Chapter 7

Resource Patterns

The following patterns are presented in this chapter.

- Critical Section Pattern: Uses resources run-to-completion
- Priority Inheritance Pattern: Limits priority inversion
- Highest Locker Pattern: Limits priority inversion
- Priority Ceiling Pattern: Limits priority inversion and prevents deadlock
- Simultaneous Locking Pattern: Prevents deadlock
- Ordered Locking Pattern: Prevents deadlock

7.1 Introduction

One of the distinguishing characteristics of real-time and embedded systems is the concern over management of finite resources. This chapter provides a number of patterns to help organize, manage, use, and share such resources. There is some overlap of concerns with the patterns in this and other chapters. For example, the Smart Pointer Pattern provides a robust access to resources of a certain type: those that are dynamically allocated. However, that pattern has already been discussed in Chapter 6. Similarly, the synchronization of concurrent threads may be thought of as resource management, but it is dealt with using the Rendezvous Pattern from Chapter 5. This chapter focuses on patterns that deal with the sharing and management of resources themselves and not the memory they use. To this end, we'll examine a number of ways to manage resources among different, possibly concurrent, clients.

A *resource,* as used here, is a thing (an object) that provides services to clients that have finite properties or characteristics. This definition is consistent with the so-called Real-Time UML Profile [1], where a resource is defined as follows.

> An element that has resource services whose effectiveness is represented by one or more Quality of Service (QoS) characteristics.

The QoS properties are the quantitative properties of the resource, such as its capacity, execution speed, reliability, and so on. In real-time and embedded systems, it is this quantifiable finiteness that must be managed. For instance, it is common for a resource to provide services in an atomic fashion; this means that the client somehow "locks" the resource while it needs it, preventing other clients from accessing that resource until the original client is done. This accomplishes the more general purpose of *serialization* of resource usage, crucial to the correct operation in systems with concurrent threads. This is often accomplished with a mutex semaphore (see the Guarded Call Pattern in Chapter 5) or may be done by serializing the requests themselves (see the Message Queuing Pattern, also in Chapter 5).

The management of resources with potentially many clients is one of the more thorny aspects of system design, and a number of patterns have evolved or been designed over time to deal specifically with just that.

The first few patterns (Priority Inheritance, Highest Locker, Priority Ceiling) address the schedulability of resources in a priority-based preemptive multitasking environment, which can be a major concern for real-time systems design. In static priority scheduling approaches (see, for example, the Static Priority Pattern in Chapter 5), the priorities of the tasks are known at design time. The priority of the task determines which tasks will run preferentially when multiple tasks are ready to run—the highest-priority task that is ready. This makes the timing analysis of such systems very easy to compute, as long as certain assumptions are not violated too badly. These assumptions are the following.

- Tasks are periodic with the deadlines coming at the end of the periods.
- Infinite preemptibility—a lower-priority task can be preempted immediately when a higher-priority task becomes ready to run.
- Independence—tasks are independent from each other.

When these conditions are true, then the following standard rate monotonic analysis formula may be applied.

$$\sum_{n} \frac{C_j}{T_j} \leq n\left(2^{\frac{1}{n}} - 1\right)$$

Note that it is "2 raised to the power of (1/n)", where C_j is the worst-case amount of time required for task j to execute, T_j is its period, and n is the number of tasks. [2], [3] If the inequality is true, then the system is *schedulable*—that is, the system will *always* meet its deadlines. Aperiodic tasks are generally handled by assuming they are periodic and using the minimum arrival time between task invocations as the period, often resulting in an overly strong but sufficient condition. The assumption of infinite preemptibility is usually not a problem if the task has very short critical sections during which it cannot be preempted—short with respect to the execution and period times. The problem of independence is, however, much stickier.

If resources are sharable (in the sense that they can permit simultaneous access by multiple clients), then no problem exists. However many, if not most, resources cannot be shared. The common solution to this problem was addressed in the Guarded Call Pattern of Chapter 5 using a mutual-exclusion semaphore to serialize access to the resource. This means that if a low-priority task locks a resource and then a higher-priority task that needs the resource becomes ready to run, it must *block* and allow the low-priority task to run until it can release the resource so that the

higher-priority task can run. A simple example of this is shown in the timing diagram in Figure 7-1.

In the figure, Task 1 is the higher-priority task. Since Task 2 runs first and locks the resource, when Task 1 is ready to run, it cannot because the needed resource is unavailable. It therefore must block and allow Task 2 to complete its use of the resource. During the period of time between marks C and D, Task 1 is said to be *blocked*. A task is blocked when it is prevented from running by a lower-priority task. This can only occur when resources are shared via mutual exclusion.

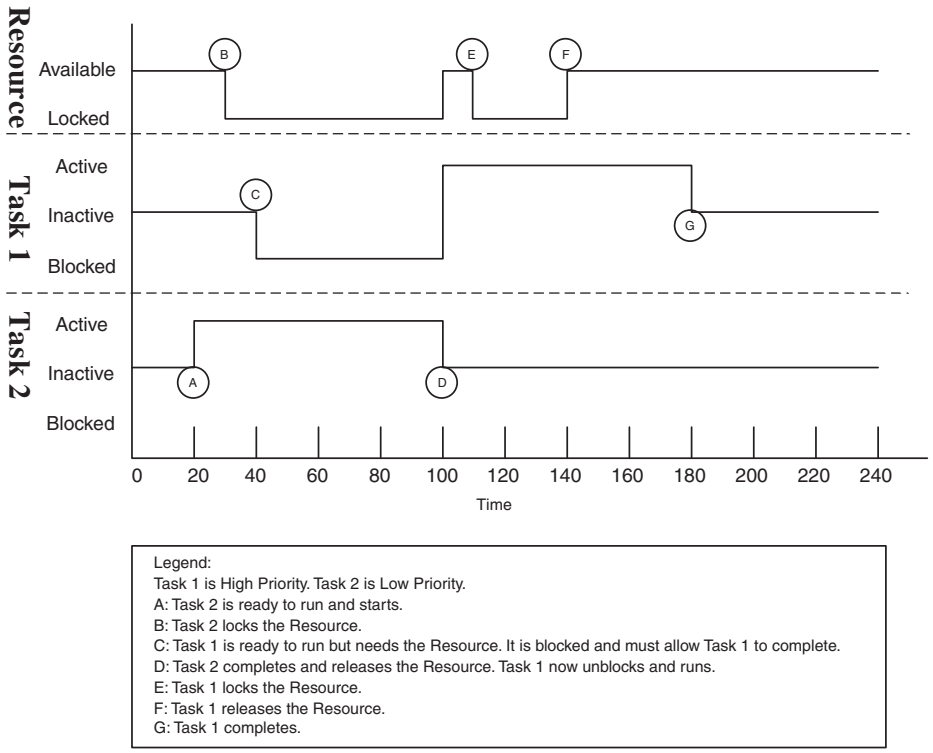


Figure 7-1: *Task Blocking*[1]

1. A note about notation. These are *timing diagrams.* They show linear time along the X-axis and "state" or "condition," a discrete value, along the Y-axis. For more information, see [4].

The problem with blocking is that the analysis of the timeliness becomes more difficult. When Task 1 is blocked, the system is said to be in a state of *priority inversion* because a lower-priority task has the thread focus even though a higher-priority task is ready to run. One can imagine third and fourth tasks of intermediate priority that don't share the resource (and are therefore able to preempt Task 2) running and preempting Task 2, thereby lengthening the amount of time before Task 2 releases the resource and allowing Task 1 to run. Because an arbitrary number of tasks can be fit in the priority scheme between Task 1 and Task 2, this problem is called *unbounded priority inversion* and is a serious problem for the schedulability of tasks. Figure 7-2 illustrates this problem by adding intermediate-priority Tasks X and Y to the system. Note that for some period of time, Task 1, the highest-priority task in the system, is blocked by *all three* remaining tasks.

To compute the schedulability for task sets with blocking, the modified RMA inequality is used.

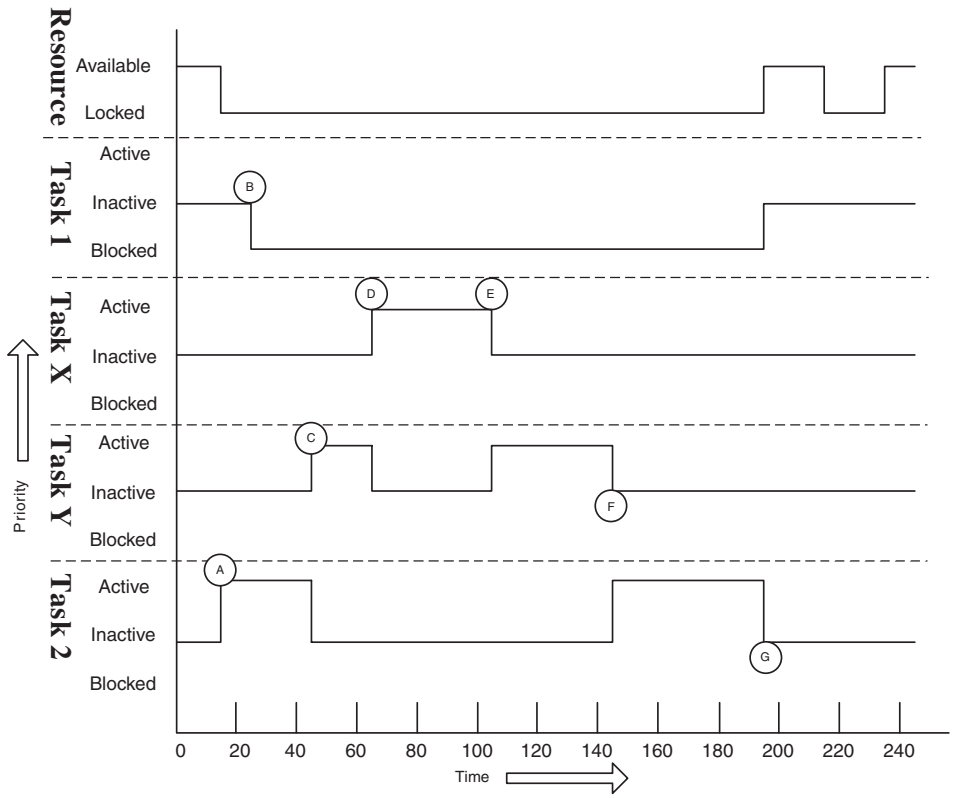
$$\sum_j \frac{C_j}{T_j} + \max\left(\frac{B_1}{T_1}, \cdots, \frac{B_{n-1}}{T_{n-1}}\right) \leq n\left(2^{\frac{1}{n}} - 1\right)$$

where B_j is the blocking time for task j—that is, the worst-case time that the task can be prevented from execution by a lower-priority task owning a needed resource. The problem is clear from the inequality—unbounded blocking means unbounded blocking time, and nothing useful can be said about the ability of such a system to meet its deadlines.

Unbounded priority inversion is a problem that is addressed by the first three patterns in this chapter. Note that priority inversion is a necessary consequence of resource sharing with mutual exclusion locking, but it can be bounded using these patterns.

These first three patterns solve, or at least address, the problem of resource sharing for schedulability purposes, but for the most part they don't deal with the issue of deadlock. A deadlock is a condition in which clients of resources are waiting for conditions to arise that cannot in principle ever occur. An example of deadlock is shown in Figure 7-3.

