High-Level Real-Time Concurrency

by

Ashif S. Harji

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2000

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

The primary goal of all real-time systems is predictability. Achieving this goal requires all levels of the system to be well defined and have a fixed worst-case execution time. These needs have resulted in the creation of overly restrictive commercial real-time systems 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 requires a more flexible and extensible approach. By giving the real-time programmer access to the underlying system data structures, the programmer is able to interact with the system to easily incorporate new ideas and to fine tune the system for a particular application.

This thesis explores this idea by examining a selection of problems related to creating a real-time system, including real-time monitors, timeouts, dynamic priority scheduling and basic priority inheritance. These features are then implemented in μC++.

Acknowledgements

I would like to thank my supervisor, Peter Buhr, for his guidance, encouragement and availability. Any clarity that exists in this thesis is a result of his input.

I am also grateful to Prabhakar Ragde for his graciousness in the face of bureaucracy and for his support, especially in reading my thesis. I would also like to thank Ajit Singh for reading my thesis and making valuable suggestions.

Thanks to Robert Zarnke for his input in developing a timeout mechanism.

In addition, I would also like to thank my friends for their support and for putting up with my quirkiness. Oliver Schuster for making sure I was well fed and entertained. Tayfun Umman for many lively discussions and for keeping me active. As well, I would like to thank my lab mates Ming Zhou, Tom Legrady, Dorota Zak and Jiongxiong Chen for making my work enjoyable.

And finally, I would like to thank my parents and my brother for their tremendous support and patience.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The most important aspect of a real-time system is 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 range from using data structures with fixed worst-case execution time to scheduling tasks using a well-defined algorithm.

Ideally, in addition to a system being predictable, it should also be flexible and extensible in order for it to be suitable for a diverse set of real-time applications. The ability to program different real-time applications with the same system discourages multiple ad-hoc systems and encourages the exploitation of new ideas. The primary motivation of this thesis is the construction of a flexible real-time system in μC++ [11, 35].

μC++ is a translator and runtime kernel that supports lightweight concurrency using a shared memory model. The μC++ language constructs are transformed into C++ by the translator. A runtime kernel to support concurrency on uniprocessor

and multiprocessor architectures is also provided. Extensions to μC++ for real-time, exception handling, debugging and profiling are ongoing.

The issues related to constructing a predictable real-time system are examined by considering a selection of specific problems that demonstrate the requirements at various levels of the system. As well, the interaction of these parts are examined where appropriate. These ideas are then implemented as real-time extensions to μC++.

1.1 Thesis Outline

The remainder of this thesis considers four specific real-time problems. Each of these problems is discussed in a separate chapter, and each chapter roughly consists of a section introducing the problem, a section describing related work, a section describing an appropriate solution and finally a section describing the implementation of this solution in μC++.

Chapter 2 considers how monitors can be extended for a real-time system. Both internal and external scheduling considerations are discussed. It also proposes a flexible implementation to support various kinds of scheduling.

Chapter 3 discusses adding a timeout mechanism for accept statements.

Chapter 4 discusses priority-based scheduling. Specifically, a method for assigning dynamic priorities and a corresponding priority queue data structure is proposed.

Chapter 5 discusses basic priority inheritance and an implementation of basic priority inheritance with mutex objects is presented.

Finally, conclusion and future considerations are discussed in Chapter 6.

Chapter 2

Real-Time Monitors

A monitor is a high level concurrency construct that can be used for indirect task synchronization and communication. Preliminary work on monitors was done by Hansen [8] and Hoare [24]. For a thorough discussion of monitors see Buhr [10].

A monitor consists of a set of data coupled with a set of routines to manipulate this data, similar to the structure of a class. The basic function of a monitor, however, is to guarantee mutually exclusive access to this data and to provide a mechanism for synchronization. To accomplish this functionality, the monitor routines, called *mutex routines*, are executed with mutual exclusion with respect to one another and special statements are available in the mutex routines to block and restart task execution.

In addition to these special statements used for synchronization, decisions regarding which task can enter the monitor next are also determined by the kind of monitor. The major difference among the various kinds of monitors relate to how tasks associated with the monitor are scheduled. As scheduling is fundamental to

real-time programming, whether it be a CPU or another resource, it is important to analyse all these decisions from a real-time perspective.

2.1 Background

Before considering the real-time issues relating to monitors in Section 2.2, basic monitor semantics are presented using Figure 2.1. Tasks enter a monitor by calling one of the mutex routines associated with the monitor. If a task is executing inside the monitor, then the monitor is referred to as *active* and this task is designated the *owner* of the monitor. Otherwise, the monitor is referred to as *inactive* and the monitor has no owner. As only one task can own the monitor at any given time, tasks calling into the monitor mutex routines may need to block. These tasks are referred to as *entry blocked* tasks. In general, the next task chosen to own the monitor from among these entry blocked tasks is the one that has been waiting the longest. This effect is usually implemented by putting calling tasks onto a first-in first-out (FIFO) queue, referred to as the *entry queue*.

2.1.1 Internal Scheduling

As previously mentioned, special statements are available to allow an active task to perform explicit synchronization in the monitor, referred to as *internal scheduling*. This synchronization is achieved using condition variables, and wait and signal statements. A *condition variable* is basically a queue of blocked tasks. For example, task x executing inside a monitor can block on condition variable B by executing the statement wait B (see Figure 2.2). When this task blocks, the monitor becomes

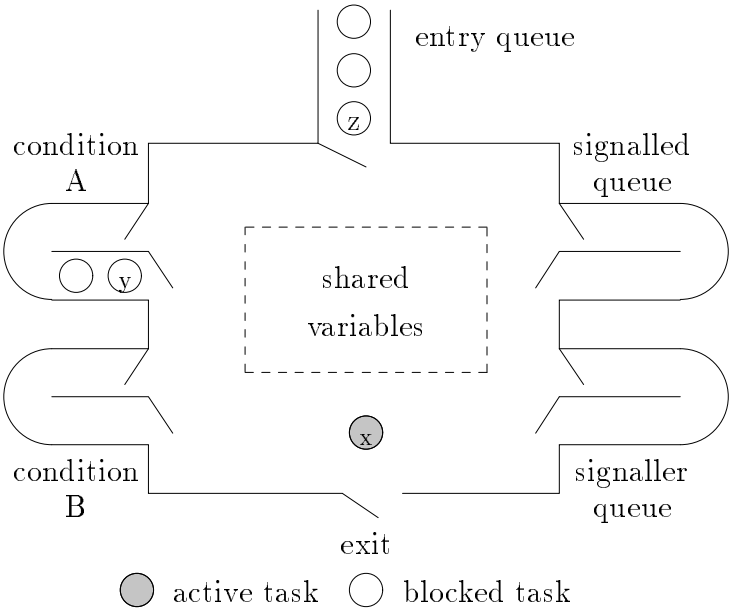

Figure 2.1: Example of a monitor.

inactive and another task can be scheduled. When a subsequent task inside the monitor issues the statement signal B, called the *signaller task*, a task is removed from the specified condition queue and placed on another queue, called the *signalled queue*. The signaller task, currently in the monitor, is then placed on the *signaller queue*. Tasks are typically removed from each of the signalled and the signaller queues in FIFO order. When the signaller task blocks, the monitor becomes inactive again and another task can be scheduled (see Figure 2.3).

Based on the discussion so far, there are three ways that an active monitor can become inactive, i.e., a task can explicitly exit the monitor, a task can block inside the monitor by issuing the wait statement or a task can block inside the monitor by issuing the signal statement. When the monitor becomes inactive, there are three queues from which a task can potentially be selected to become the next monitor

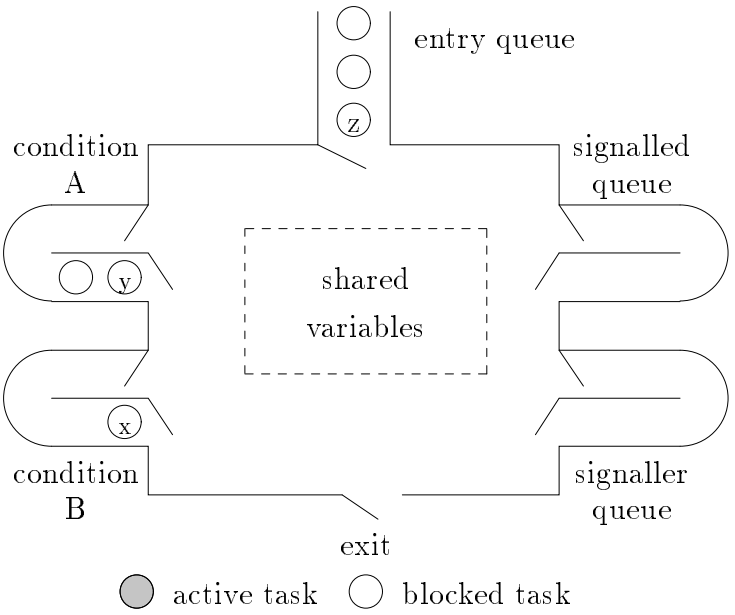

Figure 2.2: Task x blocks on condition queue B.

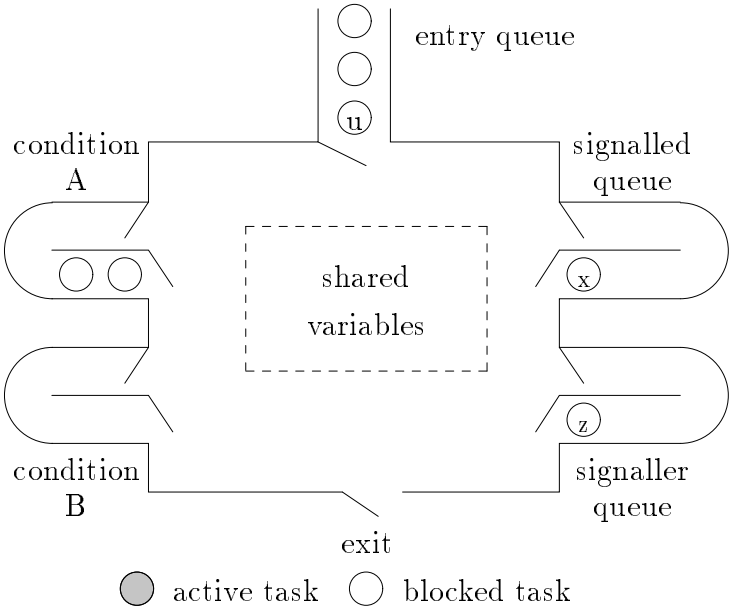

Figure 2.3: Task z signals condition variable B.

owner, i.e., the entry, the signaller or the signalled queue. As the tasks blocked on either the signaller queue or the signalled queue are blocked inside the monitor, these queues are referred to as *internal queues*, whereas the calling tasks on the entry queue are blocked outside the monitor, and this queue is referred to as an *external queue*. Note that the condition queues are not considered during scheduling decisions as the tasks on these queues are blocked waiting to be signalled, and hence, not yet eligible for re-entry into the monitor. The decision as to the order in which these three queues are considered is determined by the kind of monitor. Not only is the order in which these three queues are considered fixed for a particular kind of monitor, but monitors are typically classified according to this order (see Buhr [10]).

One example of a useful kind of monitor (pages 72-73, Buhr [10]) is the priority non-blocking monitor. Here, *priority* refers to the property that internal queues are given preference over external queues, and *non-blocking* refers to the property that the signaller queue is scheduled first among the three queues. With this kind of monitor, the signaller queue has highest priority, followed by the signalled queue and the entry queue has lowest priority. Another kind of monitor is the no priority non-blocking monitor. With a no priority non-blocking monitor, the signaller queue is still given highest priority but the signalled queue and entry queue have equal priority, so one internal queue is not given priority over the external queue. If two or more queues have equal priority, often FIFO ordering across the queues is supported in the implementation by merging these queues. It is important to note that not all orderings for scheduling these three queues result in useful monitors. For example, any ordering in which the entry queue is given highest priority can

result in starvation and synchronization difficulties with tasks blocked inside the monitor.

While three queues exist from a theoretical point of view, typically at most two queues are needed to implement a monitor. Normally, one of the signaller queue or the signalled queue can be eliminated for priority monitors, because one of these queues has a maximum size of one. This queue has a maximum size of one because tasks are only placed on these queues as a result of a signal statement. As a signal statement causes the monitor to become inactive, requiring a scheduling decision, the task placed on the highest priority queue is immediately selected as the next task to execute inside the monitor. Thus, the actual queue is eliminated and any task that would have been placed on this queue is directly made the new monitor owner.

2.1.2 External Scheduling

An interesting extension to the monitor described by Hoare [24] is the notion of *external scheduling*. With external scheduling, the decision regarding which task is allowed to execute inside the monitor next is made by selecting a particular task waiting outside the monitor on the entry queue. External scheduling is typically achieved by the monitor owner specifying which calls to mutex routines are considered *callable*. Only a task calling one of these callable mutex routines is permitted to enter the monitor when the next scheduling decision is made, i.e., when the monitor becomes inactive. Tasks blocked inside the monitor are usually not considered when external scheduling decisions are made. A common implementation of

external scheduling, using an *accept statement*, is described below, but other implementations of external scheduling have been proposed, including protected entries in Ada [59], operation avoidance in Sylph [17] and path expressions [2].

An accept statement consists of a list of callable mutex routines, e.g., accept(M_1, M_3). When a task inside the monitor executes an accept statement, this task blocks waiting for another task to call one of the specified mutex routines (see Figure 2.4). The task executing the accept statement is referred to as the *acceptor* and this task is said to be *accept blocked* while waiting for this call to occur. Tasks blocked inside the monitor are not considered when the acceptor relinquishes ownership of the monitor by accept blocking. Only a task already blocked on the entry queue or subsequently calling into one of the specified mutex routines is allowed entry into the monitor (see Figure 2.5). In general, the acceptor must wait for the calling task to exit the monitor before it can continue. Requiring the calling task to finish first allows it to satisfy whatever conditions the acceptor may require to continue executing, as well as, providing a method for synchronization, called a *rendezvous*.

However, it is also possible for this calling task to subsequently block on an accept statement while in the monitor, and, in order to maintain the described semantics, tasks blocked on accept statements are restarted in last-in first-out (LIFO) order. This ordering can be implemented by using a stack, called the *acceptor stack*, for managing the accept blocked tasks.

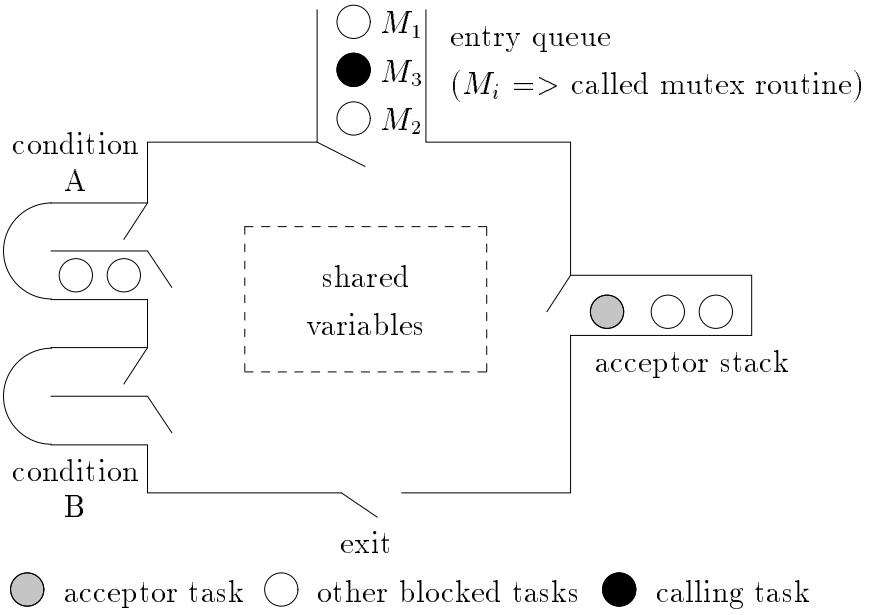

Figure 2.4: Accept blocking in a monitor.

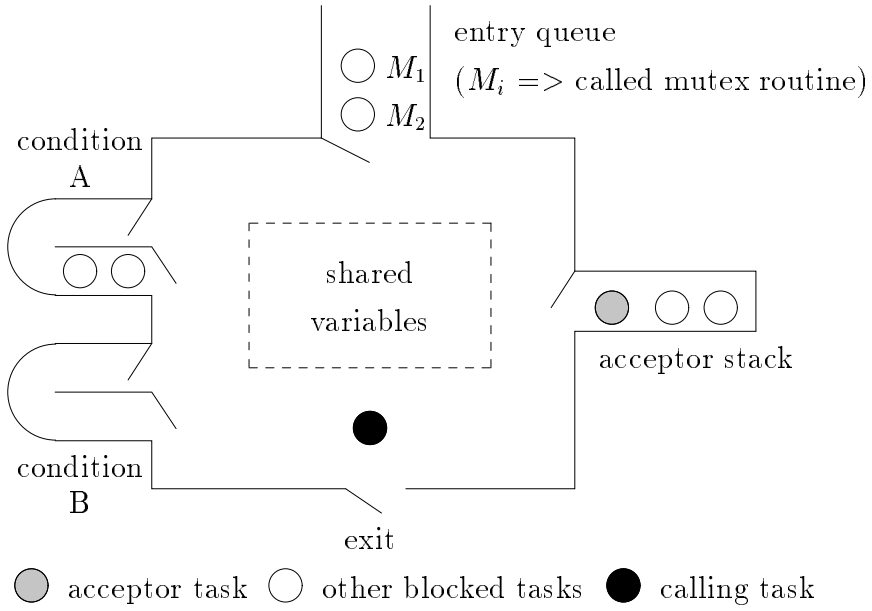

Figure 2.5: Only specified entry calls are allowed in the monitor.

2.1.3 Internal and External Scheduling

When internal and external scheduling are combined, conflict can occur (see Figure 2.6). The problem is determining how accept statements fit into the earlier discussion on internal scheduling. Due to the strict semantics imposed by accept statements, having separate queues for internal and external scheduling can cause potential starvation and semantic problems. With external scheduling, not only must the calling task have highest priority, but the acceptor must be the next task to execute when the calling task exits the monitor, which means scheduling it ahead of tasks on internal scheduling queues. The following discussion considers the order in which the acceptor stack could be scheduled relative to the other internal queues.

If the acceptor stack has lower priority than the other internal queues, then when a calling task exits a monitor, the acceptor may not be the next task to execute. In the example given in Figure 2.6, the acceptor would not be scheduled when the calling task exits because tasks exist on the other internal queues.

On the other hand, if the acceptor stack is given higher priority than the other internal queues, then a calling task cannot employ the signalling facilities in a useful manner. After a calling task performs a signal, the monitor must make a scheduling decision (see Figure 2.7). If the acceptor stack has higher priority, then the acceptor is restarted even though the calling task has not exited the monitor, nor has the signalled task executed. In this situation, it is more appropriate for a task on either the signalled queue or signaller queue to be scheduled next because the calling task may have prepared some event for the signalled task.

In both cases, having separate queues for internal and external scheduling results

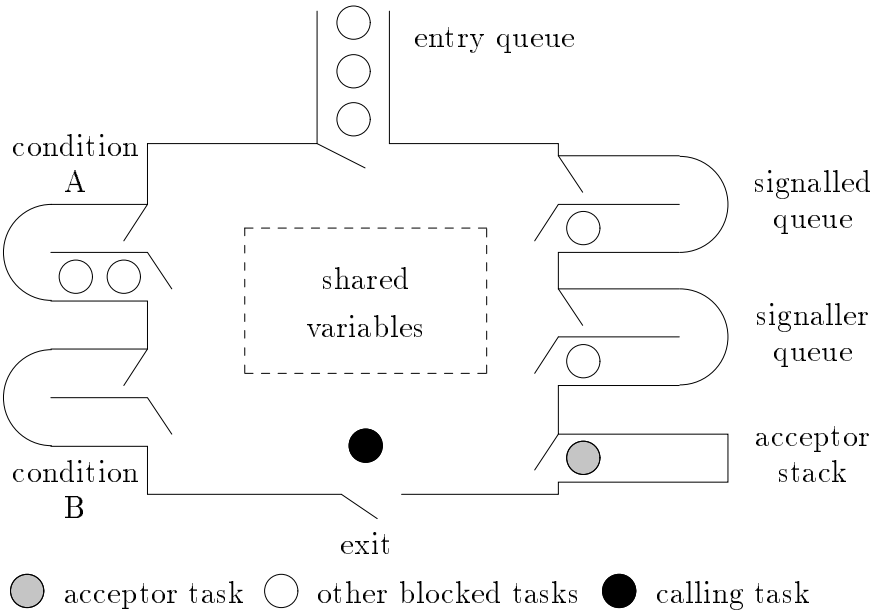

Figure 2.6: Internal and external scheduling in a monitor.

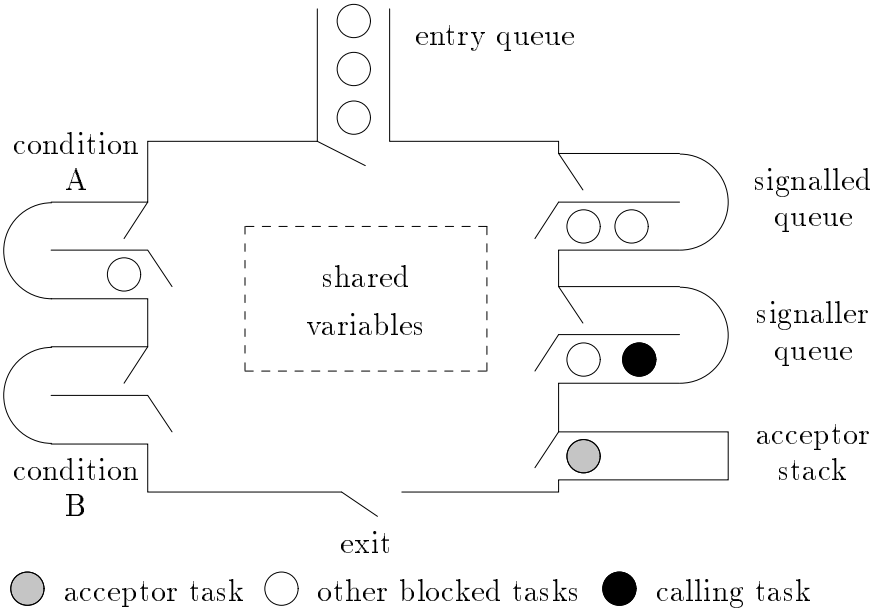

Figure 2.7: Calling task performs a signal.

in problems. The most reasonable semantics are obtained when the tasks associated with accept statements are integrated into the existing monitor data structures. This integration is achieved by using the signaller and signalled queues to schedule calling and acceptor tasks as well. The following analysis shows the constraints that must be dealt with.

To begin, consider the semantics of internal scheduling (see left side of Figure 2.8). First, a task inside the monitor blocks on a condition variable (Wait C). At some point, another task enters the monitor, satisfies the conditions required by the blocked task (Prepare X) and subsequently signals the blocked task (Signal C). Depending on the kind of monitor, when the signal occurs either the signalled task (T_1) or the signaller task (T_2) is scheduled next (Use X). (While it is also possible for a task on the entry queue to be scheduled next, depending on the kind of monitor, this situation is omitted because the internal and external scheduling data structures cannot be merged for this kind of monitor (see below)). As the signaller task must own the monitor in order to perform the signal, it is possible for this task to satisfy any conditions required by the blocked task before it performs the signal (Prepare X). Therefore, executing either the signaller or signalled task after a signal is reasonable, as long as the signaller does not invalidate X before exiting or waiting.

Furthermore, if the signaller task is given higher priority than the signalled task, e.g., with a Priority Non-Blocking monitor, then it is possible to delay satisfying the conditions required by a task blocked on a condition variable until after the signal is performed because the signaller task is always scheduled first (see right side of

Figure 2.8). In this example, the Signal C statement corresponds to a scheduling decision between tasks T_1 and T_2. While the diagram on the left works regardless of the kind of monitor because the condition is prepared first, the diagram on the right is only appropriate in situations where the signaller task is given priority over the signalled task.

It is possible to relate the semantics of an accept statement to the semantics described above for internal scheduling (see Figure 2.9). Similar to a task blocking on a condition variable (Wait C), the acceptor blocks on an accept statement (Accept(...)). A calling task can then enter the monitor (Call) in order to satisfy whatever conditions the acceptor requires to continue execution (Prepare X). When this calling task finishes, the acceptor is restarted (Use X). The equivalence suggested is that the acceptor (T_1) is the waiting / signalled task, the calling task (T_2) is the signaller and the actual call is the signal. In this situation, a Call acts like a Signal because at this point a scheduling decision between T_1 and T_2 must be made. Note that the Signal cannot be related to a calling task exiting a monitor because when the calling task exits, it is no longer eligible to be scheduled by the monitor.

From this analogy, the calling task is placed on the signaller queue and the acceptor is placed on the signalled queue. However, the problem with this equivalence is that the acceptor and the calling task must be executed in a specific order, i.e., the calling task (signaller) must be executed before the acceptor (signalled). However, certain kinds of monitors may schedule tasks from the signaller and signalled (or even the entry) queues in a different order. Therefore, this equivalence is only appropriate for certain kinds of monitors.

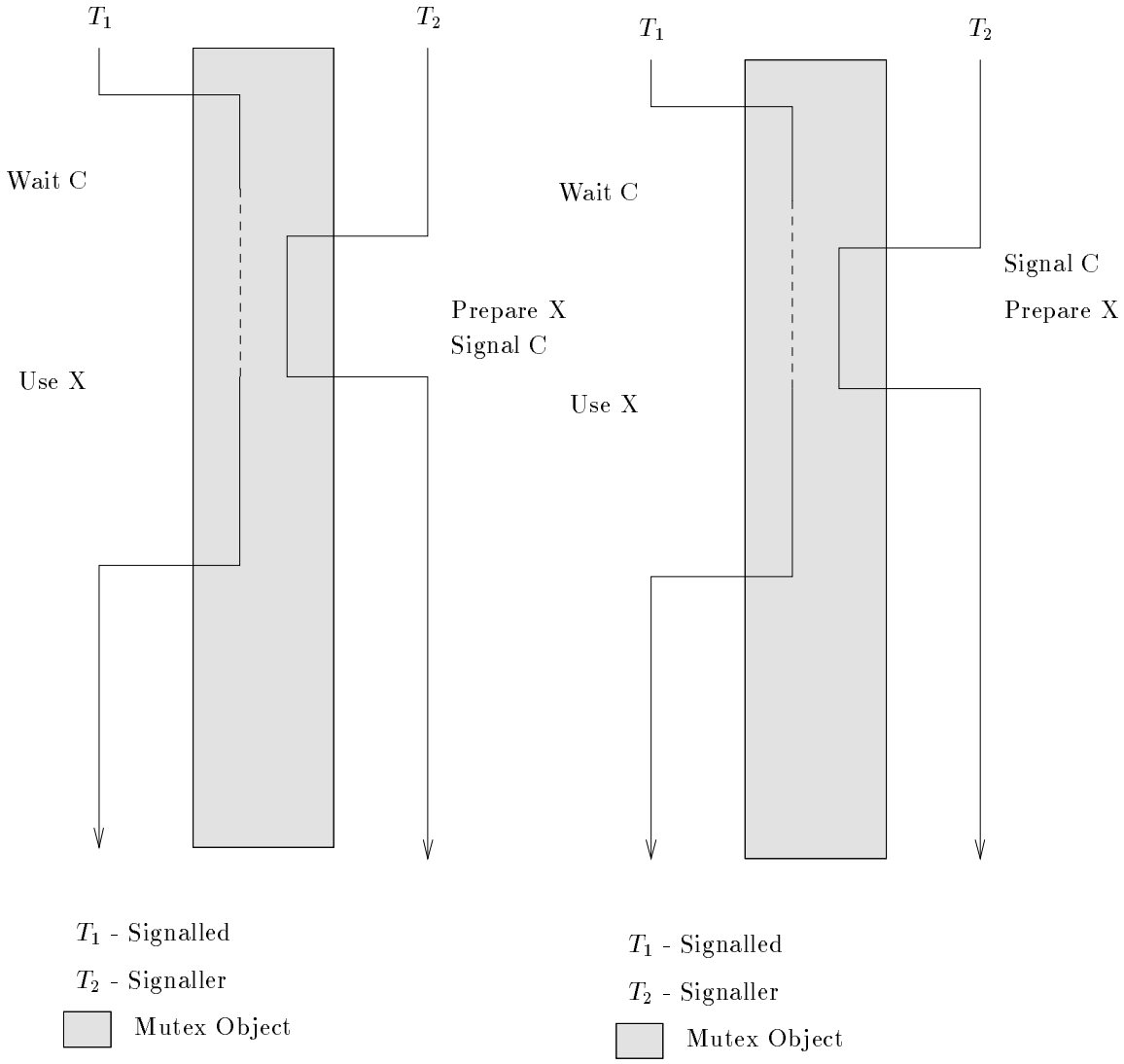

Figure 2.8: Satisfying conditions after signalling.

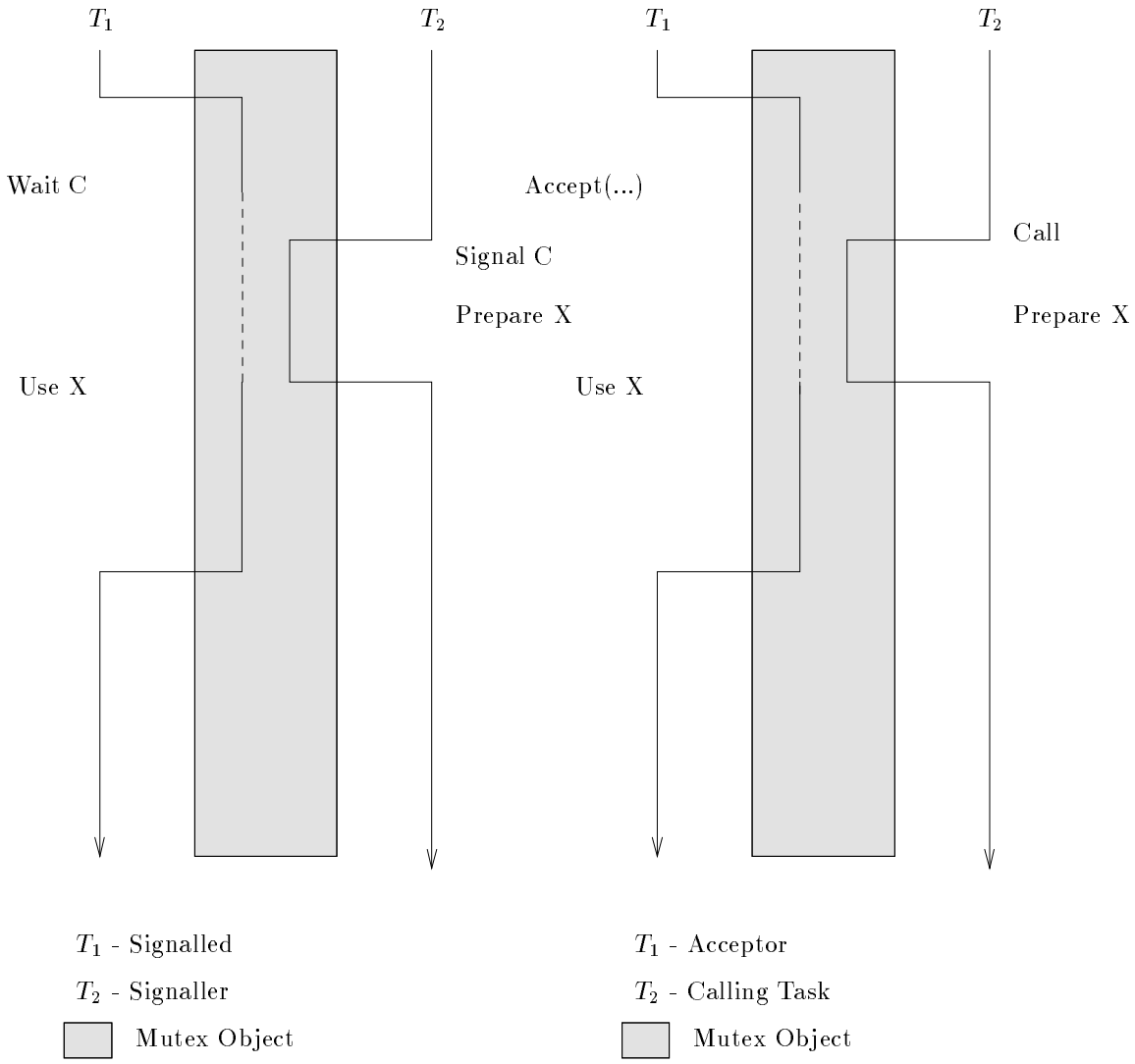

Figure 2.9: First equivalence relating internal and external scheduling.

