Object-Oriented Real-Time Concurrency

Peter A. Buhr, Ashif S. Harji, Philipp E. Lim, and Jiongxiong Chen
University of Waterloo
Waterloo, Ontario, CANADA, N2L 3G1
pabuhr,asharji,j7chen@plg.uwaterloo.ca

ABSTRACT

The primary goal of a real-time system is predictability. Achieving this goal requires *all* levels of the system to work in concert to provide fixed worst-case execution-times. Unfortunately, many real-time systems are overly restrictive, providing only ad-hoc scheduling facilities and basic concurrent functionality. Ad-hoc scheduling makes developing, verifying, and maintaining a real-time system extremely difficult and time consuming. Basic concurrent functionality forces programmers to develop complex concurrent programs without the aid of high-level concurrency features.

Encouraging the use of sophisticated real-time theory and methodology, in conjunction with high-level concurrency features, requires flexibility and extensibility. Giving real-time programmers access to the underlying system data-structures makes it possible to interact with the system to incorporate new ideas and fine-tune specific applications. This paper explores this approach by examining its effect on a selection of crucial real-time issues: real-time monitors, timeouts, dynamic-priority scheduling and basic priority inheritance. The approach is implemented in μC++.

1. INTRODUCTION

A real-time system is characterized by its ability to meet specified timing constraints. In order to achieve this goal, all aspects of the system must be predictable. The criteria to achieve this predictability ranges from language mechanisms to specify concurrency and time-dependent operations, through fixed worst-case execution-time data-structures within the runtime system, to scheduling tasks using a well-defined algorithm. Predictability *cannot* be achieved solely through task scheduling. Like concurrency, real-time has a pervasive effect throughout a system, requiring many internal components to know and react differently to achieve predictability [5, 33]. In addition to predictability, a real-time system should be flexible and extensible in order for it to be suitable for a diverse set of applications and to take advantage of new techniques and algorithms. The ability to program different real-time applications and approaches with the same system discourages multiple ad-hoc systems and encourages the exploitation of new ideas.

OOPSLA '00 10/00 Minneapolis, MN, USA
© 2000 ACM ISBN 1-58113-200-X/00/0010...$5.00
ACM SIPLAN Notices 35(10):29–46, Oct. 2000

The primary motivation of this work is the construction of a flexible real-time system in μC++ [10, 20, 27]. μC++ is a translator and runtime kernel for C++ supporting lightweight concurrency using a shared-memory model. The translator transforms the μC++ language constructs into C++; the runtime kernel supports both uniprocessor and multiprocessor architectures. Extensions to μC++ for real-time, exception handling, debugging and profiling are ongoing.

Four specific real-time issues related to constructing a predictable object-oriented real-time system are presented; each issue represents a fundamental component of any real-time system. As well, the interaction of these components are examined where appropriate. Specific versions of these components are then implemented as real-time extensions to μC++. The first component discusses how high-level object-oriented constructs, e.g., monitors, can be extended for a real-time system. Both mutual exclusion and synchronization are discussed. The second component discusses a timeout mechanism for accept statements. The third component discusses priority-based scheduling; specifically, a method for assigning dynamic priorities and a particular constant-time priority-queue data-structure is proposed. The fourth component discusses basic priority inheritance and its implementation. All four components are relevant to both real-time language and system developers.

2. CONCURRENCY CONSTRUCTS

A simple lock construct, e.g., a semaphore, is necessary and sufficient for concurrent programming. While a lock may be an object, it is low-level, error-prone, and not integrated into the object model. Furthermore, unless locks are made part of the programming language, the resulting concurrency is either unsound and/or inefficient [8]. Therefore, only high-level object-integrated constructs are examined. The *monitor* construct is used as the basis for discussion, but multiple issues are covered to include other object-integrated concurrent constructs, such as the Ada [23] task.

Preliminary work on the monitor was done by Hansen [7] and Hoare [21], and it is essentially the first high-level object-integrated concurrency-construct. The monitor is normally based on the class construct, with the addition of integrating mutual exclusion, with respect to the monitor instance, into method call; serialized methods are called *mutex methods*. Internally, monitors provide an *explicit scheduling* mecha-

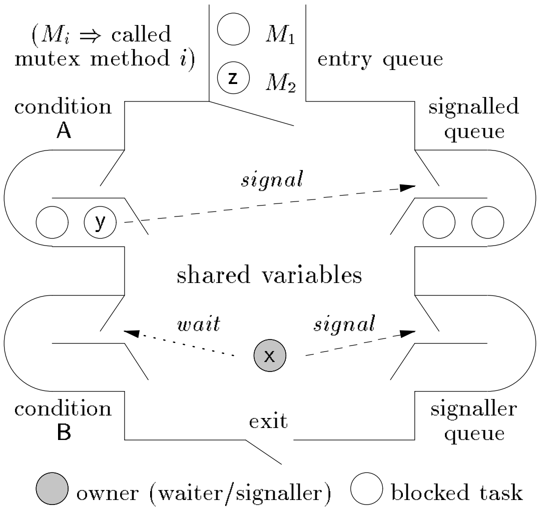

Figure 1: Internal Scheduling

nism for synchronization, via special statements in the mutex methods to block and unblock task execution, e.g., signal and wait. As well, a monitor has an *implicit scheduling* mechanism to keep the monitor active, which varies depending on the kind of monitor [9]. A monitor instance is used for indirect task synchronization and communication.

Much of the theoretical work on real-time scheduling deals with CPU scheduling and mutual exclusion of resources protected with semaphores, with the suggestion that monitors follow directly [32, p. 16]. However, this suggestion is only correct for mutual exclusion; for synchronization scheduling, most theoretical work does not apply. Since *all* scheduling is fundamental to real-time systems, it is important to analyse monitor scheduling to understand its effect on real-time behaviour. The following discussion generalizes implicit scheduling in objects providing mutual exclusion and synchronization through the internal data-structures used for scheduling. The manipulation of these data structures during implicit scheduling defines monitor semantics. However, real-time scheduling must interact with these objects through these data structures. Therefore, understanding implicit scheduling is necessary to know what can and cannot be done with respect to real-time interactions.

2.1 Scheduling

Basic monitor semantics are presented using the general form in Figure 1. A task enters a monitor by calling one of its mutex methods. When a task is executing inside the monitor, the monitor is *active* and this task is designated the monitor *owner*; otherwise, the monitor is *inactive*. When a task calls a mutex method of an active monitor, it blocks on the *entry queue*; such tasks are *entry blocked*. If no explicit scheduling is performed in the monitor, implicit scheduling usually processes the entry-blocked tasks in first-in first-out (FIFO) order. An alternative approach is to have a separate queue for each mutex method, called a *mutex queue*, and to choose arbitrarily among these queues.

2.1.1 Internal Scheduling

When the monitor owner synchronizes with tasks blocked on monitor condition-queues, it is called *internal scheduling*. This form of synchronization is achieved using condition variables and wait/signal statements. (Ada offers similar functionality through the requeue statement.) In Figure 1, if owner task x performs statement wait B, x blocks on condition B, the monitor becomes inactive, and another task is implicitly scheduled into the monitor from the entry, signalled, or signaller queues. Alternatively, if task x performs statement signal A, it blocks on the implicit *signaller queue*; task y is removed from condition A and placed on the implicit *signalled queue*. The monitor becomes inactive and another task is implicitly scheduled into the monitor from one of the entry, signalled, or signaller queues.

Thus, there are three ways an active monitor becomes inactive: the owner exits the monitor, blocks on a condition variable, or signals a non-empty condition variable. When the monitor becomes inactive, there are three queues from which a task can potentially be selected to become the new monitor owner: entry, signalled or signaller queues. The signalled and signaller queues are referred to as *internal queues*, whereas the entry queue is referred to as an *external queue*. Note, condition queues are ineligible for implicit scheduling as the tasks on these queues are blocked. The order these three queues are considered during implicit scheduling determines the kind of monitor, where a queue can have priority less than, equal to, or greater than another queue. This semantics describes explicit and implicit scheduling in all extant monitors [9]. For example, in Java [18], the signaller queue has highest priority, and the signalled and entry queues have equal priority so the choice is arbitrary. For equal priority queues, the implementation may merge these queues, providing FIFO order across them. Not all orderings for selecting among the queues result in useful monitors. Any ordering in which the entry queue has highest priority can result in starvation and synchronization difficulties with tasks blocked inside the monitor. The key point is that the implicit selection order is fixed for a particular kind of monitor, and a programmer develops a monitor based on this guaranteed behaviour.

Finally, while three implicit queues exist from a theoretical point of view, typically at most two queues are needed to implement a monitor. For certain kinds of monitors, either the signaller or the signalled queue is eliminated because the task that blocks on it is immediately unblocked, as for the signaller task in Java. Thus, the queue is eliminated and any task that would have been placed on this queue immediately becomes the monitor owner.

2.1.2 External Scheduling

An alternative to internal scheduling is *external scheduling*. A common implementation of external scheduling is with an accept statement, but other implementations are protected entries in Ada, operation avoidance in Sylph [14] and path expressions [1]. While external scheduling is often associated with rendezvous among task objects, it is equally applicable to the monitor, where the monitor owner synchronizes with tasks on the entry queue (or mutex queues). This form of synchronization is typically achieved by the monitor owner specifying which calls to mutex methods are *eligible*. Only a task calling an eligible mutex-method is permitted to enter the monitor when the monitor becomes inactive and a selection decision is made. In Figure 2, if task x performs the

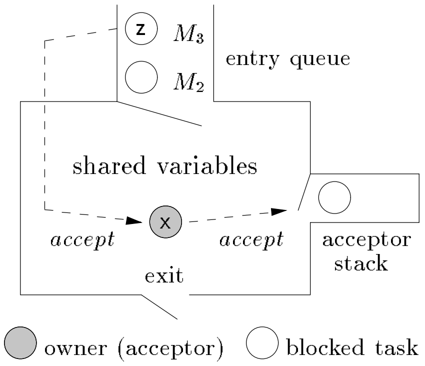

Figure 2: External Scheduling

statement accept(M_1,M_3), it blocks on the implicit *acceptor stack*; task z is removed from the entry queue and becomes the monitor owner. In general, the acceptor task waits for the calling task to exit the monitor before continuing, i.e., the *rendezvous*, allowing the calling task to satisfy whatever conditions the acceptor may require to continue. Nested accepts are possible, e.g., task z executes an accept statement, and in order to maintain the described semantics, accept-blocked tasks unblock in last-in first-out (LIFO) order. The key point is that the LIFO selection order is fixed for external scheduling, and a programmer develops a monitor based on this guaranteed behaviour.

2.1.3 Internal and External Scheduling

Unifying internal and external scheduling is possible for both monitor and task types providing scheduling. Supporting both scheduling mechanisms is important because each provides unique capabilities. (The Ada approach using requeue is not as powerful as having internal scheduling.) Where appropriate, external scheduling seems easier to use and understand than internal scheduling [11, p. 235]. However, external scheduling cannot deal adequately with mutex-method parameters or breaking a rendezvous, which necessitates internal scheduling. Combining internal and external scheduling results in a monitor (or task) having either the implicit signalled or signaller queue (only one is usually necessary), and the acceptor stack (see Figure 3). When a monitor becomes inactive, the kind of monitor dictates a particular selection while rendezvous dictates another, resulting in a potential conflict. Without a careful understanding of the scheduling interactions, both starvation and undesirable semantics can result.