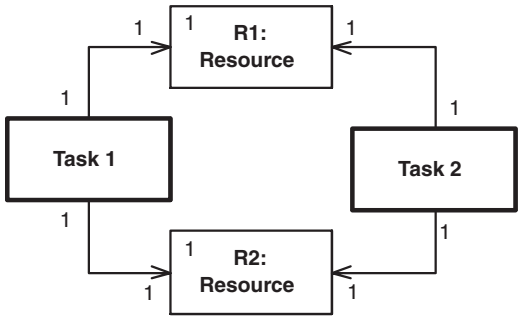
In Figure 7-3, there are two tasks, Task 1 and Task 2, that share two resources, R1 and R2. Task 1 plans to lock R2 and then lock R1 and release them in the opposite order. Task 2 plans to lock R1 and then R2 and release them in the reverse order. The problem arises when Task 1 preempts Task 2 when it has a single resource (R1) locked. Task 1 is a higher



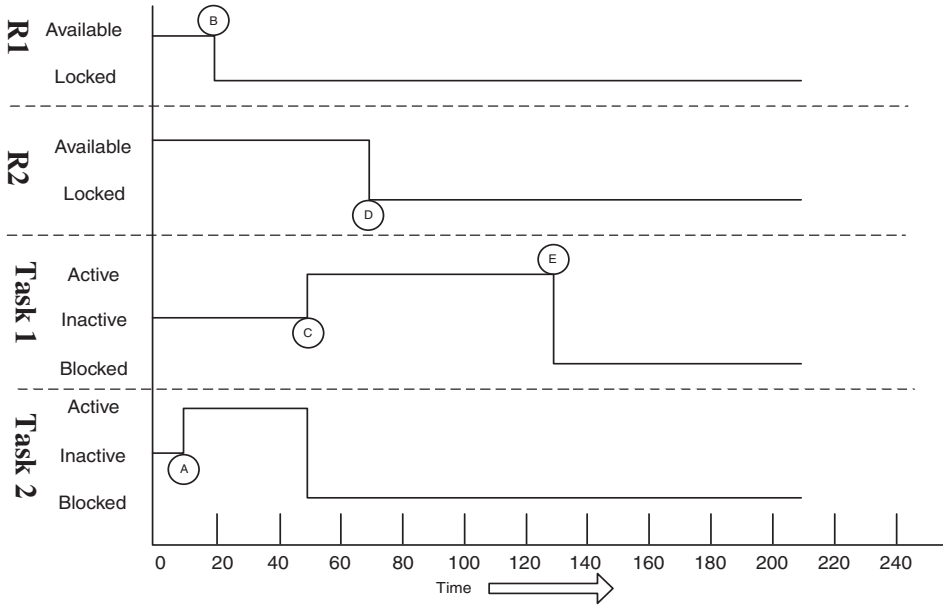
Legend:
Priorities: Task 1 > Task X > Task Y > Task 2
A: Task 2 is ready to run and starts.
B: Task 1 is ready to run but needs the Resource. It is blocked and must allow Task 2 to complete.
C: Task Y, which is a higher priority than Task 2, is ready to run. Since it doesn't need the resource, it preempts Task 2. Task 1 is now effectively blocked by both Task 2 and Task Y.
D: Task X, which is a higher priority than Task Y, is ready to run. Since it doesn't need the resource, it preempts Task Y. Task 1 is now effectively blocked by 3 tasks.
E: Task X completes, allowing Task Y to resume.
F: Task Y completes, allowing Task 2 to resume.
G: Task 2 (finally) completes and releases the resource, allowing Task 1 to access the resource.

Figure 7-2: *Unbounded Task Blocking*

priority, so it can preempt Task 1, and it doesn't need a currently locked resource, so things are fine. It goes ahead and locks R2. Now it decides that it needs the other resource, R1, which, unfortunately is locked by the



a. Deadlocked Class Structure



Legend:
Priority: Task 1 > Task 2
A: Task 2 runs with the intent of locking R1, then R2.
B: Task 2 locks R1 and is about to lock R2, when...
C: Task 1 runs, preempting Task 2, with the intent of locking R2, then R1.
D: Task 2 locks R2.
E: Now Task 2 needs to lock R1, but R1 is already locked. So Task 2 must block until Task 1 can release R1. However, Task 1 cannot run to release R1 because it needs R2, which is locked by Task 2.

b. Timing

Figure 7-3: *Deadlock*

blocked task, Task 2. So Task 1 cannot move forward and must block in order to allow Task 2 to run until it can release the now needed resource (R1). So Task 2 runs but finds that it now needs the other resource (R2) owned by the blocked Task 1. At this point, each task is waiting for a condition that can never be satisfied, and the system stops.

In principle, a deadlock needs the following four conditions to occur.

1. Mutual exclusion (locking) of resources
2. Resources are held (locked) while others are waited for
3. Preemption while holding resources is permitted
4. A circular wait condition exists (for example, P1 waits on P2, which waits on P3, which waits on P1)

The patterns for addressing deadlock try to ensure that at least one of the four necessary conditions for deadlock cannot occur. The Simultaneous Locking Pattern breaks condition 2, while the Ordered Locking Pattern breaks condition 4. The Priority Ceiling Pattern is a pattern that solves both the scheduling problem and the deadlock problem.

7.2 CRITICAL SECTION PATTERN

The Critical Section Pattern is the simplest pattern to share resources that cannot be shared simultaneously. It is lightweight and easy to implement, but it may prevent high priority tasks, even ones that don't use *any* resources, from meeting their deadlines if the critical section lasts too long.

7.2.1 Abstract

This pattern has been long used in the design of real-time and embedded systems whenever a resource must have at most a single owner at any given time. The basic idea is to lock the Scheduler whenever a resource is accessed to prevent another task from simul-

taneously accessing it. The primary advantage of this pattern is its simplicity, both in terms of understandability and in terms of implementation. It becomes less applicable when the resource access may take a long time because it means that higher-priority tasks may be blocked from execution for a long period of time.

7.2.2 Problem

The main problem addressed by the Critical Section Pattern is how to robustly share resources that may have, at most, a single owner at any given time.

7.2.3 Pattern Structure

Figure 7-4 shows the basic structural elements in the Critical Section Pattern.

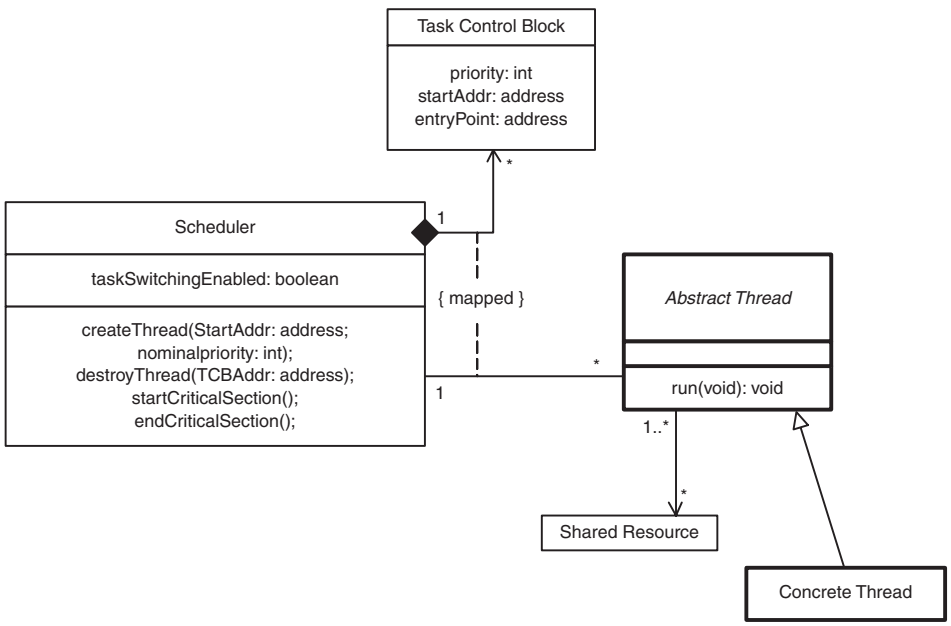


Figure 7-4: *Critical Section Pattern*

7.2.4 Collaboration Roles

- *Abstract Thread*
  The *Abstract Thread* class is an abstract (noninstantiable) superclass for *Concrete Thread. Abstract Thread* associates with the *Scheduler.* Since *Concrete Thread* is a subclass, it has the same interface to the *Scheduler* as the *Abstract Thread.* This enforces interface compliance. The *Abstract Thread* is an «active» object, meaning that when it is created, it creates an OS thread in which to run. It contains (that is, it has composition relations with) more primitive application objects that execute in the thread of the composite «active» object.

- *Concrete Thread*
  The *Concrete Thread* is an «active» object most typically constructed to contain passive "semantic" objects (via the composition relation) that do the real work of the system. The *Concrete Thread* object provides a straightforward means of attaching these semantic objects into the concurrency architecture. *Concrete Thread* is an instantiable subclass of *Abstract Thread.*

- *Scheduler*
  This object orchestrates the execution of multiple threads based on some scheme requiring preemption. When the «active» *Thread* object is created, it (or its creator) calls the *createThread* operation to create a thread for the «active» object. Whenever this thread is executed by the *Scheduler,* it calls the StartAddr address (except when the thread has been blocked or preempted—in which case it calls the *EntryPoint* address).

  In this pattern, the *Scheduler* has a Boolean attribute called taskSwitchingEnabled and two operations, startCriticalSection() and endCriticalSection(), which manipulate this attribute. When FALSE, it means that the *Scheduler* will not perform any task switching; when TRUE, tasks will be switched according to the task scheduling policies in force.

- *Shared Resource*
  A resource is an object shared by one or more *Threads* but cannot be reliably accessed by more than one client at any given time. All operations defined on this resource that access any part of the resource that is not simultaneously sharable (its nonreentrant parts) should call *Scheduler.startCriticalSection()* before

they manipulate the internal values of the resource and should call *Scheduler.endCriticalSection()* when they are done.

- *Task Control Block*
  The TCB contains the scheduling information for its corresponding *Thread* object. This includes the priority of the thread, the default start address, and the current entry address if it was preempted or blocked prior to completion. The *Scheduler* maintains a TCB object for each existing *Thread.* Note that TCB typically also has a reference off to a call and parameter stack for its *Thread,* but that level of detail is not shown in Figure 7-4.

7.2.5 Consequences

The designers and programmers must show good discipline in ensuring that every resource access locks the resource before performing any manipulation of the source. This pattern works by effectively making the current task the highest-priority task in the system. While quite successful at preventing resource corruption due to simultaneous access, it locks out all higher-priority tasks from executing during the critical section, even if they don't require the use of the resource. Many systems find this blocking delay unacceptable and must use more elaborate means for resource sharing. Further, if the initial task that locks the resource neglects to deescalate its priority, then all other tasks are permanently prevented from running. Calculation of the worst-case blocking for each task is trivial with this pattern: It is simply the longest critical section of any single task of lesser priority.

It is perhaps obvious, but should nevertheless be stated, that when using this pattern a task should never suspend itself while owning a resource because task switching is disabled so that in a situation like that no tasks are permitted to run at all. This pattern has the advantage in that it avoids deadlock by breaking the second condition (holding resources while waiting for others) as long as the task releases the resource (and reenables task switching) before it suspends itself.

7.2.6 Implementation Strategies

All commercial RTOSs have a means for beginning and ending a critical section. Invoking this Scheduler operation prevents all task

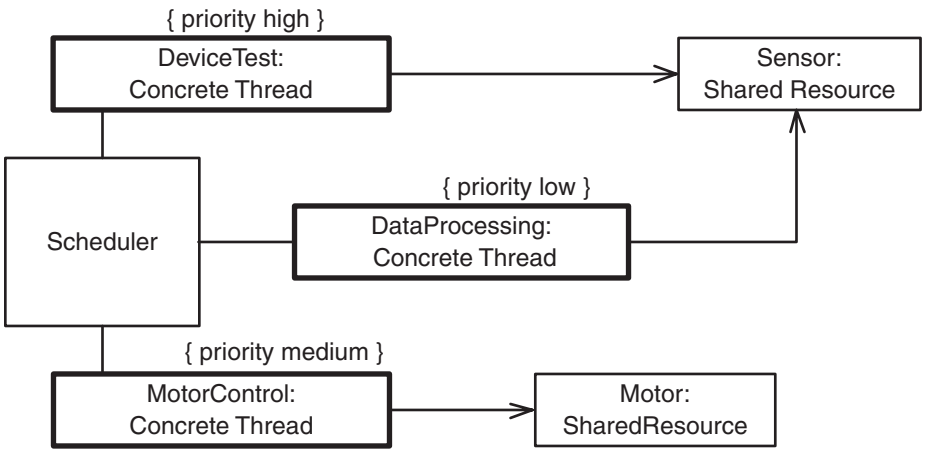
switching from occurring during the critical section. If you write your own RTOS, the most common way to do this is to set the Disable Interrupts bit on your processor's flags register. The precise details of this vary, naturally, depending on the specific processor.

7.2.7 Related Patterns

As mentioned, this is the simplest pattern that addresses the issue of sharing nonreentrant resources. Other resource sharing approaches, such as Priority Inheritance, Highest Locker, and Priority Ceiling Patterns, solve this problem as well with less impact on the schedulability of the overall system but at the cost of increased complexity. This pattern can be mixed with all of the concurrency patterns from Chapter 5, except the Cyclic Executive Pattern, for which resource sharing is a nonissue.

