_Cormonitor gatekeeper : public uBarrier {
    bool done;
    unsigned int rows, cols;
    int (*matrix)[10], *subtotals;
    void main() {
        eof: for ( ;; ) {
            for ( int r = 0; r < rows; r += 1 )      {     // read matrix
                for ( int c = 0; c < cols; c += 1 ) {
                    cin >> matrix[r][c];
                  if ( cin.eof() ) break eof;              // multi-level exit
                }
            }
            suspend();                                     // back for next computation
            int total = 0;
            for ( int r = 0; r < rows; r += 1 )            // total all subtotals
                total += subtotals[r];
            cout << total << endl;
        }
        done = true;
        suspend();
    }
  public:
    gatekeeper( int (*matrix)[10], unsigned int rows, unsigned int cols, int subtotals[ ] ) :
            uBarrier(rows), done(false), rows(rows), cols(cols), matrix(matrix), subtotals(subtotals) {
        resume();                                          // prime barrier
    }
    bool eod() { return done; }
};
_Task Adder {
    int *row, size, &subtotal;
    gatekeeper &g;
    void main() {
        for ( ;; ) {
          if ( g.eod() ) break;                            // check for more work
            subtotal = 0;
            for ( int r = 0; r < size; r += 1 )            // sum row of matrix
                subtotal += row[r];
            g.block();                                     // wait for other adders to finish
        }
    }
  public:
    Adder(int row[ ], int size, int &subtotal, gatekeeper &g) : row(row), size(size), subtotal(subtotal), g(g) {}
};
void uMain::main() {
    const int rows = 10, cols = 10;
    int matrix[rows][cols], subtotals[rows], r;
    Adder *adders[rows];
    gatekeeper g( matrix, rows, cols, subtotals );

    for ( r = 0; r < rows; r += 1 )                        // create summation tasks
        adders[r] = new Adder( matrix[r], cols, subtotals[r], g );
    for ( r = 0; r < rows; r += 1 )                        // wait for termination of summation tasks
        delete adders[r];
    g.last();                                              // terminate barrier (optional)
}
```

Figure 7.7: Matrix Summation with Barrier

uMain::main declares the necessary storage for the matrix, the subtotal array, the array of pointers to the Adder tasks, and the gatekeeper. As part of the declaration of the gatekeeper, its constructor does a resume, which starts the coroutine main. The coroutine main reads the first matrix, and then suspends back to the constructor to complete the declaration. This resume in the constructor primes the first matrix for computation by the Adder tasks. The Adder tasks are then dynamically allocated, and subsequently deleted, which blocks uMain::main until all Adder tasks complete after summing all the matrices.

The only problem with this technique is that the resume in the constructor of gatekeeper precludes further inheritance. If another task inherits from gatekeeper, gatekeeper's constructor starts the coroutine main before all the constructors have completed, which in most cases is not what the programmer wants. Unfortunately, this is a general problem in C++ and can only be solved by a language extension.

7.3.4 Semaphore

Does a counting semaphore provide a partial solution by initializing the semaphore to the number of recursive invocations of f? No, because a counting semaphore allows other tasks to enter the critical section not just the one that first acquired it.

The interface for semaphores, including its name, is taken from Edsger W. Dijkstra [Dij65]. In general, a "semaphore" is a system of visual signalling by positioning movable arms or flags to encode letters and numbers [HP94]. In concurrent programming, tasks signal one another using shared variables instead of flags, as in the communication between tasks in Section 5.12, p. 131 but without the busy wait. Two kinds of semaphores are discussed: binary semaphore and general or counting semaphore; other kinds of semaphore exist []. The **binary semaphore** is like a lock with two values: open and closed; the **general semaphore** or **counting semaphore** has multiple values.

A semaphore has two parts: a counter and a list of waiting tasks; both are managed by the semaphore. In the case of the binary semaphore, the counter has a maximum value of one. After a semaphore is initialized, there are only two operations that affect the value of the semaphore: acquire (entry protocol) and release (exit protocol). The names of the acquire and release routine for a semaphore are P and V, respectively. Initially, Dijkstra stated these names have no direct meaning and are used "for historical reasons" [Dij68a, p. 345], but subsequently gave a more detailed explanation:

> P is the first letter of the Dutch word "passeren", which means "to pass"; V is the first letter of "vrijgeven", the Dutch word for "to release". Reflecting on the definitions of P and V, Dijkstra and his group observed the P might better stand for "prolagen" formed from the Dutch word "proberen" (meaning "to try") and "verlagen" (meaning "to decrease") and V for the Dutch word "verhogen" meaning "to increase". [AS83, p. 12]

The semaphore counter keeps track of the number of tasks that can make progress, i.e., do not have to wait. A list of waiting tasks forms only when the counter is 0. The list is the basis of cooperation between acquiring and releasing tasks, allowing the V operation to find a waiting task to restart. In the semaphore case, a waiting task does not spin; it moves itself directly to the blocked state when it finds the semaphore counter is 0. Therefore, each waiting task *relies* on a corresponding releasing task to wake it up; this is in contrast to a spin lock where the releaser only marks the lock open, and it is the waiting task's responsibility to notice the change. Cooperation does mean the individual cost of releasing a semaphore is greater but there is a global reduction in wasted time because of the elimination of unnecessary spinning by waiting tasks.

7.3.4.1 Semaphore Implementation

Figure 7.8 shows a possible implementation of a semaphore and the data structure created by the semaphore. Each semaphore has a counter and pointer to a linked list of waiting tasks. A task is added to the linked list in the P routine, if it cannot make progress, and made ready in the V routine. In this implementation, the semaphore counter goes negative when tasks are blocked waiting, and the absolute value of a negative counter value is the number of waiting tasks. Knowing how many tasks are waiting on a semaphore might be useful, but is not a requirement of a semaphore implementation. Nevertheless, a semaphore user cannot determine that the implementation allows the counter to go negative; a user only sees the logical behaviour of the semaphore through its interface, which is a counter going from 0-1 or 0-N, depending on the kind of semaphore. Notice, when a releasing task restarts a waiting task, it passes

```
class uSemaphore {
    int cnt;
    Tasks *list;
  public:
    void P() {                    // executed atomically
        cnt -= 1;
        if ( cnt < 0 ) {
            // task adds itself to task list and blocks,
            // which results in the next ready task starting
        }
    }
    void V() {                    // executed atomically
        cnt += 1;
        if ( cnt <= 0 ) {
            // move blocked task from the waiting list to the ready queue
        }
    }
};
```

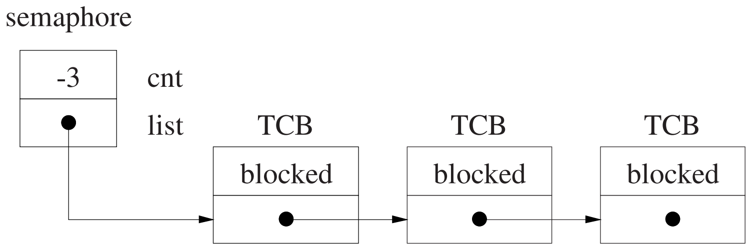


Figure 7.8: Semaphore Implementation

control of the semaphore to that waiting task. All other tasks trying to access the semaphore are either blocked waiting or are in the process of blocking. The task released by the V operation knows that it is the only task restarted, and therefore, when it eventually moves to the running state, it can exit the P routine without further checks. In essence, the cooperation ensures a task is not made ready unless the condition it is waiting for is true.

Parts of the semaphore implementation are given as comments to simplify the discussion, but it does leave several open questions. For a binary semaphore, where the counter logically only goes between 0-1, what does the implementation of the V operation do in the case where a user attempts to increment the counter above the value of 1, e.g., by executing two consecutive V operations on the same semaphore when there are no blocked tasks? (If there are N blocked tasks, it is possible to V $N+1$ times, unblocking all the tasks and leaving the counter at 1, without causing a problem.) Two possible answers are to ignore the second V and leave the counter at 1, or generate an error message and terminate the program. There is no fixed rule for handling this scenario, and a semaphore implementor may choose any appropriate action so long as the semantics of the semaphore are preserved. In general, a user should not rely on any particular semantics for this situation, and should write programs so that the situation does not occur.

When the V routine finds multiple waiting tasks, which task is scheduled next from the list? The intuitive answer is that the waiting tasks are serviced in FIFO order to ensure a bound on waiting time. However, there are other possible algorithms that still guarantee a bound but do not service in FIFO order. For example, tasks may have execution priorities, and these priorities are factored into the selection of which task executes next, while still ensuring a bound.

Another important question is how the mutual execution is achieved for the critical sections of the P and V routines? The most likely implementation is that a spin lock is used to provide mutual exclusion, as in:

```
class uSemaphore {
    . . .
    uLock lock;        // spin lock
  public:
    void P() {                      void V() {
        lock.acquire();                 lock.acquire();
        . . .                           . . .
        lock.release();                 lock.release();
    }                               }
```

(Actually, the implementation of the P routine is more complex, and this complexity is developed in subsequent discussion.) Has the busy wait been removed completely? No, it still exists for the time to modify the counter, and possibly to perform any necessary list manipulations. In most systems, these operations can be done quickly (20-100) instructions. Therefore, the busy waiting time is very small in comparison to busy waiting until a critical section becomes free or a synchronization event occurs. The short busy wait reduces the potential for a task never gaining entry to the P or V routine for a particular semaphore; i.e., the shorter the busy wait, the probabilistically smaller the chance of starvation. In general, concurrent systems all have one or more busy waits and rely on some probabilistic argument concerning starvation.

One additional implementation technique for uniprocessor systems is to *cheat* in an effort to make the implementation very fast. The cheat is to ignore rule 2: arbitrary order and speed of execution. The implementation simply sets a flag when entering special critical sections, like the P and V routines of a semaphore. If a time-slice occurs during execution of these routines, the time-slice code checks if the flag is set and returns *without* switching to another task. Not switching ensures the critical section is not interrupted, and therefore is executed atomically. However, ignoring time-slices means it is theoretically possible for one task to gain sole control of the CPU if the timer goes off only when that task is executing in special critical sections. In practice, these special critical sections are kept small enough that the probability of this situation occurring is very small. However, on a multiprocessor, spin locks must be used to protect the critical sections, because a single flag would prevent legitimate time-slices for tasks executing on other CPUs. (Make sure you understand this last point before proceeding.)

7.3.4.2 Semaphore Details

Up to this point, locks have two values, open and closed, e.g., the binary semaphore. The details of the μC++ semaphore show that it supports more than two values. However, the discussion of multi-valued locks is presented later in Section 7.3.7; for the moment, the μC++ semaphores are used as binary semaphores, i.e., only have values 0 and 1.

In μC++, the semaphore type is:

```
class uSemaphore {
  public:
    uSemaphore( unsigned int cnt = 1 );
    void P();
    void P( uSemaphore &s );
    void V( unsigned int times = 1 );
    bool empty();
};
uSemaphore x, y(0), *z;
z = new uSemaphore(4);
```

The declarations create three semaphore variables and initialize them to the value 1, 0, and 4, respectively.

The constructor routine uSemaphore has the following form:

uSemaphore(**unsigned int** cnt) – this form specifies an initialization value for the semaphore counter. Appropriate count values are ≥ 0. The default value is 1.

The member routines P and V are used to perform the counting semaphore operations. P decrements the semaphore counter if the value of the semaphore counter is greater than zero and continues; if the semaphore counter is equal to zero, the calling task blocks. The second form of P is discussed in Section 7.6.6, p. 215. V wakes up the task blocked for the longest time (FIFO service) if there are tasks blocked on the semaphore and increments the semaphore counter. If V is passed a positive integer value, the semaphore is Ved that many times. The member routine empty() returns false if there are tasks blocked on the semaphore and true otherwise.

It is *not* meaningful to read or to assign to a semaphore variable, or copy a semaphore variable (e.g., pass it as a value parameter). (See the end of Section 7.2.1, p. 179 for the reason.)

7.3.5 Synchronization

In synchronization, two or more tasks must execute a section of code in a particular order, for example, a block of code S2 must be executed only after S1 has completed (see Figure **??**, p. ??). Two test tasks are created in uMain::main and each is passed a reference to an instance of uSemaphore because a lock cannot be copied, i.e., passed by value. Again, the semaphore starts closed instead of opened so that if task X reaches S2 first, it waits until the semaphore is released by task Y. As well, the synchronization protocol is separated between the tasks instead of surrounding a critical section.

7.3.6 Mutual Exclusion

Mutual exclusion is constructed in the obvious way using a uSemaphore (see Figure **??**, p. ??). Two test tasks are created in uMain::main and each is passed a reference to an instance of uSemaphore. (Remember a semaphore cannot be copied so it cannot be passed by value.) Each task uses the semaphore to provide mutual exclusion for two critical sections in their main routine by acquiring and releasing the semaphore before and after the critical section code. When one task's thread is in a critical section, the other task's thread can execute outside the critical sections but must wait to gain entry to the critical section.

Does this solution afford maximum concurrency, that is, are tasks waiting unnecessarily to enter a critical section? Again, the answer depends on the critical sections. If the critical sections are disjoint, two semaphores are needed; if the critical sections are not disjoint, one semaphore is sufficient.

7.3.7 General or Counting Semaphore

All uses of semaphores thus far have been binary semaphores, i.e., the value of the semaphore counter is either 0 or 1. Binary semaphores are sufficient for mutual exclusion and simple synchronization. However, there is nothing that *precludes* the value of the semaphore from increasing above 1. Edsger W. Dijkstra gives credit to:

> the Dutch physicist and computer designer C. S. Scholten to have shown a considerable field of applicability for semaphores that can also take on larger values *(values greater than 1)*. [Dij65, p. 67]

As well, Dijkstra presented a weak equivalence between binary and counting semaphores [Dij65, pp. 72–76]. The difficult question is defining a meaning for a multi-valued semaphore. How can a multi-valued semaphore help with synchronization? In the case of mutual exclusion, what is a multi-valued lock?

7.3.7.1 Synchronization

A general semaphore can be used to solve more complex synchronization problems involving multiple threads of control. For example, three tasks must execute a section of code in a particular order. S2 and S3 only execute after S1 has completed.

```
X::main() {        Y::main() {        Z::main() {
    ...                ...                S1
    s.P();             s.P();             s.V();
    S2                 S3                 s.V();
    ...                ...                ...
}                  }                  }
void uMain::main() {
    uSemaphore s(0);
    X x(s);
    Y y(s);
    Z z(s);
}
```

Similar to previous synchronization examples, the semaphore starts at 0 (closed) instead of 1 (open) so if task X or Y reaches S2 or S3 first, they wait until the lock is released by task Z. The difference is that task Z Vs the semaphore twice to ensure both X and Y can make progress, which may raise the semaphore as high as two. Try to construct a scenario where the semaphore does not reach two. Notice, it does not matter if Z is interrupted between the calls to

V or which of X or Y makes progress first, the requirement that S1 be executed before S2 and S3 is still achieved. Finally, this technique can be used to solve the synchronization problem in Section 7.4:

```
COBEGIN
    BEGIN a := 1; V(L1); V(L1); END;
    . . .
    BEGIN P(L1); P(L2); c := a + b; V(L3); V(L3); END;
    . . .
COEND
```

By Ving the semaphore L1 and L3 twice, both P operations on each semaphore eventually make progress and the semaphores are both closed at the end of the COBEGIN. The drawback of this approach is the need to know in advance the total number of Ps on the semaphore so the corresponding number of Vs are performed.

7.3.7.2 Mutual Exclusion

A multi-valued lock allows more than one task to end a block of code, but restricts the total number that can enter. (The block of code is no longer a critical section because a critical section is defined to only allow a single task in it.) Going back to the bathroom example, imagine a public bathroom with multiple toilets. A general semaphore can be used to restrict the number of people that enter the bathroom to the number of toilets. The general semaphore is initialized to N, the number of toilets, and each P operation decreases the semaphore counter until it is 0, after which any additional tasks attempting entry are blocked. As each task leaves, the general semaphore is increased up to the maximum of N. The semaphore counter oscillates between 0 and N, as tasks enter and exit, without causing any task to block. Therefore, a general semaphore is used to control access to a resource with multiple instances.

If a general semaphore is used to control entry to a public bathroom, do the toilets still need locks? Clearly, the answer is yes! In fact, it is often the case that one or more binary semaphores are used in conjunction with a general semaphore. For example, a computer may have 3 identical tape drives, and allocation of the drives is controlled by a general semaphore initialized to 3. If all the drives are used, the general semaphore causes allocating tasks to block. However, it is still necessary to use a binary semaphore to ensure mutual exclusion for adding and removing the nodes that represent the tape drives from the list protected by the general semaphore. For example, three tasks might arrive simultaneously and all try to use the first tape drive.

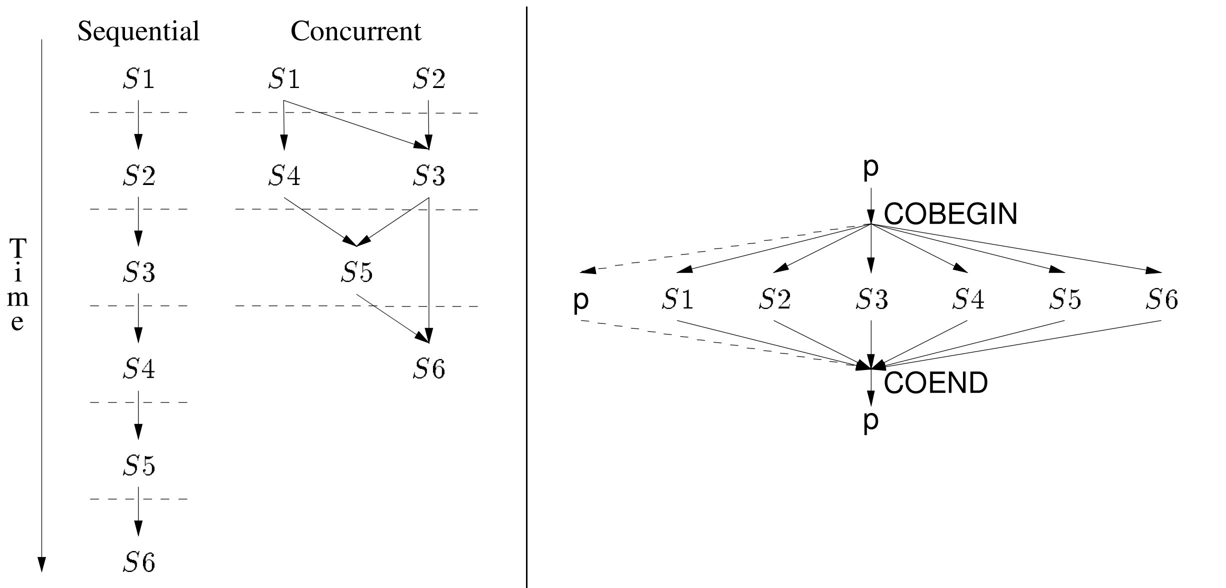
7.4 Lock and COBEGIN

As was mentioned in Section 5.8, p. 125, COBEGIN/COEND can only generate trees of tasks, while FORK/JOIN can generate an arbitrary graph of tasks. The following discussion shows that locks in conjunction with COBEGIN/COEND are strongly equivalent to FORK/JOIN. The locks used are semaphores to eliminate busy waiting, but spin locks can be used instead. The approach is straightforward, use the lock to control synchronization of the statements in the COBEGIN. A complex example is created to illustrate the approach.

Given the following series of statements:

$$\begin{aligned} S_1 &: a \leftarrow 1 \\ S_2 &: b \leftarrow 2 \\ S_3 &: c \leftarrow a + b \\ S_4 &: d \leftarrow 2 * a \\ S_5 &: e \leftarrow c + d \\ S_6 &: f \leftarrow c + e \end{aligned}$$

what is the maximum concurrency that generates the same result as sequential execution of the statements? To determine maximum concurrency requires analysing which data and code depend on each other. For example, statements S_1 and S_2 are independent; they can be executed in either order or at the same time, and not interfere with each other. However, statement S_3 depends on both S_1 and S_2 because it uses the results of the assignments to variables a and b. If statement S_3 executes before or between or during the execution of S_1 and S_2, the value of c might not be the same as for sequential execution because old values of a and b are used in the calculation of c.



(a) Precedence Graph

(b) Thread Graph

Figure 7.9: Precedence versus Thread

Dependencies among statements can be shown graphically in a **precedence graph**. A precedence graph is different from a thread graph (see Section 5.5.1, p. 120). A precedence graph shows when tasks must start to ensure dependences among data and code for correct execution; a thread graph shows when tasks start and end but no order of execution is specified to produce consistent results.

Figure 7.9(a) shows the precedence graph for the above series of statements. Relative time runs down the graph showing when statements execute. The graph on the left shows sequential execution of the statements; the graph on the right shows maximum concurrent execution. The dashed lines separate levels where statements execute concurrently because they are independent. The time between dashed lines is the time to execute all the statements at that level; if the statements are executed in parallel at each level, the total time to execute the program decreases. Look through the statements and precedence graph to see why each statement appears at its particular level in the graph.

It is impossible to achieve the concurrency indicated in the precedence graph with only COBEGIN because the join points overlap, forming a network rather than a tree structure. However, with the addition of locks, it is possible to precisely control the execution of the threads executing the statements of the COBEGIN. The statements can be written using COBEGIN and semaphores to stop and start (synchronize) execution appropriately, as in the following pseudo code:

```
VAR L1, L2, L3, L4, L5 : SEMAPHORE := 0;
COBEGIN
    BEGIN                    a := 1;      V(L1);   END;    // S1
    BEGIN                    b := 2;      V(L2);   END;    // S2
    BEGIN P(L1); P(L2); c := a + b;  V(L3);   END;    // S3
    BEGIN P(L1);             d := 2 * a;  V(L4);   END;    // S4
    BEGIN P(L3); P(L4); e := c + d;  V(L5);   END;    // S5
    BEGIN P(L3); P(L5); f := c + e;           END;    // S6
COEND
```

Figure 7.9(b) shows the process graph for the above series of statements. The approach is to create a semaphore for each statement, except the last statement, and V each statement's semaphore once the statement has completed execution. Each statement with a dependency starts with a P operation for each statement it depends on. The P operations cause a statement to "wait" until its dependent statements have completed execution, including their V operation. This approach is nothing more than a complex version of the synchronization shown in Section 7.3.5, p. 196, which states code S2 cannot be executed before code S1. Finally, the thread graph for this program is a simple tree that branches six ways and joins at the COEND, which is quite different from the precedence graph, which expresses the notion of dependency among the threads *after* they are created.

Does this solution work? In fact, it does not work. Notice that statements $S3$ and $S4$ both depend on $S1$, however there is only one V at the end of $S1$ on semaphore L1 but two Ps at the start of $S3$ and $S4$. Now remember, V opens the semaphore, and P closes the semaphore. The problem is that the first P closes L1 so the second P never finds the lock open, independent of the order of execution of the three statements. The same problem occurs with statements $S5$ and $S6$ and semaphore L3.

There are several ways to fix this problem; two are discussed here and another in the next section. One fix is to V the semaphore L1 after it is Ped so that the semaphore is open for the other thread, as in:

```
COBEGIN
    . . .
    BEGIN P(L1); V(L1); P(L2); c := a + b; V(L3); END;
    BEGIN P(L1); V(L1); d := 2 * a; V(L4); END;
    . . .
```

However, this solution leaves the semaphore L1 open after execution of the COBEGIN because the number of Vs and Ps does not balance, which may or may not be a problem, but should be noted. Another fix is to introduce another semaphore, as in:

```
VAR L11, L12, L2, L3, L4, L5 : SEMAPHORE := 0;
COBEGIN
    BEGIN a := 1; V(L11); V(L12); END;
    BEGIN b := 2; V(L2); END;
    BEGIN P(L11); P(L2); c := a + b; V(L3); END;
    BEGIN P(L12); d := 2 * a; V(L4); END;
    . . .
```

Both approaches can be used to solve the same problem for statements $S5$ and $S6$ and semaphore L3.

7.5 Producer-Consumer with Buffer

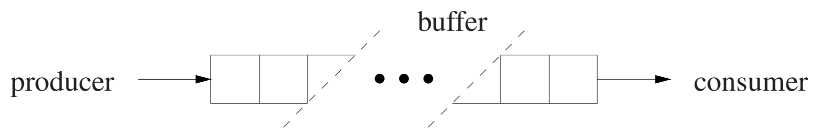
As for coroutines, it is common for one task to produce data that is consumed by another task, which is a producer-consumer problem (see Section 4.8, p. 88). Also, Section 5.12, p. 131 presented the first attempt at a producer and consumer task communicating during their lifetime (not just at task termination). In addition, the communication was accomplished through a buffer of size one (i.e., the data variable in Figure 5.2, p. 132). However, it was dismissed as too inefficient because of the busy waiting; similarly, changing the program to use spin locks would result in the same problem. This problem can be fixed by using semaphores.

Now, because tasks execute at different speed and in different order, the producer or consumer may temporarily speed up or slow down. Sometimes it is useful to smooth out differences in speed by increasing the size of the buffer used for communication. A larger buffer allows the producer task to continue producing elements even if the consumer is blocked temporarily, and the consumer task to continue consuming elements even if the producer is blocked temporarily.

In detail, the producer-consumer problem with tasks using a buffer is defined as follows. Two tasks are communicating unidirectionally through a queue. The producer puts elements at the end of a queue, and the consumer takes these elements from the front of the queue. The queue is shared between the producer and the consumer. There are restrictions placed on the producer and consumer depending on the size of the queue, which are discussed in the next two sections.

7.5.1 Unbounded Buffer

In this fictitious situation, the queue is of unbounded or infinite length:



Because the queue is infinitely long, the producer never has to wait. However, the consumer may have to wait when the buffer is empty for a producer to insert an element. As a result, service between the producer and consumer may not

```
#define QueueSize ∞

int front = 0, back = 0;
int elements[QueueSize];
uSemaphore full(0);

_Task Producer {
    void main() {
        for ( ;; ) {
            // produce an element

            // add to back of queue
            full.V();
        }
        // produce a stopping value
    }
};
_Task Consumer {
    void main() {
        for ( ;; ) {
            full.P();
            // remove element from queue
          if ( stopping value ? ) break;
            // consume element

        }
    }
};
```

(a) Unbounded Buffer

```
#define QueueSize 10

int front = 0, back = 0;
int elements[QueueSize];
uSemaphore full(0), empty(QueueSize);

_Task Producer {
    void main() {
        for ( ;; ) {
            // produce an element
            empty.P();
            // add to back of queue
            full.V();
        }
        // produce a stopping value
    }
};
_Task Consumer {
    void main() {
        for ( ;; ) {
            full.P();
            // remove element from queue
          if ( stopping value ? ) break;
            // consume element
            empty.V();
        }
    }
};
```

(b) Bounded Buffer

Figure 7.10: Producer-Consumer Simple

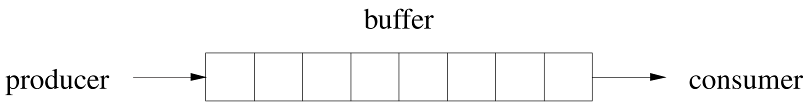
be in the order that requests are made to access the buffer, i.e., service is non-FIFO with respect to the buffer. In this situation, accommodating non-FIFO service it essential. Figure 7.10(a) shows the major components of the producer-consumer using an unbounded buffer. The shared declarations are divided into two components: communication and task management. The variables front, back and elements are an array implementation of an infinite queue, with indices to the front and back of the queue, respectively, used for communication. The variable full is a counting semaphore that indicates how many full slots are in the buffer; the semaphore is initialized to 0 indicating the buffer is initially empty. Notice the producer Vs the semaphore *after* an element is inserted into the buffer, so it counts up with each element inserted; the consumer Ps the semaphore *before* an element is removed so the semaphore counts down with each element removed. If the semaphore counter drops to 0, indicating no elements in the queue, the consumer blocks until the producer inserts an element and Vs the semaphore (non-FIFO order).

Is there a problem adding and removing elements from the shared queue? Normally, adding and removing an element from a queue must be done with mutual exclusion for the following reason. When the queue has multiple elements in it, manipulation of the front and back are often independent and do not require mutual exclusion. The problem arises when the queue is empty and the producer starts to insert an element while at the same time the consumer tries to remove it. For an array implementation of a queue, this situation may not cause a problem. However, for a linked-list implementation of a queue there are usually problems for this case. Fortunately, the semaphore full ensures that there are no simultaneous operations on the shared queue for this boundary case. The consumer cannot remove from an empty queue until the producer has completely inserted at least one element. In effect, the producer never tells the consumer something has occurred, until it is done.

Is this a synchronization problem or a mutual exclusion problem? All the signs indicate that this is a synchronization problem. Look at how the semaphore full is used. First, it is initialized to closed instead of open. Second, the V is in one task and the P is in another task instead of bracketing a block of code. Therefore, semaphore full is used for synchronization.

7.5.2 Bounded Buffer

An unbounded buffer cannot exist, but it is a simple starting point for introducing the problems associated with a concurrently accessed buffer. In this realistic situation, the queue is of bounded or finite length:



Because the queue is finite in length, the producer may have to wait for the consumer to remove an element. As for the unbounded buffer, the consumer may have to wait until elements are produced. Again, service between producer and consumer is non-FIFO depending on the state of the buffer. Figure 7.10(b) shows the major components of the producer-consumer using a bounded buffer. The only change from the unbounded buffer program is the additional counting-semaphore empty, which indicates the number of empty slots in the buffer. The producer now has to contend with the possibly that the buffer has no empty slots for it to fill. To accomplish this, semaphore empty is used in the reverse way to semaphore full; empty is initialized to the queue size, indicating that initially all buffer slots are empty, and it counts down as each empty slot is filled. Notice the producer Ps the semaphore *before* inserting an element into the buffer, so it counts down with each element inserted; the consumer Vs the semaphore *after* an element is removed so the semaphore counts up with each element removed. The sum of the semaphore counters for full and empty is always the queue size; that is, the number of full and empty slots in the buffer always equals the queue size. The use of two counting semaphores to control access to the bounded buffer is a powerful example of synchronization. The program has no **if** statements to check the status of the buffer; all checking occurs within the semaphore abstraction. However, notice that the semaphore empty starts at the queue size and not 0, which is slightly unusual for a synchronization semaphore, but consistent with a counting semaphore managing a resource with multiple instances (see Section 7.3.7.2, p. 197).

Does this solution produce maximum concurrency? In fact, it may not. Look at the code for the bounded-buffer consumer. Notice that the semaphore empty is Ved *after* the element is consumed. When the buffer is full, should the producer have to wait for the consumer to process an element before it can insert another into the buffer? The answer depends on where the element is consumed. In some situations, the consumer may not copy the element out of the buffer because the element is large, making copying expensive, or copying may not be allowed; therefore, the element must be consumed in place. Hence, the producer must wait until the buffer slot becomes free. Alternatively, the consumer may copy an element out of the buffer and consume the copy. In this case, the consumer should let the producer know that a slot is empty immediately after the element is removed (copied). To accomplish this, the statement empty.V() needs to be moved after the remove from the buffer to achieve maximum concurrency. A similar situation is possible for the producer; the producer may produce elements directly in a buffer slot or produce them elsewhere and copy them into a buffer slot when complete.

Does this solution handle multiple producers and consumers? It is reasonable to imagine multiple producers inserting elements into the buffer, and correspondingly, multiple consumers taking elements out of the buffer. If the producer is much faster than the consumer, multiple consumers may be needed to keep pace with the producer; the reverse may also be true. Does the current solution properly handle this situation? In fact, the two semaphores, full and empty, ensure *multiple* producers or consumers block if the buffer is full or empty. A semaphore does not know or care if it is decremented or incremented by a single or multiple tasks; once the semaphore is at 0, it blocks any number of tasks. However, the problem of simultaneous access to the buffer occurs again. In this case, the problem is not between operations occurring at opposite ends of the queue, but operations occurring on the same end of the queue. The semaphore empty does not prevent multiple producers from simultaneously attempting to insert elements into the same empty buffer slot; similarly, the semaphore full does not prevent multiple consumers from simultaneously attempting to remove elements from the same full buffer slot. This situation is the same at that discussed in Section 7.3.7.2, p. 197: locks are still needed on the toilet doors of a public bathroom protected with a counting semaphore. Thus, the queue itself must be protected with a binary semaphore so that insertions and removals are serialized. How many binary semaphores are needed to ensure maximum concurrency? The answer depends on the queue implementation. If insert and removal are independent, other than at the boundary case of an empty queue, then separate semaphores can protect each end of the queue to allow insertion by a producer at the same time as removal by a consumer; otherwise, a single semaphore must protect both insertion and removal.

Figure 7.11, p. 203 shows a complete producer-consumer program using a bounded buffer that supports multiple producers and consumers. The bounded buffer is generic in the type of the elements stored in the buffer. An array implementation is used for the buffer queue, with two subscripts that move around the buffer in a cycle. The buffer size is specified when the buffer is created, so individual buffer instances can be different sizes, but the size does not change after creation. There are 4 semaphores: full and empty ensure producers and consumers delay if the buffer fills or empties, respectively, and ilock and rlock ensure the queue insert and remove operations are atomic with regard to multiple producers and consumers, respectively. The buffer declares a null copy constructor and assignment operator to preclude any form of copying because its local semaphores cannot be copied. This requirement is a logical consequence of the restriction that an object cannot be copied unless all of its members can be copied.

The producer and consumer tasks communicate integer values in the range 1–100 inclusive using an instance of the buffer. uMain::main creates the buffer, and then dynamically creates an array of N producers and another array of M consumers, each element of both arrays is passed a reference to the buffer. Then uMain's thread deletes the producers, which cause it to wait for those threads to terminate (i.e., termination synchronization). After the producers have finished, enough stopping values are inserted in the buffer to cause all the consumers to terminate. Then uMain's thread deletes the consumers, which cause it to wait for those threads to terminate (i.e., termination synchronization).

7.6 Readers and Writer

The next important standard problem in concurrency is the readers and writer problem. This problem stems from the observation made in Section 6.1, p. 137 that multiple tasks can simultaneously read data without interfering with one another; only writing cause problems. This capability is possible because reading does not change the resource, and so, one reader does not affect another. However, writing tasks must have mutual exclusion to a resource before changing it. The mutual exclusion ensures that readers do not read partial results that could cause the reader to subsequently fail. As well, it ensures writers do not interfere with each other by writing over each others partial calculations.

This observation is important because most of the operations performed on shared resources are reading and only a small amount is writing. For example, most files or linked lists are read for searching reasons, rather than update reasons. Therefore, there is significant potential for increasing concurrency by differentiating between reading and writing, since it is possible to have multiple concurrent readers. As mentioned in Section 7.3.7.2, p. 197, a new kind of critical section needs to be defined: one where multiple tasks can be in it if they are only reading, but only one writer task can be in the critical section at a time.

While the statement of the readers and writer problem seems straightforward, the solution is not. Solving the readers and writer problem is like solving the N-task mutual exclusion game, with the added complexity that multiple readers can be in the critical section but only one writer. However, unlike the original mutual exclusion game, which was strictly a software solution, the readers and writer solution can use hardware locks. Still, the structure of the solution is very similar: construct some entry and exit code before and after the read and the write operation(s) that ensures the correct access to the critical section.

Like the original mutual exclusion problem, it is useful to have an analogy to explain the problems. The analogy is that readers and writers arrive at the entrance to a room in which there is a blackboard and chalk (see Figure 7.12, p. 204). The reader tasks read the data on the black board, and the writer tasks may read the data on the black board and use the chalk to write new data. The readers and writers queue outside the room, and some appropriate entry and exit protocol ensures the readers and writer mutual exclusion.

Like software solutions to mutual exclusion, there is a history of attempts at solving this problem. Probably the most well known pair of solutions using semaphores were given by Courtois et al [CHP71]; both solutions are examined (see Figure 7.13, p. 204). The general outline is as follows. A reader task calls the routine reader to perform a read and a writer task calls writer to write. The protocol must allow multiple readers but serialize writers. Normally, each routine would have the values to be read or written passed as arguments, but that is superfluous for this discussion.

The left solution in Figure 7.13, p. 204 is the simplest. It has a counter, rdcnt, to count the number of simultaneous reader tasks using a resource; a semaphore, e, to provide mutual exclusion for modifying rdcnt and examining its value; and a semaphore, rw, to block the first reader of a group of readers, as well as writers waiting entry.

The reader entry protocol acquires mutual exclusion to safely modify and check the reader counter. After acquiring mutual exclusion, the reader counter is incremented and a check is made for the first reader. The first reader of a group P's on semaphore rw, which is 1 if there is no writer, so the reader does not wait, or which is 0 if there is a writer, so the reader waits while holding the entry semaphore, e. If no writer task is using the resource, the first reader task exits the entry protocol, releasing the semaphore e, and starts to read. Any additional readers that come along acquire the entry semaphore, determine they are not the first reader, and start reading. If a writer task is using the resource,

Bounded Buffer	Producer-Consumer

```
#include <uC++.h>
#include <uSemaphore.h>

template<class ELEMTYPE> class Buffer {
    const int size;
    int front, back;
    uSemaphore full, empty;
    uSemaphore ilock, rlock;
    ELEMTYPE *Elements;

    Buffer( Buffer & ); // no copy
    Buffer &operator=( Buffer & ); // no assignment
  public:
    Buffer( const int size = 10 ) :
            size( size ), full( 0 ), empty( size ) {
        front = back = 0;
        Elements = new ELEMTYPE[size];
    }
    ~Buffer() {
        delete  Elements;
    }
    void insert( ELEMTYPE elem ) {
        empty.P();        // wait if queue is full

        ilock.P();        // serialize insertion
        Elements[back] = elem;
        back = ( back + 1 ) % size;
        ilock.V();

        full.V();         // signal a full queue slot
    }
    ELEMTYPE remove() {
        ELEMTYPE elem;

        full.P();         // wait if queue is empty

        rlock.P();        // serialize removal
        elem = Elements[front];
        front = ( front + 1 ) % size;
        rlock.V();

        empty.V();        // signal empty queue slot
        return elem;
    }
};
```

```
_Task producer {
    Buffer<int> &buf;
    void main() {
        const int NoOfElems = rand() % 20;
        int elem;

        for ( int i = 1; i <= NoOfElems; i += 1 ) {
            yield( rand() % 20 ); // pretend to produce
            elem = rand() % 100 + 1;
            buf.insert( elem );
        }
    }
  public:
    producer( Buffer<int> &buf ) : buf( buf ) {}
};
_Task consumer {
    Buffer<int> &buf;
    void main() {
        int elem;

        for ( ;; ) {
            elem = buf.remove();
          if ( elem == -1 ) break; // stopping value ?
            yield( rand() % 20 ); // pretend to consume
        }
    }
  public:
    consumer( Buffer<int> &buf ) : buf( buf ) {}
};
void uMain::main() {
    const int NoOfCons = 3, NoOfProds = 4;
    Buffer<int> buf;     // create shared buffer
    producer *prods[NoOfProds];
    consumer *cons[NoOfCons];

    // create produces and consumers
    for ( int i = 0; i < NoOfCons; i += 1 )
        cons[i] = new consumer( buf );
    for ( int i = 0; i < NoOfProds; i += 1 )
        prods[i] = new producer( buf );

    for ( int i = 0; i < NoOfProds; i += 1 )
        delete prods[i];  // wait for producers to end

    // terminate each consumer
    for ( int i = 0; i < NoOfCons; i += 1 )
        buf.insert( -1 );
    for ( int i = 0; i < NoOfCons; i += 1 )
        delete cons[i];   // wait for consumers to end
}
```

Figure 7.11: Producer-Consumer Complex

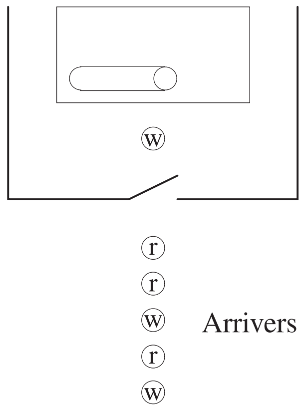


Figure 7.12: Readers and Writer

Solution 1

```
int rdcnt = 0;
uSemaphore e(1), rw(1);

void reader() {
    e.P();              // entry protocol
        rdcnt += 1;
        if ( rdcnt == 1 ) rw.P();
    e.V();

    // READ

    e.P();              // exit protocol
        rdcnt -= 1;
        if ( rdcnt == 0 ) rw.V();
    e.V();
}
void writer() {
    rw.P();             // entry protocol

    // WRITE

    rw.V();             // exit protocol
}
```

Solution 2

```
int rdcnt = 0, wrtcnt = 0;
uSemaphore e1(1), e2(1), e3(1), r(1), rw(1);

void reader() {
    e3.P();             // entry protocol
        r.P();
            e1.P();
                rdcnt += 1;
                if ( rdcnt == 1 ) rw.P();
            e1.V();
        r.V();
    e3.V();

    // READ

    e1.P();             // exit protocol
        rdcnt -= 1;
        if ( rdcnt == 0 ) rw.V();
    e1.V();
}
void writer() {
    e2.P();             // entry protocol
        wrtcnt += 1;
        if ( wrtcnt == 1 ) r.P();
    e2.V();
    rw.P();

    // WRITE

    rw.V();             // exit protocol
    e2.V();
        wrtcnt -= 1;
        if ( wrtcnt == 0 ) r.V();
    e2.V();
}
```

Figure 7.13: Courtois et al: Readers and Writer Solutions

holding the semaphore e, prevents other reader tasks from attempting entry; these readers block outside the critical section on semaphore e.

As readers finish reading, they execute the reader exit protocol. Note, readers can finish reading in any order. As for the entry protocol, the exit protocol acquires the entry semaphore to modify and examine the reader counter. After acquiring mutual exclusion, the reader counter is decremented and a check is made for the last reader. The last reader of a group, V's on semaphore rw to start any writer that might be waiting (cooperation) or to leave the semaphore with a value of 1 for the next group of reader tasks.

The writer entry protocol acquires semaphore rw, which is 1 if there are no readers, so the writer does not wait, or which is 0 if there is at least one reader, so the writer waits. After acquiring mutual exclusion, writing begins. When the writer finishes writing, the write exit protocol V's on semaphore rw to restart either the first reader of group of readers, a writer, or leave the semaphore with a value of 1 to prepare for the next group of reader tasks or a writer.

Unfortunately, this solution breaks rule 5: starvation. The problem is that an infinite stream of readers causes writers to starve, which occurs because a reader never checks in its entry protocol for a waiting writer if there are readers already reading. The next solution attempts to correct this problem.

The right solution in Figure 7.13 is very complicated. In fact, it is so complex it is not worth struggling through a detailed explanation. (Some of the complexity results from not making any assumptions about the order tasks are released after a V operation, i.e., they do not assume FIFO release of tasks.) Furthermore, this solution still allows starvation, but it is the readers that can starve if there is an infinite stream of writers. This problem occurs because a writer never checks in its entry protocol for a waiting reader if there is a writer already writing.

There is also a peculiar problem with non-FIFO order of writers, i.e., one writer can arrive at its entry protocol *before* another writer but execute *after* it. This phenomenon occurs if the first writer is pre-empted or the two writers race between the statements at the end of the writer entry protocol:

```
e2.V();
// pre-emption on uniprocessor or race on multiprocessor
rw.P();
```

The second writer can execute its entry protocol, find wrtcnt is 1, acquire semaphore rw, change the resource, and execute its exit protocol, before the first writer executes, even though the first writer executed the entry protocol *ahead* of the second! Having values written out of order is unacceptable for many problems. All these problems and additional ones are addressed in subsequent solutions.

7.6.1 Split Binary Semaphores and Baton Passing

Neither of the previous two solutions solves the readers and writer problem because both suffer from starvation. Furthermore, the second solution it too complex to understand and prove correct. To deal with the complexity problem, techniques are needed to help program complex semaphore solutions. Two techniques are introduced to provide a methodology for writing complex semaphore programs: split-binary semaphore [Dij79], developed by Edsger W. Dijkstra, and baton passing [And89], developed by Gregory R. Andrews; these techniques can be used separately or together.

A split-binary semaphore is a technique (coding convention) for collectively ensuring that a group of critical sections, possibly protected with different semaphores, execute as one critical section; i.e., only one of the critical sections executes at a time. The reason one semaphore cannot be used to protect all the critical sections is:

- Complex protocol code must be executed atomically either before continuing or blocking, e.g., managing the reader counter in the previous example requires atomicity.

- Specific tasks must wait together on separate queues, e.g., reader and writer tasks wait on different semaphores. If different kinds of tasks wait together on the same semaphore, it becomes difficult to ensure a particular kind of task restarts after the next V operation. That is, it is usually impossible to search down the list of blocked tasks for a particular kind of task and unblock it.

Thus, a **split-binary semaphore** is a collection of semaphores where at most one of the collection has the value 1. Formally, the sum of all semaphore counters used in the entry and exit code is always less than or equal to one (assuming the semaphore counters do not go negative, see Section 7.3.4.1, p. 193). Therefore, when one task is in a critical section protected by the split-binary semaphore, no other task can be in a critical section protected with the split-binary semaphore, even though different semaphores are used to protect the critical sections. The term "split" comes from the fact that the action of a single binary semaphore is divided among a group of binary semaphores. The notion of splitting can be further applied to construct a weak equivalence between binary and general semaphores. The multiple states of the general semaphore are split into binary semaphores and explicit counters [Dij80].

Figure 7.14 illustrates split-binary semaphores by transforming the producer-consumer problem from general semaphores (see Section 7.5.2, p. 201) to split-binary semaphores. The original solution has two general semaphores, full and empty, which are each replaced by *splitting* them into two binary semaphores, and an explicit counter. For example, the general semaphore full is replaced by the binary semaphores e1 and full and the counter fullcnt. Thus, there are *two* split-binary semaphores. The binary semaphores e1 and e2 control all protocol operations that require mutual exclusion; each of these semaphores start open (i.e., with the value 1). The counting nature of the general semaphore is mimicked in the protocol via the two counter variables emptycnt and fullcnt, respectively. Finally, the binary semaphores full and empty are where tasks block if they cannot proceed.

The general semaphore P operation is transformed into a critical section protected by an entry semaphore, which decrements the associated counter variable and checks if the current task should block or continue. If the counter is negative, the task blocks on the appropriate waiting semaphore, but first the entry semaphore is released so another task can enter the critical section as for the case when the task does not block. In the former case, the entry semaphore must be released *before* blocking on the appropriate waiting semaphore, otherwise the task blocks first and never releases the entry semaphore so tasks cannot enter the critical section. The general semaphore V operation is also transformed into a critical section protected by an entry semaphore, which checks for waiting tasks. If the counter is negative, a blocked task must be present, and it is made ready by Ving the appropriate (binary) waiting semaphore, and then the associated counter variable is incremented and the entry semaphore is released. Restarting a blocked task is the cooperation that removes the busy wait.

Baton passing is a technique (coding convention) for precisely controlling the cooperation between a releasing and a blocked task. Note, that there is no actual baton; the baton is a conceptual metaphor to help understand and explain the complexity of the problem, just as there is no split-binary semaphore. The rules of baton passing are:

- there is exactly one baton,
- no task makes progress unless it has the baton,
- once the baton is given up, a task cannot change variables in the entry or exit protocol.

Examples of baton passing are presented in solutions to the readers and writer problem.

Six solutions to the readers and writer problem using split-binary semaphores and baton passing are analysed. The first two solutions have the same problems as the previous solutions but illustrate the techniques of split-binary semaphores with baton passing. The next three solutions remove problems associated with the first two solutions, in particular, starvation and a new phenomenon called staleness. The last solution deals with the peculiar problem of non-FIFO order of writers.

7.6.2 Solution 1

The analogy presented at the start of Section 7.6, p. 202 is extended to handle split-binary semaphores and baton passing. In Figure 7.15, p. 208, a table is placed at the entrance to the room and a baton is placed on the table. Readers and writers initially approach the front of the table in a queue and use the baton to determine if they have to wait or can enter the room. If they have to wait, they do so in the appropriate queue on the left (readers) or right (writers) of the table.

Figure 7.16, p. 209 shows the program that implements this model. Notice, the baton never enters the room; it is used solely to manage cooperation among arriving, waiting, and leaving tasks. The program contains three semaphores, all of which compose a single split-binary semaphore, and four counters used in the protocol. The semaphores are discussed first. entry is used to provide mutual exclusion for the entry and exit protocols. The other two semaphores are used to block readers or writers that must wait to enter the room. This split-binary semaphore not only protects multiple critical sections (all the entry and exit protocols) but also the two waiting semaphores containing different kinds of tasks (readers and writers). Again, the readers and writers wait on different semaphores so it is possible to restart an appropriate kind of task. The first two counters, rdelay and wdelay, keep track of the number of delayed readers and writers that cannot enter the room, and so, are blocked on semaphores rwait and wwait, respectively. The last two counters keep track of the number of readers and writers in the room, rcnt and wcnt, respectively. wcnt never increases above one because of the serial nature of writing.

```
int front = 0, back = 0;
int elements[10];
uSemaphore full(0), empty(10);

_Task Producer {
    void main() {
        for ( ;; ) {
            // produce an element
            empty.P();
            // add to back of queue
            full.V();
        }
        // produce a stopping value
    }
};
_Task Consumer {
    void main() {
        for ( ;; ) {
            full.P();
            // remove element from queue
            empty.V();
          if ( stopping value ? ) break;
            // consume element
        }
    }
};
```

(a) General Semaphore

```
int front = 0, back = 0;
int elements[10];
uSemaphore e1(1), full(0), e2(1), empty(0);
int fullcnt = 0, emptycnt = 10;

_Task Producer {
    void main() {
        for ( ;; ) {
            // produce an element
            e2.P();      // simulate general P
            emptycnt -= 1;
            if ( emptycnt < 0 ) {
                e2.V(); empty.P();
            } else
                e2.V();

            // add to back of queue

            e1.P();      // simulate general V
            if ( fullcnt < 0 )
                full.V();
            fullcnt += 1;
            e1.V();
        }
        // produce a stopping value
    }
};
_Task Consumer {
    void main() {
        for ( ;; ) {
            e1.P();      // simulate general P
            fullcnt -= 1;
            if ( fullcnt < 0 ) {
                e1.V(); full.P();
            } else
                e1.V();

            // remove element from queue

            e2.P();      // simulate general V
            if ( emptycnt < 0 )
                empty.V();
            emptycnt += 1;
            e2.V();
          if ( stopping value ? ) break;
            // consume element
        }
    }
};
```

(b) Split Binary Semaphore

Figure 7.14: Split Binary Semaphore

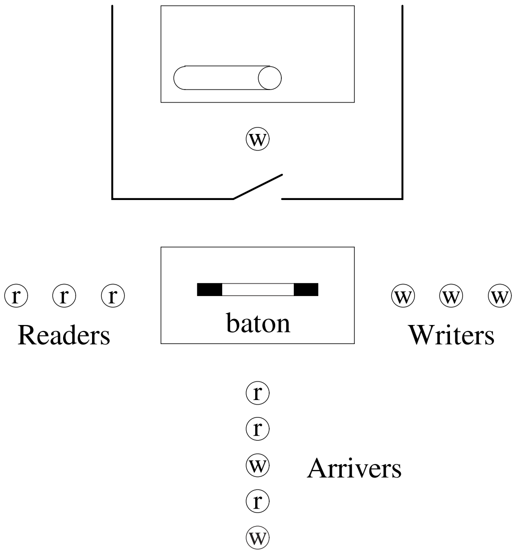


Figure 7.15: Readers and Writer

To explain the algorithm, assume that the room is currently occupied by a writer and a reader arrives. The reader conceptually picks up the baton by Ping on the entry semaphore, which decrements to zero. (Now all semaphores have a zero value.) The reader now has mutual exclusion and can examine all the protocol variables knowing no other task can read or modify them. The reader first checks if there is a writer in the room. If there is a writer, the reader increments the number of delayed readers, puts down the baton by Ving on entry and waits by Ping on rwait. A reader assumes it will be restarted eventually via cooperation when the writer, currently in the room, leaves. If there is no writer, or the reader is restarted after waiting, the reader first increments the number of concurrent readers and then checks if there are any other readers that it can read with. If the number of delayed readers in greater than zero, it is decremented and a waiting reader is Ved from semaphore rwait. As a result, a group of waiting readers take turns unblocking each other, one after the other, which is referred to as **daisy-chain unblocking**, and the readers begin reading in FIFO order. Otherwise, the baton is put back on the table for any new arrivers and the reader starts reading.

Notice, the baton is not put back on the table when the next waiting reader is V*ed.* Conceptually, the baton is passed directly from the current reader to the waiting reader. When the waiting reader wakes up, it does so holding the conceptual baton, which means it has mutual exclusion over the entry protocol when it restarts execution after rwait.P() and decrements the number of delayed readers. The notion of passing the baton forms the basis of cooperation among tasks entering and leaving the room. It ensures that when an appropriate task is woken up, that task is the only one that can legitimately continue execution with respect to the resource.

When a reader finishes reading it must pick up the baton because the exit protocol examines shared variables. Note, readers can finish reading in any order. The exiting reader can then safely decrement the number of concurrent readers, and if it is the last reader of the current group, it must perform the necessary cooperation by checking if there is a writer waiting. If the number of delayed writers is greater than zero, a waiting writer is Ved from semaphore wwait. Otherwise the baton is put back on the table for any new arrivers and the reader leaves. A reader does not check for any delayed readers because newly arriving readers immediately enter the room if there are readers already in the room. Again, notice the baton is not put back on the table if a waiting writer is Ved. The baton is conceptually passed directly from the last reader to the waiting writer. When the waiting writer wakes up, it does so holding the conceptual baton, which means it has mutual exclusion over the entry protocol when it restarts execution after wwait.P() and decrements the number of delayed writers.

Now assume that the room is currently occupied by a writer and another writer arrives. The arriving writer picks up the baton and checks if there are readers or a writer in the room. If there is a group of readers or a writer, the writer increments the number of delayed writers, puts down the baton, and waits. A writer assumes it will be restarted eventually via cooperation when the last reader of a group or a writer leaves the room. If there is neither a group of readers nor a writer, or the writer is restarted after waiting, the writer increments the number of writers (it could just

```
uSemaphore entry(1), rwait(0), wwait(0);
int rdelay = 0, wdelay = 0, rcnt = 0, wcnt = 0;

void Read() {
    entry.P();                              // entry protocol
    if ( wcnt > 0 ) {                       // resource in use ?
        rdelay += 1; entry.V(); rwait.P(); rdelay -= 1;
    }
    rcnt += 1;
    if ( rdelay > 0 )                       // more readers ?
        rwait.V();                          // pass baton
    else
        entry.V();                          // put baton down

    // READ

    entry.P();                              // exit protocol
    rcnt -= 1;
    if ( rcnt == 0 && wdelay > 0 )          // last reader ?
        wwait.V();                          // pass baton
    else
        entry.V();                          // put baton down
}
void Write() {
    entry.P();                              // entry protocol
    if ( rcnt > 0 || wcnt > 0 ) {           // resource in use ?
        wdelay += 1; entry.V(); wwait.P(); wdelay -= 1;
    }
    wcnt += 1;
    entry.V();                              // put baton down

    // WRITE

    entry.P();                              // exit protocol
    wcnt -= 1;
    if ( rdelay > 0 )                       // waiting readers ?
        rwait.V();                          // pass baton
    else if ( wdelay > 0 )                  // waiting writers ?
        wwait.V();                          // pass baton
    else
        entry.V();                          // put baton down
}
```

Figure 7.16: Readers and Writer: Solution 1

set wdelay to 1), puts down the baton and starts writing.

When a writer finishes writing it must pick up the baton because the exit protocol examines shared variables. The exiting writer can then safely decrement the number of writers, and check for any delayed readers or writers. If the number of delayed readers or writers in greater than zero, a waiting reader or writer is Ved from the appropriate semaphore. Otherwise the baton is put back on the table for any new arrivers and the writer leaves.

One interesting situation needs to be examined. Assume a reader has started the entry protocol but detected a writer in the room. The reader increments rdelay, Vs semaphore entry and then is interrupted by a time-slice *before* it can block on rwait. The writer in the room now continues execution and leaves the room. Since the reader has conceptually put down the baton, the exiting writer can pick it up and enter its exit protocol. It detects the waiting reader because the reader incremented rdelay *before* putting the baton down, and Vs semaphore rwait; however, the reader is not blocked on the semaphore, yet. Is this a problem? The answer is no because the V increments rwait to 1 so whenever the reader restarts execution it will not block but continue execution, and decrement rwait back to 0. Therefore, the order and speed of execution among the readers and writers does not affect the entry and exit protocol, which is as it should be. Essentially, the counting property of the semaphore remembers the V occurring *before* its matching P.

An interesting consequence of this situation is that the counters rdelay and wdelay are essential and cannot be replaced given additional semaphore information. For example, uSemaphore provides a member empty indicating if there are tasks blocked on the semaphore. It would appear the counters rdelay and wdelay could be eliminated and replaced with tests for non-empty reader or writer semaphores indicating delayed reader and writer tasks, respectively, i.e., a test like rdelay > 0 becomes !rwait.empty(). However, this simplification does not work because of the possibility of interruption between putting the baton down and blocking, e.g., the test !rwait.empty() returning false when in fact there is a reader task just about to wait. Hence, the delay counters not only indicate how many tasks are delayed on a semaphore but provide additional temporal information about *wanting* to block (i.e., declaring intent in Section 6.3.3, p. 143) and *actually* being blocked.

Unfortunately, there are two problems with this solution: one major and one minor. The major problem is discussed now and the minor problem is postponed to a subsequent solution. The major problem is that as long as readers keep coming along, no writer can make progress, resulting in starvation of the writers. This problem can be seen by looking at the reader's entry protocol. A reader only checks for a writer in the room; it does not check for any waiting writers, so effectively readers have higher priority than writers (as in the prioritize retract entry solution to mutual exclusion in Section 6.3.5, p. 144).

7.6.3 Solution 2

One way to alleviate, but not eliminate, the starvation problem is to switch the priority of the readers and writers so a writer has the highest priority. This approach is reasonable because on most computer systems approximately 80% of operations are reads and 20% writes; therefore, there are many more readers than writers in a normal concurrent system. Thus, the probability of a continuous stream of writers is extremely rare because there are not that many writers in a system, making starvation of readers very unlikely (i.e., a probabilistic solution).

Switching priority between readers and writers in very simple. The second line of the entry protocol for a reader is changed to the following:

Old	New
`...` `if ( wcnt > 0 ) {` `...`	`...` `if ( wcnt > 0 || wdelay > 0 ) {` `...`

A reader now checks for both a writer using the resource *and* for waiting writers, and waits appropriately. Hence, a reader defers access to the resource to any writer.

The other change starts at the third line of a writer's exit protocol. The two **if** statements are interchanged so that delayed writers are serviced ahead of delayed readers:

Old	New
`...` `if ( rdelay > 0 )` `    rwait.V();` `else if ( wdelay > 0 )` `    wwait.V();` `...`	`...` `if ( wdelay > 0 )` `    wwait.V();` `else if ( rdelay > 0 )` `    rwait.V();` `...`

Now when a writer leaves, it first checks for waiting writers and servers them before checking for waiting readers.

7.6.4 Solution 3

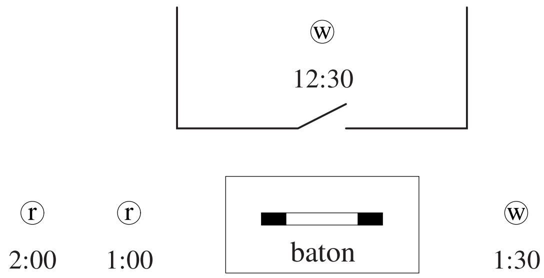
Probabilistic solutions, like the previous one, are usually unacceptable. In fact, this starvation problem can be solved using a technique from Dekker's algorithm (see Section 6.3.6.1, p. 146). Dekker's algorithm alternates between tasks for simultaneous arrivals, which can be transformed into alternating between a delayed group of readers and a delayed writer for the readers and writer problem. In other words, when there are both delayed readers and writers, alternate service to them to ensure both make progress.

The only change necessary to implement the alternation is to reverse the order of the **if** statements in the writer's exit protocol in the previous solution (i.e., put the code back to the same order as the first solution):

Old	New
. . . **if** (wdelay > 0) wwait.V(); **else if** (rdelay > 0) rwait.V(); . . .	. . . **if** (rdelay > 0) rwait.V(); **else if** (wdelay > 0) wwait.V(); . . .

Now readers check for writers first on leaving and writers check for readers first on leaving, which results in alternation when there are tasks waiting on both semaphores. Thus, the starvation problem is dealt with.

However, there is another new problem that exists with these three solutions, which was mentioned as a minor problem at the end of Section 7.6.2, p. 206. The problem is that a reader task may read information out of temporal order with writer tasks, and therefore, read information that is older then the value should be, called **staleness**. For example, a reader task arrives at 1:00 but cannot read because at 12:30 a writer arrived and is still writing. Another writer now arrives at 1:30 and then another reader arrives at 2:00; both wait for the current writer to finish, giving:



Assuming the current writer finally finishes at 2:30, and the current group of readers is restarted, i.e., the reader that arrived at 1:00 and the reader that arrived at 2:00. Unfortunately, the reader that arrived at 2:00 now reads the value written from 12:30 to 2:30, however it really wants to read the new value from the writer that arrived at 1:30, not the old value that was written starting at 12:30. The problem occurs because *all* readers in a waiting group are restarted regardless of their order of arrival with respect to waiting writers.[1] For some situations, staleness is not a problem; for other situations, staleness can result in logical inconsistencies, e.g., stock trading, or in catastrophic failure, e.g., air-traffic control.

For writer tasks, there is a similar problem with respect to readers, called **freshness**, if multiple writers are serviced before readers. (This situation happens if there is a temporary starvation scenario where writers are serviced ahead of waiting readers.) In this case, reader tasks always read the freshest data, but may miss reading older, possibly important, values written over by other writers. For some situations, freshness is not a problem; for other situations, freshness can result in problems, e.g., if a reader task is waiting for a particular data value, it may never see it because it is overwritten.

While solutions 1 and 3 suffer from staleness, solution 2 suffers from freshness. This situation occurs because writers have higher priority in solution 2 so many writes may occur while readers are waiting. Therefore, when readers eventually restart, they miss reading older, possibly important, values that have been written over by other writers.

7.6.5 Solutions 4 and 5

The problem with reading data out of order in the previous solutions comes from separating the waiting readers and writers into two queues. For example, at one time banks used a separate line of waiting customers for each bank teller. The problem is that you can arrive ahead of someone else but receive service *after* them because your line is delayed by a long transaction between a customer and the teller. In effect, you are receiving "stale" service (but there is a bound on service: the number of people ahead of you in your line). Stale service can result in real problems not just angry customers. Imagine writing a cheque without sufficient funds to cover the cheque. You rush to the bank to

[1]This situation is analogous to entering a bank with multiple service queues and tellers, and receiving service after people that arrived later than you, i.e., stale service. The reason for this anomaly is that you selected the queue for a teller that is slow or who must service a client with a large request. This situation is normally solved by using a single service queue with multiple tellers.

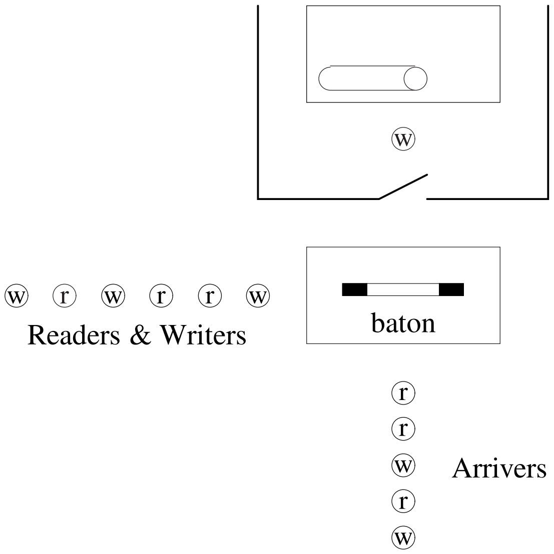


Figure 7.17: Readers and Writer

deposit enough money to cover the cheque, but the cheque casher also appears at the bank to cash the cheque. This situation is not a problem because you arrived ahead of them; however, you receive service *after* the cheque casher, and as a result, the cheque bounces (i.e., is not cashed due to insufficient funds). Most banks now use a single line; when a teller becomes free, the person at the head of the line is serviced next. This approach precludes stale service because the single line provides a relative temporal ordering of customers. Notice it is sufficient to know that everyone ahead of you arrived before you and everyone after you arrived after you; it is unnecessary to know the absolute time of arrive of any customer for the scheme to work, and therefore, a clock is unnecessary.

To mimic this solution in the readers and writer program both readers and writers must block on the same semaphore, which ensures a relative temporal ordering (see Figure 7.17). Unfortunately, information is lost when readers and writers wait on the same semaphore: it is no longer possible to distinguish whether a waiting task is a reader or writer task. The loss of this information means a reader cannot tell if the next waiting task is a reader or a writer so it does not know if it should wake up the next task. If the reader wakes up a writer, there is no way that the writer can wait again at the *front* of the semaphore because semaphores are usually implemented with FIFO waiting. This point was made with regard to split-binary semaphores in Section 7.6.1, p. 205. Two possible solutions are discussed to recover the lost information. Both solutions use a single semaphore to block both readers and writers to maintain relative order of arrival, and hence, prevent stale readers, but each solution uses a different technique to deal with the lack of knowledge about the kind of blocked tasks.

In the first approach (see Figure 7.18), a reader assumes the next task on the waiting semaphore is a reader and wakes it up. Because of the potential for inadvertently waking a writer, each writer must recheck if it can use the resource. If it cannot use the resource because there are readers still using it, the writer waits again on a special semaphore that has at most one writer task waiting on it. The special semaphore is given the highest priority whenever the baton is passed so a writer that was inadvertently restarted is always serviced next. Notice, this approach is not busy waiting because there is a bound of at most two checks and blocks for any writer. While viable, this solution requires additional complexity to manage the special semaphore, and additional execution time to block and unblock a writer task if it is inadvertently awakened.

The next solution (see Figure 7.19, p. 214) is the same as solution 3, but gets back the lost information when both readers and writers wait on the same semaphore queue by maintaining an additional queue, which is always the same length as the semaphore queue, and each node of the additional queue indicates if the corresponding task blocked on the semaphore is a reader or writer. Maintaining the additional queue can be seen in the program in both locations where tasks block. A node is created and initialized to the particular kind of task, and then it is put on the back of the additional queue before the baton is put down and the task waits on the semaphore. When a waiting task is restarted, it immediately takes its node off the front of the additional queue. Notice that each node is not dynamically allocated;

```
uSemaphore entry(1), rwwait(0), wwait(0);
int rwdelay = 0, wdelay = 0, rcnt = 0, wcnt = 0;

void Read() {
    entry.P();                          // entry protocol
    if ( wcnt > 0 || wdelay > 0 || rwdelay > 0 ) { // resource in use ?
        rwdelay += 1; entry.V(); rwwait.P(); rwdelay -= 1;
    }
    rcnt += 1;
    if ( rwdelay > 0 )                  // more readers ?
        rwwait.V();                     // pass baton
    else
        entry.V();                      // put baton down

    // READ

    entry.P();                          // exit protocol
    rcnt -= 1;
    if ( rcnt == 0 ) {                  // last reader ?
        if ( wdelay != 0 )              // writer waiting ?
            wwait.V();                  // pass baton
        else if ( rwdelay > 0 )         // anyone waiting ?
            rwwait.V();                 // pass baton
        else
            entry.V();                  // put baton down
    } else
        entry.V();                      // put baton down
}
void Write() {
    entry.P();                          // entry protocol
    if ( rcnt > 0 || wcnt > 0 ) {       // resource in use ?
        rwdelay += 1; entry.V(); rwwait.P(); rwdelay -= 1;
        if ( rcnt > 0 ) {               // wait once more ?
            wdelay += 1; entry.V(); wwait.P(); wdelay -= 1;
        }
    }
    wcnt += 1;
    entry.V();                          // put baton down

    // WRITE

    entry.P();                          // exit protocol
    wcnt -= 1;
    if ( rwdelay > 0 )                  // anyone waiting ?
        rwwait.V();                     // pass baton
    else
        entry.V();                      // put baton down
}
```

Figure 7.18: Readers and Writer: Solution 4

```
uSemaphore entry(1), rwwait(0);
int rwdelay = 0, rcnt = 0, wcnt = 0;

enum RW { READER, WRITER };          // kinds of tasks
struct RWnode : public uColable {
    RW rw;                           // kind of task
    RWnode( RW rw ) : rw(rw) {}
};
uQueue<RWnode> rwid;                 // queue of RWnodes

void Read() {
    entry.P();                       // entry protocol
    if ( wcnt > 0 || rwdelay > 0 ) { // resource in use ?
        RWnode r( READER );
        rwid.add( &r );              // remember kind of task
        rwdelay += 1; entry.V(); rwwait.P(); rwdelay -= 1;
        rwid.drop();
    }
    rcnt += 1;
    if ( rwdelay > 0 && rwid.head()->rw == READER ) // more readers ?
        rwwait.V();                  // pass baton
    else
        entry.V();                   // put baton down

    // READ

    entry.P();                       // exit protocol
    rcnt -= 1;
    if ( rcnt == 0 && rwdelay > 0 )  // last reader ?
        rwwait.V();                  // pass baton
    else
        entry.V();                   // put baton down
}
void Write() {
    entry.P();                       // entry protocol
    if ( rcnt > 0 || wcnt > 0 ) {    // resource in use ?
        RWnode w( WRITER );
        rwid.add( &w );              // remember kind of task
        rwdelay += 1; entry.V(); rwwait.P(); rwdelay -= 1;
        rwid.drop();
    }
    wcnt += 1;
    entry.V();                       // put baton down

    // WRITE

    entry.P();                       // exit protocol
    wcnt -= 1;
    if ( rwdelay > 0 )               // anyone waiting ?
        rwwait.V();                  // pass baton
    else
        entry.V();                   // put baton down
}
```

Figure 7.19: Readers and Writer: Solution 5

it is automatically allocated at the top of the task's stack before blocking, which is very efficient. The reader entry protocol can now determine if the task blocked at the front of the semaphore queue is a reader or writer by examining the value in the node at the front of the additional queue. Note, it is necessary to check for the existence of a delayed task, i.e., non-empty queue, before checking the node at the head of the queue or an invalid pointer dereference occurs. Clearly, this solution only works if the semaphore uses FIFO scheduling of blocked tasks; if the semaphores used some other scheduling scheme for selecting tasks from the blocked list (while still maintaining a bound), it would have to be mimicked precisely otherwise the two lists would get out of synchronization.

Finally, a variation on this solution is for the writer to awaken the next group of readers to use the resource when it exits the resource rather than have the readers awaken each other, which is referred to as **multiple unblocking**. The cooperation for starting multiple readers is moved from the reader's entry protocol:

Old	New
... **if** (rwdelay > 0 && rwid.head()->rw == READER) rwwait.V(); **else** entry.V(); ...	... entry.V(); ...

to the writer's exit protocol:

Old	New
... **if** (rwdelay > 0) rwwait.V(); **else** entry.V(); ...	... **if** (rwdelay > 0) **while** (rwdelay > 0 && rwid.head()->rw == READER) rwwait.V(); **else** entry.V(); ...

While the total work done by each variation is the same, the difference is which task is doing the work. It would appear the writer task now does more than its share of the cooperation work, making the latter variation less fair. However, the issue of fairness is always difficult to judge. Further discussion on this issue appears at the end of Section 9.7.2, p. 277.

7.6.6 Solution 6

Unfortunately, solution 5 does not work. The problem was mentioned earlier in the discussion of the Courtois et al solution; that is, the peculiar problem with non-FIFO order of writers, i.e., one writer can arrive at its entry protocol before another writer but execute *after* it. As a result, writers may execute in a very bizarre sequence, which makes reasoning about the values read by the reader tasks impossible. In all the solutions so far, both the reader and writer entry protocols have a code sequence like the following:

```
entry.V(); Xwait.P();
```

Assume a writer picks up the baton and there are readers currently using the shared resource, so it puts the baton down, entry.V(), but is time-sliced before it can wait, Xwait.P(). Another writer does the same thing, and this can occur to any depth. Now the writers may not restart execution in the same order that they were interrupted, even in a system that places time-sliced tasks at the end of a FIFO ready queue because a double time-slice of the same task can occur before it eventually waits, which means writer tasks may not execute in FIFO order of arrival. This case is *also* a problem for the readers because, even after all the work done in the previous solutions, readers may still read stale information. Assume a reader picks up the baton and there is a writer currently using the shared resource, so it puts the baton down, entry.V(), but is time-sliced before it can wait, Xwait.P(). A writer can then arrive and block on the waiting semaphore ahead of the reader even though the writer arrived at the entry protocol after the reader. This behaviour causes the failure of solution 5 because it is impossible to keep the explicit list of task-kinds, i.e., reader or writer, consistent with the implicit list of blocked tasks on the semaphore rwwait. After a node is added to the explicit list, there is no guarantee the task adding the node is the next to block on the semaphore.

The failure to maintain a temporal ordering of arriving reader and writer tasks is significant. As mentioned, incrementing the rwdelay counter before releasing the protocol semaphore ensures an exiting reader or writer never misses waking up a waiting task. However, the rwdelay counter *does not* control the order tasks wait on Xwait, only that *some* waiting task is awakened. In fact, it is impossible to ensure waiting order with only two semaphores, like entry and Xwait. The proof is straightforward. Semaphore entry must be released (Ved) before waiting on Xwait; reversing the order of these operations does block the waiting task but leaves the entry and exit protocol locked so new tasks cannot enter the system nor can tasks leave the system. Because entry must be released before waiting on Xwait, and there is no control over order or speed of execution, it is always possible for a blocking task to be prevented from blocking for an arbitrary time period, allowing other tasks to barge ahead of it on the waiting queue.

What is needed is a way to atomically block and release the entry lock. The μC++ type uSemaphore provides this capability through an extended P member (see Section 7.3.4.2, p. 195):

```
Xwait.P( entry );
```

which atomically Vs the argument semaphore, entry, and then blocks the calling task if necessary. The implementation of the extended P member is simply:

```
void uSemaphore::P( uSemaphore &s ) { // executed atomically
    s.V();
    P();
}
```

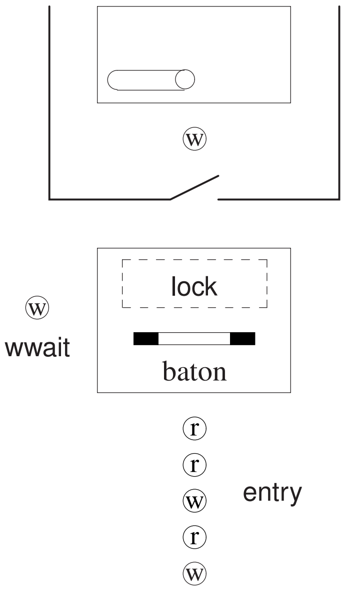
Notice that the task executing the extended P can still be interrupted between the V and P operations. However, any new task that acquires the entry semaphore and rushes to block on the waiting semaphore cannot acquire the waiting semaphore until after the task using it has blocked. By replacing all sequences of the form entry.V(); Xwait.P(); in Figure 7.18, p. 213 with the form Xwait.P(entry), the problem of stale readers and writers is finally eliminated. Furthermore, it is possible to remove the delay counters and replace them with tests of non-empty semaphores queues (see the discussion at the end of Section 7.6.2, p. 206). Unfortunately, most semaphores do not provide such an extended P operation; therefore, it is important to find a solution using only the traditional semaphore operations.

One possible, but very inefficient, solution is to have the reader and writer tasks take a ticket before putting the baton down (like the hardware solution in Section 6.4.3, p. 163). To pass the baton, a serving counter is first incremented and then all tasks blocked on the semaphore are woken up. Each task then compares its ticket value with the serving value, and proceeds if the values are equal and blocks otherwise. Starvation is not an issue because the waiting queue is a bounded length. This approach uses the fact that a task establishes its FIFO position as soon as it takes a ticket, which is independent of when the task tries to block on the waiting semaphore; even if a task is time-sliced multiple times before it can block on the waiting semaphore, no other tasks can make progress because their ticket value is not being served. Unfortunately, the cost of waking up all waiting tasks for each baton pass is prohibitively expensive. In fact, this is largely the solution that people use with tickets and a server in a store. When the server increments the server counter, everyone checks their ticket, but only the person with the matching ticket goes to the counter for service. Notice, each person must be vigilant, otherwise they will be skipped over, and therefore, it is difficult to do other work, such as reading a book, while waiting. (How could cooperation be used to allow waiting people to read their books without missing their turn?)

An efficient solution is possible by introducing a **private semaphore** for each task instead of having one semaphore on which the readers and writers wait (see Figure 7.20). (This approach is similar to the private busy waiting flag in the MCS hardware solution in Section 6.4.4, p. 165.) Initially, this seems wrong because stale readers occurred in previous solutions using separate waiting semaphores for the readers and writers. To ensure FIFO service, the queue indicating the kind of waiting task, i.e., reader or writer, is used like the ticket in the previous solution but with cooperation. Before putting the baton down, a reader or writer creates a node containing the kind of task and a private semaphore on which it waits, and puts the node on the tail of the queue. The task then attempts to block on its private semaphore. To pass the baton, the private semaphore at the head of the queue is Ved, if present. If the task associated with the private semaphore is blocked, it is woken up. If the task is not blocked yet because of a time-slice, the V is remembered by the semaphore and the task does not need to block when it eventually executes the P. Notice that it no longer matters if a task is time-sliced (one or more times) between putting the baton down and waiting. Once a task puts its node on the list it establishes its FIFO position in the queue (like taking a ticket), which is independent of when the task tries to block on its private semaphore, and only one task attempts to block on each private semaphore so it is immaterial whether the semaphore implementation unblocks task in FIFO or non-FIFO order. The cooperation comes from the fact that the exiting task (server) knows who has the next ticket (i.e., the task at the front of the queue) and can wake

```
uSemaphore entry(1);
int rcnt = 0, wcnt = 0;

enum RW { READER, WRITER };          // kinds of tasks
struct RWnode : public uColable {
    RW rw;                           // kind of task
    uSemaphore sem;                  // private semaphore
    RWnode( RW rw ) : rw(rw), sem(0) {}
};
uQueue<RWnode> rwid;                 // queue of RWnodes

void Read() {
    entry.P();                       // entry protocol
    if ( wcnt > 0 || ! rwid.empty() ) {  // resource in use ?
        RWnode r( READER );
        rwid.add( &r );              // remember kind of task
        entry.V(); r.sem.P();
        rwid.drop();
    }
    rcnt += 1;
    if ( ! rwid.empty() && rwid.head()->rw == READER ) // more readers ?
        rwid.head()->sem.V();        // pass baton
    else
        entry.V();                   // put baton down

    // READ

    entry.P();                       // exit protocol
    rcnt -= 1;
    if ( rcnt == 0 && ! rwid.empty() )   // last reader ?
        rwid.head()->sem.V();        // pass baton
    else
        entry.V();                   // put baton down
}
void Write() {
    entry.P();                       // entry protocol
    if ( rcnt > 0 || wcnt > 0 ) {    // resource in use ?
        RWnode w( WRITER );
        rwid.add( &w );              // remember kind of task
        entry.V(); w.sem.P();
        rwid.drop();
    }
    wcnt += 1;
    entry.V();                       // put baton down

    // WRITE

    entry.P();                       // exit protocol
    wcnt -= 1;
    if ( ! rwid.empty() )            // anyone waiting ?
        rwid.head()->sem.V();        // pass baton
    else
        entry.V();                   // put baton down
}
```

Figure 7.20: Readers and Writer: Solution 6




```
uSemaphore entry(1);
uSemaphore lock(1), wwait(0);
int rcnt = 0, wcnt = 0;

void Read() {
    entry.P();                      // entry protocol
    lock.P();
    rcnt += 1;
    lock.V();
    entry.V();                      // put baton down

    // READ

    lock.P();
    rcnt -= 1;
    if ( rcnt == 0 && wcnt == 1 ) { // last reader and writer waiting ?
        lock.V();
        wwait.V();                  // pass baton
    } else
        lock.V();
}
void Write() {
    entry.P();                      // entry protocol
    lock.P();
    if ( rcnt > 0 ) {               // readers waiting ?
        wcnt += 1;
        lock.V();
        wwait.P();                  // wait for readers to finish
        wcnt -= 1;
    } else
        lock.V();

    // WRITE

    entry.V();                      // exit protocol
}
```

Figure 7.21: Readers and Writer: Solution 7

them directly.

Clearly, the cost of creating and deleting the private semaphore is greater than having an extended P that atomically blocks and releases another semaphore. Subsequent discussion comes back to this special form of P to generate an efficient solution. Nevertheless, it is possible to construct a correct readers and writer solution using only a traditional semaphore.

7.6.7 Solution 7

Because there is a fundamental problem in baton passing when the baton is released and a task attempts to block, is there an alternative approach programmatic solution?

ad hoc, not baton passing

bench and chair

nested locking

turn stile ?

7.7 Summary

The spin lock and semaphore are an abstraction for some hardware-level mutual-exclusion facility. The difference between a spin lock and a binary semaphore occurs when a task cannot acquire the lock: the task's state remains ready or active for a spin lock but becomes blocked for a semaphore. Many different kinds of semaphores can and have been built, each providing more functionality to support complex synchronization and mutual exclusion, e.g., the counting semaphore and owner lock. Regardless of the sophistication of a semaphore, they all begin to reach their complexity limit fairly quickly. Using a counting semaphore produces an efficient solution for communication among tasks through a buffer, but still not a particularly elegant one because of all the globally shared information. The semaphore solutions for the readers and writer problem are complex enough that reasoning about their correctness is difficult. Also, some of these solutions assume that tasks blocked on a semaphore are released in FIFO order, which may not be true in all concurrent systems. Therefore, it is necessary to examine other approaches that can simplify some of the complexity. However, there are cases where efficiency is paramount so that spin locks and semaphores might provide the best solutions, but in general, this is infrequent.

7.8 Questions

1. The following is a collection of assignments:

$$\begin{aligned} S_1 &: B \leftarrow A \\ S_2 &: C \leftarrow B \\ S_3 &: A \leftarrow A+1 \\ S_4 &: D \leftarrow C+1 \\ S_5 &: E \leftarrow B+D \\ S_6 &: F \leftarrow D+A \\ S_7 &: G \leftarrow E+F \end{aligned}$$

   Thus the environment consists of seven variables. Initially, the environment is in the following state:

$$A = 1, B = 2, C = 3, D = 4, E = 5, F = 6, G = 7$$

   Throughout the following questions, assume that each of the statements is executed atomically; that is, once a statement begins executing, it finishes without interference from other statements.

   - If statements 1 through 7 above are executed sequentially, what is the final state of the environment?
   - Draw a precedence graph for the statements, showing the maximum amount of parallelism that can be obtained in executing these statements while obtaining the same results as sequential execution.
   - Suppose that the assignments are grouped into two processes: one process consists of statements 1 through 6, in that order, while the other consists of statement 7. The two processes execute concurrently (but remember that each statement executes atomically). What are the possible final states of the environment?

2. Assume a concurrency system that has only a binary semaphore, `bSem`, with member routines `P` and `V`, respectively. Use the binary semaphore to build a counting semaphore, `cSem`, with member routines `P` and `V`, respectively. Efficiency is not an issue, i.e., busy waiting is allowed.

3. A PRAM (pronounced "pee-ram") is one mathematical model that is used to design algorithms for computers that use massive parallelism. Massively parallel computers have not just dozens of processors, but thousands. The difficulty is to figure out how these thousands of processors can most efficiently be used to solve problems. In this question, you are to simulate a PRAM running an algorithm for finding the maximum of n distinct integers.

   A good (fast) PRAM algorithm for finding the maximum of n distinct integers x_1, x_2, ..., x_n is as follows. Initialize an array of flags `NotMax[]` with n entries to FALSE. Assume the PRAM has enough processors that it can assign one to each distinct pair of integers. Let $P(i, j)$ be the processor assigned to the pair (x_i, x_j) where $1 \leq i < j \leq n$. $P(i, j)$ compares x_i to x_j. After this comparison, $P(i, j)$ knows which one of the two could not be the maximum, for example, if $x_j < x_i$ then x_j could not be the maximum. After determining which of the two could not be the maximum, $P(i, j)$ writes a TRUE to the NotMax flag for that integer, for example, if $x_j < x_i$ then set NotMax[j] to TRUE.

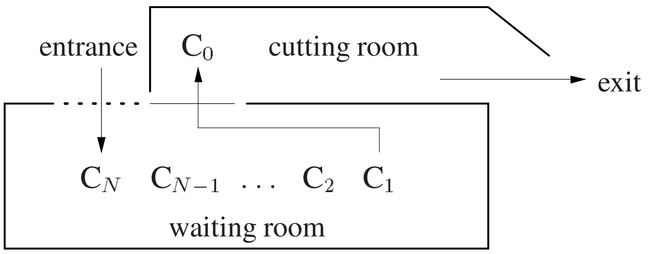
After each processor has done this, all but one of the NotMax[ ] flags will be TRUE. The one that is FALSE tells which integer is the maximum. To find which flag is FALSE, assign one processor to each flag in NotMax[ ]. Exactly one of these processors will find that its flag is FALSE, and that processor can write the maximum integer into some chosen location.

You are to implement this algorithm as follows:

- Input a list of numbers x_1 to x_n from standard input. The list is terminated with a -1 value. Assume that n is not known in advance.
- Create the PRAM. This means that you generate as many processes as you will need to run the algorithm. Do not allow any process to start its PRAM algorithm until all processes are created.
- Run the PRAM algorithm to find the maximum. Let each process die after completing its algorithm.
- Return to Step 1 again, until the end of file is reached.

Things to Note: Use as much parallelism as possible! For each set of integers, create an appropriate number of processes. Do not create any more than you actually use. Create the processes in an efficient way. Do not let any process start its PRAM algorithm until all needed processes are created, i.e., have a synchronized start. A correct program will end up having at least two synchronization points (three if you are smart). Handling the synchronization is the interesting part of this question, the rest is fairly easy. If every process writing to a shared variable is trying to write the same thing, then you do not have to worry about mutual exclusion among the writers. You may use only semaphores to solve this question.

4. The sleeping barber is a problem proposed by E. W. Dijkstra. A barber shop has a cutting room with one chair and a waiting room with N chairs.



Customers enter the waiting room one at a time if space is available; otherwise they go to another shop (called balking). The waiting room has an entry, and next to it is the entry into the cutting room with the barber's chair; the entry and exit share the same sliding door, which always closes one of them. The sliding door ensures that when the barber opens the door to see if anyone is waiting, a new customer cannot simultaneously enter the room and not be seen.

When the barber has finished a haircut, he opens the door to the waiting room and checks for waiting customers. If the waiting room is not empty, he invites the next customer (FIFO service order) for a haircut, otherwise he goes to sleep in the barber's chair. Customers may enter the waiting room at any time. If a customer finds the waiting room is not empty, they wait their turn in the waiting room; if the waiting room is empty, the customer opens the door to the cutting room to see if the barber is there. If the barber is not there, the customer sits down in the waiting room and waits for the barber to appear; otherwise, the customer wakes the barber.

The interface for the barber shop is (you may add only a public destructor and private members):

```
class BarberShop {
  public:
    BarberShop( Printer &p, const int MaxWaitingCust );
    bool enterShop( int id );                    // called by customer
    int startCut();                              // called by barber
    void endCut();                               // called by barber
};
```

A customer calls the enterShop member to enter the barber shop. This member returns true if the customer gets a haircut, and false if the waiting room is full. The barber calls the startCut member to obtain the next customer; if there are no customers, the barber goes to sleep (blocks). startCut returns the customer identifier for the longest waiting customer. The barber calls endCut after completing the haircut.

The interface for the barber is (you may add only a public destructor and private members):

```
_Task Barber {
    void main();
  public:
    Barber( Printer &p, BarberShop &bs, int CutDelay );
};
```

The barber begins by randomly yielding between 0-CutDelay times so it may not arrive before the first customers start entering the barber shop. (Yielding is performed by calling task member yield.) After indicating arrival at the shop, the barber begins the following steps:

a) indicate about to get a customer
b) call startCut in the barber shop to get customer
c) indicate sleeping if no customer waiting
d) indicate a customer is no longer in the waiting room
e) indicate starting to cut hair
f) randomly yield between 0-CutDelay times to simulate haircut
g) call endCut in the barber shop to indicate ending of haircut

The barber terminates when he receives a customer identifier of -1 from startCut.

The interface for a customer is (you may add only a public destructor and private members):

```
_Task Customer {
    void main();
  public:
    Customer( Printer &p, BarberShop &bs, int id, int CustDelay );
};
```

A customer begins by randomly yielding between 0-CustDelay times so the customers arrive at the barber shop at random times. A customer makes a single call to enterShop in the barber shop to get a haircut. After the call, the customer indicates if it fails to receive a haircut because the waiting room is full.

All output from the program is generated by calls to a (special) printer coroutine, excluding error messages. The interface for the printer is (you may add only a public destructor and private members):

```
_Cormonitor Printer {
    void main();
  public:
    Printer( const int MaxWaitingCust );
    void Barber( char status );                  // called by barber
    void CustomerCut();                          // called by barber
    void Customer( int id );                     // called by customer
};
```

(You do not need to know the details of a **_Cormonitor**; treat it solely as a coroutine with the magic property that only task can execute it at a time. Note, the coroutine main of a **_Cormonitor** *must* be a **private** memberThe printer attempts to reduce output by condensing multiple barber state-changes onto a single line. Figure 7.22 shows two example outputs from different runs of the program. The output consists of four columns:

(a) barber states

The barber has the following states:

A arrival at the barber shop

G attempting to get a customer from the waiting room

S going to sleep in the barber chair waiting for a customer

C cutting a customer's hair

F leaving the barber shop

(b) customer having haircut

(c) customer balked because waiting room is full

(d) list of customers in the waiting room

All printing must occur in the coroutine main, i.e., no printing can occur in the coroutine's member routines. All output spacing can be accomplished using the standard 8-space tabbing, so it is unnecessary to build and store strings of text for output. Calls to the printer coroutine to perform printing may be performed in the barber-shop members or in the barber/customer tasks (you decide where to print).

uMain::main creates the printer, barber shop, barber and $NoOfCustomers$ customer tasks. After all the customer tasks have terminated, uMain::main calls enterShop with a customer identifier of -1.

The shell interface to the sleepingbarber program is as follows:

sleepingbarber [$MaxWaitingCust$ [$NoOfCustomers$ [$CutDelay$ [$CustDelay$]]]]

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) Where the meaning of each parameter is:

$MaxWaitingCust$**:** maximum number of customers that can wait in the barber shop waiting room, i.e., $MaxWaitingCust$ equals N chairs. The default value if unspecified is 5.

$NoOfCustomers$**:** number of customers attempting to obtain a haircut during a run of the program. The default value if unspecified is 20.

$CutDelay$**:** number of times the barber yields to simulate the time required to cut a customer's hair. The default value if unspecified is 2.

$CustDelay$**:** number of times a customer yields *before* attempting entry to the barber shop. The default value if unspecified is 40.

Assume all arguments are valid positive integer values, i.e., no error checking is required.

5. This is a modified version of the Cigarette Smokers Problem[2] [Patil71]. There are N smokers; each smoker repeatedly makes a cigarette and smokes it. To smoke a cigarette, 3 ingredients are needed: paper, tobacco and a match. Interestingly, smokers are divided into three kinds:

   a) one has an infinite supply of papers and matches,
   b) another an infinite supply of tobacco and matches,
   c) and finally an infinite supply of tobacco and papers.

   On a table accessible to all the smokers is a tobacco pouch with an infinite amount of tobacco, a package of cigarette papers with an infinite number of papers, and a box of matches with an infinite number of matches. A smoker goes to the table and chooses the ingredient needed to smoke a cigarette (depending on which kind of smoker it is), takes the ingredient off the table, uses it to make and light a cigarette, puts the ingredient back on the table, and finally smokes the cigarette. If a smoker arrives at the table and the ingredient needed is unavailable, they must wait (i.e., block) at the table until the ingredient is returned (busy-waiting is not allowed).

[2] Now politically incorrect.

