μC++ Annotated Reference Manual

Version 6.1.0

http://plg.uwaterloo.ca/∼usystem/uC++.html

μC++ download or Github (installation: sudo sh u++-6.1.0.sh)

September 13, 2016

Contents

Preface 1

1 μC++ Extensions 3
- 1.1 Design Requirements 3
- 1.2 Elementary Execution Properties 4
- 1.3 High-level Execution Constructs 5

2 μC++ Translator 7
- 2.1 Extending C++ 7
- 2.2 Compile Time Structure of a μC++ Program 8
- 2.3 μC++ Runtime Structure 8
  - 2.3.1 Cluster 8
  - 2.3.2 Virtual Processor 9
- 2.4 μC++ Kernel 10
  - 2.4.1 Compiling a μC++ Program 10
  - 2.4.2 Preprocessor Variables 11
- 2.5 Labelled Break / Continue 11
- 2.6 Finally Clause 12
- 2.7 Coroutine 13
  - 2.7.1 Coroutine Creation and Destruction 14
  - 2.7.2 Inherited Members 15
  - 2.7.3 Coroutine Control and Communication 17
- 2.8 Mutex Type 17
- 2.9 Scheduling 21
  - 2.9.1 Implicit Scheduling 21
  - 2.9.2 External Scheduling 21
    - 2.9.2.1 Accept Statement 21
    - 2.9.2.2 Breaking a Rendezvous 23
    - 2.9.2.3 Accepting the Destructor 24
    - 2.9.2.4 Commentary 24
  - 2.9.3 Internal Scheduling 25
    - 2.9.3.1 Condition Variables and Wait/Signal Statements 25
    - 2.9.3.2 Commentary 26
- 2.10 Monitor 27
  - 2.10.1 Monitor Creation and Destruction 27
  - 2.10.2 Monitor Control and Communication 27
- 2.11 Coroutine Monitor 28
  - 2.11.1 Coroutine-Monitor Creation and Destruction 28
  - 2.11.2 Coroutine-Monitor Control and Communication 28
- 2.12 Thread Creation and Destruction 28
- 2.13 Task 30
  - 2.13.1 Task Creation and Destruction 30

2.13.2 Inherited Members 31
2.13.3 Task Control and Communication 33
2.14 Commentary 33
2.15 Inheritance 34
2.16 Explicit Mutual Exclusion and Synchronization 36
2.16.1 Counting Semaphore 36
2.16.1.1 Commentary 37
2.16.2 Lock 37
2.16.3 Owner Lock 38
2.16.4 Condition Lock 38
2.16.5 Barrier 39
2.17 Atomic Instructions 40
2.18 User Specified Context 42
2.18.1 Predefined Floating-Point Context 43
2.19 Implementation Restrictions 44

3 Asynchronous Communication **47**
3.1 Futures 47
3.1.1 Client Operations 48
3.1.2 Server Operations 48
3.1.3 Explicit Storage Management 49
3.1.4 Example 50
3.1.5 Implicit Storage Management 50
3.1.6 Example 51
3.2 Future Access 51
3.2.1 Select Statement 51
3.2.2 Wait Queue 53
3.3 Servers 54
3.3.1 Executors 57

4 Input/Output **59**
4.1 Nonblocking I/O 59
4.2 C++ Stream I/O 59
4.3 UNIX File I/O 61
4.3.1 File Access 62
4.4 BSD Sockets 63
4.4.1 Client 65
4.4.2 Server 66
4.4.3 Server Acceptor 68

5 Exceptions **71**
5.1 EHM 71
5.2 *μ*C++ EHM 72
5.3 Exception Type 72
5.3.1 Creation and Destruction 72
5.3.2 Inherited Members 73
5.4 Raising 73
5.4.1 Nonlocal Propagation 74
5.4.2 Enabling/Disabling Propagation 74
5.4.3 Concurrent Propagation 76
5.5 Handler 76
5.5.1 Termination 76
5.5.2 Resumption 76
5.5.3 Termination/Resumption 78

5.5.3.1 Recursive Resuming 78
5.5.3.2 Preventing Recursive Resuming 79
5.5.3.3 Commentary 80
5.6 Bound Exceptions 81
5.6.1 C++ Exception-Handling Deficiencies 81
5.6.2 Object Binding 82
5.6.3 Bound Handlers 82
5.6.3.1 Matching 82
5.6.3.2 Termination 82
5.6.3.3 Resumption 82
5.7 Inheritance 83
5.8 Predefined Exception Routines 84
5.8.1 terminate/set_terminate 84
5.8.2 unexpected/set_unexpected 84
5.8.3 uncaught_exception 85
5.9 Programming with Exceptions 85
5.9.1 Terminating Propagation 86
5.9.2 Resuming Propagation 86
5.9.3 Terminating/Resuming Propagation 86
5.10 Predefined Exception-Types 86
5.10.1 Implicitly Enabled Exception-Types 86
5.10.2 Unhandled Exception in Coroutine 86
5.10.3 Breaking a Rendezvous 88

6 Cancellation **91**
6.1 Using Cancellation 92
6.2 Enabling/Disabling Cancellation 92
6.3 Commentary 92

7 Errors **95**
7.1 Static (Compile-time) Warnings/Errors 95
7.2 Dynamic (Runtime) Warnings/Errors 98
7.2.1 Assertions 98
7.2.2 Termination 99
7.2.3 Messages 99
7.2.3.1 Default Actions 99
7.2.3.2 Coroutine 104
7.2.3.3 Mutex Type 106
7.2.3.4 Task 109
7.2.3.5 Condition Variable 109
7.2.3.6 Accept Statement 110
7.2.3.7 Calendar 111
7.2.3.8 Locks 111
7.2.3.9 Cluster 111
7.2.3.10 Heap 112
7.2.3.11 I/O 113
7.2.3.12 Processor 113
7.2.3.13 UNIX 113

8 *μ*C++ Kernel **115**
8.1 Pre-emptive Scheduling and Critical Sections 115
8.2 Memory Management 115
8.3 Cluster 115
8.4 Processors 117

8.4.1 Implicit Task Scheduling . . . 119
8.4.2 Idle Virtual Processors . . . 119
8.4.3 Blocking Virtual Processors . . . 120

9 POSIX Threads (pthreads) 121
9.1 Combining Pthreads with *μ*C++ . . . 121
9.1.1 Linking *μ*C++ and Pthreads Binaries . . . 121
9.1.2 Recompiling Pthreads Modules . . . 121
9.2 *μ*C++ Task as a Pthreads Thread . . . 123
9.3 Semantic Inconsistencies between Pthreads and *μ*C++ . . . 124
9.3.1 Termination of main . . . 124
9.3.2 Cleanup Semantics . . . 124
9.4 Commentary . . . 124

10 Real-Time 127
10.1 Duration and Time . . . 127
10.2 Timeout Operations . . . 130
10.2.1 Time Delay . . . 130
10.2.2 Accept Statement . . . 130
10.2.3 Select Statement . . . 131
10.2.4 I/O . . . 132
10.3 Clock . . . 132
10.4 Periodic Task . . . 133
10.5 Sporadic Task . . . 134
10.6 Aperiodic Task . . . 135
10.7 Priority Inheritance Protocol . . . 135
10.8 Real-Time Scheduling . . . 136
10.9 User-Supplied Scheduler . . . 137
10.10Real-Time Cluster . . . 138
10.10.1 Deadline Monotonic Scheduler . . . 138

11 Miscellaneous 141
11.1 Default Values . . . 141
11.1.1 Task . . . 141
11.1.2 Processor . . . 141
11.1.3 Heap . . . 142
11.2 Symbolic Debugging . . . 142
11.3 Installation Requirements . . . 142
11.4 Installation . . . 142
11.5 Reporting Problems . . . 143
11.6 Contributors . . . 143

A *μ*C++ Grammar 145

B Heap Allocator 147

C Data Structure Library (DSL) 149
C.1 Stack . . . 149
C.1.1 Iterator . . . 150
C.2 Queue . . . 150
C.2.1 Iterator . . . 151
C.3 Sequence . . . 152
C.3.1 Iterator . . . 153

D Example Programs **155**
D.1 Readers And Writer . . . 155
D.2 Bounded Buffer . . . 157
D.2.1 Using Monitor Accept . . . 157
D.2.2 Using Monitor Condition . . . 158
D.2.3 Using Task . . . 159
D.2.4 Using P/V . . . 160
D.3 Disk Scheduler . . . 161
D.4 UNIX File I/O . . . 165
D.5 UNIX Socket I/O . . . 167
D.5.1 Client - UNIX/Datagram . . . 167
D.5.2 Server - UNIX/Datagram . . . 169
D.5.3 Client - INET/Stream . . . 170
D.5.4 Server - INET/Stream . . . 172

Bibliography **175**

Index **179**

Preface

The goal of this work is to introduce concurrency into the object-oriented language C++ [Str97]. To achieve this goal a set of important programming language abstractions were adapted to C++, producing a new dialect called *μ*C++. These abstractions were derived from a set of design requirements and combinations of elementary execution properties, different combinations of which categorized existing programming language abstractions and suggested new ones. The set of important abstractions contains those needed to express concurrency, as well as some that are not directly related to concurrency. Therefore, while the focus of this work is on concurrency, all the abstractions produced from the elementary properties are discussed. While the abstractions are presented as extensions to C++, the requirements and elementary properties are generally applicable to other object-oriented languages.

This manual does not discuss how to use the new constructs to build complex concurrent systems. An in-depth discussion of these issues, with respect to *μ*C++, is available in "Understanding Control Flow with Concurrent Programming using *μ*C++". This manual is strictly a reference manual for *μ*C++. A reader should have an intermediate knowledge of control flow and concurrency issues to understand the ideas presented in this manual as well as some experience programming in C++.

This manual contains annotations set off from the normal discussion in the following way:

□ Annotation discussion is quoted with quads. □

An annotation provides rationale for design decisions or additional implementation information. Also a chapter or section may end with a commentary section, which contains major discussion about design alternatives and/or implementation issues.

Each chapter of the manual does *not* begin with an insightful quotation. Feel free to add your own.

Chapter 1

μC++ Extensions

μC++ [BD92] extends the C++ programming language [Str97] in somewhat the same way that C++ extends the C programming language. The extensions introduce new objects that augment the existing set of control flow facilities and provide for lightweight concurrency on uniprocessor and parallel execution on multiprocessor computers running the UNIX operating system. The following discussion is the rationale for the particular extensions that were chosen.

1.1 Design Requirements

The following requirements directed this work:

- Any linguistic feature that affects code generation *must* become part of the language. In other words, if the compiler can generate code that invalidates the correctness of a library package implementing a particular feature, either the library feature cannot be implemented safely or additional capabilities must be added to the programming language to support the feature. Concurrency is a language feature affected by code generation, and hence, must be added to the programming language [Buh95]. In the case of C++, the concurrency extensions are best added through new kinds of objects.

- All communication among the new kinds of objects must be statically type checkable because static type checking is essential for early detection of errors and efficient code generation. (As well, this requirement is consistent with the fact that C++ is a statically typed programming language.)

- Interaction among the different kinds of objects should be possible, and in particular, interaction among concurrent objects, called tasks, should be possible. This requirement allows a programmer to choose the kind of object best suited to the particular problem without having to cope with communication restrictions.

  In contrast, some approaches have restrictions on interactions among concurrent objects, such as tasks can only interact indirectly through another non-task object. For example, many programming languages that support monitors [Bri75, MMS79, Hol92] require that all communication among tasks be done indirectly through a monitor; similarly, the Linda system [CG89] requires that all communication take place through one or possibly a small number of tuple spaces. This restriction increases the number of objects in the system; more objects consume more system resources, which slows the system. As well, communication among tasks is slowed because of additional synchronization and data transfers with the intermediate object.

- All communication among objects is performed using routine calls; data is transmitted by passing arguments to parameters and results are returned as the value of the routine call. It is confusing to have multiple forms of communication in a language, such as message passing, message queues, or communication ports, as well as normal routine call.

- Any of the new kinds of objects should have the same declaration scopes and lifetimes as existing objects. That is, any object can be declared at program startup, during routine and block activation, and on demand during execution, using a **new** operator.

- All mutual exclusion must be implicit in the programming language constructs and all synchronization should be limited in scope. Requiring users to build mutual exclusion out of locks often leads to incorrect programs. Also, reducing the scope in which synchronization can be used, by encapsulating it as part of language constructs, further reduces errors in concurrent programs.

- Both synchronous and asynchronous communication are needed in a concurrent system. However, the best way to support this is to provide synchronous communication as the fundamental mechanism; asynchronous mechanisms, such as buffering or futures [Hal85], can then be built using synchronous mechanisms. Building synchronous communication out of asynchronous mechanisms requires a protocol for the caller to subsequently detect completion, which is error prone because the caller may not obey the protocol (e.g., never retrieve a result). Furthermore, asynchronous requests require the creation of implicit queues of outstanding requests, each of which must contain a copy of the arguments of the request. This implementation requirement creates a storage management problem because different requests require different amounts of storage in the queue. Therefore, asynchronous communication is too complicated and expensive a mechanism to be hidden in a system.

- An object that is accessed concurrently must have some control over which requester it services next. There are two distinct approaches: control can be based on the kind of request, for example, selecting a requester from the set formed by calls to a particular entry point; or control can be based on the identity of the requester. In the former case, it must be possible to give priorities to the sets of requesters. This requirement is essential for high-priority requests, such as a time out or a termination request. (This priority is to be differentiated from execution priority.) In the latter case, selection control is very precise as the next request must only come from the specified requester. In general, the former case is usually sufficient and simpler to express.

- There must be flexibility in the order that requests are completed. That is, a task can accept a request and subsequently postpone it for an unspecified time, while continuing to accept new requests. Without this ability, certain kinds of concurrency problems are quite difficult to implement, e.g., disk scheduling, and the amount of concurrency is inhibited as tasks are needlessly blocked [Gen81].

All of these requirements are satisfied in μC++ except the first, which requires compiler support. Even through μC++ lacks compiler support, its design assumes compiler support so the extensions are easily added to any C++ compiler.

1.2 Elementary Execution Properties

Extensions to the object concept were developed based on the following execution properties:

thread – is execution of code that occurs independently of and possibly concurrently with other execution; the execution resulting from a thread is sequential. A thread's function is to advance execution by changing execution state. Multiple threads provide concurrent execution. A programming language must provide constructs that permit the creation of new threads and specify how threads are used to accomplish computation. Furthermore, there must be programming language constructs whose execution causes threads to block and subsequently be made ready for execution. A thread is either blocked or running or ready. A thread is **blocked** when it is waiting for some event to occur. A thread is **running** when it is executing on an actual processor. A thread is **ready** when it is eligible for execution but not being executed.

execution state – is the state information needed to permit independent execution. An execution state is either **active** or **inactive**, depending on whether or not it is currently being used by a thread. In practice, an execution state consists of the data items created by an object, including its local data, local block and routine activations, and a current execution location, which is initialized to a starting point. The local block and routine activations are often maintained in a contiguous stack, which constitutes the bulk of an execution state and is dynamic in size, and is the area where the local variables and execution location are preserved when an execution state is inactive. A programming language determines what constitutes an execution state, and therefore, execution state is an elementary property of the semantics of a language. When control transfers from one execution state to another, it is called a **context switch**.

mutual exclusion – is the mechanism that permits an action to be performed on a resource without interruption by other actions on the resource. In a concurrent system, mutual exclusion is required to guarantee consistent generation of results, and cannot be trivially or efficiently implemented without appropriate programming language constructs.

The first two properties represent the minimum needed to perform execution, and seem to be fundamental in that they are not expressible in machine-independent or language-independent ways. For example, creating a new thread requires creation of system runtime control information, and manipulation of execution states requires machine specific operations (modifying stack and frame pointers). The last property, while expressible in terms of simple language statements, can only be done by algorithms that are error-prone and inefficient, e.g., Dekker-like algorithms, and therefore, mutual exclusion must also be provided as an elementary execution property, usually through special atomic hardware instructions.

1.3 High-level Execution Constructs

A programming language designer could attempt to provide these 3 execution properties as basic abstractions in a programming language [BLL88], allowing users to construct higher-level constructs from them. However, some combinations might be inappropriate or potentially dangerous. Therefore, all combinations are examined, analyzing which ones make sense and are appropriate as higher-level programming language constructs. What is interesting is that enumerating all combination of these elementary execution properties produces many existing high-level abstractions and suggests new ones.

The three execution properties are properties of objects. Therefore, an object may or may not have a thread, may or may not have an execution state, and may or may not have mutual exclusion. Different combinations of these three properties produce different kinds of objects. If an object has mutual exclusion, this means that execution of certain member routines are mutually exclusive of one another. Such a member routine is called a mutual-exclusion member (**mutex member**). In the situation where an object does not have the minimum properties required for execution, i.e., thread and execution state, those of its caller are used.

Table 1.1 shows the different abstractions possible when an object possesses different execution properties:

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 1.1: Fundamental Abstractions

Case 1 is an object, such as a **free routine** (a routine not a member of an object) or an object with member routines neither of which has the necessary execution properties, called a **class object**. In this case, the caller's thread and execution state are used to perform execution. Since this kind of object provides no mutual exclusion, it is normally accessed only by a single thread. If such an object is accessed by several threads, explicit locking may be required, which violates a design requirement. 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 abstraction is a **monitor** [Hoa74]. Case 3 is an object that has its own execution state but no thread. 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. This abstraction is a **coroutine** [Mar80]. Case 4 is like Case 3 but deals with the concurrent-access problem by implicitly ensuring mutual exclusion; the name **coroutine monitor** has been adopted for this case. 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 successively executing portions of a single computation, which does not seem useful. 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, servicing of requests would, in general, require explicit locking, which violates a design requirement. Furthermore, there is no 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, called a **task**.

The abstractions suggested by this categorization come from fundamental properties of execution and not ad hoc decisions of a programming language designer. While it is possible to simplify the programming language design by only supporting the task abstraction [SBG+90], which provides all the elementary execution properties, this would unnecessarily complicate and make inefficient solutions to certain problems. As will be shown, each of the non-rejected abstractions produced by this categorization has a particular set of problems it can solve, and therefore, each has a place in a programming language. If one of these abstractions is not present, a programmer may be forced to contrive a solution for some problems that violates abstraction or is inefficient.

Chapter 2

μC++ Translator

The μC++ translator[1] reads a program containing language extensions and transforms each extension into one or more C++ statements, which are then compiled by an appropriate C++ compiler and linked with a concurrency runtime library. Because μC++ is only a translator and not a compiler, some restrictions apply that would be unnecessary if the extensions were part of the C++ programming language. Similar, but less extensive translators have been built: MC [RH87] and Concurrent C++ [GR88].

2.1 Extending C++

Operations in μC++ are expressed explicitly, i.e., the abstractions derived from the elementary properties are used to structure a program into a set of objects that interact, possibly concurrently, to complete a computation. This situation is to be distinguished from implicit schemes, such as those that attempt to *discover* concurrency in an otherwise sequential program, e.g., by parallelizing loops and access to data structures. While both schemes are complementary, and hence, can appear together in a single programming language, implicit schemes are limited in their capacity to *discover* concurrency, and therefore, the explicit scheme is essential. Currently, μC++ only supports the explicit approach, but nothing in its design precludes the addition of the implicit approach.

The abstractions in Table 1.1, p. 5 are expressed in μC++ using two new type specifiers, **_Coroutine** and **_Task**, which are extensions of the **class** construct, and hence, define new types. In this manual, a type defined by the **class** construct and the new constructs are called **class type**, **monitor type**, **coroutine type**, **coroutine-monitor type** and **task type**, respectively. The terms **class object**, **monitor**, **coroutine**, **coroutine monitor** and **task** refer to the objects created from such types. The term **object** is the generic term for any instance created from any type. All objects can be declared externally, in a block, or using the **new** operator. Two new type qualifiers, **_Mutex** and **_Nomutex**, are also introduced to specify the presence or absence of mutual exclusion on the member routines of a type (see Table 2.1). The default qualification values have been chosen based on the expected frequency of use of the new types. Several new statements are added to the language; each is used to affect control in objects created by the new types. Appendix A, p. 145 shows the grammar for all the μC++ extensions.

object properties		object's member routine properties	
thread	execution state	no mutual exclusion	mutual exclusion
no	no	[**_Nomutex**]† **class**	**_Mutex class**
no	yes	[**_Nomutex**] **_Coroutine**	**_Mutex _Coroutine**
yes	yes	N/A	[**_Mutex**] **_Task**

† [ ] implies default qualification if not specified

Table 2.1: New Type Specifiers

μC++ executes on uniprocessor and multiprocessor shared-memory computers. On a uniprocessor, concurrency is achieved by interleaving execution to give the appearance of parallel execution. On a multiprocessor computer, con-

[1] The term “translator” is used rather than preprocessor because μC++ programs are partially parsed and symbol tables are constructed. A preprocessor, such as cpp, normally only manipulates strings.

currency is accomplished by a combination of interleaved execution and true parallel execution. Furthermore, μC++ uses a **shared-memory model**. This single memory may be the address space of a single UNIX process or a memory shared among a set of kernel threads. A memory is populated by routine activations, class objects, coroutines, monitors, coroutine monitors and concurrently executing tasks, all of which have the same addressing scheme for accessing the memory. Because these entities use the same memory they can be **lightweight**, so there is a low execution cost for creating, maintaining and communicating among them. This approach has its advantages as well as its disadvantages. Communicating objects do not have to send large data structures back and forth, but can simply pass pointers to data structures. However, this technique does not lend itself to a distributed environment with separate address spaces.

> □ Approaches taken by distributed shared-memory systems may provide the necessary implementation mechanisms to make the distributed memory case similar to the shared-memory case. □

2.2 Compile Time Structure of a μC++ Program

A μC++ program is constructed exactly like a normal C++ program with one exception: the main (starting) routine is a member of an initial task called uMain, which has the following structure (Section 2.13, p. 30 details the task construct):

```
_Task uMain {
  private:
    int argc;                  // number of arguments on the shell command line
    char **argv;               // pointers to tokens on the shell command line
    int &uRetCode;             // return value to the shell
    void main();               // user provides body for this routine
  public:
    uMain( int argc, char *argv[ ] ) : argc(argc), argv(argv) {}
};
```

A μC++ program must define the body for the main member routine of this initial task, e.g.:

```
. . . // normal C++ declarations and routines
void uMain::main() {           // body for initial task uMain
    . . .
    switch( argc ) {           // use argc from uMain
      case 2:
        no = atoi(argv[1]);    // use argv from uMain
    . . .
    uRetCode = 0;              // use uRetCode from uMain
}
```

μC++ supplies the free routine main to initialize the μC++ runtime environment and creates the task uMain, of which routine uMain::main is a member. Member uMain::main has available, as local variables, the same two arguments that are passed to the free routine main: argc, and argv. To return a value back to the shell, set the variable uRetCode and return from uMain::main; uRetCode is initialized to zero.

2.3 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.

2.3.1 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 10, p. 127 for other kinds of schedulers). This simple scheduling results in automatic load balancing of tasks on processors. Figure 2.1 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++.

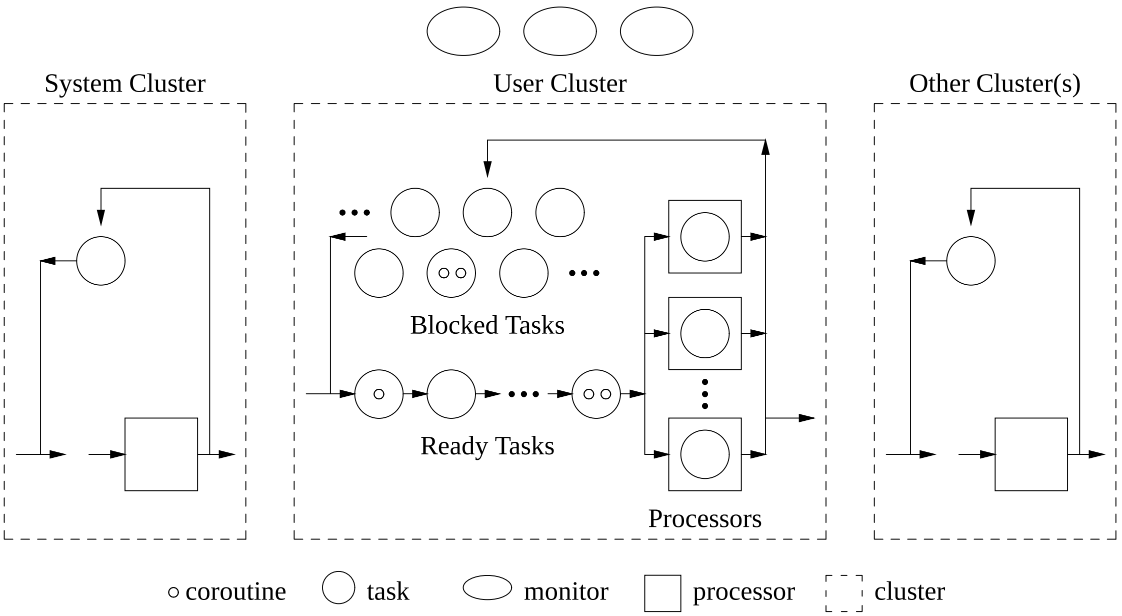


Figure 2.1: Runtime Structure of a μC++ Program

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.

2.3.2 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 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.

2.4 μC++ Kernel

After a μC++ program is translated and compiled, a runtime concurrency library is linked in with the resulting program, called the μC++ kernel. There are two versions of the μC++ kernel: the unikernel, which is designed to use a single processor (in effect, there is only one virtual processor); and the multikernel, which is designed to use several processors. Thus, the unikernel is sensibly used on systems with a single hardware processor or when kernel threads are unavailable; the multikernel is sensibly used on systems that have multiple hardware processors and when kernel threads are available. Table 2.2 shows the situations where each kernel can be used. The unikernel can be used in a system with multiple hardware processors and kernel threads but does not take advantage of either of these capabilities. The multikernel can be used on a system with a single hardware processor and kernel threads but performs less efficiently than the unikernel because it uses multiprocessor techniques unnecessarily.

	no kernel threads	kernel threads
single processor	unikernel, yes multikernel, no	unikernel, yes multikernel, yes, but inefficient
multiple processors	unikernel, yes multikernel, no	unikernel, yes, but no parallelism multikernel, yes

Table 2.2: When to Use the Unikernel and Multikernel

Each of the μC++ kernels has a debugging version, which performs a number of runtime checks. For example, the μC++ kernel provides no support for automatic growth of stack space for coroutines and tasks because this would require compiler support. The debugging version checks for stack overflow whenever context switches occur among coroutines and tasks, which catches many stack overflows; however, stack overflow can still occur if insufficient stack area is provided, which can cause an immediate error or unexplainable results. Many other runtime checks are performed in the debugging version. After a program is debugged, the non-debugging version can be used to increase performance.

2.4.1 Compiling a μC++ Program

The u++ command is used to compile a μC++ program. This command works just like the GNU g++ [Tie90] command for compiling C++ programs, e.g.:

```
u++ [C++ options] yourprogram.C [assembler and loader files]
```

The following additional options are available for the u++ command:

-debug The program is linked with the debugging version of the unikernel or multikernel. The debug version performs runtime checks to help during the debugging phase of a μC++ program, but substantially slows the execution of the program. The runtime checks should only be removed after the program is completely debugged. **This option is the default.**

-nodebug The program is linked with the non-debugging version of the unikernel or multikernel, so the execution of the program is faster. ***However, no runtime checks or asserts are performed so errors usually result in abnormal program termination.***

-yield When a program is translated, a random number of context switches occur at the beginning of each member routine so that during execution on a uniprocessor there is a better simulation of parallelism. (This non-determinism in execution is in addition to random context switching due to pre-emptive scheduling, see Section 8.4.1, p. 119). The extra yields of execution can help during the debugging phase of a μC++ program, but substantially slows the execution of the program.

-noyield Additional context switches are not inserted in member routines. **This option is the default.**

-verify When a program is translated, a check to verify that the stack has not overflowed occurs at the beginning of each member routine. (This checking is in addition to checks on each context switch provided by the -debug option.) Verifying the stack has not overflowed is important during the debugging phase of a μC++ program, but slows the execution of the program.

-noverify Stack-overflow checking is not inserted in member routines. **This option is the default.**

- -multi The program is linked with the multikernel.
- -nomulti The program is linked with the unikernel. **This option is the default.**
- -quiet The *μ*C++ compilation message is not printed at the beginning of a compilation.
- -noquiet The *μ*C++ compilation message is printed at the beginning of a compilation. **This option is the default.**
- -U++ Only the C preprocessor and the *μ*C++ translator steps are performed and the transformed program is written to standard output, which makes it possible to examine the code generated by the *μ*C++ translator.
- -compiler *path-name* The path-name of the compiler used to compile a *μ*C++ program(s). The default is the compiler used to compile the *μ*C++ runtime library. It is unsafe to use a different compiler unless the generated code is binary compatible. (See Section 11.3, p. 142 for supported compilers.)

When multiple conflicting options appear on the command line, e.g., -debug followed by -nodebug, the last option takes precedence.

2.4.2 Preprocessor Variables

When programs are compiled using u++, the following preprocessor variables are available:

- __U_CPLUSPLUS__ is always available during preprocessing and its value is the current major version number of *μ*C++.[2]
- __U_CPLUSPLUS_MINOR__ is always available during preprocessing and its value is the current minor version number of *μ*C++.
- __U_CPLUSPLUS_PATCH__ is always available during preprocessing and its value is the current patch version number of *μ*C++.
- __U_DEBUG__ is available during preprocessing if the -debug compilation option is specified.
- __U_YIELD__ is available during preprocessing if the -yield compilation option is specified.
- __U_VERIFY__ is available during preprocessing if the -verify compilation option is specified.
- __U_MULTI__ is available during preprocessing if the -multi compilation option is specified.

These preprocessor variables allow conditional compilation of programs that must work differently in these situations. For example, to allow a normal C/C++ program to be compiled using *μ*C++, the following is necessary:

```
#ifdef __U_CPLUSPLUS__
void uMain::main() {
#else
int main( int argc, char *argv[] ) {
#endif
    // body of main routine
}
```

which conditionally includes the correct definition for main if the program is compiled using u++.

2.5 Labelled Break / Continue

While C++ provides **continue** and **break** statements for altering control flow, both are restricted to one level of nesting for a particular control structure. Unfortunately, this restriction forces programmers to use **goto** to achieve the equivalent control-flow for more than one level of nesting. To prevent having to switch to the **goto**, *μ*C++ extends the **continue** and **break** with a target label to support static multi-level exit [Buh85, GJSB00]. For both **continue** and **break**, 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**, **if** or compound ({}) statement.

The following example shows the labelled **continue** specifying which control structure is the target for the next loop iteration:

[2] The C preprocessor allows only integer values in a preprocessor variable so a value like “6.1.0” is not allowed. Hence, the need to have three variables for the major, minor and patch version number.

```
          C++                            μC++
do {                             L1: do {
    while ( ... ) {                  L2: while ( ... ) {
        for ( ... ) {                    L3: for ( ... ) {
            ... goto L1; ...                 ... continue L1; ...    // continue do
            ... goto L2; ...                 ... continue L2; ...    // continue while
            ... goto L3; ...                 ... continue L3; ...    // continue for
        L3: ; }                          } // for
    L2: ; }                          } // while
L1: ; } while ( ... );           } while ( ... );
```

The innermost loop has three restart points, which cause the next loop iteration to begin.

The following example shows the labelled **break** specifying which control structure is the target for exit:

```
            C++                                μC++
{                                    L1: {
    ... declarations ...                 ... declarations ...
    switch ( ... ) {                     L2: switch ( ... ) {
      case 3:                              case 3:
        if ( ... ) {                         L3: if ( ... ) {
            for ( ... ) {                         L4: for ( ... ) {
                ... goto L1; ...                      ... break L1; ...    // exit compound statement
                ... goto L2; ...                      ... break L2; ...    // exit switch
                ... goto L3; ...                      ... break L3; ...    // exit if
                ... goto L4; ...                      ... break L4; ...    // exit loop
            } L4: ;                               } // for
        } else {                              } else {
            ... goto L3; ...                      ... break L3; ...        // exit if
        } L3: ;                               } // if
    } L2: ;                              } // switch
} L1: ;                              } // compound
```

The innermost loop has four exit points, which cause termination of one or more of the four nested control structures.

Both **continue** and **break** with target labels are simply a **goto** restricted in the following ways:

- They cannot be used to create a loop. This means that only the looping construct can be used to create a loop. This restriction is important since all situations that can result in repeated execution of statements in a program are clearly delineated.

- Since they always transfer out of containing control structures, they cannot be used to branch into a control structure.

The advantage of the labelled **continue**/**break** is allowing static multi-level exits without having to use the **goto** statement and tying 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 is at the end of the control structure, which 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 constructs to be added or removed without affecting existing constructs. The implicit targets of the current **continue** and **break**, i.e., the closest enclosing loop or **switch**, change as certain constructs are added or removed.

2.6 Finally Clause

The C++ **try** block is augmented with a Java-style finally clause:

```
try {
    ...
} ... // optional catch clauses
} _Finally compound-statement
```

which must appear *after* all catch clauses. The finally clause is always executed, i.e., if the **try** block ends normally or if an exception is raised. If an exception is raised and caught, the handler is run *before* the finally clause. Like a destructor, a finally clause can raise an exception but *not* if there is an exception being propagated. A finally handler can access any types and variables visible in its local scope, but it cannot perform a **break**, **continue** or **return** from within the handler.

> □ While early exit from a finally clause is supported in Java, it is questionable programming. A finally clause *always* cleans up the execution environment. Not completing the finally clause by an early exit implies some clean up is not performed, which is counter-intuitive. □

2.7 Coroutine

A coroutine is an object with its own execution state, so its execution can be suspended and resumed. Execution of a coroutine is suspended as control leaves it, only to carry on from that point when control returns at some later time. This property means a coroutine is not restarted at the beginning on each activation and its local variables are preserved. Hence, a coroutine solves the class of problems associated with finite-state machines and push-down automata, which are logically characterized by the ability to retain state between invocations. In contrast, a free routine or member routine always executes to completion before returning so its local variables only persist for a particular invocation.

> A coroutine executes serially, and hence there is no concurrency implied by the coroutine construct. However, the ability of a coroutine to suspend its execution state and later have it resumed is the precursor to true tasks but without concurrency problems; hence, a coroutine is also useful to have in a programming language for teaching purposes because it allows incremental development of these properties [Yea91].

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

```
[ _Nomutex] _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
};
```

The coroutine type has one distinguished member, named main; this distinguished member is called the **coroutine main**. Instead of allowing direct interaction with main, its visibility is normally **private** or **protected**; therefore, a coroutine can only be activated indirectly by one of the coroutine's member routines. 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. Hence, a user interacts with a coroutine indirectly through its member routines. This approach 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 global variables, called **communication variables**, which are accessible from both the coroutine's member and main routines.

A coroutine can suspend its execution at any point by activating another coroutine, which is done in two ways. First, a coroutine can implicitly reactivate the coroutine that previously activated it via member suspend. Second, a coroutine can explicitly invoke a member of another coroutine, which causes activation of that coroutine via member resume. These two forms result in two different styles of coroutine control flow. A **full coroutine** is part of a resume cycle, while a **semi-coroutine** [Mar80, p. 4, 37] is not part of a resume cycle. A full coroutine can 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.

> □ Simulating a coroutine with a subroutine requires retaining data in variables with global scope or variables with **static** storage-class between invocations. However, retaining state in these ways violates the principle of abstraction and does not generalize to multiple instances, since there is only one copy of the storage in both cases. Also, without a separate execution state, activation points must be managed explicitly, requiring the execution logic to be written as a series of cases, each ending by recording the next case to be executed on re-entry. However, explicit management of activation points is complex and error-prone, for more than a small number of activation points.

Simulating a coroutine with a class solves the problem of abstraction and does generalize to multiple instances, but does not handle the explicit management of activation points. Simulating a coroutine with a task, which also has an execution state to handle activation points, is non-trivial because the organizational structure of a coroutine and task are different. Furthermore, simulating full coroutines, which form a cyclic call-graph, may be impossible with tasks because of a task's mutual-exclusion, which could cause deadlock (not a problem in μC++ because multiple entry is allowed by the same thread). 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 this implementation, 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. □

2.7.1 Coroutine Creation and Destruction

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

```
_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( . . . );       // call a member routine that activates the coroutine
    cp->r( . . . );         // call a member routine that activates the coroutine
    . . .
} // 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 any base-class constructors are executed in the normal order. 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, which includes constructors. 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.

A coroutine terminates when its main routine returns. When a coroutine terminates, it activates the coroutine or task that caused main to *start* execution. This choice ensures the starting sequence is a tree, i.e., there are no cycles. A thread can move in a cycle among a group of coroutines but termination always proceeds back along the branches of the starting tree. This choice of activation does impose certain requirements on the starting order of coroutines, but it is essential to ensure that cycles can be broken at termination. *Activating a terminated coroutine is an error.* A coroutine's destructor is invoked by the deallocating thread when the block containing the coroutine declaration terminates or by an explicit **delete** statement for a dynamically allocated coroutine. If the coroutine is not terminated when it is deallocated, termination is forced *before* executing the destructor, which unwinds the coroutine's stack executing any destructors for objects on the stack. A consequence of this semantics is that the destructor may not resume a coroutine, so it is asymmetric with the coroutine's constructor.

Like a class object, a coroutine may be deleted at any time *even if the coroutine's* main *routine is started but not terminated*, i.e., the coroutine is still suspended in its main routine. Before the coroutine's destructor is run, the coroutine's stack is unwound via the cancellation mechanism (see Section 6, p. 91), to ensure cleanup of resources allocated on the coroutine's stack. This unwinding involves an implicit resume of the coroutine being deleted.

Like a routine or class, a coroutine can access all the external variables of a C++ program and the heap area. Also, any **static** member variables declared within a coroutine are shared among all instances of that coroutine type. If a coroutine makes global references or has **static** variables and is instantiated by different tasks, there is the general problem of concurrent access to these shared variables. Therefore, it is suggested that these kinds of references be used with extreme caution.

2.7.2 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 );
    uBaseCoroutine( void *storage, unsigned int storageSize );

    void *stackPointer() const;                  // stack info
    unsigned int stackSize() const;
    void *stackStorage() const;
    ptrdiff_t stackFree() const;
    ptrdiff_t stackUsed() const;
    void verify();

    const char *setName( const char *name );  // coroutine info
    const char *getName() const;
   enum State { Start, Inactive, Active, Halt };
    State getState() const;
    uBaseCoroutine &starter() const;
    uBaseCoroutine &resumer() const;

    static int asyncpoll();                      // asynchronous exceptions

    enum CancellationState { CancelEnabled, CancelDisabled };
    void cancel();                               // cancellation
    bool cancelled();
    bool cancelInProgress();

    _Event Failure;                              // exceptions
    _Event UnhandledException;
};
```

The member routines resume and suspend are discussed in Section 2.7.3, p. 17.

The overloaded constructor routine uBaseCoroutine has the following forms:

uBaseCoroutine() – creates a coroutine on the current cluster with the cluster's default stack size.

uBaseCoroutine(**unsigned int** stackSize) – creates a coroutine on the current cluster with the specified *minimum* stack size (in bytes). The amount of storage for the coroutine's stack is always greater than this stack size, as extra information is stored on the stack.

uBaseCoroutine(**void** *storage, **unsigned int** storageSize) – creates a coroutine on the current cluster using the specified storage and *maximum* storage size (in bytes) for the coroutine's stack. The amount of storage for the coroutine's stack is always less than actual storage size, as extra information is stored on the stack. ***This storage is NOT freed at coroutine deallocation***. If the specified storage address is zero (NULL), the storage size becomes a stack size, as in the previous constructor.

A coroutine type can be designed to allow declarations to specify the stack storage and size by doing the following:

```
_Coroutine C {
  public:
    C() : uBaseCoroutine( 8192 ) {};              // default 8K stack
    C( unsigned int s ) : uBaseCoroutine( s ) {}; // user specified stack size
    C( void *st, unsigned int s ) : uBaseCoroutine( st,s ) // user specified stack storage and size
    . . .
};
C x, y( 16384 ), z( area, 32768 );     // x => 8K stack, y => 16K stack, z => stack < 32K at "area"
```

The member routine stackPointer returns the address of the stack pointer. If a coroutine calls this routine, its current stack pointer is returned. If a coroutine calls this routine for another coroutine, the stack pointer saved at the last context switch of the other coroutine is returned; this may not be the current stack pointer value for that coroutine.

The member routine stackSize returns the maximum amount of stack space that is allocated for this coroutine. A coroutine cannot exceed this value during its execution. The member routine stackStorage returns the address of the stack storage for this coroutine. On most computers, the stack grows down (high address) towards the stack-storage address (low address). If a coroutine is created with specific stack storage, the address of that storage is returned; otherwise, the address of the µC++ created stack storage is returned.

The member routine stackFree returns the amount of free stack space. If a coroutine calls this routine, its current free stack space is returned. If a coroutine calls this routine for another coroutine, the free stack space at the last context switch of the other coroutine is returned; this may not be the current free stack space for that coroutine.

The member routine stackUsed returns the amount of used stack space. If a coroutine calls this routine, its current used stack space is returned. if a coroutine calls this routine for another coroutine, the used stack space at the last context switch of the other coroutine is returned; this may not be the current used stack space for that coroutine.

The member routine verify 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.

□ When the -verify option is used, calls to verify are automatically inserted at the beginning of each member routine, but not after each set of declarations. □

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.

The member routine starter returns the coroutine's starter, i.e., the coroutine that performed the first resume of this coroutine (see Section 2.7.1, p. 14). The member routine resumer returns the coroutine's last resumer, i.e., the coroutine that performed the last resume of this coroutine (see Section 2.7.3).

The member routine asyncpoll returns the number of asynchronous exceptions propagated by this call to poll. asyncpoll is a static member-routine that always polls the calling thread's asynchronous event-queue; use uBaseCoroutine::asyncpoll() for a call outside of a coroutine or task type. Section 5.4.2, p. 74 discusses propagation control in detail.

The member routine cancel marks the coroutine/task for cancellation. The member routine cancelled returns true if the coroutine/task is marked for cancellation, and false otherwise. The member routine cancelInProgress returns true if cancellation is started for the coroutine/task. Section 6, p. 91 discusses cancellation in detail.

The type **_Event** is defined in Section 5.3, p. 72.

The free 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 free 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 freeRtn( . . . ) {
        . . .                        // declarations
        uThisCoroutine().verify();       // check for stack overflow
        . . .                        // code
    }
```

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

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

2.7.3 Coroutine Control and Communication

Control flow among coroutines is specified by the protected members resume and suspend. 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. Members resume and suspend are composed of two parts. The first part inactivates the coroutine that calls the member and the second part reactivates another coroutine; the difference is which coroutine is reactivated. 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.

Figure 2.2 shows a semi-coroutine producer and consumer coroutine, and a driver routine. Notice the explicit call from Prod's main routine to delivery and then the return back when delivery completes. delivery always activates its coroutine, which subsequently activates delivery.

Figure 2.3, p. 19 shows a full-coroutine producer and consumer coroutine, and a driver routine. Notice the calls to member resume in routines payment and delivery. The resume in routine payment activates the execution state associated with Prod::main and that execution state continues in routine Cons::delivery. Similarly, the resume in routine delivery activates the execution state associated with Cons::main and that execution state continues in Cons::main initially and subsequently in routine Prod::payment. This cyclic control flow and the termination control flow is illustrated in Figure 2.4, p. 20.

2.8 Mutex Type

A mutex type consists of a set of variables and a set of mutex members that operate on the variables. ***A mutex type has at least one mutex member.*** Objects instantiated from mutex types have the property that mutex members are executed with mutual exclusion; that is, only one task at a time can be executing in the mutex members. Similar to an execution state, a mutex object is either active or inactive, depending on whether or not a task is executing a mutex member (versus a task executing the coroutine main). Mutual exclusion is enforced by **locking** the mutex object when execution of a mutex member begins and **unlocking** it when the active task voluntarily gives up control of the mutex object by waiting in or exiting from the monitor. If another task invokes a mutex member while a mutex object is locked, the task is blocked until the mutex object becomes unlocked. ***An active task may call other mutex members either directly from within the mutex type or indirectly by calling another object, which subsequently calls back into the mutex object.*** If an active task enters multiple mutex objects, it owns the mutex locks for these objects and can enter anyone of them again without having to reacquire their locks. If an active task releases control of one of these mutex objects by waiting within it, which implicitly unlocks that object, ***the task does not unlock any other mutex objects it currently owns***. If an active task releases control of one of these mutex objects by exiting from it, which implicitly unlocks that object, ***the task must do so in strict nested order***, i.e., Last-In first-Out (LIFO) order

Consumer

```
_Coroutine Cons {
    int p1, p2, status;   // communication
    bool done;
    void main() {
        // 1st resume starts here
        int money = 1;
        for ( ;; ) {
            cout << "cons receives: " <<
                p1 << ", " << p2;
          if ( done ) break;
            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
    }
}; // Cons
```

Producer

```
_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
    }
}; // Prod

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

Figure 2.2: Semi-Coroutine Producer-Consumer

of mutex-object acquisition (see Section 7.2.3.3, p. 106). This LIFO restriction results solely because there does not seem to be any useful examples for non-LIFO locking, and it is often an indication of an error in a program.

When **_Mutex** or **_Nomutex** qualifies a type, e.g.:

```
_Mutex class M {
  private:
    char z( . . . );        // default nomutex
  public:
    M();                    // default nomutex
    ~M();                   // default mutex
    int x( . . . );         // default mutex
    float y( . . . );       // default mutex
};
```

it defines the default form of mutual exclusion on *all* public member routines, except the constructor, which is never mutex, and the destructor, which is always mutex for a mutex type. Hence, public member routines x and y of mutex type M are mutex members executing mutually exclusively of one another. Member routines that are **protected** and **private** are *always* implicitly **_Nomutex**, except for the destructor of a mutex type, which is always **_Mutex** regardless of its visibility. ***Because the destructor of a mutex type is always executed with mutual exclusion, the call to the destructor may block, either at termination of a block containing a mutex object or when deleting a dynamically allocated mutex object.*** If a mutex qualifier is specified on a forward declaration, e.g.:

```
_Mutex class M;           // forward declaration
. . .
_Mutex class M {. . .}    // actual declaration
```

it must match with the actual declaration. In general, it is best *not* to put a mutex qualifier on a forward declaration so the default can be changed on the actual declaration without having to change the forward declaration.

Consumer

```
_Coroutine Cons {
    Prod &prod;          // communication
    int p1, p2, status;
    bool done;
    void main() {
        // 1st resume starts here
        int money = 1, receipt;
        for ( ;; ) {
            cout << "cons receives: " <<
                p1 << ", " << p2;
          if ( done ) break;
            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();
    }
}; // Cons
```

Producer

```
_Coroutine Prod {
    Cons *cons;          // communication
    int N, money, receipt;
    void main() {        // starter umain
        // 1st resume starts here
        int i, p1, p2, status;
        for ( i = 0; i < N; i += 1 ) {
            p1 = rand() % 100;
            p2 = rand() % 100;
            cout << "prod delivers: " <<
                p1 << ", " << p2 << endl;
            status = cons->delivery( p1, p2 );
            cout << "prod payment of $" <<
                money << endl;
            cout << "prod status: " <<
                status << endl;
            receipt += 1;
        }
        cons->stop();
        cout << "prod stops" << endl;
    }
  public:
    Prod() : receipt(0) {}
    int payment( int money ) {
        Prod::money = money;
        resume();        // restart prod
        return receipt;
    }
    void start( int N, Cons &c ) {
        Prod::N = N;
        cons = &c;
        resume();
    }
}; // Prod
void uMain::main() {
    Prod prod;
    Cons cons( prod );
    prod.start( 5, cons );
}
```

Figure 2.3: Full-Coroutine Producer-Consumer

A mutex qualifier may be needed for **protected** and **private** member routines in mutex types, e.g.:

```
_Mutex class M {
  private:
    _Mutex char z( . . . );    // explicitly qualified member routine
    . . .
};
```

because another task may need access to these member routines. For example, when a **friend** task calls a **protected** or **private** member routine, these calls may need to provide mutual exclusion.

A public member of a mutex type can be explicitly qualified with **_Nomutex**. In general, a **_Nomutex** routine is error-prone because the lack of mutual exclusion permits concurrent updating to object variables. However, there are two situations where a nomutex public member are useful: first, for read-only member routines where execution speed is of critical importance; and second, to encapsulate a sequence of calls to several mutex members to establish a protocol, which ensures that a user cannot violate the protocol since it is part of the type's definition.

The general structure of a mutex object is shown in Figure 2.5. All the implicit and explicit data structures associated with a mutex object are discussed in the following sections. Notice each mutex member has a queue associated with it on which calling tasks wait if the mutex object is locked. A nomutex member has no queue.

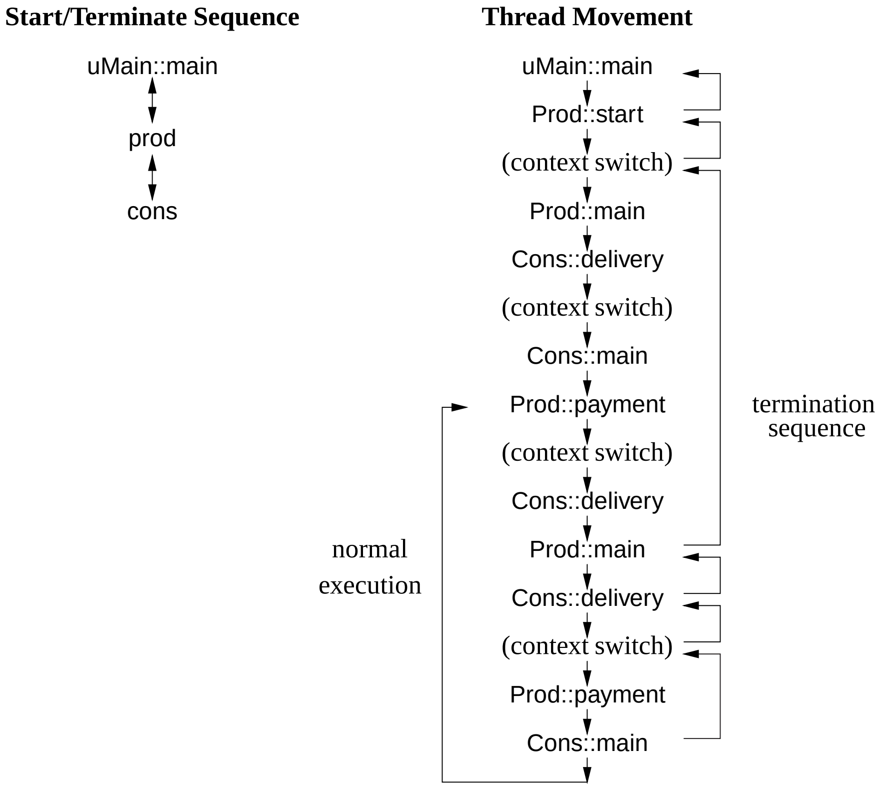


Figure 2.4: Cyclic Control Flow in Full Coroutine

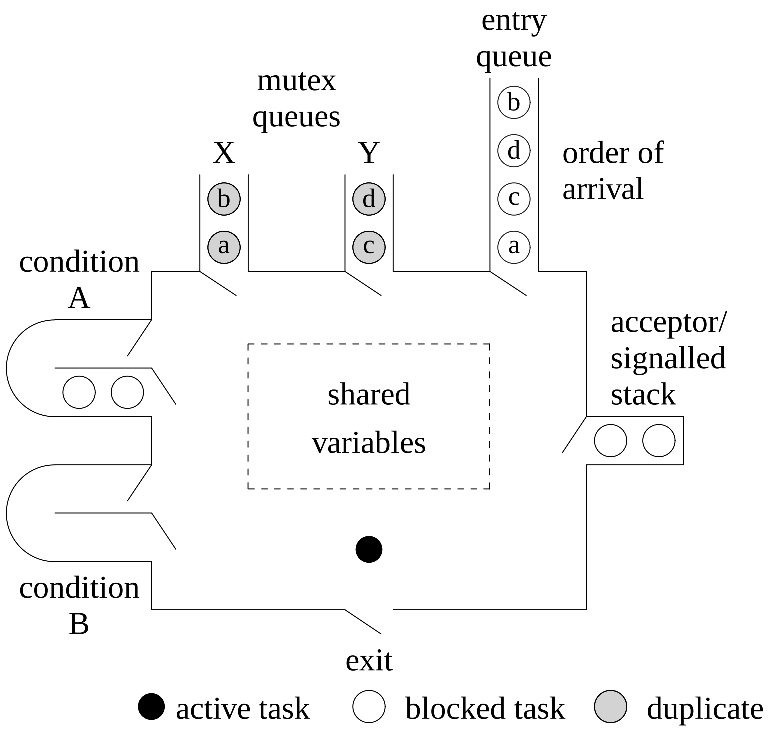


Figure 2.5: μC++ Mutex Object

2.9 Scheduling

For many purposes, the mutual exclusion that is provided automatically by mutex members is all that is needed, e.g., an atomic counter:

```
_Mutex class atomicounter {
      int cnt;
   public:
      atomicounter() { cnt = 0; }
      inc() { cnt += 1; }           // atomically increment counter
}
```

However, it is sometimes necessary to synchronize with tasks calling or executing within the mutex object forming different scheduling patterns. For this purpose, a task in a mutex object can block until a particular external or internal event occurs. At some point after a task has blocked, it must be reactivated either implicitly by the implicit scheduler (discussed next) or explicitly by another (active) task.