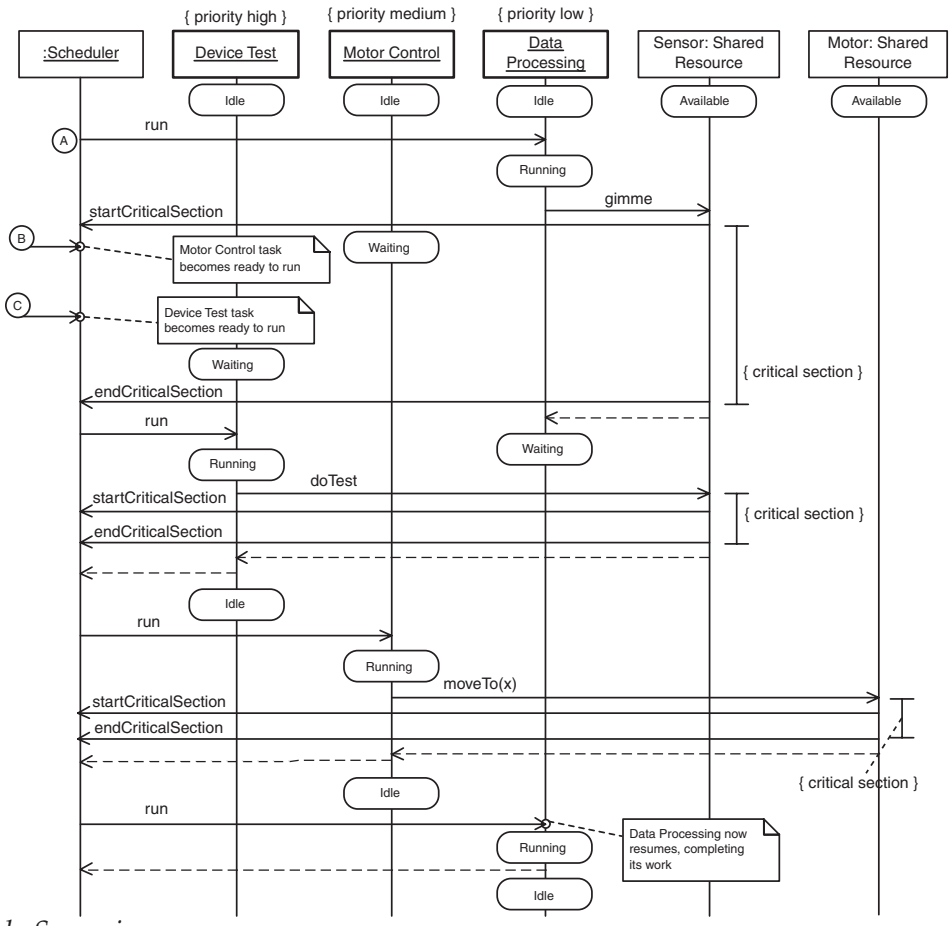
7.2.8 Sample Model

An example of the use of this pattern is shown in Figure 7-5. This example contains three tasks: Device Test (highest priority), Motor Control (medium priority), and Data Processing (lowest priority). Device Test and Data Processing share a resource called Sensor, whereas Motor Control has its own resource called Motor.



a. Structure

Figure 7-5: *Critical Section Pattern Example*



b. Scenario

Figure 7-5: *Critical Section Pattern Example (continued)*

The scenario starts off with the lowest-priority task, Data Processing, accessing the resource that starts up a critical section. During this critical section both the Motor Control task and the Device Test task become ready to run but cannot because task switching is disabled. When the call to the resource is almost done, the *Sensor.gimme()* operation makes a call to the scheduler to end the critical section. The scenario shows three critical sections, one for each of the running tasks. Finally, at the end, the lowest-priority task is allowed to complete its work and then returns to its Idle state.

7.3 PRIORITY INHERITANCE PATTERN

The Priority Inheritance Pattern reduces priority inversion by manipulating the executing priorities of tasks that lock resources. While not an ideal solution, it significantly reduces priority inversion at a relatively low run-time overhead cost.

7.3.1 Abstract

The problem of unbounded priority inversion is a very real one and has accounted for many difficult-to-identify system failures. In systems running many tasks, such problems may not be at all obvious, and typically the only symptom is that occasionally the system fails to meet one or more deadlines. The Priority Inheritance Pattern is a simple, low-overhead solution for limiting the priority inversion to at most a single level—that is, at most, a task will only be blocked by a single, lower-priority task owning a needed resource.

7.3.2 Problem

The unbounded priority inversion problem is discussed in the chapter introduction in some detail. The problem addressed by this pattern is to bound the maximum amount of priority inversion.

7.3.3 Pattern Structure

Figure 7-6 shows the structure of the pattern. The basic elements of this pattern are familiar: Scheduler, Abstract Task, Task Control Block, and so on. This can be thought of as an elaborated subset of the Static Priority Pattern, presented in Chapter 5. Note the use of the «frozen» constraint applied to the Task Control Block's *nominalPriority* attribute. This means the attribute is unchangeable once the object is created.

7.3.4 Collaboration Roles

- *Abstract Thread*
  The *Abstract Thread* class is an abstract (noninstantiable) superclass for *Concrete Thread. Abstract Thread* associates with the *Scheduler.*

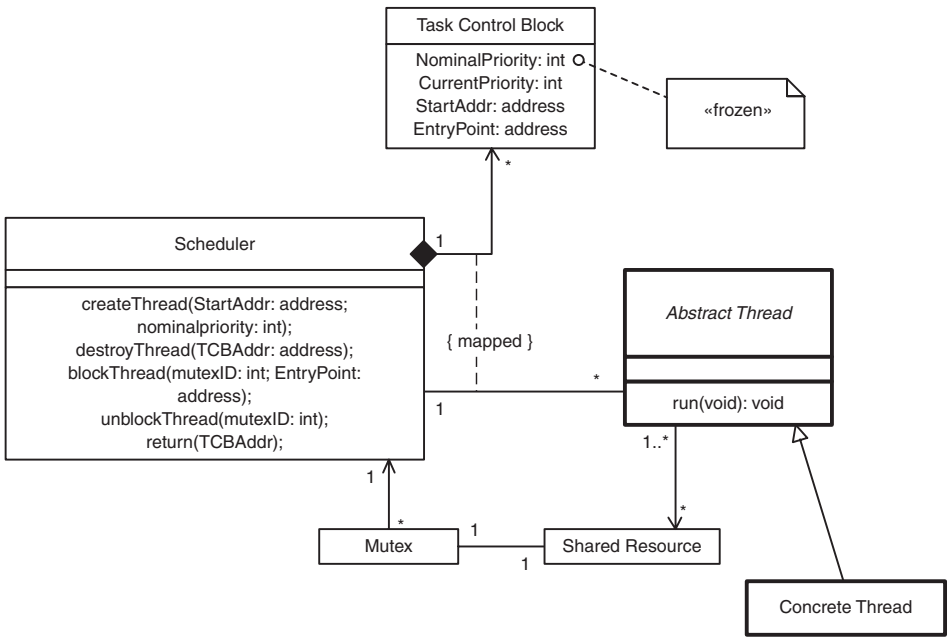


Figure 7-6: *Priority Inheritance Pattern*

Since *Concrete Thread* is a subclass, it has the same interface to the *Scheduler* as the *Abstract Thread*. This enforces interface compliance. The *Abstract Thread* is an «active» object, meaning that when it is created, it creates an OS thread in which to run. It contains (that is, it has composition relations with) more primitive application objects that execute in the thread of the composite «active» object.

- *Concrete Thread*
  The *Concrete Thread* is an «active» object most typically constructed to contain passive "semantic" objects (via the composition relation) that do the real work of the system. The *Concrete Thread* object provides a straightforward means of attaching these semantic objects into the concurrency architecture. *Concrete Thread* is an instantiable subclass of *Abstract Thread*.
- *Mutex*
  The *Mutex* is a mutual exclusion semaphore object that permits only a single caller through at a time. The operations of the *Shared Resource* invoke it whenever a relevant service is called, locking it

prior to starting the service and unlocking it once the service is complete. *Threads* that attempt to invoke a service when the services are already locked become blocked until the *Mutex* is in its unlocked state. This is done by the *Mutex* semaphore signaling the *Scheduler* that a call attempt was made by the currently active thread, the *Mutex* ID (necessary to unlock it later when the *Mutex* is released), and the entry point—the place at which to continue execution of the *Thread.*

- *Scheduler*
  This object orchestrates the execution of multiple threads based on their priority according to a simple rule: Always run the ready thread with the highest priority. When the «active» *Thread* object is created, it (or its creator) calls the *createThread* operation to create a thread for the «active» object. Whenever this thread is executed by the *Scheduler,* it calls the StartAddr address (except when the thread has been blocked or preempted, in which case it calls the *EntryPoint* address).

  In this pattern, the *Scheduler* has some special duties when the *Mutex* signals an attempt to access a locked resource: Specifically, it must block the requesting task (done by stopping that task and placing a reference to it in the *Blocked Queue* (not shown—for details of the *Blocked Queue,* see Static Priority Pattern in Chapter 5), and it must elevate the priority of the task owning the resource to that of the highest priority *Thread* being blocked. This is easy to determine since the *Blocked Queue* is a priority FIFO—the highest-priority blocked task is the first one in that queue. Similarly, when the *Thread* releases the resource, the *Scheduler* must lower its priority back to its nominal priority.

- *Shared Resource*
  A *Shared Resource* is an object shared by one or more *Threads.* For the system to operate properly in all cases, all shared resources must either be reentrant (meaning that corruption from simultaneous access cannot occur) or they must be protected. In the case of a protected resource, when a *Thread* attempts to use the resource, the associated *Mutex* semaphore is checked, and if locked, the calling task is placed into the *Blocked Queue.* The task is terminated with its reentry point noted in the TCB.

- *Task Control Block*
  The TCB contains the scheduling information for its corresponding *Thread* object. This includes the priority of the thread, the default start address and the current entry address, if it was preempted or blocked prior to completion. The *Scheduler* maintains a TCB object for each existing *Thread.* Note that TCB typically also has a reference off to a call and parameter stack for its *Thread,* but that level of detail is not shown in Figure 7-6. The TCB tracks both the current priority of the thread (which may have been elevated due to resource access and blocking) and its nominal priority.

7.3.5 Consequences

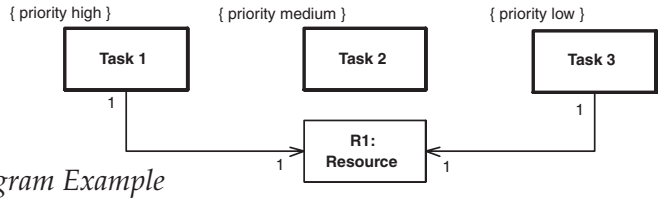
The Priority Inheritance Pattern handles well the problem of priority inversion when at most a single resource is locked at any given time and prevents unbounded priority inversion in this case. This is illustrated in Figure 7-7. With naïve priority management, Task 1, the highest-priority task in the system, is delayed from execution until Task 2 has completed. Using the Priority Inheritance Pattern, Task 1 completes as early as possible.

When there are multiple resources that may be locked at any time, this pattern exhibits behavior called *chain blocking.* That is, one task may block another, which blocks another, and so on. This is illustrated in the only slightly more complex example in Figure 7-8. The timing diagram in Figure 7-8b shows that Task 1 is blocked by Task 2 *and* Task 3 at Point G.

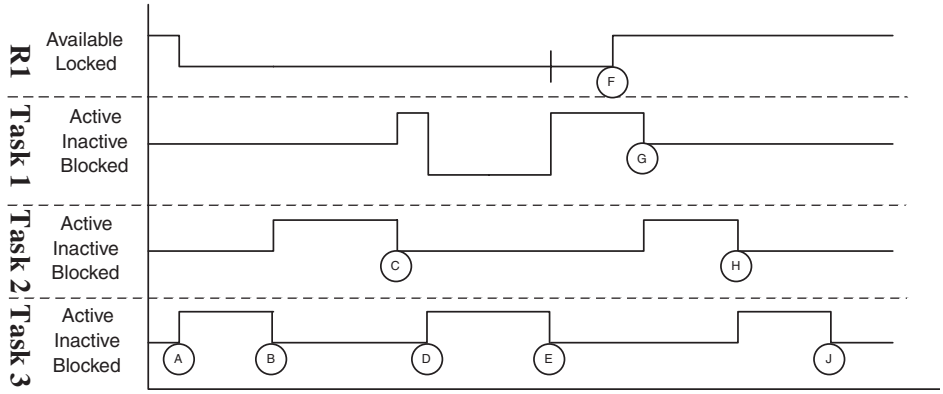
In general, the Priority Inheritance Pattern greatly reduces unbounded blocking. In fact, though, the number of blocked tasks at any given time is bounded only by the lesser of the number of tasks and the number of currently locked resources. There is a small amount of overhead to pay when tasks are blocked or unblocked to manage the elevation or depression of the priority of the tasks involved. Computation of a single task's worst-case blocking time involves computation of the worst-case chain blocking of all tasks of lesser priority.