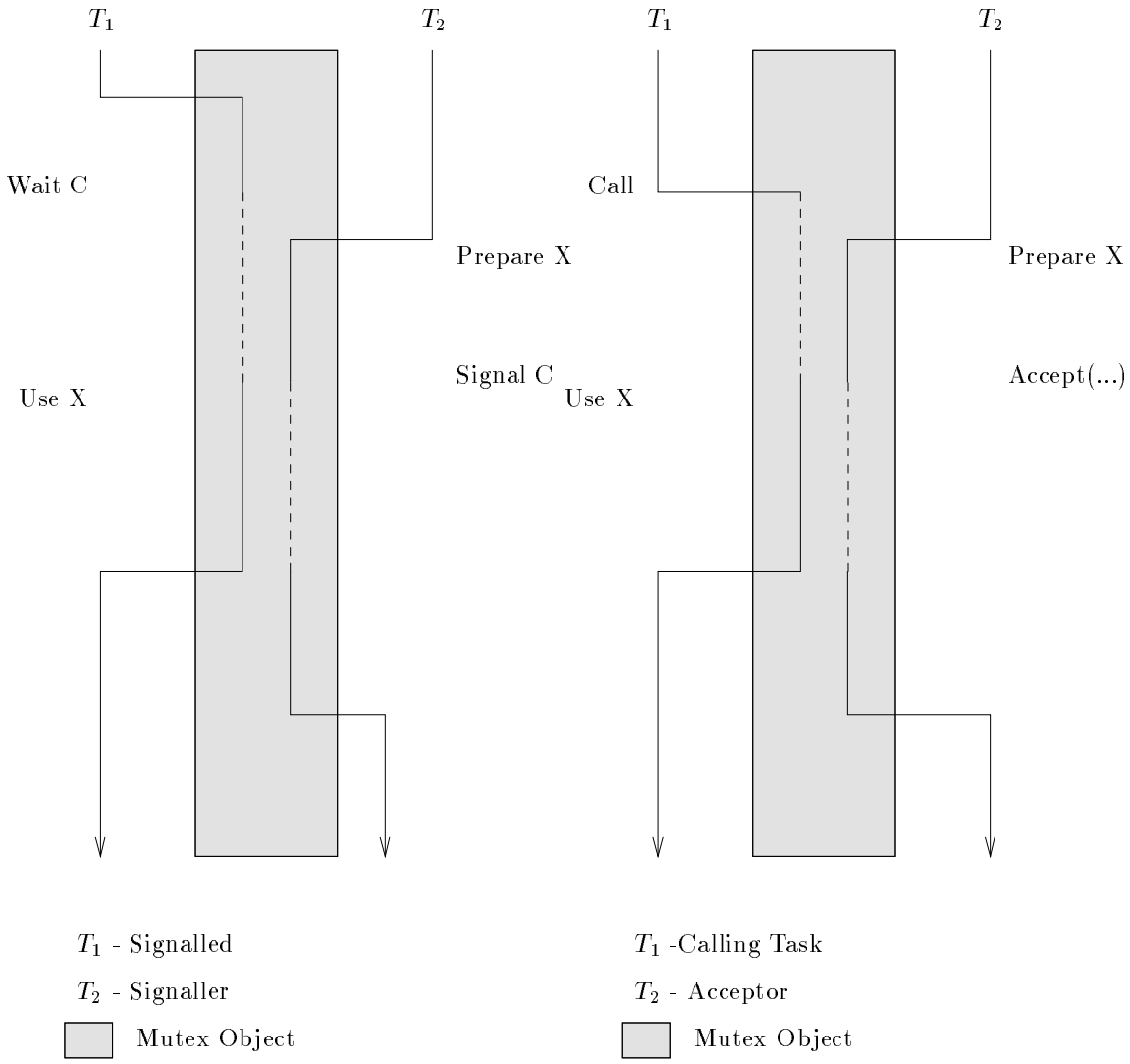

Figure 2.10: Second equivalence relating internal and external scheduling.

In fact, even if the equivalence is reversed, i.e., the acceptor is the signaller and the calling task is the waiting / signalled task, the same problem occurs (see Figure 2.10). In this case, the equivalence suggested is that the calling task (Call) must block waiting for a condition to be satisfied before it can enter the monitor. When the acceptor has satisfied the condition required by the calling task (Prepare X), the calling task is signalled (Accept(...)) to proceed (Use X). In this case, the execution of the accept statement can be considered the signal. With this analogy, the calling task is placed on the signalled queue and the acceptor is placed on the signaller queue. However, the order of execution of the acceptor and the calling task must still be maintained to achieve reasonable semantics. Again, this equivalence is only appropriate for certain kinds of monitors.

While these equivalences suggest that it is possible to use the internal scheduling queues for processing accept statements, two problems must be addressed. First, the acceptor and the calling task must be executed in a specific order regardless of the order in which the monitor schedules signaller, signalled and entry blocked tasks. Second, acceptors are processed in LIFO order but tasks involved in internal scheduling are processed in FIFO order.

If one internal queue is not given higher priority (remember monitors where the external queue is given higher priority than both internal queues are rejected), then both the calling task and the acceptor must be placed on the internal queue with higher priority in order to guarantee that they are executed before other entry blocks tasks. However, placing both the acceptor task and the calling task on the internal queue of higher priority, again, prevents the calling task from employing

the signalling facilities in a useful manner. The problem arises because the acceptor, as it is on the queue with highest priority, is scheduled before any task on the lower priority internal queue including either the calling task or any tasks signalled by the calling task depending on the kind of the monitor. For this kind of monitor, separate queues must be used for accept statements and the user must address any starvation or semantic issues at the implementation level.

As well, if the external and internal queues have equal priority and the internal scheduling queues are used to process accept statements, then it is impossible to guarantee that the acceptor and the calling task are scheduled before entry blocked tasks without violating the semantics of the kind of monitor. This violation occurs because any changes giving the acceptor and the calling task priority over entry blocked tasks also applies to the signaller and signalled tasks. Therefore, separate queues must also be used for accept statements with this kind of monitor and the user must address any associated problems at the implementation level.

In fact, the first problem can only be solved if the monitor assigns the internal queues higher priority than the external queue during scheduling decisions. This situation can be further subdivided depending on whether the internal queues have equal priority.

In order to simplify the analysis, assume that if both internal queues have equal priority, with respect to scheduling decisions, then these queues are merged to form a single queue. Now, the acceptor and the calling task should be placed on this single queue such that the calling task is scheduled first (see Figure 2.11). For this kind of monitor, it is irrelevant whether the calling task or the acceptor is associated

with the signaller or signalled task, respectively because all these tasks are placed on the same queue.

On the other hand, if the two internal queues have different priorities, then the calling task must be placed on the internal queue with highest priority and the acceptor should be placed on the other internal queue (see Figure 2.12). The freedom to place the calling task on the internal queue with higher priority, irrespective of whether it is the signaller or the signalled queue, exists because, depending on the actual ordering of the internal queues, one of the previous equivalences applies. If the signaller queue has higher priority, then the first case, where the calling task is considered the signaller and the acceptor is considered the signalled task, applies. If, however, the signalled queue is given preference, then the second case, where the calling task is considered the signalled task and the acceptor is considered the signaller, applies.

The problem relating to the different order in which tasks on the various queues are processed remains. When the internal queues have differing priorities, the queue that the calling task is placed on has a maximum size of one (see the discussion above) so it does not matter whether this queue is a FIFO queue or a LIFO queue. In fact, this queue can be eliminated and the calling task made the owner of the monitor directly. The problem occurs when the acceptors are merged into the remaining queue because the acceptors are restarted in LIFO order, but the tasks on the other queues are restarted in FIFO order. When these two queues are merged, the merged queue can be implemented in LIFO order and FIFO ordering can be achieved by the programmer for the non-acceptor tasks through an explicit

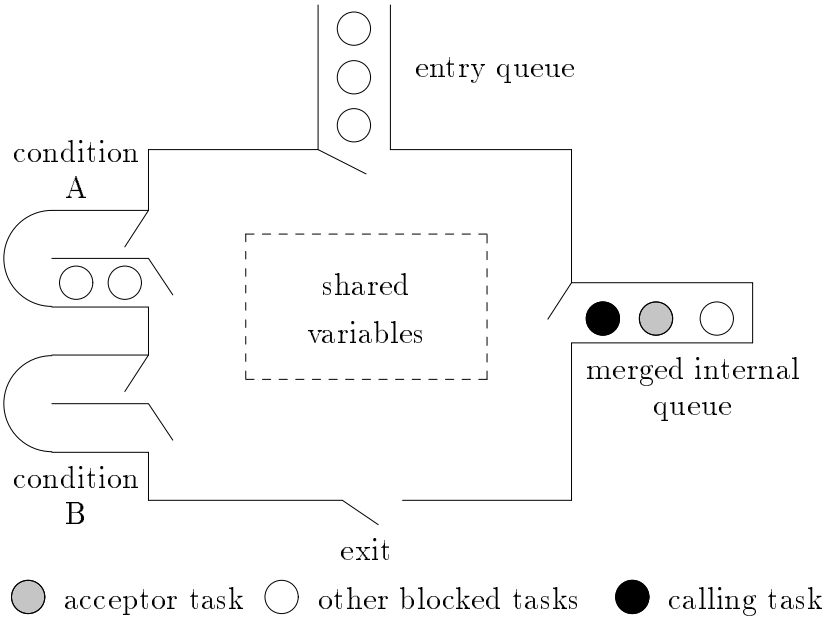

Figure 2.11: Internal monitor queues are merged.

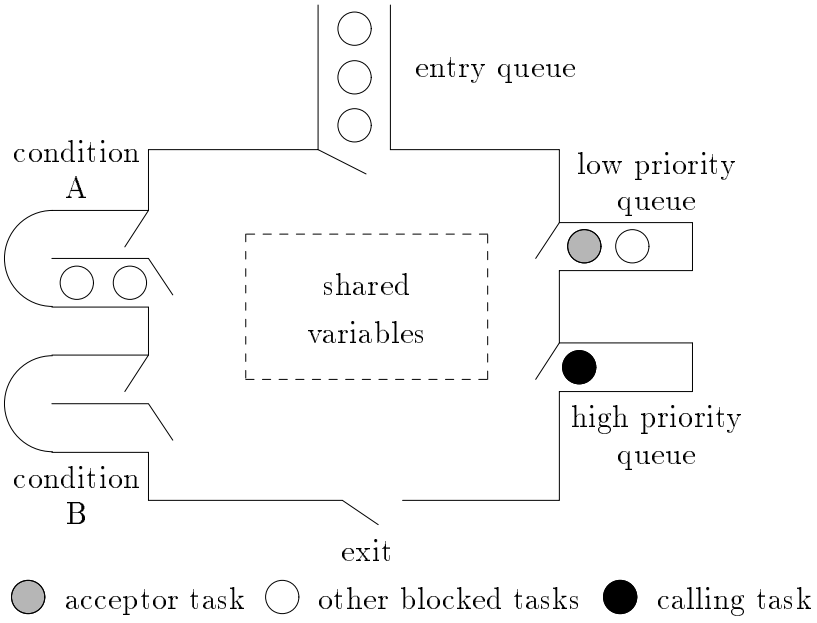

Figure 2.12: One internal monitor queue has higher priority.

technique called daisy-chain signalling (Chapter 8, Buhr [9]).

If the non-acceptor tasks are signalled tasks, then with daisy-chain signalling, instead of signalling multiple tasks, only one task is signalled initially and the other tasks to be signalled are left on their respective condition queues. Then, when the currently executing signalled task reaches a specified point, it signals the next task to proceed. In this way, the signalled tasks can be processed in FIFO order.

On the other hand, if the non-acceptor tasks are signaller tasks, then daisy-chain signalling is more complicated. In this situation, signaller tasks are placed directly on the merged LIFO queue without blocking on a condition variable first. In order to process the signaller tasks in FIFO order, a count of the number of signaller tasks is required. Then, every signaller task on the queue except the last one immediately blocks on a condition variable when it restarts. The last signaller task, however, actually executes and then signals the next signaller task on the condition queue when it finishes. This signal restarts the next signaller task and causes the previous signaller task to be placed on the LIFO queue again. Repeating this sequence for all the signaller tasks on the condition queue causes the signaller tasks to execute in FIFO order but leaves all the signaller tasks blocked on the LIFO queue again. As the signaller tasks are finished executing, they can be restarted in a LIFO order as they simply exit when restarted. The biggest problem with this approach is that the signaller tasks remain blocked in the monitor until all the signaller tasks are finished executing.

Finally, in the case where the two queues have equal priority, the single merged internal queue should also be implemented in LIFO order, with the acceptor and

calling tasks placed on this queue so that the calling task is scheduled first. It should also be possible to use daisy-chain signalling to process the non-acceptor tasks in FIFO order, but the process is even more complicated than that described above.

An interesting exception to the semantics of the accept statement arise when both accept statements and condition variables are used together. Consider the situation where the calling task of an accept statement blocks on a condition variable (see Figure 2.13). In this situation, it is possible for the acceptor to be restarted even though the calling task has not finished execution. When a calling task blocks on a condition variable, another task is scheduled for entry into the monitor. Depending on the order in which the queues are considered, it is possible for the acceptor to be scheduled despite the fact that the calling task has not finished.

There are certain types of problems that must be solved using this semantics. The logical explanation behind this semantics is that the action (cooperation) the acceptor was waiting for did not occur at this time, i.e., the calling task could not satisfy the cooperation and/or the rendezvous is broken. In this case, the acceptor must subsequently restart the calling task when the calling task can satisfy the cooperation or the rendezvous can complete. This situation differs from the situation described above in which the calling task performs a signal. In that situation, the calling task is ready to proceed but, because of the ordering of the queues, the acceptor is scheduled first. The semantics resulting from the use of both accept statements and condition variables can be difficult to understand and to program but is essential in certain cases. As will be seen next, maintaining the

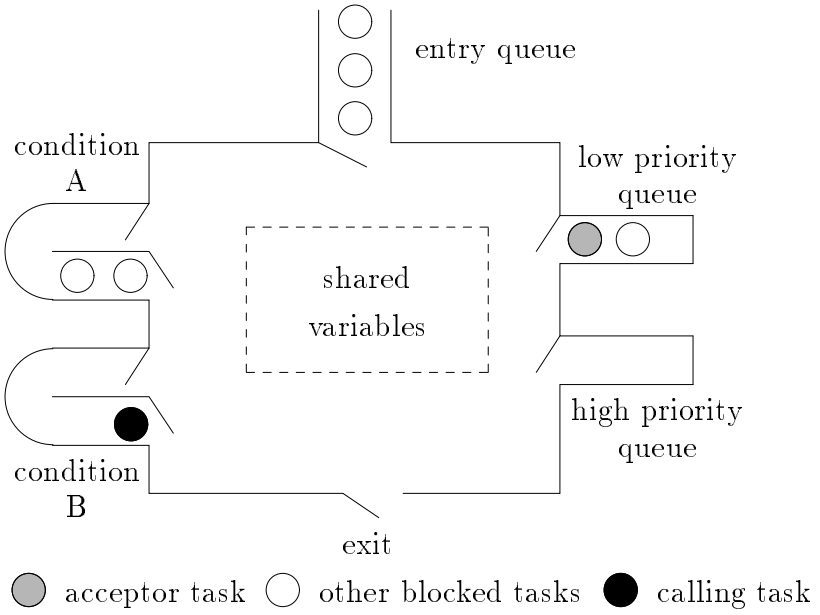

Figure 2.13: Calling task blocks on condition variable B.

logical behaviour of monitors is crucial when they are extended in the real-time domain.

2.2 Real-Time Considerations

The most important criteria for a real-time system is that tasks meet their specified deadline. As access to a monitor is serialized, it is possible that a high priority task calling into a monitor may have to wait for many other low priority tasks to proceed through the monitor, referred to as *priority inversion*, when the default FIFO ordering is used for the entry queue. This priority inversion can result in unacceptable delays that could cause deadlines to be missed. The desirable change in behaviour is for high priority tasks to be given preference over lower priority

tasks when determining which task should gain control over the monitor next. This section discusses how the behaviour of a monitor can be modified to better suit a real-time environment.

Unfortunately, the notion of a real-time monitor is complicated by the fact that the logical behaviour of a monitor must be maintained. The order of scheduling the three queues, i.e., the entry queue, the signaller queue and the signalled queue, determines which task gains control over a monitor and defines the kind of monitor. When augmenting monitors for a real-time environment, it is important to preserve the order in which these queues are considered. Allowing a user to modify the behaviour of a monitor in a limited and controlled manner is reasonable, but not if it changes the semantics of the kind of monitor in an unpredictable way, making it impossible to program. Thus, the semantics of the kind of monitor must be fixed, which in turn fixes the order in which the three queues are considered for scheduling decisions.

Fixing the order that the queues in a monitor are scheduled limits the possible real-time changes to the ordering of the tasks within a particular queue. A naive idea might be to simply prioritize all the various queues in the monitor, but, unfortunately, this is not necessarily possible. It is certainly reasonable to prioritize the entry queue as this expedites the entry of higher priority tasks into the monitor. If the monitor does not support accept statements, then it is also possible to prioritize the signaller and signalled queues. The only caveat with signals is that this deviation from FIFO ordering is the same as switching to LIFO, that is, daisy-chain signalling must be used to control the order in which tasks are processed or the

prioritized scheduling resulting from multiple signalling must be adopted.

If, however, the monitor allows accept statements, then from section 2.1, there are two possibilities. If the signaller and signalled queues do not have higher priority than the external queue, then separate queues must be used for accept statements. As the calling task is always immediately scheduled, its queue has a maximum size of one and can be eliminated in the implementation. The acceptor stack, however, can contain more than one task but cannot be prioritized, as it must be processed in LIFO order. The remaining internal queues, however, can be prioritized as discussed above. Unfortunately, the problems discussed in section 2.1 with these kinds of monitors still exist.

If the internal queues do have higher priority than the external queue, then either one internal queue has higher priority or both queues have equal priority. If one queue has a higher priority (see Figure 2.12), then this queue has a maximum size of one. As prioritizing a queue of size one has no effect, this queue can be eliminated in the implementation. The other queue, however, can contain more than one task. If this queue is merged with the acceptor stack, then it must be processed in LIFO order because of the acceptors. As described in section 2.1, the acceptors must be restarted in LIFO order so that a calling task can satisfy whatever conditions are required to restart an acceptor. In this case, the second queue cannot be prioritized.

Unfortunately, merging the acceptor stack with the internal queue of lower priority prevents this internal queue from being prioritized. By using the technique of daisy-chain signalling, however, it is possible to process subgroups of tasks associ-

ated with signalling in priority order. While this prioritization may not be global to the entire queue, subgroups of non-acceptor tasks that appear contiguously on the queue can be processed in locally prioritized order.

If the internal queues have equal priority, then to simplify the analysis, again assume that these queues are merged (see Figure 2.11). This merged queue must be LIFO because it may contain acceptors. Again, the technique of daisy-chain signalling can be used to process subgroups of non-acceptor tasks in locally prioritized order.

Now, consider the ordering of tasks within a particular condition queue. It would seem reasonable to allow this queue to be prioritized. Unfortunately, this presents a problem if *dynamic priorities* are used. A task's priority is considered dynamic if its priority is allowed to changed over time. Dynamic priorities present a queue maintenance problem because the priorities of blocked tasks can change when dynamic priorities are used, but condition queues are usually only modifiable by the task currently owning the monitor, as they are considered internal data structures. In order to support dynamic priorities, a condition queue may need to be adjusted by a task outside of the monitor to re-order the queue if the priority of blocked tasks change. To allow tasks outside the monitor to change these queues, locks would have to be added in order to protect condition queues. This solution, however, introduces additional complexity and overhead that may not be generally required. Rather than forcing all users to pay for the costs of locks on condition variables, it is reasonable to suggest that the programmer enforce whatever queuing scheme is appropriate for a given situation with the monitor, which requires priority

values to be accessible at the user-level.

Finally, always using a priority ordering on the entry queue is incorrect in some cases, e.g., the readers/writer problem could result in stale information if tasks are processed in priority order rather than FIFO ordering. But, from a real-time perspective, priority ordering is exactly the behaviour that is desired for a typical monitor. These problems, as well as the desire to use schemes to deal with priority inversion or to perform dynamic scheduling suggest that a more flexible approach is required.

2.3 Implementation

This section discusses the design and implementation of the real-time extensions I developed for monitors in μC++. μC++ monitors differ from those presented by Hoare [24] in several ways. These extensions also exist in similar constructs provided by other languages and are intended to make monitors easier to use without limiting their inherent functionality.

First, the monitors in μC++ can be classified as priority non-blocking monitors. The priority property requires tasks scheduled in the monitor (using internal scheduling) to be serviced before tasks entering the monitor. This property allows cooperation to be established between the signaller and signalled tasks, and eliminates inefficient busy waiting (e.g., loops around wait statements) because calling tasks cannot barge into the monitor. The non-blocking property requires the signaller task to continue executing after a signal statement. This property makes monitors easier to use and more efficient for the following reasons. Ease of use arises

because it is more intuitive for the programmer when the signaller task continues executing after a signal. Efficiency arises because signal statements usually occur before a task exits or blocks inside the monitor. Therefore, allowing the signaller task to continue execution eliminates a context switch and increases concurrency. This non-blocking property occurs with monitors in Turing [25] and with mutex and condition variables in POSIX [1].

The second difference is that μC++ monitors support recursive entry, i.e., a monitor owner is allowed to call back into the monitor. Recursive entry is useful because it eliminates a common source of deadlock, e.g., if the monitor owner calls another mutex member routine, either directly or indirectly. While it is possible to restructure the monitor to prevent this problem, the rewritten code is unnecessarily complex. Recursive entry is also supported by monitors in Java [22] and serialized objects in Obliq [14].

The final major difference is that μC++ monitors support both internal and external scheduling. As described in Section 2.1.2, external scheduling in μC++ is achieved using accept statements similar to those in Ada tasks [59], while internal scheduling follows the Hoare monitor. The advantage of accept statements is that they provide high-level synchronization and are also easier to use and understand than condition variables in many cases.

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

```
uMutex class monitor {
  private:
  ...                 // default no mutex routines
  public:
  ...                 // default mutex routines
};
```

Figure 2.14: μC++ Monitor

for the programmer.

Monitors in μC++ are structured as C++ classes. In order to specify a class as a mutex object, the qualifier **uMutex** is added before **class** (see Figure 2.14), which results in all public members of the class being mutex routines. Alternatively, the keywords **uMutex** and **uNoMutex** can be used to explicitly indicate whether a member routine should be executed with mutual exclusion. For example, **uMutex int** f(...), specifies that member routine f is be executed with mutual exclusion. All private and protected member routines default to no mutual exclusion, but can be explicitly changed.

The structure of a μC++ mutex object is given in Figure 2.15. As μC++ monitors are priority non-blocking monitors, the signaller queue has highest priority and is eliminated from the implementation. As well, the acceptor stack and the signalled queue are merged to create the acceptor / signalled stack. The only feature not yet discussed is the *mutex member queues*. A separate member queue is associated with each mutex member routine. When a task calling into the monitor blocks, it is placed on both the entry queue and the mutex queue associated with the member routine it called. The mutex member queues are an optimization to allow accept statements to be processed quickly. If the mutex member queues are not present, then each **uAccept** clause must linearly search the entire entry queue for calls to a

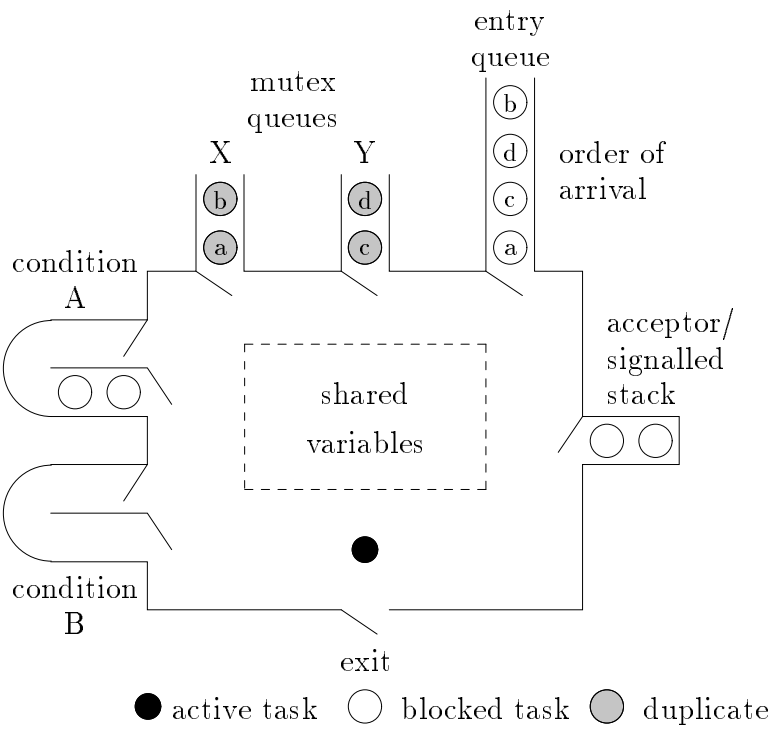

Figure 2.15: Structure of a μC++ Mutex Object [11]

particular member routine. Associating a mutex queue with each member routine eliminates this O(n) search, where n is the number of tasks on the queue, because outstanding calls can be located in O(1). The entry queue is still maintained to allow entry tasks to be processed efficiently in FIFO order, otherwise every mutex queue would need to be consulted and each entry timestamped.

The basic approach to implementing monitors is for the language to implement a different version of monitor for each kind that is desired, such as a version with FIFO ordering, as well as a variety of versions for different real-time schemes. This approach is problematic for obvious reasons, such as excessive code duplication, maintaining multiple versions, etc., as well as, the fact that a language extension

is necessary for each new type of monitor that is implemented.

However, considering the restrictions discussed above, there are only a few real differences possible among the various types of real-time monitors. In order to maintain the semantics of a priority non-blocking monitor, the order in which the various queues are considered cannot change. Therefore, the possible changes are restricted to the order in which tasks are processed on the entry, mutex and condition queues, as well as, allowing tasks to perform some additional functionality when entering or leaving a monitor. As suggested in the previous section, the condition queues are left as FIFO and the user is responsible for different scheduling schemes within the monitor. The functionality required when tasks enter or leave a monitor 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 extensions I designed and implemented for μC++ generalize the internal scheduling mechanism of the monitor to achieve the desired range of functionality. First, the specification of a monitor class is extended by requiring two additional types to be specified. The new format is based on C++ template syntax and is given below.

```
uMutex<EntryQueueType, MutexQueueType> class monitor {
  private:
  ...
  public:
  ...
};
```

The first parameter specifies the type for the entry queue and the second parameter

```
class uBasePrioritySeq : public uBaseScheduleFriend {
    uBaseTaskSeq list;
  public:
    virtual bool uEmpty() const;
    virtual uBaseTaskDL *uHead() const;
    virtual int uAdd( uBaseTaskDL *node, uBaseTask *uOwner, uSerial *s );
    virtual uBaseTaskDL *uDrop();
    virtual void uRemove( uBaseTaskDL *node );
    virtual void uOnAcquire( uBaseTask &uOwner, uSerial *s );
    virtual void uOnRelease( uBaseTask &uOldOwner, uSerial *s );
}; // uBasePrioritySeq
```

Figure 2.16: uBasePrioritySeq

specifies the type for the mutex queues. These types must be derived from the class uBasePrioritySeq (see Figure 2.16). The ability to specify different types for the entry and mutex queues can be useful for certain kinds of monitors. For example, if FIFO queues are used for both the monitor entry and mutex queues, then less overhead can be achieved by using a singly linked list for the mutex queues. This efficiency arises because tasks are only removed from the front of the mutex queues with this kind of monitor, but removed from anywhere in the entry queue.

In uBasePrioritySeq, the routines uEmpty, uHead, uAdd, uDrop and uRemove provide a generalized interface to a queue. The routine uOnAcquire provides a hook so that work can be performed when a task acquires control of a monitor. Similarly, the routine uOnRelease also provides a hook, but this hook is executed as the task leaves the monitor. No hook is invoked when a task blocks on an entry queue because the uAdd routine is already invoked to add the task to the entry queue. Therefore, any additional work can be added to the end of the uAdd routine.

These hooks allow advanced schemes such as priority inheritance (see Section 5.4) to be implemented. In the current implementation, the uOnAcquire and

uOnRelease routines are not called for mutex queues. The basic idea is to allow a task about to block on the monitor the ability to influence the task currently running in the monitor. Then, when a task exits the monitor, it can reevaluate any modifications it underwent while it was in the monitor.

In order to provide backwards compatibility with existing μC++ programs, the monitor arguments are optional. If no arguments are specified, as in Figure 2.14, then the system defaults to using internally defined FIFO queues for the monitor. Therefore, the header in Figure 2.14 is expanded to **uMutex**<uBasePrioritySeq,uBasePriorityQueue> **class** monitor, where uBasePrioritySeq is a doubly linked FIFO queue provided with μC++ and uBasePriorityQueue is a singly linked FIFO queue provided with μC++. It is also possible to allow this default behaviour to be modified, so, when no monitor arguments are specified, user specified default queues are used instead. Note that while no changes to the source code are required, programs must be recompiled for these changes to take effect.

Another interesting aspect of this implementation is that it is independent of the template features available in C++. Therefore, it is possible to combine template features with the monitor extension capability by basing the parameters required for the new definition of a monitor on the template parameters, as in:

```
template<class x,class y> uMutex<x,y> class monitor {
  private:
  ...
  public:
  ...
};
```

This definition uses template parameter x for the type of the entry queue and template parameter y for the type of the mutex queues. This monitor template can be instantiated with various types of queues to get different kinds of monitors.

This feature allows a monitor to be defined and then instantiated with different queue types in order to function in various real-time and non-real-time situations. For example, assume that two versions of the same type of monitor are required, i.e., a non-real-time version called NonRTmon and a real-time version called RTmon. The first step is to define an appropriate FIFO queue data structure and an appropriate priority queue data structure, e.g., uFIFOQueue and uPriorityQueue, respectively. Then, the statement monitor<uFIFOQueue, uFIFOQueue> NonRTmon instantiates the non-real-time version of the monitor and the statement monitor<uPriorityQueue, uPriorityQueue> RTmon instantiates a real-time version of the same monitor.