```
% a.out 3 20 5 100
Barber  Chair   Balked  Waiting
/A/G/S
                        15
/C      15
/G
                        8
/C      8
/G/S
                        14
/C      14
                        1
                        1 4
                        1 4 2
/G/C    1               4 2
                        4 2 12
                18
                10
/G/C    4               2 12
/G/C    2               12
                        12 16
/G/C    12              16
                        16 9
                        16 9 3
/G/C    16              9 3
                        9 3 19
/G/C    9               3 19
/G/C    3               19
/G/C    19
                        6
/G/C    6
/G/S
                        0
/C      0
/G/S
                        5
/C      5
                        11
                        11 13
/G
                        11 13 7
/C      11              13 7
                        13 7 17
/G/C    13              7 17
/G/C    7               17
/G/C    17
/G/S
                        −1
/C      −1
/F
```

```
% a.out 3 20 5 100
Barber  Chair   Balked  Waiting
                        1
                        1 14
/A/G/C  1               14
/G/C    14
                        7
/G/C    7
/G/S
                        10
/C      10
                        17
/G/C    17
/G/S
                        8
/C      8
                        19
                        19 18
                        19 18 3
                11
/G/C    19              18 3
/G
                        18 3 16
/C      18              3 16
                        3 16 4
/G/C    3               16 4
/G/C    16              4
/G/C    4
/G/S
                        15
/C      15
                        5
/G/C    5
                        13
/G/C    13
                        9
/G/C    9
/G
                        2
                        2 6
/C      2               6
                        6 0
                        6 0 12
/G/C    6               0 12
/G/C    0               12
/G/C    12
/G/S
                        −1
/C      −1
/F
```

Figure 7.22: Sleeping Barber: Example Output

Each smoker has the following interface (you may add only a public destructor and private members):

```
_Task Smoker {
  public:
    enum states { needs, blocking, smoking };
    Smoker( Table<NoOfKinds> &table,    // shared table, NoOfKinds is the number of kinds
            const unsigned int Id,      // smoker identifier, value between 0 and N-1
            Printer &printer            // OPTIONAL, depends on where printing is performed
     );
};
```

A smoker begins by randomly choosing which kind of smoker it will be for its lifetime (one of the 3 kinds above); a value from 0–2. Next, the smoker randomly chooses the number of cigarettes it will smoke during its lifetime; a value between 0–19. Then the smoker begins obtaining ingredients to make and smoke the specified number of cigarettes, which involves the following conceptual steps for each cigarette:

a) obtain the package with the necessary ingredient from the table
b) removed the ingredient from the package (randomly yield between 0-2 times)
c) return the package to the table
d) make and smoke the cigarette (randomly yield between 0-9 times)

(Yielding multiple times is performed by calling task member yield(N).)

The interface for the table is (you may add only a public destructor and private members):

```
template<int kinds> class Table {
  public:
    Table( Printer &printer );
    void acquire( unsigned int Id, unsigned int kind );
    void release( unsigned int Id, unsigned int kind );
};
```

The number of kinds of smokers is passed to the table via the template parameter. (Essentially, the table does not need to know about the meaning of the different kinds of tasks.) A smoker calls the acquire member, passing its identification and kind of smoker, to obtain the package containing the missing ingredient needed to smoke a cigarette. The acquire member blocks the smoker if the particular package is currently unavailable. After conceptually removing the needed ingredient from the package, the smoker calls the release member to return the package to the table, and then pretends to smoke a cigarette. The table should check that the kind-of-smoker is valid; if the kind-of-smoker is invalid, print an appropriate message and exit the program. You may **NOT** use the μC++ semaphore member lock1.P(lock2) to simplify the program; your solution has to handle all synchronization and mutual exclusion cases using only the basic P and V operations.

All output from the program is generated by calls to a printer coroutine, excluding error messages. The interface for the printer is (you may add only a public destructor and private members):

```
_Cormonitor Printer {
  public:
    Printer( unsigned int NoOfTasks );
    void print( unsigned int Id, Smoker::states state, unsigned int kind );
};
```

(You do not need to know the details of a **_Cormonitor**; treat it solely as a coroutine with the magic property that only task can execute it at a time. Note, the coroutine main of a **_Cormonitor** *must* be a **private** memberThe printer attempts to reduce output by storing information for each smoker until one of the stored elements is overwritten. When information is going to be overwritten, all the stored information is flushed and storing starts again. Output should look similar to the following:

line no	output		
	S0	S1	S2
1	N P	N T	N T*
2	S*		B
3	N P*	S	
4	S		
5	N P		
6	S	N T*	S
7	N P*	S	
8	S		
9	N P		
10	S		
11	N P		
12	S		

Each column is assigned to a smoker with an appropriate title, e.g., "S0". A column entry is the state transition for that smoker, containing one of the following states:

State	Meaning
N k	needs a k resource from the table, where k is one of T (tobacco), M (matches), P (paper)
B	blocking for a resource
S	smoking

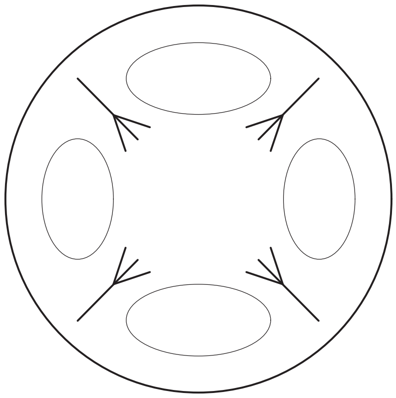
The asterisk $*$ marks the prior buffer information for the task causing the buffered data to be flushed. That is, the $*$ on line i appears above the value on line $i+1$ that caused the buffer to be flushed. The buffer is cleared by a flush so previously stored values do not appear when the next task flushes the buffer and an empty column is printed. For example, in the first line of the above output, S2 has the value "N T" in its buffer slot, and all the other buffer slots are full. When S2 attempts to print "B", which overwrites its current buffer value of "N T", the buffer must be flushed and a $*$ is placed beside its "N T" to indicate that S2 caused the flush. S2's new value of "B" appears on the next line. Note, a $*$ is **NOT** printed again for consecutive state transitions by the same task causing a flush (see lines 4–5 and 8–12). In the second line of the example, the buffer begins filling with values, until S0 needs to overwrite its value of "S" with "N P"; the buffer is flushed with an $*$ beside the "S" value for S0, and S0's new value of "N P" is placed in the buffer. Then S1 places a new value of "S" into the buffer. S0's next value of "S" causes a flush *and* a $*$ because there was an intervening insertion by S1. All output spacing can be accomplished using the standard 8-space tabbing, so it is unnecessary to build and store strings of text for output. Calls to the printer coroutine to perform printing may be performed from the table and/or a smoker task (you decide where to print).

The executable program is named smokers and has the following shell interface:

smokers [*no-of-smokers (1-10)*]

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) if no value is given for the number of smokers, assume 5. Assume the value is a valid integer, but check if it is in the range 1–10, and print an appropriate usage message and exit the program if outside the range.

6. A group of N (N $>$ 1) philosophers plans to spend an evening together eating and thinking (Dijkstra). Unfortunately, the host philosopher has only N forks and is serving a special spaghetti that requires 2 forks to eat. Luckily for the host, philosophers (like university students) are interested more in knowledge than food; after eating a few bites of spaghetti, a philosopher is apt to drop her forks and contemplate the oneness of the universe for a while, until her stomach begins to growl, when she again goes back to eating. So the table is set with N plates of spaghetti with one fork between adjacent plates. For example, the table would look like this for N=4:



Consequently there is one fork on either side of a plate. Before eating, a philosopher must obtain the two forks on either side of the plate; hence, two adjacent philosophers cannot eat simultaneously. The host reasons that, when a philosopher's stomach begins to growl, she can simply wait until the two philosophers on either side begin thinking, then *simultaneously pick up the two forks on either side of her plate* and begin eating. If a philosopher cannot get both forks immediately, then she must wait until both are free. (Imagine what would happen if all the philosophers simultaneously picked up their right forks and then waited until their left forks were available.)

The table manages the forks and must be written as a class using μC++ semaphores to provide mutual exclusion and synchronization. The table implementation has the following interface (you may add only a public destructor and private members):

```
class Table {
  public:
    Table( unsigned int noOfPhil, Printer &prt );
    void pickup( unsigned int id );
    void putdown( unsigned int id );
};
```

Member routines pickup and putdown are called by each philosopher, passing the philosopher's identifier (value between 0 and $N-1$), to pick up and put down both forks, respectively. Member routine pickup does not return until both forks can be picked up. To simultaneously pick up and put down both forks may require locking the entire table for short periods of time. No busy waiting is allowed; use cooperation among philosophers putting down forks and philosophers waiting to pick up forks. Your solution does not have to preclude starvation; it is sufficient that there is a finite number of noodles to be eaten.

A philosopher eating at the table is simulated by a task, which has the following interface (you may add only a public destructor and private members):

```
_Task Philosopher {
  public:
    enum states { THINKING, HUNGRY, EATING, WAITING, FINISHED };
    Philosopher( unsigned int id, unsigned int noodles, Table &table, Printer &prt );
};
```

Start each philosopher in the hungry state with the number of noodles on their plate passed to the constructor. Each time a philosopher gains control of both forks, she eats a random number of noodles, between 1 and 5 inclusive (see rand). Delay a random number of times, between 0 and 4 inclusive, to simulate the time to eat the noodles by using yield to give up the CPU time slice. When a philosopher finishes eating, she puts both forks down and delays a random number of times, between 0 and 19 inclusive, to simulate the time to think, unless there are no more noodles at which time the philosopher terminates.

All output from the program is generated by calls to a printer coroutine, excluding error messages. The interface for the printer is (you may add only a public destructor and private members):

```
% phil 5 20
Phil0  Phil1  Phil2  Phil3  Phil4
****** ****** ****** ****** ******
       H*     H
       E4,16
H      T*     H             H
H      H      H*     H      H
H      H*     E3,17  H      H
H      W1,2   E3,17* H      H
H      W1,2*  T      H      H
       E1,15
H      T      T*     H      H
H      T      H      H      H*
H      T*     H      H      E1,19
H      H      H      H      E1,19*
H*     H      H      H      T
E5,15  H      H*     H      T
E5,15* H      E5,12  H      T
T      H      E5,12  H*     T
T      H      E5,12* W3,4   T
T      H      T      W3,4*  T
                     E1,19
T      H      T      T      T*
T      H      T*     T      H
T      H*     H      T      H
T      E5,10  H*     T      H
T      E5,10* W2,3   T      H
T      T      W2,3*  T      H
T      T      E1,11  T*     H
T      T      E1,11* H      H
T*     T      T      H      H
H      T*     T      H      H
H      H      T      H*     H
H      H      T      E1,18  H*
H      H      T      E1,18* W4,0
H      H      T      T      W4,0*
H      H      T      T*     E1,18
H      H      T      H      E1,18*
H      H      T*     H      T
H*     H      H      H      T
E4,11
T      H      H*     H      T
T      H*     E1,10  H      T
T      W1,2   E1,10* H      T
              T
T      W1,2*  H      H      T
T      E5,5   H*     H      T
T      E5,5   W2,3   H      T*
                            H
T      E5,5   W2,3   H*     E5,13
T*     E5,5   W2,3   W3,4   E5,13
H      E5,5*  W2,3   W3,4   E5,13
H      T      W2,3*  W3,4   E5,13
H      T      E3,7   W3,4   E5,13*
H      T      E3,7*  W3,4   T
H      T      T      W3,4*  T
                     E3,15
H      T      T      T      T*
                            H
```

```
H      T      T      T*     E2,11
H      T      T      H      E2,11*
                            T
H      T*     T      H      H
H*     H      T      H      H
E3,8   H      T      H*     H
E3,8*  H      T      E2,13  H
T      H      T      E2,13* H
T      H      T*     T      H
              H
T      H      E2,5   T      H*
T      H      E2,5*  T      E4,7
T      H*     T      T      E4,7
       E2,3
T      T      T      T      E4,7*
                            T
T      T*     T      T      H
T*     H      T      T      H
H      H      T      T*     H
H      H*     T      H      H
       E3,0
...    F      ...    ...    ...
H             T      H      H*
H             T*     H      E5,2
H             H      H      E5,2*
                            T
H*            H      H      H
E5,3          H      H*     H
E5,3          H      E4,9   H*
E5,3*         H      E4,9   W4,0
T             H      E4,9*  W4,0
T             H      T      W4,0*
T             H      T*     E2,0
T             H*     H      E2,0
T*            E2,3   H      E2,0
H             E2,3*  H      E2,0
              T
H             H      H*     E2,0
                     E4,5
H             H      T      E2,0*
...    ...    ...    ...    F
H             H*     T
H*            E3,0   T
E2,1
T             E3,0*  T
...    ...    F      ...    ...
T                    T*
T*                   H
H                    H*
                     E4,1
                     T
                     H
H*                   E1,0
E1,0                 E1,0*
...    ...    ...    F      ...
E1,0*
F      ...    ...    ...    ...
**********************
Philosophers terminated
```

Figure 7.23: Output for 5 Philosophers and 20 Noodles

```
_Cormonitor Printer {
  public:
    Printer( unsigned int noOfPhil );
    void print( unsigned int id, Philosopher::states state );
    void print( unsigned int id, Philosopher::states state, unsigned int bite, unsigned int noodles );
};
```

A philosopher calls the print member when it enters states: thinking, hungry, eating, waiting, finished. The table calls the print member *before* it blocks a philosopher that must wait for its forks to become available. The printer attempts to reduce output by buffering information for each philosopher until one of the stored elements is overwritten. Output should look similar to that in Figure 7.23. Each column is assigned to a philosopher with an appropriate title, e.g., “Phil0”. A column entry is the state transition for that philosopher, containing one of the following states:

State	Meaning
H	hungry
T	thinking
W l,r	waiting for the left fork l and the right fork r to become free
E n,r	eating n noodles, leaving r noodles on plate
F	finished eating all noodles

Identify the left fork of philosopher$_i$ with number i and the right fork with number $i + 1$. For example, W2,3 means forks 2 and 3 are unavailable, so philosopher 2 must wait, and E3,29 means philosopher 1 is eating 3 noodles, leaving 29 noodles on their plate to eat later. When a philosopher finishes, the state for that philosopher is marked with F and all other philosophers are marked with `"..."`.

Buffered data is printed when existing data in the buffer is going to be overwritten. If a philosopher performs consecutive prints, then only the buffer data for that philosopher is printed (excluding the asterisk) and the other columns are empty. Otherwise, the entire buffer is flushed. Note, the buffer is not cleared by a flush so previously stored values appear when the next task flushes the buffer. An asterisk * appears next to the state that is about to be overwritten, when the buffer is flushed. In general, the * on line i appears above the value on line $i + 1$ that caused the buffer to be flushed. After a philosopher has finished, no further output appears in that column. All output spacing can be accomplished using the standard 8-space tabbing, so it is unnecessary to build and store strings of text for output.

In addition, you are to devise and include a way to test your program for erroneous behavior. This testing should be similar to the check performed in the routine CriticalSection, from software solutions for mutual exclusion, in that it should, with high probability, detect errors. When a problem is detected, use μC++’s uAbort routine to halt the program with an appropriate message. (HINT: once a philosopher has a pair of forks, what must be true about the philosophers on either side?)

The executable program is named phil and has the following shell interface:

```
phil [ P [ N ] ]
```

P is the number of philosophers; if P is not present, assume a value of 5. N is the number of noodles per plate; if N is not present, assume a value of 30. When present, assume each argument is a valid integer value, but ensure the number of philosophers is greater than 1 and number of noodles is greater than 0; print an appropriate usage message and terminate the program if either value is outside its range. The program must handle an arbitrary number of philosophers and noodles, but we will only test with values less than 10 and 100 for the two parameters, so the output tabbing works correctly.

(WARNING: in GNU C, -1 % 5 != 4, and on UNIX, the name fork is reserved.)

Bonus (for an additional 10% of the question’s value): The algorithm presented above has each philosopher wait to eat until both forks can be picked up simultaneously (using table locking), which reduces concurrency. Instead, implement an alternative strategy allowing a philosopher to pick up one fork at a time (do not implement both strategies). In addition, you must explain why your alternative strategy does not suffer from any situation like the one where each philosopher picks up their right fork and then waits until their left fork is available. For this version, no waiting W states need to be printed.

7. There are n children, all of whom eat Sema Four Grain Cereal, each box of which contains a plastic toy. The toys come in j styles and each style comes in k colours. Since it cannot be determined without opening the package (and removing most of the cereal) what style/colour combination is in a given box, and equal numbers of the $j * k$ style/colour combinations were produced, the effect of buying a box of cereal is to make a uniform random selection among $j * k$ alternatives. Even for quite small values of j and k, it can take rather a long time to accumulate a complete set (and, of course, before long the cereal manufacturer will switch to a completely different kind of toy). Therefore, the n children trade the toys among themselves in order to get a complete set sooner. (Each child starts with two toys, whose styles and colours are randomly determined.)

   You are to simulate this trading activity. The simulation has one task for each child and at least two additional tasks as specified below; additional tasks may be used if you wish. The interface for the child task is:

```
_Task Child {
  public:
    Child( int whoami, ToyBox &toybox );
    ~Child();
};
```

   Restrict the trading so that each child offers at most one toy for trading at a time, though multiple independent trades must be capable of being conducted simultaneously. All communication among tasks is done through a shared data-area. The interface for the shared data-area is:

```
class ToyBox {
  public:
    ToyBox( const int nChildren, const int nStyles, const int nColours );
    ~ToyBox();
    void wakeChildren();                      // Toymaker signals toy creation complete.
    void wakeEverybody();                     // Wakes up any sleeping tasks when game is over.
    void putChildToSleep( int who );          // Child calls this to block at start until gets a toy.
    void putToyMakerToSleep();                // Toymaker calls this to block.
    void wakeToyMaker();                      // Timer wakes up the toymaker.
    void receiveToy( int who, Toy newToy );   // Toymaker gives a toy for someone to toybox.
    Toy  findTradeToy( int who );             // Child finds trade toy.
    void tradeToy( int who, Toy newToy );     // Child trades toy or blocks if no trade partner.
    int  isWinnerYet();                       // Has a winner been declared yet?
};
```

   If you wish, a toy may be offered for trade for "any other" or for a more restricted possible set of exchanges, but the information given (at least on any single trade offer) must not reveal the current holdings of the trader, i.e., the other children should not be assumed to be completely co-operative. The tasks interact through shared storage protected by semaphores. Thus, you will likely have a storage area for each "child" task indicating whether that child is currently offering a toy for trading, what that toy is, and what restrictions have been put on the toy to be traded for it. An associated semaphore is used to protect access and modification of the area. Similarly, semaphores are used to block tasks, notably child tasks that are waiting for a suitable trade to be offered.

   Another per-child storage area within the ToyBox, protected by a semaphore, is used to pass new toys to a child. It is possible that a child fails to pick up a toy from this area before the next toy arrives. (This is likely to be a bug, but it could be the result of unfortunate parameter settings and an overload of trading activity.) In such a case, it is legitimate for the toy-generation task to discard the new toy. In designing the use of shared storage and semaphores in your program, be sure to use the class mechanism of C++ appropriately, rather than using an arbitrary collection of global variables.

   The ToyMaker task creates new toys (i.e., finding the toy in the cereal box). It periodically generates a new toy for each child. The new toys can be for all children simultaneously or a toy may be generated for child 1, followed by a delay, then a toy for child 2, a delay, and so on, provided each child gets toys at approximately the same rate. A pseudo-random number generator is used to determine the style and colour of the new toy. If a child task is blocked waiting for a trade when a new toy is generated, the toy-generation task must unblock it to accept the new toy. The interface for the toy maker is:

```
_Task ToyMaker {
  public:
    ~ToyMaker();
    ToyMaker( ToyBox &toybox, const int nChildren, const int nStyles, const int nColours );
};
```

Every time it wakes, or every mth times it wakes for some m you select, it generates a new toy or set of toys.

One additional task provides a simulated timing facility. It is used to implement the "periodic" aspect of the ToyMaker task. The interface for the timer task is:

```
_Task Timer {
  public:
    Timer( const int delay, ToyBox &toybox );                  // Constructor
    ~Timer();
};
```

The timing task contains a loop with a yield(newdelay) followed by a call to wakeToyMaker to "wakes up" the toy-generation task. You do not need to use the timing task for any other purpose, but you are required to implement it separately from the toy-generation task to keep open the possibility of using it for other purposes. For example, you could use it to unblock child tasks that have been blocked for rather a long time, allowing the child to select an alternative toy to offer for trading. Note that such unblocking is not a requirement of the assignment, although unblocking on arrival of a new toy is required.

Stop the simulation as soon as any child obtains a complete set of toys. Print out the identity of the child that has a complete set, the exact holdings of each child, and the total number of toys held by all. Note that you need to shut down all tasks to terminate your program cleanly. This requirement means each child task must be informed a complete set has been obtained, unblocking it if it is currently blocked, and the timing and toy-generation tasks must also be told to terminate. You are to decide the most appropriate means to use in informing your various tasks that they should terminate. You must provide appropriate trace output showing the arrival of new toys and all trading activity, so that the actions of the program can be followed. The precise form of the trace output is not specified here; designing good trace output is part of the assignment. Note that thoroughness and conciseness are both desirable properties of trace output. You should provide a trace that contains substantial information that is genuinely useful, but not excessive information or information presented in an unnecessarily verbose fashion.

Name your program toys. The number of children may be compiled into your program, but must be easy to change. Use the value $n = 4$ for testing. Other parameters are specified on the command line:

toys *j k [seed]*

The parameters *j* and *k* are as specified above; each must be a positive integer. The parameter *seed* specifies the seed value for the random-number generator. The last parameter is optional, the other two are required. You may assume that the syntax of the command line is correct in that the appropriate number of integer values will be present. If the *seed* parameter is omitted, a value is generated in the program. (Check the documentation for the random-number generator you choose to determine whether there are restrictions on seed values.)

Test your program using reasonably small parameter values, for example $j = 2$ and $k = 3$, to avoid excessive execution and trace output before a complete set of toys is achieved by some child. A program such as this one is extremely difficult to test thoroughly because you cannot directly control which parts of the algorithm are executed. Instead, you must inspect trace output to determine what was exercised on a particular execution. Make sure that you have dealt with all potential race conditions.

This program is significantly more complex than preceding ones. Be sure to begin by implementing the simplest possible version of the program. For example, begin by making all offers to trade for "any other" and only add more complex trading restrictions if (a) you have the simple program working and (b) you really want to.

Chapter 8

Concurrency Errors

The introduction of threads and locks into a programming language introduces new kinds of errors not present in sequential programming; several of these errors have been mentioned in previous chapters. When writing concurrent programs, it is important to understand the new kinds of errors so they can be avoided or debugged when they occur. Therefore, it is appropriate to take a short diversion from the discussion of specifying concurrency to explain these new programming problems.

As mentioned at the end of Section 5.2, p. 116, the temporal nature of concurrent programs makes avoiding and debugging errors extremely complex and very frustrating. The following discussion revisits some of the problems already discussed and introduces several new ones in an attempt to give a deeper understanding of the issues.

8.1 Race Error

A **race condition** occurs when two or more tasks share a resource without synchronization or mutual exclusion. As a result, the shared resource can change (be written to) by one task while another task is reading or writing it. A race condition does not imply an error, e.g., a race condition is used in Peterson's algorithm (see Section 6.3.6.2, p. 148) to break ties for simultaneous arrivals. In general, a race condition is used to generate some form of non-determinism among tasks, which can be used in various ways, such as a random-number generator or making an arbitrary decision.

A **race error** occurs when a race condition is inadvertently created by a programmer by failing to synchronize before performing some action on a shared resource or provide mutual exclusion for a critical section using the shared resource. As a result, two or more tasks race along erroneously assuming that some action has occurred or race simultaneously into a critical section. In both cases, the tasks eventually corrupt the shared data (and possible the non-shared data indirectly through the shared data), i.e., variables become incorrect in any number of ways. A race error is the most insidious error in concurrent programming and the most difficult to debug.

There are many issues in locating a race error. The first issue goes back to the basics of concurrent programming: identifying shared variables and ensuring proper synchronization of and mutual exclusion for these variables by multiple tasks. Sometimes subtle interactions occur that the programmer is unaware of, e.g., unknowingly calling a library routine using shared variables but having no mutual exclusion. (This situation occurs frequently in UNIX systems that are not thread-safe.) However, imagining all the possible interactions among tasks and shared data in a concurrent program can be extremely difficult, so it is possible in a complex concurrent system for a synchronization point or mutual exclusion to be missed. The next chapter begins the discussion of high-level concurrency constructs, combined with software engineering techniques, to help localize access to shared data, and hence, mitigate both of these problems.

The second issue in locating a race error is that the program becomes corrupted because of incorrect access to shared data, but the corruption does not occur all the time nor does it cause immediate failure. A program might run for years before a particular execution sequence results in a race that corrupts shared data. Furthermore, once the shared data is corrupted, the program might continue for a significant time before the corruption cascades sufficiently to cause an actual failure. In this case, the error occurs at a substantial distance, both in time and location, from the original problem. While this situation occurs even in sequential programs, multiple threads make it more pronounced.

All of these issues conspire to make locating the original problem extremely difficult, because to a large extent there is little or no pointer back to the problem area. Some systems provide the ability to generate information events for certain kinds of changes and/or interactions in a concurrent program. This stream of events can be analysed to look for situations that might represent missing synchronization or mutual exclusion. However, in many cases, generating the event stream perturbs the program sufficiently to change its behaviour so the race error does not occur. Such a perturbation is called the **probe effect**, i.e., connecting a probe to the program for examination changes its behaviour resulting in a "Heisenbug" (see page 117). In this case, the only way to locate a race error is by examination of the program code and performing thought experiments about what might happen with respect to different execution sequences.

However, there is often sufficient latitude to insert some checking code without causing the problem to vanish. One simple way to check for violations of mutual exclusion is by using a trap, like the technique shown in the self-checking bathroom (see Section 6.2.2, p. 139). Traps can be placed at various locations in a program validating mutual exclusion at very low cost, and hence, having little probe effect. A similar trap can be constructed for synchronization by setting a shared variable indicating work completed by one task and checking the variable before continuing in another task.

8.2 No Progress

The next group of errors are all similar in one respect: each results in one or more tasks ceasing to make forward progress with respect to its computation. That is, a task stops performing its specified computation and appears to do nothing. While all the following errors share this common symptom, each has a subtly different way of manifesting itself, and hence, a correspondingly different kind of solution to solve the problem. Understanding each form of no-progress error is crucial to knowing how to fix it.

8.2.1 Live-lock

Live-lock was introduced and defined in Section 6.2.1, p. 138. In general, it occurs when a selection algorithm, e.g., mutual exclusion, can postpone indefinitely, i.e., the "You go first" problem. All live-lock problems occur on simultaneous arrival, and all can be solved given a good tie breaking rule. Notice, there is no requirement to be fair in breaking the tie; fairness is a separate issue. The goal is to make a selection as quickly as possible on simultaneous arrival, e.g., to get a task into a critical region as fast as possible.

An excellent example of live-lock is a traffic intersection with a 4-way stop. In general, when cars arrive simultaneously at a 4-way stop, the rule to break the tie is the car on the right[1] has the right of way and goes first. However, what about the situation where 4 cars arrive simultaneously (see Figure 8.1(a))? This situation results in a live-lock because each driver is deferring to the car on the right. As a result, the cars sit at the intersection forever! (It is humorous to note this potentially life threatening situation should be added to the many others that occur when driving a car.) The problem is that the tie breaking rule is simply insufficient to handle all the simultaneous arrival cases. Augmenting the rule to say the driver with the largest license plate number goes first would solve the problem (and there are other possible solutions, too). In fact, any system in which the components can be uniquely numbered, e.g., unique task identifiers, can solve the live-lock problem by devising a tie breaking rule using this value for all simultaneous arrival situations.

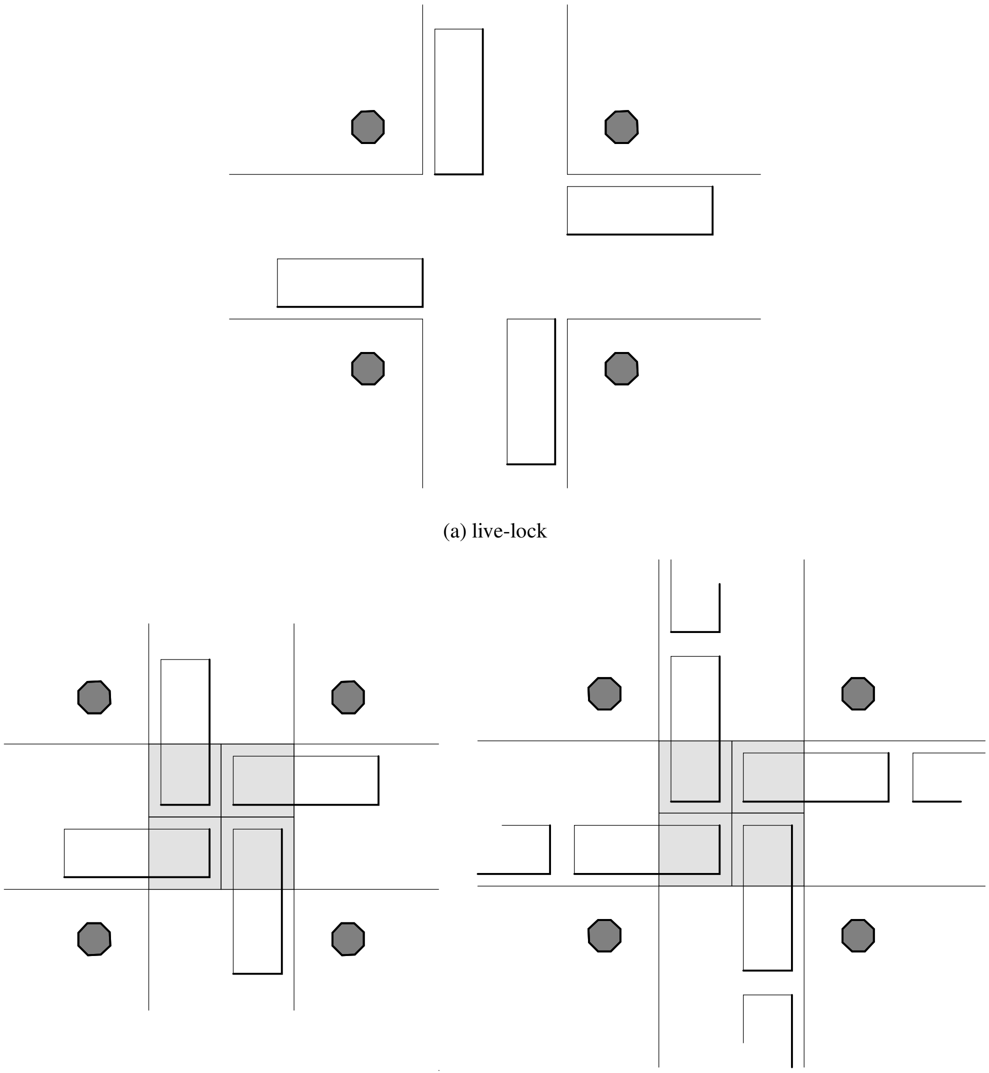
Live-lock problems are usually easy to find in a concurrent program either with a concurrent debugger or using print statements. The usual symptom for live-lock is a non-terminating application using large amounts of CPU time. The reason for this symptom is that two or more tasks are spinning in a live-lock. By attaching a concurrent debugger to the program, stopping all execution, and examining the execution locations of the tasks, it is usually possible to quickly spot the spinning tasks and where they are spinning. Using print statements is not as good because it may perturb execution sufficiently so the simultaneous arrival does not occur often or at all. Furthermore, it can generate a huge amount of output before a live-lock occurs.

What is unusual about live-lock is that a task is not making progress yet it is conceptually ready at any time, switching between the active and ready states, so it is consuming CPU resources but accomplishing nothing.

8.2.2 Starvation

Starvation was introduced and defined in Section 6.2.1, p. 138. In general, it occurs when a selection algorithm, e.g., mutual exclusion, allows one or more tasks to make progress while ignoring a particular task or set of tasks so they are never given an opportunity to make progress. (In live-lock, no task is making progress.) All starvation problems occur when there is a persistent lack of fairness in task selection; temporary unfairness is acceptable as long as there

[1] The car on the left if you drive on the other side of the road.



(a) live-lock

(b) deadlock

(c) gridlock

Figure 8.1: No Progress

is a bound on the length of the wait. Hence, starvation requires some notion of high and low priority, where the priority can be based on anything. For true starvation to occur, a low priority task must never be selected, which requires that a high priority task always be available for selection over the low priority task. In practice, there is usually some window where no high priority task is available for selection and so a low priority task makes progress. Hence, starvation usually requires an extremely busy system, often so busy that starvation is the least of the problems. Therefore, starvation is more a theoretical problem than a practical one.

However, just hoping that a concurrent system does not have starvation is asking for trouble. It is always better if an algorithm can be designed to be free of starvation. The reason is that it may appear a particular selection algorithm, with respect to a set of tasks and resources, seems fine, but in conjunction with another selection algorithm, problems may arise. Even though there is no true starvation in finite programs, there may be unwanted periods of unfairness, which result in the short-term delay of tasks. The short-term delay may not cause an error but may produce a behaviour that was neither anticipated nor desired by the programmer. In the case of real-time programs (see Chapter 14, p. 415), even a short-term delay can result in catastrophic errors. Finding short-term unfairness usually requires a trace of task scheduling to reveal the order tasks were delayed and made ready. Often this is sufficient information to locate where and why tasks were scheduled in that order.

Like live-lock, a starving task is not making progress yet it is conceptually ready at any time, switching between the active, ready and possibly blocked (depending on the implementation) states, so it is possibly consuming CPU resources but accomplishing nothing.

8.2.3 Deadlock

This error has been mentioned but not discussed before and is associated with the phenomenon of tasks blocking for either synchronization or mutual exclusion. It is possible for one or more tasks to get into a state where no progress is possible because the tasks become blocked forever, called **deadlock**. This state is in contrast to live-lock or starvation where the tasks are conceptually ready but cannot get scheduled; in deadlock, tasks are actually blocked. In this discussion, traditional deadlock is divided into two forms. The reason for the subdivision is that traditional deadlock encompasses two quite different problems, each with its own particular kind of solution. The two different problems are associated with synchronization and mutual exclusion, respectively. Systems that report deadlock usually do not differentiate between the kind of deadlock which has occurred. Using different terms more precisely identifies the problem, which correspondingly defines the potential solutions, because synchronization deadlock is usually a trivial mistake in cooperation, while mutual-exclusion deadlock is usually a complex mistake in resource allocation.

8.2.3.1 Synchronization Deadlock

When there is a failure in cooperation, so a task blocks but is never unblocked by a cooperating task, it is called **synchronization deadlock** (sometimes called **stuck waiting**). The simplest example of synchronization deadlock is the following:

```
void uMain::main() {
    uSemaphore s(0);    // lock is closed
    s.P();              // wait for lock to open
}
```

In this program, task uMain blocks on semaphore s but there is no other task to subsequently unblock it. Clearly, there is a failure in cooperation when a task waits for some event that never happens. Alternatively, the semaphore is incorrectly initialized to 0 instead of 1. Notice, this is not the same as a race error where a task forgets to wait; in this case, the task waits but no other task restarts it.

Synchronization deadlock problems are usually simple mistakes in cooperation. The usual symptom for synchronization deadlock is application termination with a deadlock message or a non-terminating application using no CPU time. The reason for this latter symptom is that one or more tasks are blocked in a synchronization deadlock, preventing the program from terminating. By attaching a concurrent debugger to the program, and examining the execution locations of the tasks, it is usually possible to quickly spot the blocked task and on which lock it is blocked.

8.2.3.2 Mutual Exclusion Deadlock

When a task blocks while attempting to acquire mutual exclusion for a shared resource, but is never unblocked because the resource never becomes free, it is called **mutual-exclusion deadlock**. An example of mutual-exclusion deadlock can be constructed using the previous traffic intersection with a 4-way stop. After the 4 drivers sit for a few minutes deferring to the car on the right, they all simultaneously decide to break the law (i.e., not defer to the driver on the right) and proceed through the intersection (see Figure 8.1(b)). In addition, all 4 cars have no reverse gear so they cannot back up! At this point, absolutely nothing can be done by the drivers to fix the situation; they are trapped forever in the intersection with no mechanism for escape. Dijkstra referred to this situation as a "deadly embrace" [Dij65, p. 105] because the only solution is to terminate one of the participants associated with the deadlock. In this case, one or more of the cars has to be towed out of the intersection to terminate the deadlock and allow further progress.

While it is highly unlikely that all 4 cars are without reverse gear, what if the cars behind move forward (see Figure 8.1(c)). This situation is called **gridlock** and happens in large cities during rush hour. In general, there is a traffic law which states it is illegal to enter an intersection unless the entire car has the opportunity to proceed completely through it. This law ensures gridlock cannot occur; unfortunately, people do not always obey this law! (Notice, this law still does not prevent deadlock in the previous case, because at the moment each driver decides not to defer to the driver on the right, it appeared safe to enter the intersection. Therefore, even if a law is created that prevents live-lock, the drivers could be fast enough to all enter the intersection and form a deadlock.)

Mutual exclusion deadlock can occur in many ways in a concurrent program. One of the simplest ways is by nesting acquisition of locks for acquiring resources, as in:

```
uSemaphore L1(1), L2(1);
task1                 task2
L1.P()          L2.P()      // acquire opposite lock
    R1              R2      // access resource
    L2.P()          L1.P()  // acquire opposite lock
        R2              R1  // access resource
```

In this example, task_1 acquires lock L1 to access resource R1 and then attempts to acquire lock L2 to access resource R2. At the same time, task_2 acquires lock L2 to access resource R2 and then attempts to acquire lock L1 to access resource R1. The result is that both task_1 and task_2 block on the inner P operation because they cannot acquire the other lock as it is held by the other task. When the tasks block, there is a deadlock because task_1 is holding lock L1 and waiting for lock L2, while task_2 is holding lock L2 and waiting for lock L1. Hence, both tasks are waiting for a resource that will not be freed, and since both tasks have blocked, neither can give up their resource so the other can make progress. Notice, the two tasks must conceptually acquire and then wait *simultaneously* (on a single CPU, the execution switching must occur such that it appears simultaneous). This code might run without problem for years until one day an execution sequence occurs that produces this particular scenario. Furthermore, these two code fragments can be located in different parts of the program where each is correct on its own.

The preceding explanation for mutual-exclusion deadlock only gives an ad hoc definition; the formal definition is as follows. There are 4 conditions [CES71, p. 70] that must be true *simultaneously* for a set of tasks to achieve a mutual-exclusion deadlock. Each of these conditions is demonstrated through the traffic intersection with a 4-way stop.

1. There exists a shared resource requiring mutual exclusion (critical section).

   The traffic intersection is the shared resource. While the entire intersection could be treated as a single resource, doing so underutilizes the resource. For example, what is the maximum number of cars that can be simultaneously in an intersection without causing an accident? The answer is 4, because 4 cars can be simultaneously turning right without interfering with each other. Therefore, an intersection clearly has multiple shared components requiring mutual exclusion because two cars cannot occupy the same space at the same time.

2. A task holds a resource while waiting for access to a resource held by another task (hold and wait).

   In the deadlock picture above, each car is holding the portion of road it is sitting on (the gray area in the picture). As well, each car is waiting for the portion of the road directly in front of it. Therefore, each car is holding a resource, part of the traffic intersection, and waiting for another resource, another part of the traffic intersection.

3. There exists a circular wait of tasks on resources (circular wait).

   In the deadlock picture, it is easy to see the cycle among the cars; each car is holding a resource that is being waited for by the car to the left of it.

4. Once a task has acquired a resource, it cannot be taken back (no pre-emption).

   Because the cars have no reverse gear, or more realistically the cars behind move forward, the resource being held (part of the traffic intersection) cannot be given back by the drivers. For tasks, this problem manifests itself because they block, and hence, cannot perform any work, such as giving up a held resource.

For example, if all 4 drivers advance halfway into the intersection, each of the previous conditions occurs, and hence, a mutual-exclusion deadlock occurs. However, if one of the drivers proceeds through the intersection before the others start, live-lock is broken, and even though 2 cars can get into a hold and wait, a deadlock cannot occur because the third car can proceed so no cycle can form. Hence, the 4 conditions, plus the simultaneous requirement, must exist for mutual-exclusion deadlock.

Why is synchronization deadlock not mutual-exclusion deadlock? This question can be answered by showing that synchronization deadlock does not satisfy any of the deadlock conditions required for mutual-exclusion deadlock. Is there a critical section requiring mutual exclusion? For synchronization deadlock, the lock a task acquires is being used for synchronization not protecting a critical section. Is the task holding a shared resource and waiting for another such resource? For synchronization deadlock, the task has neither acquired a resource (not even the lock) nor is it waiting for another resource. Can the acquired resource be given back through some form of pre-emption? Since no resource is acquired, there is nothing that can be given back. Is there a circular wait among tasks holding and waiting on resources? Since there are no resources and there is only one task, there cannot be a circular wait. Finally, there is no set of circumstances or conditions that must occur simultaneously to produce a synchronization deadlock.

Interestingly, both mutual exclusion and synchronization deadlock share the same visible symptoms: application termination with a deadlock message or a non-terminating application using no CPU time. The reason for this latter symptom is that one or more tasks is blocked in a mutual-exclusion deadlock, preventing the program from terminating. Because of the similar symptoms, it is common to refer to both kinds of deadlock with the generic term deadlock, but the cause of each kind of error is very different. By attaching a concurrent debugger to the program and examining the execution locations of the tasks, it is usually possible to determine resources allocated to tasks and outstanding resource requests causing tasks to block. However, this information only indicates the deadly embrace, not how the embrace formed; therefore, it is necessary to work backwards through task execution histories to determine why the embrace formed, which can be a complex analysis. Trace events showing the execution and allocation ordering may be necessary to locate the reason for a complex deadlock. If the concurrent system does not provide a tracing facility, it can be mimicked with debug print statements, albeit with substantial work on the part of the programmer.

Mutual exclusion deadlock problems are more complex mistakes involving multiple tasks accessing multiple resources. The usual symptom for mutual-exclusion deadlock is termination of the application with some form of deadlock message, or the application does not terminate and is using no CPU time. The reason for this latter symptom is that one or more tasks is blocked in a mutual-exclusion deadlock, preventing the program from terminating. (Because this symptom is the same as for synchronization deadlock, it is common to refer to both kinds of errors with the generic term deadlock.) By attaching a concurrent debugger to the program, and examining the execution locations of the tasks, it is usually possible to determine which tasks have resources allocated and which resources tasks are blocked on. However, this information only indicates the deadly embrace, not how the embrace formed; therefore, it is necessary to work backwards to some point during the execution of the tasks to determine why the embrace formed, which can be a complex analysis. Trace events showing the execution and allocation ordering may be necessary to locate the reason for a complex deadlock. If the concurrent system does not provide a tracing facility, it can be mimicked with debug print statements, albeit with substantial work on the part of the programmer.

8.3 Deadlock Prevention

Is it possible to eliminate deadlock from a concurrent algorithm? Yes, but only if some conditions that lead to the deadlock can be eliminated. Notice, prevention occurs during the static design of an algorithm not when the algorithm is running, which means the program using a prevention algorithm cannot deadlock no matter what order or speed the tasks execute. Since the requirements are different for synchronization and mutual-exclusion deadlock, each is discussed separately.

8.3.1 Synchronization Deadlock Prevention

To prevent synchronization deadlock all synchronization must be eliminated from a concurrent program. Without synchronization, no task waits for an event to occur, and hence, there is no opportunity to miss an uncommunicated event from a cooperating task. However, without synchronization, tasks cannot communicate as it is impossible to establish safe points for transferring data, which means the only programs that can be written are ones where tasks run independently, possibly causing a side-effect, such as a clock task managing a clock on the terminal screen. Notice, this restriction even eliminates divide-and-conquer algorithms (see Section 5.11, p. 130) because they require termination synchronization. Therefore, preventing synchronization deadlock severely limits the class of problems that can be solved, making it largely impractical for almost all concurrent programming.

8.3.2 Mutual Exclusion Deadlock Prevention

When designing a concurrent algorithm, it may be possible to statically ensure that one or more of the conditions for mutual-exclusion deadlock cannot occur. Each of the conditions are examined as to the feasibility of removing it from a concurrent algorithm.

1. no mutual exclusion

If a concurrent algorithm requires no mutual exclusion, there can be no mutual-exclusion deadlock. For example, divide-and-conquer algorithms require only synchronization but no mutual exclusion, and therefore, cannot encounter mutual-exclusion deadlock. However, no mutual exclusion means virtually no shared resources, which significantly limits the class of problems that can be solved, but still allows many reasonable programs.

2. no hold and wait

   To eliminate hold and wait requires no shared resource be given to a task unless all shared resources it requests can be supplied so the task never waits for a resource. In general, this requirement means that before a concurrent program begins execution it must identify its worst case allocation to the runtime system. Based on this information, a scheduling scheme is statically developed that precludes deadlock.

   While this restriction does preclude mutual-exclusion deadlock, it has unfortunate side-effects. First, identifying worst-case resource requirements may be impossible because resource needs may depend on dynamic information, like an input value. Second, even if worst case resource requirements can be identified statically, they may never be reached during execution nor all needed at the start of the program. As a result, there is poor resource utilization because more resources are allocated than actually needed to handle a situation that rarely occurs, and these resources may be held longer than is necessary, e.g., from start to end of program execution. For certain programs with high worst case resource requirements, there is the further problem of a long delay or possible starvation because the necessary set of resources is difficult or impossible to accumulate while other programs are running on the system.

   For example, in the previous traffic intersection problem, the law that a car must not proceed into an intersection unless it can proceed completely through the intersection imposes a control on hold and wait. A driver must have sufficient space on one of the 3 exit roads before being allocated any space in the intersection, i.e., 1 square to turn right, 2 squares to drive straight through, and 3 squares to turn left. By ensuring that all drivers follow this policy, gridlock cannot occur.

3. allow pre-emption

   Interestingly, pre-emption is a dynamic phenomenon, i.e., the runtime system makes a dynamic decision at some arbitrary time, from the perspective of the executing tasks, to take back some resource(s) given out earlier. Therefore, a programmer cannot know statically which resource(s) or when or how often the request to return one might occur. Hence, statically, the programmer must assume any or all resources can be pre-empted at any time and must attempt to compose an algorithm given this severe requirement. In general, it is impossible to compose reasonable algorithms given such a requirement.

   It is possible to show that pre-emption is unnecessary in a system by doing the following. When a task holding resources attempts to acquire another resource that is currently unavailable, the task does not block but is required to give up all the resources it has acquired and start again. In effect, this approach is a form of self-pre-emption. Again, this is largely impractical, except in very unusual circumstances, where the number of acquired resources is small and/or the chance of a resource being unavailable is extremely low. As well, the problem of restarting some or all of the program's execution may require undoing changes made to the environment. (This point is discussed further.)

4. no circular wait

   If it is possible to show a circular wait cannot occur, then mutual-exclusion deadlock cannot occur. For many specific situations it is possible to eliminate the potential for circular wait by controlling the order that resources are acquired by tasks. In the previous example of nested acquisition of locks, it is possible to prevent deadlock by ensuring both tasks allocate the resources in the same order, as in:

```
uSemaphore L1(1), L2(1);
task1                  task2
L1.P()          L1.P()       // acquire same lock
    R1              R1       // access resource
    L2.P()          L2.P()   // acquire same lock
        R2              R2   // access resource
```

Since only one task can acquire L1, the other task cannot make progress toward acquiring L2, so it cannot hold the other resource; therefore, the task holding L1 never has to wait to acquire L2. While this order of resource allocation precludes mutual-exclusion deadlock, it has the drawback that task_2 is now allocating resources in the reverse order it needs them, which may result in poor resource utilization. In fact, virtually all resource allocation ordering schemes have a side-effect of either inhibiting concurrency and/or reducing resource utilization, in exchange for providing mutual-exclusion deadlock free execution. Therefore, the choice is between performance versus robustness.

Controlling the order that resources are allocated can be specific or general. Many concurrent algorithms can be subtly modified to control allocation ordering, while limiting negative side-effects. Clearly, the more known about an algorithm, the more specific the control ordering can be made to optimize for the algorithm's particular behaviour. For example, if tasks are organized in a ring with a shared resource between each (see questions 6, p. 225 and 9, p. 301), as in:

R_5 T_1

T_5 R_1

R_4 T_2

T_4 R_2

R_3 T_3

and each task needs *both* the resource on the left and right occasionally, there is the potential for mutual-exclusion deadlock if each task acquires the right resource and then tries to acquire the left resource because of the hold and wait cycle. However, there are many control orderings (schedulings) that can be applied in this specific scenario to prevent mutual-exclusion deadlock. One example is that a task cannot acquire a resource unless it can acquire both of them. Alternatively, have each task acquire the right resource first and then the left, except for one task, which acquires the left resource first and then the right. This control ordering ensures the task on the right of the special task can always acquire both resources and make progress. The negative side-effect, in both cases, is inhibiting some concurrency and under utilizing the resources. Concurrency is inhibited because a task may be able to make some progress with only one of the two resources, and resources are underutilized because of the required order of allocation.

While many algorithm specific schemes exist for controlling resource allocation ordering, there are also several general approaches, such as the **ordered resource policy** (ORP) [Hav68, p. 78]. These control ordering schemes work for different kinds of algorithms, but must assume worst case situations because of a lack of detailed knowledge of the algorithms; therefore, they are not as efficient as specific control schemes. The ORP approach is similar to the N-task software solution presented in Section 6.3.7, p. 149, in that a task acquires resources from lowest priority to highest. Formally, the ORP approach is as follows:

- divide all resources into classes: R_1, R_2, R_3, etc.

  This rule creates logical classes and assigns monotonically increasing priorities to each class (i.e., the resource class subscript is the priority). A class can contain multiple instances, which might be identical or different.

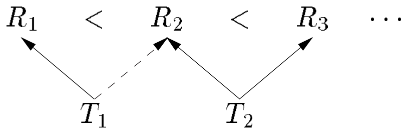
- tasks can only request a resource from class R_i if holding no resources from any class R_j such that $j \geq i$

  This rule requires resources to be allocated strictly from lowest priority class to highest.

- if a resource contains multiple instances, requesting several instances must occur simultaneously

  This rule follows from the fact that an allocated resource must be from a class with higher priority than the currently held maximum priority for a task. Hence, a task cannot acquire one R_i, and then attempt to acquire another. To get N instances of a particular resource requires all N be obtained in a single request. If this rule is relaxed to allow an equality, $i = j$, two tasks could both be holding an instance of R_i and request another that is unavailable, resulting in a mutual-exclusion deadlock.

Because the preceding inequality is strict, a circular wait is impossible. The proof is straightforward.



Denote the highest class number for which task T holds a resource by $h(T)$. If task T_1 requests a resource of class k and is blocked because that resource is held by task T_2, then it follows $h(T_1) < k \leq h(T_2)$. Allowing T_1 to block cannot affect T_2 because T_2 has already acquired all resources greater than or equal to k that it needs, and hence, T_1 cannot be holding resources needed by T_2. Therefore, there can be no hold and wait cycle.

In some cases there is a natural division of resources into classes, making the ORP work nicely. In other cases, some tasks are forced to acquire resources in an unnatural sequence, complicating their code, inhibiting concurrency, and producing poor resource utilization.

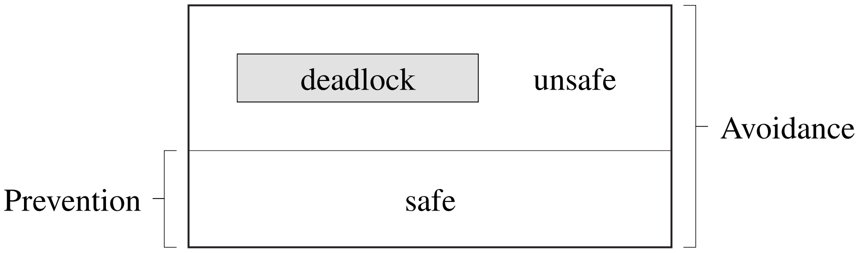
For the traffic intersection, preventing mutual-exclusion deadlock through no circular wait would involve rules that allow cars into the intersection, e.g., to start a left turn, but ensure the car can ultimately clear the intersection. Traffic lights (as opposed to a 4-way stop) provide additional control to preclude some circular waits from forming, such as a left turning car.

Finally, there may be programs where each of the four conditions may hold among tasks at different times, but it is possible to demonstrate there is no point during execution when all four conditions are true at the same time so deadlock cannot occur.

8.4 Deadlock Avoidance

Deadlock prevention is achieved by a programmer constructing a concurrent program using carefully designed algorithms. Prevention ensures an algorithm is deadlock free *before* it is executed by a concurrent program, establishing a level of robustness that may be crucial for certain applications. The drawback of deadlock prevention is that certain algorithms are not amenable (e.g., dynamic resource requirements), and resources may be underutilized or concurrency inhibited or both. For certain situations, these drawbacks are not as important as the robustness. For example, NASA is happy to buy more powerful computers and add more resources if it means the software in the space shuttle computers is deadlock free. The last problem NASA needs is a deadlock occurring during take off.

However, when the restrictions imposed by deadlock prevention are unacceptable, it is necessary to switch to a dynamic approach where algorithms essentially take risks, moving from a guaranteed safe status to a potentially unsafe status to achieve better performance and utilization, but are prevented from actually entering deadlock during execution by the runtime system.



While deadlock prevention is an active approach with respect to the programmer, i.e., building deadlock free algorithms, deadlock avoidance is a passive approach where the runtime system does all the work. However, there is an administrative cost for avoiding deadlock at runtime: the runtime system must monitor execution at varying levels of detail depending on the particular technique used. This monitoring consumes both memory and CPU resources.

If the runtime system detects a deadlock would occur when a task is about to block on a lock for synchronization or mutual exclusion reasons, the system basically has two options with respect to the blocking task:

1. terminate the blocking task or program containing it

   In many situations this is the only option, regardless of the severity of the option. However, it gives the programmer no possibility to deal with the problem programmatically.

2. return immediately from the lock request indicating a refusal to allow the task to acquire the lock and spin or block waiting for it to become free.

   In this option, the task must not spin or block waiting for the resource to become free. The task can only come back later and try again or release some held resources to ultimately acquire the resource associated with the lock. However, there is a significant design cost for programmers as a program must be written to deal with the potential of lock refusal. A programmer may have to add substantial code to deal with this new situation during runtime, which may have little or nothing to do with the algorithm performed in the program.

The following discussion explains how the runtime system can detect an approaching deadlock when a task is about to block.

8.4.1 Synchronization Deadlock Avoidance

Avoiding synchronization deadlock involves detecting when a blocking task will not be subsequently unblocked; this requires a crystal ball to look into the future. In general, it is impossible to tell at the time a task blocks on a lock if another task will eventually unblock it. Essentially, only if the last task in the system is about to block on a lock and all the other tasks are blocked on locks, is it possible to conclude that synchronization deadlock is about to occur, and this task should not be allowed to block. (This technique is used in μC++ to know when to print the deadlock error-message and terminate execution.) At all intermediate system states, it is always possible that some task, with visibility to the lock(s), could unblock a task from it. Extensive static analysis might determine that some or all tasks cannot see every lock, or that each task cannot lock and unlock every lock, and hence, the tasks and locks could be formed into logical groups, and a synchronization deadlock only occurs if all tasks in the logical group block. However, the existence of pointers to locks makes such analysis extremely difficult, if not impossible, in many cases. Therefore, without extensive static analysis, avoidance of synchronization deadlock can only be accomplished by checking for the last running task of a program attempting to block, and avoiding this block.

8.4.2 Mutual Exclusion Deadlock Avoidance

Avoiding mutual-exclusion deadlock requires the ability to dynamically track all resource allocation and ownership by tasks, which usually requires a single resource allocator managing all resource allocation and free requests. The resource allocator gives out available resources, and blocks tasks if a resource is unavailable, unless blocking might lead to deadlock. The question the resource allocator must answer is: will blocking a task lead to deadlock? Therefore, it must maintain sufficient information about what is occurring in the system to decide how to respond to each resource request. The decision regarding a request does not have to be absolutely accurate. In other words, denying a request does not imply that blocking the requester would produce a deadlock. The decision can be conservative, meaning denying a request may not lead directly to deadlock but could be too close to deadlock for the allocator to allow blocking. For resource allocation, there are criteria on which such a decision can be made.

Two resource allocation schemes are examined. The first scheme is historical and not precise in that it might deny a request which would not lead to deadlock. The second scheme is precise by maintaining sufficient information to know for each request whether it will cause a mutual-exclusion deadlock. While neither scheme allows a request that leads to deadlock, both have a high runtime cost. These two schemes represent two ends of a spectrum for resource allocation schemes. Many different schemes exist along the spectrum, each trading off precision of detection with cost of detection.

8.4.2.1 Banker's Algorithm

The Banker's algorithm [Dij65, pp. 105-110] was suggested by Dijkstra, and was the first attempt to construct an algorithm for a runtime system to detect pending deadlock. The algorithm checks for a safe execution sequence that precludes hold and wait. As mentioned, precluding hold and wait means not starting execution unless all needed shared resources are available. This scheme ensures that sufficient resources are always available for a task to obtain its worst case resource allocation but performs the check incrementally rather than statically. The main requirement of this approach is that each task know its maximum resource requirements before starting execution, and not exceed the maximum during execution. As mentioned, this is impractical in many cases, as tasks may not know the maximum resource requirements before starting execution nor does the maximum represent the average allocation needs. Another drawback of the algorithm is that it is conservative (i.e., not precise) with respect to avoiding deadlock, but it never allows a request that does lead to deadlock.

The Banker's algorithm is best explained through an example. Imagine a system with 3 tasks and 4 kinds of resources. The number of total instances of each kind of resource are:

total available resources (TR)

R_1	R_2	R_3	R_4
6	12	4	2

i.e., there are 6 R_1 instances available for allocation by tasks, 12 R_2 available for allocation, and so on. As well, each task has the following maximum resource needs and cannot exceed them during execution:

	R_1	R_2	R_3	R_4	
T_1	4	10	1	1	maximum (M)
T_2	2	4	1	2	needed for
T_3	5	9	0	1	execution

Here, task T_1 requires, in its worst case scenario, 4 R_1, 10 R_2, 1 R_3, 1 R_4. In general, it may never come close to this worst case scenario during execution, but in rare circumstances it could reach these levels. Also, this matrix is usually sparse because, unlike this example, few tasks use all the different resources in a system.

Now pretend the system has been running for a while and some resources have already been given out, so the resource allocator has the following table of currently allocated resources:

	R_1	R_2	R_3	R_4	
T_1	2	5	1	0	currently (C)
T_2	1	2	1	0	allocated
T_3	1	2	0	0	

Here, task T_1 is currently allocated (holding), 2 R_1, 5 R_2, 1 R_3, 0 R_4. As can be seen, none of the tasks are close to their worst case scenario in *all* resource categories.

Now a resource request is made by T_1 for another R_1, attempting to increase the number of R_1 resources it is holding from 2 to 3. The resource allocator must decide if granting this request is safe for the entire system with respect to deadlock. Therefore, before it gives out the resource, the resource allocator must ensure no hold and wait can occur for all or a subset(s) of tasks assuming each jumps to its worst case allocation. To accomplish this check, the resource allocator performs a forward simulation, *pretending* to execute the tasks to completion (at which point the task would release its allocated resources) and *pretending* each jumps to its worst case allocation during the execution. It starts by changing the number of R_1s allocated to T_1 from 2 to 3 in matrix C; all the following calculations assume this change. If it can be shown each task can successfully execute to completion with its worst case allocation, then there is a safe execution sequence that ensures no hold and wait. But if no such sequence can be constructed, it does not imply the system will deadlock for the resource request because the calculated worst case scenario will probably not occur. Therefore, the Banker's algorithm is conservative as it might deny the resource request even though deadlock would not result from granting the request; on the other hand, it never grants a request that results in deadlock.

To simplify the calculation, the resource allocator builds or maintains a matrix that is the amount each task would need to increase from its current allocated resources to its maximum.

	R_1	R_2	R_3	R_4	
T_1	1	5	0	1	needed (N) to
T_2	1	2	0	2	achieve maximum
T_3	4	7	0	1	(N = M - C)

Here, task T_1 needs 1 R_1, 5 R_2, 0 R_3, 1 R_4 to jump from its current allocation to its worst case scenario. (Remember, the number of currently allocated R_1 resources has been increased to 3.) This matrix is just the difference between the maximum matrix (M) and the current allocation matrix (C).

To determine if there is a safe sequence of execution, a series of pretend allocations, executions and frees are performed. Starting with the total resources, the first step is to subtract the currently allocated resources to find the currently available resources. This step is accomplished by adding the columns of the currently allocated matrix, C, and subtracting from the total available resources:

current available resources (AR)

1 3 2 2 $(AR = TR - \sum C_{cols})$

Given the available resources, AR, look through the rows of the need matrix, N, to determine if there are sufficient resources for any task to achieve its worst case allocation. Only task T_2 (row 2 of N) has values in each column less than or equal to the available resources. Therefore, it is the only task that could be started and allowed to jump to its worst case scenario without possibly encountering a hold and wait. Now pretend to run task T_2, and assume it does jump to its worst case scenario, so the available resources drop by the amount in the row of matrix N for T_2. Then pretend T_2 completes execution, at which time it releases all the resources it is holding so the available resources increase by the amount in the row of matrix M for T_2. These two steps occur as follows:

$$\begin{array}{c|cccc|l} \mathrm{T}_2 & 0 & 1 & 2 & 0 & (\mathrm{AR} = \mathrm{AR} - \mathrm{N}_{T_2}) \\ \hline & 2 & 5 & 3 & 2 & (\mathrm{AR} = \mathrm{AR} + \mathrm{M}_{T_2}) \end{array}$$

where the first line is the decrease in available resources when T_2 jumps to its maximum, and the second line is the increase after T_2 terminates and releases its held resources. This step is now repeated, except T_2 is finished so its row is ignored in any of the matrices.

Given the newly available resources, look through the rows of the need matrix, N, (excluding row 2) to determine if there are sufficient resources for any task to achieve its worst case allocation. Only task T_1 has values in each column less than or equal to the available resources. Therefore, it is the only task that could be started and allowed to jump to its worst case scenario without possibly encountering a hold and wait. Now pretend to run task T_1, and assume it does jump to its worst case scenario, so the available resources drop by the amount in the row of matrix N for T_1. Then pretend T_1 completes execution, at which time it releases all the resources it is holding so the available resources increase by the amount in the row of matrix M for T_1. These two steps occur as follows:

$$\begin{array}{c|cccc|l} \mathrm{T}_1 & 1 & 0 & 3 & 1 & (\mathrm{AR} = \mathrm{AR} - \mathrm{N}_{T_1}) \\ \hline & 5 & 10 & 4 & 2 & (\mathrm{AR} = \mathrm{AR} + \mathrm{M}_{T_1}) \end{array}$$

where the first line is the decrease in available resources when T_1 jumps to its maximum, and the second line is the increase after T_1 terminates and releases its held resources.

Repeating this step for the remaining task, T_3, shows it could be started and allowed to jump to its worst case scenario without possibly encountering a hold and wait. Pretending to run task T_3, assuming it jumps to its worst case scenario, and releasing its held resources on completion generates:

$$\begin{array}{c|cccc|l} \mathrm{T}_3 & 1 & 3 & 4 & 1 & (\mathrm{AR} = \mathrm{AR} - \mathrm{N}_{T_3}) \\ \hline & 6 & 12 & 4 & 2 & (\mathrm{AR} = \mathrm{AR} + \mathrm{M}_{T_3}) \end{array}$$

The check for correct execution of the Banker's algorithm is that AR is now equal to TR, which follows from the fact that there are no tasks executing so all the resources are available. (If AR does not equal TR, one (or more) of the tasks has walked off with some resources.) Therefore, the Banker's algorithm has demonstrated a safe execution order exists, T_2, T_1, T_3, where giving a new resource R_1 to T_1 does not result in a hold and wait even in the worst case scenario for all tasks in the system.

This particular example behaved nicely because at each step there is only one task that could be allowed to jump to its worst case scenario without performing a hold and wait. What if there is a choice of tasks for execution? It turns out it does not matter which task is selected as all possible selections produce the same result. For example, if T1 or T3 could go to their maximum with the current resources, then choose either as a safe order exists starting with T1 if and only if a safe order exists starting with T3. The proof is intuitive because no matter which task goes first, the other task subsequently starts it turn with the same or more resources available never less, and both tasks can execute with the current available resources.

If a safe ordering is found, does that imply task scheduling is adjusted to the safe order sequence discovered by the algorithm? No, tasks can continue to execute in arbitrary order and speed because the safe ordering implies there cannot be a hold and wait resulting in deadlock with the currently allocated resources. Furthermore, if the next allocation request is from the starting task specified by the previous safe order sequence, the resource can be granted without computing a new sequence because the previous calculation already showed this task can reach its worst case without problems. If any other task requests a resource, the banker's algorithm must be performed again to determine if a safe execution sequence exists. Therefore, the safe execution ordering generated by the Banker's algorithm does not imply tasks must subsequently execute in that order.

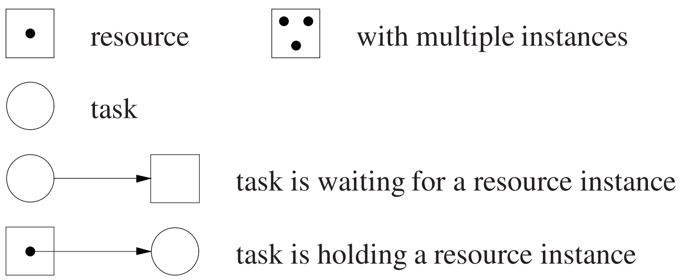
The execution cost of the Banker's algorithm is somewhat high. For each task, $O(RT)$ comparisons are required to find the next task in the safe execution order, where R is the number of resources and T is the number of tasks;

therefore $O(RT^2)$ comparisons are needed for all T tasks to be safely scheduled. $O(R)$ arithmetic operations are needed to update vector AR for each task chosen, for a total of $O(RT)$; another $O(RT)$ operations are needed to compute the initial state of AR. The storage cost is $O(RT)$ for matrices M, C, and N, which dominates the additional cost of vector AR.

8.4.2.2 Allocation Graph

The allocation graph was suggested by Ric Holt [Hol72] and involves graphing task and resource usage at each resource allocation. This graph is an exact representation of the system resource allocations, and therefore, is an exact model of the system. With this model, it is possible to determine precisely if an allocation request results in deadlock.

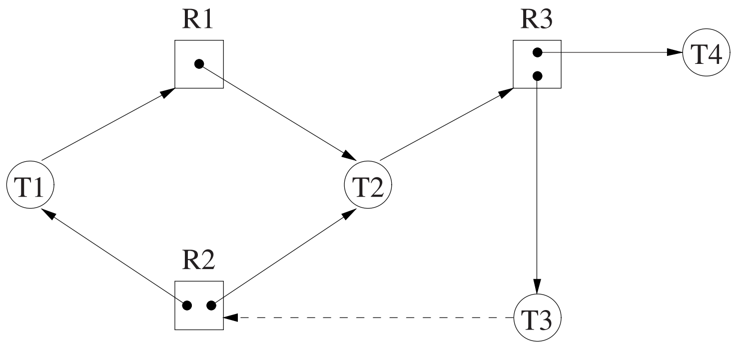
The allocation graph is defined with the following nodes and arcs:



A resource is denoted by a square; each instance of the resource is represented by a dot in the square. A task is represented by a circle. A directed arc from a task (circle) to a resource (square) indicates that the task is waiting for the resource to become free. A directed arc from a resource instance (dot in a square) to a task (circle) indicates the resource instance is being held (used) by the task.

The reason a resource may have multiple instances, e.g., a tape drive resource might have 3 drives, is so a task can request a generic tape drive when the drives provide identical capabilities. A generic request for a tape drive, instead of a specific request for a particular tape drive, prevents waiting if the specific drive is in use but other equally usable drives are available; hence, a generic request enhances concurrency and the program is simplified.

The following allocation graph shows an instant in time during the execution of a system:



There are 4 tasks and 3 resources. T1 is holding an R2 and waiting for an R1. T2 is holding an R1 and R2, and waiting for an R3. T3 is holding an R3 and attempting to acquire an R2 (dashed line). T4 is holding an R3. For the moment, ignore T4 and the resource it is holding.

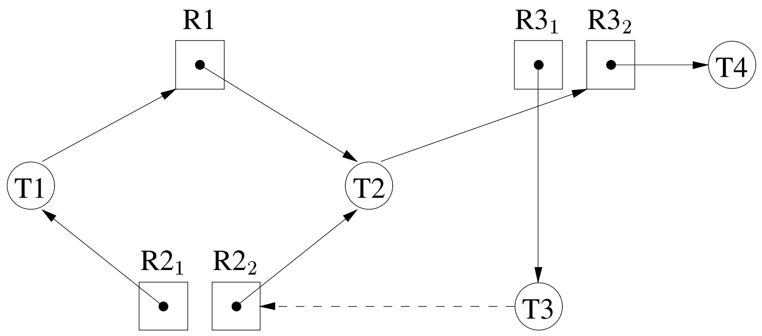
The resource allocator must decide if the request by T3 for an R2 will result in deadlock. To do this, it *pretends* to let T3 block waiting for an R2, because both instances are currently in use, by drawing the dashed line into the graph. The resource allocator now looks for cycles in the graph; if the graph (excluding T4) contains no cycles, the system is not deadlocked. This result follows from the 5 conditions for deadlock discussed previously. A cursory examination of the graph results in two cycles:

$$\text{T1} \rightarrow \text{R1} \rightarrow \text{T2} \rightarrow \text{R3} \rightarrow \text{T3} \rightarrow \text{R2} \rightarrow \text{T1}$$
$$\text{T2} \rightarrow \text{R3} \rightarrow \text{T3} \rightarrow \text{R2} \rightarrow \text{T2}$$

Trace through the graph to locate the two cycles. Therefore, the resource allocator would deny the request for an R2 by T3.

Note, when T3 receives the denial, it can neither busy wait nor block. If T3 busy waits for an R2 to become free, it becomes live-locked because none of the other tasks can make progress until T3 releases the R3 it is holding, but T3 will never release the R3 because the event it is spinning for cannot occur. Therefore, the system is in an undetectable mutual-exclusion deadlock because a cycle has logically been created by the live-blocking (spinning) of T3. It is undetectable because the system cannot determine that T3 is not making progress as it appears to be using CPU time (busy waiting). If T3 blocks for an R2 to become free, e.g., on a separate semaphore, it becomes synchronization deadlocked because none of the other tasks can make progress until T3 releases the R3 it is holding so none of these tasks can unblock the waiting T3. Therefore, the system is in a mutual-exclusion deadlock because a cycle has logically been created by the blocking of T3. Notice that different kinds of concurrency errors are occurring in this discussion depending on the perspective of task T3 or of the system.

Now re-introduce T4 and its resource into the graph. With this arc in the graph, there is no longer a deadlock, even with the cycles in the graph, because T4 eventually releases the R3 it is holding, which breaks the cycles because the arc between T2 and R3 changes direction. The reason the graph indicated a false deadlock is because the resources have multiple instances. If an isomorphic graph is created, where each resource has only one instance, as in:



both cycles can be made to disappear by having T2 wait for R3$_2$, and hence, it is possible to allow T3 to block waiting for an R2. However, building this isomorphic graph is non-trivial; the decision to have T2 wait on R3$_2$, instead of R3$_1$ to break the cycle, is a difficult selection. Basically for each resource with multiple instances, the instances are separated, and for each task waiting for these resources, it can now wait for any of the separated instances; an edge from a process to a resource now has multiple vertices it can point to, and each arrangement of edges must be examined for cycles. Regardless of how the isomorphic graph is generated, a cycle now implies a deadlock; it a graph is found without a cycle, there is no deadlock. General cycle detection involves a depth-first search, which is $O(N+M)$ where N is the number of nodes and M is the number of edges. For the deadlock graphs, the number of nodes is $R+T$ and the number of edges is RT if each task is waiting for each resource and one task has all the resources. Therefore, cycle detection for the isomorphic graph is dominated by the $O(RT)$ term. One way to construct the isomorphic graph is to restrict allocation of resources to one unit at a time, which has the property of at most one request edge from any task node. This restriction can be imposed on the user or implemented implicitly by the resource allocator, i.e., a request for N resources is processed as N separate requests. To check for deadlock, cycle detection has to be perform for each allocated unit of resource versus an allocation of multiple units by a single request and then performing a single deadlock check.

An alternative approach to locating deadlocks directly from the original allocation graph and handling multiple units in a single request is a technique called **graph reduction** [Hol72, pp. 188–189]. The approach is the same as the Banker's algorithm, where the resource allocator performs a forward simulation, *pretending* to execute and free resources to eliminate those parts of the graph where tasks are clearly not deadlocked (see Figure 8.2); if any tasks remain after all non-deadlock ones are removed, there must a deadlock. Starting with the graph on page 243 with the request by T3 for an R2, the algorithm searches the graph for a task that is not blocked, such as T4 (it has no directed arc from itself to a resource). Like the Banker's algorithm, the graph reduction algorithm pretends T4 completes and releases its resources, producing Figure 8.2(a). Because there is now a free R3, it is possible to grant the allocation request of T2 for an R3, which unblocks T2. It is now possible to pretend T2 completes and releases its resources, producing Figure 8.2(b). There is now a choice of unblocked tasks as either T1 or T3 can acquire resources R1 and R2, respectively. Like the Banker's algorithm, when there is a choice, it does not matter which is chosen as the algorithm

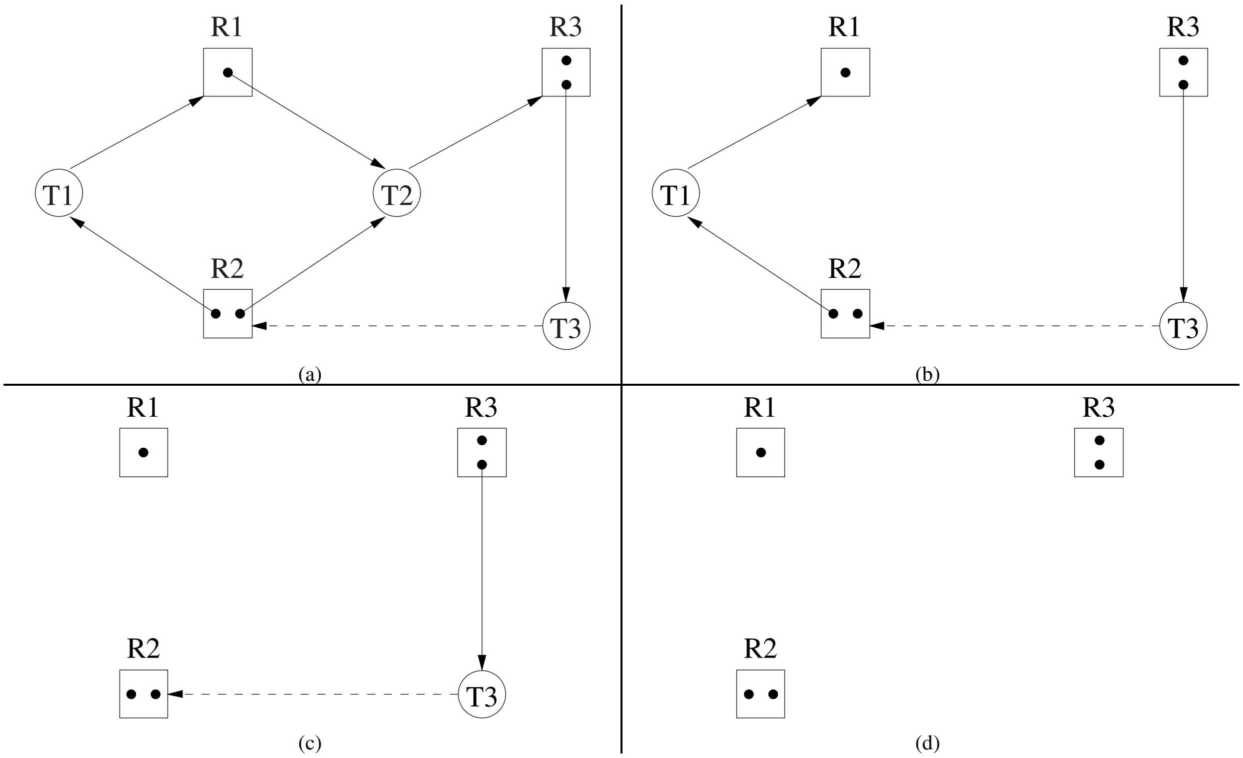


Figure 8.2: Deadlock Graph Reduction

produces the same reduced graph. Choosing task T1, it is now possible to pretend T1 completes and releases its resources, producing Figure 8.2(c). Finally, it is now possible to pretend T3 completes and releases its resources, producing Figure 8.2(d). As can be seen, there are no remaining tasks in the reduced graph, which implies there is no deadlock. Furthermore, like the check for correct execution of the Banker's algorithm, all the resources are now available.

The straightforward algorithm for graph reduction is similar in cost to the Banker's algorithm. Depending on the implementation approach, e.g., an adjacency matrix or adjacency lists, there are $O(RT)$ or $O(T)$ comparison operations to find an unblocked task, respectively. Once an unblocked task is located, it requires $O(R)$ list operations (assuming a task is waiting for every resource) to release its resources and assign them to any task waiting on each released resource (assuming there are links from resources to waiting tasks). Since there are T tasks, the total is $O(R^2T^2)$ or $O(RT^2)$ operations depending on the implementation approach; the cost increases by another search (at least $O(R)$) if there are no links from resources to waiting tasks. The storage cost is $O(RT)$.

Holt [Hol72, pp. 193–194] showed it is possible to reduce the computation cost to $O(RT)$ comparisons and arithmetic operations, if care is taken in the implementation. For each process, a count is maintained of the resources it is waiting for; for each resource, a list of requesting processes is maintained in order by amount requested. Finally, a list of unblocked tasks is maintained. When a reduction occurs (the pretend execution of a task and release of its allocated resources), the wait counts and resource lists are updated; if a wait count reaches zero, the task is added to the unblocked list, which eliminates the search for an unblocked task. Each edge in the graph is examined at most once in updating wait counts, and there are at most $O(RT)$ edges. The amount of other work required to do a single reduction is $O(R)$, and thus $O(RT)$ overall. The amount of space required to store the graph is $O(RT)$, which dominates the cost of the other lists.

8.5 Deadlock Detection and Recovery

The final approach for dealing with deadlock is to just let it happen, check for it occasionally, and use pre-emption to recover resources to break the deadlock(s). Clearly, this is a dynamic approach, and it requires the ability to detect deadlock and preempt resources.

As seen in the previous discussion, discovering deadlock is not easy, has a non-trivial administration cost, and requires a reasonable amount of cooperation among all executing tasks so the resource allocator always knows which tasks own what resources (imagine a task giving an allocated resource to another task and not telling the resource allocator). In effect, detecting deadlock requires constructing an allocation graph and performing a graph reduction. How does this differ from deadlock avoidance? Instead of checking for deadlock on each resource allocation, it is possible to check for deadlock much less frequently. For example, only check for deadlock(s) every T seconds or every time a resource cannot be immediately allocated. The latter choice is reasonable because most resources can be granted immediately, and hence, do not involve a blocking wait. If a resource cannot be granted immediately, either the system is busy or a deadlock has occurred and tasks are holding resources. In this situation, the system (or part of it) may lock up for T seconds until the deadlock is discovered and dealt with. For people, this kind of delay in discovering deadlock is usually not a problem if it occurs infrequently. However, for time-critical situations, such a delay may be unacceptable.

Finally, assuming the deadlock can be detected with reasonable administrative cost, how does the system recover? Clearly, recovery involves pre-emption of one or more tasks in each deadlock cycle. However, the pre-emption decision is not easy and should prevent starvation by not preempting the same task multiple times. First, given a set of tasks in a deadlock, which and how many of the tasks must be pre-empted? Second, given a selected task(s) to preempt, what form of pre-emption is performed? The form of pre-emption can be kind, by unblocking a task and informing it to release some held resource, or cruel, by killing the task and recovering all of its held resources, and possibly restarting it again at some time in the future. The former requires cooperation by the task to release a held resource (and how does the task decide which resource to release?). The latter has problems because the task may have changed the environment so it is now incorrect or incomplete. For example, the killed task may have gotten half way through updating a file. If the file is left updated, it contains erroneous information; if the task is restarted, it may start updating the file from the beginning instead of the pre-emption point, possibly corrupting the data in the first half of the file. While some resources like a printer cannot be easily pre-empted and subsequently returned, some resources can be pre-empted and returned without affecting the task holding the resource. For example, memory is a resource that can be pre-empted from a task by writing the task's memory out to disk and reading it back in again at a later time. While the task is on disk, the memory can be used by another task; when the task is read back in, it continues execution as if nothing happened. However, even this situation may not always work if the task has time critical components, like periodic reading from a device.

8.6 Summary

Finding and fixing errors in a concurrent program can be a daunting task. The temporal nature of a concurrent program, the non-determinism during execution, the need to synchronize threads, and the requirement for atomic access to shared data introduces concurrency specific errors: race error, live-lock, starvation, and synchronization and mutual-exclusion deadlock. Unlike sequential programming, where most debugging occurs by inserting print statements to determine execution order and data values, concurrent programming cannot always use this technique because the print statements cause probe effects and the amount of output may be enormous due to the rarity of an error. The best approach is always prevention, so algorithms can statically eliminate concurrency errors. However, there is a price to be paid for prevention, either in inhibiting concurrency or under utilizing resources. For specific cases, prevention is a viable approach for reducing or eliminating concurrency errors. For the general case, the cost of prevention is usually unacceptable. The general case, which includes a mixture of different kinds of concurrent programs, is best handled by avoidance, if there is sufficient space and time, or recovery, if the cost of error detection must be kept low. Unfortunately, most concurrent systems provide little or no support for detecting errors nor aid in the debugging process; capabilities like deadlock avoidance or concurrent profilers, event traces, and debuggers are still the exception rather than the rule. The most common solution for most concurrent problems is for the computer operator to monitor the system and kill processes that might be causing problems or reboot the system when that fails. Clearly, rebooting the system is the ultimate pre-emption.

8.7 Questions

1. Use the banker's algorithm to determine whether the following state is safe.

Task	Resource 1		Resource 2		Resource 3		Resource 4	
	Alloc.	Max.	Alloc.	Max.	Alloc.	Max.	Alloc.	Max.
T1	0	1	2	2	1	3	1	1
T2	4	5	2	5	1	2	1	1
T3	1	7	0	2	1	2	1	1
T4	2	4	1	1	0	1	0	0
Total units	9		5		5		4	

Show the "need" matrix, and for each step in the banker's algorithm, show the remaining amount of each available resource both after theoretically granting them to a task and after the task has theoretically finished with them.

2. Sometimes sufficient information about the maximum resource needs of tasks is available to enable deadlock avoidance without considerable runtime overhead. Consider the following situation: A set of N tasks shares M identical resources. Resources may be acquired and released strictly one at a time. Each task uses at least one resource but no task ever needs more than M resources. The sum of the maximum needs for all tasks is strictly less than $N + M$. Show that deadlock cannot occur in this system.

Chapter 9

High-level Concurrency Constructs

Chapter 7, p. 177 on threads and locks ended with the readers and writer problem, which is an important concurrent problem. However, it is clear that the solutions to the readers and writer problem were becoming complex; too complex to feel comfortable about the correctness of the solution without extensive analysis. While coding conventions like split-binary semaphores and baton passing give a formal design approach, the resulting programs are still complex, both to manage and maintain; basically, relying on programmers following coding conventions to assure correctness is a poor approach. As well, there is the subtle problem of properly handling staleness and writing because of the "window" for interruption between releasing the entry semaphore and blocking. In essence, we have arrived at the same impasse that occurred with software solutions for mutual exclusion, that is, the complexity and efficiency of the solution is significantly greater than the complexity of the problem being solved. Therefore, it is worthwhile to search for a different approach to reduce the complexity, as for hardware solutions for mutual exclusion. However, in this case, the programming language, not the hardware, provides the mechanism to simplify the solution.

The conclusion is that explicit locks, such as semaphores, are too low level for concurrent programming.

> We must therefore conclude that semaphores do not enable a compiler to give a programmer the effective assistance in error detection that he should expect from an implementation of a high-level language. [Bri73, p. 239]

Essentially, using explicit locks is the concurrent equivalent of assembler programming. Just as most assembler programming is replaced with programming in a high-level language, explicit locks can be replaced with high-level concurrency constructs in a programming language. The goal is to get the compiler to check for correct usage and follow any complex coding conventions implicitly. The drawback is that language constructs may preclude certain specialized techniques, therefore introducing inefficiency or inhibiting concurrency. For most concurrent programs, these drawbacks are insignificant in comparison to the speed of composition, and subsequent reliability and maintainability of the high-level concurrent program. (The same is true for high-level programming versus assembler programming.) Only very rarely should it be necessary to drop down to explicit locks to apply a specialized technique to achieve maximum speed or concurrency.

As well, concurrency constructs are essential for sound and efficient code generation by a compiler [Buh95]. If a compiler is unaware of concurrency in a program, it may perform valid sequential code optimizations that *invalidate* the concurrent program. For example, in the code fragment:

```
P(sem);
i += 1;
V(sem);
```

a compiler might move the call to V *before* the increment because the variables i and sem appear to be independent. While this optimization is valid for a sequential program, it clearly invalidates a concurrent program. If this optimization is turned off to prevent this problem, many valid optimizations do not occur, reducing the overall efficiency of the program because of a small number of problem situations. High-level constructs delineate all the cases where a compiler can or cannot perform particular optimizations.

This chapter begins to examine high-level concurrent program concepts to help simplify the complexity of writing a concurrent program, and provide for sound and efficient implementation. Subsequent chapters examine more complex concurrent constructs.

9.1 Critical Region

One of the first high-level constructs to be proposed was the **critical region**. It was first suggested and discussed by C. A. R. Hoare [Hoa72], and subsequently extended by Per Brinch Hansen [Bri72, Bri73]; the extended form of Per Brinch Hansen is presented. Two new features were proposed: a declaration qualifier, **shared**, and a new control structure, **region**. The declaration qualifier is used to indicate variables accessed by multiple threads of control, which are shared resources, as in:

```
shared int v;
```

The variable v is an integer accessible by multiple threads of control. Now, all access to a **shared** variable *must* occur in a **region** statement, and within the region, mutual exclusion is guaranteed, as in:

```
// access to v disallowed
region v {
    // access to v allowed
}
// access to v disallowed
```

A simple implementation would be nothing more than creating an implicit semaphore for each **shared** variable, and Ping and Ving on the semaphore at the start and end of the **region** statement, as in:

```
shared int v;       semaphore v_lock;
region v {          P( v_lock );
    . . .
}                   V( v_lock );
```

One additional extension to the critical region is allowing reading of a shared variable *outside* the critical region and modification only *within* a region, e.g.:

```
shared int v;
int i = v;        // reading allowed outside a region
region v {
    v = i;        // writing allowed only in a region
    i = v;        // reading still allowed in a region
}
```

The compiler can easily check this particular semantics, and enforce it. (Notice, this capability is already possible when working with semaphores.) However, it is unclear if this extension provides any significant new capability. In particular, there is the potential problem of reading partially updated information while a task is modifying a shared variable in a region, which can result in obtaining inconsistent values (e.g., a negative bank account balance because the balance temporarily goes negative at one point during updating). Even if the values are consistent, reading a shared variable outside a region is speculative, as the value may change immediately. However, this capability might provide a low-cost mechanism, for example, to statistically chart a shared variable over time, where the chart is a representative sample of the data values. Not having to acquire mutual exclusion to sample (read) the shared variable reduces execution cost and does not inhibit concurrency for other tasks modifying the variable.

While not a significant step forward over semaphores, the critical region does have one compelling advantage: it enforces the entry and exit protocol through the syntax. Forgetting either of the braces for the **region** block results in a compilation error, i.e., syntax error; forgetting either a P or V results in a runtime error, which is much harder to detect, e.g., the race error. Furthermore, the critical region delineates the boundary for certain compiler optimizations, e.g., code cannot be moved into or out of the region block. Thus, the **region** statement is a simple example of a higher-level concurrency construct, and it illustrates the important idea of telling the compiler that the program is, in fact, a concurrent program.

Nesting of critical regions is also possible, as in:

```
shared int x, y;
region x {
    . . .
    region y {
        . . .
    }
    . . .
}
```

where mutually exclusive access is acquired for shared variable x, and subsequently, acquired for shared variable y. The inner critical region could be in a conditional statement, e.g., an **if** statement, and hence, not always executed. Nesting allows acquiring shared resources only when needed, which maximizes the potential for concurrency. However, all forms of nested acquisitions of locks have the potential for deadlock, as in:

```
shared int x, y;
  task1                task2
region x {           region y {
    . . .                . . .
    region y {           region x {
        . . .                . . .
    }                    }
    . . .                . . .
}                    }
```

While the identical problem can be constructed with semaphores, it is (usually) impossible for the compiler to detect the situation and issue a warning or error. On the other hand, the critical region statement gives the compiler the potential to statically check for this deadlock situation if it can examine all code that composes a program.[1]

Even though the critical region has advantages over semaphores, it only handles critical sections; no support is available for synchronization, and hence, communication, whereas P and V can be used for both purposes. Using critical regions for mutual exclusion and semaphores for synchronization is not ideal; these two concepts need to be integrated.

9.2 Conditional Critical Region

To deal with the lack of synchronization capability, the critical region is augmented with a conditional delay capability involving the shared variable. The first form of the **conditional critical-region** was devised by C. A. R. Hoare [Hoa72], and only implemented in a restricted form[2] in the programming language Edison [Bri81, p. 371]:

```
region shared-variable {
    await ( conditional-expression );
    . . .
}
```

To enter a conditional critical-region, it must be simultaneously true that no task is in any critical region for the specified shared variable and the value of the **await** expression for the particular critical region is true. If the **await** expression is false, the task attempting entry is blocked (busy waiting is unacceptable) and the region lock is implicitly released so that other tasks can continue to access the shared variable (otherwise there is synchronization deadlock). A crucial point is that the block and release occur atomically, as for the extended P operation in Section 7.6.6, p. 215, so there is no possibility for staleness problems associated with interruption between these operations.

A good way to illustrate the conditional critical-region is through the bounded buffer example, which requires both mutual exclusion and synchronization.

[1] Aliasing problems may preclude complete checking for nested deadlock situations.

[2] No shared variables were supported; hence the **region** statement had only a conditional expression so only one critical region could be active in a program.