This pattern does not address deadlock issues at all, so it is still possible to construct task models using this pattern that have deadlock.

Another consequence of the use of the priority inheritance patterns (Priority Inheritance Pattern, Highest Locker Pattern, and Priority Ceiling Pattern) is the overhead. The use of semaphores and



a. Task Diagram Example

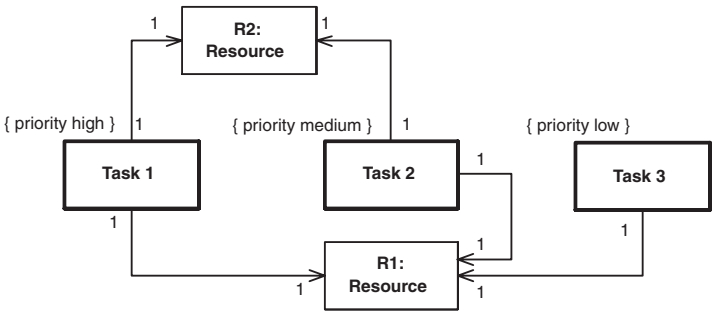


Legend:
Priority: Task 1 > Task 2 > Task 3
A: Task 3 runs at its nominal (i.e., the lowest) priority and locks R1.
B: Task 2 runs, and since it is a higher priority and doesn't need R1, it preempts Task 3.
C: Task 1 runs, preempting Task 2.
D: Task 1 attempts to lock R1. Because it is owned by Task 3, Task 1 must block. Task 3's priority is elevated to that of Task 1. Task 3 now continues.
E: Task 3 releases R1. Its priority is reduced, and Task 1 unblocks and runs, locking R1.
F: Task 1 releases R1.
G: Task 1 completes. Task 2, the highest-priority task ready to run, now continues.
H: Task 2 completes. Task 3 now can run to completion.
J: Task 3 completes.

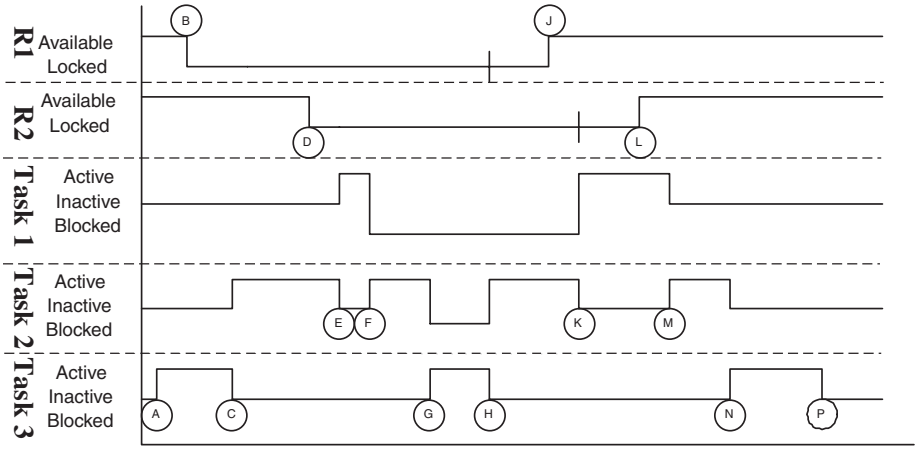
b. Behavior

Figure 7-7: *Priority Inheritance Pattern*

blocking involves task switching whenever a locked mutex is requested and another task switch whenever a waited-for mutex is released. In addition, the acts of blocking and unblocking tasks during those task context switches involves the manipulation of priority queues. Further, the use of priority inheritance means that there is some overhead in the escalation and deescalation of priorities. If blocking occurs infrequently, then this overhead will be slight, but if there is a great deal of contention for resources, then the overhead can be severe.



a. Task Diagram with Chain-Blocking Example



Legend:
Priority: Task 1 > Task 2 > Task 3
A: Task 3 runs.
B: Task 3 locks R1.
C: Task 2 becomes ready to run and preempts Task 3.
D: Task 2 locks R2.
E: Task 1 becomes ready to run and preempts Task 2.
F: Task 1 attempts to lock R2. But it must block and allow Task 2 to run. Task 2's priority is elevated to that of Task 1, and Task 2 runs.
G: Task 2 attempts to access R1, which is locked by Task 3. It must block to allow Task 3 to run. Task 3 runs at Task 1's priority. Task 1 is now blocked by both Task 2 and Task 3.
H: Task 3 releases R1. Task 2 immediately preempts Task 3 and locks R1 and runs.
J: Task 2 releases R1 and continues to run.
K: Task 2 releases R2 (the resource Task 1 is blocked on). Task 1 immediately preempts Task 2 and locks R2.
L: Task 1 releases R2.
M: Task 1 completes. This allows Task 2 to run.
N: Task 2 completes. This allows Task 3 to run.
P: Task 3 completes.

b. Chain-Blocking Behavior

Figure 7-8: *Priority Inheritance Pattern*

7.3.6 Implementation Strategies

Some RTOS directly support the notion of priority inheritance, and so it is very little work to use this pattern with such an RTOS. If you are using an RTOS that does not support it, or if you are writing your own RTOS, then you must extend the RTOS (many RTOSs have API for just this purpose) to call your own function when the mutex blocks a task on a resource. The *Scheduler* must be able to identify the priority of the thread being blocked (a simple matter because it is in the Task Control Block for the task) in order to elevate the priority of the task currently owning the resource.

It is possible to build in the nominal priority as a constant attribute of the Concrete Thread. When the Concrete Thread always runs at a given priority, then the constructor of the «active» object should do exactly that. Otherwise, the creator of that active object should specify the priority at which that task should run.

In virtually all other ways, the implementation is very similar to the implementation of standard concurrency patterns, such as the Static Priority Pattern presented in Chapter 5.

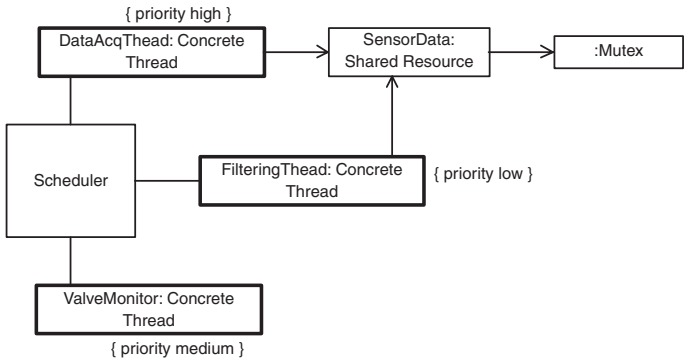
7.3.7 Related Patterns

The Priority Inheritance Pattern exists to help solve a particular problem peculiar to priority-based preemption multitasking, so all of the concurrency patterns having to do with that style of multitasking can be mixed with this pattern.

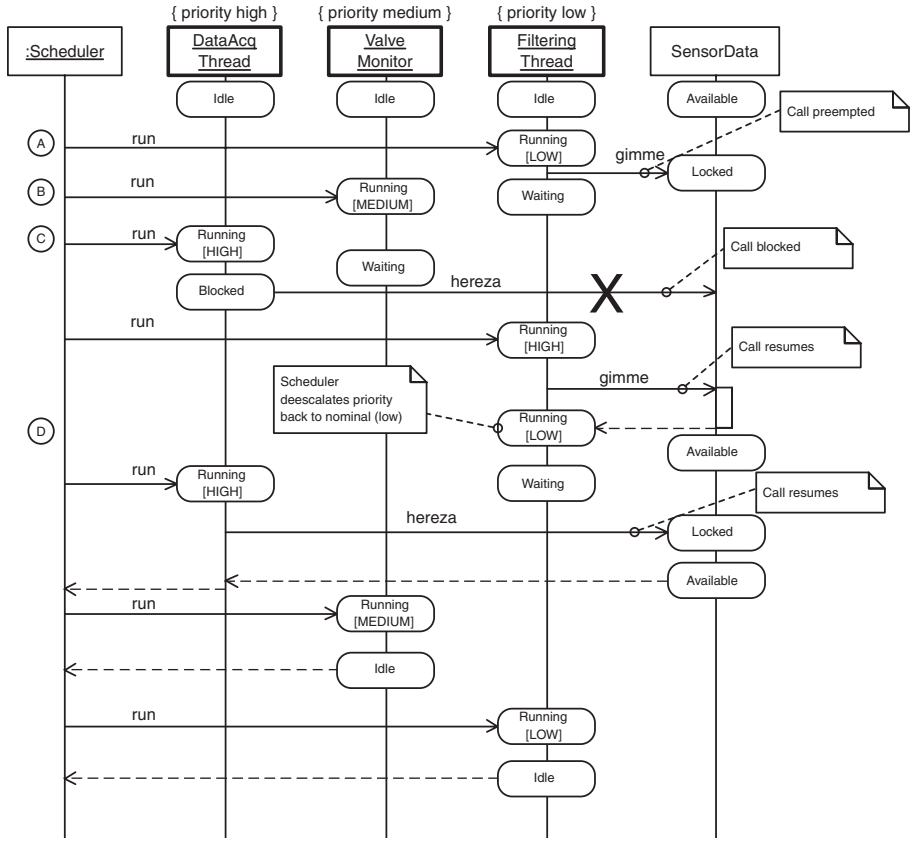
While this pattern is lightweight, it greatly reduces priority inversion in multitasking systems. However, there are other approaches that can reduce it further, such as Priority Ceiling Pattern and Highest Locker Pattern. In addition, Priority Ceiling Pattern also removes the possibility of deadlock.

7.3.8 Sample Model

Figure 7-9 provides an example to illustrate how the Priority Inheritance Pattern works. States of the objects are shown using standard UML—that is, as state marks on the instance lifelines. Some of the returns are shown, again using standard UML dashed lines. Showing that a call cannot complete is indicated with a large X on the call—*not* standard UML, but clear as to its interpretation.



a. Priority Inheritance Pattern Example



b. Scenario

Figure 7-9: *Priority Inheritance Pattern*

The flow of the scenario in Figure 7-9b is straightforward. All tasks begin the scenario in the Idle state. Then, at point A, the *FilteringThread* task becomes ready to run. It runs at its nominal priority, which is LOW (the priority of the thread is shown inside square brackets in the Running state mark—again, not quite standard UML, but parsimonious). It then calls the resource *SensorData* that then enters the Locked state.

At point B, the *ValveMonitor* task becomes ready to run. It preempts the *FilteringThread* because the former is of higher priority. The *ValveMonitor* task runs for a while, but at point C, task *DataAcqThread* becomes ready to run. Since it is the highest priority, it preempts the *ValveMonitor* thread. *DataAcqThread* object then tries to access the *SensorData* object and finds that it cannot because the latter is locked with a *Mutex* semaphore (not shown in the scenario). The *Scheduler* then blocks the *DataAcqThread* thread and runs the *FilteringThread* at the same priority as *DataAcqThread* because the *FilteringThread* inherits the priority from the highest blocking task—in this case the *DataAcqThread* task. Note at this point, the medium-priority task, *ValveMonitor,* is in the state Waiting. Without priority inheritance, if *DataAcqThread* is blocked, the *ValveMonitor* would run because it has the next highest priority.

At point D, *FilteringThread's* use of the resource is complete, and it releases the resource (done at the end of the *SensorData.gimme* operation). As it returns, the *Mutex* signals the *Scheduler* that it is now available, so the Scheduler deescalates *FilteringThread's* priority to its nominal value (LOW) and unblocks the highest-priority task, *DataAcqThread.* This task now runs to completion and returns. The *Scheduler* then runs the next highest-priority waiting task, *ValveMonitor,* which runs until it is done and returns. Finally, the lowest-priority task, *FilteringThread,* gets to complete.

The worst-case blocking time for the *DataAcqThread* task is then the amount of time that *FilteringThread* locks the *SensorData* resource. Without the Priority Inheritance Pattern, the worst-case blocking for *DataAcqThread* task would be the amount of time *FilteringThread* locks the *SensorData* resource plus the amount of time that *ValveMonitor* executes.

7.4 HIGHEST LOCKER PATTERN

The Highest Locker Pattern defines a *priority ceiling* with each resource. The basic idea is that the task owning the resource runs at the highest-priority ceiling of all the resources that it currently owns, provided that it is blocking one or more higher-priority tasks. This limits priority inversion to at most one level.

7.4.1 Abstract

The Highest Locker Pattern is another solution to the unbounded blocking/unbounded priority inversion problem. It is perhaps a minor elaboration from the Priority Inheritance Pattern, but it is different enough to have some different properties with respects to schedulability. The Highest Locker Pattern limits priority inversion to a single level as long as a task does not suspend itself while owning a resource. In this case, you may get chained blocking similar to the Priority Inheritance Pattern. Unlike the Priority Inheritance Pattern, however, you cannot get chained blocking if a task is preempted while owning a resource.