For external scheduling, not only must the calling task have highest selection priority, but the acceptor must be the next task to execute after the calling task exits the monitor to preserve rendezvous semantics, which means selecting the acceptor ahead of tasks on internal scheduling queues. Therefore, if an accepted task signals a condition variable, it is moved to the signaller queue and the signalled task to the signalled queue, but the acceptor unblocks next from the acceptor stack rather than either of these tasks. This scheduling decision can result in starvation, making the internal scheduling facility useless. Basically, having both signal and acceptor queues is the problem. The most reasonable semantics is obtained when these queues are merged, but there are constraints in adopting this approach.

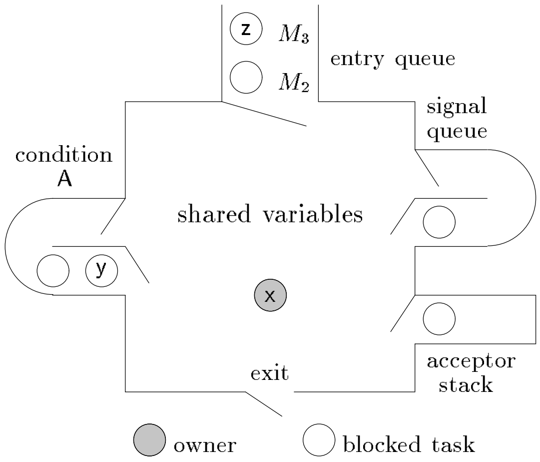

Figure 3: Internal/External Scheduling

First, acceptors must be processed in LIFO order but tasks involved in internal scheduling are usually processed in FIFO order. If external scheduling must be LIFO, can internal scheduling be changed to LIFO and still preserve the kind of monitor? When a task signals at most one task, the kind of monitor is preserved because there is only one task on the signal queue, so the queue order is immaterial. Only when a task signals multiple tasks do those tasks now reenter the monitor in LIFO as opposed to FIFO order. However, this semantics does not invalidate the kind of monitor because the implicit queues are still selected in the appropriate order, and often the order multiple tasks unblock is unimportant when signalled as a group, furthermore, it is impossible to guarantee an external view of FIFO exiting with preemption. If multiple tasks are signalled and FIFO release order is essential, it can be achieved using an explicit technique called daisy-chain signalling, where each task signals the next task in a sequence rather than having a single task signal a group of tasks. Therefore, if a programmer is aware of the LIFO order and FIFO unblocking is important, it is possible to mimic this semantics given a particular kind of monitor.

Second, the acceptor/calling task order must be preserved even when intervening internal scheduling occurs, e.g., by an accepted task. Essentially, signalled (or signaller) tasks are pushed on the stack above the acceptor, and are processed as part of the rendezvous *before* the acceptor is unblocked. Clearly, the acceptor is free to use internal scheduling before or after the rendezvous without problems.

Finally, it is possible for an accepted task to block on a condition variable *before* a rendezvous is complete. For example, the action the acceptor is waiting for from an accepted task could not be satisfied at this time, and hence, the rendezvous is broken. The next task to be selected now depends on the kind of monitor, and it is possible for the acceptor to be unblocked even though an accepted task has not finished execution. Basically, if a rendezvous can fail, a programmer has to be ready to deal with it.

The crucial point is that certain monitor actions cannot be changed without invalidating the monitor's semantics, and a programmer relies on these semantics for correct execution behaviour. Maintaining the semantic behaviour of a monitor is equally crucial in the real-time domain.

2.2 Real-Time Considerations

The most important criteria for a real-time system is that a task meet its deadlines. Since monitor access is serialized, it is possible for a high-priority task to call into a monitor but have to wait for a low-priority task to exit the monitor, called *priority inversion*, when FIFO order is used for the entry queue. Priority inversion can result in unacceptable delays, resulting in missed deadlines. Therefore, high-priority tasks should be given preference over lower-priority tasks when scheduling tasks into a monitor. How can a monitor's behaviour be modified to suit this real-time requirement?

Real-time monitor modifications are restricted by the requirement to preserve semantic behaviour, otherwise programming becomes difficult and reuse impossible. Since the monitor semantics largely dictates selection *among* queues, real-time changes are limited to the ordering of the tasks *within* a particular queue. A naive approach is to simply prioritize the various queues in the monitor; however, this is not always reasonable. For internal scheduling, it is reasonable to prioritize the entry, signaller and signalled queues to expedite entry of high-priority tasks into the monitor. However, for the signaller and signalled queues, deviating from FIFO order is like switching to LIFO: daisy-chain signalling may need to be used to control scheduling order if tasks are processed in priority order for multiple signalling. As well, using priority order for the entry queue is incorrect for applications like the readers/writer problem, resulting in stale information if tasks are not processed in FIFO order. For external scheduling, it is reasonable to prioritize among tasks calling eligible mutex methods. However, the acceptor stack cannot be prioritized; LIFO order is required to retain rendezvous semantics. For internal/external scheduling with an acceptor/signal stack, it is possible to prioritize the entry queue and prioritize among tasks calling eligible mutex methods for external scheduling, but the acceptor stack cannot be prioritized for the same reason as above.

It also seems reasonable to allow prioritizing tasks on condition queues; unfortunately, this presents problems. First, starvation of tasks waiting on the condition variable becomes an issue, as a low-priority task may be delayed indefinitely while higher-priority tasks are removed from the condition queue. Second, there is an implementation issue if task priorities change over time, called *dynamic priorities*, where a task's priority can change at any time, even if it is blocked. Changing priorities present a queue maintenance problem as condition queues are usually only modifiable by the monitor owner, as they are considered internal data structures (as is the acceptor/signal stack). Dynamic priorities may require a condition queue to be occasionally reordered by a task *outside* of the monitor, which requires locking the condition queue. Rather than forcing all users to pay for locking on condition variables, it is reasonable to ask the programmer to explicitly code a prioritized condition queue if needed, requiring priority values to be accessible at the user-level.

These problems, as well as the desire to use schemes to deal with priority inversion and perform dynamic scheduling, require a certain flexibility in monitor implementation and extending that flexibility to the real-time programmer.

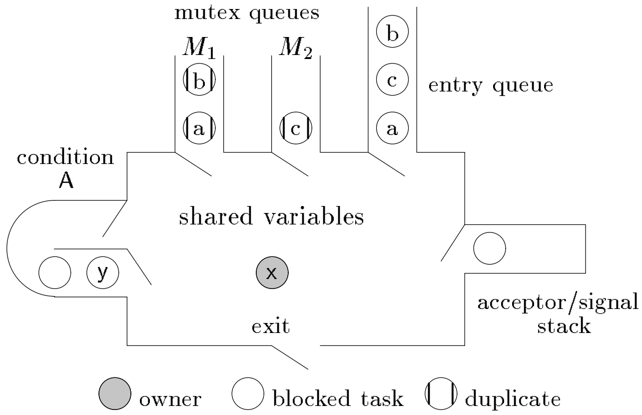

Figure 4: μC++ Monitor

2.3 μC++ Monitor Design & Implementation

In μC++, monitors are integrated into C++ classes through mutex methods, and have all the capabilities of a C++ class (including inheritance); objects generated from these types can be created in any storage class. (Coroutines and tasks are also integrated into C++ classes with mutex methods, and the following discussion also applies to these type generators.) Mutex methods are created implicitly or explicitly by qualifying `class` with `uMutex`, which implicitly makes all public methods into mutex methods, or explicitly by qualifying individual methods (public/protected/private) with `uMutex` or `uNoMutex`, e.g.:

```
uMutex class M {              // => public methods implicitly mutex
  private:
     uMutex int m1();         // => explicit mutex method
     int m2();                // => implicit no mutex method
  public:
     int m3();                // => implicit mutex method
     uNoMutex int m4();       // => explicit no mutex method
};
```

The destructor of a monitor is always a mutex method.

μC++ monitors differ from the basic Hoare monitor in several ways (see Figure 4). First, the kind of monitor in μC++ is priority non-blocking [9], with a combined acceptor/signal stack. For internal scheduling, priority non-blocking means the signaller queue is selected first (and so it is eliminated), then the acceptor/signal stack, and finally, the entry queue. Having the signaller remain the monitor owner seems intuitive for many programmers. It is also more efficient because most `signal` statements occur immediately before a task exits, allowing the signaller to exit immediately versus waiting for the signalled task, which eliminates a context switch and increases concurrency. Selecting from the acceptor/signal stack next means tasks scheduled in the monitor are serviced before tasks entering the monitor, which eliminates inefficient busy waiting (e.g., loops around `wait` statements, as in Java) because calling tasks cannot barge into the monitor. Second, μC++ monitors support both internal (condition/signal/wait) and external scheduling (accept statement) as both are essential capabilities. When a calling task blocks because the monitor is active, it is placed on *both* the entry queue and the mutex queue associated with the method it called. The entry queue is needed to maintain FIFO ordering of callers for implicit scheduling. The mutex queues are solely an optimization to allow an accept statement to check each queue for callers in $O(1)$ time, eliminating an $O(n)$ search of the entry queue. Finally, μC++ monitors support recursive entry, i.e., a monitor owner can call back into the monitor, either directly or indirectly, eliminating a common source of deadlock. While it is possible to restructure a monitor to prevent this deadlock, it is unnecessarily complex.

A potential drawback of these enhancements is requiring programmers to learn additional semantics and adopt a coding style that differs slightly from that used with the Hoare monitor. However, experience with these enhancements suggests they make monitors easier to use and more intuitive.

Augmenting monitors (or tasks) for real-time is often done using techniques like attributes to mutex-locks/condition variables (POSIX [12]) or language pragmas (Ada) or subclassing (Java [5]). Often there are a fixed set of real-time capabilities supported, with no provision for user extensions. However, considering the restrictions discussed above, there are only a few modifications possible when moving to real-time monitors, i.e., the order tasks are processed on the entry, mutex and condition queues. As suggested in the previous section, the condition queues are best left as FIFO and the programmer is responsible for different scheduling schemes within the monitor. As well, tasks may need to perform additional actions when entering or leaving a monitor. These actions can be achieved by providing hooks that are invoked in the monitor entry and exit code and the remaining changes can be encapsulated within the functionality of the various queues.

The μC++ monitor is extended to allow the type of the entry and mutex queues to be specified on the class uMutex/uNoMutex qualifier, using template-like syntax:

```
uMutex<EntryQueueType, MutexQueueType> class M { ...
```

These types must be derived from the class uBasePrioritySeq:

```
class uBasePrioritySeq {
  public:
    virtual bool uEmpty() const;
    virtual uBaseTaskDL *uHead() const;
    virtual int uAdd( uBaseTaskDL *n, uBaseTask *o, uSerial *s );
    virtual uBaseTaskDL *uDrop();
    virtual void uRemove( uBaseTaskDL *n );
    virtual void uOnAcquire( uBaseTask &owner, uSerial *s );
    virtual void uOnRelease( uBaseTask &oldowner, uSerial *s );
};
```

Differentiating entry and mutex queue types allows, e.g., the entry queue to be a FIFO doubly-linked list and the mutex queues to be FIFO singly-linked lists, which is the default if no queues are specified. In this case, nodes can be removed from anywhere in the entry queue in $O(1)$ time during external scheduling with accept statements, and removed in $O(1)$ time from the front of a mutex queue. The methods uEmpty, uHead, uAdd, uDrop and uRemove provide a generalized interface to a queue. The methods uOnAcquire and uOnRelease provide hooks that are called after/before a task acquires/releases control of a monitor. No hook is needed when a task blocks on the entry queue because the uAdd method is already invoked to add it to the entry queue. These hooks allow advanced schemes such as priority inheritance (see Section 5.4) to be implemented. The basic idea is to give a task about to entry block on a monitor the ability to influence the active task in the monitor, e.g., raise its priority. Also, when a task exits the monitor, it can reevaluate any modifications it underwent while in the monitor. Finally, the monitor extensions work with C++ templates:

```
template<class EQ,class MQ> uMutex<EQ,MQ> class M { ...
```

This monitor template can be instantiated with various types of queues to get different kinds of monitors, e.g., for both normal and real-time applications.