2.9.1 Implicit Scheduling

Implicit scheduling occurs when a mutex object becomes unlocked because the active task blocks in or exits from a mutex member. The next task to use the mutex object is then chosen from one of a number of lists associated with the mutex object. Figure 2.5 shows a mutex object with a set of tasks using or waiting to use it. When a calling task finds the mutex object locked, it is added to both the **mutex queue** of the member routine it called and the **entry queue**; otherwise it enters the mutex object and locks it. The entry queue is a list of all the calling tasks in chronological order of arrival, which is important for selecting a task when there is no active task in a mutex object. When a task in the mutex object is blocked implicitly (see Section 2.9.2) or is reactivated by another (active) task (see Section 2.9.3, p. 25), it is added to the top of the **acceptor/signalled stack**.

When a mutex object becomes unlocked, the next task to execute is selected by an **implicit scheduler**. For some of the following scheduling statements, the implicit scheduler is directed to select from a specific set of queues; hence, there is no choice with regard to which queues are examined. For other scheduling statements, the implicit scheduler may make a choice among the queues. When a choice is possible, the implicit scheduler for μC++ makes selections based on the results presented in [BFC95] to give the user the greatest possible control and produce efficient performance. These selection rules are:

1. Select tasks that have entered the mutex object, blocked, and now need to continue execution over tasks that have called and are waiting to enter.

2. When one task reactivates a task that was previously blocked in the mutex object, the restarting task always continues execution and the reactivated task continues to wait until it is selected for execution by rule 1. (signalBlock is an exception to this rule, see page 26.)

All other tasks must wait until the mutex object is again unlocked. Therefore, when selection is done implicitly, the next task to resume is not under direct user control, but is selected by the implicit scheduler.

2.9.2 External Scheduling

External scheduling controls state changes to a mutex object by scheduling calls to specified mutex members, which indirectly schedules tasks calling from *outside* the mutex object. This technique takes advantage of the entry queue to block tasks unconditionally when the mutex object is active (i.e., block outside) and the acceptor stack to block tasks conditionally that have entered the monitor (i.e., block inside). Much 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. External scheduling is accomplished with the accept statement.

2.9.2.1 Accept Statement

An **_Accept** statement dynamically chooses the mutex member(s) that executes next, which indirectly controls the next accepted caller, i.e., the next caller to the accepted mutex member(s). The simple form of the **_Accept** statement is:

```
_When ( conditional-expression )                // optional guard
      _Accept( mutex-member-name-list );
```

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. That is, the object does not exist until after the **new** routine is completed and a constructor starts; similarly, the object does not exist when **delete** is called. In all these cases, member routines cannot be called, and hence accepted, because the object does not exist or is not initialized. **_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 an **_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. □

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. A list of mutex members can be specified in an **_Accept** clause, e.g.:

```
_Accept( insert, remove );
```

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, execution continues without accepting any call; in this case, the guard is the same as an **if** statement, e.g.:

```
_When ( count == 0 ) _Accept( mem );   ≡   if ( count == 0 ) _Accept( mem );
```

Note, an accept statement with a true guard accepts only one call, regardless of the number of mutex members listed in the **_Accept** clause.

When an **_Accept** statement is executed, the acceptor is blocked and pushed on the top of the implicit acceptor/signalled stack and the mutex object is unlocked. The internal scheduler then schedules a task from the specified mutex-member queue(s), possibly waiting until an appropriate call occurs. 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. When the caller's thread exits the mutex member (or waits, as is discussed shortly), the mutex object is unlocked. Because the internal scheduler gives priority to tasks on the acceptor/signalled stack of the mutex object over calling tasks, the acceptor is popped from the acceptor/signalled stack and made ready. When the acceptor becomes active, it has exclusive access to the object. Hence, the execution order between acceptor and caller is stack order, as for a traditional routine call.

The extended form of the **_Accept** statement conditionally accepts one of a group of mutex members and then allows a specific action to be performed *after* the mutex member is called, e.g.:

```
_When ( conditional-expression )          // optional guard
    _Accept( mutex-member-name-list )
        statement                         // action
or _When ( conditional-expression )       // optional guard
    _Accept( mutex-member-name-list )
        statement                         // action
or
    . . .
        . . .
_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, the statement does nothing, 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 ( C1 ) _Accept( mem1 );            _When ( C1 ) _Accept( mem1 );
else if ( C2 ) _Accept( mem2 );       or _When ( C2 ) _Accept( mem2 );
```

The left example accepts only mem1 if C1 is true or only mem2 if C2 is true. The right example accepts either mem1 or mem2 if C1 and C2 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.

□ Generalizing the previous example from 2 to 3 accept clauses with conditionals results in the following expansion:

```
if      ( C1 && C2 && C3 ) _Accept( mem1, mem2, mem3 );
else if ( C1 && C2 ) _Accept( mem1, mem2 );
else if ( C1 && C3 ) _Accept( mem1, mem3 );
else if ( C2 && C3 ) _Accept( mem2, mem3 );
else if ( C1 ) _Accept( mem1 );
else if ( C2 ) _Accept( mem2 );
else if ( C3 ) _Accept( mem3 );
```

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. 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. □

□ Note, the syntax of the **_Accept** statement precludes the caller's argument values from being accessed in the *conditional-expression* of a **_When**. However, this deficiency is handled by the ability of a task to postpone requests (see Section 2.9.3.2, p. 26). □

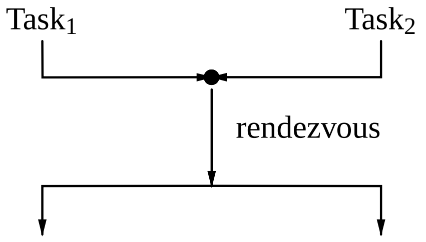
□ **WARNING:** Beware of the following difference between the **or** connector and the terminating **_Else** clause:

```
_Accept( mem1 );                 _Accept( mem1 );
or _Accept( mem2 );              _Else _Accept( mem2 );
```

The left example accepts a call to either member mem1 or mem2. The right example accepts a call to member mem1, if one is currently available; otherwise it accepts a call to member mem2. The syntactic difference is subtle, and yet, the execution is significantly different (see also Section 10.2.2, p. 130). □

2.9.2.2 Breaking a Rendezvous

The accept statement forms a **rendezvous** between the acceptor and the accepted tasks, where a rendezvous is a point in time at which both tasks wait for a section of code to execute before continuing.



The start of the rendezvous begins when the accepted mutex member begins execution and ends when the acceptor task restarts execution, either because the accepted task finishes executing of the mutex member *or the accepted task waits*. In the latter case, correctness implies sufficient code has been executed in the mutex member before the wait occurs for the acceptor to continue successfully. Finally, for the definition of rendezvous, it does not matter which task executes the rendezvous, but in μC++, it is the accepted task that executes it. It can be crucial to correctness that the acceptor know if the accepted task does not complete the rendezvous code, otherwise the acceptor task continues under the incorrect assumption that the rendezvous action has occurred. To this end, a concurrent exception is implicitly raised at the acceptor task if the accepted member terminates abnormally (see Section 5.10.3, p. 88).

2.9.2.3 Accepting the Destructor

Accepting the destructor in an **_Accept** statement is used to terminate a mutex object when it is deallocated (like the terminate clause of the select statement in Ada [Uni83, Sections 9.4, 9.7.1]). The destructor is accepted in the same way as a mutex member, e.g.:

```
for ( ;; ) {
    _Accept( ~DiskScheduler ) {           // request to terminate DiskScheduler
        break;
    } or _Accept( WorkRequest ) {         // request from disk
    } or _Accept( DiskRequest ) {         // request from clients
    } // _Accept
}
// cleanup code
```

However, the semantics for accepting a destructor are different from accepting a normal mutex member. When the call to the destructor occurs, the caller blocks immediately because a mutex object's storage cannot be deallocated if it is being used by a thread. 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 member, which allows a mutex object to cleanup before it terminates. (This semantics is the same as signal, see page 26.) Only when the caller to the destructor is popped off the acceptor/signalled stack by the internal scheduler can the destructor execute. The destructor can reactivate any blocked tasks on the acceptor/signalled stack; at this point, the task behaves like a monitor because its thread is halted.

> □ While a mutex object can always be setup so that the destructor does all the cleanup, this can force variables that logically belong in member routines into the mutex object. Furthermore, the fact that control would not return to the **_Accept** statement when the destructor is accepted seemed more confusing than having special semantics for accepting the destructor. □

Accepting the destructor can be used by a mutex object to know when to stop without having to accept a special call. For example, by allocating tasks in a specific way, a server task for a number of clients can know when the clients are finished and terminate without having to be explicitly told, e.g.:

```
{
    DiskScheduler ds;                     // start DiskScheduler task
    {
        Clients c1(ds), c2(ds), c3(ds);   // start clients, which communicate with ds
    } // wait for clients to terminate
} // implicit call to DiskScheduler′s destructor
```

2.9.2.4 Commentary

In contrast to Ada, an **_Accept** statement in μC++ places the code to be executed in a mutex member; thus, it is specified separately from the **_Accept** statement. An Ada-style accept specifies the accept body as part of the accept statement, requiring the accept statement to provide parameters and a routine body. Since we have found that having more than one accept statement per member is rather rare, our approach gives essentially the same capabilities as Ada. As well, accepting member routines also allows virtual routine redefinition, which is impossible with accept bodies. It is important to note that anything that can be done in Ada-style accept statements can be done within member routines, possibly with some additional code. If members need to communicate with the block containing the **_Accept** statements, it can be done by leaving "memos" in the mutex-type's variables. In cases where there would be several different Ada-style accept statements for the same entry, accept members would have to start with switching logic to determine which case applies. While neither of these solutions is particularly appealing, the need to use them seems to arise only rarely.

2.9.3 Internal Scheduling

A complementary approach to external scheduling is internal scheduling. Instead of scheduling tasks from outside the mutex object from the entry queue (the entry queue is still necessary), most of the scheduling occurs inside the monitor. To do scheduling *inside* the monitor requires additional queues *inside* the monitor on which tasks can block and subsequently be unblocked by other tasks. For that purpose, condition variables are provided, with an associated wait and signal statement.

2.9.3.1 Condition Variables and Wait/Signal Statements

The type uCondition creates a queue object on which tasks can be blocked and reactivated in first-in first-out order, and is defined:

```
class uCondition {
  public:
    void wait();                        // wait on condition
    void wait( uintptr_t info );        // wait on condition with information
    void signal();                      // signal condition
    void signalBlock();                 // signal condition
    bool empty() const;
    uintptr_t front() const;

    _Event WaitingFailure;
};
uCondition DiskNotIdle;
```

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.

> □ It is common to associate with each condition variable an assertion about the state of the mutex object. For example, in a disk-head scheduler, a condition variable might be associated with the assertion “the disk head is idle”. Waiting on that condition variable would correspond to waiting until the condition is satisfied, that is, until the disk head is idle. Correspondingly, the active task would reactivate tasks waiting on that condition variable only when the disk head became idle. The association between assertions and condition variables is implicit and not part of the language. □

To block a task on a condition queue, the active task in a mutex object calls member wait, e.g.,

```
DiskNotIdle.wait();
```

This statement causes the active task to block on condition DiskNotIdle, which unlocks the mutex object and invokes the internal scheduler. Internal scheduling first attempts to pop a task from the acceptor/signalled stack. If there are no tasks on the acceptor/signalled stack, the internal scheduler selects a task from the entry queue or waits until a call occurs if there are no tasks; hence, the next task to enter is the one blocked the longest. If the internal scheduling did not accept a call at this point, deadlock would occur.

When waiting, it is possible to optionally store an integer (or pointer) value with a waiting task on a condition queue by passing an argument to wait, e.g.:

```
DiskNotIdle.wait( integer-expression );
```

If no value is specified in a call to wait, the value for that blocked task is undefined. The 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. For example, the value of the front blocked task on a condition can be examined by a signaller to help make a decision about which condition variable it should signal next. This capability is useful, for example, in a problem like the readers and writer. (See Appendix D.1, p. 155 for an example program using this feature, but only after reading Section 2.10, p. 27 on monitors.) In that case, reader and writer tasks wait on the same condition queue to preserve First-In First-Out (FIFO) order and each waiting task is marked with a value for reader or writer, respectively. A task that is signalling can first check if the awaiting task at the head of a condition queue is a reader or writer task by examining the stored value before signalling.

> □ The value stored with a waiting task and examined by a signaller should not be construed as a message between tasks. The information stored 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.
>
> If this capability did not exist, it can be mimicked by creating and managing an explicit queue in the monitor that contains the values. Nodes would have to be added and removed from the explicit queue as tasks are blocked and restarted. Since there is already a condition queue and its nodes are added and removed at the correct times, it seemed reasonable to allow users to store some additional data with the blocked tasks. □

To unblock a task from a condition variable, the active task in a mutex object calls either member signal or signalBlock. For member signal, e.g.:

```
DiskNotIdle.signal();
```

the effect 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 scheduled by the internal scheduler when the mutex object is next unlocked. This semantics is *different* from the **_Accept** statement, which always blocks the acceptor; *the signaller does not block for* signal. For member signalBlock, e.g.:

```
DiskNotIdle.signalBlock();
```

the effect 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. The signalled task continues execution and the signaller is scheduled by the internal scheduler when the mutex object is next unlocked. This semantics is *like* the **_Accept** statement, which always blocks the acceptor. For either kind of signal, signalling an empty condition just continues executions, i.e., it does nothing.

> □ The **_Accept**, wait, signal and signalBlock can be executed by any routine of a mutex type. Even though these statements block the current task, they can be allowed in any member routine because member routines are executed by the caller, not the task the member is defined in. This capability is to be contrasted to Ada where waiting in an accept body would cause the task to deadlock. □

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; therefore, a condition must be checked to verify that there is a blocked task, e.g.:

```
if ( ! DiskNotIdle.empty() && DiskNotIdle.front() == 1 ) . . .
```

(This capability is discussed in detail shortly.)

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.

2.9.3.2 Commentary

The ability to postpone a request is an essential requirement of a programming language's concurrency facilities. Postponement may occur multiple times during the servicing of a request while still allowing a mutex object to accept new requests.

In simple cases, the **_When** construct can be used to accept only requests that can be completed without postponement. However, when the selection criteria become complex, e.g., when the parameters of the request are needed to do the selection or information is needed from multiple queues, it is simpler to unconditionally accept a request and subsequently postpone it if it does not meet the selection criteria. This approach avoids complex selection expressions and possibly their repeated evaluation. In addition, all the normal programming language constructs and data structures can be used in the process of making a decision to postpone a request, instead of some fixed selection mechanism provided in the programming language, as in SR [AOC$^+$88] and Concurrent C++ [GR88].

Regardless of the power of a selection facility, none can deal with the need to postpone a request after it is accepted. In a complex concurrent system, a task may have to make requests to other tasks as part of servicing a request. Any of these further requests can indicate that the current request cannot be completed at this time and must be postponed. Thus, it is essential that a request be postponable even after it is accepted because of any number of reasons during the servicing of the request. Condition variables seem essential to support this facility.

An alternative approach to condition variables is to send the request to be postponed to another (usually non-public) mutex member of the object (like Ada 95's requeue statement). This action re-blocks the request on that mutex member's entry queue, which can be subsequently accepted when the request can be restarted. However, there are problems with this approach. First, the postponed request may not be able to be sent directly from a mutex member to another mutex member because deadlock occurs due to synchronous communication. (Asynchronous communication solves this problem, but as stated earlier, imposes a substantial system complexity and overhead.) The only alternative is to use a nomutex member, which calls a mutex member to start the request and checks its return code to determine if the request must be postponed. If the request is to be postponed, another mutex member is invoked to block the current request until it can be continued. Unfortunately, structuring the code in this fashion becomes complex for non-trivial cases and there is little control over the order that requests are processed. In fact, the structuring problem is similar to the one when simulating a coroutine using a class or subroutine, where the programmer must explicitly handle the different execution states. Second, any mutex member servicing a request may accumulate temporary results. If the request must be postponed, the temporary results must be returned and bundled with the initial request that are forwarded to the mutex member that handles the next step of the processing; alternatively, the temporary results can be re-computed at the next step if that is possible. In contrast, waiting on a condition variable automatically saves the execution location and any partially computed state.

2.10 Monitor

A monitor is an object with mutual exclusion and so it can be accessed simultaneously by multiple tasks. A monitor provides a mechanism for indirect communication among tasks and is particularly useful for managing shared resources. A monitor type has all the properties of a **class**. The general form of the monitor type is the following:

```
_Mutex class monitor-name {
  private:
    . . .          // these members are not visible externally
  protected:
    . . .          // these members are visible to descendants
  public:
    . . .          // these members are visible externally
};
```

The macro name **_Monitor** is defined to be "**_Mutex class**".

2.10.1 Monitor Creation and Destruction

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

```
_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
```

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 a monitor, which can delay the termination of the block containing a monitor or the deletion of a dynamically allocated monitor.

2.10.2 Monitor Control and Communication

In μC++, both internal and external scheduling are provided, where most traditional monitors provide only internal scheduling. Figure 2.6 compares the traditional internal scheduling style using explicit condition variables to the external scheduling style using accept statements. The problem is the exchange of values (telephone numbers) between two kinds of tasks (girls and boys). (While **_Accept** allows the removal of all condition variables in this case, this is not always possible.)

<table>
<tr><th>Internal Scheduling</th><th>External Scheduling</th></tr>
<tr><td><pre>
_Monitor DatingService {
        int GirlPhoneNo, BoyPhoneNo;
        uCondition GirlWaiting, BoyWaiting;
    public:
        int Girl(int PhoneNo) {
            if (BoyWaiting.empty()) {
                GirlWaiting.wait();
                GirlPhoneNo = PhoneNo;
            } else {
                GirlPhoneNo = PhoneNo;
                BoyWaiting.signalBlock();
            }
            return BoyPhoneNo;
        } // Girl
        int Boy(int PhoneNo) {
            if (GirlWaiting.empty()) {
                BoyWaiting.wait();
                BoyPhoneNo = PhoneNo;
            } else {
                BoyPhoneNo = PhoneNo;
                GirlWaiting.signalBlock();
            }
            return GirlPhoneNo;
        } // Boy
}; // DatingService
</pre></td><td><pre>
_Monitor DatingService {
        int GirlPhoneNo, BoyPhoneNo;
    public:
        DatingService() {
            GirlPhoneNo = BoyPhoneNo = -1;
        } // DatingService
        int Girl(int PhoneNo) {
            GirlPhoneNo = PhoneNo;
            if (BoyPhoneNo == -1) {
                _Accept(Boy);
            }
            int temp = BoyPhoneNo;
            BoyPhoneNo = -1;
            return temp;
        } // Girl
        int Boy(int PhoneNo) {
            BoyPhoneNo = PhoneNo;
            if (GirlPhoneNo == -1) {
                _Accept(Girl);
            }
            int temp = GirlPhoneNo;
            GirlPhoneNo = -1;
            return temp;
        } // Boy
}; // DatingService
</pre></td></tr>
</table>

Figure 2.6: Internal versus External Scheduling

2.11 Coroutine Monitor

The coroutine monitor is a coroutine with mutual exclusion, making it safely accessible 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-state machine that is used by multiple tasks. A coroutine-monitor type has all the properties of a **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
};
```

The macro name **_Cormonitor** is defined to be "**_Mutex _Coroutine**".

2.11.1 Coroutine-Monitor Creation and Destruction

A coroutine monitor is the same as a monitor with respect to creation and destruction.

2.11.2 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. When creating a cyclic call-graph using a coroutine monitor, it is the programmer's responsibility to ensure that at least one of the members in the cycle is a **_Nomutex** member or deadlock occurs because of the mutual exclusion.

2.12 Thread Creation and Destruction

Three macros are provided for implicit thread creation and destruction.

1. Macro COBEGIN/COEND starts N thread, one to execute each nested BEGIN/END statement.

```
#include <uCobegin.h>
void uMain::main() {
    int i;
    void p1( int i ) {...}   void p2( int i ) {...}   void p3( int i ) {...}
    COBEGIN                 // thread created for each statement in block
        BEGIN i = 1; ... END
        BEGIN
            COBEGIN         // nested COBEGIN
                BEGIN p1( 3 ); ... END // order and speed of internal
                BEGIN p1( 5 ); ... END //   thread execution is unknown
            COEND           // initial thread waits for internal threads to finish
        END
        BEGIN p2( 7 ); ... END
        BEGIN p3( 9 ); ... END
    COEND                   // initial thread waits for internal threads to finish
}
```

Use recursion to create a dynamic number of threads.

```
void loop( int N ) {
    if ( N != 0 ) {
        COBEGIN
            BEGIN p1( ... ); END
            BEGIN loop( N - 1 ); END // recursive call
        COEND               // wait for return of recursive call
    }
}
```

2. Macros START/WAIT, starts a thread running in the specified routine, generating a thread handle, used to subsequently join (termination synchronization).

```
#include <uCobegin.h>
void p( int i ) {...}   int f( int i ) {...}
void uMain::main() {
    auto tp = START( p, 5 ); // thread starts in p(5)
    s1      // continue execution, do not wait for p
    auto tf = START( f, 8 ); // thread starts in f(8)
    s2      // continue execution, do not wait for f
    WAIT( tp );             // wait for p to finish
    s3
    int i = WAIT( tf );     // wait for f to finish
    s4
}
```

3. Macro COFOR(index, start, end, body) starts end - start threads, and each loop number is identified by index in the loop body.

```
#include <uCobegin.h>
void uMain::main() {
    const int rows = 10, cols = 10;
    int matrix[rows][cols], subtotals[rows], total = 0;
    // read matrix
    COFOR( row, 0, rows,    // for ( int row = 0; row < rows; row += 1 )
        subtotals[row] = 0;  // row is loop number
        for ( int c = 0; c < cols; c += 1 ) {
            subtotals[row] += matrix[row][c];
        }
    ); // wait for threads
    for ( int r = 0; r < rows; r += 1 ) {
        total += subtotals[r];  // total subtotals
    }
    cout << total << endl;
}
```

2.13 Task

A task is an object with its own thread of control and execution state, and whose public member routines provide mutual exclusion. A 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();       // starting member
  public:
    . . .              // these members are visible externally
};
```

The task type has one distinguished member, named main, in which the new thread starts execution; this distinguished member is called the **task main**. Instead of allowing direct interaction with main, its visibility is normally **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 task's global variables, called **communication variables**, which are accessible from both the task's member and main routines.

2.13.1 Task Creation and Destruction

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

```
_Task T {
    void main() . . .        // task main
  public:
    void r( . . . ) . . .
};
T *tp;                       // pointer to a T task
{ // start a new block
    T t, ta[3];              // local creation
    tp = new T;              // dynamic creation
    . . .
    t.r( . . . );            // call a member routine that must be accepted
    ta[1].r( . . . );        // call a member routine that must be accepted
    tp->r( . . . );          // call a member routine that must be accepted
    . . .
} // 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 task's execution-state and thread are created and the starting point for the new thread (activation point) is initialized to the task's main routine visible by the inheritance scope rules from the task type. After this point, the creating task executes concurrently with the new task. 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. main executes until its thread blocks or terminates.

A task terminates when its main routine terminates. When a task terminates, so does the task's thread of control. At this point, the task becomes a monitor and can still be used in that form. 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 a task is a mutex object, a block cannot terminate until all tasks declared in the block 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 area. Also, any **static** member variables declared within a task are shared among all instances of that task type. If a task makes global references or has **static** variables, there is the general problem of concurrent access to these shared variables. Therefore, it is suggested that these kinds of references be used with extreme caution.

> □ A coroutine is not owned by the task that creates it; it can be "passed" to another task. However, to ensure that only one thread is executing a coroutine at a time, the passing around of a coroutine must involve a protocol among its users, which is the same sort of protocol required when multiple tasks share a data structure. □

2.13.2 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 );
    uBaseTask( void *storage, unsigned int storageSize );
    uBaseTask( uCluster &cluster );
    uBaseTask( uCluster &cluster, unsigned int stackSize );
    uBaseTask( uCluster &cluster, void *storage, unsigned int storageSize );

    static void yield( unsigned int times = 1 );
    static uCluster &migrate( uCluster &cluster );

    uCluster &getCluster() const;
    uBaseCoroutine &getCoroutine() const;

    enum State { Start, Ready, Running, Blocked, Terminate };
    State getState() const;

    int getActivePriority();
    int getBasePriority();
};
```

The public member routines of uBaseCoroutine are inherited and have the same functionality (see Section 2.7.2, p. 15).

The overloaded constructor routine uBaseTask has the following forms:

uBaseTask() – creates a task on the current cluster with the cluster's default stack size (see uBaseCoroutine() p. page 15).

uBaseTask(**unsigned int** stackSize) – creates a task on the current cluster with the specified *minimum* stack size (in bytes) (see uBaseCoroutine(**int** stackSize) p. page 15).

uBaseTask(**void** *storage, **unsigned int** storageSize) – creates a task on the current cluster using the specified storage and *maximum* storage size (in bytes) for the task's stack
(see uBaseCoroutine(**void** *storage, **unsigned int** storageSize) p. page 15).

uBaseTask(uCluster &cluster) – creates a task on the specified cluster with that cluster's default stack size.

uBaseTask(uCluster &cluster, **unsigned int** stackSize) – creates a task on the specified cluster with the specified stack size (in bytes).

uBaseTask(uCluster &cluster, **void** *storage, **unsigned int** storageSize) – creates a task on the specified cluster using the specified storage and *maximum* storage size (in bytes) for the task's stack.

A task type can be designed to allow declarations to specify the cluster on which creation occurs and the stack size by doing the following:

```
_Task T {
  public:
    T() : uBaseTask( 8192 ) {};                    // current cluster, default 8K stack
    T( unsigned int s ) : uBaseTask( s ) {};       // current cluster and user specified stack size
    T( void *st, unsigned int s ) : uBaseCoroutine( st,s ) // current cluster and user stack storage & size
    T( uCluster &c ) : uBaseTask( c ) {};          // user cluster
    T( uCluster &c, unsigned int s) : uBaseTask( c, s ) {}; // user cluster and stack size
    T( uCluster &c, void *st, unsigned int s) : uBaseTask( c, st, s ) {}; // user cluster, stack storage & size
    . . .
};
uCluster c;                          // create a new cluster
T x, y( 16384 ), z( area1, 32768 );  // x => 8K stack, y => 16K stack, z => stack < 32K at "area1"
T q( c ), r( c, 16384 );             // q => cluster c & 8K stack, r => cluster c & 16K stack
T s( c, area2, 32768 );              // s => cluster c, stack < 32K at "area2"
```

The member routine yield gives up control of the virtual processor to another ready task the specified number of times. yield is a static member-routine that always yields the calling thread's virtual processor; use uBaseTask::yield(. . .) for a call outside of a task type. For example, the call yield(5) immediately returns control to the μC++ kernel and the next 4 times the task is scheduled for execution. If there are no other ready tasks, the yielding task is simply stopped and restarted 5 times (i.e., 5 context switches from itself to itself). 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. An example of the former situation is when a task is polling for an event, such as a hardware event. After the polling task has determined the event has not occurred, it can relinquish control to another ready task, e.g., yield(1). An example of the latter situation is when a task is creating many other tasks. The creating task may not want to create a large number of tasks before the created tasks have a chance to begin execution. (Task creation occurs so quickly that it is possible to create 100–1000 tasks before pre-emptive scheduling occurs.) If after the creation of several tasks the creator yields control, some created tasks 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; pre-emptive scheduling and/or multiple processors make this impossible.

□ When the -yield option is used, calls to yield(rand() % 3) are automatically inserted at the beginning of each member routine. □

The member routine migrate allows a task to move itself from one cluster to another so that it can access resources dedicated to that cluster's processor(s), e.g.:

```
from-cluster-reference = migrate( to-cluster-reference )
```

migrate is a static member-routine that always moves itself from one cluster to another; use uBaseTask::migrate(. . .) for a call outside of a task type.

The member routine getCluster returns the current cluster a task is executing on. The member routine getCoroutine returns the current coroutine being executed by a task or the task itself if it is not executing a coroutine.

The member routine getState returns the current state of a task, which is one of the enumerated values uBaseTask::Start, uBaseTask::Ready, uBaseTask::Running, uBaseTask::Blocked or uBaseTask::Terminate.

Two member routines are used in real-time programming (see Chapter 10, p. 127). The member routine getActivePriority returns the current active priority of a task, which is an integer value between 0 and 31. The member routine getBasePriority returns the current base priority of a task, which is an integer value between 0 and 31.

The free 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, for the current task, this reference can only be used to access the public routines of type uBaseCoroutine and uBaseTask. For example, a free routine can obtain the executing task's name or which coroutine a task is executing by performing the following:

```
uThisTask().getName();
uThisTask().getCoroutine();
```

As well, printing a task's address for debugging purposes is done like this:

```
cout << "task:" << &uThisTask() << endl;   // notice the ampersand (&)
```

2.13.3 Task Control and Communication

A task can make use of **_Accept** and uCondition variables, wait, signal and signalBlock to block and unblock tasks that enter it. Appendix D.3, p. 161 shows the archetypical disk scheduler implemented as a task that must process requests in an order other than first-in first-out to achieve efficient utilization of the disk.

2.14 Commentary

Initially, every attempt was made to add the new *μ*C++ types and statements by creating a library of **class** definitions that were used through inheritance and preprocessor macros. This approach has been used by others to provide coroutine facilities [Sho87, Lab90] and simple parallel facilities [DG87, BLL88]. However, after discovering many limitations with all library approaches, it was abandoned in favour of language extensions.

The most significant problem with all library approaches to concurrency is the lack of soundness and/or efficiency [Buh95]. A compiler and/or assembler may perform valid sequential optimizations that invalidate a correct concurrent program. Code movement, dead code removal, and copying values into registers are just some examples of optimizations that can invalidate a concurrent program, e.g., moving code into or out of a critical section, removing a timing loop, or copying a shared variable into a register making it invisible to other processors. To preserve soundness, it is necessary to identify and selectively turn off optimizations for those concurrent sections of code that might cause problems. However, a programmer may not be aware of when or where a compiler/assembler is using an optimization that affects concurrency; only the compiler/assembler writer has that knowledge. Furthermore, unless the type of a variable/parameter conveys concurrent usage, neither the compiler nor the assembler can generate sound code for separately compiled programs and libraries. Therefore, when using a concurrent library, a programmer can at best turn off all optimizations in an attempt to ensure soundness, which can have a significant performance impact on the remaining execution of the program, which is composed of large sections of sequential code that can benefit from the optimizations.

Even if a programmer can deal with the soundness/efficiency problem, there are other significant problems with attempting to implement concurrency via the library approach. In general, a library approach involves defining an abstract class, Task, which implements the task abstraction. New task types are created by inheritance from Task, and tasks are instances of these types.

On this approach, thread creation must be arranged so that the task body does not start execution until all of the task's initialization code has finished. One approach requires the task body (the code that appears in a *μ*C++ task's main) to be placed at the end of the new class's constructor, with code to start a new thread in Task::Task(). One thread then continues normally, returning from Task::Task() to complete execution of the constructors, while the other thread returns directly to the point where the task was declared. This forking of control is accomplished in the library approach by having one thread "diddle" with the stack to find the return address of the constructor called at the declaration. However, this scheme prevents further inheritance; it is impossible to derive a type from a task type if the new type requires a constructor, since the new constructor would be executed only *after* the parent constructor containing the task body. It also seems impossible to write stack-diddling code that causes one thread to return directly to the declaration point if the exact number of levels of inheritance is unknown. Another approach that does not rely on stack diddling while still allowing inheritance is to determine when all initialization is completed so that the new thread can be started. However, it is impossible in C++ (and most other object-oriented programming languages) for a constructor to determine if it is the last constructor executed in an inheritance chain. A mechanism like Simula's [Sta87] inner could be used to ensure that all initialization had been done before the task's thread is started. However, it is not obvious how inner would work in a programming language with multiple inheritance.

PRESTO (and now Java [GJSB00]) solved this problem by providing a start() member routine in class Task, which must be called after the creation of a task. Task::Task() would set up the new thread, but start() would set it running. However, this two-step initialization introduces a new user responsibility: to invoke start before invoking any member routines or accessing any member variables.

A similar two-thread problem occurs during deletion when a destructor is called. The destructor of a task can be invoked while the task body is executing, but clean-up code must not execute until the task body has terminated. Therefore, the code needed to wait for a thread's termination cannot simply be placed in Task::~Task(), because it would be executed after all the derived class destructors have executed. Task designers could be required to put the termination code in the new task type's destructor, but that prevents further inheritance. Task could provide a finish() routine, analogous to start(), which must be called before task deletion, but that is error-prone because a user may fail to call finish appropriately, for example, before the end of a block containing a local task.

Communication among tasks also presents difficulties. In library-based schemes, it is often done via message queues. However, a single queue per task is inadequate; the queue's message type inevitably becomes a union of several "real" message types, and static type checking is compromised. (One could use inheritance from a Message class, instead of a union, but the task would still have to perform type tests on messages before accessing them.) If multiple queues are used, some analogue of the Ada select statement is needed to allow a task to block on more than one queue. Furthermore, there is no statically enforceable way to ensure that only one task is entitled to receive messages from any particular queue. Hence the implementation must handle the case of several tasks that are waiting to receive messages from overlapping sets of queues. For example,

```
class TaskType : Task {
  public:
    MsgQueueType A;              // queue associated with each instance of the task
    static MsgQueueType B;       // queue shared among all instances of the task type
  protected:
    void main() {
        . . .
        _Accept i = A.front();   // accept from either message queue
        or _Accept i = B.front();
        . . .
    }
};
TaskType T1, T2;
```

Tasks T1 and T2 are simultaneously accepting from two different queues. While it is straightforward to check for the existence of data in the queues, if there is no data, both T1 and T2 block waiting for data to appear 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. Implementing this operation is expensive since the addition or removal of a message to/from a queue must be an atomic operation across all queues involved in a waiting task's accept statement to ensure that only one data item from the accepted set of queues is given to the accepting task.

If the more natural routine-call mechanism is to be used for communication among tasks, each public member routine would have to have special code at the start and possibly at the exits of each public member, which the programmer would have to provide. Other object-oriented programming languages that support inheritance of routines, such as LOGLAN'88 [CKL+88] and Beta [MMPN93], or wrapper routines, as in GNU C++ [Tie88], might be able to provide automatically any special member code. Furthermore, we could not find any convenient way to provide an Ada-like select statement without extending the language.

In the end, we found the library approach to be completely unsatisfactory. We decided that language extensions would better suit our goals by providing soundness and efficiency, greater flexibility and consistency with existing language features, and static checking.

2.15 Inheritance

C++ provides two forms of inheritance: **private** and **protected** inheritance, which provide code reuse, and **public** inheritance, which provides reuse and subtyping (a promise of behavioural compatibility). (These terms must not be confused with C++ visibility terms with the same names.)

In C++, class definitions can inherit from one another using both single and multiple inheritance. In μC++, there are multiple kinds of types, e.g., class, mutex, coroutine, and task, so the situation is more complex. The problem is that mutex, coroutine and task types provide implicit functionality that cannot be arbitrarily mixed. While there are some implementation difficulties with certain combinations, the main reason is a fundamental one. Types are written as a class, mutex, coroutine or task, and the coding styles used in each cannot, in general, be arbitrarily mixed. For example, an object produced by a class that inherits from a task type appears to be a non-concurrent object but its behaviour is concurrent. While object behaviour is a user issue, there is a significantly greater chance of problems if users casually combine types of different kinds. Table 2.3 shows the forms of inheritance allowed in μC++.

First, the case of *single* private/protected/public inheritance among homogeneous kinds of type, i.e., the kinds of the base and derived type are the same, is supported in μC++ (major diagonal in Table 2.3), e.g.:

base		*NO multiple inheritance*			
derived	struct/class	coroutine	monitor	coroutine monitor	task
struct/class	√	X	X	X	X
coroutine	√	√	X	X	X
monitor	√	X	√	X	X
coroutine monitor	√	√	√	√	X
task	√	X	√	X	√

Table 2.3: Inheritance among Type Generators

```
_Coroutine Cbase {};
_Coroutine Cderived      : private Cbase {};      // homogeneous private inheritance
_Monitor Mbase {};
_Monitor Mderived        : protected Mbase {};    // homogeneous protected inheritance
_Cormonitor CMbase {};
_Cormonitor CMderived    : public CMbase {};      // homogeneous public inheritance
_Task Tbase {};
_Task Tderived           : protected Tbase {};    // homogeneous protected inheritance
```

In this situation, all implicit functionality matches between base and derived types, and therefore, there are no problems.

Second, the case of *single* private/protected/public inheritance among heterogeneous kinds of type, i.e., the kinds of the base and derived type are different, is supported in μC++ only if the derived kind is more specific than the base kind with respect to the elementary execution properties (see Section 1.2, p. 4), e.g.:

```
class cbase {};

_Coroutine Cderived      : public cbase {};       // heterogeneous public inheritance
_Monitor Mderived        : public cbase {};       // heterogeneous public inheritance
_Cormonitor CMderived1   : private cbase {};      // heterogeneous private inheritance
_Cormonitor CMderived2   : protected Cbase {};    // heterogeneous protected inheritance
_Cormonitor CMderived3   : public Mbase {};       // heterogeneous public inheritance
_Task Tderived1          : protected cbase {};    // heterogeneous protected inheritance
_Task Tderived2          : public Mbase {};       // heterogeneous public inheritance
```

For example, a coroutine monitor can inherit from a class, a monitor, or a coroutine because the coroutine monitor has the elementary execution properties of each of these kinds of type: The only exception to this rule is between a task and coroutine because the logical use of main is completely different between these kinds of type. It seems unlikely that a task could inherit the main routine from a coroutine and have the coroutine's main perform any reasonable action with respect to the task's thread and mutex members.

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, e.g.:

```
class Queue {                                    // sequential queue
  public:
    void insert( . . . ) . . .
    virtual void remove( . . . ) . . .
};
_Mutex class MutexQueue : public Queue { // concurrent queue
    virtual void insert( . . . ) . . .
    virtual void remove( . . . ) . . .
};
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
```

However, there is a fundamental problem with non-virtual members in C++, which can cause significant confusion because non-virtual routine calls are statically bound. For example, routines Queue::insert and Queue::remove do not provide mutual exclusion because they are members of the class, while routines MutexQueue::insert and

MutexQueue::remove do provide mutual exclusion because they are members of a mutex type. Because the pointer variable qp is of type Queue, the call qp->insert calls Queue::insert even though insert is redefined in MutexQueue; so no mutual exclusion occurs. In contrast, the call to remove is dynamically bound, so the redefined routine in the monitor is invoked and appropriate synchronization occurs. The unexpected lack of mutual exclusion would cause errors. In object-oriented programming languages that have only virtual member routines, this is not a problem. The problem does not occur with private or protected inheritance because no subtype relationship is created, and hence, the assignment to qp would be invalid.

Multiple inheritance is allowed, with the restriction that at most one of the immediate base classes may be a mutex, coroutine, or task type, e.g.:

```
_Coroutine Cderived       : public Cbase, public cbase {};
_Monitor Mderived         : public Mbase, public cbase {};
_Cormonitor CMderived     : protected Cbase, public cbase {};
_Task Tderived            : public Mbase, protected cbase {};
```

Some of the reasons for this restriction are technical and some relate to the coding styles of the different kinds of type. Multiple inheritance is conceivable for the mutex property, but technically it is difficult to ensure a single root object to manage the mutual exclusion. Multiple inheritance of the execution-state property is technically difficult for the same reason, i.e., to ensure a single root object. As well, there is the problem of selecting the correct main to execute on the execution state, e.g., if the most derived class does not specify a main member, there could be multiple main members to choose from in the hierarchy. Multiple inheritance of the thread property is technically difficult because only one thread must be started regardless of the complexity of the hierarchy. In general, multiple inheritance is not as useful a mechanism as it initially seemed [Car90].

2.16 Explicit Mutual Exclusion and Synchronization

The following locks are low-level mechanisms for providing mutual exclusion of critical sections and synchronization among tasks. In general, explicit locks are unnecessary to build highly concurrent systems; the mutual exclusion provided by monitors, coroutine monitors and tasks, and the synchronization provided by **_Accept**, wait, signal and signalBlock are sufficient. Nevertheless, several low-level lock mechanisms are provided for teaching purposes and for special situations.

2.16.1 Counting Semaphore

A semaphore in μC++ is implemented as a counting semaphore as described by Dijkstra [Dij65]. A counting semaphore has two parts: a counter and a list of waiting tasks. Both the counter and the list of waiting tasks is managed by the semaphore. The type uSemaphore defines a semaphore:

```
class uSemaphore {
  public:
    uSemaphore( int count = 1 );
    void P();
    bool P( uDuration duration );
    bool P( uTime time );
    void P( uSemaphore &s );
    bool P( uSemaphore &s, uDuration duration );
    bool P( uSemaphore &s, uTime time );
    bool TryP();
    void V( int times = 1 );
    int counter() const;
    bool empty() const;
};
uSemaphore x, y(1), *z;
z = new uSemaphore(4);
```

The declarations create three semaphore variables and initializes them to the value 1, 0, and 4, respectively.

The constructor routine uSemaphore has the following form:

uSemaphore(**int** count) – this form specifies an initialization value for the semaphore counter. Appropriate values are ≥ 0. The default count is 1.

The member routines P and V are used to perform the classical 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. If P is passed a semaphore, that semaphore is Ved before Ping on the semaphore object; the two operations occur atomically. If P is passed a duration or time value, the waiting task is unblocked after that period or when the specified time is exceeded even if the task has not been Ved; this form of P returns **true** if the waiting task is Ved and **false** otherwise (meaning timeout occurred). (See Section 10.1, p. 127 for information on types uDuration and uTime.) The member routine TryP attempts to acquire the semaphore but does not block. TryP returns **true** if the semaphore is acquired and **false** otherwise. V wakes up the task blocked for the longest time 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 counter returns the value of the semaphore counter, N, which can be negative, zero, or positive: negative means abs(N) tasks are blocked waiting to acquire the semaphore, and the semaphore is locked; zero means no tasks are waiting to acquire the semaphore, and the semaphore is locked; positive means the semaphore is unlocked and allows N tasks to acquire the semaphore. The member routine empty returns **false** if there are threads 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).

To use counting semaphores in a μC++ program, include the file:

```
#include <uSemaphore.h>
```

2.16.1.1 Commentary

The wait and signal operations on conditions are very similar to the P and V operations on counting semaphores. The wait statement can block a task's execution while a signal statement can cause resumption of another task. There are, however, differences between them. The P operation does not necessarily block a task, since the semaphore counter may be greater than zero. The wait statement, however, always blocks a task. The signal statement can make ready (unblock) a blocked task on a condition just as a V operation makes ready a blocked task on a semaphore. The difference is that a V operation always increments the semaphore counter; thereby affecting a subsequent P operation. A signal statement on an empty condition does not affect a subsequent wait statement, and therefore, is lost. Another difference is that multiple tasks blocked on a semaphore can resume execution without delay if enough V operations are performed. In the mutex-type case, multiple signal statements do unblock multiple tasks, but only one of these tasks is able to execute because of the mutual-exclusion property of the mutex type.

2.16.2 Lock

A lock is either closed (0) or opened (1), and tasks compete to acquire the lock after it is released. Unlike a semaphore, which blocks tasks that cannot continue execution immediately, a lock may allow tasks to loop (spin) attempting to acquire the lock (busy wait). Locks do not ensure that tasks competing to acquire it are served in any particular order; in theory, starvation can occur, in practice, it is usually not a problem.

The type uLock defines a lock:

```
class uLock {
  public:
    uLock( unsigned int value = 1 );
    void acquire();
    bool tryacquire();
    void release();
};
uLock x, y, *z;
z = new uLock( 0 );
```

The declarations create three lock variables and initializes the first two to open and the last to closed.

The constructor routine uLock has the following form:

uLock(**int** value) – this form specifies an initialization value for the lock. Appropriate values are 0 and 1. The default value is 1.

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 spin waiting. tryacquire returns **true** if the lock is acquired and **false** otherwise. 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 simply sets the lock to opened (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).

2.16.3 Owner Lock

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. (Owner locks are used in the implementation of the non-blocking I/O stream library, see Section 4.2, p. 59). As a result, an owner lock can *only* be used for mutual exclusion, because synchronization requires the locking task to be different from the unlocking one. 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;
```

The declarations create three owner-lock variables and initializes them to open.

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).

2.16.4 Condition Lock

The condition lock is like a condition variable (see Section 2.9.3.1, p. 25), creating a queue object on which tasks 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 owner 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 type uCondLock defines a condition lock:

```
class uCondLock {
  public:
    uCondLock();
    bool empty();
    void wait( uOwnerLock &lock );
    bool wait( uOwnerLock &lock, uDuration duration );
    bool wait( uOwnerLock &lock, uTime time );
    void signal();
    void broadcast();
};
uCondLock x, y, *z;
z = new uCondLock;
```

The declarations create three condition locks and initializes them to closed (i.e., always block).

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. If wait is passed a duration or time value, the waiting task is unblocked after that period or when the specified time is exceeded even if the task has not been signalled; this form of wait returns **true** if the task is signalled and **false** otherwise (meaning timeout occurred). (See Section 10.1, p. 127 for information on types uDuration and uTime.) 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 can be safely called without acquiring any owner lock associated with tasks waiting on the condition. 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).

2.16.5 Barrier

A barrier allows *N* tasks to synchronize, possible multiple times, during their life time. Barriers are used to repeatedly coordinate a group of tasks performing a concurrent operation followed by a sequential operation. In μC++, a barrier is a mutex coroutine, i.e., **_Cormonitor**, to provide the necessary mutual exclusion and to allow code to be easily executed both before and after the *N* tasks synchronize on the barrier. The type uBarrier defines a barrier:

```
_Mutex _Coroutine 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 );
    virtual void block();
    virtual void last() {
        resume();
    }
};
uBarrier x(10), *y;
y = new uBarrier( 20 );
```

The declarations create two barrier variables and initializes the first to work with 10 tasks and the second to work with 20 tasks.

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. A 0 initialization value implies the barrier does not cause any task to wait, but each calling task runs member routine last.

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 total tasks; tasks block until any total 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.

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 (termination)
    }
}
```

Normally, the last action of the constructor for the subclass is resuming, which switches 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).

To use barriers in a μC++ program, include the file:

```
#include <uBarrier.h>
```

2.17 Atomic Instructions

The hardware provides an assortment of atomic read/write instructions, some of which are used to build the higher-level locks or lock-free data-structures. The following hardware instructions are accessible:

- **unsigned long long int** uRdtsc(); *// x86 only*

  For x86 only, the routine reads the time stamp counter register (32 or 64 bit), which contains the cycles since reset. This value can be converted into time by dividing by the processor frequency. The counters are *not* synchronized among processor, so there is significant drift over time.

- **void** uPause()

  For x86 and SPARC, the routine causes a hardware pause, which is a hint to the processor of a busy loop in progress. The pause prevents the processor pipeline filling with load/compare instructions, which takes resources (CPU), space (instruction cache), and power (heat). The test-and-set routine performs an atomic read and fixed assignment. The routine sets some location within the parameter to an implementation defined nonzero value (often the value 1) and the return value is true if and only if the previous contents had the same nonzero value. The equivalent action as a C++ routine:

  ```
  bool uTestSet( T &lock ) {      // atomic execution
      T temp = lock;              // read
      lock = 1;                   // write
      return temp == 1;           // previous value set ?
  }
  ```

- **template**< **typename** T > **bool** uTestSet(**volatile** T &lock);
  template< **typename** T > **void** uTestReset(**volatile** T &lock);

  The test-and-set routine performs an atomic read and fixed write. The routine sets some location within the parameter to an implementation defined nonzero value (often the value 1) and the return value is true if and only if the previous contents had the same nonzero value. The equivalent action as a C++ routine:

  ```
  bool uTestSet( T &lock ) {      // atomic execution
      T temp = lock;              // read
      lock = 1;                   // write
      return temp == 1;           // previous value set ?
  }
  ```

  The test-and-reset instruction instruction performs an atomic clear. The routine resets some location within the parameter to 0. It should be used conjunction with uTestSet. It can be used to construct a spinlock:

```
int Lock;                       // shared
uTestReset( Lock );             // initialize

void Task::main() {             // each task does
    while( TestSet( Lock ) );
    // critical section
    uTestReset( Lock );
}
```

- **template**< **typename** T > **static inline** T uFetchAssign(**volatile** T &loc, T repl);

  The fetch-and-assign routine generalizes the test-and-set to any value rather than a fixed value. The routine sets the parameter to the specified replacement value and returns the previous value of the assigned variable. The equivalent action as a C++ routine:

  ```
  T uFetchAssign( T &lock, T val ) { // atomic execution
      T temp = lock;              // read
      lock = val;                 // write
      return temp;                // return previous value
  }
  ```

  It can be used to construct a spinlock:

  ```
  int Lock;                       // shared
  uFetchAssign( OPEN );           // initialize

  void Task::main() {             // each task does
      while( uFetchAssign( Lock ) == CLOSED );
      // critical section
      uFetchAssign( CLOSED );
  }
  ```

- **template**< **typename** T > **inline** T uFetchAdd(**volatile** T &counter, **int** increment);

  The fetch-and-add routine performs an atomic read, add, and write. The routine adds the increment value, which can be negative, to the counter and returns the previous value of the counter variable. The equivalent action as a C++ routine:

  ```
  T uFetchAdd( T &cnt, T val ) {  // atomic execution
      T temp = cnt;               // read
      cnt += val;                 // write
      return temp;                // return previous value
  }
  ```

  It can be used to construct a spinlock:

  ```
  int Lock;                       // shared
  uFetchAssign( 0 );              // initialize

  void Task::main() {             // each task does
      while( uFetchAdd( Lock ) != 1 );
      // critical section
      uFetchAssign( 0 );
  }
  ```

- **template**< **typename** T > **inline bool** uCompareAssign(T &loc, T comp, T repl);
  template< **typename** T > **inline bool** uCompareAssignValue(T &loc, T &comp, T repl);

  The compare-and-assign routine performs an atomic read, compare, and conditional write. The first routine compares the assignment value with the comparison value, and if equal, writes the assignment variable to the replacement value. The second routine does the same as the first, but if unequal, writes the updated assignment value to the comparison variable. Both routines return true if the assignment value is equal to the comparison value, and false otherwise. The equivalent action as a C++ routine:

```
bool uCompareAssign( T &loc, T comp, T repl ) { // atomic execution
    if ( loc == comp )              // read
        loc = repl;                 // write
        return true;
    }
    return false;
}
bool uCompareAssignValue( T &loc, T &comp, T repl ) { // atomic execution
    if ( loc == comp )              // read
        loc = repl;                 // write
        return true;
    }
    comp = loc;                     // write
    return false;
}
```

It can be used to construct a spinlock:

```
int Lock;                           // shared
uFetchAssign( OPEN );               // initialize

void Task::main() {                 // each task does
    while( uCompareAssign( Lock, OPEN, CLOSED ) );
    // critical section
    uFetchAssign( CLOSED );
}
```

2.18 User Specified Context

The following facilities allow users to specify additional coroutine and task context to be saved and restored during a context switch. This facility should only be used to save and restore processor specific data, for example, coprocessor or graphics hardware data that is specific to each processor's execution. This facility does *not* allow a shared resource, like a single graphics device, to be accessed mutually exclusively by multiple tasks in a multiprocessor environment. In a multiprocessing environment, tasks executing in parallel corrupt the shared resource because their context switches overlap. To share a resource in a multiprocessor environment requires proper mutual exclusion, for example, by using a server task. In a uniprocessor environment, this facility can be used to guarantee mutual exclusion to a shared resource because only one task is executing at a time so the context of the shared resource is saved and restored on each context switch. We *strongly* discourage using this facility for mutual exclusion of a non-processor-specific resource because it does not scale to the multiprocessor environment.

The user-context facility has two parts: the definition of a context save-area, containing the storage for the context and routines to save and restore the context, and the declaration and initialization of a context save-area. The association of the additional context with a coroutine or task depends on which execution state is active when the declaration of the context save-area occurs.

A context area *must* be derived from the abstract class uContext:

```
class uContext {
  public:
    uContext();
    uContext( void *key );
    virtual void save() = 0;
    virtual void restore() = 0;
}; // uContext
```

The overloaded constructor routine uContext has the following forms:

uContext() – creates a context with a unique search key (discussed shortly).

uContext(**void** *key) – creates a context with the user supplied search key.