7.4.2 Problem

The unbounded priority inversion problem is discussed in the chapter introduction in some detail. The problem addressed by this pattern is to limit the maximum amount of priority inversion to a single level—that is, there is *at most* a single lower-priority task blocking a higher-priority task from executing.

7.4.3 Pattern Structure

The Highest Locker Pattern is shown in Figure 7-10. The structural elements of the pattern are the same as for the Priority Inheritance Pattern, with the addition of an attribute *priorityCeiling* for the *SharedResource.*

The pattern works by defining each lockable resource with a priority ceiling. The priority ceiling is just greater than the priority of the

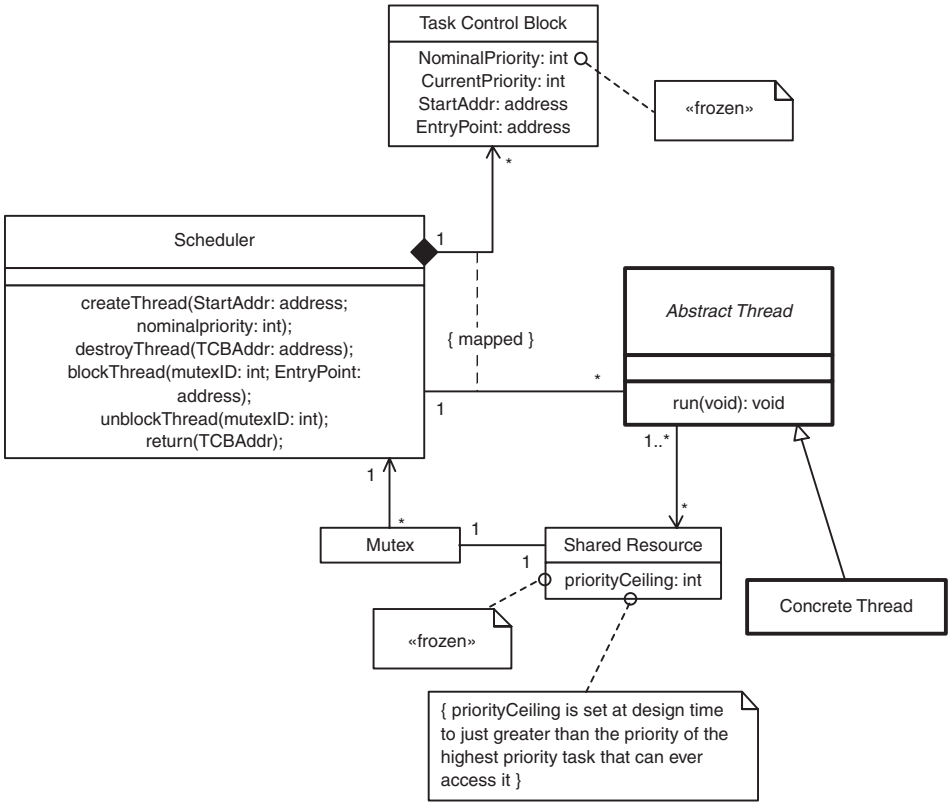


Figure 7-10: *Highest Locker Pattern*

highest-priority client of the resource—this is known at design time in a static priority scheme. When the resource is locked, the priority of the locking task is augmented to the priority ceiling of the resource.

7.4.4 Collaboration Roles

- *Abstract Thread*
  The *Abstract Thread* class is an abstract (noninstantiable) superclass for *Concrete Thread*. *Abstract Thread* associates with the

Scheduler. Since *Concrete Thread* is a subclass, it has the same interface to the *Scheduler* as the *Abstract Thread.* This enforces interface compliance. The *Abstract Thread* is an «active» object, meaning that when it is created, it creates an OS thread in which to run. It contains (that is, it has composition relations with) more primitive application objects that execute in the thread of the composite «active» object.

- *Concrete Thread*
  The *Concrete Thread* is an «active» object most typically constructed to contain passive "semantic" objects (via the composition relation) that do the real work of the system. The *Concrete Thread* object provides a straightforward means of attaching these semantic objects into the concurrency architecture. *Concrete Thread* is an instantiable subclass of *Abstract Thread.*

- *Mutex*
  The *Mutex* is a mutual exclusion semaphore object that permits only a single caller through at a time. The operations of the *Shared Resource* invoke it whenever a relevant service is called, locking it prior to starting the service and unlocking it once the service is complete. *Threads* that attempt to invoke a service when the services are already locked become blocked until the *Mutex* is in its unlocked state. This is done by the *Mutex* semaphore signaling the *Scheduler* that a call attempt was made by the currently active thread, the *Mutex* ID (necessary to unlock it later when the mutex is released), and the entry point—the place at which to continue execution of the *Thread.*

- *Scheduler*
  This object orchestrates the execution of multiple threads based on their priority according to a simple rule: Always run the ready thread with the highest priority. When the «active» *Thread* object is created, it (or its creator) calls the *createThread* operation to create a thread for the «active» object. Whenever this thread is executed by the *Scheduler,* it calls the StartAddr address (except when the thread has been blocked or preempted—in which case it calls the *EntryPoint* address).

  In this pattern, the *Scheduler* has some special duties when the *Mutex* signals an attempt to access a locked resource. Specifically, it must block the requesting task (done by stopping that task and placing a reference to it in the *Blocked Queue* (not shown—for

details of the *Blocked Queue,* see the Static Priority Pattern in Chapter 5), and it must elevate the priority of the task owning the resource to the *Shared Resource's priorityCeiling.*

- *Shared Resource*
  A resource is an object shared by one or more *Threads.* For the system to operate properly in all cases, all *Shared Resources* must either be reentrant (meaning that corruption from simultaneous access cannot occur), or they must be protected. In the case of a protected resource, when a *Thread* attempts to use the resource, the associated *Mutex* semaphore is checked, and if locked, the calling task is placed into the *Blocked Queue.* The task is terminated with its reentry point noted in the TCB.

  The *SharedResource* has a constant attribute (note the «frozen» constraint in Figure 7-10), called priorityCeiling. This is set during design to just greater than the priority of the highest-priority task that can ever access it. In some RTOSs, this means that the priority will be one more (when a larger number indicates a higher priority), and in some it will be one less (when a lower number indicates a higher priority). This ensures that when the resource is locked, no other task using that resource can preempt it.
- *Task Control Block*
  The TCB contains the scheduling information for its corresponding *Thread* object. This includes the priority of the thread, the default start address, and the current entry address if it was preempted or blocked prior to completion. The *Scheduler* maintains a TCB object for each existing *Thread.* Note that TCB typically also has a reference off to a call and parameter stack for its *Thread,* but that level of detail is not shown in Figure 7-10. The TCB tracks both the current priority of the thread (which may have been elevated due to resource access and blocking) and its nominal priority.

7.4.5 Consequences

The Highest Locker Pattern has even better priority inversion-bounding properties than the Priority Inheritance Pattern. It allows higher-priority tasks to run, but only if they have a priority higher than the priority ceiling of the resource. The priority ceiling can be determined at design time for each resource by examining the clients

of a given resource and identifying to which active object they belong and selecting the highest from among those. The priority ceiling is this value augmented by one. Computation of worst-case blocking is the length of the longest critical section (that is, resource locking time) of any task of lesser priority as long as a task never suspends itself while owning a resource.

The pattern has the disadvantage that while it bounds priority inversion to a single level, that level happens more frequently than with some other approaches. For example, if the lowest-priority task locks a resource with the highest-priority ceiling, and during that time an intermediate priority task becomes ready to run, then it is blocked even though in this case one would prefer that the normal priority rules apply. One way to handle that is to elevate the priority of the task owning the resource only when another task attempts to lock it; until then, the locking tasks runs at its nominal priority.

In this pattern, care must be taken to ensure that a task never suspends itself while owning a resource. It is fine if it is preempted, but voluntary preemption while owning a resource can lead to chain blocking, a problem previously identified with the Priority Inheritance Pattern in the previous section. If the system allows tasks to suspend themselves while owning a resource, then the computation of worst-case blocking is computed in the same way as with the Priority Inheritance Pattern—the longest case of chain blocked must be traversed.

This pattern avoids deadlock as long as no task suspends itself while owning a resource because no other task is permitted to wait on the resource (condition 4). This is because the locking task runs at a priority higher than any of the other clients of the resource. As previously noted, there is also a consequence of computational overhead associated with the Highest Locker Pattern.

7.4.6 Implementation Strategies

Fewer RTOSs support the Highest Locker Pattern more than the basic Priority Inheritance Pattern. Implementation of this pattern in your own RTOS is fairly straightforward, with the addition of priority ceiling attributes in the *Shared Resource.* When the mutex is locked, it must notify the *Scheduler* to elevate the priority of the locking task to that resource's priority ceiling.

7.4.7 Related Patterns

The Highest Locker Pattern exists to help solve a particular problem peculiar to priority-based preemption multitasking, so all of the concurrency patterns having to do with that style of multitasking can be mixed with this pattern.

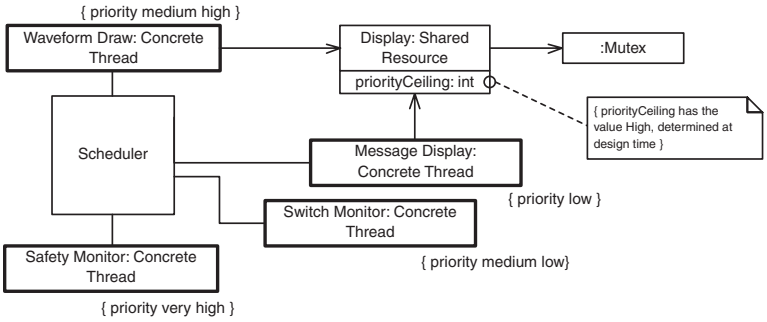
7.4.8 Sample Model

In the example shown in Figure 7-11, there are four tasks with their priorities shown using constraints, two of which, *Waveform Draw* and *Message Display,* share a common resource, *Display.* The tasks, represented as active objects in order of their priority, are *Message Display* (priority Low), *Switch Monitor* (priority Medium Low), *Waveform Draw* (priority Medium High), and *Safety Monitor* (priority Very High), leaving priority High unused at the outset. *Message Display* and *Waveform Draw* share *Display,* so the priority ceiling of *Display* is just above *Waveform Draw* (that is, High).

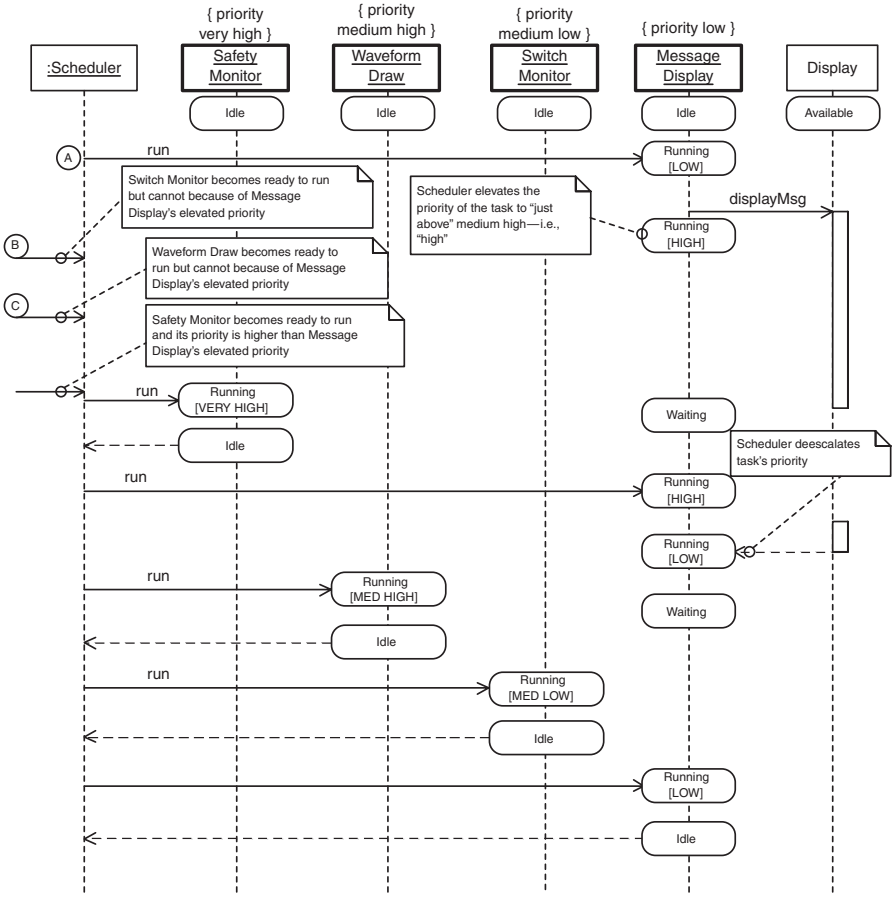
The scenario runs as follows: First, the lowest-priority task, *Message Display,* runs, calling the operation *Display.displayMsg().* Because the *Display* has a mutex semaphore, this locks the resource, and the *Scheduler* (not shown in Figure 7-11) escalates the priority of the locking task, *Message Display,* to the priority ceiling of the resource—that is, the value High.