```
shared queue<int> q; // shared queue of integers
    producer task                    consumer task
region q {                         region q {
  await ( ! q.full() );              await ( ! q.empty() );
    // add element to queue            // remove element from queue
}                                  }
```

The left conditional critical-region can only be entered by a producer if there is no other task in either critical region for q, and the queue is not full. The right conditional critical-region can only be entered by a consumer if there is no other task in either critical region for q, and the queue is not empty. As a result, the producer cannot attempt to insert elements into a full buffer, nor can the consumer attempt to remove elements from an empty buffer. Hence, the conditional critical-region supports the necessary non-FIFO servicing of producer and consumer when the buffer is empty or full.

Does this solution handle multiple producers and multiple consumers using the shared queue q? The answer is yes, but it does inhibit concurrency slightly. Recall that the bounded buffer in Section 7.5.2, p. 201 uses different locks to protect simultaneous insertion or removal by producers and consumers, respectively. Therefore, a producer can be inserting at the same time as a consumer is removing, except when the list is empty. However, the critical region ensures that only one task is using the shared queue at a time; therefore, there can never be simultaneous insertions and removals at either ends of the queue. Is this loss of concurrency justified? It largely depends on the problem, but, in general, the loss is justified because of the other gains associated with using high-level constructs. In particular, the time to insert or remove is short so the chance of a simultaneous insert and remove occurring is relatively small, and hence, any delay is very short.

What about more complex problems, like the readers and writer problem? While it is possible to build an atomic increment or a binary/general semaphore using a critical region (see Figure 9.1), and hence construct a solution to the readers and writer problem using the previous solutions, this defeats the purpose of having high-level concurrency constructs. However, the obvious solution for the readers and writer problem using conditional critical-regions does not work:

```
shared struct {
    int rcnt, wcnt;
} rw = { 0, 0 };
   reader task              writer task
region rw {                region rw {
  await ( wcnt == 0 );       await ( rcnt == 0 );
    rcnt += 1;                 wcnt += 1;
    // read                    // write
    rcnt -= 1;                 wcnt -= 1;
}                          }
```

Here, a reader task waits until there is no writer (wcnt = 0), and a writer task waits until there is no reader (rcnt = 0). Furthermore, only one writer can be in the critical region for shared variable rw at a time so writing is serialized. However, the problem is reading the resource in the critical region, because that prevents simultaneous readers; i.e., only one reader can be in the critical region for shared variable rw at a time.

This problem can be solved by moving the reading of the resource operation *outside* the reader critical region by dividing it in two:

```
region rw {                // entry protocol
  await ( wcnt == 0 );
    rcnt += 1;
}
// read
region rw {                // exit protocol
    rcnt -= 1;
}
```

Now a reader only establishes mutual exclusion while manipulating the shared counters, not while reading the resource. This structure is unnecessary for writing because there is at most one writer and that writer can be the only task with mutual exclusion in the critical region. It is assumed there is no potential for starvation of readers, because when a

Atomic Increment	General Semaphore
<pre>int atomicInc(shared int &val) { int temp; region val { temp = val; val += 1; } return temp; }</pre>	<pre>shared struct semaphore { int cnt; }; void P(shared semaphore &s) { region s { await (s.cnt > 0); s.cnt -= 1; } } void V(shared semaphore &s) { region s { s.cnt += 1; } }</pre>

Figure 9.1: Low-level Concurrency Primitives using Critical Region

writer leaves the critical region some fair choice must be made by the critical region implementation between a reader or writer task as both counters wcnt and rcnt are 0. However, this solution has the potential for starvation of writers; a continuous stream of readers keeps rcnt $>$ 0, and hence wcnt == 0, so no writer can enter the writer critical region.

Per Brinch Hansen made a subtle but important modification to the conditional critical-region to solve this problem: allow placement of the synchronization condition anywhere within the critical region. (This modification is similar to the notion of an exit in the middle of a loop.) For example, in:

```
region v {
    s1
  await ( ... );                    // condition in middle of critical region
    s2
}
```

mutual exclusion for v is acquired at the start of the region and *s1* is executed. If the condition of the **await** is true, *s2* is executed and mutual exclusion is released on exit from the region. If the condition of the **await** is false, the task blocks at the **await** and releases mutual exclusion (atomically) so other tasks may enter regions for the shared variable. Only when the condition is made true by another task does the blocked task continue execution; it does so *after* the **await**, executes *s2*, and mutual exclusion is released on exit from the region. Notice, a task does not *restart* the region statement when the **await** condition becomes true, it continues from the **await** clause where it blocked (like continuing after a P operation on a semaphore after it is Ved).

Brinch Hansen then presented the following writer code:

```
region rw {
    wcnt += 1;
  await ( rcnt == 0 );              // condition in middle of critical region
    // write
    wcnt -= 1;
}
```

(Actually, the writer code in [Bri72, p. 577] is more complex and the reason is discussed shortly.) This subtle change allows each writer task to acquire unconditional mutual exclusion up to the point of the condition before possibly blocking, allowing an arriving writer to set the writer counter so that readers know if there are waiting writers. In essence, the code before the condition of the critical region is equivalent to part of the code in the entry protocol of the split-binary semaphore solutions where the entry semaphore allows incrementing counters before blocking. Unfortunately, this writer code gives priority to writer tasks, and hence, can cause starvation of reader tasks, e.g., a continuous stream of writers keeps wcnt $>$ 0, and hence rcnt == 0, so no reader can enter its critical region. It turns out that Brinch Hansen *wanted* this semantics; but it is not what we want.

To build a solution with no starvation and no staleness using conditional critical-regions is more complicated, and requires using tickets, as in Section 6.4.3, p. 163 and suggested again in Section 7.6.6, p. 215. The basic approach is for each reader and writer task to first take a ticket to define arrival order, and then conditionally check if it can proceed by comparing the ticket with a serving value, as in:

```
shared struct {
    int rcnt, wcnt, tickets, serving;
} rw = { 0, 0, 0, 0 };
```

reader task	writer task
`region rw {                 // entry protocol`	`region rw {`
`    int ticket = tickets;`	`    int ticket = tickets;`
`    tickets += 1;`	`    tickets += 1;`
`  await ( ticket == serving && wcnt == 0 );`	`  await ( ticket == serving && rcnt == 0 );`
`    rcnt += 1;`	`    wcnt += 1;`
`    serving += 1;`	`    serving += 1;`
`}`	
`// read`	`    // write`
`region rw {                 // exit protocol`	
`    rcnt -= 1;`	`    wcnt -= 1;`
`}`	`}`

The conditions for the reader and writer task are the conjunction of whether it is their turn and a task of the other kind is not using the resource. Only when both are true can a task proceed. Notice the reader entry protocol increments the serving counter *before* exiting the first **region** so that another reader task may enter. If a writer task has the next ticket, it remains blocked because rcnt $>$ 0, as do all subsequent readers because their ticket values are greater than the writer's. While the writer protocol appears to do the same, mutual exclusion is not released from the **await** until the end of the **region**, so no task accessing the same shared variable may enter the critical region. As a result, once a writer task's condition becomes true, neither a reader nor a writer task can take a ticket nor have its condition become true because of the mutual-exclusion property of a region.

A consequence of this particular solution is that there may be a significant delay before any task may enter the critical region because writing the resource is done with mutual exclusion. Therefore, newly arriving reader and writer tasks must block *outside* the critical region waiting to obtain a ticket. Now if the internal implementation of a critical region does not maintain and service these tasks in FIFO order, there is the potential for staleness or writing. If there is no FIFO ordering, it is possible to compensate by modifying the writer protocol so writing the resource is not done in the critical region (like reading), as in:

```
region rw {                 // entry protocol
    int ticket = tickets;
    tickets += 1;
  await ( ticket == serving && rcnt == 0 );
    wcnt += 1;
}
// write
region rw {                 // exit protocol
    wcnt -= 1;
    serving += 1;
}
```

It is now possible for reader and writer tasks to obtain a ticket while writing is occurring; the expectation is that tasks will only block inside the entry protocol on the await condition because the unconditional code at the start of the entry protocol is so short. Clearly, there is no guarantee this is always true, but it may be as probabilistically correct as such a system can come. (Brinch Hansen used this form for his writer protocol in [Bri72, p. 577] for the same reason.) Notice that it is necessary to move the increment of the serving counter to the writer exit protocol, otherwise if a writer task has the next ticket, it could violate the mutual exclusion because rcnt == 0. By incrementing the serving counter *after* the current write is complete, no progress can be made by the next writer in this scenario even though rcnt == 0.

9.2.1 Critical Region Implementation

While the conditional critical-region is appealing, it virtually precludes any form of direct cooperation between the releaser of the mutual exclusion and the next task to acquire mutual exclusion. Since there are no explicit queues a task can block on, it is impossible for an exiting task to directly select the next task to execute in the critical region. Thus, an exiting task may know precisely which task is next to use the resource, however, it cannot express this information

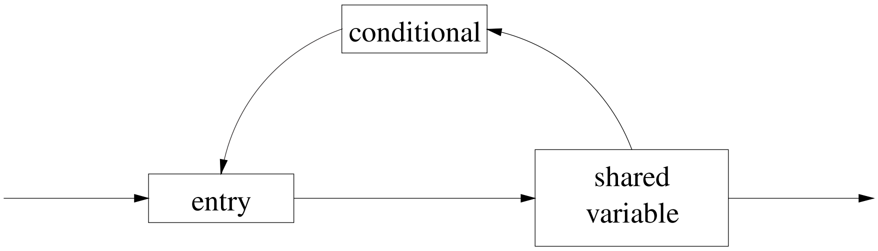


Figure 9.2: Implementation of Conditional Critical Region

to the implementation of the critical region. At best, cooperation can be accomplished indirectly through a mechanism like a ticket; unfortunately, managing tickets is additional complexity, which, in turn, raises the complexity of the solution. Therefore, for problems like the readers and writer, where service order is crucial, the conditional critical-region provides no direct help, which negates the reason of using a high-level concurrency construct.

In addition to this lack of expressibility, the implementation of the conditional critical-region is expensive.

> The main difficulty of achieving an efficient implementation is the reevaluation of synchronizing conditions each time a critical region is completed. [Bri73, p. 243]

The basic implementation of a conditional critical-region is illustrated in Figure 9.2. Each shared variable has associated with it two queues: an **entry queue** for tasks waiting to enter the region for the shared variable and a **conditional queue** for tasks having entered the region but unable to execute because of a false condition. All tasks with a false condition wait on the same conditional queue independent of the conditional expression, i.e., there is only one conditional queue for all the different conditionals for a particular shared variable. When a task leaves the region, either because of a false condition for an **await** or completion of the region code, it may have changed the state associated with the shared variable so that one or more of the conditions for the waiting tasks are now true. Therefore, all tasks on the conditional queue are transferred to the entry queue, and some task on the entry queue (usually the front one) is moved to the ready state to attempt execution of the region. Specifically, each **await**(cond) is translated into:

```
if ( ! cond ) {
    // move all tasks from conditional to entry queue
    do {
        // block on conditional queue
    } while ( ! cond );
}
```

and on exit from the region, all tasks are moved from the conditional to entry queue. Notice, a task waiting on an **await** only moves tasks from the conditional to entry queue once; thereafter, it wakes up inside a repeat loop, checks its condition, and either proceeds or blocks again, otherwise tasks would endlessly cycle between the two queues. (Think about this last point and make sure you understand it before proceeding.)

An alternative implementation, involving cooperation, has the task blocking or leaving the region reevaluate the conditional expressions on behalf of the waiting tasks on the conditional queue to see if any task can enter the region. While this latter approach seems reasonable, it imposes limitations on the form of the conditional expression; in particular, the conditional may only involve the shared variable and constants, otherwise the state of the waiting tasks has to be accessed for local variables. This particular implementation restriction is discussed further in this chapter.

While neither of these implementations are busy waiting, because there is always a bound on the number of tasks to be tested, i.e., the number of tasks on the conditional queue, both are clearly expensive, especially if there are a large number of tasks accessing the critical regions for a particular shared variable, and the conditions are complex. Furthermore, there must be some level of fairness to ensure starvation and possibly staleness cannot occur. For example, after a task leaves a region, tasks waiting on the conditional queue might be moved to the entry queue in FIFO order and *before* tasks that are waiting on the entry queue, so tasks waiting the longest have a greater opportunity of executing first. In all cases, the underlying reason for the complexity, and hence, inefficiency of the conditional critical-region is the lack of direct cooperation; this issue is developed further in this chapter.

9.3 Monitor

While the conditional critical-region was an important first step towards moving concurrency into a programming language to provide a high-level concurrency construct, it fails to provide simple, efficient solutions for medium difficulty concurrent problems, like readers and writer. This point was identified very quickly after introducing the conditional critical-region by its initial creators. As a result, they went back to the drawing board and created the next level of high-level concurrency constructs, this time dealing with the shortcomings of the conditional critical-region.

Interestingly, at the same time work was occurring on the development of high-level concurrency language constructs, the initial work on object-oriented programming was occurring. It was inevitable that interaction occur, and the result was the first object-oriented concurrency construct, called a **monitor**. Again, it was C. A. R. Hoare and Per Brinch Hansen who did the initial work on the development of the monitor:

> The main problem is that the use of critical regions scattered throughout a program makes it difficult to keep track of how a shared variable is used by concurrent processes. It has therefore recently been suggested that one should combine a shared variable and the possible operations on it in a single, syntactic construct called a *monitor*. [Bri73, p. 244]

> Such a collection of associated data and procedures is known as a *monitor*; and a suitable notation can be based on the *class* notation of SIMULA67[3]. [Hoa74, p. 549]

Hence, the basic monitor is an abstract data type combining shared data with serialization of its modification through the operations on it. In μC++, a monitor is an object with mutual exclusion defined by a monitor type that has all the properties of a **class**. The general form of the monitor type is the following:

```
_Monitor monitor-name {
  private:
    . . .           // these members are not visible externally
  protected:
    . . .           // these members are visible to descendants
  public:
    . . .           // these members are visible externally
};
```

In addition, it has an implicit mutual-exclusion property, like a critical region, i.e., only one task at a time can be executing a monitor operation on the shared data. The monitor shared data is normally **private** so the abstraction and mutual-exclusion property ensure serial access to it. However, this simple description does not illustrate the additional synchronization capabilities of the monitor, which are discussed shortly. A consequence of the mutual-exclusion property is that only one member routine can be *active* at a time because that member can read and write the shared data. Therefore, a call to a member of a monitor may block the calling task if there is already a task executing a member of the monitor. Only after the task in the monitor returns from the member routine can the next call occur. μC++ defines a member with this implicit mutual-exclusion property as a **mutex member** (short for mutual exclusion member), and the public members of a monitor are normally mutex members (exceptions are discussed shortly). Similar to coroutines (see Section 4.1.3, p. 67), a monitor is either **active** or **inactive**, depending on whether or not a task is executing a mutex member (versus a task executing a coroutine main). The monitor mutual exclusion is enforced by **locking** the monitor when execution of a mutex member begins and **unlocking** it when the active task voluntarily gives up control of the monitor. In essence, the fields of a shared record have become the member variables of a monitor, and a region statement has become a mutex member, with the addition of parameters.

The following example compares an atomic counter written as a critical region and as a monitor:

[3] SIMULA67 [DMN70] was the first object-oriented programming language.

Critical Region

```
void inc( shared int &counter ) {
    region counter {
        counter += 1;
    }
}
shared int cnt = 0;
inc( cnt );    // atomically increment counter
```

Monitor

```
_Monitor atomicCnt {
    int counter;
  public:
    atomicCnt( int start ) : counter(start) {}
    void inc() {
        counter += 1;
    }
};
atomicCnt cnt(0);
cnt.inc();     // atomically increment counter
```

(In both cases, additional routines are necessary to read the counter and provide other operations.) Notice, a monitor type can generate multiple monitor objects (called monitors) through declarations, just like a coroutine type can generate multiple coroutines. Similarly, it is possible to create multiple instances of a shared type with associated critical regions.

As for a critical region, each monitor has a lock, which is basically Ped on entry to a mutex member and Ved on exit, as in:

```
_Monitor atomicCnt {
    uSemaphore MonitorLock(1);          // implicit code
  public:
    . . .
    void inc(. . .) {
        MonitorLock.P();                // implicit code
        counter += 1;
        MonitorLock.V();                // implicit code
    }
}
```

Because a monitor is like a shared variable, each monitor must have an implicit entry queue on which calling tasks block if the monitor is active (busy waiting is unacceptable). In μC++, arriving tasks wait to enter the monitor on the entry queue, and this queue is maintained in order of arrival of tasks to the monitor (see top of Figure 9.3). When a task exits a mutex member, the next task waiting on the entry queue is made ready so tasks are serviced in FIFO order, and the monitor mutual exclusion is implicitly passed to this task (i.e., the baton is passed) so when it restarts no further checking is necessary.

At this stage, the main difference between the critical region and monitor is strictly software engineering with respect to localizing the code manipulating the shared data. However, if this was the only difference between the critical region and the monitor, there would be little improvement in *expressing* complex concurrency problems. The major advancement of the monitor is the conditional blocking capabilities for synchronization over those of the conditional critical-region so that simple solutions can be constructed for problems like the readers and writer. Most of the remainder of this chapter is devoted to explaining these advanced conditional blocking capabilities.

9.3.1 Mutex Calling Mutex

One property that varies among different kinds of monitors is the ability of a mutex member to call another mutex member. In the simple monitor implementation given above using a MonitorLock semaphore, the first call to the mutex member acquires the lock and a second call by the same task to the same or different mutex member results in mutual-exclusion deadlock of the task with itself. (Why is this not a synchronization deadlock?) Hence, one mutex member cannot call another. However, this restriction can be dealt with by moving the shared code into a no-mutex member and having the mutex members call the no-mutex member, as in:

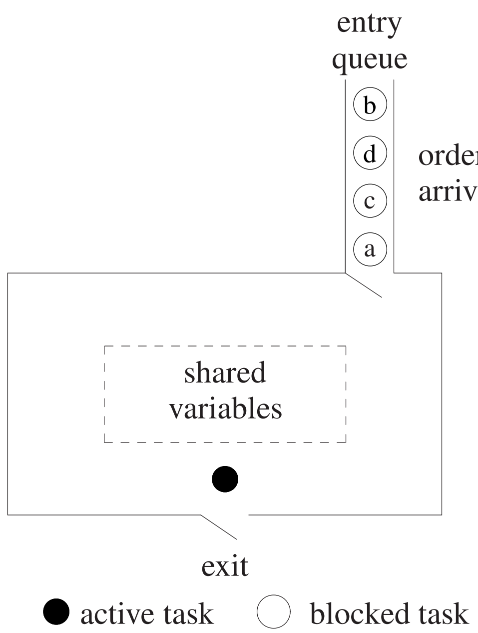


Figure 9.3: Entry Queue

Mutex calling Mutex	Mutex calling No-Mutex
`_Monitor M {` `public:` `mem1(...) { s1 }` `mem2(...) { mem1(); ... }` `}`	`_Monitor M {` `mem3(.. ) { s1 }   // no-mutex, shared code` `public:` `mem1(...) { mem3(...) }   // mutex` `mem2(...) { mem3(...); ... }   // mutex` `}`

Here, mutex member mem2 is calling mutex member mem1 and then performing some additional work. To prevent a deadlock, the code for mem1 is factored into a no-mutex routine, mem3, and mem3 is called from mutex members mem1 and mem2. This restructuring ensures that only one attempt is made to acquire the monitor lock.

While the restructured monitor is strongly equivalent to the original monitor, it introduces additional complexity having nothing to do with the monitor's actual implementation. Therefore, it is more convenient for programmers if the language allows calls from one mutex member to another. μC++ *does* allow the active task in a monitor to call among mutex members without deadlocking. In effect, once a task acquires the monitor lock it can acquire it again, like the owner lock discussed in Section 7.3.1, p. 181. This capability allows a task to call into the mutex member of one monitor, and from that mutex member call into the mutex member of another monitor, and then call back to a mutex member of the first monitor, i.e., form a cycle of calls among mutex members of different monitors. Therefore, in μC++, once a task acquires a monitor, there is no restriction on the use of the monitor's functionality.

9.4 Scheduling

The conditional blocking capabilities of a monitor are referred to as **scheduling**, where tasks schedule themselves by explicitly blocking when execution cannot continue and explicitly unblocking tasks that can execute when the state of the monitor changes (i.e., cooperation). The type of scheduling capability differs with the kind of monitor, and the kind of monitor controls how much and how easy it is to build cooperation. (This issue is discussed in detail in Section 9.11, p. 285.) Notice, scheduling in this context does not imply that tasks are made running; it means a task waits on a queue until it can continue execution and then is made ready. Only when a task waits on the ready queue is it finally scheduled by the CPU and made running. Hence, there can be multiple levels of scheduling occurring in a system, and a task may have to wait at each level before making progress. As a concurrent programmer, it is necessary to write complex scheduling schemes for different shared resources to achieve various effects. For example,

```
template<class ELEMTYPE> _Monitor BoundedBuffer {
      int front, back, count;
      ELEMTYPE Elements[20];
   public:
      BoundedBuffer() : front(0), back(0), count(0) {}
      _Nomutex int query() { return count; }
      void insert( ELEMTYPE elem );
      ELEMTYPE remove();
};
void BoundedBuffer::insert( ELEMTYPE elem ) {
      if ( count == 20 ) _Accept( remove );      // hold calls to insert
      Elements[back] = elem;
      back = ( back + 1 ) % 20;
      count += 1;
}
ELEMTYPE BoundedBuffer::remove() {
      if ( count == 0 ) _Accept( insert );       // hold calls to remove
      ELEMTYPE elem = Elements[front];
      front = ( front + 1 ) % 20;
      count -= 1;
      return elem;
};
```

Figure 9.4: Monitor Bounded Buffer: External Scheduling

for a file containing shared information, access to the file by multiple tasks may be scheduled using the readers and writer algorithm to achieve maximum concurrency among readers and writers of the file. A file can be implemented as a monitor with a simple interface called by tasks for reading and writing data from the file, and within the monitor, the calling tasks are scheduling for reading and writing. More complex scheduling occurs for certain disk scheduling algorithms, where multiple task requests for I/O to different files on a particular disk are ordered from highest to lowest by track number, and serviced in that order. A disk can be implemented as a monitor with a simple interface called by tasks for I/O of data on the disk, and within the monitor, the calling tasks are scheduled from highest to lowest by track number. Notice, there are three levels of scheduling occurring: first, a task is scheduled for reading or writing on the file by the file monitor, then scheduled for I/O on the disk containing the file by the device monitor, and finally, scheduled for access to a CPU to interact with the disk device performing the data transfer by the operating system. Such complex scheduling patterns are common in computer systems.

Two basic techniques for performing scheduling are introduced and discussed in detail: external and internal scheduling. Most high-level concurrency systems support at least one of these techniques, and μC++ supports both. External scheduling schedules tasks outside the monitor and is accomplished with the accept statement. Internal scheduling schedules tasks inside the monitor and is accomplished using wait and signal. A system can exist with only one of the two scheduling schemes, but it will be shown there is only a weak equivalence between the schemes.

9.4.1 External Scheduling

To illustrate external monitor scheduling, a μC++ generic bounded buffer example is presented in Figure 9.4. The basic monitor definition given thus far is sufficient to understand the example; a detailed description of a μC++ monitor is presented later in Section 9.6, p. 271. The buffer implementation is the same as that in Section 7.5.2, p. 201: a fixed sized array with front and back pointers cycling around the array, with an additional counter in this implementation for the number of full slots in the buffer. The semaphore solution implicitly counted the full slots with the full semaphore, but in this solution, the counter must be explicitly managed. The BoundedBuffer interface has four members: the constructor, query, insert and remove. In general, public members of a μC++ monitor are mutex members, and hence, execute mutually exclusively with one another; however, some exceptions are shown.

A monitor's construction appears not to require mutual exclusion. The reason is that, until the monitor is initialized, its declaration is incomplete; only after completion of the declaration can tasks reference the monitor, and hence, make calls to it. Since tasks cannot call the monitor until its declaration is complete, there can be no concurrency during its construction so no mutual exclusion is required. Unfortunately, there is a flaw in this argument. It is possible for a monitor constructor to place the monitor's address into a shared variable *before* its declaration is complete. Hence, a task polling the shared variable could use the address to make a call during construction. (While this is extremely poor programming style, it is possible.) Therefore, μC++ provides mutual exclusion during the initialization phase of a monitor so calls to a monitor's mutex members block until the monitor's declaration is complete.

There is a true exception for providing mutual exclusion to a monitor's public members. The query member for the bounded buffer has the declaration qualifier **_Nomutex**, which explicitly states there is no mutual exclusion on calls to this public member. Therefore, a monitor may be inactive, i.e., no mutex member is executing, but still have tasks executing no-mutex members. Clearly, this capability must be used judiciously to prevent violation of the shared data, and hence, possible race errors. On the other hand, it is useful for enhancing concurrency and other software engineering aspects of a monitor (discussed later). In the case of the bounded buffer, the no-mutex member query only reads the shared data, i.e., the number of elements in the buffer, so it cannot violate the shared data. This same capability was suggested for the critical region (see page 250), where the shared variable can be read outside of the critical region. However, is the value read meaningful? Because there is no mutual exclusion on the call to query, another task can be in the monitor incrementing or decrementing the value of count. Assuming atomic assignment, as for Peterson's algorithm, query always returns a consistent value, either the previous value or the current value of count; hence, query always returns a value in the range 0–20 (buffer size + 1), never a value like -37. However, since the value returned by query can change immediately after it is read, is such a no-mutex member useful? For example, the following code:

```
if ( m.query() < 20 ) m.insert(. . .);
```

does not guarantee the calling task will not block on the call to insert because the value returned by query may change before the call to insert occurs. However, making query a mutex member does not solve this problem; exactly the same problem can occur because there is still a window for another task to race into the monitor between the mutex calls query and insert (albeit a smaller window). Therefore, there is no advantage to making query a mutex member, and doing so only inhibits concurrency to the insert and remove members, which can execute concurrently with query. What good is the query member? As suggested for reading a shared variable outside a critical region, it can be used to statistically sample the state of the buffer and print a graph. If the graph indicates the buffer is constantly empty or full, the buffer can be made smaller or larger, respectively. In other words, exact data is unnecessary to determine trends in behaviour.

The insert and remove members not only require mutual exclusion, but synchronization through conditional mutual exclusion. That is, when the buffer is empty or full, consumer and producer tasks must block, and cooperation should be used to unblock them when the buffer changes. Examine the code for mutex members insert and remove in Figure 9.4. Both start by checking the buffer state to determine if progress can be made, like the **await** statement for the critical region. However, the mechanism to control progress is more direct and allows cooperation. The **_Accept** statement indicates which call is acceptable given the current state of the monitor. If the buffer is full, the only acceptable call is to remove, which frees a buffer slot; all calls to insert, *including* the insert call executing the **_Accept** statement, are held (blocked) until a call to remove occurs. Similarly, if the buffer is empty, the only acceptable call is to insert, which fills a buffer slot; all calls to remove, *including* the remove call executing the **_Accept** statement, are held (blocked) until a call to insert occurs. Unlike the conditional critical-region, when a task in a monitor knows it cannot execute, it does not passively wait for its condition to become true, it actively indicates that its condition can only become true after a call occurs to a particular mutex member, which is a very precise statement about what should happen next in the monitor, and hence, a very precise form of cooperation.

Notice, the **_Accept** statement does not specify the next task to enter the monitor, only the next mutex member to execute when calls occur. Hence, a task is selected indirectly through the mutex member it calls. Notice, also, the **_Accept** statement allows tasks to use the monitor in non-FIFO order, like the conditional of the conditional critical-region. That is, the entry queue may have several producer tasks blocked on it because the buffer is full, but when a consumer arrives its call to remove is accepted ahead of the waiting producers. In the case where a consumer is already waiting on the entry queue when a producer accepts remove, the waiting consumer is immediately removed from the entry queue (regardless of its position in the queue) and allowed direct entry to the monitor. Hence, when the buffer is full, a waiting consumer is processed immediately or a delay occurs until a consumer calls the monitor. Therefore, the monitor implementation for **_Accept** must check if there is any task waiting on the entry queue for the accepted member, which may require a linear search of the entry queue. If no appropriate task is found, the monitor is set up to only accept a call to the accepted member. Because linear search is slow, the monitor implementation usually puts a calling task on two queues: the entry queue, in order of arrival, and a **mutex queue** for the called member,

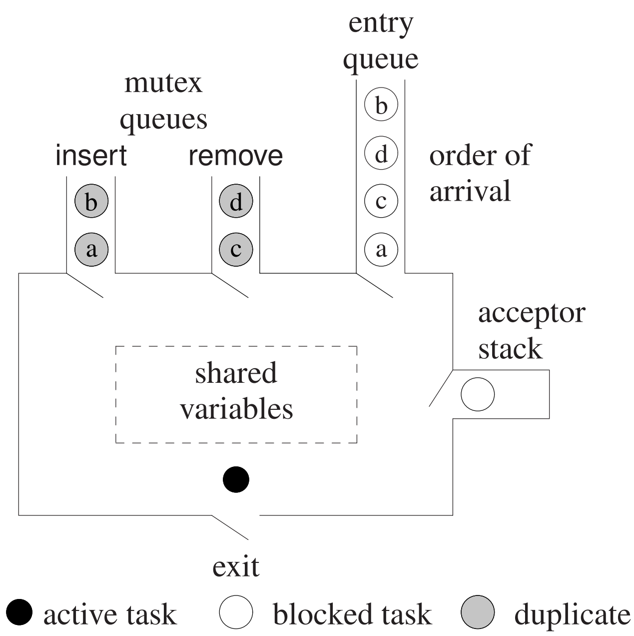


Figure 9.5: External Scheduling: Implicit Queues

also in order of arrival. These queues are illustrated at the top of Figure 9.5, where each task appears on two queues (duplicates are shaded). Hence, it is possible to know without a search if there are tasks waiting to call any particular mutex member. Notice, the entry queue is always necessary for selecting the next task in FIFO order when a task exits the monitor, i.e., returns from a mutex member, but the mutex queues are optional, only enhancing the efficiency of the implementation over searching the entry queue for an accepted caller.

As mentioned, the task executing the **_Accept** blocks, which implicitly releases mutual exclusion on the monitor so a call can proceed to the accepted mutex member. Like the extended P operation in Section 7.6.6, p. 215 and the conditional critical-region, the block and release occur atomically so there is no possibility for staleness problems associated with interruption between these operations. However, where does the accepting task block, i.e., on what queue does it block? In the case of a conditional critical-region, a task executing an **await** statement with a false condition blocks on the shared variable's implicit conditional queue (see Section 9.2.1, p. 254). In the case of a task executing a **_Accept** statement, the task blocks on an implicit stack associated with the monitor, called the **acceptor stack** (see right side of Figure 9.5). Why a stack rather than a queue? The reason is that accepts can be nested, i.e., one task can accept member X, which when called can accept member Y, which when called can accept member Z. When mutex member Z exits, which task should execute next in the monitor? This question can be answered by thinking about cooperation. An accepted task (normally) changes the monitor so the blocked acceptor waiting for this change can continue. If an inappropriate task is scheduled next, the cooperation may be invalidated. Therefore, the most sensible task to execute next is the acceptor, which unblocks knowing that the monitor is in a validate state for it to continue (assuming the accepted member is written correctly with respect to cooperation). Hence, when mutex member Z exits, the task accepting it is unblocked, and when it exits member Y, the task accepting it is unblocked, in member X. This sequence is like routine call/return, where X calls Y calls Z, returning from Z to Y to X. In the case of the routine call sequence, the calls are performed by a single task; in the case of the accept call sequence, the calls are performed by different tasks. To implement this accept/call/unblock sequence requires a stack instead of queue to manage the blocked accepting tasks. In the above scenario, the task performing an accept in X and then the task performing and accept in Y are pushed onto the acceptor stack, and subsequently, popped off in Last-In First-Out (LIFO) order. Unless the task performing Z does a further accept, it never blocks on the acceptor stack, but may have blocked on the entry queue waiting for its call to be accepted.

Finally, notice that the bodies of members insert and remove are defined *outside* of the monitor type. This code organization results from the mutually recursive references in the accept statements between these two members and the C++ *definition before use rule*. If both members are defined in the monitor, one accept statement references the

```
template <class ELEMTYPE> _Monitor BoundedBuffer {
    int front, back, count;
    ELEMTYPE Elements[20];
    uCondition Full, Empty;                // waiting consumers & producers
  public:
    BoundedBuffer() : front(0), back(0), count(0) {}
    _Nomutex int query() { return count; }

    void insert( ELEMTYPE elem ) {
        if ( count == 20 ) Empty.wait();   // block producer
        Elements[back] = elem;
        back = ( back + 1 ) % 20;
        count += 1;
        Full.signal();                     // unblock consumer
    }
    ELEMTYPE remove() {
        if ( count == 0 ) Full.wait();     // block consumer
        ELEMTYPE elem = Elements[front];
        front = ( front + 1 ) % 20;
        count -= 1;
        Empty.signal();                    // unblock producer
        return elem;
    }
};
```

Figure 9.6: Bounded Buffer: Internal Scheduling

other before it is defined, generating a compile-time error. Hence, the member prototypes are defined in the monitor, and the member bodies are defined afterwards, allowing them to reference any of the monitor members.

Several advanced features relating to accepting mutex members, such as accepting multiple mutex-members and private mutex-members are postponed until later.

9.4.2 Internal Scheduling

A complementary approach to external scheduling is internal scheduling. Essentially, external scheduling controls state changes to the monitor by scheduling calls to specified mutex members, which indirectly schedule tasks calling from *outside* the monitor. External scheduling takes advantage of the entry queue to block tasks unconditionally when the monitor is active (i.e., block outside the monitor) and of the acceptor stack to block tasks conditionally that have entered the monitor (i.e., block inside the monitor). Most of the scheduling that occurs and the programmer thinks about is the outside scheduling from the entry queue rather than the internal scheduling on the acceptor stack, which occurs implicitly as part of the accept statement semantics. Internal scheduling turns this around, where most of the scheduling occurs inside the monitor instead of from the entry queue (the entry queue must still exist). To do scheduling *inside* the monitor requires additional queues *inside* the monitor on which tasks can block and subsequently be unblocked by other tasks. In fact, this kind of scheduling is very similar to the synchronization capabilities provided by semaphores.

To illustrate internal monitor scheduling, a μC++ generic bounded buffer example is presented in Figure 9.6. The bounded buffer is essentially the same as the external scheduling version in Figure 9.4, p. 259, except for the mechanism for conditional blocking. Notice the declaration of the two uCondition variables at the top of the bounded buffer. Like a semaphore, a **condition variable** is a queue of blocked tasks, except a condition variable does not have an implicit counter. The lack of a counter means a condition variable does not remember release operations that occur before acquires. In contrast, a V before a P implies the P does not block.

It is common to associate with each condition variable an assertion about the state of the mutex object. For example, in the bounded buffer, a condition variable might be associated with the assertion "waiting for an empty buffer slot". Blocking on that condition variable corresponds to blocking until the condition is satisfied, that is, until an empty buffer slot appears. Correspondingly, a task unblocks a task blocked on that condition variable only when there is an empty buffer slot. However, the association between assertion and condition variable is implicit and not

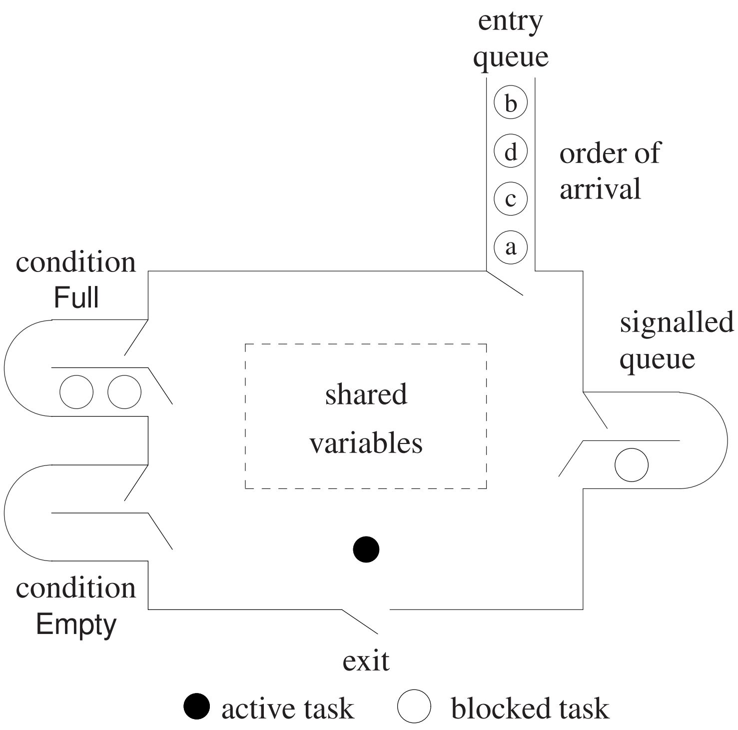


Figure 9.7: Internal Scheduling: Implicit Queues

part of the language; assertions are often used to reason about correctness of cooperation.

Nevertheless, assertions often affect the name given to a condition variable, where naming usually falls into these categories:

1. The condition name identifies a precise event that must be true to fulfill the cooperation of a waiter, e.g.:

   ```
   uCondition Empty;
   ```

   so Empty.wait() means "wait for an empty buffer slot" and Empty.signal() means "there is now an empty buffer slot".

2. Alternatively, the condition name identifies a property of the shared resource that must be true before waiters can continue execution, e.g.:

   ```
   uCondition NonFull;
   ```

   so NonFull.wait() means "wait until the buffer is not full" and NonFull.signal() means "the buffer is not full".

The difference is subtle and largely depends on the perspective of the programmer writing the solution. That is, the programmer thinks from the perspective of the tasks using the resource, or from the perspective of the resource about how tasks use it.

Now examine the start of mutex members insert and remove in Figure 9.6. Both begin by checking the buffer state to determine if progress can be made, like the accept version. However, the mechanism to control progress is even more explicit for internal scheduling. In fact, it is very similar to the semaphore version (see right example of Figure 7.10, p. 200), except the monitor version has an explicit counter, count, because a condition variable has no counter.[4] The lack of an implicit counter also necessitates explicit checking of the counter, which is done implicitly for a semaphore. If progress cannot be made, i.e., the buffer is full or empty, the task in the monitor explicitly executes a wait to block its execution at this time (like a P), which implicitly unlocks the monitor so another task can enter and

[4] In fact, a counter is unnecessary because the state of the buffer, i.e., full or empty, can be determined by appropriate comparisons between front and back. However, these comparisons can be subtle, whereas the counter is straightforward.

change the monitor state. Again, the block and unlock are performed atomically. However, unlike the P operation, which may not block if the semaphore counter is greater than 0, a wait *always* blocks the task executing it. The blocked task relies on cooperation to unblock it at some appropriate time. If a task can make progress, an element is added or removed from the buffer, respectively.

Cooperation occurs as a task exits either mutex member insert or remove. Both routines attempt to wake up a task that is waiting for a specific event to occur. For example, if a producer task has blocked itself on condition Empty waiting for an empty buffer slot, cooperation suggests the next consumer task wake a waiting producer because it has created an empty slot in the buffer. The reverse situation exists when the buffer is empty and a consumer task is waiting for a full buffer slot; the next producer should wake a waiting consumer because it has created a full slot in the buffer. Restarting a blocked task is done with a signal on the specified condition variable, which restarts one (and only one) waiting task (like a V), and both insert and remove end by unconditionally signalling the appropriate condition variable. The signals can be unconditional because of the lack of an implicit counter for condition variables; hence, if the condition has no waiting tasks, the signal does nothing. If the speed of the producer and consumer tasks are reasonably balanced, the buffer is seldom full or empty, and hence, the signals do nothing because tasks are not blocked. (The ideal scenario.)

However, occasionally the buffer fills or empties and tasks must wait. Figure 9.7 shows a snapshot of a monitor with tasks waiting. First, tasks are waiting on the entry queue because there is an active task in the monitor. Second, some tasks have previously entered the monitor and blocked on a condition queue by executing a wait (see left side of Figure 9.7). Finally, the active task in the monitor has signalled one of the tasks from a condition queue. The semantics of the μC++ signal is that the signalled task waits until the signalling task exits the monitor or waits. (Details of other possible semantics are presented in Section 9.11, p. 285.) Interestingly, this semantics appears to be the exact opposite of the accept, which blocks the acceptor until the accepted task exits the monitor or waits. However, an accept is more than than a signal, it is a combination of a wait and signal. The wait component of an accept occurs implicitly and blocks the acceptor on the top of the acceptor stack. The signal component of an accept is performed implicitly when the accepted task finishes a mutex member or waits, i.e., there is an implicit signal of the acceptor task from the top of the acceptor stack. The effect is that the acceptor task waits until the accepted task exits the monitor or waits.

As a result of the signal semantics, the signalled task is removed from the condition queue and must be temporarily stored (like an acceptor) until the signalling task exits or waits. One way of accomplishing this is by moving the signalled task to an implicit queue associated with the monitor, called the **signalled queue** (see right side of Figure 9.7). When the signalling task exits or waits, the task at the front of the signalled queue becomes the active task in the monitor (i.e., the baton is passed from the signaller to the signalled task). Notice, the signalled queue serves a similar purpose to the acceptor stack, i.e., to facilitate the implicit management of tasks in the monitor. If the signalling task does multiple signals, multiple tasks are moved to the signalled queue, and when the signaller exits or waits, the front signalled task becomes the active monitor task, and when it exits or waits, the next signalled task becomes the active monitor task, and so on. When there are no tasks on the signalled queue, a task at the front of the entry queue is selected if one is present. Processing the signalled queue before the entry queue ensures an external task cannot barge ahead of an internal task, which has already waited its turn in the monitor.

As can be seen, internal scheduling is slightly more complex than external scheduling, but simpler than semaphores. It is simpler than semaphores because the monitor implicitly handles all the baton passing necessary when a task waits or exits the monitor to implicitly schedule the next task to enter the monitor. *Finally, there is no legitimate circumstance where a semaphore should be used in a monitor.* If a task blocks on a semaphore inside a monitor, no task can enter the monitor because the task blocks holding the implicit monitor lock. Therefore, unless some task has access to the semaphore from outside the monitor, the monitor is locked forever. Having external access to the semaphore used in the monitor clearly violates the abstraction of the monitor. Only condition variables should be used in a monitor for synchronization because blocking on a condition variable implicitly releases the monitor lock.

9.5 External and Internal Scheduling

As mentioned, external and internal scheduling are complementary techniques, and therefore, can appear together in the same monitor. This section discusses what it means to combine the two techniques, followed by the appropriateness of each technique.

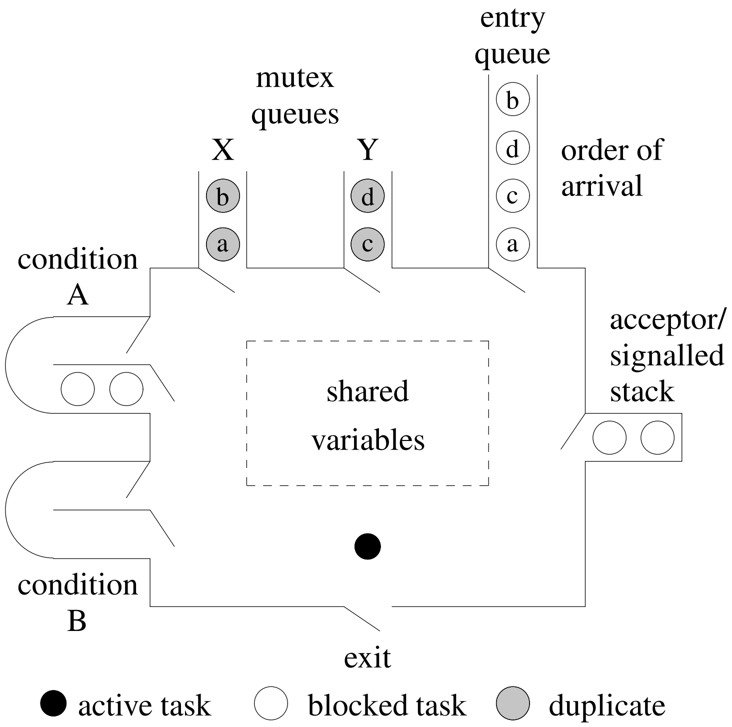


Figure 9.8: μC++ Mutex Object

9.5.1 Combining Scheduling Techniques

Combining external and internal scheduling implies controlling some tasks outside the monitor and others inside the monitor, which means the monitor uses both accept and signal/wait. While the concept seems straightforward, it has ramifications on the way the monitor behaves. Figure 9.8 shows the general structure of a μC++ monitor using a combination of external and internal scheduling. Looking around the figure, there are the external entry queue and mutex queues (top), as well as the internal condition queues (left). However, there is only an acceptor stack (right), not an acceptor stack *and* a signalled queue.

The reason for having only one implicit location for managing both acceptor and signalled tasks is to simplify understanding, and hence, programming of the monitor. If both acceptor stack and signalled queue exist, complexity increases when tasks appeared on both. This scenario occurs when one task accepts another task, and the accepted task signals a task from a condition queue. When the accepted task exits the mutex member, there is now a choice between taking a task from the top of the acceptor stack or the front of the signalled queue. Whatever choice is made, it is implicitly part of the monitor implementation, and therefore, beyond the control of the programmer. If tasks were chosen first from either the stack or queue, e.g., take from the acceptor stack until empty and then take from the signalled queue, there is the potential for starvation and/or staleness problems. If tasks are chosen alternately from the stack and queue to prevent starvation, staleness is still a problem, and internal scheduling within the monitor becomes extremely complex because the programmer must compensate if the monitor selects from an inappropriate structure. As noted in earlier discussions, using a single data structure for two different kinds of tasks (e.g., reader and writer tasks) does not have starvation and staleness problems, and makes programming in the monitor simpler.

Therefore, it is appropriate to place acceptor and signalled tasks on a single data structure, and when a task exits or waits, select an element from this structure, if there are tasks waiting, otherwise select from the entry queue. Should this structure be a stack or a queue? The obvious answer is a queue because temporal ordering is preserved, which ensures no starvation or staleness. Nevertheless, the choice in μC++ is a stack, called the **acceptor/signalled stack**. The reason is to preserve the semantics of the accept statement, which services calls to mutex members in LIFO order. Otherwise, nested accepts restart in reverse order from normal routine calls, which most programmers would find counter intuitive, leading to mistakes. The problem with a stack is the potential for starvation and staleness because tasks can be pushed on the stack and never popped, and temporal order is not maintained. However, unlike the situation where the monitor implementation is implicitly selecting from a stack or a queue and the programmer has no control, here the programmer has complete control and can easily compensate for any problems associated with the implicit acceptor/signalled stack. In fact, the only anomaly occurs during multiple signalling, where multiple signalled tasks are pushed onto the implicit stack and subsequently restarted (popped off) in reverse order from their waiting time on the condition queue. If this semantics is incorrect for an application, it is usually possible to convert multiple signalling to daisy-chain signalling, where each restarted task signals the next task to be restarted. (Both daisy-chain signalling and multiple signalling are discussed shortly). Resolving the conflict between the semantics of accept and signal/wait is an example of a design trade off between interacting language constructs.

In summary, there are two kinds of scheduling: external and internal. External scheduling blocks the acceptor and other inappropriate tasks by only accepting calls from tasks that can legitimately execute given the current state of the monitor. When the active monitor task blocks on an accept statement, the next active task is chosen from the specified mutex queue. Because nested accepts are possible, acceptors block on an implicit stack associated with each monitor until the accepted call completes; therefore, nested accept calls behave like nested routine calls, except the calls are executed by different threads. Internal scheduling blocks tasks on explicit condition queues internal to the monitor, when the state of the monitor is inappropriate. Specifically, when the active monitor task blocks on a wait or exits the monitor, the next active task is selected first from the acceptor/signalled stack and then from the entry queue. Unlike blocked acceptor tasks, which are implicitly restarted, tasks blocked on condition variables are explicitly restarted via signalling. A signalled task does not restart until *after* the signaller task exits or waits. Both acceptor and signalled tasks block on the same implicit stack in the monitor. Because a stack is necessary to preserve accept semantics, care must be taken with multiple signalling to prevent starvation and/or staleness.

9.5.2 Scheduling Selection

Given the two approaches to scheduling, external and internal, what guidelines can a programmer use to select the most appropriate approach? It turns out there is only a weak equivalence between the two scheduling schemes. As a result, there are situations where only one scheduling approach is appropriate.

In general, external scheduling is easier to understand and code because it is only necessary to specify the next mutex member to execute and the monitor implicitly handles all the details. Specifying the next mutex member to execute is a reasonably direct and powerful way to control synchronization among tasks using a monitor. From an implementation standpoint, external scheduling can be more efficient in specialized circumstances, e.g., when the monitor is heavily used. During heavy usage, tasks block on the entry queue because the monitor is active. With internal scheduling, tasks must unblock from the entry queue and possibly block again on a condition queue because the monitor state is currently inappropriate. In this case, a task blocks and unblocks twice (**double blocking**) before completion of a mutex member. With external scheduling, it may be possible for some tasks to block only on the entry queue, and hence, save some overhead costs. An example where double blocking is not eliminated is the bounded buffer using external scheduling (see Figure 9.4, p. 259). When the buffer is full, a perfect interleaving of producers and consumers can result in producers waiting on the entry queue and then on the acceptor stack until the next consumer removes an element. Hence, double blocking can occur with both external and internal scheduling; however, in some situations external scheduling experiences less double blocking than internal scheduling.

If external scheduling is simpler than internal scheduling, why not use it all the time? Unfortunately, the following situations cannot be handled adequately by external scheduling:

9.5.2.1 Member Parameter Scheduling

The best way to understand this situation is to note that in a normal routine call, the calling arguments are inaccessible until *after* the routine begins execution. Only in the special case of a call to a mutex member, which may block the caller, is it reasonable for a task in the monitor to consider examining the calling arguments of the tasks blocked on a mutex queue. Since the calling tasks are blocked, it is safe to read the arguments as they cannot change. Why is this capability desirable? Imagine servicing monitor requests in decreasing order of the argument value, e.g., if the argument is age, service the youngest person first. (The issue of starvation is ignored to simplify the discussion, but it only increases the complexity of these kinds of scheduling schemes.) To provide such a capability requires adding a new feature to the accept statement to search a mutex queue and examine the arguments of tasks blocked on the queue, e.g.:

```
_Accept( mem( age ) where age is the smallest );
```

This capability is clearly peculiar to mutex members and is not an intuitive extension of the normal routine call mechanism. Nor is it clear how to specify the conditional selection, e.g., determining which age is the smallest. (Attempts at providing this form of conditional selection are presented in Section 13.4.2, p. 398.) One possible solution to this problem, without extending the accept statement, is to have a public member for each age (0–130), and the requester must call the appropriate member. (To accomplish this requires using a more advanced feature of the accept statement, e.g., accepting multiple mutex members, which is discussed shortly.) However, enumerating the necessary members, selecting the members, and making the appropriate call, is unacceptable because the amount of program text is proportional to the range of the parameter value. For example, a parameter range of 1,000,000 requires 1,000,000 accept clauses, and in most cases, the range is only sparsely populated. Furthermore, this situation is exacerbated if the criteria for scheduling depends on the arguments from two or more different mutex queues. Imagine a situation where two different kinds of tasks call two different mutex members both passing an argument used in scheduling. To decide which task to select requires searching both mutex queues, and it is unclear how such a complex mechanism can be provided.

Internal scheduling provides a solution to this scenario without requiring language extensions. Instead of the active task attempting to access task arguments outside the monitor on the mutex queues, internal scheduling has each task begin its mutex member, where the parameters now represent the calling arguments, and now the decision to block or not block on a condition variable can be made by examining both the parameters and the monitor state using standard language control structures. Clearly, the style of cooperation may change significantly between these alternatives.

Figure 9.9 shows two versions of an interesting problem illustrating the need to access parameter information from multiple mutex queues. The problem is a simple dating service, where girls and boys call a corresponding mutex member with a phone number and a compatibility code (0–19). When a client calls with a matching compatibility code, the phone numbers of the pair are exchanged so they can arrange a date. A girl or boy blocks in the monitor until a matching date appears. Parameter information is needed from both the Boy and Girl mutex queues to construct a match *and* to return the phone numbers. With internal scheduling, the parameter information is received and stored in the monitor where it can be manipulated as necessary to solve the problem.

The left version in Figure 9.9 is discussed first. An array of conditions is used for blocking clients of the same gender with the same compatibility until a date arrives. Since there are 20 compatibility codes, the array of conditions is dimensioned to 20. When a client arrives, it checks the opposite array at the position with the same compatibility code, and if that condition queue is empty, it blocks on its gender array at the position denoted by its compatibility code. If there is a date waiting on the appropriate condition queue, the phone number is copied into a global monitor communication variable, and the date is signalled. However, there is a problem because the signaller needs to wait for the date to unblock and copy its phone number from a local variable into a global monitor communication variable to get the date's phone number. The problem is that the signaller continues to execute after the signal, and the signalled task only unblocks *after* the signaller exits or waits. Since the signaller cannot exit without a phone number, it has to wait. (If it does exit immediately, the phone number from the previous date is erroneously returned.) Therefore, there is an additional condition variable, Exchange, used just for the necessary communication in the monitor to exchange phone numbers (see page 274 for a mechanism to eliminate condition Exchange). The task blocks on the exchange condition, and the next task to become active in the monitor is at the top of the acceptor/signalled stack, which is the signalled date. The signalled date unblocks after the wait on the array of conditions, copies its phone number into the appropriate global monitor communication variable, signals its date on the Exchange condition, and exits with the appropriate date's phone number. The next task to become active in the monitor is the one at the top of the acceptor/signaller stack, which is the signalled date. It unblocks after the wait on Exchange, and exits with the appropriate date's phone number.

An alternative approach, which reduces the amount of internal blocking for exchanging phone numbers, is to store the compatibility codes for all the blocked tasks in a global monitor data structure. Then the task that finds a date can signal the date, copy their phone number from the global data structure, and return, without having to wait for the date to unblock to obtain their phone number from a local variable. However, the data structure to store the compatibility codes is reasonably complex, i.e., an array of queues, because there may be many tasks waiting on each compatibility condition queue. Furthermore, data must be synchronized between the compatibility data structure and the condition queues at all times to correctly reflect the necessary information for each blocked task.

μC++ provides a language feature that simplifies this approach by folding the scheduling data structure into the condition queue (see right version in Figure 9.9). Since a condition queue already builds a queue data structure, it seems wasteful to build another complete queue to store the additional compatibility code information. As well, nodes

Using Exchange Condition

```
_Monitor DatingService {
    uCondition Girls[20], Boys[20];
    uCondition Exchange;
    int GirlPhoneNo, BoyPhoneNo;
  public:
    int Girl( int PhoneNo, int ccode ) {
        if ( Boys[ccode].empty() ) {
            Girls[ccode].wait();
            GirlPhoneNo = PhoneNo;
            Exchange.signal();
        } else {
            GirlPhoneNo = PhoneNo;
            Boys[ccode].signal();
            Exchange.wait();
        }
        return BoyPhoneNo;
    }
    int Boy( int PhoneNo, int ccode ) {
        if ( Girls[ccode].empty() ) {
            Boys[ccode].wait();
            BoyPhoneNo = PhoneNo;
            Exchange.signal();
        } else {
            BoyPhoneNo = PhoneNo;
            Girls[ccode].signal();
            Exchange.wait();
        }
        return GirlPhoneNo;
    }
};
```

Using Condition-Queue Node

```
_Monitor DatingService {
    uCondition Girls[20], Boys[20];

    int GirlPhoneNo, BoyPhoneNo;
  public:
    int Girl( int PhoneNo, int ccode ) {
        if ( Boys[ccode].empty() ) {
            Girls[ccode].wait( PhoneNo );


        } else {
            GirlPhoneNo = PhoneNo;
            BoyPhoneNo = Boys[ccode].front();
            Boys[ccode].signal();
        }
        return BoyPhoneNo;
    }
    int Boy( int PhoneNo, int ccode ) {
        if ( Girls[ccode].empty() ) {
            Boys[ccode].wait( PhoneNo );


        } else {
            BoyPhoneNo = PhoneNo;
            GirlPhoneNo = Girls[ccode].front();
            Girls[ccode].signal();
        }
        return GirlPhoneNo;
    }
};
```

Figure 9.9: Scheduling using Parameter Information: Solution 1

must be added and removed from both data structures at exactly the same times and in the same order (e.g., non-FIFO scheduling of tasks from a condition variable). Therefore, μC++ allows a single integer value to be stored with each blocked task on a condition queue by passing the value to wait (see details in Section 9.6.3, p. 273). This capability can be seen in the right example of Figure 9.9 where each call to wait has the form:

gender[ccode].wait(PhoneNo);

When each task blocks, the value stored in each condition node of the condition queue is a phone number. When a task finds a date, it is sufficient to examine the information stored with the blocked task at the front of the condition queue to obtain a date's phone number, which is accomplished with the statement:

*gender*PhoneNo = *gender*[ccode].front();

which returns the phone number stored with the blocked task on the front of the condition queue (or stored with the next task to be unblocked if non-FIFO scheduling is used). Notice, the date is signalled *after* the phone number is copied from the condition queue, because a signal moves the blocked task from the condition queue to the acceptor/signalled stack, which is inaccessible in a monitor.

Unfortunately, both of the previous solutions make unrealistic assumptions about the compatibility code. The previous approaches are reasonable only when the range of the scheduling values is small, as the storage usage is proportional to the value range. As mentioned, the range of the compatibility code may be large and sparse, which means an array of condition variables the size of the compatibility code is inappropriate in general. Furthermore, the compatibility code may not be a single value but a table of values (e.g., answers to a questionnaire), and only a single integer value can be stored in a μC++ condition-queue node.

Figure 9.10, p. 270 shows a monitor that deals with both these problems using a list data structure and the notion of a **private condition**, similar to a private semaphore approach suggested in Section 7.6.6, p. 215. A list is constructed with nodes containing a task's compatibility code information (or a pointer to it) and a private condition variable. When a task must wait, it blocks on the private condition variable and is the only task to block on that condition variable. This approach uses storage proportional to the number of blocked tasks, which in many cases is significantly less than the range of the scheduling values. Also, the scheduling value can be arbitrarily complex, and the insert and search can also be arbitrarily complex.

Both Girl and Boy members start by searching the opposite list for a compatible date. The search returns a pointer to a Node data structure, which should be a compatible date, or NULL, which means there is no compatible date. If there is no compatible date, the task creates a Node on its execution stack, initialized with its compatibility information, inserts the node on the appropriate waiting list, and blocks on its private condition variable in the node. When a task unblocks, it removes its node from the list, and the remainder of the code is the same as the left example in Figure 9.9, except a date is signalled from the private condition of the found node. A solution based on private conditions (like the private semaphore approach in Section 7.6.6, p. 215) can be constructed for the right example in Figure 9.9, by storing the phone number in the list node along with the compatibility code and a private condition variable.

It might be argued that this problem can be restructured so that tasks call a single member, passing information identifying the kind of task (i.e., girl or boy). However, restructuring is not always possible because of other factors, such as legacy issues. Therefore, concurrency constructs must be able to handle this level of scheduling, and internal scheduling provides the necessary flexibility.

9.5.2.2 Delay Scheduling

This situation arises when a request may need to block (delay) during its execution. Delay scheduling *does not* occur in the external-scheduling bounded-buffer (see Figure 9.4, p. 259), as the cooperation ensures that once a call to insert or remove begins, it can always execute to completion. In contrast, calls to the DatingService may have to block because a partner is unavailable. The delay scheduling could be removed by immediately returning a -1 to indicate no compatible partner is available. However, this approach presents a significant problem to the caller if it must achieve a successful call as its only approach is to busy wait making repeated calls, which is execution-time expensive and can result in starvation. To provide delay scheduling using external scheduling requires a calling task to perform member-parameter scheduling of the opposite gender with a particular compatibility code. For example, if a Boy task calls into the DatingService, it must execute an accept statement like:

_Accept(Girl *with same ccode*);

but this prevents other exchanges from occurring until the appropriate call occurs. Hence, delaying until an appropriate event occurs while continuing to service requests cannot be done solely with external scheduling.

The issue forcing internal scheduling when delay scheduling is needed is that a task can only block once on the entry queue, i.e., it is impossible to go back out of the monitor and wait again on the entry queue. Hence, after a task enters a monitor, any further blocking can only occur with nested accepts, which forces a stack ordering on request processing. While most concurrent languages disallow scheduling from inside a mutex object back outside onto the entry queue, it is possible. This capability might be provided by a statement of the form:

requeue *mutex-member-name*

which blocks the executing task, places it back on the specified mutex queue, and schedules the next task for the monitor. In effect, the original call has been transformed into a new call to the same or another mutex member with possibly different arguments. This requeued call is then accepted again, when the monitor state is appropriate for the call to proceed.

On the surface, requeueing appears equivalent to internal scheduling, but this is not the case. A mutex member working on a request may accumulate complex execution and data state. To requeue a request, the work accomplished thus far and temporary results must be bundled and forwarded to the requeue mutex member handling the next step of the processing, usually by placing this information into global task variables; alternatively, the work is re-computed at the start of the requeue mutex member, if possible. In contrast, waiting on a condition variable automatically saves the execution location and any partially computed state. In fact, simulating internal scheduling with requeue is similar to simulating a coroutine using a routine or class, where the programmer must explicitly handle the execution and/or data state, possibly using global variables. Therefore, there is only a weak equivalence between requeueing and internal scheduling.

```
_Monitor DatingService {
    struct Node : public uSeqable {
        int ccode;
        uCondition block;                                // private condition
        Node( int ccode ) : ccode( ccode ) {}
    };
    Node *Search( uSequence<Node> &gender, int ccode ) {
        // return node with matching compatibility code or NULL
    }
    uSequence<Node> Girls, Boys;
    uCondition Exchange;
    int GirlPhoneNo, BoyPhoneNo;
  public:
    int Girl( int PhoneNo, int ccode ) {
        Node *boy = Search( Boys, ccode );               // check for match
        if ( boy == NULL ) {                             // no match ?
            Node n( ccode );                             // create and add node
            Girls.uAdd( &n );
            n.block.wait();                              // block on private condition
            Girls.remove( &n );
            GirlPhoneNo = PhoneNo;                       // set communication variable
            Exchange.signal();                           // restart date
        } else {
            GirlPhoneNo = PhoneNo;                       // set communication variable
            boy->block.signal();                         // unblock date
            Exchange.wait();                             // wait for date phone number
        }
        return BoyPhoneNo;                               // return date phone number
    }
    int Boy( int PhoneNo, int ccode ) {
        Node *girl = Search( Girls, ccode );             // check for match
        if ( girl == NULL ) {                            // no match ?
            Node n( ccode );                             // create and add node
            Boys.uAdd( &n );
            n.block.wait();                              // block on private condition
            Boys.remove( &n );
            BoyPhoneNo = PhoneNo;                        // set communication variable
            Exchange.signal();                           // restart date
        } else {
            BoyPhoneNo = PhoneNo;                        // set communication variable
            girl->block.signal();                        // unblock date
            Exchange.wait();                             // wait for date phone number
        }
        return GirlPhoneNo;                              // return date phone number
    }
};
```

Figure 9.10: Scheduling using Parameter Information: Solution 2

9.6 Monitor Details

The following μC++ monitor details are presented to understand subsequent examples or to help in solving questions at the end of the chapter.

While there is a **_Nomutex** qualifier, which implies no mutual exclusion, there is also a **_Mutex** qualifier, which implies mutual exclusion. Both qualifiers can qualify **private**, **protected** and **public** member routines. As well, both qualifiers can also qualify a **class** definition:

```
_Mutex class M1 { ...   _Nomutex class M2 { ...
```

When the **_Mutex** qualifier is placed on a **class** definition, it indicates all public member routines have the mutual-exclusion property, unless overridden on specific member routines with the **_Nomutex** qualifier. When the **_Nomutex** qualifier is placed on a **class** definition, it indicates all public member routines have the no-mutual-exclusion property, which is the same as a **class**, unless overridden on specific member routines with the **_Mutex** qualifier. The default qualifier for a class, i.e., if no qualifier is specified, is **_Nomutex** because the mutual-exclusion property for public members is not needed. Therefore, a **class** creates a monitor if and only if it has at least one **_Mutex** member routine, and that **_Mutex** member routine can have any visibility, i.e., **private**, **protected** or **public**. As well, the destructor of a monitor is always mutex, regardless of where it is defined or if it makes no references to the monitor's shared variables. The reason for this requirement is same as that for a task, i.e., the storage for the monitor cannot be deallocated if the monitor is active (a task's thread is executing in the monitor). Hence, a monitor always has one mutex member, its destructor, and a monitor's destructor cannot be qualified with **_Nomutex**, e.g., the following is the monitor with minimum number of mutex members:

```
class M {
  public:
    _Mutex ~M() {}      // only mutex member
};
```

(However, it is doubtful if this trivial case is useful.) The following examples illustrates most of the different possible combinations for implicit and explicit creation of mutex members of a mutex object:

```
_Mutex class M1 {  // => publics implicitly mutex
  private:
    _Mutex int m1();     // => explicit mutex
    int m2();            // => implicit no mutex
  public:
    ~M1();               // => implicit mutex
    int m3();            // => implicit mutex
    _Nomutex int m4();   // => explicit no mutex
}
```

```
_Nomutex class M2 {  // => publics implicitly no mutex
  private:
    _Mutex int m1();     // => explicit mutex
    int m2();            // => implicit no mutex
  public:
    ~M1();               // => implicit mutex
    int m3();            // => implicit no mutex
    _Mutex int m4();     // => explicit mutex
}
```

In fact, the name **_Monitor** used so far is just a preprocessor macro for "**_Mutex class**" defined in include file uC++.h.

9.6.1 Monitor Creation and Destruction

A monitor is the same as a class with respect to creation and destruction, except for a possible delay for the destructing task, as in:

```
_Mutex class M {
  public:
    void r( ... ) ...      // mutex member
};
M *mp;                     // pointer to a M
{ // start a new block
    M m, ma[3];            // local creation
    mp = new M;            // dynamic creation
    ...
} // wait for m, ma[0], ma[1] and ma[2] to terminate and then deallocate
...
delete mp;  // wait for mp's instance to terminate and then deallocate
```

When a monitor is created, the appropriate monitor constructor and any base-class constructors are executed in the normal order by the creating thread. Because a monitor is a mutex object, the execution of its destructor waits until it can gain access to the monitor, just like the other mutex members of the monitor, which can delay the termination of the block containing a monitor or the deletion of a dynamically allocated monitor.

9.6.2 Accept Statement

A **_Accept** statement dynamically chooses the mutex member(s) that executes next, which indirectly controls the next accepted caller, that is, the next caller to the accepted mutex member. The simplest form of the **_Accept** statement is:

```
_Accept( mutex-member-name );
```

with the restriction that constructors, **new**, **delete**, and **_Nomutex** members are excluded from being accepted. The first three member routines are excluded because these routines are essentially part of the implicit memory-management runtime support. **_Nomutex** members are excluded because they contain no code affecting the caller or acceptor with respect to mutual exclusion. The syntax for accepting a mutex operator member, such as operator =, is:

```
_Accept( operator = );
```

Currently, there is no way to accept a particular overloaded member. Instead, when an overloaded member name appears in a **_Accept** statement, calls to any member with that name are accepted. A consequence of this design decision is that once one routine of a set of overloaded routines becomes mutex, all the overloaded routines in that set become mutex members. The rationale is that members with the same name should perform essentially the same function, and therefore, they all should be eligible to accept a call.

When a **_Accept** statement is executed, the acceptor is blocked and pushed on the top of the implicit acceptor/signalled stack and a task is scheduled from the mutex queue for the specified mutex member. If there is no outstanding call to that mutex member, the acceptor is **accept-blocked** until a call is made. The accepted member is then executed like a member routine of a conventional class by the caller's thread. If the caller is expecting a return value, this value is returned using the **return** statement in the member routine. *Notice, an accept statement accepts only one call, regardless of the number of mutex members listed in the statement.* When the caller's thread exits the mutex member or waits, further implicit scheduling occurs. First, a task is unblocked from the acceptor/signalled stack if present, and then from the entry queue. Therefore, in the case of nested accept calls, the execution order between acceptor and caller is stack order, as for a traditional routine call. If there are no waiting tasks present on either data structure, *the next call to any mutex member is implicitly accepted*.

The extended form of the **_Accept** statement is used to accept one of a group of mutex members using a list of mutex member names and/or multiple accept clauses, as in:

```
_Accept( mutex-member-name-list )
        statement                       // optional statement
else _Accept( mutex-member-name )
        statement                       // optional statement
...
    ...
```

A list of mutex members in an **_Accept** clause, e.g.:

```
_Accept( insert, remove );
```

it is equivalent to:

```
_Accept( insert ) else _Accept( remove );
```

Before a **_Accept** clause is executed, an outstanding call to the corresponding member must exist. If there are several mutex members that can be accepted, the **_Accept** clause nearest the beginning of the statement is executed. Hence, the order of the **_Accept**s indicates their relative priority for selection if several accept clauses can execute. Once the accepted call has completed *or the caller waits*, the statement after the accepting **_Accept** clause is executed and the accept statement is complete. If there are no outstanding calls to these members, the task is accept-blocked until a call to one of these members is made.

Note, when there are multiple **_Accept** clauses in the accept statement, *only one call is accepted* ("or" not "and"). This semantics is often confusing when using an accept statement and it may be helpful to draw an analogue between an accept statement with multiple accept clauses and an **if** statement with multiple **else if** clauses, as in:

```
if ( C1 ) {                _Accept( M1 ) {
    S1                         S1
} else if ( C2 ) {         } else _Accept( M2 ) {
    S2                         S2
} else if ( C3 ) {         } else _Accept( M3 ) {
    S3                         S3
}                          }
```

The **if** statement only executes one of S1, S2, S3; it does not execute them all. The reason is that the first conditional to evaluate to true executes its corresponding "then" clause (S1, S2, or S3), and the **if** statement terminates. Similarly, the accept statement accepts the first call to one of mutex members M1, M2, or M3, then executes the corresponding "then" clause (S1, S2, or S3) and the **_Accept** statement terminates. The analogue also holds when there is a list of mutex member names in an accept clause, as in:

```
if ( C1 || C2 ) {          _Accept( M1, M2 ) {
    S1                         S1
} else if ( C3 ) {         } else _Accept( M3 ) {
    S2                         S2
}                          }
```

The analogy differs when all the conditions are false for the **if** statement and no outstanding calls exist for the accept statement. For the **if** case, control continues without executing any of the "then" clauses; for the **_Accept** case, control *blocks* until a call to one of the accepted members occurs and then the corresponding "then" clause is executed. Hence, an accept statement accepts only one call, regardless of the number of mutex member names specified in the accept clauses forming the statement, and then any code associated with accepting that mutex member is executed. (See Section 10.7.1, p. 329 for further extensions to the **_Accept** statement, including how to accept no calls.) To accept multiple calls requires executing multiple accept *statements*, not multiple accept *clauses*.

9.6.3 Condition Variables and Wait/Signal Statements

In μC++, the type uCondition creates a queue object on which tasks can be blocked and reactivated in FIFO order, and is defined:

```
class uCondition {
  public:
    bool empty() const;
    long int front() const;
};
uCondition a, *b, c[5];
```

A condition variable is owned by the mutex object that performs the first wait on it; subsequently, only the owner can wait and signal that condition variable. The member routine empty() returns **false** if there are tasks blocked on the queue and **true** otherwise. The member routine front returns an integer value stored with the waiting task at the front of the condition queue. It is an error to examine the front of an empty condition queue. (front is used again on page 278.)

It is *not* meaningful to read or to assign to a condition variable, or copy a condition variable (e.g., pass it as a value parameter), or use a condition variable if not its owner.

To join such a condition queue, the active task calls member wait of the condition variable, e.g.,

```
Empty.wait();
```

which causes the active task to block on condition Empty, which causes further implicit scheduling. First, a task is unblocked from the acceptor/signalled stack if present, and then from the entry queue. If there are no waiting tasks present on either data structure, *the next call to any mutex member is implicitly accepted.*

When waiting, it is possible to optionally store an integer value with a waiting task on a condition queue by passing an argument to wait, e.g.:

```
Empty.wait( 3 );
```

If no value is specified in a call to wait, the value for that blocked task is undefined. The integer value can be accessed by other tasks through the uCondition member routine front. This value can be used to provide more precise information about a waiting task than can be inferred from its presence on a particular condition variable. The value stored with a waiting task and examined by a signaller should not be construed as a message between tasks. The information stored

```
_Monitor DatingService {
    uCondition Girls[20], Boys[20];
    int GirlPhoneNo, BoyPhoneNo;
  public:
    int Girl( int PhoneNo, int ccode ) {
        if ( Boys[ccode].empty() ) {
            Girls[ccode].wait();
            GirlPhoneNo = PhoneNo;
        } else {
            GirlPhoneNo = PhoneNo;
            Boys[ccode].signalBlock();      // wait for date to copy phone number
        }
        return BoyPhoneNo;
    }
    int Boy( int PhoneNo, int ccode ) {
        if ( Girls[ccode].empty() ) {
            Boys[ccode].wait();
            BoyPhoneNo = PhoneNo;
        } else {
            BoyPhoneNo = PhoneNo;
            Girls[ccode].signalBlock();      // wait for date to copy phone number
        }
        return GirlPhoneNo;
    }
};
```

Figure 9.11: Scheduling using Parameter Information: Solution 3

with the waiting task is not meant for a particular task nor is it received by a particular task. Any task in the monitor can examine it. Also, the value stored with each task is *not* a priority for use in the subsequent selection of a task when the monitor is unlocked.

A task is reactivated from a condition variable when another (active) task executes a signal. There are two forms of signal. The first form is the signal, e.g.:

```
Full.signal();
```

The effect of a call to member signal is to remove one task from the specified condition variable and push it onto the acceptor/signalled stack. The signaller continues execution and the signalled task is unblocked when it is next popped off the acceptor/signalled stack. The second form is the signalBlock, e.g.:

```
Full.signalBlock();
```

The effect of a call to member signalBlock is to remove one task from the specified condition variable and make it the active task, and push the signaller onto the acceptor/signalled stack, like an accept. The signalled task continues execution and the signaller is unblocked when it is next popped off the acceptor/signalled stack.

Figure 9.11 shows how the signalBlock can be used to remove the Exchange condition from the left example in Figure 9.9, p. 268. By replacing the calls to signal on the gender condition variables with a calls to signalBlock, the signaller now waits for its date to unblock and copy its phone number from a local variable into in a global monitor communication variable, and exit with the signaller's phone number. When the date exits its mutex member, the signaller unblocks and exits with the date's phone number from the global monitor communication variable. Notice, the total amount of blocking and unblocking is identical in both versions of the program.

9.7 Readers and Writer Problem

It is now time to go back to the problem that caused the move to high-level concurrency constructs: the readers and writer problem. A series of solutions are presented that correspond to semaphore solutions 3–6 in Section 7.6, p. 202. However, before examining the monitor solutions, the basic structure of each solution is examined from a software engineering perspective.