Multiple context areas can be declared, and hence, associated with a coroutine or task. However, a context is only associated with an execution state if its search key is unique. This requirement prevents the same context from being associated multiple times with a particular coroutine or task.

```
class CoProcessorCxt : public uContext {
    static int uUniqueKey;              // unique address across all instances
    int reg[3];                         // coprocessor has 3 integer registers
  public:
    CoProcessorCxt() : uContext( &uUniqueKey ) {};
    void save();
    void restore();
};

int CoProcessorCxt::uUniqueKey; // must be defined in only one translation unit

void CoProcessor::save() {
    // assembler code to save coprocessor registers into context area
}
void CoProcessor::restore() {
    // assembler code to restore coprocessor registers from context area
}

_Task worker {
    . . .
    void main() {
        CoProcessorCxt cpcxt;           // associate additional context with task
        . . .
    }
    . . .
};
```

Figure 2.7: Saving Co-processor Context

Figure 2.7 shows how the context of a hardware coprocessor can be saved and restored as part of the context of task worker. A unique search-key for all instances of CoProcessorCxt is created via the address of the static variable, uUniqueKey, because the address of a static variable is unique within a program. Therefore, the value assigned to uUniqueKey is irrelevant, but a value must be assigned in one translation unit for linking purposes. This address is implicitly stored in each instance of CoProcessorCxt. When a context is added to a task, a search is performed for any context with the same key. If a context with the same key is found, the new context is not added; otherwise it is added to the list of user contexts for the task.

□ **WARNING:** Put no code into routines save and restore that results in a context switch, e.g., printing using cout or cerr (use write if necessary). These routines are called during a context switch, and a context switch cannot be recursively invoked. □

2.18.1 Predefined Floating-Point Context

In most operating systems, the entire state of the actual processor is saved during a context switch between execution states because there is no way to determine if a particular object is using only a subset of the actual processor state. All programs use the fixed-point registers, while only some use the floating-point registers. Because there is a significant execution cost in saving and restoring the floating-point registers, they are not saved automatically. If a coroutine or task performs floating-point operations, saving the floating-point registers must become part of the context-switching action for the execution state of that coroutine or task.

To save and restore the float-point registers on a context switch, declare a single instance of the predefined type uFloatingPointContext in the scope of the floating-point computations, such as the beginning of the coroutine's or task's main member, e.g.:

```
_Coroutine C {
    void main() {
        uFloatingPointContext fpcxt; // the name of the variable is insignificant
        . . .  // floating-point computations can be performed safely in this scope
    }
    . . .
};
```

Once main starts, both the fixed-point and floating-point registers are restored or saved during a context switch to or from instances of coroutine C.

> □ **WARNING:** The member routines of a coroutine or task are executed using the execution state of the caller. Therefore, if floating-point operations occur in a member routine, including the constructor, the caller must also save the floating-point registers. Only a coroutine's or task's main routine and the routines called by main use the coroutine's or task's execution state, and therefore, only these routines can safely perform floating-point operations. □

> □ **WARNING:** Some processors, like the SPARC, implicitly save both fixed and floating-point registers, which means it is unnecessary to create instances of uFloatingPointContext in tasks performing floating-point operations. However, leaving out uFloatingPointContext is dangerous because the program is not portable to other processors. Therefore, it is important to always include an instance of uFloatingPointContext in tasks performing floating-point operations. For processors like the SPARC, uFloatingPointContext does nothing, so there is no cost. □

Additional context can be associated with a coroutine or task in a free routine, member routine, or as part of a class object to temporarily save a particular context. For example, the floating-point registers are saved when an instance of the following class is declared:

```
class c {
  private:
    uFloatingPointContext fpcxt;
  public:
    void func() {
        // perform floating-point computations
    }
};
```

When a coroutine or task declares an instance of c, its context switching is augmented to save the floating-point registers for the duration of the instance. This capability allows the implementor of c to ensure that the integrity of its floating-point calculations are not violated by another coroutine or task performing floating-point operations. It also frees the user from having to know that the floating-point registers must be saved when using class c. Remember, if the floating-point registers are already being saved, the additional association is ignored because of the unique search key.

2.19 Implementation Restrictions

The following restrictions are an artifact of this implementation. In some cases the restriction results from the fact that μC++ is only a translator and not a compiler. In all other cases, the restrictions exist simply because time limitations on this project have prevented it from being implemented.

- While μC++ has extended C++ with concurrency constructs, it is not a compiler. Therefore, it suffers from the soundness/efficiency problem related to all concurrency library approaches (see Section 2.14, p. 33). However, every attempt is made to ensure μC++ does generate sound code.

- Some runtime member routines are publicly visible when they should not be; therefore, μC++ programs should not contain variable names that start with a "u" followed by a capital letter. This problem is an artifact of μC++ being a translator.

- By default, μC++ allows at most 128 mutex members because a 128-bit mask is used to test for accepted member routines. When μC++ is compiled, this value can be modified by setting the preprocessor variable __U_MAXENTRYBITS__.

  Unfortunately, bit masks, in general, do not extend to support multiple inheritance. We believe the performance degradation required to support multiple inheritance is unacceptable.

- When defining a derived type from a base type that is a task or coroutine and the base type has default parameters in its constructor, the default arguments must be explicitly specified if the base constructor is an initializer in the definition of the constructor of the derived type, e.g.:

```
_Coroutine Base {
  public:
    Base( int i, float f = 3.0, char c = 'c' );
};

_Coroutine Derived : public Base {
  public:
    Derived( int i ) : Base( i, 3.0, 'c' );       // values 3.0 and 'c' must be specified
};
```

All other uses of the constructor for Base are *not* required to specify the default values. This problem is an artifact of *µ*C++ being a translator.

- Anonymous coroutine and task types are not supported, e.g.:

```
_Task /* no name */ {          // must have a name
    . . .
} t1, t2, t3;
```

Both a coroutine and a task must have a constructor and destructor, which can only be created using the name of the type constructor. Having the translator generate a hidden unique name is problematic because the order of include files may cause the generation of a different name for different compilations, which plays havoc with linking because of name mangling.

- There is no discrimination mechanism in the **_Accept** statement to differentiate among overloaded mutex member routines. When time permits, a scheme using a formal declarer in the **_Accept** statement to disambiguate overloaded member routines will be implemented, e.g.:

```
_Accept( mem(int) );
or _Accept( mem(float) );
```

Here, the overloaded member routines mem are completely disambiguated by the type of their parameters because C++ overload resolution does not use the return type.

- A **try** block surrounding a constructor body is not supported, e.g.:

```
class T2 : public T1 {
    const int i;
  public:
    T2();           // constructor
};
T2::T2() try : T1(3), i(27) {
    // body of constructor
} catch {
    // handle exceptions from initialization constructors (e.g., T1)
}
```

This problem is an artifact of *µ*C++ being a translator.

- No **break**, **continue**, **goto**, **return** from a resumption handler. This problem is because the handler is transformed into a local lambda routine, and hence, it cannot branch back within its lexical scope.

Chapter 3

Asynchronous Communication

Parallelism occurs when multiple threads execute simultaneously to decrease a program's execution, i.e., the program takes less real (wall-clock) time to complete a computation. The computation must be divided along some dimension(s) and these subdivisions are executed asynchronously by the threads. The decrease in execution time is limited by the number of these subdivisions that can be executed simultaneously (Amdahl's law [Amd67]).

Every practical concurrent program involves some communication among threads. One thread may communicate with another in order to provide inputs (arguments), and/or to receive the output (results) produced by the other thread. If the thread providing the inputs is the same thread that later receives the output, then the communication pattern is analogous to a sequential routine call, where one routine provides arguments to another and receives the result. A call by one thread to a **_Mutex** member of a task is an example of this communication pattern. Such a call is known as a **synchronous call** because the two tasks must synchronize in order to pass the arguments from caller to callee, and because the caller remains blocked until the callee returns the result.

While a synchronous call is simple and useful, it may limit parallelism because the caller task is forced to block until the result is returned. In some cases there is a subdivision of the computation that the caller task can perform while the callee task is computing the caller's result. In such a case, it is more appropriate to use an **asynchronous call**. An asynchronous call can be thought of as two synchronous calls, one to provide the inputs and a second one to receive the output, e.g.:

```
callee.start( arg );              // provide arguments
// caller performs other work asynchronously
result = callee.finish();         // obtain result
```

Here, the call to start returns as soon as the arguments are transferred from caller to callee. Computation then proceeds for both the caller and callee, concurrently. In an asynchronous call, the caller and callee are known as the **client** and **server**, respectively. Note, the client may still have to block (or poll) at the call to finish, if the server has not yet finished the client's computation. The amount of parallelism that can be obtained in this way depends on the amount of concurrent computation that can be done by the client and server. If there is little concurrency possible, then the overhead of two synchronous calls and creating the server outweighs the benefits gained by any potential parallelism, and a single synchronous call is sufficient.

A client may also have to block when calling the start method to transmit arguments, if the server is performing some other operation at the time of the call. If the server only handles one outstanding asynchronous call at a time from one client task it should always be ready to receive and respond to the start method immediately, minimizing blocking time for the client. Depending on the application it may be necessary to have a more complicated server, one that can manage multiple outstanding asynchronous calls from multiple clients simultaneously. Constructing a server that can handle calls efficiently while minimizing blocking time for clients generally requires additional buffering of arguments and results. Different designs for servers are discussed in Section 3.3, p. 54.

3.1 Futures

The major problem with the above approach for asynchronous call is the two-step protocol: start the call with arguments and finish the call to collect a result. In general, protocols are error prone because the caller may not obey them (e.g., never retrieve a result, try to retrieve the result twice, etc.).[1] A **future** is an abstraction that attempts to hide some of the details involved in an asynchronous call, in particular buffering and retrieving the return value. The previous two synchronous calls are transformed into a single explicit synchronous call and an implicit second synchronous call when the future is accessed:

```
future = callee.work( arg );        // provide arguments and get future result
// perform other work asynchronously
i = future() + . . .;               // obtain actual result, may block if result not ready
```

In general, a future is generic in the type of the return value and acts as a surrogate for this value. Instead of making two calls to send arguments and then retrieve the result, a single call is made and a future representing the result is returned immediately. The client continues execution as if the call had returned an actual result. The future is filled in at some later time (in the "future"), after the server calculates the result. If the client tries to use the future before a result is inserted, the client implicitly blocks and is subsequently unblocked by the server after it insertions a result in the future. Hence, there is no explicit protocol between client and server to retrieve a result; the protocol is implicit within the future.

*μ*C++ provides two forms of futures, which differ in their storage-management interface. The explicit-storage-management future (Future_ESM) must be allocated and deallocated explicitly by the client. The implicit-storage-management future (Future_ISM) automatically allocates required storage and automatically frees the storage when the future is no longer in use. The advantage of Future_ESM is that it allows the programmer to choose the method of allocation, whether on the heap, on the stack, or statically, which can result in more predictable and efficient allocation compared to Future_ISM, which always allocates storage on the heap. The disadvantage of Future_ESM is that the client must ensure that the future is deallocated, but not before the server thread has inserted the result (or the operation has been cancelled).

There is a basic set of common operations available on both types of futures. These consist of client operations, used by a client task to retrieve the return value, and server operations, used by a server task to fill in the value.

3.1.1 Client Operations

The future operations available to a client are:

available – returns **true** if the asynchronous call has completed and **false** otherwise. Note, the call can complete because a result is available, because the server has generated an exception, or because the call has been cancelled (through the cancel method, below).

operator() – (function call) returns a copy of the future result. The client blocks if the future result is currently unavailable. If an exception is returned by the server, that exception is thrown. A future result can be retrieved multiple times by any task until the future is reset or destroyed.

operator T – (conversion to type T) returns a copy of the future result. ***This kind of access must be performed only after a blocking access, or after a call to available returns true.*** This operation is a low-cost way of accessing a future result *after* the result is known to have been delivered. As with **operator**(), if the server returns an exception, that exception is thrown.

cancelled – returns **true** if the future is cancelled and **false** otherwise.

cancel – attempts to cancel the asynchronous call associated with the future. All clients waiting for the result are unblocked, and an exception of type Future_ESM::Cancellation is thrown at any client attempting to access the result. Depending on the server, this operation may also have the effect of preventing the requested computation from starting, or it may interrupt the computation in progress.

3.1.2 Server Operations

The future operations available to a server are:

delivery(T result) – copy the server-generated result into the future, unblocking any clients that are waiting for the result. This result is the value returned to the client. Returns **true** if the result is copied and **false** if the asynchronous call has already completed.

[1] Approaches for asynchronous call involving tickets and/or call backs both require an explicit protocol to retrieve a result.

```
template<typename T, typename ServerData> _Monitor Future_ESM {
  public:
    Future_ESM();

    // used by client
    _Nomutex bool available();              // future result available ?
    T operator()();                         // access result, possibly having to wait
    _Nomutex operator T();                  // cheap access of result after waiting

    _Event Cancellation {};                 // raised if future cancelled
    _Nomutex bool cancelled();              // future result cancelled ?
    void cancel();                          // cancel future result

    // used by server
    ServerData serverData;                  // information needed by server

    bool delivery( T result );              // make result available in the future
    void reset();                           // mark future as empty (for reuse)
    bool exception( uBaseEvent *ex );       // make exception available in the future
};
```

Figure 3.1: Future : Explicit Storage Management

reset – mark the future as empty so it can be reused, after which the current future value is no longer available.

exception(uBaseEvent *cause) – copy a server-generated exception into the future. All clients waiting for the result are unblocked, and the exception cause is thrown at any client attempting to access the result. Returns **true** if the exception is copied and **false** if the asynchronous call has already completed.

```
_Event E {};
Future_ISM<int> result;
result.exception( new E );    // exception deleted by future
```

exception deleted by reset or when future deleted

A server may require storage to buffer call arguments and other data needed for cancellation of futures. This storage is allocated as part of the future; hence, the future may also be generic in the type of server-management data. A server exports this type information for use with a future (see Section 3.3, p. 54).

Future cancellation affects the server computing the future's value. Depending on the server, cancellation may prevent the requested computation from starting, or it may interrupt the computation in progress. In both cases, the server does not insert a result into the future. If the server computation cannot be interrupted, the server may deliver a result even though the future has been cancelled.

An ESM future's cancel member cannot return until it is known that the server no longer references the cancelled future because the future's storage may be deallocated. Therefore, the server must inform the future if it will or will not not deliver a value, by supplying a member in the ServerData type with the following interface:

```
bool cancel();
```

It returns **true** if the result of the asynchronous call will not be delivered to the future, and hence the server computation has been interrupted, and **false** otherwise.

An ISM future allows server-specific data to be included in the future through a special constructor parameter, which must implement a similar cancel member. However, no action need be taken by the ISM server, since it is always safe for the client to delete its copy of the future. In this case, the cancel method is purely advisory, allowing the server to avoid unnecessary computation.

3.1.3 Explicit Storage Management

The explicit storage-management (ESM) future (see Figure 3.1) makes the client responsible for storage management by preallocating the future and passing it as an argument to the asynchronous call.

This kind of future is *not* copyable, i.e., no assignment or pass-by-value is allowed. Both client and server must operate on the same future, since the future is the buffer through which the result is returned. To copy the future would be to create two buffers; a client could wait for a value to be delivered to one buffer, whereas the server could deliver

```
template<typename T> class Future_ISM {
  public:
    Future_ISM();
    Future_ISM( ServerData *serverData );

    // used by client
    bool available();                      // future result available ?
    T operator()();                        // access result, possibly having to wait
    operator T();                          // cheap access of result after waiting

    _Event Cancellation {};                // raised if future cancelled
    bool cancelled();                      // future result cancelled ?
    void cancel();                         // cancel future result

    bool equals( const Future_ISM<T> &other ); // equality of reference

    // used by server
    bool delivery( T result );             // make result available in the future
    void reset();                          // mark future as empty (for reuse)
    bool exception( uBaseEvent *ex );      // make exception available in the future
};
```

Figure 3.2: Future : Implicit Storage Management

the future to the other buffer. As a result it is necessary to pass the future by pointer or by reference. It is possible for many threads to wait on the same future, so long as each uses a pointer or reference.

It is the client's responsibility to ensure a future continues to exist after a call as long as it is possible for the server to deliver a return value or exception. It is safe to delete the future after **operator**() returns, when available returns **true**, or after cancel returns. Note that because of this guarantee provided by cancel, it may be the case that cancel blocks until the server task acknowledges the cancellation.

3.1.4 Example

This example illustrates how a client uses a number of futures to communicate asynchronously with a server:

```
Server server;                            // server thread to process async call
Future_ESM<int, Server::IMsg> f[10];      // created on the stack
for ( int i = 0; i < 10; i += 1 ) {       // start a number of calls
    server.mem( f[i], i, 'c' );           // async call
}
// work asynchronously while server processes requests
for ( int i = 0; i < 10; i += 1 ) {       // retrieve async results
    osacquire( cout ) << f[i]() << " " << // may block on first attempt to retrieve value
                     f[i] << endl;        // use value again (cheap access)
}
```

The client creates an array of N futures for **int** values. In general, these futures can appear in any context requiring an **int** value and are used to make N asynchronous calls to the server. For each call to server.mem, a future is passed, in which the server returns a result, along with appropriate arguments, which are used by the server to perform the computation. The client then proceeds asynchronously with the server to perform other work, possibly in parallel with the server (if running multiprocessor). Finally, the client retrieves the results from the server by first performing a blocking access to each future. After that future is retrieved, it can be retrieved again using the cheap nonblocking-form (or the expensive blocking-form, but there is no point in synchronizing more than once for each asynchronous call.)

The key point for explicit futures is that the client preallocates the future storage so the server does not perform any dynamic memory-allocation for the futures, which can provided a substantial performance benefit. In the example, the client is able to use low-cost stack storage for the futures needed to interact with the server.

3.1.5 Implicit Storage Management

The implicit storage-management (ISM) future (see Figure 3.2) simplifies the future interface relative to Future_ESM by automatically managing the storage required for the asynchronous call.

Unlike the ESM future, an ISM future *is* copyable, i.e., both assignment and pass-by-value are allowed. The ISM future functions as a “handle” or smart pointer [Ale01] that refers to the result value. Any copy of an ISM future refers to the same result value as the original. Although ISM futures may be allocated on the stack, on the heap, or statically, the underlying storage for the result value (and possibly for server-management data as well) is always implicitly allocated on the heap. This storage is freed when all futures referring to that value are destroyed.

Server-specific data (see Section 3.1.2, p. 48) can be passed to an ISM future via its constructor.

Occasionally it is useful to know whether two futures refer to the result of the same asynchronous call. For this reason, Future_ISM has one member not found in Future_ESM. The member routine equals returns **true** if the argument future refers to the same asynchronous call as this future and **false** otherwise.

3.1.6 Example

This example uses ISM futures in the same way the previous example used ESM futures:

```
Server server;                              // server thread to process async call
Future_ISM<int> f[10];                      // created on the stack, but also uses heap
for ( int i = 0; i < 10; i += 1 ) {         // start a number of calls
    f[i] = server.mem( i, 'c' );            // async call
}
// work asynchronously while server processes requests
for ( int i = 0; i < 10; i += 1 ) {         // retrieve async results
    osacquire( cout ) << f[i]() << " " <<   // may block on first attempt to retrieve value
                         f[i] << endl;      // use value again (cheap access)
}
```

Note that the asynchronous call to the server has the future as its return value, resembling a traditional return call, unlike the ESM future. Also, an ISM future allows the internal server-management data to be hidden from the client.

3.2 Future Access

After a client has created a future, passed it to a server, and then continued asynchronously, it normally accesses the future to retrieve its value. The simplest way to access a future is to call its **operator**() member. If the client’s computation is reasonably structured, this approach may provide good asynchrony, with only occasional short-blocking because the future’s value is unavailable. However, asynchrony can be curtailed if the client accesses a future too early and blocks when it could do other work. A more complicated way of accessing a future is to check, using the available method, whether the future is accessible before performing a potentially blocking access operation.

When a client creates multiple futures, and correspondingly makes multiple asynchronous calls using these futures, neither of previous approaches may be satisfactory. The client may only be able to proceed when some combination of the future results are available, or it may be able to proceed in different ways depending on the order in which results become available. Although it is possible to use the available method to check for accessibility of a set of futures, it is impossible to use available to wait on a future set without polling the futures (busy waiting). Hence, a more complex future-selection mechanism is necessary. This mechanism can be divided into two basic forms: heterogeneous and homogeneous.

heterogeneous: In this case, there are a number of futures that may have different types. Complicated selection conditions are constructed by naming individual futures in expressions. This style of selection provides great flexibility, but does not scale to large numbers of futures.

homogeneous: In this case, there are a number of futures of related types. The set of futures are stored together in a data structure like a container or array, and hence, must have some notion of common type. Two common selection operations on the futures within the data structure are wait-for-any and wait-for-all, i.e., wait for the first future in the set to becomes available, or wait for all futures in the set to become available. This style of selection is practical for large numbers of futures, but lacks the flexibility of heterogeneous selection.

3.2.1 Select Statement

μC++ provides a select statement to handle heterogeneous future selection by waiting for one or more available futures based on a logical selection-criteria. The simplest form of the select statement has a single **_Select** clause, e.g.:

_Select(*selector-expression*);

The selector-expression must be satisfied before execution continues. When the selector-expression consists of a reference to a single future, the expression is satisfied if and only if the future is available. For example, in:

```
_Select( f1 );
```

the selector becomes select blocked until f1.available() is true. This select is not equivalent to calling the future access-operator (f1()), which also gets the value, which can throw an exception. More interesting is when multiple futures appear in a compound selector-expression, where the futures are related using logical operators || and && to specify a compound selection criteria, e.g.:

```
_Select( f1 || f2 && f3 );
```

Normal operator precedence applies so the expression is implicitly parenthesized as: (f1 || (f2 && f3)). Execution waits until either future f1 is available or both futures f2 and f3 are available. Hence, for any selector-expression containing an || operator, some futures in the expression may be unavailable after the selector-expression is satisfied. For example, in the above selection expression, if future f1 becomes available, neither, one or both of f2 and f3 may be available.

A **_Select** clause may be guarded with a logical expression, e.g.:

```
_When ( conditional-expression ) _Select( f1 );    ≡    if ( conditional-expression ) _Select( f1 );
```

The selector task is select blocked while the guard is true and there is no available future. A **_When** guard is considered true if it is omitted or if its *conditional-expression* evaluates to non-zero. If the guard is false, execution continues without waiting for any future to become available; for this example, the guard is the same as an **if** statement. Note, a simple select-statement always waits until at least one future is available unless its guard is false.

The complex form of the select statement conditionally executes a specific action *after* each selector-expression evaluates to true (see select-statement grammar in Chapter A, p. 145 for complete syntax), e.g.:

```
_Select( selector-expression )
    statement                          // action
```

After the selector-expression is satisfied, the action statement is executed; in this case, the action could simply follow the select statement. However, the complex form of the select statement allows relating multiple **_Select** clauses using keywords **or** and **and**, each with a separate action statement. The **or** and **and** keywords relate the **_Select** clauses in exactly the same way operators || and && relate futures in a select-expression, including the same operator precedence; parentheses may be used to specify evaluation order. For example, the previous select statement with a compound selector-expression can be rewritten into its equivalent complex form with actions executed for each future that becomes available (superfluous parentheses show precedence of evaluation):

```
(                                      // superfluous parentheses
  _Select( f1 )
        statement-1                    // action
  or (                                 // superfluous parentheses
          _Select( f2 )                // optional guard
               statement-2             // action
          and _Select( f3 )            // optional guard
               statement-3             // action
     ) // and
) // or
```

The original selector-expression is now three connected **_Select** clauses, where each **_Select** clause has its own action. During execution of the statement, each **_Select**-clause action is executed when its sub-selector-expression is satisfied, i.e., when each future becomes available; however, control does not continue until the selector-expression associated with the entire statement is satisfied. For example, if f2 becomes available, statement-2 is executed but the selector-expression associated with the entire statement is not satisfied so control blocks again. When either f1 or f3 become available, statement-1 or 3 is executed, and the selector-expression associated with the entire statement is satisfied so control continues. For this example, within the action statement, it is possible to access the future using the non-blocking access-operator since the future is known to be available.

An action statement is triggered only once for its selector-expression, even if the selector-expression is compound. For example, in:

```
_Select( f1 || f2 )
    statement-1
and _Select( f3 )
    statement-2
```

statement-1 is only executed once even though both futures f1 and f2 may become available while waiting for the selector-expression associated with the entire statement to become satisfied. Also, in statement-1, it is unknown which of futures f1 or f2 satisfied the sub-selector-expression and caused the action to be triggered; hence, it is necessary to check which of the two futures is available.

Note, a complex select-statement with **_When** guards is not the same as a group of connected **if** statements, e.g.:

```
if ( C1 ) _Select( f1 );              _When ( C1 ) _Select( f1 );
else if ( C2 ) _Select( f2 );         or _When ( C2 ) _Select( f2 );
```

The left example waits for only future f1 if C1 is true or only f2 if C1 is false and C2 is true. The right example waits for either f1 or f2 if C1 and C2 are true. Like the **_Accept** statement, it takes $2^N - 1$ **if** statements to simulate a compound **_Select** statement with N **_When** guards (see p. page 23).

Finally, a select statement can be made non-blocking using a terminating **_Else** clause, e.g.:

```
_Select( selector-expression )
    statement                              // action
_When ( conditional-expression ) _Else    // optional guard & terminating clause
    statement                              // action
```

The **_Else** clause *must* be the last clause of a select statement. If its guard is true or omitted and the select statement is not immediately true, then the action for the **_Else** clause is executed and control continues. If the guard is false, the select statement blocks as if the **_Else** clause is not present. (See Section 10.2.3, p. 131 for timeout with **_Select**.)

3.2.2 Wait Queue

μC++ provides two data structures to handle homogeneous future selection. As with the two future types, they have similar behaviour but different approaches to storage management.

```
                    ESM                          |                     ISM
template< typename Selectee >                    | template< typename Selectee >
class uWaitQueue_ESM {                           | class uWaitQueue_ISM {
  public:                                        |   public:
    uWaitQueue_ESM();                            |     uWaitQueue_ISM();
    template< typename Iterator >                |     template< typename Iterator >
        uWaitQueue_ESM( Iterator begin, Iterator end ); |   uWaitQueue_ISM( Iterator begin, Iterator end );
    bool empty() const;                          |     bool empty() const;
    void add( Selectee *n );                     |     void add( Selectee n );
    template< typename Iterator >                |     template< typename Iterator >
        void add( Iterator begin, Iterator end );|         void add( Iterator begin, Iterator end );
    void remove( Selectee n );                   |     void remove( Selectee n );
    Selectee *drop();                            |     Selectee drop();
};                                               | };
```

To use uWaitQueue_ISM, futures are added to the queue at construction or using the add methods, and are removed using the drop method as each becomes available. uWaitQueue_ESM is similar, except it operates on future pointers. For uWaitQueue_ESM, the client must ensure added futures remain valid, i.e., their storage persists, as long as they are in a uWaitQueue_ESM. For uWaitQueue_ISM, the added futures must be copyable, so ISM futures can be used but not ESM futures; uWaitQueue_ESM is the only queue that can be used with ESM futures.

The operations available on both kinds of queue are:

uWaitQueue_ISM() / uWaitQueue_ESM() – constructs an empty queue.

uWaitQueue_ISM(Iterator begin, Iterator end) / uWaitQueue_ESM(Iterator begin, Iterator end) – constructs a queue, adding all of the futures in the range referenced by the iterators begin and end (inclusive of begin, but exclusive of end). For the ESM queue, it is pointers to the futures that are added to the queue.

empty – returns true if there are no futures in the queue, false otherwise.

add(Selectee n) – adds a single future to the queue (ISM).

add(Selectee *n) – adds a single pointer to a future to the queue (ESM).

add(Iterator begin, Iterator end) – adds all futures in the range given by the iterators begin and end (inclusive of begin, but exclusive of end). For the ESM queue, it is pointers to the futures that are added to the queue.

remove(Selectee n) – removes any futures in the queue that refer to the same asynchronous call as n (ISM).

remove(Selectee *n) – removes any occurrence of the future pointer n from the queue (ESM).

drop – returns an available future from the queue, removing it from the queue. The client blocks if there is no available future. If multiple futures are available, one is chosen arbitrarily to return; other available futures can be obtained by further calls to drop. Calling drop on an empty ISM queue is an error; calling drop on an empty ESM queue returns NULL.

The drop method is an example of "wait-any" semantics in homogeneous selection: execution blocks until at least one future is available. To provide "wait-all" semantics, where execution only continues when all futures are available, a simple loop suffices:

```
uWaitQueue_ISM<Future_ISM<int> > queue; // or ESM
// add futures to queue
while ( ! queue.empty() ) {                    // wait for all futures to become available
    queue.drop();
}
```

Other semantics, such as "wait-*n*" (block until *n* futures are available), can be obtained using more complex control logic. Indeed, it is possible to use wait queues to simulate some forms of the **_Select** statement:

```
                          | uWaitQueue_ISM<Future_ISM<int> > queue;
                          | queue.add( f1 ); queue.add( f2 ); queue.add( f3 );
                          | for ( ;; ) {
                          |     queue.drop();
_Select( f1 )             |     if ( f1.available() ) {
    statement-1           |         statement-1;  break;
or _Select( f2 && f3 )    |     } else if ( f2.available() && f3.available() ) {
    statement-2           |         statement-2;  break;
                          |     }
                          | }
```

However, for more complex selection, the complexity of the simulation grows faster than the complexity of the equivalent **_Select** statement. Furthermore, the **_Select** statement allows for different types of futures (including both ESM and ISM futures) to be mixed in a single selection, whereas the futures in a uWaitQueue must all have the same type.

3.3 Servers

A server performs a computation on behalf of a client allowing the client to execute asynchronously until it needs the result of the computation. Figure 3.3 shows three basic organizational structures for servers, from simple to complex (top to bottom). The top structure is the simplest, where a single client uses a direct asynchronous call to pass a future to the server for computation and retrieves the result of this computation from the future before passing another future (one-to-one relationship between client and server). This structure ensures the single client cannot block on the asynchronous call because it synchronizes with the server when it accesses the future, so the server should always be available to receive arguments for the next call. However, this structure may result in the server spending most of its time blocked if the single client does significant additional computation (such as processing the future result) before making the next call. Note, attempting to increase the server's work by sending multiple futures produces no additional asynchrony because the server cannot accept these calls while it is working nor does it have any place to store the additional arguments for subsequent processing.

To mitigate server blocking, a server must be restructured to support multiple asynchronous calls while it is working. This approach allows one or more clients (many-to-one relationship between clients and server) to make one or more asynchronous calls, supplying the server with more work to keep it from blocking. Two key changes are required. First, a server must provide a request buffer to store arguments for multiple asynchronous calls. Second, the server must poll periodically for new asynchronous calls while it is working, otherwise clients block attempting to insert requests into the buffer until their call is accepted. This latter requirement is necessary because the buffer is a shared resource that requires mutual exclusion, i.e., clients add to the buffer and the server removes from the buffer.

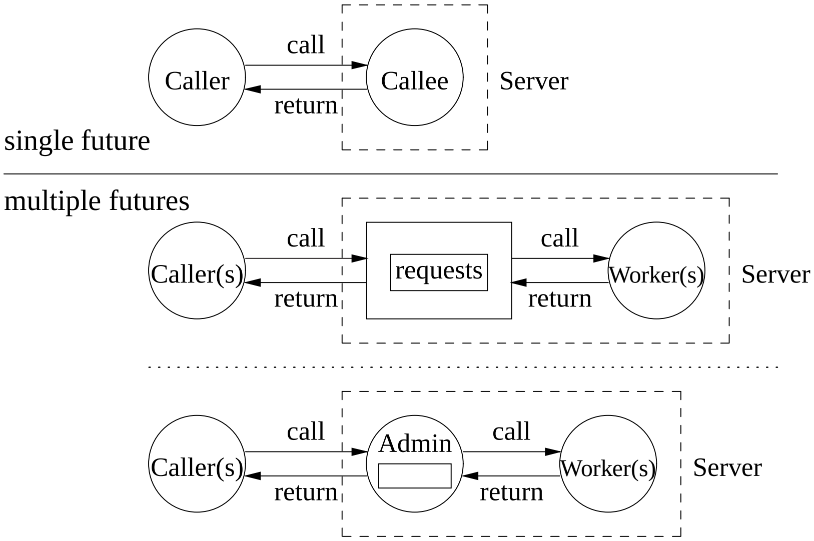


Figure 3.3: Basic Server Structures

However, polling can obscure server code and polling frequency is always an issue. The only way to remove polling is to separate the buffer's mutual-exclusion from the server's.

The middle structure in Figure 3.3 handles multiple asynchronous calls by transforming the direct communication between client and server into indirect communication by composing the server as one of more worker tasks to perform computations and a monitor buffering future requests from asynchronous calls between client(s) and worker(s). A client places arguments into the input buffer along with a future to return a result, and then continues. A worker removes arguments from the input buffer for computation and places the result of the computation into the supplied future; inserting the result implicitly unblocks any waiting client(s) attempting access to the future. Clients may block if there is contention on accessing the buffer or the buffer is full; workers may block if there is contention on accessing the buffer or the buffer is empty. Any buffer management is performed by the client and/or worker when manipulating the buffer(s).

The bottom structure in Figure 3.3 transforms the monitor into a task, called a administrator [Gen81], and its thread is used to perform complex coordination operations between clients and workers. Notice, the administrator task still needs internal buffers to hold multiple arguments passed asynchronously by clients. Note, this approach now shares the buffer mutual-exclusion with the task; however, the administrator task can mitigate this issue by not making blocking calls and only performing simple administration work, so it is mostly ready to accept asynchronous calls from clients. In this case, the administrator may spend most of its time blocked waiting for client and/or worker calls, but this behaviour is often a reasonable tradeoff to allow centralizing of administrative duties when managing complex requests and interactions.

Figure 3.4 illustrates a server composed of a monitor buffer and worker task (middle structure in Figure 3.3). Both an ESM and ISM version of the server are presented, where the differences are storage management and cancellation of a future. Each server has server-specific data, ServerData, created in each future for use in cancellation. When a client cancels a future associated with this server, member ServerData::cancel is called, and both servers mark the position in the request queue to indicate that future is cancelled. The worker-task type, InputWorker, and an instance of it, is, are local to the server for abstraction and encapsulation reasons. InputWorker reads an $< integer, string >$ tuple and communicates the tuple to the server via a synchronous call to the private mutex-member input, which checks if a future exists with a matching integer key, and if so, places the string into that future as its result value. The ESM server conditionally inserts the string into the future by checking if the future at position value is NULL indicating it has been cancelled. The ISM server does not conditionally insert the string because an empty future is inserted at position value to hold the string if the original future is cancelled. Asynchronous calls from clients are made by calling mutex member request, specifying an integer key and a future to return the associated string read by the input worker. The ESM server resets the future passed to it as it reuses the future, and the ISM server creates a new future. If the new request is greater than the vector size, the vector size is increased. The future is then buffered in vector requests until the input worker subsequently fills in a value, and server-specific data is filled into the future in case the client cancels

ESM

```
_Monitor InputServer {
    struct ServerData {
        InputServer *server;
        int requested;

        bool cancel() {
            server->requests[requested] = NULL;
            return true;
        } // ServerData::cancel
        ServerData() {}
    }; // ServerData

    _Task InputWorker {
        InputServer &is;

        void main() {
            int id;
            string text;

            while ( cin >> id ) {
                getline( cin, text );
                is.input( id, text );
            }
        } // main
      public:
        InputWorker( InputServer &is ) : is( is ) {}
    }; // InputWorker
  public:
    typedef Future_ESM< string,
                        ServerData > FutureType;
  private:
    InputWorker iw;
    vector< FutureType * > requests;

    _Mutex void input( int value, string text ) {
        if ( requests.size() > value ) {
            if ( requests[value] != NULL ) {
                requests[value]->delivery( text );
            }
        }
    } // input
  public:
    InputServer() : iw( *this ) {}

    void request( FutureType *f, int requested ) {
        f->reset();
        if ( requests.size() <= requested ) {
            requests.resize( requested + 1 );
        }
        requests[requested] = f;
        f->serverData.server = this;
        f->serverData.requested = requested;
    } // request
}; // InputServer
```

ISM

```
_Monitor InputServer {
    struct ServerData : public Future_ISM< string >::ServerData {
        InputServer *server;
        int requested;

        bool cancel() {
            server->requests[requested] = Future_ISM< string >();
            return true;
        } // ServerData::cancel
        ServerData(InputServer *s, int r) : server(s), requested(r) {}
    }; // ServerData

    _Task InputWorker {
        InputServer &is;

        void main() {
            int id;
            string text;

            while ( cin >> id ) {
                getline( cin, text );
                is.input( id, text );
            } // while
        } // main
      public:
        InputWorker( InputServer &is ) : is( is ) {}
    }; // InputWorker
  public:
    typedef Future_ISM< string > FutureType;

  private:
    InputWorker iw;
    vector< FutureType > requests;

    _Mutex void input( int value, string text ) {
        if ( requests.size() > value ) {

            requests[value].delivery( text );

        }
    } // input
  public:
    InputServer() : iw( *this ) {}

    FutureType request( int requested ) {
        FutureType f( new ServerData( this, requested ) );
        if ( requests.size() <= requested ) {
            requests.resize( requested + 1 );
        }
        requests[requested] = f;

        return f;
    } // request
}; // InputServer
```

Figure 3.4: Example Server

the future.

3.3.1 Executors

An executor is a predefined, generic server with a fixed-size pool of worker threads performing submitted units of work, where work is formed by a routine or functor.

```
class uExecutor {
    enum { DefaultWorkers = 16, DefaultProcessors = 2 };
  public:
    enum Cluster { Same, Sep };             // use same or separate cluster
    uExecutor();
    uExecutor( Cluster clus );
    uExecutor( unsigned int nworkers, Cluster clus = Same );
    uExecutor( unsigned int nworkers, unsigned int nprocessors, Cluster clus = Same );
    template <typename Func> void send( Func action );
    template <typename Func> auto sendrecv( Func action ) -> Future_ISM<decltype(action())>;
};
```

The constructor routines have the following form:

uExecutor() – creates an executor containing a work queue, DefaultWorkers worker threads, running on DefaultProcessors processors, which are created on the current cluster with the cluster's default stack size.

uExecutor(Cluster clus) – creates an executor containing a work queue, DefaultWorkers worker threads, running on DefaultProcessors processors, which are created on the current or separate cluster with the cluster's default stack size.

uExecutor(**unsigned int** nworkers, Cluster clus = Same) – creates an executor containing a work queue, nworkers worker threads, running on DefaultProcessors processors, which are created on the current or separate cluster with the cluster's default stack size.

uExecutor(**unsigned int** nworkers, **unsigned int** nprocessors, Cluster clus = Same) – creates an executor containing a work queue, nworkers worker threads, running on nprocessors processors, which are created on the current or separate cluster with the cluster's default stack size.

The member routine send queues a unit of work action, with *no* return value, on a FIFO buffer in the executor to be eventually executed by one of the worker threads. The member routine sendrecv queues a unit of work action, with a return value, on a FIFO buffer in the executor to be eventually executed by one of the worker threads.

Figure 3.5 shows an example where work is submitted to an executor in the form of a routine, functor and lambda, and then different types of values (**int**, **double**, **char**) are returned.

```
#include <iostream>
#include <uFuture.h>
using namespace std;

int routine() {
    return 3;                                   // preform work and return result
}
struct Functor {                                // closure: allows arguments to work
    double val;
    double operator()() {                       // function-call operator
        return val;                             // preform work and return result
    }
    Functor( double val ) : val( val ) {}
} functor( 4.5 );

void routine2() {
    osacquire( cout ) << -1 << endl;            // preform work and no result
}
struct Functor2 {                               // closure: allows arguments to work
    double val;
    void operator()() {                         // function-call operator
        osacquire( cout ) << val << endl;       // preform work and no result
    }
    Functor2( double val ) : val( val ) {}
} functor2( 7.3 );

void uMain::main() {
    uExecutor executor;                         // work-pool of threads and processors
    Future_ISM<int> fi[10];
    Future_ISM<double> fd[10];
    Future_ISM<char> fc[10];

    for ( int i = 0; i < 10; i += 1 ) {         // send off work for executor
        fi[i] = executor.sendrecv( routine );   //   and get return future
        fd[i] = executor.sendrecv( functor );
        fc[i] = executor.sendrecv( []() { return 'a'; } );
    } // for
    executor.send( routine2 );                  // send off work but no return value
    executor.send( functor2 );
    executor.send( []() { osacquire( cout ) << 'd' << endl; } );
    for ( int i = 0; i < 10; i += 1 ) {         // wait for results
        osacquire( cout ) << fi[i]() << " " << fd[i]() << " " << fc[i]() << " ";
    } // for
    cout << endl;
}
```

Figure 3.5: Executor Example

Chapter 4

Input/Output

A major problem with concurrency and the file system is that, like the compiler, the file system is unaware if a program is concurrent (see Section 2.14, p. 33). To ensure multiple tasks are not performing I/O operations simultaneously on the same file descriptor, each µC++ file is implemented as a monitor that provides mutual exclusion on I/O operations. However, there are more complex issues relating to I/O operations in a concurrent system.

4.1 Nonblocking I/O

For a sequential program performing an I/O operation that cannot proceed immediately, the normal action for the file system is to block the program until the operation can continue. For example, when a program needs input from the keyboard, the file system blocks the program until data is entered. This action is correct for a sequential program because there is no other work for it to do until the new data is supplied by the user. However, this action may be incorrect for a concurrent program because there may be other work to do even without the user data. Therefore, the normal action by the file system, called heavy blocking (see Section 8.4.3, p. 120), is usually inappropriate for a concurrent program because it inhibits concurrency. Therefore, I/O operations must be transformed from heavy blocking to light blocking so that execution of other tasks can continue. This transformation is achieved by nonblocking I/O. To simplify the complexity of nonblocking I/O, µC++ supplies a nonblocking I/O library.

While I/O operations can be made nonblocking, this requires special action as the nonblocking I/O operations may not restart automatically when the operation completes. Instead, it may be necessary to poll for I/O completions, which is done through the select operation in UNIX, while other tasks execute. Only when all tasks on a cluster are directly or indirectly (light-) blocked, waiting for I/O operations to complete, can the virtual processor be heavy blocked.

This scenario is implemented automatically by µC++ choosing a task performing I/O to poll for completion of *any* I/O operation, called the **poller task**; all other tasks performing I/O are light blocked. When an I/O operation completes (e.g., a read or write), the task waiting for that operation is unblocked by the poller task. If the poller's I/O completes, it unblocks one of the I/O blocked tasks and that task becomes the I/O poller. Only when the poller detects that no I/O operations have completed and there are no tasks on the cluster to execute (i.e., the cluster's ready queue is empty) does the poller perform a heavy block. This scheme allows other tasks to progress with only a slight degradation in performance due to the polling task.

4.2 C++ Stream I/O

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:

task$_1$: cout << "abc " << "def " << endl;
task$_2$: cout << "uvw " << "xyz " << endl;

some of the different outputs that can appear are:

```
abc def | abc uvw xyz | uvw abc xyz def | abuvwc dexf | uvw abc def
uvw xyz | def         |                 | yz          | xyz
```

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 or within a monitor, 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.

*µ*C++ provides a fine-grained solution where an owner lock is acquired and released indirectly by instantiating a type that is specific to the kind of stream: type isacquire for input streams and type osacquire for output streams *located in namespace* std. 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
```

For an fstream, which can perform both input and output, both isacquire and osacquire can be used. The only restriction is that the kind of stream locker has to match with kind of I/O operation, e.g.:

```
fstream file( "abc" );
osacquire( file ) << ... // output operations
...
isacquire( file ) >> ... // input operations
```

For protecting multiple I/O statements on an fstream, either isacquire or osacquire can be used to acquire the stream lock, e.g.:

```
fstream file( "abc" );
{   // acquire the lock for stream file for block duration
    osacquire acq( file ); // or isacquire acq( file )
    file >> ... // input operations
    ...
    file << ... // output operations
}   // implicitly release the lock when "acq" is deallocated
```

WARNING: Deadlock can occur if routines are called in an I/O sequence that might block, e.g.:

```
osacquire( cout ) << "data:" << Monitor.rtn(...) << endl;
```

The problem occurs if the task executing the I/O sequence blocks in the monitor when it is holding the I/O lock for stream cout. Any other task that attempts to write on cout blocks until the task holding the lock is unblocked and releases it. This scenario can lead to deadlock if the task that is going to unblock the task waiting in the monitor first writes to cout. One simple precaution is to factor the call to the monitor routine out of the I/O sequence, e.g.:

```
int data = Monitor.rtn(...);
osacquire( cout ) << "data:" << data << endl;
```

WARNING: A stream may be *tied* to another output stream, so when the first stream performs any I/O, the second stream is implicitly flushed first. For example, cin is tied to cout, so in:

```
cout << "Enter number:";      // prompt for data
cin >> number;                // read data
```

the prompt is guaranteed to be flushed before the data is read. If cin and cout were not tied, the read may occur without the prompt message because the message is in cout's output buffer. While tieing streams is an important capability, it can cause a race condition between tasks using the tied streams. For example, if one task is reading from cin and another is writing to cout, an implicit flush can occur to cout from the first task at the same time as an explicit write from the second task. Having to lock in this situation is non-intuitive because the stream operations seem disjoint, e.g.:

```
             Task1                                    Task2
                                                {
osacquire( cout ) << . . . // acquire cout          osacquire lock( cout );  // acquire cout due to tie
                                                    cin >> number;
                                                }
```

Task$_1$ must lock cout even though it may be the only task writing to it; Task$_2$ must also lock cout even though it is reading from cin. In general, it is rare for separate tasks to be prompting and reading in this manner; normally, these steps are performed by a single task. However, it is reasonable for different tasks to be using cin and cout but not need implicit flushing. In this case, the best solution is to remove the tie between cin and cout, e.g.:

```
cin.tie( NULL );        // set tie partner to NULL
```

eliminating the implicit flushing and the race condition (see examples in Sections D.5.1, p. 167 and D.5.3, p. 170).

WARNING: printf, scanf and their derivatives are unsafe when used with user-level threading, as in μC++. These stream routines use kernel-thread locking (futex), which block kernel threads, to prevent interleaving of I/O. However, the following simple example illustrates how a deadlock can occur (other complex scenarios are possible). Assume a single kernel thread and two user-level threads calling printf. One user-level thread acquires the I/O lock and is time-sliced while performing printf. The other user-level thread then starts execution, calls printf, and blocks the only kernel thread because it cannot acquire the I/O lock. It does not help if the kernel lock is multiple acquisition, i.e., the lock owner can acquire it multiple times, because it then results in two user threads in the printf critical section, corrupting the stream.

4.3 UNIX File I/O

A UNIX file is a passive object that has information written into and read from it by tasks; therefore, a file is like a monitor, which provides indirect communication among tasks. The difference between a file and a monitor is that the file is on secondary storage, and hence, is not directly accessible by the computer's processors; a file must be made explicitly accessible before it can be used in a program. Furthermore, a file may have multiple accessors—although it is up to UNIX to interpret the meaning of these potentially concurrent accessors—so there is a many-to-one relationship between a file and its accessors. This relationship is represented in a μC++ program by a declaration for a file and subsequent declarations for each accessor.

Traditionally, access to a file is explicit and is achieved procedurally by a call to "open" and a subsequent call to "close" to terminate the access. In μC++, the declaration of a special **access object** performs the equivalent of the traditional open and its deallocation performs the equivalent of the traditional close. In many cases, the access object is a local variable so that the duration of access is tied to the duration of its containing block. However, by dynamically allocating an access object and passing its pointer to other blocks, the equivalent access duration provided by traditional "open" and "close" can be achieved.

In μC++, a connection to a UNIX file is made by declaration of a uFile object, e.g.:

```
uFile infile( "abc" ), outfile( "xyz" );
```

which creates two connection variables, infile and outfile, connected to UNIX files abc and xyz, respectively. The operations available on a file object are listed in Figure 4.1:

There are no parameters for the first constructor of uFile and it is used with setName. The parameter for the second constructors of uFile are as follows. The name parameter is the UNIX name of the file, which is connected to the program. The destructor of uFile checks if there are any registered accessors using the file, and raises the exception TerminateFailure if there are.

It is *not* meaningful to read or to assign to a uFile object, or copy a uFile object (e.g., pass it as a value parameter).

The member routine setName associates a string name with a file (usually for subsequent open). The member routine getName returns the string name associated with a file.

The parameter for member routine status is explained in the UNIX manual entry for stat. (The first parameter to the UNIX stat routine is unnecessary, as it is provided implicitly by the uFile object.) Because a file object is still inaccessible after a connection is made, there are no member routines to access its contents.