While this operation executes, first the *Switch Monitor* and then the *Waveform Draw* tasks both become ready to run but cannot because the *Message Display* task is running at a higher priority than either of them. The *Safety Monitor* task becomes ready to run. Because it runs at a priority Very High, it can, and does, preempt the *Message Display* task.

After the *Safety Monitor* task returns control to the *Scheduler,* the *Scheduler* continues the execution of the *Message Display* task. Once it releases the resource, the mutex signals the *Scheduler,* and the latter deescalates the priority of the *Message Display* task to its nominal priority level of Low. At this point, there are two tasks of a higher priority waiting to run, so the higher-priority waiting task (*Waveform Draw*) runs, and when it completes, the remaining higher-priority task (*Switch Monitor*) runs. When this last task completes, the *Message Display* task can finally resume its work and complete.



a. Highest Locker Pattern Example



b. Scenario

Figure 7-11: *Highest Locker Pattern*

7.5 PRIORITY CEILING PATTERN

The Priority Ceiling Pattern, or Priority Ceiling Protocol (PCP) as it is sometimes called, addresses both issues of bounding priority inversion (and hence bounding blocking time) and removal of deadlock. It is a relatively sophisticated approach, more complex than the previous methods. It is not as widely supported by commercial RTOSs, however, and so its implementation often requires writing extensions to the RTOS.

7.5.1 Abstract

The Priority Ceiling Pattern is used to ensure bounded priority inversion and task blocking times and also to ensure that deadlocks due to resource contention cannot occur. It has somewhat more overhead than the Highest Locker Pattern. It is used in highly reliable multitasking systems.

7.5.2 Problem

The unbounded priority inversion problem is discussed in the chapter introduction in some detail. The Priority Ceiling Pattern exists to limit the maximum amount of priority inversion to a single level and to completely prevent resource-based deadlock.

7.5.3 Pattern Structure

Figure 7-12 shows the Priority Ceiling Pattern structure. The primary structural difference between the Priority Ceiling Pattern and the Highest Locker Pattern is the addition of a System Priority Ceiling attribute for the Scheduler. Behaviorally, there are some differences as well. The algorithm for starting and ending a critical section is shown in Figure 7-13.

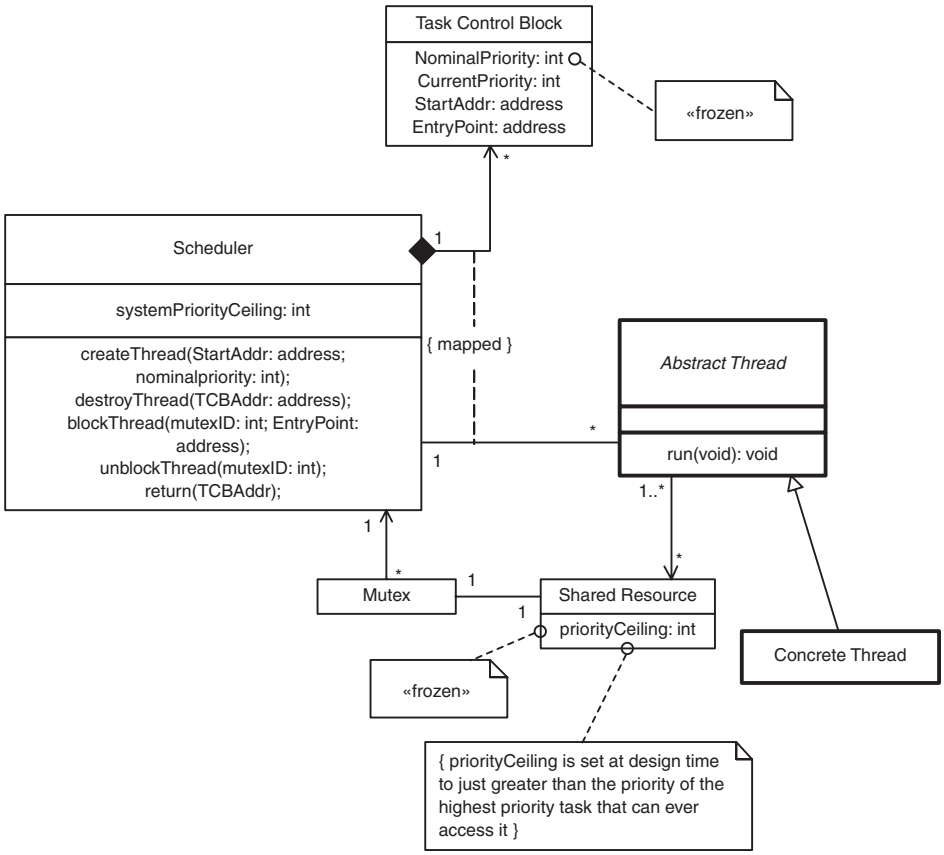


Figure 7-12: *Priority Ceiling Pattern*

7.5.4 Collaboration Roles

- *Abstract Thread*
  The *Abstract Thread* class is an abstract (noninstantiable) superclass for *Concrete Thread*. *Abstract Thread* associates with the *Scheduler*. Since *Concrete Thread* is a subclass, it has the same interface to the *Scheduler* as the *Abstract Thread*. This enforces interface compliance. The *Abstract Thread* is an «active» object, meaning that when it is created, it creates an OS thread in which to run. It

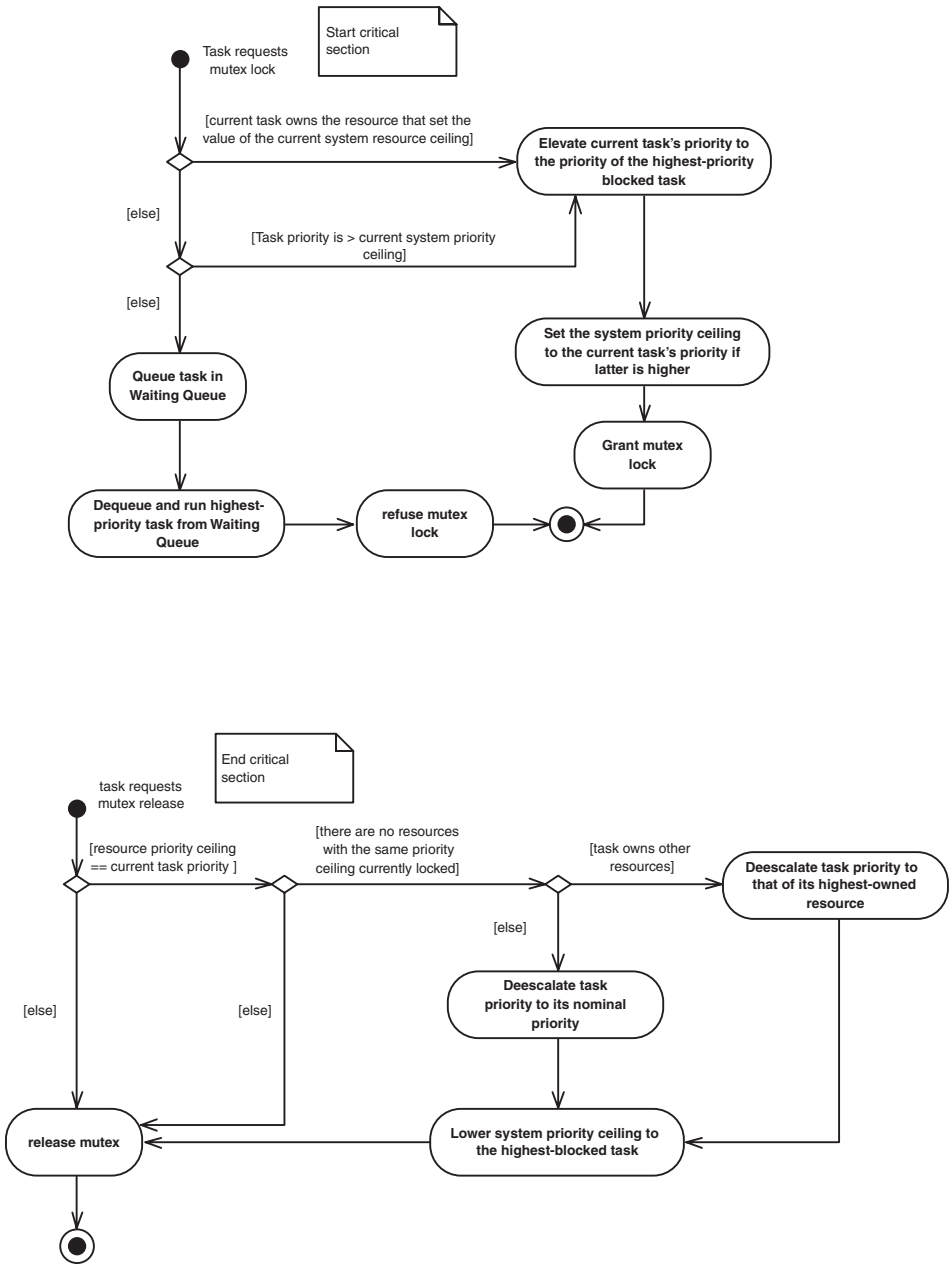


Figure 7-13: *Priority Ceiling Pattern Resource Algorithm*

contains (that is, it has composition relations with) more primitive application objects that execute in the thread of the composite «active» object.

- *Concrete Thread*
  The *Concrete Thread* is an «active» object most typically constructed to contain passive "semantic" objects (via the composition relation) that do the real work of the system. The *Concrete Thread* object provides a straightforward means of attaching these semantic objects into the concurrency architecture. *Concrete Thread* is an instantiable subclass of *Abstract Thread.*
- *Mutex*
  The *Mutex* is a mutual exclusion semaphore object that permits only a single caller through at a time. The operations of the *Shared Resource* invoke it whenever a relevant service is called, locking it prior to starting the service and unlocking it once the service is complete. *Threads* that attempt to invoke a service when the services are already locked become blocked until the *Mutex* is in its unlocked state. This is done by the *Mutex* semaphore signaling the *Scheduler* that a call attempt was made by the currently active thread, the *Mutex* ID (necessary to unblock the correct *Thread* later when the *Mutex* is released), and the entry point—the place at which to continue execution of the *Thread.* See Figure 7-13 for the algorithms that control locking, blocking, and releasing the *Mutex.*
- *Scheduler*
  This object orchestrates the execution of multiple threads based on their priority according to a simple rule: Always run the ready thread with the highest priority. When the «active» *Thread* object is created, it (or its creator) calls the *createThread* operation to create a thread for the «active» object. Whenever this thread is executed by the *Scheduler,* it calls the StartAddr address (except when the thread has been blocked or preempted—in which case it calls the *EntryPoint* address).

  In this pattern, the *Scheduler* has some special duties when the *Mutex* signals an attempt to access a locked resource. Specifically, under some conditions, it must block the requesting task (done by stopping that task and placing a reference to it in the *Blocked Queue* (not shown—for details of the *Blocked Queue,* see the Static Priority Pattern in Chapter 5), and it must elevate the priority of

the highest-priority blocked *Thread* being blocked. This is easy to determine, since the *Blocked Queue* is a priority FIFO—the highest-priority blocked task is the first one in that queue. Similarly, when the *Thread* releases the resource, the *Scheduler* must lower its priority back to its nominal priority. The *Scheduler* maintains the value of the highest-priority ceiling of all currently locked resources in its attribute *systemPriorityCeiling.*

- *Shared Resource*
  A resource is an object shared by one or more *Threads.* For the system to operate properly in all cases, all *Shared Resources* must either be reentrant (meaning that corruption from simultaneous access cannot occur), or they must be protected. In the case of a protected resource, when a *Thread* attempts to use the resource, the associated mutex semaphore is checked, and if locked, the calling task is placed into the *Blocked Queue.* The task is terminated with its reentry point noted in the TCB.