```
class ReadersWriter {
    _Mutex void StartRead();
    _Mutex void EndRead();
    _Mutex void StartWrite();
    _Mutex void EndWrite();
  public:
    void Read( ... ) {
        StartRead();
        // read
        EndRead();
    }
    void Write( ... ) {
        StartWrite();
        // write
        EndWrite();
    }
};
```

Figure 9.12: Basic Structure for Readers/Writer Solutions

Stepping back to the conditional critical-region solution for the readers and writer problem, the solution is structured as an entry and exit protocol for the reader and possibly for the writer (see page 254). As for locks (see start of Chapter 7, p. 177), the entry and exit protocol for readers and writers needs to be abstracted into routines so it is not duplicated and the implementation is independent of the interface. For conditional critical-region, the protocol can be encapsulated into four routines, used as follows:

```
shared struct rw { ... }
    reader              writer
StartRead( rw );     StartWrite( rw );
// read              // write
EndRead( rw );       EndWrite( rw );
```

However, this style of coding is exactly the same as bracketing a critical section with P and V, with the same potential for mistakes, e.g., forgetting an EndRead or EndWrite call. A better structure is to encapsulate both the protocol and the operation into two routines, as in:

```
   reader              writer
Read( rw, ... );     Write( rw, ... );
```

where "..." are arguments passed to the read or write operation within the protocol routines, respectively. The object-oriented form makes rw into a monitor type with mutex members Read and Write, as follows:

```
  reader             writer
rw.Read( ... );     rw.Write( ... );
```

Unfortunately, for reading, this organization presents exactly the same problem as putting both the read protocol and read operation together in a single critical region, i.e., only one reader can be in the monitor (critical region) at a time, and hence, there are no simultaneous readers. Therefore, it is necessary to put the reader entry and exit protocol back into separate mutex members, which is back to the initial structure. The answer is to put the protocol and operation into a **_Nomutex** interface member of the monitor, which the user calls. This structure is identical to the above conditional critical-region example but retains the object-oriented style by using member routines instead of normal routines. Therefore, the basic structure of all the following monitor solutions for the readers and writer problem is shown in Figure 9.12:

There are two interface routines, Read and Write, both of which have no mutual exclusion (i.e., **_Nomutex** members); both interface routines make calls to private mutual exclusion routines. (Remember, the presence of a single mutex member makes a type a mutex type.) Notice, like the conditional critical-region case, the Write routine can have the protocol and operation together because there is at most one writer and that writer can be the only task with mutual exclusion in the monitor. However, for consistency, both Reader and Writer members are structured the same way. This situation illustrates how a complex protocol can be enforced through no mutex members, even when the protocol

```
class ReadersWriter {
    uCondition rwait, wwait;
    int rcnt, wcnt;

    _Mutex void StartRead() {
        if ( wcnt > 0 || ! wwait.empty() ) rwait.wait();
        rcnt += 1;
        rwait.signal();
    }
    _Mutex void EndRead() {
        rcnt -= 1;
        if ( rcnt == 0 ) wwait.signal();          // last reader ?
    }
    _Mutex void StartWrite() {
        if ( rcnt > 0 || wcnt > 0 ) wwait.wait();
        wcnt += 1;
    }
    _Mutex void EndWrite() {
        wcnt -= 1;
        if ( ! rwait.empty() ) rwait.signal();    // anyone waiting ?
        else wwait.signal();
    }
  public:
    ReadersWriter() : rcnt(0), wcnt(0) {}
    _Nomutex int readers() { return rcnt; }
    . . .
};
```

Figure 9.13: Readers and Writer: Solution 3

requires multiple mutex calls. The Mesa programming language provides exactly the same capability:

> For example, a public external procedure *(no mutex member)* might do some preliminary processing and then make repeated calls into the monitor proper before returning to its client. [MMS79, p. 157]

9.7.1 Solution 3

The first solution examined is solution 3, which is the first semaphore solution to deal with no starvation of readers or writers, but still allows staleness. The monitor version is presented in Figure 9.13.

Notice the delay counters from the semaphore solution are gone. Instead, these counters have been replaced by checks for non-empty condition queues. As pointed out at the end of Section 7.6.2, p. 206, it is impossible to do this for a semaphore solution without a special P routine, which atomically Vs the parameter semaphore and then blocks the calling task if necessary (see end of Section 7.6.6, p. 215). It is possible for a monitor solution because of the ability to atomically release the monitor lock and wait on a condition variable.

The algorithm in the monitor is basically the same as its semaphore counterpart. The reader entry protocol checks if there is a writer using the resource or waiting writers, and if so, waits. Otherwise, the reader increments the read counter and signals another waiting reader. As a result, a group of waiting readers take turns unblocking each other, one after the other, called **daisy-chain signalling**, and the readers begin reading in FIFO order. Notice the signal is unconditional, unlike the semaphore version, because signalling an empty condition variable does not affect future waits. The reader exit protocol decrements the read counter, and the last reader of a group signals a waiting writer. Again, the signal is unconditional, unlike the semaphore version, for the same reason. The writer entry protocol checks if there are reader(s) or a writer using the resource, and if so, waits. Otherwise, the writer simply increments the write counter. The writer exit protocol decrements the write counter, and a check is made for a waiting reader, and if none, a waiting writer.

In all the mutex members, the notion of a baton is implicitly managed by the monitor. In effect, the additional support provided by the monitor mitigates the need to use an analogy like baton passing to cope with complexity. As well, the delay counters are superfluous and temporal anomalies between unlocking the monitor and waiting on a

```
class ReadersWriter {
    uCondition rwwait, wwait;
    int rcnt, wcnt;

    _Mutex void StartRead() {
        if ( wcnt > 0 || ! wwait.empty() || ! rwwait.empty() ) rwwait.wait();
        rcnt += 1;
        rwwait.signal();
    }
    _Mutex void EndRead() {
        rcnt -= 1;
        if ( rcnt == 0 ) {                              // last reader ?
            if ( ! wwait.empty() ) wwait.signal();      // writer waiting ?
            else rwwait.signal();                       // anyone waiting ?
        }
    }
    _Mutex void StartWrite() {
        if ( rcnt > 0 || wcnt > 0 ) {
            rwwait.wait();                              // wait once
            if ( rcnt > 0 ) wwait.wait();               // wait once more ?
        }
        wcnt += 1;
    }
    _Mutex void EndWrite() {
        wcnt -= 1;
        rwwait.signal();                                    // anyone waiting ?
    }
  public:
    . . .
};
```

Figure 9.14: Readers and Write: Solution 4

condition do not exist. However, there is still staleness because of the use of two waiting queues for the readers and writers.

9.7.2 Solutions 4 and 5

To eliminate staleness requires delaying both readers and writers on a single waiting queue to preserve temporal order of arrival. The monitor solutions in Figures 9.14 and 9.15, p. 279 present two alternatives for eliminating staleness, and both alternatives were discussed in Section 7.6.5, p. 211 using semaphores. Solution 4 has readers about to enter the resource assume the next task on the combined waiting condition (where both readers and writers wait in temporal order) is a reader and wake it up. Because of the potential for inadvertently waking a writer, each writer must recheck if it can use the resource. If the writer cannot use the resource because there are readers using it, the writer waits again on a special condition, which has at most one writer task waiting on it. The special condition is given the highest priority so a writer inadvertently restarted is always serviced next. Notice, this potential double blocking *in the monitor* by a writer requires internal scheduling (see Section 9.5.2, p. 266) because of the delay scheduling.

The reader entry protocol checks if there is a writer using the resource or a waiting high-priority writer or any waiting readers or writers, and if so, waits. Otherwise, the reader increments the read counter and unconditionally signals another waiting reader or writer; it is this signal that may inadvertently wake a writer, which must then be treated as having the highest priority. Again, daisy-chain signalling is used to restart the reader tasks. The reader exit protocol decrements the read counter, and the last reader of a group checks first for a high-priority writer and signals it if present, otherwise it unconditionally signals any waiting reader or writer. The writer entry protocol checks if there are readers or a writer using the resource, and if so, waits. When the writer wakes up, it must check again if there are any readers because it could have been inadvertently woken; if there are readers, the writer waits again on the high-priority condition. Otherwise, the writer simply increments the write counter. The writer exit protocol decrements the write counter, and unconditionally signals any waiting reader or writer.

The next solution (see Figure 9.15) is the same as solution 3, but gets back the lost information when both readers and writers wait on the same condition queue by maintaining a separate queue, which is always the same length as the condition queue, and each node of the separate queue indicates if the corresponding task blocked on the condition is a reader or writer. Maintaining the additional queue can be seen in the program in both locations where tasks block. A node is created and initialized to the particular kind of task, and then it is put on the back of the separate queue and the task waits on the condition. When a waiting task is restarted, the cooperation ensures it can immediately take its node off the front of the additional queue. Notice that a queue node is not dynamically allocated; it is allocated at the top of the task's stack before blocking, which is very efficient. The reader entry protocol can now determine if the task blocked at the front of the condition queue is a reader or writer by examining the value in the node at the front of the separate queue. Note, it is necessary to check for the existence of a blocked task (or non-empty queue) before checking the node at the head of the queue or an invalid pointer dereference occurs. This solution is important because its corresponding semaphore solution (see Figure 7.19, p. 214) did not work. The reason the monitor solution works is the atomic unlocking of the monitor and blocking of a task by the wait. Hence, there is no possibility for the explicit list of task-kinds and the implicit list of blocked tasks on the condition variable to get out of synchronization.

Like the DatingService monitor on page 268, it is possible to simplify the previous solution by folding the separate queue of reader/writer information into the condition queue by storing the reader/writer information for each task in the condition-queue node. In this case, the value is the kind of task, i.e., reader or writer, which is stored in each condition node (see Figure 9.16, p. 280). It is sufficient for a reader task to examine the kind of task at the front of the queue to know when to stop unblocking reader tasks. This solution is the same as solution 5(a), except the explicit queue is gone and replaced by storing a value with a blocked task at both calls to wait (one in StartRead and one in StartWrite). As well, the condition member front is used in StartRead to examine the user value stored with the task blocked at the front of the condition queue to determine if it should be signalled, but only after checking that there is a node to examine.

A variation on this solution is for the writer to awaken the next group of readers to use the resource when it exits the resource rather than have the readers awaken each other (see Figure 9.17, p. 280). Instead of daisy-chain signalling the waiting readers in StartRead, the readers are multiply signalled in EndWrite. Here, the cooperation for signalling a group of readers has been moved from the reader tasks to the writer, so when a reader task restarts after waiting it only has to increment the read counter. Now each signal moves a reader from the condition queue onto the acceptor/signalled stack. Because μC++ uses a stack for storing signalled tasks, the readers exit in LIFO order. Fortunately, LIFO restarting of readers does not cause problems with respect to correctness or staleness.

The final solutions (see Figure 9.18, p. 281) show how external scheduling can be used instead of internal scheduling to provide the simplest solutions. These solutions work because the monitor's entry queue maintains calls in FIFO order (maintaining temporal order of arrival), and tasks are selected from this queue when the monitor is inactive. Two solutions are shown because one is significantly better than the other. The obvious solution, on the left of Figure 9.18, p. 281, works as follows. A reader calling StartRead, checks for a writer using the resource, and if so, accepts a call to EndWrite, which blocks it on the acceptor/signalled stack and allows *only* a call to EndWrite. All other calls block outside of the monitor. After the writer using the resource calls EndWrite and decrements the write counter, the reader is restarted in the monitor, increments the reader counter, and can return to read the resource. The monitor is now inactive, and the next task to enter is the one at the head of the entry queue, i.e., the task waiting the longest, which might be either a reader or a writer task.

A writer calling StartWrite checks for either a writer or reader using the resource. If there is a writer using the resource, the writer in the monitor accepts a call to EndWrite, which blocks it on the acceptor/signalled stack and blocks all other calls outside the monitor until the writer using the resource calls EndWrite. If there are readers using the resource, the writer in the monitor accepts a call to EndRead, which blocks it on the acceptor/signalled stack and blocks all other calls outside the monitor, effectively marking the end for any new reader tasks joining this group (i.e., no more calls to StartRead are allowed).

In either case, there is no starvation possible by a continuous stream of readers or writers because tasks are always removed from the front of the FIFO entry queue, and the reader or writer wanting entry to the resource blocks other tasks outside the monitor. The first reader to finish calls EndRead, which is accepted by a waiting writer, if there is one, or the call is implicitly accepted because the monitor is inactive as there are no calling tasks. This reader and subsequent readers each decrement the read counter and accept the next finishing reader's call to EndRead, except the last reader, resulting in nested accepts. As a writer finishes, its call to EndWrite is accepted by either an accepting

```
class ReadersWriter {
    uCondition rwwait;
    int rcnt, wcnt;

    enum RW { READER, WRITER };
    struct RWnode : public uColable {
        RW rw;                                      // kind of task
        RWnode( RW rw ) : rw(rw) {}
    };
    uQueue<RWnode> rwid;                            // queue of kinds

    _Mutex void StartRead() {
        if ( wcnt > 0 || ! rwwait.empty() ) {
            RWnode r( READER );
            rwid.uAdd( &r );                        // remember kind of task
            rwwait.wait();
            rwid.uDrop();
        }
        rcnt += 1;
        if ( ! rwwait.empty() && rwid.head()->rw == READER ) // more readers ?
            rwwait.signal();
    }
    _Mutex void EndRead() {
        rcnt -= 1;
        if ( rcnt == 0 ) rwwait.signal();           // last reader ?
    }
    _Mutex void StartWrite() {
        if ( rcnt > 0 || wcnt > 0 ) {
            RWnode w( WRITER );
            rwid.uAdd( &w );                        // remember kind of task
            rwwait.wait();
            rwid.uDrop();
        }
        wcnt += 1;
    }
    _Mutex void EndWrite() {
        wcnt -= 1;
        rwwait.signal();                            // anyone waiting ?
    }
  public:
    . . .
};
```

Figure 9.15: Readers and Write: Solution 5(a)

reader or writer, if there is one, or the call is implicitly accepted because the monitor is inactive as there are no calling tasks. The writer then decrements the write counter. Thus, all the accepts match with appropriate routine calls.

The problem with this solution is that it inhibits concurrency in two ways. First, concurrency is inhibited by making each exiting reader task from a group, except the last reader, wait for *all* reader tasks in that group to exit. This problem results from the nested accepts, which stack up the exiting readers as they try to leave EndRead. Hence, there is a long delay before any reader in that group can exit the monitor and continue performing new work. Second, when an exiting reader is waiting for the next reader to exit, no new readers can enter the monitor because only EndRead calls are accepted. Hence, there is a long delay before a new reader can enter the monitor and subsequently read the resource.

To prevent these problems, it is clear that neither EndRead nor EndWrite can do an accept because that blocks the exiting task, which wants to leave immediately, and new reader tasks, which want to read simultaneously with the current group of readers. However, the accept cannot be removed; it must be relocated, and there is only one

```
class ReadersWriter {
    uCondition rwwait;
    int rcnt, wcnt;
    enum RW { READER, WRITER };

    _Mutex void StartRead() {
        if ( wcnt > 0 || ! rwwait.empty() ) rwwait.wait( READER );
        rcnt += 1;
        if ( ! rwwait.empty() && rwwait.front() == READER ) rwwait.signal();
    }
    _Mutex void EndRead() {
        rcnt -= 1;
        if ( rcnt == 0 ) rwwait.signal();              // last reader ?
    }
    _Mutex void StartWrite() {
        if ( rcnt > 0 || wcnt > 0 ) rwwait.wait( WRITER );
        wcnt += 1;
    }
    _Mutex void EndWrite() {
        wcnt -= 1;
        rwwait.signal();                               // anyone waiting ?
    }
  public:
    . . .
};
```

Figure 9.16: Readers and Write: Solution 5(b)

```
class ReadersWriter {
    uCondition rwwait;
    int rcnt, wcnt;
    enum RW { READER, WRITER };

    _Mutex void StartRead() {
        if ( wcnt > 0 || ! rwwait.empty() ) rwwait.wait( READER );
        rcnt += 1;
    }
    _Mutex void EndRead() {
        rcnt -= 1;
        if ( rcnt == 0 ) rwwait.signal();              // last reader ?
    }
    _Mutex void StartWrite() {
        if ( rcnt > 0 || wcnt > 0 ) rwwait.wait( WRITER );
        wcnt += 1;
    }
    _Mutex void EndWrite() {
        wcnt -= 1;
        if ( ! rwwait.empty() && rwwait.front() == WRITER )
            rwwait.signal();                           // unblock writer
        else                                           // multiple signalling
            while ( ! rwwait.empty() && rwwait.front() == READER )
                rwwait.signal();                       // unblock reader
    }
  public:
    . . .
};
```

Figure 9.17: Readers and Write: Solution 5(c)

Daisy-Chaining Accepting	Multiple Accepting
<pre>class ReadersWriter { int rcnt, wcnt; _Mutex void EndRead() { rcnt −= 1; if (rcnt > 0) _Accept(EndRead); } _Mutex void EndWrite() { wcnt = 0; } _Mutex void StartRead() { if (wcnt > 0) _Accept(EndWrite); rcnt += 1; } _Mutex void StartWrite() { if (wcnt > 0) _Accept(EndWrite); else if (rcnt > 0) _Accept(EndRead); wcnt = 1; } public: . . . };</pre>	<pre>class ReadersWriter { int rcnt, wcnt; _Mutex void EndRead() { rcnt −= 1; } _Mutex void EndWrite() { wcnt = 0; } _Mutex void StartRead() { if (wcnt > 0) _Accept(EndWrite); rcnt += 1; } _Mutex void StartWrite() { if (wcnt > 0) _Accept(EndWrite); else while (rcnt > 0) _Accept(EndRead); wcnt = 1; } public: . . . };</pre>

Figure 9.18: Readers and Write: Solution 5 (d & e)

place it can be moved to. The right solution of Figure 9.18 moves the accepts of the completing readers into a loop in StartWrite. In the left solution, the writer only accepted the last reader of a group; the right solution accepts all the exiting readers of the current reader group. The longest delay an exiting reader experiences is the time to restart the writer and for it to cycle through the loop. Notice, the technique used to solve this problem is to switch from daisy-chain accepting to multiple accepting. Does this solution inhibit concurrency for the writer task performing the accepts of the readers? No, because that writer cannot enter the resource until the current group of readers completes, so it would otherwise be blocked. Clearly, a writer is doing a little more of the cooperation work over the previous solution, which amortized the cooperation work evenly across all the tasks. This scheme might be a problem if each task is charged separately for performing its work because a writer does the additional cooperation work. In general, the extra work is extremely small, unless there are hundreds or thousands of reader tasks in a group.

9.7.3 Solution 6

There is no need for an equivalent to semaphore solution 6 because a monitor's wait atomically unlocks the monitor and blocks the current task on an exit, wait, or signal so using a private condition is unnecessary. Interestingly, it is still possible to construct a monitor solution using private conditions, which mimics the corresponding semaphore solution exactly.

9.8 Condition, Wait, Signal vs. Counting Semaphore, P, V

As mentioned, the wait and signal operations on condition variables in a monitor are similar to P and V operations on counting semaphores. The wait can block a task's execution, while a signal can cause another task to be unblocked. In fact, it is easy to construct a semaphore with operations P and V using a monitor (see Figure 9.19). Semaphores can also be used to build monitors, although any solution requires following complex coding conventions, which the compiler cannot check. As well, other important criteria, such as ease of use in programming complex concurrent problems, like the readers and writer problem, must be considered. Therefore, there is only a weak equivalence between the two mechanisms.

Other specific differences between condition variables and semaphores are as follows. When a task executes a P

External Scheduling	Internal Scheduling
`_Monitor uSemaphore {` `    int cnt;` `  public:` `    uSemaphore( int cnt = 1 ) : cnt(cnt) {}` `    void V() {` `        cnt += 1;` `    }` `    void P() {` `        if ( cnt == 0 ) _Accept( V );` `        cnt -= 1;` `    }` `};`	`_Monitor uSemaphore {` `    int cnt;` `    uCondition cntPos;` `  public:` `    uSemaphore( int cnt = 1 ) : cnt(cnt) {}` `    void V() {` `        cnt += 1;` `        cntPos.signal();` `    }` `    void P() {` `        if ( cnt == 0 ) cntPos.wait();` `        cnt -= 1;` `    }` `};`

Figure 9.19: Mimic Semaphore with Monitor

operation, it does not necessarily block since the semaphore counter may be greater than zero. In contrast, when a task executes a wait it always blocks. As a consequence, a condition can only be used for synchronization not mutual exclusion. When a task executes a V operation on a semaphore it either unblocks a task waiting on that semaphore or, if there is no task to unblock, increments the semaphore counter. In contrast, if a task executes a signal when there is no task to unblock, there is no effect on the condition variable; hence, signals may be lost so the order of executing wait and signals in a monitor is critical. Another difference between semaphores and monitors is that tasks awakened by a V can resume execution without delay. In contrast, because tasks execute with mutual exclusion within a monitor, tasks awakened from a condition variable are restarted only when the monitor is next unlocked. Finally, monitors may inhibit some concurrency because there is only one monitor lock, e.g., simultaneously inserting and removing from a bounded buffer (see the discussion in Section 9.2, p. 251). Essentially, the semaphore can provide finer grain concurrency than a monitor, but at a high cost in complexity.

9.9 Monitor Errors

As mentioned previously in Section 9.3.1, p. 257, some monitor implementations result in deadlock if the active monitor task calls a mutex member, which can be solved with code restructuring. While code restructuring can avoid this particular error, there are other problems when using monitors.

One problem that has been studied extensively is **nested monitor call**. When a task blocks in a monitor, either through **_Accept** or wait, other tasks can enter the monitor so that progress can be made. However, when a task, T1, calls from one monitor, M1, to another monitor, M2, and blocks in M2, M2 can continue to process requests but not M1. The reason M1 cannot process requests is that task T1 still has M1 locked and blocking in M2 only releases M2's monitor lock. Therefore, acquiring one monitor and blocking in another monitor results in a hold and wait situation, which may reduce concurrency in most cases and may result in a synchronization deadlock in others.

In some situations, holding all the monitor resources except the last is exactly the desired semantics; in other cases, it is not the desired semantics. Suggestions have been made for special monitor semantics to handle this situation, e.g., when a task blocks in a monitor, all the monitor locks it is currently holding are released. Clearly, neither approach is perfect for all circumstances, and hence, both or more semantics would need to be supported, further increasing the complexity of the monitor. In μC++, there is no special semantics for the nested monitor situation; monitor locks are held if a task holding them blocks. Therefore, this situation must be considered when designing complex resource allocation schemes, otherwise it might lead to deadlock.

9.10 Coroutine-Monitor

The coroutine-monitor is a coroutine with mutual exclusion and so it can be accessed simultaneously by multiple tasks. A coroutine-monitor type has a combination of the properties of a coroutine and a monitor, and can be used where a combination of these properties are needed, such as a finite automata that is used by multiple tasks.

A coroutine-monitor type has all the properties of a **_Coroutine** and a **_Mutex class**. The general form of the coroutine-monitor type is the following:

```
_Mutex _Coroutine coroutine-name {
  private:
    . . .             // these members are not visible externally
  protected:
    . . .             // these members are visible to descendants
    void main();      // starting member
  public:
    . . .             // these members are visible externally
};
```

Like the **_Mutex** qualifier on a **class**, the **_Mutex** qualifier on a **_Coroutine** means all public member routines of the coroutine have the mutual-exclusion property, unless overridden on specific member routines with the **_Nomutex** qualifier. When the **_Nomutex** qualifier is placed on a **_Coroutine** definition, it indicates all public member routines have the no-mutual-exclusion property, which is the same as a **_Coroutine**, unless overridden on specific member routines with the **_Mutex** qualifier. The default for a coroutine if no qualifier is specified is **_Nomutex** because the mutual-exclusion property for public members is typically not needed.

```
_Mutex _Coroutine MC1 {                 _Nomutex _Coroutine MC2 {
    . . .                                   . . .
  public:                                 public:
    mem1() . . .          // mutex          mem1() . . .          // no mutex
    _Nomutex mem2(). . . // no mutex        _Mutex mem2() . . .   // mutex
    ~MC1() . . .          // mutex          ~MC2() . . .          // mutex
}                                       }
```

Therefore, a **_Coroutine** creates a coroutine-monitor if and only if it has at least one **_Mutex** member routine, and that **_Mutex** member routine can have any visibility, i.e., **private**, **protected** or **public**. As well, the destructor of a coroutine-monitor is always mutex, regardless of where it is defined or if it makes no references to the coroutine-monitor's shared variables. Hence, a coroutine-monitor always has one mutex member, its destructor, and a coroutine-monitor's destructor cannot be qualified with **_Nomutex**. For convenience, the preprocessor macro name **_Cormonitor** is defined to be "**_Mutex _Coroutine**" in uC++.h.

9.10.1 Coroutine-Monitor Creation and Destruction

A coroutine-monitor is the same as a monitor with respect to creation and destruction.

9.10.2 Coroutine-Monitor Control and Communication

A coroutine monitor can make use of suspend, resume, **_Accept** and uCondition variables, wait, signal and signalBlock to move a task among execution states and to block and restart tasks that enter it. As for monitors, when creating a cyclic call-graph (i.e., full coroutines) using coroutine-monitors and multiple threads, it is the programmer's responsibility to ensure the cycle does not result in deadlock, often by having at least one of the members in the cycle be a **_Nomutex** member.

Figure 9.20 shows the implementation of the coroutine-monitor uBarrier discussed in Section 7.3.3, p. 188. This example illustrates how all of the ideas presented so far are combined. First, because a barrier is accessed by multiple tasks, it must have mutual exclusion; as well, because a barrier needs to retain state between calls to last, it should be a coroutine. The constructor initializes the total number of tasks using the barrier and other internal counters. The two public **_Nomutex** members, total and waiters, return barrier state information, i.e., the total number of tasks using the barrier and the number of tasks currently waiting on the barrier, respectively. Even though both members are **_Nomutex**, consistent values are always returned. The public member reset changes the total number of tasks using the barrier. No tasks may be waiting on the barrier when this total is changed. The public member block counts the tasks as they arrive at the barrier and blocks all but the last one. The last task invokes the member last to reset the barrier and unblocks any waiting tasks using multiple signalling. (Member last is made **virtual** so it can be overridden by subclasses and work correctly with subtype polymorphism.) The default last member resumes the default virtual coroutine main, which simply suspends back. Normally, a user supplies more complex versions of these two members

```
_Mutex _Coroutine uBarrier {
    uCondition Waiters;
    unsigned int Total, Count;

    void init( unsigned int total ) {
        Count = 0;
        Total = total;
    }
  protected:
    void main() {
        for ( ;; ) {
            suspend();
        }
    }
  public:
    uBarrier( unsigned int total ) {
        init( total );
    }
    _Nomutex unsigned int total() const {      // total participants in the barrier
        return Total;
    }
    _Nomutex unsigned int waiters() const { // number of waiting tasks
        return Count;
    }
    void reset( unsigned int total ) {
        init( total );
    }
    void block() {
        Count += 1;
        if ( Count < Total ) {                  // all tasks arrived ?
            Waiters.wait();
        } else {
            last();                             // call the last routine
            Count = 0;
            for ( ; ! Waiters.empty(); ) {      // restart all waiting tasks
                Waiters.signal();
            }
        }
    }
    virtual void last() {                       // called by last task to reach the barrier
        resume();
    }
};
```

Figure 9.20: Barrier Implementation

```
monitor BoundedBuffer {
        int front, back, count;
        int Elements[20];
    public:
        BoundedBuffer() : front(0), back(0), count(0) {}

        void insert( int elem ) {
            waituntil count != 20;            // buffer full ?
            Elements[back] = elem;
            back = ( back + 1 ) % 20;
            count += 1;
        }
        int remove() {
            waituntil count != 0;             // buffer empty ?
            int elem = Elements[front];
            front = ( front + 1 ) % 20;
            count -= 1;
            return elem;
        }
};
```

Figure 9.21: Bounded Buffer – Implicit Synchronization

through inheritance (see Figure **??**, p. ??), which manipulates complex state. Notice the use of resume, suspend, wait, and signal in this coroutine-monitor, illustrating the combination of capabilities from the coroutine and the monitor.

9.11 Monitor Taxonomy

Interestingly, the term "monitor" does not denote a single language construct with fixed semantics. While all monitors imply some form of implicit mutual exclusion to ensure data associated with the monitor is accessed serially, it is the scheduling aspect that varies greatly among monitors. Without a clear and precise understanding of these variations it is extremely difficult to write correct concurrent programs using monitors. Therefore, understanding the differences is essential when trying to select the best monitor to solve a problem, or when confronted with a specific monitor, knowing how to use this monitor to solve a problem.

9.11.1 Explicit and Implicit Signal Monitor

The monitors discussed thus far all use explicit synchronization mechanisms, such as accepting or signal/wait. This form of monitor is referred to as an **explicit signal monitor** because a programmer explicitly performs the synchronization, usually using some form of cooperation. The alternative to explicit synchronization is conditional delay, similar to the **await** of the conditional critical-region, but for monitors. In this kind of monitor, the synchronization is implicitly performed by the implementation, and hence, is referred to as an **implicit signal monitor** or **automatic-signal monitor** because the unblocking (signalling) of waiting tasks is done automatically. The implicit/automatic-signal monitor was proposed by Hoare [Hoa74, p. 556].

An implicit/automatic-signal monitor has the unconditional wait on a condition variable changed to use a wait on a conditional expression, e.g.:

waituntil *conditional_expression*

If the conditional expression is false, a task blocks and the monitor automatically checks for a task with a true condition or selects the next task from the entry queue. A waiting task is only unblocked when the expression it is waiting on becomes true. A consequence of this change is the elimination of condition variables and signals, which simplifies the monitor. However, conditional delays cannot express direct cooperation and can be runtime expensive (see Section 9.2.1, p. 254). Figure 9.21 illustrates a fictitious automatic-signal monitor (i.e., not supported in μC++) for a bounded buffer.

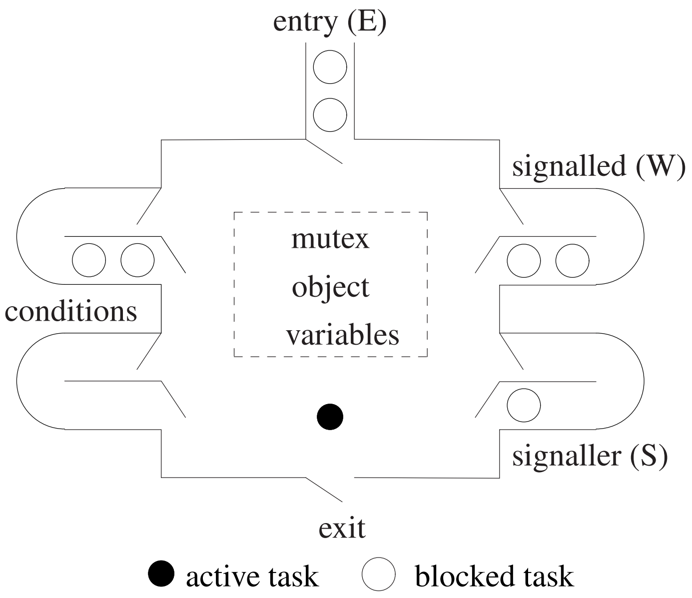


Figure 9.22: General Mutex Object

9.11.2 Implicit Monitor Scheduling

Previous discussion has centred on how a programmer can schedule tasks in a monitor, either internally or externally. However, a monitor has a small amount of implicit scheduling, which attempts to keep the monitor as busy as possible if there are waiting tasks. For example, in μC++, when a task exits the monitor, implicit scheduling occurs to restart the next blocked task on the acceptor/signalled stack or entry queue. A programmer cannot change this implicit scheduling; it is part of the internal implementation of the monitor. However, the particular implicit scheduling for μC++ is not the only approach; other implicit scheduling schemes are valid.

In general, implicit scheduling can sensibly be applied in the following three situations: when a task executes a wait, an accept/signal or exits from a mutex member. In these three situations, the monitor can become inactive, and hence, implicit scheduling occurs to make the monitor active again if tasks are waiting. As well, the implicit scheduler must know where to look for waiting tasks so one can be scheduled to use the monitor. Waiting tasks are normally blocked on a number of queues internal to the monitor (e.g., the entry queue). Figure 9.22 shows the general form of a monitor with a set of tasks using, or waiting to use, the monitor. The entry, signalled, and condition queues have been discussed previously. (It is sufficient for this general discussion to use queues; the stack in μC++ is a special case related to a different issue.) The new queue in the diagram is the **signaller queue**, which is used for signal-block scheduling, where the signaller task must wait for the signalled task to execute. For accepting, it is possible to consider the mutex queues like condition queues and the signaller queue as the acceptor stack; accepting is then like signal-block, where the acceptor waits for the signalled task, on one of the specified mutex queues, to execute. (The difference is that signal-block on an empty condition does not block, whereas an accept on an empty mutex queue(s) blocks.) For this discussion, it is assumed that any signal moves the signalled task to the signalled queue and the signaller task to the signaller queue, before implicit scheduling occurs. In this way, the implicit scheduler has a complete and simple picture of all the tasks waiting to use the monitor. In the case of an automatic-signal monitor, there are no condition queues and only one queue is needed to manage the tasks with false conditional expressions (similar to the implementation of conditional critical-region); the tasks with false conditional expressions can be put on the signalled queue because each waiting task is effectively eligible to run when the monitor is unlocked so that it can recheck its conditional expression. Thus, the diagram in Figure 9.22 can be used to locate all waiting tasks for both explicit and implicit signal monitors. The implicit scheduler can now select a task from any of the entry, signalled, or signaller queues. In certain situations, one or more of these queues can be empty. Depending on the kind of monitor, a particular choice is made. All other tasks must wait until the next implicit scheduling point is reached by the active monitor task.

	relative priority
1	$E_p = W_p = S_p$
2	$E_p = W_p < S_p$
3	$E_p = S_p < W_p$
4	$E_p < W_p = S_p$
5	$E_p < W_p < S_p$
6	$E_p < S_p < W_p$
7	$S_p = W_p < E_p$
8	$W_p < S_p = E_p$
9	$W_p < E_p < S_p$
10	$S_p < W_p = E_p$
11	$S_p < E_p < W_p$
12	$W_p < S_p < E_p$
13	$S_p < W_p < E_p$

Table 9.1: Relative Priorities for Internal Monitor Queues

9.11.3 Monitor Classification

To understand the differences among monitors, it is necessary to have a classification scheme that clearly delineates the differences in a meaningful way. The classification must provide a criteria for identifying any differences among monitors. The criteria should generate some form of taxonomy, in which all extant monitors should appear; the taxonomy may be general enough to suggest forms for which there are no existing monitors.

The following monitor taxonomy covers all extant monitors and uncovers several new ones. Initially, the monitors are divided into two groups based on whether the signal operation is explicit or implicit; within these two broad categories, several additional sub-categories are developed based on the semantics of the signal operation. (A more comprehensive discussion of the classification is presented in [BFC95].)

9.11.3.1 Explicit-Signal Monitors

The classification scheme for explicit-signal monitors is based on an exhaustive case analysis of the scheduling possibilities for the three internal monitor queues—the entry, signalled and signaller queues—when an implicit scheduling point is reached by the active monitor task. The different kinds of monitors are classified based on the relative priorities associated with these three queues. Each queue has a specific static priority, referred to as entry priority (E_p), signalled (waiting) priority (W_p), and signaller priority (S_p), respectively. The relative orderings of these three priorities yields 13 different possibilities, which are given in Table 9.1. There is one case in which the 3 priorities are equal. There are $\binom{3}{2} = 3$ cases in which exactly two priorities are equal, and each equal pair can be either greater than or less than the third priority, hence $\binom{3}{2} \times 2 = 6$ cases. Finally, there are $3! = 6$ cases in which the priorities are all different.

Cases 7–13 are rejected because the entry queue has priority over an internal queue. Giving the entry queue this level of priority has several severe disadvantages. First, it inhibits concurrency: it blocks tasks that could otherwise exit the monitor and continue their normal execution. Second, it creates the potential for unbounded waiting if there is a steady stream of tasks entering the monitor. Moreover, there are few, if any, compensating advantages to giving the entry queue such priority. Therefore, only cases 1–6 of Table 9.1 are examined.

If two or more queues have equal priority, the scheduler chooses one arbitrarily. Notice, it is acceptable for the entry queue to have equal priority with the other queues. In these cases, the arbitrary choice for selection ensures some bound on waiting among the queues so a continuous stream of calling tasks does not preclude tasks from being scheduled from other queues. Finally, the highest priority queue can have at most one task waiting on it because after a task blocks on the highest priority queue, it is immediately restarted. As a result, most implementations optimize out the queue associated with the highest priority task, which is why μC++ has only a single queue (stack). Nevertheless, for purposes of understanding the classification scheme, all queues need to be identified and understood, even though some queues may not appear in the implementation.

	relative priority
1	$E_p = W_p$
2	$E_p < W_p$
3	$W_p < E_p$

Table 9.2: Relative Priorities for Extended Immediate-Return Monitor

9.11.3.2 Immediate-Return Monitor

One additional explicit signal monitor needs mentioning for completeness, and because it identifies an interesting property of monitors that is important in later discussions. The interesting property is that most signals occur directly or indirectly before a return statement.

> The question whether the signal should always be the last operation of a monitor procedure is still open. [Hoa74, p. 557].

This property is true for all the monitors in this chapter, except two versions of the dating service monitor, which have a signal before a wait. In Brinch-Hansen's Concurrent Pascal [Bri75, p. 205] this property was enforced by having the signal also return the signaller from the mutex member. This kind of monitor is called an **immediate return monitor** because the signal implicitly does a return. Since the signaller leaves the monitor, the next task to enter the monitor is the signalled task, if there is one. Such a monitor precludes a task from multiple signalling or executing code after the signal with the monitor lock.

While this property occurs frequently in monitors, it is not true for all monitors. In fact, as has been pointed out by Howard [How76, p. 51] and Andrews [And91, p. 312], some monitor programs that can be written with ordinary explicit-signal monitors cannot be written with immediate-return monitors unless the interface to the monitor is changed. As a result, there is only a weak equivalence between the immediate-return monitor and other explicit-signal monitors. Howard proposed an **extended immediate-return monitor** [How76, p. 51] allowing a signal to also precede a wait; in this case, the signaller does not leave the monitor because it blocks immediately on the specified condition queue for the wait after the signal. Again, the signalled task, if there is one, enters the monitor next. The extended immediate-return monitor has been shown to be as general as the other explicit-signal monitors.

However, there are still useful situations where a signaller may continue executing in a monitor immediately after a signal, i.e., not return or block. For example, Figure 9.17, p. 280 shows a solution for the readers and writer problem where the finishing writer task does all of the signalling for the next task(s) to use the resource, i.e., it signals the next writer or the next group of reader tasks using multiple signalling rather than daisy-chain signalling. While it might appear that the finishing writer does more than its fair share of work with respect to the cooperation, there are cases where it is more efficient to localize the cooperation rather than spread it out [BFC95, p. 101]

The monitor categorization-scheme is applicable to both the immediate-return and extended immediate-return monitors; however, there is no signaller queue because either the signaller task leaves the monitor immediately or it is put on a condition queue. Therefore, the next active task is either a calling or a signalled task. Table 9.2 shows these possibilities. Again, the case where the entry queue has priority over an internal queue is rejected.

9.11.3.3 Automatic-Signal Monitors

It is possible to further classify the implicit/automatic-signal monitor based on the kinds of variables allowed in the conditional expression. If both monitor variables and local variables of a mutex member may appear in the conditional expression, the monitor is called a **general automatic-signal monitor**. If only monitor (global) variables are allowed in the conditional expression, the monitor is called a **restricted automatic-signal monitor**.

General automatic-signal monitors require an expensive implementation, as for conditional critical-region (see page 255), because, in the worst case, it involves re-evaluating the conditional expressions of all waiting tasks. Since a conditional expression can potentially depend on local variables of a task, including parameters of a mutex member, the local context of a task must be accessed to evaluate its conditional expression. The simplest way to accomplish this is to awaken tasks from the signalled queue one at a time so each can re-evaluate its conditional expression. If a task evaluates its condition and finds the condition true, it proceeds; otherwise it again blocks on the signalled queue and allows another task to try. The problem with this approach is that many context switches may occur before some task can proceed.

	relative priority
1	$E_p = W_p$
2	$E_p < W_p$
3	$W_p < E_p$

Table 9.3: Relative Priorities for Automatic-Signal Monitor

Restricted automatic-signal monitors have a much more efficient implementation. Since all variables in the conditional expressions are monitor variables, and hence do not depend on the context of individual tasks, the conditional expressions can be evaluated efficiently by the task that is about to unlock the monitor. The efficiency can be further improved by noting which conditional expressions represent distinct conditions; if two or more tasks are waiting for the same condition, it need only be evaluated once. Kessels [Kes77] proposed a notation that both restricts conditional expressions to use only monitor variables and also allows the programmer to specify which conditional expressions represent distinct conditions. His notation moves the conditional expressions from the wait into the monitor declarations and gives each a name; these names are then used in the wait instead of an expression, e.g.:

```
monitor M {
    int a, b;
    condexpr(a > b) c;    // only monitor variables allowed in the expression
    int r(. . .) {
        . . . waituntil c; . . .    // wait until the conditional expression, c, is true
```

Since only monitor variables are allowed in a conditional expression, the time it takes to find the next task to execute is determined by the cost of re-evaluating the conditional expressions. Thus, compared to general automatic-signal monitors, there is the potential for significant execution time saving in determining the next task to execute. The drawback is that local mutex member information, including parameters, cannot be used in a conditional expression of a restricted automatic-signal wait, which precludes solutions to certain problems, like the dating service.

The monitor-categorization scheme is applicable to either kind of automatic-signal monitors; however, there are no condition queues and no signaller queue. Therefore, the next active monitor task is either a calling task or a task on the signalled queue waiting for its conditional expression to evaluate to true. Table 9.3 shows these possibilities. Again, the case where the entry queue has priority over an internal queue is rejected.

9.11.3.4 Simplified Classification

The remaining "useful" monitors are now organized along the following lines (see Table 9.4). First, the monitors are divided into two groups (the columns) based on the priority of the entry queue. This division is useful because when the priority of the calling tasks is equal to either the signaller tasks or signalled tasks, it may make writing a monitor more complex. (This complexity is discussed further in Section 9.13.1, p. 296.) These two groups are called the **priority monitor** and **no-priority monitor**, respectively. In priority monitors, tasks already in the monitor have priority over calling tasks; in no-priority monitors, they do not. The importance of this crucial property has often been highly underrated and even ignored in the explanation of a monitor's semantics behaviour.

Within the two groups, it is possible to pair the monitors based on aspects of the signal (the rows). In the **blocking monitor**, the signaller task blocks on the signaller queue while the signalled task re-enters the monitor. In contrast, in the **non-blocking monitor**, the signaller task continues execution in the monitor and the signalled task can re-enter the monitor only when the monitor is unlocked. In the **quasi-blocking monitor**, either signaller or signalled tasks continue execution in the monitor and the other blocks. Finally, the extended immediate return and automatic-signal monitors form the last two pairs of monitors. This organization makes it easy to remember and understand the different kinds of monitors. Interestingly, there are four new kinds of monitors identified by the classification scheme over those discussed in the past: PQB, NPQB, NPRET and NPAS.

μC++ only supports priority monitors, and there is a choice between blocking or non-blocking signal with, signalBlock and signal, respectively.

9.12 Monitor Equivalence

All monitors are equivalent in a very weak sense by noting that any kind of monitor can be implemented with semaphores and can in turn be used to implement semaphores. Thus any monitor program for one kind of moni-

Signal Characteristics	Priority	No Priority
Blocking	$E_p < S_p < W_p$ Priority Blocking (PB)	$E_p = S_p < W_p$ No Priority Blocking (NPB)
NonBlocking	$E_p < W_p < S_p$ Priority Non-Blocking (PNB)	$E_p = W_p < S_p$ No-Priority Non-Blocking (NPNB)
Quasi-Blocking	$E_p < W_p = S_p$ Priority Quasi-Blocking (PQB)	$E_p = W_p = S_p$ No-Priority Quasi-Blocking (NPQB)
Extended Immediate Return	$E_p < W_p$ Priority Immediate Return (PRET)	$E_p = W_p$ No-Priority Immediate Return (NPRET)
Automatic Signal	$E_p < W_p$ Priority Automatic Signal (PAS)	$E_p = W_p$ No-Priority Automatic Signal (NPAS)

Table 9.4: Useful Monitors

tor can be mechanically translated into a program for any other kind of monitor by "compiling" the monitor program into code that synchronizes using semaphore operations and then using the other kind of monitor to implement the semaphore operations. However, this transformation is unsatisfying because it yields a much lower level program than the original. (This kind of very weak equivalence between language features has been called the "Turing tar pit" by Howard [How76, p. 49].) Only with transformations preserving the basic structure of the monitor program, i.e., transformations that do not introduce any new types, nor change the monitor interface or its overall structure, can monitor kinds be considered strongly equivalent. These transformations are amenable to programmers and language translators when converting from one kind of monitor in one language to a different kind of monitor in another language.

Because there are 10 kinds of monitors, there are $\binom{10}{2}$ different transformations. In fact, all transformations are possible [BFC95], so any monitor can be transformed into another, while preserving the basic structure of the monitor program. However, many of the transformations are complex, illustrating a weak equivalence between certain kinds of monitors, although still structure preserving. The only transformations examined here are the ones from explicit signal monitors to automatic signal. The reason these transformation are especially interesting is because many languages with monitors do not support automatic signalling, and automatic signalling is a useful technique for quick prototyping of a monitor, where speed of software development overrides execution performance. After a concurrent system is working, any performance bottleneck resulting from an automatic-signal monitor can be converted to an explicit-signal monitor.

9.12.1 Non-FIFO Simulations

Two non-FIFO simulations are presented for PB and PNB explicit-signal monitors to a PAS monitor. These simulations illustrate the techniques needed to construct an automatic-signal monitor. Both simulations use an additional condition variable. However, these non-FIFO simulations suffer from starvation and staleness/freshness, which are both dealt with in Section 9.12.2, p. 293. The basic approach in these transformations is to have each task check its own conditional expression. If the condition is true the task proceeds to use the monitor; if it is false, the task wakes up the next waiting task in the monitor so it can check its condition. This approach results in repeatedly waking all waiting tasks to let them recheck their conditions.

In general, a simulation involves changes in three or four locations of the automatic-signal monitor to convert to an explicit-signal monitor:

1. monitor-variable declarations: these are additional definitions and declarations added solely for the simulation and not part of the original automatic-signal monitor.

2. monitor-routine entry: this is additional definitions, declarations, and code involving signal/wait inserted at the start of each monitor routine in preparation for using the automatic-signal monitor by a particular task. This section may be empty in a simulation, i.e., no code is required.

3. waituntil statement: this statement is transformed into declarations and code, involving signal/wait, mimicking the waituntil statement.

4. monitor-routine exit: this is additional declarations and code, involving signal/wait, inserted before the return of each monitor routine usually to complete the current predicate checking-cycle and/or initiate the next one. This section may be empty in a simulation, i.e., no code is required.

Each simulation is shown as a table with a line for each of the change points and one column containing the automatic-signal code and the other column the equivalent explicit-signal code.

Two transformations from a priority automatic-signal (PAS) monitor to a priority nonblocking (PNB) monitor are shown in Table 9.5. For these transformations, a condition variable is declared, named IS, on which tasks with false predicates wait. The difference between the transformations is at the start of the waituntil transformation of the automatic-signal monitor. The transformation PNB_1 has an initial check if the predicate expression is true, and if so, continue; the transformation PNB_2 does not have this initial check. Then, all tasks blocked on condition IS are signalled, which is an example of multiple signalling (see Section 9.5.1, p. 265), to recheck their predicate expressions because the monitor state may have changed. The current task then blocks in a loop rechecking its predicate expression each time it is signalled. On return from a monitor routine, all tasks on condition IS are signalled (using multiple signalling) so they can recheck their predicate expressions because the monitor state may have changed. Notice, when a task finds its predicate is true and continues to use the monitor, there may be tasks blocked on the signaller (W) queue. These tasks form the front of the next checking cycle started by a return or a rewait by the task using the monitor.

Because transformation PNB_2 does not perform an initial predicate check, it gives priority to waiting tasks. That is, a task performing a waituntil does not check its predicate until all other waiting tasks have checked their predicates. This approach is attempting to deal with starvation and staleness/freshness issues by trying to service waiting tasks in FIFO order. However, subsequent discussion shows that this approach is not entirely successful.

The transformation is straightforward, except to ensure the loop signalling all tasks on condition IS terminates. Using the nonblocking signal, all tasks on condition IS are moved to the signaller queue before another task can execute. Therefore, condition IS always empties and the loop terminates. Interestingly, this transformation does not work if the nonblocking signal is changed to a blocking signal. The problem occurs while attempting to signal all the tasks blocked on condition IS. Because the signalled task executes first, it can find its condition false and block again on condition IS. Hence, when the signaller task restarts, it always finds condition IS non-empty, so it loops forever checking predicates. Finally, the multiple signalling in this algorithm is performed with a loop, which is $O(n)$; if a signalAll (broadcast) is available, the entire condition list can be moved by a single linked-list operation, which is $O(1)$. Nevertheless, there are always $O(n)$ predicate checks.

The transformation from a priority general-automatic-signal monitor to a priority-blocking (PB) monitor is shown in Table 9.6. As before, a condition variable is declared, named IS, on which tasks with false predicates wait. As well, the waituntil of the automatic-signal monitor can begin with or without an initial predicate check, i.e., the predicate check can be removed from the top of the **for** loop, in an attempt to deal with starvation and staleness/freshness. Otherwise, the first task on condition IS is signalled, which starts daisy-chain signalling (see Section 9.7.1, p. 276), to recheck its predicate expression because the monitor state may have changed. Because the signaller blocks for this signal, it must again recheck its predicate expression when it restarts because the monitor state may have changed. Only if the predicate expression is still false does the task block on condition IS. When a task is signalled, it loops back (and possibly rechecks its predicate expression, and if still false) signals the first task on condition IS. On return from a monitor routine, the first task on condition IS is signalled, which starts daisy-chain signalling, so it can recheck its predicate expressions because the monitor state may have changed. Notice, when a task finds its predicate is true and continues to use the monitor, there may be tasks blocked on the signaller queue. These tasks form the front of the next checking cycle started by a return or a rewait by the task using the monitor.

Section 9.13.2, p. 297 points out that a returning task with a blocking signal can have its exit delayed, possibly significantly, because of the blocking-signal semantics. With respect to the transformation, there can be an $O(n)$ delay if multiple waiting tasks have a true predicate as a result of changes from a returning task. As a result, tasks may not exit the monitor in FIFO order. Since this is a fundamental problem of blocking signal, no simulation using blocking signal can completely prevent its occurrence.

The transformation is straightforward, except to ensure the loop signalling all tasks on condition IS terminates. With the blocking signal, the signaller is moved to the signaller queue *before* another task can execute. Because the signalled task cannot wait again on condition IS without first doing a blocking signal on condition IS, which puts it on the signaller queue, eventually condition IS becomes empty. Interestingly, this transformation does not work if the blocking signal is changed to a nonblocking signal. Again, the problem occurs while attempting to signal all the tasks

PAS	PNB$_1$	PNB$_2$
declarations	condition IS;	condition IS;
waituntil C_i;	**if** (! C_i) { **while** (! IS.empty()) signal IS; **do** { wait IS; } **while** (! C_i); }	**while** (! IS.empty()) signal IS; **while** (! C_i) { wait IS; }
return $expr_{opt}$;	**while** (! IS.empty()) signal IS; **return** $expr_{opt}$;	**while** (! IS.empty()) signal IS; **return** $expr_{opt}$;

Table 9.5: Transform PAS Monitor to PNB Monitor

PAS	PB
declarations	condition IS;
waituntil C_i;	**for** (; ! C_i;) { *// optional check* signalblock IS; **if** (C_i) **break**; wait IS; }
return $expr_{opt}$;	signalblock IS; **return** $expr_{opt}$;

Table 9.6: Transform PAS Monitor to PB Monitor

blocked on condition IS. Because the signaller task executes first, it blocks immediately on condition IS at the wait. Hence, when a signalled task restarts, it always finds condition IS non-empty, so it loops forever checking predicates.

9.12.1.1 Problems

Unfortunately, these transformations deal with starvation and/or staleness/freshness. The problem with the transformations is that some notion of fairness, like FIFO ordering, of task condition-checking is not ensured. Figure 9.23 shows one possible starvation/staleness/freshness scenario for the PNB simulation.[5] The failing PNB scenario requires a task to enter the monitor with waiting tasks (false predicates), and the entering task's predicate is also false. The entering task then moves all the tasks to the signaller queue (multiple signalling) so these tasks can recheck their predicates, as the monitor state may have changed. However, after the waiting tasks are moved, the entering task waits on the empty IS condition variable, putting it at the head of the queue. As the waiting tasks unblock, test and wait again, they line up behind the entering task, e.g. (and see line 8 in Figure 9.23):

	IS	monitor	signaller	
1	A B	C		enter; C_i false
2		C	A B	signal all IS
3	C	A	B	wait IS
4	C	A	B	unblock; C_i false
5	C A	B		wait IS
6	C A	B		unblock; C_i false
7	C A B			wait IS

[5] For all tables of this form, the queue head is on the left. Also, for a line with a "wait" comment, the wait operation has already been performed by the task in the monitor on the previous line, and the "wait" line shows that task already blocked on the specified condition variable.

	IS	monitor	signaller	
1		TW_1		enter; C_i false
2		TW_1		signal all IS
3	TW_1			wait IS
4	TW_1	TW_2		enter; C_i false
5		TW_2	TW_1	signal all IS
6	TW_2	TW_1		wait IS;
7	TW_2	TW_1		unblock; C_i false
8	TW_2 TW_1			wait IS
9	TW_2 TW_1	TS_3		enter; make C_i true
10		TS_3	TW_2 TW_1	signal all; return
11		TW_2	TW_1	unblock; C_i true; make C_i false
12		TW_2	TW_1	signal all; return
13		TW_1		unblock; C_i false
14	TW_1			wait IS

Figure 9.23: Starvation/Staleness PNB

Hence, the entering task checks its predicate first on the next checking cycle. In Figure 9.23, task TW_2 acquires the resource ahead of task TW_1. If task TW_4 arrives and steps 4–14 are repeated, task TW_1 is in a starvation scenario.

Figure 9.24 shows one possible starvation/staleness/freshness scenario for the PB simulation. The failing PB scenario is more complex, requiring two tasks to enter and wait before the problem occurs. As above, the entering task does not end up at the end of the waiting condition IS nor at the head; instead, it appears second from the head, and the task previously at the tail is at the head. The entering task gets into the wrong position because the blocking signal puts it at the head of the signaller queue list, and similarly, the other waiting tasks block behind it if their predicates are false or there is no predicate check before signalling. As a result, the entering task is unblocked first from the signaller queue, and hence, performs its predicate check ahead of the other waiting tasks for the next predicate check. The last waiting task gets into the wrong position because it signals an empty condition (IS), which does nothing, and hence, it is the first to wait on IS, e.g. (see also lines 11–21 in Figure 9.24):

	IS	monitor	signaller	
1	A B C	D		enter; C_i false
2	A B C	D		signalblock IS
3	B C	A	D	unblock; signalblock IS
4	C	B	D A	unblock; signalblock IS
5		C	D A B	unblock; signalblock IS
6	C	D	A B	C_i false; wait IS
7	C D	A	B	C_i false; wait IS
8	C D A	B		C_i false; wait IS
9	C D A B			

Hence, the previous tail of the waiting tasks checks its predicate first on the next checking cycle. In Figure 9.24, task TW_2 acquires the resource ahead of task TW_1. If task TW_5 arrives and steps 11–34 are repeated, task TW_1 is in a starvation scenario.

Hence, for these simulations, staleness and/or freshness could occur for different applications because the tasks do not recheck their conditions in FIFO order. These simulation algorithms, which on the surface seem to provide the desired semantics, may fail because of subtle execution behaviour of the explicit-signal monitor used for the implementation or because of small deficiencies in the simulation algorithm itself.

9.12.2 FIFO Simulation

If FIFO testing of predicates is unimportant, the previous simulations are sufficient. However, many programmers assume (rightly or wrongly) that FIFO testing occurs, and in some cases, FIFO testing is required for correctness. It is often argued that if FIFO ordering is required, it is the programmer's responsibility to code it. However, adding FIFO

	IS	monitor	signaller	
1		TW_1		enter; C_i false
2		TW_1		signalblock IS
3	TW_1			wait IS
4	TW_1	TW_2		enter; C_i false
5	TW_1	TW_2		signalblock IS
6		TW_1	TW_2	unblock; C_i false
7		TW_1	TW_2	signalblock IS;
8	TW_1	TW_2		wait IS
9	TW_1	TW_2		unblock; C_i false
10	TW_1 TW_2			wait IS
11	TW_1 TW_2	TW_3		enter; C_i false
12	TW_1 TW_2	TW_3		signalblock IS
13	TW_2	TW_1	TW_3	unblock; C_i false
14	TW_2	TW_1	TW_3	signalblock IS
15		TW_2	TW_3 TW_1	unblock; C_i false
16		TW_2	TW_3 TW_1	signalblock IS
17	TW_2	TW_3	TW_1	wait IS
18	TW_2	TW_3	TW_1	unblock; C_i false
19	TW_2 TW_3	TW_1		wait IS
20	TW_2 TW_3	TW_1		unblock; C_i false
21	TW_2 TW_3 TW_1			wait IS
22	TW_2 TW_3 TW_1	TS_4		enter; make C_i true
23	TW_2 TW_3 TW_1	TS_4		signalblock IS (begin return)
24	TW_3 TW_1	TW_2	TS_4	unblock; C_i true; make C_i false
25	TW_3 TW_1	TW_2	TS_4	signalblock IS (begin return)
26	TW_1	TW_3	TS_4 TW_2	unblock; C_i false
27	TW_1	TW_3	TS_4 TW_2	signalblock IS
28		TW_1	TS_4 TW_2 TW_3	unblock; C_i false
29		TW_1	TS_4 TW_2 TW_3	signalblock IS
30	TW_1	TS_4	TW_2 TW_3	wait IS
31	TW_1	TS_4	TW_2 TW_3	return
32	TW_1	TW_2	TW_3	return
33	TW_1	TW_3		unblock; C_i false
34	TW_1 TW_3			wait IS

Figure 9.24: Starvation/Staleness PB

ordering on top of an existing automatic-signal implementation may be difficult or impossible because of the need to control ordering sufficiently to ensure correctness.

The following simulation guarantees waiting tasks have their predicates checked in FIFO order for each particular waituntil. That is, ordering is not FIFO with respect to calling the monitor but *with respect to each execution of a* waituntil *statement during a single access to a monitor*. This definition requires a task with a true predicate to first check other waiting tasks, which handles the case where two or more tasks wait on the same or equivalent predicates multiple times during a single monitor access, as in:

```
waituntil( x == y ); // first wait
waituntil( x == z ); // second wait
```

Assume a task sets y and z to the value of x, exits the monitor, and there are two tasks waiting on the first waituntil. FIFO order selects the longest waiting task to check its predicate; this task restarts and immediately checks the second equivalent predicate, which is true. However, the other task waiting for the first equivalent predicate is now waiting longer by the FIFO definition, so even though the checking task has a true predicate, it must be placed on the end of the FIFO queue to allow the other task to discover its first predicate is true. Only if all waiting tasks have false predicates

PAS	PNB – N Queue
declarations	**struct** IS_NODE_T : **public** seqable { condition cond; }; **struct** IS_T { sequence<IS_NODE_T> waiting; IS_NODE_T *cursor; } IS
waituntil C_i;	**if** (! IS.waiting.empty() || ! C_i) { IS.cursor = IS.waiting.head(); IS_NODE_T IS_node; IS.waiting.addTail(&IS_node); **for** (;;) { *// checking cycle* **if** (IS.cursor != 0) signal IS.cursor->cond; wait IS_node.cond; **if** (C_i) **break**; IS.cursor = IS.waiting.succ(&IS_node); } IS.waiting.remove(&IS_node); }
return $expr_{opt}$;	**if** (! IS.waiting.empty()) { IS.cursor = IS.waiting.head(); signal IS.cursor->cond; } **return** $expr_{opt}$;

Table 9.7: Transform PAS Monitor to PNB Monitor – N Queue

can the task with the true predicate continue. This case is subtle but important; however, it has a performance impact. Finally, for languages that allow side-effects in expressions, such as C++ and Java, it is important to only evaluate predicates when necessary in a simulation. The presented simulation performs the minimal number of predicate evaluations.

Table 9.7 is the transformation from a priority general-automatic-signal monitor to a priority nonblocking monitor using a linked list of condition variables, and any number of counters and flag variables. Each waiting task blocks on its own condition; hence, there is only one task blocked on each condition variable. The list order defines the FIFO relationship among waiting tasks.

The waituntil of the automatic-signal monitor is transformed into a check if no waiting tasks and the predicate expression is true, and if so, continue. Otherwise, a cursor used to traverse the FIFO linked-list is set to the head of the list, and a node with a condition variable is created and inserted at the end of the FIFO linked-list. The order of these two operations is important because the next operation needs to know if the list was empty before the new node was added.[6] Also, the list node is allocated each time a task waits, which could be done once on entry. On return from a monitor routine, if the list of waiting tasks is not empty, the cursor used to traverse the FIFO linked-list is set to the head of the list, one waiting task is signalled to start a daisy-chain checking-cycle, and the returning task exits directly. The check for an empty list is necessary because the signal accesses the list through the cursor, which is NULL for an empty list.

The main body of the waituntil code is a loop performing the checking cycle. A waiting task executes this loop until its predicate is true. The first action in the loop is to check if the cursor is not NULL (0), which implies there are other tasks to signal in this checking cycle so the next task on the FIFO list of conditions is signalled (via cursor) to continue the daisy-chain signalling. Now the task waits on the condition in its node until it is signalled to recheck its predicate. When a task unblocks to perform a check, it checks its predicate. If its predicate is true, the task exits the checking loop and proceeds to use the monitor. Otherwise, the cursor is moved to the next node of the linked-list, the

[6] Again, no dynamic allocation is performed by this algorithm, as the list node is created as a local variable on the task's stack, which eliminates the need for dynamic storage-allocation.

next waiting task is signalled, and this task waits again.

The predicate check at the start of the waituntil simulation ensures a task with a true predicate cannot proceed if there are any waiting tasks. In this case, the task with the true predicate places itself at the back of the waiting list, and a complete check is performed (as for a false predicate). If no other task has a true predicate, the original task with the true predicate is guaranteed to unblock as it is the last node on the list. Whereupon, it rechecks its true predicate and continues.

During the checking cycle, each task moves the cursor if its predicate is false, and this process continues through all nodes of the FIFO list, ensuring every waiting task checks its predicate and the predicates are checked in FIFO order. Because the last task on the linked-list does *not* signal a task before waiting, the checking cycle is guaranteed to stop as there can be no task on the signaller queue to continue further checking. Because daisy-chain signalling is used, at most one task appears on the signaller queue. This can be seen by observing that signals appear directly or indirectly before a **return** or wait, hence no other signal can occur before the signalled task restarts. Hence, during the checking cycle when a task finds its predicate is true, it proceeds to use the monitor with a guarantee of an empty signaller-queue due to the daisy-chain signalling. Therefore, at the next unlocking point, all waiting tasks are blocked on their specific condition variable in the FIFO queue before starting the checking cycle. The algorithm is $O(n)$ because for each of the n tasks on the linked-list, only one predicate check is performed. Also, this simulation is optimal with respect to minimizing context switching, as no additional searching is required to process waiting tasks in FIFO order, and no flushing is required to maintain FIFO order, both of which introduce additional context switching for the waiting tasks.

9.13 Monitor Comparison

The following is a summation of the criteria a programmer can use in selecting a particular kind of monitor to solve a concurrency problem. (Although, in most cases, the language designer has already selected the kind of monitor(s), and a programmer must conform to it.)

9.13.1 Priority/No-Priority

When a condition is signalled in a priority monitor, control transfers directly or indirectly to a task waiting for the condition to occur (unless the condition variable is empty). In either case, this transfer allows the signaller and signalled tasks to establish an internal protocol for synchronization and communication through monitor variables (i.e., cooperation). In contrast, a signal in a no-priority monitor, as in the programming languages Mesa, Modula-2+, Modula-3[7] and Java, act as a “hint” to a waiting task to resume execution at some convenient future time [LR80, p. 111][Nel91, p. 102][Lea97, p. 93]. In this approach, a task may signal a condition whenever it *might* be true; the signalled task is responsible for checking whether it actually *is* true. This style is a cautious approach to concurrency control, where the signalled and signaller tasks can make no assumptions about order of execution, even within the monitor. Hence, for no-priority monitors, the assertion for the condition that is signalled may no longer hold by the time the signalled task gains control of the monitor, making cooperation difficult or impossible. Thus, when a waiting task restarts, it must determine if the monitor state it was waiting for is true instead of assuming it is true because it was signalled. This lack of guarantee often results in a wait being enclosed in a **while** loop so a task rechecks if the event for which it was waiting has occurred (as is done implicitly in the automatic-signal monitor). This coding style is busy waiting as there is no bound on the number of times a tasks may awake, only to wait again.

As well, it is difficult to implement certain scheduling schemes using no-priority monitors because a calling task can *barge* ahead of tasks that have already been waiting in the monitor. Unless special care is taken by including extra code to deal with this situation, it is impossible to guarantee FIFO scheduling, which may be critical to a particular problem (e.g., readers and writer problem). In such cases, it is necessary to simulate a priority monitor to prevent barging, which increases both the complexity of the solution and the execution cost.

Finally, there is the potential for unbounded waiting in no-priority monitors. If the implementation chooses randomly among internal queues of the same priority, there is no bound on the number of tasks that are serviced before a task is selected from a particular queue. In practice, however, this problem is usually solved by the implementation, which combines queues with the same priority so that tasks placed on them have a bounded number of tasks ahead of

[7] Modula-3 takes this one step further by defining the signal routine to wake up at least one task, which implies that it may wake up more than one task. This semantics is modelled in the taxonomy as a no-priority non-blocking monitor with a loop around each signal that executes a random number of times.

them. Thus, the only advantage of a no-priority monitor is that a task does not have to wait as long on average before entering the monitor.

9.13.2 Blocking/Non-blocking

A blocking signal guarantees control goes immediately to the signalled task. This direct transfer is conceptually appealing because control transfers to a task waiting for a particular assertion to become true; thus, the signaller does not have an opportunity to inadvertently falsify the assertion that was true at the time of the signal. However, it is possible for the signalled task to incorrectly alter the monitor state prior to resumption of the signaller. This kind of signal often results in monitor programs in which a signaller delegates all responsibility for clean up and for further signalling to the signalled, e.g., daisy-chain signalling.

A non-blocking signal guarantees control continues with the signaller task. This non-transfer is conceptually appealing because it is what most programmers naturally think of; thus, the signaller is obligated to establish the proper monitor state only when the monitor is unlocked as the monitor state at the time of the signal is irrelevant. However, it is possible with multiple signalling for one of the signalled tasks to incorrectly alter the monitor state for the other signalled tasks. This kind of signal often results in a monitor where a signaller takes on the responsibility for clean up and for signalling other tasks that can execute, e.g., multiple signalling.

The main differences between the two kinds of monitors are performance and coding problems. A major performance issue stems from the fact that signals are often the last operation of a mutex member. In this case, it is important for the signaller to exit the monitor immediately so it can continue concurrently outside the monitor with the signalled task inside the monitor, which is exactly the behaviour of the non-blocking monitor. However, it is impossible to simulate this behaviour with a blocking signal because the signaller blocks before the return can occur, which results in the unnecessary cost of blocking the signaller and the inhibited concurrency while the signaller needlessly waits for the signalled task to run. This situation is exacerbated by daisy-chain signalling because no task leaves the monitor until the last task in the chain is signalled and leaves the monitor. Hence, the blocking signal makes inefficient the most common monitor idiom, which is poor design. It might be possible to mitigate the problem of a blocking signal before a return by having the compiler optimize that signal into a non-blocking signal, but this changes the semantics of the monitor.

Two coding problems between non-blocking and blocking signal have already been illustrated in the previous monitor transformations. The transformations show how subtle issues in the signalling semantics can result in an infinite loop. The following are additional examples of coding problems between blocking and non-blocking kinds of monitors. In the blocking case, the signalled task cannot restart the signaller task from a condition on which it will eventually wait because the signaller is currently on the signaller queue. For example, the straightforward approach to communicating data to and from task A and B in a monitor:

Task A	Task B
msg = ...	**wait** MsgAvail
signal MsgAvail	print msg
wait ReplyAvail	reply = ...
print reply	**signal** ReplyAvail

fails for a blocking monitor because task B's signal of condition ReplyAvail is lost because task A has not yet blocked on condition ReplyAvail because it is on the acceptor/signalled stack. In the non-blocking case, the signaller blocks on the condition *before* the signalled task starts so the signalled task knows that it can restart its signaller if that is appropriate. The opposite problem occurred in the dating service solution:

Boy Task	Girl Task
boyPhoneNo = phoneNo	**wait** Girl
signal Girl	girlPhoneNo = phoneNo
print girlPhoneNo	print boyPhoneNo

where the straightforward solution fails for a non-blocking monitor because the Boy task does not wait for the Girl task to restart and copy her phone number into the communication variable. In the blocking case, the signaller blocks on the acceptor/signalled stack *before* accessing the communication variable that is set by the signalled task.

9.13.3 Quasi-blocking

Because the order of execution of the signaller and signalled tasks is unknown, cooperation among tasks can be difficult to establish. Therefore, this kind of monitor is not particularly useful, even though it appears implicitly in some situations such as an operating system kernel.

9.13.4 Extended Immediate Return

The immediate-return monitor was invented to optimize the signal–return case that occurs frequently in monitors at the cost of restricting the ability to write certain monitor algorithms. The extended immediate-return monitor allows any monitor to be written, with the restriction that a signal must appear before a wait or return. The restrictions on signal placement allow efficient implementation of a monitor. However, the restrictions also mandate a particular coding style, signals before waits and returns, which may unnecessarily complicate the coding of a monitor and obscure its algorithm.

9.13.5 Automatic Signal

Unfortunately, the general automatic-signal monitor becomes expensive when the number of tasks in the monitor is large. However, restricted automatic-signal monitors are competitive with explicit-signal monitors, especially when there are only a few condition variables, because they depend only on the number of conditional expressions and not the number of tasks in the monitor. Yet, the restricted automatic-signal monitor's conditional expressions cannot involve local variables or parameters of a request, e.g., a disk-scheduling algorithm where requests are serviced in track number rather than request arrival order. Hence, there is a class of important problems that cannot be handled by a restricted automatic-signal monitor.

Where the automatic-signal monitor excels is in prototyping a concurrent system. It is usually simpler to ignore cooperation during the initial stages of development and concentrate on the logical properties necessary for correctness. The conditional expressions of an automatic-signal monitor capture these logical properties. After the system is functioning, performance problems can be addressed by converting those monitors involved in bottlenecks into explicit-signal monitors, using appropriate cooperation.

9.14 Summary

Concurrency constructs are crucial to sound and efficient concurrency because the compiler is aware the program is concurrent. Furthermore, high-level concurrency constructs can substantially reduce the complexity of solving a concurrent problem, e.g., readers and writer. The first concurrency construct was the critical region for handling mutual exclusion, followed immediately by the conditional critical-region for handling both synchronization and mutual exclusion. While primitive, these constructs paved the way for the first successful (and object-oriented) concurrency construct, the monitor. A monitor combines concurrency capabilities, such as mutual exclusion and synchronization, along with software engineering capabilities, such as abstraction and encapsulation. Since its inception, several different kinds of monitors have been suggested and implemented in different programming languages. The fundamental difference among the kinds of monitors is directly related to the form of signalling used in synchronization, be it implicit or explicit. Priority monitors are demonstrably better than no-priority monitors for constructing complex cooperation among tasks. Both non-blocking and blocking signal are useful, but non-blocking is more powerful because it efficiently handles the common case of a signal as the last action of a mutex member. Without a clear understanding of the signalling differences among monitors, it is very easy to make errors using them.

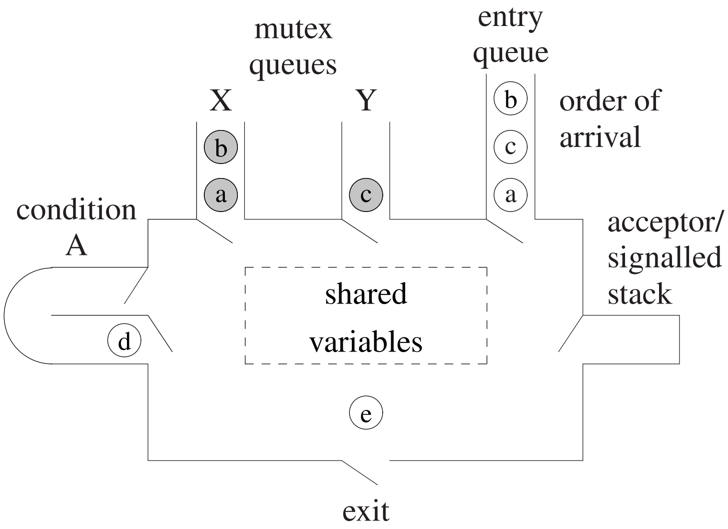
9.15 Questions

1. Some programming language systems have only a `P` and `V` operation while others have only a monitor construct. Yet either system can be used to solve the same problems in essentially the same way. How do we know this is true?

2. Figure 9.19, p. 282 shows a semaphore written using a monitor, with either external or internal scheduling. If the semaphore is extended with member `P(s)`, which atomically `V`s the argument semaphore, `s`, and then blocks the calling task if necessary (see page 216 for details), show that it cannot be written with external scheduling.

3. Explain why the "wait *condition–variable*" of a monitor cannot be implemented with just a P(*condition–variable*) and V(*monitor–lock*) operation?

4. Consider the Hoare monitor:



   - which task(s) is running?
   - which task(s) is ready to enter the monitor?
   - which task(s) is blocked?
   - which task(s) has executed a WAIT? On what condition variable?
   - If task e becomes blocked on the acceptor/signalled stack, does this imply that task d immediately enters the monitor? Explain.
   - Suppose tasks d and e complete. Which task next enters the monitor?

5. Explain the difference between external and internal scheduling, and give two situations that preclude using external scheduling.

6. Explain why automatic-signal monitors are easier to use than explicit signal monitors but more expensive in terms of execution time.

7. Write a monitor (semaphores are not allowed) that implements the following kind of access to a resource. There are two kinds of tasks, B and G, which are trying to get access to a resource. To get access to the resource, each task must call the StartDating routine in the monitor, passing it an appropriate value for a B task and an appropriate value for a G task. When a task is finished with the resource, it must call EndDating to release the resource. The monitor must guarantee that a single B task and a single G task start sharing the resource together (but not necessarily at the exact same instant). No other pair of B-G tasks can start sharing the resource until the previous pair has finished. Either partner of the pair using the resource can leave at any time after they have started together. That is, if the B task leaves, do not start another B; wait for the partner G to leave and then start a new pair. You must guarantee that starvation of a particular B or G task does not occur assuming that an appropriate partner will always appear.

   To test the monitor, write two tasks, called Boy and Girl, that each start dating and then delay for a random number of times between 1-5 inclusive (use yield()). The body of these task's main member looks like:

```
StartDating();
// verify that the monitor is working correctly
// delay for a random period
// verify that the monitor is working correctly
EndDating();
```

The check may require using shared variables between the monitor and the Boy and Girl tasks. Start 30 of the Girl and Boy tasks.

Notice the following scenario is possible. A Boy task is signalled, it charges out of the monitor, flys through the resource, and charges back into the monitor before the Girl task of the pair has even been signalled. Nevertheless, the monitor must fulfill its mandate of pairing and release the Girl task and wait for it to finish with the resource before another pair of tasks is released. So the Girl task may get to the resource and discover that she has been stood up for the date (sigh), even though the matching-making monitor arranged a date. That's not the monitor's problem. Yes, something could be done about this, but NOT for this assignment.

Before and after a process delays, see if you can devise some check that verifies that the monitor is working correctly (e.g., similar to the check performed in the routine CriticalSection). You are allowed to use a global variable(s) in this check since it is strictly for verifying the correctness of the monitor, and would be removed after testing. It is possible to do the testing without globals by passing the address of the test variables but since they will ultimately be removed, the global variables is the cleanest testing facility.

8. Write a simple "alarm clock" monitor that tasks can use to put themselves to sleep until a certain time. The monitor measures time in arbitrary time units called "ticks". The clock should be initialized to 0 ticks.

   The clock monitor should have four entry routines:

```
_Monitor Clock {
    friend class ticker;

    void tick() {
        // Used only by a clock task to tell the monitor that a tick
        // has elapsed.
    }
  public:
    int ctime() {
        // This routine returns the current time in ticks.
    }
    void sleep_until( int t ) {
        // This routine causes the calling task to sleep until time t is
        // reached. If time t is already passed, the task does not sleep.
    }
    void sleep_for( int n ) {
       // This routine causes the calling task to sleep for n ticks.
       // If n < 0, the task does not sleep.
    }
};
```

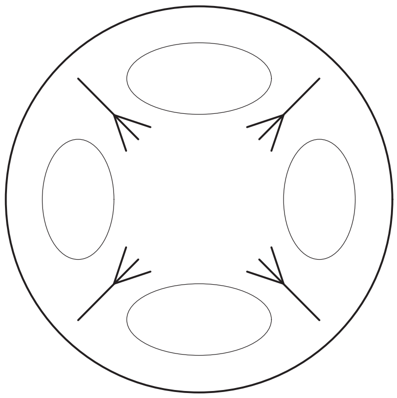
   The monitor should be designed to be used in a situation where there is no limit on the number of sleeping tasks. The monitor can use any data structure appropriate to handle any number of sleeping tasks.

   In a real system, clock hardware might generate an interrupt that would be used to call tick(). In your program, start a task that calls tick() at regular intervals; something similar to the following is adequate:

```
_Task ticker {
    Clock &clock;

    void main() {
        for ( ;; ) {
            _Accept( ~ticker ) {
                break;
            } else {
                _Timeout( uDuration( 0, 1000 ) );
                clock.tick();
            } // _Accept
        } // for
    }
  public:
    ticker( Clock &clock ) : clock(clock) {}
};
```

Use the Clock monitor in a program that creates tasks that sleep for varying amounts of time. The output of the program should show that the tasks wake up at the right times.

9. A group of N (N $>$ 1) philosophers plans to spend an evening together eating and thinking (Dijkstra). Unfortunately, the host philosopher has only N forks and is serving a special spaghetti that requires 2 forks to eat. Luckily for the host, philosophers (like university students) are interested more in knowledge than food; after eating a few bites of spaghetti, a philosopher is apt to drop her forks and contemplate the oneness of the universe for a while, until her stomach begins to growl, when she again goes back to eating. So the table is set with N plates of spaghetti with one fork between adjacent plates. For example, the table would look like this for N=4:



Consequently there is one fork on either side of a plate. Before eating, a philosopher must obtain the two forks on either side of the plate; hence, two adjacent philosophers cannot eat simultaneously. The host reasons that, when a philosopher's stomach begins to growl, she can simply wait until the two philosophers on either side begin thinking, then *simultaneously pick up the two forks on either side of her plate* and begin eating. If a philosopher cannot get both forks immediately, then she must wait until both are free. (Imagine what would happen if all the philosophers simultaneously picked up their right forks and then waited until their left forks were available.)

The table manages the forks and must be written as a:

(a) class using μC++ semaphores to provide mutual exclusion and synchronization.

(b) μC++ monitor using internal scheduling.

(c) μC++ monitor that simulates an automatic-signal monitor.

(d) Java monitor using internal scheduling. Hint, use an approach similar to the automatic-signal monitor. You may not use java.util.concurrent available in Java 1.5.

Each μC++ table implementation has a different interface (you may add only a public destructor and private members):

```
#if defined( TABLETYPE_SEM )              // semaphore solution
// includes for this kind of table
class Table {
    // private declarations for this kind of table
#elif defined( TABLETYPE_INT )            // internal scheduling monitor solution
// includes for this kind of table
_Monitor Table {
    // private declarations for this kind of table
#elif defined( TABLETYPE_AUTO )           // automatic-signal monitor solution
// includes for this kind of table
_Monitor Table {
    // private declarations for this kind of table
#else
    #error unsupported table
#endif
    // common declarations
  public:                                 // common interface
    Table( unsigned int noOfPhil, Printer &prt );
    void pickup( unsigned int id );
    void putdown( unsigned int id );
};
```

where the preprocessor is used to conditionally compile a specific interface. This form of header file removes duplicate code. An appropriate preprocessor variable is defined on the compilation command using the following syntax:

```
u++ -DTABLE_SEM -c TableSEM.cc
```

Member routines pickup and putdown are called by each philosopher, passing the philosopher's identifier (value between 0 and $N-1$), to pick up and put down both forks, respectively. Member routine pickup does not return until both forks can be picked up. To simultaneously pick up and put down both forks may require locking the entire table for short periods of time. No busy waiting is allowed; use cooperation among philosophers putting down forks and philosophers waiting to pick up forks. Your solution does not have to preclude starvation; it is sufficient that there is a finite number of noodles to be eaten. Finally, the approach used in the first two implementations of the table monitor should be consistent; you should not have completely different algorithms in these implementations. Only the automatic-signal and Java implementations should be different from the explicit signal because there is no (or little) cooperation.

For the automatic-signal monitor, create an include file, called AutomaticSignal.h, which defines the following preprocessor macros:

```
#define AUTOMATIC_SIGNAL ...
#define WAITUNTIL( pred, expr... ) ... // gcc variable number of parameters
#define RETURN( expr... ) ...    // gcc variable number of parameters
```

Macro AUTOMATIC_SIGNAL is placed only once in an automatic-signal monitor as a private member, and contains any private variables needed to implement the automatic-signal monitor. Macro WAITUNTIL is used to delay until the pred evaluates to true; expr is optionally specified and evaluated each time pred evaluates to false and *before* the task delays (waits). Macro RETURN is used to return from a public routine of an automatic-signal monitor, where expr is optionally used for returning a value. For example, a bounded buffer implemented as an automatic-signal monitor looks like:

```
_Monitor BoundedBuffer {
    AUTOMATIC_SIGNAL;
    int front, back, count;
    int Elements[20];
  public:
    BoundedBuffer() : front(0), back(0), count(0) {}
    _Nomutex int query() { return count; }

    void insert( int elem ) {
        WAITUNTIL( count < 20 );
        Elements[back] = elem;
        back = ( back + 1 ) % 20;
        count += 1;
        RETURN();
    }

    int remove() {
        WAITUNTIL( count > 0 );
        int elem = Elements[front];
        front = ( front + 1 ) % 20;
        count -= 1;
        RETURN( elem );
    }
};
```

Make absolutely sure to *always* have a RETURN() macro at the end of each mutex member.

A philosopher eating at the table is simulated by a task, which has the following interface (you may add only a public destructor and private members):

```
_Task Philosopher {
  public:
    enum states { THINKING, HUNGRY, EATING, WAITING, FINISHED };
    Philosopher( unsigned int id, unsigned int noodles, Table &table, Printer &prt );
};
```

Start each philosopher in the hungry state with the number of noodles on their plate passed to the constructor. Each time a philosopher gains control of both forks, she eats a random number of noodles, between 1 and 5 inclusive (see rand). Delay a random number of times, between 0 and 4 inclusive, to simulate the time to eat the noodles by using yield to give up the CPU time slice. When a philosopher finishes eating, she puts both forks down and delays a random number of times, between 0 and 19 inclusive, to simulate the time to think, unless there are no more noodles at which time the philosopher terminates.

All output from the program is generated by calls to a printer coroutine, excluding error messages. The interface for the printer is (you may add only a public destructor and private members):

```
_Cormonitor Printer {
  public:
    Printer( unsigned int noOfPhil );
    void print( unsigned int id, Philosopher::states state );
    void print( unsigned int id, Philosopher::states state, unsigned int bite, unsigned int noodles );
};
```

(Since Java does not have a coroutine-monitor, the coroutine must be simulated.) A philosopher calls the print member when it enters states: thinking, hungry, eating, waiting, finished. The table calls the print member *before* it blocks a philosopher that must wait for its forks to become available. The printer attempts to reduce output by buffering information for each philosopher until one of the stored elements is overwritten. Output should look similar to that in Figure 9.25. Each column is assigned to a philosopher with an appropriate title, e.g., "Phil0". A column entry is the state transition for that philosopher, containing one of the following states:

```
% phil 5 20
Phil0  Phil1  Phil2  Phil3  Phil4
****** ****** ****** ****** ******
       H*     H
       E4,16
H      T*     H             H
H      H      H*     H      H
H      H*     E3,17  H      H
H      W1,2   E3,17* H      H
H      W1,2*  T      H      H
       E1,15
H      T      T*     H      H
H      T      H      H      H*
H      T*     H      H      E1,19
H      H      H      H      E1,19*
H*     H      H      H      T
E5,15  H      H*     H      T
E5,15* H      E5,12  H      T
T      H      E5,12  H*     T
T      H      E5,12* W3,4   T
T      H      T      W3,4*  T
                     E1,19
T      H      T      T      T*
T      H      T*     T      H
T      H*     H      T      H
T      E5,10  H*     T      H
T      E5,10* W2,3   T      H
T      T      W2,3*  T      H
T      T      E1,11  T*     H
T      T      E1,11* H      H
T*     T      T      H      H
H      T*     T      H      H
H      H      T      H*     H
H      H      T      E1,18  H*
H      H      T      E1,18* W4,0
H      H      T      T      W4,0*
H      H      T      T*     E1,18
H      H      T      H      E1,18*
H      H      T*     H      T
H*     H      H      H      T
E4,11
T      H      H*     H      T
T      H*     E1,10  H      T
T      W1,2   E1,10* H      T
              T
T      W1,2*  H      H      T
T      E5,5   H*     H      T
T      E5,5   W2,3   H      T*
                            H
T      E5,5   W2,3   H*     E5,13
T*     E5,5   W2,3   W3,4   E5,13
H      E5,5*  W2,3   W3,4   E5,13
H      T      W2,3*  W3,4   E5,13
H      T      E3,7   W3,4   E5,13*
H      T      E3,7*  W3,4   T
H      T      T      W3,4*  T
                     E3,15
H      T      T      T      T*
                            H
H      T      T      T*     E2,11
H      T      T      H      E2,11*
                            T
H      T*     T      H      H
H*     H      T      H      H
E3,8   H      T      H*     H
E3,8*  H      T      E2,13  H
T      H      T      E2,13* H
T      H      T*     T      H
              H
T      H      E2,5   T      H*
T      H      E2,5*  T      E4,7
T      H*     T      T      E4,7
       E2,3
T      T      T      T      E4,7*
                            T
T      T*     T      T      H
T*     H      T      T      H
H      H      T      T*     H
H      H*     T      H      H
       E3,0
...    F      ...    ...    ...
H             T      H      H*
H             T*     H      E5,2
H             H      H      E5,2*
                            T
H*            H      H      H
E5,3          H      H*     H
E5,3          H      E4,9   H*
E5,3*         H      E4,9   W4,0
T             H      E4,9*  W4,0
T             H      T      W4,0*
T             H      T*     E2,0
T             H*     H      E2,0
T*            E2,3   H      E2,0
H             E2,3*  H      E2,0
              T
H             H      H*     E2,0
                     E4,5
H             H      T      E2,0*
...    ...    ...    ...    F
H             H*     T
H*            E3,0   T
E2,1
T             E3,0*  T
...    ...    F      ...    ...
T                    T*
T*                   H
H                    H*
                     E4,1
                     T
                     H
H*                   E1,0
E1,0                 E1,0*
...    ...    ...    F      ...
E1,0*
F      ...    ...    ...    ...
***********************
Philosophers terminated
```

Figure 9.25: Output for 5 Philosophers and 20 Noodles

State	Meaning
H	hungry
T	thinking
W l,r	waiting for the left fork l and the right fork r to become free
E n,r	eating n noodles, leaving r noodles on plate
F	finished eating all noodles

Identify the left fork of philosopher$_i$ with number i and the right fork with number $i + 1$. For example, W2,3 means forks 2 and 3 are unavailable, so philosopher 2 must wait, and E3,29 means philosopher 1 is eating 3 noodles, leaving 29 noodles on their plate to eat later. When a philosopher finishes, the state for that philosopher is marked with F and all other philosophers are marked with `"..."`.

Buffered data is printed when existing data in the buffer is going to be overwritten. If a philosopher performs consecutive prints, then only the buffer data for that philosopher is printed (excluding the asterisk) and the other columns are empty. Otherwise, the entire buffer is flushed. Note, the buffer is not cleared by a flush so previously stored values appear when the next task flushes the buffer. An asterisk * appears next to the state that is about to be overwritten, when the buffer is flushed. In general, the * on line i appears above the value on line $i + 1$ that caused the buffer to be flushed. After a philosopher has finished, no further output appears in that column. All output spacing can be accomplished using the standard 8-space tabbing, so it is unnecessary to build and store strings of text for output.

In addition, you are to devise and include a way to test your program for erroneous behavior. This testing should be similar to the check performed in the routine CriticalSection, from software solutions for mutual exclusion, in that it should, with high probability, detect errors. When a problem is detected, use μC++'s uAbort routine to halt the program with an appropriate message. (HINT: once a philosopher has a pair of forks, what must be true about the philosophers on either side?)

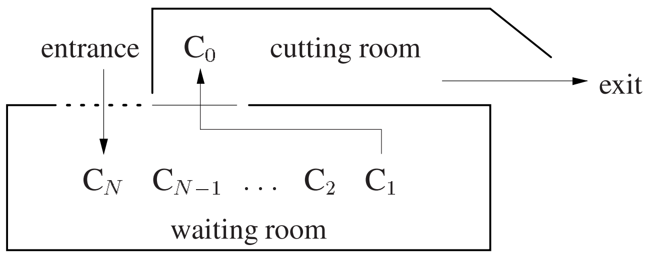
The executable program is named phil and has the following shell interface:

```
phil [ P [ N ] ]
```

P is the number of philosophers; if P is not present, assume a value of 5. N is the number of noodles per plate; if N is not present, assume a value of 30. When present, assume each argument is a valid integer value, but ensure the number of philosophers is greater than 1 and number of noodles is greater than 0; print an appropriate usage message and terminate the program if either value is outside its range. The program must handle an arbitrary number of philosophers and noodles, but we will only test with values less than 10 and 100 for the two parameters, so the output tabbing works correctly.

(WARNING: in GNU C, -1 % 5 != 4, and on UNIX, the name fork is reserved.)

10. The sleeping barber is a problem proposed by E. W. Dijkstra. A barber shop has a cutting room with one chair and a waiting room with N chairs.



Customers enter the waiting room one at a time if space is available; otherwise they go to another shop (called balking). The waiting room has an entrance, and next to the entrance is the cutting room with the barber's chair; the entrances to the waiting and cutting room share the same sliding door, which always closes one of them. The sliding door ensures that when the barber opens the door to see if anyone is waiting, a new customer cannot simultaneously enter the room and not be seen.

When the barber has finished a haircut, he opens the door to the waiting room and checks for waiting customers. If the waiting room is not empty, he invites the next customer (first-in first-out (FIFO) service order) for a haircut, otherwise he goes to sleep in the barber's chair. Customers may enter the waiting room at any time. If a customer finds the waiting room is not empty, they wait their turn in the waiting room; if the waiting room is empty, the customer opens the door to the cutting room to see if the barber is there but sleeping. If the barber is not there, the customer sits down in the waiting room and waits for the barber to appear; otherwise, the customer wakes the barber.

Implement the barber shop as a:

(a) class using μC++ semaphores to provide mutual exclusion and synchronization.

(b) μC++ monitor using internal scheduling.

(c) μC++ monitor that simulates an automatic-signal monitor.

(d) Java monitor using internal scheduling. Hint, use an approach similar to the automatic-signal monitor. You may not use java.util.concurrent available in Java 1.5.

The μC++ semaphore and internal scheduling implementations must service waiting customers in FIFO order. The μC++ automatic-signal monitor and Java internal scheduling implementations do not have to service waiting customers FIFO order. The interface for the barber shop is (you may add only a public destructor and private members):

```
#if defined( BARBERSHOPTYPE_SEM )        // semaphore solution
// includes for this kind of barber shop
class BarberShop {
    // private declarations for this kind of barber shop
#elif defined( BARBERSHOPTYPE_INT )      // internal scheduling monitor solution
// includes for this kind of barber shop
_Monitor BarberShop {
    // private declarations for this kind of barber shop
#elif defined( BARBERSHOPTYPE_AUTO )   // automatic-signal monitor solution
// includes for this kind of barber shop
_Monitor BarberShop {
    // private declarations for this kind of barber shop
#else
   #error unsupported barber shop
#endif

    // all common declarations
  public:
    BarberShop( Printer &prt, const int MaxWaitingCust );
    bool enterShop( int id );                 // called by customer
    int startCut();                           // called by barber
    void endCut();                            // called by barber
};
```

where the preprocessor is used to conditionally compile a specific interface. You can define the appropriate preprocessor variable on the compilation command using the following syntax:

```
u++ -DBARBERSHOPTYPE_INT -c BarberShopINT.cc
```

A customer calls the enterShop member to enter the barber shop. This member returns true if the customer gets a haircut, and false if the waiting room is full. The barber calls the startCut member to obtain the next customer; if there are no customers, the barber goes to sleep (blocks). startCut returns the customer identifier for a waiting customer. The barber calls endCut after completing the haircut. The automatic-signal implementation should be slightly different from the explicit signal because there is no (or little) cooperation.

For the automatic-signal monitor, create an include file, called AutomaticSignal.h, which defines the following preprocessor macros:

```
#define AUTOMATIC_SIGNAL ...
#define WAITUNTIL( pred ) ...
#define RETURN( expr... ) ...   // gcc variable number of parameters
```

Macro AUTOMATIC_SIGNAL is placed only once in an automatic-signal monitor as a private member, and contains any private variables needed to implement the automatic-signal monitor. Macro WAITUNTIL is used to delay until the pred evaluates to true. Macro RETURN is used to return from a public routine of an automatic-signal monitor, where expr is optionally used for returning a value. For example, a bounded buffer implemented as an automatic-signal monitor looks like:

```
_Monitor BoundedBuffer {
    AUTOMATIC_SIGNAL;
    int front, back, count;
    int Elements[20];
  public:
    BoundedBuffer() : front(0), back(0), count(0) {}
    _Nomutex int query() { return count; }

    void insert( int elem ) {
        WAITUNTIL( count < 20 );
        Elements[back] = elem;
        back = ( back + 1 ) % 20;
        count += 1;
        RETURN();
    }

    int remove() {
        WAITUNTIL( count > 0 );
        int elem = Elements[front];
        front = ( front + 1 ) % 20;
        count -= 1;
        RETURN( elem );
    }
};
```

Make absolutely sure to *always* have a RETURN() macro at the end of each mutex member.

The interface for the barber is (you may add only a public destructor and private members):

```
_Task Barber {
    void main();
  public:
    Barber( Printer &prt, BarberShop &bs, int CutDelay );
};
```

The barber begins by randomly yielding between 0-CutDelay times so it may not arrive before the first customers start entering the barber shop. (Yielding is performed by calling task member yield.) After indicating arrival at the shop, the barber begins the following steps:

a) indicate about to get a customer
b) call startCut in the barber shop to get customer
c) indicate sleeping if no customer waiting
d) indicate a customer is no longer in the waiting room
e) indicate starting to cut hair
f) randomly yield between 0-CutDelay times to simulate haircut
g) call endCut in the barber shop to indicate ending of haircut

The barber terminates when he receives a customer identifier of -1 from startCut.

The interface for a customer is (you may add only a public destructor and private members):

```
_Task Customer {
    void main();
  public:
    Customer( Printer &prt, BarberShop &bs, int id, int CustDelay );
};
```

A customer begins by randomly yielding between 0-CustDelay times so the customers arrive at the barber shop at random times. A customer makes a single call to enterShop in the barber shop to get a haircut. After the call, the customer indicates if it fails to receive a haircut because the waiting room is full.

All output from the program is generated by calls to a (special) printer coroutine, excluding error messages. The interface for the printer is (you may add only a public destructor and private members):

```
_Cormonitor Printer {
    void main();
  public:
    Printer( const int MaxWaitingCust );
    void Barber( char status );              // called by barber
    void CustomerCut();                      // called by barber
    void Customer( int id );                 // called by customer
};
```

The printer attempts to reduce output by condensing multiple barber state-changes along a single line. Figure 9.26 shows two example outputs from different runs of the program. The output consists of four columns:

(a) barber states

A arrival at the barber shop

G attempting to get a customer from the waiting room

S going to sleep in the barber chair waiting for a customer

C cutting a customer's hair

F leaving the barber shop

(b) customer having haircut

(c) customer balked because waiting room is full

(d) list of customers in the waiting room

All printing must occur in the coroutine main, i.e., no printing can occur in the coroutine's member routines. All output spacing can be accomplished using the standard 8-space tabbing, so it is unnecessary to build and store strings of text for output. Calls to the printer coroutine to perform printing may be performed in the barber-shop members or in the barber/customer tasks (you decide where to print).

uMain::main creates the printer, barber shop, barber and $NoOfCustomers$ customer tasks. After all the customer tasks have terminated, uMain::main calls enterShop with a customer identifier of -1.

The shell interface to the sleepingbarber program is as follows:

sleepingbarber [$MaxWaitingCust$ [$NoOfCustomers$ [$CutDelay$ [$CustDelay$]]]]

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) Where the meaning of each parameter is:

$MaxWaitingCust$**:** maximum number of customers that can wait in the barber shop waiting room, i.e., $MaxWaitingCust$ equals N chairs. The default value if unspecified is 5.

$NoOfCustomers$**:** number of customers attempting to obtain a haircut during a run of the program. The default value if unspecified is 20.

$CutDelay$**:** number of times the barber yields to simulate the time required to cut a customer's hair. The default value if unspecified is 2.