```
class uFile {
  public:
    uFile();
    uFile( const char *name );
    ~uFile();

    class FileAccess {
      public:
        FileAccess();
        FileAccess( uFile &f, int flags, int mode = 0644 );
        FileAccess( const char *name, int flags, int mode = 0644 );
        ~FileAccess();

        int read( char *buf, int len, uDuration *timeout = NULL );
        int readv( const struct iovec *iov, int iovcnt, uDuration *timeout = NULL );
        _Mutex int write( const char *buf, int len, uDuration *timeout = NULL );
        int writev( const struct iovec *iov, int iovcnt, uDuration *timeout = NULL );
        void open( uFile &f, int flags, int mode = 0644 );
        off_t lseek( off_t offset, int whence );
        int fsync();
        void status( struct stat &buf );
        int fd();

        _Event Failure;
        _Event OpenFailure;
        _Event CloseFailure;
        _Event SeekFailure;
        _Event SyncFailure;
        _Event ReadFailure;
        _Event ReadTimeout;
        _Event WriteFailure;
        _Event WriteTimeout;
    }; // FileAccess

    const char *setName( char *name );
    const char *getName() const;
    void status( struct stat &buf );

    _Event Failure;
    _Event TerminateFailure;
    _Event StatusFailure;
}; // uFile
```

Figure 4.1: uFile Interface

To use the interface, include the file:

```
#include <uFile.h>
```

at the beginning of each source file. uFile.h also includes the following UNIX system file: <fcntl.h>

4.3.1 File Access

Once a connection is made to a UNIX file, its contents can be accessed by declaration of a FileAccess object, e.g.:

```
uFile::FileAccess input( infile, O_RDONLY ), output( outfile, O_CREAT | O_WRONLY );
```

which creates one access object to read from the connection to file abc and one object to write to the connection made to file xyz.

There are no parameters for the first constructor of uFileAccess and it is used with open. The parameters for the second constructor FileAccess are as follows. The f parameter is a uFile object to be opened for access. The flags and mode parameters are explained in the UNIX manual entry for open. The destructor of FileAccess terminates access to the file and deregisters with the associated uFile object.

The parameters for the third constructor FileAccess are as follows. The name parameter is the UNIX name of the file, which is connected to the program. In this case, an implicit uFile object is created to represent the file; hence, there is a one-to-one relationship for this access with the specified file. The flags and mode parameters are explained in the UNIX manual entry for open.

It is *not* meaningful to read or to assign to a FileAccess object, or copy a FileAccess object (e.g., pass it as a value parameter).

The parameters and return value for member routines read, readv, write, writev, open, lseek and fsync are explained in their corresponding UNIX manual entries. The first parameter to these UNIX routines is unnecessary (except open), as it is provided implicitly by the FileAccess object. The only exception is the optional parameter timeout, which points to a maximum waiting time for completion of the I/O operation before aborting the operation by raising an exception (see Section 10.2.4, p. 132). (The type uDuration is defined in Section 10.1, p. 127.) Appendix D.4, p. 165 shows reading and writing to UNIX files.

The member routine fd returns the file descriptor for the open UNIX file.

4.4 BSD Sockets

A socket is an end point for communicating among tasks in different processes, possibly on different computers. A socket endpoint is accessed in one of two ways:

1. as a **client**, which is one-to-many for connectionless communication with multiple server socket-endpoints, or one to one for peer-connection communication with a server's acceptor socket-endpoint.

2. as a **server**, which is one-to-many for connectionless communication with multiple client socket-endpoints, or one to one for peer-connection communication with a server's *acceptor* socket-endpoint.

The relationship between connectionless and peer-connection communication is shown in Figures 4.2 and 4.3. For connectionless communication (see Figure 4.2), any of the client socket-endpoints can communicate with any of the server socket-endpoints, and vice versa, as long as the other's address is known. This flexibility is possible because each communicated message contains the address of the sender or receiver; the network then routes the message to this address. For convenience, when a message arrives at a receiver, the sender's address replaces the receiver's address, so the receiver can reply back. For peer-connection communication (see Figure 4.3), a client socket-endpoint can *only* communicate with the server socket-endpoint it has connected to, and vice versa. The dashed lines show the connection of the client and server. The dotted lines show the creation of an acceptor to service the connection for peer communication. The solid lines show the bidirectional communication among the client and server's acceptor. Since a specific connection is established between a client and server socket-endpoints, messages do not contain sender and receive addresses, as these addresses are implicitly known through the connection. Notice there are fewer socket endpoints in the peer-connection communication versus the connectionless communication, but more acceptors. For connectionless communication, a single socket-endpoint sequentially handles both the connection and the transfer of data for each message. For peer-connection communication, a single socket-endpoint handles connections and an acceptor transfers data in parallel. In general, peer-connection communication is more expensive (unless large amounts of data are transferred) but more reliable than connectionless communication.

A server socket has a name, either a character string for UNIX pipes or port-number/machine-address for an INET address, that clients must know to communicate. For connectionless communication, the server usually has a reader task that receives messages containing the client's address. The message can be processed by the reader task or given to a worker task to process, which subsequently returns a reply using the client's address present in the received message. For peer-connection communication, the server usually has one task in a loop accepting connections from clients, and each acceptance creates an acceptor task. The acceptor task receives messages from only one client socket-endpoint, processes the message and subsequently returns a reply, in parallel with accepting clients. Since the acceptor and client are connected, communicated messages do not contain client addresses. These relationships are represented in a *μ*C++ program by declarations of client, server and acceptor objects, respectively.

The *μ*C++ socket interface provides a convenience feature for connectionless communication to help manage the addresses where messages are sent. It is often the case that a client only sends messages from its client socket-endpoint to a single server socket-endpoint or sends a large number of messages to a particular server socket-endpoint. In these cases, the address of the server remains constant for a long period of time. To mitigate having to specify the server address on each call for a message send, the client socket-endpoint *remembers* the last server address it receives a message from, and there is a short form of send that uses this remembered address. The initial remembered (default) address can be set when the client socket-endpoint is created or set/reset at any time during its life-time. A similar

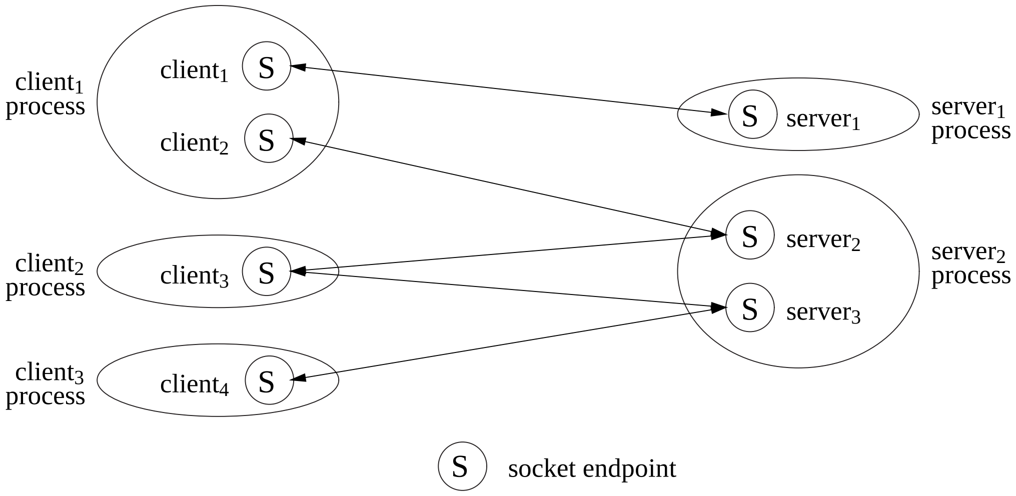


Figure 4.2: Client/Server Connectionless

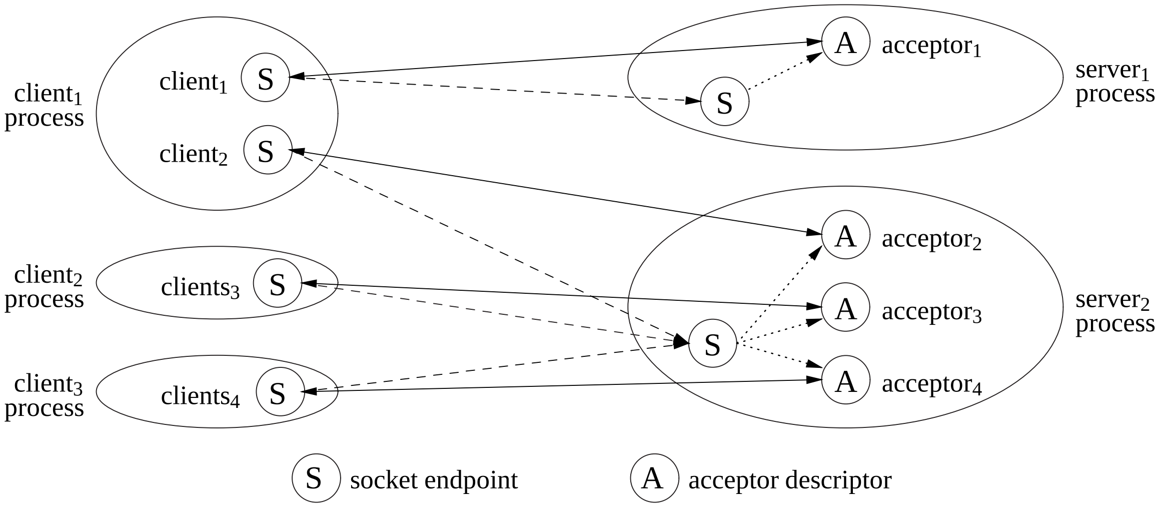


Figure 4.3: Client/Server Peer Connected

convenience feature exists for the server socket-endpoint, where the last client address it receives a message from is remembered and can be implicitly used to send a message directly back to that client.

To use the interface in a µC++ program, include the file:

```
#include <uSocket.h>
```

at the beginning of each source file. uSocket.h also includes the following UNIX system files: <sys/fcntl.h>, <sys/types.h>, <sys/socket.h>,<sys/un.h>, <netdb.h>.

The following helper routines are available for converting different forms of Internet address to an ip address used by a client or a server. These routines are static within the type uSocket:

```
in_addr uSocket::gethostbyname( const char *name );
in_addr uSocket::gethostbyip( const char *ip );
in_addr uSocket::itoip( in_addr_t ip );
```

gethostbyname takes a machine name, e.g., plg.uwaterloo.ca, and returns the 4-byte ip address for this machine. gethostbyip takes a character string ip address, e.g., "123.123.123.123", and returns the 4-byte ip address equivalent. itoip takes an integer containing an ip address and returns an in_addr object containing this integer ip value. If routines gethostbyname or gethostbyip fail to lookup a machine name or convert the string ip address, the exception uSocket::IPConvertFailure is raised.

```
_Monitor uSocketClient {
  public:
    // AF_UNIX
    uSocketClient( const char *name, int type = SOCK_STREAM, int protocol = 0 );
    uSocketClient( const char *name, uDuration *timeout, int type = SOCK_STREAM, int protocol = 0 );
    // AF_INET, local host
    uSocketClient( unsigned short port, int type = SOCK_STREAM, int protocol = 0 );
    uSocketClient( unsigned short port, uDuration *timeout, int type = SOCK_STREAM, int protocol = 0 );
    // AF_INET, other host
    uSocketClient( unsigned short port, in_addr ip, int type = SOCK_STREAM, int protocol = 0 );
    uSocketClient( unsigned short port, in_addr ip, uDuration *timeout, int type = SOCK_STREAM,
                   int protocol = 0 );
    ~uSocketClient();

    void setServer( struct sockaddr *addr, socklen_t len );
    void getServer( struct sockaddr *addr, socklen_t *len );

    const struct sockaddr *getsockaddr();    // must cast result to sockaddr_in or sockaddr_un
    int getsockname( struct sockaddr *name, socklen_t *len );
    int getpeername( struct sockaddr *name, socklen_t *len );

    int read( char *buf, int len, uDuration *timeout = NULL );
    int readv( const struct iovec *iov, int iovcnt, uDuration *timeout = NULL );
    _Mutex int write( const char *buf, int len, uDuration *timeout = NULL );
    int writev( const struct iovec *iov, int iovcnt, uDuration *timeout = NULL );
    int send( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int sendto( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int sendto( char *buf, int len, struct sockaddr *to, socklen_t tolen, int flags = 0, uDuration *timeout = NULL );
    int sendmsg( const struct msghdr *msg, int flags = 0, uDuration *timeout = NULL );
    int recv( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int recvfrom( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int recvfrom( char *buf, int len, struct sockaddr *from, socklen_t *fromlen, int flags = 0,
                   uDuration *timeout = NULL );
    int recvmsg( struct msghdr *msg, int flags = 0, uDuration *timeout = NULL );
    ssize_t sendfile( FileAccess &file, off_t *off, size_t len, uDuration *timeout = NULL );
    int fd();

    _Event Failure;
    _Event OpenFailure;
    _Event OpenTimeout;
    _Event CloseFailure;
    _Event ReadFailure;
    _Event ReadTimeout;
    _Event WriteFailure;
    _Event WriteTimeout;
    _Event SendfileFailure;
    _Event SendfileTimeout;
};
```

Figure 4.4: uSocketClient Interface

4.4.1 Client

In μC++, a client, its socket endpoint, and possibly a connection to a server are created by declaration of a uSocketClient object, e.g.:

```
uSocketClient client( "abc" );
```

which creates a client variable, client, connected to the UNIX server socket, abc. The operations provided by uSocketClient are listed in Figure 4.4:

The first two constructors of uSocketClient are for use with the UNIX address family. The parameters for the constructors are as follows. The name parameter is the name of an existing UNIX stream that the client is connecting to. The name parameter can be NULL for type SOCK_DGRAM, if there is no initial server address. The optional default type and protocol parameters are explained in the UNIX manual entry for socket. Only types SOCK_STREAM and SOCK_DGRAM communication can be specified, and any protocol appropriate for the specified communication type (usually 0). The optional timeout parameter is a pointer to a maximum waiting time for completion of a connection for type SOCK_STREAM before aborting the operation by raising an exception (see Section 10.2.4, p. 132); this parameter is only applicable for peer-connection, SOCK_STREAM, communication.

The next two constructors of uSocketClient are for use with the INET address family on a local host. The parameters for the constructors are as follows. The port parameter is the port number of an INET port on the local host machine. The optional default type and protocol parameters are explained in the UNIX manual entry for socket. Only types SOCK_STREAM and SOCK_DGRAM communication can be specified, and any protocol appropriate for the specified communication type (usually 0). The optional parameter timeout is a pointer to a maximum waiting time for completion of a connection for type SOCK_STREAM before aborting the operation by raising an exception; this parameter is only applicable for peer-connection, SOCK_STREAM, communication.

The last two constructors of uSocketClient are for specifying a specific ip address with the INET address family on a nonlocal host. All parameters are the same as for the local host case, except the specific nonlocal address is specified by the ip parameter.

The destructor of uSocketClient terminates the socket (close) and removes any temporary files created implicitly for SOCK_STREAM and SOCK_DGRAM communication.

It is *not* meaningful to read or to assign to a uSocketClient object, or copy a uSocketClient object (e.g., pass it as a value parameter).

The member routine setServer changes the address of the default server for the short forms of sendto and recvfrom. The member routine getServer returns the address of the default server.

The parameters and return value for the I/O members are explained in their corresponding UNIX manual entries, with the following exceptions:

- getpeername is only applicable for connected sockets.
- The first parameter to these UNIX routines is unnecessary, as it is provided implicitly by the uSocketClient object.
- The lack of address for the overloaded member routines sendto and recvfrom.

  The client implicitly remembers the address of the initial connection and each recvfrom call. Therefore, no address needs to be specified in the sendto, as the data is sent directly back to the last address received. If a client needs to communicate with multiple servers, explicit addresses can be specified in both sendto and recvfrom.

  This capability eliminates the need to connect datagram sockets to use the short communication forms send and recv, using the connected address. In general, connected datagram sockets have the same efficiency as unconnected ones, but preclude specific addressing via sendto and recvfrom. The above scheme provides the effect of a connected socket while still allowing specific addressing if required.
- The optional parameter timeout, which points to a maximum waiting time for completion of the I/O operation before aborting the operation by raising an exception (see Section 10.2.4, p. 132).

The member routine fd returns the file descriptor for the client socket.

Appendix D.5.1, p. 167 shows a client communicating with a server using a UNIX socket and datagram messages. Appendix D.5.3, p. 170 shows a client connecting to a server using an INET socket and stream communication with an acceptor.

4.4.2 Server

In μC++, a server, its socket endpoint, and possibly a connection to a client are created by declaration of a uSocketServer object, e.g.:

```
uSocketServer server( "abc" );
```

which creates a server variable, server, and a UNIX server socket endpoint, abc. The operations provided by uSocketServer are listed in Figure 4.5:

The first constructor of uSocketServer is for use with the UNIX address family. The parameters for the constructors are as follows. The name parameter is the name of a new UNIX server socket that the server is creating. The optional

```
_Monitor uSocketServer {
  public:
    // AF_UNIX
    uSocketServer( const char *name, int type = SOCK_STREAM, int protocol = 0, int backlog = 10 );
    // AF_INET, local host
    uSocketServer( unsigned short port, int type = SOCK_STREAM, int protocol = 0, int backlog = 10 );
    uSocketServer( unsigned short *port, int type = SOCK_STREAM, int protocol = 0, int backlog = 10 );
    uSocketServer( unsigned short port, in_addr ip, int type = SOCK_STREAM, int protocol = 0, int backlog = 10 );
    uSocketServer( unsigned short *port, in_addr ip, int type = SOCK_STREAM, int protocol = 0, int backlog = 10 );
    ~uSocketServer();

    void setClient( struct sockaddr *addr, socklen_t len );
    void getClient( struct sockaddr *addr, socklen_t *len );

    const struct sockaddr *getsockaddr();     // must cast result to sockaddr_in or sockaddr_un
    int getsockname( struct sockaddr *name, socklen_t *len );
    int getpeername( struct sockaddr *name, socklen_t *len );

    int read( char *buf, int len, uDuration *timeout = NULL );
    int readv( const struct iovec *iov, int iovcnt, uDuration *timeout = NULL );
    _Mutex int write( const char *buf, int len, uDuration *timeout = NULL );
    int writev( const struct iovec *iov, int iovcnt, uDuration *timeout = NULL );
    int send( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int sendto( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int sendto( char *buf, int len, struct sockaddr *to, socklen_t tolen, int flags = 0, uDuration *timeout = NULL );
    int sendmsg( const struct msghdr *msg, int flags = 0, uDuration *timeout = NULL );
    int recv( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int recvfrom( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int recvfrom( char *buf, int len, struct sockaddr *from, socklen_t *fromlen, int flags = 0,
                  uDuration *timeout = NULL );
    int recvmsg( struct msghdr *msg, int flags = 0, uDuration *timeout = NULL );
    ssize_t sendfile( FileAccess &file, off_t *off, size_t len, uDuration *timeout = NULL );
    int fd();

    _Event Failure;
    _Event OpenFailure;
    _Event CloseFailure;
    _Event ReadFailure;
    _Event ReadTimeout;
    _Event WriteFailure;
    _Event WriteTimeout;
    _Event SendfileFailure;
    _Event SendfileTimeout
};
```

Figure 4.5: uSocketServer Interface

default type and protocol parameters are explained in the UNIX manual entry for socket. Only types SOCK_STREAM and SOCK_DGRAM communication can be specified, and any protocol appropriate for the specified communication type (usually 0). The optional default backlog parameters is explained in the UNIX manual entry for listen; it specifies a limit on the number of incoming connections from clients and is only applicable for peer-connection, SOCK_STREAM, communication.

The next two constructors of uSocketServer are for use with the INET address family on a local host. The parameters for the constructors are as follows. The port parameter is the port number of an INET port on the local host machine, or a pointer to a location where a free port number, selected by the UNIX system, is placed. The optional default type and protocol parameters are explained in the UNIX manual entry for socket. Only types SOCK_STREAM and SOCK_DGRAM communication can be specified, and any protocol appropriate for the specified communication type (usually 0). The optional default backlog parameters is explained in the UNIX manual entry for listen; it specifies a limit on the number of incoming connections from clients and is only applicable for peer-connection, SOCK_STREAM, communication.

The last two constructors of uSocketServer are for specifying a specific ip address with the INET address family on the local host. All parameters are the same as for the local host case, except the specific local address is specified by the ip parameter.

The destructor of uSocketServer terminates the socket (close) and checks if there are any registered accessors using the server, and raises the exception CloseFailure if there are.

It is *not* meaningful to read or to assign to a uSocketServer object, or copy a uSocketServer object (e.g., pass it as a value parameter).

The member routine setClient changes the address of the default client for the short forms of sendto and recvfrom. The member routine getClient returns the address of the default client.

The parameters and return value for the I/O members are explained in their corresponding UNIX manual entries, with the following exceptions:

- getpeername is only applicable for connected sockets.
- The first parameter to these UNIX routines is unnecessary, as it is provided implicitly by the uSocketServer object.
- The lack of address for the overloaded member routines sendto and recvfrom.

  The server implicitly remembers the address of the initial connection and each recvfrom call. Therefore, no address needs to be specified in the sendto, as the data is sent directly back to the last address received. If a server needs to communicate with multiple clients without responding back immediately to each request, explicit addresses can be specified in both sendto and recvfrom.

  This capability eliminates the need to connect datagram sockets to use the short communication forms send and recv, using the connected address. In general, connected datagram sockets have the same efficiency as unconnected ones, but preclude specific addressing via sendto and recvfrom. The above scheme provides the effect of a connected socket while still allowing specific addressing if required.
- The optional parameter timeout, which points to a maximum waiting time for completion of the I/O operation before aborting the operation by raising an exception (see Section 10.2.4, p. 132).

The member routine fd returns the file descriptor for the server socket.

Appendix D.5.2, p. 169 shows a server communicating with multiple clients using a UNIX socket and datagram messages. Appendix D.5.4, p. 172 shows a server communicating with multiple clients using an INET socket and stream communication with an acceptor.

4.4.3 Server Acceptor

After a server socket is created for peer-connection communication, it is possible to accept connections from clients by declaration of a uSocketAccept object, e.g.:

```
uSocketAccept acceptor( server );
```

which creates an acceptor object, acceptor, that blocks until a client connects to the UNIX socket, abc, represented by server object server. The operations provided by uSocketAccept are listed in Figure 4.6:

The parameters for the constructors of uSocketAccept are as follows. The s parameter is a uSocketServer object through which a connection to a client is made. The optional default adr and len parameters, are explained in the UNIX manual entry for accept, and are used to determine information about the client the acceptor is connected to. The optional timeout parameter is a pointer to a maximum waiting time for completion of the connection before aborting the operation by raising an exception (see Section 10.2.4, p. 132). The optional doAccept parameter is a boolean where true means do an initial accept during initialization of the acceptor and false means do not do an initial accept. If the doAccept parameter is not specified, its value is true.

The destructor of uSocketAccept terminates access to the socket (close) and deregisters with the associated uSocketServer object.

It is *not* meaningful to read or to assign to a uSocketAccept object, or copy a uSocketAccept object (e.g., pass it as a value parameter).

The member routine accept closes any existing connection to a client, and accepts a new connection with a client. This routine uses the default values adr, len and timeout as specified to the uSocketAccept constructor for the new connection, unless the optional timeout parameter is specified, which is used for the current accept and replaces the default timeout for subsequent accepts. The member routine close closes any existing connection to a client.

```
_Monitor uSocketAccept {
  public:
    uSocketAccept( uSocketServer &s, struct sockaddr *adr = NULL, socklen_t *len = NULL );
    uSocketAccept( uSocketServer &s, uDuration *timeout, struct sockaddr *adr = NULL, socklen_t *len = NULL );
    uSocketAccept( uSocketServer &s, bool doAccept, struct sockaddr *adr = NULL, socklen_t *len = NULL );
    uSocketAccept( uSocketServer &s, uDuration *timeout, bool doAccept, struct sockaddr *adr = NULL,
                    socklen_t *len = NULL );
    ~uSocketAccept();

    void accept();
    void accept( uDuration *timeout );
    void close();

    _Mutex const struct sockaddr *getsockaddr();    // must cast result to sockaddr_in or sockaddr_un
    _Mutex int getsockname( struct sockaddr *name, socklen_t *len );
    _Mutex int getpeername( struct sockaddr *name, socklen_t *len );

    int read( char *buf, int len, uDuration *timeout = NULL );
    int readv( const struct iovec *iov, int iovcnt, uDuration *timeout = NULL );
    _Mutex int write( const char *buf, int len, uDuration *timeout = NULL );
    int writev( const struct iovec *iov, int iovcnt, uDuration *timeout = NULL );
    int send( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int sendto( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int sendto( char *buf, int len, struct sockaddr *to, socklen_t tolen, int flags = 0, uDuration *timeout = NULL );
    int sendmsg( const struct msghdr *msg, int flags = 0, uDuration *timeout = NULL );
    int recv( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int recvfrom( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int recvfrom( char *buf, int len, struct sockaddr *from, socklen_t *fromlen, int flags = 0,
                    uDuration *timeout = NULL );
    int recvmsg( struct msghdr *msg, int flags = 0, uDuration *timeout = NULL );
    ssize_t sendfile( FileAccess &file, off_t *off, size_t len, uDuration *timeout = NULL );
    int fd();

    _Event Failure;
    _Event OpenFailure;
    _Event OpenTimeout;
    _Event CloseFailure;
    _Event ReadFailure;
    _Event ReadTimeout;
    _Event WriteFailure;
    _Event WriteTimeout;
    _Event SendfileFailure;
    _Event SendfileTimeout;
};
```

Figure 4.6: uSocketAccept Interface

The parameters and return value for the I/O members are explained in their corresponding UNIX manual entries, with the following exceptions:

- The first parameter to these UNIX routines is unnecessary, as it is provided implicitly by the uSocketAccept object.
- The optional parameter timeout, which points to a maximum waiting time for completion of the I/O operation before aborting the operation by raising an exception (see Section 10.2.4, p. 132).

The member routine fd returns the file descriptor for the accepted socket.

□ μC++ does *not* support out-of-band data on sockets. Out-of-band data requires the ability to install a signal handler (see Section 4.1, p. 59). Currently, there is no facility to do this. □

Chapter 5

Exceptions

C++ has an exception handling mechanism (EHM) based on throwing and catching in sequential programs; however, this mechanism does not extend to a complex execution-environment. The reason is that the C++ EHM only deals with a single raise-mechanism and a simple execution-environment, i.e., throwing and one stack. The *μ*C++ execution environment is more complex, and hence, it provides additional raising-mechanisms and handles multiple execution-states (multiple stacks). These enhancements require additional language semantics and constructs; therefore, the EHM in *μ*C++ is a superset of that in C++, providing more advanced exception semantics. As well, with hindsight, some of the poorer features of C++'s EHM are replaced by better mechanisms.

5.1 EHM

An **exceptional event** is an event that is (usually) known to exist but which is *ancillary* to an algorithm, i.e., an exceptional event usually occurs with low frequency. Some examples of exceptional events are division by zero, I/O failure, end of file, pop from an empty stack, inverse of a singular matrix. Often an exceptional event occurs when an operation cannot perform its desired computation (Eiffel's notion of contract failure [Mey92, p. 395]). While errors occur infrequently, and hence, are often considered an exceptional event, it is incorrect to associate exceptions solely with errors; exceptions can be a standard part of a regular algorithm.

An exceptional event is often represented in a programming language by a type name, called an **exception type**. An **exception** is an instance of an exception type, which is used in a special operation, called **raising**, indicating an ancillary (exceptional) situation. Raising results in an *exceptional* change of control flow in the normal computation of an operation, i.e., control propagates immediately to a dynamically specified **handler**. To be useful, the handler location must be dynamically determined, as opposed to statically determined; otherwise, the same action and context for that action is executed for every exceptional change.

Two actions can sensibly be taken for an exceptional event:

1. The operation can fail requiring **termination** of the expression, statement or block from which the operation is invoked. In this case, if the handler completes, control flow continues *after* the handler, and the handler acts as an alternative computation for the incomplete operation.

2. The operation can fail requiring a corrective action before **resumption** of the expression, statement or block from which the operation is invoked. In this case, if the handler completes, control flow *returns* to the operation, and the handler acts as a corrective computation for the incomplete operation.

Both kinds of actions are supported in *μ*C++. Thus, there are two possible outcomes of an operation: normal completion possibly with a correction action, or failure with change in control flow and alternate computation.

> □ Even with the availability of modern EHMs, the common programming techniques often used to handle exceptional events are return codes and status flags (although this is slowly changing). The **return code** technique requires each routine to return a correctness value on completion, where different values indicate a normal or exceptional result during a routine's execution. Alternatively, or in conjunction with return codes, is the **status flag** technique requiring each routine to set a shared variable on completion, where different values indicate a normal or exceptional result during a routine's execution, e.g., `errno` in UNIX systems. The status value remains as long as it is not overwritten by another routine. □

5.2 *μ*C++ EHM

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

- *μ*C++ exceptions are generated from a specific kind of type, which can be thrown and/or resumed. All exception types are also grouped into a hierarchy by publicly inheriting among the exception types. *μ*C++ extends the C++ set of predefined exception-types[1] covering *μ*C++ exceptional runtime and I/O events.
- *μ*C++ restricts raising of exceptions to the specific exception-types; C++ allows any instantiable type to be raised.
- *μ*C++ supports two forms of raising, throwing (terminating) and resuming; C++ only supports throwing. All *μ*C++ exception-types can be either thrown or resumed. *μ*C++ adopts a marking scheme to eliminate recursive resuming (see Section 5.5.3.1, p. 78). Essentially, *μ*C++ follows a common rule for throwing and resuming: between a raise and its handler, each handler is eligible only once.
- *μ*C++ supports two kinds of handlers, termination and resumption, which match with the kind of raise; C++ only supports termination handlers.
- *μ*C++ supports raising of nonlocal and concurrent exceptions so that exceptions can be used to affect control flow *among* coroutines and tasks. A **nonlocal exception** occurs when the raising and handling execution-states are different, and control flow is sequential, i.e., the thread raising the exception is also the thread handling the exception. A **concurrent exception** also has different raising and handling execution-states (hence, concurrent exceptions are also nonlocal), but control flow is concurrent, i.e., the thread raising the exception is different from the thread handling the exception. The *μ*C++ kernel implicitly polls for both kinds of exceptions at the soonest possible opportunity. It is also possible to (hierarchically) block these kinds of exceptions when delivery would be inappropriate or erroneous.

5.3 Exception Type

While C++ allows any type to be used as an exception type, *μ*C++ restricts exception types to those defined by **_Event**, which has all the properties of a **class**:

```
_Event exception-type name {
    . . .
};
```

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

□ 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 builtin 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 exception-types. This practice is codified in *μ*C++. □

5.3.1 Creation and Destruction

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

```
_Event E { . . . };
E e;                    // local exception
_Resume e;
E *ep = new E;          // dynamic exception
_Resume *ep;
delete ep;
_Throw E();             // temporary local exception
```

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

5.3.2 Inherited Members

Each exception type, if not derived from another exception type, is implicitly derived from the event type uBaseEvent, e.g.:

```
_Event exception-type name : public uBaseEvent . . .
```

where the interface for the base-class uBaseEvent is:

```
class uBaseEvent {
  protected:
    uBaseEvent( const char *const msg = "" );
    void setMsg( const char *const msg );
  public:
    enum RaiseKind { ThrowRaise, ResumeRaise };

    const char *const message() const;
    const uBaseCoroutine &source() const;
    const char *const sourceName() const;
    RaiseKind getRaiseKind();
    void reraise() const;
    virtual uBaseEvent *duplicate() const;
    virtual void defaultTerminate();
    virtual void defaultResume();
};
```

The constructor routine uBaseEvent has the following form:

uBaseEvent(**const char** ***const** msg = "") – creates an exception with specified message, which is printed in an error message if the exception is not handled. The message is copied when an exception is created so it is safe to use within an exception even if the context of the raise is deleted.

The member routine setMsg is an alternate way to associate a message with an 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; if the exception has been raised locally, the value NULL is returned. 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; if the exception has been raised locally, the value "*unknown*" is returned. 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 getRaiseKind returns whether the exception is thrown (ThrowRaise) or resumed (ResumeRaise) at the raise. The member routine reraise either rethrows or reresumes the exception depending on how the exception was originally raised. The member routine duplicate returns a copy of the raised exception, which can be used to raise the same exception in a different context after it has been caught; the copy is allocated on the heap, so it is the responsibility of the caller to delete the exception.

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. 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. In both cases, a user-defined default action may be implemented by overriding the appropriate virtual member.

5.4 Raising

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

```
_Throw [ exception-type ] ;
_Resume [ exception-type ] [ _At uBaseCoroutine-id ] ;
```

If **_Throw** has no *exception-type*, it is a **rethrow**, meaning the currently thrown exception continues propagation. If there is no current thrown exception but there is a currently resumed exception, that exception is thrown. Otherwise, the rethrow results in a runtime error. If **_Resume** has no *exception-type*, it is a **reresume**, meaning the currently resumed exception continues propagation. If there is no current resumed exception but there is a currently thrown exception, that exception is resumed. Otherwise, the reresume results in a runtime error. The optional **_At** clause allows the specified exception or the currently propagating exception (reresume) to be raised at another coroutine or task. Nonlocal and concurrent raise is restricted to resumption because the raising execution-state is often unaware of the current state for the handling execution-state. Resumption allows the handler the greatest flexibility in this situation because it can process the exception as a resumption or rethrow the exception for termination (which is the default behaviour, see Section 5.3.2).

Exceptions in *µ*C++ are propagated differently from C++. In C++, the **throw** statement creates a copy of the static type of the exception, and propagates the copy. In *µ*C++, the **_Throw** and **_Resume** statements create a copy of dynamic type of the exception, and propagates the copy. For example:

C++	*µ*C++

class B {};
class D : **public** B {};
void f(B &t) {
    throw t;
}
D m;
f(m);
	_Event B {};
	_Event D : **public** B {};
	void f(B &t) {
	    _Throw t;
	}
	D m;
	f(m);

in the C++ program, routine f is passed an object of derived type D but throws a copy of an object of base type B, because the static type of the operand for throw, t, is of type B. However, in the *µ*C++ program, routine f is passed an object of derived type D and throws a copy of the original object of type D. 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 its dynamic type to the static type specified at the handler, and cannot be down-cast to the dynamic type. Notice, catching truncation is different from raising truncation, which does not occur in *µ*C++. □

5.4.1 Nonlocal Propagation

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 (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 5.1 coroutine C is started in its constructor so it can enable nonlocal exceptions and suspend after initialization. uMain then raises a nonlocal exception 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 propagates from the suspend and is caught by the handler for E. 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 5.4.2).

5.4.2 Enabling/Disabling Propagation

*µ*C++ 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 5.3.2 and 5.7, p. 83), 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.

```
_Event E {};
_Coroutine C {
    void main() {
        try {
            _Enable {   // allow nonlocal exceptions
                . . . suspend(); . . .
            }           // disable all nonlocal exceptions
        } catch( E ) { . . . }
    }
  public:
    C() { resume(); }   // prime loop
    void mem() { resume(); }
};
void uMain::main() {
    C c;
    _Resume E() _At c; // exception pending
    c.mem();            // trigger exception
}
```

Figure 5.1: Nonlocal Propagation

_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 5.10.1, p. 86).

The μC++ kernel implicitly polls for nonlocal exceptions (and cancellation, see Section 6, p. 91) at these points:

- after a call to uBaseTask::yield,
- when an **_Enable** statement is encountered,
- when a uEnableCancel object is instantiated (see Section 6.2, p. 92)
- after a task migrates to another cluster,
- after a task unblocks if it blocked when trying to enter a monitor,
- after a task unblocks if it blocked on an **_Accept** statement,
- after a task unblocks if it blocked when acquiring a uLock,
- after a task unblocks if it blocked when trying to perform I/O,
- the first time a coroutine/task's main routine is executed,
- after uBaseCoroutine::suspend/uBaseCoroutine::resume return.

If this level of polling is insufficient, explicit polling is possible by calling:

```
int uBaseCoroutine::asyncpoll();
```

which returns the number of exceptions propagated by this call to poll. 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.

5.4.3 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 (see Section 5.8.1, p. 84). 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.

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 == . . . )
                    _Resume 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.

5.5 Handler

A handler catches a propagated exception and attempts to deal with the exceptional event. Each handler is associated with a particular block of code, called a **guarded block**. μC++ supports two kinds of handlers, termination and resumption, which match with the kind of raise. An unhandled exception is dealt with by an exception default-member (see Section 5.3.2, p. 73).

5.5.1 Termination

A **termination handler** is a corrective action *after* throwing an exception during execution of a guarded block. When a termination handler begins execution, the stack from the point of the throw up to and including the guarded block is unwound; hence, all block and routine activations on the stack at or below the guarded block are deallocated, including all objects contained in these activations. After a termination handler completes, i.e., it does not perform another throw, control continues after the guarded block it is associated with. A termination handler often only has approximate knowledge of where an exceptional event occurred in the guarded block (e.g., a failure in library code), and hence, any partial results of the guarded-block computation are suspect. 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 5.2 shows how C++ and μC++ throws an exception to a termination handler. The differences are using **_Throw** instead of **throw**, throwing the dynamic type instead of the static type, and requiring a special exception type for all exceptions.

5.5.2 Resumption

A **resumption handler** is an intervention action *after* resuming an exception during execution of a guarded block. When a resumption handler begins execution, the stack is *not* unwound; hence, all block and routine activations on the stack at or below the guarded block are retained, including all objects contained in these activations. After a resumption handler completes, i.e., it does not perform a throw, control returns after the raise statement initiating the propagation. To obtain precise knowledge of the exceptional event, information about the event and variables at the resumption raise are passed to the handler so it can effect a change before returning. Alternatively, the resumption handler may determine a correction is impossible and throw an exception, effectively changing the original resume

```
C++                                  μC++
class E {                            _Event E {
  public:                              public:
    int i;                               int i;
    E( int i ) : i( i ) {}               E( int i ) : i( i ) {}
};                                   };

void f() {                           void f() {
    throw E( 3 );                        _Throw E( 3 );
}                                    }
int main() {                         void uMain::main() {
    try {                                try {
        f();                                 f();
    } catch( E e ) {                     } catch( E e ) {
        cout << e.i << endl;                 cout << e.i << endl;
        throw;                               _Throw;
    } // try                             } // try
}                                    }
```

Figure 5.2: C++ versus μC++ Terminating Propagation

into a throw. 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.

To provide resumption, μC++ extends the **try** block to include resumption handlers, where the resumption handler is denoted by a **_CatchResume** clause at the end of a **try** block:

```
try {
    ...
}   _CatchResume( E1 & ) { ... }   // must appear before catch clauses
    // more _CatchResume clauses
    _CatchResume( ... ) { ... }    // must be last _CatchResume clause
    catch( E2 & ) { ... }          // must appear after _CatchResume clauses
    // more catch clauses
    catch( ... ) { ... }           // must be last catch clause
```

Any number of resumption handlers can be associated with a **try** block. All **_CatchResume** handlers must precede any **catch** handlers in a **try** statement. Like **catch**(...) (catch-any), **_CatchResume**(...) must appear at the end of the list of the resumption handlers. A resumption handler is, in effect, a nested routine called from the raise site when an exception is propagated with resumption; when the resumption handler completes, control returns back to the point of the raise. A resumption handler can access any types and variables visible in its local scope, but it cannot perform a **break**, **continue** or **return** from within the handler.

> □ A resumption handler is a corrective action so a computation can continue. If a correction is impossible, the resumption handler should throw an appropriate exception not step into an enclosing block to cause the stack to unwind. There may be recovery actions closer to the raise point that can better deal with the problem. □

Values at the raise site can be modified directly in the handler if variables are visible in both contexts, or indirectly through reference or pointer variables in the caught exception:

```
_Event E {
  public:
    int &r;                        // reference to something
    E( int &r ) : r( r ) {}
};
void f() {
    int x;
    ... _Resume E( x ); ...        // set exception reference to point to x
}
```

```
void g() {
    try {
        f();
    } _CatchResume( E &e ) {
        . . . e.r = 3; . . .          // change x at raise via reference r
    }
}
```

5.5.3 Termination/Resumption

The form of the raise dictates the set of handlers examined during propagation:

- terminating propagation (**_Throw**) only examines termination handlers (**catch**),
- resuming propagation (**_Resume**) only examines resumption handlers (**_CatchResume**). However, the standard default resumption handler converts resuming into terminating propagation (see Section 5.3.2, p. 73).

Often the set of exception types for termination and resumption handlers are disjoint because each exception type has a specific action. However, it is possible for the set of exception types in each handler set to overlap. For example, the exception type R appears in both the termination and resumption handler-sets:

```
_Event E {};
void rtn() {
    try {
        _Resume E();
    }   _CatchResume( E & ) { _Throw E(); }   // H1
        catch( E & ) { . . . }                 // H2
}
```

The body of the **try** block resumes exception-type E, which is caught by resumption-handler **_CatchResume**(E) and handler H1 is invoked. The blocks on the call stack are now (stack grows from left to right):

rtn → **try _CatchResume**(E),**catch**(E) → H1

Handler H1 throws E and the stack is unwound until the exception is caught by termination-handler **catch**(E) and handler H2 is invoked.

rtn → H2

The termination handler is available because resuming did not unwind the stack.

5.5.3.1 Recursive Resuming

Resuming does not unwind the stack. As result, handlers defined in previous scopes continue to be present during resuming propagation. In terminating propagation, the handlers in previous scopes disappear as the stack is unwound. In some languages with a resuming propagation [Mac77, BMZ92, Geh92], the presence of resumption handlers in previous scopes can cause a situation called **recursive resuming**. The simplest situation where recursive resuming can occur is when a handler for a resuming exception-type resumes the same exception, e.g.:

```
_Event R {};
void rtn() {
    try {
        _Resume R();
    } _CatchResume( R ) {. . . _Resume R(); . . .} // H1
}
```

Routine rtn sets up a try block for resuming exception-type R with resumption-handler **_CatchResume**(R), respectively. Handler H1 is invoked by the resume in the body of the **try** block, and the blocks on the call stack are:

rtn → **try _CatchResume**(R) → H1

Then handler H1 resumes an exception of type R again, which finds the handler just above it at **_CatchResume**(R) and invokes H1 again; these steps continue 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.

□ An implicit form of recursive resuming can occur if yield or asyncpoll is called from within the resumption handler. Each of these operations results in a check for delivered exceptions, which can then result in a call to another resumption handler. As a result, the stack can grow, possibly exceeding the task's stack size. In general, this error is rare because there is usually sufficient stack space and the number of delivered resuming exceptions is small. Nevertheless, care must be taken when calling yield or asyncpoll directly or indirectly from a resumption handler. □

5.5.3.2 Preventing Recursive Resuming

Recursive resuming is probably the only legitimate criticism against resuming propagation. 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. Because the resuming propagation suggested previously searches for a handler by simply going down the runtime stack one stack frame at a time, it has the recursive resuming problem. μC++ has a modified propagation mechanism that provides a solution to the recursive resuming problem. Furthermore, the mechanism is extended to cover concurrent exceptions.

The modified propagation mechanism goes down the execution stack one level at a time as it does normally to find a handler capable of handling the exception being propagated. However, during propagation all the resumption handlers at each guarded block being "visited" are marked ineligible (denoted by italics), whether or not a handler is found. The mark is cleared only if the exception is handled either by a termination or resumption handler.

How does this new propagation mechanism make a difference? Given the previous runtime stack:

rtn → *try _CatchResume(R)* → H1

the handler **_CatchResume**(R) is marked ineligible (italics) when R is caught at the **try** block and H1 is called. Hence, the exception cannot be handled by **_CatchResume**(R), and the recursion is avoided and the default action occurs for R. Essentially, μC++ follows a common rule for terminating and resuming propagation: between a raise and its handler, each handler is eligible only once.

In handling exceptions, it is common for routines to create additional guarded blocks. For example, if the above example is augmented to:

```
void f() {
    try {
        _Resume R();
    } _CatchResume( R ) { . . . }      // H2
}
void rtn() {
    try {
        _Resume R();
    } _CatchResume( R ) { f(); }      // H1
}
```

where f creates an additional resumption handler, the call stack is extended to the following:

rtn → *try _CatchResume(R)* → H1 → f → **try _CatchResume**(R) → H2

and H2 is selected as it is unmarked. Using this technique, it is still possible to construct infinite recursions with respect to propagation; i.e., μC++ resuming propagation does not preclude all infinite recursions, e.g.:

```
_Event R {};
void rtn() {
    try {
        _Resume R();
    } _CatchResume( R ) { rtn(); }  // H1
}
```

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

rtn → *try _CatchResume(R)* → rtn → **try _CatchResume**(R) → . . .

As a result, there is always an eligible handler to catch the next exception in the recursion. This situation is considered a programming error with respect to recursion not propagation.

There is an interesting interaction between resuming, defaultResume (see Section 5.3.2, p. 73), and throwing.

```
_Event R {};
void rtn() {
    try {
        _Resume R();         // resume not throw
    } catch( R & ) { . . . }  // H1, no _CatchResume!!!
}
```

which results in the following call stack:

rtn → **try**{} **catch**(R) → defaultResume

When R is resumed, there is no eligible handler (or there is a handler but it is marked ineligible). However, when the base of the stack is reached, defaultResume is called, and its default action is to throw R. Terminating propagation then unwinds the stack until there is a match with the **catch** clause in the **try** block, so the behaviour is same as the example in Section 5.5.3, p. 78.

Finally, 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 can be chosen to handle a delivered nonlocal exception, reflecting the fact that a new propagation has started.

5.5.3.3 Commentary

Of the few languages with resumption, the language Mesa [MMS79] is probably the only one that also solved the recursive resuming problem. The Mesa scheme prevents recursive resuming by not reusing a handler clause bound to a specific invoked block, i.e., once a handler is used as part of handling an exception, it is not used again. The propagation mechanism always starts from the top of the stack to find an unmarked handler for a resume exception. However, this unambiguous semantics is often described as confusing.

The following more complex program demonstrates how *μ*C++ and Mesa solve recursive resuming, but with different solutions:

```
_Event R1 {};
_Event R2 {};

void rtn() {
    try {
        try {
            try {
                _Resume R1();
            } _CatchResume( R2 ) { _Resume R1(); }  // H1  -- cycle between handler H2 & H1
        _CatchResume( R1 ) { _Resume R2(); }         // H2
    _CatchResume( R2 ) { . . . }                     // H3
}
```

The following stack is generated at the point when handler H2 is invoked by the raise of an exception of type R1 in the inner-most **try** block:

rtn → **try _CatchResume**(R2) → **try _CatchResume**(R1) → **try _CatchResume**(R2) → H2

Handler H2 now raises an exception of type R2. The potential infinite recursion occurs because there is handler **_CatchResume**(R2), which resumes an exception of type R1, while handler **_CatchResume**(R1) is still on the stack. Hence, handler H2 invokes H1 and vice versa with no base-case to stop the recursion.

*μ*C++ propagation prevents the infinite recursion by marking *both* visited resumption handlers as ineligible before invoking handler H2, e.g.:

rtn → **try _CatchResume**(R2) → *try _CatchResume(R1)* → *try _CatchResume(R2)* → H2

Therefore, when handler H2 resumes an exception of type R2 the next eligible handler is the one associated with H3. Mesa propagation prevents the infinite recursion by only marking an unhandled handler, i.e., a handler that has not returned, as ineligible, resulting in:

rtn → **try _CatchResume**(R2) → *try _CatchResume(R1)* → **try _CatchResume**(R2) → H2

Hence, when handler H2 resumes an exception of type R2 the next eligible handler is the one associated with H1. As a result, H1 resumes an exception of type R1 and there is no infinite recursion. However, the confusion with the Mesa semantics is that there is no longer any handler for R1 due to its marking, even though the nested **try** blocks appear to properly deal with this situation. In fact, looking at the static structure, a programmer might assume there is an infinite recursion between handlers 1 and 2, as they resume one another. These confusions results in a reticence by language designers to incorporate resuming facilities in new languages. However, as *μ*C++ shows, there are reasonable solutions to these issues, and hence, there is no reason to preclude resuming facilities.

5.6 Bound Exceptions

To allow for additional control over the handling of exceptions, *μ*C++ supports the notion of *bound exceptions*. This concept *binds* the object raising an exception with the raised exception; a reference to the object can be used in a handler clause for finer-grain matching, which is more consistent with the object-oriented design of a program.

5.6.1 C++ Exception-Handling Deficiencies

In C++, only the exception type of the raised exception is used when matching **catch** clauses; the object raising the exception does not participate in the matching. In many cases, it is important to know which object raised the exception type for proper handling. For example, when reading from a file object, the exception-type IOError may be raised:

```
file Datafile, Logfile;
try {
    . . . Datafile.read(); . . .
    . . . Logfile.read(); . . .
} catch ( IOError ) {
    // handle exception from which object ?
}
```

The **try** block provides a handler for IOError exceptions generated while reading file objects Datafile and Logfile. However, if either read raises IOError, it is impossible for the handler to know which object failed during reading. The handler can only infer the exception originates in some instance of the file class. If other classes throw IOError, the handler knows even less. Even if the handler can only be entered by calls to Datafile.read() and Logfile.read(), it is unlikely the handler can perform a meaningful action without knowing which file raised the exception. Finally, it would be inconvenient to protect each individual read with a **try** block to differentiate between them, as this would largely mimic checking return-codes after each call to read.

Similar to package-specific exceptions in Ada [Int95], it is beneficial to provide object-specific handlers, e.g.:

```
try {
    . . . Datafile.read(); . . .
    . . . Logfile.read(); . . .
} catch ( Datafile.IOError ) {
    // handle Datafile IOError
} catch ( Logfile.IOError ) {
    // handle Logfile IOError
} catch ( IOError ) {
    // handler IOError from other objects
}
```

The first two **catch** clauses qualify the exception type with an object to specialize the matching. That is, only if the exception is generated by the specified object does the match occur. It is now possible to differentiate between the specified files and still use the unqualified form to handle the same exception type generated by any other objects.

□ Bound exceptions cannot be trivially mimicked by other mechanisms. Deriving a new exception type for each file object (e.g., Logfile_IOError from IOError) 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( IOError e ) {          // pass file-object address at raise
    if ( e.obj == &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:

```
class B {...};              // base exception-type
class D : public B {...};   // derived exception-type
...
try {
    ... throw D(this);      // pass object address   | // bound form
} catch( D e ) {                                     | } catch( o1.D ) {
    if ( e.o == &o1 ) ...   // deal only with o1     |
    else throw              // reraise exception     |
} catch( B e ) {                                     | } catch( o2.B ) {
    if ( e.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 immediately terminates the current guarded block, which precludes the handler for the base exception-type in that guarded block from being considered. The bound form (on the right) matches the handler for the base exception-type. Therefore, the "catch first, then reraise" approach is an incomplete substitute for bound exceptions. □

5.6.2 Object Binding

In *μ*C++, every exception derived from the three basic exception types can potentially be bound. Binding occurs implicitly when using *μ*C++'s raising statements, i.e., **_Resume** and **_Throw**. In the case of a local raise, the binding is to the object in whose member routine the raise occurs. In the previous example, an exception raised in a call to Datafile.read() is bound to Datafile; an exception raised in a call to Logfile.read() is bound to Logfile. If the raise occurs inside a static member routine or in a free routine, there is no binding. In the case of a non-local raise, the binding is to the coroutine/task executing the raise.

5.6.3 Bound Handlers

Bound handlers provide an object-specific handler for a bound exception. Matching is specified by prepending the binding expression to the exception type using the "." field-selection operator; the "catch-any" handler, ..., does not have a bound form.

5.6.3.1 Matching

A bound handler matches when the binding at the handler clause is identical to the binding associated with the currently propagated exception *and* the exception type in the handler clause is identical to or a base-type of the currently propagated exception type.

Bound handler clauses can be mixed with normal (unbound) handlers; the standard rules of lexical precedence determine which handler matches if multiple are eligible. Any expression that evaluates to an *lvalue* is a valid binding for a handler, but in practice, it only makes sense to specify an object that has a member function capable of raising an exception. Such a binding expression may or may not be evaluated during matching, and in the case of multiple bound-handler clauses, in undefined order. Hence, care must be taken when specifying binding expressions containing side-effects.

5.6.3.2 Termination

Bound termination handlers appear in the C++ **catch** clause:

catch(*raising-object* . *exception-declaration*) { ... }

In the previous example, **catch**(Logfile.IOError) is a catch clause specifying a bound handler with binding Logfile and exception-type IOError.