2.4 Related Work

While many real-time languages do not support monitors as a language construct, many of the considerations relevant to monitors apply to the constructs that are supported in these languages. This section considers accept statements and protected objects in Ada [59] and mutexes and condition variables in POSIX [1, 29].

2.4.1 Ada

Although Ada does not provide a monitor construct, a similar method of synchronization is provided with protected objects. A protected object is a class that allows access to its data via protected procedures, protected functions and protected entries. Protected procedures and protected entries are like mutex routines and are executed with mutual exclusion with respect to each other and protected functions. Protected functions, however, can execute concurrently, but are restricted to read-only access to any data. Protected functions are similar to no mutex routines except that no mutex routines can execute concurrently with mutex routines but protected functions can only execute concurrently with protected functions.

Protected objects support external scheduling by allowing *barriers* to be specified in the definition of protected entries. This capability does not exist with protected procedures or protected functions. Barriers are conditional expressions and a call to a protected entry is only allowed to proceed if the associated barrier condition evaluates to true. A call to a protected object that cannot proceed because of mutual exclusion or invalid barrier conditions is placed on the entry queue associated with the call. Figure 2.17 contains an example of a bounded buffer in Ada. In this example, Size and item are defined external to the bounded buffer. Size is a Natural variable containing the size of the buffer and item is a type defining the kinds of elements stored in the buffer. The when clauses on the Insert and Remove entries determine when calls to each entry are allowed to proceed.

In Ada, entries are also associated with tasks which are pseudo-classes, i.e, having no member routines. Instead, entry points are part of the accept statement,

```
type ItemArray is array(1..Size) of item;

protected type BoundedBuffer is
   entry Insert( Element : in item);
   entry Remove( Element  : out item);
  private
   Front, Back : Integer := 1;
   Count : Integer := 0;
   Elements : ItemArray;                -- bounded buffer
end BoundedBuffer;

protected body BoundedBuffer is
   entry Insert( Element : in item) when Count < Size is
   begin
      Elements(Back) := Element;        -- insert into buffer
      Back := Back mod Size + 1;
      Count := Count + 1;
   end Insert;

   entry Remove( Element : out item) when Count > 0 is
   begin
      Element := Elements(Front);   -- remove from buffer
      Front := Front mod Size + 1;
      Count := Count - 1;
   end Remove;
end BoundedBuffer;
```

Figure 2.17: Example of a bounded buffer in Ada.

combining control structure and entry code. A task services its entries by using a select statement composed of accept clauses. The synchronization that occurs when an accept clause and a corresponding call to an entry occur is a rendezvous. When the accept statement is finished, both the calling task and the acceptor resume independent execution after their call or accept, respectively. If an entry call cannot proceed immediately, it is placed on the entry queue associated with the call.

Each entry associated with tasks and protected objects is serviced by a separate entry queue. Ada entry queues are processed in FIFO order by default, but the Real-Time Annex also allows priority ordering to be selected through the use of a pragma. With priority ordering, the highest priority eligible task is selected to proceed. If several tasks among the eligible entries have this highest priority, then the entries are processed in the textual order of the accept clauses in the select statement. However, if a particular entry has several tasks with this highest priority, then the task blocked the longest is selected. The Real-Time Annex also allows support for priority ceiling (see chapter 5) to be selected through the use of a pragma.

While pragmas allow easy specification of appropriate real-time facilities for use in conjunction with high-level constructs, they do not provide the flexibility to allow users to incorporate any additional real-time functionality, for example, different priority inheritance protocols. This limitation, however, does allow more thorough run-time checks to be incorporated into the system and allows the implementation to be optimized for a particular scheduling strategy.

Ada offers functionality similar to internal scheduling through the use of the requeue statement. Requeue statements are used within entry bodies or accept statements and allow the current entry call to be requeued onto the specified entry queue. Requeuing an entry call also ends the current entry body or accept statement. This facility is more flexible than internal scheduling because a task can be requeued on a task or protected object different from that specified by the initial call. The drawback of using this approach for internal scheduling is that any intermediate results calculated (locally) by the entry call are lost when a requeue occurs. This approach also tends to be more expensive because internal scheduling with a monitor can typically be achieved without additional locking.

Despite the limitations described above, the facilities provided by Ada are adequate for many applications. The trade off for these limitations, however, is the potential for a simpler and in some cases more efficient implementation.

2.4.2 POSIX

While POSIX does not explicitly support monitors, similar functionality can be obtained by using mutexes and condition variables. In POSIX, a mutex variable is like a binary semaphore initialized to one and can be used to guarantee mutually exclusive access to a critical region. The big difference between a binary semaphore and a mutex variable is that a mutex variable has an owner associated with it. This owner is the task that locked the mutex variable and it is the only task allowed to unlock the mutex variable. To obtain functionality similar to a monitor, the code in every routine associated with the monitor is enclosed in calls to lock and unlock

the associated mutex variable. As only one task can own a mutex variable at any given time, additional calls to lock the mutex variable are queued. This queue is considered an external queue, and is equivalent to the entry queue described above for monitors.

Condition variables are also supported in POSIX. POSIX condition variables allow tasks to synchronize with respect to a mutex variable. A task in the monitor can atomically block on a condition variable and release the mutex variable. A subsequent owner of the mutex can then signal a condition variable to wake up a blocked task. It is interesting to note that a signal operation in POSIX is defined to unblock at least one of the threads blocked on the condition variable. Furthermore, a broadcast operation can be used to wake up all the tasks blocked on a particular condition variable. The order in which tasks are unblocked depends on the scheduling policy in effect. A signalled task implicitly reacquires the mutex variable, competing with attempts by new calling tasks and tasks already blocked on the entry queue. The order in which these tasks lock the mutex variable also depends on the current scheduling policy. If the signaller task owns the mutex before it signals, then it continues execution as the owner of the mutex.

The semantics described above are similar to a no priority non-blocking monitor, as the entry queue and the signaller queue have equal priority. The problem with not giving signaller tasks priority over tasks on the entry queue, however, is that the condition a signalled task is blocked waiting on must be reevaluated when the task regains control of the mutex, resulting in further waiting, as the condition may no longer be true by this time.

There are three scheduling policies supported in POSIX. These policies are referred to as SCHED_FIFO, SCHED_RR and SCHED_OTHER. With respect to mutex objects and condition variables, SCHED_FIFO and SCHED_RR restart the highest priority task that has been blocked the longest. SCHED_OTHER is implementationally defined, and thus, tasks can be restarted in arbitrary order. For real-time purposes, SCHED_FIFO and SCHED_RR seem the most appropriate. POSIX mutexes also support priority inheritance protocols (see chapter 5). The desired priority inheritance protocol and the scheduling policy can be specified as attributes of the mutex or condition variable's initialization. This feature allows different scheduling policies to be specified for mutex and condition variables.

While the facilities provided in POSIX are reasonable for dealing with priority inversion with respect to mutex objects, they lack a certain flexibility. For example, it would be difficult to program a FIFO version of the readers/writer problem or add support for a different priority inheritance protocol. However, these limitations are adequate for a commercial system that only needs basic real-time features.

2.5 Summary

When considering the changes required to make monitors appropriate for use in a real-time system, the semantics of the kind of monitor must be maintained, otherwise, programming with these monitors becomes very difficult. Therefore, these semantics limit the possible changes to the ordering of tasks on a particular queue.

Enhancing monitors for a real-time environment, however, requires a flexible implementation to capture various semantics. These semantics may include non-

real-time and real-time, as well as, the ability to handle dynamic priorities, priority inversion and other real-time schemes. This flexibility is also needed when trying to solve various problems that may require specific behaviour, e.g., even in a real-time program the readers/writer problem must deal with stale information, and thus, require certain queues to be processed in FIFO order, but it still may be desirable to deal with priority inversion. However, simply allowing the order of tasks on the entry queue to change only addresses problems related to external scheduling.

Unfortunately, condition variables are a problem because they involve internal scheduling. As the condition queues are internal to the monitor, it is inefficient to modify them from outside the monitor to maintain real-time properties. It is left for the programmer to resolve these problems in the most efficient manner for a given situation through programming. This situation is not entirely new, however, as other problems associated with internal scheduling, for example processing non-acceptor tasks in FIFO order, rather than LIFO order, when the acceptor stack is merged with an internal monitor queue, are also left for the programmer to solve and any real-time changes are an extension of this problem.

The design and implementation of monitors I developed for μC++, tries to address some of the flexibility requirements needed to solve a broader class of problems. The basic idea is to encapsulate the possible variations in functionality into the definition of the entry and mutex queues used by the internal scheduler and to provide hooks to achieve the remaining functionality. The user is given the ability to affect the order in which tasks enter the monitor, as well as, the freedom to add some additional functionality when a task acquires or releases a monitor.

These enhancements allow a user to program monitors for a variety of real-time and non-real-time systems. However, even with the limited flexibility that is provided, a real-time developer must be careful. For example, it is important for the operations in the queue data structures to maintain constant execution time for real-time systems. Therefore, much of the problem is now moved into the data structure domain.

Chapter 3

Timeouts

A fundamental part of real-time programming is the ability to specify timing constraints for tasks. These constraints are necessary to create a system which is both predictable and schedulable. 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 some operations it is reasonable to expect the user to address the potential for unboundedness in their implementation. For example, loops can be bounded by specifying a fixed number of iterations and recursion can be limited to a fixed depth. For operations such as I/O, and process synchronization and communication, however, this expectation may not be quite as reasonable. Clearly, making a requirement that all tasks in a system be independent (i.e., no synchronization and communication) or perform no I/O is unrealistic. Therefore, a method to bound these operations is required. The real problem with these operations is not that they may execute for an unbounded amount of time but rather that they might block indefinitely.

To address this potential for unbounded blocking, it is necessary to implement a *timeout mechanism*. A timeout mechanism is a method to limit operations by causing them to be aborted if they have not progressed to a certain point within a specified amount of time. The amount of progress to prevent abnormal termination varies depending on the type of operation. For operations that could block for a potentially unbounded amount of time, as opposed to operations that might execute for an unbounded amount of time, the requirement is usually that the operation has begun executing within the specified time limit. This degree of progression is reasonable for these operations because once the operation has actually started, it is possible for the user to limit the worst-case execution time.

With I/O, unless the operating system provides an explicit mechanism to abort an operation before it is completed, it is impossible to limit the blocking time of these operations. Without operating system support, it would be necessary to limit I/O operations at the user level, which is generally infeasible.

With task synchronization and communication, however, it may be possible, though not necessarily practical, to limit the blocking time of these operations at the user level. The reason for the difficulty is that the completion of these operations is dependent on other tasks in the system and this makes it difficult to characterize the exact behaviour of the tasks. Not only must transient overloads and error conditions be considered, but also an extremely large number of execution paths, if time slicing is enabled. As suggested, the most reasonable approach to deal with this potential for indefinite blocking is to implement a timeout mechanism. However, similar to limiting loops or recursion, the onus is on the user to use the

timeout facilities where appropriate.

For example, consider a simple timeout mechanism to block a task for a specified period of time, e.g., sleep(1.0) to block a task for one second. Is it possible for a user to construct a timeout mechanism for synchronization and communication using this simple time delay? Suppose task T_1 blocks on a synchronization or communication operation. In order to limit the blocking time associated with this operation, T_1 creates a separate task, before it blocks, to serve as a *timeout task*. This timeout task blocks for the specified amount of time using sleep and then tries to wake up T_1. If the expected action occurs before the timeout expires, the timeout needs to be *short-circuited* and the wake up from the timeout task prevented. Short-circuiting the timeout requires waking up the timeout task immediately, rather than waiting until the timeout actually expires. This approach is reasonably complex, requiring a fair amount of setup and coordination on the part of the user, and it also incurs the additional overhead of creating and managing many timeout tasks, depending on their degree of usage.

The other option to address the indefinite blocking problem is to allow a timeout value to be specified with the blocking operations. Similar to the idea described in the user level implementation, if an operation is not completed within a specified time limit, the operation is terminated and the task is restarted. The easiest approach to implementing this timeout facility might be to mimic the actions described with the user-level implementation. While this approach would insulate the user from the complexity, it does not address the concerns regarding cost. This chapter describes a more integrated and efficient approach to implementing timeout

facilities for accept statements, which are a blocking operation for synchronization and communication.

3.1 Background

The notion of time is extremely important in all real-time systems, e.g., timeout facilities in ADA [59] and (Real-Time) Concurrent C [19, 20]. However, μC++ did not have timeout facilities available for any operations. Therefore, I augmented μC++ to allow timeouts to be specified for accept statements because most synchronization and communication operations in μC++ involve accept statements. Specific comparisons with ADA and Concurrent C are presented as each component of the timeout mechanism for μC++ is discussed.

Note that limiting operator delay should not be confused with techniques for guaranteeing general timing constraints. The idea behind augmenting these operations is to eliminate indefinite blocking, and thus, allowing timing constraints to be specified. More general timing mechanisms exist to guarantee that specified timing constraints are met.

3.2 Goals

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.

From an implementation perspective, it is important that any changes made have minimal impact when the timeout facility is not employed. As well, the design has to guarantee that no potential for deadlock is introduced into the system. This requirement does not mean that deadlock cannot exist in the system when these operations are used, but rather that using timeouts does not introduce any additional potential for deadlock into the system. Another important condition, especially from a real-time perspective, is to design the facility so that its usage only incurs a small, fixed amount of overhead. Finally, for maintenance reasons it is also important for the implementation to have limited complexity.

3.3 Syntax and Semantics

In μC++, the communication facility is the call, in conjunction with the accept statement / mutex object combination. (See chapter 2 for a complete discussion of the interaction between accept statements and mutex objects.) The syntax described is for μC++ [11], but similar syntax is used in Ada and Concurrent C.

Consider the syntax of a general accept statement in μC++:

```
uWhen ( conditional-expression )                    // optional guard
    uAccept( mutex-member-name )
        statement                                   // optional statement
uOr uWhen ( conditional-expression )                // optional guard
    uAccept( mutex-member-name )
        statement                                   // optional statement
...
    ...
        ...
uElse                                               // optional default clause
    statement
```

A communication (rendezvous) between two tasks involves a calling task and an acceptor. The acceptor executes an accept statement as given above. Each **uAccept** statement consists of a list of the kinds of messages the acceptor is willing to accept next. In μC++, accept statements are associated with mutex objects and the kinds of messages consist of the names of mutex routines that may proceed. Each **uAccept** clause is associated with exactly one mutex routine from the mutex object. The calling task, on the other hand, performs its side of the operation by calling one of the mutex routines in the object. As described in chapter 2, it is possible for a call into a mutex object to proceed without using an accept statement, e.g., when the mutex object is inactive.

The optional **uWhen** clauses are referred to as guards and consist of a conditional expression. If the guard evaluates to true or does not exist, the accept alternative 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 can be immediately accepted, i.e., an outstanding call to the routine associated with the accept statement exists. If there is more than one such clause, then the first

available alternative is chosen, as opposed to a non-deterministic selection.

Only if there is no clause that is immediately *acceptable*, i.e., no outstanding calls exist for the specified mutex routines, is the optional **uElse** clause processed. The **uElse** clause is referred to as an *immediate alternative* and is always chosen if no **uAccept** clause is immediately acceptable. In μC++, only one **uElse** clause is allowed and it must appear as the last clause in the accept statement. These restrictions are reasonable because only one immediate alternative can be selected even if several are allowed to exist and forcing the **uElse** statement to appear last, not only fits logically with the semantics of other language constructs, but implies the order of processing that is enforced in μC++.

If no immediate alternative exists, then the accept statement blocks until a call to one of the open accept alternatives occurs. When such an acceptable call occurs, the calling task is allowed to continue inside the mutex object until it exits, at which time the accept blocked task is typically restarted. (It is possible for the acceptor to be restarted before the calling task exits, see chapter 2, page 25 for details.) When the acceptor restarts, it continues execution of the accept statement by executing the block of code associated with the clause for the accepted routine.

The semantics for the accept statement available in Ada differs from the μC++ version. While Ada also allows only one immediate alternative, if more than one calling task is immediately acceptable then a task is chosen according to the entry queuing policy in effect. This selection is made from among all the tasks associated with open accept alternatives. For the default queuing policy, an arbitrary choice is made. The ability to consider all open alternatives before selecting a task is

advantageous for a real-time system as it can be used to allow the acceptable task with the highest priority to continue execution. However, this flexibility also incurs more overhead, for example, all the accept alternatives must be examined before a choice can be made.

The semantics for the accept statement in Concurrent C also differ somewhat from μC++. First, if more than one accept alternative is immediately acceptable then the choice as to which one is selected is arbitrary. As well, Concurrent C allows multiple immediate alternatives to be specified but, again as only one alternative can be selected, an arbitrary choice is made. The deterministic choice made in μC++ however, seems more appropriate to achieve the real-time goal of predictability.

To incorporate a timeout capability in μC++, I augmented the accept statement in the following way:

```
uWhen ( conditional-expression )                    // optional guard
    uAccept( mutex-member-name )
        statement                                   // statement
...
    ...
        ...
uOr uWhen ( conditional-expression )                // optional guard
    uTimeout( duration or time value )              // optional timeout
        statement                                   // statement
```

In this version of the accept statement, the **uTimeout** clause replaces the terminating **uElse** clause, making the **uElse** clause and the **uTimeout** clause mutually exclusive. The **uOr** separator is necessary for **uTimeout** but not **uElse** because the keyword **uTimeout** is also used as a time delay in μC++, e.g., **uTimeout**(1.0) causes a task to delay (block) for one second. Without the **uOr** separator, accept statements using the timeout facilities are ambiguous, e.g.:

```
uAccept( ... )
  a;
uTimeout( ... );
```

if the **uOr** separator is not required, then it is unclear if this example represents an accept statement with a timeout clause or an accept statement followed by a delay request. Since the **uOr** separator is required for accept statements, this example represents an accept statement followed by a delay request.

Choosing to make the **uElse** clause and the **uTimeout** clause mutually exclusive seems reasonable because if an immediate alternative exists, then the timeout clause would never be selected. As well, mimicking the semantics of the **uElse** clause, it is invalid to have more than one active timeout clause in an accept statement. This restriction is reasonable, as well, because if several choices are available, then logically the timeout alternative with the smallest value is always selected. Furthermore, it is possible to achieve the functionality of multiple timeout statements by using conditional operators when specifying the duration of the timeout and inside the associated code segment for the timeout clause. In the example presented in Figure 3.1, the routine selecttimeout returns the appropriate timeout value and which choice is made. The code segmented associated with this choice can then be executed if the accept statement times-out.

The evaluation of this augmented accept statement begins by determining if an open **uAccept** clause exists. Again, if there is more than one such clause, then the first available alternative is chosen. Similar to the **uElse** version, only if no open accept alternative is available is the **uTimeout** clause processed. This semantics

```
uAccept( ... )                                  // accept clauses
...

uOr uTimeout( T = selecttimeout( ... ) )        // timeout clause
    if( T == ... )                              // choice 1
        statement                               // statement
    else if ( T == ... )                        // choice 2
        statement                               // statement
    ...
    else                                        // default
        statement                               // statement
```

Figure 3.1: Simulating multiple timeout statements in μC++

means that if a **uAccept** alternative is immediately available, then no additional overhead is incurred for including a **uTimeout** clause. Otherwise, assuming the guard on the **uTimeout** clause is true, the accept statement blocks no longer than the duration specified. If no calls occur to an acceptable member routine within this time, the acceptor is restarted and the block of code associated with the **uTimeout** clause is executed. If a *valid call* does occur, i.e., a call to an acceptable member routine, the timeout is cancelled and the accept statement proceeds as normal.

Again, the semantics for Concurrent C and Ada differ somewhat when timeouts are used. First, several timeout clauses are permitted in these languages, and the clause with the smallest time value is selected. Furthermore, in Concurrent C both timeout clauses and immediate alternatives are permitted in the same accept statement, but an arbitrary immediate alternative is always selected. As discussed above, the restrictions imposed by the existing μC++ design, with respect to the types of clause that can be used together in an accept statement, plus the extensions I made are reasonable and do not exclude any significant functionality.

3.4 Design

There are many different ways in which to design a timeout facility. One key consideration is to design a facility that fits naturally into the semantics of the operation being augmented. This type of design not only makes the facilities easier for the user to understand, but also tends to allow the operations to be terminated in a graceful manner. This section begins by considering how a user might augment accept statements to allow for a timeout facility and then tries to evolve this technique into an integrated and efficient facility provided by the system.

In order to implement a timeout facility, there are some minimum system requirements. First, the system must provide some notion of time. Ideally, the system should provide a time delay operation that allows a task to block for a specified duration. While an approximation of a time delay operation can be constructed using a loop and an operation to get the current time, this method is inefficient and inaccurate. The discussion in this section assumes that a time delay operation is available.

3.4.1 Timeout Tasks: User Level Implementation

To begin, reconsider the idea of implementing timeouts at the user level, presented at the beginning of the chapter, but now for accept statements. The basic idea is to create a separate task to serve as a timeout task. This task blocks for a specified amount of time and then calls into the mutex object, for example in μC++:

```
// routine start is passed time value and stores reference to acceptor
uAccept(start);          // synchronize with acceptor
uTimeout(time);          // block for specified time
acceptor.timeout();      // call acceptor
```

The timeout task begins by blocking on an accept statement for a call specifying a timeout value. The acceptor task then calls this timeout task with a particular time value, in order to activate the timeout. Having the timeout task begin by blocking eliminates the uncertainty introduced by the overhead of task creation and allows a more accurate timeout value to be obtained. This uncertainty exists because the amount of time required to create a task can vary depending on the speed of the system. This task then blocks for the specified amount of time and then calls the acceptor when it wakes up. As well, an empty mutex routine, i.e., a mutex routine with an empty code body, must be created for the timeout task to invoke and this routine is accepted in every accept statement requiring a timeout.

If a valid entry call occurs before the timeout expires, then some method is needed to short-circuit the timeout. It would be incorrect to leave the timeout outstanding as it may interfere with a subsequent accept statement using the timeout facility. Typically, there are two methods available for short-circuiting a timeout, i.e., deleting the timeout task or using a special capability to immediately unblock the timeout task. Typically, the timeout task is short-circuited by the acceptor, after it restarts, as the acceptor has explicit knowledge of the timeout task.

Deleting the Timeout Task

With this technique, if a valid entry call occurs before the timeout expires, the acceptor deletes the timeout task when it restarts. However, deleting the timeout task creates a potential race condition. This potential race condition exists because there is a window between a valid entry call occurring and the call to delete the timeout task (see Figure 3.2). During this window, the delay in the timeout task could expire, causing the timeout task to call into the mutex object. The result of this potential race condition is that sometimes there may be an outstanding call to the timeout mutex routine after a calling task has been accepted. However, it is impossible to process an entry call for a task that no longer exits (see Figure 3.2). Furthermore, checking to see if the timeout has already expired before attempting to delete the task does not eliminate the race condition. There is no easy solution to this potential race condition.

Waking up the Timeout Task

Another technique for short-circuiting the timeout is to use a special capability to immediately unblock the timeout task. The timeout task then restarts and calls into the acceptor exactly as if the timeout had expired. Short-circuiting the timeout task in this manner eliminates the concern regarding the potential race condition between the valid entry call and the call to short-circuit the timeout. By requiring the acceptor to wake up the timeout task, instead of deleting this task, an outstanding call always exists to the timeout mutex routine regardless of whether the timeout is short-circuited or expires on its own. If the accept statement timed-

out, then the call to the timeout routine is used to restart the acceptor. On the other hand, if a valid call occurs before the accept statement times-out, then a call to the timeout routine is still outstanding. In this case, the acceptor executes another accept statement with the timeout routine as the only valid call to finish cleaning up the timeout task (see Figure 3.3).

On the other hand, if the timeout expires before a valid call occurs, then the timeout task gains entry into the monitor (see Figure 3.4). As the timeout mutex routine is empty, the timeout task immediately exits, which restarts the acceptor. The acceptor, detecting that the timeout mutex routine was invoked, can then perform the behaviour associated with a timeout. Any entry call that occurs subsequently simply blocks as required because of the default behaviour of the monitor when a valid entry call occurs, in this case a call to the timeout mutex routine.

This approach can be implemented in μC++ as follows:

```
timeoutTask = createTimeout();              // create timeout task
timedout = false;
timeoutTask.start( time );                  // synchronize with timeout task

uAccept( ... )                              // accept statement with timeout
  ...;
uOr uAccept( timeout ) {                    // timeout mutex routine
  timedout = true;
  ...
} // uAccept

if ( !timedeout ) {                         // if accept did not timeout
  shortcircuit( timeoutTask );              // wake up timeout task
  uAccept( timeout );                       // ignore timeout mutex routine call
} // if
```

Thus, it is possible to implement timeouts for accept statements at the user level.

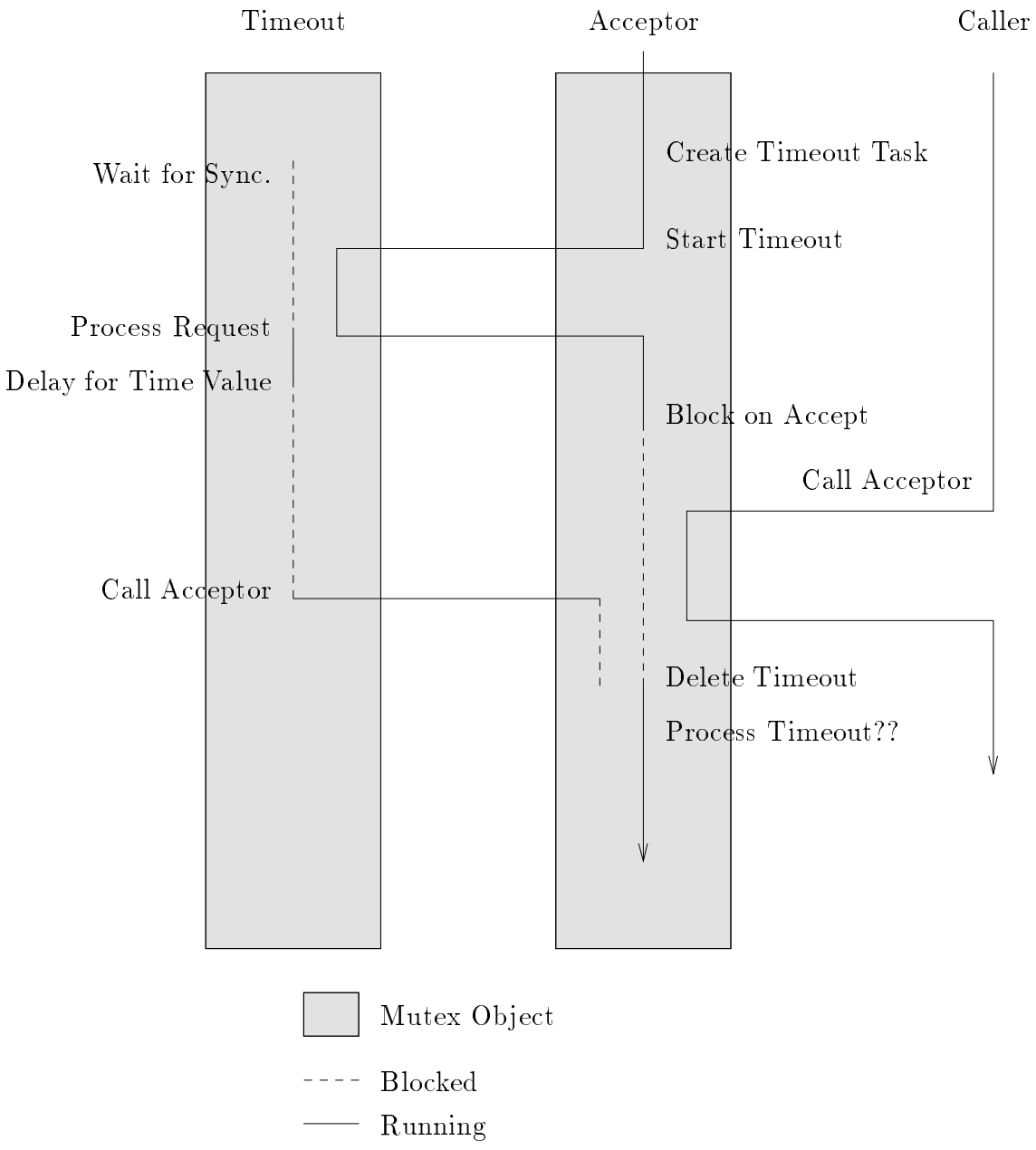

Figure 3.2: Timeout task is deleted

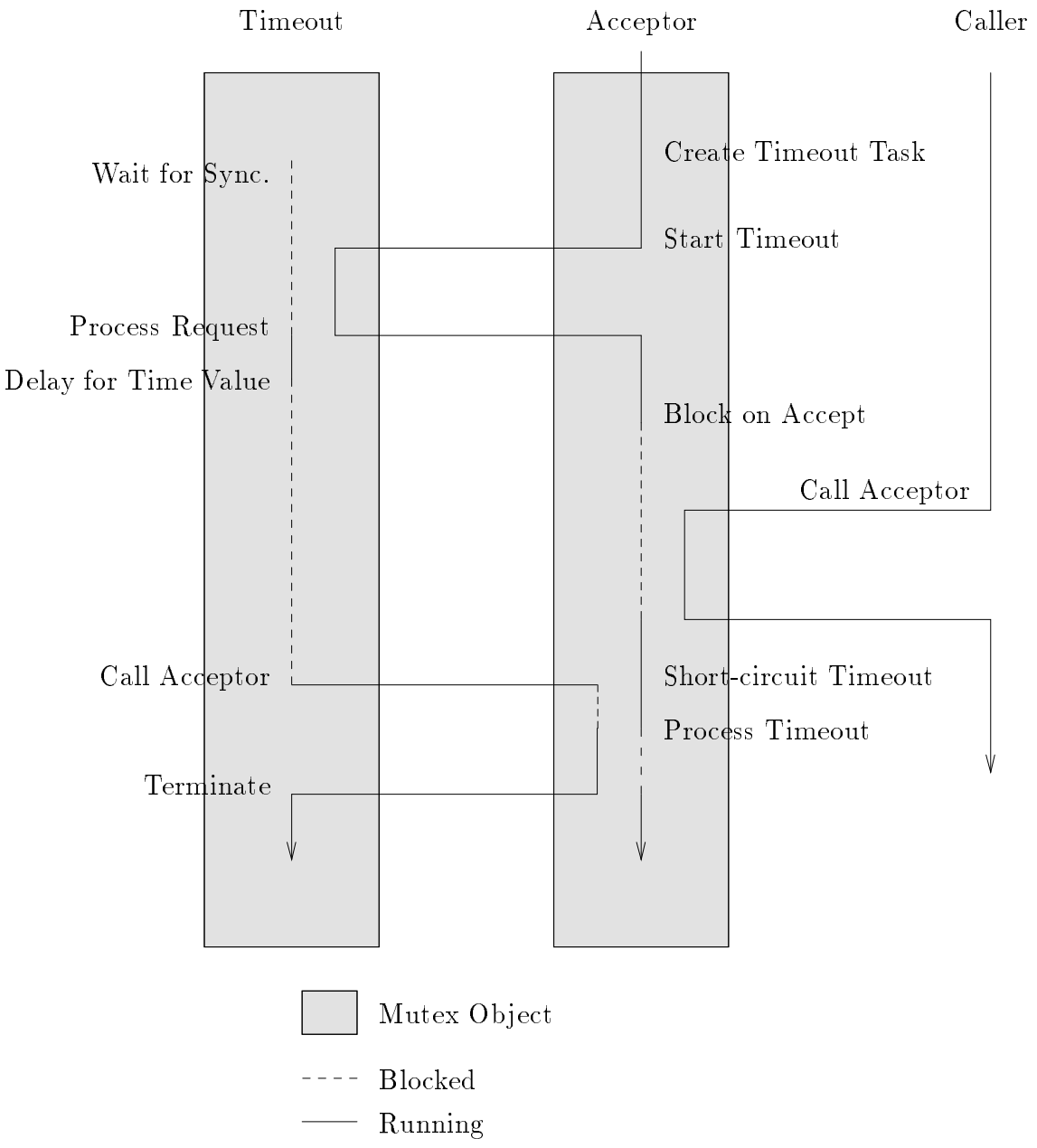

Figure 3.3: Timeout task is short-circuited

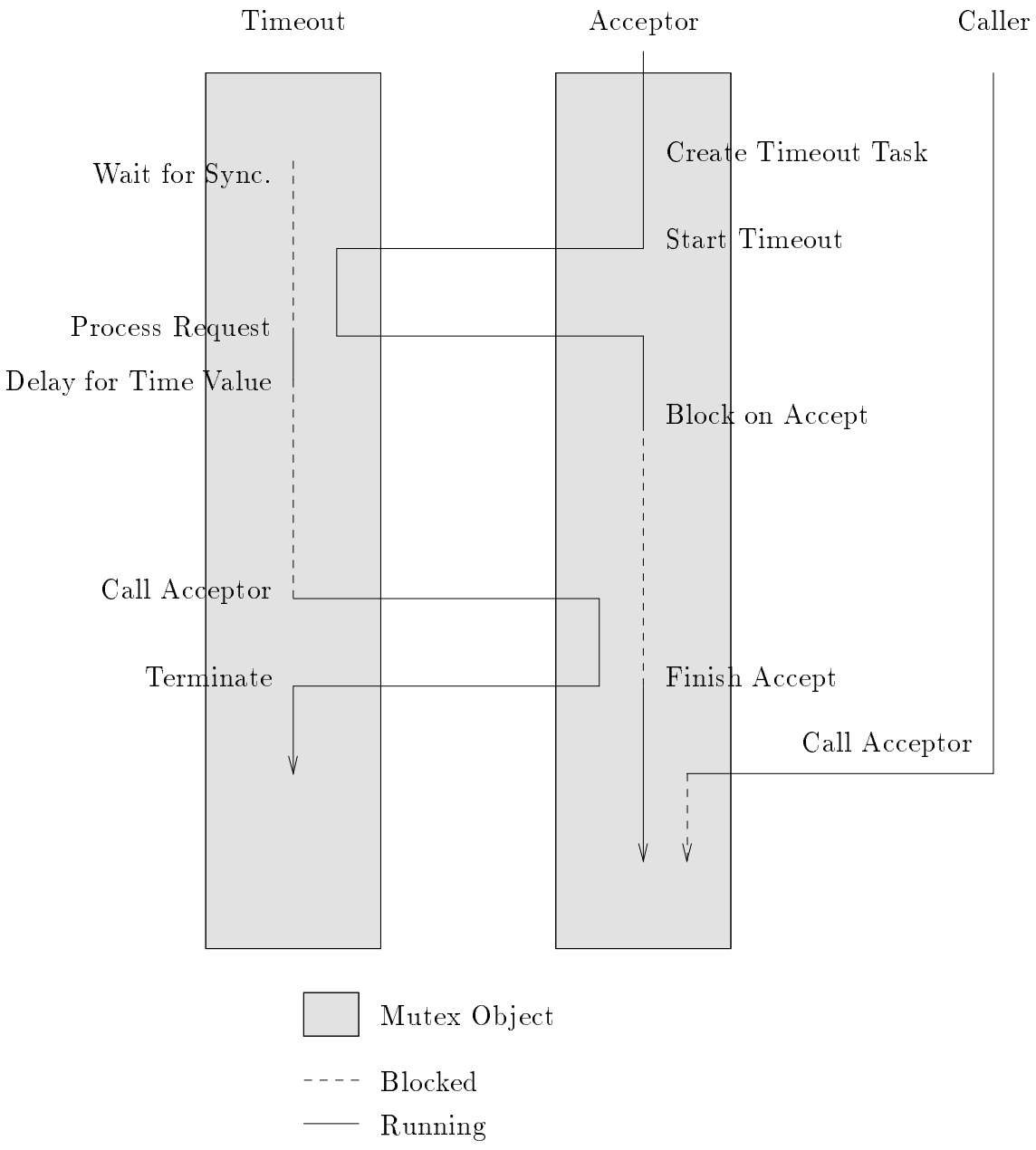

Figure 3.4: Timeout task expires before call occurs

The problem with this approach, however, is that it is complex to use from the user's point of view and incurs high overhead, including task creation and deletion and the additional synchronization to coordinate the timeout task, plus it requires the system to also implement a short-circuit capability.