```
% sleepingbarber 3 20 5 100
Barber Chair  Balked Waiting
AGS
                     3
C      3
                     0
GC     0
                     7
                     7 17
                     7 17 14
G
              2
C      7             17 14
                     17 14 13
GC     17            14 13
                     14 13 8
              10
              15
GC     14            13 8
GC     13            8
                     8 6
                     8 6 9
GC     8             6 9
                     6 9 16
GC     6             9 16
                     9 16 18
              11
GC     9             16 18
GC     16            18
                     18 5
GC     18            5
GC     5
                     4
                     4 12
                     4 12 19
GC     4             12 19
GC     12            19
                     19 1
GC     19            1
GC     1
GS
                     -1
C      -1
F
```

```
% sleepingbarber 3 20 5 100
Barber Chair  Balked Waiting
                     6
                     6 12
AGC    6             12
GC     12
                     7
                     7 15
                     7 15 10
GC     7             15 10
                     15 10 1
GC     15            10 1
GC     10            1
                     1 19
G
                     1 19 14
C      1             19 14
                     19 14 4
GC     19            14 4
                     14 4 8
GC     14            4 8
GC     4             8
                     8 5
                     8 5 13
              17
              3
G
              16
              2
C      8             5 13
                     5 13 0
GC     5             13 0
GC     13            0
GC     0
GS
                     18
C      18
                     11
GC     11
                     9
GC     9
GS
                     -1
C      -1
F
```

Figure 9.26: Sleeping Barber: Example Output

$CustDelay$**:** number of times a customer yields *before* attempting entry to the barber shop. The default value if unspecified is 40.

Assume all arguments are valid positive integer values, i.e., no error checking is required.

11. A small optical disk drive can be modelled by the following monitor.

```
_Monitor DiskDrive {
    const char *data;
    int currentTrack;
    int tracksMoved;
  public:
    static const int NoOfTracks = 100, TrackSize = 20;

    DiskDrive() : data( "abcdefghijklmnopqrs" ) {
        currentTrack = 0;
        tracksMoved = 0;
    } // DiskDrive::DiskDrive

    void Seek( int track ) {
        int distance = abs( currentTrack - track );
        uThisTask().yield( distance );
        tracksMoved += distance;
        currentTrack = track;
    } // DiskDrive::Seek

    void Read( char *buffer ) {
        strcpy( buffer, data );
    } // DiskDrive::Read

    int Distance() {
        return tracksMoved;
    } // DiskDrive::Distance
}; // DiskDrive
```

This disk drive is read-only and has one hundred tracks, numbered 0 through 99, each of which stores only twenty characters (which, in this simulation, are always the same 20 characters). For your convenience, the last character on each track has the '\0' character.

The user interface consists of three members. Seek moves the read-head to a specified track and records the distance moved. Read copies the current track into a buffer. Distance returns the total distance that the head has traveled since the disk drive has been created.

Disk *access time* is the sum of the time to move the disk head to the requested track, the time for the disk to rotate to the desired sector on the track, and the time to transfer data to/from the sector. These times are named the *seek time*, *rotation delay* and *data transfer*, respectively. Because the time to move the head is proportional to the distance between tracks, and the rotation and transfer time combined is often less than the time to move the head just one track, the best way to reduce the average disk access time is to shorten the seek time by minimizing the head movement.

Write a monitor to **efficiently** schedule disk seek-time using the shortest seek time first (SSTF) algorithm. The SSTF algorithm picks, from all pending requests, the sequence of requests that are closest to the current head position in either direction. If the closest two requests in either direction are equal distance from the current position, the scheduler must pick the lower numbered track.

The user interface for the disk scheduler is:

```
_Monitor DiskScheduler {
  public:
    DiskScheduler( DiskDrive &disk );
    _Nomutex void ScheduleTrack( int track );
    void Done();
};
```

The disk parameter to the constructor is used to seek to the appropriate track before returning from ScheduleTrack. ScheduleTrack is *not* a mutex member so that the disk seek can be performed without locking the scheduler; otherwise, new requests cannot be scheduled while the seek is occurring.

To read a track, a client executes the following protocol:

```
scheduler.ScheduleTrack( track );
disk.Read( buf );
scheduler.Done();
```

Note, the ScheduleTrack entry must block the invoking task until it is that task's turn to read. The Done entry is invoked to inform the disk scheduler that the read operation completed and the disk drive is now free.

Your main program creates the scheduler, reads in a series of track numbers from cin, and for each track number, start a disk client task. The series of track numbers are terminated by end of file and are error free (i.e., assume all values are numeric and in the specified range). Each client initially yields a random number of times from 0 to M, inclusive, and then reads its track so that all clients do not attempt to read the disk simultaneously.

The output from the program should show clearly the order in which tracks are requested, the order in which clients are serviced, and how these events are interleaved. Also, when all the requests have been serviced, the program should print the average distance the disk head has moved per request (i.e., the total distance moved divided by the number of requests). **NOTE**: the routine names read and write are used by the UNIX operating system and **cannot** be used as routine names in the program.

The shell interface to the diskscheduler program is as follows:

```
diskscheduler [ M ]
```

(Square brackets indicate optional command line parameters.) if M is missing, assume 0. Assume the value M is a valid integer value, i.e., no error checking is required.

12. A semaphore is not the only kind of lock that can provide both synchronization and mutual exclusion. For example, a sequence/event-counter mechanism is based on two objects: a **sequencer** and an **event counter**. The metaphor for this mechanism is the ticket machine used to control the order of service at a store. A sequencer, S, corresponds to the numbered tags that are obtained from a ticket dispenser by arriving customers. An event counter, E, corresponds to the sign labeled "now serving", which is controlled by the servers. (As with all metaphors, its purpose is to understand a mechanism not to appear in the solution.)

    The interface for the sequencer is (you may add only a public destructor and private members):

```
#if defined( EVENTTYPE_MON )                          // explicit-monitor solution
// includes for this kind of buffer
_Monitor Sequencer {
    // private declarations for this kind of buffer
#elif defined( EVENTTYPE_AUTOMON )                    // automatic-signal monitor solution
// includes for this kind of buffer
_Monitor Sequencer {
    // private declarations for this kind of buffer
#else
    #error unsupported sequencer type
#endif
    // all common declarations
  public:
    Sequencer();
    unsigned int ticket();
    _Nomutex unsigned int check();
};
```

The value of a sequencer is, in effect, the value of the last numbered ticket to be taken. The first ticket is numbered 1, so the the initial value of a sequencer is 0. A sequencer has one mutex member, ticket, which atomically advances the sequencer by one and returns the sequencer's value. Hence, a sequencer yields a non-negative, increasing, contiguous sequence of integers. A ticket operation corresponds to a newly arriving customer taking a unique numbered tag. A sequencer has one non-mutex member, check, which returns the last ticket value taken.

The interface for the event counter is (you may add only a public destructor and private members):

```
#if defined( EVENTTYPE_MON )                          // explicit-monitor solution
// includes for this kind of buffer
_Monitor EventCounter {
    // private declarations for this kind of buffer
#elif defined( EVENTTYPE_AUTOMON )                    // automatic-signal monitor solution
// includes for this kind of buffer
_Monitor EventCounter {
    // private declarations for this kind of buffer
#else
    #error unsupported event-counter type
#endif
  public:
    EventCounter( unsigned int event = 0 );
    void advance();
    void await( unsigned int ticket );
    _Nomutex unsigned int check();
};
```

The event counter, in effect, holds the highest number of any customer being served. The initial number of events that can be handled before blocking is specified to the constructor, with the default being 0. For example, if a store has 3 salespeople, the event counter is initialized to 3 so that the first 3 customers, all having tickets less that or equal to the counter, do not block. The await member corresponds to a customer beginning to wait for service. A semaphore-like lock is simulated through the following usage of a Sequencer and an Event Counter:

```
E.await( S.ticket() );
```

which causes the task (customer) to take a ticket and wait until E reaches the ticket value. That is, an E.await(v) suspends the calling task if the value in E < v; otherwise the task proceeds. The advance member, executed by a server, corresponds to starting service with a new customer. The event counter is incremented and the next task (customer) is admitted for service. (This corresponds to incrementing the "now serving" sign and calling out the new number.) Hence, an E.advance() operation reawakens a task waiting for the event counter to reach its ticket value. Multiple tasks are reawakened when multiple servers simultaneously become free. Tickets may not be serviced in order because they may not be presented in order due to the non-atomic steps of taking a ticket and waiting. When a lower numbered ticket eventually appears, it is always serviced *before* any higher numbered tickets that are waiting. In essence, if you miss seeing your ticket number appear on the serving sign, your ticket is still valid and can be presented for service at any time. The non-mutex check member, returns the current event value for the event counter.

(For example, assume an event counter starts at 3 representing 3 servers. The first 3 customers arrive and take tickets 1-3. Customer 3 presents its ticket first and is allowed to proceed. Customer 4 arrives and takes ticket 4. Customer 3 completes, and advances the service counter to 4. Now any customer with ticket ≤ 4 can obtain service but there are at most 3 of them. Customer 5 arrives, takes ticket 5 and presents its ticket for service but must block because the service counter is only 4. Customers 1 and 4 present their tickets and are allowed to proceed. Customer 6 arrives, takes ticket 6 and presents its ticket for service but must block because the service counter is only 4. Customer 1 completes, and advances the service counter to 5. Now any customer with ticket ≤ 5 can obtain service but there are at most 3 of them. And so on. Hence, there are never more than 3 customers with the potential to obtain service at any time, which is crucial because there are only 3 servers able to process requests. There is a small inhibiting of concurrency because a server is prevented from servicing a large waiting ticket-value until all the lower ones are presented. But the delay is usually extremely short because, in general, the ticket is taken and immediately presented for service. Note, all of these actions rely on the programmer using the sequencer and event counter correctly. All the customers *must* perform the necessary cooperation in the correct order or the process fails.)

Implement two versions of the Sequencer and EventCounter monitors, one using explicit signalling and the other using automatic signalling. You can define the appropriate preprocessor variable on the compilation command using the following syntax:

```
u++ -DEVENTTYPE_MON -c EventCounter.cc
```

For the automatic-signal monitor, it is not required to service a lower numbered ticket *before* any higher numbered tickets that are waiting. Create an include file, called AutomaticSignal.h, which defines the following preprocessor macros:

```
#define AUTOMATIC_SIGNAL ...
#define WAITUNTIL( pred, expr... ) ... // gcc variable number of parameters
#define RETURN( expr... ) ...   // gcc variable number of parameters
```

Macro AUTOMATIC_SIGNAL is placed only once in an automatic-signal monitor as a private member, and contains any private variables needed to implement the automatic-signal monitor. Macro WAITUNTIL is used to delay until the pred evaluates to true; expr is optionally specified and evaluated each time pred evaluates to false and *before* the task delays (waits). Macro RETURN is used to return from a public routine of an automatic-signal monitor, where expr is optionally used for returning a value. For example, a bounded buffer implemented as an automatic-signal monitor looks like:

```
_Monitor BoundedBuffer {
    AUTOMATIC_SIGNAL;
    int front, back, count;
    int Elements[20];
  public:
    BoundedBuffer() : front(0), back(0), count(0) {}
    _Nomutex int query() { return count; }

    void insert( int elem ) {
        WAITUNTIL( count < 20 );
        Elements[back] = elem;
        back = ( back + 1 ) % 20;
        count += 1;
        RETURN();
    }
```

```
        int remove() {
            WAITUNTIL( count > 0 );
            int elem = Elements[front];
            front = ( front + 1 ) % 20;
            count -= 1;
            RETURN( elem );
        }
    };
```

Make absolutely sure to *always* have a RETURN() macro at the end of each mutex member.

Use the sequence/event-counter mechanism to construct a generalized bounded-buffer for a producer/consumer problem with the following interface:

```
    template<class T> class BoundedBuffer {
      public:
        BoundedBuffer( Printer<T> &prt, const int BufferSize );
        ~BoundedBuffer();
        void insert( const int Id, T elem );
        T remove( const int Id );
    };
```

which creates a bounded buffer of size BufferSize, and supports multiple producers and consumers. The insert and remove members are passed the Id of the calling producer/consumer, respectively. You must devise a way to use the sequencer/event-counter to implement the necessary synchronization and mutual exclusion needed by the bounded buffer.

Test the bounded buffer and its use of the sequencer/event-counter with a number of producers and consumers. The interface for a producer is (you may add only a public destructor and private members):

```
    _Task Producer {
        void main();
      public:
        Producer( Printer<int> &prt, BoundedBuffer<int> &buf,
                  const int Id, const int ItemsProd, const int Delay );
    };
```

The producer generates random integers from 1-100 inclusive and inserts it into buf. Each producer generates a random number of items from 5 to ItemsProd+5 inclusive. Before producing an item, a producer randomly yields between 0-Delay times.

The interface for a consumers is (you may add only a public destructor and private members):

```
    _Task Consumer {
        void main();
      public:
        Consumer( Printer<int> &prt, BoundedBuffer<int> &buf, const int Id, const int Delay );
    };
```

The consumer removes items from buf, and terminates when it removes -1 from the buffer. Before consuming an item, a consumer randomly yields between 0-Delay times.

All output from the program is generated by calls to a (special) printer coroutine, excluding error messages. The interface for the printer is (you may add only a public destructor and private members):

```
    template<class T> _Cormonitor Printer {
        void main();
      public:
        Printer( const unsigned int NoOfProd, const unsigned int NoOfCons );
        void change( const unsigned int Id, const char state, const T value );
        void change( const unsigned int Id, const char state );
    };
```

The printer generates output like that in Figure 9.27. Each column is assigned to a consumer (C) or producer (P), and a column entry indicates its current status:

State	Meaning	Additional Information
S	starting (C/P)	
F	finished (C/P)	
G	getting value (C)	
I	inserting (P)	value
R	removing (C)	value
M	mutual exclusion (C/P)	ticket
Y	synchronization (C/P)	ticket

When a consumer/producer changes state or the buffer changes a consumer/producer's state, a new row is added displaying the states of all the consumers/producers. A '*' is printed beside the state change of the consumer/producer associated with the current change. If a consumer/producer makes consecutive calls to the printer with no intervening call from another consumer/producer, only that consumer/producer's information is printed. When a consumer/producer finishes, the state for that consumer/producer is marked with F and all other consumer/producer are marked with `"..."`. All printing must occur in the coroutine main, i.e., no printing can occur in the coroutine's member routines. All output spacing can be accomplished using the standard 8-space tabbing, so it is unnecessary to build and store strings of text for output. Calls to the printer coroutine to perform printing may be performed in the buffer or in the consumer/producer tasks (you decide where to print).

uMain::main creates the printer, bounded buffer, the producer and consumer tasks. After all the producer tasks have terminated, uMain::main inserts an appropriate number of -1 values into the buffer to terminate the consumers.

The shell interface to the eventcounter program is as follows:

eventcounter [$BufferSize$ [$NoOfProd$ [$NoOfCons$ [$ItemsProd$ [$Delays$]]]]]

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) Where the meaning of each parameter is:

$BufferSize$**:** number of elements in the bounder buffer The default value if unspecified is 5.

$NoOfProd$**:** number of producers to create. The default value if unspecified is 4.

$NoOfCons$**:** number of consumers to create. The default value if unspecified is 4.

$ItemsProd$**:** used to determine the ranger of items generated by each producer. The default value if unspecified is 6.

$Delays$**:** number of times a producer/consumer yields *before* inserting/removing an item into/from the buffer. The default value if unspecified is 5.

Assume all arguments are valid positive integer values, i.e., no error checking is required.

```
% eventcounter 3 2 2 2 1
Cons:0  Cons:1  Prod:0  Prod:1
******  ******  ******  ******
S*
G
Y:1
Y:1     S*
        G
        Y:2
Y:1     Y:2     S*
                I:88
                Y:1
                M:1
                I:38
                Y:2
                M:2
Y:1     Y:2     M:2     S*
                        I:98
                        Y:3
                        M:3
M:1*    Y:2     M:2     M:3
R:88
G
Y:3
Y:3     Y:2     I:8*    M:3
                Y:4
                M:4
Y:3     M:2*    M:4     M:3
        R:38
        G
        Y:4
Y:3     Y:4     M:4     I:51*
                        Y:5
                        M:5
M:3*    Y:4     M:4     M:5
R:98
G
Y:5
Y:5     Y:4     I:3*    M:5
                Y:6
                M:6
Y:5     M:4*    M:6     M:5
        R:8
        G
        Y:6
Y:5     Y:6     M:6     I:81*
                        Y:7
                        M:7
```

```
M:5*    Y:6     M:6     M:7
R:51
G
Y:7
Y:7     Y:6     I:20*   M:7
                Y:8
                M:8
Y:7     M:6*    M:8     M:7
        R:3
        G
        Y:8
Y:7     Y:8     M:8     I:97*
                        Y:9
                        M:9
M:7*    Y:8     M:8     M:9
R:81
G
Y:9
...     ...     F       ...
Y:9     M:8*    F       M:9
        R:20
        G
        Y:10
Y:9     Y:10    F       I:52*
                        Y:10
                        M:10
M:9*    Y:10    F       M:10
R:97
G
Y:11
...     ...     ...     F
Y:11    M:10*   F       F
        R:52
        G
        Y:12
Y:11    Y:12    I:-1*   F
                Y:11
                M:11
                I:-1
                Y:12
                M:12
M:11*   Y:12    M:12    F
R:-1
F       ...     ...     ...
F       M:12*   M:12    F
        R:-1
...     F       ...     ...
**********************
```

Figure 9.27: Event Counter: Example Output

Chapter 10

Active Objects

Up to this point, tasks have been used solely to create a new thread of control and provide termination synchronization. In this role, such tasks are called an **active object**. Hence, the only member routines in a task type have been public constructors and possibly a destructor. This chapter expands on the capabilities of these active objects by introducing uses for other public members, called by other tasks.

Since a task type is like a class, it might seem obvious that it can have public members, which could be called by other tasks; however, there is a problem. When a task is created, it has a thread executing in its task main, and the scope of the task main allows the task's thread to access all the task's variables. If other tasks call public members of a task, these public members *also* have access to all the task's variables. Unless the calling threads are restricted to reading data, there is the potential for interference (i.e., race error). Even reading can result in problems because inconsistent values can be read when the task's thread is updating its variables. In essence, there are multiple threads accessing the same shared data without any mutual exclusion. While it is conceivable to use explicit locks in the member routines and task main to control access, this defeats the purpose of using a high-level language construct. Therefore, a task needs the mutual-exclusion property, like a monitor, if it is going to have public members called by other tasks.

10.1 Execution Properties

At this juncture, it is reasonable to review the basic properties of execution discussed so far to see how these properties relate to the language constructs in μC++. Three new execution properties have been introduced and discussed: execution state (Section 4.1.3, p. 67), thread (Chapter 5, p. 115), and mutual exclusion (Chapter 6, p. 137). An execution state is necessary so each coroutine and task has its own stack on which is can suspend (block) and subsequently resume (unblock), implicitly saving all necessary state during the suspension and reestablishing it on resumption. A thread is necessary to create a concurrent program with multiple execution points. Remember, an execution state and thread are the minimum properties required for independent execution. Finally, mutual exclusion is necessary to protect shared data accessed by multiple threads.

The first two properties are fundamental, i.e., it is impossible to create them from simpler constructs in a programming language. No amount of effort using basic control structures and variables can create new execution stacks and have the program context switch among them, nor is it possible to create a new thread of control that executes concurrently with other threads. Only mutual exclusion can be generated using basic control structures and variables (see Chapter 6, p. 137), but these software algorithms are complex and inefficient to the point of being impractical in all but a few special circumstances. For example, without atomic hardware instructions, N-task mutual exclusion is impractical, and atomic read/write instructions are not accessible through any of the basic language constructs. Therefore, all 3 of the new execution properties must be built into new language constructs tailored to achieve practical concurrent execution. Only the language compiler knows what constitutes an execution state and what has to be saved and restored for a context switch, and only it knows what atomic hardware instructions are available on the target architecture and where it is safe to perform certain optimizations during code generation. Finally, only the language runtime system knows how to manage and schedule multiple tasks. Language designers incorporate some or all of these three properties in different ways into a programming language (see Chapter 13, p. 359).

One factor affecting language design is the orthogonality of the three execution properties, resulting in 8 potential language constructs. However, in general, not all of the combinations are useful, so it is worth examining all the

object properties		object's member routine properties	
thread	execution state	no mutual exclusion	mutual exclusion
no	no	1 class-object	2 monitor
no	yes	3 coroutine	4 coroutine-monitor
yes	no	5 (rejected)	6 (rejected)
yes	yes	7 (rejected)	8 task

Table 10.1: Language Constructs from Execution Properties

combinations. In object-oriented languages, the three execution properties should be properties of an object. (In non-object-oriented languages, associating the three execution properties with language features is more difficult.) Therefore, an object may or may not have an execution state, may or may not have a thread, and may or may not have mutual exclusion. In the situation where an object does not have the minimum properties required for execution, i.e., execution state and thread, those of its caller must be used.

Table 10.1 shows the different high-level constructs possible when an object possesses different execution properties. Case 1 is a basic object, with calls to its member routines, which has none of the execution properties. In this case, the caller's execution-state and thread are used to perform the execution and there is no mutual exclusion. The lack of mutual exclusion means access must be serialized among threads or controlled through some explicit locking mechanism. Case 2 is like Case 1 but deals with the concurrent access problem by implicitly ensuring mutual exclusion for the duration of each computation by a member routine. This construct is a monitor. Case 3 is an object that has its own execution-state but no thread or mutual exclusion. Such an object uses its caller's thread to advance its own execution-state and usually, but not always, returns the thread back to the caller (semi- and full coroutining, respectively). This construct is a coroutine. Case 4 is like Case 3 but deals with the concurrent access problem by implicitly ensuring mutual exclusion. This construct is a coroutine-monitor. Cases 5 and 6 are objects with a thread but no execution state. Both cases are rejected because the thread cannot be used to provide additional concurrency. First, the object's thread cannot execute on its own since it does not have an execution state, so it cannot perform any independent actions. Second, if the caller's execution-state is used, assuming the caller's thread can be blocked to ensure mutual exclusion of the execution state, the effect is to have two threads serially executing portions of a single computation, which does not provide additional concurrency. Case 7 is an object that has its own thread and execution state. Because it has both a thread and execution state it is capable of executing on its own; however, it lacks mutual exclusion. Without mutual exclusion, access to the object's data is unsafe; therefore, having public members in such an object would, in general, require explicit locking, which is too low-level. Furthermore, there is no significant performance advantage over case 8. For these reasons, this case is rejected. Case 8 is like Case 7 but deals with the concurrent access problem by implicitly ensuring mutual exclusion. This construct is a task.

It is interesting that Table 10.1 generates all of the higher-level language constructs present in existing languages, but the constructs are derived from fundamental properties of execution not ad hoc decisions of a programming-language designer. As mentioned previously, if one of these constructs is not present, a programmer may be forced to contrive a solution that violates abstraction or is inefficient. Therefore, when trying to decide which construct to use to solve a problem in μC++, start by asking which of the fundamental execution properties does an object need. If an object needs to suspend/resume (wait/signal) and retain state during this period, the execution-state property is needed. If an object needs its own thread of control to execute concurrently, the thread property is needed. If an object is accessed by multiple threads, the mutual-exclusion property is needed. The answers to these 3 questions define precisely which μC++ construct to select for a solution. Therefore, execution state, thread and mutual exclusion are the three building blocks for all advanced program construction.

10.2 Direct/Indirect Communication

Why have public members in a task, especially since none have been required up to this point? The answer follows from the fact that there are two fundamental forms of communication: **direct communication** and **indirect communication**. Direct communication is where two active objects communicate directly with one another, e.g., when people communicate via talking, possibly over a telephone or electronic chat, that communication is direct. In essence, direct communication implies a point-to-point communication channel and both parties are immediately available to participate in the dialogue. Indirect communication is where two active objects communicate through a third party (usually an inactive object), e.g., when people communicate via writing, possibly through a letter or electronic mail, communication is indirect. In essence, indirect communication implies a station-to-station communication and neither party is immediately available during the dialogue.

Each form of communication is important, providing capabilities unavailable by the other. Direct communication provides immediate response in a conversation but requires both parties be constantly synchronized and available. Indirect communication provides delayed response but does not require either party to be constantly synchronized or available. Imagine if people could only communicate by talking (chat) or by writing letters (email). Neither form of communication is sufficient to satisfy all the basic communication requirements. While one form can simulate the other in a very weak sense (i.e., weak equivalence), both are needed to deal with quite different communication needs. Hence, the two kinds of communication are complementary, and both are needed in a concurrent system. Up to now, all forms of communication among tasks has been indirect. That is, tasks communicate through some shared data, protected by explicit locks (e.g., semaphores) or implicit locks (e.g., monitors).

10.2.1 Indirect Communication

The bounded buffer is a common example of a third party object through which tasks communicate information indirectly. Figure 10.1 illustrates the usage of a buffer by a producer and a consumer task with a **timing graph**, where time advances from top to bottom. Figures a) and b) illustrate the ideal cases where a producer or consumer find the buffer neither full or empty, and so they can insert and remove data from the buffer without synchronization blocking (mutual exclusion blocking is only necessary for multiple producers and consumers). Figures c) and d) illustrate the cases where a producer or consumer has to wait for data to be inserted or removed. (The waiting time of a task is indicated in the timing graph by a dashed line.) Notice, the consumer has to wait when the buffer is empty for a producer to insert an element, and then the consumer is unblocked via cooperation as the producer exits the buffer. Similarly, the producer has to wait when the buffer is full for a consumer to remove an element, and then the producer is unblocked via cooperation as the consumer exits the buffer. What is important in this example is that the ideal case involves no blocking (the minimum synchronization time), which is possible only with indirect communication. Blocking is eliminated via the buffer between the producer and consumer, which keeps the two tasks largely independent. Only boundary conditions on the buffer, e.g., full or empty, cause a blocking situation.

In the bounded buffer example, the data transfer is unidirectional, i.e., the data flows in one direction between producer and consumer, and hence, the producer does not obtain a response after inserting data in the buffer. Figure 10.2 illustrates the usage of a monitor to indirectly transfer information *bidirectionally* between two tasks. Like sending email, task_1 drops off a message in the monitor and returns at some time to check for a response. Task_2 arrives at the monitor to read the message, formulate a reply, and then continue. Figure a) illustrates the ideal case where task_1 arrives first and drops off the message, followed by the arrival of task_2 to process the message, and finally the re-arrival of task_1 to retrieve the reply. Again, this ideal case requires no blocking. Figures b), c), and d) illustrate the cases where task_2 arrives ahead of task_1 and has to wait until the message arrives, where task_1 arrives for the reply before it is ready and has to wait, and finally, a combination of the last two, where task_2 arrives before the message is ready and task_1 arrives before the reply is ready.

If task_1 cannot continue without a reply (i.e., the work it has to do requires the reply), it must wait until task_2 arrives and generates a reply before continuing execution. Figure 10.3 illustrates the usage of a monitor to indirectly transfer information bidirectionally between two tasks, where an immediate reply is required. Like electronic chat, task_1 drops off a message in the monitor and waits for a response. Task_2 arrives at the monitor to read the message, formulate a reply, and then continue. Figure a) illustrates the ideal case where task_1 arrives first with the message and waits, followed by the immediate arrival of task_2 to process the message, and finally the unblocking of task_1 to retrieve the reply. Here, even the ideal case requires blocking. Figures b) and c) illustrate the cases where task_2 arrives ahead of task_1 and has to wait until the message arrives, and task_1 arrives ahead of task_2 and has to wait for the reply, respectively.

In these 3 scenarios, observe that the first scenario has a buffer of size N, and in the last two cases a buffer of size 1. As well, in the first two cases, both synchronization and mutual exclusion are necessary for the buffer (supplied by the . The buffer reduces the amount of blocking by eliminating the need for the tasks to interact directly. In the last case, only synchronization is necessary; mutual exclusion is not required (although the monitor still provides mutual exclusion). In other words, blocking is always required for the immediate reply case, which correspondingly implies synchronization always occurs.

Finally, these scenarios can be made more complex when multiple producers and consumers are introduced. The

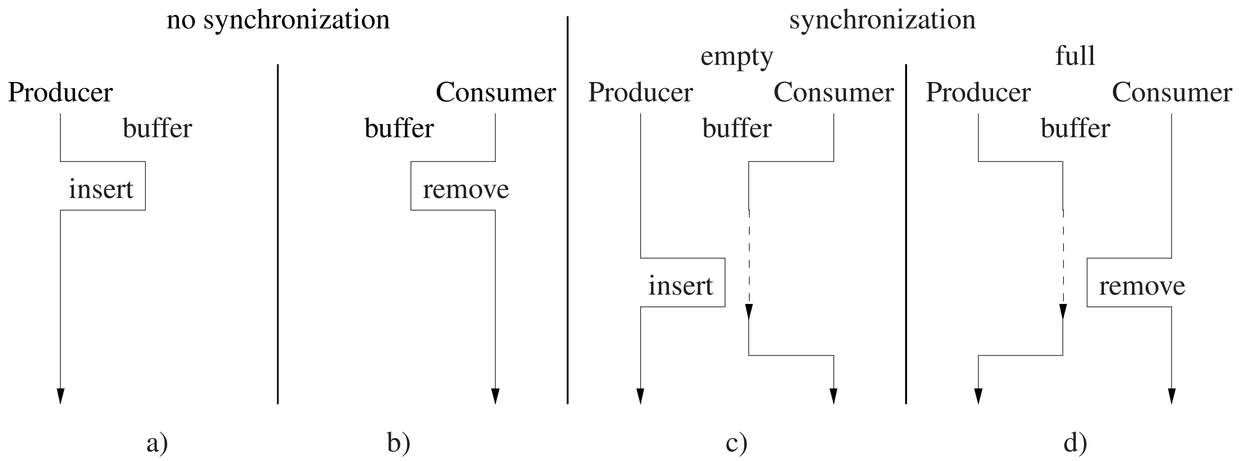


Figure 10.1: Unidirectional Indirect Communication

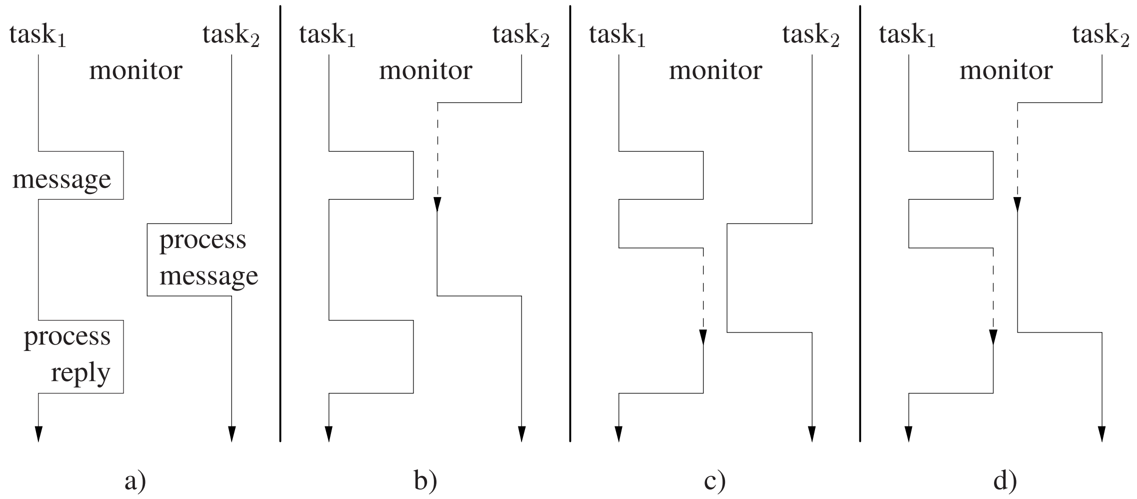


Figure 10.2: Bidirectional Indirect Communication

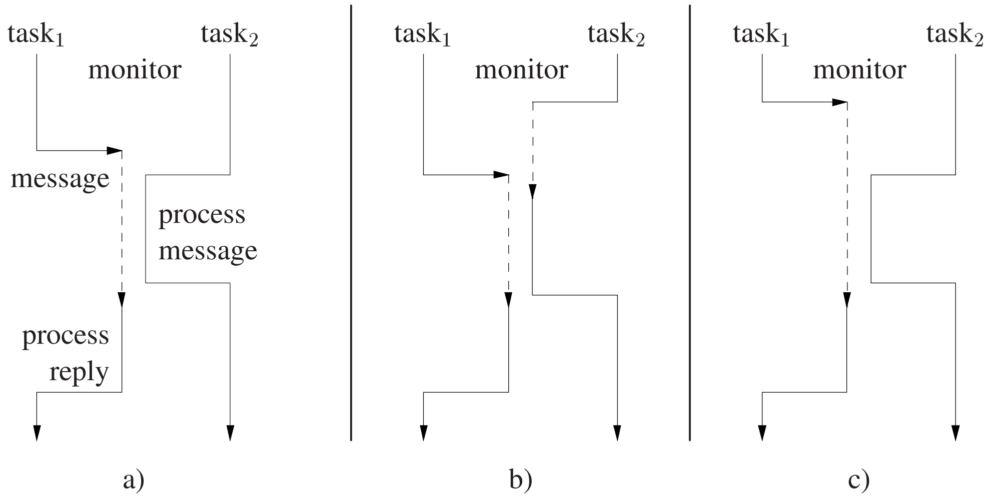


Figure 10.3: Reply Required Indirect Communication

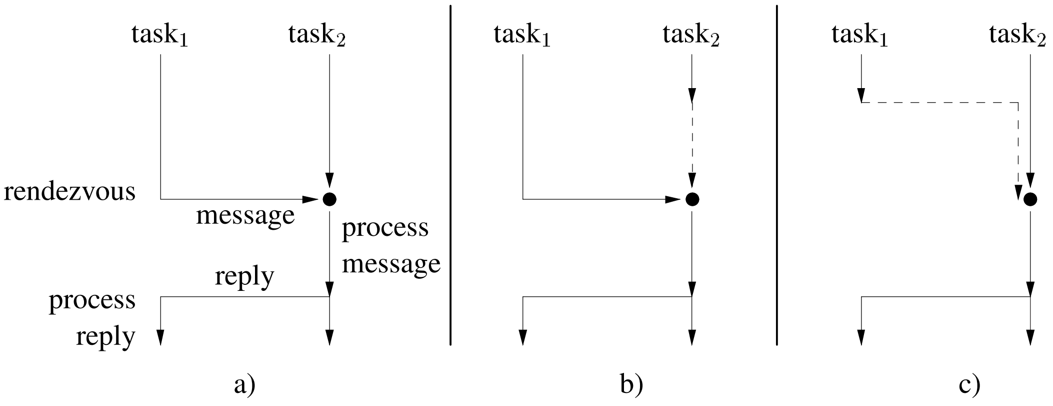


Figure 10.4: Direct Communication

complexity occurs because processing the message and/or reply in the monitor precludes other tasks from entering the monitor because of the mutual-exclusion property, so unless the processing is very short, the data should be dealt with outside the monitor so concurrency is not inhibited for other communicating tasks (see Chapter 11, p. 339 for techniques to accomplish this). For task$_1$, the solution is to simply copy the reply out of the monitor and then processes it. For task$_2$, the solution is to copy the message out of the monitor, and subsequently reenter the monitor once it has processed the message and generated a reply. The reply must be copied and stored in the monitor, and cooperation requires the matching receiver for a reply be unblocked, if waiting in the monitor. (Similarly, cooperation requires the matching receiver for a message be unblocked, if waiting in the monitor.) It may also be necessary to have an additional buffer for the replies as well as the messages.

In summation, indirect communication takes advantage of the fact that blocking is often unnecessary because an immediate response to a message is not required. Blocking is eliminated by using buffers to temporarily store information until the other party in the communication appears. If an immediate response is required, blocking is always required, which eliminates the need for a buffer.

10.2.2 Direct Communication

The last scenario of indirect communication mimics direct communication. The key point is that the monitor provides no shared data (i.e., no buffer), and hence, is only used for synchronization. Direct communication optimizes this particular scenario by providing the necessary synchronization *without* the need to create and manage an additional indirect object (which has no data anyway).

Figure 10.4 illustrates the direct transfer of information bidirectionally between two tasks, where an immediate reply is required. In this scenario, task$_1$ transfers a message to task$_2$ directly. The two tasks must first synchronize to perform the communication; when tasks synchronize directly it is called a **rendezvous**. Figure a) illustrates the rare but ideal case where task$_1$ and task$_2$ rendezvous simultaneously. Once the rendezvous has occurred, one of the threads of the two tasks processes the message. Which one depends on the concurrency system: in μC++, it is task$_1$'s (caller's) thread while task$_2$'s (called) thread remains blocked during the rendezvous. The message must be copied into task$_2$'s member, where it is processed and a reply generated. When the reply is ready, both tasks continue execution, with task$_1$'s thread copying the reply. Notice, no other thread can execute in task$_2$ while the message is being processed because of the mutual exclusion property of a task in μC++. Figures b) and c) illustrate the cases where task$_2$ arrives ahead of task$_1$ and has to wait for the rendezvous, and task$_1$ arrives ahead of task$_2$ and has to wait for the rendezvous, respectively.

While the message itself has to be processed in task$_2$'s member, the reply could also be processed in task$_2$'s member instead of being copied out for processing by task$_1$. However, processing the reply in task$_2$'s member precludes other tasks from entering it because of the mutual-exclusion property, so unless the processing is very short, the reply should be dealt with outside the called task.

As mentioned, at the point of the rendezvous, either thread could be used to process the message. Noticing which thread reaches the rendezvous first suggests which thread should execute the member routine processing the message. Figure 10.4 a) is simultaneous arrival, which occurs rarely; in this case, either task could process the message. In

Figure b), $task_2$ arrives at the rendezvous first and blocks. If this case is common, using $task_2$'s thread is inefficient because $task_1$'s thread must block and $task_2$'s thread unblock. It is more efficient to leave $task_2$'s thread blocked and allow $task_1$'s thread to execute $task_2$'s member. In Figure c), $task_1$ arrives at the rendezvous first and blocks. If this case is common, using $task_1$'s thread is inefficient because $task_2$'s thread must block and $task_1$'s thread unblock. It is more efficient to leave $task_1$'s thread blocked and allow $task_2$'s thread to execute its member. It is possible to dynamically decide which task to block and continue, by checking if the rendezvous partner is or is not available. If the rendezvous partner is not available, the task blocks; if the rendezvous partner is available, the task executes the member. However, making the decision as to which thread blocks or continues adds to the complexity of the implementation. In fact, after experimentation, the most common case in most programs is where the caller task finds the called task blocked, e.g., Figure b). (Why this case is the most common is discussed in Chapter 11, p. 339.) Therefore, the best average performance is obtained by having the caller's thread execute a task's mutex member, which is what is done in μC++. As well, having the caller's thread execute the member in μC++ is consistent with how coroutine and monitor members are executed; for both, the caller's thread and execution-state are used. Therefore, when routines uThisCoroutine or uThisTask are invoked in a public member, they never return the coroutine or task identifier associated with the called object; instead they return the one associated with the caller, which reflects the fact that the caller's execution-state is being used for the execution because no context switch has occurred to the other coroutine or task at this point.

10.3 Task

The basic features of a task were discussed in Section 5.9, p. 127: task main, creation and destruction, and inherited members. Similar to a monitor (see Section 9.3, p. 256) a task is **active** if a thread is executing a mutex member or the task main, otherwise it is **inactive** The task's mutual exclusion is enforced by **locking** the task when executing a mutex member or the task main, and **unlocking** it when the active thread voluntarily gives up control of the task. As for monitors, public members of a task are mutex by default, and a thread executing a task's mutex member is allowed to call another mutex member of that task without generating mutual-exclusion deadlock (see Section 9.3.1, p. 257).

10.4 Scheduling

Scheduling for a task is similar to that of a monitor except the programmer must be aware of the additional thread associated with the task. Unlike a monitor, when a task is created it starts *active*, i.e., with a thread running in it. Therefore, no calls to mutex members can occur until the initial thread either accepts a mutex member or waits; in both cases, the initial task thread blocks, allowing another thread to acquire mutual exclusion of the task and begin execution of a mutex member. As for monitors, there are two scheduling techniques: external and internal scheduling. Finally, because the mutex property is the same for all mutex types, the implicit scheduling mechanism and data structures for a task are the same as for a monitor (see Figure 10.5 as a reminder, which is the same as Figure 9.8, p. 265).

10.4.1 External Scheduling

External scheduling uses the accept statement to schedule tasks waiting on the implicit mutex queues (because these tasks called a mutex member and blocked). To illustrate external task scheduling, a generic μC++ bounded buffer example is presented in Figure 10.6, built using a task rather than a monitor. In this case, using a task is unnecessary and inefficient; a monitor is clearly the best construct because of the indirect communication. Nevertheless, it is illustrative to see how easy it is to convert a monitor to a task to solve this problem.

The buffer implementation is the same as that in Section 9.4.1, p. 259 (see Figure 9.4, p. 259), i.e., a fixed sized array with front and back pointers cycling around the array, with an additional counter in this implementation for the number of full slots in the buffer. The interface has four members: the constructor, query, insert and remove. The only difference in the code from the external monitor solution is that the synchronization is moved from routines insert and remove into the task main. The task main loops forever (stopping it is discussed shortly) accepting calls to either insert or remove. This accept statement introduces the ability to accept a call to one of a number of mutex members using the **else** clause. (The **else** clause is discussed in detail in Section 9.6.2, p. 272.)

As calls are made to members insert and remove, the task main accepts each one, which causes its thread to block on the acceptor/signalled stack and unblocks a waiting caller from the appropriate mutex queue closest to the start of the accept statement, or if no waiting caller, for a call to one of the specified mutex member(s) to occur. When the caller eventually exits the mutex member, the task's thread is popped off the acceptor/signalled stack and continues

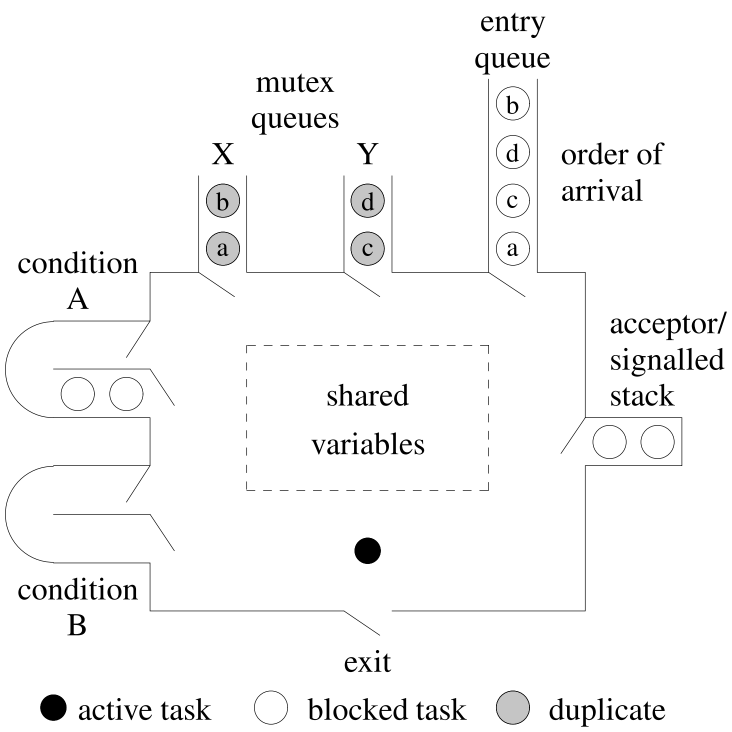


Figure 10.5: μC++ Mutex Object

```
template<class ELEMTYPE> _Task BoundedBuffer {
    int front, back, count;
    ELEMTYPE Elements[20];
  public:
    BoundedBuffer() : front(0), back(0), count(0) {}
    _Nomutex int query() { return count; }

    void insert( ELEMTYPE elem ) {
        Elements[back] = elem;
        back = (back + 1) % 20;
        count += 1;
    }
    ELEMTYPE remove() {
        ELEMTYPE elem = Elements[front];
        front = (front + 1) % 20;
        count -= 1;
        return elem;
    }
  protected:
    void main() {
        for ( ;; ) {
            _When (count < 20) _Accept(insert);
            else _When (count > 0) _Accept(remove);
        }
    }
};
```

Figure 10.6: Task Bounded Buffer: External Scheduling

execution *after* the accept statement (remember, only one call is accepted). The thread of the task main then loops around to accept the next caller. Notice, the task's thread must alternate its execution in the task main with the callers to members insert and remove, otherwise the task's thread would prevent the callers from entering the mutex members. (Therefore, there is no advantage in having a thread to accept the calls; a monitor implicitly accepts all the mutex members when inactive without the need of an additional thread.) In addition, some other mechanism is needed to provide synchronization when the buffer is full or empty.

To provide the necessary synchronization, the task's thread must conditionally accept the mutex members; otherwise, a caller has to block in the insert or remove routines, which requires internal scheduling (discussed next). Notice, moving the conditional control from the external monitor solution into the task main's loop does not work:

```
for ( ;; ) {
    if ( count == 0  ) _Accept( insert );
    else if ( count == 20 ) _Accept( remove );
}
```

This approach fails because the task's thread only accepts members when the buffer is full or empty, not at any intermediate buffer state. To deal with this case requires augmenting the conditional control:

```
for ( ;; ) {
    if ( 0 < count && count < 20 ) _Accept( insert, remove );
    else if ( count < 20 ) _Accept( insert );
    else /* if ( 0 < count  ) */ _Accept( remove );
}
```

The first **if** statement checks if both conditions are true, i.e., the buffer is neither empty nor full, and if so, accepts both members. If the first condition is false, either the buffer is full or empty. The next **if** statement determines which of these cases is true and accepts the appropriate member. (Notice the optimization of eliminating the last check because it must be true given the other checks.)

While this conditional control now works, it points out a major problem with this approach to conditionally accepting mutex members. Generalizing the conditional control based on the first failing buffer example results in:

```
if ( C1 ) _Accept( M1 );
else if ( C2 ) _Accept( M2 );
```

However, this does not result in either M1 or M2 being accepted if both conditions are true. The order imposed by the **if** statements together with the meaning of **_Accept** results in a something weaker. It says accept M1 when C1 is true, accept M2 when C1 is false and C2 is true, and *only* accept M1 when C1 and C2 are true instead of accepting both M1 and M2. Therefore, this approach results in starvation of callers to M2 if condition C1 is always true even when there are no calls to M1. For this reason, the first buffer example failed.

Now generalize the second working buffer example but extend it from 2 to 3 conditionals:

```
if      ( C1 && C2 && C3 ) _Accept( M1, M2, M3 );
else if ( C1 && C2 ) _Accept( M1, M2 );
else if ( C1 && C3 ) _Accept( M1, M3 );
else if ( C2 && C3 ) _Accept( M2, M3 );
else if ( C1 ) _Accept( M1 );
else if ( C2 ) _Accept( M2 );
else if ( C3 ) _Accept( M3 );
```

This form is necessary to ensure that for every true conditional, only the corresponding members are accepted. The general pattern for N conditionals is:

$$\binom{N}{N} + \binom{N}{N-1} + \ldots + \binom{N}{1} = (1+1)^N - 1 \quad \text{from the binomial theorem.}$$

Having to write an exponential number of statements, i.e., $2^N - 1$, to handle this case is clearly unsatisfactory, both from a textual and performance standpoint, and is an example of weak equivalence with the following language construct.

To prevent the exponential growth for conditional accepting, μC++ provides a **_When** clause. The two **_When** clauses in the task main of Figure 10.6 make the accept clause conditional on the condition specified in the **_When**. A condition must be true (or omitted) before a member is accepted. If the condition is false, the accept does not occur even if there is a calling task waiting on the mutex queue of the specified member. In the bounded buffer example, the first **_When** clause only accepts calls to insert if the buffer has empty slots (count < 20), and the second **_When** clause only accepts calls to remove if the buffer has non-empty slots (count > 0). Notice only one of the two conditionals can be false simultaneously, so the accept statement always accepts either one or both of the two members. If either or both of conditionals are true, but there are no outstanding calls, the acceptor is blocked until a call to an appropriate member is made. The exponential number of statements are eliminated because the **_When** and the **_Accept** clauses are checked *simultaneously* during execution of the accept statement instead of having to first check the conditionals and then perform the appropriate accept clauses in an accept statement.

While the **_When** clause is the same as the conditional of a conditional critical-region (see Section 9.2, p. 251), the **_When** clause does not result in busy waiting; the busy wait is eliminated by the cooperation in the task solution for the bounded buffer. Instead of having each producer and consumer constantly check if the buffer is accessible, they block outside the buffer (external scheduling) until accepted when the buffer is in an appropriate state. Hence, conditional control does not imply busy waiting, it is lack of cooperation that implies busy waiting.

One outstanding issue is the case when all the conditionals are false, which means no call is accepted. There are two options available, both problematic. The first option is to block at the end of the accept waiting for a call, which is the normal semantics for an accept statement. However, the task is now in a synchronization deadlock because it is waiting for an event that cannot occur, i.e., a call when no call is accepted. The second option is to continue execution after the accept without accepting a call. However, after the accept statement, the task must reevaluate all the conditionals to know if a call was accepted or use a flag variable, as in:

```
flag = false;
_When ( C1 ) _Accept( M1 ) {
    flag = true;
} else _When ( C2 ) _Accept( M2 ) {
    flag = true;
}
if ( flag ) {
    // an accept occurred
} else {
    // all conditionals are false
}
```

This second option could easily result in a mistake if appropriate checking is not made, as the acceptor incorrectly assumes some cooperation has occurred by an accepted call, when in fact no call may be accepted because all the conditionals are false. This situation can lead to an infinite loop around an accept statement accepting no calls.

μC++ deals with both these situations. The default action for the case when all the accepts are conditional (as in the bounded buffer example) and all the **_When** conditions are false is to generate an error. This semantics prevents the synchronization deadlock or incorrectly assumed cooperation. In most cases, this error indicates a malformed set of conditionals for the **_When** clauses. As a result, it is incorrect to use the **_When** clause in the external monitor solution for the bounded buffer (see page 259), as in:

```
void BoundedBuffer::insert( int elem ) {
    _When ( count == 20 ) _Accept( remove );  // hold calls to insert
    . . .
int BoundedBuffer::remove() {
    _When ( count == 0 ) _Accept( insert ); // hold calls to remove
    . . .
```

because all calls, except when the buffer is full or empty, result in an error. Using **if** statements in the monitor solution results in the accept statements not being executed when the buffer is neither full nor empty. In the rare case when a programmer only wants to check for calls but not block if no call(s) exist, a terminating **else** clause can be used on the accept statement. If all the accepts are conditional and all of the **_When** conditions are false but there is a terminating **else** clause (with a true **_When** condition), the terminating **else** clause is executed instead of blocking, as in:

```
void BoundedBuffer::insert( int elem ) {
    _When ( count == 20 ) _Accept( remove );  // hold calls to insert
    else {   // terminating else
        // executed instead of blocking
    }
```

which simulates the version using an **if** statement. Therefore, the programmer knows all conditionals are false without having to reevaluate them. Nevertheless, care must be taken in using the **else** clause as it can easily lead to busy waiting or possibly an infinite loop.

Now look back at Figure 10.6, p. 323 with the following question: why is BoundedBuffer::main defined at the end of the task? The reason is the *definition before use rule* in C++ (see Section 9.4.1, p. 259 for a similar situation). Since the accept statement references members insert and remove, both must be defined *before* the accept statement refers to them. It is a common mistake to put the accept statement before the routine it is accepting, and get an error from μC++ that one or more members are not mutex (only mutex members can be accepted). The reason for this error is that μC++ has not seen the routine definition so it assumes the routine is no-mutex.

One final point about potential starvation when using the **_Accept** statement. When several members are accepted and outstanding calls exist to some subset of these members, a call is selected based on the order of appearance of the **_Accept** clauses in the accept statement. Hence, the order of the accept clauses indicates the relative priority for selection if there are several outstanding calls. As will be shown in future examples, this is an important feature. In the bounded-buffer task, calls to insert and remove require equal priority when the buffer is neither full nor empty for fairness; however, priority is given to producers over consumers because member insert is accepted first. This priority can lead to short term unfairness for the consumers, but the bound on the length of the buffer (specified through the **_When** clause) ensures consumers eventually make progress (i.e., when the buffer is full). However, imagine an infinite sized buffer; in this case, the **_When** clause is removed from the **_Accept**(insert). Now if a continuous supply of producer calls occurs, it could prevent a consumer from ever removing an element from the buffer, and hence, result in consumer starvation.

One simple way of dealing with this problem is to use the Dekker solution of alternation, as in:

```
void BoundedBuffer::main() {
    for ( ;; ) {
        _When ( count < 20 ) _Accept( insert ); // "insert" has highest priority
        else _When ( count > 0 ) _Accept( remove );

        _When ( count > 0 ) _Accept( remove ); // "remove" has highest priority
        else _When ( count < 20 ) _Accept( insert );
    }
}
```

Here the accept statement has been duplicated in the **for** loop but with the order of accept clauses reversed in the second accept statement. Each time through the loop, each of the two mutex members is treated as the highest priority in the accept statements. Hence, when there are simultaneous calls from both producers and consumers, the calls are handled alternately, resulting in fairness. Notice, this situation is largely fictitious because it requires an infinite sized buffer and a continuous stream of producer calls to produce starvation. For normal finite resource situations, temporary periods of unfairness occurring in a few rare situations from the priority associated with the order of the accept clauses do not cause problems. Therefore, duplicating the accept clauses in this way is largely unnecessary, except in a few rare situations.

10.4.2 Internal Scheduling

The complementary approach to external scheduling is internal scheduling, where calling tasks are largely scheduled inside a task from explicit condition variables rather than from outside the task from implicit mutex queues. As for monitors, internal scheduling is accomplished using condition variables, and signal/wait. To illustrate internal task scheduling, a generic μC++ bounded buffer example is presented in Figure 10.7, built using a task rather than a monitor. Again, using a task is unnecessary and inefficient, but it is illustrative to see how easy it is to convert a monitor to a task to solve this problem.

The first point to note is that, except for the existence of the task main, the code is identical to the monitor solution in Figure 9.6, p. 262. The second point to note is that the conditional control is removed from the accept statement in the task main. The last point to note is that even though this is internal scheduling, there is still an accept statement in the task main performing external scheduling.

This example illustrates even more why using a task for this problem is inappropriate. When the buffer task is created, no producer or consumer can get in to use the buffer because of the task's thread. The accepts in the buffer's task main are necessary to ensure the buffer's thread blocks so the producer and consumer can access the buffer. However, the conditional control is now performed by cooperation between the producer and consumer, without help

```
template<class ELEMTYPE> _Task BoundedBuffer {
        uCondition Full, Empty;
        int front, back, count;
        ELEMTYPE queue[20];
    public:
        BoundedBuffer() : front(0), back(0), count(0) {}
        _Nomutex int query() { return count; }
        void insert( ELEMTYPE elem ) {
            if ( count == 20 ) Empty.wait();
            queue[back] = elem;
            back = (back + 1) % 20;
            count += 1;
            Full.signal();
        }
        ELEMTYPE remove() {
            if ( count == 0 ) Full.wait();
            ELEMTYPE elem = queue[front];
            front = (front + 1) % 20;
            count -= 1;
            Empty.signal();
            return elem;
        }
    protected:
        void main() {
            for ( ;; ) {
                _Accept( insert );
                else _Accept( remove );
            }
        }
};
```

Figure 10.7: Task Bounded Buffer: Internal Scheduling

from the buffer task. Remember, the caller's thread is used to execute the member routines, so it is the producer and consumer executing the waits and signals in the members insert and remove. If either a producer or consumer find the buffer full or empty, respectively, they block on the appropriate condition, and that condition is subsequently signalled by the opposite kind of task on exit from the member routine. When the buffer is not full or empty, the buffer task unblocks and does another iteration of the loop in its task main between each producer and consumer call. When the buffer is either full or empty, the buffer task is still unblocked because it is at the top of the acceptor/signalled stack after a task waits on a condition variable.

So in this particular example, the buffer's thread does absolutely nothing to help solve the problem, but it does slow down the entire process by constantly blocking and unblocking between calls to insert and remove.

10.5 External and Internal Scheduling

The last example illustrates the use of both internal and external scheduling by a task. In most tasks, some external scheduling is always required, and in addition some internal scheduling may be required. So the two common forms are strictly external or a combination of external and internal; it is rare for a task to only use internal scheduling. The reason is that the task's thread must block for other tasks to interact, and the management of the task's thread to achieve this effect requires that it accept calls to the interacting tasks.

10.6 When a Task becomes a Monitor

As discussed in Section 10.1, p. 317, the mutual-exclusion property of a type is independent of whether the type has an associated thread. Interestingly, if a task's thread waits or terminates, the task can continue to be used. When the task's thread is blocked or terminated, the object still exists with the mutex property but no associated thread, so it becomes a monitor. (In a similar way, when a coroutine terminates it becomes a class-object because the execution state is no longer accessible.) Now when the task's thread blocks or terminates, the object is inactive and its implicit scheduling attempts to first start a task on the acceptor/signaller stack and then from the entry queue. As mentioned, the destructor of a mutex object is always mutex; so an implicit call to the destructor can be waiting on the entry queue. As for a monitor, if there are tasks blocked on conditions, or the acceptor/signaller stack, or the entry queue, the destructor can still be invoked if the task's thread waits. However, since the task's thread must be blocked on a condition queue associated with the task, an error is generated because the destructor cannot complete while threads are blocked in the task. The destructor can use blocking signals to wake the blocked threads, and hence, deal with this case.

As mentioned, it is rare for a task to only use internal scheduling, but it is possible by cheating: let the task's thread terminate or block so the task becomes a monitor, then use monitor internal scheduling techniques. The following are modifications to the bounder-buffer task in Figure 10.7 so only internal scheduling is used:

<pre>... _Task BoundedBuffer { ... public: ... protected: void main() { // no code } };</pre>	<pre>... _Task BoundedBuffer { ... uCondition cheating; public: ... ~BoundedBuffer() { cheating.signalBlock(); } protected: void main() { cheating.wait(); } };</pre>

In the left program, the accept statement has been removed from the task main so when the thread starts, it immediately terminates, changing the task into a monitor. This solution is now identical to the monitor solution using internal scheduling except for the unnecessary expense of creating an execution state and starting and terminating a thread, neither of which are used. In the right program, the accept statement has been replaced with a wait, so when the thread starts, it immediately blocks changing the task into a monitor. The solution is now identical to the monitor solution using internal scheduling, except the bounded-buffer's thread must be restarted in the destructor so that it can terminate before the destructor can complete. This unblocking is accomplished using a signalBlock because the task executing the destructor must wait until the signalled task completes (and terminates). Both cases cheat because they convert the task to a monitor for its entire lifetime, which begs the question of why use a task in the first place.

10.7 Task Details

Like a class and coroutine, the qualifiers **_Mutex** and **_Nomutex** can qualify the **_Task** definition, as well as the **private**, **protected** and **public** member routines. When the **_Mutex** qualifier is placed on a **_Task** definition, it indicates all public member routines have the mutual-exclusion property, unless overridden on specific member routines with the **_Nomutex** qualifier. When the **_Nomutex** qualifier is placed on a **_Task** definition, it indicates all public member routines have the no-mutual-exclusion property, unless overridden on specific member routines with the **_Mutex** qualifier. The default for a task if no qualifier is specified is **_Mutex** because the mutual-exclusion property for public members is typically needed.

```
_Mutex _Task T1 {
    ...
  public:
    mem1() ...              // mutex
    _Nomutex mem2() ... // no mutex
    ~T1() ...               // mutex
}
```

```
_Nomutex class T2 {
    ...
  public:
    mem1() ...              // no mutex
    _Mutex mem2() ...   // mutex
    ~T2() ...               // mutex
}
```

Since a task always has the mutual-exclusion property, it means, like a monitor, a task always has one mutex member, its destructor, and a task's destructor cannot be qualified with **_Nomutex**.

10.7.1 Accept Statement

The basic structure of the accept statement is presented in Section 9.6.2, p. 272. The following are additional capabilities.

The simple form of the **_Accept** statement is:

```
_When ( conditional-expression )            // optional guard
    _Accept( mutex-member-name-list );
```

A **_When** guard is considered true if it is omitted or if its *conditional-expression* evaluates to non-zero. The *conditional-expression* of a **_When** may call a routine, **but the routine must not block or context switch.** The guard must be true and an outstanding call to the specified mutex member(s) must exist for a call to be accepted. If there are several mutex members that can be accepted, selection priority is established by the left-to-right placement of the mutex members in the **_Accept** clause of the statement. Hence, the order of the mutex members in the **_Accept** clause indicates their relative priority for selection if there are several outstanding calls. If the guard is true and there is no outstanding call to the specified member(s), the acceptor is accept-blocked until a call to the appropriate member(s) is made. If the guard is false, the program is aborted; hence, the **_When** clause can act as an assertion of correctness in the simple case. Therefore, a guard is not the same as an **if** statement, e.g.:

```
if ( count == 0 ) _Accept( mem );          _When ( count == 0 ) _Accept( mem );
```

In the right example, the program aborts if the conditional is false. (It is possible to simulate the **if** version using the **else** clause discussed below.)

The extended form of the **_Accept** statement is augmented to conditionally accept one of a group of mutex members, e.g.:

```
_When ( conditional-expression )            // optional guard
    _Accept( mutex-member-name-list )
        statement                           // statement
else _When ( conditional-expression )       // optional guard
    _Accept( mutex-member-name-list )
        statement                           // statement
. . .
    . . .
        . . .
_When ( conditional-expression )            // optional guard
else                                        // optional terminating clause
  statement
```

Before an **_Accept** clause is executed, its guard must be true and an outstanding call to its corresponding member(s) must exist. If there are several mutex members that can be accepted, selection priority is established by the left-to-right then top-to-bottom placement of the mutex members in the **_Accept** clauses of the statement. If some accept guards are true and there are no outstanding calls to these members, the task is accept-blocked until a call to one of these members is made. If all the accept guards are false and no **_Accept** clause can be executed immediately, the program is aborted, unless there is a terminating **else** clause with a true guard, which is executed instead. Hence, the terminating **else** clause allows a conditional attempt to accept a call without the acceptor blocking. Again, a group of **_Accept** clauses is not the same as a group of **if** statements, e.g.:

```
if ( Ci ) _Accept( Mi );                    _When ( Ci ) _Accept( Mi );
else if ( Cj ) _Accept( Mj );               else _When ( Cj ) _Accept( Mj );
```

The left example accepts only M_i if C_i is true or only M_j if C_i is false and C_j is true. The right example accepts either M_i or M_j if C_i and C_j are true. Once the accepted call has completed *or the caller waits*, the statement after the accepting **_Accept** clause is executed and the accept statement is complete.

10.7.2 Accepting the Destructor

As mentioned, the destructor of a mutex object is always mutex and the accept statement accepts any mutex member; therefore, the destructor is eligible to be accepted in an accept statement like any other mutex member. But why accept the destructor of a mutex object, especially a task?

The answer to this question starts with another question: who calls the destructor of an object? The compiler implicitly inserts calls to the destructor before storage is deallocated (at the end of a block for a local variable or as part of the **delete** for a dynamically allocated variable). An executing thread then makes the destructor calls via the implicitly inserted calls; when a thread invokes an implicit destructor call for a mutex object, it must block until the destructor is accepted (all calls to mutex members must be accepted). However, up to this point, no explicit accepting of a mutex object's destructor has occurred in any monitor, coroutine-monitor or task (i.e., all the kinds of mutex types). How then has the call to the destructor been accepted?

In the case of a monitor or coroutine-monitor, the destructor is implicitly accepted by the implicit scheduling when a task exits a mutex object and there is no task on the acceptor/signalled stack. In this case, the next task from the entry queue (C) is implicitly accepted, and if that call happens to be to the destructor, it is accepted. The destructor is then executed and the object deallocated by the calling task's thread. If there are tasks blocked on the entry queue or condition variables associated with the mutex object, the destructor cannot complete so the program terminates with an error. For implicit acceptance of the destructor, there can never be tasks blocked on the acceptor/signaller stack as it must be empty before the implicit scheduling selects a task from the entry queue ($C < W < S$). Thus, in all the previous programs using monitors, code has been set up so the call to the destructor is the last call made to the monitor, as in:

```
{
    BoundedBuffer buf;
    {
        Consumer cons( buf );       // pass buffer to consumer task
        Producer prod( buf );       // pass buffer to producer task
    } // implicit call to consumer and producer destructor (wait for termination)
} // implicit call to buffer destructor (no tasks using the buffer now)
```

The inner block cannot terminate until both the consumer and producer tasks have terminated. Once the inner block has terminated, the outer block terminates and implicitly calls the destructor of the bounded buffer, buf. Since the consumer and producer have terminated, there are no tasks either calling or in the buffer. Therefore, the call to the destructor by the deleting task (most likely uMain) is implicitly accepted from the entry queue, the destructor is executed, and the storage deallocated (from the stack). (In fact, the nested blocks are unnecessary in this example because deallocation is defined in C++ to occur in the reverse order to allocation.)

What about the producer and consumer tasks in the previous example as both their destructors are implicitly called at the end of the inner block? As noted, the destructor call to a task must wait for the task's thread to terminate before it can execute, but up to this point, no explicit acceptance of a task's destructor has occurred in any task so how does the call get accepted? The mechanism follows from the fact that after a task's thread terminates, the task becomes a monitor (see Section 10.6, p. 327). Hence, when the task's thread exits the mutex object (by termination), the implicit scheduling first checks the acceptor/signalled stack (which is empty) and then checks the entry queue (which has the call to the destructor). Thus, the last action done by the terminating task is to implicitly schedule the call to its destructor (cooperation). If there are other outstanding calls ahead of the destructor call, it is responsibility of the task designer to ensure the call to the destructor is eventually implicitly accepted, which can be accomplished by having the other tasks immediately exit the monitor.

With this deeper understanding of how implicit scheduling accepts the destructor, it is now possible to examine explicit scheduling of the destructor. Which leads back to the original question: why accept the destructor of a mutex object, especially a task? The answer is to provide a mechanism for determining when to terminate a task's thread. Up to now, tasks have been self terminating, i.e., they perform a set piece of work, and once completed, they terminate, which through a series of circumstances (see above) causes the destructor to be implicitly accepted. What if a task needs to terminate because of an external event? That is, one task tells another it is time to stop execution and terminate. The only mechanism to accomplish this is with indirect communication by using a shared variable that is set by one task and polled for change by the other; but polling is unacceptable. It is now possible to use direct communication using a public member like the stop member in the consumer coroutine of Figure 4.15, p. 90, which sets a flag to stop the loop in the task's main member. Hence, it is possible to stop the infinite loop in the task main of the bounded buffer task by augmenting it to:

```
template<class ELEMTYPE> _Task BoundedBuffer {
        . . .
        bool done;                    // indicates if task should terminate
    public:
        BoundedBuffer() : . . ., done( false ) {}
        . . .
        void stop() { done = true; }
    protected:
        void main() {
            for ( ; ! done; ) {
                _Accept( stop );
                else _When ( count < 20 ) _Accept( insert );
                else _When ( count > 0 ) _Accept( remove );
            . . .
```

Now, another task can call the stop member at any time, as in the following:

```
{
    BoundedBuffer buf;
    {
        Consumer cons( buf );        // pass buffer to consumer task
        Producer prod( buf );        // pass buffer to producer task
    } // implicit call to consumer and producer destructor (wait for termination)
    buf.stop();
} // implicit call to buffer destructor (no tasks using the buffer now)
```

While this approach works, and is essential in certain circumstances (see Section 10.9, p. 334), notice that the explicit call to stop occurs immediately before the implicit call to the destructor, which is an unfortunate duplication as both calls are involved in the termination and deallocation of the buffer task. Therefore, instead of having a stop routine and a flag variable, it is possible to simply accept the destructor, as in:

```
void main() {
    for ( ;; ) {
        _Accept ( ~BoundedBuffer )
            break;
        else _When (count != 20) _Accept(insert);
        else _When (count != 0) _Accept(remove);
    }
    // clean up code
}
```

Now when the implicit call occurs to the destructor, it is detected by the buffer task and the loop can be exited, the task can clean up (e.g., free storage or close files), and finally the task's thread terminates by returning from the task main. What is happening here is the task calling the destructor and the task accepting the destructor are synchronizing without communicating (i.e., neither stop nor the destructor have parameters nor return a result). The synchronization is sufficient for the called task to know it must terminate.

While accepting the destructor appears to provide a convenient mechanism of informing a task to terminate, there is a problem: the semantics of accepting a mutex member cause the acceptor to block and the caller to execute. Now if the call to the destructor is executed, the destructor completes and the object is deallocated. Hence, the task's thread does not restart to finish the accept statement nor the task main because the task object is deallocated. (This behaviour is similar to deallocating a coroutine that has not terminated.) Therefore, with the current semantics for the accept statement it is impossible for the task's thread to restart and perform clean up in the task main because it is deallocated immediately after accepting the destructor. As a result, all clean up is forced into the destructor. However, this restriction may force variables that logically belong in the task's main into the task, making these variables have a greater visibility than is necessary from a software engineering standpoint. Furthermore, the fact that control does not return to the **_Accept** statement when the destructor is accepted seems confusing, as that is what occurs in all other cases.

To solve this problem, the semantics for accepting a destructor are different from accepting a normal mutex member. When the destructor is accepted, the caller is blocked and pushed onto the acceptor/signalled stack instead of the acceptor. Therefore, control restarts at the accept statement *without* executing the destructor, which allows a task to clean up before it terminates. When it terminates, the call to the destructor is popped off the acceptor/signalled stack (remember it is at the top of the stack) by the implicit scheduling and the destructor executes followed by the deallocation of the storage. Interestingly, the semantics for accepting the destructor are the same as for a signal (see page 274), where the acceptor task is the signaller task and the task calling the destructor is the signalled task.

One common error made with task termination is for a destructor call, intended as a signal to stop a task, to occur too early in a computation. Once accepted, the task's thread can terminate and then the task is de-allocated, all before the computation is complete, which almost always results in an error. This problem is demonstrated by modifying the example using the stop routine to:

```
{
    BoundedBuffer buf;
    Consumer cons( buf );        // pass buffer to consumer task
    Producer prod( buf );        // pass buffer to producer task
    buf.stop();                  // call too early: cons and prod tasks still executing
} // implicit calls to destructor for prod, cons, and buf (in this order)
```

The previous inner block seems superfluous because deallocation occurs in reverse order to allocation. However, the call to buf.stop() now occurs *before* the calls to the destructor for prod and cons because these calls now occur at the end of the outer block instead of in the middle due to the previous inner block. As a result, the call to the buffer's destructor can occur while the producer and consumer tasks are using the buffer, which results in the buffer terminating and cleaning up, becoming a monitor, and most likely failing thereafter when used again by a producer or consumer. (In fact, the bounded-buffer task using internal scheduling does work correctly in this scenario but the external scheduling version does not. Why does one work and the other not work?) Interestingly, the previous example does work correctly if the bounded-buffer task accepts its destructor instead of using the stop member, as in:

```
{
    BoundedBuffer buf;
    Consumer cons( buf );        // pass buffer to consumer task
    Producer prod( buf );        // pass buffer to producer task
} // implicit calls to destructor for prod, cons, and buf
```

The reason is that the destructor for the bounder-buffer task is called last on block exit so it cannot be deallocated until after the producer and consumer have finished using it.

However, it is still possible to generate a problem scenario when accepting the destructor. One common mistake is to have the consumer and/or producer task accept its destructor in order to know when to stop production and/or consumption. Again, the problem is that the call to the destructor occurs immediately at the end of the block in which the producer and consumer are declared, possibly before the task's main starts execution. As a result, the first call accepted is the destructor call, so the tasks terminate without having done any work. Only if there is a delay *before* the end of the block can the producer and consumer make progress, as in:

```
{
    BoundedBuffer buf;
    Consumer cons( buf );        // pass buffer to consumer task
    Producer prod( buf );        // pass buffer to producer task
    // delay for some period of time
} // implicit calls to destructor for prod, cons, and buf
```

The problem with this approach is that after the time delay, both the producer and consumer may accept their destructors immediately, and as a result, the consumer does not have the opportunity to clean out all the elements in the buffer before it is deleted, which may result in an error. To ensure the buffer is emptied, the consumer has to check the buffer state in its clean up code at the end of its task main, and continue processing data until the buffer state becomes empty.

Another way to generate a problem scenario is with dynamic allocation, as in:

```
template<class ELEMTYPE> _Task BoundedBuffer {
    . . .
  public:
    . . .
    void insert( ELEMTYPE elem ) {
        Elements[back] = elem;
    }
    ELEMTYPE remove() {
        return Elements[front];
    }
  protected:
    void main() {
        for ( ;; ) {
            _When ( count < 20 ) _Accept( insert ) {
                back = (back + 1) % 20;
                count += 1;
            } else _When ( count > 0 ) _Accept( remove ) {
                front = (front + 1) % 20;
                count -= 1;
            }
        }
    }
};
```

Figure 10.8: Task Bounded Buffer: Performing Work