5.6.3.3 Resumption

Bound resumption handlers appear in the *μ*C++ **_CatchResume** clause (see Section 5.5.2, p. 76):

_CatchResume(*raising-object* . *exception-declaration*) { ... }

5.7 Inheritance

Table 5.1 shows the forms of inheritance allowed among C++ types and μC++ exception-types. First, the case of *single* public inheritance among homogeneous kinds of exception type, i.e., base and derived type are the both **_Event**, is supported in μC++ (major diagonal), e.g.:

```
_Event Ebase {};
_Event Ederived : public Ebase {};  // homogeneous public inheritance
```

In this case, all implicit functionality matches between base and derived types, and hence, there are no problems. Public derivation of exception types is for building the exception-type hierarchy, 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:

```
_Event Ederived : private Ebase {};      // homogeneous private inheritance, not allowed
_Event Ederived : protected Ebase {};   // homogeneous protected inheritance, not allowed
```

because each exception type must appear in the exception-type hierarchy, and hence must be a subtype of another exception type. Neither **private** nor **protected** inheritance establishes a subtyping relationship.

base		*public only / NO multiple inheritance*
derived	struct/class	event
struct/class	√	X
event	√	√

Table 5.1: 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 struct/class allowed

_Event Ederived : 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 from exception types by other kinds of class, exception type, coroutine, mutex or task, is not allowed, e.g.:

```
_Event Ebase {};

struct StructDerived          : public Ebase {};    // not allowed
class ClassDerived            : public Ebase {};    // not allowed
_Coroutine CorDerived         : public Ebase {};    // not allowed
_Monitor MonitorDerived       : public Ebase {};    // not allowed
_Task TaskDerived             : public Ebase {};    // 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 {};
_Event Ederived               : 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 from **struct**/**class** for the same reason as single inheritance.

5.8 Predefined Exception Routines

C++ supplies several *builtin* routines to provide information and deal with problems during propagation. The semantics of these builtin routines changes in a concurrent environment.

5.8.1 terminate/set_terminate

The terminate routine is called implicitly in a number of different situations when a problem prevents successful propagation (see a C++ reference manual for a complete list of propagation problems). The most common propagation problem is failing to locate a matching handler. The terminate routine provides an indirect mechanism to call a terminate-handler, which is a routine of type terminate_handler:

```
typedef void (*terminate_handler)();
```

and is set using the builtin routine set_terminate, which has type:

```
terminate_handler set_terminate( terminate_handler handler) throw();
```

The previously set terminate-handler is returned when a new handler is set. The default terminate-handler aborts the program; a user-defined terminate-handler must also terminate the program, i.e., it may not return or raise an exception, but it can perform some action before terminating, e.g.:

```
void new_handler() {
    // write out message
    // terminate execution (abort/exit)
}
terminate_handler old_handler = set_terminate( new_handler );
```

In a sequential program, there is only one terminate-handler for the entire program, which can be set and restored as needed during execution.

In a concurrent program, having a single terminate-handler for all tasks does not work because the value set by one task can be changed by another task at any time. In other words, no task can ensure that the terminate-handler it sets is the one that is used during a propagation problem. Therefore, in μC++, each task has its own terminate-handler, set using the set_terminate routine. Hence, each task can perform some specific action when a problem occurs during propagation, but the terminate-handler must still terminate the program, i.e., no terminate-handler may return (see Section 7.2.2, p. 99). The default terminate-handler for each task aborts the program.

Notice, the terminate-handler is associated with a task versus a coroutine. The reason for this semantics is that the coroutine is essentially subordinate to the task because the coroutine is executed by the task's thread. While propagation problems can occur while executing on the coroutine's stack, these problems are best dealt with by the task executing the coroutine because the program must terminate at this point. In fact, for the propagation problem of failing to locate a matching handler, the coroutine implicitly raises the predefined exception uBaseCoroutine::UnhandledException in its last resumer coroutine (see Section 7.2.3.1, p. 99), which ultimately transfers back to a task that either handles this exception or has its terminate-handler invoked.

5.8.2 unexpected/set_unexpected

The unexpected routine is called implicitly for the specific propagation problem of raising an exception that does not appear in a routine's exception specification (**throw** list), e.g.:

```
int rtn(...) throw(Ex1) {      // exception specification
    ... throw Ex2; ...         // Ex2 not in exception specification
}
```

The unexpected routine provides an indirect mechanism to call an unexpected-handler, which is a routine of type unexpected_handler:

```
typedef void (*unexpected_handler)();
```

and is set using the builtin routine set_unexpected, which has type:

```
unexpected_handler set_unexpected( unexpected_handler handler) throw();
```

The previously set unexpected-handler is returned when a new handler is set. The default unexpected-handler calls the terminate routine; like a terminate-handler, a user-defined unexpected-handler may not return, but it can perform some action and either terminate *or* raise an exception, e.g.:

```
void new_handler() {
    // write out message
    // raise new exception
}
unexpected_handler old_handler = set_unexpected( new_handler );
```

In a sequential program, there is only one unexpected-handler for the entire program, which can be set and restored as needed during execution.

In a μC++ program, having a single unexpected-handler for all coroutines/tasks does not work for the same reason as for the terminate-handler, i.e., the value can change at any time. Because it is possible to handle this specific propagation-problem programmatically (e.g., raise an exception) versus terminating the program, a coroutine can install a handler and deal with this problem during propagation on its stack. Therefore, in μC++, each coroutine (and hence, task) has its own unexpected-handler, set using the set_unexpected routine. The default unexpected-handler for each coroutine/task calls the terminate routine.

5.8.3 uncaught_exception

The uncaught_exception routine returns true if propagation is in progress. In a μC++ program, the result of this routine is specific to the coroutine/task that raises the exception. Hence, the occurrence of propagation in one coroutine/task is independent of that occurring in any other coroutine/task. For example, a destructor may not raise a new exception if it is invoked during propagation; if it does, the terminate routine is called. It is possible to use uncaught_exception to check for this case and handle it differently from normal destructor execution, e.g.:

```
~T() {    // destructor
    if ( . . . && ! uncaught_exception() ) {       // prevent propagation problem
        // raise an exception because cleanup problem
    } else {
        // cleanup as best as possible
    }
}
```

5.9 Programming with Exceptions

Like many other programming features, an EHM aims to make certain programming tasks easier and improve the overall quality of a program. Indeed, choosing to use the EHM over other available flow control mechanisms is a tradeoff. For example, a programmer may decide to use exceptions over some conditional statement for clarity. This decision may sacrifice runtime efficiency and memory space. In other words, universal, crisp criteria for making a decision do not exist. Nevertheless, some important guidelines are given to encourage good use of exceptions.

First, use exceptions to indicate exceptional event in library code to ensure a library user cannot ignoring the event, as is possible with return codes and status values. Hence, exceptions improve safety and robustness, while still allowing a library user to explicitly catch and do nothing about an exception. Second, use exceptions to improve clarity and maintainability over techniques like status return values and status flags where normal and exceptional control-flow are mixed together within a block. Using exceptions not only separates the normal flow in a guarded block from the exceptional flow in handlers, but also avoids mixing normal return-values with exceptional return-values. This separation makes subsequent changes easier. Third, use exceptions to indicate conditions that happen rarely at runtime for the following reasons:

- The normal flow of the program should represent what should happen most of the time, allowing programmers to easily understand the common functionality of a code segment. The exceptional flow then represents subtle details to handle rare situations, such as boundary conditions.

- Because the propagation mechanism requires a search for the handler, it is usually expensive. Part of the cost is a result of the dynamic choice of a handler. Furthermore, this dynamic choice can be less understandable than a normal routine call. Hence, there is a potential for high runtime cost with exceptions and control flow can be more difficult to understand. Nevertheless, the net complexity is reduced using exceptions compared to other approaches.

5.9.1 Terminating Propagation

Typical use of terminating propagation is for graceful termination of an operation, coroutine, task or program. Termination is graceful if it triggers a sequence of cleanup actions in the execution context. Examples of abrupt (or non-graceful) termination include the uAbort routine (abort in C) and the kill -9 command in UNIX. Graceful termination is more important in a concurrent environment because one execution can terminate while others continue. The terminating operation must be given a chance to release any shared resources it has acquired (the cleanup action) in order to maintain the integrity of the execution environment. For example, deadlock is potentially a rare condition and a thrown exception can force graceful termination of an operation, consequently leading to the release of some shared resources and breaking of the deadlock.

5.9.2 Resuming Propagation

Typical use of resuming propagation is to do additional computation, in the form of a resumption handler, for an exceptional event or as a form of polymorphism, where an action is left unspecified (e.g., in a library routine) and specified by a user using dynamic lookup (similar to a virtual routine in a class). The additional computation may modify the state of the execution, which can be useful for error recovery. Alternatively, it may cause information about the execution to be gathered and saved as a side-effect without effectively modifying the execution's computation.

5.9.3 Terminating/Resuming Propagation

Any exception type can be both thrown or resumed. When feasible, it is best to initially resume an exception-type to avoid loss of local information. If no resumption handler handles the exception, the same exception-type can be thrown (default action). For example, in a real-time application, missing a real-time constraint, say an execution cannot finish before a deadline, is considered an exceptional event. For some applications, the constraint violation can result in termination. Other applications can modify internal parameters to increase execution by sacrificing the quality of the solution or by acquiring more computing resources to speedup execution.

5.10 Predefined Exception-Types

μC++ provides a number of predefined exception-types, which are structured into the hierarchy in Figure 5.3, 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 errors arise performing network and file I/O. Only the kernel exception-types are discussed, as the I/O exception-types are OS specific.

5.10.1 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 uMutexFailure 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 blocked on an **_Accept** or a wait. The predefined exception-type uMutexFailure::RendezvousFailure is implicitly enabled and polling is performed when an acceptor task restarts after blocking for a rendezvous to finish.

5.10.2 Unhandled Exception in Coroutine

An exception raised and not handled inside a coroutine terminates it and implicitly resumes a nonlocal exception of type uBaseCoroutine::UnhandledException at the coroutine's last resumer rather than performing the default action (which may abort the program). For example, in:

```
uBaseEvent
    uKernelFailure
        uMutexFailure
            uMutexFailure::EntryFailure
            uMutexFailure::RendezvousFailure
        uCondition::WaitingFailure
        uBaseCoroutine::Failure
            uBaseCoroutine::UnhandledException
        uPthreadable::Failure
            uPthreadable::CreationFailure
    uIOFailure
        uFile::Failure
            uFile::TerminateFailure
            uFile::StatusFailure
            uFile::FileAccess::Failure
                uFile::FileAccess::OpenFailure
                uFile::FileAccess::CloseFailure
                uFile::FileAccess::SeekFailure
                uFile::FileAccess::SyncFailure
                uFile::FileAccess::ReadFailure
                    uFile::FileAccess::ReadTimeout
                uFile::FileAccess::WriteFailure
                    uFile::FileAccess::WriteTimeout
        uSocket::IPConvertFailure
        uSocket::Failure
            uSocket::OpenFailure
            uSocket::CloseFailure
            uSocketServer::Failure
                uSocketServer::OpenFailure
                uSocketServer::CloseFailure
                uSocketServer::ReadFailure
                    uSocketServer::ReadTimeout
                uSocketServer::WriteFailure
                    uSocketServer::WriteTimeout
                uSocketServer::SendfileFailure
                    uSocketServer::SendfileTimeout
            uSocketAccept::Failure
                uSocketAccept::OpenFailure
                    uSocketAccept::OpenTimeout
                uSocketAccept::CloseFailure
                uSocketAccept::ReadFailure
                    uSocketAccept::ReadTimeout
                uSocketAccept::WriteFailure
                    uSocketAccept::WriteTimeout
                uSocketAccept::SendfileFailure
                    uSocketAccept::SendfileTimeout
            uSocketClient::Failure
                uSocketClient::OpenFailure
                    uSocketClient::OpenTimeout
                uSocketClient::CloseFailure
                uSocketClient::ReadFailure
                    uSocketClient::ReadTimeout
                uSocketClient::WriteFailure
                    uSocketClient::WriteTimeout
                uSocketClient::SendfileFailure
                    uSocketClient::SendfileTimeout
    uBaseFuture<T>::Cancellation
```

Figure 5.3: μC++ Predefined Exception-Type Hierarchy

```
_Event E {};

_Coroutine C {
    void main() { _Throw E(); }
  public:
    void mem() { resume(); }
};
void uMain::main() {
    C c;
    try {
        c.mem();
    } catch ( uBaseCoroutine::UnhandledException ) {
        . . .
    }
}
```

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 terminated and the nonlocal exception of type uBaseCoroutine::UnhandledException is implicitly raised at c's last resumer, uMain, rather than c's starter. (This action replaces the normal call to defaultTerminate/defaultResume (see Section 5.3.2, p. 73) because the exception has been caught and transformed.) This behaviour 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 (coroutine or task) 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.

In many cases, the resumer of a coroutine is unaware of the resumed coroutine's implementation, and hence, cannot respond directly to unhandled exceptions forwarded from the resumed coroutine, which is why the general uBaseCoroutine::UnhandledException exception is raised at the resumer rather than the specific coroutine exception. To provide for the case where a resumer does know about exceptions that may be forwarded from a resumed coroutine, the exception uBaseCoroutine::UnhandledException contains a copy of the initial exception not caught in the resumed coroutine. When handling a uBaseCoroutine::UnhandledException, a resumer can trigger the copied exception in the handler using member routine triggerCause. For example, in:

```
. . . as above . . .
void uMain::main() {
    C c;
    try {
        c.mem();             // first call fails
    } catch ( uBaseCoroutine::UnhandledException &ex ) {
        ex.triggerCause();   // trigger copied exception
    }
}
```

the call to c.mem indirectly _Resumes a nonlocal exception of type uBaseCoroutine::UnhandledException, because c.main does not handle exceptions of type E. uMain::main has no resumption handler for uBaseCoroutine::UnhandledException; hence, its default handler _Throws the exception (see Section 5.3.2, p. 73), which is caught. The handler triggers a copy of the initial exception of type E, which is raised in exactly the same way as the raise in the resumed coroutine (i.e., matching _Throw or _Resume). In this way, the resumer coroutine can use all exception matching mechanisms provided by μC++ to identify the initial exception.

5.10.3 Breaking a Rendezvous

As mentioned in Section 2.9.2.2, p. 23, the accept statement forms a rendezvous between the acceptor and the accepted tasks, where a rendezvous is a point in time at which both tasks wait for a section of code to execute before continuing. It can be crucial to correctness that the acceptor know if the accepted task does not complete the rendezvous code, otherwise the acceptor task continues under the incorrect assumption that the rendezvous action has occurred. To this end, an exception of type uMutexFailure::RendezvousFailure is raised at the acceptor task if the accepted member terminates abnormally. It may also be necessary for a mutex member to know if the acceptor has restarted, and hence, the rendezvous has ended. This situation can happen if the mutex member calls a private member, which may conditionally wait, which ends the rendezvous. The macro uRendezvousAcceptor can be used only inside mutex types to determine if a rendezvous has ended:

```
    uBaseCoroutine *uRendezvousAcceptor();
```

It returns NULL if the rendezvous is ended; otherwise it returns the address of the rendezvous partner. In addition, calling uRendezvousAcceptor has the side effect of cancelling the implicit resume of uMutexFailure::RendezvousFailure at the acceptor. This capability allows a mutex member to terminate with an exception without informing the acceptor.

Chapter 6

Cancellation

Cancellation is a mechanism to safely terminate the execution of a coroutine or task. Any coroutine/task may cancel itself or another coroutine/task by calling uBaseCoroutine::cancel() (see Section 2.7.2, p. 15). ***The deletion of a non-terminated coroutine (see page 14) implicitly forces its cancellation.*** Cancelling a coroutine/task does not result in immediate cancellation of the object; cancellation only begins when the coroutine/task encounters a **cancellation checkpoint**, such as asyncpoll() or uBaseTask::yield() (see page 75 for a complete list), which starts the cancellation for the cancelled object. Note, all cancellation points are polling points for asynchronous exceptions and vice-versa. The more frequently cancellation checkpoints are encountered, the timelier the cancellation starts. There is no provision to "uncancel" a coroutine/task once it is cancelled. However, it is possible for the cancelled coroutine/task to control if and where cancellation starts (see Section 6.2).

Once cancellation starts, the stack of the coroutine/task is unwound, which executes the destructors of objects allocated on the stack ***as well as catch-any exception handlers*** (i.e., **catch** (...)). Executing this additional code during unwinding allows safe cleanup of any objects declared in a cancelled coroutine/task via their destructors, and supports the common C++ idiom of using catch-any handlers to perform cleanup actions during exceptional control-flow and then reraising the exception. The C++ idiom follows from the fact that a catch-any handler has no specific information about an exception, and hence, cannot properly handle it; therefore, it only makes sense to execute local cleanup in the catch-any handler and continue propagation by reraising the exception so a specific handler can be found.

There are two scenarios in which a catch-any handler may finish. In the first scenario, all exceptions raised directly or indirectly from a guarded block are handled by a common action, with normal program execution continuing after it. However, using a catch-any handler to specify the common action is considered poor style. If a group of exceptions has a common handling action, it is highly likely all its members are logically related, and hence, should be structured into an exception hierarchy (see Section 5.7, p. 83) allowing all the group members to be caught by the hierarchy's root rather than a catch-any. In the second scenario, all exceptions are caught at a high level (often the top-most level) in a task in order to prevent the program's termination due to an uncaught exception. In this case, code after the handler is often finalization/restart code to be performed unconditionally before ending or restarting the task.

Unlike a nonlocal exception (see Section 5.4, p. 73), cancellation cannot be caught or stopped unless the cleanup code aborts the program, which is the ultimate termination of all coroutines/tasks. Therefore, if a catch-any handler finishes during cancellation, i.e., without throwing or rethrowing, the only logical behaviour is for stack unwinding to continue. This behaviour is different from normal completion of a catch-any handler, which continues after the handler. The correctness of a program relying on execution to continue after a catch-any handler for convenience (scenario 1) or restart (scenario 2) reasons is unaffected since cancellation ultimately terminates the task, and hence, normal execution/restart cannot be expected. However, the correctness of programs relying on execution to continue after a catch-any handler for finalization reasons (scenario 2) is compromised by cancellation. Such programs are incompatible with cancellation as control cannot logically continue after the handler. In this situation, the program must be restructured to check for cancellation and invoke the finalization code within the handler. It is possible to programmatically check for an ongoing cancellation by calling routine uBaseCoroutine::cancelInProgress (see Section 2.7.2, p. 15) during cleanup (catch-any handler or destructor), which is analogous to using std::uncaught_exception.

Cancellation does not work if a *new* exception is thrown inside a catch-any handler because the stack unwinding due to termination cannot be altered, e.g.:

```
catch (. . .) {
    . . . _Throw anotherException(); . . .
}
```

The resulting program behaviour is undefined, and such constructs must be avoided if cancellation is to be used. Again, routine uBaseCoroutine::cancelInProgress (see Section 2.7.2, p. 15) can be used to check for this situation, so the throw can be conditional. Alternatively, ensuring a coroutine's main routine terminates prevents implicit cancellation.

6.1 Using Cancellation

Cancellation is used in situations where the work of a task is not required any more and its resources should be freed. Figure 6.1 shows a generic example in which a solution space is divided into sub-domains and worker tasks are dispatched to search their respective sub-domain for a suitable solution. For this particular problem class, any specific solution is sufficient. In the program, after uMain creates the tasks, it waits for a solution to be found by any of the Worker tasks. If a Worker task finds a solution, it stores it in the Result monitor and restarts uMain (if appropriate). Since a solution has been found, the other worker solutions are not required and allowing these workers to proceed is a waste of resources. Hence, uMain marks them all for cancellation and uses the result. After the result has been processed, uMain deletes the worker tasks, which allows for execution overlap of result processing with the worker tasks detecting, starting, and finishing cancellation. Alternatively, uMain can delete the workers right away, with the consequence that it may have to wait for the worker tasks to finish cancellation before processing the result.

6.2 Enabling/Disabling Cancellation

A cancellee may not stop cancellation once in progress, but it can control when the cancellation starts. The ability to defer the start of cancellation can be used to ensure a block of code is completely executed, similar to enabling/disabling propagation (see Section 5.4.2, p. 74).

By default, cancellation is implicitly enabled for a coroutine/task (which is the opposite of nonlocal exceptions). Explicitly enabling/disabling cancellation is controlled by declaring an instance of one of the following types: uEnableCancel or uDisableCancel. The object's constructor saves the current cancellation state (enabled or disabled) and sets the state appropriately; the object's destructor resets the cancellation state to the previous state, e.g.:

```
{
    uDisableCancel cancelDisable;    // save current state, set to disable (variable name unimportant)
    . . .
    {
        uEnableCancel cancelEnable; // save current state, set to enable (variable name unimportant) and
        . . .                       // implicit poll/cancellation checkpoint
    }   // revert back to disabled
    . . .
}   // revert back to previous cancellation status
```

Note, creating an instance of uEnableCancel is a cancellation checkpoint, which polls for both cancellation *and* asynchronous exceptions.

6.3 Commentary

Despite their similarities, cancellation and nonlocal exceptions are fundamentally different mechanisms in μC++. As a result, the approach of using **_Enable**/**_Disable** with a special uCancellation type to control cancellation delivery was rejected, e.g.:

```
_Enable <uCancellation> <. . .> /* asynchronous exceptions */ {
    . . .
}
```

This approach is rejected because it suggests cancellation is part of the exception handling mechanism represented by the exception type uCancellation, which is not the case. There is no way to raise or catch a cancellation as there is with exceptions. In addition, the blanket **_Enable**/**_Disable**, which applies to all nonlocal exceptions, does not affect cancellation.

```
const int NumOfWorkers = 16;
const unsigned int Domain = 0xffffffff;

_Monitor Result {
    int res;
    uCondition c;
  public:
    Result() : res(0) {}
    int getResult() {
        if ( res == 0 ) c.wait();            // wait if no result has been found so far
        return res;
    }
    void finish( int r ) {
        res = r;                             // store result
        c.signal();                          // wake up uMain
    }
};

_Task Worker {
    Result &r;
    int subdomain;
  public:
    Worker( int sub, Result &res ) : subdomain( sub ), r( res ) {}
    void main() {
        int finalresult;
        // perform calculations with embedded cancellation checkpoints
        r.finish( finalresult );             // if result is found, store it in Result
    }
};

void uMain::main () {
    Worker *w[NumOfWorkers];
    Result r;
    for ( int i = 0; i < NumOfWorkers; i += 1 ) {
        w[i] = new Worker( i * Domain / NumOfWorkers, r );  // create worker tasks
    }
    int result = r.getResult();
    for ( int i = 0; i < NumOfWorkers; i += 1 ) {
        w[i]->cancel();                      // mark workers for cancellation
    }
    // do something with the result
    for ( int i = 0; i < NumOfWorkers; i += 1 ) {
        delete w[i];                         // only block if cancellation has not terminated worker
    }
}
```

Figure 6.1: Cancellation Example

Chapter 7

Errors

The following are examples of the static/dynamic warnings/errors that can occur during the compilation/execution of a *μ*C++ program.

7.1 Static (Compile-time) Warnings/Errors

These static warnings/errors are generated by the *μ*C++ translator not by the underlying C++ compiler. These warnings/errors are specific to usage problems with the *μ*C++ concurrency extensions. The following examples show different situations, the message generated and an explanation of the message. While not all warning/error situations are enumerated, the list covers the common one present in most *μ*C++ programs.

The following program:

```
_Task T {
  public:
    void mem() {}
  private:
    void main() {
      fini:
        for ( int i = 0; i < 10; i += 1 ) {
            _Accept( mem ) {
                break fini;
            } _Else;
        }
    }
};
```

generates these warnings when using the -Wall compiler flag (actually generated by the C++ compiler not *μ*C++):

```
test.cc:17: warning: label '_U_C_fini' defined but not used
test.cc:11: warning: label '_U_L000001' defined but not used
test.cc:8: warning: label 'fini' defined but not used
```

These warning messages appear due to the way *μ*C++ generates code. Labels are generated in a number of places but are not always used depending on what happens later in the code. It is too difficult to detect all these cases and remove the labels that are unnecessary. All of these kinds of warnings can be suppressed by adding the extra flag:

```
-Wall -Wno-unused-label
```

The following program:

```
_Task T {
    void main() {
        _Accept( mem );
    }
  public:
    void mem() {}
};
```

generates this error:

```
test.cc:4: uC++ Translator error: accept on a nomutex member "mem", possibly caused by accept state-
ment appearing before mutex-member definition.
```

because the accept of member mem appears *before* the definition of member mem, and hence, the μC++ translator encounters the identifier mem before it knows it is a mutex member. C++ requires definition before use in most circumstances.

The following program:

```
_Task T {
  public:
    void mem() {}
  private:
    void main() {
        _Accept( mem );
        or _Accept( mem );
    }
};
```

generates this error:

```
test.cc:8: uC++ Translator error: multiple accepts of mutex member "mem".
```

because the accept statement specifies the same member, mem, twice. The second specification is superfluous.

The following program:

```
_Task T1 {};
_Task T2 {
  private:
    void main() {
        _Accept( ~T1 );
    }
};
```

generates this error:

```
test.cc:6: uC++ Translator error: accepting an invalid destructor; destructor name must be the same as the
containing class "T2".
```

because the accept statement specifies the destructor from a different class, T1, within class T2.

The following program:

```
_Mutex class M {};
_Coroutine C : public M {};
_Task T1 : public C {};
_Task T2 : public M, public C {};
```

generates these errors:

```
test.cc:3: uC++ Translator error: derived type "C" of kind "COROUTINE" is incompatible with the base type
"M" of kind "MONITOR"; inheritance ignored.
test.cc:4: uC++ Translator error: derived type "T1" of kind "TASK" is incompatible with the base type "C" of
kind "COROUTINE"; inheritance ignored.
test.cc:5: uC++ Translator error: multiple inheritance disallowed between base type "M" of kind "MONITOR"
and base type "C" of kind "COROUTINE"; inheritance ignored.
```

because of inheritance restrictions among kinds of types in μC++ (see Section 2.15, p. 34).

Similarly, the following program:

```
_Event T1 {};
_Event T2 : private T1 {};
_Event T3 : public T1, public T2 {};
```

generates these errors:

```
test.cc:3: uC++ Translator error: non-public inheritance disallowed between the derived type "T2" of kind
"EVENT" and the base type "T1" of kind "EVENT"; inheritance ignored.
test.cc:4: uC++ Translator error: multiple inheritance disallowed between base type "T1" of kind "EVENT"
and base type "T2" of kind "EVENT"; inheritance ignored.
```

because of inheritance restrictions among exception types in μC++ (see Section 5.7, p. 83).

The following program:

```
_Task T;              // prototype
_Coroutine T {};      // definition
```

generates this error:

```
test.cc:3: uC++ Translator error: ”T” redeclared with different kind.
```

because the kind of type for the prototype, **_Task**, does not match the kind of type for the definition, **_Coroutine**.

The following program:

```
_Mutex class M1 {};
_Mutex class M2 {};
_Mutex class M3 : public M1, public M2 {}; // multiple inheritance
```

generates this error:

```
test.cc:4: uC++ Translator error: multiple inheritance disallowed between base type ”M1” of kind ”MONI-
TOR” and base type ”M2” of kind ”MONITOR”; inheritance ignored.
```

because only one base type can be a mutex type when inheriting.

The following program:

```
_Task T {
  public:
    _Nomutex void mem();
};
_Mutex void T::mem() {}
```

generates this error:

```
test.cc:6: uC++ Translator error: mutex attribute of ”T::mem” conflicts with previously declared nomutex
attribute.
```

because the kind of mutual exclusion, **_Nomutex**, for the prototype of mem, does not match the kind of mutual exclusion, **_Mutex**, for the definition.

The following program:

```
_Task T {
  public:
    _Nomutex T() {}                              // must be mutex
    _Mutex void *operator new( size_t ) {}       // must be nomutex
    _Mutex void operator delete( void * ) {}     // must be nomutex
    _Mutex static void mem() {}                  // must be nomutex
    _Nomutex ~T() {}                             // must be mutex
};
```

generates these errors:

```
test.cc:4: uC++ Translator error: constructor must be mutex, nomutex attribute ignored.
test.cc:5: uC++ Translator error: ”new” operator must be nomutex, mutex attribute ignored.
test.cc:6: uC++ Translator error: ”delete” operator must be nomutex, mutex attribute ignored.
test.cc:7: uC++ Translator error: static member ”mem” must be nomutex, mutex attribute ignored.
test.cc:9: uC++ Translator error: destructor must be mutex for mutex type, nomutex attribute ignored.
```

because certain members may or may not have the mutex property *for any mutex type*. The constructor(s) of a mutex type must be mutex because the thread of the constructing task is active in the object. Operators **new** and **delete** of a mutex type must be nomutex because it is superfluous to make them mutex when the constructor and destructor already ensure the correct form of mutual exclusion. The **static** member(s) of a mutex type must be nomutex because it has no direct access to the object's mutex properties, i.e., there is no **this** variable in a **static** member to control the mutex object. Finally, a destructor must be mutex if it is a member of a mutex type because deletion requires mutual exclusion.

The following program:

```
_Mutex class T1;
class T1 {};                // conflict between forward and actual qualifier

class T2 {};
_Mutex class T2;            // conflict between forward and actual qualifier

_Mutex class T3;            // conflicting forward declaration qualifiers
_Nomutex class T3;          // ignore both forward declaration qualifiers

_Mutex class T4 {
      void mem( int );      // default nomutex
   public:
      void mem(int, int);   // default mutex
};
```

generates these errors:

```
test.cc:3: uC++ Translator error: may not specify both mutex and nomutex attributes for a class. Ignoring
previous declaration.
test.cc:6: uC++ Translator error: may not specify both mutex and nomutex attributes for a class. Ignoring
this declaration.
test.cc:9: uC++ Translator error: may not specify both mutex and nomutex attributes for a class. Assuming
default attribute.
test.cc:14: uC++ Translator error: mutex attribute of ”T4::mem” conflicts with previously declared nomutex
attribute.
```

because there are conflicts between mutex qualifiers. For type T1, the mutex qualifier for the forward declaration does not match with the actual declaration because the default qualifier for a **class** is **_Nomutex**. For type T2, the mutex qualifier for the later forward declaration does not match with the actual declaration for the same reason. For type T3, the mutex qualifiers for the two forward declarations are conflicting so they are ignored at the actual declaration. For mutex type T4, the default mutex qualifiers for the overloaded member routine, mem, are conflicting because one is private, default **_Nomutex**, the other is public, default **_Mutex**, and *μ*C++ requires overloaded members to have identical mutex properties (see Sections 2.9.2.1, p. 21 and 2.19, p. 44).

The following program:

```
_Task /* no name */ {};
```

generates this error:

```
test.cc:2: uC++ Translator error: cannot create anonymous coroutine or task because of the need for
named constructors and destructors.
```

because a type without a name cannot have constructors or destructors since both are named after the type, and the *μ*C++ translator needs to generate constructors and destructors if not present for certain kinds of types.

7.2 Dynamic (Runtime) Warnings/Errors

These dynamic warnings/errors are generated by the *μ*C++ runtime system not by the C++ runtime system. These warnings/errors are specific to usage problems with the *μ*C++ concurrency extensions. The following examples show different situations, the message generated and an explanation of the message. While not all warning/error situations are enumerated, the list covers the common one present in most *μ*C++ programs.

7.2.1 Assertions

Assertions define runtime checks that must be true or the basic algorithm is incorrect; if the assertion is false, a message is printed and the program is aborted. Assertions are written using the macro assert:

```
assert( boolean-expression );
```

Asserts can be turned off by defining the preprocessor variable NDEBUG before including assert.h.

To use assertions in a *μ*C++ program, include the file:

```
#include <assert.h>
```

7.2.2 Termination

A μC++ program can be terminated due to a failure using the UNIX routine abort, which stops all thread execution and generates a core file for subsequent debugging (assuming the shell limits allow a core file to written). To terminate a program, generate a core file, *and* print an error message, use the μC++ free routine uAbort:

```
void uAbort( char *format = "", ... )
```

format is a string containing text to be printed and printf style format codes describing how to print the following variable number of arguments. The number of elements in the variable argument list must match with the number of format codes, as for printf. In addition to printing the user specified message, which normally describes the error, routine uAbort prints the name of the currently executing task type, possibly naming the type of the currently executing coroutine if the task's thread is not executing on its own execution state at the time of the call.

A μC++ program can be terminated using the UNIX routine exit, which stops all thread execution and returns a status code to the invoking shell:

```
void exit( int status );
```

Note, when exit is used to terminate a program, all global destructors are still executed. Any tasks, clusters, or processors not deleted by this point ***are not flagged with an error***, unlike normal program termination.

□ Because routine exit eliminates some error checking, it should not be used to end uMain::main to pass back a return code to the shell, e.g.:

```
void uMain::main() {
    . . .
    exit( EXIT_SUCCESS );
}
```

Use the variable uRetCode from uMain::main instead (see Section 2.2, p. 8). □

7.2.3 Messages

The following examples show different error situations, the error message generated and an explanation of the error. While not all error situations are enumerated, the list covers the common errors present in most μC++ programs. Finally, most of these errors are generated only when using the -debug compilation flag (see Section 2.4.1, p. 10).

7.2.3.1 Default Actions

The following examples show the default actions taken when certain exceptions are not caught and handled by the program (see Section 5.3.2, p. 73). For coroutines the default action raises another exception at the coroutine's resumer, while for tasks the default action prints an appropriate error message and terminates the program. While not all default actions are enumerated, the list covers the common problems present in many μC++ programs.

The following program:

```
void f() throw() { // throw no exceptions
    throw 1;
}
void uMain::main() {
    f();
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:20242) Exception propagated through a function whose exception-specification
does not permit exceptions of that type. Type of last active exception: int Error occurred while executing
task uMain (0xffbef828).
```

because routine f defines it raises no exceptions and then an exception is raised from within it.

The following program:

```
_Event E {};

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

generates this error:

```
uC++ Runtime error (UNIX pid:23979) (uBaseCoroutine &)0xffbef008 : Unhandled exception in coroutine
uMain raised non-locally from resumed coroutine C (0x82970), which was terminated due to an unhandled
exception of type E. Error occurred while executing task uMain (0xffbef008).
```

because the call to c.mem resumes coroutine c and then coroutine c throws an exception it does not handle. As a result, when the top of c's stack is reached, an exception of type uBaseCoroutine::UnhandledException is raised at uMain, since it last resumed c. A more complex version of this situation occurs when there is a resume chain and no coroutine along the chain handles the exception. The following program:

```
_Event E {};

_Coroutine C2 {
    void main() { _Throw E(); }
  public:
    void mem() { resume(); }
};
_Coroutine C1 {
    void main() {
        C2 c2;
        c2.mem();
    }
  public:
    void mem() { resume(); }
};
void uMain::main() {
    C1 c1;
    c1.mem();                   // first call fails
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:24080) (uBaseCoroutine &)0xffbef008 : Unhandled exception in coroutine
uMain raised non-locally from coroutine C1 (0x82ec0), which was terminated due to a series of unhandled
exceptions – originally an unhandled exception of type E inside coroutine C2 (0x8acc0). Error occurred
while executing task uMain (0xffbef008).
```

because the call to c1.mem resumes coroutine c1, which creates coroutine c2 and call to c2.mem to resume it, and then coroutine c2 throws an exception it does not handle. As a result, when the top of c2's stack is reached, an exception of type uBaseCoroutine::UnhandledException is raised at uMain, since it last resumed c.

The following program:

```
void uMain::main() {
    throw 1;
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:13901) Propagation failed to find a matching handler. Possible cause is
a missing try block with appropriate catch clause for specified or derived exception type, or throwing an
exception from within a destructor while propagating an exception. Type of last active exception: int Error
occurred while executing task uMain (0xffbef000).
```

because no **try** statement with an appropriate **catch** clause is in effect for task uMain so propagation fails to locate a matching handler.

The following program:

```
void uMain::main() {
    throw;        // rethrow
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:13291) Attempt to rethrow/reresume but no active exception. Possible
cause is a rethrow/reresume not directly or indirectly performed from a catch clause. Error occurred while
executing task uMain (0xffbef000).
```

because a rethrow must occur in a context with an active (already raised) exception so that exception can be raised again.

The following program:

```
_Monitor M {
  public:
    void mem() {}
};
void uMain::main() {
    M *m = new M;
    delete m;               // delete storage
    m->mem();               // make call to mutex member
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:4258) (uSerial &)0x670c00 : Entry failure while executing mutex destructor:
mutex object has been destroyed. Error occurred while executing task uMain (0xffbef000).
```

because task uMain deletes the monitor m and then calls the member routine mem through the deleted pointer. As a result, task uMain finds the mutex object has been destroyed.

The following program:

```
_Task T1 {
    uCondition w;
  public:
    void mem() { w.wait(); }
  private:
    void main() {
        _Accept( mem );     // let T2 in so it can wait
        w.signal();         // put T2 on acceptor/signalled stack
        _Accept( ~T1 );     // uMain is calling the destructor
    }
};
_Task T2 {
    T1 &t1;
    void main() { t1.mem(); }
  public:
    T2( T1 &t1 ) : t1( t1 ) {}
};
void uMain::main() {
    T1 *t1 = new T1;
    T2 *t2 = new T2( *t1 );
    delete t1;              // delete in same order as creation
    delete t2;
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:23337) (uSerial &)0x84470 : Entry failure while executing mutex destructor:
task uMain (0xffbef008) found blocked on acceptor/signalled stack. Error occurred while executing task T2
(0x8d550).
```

because task t2 is allowed to wait on condition variable w in t1.mem, and then task t1 signals condition w, which moves task t2 to the acceptor/signalled stack, and accepts its destructor. As a result, when task uMain attempts to delete task t1, it finds task t2 still blocked on the acceptor/signalled stack. Similarly, the following program:

```
_Task T1 {
  public:
    void mem() {}
  private:
    void main() { _Accept( ~T1 ); }
};
_Task T2 {
    T1 &t1;
  public:
    T2( T1 &t1 ) : t1( t1 ) {}
  private:
    void main() { t1.mem(); }
};
void uMain::main() {
    T1 *t1 = new T1;
    T2 *t2 = new T2( *t1 );
    delete t1;
    delete t2;
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:23425) (uSerial &)0x84230 : Entry failure while executing mutex destructor:
task uMain (0xffbef008) found blocked on entry queue. Error occurred while executing task T2 (0x8d310).
```

because task t2 happens to block on the call to t1.mem, and then task t1 accepts its destructor. As a result, when task uMain attempts to delete task t1, it finds task t2 still blocked on the entry queue of t1.

The following program:

```
_Event E {};

_Task T {
    uBaseTask &t;
  public:
    T( uBaseTask &t ) : t( t ) {}
    void mem() {
//          uRendezvousAcceptor();
        _Throw E();
    }
  private:
    void main() {
        _Accept( mem );
    }
};
void uMain::main() {
    T t( uThisTask() );
    try {
        t.mem();
    } catch( E &e ) {
    }
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:23512) (uSerial &)0x83120 : Rendezvous failure in accepted call from task
uMain (0xffbef008) to mutex member of task T (0x82ff0). Error occurred while executing task T (0x82ff0).
```

because in the call to t.mem from task uMain, the rendezvous terminates abnormally by raising an exception of type E. As a result, uMain implicitly resumes an exception of type uMutexFailure::RendezvousFailure concurrently at task t so it knows the call did not complete and can take appropriate corrective action (see Section 5.10.3, p. 88). If the call uRendezvousAcceptor() is uncommented, an exception of type uMutexFailure::RendezvousFailure is not resumed at task t, and task t restarts as if the rendezvous completed. A more complex version of this situation occurs when a blocked call is aborted, i.e., before the call even begins. The following program:

```
_Event E {};

_Task T {
    uBaseTask &t;
  public:
    T( uBaseTask &t ) : t( t ) {}
    void mem() {}
  private:
    void main() {
        _Resume E() _At t;
        _Accept( mem );
    }
};
void uMain::main() {
    T t( uThisTask() );
    try {
        _Enable {
            t.mem();
        }
    } catch( E &e ) {
    }
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:23656) (uSerial &)0x83260 : Rendezvous failure in accepted call from task
uMain (0xffbef008) to mutex member of task T (0x83130). Error occurred while executing task T (0x83130).
```

because the blocked call to t.mem from task uMain is interrupted by the concurrent exception of type E. When the blocked call from uMain is accepted, uMain immediately detects the concurrent exception and does not start the call. As a result, uMain implicitly resumes an exception of type uMutexFailure::RendezvousFailure concurrently at task t so it knows the call did not occur and can take appropriate corrective action (see Section 5.10.3, p. 88).

The following program:

```
_Task T1 {
    uCondition w;
  public:
    void mem() { w.wait(); }
  private:
    void main() { _Accept( mem ); }
};
_Task T2 {
    T1 &t1;
    void main() { t1.mem(); }
  public:
    T2( T1 &t1 ) : t1( t1 ) {}
};
void uMain::main() {
    T1 *t1 = new T1;
    T2 *t2 = new T2( *t1 );
    delete t1;
    delete t2;
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:23856) (uCondition &)0x84410 : Waiting failure as task uMain (0xffbef008)
found blocked task T2 (0x8d470) on condition variable during deletion. Error occurred while executing task
T2 (0x8d470).
```

because the call to t1.mem blocks task t2 on condition queue w and then task t1 implicitly accepts its destructor when its main terminates. As a result, when task uMain attempts to delete task t1, it finds task t2 still blocked on the condition queue.

7.2.3.2 Coroutine

Neither resuming to nor suspending from a terminated coroutines is allowed; a coroutine is terminated when its main routine returns. The following program:

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

generates this error:

```
uC++ Runtime error (UNIX pid:24169) Attempt by coroutine uMain (0xffbef008) to resume terminated
coroutine C (0x823a0). Possible cause is terminated coroutine's main routine has already returned. Error
occurred while executing task uMain (0xffbef008).
```

because the first call to c.mem resumes coroutine c and then coroutine c terminates. As a result, when uMain attempts the second call to c.mem, it finds coroutine c terminated. A similar situation can be constructed using suspend, but is significantly more complex to generate, hence it is not discussed in detail.

Member suspend resumes the last resumer, and therefore, there must be a resume before a suspend can execute (see Section 2.7.3, p. 17). The following program:

```
_Coroutine C {
    void main() {}
  public:
    void mem() {
        suspend();          // suspend before any resume
    }
};
void uMain::main() {
    C c;
    c.mem();
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:24258) Attempt to suspend coroutine C (0x82390) that has never been
resumed. Possible cause is a suspend executed in a member called by a coroutine user rather than by the
coroutine main. Error occurred while executing task uMain (0xffbef008).
```

because the call to C::mem executes a suspend before the coroutine's main member is started, and hence, there is no resumer to reactivate. In general, member suspend is only called within the coroutine main or non-public members called directly or indirectly from the coroutine main, not in public members called by other coroutines.

Two tasks cannot simultaneously execute the same coroutine; only one task can use the coroutine's execution at a time. The following program:

```
_Coroutine C {
    void main() {
        uBaseTask::yield();
    }
  public:
    void mem() {
        resume();
    }
};
```

```
_Task T {
    C &c;
    void main() {
        c.mem();
    }
  public:
    T( C &c ) : c( c ) {}
};
void uMain::main() {
    C c;
    T t1( c ), t2( c );
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:24393) Attempt by task T (0x82ea0) to resume coroutine C (0x831e0) cur-
rently being executed by task T (0x83040). Possible cause is two tasks attempting simultaneous execution
of the same coroutine. Error occurred while executing task T (0x82ea0).
```

because t1's thread first calls routine C::mem and then resumes coroutine c, where it yields the processor. t2's threads now calls routine C::mem and attempts to resume coroutine c but t1 is currently using c's execution-state (stack). This same error occurs if the coroutine is changed to a coroutine monitor and task t1 waits in coroutine c after resuming it:

```
_Cormonitor CM {
    uCondition w;
    void main() {
        w.wait();
    }
  public:
    void mem() {
        resume();
    }
};
_Task T {
    CM &cm;
    void main() {
        cm.mem();
    }
  public:
    T( CM &cm ) : cm( cm ) {}
};
void uMain::main() {
    CM cm;
    T t1( cm ), t2( cm );
}
```

When a coroutine (or task) is created, there must be sufficient memory to allocate its execution state. The following program:

```
unsigned int uMainStackSize() {
    return 1000000000;      // very large stack size for uMain
}
void uMain::main() {
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:24848) Attempt to allocate 1000000000 bytes of storage for coroutine or
task execution-state but insufficient memory available. Error occurred while executing task uBootTask
(0x4d6b0).
```

because the declaration of uMain by the uBootTask fails due to the request for a 1000000000-byte stack for uMain.

As mentioned in Section 2.4, p. 10, the μC++ kernel provides no support for automatic growth of stack space for coroutines and tasks. Several checks are made to mitigate problems resulting from lack of dynamic stack growth. The following program:

```
void uMain::main() {
    char x[uThisCluster().getStackSize()];      // array larger than stack space
    verify();
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:24917) Stack overflow detected: stack pointer 0x7a650 below limit 0x7a820.
Possible cause is allocation of large stack frame(s) and/or deep call stack. Error occurred while executing
task uMain (0xffbef008).
```

because the declaration of the array in uMain uses more than the current stack space.

The following program:

```
void uMain::main() {
    {
        char x[uThisCluster().getStackSize()];      // array larger than stack space
        for ( int i = 0; i < uThisCluster().getStackSize(); i += 1 ) {
            x[i] = 'a';         // write outside stack space
        }
    } // delete array
    verify();
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:24968) Stack corruption detected. Possible cause is corrupted stack frame
via overwriting memory. Error occurred while executing task uMain (0xffbef008).
```

because the declaration of the array in uMain uses more than the current stack space, and by writing into the array, the current stack space is corrupted (and possibly another stack, as well).

7.2.3.3 Mutex Type

It is a restriction that a task must acquire and release mutex objects in nested (LIFO) order (see Section 2.8, p. 17). The following program:

```
_Task T;

_Cormonitor CM {
    T *t;
    void main();
  public:
    void mem( T *t ) {  // task owns mutex object
        CM::t = t;
        resume();           // begin coroutine main
    }
};
```

```
_Task T {
    CM &cm;
    void main() {
        cm.mem( this ); // call coroutine monitor
    }
  public:
    T( CM &cm ) : cm( cm ) {}
    void mem() {
        resume();           // restart task in CM::mem
    }
};
void CM::main() {
    t->mem();               // call back into task
}
void uMain::main() {
    CM cm;
    T t( cm );
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:25043) Attempt to perform a non-nested entry and exit from multiple ac-
cessed mutex objects. Error occurred while executing task T (0x835f0).
```

because t's thread first calls mutex routine CM::mem (and now owns coroutine monitor cm) and then resumes coroutine cm, which now calls the mutex routine T::mem (t already owns itself). The coroutine cm resumes t from within T::mem, which restarts in CM::mem (full coroutining) and exits before completing the nested call to mutex routine T::mem (where cm is suspended). Therefore, the calls to these mutex routines do not terminate in LIFO order. The following program is identical to the previous one, generating the same error, but the coroutine monitor has been separated into a coroutine and monitor:

```
_Monitor M;
_Task T;

_Coroutine C {
    M *m;
    void main();
  public:
    void mem( M *m ) {
        C::m = m;
        resume();         // begin coroutine main
    }
};
_Monitor M {
    C &c;
    T *t;
  public:
    M( C &c ) : c( c ) {}
    void mem1( T *t ) { // task owns mutex object
        M::t = t;
        c.mem( this );
    }
    void mem2();
};
```

```
void C::main() {
    m->mem2();
}
_Task T {
    M &m;
    C &c;
    void main() {
        m.mem1( this ); // call monitor
    }
  public:
    T( M &m, C &c ) : m( m ), c( c ) {}
    void mem() {
        resume();         // restart task in C::mem
    }
};
void M::mem2() {
    t->mem();             // call back into task
}
void uMain::main() {
    C c;
    M m( c );
    T t( m, c );
}
```

Ownership of a mutex object by a task applies through any coroutine executed by the task. The following program:

```
_Task T;

_Coroutine C {
    T *t;
    void main();
  public:
    void mem( T *t ) {
        C::t = t;
        resume();
    }
};
```

```
_Task T {
    C &c;
    void main() {
        c.mem( this );
        yield();
    }
  public:
    T( C &c ) : c( c ) {}
    void mem() {
        resume();
    }
};
```

```
void C::main() {
    t->mem();
}
void uMain::main() {
    C c;
    T t1( c ), t2( c );
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:25216) Attempt by task T (0x83050) to activate coroutine C (0x833c0) currently executing in a mutex object owned by task T (0x83208). Possible cause is task attempting to logically change ownership of a mutex object via a coroutine. Error occurred while executing task T (0x83050).
```

because t1's thread first calls routine C::mem and then resumes coroutine c, which now calls the mutex routine T::mem. t1 restarts in C::mem and returns back to T::main and yields the processor. t2's threads now calls routine C::mem and attempts to resume coroutine c, which would restart t2 via c in T::mem. However, this resumption would result in a logical change in ownership because t2 has not acquired ownership of t1. This same error can occur if the coroutine is changed to a coroutine monitor and task t1 waits in coroutine c after resuming it:

```
_Task T;

_Coroutine C {
    T *t;
    void main();
  public:
    void mem( T *t ) {
        C::t = t;
        resume();
    }
};
```

```
_Task T {
    uCondition w;
    C &c;
    void main() {
        c.mem( this );
        w.wait();
    }
  public:
    T( C &c ) : c( c ) {}
    void mem() {
        resume();
    }
};
```

```
void C::main() {
    t->mem();
}
void uMain::main() {
    C c;
    T t1( c ), t2( c );
}
```

It is incorrect storage management to delete any object if there are outstanding nested calls to the object's members. μC++ detects this case only for mutex objects. The following program:

```
class T;

_Monitor M {
  public:
    void mem( T *t );
};
class T {
    M *m;
  public:
    void mem1() {
        m = new M;          // allocate object
        m->mem( this );     // call into object
    }
    void mem2() {
        delete m;           // delete object with pending call
    }
};
void M::mem( T *t ) {
    t->mem2();              // call back to caller
}
void uMain::main() {
    T t;
    t.mem1();
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:25337) Attempt by task uMain (0xffbef008) to call the destructor for uSerial
0x83278, but this task has outstanding nested calls to this mutex object. Possible cause is deleting a
mutex object with outstanding nested calls to one of its members. Error occurred while executing task
uMain (0xffbef008).
```

It is incorrect to perform more than one delete on a mutex object, which can happen if multiple tasks attempt to perform simultaneous deletes on the same object. μC++ detects this case only for mutex objects. The following program:

```
_Monitor M {
    uCondition w;
  public:
    ~M() {
        w.wait();           // force deleting task to wait
    }
};
```

```
_Task T {
    M *m;
    void main() {
        delete m;            // delete mutex object
    }
  public:
    T( M *m ) : m(m) {}
};
void uMain::main() {
    M *m = new M;            // create mutex object
    T t( m );                // create task
    delete m;                // also delete mutex object
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:25431) Attempt by task T (0x82cd0) to call the destructor for uSerial
0x83a48, but this destructor was already called by task uMain (0xffbef008). Possible cause is multiple
tasks simultaneously deleting a mutex object. Error occurred while executing task T (0x82cd0).
```

7.2.3.4 Task

The destructor of a task cannot execute if the thread of that task has not finished (halted) because the destructor deallocates the environment in which the task's thread is executing. The following program:

```
_Task T {
    uCondition w;
    void main() {
        _Accept( ~T );       // uMain invokes destructor
        w.wait();            // T continues but blocks, which restarts uMain
    }
};
void uMain::main() {
    T t;
} // implicitly invoke T::~T
```

generates this error:

```
uC++ Runtime error (UNIX pid:25719) Attempt to delete task T (0x82900) that is not halted. Possible cause
is task blocked on a condition queue. Error occurred while executing task uMain (0xffbef008).
```

because the call to the destructor restarts the accept statement (see Section 2.9.2.3, p. 24), and the thread of t blocks on condition w, which restarts the destructor. However, the destructor cannot cleanup without invalidating any subsequent execution of task t.

7.2.3.5 Condition Variable

Only the owner of a condition variable can wait and signal on it (see Section 2.9.3.1, p. 25). The following program:

```
_Task T {
    uCondition &w;
    void main() {
        w.wait();
    }
  public:
    T( uCondition &w ) : w( w ) {}
};
void uMain::main() {
    uCondition w;
    T t( w );
    w.wait();
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:6605) Attempt to wait on a condition variable for a mutex object not locked
by this task. Possible cause is accessing the condition variable outside of a mutex member for the mutex
object owning the variable. Error occurred while executing task T (0x826c8).
```

because the condition variable w is passed from uMain to t, and then there is a race to wait on the condition. The error message shows that uMain waited first so it became the condition owner, and then t's attempt to wait fails. Changing wait in T::main to signal generates a similar message with respect to signalling a condition not owned by mutex object t. It is possible for one mutex object to create a condition and pass it to another, as long as the creator does not wait on it before passing it.

The same situation can occur if a wait or signal is incorrectly placed in a nomutex member of a mutex type. The following program:

```
_Task T {
    uCondition w;
    void main() { w.wait(); }
  public:
    _Nomutex void mem() {
        w.signal();
    }
};
void uMain::main() {
    T t;
    yield();
    t.mem();
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:6502) Attempt to signal a condition variable for a mutex object not locked
by this task. Possible cause is accessing the condition variable outside of a mutex member for the mutex
object owning the variable. Error occurred while executing task uMain (0xffbef008).
```

because task t is first to wait on condition variable w due to the yield in uMain::main, and then uMain does not lock mutex-object t when calling mem as it is nomutex. Only if uMain has t locked can it access any condition variable owned by t. Changing signal in T::mem to wait generates a similar message with respect to waiting on a condition not locked by mutex object uMain.

A condition variable must be non-empty before examining data stored with the front task blocked on the queue (see Section 2.9.3.1, p. 25). The following program:

```
void uMain::main() {
    uCondition w;
    int i = w.front();
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:2411) Attempt to access user data on an empty condition. Possible cause
is not checking if the condition is empty before reading stored data. Error occurred while executing task
uMain (0xffbef870).
```

because the condition variable w is empty so there is no data to return.

7.2.3.6 Accept Statement

An _Accept accept statement can only appear in a mutex member. The following program:

```
_Monitor M {
  public:
    void mem1() {}
    _Nomutex void mem2() {
        _Accept( mem1 );   // not allowed in nomutex member
    }
};
void uMain::main() {
    M m;
    m.mem2();
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:2159) Attempt to accept in a mutex object not locked by this task. Possible cause is accepting in a nomutex member routine. Error occurred while executing task uMain (0xffbef008).
```

7.2.3.7 Calendar

When creating an absolute time value using uTime (see Section 10.2.1, p. 130), the value must be in the range 00:00:00 UTC, January 1, 1970 to 03:14:07 UTC, January 19, 2038, which is the UNIX start and end epochs. The following program:

```
void uMain::main() {
    uTime t( -17 );
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:2243) Attempt to create uTime( year=1970, month=0, day=0, hour=0, min=0, sec=-17, nsec=0 ), which exceeds range 00:00:00 UTC, January 1, 1970 to 03:14:07 UTC, January 19, 2038. Error occurred while executing task uMain (0xffbef008).
```

7.2.3.8 Locks

The argument for the uLock constructor (see Section 2.16.2, p. 37) must be 0 or 1. The following program:

```
void uMain::main() {
    uLock l(3);
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:2328) Attempt to initialize uLock 0x91030 to 3 that exceeds range 0-1. Error occurred while executing task uMain (0xffbef008).
```

because the value 3 passed to the constructor of uLock is outside the range 0–1.

7.2.3.9 Cluster

A cluster cannot be deleted with a task still on it, regardless of what state the task is in (i.e., blocked, ready or running). The following program:

```
_Task T {
    void main() {}
};
void uMain::main() {
    T *t = new T;
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:2404) Attempt to delete cluster userCluster (0x82260) with task T (0x92770) still on it. Possible cause is the task has not been deleted. Error occurred while executing task uBootTask (0x5d6f0).
```

because the uBootTask happens to delete the user cluster (see Section 8.3, p. 115) after uMain::main terminates before the dynamically allocated task t has terminated. Deleting the task associated with t *before* uMain::main terminates solves the problem.

Similarly, a cluster cannot be deleted with a processor still located on it, regardless of what state the processor is in (i.e., running or idle). The following program:

```
void uMain::main() {
    uProcessor &p = *new uProcessor( uThisCluster() );
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:2488) Attempt to delete cluster userCluster (0x81770) with processor 0x91c80 still on it. Possible cause is the processor has not been deleted. Error occurred while executing task uBootTask (0x5cc00).
```

because the uBootTask deletes the user cluster (see Section 8.3, p. 115) after uMain::main terminates but the dynamically allocated processor p is still on the user cluster. Deleting the processor associated with p *before* uMain::main terminates solves the problem.

7.2.3.10 Heap

μC++ provides its own concurrent dynamic memory allocation routines. Unlike most C/C++ dynamic memory allocation routines, μC++ does extra checking to ensure that some aspects of dynamic memory usage are done correctly. The following program:

```
void uMain::main() {
    int *ip = (int *)1;           // invalid pointer address
    delete ip;
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:2535) Attempt to free storage 0x1 outside the current heap range:0x5e468
to 0x91b78. Possible cause is invalid pointer. Error occurred while executing task uMain (0xffbef008).
```

because the value of pointer ip is not within the heap storage area, and therefore, cannot be deleted.