Note, this extensibility is beyond the polymorphism in the base language, i.e., template in C++. Normal polymorphism is between user/user code, while the extension is between user/system code. Alternative approaches usually involve "magic" type names for inheritance or templates, which violates good language design. The key point is that extensibility is crucial because real-time knowledge is growing and applications require *very* specific solutions.

3. TIMEOUT

A fundamental part of real-time programming is the ability to specify timing constraints for tasks. These constraints are necessary to create both predictable and schedulable systems. To satisfy these conditions, it must be possible to specify the worst-case execution-time of a task. Thus, it is inappropriate to use potentially unbounded operations in a real-time system. For certain operations, it is reasonable for the programmer to address the potential for unboundedness by bounding loop iterations and recursion depth. However, for operations like task synchronization and communication or I/O, this expectation may be unreasonable. For example, it is unrealistic to eliminate synchronization and communication, by requiring all tasks to be independent, or perform no I/O. Therefore, a technique to bound these operations is required. The real problem with these operations is not unbounded execution, but rather, unbounded blocking.

To address unbounded blocking, it is necessary to implement a *timeout mechanism* that temporally limits an operation by aborting it if no progress has occurred within a specified time. Clearly, the amount of progress varies depending on the type of operation. For operations that might block for a potentially unbounded time, as opposed to operations that might execute for an unbounded time, the requirement is usually that the operation begins executing within a specified time. This degree of progression is reasonable because once the operation actually starts, it is possible for a programmer to limit the worst-case execution-time.

For task synchronization and communication it is possible, though largely impractical, to limit the blocking time at the user level. The problem is that the completion of these operations is dependent on other tasks in the system, making it difficult to characterize exact task behaviour. Not only must transient overloads and error conditions be considered, but also an extremely large number of execution paths, if preemption is allowed. For example, one user-level approach is for a task, T_1, to create a *timeout task*, before it blocks. The timeout task blocks for the specified amount of time using a time delay, e.g., sleep(t), and then tries to wake up T_1. If an eligible call occurs before the timeout expires, the timeout must be *short-circuited* to prevent the wake up from the timeout task, requiring a mechanism to immediately unblock the timeout task. While possible, this approach is reasonably complex, requiring setup and coordination on the part of the programmer, and it incurs the overhead of creating and managing timeout tasks. As suggested, a more reasonable approach for preventing unbounded blocking is to implement a timeout mechanism, placing the onus on the programmer to use it where appropriate. For I/O, unless the operating system provides an explicit mechanism to abort an operation before it completes, it is impossible to limit the blocking time of these operations at the user level.

Therefore, in designing a timeout facility, several issues and conditions are important. In terms of syntax and semantics, the facilities should be easy to use and provide a natural extension to the existing syntax. As well, the details of the implementation should be transparent to the user. The implementation must have: minimal impact when not employed, not introduce any additional potential for deadlock into the system, and incur a small, fixed overhead. Finally, for maintenance reasons it is also important for the implementation to have limited complexity.

A timeout capability was added to the μC++ synchronization and I/O operations to cancel the operation after a specified delay. For synchronization, only external scheduling (accept statement) is augmented. Providing timeout for internal scheduling (condition/wait/signal) is problematic, and the reasons are presented below. Only the synchronization timeout capability is discussed; the I/O timeouts are embedded in the I/O library but employ the facilities used by the synchronization timeout.

3.1 μC++ Accept Timeout

The accept statement in μC++ is similar to that in Ada and Concurrent C [16]:

accept:
  when$_{opt}$ uAccept (*name-list*) *statement*
  when$_{opt}$ uAccept (*name-list*) *statement* uOr *accept*
  when$_{opt}$ uAccept (*name-list*) *statement* uElse *statement*
  timeout
when:
  uWhen (*expression*)
timeout:
  when$_{opt}$ uTimeout (*time-value*) *statement*

The optional uWhen clause is referred to as a guard and consists of a conditional expression. If the guard evaluates to true or is omitted, the accept clause is referred to as *open*, otherwise it is referred to as *closed*. The evaluation of an accept statement begins by determining if an open uAccept clause is *immediately acceptable*, i.e., if an outstanding call to the method associated with the open accept clause exists. If multiple immediately acceptable clauses exist, the first one in textual order is chosen, as opposed to a non-deterministic selection. The uElse clause or an open timeout-clause is referred to as an *alternative*.

If there is no immediately acceptable clause and no alternative exists, the acceptor blocks until a call to one of the open accept clauses occurs. When a call occurs to an eligible mutex-method, the monitor becomes active with the calling task until it exits, at which time the acceptor is typically unblocked (unless the caller uses internal scheduling). When the acceptor unblocks, it executes the statement following the accept clause for the called method. If there is no immediately acceptable clause but an alternative clause exists, the alternative is executed. For the uElse clause, the acceptor does not block, making it possible to poll for outstanding calls. For the uTimeout clause, the acceptor blocks no longer than the specified delay for a call to one of the open accept clauses. Note, uTimeout is also used for simple time delay, e.g., uTimeout(1) delays a task for one second.

In μC++, only one uElse or uTimeout clause is allowed and it must appear as the last clause in an accept statement. These restrictions are reasonable because only one alternative can be selected even if several could exist, and forcing these clauses to appear last fits logically with the selection order of clauses in an accept statement. The equivalent of multiple timeout clauses is possible using assignment in the timeout expression and checking inside the following statement of the timeout clause, e.g.:

```
uAccept( ... ) ...
uOr uTimeout( T = selectTime( ... ) ) // select a timeout delay
    if ( T == ... ) ...               // choice 1
    else if ( T == ... ) ...          // choice 2
```

The semantics of the μC++ accept statement differ from that in Ada and Concurrent C. For Ada and Concurrent C, multiple alternative clauses can be specified, and for multiple timeouts, the smallest time value is selected. As well, the selection policy among open accept clauses depends on the entry queueing policy, where the default policy is arbitrary selection. Both consider all open clauses before selecting a task, which is advantageous in a real-time system to select the highest-priority calling task, but incurs more overhead as all open clauses must be examined before a choice can be made. However, an accept statement is only necessary when a subset of the mutex methods are specified; hence, its use implies a controlled potential for priority-inversion. In μC++, this control is carried further to the textual order of accept clauses. Complete determinism seems appropriate for most non-real-time programming, and we are evaluating its advantages/disadvantages for real-time programming.

3.2 μC++ Timeout Design & Implementation

In order to implement a timeout facility, the system must provide a notion of time, such as a clock with time values, and a timer operation, which generates an interrupt after a specified time or delay. While a timer operation can be constructed using a loop and an operation to get the current time, this method is inefficient and inaccurate.

There are many different ways to design a timeout facility. One key consideration is that the facility fit naturally into the semantics of the operation being augmented, making the facility easier to understand and allowing the operations to be terminated in a graceful manner. For μC++, modelling the timeout after a call to a mutex method (*timeout method*) has the advantage of integrating the timeout facility more naturally with existing accept-statement semantics and making the implementation simpler as existing accept-statement functionality can be used. This approach resulted in a specialized, implicit timeout-method, simpler than a normal mutex-method, which is invoked when a timeout expires. The timeout method unblocks the acceptor if an eligible call has not arrived.

The problem with this approach is the need for a thread to call the timeout method. As mentioned, creating a timeout task is rejected. The approach taken in μC++ is to share the timeout processing among the executing tasks. When the timer expires, the interrupted task makes any timeout calls, where such a delay has to be factored into the execution time for tasks. The timeout calls are made directly from the interrupt routine before resetting the timer, so the calls execute at the equivalent of highest priority.

In detail, when an acceptor task begins an accept statement with a timeout clause and no immediately-acceptable mutex-methods, a time delay is registered. The μC++ kernel provides support for registering timer events through the use of an event queue and interrupts. The notion of time is provided by an operating-system timer, e.g., setitimer, and interrupt, e.g., SIGARLM. The event queue is a time-ordered list of events protected by the *event-lock* spinlock; different event types include time slice and timeout events. Each event specifies the expiry time, an event-specific *handler routine* to be invoked at that time, and a flag indicating whether the handler routine is executed with the event-queue locked or unlocked. Events are added to the list in increasing order by time; the timer is set to expire at the time indicated by the first event. When the timer expires, the currently executing task is interrupted and this task processes the event queue. The interrupted task begins by acquiring the event-queue lock, removing an expired event, and invoking its handler routine; the event lock is released before or after the handler routine depending on the event flag. This processing is repeated for all expired nodes on the event queue, as there could be more than one. Releasing and re-acquiring the event lock between the processing of each node allows other tasks to manipulate the event queue without waiting for all expired nodes to be processed, minimizing blocking time for high-priority tasks.

The timeout method, called indirectly by the handler routine, acquires the monitor *entry-lock* spinlock, which protects the entry/mutex queues and other entry data-structures. If the timeout method is ineligible, it means an eligible call has occurred, and the timeout is discarded; otherwise, the acceptor is unblocked. The entry lock is then released.

If an eligible call is accepted, the outstanding timeout must be short-circuited *before* executing another accept statement with a timeout clause by the calling or acceptor task. One approach is for the calling task to check for and short-circuit any outstanding timeout before beginning execution of the mutex method. However, this incurs a penalty for all calling tasks because it is impossible for the calling task to know if the timeout facility is being used and so it must always check if the timeout needs to be short-circuited. For example, if a mutex method appears in an accept statement with and without a timeout, the check must be inserted in the mutex method even though it is unnecessary in certain cases. This violates the objective of limiting the impact of the timeout facility when it is not being used.

The approach taken in μC++ is to delay cancelling an outstanding timeout until the acceptor unblocks, which is not a problem as the timeout call is discarded if it arrives after an eligible call begins. However, this approach fails if the caller performs an accept with a timeout clause, e.g., in the mutex method, because another timeout is registered before control returns to the original acceptor to cancel the current one. To deal with this problem, each accept statement with a timeout clause begins by checking for and short-circuiting an outstanding timeout. This approach means removing the timeout node must be idempotent so additional attempts to remove it do not cause an error. The cost of an idempotent remove is negligible and all idempotent removes are isolated to accept statements using the timeout facilities, so there is no additional cost when timeouts are not used.