```
{
    BoundedBuffer *buf = new BoundedBuffer;
    Consumer *cons = new Consumer( buf );
    Producer *prod = new Producer( buf );
    delete buf;
    delete cons;
    delete prod;
}
```

Here the order of deallocation is the same at the order of allocation, instead of the reverse order (as at the end of a block). As a result, the destructor for the buffer is called immediately, and it terminates without having been used or in the middle of usage by the producer and consumer.

10.8 When to Create a Task

While the example of the bounded-buffer task allowed many of the "dos and don'ts" of a task with public members to be illustrated, it is not a good example of when to use a task. A task is largely inappropriate for a bounded-buffer because the thread of the bounded-buffer task has no work to do with respect to management of the buffer. In fact, the task thread slows down the entire process by alternating execution between calling tasks, and yet does nothing when it restarts.

Figure 10.8 shows that it is possible to move some of the buffer administration from the member routines into the task main. Notice members insert and remove now contain the absolute minimum of code needed to deliver or retrieve an element to/from the buffer. Therefore, the producer and consumer now do less work in the buffer, and hence, can spend more time producing or consuming elements, respectively. However, the buffer management work has not disappeared; it has been moved into the accept statement in the buffer's task main. When the buffer's thread restarts after accepting a call, it performs the appropriate buffer management work for the particular call that was accepted. Now the buffer is actually doing some useful work each time through the loop. Thus, on a multiprocessor computer, there would be a slight speedup because there is now additional concurrency in the program resulting in increased parallelism, i.e., the production and consumption of elements overlaps with the buffer management.

Unfortunately, the amount of additional concurrency generated by executing two lines of buffer management by another task is insufficient to cover the cost of the blocking and unblocking of the buffer's thread for each call to the

```
_Task Cons {
    uCondition check;
    Prod &prod;          // communication
    int p1, p2, status;
    void main() {
        int money = 1, receipt;
        for ( ;; ) {
            _Accept( stop ) {
                break;
            } else _Accept( delivery ) {
                cout << "cons receives: " <<
                    p1 << ", " << p2;
                status += 1;
                check.signalBlock();
                cout << " and pays $" <<
                    money << endl;
                receipt = prod.payment( money );
                cout << "cons receipt #" <<
                    receipt << endl;
                money += 1;
            }
        }
        cout << "cons stops" << endl;
    }
  public:
    Cons( Prod &p ) : prod(p), status(0) {}
    int Cons::delivery( int p1, int p2 ) {
        Cons::p1 = p1; Cons::p2 = p2;
        check.wait();      // let cons check elements
        return status;
    }
    void stop() {}
};
```

```
_Task Prod {
    Cons *cons;          // communication
    int N, money, receipt;
    void main() {
        int i, p1, p2, status;
        _Accept( start );
        for ( i = 1; i <= N; i += 1 ) {
            p1 = rand() % 100;
            p2 = rand() % 100;
            cout << "prod delivers: " <<
                p1 << ", " << p2 << endl;
            status = cons->delivery( p1, p2 );
            _Accept( payment );
            cout << "prod status: " <<
                status << endl;
        }
        cons->stop();
        cout << "prod stops" << endl;
    }
  public:
    Prod() : receipt(0) {}
    int payment( int money ) {
        Prod::money = money;
        cout << "prod payment of $" <<
            money << endl;
        receipt += 1;
        return receipt;
    }
    void start( int N, Cons &c ) {
        Prod::N = N; cons = &c;
    }
};
void uMain::main() {
    Prod prod;
    Cons cons( prod );
    prod.start( 5, cons );
}
```

Figure 10.9: Bidirectional Task: Producer-Consumer

buffer. In other words, the administrative cost is greater than the cost of having the tasks do the work themselves. People constantly have to perform this kind of cost-benefit analysis in their lives: do you hire someone to get work done faster, or is it faster to work a little more and do the work yourself? The next chapter explores this question in detail.

10.9 Producer-Consumer Problem

Figure 10.9 shows a task based solution for the producer-consumer problem derived from the full-coroutine version in Figure 4.16, p. 92. Because information flows bidirectionally between the producer and consumer, a communication cycle is required, which necessitates mutual references. The mutual reference is accomplished using the approach mentioned in Section 4.7.2, p. 86, that is, pass the final partner after the other task is instantiated, which is accomplished through the Prod::start member.

Both producer and consumer tasks start execution after their declaration. The producer task's main immediately accepts its start member to obtain the number of elements it is to produce and its consumer partner. The main member executes N iterations of generating two random integer values between 0–99, printing the two values, calling the consumer to deliver the two values, accepting the payment, and printing the status returned from the consumer.

The call from the producer to the consumer's delivery routine transfers the generated values. When a call to delivery is accepted, it is the producer task, prod, executing the member. The values delivered by the producer are copied into communication variables in the consumer and the producer blocks on a conditional variable (internal scheduling). The blocking restarts cons where it accepts delivery so it can examine the transfered values and prepare a status value to be returned.

The consumer task's main member iterates accepting calls to either its stop or delivery members. After accepting a call to the stop member, the loop terminates. The stop member is used solely for synchronization so it contains no code. After accepting a call to the delivery member, the delivered values are printed, the return status is incremented, and the consumer task signals the producer to restart so the producer can return the status. A signalBlock is used to ensure the producer is restarted immediately. The consumer restarts after the producer exits the delivery member, prints the amount it pays for the values, calls back to the producer's payment member with the money, prints the receipt from the producer, and increments the amount of money for the next payment.

The delivery member returns the status value to the call in prod's main member, where the status is printed. The loop then repeats calling delivery, where each call is accepted by the consumer task. When the consumer restarts, it continues in the statement after **_Accept**(delivery). The consumer restarts the producer and calls prod's payment, where these calls only restarts the producer after the member exits, not during the member, as for delivery. These operations must be done in this order, otherwise the call to payment results in a mutual-exclusion deadlock as prod's thread is blocked on condition check preventing calls to prod.

After iterating N times, the producer call the stop member in cons. Notice, member stop is being accepted along with delivery each time through the loop in Cons::main. When the call is accepted, the producer exits immediately. The consumer then continues terminating its loop, prints a termination message, and exits the main member terminating the consumer's thread, which makes it a monitor so its destructor is now accepted. The producer also continues, prints a termination message and exits its main member terminating the producer's thread, which makes it a monitor so its destructor is now accepted. Task uMain is restarted as it was previously blocked on calls to the destructors of the producer and consumer, which are now accepted. uMain then deallocates the producer and consumer, and exits its main member.

The consumer uses a stop member instead of accepting its destructor because the call to its destructor occurs too early (in uMain). Accepting the destructor could have been used if the producer created the consumer instead of uMain; however, such restructuring is not always possible. The producer does not need a stop member or to accept its destructor because it is self terminating.

10.10 Tasks and Coroutines

It is a common mistake to assume a coroutine created by a task is owned by it and only that task can use it. In fact, any coroutine can be "passed off" to another task. The only requirement is that only one task use the coroutine at a time. To ensure mutual execution of coroutine execution, the passing around of a coroutine must involve some form of cooperation such that only one task uses the coroutine at a time. Alternatively, the coroutine can be made a coroutine-monitor to provide mutual exclusion. However, the cost in locking and unlocking the coroutine-monitor may be unnecessary if serial usage cooperation can be established.

In Chapter 4, p. 65, it was shown there is a weak equivalence between a coroutine and a routine or class. There is also a weak equivalence between a coroutine and a task, because both have a separate execution state for their distinguished member. However, simulating a coroutine with a task is non-trivial because the organizational structure of a coroutine and a task are different. Furthermore, simulating full coroutines that form a cyclic call-graph may be impossible with tasks because a task's mutual exclusion may disallow multiple entries by the same task, which would cause deadlock. Finally, a task is inefficient for this purpose because of the higher cost of switching both a thread and execution state as opposed to just an execution state. In μC++, the cost of communication with a coroutine is, in general, less than half the cost of communication with a task, unless the communication is dominated by transferring large amounts of data.

10.11 Inheritance Anomaly Problem

As for class, coroutine, monitor and coroutine-monitor, it is possible for tasks to inherit from one another to establish different kinds of reuse, i.e., subtyping and implementation. When task's inherit from one another there is a problem, called **inheritance anomaly**. This problem results from the need to duplicate synchronization logic in the subclass's task-main. ...

10.12 Summary

A concurrent system needs two forms of communication, indirect and direct, to handle two different kinds of interactions among tasks. In direct communication, a task calls another task's member routine, which implies a task needs the mutex property to prevent simultaneous access of its data by the calling thread and the task's thread. Because a task has the mutex property, it also has the capability to schedule calls to it using external scheduling and calling tasks within it using internal scheduling. Accepting the destructor of a task is a useful mechanism for knowing when a task should stop executing.

10.13 Questions

1. Assume a series of nested accepts has occurred in a task so there are several tasks blocked on the acceptor/signalled stack. Now the task accepts the destructor. Show how tasks below the task accepting the destructor on the acceptor/signalled stack can be released so the task's destructor terminates without error.

2. Write a program that plays the following simple card game. Each player takes a number of cards from a deck of cards and passes the deck to the player on the left. A player must take at least one card and no more than a certain maximum. The player who takes the last card wins.

   Each player is a task with the following interface (you may add only a public destructor and private members):

   ```
   _Task player {
     public:
       player();
       void start( player &partner );
       void play( int deck );
   };
   ```

   Member start is called after creation to supply the player on the left. Member play is called by a player's partner to pass on the remaining deck of cards after a play has made a play. Task uMain starts the game by passing the deck of cards to one of the players, and that player begins the game.

   Write a program that plays 3 games sequentially (i.e., one game after the other, not 3 concurrent games). For each game, generate a random number of players in the range from 2 to 8 inclusive, and a random number of cards in the deck in the range from 20 to 52 inclusive. Have each player follow the simple strategy of taking a random number of cards in the range from 1 to 5 (i.e., do not get fancy).

   The output should show a dynamic display of the game in progress. That is, which task is taking its turn, how many cards it received from the player on the right, and how many cards it took. At the end of each game, make sure that all tasks terminate.

   Use only direct communication among the players for task communication, i.e., no monitors or semaphores.

3. Consider a vault protected by three locks A, B, and C, each operated by its own unique key. Suppose there are the same number of guards as keys, and each guard carries a single key for the corresponding lock A, B, and C, respectively. Each of the guards must periodically enter the vault alone to verify that nothing has been stolen. A supervisor possesses one copy of each key, but never leaves her office.

   To enter the vault, each guard must request from the supervisor each of the two additional keys needed to enter the vault, but may request only one key at a time. The supervisor replies to a key request when that key is available. Once in possession of all three keys, a guard checks the vault and returns the two borrowed keys one at a time. The supervisor only receives requests for keys, and each guard only makes requests to the supervisor for keys.

   Write a μC++ program that simulates the above scenario. The supervisor has the following user interface:

```
_Task Supervisor {
  public:
    void get( int who, int key );
    void put( int who, int key );
};
```

A guard calls member get, passing who they are (a value in the range 0 to $(N - 1)$, where N is the number of keys to open the vault), and the value of the key they are trying to get from the supervisor (a value in the range 0 to $(N - 1)$, where N is the number of keys to open the vault). A guard calls member put, passing who they are and the value of the key they are returning to the supervisor. The guard has the following user interface:

```
_Task Guard {
    int who;
    Supervisor &super;

    void CheckVault() {
        yield( rand() % 20 );
    }

    void CoffeeBreak() {
        yield( rand() % 20 );
    }
  public:
    Guard( int who, Supervisor &super ) : who(who), super(super) {}
};
```

Each guard is started with the value of who they are (a value in the range 0 to $(N-1)$, where N is the number of keys to open the vault), and a reference to the supervisor. A guard enters the vault 10 times, each time acquiring the necessary $N - 1$ keys. When a guard has the keys needed to check the vault, the routine CheckVault() is called. After checking the vault, the guard returns the $N - 1$ keys and calls CoffeeBreak().

You must devise a solution that the guards follow so they do not deadlock (any solution is allowed as long as it is not grossly inefficient). (Hint: use a simple solution discussed in the notes.) Explain why your solution is deadlock free.

4. Willy, Wally, Wesley and Warren are four workers on an assembly line. Each has their own position on the line and each does one step in the assembly of some product. Sally is the supervisor of the assembly line, and her responsibility is to move the assembly line forward one position when all of the workers are finished with the work that is currently at their position. Willy, Wally, Wesley and Warren each (independently) tell Sally when they are finished with their work (ready for the line to move).

   Ian is a roving inspector who checks to see if the workers are doing their work properly. When Ian wishes to do an inspection, he sends a message to Sally and gets back a list of workers who are finished with the work at their station. (If no workers are finished at the time Ian wants to inspect them, then Sally waits until at least two are finished before responding to Ian's message.) Then Ian randomly chooses one of these finished workers, goes to their position and inspects their work. NOTE: During an inspection, the line is NOT allowed to move. After the inspection, Ian reports his findings to Sally. If Ian is unsatisfied with the work, Sally must inform the appropriate worker to re-do the job before allowing the assembly line to move.

   As a real-world analogue, think of a car assembly line with 4 workers installing 4 doors on a car. A car appears in front of Willy, Wally, Wesley and Warren at essentially the same time and they start installing the doors. The car cannot move until the doors are all on the car and Sally is told this. As soon as Sally moves the car forward, a new car appears in front of Willy, Wally, Wesley and Warren for them to work on. You may assume that Willy, Wally, Wesley and Warren never have to wait for doors after a car appears in front of them. Ian is walking up and down the car assembly line, stopping at different stations to inspect the work.

   Simulate the assembly line. Willy, Wally, Wesley, Warren, Sally and Ian are all tasks. All communication between tasks must be direct. To simulate the time required by a worker to do his work, have the worker pick a random N between 10 and 15, and then yield(N). To simulate the time required by Ian to do his work, have him pick a random M between 5 and 10, and then yield(M). Assume that in each inspection, there is a 75% chance that Ian will be satisfied with the work. Note that if Ian is not satisfied, he demands that the work be re-done but does not re-check it afterwards. After each inspection, Ian takes a coffee break before doing the next inspection. Simulate the coffee break by having Ian pick a random B between 30 and 50 and then yield(B). The simulation ends after the assembly line has moved forward some given number of times, say 50.

Test your simulation thoroughly and explain your tests. Feel free to change the parameters (e.g., coffee break delay time) to aid in your testing.

NOTE: Do not generate large amounts of output. Part of your job is to produce concise test cases and present them in a nice way. This means every task cannot just print what it is doing.

Chapter 11

Enhancing Concurrency

Up to this point, the concurrency discussion has been at the level of programming in the small. That is, how to write small concurrent programs involving small numbers of tasks or how to write fragments of larger concurrency programs. It is now time to examine techniques that are used to construct large concurrent programs.

> Although some have argued against a strongly anthropomorphic view of process structuring, we think it is essential in the initial stages of problem decomposition and structuring for concurrency or parallelism. There is ample opportunity for establishing mathematical properties and proofs after the structure is chosen. All branches of science and engineering rely heavily on analogies and visualization to prompt insight and intuition, and the closest experience most of us have with collections of asynchronous processes is the organizations of people. Moreover, organizations of people function effectively (through perhaps not ideally) with a much higher degree of parallelism than programs anyone has yet been able to build. . . . The human organization analogy is of considerable help in choosing what processes should be used, what their responsibilities should be, and how they should interact. It even helps in suggesting what information should be communicated between processes, and when they need to synchronize. [Gen81, p. 445]

- 2 task involved in direct communication: client (caller) & server (callee)
- possible to increase concurrency on both the client and server side

11.1 Server Side

- Use server thread to do administrative work so client can continue concurrently (assuming no return value).
- E.g., move administrative code from the member to the statement executed after the member is accepted:

```
_Task t1 {                          _Task t2 {
  public:                             public:
    void xxx(...) { S1 }                void xxx(...) { S1.copy }
    void yyy(...) { S2 }                void yyy(...) { S2.copy }
    void main() {                       void main() {
        ...                                 ...
        _Accept( xxx );                     _Accept( xxx ) { S1.admin }
        else _Accept( yyy );                    else _Accept( yyy ) { S2.admin };
    }                                   }
}                                   }
```

To achieve greater concurrency in the bounded buffer, change to:

```
    void insert( int elem ) {
        Elements[back] = elem;
    }
    int remove() {
        return Elements[front];
    }
  protected:
    void main() {
        for ( ;; ) {
            _When (count != 20) _Accept(insert) {
                back = (back + 1) % 20;
                count += 1;
            } else _When (count != 0) _Accept(remove) {
                front = (front + 1) % 20;
                count -= 1;
            }
        }
    }
```

- overlap between client and server increases potential for concurrency

In task t1, a caller of member routines xxx or yyy is blocked until code *code1* or *code2* is executed, respectively. In task t2, control returns back to the caller after the parameters have been copied into local variables of the task. Then the local variables are manipulated in *code1* or *code2* of the accept statement, respectively, by the task.

11.1.1 Buffers

The previous technique provides buffering of size one between the caller and the receiving task. If a call is made before the receiving task can execute the code in the statements of the accept statement, the call waits. To allow the caller to get further ahead, a buffer must be used to store the arguments of the call until the receiver processes them. Appendix 7.5.2, p. 201 shows several ways that generic bounded buffers can be built. However, unless the average time for production and consumption is approximately equal with only a small variance, the buffer is either always full or empty.

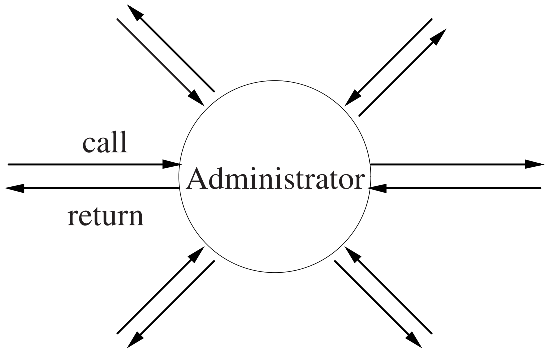
11.1.1.1 Internal Buffer

- The previous technique provides buffering of size 1 between the client and server.
- Use a larger internal buffer to allow clients to get in and out of the server faster?
- I.e., an internal buffer can be used to store the arguments of multiple clients until the server processes them.
- However, there are several issues:
  - Unless the average time for production and consumption is approximately equal with only a small variance, the buffer is either always full or empty.
  - Because of the mutex property of a task, no calls can occur while the server is working, so clients cannot drop off their arguments.

    The server could periodically accept calls while processing requests from the buffer (awkward).
  - Clients may need to wait for replies, in which case a buffer does not help unless there is an advantage to processing requests in non-FIFO order.
- These problems can be handled by changing the server into an administrator.

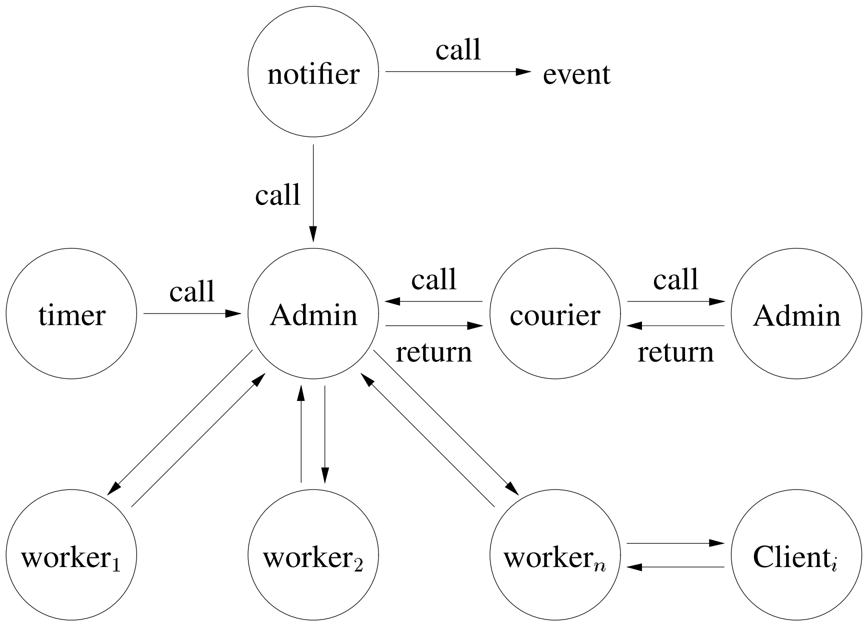
11.1.2 Administrator

- An administrator is used to manage a complex interaction or complex work or both.
- The key is that an administrator does little or no work; its job is to manage.
- Management means delegating work to others, receiving and checking completed work, and passing completed work on.
- An administrator is called by others; hence, an administrator is always accepting calls.



- An administrator rarely makes a call to another task; the reason is that calls may block putting the administrator to sleep.
- An administrator usually maintains a list of work to pass to "workers".
- Typical workers are:

  timer - prompt the administrator at specified time intervals
  notifier - perform a potentially blocking wait for an external event (key press)
  simple worker - do work given to them by and return the result to the administrator
  complex worker - do work given to them by administrator and interact directly with client of the work
  courier - perform a potentially blocking call on behalf of the administrator



11.2 Client Side

While μC++ provides only synchronous communication facilities, it is possible to build asynchronous facilities out of the synchronous ones. The following discussion shows some techniques for doing this.

When a call occurs to a mutex member, the caller blocks until the mutex member completes. If the purpose of the call is to only transfer data to be manipulated by the receiver, the caller should not have to wait until the receiver performs the manipulation. This can be handled by moving code from the member routine to the statement executed after the member is accepted, as in:

- μC++ provides only synchronous communication.
  - i.e., the caller blocks from the time the arguments are delivered to the time the result is returned (like a procedure call).
- It is possible to build asynchronous facilities out of the synchronous ones.
  - i.e., the caller blocks only to deliver the arguments and then continues, picking up the result at a later time.
- If the call only transfers data, the caller should not have to wait while the data is manipulated.
- Handled by moving code from the member to the statement executed after the member is accepted:

11.3 Returning Values

- Neither of the previous cases provide asynchrony if a call supplies arguments *and* a result is returned.
- To achieve asynchrony in this case, the single call must be transformed into two calls:
  1. transmit the arguments
  2. retrieve the results
- The time between the two calls allows the calling task to execute asynchronously with the task performing the operation on the caller's behalf.
- If the result is not ready when the second call is made, the caller blocks or the caller has to call again (poll).
- However, this requires a protocol so that when the client makes the second call, the correct result can be found and returned.

11.3.1 Tickets

- One form of protocol is the use of a token or ticket.
- The first part of the protocol transmits the arguments specifying the desired work and a ticket (like a laundry ticket) is returned immediately.
- The second call passes the ticket to retrieve the result.
- The ticket is matched with a result, and the result is returned if available or the caller is blocked until the result for that ticket is available.
- However, protocols are error prone because the caller may not obey the protocol (e.g., never retrieve a result or use the same ticket twice).

11.3.2 Call-Back Routine

- Another protocol is to transmit (register) a routine on the initial call.
- When the result is ready, the routine is called by the task generating the result, passing it the result.
- The call-back routine cannot block the caller; it can only store the result and set an indicator (e.g., V a semaphore) known to the original caller.
- The original caller must *poll* the indicator or block until the indicator is set.
- The advantage is that the producer does not have to store the result, but can drop it off immediately.
- Also, the consumer can write the call-back routine, so they can decide to poll or block or do both.

11.3.3 Futures

Neither of the previous cases can provide any asynchrony if a result is returned as part of the call. To achieve asynchrony when a result is required, the single call must be transformed into two calls: the first call to transmit the arguments and a subsequent call to retrieve the result. The time between the two calls allows the calling task to execute asynchronously with the task performing the operation on the caller's behalf. If the result is not ready when the second call is made, the caller blocks. However, this requires a protocol so that when the caller makes the second call and the correct results can be returned.

One form of protocol is the use of a token or ticket. The first part of the protocol transmits the arguments specifying the desired work and a ticket (like a laundry ticket) is returned immediately. The second call passes the ticket to retrieve the result. The ticket is matched with a result, and the result is returned if available or the caller is blocked until the result for that ticket is available. However, protocols are error prone because the caller may not obey the protocol (e.g., never retrieve a result or use the same ticket twice).

A **future** [Hal85] is a mechanism to provide the same asynchrony as above but without an explicit protocol. The protocol becomes implicit between the future and the task generating the result. Furthermore, it removes the difficult problem of when the caller should try to retrieve the results. In detail, a future is an object that is a subtype of the result type expected by the caller. Instead of two calls as before, a single call is made, passing the appropriate arguments, and a future is returned. In general, the future is return immediately and it is empty. In other words, the caller "believes" the call completed and continues execution with an empty result value. The future is filled in after the actual result is calculated. If the caller tries to use the future before its value is filled in, the caller is implicitly blocked.

A simple future can be constructed out of a semaphore and link field, as in:

```
class future {
    friend _Task server;            // allow server to access internal state

    uSemaphore resultAvailable;
    future *link;
    ResultType result;
  public:
    future() : resultAvailable( 0 ) {}

    ResultType get() {
        resultAvailable.P();        // wait for result
        return result;
    }
};
```

The semaphore is used to block the caller if the future is empty and the link field is used to chain the future onto a work list of the task generating results. Unfortunately, the syntax for retrieving the value of the future is awkward as it requires a call to the get routine.

- A **future** provides the same asynchrony as above but without an explicit protocol.
- The protocol becomes implicit between the future and the task generating the result.

- Furthermore, it removes the difficult problem of when the caller should try to retrieve the result.
- In detail, a future is an object that is a subtype of the result type expected by the caller.
- Instead of two calls as before, a single call is made, passing the appropriate arguments, and a future is returned.
- The future is returned immediately and it is empty.
- The caller "believes" the call completed and continues execution with an empty result value.
- The future is filled in at some time in the "future", when the result is calculated.
- If the caller tries to use the future before its value is filled in, the caller is implicitly blocked.
- A simple future can be constructed out of a semaphore and link field, as in:

```
class future {
    friend _Task producer;       // can access internal state

    uSemaphore resultAvailable;
    future *link;
    ResultType result;
  public:
    future() : resultAvailable( 0 ) {}

    ResultType get() {
        resultAvailable.P();     // wait for result
        return result;
    }
};
```

  - the semaphore is used to block the caller if the future is empty
  - the link field is used to chain the future onto a work list of the producer.

- Unfortunately, the syntax for retrieving the value of the future is awkward as it requires a call to the get routine.
- Also, in languages without garbage collection, the future must be explicitly deleted.

11.4 Concurrent Exception Communication

Concurrent exceptions not only changed control but also communicate information via exception parameters. This communication can be complex.

11.4.1 Communication

The source execution needs to deliver an exception to the faulting execution for propagation. This requires a form of direct communication not involving shared objects. In essence, an exception is transmitted from the source to the faulting execution.

There are two major categories of direct communication: blocking and nonblocking. In the first, the sender blocks until the receiver is ready to receive the event; in the second, the sender does not block.

11.4.1.1 Source Execution Requirement

Using blocking communication, the source execution blocks until the faulting execution executes a complementary receive. However, an execution may infrequently (or never) check for incoming exception events. Hence, the source can be blocked for an extended period of time waiting for the faulting execution to receive the event. Therefore, blocking communication is rejected. Only nonblocking communication allows the source execution to raise an exception on one or more executions without suffering an extended delay.

11.4.1.2 Faulting Execution Requirement

Nonblocking communication for exceptions is different from ordinary nonblocking communication. In the latter case, a message is delivered only after the receiver executes some form of receive. The former requires the receiver to receive an exception event without explicitly executing a receive because an EHM should preclude checking for an abnormal condition. The programmer is required to set up a handler only to handle the rare condition. From the programmer's perspective, the delivery of an asynchronous exception should be transparent. Therefore, the runtime system of the faulting execution must poll for the arrival of asynchronous exceptions, and propagate it on arrival. The delivery of asynchronous exceptions must be timely, but not necessarily immediate.

There are two polling strategies: **implicit polling** and **explicit polling**. Implicit polling is performed by the underlying system. (Hardware interrupts involve implicit polling because the CPU automatically polls for the event.) Explicit polling requires the programmer to insert explicit code to activate polling.

Implicit polling alleviates programmers from polling, and hence, provides an apparently easier interface to programmers. On the other hand, implicit polling has its drawbacks. First, infrequent implicit polling can delay the handling of asynchronous exceptions; polling too frequently can degrade the runtime efficiency. Without specific knowledge of a program, it is difficult to have the right frequency for implicit polling. Second, implicit polling suffers the nonreentrant problem (see Section 11.4.2).

Explicit polling gives a programmer control over when an asynchronous exception can be raised. Therefore, the programmer can delay or even completely ignore pending asynchronous exceptions. Delaying and ignoring asynchronous exceptions are both undesirable. The other drawback of explicit polling is that a programmer has to worry about when to and when not to poll, which is equivalent to explicitly checking for exceptions.

Unfortunately, an EHM with asynchronous exceptions needs to employ both implicit and explicit polling. Implicit polling simplifies using the EHM and reduces the damage a programmer can do by ignoring asynchronous exceptions. However, the frequency of implicit polling should be low to avoid unnecessary loss of efficiency. Explicit polling allows programmers to have additional polling when it is necessary. The combination of implicit and explicit polling gives a balance between programmability and efficiency. Finally, certain situations can require implicit polling be turned off, possibly by a compiler or runtime switch, e.g., in low-level system code where execution efficiency is crucial or real-time programming to ensure deadlines.

11.4.2 Nonreentrant Problem

Concurrent events introduce a form of concurrency into sequential execution because delivery is nondeterministic with implicit polling. The event delivery can be considered as temporarily stealing a thread to execute the handler. As a result, it is possible for a computation to be interrupted while in an inconsistent state, a handler to be found, and the handler recursively call the inconsistent computation, called the **nonreentrant problem**. For example, while allocating memory, an execution is suspended by delivery of an asynchronous event, and the handler for the exception attempts to allocate memory. The recursive entry of the memory allocator may corrupt its data structures.

The nonreentrant problem cannot be solved by locking the computation because either the recursive call deadlocks, or if recursive locks are used, reenters and corrupts the data. To ensure correctness of a nonreentrant routine, an execution must achieve the necessary mutual exclusion by blocking delivery, and consequently the propagation of asynchronous exceptions, hence temporarily precluding delivery.

Hardware interrupts are also implicitly polled by the CPU. The nonreentrant problem can occur if the interrupt handler enables the interrupt and recursively calls the same computation as has been interrupted. However, because hardware interrupts can happen at times when asynchronous exceptions cannot, it is more difficult to control delivery.

11.4.3 Disabling Concurrent Exceptions

Because of the nonreentrant problem, facilities to disable asynchronous exceptions must exist. There are two aspects to disabling: the specific event to be disabled and the duration of disabling. (This discussion is also applicable to hardware interrupts and interrupt handlers.)

11.4.3.1 Specific Event

Without derived exceptions, only the specified exception is disabled; with derived exceptions, the exception and all its descendants can be disabled. Disabling an individual exception but not its descendents, called **individual disabling**, is tedious as a programmer must list all the exceptions being disabled, nor does it complement the exception hierarchy. If a new derived exception should be treated as an instance of its ancestors, the exception must be disabled wherever its ancestor is disabled. Individual disabling does not automatically disable the descendents of the specified exceptions, and therefore, introducing a new derived exception requires modifying existing code to prevent it from activating a handler bound to its ancestor. The alternative, **hierarchical disabling**, disables an exception and its descendents. The derivation becomes more restrictive because a derived exception also inherits the disabling characteristics of its parent. Compared to individual disabling, hierarchical disabling is more complex to implement and usually has a higher runtime cost. However, the improvement in programmability makes hierarchical disabling attractive.

A different approach is to use priorities instead of hierarchical disabling, allowing a derived exception to override its parent's priority when necessary. Selective disabling can be achieved by disabling exceptions of priority lower than or equal to a specified value. This selective disabling scheme trades off the programmability and extensibility of hierarchical disabling for lower implementation and runtime costs. However, the problem with priorities is assigning priority values. Introducing a new exception requires an understanding of its abnormal nature plus its priority compared to other exceptions. Hence, defining a new exception requires an extensive knowledge of the whole system with respect to priorities, which makes the system less maintainable and understandable. It is conceivable to combine priorities with hierarchical disabling; a programmer specifies both an exception and a priority to disable an asynchronous exception. However, the problem of maintaining consistent priorities throughout the exception hierarchy still exists. In general, priorities are an additional concept that increases the complexity of the overall system without significant benefit.

Therefore, hierarchical disabling with derived exceptions seems the best approach in an extensible EHM. Note that multiple derivation (see Section 3.7.1, p. 37) only complicates hierarchical disabling, and the same arguments can be used against hierarchical disabling with multiple derivation.

11.4.3.2 Duration

The duration for disabling could be specified by a time duration, but normally the disabling duration is specified by a region of code that cannot be interrupted. There are several mechanisms available for specifying the region of uninterruptable code.

One approach is to supply explicit routines to turn on and off the disabling for particular asynchronous exceptions. However, the resulting programming style is like using a semaphore for locking and unlocking, which is a low-level abstraction. Programming errors result from forgetting a complementary call and are difficult to debug.

An alternative is a new kind of block, called a **protected block**, which specifies a list of asynchronous events to be disabled across the associated region of code. On entering a protected block, the list of disabled asynchronous events is modified, and subsequently enabled when the block exits. The effect is like entering a guarded block so disabling applies to the block and any code dynamically accessed via that block, e.g., called routines.

An approach suggested for Java [Rea99] associates the disabling semantics with an exception named AIE. If a member routine includes this exception in its exception list, interrupts are disabled during execution of the member; hence, the member body is the protected block. However, this approach is poor language design because it associates important semantics with a name, AIE, and makes this name a hidden keyword.

The protected block seems the simplest and most consistent in an imperative language with nested blocks. Regardless of how asynchronous exceptions are disabled, all events (except for special system events) should be disabled initially for an execution; otherwise an execution cannot install handlers before asynchronous events begin arriving.

11.4.4 Multiple Pending Concurrent Exceptions

Since asynchronous events are not serviced immediately, there is the potential for multiple events to arrive between two polls for events. There are several options for dealing with these pending asynchronous events.

If asynchronous events are not queued, there can be only one pending event. New events must be discarded after the first one arrives, or overwritten as new ones arrive, or overwritten only by higher priority events. However, the risk of losing an asynchronous event makes a system less robust; hence queuing events is usually superior.

If asynchronous events are queued, there are multiple pending events and several options for servicing them. The order of arrival (FIFO) can be chosen to determine the service order for handling pending events. However, a strict FIFO delivery order may be unacceptable, e.g., an asynchronous event to stop an execution from continuing erroneous computation can be delayed for an extended period of time in a FIFO queue. A more flexible semantics for handling pending exceptions is user-defined priorities. However, Section 11.4.3 discusses how a priority scheme reduces extensibility, making it inappropriate in an environment emphasizing code reuse.

Therefore, FIFO order seems acceptable for its simplicity in understanding and low implementation cost. However, allowing a pending event whose delivery is disabled to prevent delivering other pending events seems undesirable. Hence, an event should be able to be delivered before earlier events if the earlier events are disabled. This out-of-order delivery has important implications on the programming model of asynchronous exceptions. A programmer must be aware of the fact that two exceptions having the same source and faulting execution may be delivered out-of-order (when the first is disabled but not the second). This approach may seem unreasonable, especially when causal ordering is proved to be beneficial in distributed programming. However, out-of-order delivery is necessary for urgent events. Currently, the most adequate delivery scheme remains as an open problem, and the answer may only come with more experience.

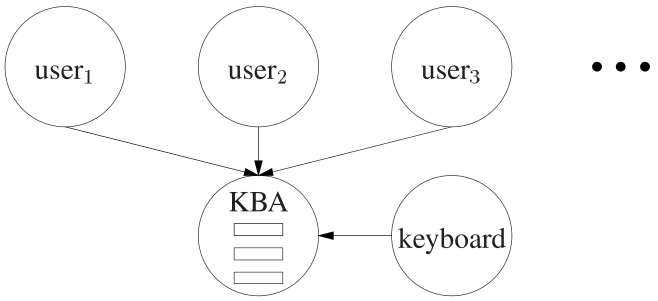
11.4.5 Converting Interrupts to Exceptions

As mentioned, hardware interrupts can occur at any time, which significantly complicates the nonreentrant problem. One technique that mitigates the problem is to convert interrupts into language-level asynchronous events, which are then controlled by the runtime system. Some interrupts target the whole program, like abort execution, while some target individual executions that compose a program, like completion of a specific thread's I/O operation. Each interrupt handler raises an appropriate asynchronous exception to the particular faulting execution or to some system execution for program faults. However, interrupts must still be disabled when enqueueing and dequeuing the asynchronous events to avoid the possibility of corrupting the queue by another interrupt or the execution processing the asynchronous events. By delivering interrupts through the EHM, the nonreentrant problem is avoided and interrupts are disabled for the minimal time. Furthermore, interrupts do not usually have all the capabilities of an EHM, such as parameters; hence, interrupts are not a substitute for a general EHM. Finally, the conversion also simplifies the interface within the language. The interrupts can be completely hidden within the EHM, and programmers only need to handle abnormal conditions at the language level, which improves portability across systems. However, for critical interrupts and in hard real-time systems, it may still be necessary to have some control over interrupts if they require immediate service, i.e., software polling is inadequate.

One final point about programming interrupt handlers is that raising a synchronous exception within an interrupt handler is meaningful only if it does not propagate outside of the handler. The reason is that the handler executes on an arbitrary execution stack, and hence, there is usually no relationship between the interrupt handler and the execution. Indeed, Ada 95 specifies that propagating an event from an interrupt handler has no effect.

11.5 Questions

1. Write the following program, which has user tasks, a keyboard administrator (KBA) task, and a keyboard task, organized in the following way:



The keyboard task has the following interface:

```
_Task Keyboard {
  public:
    Keyboard( KBA &kba );
};
```

It reads characters from standard input and sends the individual character to the KBA. The whitespace newline character, `'\n'`, is never sent to the KBA from the keyboard task; all other characters are sent, including the other whitespace characters. When the keyboard task encounters end-of-file, it returns the character `'\377'` to the KBA and terminates. Note, characters typed at the keyboard are buffered until RETURN (or ENTER) is typed, and then the whole line is sent to your program. The need to type RETURN before your program sees the data is fine for this assignment.

A user task has the following interface:

```
_Task User {
  public:
    User( KBA &kba, int taskNo );
};
```

It makes requests to the KBA to get characters from the keyboard. If a user task gets the character `'-'`, it terminates. Otherwise, it prints its task number (see below) and the character it received. Also, a user task must print its task number when it starts and finishes.

The KBA task has the following interface:

```
_Task KBA {
  public:
    KBA();
    void nextChar( char ch );     // called by keyboard task
    char getChar();               // called by user task
};
```

It keeps a list of user tasks. Only one of the users in the list is considered to be *active*. When the KBA gets a character from the keyboard task, it gives the character to the active user. The KBA deals with the following command characters:

'-' which means terminate the active user and remove it from the user list. The next task to become active is the next one in the list. If the current active user is at the end of the list, the remove makes the new active user the one at the beginning of the list. The character `'-'` *is* passed to the user task.

'+' which means start a new user task, add it after the currently active task in the list, and make it the active user task. When a user task is created, a task number is passed to its constructor. The KBA keeps a counter, which starts at zero (i.e., the first task has name 0), and it is incremented by one for each user task created; the current value of the counter is the task name, and deleting tasks does not affect the next task name. This uniform naming makes it easier to check your output. The character `'+'` is *not* passed to any user task.

'>' which means switch to the next user in the list and make it the active user. If the current active user is at the end of the list, the switch makes the new active user the one at the beginning of the list. All subsequent characters are given to the new active user. The character `'>'` is *not* passed to any user task.

'@' which means broadcasts the next character to all users. Start with the current active user and cycle through all users in the same order as for the special character `'>'`. The character broadcast to user tasks is not interpreted by the KBA unless it is the `'-'` or `'\377'` character. For `'-'`, terminate each active user task in turn and remove it from the KBA user list. For `'\377'` terminate all users and the KBA. The character `'@'` is *not* passed to any user task unless it is preceded by an `'@'` character.

'&' sends the last non-command character again to the current user. The character `'&'` is *not* passed to the user task. If there is no last character for any reason, such as an initial input of `'+&'`, the `'&'` is ignored.

'\377' which means shut down all user tasks, whereupon the KBA terminates. The character `'\377'` is *not* passed to any user task.

To increase concurrency, the KBA must buffer the characters from the keyboard separately for each task and then process them in FIFO order for a particular task. *Therefore, a special character is dealt with after any normal characters before it in the buffer.* Otherwise, a user task could receive data that is not logically sent to it. Ensure that if the active user is running slowly, that the keyboard task can always send characters to the KBA (i.e., between calls to getChar, the keyboard can continue to make calls). As well, when the active user changes, the previously active user must be allowed to continue execution (i.e., not be blocked) until it requests the next character. Finally, if there is no active user (i.e., no user tasks), characters are ignored instead of buffered, except for ´+´ and ´\377´.

The main program should start the KBA and wait for its completion. The KBA should start the keyboard task. Then you can play with the program, creating user tasks, typing characters to each user task, switching between user tasks, and terminating the user tasks. Each user task must print when it starts, when it receives a character (and what the character is), and when it terminates. The keyboard task must print the same as a user task. The KBA task must print any ignored characters (including whitespace characters), any new active tasks, changes to the active task, a deleted task and the subsequent new active task, shut down and any tasks deleted during shut down. Print whitespace characters as two character strings so they are visible, i.e., use character strings like "\t" for tab and "\s" for space. The following is an example of the required behaviour, the exact output format is up to you. (All output spacing can be accomplish using tabs so it is unnecessary to build strings of output messages.)

```
INPUT   KEYBOARD        KBA                     USERS
=====   ========        ===                     =====
                        starting
        starting
a
        read:a
        read:\n
                        ignore char:a
+
        read:+
        read:\n
                        cmnd:+, new active user:0
                                                task:0 starting
```

```
&
        read:&
        read:\n
                No last character to repeat
b
        read:b
        read:\n
                        task:0 char:b
+
        read:+
        read:\n
                cmnd:+, new active user:1
                        task:1 starting
&
        read:&
        read:\n
                        task:1 char:b
–
        read:–
        read:\n
                cmnd:–, delete active user:1
                        task:1 char:–
                cmnd:–, next active user:0
                        task:1 terminating
+
        read:+
        read:\n
                cmnd:+, new active user:2
                        task:2 starting
+
        read:+
        read:\n
                cmnd:+, new active user:3
                        task:3 starting
&
        read:&
        read:\n
                        task:3 char:b
+
        read:+
        read:\n
                cmnd:+, new active user:4
                        task:4 starting
>
        read:>
        read:\n
                cmnd:>, next active user:0
c
        read:c
        read:\n
                        task:0 char:c
&
        read:&
        read:\n
                        task:0 char:c
d
        read:d
        read:\n
                        task:0 char:d
```

```
>
        read:>
        read:\n
                        cmnd:>, next active user:2
e
        read:e
        read:\n
                                task:2 char:e
>
        read:>
        read:\n
                        cmnd:>, next active user:3
f
        read:f
        read:\n
                                task:3 char:f
>
        read:>
        read:\n
                        cmnd:>, next active user:4
g
        read:g
        read:\n
                                task:4 char:g
>
        read:>
        read:\n
                        cmnd:>, next active user:0
@
        read:@
        read:\n
h
        read:h
        read:\n
                        cmnd:@h
                                task:0 char:h
                                task:2 char:h
                                task:3 char:h
                                task:4 char:h
–
        read:–
        read:\n
                        cmnd:–, delete active user:0
                                task:0 char:–
                        cmnd:–, next active user:2
                                task:0 terminating
–
        read:–
        read:\n
                        cmnd:–, delete active user:2
                                task:2 char:–
                        cmnd:–, next active user:3
                                task:2 terminating
>
        read:>
        read:\n
                        cmnd:>, next active user:4
```

```
space (not actually printed)
        read:\s
        read:\n
                                        task:4 char:\s
>
        read:>
        read:\n
                        cmnd:>, next active user:3
j
        read:j
        read:\n
                                        task:3 char:j
^D (not actually printed)
        terminating
                                        task:3 char:-
                                        task:3 terminating
                                        task:4 char:-
                                        task:4 terminating
                        terminating
```

The executable program is named kba and has the following shell interface:

```
kba
```

Chapter 12

Parallel Execution

12.1 μC++ Runtime Structure

The dynamic structure of an executing μC++ program is significantly more complex than a normal C++ program. In addition to the five kinds of objects introduced by the elementary properties, μC++ has two more runtime entities that are used to control concurrent execution.

12.1.1 Cluster

A **cluster** is a collection of tasks and virtual processors (discussed next) that execute the tasks. The purpose of a cluster is to control the amount of parallelism that is possible among tasks, where **parallelism** is defined as execution which occurs simultaneously. Parallelism can only occur when multiple processors are present. **Concurrency** is execution that, over a period of time, appears to be parallel. For example, a program written with multiple tasks has the potential to take advantage of parallelism but it can execute on a uniprocessor, where it may *appear* to execute in parallel because of the rapid speed of context switching.

Normally, a cluster uses a single-queue multi-server queueing model for scheduling its tasks on its processors (see Chapter **??**, p. ?? for other kinds of schedulers). This simple scheduling results in automatic load balancing of tasks on processors. Figure 12.1 illustrates the runtime structure of a μC++ program. An executing task is illustrated by its containment in a processor. Because of appropriate defaults for clusters, it is possible to begin writing μC++ programs after learning about coroutines or tasks. More complex concurrency work may require the use of clusters. If several clusters exist, both tasks and virtual processors, can be explicitly migrated from one cluster to another. No automatic load balancing among clusters is performed by μC++.

When a μC++ program begins execution, it creates two clusters: a system cluster and a user cluster. The system cluster contains a processor that does not execute user tasks. Instead, the system cluster handles system related operations, such as, catching errors that occur on the user clusters, printing appropriate error information, and shutting down μC++. A user cluster is created to contain the user tasks; the first task created in the user cluster is uMain, which begins executing the member routine uMain::main. Having all tasks execute on the one cluster often maximizes utilization of processors, which minimizes runtime. However, because of limitations of the underlying operating system or because of special hardware requirements, it is sometimes necessary to have more than one cluster. Partitioning into clusters must be used with care as it has the potential to inhibit parallelism when used indiscriminately. However, in some situations partitioning is essential, e.g., on some systems concurrent UNIX I/O operations are only possible by exploiting the clustering mechanism.

12.1.2 Virtual Processor

A μC++ virtual processor is a "software processor" that executes threads. A virtual processor is implemented by kernel thread (normally created through a UNIX process) that is subsequently scheduled for execution on a hardware processor by the underlying operating system. On a multiprocessor, kernel threads are usually distributed across the hardware processors and so some virtual processors are able to execute in parallel. μC++ uses virtual processors instead of hardware processors so that programs do not actually allocate and hold hardware processors. Programs can be written to run using a number of virtual processors and execute on a machine with a smaller number of hardware

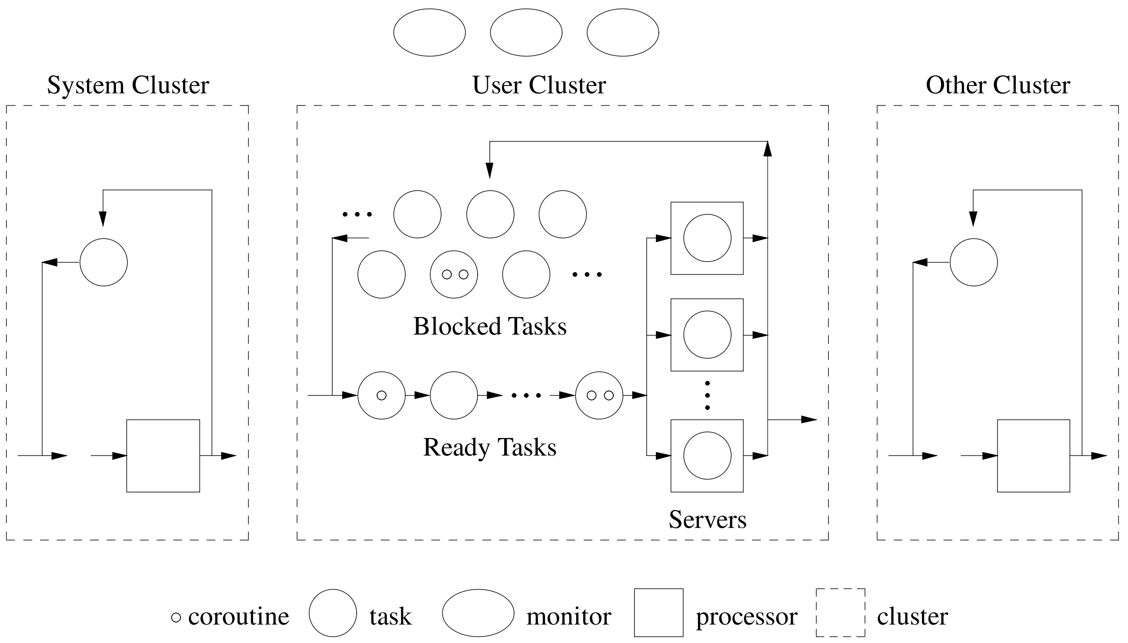


Figure 12.1: Runtime Structure of a μC++ Program

processors. Thus, the way in which μC++ accesses the parallelism of the underlying hardware is through an intermediate resource, the kernel thread. In this way, μC++ is kept portable across uniprocessor and different multiprocessor hardware designs.

When a virtual processor is executing, μC++ controls scheduling of tasks on it. Thus, when UNIX schedules a virtual processor for a runtime period, μC++ may further subdivide that period by executing one or more tasks. When multiple virtual processors are used to execute tasks, the μC++ scheduling may automatically distribute tasks among virtual processors, and thus, indirectly among hardware processors. In this way, parallel execution occurs.

12.2 Cluster

A cluster is a collection of μC++ tasks and processors; it provides a runtime environment for execution. This environment controls the amount of parallelism and contains variables to affect how coroutines and tasks behave on a cluster. Environment variables are used implicitly, unless overridden, when creating an execution state on a cluster:

stack size is the default stack size, in bytes, used when coroutines or tasks are created on a cluster.

The variable(s) is either explicitly set or implicitly assigned a μC++ default value when the cluster is created. A cluster is used in operations like task or processor creation to specify the cluster on which the task or processor is associated. After a cluster is created, it is the user's responsibility to associate at least one processor with the cluster so it can execute tasks.

The cluster interface is the following:

```
class uCluster {
  public:
    uCluster( unsigned int stacksize = uDefaultStackSize(), const char *name = "*unnamed*" );
    uCluster( const char *name );

    const char *setName( const char *name );
    const char *getName() const;
    unsigned int setStackSize( unsigned int stacksize );
    unsigned int getStackSize() const;
    const uBaseTaskSeq &getTasksOnCluster();
    const uProcessorSeq &getProcessorsOnCluster();
};

uCluster clus( 8196, "clus" )  // 8K default stack size, cluster name is "clus"
```

The overloaded constructor routine uCluster has the following forms:

uCluster(**unsigned int** stacksize = uDefaultStackSize(), **const char** *name = "*unnamed*") – this form uses the user specified stack size and cluster name.

uCluster(**const char** *name) – this form uses the user specified name for the cluster and the current cluster's default stack size.

When a cluster terminates, it must have no tasks executing on it and all processors associated with it must be freed. It is the user's responsibility to ensure no tasks are executing on a cluster when it terminates; therefore, a cluster can only be deallocated by a task on another cluster.

The member routine setName associates a name with a cluster and returns the previous name. The member routine getName returns the string name associated with a cluster.

The member routine setStackSize is used to set the default stack size value for the stack portion of each execution state allocated on a cluster and returns the previous default stack size. The new stack size is specified in bytes. For example, the call clus.setStackSize(8000) sets the default stack size to 8000 bytes.

The member routine getStackSize is used to read the value of the default stack size for a cluster. For example, the statement i = clus.getStackSize() sets i to the value 8000.

The member routine getTasksOnCluster returns a list of all the tasks currently on the cluster. The member routine getProcessorsOnCluster returns a list of all the processors currently on the cluster.

The routine:

```
uCluster &uThisCluster();
```

is used to determine the identity of the current cluster a task resides on.

12.3 Processors

A μC++ virtual processor is a "software processor" that executes threads. it provides a runtime environment for parallel thread execution. This environment contains variables to affect how thread execution is performed on a processor.

As stated previously, there are two versions of the μC++ kernel: the unikernel, which is designed to use a single processor; and the multikernel, which is designed to use several processors. The interfaces to the unikernel and multikernel are identical; the only difference is that the unikernel has only one virtual processor. In particular, in the unikernel, operations to increase or decrease the number of virtual processors are ignored. The uniform interface allows almost all concurrent applications to be designed and tested on the unikernel, and then run on the multikernel after re-linking.

The processor interface is the following:

```
class uProcessor {
  public:
    uProcessor( unsigned int ms = uDefaultPreemption(), unsigned int spin = uDefaultSpin() );
    uProcessor( bool detached, unsigned int ms = uDefaultPreemption(),
                unsigned int spin = uDefaultSpin() );
    uProcessor( uCluster &cluster, unsigned int ms = uDefaultPreemption(),
                unsigned int spin = uDefaultSpin() );
    uProcessor( uCluster &cluster, bool detached, unsigned int ms = uDefaultPreemption(),
                unsigned int spin = uDefaultSpin() );

    uClock &getClock() const;
    uPid_t getPid() const;
    uCluster &setCluster( uCluster &cluster );
    uCluster &getCluster() const;
    uBaseTask &getTask() const;
    bool getDetach() const;
    unsigned int setPreemption( unsigned int ms );
    unsigned int getPreemption() const;
    unsigned int setSpin( unsigned int spin );
    unsigned int getSpin() const;
    bool idle() const;
};
```

```
uProcessor proc( clus );  // processor is attached to cluster clus
```

A processor can be non-detached or detached with respect to its associated cluster. A non-detached processor is automatically/dynamically allocated and its storage is managed by the programmer. A detached processor is dynamically allocated and its storage is managed by its associated cluster, i.e., the processor is automatically deleted when its cluster is deleted.

The overloaded constructor routine uProcessor has the following forms:

uProcessor(**unsigned int** ms = uDefaultPreemption(), **unsigned int** spin = uDefaultSpin()) – creates a non-detached processor on the current cluster with the user specified time-slice and processor-spin duration (see Section **??**, p. ?? for the default values).

uProcessor(**bool** detached, **unsigned int** ms = uDefaultPreemption(), **unsigned int** spin = uDefaultSpin()) – creates a detached/non-detached processor on the current cluster with the user specified time-slice and processor-spin duration (see Section ??, p. ?? for the default values). The indicator for detachment is **false** for non-detached and **true** for detached.

uProcessor(uCluster &cluster, **unsigned int** ms = uDefaultPreemption(), **unsigned int** spin = uDefaultSpin()) – creates a non-detached processor on the specified cluster using the user specified time-slice and processor-spin duration.

uProcessor(uCluster &cluster, **bool** detached, **unsigned int** ms = uDefaultPreemption(), **unsigned int** spin = uDefaultSpin()) – creates a detached/non-detached processor on the specified cluster using the user specified time-slice and processor-spin duration. The indicator for detachment is **false** for non-detached and **true** for detached.

The member routine getClock) returns the clock used to control timing on this processor (see Section **??**, p. ??).

The member routine getPid returns the current UNIX process id that the processor is associated with.

The member routine setCluster moves a processor from its current cluster to another cluster and returns the current cluster. The member routine getCluster returns the current cluster the processor is associated with, and hence, executing tasks for.

The member routine getTask returns the current task that the processor is executing.

The member routine getDetach returns if the processor is non-detached (**false**) or detached (**true**).

The member routine setPreemption is used to set the default pre-emption duration for a processor (see Section 12.3.1) and returns the previous default pre-emption duration. The time duration between interrupts is specified in milliseconds. For example, the call proc.setPreemption(50) sets the default pre-emption time to 0.05 seconds for a processor. To turn pre-emption off, call proc.setPreemption(0). The member routine getPreemption is used to read the current default pre-emption time for a processor. For example, the statement i = proc.getPreemption() sets i to the value 50.

The member routine setSpin is used to set the default spin-duration for a processor (see Section 12.3.2) and returns the previous default spin-duration. The spin duration is specified as the number of times the cluster's ready queue is checked for an available task to execute before the processor blocks. For example, the call proc.setSpin(500) sets the default spin-duration to 500 checks for a processor. To turn spinning off, call proc.setSpin(0). The member routine getSpin is used to read the current default spin-duration for a processor. For example, the statement i = proc.getSpin() sets i to the value 500.

The member routine idle indicates if this processor is currently idle, i.e., the UNIX process has blocked because there were no tasks to execute on the cluster it is associated with.

The free routine:

```
uBaseProcessor &uThisProcessor();
```

is used to determine the identity of the current processor a task is executing on.

The following are points to consider when deciding how many processors to create for a cluster. First, there is no advantage in creating significantly more processors than the average number of simultaneously active tasks on the cluster. For example, if on average three tasks are eligible for simultaneous execution, creating significantly more than three processors does not achieve any execution speedup and wastes resources. Second, the processors of a cluster are really virtual processors for the hardware processors, and there is usually a performance penalty in creating more virtual processors than hardware processors. Having more virtual processors than hardware processors can result in extra context switching of the underlying kernel threads or operating system processes (see Section 12.3.3) used to implement a virtual processor, which is runtime expensive. This same problem can occur among clusters. If a computational problem is broken into multiple clusters and the total number of virtual processors exceeds the number of hardware processors, extra context switching occurs at the operating system level. Finally, a μC++ program usually shares the hardware processors with other user programs. Therefore, the overall operating system load affects how many processors should be allocated to avoid unnecessary context switching at the operating system level.

□ Changing the number of processors is expensive, since a request is made to the operating system to allocate or deallocate kernel threads or processes. This operation often takes at least an order of magnitude more time than task creation. Furthermore, there is often a small maximum number of kernel threads and/or processes (e.g., 20–40) that can be created in a program. Therefore, processors should be created judiciously, normally at the beginning of a program. □

12.3.1 Implicit Task Scheduling

Pre-emptive scheduling is enabled by default on both unikernel and multikernel. Each processor is periodically interrupted in order to schedule another task to be executed. Note that interrupts are not associated with a task but with a processor; hence, a task does not receive a time-slice and it may be interrupted immediately after starting execution because the processor's pre-emptive scheduling occurs and another task is scheduled. A task is pre-empted at a nondeterministic location in its execution when the processor's pre-emptive scheduling occurs. Processors on a cluster may have different pre-emption times. The default processor time-slice is machine dependent but is approximately 0.1 seconds on most machines. The effect of this pre-emptive scheduling is to simulate parallelism. This simulation is usually accurate enough to detect most situations on a uniprocessor where a program might be dependent on order or speed of execution of tasks.

□ On many systems the minimum pre-emption time may be 10 milliseconds (0.01 of a second). Setting the duration to an amount less than this simply sets the interrupt time interval to this minimum value. □

□ The overhead of pre-emptive scheduling depends on the frequency of the interrupts. Furthermore, because interrupts involve entering the UNIX kernel, they are relatively expensive if they occur frequently. An interrupt interval of 0.05 to 0.1 seconds gives adequate concurrency and increases execution cost by less than 1% for most programs. □

12.3.2 Idle Virtual Processors

When there are no ready tasks for a virtual processor to execute, the idle virtual processor has to spin in a loop or block or both. In the μC++ kernel, an idle virtual processor spins for a user-specified number of checks of the cluster's ready queue before it blocks. During the spinning, the virtual processor is constantly checking for ready tasks, which would be made ready by other virtual processors. An idle virtual processor is ultimately blocked so that machine resources are not wasted. The reason that the idle virtual processor spins is because the block/unblock time can be large in comparison to the execution of tasks in a particular application. If an idle virtual processor is blocked immediately upon finding no ready tasks, the next executable task has to wait for completion of an operating system call to restart the virtual processor. If the idle processor spins for a short period of time, any task that becomes ready during the spin duration is processed immediately. Selecting a spin amount is application dependent and it can have a significant effect on performance.

12.3.3 Blocking Virtual Processors

To ensure maximum parallelism, it is desirable that a task not execute an operation that causes the processor it is executing on to block. It is also essential that all processors in a cluster be interchangeable, since task execution may be performed by any of the processors of a cluster. When tasks or processors cannot satisfy these conditions, it is essential that they be grouped into appropriate clusters in order to avoid adversely affecting other tasks or guarantee correct execution. Each of these points is examined.

There are two forms of blocking that can occur:

heavy blocking which is done by the operating system on a virtual processor as a result of certain system requests (e.g., I/O operations).

light blocking which is done by the μC++ kernel on a task as a result of certain μC++ operations (e.g., **_Accept**, wait and calls to a mutex routine).

The problem with heavy blocking is that it removes a virtual processor from use until the operation is completed; for each virtual processor that blocks, the potential for parallelism decreases on that cluster. In those situations where maintaining a constant number of virtual processors for computation is desirable, tasks should block lightly rather than heavily, which is accomplished by keeping the number of tasks that block heavily to a minimum and relegated to a separate cluster. This can be accomplished in two ways. First, tasks that would otherwise block heavily instead make requests to a task on a separate cluster which then blocks heavily. Second, tasks migrate to the separate cluster and perform the operation that blocks heavily. This maintains a constant number of virtual processors for concurrent computation in a computational cluster, such as the user cluster.

On some multiprocessor computers not all hardware processors are equal. For example, not all of the hardware processors may have the same floating-point units; some units may be faster than others. Therefore, it may be necessary to create a cluster whose processors are attached to these specific hardware processors. (The mechanism for attaching virtual processors to hardware processors is operating system specific and not part of μC++. For example, the Dynix operating system from Sequent provides a routine tmp_affinity to lock a UNIX process on a processor.) All tasks that need to perform high-speed floating-point operations can be created/placed on this cluster. This segregation still allows tasks that do only fixed-point calculations to continue on another cluster, potentially increasing parallelism, but not interfering with the floating-point calculations.

12.4 Questions

1. xxx

Chapter 13

Control Flow Paradigms

This chapter compares and contrasts different approaches to specifying control flow and different implementations of the approaches in programming languages. Thus far, only μC++ has been used to specify advanced control flow; it is important to look at other approaches to see if there are strong or weak equivalences among them.

It will be shown that some systems are wasteful of time and/or space in an attempt to simplify writing complex control flow, particularly concurrency. While simplifying the job of programming is a laudable goal, it always comes at some cost. In general, simple problems can be solved with simple solutions and complex problems can *only* be solved with complex solutions. Always beware of simple solutions to complex problems because usually some aspect of the complexity is ignored in the solution, which requires one or more compromises, or the complexity is hidden away beyond a programmer's control. Be especially wary of implicit concurrency (see Section 5.4, p. 119), where compilers convert sequential programs into concurrent ones, e.g., by converting sequential looping into parallel looping by creating a thread to execute each loop iteration. Unfortunately, there are severe limitations on the capabilities of such systems. Many concurrent situations, like readers and writer, simply do not exist in a sequential form, and hence, cannot be expressed through such systems. In some cases the compromises in these systems are acceptable, but in many cases, they are not. Finally, many concurrent systems are unsound because the compiler is unaware that the program is concurrent. All concurrent library approaches have this problem because the compiler may perform valid sequential optimizations that invalidate a concurrent program.

13.1 Coroutines

While it is true that coroutines occur infrequently in programming, that does not imply a lack of usefulness. Nevertheless, only a few programming languages support coroutines [DMN70, MMS79, Wir88, MMPN93]. A language without coroutines results in awkward solutions for an important class of problems, e.g., automata, resulting in code that is difficult to generate, understand and maintain (see Chapter 4, p. 65).

The first language to provide a comprehensive coroutine facility was Simula [Sta87]. The Simula constructs are more general than those in μC++, but too much generality can be dangerous, e.g., like a **goto** versus a **while** loop. μC++ attempts to restrict the coroutine constructs, forcing a particular coding style and limiting control transfer, in an attempt to make using coroutines easier. Like structured programming, only certain control patterns are allowed to simplify the understanding of a program. The difficulty in introducing restrictions is to ensure the restricted forms easily and conveniently cover the common cases but do not preclude the esoteric ones, albeit with some additional work. In fact, one reason coroutines are not a standard programming mechanism is the complexity and problems resulting from earlier facilities that are too general. μC++ attempts to make coroutines an accessible programming facility.

Simula provides three constructs for coroutining: Call, Detach, and Resume. Call and Detach work like μC++ resume and suspend for building semi-coroutines, and Resume works like resume for building full coroutines.

In μC++, the kind of object is indicated by its type, e.g., coroutine or task, and it becomes this kind of object on creation and essentially remains that kind of object for its lifetime. (While a terminated coroutine is a class object and a terminated task is a monitor, these transformations occur only once and are not used in most programming situations.) In Simula, any object can become a coroutine depending on what is executed in its constructor, and convert back, possibly multiple times during its lifetime. The first Detach statement executed in an object's constructor causes the object to become a coroutine by changing it so it has its own execution state; control then suspends back to the declaration invoking the constructor, leaving the constructor suspended. Hence, in Simula, the constructor and the coroutine main are combined in the same block of code. Combining these two blocks of code requires a special facility called **inner** to allow inheritance to work correctly, otherwise the coroutine code for a base class is executed instead of the derived class. (See Section 13.2.2.1, p. 374 for details of **inner**.) To allow inheritance in μC++, the constructor and coroutine main are separate routines, and all constructors in a derivation chain are executed but only the coroutine main of the most derived class is started as the coroutine. The μC++ coroutine structure can be mimicked in Simula by placing the coroutine code in a routine called main, calling this routine at the end of the constructor, but detaching just before the call. (This simple simulation does not handle inheritance, which requires using **inner**.) Hence, control returns back to the declaration after the constructor code is executed and the first activation of the coroutine calls main. It is also possible to eliminate the Detach before the call to main, so control continues directly into main until the first Detach is encountered. This style can be mimicked in μC++ by putting a call to resume at the end of the constructor to start main (see Section 4.3, p. 71).

Semi-coroutining is performed using the Call and Detach pairing, where Call corresponds to a resume in a member routine and Detach corresponds to a suspend in the coroutine main. Simula Call explicitly specifies the target coroutine to be activated, as in Call(X). The current active coroutine becomes inactive and the target coroutine, i.e., X, becomes active. In μC++, resume makes the current coroutine inactive and the target coroutine is implicitly the **this** of the coroutine member containing the call to resume. In Simula, coroutine X can make its caller active by executing Detach, which is like suspend. However, Detach is subtlety different from suspend because it de-activates **this** and activates the caller versus de-activating the current coroutine and activating the caller. Therefore, a matching Detach for a Call must be executed from *within* the coroutine's code. In μC++, it is possible for a coroutine to suspend itself in another coroutine's member because the current coroutine does not changed on the call.

Notice the Call mechanism allows one coroutine to transfer control to another coroutine without calling one of its member routines. In μC++, each call to resume and suspend is located inside one of the member routines of a coroutine, which results in an encapsulation of all activations and de-activations within the coroutine itself, making the coroutine the only object capable of changing this property. As a result, it is only necessary to look at a coroutine's definition to understand all its activation and de-activation points, excluding external calls to other coroutine members performing suspend and resume. Furthermore, calling a member routine is consistent with normal object-oriented programming and provides a mechanism for passing argument information to the coroutine in a type-safe way. μC++ semi-coroutining can be mimicked in Simula by always calling a coroutine member and that member contains a Call(THIS *type-name*[1]) to activate the coroutine.

Full-coroutining is performed using Resume, where Resume corresponds to resume in a member routine. Simula Resume specifies the target coroutine to be activated, as in Resume(X). The current coroutine becomes inactive and the target coroutine, i.e., X, becomes active. Through this mechanism it is possible to construct arbitrarily complex resume-resume cycles among coroutines, exactly as in μC++. While Call appears to have the same capability as Resume, because both specify a target coroutine, Call cannot be used to construct cycles. The target coroutine of a Call is actually attached to the calling coroutine, i.e., turned back into an object, until it does a Detach, whereupon the target is turned back into a coroutine (like the initial Detach). It is illegal to Resume a coroutine that is in the attached state because it is not a coroutine.

Like Call, the Resume mechanism allows one coroutine to transfer control to another coroutine without calling one of its member routines, with the same problems. μC++ full coroutining can be mimicked in Simula by always calling a coroutine member and that member contains a Resume(THIS *type-name*) to activate the coroutine. Interestingly, the semantics of Detach change when a coroutine is resumed instead of called: Detach transfers back to the block sequence containing the object's lexical type rather than transferring back to the last caller, as in semi-coroutining. From a μC++ perspective, this semantics would normally cause the implicit coroutine associated with uMain to become active.

A Simula coroutine terminates when its constructor completes, at which point, an implicit Detach is executed. The behaviour of this Detach depends on whether the terminating coroutine was called or resumed. In μC++, a coroutine terminates when its coroutine main completes, at which point, the coroutine's starter is resumed (see Section 4.6, p. 80). Unlike μC++, the semantics of Simula Detach preclude transferring back to a terminated coroutine, because either the caller or the lexical block containing the definition must exist.

Figure 13.1 shows the Simula version of the semi-coroutine output formatter discussed in Section 4.2, p. 69, written in the μC++ coroutine style. Interestingly, using the μC++ coroutine style deals with another problem with Simula coroutines: the inability to pass and return information with Call/Resume/Detach. By embedding all coroutine

[1]Having to specify the type name for THIS is a peculiarity of Simula.

```
BEGIN
    CLASS FmtLines;
        HIDDEN ch, main;                    ! private members;
    BEGIN
        CHARACTER ch;                       ! communication;

        PROCEDURE main;                     ! mimic uC++ coroutine main;
        BEGIN
            INTEGER g, b;

            WHILE TRUE DO BEGIN             ! for as many characters;
                FOR g := 1 STEP 1 UNTIL 5 DO BEGIN ! groups of 5;
                    FOR b := 1 STEP 1 UNTIL 4 DO BEGIN ! blocks of 4;
                        OutChar( ch );
                        Detach;             ! suspend();
                    END;
                    OutText( "  " );
                END;
                OutImage;                   ! start new line
            END;
        END;

        PROCEDURE prt( chp );
            CHARACTER chp;
        BEGIN
            ch := chp;                      ! communication
            Call( THIS FmtLines );          ! resume();
        END;
        ! FmtLines constructor code;
        Detach;                             ! return to declaration;
        main;                               ! call main as last line of constructor;
    END FmtLines;
    ! uMain::main equivalent;
    REF(FmtLines) fmt;                      ! objects are references;
    INTEGER i;

    fmt :- NEW FmtLines;
    FOR i := Rank( ´ ´ ) STEP 1 UNTIL Rank( ´z´ ) DO BEGIN
        fmt.prt( Char( i ) );
    END
END;
```

Figure 13.1: Simula Semi-Coroutine: Output Formatter

transfers within object members and using communication variables, it is possible to pass and return information among Simula coroutines is a type-safe way. The coroutine formatter, FmtLines, is passed characters one at a time by calls to member prt and formats them into blocks of 4 characters and groups the blocks of characters into groups of 5. Notice the coroutine code is placed in a private member called main, and this routine is called at the end of the constructor, after the Detach. Also, the Call in member prt explicitly specifies THIS as the coroutine to be activated. In this specific example, the Call in member prt could be replaced by a Resume for two reasons. First, the instance of the coroutine, fmt is not programmatically terminated. The main block simply terminates and the coroutine is garbage collected at the end of the program. Second, if the coroutine was programmatically terminated, it would restart the main block because it was previously resumed (not called), and the type of the coroutine instance, FmtLines, is located in that block. In this simple program, it is coincidence that the implicit Detach on termination just happens to activate the same block regardless of whether the terminating coroutine is called or resumed; in general, this is not the case.

Figure 13.2, p. 363 shows the Simula version of the full coroutine producer/consumer discussed in Section 4.8.2, p. 91, written in the μC++ coroutine style. The program is almost identical to the μC++ version, but notice the use of Call instead of Resume in member routine stop for the last activation of the consumer by the producer. The Call ensures the final implicit Detach at the end of cons acts like a suspend to re-activate prod and not the main block, which is suspended at the Resume in member routine start. When prod terminates, it was last resumed, not called, by cons, and therefore, its final implicit Detach re-activates the main block in start. The Resume in start cannot be a Call because that would attach prod to the main block making it impossible for cons to resume prod, because prod would no longer be a coroutine.

Finally, Simula coroutines have further complexity associated with the ability to nest blocks in the language. However, this capability does not exist in most other object-oriented programming languages nor does it contribute to the notion of coroutines, and therefore, this capability is not discussed here.

13.1.1 Iterators

Some languages, e.g., Alphard [Sha81], CLU [LAB+81] and Sather [MOSS96] provide a limited form of coroutine solely for the purpose of constructing iterators (see Sections 4.5, p. 77 and **??**, p. ??). The iterator in CLU is presented as an example.

A CLU iterator is defined like a procedure, which may have parameters but must have a return value. In addition, an iterator has a yield statement as well as a **return**, which is equivalent to suspend but specifies a return value. A return value is always necessary because the iterator can only be used for setting the index of a **for** loop. For example, an iterator to traverse a linked list:

```
node = record[ val : int, next : node ]          % recursive data structure
list = record[ head, tail : node ]               % list top pointers
listIter = iter( l : list ) yields( n : node )   % iterator to traverse list
    n : node := list.head;
    while n ~= nil do
        yield( n )                               % suspend equivalent
        n := n.next
    end
end listIter
```

The iterator, listIter, accepts a list to be traversed and yields (returns) the nodes from the list one at a time. The yield statement in the **while** loop, suspends execution and returns the current node. Subsequent calls to the iterator restart execution after the yield.

The CLU iterator can only be used in conjunction with the **for** statement, as in:

```
l : list
sum : int := 0
for n : node in listIter( l ) do
    sum := sum + n.val
end
```

Here, the index variable n successively points at each node of the list l. Because an iterator is restricted for use with looping control structures, like **for**, it is not as general as a coroutine.

13.2 Threads and Locks Library

As the name implies, a threads and locks library is a set of routines providing the ability to create a new thread of control, and a few simple locks to control synchronization and mutual exclusion among threads. As is shown in Chapter 7, p. 177, working at this level is too detailed and low level for solving even medium-complex concurrent-application. Furthermore, all concurrent libraries are either unsound or inefficient because the compiler has no knowledge that a program using the library is concurrent. Even given these significant problems, there are still many thread libraries being developed and used.

While thread libraries have been written for many languages, only ones associated with C and C++ are discussed here, as that is sufficient to present the fundamental ideas and approaches. Even by restricting to C/C++, there are still several thread/lock libraries for C [BMS94, **?**, **?**, **?**] and C++ [BLL88, WL96].