The following program:

```
void uMain::main() {
    int *ip = new int[10];
    delete &ip[5];                // not the start of the array
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:2607) Attempt to free storage 0x91c14 with corrupted header. Possible
cause is duplicate free on same block or overwriting of header information. Error occurred while executing
task uMain (0xffbef008).
```

because the pointer passed to **delete** must always be the same as the pointer returned from **new**. In this case, the value passed to **delete** is in the middle of the array instead of the start.

The following program:

```
_Task T {
    void main() {}
  public:
    void mem() {}
};
void uMain::main() {
    T *t = new T;
    delete t;
    t->mem();                     // use deleted storage
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:2670) (uSpinLock &)0x92a50.acquire() : internal error, attempt to multiply
acquire spin lock by same task. Error occurred while executing task uMain (0xffbef008).
```

because an attempt is made to use the storage for task t after it is deleted, which is always incorrect. This storage may have been reallocated to another task and now contains completely different information. The problem is detected inside of the μC++ kernel, where there are assertion checks for invalid pre- or post-conditions. In this case, the invalid storage happened to trigger a check for a task acquiring a spin lock twice, which is never suppose to happen. Using storage incorrectly can trigger other "internal errors" from the μC++ kernel.

As well, a warning message is issued at the end of a program if all storage is not freed.

```
uC++ Runtime warning (UNIX pid:3914) : program terminating with 32(0x20) bytes of storage allocated
but not freed. Possible cause is unfreed storage allocated by the program or system/library routines called
from the program.
```

This is not an error; it is a warning. While this message indicates unfreed storage, it does not imply the storage is allocated by the user's code. Many system (e.g., exceptions) and library (e.g., string type and socket I/O) operations allocate storage (such as buffers) for the duration of the program, and therefore, there is little reason to free the storage at program termination. (Why cleanup and then terminate?) There is nothing that can be done about this unfreed storage. Therefore, the value printed is only a guide in determining if all of a user's storage is freed.

What use is this message? Any sudden increase of unfreed storage from some base value may be a strong indication of unfreed storage in the user's program. A quick check of the dynamic allocation can be performed to verify all user storage is being freed.

7.2.3.11 I/O

There are many different I/O errors; only those related to the μC++ kernel are discussed. The following program:

```
void uMain::main() {
    uThisCluster().select( -1, 0, NULL );
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:2962) Attempt to select on file descriptor -1 that exceeds range 0-1023.
Error occurred while executing task uMain (0xffbef008).
```

The following program:

```
void uMain::main() {
    uThisCluster().select( -1, NULL, NULL, NULL, NULL );
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:3008) Attempt to select with a file descriptor set size of -1 that exceeds
range 0-1024. Error occurred while executing task uMain (0xffbef008).
```

7.2.3.12 Processor

The following program:

```
#include <uSemaphore.h>
void uMain::main() {
    uSemaphore s(0);
    s.P();                    // block only thread => synchronization deadlock
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:3110) No ready or pending tasks. Possible cause is tasks are in a synchro-
nization or mutual exclusion deadlock. Error occurred while executing task uProcessorTask (0x82740).
```

because the only thread blocks so there are no other tasks to execute, resulting in a synchronization deadlock. This message also appears for the more complex form of deadlock resulting from mutual exclusion.

7.2.3.13 UNIX

There are many UNIX related errors, of which only a small subset are handled specially by μC++.

A common error in C++ programs is to generate and use an invalid pointer. This situation can arise because of an incorrect pointer calculation, such as an invalid subscript. The following program:

```
void uMain::main() {
    int *ip = NULL;           // set address to 0
    *ip += 1;                 // use the bad address
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:3241) Attempt to address location 0x0. Possible cause is reading out-
side the address space or writing to a protected area within the address space with an invalid pointer or
subscript. Error occurred while executing task uMain (0xffbef008).
```

because the value of pointer ip is probably within the executable code, which is read-only, but an attempt to write is occurring.

If a μC++ program is looping for some reason, it may be necessary to terminate its execution. Termination is accomplished using a shell kill command, sending signal SIGTERM to the UNIX process. μC++ receives the termination signal and attempts to shutdown the application, which is important in multikernel mode with multiple processors. The following program:

```
#include <unistd.h>                    // getpid prototype
void uMain::main() {
    kill( getpid(), SIGTERM );         // send SIGTERM signal to program
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:3315) Application interrupted by a termination signal. Error occurred while
executing task uMain (0xffbef008).
```

because the μC++ program sent itself a termination (SIGTERM) signal.

Chapter 8

μC++ Kernel

The μC++ kernel is a library of classes and routines that provide low-level lightweight concurrency support on uniprocessor and multiprocessor computers running the UNIX operating system. On uniprocessors, parallelism is simulated by rapid context switching at non-deterministic points so a programmer cannot rely on order or speed of execution. Some of the following facilities only have an effect on multiprocessor computers but can be called on a uniprocessor so that a program can be seamlessly transported between the two architectures.

The μC++ kernel does not call the UNIX kernel to perform a context switch or to schedule tasks, and uses shared memory for communication. As a result, performance for execution of and communication among large numbers of tasks is significantly increased over UNIX processes. The maximum number of tasks that can exist is restricted only by the amount of memory available in a program. The minimum stack size for an execution state is machine dependent, but can be as small as 256 bytes. The storage management of all μC++ objects and the scheduling of tasks on virtual processors is performed by the μC++ kernel.

8.1 Pre-emptive Scheduling and Critical Sections

Care must be taken when writing threaded programs calling certain UNIX library routines that are *not* thread-safe. For example, the UNIX random number generator rand maintains an internal state between successive calls and there is no mutual exclusion on this internal state. Hence, one task executing the random number generator can be pre-empted and the generator state can be modified by another task, which may result in problems with the generated random values or errors. Therefore, when writing μC++ programs, always use the thread-safe versions of UNIX library routines, such as rand_r to generate random numbers.

For some non-thread-safe UNIX library-routines, μC++ provides a thread-safe equivalent, such as abort/uAbort, exit (see Section 7.2.2, p. 99), sleep, usleep, and the μC++ I/O library (see Chapter 4, p. 59).

8.2 Memory Management

In μC++, all user data is located in memory that is accessible by all kernel threads started by μC++. In order to make memory management operations safe, the C++ memory management operators **new** and **delete** are indirectly redefined through the C routines malloc and free to allocate and free memory correctly by multiple tasks. These memory management operations provide identical functionality to the C++ and C equivalent ones.

8.3 Cluster

As mentioned in Section 2.3.1, p. 8, 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 );
    uCluster( uBaseSchedule<uBaseTaskDL> &ReadyQueue,
          unsigned int stackSize = uDefaultStackSize(), const char *name = "*unnamed*" );
    uCluster( uBaseSchedule<uBaseTaskDL> &ReadyQueue, const char *name = "*unnamed*" );

    const char *setName( const char *name );
    const char *getName() const;
    unsigned int setStackSize( unsigned int stackSize );
    unsigned int getStackSize() const;

    enum { ReadSelect = 1, WriteSelect = 2,  ExceptSelect = 4 };

    int select( int fd, int rwe, timeval *timeout = NULL );
    int select( int nfds, fd_set *rfd, fd_set *wfd, fd_set *efd, timeval *timeout = NULL );

    const uBaseTaskSeq &getTasksOnCluster();
    unsigned int getProcessors() const;
    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 (see Section 11.1, p. 141 for the first default value).

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 overloaded member routine select works like the UNIX select routine, but on a per-task basis per cluster. That is, all I/O performed on a cluster is managed by a poller task for that cluster (see Section 4.1, p. 59). In general, select is used only in esoteric situations, e.g., when μC++ file objects are mixed with standard UNIX file objects on the same cluster. These members return the total number of file descriptors set in all file descriptor masks, and each routine has the following form:

select(**int** fd, **int** rwe, timeval *timeout = NULL) – this form is a shorthand select for a single file descriptor. The mask, rwe, is composed of logically "or"ing flags ReadSelect, WriteSelect, and ExceptSelect. The timeout value points to a maximum delay value, specified as a timeval, to wait for the I/O to complete. If the timeout pointer is null, the select blocks until the I/O operation completes or fails. This form is more efficient than the next forms with complete file descriptor sets, but handles only a single file.

select(**int** nfds, fd_set *rfd, fd_set *wfd, fd_set *efd, timeval *timeout = NULL) – this form examines the first nfds I/O file descriptors in the sets pointed to by rfd, wfd, and efd, respectively, to see if any of their file descriptors are ready for reading, or writing, or have exceptional conditions pending. The timeout value points to a maximum delay value, specified as a timeval, to wait for an I/O to complete. If the timeout pointer is null, the select blocks until one of the I/O operations completes or fails.

There does not seem to be any standard semantics action when multiple kernel threads access the same file descriptor in select. Some systems wake all kernel threads waiting for the same file descriptor; others wake the kernel threads in FIFO order of their request for the common file descriptor. μC++ adopts the former semantic and wakes all tasks waiting for the same file descriptor. In general, this is not a problem because *all* μC++ file routines retry their I/O operation, and only one succeeds in obtaining data (which one is non-deterministic).

Finally, it is impossible to precisely deliver select errors to the task that caused it. For example, if one task in waiting for I/O on a file descriptor and another task closes the file descriptor, the UNIX select fails but with no information about which file descriptor caused the error. Therefore, μC++ wakes up all tasks waiting on the select at the time of the error and the tasks must retry their I/O operation. Again, *all* μC++ file routines retry their I/O operations after waiting on select.

> □ Unfortunately, UNIX does not provide adequate facilities to ensure that signals sent to wake up a blocked UNIX process or kernel thread is always delivered. There is a window between sending a signal and blocking using a UNIX select operation that cannot be closed. Therefore, the poller task has to wake up once a second to deal with the rare event that a signal sent to wake it up is missed. This problem only occurs when a task is migrating from one cluster to another cluster on which I/O is being performed. □

The member routine getTasksOnCluster returns a list of all the tasks currently on the cluster. The member routine getProcessors returns the number of processors currently on the cluster. The member routine getProcessorsOnCluster returns a list of all the processors currently on the cluster. These routines are useful for profiling and debugging programs.

The free routine:

```
uCluster &uThisCluster();
```

is used to determine the identity of the current cluster a task resides on.

8.4 Processors

As mentioned in Section 2.3.2, p. 9, a μC++ virtual processor is a "software processor"; it provides a runtime environment for parallel thread execution. This environment contains variables to affect how thread execution is performed on a processor. Environment variables are used implicitly, unless overridden, when executing threads on a processor:

pre-emption time is the default time, in milliseconds, to the next implicit yield of the currently executing task to simulate non-deterministic execution (see Section 8.4.1, p. 119).

spin amount is the default number times the cluster's ready queue is checked for an available task to execute before the processor blocks (see Section 8.4.2, p. 119).

processors is the default number of processors created implicitly on a cluster.

The variables are either explicitly set or implicitly assigned a μC++ default value when the processor is created.

In μC++, a virtual processor is implemented as a kernel thread (possibly via 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 execute in parallel. The maximum number of virtual processors that can be created is indirectly limited by the number of kernel/processes the operating system allows a program to create, as the sum of the virtual processors on all clusters cannot exceed this limit.

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 11.1, p. 141 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 11.1, p. 141 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 10.3, p. 132).

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 8.4.1) 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 8.4.2) 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 8.4.3) 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 *µ*C++ 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. □

8.4.1 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 non-deterministic 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. □

8.4.2 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 *µ*C++ 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.

8.4.3 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 *µ*C++ kernel on a task as a result of certain *µ*C++ 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 *µ*C++. 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.

> □ *µ*C++ tasks are not implemented with kernel threads or operating system processes for two reasons. First, kernel threads have a high runtime cost for creation and context switching. Second, an operating system process is normally allocated as a separate address space (or perhaps several) and if the system does not allow memory sharing among address spaces, tasks have to communicate using pipes and sockets. Pipes and sockets are runtime expensive. If shared memory is available, there is still the overhead of entering the operating system, page table creation, and management of the address space of each process. Therefore, kernel threads and processes are called **heavyweight** because of the high runtime cost and space overhead in creating a separate address space for a process, and the possible restrictions on the forms of communication among them. *µ*C++ provides access to kernel threads only indirectly through virtual processors (see Section 2.3.2, p. 9). A user is not prohibited from creating kernel threads or processes explicitly, but such threads are not administrated by the *µ*C++ runtime environment. □

Chapter 9

POSIX Threads (pthreads)

Posix threads (pthreads) is a relatively low-level C-language thread-library providing two basic concurrency mechanisms: threads and locks. As pthreads is designed for C rather than C++, pthreads does not take advantage of any high-level features of C++. A thread is started (forked) in a routine, possibly passing a single type-unsafe argument, and another thread can wait for this thread's termination (join), possibly returning a single type-unsafe value. Two kinds of locks are available: for synchronization, pthread_cond, which is like μC++'s uCondLock (see Section 2.16.4, p. 38), and for mutual exclusion, pthread_mutex, which is like μC++'s uOwnerLock (see Section 2.16.3, p. 38). See a pthreads reference-manual [But97] for complete details on the syntax and semantics of using this library to construct concurrent programs.

9.1 Combining Pthreads with μC++

Pthreads is the threading standard for POSIX-compliant systems and has become the de-facto standard for concurrent programming in C; hence, there exists a significant number of applications and libraries built using pthreads. To help concurrent programmers leverage this existing resource, while still writing high-level concurrent programs in C++, μC++ provides a subset of the most commonly used pthreads routines (see Figure 9.1). Thus, a μC++ programmer can use high-level concurrent features in μC++, along with existing C/C++ source code or pre-compiled code that uses pthreads functionality. It is also possible to take a native pthreads programs and – with only minor changes – convert it to a μC++ program using μC++'s pthreads support. In this way, programmers can incrementally transform and extend pthreads applications to use higher-level C++ approaches, starting with a combination of low-level pthreads threads/locks and high-level μC++ capabilities, and ultimately moving to all high-level mechanisms. There are three basic ways in which μC++ can be used in conjunction with pthreads:

- linking against pre-compiled binaries using pthreads,
- recompilation of separate translation units using pthreads,
- recompilation of entire pthreads programs.

9.1.1 Linking μC++ and Pthreads Binaries

In most cases, existing object files using pthreads routines can be directly linked with a μC++ program. For example, a μC++ program, program.cc, is compiled and linked with a pthreads-dependent object-file, plib.o, by:

```
    u++ [C++ options] program.cc plib.o
```

Note, the standard pthreads library, libpthread, must *not* be linked with this program (i.e., -lpthreads) as only the pthreads routines provided by μC++ work. However, this approach fails if plib.o uses routines not provided by μC++ (see Figure 9.1), the object file defines the entry point into the program (i.e., routine main), or if plib.o calls pthreads routines inside global constructors or destructors. The second and third cases can be corrected if the source for plib.o can be changed and recompiled (described next).

9.1.2 Recompiling Pthreads Modules

If a translation unit declares global objects with constructors and destructors calling pthreads routines (i.e., *before/after* the initial starting routine), the translation unit must be recompiled with the u++ command (see Section 2.4.1, p. 10).

```
//############################ Creation ############################
int    pthread_create( pthread_t *new_thread_id, const pthread_attr_t *attr, void * (*start_func)( void * ), void *arg );
int    pthread_attr_init( pthread_attr_t *attr );
int    pthread_attr_destroy( pthread_attr_t *attr );
int    pthread_attr_setscope( pthread_attr_t *attr, int contentionscope );
int    pthread_attr_getscope( const pthread_attr_t *attr, int *contentionscope );
int    pthread_attr_setdetachstate( pthread_attr_t *attr, int detachstate );
int    pthread_attr_getdetachstate( const pthread_attr_t *attr, int *detachstate );
int    pthread_attr_setstacksize( pthread_attr_t *attr, size_t stacksize );
int    pthread_attr_getstacksize( const pthread_attr_t *attr, size_t *stacksize );
int    pthread_getattr_np( pthread_t threadID, pthread_attr_t *attr ); // GNU extension
int    pthread_yield( void ); // GNU extension
//############################ Exit ############################
void pthread_exit( void *status );
int    pthread_join( pthread_t threadID, void **status );
int    pthread_tryjoin_np( pthread_t threadID, void **status ); // GNU extension
int    pthread_timedjoin_np( pthread_t threadID, void **status, const struct timespec *abstime ); // GNU extension
int    pthread_detach( pthread_t threadID );
//############################ Parallelism ############################
int    pthread_getconcurrency( void );      // XOPEN extension
int    pthread_setconcurrency( int new_level ); // XOPEN extension
//############################ Thread Specific Data ############################
void pthread_deletespecific_( void *pthreadData ); // see uMachContext::invokeTask
int    pthread_key_create( pthread_key_t *key, void (*destructor)( void * ) );
int    pthread_key_delete( pthread_key_t key );
int    pthread_setspecific( pthread_key_t key, const void *value );
void *pthread_getspecific( pthread_key_t key );
//############################ ID ############################
pthread_t pthread_self( void );
int    pthread_once( pthread_once_t *once_control, void (*init_routine)( void ) );
//############################ Cancellation ############################
int    pthread_cancel( pthread_t threadID );
int    pthread_setcancelstate( int state, int *oldstate );
int    pthread_setcanceltype( int type, int *oldtype );
void pthread_testcancel( void );
void pthread_cleanup_pop( int ex );
void pthread_cleanup_push( void (*routine) (void *), void *args );
//############################ Mutex ############################
int    pthread_mutex_init( pthread_mutex_t *mutex, const pthread_mutexattr_t *attr );
int    pthread_mutex_destroy( pthread_mutex_t *mutex );
int    pthread_mutex_lock( pthread_mutex_t *mutex );
int    pthread_mutex_trylock( pthread_mutex_t *mutex );
int    pthread_mutex_unlock( pthread_mutex_t *mutex );
//############################ Condition ############################
int    pthread_cond_init( pthread_cond_t *cond, const pthread_condattr_t *attr );
int    pthread_cond_destroy( pthread_cond_t *cond );
int    pthread_cond_wait( pthread_cond_t *cond, pthread_mutex_t *mutex );
int    pthread_cond_timedwait( pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime );
int    pthread_cond_signal( pthread_cond_t *cond );
int    pthread_cond_broadcast( pthread_cond_t *cond );
```

Figure 9.1: Implemented Pthreads Routines

As well, the pthreads module containing the initial entry point main must be converted to the μC++ member uMain::main (see Section 2.2, p. 8). This change can be made conditionally using **#ifdef**s (see Section 2.4.2, p. 11) to allow a pthreads program to conditionally switch between running native pthreads or simulated pthreads in μC++ depending on whether the program is compiled with g++ or u++, e.g.:

```
#ifdef __U_CPLUSPLUS__
void uMain::main() {      // only change starting routine if compiled with u++
#else
int main( int argc, char *argv[] ) {
#endif
    // body of pthreads main routine
}
```

With these changes, the entire program can be compiled with g++, or another compatible C++ compiler, to use native pthreads and linked with the pthreads (-lpthread) library. Alternatively, the program can be complied with u++, using the pthreads simulation provided by μC++.

9.2 μC++ Task as a Pthreads Thread

In order to ease the co-operation with pthreads code, and possibly help transition from low-level pthreads concurrency to high-level μC++ concurrency, a μC++ task can become a quasi-pthreads thread by inheriting from uPthreadable:

```
_Task uPthreadable {                              // abstract class (inheritance only)
  protected:
    void *joinval;                                // pthreads return value
    pthread_attr_t pthread_attr;                  // pthread attributes

    uPthreadable( const pthread_attr_t *attr_ );
    uPthreadable( uCluster &cluster, const pthread_attr_t *attr_ );
    uPthreadable();                               // same constructors as for uBaseTask
    uPthreadable( unsigned int stackSize );
    uPthreadable( uCluster &cluster );
    uPthreadable( uCluster &cluster, unsigned int stackSize );
  public:
    _Nomutex pthread_t pthreadId();               // returns a pthread id for a uC++ task

    _Event Failure;                               // exceptions
    _Event CreationFailure;
};

_Task T : public uPthreadable {                   // inherit so uC++ task can mimic pthreads task
    . . .
  public:
    T(. . .) : uPthreadable(. . .) {}             // initialize uPthreadable as for uBaseTask
    . . .
};
```

It is best to think of a uPthreadable task as a μC++ task that can mimic a pthreads thread by providing some pthreads properties and capabilities. (Note, type uPthreadable is an abstract class for inheritance only; it cannot be instantiated directly.) The duality of a uPthreadable task allows it to use all the high-level features of μC++ concurrency and yet interact with existing pthreads code, which is helpful in situations where pthreads and μC++ are mixed, and provides a path to transition from pthreads to μC++ concurrency.

A derived class of type uPthreadable has direct access to variables joinval and pthread_attr. Variable joinval must be assigned by the derived class to return a value from pthread_join. Variable pthread_attr is the task's pthreads attributes, which can be read and written by appropriate pthreads attribute routines.

The overloaded constructor routine uPthreadable has the following forms:

uPthreadable(**const** pthread_attr_t *attr_) – creates a task on the current cluster with the specified pthreads attributes. Currently, only the stack-size attribute is observed by the uPthreadable task. The other values are stored, but are otherwise ignored by the uPthreadable task.

uPthreadable(uCluster &cluster, **const** pthread_attr_t *attr_) – creates a task on the specified cluster with the specified pthreads attributes.

uPthreadable(...) are the same as for uBaseTask (see Section 2.13.2, p. 31).

An exception of type uPthreadable::CreationFailure is thrown during task instantiation if a pthread identifier cannot be created.

The member routine pthreadId returns a unique pthreads identifier for the task. This pthreads identifier, which is also returned when a uPthreadable task calls pthread_self, can be passed to any pthreads routine taking a pthread_t type, including pthread_join and pthread_cancel. As a result, pthreads threads can join with or cancel uPthreadable tasks with correct pthreads cleanup functionality. Note, (de)registering a cleanup handler using pthread_cleanup_push/-pop can be performed by any kind of thread but only when executing on a pthreads or uPthreadable task's stack; the cleanup handler is associated with the stack frame on the task's stack where the (de)registration occurs.

It is important to note that a uPthreadable task follows μC++ semantics rather than pthreads semantics and is not considered a pthreads thread, which is defined as a task created by pthread_create. In particular, the life time of a uPthreadable task is the same as an ordinary μC++ task, and it becomes a monitor after its main routine ends. The task uMain is a uPthreadable task.

9.3 Semantic Inconsistencies between Pthreads and μC++

The combination of pthreads and μC++ creates some conflicts, which are resolved in the following ways:

9.3.1 Termination of main

When the starting main routine ends, pthreads semantics cause immediate termination of the program and any outstanding threads, whereas μC++ semantics require that all threads be programmatically terminated. This inconsistency is resolved dynamically in favour of pthreads if any pthreads threads are running when uMain::main ends. In this case, the application is shut-down causing all threads (pthreads and μC++) to be terminated abruptly, i.e., no finalization code is executed for any of these threads. Recall that uPthreadable tasks are not considered pthreads threads.

9.3.2 Cleanup Semantics

The ability to perform cleanup is important in writing robust concurrent programs. Pthreads generalized the notion of cleanup to thread termination (exit or cancellation) by allowing cleanup routines to be registered/de-registered using pthread_cleanup_push and pthread_cleanup_pop. An equivalent capability is provided in C++ using class destructors and **catch** clauses. When a μC++ program has a combination of both C++ and pthreads cleanup mechanisms, both are invoked during thread termination.

Cleanup routines are executed in the reverse order of registration (LIFO). However, within a routine, the ordering between C++ and pthreads cleanups is undefined, e.g., if an object is declared before a call to pthread_cleanup_push in the same routine, its destructor may be executed before the pthreads cleanup. Note, among themselves, C++ and pthreads cleanups always execute in proper order.

As part of a pthreads thread termination, all C++'s catch-any handlers (**catch**(...)) are also executed, which is the same behaviour as for μC++ cancellation (see Section 6, p. 91). When all C++ and pthreads cleanups have been executed, the thread terminates. The reason for executing catch-any handlers is the same as for μC++ cancellation, which is to support the common C++ idiom of using catch-any handlers to perform cleanup actions. Like μC++ cancellation, if a catch-any handler finishes, control does not resume after the try-block it guards, but instead, termination continues with its associated stack unwinding. This behaviour differs from some pthreads implementations that mix C++ and pthreads cleanup. In these implementations, the program is aborted if a catch-any handler finishes during pthreads cleanup. The rationale for aborting is that the normal semantics of resuming execution after the handler's try-block is incompatible with the termination semantics for a thread. However, such a design is inconsistent with μC++ cancellation (see Section 6, p. 91 for the rationale). As well, if the catch-any handler throws a *new* exception during cancellation, the program's behaviour is undefined, as for μC++ cancellation.

9.4 Commentary

The pthreads simulation in μC++ also provides implicit compatibility and safety for programs calling POSIX-compliant library routines in UNIX:

> All functions defined by this volume of IEEE Std 1003.1-2001 shall be thread-safe, except that the following functions need not be thread-safe.

> *... small subset of POSIX functions ...*
> Implementations shall provide internal synchronization *(mutual exclusion)* as necessary in order to satisfy this requirement. [POS08, pp. 507–508]

The most common mechanism to provide mutual exclusion within such library routines is to use pthreads locks. As well, some POSIX compliant routines rely on thread-specific data provided by pthreads. Once pthreads calls are embedded into standard UNIX implementations, it is difficult to use other thread designs due to the problems of interaction between thread libraries. For example, a conflict occurs if a language/library concurrency system (e.g., *μ*C++) does not use pthreads for its underlying concurrency, i.e., the language/library implements the whole concurrency system directly using atomic instructions and kernel threads. The reason a language/library may build its own concurrency runtime is to achieve specialized behaviour that is different from pthreads (e.g., unblocking order, task scheduling, priorities, thread model, etc.). However, when a language/library thread calls a POSIX routine, the routine may call a pthreads routine for thread safety resulting in two different concurrency systems' attempting to manage the same thread. For example, a pthreads lock in a POSIX routine may attempt to block the executing thread, but if the thread is created and managed by a different concurrency system, this operation is logically inconsistent and is likely to fail. The *μ*C++ pthreads simulation handles this problem by interposing its pthreads routines so they are called from within the POSIX-compliant library routines. The simulation routines correctly interact with the *μ*C++ runtime system, while still providing thread-safe access to POSIX library routines. As a result, a *μ*C++ program is portable among POSIX-compliant systems and provides access to most legacy pthreads code.

Chapter 10

Real-Time

Real-time programming is defined by the correctness of a program depending on both the accuracy of the result *and* when the result is produced. The latter criterion is not present in normal programming. Without programming language facilities to specify timing constraints, real-time programs are usually built in ad-hoc ways (e.g., cyclic executive), and the likelihood of encountering timing errors increases through manual calculations. The introduction of real-time constructs is a necessity for accurately expressing time behaviour, as well as providing a means for the runtime system to evaluate whether any timing constraints have been broken. Furthermore, explicit time-constraint constructs can drastically minimize coding complexity as well as analysis. Various programming language constructs for real-time environments are discussed in [SD92, Mar78, LN88, KS86, KK91, ITM90, HM92, GR91, CD95, Rip90].

10.1 Duration and Time

The convenient manipulation of **time** is an essential characteristic in any time-constrained environment. Manipulating time, in turn, yields another metric that expresses a span or **duration** of time. uDuration is a class whose instances represent a span of time, e.g., subtracting two time values results in a difference that is a time duration (2:00 − 1:30 = 30 minute duration). The creation and manipulation of uDuration values are performed through the member routines of class uDuration (see Figure 10.1).

The overloaded constructor routines uDuration provide a choice of specifying a duration value. The parameters have the following meanings:

sec – a number of seconds.

nsec – a number of seconds and nanoseconds.

timeval / timespec – a UNIX timeval or timespec, which is converted to a uDuration value.

A UNIX timeval and timespec value can be used to initialize or assign to a uDuration value, and a uDuration value can be cast into a timeval or timespec value, e.g.:

```
timeval d1 = { 1, 0 };          // (seconds / microseconds)
uDuration d2 = d1, d3;          // convert from timeval to uDuration (initialization)
timespec d4 = d2;               // convert from uDuration to timespec (seconds / nanoseconds) (implicit cast)
d1 = (timeval)d2;               // convert from uDuration to timeval (cast)
d3 = d1;                        // convert from timeval to uDuration (assignment)
d1 = d3;                        // convert from uDuration to timeval (implicit cast)
```

Conversion is guaranteed to be exact. The member routine nanoseconds returns the duration value as a 64 bit number in nanoseconds.

Arithmetic operations may be performed on uDuration values, e.g.:

```
class uDuration {
  public:
    uDuration();
    uDuration( long int sec );
    uDuration( long int sec, long int nsec );
    uDuration( const timeval &t );
    uDuration( const timespec &t );

    uDuration &operator=( const timeval &t ) {
    uDuration &operator=( const timespec &t ) {
    operator timeval() const;
    operator timespec() const;
    long long int nanoseconds() const;

    uDuration &operator-=( uDuration op );
    uDuration &operator+=( uDuration op );
    uDuration &operator*=( long long int op );
    uDuration &operator/=( long long int op );
}; // uDuration

uDuration operator-( uDuration op );
uDuration operator-( uDuration op1, uDuration op2 );
uDuration operator+( uDuration op );
uDuration operator+( uDuration op1, uDuration op2 );
uDuration operator*( uDuration op1, long long int op2 );
uDuration operator*( long long int op1, uDuration op2 );
uDuration operator/( uDuration op1, long long int op2 );
long long int operator/( uDuration op1, uDuration op2 );
bool operator>( uDuration op1, uDuration op2 );
bool operator<( uDuration op1, uDuration op2 );
bool operator>=( uDuration op1, uDuration op2 );
bool operator<=( uDuration op1, uDuration op2 );
bool operator==( uDuration op1, uDuration op2 );
bool operator!=( uDuration op1, uDuration op2 );
ostream &operator<<( ostream &os, const uDuration op );
```

Figure 10.1: Duration Class

```
uDuration x, y, z;
int n;
x += 1;                 // implicitly create a uDuration of length 1 second
x = y + z;              // add two uDurations producing a uDuration
x = y - z;              // subtract two uDurations producing a uDuration
x = y * n;              // multiply a uDuration n times
x = n * y;              // multiply a uDuration n times
x = y / n;              // divide a uDuration by n
```

In addition, relational comparison operators are defined for uDuration objects.

uTime is a class, whose instance represents an absolute time. Time can be specified using some combination of year, month, day, hour, minute, second, and nanosecond in UTC. It is important to note that a time value must be in the range 00:00:00 UTC, January 1, 1970 to 03:14:07 UTC, January 19, 2038, which is the UNIX start and end epochs. The creation and manipulation of uTime values are performed through the member routines of class uTime (see Figure 10.2).

The overloaded constructor routines uTime provide a choice of specifying a time value. The parameters have the following meanings:

year – a year greater than or equal to 1970 and less than or equal to 2038.

month – a number between 0 and 11 inclusive, where 0 represents January and 11 represents December. The default value for a constructor without this argument is 0.

day – a number between 0 and 30 inclusive, where 0 represents the first day of the month and 30 the last day. The default value for a constructor without this argument is 0.

```
class uTime {
  public:
    uTime();
    uTime( long int sec );
    uTime( long int sec, long int nsec );
    uTime( int min, int sec, long int nsec );
    uTime( int hour, int min, int sec, long int nsec );
    uTime( int day, int hour, int min, int sec, long int nsec );
    uTime( int month, int day, int hour, int min, int sec, long int nsec );
    uTime( int year, int month, int day, int hour, int min, int sec, long int nsec );
    uTime( const timeval &t );
    uTime( const timespec &t );

    uTime &operator=( const timeval &t );
    uTime &operator=( const timespec &t );
    operator timeval() const;
    operator timespec() const;
    long long int nanoseconds() const;

    uTime &operator-=( uDuration op );
    uTime &operator+=( uDuration op );
}; // uTime

uDuration operator-( uTime op1, uTime op2 );
uTime operator-( uTime op1, uDuration op2 );
uTime operator+( uTime op1, uDuration op2 );
uTime operator+( uDuration op1, uTime op2 );
bool operator>( uTime op1, uTime op2 );
bool operator<( uTime op1, uTime op2 );
bool operator>=( uTime op1, uTime op2 );
bool operator<=( uTime op1, uTime op2 );
bool operator==( uTime op1, uTime op2 );
bool operator!=( uTime op1, uTime op2 );
ostream &operator<<( ostream &os, const uTime op );
```

Figure 10.2: Time Class

hour – a number between 0 and 23 inclusive, where 0 represents 12:00am and 23 represents 11:00pm. The default value for a constructor without this argument is 0.

min – a number between 0 and 59 inclusive, where 0 is the first minute of the hour and 59 the last. The default value for a constructor without this argument is 0.

sec – a number between 0 and 59 inclusive, where 0 is the first second of the minute and 59 the last.

nsec – a number between 0 and 999999999 inclusive, where 0 is the first nanosecond of the second and 999999999 the last.

timeval / timespec – a UNIX timeval or timespec, which is converted to a uTime value.

It is permissible to *exceed* the logical ranges for the time components; any excess is accumulative, e.g., the following declarations are valid:

```
uTime t1( 0,48,0,60,1000000000 );   // 1970 Jan 3 0:01:01:000000000 (GMT)
uTime t2( 818227413, 0 );           // 1995 Dec 6 05:23:33:000000000 (GMT)
```

A UNIX timeval and timespec value can be used to initialize or assign to a uTime value, and a uTime value may be cast into a timeval or timespec value, e.g.:

```
timeval t1;
gettimeofday( &t1, NULL );   // current time (seconds / microseconds)
uTime t2 = t1, t3;           // convert from timeval to uTime (initialization)
timespec t4 = t2;            // convert from uTime to timespec (seconds / nanoseconds) (implicit cast)
t3 = t1;                     // convert from timeval to uTime (assignment)
t1 = (timeval)t2;            // convert from uTime to timeval (cast)
t1 = t3;                     // convert from uTime to timeval (implicit cast)
```

Conversion is guaranteed to be exact. The member routine nanoseconds returns the absolute time value as a 64 bit number in nanoseconds from the start of the UNIX epoch.

As for uDuration values, arithmetic and relational operations may be performed on uTime values. As well, mixed mode operations are possible involving durations and time. A duration may be added to or subtracted from a time to yield a new time; two times can be subtracted from each other producing a duration.

10.2 Timeout Operations

It is sometimes necessary to delay a task's execution for a relative duration or until an absolute time. It is also necessary to prevent certain operations from blocking indefinitely. Two such common operations are waiting for an accepted call to occur and waiting for I/O to complete. μC++ provides mechanisms to delay execution for a time interval or terminate an operation after a time interval.

10.2.1 Time Delay

In μC++, a time delay is expressed by either of the following two statements:

```
_Timeout( duration );           // parenthesis required
_Timeout( time );               // parenthesis required
```

With a duration value, **_Timeout** specifies a delay time relative to the start of execution of the statement (i.e., a duration). That is, a task blocks for at least the span of time indicated by the duration value; the task does not consume any resources during this period, nor does it respond to any requests. With a time value, **_Timeout** specifies a delay to an absolute time in the future. That is, a task blocks until at least the specified absolute time has occurred. If the duration value be less than or equal to zero, the task does not block. Similarly, if the time value has already occurred, the task does not block. The UNIX routines sleep and usleep can also be used to sleep for a duration of seconds and microseconds, respectively.

The extended form of the **_Timeout** statement is:

```
_When ( conditional-expression )              // optional guard
    _Timeout( duration or time )              // optional timeout clause
```

A **_When** guard is considered true if it is omitted or if its *conditional-expression* evaluates to non-zero. Before the **_Timeout** statement is executed, the guard must be true. In this case, the guard is the same as an **if** statement, e.g.:

```
_When ( count == 0 ) _Timeout( ... );     ≡   if ( count == 0 ) _Timeout( ... );
```

10.2.2 Accept Statement

The extended form of the **_Accept** statement may specify a timeout value through a **_Timeout** clause, e.g.:

```
_When ( conditional-expression )              // optional guard
    _Accept( mutex-member-name-list )
        statement-1                           // action
or
    ...
        ...
or _When ( conditional-expression )           // optional guard
    _Timeout( duration or time )              // optional timeout clause
        statement-2                           // action
```

The **_Timeout** clause must be the last clause in an **_Accept** statement, or the second last if followed by a terminating **_Else** clause (see Section 2.9.2.1, p. 21). When a **_Timeout** clause and a terminating **_Else** clause appear in the same **_Accept** statement, the pairing is only meaningful if the terminating **_Else** clause is conditional, i.e., has a **_When** guard; otherwise, the **_Else** clause always overrides the **_Timeout**. If there is no guard on a timeout or the guard is true, but a call is accepted before the timeout interval expires, the statement behaves exactly like a normal **_Accept** statement. If there is no guard on a timeout or the guard is true, and no call is accepted before the timeout interval expires, the acceptor is removed from the acceptor/signalled stack, restarts, and executes the statement associated with the **_Timeout** clause.

□ **WARNING:** Beware of the following possible syntactic confusion with the timeout clause:

```
_Accept( mem );                        _Accept( mem );
or _Timeout( uDuration( 1 ) );         _Timeout( uDuration( 1 ) );
```

The left example accepts a call to member mem or times out in 1 second. The right example accepts a call to member mem and then delays for 1 second. The left example is a single accept statement, while the right example is an accept statement and a timeout statement. □

□ **WARNING:** Beware of the following possible syntactic confusion with the timeout clause:

```
_Accept( mem );                      _Accept( mem );
or _Timeout( uDuration( 1 ) );       _When( C1 ) _Else
_When( C1 ) _Else                    _Timeout( uDuration( 1 ) );
```

The left example accepts a call to member mem or times out in 1 second or performs the terminating **_Else**, depending on the value of its guard. The right example accepts a call to member mem or performs the terminating **_Else**, depending on the value of its guard; if the terminating **_Else** is performed, it then delays for 1 second. The left example is a single accept statement, while the right example is an accept statement and a timeout statement, bracketed as follows.

```
_Accept( mem );
_When( C1 ) _Else {
    _Timeout( uDuration( 1 ) );
}
```

□

10.2.3 Select Statement

The extended form of the **_Select** statement may specify a timeout value through a **_Timeout** clause, e.g.:

```
_When ( conditional-expression )                // optional guard
    _Select( selector-expression )
        statement-1                             // action
and/or
   . . .
        . . .
or _When ( conditional-expression )             // optional guard
    _Timeout( duration or time )                // optional timeout clause
        statement-n                             // action
```

The **_Timeout** clause must be the last clause in a **_Select** statement, or the second last if followed by a terminating **_Else** clause (see Section 3.2.1, p. 51). When a **_Timeout** clause and a terminating **_Else** clause appear in the same **_Select** statement, the pairing is only meaningful if the terminating **_Else** clause is conditional, i.e., has a **_When** guard; otherwise, the **_Else** clause always overrides the **_Timeout**. If there is no guard on a timeout or the guard is true, but the selector-expression associated with the entire statement becomes satisfied before the timeout interval expires, the statement behaves exactly like a normal **_Select** statement. If there is no guard on a timeout or the guard is true, and the selector-expression associated with the entire statement fails to become satisfied before the timeout interval expires, the selector task restarts and executes the statement associated with the **_Timeout** clause. One or more of the actions for select clauses composing the select statement may have already triggered when a timeout occurs.

□ **WARNING:** Beware of the following possible syntactic confusion with the timeout clause:

```
_Select( f1 );                       _Select( f1 );
or _Timeout( uDuration( 1 ) );       _Timeout( uDuration( 1 ) );
```

The left example waits for future f1 to becomes available or times out in 1 second. The right example waits for future f1 to becomes available and then delays for 1 second. The left example is a single select statement, while the right example is a select statement and a timeout statement. □

□ **WARNING:** Beware of the following possible syntactic confusion with the timeout clause:

```
_Select( f1 );                       _Select( f1 );
or _Timeout( uDuration( 1 ) );       _When( C1 ) _Else
_When( C1 ) _Else                    _Timeout( uDuration( 1 ) );
```

The left example waits for future f1 to becomes available or times out in 1 second or performs the terminating **_Else**, depending on the value of its guard. The right example waits for future f1 to becomes available or performs the terminating **_Else**, depending on the value of its guard; if the terminating **_Else** is performed, it then delays for 1 second. The left example is a single select statement, while the right example is a select statement and a timeout statement, bracketed as follows.

```
_Select( mem );
_When( C1 ) _Else {
    _Timeout( uDuration( 1 ) );
}
```

□

10.2.4 I/O

Similarly, timeouts can be set for certain I/O operations that block waiting for an event to occur (see details in Appendix D.5.2, p. 169). Only a duration is allowed as a timeout because a relationship between absolute time and I/O seems unlikely. A pointer to the duration value is used so it is possible to distinguish between no timeout value (NULL pointer) and a zero-timeout value. The former usually means to wait until the event occurs (i.e., no timeout), while the latter can be used to poll by trying the operation and returning immediately if the event has not occurred. The I/O operations that can set timeouts are read, readv, write, writev, send, sendto, sendmsg, recv, recvfrom and readmsg. If the specified I/O operation has not completed when the delay expires, the I/O operation fails by throwing an exception. The exception types are ReadTimeout for read, readv, recv, recvfrom and readmsg, and WriteTimeout for write, writev, send, sendto and sendmsg, respectively. For example, in:

```
try {
    uDuration d( 3, 0 );        // 3 second duration
    fa.read( buf, 512, &d );
    // handle successful read
} catch( uFileIO::ReadTimeout ) {
    // handle read failure
}
```

the read operation expires after 3 seconds if no data has arrived.

As well, a timeout can be set for the constructor of a uSocketAccept and uSocketClient object, which implies that if the acceptor or client has not made a connection when the delay expires, the declaration of the object fails by throwing an exception (see details in Appendix D.5.4, p. 172). For example, in:

```
try {
    uDuration d( 60, 0 );       // 60 second duration
    uSocketAccept acceptor( sockserver, &d );     // accept a connection from a client
    // handle successful accept
} catch( uSocketAccept::OpenTimeout ) {
    // handle accept failure
} // try
```

See, also, the server examples in Appendix D.5, p. 167.

10.3 Clock

A clock defines an absolute time and is used for interrogating the current time. Multiple clocks can exist; each one can be set to a different time. In theory, all clocks tick together at the lowest clock resolution available on the computer.

The type uClock creates a clock object, and is defined:

```
class uClock {
  public:
    uClock();
    uClock( uTime adj );
    void resetClock();
    void resetClock( uTime adj );
    uTime getTime();
    void getTime( int &year, int &month, int &day, int &hour, int &minutes,
                  int &seconds, long int &nsec );
    static void convertTime( uTime time, int &year, int &month, int &day, int &hour, int &minutes,
                             int &seconds, long int &nsec );
}; // uClock
```

The overloaded constructor routine uClock has the following forms:

uClock() – creates a clock as a real-time clock running at the same time as the underlying virtual process.

uClock(uTime adj) – creates a clock as a virtual clock starting at time adj.

The overloaded member routine resetClock resets the kind of clock between real-time and virtual, and each routine has the following form:

resetClock() – this form sets the clock to a real-time clock, so it returns the current time of the underlying virtual processor.

resetClock(uTime adj) – this form sets the clock to a virtual clock starting at time adj.

The overloaded member routine getTime returns the current time, and each routine has the following form:

getTime() – this form returns the current time as a uTime value, i.e., in nanoseconds from the start of the UNIX epoch.

getTime(**int** &year, **int** &month, **int** &day, **int** &hour, **int** &minutes, **int** &seconds, **long int** &nsec) – this form returns the current time broken up into the traditional non-fixed radix units of time.

The static member routine convertTime converts the specified time in nanoseconds from the start of the UNIX epoch into a traditional non-fixed radix units of time.

As mentioned, each virtual processor has its own real-time clock. The current time is available from a virtual processor via the call uThisProcessor().getClock().getTime(); hence, it is unnecessary to create a clock to get the current time.

10.4 Periodic Task

Without a programming language construct to specify periodicity, and without programming language facilities to express time, it is almost impossible to accurately express time specifications within a program. Specifying a periodic task in a language without proper time constructs can introduce catastrophic inaccuracies. For example, in:

```
1  for ( ; ; ) {
2      // periodic work
3      uDuration DelayTime = NextTime - CurrentTime();
4      _Timeout( DelayTime );
5  }
```

if the task is context-switched after executing line 3 (or context-switched after the call to CurrentTime in line 3), the DelayTime would be inaccurate. As a result, the blocking time of the program is erroneous.

The above problem can be eliminated by specifying an absolute time to **_Timeout** (specifying NextTime as the parameter to **_Timeout**). However, with this form of periodic task specification, it is infeasible to specify other forms of deadlines. Ada only supports the periodic task specification using delays, and the system guarantees a periodic task delays for a minimum time specified in DelayTime, but makes no guarantee as to when the periodic task actually gets to execute [BP91]. As a result, a task can request to block for 10 seconds (and Ada guarantees it blocks for at least 10 seconds), but end up executing 20 seconds later.

To circumvent this problem, *μ*C++ provides a periodic task. The general form of the periodic task type is the following:

```
_PeriodicTask task-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 a task, a periodic task type has one distinguished member, named main, in which the new thread starts execution. If not derived from some other periodic task type, each periodic task type is implicitly derived from the task type uPeriodicBaseTask, e.g.:

```
_Task task-name : public uPeriodicBaseTask {
      . . .
};
```

where the interface for the base class uPeriodicBaseTask is:

```
_Task uPeriodicBaseTask {
  protected:
    uTime firstActivateTime;
    uEvent firstActivateEvent;
    uTime endTime;
    uDuration period;
  public:
    uPeriodicBaseTask( uDuration period, uCluster &cluster = uThisCluster() );
    uPeriodicBaseTask( uDuration period, uTime firstActivateTask, uTime endTime,
                       uDuration deadline, uCluster &cluster = uThisCluster() );
    uPeriodicBaseTask( uDuration period, uEvent firstActivateEvent, uTime endTime,
                       uDuration deadline, uCluster &cluster = uThisCluster() );
    uPeriodicBaseTask( uDuration period, uTime firstActivateTask, uEvent firstActivateEvent,
                       uTime endTime, uDuration deadline, uCluster &cluster = uThisCluster() );
    uDuration getPeriod() const;
    uDuration setPeriod( uDuration period );
};
```

A periodic task starts by one of two mechanisms. The first is by specifying a start time, FirstActivateT, at which the periodic task begins execution. The second is by specifying an event, FirstActivateE (an interrupt), upon receipt the event the periodic task begins execution. If both start time and event are specified, the task starts either on receipt of an event or when the specified time arrives, whichever comes first. If neither time nor event are specified, the periodic task starts immediately. An end time, EndTime, may also be specified. When the specified end time occurs, the periodic task halts after execution of the current period. A deadline, Deadline, may also be specified. A deadline is expressed as the duration from the beginning of a task's period by which its computation must be finished. A zero argument for any of the parameters indicates the task is free from the constraints represented by the parameter (the exception is Period, which cannot have a zero argument). For example, if the FirstActivate parameter is zero, the task is scheduled for initial execution at the next available time it can be accommodated. Finally, the cluster parameter specifies which cluster the task should be created in. Should this parameter be omitted, the task is created on the current cluster.

An example of a periodic task declaration that starts at a specified time and executes indefinitely (without any deadline constraints) is:

```
_PeriodicTask task-name {
    void main() { periodic task body }
  public:
    task-name( uDuration period, uTime time ) : uPeriodicBaseTask( period, time, 0, 0 ) { };
};
```

The task body, i.e., routine main, is implicitly surrounded with a loop that performs the task body periodically. As a result, terminating the task body requires a **return** (or the use of an end time); falling off the end of the main routine does not terminate a periodic task.

10.5 Sporadic Task

A sporadic task is similar to a periodic task, except there is a minimum duration between executions instead of a fixed period. In the declaration of a sporadic task, this minimum duration is specified as a **frame**. It is the user's responsibility to ensure the execution does not exceed the specified minimum duration (i.e., frame); otherwise, the scheduler cannot ensure correct execution. The reason the scheduler cannot automate this process, as it does for periodic tasks, is because of the unpredictable nature of the inter-arrival time of sporadic tasks.

In μC++, a sporadic task is similar to a periodic task. A **_SporadicTask** task type, if not derived from some other sporadic task type, is implicitly derived from the task type uSporadicBaseTask, e.g.:

```
_Task uSporadicBaseTask {
  protected:
    uTime firstActivateTime;
    uEvent firstActivateEvent;
    uTime endTime;
    uDuration frame;
  public:
    uSporadicBaseTask( uDuration frame, uCluster &cluster = uThisCluster() );
    uSporadicBaseTask( uDuration frame, uTime firstActivateTask, uTime endTime,
                       uDuration deadline, uCluster &cluster = uThisCluster() );
    uSporadicBaseTask( uDuration frame, uEvent firstActivateEvent, uTime endTime,
                       uDuration deadline, uCluster &cluster = uThisCluster() );
    uSporadicBaseTask( uDuration frame, uTime firstActivateTask, uEvent firstActivateEvent,
                       uTime endTime, uDuration deadline, uCluster &cluster = uThisCluster() );
    uDuration getFrame() const;
    uDuration setFrame( uDuration frame );
};
```

10.6 Aperiodic Task

An aperiodic task has a non-deterministic start pattern. As a result, aperiodic tasks should only be used in soft real-time applications.

In μC++, an aperiodic task is similar to a periodic task. A **_RealTimeTask** task type, if not derived from some other aperiodic task type, is implicitly derived from the task type **_RealTimeTask**, e.g.:

```
_Task uRealTimeBaseTask {
  protected:
    uTime firstActivateTime;
    uEvent firstActivateEvent;
    uTime endTime;
  public:
    uRealTimeBaseTask( uCluster &cluster = uThisCluster() );
    uRealTimeBaseTask( uTime firstActivateTask, uTime endTime, uDuration deadline,
                       uCluster &cluster = uThisCluster() );
    uRealTimeBaseTask( uEvent firstActivateEvent, uTime endTime, uDuration deadline,
                       uCluster &cluster = uThisCluster() );
    uRealTimeBaseTask( uTime firstActivateTask, uEvent firstActivateEvent, uTime endTime,
                       uDuration deadline, uCluster &cluster = uThisCluster() );
    uDuration getDeadline() const;
    uDuration setDeadline( uDuration deadline );
};
```

10.7 Priority Inheritance Protocol

The **priority-inheritance protocol** attacks the problem of priority inversion, where a high-priority task waits while lower-priority tasks prevent it from executing. Rajkumar proposed the **basic priority-inheritance protocol** [RSL88, SRL90, Raj91], which puts a bound on the occurrence of priority inversion. The solution is to execute a critical section at the priority of the highest blocked task waiting to enter it. The basic priority-inheritance protocol bounds the time priority inversion occurs: should there be n lower-priority tasks in the system, and the n lower-priority tasks access m distinct critical sections, a task can be blocked by at most $min(n,m)$ critical sections. Despite this bound, the blocking duration for a task can still be significant, however. Suppose at time t_0, a low-priority task τ_2 arrives and locks monitor M_0. At time t_1, a medium priority task τ_1 arrives, pre-empts τ_2, and locks monitor M_1. At time t_2, a high-priority task τ_0 arrives, needing to sequentially access both monitors M_0 and M_1. Since both monitors are locked by two lower-priority tasks, τ_0 must wait for the duration of two critical sections (till M_1 is released by τ_1, then, M_0 is released by τ_2). This problem is known as **chain blocking**. Finally, this priority-inheritance protocol does not deal with the problem of deadlock.

In μC++, tasks wait for entry into a mutex object on a prioritized-entry queue. More specifically, each of the mutex object's member routines have an associated prioritized-entry queue. When the mutex object becomes unlocked, the next task that enters is the one with the highest priority among all the entry queues. Should a mutex object be locked and a higher-priority task arrives, the current task executing inside the mutex object "inherits" the priority of the highest-priority task awaiting entry. This semantics ensures the task inside the mutex object can only be interrupted by a higher-priority task, allowing the task in the mutex object to complete and leave as soon as possible, which speeds entry of a waiting higher-priority task.

Condition variables and their associated queues of waiting tasks are also a fundamental part of mutex objects. Signalling a condition variable makes the highest-priority task on the queue eligible to run. In μC++, the signaller continues execution after signalling, at the priority of the highest-priority task awaiting entry to the mutex object. As well, the signalled task is given preference over other tasks awaiting entry to the mutex object. Therefore, the signalled task is the next to execute in the mutex object, regardless of whether there are higher-priority tasks waiting in the entry queues. This behaviour, in turn, creates the possibility of priority inversion. Should a high-priority task be awaiting entry to the mutex object, and a lower-priority task executing in the mutex object signals a condition queue whose most eligible task has a lower priority than a task awaiting entry to the mutex object, priority inversion results. Hence, the semantics of μC++ mutex objects increases the original algorithm's bound for priority inversion by the amount it takes to complete the execution of all the tasks in the signalled stack.

Finally, in μC++, tasks running inside a mutex object have the additional capability of specifically accepting any one of the mutex member routines. This capability also brings about the possibility of bypassing higher-priority tasks waiting on other entry queues. When a member routine is accepted, the acceptor is moved to the signalled stack, thus causing the acceptor to block; the highest-priority task waiting on the accepted member routine then executes. When a task leaves a mutex object, the next task that executes is selected first from the signalled stack not the entry queues. Thus, the amount of time when priority inversion can take place when accepting specific member routines is unbounded, since tasks can continually arrive on a member routine's entry queue, and tasks executing in the mutex object can continually accept the same specific member routine.

10.8 Real-Time Scheduling

The notion of priority becomes a crucial tool for implementing various forms of scheduling paradigms [AGMK94, BW90, Gol94]. In general, the term **priority** has no single meaning. The priority of a task may signify its logical importance to a programmer, or may simply be a property determined by its periodic characteristics, as is the case with certain scheduling algorithms.

In μC++, the notion of priority simply determines the order by which a set of tasks executes. As far as the real-time system is concerned, the ready task with the highest priority is the most eligible task to execute, with little or no regard for the possible starvation of lower-priority ready tasks. This form of scheduling is referred to as a **prioritized pre-emptive scheduling**.

Each task's priority can be redefined and queried by the routines provided from the following abstract class (discussed further in the next section):