Care must be taken when locking the event and entry locks to prevent deadlock. Deadlock can only occur when two tasks acquire at least two locks in alternate order. There is only one case where this occurs: when a caller executes an accept statement with a timeout clause and the outstanding timeout call associated with its acceptor expires:

	caller at accept statement	timeout call
1.	acquire and release event lock to remove timeout	acquire event lock to remove event node
2.	acquire entry lock to process accept statement	call timeout method with event lock acquired
3.	acquire and release event lock to register timeout	acquire entry lock to examine monitor
4.	release entry lock	release entry & event lock

Before processing the accept statement, the caller acquires the event lock and removes the timeout, which delays the processing of expiring events until the node is removed. If the timeout call arrives, it arrives with the event lock acquired, which delays the caller at the start of the accept statement until the node is removed. By ensuring there is never an outstanding timeout before an accept statement with a timeout, there is no potential for deadlock.

To prevent an expensive and potentially blocking dynamic storage allocation, a timeout node is statically allocated inside every monitor. As the design guarantees there is at most one outstanding timeout event associated with a particular monitor, it is possible to use the same timeout node for every accept statement with a timeout clause in the object. Furthermore, statically allocating the node makes the remove operation trivially idempotent. The drawback is that a non-real-time monitor requires 32/64 bytes of additional storage, which seems like an acceptable tradeoff.

The only non-fixed costs of the timeout facility are adding a node to the event queue in temporal order and starvation problems related to the use of spinlocks. The costs associated with the event queue are not fixed as they depend on the number of nodes on the event queue. The number of these outstanding events are application dependent, and can be limited at the user level and incorporated into the schedulability analysis. As the need for spinlocks with a multiprocessor implementation is unavoidable, this is a general issue and not specific to the timeout facility. However, several approaches to bound the execution time relating to spinlocks have been proposed [39]. Finally, by modelling the timeout after a call to a mutex method, the changes required to support the timeout facility are small.

Timeouts for tasks waiting on condition variables, as in Java, are problematic because multiple timeouts can arrive at any time, unrelated to the active monitor task. Therefore, timeout for conditions normally requires locking the condition queues and acceptor/signal stack to safely deliver the timeout, which inhibits efficiency and concurrency when the timeout facility is not used. For an accept statement, only one timeout exists at a time and there are only three tasks involved: acceptor, caller, and the task making the timeout call. As well, the monitor is inactive during the accept.

4. SCHEDULING CONSIDERATIONS

Scheduling is generally considered the most important aspect of a real-time system. The goal of scheduling is to determine whether a set of tasks can meet their specified timing requirements. Any useful scheduling algorithm must determine if a feasible schedule exists and provide an ordering of the tasks that satisfies the specified constraints.

One common approach to ordering a set of tasks is referred to as *priority-based scheduling*, where each task is assigned a priority value. In many cases, the assigned priority value has little relevance to the actual importance of the task. A task's priority is typically a function of its relative timing characteristics. Then, when the system needs to make a scheduling decision, the ready task with the highest priority is always selected. While new non-priority-based scheduling-techniques are an increasingly important part of real-time research, priority-based scheduling is still an important research area as most commercial real-time systems are based on priority scheduling. This section considers some of the practical issues a system must deal with in order to dispatch a set of tasks using priority-based scheduling.

Not only is priority-based scheduling simple to implement, but it is flexible enough to support a variety of *static* and *dynamic* scheduling algorithms (for details see [11, 24, 36]). With static scheduling, decisions are based on the entire task set, while with dynamic scheduling, decisions are based only on the current task set. In most cases, the differences between static and dynamic algorithms lie in the ability to handle *aperiodic* tasks. Many real-time tasks tend to repeat the same set of actions with a specific frequency, e.g., reading a set of sensors once every minute. These *periodic* tasks lend themselves well to static analysis and scheduling. Aperiodic tasks may still have timing constraints but tend to have unpredictable (dynamic) arrival times. A real-time task has a *hard deadline* if the consequences of missing the deadline are severe, and a *soft deadline* if missing the deadline is not disastrous. Ideally, a static scheduler should miss no deadlines; a dynamic scheduler should miss no hard deadlines but still provide efficient service for soft deadlines.

Static and dynamic scheduling algorithms usually have an *off-line* and *online* component. The off-line component ranges from producing a fixed schedule *a priori* for static scheduling to calculating appropriate task scheduling parameters for dynamic scheduling. The on-line component ranges from dispatching the tasks according to a fixed schedule with static scheduling to determining if a new task can be scheduled with dynamic scheduling [36].

Scheduling decisions are based on a variety of constraints and criteria, e.g.: 1) period: inter-arrival time between successive occurrences of the same task, 2) computation: worst-case execution-time for an instance of the task, 3) deadline: time by which an instance of the task must be completed, 4) importance: value indicating the relative importance of the task, 5) start: time at which the task must begin execution.

Priority-based scheduling algorithms are classified as either *fixed-priority* or *dynamic-priority* scheduling algorithms. This classification is independent of whether the algorithm is used statically or dynamically. With a fixed-priority scheduling algorithm a task's priority is fixed during run-time, whereas a task's priority can change as it executes with a dynamic-priority scheduling algorithm. These definitions are somewhat misleading, because in practice, a task's priority can change in an online fixed-priority scheduling algorithm as tasks are added and removed from the system. However, if this algorithm is clairvoyant, these changes can be accounted for and each task could be assigned a fixed priority value. Even with a clairvoyant dynamic-priority scheduling algorithm, a task's priority typically changes during its execution for other reasons (see Section 4.2).

4.1 Fixed-Priority Scheduling

One of the first fixed-priority scheduling algorithms is the *rate-monotonic algorithm* [28]. The three primary requirements for a task to be scheduled by the rate-monotonic scheduling algorithm are: 1) periodic with the deadline equal to the end of the period, 2) independent, i.e., no communication or synchronization, 3) preemptable. The rate-monotonic algorithm assigns higher priorities to tasks with shorter periods. Schedulability tests are available for the rate-monotonic algorithm, but tasks not satisfying these tests may still be schedulable.

The *deadline-monotonic algorithm* [26] can be used to schedule a task with a deadline less than its period. With this algorithm, tasks with shorter deadlines are assigned higher priorities. Several schedulability tests are available for the deadline-monotonic algorithm [2], but tasks not satisfying these tests may still be schedulable.

Unfortunately, these scheduling algorithms do not consider aperiodic tasks. Several approaches have been proposed to deal with these kinds of tasks. Two common approaches are aperiodic servers and slack stealing algorithms. An *aperiodic server* is a periodic task, but its execution time is used to service aperiodic tasks. Various types of servers exist, including the deferrable server [25], the priority exchange server [25] and the sporadic server [35]. *Slack stealing algorithms* [15] try to find time to execute aperiodic tasks by delaying the execution of periodic tasks as long as possible, without causing any periodic task to miss its deadline, and using the recovered time for aperiodic tasks.

4.2 Dynamic-Priority Scheduling

The most common dynamic-priority algorithm is the *earliest deadline first (EDF) algorithm* [28]. This algorithm can be used to schedule a set of independent periodic or hard aperiodic tasks. With this algorithm, the task with the closest deadline at any given point in time is assigned the highest priority. The EDF algorithm has been shown to be optimal in the uniprocessor case [28].

The *least slack time algorithm* [30] is another common dynamic scheduling algorithm. Slack time is the measure of the amount of time a task can be delayed before it misses its deadline. With this algorithm, the task with the smallest slack time is executed first. This algorithm is also optimal in the uniprocessor case.

The problem with these dynamic algorithms is that under transient overload conditions, unpredictable behaviour can occur, resulting in a potential cascade of missed deadlines. Furthermore, most practical real-time scheduling problems are NP-hard [30, 36], such as tasks with arbitrary precedence constraints, multiprocessor scheduling, etc. In order to use dynamic-priority scheduling under these circumstances, priorities are assigned using heuristics. Common heuristics used to assign priorities are given in [24], e.g., EDF, minimum processing time first, etc.

While the algorithms described above are appropriate for servicing periodic and hard aperiodic tasks, they tend to be too restrictive when dealing with soft aperiodic tasks. Various techniques have been proposed to allow these algorithms to provide efficient service to aperiodic tasks with soft deadlines while still meeting all hard deadlines [17, 22]. Again, aperiodic servers and slack stealing algorithms are used to service these types of tasks but because the active task set may change, these algorithms must be more flexible and adjust as new tasks enter the system.

4.3 Implementing Priority-Based Scheduling

Aside from the theoretical limitations imposed by the various priority-scheduling algorithms, many practical issues exist. To achieve predictability, all online scheduling operations must be bounded by a fixed, worst-case execution-time. Fixed, worst-case execution is typically achieved by bounding the number of tasks, the number of priority levels, or some other parameter of the scheduling algorithm. Bounding the system overheads incurred by scheduling allows these costs to be included in the feasibility analysis of the system and to enhance predictability. While achieving predictability is reasonable for fixed-priority scheduling, the runtime overheads incurred for dynamic-priority scheduling are much greater, making them problematic.

4.3.1 Implementing Fixed-Priority Scheduling

For fixed-priority scheduling, scheduling analysis takes place *a priori* so each task can be assigned a static priority value from a fixed range. Typically, the priorities are sorted using the appropriate criteria for the scheduling algorithm and tasks assigned priorities consecutively starting at one. The number of priority values supported by the system is usually limited; 32 and 256 are common ranges. To efficiently schedule an eligible task with the highest priority, tasks are placed on a priority queue. As the number of priorities is typically small, it is possible to use an *array-based priority queue*. Each element of the array is the head of a FIFO queue for the priority value corresponding to the element's array index. Tasks of a particular priority value are placed on the appropriate FIFO queue.

The basic array-based priority queue offers efficient, constant time operations, where the necessary operations are: Empty, Insert, Delete, and Max/Min. The Empty operation searches the array for a non-empty priority queue in constant time, $O(p)$, where p is a fixed number of priorities. Using a doubly-linked list for the FIFO queues allows the cost of Insert to be $O(1)$, and the same for Delete if the task being removed from the queue maintains a reference to its associated node, otherwise a search of the FIFO queue is required. Finally, the Max/Min operation searches the array for the highest-priority non-empty FIFO queue in constant time, and the cost is $O(p)$.

4.3.2 Implementing Dynamic-Priority Scheduling

With a dynamic-priority scheduling algorithm, scheduling decisions are typically based on the current task-set. When a new task enters the system, its calculated priority may be between existing task priorities. If priorities are assigned consecutively, existing task priorities must be re-shuffled. Furthermore, with algorithms such as EDF, priorities are updated after each task's period ends. Constantly sorting the task priorities and assigning new values is impractical.

There is also a *priority-queue synchronization problem*, i.e., the various priority-queue data-structures required to schedule these tasks must also be updated. Priority-queue data-structures occur in the ready queue, as well as other concurrency constructs, like semaphores and monitors, to expedite entry of high-priority tasks. Finding these priority queues can be difficult and updating is expensive. As well, some high-level constructs may have internal priority queues that cannot be modified without changes to the construct, e.g., monitor condition queues. This problem is further exacerbated because the worst-case execution-time for modifying task priorities and updating the required priority queues must be incorporated into the execution-time analysis of the system. Accounting for these overheads can lead to overly pessimistic worst-case execution-times, reducing the schedulability of potential task sets. Interestingly, the update problem associated with the priority queues can be eliminated by not using an array-based priority queue.