```
BEGIN
    CLASS Consumer( prod );
        REF(Producer) prod; ! constructor parameter;
        HIDDEN p1, p2, status, done, Main;
    BEGIN
        INTEGER p1, p2, status;
        BOOLEAN done;

        PROCEDURE main;
        BEGIN
            INTEGER money, receipt;

            money := 1;
            WHILE NOT done DO BEGIN
                OutText( "cons receives: " );
                OutInt( p1, 3 );
                OutText( ", " );
                OutInt( p2, 3 );
                status := status + 1;
                OutText( " and pays $" );
                OutInt( money, 3 ); OutImage;
                receipt := prod.payment( money );
                OutText( "cons receipt #" );
                OutInt( receipt, 3 ); OutImage;
                money := money + 1;
            END;
            OutText( "cons stops" ); OutImage;
        END;

        INTEGER PROCEDURE delivery( p1p, p2p );
            INTEGER p1p, p2p;
        BEGIN
            p1 := p1p;
            p2 := p2p;
            Resume( THIS Consumer );
            delivery := status;
        END;

        PROCEDURE stop;
        BEGIN
            done := TRUE;
            Call( THIS Consumer );
        END;
        ! Consumer constructor code;
        status := 0;
        done := FALSE;
        Detach;
        main;
    END Consumer;

    CLASS Producer;
        HIDDEN cons, N, money, receipt, Main;
    BEGIN
        REF(Consumer) cons;
        INTEGER N, money, receipt;

        PROCEDURE main;
        BEGIN
            INTEGER i, p1, p2, status;

            FOR i := 1 STEP 1 UNTIL N DO BEGIN
                p1 := RandInt( 1, 100, p1 );
                p2 := RandInt( 1, 100, p2 );
                OutText( "prod delivers: " );
                OutInt( p1, 3 ); OutText( ", " );
                OutInt( p2, 3 ); OutImage;
                status := cons.delivery( p1, p2 );
                OutText( "prod status: " );
                OutInt( status, 3 ); OutImage;
             END;
            cons.stop;
            OutText( "prod stops" ); OutImage;
        END;

        INTEGER PROCEDURE payment( moneyp );
            INTEGER moneyp;
        BEGIN
            money := moneyp;
            OutText( "prod payment of $" );
            OutInt( money, 3 ); OutImage;
            Resume( THIS Producer );
            receipt := receipt + 1;
            payment := receipt;
        END;

        PROCEDURE start( Np, consp );
            INTEGER Np;
            REF(Consumer) consp;
        BEGIN
            N := Np;
            cons :- consp;
            Resume( THIS Producer );
        END;
        ! Producer constructor code;
        receipt := 0;
        Detach;
        main;
    END Producer;
    ! uMain::main equivalent;
    REF(Producer) prod;
    REF(Consumer) cons;
    prod :- NEW Producer;
    cons :- NEW Consumer( prod );
    prod.start( 5, cons );
END;
```

Figure 13.2: Simula Full-Coroutine: Producer/Consumer

13.2.1 C Thread Library: PThreads

One of the currently popular thread libraries is the POSIX PThreads [But97]. Because PThreads is designed for C, all of the PThreads routines are written in a non-object-oriented style, i.e., each routine explicitly receives the object being manipulated as one of its parameters. All non-object oriented library approaches are based on the START/WAIT model discussed in Section 5.6, p. 121. Hence, routines are used both to create a new thread of control and as the location for the new thread to start execution. In contrast, μC++ uses a declaration of a task type to start a new thread, and the thread implicitly starts execution in the task's main member routine.

13.2.1.1 Thread Creation and Termination

The routine used to start a new thread in PThreads is:

```
int pthread_create( pthread_t *new_thread_ID, const pthread_attr_t *attr,
    void * (*start_func)(void *), void *arg );
```

The type of a thread identifier is pthread_t, analogous to uBaseTask in μC++. However, a thread identifier is not a pointer to a thread because there is no thread object, or if there is one, it is an internal structure opaque to programmers. In μC++, the opaque thread object is hidden inside a task object, through implicit inheritance from uBaseTask, and the address of the task object is used to indirectly identify a thread.

The thread creation routine returns two results: one through the routine return value and the other through the first argument. The return value is a return code, versus an exception, indicating if the thread is successfully created. If the thread is successfully created, the first argument is assigned the thread identifier of the newly created thread. The second argument is a pointer to a structure containing attribute information for controlling creation properties for the new thread. The attribute information controls properties such as interaction with the operating-system kernel threads, termination properties, stack size, and scheduling policy. If the default attributes are sufficient, the argument can be set to NULL. Similar properties can be specified in μC++ through arguments to uBaseTask and via cluster schedulers. The last two arguments are a pointer to the routine where the thread starts execution and a pointer to a single argument for the starting routine. In effect, the new thread begins execution with a routine call of the form:

```
start_func( arg );
```

If more than one argument needs to be passed to start_func, the arguments must be packed into a structure, and the address of that structure used as the arg value (see message passing in Section 13.3, p. 379).

The following example:

```
void *rtn( void *arg ) { . . . }
int i = 3, rc;
pthread_t t;                              // thread id
rc = pthread_create( &t, NULL, rtn, &i ); // create new thread
if ( rc != 0 ) . . .                      // check for error
```

starts a new thread, with default attributes, beginning execution at routine rtn and passes a pointer to i as the argument of rtn. pthread_create assigns the new thread identifier into the reference argument t only if the creation is successful, and returns a return code indicating success or failure of the thread creation.

Notice the type of parameter arg for rtn is **void** * to allow a pointer of any C type to be passed from the caller, through pthread_create, and assigned to arg in a starting routine. As a result, the original type of the argument is lost when the pointer arrives in the start routine. Therefore, it is the start routine's responsibility to cast the argument pointer back to the original type. As well, since all arguments are passed by reference, the start routine may need to copy the argument if it is shared. Therefore, the first line of a typical start routines looks like:

```
void *rtn( void *arg ) {
    int a = *(int *)arg;    // cast and copy argument
    . . .
```

For all C library approaches, passing arguments to the start routine is type unsafe, as the compiler cannot ensure correct data is received by the start routine. For example, a programmer can pass an argument pointer of any type to rtn, through pthread_create, and rtn then dereferences it as an integer. Violating the type system in this way defeats the purpose of using a high-level language and relies on programmers following strict coding conventions without making errors.

A thread terminates when its starting routine returns or the thread calls pthread_exit, which is like uExit in μC++. Termination synchronization (see Section 5.6.1, p. 122) is possible using routine:

```
int pthread_join( pthread_t target_thread, void **status );
```

The thread join routine returns two results: one through the routine return value and the other through the second argument. The return value is a return code, versus an exception, indicating if the thread is successfully joined. If the thread is successfully joined, the second argument is assigned a pointer to a return value from the terminating thread. The first argument is the thread identifier of the thread with which termination synchronization is performed. The calling thread blocks until the specified thread, target_thread, has terminated; if the specified thread has already terminated, the calling thread does not block.

It is possible to pass a result back from the terminating thread, either through the starting routine's return-value or through routine pthread_exit. The value is received from the terminating thread through the reference parameter status, which is a pointer to an argument of unknown type. If more than one return value needs to be passed to the caller, the results must be packed into a structure, and the address of that structure passed back to the caller. If the calling thread is uninterested in the return value, the argument can be set to NULL.

The following example:

```
void *rtn( void *arg ) {
    . . .
    int *ip = malloc( sizeof(int) );        // allocate return storage
    *ip = 3;                                 // assign return value
    return ip;                               // return pointer to return storage
}
int i, *ip, rc;
rc = pthread_join( t, &ip );                 // wait for result
if ( rc != 0 ) . . .                         // check for error
i = *ip;                                     // copy return data
free( ip );                                  // free storage for returned value
```

has the calling thread wait for thread t to terminate and return a pointer to an integer return value. pthread_join returns a return code indicating success or failure of the join, and if successful, assigns the return pointer into the reference argument ip.

Notice the awkward mechanism needed to return a value because the storage for the terminated thread no longer exists after it finishes. Therefore, the terminating thread must allocate storage for the return value and the caller must subsequently delete this storage. Some of this complexity can be eliminated by using global shared variables or passing a return storage area as an argument to the start routine, so new storage does not have to be allocated for the return value. Also, the type of parameter status and the return type from the starting routine are **void** * to allow a pointer of any C type to be passed from the starting routine, through pthread_join, and assigned to status in a call to pthread_join.

For all C library approaches, passing return values from the start routine is type unsafe, as the compiler cannot ensure correct data is received by the caller of pthread_join. For example, a programmer can return a pointer of any type through pthread_join and the caller then dereferences it as an integer. Again, violating the type system in this way defeats the purpose of using a high-level language and relies on programmers following strict coding conventions without making errors.

In μC++, the starting routine, i.e., the task main, has no parameters and returns no result. Information is transfered to and from the task main, in a type safe manner, using the task member routines, albeit via copying information through global task variables. Some of this additional copying also occurs with PThreads because the argument is a pointer that may need its target copied on entry to the starting routine, and the result is a pointer that may need its target copied on termination. In general, the cost of any additional copying is significantly offset by the additional type safety. The μC++ approach to safe communication cannot be directly mimicked in PThreads because it requires multiple entry points. A routine has only one entry point, which is the starting point at the top of the routine. A class provides multiple entry points that can be invoked during the lifetime of its objects, and each call to an entry point is statically type checked.

Finally, it is possible for a thread to determine its identity through routine:

```
pthread_t pthread_self( void );
```

which is equivalent to uThisTask. As mentioned, the thread identifier returned is not a pointer but an opaque value. For example, a thread identifier could be a subscript into an array of pointers to pthread_t objects for a particular implementation of the thread library. The only operations allowed on an opaque thread-identifier is to pass it into and out of PThreads routines and to compare thread identifiers using the routine:

```
int pthread_equal( pthread_t t1, pthread_t t2 );
```

Figure 13.3 illustrates the PThreads equivalent for starting and terminating threads in a bounded buffer example (see the right hand side of Figure 7.11, p. 203). The bounded buffer is presented shortly.

13.2.1.2 Thread Synchronization and Mutual Exclusion

PThreads provides two kinds of locks, mutex and condition, through which all synchronization and mutual exclusion is constructed. The mutex lock is like a binary semaphore with the additional notion of ownership but no recursive acquisition[2], so the thread that locks the mutex must be the one that unlocks it, but only once. As a result, a mutex lock can *only* be used for mutual exclusion, because synchronization requires the locking thread be different from the unlocking one. The PThreads mutex lock has the following interface:

```
pthread_mutex_t mutex;
int pthread_mutex_init( pthread_mutex_t *mp, const pthread_mutexattr_t *attr );
int pthread_mutex_destroy( pthread_mutex_t *mp );
int pthread_mutex_lock( pthread_mutex_t *mp );
int pthread_mutex_trylock( pthread_mutex_t *mp );
int pthread_mutex_unlock( pthread_mutex_t *mp );
```

The type of a mutex lock is pthread_mutex_t. Each mutex has to be initialized and subsequently destroyed. Since there are no constructors or destructors in C, this step must be done manually through routines pthread_mutex_init and pthread_mutex_destroy, respectively. It is a common mistake to miss calling one of these routines for a mutex lock.

The routine pthread_mutex_init is used to initialize a mutex. A mutex is always initialized to unlocked (open) with no owner. The first argument is a pointer to the mutex to be initialized; a mutex can be reinitialized only after it is destroyed. The second argument is a pointer to a structure containing attribute information for specifying properties of the new mutex. The attribute information controls operating system related properties. If the default attributes are sufficient, the argument pointer can be set to NULL.

The routine pthread_mutex_destroy is used to destroy a mutex. Destruction must occur when the mutex is unlocked and no longer in use, which means no thread is currently waiting on or attempting to acquire/release the mutex; otherwise behaviour is undefined (i.e., PThreads may or may not generate an error for this case).

The routines pthread_mutex_lock and pthread_mutex_unlock are used to atomically acquire and release the lock, closing and opening it, respectively. pthread_mutex_lock acquires the lock if it is open, otherwise the calling thread spins/blocks until it can acquire the lock; whether a thread spins or blocks is implementation dependent. The member routine pthread_mutex_trylock makes one attempt to try to acquire the lock, i.e., it does not block. pthread_mutex_trylock returns 0 if the lock is acquired and non-zero otherwise. pthread_mutex_unlock releases the lock, and if there are waiting threads, the scheduling policy and/or thread priorities determine which waiting thread next acquires the lock.

The condition lock is like a condition variable, creating a queue object on which threads block and unblock; however, there is no monitor construct to simplify and ensure correct usage of condition locks. Instead, a condition lock is dependent on the mutex lock for its functionality, and collectively these two kinds of locks can be used to build a monitor, providing both synchronization and mutual exclusion. As for a condition variable, a condition lock can *only* be used for synchronization, because the wait operation always blocks. A condition lock has the following interface:

```
pthread_cond_t cond;
int pthread_cond_init( pthread_cond_t *cp, const pthread_condattr_t *attr );
int pthread_cond_destroy( pthread_cond_t *cp );
int pthread_cond_wait( pthread_cond_t *cp, pthread_mutex_t *mutex );
int pthread_cond_signal( pthread_cond_t *cp );
int pthread_cond_broadcast( pthread_cond_t *cp );
```

The type of a condition lock is pthread_condition_t. Each condition has to be initialized and subsequently destroyed. Since there are no constructors or destructors in C, this step must be done manually through routines pthread_cond_init and pthread_cond_destroy, respectively. It is a common mistake to miss calling one of these routines for a condition lock.

[2] recursive behaviour is available but only as an implementation-dependent option

```
void *producer( void *arg ) {
    BoundedBuffer *buf = (BoundedBuffer *)arg;
    const int NoOfElems = rand() % 20;
    int elem, i;

    for ( i = 1; i <= NoOfElems; i += 1 ) {           // produce elements
        elem = rand() % 100 + 1;                      // produce a random number
        printf( "Producer:0x%p, value:%d\n", pthread_self(), elem );
        insert( buf, elem );                          // insert element into queue
    }
}
void *consumer( void *arg ) {
    BoundedBuffer *buf = (BoundedBuffer *)arg;
    int elem;

    for ( ;; ) {                                      // consume until a negative element
        elem = remove( buf );                         // remove from front of queue
        printf( "Consumer:0x%p, value:%d\n", pthread_self(), elem );
      if ( elem == -1 ) break;
    }
}
int main() {
    const int NoOfCons = 3, NoOfProds = 4;
    BoundedBuffer buf;                                // create a buffer monitor
    pthread_t cons[NoOfCons];                         // pointer to an array of consumers
    pthread_t prods[NoOfProds];                       // pointer to an array of producers
    int i;

    ctor( &buf );                                     // initialize buffer

    for ( i = 0; i < NoOfCons; i += 1 ) {             // create consumers
        if ( pthread_create( &cons[i], NULL, consumer, &buf ) != 0 ) exit( -1 );
    }
    for ( i = 0; i < NoOfProds; i += 1 ) {            // create producers
        if ( pthread_create( &prods[i], NULL, producer, &buf ) != 0 ) exit( -1 );
    }
    for ( i = 0; i < NoOfProds; i += 1 ) {            // wait for producer to terminate
        if ( pthread_join( prods[i], NULL ) != 0 ) exit( -1 );
    }
    for ( i = 0; i < NoOfCons; i += 1 ) {             // terminate each consumer
        insert( &buf, -1 );
    }
    for ( i = 0; i < NoOfCons; i += 1 ) {             // wait for consumers to terminate
        if ( pthread_join( cons[i], NULL ) != 0 ) exit( -1 );
    }
    dtor( &buf );                                     // destroy buffer
}
```

Figure 13.3: PThreads: Producer/Consumer Creation/Termination

The routine pthread_cond_init is used to initialize a condition. The first argument is a pointer to the condition to be initialized; a condition can be reinitialized only after it is destroyed. The second argument is a pointer to a structure containing attribute information for specifying properties of the new condition. The attribute information controls operating system related properties. If the default attributes are sufficient, the argument pointer can be set to NULL.

The routine pthread_cond_destroy is used to destroy a condition. Destruction must occur when the condition is no longer in use, which means no thread is currently waiting on or attempting to signal/wait on the condition; otherwise behaviour is undefined (i.e., PThreads may or may not generate an error for this case).

The routines pthread_cond_wait and pthread_cond_signal are used to block a thread on and unblock a thread from the queue of a condition, respectively. However, operations on any queue must be done atomically, so both operations must be protected with a lock; hence, the need for the companion mutex lock to provide mutual exclusion. For example, given the following mutex and condition:

```
pthread_mutex_t m;
pthread_cond_t c;
pthread_mutex_init( &m, NULL );
pthread_cond_init( &c, NULL );
```

it is straightforward to protect the signal operation on a condition lock by:

```
pthread_mutex_lock( &m );
pthread_cond_signal( &c );      // protected operation
pthread_mutex_unlock( &m );
```

Hence, only one thread at a time is removing and unblocking waiting threads from the queue of the condition. Interestingly, PThreads allows a condition variable to be signalled by a thread whether or not it currently owns the mutex associated with the wait on the condition; hence, implicit locking must occur within the condition variable itself to ensure an atomic removal and unblocking of a waiting task. Nevertheless, PThreads advises the mutex lock must held during signalling if predictable scheduling behaviour is desired.

However, the wait operation must be performed with mutual exclusion but accomplishing this is problematic, as in:

```
pthread_mutex_lock( &m );
pthread_cond_wait( &c );        // protected operation
pthread_mutex_unlock( &m );     // never execute this statement
```

because after the mutex lock is acquired, the thread blocks and cannot unlocking the mutex. As a result, no other thread can acquire the mutex to access the condition lock; hence, there is a synchronization deadlock. Interchanging the last two lines clearly does not solve the problem, either. To solve this problem, the wait operation is passed the mutex and it atomically unlocks the mutex and blocks on the queue of the condition lock, as in:

```
pthread_mutex_lock( &m );
pthread_cond_wait( &c, &m );    // protected operation, and unlock mutex
```

This two-step operation by pthread_cond_wait is the same as that provided by the special version of P in Section 7.6.6, p. 215, which atomically Vs the argument semaphore, and then blocks the calling thread if necessary. Similarly, a monitor wait atomically blocks the monitor owner and releases the monitor lock.

pthread_cond_wait has the additional semantics of re-acquiring the argument mutex lock before returning, as if a call to pthread_mutex_lock had been made after unblocking from the condition lock. This semantics slightly reduces the complexity of constructing monitors but allows calling threads to barge into a monitor between a signaller leaving it and the signalled task restarting within it; hence, unless additional work is done, a monitor has the no-priority property (see Section 9.13.1, p. 296). pthread_cond_signal unblocks at least one waiting thread from the queue of a condition lock; it may unblock more than one. There is no compelling justification for this latter semantics and it only makes constructing a monitor more difficult. Since signal does not change the mutex lock, the signaller retains control of it (if acquired), so a monitor has the non-blocking property (see Section 9.11.3.4, p. 289). As always, a signal on an empty condition lock is lost, i.e., there is no counter to remember signals before waits. pthread_cond_broadcast unblocks all waiting thread from the queue of a condition lock, and is necessary in certain cases because there is no operation to check if the queue of a condition lock is empty.

Figure 13.4 illustrates the PThreads equivalent for the internal-scheduling bounded-buffer monitor in Figure 9.6, p. 262. The structure BoundedBuffer contains the shared monitor variables. The first field is the single mutex lock corresponding to the implicit monitor lock in Section 9.3, p. 256, which provides mutual exclusion among all mutex

```
typedef struct {
    pthread_mutex_t mutex;                              // monitor lock
    int front, back, count;
    int Elements[20];                                   // bounded buffer
    pthread_cond_t Full, Empty;                         // waiting consumers & producers
} BoundedBuffer;

void ctor( BoundedBuffer *buf ) {
    buf->front = buf->back = buf->count = 0;
    pthread_mutex_init( &buf->mutex, NULL );
    pthread_cond_init( &buf->Full, NULL );
    pthread_cond_init( &buf->Empty, NULL );
}
void dtor( BoundedBuffer *buf ) {
    pthread_mutex_lock( &buf->mutex );                  // cannot destroy if in use
    pthread_cond_destroy( &buf->Empty );
    pthread_cond_destroy( &buf->Full );
    pthread_mutex_unlock( &buf->mutex );                // release before destroy
    pthread_mutex_destroy( &buf->mutex );
}
int query( BoundedBuffer *buf ) {
    return buf->count;
}
void insert( BoundedBuffer *buf, int elem ) {
    pthread_mutex_lock( &buf->mutex );
    while ( buf->count == 20 )
        pthread_cond_wait( &buf->Empty, &buf->mutex ); // block producer

    buf->Elements[buf->back] = elem;                    // insert element into buffer
    buf->back = ( buf->back + 1 ) % 20;
    buf->count += 1;

    pthread_cond_signal( &buf->Full );                  // unblock consumer
    pthread_mutex_unlock( &buf->mutex );
}
int remove( BoundedBuffer *buf ) {
    int elem;
    pthread_mutex_lock( &buf->mutex );
    while ( buf->count == 0 )
        pthread_cond_wait( &buf->Full, &buf->mutex );  // block consumer

    elem = buf->Elements[buf->front];                   // remove element from buffer
    buf->front = ( buf->front + 1 ) % 20;
    buf->count -= 1;

    pthread_cond_signal( &buf->Empty );                 // unblock producer
    pthread_mutex_unlock( &buf->mutex );
    return elem;
}
```

Figure 13.4: PThreads: Bounded Buffer

members. The routines ctor and dtor perform the construction and destruction of the shared variables, by appropriately initializing and destroying the necessary variables. These routines must be called before/after each use of a buffer instance. Routine ctor *does not* acquire and release the monitor lock as the monitor must not be used until after it has been initialized. Routine dtor *does* acquire and release the monitor lock because the monitor cannot be destroyed if a thread is currently using it. (Actually, acquiring the mutex lock is optional if the programmer can ensure no thread is using the monitor when it is destroyed.)

Routines query, insert and remove are equivalent to those in the bounded buffer monitor. Routine query does not acquire and release the monitor lock because it is a non-mutex member; routines insert and remove do acquire and release the monitor lock because they are mutex members and the signals are non-blocking. The waits in insert and remove are no-priority; as a result, calling threads can barge ahead of threads signalled from internal condition locks, which means the state of the monitor at the time of the signal may have changed when the waiting thread subsequently restarts in the monitor. To deal with this problem, each wait is enclosed in a **while** loop to recheck if the event has occurred, which is a busy waiting because there is no bound on service (see Section 9.13.1, p. 296).

An additional problem exists when returning values from mutex routines. The values of global monitor variables cannot be returned directly; only local mutex routine variables can be returned directly. For a return, such as:

```
pthread_mutex_lock( &m );
return v;          // return global monitor variable
```

there is a race condition after unlocking the monitor lock and returning the monitor variable v. If a time-slice interrupt occurs between these two lines of code, another thread can enter a mutex routine and change the value of v before its previous value is returned to the interrupted thread. This problem is exacerbated by returning expression values involving global monitor variables, as in:

```
pthread_mutex_lock( &m );
return v1 + v2; // return expression involving global monitor variables
```

For example, the remove routine for the bounded buffer could return the element directly from the buffer, as in:

```
int remove( BoundedBuffer *buf ) {
    int prev;
    . . .
    prev = buf->front;                              // remember current position of front
    buf->front = ( buf->front + 1 ) % 20;
    buf->count -= 1;

    pthread_cond_signal( &buf->Empty );             // unblock producer
    pthread_mutex_unlock( &buf->mutex );
    return buf->Elements[prev];                     // return directly from buffer
}
```

Unfortunately, this fails because the contents of the buffer can change by the time the value is returned. The solution is to copy all return values involving global monitor variables into local mutex routine variables before releasing the monitor lock and returning, as in:

```
int temp = v1 + v2;          // make local copy
pthread_mutex_unlock( &m );
return temp;                 // return copy
```

It is the programmer's responsibility to detect these situations and deal with them appropriately.

In all these cases, there is no compiler support to ensure a programmer uses mutex and condition locks correctly or builds a monitor correctly. Common errors are to forget to acquire mutual exclusion before signalling or waiting and/or using the wrong mutex lock when signalling and waiting. In general, a programmer must manually perform the following conventions when building a monitor using PThreads:

1. each monitor must have a single mutex lock for the monitor lock
2. there must be the equivalent of a constructor and destructor to initialize/destroy the monitor lock and any condition variables
3. each mutex member, excluding any initialization members, must start by acquiring the monitor lock, and the monitor lock must be released by all return paths from a member
4. while PThreads allows signalling even if the signaller currently does not own the mutex logically associated with the condition variable during waits, such signalling can make scheduling behaviour unpredictable; hence,

it is good practise to only signal while holding the mutex logically associated with the condition variable
5. when waiting for an event, it is necessary to recheck for the event upon restarting because of barging
6. global monitor variables, versus local mutex routine variables, cannot be returned directly

Because a PThreads mutex lock can only be acquired once, a simulated mutex member cannot call another, unlike μC++; if it does, mutual-exclusion deadlock occurs (see possible workaround in Section 9.3.1, p. 257).

Some of these programming conventions can be simplified when using PThreads in C++, by using constructors and destructors. Figure 13.5 shows the previous PThreads bounded buffer written in PThreads/C++. First, the constructors and destructors for the BoundedBuffer class ensure the monitor lock and condition variables are correctly created and destroyed on object allocation and deallocation. Second, a special class MutexMem, which can be used for building any monitor, takes a reference to the monitor lock, and its constructor and destructor acquire and release the monitor lock. An instance of MutexMem, named lock, is then declared at the start of each monitor mutex member. (The name of the instance is unimportant.) Now the scope rules of C++ ensure the constructor of lock is executed before continuing after the declaration, which acquires the monitor lock. As well, the scope rules ensure the destructor of lock is called regardless of where or how the mutex member returns, including exit via an exception, so the monitor lock is guaranteed to be released. Furthermore, expressions in **return** statements are guaranteed to be evaluated and stored in an implicit temporary *before* local destructors are run because a destructor could change global monitor variables referenced in a return expression; hence, global monitor variables can be returned directly. While these C++ features mitigate some of the problems with building PThreads monitor, there is still ample opportunity for errors.

Finally, a solution to the readers and writer problem written using PThreads and C++ is presented in Figure 13.6, p. 373, which has no starvation and no staleness. The solution is problematic because of the signalling semantics of waking up more than one thread and the barging issue because priority is not given to signalled threads. The signalling problem is handled using a queue of private condition locks, like private semaphores in Section 7.6.6, p. 215, where a new condition is allocated for each blocking thread. Since there is only one thread blocked on each private condition, a signal can restart at most one thread. The barging issue is handled by not decrementing rwdelay by the signalling thread; instead, the signalled thread removes its node from the list and decrements rwdelay. As a result, a barging thread always sees a waiting thread and blocks, even if the last thread has been signalled from the condition lock. The only addition is a check at the start of StartWrite for rwdelay > 0, because a barging writer could see rcnt and wcnt at zero, but the last reader or writer may have signalled the only waiting thread from a condition lock. This additional check is unnecessary when there is no barging because control of the monitor goes directly to the signalled thread. This simple scheme for handling barging threads only works for problems where all threads wait on the same condition queue. If threads wait on different queues after entering the monitor, a more sophisticated technique must be used to deal with barging, involving creation of a special condition on which barging threads wait on entry; threads are then signalled from this special condition when there are no outstanding signals.

13.2.2 Object-Oriented Thread Library: C++

In the previous discussion, it is shown that C++ features, like constructors and destructor, can be used to simplify writing certain aspects of concurrent programs using a threads and locks library. However, the basic threads and locks library is not designed to fully utilize all the features of an object-oriented language to simplify implementation and/or usage, usually because the library is written in a non-object-oriented language, such as C. This section examines how a threads and locks library can be designed for an object-oriented language, in particular C++. Several different library approaches are examined because there is no library that is clearly the most popular, and there is no standard thread library as part of the C++ Standard Template Library (STL). As for non-object-oriented concurrency libraries, object-oriented libraries are either unsound or inefficient because the compiler has no knowledge that a program using the library is concurrent.

13.2.2.1 Thread Creation and Termination

In an object-oriented language, the natural way to provide concurrency through a library is to define an abstract class, Thread, that implements the thread abstraction:

```
class MutexMem {                                        // used by any monitor
    pthread_mutex_t &mutex;
  public:
    MutexMem( pthread_mutex_t &mutex ) : mutex( mutex ) {
        pthread_mutex_lock( &mutex );
    }
    ~MutexMem() {
        pthread_mutex_unlock( &mutex );
    }
};
template <class ELEMTYPE> class BoundedBuffer {
    pthread_mutex_t mutex;
    int front, back, count;
    ELEMTYPE Elements[20];
    pthread_cond_t Full, Empty;                         // waiting consumers & producers
  public:
    BoundedBuffer() {
        front = back = count = 0;
        pthread_mutex_init( &mutex, NULL );
        pthread_cond_init( &Full, NULL );
        pthread_cond_init( &Empty, NULL );
    }
    ~BoundedBuffer() {
        {   // cannot destroy if in use
            MutexMem lock( mutex );                     // acquire and release monitor lock
            pthread_cond_destroy( &Empty );
            pthread_cond_destroy( &Full );
        }   // release before destroy
        pthread_mutex_destroy( &mutex );
    }
    int query() { return count; }

    void insert( ELEMTYPE elem ) {
        MutexMem lock( mutex );                         // acquire and release monitor lock

        while ( count == 20 )
            pthread_cond_wait( &Empty, &mutex );        // block producer
        Elements[back] = elem;
        back = ( back + 1 ) % 20;
        count += 1;
        pthread_cond_signal( &Full );                   // unblock consumer
    }
    ELEMTYPE remove() {
        MutexMem lock( mutex );                         // acquire and release monitor lock

        while ( count == 0 )
            pthread_cond_wait( &Full, &mutex );         // block consumer
        ELEMTYPE elem = Elements[front];
        front = ( front + 1 ) % 20;
        count -= 1;
        pthread_cond_signal( &Empty );                  // unblock producer
        return elem;
    }
};
```

Figure 13.5: PThreads/C++: Bounded Buffer

```
class ReadersWriter {
    enum RW { READER, WRITER };                          // kinds of threads
    struct RWnode {
        RW kind;                                         // kind of thread
        pthread_cond_t cond;                             // private condition
        RWnode( RW kind ) : kind(kind) { pthread_cond_init( &cond, NULL ); }
        ~RWnode() { pthread_cond_destroy( &cond ); }
    };
    queue<RWnode *> rw;                                  // queue of RWnodes
    pthread_mutex_t mutex;
    int rcnt, wcnt, rwdelay;

    void StartRead() {
        MutexMem lock( mutex );
        if ( wcnt > 0 || rwdelay > 0 ) {
            RWnode r( READER );
            rw.push( &r ); rwdelay += 1;                 // remember kind of thread
            pthread_cond_wait( &r.cond, &mutex );
            rw.pop(); rwdelay -= 1;                      // remove waiting task from condition list
        }
        rcnt += 1;
        if ( rwdelay > 0 && rw.front()->kind == READER )
            pthread_cond_signal( &(rw.front()->cond) );
    }
    void EndRead() {
        MutexMem lock( mutex );
        rcnt -= 1;
        if ( rcnt == 0 && rwdelay > 0 )                  // last reader ?
            pthread_cond_signal( &(rw.front()->cond) );
    }
    void StartWrite() {
        MutexMem lock( mutex );
        if ( rcnt > 0 || wcnt > 0 || rwdelay > 0 ) {
            RWnode w( WRITER );
            rw.push( &w ); rwdelay += 1;                 // remember kind of thread
            pthread_cond_wait( &w.cond, &mutex );
            rw.pop(); rwdelay -= 1;                      // remove waiting task from condition list
        }
        wcnt += 1;
    }
    void EndWrite() {
        MutexMem lock( mutex );
        wcnt -= 1;
        if ( rwdelay > 0 )                               // anyone waiting ?
            pthread_cond_signal( &(rw.front()->cond) );
    }
  public:
    ReadersWriter() : rcnt(0), wcnt(0), rwdelay(0) {
        pthread_mutex_init( &mutex, NULL );
    }
    ~ReadersWriter() {
        {   // cannot destroy if in use
            MutexMem lock( mutex );
        }   // release before destroy
        pthread_mutex_destroy( &mutex );
    }
    . . .
};
```

Figure 13.6: PThreads/C++: Readers and Writer

```
class Thread {
    // any necessary locks and data structures
  public:
    Thread() {
        // create a thread and start it running in the most derived task body
    }
    // general routines for all tasks, like getState, yield, etc.
};
```

(Thread is like uBaseTask in Section 5.9.2, p. 128.) The constructor for Thread creates a thread to “animate” the object. A user-defined task class inherits from Thread, and a task is an object of this class:

```
class T1 : public Thread { . . .    // inherit from Thread
```

This approach has been used to define C++ libraries that provide coroutine facilities [Sho87, Lab90] and simple parallel facilities [DG87, BLL88].

When this approach is used, task classes should have the same properties as other classes, so inheritance from task types should be allowed. Similarly, task objects should have the same properties and behaviour as class objects. This latter requirement suggests that tasks should communicate via calls to member routines, since ordinary objects receive requests that way, and since the semantics of routine call matches the semantics of synchronous communication nicely. The body of the task, e.g., the task main, has the job of choosing which member-routine call should be executed next.

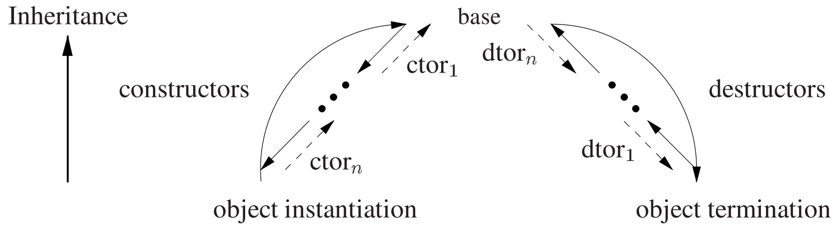
When one task creates another, the creating task’s thread executes statements in Thread’s constructor that create a new thread. The question arises which thread does what jobs in this process? The approach that produces the greatest concurrency has the new thread execute the new task’s constructors and body, while the creating thread returns immediately to the point of the declaration of the object. However, the normal implementation of constructors in most object-oriented languages makes this difficult or impossible if inheritance from task types is allowed. Each constructor starts by calling the constructors of its parent classes. By the time Thread’s constructor is called, there can be an arbitrary number of constructor activations on the stack, one for each level of inheritance. For a library approach, it is impossible for the initialization code for Thread to examine the stack to locate the return point for the original constructor (most derived) so the creating thread can return and the new thread can complete the initialization. Only compiler support, such as marking the stack at the point of declaration or passing implicitly the return address for the creating thread up the inheritance chain, can make this approach work. In the absence of compiler support, the creating thread must execute the new task’s constructors, after which the new thread can begin executing the task body, which inhibits concurrency somewhat.

Thread Body Placement The body of a task must have access to the members of a task, and the Thread constructor must be able to find the body in order to start the task’s thread running in it. Therefore, in the library approach, the task body must be a member of the task type, and there is hardly any other sensible choice. At first glance, the task’s constructor seems like a reasonable choice. However, the requirement that it be possible to inherit from a task type forbids this choice for most object-oriented languages. For example, let T1 be a task type, with a constructor that contains initialization code for private data and the task body. Now consider a second type T2 that inherits from T1:

```
class T1 : public Thread {        class T2 : public T1 {
  public:                           public:
    T1() {                            T2() {
        // initialization                 // initialization
        // task body                      // task body
    }                                 }
};                                };
```

T2’s constructor must specify a new task body if it overrides or adds new mutex members. Therefore, it must somehow override the task body in T1’s constructor, but still execute T1’s initialization code. Because both are contained in the same block of code, it is impossible. One solution is to put the body in a special member routine, like main, which can be declared by Thread as an abstract virtual member so that task types must supply one.

Simula provides an interesting mechanism, called **inner**, to address this issue. In a single inheritance hierarchy, an **inner** statement in a constructor (or destructor) of a base class acts like a call to the constructor (or destructor) of the derived class:

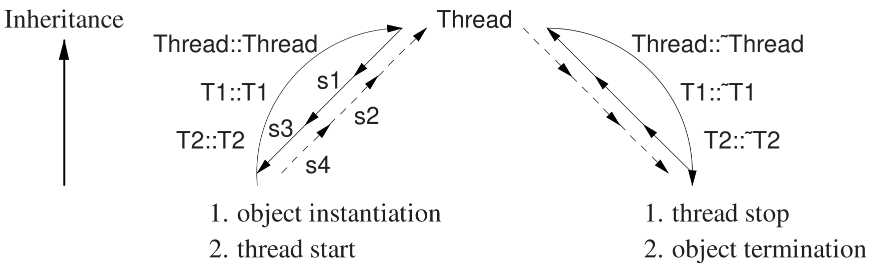


where the solid-lines are the normal execution path for constructors and destructors and the dashed-lines are the **inner** execution paths. For example, given:

```
class T1 : public Thread {
  public:
    T1() { s1; inner; s2; };
    mem() { ... }
};
T1 t1;
T2 t2;
```

```
class T2: public T1 {
  public:
    T2() : { s3; inner; s4; };
    mem() { ... }
};
```

the initialization of t1 executes statements Thread::Thread, s1, and s2, and the initialization of t2 executes Thread::Thread, s1, s3, s4, and s2, in that order.



In general, the code before **inner** is executed top-down in the inheritance structure (as for normal constructor initialization) and code after **inner** is executed bottom-up, so initialization is composed of two passes: down and then up the constructor chain. Similarly, for destructors, the code before **inner** is executed bottom-up in the inheritance structure (as for normal destructor initialization) and code after **inner** is executed top-down, so termination is composed of two passes: up and then down the destructor chain. During the initialization of t2, in T1::T1, before the **inner** statement, t2 is considered to be an instance of class T1. After the **inner** statement, because initialization is complete, it is an instance of T2, and the meaning of calls to virtual routines changes accordingly. In this case, a call to mem in s1 invokes T1::mem, while a call to mem in s2 invokes T2::mem because mem is overridden in T2.

How can **inner** be used to solve the previous problem? By following a convention of counting down one side of the initialization and then comparing the counter value on the up side. In this way, it is possible to programmatically locate the most derived constructor and only the task body of that constructor executes, as in:

```
class T1 : public Thread {
    int cnt;
  public:
    T1() {
        cnt = Thread::cnt += 1;
        // initialization
        inner;
        if ( cnt == Thread.cnt ) // task body
    };
};
```

```
class T2 : public T1 {
    int cnt;
  public:
    T2() {
        cnt = Thread::cnt += 1;
        // initialization
        inner;
        if ( cnt == Thread.cnt ) // task body
    };
};
```

Each constructor increments a global class counter in Thread and stores its position in the constructor chain in a local class counter during the execution down the constructor chain. On the way up the constructor chain, only the most derived constructor code is executed because of the conditional check.

However, it is not obvious how **inner** could be added to C++. **inner** must invoke constructors and possibly destructors in the order defined by C++, taking multiple inheritance and virtual inheritance into account. Furthermore, a class can have many constructors, and descendants specify which of their base class's constructors are called. Simula has only one constructor, no destructors, and single inheritance. Finally, this approach relies on programmers following this convention for it to work correctly, which is always a questionable requirement.

Thread Initialization and Execution There is a strict ordering between task initialization and task body execution. The task's thread must not begin to execute the task body until after the task's constructors have finished. However, in the library approach, the code to start the thread running in the task body appears in Thread's constructor, as in:

```
class Thread {
  public:
    Thread() { // create a thread and start it running in the most derived task body ...
    ~Thread() { // wait for thread to terminate ...
};
```

In C++, the thread creation code is executed first, *before* the constructors of any derived classes. Hence, the new thread must be created in the "blocked" state, and must somehow be unblocked after the most derived constructor finishes. A more subtle problem results from the semantics of initialization. While Thread's constructor is executing, the new task is considered to be an instance of class Thread, not the actual task class being instantiated. This means that, within Thread's constructor, the virtual main routine that contains the task's body is inaccessible.

The most common solution to these problems is requiring an explicit action to unblock the thread *after* all the constructors are executed. In this approach, the Thread class provides a start() member routine that must be called after the declaration of a task, but before any calls to member routines depending on task properties:

```
T1 t1;       // declare task object
t1.start()   // start thread running in most derived task body
```

After the declaration, the constructors have all finished and main refers to the most derived task body. However, this two-step creation protocol opens a window for errors: programmers may fail to start their tasks.

In object-oriented languages with destructors, like C++, a similar interaction exists between task-body execution and task termination. When one task deletes another, it calls the deleted task's destructor. The destructor must not begin execution until after the deleted task's body has finished:

```
{
    T1 t1; ...
} // call destructor for t1 and wait for task body to complete
```

otherwise the object's variables are terminated and storage is released while the thread is still using it. However, the code that waits for the task body to finish cannot be placed in Thread's destructor, because it would be executed last in the destructor sequence, *after* the destructors of any derived classes. Since the thread is still running in the most derived task body and referring to variables at that level, undefined behaviour would result. The task's termination code cannot simply be moved from the destructors to the end of the task body, because that would prevent further inheritance: derived classes would have no way to execute their base class's termination code. Thread could provide a join routine, analogous to start, which must be called before task deletion, but this two-step termination protocol is even more error-prone than the creation protocol, as forgetting the call to join deletes the task's storage with the thread still running.

If an **inner** mechanism is available, it could be used to solve this problem by not starting the thread until after the initialization, as in:

```
class Thread {
  public:
    Thread() {
        /* initialization */ inner; /* start thread in most derived task body */
    }
};
```

When the thread is started after **inner**, all the initialization is completed and a reference to main at this point is the most derived one. Thread's destructor would start by waiting for the task body to finish, and then use **inner** to execute the task class's destructors. As mentioned previously, this technique for creating the thread inhibits concurrency because the creator's thread is used to execute all the constructors, and similarly the destroying thread executes all the destructors.

More Inheritance Problems Regardless of whether a concurrency library or language extensions are used to provide concurrency in an object-oriented language, new kinds of types are introduced, like coroutine, monitor, and task. These new kinds of types complicate inheritance. The trivial case of single inheritance among homogeneous kinds, i.e., a monitor inheriting from another monitor, is straightforward because any implicit actions are the same throughout the hierarchy. (An additional requirement exists for tasks: there must be at least one task body specified in the hierarchy where the thread starts.) For a task or a monitor type, new member routines defined by a derived class can be accepted by statements in a new task body or in redefined virtual routines.

Inheritance among heterogeneous types can be both useful and confusing. Heterogeneous inheritance is useful for generating concurrent types from existing non-concurrent types, e.g., to define a mutex queue by deriving from a simple queue, or for use with container classes requiring additional link fields. For example, to change a simple queue to a mutex queue requires a monitor to inherit from the class Queue and redefine all of the class's member routines so mutual exclusion occurs when they are invoked, as in:

```
class Queue {                          // sequential queue
  public:
    void insert( ... ) ...
    virtual void remove( ... ) ...
};
class MutexQueue : public Queue {  // concurrent queue
    virtual void insert( ... ) {
        // provide mutual exclusion
        Queue::insert(...);            // call base-class member
    }
    virtual void remove( ... ) {
        // provide mutual exclusion
        Queue::remove(...);            // call base-class member
    }
};
```

However, this example demonstrates the dangers caused by non-virtual routines:

```
Queue *qp = new MutexQueue;  // subtyping allows assignment
qp->insert( ... );           // call to a non-virtual member routine, statically bound
qp->remove( ... );           // call to a virtual member routine, dynamically bound
```

Queue::insert does not provide mutual exclusion because it is a member of Queue, while MutexQueue::insert does provide mutual exclusion (along with MutexQueue::remove). Because the pointer variable qp is of type Queue, the call qp->insert calls Queue::insert even though insert is redefined in MutexQueue; hence, no mutual exclusion occurs. The unexpected lack of mutual exclusion results in errors. In contrast, the call to remove is dynamically bound, so the redefined routine in the monitor is invoked and appropriate synchronization occurs. In object-oriented programming languages that have only virtual member routines (i.e., dynamic dispatch is always used), this is not an issue. Nor is there a problem with C++'s private inheritance because no subtype relationship is created, and hence, the assignment to qp is invalid.

Heterogeneous inheritance among entities like monitors, coroutines and tasks can be very confusing. While some combinations are meaningful and useful, others are not, for the following reason. Classes are written as ordinary classes, coroutines, monitors, or tasks, and the coding styles used in each cannot be arbitrarily mixed. For example, an instance of a class that inherits from a task can be passed to a routine expecting an instance of the class. If the routine calls one of the object's member routines, it could inadvertently block the current thread indefinitely. While this could happen in general, there is a significantly greater chance if users casually combine types of different kinds. The safest rule to follow for heterogeneous inheritance is to require the derived kind to be equal or more specific than the base kind with respect to execution properties, e.g., a type with mutual exclusion can inherit from a type without mutual exclusion, but not vice versa.

Multiple inheritance simply exacerbates the problem and it significantly complicates the implementation, which slows the execution. For example, accepting member routines is significantly more complex with multiple inheritance because it is impossible to build a static mask to test on routine entry. As is being discovered, multiple inheritance is not as useful a mechanism as it initially seemed [BCK89, Car90].

13.2.2.2 Thread Synchronization and Mutual Exclusion

Synchronization As suggested in Section 13.2.2.1, p. 371, the obvious mechanism for communication among tasks is via calls to member routines, since this matches with normal object communication; however, not all object-oriented concurrency systems adopt this approach. In some library-based schemes (and some languages), communication is done via message queues, called ports or channels [Gal96], as in:

```
MsgQueue<int> Q1, Q2;           // global message queue of integers
class ThreadType : public Thread {
    void main() {               // task body
        Q1.add( 3 );            // add data to message queue
        int i = Q2.remove();    // remove data from message queue
    }
};
```

Particular message queues are shared among tasks that need to communicate, either through global message-queues or passing message-queue arguments. Depending on the kind of message queue, e.g., a prioritized queue, different delivery effects can be achieved. If tasks communicate using message queues, the meaning of any publicly available member routines needs to be addressed: in general, public members should be disallowed to prevent multiple threads accessing a task's data.

However, a single typed queue per task, such as a queue of integers, is inadequate; the queue's message type inevitably becomes a union of several message types, and static type-checking is compromised. Inheritance from an abstract Message class can be used, instead of a union, but then a task has to perform type tests on messages before accessing them with facilities like C++'s dynamic cast. However, runtime type-checking is still discouraged in C++ design philosophy and has a runtime cost.

When multiple queues are used, a library facility analogous to the μC++ **_Accept** statement is needed to wait for messages to arrive on more than one queue, as in:

```
waitfor i = Q1.remove() || f = Q2.remove ...
```

In some cases, it is essential to also know which message queue delivered the data, i.e., Q1 or Q2, as that knowledge might affect the interpretation of the data or the next action to be taken. This case is analogous to the code after an accept clause in a μC++ **_Accept** statement. One approach is for the waitfor statement to return the message queue identifier from which data was received. A programmer would then use a **switch** statement to discriminate among the message queue identifiers. A more general solution, requires λ-expressions (anonymous nested routine bodies) to support a block of code that may or may not be invoked depending on the selection criteria, e.g.:

```
waitfor i = Q1.remove(), {code-body} || f = Q2.remove, {code-body} ...
```

where the *code-body* is a λ-expression and represents the code executed after a particular message queue is accepted.

An implementation problem with message queues occurs when multiple tasks receive messages from the same set of message queues, e.g.:

```
MsgQueue<T1> Q1;
MsgQueue<T2> Q2;
class ThreadType : public Thread {
    void main() {               // task body
        waitfor i = Q1.remove(), {code-body} || f = Q2.remove, {code-body} ...
    }
};
ThreadType t1, t2;
```

Threads t1 and t2 simultaneously accept messages from the same queues, Q1 and Q2. It is straightforward to check for the existence of available data in each queue. However, if there is no data, both t1 and t2 must wait for data on either queue. To implement this, tasks have to be associated with both queues until data arrives, given data when it arrives, and then removed from both queues. This implementation is expensive since the addition or removal of a message from a queue must be atomic across all queues involved in a waiting task's accept statement to ensure only one data element from the accepted set of queues is given to the accepting task. In languages with concurrency support, the compiler can disallow accepting from overlapping sets of message queues by restricting the waitfor statement to queues the task declares. Compilers for more permissive languages, like SR [AOC+88], perform global analysis to determine if tasks are receiving from overlapping sets of message queues; in the cases where there is no overlap, less expensive code can be generated. In a library approach, access to the message queues must always assume the worst case scenario.

Alternatively, if the routine-call mechanism is used for communication among tasks (as in μC++), it is necessary to insert code at the start and exit of each mutex member to manage selective entry, and provide a facility analogous to the μC++ **_Accept** statement to allow a task to wait for calls to arrive. The code for selective entry must be provided by the programmer following a coding convention, as is done at the end of Section 13.2.1.2, p. 366 when using PThreads and C++. Building a library facility similar to a **_Accept** statement is difficult, e.g., ABC++ [OEPW96, p. 8] provides member routine, Paccept(*task-type*::*member-name*, ...) inherited from Thread, to control calls to mutex members. However, Paccept has many limitations, like a restricted number of member names, no mechanism to determine which call is accepted, efficiency problems, and type-safety issues.

Mutual Exclusion Providing mutual exclusion can be done in the same way as for controlling selective entry by a coding convention of explicitly locking at the start and unlocking at the exit points for each mutex member. Alternatively, the call to each mutex member is modified to perform the necessary mutual exclusion (which could also be done to provide selective entry). For example, in ABC++ [OEPW96], calling a mutex member is done indirectly by calling a special routine that then calls the mutex member, as in:

```
P_call( monitor_object, monitor_type::member, varible list of up to 7 arguments );
```

However, changing the call syntax in this way is usually unacceptable as it often restricts the routine call and makes programs difficult to read. Object-oriented programming languages supporting inheritance of routines, such as LOGLAN'88 [CKL+88] and Beta [MMPN93], can provide special member code automatically. (The use of **inner** in a constructor is a special case of routine inheritance, where the derived class's constructor inherits from the base class's constructor.) Whatever the mechanism, it must allow the special code to be selectively applied to the member routines. For example, there are cases where not all public member routines require mutual exclusion and where some private members require mutual exclusion. In languages with concurrency support, the compiler can efficiently handle all of these issues.

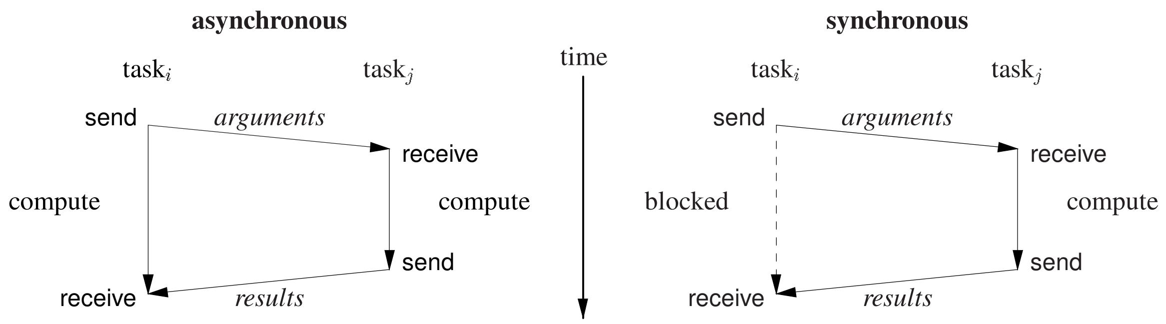
Finally, mutual exclusion can interact with synchronization, as in internal scheduling, via wait and possibly signal. The wait routine must be passed the monitor mutex-lock, either directly or indirectly through the **this** parameter if the lock is provided via inheritance, along with the condition variable, so it can perform any necessary baton passing to manage the mutex object correctly. However, in first case, users have to be aware of the mutex lock and could pass the wrong lock, and in the second case, having to pass **this** is obscure.

13.3 Threads and Message Passing

The next major style of library approach for concurrency is threads and message passing. Threads are usually created in the same style as in the previous section but synchronization and mutual exclusion are provided by a technique called message passing.

The term message passing has a number of connotation (see Section 5.3, p. 118). First, the term is often incorrectly used to imply distributed communication. While it is true that all distributed systems perform message passing, message passing can be performed on shared-memory systems. Second, the term is often used to imply passing data by value rather than by address. While it is true that most distributed systems only pass data by value because of the non-shared memories, addresses can be passed in messages for shared-memory systems. However, to ensure portability between distributed and shared-memory systems for programs that might execute on either, it is common practice to preclude passing pointers so that a program does not have to be aware of which environment it is executing on. Here, **message passing** has only one meaning: an alternative mechanism to parameter passing. That is, in message passing, all communication among threads is transmitted by grouping the data into a single argument and passing it by value. This semantics is a restricted form of parameter passing, where only one parameter is allowed for communication versus any number and type of parameters. If more than one value needs to be passed in a message, the value must be packed into a structure, and the structure passed in the message. While message passing appears to be both syntactically and semantically different from approaches seen thus far, it will be shown that there is a strong equivalence with prior approaches.

The basic mechanism for performing message passing is through two routines, send and receive, which transmit and receive a message, respectively. The resulting communication is often asynchronous, and synchronous communication can be simulated:



A third routine, reply, is sometimes available. All three mechanisms are discussed shortly. Messages are usually sent directly from thread to thread, so message passing is a form of direct communication. Each thread involved in message passing has an implicit buffer, analogous to the monitor entry queue, where messages are stored, like call arguments, and senders may wait for messages to be received (accepted).

All message passing approaches are implemented using a library approach, versus augmenting the programming language, and therefore, all concurrent message passing approaches are either unsound or inefficient because the compiler has no knowledge that a program using the library is concurrent. Even given these significant problems, there are still many message passing libraries being developed and used. **Discuss OCCAM**

13.3.1 Send

Send is the basic mechanism to communicate information to another thread and corresponds to a call of a task's mutex member. The minimum arguments to send are the thread identifier where the messages is being sent and the message, which can be empty, e.g., send(Tid,msg). The semantics of send vary depending on the message passing approach. Most message passing approaches support several different kinds of send, and a programmer selects one appropriate to the communication needs. The following is a list of different kinds of message sends.

13.3.1.1 Blocking Send

A **blocking send** implements synchronous communication, i.e., the sending thread blocks until at least the message has been received. Hence, information is transferred via a rendezvous between the sender and the receiver. For example, a producer thread would send messages to a consumer thread, Cid, by:

```
Producer() {
    for ( ;; ) {
        // produce element and place in msg
        SendBlock( Cid, msg );          // block until the message is received
    }
}
```

Blocking send corresponds to synchronous routine call, where the message corresponds to the argument list.

13.3.1.2 Nonblocking Send

A **nonblocking send** implements asynchronous communication, i.e., the sending thread blocks only until the message has been copied into a buffer associated with the receiver. The receiver normally gets the messages in the order they are placed into its message buffer. Because a receiver's message buffer can fill, a nonblocking send can temporarily become a blocking send until an empty buffer slot becomes available. Hence, a nonblocking send is only mostly nonblocking; a programmer cannot rely on true nonblocking semantics because that would require infinite buffer space. For example, a producer thread would send messages to a consumer thread, Cid, by:

```
Producer() {
    for ( ;; ) {
        // produce element and place in msg
        SendNonBlock( Cid, msg );       // do not block
    }
}
```

Nonblocking send corresponds to asynchronous routine call, where the message corresponds to the argument list.

13.3.1.3 Multicast/Broadcast Send

Both blocking and nonblocking send can have a multicast and broadcast form, which sends a message to a list of receivers or a special group of receivers or any receiver listening to a particular broadcast. For multicast, the group of receivers is usually defined using some thread grouping mechanism. For example, when tasks are created they may be placed into a named group and that group name can be used for a multicast send; tasks may enter and leave groups at any time during their lifetime. In essence, a multicast send is just a convenience mechanism to send the same message to a number of threads. For broadcast, there may be one or more broadcast channels on which messages are sent and received. When a channel is used for communication, the message passing becomes indirect because there is no specific target thread for the message, i.e., any channel receiver can receive the message. For example, a producer thread can send messages to a list of consumers, CidList, or broadcast to a consumer channel, ConChan, by:

```
Producer() {
    for ( ;; ) {
        // produce element
        MulticastNonBlock( CidList, msg );     // nonblocking form
        MulticastBlock( CidList, msg );        // blocking form
        BroadcastNonBlock( ConChan, msg ); // nonblocking form
        BroadcastBlock( ConChan, msg );     // blocking form
    }
}
```

There is no equivalent to multicast/broadcast send in non-message passing system because routine call is used is initiate the communication. However, both multicast and broadcast can be simulated either by calling a list of tasks or using indirect communication via a monitor.

13.3.2 Receive

Receive is the basic mechanism to receive a communication and corresponds to a task's mutex member. The minimum arguments to receive are the messages being sent, e.g., receive(msg), where msg is an output argument. The semantics of receive vary depending on the message passing approach. Most message passing approaches support several different kinds of receive, and a programmer selects one appropriate to the communication needs. The following is a list of different kinds of message receives.

13.3.2.1 Receive Any

The simplest form of receive is one accepting any message sent to it, called **receive any**. The receive blocks the calling thread until any message arrives. For example, a consumer thread would receive messages sent from any producer by:

```
Consumer() {
    for ( ;; ) {
        Receive( msg );                        // receive any message
        // consume element from msg
    }
}
```

This kind of receive works like a monitor used only for mutual exclusion purposes, where after processing a call, the next call to any mutex member is processed.

13.3.2.2 Receive Message Specific

A receiver can control messages by indicating the kind of message it is accepting from senders, called **receive message specific**. The kind of message can be derived automatically from the type of the message or it can be an arbitrary value created by the programmer. The former is safer than the latter. In the latter case, the sender has to explicitly specify the message kind in the send, e.g., SendBlock(Cid,msgId,msg) or embed the message kind at some known location in the message (often the first byte or word of the message). The receive blocks the calling thread until a message of the specified kind arrives, and hence, provides external scheduling. It may be possible to specify a list of acceptable message kinds. For example, a consumer thread would receive only certain kinds of messages sent from producers by:

```
Consumer() {
    for ( ;; ) {
        Receive( msg, msgId );          // receive a message of kind msgId
        Receive( msg, msgIdList );      // receive a message of from any kind in the list
        // consume element from msg
    }
}
```

Receive message specific is like an accept statement, where the kind of accepted message is equivalent to the kind of accepted mutex member. A list of message kinds is like an accept statement with multiple accept clauses, accepting multiple mutex members.

13.3.2.3 Receive Thread Specific

A receiver can control messages by indicating the thread it is accepting messages from, called **receive thread specific**. The send does not specify any additional information for receive thread specific, but the thread identifier of the sender must be implicitly sent with the message. The receive blocks the calling thread until a message arrives from the specified thread, and hence, provides external scheduling. It may be possible to specify a list of thread identifiers from which messages are accepted. For example, a consumer thread would receive only messages sent from a particular producer by:

```
Consumer() {
    for ( ;; ) {
        Receive( msg, Pid );            // receive a message from task Pid
        Receive( msg, PidList );        // receive a message from any task Pid in the list
        // consume element from msg
    }
}
```

Receive thread specific is more precise than an accept statement because it indicates a specific thread sending a message not just the kind of message, which might be sent by multiple threads. This capability can only be simulated in μC++ by having a special mutex member (possible private and accessible only through friendship) that is called only by the specific thread. Accepting this mutex member means waiting for the specific thread to call. Interestingly, there is nothing precluding an accept statement from supporting a thread-specific feature, e.g., **_Accept**(Tid, mutex-name).

13.3.2.4 Receive Reply

All the previous receive statements perform two distinct operations. First, an appropriate message is extracted from a thread's message buffer. Second, for blocking sends, the sender is unblocked, ending the rendezvous. There is no reason these two operations have to be bounded together, i.e., it is possible to separate the operations and provide different mechanism for each, called **receive reply** [Gen81]. In this scheme, the receiver blocks until a message arrives but only extracts an appropriate message; unblocking the sender is performed with a separate Reply mechanism. For example, a consumer thread would receive messages sent from any producer by:

```
Consumer() {
    for ( ;; ) {
        Pid = ReceiveReply( msg );    // receive any message
        // consume element from msg
        Reply( Pid );                 // nonblocking
    }
}
```

Since, the sender is not implicitly unblocked at the receive, the thread identifier of the sending thread is returned, which must be stored for use in a subsequent reply. Notice if the reply immediately follows the receive, it is identical to a receive any. In this case, the consumer forces the producer to remain blocked until the element is consumed, which may be necessary if there is a side-effect associated with consuming. Interestingly, the receiver is not required to reply immediately to the last sending thread. Instead, the receiver can store the thread identifier of the sender and reply to it at some arbitrary time in the future, which provides internal scheduling. A condition variable can be mimicked by linking together unreplied thread identifiers and replying to them when appropriate events occur.

For blocking send, the reply can also be used as a mechanism to return a result to the original send, as for synchronous call, rather than having the sender perform a second send to obtain a result. In this case, both the send and reply are modified to receive and send a reply message. Notice the role reversal here, where the send becomes a receive and the reply (which is part of a receive) becomes a send. For example, a producer and consumer thread could perform a bidirectional communication by:

```
Producer() {
    for ( ;; ) {
        // produce element
        SendReply( Cid, replyMsg, sendMsg );
    }
}
```

```
Consumer() {
    for ( ;; ) {
        Pid = ReceiveReply( recvMsg );
        // check element
        Reply( Pid, replyMsg )
        // consume element
    }
}
```

In this scenario, the consumer checks the data before replying and sends back a message to the producer indicating if the element is valid; only if the element is valid it is consumed.

13.3.2.5 Receive Combinations

All of the previous receive control mechanisms are orthogonal, and hence, can be applied together in any combination, with the exception that returning a message with reply to a nonblocking send is either an error or the return message is discarded. For example, it is possible to combine all the control mechanisms:

```
Consumer() {
    for (;;) {
        Pid = ReceiveReply( recvMsg, msgIdList, PidList );
        // check element
        Reply( Pid, replyMsg )
        // consume element
    }
}
```

where the consumer only accepts messages of the specified kinds and only from the specified producer identifiers, and then the consumer replies back to the selected sender with a reply message.

13.3.3 Message Format

Message passing systems can specify restrictions on the form and content of the message itself. One restriction mentioned already is disallowing pointers in a message, but there are others.

Another possible restriction is message size. Some message-passing systems allow variable sized messages of arbitrary size (or very large size, e.g., 2^{32} bytes). Other systems chose a fixed sized message, with the size usually falling in the range 32 to 64 bytes. (This range is also the average size of data passed as arguments for a routine call.) The particular size is usually chosen by monitoring communication activity of a variable sized system and finding the average message size, which usually falls in the range 32 to 64 bytes. In some cases, the decision between variable and fixed size messages may depend on the underlying hardware transport mechanism. For example, Ethernet communication channels support variable sized messages while ATM communication channels only support fixed sized messages.

In general, variable sized messages are more complex to implementation both in the lower-level software and hardware, and the complexity can slow the implementation, but variable sized messages are easy to use because arbitrarily long messages can be sent and received. On the other hand, fixed sized messages are simple to implement and the implementation is usually very fast, but long messages must be broken up and transmitted in pieces and reconstructed by the receiver, which is error prone.

Combining the two approach leads to the worst of both worlds. Because of the two kinds of messages, many routines have to be written twice: one to deal with variable sized messages and one to deal with fixed sized messages, which doubles every possible problem.

13.3.4 Typed Messages

Most complex applications send and receive a number of different kinds of messages. However, when a message arrives, it is only a string of unknown bytes of some particular length; hence, the receive is normally type-unsafe. It is a common convention in message passing for the first byte or word to contain a value identifying the type of

```
void BoundedBuffer() {
    int front = 0, back = 0, count = 0;
    int Elements[20];                                  // bounded buffer
    enum { insert, remove, both } msgtag = insert;

    for ( ;; ) {
        msg = pvm_recv( -1, msgtag );
        switch ( msgtag ) {
          case insert:
            Elements[back] = elem;                     // insert element into buffer
            back = ( back + 1 ) % 20;
            count += 1;
            if ( count > 20 ) msgtag = remove;
            else msgtag = both;
          case remove:
            elem = Elements[front];                    // remove element from buffer
            front = ( front + 1 ) % 20;
            count -= 1;
          case both:
            if
        }
    }
}
```

the message. This message code can be used for receive selection, as in receive-message-specific, and/or to assign a meaning to the string of bytes, i.e., is the data three integers or seven pointers. Some message-passing systems implicitly assign the message codes by generating a unique value based on the field types in the message. However, after each receive, there is a **switch** statement performing a dynamic type-check to determine the kind of the message, which is costly and still potentially error prone. This behaviour is in contrast to argument/parameter passing for routine call, which is statically type-safe.

Furthermore, having two forms of communication, i.e., routine/member call and message passing, leads to confusion deciding when to use each. It is much simpler for programmers to have a single consistent mechanism for communication and routine call seems to be the superior approach because of the parameter/argument lists and static type-checking. It is common in message-passing systems to call routines, using arguments for communication, and have the routine pack or unpack the data into or out of a message and performs the send or receive, respectively. If programmers are going to hide message passing in this way, it begs the question of why use it at all.

Static type-safety and routine call are essential elements of all modern programming-systems. Message passing is type-unsafe and does not following the standard routine call (arguments/parameters) model. Having to work with two programming models is complex, confusing and largely unnecessary. Therefore, there is no long-term future for the message-passing approach; existing message-passing systems will gradually be supplanted by systems using statically-typed routine-call.

13.3.5 PVM/MPI

The current popular message-passing systems for C/C++ are PVM and MPI. Both systems focus on distributed environments, and hence, contain additional routine parameters and routines to deal with threads running on different computers.

13.3.5.1 Thread Creation and Termination

13.3.5.2 Thread Synchronization and Mutual Exclusion

Both systems support receive-message specific, which provides external scheduling, therefore, implementing a bounded buffer is straightforward.

readers-writer

13.4 Concurrent Languages

Concurrent languages are programming languages that contain language features that:

1. provide high-level concurrency constructs to simplify the details and complexities of writing concurrent programs

2. inform the compiler that parts of a program are concurrent rather than sequential

The quality, integration and consistency of the features provided in the first point differentiate a good concurrent programming language from an average or bad one. The second point is crucial for correct and efficient code generation, because certain sequential code optimizations invalidate a concurrent program; only when a compiler can differentiate sequential from concurrent code can it highly optimize the former and generate correct code for the latter. Over the years, many languages have provided different styles and mechanisms for concurrency; only a small subset of these languages are discussed. The languages chosen are ones the reader might encounter at some time in their programming career or are historically significant.

13.4.1 Ada 95

The programming language Ada was developed in two major design efforts: one culminating in Ada 83 [IBFW86], followed by Ada 95 [Int95]. Ada 83 provides tasks, which are loosely based on the concept of objects; Ada 95 provides object-based monitors. All the concurrency features are embedded in the language, i.e., there is no library component, so the concurrency features are both sound and efficient.

13.4.1.1 Thread Creation and Termination

Threads are defined in Ada 95 by a special task type constructor, as in μC++ with **_Task**, e.g.:

```
task type Tt is                         -- interface
     entry mem( input : in Integer, output : out Integer ); -- public mutex member prototypes
  private
     -- implementation declarations, including more entry member prototypes
end Tt;
task body Tt is                         -- implementation
     -- local declarations for task body, including functions and procedures
  begin                                 -- thread starts here
     -- task body (equivalent to constructor and task main in uC++)
          -- all entry member bodies defined within task body
end Tt;
```

The interface and the implementation are separate, with the interface listing the mutex members, called entries, followed by private variables for the task. Mutex members can be overloaded, as in C++, and private; private mutex members are used for scheduling (see requeue discussion). This separation is the same as defining all member bodies outside a class definition in C++, i.e., no inline definitions of routine members. Unlike μC++, the bodies of the entry (mutex) members are specified within the task body, which is discussed in the next section. As well, an entry member can only return a value through an out parameter, i.e., no function members. The task body is where the new thread begins execution, and it combines the constructor and the task main of μC++. This concatenation is possible because there is no inheritance among task types, so separation of construction from the task main or having an inner facility is not required (see Section 13.2.2.1, p. 374).

Task types and objects can be declared in any Ada declaration context:

```
declare
     type ptype is access Tt;           -- must have named pointer type
     loc : Tt;                          -- local creation
     arr : array(1..10) of Tt;          -- local creation, array
     prt : ptype := new Tt;             -- dynamic creation
begin
     . . .
end; -- wait for termination of loc, arr, prt
```

When a task is created, the appropriate declaration initializations are performed by the creating thread. The stack component of the task's execution-state is created and the starting point is initialized to the task body. Then a new thread of control is created for the task, which begins execution at the task body. From this point, the creating thread executes concurrently with the new task's thread; the task body executes until its thread blocks or terminates.

A task terminates when its task body terminates. When a task terminates, so does the task's thread of control and execution-state. As in μC++, storage for a task cannot be deallocated while a thread is executing, which means a block cannot terminate until all tasks declared in it terminate. Notice, in the above example, the access (pointer) variable prt is terminated at the end of the block; this results from Ada's accessibility rule, which states the duration of a dynamically allocated object cannot outlive the scope of its access type. Therefore, the dynamically allocated task referenced by prt cannot exceed the scope of the type ptype and the task object is implicitly deleted, resulting in the block waiting for the task's termination.

Unlike μC++, where a task turns into a monitor after its thread terminates, an Ada task does not turn into a protected object (Ada equivalent of a monitor, see Section 13.4.1.2). The reason has to do with the placement of a task's entry-members *within* the accept statement rather than having separate member routines, as in C++. Because the entry-members are inlined, it may be necessary to have multiple "overload" versions of the same entry member. In this case, a call to this entry member is ambiguous without an accept statement being executed by the task's thread to differentiate among multiple entry-member definitions.

Figure 13.7 illustrates the Ada 95 equivalent for starting and terminating threads in a generic bounded-buffer example (see the right hand side of Figure 7.11, p. 203). The bounded buffer is presented shortly. Notice, the interface for both producer and consumer specify a parameter, called a discriminate, which is equivalent to a template parameter in C++. An argument must be specified when the type is used in the declaration of objects.

13.4.1.2 Thread Synchronization and Mutual Exclusion

Ada 95 provides two mechanism for mutual exclusion: the protected object (monitor-like) and the task. Both constructs have mutex members, which execute mutually exclusively of one another, and both support synchronization via external scheduling, albeit with some restrictions. However, the mechanisms for external scheduling are different for protected objects and tasks. The default scheduling implementation for both protected objects and tasks has a queue for each entry member (mutex queues) but no single entry-queue to maintain overall temporal order of arrival. (Ada actually uses the term "entry queue" for a mutex queue, but this conflicts with prior terminology.) As a result, when there is a choice for selecting among entry members, an arbitrary selection is made rather than selecting the task waiting the longest. This semantics presents problems in implementing certain solutions to concurrent problems.

Protected Object An Ada 95 protected type is like a class and has three kinds of member routines: function, procedure and entry:

```
protected type Pt is                -- interface
    function(...) Op1 return ...;   -- public mutex member prototypes
    procedure(...) Op2;
    entry(...) Op3;
  private
    -- implementation declarations, including more function, procedure, and entry members
end Pt;
protected body Pt is                -- implementation
    function(...)  Op1 return ... is begin function-body end Op1;
    procedure(...) Op2 is begin procedure-body end Op2;
    entry(...) when ... Op3 is begin entry-body end Op3;
end Pt;
```

Mutex members can be overloaded, as in C++, and private; private mutex members are used for scheduling (see requeue discussion).

Function members are read only with respect to the protected-object's data, while procedure and entry members are read/write. Function members have mutual exclusion with respect to procedures and entries but not with other functions, i.e., multiple function members can run simultaneously but not with a procedure or entry member. Hence, there is a simple built-in readers and writer facility. There is no direct equivalent to function members in μC++ as **_Nomutex** members have no mutual-exclusion property, and run simultaneously with mutex members; however, it is possible to explicitly code an equivalent facility, as is done with readers in the readers/writer problem. Conversely, there is no equivalent to **_Nomutex** members in Ada protected objects as all protected-object members acquire some kind of mutual exclusion. Function members have no external scheduling. Procedure members are like simple mutex

```
with Ada.Numerics.Discrete_Random;
procedure bbmon is
    package Random_Integer is
        new Ada.Numerics.Discrete_Random(Integer);
    use Random_Integer;
    Gen : Generator;

    package IntBuf is -- generic bounded buffer
        new BoundedBufferPkg(Integer, 20);
    use IntBuf;

    type BBptr is access all BoundedBuffer;

    task type producer( buf : BBptr ) is -- interface
    end producer;

    task body producer is -- implementation
        NoOfElems : constant Integer := 20;
        elem : Integer;
    begin
        for i in 1..NoOfElems loop
            -- no equivalent to yield
            elem := Random(Gen) mod 100 + 1;
            buf.insert( elem );
        end loop;
    end producer;

    task type consumer( buf : BBptr ) is -- interface
    end consumer;
    task body consumer is -- implementation
        elem : Integer;
    begin
        loop
            buf.remove( elem );
          exit when elem = -1;
            -- no equivalent to yield
        end loop;
    end consumer;

    NoOfCons  : constant Integer := 3;
    NoOfProds : constant Integer := 4;
    buf : aliased BoundedBuffer;
begin
    declare
        type conPtr is access all consumer;
        cons : array(1..NoOfCons) of conPtr;
    begin
        declare
            type prodPtr is access all producer;
            prods : array(1..NoOfProds) of prodPtr;
        begin
            for i in 1..NoOfCons loop  -- start consumers
                cons(i) := new consumer( buf'access );
            end loop;
            for i in 1..NoOfProds loop -- start producers
                prods(i) := new producer( buf'access );
            end loop;
        end; -- wait for producers
        for i in 1..NoOfCons loop -- terminate consumers
            buf.insert( -1 );
        end loop;
    end; -- wait for consumers
end bbmon;
```

Figure 13.7: Ada 95: Producer/Consumer Creation/Termination

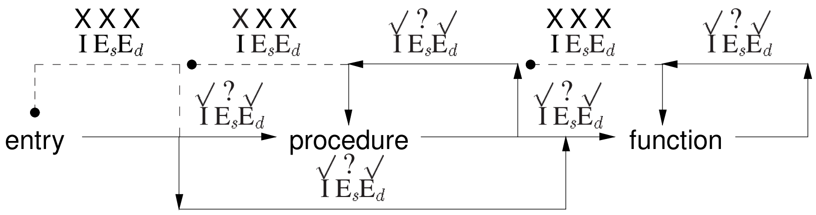
members in μC++, providing read/write mutual exclusion but no external scheduling. Entry members are equivalent to mutex members in μC++, providing read/write mutual exclusion and a kind of external scheduling.

Operations allowed in a protected object's members are restricted by the requirement that no operation can block. The reason for this severe restriction is to allow nonqueued locking mechanisms, e.g., spin lock versus semaphore, to be used to implement the mutual exclusion. Therefore, once a task has entered a protected object it must complete the member and leave without needing to block and be placed on a queue of waiting tasks. A task can make a call to another protected object and delay while attempting to acquire the spin lock for that object; in this case, the task does not have to queue, it can spin. Since a spin lock can only be acquired once, this restriction precludes a call to other members in a protected object, which is too restrictive. To allow some calls within a protected object, Ada differentiates between internal and external calls, e.g.:

```
declare
    protected type Pt is
        -- as above
    end Pt;

    PO : Pt;                           -- declarations
    Other_Object : Some_Other_Protected_Type;

    protected body Pt is               -- implementation
        . . .
        procedure Op2 is begin
            Op1;                       -- internal call, like C++ this->Op1
            Op2;                       -- internal call, like C++ this->Op2
            Pt.Op1;                    -- internal call, like C++ this->Pt::Op1
            PO.Op1;                    -- external call, same object
            Other_Object.Some_Op; -- external call, different object
        end Op2;
        . . .
    end Pt;
begin
    PO.Op2;                            -- external call
end
```

An internal call is any call to an object's member made through the equivalent of the object's **this** variable, and such calls do not acquire the protected-object's mutex lock. Otherwise a call is an external call, where there is a further distinction between calls to the same object or a different object.

Given these definitions, the following calls are allowed:



√ call allowed, X call disallowed, ? call implementation dependent
I internal call; E_s external call, same object; E_d external call, different object

From left to right, an entry member can make internal calls to a function or procedure, as this does not require the protected-object's mutex lock. An external call is allowed to a function or procedure in another protected object, but an external call to the same object is defined as an error; however, the implementation can allow the call, but such calls are not portable and should be avoided. An entry member cannot call an entry member because the call is conditional, which can result in blocking until the condition becomes true; however, blocking is disallowed. A procedure member can make internal calls to a function or procedure, as this does not require the protected-object's mutex lock. An external call is allowed to a function or procedure in another protected object, but an external call to the same object is defined as an error; however, the implementation can allow the call, but such calls are not portable and should be avoided. A procedure member cannot call an entry member because the call is conditional, which can result in blocking until the condition becomes true; however, blocking is disallowed. A function member can only make internal calls to a function because of the read-only property; calls to an entry or procedure could violate the read-only property because the entry or procedure can change the object. An external call is allowed to a function in another protected object, but an external call to the same object is defined as an error; however, the implementation can allow the call, but such calls are not portable and should be avoided. These complex rules, except for the read-only property of functions, could be eliminated by allowing recursive calls for any kind of member at the cost of a more expensive locking mechanism.

Protected-object external scheduling is accomplished using guards at the beginning of entry members, which is a form of restricted automatic-signal monitor (see Section 9.11.3.3, p. 288), as in:

```
entry insert( elem : in ELEMTYPE ) when count < Elements'length is ...
```

where the when clause on the member definition must be true before a call is allowed to proceed. (The syntax 'length is called an attribute in Ada, where attributes provides access to both compile time and runtime information, accessing the length of the array.) Notice, this syntax and semantics is almost identical to the conditional critical-region in Section 9.2, p. 251. This form of automatic-signalling is external scheduling because a task does not enter the protected object first, possibly reading or changing it, before blocking on the conditional expression. It is restricted because only global protected-object variables can appear in the conditional expression, parameter or local member variables are disallowed. Only entry members can have a when clause; functions and procedures cannot. Also, creating an efficient implementation for the conditional expression can be challenging. Since only global protected-object variables are allowed in a conditional expression, the time it takes to find the next task to execute is determined by the cost of re-evaluating the conditional expressions. It is unnecessary to wake up some or all of the tasks to access their local variables, as for general automatic-signal monitors. However, the cost is usually greater than using cooperation with accept statements. Finally, this form of external scheduling suffers the same restrictions as external scheduling in μC++ (see Section 9.5.2, p. 266): entry selection cannot depend on local mutex-member values (including parameter values), and once selected, a task cannot block in the mutex-member body using this mechanism. Therefore, some form of internal scheduling is necessary.

While there is no internal scheduling for protected objects, Ada provides a mechanism to simulate it using external scheduling. Rather than using condition variables with signal/wait, a requeue statement is provided but it can only be used in an entry member. The requeue statement terminates the current execution of the entry member and requeues the original call to an equivalent entry member in the current protected object, or another protected object or task so it can be re-accepted at some later time, as in:

```
requeue mem;   -- internal requeue to "mem" of current protected object
requeue m.mem; -- external requeue to "mem" of different protected object
```

The requeue is not a call, but a redirection of a call, so it does not specify any arguments, just the new location for the original call to restart. The specified entry member must have the same prototype as the original or have no parameters. The requeued call is then handled as a new entry-member call by the protected object and is subsequently selected by the entry-member guards. The semantics of external requeue to the same object are implementation dependent, and should be avoided.

Unfortunately, Ada requeue suffers the same problems discussed in Section 9.5.2.2, p. 269. An entry member working on a request may accumulate complex execution and data state. Unfortunately, the execution location and temporary results cannot be bundled and forwarded to the new entry member handling the next step of the processing because the signature of the new entry member must be the same or empty; hence, there is no way to pass new information. Therefore, the accumulated information must be stored in global protected-object variables for access by the new entry member. Alternatively, the temporary results can be re-computed at the start of the requeued entry member, if possible. In contrast, waiting on a condition variable automatically saves the execution location and any partially computed state.

Figure 13.8 illustrates the Ada 95 equivalent for the external-scheduling bounded-buffer monitor in Figure 9.4, p. 259. Unfortunately, the query member has unnecessary mutual exclusion with entry members insert and remove, which inhibits concurrency. Notice the when clauses controlling entry into insert and remove. Both conditional expressions only reference global protected-object variables, i.e., count and the array size.

The lack of real internal scheduling, and in particular condition queues, in Ada presents difficulties for certain kinds of problems. For example, the dating service problem presented in Section 9.5.2.1, p. 266 is a problem in Ada because the different compatibility codes require different places to block when waiting for a corresponding partner. Two approaches are presented in Section 9.5.2.1, p. 266: arrays of conditions, when the compatibility codes are small and dense, and lists of conditions, when the codes are large and sparse. Ada can handle the former case easily by using arrays of entry members, called entry families, but the latter case is more difficult as it requires the equivalent of a dynamically allocated condition variable. The array of entry members acts like the array of condition variables, providing a place to requeue and wait for a compatible partner. However, arrays of entry members introduces significant new syntax and semantics into the language, and still cannot handle large, sparse codes.

Figure 13.9, p. 391 shows the use of entry families to provide a dating service with small, dense compatibility codes. The two public entry members, Girl and Boy, have three parameters because entry routines cannot return

```
generic
    type ELEMTYPE is private;                    -- template parameters
    Size: in Natural;
package BoundedBufferPkg is
    type BufType is array(0..Size-1) of ELEMTYPE;

    protected type BoundedBuffer is              -- interface
        function query return Integer;
        entry insert( elem : in ELEMTYPE );
        entry remove( elem : out ELEMTYPE );
      private
        front, back, count : Integer := 0;
        Elements : BufType;                      -- bounded buffer
    end BoundedBuffer;
end BoundedBufferPkg;

package body BoundedBufferPkg is
    protected body BoundedBuffer is              -- implementation
        function query return Integer is
        begin
            return count;
        end query;

        entry insert( elem : in ELEMTYPE ) when count < Elements'length is
        begin
            Elements(back) := elem;              -- insert into buffer
            back := ( back + 1 ) mod Elements'length;
            count := count + 1;
        end insert;

        entry remove( elem : out ELEMTYPE ) when count > 0 is
        begin
            elem := Elements(front);             -- remove from buffer
            front := ( front + 1 ) mod Elements'length;
            count := count - 1;
        end remove;
    end BoundedBuffer;
end BoundedBufferPkg;
```

Figure 13.8: Ada 95: Protected-Object Bounded Buffer, External Scheduling

values, so the phone number of a task's partner is returned through the out parameter partner. The private entry members are two entry arrays, Girls and Boys, on which tasks wait if there is no available partner with a corresponding compatibility code, plus an additional entry member, Exchange, on which tasks must wait to complete an exchange of phone numbers. The entry Exchange is used in the same way as the condition Exchange in the μC++ monitor solutions.

The two public entry members, Girl and Boy, both have a guard that accepts calls at any time, except when an exchange of phone numbers is occurring within the protected object. The expression exchange'count returns the number of tasks blocked on the mutex queue for member exchange, which is like using empty on a condition queue in μC++ to know if any tasks are blocked on it. The private entry members are all controlled using flag variables, called triggers, to indicate when it is possible to continue execution. The triggers for the entry family members are the arrays GTrig and BTrig, plus the variable ExTrig. Triggers are often used instead of boolean expressions when the boolean expressions become complex and must be reevaluated ever time the protected object becomes inactive.

Both entry members, Girl and Boy, start by checking for a corresponding partner, which is accomplished by checking the number of tasks waiting on the mutex queue for the corresponding entry family, e.g., Boys(ccode)'count. If there is no available partner, a task requeues itself onto the appropriate entry family with the same compatibility code. The current entry member is now terminated and the original call is requeued on the mutex queue for the specified

```
generic
    type ccset is range<>;                      -- template parameters
package DatingServicePkg is
    type triggerType is array(ccset) of Boolean;

    protected type DatingService is             -- interface
        entry Girl( Partner : out Integer; PhNo : in Integer; ccode : in ccset );
        entry Boy( Partner : out Integer; PhNo : in Integer; ccode : in ccset );
      private
        entry Girls(ccset)( Partner : out Integer; PhNo : in Integer; ccode : in ccset );
        entry Boys(ccset)( Partner : out Integer; PhNo : in Integer; ccode : in ccset );
        entry Exchange( Partner : out Integer; PhNo : in Integer; ccode : in ccset );

        ExPhNo, GPhNo, BPhNo : Integer;
        GTrig, BTrig : TriggerType := (ccset => false); -- initialize array to false
        ExTrig : Boolean := false;
    end DatingService;
end DatingServicePkg;

package body DatingServicePkg is
    protected body DatingService is             -- implementation
    entry Girl( Partner : out Integer;
            PhNo : in Integer; ccode : in ccset )
      when exchange'count = 0 is
    begin
        if Boys(ccode)'count = 0 then
            requeue Girls(ccode);-- no return
        else
            GPhNo := PhNo;
            BTrig(ccode) := true;
            requeue exchange;   -- no return
        end if;
    end Girl;

    entry Boy( Partner : out Integer;
            PhNo : in Integer; ccode : in ccset )
      when exchange'count = 0 is
    begin
        if Girls(ccode)'count = 0 then
            requeue Boys(ccode);-- no return
        else
            BPhNo := PhNo;
            GTrig(ccode) := true;
            requeue exchange;   -- no return
        end if;
    end Boy;

    entry Girls(for code in ccset)( Partner : out Integer;
            PhNo : in Integer; ccode : in ccset )
      when GTrig(code) is
    begin
        GTrig(ccode) := false;
        ExPhNo := PhNo;
        Partner := BPhNo;
        ExTrig := true;
    end Girls;

    entry Boys(for code in ccset)( Partner : out Integer;
            PhNo : in Integer; ccode : in ccset )
      when BTrig(code) is
    begin
        BTrig(ccode) := false;
        ExPhNo := PhNo;
        Partner := GPhNo;
        ExTrig := true;
    end Boys;

    entry exchange( Partner : out Integer;
            PhNo : in Integer; ccode : in ccset )
      when ExTrig is
    begin
        ExTrig := false;
        Partner := ExPhNo;
    end Exchange;

    end DatingService;
end DatingServicePkg;
```

Figure 13.9: Ada 95: Scheduling using Parameter Information, Solution 1

entry family. If there is an available partner, the caller's phone number is copied into the corresponding global variable, GPhNo or BPhNo, respectively, the trigger for the partner entry-family member is set to true, and the task requeues itself on the entry exchange to wait for the partner to wake up and place its phone number in the global variable, ExPhNo. Once a task requeues on exchange, new tasks cannot enter the dating-service object until the exchange is complete.

When the partner is unblocked because its trigger is true, it immediately resets the appropriate trigger, GTrig(ccode) or BTrig(ccode), copies its phone number into the global variable ExPhNo, returns its partner's phone number by assigning from the corresponding global variable, GPhNo or BPhNo, to the output parameter Partner, and finally sets the trigger for the partner blocked on entry-member exchange to true. The entry call returns, completing the original call to the dating service.

When the partner waiting on exchange is unblocked, it immediately resets trigger ExTrig, returns its partner's phone number by assigning from the global variable ExPhNo to the output parameter Partner, and the entry call returns, which completes the original call to the dating service.

Having to use large numbers of trigger flags is the main drawback to this style of controlling tasks with an Ada protected-object. As pointed out in Chapter 2, p. 7, flag variables should be avoid wherever possible, and replaced by explicit control flow. In many cases, Ada protected-objects result in the opposite scenario, resulting in a correspondingly poor programming style. Finally, the technique of entry families, i.e., arrays of entry members, does not scale to large, sparse codes.

To handle large, sparse codes requires a different data structure, e.g., linked list, which in turn, requires the ability to dynamically create some form of condition variable. Figure 13.10 shows it is possible to mimic dynamically allocated condition variables in Ada by creating a linked list of Ada protected-objects, and using each one as a condition variable. The protected-object condition has two members, Wait and Signal, used to block and unblock a waiting task, respectively. Because the waitTrig is initialized to false, a call to Wait blocks. The Signal routine resets waitTrig so a call to Wait can now proceed. As well, Signal is used to transfer information from the signalling task into the waiting task for the condition variable, so when a task enters Wait, there is data for it to return to the original call. In this case, the data transfered from signaller to waiting task is the phone number of the signaller. The dating-service protected-object creates two linked lists: one for waiting girl and boy tasks. Each node of the lists contains the compatibility code and phone number of the waiting task, along with the condition variable on which it waits. Tasks call the appropriate entry members of DatingService, which always have true guards. A task then searches the oppose gender list for a compatible partner. If a partner is not found, a node is created, initialized, and linked to the end of the appropriate list. Then the task blocks by requeueing on the Wait member for the condition variable in the node; there is only one task blocked on this condition variable. If a partner is found, the search removes the partner from the list and sets N to point to this partner's node. The partner's phone number is now copied from the node to return to the original call, and the Signal routine is called for the partner's condition variable, passing it the phone number of its partner. When a waiting task unblocks, it copies this phone number back to its original call. Notice, the control logic is simpler in this solution than the previous solution because of the cooperation used.

Task As described in Section 13.4.1.1, p. 385, an Ada 95 task may have both public and private entry-member routines that are equivalent to mutex members in μC++, providing mutual exclusion.

Task external scheduling is accomplished using an accept statement, as in:

```
select
  when expression =>        -- guard
    accept mem(...) do      -- entry member definition
        ...                 -- entry member body
    end mem
    ...                     -- executed after accepted call
or when ...
    ...
or                          -- optional
    terminate               -- accept destructor
end select
```

which is very similar to that in μC++. In fact, μC++ adapted the accept statement from Ada for its external scheduling. The main difference is the placement of the entry member body. Instead of having true member routines, entry members are nested within the accept statement, making the accept clause have parameters and a routine body; hence,

```
generic
    type ccset is range<>;              -- template parameters
package DatingServicePkg is
    type triggerType is array(ccset) of Boolean;

    protected type DatingService is     -- interface
        entry Girl( Partner : out Integer; PhNo : in Integer; ccode : in ccset );
        entry Boy( Partner : out Integer; PhNo : in Integer; ccode : in ccset );
      private
    end DatingService;
end DatingServicePkg;

package body DatingServicePkg is

    protected type condition is         -- interface
        entry Wait( Partner : out Integer;
                PhNo : in Integer; ccode : in ccset );
        procedure Signal( PhNop : in Integer );
    private
        waitTrig : Boolean := false;
        PhNoEx : Integer;
    end condition;

    protected body condition is         -- implementation
        entry Wait( Partner : out Integer;
                PhNo : in Integer; ccode : in ccset )
          when waitTrig is
        begin
            Partner := PhNoEx;
            waitTrig := false;
        end Wait;

        procedure Signal( PhNop : in Integer ) is
        begin
            WaitTrig := true;
            PhNoEx := PhNop;
        end Signal;
    end condition;

    type Node is record
        ccode : ccset;
        PhNo : Integer;
        cond : condition;
    end record;

    package NodeList is new Gen_List( Node );
    use NodeList;

    girls, boys : List;

    protected body DatingService is -- implementation
        entry Girl( Partner : out Integer;
                PhNo : in Integer; ccode : in ccset )
          when true is
            N : Elmtp;
        begin
            SearchRm( N, boys, ccode );
            if N = null then -- no partner
                N := new Node;
                N.ccode := ccode; N.PhNo := PhNo;
                girls := Append( girls, N );
                requeue N.cond.Wait; -- no return
            else -- partner
                Partner := N.PhNo;
                N.cond.Signal( PhNo );
            end if;
        end Girl;

        entry Boy( Partner : out Integer;
                PhNo : in Integer; ccode : in ccset )
          when true is
            N : Elmtp;
        begin
            SearchRm( N, girls, ccode );
            if N = null then -- no partner
                N := new Node;
                N.ccode := ccode; N.PhNo := PhNo;
                boys := Append( boys, N );
                requeue N.cond.Wait; -- no return
            else -- partner
                Partner := N.PhNo;
                N.cond.Signal( PhNo );
            end if;
        end Boy;
    end DatingService;

end DatingServicePkg;
```

Figure 13.10: Ada 95: Scheduling using Parameter Information, Solution 2

the static scope of the entry body is different than in μC++. The Ada approach allows different routine bodies for the same entry name in different accept statements. The main problem with placing the entry in the accept statement is that it precludes virtual routine redefinition, which is not an issue in Ada as task types cannot inherit. A further Ada restriction is that the accept statement can only appear in the task body, not any of its local subprograms, which can force an awkward coding style in certain situations. This restriction ensures a nested task cannot call a local subprogram of an enclosing task that subsequently accepts an entry routine of the enclosing task; a task can only accept its own entry routines to ensure correct locking for mutual exclusion. As well, an entry member cannot accept itself because of ambiguity in the resolution of formal parameter names. Finally, the optional terminate clause at the end of the accept statement is chosen when the block containing a local task declaration ends or the block containing the task's access-type ends for dynamic task declaration. The termination clause is like accepting the destructor in μC++.

Like Ada protected-objects, internal scheduling for tasks is accomplished by converting it into external scheduling using the requeue statement. The requeue statement terminates the current execution of the entry member and requeues the original call to a type-equivalent entry-member in the current task, or another protected object or task so it can be re-accepted at some later time. However, the use of requeue for tasks suffers the same problems as for protected objects with respect to loss of any accumulated execution and data.

Figure 13.11 illustrates the Ada 95 equivalent for the external-scheduling bounded-buffer task in Figure 10.6, p. 323. Unfortunately, the query member has unnecessary mutual exclusion with entry members insert and remove, which inhibits concurrency. Like μC++, the when clauses control the accepts of insert and remove when the buffer is full or empty, respectively, and the buffer management code has been moved outside of the entry body to maximize concurrency. As well, the loop around the select statement is stopped by accepting terminate, which immediately terminates the task. Immediate termination prevents a task from cleaning up directly, but garbage collection and controlled types can provide indirect clean up.

The next example is the readers and writer problem. Figure 13.12, p. 396 shows the first solution to the readers/writer problem in Ada. This solution relies on the built-in semantics provided by Ada protected-objects, i.e., multiple read-only functions can execute simultaneously with mutual exclusion from procedure and entry members, while procedures execute with mutual exclusion with respect to all protected-object members. With this built-in semantics, it is only necessary to make the Read member a function and the Write member a procedure to construct a solution. Notice, the Read routine returns a dummy result because all Ada functions must return a value.

Unfortunately, this trivial solution may suffer starvation and does suffer from staleness. The reason is that the default selection policy chooses arbitrarily *among* mutex queues when multiple queues are eligible. (Other selection policies are available for real-time programming in Ada.) This semantics applies for protected objects with multiple true guards or accept statements with multiple acceptable accept clauses. As a result, it is conceivable for starvation to occur if readers and writers are always queued on both entry members and the implementation always chooses one over the other. However, a reasonable implementation for servicing the mutex queues should provide some level of fairness. Nevertheless, the arbitrary selection among queues does result in staleness because temporal order is no longer maintained among the tasks waiting on the different mutex queues. μC++ deals with this problem by having a single FIFO entry queue, instead of multiple FIFO mutex-queues as in Ada.

Staleness can only be reduced for the Ada default-selection policy due to the arbitrary selection among mutex queues. One way to reduce staleness is by converting from an external-scheduling style to an internal-scheduling style. Hence, tasks are removed from the public-entry mutex-queues as quickly as possible, and requeued onto a single private-entry mutex-queue, which is FIFO. A consequence of this conversion is that public entries always have open (true) guards to allow immediate entry for subsequent FIFO requeueing, if necessary. However, the arbitrary selection among all acceptable entry members, public and private, means external tasks can barge into the protected object at any time, so the protected object has to be treated as a no-priority monitor. Notice, there is still a small window where multiple tasks can queue on public entry members while the protected-object is active, e.g., if the task in the protected object is time-sliced, and the subsequent arbitrary selection results in staleness due to non-FIFO selection among mutex queues.

Figure 13.13, p. 396 shows a solution to the readers/writer problem, which reduces staleness by converting from an external-scheduling to an internal-scheduling style. The solution is based on the PThreads' solution in Figure 13.6, p. 373, which is problematic because once a task requeues on rwcond it is impossible to tell if it is a reader or writer task, and there is the barging issue because of arbitrary selection among acceptable entry-members. A task's kind, i.e., reader or writer, is handled by maintaining an explicit queue of these values, which is added to and removed from

```
generic
    type ELEMTYPE is private;                    -- template parameters
    Size: in Natural;
package BoundedBufferPkg is
    type BufType is array(0..Size-1) of ELEMTYPE;

    task type BoundedBuffer is                   -- interface
        entry query( cnt : out Integer );
        entry insert( elem : in ELEMTYPE );
        entry remove( elem : out ELEMTYPE );
    end BoundedBuffer;
end BoundedBufferPkg;

package body BoundedBufferPkg is
    task body BoundedBuffer is                   -- implementation
        front, back, count : Integer := 0;
        Elements : BufType;                      -- bounded buffer
    begin
        loop
            select
                accept query( cnt : out Integer ) do
                    cnt := count;
                end;
            or when count < Elements'length =>   -- guard
                accept insert( elem : in ELEMTYPE ) do
                    Elements(back) := elem;      -- insert into buffer
                end insert;
                back := ( back + 1 ) mod Elements'length;
                count := count + 1;
            or when count > 0 =>                 -- guard
                accept remove( elem : out ELEMTYPE ) do
                    elem := Elements(front);     -- remove from buffer
                end;
                front := ( front + 1 ) mod Elements'length;
                count := count - 1;
            or
                terminate;                       -- task stops here
            end select;
        end loop;
    end BoundedBuffer;
end BoundedBufferPkg;
```

Figure 13.11: Ada 95: Task Bounded Buffer, External Scheduling

when a task blocks and unblocks, respectively. The barging issue is handled by checking on entry to StartRead and StartWrite for any blocked tasks waiting to enter rwcond; these blocked tasks have already been waiting, and therefore, have priority over any arriving task. Interestingly, because both reader and writer tasks requeue on the same entry, rwcond, the number and type of parameters of all three members must be identical, which may not always be possible. If the parameter lists are different for the readers and writers, a more complex approach must be used, creating a single list of two kinds of protected-objects using the approach shown in the dating-service solution in Figure 13.10, p. 393. Again, notice the use of a trigger, rwTrig, to restart waiting tasks from the FIFO mutex queue for rwcond. This trigger is toggled true or false by a restarting task depending on the kind (reader or writer) of the next waiting task on the rwcond mutex queue. Finally, the interface has four members, which must be called in pairs by the user, instead of two members, as in the first solution. In this approach, it is impossible to create the same interface as the first solution with a protected object because all protected-object members acquire some form of mutual exclusion. To achieve the interface of the first solution requires embedding this protected-object within another object that has the correct interface but no mutual exclusion on the calls; this wrapper object then makes the four calls to the protected object.