```
template<typename Node> class uBaseSchedule {
  protected:
    uBaseTask &getInheritTask( uBaseTask &task ) const;
    int getActivePriority( uBaseTask &task ) const;
    int setActivePriority( uBaseTask &task1, uBaseTask &task2 );
    int getBasePriority( uBaseTask &task ) const;
    int setBasePriority( uBaseTask &task, int priority );
}; // uBaseScheduleFriend
```

Scheduler objects inherit from uBaseSchedule to use these routines, not replace them. These routines provide sufficient information about the dynamic behaviour of tasks on a cluster to schedule them in various ways.

To provide the facilities for implementing various priority-changing scheduling algorithms (such as priority inheritance), a μC++ task has two priorities associated with it: a **base priority** and an **active priority**. It is up to the scheduler implementor or programmer to set the appropriate priority values, or to determine whether the base priority or the active priority is the priority utilized in scheduling tasks, if used at all.[1]

The member routine getInheritTask returns the task that this task inherited its current active priority from or NULL. The member routines getActivePriority and setActivePriority read and write a task's active priority, respectively. The member routines getBasePriority and setBasePriority read and write a task's base priority, respectively.

A task's priority can be used for more than just determining which task executes next; priorities can also dictate the behaviour of various synchronization primitives such as semaphores and monitors [BW90]. μC++ monitors

[1] μC++ sets the base and the active priority of a task to a uniform default value, if no other priority is specified.

have been extended so that entry queues (see Section 2.9.1, p. 21) are prioritized.[2] The highest-priority task that calls into a monitor always enters the monitor first, unless a particular entry queue is explicitly accepted (see Section 2.9.2.1, p. 21), in which case, the highest-priority task in the particular entry queue executes. Condition queues (see Section 2.9.3.1, p. 25) within a monitor are also prioritized: signaling a condition queue schedules the highest-priority task waiting on the queue. Thus, both the monitor entry queues and the condition queues are prioritized, with FIFO used within each priority level. The current implementation provides 32 priority levels. Support for more or less priority levels can be implemented (see Section 10.9).

If an application is not real-time, all tasks are assigned an equal, default priority level. Thus, all tasks have one active priority, and the scheduling is FIFO.

10.9 User-Supplied Scheduler

One of the goals of real-time in μC++ is to provide a flexible system, capable of being adapted to various real-time environments and applications. The wide availability of various real-time scheduling algorithms, coupled with each algorithm's suitability for different forms of real-time applications, makes it essential that the language and runtime system provide as few restrictions as possible on which algorithms may be utilized and implemented.

Scheduling is the mechanism by which the next task to run is chosen from a set of runnable tasks. However, this selection mechanism is closely tied to the data structure representing the set of runnable tasks. In fact, the data structure containing the set of runnable tasks is often designed with a particular scheduling algorithm in mind.

To provide a flexible scheduler, the ready "queue"[3] is packaged as an independent entity – readily accessible and replaceable by a scheduler designer. Consequently, the rules and mechanisms by which insertion and removal take place from the ready data-structure is completely up to the implementor.

A ready data-structure is generic in the type of nodes stored in the structure and must inherit from the abstract class:

```
template<typename Node> class uBaseSchedule {
  public:
    virtual void add( Node *node ) = 0;
    virtual Node *pop() = 0;
    virtual bool empty() const = 0;
    virtual bool checkPriority( Node &owner, Node &calling ) = 0;
    virtual void resetPriority( Node &owner, Node &calling ) = 0;
    virtual void addInitialize( uSequence<uBaseTaskDL> &taskList ) = 0;
    virtual void removeInitialize( uSequence<uBaseTaskDL> &taskList ) = 0;
    virtual void rescheduleTask( uBaseTaskDL *taskNode, uBaseTaskSeq &taskList ) = 0;
};
```

The μC++ kernel uses the routines provided by uBaseSchedule to interact with the user-defined ready queue.[4] A user can construct different scheduling algorithms by modifying the behaviour of member routines add and pop, which add and remove tasks from the ready queue, respectively. To implement a dynamic scheduling algorithm, an analysis of the set of runnable tasks is performed for each call to add and/or pop by the kernel; these routines alter the priorities of the tasks accordingly. The member routine empty returns true if the ready queue is empty and false otherwise. The member routine checkPriority provides a mechanism to determine if a calling task has a higher priority than another task, which is used to compare priorities in priority changing protocols, such as priority inheritance. Its companion routine resetPriority performs the same check, but also raises the priority of the owner task to that of the calling task if necessary. addInitialize is called by the kernel whenever a task is added to the cluster, and removeInitialize is called by the kernel whenever a task is deleted from the cluster. In both cases, a pointer to the ready queue for the cluster is passed as an argument so it can be reorganized if necessary. The type uSequence<uBaseTaskDL> is the type of a system ready queue (see Appendix C, p. 149 for information about the uSequence collection). The list node type, uBaseTaskDL, stores a reference to a task, and this reference can be retrieved with member routine task:

[2] A task's active priority is utilized by a μC++ monitor to determine a task's priority value

[3] The term "ready queue" is no longer appropriate because the data structure may not be a queue.

[4] Operating systems such as Amoeba [TvRvS+90], Chorus [RAA+88], and Apertos [Yok92] employ a similar mechanism by which the kernel utilizes external modules to modify its behaviour.

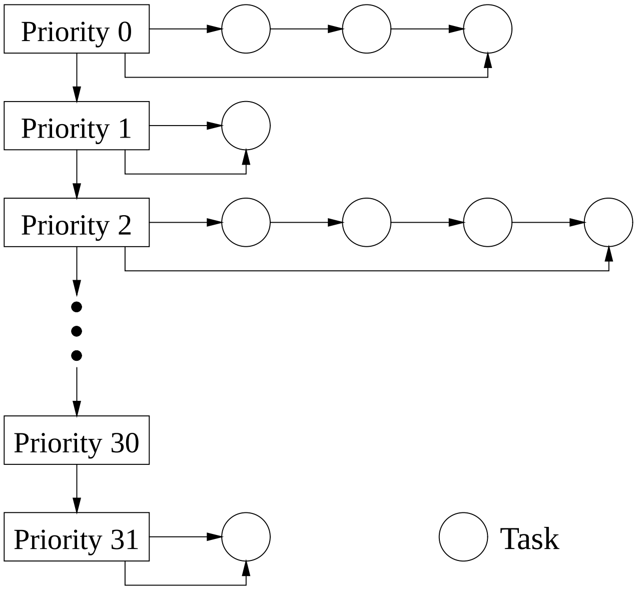


Figure 10.3: Deadline Monotonic Ready-Queue

```
class uBaseTaskDL : public uSeqable {
  public:
    uBaseTaskDL( uBaseTask &_task );
    uBaseTask &task() const;
}; // uBaseTaskDL
```

Note, adding (or deleting) tasks to (or from) a cluster is not the same as adding or popping tasks from the ready queue. With a static scheduling algorithm, for example, task-set analysis is only performed upon task creation, making the addInitialize function an ideal place to specify such analysis code. The member routine rescheduleTask is used to recalculate the priorities of the tasks on a cluster based on the fact that a given task, taskNode, may have changed some of its scheduling attributes.

10.10 Real-Time Cluster

A **real-time cluster** behaves just like a normal μC++ cluster, except a real-time cluster can have a special ready data-structure associated with it (the ready data-structure, in turn, has a scheduling or task-dispatching policy associated with it). The ready data-structure must inherit from the uBaseSchedule class, however, and passed as an argument when creating a real-time cluster. A real-time cluster has the following constructors:

```
class uRealTimeCluster : public uCluster {
  public:
    uRealTimeCluster( uBaseSchedule<uBaseTaskDL> &rq, int size = uDefaultStackSize(),
                      const char *name = "" );
    uRealTimeCluster( uBaseSchedule<uBaseTaskDL> &rq, const char *name );
    ~uRealTimeCluster() {};
};
```

10.10.1 Deadline Monotonic Scheduler

The **deadline monotonic** scheduling algorithm is an example of a task-dispatching policy requiring a special ready data-structure, which can be plugged into a real-time cluster. The underlying ready data-structure for the deadline monotonic implementation is a prioritized ready-queue, with support for 32 priority levels. The add routine adds a task to the ready-queue in a FIFO manner within a priority level. The pop routine returns the most eligible task with the highest priority from the ready-queue. Both add and pop utilize a constant-time algorithm for the location of the highest-priority task. Figure 10.3 illustrates this prioritized ready-queue.

The addInitialize routine contains the heart of the deadline monotonic algorithm. In addInitialize, each task in the ready-queue is examined, and tasks are ordered in increasing order by deadline. Priorities are, in turn, assigned to

```
#include<uDeadlineMonotonic.h>

_PeriodicTask PeriodicTask1 {
  public:
    PeriodicTask1( uDuration period, uTime endtime, uDuration deadline, uCluster &cluster ) :
                uPeriodicBaseTask( period, uTime(0), endtime, deadline, cluster ) {
    }
    void main() {
        // periodic task body
    }
};

_PeriodicTask PeriodicTask2 {
  public:
    PeriodicTask2( uDuration period, uTime endtime, uDuration deadline, uCluster &cluster ) :
                uPeriodicBaseTask( period, uTime(0), endtime, deadline, cluster ) {
    }
    void main() {
        // periodic task body
    }
};

void uMain::main() {
    uDeadlineMonotonic dm;              // create real-time scheduler
    uRealTimeCluster RTClust( dm );     // create real-time cluster with scheduler
    uProcessor *processor;
    {
        // These tasks are created, but they do not begin execution until a
        // processor is created on the “RTClust” cluster.  This is ideal, as
        // “addInitialize” is called as each task is added to the cluster.

        uTime currTime = uThisProcessor().getClock().getTime();
        PeriodicTask1 t1( 15, currTime+90,  5, RTClust );     // 15 sec period, 5  sec deadline
        PeriodicTask2 t2( 30, currTime+90, 10, RTClust );     // 30 sec period, 5  sec deadline
        PeriodicTask1 t3( 60, currTime+90, 20, RTClust );     // 60 sec period, 20 sec deadline

        // Only when all tasks are on the cluster, and the scheduling algorithm
        // as ordered the tasks, is a processor associated with cluster
        // “RTClust” to execute the tasks on the cluster.

        processor = new uProcessor( RTClust );
    } // wait for t1, t2, and t3 to finish
    delete processor;
}
```

Figure 10.4: Sample Real-Time Program

every task. With the newly assigned priorities, the ready queue is re-evaluated, to ensure it is in a consistent state. As indicated in Section 10.9, p. 137, this routine is usually called only by the kernel. If a task is removed from the cluster, the relative order of the remaining tasks is unchanged; hence, the task is simply deleted without a need to re-schedule.

A sample real-time program is illustrated in Figure 10.4. To utilize the deadline-monotonic algorithm include header file uDeadlineMonotonic.h. In the example, the creation of the real-time scheduler and cluster is done at the beginning of uMain::main. Note, the argument passed to the constructor of uRealTimeCluster is an instance of uDeadlineMonotonic, which is a ready data-structure derived from uBaseSchedule.

The technique used to ensure that the tasks start at a critical instance is not to associate a processor with the cluster until after all tasks are created *and scheduled* on the cluster. As each task is added to the cluster addInitialize is called, and cluster's task-set is analyzed and task priorities are (re)assigned. After priority assignment, the task is added to the ready queue, and made eligible to execute. Only when all tasks are created is a processor finally associated with the real-time cluster. This approach ensures that when the processor is put in place, the task priorities are fully determined, and the critical instant is ensured.

Chapter 11

Miscellaneous

11.1 Default Values

μC++ has a number of environment variables set to reasonable initial values for a basic concurrent program. However, some concurrent programs may need to adjust these values to obtain correct execution or enhanced performance. Currently, these variables affect tasks, processors, and the heap.

A default value is specified indirectly via a default routine, which returns the specific default value. A routine allows an arbitrary computation to generate an appropriate value. Each default routine can be replaced by defining a routine with the same name and signature in an application, e.g.:

```
unsigned int uDefaultStackSize() {
    return 64 * 1024;                  // 64K default stack size
}
```

If the value of a global variable is used in the computation, the application can change the default value dynamically by changing this global variable; hence, actions performed at different times are initialized with different default values (unless overridden locally). However, the global variable *must be statically initialized* because its value may be used to initialize objects at the start of the μC++ runtime, i.e., before the application's code starts execution.

11.1.1 Task

The following default routines directly or indirectly affect tasks:

```
unsigned int uDefaultStackSize();   // cluster coroutine/task stack size (bytes)
unsigned int uMainStackSize();      // uMain task stack size (bytes)
unsigned int uDefaultPreemption();  // processor scheduling pre-emption duration (milliseconds)
```

Routine uDefaultStackSize returns a stack size *to initialize a cluster's default stack-size* (versus being used directly to initialize a coroutine/task stack-size). A coroutine/task created on a cluster without an explicit stack size is initialized to the cluster's default stack-size; hence, there is a level of indirection between this default routine and its use for initializing a stack size. As well, a cluster's default stack-size can be explicitly changed after the cluster is created (see Section 8.3, p. 115). Routine uMainStackSize is used directly to provide a stack size for the implicitly declared initial task of type uMain (see Section 2.2, p. 8). Since this initial task is defined and created by μC++, it has a separate default routine so it can be adjusted differently from the application tasks. Routine uDefaultPreemption returns a time in milliseconds *to initialize a virtual processor's default pre-emption time* (versus being used directly to initialize a task's pre-emption time). A task executing on a processor is rescheduled after no more than this amount of time (see Section 8.4, p. 117).

11.1.2 Processor

The following default routines directly affect processors:

```
unsigned int uDefaultSpin();        // processor spin amount before becoming idle
unsigned int uDefaultProcessors();  // number of processors created on the user cluster
```

Routine uDefaultSpin returns the maximum number of times the cluster's ready queue is checked for an available task to execute before the processor blocks. As well, a processor's default spin can be explicitly changed after the processor is created (see Section 8.4, p. 117). Routine uDefaultProcessors returns the number of implicitly created virtual processors on the user cluster (see Section 2.3.2, p. 9). When the user cluster is created, at least this many processors are implicitly created to execute tasks concurrently.

11.1.3 Heap

The following default routine directly affects the heap:

```
unsigned int uDefaultHeapExpansion(); // heap expansion size (bytes)
unsigned int uDefaultMmapStart();    // division point (bytes)
```

Routine uDefaultHeapExpansion returns the amount to extend the heap size once all the current storage in the heap is allocated (see Section 7.2.3.10, p. 112). Routine uDefaultMmapStart returns the division point after which allocation requests are separately mmapped rather than being allocated from the heap area.

11.2 Symbolic Debugging

The symbolic debugging tools (e.g., dbx, gdb) do not work perfectly with *μ*C++, because each coroutine and task has its own stack, and the debugger does not know about these stacks. When a program terminates with an error, only the stack of the coroutine or task in execution at the time of the error is understood by the debugger. During execution, the debugger cannot step through a context switch for either a coroutine or task. Therefore, it is necessary to put break points after the suspend/resume or signal/wait to acquire control, and just continue execution through the context switch. Once the breakpoint is reached, it is possible to next/step through the lines of the coroutine/task until the next context switch. For debuggers that handle multiple kernel threads (corresponding to *μ*C++ virtual processors), it is possible to examine the active task running on each kernel thread.

```
(gdb) info threads    # list all kernel threads
(gdb) thread 2        # switch to kernel thread 2
```

Finally, it is necessary to tell the debugger that *μ*C++ is handling UNIX signals SIGALRM and SIGUSR1 to perform pre-emptive scheduling. For gdb, the following debugger commands allows the application program to handle signal SIGALRM and SIGUSR1:

```
(gdb) handle SIGALRM nostop noprint pass
(gdb) handle SIGUSR1 nostop noprint pass
```

11.3 Installation Requirements

*μ*C++ comes configured to run on any of the following platforms (single and multiple processor):

- solaris-sparc : Solaris 9/10, SPARC
- linux-x86 : Linux 2.4.x or greater, Intel IA-32
- linux-ia64 : Linux 2.4.x or greater, Intel IA-64 (Itanium)
- linux-x86_32 : Linux 2.4.x or greater, AMD 32
- linux-x86_64 : Linux 2.4.x or greater, AMD 64

*μ*C++ requires at least GNU [Tie90] g++-4.5.0 or greater. These compilers can be obtained free of charge. ***μC++ does NOT compile using other compilers or operating systems.***

11.4 Installation

The current version of *μ*C++ can be obtained from Github. Execute the following command to install:

```
$ git clone https://github.com/pabuhr/uCPP.git
$ cd uCPP
$ sudo sh install.sh
```

or from:

http://plg.uwaterloo.ca/∼usystem/pub/uSystem/u++-6.1.0.sh

Execute the following command to install:

```
$ sudo sh u++-6.1.0.sh
```

11.5 Reporting Problems

If you have problems or questions or suggestions, send e-mail to usystem@plg.uwaterloo.ca or mail to:

> μSystem Project
> c/o Peter A. Buhr
> School of Computer Science
> University of Waterloo
> Waterloo, Ontario
> N2L 3G1
> CANADA

As well, visit the μSystem web site at: http://plg.uwaterloo.ca/~usystem

11.6 Contributors

While many people have made numerous suggestions, the following people were instrumental in turning this project from an idea into reality. The original design work, Version 1.0, was done by Peter Buhr, Glen Ditchfield and Bob Zarnke [BDZ89], with additional help from Jan Pachl on the train to Wengen. Brian Younger built Version 1.0 by modifying the AT&T 1.2.1 C++ compiler [You91]. Version 2.0 was designed by Peter Buhr, Glen Ditchfield, Rick Stroobosscher and Bob Zarnke [BDS+92]. Version 3.0 was designed by Peter Buhr, Rick Stroobosscher and Bob Zarnke. Rick Stroobosscher built both Version 2.0 and 3.0 translator and kernel. Peter Buhr wrote the documentation and built the non-blocking I/O library as well as doing other sundry coding. Version 4.0 kernel was designed and implemented by Peter Buhr. Nikita Borisov and Peter Buhr fixed several problems in the translator. Amir Michail started the real-time work and built a working prototype. Philipp Lim and Peter Buhr designed the first version of the real-time support and Philipp did most of the implementation with occasional help from Peter Buhr. Ashif Harji and Peter Buhr designed the second version of the real-time support and Ashif did most of the implementation with occasional help from Peter Buhr. Russell Mok and Peter Buhr designed the first version of the extended exception handling and Russell did most of the implementation with occasional help from Peter Buhr. Roy Krischer and Peter Buhr designed the second version of the extended exception handling and Roy did most of the implementation with occasional help from Peter Buhr. Version 5.0 kernel was designed and implemented by Richard Bilson and Ashif Harji, with occasional help from Peter Buhr. Tom, Sasha, Tom, Raj, and Martin, the “gizmo guys”, all helped Peter Buhr and Ashif Harji with the gizmo port. Finally, the many contributions made by all the students in CS342/CS343 (Waterloo) and CSC372 (Toronto), who struggled with earlier versions of μC++, is recognized.

The indirect contributers are Richard Stallman for providing emacs and gmake so that we could accomplish useful work in UNIX, Michael D. Tiemann and Doug Lea for providing the initial version of GNU C++ and Dennis Vadura for providing dmake (used before gmake).

Appendix A

*μ*C++ Grammar

The grammar for *μ*C++ is an extension of the grammar for C++ given in [C++98, Annex A]. The ellipsis in the following rules represent the productions elided from the C++ grammar.

function-specifier :
. . .
mutex-specifier

mutex-specifier :
_Mutex *queue-types*$_{opt}$
_Nomutex *queue-types*$_{opt}$

queue-types :
< *class-name* >
< *class-name* , *class-name* >

class-key :
mutex-specifier$_{opt}$ **class**
. . .
mutex-specifier$_{opt}$ **_Coroutine**
mutex-specifier$_{opt}$ **_Task** *queue-types*$_{opt}$
_RealTimeTask *queue-types*$_{opt}$
_PeriodicTask *queue-types*$_{opt}$
_SporadicTask *queue-types*$_{opt}$
_Event

statement :
. . .
accept-statement ;
select-statement ;
_Disable *(exception-)identifier-list*$_{opt}$ *statement* ;
_Enable *(exception-)identifier-list*$_{opt}$ *statement* ;

jump-statement :
break *identifier*$_{opt}$;
continue *identifier*$_{opt}$;
. . .

accept-statement :
or-accept
or-accept timeout-clause
or-accept else-clause
or-accept timeout-clause else-clause

or-accept :
accept-clause
or-accept **or** *accept-clause*

accept-clause :
when-clause$_{opt}$ **_Accept** (*(mutex-)identifier-list*) *statement*

select-statement :
or-select
or-select timeout-clause
or-select else-clause
or-select timeout-clause else-clause

or-select :
and-select
or-select **or** *and-select*

and-select :
select-clause
and-select **and** *select-clause*

select-clause :
when-clause$_{opt}$ (*or-select*)
when-clause$_{opt}$ **_Select** (*(selector-)expression*) *statement*

when-clause :
_When (*expression*)

else-clause :
when-clause$_{opt}$ **_Else** *statement*

timeout-clause :
or *when-clause*$_{opt}$ **_Timeout** (*(time-)expression*) *statement*

try-block :
try *compound-statement handler-seq finally*$_{opt}$

handler :
_CatchResume (*exception-declaration*) *compound-statement*
_CatchResume (*lvalue* . *exception-declaration*) *compound-statement*
catch (*exception-declaration*) *compound-statement*
catch (*lvalue* . *exception-declaration*) *compound-statement*

finally :
_Finally *compound-statement*

throw-expression :
. . .
_Throw *assignment-expression*$_{opt}$
_Resume *assignment-expression*$_{opt}$ *at-expression*$_{opt}$

at-expression :
_At *assignment-expression*

Appendix B

Heap Allocator

μC++ has its own heap allocator. The heap is a binning allocator where object requests are rounded up to a bin size, and it uses a lock per bin. This approach tries to reduce contention by using fine-grain locking for different request sizes. The bin sizes are closer together for smaller sizes and further apart for larger sizes. A free list is maintained for each bin size. The free list for the corresponding bin size is quickly checked for a free object of the correct size. If there are no free objects to reuse, a new object is allocated from an allocation buffer. An adjustable-sized allocation-buffer is allocated using sbrk, and it uses a single lock to protect expansion. Each object has a header at the start. At an adjustable point, the allocator switches from binning in the heap area to using mmap for allocations; these allocations are returned to the operating system immediately after deallocation. This approach avoids external fragmentation if large-object requests are infrequent.

The μC++ allocator has the standard heap-operations:

malloc(size) allocate size bytes and return a pointer to the allocated memory; memory is uninitialized.

calloc(N, size) allocate N elements of size bytes (N × size bytes) and return a pointer to the allocated memory; memory is zero filled.

realloc(ptr, size) change the size of the memory referenced by ptr to size bytes and return a pointer to the allocated memory; if the size change requires new storage (smaller or larger), as much data as possible it copied from the old storage and the old storage is freed. Any uncopied memory in the new storage is uninitialized.

posix_memalign(alignment, size) allocate size bytes and return a pointer to the allocated memory; the address returned is a multiple of alignment and memory is uninitialized.

free(ptr) deallocate memory referenced by ptr, which must have been allocated by a call to malloc, calloc or realloc.

The following extended routines (not defined by POSIX) are supported:

memalign(alignment, size) allocate size bytes and return a pointer to the allocated memory; the address returned is a multiple of alignment and memory is uninitialized.

mallopt(parameter, value) adjust parameters of the memory allocator for a specific application. Supported parameters are (see also Section 11.1.3, p. 142):

M_TOP_PAD sets the amount to extend the heap size once all the current storage in the heap is allocated.

M_MMAP_THRESHOLD sets the division point after which allocation requests are separately memory mapped rather than being allocated from the heap area.

malloc_usable_size(ptr) returns the usable size of the memory referenced by ptr.

calloc zero fills storage up to malloc_usable_size.

malloc_stats() print memory-allocation statistics on standard error (2, default).

malloc_stats_fd(int fd) set file-descriptor number for memory-allocation statistics printing (1 ⇒ standard out).

Finally, the heap interface is further extended across the following properties: zero fill and alignment.

	minimum alignment	specified alignment
no zero fill	malloc	memalign
zero fill	calloc	**cmemalign**

Note the missing interface to allocate aligned storage that is zero filled; therefore, the interface is extended to complete the table with routine:

cmemalign(alignment, size) allocate size bytes and return a pointer to the allocated memory; the address returned is a multiple of alignment and memory is zero filled.

This routine completes programmer expectation with respect to accessing the allocation properties. As well, current heap semantics do not preserve zero fill and alignment properties with respect to re-allocation. For example, when memory is allocated by calloc, memalign or cmemalign with zero fill and/or alignment properties, that memory can be given to realloc to change its size. If realloc needs to allocate new storage to satisfy a size change, POSIX does *not require* the new storage to preserve any initial properties; therefore, subsequent use of the new storage from realloc may cause inefficient execution or errors due to lack of zero fill or alignment. This silent generation of a problem is unintuitive to programmers and difficult to locate because it is transient. Therefore, the semantics of realloc are augmented and some additional query routines are added:

realloc(ptr, size) same semantics as before with the addition that zero fill and alignment properties are preserved if a new allocation is made.

malloc_alignment(ptr) returns the alignment of the memory referenced by ptr. If ptr is NULL, the minimal alignment for any allocation is returned, e.g., multiple of 8.

malloc_zero_fill(ptr) returns a boolean result indicating if memory referenced by ptr was allocated with zero fill (i.e., by calloc/cmemalign).

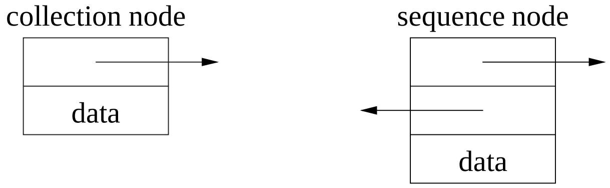
Appendix C

Data Structure Library (DSL)

*μ*C++ makes use of several basic data structures to manage objects in its runtime environment: stack, queue and sequence. Since these data structures are needed at compile time because of inlining, it is possible to use them in a *μ*C++ application program. When appropriate, reusing code by an application programmer can save significant time and effort. However, be forewarned that the *μ*C++ DSL is only as extensive as needed to implement *μ*C++; it is not meant to be a complete data structure library (such as LEDA or the STL).

A data structure is defined to be a group of nodes, containing user data, organized into a particular format, with specific operations peculiar to that format. For all data structures in this library, it is the user's responsibility to create and delete all nodes. Because a node's existence is independent of the data structure that organizes it, all nodes are manipulated by address not value; hence, all data structure routines take and return pointers to nodes and not the nodes themselves.

Nodes are divided into two kinds: those with one link field, which form a collection, and those with two link fields, which form a sequence.



uStack and uQueue are collections and uSequence is a sequence. To get the appropriate link fields associated with a user node, it must be a public descendant of uColable or uSeqable, respectively, e.g.:

```
class stacknode : public uColable { . . . }
class queuenode : public uColable { . . . }
class seqnode : public uSeqable { . . . }
```

A node inheriting from uSeqable can be put in a collection data structure but not vice versa. Along with providing the appropriate link fields, the types uColable and uSeqable also provide one member routine:

```
bool listed() const;
```

which returns **true** if the node is an element of any collection or sequence and **false** otherwise.

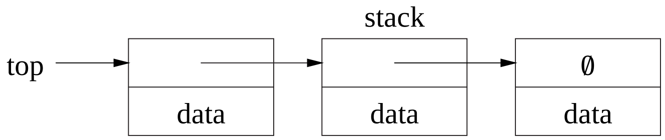
Finally, no header files are necessary to access the *μ*C++ DSL.

Some *μ*C++ DSL restrictions are:

- None of the member routines are virtual in any of the data structures for efficiency reasons. Therefore, pointers to data structures must be used with care or incorrect member routines may be invoked.

C.1 Stack

A uStack is a collection that defines an ordering among the nodes: nodes are returned by pop in the reverse order that they are added by push.



```
template<typename T> class uStack {
  public:
    uStack();
    bool empty() const;
    T *head() const
    T *top() const;
    void addHead( T *n );
    void add( T *n );
    void push( T *n );
    T *drop();
    T *pop();
};
```

T must be a public descendant of uColable.

The member routine empty returns **true** if the stack has no nodes and **false** otherwise. The member routine head returns a pointer to the top node of the stack without removing it or NULL if the stack has no nodes. The member routine top is a synonym for head. The member routine addHead adds a node to the top of the stack. The member routine add is a synonym for addHead. The member routine push is a synonym for addHead. The member routine drop removes a node from the top of the stack and returns a pointer to it or NULL if the stack has no nodes. The member routine pop is a synonym for drop.

C.1.1 Iterator

The iterator uStackIter<T> generates a stream of the elements of a uStack<T>.

```
template<typename T> class uStackIter {
  public:
    uStackIter();
    uStackIter( const uStack<T> &s );
    void over( const uStack<T> &s );
    bool operator>>( T *&tp );
};
```

It is used to iterate over the nodes of a stack from the top of the stack to the bottom.

The overloaded constructor routine uStackIter has the following forms:

uStackIter() – creates an iterator without associating it with a particular stack; the association must be done subsequently with member over.

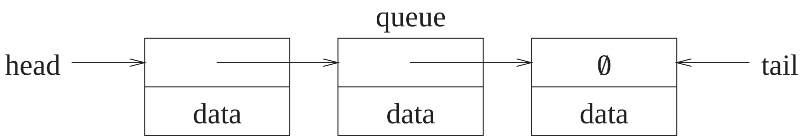
uStackIter(**const** uStack<T> &s) – creates an iterator and associates it the specified stack; the association can be changed subsequently with member over.

The member routine over resets the iterator to the top of the specified stack. The member routine >> attempts to move the iterator's internal cursor to the next node. If the bottom (end) of the stack 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.

Figure C.1 illustrates creating and using a stack and stack iterator.

C.2 Queue

A uQueue is a collection that defines an ordering among the nodes: nodes are returned by drop in the same order that they are added by add.



```
struct stackNode : public uColable {
    int v;
    stackNode( int v ) : v( v ) {}
};
void uMain::main() {
    uStack<stackNode> stack;
    uStackIter<stackNode> stackiter;
    stackNode *sp;
    int i;

    for ( i = 0; i < 10; i += 1 ) {                    // fill stack
        stack.push( new stackNode( 2 * i ) );
    }
    for ( stackiter.over(stack); stackiter >> sp; ) {  // print stack
        cout << sp->v << " ";
    }
    cout << endl;
    for ( i = 0; i < 10; i += 1 ) {                    // empty stack
        sp = stack.pop();
        delete sp;
    }
}
```

Figure C.1: DSL Stack

```
template<typename T> class uQueue {
  public:
    uQueue();
    bool empty() const;
    T *head() const
    T *tail() const;
    T *succ( T *n ) const;
    void addHead( T *n );
    void addTail( T *n );
    void add( T *n );
    T *dropHead();
    T *drop();
    T *dropTail();
    void remove( T *n );
    void transfer( uQueue<T> &from );
};
```

T must be a public descendant of uColable.

The member routine empty returns **true** if the queue has no nodes and **false** otherwise. The member routine head returns a pointer to the head or first node of the queue without removing it or NULL if the queue has no nodes. The member routine tail returns a pointer to the tail or last node of the queue without removing it. The member routine succ returns a pointer to the successor node after the specified node (toward the tail) or NULL if the specified node is the last node in the sequence. The member routine addHead adds a node to the head or front of the queue. The member routine addTail adds a node to the tail or end of the queue. The member routine add is a synonym for addTail. The member routine dropHead removes a node from the head or front of the queue and returns a pointer to it or NULL if the queue has no nodes. The member routine drop is a synonym for dropHead. The member routine dropTail removes a node from the tail or end of the queue and returns a pointer to it or NULL if the queue has no nodes. The member routine remove removes the specified node from the queue (any location). The member routine transfer transfers all nodes from the “from” list to the tail or end of the specified queue; the “from” list is empty after the transfer.

C.2.1 Iterator

The iterator uQueueIter<T> generates a stream of the elements of a uQueue<T>.

```
struct queueNode : public uColable {
    int v;
    queueNode( int v ) : v( v ) {}
};
void uMain::main() {
    uQueue<queueNode> queue;
    uQueueIter<queueNode> queueiter;
    queueNode *qp;
    int i;

    for ( i = 0; i < 10; i += 1 ) {                     // fill queue
        queue.add( new queueNode( 2 * i ) );
    }
    for ( queueiter.over(queue); queueiter >> qp; ) {    // print queue
        cout << qp->v << " ";
    }
    cout << endl;
    for ( i = 0; i < 10; i += 1 ) {                     // empty queue
        qp = queue.drop();
        delete qp;
    }
}
```

Figure C.2: DSL Queue

```
template<typename T> class uQueueIter {
  public:
    uQueueIter();
    uQueueIter( const uQueue<T> &q );
    void over( const uQueue<T> &q );
    bool operator>>( T *&tp );
};
```

It is used to iterate over the nodes of a queue from the head of the queue to the tail.

The overloaded constructor routine uQueueIter has the following forms:

uQueueIter() – creates an iterator without associating it with a particular queue; the association must be done subsequently with member over.

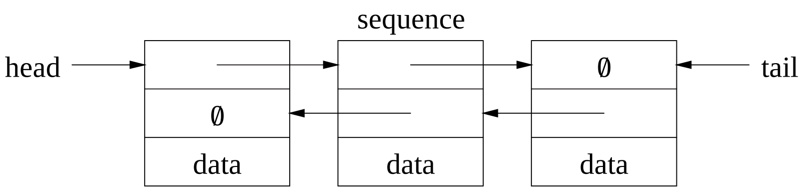
uQueueIter(**const** uQueue<T> &q) – creates an iterator and associates it the specified queue; the association can be changed subsequently with member over.

The member routine over resets the iterator to the head of the specified queue. The member routine >> attempts to move the iterator's internal cursor to the next node. If the tail (end) of the queue 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.

Figure C.2 illustrates creating and using a queue and queue iterator.

C.3 Sequence

A uSequence is a collection that defines a bidirectional ordering among the nodes: nodes can be added and removed from either end of the collection; furthermore, nodes can be inserted and removed anywhere in the collection.



```
template<typename T> class uSequence {
  public:
    uSequence();
    bool empty() const;
    T *head() const
    T *tail() const;
    T *succ( T *n ) const;
    T *pred( T *n ) const;
    void insertBef( T *n, T *bef );
    void insertAft( T *aft, T *n );
    void addHead( T* n );
    void addTail( T* n );
    void add( T* n );
    T *dropHead();
    T *drop();
    T *dropTail();
    void remove( T *n );
    void transfer( uSequence<T> &from );
};
```

T must be a public descendant of uSeqable.

The member routine empty returns **true** if the sequence has no nodes and **false** otherwise. The member routine head returns a pointer to the head or first node of the sequence without removing it or NULL if the sequence has no nodes. The member routine tail returns a pointer to the tail or last node of the sequence without removing it or NULL if the sequence has no nodes. The member routine succ returns a pointer to the successor node after the specified node (toward the tail) or NULL if the specified node is the last node in the sequence. The member routine pred returns a pointer to the predecessor node before the specified node (toward the head) or NULL if the specified node is the first node in the sequence. The member routine insertBef adds a node before the specified node or at the end (tail) if bef is NULL. The member routine insertAft adds a node after the specified node or at the beginning (head) if aft is NULL. The member routine addHead adds a node to the head or front of the sequence. The member routine addTail adds a node to the tail or end of the sequence. The member routine add is a synonym for addTail. The member routine dropHead removes a node from the head or front of the sequence and returns a pointer to it or NULL if the sequence has no nodes. The member routine drop is a synonym for dropHead. The member routine dropTail removes a node from the tail or end of the sequence and returns a pointer to it or NULL if the sequence has no nodes. The member routine remove removes the specified node from the sequence (any location). The member routine transfer transfers all nodes from the "from" list to the tail or end of the specified sequence; the "from" list is empty after the transfer.

A sequence behaves like a queue when members add and drop are used. The example program in Section D.3, p. 161 makes use of a sequence and modifies it so that nodes are maintained in order.

C.3.1 Iterator

The iterator uSeqIter<T> generates a stream of the elements of a uSequence<T>.

```
template<typename T> class uSeqIter {
  public:
    uSeqIter();
    uSeqIter( const uSequence<T> &s );
    void over( const uSequence<T> &s );
    bool operator>>( T *&tp );
};
```

It is used to iterate over the nodes of a sequence from the head of the sequence to the tail.

The iterator uSeqIterRev<T> generates a stream of the elements of a uSequence<T>.

```
template<typename T> class uSeqIterRev {
  public:
    uSeqIterRev();
    uSeqIterRev( const uSequence<T> &s );
    void over( const uSequence<T> &s );
    bool operator>>( T *&tp );
};
```

It is used to iterate over the nodes of a sequence from the tail of the sequence to the head.

```
struct seqNode : public uSeqable {
    int v;
    seqNode( int v ) : v( v ) {}
};
void uMain::main() {
    uSequence<seqNode> seq;
    uSeqIter<seqNode> seqiter;
    seqNode *sp;
    int i;

    for ( i = 0; i < 10; i += 1 ) {                          // fill sequence
        seq.add( new seqNode( 2 * i ) );
    }
    for ( seqiter.over(seq); seqiter >> sp; ) {              // print sequence forward
        cout << sp->v << " ";
    }
    cout << endl;
    for ( uSeqIterRev<seqNode> seqiterrev(seq); seqiterrev >> sp; ) { // print sequence reverse
        cout << sp->v << " ";
    }
    cout << endl;
    for ( seqiter.over(seq); seqiter >> sp; ) {              // empty sequence
        seq.remove( sp );                                    // can remove nodes during iteration
        delete sp;
    }
}
```

Figure C.3: DSL Sequence

The overloaded constructor routine uSeqIter has the following forms:

uSeqIter() – creates an iterator without associating it with a particular sequence; the association must be done subsequently with member over.

uSeqIter(**const** uSeq<T> &q) – creates an iterator and associates it the specified sequence; the association can be changed subsequently with member over.

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 depending on which iterator is used, 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.

Figure C.3 illustrates creating and using a sequence and sequence iterator.

Appendix D

Example Programs

D.1 Readers And Writer

The readers and writer problem deals with controlling access to a resource that can be shared by multiple readers, but only one writer can use it at a time (e.g., a sequential file). While there are many possible solutions to this problem, each solution must deal with unbounded waiting of reader and/or writer tasks if a continuous stream of one kind of task is arriving at the monitor. For example, if readers are currently using the resource, a continuous stream of reader tasks should not make an arriving writer task wait forever. Furthermore, a solution to the readers and writer problem should provide FIFO execution of the tasks so that a read that is requested after a write does not execute before the write, thus reading old information. This phenomenon is called the **stale readers** problem. Hoare gave a monitor solution in [Hoa74] that has a bounded on waiting but non-FIFO execution.

```
//                          -*- Mode: C++ -*-
//
// uC++ Version 6.1.0, Copyright (C) Peter A. Buhr 1994
//
// RWEx1.cc -- Readers and Writer Problem
//
// Author           : Peter A. Buhr
// Created On       : Thu Aug  2 11:51:34 1990
// Last Modified By : Peter A. Buhr
// Last Modified On : Fri Apr 29 12:10:06 2011
// Update Count     : 102
//

#include <iostream>
using std::cout;
using std::osacquire;
using std::endl;

_Monitor ReadersWriter {
    int rcnt, wcnt;
    uCondition RWers;
    enum RW { READER, WRITER };
  public:
    ReadersWriter() : rcnt(0), wcnt(0) {}

    void startRead() {
        if ( wcnt !=0 || ! RWers.empty() ) RWers.wait( READER );
        rcnt += 1;
        if ( ! RWers.empty() && RWers.front() == READER ) RWers.signal();
    } // ReadersWriter::startRead

    void endRead() {
        rcnt -= 1;
        if ( rcnt == 0 ) RWers.signal();
    } // ReadersWriter::endRead

    void startWrite() {
        if ( wcnt != 0 || rcnt != 0 ) RWers.wait( WRITER );
        wcnt = 1;
    } // ReadersWriter::startWrite

    void endWrite() {
        wcnt = 0;
        RWers.signal();
    } // ReadersWriter::endWrite
}; // ReadersWriter


volatile int SharedVar = 0;                                 // shared variable to test readers and writers

_Task Worker {
    ReadersWriter &rw;

    void main() {
        yield( rand() % 100 );                              // don′t all start at the same time
        if ( rand() % 100 < 70 ) {                          // decide to be a reader or writer
            rw.startRead();
            osacquire( cout ) << "Reader:" << this << ", shared:" << SharedVar << endl;
            yield( 3 );
            rw.endRead();
        } else {
            rw.startWrite();
            SharedVar += 1;
            osacquire( cout ) << "Writer:" << this << ",  wrote:" << SharedVar << endl;
            yield( 1 );
            rw.endWrite();
        } // if
    } // Worker::main
  public:
    Worker( ReadersWriter &rw ) : rw( rw ) {
    } // Worker::Worker
}; // Worker


void uMain::main() {
    enum { MaxTask = 50 };
    ReadersWriter rw;
    Worker *workers[MaxTask];

    for ( int i = 0; i < MaxTask; i += 1 ) {
        workers[i] = new Worker( rw );
    } // for
    for ( int i = 0; i < MaxTask; i += 1 ) {
        delete workers[i];
    } // for

    osacquire( cout ) << "successful completion" << endl;
} // uMain::main


// Local Variables: //
// tab-width: 4 //
// compile-command: "u++ RWEx1.cc" //
// End: //
```

D.2 Bounded Buffer

Two processes communicate through a unidirectional queue of finite length.

D.2.1 Using Monitor Accept

```
//                              -*- Mode: C++ -*-
//
// uC++ Version 6.1.0, Copyright (C) Peter A. Buhr 1994
//
// MonAcceptBB.cc -- Generic bounded buffer problem using a monitor and uAccept
//
// Author           : Peter A. Buhr
// Created On       : Thu Aug  2 11:35:05 1990
// Last Modified By : Peter A. Buhr
// Last Modified On : Wed Nov 30 08:41:30 2005
// Update Count     : 124
//


template<typename ELEMTYPE> _Monitor BoundedBuffer {
    const int size;                         // number of buffer elements
    int front, back;                        // position of front and back of queue
    int count;                              // number of used elements in the queue
    ELEMTYPE *Elements;
  public:
    BoundedBuffer( const int size = 10 ) : size( size ) {
        front = back = count = 0;
        Elements = new ELEMTYPE[size];
    } // BoundedBuffer::BoundedBuffer

    ~BoundedBuffer() {
        delete [] Elements;
    } // BoundedBuffer::~BoundedBuffer

    _Nomutex int query() {
        return count;
    } // BoundedBuffer::query

    void insert( ELEMTYPE elem );
    ELEMTYPE remove();
}; // BoundedBuffer

template<typename ELEMTYPE> inline void BoundedBuffer<ELEMTYPE>::insert( ELEMTYPE elem ) {
    if ( count == size ) {                  // buffer full ?
        _Accept( remove );                  // only allow removals
    } // if

    Elements[back] = elem;
    back = ( back + 1 ) % size;
    count += 1;
} // BoundedBuffer::insert

template<typename ELEMTYPE> inline ELEMTYPE BoundedBuffer<ELEMTYPE>::remove() {
    ELEMTYPE elem;
    if ( count == 0 ) {                              // buffer empty ?
        _Accept( insert );                           // only allow insertions
    } // if

    elem = Elements[front];
    front = ( front + 1 ) % size;
    count -= 1;

    return elem;
} // BoundedBuffer::remove

#include "ProdConsDriver.i"

// Local Variables: //
// tab-width: 4 //
// compile-command: "u++ MonAcceptBB.cc" //
// End: //
```

D.2.2 Using Monitor Condition

```
//                          -*- Mode: C++ -*-
//
// uC++ Version 6.1.0, Copyright (C) Peter A. Buhr 1994
//
// MonConditionBB.cc -- Generic bounded buffer problem using a monitor and condition variables
//
// Author           : Peter A. Buhr
// Created On       : Thu Aug  2 11:35:05 1990
// Last Modified By : Peter A. Buhr
// Last Modified On : Wed Nov 30 08:41:43 2005
// Update Count     : 57
//


template<typename ELEMTYPE> _Monitor BoundedBuffer {
    const int size;                                  // number of buffer elements
    int front, back;                                 // position of front and back of queue
    int count;                                       // number of used elements in the queue
    ELEMTYPE *Elements;
    uCondition BufFull, BufEmpty;
  public:
    BoundedBuffer( const int size = 10 ) : size( size ) {
        front = back = count = 0;
        Elements = new ELEMTYPE[size];
    } // BoundedBuffer::BoundedBuffer

    ~BoundedBuffer() {
        delete [] Elements;
    } // BoundedBuffer::~BoundedBuffer

    _Nomutex int query() {
        return count;
    } // BoundedBuffer::query

    void insert( ELEMTYPE elem ) {
        if ( count == size ) {
            BufFull.wait();
        } // if
        Elements[back] = elem;
        back = ( back + 1 ) % size;
        count += 1;

        BufEmpty.signal();
    }; // BoundedBuffer::insert

    ELEMTYPE remove() {
        ELEMTYPE elem;

        if ( count == 0 ) {
            BufEmpty.wait();
        } // if

        elem = Elements[front];
        front = ( front + 1 ) % size;
        count -= 1;

        BufFull.signal();
        return elem;
    }; // BoundedBuffer::remove
}; // BoundedBuffer

#include "ProdConsDriver.i"

// Local Variables: //
// tab-width: 4 //
// compile-command: "u++ MonConditionBB.cc" //
// End: //
```

D.2.3 Using Task

```
//                          -*- Mode: C++ -*-
//
// uC++ Version 6.1.0, Copyright (C) Peter A. Buhr 1994
//
// TaskAcceptBB.cc -- Generic bounded buffer using a task
//
// Author           : Peter A. Buhr
// Created On       : Sun Sep 15 20:24:44 1991
// Last Modified By : Peter A. Buhr
// Last Modified On : Sun Jul 31 18:50:16 2005
// Update Count     : 74
//


template<typename ELEMTYPE> _Task BoundedBuffer {
    const int size;                                     // number of buffer elements
    int front, back;                                    // position of front and back of queue
    int count;                                          // number of used elements in the queue
    ELEMTYPE *Elements;
  public:
    BoundedBuffer( const int size = 10 ) : size( size ) {
        front = back = count = 0;
        Elements = new ELEMTYPE[size];
    } // BoundedBuffer::BoundedBuffer

    ~BoundedBuffer() {
        delete [ ] Elements;
    } // BoundedBuffer::~BoundedBuffer

    _Nomutex int query() {
        return count;
    } // BoundedBuffer::query

    void insert( ELEMTYPE elem ) {
        Elements[back] = elem;
    } // BoundedBuffer::insert

    ELEMTYPE remove() {
        return Elements[front];
    } // BoundedBuffer::remove
  protected:
    void main() {
        for ( ;; ) {
            _Accept( ~BoundedBuffer )
                break;
            or _When ( count != size ) _Accept( insert ) {
                back = ( back + 1 ) % size;
                count += 1;
            } or _When ( count != 0 ) _Accept( remove ) {
                front = ( front + 1 ) % size;
                count -= 1;
            } // _Accept
        } // for
    } // BoundedBuffer::main
}; // BoundedBuffer

#include "ProdConsDriver.i"

// Local Variables: //
// tab-width: 4 //
// compile-command: "u++ TaskAcceptBB.cc" //
// End: //
```

D.2.4 Using P/V

```
//                              -*- Mode: C++ -*-
//
// uC++ Version 6.1.0, Copyright (C) Peter A. Buhr 1994
//
// SemaphoreBB.cc --
//
// Author              : Peter A. Buhr
// Created On          : Thu Aug 15 16:42:42 1991
// Last Modified By    : Peter A. Buhr
// Last Modified On    : Mon May 11 23:09:51 2015
// Update Count        : 55
//

#include <uSemaphore.h>

template<typename ELEMTYPE> class BoundedBuffer {
    const int size;                                 // number of buffer elements
    int front, back;                                // position of front and back of queue
    uSemaphore full, empty;                         // synchronize for full and empty BoundedBuffer
    uSemaphore ilock, rlock;                        // insertion and removal locks
    ELEMTYPE *Elements;

    BoundedBuffer( BoundedBuffer & );                       // no copy
    BoundedBuffer &operator=( BoundedBuffer & );            // no assignment
  public:
    BoundedBuffer( const unsigned int size = 10 ) : size( size ), full( 0 ), empty( size ) {
        front = back = 0;
        Elements = new ELEMTYPE[size];
    } // BoundedBuffer::BoundedBuffer

    ~BoundedBuffer() {
        delete  Elements;
    } // BoundedBuffer::~BoundedBuffer

    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 space
    } // BoundedBuffer::insert

    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 space
        return elem;
    } // BoundedBuffer::remove
}; // BoundedBuffer

#include "ProdConsDriver.i"

// Local Variables: //
// tab-width: 4 //
// compile-command: “u++ SemaphoreBB.cc” //
// End: //
```

D.3 Disk Scheduler

The following example illustrates a fully implemented disk scheduler. The disk scheduling algorithm is the elevator algorithm, which services all the requests in one direction and then reverses direction. A linked list is used to store incoming requests while the disk is busy servicing a particular request. The nodes of the list are stored on the stack of the calling processes so that suspending a request does not consume resources. The list is maintained in sorted order by track number and there is a pointer which scans backward and forward through the list. New requests can be added both before and after the scan pointer while the disk is busy. If new requests are added before the scan pointer in the direction of travel, they are serviced on that scan.

The disk calls the scheduler to get the next request that it services. This call does two things: it passes to the scheduler the status of the just completed disk request, which is then returned from scheduler to disk user, and it returns the information for the next disk operation. When a user’s request is accepted, the parameter values from the request are copied into a list node, which is linked in sorted order into the list of pending requests. The disk removes work from the list of requests and stores the current request it is performing in CurrentRequest. When the disk has completed a request, the request's status is placed in the CurrentRequest node and the user corresponding to this request is reactivated.