3.4.2 Timeout Tasks: System Level Implementation

The easiest approach to extending accept statements to incorporate a timeout facility is for the system to perform the same actions as the user implementation. The first step is for the system to implicitly define an empty mutex member routine to be used for timeouts. Then, whenever the user specifies a timeout clause in an accept statement, the system automatically creates a timeout task. As described earlier, this task blocks for an amount of time specified by the user and then calls the mutex object. To avoid the problems associated with the potential race condition created by deleting the timeout task, the timeout task is short-circuited by immediately restarting the task and allowing it to call the mutex object.

In order to handle the situation where the timeout is short-circuited or expires after a valid call occurs the compiler inserts the required additional accept statement. Similarly, the additional call required to activate the initially blocked timeout task needs to be inserted as well. It might even be possible to reuse the timeout task for several timeouts to partially reduce the cost. While it is possible to hide most of the complexity from the user with this approach, especially if the compiler augments the accept statement code, it does not deal with all the concerns regarding overhead.

3.4.3 Kernel Level Implementation

The biggest overhead involved in the two previous methods is the creation and subsequent handling of a new task to implement the timeouts. A natural enhancement is to use the kernel task to perform the timeout activities. However, this approach suffers from a major problem, i.e., the kernel task is usually not allowed to block. This restriction exists because the kernel task is used to block and unblock tasks and it is impossible for the kernel task to perform this function for itself. Furthermore, the kernel task also performs other functions, like processing timer events, that require prompt service. So, it would be inappropriate for the kernel task to call a mutex member routine as this call might block, i.e., if a valid call has already occurred and the calling task is currently executing inside the mutex object, which leads to undefined behaviour.

This problem can be resolved by considering why the blocking is necessary. By calling a member routine, the kernel may have to block while waiting to execute this routine because another task is currently inside the monitor. But, the actual member routine called by the kernel is empty (i.e., no code), so there is no requirement that the kernel actually execute this routine. In fact, all the kernel, or even the timeout task, needs to do is to restart the acceptor when the timeout occurs (i.e., synchronize via rendezvous). For the kernel to perform this timeout functionality, a special type of time event or timeout is required; when this time event expires the kernel calls into the mutex object, similar to what a timeout task does when it wakes up. Now, if the kernel cannot immediately execute the empty member routine, then a valid entry call has already occurred and the timeout call can be

discarded and the kernel does not need to block. This suggests that a specialized entry routine can eliminate the problem of the kernel blocking. Furthermore, if a specialized entry routine is used, then calls associated with an expired timeout can be distinguished from general entry calls and an actual timeout member routine is unnecessary.

Using a specialized timeout entry routine to eliminate blocking also means that timeout calls are not queued. Therefore, unlike with the timeout task method, a subsequent accept statement is not required to deal with a timeout expiring in the window between the entry call occurring and the timeout being short-circuited because the timeout call is discarded. So, the technique described earlier to eliminate the race condition when timeouts are short-circuited is no longer necessary. Furthermore, as no subsequent accept statement is used, it is no longer advantageous to call the timeout member when the timeout is short-circuited, i.e., the timeout member is only called when the timeout actually expires. However, the outstanding timeout event must be dealt with by guaranteeing that if a valid entry call occurs, then the timeout entry call is uncallable until either the timeout has been short-circuited or until the kernel has finished executing the specialized timeout entry routine. Remember that when a valid entry call occurs, all mutex routines (including the timeout) become uncallable.

The obvious approach to enforcing this guarantee is to have the calling task short-circuit the timeout before it begins executing the member routine. However, this method incurs an additional penalty for all calling tasks because it is impossible for the calling task to know if the timeout facilities are being used and so it must

always check if the timeout needs to be short-circuited. For example, if a mutex member appears in an accept with a timeout and an accept without a timeout, the check must be inserted in the mutex routine 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 most practical solution is to remove the outstanding timeout before starting a new accept statement with a timeout for a particular mutex object. This solution satisfies the required guarantee because the timeout member is not acceptable again until this next timeout accept statement is executed. Thus, if the timeout does expire, as the kernel cannot gain immediate entry into the monitor, the timeout is discarded. This choice also limits any additional costs to only those accepts statements using the timeout facilities. The main drawback to this approach is that it requires compiler support in order to insert the call to short-circuit the timeout at the start of an accept statement using a timeout. The requirement that the kernel task must finish executing the specialized timeout entry routine before the short-circuiting is finished can usually be achieved by a locking mechanism, but tends to be implementation specific. I chose to adopt this final approach for implementing timeouts in μC++.

3.5 Implementation

In order to better understand my implementation of the described timeout facilities for μC++, an understanding of some of the relevant components of μC++ is required.

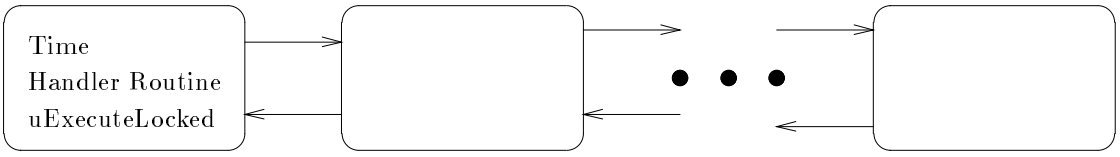

Figure 3.5: Nodes on the event queue

3.5.1 μC++ Background

In order to handle timeouts, a method of registering and processing time related events is necessary. The μC++ kernel provides support for registering time related events through the use of an event queue and interrupts. This section provides a description of the event handling facilities in μC++, as well as, a description of the implementation of mutex objects.

1. Event Queue and Interrupts.

   An *event queue* in μC++ is a time ordered list of *events* the kernel manages. These events can include time-slice context switches and requests by blocked tasks to be woken up at a specified time. Each node in the event queue specifies the time at which the event expires, a handler routine to be invoked at that time and a flag indicating whether the handler routine should be executed with the event queue lock acquired (see Figure 3.5). This additional flag is named uExecuteLocked and is discussed further on page 85. The handler routine is invoked by the kernel task and performs the required action depending on whether the node is associated with a context-switch, a delay, a timeout, etc.

   All access to the event queue is protected by a spinlock. The timing property

of the event queue is provided by an operating system timer. Nodes are added to the list in increasing order by time; the timer is set to expire at the time indicated by the node at the front of the list. When the timer expires, the currently executing task is interrupted and this task processes the event queue on behalf of the kernel. (In fact, there is no actual kernel task, but this co-opted task is referred to as the kernel task.) This task begins by acquiring the event queue lock, and then the next expired event is removed from the queue and the associated handler routine is invoked. This processing is repeated for all expired nodes on the event queue, as there could be more than one. The lock is released and reacquired between the processing of each node. This semantics allows other tasks the chance to manipulate the event queue without having to wait for the entire set of expired nodes to be processed, which is useful in minimizing the blocking time experienced by high priority tasks.

2. Mutex Objects.

   There are two kinds of mutex objects present in μC++, namely monitors and tasks. This discussion focuses on monitors, as a task can simply be considered a monitor with a thread of execution. (See chapter 2 for a thorough description of the semantics associated with a monitor.) A spinlock protects the monitor's entry queue and mutex queues. As well, the current task inside the monitor is considered the owner and has mutual exclusion over all internal data structures of the monitor. These internal data structures do not include the entry queue and the mutex queues.

Ownership of the monitor is typically changed in one of two ways. Control is either passed to a task made ready inside the monitor or control is passed to a task external to the monitor. In the first case, a task is removed from the internal acceptor/signalled stack and ownership of the monitor can be transferred without consulting any external queues. As no external data structures need to be consulted in this situation, the entry lock does not need to be acquired in order to change ownership.

In the second case, either the monitor is inactive and a task calling into the monitor becomes the new owner or the current owner hands ownership of the monitor to a task blocked on the entry queue. In both of these situations, the entry lock is acquired while ownership of the monitor is changed. If the monitor is inactive, the task calling into the monitor acquires the entry lock, sets itself as the owner of the monitor and subsequently releases the entry lock. Requiring the calling task to acquire the entry lock, even though it does not manipulate any of the external queues, is necessary to eliminate a potential race condition. If the entry lock is not acquired, then several tasks, seeing the monitor is currently empty, could try to set themselves as the new owner of the monitor.

If the monitor is active and the owner is exiting or blocking, then the owner acquires the entry lock so that it can examine the entry and mutex queues. If the owner cannot find an eligible task on the entry queue, the owner relinquishes control of the monitor, making the monitor inactive. Otherwise, the current owner selects the next eligible task from these queues and passes own-

ership of the monitor to this task. Then, the owner atomically wakes up this eligible task and releases the entry lock; at this point, the task is no longer the owner. The selected task wakes up already owning the monitor and can begin executing its mutex routine. Requiring the monitor owner to acquire the entry lock when selecting an external task is necessary to maintain the integrity of the entry queue and mutex queues because other tasks calling into the monitor also manipulate these data structures when blocking on the monitor.

A calling task that cannot enter the monitor immediately blocks on the mutex queue associated with its call, as well as the entry queue. There are a couple of reasons a calling task might block. Either a task is already inside the monitor or the member routine that the calling task invoked is uncallable for the accept statement currently being processed.

3.5.2 Data Structure Enhancements

In a μC++ mutex object, there are four significant data structures: the entry queue, the mutex queues, a set of flags indicating which calls to the mutex object are immediately acceptable, and a variable indicating which mutex routine was invoked by the current owner of the monitor. A mutex queue and a flag variable is associated with each mutex routine.

The extensions needed for the timeout facilities only require an additional flag variable to indicate if a timeout call is acceptable. The same variable indicating which mutex routine was invoked by the current owner of the monitor can also

indicate if the fictitious timeout routine is invoked. Furthermore, as timeouts must be accepted immediately or discarded, no queueing of timeout calls is possible; therefore, no mutex queue is required.

The fact that logically a monitor can only be blocked on one accept statement at a time means that it is only necessary to allow for one outstanding timeout node / event per mutex object. This policy is strictly enforced by always removing the timeout node at the start of every accept statement using a **uTimeout** clause. Unfortunately, it is impossible to only perform the remove if the timeout clause is actually activated because it is impossible to determine in advance if the timeout is needed, i.e., no accept clause will be immediately acceptable. Furthermore, a call to remove the timeout node must always occur if the timeout facility is used because it is impossible to statically (at compile time) determine if there is an outstanding timeout event. As well, attempting to remove the timeout node after the processing of the actual accept statement has begun, i.e., after determining that no accept clause is immediately acceptable, can result in a potential livelock situation (as spinlocks do not block).

This potential livelock arises because the accept statement is processed with the mutex entry lock acquired, as the entry and mutex queues need to be accessed. If the timeout node is removed after the entry lock is acquired, there is a window between acquiring the mutex entry lock and acquiring the event queue lock, in order to access the event queue, in which an outstanding timeout event could expire. If the timeout expires, the kernel acquires the event lock and then spins waiting to acquire the entry lock so it can process the timeout. But, the acceptor has already

acquired the entry lock and subsequently spins waiting for the event lock (see Figure 3.6).

This potential for livelock cannot be resolved by having the acceptor release the entry lock before trying to remove the timeout node. If the entry lock is released, then entry calls are allowed to proceed. If entry calls are set uncallable during the time the entry lock is temporarily released, then all calls block and the accept clauses need to be reprocessed, after the timeout has been removed, to detect if a subsequent valid call has occurred before the acceptor is allowed to block (see Figure 3.7). On the other hand, a valid entry call cannot be allowed to proceed, while the entry lock is temporarily released, as the acceptor has not finished processing the accept statement.

The most reasonable solution is to always attempt to remove the timeout node, if a **uTimeout** clause is present, before acquiring the mutex entry lock to process an accept statement. With this semantics, the additional overhead is not only small and fixed, but the potential for livelock is avoided. Furthermore, this overhead is only incurred by accept statements using the timeout facility. The only drawback is the occasional attempt to remove an unused timeout node, which does nothing (idempotent).

As well, to prevent an expensive and potentially blocking dynamic storage allocation, a timeout node is statically allocated inside every mutex object. As the design guarantees that there is at most one outstanding timeout event associated with a particular mutex object, it is possible to use the same timeout node for every accept statement in the object. Statically allocating the timeout node also elim-

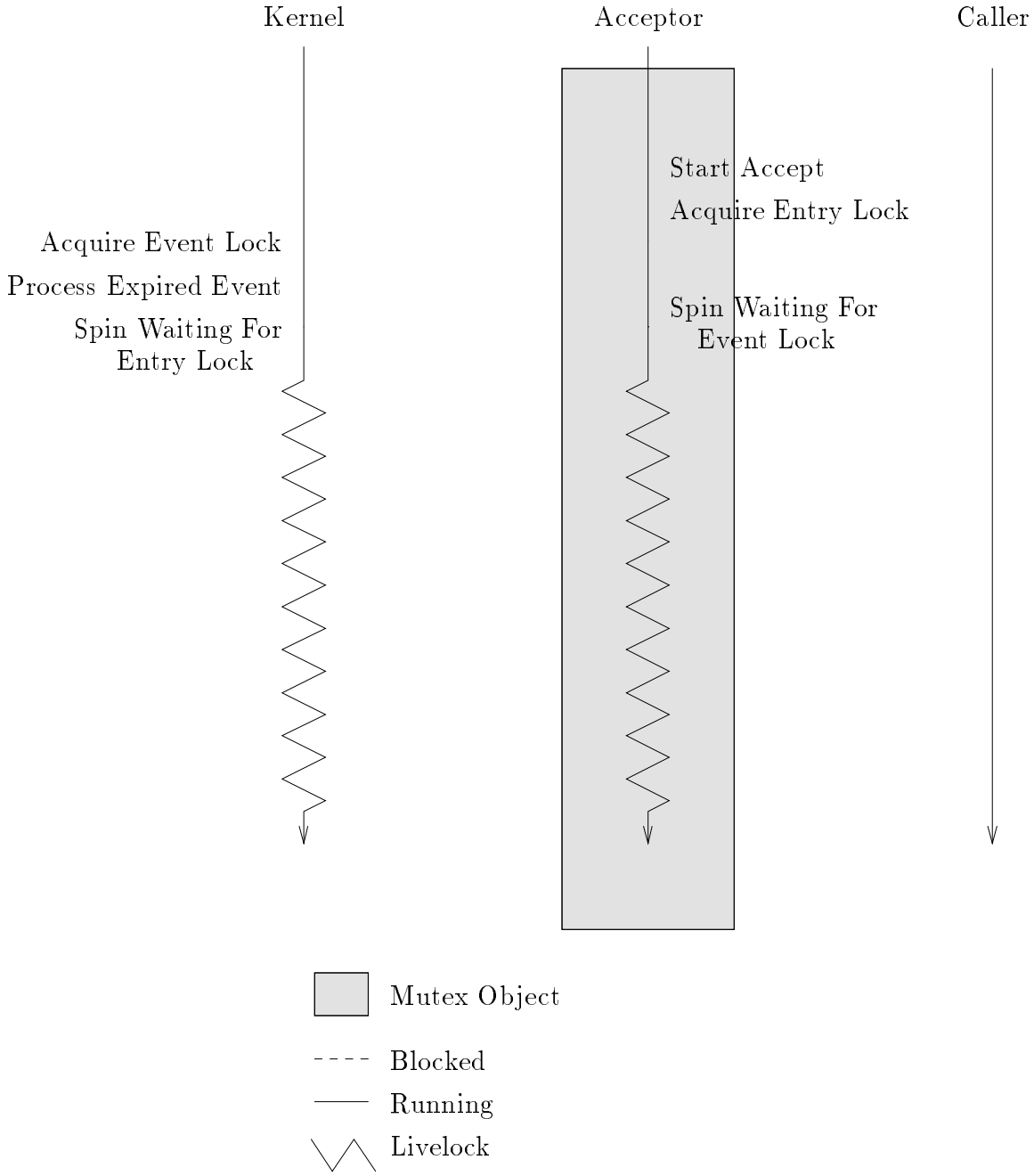

Figure 3.6: Livelock occurs if the timeout is removed after processing of accept statement.

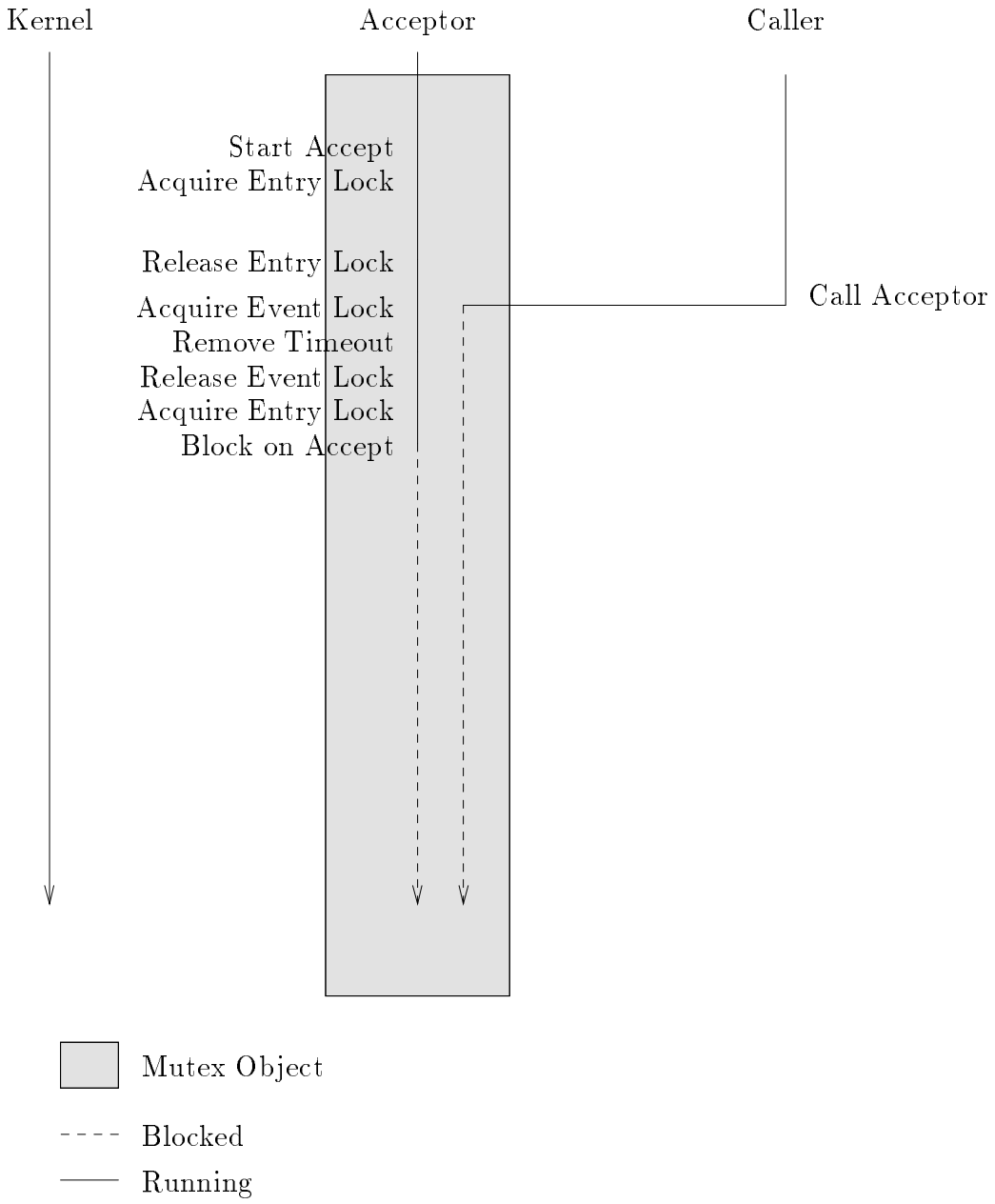

Figure 3.7: Entry calls can be missed if entry lock is released.

inates problems relating to scoping and allocation / deallocation associated with dynamic storage. Furthermore, statically allocating the node is not only more attractive from a real-time perspective, but it also has the advantage that it makes the remove operation idempotent. The drawback is that non-real-time mutex objects waste about 32 bytes of storage, which seems like an acceptable tradeoff.

3.5.3 Scenarios to Consider

In general, there are four scenarios that need to be dealt with when a task blocks on an accept statement with a timeout. These scenarios can be classified by considering which of the two events successfully calls into the monitor first. This classification is complete as both a timeout and an entry call must enter the monitor. Though a race can exist between a valid entry call and a timeout, access to the monitor is serialized because of the entry lock. Thus, any race must result in one of the last two cases (see scenarios 3 and 4 below), depending on which event gains control of the monitor first.

1. A call to a valid member routine occurs before the timeout expires.

2. The timeout occurs before a call to a valid member routine.

3. The timeout expires while a valid entry call is being processed.

4. A valid entry call occurs while the timeout is being processed.

Note that it is also possible to have nested accept statements. An example of a nested accept statement appears in Figure 3.8. For this example, the acceptor

```
uAccept(M1)
    uAccept(M2)
        x;
    uOr uTimeout(T1)
        y;
uOr uTimeout(T2)
    uAccept(M3)
        z;
```

Figure 3.8: Nested Accept Statement

blocks for time T2 waiting for a call to routine M1. If a call to M1 occurs, the acceptor blocks again, this time for time T1, waiting for a call to routine M2. If the call to M1 does not occur within time T2, the outer accept statement times out and then the acceptor blocks waiting for a call to routine M3. This nesting can be arbitrarily deep with any or all levels using the timeout facility. The implication of this nesting is that one accept statement may not be entirely completed before another begins.

As well, a calling task may also block on an accept statement with a timeout after entering a mutex routine. In this case, both the original acceptor and the calling task are accept blocked. As the original acceptor has already received a valid call, however, its timeout is no longer in effect. As discussed in chapter 2, accept blocked tasks are processed in LIFO order, so the calling task is restarted before the original acceptor. These two situations make it impossible to rely on the fact that the acceptor is restarted before another accept with a timeout occurs.

3.5.4 Implementation Details

The implementation can be divided into three parts, i.e., the acceptor, the calling and the timeout. Each part is considered individually and then the possible interactions are considered.

Acceptor:

Consider a sample μC++ accept statement:

```
uAccept(M1)
    x;
uOr uAccept(M2)
    y;
uOr uTimeout(T1)
    z;
```

As μC++ is a translator, this accept statement is expanded into C++ code (see Figure 3.9). The accept statement begins by creating an instance of uProtectAcceptStatement. The constructor for this class performs the setup required to begin processing the actual accept statement. An object is created so that its destructor can reset the object if an exception is thrown while processing the accept statement. The version of the constructor used for the timeout case (see Figure 3.10) begins with the acceptor attempting to remove the timeout node from the event queue by calling uRemoveEvent (see Figure 3.11) from uRemoveTimeout:

```
void uSerial::uRemoveTimeout() {
    if ( uEvents != NULL ) {          // make idempotent
        uEvents->uRemoveEvent( uTimeoutEvent, *uProc );
    } // if
} // uSerial::uRemoveTimeout
```

This step is bypassed if a timeout clause is not present in the accept statement by generating a different constructor without a call to uRemoveTimeout. Using a different constructor avoids additional runtime overhead that would be incurred by using a conditional expression. These constructors are distinguished by including an additional boolean parameter for the timeout case.

The constructor for uProtectAcceptStatement finishes by calling uAcceptStart (see Figure 3.10):

```
void uSerial::uAcceptStart( unsigned int &uMutexMaskPosn ) {
    lock.uAcquire();
    uMask.zero();       // set all mutex routines uncallable
    uMutexMaskPosn = uSerial::uMutexMaskPosn;
} // uSerial::uAcceptStart
```

to begin processing the accept statement. This processing begins with the acceptor acquiring the monitor entry lock and setting all mutex routines uncallable. The next step is to determine if any of the **uAccept** clauses are immediately acceptable (see Figure 3.9). The routine uAcceptTry is used to determine if any outstanding calls exist for a given mutex routine, and if no outstanding calls exist, to set the given mutex routine as callable. If no mutex member is immediately acceptable, then the timeout clause is processed by calling uAcceptPause (see Figure 3.12) with the associated timeout value. uAcceptPause calls uAddEvent:

```
void uEventList::uAddEvent( uEventNode &newAlarm, uProcessor &proc ) {
    uAcquireEventLock();
    // insert into sorted list, return queue position
    if ( uL.uInsert( &newAlarm ) == 0 ) {          // insert at the front ?
        proc.uSetTimer( newAlarm->timerT );        // reset system timer
    } // if
    uReleaseEventLock();
} // uEventList::uAddEvent
```

to add the timeout node to the event queue and then atomically blocks and releases the entry lock (see Figure 3.12).

When the acceptor is restarted, it finishes execution of uAcceptPause by calling uRemoveEvent (see Figure 3.11) to remove the timeout node from the event queue. The call to uRemoveEvent is idempotent, so it does not matter if the acceptor is restarted due to the timeout expiring or a valid entry call occurring. When uAcceptPause is completed, the block of code associated with the accepted clause is executed and this completes the accept statement (see Figure 3.9) by determining which mutex member was called and transferring to the associated code block for the **uAccept** clause.