Consider how the priority values in a dynamic system change. If a task is removed from the system, the priority values might be adjusted to remove the gap in priorities but the ordering of the remaining tasks is unchanged. Similarly, if a task is added to the system, the priorities of the current tasks may need to shift in order to accommodate a new priority, but the relative ordering of the current task set remains unchanged. Also, as the added task has not begun executing, it has not yet been placed on any priority queues. In both of these cases, the relative order of the tasks on the various priority queues remains unchanged. Therefore, a priority-queue data-structure that allows the actual key values to change relative to one another without requiring the data structure to be updated is most appropriate for dynamic-priority scheduling, e.g., a heap [40]. While such a data structure may resolve the priority-queue synchronization problem, it does not address the problem that the task priority values may be continually changing.

One scheme to solve the *shifting priorities problem* is to assign priority values that do not need to change when new tasks enter the system. One way to accomplish this is to space out the task priorities, e.g., rather than assign priorities consecutively, gaps are left throughout the entire range. This spacing allows tasks entering the system to be assigned priority values between currently existing tasks without needing to re-shuffle the existing task priorities. Unfortunately, spacing out the priority values cannot completely eliminate the need to re-shuffle priorities because the spaces between the tasks eventually fill. While this approach works well for a small number of tasks entering and leaving a system, it fails quickly when task priorities shift as part of the scheduling algorithm, e.g., EDF.

Another scheme is to assign task priority values based on the actual characteristics used to order the task set. For example, with EDF, assign the actual deadline value as a task's priority rather than sorting the deadline values and assigning an artificial number. It is still desirable to limit the number of different priority values, but allow the actual priorities to range over a much larger set of values, e.g., 256 values out of 2^{32}. Hence, the number of priorities assigned is sparse compared to the size of the range.

With this approach, a task's deadline is independent of other tasks in the system as new tasks entering the system can be assigned priorities without affecting the priorities of existing tasks. Furthermore, when a task's priority is updated, the update can be performed independently of the other tasks in the system. For example, with EDF, while the implicit priority, i.e., the relative order, of the tasks may increase when the highest-priority task finishes its execution, the actual priority values for these tasks remain unchanged.

The problem with this scheme is the possibility of overflow as the actual priority values can become large over time. This problem can be mitigated either by periodically reducing all task priorities by an equal amount or by using a large range to make this extremely unlikely. For example, if deadlines are specified in microseconds, a 64 bit value is often sufficient and practical with newer hardware.

Both schemes require a priority-queue data-structure that supports constant-time operations and takes advantage of the sparse nature of the priorities. Maheshwari [29] performed a thorough evaluation of priority-queue data-structures to determine which algorithms are appropriate for a real-time environment. The results indicate that rings, heaps, D-trees and bit vectors are most suitable for a real-time environment. These data structures are considered for their applicability to the priority scheme described above:

Ring is a circular list of priority ordered nodes. While this list can be implemented with arrays, for best performance it should be a doubly linked-list. While Max/Min is constant time, insert and delete have $O(n)$ worst-case execution-time, where n is the number of nodes, which is unacceptable for reasonably sized queues.

Heap is a complete binary tree such that every parent node has higher priority than its children [40]. A heap is typically implemented using an array, with the root of the tree as the first element of the array. Max/Min is constant time, and insert and delete have $O(\lg n)$ worst-case execution-time. Efficient implementations yielding reasonably good performance exist for heaps.

D-tree is an extension of a heap [29]. It is a complete binary tree, but the leaves are the elements of the priority queue and the interior nodes form a binary decision tree; hence, every parent node is assigned the higher value of its two children. A D-tree can also be implemented using an array, with the root of the tree as the first element of the array. As elements are inserted and deleted from the leaves of the D-tree, these changes are propagated up the tree. Max/Min is constant time, and insert and delete have $O(\lg n)$ worst-case execution-time. According to Maheshwari, D-tree performance is somewhat faster than a heap, but requires about twice the storage and is more complicated to implement.

Bit-vector can range from a simple bit map to a Van Emde Boas [38] priority queue. The simple bitmap usually consists of a separate bit vector representing the queue for each priority level and each task being assigned a particular bit in each bit vector. With bit-vector algorithms, at least one bit is allocated for each element in the range, which is unacceptable for a large, sparse range.

In Maheshwari's work, the elements of the priority queues are nodes representing tasks, making the algorithm bounds dependent on the number of tasks in the system. If the system supports a large or arbitrary number of tasks, this results in large worst-case execution-times.

The approach presented here is based on an idea similar to that presented for the array-based priority queue, where the elements in the priority queue represent FIFO queues for a particular priority level rather than tasks. Thus, there is exactly one node in the priority queue for each priority level, and tasks are added and removed from the appropriate FIFO queue. The advantage of this approach is that the worst-case execution-times for the priority-queue operations are based on the number of priorities and not on the number of tasks. Typically, a system supports a small fixed number of priorities but can support a large number of tasks. The worst-case execution-time, in this case, is a small, fixed value and much better suited for a real-time system.

In the cases where the scheduling algorithm requires each task to have a unique priority, using the FIFO queues as the nodes on the priority queue is slightly more expensive than using the actual tasks. If tasks do not have unique priority values, this method has the advantage that it is *stable*, i.e., all the tasks with a particular priority value are processed in FIFO order. When the actual tasks are used as the elements of the priority queue, many of the priority queue algorithms are unstable.

Of the priority-queue data-structures described above, not all of them are appropriate for use with the suggested approach. First, the $O(n)$ worst-case execution-time for rings eliminates it from consideration when more than a small number of priorities are supported. As well, the sparse usage of the large priority range associated with the suggested approach makes bit vectors impractical because at least one bit must be allocated for every value in the range. Thus, it seems heaps or D-trees are the best choices for implementing the suggested scheduling technique.

4.4 μC++ Scheduling Implementation

To provide a general mechanism for scheduling, μC++ provides an administrative grouping mechanism, called a *cluster*, to restrict the execution of a task set to a number of processors. A cluster is generalized with a *scheduler* to select tasks from the task set to execute on the processors; the default scheduler for non-real-time execution is round-robin. The runtime environment can be composed of multi-

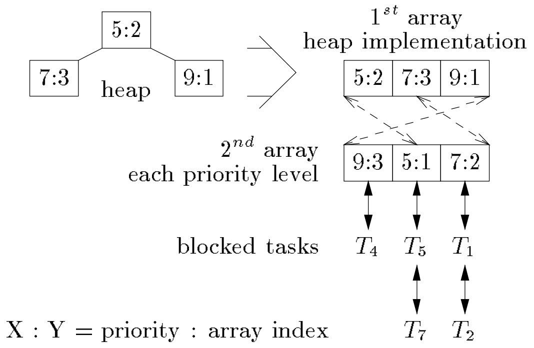

Figure 5: 2 array FIFO queue heap.

ple clusters, and tasks and processors can migrated among clusters dynamically. In general, migrating real-time tasks is problematic when priority dependence schemes are used (see Section 5).

For real-time applications, the scheduler is easily replaced, and several schedulers are available in the μC++ scheduler library. For off-line scheduling, we are building an analysis tool (not discussed in this paper) to help select from different scheduling techniques to satisfy the requirements of a task set. The resulting scheduler is often very simple because the cost of off-line analysis can be large and the task set is static, which covers many hard real-time problems. For on-line scheduling, we are building a class of schedulers for soft real-time problems, which are becoming more common, dealing with quality-of-service issues like Internet audio/video. These schedulers are characterized by the ability to quickly schedule tasks entering and dispatch tasks within the task set, and can be generalized by the data structures used in the implementation.

This section discusses off-line schedulers by examining a particular heap-based priority-queue data-structure we developed for dynamic schedulers, like deadline rate-monotonic. While the heap may support $O(\lg n)$ insertion and deletion, these operations require a reference to the actual node in the heap. As the heap does not support an efficient find operation, i.e., it has $O(n)$ worst-case execution-time, a faster method of finding a particular heap node is required. This operation is needed for inserting and deleting elements in the heap, and to allow a task to access its associated FIFO queue. Unfortunately, it is unreasonable for each task to maintain a reference to the heap node containing their associated FIFO queue because, as the heap is modified, updating these references is expensive.

In order to minimize this update problem, two array data structures are used (see Figure 5). The first array is the actual heap, but the elements of the heap index the nodes in the second array. Each element of the second array consists of a FIFO queue and an index back to its associated position in the first array. Each FIFO queue in this array is assigned to a particular priority value. Then, as the heap is updated, only the indexes in the second array need to be maintained. For any element on the heap, its index into the second array is fixed while on the heap, so no updating is required, i.e., the FIFO queue used for a particular priority level does not change while tasks for that particular priority value still exist. Thus, a task only has to maintain an index to the node in the second array associated with its priority value. Through this node, the task can access both its associated FIFO queue and its node on the heap in $O(1)$ time. Similar to the reasoning explaining why the elements in the first array do not need to be updated, the indexes in the tasks also do not need to be updated.

Unfortunately, this technique does not eliminate the find problem. When a task enters the system, it is assigned a priority value. After being assigned a priority value, this task needs to determine which FIFO queue its priority is associated with. A task can make this determination by performing a linear search through the assigned nodes in the second array. If a node associated with the required priority value is not found, then the next free node in the second array is selected and associated with the task's priority value. All subsequent tasks with the same priority value use this node as well. While this search is $O(p)$, where p is the number of priorities, it is usually only necessary when a task is created or its priority changes. Therefore, even though this technique does not eliminate the find and reference problems, both are reduced to reasonable levels, in relatively nonessential situations. Furthermore, the expected search cost can be reduced by using techniques like hashing to associate priorities to nodes in the second array.

As a task has a reference to the node associated with its priority level in the second array, the associated node in the heap can also be accessed without searching. While the ability to directly reference the heap node has no advantages when a task is inserted into the priority queue, it can have advantages during deletion. When a task is inserted into the priority queue, first, the task is placed on the FIFO queue associated with its priority value. If this FIFO queue is empty, then a node for this priority value must also be added to the heap. Similarly, when a task is deleted from the priority queue, first, the task is removed from its associated FIFO queue. If this FIFO queue is now empty, then the node for this priority level must also be removed from the heap. As the task can access the heap node directly, removing the node from the heap is an $O(\lg p)$ operation. Maintaining the heap in this manner, still allows it to return the nonempty FIFO queue of highest priority in constant time. Of course, if this FIFO queue of highest priority is empty after a task is removed, then an $O(\lg p)$ delete operation must be performed on the heap.

The problem in many systems is that several priority-queue data-structures exist. A priority queue is associated with the ready queue, as well as with real-time *mutex objects*, e.g., semaphores, monitors and tasks. It is infeasible for a task to remember the index associated with its priority level for all the possible priority queues in the system. Interestingly, it is possible for every task in the system to simply use the same index value regardless of the priority queue. If this ordering is consistent for one priority queue, then as long as the same tasks and priority levels are used for all other queues, then this ordering is consistent for all priority queues. This technique also eliminates the extra search required the first time a task is placed on another priority queue. Note that the initial search when a task is created or its priority value is changed must still occur in order for the task to determine the appropriate queue value.

In summary, μC++ provides the ability to replace the scheduler of a cluster for both non-real-time and real-time concurrency. The previous example illustrates a dynamic scheduler, which is one of the most complex kinds of scheduler available in the μC++ scheduler library; other schedulers are also provided. The key point in developing any real-time scheduler is maintaining fixed worst-case performance, and ensuring correct interaction with other priority queues in the system.