```
protected type ReadersWriter is            -- interface
    function Read(...) return Integer;
    procedure Write(...);
  private
    Rcnt : Integer := 0;
end ReadersWriter;

protected body ReadersWriter is            -- implementation
    function Read(...) return Integer is
    begin
        -- read
        return ...;
    end Read;

    procedure Write(...) is
    begin
        -- write
    end Write;
end ReadersWriter;
```

Figure 13.12: Ada 95: Readers/Writer, Solution 1

```
type RWkind is ( READER, WRITER );
package rwlist is new Gen_List( RWkind );
use rwlist;

protected type ReadersWriter is
    entry StartRead(...);
    entry EndRead;
    entry StartWrite(...);
    entry EndWrite;
  private
    entry rwcond(...);

    rw : List := Nil;
    rcnt, wcnt : Integer := 0;
    rwTrig : Boolean := false;
end ReadersWriter;

protected body ReadersWriter is
    entry StartRead(...) when true is
    begin
        if wcnt > 0 or else rwcond'count > 0 then
            rw := Append( rw, READER );
            requeue rwcond; -- no return
        end if;
        rcnt := rcnt + 1;
    end StartRead;

    entry EndRead when true is
    begin
        rcnt := rcnt - 1;
        if rcnt = 0 and then rwcond'count > 0 then
            rwTrig := true;
        end if;
    end EndRead;

    entry StartWrite(...) when true is
    begin
        if rcnt > 0 or else wcnt > 0 or
                else rwcond'count > 0 then
            rw := Append( rw, WRITER );
            requeue rwcond; -- no return
        end if;
        wcnt := 1;
    end StartWrite;

    entry EndWrite when true is
    begin
        wcnt := 0;
        if rwcond'count > 0 then
            rwTrig := true;
        end if;
    end EndWrite;

    entry rwcond(...) when rwTrig is
        kind : RWkind;
    begin
        rwTrig := false;
        kind := Element( rw );
        rw := Remove( rw );
        if kind = READER then
            if rwcond'count > 0 and
                 then Element( rw ) = READER then
                rwTrig := true;
            end if;
            rcnt := rcnt + 1;
        else
            wcnt := 1;
        end if;
    end rwcond;
end ReadersWriter;
```

Figure 13.13: Ada 95: Readers/Writer, Solution 2

Protected Object

```
protected type ReadersWriter is
    entry StartRead(. . .);
    entry EndRead;
    entry StartWrite(. . .);
    entry EndWrite;
  private
    entry wcond(. . .);
    rcnt, wcnt : Integer := 0;
    wTrig : Boolean := false;
end ReadersWriter;

protected body ReadersWriter is
    entry StartRead(. . .)
      when wcnt = 0 and then wcond'count = 0 is
    begin
        rcnt := rcnt + 1;
    end StartRead;

    entry EndRead when true is
    begin
        rcnt := rcnt - 1;
        if rcnt = 0 then
            if wcond'count > 0 then
                wTrig := true;
            end if;
        end if;
    end EndRead;

    entry StartWrite(. . .)
      when wcnt = 0 and then wcond'count = 0 is
    begin
        if rcnt > 0 then
            requeue wcond;
        end if;


        wcnt := 1;
    end StartWrite;

    entry EndWrite when true is
    begin
        wcnt := 0;
    end EndWrite;

    entry wcond(. . .) when wTrig is
    begin
        wTrig := false;
        wcnt := 1;
    end wcond;
end ReadersWriter;
```

Task

```
task type ReadersWriter is
    entry StartRead(. . .);
    entry EndRead;
    entry StartWrite(. . .);
    entry EndWrite;
end ReadersWriter;


task body ReadersWriter is
    rcnt, wcnt : Integer := 0;
begin
    loop
        select
          when wcnt = 0 =>
            accept StartRead(. . .) do
                rcnt := rcnt + 1;
            end StartRead;
        or
            accept EndRead do

                rcnt := rcnt - 1;
            end EndRead;







        or
          when wcnt = 0 =>
            accept StartWrite(. . .) do
                while rcnt > 0 loop
                    accept EndRead do
                        rcnt := rcnt - 1;
                    end EndRead;
                end loop;
                wcnt := 1;
            end StartWrite;
        or
            accept EndWrite do

                wcnt := 0;
            end EndWrite;
        or
            terminate;
        end select;
    end loop;


end ReadersWriter;
```

Figure 13.14: Ada 95: Readers/Writer, Solution 3 (a & b)

This structure is accomplished in μC++ by combining no-mutex and mutex members.

Figure 13.14 shows two approaches for a third solution to the readers/writer problem; the left approach uses a protected-object and the right a task. The solution is based on the right solution in Figure 9.18, p. 281, which uses external scheduling. For the protected-object solution, the two end members, EndRead and EndWrite, both have true guards as all the control is done in the two start members, i.e., once access to the resource is granted, a task can always end without additional blocking. The guards for StartRead and StartWrite preclude calls if a writer is using the resource or there is a writer delayed because readers are using the resource. After entering StartWrite, a writer delays if readers are using the resource, and requeues its call to wcond. No new readers or writers can start because wcond'count is now 1 so there is no starvation. When the last reader of the current group finishes, it checks for a delayed writer and unblocks it by setting wTrig. The unblocked writer resets wTrig but sets wcnt to 1, so no new readers and writers can enter. When a writer finishes and sets wcnt to 0, Ada now makes an arbitrary choice between tasks waiting on the StartRead and StartWrite mutex queues, which maximizes the potential for staleness.

For the task solution, accept statements can be used. However, switching to a task is inefficient because the task's thread is superfluous; the task's thread mostly loops in the task body accepting calls to the four entry members. Like the protected-object solution, the two end members, EndRead and EndWrite, both are accepted unconditionally as all the control is done in the two start members. The guards for StartRead and StartWrite preclude calls if a writer is using the resource. After entering StartWrite, a writer checks for readers, and if present, accepts only calls to EndRead until that group of readers has finished using the resource. No new readers or writers can start while the writer is accepting EndRead as it always restarts after a call is accepted. Notice, the duplicated code in the nested accept because each accept clause defines a new entry (mutex) body. While the duplicated code can be factored into a subprogram, the Ada style appears to require more of this restructuring than the μC++ style. When a writer finishes and sets wcnt to 0, Ada now makes an arbitrary choice between tasks waiting on the StartRead and StartWrite mutex queues, which maximizes the potential for staleness.

The readers/writer illustrates the need for basic FIFO service for calls to monitor/task mutex-members so that a programmer can then chose how calls are serviced. When the language/system makes arbitrary choices on behalf of a programmer, the programmer can never regain sufficient control for certain purposes. The only way to eliminate staleness in Ada is to have the readers and writers call the same entry member as these calls are serviced in FIFO order. As has been noted, changing the interface is not always an option; the interface may be given to the programmer via the specifications.

13.4.2 SR/Concurrent C++

Both SR [AOC+88] and Concurrent C++ [GR89] have tasks with external scheduling using an accept statement. However, neither language has condition variables or a requeue statement, so there is no internal scheduling or way to simulate it using external scheduling. In fact, Ada 83 did not have the requeue statement, but it was subsequently added in Ada'95. Only one feature of these two languages is discussed: extensions to the accept statement to try to ameliorate the need for requeue.

The first extension is to the when clause to allow referencing the caller's arguments through the parameters of the called mutex member, as in:

```
select
    accept mem( code : in Integer )
        when code % 2 = 0 do ...        // accept call with even code
or
    accept mem( code : in Integer )
        when code % 2 = 1 do ...        // accept call with odd code
end select;
```

Notice the placement of the when clause *after* the accept clause so the parameter names are defined. When a when clause references a parameter, selecting an accept clause now involves an implicit search of all waiting tasks on the mutex queue. This search may be large if there are many waiting tasks, and the mutex queue must be locked during the search to prevent a race condition between an appropriate task blocking on the mutex queue and the search missing it. While the mutex queue is locked, calling tasks must busy wait to acquire the lock as there is nowhere else to block. Furthermore, if no waiting task immediately satisfies the when clauses, the acceptor blocks, and each arriving call to an open member must either evaluate the when expression, or more likely, restart the acceptor task so it can because the expression could contain local variables of the task body. Both SR and Concurrent C select the task waiting the longest if there are multiple waiting tasks satisfying the when clause. This semantics is easily implemented if the mutex queue is maintained in FIFO order by arrival, as the first task with a true when expression is also the one waiting the longest.

The second extension is the addition of a selection clause to more precisely select when there are multiply true when expressions among waiting tasks rather than using order of arrival, as in:

```
select
    accept mem( code : in Integer )
        when code % 2 = 0 by -code do ...      // accept call with largest even code
or
    accept mem( code : in Integer )
        when code % 2 = 1 by code do ...       // accept call with smallest odd code
end select;
```

The by clause is calculated for each true when clause and the minimum by clause is selected. In essence, the when clause selects only those tasks that can legitimately enter at this time (cooperation) and the by clause sub-selects among these legitimate tasks. If there are multiple by expressions with the same minimum value, the task waiting the longest is selected. If the by clause appears without a when clause, it is applied to all waiting tasks on the mutex queue, as if when **true** was specified. Notice, the by clause exacerbates the execution cost of executing an accept clause because now *all* waiting tasks on the mutex queue must be examined to locate appropriate ones the minimal by expression.

In all cases, once an accept clause is selected and begins execution it must execute to completion without blocking again with respect to the called resource. Calling other resources may result in blocking, but the called resource has no mechanism (internal scheduling or requeue) to perform further blocking with respect to itself. Therefore, all cooperation must occur when selecting tasks from the mutex queues.

While these two extensions go some distance towards removing internal scheduling and/or requeue, constructing expressions for when and by clauses can be complex and there are still valid situations neither can deal with. For example, if the selection criteria involves multiple parameters, as in:

```
accept mem( code1, code2 : in Integer ) ...
```

and the selection algorithm requires multiple passes over the data, such as select the lowest even value of code1 and the highest odd value of code2 if there are multiple lowest even values. Another example is if the selection criteria involves information from other mutex queues such as the dating service where a calling girl task must search the mutex queue of the boy tasks to find a matching compatibility code. While it is conceivable for the language to provide special constructs to access and search all the mutex queues, it is often simpler to unconditionally accept a request, perform an arbitrarily complex examination of what to do next based on the information in the request, and either service or postpone depending on the selection criteria. This avoids complex selection expressions and possibly their repeated evaluation. In addition, it allows all the normal programming language constructs and data structures to be used in making the decision to postpone a request, instead of some fixed selection mechanism provided in the programming language, as in SR and Concurrent C++.

13.4.3 Modula-3/Java

Both Modula-3 and Java are object-oriented languages and both use a combination of library and language features to provide concurrency. The library features create threads and provide synchronization, and the language features provide mutual exclusion. Unfortunately, library features render any concurrent system unsound. Modula-3 does not deal with the unsoundness issue; Java deals with it by giving certain member routines special properties, which is equivalent to making these routines into language constructs.

13.4.3.1 Thread Creation and Termination

Both Modula-3 and Java use the inheritance approach for associating the thread abstraction with a user defined task type (see Section 13.2.2.1, p. 371). The abstraction for the superclass Thread for each languages is given in Figure 13.15. Only those member routines necessary to explain the basic concurrent facilities are presented. Both of these special base classes serve the same purpose as the special type uBaseTask in μC++, from which all μC++ tasks implicitly inherit.

Starting with Modula-3, notice the interface is non-object-oriented, as the routines to start a thread executing and wait for thread termination, Fork and Join, are not part of the basic thread type Closure. Instead, these routines are free routines accepting the thread type as a parameter. The non-object-oriented structure is also evident for the two kinds of locks Mutex and Condition, which are not object types, having free routines Acquire/Release and

Modula-3	Java
INTERFACE Thread; TYPE T <: REFANY; Mutex = MUTEX; Condition <: ROOT Closure = OBJECT (* task type *) METHODS apply() : REFANY; END; PROCEDURE Fork(cl : Closure) : T; PROCEDURE Join(t : T) : REFANY; PROCEDURE Acquire(m : Mutex); PROCEDURE Release(m : Mutex); PROCEDURE Wait(m : Mutex, c : Condition); PROCEDURE Signal(c : Condition); PROCEDURE Broadcast(c : Condition); PROCEDURE Self() : T; END Thread.	**class** Thread implements Runnable { **public** Thread(); **public** Thread(Runnable target); **public** Thread(String name); **public** Thread(Runnable target, String name); **public void** run(); **public** synchronized **void** start(); **public** final **void** join() throws InterruptedException; **public** wait(); **public** notify(); **public** notifyall() **public static** Thread currentThread(); **public static void** yield(); }

Figure 13.15: Thread Abstract Class

Wait/Signal/Broadcast, respectively. Overall, the name Thread defines the module interface packaging *all* the concurrency types and routines, and the actual thread type is misleadingly named Closure. As will be seen, the Modula-3 interface resembles the PThreads style of concurrency rather than an object-oriented style.

The thread type Closure is like uBaseTask in μC++, and provides any necessary variables and members needed by user subtypes, especially the apply member, which is the same as the task main in μC++, e.g.:

```
TYPE myTask = Thread.Closure OBJECT (* inheritance *)
      arg      : INTEGER;      (* communication variables *)
      result   : INTEGER;
    METHODS:
      (* "init" member, which acts as constructor *)
      (* "result" member(s) for termination synchronization *)
      (* unusual to have more members because no external scheduling *)
    OVERRIDES
      apply := myMain;         (* task body *)
END;
```

Interestingly, the apply member returns a refany (like **void** *) pointer, as for a routine started concurrently in PThreads. The difference is that all references in Modula-3 are dynamically type-checked to ensure correct usage. However, the Modula-3 authors neither discuss this return value nor show examples of its use. Instead, it is simpler to have a member that returns any necessary values from the task's global variables. It is possible to have other members, which means it is theoretically possible to construct a μC++-like task. However, in practice, the absence of an external scheduling mechanism to manage direct calls, i.e., direct-communication/rendezvous, makes it virtually impossible, as manually constructing some form of accept statement is very difficult.

The declaration of a thread in Modula-3 creates a pointer to a thread object, i.e., an object of type Closure. (All objects in Modula-3 are references.) The type of this pointer is T, which is a subtype of pointer refany, and it contains an opaque handle to the thread object, like the PThreads type pthread_t (see Section 13.2.1.1, p. 364). Therefore, the value generated by Fork and passed to Join is used internally solely by the thread implementation. Modula-3 defines assignment and equally operations for such opaque values. For example, the following shows Modula-3 task creation and termination synchronization:

```
VAR t   : mytask    := NEW(myTask).init(...);   (* create task object and initialized it *)
    th  : Thread.T  := Thread.Fork( t );        (* create thread object and start it running in "apply" *)
    a1  : REFANY    := Thread.Join( th );       (* wait for thread termination *)
    a2  : INTEGER   := t.result();              (* retrieve answer from task object *)
```

Like μC++, when the task's thread terminates, it becomes an object, which continues to exist after the termination synchronization, hence allowing the call to member result to retrieve a result in a statically type-safe way. The use of an object for a task, i.e., Closure, rather than just a routine, as in PThreads, provides this capability.

In Java, the interface is object-oriented, as the routines to perform thread start and wait, start/join, are part of the basic thread type Thread. Interestingly, the routines associated with condition locks, wait/notify/broadcast, are part of the thread type not a condition type. This issue is discussed further in the next section.

The thread type Thread is like uBaseTask in μC++, and provides any necessary variables and members needed by user subtypes of it, especially the run member, which is the same as the task main in μC++, e.g.:

```
class myTask : Thread {          // inheritance
    private int arg;             // communication variables
    private int result;
    public mytask() {...}        // task constructors
    public int result() {...}    // return result after termination synchronization
    // unusual to have more members because no external scheduling
    public void run() {...}      // task body
}
```

Returning a result on thread termination is accomplished by a member that returns any necessary values from the task's global variables. It is possible to have other members, which means it is possible, in theory, to construct a μC++-like task. However, in practice, there is no external scheduling mechanism so managing direct calls to these member, i.e., direct-communication/rendezvous, is difficult because it necessitates manually constructing some form of accept statement.

The declaration of a thread in Java creates a pointer to a thread object. (All objects in Java are references.) Java defines assignment and equally operations for these pointer values. For example, the following shows Java task creation and termination synchronization:

```
mytask th = new myTask(...);   // create task/thread object and initialized it
th.start();                    // start thread running in "run"
th.join();                     // wait for thread termination
a2 = th.result();              // retrieve answer from task object
```

Like μC++, when the task's thread terminates, it becomes an object, which continues to exist after the termination synchronization, hence allowing the call to member result to retrieve a result in a statically type-safe way.

Both Modula-3 and Java require explicit starting of a thread after the task's declaration, which is a coding convention that is potential source of error. The problem is that not starting a thread does not necessarily produce an error; a program can continue to use the task as an object, generating correct or incorrect results but having no concurrency. In both languages, it is possible to mimic starting the thread on declaration by inserting a call to Fork/start as the *last* operation of the constructor. It is crucial to strictly follow this convention, because once the thread is started, it is conceivably running in the task at the same time as the the thread executing the constructor. Therefore, there is a mutual exclusion issue with respect to completion of object initialization in the constructor and accessing the object in the apply/run member. Unfortunately, as mentioned previously, this technique does not generalize to cover inheritance (see Section 13.2.2.1, p. 374) because both the base and derived class constructors start the task's thread. Therefore, explicitly starting a task's thread is not as powerful as implicitly starting the thread after declaration is completed.

However, there are cases where a programmer may not want a task to start immediately after declaration. Can this case be handled by schemes that implicitly start a task's thread after declaration? In μC++, it is trivial to mimic this capability by creating a mutex member start and accepting it at the beginning of the task main, as in:

```
_Task T {
    void main() {
        _Accept( start );        // wait for initial call
        ...
    }
  public:
    void start() {}              // no code necessary in member body
    ...
};
```

This same behaviour can be created with internal scheduling too, by having the task's thread block on a condition variable that the start member signals. Thus, there is a weak equivalence between the two approaches because only

```
MODULE Main;
IMPORT IntBoundedBuffer, Thread, Fmt, Random, Stdio, Wr;

TYPE Producer = Thread.Closure OBJECT
    buf : IntBoundedBuffer.T;
  OVERRIDES
    apply := ProdMain;
END;

PROCEDURE ProdMain( S : Producer ) : REFANY =
VAR
    NoOfElems : INTEGER := 40;
    elem : INTEGER;
BEGIN
    FOR i := 0 TO NoOfElems DO
        elem := rand.integer(0,99);
        Wr.PutText( Stdio.stdout,
            "Producer:" & Fmt.Ref(Thread.Self()) &
            " value:" & Fmt.Int(elem) & "\n" );
        S.buf.insert( elem );
    END;
    RETURN NIL;
END ProdMain;
```

```
TYPE Consumer = Thread.Closure OBJECT
    buf : IntBoundedBuffer.T;
  OVERRIDES
    apply := ConsMain;
END;

PROCEDURE ConsMain( S : Consumer ) : REFANY =
VAR

    elem : INTEGER;
BEGIN
    LOOP   (* consumer until negative elem *)
        elem := S.buf.remove();
        Wr.PutText( Stdio.stdout,
            "Consumer:" & Fmt.Ref(Thread.Self()) &
            " value:" & Fmt.Int(elem) & "\n" );
      IF elem = -1 THEN EXIT END;
    END;
    RETURN NIL;
END ConsMain;
```

```
CONST
    NoOfCons  : CARDINAL = 3; NoOfProds : CARDINAL = 4;
VAR
    buf   : IntBoundedBuffer.T := NEW(IntBoundedBuffer.T).init(20);
    cons  : ARRAY[1..NoOfCons] OF Thread.T;
    prods : ARRAY[1..NoOfProds] OF Thread.T;
    rand  : Random.T := NEW(Random.Default).init();
BEGIN
    FOR i := 1 TO NoOfCons DO cons[i] := Thread.Fork( NEW( Consumer, buf := buf ) ); END;
    FOR i := 1 TO NoOfProds DO prods[i] := Thread.Fork( NEW( Producer, buf := buf ) ); END;
    FOR i := 1 TO NoOfProds DO EVAL Thread.Join( prods[i] ); END;
    FOR i := 1 TO NoOfCons DO buf.insert( -1 ); END;   (* terminate each consumer *)
    FOR i := 1 TO NoOfCons DO EVAL Thread.Join( cons[i] ); END;
END Main.
```

Figure 13.16: Modula-3: Producer/Consumer Creation/Termination

one can mimic the other trivially.

As mentioned, both Modula-3 and Java use a combination of library and language features to provide concurrency. The following situations present problems for the library features with respect to soundness:

```
t.init();            // modify task object
t.start();           // start thread
t.join();            // termination synchronization
i = t.result();      // obtain result
```

A compiler could interchanges either the first two lines or the last two lines, because start or join may not access any variables accessed by init or result. Similar problems can occur with global variables accessed before or after start/join, as these global accesses can be moved a substantial distance with respect to the calls to start/join. Modula-3 makes no mention of any special requirements in the language to deal with these situations. Java defines both Thread.start and Thread.join as special routines so the compiler only performs appropriate optimizations.

Figures 13.16 and 13.17 illustrate the Modula-3 and Java equivalent for starting and terminating threads in a bounded buffer example (see the right hand side of Figure 7.11, p. 203). The bounded buffer is presented shortly.

```
class Producer extends Thread {
    private BoundedBuffer buf;
    public Producer( BoundedBuffer bufp ) {
        buf = bufp;
    }
    public void run() {
        int NoOfElems = 40;
        for ( int i = 1; i <= NoOfElems; i += 1 ) {
            int elem = (int)(Math.random() * 100);
            System.out.println( "Producer:" + this +
                " value:" + elem );
            buf.insert( elem );
        }
    }
}
```

```
class Consumer extends Thread {
    private BoundedBuffer buf;
    public Consumer( BoundedBuffer bufp ) {
        buf = bufp;
    }
    public void run() {

        for ( ;; ) {
            int elem = buf.remove();
            System.out.println( "Consumer:" + this +
                " value:" + elem );
          if ( elem == -1 ) break;
        }
    }
}
```

```
class BB {
    public static void main( String[ ] args ) {
        int NoOfCons = 3, NoOfProds = 4;
        BoundedBuffer buf = new BoundedBuffer( 10 );
        Consumer cons[ ]  = new Consumer[NoOfCons];
        Producer prods[ ] = new Producer[NoOfProds];
        for ( int i = 0; i < NoOfCons; i += 1 ) { cons[i] = new Consumer( buf ); cons[i].start(); }
        for ( int i = 0; i < NoOfProds; i += 1 ) { prods[i] = new Producer( buf ); prods[i].start(); }
        for ( int i = 0; i < NoOfProds; i += 1 ) { try { prods[i].join(); } catch( InterruptedException ex ) {} }
        for ( int i = 0; i < NoOfCons; i += 1 ) { buf.insert( -1 ); }  // terminate each consumer
        for ( int i = 0; i < NoOfCons; i += 1 ) { try { cons[i].join(); } catch( InterruptedException ex ) {} }
        System.out.println( "successful completion" );
    }
}
```

Figure 13.17: Java: Producer/Consumer Creation/Termination

13.4.3.2 Thread Synchronization and Mutual Exclusion

Modula-3 is like PThreads with respect to synchronization and mutual exclusion, i.e., the primitives are low-level and the programmer is involved in many details, e.g., there are no monitors in the language. Java's synchronization and mutual exclusion are provided through a monitor construct, but the monitors have peculiar semantics and restrictions, making programming unnecessarily complex and difficult. Both languages essentially provide support for only internal scheduling; external scheduling could be mimicked but only in a very weak way.

Like PThreads, Modula-3 provides two kinds of locks, mutex and condition, through which all synchronization and mutual exclusion is constructed. The Modula-3 mutex lock is like a binary semaphore with the additional notion of ownership but no recursive acquisition, so the thread that locks the mutex must unlock it only once. As a result, a mutex lock can *only* be used for mutual exclusion, because synchronization requires the locking thread be different from the unlocking one. The Modula-3 mutex lock has the following interface:

```
MUTEX = OBJECT ... END; (* built-in opaque type defined by implementation *)
PROCEDURE Acquire( m : Mutex );
PROCEDURE Release( m : Mutex );
```

The type of a mutex lock is MUTEX and is renamed to Mutex within the Thread interface, so either name is appropriate and identical; locks declared from this type are automatically initialized to open. There is also a single language statement that works with mutex locks:

```
LOCK m DO statement END; (* m is a mutex variable *)
```

which is like the critical region statement in Section 9.1, p. 250, where m is a lock variable rather than a shared variable. This statement serves two purposes:

1. indicate to the compiler boundaries for code optimizations involving code movement to ensure soundness for

concurrent programs

2. provide a language statement to implicitly acquire the lock on entry to the block and release the lock on exit from the block regardless of whether the block terminates normally or with an exception.

Therefore, the explicit use of routines Acquire and Release are discouraged because it can violate one or both of these requirements.

The Modula-3 condition lock is like a condition variable, creating a queue object on which threads block and unblock; however, there is no monitor construct to simplify and ensure correct usage of condition locks. Instead, a condition lock is dependent on the mutex lock for its functionality, and collectively these two kinds of locks can be used to build a monitor, providing both synchronization and mutual exclusion. As for a condition variable, a condition lock can *only* be used for synchronization, because the wait operation always blocks. The Modula-3 condition lock has the following interface:

```
Condition = OBJECT ... END; (* built-in opaque type defined by implementation *)
PROCEDURE Wait( M : Mutex, c : Condition );
PROCEDURE Signal( c : Condition );
PROCEDURE Broadcast( c : Condition );
```

The type of a condition lock is Condition; locks declared from this type are automatically initialized. The routine Wait blocks a thread on a condition variable, and routines Signal and Broadcast unblock threads from a condition variable. As always, operations on the condition queue must be done atomically, so these operations must be protected with a lock; hence, the need for the companion mutex lock to provide mutual exclusion. For example, given the following mutex and condition:

```
MUTEX m;
Condition c;
```

it is straightforward to protect the signal operation on a condition lock by:

```
LOCK m DO
    Signal( c );
END;
```

but protecting the wait operation is problematic, as in:

```
LOCK m DO
    Wait( c );        (* not Modula-3 *)
END;
```

because after the mutex lock is acquired, the thread blocks and cannot unlock the mutex. As a result, no other thread can acquire the mutex to access the condition lock; hence, there is a synchronization deadlock. To solve this problem, the wait operation of Modula-3 is passed the mutex and it atomically unlocks the mutex and blocks on the queue of the condition lock, as in:

```
LOCK m DO
    Wait( m, c );     (* atomic block and release lock *)
END;
```

Wait has the additional semantics of re-acquiring the argument mutex lock before returning, as if a call to Acquire had been made after unblocking from the condition lock in a LOCK statement. This semantics slightly reduces the complexity of constructing monitors but allows calling threads to barge into a monitor between a signaller leaving it and the signalled task restarting within it; hence, unless additional work is done, a monitor has the no-priority property (see Section 9.13.1, p. 296). Signal unblocks at least one waiting thread from the queue of a condition lock; it may unblock more than one. There is no compelling justification for this latter semantics and it only makes constructing a monitor more difficult. Since signal does not change the mutex lock, the signaller retains control of it (if acquired), so a monitor has the non-blocking property. As always, a signal on an empty condition lock is lost, i.e., there is no counter to remember signals before waits. Broadcast unblocks all waiting thread from the queue of a condition lock, and is necessary in certain cases because there is no operation to check if the queue of a condition lock is empty.

There is no compiler support to ensure a programmer uses mutex and condition locks correctly or builds a monitor correctly. Like PThreads, common errors are to forget to acquire mutual exclusion before signalling or waiting and/or using the wrong mutex lock when signalling and waiting. As well, a programmer must manually perform the following conventions when building a monitor using Modula-3:

1. each monitor must have a single mutex lock for the monitor lock
2. there must be the equivalent of a constructor, i.e., an init member, to allocate the monitor lock and any condition variables because both are objects
3. each mutex member, excluding any initialization members, must start by acquiring the monitor lock, and the monitor lock must be released by all return paths from a member
4. when waiting for an event, it is necessary to recheck for the event upon restarting because of barging
5. global monitor (object) variables, versus local mutex-member variables, cannot be returned directly unless the return is within a LOCK statement

Because a Modula-3 mutex lock can only be acquired once, one mutex member cannot call another, unlike μC++; if it does, synchronization deadlock occurs.

Java provides an implicit mutex lock and condition variable in each a monitor (mutex) object, through which all synchronization and mutual exclusion is constructed. The hidden Java mutex lock is like a binary semaphore, which can be recursively acquired by the thread currently locking it. The following language constructs ensure that the thread that locks the lock also unlocks it. A monitor object is created when at least of one of its members is specified as synchronized or one of the members contains a synchronized statement, as in:

```
class mon {           // monitor object
    public synchronized int mem1() {...}
    public int mem2() {
        synchronized( this ) {...}
    }
}
```

The synchronized qualifier is identical to the **_Mutex** qualifier in μC++, except it can only appear on a member routine not on the class itself; therefore, it must be repeated for most public members. Also, the word synchronized incorrectly describes the effect it produces, as it generates mutual exclusion not synchronization. The synchronized statement is like the LOCK statement in Modula-3. However, instead of specifying a mutex lock, a mutex object is given, as the mutex lock is abstracted within the mutex object. Normally, the object specified in a synchronized statement is the current monitor object, i.e., **this**, but it is possible to specify a different monitor object. In general, specifying any object other than **this** seems like poor programming practice, as it implies a monitor object is no longer in control of its mutual exclusion property. Like the LOCK statement in Modula-3, the synchronized qualifier and statement serve two purposes:

1. indicate to the compiler boundaries for code optimizations involving code movement to ensure soundness for concurrent programs

2. provide a language statement to implicitly acquire the lock on entry to the routine/block and release the lock on exit from the routine/block regardless of whether the block terminates normally or with an exception.

In general, the compiler support for monitor synchronized routines should be used rather than explicitly building a monitor using the synchronized statement. In other words, let the compiler do the job it is designed to do.

The Java condition lock is like a condition variable, creating a queue object on which threads block and unblock; however, there is only *one* condition variable implicitly created for each monitor object (like the mutex lock). There is no compelling justification for this latter semantics and it only makes constructing a monitor more difficult. Because there is only one condition variable, it can be anonymous, and implicitly known to the operations that use it. The Java condition lock has the following interface:

```
class condition { ... }    // built-in opaque type defined in each mutex object
condition anon;            // built-in declaration in each mutex object
public wait();             // operations manipulate "anon"
public notify();
public notifyall()
```

The routine wait blocks a thread on a condition variable, and routines notify and notifyall unblock threads from a condition variable. As always, operations on the condition queue must be done atomically, so these operations must be protected within a synchronized member or statement. Java atomically releases the monitor lock as part of the call to wait, as in μC++.

wait has the additional semantics of re-acquiring the monitor mutex lock before returning, as if a call to acquire the monitor lock had been made after unblocking from the implicit condition variable. This semantics allows calling threads to barge into a monitor between a signaller leaving it and the signalled task restarting within it; hence, unless additional work is done, a monitor has the no-priority property. notify unblocks only one waiting thread from the implicit condition lock; however, threads may be restarted in any order from the condition queue. There is no compelling justification for this latter semantics and it only makes constructing a monitor more difficult. A signaller retains control of the monitor, so a monitor has the non-blocking property. As always, a signal on an empty condition lock is lost, i.e., there is no counter to remember signals before waits. notifyall unblocks all waiting thread from the queue of a condition lock, and is necessary in certain cases because there is no operation to check if the queue of the implicit condition variable is empty.

Figure 13.18 illustrates the Modula-3 and Java equivalent for the internal-scheduling bounded-buffer monitor in Figure 9.6, p. 262. The Modula-3 version is generic in the type of the elements stored in the buffer; Java does not support generic definitions. In the Modula-3 program, the init routine acts as the constructor. Neither language has destructors because both support garbage collection.

Routines query, insert and remove are equivalent to those in the bounded buffer monitor. Routine query does not acquire and release the monitor lock because it is a non-mutex member; routines insert and remove do acquire and release the monitor lock because they are mutex members, through the LOCK statement and synchronized qualifier, respectively. The signals in insert and remove are non-blocking (see Section 9.11.3.4, p. 289) because the monitor lock is still held by the thread performing the signalling. The waits in insert and remove are no-priority (see Section 9.11.3.4, p. 289), because after being signalled, a thread implicitly re-acquires the monitor lock as part of returning from call to wait, which means it competes with calling threads to acquire this mutex lock. As a result, calling threads can barge ahead of threads signalled from internal condition locks, which means the state of the monitor at the time of the signal may have changed when the waiting thread subsequently restarts in the monitor (see Section 9.13.1, p. 296). To deal with this problem, each wait is enclosed in a **while** loop to recheck if the event has occurred, which is a busy waiting because there is no bound on service (see Section 9.13.1, p. 296).

Finally, solutions to the readers and writer problem written for Modula-3 and Java are presented, which have no starvation and no staleness. The Modula-3 solution is essentially identical in form to the PThreads solution in Figure 13.6, p. 373 so no code is presented. In both systems the signalling semantics may wake up more than one thread and there is a barging issue because priority is not given to signalled threads. The exact same techniques are used in Modula-3 to deal with these issues, i.e., using private conditions so only one thread is blocked on each condition, and using the testing trick with rwdelay to prevent barging.

The Java solution is more complex because of other issues. While Java does not have the problem of waking up more than one thread on a signal, it does have the restriction of only one condition variable per object, signalled threads are not necessarily restarted in FIFO order, and there is a barging issue because priority is not given to signalled threads. Like the PThreads solution, appropriate testing of rwdelay can prevent barging. As well, the signalling problem can be handled using a queue of private condition locks to ensure tasks are signalled in FIFO order. However, managing the queue of private conditions is complex because each object on the queue is more than just a condition queue, it is in fact a monitor with an implicit monitor lock. Therefore, it is easy to cause the nested monitor problem (see Section 9.9, p. 282), as in:

```
private synchronized void StartRead() {
    if ( wcnt > 0 || rwdelay > 0 ) {          // writer active or waiting tasks ?
        RWnode r = new RWnode( READER );      // remember kind of task
        rw.push( r ); rwdelay += 1;
        r.wait();                             // nested monitor problem
    . . .
```

Because r is itself a monitor, the thread blocks holding two monitor locks, the one for the readers/writer monitor and the one for the node r. The reason is that r.wait() atomically blocks the thread and releases the mutual-exclusion lock but only for the RWnode monitor, not the mutual-exclusion lock for the reader/writer monitor. Now it is impossible for any task to enter the reader/writer monitor to signal the thread blocked in r, so there is a mutual-exclusion deadlock.

A technique to deal with this problem is to first release the readers/writer monitor lock and then block on the node. To do this requires using the synchronized statement, like the Modula-3 LOCK statement, as in:

Modula-3

```
GENERIC MODULE BoundedBuffer( Elem );
IMPORT Thread;
REVEAL
    T = Public BRANDED OBJECT
        front, back, count : CARDINAL := 0;
        elements     : REF ARRAY OF Elem.T;
        mutex        : Thread.Mutex;
        full, empty  : Thread.Condition;
      OVERRIDES
        init     := Init;
        query    := Query;
        insert   := Insert;
        remove   := Remove;
    END;
PROCEDURE Init( S : T; size : CARDINAL ) : T =
BEGIN
    S.elements  := NEW(REF ARRAY OF Elem.T, size);
    S.mutex     := NEW(Thread.Mutex);
    S.full      := NEW(Thread.Condition);
    S.empty     := NEW(Thread.Condition);
    RETURN S;
END Init;

PROCEDURE Query( S : T ) : CARDINAL =
BEGIN
    RETURN S.count;
END Query;

PROCEDURE Insert( S : T; READONLY elem : Elem.T ) =
BEGIN
    LOCK S.mutex DO
        WHILE S.count = NUMBER(S.elements^) DO
            Thread.Wait( S.mutex, S.empty ); END;
        S.elements[S.back] := elem;
        S.back := (S.back + 1) MOD NUMBER(S.elements^);
        S.count := S.count + 1;
        Thread.Signal( S.full );
    END;
END Insert;

PROCEDURE Remove( S : T ) : Elem.T =
VAR R : Elem.T;
BEGIN
    LOCK S.mutex DO
        WHILE S.count = 0 DO
            Thread.Wait( S.mutex, S.full ); END;
        R := S.elements[S.front];
        S.front := (S.front + 1) MOD NUMBER(S.elements^);
        S.count := S.count - 1;
        Thread.Signal( S.empty );
    END;
    RETURN R;
END Remove;

BEGIN
END BoundedBuffer.
```

Java

```
class BoundedBuffer {

    private int front = 0, back = 0, count = 0;
    private int[] Elements;
    private int NoOfElems;

    public BoundedBuffer( int size ) {

        NoOfElems = size;
        Elements = new int[NoOfElems];
    }

    public int query() {

        return count;
    }

    public synchronized void insert( int elem ) {

        while ( count == NoOfElems )
            try { wait();
            } catch( InterruptedException ex ) {};
        Elements[back] = elem;
        back = ( back + 1 ) % NoOfElems;
        count += 1;
        notify();

    }

    public synchronized int remove() {
        int elem;

        while ( count == 0 )
            try { wait();
            } catch( InterruptedException ex ) {};
        elem = Elements[front];
        front = ( front + 1 ) % NoOfElems;
        count -= 1;
        notify();

        return elem;
    }

}
```

Figure 13.18: Modula-3/Java: Bounded Buffer

```
private void StartRead() {
    RWnode r = null;
    synchronized( this ) {
        if ( wcnt > 0 || rwdelay > 0 ) {          // writer active or waiting tasks ?
            r = new RWnode( READER );             // remember kind of task
            rw.push( r ); rwdelay += 1;
        } else
            rcnt += 1;                            // do not have to block
    }                                             // release R/W monitor lock
    if ( r != null ) {                            // cannot proceed ? => block
        r.wait();                                 // wait on node monitor
        synchronized( this ) {                    // re-acquire R/W monitor lock
            . . .
        }
    }
```

Unfortunately, there is now a race condition between signalling and waiting on the node. The problem occurs if a task is interrupted after it releases the readers/writer lock but before it waits on node r. Another reader task, for example, might take the node off the list and signal it, and that signal is lost. When the interrupted task is restarted, it blocks having missed its signal, and hence, is synchronization deadlocked. To deal with this problem, the node is made into a simple binary semaphore, which remembers if it is signalled; now if a wait occurs after a signal, the wait is ignored.

Figure 13.19 shows the readers and writer problem written using Java, which has no starvation and no staleness. The basic solution is based on the PThreads version, with appropriate changes made to eliminate the nested monitor problem. The type RWnode has an additional field beenNotified along with two synchronized routines: Notify and Wait. For a thread to signal or wait on nodes of type RWnode, the routines Notify and Wait are called, respectively. The Notify routine acquired mutual exclusion on the node and marks the node as being notified, as well as signalling the condition. If there is no thread blocked on the condition, the signal is lost but the flag is set. The Wait routine acquired mutual exclusion on the node, and only waits if the notification flag is false. Since a node is only used once in this solution, it is unnecessary to reset the notification flag after waiting.

All other code changes from the PThreads solution are just transformations to deal with the nested monitor problem. The readers/writer monitor lock is acquired to safely examine the monitor variables. When a thread needs to block, it creates a node and chains it onto the list of private conditions. Then, the readers/writer monitor lock is released and a check is made to determine if the thread has to block. If so, the thread calls Wait for the node and either blocks or continues, depending on the value of beenNotified. When a waiting thread restarts, it re-acquires the readers/writer monitor lock, if necessary, and continues as for the PThreads solution. Notice, all calls to Notify are performed outside the synchronized statements for ReadersWriter. This placement is possible because the signalled thread is blocked on a separate monitor in the queue node, and it increases concurrency because the unblocked task immediately executes a synchronized statement for ReadersWriter, meaning it does not have to wait for the signaller thread to release monitor ReadersWriter.

Finally, both Modula-3 and Java have the ability to interrupt a task blocked on a condition variable. In general, this facility is expensive and is used for exceptional situations, such as thread cancellation. However, the interrupt mechanism, in conjunction with exceptions, can be used to explicitly signal an arbitrary task from a condition variable. Hence, it is possible to explicitly build a list of task identifiers blocked on a condition, and subsequently unblock them in any order by selecting a task in the list and interrupting it. This approach is not recommended because it confuses signalling with interrupts, has a high runtime cost, and largely precludes using interrupts for situations like cancellation.

13.5 Concurrent Models

Concurrency models are language-independent approaches for writing concurrent programs based on some encompassing design philosophy. Models are largely language independent and can be introduced into existing sequential languages or incorporated from the start into new languages. In general, models adopt a single approach/methodology that can be used solve most problems, but at the cost of solving some problems well and others poorly.

```
class ReadersWriter {
    private static final int READER = 0, WRITER = 1;
    private static class RWnode {
        public int kind;
        public boolean beenNotified;
        public RWnode next;

        public RWnode( int kindp ) {
            kind = kindp;
            beenNotified = false;
            next = null;
        }
        public synchronized void Notify() {
            beenNotified = true;
            notify();
        }
        public synchronized void Wait() {
            if ( ! beenNotified ) // race condition ?
                try { wait();
                } catch(InterruptedException ex) {}
        }
    }
    private int rcnt = 0, wcnt = 0, rwdelay = 0;
    private RWqueue rw = new RWqueue();

    private void StartRead() {
        RWnode r = null;
        synchronized ( this ) {
            if ( wcnt > 0 || rwdelay > 0 ) {
                r = new RWnode( READER );
                rw.push( r ); // prepare to block
                rwdelay += 1;
            } else
                rcnt += 1; // no block
        }
        if ( r != null ) { // must block ?
            r.Wait();
            RWnode n = null;
            synchronized ( this ) {
                rcnt += 1;
                rw.pop(); // cleanup after waiting
                rwdelay -= 1;
                if ( rwdelay > 0 && // more readers ?
                        rw.front().kind == READER )
                    n = rw.front();
            }
            if ( n != null ) n.Notify(); // more tasks ?
        }
    }
    private void EndRead() {
        RWnode n = null;
        synchronized ( this ) {
            rcnt -= 1;
            if ( rcnt == 0 && rwdelay > 0 )
                n = rw.front();
        }
        if ( n != null ) n.Notify(); // more tasks ?
    }
    private void StartWrite() {
        RWnode w = null;
        synchronized ( this ) {
            if ( rcnt > 0 || wcnt > 0 || rwdelay > 0 ) {
                w = new RWnode( WRITER );
                rw.push( w ); // prepare to block
                rwdelay += 1;
            } else
                wcnt += 1; // no block
        }
        if ( w != null ) { // must block ?
            w.Wait();
            synchronized ( this ) {
                wcnt += 1;
                rw.pop(); // cleanup after waiting
                rwdelay -= 1;
            }
        }
    }
    private void EndWrite() {
        RWnode n = null;
        synchronized ( this ) {
            wcnt -= 1;
            if ( rwdelay > 0 )
                n = rw.front();
        }
        if ( n != null ) n.Notify(); // more tasks ?
    }
    public void Read() {
        StartRead();
        // read
        EndRead();
    }
    public void Write() {
        StartWrite();
        // write
        EndWrite();
    }
}
```

Figure 13.19: Java: Readers and Writer

13.5.1 Actor Model

The Actor model is essentially a message passing system with a complex abstraction for receiving messages. The Actor model defines an Actor as a message queue containing all the messages sent to it. Associated with each Actor is a list of behaviours, which are threads that receive consecutive communications from the Actor message queue. A behaviour is restricted to receiving a single message from its actor message queue but has no restriction on the number of sends to other actors. This restriction allows a behaviour to send a message to its underlying actor without deadlocking, possibly even recursively if the next behaviour to receive the message is the same as the one that sent it. By our definition an Actor is a task as there is always one behaviour (thread) active in the Actor.

The Actor model can be implemented in a number of ways, e.g. two implementations are given in [Agh86, p. 38-43]. While the model requires that Actors be objects, these objects could be implemented as package- or class-based objects. Furthermore, there is nothing in the model that precludes building a statically type-safe implementation of the Actor model. The only aspect about the Actor model that affects typing is that Actors have a single queue of messages. If the type of the Actor specifies the type of the messages that can be sent to it, and the messages are typed, the behaviours can use the same scheme used in direct message-passing systems, that is, use a dynamic check and a type discriminator. In this model of Actors, incorrect messages cannot be sent because the Actors type specifies what type of messages can be sent.

13.5.2 Linda Model

The Linda model [CG89] is based on the concept of a **tuple space**, which is a multiset of tuples, i.e., duplicates are allowed, that is accessed associatively by threads for synchronization and communication. Threads are created in the Linda model through the tuple space, and communicate by adding, reading and removing data from the tuple space. It is a “tuple” space because elements within the multiset are grouped into tuples, e.g.:

```
( 1 )                 // 1 element tuple of int
( 1.0, 2.0, 3.0 )     // 3 element tuple of double
( "Peter", 1, 2 )     // 3 element tuple of char [], and 2 integers
( 'M', 5.5 )          // 2 element tuple of char, double
```

A tuple is accessed atomically and by content, where content is based on the number of tuple elements, the data types of the elements, and possibly their values. Often there is only one tuple space implicitly known to all threads; hence, operations on the tuple space do not have to name it. Work has been done for multiple tuple spaces, which requires explicit tuple names. Because of its simplicity, the Linda model can be incorporated into many programming languages and systems. Usually the language in which the Linda model is embedded is used as a prefix to denote the particular kind of Linda, as in Ada-Linda, C-Linda, etc.

At least 4 operations are available for accessing the tuple space:

1. The read operation reads a tuple from the tuple space. To indicate which kind of tuple to read, a tuple pattern is specified. A pattern indicates both the number of elements in the matching tuple and the keys which must match both in type and value with corresponding fields of a tuple in the tuple space, e.g.:

   ```
   read( 'M', 34, 5.5 );
   ```

   blocks until a matching 3 field tuple with values ('M', 34, 5.5) appears in the tuple space. A variable can be used instead of a constant, e.g.:

   ```
   int i = 34;
   read( 'M', i, 5.5 );
   ```

   Such a read can be used for synchronizing with the thread generating the particular tuple, i.e., the reader cannot proceed until the write has occurred.

   A general form of pattern matching is possible using a type rather than a value, e.g.:

   ```
   int i;
   read( 'M', ?i, 5.5 );
   ```

   blocks until a matching 3 field tuple with values ('M', **int**, 5.5) appears in the tuple space (i is of type **int**), where **int** matches any integer value. After a match is found, the value for any type match is copied from the tuple space into the corresponding variable in the argument list. In this example, the value of the type match is copied into the variable i after a match is found.

2. The in operation is identical to the read operation, except it removes the matching tuple from the tuple space so it cannot participate in further associative searching.

3. The out operation writes a tuple to the tuple space, e.g.:

```
out( 'M', 34, 5.5 );
```

adds the tuple ('M', 34, 5.5) to the tuple space. The out operation is non-blocking. A variable can be used instead of a constant, e.g.:

```
int i = 34;
out( 'M', i, 5.5 );
```

4. The eval operation is identical to the out operation, except any routine argument is implicitly executed with its own thread, e.g.:

```
int f( int, bool );
int i = 34;
eval( 'M', f( i, true ), 5.5 );
```

adds the tuple ('M', **fork** f(i, **true**), 5.5) to the tuple space, where **fork** starts a new thread executing routine f. In contrast, out evaulates all arguments sequentially before the tuple is added to the tuple space. Subsequent reading of this tuple, e.g.:

```
read( 'M', ?i, 5.5 );
```

blocks until both the tuple is available *and* the thread executing argument f has terminated.

The operation names in Linda are from the perspective of the tuple space rather than the program, e.g., out moves data out of the tuple space into the program. Some Linda systems also provide non-blocking versions of in and read: inp and readp. If a matching tuple is not found immediately, control returns with a boolean result indicating failure to obtain data.

It is possible to create a binary semaphore to provide mutual exclusion or synchronization by appropriately inserting and removing an element from the tuple space, as in:

Mutual Exclusion	Synchronization	
	Task$_1$	Task$_2$
in("semaphore");		
// critical section	in("semaphore");	S1
out("semaphore");	S2	out("semaphore");

For mutual exclusion, the semaphore element must be initially inserted into the tuple space, which corresponds to setting the semaphore to open (1). Tasks then race to remove the semaphore element from the tuple space to enter the critical section. On exit from the critical section, the semaphore element is written back into the tuple space to reset it. The atomicity on operations provided by the tuple space provides an N-task solution for mutual exclusion. Hopefully, there is some bound on access to the element by multiple tasks or starvation can result. For synchronization, the semaphore element must not be inserted into the tuple space, which corresponds to setting the semaphore to closed (0). The task that writes the element into the tuple space marks the completion of an operation, which the other task detects by reading the element. The reading task only blocks if the element is not yet written.

Figure 13.20 illustrates the Linda equivalent for the semaphore bounded-buffer monitor in Figure 7.11, p. 203. The tuple space is already a buffer, and handles the unbounded-buffer case since a producer can insert elements into the tuple space and never blocks (but could block if the tuple space fills). A consumer implicitly blocks if there is no element in the tuple space when it attempts to remove one. To simulate a bounded buffer, it is necessary to block the producer when the buffer is conceptually full, e.g., to prevent producers from filling the tuple space. Synchronization blocking for the producers is accomplished by simulating a counting semaphore. The main program inserts N buffer elements into the tuple space. Each producer removes a buffer element before inserting an element into the tuple space; when there are no elements, the producer blocks. An element is inserted back into the tuple space by the consumer

```
int producer( int id ) {
    const int NoOfElems = rand() % 20;
    int elem, i, size;

    for ( i = 1; i <= NoOfElems; i += 1 ) {          // produce elements
        elem = rand() % 100 + 1;                      // produce a random number
        in( "bufferSem" );                            // wait for empty buffer slot
        out( "buffer", elem );                        // insert element into tuple space
    }
    return id;
}
int consumer( int id ) {
    int elem;

    for ( ;; ) {
        in( "buffer", ?elem );                        // remove element from tuple space
        out( "bufferSem" );                           // indicate empty buffer slot
      if ( elem == -1 ) break;                        // consume until a negative element
        // consume element
    }
    return id;
}
int main() {
    const int bufSize = 5, NoOfCons = 3, NoOfProds = 4;
    int i, id;

    for ( i = 0; i < bufSize; i += 1 ) {              // number of buffer elements
        out( "bufferSem" );
    }
    for ( i = 0; i < NoOfCons; i += 1 ) {             // create consumers
        eval( "consumer", consumer( i ) );
    }
    for ( i = 0; i < NoOfProds; i += 1 ) {            // create producers
        eval( "producer", producer( i ) );
    }
    for ( i = 0; i < NoOfProds; i += 1 ) {            // wait for producer to terminate
        read( "producer", ?id );
    }
    for ( i = 0; i < NoOfCons; i += 1 ) {             // terminate each consumer
        out( "buffer", -1 );
    }
    for ( i = 0; i < NoOfCons; i += 1 ) {             // wait for consumers to terminate
        read( "consumer", ?id );
    }
}
```

Figure 13.20: Linda: Producer/Consumer Bounded Buffer

only after an element has been removed from the tuple space, indicating an empty buffer slot. This solution handles multiple consumers and producers because insertion and removal of elements from the tuple space is atomic.

Figure 13.21 illustrates the Linda equivalent for the semaphore readers and writer solution in Figure 7.20, p. 217 using C++-Linda. This solution uses a private semaphore to deal with the need to atomically block and release the entry lock. To ensure FIFO service, a queue indicating the kind of waiting task, i.e., reader or writer, is maintained. To simulate the entry semaphore, the name `"entry"` *plus* the address of the ReadersWriter object is used. The object address allows multiple entry semaphores to occur in the tuple space simultaneously. Similarly, to uniquely name the private semaphores, each private semaphore has the name `"privSem"` *plus* the address of the list node holding the kind of task. As in the semaphore solution, an interrupt between tuple operations `in( "entry", this )` and `out( "privSem", &r )` is not a problem because the blocking task adds its node *before* releasing the entry lock. Any subsequent release of the private lock is remembered in the tuple space until the blocking task can remove it.

As can be seen from the Linda example programs, the facilities provided are at the level of threads and locks (see Section 13.2, p. 362). Furthermore, there is essentially no abstraction or encapsulation facilities in the tuple space. As well, sharing the tuple space with multiple applications can easily result in problems with incorrect retrieval unless tuples are tagged with unique values; some work has been done with multiple tuple spaces. Notice also that the Linda primitives have a variable number of parameters, which most languages do not support, and depend on the type system of the language, which varies greatly among languages. There is also a dichotomy between the kinds of data structures in the language and those allowed in the tuple space, e.g., pointers. Therefore, the Linda primitives must be constructs in the programming language in which they are embedded not merely library routines. There are also efficiency considerations for the associative tuple-search since each removal from the tuple space checks both types and values. Linda is dynamically type-safe as a side-effect of the selection from the tuple-space. What is unclear about the Linda model is the canonical form for the type of a tuple, particularly if tuples are allowed to come from different languages with potentially incompatible type systems.

13.6 Summary

Simula coroutines are too general because of direct activation in the Call and Resume statements, making it easy to make mistakes. However, coroutines are a crucial language feature, which are under-supported in modern programming-languages. Furthermore, coroutines must be integrated with other high-level language-features such as exception handling and concurrency. PThreads concurrency is neither object-oriented nor high-level, and hence, there is little help from the language in writing a concurrent program. The low-level concurrency features and type-unsafe communication conspire to make writing a concurrent program difficult and error prone. While it is possible to mimic high-level concurrency constructions, e.g., monitor, there is only a weak equivalence because of the complex coding-conventions that must be followed. Message passing is largely an operating system mechanism to transfer data among computers. The lack of static type-safety and the shift from standard routine call (arguments/parameters) makes message passing an unacceptable technique in most modern programming languages. Ada design suffers from two designs in two different time periods, which are not unified due to backwards compatibility issues. While the Ada design is high-level, there is a strange mixture of non-orthogonal features coupled with programming restrictions imposed by using requeue for internal scheduling, guards for protected-objects, non-deterministic selection from entry queues, and trying to allow many forms of implementation. Modula-3 concurrency is heavily based on the PThreads design. Moreover, this design is not object-oriented, which seems peculiar for an object-oriented programming language, nor is it high-level, making the programmer deal with many low-level technical issues, which results in errors. Java concurrency is object-oriented and high-level but it over-simplifies concurrent programming to the point where it is difficult to solve even medium-difficult problems. The design decision to make all objects monitors with a single condition queue per monitor and no-priority semantics for signalling significantly complicates writing concurrent programs. General concurrency models, such as actors and tuple spaces, attempt to oversimplify the complexity of concurrent programming. Concurrency is inherently complex and will always require a reasonable level of sophisticated language-constructs to deal with the complexity.

13.7 Questions

1.

```
class ReadersWriter {
    enum RW { READER, WRITER };                      // kinds of threads
    struct RWnode {
        RW kind;                                     // kind of thread
        RWnode( RW kind ) : kind(kind) {}
    };
    queue<RWnode *> rw;                              // queue of RWnodes
    int rcnt, wcnt, rwdelay;

    void StartRead() {
        out( "entry", this );
        if ( wcnt > 0 || rwdelay > 0 ) {
            RWnode r( READER );
            rw.push( &r ); rwdelay += 1;             // remember kind of thread
            in( "entry", this ); out( "privSem", &r );
            rw.pop(); rwdelay -= 1;                  // remove waiting task from condition list
        }
        rcnt += 1;
        if ( rwdelay > 0 && rw.front()->kind == READER )
            in( "privSem", rw.front() );
        else
            in( "entry", this );
    }
    void EndRead() {
        out( "entry", this );
        rcnt -= 1;
        if ( rcnt == 0 && rwdelay > 0 )              // last reader ?
            in( "privSem", rw.front() );
        else
            in( "entry", this );
    }
    void StartWrite() {
        out( "entry", this );
        if ( rcnt > 0 || wcnt > 0 ) {
            RWnode w( WRITER );
            rw.push( &w ); rwdelay += 1;             // remember kind of thread
            in( "entry", this ); out( "privSem", &w );
            rw.pop(); rwdelay -= 1;                  // remove waiting task from condition list
        }
        wcnt += 1;
        in( "entry", this );
    }
    void EndWrite() {
        out( "entry", this );
        wcnt -= 1;
        if ( rwdelay > 0 )                           // anyone waiting ?
            in( "privSem", rw.front() );
        else
            in( "entry", this );
    }
  public:
    ReadersWriter() : rcnt(0), wcnt(0), rwdelay(0) {
        in( "entry", this );
    }
    . . .
};
```

Figure 13.21: Linda: Readers and Writer

Chapter 14

Real-Time Programming

A non-real-time program (i.e., normal programming) is one in which correctness depends only on the value domain, i.e., the program must produce the correct result in a reasonable amount of time, where the time requirement is from the human perspective (e.g., seconds, minutes, hours). A **real-time program** is one in which correctness depends on the time domain as well as the value domain, i.e., the program must produce the correct result within a strict time requirement, where the time requirement is from a computer perspective (e.g., microseconds, milliseconds, seconds). If the consequence of a failure in the time domain is not catastrophic, the system is called **soft real-time**, otherwise it is called a **hard real-time**. For example, if a program is reading a sensor, like water temperature, missing an occasional reading may not be a problem if the value is only used to update a display on the dashboard of a car, or it may be a problem if the value is used to precisely control a valve on a nuclear reactor. Therefore, the exact difference between soft and hard real-time programming is problem specific and open to a certain degree of interpretation.

A **periodic task** is one whose release is periodic in nature. A is one whose release is not periodic in nature. Such tasks can be subdivided into two categories: aperiodic and sporadic. The difference between these categories lies in the nature of their release frequencies. An **aperiodic task** is one whose release frequency is unbounded. In the extreme, this could lead to an arbitrary large number of simultaneously active tasks. A **sporadic task** has a maximum frequency such that only one instance of a particular sporadic task can be active at a time.

14.1 Soft Real-Time

14.2 Real-Time Exceptions

In the design and implementation of real-time programs, various timing constraints are guaranteed through the use of scheduling algorithms, as well as an EHM. Exceptions are extremely crucial in real-time systems, e.g., deadline expiry or early/late starting exceptions, as they allow a system to react to abnormal situations in a timely fashion. Hecht and Hecht [HH86] demonstrated, through various empirical studies, that the introduction of even the most basic fault-tolerance mechanisms into a real-time system drastically improves its reliability.

The main conflict between real-time and an EHM is the need for constant-time operations and the dynamic choice of a handler [LS98]. As pointed out in Section 3.6.2, p. 30, the dynamic choice of a handler is crucial to an EHM, and therefore, it may be impossible to resolve this conflict. At best, exceptions may only be used in restricted ways in real-time systems when a bound can be established on call stack depth and the number of active handlers, which indirectly puts a bound on propagation.

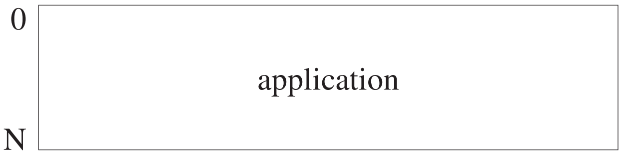
Chapter 15

Distributed Concurrency

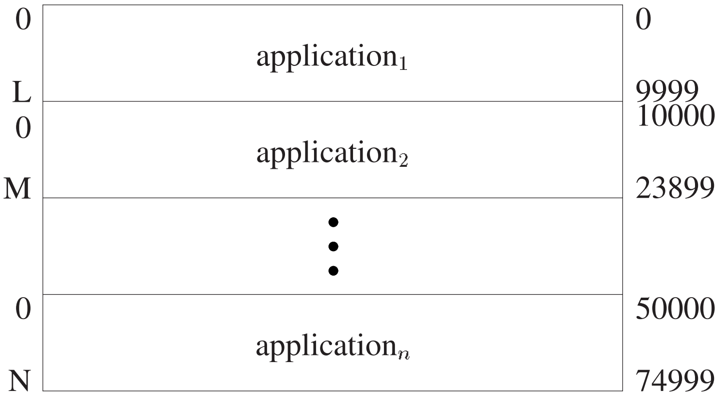
difference between shared memory and non-shared memory

15.1 Remote Procedure Call (RPC)

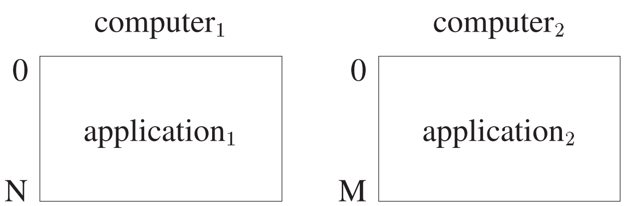
- An application executes in an address space (AS), which has its own set of addresses from 0 to N.



- Most computers can divide physical memory into multiple independent ASs (virtual memory).



- Often a process has its own AS, while a task does not (it exists inside a process's AS).
- Another computer's memory is clearly another AS.

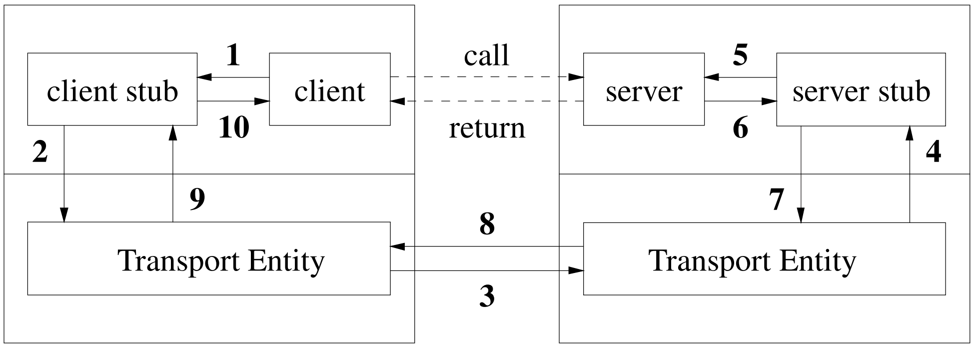


- The ability to call a routine in another address space is an important capability.

- Concurrency is implied by remote calls, but implicit, because each AS (usually) has its own thread of control and each computer has its own CPU.
- Communication is synchronous because the remote call's semantics are the same as a local call.
- Many difficult problems:
  1. How does the code for the called routine get on the other computer? Either it already exists on the other machine or it is shipped there dynamically.
  2. How is the code's address determined for the call once it is available on the other machine?
  3. How are the arguments made available on the other machine?
  4. How are pointer arguments dealt with? Disallowed or completely dereferenced (called marshalling)?
  5. How are different data representations dealt with? E.g., big/little endian addressing, floating point representation, 7,8,16-bit characters.
  6. How are different languages dealt with?
  7. How are arguments and parameters type checked?
     - dynamic checks on each call $\Rightarrow$ additional runtime errors, uses additional CPU time to check and storage for type information.
     - have persistent symbol table information that describes the called interface, and can be located and used during compilation of the call.
- Some network-standards exist that convert data as it is transferred among computers.
- Some interface definition languages (IDL) exist that provide language-independent persistent interface definitions.
- Some systems provide a name-service to look up remote routines (begs the question of how do you find the name-service).
- All schemes must provide a mechanism to:
  - make the call, which requires OS assistance:
    * from the calling program (calling stub)
    * through the OS
    * across the network (possibly)
    * through the other OS
    * to the called routine
  - transmit the arguments (possibly marshalling) and return a result (possibly marshalling)

15.1.1 RPC (without shared memory)

- Desired effect: call from client to server, with possible return value.



1 local call from client to client stub

2 client stub collects arguments into a message (marshalling) and passes it to the local transport (may be an OS call)

3 client transport entity sends message to server transport entity

4 server transport entity passes message to srever stub

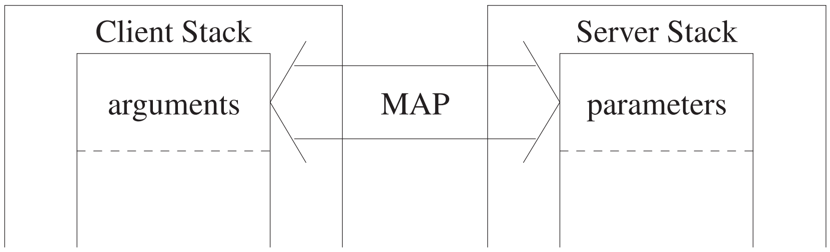
5 server stub unmarshalls arguments and calls local procedure

6 server performs its processing and returns to server stub

7-10 marshall return value(s) into message, pass back to client in similar fashion.

15.1.2 RPC (with shared memory)

- Same as above with regard to call.
- However, the arguments can be passed in shared memory:



- It may be possible to pass data structures with pointers using shared memory.

15.2 Communication Exceptions

- Usually handled by the transport entity.
- Caller dies $\Rightarrow$ callee blocks forever.
- Callee dies $\Rightarrow$ caller blocks forever
- Defective communication:

  lost message - detected by time-out waiting for result by caller

  damaged message - detected by error check at caller or callee

  out-of-order message - detected by callee using information attached to the message

  duplicate message - detected by callee using information attached to the message

- In the first case, the caller will retransmit the message.
- In the second case, the acceptor could request retransmission.
- But if it just ignores the message, it will look like the first case to the caller.
- Reordering messages and discarding duplicates can be done entirely at the callee.

Bibliography

[ABC$^+$93] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hølzle, John H. Maloney, Randall B. Smith, David Ungar, and Mario Wolczko. *The SELF 3.0 Programmer's Reference Manual*. Sun Microsystems, Inc., and Stanford University, 1993. 27

[Agh86] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass., 1986. 410

[And89] Gregory R. Andrews. A Method for Solving Synronization Problems. *Science of Computer Programming*, 13(4):1–21, December 1989. 205

[And90] Thomas E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990. 163

[And91] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1991. 288

[AOC$^+$88] Gregory R. Andrews, Ronald A. Olsson, Michael Coffin, Irving Elshoff, Kelvin Nilsen, Titus Purdin, and Gregg Townsend. An Overview of the SR Language and Implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988. 378, 398

[AS83] Gregory R. Andrews and Fred B. Schneider. Concepts and Notations for Concurrent Programming. *ACM Computing Surveys*, 15(1):3–43, March 1983. 115, 193

[BBG$^+$63] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samuelson, B. Vauquois, J.H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised Report on the Algorithmic Language Algol 60. *Communications of the ACM*, 6(1):1–17, January 1963. 13

[BCK89] Harry Bretthauer, Thomas Christaller, and Jürgen Kopp. Multiple vs. Single Inheritance in Object-oriented Programming Languages. What do we really want? Technical Report Arbeitspapiere der GMD 415, Gesellschaft Für Mathematik und Datenverarbeitung mbH, Schloβ Birlinghoven, Postfach 12 40, D-5205 Sankt Augustin 1, Deutschland, November 1989. 377

[BD93] Alan Burns and Geoff Davies. *Concurrent Programming*. Addison-Wesley, 1993. 2

[BDS$^+$92] P. A. Buhr, Glen Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. μC++: Concurrency in the Object-Oriented Language C++. *Software—Practice and Experience*, 22(2):137–172, February 1992. 2

[BFC95] Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor Classification. *ACM Computing Surveys*, 27(1):63–107, March 1995. 287, 288, 290

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, December 1994. 119

[BHM02] Peter A. Buhr, Ashif Harji, and W. Y. Russell Mok. Exception Handling. In Marvin V. Zelkowitz, editor, *Advances in COMPUTERS*, volume 56, pages 245–303. Academic Press, 2002. 21

[BJ66] C. Böhm and G. Jacopini. Flow diagrams, Turing Machines and Languages with only two Formation Rules. *Communications of the ACM*, 9(5):366–371, May 1966. 8

[BL80] James E. Burns and Nancy A. Lynch. Mutual Exclusion using Indivisible Reads and Writes. In *Proceedings of the 18th Annual Allerton Conference on Communications, Control and Computing*, pages 833–842. ???, ???, 1980. 152

[BLL88] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A System for Object-oriented Parallel Programming. *Software—Practice and Experience*, 18(8):713–732, August 1988. 362, 374

[BMS94] Peter A. Buhr, Hamish I. Macdonald, and Richard A. Stroobosscher. μSystem Annotated Reference Manual, Version 4.4.3. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, September 1994. ftp://plg.uwaterloo.ca/pub/uSystem/uSystem.ps.gz. 362

[BMZ92] Peter A. Buhr, Hamish I. Macdonald, and C. Robert Zarnke. Synchronous and Asynchronous Handling of Abnormal Events in the μSystem. *Software—Practice and Experience*, 22(9):735–776, September 1992. 30, 36, 45, 46, 48

[Bri72] Per Brinch Hansen. Structured Multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972. 250, 253, 254

[Bri73] Per Brinch Hansen. Concurrent Programming Concepts. *Software—Practice and Experience*, 5(4):223–245, December 1973. 165, 249, 250, 255, 256

[Bri75] Per Brinch Hansen. The Programming Language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 2:199–206, June 1975. 288

[Bri81] Per Brinch Hansen. The Design of Edison. *Software—Practice and Experience*, 11(4):363–396, April 1981. 251

[Buh85] P. A. Buhr. A Case for Teaching Multi-exit Loops to Beginning Programmers. *SIGPLAN Notices*, 20(11):14–22, November 1985. 7

[Buh95] Peter A. Buhr. Are Safe Concurrency Libraries Possible? *Communications of the ACM*, 38(2):117–120, February 1995. 32, 249

[Buh06] Peter A. Buhr. μC++ Annotated Reference Manual, Version 5.3.0. Technical report, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, January 2006. ftp://-plg.uwaterloo.ca/pub/uSystem/uC++.ps.gz. 2, 36

[Bur78] James E. Burns. Mutual Exclusion with Linear Waiting Using Binary Shared Variables. *SIGACT News*, 10(2):42–47, Summer 1978. 161

[But97] David R. Butenhof. *Programming with POSIX Threads*. Professional Computing. Addison-Wesley, 1997. 364

[Car90] T. A. Cargill. Does C++ Really Need Multiple Inheritance? In *USENIX C++ Conference Proceedings*, pages 315–323, San Francisco, California, U.S.A., April 1990. USENIX Association. 38, 377

[CES71] E. G. Coffman, Jr., M. J. Elphick, and A. Shoshani. System Deadlocks. *ACM Computing Surveys*, 3(2):67–78, June 1971. 235

[CG89] Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, April 1989. 410

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent Control with Readers and Writers. *Communications of the ACM*, 14(10):667–668, October 1971. 202

[CKL$^+$88] Boleslaw Ciesielski, Antoni Kreczmar, Marek Lao, Andrzej Litwiniuk, Teresa Przytycka, Andrzej Salwicki, Jolanta Warpechowska, Marek Warpechowski, Andrzej Szalas, and Danuta Szczepanska-Wasersztrum. Report on the Programming Language LOGLAN'88. Technical report, Institute of Informatics, University of Warsaw, Pkin 8th Floor, 00-901 Warsaw, Poland, December 1988. 379

[CLR92] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. Electrical Engineering and Computer Science Series. MIT Press/McGraw-Hill, 1992. 176

[Con63] Melvin E. Conway. Design of a Separable Transition-Diagram Compiler. *Communications of the ACM*, 6(7):396–408, July 1963. 67

[Cos03] Pascal Costanza. Dynamic Scoped Functions as the Essence of AOP. *SIGPLAN Notices*, 38(8):29–35, August 2003. 26

[DG87] Thomas W. Doeppner and Alan J. Gebele. C++ on a Parallel Machine. In *Proceedings and Additional Papers C++ Workshop*, pages 94–107, Santa Fe, New Mexico, U.S.A, November 1987. USENIX Association. 374

[DG94] Steven J. Drew and K. John Gough. Exception Handling: Expecting the Unexpected. *Computer Languages*, 20(2), May 1994. 35, 36

[Dij65] Edsger W. Dijkstra. Cooperating Sequential Processes. Technical report, Technological University, Eindhoven, Netherlands, 1965. Reprinted in [Gen68] pp. 43–112. 147, 193, 196, 234, 240

[Dij68a] E. W. Dijkstra. The Structure of the "THE"–Multiprogramming System. *Communications of the ACM*, 11(5):341–346, May 1968. 193

[Dij68b] Edsger W. Dijkstra. Go To Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148, March 1968. Reprinted in [You79] pp. 29–36. 8

[Dij79] Edsger W. Dijkstra. A Tutorial on the Split Binary Semaphore. Technical Report EWD703, Nuenen, Netherlands, March 1979. 205

[Dij80] Edsger W. Dijkstra. The Superfluity of the General Semaphore. Technical Report EWD734, Nuenen, Netherlands, April 1980. 206

[DMN70] O-J Dahl, B. Myhrhaug, and K. Nygaard. *Simula67 Common Base Language*. Norwegian Computing Center, Oslo Norway, October 1970. 256, 359

[DT80] R. W. Doran and L. K. Thomas. Variants of the Software Solution to Mutual Exclusion. *Information Processing Letters*, 10(4/5):206–208, July 1980. 149

[EM72] Murray A. Eisenberg and Michael R. McGuire. Further Comments on Dijkstra's Concurrent Programming Control Problem. *Communications of the ACM*, 15(11):999, November 1972. 152

[FL91] Charles N. Fischer and Richard J. LeBlanc, Jr. *Crafting a Compiler*. Benjamin Cummings, 1991. 30

[Gal96] John Galletly. *OCCAM 2: Including OCCAM 2.1*. UCL (University College London) Press Ltd., second edition, 1996. 378

[GC96] Michael Greenwald and David Cheriton. The Synergy between Non-blocking Synchronization and Operating System Structure. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, pages 123–136, Seattle, Washington, U.S.A., October 1996. USENIX Association. 168

[Geh92] N. H. Gehani. Exceptional C or C with Exceptions. *Software—Practice and Experience*, 22(10):827–848, October 1992. 30, 31, 34, 35, 36, 46

[Gen68] F. Genuys, editor. *Programming Languages*. Academic Press, New York, 1968. NATO Advanced Study Institute, Villard-de-Lans, 1966. 423

[Gen81] W. Morven Gentleman. Message Passing between Sequential Processes: the Reply Primitive and the Administrator Concept. *Software—Practice and Experience*, 11(5):435–466, May 1981. 339, 382

[GG83] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice-Hall, 1983. 77

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995. 78

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000. 4, 33

[Goo75] J. B. Goodenough. Exception Handling: Issues and a Proposed Notation. *Communications of the ACM*, 18(12):683–696, December 1975. 25, 26, 36, 46

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983. 27

[GR89] N. H. Gehani and W. D. Roome. *The Concurrent C Programming Language*. Silicon Press, NJ, 1989. 398

[Hal85] Robert H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Programming. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985. 343

[Hav68] J. W. Havender. Avoiding Deadlock in Multitasking Systems. *IBM Systems Journal*, 7(2):74–84, 1968. 238

[HF92] Paul Hudak and Joseph H. Fasel. A Gentle Introduction to Haskell. *SIGPLAN Notices*, 27(5):T1–53, May 1992. 13

[HH86] H. Hecht and M. Hecht. Software Reliability in the Systems Context. *IEEE Transactions on Software Engineering*, 12(1):51–58, 1986. 415

[Hoa72] C. A. R. Hoare. Towards a Theory of Parallel Programming. In C. A. R. Hoare and R. H. Perott, editors, *Operating Systems Techniques*, pages 61–71. Academic Press, New York, 1972. 250, 251

[Hoa74] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974. 256, 285, 288

[Hol72] Richard C. Holt. Some Deadlock Properties of Computer Systems. *ACM Computing Surveys*, 4(3):179–196, September 1972. 243, 244, 245

[How76] J. H. Howard. Signaling in Monitors. In *Proceedings Second International Conference Software Engineering*, pages 47–52, San Francisco, U.S.A, October 1976. IEEE Computer Society. 288, 290

[HP94] Gerard J. Holzmann and Björn Pehrson. The First Data Networks. *Scientific American*, 12(1):124–129, January 1994. 193

[HS81] Eric C. R. Hehner and R. K. Shyamasundar. An Implementation of P and V. *Information Processing Letters*, 12(4):196–198, August 1981. 155

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979. 73, 75

[IBFW86] Jean D. Ichbiah, John G. P. Barnes, Robert J. Firth, and Mike Woodger. *Rationale for the Design of the ADA Programming Language*. Under Secretary of Defense, Research and Engineering, Ada Joint Program Office, OUSDRE(R&AT), The Pentagon, Washington, D. C., 20301, U.S.A., 1986. 385

[IBM81a] *IBM System/370 Principles of Operation*. Number GA22-7000-8. IBM, 9th edition, October 1981. 169

[IBM81b] International Business Machines. *OS and DOS PL/I Reference Manual*, first edition, September 1981. Manual GC26-3977-0. 32

[Int95] Intermetrics, Inc. *Ada Reference Manual*, international standard ISO/IEC 8652:1995(E) with COR.1:2000 edition, December 1995. Language and Standards Libraries. 385

[Kes77] J. L. W. Kessels. An Alternative to Event Queues for Synchronization in Monitors. *Communications of the ACM*, 20(7):500–503, July 1977. 289

[KGB88] Lawrence J. Kenah, Ruth E. Goldenberg, and Simon F. Bate. *VAX/VMS Internals and Data Structures Version 4.4*. Digital Press, 1988. 47, 51

[Knu84] Jørgen Lindskov Knudsen. Exception Handling — A Static Approach. *Software—Practice and Experience*, 14(5):429–449, May 1984. 29, 30

[Knu87] Jørgen Lindskov Knudsen. Better Exception Handling in Block Structured Systems. *IEEE Software*, 4(3):40–49, May 1987. 29

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Software Series. Prentice Hall, second edition, 1988. 32

[KS90] Andrew Koenig and Bjarne Stroustrup. Exception Handling for C++. *Journal of Object-Oriented Programming*, 3(2):16–33, July/August 1990. 36, 38

[LAB+81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981. 4, 77, 362

[Lab90] Pierre Labrèche. Interactors: A Real-Time Executive with Multiparty Interactions in C++. *SIGPLAN Notices*, 25(4):20–32, April 1990. 374

[Lam74] Leslie Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, 17(8):453–455, August 1974. 155

[Lea97] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, first edition, 1997. 296

[LR80] B. W. Lampson and D. D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980. 296

[LS79] Barbara H. Liskov and Alan Snyder. Exception Handling in CLU. *IEEE Transactions on Software Engineering*, SE-5(6):546–558, November 1979. 33, 36

[LS98] Jun Lang and David B. Stewart. A Study of the Applicability of Existing Exception-Handling Techniques to Component-Based Real-Time Software Technology. *ACM Transactions on Programming Languages and Systems*, 20(2):274–301, March 1998. 415

[Mac77] M. Donald MacLaren. Exception Handling in PL/I. *SIGPLAN Notices*, 12(3):101–104, March 1977. Proceedings of an ACM Conference on Language Design for Reliable Software, March 28–30, 1977, Raleigh, North Carolina, U.S.A. 30, 32

[Mar80] Christopher D. Marlin. *Coroutines: A Programming Methodology, a Language Design and an Implementation*, volume 95 of *Lecture Notes in Computer Science, Ed. by G. Goos and J. Hartmanis*. Springer-Verlag, 1980. 84

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithm for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991. 166, 189

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall Object-Oriented Series. Prentice-Hall, 1992. 34

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978. 21

[Mip91] *MIPS R4000 Microprocessor User's Manual*. MIPS Computer Systems Inc, 1991. 160

[MMG96] G. Motet, A. Mapinard, and J. C. Geoffroy. *Design of Dependable Ada Software*. Prentice Hall, 1996. 29, 30

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993. 32, 359, 379

[MMS79] James G. Mitchell, William Maybury, and Richard Sweet. Mesa Language Manual. Technical Report CSL–79–3, Xerox Palo Alto Research Center, April 1979. 34, 51, 276, 359

[Mok97] Wing Yeung Russell Mok. Concurrent Abnormal Event Handling Mechanisms. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, September 1997. ftp://plg.uwaterloo.ca/pub/-theses/MokThesis.ps.gz. 45

[MOSS96] Stephan Murer, Stephen Omohundro, David Stoutamire, and Clemens Szyperski. Iteration Abstraction in Sather. *ACM Transactions on Programming Languages and Systems*, 18(1):1–15, January 1996. 77, 78, 362

[Mot92] *M68000 Family Programmer's Reference Manual*. Motorola, 1992. 169

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, Cambridge, Massachusetts, U.S.A., 1991. 33

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall Series in Innovative Technology. Prentice-Hall, Inc., 1991. 21, 296

[OEPW96] William G. O'Farrell, Frank Ch. Eigler, S. David Pullara, and Gregory V. Wilson. ABC++. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming in C++*, Scientific and Engineering Computation Series, pages 1–42. MIT Press, 1996. 379

[Pet81] G. L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, June 1981. 148, 149

[PKT73] W. W. Peterson, T. Kasami, and N. Tokura. On the Capabilities of While, Repeat, and Exit Statements. *Communications of the ACM*, 16(8):503–512, August 1973. 9

[RC86] Jonathan Rees and William Clinger. Revised^3 Report on the Algorithmic Language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986. 27

[Rea99] Real Time for Java Experts Group, http://www.rtj.org. September 1999. 346

[Sha81] Mary Shaw, editor. *ALPHARD: Form and Content*. Springer-Verlag, 1981. 362

[Sho87] Jonathan E. Shopiro. Extending the C++ Task System for Real-Time Control. In *Proceedings and Additional Papers C++ Workshop*, pages 77–94, Santa Fe, New Mexico, U.S.A, November 1987. USENIX Association. 374

[Sit92] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, One Burlington Woods Drive, Burlington, MA, U. S. A., 01803, 1992. 160

[SPG91] Abraham Silberschatz, James L. Peterson, and Peter Galvin. *Operating System Concepts*. Addision-Wesley, third edition, 1991. 149

[SPH01] Neil Schemenauer, Tim Peters, and Magnus Lie Hetland. Simple Generators. Technical report, May 2001. http://www.python.org/peps/pep-0255.html. 77

[Sta87] Standardiseringskommissionen i Sverige. *Databehandling – Programspråk – SIMULA*, 1987. Svensk Standard SS 63 61 14. 359

[Ste84] G. Steele. *COMMON LISP: The Language*. Digital Press, 1984. 27

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994. 31, 36, 42

[Str96] Bjarne Stroustrup. A Perspective on Concurrency in C++. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming in C++*, Scientific and Engineering Computation Series, pages xxvi–xxvii. MIT Press, 1996. 2

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997. 2

[Ten77] R. D. Tennent. Language Design Methods Based on Semantic Principles. *Acta Informatica*, 8(2):97–112, 1977. reprinted in [Was80]. 28

[Uni83] United States Department of Defense. *The Programming Language Ada: Reference Manual*, ANSI/MIL-STD-1815A-1983 edition, February 1983. Published by Springer-Verlag. 4

[VAX82] *VAX-11 Architecture Reference Manual*. Digital Press, May 1982. 172

[Was80] Anthony I. Wasserman, editor. *Tutorial: Programming Language Design*. Computer Society Press, 1980. 427

[Wir88] Niklaus Wirth. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer-Verlag, fourth edition, 1988. 359

[WL96] Gregory V. Wilson and Paul Lu, editors. *Parallel Programming in C++*. Scientific and Engineering Computation Series. MIT Press, 1996. 362

[YB85] Shaula Yemini and Daniel M. Berry. A Modular Verifiable Exception-Handling Mechanism. *ACM Transactions on Programming Languages and Systems*, 7(2):214–243, April 1985. 30, 31, 33, 36

[Yea91] Dorian P. Yeager. Teaching Concurrency in the Programming Languages Course. *SIGCSE BULLETIN*, 23(1):155–161, March 1991. The Papers of the Twenty-Second SIGCSE Technical Symposium on Computer Science Education, March. 7–8, 1991, San Antonio, Texas, U.S.A. 5

[You79] Edward Nash Yourdon, editor. *Classics in Software Engineering*. Yourdon Press, 1979. 423

Index