Caller:

In μC++, every mutex routine in a monitor is expanded by adding a variable of type uSerialMember:

```
void mem ( ) {
  uSerialMember uSerialMemberInstance ( uSerialInst , uMutex05 , 0x05 ) ; {
    . . . // user code
  }
}
```

μC++ Statement	C++ Expansion
	{
	uSerial::uProtectAcceptStmt uPASInst (uSerialInst , **true**) ;
uAccept(M1)	**if** (uSerialInst.uAcceptTry (uMutex02 , 0x02)) {
	u0001 :
x;	x ;
	} **else** {
uOr uAccept(M2)	**if** (uSerialInst.uAcceptTry (uMutex03 , 0x03)) {
	u0002 :
y;	y ;
	} **else** {
	{
	uSerialInst.uAcceptTry ( ) ;
uOr uTimeout(T1)	uSerialInst.uAcceptPause ((T1)) ;
	}
	switch (uSerialInst.uMutexMaskPosn) {
	case 0x03 :
	goto u0002 ;
	case 0x02 :
	goto u0001 ;
	} *// switch*
z;	z ;
	} *// if*
	} *// if*
	} *// accept*

Figure 3.9: Expansion of Accept Statement

```
uSerial::uProtectAcceptStmt::uProtectAcceptStmt(uSerial &s, bool ) : s(s) {
    s.uRemoveTimeout();
    s.uAcceptStart( uMutexMaskPosn );
} // uSerial::uProtectAcceptStmt::uProtectAcceptStmt
```

Figure 3.10: uProtectAcceptStatement

```
void uEventList::uRemoveEvent( uEventNode &oldNode, uProcessor &proc ) {
    uAcquireEventLock();

    uEventNode *headNode;

    if ( ! oldNode.uListed() ) {                // node already removed ?
        uReleaseEventLock();
        return;
    } // if

    // remember current list head and then remove node
    headNode = uL.uHead();
    uL.uRemove( &oldNode );

    if ( headNode == &oldNode ) {               // removing at head ? reset alarm
        headNode = uL.uHead();
        if ( ! headNode ) {                     // list empty ?
            // cancel alarm
            proc.uSetTimer( uDuration( 0L, 0L ) );
        } else {
            // reset alarm
            proc.uSetTimer( headNode->timerT );
        } // if
    } // if
    uReleaseEventLock();
} // uEventList::uRemoveEvent
```

Figure 3.11: uRemoveEvent

```
void uSerial::uAcceptPause( uTime time ) {
    // handler to wake up blocking task
    uTimeoutHndlr handler( uThisTask(), *this );

    // initialize node for timeout event
    uTimeoutEvent.uExecuteLocked = true;
    uTimeoutEvent.timerT = time;
    uTimeoutEvent.uWho = &uThisTask();
    uTimeoutEvent.SigHandler = &handler;

    // add timeout to event list
    uProc = &uThisProcessor();
    uEvents = uProc->uEvents;
    uEvents->uAddEvent( uTimeoutEvent, *uProc );

    // lock is acquired at beginning of accept statement
    uBaseTask &uCallingTask = uThisTask();          // optimization

    // save acceptor thread of a rendezvous
    uLastAcceptor = &uCallingTask;

    // suspend current task on top of accept/signalled stack
    uAcceptSignalled.uAdd( &(uCallingTask.uMutexRef) );

    // unnecessary to set uMutexOwner to NULL because mask is open
    // find someone else to execute; release lock on kernel stack
    uActiveProcessorKernel->uSchedule( &lock );

    // remove timeout
    uEvents->uRemoveEvent( uTimeoutEvent, *uProc );

    // accepted entry is suspended if true
    if ( uCallingTask.uAcceptedCall ) {
        // acceptor resumes
        uCallingTask.uAcceptedCall->uAcceptorSuspended = false;
        uCallingTask.uAcceptedCall = NULL;
    } // if
} // uSerial::uAcceptPause
```

Figure 3.12: uAcceptPause

An object is used so the destructor can reset on exceptions. The constructor of uSerialMember:

```
uSerialMember::uSerialMember( uSerial &s, uBasePrioritySeq &e, int m ) :
                                                    s( s ) ) {
    // perform monitor entry code
    s.uEnter( mutexRecursion, e, m );

    acceptor = s.uLastAcceptor;
    uAcceptorSuspended = acceptor != NULL;
    if ( uAcceptorSuspended ) {
        acceptor->uAcceptedCall = this;
    } // if
    // avoid messing up subsequent mutex method invocation
    s.uLastAcceptor = NULL;
} // uSerialMember::uSerialMember
```

begins with a call to uEnter (see Figure 3.13) with a variable, mutexID, identifying which mutex routine the task called. uEnter begins by acquiring the monitor entry lock. If the called mutex member is callable, then the calling task sets all mutex member routines as uncallable, becomes the new monitor owner and releases the entry lock. Otherwise, the calling task adds itself to the entry queue and to the mutex queue for the called member routine and then it atomically blocks and releases the entry lock. When a blocked task is eventually restarted, it finishes execution of uEnter by making itself the monitor owner, and returns to perform the actual member routine code.

Once the mutex routine finishes, the tasks exits the monitor by invoking the destructor for uSerialMember, which calls uLeave (see Figure 3.14). uLeave first tries to hand ownership of the monitor to a task blocked inside the monitor. If no tasks are blocked inside the monitor, then the entry lock is acquired, the task at

the front of the entry queue is removed and made the new owner of the monitor and the entry lock is released. On the other hand, if there are no tasks blocked on the entry queue, then every mutex member routine is set as callable (except the special timeout entry) and the entry lock is released.

Timeout:

When the timer expires, the kernel task processes every expired node on the event queue by repeated calls to the iterator **operator>>** (see Figure 3.16) routine for the event list. This routine begins by acquiring the event lock. If the first event has expired, then this event node is removed from the event queue and its associated handler routine is invoked; otherwise the alarm is spurious and ignored. In general, the handler routine is invoked after the event lock is released, but for a timeout node, the timeout handler:

```
void uTimeoutHndlr::uHandler() {
    userial.uEnterTimeout();
} // uTimeoutHndlr::uHandler
```

is called with the event lock acquired. The uExecuteLocked variable in the actual node determines whether the handler for a specific node should be invoked with or without the event lock acquired.

The timeout handler calls uEnterTimeout (see Figure 3.15), the specialized entry routine discussed in section 3.4, for the associated monitor. uEnterTimeout begins by acquiring the entry lock. If the timeout call is too late, i.e., a call to an appropriate mutex routine has already occurred, then the entry lock is released and the handler

```
void uSerial::uEnter( uBasePrioritySeq &entry, int mutexID ) {
    lock.uAcquire();

    uBaseTask &uCallingTask = uThisTask();          // optimization
    if ( uMask.isSet( mutexID ) ) {                 // member acceptable ?
        uMask.zero();                               // clear all member valid flags
        mr = uCallingTask.uMutexRecursion;          // save previous recursive count
        uMutexOwner = &uCallingTask;                // set the current mutex owner
        lock.uRelease();
    } else if ( uMutexOwner == &uCallingTask ) {        // already hold mutex ?
        // another recursive call at the mutex object level
        uCallingTask.uMutexRecursion += 1;
        lock.uRelease();
    } else {                                        // otherwise block calling task
        // add to end of mutex queue
        entry.uAdd( &(uCallingTask.uMutexRef) );

        // remember which entry was called
        uCallingTask.uCalledEntryMem = &entry;

        // add mutex object to end of general entry deque
        uEntryList.uAdd( &(uCallingTask.uEntryRef) );

        // find someone else to execute; release lock on kernel stack
        uActiveProcessorKernel->uSchedule( &lock );
        mr = uCallingTask.uMutexRecursion;          // save previous recursive count
        uCallingTask.uMutexRecursion = 0;           // reset recursive count
    } // if
    uMutexID = mutexID;                             // set active mutex member
} // uSerial::uEnter
```

Figure 3.13: uEnter

```
void uSerial::uLeave( ) {
    uBaseTask &uCallingTask = uThisTask();          // optimization

    // recursively hold mutex ?
    if ( uCallingTask.uMutexRecursion != 0 ) {
        uCallingTask.uMutexRecursion -= 1;
        return;
    } // if

    // no tasks waiting re-entry to mutex object ?
    if ( uAcceptSignalled.uEmpty() ) {
        lock.uAcquire();

        // no tasks waiting entry to mutex object ?
        if ( uEntryList.uEmpty() ) {
            uMask.one();                            // accept all members
            uMutexOwner = (uBaseTask *)0;           // reset no task in mutex object
            lock.uRelease();
        } else {                                    // tasks waiting entry to mutex object
            // next task to gain control of the mutex object
            uMutexOwner = &(uEntryList.uDrop()->uGet());

            // remove member from front of entry queue
            uMutexOwner->uCalledEntryMem->uDrop();
            lock.uRelease();

            // wake up next task to use this mutex object
            uMutexOwner->uWake();
        } // if
    } else {                                        // tasks waiting re-entry to mutex object
        // next task to gain control of the mutex object
        uMutexOwner = &(uAcceptSignalled.uDrop()->uGet());

        // wake up next task to use this mutex object
        uMutexOwner->uWake();
    } // if
    uCallingTask.uMutexRecursion = mr;              // restore previous recursive count
} // uSerial::uLeave
```

Figure 3.14: uLeave

```
void uSerial::uEnterTimeout() {
    lock.uAcquire();

    if ( uMask.isSet( timeoutMutexID ) ) {        // timeout member acceptable?
        uMask.zero();                             // clear all member valid flags
        uMutexID = timeoutMutexID;                // set timeout mutex member

        // next task to gain control of the mutex object
        uMutexOwner = &(uAcceptSignalled.uDrop()->uGet());
        // wake up next task to use this mutex object
        uMutexOwner->uWake();
    } // if

    lock.uRelease();
} // uSerial::uEnterTimeout
```

Figure 3.15: uEnterTimeout

routine is complete. Otherwise, the timeout is accepted and the acceptor must be restarted. To begin, every mutex member routine is set as uncallable, similar to a call to any other callable mutex routine. Then the timeout is set as the calling routine and the acceptor is restarted as the owner of the monitor. The handler finishes execution by releasing the entry lock. Once the handler is finished, control returns to the **operator>>** routine (see Figure 3.16) and this routine completes by releasing the event lock.

3.5.5 Interaction Details

The four scenarios in Section 3.5.3 are now considered in detail with respect to a task blocking on an accept statement with a timeout, and this particular implementation approach.

1. A call to a valid member routine occurs before the timeout expires. In this

```
bool uEventListPop::operator>>( uEventNode *&node ) {
    uEL->uAcquireEventLock();

    // get event at the start of the list with the shortest time delay
    node = uEL->uL.uHead();

    if ( ! node ) {                                    // no events ?
        uEL->uReleaseEventLock();
        return false;
    } // if

    // event time delay greater than the start time for the iteration ?
    if ( uCheckTime(node->timerT) ) {
        uEL->uReleaseEventLock();
        return false;
    } // if

    uEL->uL.uRemove( node );

    if ( node->SigHandler ==
         (uSignalHandler *)(uThisProcessor().uContextSwitchHandler) ) {
        // remember the context switch event
        uCxtSwHandler = node->SigHandler;
        uEL->uReleaseEventLock();
    } else {                                           // not a ContextSwitch event
        if ( node->uExecuteLocked ) {                  // invoke handler with lock ?
            node->SigHandler->uHandler();              // invoke the handler for event
            uEL->uReleaseEventLock();
        } else {
            uEL->uReleaseEventLock();
            node->SigHandler->uHandler();              // invoke the handler for event
        } // if
    } // if

    return true;
} // uEventListPop::operator>>
```

Figure 3.16: operator>>

case, the outstanding timeout must be removed. As the timeout has not expired, the timeout is either removed when the acceptor is restarted at the end of the accept statement (this includes nesting by the acceptor (see Figure 3.8)) or by a subsequent accept statement with a timeout clause that occurs before the acceptor is restarted (this includes nesting that occurs when a calling task performs an accept). In the first case, the outstanding timeout is removed when the acceptor wakes up and finishes uAcceptPause (see Figure 3.12). In the second case, if a subsequent accept statement with a timeout clause occurs before the acceptor is restarted, then a call is made to remove the timeout node before the new accept statement is processed in uProtectAcceptStatement (see Figure 3.10). This call to remove the timeout before starting the new accept statement allows the same timeout node to be reused for all accept statements in the monitor and prevents old timeouts from affecting subsequent accept statements. In this case, however, the timeout node has already been removed by the acceptor associated with the subsequent accept statement. The fact that the initial acceptor still tries to perform the remove does not present a problem because the remove operation is idempotent.

2. The timeout occurs before a call to a valid member routine. As no entry call has occurred, the timeout acquires the entry lock and determines that it is a valid call (see uEnterTimeout, Figure 3.15). The timeout entry then makes all other mutex routines uncallable and sets the acceptor as the owner of the monitor. The timeout finishes by atomically waking up the acceptor and releasing the entry lock. When the acceptor unblocks, it tries to remove

the timeout event at the end of the accept statement (see uAcceptPause, Figure 3.12), but it is already removed. This extraneous remove is not a problem as the remove is idempotent. As well, all calls into the monitor are no longer acceptable and block.

3. The timeout expires while a valid entry call is being processed. The calling task begins uEnter (see Figure 3.13) by acquiring the entry lock and blocking calls to all mutex member routines. If a subsequent timeout occurs, it calls uEnterTimeout (see Figure 3.15) and must first spin until it can acquire the entry lock. When the timeout finally acquires the entry lock, the timeout call is already marked uncallable, so the timeout releases the entry lock and exits. If the owner of the monitor, either the acceptor or another task, tries to remove the timeout node it must wait for the kernel to release the event lock. As the timeout handler does not release the event lock until it is finished executing, in this scenario, the remove can only continue after the kernel has removed the timeout event and the handler is finished executing. At this point in time, the timeout node is no longer active and can be reused. Again, any extraneous removes at the end of accept statements are not a problem as the remove is idempotent.

   Furthermore, the only way for the timeout call to be made valid again is to execute another accept statement with a timeout clause. As this accept statement always begins with a call to remove the timeout, it is guaranteed that an old timeout is cancelled before timeout calls become valid again.

   There is no potential for deadlock when the monitor owner tries to remove the

timeout node while the timeout is being processed because only the timeout handler is allowed to own both the event lock and the entry lock simultaneously. This rule is maintained for all tasks, including the owner of the monitor. This results in the timeout handler owning the event lock while trying to acquire the entry lock and the monitor owner owning the monitor and trying to acquire the event lock. As the only common lock is the event lock, no potential for deadlock is introduced into the system. It is important to note that owning the monitor does not require the entry lock to be acquired. In fact, only tasks calling the monitor need the entry lock to guarantee mutually exclusive access to the various entry data structures. Thus, the entry lock is only acquired when these entry data structures are being read or manipulated.

4. A valid entry call occurs while a timeout is being processed. In this case, the timeout begins uEnterTimeout (see Figure 3.15) by acquiring the entry lock. The timeout marks all mutex routines uncallable and then makes the acceptor the owner of the monitor and releases the entry lock. During this time, any entry calls spin trying to acquire the entry lock. When an entry call does finally acquire the entry lock, either the mutex routine is marked as uncallable and the call blocks, or a new accept statement may have occurred, resulting in the mutex routine being made callable, and the call is allowed to proceed. In either case, the calling task cannot enter or even block waiting for entry until the timeout finishes, so no potential for deadlock is possible.

   The acceptor still removes the timeout node once it is woken up at the end

of the accept statement. This extraneous remove is fine because the remove is idempotent, and because the timeout handler is finished and in the process of releasing all the locks it owns. Thus, the event lock is eventually available for the acceptor when it calls remove.

Finally, the notion of nested accept statements with timeouts (see Figure 3.8) needs to be considered. In order to proceed to the next level of nesting, regardless of whether a valid call occurs or the accept statement times out, the acceptor must be restarted before the next accept statement can be executed. When the acceptor is restarted, if the timeout facilities are used, the acceptor always begins by removing the timeout node from the event queue (see Figure 3.12). Thus, it is impossible for the timeout to affect accept statements in subsequent levels of nesting. Furthermore, the timeout node is now available for use by any subsequent accept statements.

This design, however, does not require a call to remove when the acceptor is restarted. The call to remove the timeout node when the acceptor is restarted is solely a convenience used to reduce overhead, i.e., an extra call to remove the timeout node is less expensive than processing an interrupt associated with an expired timeout. When the acceptor is restarted, all mutex routines are uncallable because the monitor is currently active. Thus, if a timeout occurs, it is discarded. The only way for the timeout call to become valid again is for the monitor owner to block on an accept statement with a timeout. But, a call to remove the timeout node (Figure 3.10) always occurs before the actual accept statement is processed. Again, this prevents the timeout from affecting accept statements in subsequent

levels of nesting.

3.5.6 Analysis

The following considers the μC++ implementation of a timeout facility for accept statement with respect to the stated goals.

First, by carefully controlling the locking order no potential for deadlock (or livelock) is introduced into the system. Crucial to this idea is that the remove operation occurs before the entry lock is acquired by the monitor. Thus, an outstanding timeout request is removed before it expires or none of the locks required to process the timeout are held by the monitor owner until the kernel task has finished processing the timeout.

Unfortunately, the timeout facility does incur some costs regardless of whether it is actually used. Not only does every mutex object require the implicit definition of a specialized entry routine and a flag indicating when calls to the timeout are acceptable, but the fixed number of mutex routines supported by mutex objects in μC++ is further reduced by one (from 128 possible mutex routines to 127). As well, every mutex object contains a node used for timeouts.

These costs are reasonable, however, and have the added advantage that the timeout facilities are appropriate for use in a real-time system as no expensive dynamic allocations are required. The only parts of the timeout facility that are not fixed cost is adding a node to the event queue in timeout 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, however, and can be limited at the user level, and incorporated into the schedulability analysis of the system. 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 bounding the execution time relating to spinlocks have been proposed [13, 65].

Finally, the changes made to the system to support the timeout facility are small. By modelling the timeouts like a mutex member routine of an accept statement, much of the complexity needed to process timeouts is handled by the existing implementation of accept statements. Furthermore, any changes relating to how events are handled in the system should be applicable to extending other operations in μC++ with a timeout facility.

3.6 Summary

In order to use synchronization and communication operations in a real-time system, it must be possible to specify their worst-case execution time. One method to accomplish this goal is to extend these operations with a timeout facility. This timeout facility provides a method to terminate an operation if it does not progress to a certain point within a specified amount of time.

To this end, an extension to accept statements to include a timeout facility is proposed. This extension works by modelling the timeout after a call to a valid entry routine. This technique has the advantage of integrating the timeout facility more naturally with existing accept statement semantics and making the implementation

simpler as the existing accept statement functionality can be used.

The basic idea is to use a specialized entry routine to handle timeouts. So, if an accept statement has a timeout clause but no mutex routines are immediately acceptable, then a timeout request is made to the kernel task. After the specified time has elapsed, the kernel task calls this specialized mutex routine in order to restart the acceptor. If a valid call occurs before the timeout expires, then the timeout request needs to be short-circuited before the timeout call becomes valid again, i.e., a subsequent accept statement with a timeout clause occurs for the object.

The only remaining problem is to handle timeout requests that expire in the window between a valid call occurring and the timeout being short-circuited. By guaranteeing that the timeout request is short-circuited before a subsequent use of the timeout facilities in the object, the calls generated by a timeout expiring in this window can be discarded if they are not immediately valid. It is also interesting to note that these semantics are sufficient to allow the use of nested accept statements.

This timeout facility was implemented for accept statements in μC++. The actual modifications required to implement this timeout facility are reasonable and fit well into the existing μC++ implementation of accept statements. While this implementation does meet many of the required goals, some interesting issues remain outstanding. These issues include bounding spinlocks and limiting the execution time relating to ordered insertion on the event queue, which is a data structure issue.

Chapter 4

Practical 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 not only determine if a feasible schedule exists, but must also 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*. The basic idea behind this scheduling technique is to assign each task 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, at any point in time when the system needs to make a scheduling decision, the ready task with the highest priority is selected. This chapter considers some of the practical issues a system must deal with in order

to dispatch a set of tasks using priority-based scheduling.

4.1 Background

Priority based scheduling is an important part of real-time scheduling theory. While more sophisticated 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. Not only is priority-based scheduling much simpler to implement, but it is flexible enough to support a variety of *static* and *dynamic* scheduling algorithms. This section provides a brief introduction to priority-based scheduling algorithms, but for a thorough discussion of priority based scheduling algorithms for real-time systems refer to [12, 38, 47, 57].

Scheduling algorithms are typically categorized as either static or dynamic. 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 their ability to handle *aperiodic* tasks. Many real-time tasks tend to repeat the same set of actions with a specific frequency, for example, reading a set of sensors once every minute. These kinds of tasks are referred to as *periodic* tasks and lend themselves well to static analysis and scheduling. Aperiodic tasks, on the other hand, may still have timing constraints but tend to be more dynamic with unpredictable arrival times. Aperiodic tasks can have *hard* or *soft deadlines*. A deadline is considered hard if the consequences of missing the deadline is severe. A soft deadline serves as an indication of desired response time, so missing a soft deadline is not disastrous.

Ideally, a dynamic system should never miss any hard deadlines but still provide efficient service to tasks with soft deadlines.

A separate criteria that is often confused with static and dynamic scheduling is off-line and online scheduling [57]. According to Stankovic, et. al. [57], all scheduling algorithms have an off-line component that can range from producing a fixed schedule *a priori* with static scheduling to calculating appropriate task scheduling parameters for dynamic scheduling. As well, all scheduling algorithms have an on-line component that can range from dispatching the tasks according to a fixed schedule with static scheduling to determining if a new task can be scheduled with dynamic scheduling.

Scheduling decisions can be based on a variety of constraints and criteria. These criteria can include:

1. Period: The inter-arrival time between successive occurrences of the same task.

2. Computation Time: The worst-case execution time for an instance of the task.

3. Deadline: The time by which an instance of the task must be completed.

4. Importance: A value indicating the relative importance of the task.

5. Start Time: The time at which the task must begin execution.

A common class of scheduling algorithms are referred to as priority-based scheduling algorithms. These algorithms can be classified as either *fixed priority* or *dynamic priority* scheduling algorithms. This classification is independent of whether

a priority-based scheduling algorithm is used statically or dynamically, as defined above. With a fixed priority scheduling algorithm a task's priority is fixed during run-time, where as with a dynamic priority scheduling algorithm a task's priority can change as a task executes. 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. If this algorithm is clairvoyant these changes can be accounted for and each task can be assigned a fixed priority value. However, even with a clairvoyant dynamic priority scheduling algorithm, a task's priority typically changes during its execution for other reasons (see Section 4.1.2).

4.1.1 Fixed Priority Scheduling

One of the first fixed priority scheduling algorithms is the rate monotonic algorithm proposed by Lui and Layland [36]. For tasks to be scheduled by the rate monotonic scheduling algorithm, several conditions must be satisfied. First, each task must be periodic with its deadline equal to the end of its period. Furthermore, the tasks must be independent, i.e., no communication or synchronization, and preemptable. The rate monotonic algorithm assigns higher priorities to tasks with shorter periods. Lui and Layland proved that a set of n tasks are schedulable if $\sum_{i=1}^{n} C_i/T_i \leq n(2^{(1/n)} - 1)$, where C_i is the computation time of task i and T_i is the period of task i. However, tasks not satisfying this condition may still be schedulable. An exact characterization of the rate monotonic algorithm is provided in [30].

The deadline monotonic algorithm [34] can be used to schedule tasks with dead-

lines less than their period. With this algorithm, tasks with shorter deadlines are assigned higher priorities. Several schedulability tests are available for the deadline monotonic algorithm [3]. Again, tasks not satisfying these tests may be schedulable. Exact characterization is available by using an algorithm [4].

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 [32, 58], the priority exchange server [32] and the sporadic server [53]. *Slack stealing algorithms* [18, 31, 33, 61, 62] 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.

4.1.2 Dynamic Priority Scheduling

The most common dynamic priority algorithm is the earliest deadline first (EDF) algorithm, also proposed by Lui and Layland [36]. 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 [36]. Furthermore, a set of tasks is schedulable under the EDF algorithm if the total processor utilization is at most one, i.e., $\sum_{i=1}^{n} C_i/T_i \leq 1$.

The least slack time algorithm is another common dynamic scheduling algo-

rithm. 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 algorithms is that under transient overload conditions, unpredictable behaviour can occur, resulting in a potential cascade of missed deadlines.

Unfortunately, most practical real-time scheduling problems tend to be NP [40, 57]. These problems include 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 [28]. These heuristics can include 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 [21, 26, 54, 55, 62]. 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.

4.2 Implementing Priority Based Scheduling

Aside from the theoretical limitations imposed by the various priority scheduling algorithms, many practical issues also exist. The overriding criteria needed for a

real-time system is predictability. To achieve this predictability, it is important that all online scheduling operations are 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 overheads incurred for dynamic scheduling are much greater.

4.2.1 Implementing Fixed Priority Scheduling

Consider how fixed priority scheduling is typically supported in a real-time system. As the scheduling analysis takes place *a priori*, each task is assigned a static priority value from a fixed range. While it is possible to use an arbitrary range, typically the tasks are sorted using the appropriate criteria for the selected scheduling algorithm and assigned priorities consecutively starting at one. The number of priority values supported by the system is usually limited: 64 and 256 are common values. In order for the system to efficiently schedule an eligible task with the highest priority, the eligible tasks are placed on a priority queue. As the number of priorities is typically limited to a small number, 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 index value of the array. Tasks of a particular priority value are placed on the appropriate FIFO queue (see Figure 4.1).

The advantage of this type of priority queue is that it offers efficient, constant

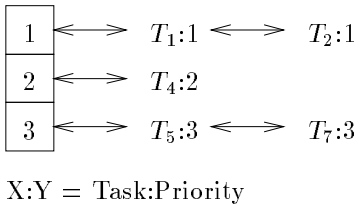

Figure 4.1: Example of an array based priority queue.

time operations. The desirable operations the priority queue needs to support are Empty, Insert, Delete, and Max/Min. With Max/Min, the actual operation required is dependent on the implementation of the priorities, i.e., if smaller numbers represent higher priorities then Min is required, otherwise Max is necessary. The Empty operation can be implemented in constant time by simply searching each level of the array for a non-empty priority queue. The cost for this search is bounded by the size of the array, which corresponds to the number of priorities supported. Using a doubly-linked list allows the insert and delete operations to be supported in constant time. For the delete operation, the task being removed from the queue needs to maintain a reference to its associated node, otherwise a search is required. Finally, the Max/Min operation, to locate the highest priority task, can also be supported in constant time. Similar to the idea behind the Empty operation, linearly searching the array for the non-empty queue of highest priority is a constant time operation. While more efficient implementations are possible for these operations, the point is that all the operations are constant time.

4.2.2 Implementing Dynamic Priority Scheduling

With a dynamic priority scheduling algorithm, however, scheduling decisions are typically based on the currently active task set. In this case, sorting the tasks and assigning priority values consecutively starting at one is impractical. First, new tasks entering the system may need to be inserted between existing tasks. If the priorities are assigned consecutively, then the priorities of the existing tasks need to be re-shuffled to accommodate this new task. Furthermore, with algorithms such as EDF, priorities need to be continually updated as tasks finish their execution for a particular period.

In addition to the overhead imposed by requiring the priorities of the tasks to be recalculated, there is also a *queue synchronization* problem, i.e., the various priority queue data structures required to schedule these tasks must also be updated. Priority queue data structures are associated with the ready queue, as well as other high-level constructs, like monitors and semaphores, that are modified in a real-time environment to expedite the entry of higher priority tasks (see chapter 2). Updating these priority queues as task priorities change can be very expensive because of the blocking that may occur as the appropriate locks are acquired. As well, some of these high-level constructs may have internal priority queues that cannot be modified without further changes to the construct, for example, condition queues inside monitors. This problem is further exacerbated because the worst-case execution time for modifying task priorities and updating the required priority queues, which can be implicit and hidden, 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.

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 might 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, while the priorities of the current tasks may need to shift in order to accommodate this new priority, 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 ordering of the tasks on the various priority queues remain unchanged. 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 more appropriate for dynamic priority scheduling, for example, the heap appearing in Figure 4.2. In this example, despite the fact the actual priority values associated with the tasks change, their ordering remains consistent, i.e., T_1, T_2, T_3, T_4, T_5, T_6, and T_7. Hence, the ordering of these tasks on the heap also remains consistent. While using a priority queue that allows the actual values of the keys to change without requiring an update may resolve the queue synchronization problem, it does not address the problem that the task priority values may be continually changing.

One possible solution to the shifting priorities problem is to assign priority values that do not need to change when new tasks enter the system. The easiest solution is to simply space out the priorities of the tasks currently in the system.

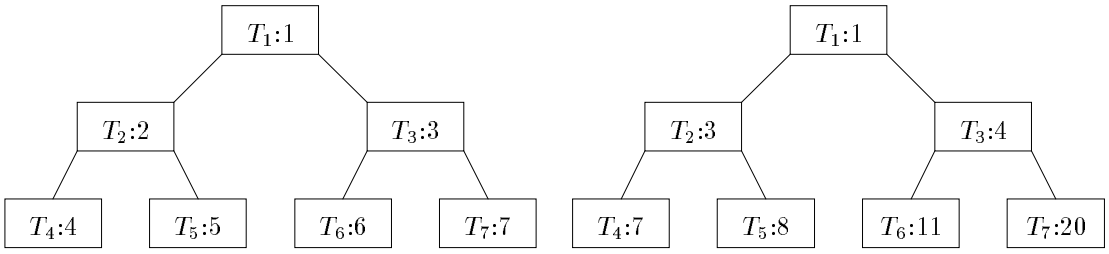

Figure 4.2: Example of relative ordering in a heap.

For example, rather than assigning priorities consecutively, such as 1, 2, 3, etc., they are spaced out over the entire range, such as 5, 10, 15, etc. 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 may eventually become full. Furthermore, this solution does not address the problem of tasks priorities shifting as part of the scheduling algorithm.

Another solution is to assign task priority values based on the actual characteristics used to order the task set. For example, if EDF scheduling is used, then rather than sorting the tasks according to their deadlines and then assigning an artificial number based on this ordering, assign the actual deadline value as the priority of the task, e.g., if task T_1 has deadline 13 and task T_2 has deadline 27, then T_1 is assigned priority 13 and T_2 is assigned priority 27. It is still desirable to limit the number of different priority values assigned to a value like 256, but the actual priorities can range over a much larger set of values. Typically, with this type of scheme, the number of priorities assigned is sparse compared to the size of

the range.

With the EDF scheduling algorithm, as a task's deadline is independent of other tasks in the system, new tasks entering the system could be assigned priorities without affecting the priorities of the existing tasks. This independence is typical of the task parameters used to make scheduling decisions. Furthermore, when a task does require its priority to be updated, this update can be performed independently of the other tasks in the system. For example, with the EDF scheduling algorithm, a task's priority can be changed to reflect its new deadline after it completes its execution for the current period without affecting the deadlines, and in turn the priorities, of any other tasks in the system, e.g., if task T_1 and task T_2 from above are periodic then at time 0 their priority is (13, 27), at time 13 their priority is (26, 27), at time 26 their priority is (39, 27), at time 27 their priority is (39, 54), etc. While the implicit priority, i.e., the relative ordering, of the remaining tasks may increase when the highest priority task finishes its execution, the actual priority values for these tasks remain constant.

The problem with this scheme is that there is a possibility of overflow if the actual priority values tend to become larger over time. This problem can be mitigated either by periodically reducing all task priorities by an equal amount or by using a large enough number of bytes to make this problem infeasible. For example, with EDF if the deadlines are specified in microseconds, then a 64 bit value is sufficient (and practical with newer hardware) as only positive priority values are needed.

However, the simple, constant time array based priority queue is impractical for any of these schemes. The space requirements for the array based priority queue

over a large range are impractical, especially because these priority queues can be associated with various high-level constructs.

A priority queue data structure that supports constant time operations and can take advantage of the sparse nature of the priorities used under this scheme is more appropriate. Maheshwari [37] performed a thorough evaluation of priority queue data structures to determine which algorithms are appropriate for a real-time environment. His results indicate that rings, heaps, D-trees and bit vectors are most suitable for a real-time environment. These algorithms are considered for their applicability to the priority scheme described above.

Rings

A ring based priority queue, is a circular list of priority ordered nodes. While this list can be implemented with either arrays or pointers, for better performance it should be doubly linked. Unfortunately, despite the fact that Max/Min is constant time, i.e., the first element in the list, insert and delete have O(n) worst-case execution time, where n is the number of nodes. This performance is not acceptable for reasonable sized queues.

Heaps

A binary heap [67] based priority queue is a complete binary tree satisfying the property that every parent node has higher priority than its children. A heap is typically implemented using an array, with the root of the tree as the first element of the array. With a heap, Max/Min is constant time, i.e., the first element of the array, and insert and delete have O(lg n) worst-case execution time. Efficient

implementations yielding reasonably good performance exist for heaps.

D-trees

A D-tree [37] can be viewed as an extension of a heap. A D-tree is also a complete binary tree, but the leaves are the elements of the priority queue and the interior nodes form a binary decision tree. In a D-tree, every parent node is assigned the higher value of its two children. A D-tree can also be implemented as 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. Again, Max/Min is constant time, i.e., the first element of the array, and insert and delete have O(lg n) worst-case execution time. According to Maheshwari [37], the performance of D-trees are similar to the performance of heaps, but somewhat faster. However, D-trees require about twice as much storage as heaps and are somewhat more complicated to implement.