```
//                              -*- Mode: C++ -*-
//
// uC++ Version 6.1.0, Copyright (C) Peter A. Buhr 1994
//
// LOOK.cc -- Look Disk Scheduling Algorithm
//
// The LOOK disk scheduling algorithm causes the disk arm to sweep bidirectionally across the disk surface until there
// are no more requests in that particular direction, servicing all requests in its path.
//
// Author              : Peter A. Buhr
// Created On          : Thu Aug 29 21:46:11 1991
// Last Modified By : Peter A. Buhr
// Last Modified On : Sun Jul 18 11:06:04 2010
// Update Count        : 281
//

#include <iostream>
using std::cout;
using std::osacquire;
using std::endl;

typedef char Buffer[50];                                  // dummy data buffer

const int NoOfCylinders = 100;
enum IOStatus { IO_COMPLETE, IO_ERROR };

class IORequest {
  public:
    int track;
    int sector;
    Buffer *bufadr;
    IORequest() {}
    IORequest( int track, int sector, Buffer *bufadr ) {
        IORequest::track = track;
        IORequest::sector = sector;
        IORequest::bufadr = bufadr;
    } // IORequest::IORequest
}; // IORequest

class WaitingRequest : public uSeqable {                  // element for a waiting request list
    WaitingRequest( WaitingRequest & );                   // no copy
    WaitingRequest &operator=( WaitingRequest & );        // no assignment
  public:
    uCondition block;
    IOStatus status;
    IORequest req;
    WaitingRequest( IORequest req ) {
        WaitingRequest::req = req;
    }
}; // WaitingRequest

class Elevator : public uSequence<WaitingRequest> {
    int Direction;
    WaitingRequest *Current;
    Elevator( Elevator & );                                 // no copy
    Elevator &operator=( Elevator & );                      // no assignment
  public:
    Elevator() {
        Direction = 1;
    } // Elevator::Elevator

    void orderedInsert( WaitingRequest *np ) {
        WaitingRequest *lp;
        for ( lp = head();                                  // insert in ascending order by track number
            lp != 0 && lp->req.track < np->req.track;
            lp = succ( lp ) );
        if ( empty() ) Current = np;                        // 1st client, so set Current
        insertBef( np, lp );
    } // Elevator::orderedInsert

    WaitingRequest *remove() {
        WaitingRequest *temp = Current;                     // advance to next waiting client
        Current = Direction ? succ( Current ) : pred( Current );
        uSequence<WaitingRequest>::remove( temp );          // remove request

        if ( Current == 0 ) {                               // reverse direction ?
            osacquire( cout ) << "Turning" << endl;
            Direction = !Direction;
            Current = Direction ? head() : tail();
        } // if
        return temp;
    } // Elevator::remove
}; // Elevator

_Task DiskScheduler;

_Task Disk {
    DiskScheduler &scheduler;
    void main();
  public:
    Disk( DiskScheduler &scheduler ) : scheduler( scheduler ) {
    } // Disk
}; // Disk

_Task DiskScheduler {
    Elevator PendingClients;                                // ordered list of client requests
    uCondition DiskWaiting;                                 // disk waits here if no work
    WaitingRequest *CurrentRequest;                         // request being serviced by disk
    Disk disk;                                              // start the disk
    IORequest req;
    WaitingRequest diskterm;                                // preallocate disk termination request

    void main();
  public:
    DiskScheduler() : disk( *this ), req( -1, 0, 0 ), diskterm( req ) {
    } // DiskScheduler
    IORequest WorkRequest( IOStatus );
    IOStatus DiskRequest( IORequest & );
}; // DiskScheduler

_Task DiskClient {
    DiskScheduler &scheduler;
    void main();
  public:
    DiskClient( DiskScheduler &scheduler ) : scheduler( scheduler ) {
    } // DiskClient
}; // DiskClient

void Disk::main() {
    IOStatus status;
    IORequest work;

    status = IO_COMPLETE;
    for ( ;; ) {
        work = scheduler.WorkRequest( status );
      if ( work.track == -1 ) break;
        osacquire( cout ) << "Disk main, track:" << work.track << endl;
        yield( 100 );                                       // pretend to perform an I/O operation
        status = IO_COMPLETE;
    } // for
} // Disk::main

void DiskScheduler::main() {
    uSeqIter<WaitingRequest> iter;                          // declared here because of gcc compiler bug

    CurrentRequest = NULL;                                  // no current request at start
    for ( ;; ) {
        _Accept( ~DiskScheduler ) {                         // request from system
            break;
        } or _Accept( WorkRequest ) {                       // request from disk
        } or _Accept( DiskRequest ) {                       // request from clients
        } // _Accept
    } // for

    // two alternatives for terminating scheduling server
#if 0
    for ( ; ! PendingClients.empty(); ) {                   // service pending disk requests before terminating
        _Accept( WorkRequest );
    } // for
#else
    WaitingRequest *client;                                 // cancel pending disk requests before terminating

    for ( iter.over(PendingClients); iter >> client; ) {
        PendingClients.remove();                            // remove each client from the list
        client->status = IO_ERROR;                          // set failure status
        client->block.signal();                             // restart client
    } // for
#endif
    // pending client list is now empty

    // stop disk
    PendingClients.orderedInsert( &diskterm );              // insert disk terminate request on list

    if ( ! DiskWaiting.empty() ) {                          // disk free ?
        DiskWaiting.signal();                               // wake up disk to deal with termination request
    } else {
        _Accept( WorkRequest );                             // wait for current disk operation to complete
    } // if
} // DiskScheduler::main

IOStatus DiskScheduler::DiskRequest( IORequest &req ) {
    WaitingRequest np( req );                               // preallocate waiting list element

    PendingClients.orderedInsert( &np );                    // insert in ascending order by track number
    if ( ! DiskWaiting.empty() ) {                          // disk free ?
        DiskWaiting.signal();                               // reactivate disk
    } // if

    np.block.wait();                                        // wait until request is serviced

    return np.status;                                       // return status of disk request
} // DiskScheduler::DiskRequest

IORequest DiskScheduler::WorkRequest( IOStatus status ) {
    if ( CurrentRequest != NULL ) {                         // client waiting for request to complete ?
        CurrentRequest->status = status;                    // set request status
        CurrentRequest->block.ignal();                      // reactivate waiting client
    } // if

    if ( PendingClients.empty() ) {                         // any clients waiting ?
        DiskWaiting.wait();                                 // wait for client to arrive
    } // if

    CurrentRequest = PendingClients.remove();               // remove next client′s request
    return CurrentRequest->req;                             // return work for disk
} // DiskScheduler::WorkRequest

void DiskClient::main() {
    IOStatus status;
    IORequest req( rand() % NoOfCylinders, 0, 0 );

    yield( rand() % 100 );                                  // don′t all start at the same time
    osacquire( cout ) << "enter DiskClient main seeking:" << req.track << endl;
    status = scheduler.DiskRequest( req );
    osacquire( cout ) << "enter DiskClient main seeked to:" << req.track << endl;
} // DiskClient::main

void uMain::main() {
    const int NoOfTests = 20;
    DiskScheduler scheduler;                                // start the disk scheduler
    DiskClient *p;

    srand( getpid() );                                      // initialize random number generator

    p = new DiskClient[NoOfTests]( scheduler );             // start the clients
    delete [] p;                                            // wait for clients to complete

    cout << "successful completion" << endl;
} // uMain::main

// Local Variables: //
// tab-width: 4 //
// compile-command: "u++ LOOK.cc" //
// End: //
```

D.4 UNIX File I/O

The following example program reads in a file and copies it into another file.

```
//                          -*- Mode: C++ -*-
```

```
//
// uC++ Version 6.1.0, Copyright (C) Peter A. Buhr 1994
//
// File.cc -- Print multiple copies of the same file to standard output
//
// Author               : Peter A. Buhr
// Created On           : Tue Jan  7 08:44:56 1992
// Last Modified By     : Peter A. Buhr
// Last Modified On     : Thu Dec  8 17:43:07 2011
// Update Count         : 44
//

#include <uFile.h>
#include <iostream>
using std::cout;
using std::cerr;
using std::endl;

_Task Copier {
    uFile &input;

    void main() {
        uFile::FileAccess in( input, O_RDONLY );
        int count;
        char buf[1];

        for ( int i = 0;; i += 1 ) {                    // copy in-file to out-file
            count = in.read( buf, sizeof( buf ) );
          if ( count == 0 ) break;                      // eof ?
            cout << buf[0];
            if ( i % 20 == 0 ) yield();
        } // for
    } // Copier::main
  public:
    Copier( uFile &in ) : input( in ) {
    } // Copier::Copier
}; // Copier

void uMain::main() {
    switch ( argc ) {
      case 2:
        break;
      default:
        cerr << "Usage: " << argv[0] << " input-file" << std::endl;
        exit( EXIT_FAILURE );
    } // switch

    uFile input( argv[1] );                             // connect with UNIX files
    {
        Copier c1( input ), c2( input );
    }
} // uMain::main

// Local Variables: //
// tab-width: 4 //
// compile-command: “u++ File.cc” //
// End: //
```

D.5 UNIX Socket I/O

The following example illustrates bidirectional communication between a client and server socket. A client starts a task to read from standard input and write the data to a server socket. The server or its acceptor for that client, reads the data from the client and writes it directly back to the client. The client also starts a task that reads the data coming back from the server or its acceptor and writes it onto standard output. Hence, a file is read from standard input and written onto standard output after having made a loop through a server. The server can deal with multiple simultaneous clients.

D.5.1 Client - UNIX/Datagram

```
//                                      -*- Mode: C++ -*-
//
// uC++ Version 6.1.0, Copyright (C) Peter A. Buhr 1999
//
// ClientUNIXDGRAM.cc -- Client for UNIX/datagram socket test. Client reads from standard input, writes the data to the
//      server, reads the data from the server, and writes that data to standard output.
//
// Author              : Peter A. Buhr
// Created On          : Thu Apr 29 16:05:12 1999
// Last Modified By : Peter A. Buhr
// Last Modified On : Sat Nov 24 15:44:15 2012
// Update Count        : 49
//

#include <uSemaphore.h>
#include <uSocket.h>
#include <iostream>
using std::cin;
using std::cout;
using std::cerr;
using std::osacquire;
using std::endl;

// minimum buffer size is 2, 1 character and string terminator, '\0'
enum { BufferSize = 65 };
const char EOD = '\377';
unsigned int rcnt = 0, wcnt = 0;

// Datagram sockets are lossy (i.e., drop packets). To prevent clients from flooding the server with packets, resulting
// in dropped packets, a semaphore is used to synchronize the reader and writer tasks so at most N writes occur before a
// read. As well, if the buffer size is increased substantially, it may be necessary to decrease N to ensure the server
// buffer does not fill.

enum { MaxWriteBeforeRead = 5 };
uSemaphore readSync( MaxWriteBeforeRead );

_Task Reader {
    uSocketClient &client;

    void main() {
        uDuration timeout( 20, 0 );                              // timeout for read
        char buf[BufferSize];
        int len;
        //struct sockaddr_un from;
        //socklen_t fromlen = sizeof( from );

        try {
            for ( ;; ) {
                len = client.recvfrom( buf, sizeof(buf), 0, &timeout );
                //len = client.recvfrom( buf, sizeof(buf), (sockaddr *)&from, &fromlen, 0, &timeout );
                readSync.V();
                // osacquire( cerr ) << “reader read len:” << len << endl;
              if ( len == 0 ) uAbort( "client %d : EOF ecountered without EOD", getpid() );
                rcnt += len;
                // The EOD character can be piggy-backed onto the end of the message.
              if ( buf[len - 1] == EOD ) {
                    rcnt -= 1;                              // do not count the EOD
                    cout.write( buf, len - 1 );             // do not write the EOD
                    break;
                } // exit
                cout.write( buf, len );
            } // for
        } catch( uSocketServer::ReadTimeout ) {
            cout << "Warning: client timeout, possible cause lost data on datagram socket" << endl;
        } // try
    } // Reader::main
  public:
    Reader( uSocketClient &client ) : client ( client ) {
    } // Reader::Reader
}; // Reader

_Task Writer {
    uSocketClient &client;

    void main() {
        char buf[BufferSize];
        //struct sockaddr_un to;
        //socklen_t tolen = sizeof( to );

        //client.getServer( (sockaddr *)&to, &tolen );
        for ( ;; ) {
            cin.get( buf, sizeof(buf), '\0' );              // leave room for string terminator
            int len = strlen( buf );
            // osacquire( cerr ) << “writer read len:” << len << endl;
          if ( buf[0] == '\0' ) break;
            wcnt += len;
            readSync.P();
            client.sendto( buf, len );
            //client.sendto( buf, len, (sockaddr *)&to, tolen );
        } // for
        readSync.P();
        client.sendto( const_cast<char *>(&EOD), sizeof(EOD) );
        //client.sendto( const_cast<char *>(&EOD), sizeof(EOD), (sockaddr *)&to, tolen );
    } // Writer::main
  public:
    Writer( uSocketClient &client ) : client( client ) {
    } // Writer::Writer
}; // Writer

void uMain::main() {
    switch ( argc ) {
      case 2:
        break;
      default:
        cerr << "Usage: " << argv[0] << " socket-name" << endl;
        exit( EXIT_FAILURE );
    } // switch
    // remove tie due to race between cin flush and cout write by writer and reader tasks
    cin.tie( NULL );
    uSocketClient client( argv[1], SOCK_DGRAM );              // connection to server
    {
        Reader rd( client );                                  // emit worker to read from server and write to output
        Writer wr( client );                                  // emit worker to read from input and write to server
    }
    if ( wcnt != rcnt ) {
        uAbort( "Error: client not all data transfered, wcnt:%d rcnt:%d", wcnt, rcnt );
    } // if
} // uMain::main

// Local Variables: //
// tab-width: 4 //
// compile-command: "u++-work ClientUNIXDGRAM.cc -o Client" //
// End: //
```

D.5.2 Server - UNIX/Datagram

```
//                                  -*- Mode: C++ -*-
//
// uC++ Version 6.1.0, Copyright (C) Peter A. Buhr 1999
//
// ServerUNIXDGRAM.cc -- Server for UNIX/datagram socket test. Server reads data from multiple clients. The server reads
//      the data from the client and writes it back.
//
// Author              : Peter A. Buhr
// Created On          : Fri Apr 30 16:36:18 1999
// Last Modified By : Peter A. Buhr
// Last Modified On : Sat Nov 24 14:24:34 2012
// Update Count        : 44
//

#include <uSocket.h>
#include <iostream>
#include <unistd.h>                                           // unlink
using std::cerr;
using std::osacquire;
using std::endl;

enum { BufferSize = 8 * 1024 };

_Task Reader {
    uSocketServer &server;

    void main() {
        uDuration timeout( 20, 0 );                           // timeout for read
        char buf[BufferSize];
        int len;
        //struct sockaddr_un to;
        //socklen_t tolen = sizeof( to );

        try {
            for ( ;; ) {
                len = server.recvfrom( buf, sizeof(buf), 0, &timeout );
                //len = server.recvfrom( buf, sizeof(buf), (sockaddr *)&to, &tolen, 0, &timeout );
                // osacquire( cerr ) << "reader read len:" << len << endl;
              if ( len == 0 ) uAbort( "server %d : EOF ecountered before timeout", getpid() );
```

```
                server.sendto( buf, len );                  // write byte back to client
                //server.sendto( buf, len, (sockaddr *)&to, tolen ); // write byte back to client
            } // for
        } catch( uSocketServer::ReadTimeout ) {
        } // try
    } // Reader::main
  public:
    Reader( uSocketServer &server ) : server( server ) {
    } // Reader::Reader
}; // Reader

void uMain::main() {
    switch ( argc ) {
      case 2:
        break;
      default:
        cerr << "Usage: " << argv[0] << " socket-name" << endl;
        exit( EXIT_FAILURE );
    } // switch

    uSocketServer server( argv[1], SOCK_DGRAM );            // create and bind a server socket
    {
        Reader rd( server );                                // execute until reader times out
    }
    unlink( argv[1] );                                      // remove socket file
} // uMain

// Local Variables: //
// tab-width: 4 //
// compile-command: "u++-work ServerUNIXDGRAM.cc -o Server" //
// End: //
```

D.5.3 Client - INET/Stream

```
//                              -*- Mode: C++ -*-
//
// uC++ Version 6.1.0, Copyright (C) Peter A. Buhr 1994
//
// ClientINETSTREAM.cc -- Client for INET/stream socket test. Client reads from standard input, writes the data to the
//     server, reads the data from the server, and writes that data to standard output.
//
// Author              : Peter A. Buhr
// Created On          : Tue Jan  7 08:42:32 1992
// Last Modified By    : Peter A. Buhr
// Last Modified On    : Thu Dec  8 17:41:00 2011
// Update Count        : 158
//

#include <uSocket.h>
#include <iostream>
using std::cin;
using std::cout;
using std::cerr;
using std::osacquire;
using std::endl;

// minimum buffer size is 2, 1 character and string terminator, '\0'
enum { BufferSize = 65 };
const char EOD = '\377';
const char EOT = '\376';
unsigned int rcnt = 0, wcnt = 0;

_Task Reader {
    uSocketClient &client;

    void main() {
        char buf[BufferSize];
        int len;

        for ( ;; ) {
            len = client.read( buf, sizeof(buf) );
            // osacquire( cerr ) << “reader read len:” << len << endl;
          if ( len == 0 ) uAbort( "client %d : EOF ecountered without EOD", getpid() );
            rcnt += len;
            // The EOD character can be piggy-backed onto the end of the message.
          if ( buf[len - 1] == EOD ) {
              rcnt -= 1;                                // do not count the EOD
              cout.write( buf, len - 1 );               // do not write the EOD
              client.write( &EOT, sizeof(EOT) );        // indicate EOD received
              break;
            } // exit
            cout.write( buf, len );
        } // for
    } // Reader::main
  public:
    Reader( uSocketClient &client ) : client ( client ) {
    } // Reader::Reader
}; // Reader

_Task Writer {
    uSocketClient &client;

    void main() {
        char buf[BufferSize];

        for ( ;; ) {
            cin.get( buf, sizeof(buf), '\0' );          // leave room for string terminator
          if ( buf[0] == '\0' ) break;
            int len = strlen( buf );
            // osacquire( cerr ) << “writer read len:” << len << endl;
            wcnt += len;
            client.write( buf, len );
        } // for
        client.write( &EOD, sizeof(EOD) );
    } // Writer::main
  public:
    Writer( uSocketClient &client ) : client( client ) {
    } // Writer::Writer
}; // Writer

void uMain::main() {
    switch ( argc ) {
      case 2:
        break;
      default:
        cerr << "Usage: " << argv[0] << " port-number" << endl;
        exit( EXIT_FAILURE );
    } // switch
    // remove tie due to race between cin flush and cout write by writer and reader tasks
    cin.tie( NULL );
    uSocketClient client( atoi( argv[1] ) );                // connection to server
    {
        Reader rd( client );                                // emit worker to read from server and write to output
        Writer wr( client );                                // emit worker to read from input and write to server
    }
    if ( wcnt != rcnt ) {
        uAbort( "Error: client not all data transfered, wcnt:%d rcnt:%d", wcnt, rcnt );
    } // if
} // uMain::main

// Local Variables: //
// tab-width: 4 //
// compile-command: "u++-work ClientINETSTREAM.cc -o Client" //
// End: //
```

D.5.4 Server - INET/Stream

```
//                              -*- Mode: C++ -*-
//
// uC++ Version 6.1.0, Copyright (C) Peter A. Buhr 1994
//
// ServerINETSTREAM.cc -- Server for INET/stream socket test. Server accepts multiple connections from clients. Each
//     client then communicates with an acceptor.  The acceptor reads the data from the client and writes it back.
//
// Author              : Peter A. Buhr
// Created On          : Tue Jan  7 08:40:22 1992
// Last Modified By : Peter A. Buhr
// Last Modified On : Thu Dec  8 17:48:55 2011
// Update Count        : 190
//

#include <uSocket.h>
#include <iostream>
using std::cout;
using std::cerr;
using std::osacquire;
using std::endl;

enum { BufferSize = 8 * 1024 };
const char EOD = '\377';
const char EOT = '\376';

_Task Server;                                               // forward declaration

_Task Acceptor {
    uSocketServer &sockserver;
    Server &server;

    void main();
  public:
    Acceptor( uSocketServer &socks, Server &server ) : sockserver( socks ), server( server ) {
    } // Acceptor::Acceptor
}; // Acceptor

_Task Server {
    uSocketServer &sockserver;
    Acceptor *terminate;
    int acceptorCnt;
    bool timeout;
  public:
    Server( uSocketServer &socks ) : sockserver( socks ), acceptorCnt( 1 ), timeout( false ) {
    } // Server::Server

    void connection() {
    } // Server::connection

    void complete( Acceptor *terminate, bool timeout ) {
        Server::terminate = terminate;
        Server::timeout = timeout;
    } // Server::complete
  private:
    void main() {
        new Acceptor( sockserver, *this );                  // create initial acceptor
        for ( ;; ) {
            _Accept( connection ) {
                new Acceptor( sockserver, *this );          // create new acceptor after a connection
                acceptorCnt += 1;
            } or _Accept( complete ) {                      // acceptor has completed with client
                delete terminate;                           // delete must appear here or deadlock
                acceptorCnt -= 1;
              if ( acceptorCnt == 0 ) break;                // if no outstanding connections, stop
                if ( timeout ) {
                    new Acceptor( sockserver, *this );      // create new acceptor after a timeout
                    acceptorCnt += 1;
                } // if
            } // _Accept
        } // for
    } // Server::main
}; // Server

void Acceptor::main() {
    try {
        uDuration timeout( 20, 0 );                         // timeout for accept
        uSocketAccept acceptor( sockserver, &timeout );     // accept a connection from a client
        char buf[BufferSize];
        int len;

        server.connection();                                // tell server about client connection
        for ( ;; ) {
            len = acceptor.read( buf, sizeof(buf) );        // read byte from client
            // osacquire( cerr ) << “Server::acceptor read len:” << len << endl;
          if ( len == 0 ) uAbort( "server %d : EOF ecountered without EOD", getpid() );
            acceptor.write( buf, len );                     // write byte back to client
            // The EOD character can be piggy-backed onto the end of the message.
          if ( buf[len - 1] == EOD ) break;                 // end of data ?
        } // for
        len = acceptor.read( buf, sizeof(buf) );            // read EOT from client
        if ( len != 1 && buf[0] != EOT ) {
            uAbort( "server %d : failed to read EOT", getpid() );
        } // if
        server.complete( this, false );                     // terminate
    } catch( uSocketAccept::OpenTimeout ) {
        server.complete( this, true );                      // terminate
    } // try
} // Acceptor::main
```

```
void uMain::main() {
    switch ( argc ) {
      case 1:
        break;
      default:
        cerr << "Usage: " << argv[0] << endl;
        exit( EXIT_FAILURE );
    } // switch

    short unsigned int port;
    uSocketServer sockserver( &port );              // create and bind a server socket to free port

    cout << port << endl;                           // print out free port for clients
    {
        Server s( sockserver );                     // execute until acceptor times out
    }
} // uMain

// Local Variables: //
// tab-width: 4 //
// compile-command: "u++-work ServerINETSTREAM.cc -o Server" //
// End: //
```

Bibliography

[AGMK94] B. Adelberg, H. Garcia-Molina, and B. Kao. Emulating Soft Real-Time Scheduling Using Traditional Operating System Schedulers. In *Proc. IEEE Real-Time Systems Symposium*, pages 292–298, 1994. 136

[Ale01] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, Boston, February 2001. 51

[Amd67] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. 47

[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. 26

[BD92] Peter A. Buhr and Glen Ditchfield. Adding Concurrency to a Programming Language. In *USENIX C++ Technical Conference Proceedings*, pages 207–224, Portland, Oregon, U.S.A., August 1992. USENIX Association. 3

[BDS$^+$92] P. A. Buhr, Glen Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. *μ*C++: Concurrency in the Object-Oriented Language C++. *Softw. Pract. Exp.*, 22(2):137–172, February 1992. 143

[BDZ89] P. A. Buhr, Glen Ditchfield, and C. R. Zarnke. Adding Concurrency to a Statically Type-Safe Object-Oriented Programming Language. *SIGPLAN Not.*, 24(4):18–21, April 1989. Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Sept. 26–27, 1988, San Diego, California, U.S.A. 143

[BFC95] Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor Classification. *ACM Computing Surveys*, 27(1):63–107, March 1995. 21

[BLL88] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A System for Object-oriented Parallel Programming. *Softw. Pract. Exp.*, 18(8):713–732, August 1988. 5, 33

[BMZ92] Peter A. Buhr, Hamish I. Macdonald, and C. Robert Zarnke. Synchronous and Asynchronous Handling of Abnormal Events in the *μ*System. *Softw. Pract. Exp.*, 22(9):735–776, September 1992. 78, 81

[BP91] T. Baker and O. Pazy. Real-Time Features of Ada 9X. In *Proc. IEEE Real-Time Systems Symposium*, pages 172–180, 1991. 133

[Bri75] Per Brinch Hansen. The Programming Language Concurrent Pascal. *IEEE Trans. Softw. Eng.*, 2:199–206, June 1975. 3

[Buh85] P. A. Buhr. A Case for Teaching Multi-exit Loops to Beginning Programmers. *SIGPLAN Not.*, 20(11):14–22, November 1985. 11

[Buh95] Peter A. Buhr. Are Safe Concurrency Libraries Possible? *Communications of the ACM*, 38(2):117–120, February 1995. 3, 33

[But97] David R. Butenhof. *Programming with POSIX Threads*. Professional Computing. Addison-Wesley, Boston, 1997. 121

[BW90] Alan Burns and A. J. Wellings. The Notion of Priority in Real-Time Programming Languages. *Computer Language*, 15(3):153–162, 1990. 136

[C++98] International Standard ISO/IEC 14882:1998 (E), www.ansi.org. *Programming Languages – C++*, 1998. 145

[Car90] Tom 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. 36

[CD95] Tai M. Chung and Hank G. Dietz. Language Constructs and Transformation for Hard Real-time Systems. In *Proc. Second ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, June 1995. 127

[CG89] Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, April 1989. 3

[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. 34

[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. 33

[Dij65] Edsger W. Dijkstra. Cooperating Sequential Processes. Technical report, Technological University, Eindhoven, Netherlands, 1965. Reprinted in [Gen68] pp. 43–112. 36

[Geh92] N. H. Gehani. Exceptional C or C with Exceptions. *Softw. Pract. Exp.*, 22(10):827–848, October 1992. 78

[Gen68] F. Genuys, editor. *Programming Languages*. Academic Press, London, New York, 1968. NATO Advanced Study Institute, Villard-de-Lans, 1966. 176

[Gen81] W. Morven Gentleman. Message Passing between Sequential Processes: the Reply Primitive and the Administrator Concept. *Softw. Pract. Exp.*, 11(5):435–466, May 1981. 4, 55

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Reading, 2nd edition, 2000. 11, 33

[Gol94] David B. Golub. Operating System Support for Coexistence of Real-Time and Conventional Scheduling. Technical report, Carnegie Mellon University, November 1994. 136

[GR88] N. H. Gehani and W. D. Roome. Concurrent C++: Concurrent Programming with Class(es). *Softw. Pract. Exp.*, 18(12):1157–1177, December 1988. 7, 26

[GR91] N. Gehani and K. Ramamritham. Real-Time Concurrent C: A Language for Programming Dynamic Real-Time Systems. *Journal of Real-Time Systems*, 3(4):377–405, December 1991. 127

[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. 4

[HM92] W.A. Halang and K. Mangold. Real-Time Programming Languages. In Michael Schiebe and Saskia Pferrer, editors, *Real-Time Systems Engineering and Applications*, chapter 4, pages 141–200. Kluwer Academic Publishers, 1992. 127

[Hoa74] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974. 5, 155

[Hol92] R. C. Holt. *Turing Reference Manual*. Holt Software Associates Inc., 3rd edition, 1992. 3

[Int95] International Standard ISO/IEC. *Ada Reference Manual*, 6.0 edition, 1995. 81

[ITM90] Y. Ishikawa, H. Tokuda, and C.W. Mercer. Object-Oriented Real-Time Language Design: Constructs for Timing Constraints. In *Proc. ECOOP/OOPSLA*, pages 289–298, October 1990. 127

[KK91] K.B. Kenny and K.J.Lin. Building Flexible Real-Time Systems using the Flex Language. *IEEE Computer*, 24(5):70–78, May 1991. 127

[KS86] E. Klingerman and A.D. Stoyenko. Real-Time Euclid: A Language for Reliable Real-Time Systems. *IEEE Transactions on Software Engineering*, pages 941–949, September 1986. 127

[Lab90] Pierre Labrèche. Interactors: A Real-Time Executive with Multiparty Interactions in C++. *SIGPLAN Not.*, 25(4):20–32, April 1990. 33

[LN88] K.J. Lin and S. Natarajan. Expressing and Maintaining Timing Constratins in FLEX. In *Proc. IEEE Real-Time Systems Symposium*, pages 96–105, 1988. 127

[Mac77] M. Donald MacLaren. Exception Handling in PL/I. *SIGPLAN Not.*, 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. 78

[Mar78] T. Martin. Real-Time Programming Language PEARL – Concept and Characteristics. In *IEEE Computer Society 2nd International Computer Software and Applications Conference*, pages 301–306, 1978. 127

[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, New York, 1980. 5, 13

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall Object-Oriented Series. Prentice-Hall, Englewood Cliffs, 1992. 71

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-oriented Programming in the BETA Programming Language*. Addison-Wesley, Boston, 1993. 34

[MMS79] James G. Mitchell, William Maybury, and Richard Sweet. Mesa Language Manual. Technical Report CSL–79–3, Xerox Palo Alto Research Center, April 1979. 3, 80

[POS08] IEEE and The Open Group. *1003.1 Standard for Information Technology – Portable Operating System Interface (POSIX), Base Specifications, Issue 7*, 2008. 125

[RAA$^+$88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Hermann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Chorus Distributed Operating Systems. *Computing Systems*, 1(4):305–370, 1988. 137

[Raj91] Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991. 135

[RH87] A. Rizk and F. Halsall. Design and Implementation of a C-based Language for Distributed Real-time Systems. *SIGPLAN Notices*, 22(6):83–100, June 1987. 7

[Rip90] David Ripps. *An Implementaion Guide to Real-Time Programming*. Yourdon Press, 1990. 127

[RSL88] Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky. Real-Time Synchronization Protocols for Multiprocessors. In *Proc. IEEE Real-Time Systems Symposium*, pages 259–269, 1988. 135

[SBG$^+$90] Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, Daniel M. Yellin, and Shaula Alexander Yemini. Hermes: A Language for Distributed Computing. Technical report, IBM T. J. Watson Research Center, Yorktown Heights, New York, U.S.A., 10598, October 1990. 6

[SD92] A.E.K. Sahraoui and D. Delfieu. ZAMAN, A Simple Language for Expressing Timing Constraints. In *Real-Time Programming, IFAC Workshop*, pages 19–24, 1992. 127

[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. 33

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990. 135

[Sta87] Standardiseringskommissionen i Sverige. *Databehandling – Programspråk – SIMULA*, 1987. Svensk Standard SS 63 61 14. 33

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley Longman, 3rd edition, 1997. 1, 3

[Tie88] Michael D. Tiemann. Solving the RPC problem in GNU C++. In *Proceedings of the USENIX C++ Conference*, pages 343–361, Denver, Colorado, U.S.A, October 1988. USENIX Association. 34

[Tie90] Michael D. Tiemann. User's Guide to GNU C++. Free Software Foundation, 1000 Mass Ave., Cambridge, MA, U.S.A., 02138, March 1990. 10, 142

[TvRvS$^+$90] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(12):46–63, December 1990. 137

[Uni83] United States Department of Defense. *The Programming Language Ada: Reference Manual*, ansi/mil-std-1815a-1983 edition, February 1983. Springer, New York. 24

[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. 13

[Yok92] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proc. Object-Oriented Programming Systems, Languages, and Applications*, pages 414–434, 1992. 137

[You91] Brian M. Younger. Adding Concurrency to C++. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 1991. 143

Index

-U++ option, 11
-compiler option, 11
-debug option, 10
-multi option, 11
-nodebug option, 10
-nomulti option, 11
-noquiet option, 11
-noverify option, 10
-noyield option, 10
-quiet option, 11
-verify option, 10, 16
-yield option, 10, 32
<<, 59
>>, 59, 150, 152, 154
_Accept, 22, 130
_At, 73
_CatchResume, 77
_Cormonitor, 28, 39
_Coroutine, 7, 13
_Disable, 74
_Enable, 74
_Event, 16, 72
_Finally, 12
_Monitor, 27
_Mutex, 7, 18
_Mutex _Coroutine, 28
    _Cormonitor, 28, 39
_Mutex class, 27
    _Monitor, 27
_Nomutex, 7, 13, 18
_PeriodicTask, 133
_RealTimeTask, 135
_Resume, 73
_Select, 52, 131
_SporadicTask, 134
_Task, 7, 30
_Throw, 73
_Timeout, 130, 131
_When, 22, 52, 130, 131
__U_CPLUSPLUS_MINOR__, 11
__U_CPLUSPLUS_PATCH__, 11
__U_CPLUSPLUS__, 11
__U_DEBUG__, 11
__U_MULTI__, 11
__U_VERIFY__, 11
__U_YIELD__, 11

abort, 99, 115
accept-blocked, 22, 23
acceptor, 63
acceptor/signalled stack, 21, 22, 24, 26
access object, 61
acquire, 37, 38
activation point, 14, 30
Active, 16
active, 4, 17
active priority, 136
add, 150, 151, 153
addHead, 150, 151, 153
addTail, 151, 153
administrator, 55
amdahl's law, 47
aperiodic task, 135
argc, 8
argv, 8
assert, 98
assert.h, 98
asynchronous call, 47
asyncpoll, 16, 75
atomic instruction
    compare/assign, 41
    fetch/add, 41
    fetch/assign, 41
    test/reset, 40
    test/set, 40

barrier, 39
base priority, 136
basic priority-inheritance protocol, 135
BEGIN, 29
binomial theorem, 23
block, 39
Blocked, 32
blocked, 4
bound exception, 81
break, 11
    labelled, 11
busy wait, 37
busy waiting, 51

C++, 40, 41

- cancel, 16
- cancelInProgress, 16
- cancellation
  - safe cleanup, 91
- Cancellation, 91
- cancellation checkpoint, 91
- cancelled, 16
- chain blocking, 135
- **class**, 7
- class object, 5, 7
- class type, 7
- client, 47, 63
- CloseFailure, 62, 65, 67–69, 87
- cluster, 8
- COBEGIN, 29
- code reuse, 34
- COEND, 29
- COFOR, 29
- communication variables, 13, 30
- compare-and-assign, 41
- compilation option
  - -U++, 11
  - -compiler, 11
  - -debug, 10
  - -multi, 11
  - -nodebug, 10
  - -nomulti, 11
  - -noquiet, 11
  - -noverify, 10
  - -noyield, 10
  - -quiet, 11
  - -verify, 10, 16
  - -yield, 10, 32
  - u++, 10, 11, 123
- compile-time
  - errors, 95
  - warnings, 95
- concurrency, 8, 117
- concurrent exception, 24, 72, 74, 76, 78, 79, 103
- condition lock, 38
- condition variable, 25
- context switch, 4, 17, 22, 119
- **continue**, 11
  - labelled, 11
- convertTime, 133
- coroutine, 5, 7, 13
  - full, 13
  - inherited members, 15
  - semi, 13
  - termination, 14
- coroutine main, 13
- coroutine monitor, 5, 7, 28
- coroutine type, 7
- coroutine-monitor type, 7
- counter, 37
- CreationFailure, 87, 123
- critical section
  - hardware
    - compare/assign, 41
    - fetch/add, 41
    - fetch/assign, 41
    - test/reset, 40
    - test/set, 40

- data structure library, 149
- dbx, 142
- deadline monotonic, 138
- deadlock, 14, 25–28, 60, 86, 113, 135
- debugging
  - symbolic, 142
- default action, 73, 79, 86, 99
- Dekker, 5
- detached, 118
- drop, 150, 151, 153
- dropHead, 151, 153
- dropTail, 151, 153
- duration, 127
- dynamic
  - errors, 98
  - warnings, 98

- empty, 26, 37, 38, 150, 151, 153
- END, 29
- entry queue, 21, 25
- EntryFailure, 87
- epoch, 128
- equals, 51
- errors, 95
  - accept statement, 110
  - calendar, 111
  - cluster, 111
  - compile-time, 95
  - condition variable, 109
  - coroutine, 104
  - default action, 99
  - dynamic, 98
  - heap, 112
  - I/O, 113
  - lock, 111
  - mutex type, 106
  - processor, 113
  - runtime, 98
  - static, 95
  - task, 109
  - UNIX, 113
  - warnings, 95, 98
- exception, 71
  - bound, 81

  - concurrent, 24, 74, 76, 78, 79, 103
  - default action, 76
  - inherited members, 73
  - local, 76
  - nonlocal, 73, 74, 76, 80, 86
  - resume, 72
  - throw, 72
  - exception type, 71
- exceptional event, 71
- execution state, 4
  - active, 4
  - inactive, 4
- exit, 99, 115
- external variables, 14, 30

- Failure, 15, 62, 65, 67, 69, 86, 87, 123
- fd, 63, 66, 68, 69
- fetch-and-add, 41
- fetch-and-assign, 41
- FileAccess, 62
  - CloseFailure, 62, 87
  - Failure, 62, 87
  - fd, 62
  - fsync, 62
  - lseek, 62
  - open, 62
  - OpenFailure, 62, 87
  - read, 62
  - ReadFailure, 62, 87
  - ReadTimeout, 62, 87
  - readv, 62
  - SeekFailure, 62, 87
  - SyncFailure, 62, 87
  - write, 62
  - WriteFailure, 62, 87
  - WriteTimeout, 62, 87
  - writev, 62
- finally clause, 12
- finite-state machine, 13, 28
- fixed-point registers, 43
- floating-point registers, 43
- frame, 134
- free, 115
- free routine, 5
- front, 26
- full coroutine, 13
- futex, 61
- future, 48
- Future_ESM
  - available, 49
  - cancel, 49
  - Cancellation, 49
  - cancelled, 49
  - delivery, 49
  - exception, 49
  - **operator** T(), 49
  - **operator**()(), 49
  - reset, 49
- Future_ISM
  - available, 50
  - cancel, 50
  - Cancellation, 50
  - cancelled, 50
  - delivery, 50
  - equals, 50
  - exception, 50
  - **operator** T(), 50
  - **operator**()(), 50
  - reset, 50

- gdb, 142
- getActivePriority, 32
- getBasePriority, 32
- getClient, 68
- getClock, 118
- getCluster, 32, 118
- getCoroutine, 32
- getDetach, 118
- gethostbyip, 64
- gethostbyname, 64
- getName, 16, 61, 116
- getPid, 118
- getPreemption, 118
- getProcessors, 117
- getProcessorsOnCluster, 117
- getServer, 66
- getSpin, 119
- getStackSize, 116
- getState, 16, 32
- getTask, 118
- getTasksOnCluster, 117
- getTime, 133
- GNU C++, 10, 123, 142
- **goto**
  - restricted, 12
- guarded block, 76

- Halt, 16
- handler, 71, 72, 76
  - resumption, 72, 76, 77
  - termination, 72, 76
- head, 150, 151, 153
- heap allocator, 147
- heap area, 14, 30, 141
  - expansion size, 142
  - mmap start, 142
- heavy blocking, 120
- heavyweight, 120

heavyweight process, 120

idle, 119
implementation problems, 44
implicit scheduler, 21
Inactive, 16
inactive, 4, 17
inheritance, 34
    multiple, 36, 83
    private, 34
    protected, 34
    public, 34
    single, 34, 35, 83
inherited members
    coroutine, 15
    exception type, 73
    task, 31
initial task
    uMain, 8
insertAft, 153
insertBef, 153
internal scheduler, 25
interrupt, 118, 119
intervention, 76
IPConvertFailure, 87
isacquire, 60
istream
    isacquire, 60
itoip, 64

kernel thread, 9, 117
keyword, additions
    _Accept, 22, 130
    _At, 73
    _CatchResume, 77
    _Coroutine, 13
    _Event, 72
    _Finally, 12
    _Mutex, 18
    _Nomutex, 13, 18
    _PeriodicTask, 133
    _RealTimeTask, 135
    _Resume, 73
    _Select, 52, 131
    _SporadicTask, 134
    _Task, 30
    _Throw, 73
    _Timeout, 130, 131
    _When, 22, 52, 130, 131

labelled
    break, 11
    continue, 11
light blocking, 120
lightweight, 8
lightweight process, 8
local exception, 76
lock, 37
locking, 17

main, 8
malloc, 115
migrate, 32
monitor, 5, 7, 27
    active, 17
    inactive, 17
monitor type, 7
multi-level exit, 11
multikernel, 10, 117
mutex member, 5, 17
mutex queue, 21
mutex type, 17
mutex-type state
    locked, 17
    unlocked, 17
mutual exclusion, 4

nested control structure, 12
non-detached, 118
nonblocking i/o, 59
nonlocal exception, 72–74, 76, 80, 86, 91

object, 7
OpenFailure, 62, 65, 67, 69, 87
OpenTimeout, 65, 69, 87
ostream
    osacquire, 60
out-of-band data, 69
over, 150, 152, 154
owner, 38
owner lock, 38

P, 37
parallel execution, 8
parallelism, 8
pause, 40
periodic task, 133
poller task, 59, 116, 117
pop, 150
POSIX Threads, 121
pre-emption, 75
    default, 141
    time, 117
    uDefaultPreemption, 141
pre-emptive
    scheduling, 10, 32, 115, 119, 142
pred, 153
preprocessor variables
    __U_CPLUSPLUS_MINOR__, 11
    __U_CPLUSPLUS_PATCH__, 11

    __U_CPLUSPLUS__, 11
    __U_DEBUG__, 11
    __U_MULTI__, 11
    __U_VERIFY__, 11
    __U_YIELD__, 11
priming
    barrier, 40
printf, 61, 99
prioritized pre-emptive scheduling, 136
priority, 136
priority level, 137
priority-inheritance protocol, 135
process
    heavyweight, 120
    lightweight, 8
    UNIX, 120
processor
    detached, 118
    non-detached, 118
    number on cluster, 117
    pre-emption time, 117
    spin amount, 117
propagate, 74
pthread_attr_destroy, 122
pthread_attr_getdetachstate, 122
pthread_attr_getscope, 122
pthread_attr_getstacksize, 122
pthread_attr_init, 122
pthread_attr_setdetachstate, 122
pthread_attr_setscope, 122
pthread_attr_setstacksize, 122
pthread_cancel, 122
pthread_cleanup_pop, 122
pthread_cleanup_push, 122
pthread_cond_broadcast, 122
pthread_cond_destroy, 122
pthread_cond_init, 122
pthread_cond_signal, 122
pthread_cond_timedwait, 122
pthread_cond_wait, 122
pthread_create, 122
pthread_deletespecific_, 122
pthread_detach, 122
pthread_exit, 122
pthread_getattr_np, 122
pthread_getconcurrency, 122
pthread_getspecific, 122
pthread_join, 122
pthread_key_create, 122
pthread_key_delete, 122
pthread_mutex_destroy, 122
pthread_mutex_init, 122
pthread_mutex_lock, 122
pthread_mutex_trylock, 122
pthread_mutex_unlock, 122
pthread_once, 122
pthread_self, 122
pthread_setcancelstate, 122
pthread_setcanceltype, 122
pthread_setconcurrency, 122
pthread_setspecific, 122
pthread_testcancel, 122
pthread_timedjoin_np, 122
pthread_tryjoin_np, 122
pthread_yield, 122
pthreads, 121
    see POSIX Threads,
push, 150
push-down automata, 13

raising, 71–73
    resuming, 72, 73
    throwing, 72, 73
read, 59
ReadFailure, 62, 65, 67, 69, 87
ReadTimeout, 62, 65, 67, 69, 87
Ready, 32
ready, 4
real-time cluster, 138
recursive resuming, 78, 79
release, 38
remove, 151, 153
rendezvous, 23, 88
RendezvousFailure, 86–88, 102, 103
reresume, 73
reset, 39
resetClock, 133
resume, 15, 17
resumer, 16
resumption, 71
resumption handler, 76, 77
rethrow, 73
return code, 71
Running, 32
running, 4
runtime
    errors, 98
    warnings, 98

scanf, 61
SeekFailure, 62, 87
select, 59, 116
select blocked, 52
semaphore, 36
semi-coroutine, 13
SendfileFailure, 65, 67, 69, 87
SendfileTimeout, 65, 67, 69, 87
server, 47, 63

set_terminate, 84
set_unexpected, 84
setClient, 68
setCluster, 118
setName, 16, 61, 116
setPreemption, 118
setServer, 66
setSpin, 119
setStackSize, 116
shared-memory model, 8
signal, 26
signalBlock, 26
sleep, 115, 130
socket, 63
    endpoint, 63
spin
    amount, 117
    default, 141
    lock, 37
    uDefaultSpin, 141
    virtual processor, 119
sporadic task, 134
stack, 4, 149
    acceptor/signalled, 21, 22, 24, 26
    amount, 16
    automatic growth, 10, 105
    current, 16
    data structure, 149
    default size, 15, 31, 57, 115, 116, 141
    diddling, 33
    free, 16
    minimum size, 115
    overflow, 16, 105
    recursive resuming, 78
    storage, 16
    uDefaultStackSize, 141
    uMainStackSize, 141
    used, 16
stackFree, 16
stackPointer, 16
stackSize, 16
stackStorage, 16
stackUsed, 16
stale readers, 155
START, 29
Start, 32
starter, 16
static
    errors, 95
    warnings, 95
static storage, 14, 30
static multi-level exit, 11
status flag, 71
StatusFailure, 62, 87

subtyping, 34
succ, 151, 153
suspend, 15, 17
SyncFailure, 62, 87
synchronous call, 47
system cluster, 9

tail, 151, 153
task, 5, 7, 30
    aperiodic, 135
    inherited members, 31
    periodic, 133
    sporadic, 134
    termination, 30
task main, 30
task type, 7
Terminate, 32
terminate, 84
terminate_handler, 84
TerminateFailure, 61, 62, 87
termination, 71
termination handler, 76
test-and-reset instruction, 40
test-and-set, 40
thread, 4
    blocked, 4
    running, 4
time, 127
time stamp counter, 40
time-slice, 119
times, 38
top, 150
total, 39
transfer, 151, 153
translator, 7
    problems, 44
tryacquire, 38
TryP, 37

u++, 10, 11, 123
uAbort, 99, 115
uBarrier, 39
    block, 39
    last, 39
    reset, 39
    total, 39
    waiters, 39
uBarrier.h, 40
uBaseCoroutine, 15, 32, 132
    asyncpoll, 15, 75, 91
    cancel, 15
    cancelInProgress, 15
    cancelled, 15
    Failure, 15, 86, 87

- getName, 15
- getState, 15
- resume, 15
- resumer, 15
- setName, 15
- stackFree, 15
- stackPointer, 15
- stackSize, 15
- stackStorage, 15
- stackUsed, 15
- starter, 15
- suspend, 15
- UnhandledException, 15, 87
- verify, 15

uBaseEvent, 73
- Cancellation, 87
- defaultResume, 73
- defaultTerminate, 73
- duplicate, 73
- getRaiseKind, 73
- message, 73
- RaiseKind, 73
- reraise, 73
- setMsg, 73
- source, 73
- sourceName, 73
- uIOFailure, 87
- uKernelFailure, 87

uBaseSchedule
- add, 137
- addInitialize, 137
- checkPriority, 137
- empty, 137
- getActivePriority, 136
- getBasePriority, 136
- getInheritTask, 136
- pop, 137
- removeInitialize, 137
- rescheduleTask, 137
- resetPriority, 137
- setActivePriority, 136
- setBasePriority, 136

uBaseTask, 31
- getActivePriority, 31
- getBasePriority, 31
- getCluster, 31
- getCoroutine, 31
- getState, 31
- migrate, 31
- yield, 31

uBaseTask::Blocked, 32

uBaseTask::Ready, 32

uBaseTask::Running, 32

uBaseTask::Start, 32

uBaseTask::Terminate, 32

*μ*C++ translator, 7

*μ*C++ kernel, 10, 115

uClock
- convertTime, 132
- getTime, 132
- resetClock, 132

uCluster, 116
- ExceptSelect, 116
- getName, 116
- getProcessors, 116
- getProcessorsOnCluster, 116
- getStackSize, 116
- getTasksOnCluster, 116
- ReadSelect, 116
- select, 116
- setName, 116
- setStackSize, 116
- WriteSelect, 116

uColable, 149
- listed, 149

uCondition, 25
- empty, 25
- front, 25
- owner, 25
- signal, 25
- signalBlock, 25
- wait, 25
- WaitingFailure, 25, 87

uCondLock, 38, 121
- broadcast, 38
- empty, 38
- signal, 38
- wait, 38

uContext, 42

uDefaultHeapExpansion, 142

uDefaultMmapStart, 142

uDefaultPreemption, 141
- see setPreemption and pre-emption,

uDefaultProcessors, 141

uDefaultSpin, 141
- see setSpin and spin,

uDefaultStackSize, 141
- see setStackSize and stack,

uDuration, 127

uExecutor
- send, 57
- sendrecv, 57

uFile, 61
- Failure, 62, 87
- getName, 62
- setName, 62
- status, 62
- StatusFailure, 62, 87

TerminateFailure, 62, 87
uFile.h, 62
uFloatingPointContext, 43
uIOFailure, 87
uKernelFailure
uMutexFailure, 87
uLock, 37
acquire, 37
release, 37
tryacquire, 37
uMain, 8
argc, 8
argv, 8
uRetCode, 8
uMain::main, 8
uMainStackSize, 141
see setStackSize and stack,
uMutexFailure, 86
EntryFailure, 87
RendezvousFailure, 87
uncaught_exception, 85
unexpected, 84
unexpected_handler, 84
UnhandledException, 15, 84, 86, 87, 100
triggerCause, 88
unikernel, 10, 117
UNIX epoch, 128
unix process, 117
unlocking, 17
uOwnerLock, 38, 121
acquire, 38
owner, 38
release, 38
times, 38
tryacquire, 38
uPeriodicBaseTask
getPeriod, 134
setPeriod, 134
uProcessor, 118
getClock, 118
getCluster, 118
getDetach, 118
getPid, 118
getPreemption, 118
getSpin, 118
getTask, 118
idle, 118
setCluster, 118
setPreemption, 118
setSpin, 118
uPthreadable, 123
CreationFailure, 87, 123
Failure, 87, 123
pthreadId, 123
uQueue
add, 151
addHead, 151
addTail, 151
drop, 151
dropHead, 151
dropTail, 151
empty, 151
head, 151
remove, 151
succ, 151
tail, 151
transfer, 151
uQueueIter, 151
>>, 152
over, 152
uRealTimeBaseTask
getDeadline, 135
setDeadline, 135
uRendezvousAcceptor, 89
uRetCode, 8
uSemaphore, 36
counter, 36
empty, 36
P, 36
TryP, 36
V, 36
uSemaphore.h, 37
uSeqable, 149
listed, 149
uSeqIter, 153
>>, 153
over, 153
uSeqIterRev, 153
>>, 153
over, 153
uSequence
add, 153
addHead, 153
addTail, 153
drop, 153
dropHead, 153
dropTail, 153
empty, 153
head, 153
insertAft, 153
insertBef, 153
pred, 153
remove, 153
succ, 153
tail, 153
transfer, 153
user cluster, 9
uSerial

- RendezvousFailure, 86, 88, 102, 103

usleep, 115, 130

uSocket, 64
- CloseFailure, 87
- Failure, 87
- IPConvertFailure, 87
- OpenFailure, 87

uSocket.h, 64

uSocketAccept, 68, 132
- accept, 69
- close, 69
- CloseFailure, 69, 87
- Failure, 69, 87
- fd, 69
- getpeername, 69
- getsockaddr, 69
- getsockname, 69
- OpenFailure, 69, 87
- OpenTimeout, 69, 87
- read, 69
- ReadFailure, 69, 87
- ReadTimeout, 69, 87
- readv, 69
- recv, 69
- recvfrom, 69
- recvmsg, 69
- send, 69
- SendfileFailure, 69, 87
- SendfileTimeout, 69, 87
- sendmsg, 69
- sendto, 69
- write, 69
- WriteFailure, 69, 87
- WriteTimeout, 69, 87
- writev, 69

uSocketClient, 65, 132
- CloseFailure, 65, 87
- Failure, 65, 87
- fd, 65
- getpeername, 65
- getServer, 65
- getsockname, 65
- OpenFailure, 65, 87
- OpenTimeout, 65, 87
- read, 65
- ReadFailure, 65, 87
- ReadTimeout, 65, 87
- readv, 65
- recv, 65
- recvfrom, 65
- recvmsg, 65
- send, 65
- SendfileFailure, 65, 87
- SendfileTimeout, 65, 87
- sendmsg, 65
- sendto, 65
- setServer, 65
- write, 65
- WriteFailure, 65, 87
- WriteTimeout, 65, 87
- writev, 65

uSocketServer, 66
- CloseFailure, 67, 87
- Failure, 67, 87
- fd, 67
- getClient, 67
- getpeername, 67
- getsockaddr, 67
- getsockname, 67
- OpenFailure, 67, 87
- read, 67
- ReadFailure, 67, 87
- ReadTimeout, 67, 87
- readv, 67
- recv, 67
- recvfrom, 67
- recvmsg, 67
- send, 67
- SendfileFailure, 67, 87
- SendfileTimeout, 67, 87
- sendmsg, 67
- sendto, 67
- setClient, 67
- write, 67
- WriteFailure, 67, 87
- WriteTimeout, 67, 87
- writev, 67

uSporadicBaseTask
- getFrame, 135
- setFrame, 135

uStack, 149
- add, 150
- addHead, 150
- drop, 150
- empty, 150
- head, 150
- pop, 150
- push, 150
- top, 150

uStackIter, 150
- >>, 150
- over, 150

uThisCluster, 117

uThisCoroutine, 17

uThisProcessor, 119

uThisTask, 32

uTime, 128

uWaitQueue_ESM

  add, 53
  drop, 53
  empty, 53
  remove, 53
uWaitQueue_ISM
  add, 53
  drop, 53
  empty, 53
  remove, 53

V, 37
verify, 16
version number, 11
virtual processor, 9, 117, 120

WAIT, 29
wait, 25
wait-all, 54
wait-any, 54
wait-for-all, 51
wait-for-any, 51
waiters, 39
WaitingFailure, 25, 87
warnings
  compile-time, 95
  runtime, 98
write, 43, 59
WriteFailure, 62, 65, 67, 69, 87
WriteTimeout, 62, 65, 67, 69, 87

yield, 117
  compilation option, 10, 32
  preprocessor, 11
yield, 32, 91