5. PRIORITY INHERITANCE

Many priority-based scheduling algorithms assume tasks are independent; however, this assumption is typically unrealistic. Shared resources are common in concurrent systems and form critical sections that must be protected with mutual exclusion. As mentioned in Section 2.2, critical sections result in priority inversion when a low-priority task in the critical section delays a high-priority task attempting entry. While, in general, it is impossible to eliminate priority inversion, it is important to bound the duration of an inversion. *Unbounded priority inversion* occurs if a low-priority task is preventing a high-priority task from executing but cannot execute itself because a medium-priority task is executing. Hence, the high-priority task waiting for the critical section may never make progress. Unbounded priority inversion is a serious problem, making it impossible to guarantee the schedulability of a system.

A technique to bound the length of this inversion is *priority inheritance*. The idea behind priority inheritance is to temporarily raise the priority of a task owning a resource in order to expedite its usage of the resource, thereby limiting the duration of priority inversion. The details regarding when a task's priority is raised and by how much vary depending on the particular priority inheritance protocol.

5.1 Priority Inheritance Protocols

To address the problem of unbounded priority inversion, several priority inheritance protocols have been proposed [34], and subsequently expanded and extended in various ways [3, 13, 32]. The idea behind the *basic priority-inheritance protocol* is that if a low-priority task is delaying the execution of higher-priority tasks due to priority inversion, the priority of the low-priority task is temporarily raised to the priority of the highest-priority task it is blocking. Raising the priority of the lower-priority task expedites its usage of a resource by letting it execute when the blocked higher-priority task would normally be scheduled. This technique bounds the length of any priority inversion and allows the worst-case execution-time of a task to be specified. Sufficient conditions have been found to allow a non-independent task set to be scheduled using the rate-monotonic scheduling algorithm using basic priority inheritance [32, 34]. Intuitively, this technique works because a medium-priority task can no longer preempt the execution of a lower-priority task if the lower-priority task is preventing the execution of a higher-priority task due to priority inversion. Furthermore, this scheme does not affect the execution of non-blocked high-priority tasks. Priority inheritance is also transitive, so the inherited priority value of a task is the highest priority of all the tasks *directly* and *indirectly* blocked by this task.

However, deficiencies exist with the basic protocol. First, despite the fact this protocol bounds the length of priority inversion, the actual blocking time experienced by tasks can be long because multiple blocking is not prevented. *Multiple blocking* occurs if a task needs to block each time it requires another resource because the resource is already acquired by a lower-priority task. Ideally, all the resources a task requires to perform an operation (transaction) should be available after waiting for any one of these resources. Second, this protocol does not avoid deadlock, an interesting side effect provided by some of the more sophisticated priority inheritance protocols (at the cost of reducing concurrency). The biggest advantage of the basic protocol, however, is that it can be implemented without requiring any additional system information. Thus, it works well with online scheduling.

The priority ceiling [32, 34] and immediate priority ceiling [4, 32] protocols deal with some of these deficiencies, but also introduce others. In particular, priority ceiling is difficult and costly to implement, while immediate ceiling is simple to implement but still has a form of priority inversion. Essentially, there is no perfect protocol for all situations and a programmer has to select the one that best suits the application. Real-time systems often provide basic and/or immediate ceiling protocols (e.g., Ada, POSIX, JavaRT), and a programmer selects them for use in an application.

5.2 Implementing Basic Priority Inheritance

The basic priority inheritance protocol is examined for the following reasons. First, it forms the base for most of the more complicated protocols. Second, a complete implementation is not straightforward.

The first step in implementing priority inheritance is to extend the notion of a task's priority. Typically, two priority values are associated with each task: a *base priority* and an *active priority*. A task's base priority is the priority value assigned by the scheduling algorithm and a task executes at this priority value when no priority inheritance is occurring. A task's active priority is the maximum of a task's base priority and the priorities of all the tasks it is blocking. A task is always executed at its active priority. The second step, in any complete implementation, is to address transitivity and priority disinheritance.

5.2.1 Transitivity

With priority inheritance, when a task blocks because a shared resource is unavailable the primary goal is to raise the priority of this task's *ultimate blocker*, i.e., the final task in the task's blocking chain. For all the tasks in Figure 6, task T_7 is the ultimate blocker. However, a secondary goal is to keep the active priority of other tasks in the blocking chain updated. This extra updating is useful not only for priority disinheritance (see below), but also to manage the priority queues these other tasks are blocked on. Without this updating, scheduling decisions are expensive because stale information in these priority queues needs to be reevaluated. For example, if a high-priority task $T_1 : 1$ blocks on R_1 in Figure 6, not only must the priority of T_7 be raised to 1, but also the priority of T_8 and its position on the entry queue of R_3 must be updated, i.e., moved to the front.

The straightforward approach to implementing transitivity has each task point to its direct blocker, so it is possible to follow a chain of blocked tasks to a task's ultimate blocker.

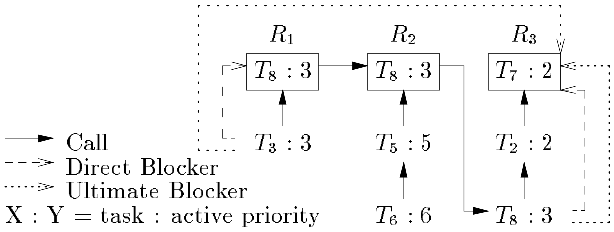

Figure 6: Transitivity

For example, in Figure 6, task T_3's direct blocker is T_8, task T_8's direct blocker is T_7, and T_7 is not blocked; T_7 is T_3's and T_8's ultimate blocker. Then, when a task blocks while trying to acquire a resource, it can use this information to update the active priority of the tasks in its blocking chain until a task with a higher active priority or the ultimate blocker is reached. As the active priority of a blocked task is updated, its position on the priority waiting queue must also be updated. If the resource maintains a reference to its current owner, a task can indirectly determine its direct blocker by simply remembering the resource it is trying to acquire. As long as the identity of this resource remains fixed while a task is blocked, no updating is required regardless of whether the task owning the resource changes.

Alternatively, a task points to its ultimate blocker; however, this approach suffers from two problems. First, when a task blocks on a resource, the ultimate blocker for all the tasks on its blocking chain must be updated from the blocking task to the ultimate blocker associated with this resource. In order to update these tasks, the blocking chain must be traversed. For example, in Figure 6, suppose T_7 releases R_3 and T_8 acquires this resource (remember, T_1 is now blocked on R_1). In this case, T_8 must be able to find all the tasks blocked on R_1, R_2 and R_3 to update their ultimate blocker from T_7 to T_8. Second, maintaining a task's ultimate blocker is much more expensive than maintaining a task's direct blocker. For example, when T_7 releases R_3, only the direct blocker of T_2 and T_8 changes, as opposed to the ultimate blocker for *all* the tasks. Essentially, the ultimate blocker and the resource associated with the ultimate blocker can change as the ultimate blocker acquires and releases resources, so there is no fixed way to locate a task's ultimate blocker.

5.2.2 *Priority Disinheritance*

Priority disinheritance is determining a task's priority when it releases a resource [31]. If a task is inheriting its active priority from a task blocked on the resource it is releasing, it must determine its new active priority based on tasks blocked on the resources it still owns. This section discusses two common techniques for solving priority disinheritance.

In the first technique, each task stores its old priority value when it acquires a resource. When a task exits a resource, its priority is restored to this stored value. This creates a stack of values for restoring a task's priority as it exits each resource. Problems exist with this idea, however. First, the use of a stack implies resource usages are nested, making it difficult to release resources in arbitrary order. For example, overlapping critical sections using semaphores may not be permitted with this approach, e.g., acquiring semaphore S_1 followed by semaphore S_2 and then releasing S_1 followed by S_2. The reason is that if a resource is removed from the middle of the stack, then restoring the task's active priority to the stored value associated with this resource is inappropriate because resources higher on the stack are still affecting the task's active priority. Second, the stored priority values can become stale, e.g., a task already owning several resources subsequently experiences priority inheritance from a resource lower down on the stack. In this case, a task's active priority is subsequently reset to a stale value after it releases a resource. To solve this problem, when the highest priority blocked task associated with a resource changes, the priority value associated with the next resource on the stack needs to be updated, and this update needs to propagate up the stack until a higher priority value occurs or the top of the stack is encountered. Finally, the stack approach is inefficient. Even if a task's active priority does not change, a significant portion of the stack may need to be updated so that a task's priority is correctly reset as it releases resources. As well, this update is dependent on the number of resources directly or indirectly blocked by the ultimate blocker, as opposed to the usually smaller blocking path defined by a chain of direct blocker tasks. The task blocking chain is usually smaller than the resource blocking chain because each direct blocker task can own several resources. Another problem occurs if resources support recursive entry, i.e., a task is allowed to call back into a resource it already owns, where this recursive call can occur after a task has acquired and released other resources. In this case, multiple values may need to be stored for each entry, eliminating the possibility of statically allocating space to store the current priority of the owner task inside the resource.

The second technique requires a task to maintain a list of resources that it owns. Then, when a task releases a resource, this resource is removed from the list, and the blocked task with the highest active priority among the remaining resources needs to be located. The running task can then set its new active priority to be the higher of this priority and its base priority. Some optimizations are available, such as a task only needs to find a new inherited priority if the task's old inherited priority is equal to the priority of the highest-priority task blocked on the resource it is releasing. This optimization is possible because if the priority of the highest-priority task associated with the resource a task is releasing is not equal to the task's active priority, then the task is inheriting its priority from a resource it still owns, and hence, no adjustment is required. Additionally, each node on the list can explicitly store the priority of the highest-priority task associated with the resource and the list can be implemented as a priority queue. Using this technique has several advantages. First, if a task's direct blocker already has a higher active priority, no priority propagation is required as the resources owned by a task are updated independently. However, if the list is implemented as a priority queue, it also needs updating. As well, only the direct blocker tasks need to be updated if priority inheritance occurs. Finally, resources that support recursive calls can be handled with variables statically declared in the resource because the list node is the same for every entry into a resource; thus, only one node needs to actually appear on the list. The only additional complexity is that the list node should only be removed when the task finally releases the resource and not on one of the interim exits.

5.3 Related Work

Several implementations have been proposed for basic priority inheritance. Unfortunately, the efficient solutions rely on simplifying assumptions that are unreasonable or do not implement the correct semantics of basic priority inheritance. This section discusses some of these approaches.

Borger and Rajkumar [6] describe an implementation of the basic priority inheritance protocol for task rendezvous in Ada'83. Their solution to transitivity is similar to the method described above, i.e., each task follows its blocking chain updating tasks as required. However, the implementation only supports task rendezvous, which simplifies priority disinheritance because each task owns exactly one resource, i.e., itself. Therefore, its active priority after disinheritance is simply the highest priority of the tasks blocked on any of its entry queues and its own base priority.

Two interesting implementations of basic priority inheritance are described by Moylan, Betz and Middleton [31]. The first implementation provides a general solution to basic priority inheritance. In this solution, each task maintains a count of the number of tasks it is directly blocking at each priority level. These counts form a priority queue containing all the tasks directly blocked by a particular task. This task's active priority is then equal to the highest priority level with a non-zero count. However, when a task exits a resource, the counts associated with all the tasks blocked on that resource must be decremented for the exiting task and incremented for the new owner of the resource. While this represents a significant overhead, some simplifications are possible if assumptions are made about the order blocked tasks are scheduled. The most interesting feature of this implementation is that it allows the tasks blocked on a resource to be unblocked in arbitrary order, e.g., FIFO or priority order. This arbitrary ordering is possible because the counts for each priority level contain information relating to all directly blocked tasks. Thus, regardless of the order tasks are actually scheduled, the priority queue associated with each task determines its active priority.