Bit-vectors

Various type of bit vectors exist. They can range from a simple bit map to a Van Emde Boas [64] 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.

The subsequent analysis varies slightly from the analysis given in Maheshwari [37]. The biggest change is that the elements of the priority queues presented

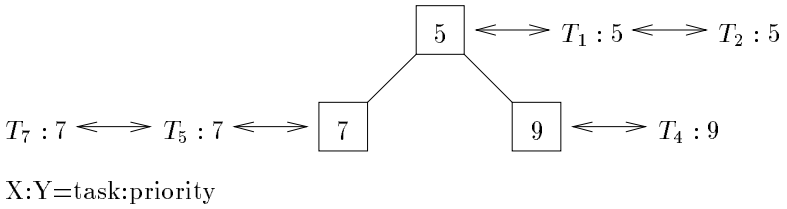

Figure 4.3: Using FIFO queues as elements of the heap.

in Maheshwari [37] are nodes representing the actual tasks. The problem with this approach is that the bounds presented for the algorithms are based on n, where n is the number of tasks. Allowing the system to support an arbitrary, or at least a large number of tasks, can result in large worst-case execution times.

My approach is based on an idea similar to that presented for the array based priority queue. Rather than using the elements in the priority queue to represent tasks, they represent FIFO queues for a particular priority level. 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 (see Figure 4.3). 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 also has the advantage that it is *stable*, i.e., all the tasks for a particular

priority value are processed in FIFO order. When the actual tasks are used as the elements of the priority queue, then many of the priority queue algorithms are not stable.

In terms of the priority queue data structures described above, not all of them are appropriate for use with the described priority scheme. First, the O(n) worst-case execution time for rings eliminates this data structure from consideration when more than a small number of priorities are supported. As well, the sparse usage of the large priority ranges associated with this scheduling technique make bit vectors impractical because at least one bit must be allocated for every value in the range. Therefore, bit vectors are also eliminated from consideration due to the large memory requirements of these algorithms. Thus, it would seem that heaps or D-trees are the best choices for implementing the described scheduling technique.

4.3 μC++ Implementation Details

For the μC++ implementation, heap based priority queue data structures were chosen. Even though it has been shown that D-trees have slightly better performance, heaps are simpler to implement, have half the memory requirements and have execution time graphs similar to D-trees [37].

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 not only needed for inserting and deleting elements in the heap but also for tasks to access their

associated FIFO queue. Unfortunately, it is impossible for each task to maintain a reference to the heap node containing their associated FIFO queue because, as the heap is modified, it is expensive to keep these references in the tasks updated.

In order to minimize this update problem, two array data structures are used (see Figure 4.4). The first array is the actual heap, but the elements of the heap point to the nodes in the second array. Each element of the second array consists of a FIFO queue and a reference back to its associated node 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 references in the second array need to be maintained. For any element that is on the heap, its reference into the second array is fixed while it is 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. Tasks in the system then only have to maintain a reference to the node in the second array associated with their priority value. Through this node, tasks can access both their associated FIFO queue and their node on the heap in constant time. Similar to the reasoning explaining why the elements in the first array do not need to be updated, these references in the task 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

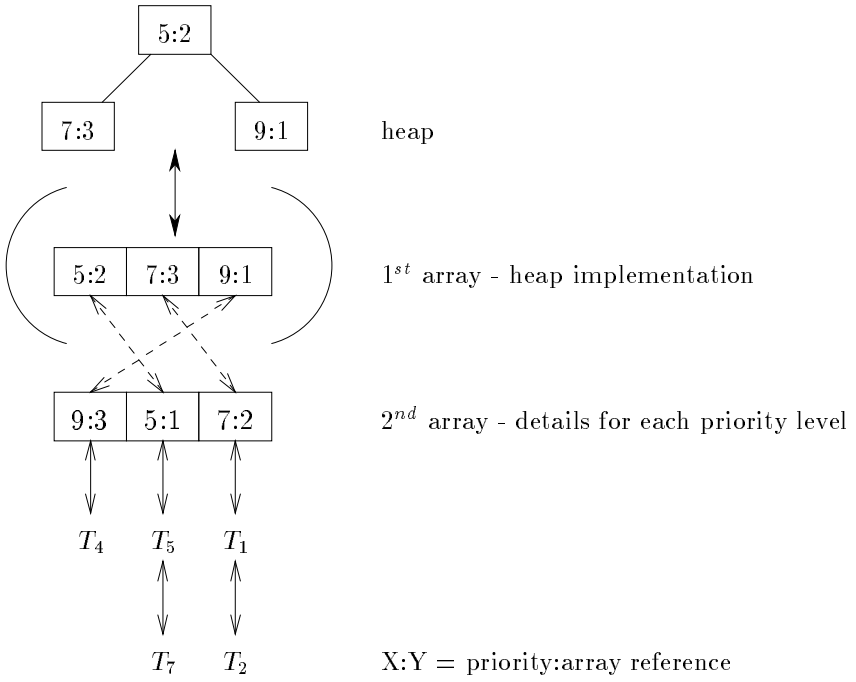

Figure 4.4: 2 array FIFO queue heap.

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(n), where n is the number of priorities, it usually only needs to be performed when a task is first created or when a task's priority value changes. Therefore, even though this technique does not eliminate the find and reference problems, they are both reduced to reasonable levels, in apparently 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 tasks have a reference to the node associated with their 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 n) operation. Maintaining the heap in this manner, still allows it to return the non-empty 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 n) delete operation must be performed on the heap.

The problem that arises 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., monitors and semaphores. 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 enough, 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 a 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.

4.4 Summary

Priority based scheduling is an important part of real-time programming. Techniques exists to schedule both periodic and aperiodic tasks with both hard and soft deadlines. These algorithms can be classified as either fixed priority or dynamic priority scheduling algorithms. While implementing fixed priority scheduling is straightforward, dynamic priority scheduling is significantly more complicated.

As dynamic priority scheduling is typically only based on the active task set, tasks entering the system may change the ordering of this task set. Furthermore, with dynamic priority scheduling, the priorities of the tasks change during execution. While these situations may change the implicit priority of the tasks, the relative ordering of the remaining tasks stays unchanged. By using the actual scheduling parameters, such as deadline, as the actual priority value of a task, a task's actual priority value is independent of the other tasks in the system. Therefore, a task's priority can be modified or tasks can be added to or removed from the system without affecting the priority value of other tasks in the system.

Unfortunately, the problem with this scheme is that the range of priorities that need to be handled is much larger, thus the array based priority queues used for the implementation of fixed priority scheduling are no longer appropriate. A priority queue data structure that is independent of the range of acceptable priority values is more appropriate, such as heaps or D-trees. As well, by using FIFO queues for each priority level as elements of the priority queue, rather than using the actual

tasks, a small, fixed worst-case execution time is obtained for all the priority queue operations. Using FIFO queues as elements of the priority queue also has the added advantage that heap and D-tree based priority queues become stable.

This scheduling technique is implemented in μC++ using heap based priority queues. However, the problem with using a heap is that the find operation has O(n) worst-case execution time. In order to minimize the impact of an expensive find operation, a second array is used for the FIFO queues. Then, both the heap and the tasks reference this second array to access the appropriate FIFO queue for a particular priority value. As the FIFO queues do not shift within this second array, the update problem is reduced to keeping the second array synchronized with the heap.

Using this second array means that tasks only needs to perform an expensive find when they are first created or when their priority changes. In all subsequent references to the priority queue, the appropriate nodes can be accessed directly. As well, forcing the tasks to use the same index value for the priority queues associated with other mutex objects allows these priority queues to be accessed without incurring any additional overhead.

Chapter 5

Priority Inheritance

Many priority-based scheduling algorithms assume the tasks being scheduled are independent; however, this assumption is typically unrealistic. In order to maintain the integrity of task interactions, critical sections are used to serialize access to shared resources, for example, data. The problem that occurs with critical sections is that a low priority task, using a shared resource, can delay the execution of a high priority task trying to access this same resource. As mentioned in chapter 2, this situation is referred to as priority inversion. While, in general, it is impossible to eliminate priority inversion, it is important to bound the duration of this inversion. Unbounded priority inversion is a serious problem making it impossible to guarantee the schedulability of a system.

Consider the following example: Two tasks share a resource, a low priority task T_L and high priority task T_H. T_H tries to acquire the resource but it is already held by T_L. In this situation, T_H must block until T_L releases the resource. However, as T_L executes at a low priority, its execution can be indefinitely delayed by tasks

executing at intermediate priorities. In this situation, the blocking time experienced by T_H is potentially unbounded.

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 and to limit the duration of priority inversion. The details regarding when a task's priority is raised and by how much vary depending on the actual priority inheritance protocol chosen.

This chapter discusses the basic priority inheritance protocol. This protocol is chosen for the following reasons. First, the basic priority inheritance protocol forms the base of other more complicated protocols. Second, the implementation of this basic protocol in not as straightforward as it might first appear.

5.1 Background

In order to address the problem of unbounded priority inversion, Rajkumar [50, 51] proposed several priority inheritance protocols. This notion of priority inheritance has been expanded and extended in various ways [5, 6, 15, 16, 44]. This section provides a brief introduction to a selection of these protocols (uniprocessor only).

5.1.1 Basic Priority Inheritance Protocol

The idea behind the basic priority inheritance protocol [44, 50, 51] 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 be scheduled. This technique bounds the length of any priority inversion and allows the worst-case execution time of a task to be specified. In fact, Rajkumar [44, 50, 51] provides sufficient conditions to allow a non-independent task set to be scheduled using the rate monotonic scheduling algorithm.

Consider the task set in Table 5.1. Each row of this table consists of the name and priority (1 is the highest priority) of a task in the system, as well as, the resources used by that task. Assume execution begins at time t_0 with both resources available and that these resources support the basic priority inheritance protocol. Suppose the sequence of events in Figure 5.1 occurs. Figure 5.2 shows the state of the resources at each of the listed events. At time t_0, execution begins and both resources are available. At time t_1, task T_3 successfully acquires resource R_1. At time t_2, task T_2 successfully acquires resource R_2. At time t_3, task T_1 blocks trying to acquire resource R_1 as it is already acquired by task T_3. When T_1 blocks, task T_3's priority is increased to the priority of T_1. At time t_4, task T_3 blocks trying to acquire R_2 as it is already acquired by task T_2. When T_3 blocks, task T_2's priority is increased to T_3's current priority, i.e., the prioirity of T_1. When task T_2 releases resource R_2 at time t_5, its priority is lowered back to its original priority and task T_3 is able to acquire R_2. When task T_3 releases resources R_1 and R_2 at time T_6, its priority is lowered back to its original priority and task T_1 is able to acquire R_1. Finally, at time t_7, task T_1 releases R_2.

Intuitively, this technique works because intermediate priority tasks can no longer preempt the execution of lower priority tasks if these lower priority tasks

Task	**Priority**	**Resources**
T_1	1	R_1
T_2	2	R_2
T_3	3	R_1, R_2

Table 5.1: Sample task set.

1. At time t_1, T_3 tries to acquire resource R_1.
2. At time t_2, T_2 tries to acquire resource R_2.
3. At time t_3, T_1 tries to acquire resource R_1.
4. At time t_4, T_3 tries to acquire resource R_2.
5. At time t_5, T_2 releases resource R_2.
6. At time t_6, T_3 releases resources R_1 and R_2.
7. At time t_7, T_1 releases resource R_1.

Figure 5.1: Sample run sequence.

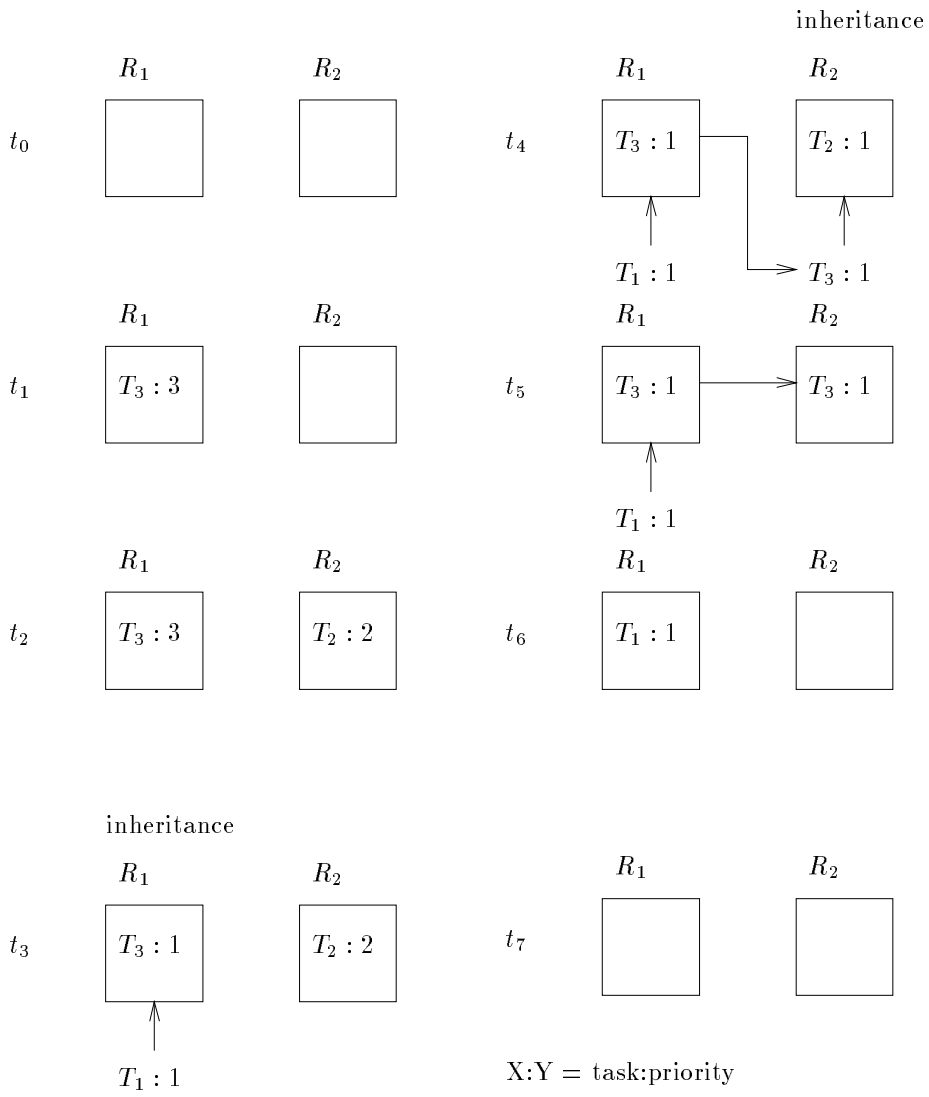

Figure 5.2: Basic Priority Inheritance Example.

are preventing the execution of higher priority tasks 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. A task T_A is indirectly blocked by a task T_B if there is a chain of blocked tasks leading from T_B to T_A. For example, if T_A is blocked by task T_C and T_C is blocked by T_B, then T_A is directly blocked by T_C and indirectly blocked by T_B.

However, deficiencies exist with the protocol. First, despite the fact that this protocol bounds the length of priority inversion, the actual blocking time experienced by tasks can be long because this protocol does not prevent multiple blocking. *Multiple blocking* refers to the fact that a task may need 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 should be available after the first time a task blocks. Second, this protocol does not avoid deadlock, an interesting side effect provided by some of the more sophisticated priority inheritance protocols. The biggest advantage of this protocol, however, is that it can be implemented without requiring any additional system information. Thus, it works well with online scheduling.

5.1.2 Priority Ceiling Protocol

With the priority ceiling protocol [44, 50, 51], every shared resource is assigned a value referred to as a priority ceiling. This priority ceiling value is the priority of the highest priority task that can access this resource. The priority ceiling protocol

extends the basic priority inheritance protocol by specifying when a task is allowed to acquire a resource. A task is only allowed to acquire a resource if the resource is available and the task's priority is higher than the priority ceiling of all resources held by all the other tasks in the system. This guarantees that the priority of the task acquiring a resource is higher than the potential inherited priority of all other tasks currently holding critical sections that it might preempt. If a resource is free but a task is forced to block because its priority is not high enough to acquire the resource, then this task is considered to be blocked by the task owning the resource with the highest priority ceiling. Given this modification to the notion of blocking, task priorities are then modified according to the method described in the basic priority inheritance protocol.

Consider the sequence of events in Figure 5.3 with the resources supporting the priority ceiling protocol. The priority ceiling values for each resource is given in Table 5.2. Figure 5.4 shows the state of the resources at each of the listed events. At time t_0, execution begins and both resources are available. At time t_1, task T_3 successfully acquires resource R_1. At time t_2, task T_2 blocks trying to acquire resource R_2 as its priority is less than the ceiling value of R_1. T_2 is considered blocked by T_3, and hence, T_3's priority is raised to the priority of T_2. At time t_3, task T_1 blocks trying to acquire resource R_1 as it is already acquired by task T_3. When T_1 blocks, task T_3's priority is increased to the priority of T_1. At time t_4, task T_3 acquires resource R_2. Note that unlike basic priority inheritance, T_3 is not required to block before it can acquire R_2. When task T_3 releases resources R_1 and R_2 at time t_5, its priority is lowered back to its original priority and task T_1 is

Resource	Ceiling
R_1	1
R_2	2

Table 5.2: Ceiling Values.

1. At time t_1, T_3 tries to acquire resource R_1.
2. At time t_2, T_2 tries to acquire resource R_2.
3. At time t_3, T_1 tries to acquire resource R_1.
4. At time t_4, T_3 tries to acquire resource R_2.
5. At time t_5, T_3 releases resources R_1 and R_2.
6. At time t_6, T_1 releases resource R_1.
7. At time t_7, T_2 releases resource R_2.

Figure 5.3: Sample run sequence.

able to acquire R_1. T_2 still cannot acquire R_2 as its priority is less than the priority ceiling of R_1, so it is now considered blocked by task T_1. At time t_6, task T_1 releases R_1 and T_2 can acquire R_2. Finally, at time t_7, task T_2 releases R_2.

The priority ceiling protocol addresses the shortcomings discussed above for the basic priority inheritance protocol. First, this protocol eliminates multiple blocking because a task can only be blocked for the duration of at most one critical section. As well, the restrictions determining when a task can acquire a resource also provide sufficient conditions to avoid deadlock. Rajkumar [44, 50, 51] also provides sufficient conditions to allow a non-independent task set to be scheduled using the rate monotonic scheduling algorithm with the priority ceiling protocol. This analysis is extended to EDF scheduling by Chen [16].

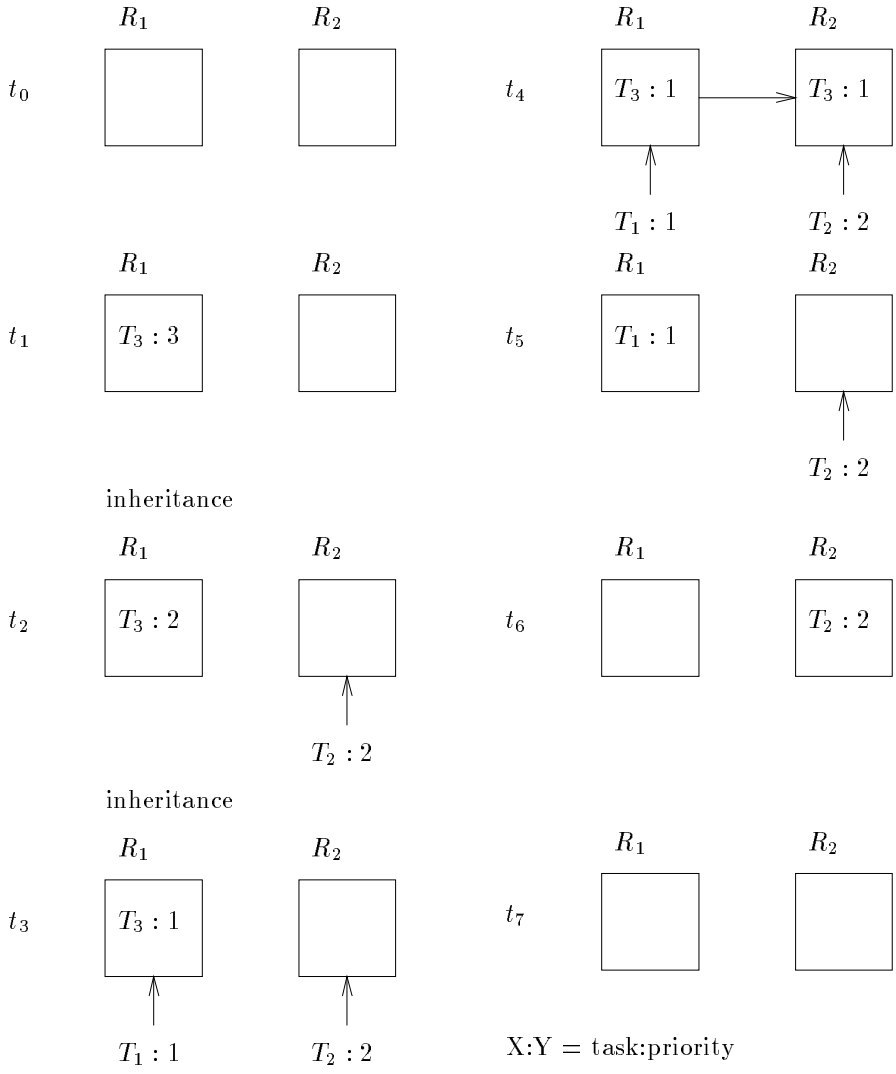

Figure 5.4: Priority Ceiling Example.

However, several drawbacks exist with this protocol. First, the priority ceiling protocol not only incurs more run-time overhead than the basic priority inheritance protocol, but it also requires more static analysis. In addition to the task information required by the scheduling algorithm, the appropriate priority ceiling value for each resource must be determined. Calculating this ceiling value requires complete information about all resources used by each task. However, as a task's run-time behaviour is dynamic, statically determining the set of resources actually accessed by a task is impossible. Instead, a pessimistic approach is typically employed; every resource that a task could potentially access is included in the set.

Second, this protocol is a pessimistic approach because the resource needs change during run-time, therefore this protocol can be unnecessarily restrictive, as well as, difficult to use with an online algorithm. For example, in Figure 5.4 task T_2 is prevented from executing despite the fact that task T_1 does not require R_2 during this time. This problem occurs because this protocol provides sufficient conditions to prevent multiple blocking and deadlock, but does not provide sufficient and necessary conditions. While the semaphore control protocol [45, 44] does provide sufficient and necessary conditions to meet these goals, the trade off is that it requires detailed resource information for each task. To avoid being overly restrictive, in addition to the information required by the priority ceiling protocol, extended resource information for each critical section is also required. Thus, a request for an available resource is only blocked when granting the resource would cause a higher priority task to block.

5.1.3 Immediate Ceiling Priority Protocol

The immediate ceiling priority protocol [5, 6, 51] is a simplification of the priority ceiling protocol. In this case, every resource is again assigned a priority value corresponding to the priority of the highest priority task that can use the resource. But when a task acquires a resource, its priority is immediately raised to this ceiling value while the task is using the resource, rather than the priority of the highest priority blocked task. This simplification removes much of the complexity and overhead involved with the first two inheritance protocols, but leads to its own form of priority inversion, i.e., intermediate priority tasks can be preempted by a low priority task using a resource with a high priority ceiling even if this low priority task is not blocking a high priority task. For example, in Figure 5.5 task T_3's priority is increased to the ceiling value of R_1 at time t_1 even though it is not blocking the execution of any higher priority tasks. In this case, priority inversion occurs because task T_3 is preempting the execution of higher priority tasks.

Consider the sequence of events in Figure 5.3 with the resources supporting the immediate ceiling priority protocol. The priority ceiling values for each resource is given in Table 5.2. Figure 5.5 shows the state of the resources at each of the listed events. At time t_0, execution begins and both resources are available. At time t_1, task T_3 successfully acquires resource R_1. Its priority is raised to the ceiling value for R_1. Ideally, at time t_2, task T_2 would like to acquire resource R_2. However, as task T_3 is executing at higher priority than T_2, T_2 is not scheduled and thus must wait to acquire resource R_2. At time t_3, task T_1 blocks trying to acquire resource R_1 as it is already acquired by task T_3. No further priority inheritance

is necessary when T_1 or any other task blocks on a resource. At time t_4, task T_3 acquires resource R_2. Again note that unlike basic prioirity inheritance, T_3 is not required to block before it can acquire resource R_2. When task T_3 releases resources R_1 and R_2 at time t_5, its priority is lowered back to its original priority and task T_1 is able to acquire R_1. Task T_1 is scheduled to execute when T_3 releases R_1 because its priority is higher than T_2. Note that the decision to allow T_1 to proceed is not affected by the fact that R_1 has a higher ceiling value than R_2, because neither task has acquired R_1 or R_2 at the time the decision is made. T_1's priority remains unchanged as it is already executing at the ceiling value for R_1. At time t_6, task T_1 releases R_1 and task T_2 can acquire resource R_2 (assuming that T_1 relinquishes the processor). Finally, at time t_7, task T_2 releases R_2.

It is interesting to note that both the priority ceiling protocol and the immediate ceiling protocol prevent task T_2 from acquiring resource R_2 at time t_2, and hence, prevent multiple blocking. The difference is that with the priority ceiling protocol, T_2 is considered blocked by T_3, resulting in T_3's priority being increased. With the immediate ceiling priority protocol, however, as T_3's priority is raised to the resource's ceiling value on entry, there is no advantage to defining T_2 as being blocked by T_3. However, preventing T_2 from executing even before it tries to acquire R_2, with the immediate ceiling protocol, results in the kind of priority inversion described above.

Like the priority ceiling protocol, this protocol also prevents multiple blocking, however, it only avoids deadlock under certain conditions, i.e., if tasks do not suspend while holding a resource. While this protocol does not suffer from the run-

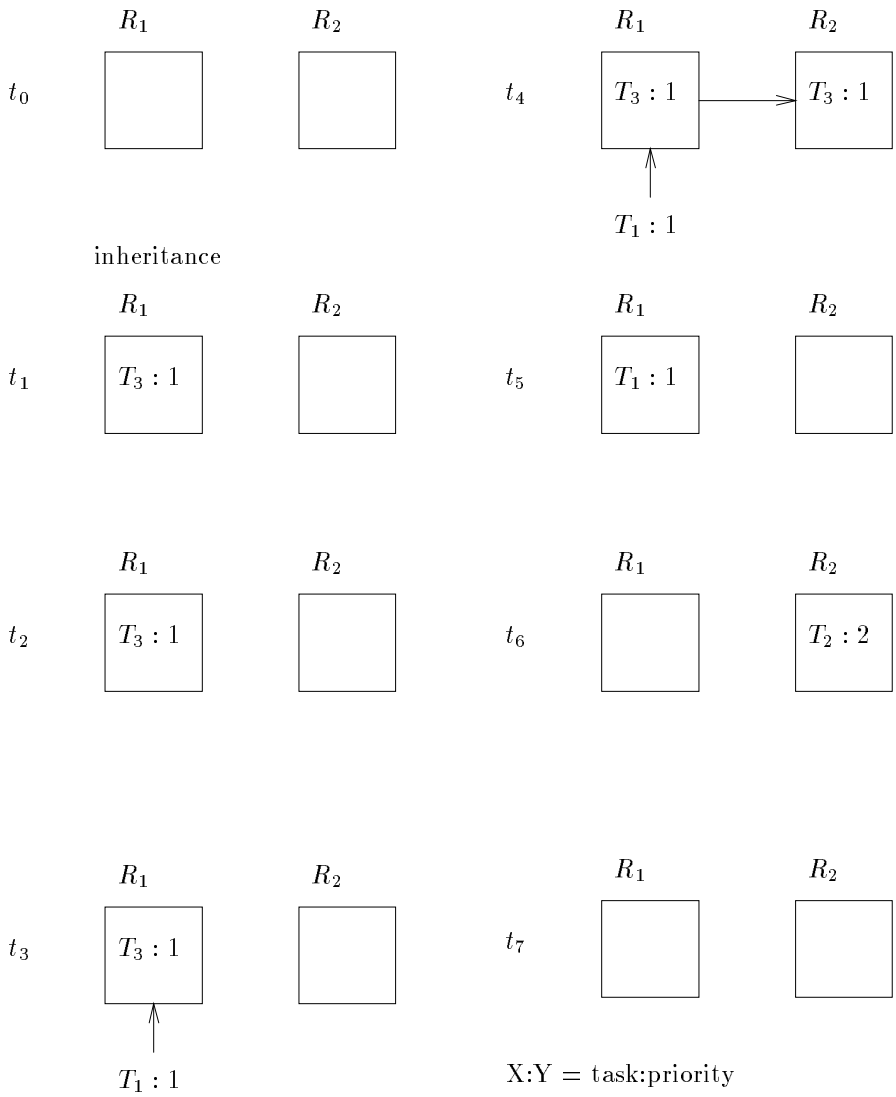

Figure 5.5: Immediate Ceiling Example.

time overheads associated with the priority ceiling protocol, it does suffer from the disadvantages associated with (statically) calculating ceiling values. As well, the priority inversion resulting from increasing a task's priority directly to a resource's ceiling value is comparable to the overly restrictive behaviour of the priority ceiling protocol. In both cases, a higher priority task is prevented from executing despite the fact that a high priority task is not blocked waiting to use the resource. In fact, by raising a task's priority directly to a resource's ceiling value, the immediate ceiling protocol is worse than the priority ceiling protocol because tasks not using any resources can also be prevented from executing.

The biggest advantage of this protocol, however, is that it is simple to implement, yet provides benefits similar to the priority ceiling protocol. For certain specialized situations, it may provide exactly the desired semantics. Thus, despite the drawbacks, this protocol is supported by many real-time systems, for example, it is included in the Ada Real-Time Annex [59] and it is an optional feature in POSIX threads [1, 29].

5.2 Implementing Basic Priority Inheritance

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, i.e., 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, however, is the highest value among a task's base priority and the priorities of all the tasks that it

is blocking. A task is always executed at its current active priority.

While the basic priority inheritance protocol provides a simple technique to bound the priority inversion experienced by non-independent tasks, allowing them to be scheduled, implementing this protocol is difficult. Two problems must be addressed in any complete implementation, i.e., transitivity and priority disinheritance [42].

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 5.6, 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 that these other tasks are typically 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 blocks on R_1 in Figure 5.6 (see Figure 5.7), not only must the priority of T_7 be raised, but the priority of T_8 and its position on the entry queue of R_3 must also be updated.

In the straightforward approach to implementing priority inheritance, each task must be able to determine which task is its direct blocker. This information can be used to follow a chain of blocked tasks to a task's ultimate blocker. For example, in Figure 5.6, task T_3's direct blocker is T_8, task T_8's direct blocker is T_7 and as

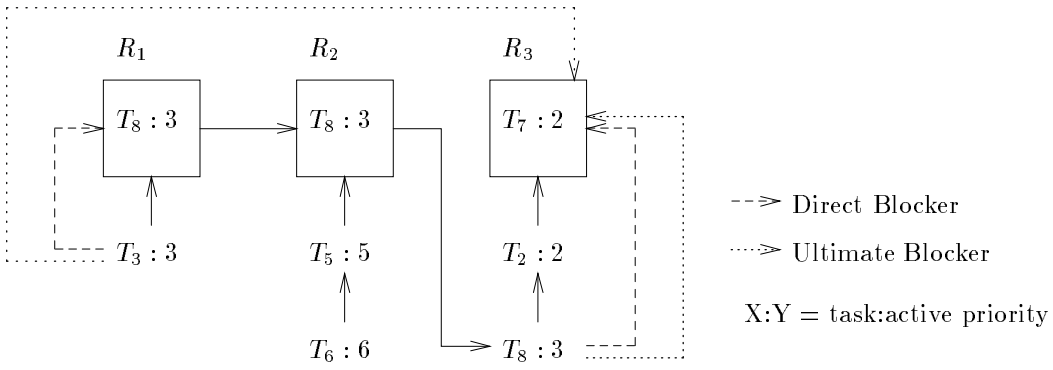

Figure 5.6: Example of Transitivity.