  The *SharedResource* has a constant attribute (note the «frozen» constraint in Figure 7-12), called priorityCeiling. This is set during design to just greater than the priority of the highest priority task that can ever access it. In some RTOSs, this means that the priority will be one more (when a larger number indicates a higher priority), and in some it will be one less (when a lower number indicates a higher priority). This ensures that when the resource is locked, no other task using that resource can preempt it.
- *Task Control Block*
  The TCB contains the scheduling information for its corresponding *Thread* object. This includes the priority of the thread, the default start address and the current entry address if it was preempted or blocked prior to completion. The *Scheduler* maintains a TCB object for each existing *Thread.* Note that TCB typically also has a reference off to a call and parameter stack for its *Thread,* but that level of detail is not shown here. The TCB tracks both the current priority of the *Thread* (which may have been elevated due to resource access and blocking) and its nominal priority.

7.5.5 Consequences

This pattern effectively enforces the desirable property that a high-priority task can at most be blocked from execution by a single critical section of a lower-priority task owning a required resource.

It can happen in the Priority Ceiling Pattern that a running task may not be able to access a resource even though it is not currently locked. This will occur if that resource's priority ceiling is less than the current system resource ceiling.

Deadlock is prevented by this pattern because condition 4 (circular wait) is prevented. Any condition that could potentially lead to circular waiting is prohibited. This does mean that a task may be prevented from accessing a resource even though it is currently unlocked.

There is also a consequence of computational overhead associated with the Priority Ceiling Pattern. This pattern is the most sophisticated of the resource management patterns presented in this chapter and has the highest computational overhead.

7.5.6 Implementation Strategies

Rather few RTOSs support the Priority Ceiling Pattern, but it can be added if the RTOS permits extension, particularly when a mutex is locked or released. If not, you can create your own Mutex and System Resource Ceiling classes that intervene with the priority management prior to handing off control to the internal RTOS scheduler. If you are writing your own scheduler, then the implementation should be a relatively straightforward extension of the Highest Locker Pattern.

7.5.7 Related Patterns

Because this pattern is the most sophisticated, it also has the most computational overhead. Therefore, under some circumstances, it may be desirable to use a less computational, if less capable, approach, such as the Highest Locker Pattern, the Priority Inheritance Pattern, or even the Critical Section Pattern.

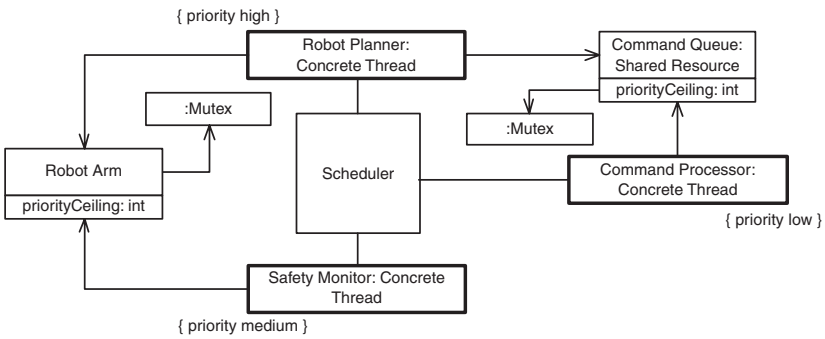
7.5.8 Sample Model

A robotic control system is given as an example in Figure 7-14a. There are three tasks. The lowest-priority task, *Command Processor,* inserts commands into a shared resource, the *Command Queue.* The middle-priority task, *Safety Monitor,* performs periodic safety monitoring, accessing the shared resource *Robot Arm.* The highest-priority task, *Robotic Planner,* accepts commands (and hence must access the *Command Queue*) and also moves the arm (and therefore must access

Robot Arm). Note that the resource ceiling of both resources must be the priority of the highest-priority task in this case because it accesses both of these resources.

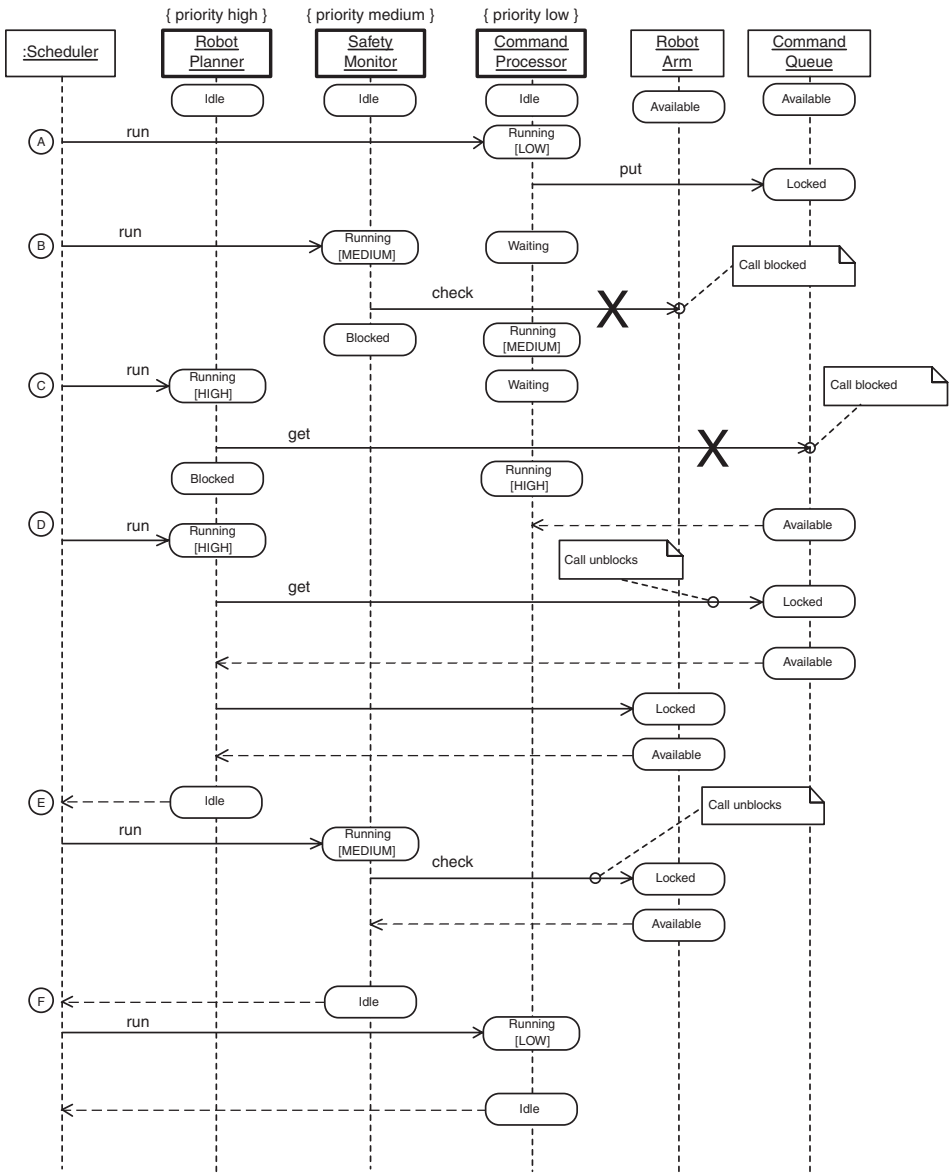
Figure 7-14b shows a scenario for the example. At point A, the *Command Processor* runs, putting set of commands into the *Command Queue.* The call to the *Command Processor* locks the resource successfully because at this point, there are no locked resources. While this is happening, the *Safety Monitor* starts to run at point B. This preempts the *Command Processor* because it is a higher priority, so *Command Processor* goes into a waiting state because it's ready to run but cannot because a higher-priority task is running. Now the *Safety Monitor* attempts to access the second resource, *Robot Arm.* Because a resource is currently already locked with same priority ceiling (found by the Scheduler examining its *systemPriorityCeiling* attribute), that call is blocked. Note that the *Safety Monitor* is prevented from running even though it is trying to access a resource that is not currently locked but *could* start a circular waiting condition, potentially leading to deadlock. Thus, the access is prevented.

When the resource access to *Safety Monitor* is prevented, the priority of the *Command Processor* is elevated to Medium, the same level as the highest-blocked task. At point C, *Robot Planner* runs, preempting the *Command Processor* task. The *Robot Planner* invokes *Command Queue.Get()* to retrieve any waiting commands but finds that this resource is locked. Therefore, its access is blocked, and it is put on the blocked queue, and the *Command Processor* task resumes but at priority High.



a. Priority Ceiling Pattern Example

Figure 7-14: *Priority Ceiling Pattern*



b. Scenario

Figure 7-14: *Priority Ceiling Pattern*

When the call to *Command Queue.put()* finally completes, the priority of the *Command Processor* task is deescalated back to its nominal priority—Low (point D). At this point in time, there are two tasks of

higher priority waiting to run. The higher priority of them, *Robot Planning* runs at its normal High priority. It accesses first the *Command Queue* resource and then the *Robot Arm* resource. When it completes, the next highest task ready to run is *Safety Monitor.* It runs, accessing the *Robot Arm* resource. When it completes, the lowest-priority task, *Command Processor* is allowed to complete its work and return control to the OS.

7.6 SIMULTANEOUS LOCKING PATTERN

The Simultaneous Locking Pattern is a pattern solely concerned with deadlock avoidance. It achieves this by breaking condition 2 (holding resources while waiting for others). The pattern works in an all-or-none fashion. Either all resources needed are locked at once or none are.

7.6.1 Abstract

Deadlock can be solved by breaking any of the four conditions required for its existence. This pattern prevents the condition of holding some resources by requesting others by allocating them all at once. This is similar to the Critical Section Pattern. However, it has the additional benefit of allowing higher-priority tasks to run if they don't need any of the locked resources.

7.6.2 Problem

The problem of deadlock is such a serious one in highly reliable computing that many systems design in specific mechanisms to detect it or avoid it. As previously discussed, deadlock occurs when a task is waiting on a condition that can never, in principle, be satisfied. There are four conditions that must be true for deadlock to occur, and it is sufficient to deny the existence of any one of these. The Simultaneous Locking Pattern breaks condition 2, not allowing any task to lock resources while waiting for other resources to be free.

7.6.3 Pattern Structure

Figure 7-15 shows the structure of the Simultaneous Locking Pattern. The special structural aspect of this pattern is the collaboration role *MultiResource.* Each *MultiResource* has a single mutex semaphore that locks only when the entire set of aggregated *Shared Resources* is available to be locked. Similarly, when the semaphore is released, all the aggregated *Shared Resources* are released.

7.6.4 Collaboration Roles

- *MultiResource*
  This object aggregates an entire set of resources needed (or possibly needed) by a *Resource Client*. *MultiResource* explicitly locks and unlocks the set of resources. This locking and unlocking action should be a noninterruptible critical section. If any of the aggregated *Shared Resources* is not available during the locking process,

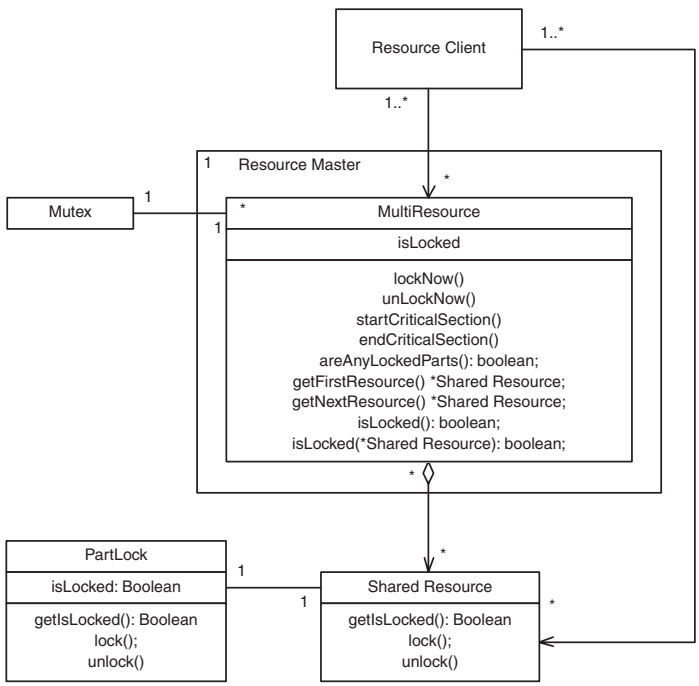


Figure 7-15: *Simultaneous Locking Pattern*

then the *MultiResource* must release all of the Shared Resources it successfully locked. *MultiResource* must define operations *startCriticalSection()* and *endCriticalSection* to prevent task switching from occurring during the locking or unlocking process. Also, *areAnyLockedParts()* returns TRUE if any of the *Shared Resources* aggregated by the *MultiResource* are still locked. For walking through the *Shared Resources*, the *MultiResource* also has the operations *getFirstResource()* and *getNextResource()*, both of which return a pointer to a *Shared Resource* (or NULL if at the end of the list) and *isLocked(*Shared Resource)*, which returns TRUE only if the referenced *Shared Resource* is currently locked by the *MultiResource.* If either unlocked or not aggregated by the *MultiResource*, then it returns FALSE. Two more operations, *lockNow()* and *unlockNow()*, simply set the *isLocked* attribute of the *MultiResource* without checking the status of the aggregated parts.