The second implementation by Moylan, Betz and Middleton assumes tasks block only because of priority inversion. With this assumption, any running task must be running at the priority of the highest-priority task in the system. This assertion is true because either the highest-priority task is running or it is blocked and so its ultimate blocker is running at this highest priority value. With this implementation, the disinheritance problem is eliminated because the running task is always executing at this highest priority value. However, it creates a scheduling problem as the active priority of a task is not stored. This problem is overcome by simply scheduling the highest-priority task or if this task is blocked, following the blocking chain of the highest-priority task and executing this task's ultimate blocker. This overhead is not excessive because, even in the general case, a task's blocking chain is typically traversed when it blocks on a resource. The blocking restriction imposed by this algorithm, however, is too limiting, as tasks cannot block on delays, accept statements, etc. Hence, this algorithm is inappropriate as a general solution to priority inheritance.

Takada and Sakamura [37] present a restricted and general implementation of priority inheritance. The restricted form assumes a task releases all its resource at once, which is too restrictive for most applications. The general form has a list of resources acquired by each task, similar to the approach in the previous section. However, the details regarding how this list is used for priority inheritance are not provided.

5.4 μC++ Priority Inheritance Implementation

Most systems provide only two or three predefined inheritance-protocol mechanisms. To support extant and new inheritance protocols, a general mechanism is needed. First, it is necessary to have specialized real-time tasks. In addition to the basic task type-generator, μC++ supplies three kinds of real-time tasks integrated into C++ classes: periodic, aperiodic, sporadic.

```
uTask task { ... };                    // basic task type
uPeriodicTask ptask { ... };           // real-time task types
uAperiodicTask atask { ... };
uSpordicTask stask { ... };
```

The constructors for these real-time tasks take period and deadline information, which is implicitly communicated to the scheduler. For periodic tasks, the periodic behaviour is performed implicitly by the scheduler. Implementing periodicity at the user level is neither sound nor appropriate for certain schedulers (e.g., slacking-stealing schedulers). Second, a real-time task points to a priority queue, called the PIQ, containing the priority inheritance information for each mutex object (resource) it owns. The PIQ type can be passed to a real-time task type like the entry and mutex queues for a monitor:

```
uMutex<EQ,MQ> uPeriodicTask<PIQType> ptask { ...
```

Here, type ptask is passed all three queues, and these three queues work in concert to provide real-time and inheritance capabilities. If no PIQ type is specified, the default is a modification of the heap priority-queue from Section 4.4, and the default EQ and MQ are heap-based priority-queues. The following discussion assumes these default values. (Be forewarned that the following discussion is complex because the problem is complex.)

The default PIQ information consists of the maximum of a task's base-priority and the priority of the highest-priority task blocked on each mutex object owned by a task. As with the scheduling heap, only one node for each required priority level appears in a task's PIQ, and this node references a node in a second array containing the details for that particular priority level (see Figure 5). However, the FIFO queues of blocked tasks in the second array are replaced by a count of the number of tasks and the queue number associated with the priority level, e.g.:

2^{nd} array	9:3	5:1	7:2	
each priority level	1	2	2	← counts

No explicit reference to the highest-priority task associated with each resource is required as these tasks are neither accessed nor scheduled using the PIQ. The only relevant information is the priority and associated queue value (as per Section 4.4) so a task's active priority can be calculated. The count value determines when a node can be removed from the PIQ. A zero count value corresponds to an empty FIFO queue and indicates the node is to be removed from the PIQ because a task is no longer eligible to inherit that particular priority value. This priority queue solves the priority disinheritance problem because after a task releases a mutex object and updates its PIQ, its active priority becomes the highest priority remaining on its PIQ. Finally, a task is not the sole updater of its PIQ; other tasks may need to update another task's PIQ during priority inheritance. Therefore, access to a PIQ needs a lock to provide mutual exclusion.

As mentioned in Section 2.3, priority inheritance for mutex objects in μC++ is implemented using hooks invoked when a task: acquires a mutex object, blocks on an entry call, and releases a mutex object. In the following discussion, the task performing inheritance is called the *updater*, while the task in the mutex object is called the *owner*. Note, the updater and the owner are the same when a task acquires the mutex object without having to block, i.e., the mutex object is inactive. In all other cases, the updater and the owner refer to different tasks. Lastly, each mutex object maintains a pointer to its current owner task.

5.4.1 Mutex Object Acquire

A task acquires a mutex object when it enters an inactive object or is unblocked from the entry queue after the current owner exits from or blocks in the object. In both cases, the new owner acquires the mutex-object entry-lock to access the entry variables, and executes the uOnAcquire hook (this action changes in Section 5.4.3), which acquires the PIQ lock, adds the new resource, and then releases the PIQ lock. Finally, the entry lock is released.

5.4.2 Entry Blocking on a Mutex Object

A calling task blocks on a mutex method when the mutex object is active or the mutex method is ineligible. In this case, the blocking task becomes an updater and may need to raise the priority of the current mutex owner. Rather than providing an explicit hook for this circumstance, any necessary code can be added to the end of the uAdd routine, which adds the calling task to the entry queue.

The updater, having acquired the entry lock to determine it must block, places itself on the entry/mutex queues in priority order. If its priority is greater than the maximum of the mutex-owner's base-priority and the active priority of the entry blocked tasks, it must perform priority inheritance. In this case, the updater acquires the PIQ lock of the owner to augment its PIQ (needed for disinheritance) and possibly performs priority inheritance on the owner. Augmenting the PIQ means decrementing the current priority level of the mutex object and incrementing the new priority level. After updating the PIQ, the updater releases the owner's PIQ lock. If the change to the PIQ results in a new Max/Min inheritance value, the owner's active priority must also be increased. Changing a task's active priority requires the ready-queue lock. If the owner is on the ready queue, its position may have to be updated; otherwise, the owner is running and its priority is just changed as it cannot block on the ready queue while the ready-queue lock is acquired. The ready-queue lock is then released. Note, attempting to increase the owner's active priority has no effect if the owner's priority is already higher than the update value. If the owner is entry blocked waiting to enter another mutex object, both the owner's active priority and its position on the entry queue must be updated, which requires the ready queue and entry locks, respectively. Changing the entry queue of this new mutex object can result in further priority inheritance on its owner (transitivity). (Releasing entry locks during transitivity is discussed in Section 5.4.6.) Finally, the updater atomically releases the entry lock for the initial mutex object and blocks.

5.4.3 Mutex Object Release

An owner task releases a mutex object when it exits. It acquires the entry lock to update the entry variables and possibly to schedule the next mutex owner. It then executes the uOnRelease hook, which acquires the task's PIQ lock to decrement the counter for the priority level associated with the current mutex object, removing the node if zero, and releases the PIQ lock. The exiting task then sets its active priority to the current PIQ Max/Min (disinheritance). If its priority has to change, the operation is simple because it is running. If the task's PIQ is empty after the removal, it sets its active priority to its base priority. Finally, the entry lock is released for the mutex object it is exiting. In the case where the exiting task schedules the next mutex owner, it must execute the uOnAcquire hook on behalf of the new owner. The new owner cannot execute uOnAcquire *after* it is scheduled because its priority must be raised to its proper value *before* it is scheduled, so it is scheduled at the appropriate time. Otherwise, the new owner can experience uncontrolled priority inversion while it waits to be scheduled so it can update its active priority.

Recursive entries and exits to/from a mutex object by a task are handled as special cases. On recursive entry, a task's PIQ is not updated nor is priority inheritance necessary as these operations are performed on entry and maintained by inheritance. What if a task calls another mutex object, receives further priority inheritance, and calls back again with higher priority? Still, no update is needed because the priority information for a mutex object only needs to reflect the highest priority value among a task's base priority value and the active priority of all the tasks directly blocked on that object (see Section 5.2.2). By the same reasoning, on recursive exit, the uOnRelease hook is not called; the uOnRelease hook is only called on the final exit.

5.4.4 Blocking within a Mutex Object

Unfortunately, priority inheritance affects internal scheduling. When a task unblocks from a condition variable or the acceptor/signal stack, the mutex object's owner must be reset so an updating task can correctly perform inheritance on the owner's PIQ. To safely update this variable, the entry lock must be acquired, which has a performance impact on internal scheduling and a concurrency impact on external scheduling. We could find no way around this problem.

In addition, when a task blocks in a mutex object (wait or accept), its priority is no longer applicable to the mutex object because it is unscheduled. Therefore, priority disinheritance must be performed after a task blocks to deal with this issue, which is accomplished by calling the uOnRelease hook. Similarly, when a task is unblocked from the acceptor/signal stack, the priority of the mutex object must be recalculated as well as the PIQ of the new owner. As for mutex-object release, this recalculation must be performed by the exiting or blocking owner task by executing the uOnAcquire hook on behalf of the new owner to prevent the same priority inversion problem.

5.4.5 *Problems*

Since updater and owner can execute concurrently, an interesting problem occurs when the updater has to increase the owner's priority: the owner can change states, from blocked to running or vice versa, and change the current mutex object it owns by exiting this object or calling a new one. The obvious solution is to employ a mechanism to prevent the owner from changing state or mutex object while its priority is being updated. Unfortunately, implementing such a mechanism is difficult without excessive locking, which tends to increase overhead and reduce concurrency. A further problem is determining which mutex object the owner is currently using, as it may be different from the one the updater is blocking on.

An elegant solution to these problems is obtained by introducing cooperation between the updater and owner. First, each task maintains the address of the current mutex-object it is accessing, which may be NULL. Second, the updater retains responsibility for changing the owner's PIQ and active priority, but the owner is occasionally responsible for adjusting its own active priority. The owner is required to check and possibly adjust its active priority only when it enters or leaves a mutex object by checking its PIQ, which requires the PIQ lock. Indeed, having the owner update its active priority, in these situations, is simpler and less expensive than having the updater do it. The reason is that the owner is running, and hence, no queues require adjustment. Furthermore, the owner's current mutex-object is fixed while it is updating its own priority.

The cooperation is used in the following ways to solve the problems. As before, the updater begins by updating the owner's PIQ. It then acquires the entry lock for the owner's current mutex object, but this information can be stale because the owner may have exited this object or called a new one. The updater *can* check if the owner is blocked on the entry queue of this mutex object as it has the entry lock. If the owner is blocked, it cannot be scheduled as unblocking it requires the entry lock; hence, the updater can safely change the owner's active priority and the position of the owner on the object's entry queue, and these changes may result in further priority inheritance (transitivity). If the owner is not blocked on the entry queue, it is either in the mutex object (i.e., its owner), exited or called out. In all three cases, the update proceeds by updating the owner's active priority, and then releasing the entry lock, completing its portion of the inheritance operation. If the owner has exited or called out, the new cooperation ensures the owner has updated its own active priority from the already updated PIQ. Thus, the owner has updated its own active priority, and if necessary, continues the inheritance operation should it entry block on a subsequent call. While extraneous updating of the owner's active-priority is idempotent, it does require the ready queue lock, which reduces concurrency. Notice, the priority inheritance being applied by the updater is always relevant because the updater is still holding the entry lock for the object it is blocking on, preventing the owner from backing up past this mutex object in the inheritance chain.