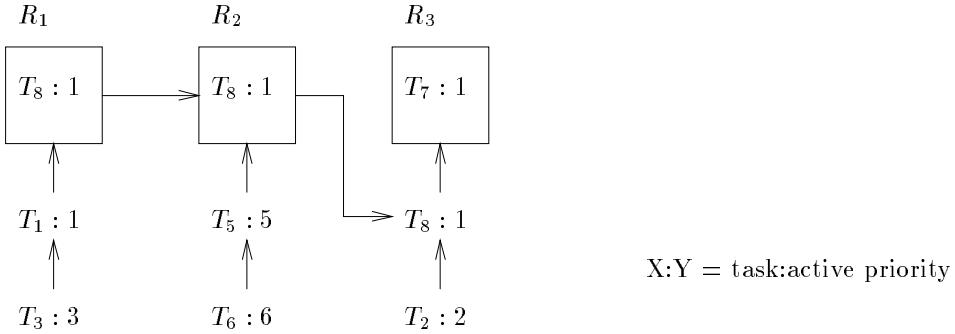

Figure 5.7: Priority queues need updating when a task blocks on a resource.

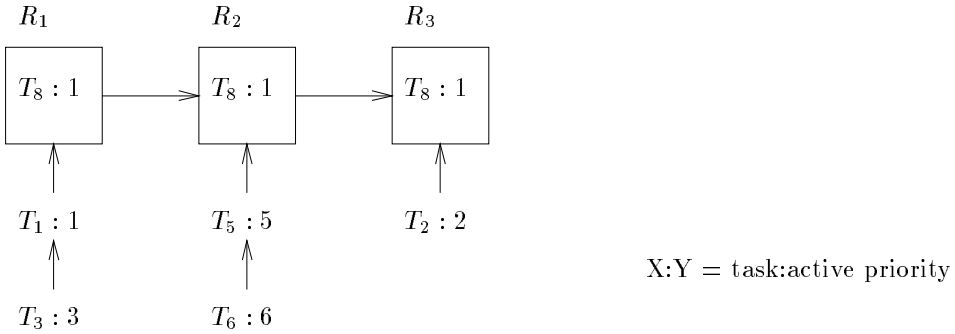

Figure 5.8: The ultimate blocker changes as tasks block on or release resources.

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 each blocked task is updated, they must also be shifted on each priority waiting queue.

Techniques that allow a task to immediately access its ultimate blocker suffer from two problems. First, the tasks on the blocking chain associated with a blocking task must be updated when its ultimate blocker changes. In order to update these tasks, the blocking chain must be traversed. For example, in Figure 5.7 suppose that T_7 releases R_3 and T_8 acquires this resource (see Figure 5.8). 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. Furthermore, a task can determine its direct blocker by simply remembering the owner of the resource it is trying to acquire. If the resource maintains its current owner and 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. However, as both the ultimate blocker and the resource associated with the ultimate blocker can change as the ultimate blocker acquires and releases resources, there is no fixed way to locate a task's ultimate blocker.

5.2.2 Priority Disinheritance

The other major problem with implementing basic priority inheritance is referred to as the *priority disinheritance problem*, i.e., determining a task's priority when it releases a resource. 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 the priority disinheritance problem.

In the first technique, each task stores its old priority value when it acquires a resource. Then when a task exits a resource, its priority is restored to this stored value. This creates a stack of values that can be used to restore a tasks priority as it exits each resource it owns. Problems exist with this idea, however.

First, the use of a stack implies that resource usages are nested, i.e., it is impossible to release resources in arbitrary order. For example, overlapping critical sections using semaphores are not 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 for this restriction 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. For example, consider the situation in Figure 5.9. If T_8 releases resource R_3 before releasing R_1 and R_2, then resetting T_8's active priority to 8 is incorrect because T_8 is still entitled to inherit priority 4 from T_4. Furthermore, this removal can also invalidate the stored priority values associated with resources higher on the stack because these stored values incorporate inheritance resulting

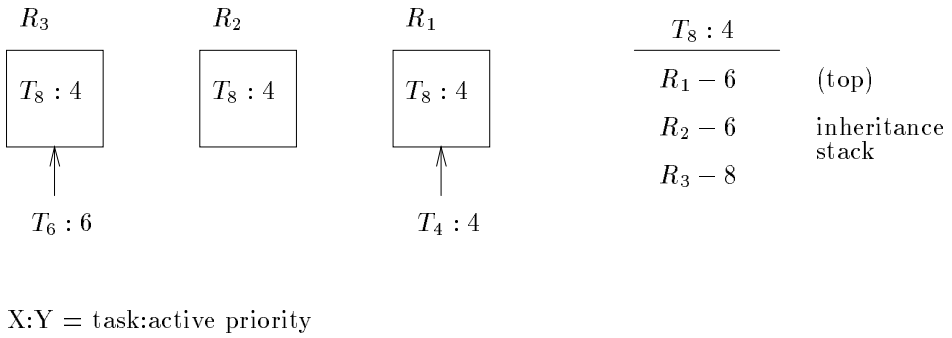

Figure 5.9: Disinheritance using a stack.

from the removed resource. Continuing the supposition that T_8 releases R_3 before releasing R_1 and R_2, the priority value stored for resource R_2 needs to change to 8 because T_8 is no longer entitled to inherit priority 6 from T_6.

Another problem with this technique is that the stored priority values can become stale, for example, if a task already owning several resources subsequently experiences priority inheritance from a resource lower down on the stack. In this situation, a task's active priority is subsequently reset to a stale value after it releases a resource. To solve this problem, if the highest priority blocked task associated with a resource changes, then 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. For example, if task T_5 with active priority 5 blocks trying to acquire resource R_3 (see Figure 5.10), then the stored priority values for R_2 and R_1 must be updated.

In general, managing this stack approach is inefficient. First, 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. Second,

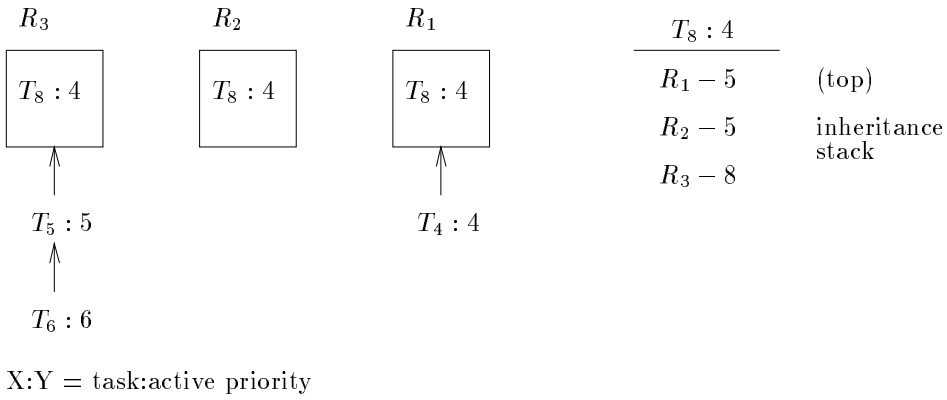

Figure 5.10: Updating stale information on a stack.

this update is dependent on the number of resources that are 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 are recursive, i.e., a task is allowed to call back into a resource it already owns. Note that 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 resource, 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 (see Figure 5.11). Then, when a task releases a resource, this resource is removed from this 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 own base priority. Some optimizations are available, however, such as a task only needs to find a

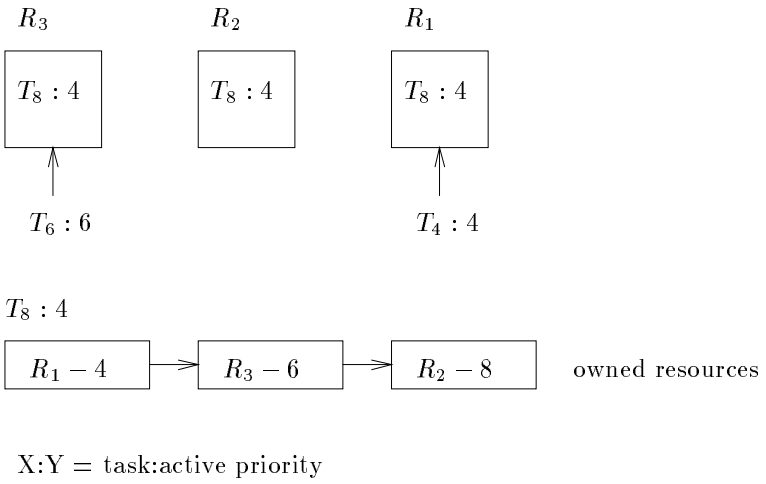

Figure 5.11: Disinheritance using a list.

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 additional overhead is incurred as the resources owned by a task are updated independently. For example, if task T_5 with active priority 5 blocks trying to acquire resource R_3 (see Figure 5.12), then only the node associated with R_3 must be updated. If the list is implemented as a priority queue, then the priority queue also needs updating. As well, only the direct blocker tasks

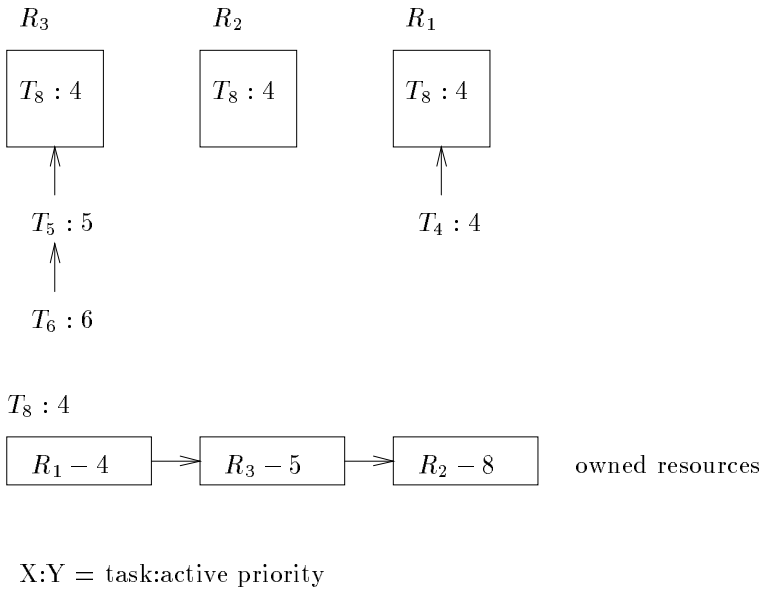

Figure 5.12: Resources in a list are independent.

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, i.e., the exit associated with the initial call that acquired the resource but not an exit associated with a recursive call into the resource.

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 [7] describe an implementation of the basic priority inheritance protocol for task rendezvous in Ada 83 [63]. The solution to transitivity is similar to the method described above, i.e., each task follows its blocking chain updating tasks as required. As the implementation only supports task rendezvous, tasks call each other directly rather than interacting through a separate resource. Limiting the implementation to task rendezvous simplifies the priority disinheritance problem because the supported resources are limited to tasks. As well, with the Ada 83 rendezvous, the entry operations are executed by the called task, i.e., the task that declared the entry. Thus, each task owns exactly one resource, i.e., itself, and a task's active priority, after disinheritance, is simply the highest priority among 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 [42]. Their first implementation provides a general solution to basic priority inheritance. With 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. For example, the priority queue for task T_8 from Figure 5.12 is given in Figure 5.13.

The problem with this technique is that 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

T_8

1	2	3	4	5	6	7	8
0	0	0	1	1	1	0	1

Figure 5.13: Example priority queue for counting technique.

represents a significant overhead, simplifications are possible if assumptions are made as to the order in which blocked tasks are scheduled. The most interesting feature of this implementation is that it allows the tasks blocked on a resource to be restarted 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 that the tasks are actually scheduled, the priority queue associated with each task determines its active priority.

Another interesting implementation is also suggested. However, this implementation assumes that tasks only block 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. This technique, however, 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, how-

ever, 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. Unfortunately, removing this blocking restriction incurs too much overhead.

Two implementations are also described by Takada and Sakamura [60]. While the details on the general implementation are sketchy, the general implementation maintains a list of resources acquired by each task, similar to the idea discussed in the previous section. The details regarding how this list is used for priority inheritance are not provided.

An interesting implementation is described, however, for situations where a task releases all the resources its owns at once. In this implementation, the disinheritance problem is solved by resetting a task's active priority back to its base priority when it releases a resource. In this case, as a task releases all these resources it owns at once, its active priority must equal its base priority at this time, as no inheritance is occurring. While this method may be appropriate for specific kinds of programs, for example, if it reasonable for a task to release all the semaphores it owns at once, it is inappropriate when the kinds of resources are arbitrary.

5.4 μC++ Implementation

This section describes an implementation of basic priority inheritance for mutex objects in μC++. In this implementation, only tasks blocked on the entry queue are eligible to donate their priority, and therefore, the entry queue is implemented as a

priority queue. Priority inheritance for mutex objects in μC++ is implemented using hooks that are invoked in the mutex entry and exit code. In order to implement priority inheritance, there are three significant times when additional work must be done, i.e., when a task acquires a mutex object, when a task releases a mutex object and when a task blocks on an entry call because it cannot acquire the mutex object.

To begin, each task maintains a priority queue, uPIQ, containing the priority inheritance information for each mutex object (resource) it owns. This information consists of the priority of the highest priority task associated with each mutex object. As in Chapter 4, only one node for each required priority level appears in a task's uPIQ. This node then references a node in a second array that contains the details for that particular priority level. In this case, however, rather than maintaining a FIFO queue of blocked tasks for a particular priority level, only a count of the number of tasks and the queue number associated with the priority level (see Section 4.3) is maintained. Figure 5.14 contains an example of the uPIQ for T_8 from Figure 5.12. In this example, the lower number in the second array is a count of the number of tasks at a particular priority level and "?" represents an unused value. Note that the example uPIQ must support (at least) the four different priorities occurring in Figure 5.12, despite the fact that task T_5 of priority 5 is not included on the queue.

No explicit reference to the highest priority task associated with each resource is required as these tasks are neither accessed nor scheduled using the uPIQ. In fact, the only relevant information is the priority and associated queue value, so a task's

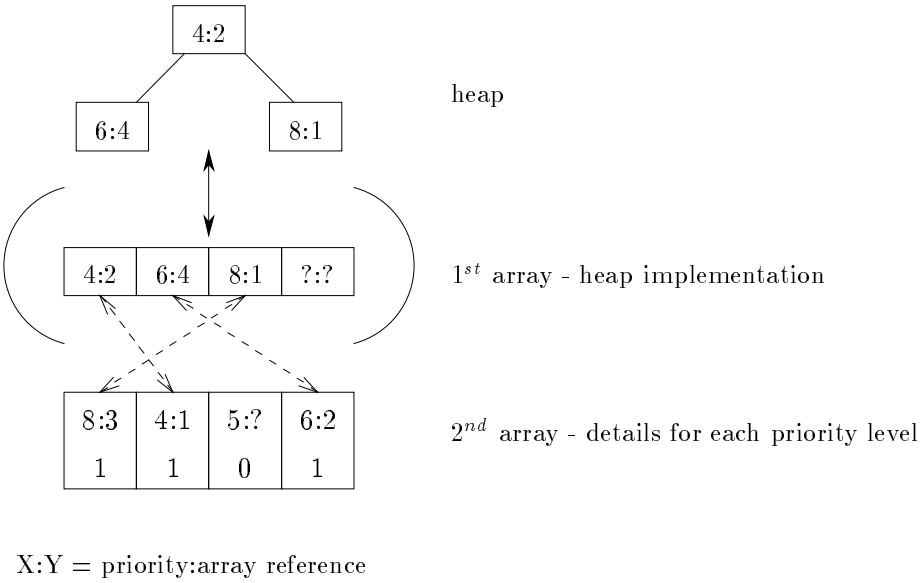

Figure 5.14: Example uPIQ.

active priority can be calculated. The count value is used to determine when nodes should be removed from the uPIQ. In this case, a zero count value corresponds to an empty FIFO queue from Chapter 4 and indicates that the associated node should be removed from the uPIQ because a task is no longer eligible to inherit that particular priority value. This priority queue solves the priority disinheritance problem because a task's active priority is the highest priority node currently on the queue. Therefore, when a task releases a mutex object, its current active priority is determined by the highest priority node remaining on the queue after its uPIQ has been updated for the released object.

5.4.1 Mutex Object Acquire

The hook uOnAcquire is invoked when a task acquires a mutex object (see line 13 in Figure 5.15). In this implementation, the task acquiring the mutex object must adjust its uPIQ (see Figure 5.16). A copy of the priority and corresponding queue information are stored in the entry queue to allow the uPIQ to be correctly updated during the remove operation, the uPIQ is updated, and then the uAfterEntry routine (see Figures 5.17 and 5.18) is invoked to perform priority inheritance from the entry queue.

The uAfterEntry routine performs priority inheritance for the mutex object based on the entry queue and the owner of the mutex object. As this routine references the entry queue and the stored priority inheritance information for the mutex object it must be executed with the entry lock acquired. This lock is already acquired, however, because uAfterEntry is invoked as part of the monitor entry code (see line 32 of Figure 5.15), i.e., at the end of the uAdd routine for the entry queue (see Figure 5.19), or from one of the hooks.

In the following discussion, the task performing the inheritance is referred to as the *updating task* and the owner of the current mutex object being referenced is referred to as the *mutex owner*. Note that it is possible for the updating task and the mutex owner to refer to the same task only when the priority inheritance is invoked by a task that acquires the mutex object without entry blocking. In this situation, the mutex owner is invoking uAfterEntry to determine the highest priority task associated with the mutex object it is acquiring, and to subsequently update its active priority if necessary. In all other situations, the updating task and the

```
1  void uSerial::uEnter( uBasePrioritySeq &entry, int mutexID ) {
2      lock.uAcquire();
3
4      uBaseTask &uCallingTask = uThisTask();      // optimization
5      if ( uMask.isSet( mutexID ) ) {                  // member acceptable ?
6          uMask.zero();                           // clear all member valid flags
7          mr = uCallingTask.uMutexRecursion;      // save previous recursive count
8          uCallingTask.uMutexRecursion = 0;       // reset recursive count
9          uMutexOwner = &uCallingTask;            // set the current mutex owner
10
11         // execute priority inheritance hook ?
12         if ( uEntryList.uExecuteHooks ) {
13             uEntryList.uOnAcquire( *uMutexOwner, this );
14         } // if
15         lock.uRelease();
16     } else if ( uMutexOwner == &uCallingTask ) {          // already hold mutex ?
17         // another recursive call at the mutex object level
18         uCallingTask.uMutexRecursion += 1;
19
20         // do not execute priority inheritance hook as task already holds mutex
21
22         lock.uRelease();
23     } else {                                        // otherwise block calling task
24         // add to end of mutex queue
25         entry.uAdd( &(uCallingTask.uMutexRef), uMutexOwner, this );
26
27         // remember which entry was called
28         uCallingTask.uCalledEntryMem = &entry;
29
30         // add mutex object to end of general entry deque
31         // perform priority inheritance in uAdd routine
32         uEntryList.uAdd( &(uCallingTask.uEntryRef), uMutexOwner, this );
33
34         // find someone else to execute; release lock on kernel stack
35         uActiveProcessorKernel->uSchedule( &lock );
36         mr = uCallingTask.uMutexRecursion;      // save previous recursive count
37         uCallingTask.uMutexRecursion = 0;       // reset recursive count
38     } // if
39     uMutexID = mutexID;                         // set active mutex member
40 } // uSerial::uEnter
```

Figure 5.15: uEnter

```
void uPriorityQ::uOnAcquire(uBaseTask &uOwner, uSerial *s) {
    // check if mutex owner's priority needs to be updated
    if( uOwner.uPIQ.uHead2() < uGetActivePriorityValue( uOwner ) ) {
        uThisCluster().uTaskSetPriority( uOwner, uOwner );
    } // if

    // remember current priority value, update task's uPIQ
    uCurrPriority = uOwner.uGetBasePriority();
    uCurrQueueNum = uOwner.uGetBaseQueue();
    uOwner.uPIQ.uAdd( uCurrPriority, uCurrQueueNum );

    // perform priority inheritance
    uAfterEntry(&uOwner, s);
} // uPriorityQ::uOnAcquire
```

Figure 5.16: uOnAcquire

mutex owner refer to different tasks.

The uAfterEntry routine begins by determining if priority inheritance is possible for the mutex object in question. In order for the mutex object to experience priority inheritance, it must have an owner, as well as at least one entry blocked task. Note that both checks are necessary because it is possible for an entry blocked task to exist, but for there to be no mutex owner during external scheduling, i.e., if the acceptor blocks because there are no outstanding calls to any acceptable member routines. As well, it is possible for a mutex owner to exist but for no tasks to be blocked on the entry queue when uAfterEntry is invoked by a new mutex owner.

The next step is to update the mutex owner based on the highest priority task blocked on the entry queue. If the mutex owner is already up to date, then no further inheritance is necessary and the operation is finished. Otherwise, the mutex owner's uPIQ is updated to reflect the highest priority task associated with the current mutex object (for subsequent disinheritance). This update may also

```
int uPriorityQ::uAfterEntry(uBaseTask *uOwner, uSerial *s) {
    // assume entry lock acquired
    int uRelPrevLock = uLockAcquired;

    // if entry queue empty (called by owner) or no owner, then no inheritance
    if (uEmpty() || uOwner == NULL ) {
        return uRelPrevLock;
    } // if

    uBaseTask &uCalling = uHead()->uGet();  // can't be NULL as not empty

    // does node need to be updated?
    if ( uCalling.uGetActivePriorityValue() < uCurrPriority ) {
        // only task with entry lock can be modifying this mutex's node
        // remove node
        uOwner->uPIQ.uRemove( uCurrPriority, uCurrQueueNum );

        // reset priority value for monitor
        uCurrPriority = uCalling.uGetActivePriorityValue();
        uCurrQueueNum = uCalling.uGetActiveQueueValue();

        // update mutex owner's uPIQ for new priority
        uOwner->uPIQ.uAdd( uCurrPriority, uCurrQueueNum ) ;

        // does inheritance occur ?
        if ( uCurrPriority < uOwner->uGetActivePriorityValue() ) {
            uSerial *uRemSerial = s->uMutexOwner->uCurrentSerial;
            // if task is blocked on entry queue, adjust and perform transitivity
            if ( uOwner->uEntryRef.uListed() ) {
                uRemSerial->lock.uAcquire();

                // if owner's mutex object changes, then it fixes its own active priority
                // Recheck if inheritance is necessary as only owner can lower its priority
                if ( uRemSerial != s->uMutexOwner->uCurrentSerial ||
                     !uOwner->uEntryRef.uListed() ||
                     uOwner->uPIQ.uHead2() >= uOwner->uGetActivePriorityValue() ) {
                    // As owner restarted, end of the blocking chain has been reached.
                    uRemSerial->lock.uRelease();
                    return uRelPrevLock;
                } // if
```

Figure 5.17: uAfterEntry, Part I

```
                // Can release previous entry lock as new entry lock is correct
                s->lock.uRelease();
                uRelPrevLock = uLockReleased;

                // proceed with transitivity
                // remove from entry queue and mutex queue
                uRemSerial->uEntryList.uRemove( &(uOwner->uEntryRef) );
                uOwner->uCalledEntryMem->uRemove( &(uOwner->uMutexRef) );

                // call cluster routine to adjust ready queue and active priority
                // as owner is not on entry queue, it can be updated based on its uPIQ
                uThisCluster().uTaskSetPriority( *uOwner, *uOwner );

                // add to mutex queue
                uOwner->uCalledEntryMem->uAdd( &(uOwner->uMutexRef),
                                uRemSerial->uMutexOwner, uRemSerial );
                // add to entry queue, automatically does transitivity
                if ( uRemSerial->uEntryList.uAdd( &(uOwner->uEntryRef),
                     uRemSerial->uMutexOwner, uRemSerial ) == uLockAcquired) {
                    // only last call does not release lock, so reacquire first entry lock
                    uThisTask().uCurrentSerial->lock.uAcquire();
                    uRemSerial->lock.uRelease();
                } // if

            } else {
                // call cluster routine to adjust ready queue and active priority
                // Note: can only raise priority to at most uCalling, otherwise updating
                // uOwner's priority can conflict with the uOwner blocking on an entry
                // queue at a particular priority level.
                // Furthermore, uCalling's priority is fixed while the entry lock of
                // where it is blocked (s->lock) is acquired, but uThisTask()'s priority
                // can change as entry lock's are released along inheritance chain.
                uThisCluster().uTaskSetPriority( *uOwner, uCalling );
            } // if

        } // if
    } // if
    return uRelPrevLock;
} // uPriorityQ::uAfterEntry
```

Figure 5.18: uAfterEntry, Part II

```
int uPriorityQ::uAdd( uBaseTaskDL *node, uBaseTask *uOwner,
                                                    uSerial *s ) {
    // check if mutex owner's priority needs to be updated
    if ( node->uGet().uPIQ.uHead2() <
                    uGetActivePriorityValue( node->uGet() ) ) {
        uThisCluster().uTaskSetPriority( node->uGet(), node->uGet() );
    } // if

    // Add node to entry queue
    ...

    // perform any priority inheritance
    return uAfterEntry( uOwner, s );
} // uPriorityQ::uAdd
```

Figure 5.19: uAdd

require the mutex owner's active priority to be increased (for current inheritance).

Changing a task's active priority, however, is not an isolated operation. In addition to changing a task's stored priority value, this change can require priority queues to be updated, as well as, causing further inheritance. Typically, the minimum requirement for μC++ is that the ready queue and all entry/mutex queues be updated. This approach is used for the implementation presented in Figures 5.17 and 5.18. As suggested in Section 2.2, updates to support the correct functioning of other application specific queues are probably best handled at the user level, rather than being included as part of the inheritance routine, e.g., condition queues.

However, a problem exists because in a concurrent (real-time) system, the mutex owner can continue to execute while its active priority is being updated. The only situation where it is possible for the mutex owner to be executing (not as the updating task) while its active priority is being updated occurs during the priority inheritance resulting from the updating task blocking on an entry queue. In all

other situations, the mutex owner is in the process of acquiring the mutex object in question and so it is also the updating task or it remains blocked while priority inheritance is invoked on its behalf (see below). In either case, the mutex owner's active priority can be safely updated.

The problem with allowing the mutex owner to execute while its priority is being updated is that the mutex owner can change mutex objects during the update. Note that regardless of the number of mutex objects a task owns, it can only be executing in one mutex object at any given time, i.e., in the task's last unfinished mutex call. For example, task T_8 (see Figure 5.8 on page 134) is currently executing in mutex object R_3 (assuming that R_1, R_2 and R_3 are mutex objects). However, the mutex object that T_8 is executing in can change if T_8 exits its current mutex object (see Figure 5.20), acquires another mutex object (see Figure 5.21), or blocks trying to acquire another mutex object (see Figure 5.22).

In order for the updating task to successfully update the mutex owner's active priority, it must be able to determine the mutex object in which the mutex owner is either currently executing or entry blocked. However, the dynamic behaviour exhibited by the mutex owner can result in the updating task chasing the mutex owner from mutex object to mutex object while trying to perform this update.

The obvious solution to this dynamic behaviour is to employ a mechanism to prevent the mutex owner from changing the mutex object in which it is currently executing while its active priority is being updated. Unfortunately, implementing such a mechanism is difficult without excessive locking, which tends to increase overhead and reduce concurrency.

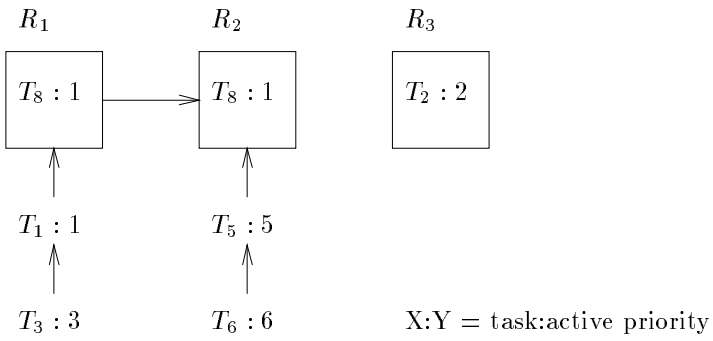

Figure 5.20: T_8's current mutex changes to R_2.

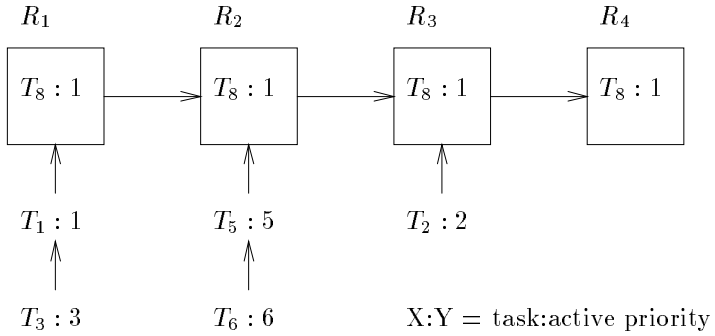

Figure 5.21: T_8's current mutex changes to R_4.

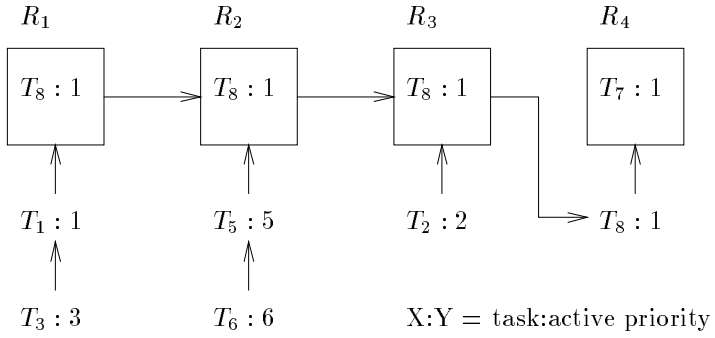

Figure 5.22: T_8's current mutex changes to R_4.

A better solution is obtained by making the following observation. If the mutex owner's current mutex object is changing, then the mutex owner is running, and therefore, represents the end of the blocking chain. In this situation, updating the mutex owner's active priority is the last step of the updating task's priority inheritance operation, i.e., no further inheritance is possible because the mutex owner is not blocked. In this situation, the responsibility for performing the last part of the inheritance operation, i.e., updating the mutex owner's active priority, is transferred to the mutex owner. By forcing the mutex owner to adjust its own active priority as its current mutex object changes, the updating task no longer has to deal with the potentially dynamic behaviour of the mutex owner. The only requirement is for the updating task to detect this volatility with respect to the mutex owner's current mutex object, so it knows when it has reached the end of its inheritance chain.

The other advantage of this approach is that having the mutex owner update its own active priority is much simpler and less expensive than if it is done by the updating task. This advantage arises because the mutex owner is running, and hence, not blocked on the entry/mutex queues or ready queue so these queues require no adjustment. Furthermore, the mutex owner's current mutex object is fixed while it is updating its own priority.

This solution can be broken down into two parts. First, if the updating task recognizes that the mutex owner's current mutex object is changing, then it has reached the end of the inheritance chain and can stop. Second, if the mutex owner changes mutex objects, it must complete the last part of any inheritance by updat-

ing its own active priority.