Finally, an updater must only attempt to increase an owner's active priority to the value it added to the owner's PIQ, even though this value may be stale because of concurrent updaters. This anomaly results from the cooperation because an owner can see the changes to its PIQ and update itself before an updater can change the active priority. If an owner calls another mutex object during an update, it can entry block using a low-priority updater's PIQ value, but a high-priority updater can change the owner's PIQ before the low-priority updater changes the owner's active priority. If the low-priority updater uses the high-priority value from the PIQ for updating the active priority, the owner's position on the entry queue is inconsistent with its active priority, which results in problems. The low-priority updater does not attempt to update the entry queue because of the cooperation. If an updater only uses its initial value during updating, the inconsistency is avoided because updating is conditional on the priority being greater. Therefore, the low-priority updater discards its updating of the active priority should the owner see a high-priority update first. Holding the entry lock during updating precludes two updates from manipulating the same mutex object simultaneously.

5.4.6 *Optimizations*

Interestingly, it is possible to proceed with inheritance without maintaining the entry lock of every mutex object in the inheritance chain. In fact, the ultimate blocker is the only task in an inheritance chain that can execute. So, every other owner in the chain is blocked waiting for the ultimate blocker to release them. However, an interesting feature of the entry lock is that it prevents a task from exiting its associated mutex object. Thus, acquiring a particular entry lock prevents tasks from backing up past that mutex object in the inheritance chain and effectively fixes the earlier portion of an inheritance chain. Therefore, entry locks associated with mutex objects earlier in the chain can be released. The advantage of this approach is that it allows tasks to still block on the entry queues of those mutex objects during priority inheritance. Finally, to maintain integrity as a task proceeds along the inheritance chain, the next entry lock must be acquired before the current entry lock is released.

However, the entry lock of the mutex object associated with the updater, i.e., the first entry lock acquired, is a special case. While releasing this lock is fine because the updater cannot be scheduled as long as an entry lock further down the chain is acquired, this entry lock must be re-acquired before the last entry lock is released. Re-acquiring this entry lock is necessary to prevent the task performing the inheritance from being scheduled before it blocks. Otherwise, as soon as the last entry lock is released, the back portion of the inheritance chain is no longer fixed and so it is possible for the updater to be scheduled before it can block. Re-acquiring the first entry lock before releasing the back portion of the inheritance chain allows this task to atomically block and release the lock as required.

Releasing the first entry lock also allows the active priority of the updater (versus owner) to increase while it is walking the inheritance chain. To maintain consistency, it is important not to switch to this updated value in the middle of the inheritance operation. The potential problem is that an owner's PIQ can be updated using one value and its active priority can be updated using a higher value. This discrepancy can lead to the owner being blocked on the entry queue at a priority value different from its active priority.

One difficulty encountered when an updater is augmenting a mutex owner's PIQ is determining the existing priority inheritance value associated with the mutex object in order to decrement the count associated with this priority. Explicitly storing the priority inheritance value used by the mutex owner, e.g., in the entry queue, eliminates re-calculating this value during subsequent inheritance.

5.5 Analysis

The algorithm described above is equivalent to the general case version of the basic-priority-inheritance algorithm by Moylan, Betz and Middleton [31]. In our case, as the entry queues are prioritized, only the task with the highest active priority needs to be counted. Otherwise, a priority queue is maintained for each task, which is used to determine a task's active priority at any given time.

The additional complexity in the μC++ implementation exists because it is tailored to dynamic-priority scheduling. In this case, a task must also remember the queue number corresponding to a particular priority and not just the priority value as this array index is needed to access the scheduling queue associated with a particular priority. However, if a fixed-priority array-based scheduling technique is used, then only the counts are necessary. (Such a scheduler is also implemented in the μC++ scheduler library.)

As the PIQ, and the entry and mutex queues are implemented based on the discussion of priority queues in Section 4.4, all the associated operations are $O(1)$. Adjusting a task's priority is also an $O(1)$ operation. Therefore, the complexity of priority inheritance is $O(k)$, for k tasks in a blocking chain, and the complexity of uOnAcquire and uOnRelease is $O(1)$. uOnAcquire retains complexity $O(1)$ despite the fact that it performs priority inheritance, because it is executed by or on behalf of the owner, so the blocking chain has size at most one.

6. CONCLUSIONS

The goal of this work is the creation of an extensible, flexible, and predictable real-time system, to encourage the use of new real-time methodology and discourage the use of adhoc approaches. Allowing users access to some of the traditionally internal data-structures of mutex objects and the runtime system provides two important advantages (see [19] for others). First, the system is extensible, allowing new ideas and theory to be tested and incorporated into applications as they become available. Second, it allows fine tuning of the system for an application, which is crucial because a real-time application can have a significant impact on the data structures and algorithms used, and vice versa.

However, there are drawbacks to this approach. A user is responsible for guaranteeing that any functionality added has a fixed worst-case execution time. Furthermore, the user is also responsible for maintaining the coherence of the system. Therefore, the goals of creating an efficient and consistent system tend to conflict with the goals of allowing the system to be flexible and extensible. In a real-time system, this tradeoff is acceptable because the user already bears a significant amount of responsibility for guaranteeing the predictability and timing constraints of their code.

The approach implemented in μC++ does not limit users to a fixed set of real-time scheduling and protocol approaches. A user can select from standard functionality already provided through existing data-structures and algorithms defined in the μC++ real-time library or augment these existing library facilities or define new ones. While the real-time attributes in POSIX and the pragmas in Ada are reasonable for dealing with priority scheduling and inversion with respect to mutex objects, there is little or no flexibility to incorporate other approaches. This limitation, however, does allow more thorough runtime checks to be incorporated into these systems and allows the implementation to be optimized for a particular scheduling strategy.

7. ACKNOWLEDGMENTS

We would like to thank Bob Zarnke for his input in developing the timeout mechanism.

8. REFERENCES

[1] G. R. Andrews and F. B. Schneider. Concepts and notations for concurrent programming. *ACM Comput. Surv.*, 15(1):3–43, Mar. 1983.

[2] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline-monotonic approach. In *Proceedings of the 1991 IFAC/IFIP Workshop on Real-Time Operating Systems and Software AND 8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 127–132, Atlanta, Georgia, U. S. A., 1992.

[3] T. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, 3(1):67–99, Mar. 1991.

[4] T. Baker and O. Pazy. Real-time features of Ada 9x. In *Proc. IEEE Real-Time Systems Symposium*, pages 172–180, 1991.

[5] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java.* The Real-Time for Java Expert Group, http://www.rtj.org. Addison-Wesley, 2000.

[6] M. W. Borger and R. Rajkumar. Implementing priority inheritance algorithms in an ada runtime system. Technical Report CMU/SEI-89-TR-15, Carnegie Mellon, 1989.

[7] P. Brinch Hansen. *Operating System Principles.* Prentice-Hall, 1973.

[8] P. A. Buhr. Are safe concurrency libraries possible? *Commun. ACM*, 38(2):117–120, Feb. 1995.

[9] P. A. Buhr, M. Fortier, and M. H. Coffin. Monitor classification. *ACM Comput. Surv.*, 27(1):63–107, Mar. 1995.

[10] P. A. Buhr and R. A. Stroobosscher. μC++ annotated reference manual, version 4.7. Technical report, Dept. of Computer Science, University of Waterloo, Aug. 1999. ftp://plg.uwaterloo.ca/pub/uSystem/uC++.ps.gz.

[11] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 1997.

[12] D. R. Butenhof. *Programming with POSIX Threads*. Professional Computing. Addison-Wesley, 1997.

[13] M. Chen and K. J. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Journal of Real-Time Systems*, 2(4):325–346, Nov. 1990.

[14] C. L. A. Clarke. Language and compiler support for synchronous message passing architectures. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 1990.

[15] R. I. Davis, K. W. Tindell, and A. Burns. Scheduling slack time in fixed priority preemptive systems. In *Proceedings of the Real-Time Systems Symposium*, pages 222–231, Raleigh-Durham, NC, U. S. A., Dec. 1993. IEEE Computer Society Press.

[16] N. H. Gehani and W. D. Roome. *The Concurrent C Programming Language*. Silicon Press, NJ, 1989.

[17] T. M. Ghazalie and T. P. Baker. Aperiodic servers in a deadline scheduling environment. *Journal of Real-Time Systems*, 9(1):31–68, July 1995.

[18] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[19] M. Haines. On designing lightweight threads for substrate software. In *USENIX 1997 Annual Technical Conference. Anaheim, CA*, pages 243–255, Jan. 1997.

[20] A. S. Harji. High-level real-time concurrency. Master's thesis, University of Waterloo, Dec. 1999. ftp://plg.uwaterloo.ca/pub/uSystem/HarjiThesis.ps.gz.

[21] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, Oct. 1974.

[22] N. Homayoun and P. Ramanathan. Dynamic priority scheduling of aperiodic tasks in hard real-time systems. *Journal of Real-Time Systems*, 6(2):207–232, Mar. 1994.

[23] Intermetrics, Inc. *Annotated Ada Reference Manual*, international standard ISO/IEC 8652:1995(E) edition, Dec. 1994. Language and Standards Libraries, V 6.0.

[24] M. Joseph, editor. *Real-time Systems, Specifications, Verification and Analysis*. Prentice Hall, 1996.

[25] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proc. IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

[26] J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation, North Holland*, 2:237–250, 1982.

[27] P. E. Lim, Jr. Real-time in a concurrent, object-oriented programming environment. Master's thesis, University of Waterloo, Aug. 1996. ftp://plg.uwaterloo.ca/pub/uSystem/LimThesis.ps.gz.

[28] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, Jan. 1973.

[29] R. Maheshwari. An empirical evaluation of priority queue algorithms for real-time applications. Master's thesis, Florida State University, 1990.

[30] A. K. Mok. *Fundamental design problems of distributed systems for the hard real-time environment*. PhD thesis, MIT, 1983.

[31] P. J. Moylan, R. E. Betz, and R. H. Middleton. The priority disinheritance problem. Technical Report EE9345, The University of Newcastle, 1993. ftp://-ee.newcastle.edu.au/pub/reports/Disinheritance.ps.Z.

[32] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[33] D. C. Schmidt and F. Kuhns. An overview of the real-time CORBA specification. *IEEE Computer*, 33(6):56–63, June 2000.

[34] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, Sept. 1990.

[35] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Journal of Real-Time Systems*, 1(1):27–60, June 1989.

[36] J. A. Stankovic, M. Spuri, M. Di Natale, and G. C. Buttazzo. Implications of classical scheduling results for real-time systems. *Computer*, 28(6):16–25, June 1995.

[37] H. Takada and K. Sakamura. Experimental implementations of priority inheritance semaphore on itron-specification kernel. In *Proceedings of 11th TRON Project International Symposium*, pages 106–113. IEEE Computer Society Press, Dec. 1994.

[38] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[39] C.-D. Wang, H. Takada, and K. Sakamura. Priority inheritance spin locks for multiprocessor real-time systems. In *Proceedings of the 1996 International Symposium on Parallel Architecture, Algorithms and Networks (ISPAN'96)*, pages 70–76, June 1996.

[40] J. W. J. Williams. Algorithm 232: Heapsort. *Commun. ACM*, 7:347–348, 1964.