Specifically, the fact that the mutex owner's current mutex object is changing is not the actual problem. The situations that present the real problems are those that involve the entry/mutex queue because the entry lock is required for the update, but the required entry lock can change as the mutex owner moves. This problem results in two scenarios, i.e., an entry blocked mutex owner is scheduled by a task blocking in or exiting a mutex object or a running mutex owner executes an entry call that blocks on another mutex object. Unfortunately, as tasks are executing concurrently, it is impossible to recognize these exact situations. The updating task can simply determine the mutex owner's current mutex object and whether the mutex owner is currently entry blocked at a particular instant and then decide how to proceed.

If the updating task acquires the entry lock for the mutex object where it believes the mutex owner is currently executing and determines that the mutex owner is not entry blocked, then the updating task is simply required to update the mutex owner's active priority. The problem that occurs is that the mutex owner can make an entry call to another mutex object that blocks while this is occurring. However, it is now the case that the mutex owner must recalculate its own active priority before it is queued as a result of a blocking entry call. As its active priority is calculated using the uPIQ (note, the mutex owner's uPIQ is always updated *before* its active priority is updated), this guarantees that it is executing at its correct priority *before* it is placed on the entry queue. Therefore, no further updating is required.

However, the updating task cannot increase the mutex owner's active priority based on the mutex owner's uPIQ; it can only increase the mutex owner's priority to at most the value it used to update the mutex owner's uPIQ. Without this restriction, it is possible for the mutex owner to be blocked on an entry queue at a priority different from its active priority because the mutex owner is only guaranteed to have seen the changes made to its uPIQ by the current updating task when it entry blocks. Thus, if the updating task increases the mutex owner's active priority to an even higher value (based on the latest information in the mutex owner's uPIQ), then the updating task could change the mutex owner's active priority but not its position on the entry queue.

For example, in Figure 5.23 task T_5 entry blocks trying to acquire R_2. As inheritance is required, T_5 begins by updating T_8's uPIQ and notices that T_8 is not currently entry blocked. However, before T_5 can update T_8's active priority, two things happen (see Figure 5.24). First, T_8 entry blocks on R_4 and updates its own active priority to 5 based on its current uPIQ. Second, T_4 entry blocks on R_1, updates T_8's uPIQ and detects that T_8 is entry blocked. Now, T_5 cannot update T_8 based on T_8's uPIQ because this would cause T_8's active priority to be increased without updating its position on the entry queue. As well, T_8's active priority and position on the entry queue will eventually be updated by T_4.

Note that a call by the updating task to increase the mutex owner's active priority has no effect if the mutex owner's priority is already higher than this update value. Furthermore, if the mutex owner entry blocks, then it has already increased its own active priority as discussed above. If, however, the mutex owner's active

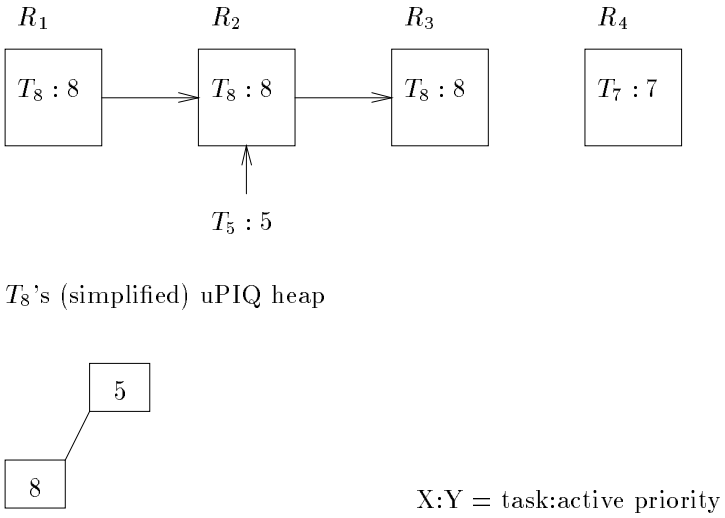

Figure 5.23: T_5 updates T_8's uPIQ but not its active priority.

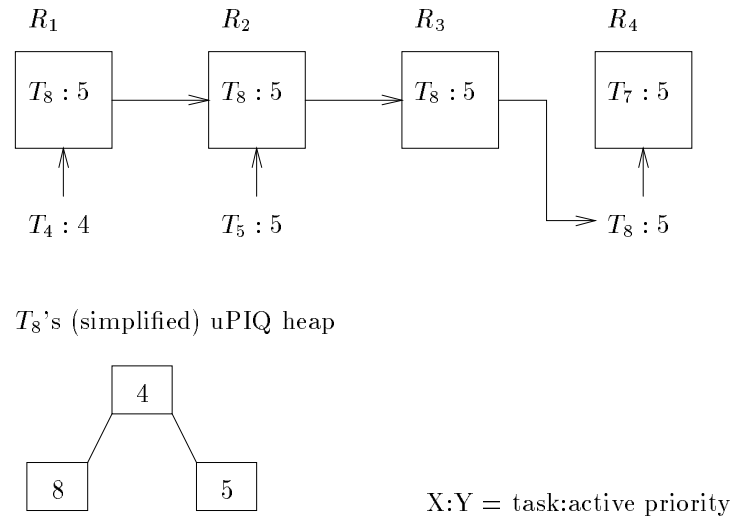

Figure 5.24: T_5 cannot use T_8's uPIQ to update its active priority.

priority does need to be increased based on its uPIQ, then the task responsible for this change adjusts the mutex owner's active priority and position on the entry queue as required. The current updating task is only responsible for adjusting the mutex owner's active priority based on its own changes to the mutex owner's uPIQ. As well, if the mutex owner's entry call does not block, then no problem arises as the mutex owner is not blocked on an entry/mutex queue and so only its active priority needs to be updated, which is exactly the behaviour that occurs.

On the other hand, if the updating task detects that the mutex owner is entry blocked, then both the mutex owner's active priority and its position on the entry queue must be updated. Therefore, the updating task begins by acquiring the appropriate entry lock. However, by the time the updating task successfully acquires the entry lock, the entry blocked mutex owner may have been scheduled and be running, or moved to another mutex object and blocked, or returned from the current mutex object. Thus, when a task, in this case the mutex owner, actually acquires a mutex object it must recalculate its own active priority. In this situation, the updating task only needs to detect that either the mutex owner is no longer entry blocked or that its current mutex object has changed. Furthermore, as the updating task has the entry lock for what it believes is the mutex owner's last mutex object, any calls by the mutex owner to this mutex object are prevented from continuing as all entry calls begin by acquiring the entry lock. Therefore, it is impossible for the mutex owner to still be in the process of blocking on this mutex object, i.e, either the mutex owner has finished blocking or it has already been scheduled. Thus, if the updating task detects the mutex owner is no longer

entry blocked on the mutex object it believes the mutex owner is blocked on, then the mutex owner has updated its own active priority and the inheritance operation is complete.

Finally, if the mutex owner is still blocked on the appropriate mutex object, then the mutex owner remains blocked as it cannot be scheduled while the updating task has the entry lock. In this situation, the updating task can update both the entry queue and the active priority of the mutex owner without difficulty as the state of the mutex owner is fixed. In fact, this situation is the only one in which the mutex owner has not been detected as running. Therefore, the priority inheritance operation must advance to the next element in the inheritance chain. This advancement occurs with the call to uAdd when the mutex owner is being adjusted on the entry queue. The call to uAdd calls uAfterEntry for the new mutex object resulting in transitivity (see Figure 5.19).

In order to maximize concurrency, however, it is possible to proceed with inheritance without maintaining the entry lock of every previous mutex object in the chain. In fact, the ultimate blocker is the only task in an inheritance chain that can execute. So, every other mutex owner in the chain is essentially blocked waiting for the ultimate blocker to release them. However, an interesting feature of the entry lock is that it also prevents tasks from releasing mutex objects. Thus, acquiring a particular entry lock prevents tasks from backing up past that mutex object in the inheritance chain and effectively fixes the back portion of an inheritance chain. Therefore, entry locks associated with mutex objects earlier in the chain can be released without causing a problem. The advantage of this approach is that it al-

lows tasks to still block on the entry queues of those mutex objects during priority inheritance. In order to maintain this locking as a task proceeds along the inheritance chain, the next entry lock must be acquired before the current entry lock is released.

The entry lock of the mutex object associated with the updating task, i.e., the first entry lock acquired, is a special case. While releasing this lock is fine because the updating task cannot be scheduled as long an entry lock further down the chain is acquired, this entry lock must be reacquired before the last entry lock is released. Reacquiring 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 updating task to be scheduled before it can block. Reacquiring 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 updating task 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 chain. The potential problem that occurs is that a mutex owner's uPIQ can be updated using one value and its active priority can be updated using a higher value. This discrepancy can lead to the mutex owner being blocked on the entry queue at a priority value different from its active priority.

As mutex objects in μC++ allow recursive calls, the uOnAcquire hook is only invoked when a task initially acquires the mutex object (see line 13 in Figure 5.15).

A check for priority inheritance does not need to occur on subsequent entries (see line 20 in Figure 5.15) because the priority information for the highest priority task associated with the resource is already on the task's uPIQ. Furthermore, even if a task's active priority on re-entry is higher than the priority information stored for the mutex object on the task's uPIQ because of further priority inheritance from other resources, this priority information still does not need to be updated because the priority information for a resource 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 the mutex.

5.4.2 Mutex Object Release

The hook uOnRelease is invoked when a task releases a mutex object. In this implementation, the task must remove the inheritance information associated with the mutex object it is releasing from its uPIQ and calculate its new active priority (see Figure 5.25). If the task's priority inheritance priority queue is empty after this inheritance information is removed, then the active priority of the task is set to its base priority. Despite the fact μC++ mutex objects are recursive, this hook is only invoked when a task exiting a mutex finally relinquishes control of the mutex (see lines 14, 23 and 34 in Figure 5.26). As a task's uPIQ is not updated on re-entry into a resource is already owns, it does not need to be reset on an interim exit (see line 5 in Figure 5.26).

In the situation where a task releasing a mutex wakes up the next task to execute inside this mutex, the uOnAcquire routine is invoked by the releasing task

```
void uPriorityQ::uOnRelease(uBaseTask &uOldOwner, uSerial *s) {
    // update task's uPIQ, reset stored values
    uOldOwner.uPIQ.uRemove( uCurrPriority, uCurrQueueNum );
    uCurrPriority = -1;
    uCurrQueueNum = -1;

    // reset active priority if necessary
    // only case where priority can decrease
    if ( uOldOwner.uPIQ.uEmpty() ||
         uOldOwner.uPIQ.uHead2() > uGetActivePriorityValue( uOldOwner ) ) {
       uThisCluster().uTaskSetPriority( uOldOwner, uOldOwner );
    } // if
} // uPriorityQ::uOnRelease
```

Figure 5.25: uOnRelease

on behalf of this new owner (see lines 24 and 35 in Figure 5.26). It is impossible to allow the new mutex owner to execute this routine *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, this task can experience uncontrolled priority inversion while it waits to be scheduled so it can updates its active priority.

5.4.3 Entry Blocking on a Mutex Object

When a task blocks on the entry queue of a mutex object, it may need to raise the priority of the current owner of the mutex object. Rather than providing an explicit hook for this circumstance, it is possible to add any additional code to the end of the routine to add a task to the entry queue. In this implementation, the routine uAfterEntry (see Figure 5.17) is invoked after a calling task has been added to the entry queue (see Figure 5.19). This routine performs the required priority inheritance as described above.

```
1  void uSerial::uLeave( ) {
2      uBaseTask &uCallingTask = uThisTask();      // optimization
3      if ( uCallingTask.uMutexRecursion != 0 ) {  // recursively hold mutex ?
4          uCallingTask.uMutexRecursion -= 1;
5          // do not execute priority inheritance hook as task still holds mutex
6          return;
7      } // if
8
9      if ( uAcceptSignalled.uEmpty() ) {    // no tasks waiting re-entry to mutex object ?
10         lock.uAcquire();
11         if ( uEntryList.uEmpty() ) {              // no tasks waiting entry to mutex object ?
12             uMask.one();                              // accept all members
13             uMutexOwner = (uBaseTask *)0;         // reset no task in mutex object
14             if ( uEntryList.uExecuteHooks ) {     // execute priority inheritance hook ?
15                 uEntryList.uOnRelease( uCallingTask, this );
16             } // if
17             lock.uRelease();
18         } else {                                  // tasks waiting entry to mutex object
19             // next task to gain control of the mutex object
20             uMutexOwner = &(uEntryList.uDrop()->uGet());
21             // also remove task from mutex queue
22             uMutexOwner->uCalledEntryMem->uRemove(&(uMutexOwner->uMutexRef));
23             if ( uEntryList.uExecuteHooks ) {     // execute priority inheritance hook ?
24                 uEntryList.uOnRelease( uCallingTask, this );
25                 uEntryList.uOnAcquire( *uMutexOwner, this );
26             } // if
27             lock.uRelease();
28             uMutexOwner->uWake(); // wake up next task to use this mutex object
29         } // if
30     } else {                                      // tasks waiting re-entry to mutex object
31         // next task to gain control of the mutex object
32         if ( uEntryList.uExecuteHooks ) {     // execute priority inheritance hook ?
33             lock.uAcquire();                  // entry lock prevents inversion during transfer
34             uMutexOwner = &(uAcceptSignalled.uDrop()->uGet());
35             uEntryList.uOnRelease( uCallingTask, this );
36             uEntryList.uOnAcquire( *uMutexOwner, this );
37             uMutexOwner->uWake();
38             lock.uRelease();
39         } else {
40             uMutexOwner = &(uAcceptSignalled.uDrop()->uGet());
41             uMutexOwner->uWake();
42         } // if
43     } // if
44     uCallingTask.uMutexRecursion = mr;      // restore previous recursive count
45 } // uSerial::uLeave
```

Figure 5.26: uLeave

5.4.4 Blocking Inside a Mutex Object

The hooks described above are also invoked when a task acquires or releases a mutex object resulting from blocking inside a mutex or being scheduled from an internal mutex queue, for example, blocking on and restarting from an accept statement. The problem is that normally (i.e., non-real-time) when a task from an internal mutex queue is scheduled, the entry lock is not acquired.

When selecting a task to run from the entry queue, the entry lock is acquired and then control of the mutex is transferred to that task. While holding the entry lock, the releasing task can adjust its active priority and the active priority of the new owner, which is still blocked (see lines 23-24 in Figure 5.26).

However, when selecting a task from an internal mutex queue, the entry lock does not normally need to be acquired. In this situation, it is possible for a task to be interrupted after it executes the uOnRelease hook but before it restarts the blocked task. The potential problem in this window is that a task's active priority may have been lowered, but it has not restarted the next mutex owner. Thus, the next mutex owner can experience uncontrolled priority inversion as it cannot be restarted until the old owner is scheduled again. Unfortunately, the old owner is no longer experiencing priority inheritance from the mutex object it is releasing. Furthermore, as the releasing task may be attempting to block on an internal mutex queue, it is impossible to wait until the new task is restarted before the priority of the releasing task is lowered because in this situation the releasing task atomically blocks and wakes up the new owner. Thus, this task does not have the opportunity to lower its priority after the new owner is restarted.

This situation is resolved by forcing the entry lock to be acquired whenever the mutex hooks are invoked. As well, whenever a task that is releasing a mutex also restarts the next mutex owner, the entry lock remains acquired while the task executes both the uOnRelease hook for itself and the uOnAcquire hook on behalf of the new mutex owner (see lines 32-37 in Figure 5.26). In fact, the entry lock is not released until the new mutex owner is restarted. Thus, the releasing task is uninterruptable during this time and the priority inversion window is eliminated.

5.4.5 Analysis

The algorithm described above is actually equivalent to the general case version of the basic priority inheritance algorithm described by Moylan, Betz and Middleton [42]. In this case, as the entry queues are prioritized, only the task with the highest active priority needs to be counted. Other than that, a priority queue is maintained for each task. This priority queue 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 the dynamic priority scheduling described in chapter 4. 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.

Priority inheritance for μC++ objects is implemented using the three routines discussed above, i.e., uOnAcquire, uOnRelease and uAfterEntry. As the priority

inheritance queue (uPIQ), and the entry and the mutex queues are implemented based on the discussion of priority queues in Chapter 4, all the associated operations are O(1). As well, adjusting a task's priority is also an O(1) operation. Therefore, the complexity of uAfterEntry 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 calls uAfterEntry, because it is executed by the mutex owner, so the blocking chain has size at most one.

5.5 Summary

When tasks share resources, access to these resources are protected using critical sections. However, serializing access to shared resources can result in a situation referred to as priority inversion. In general, it may be impossible to eliminate priority inversion from a system, so it is important to bound the length of this inversion. Priority inheritance protocols are a method to bound this inversion.

While several priority inheritance protocols exist, a common protocol is the basic priority inheritance protocol. This protocol provides a reasonable solution to uncontrolled priority inversion and forms the base for more sophisticated protocols. More sophisticated algorithms, like the priority ceiling protocol, have the advantage of preventing multiple blocking and deadlock, but require additional static information, incur more runtime overhead, and tend to be overly restrictive. Therefore, the basic priority inheritance protocol is a good choice for dynamic systems or where the potential for excessive multiple blocking is small. However, in situations where static analysis is possible and multiple blocking is a problem, it is reasonable to

pay the costs of a more sophisticated algorithm.

The most difficult parts of implementing basic priority inheritance are dealing with transitivity and priority disinheritance. For transitivity, the straightforward solution is most appropriate. This solution involves following a task's blocking chain when it cannot acquire a resource, and updating the priority of tasks on this blocking chain as required. This technique has the added advantage of eliminating stale information associated with blocked tasks.

With priority disinheritance, however, a variety of solutions are possible. Unfortunately, the efficient solutions can only be applied in cases where simplifying assumptions are appropriate. In the general solution, a list of tasks acquired by a resource is maintained and then a task can determine its new active priority by finding the highest priority among its own base priority and the active priority of tasks blocked on these resources.

The implementation of priority inheritance for mutex objects in μC++ is similar to an algorithm described by Moylan, Betz and Middleton [42]. The basic idea is to maintain a priority queue in each task containing information about the highest priority task associated with each mutex object the task owns. Thus, a task's active priority is the highest priority value on this priority queue.

The problem in the μC++ implementation is that extra blocking is required when the next task acquiring the mutex is selected from an internal mutex queue. In this case, a window exists in which uncontrolled priority inversion can occur. This situation is dealt with by making the mutex object uninterruptable by acquiring the entry lock while control is passed to the next task. The problem with this approach,

however, is that the low overhead usually associated with internal scheduling is lost (but only for real-time mutex types). As well, concurrency is reduced because tasks are also prevented from adding themselves to the entry queue during this window. Unfortunately, as the priority inheritance data structures need to be modified by tasks blocking on the monitor, it is natural to associate them with the entry data structures, and hence, require the use of the entry lock for mutual exclusion.

Chapter 6

Conclusion

The goal of this thesis is to discuss the creation of a predictable real-time system that is also extensible and flexible. The creation of an extensible real-time system is important to both encourage the use of new theory and to discourage the use of ad-hoc systems.

Ad-hoc systems exist because current commercial real-time systems are neither flexible nor allow access to the data structures used by the system. Allowing access to these data structures provides two important advantages to the real-time programmer. First, the system is extensible, allowing new ideas and theory to be tested and incorporated into the system. Second, it allows the programmer to fine tune the system specifically for an application. This point is important because the kind of real-time application can have a significant impact on the types of data structures that are appropriate.

For example, allowing the user to control the priorities of the tasks in a system without allowing access to the ready queue data structure would eliminate the

possibility of using the scheduling technique described in Chapter 4.

The approach taken in μC++ is that rather than limit the user to a fixed set of alternatives, the user is given the ability to plug in additional functionality. This approach allows the user a richer set of possibilities. As well, standard functionality is provided by including a predefined library of data structures with the system. This approach is demonstrated in Chapter 5 by creating a priority inheritance mechanism based on the dynamic scheduling technique described in Chapter 4.

However, there are several drawbacks to this approach. First, the user is responsible for guaranteeing that any functionality that is 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 predictable 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 user code. Thus, it is reasonable to allow the user more control over the system code.

6.1 Future Work

6.1.1 μC++

There are a number of extensions that would increase the real-time functionality of μC++.

Adding timeouts to accept statements is just the first step. Other operations

would benefit from a timeout mechanism, including entry calls and calls to block on a condition variable. In addition to adding timeout mechanisms to specific operations, a timeout mechanism that can be applied to an arbitrary block of code would also be useful. In order to implement such a mechanism, the exception handling facilities in μC++ [41] must also be modified for a real-time environment. These changes would require that the costs of exception handling have a fixed worst-case execution time.

Including more scheduling and inheritance algorithms [15, 16, 44] with μC++ would also be useful for users, as well as, to test the extensibility of the mechanism. Additional inheritance algorithms include priority ceiling and dynamic priority ceiling. Furthermore, inheritance can also be expanded to include spinlocks and semaphores [65].

In addition to further extending the real-time features of mutex objects and other language constructs in μC++, it is also necessary to calculate the overheads and costs associated with these mechanisms. Moreover, it would also be interesting to determine the impact of allowing users to access certain system data structures with respect to debugging and to explore the learning curve involved in actually implementing a new inheritance protocol.

As well, another interesting area is to extend KDB, μC++'s concurrent debugger, to handle real-time programs [49]. Typical debugging methods for real-time programs include deadline monitoring to determine if any task timing constraints are being violated and time distortion to maintain correct time values to compensate for the overhead and interference of debugging [43].

Finally, it is important to separate the scheduler and dispatcher in μC++. Typically, the job of the scheduler is to determine if new tasks can be added to the schedule and to create a feasible schedule as tasks are added and removed. The dispatcher, on the other hand, is required to execute this schedule by selecting which task is entitled to run at any given time. At the moment, the scheduler and the dispatcher are together, resulting in dispatching delays as the schedule is modified. An interesting approach is to treat the scheduler as a periodic task and to allow the schedule only to be modified during its execution time. This allows the costs of scheduling to be worked into the schedule and enhances the predictability of the system. The drawback is that new tasks would suffer from slower response times.

6.1.2 Scheduling

Currently many real-time systems are priority-based. However, it is difficult to encapsulate all the relevant scheduling information into a single priority value. Instead, a promising area of real-time scheduling is time-based scheduling using heuristic and planning approaches from AI [28, 46, 52, 56, 66]. These algorithms are generally dynamic and try to guarantee that all the resources and interactions that a task requires are available as per the task's request. These types of approaches provide end-to-end scheduling that is important for use with soft real-time applications such as multimedia and Internet video.

6.1.3 Lock Free Systems

Another interesting area to explore is the notion of a lock free kernel. In concurrent systems, locks are generally used to prevent multiple access to queues that govern entry to critical sections. However, the use of instructions that allow tasks to atomically add and remove themselves from queues eliminates the need for locking [23, 39, 48]. Eliminating locking from the system tends to increase concurrency and reduce response time. Both important goals for a real-time system.

Finally, the notion of an interruptible critical section [27] can also be used to reduce the need for locking and queuing in the system. In this case, priority inversion is eliminated because a higher priority task can always preempt the execution of a lower priority task even if the lower priority task is in a critical section.

Bibliography

[1] *Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language].* Information technology—Portable Operating System Interface (POSIX). IEEE Computer Society, 345 E. 47th St, New York, NY 10017, USA, 1996.

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

[3] Audsley, N. C. Deadline monotonic scheduling. Tech. rep., University of York, 1990. YCS 146.

[4] Audsley, N. C., Burns, A., Richardson, M. F., and Wellings, A. J. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software* (Atalanta, May 1991).

[5] Baker, T. P. A stack-based resource allocation policy for realtime processes. In *Proceedings of the Real-Time Systems Symposium - 1990* (Lake Buena Vista,

Florida, USA, Dec. 1990), I. C. S. Press, Ed., IEEE Computer Society Press, pp. 191–200.

[6] Baker, T. P. Stack-based scheduling of realtime processes. *Journal of Real-Time Systems 3*, 1 (Mar. 1991), 67–99.

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

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

[9] Buhr, P. A. Understanding control flow: with concurrent programming using μC++. Textbook in preparation. Available via ftp from plg.uwaterloo.ca in pub/uSystem/uC++book.ps.gz, 1999.

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

[11] Buhr, P. A., and Stroobosscher, R. A. μC++ annotated reference manual, version 4.7. Tech. rep., Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, Feb. 1999. Available via ftp from plg.uwaterloo.ca in pub/uSystem/uC++.ps.gz.

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

[13] Burns, J. E. Mutual exclusion with linear waiting using binary shared variables. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory) 10*, 2 (1978), 42–47.

[14] Cardelli, L. A language with distributed scope. In *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: San Francisco, California, January 22–25, 1995* (New York, NY, USA, 1995), ACM, Ed., ACM Press, pp. 286–297.

[15] Chen, C.-M., and Tripathi, S. K. Multiprocessor priority ceiling based protocols. Technical Report CS-TR-3252, University of Maryland, College Park, Apr. 7, 1994.

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

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

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

[19] Gehani, N., and Ramamritham, K. Real-time Concurrent C: A language for programming dynamic real-time systems. *Journal of Real-Time Systems 3*, 4 (Dec. 1991), 377–405.

[20] Gehani, N., and Roome, W. *The Concurrent C Programming Language.* Silicon Press, 1989.

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

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

[23] Herlihy, M. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems 13*, 1 (Jan. 1991), 124–149.

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

[25] Holt, R. C. *Turing Reference Manual*, third ed. Holt Software Associates Inc., 1992.

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

[27] Johnson, T. Interruptible critical sections for real-time systems. Technical Report 93–017, University of Florida, 1993.

[28] JOSEPH, M., Ed. *Real-time Systems, Specifications, Verification and Analysis.* Prentice Hall, 1996.

[29] KLEIMAN, S., SHAH, D., AND SMAALDERS, B. *Programming With Threads.* SunSoft Press, Mountainview, CA, USA, 1995.

[30] LEHOCZKY, J., SHA, L., AND DING, Y. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proc. IEEE 10th Real-Time Systems Symp.* (Dec. 1989), pp. 166–171.

[31] LEHOCZKY, J. P., AND RAMOS-THUEL, S. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proceedings of the Real-Time Systems Symposium - 1992* (Phoenix, Arizona, USA, Dec. 1992), R. Werner, Ed., IEEE Computer Society Press, pp. 110–124.

[32] LEHOCZKY, J. P., SHA, L., AND STROSNIDER, J. K. Enhanced aperiodic responsiveness in hard real-time environments. In *Proc. IEEE Real-Time Systems Symposium* (Dec. 1987), pp. 261–270.

[33] LEHOCZKY, J. P., AND THUEL, S. R. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *Proceedings of the Real-Time Systems Symposium - 1994* (Dec. 1994), IEEE Computer Society Press, pp. 22–33.

[34] LEUNG, J. Y. T., AND WHITEHEAD, J. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation North Holland 2* (1982), 237–250.

[35] LIM, P. E. Real-time in a concurrent, object-oriented programming environment. Master's thesis, University of Waterloo, 1996.

[36] LIU, C. L., AND LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM 20*, 1 (Jan. 1973), 46–61.

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

[38] MERCER, C. W. An Introduction to Real-Time Operating Systems: Scheduling Theory, 1992. Unpublished manuscript. Available via http from http://www.cs.cmu.edu/afs/cs/project/rtmach/public/papers/sur1.review.ps.

[39] MICHAEL, M. M., AND SCOTT, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. Technical Report TR600, University of Rochester, Computer Science Department, December 1995. Available via ftp from ftp://ftp.cs.rochester.edu/pub/papers/systems/95.tr600.Nonblocking_and_blocking_concurrent_queue_algorithms.ps.gz.

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

[41] MOK, W. Y. R. Concurrent abnormal event handling mechanisms. Master's thesis, University of Waterloo, 1997.

[42] MOYLAN, P., BETZ, R., AND MIDDLETON, R. The priority disinheritance problem. Tech. Rep. EE9345, The University of Newcastle, 1993.

[43] MUELLER, F., AND WHALLEY, D. On debugging real-time applications. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems* (June 1994).

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

[45] RAJKUMAR, R., SHA, L., LEHOCZKY, J., AND RAMAMRITHAM, K. *Principles of Real-Time Systems.* Prentice-Hall, 1994, ch. An optimal priority inheritance protocol for real-time synchronization, pp. 249–271. Sang Son, Ed.

[46] RAMAMRITHAM, K., SHIAH, P.-F., AND STANKOVIC, J. A. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems 1*, 2 (April 1990), 184–195.

[47] RAMAMRITHAM, K., AND STANKOVIC, J. Scheduling algorithms and operating systems support for real-time systems. In *Proceedings of IEEE* (Jan 1994), pp. 55–67. Invited paper.

[48] RAMAMURTHY, S., MOIR, M., AND ANDERSON, J. H. Real-time object sharing with minimal system support (extended abstract). In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)* (New York, USA, May 1996), ACM, pp. 233–242.

[49] SCHUSTER, O. Replay of shared memory programs. Master's thesis, University of Mannheim, 1999.

[50] Sha, L., Rajkumar, R., and Lehoczky, J. P. Priority inheritance protocols. Tech. Rep. CMU-CS-87-181, Carnegie Mellon, 1987.

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

[52] Shen, C. *An Integrated Approach to Dynamic Task and Resource Management in Multiprocessor Real-Time Systems.* PhD thesis, University of Massachusetts, Amherst, Computer Science, 1992.

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

[54] Spuri, M., and Buttazzo, G. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the Real-Time Systems Symposium - 1994* (Dec. 1994), IEEE Computer Society Press, pp. 2–11.

[55] Spuri, M., and Buttazzo, G. Scheduling aperiodic tasks in dynamic priority systems. *Journal of Real-Time Systems 10*, 2 (March 1996), 179–210.

[56] Stankovic, J. A., and Ramamritham, K. The spring kernel: a new paradigm for real-time systems. *IEEE Software 8*, 3 (May 1991), 62–72.

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

[58] Strosnider, J. K., Lehoczky, J. P., and Sha, L. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers 44*, 1 (January 1995).

[59] Taft, S. T., and Duff, R. A. *Ada 95 reference manual: language and standard libraries*, vol. 1246 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1997. International standard ISO/IEC 8652:1995(E). Available via http from http://www.adaic.org/standards/95lrm/LRMps/rm.ps.

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

[61] Thomadakis, M., and Liu, J.-C. On the efficient scheduling of non-periodic tasks in hard real-time systems. Technical Report TR99-012, Texas A&M University, May 10, 1999.

[62] Tia, T. S. *Utilizing Slack Time For Aperiodic and Sporadic Requests Scheduling in Real-Time Systems*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[63] United States. Dept. of Defense, and American National Standards Institute. *Reference manual for the Ada programming language: ANSI/MIL-STD-1815A-1983*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1983. Approved February 17, 1983.

[64] VAN EMDE BOAS, P., KAAS, R., AND ZIJLSTRA, E. Design and implementation of an efficient priority queue. *Mathematical Systems Theory 10* (1977), 99–127.

[65] WANG, C.-D., TAKADA, H., AND SAKAMURA, K. Priority inheritance spin locks for multiprocessor real-time systems. In *Proceedings of the 1996 International Symposium on Parallel Architecture, Algorithms and Networks (ISPAN'96)* (June 1996), pp. 70–76.

[66] WANG, F. *Issues related to dynamic scheduling in real-time systems.* PhD thesis, University of Massachusetts, 1993.

[67] WILLIAMS, J. W. J. Algorithm 232: Heapsort. *Communications of the ACM 7* (1964), 347–348.