- *Mutex*
  The *Mutex* is a mutual exclusion semaphore object that associates with *MultiResource.* In this pattern the shared resources are locked for a longer duration than with the priority inheritance-based patterns. This is because *Resource Client* needs to own all the resources for the entire critical section so that the *Resource Client* never owns a resource while trying to lock another. The policy is that the *Mutex* is only locked if *all* of the required *ShareResource PartLocks* are successfully locked. *Mutex* is an OS-level mutex and signals the *Scheduler* to take care of blocking tasks that attempt to lock the *SharedResource.*
- *PartLock*
  The *PartLock* is a special mutual exclusion semaphore that associates to *Shared Resource.* This *Mutex* is queryable as to its lock status, using the *getIsLocked()* operation. This semaphore does not signal the *Scheduler* unlike the *Mutex*, because there is no need; the OS-level locking is done by the *Mutex* and not by the *PartLock.* Nevertheless, the *MultiResource* needs to be able to ascertain the locking status of all the resources before attempting to lock any of them.
- *Resource Client*
  The *Resource Client* is a user of *Shared Resource* objects. It locks potentially multiple *Shared Resources* via the *MultiResource.* The policy enforced in this pattern is that all resources used in a criti-

cal section must be locked at the same time, or the entire lock will fail. The *Resource Client* is specifically prohibited from locking one resource and later, while still owning that lock, attempting to lock another. Put another way, an attempt to lock a set of resources is only permitted if the *Resource Client* currently owns no locks at all, and if any of the requested resources are unavailable, the entire lock will fail and the *Resource Client* must wait and block on the set of resources (that is, it blocks on the mutex owned by its associated *MultiResource*).

- *ResourceMaster*
  The *ResourceMaster* orchestrates the locking and unlocking of *Mutexes* associated with *MultiResources.* Whenever a *MultiResource* locks a *Mutex,* the *ResourceMaster* searches its list of all *MultiResources* and locks any that share one of the *SharedResources.* That way, if a *Thread* tries to lock its *MultiResource* and another one owns a needed *SharedResource,* the *Thread* can block on the *Mutex* of its associated *MultiResource.* Conversely, when a *MultiResource* releases all of its *Shared Resources,* that *MultiResource* notifies the *ResourceMaster* and it tracks down all of the other *MultiResources* and sees if it can unlock them as well (it may not be able to if another *MultiResource* has locked a *SharedResource* unused by the first).
- *Shared Resource*
  A resource is a part object owned by the *MultiResource* object. In this pattern, a *Shared Resource* does not connect to a *Mutex* because it is not locked individually. As implied by its name, the same *Shared Resource* object may be an aggregated part of different *MultiResource* objects. The pattern policy is that no resource that is aggregated by one *MultiResource* is allowed to be directly locked by a *Thread,* although it may be accessed by a *Thread* to perform services. The *Shared Resource* contains operations to explicitly lock, unlock, and to query its locked status, and these simply invoke services in the associated *PartLock.*

7.6.5 Consequences

The Simultaneous Locking Pattern prevents deadlock by breaking condition 2, required for deadlock to occur—namely locking some resources while waiting for others to become available. It does this

by locking all resources needed at once and releasing them all at once. This resource management pattern can easily be used in most scheduling patterns, such as the Static Priority Pattern.

There are two primary negatives to the use of this pattern. First, priority inversion is not bounded. A higher-priority task is free to preempt and run as long as it doesn't use any currently locked resource. This pattern could be mixed in with the priority inheritance pattern to address that problem.

The second issue is that this pattern invokes some computational overhead, which may become severe in situations in which there are many shared resources. Each time a request to lock a resource is made, each of the *Shared Resources* must be locked *and* all of the other *MultiResources* must be checked to see if they aggregate any of these locked *Shared Resources.* Any *MultiResource* that shares one of the just-locked *Shared Resources* must itself be locked. On release of a lock on a particular *MultiResource,* all of its *Shared Resources* must be unlocked, and then each of the other *MultiResources* must be examined using the *areAnyLockedParts()* operation. If it returns TRUE, then that *MultiResource* must remain locked; otherwise is must be unlocked.

Another issue is that programmer/designer discipline is required not to access the *Shared Resources* without first obtaining a lock by going through the *MultiResource* mechanism. Because *Shared Resources* don't use standard OS mutexes for locking (since we don't want *Threads* blocking on them rather then the *MultiResources*), it is possible to directly access the *Shared Resource,* bypassing the locking mechanisms. This is a Bad Idea. One possible solution to enforce the locking is to propagate all of the operations from the resources to the *MultiResource,* make the operations public in the *MultiResource* and private in the *Shared Resource,* and making the *MultiResource* a friend of the *Shared Resource.* This adds some additional computational overhead, but in some languages the propagated operations could be made inline to minimize this. Alternatively, each *Shared Resource* could be told, during the locking process, who its owner is. Then on each service call, the owner would have to pass an owner ID to prove it had rights to request the service.

7.6.6 Implementation Strategies

Care must be taken that the locking of all the resources in *MultiResource.lock()* and *MultiResource.unlock()* must be done in a critical sec-

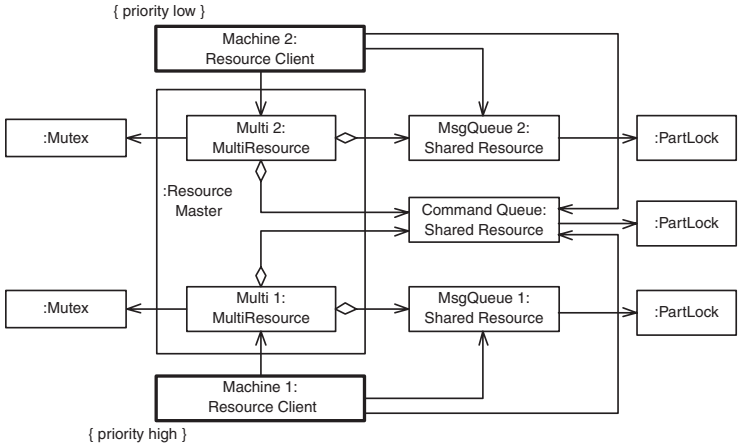
tion to prevent deadlock condition 2 from occurring. Other than that, the implementation of this pattern is straightforward.

7.6.7 Related Patterns

This pattern removes deadlock by breaking condition 2 required for deadlock. There are other approaches to avoiding deadlock. One of this is presented in the Ceiling Priority Pattern and another in the Ordered Locking Pattern, both presented in this chapter. This pattern is normally mixed with a concurrency management policy, such as the Static Priority Pattern, but other patterns can be used as well. If it is desirable to limit priority inversion, then this pattern can be mixed with the Priority Inheritance Pattern.

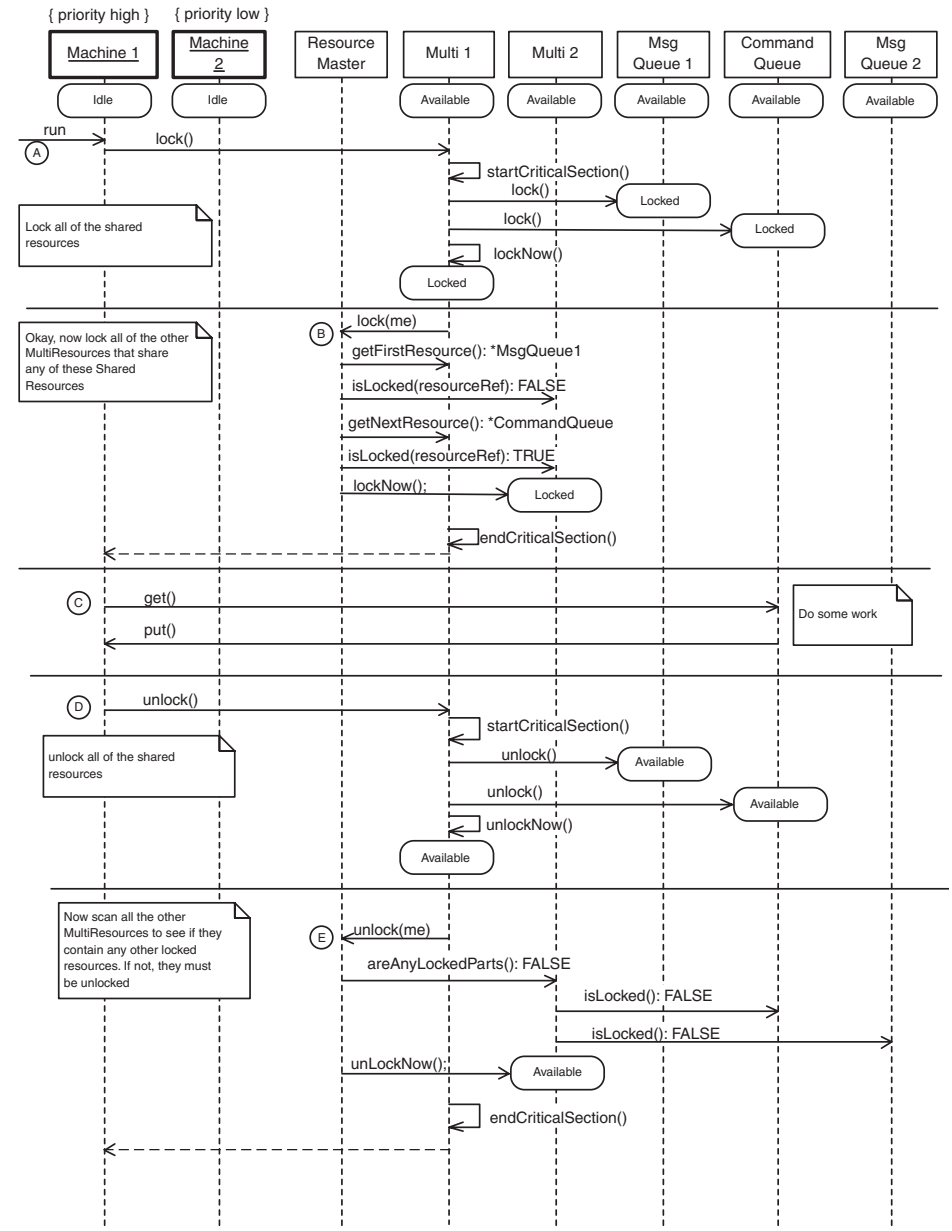
7.6.8 Sample Model

Figure 7-16a shows a simple example of the application of this pattern. Two *Concrete Threads, Machine 1* and *Machine 2,* share three resources: *MsgQueue 1* (Machine 1 only), *Command Queue* (both), and *MsgQueue 2* (Machine 2 only). To avoid the possibility of a deadlock occurring, the Simultaneous Locking Pattern is used. Two *MultiResources* (*Multi 1* and *Multi 2*) are created as composite parts of an instance of *Resource-Master.* Figure 7-16b shows the behavior when *Machine 1* locks its



a. Simultaneous Locking Pattern Example

Figure 7-16: *Simultaneous Locking Pattern (continued)*



b. Scenario

Figure 7-16: *Simultaneous Locking Pattern (continued)*

resources, does some work (moving messages from the *Command Queue* to *MsgQueue 1*), and then unlocks the resources.

What is not shown is what happens if *Machine 2* runs during the execution of the get() and put() operations, but it is clear that as soon as *Machine 2* attempts to lock its *MultiResource,* it will be blocked.

7.7 ORDERED LOCKING PATTERN

The Ordered Locking Pattern is another way to ensure that deadlock cannot occur—this time by preventing condition 4 (circular waiting) from occurring. It does this by ordering the resources and requiring that they always be accessed by any client in that specified order. If this is religiously enforced, then no circular waiting condition can ever occur.

7.7.1 Abstract

The Ordered Locking Pattern eliminates deadlock by ordering resources and enforcing a policy in which resources must be allocated only in a specific order. Unlike "normal" resource access, but similar to the Simultaneous Locking Pattern, the client must explicitly lock and release the resources, rather than doing it implicitly by merely invoking a service on a resource. This means that the potential for neglecting to unlock the resource exists.

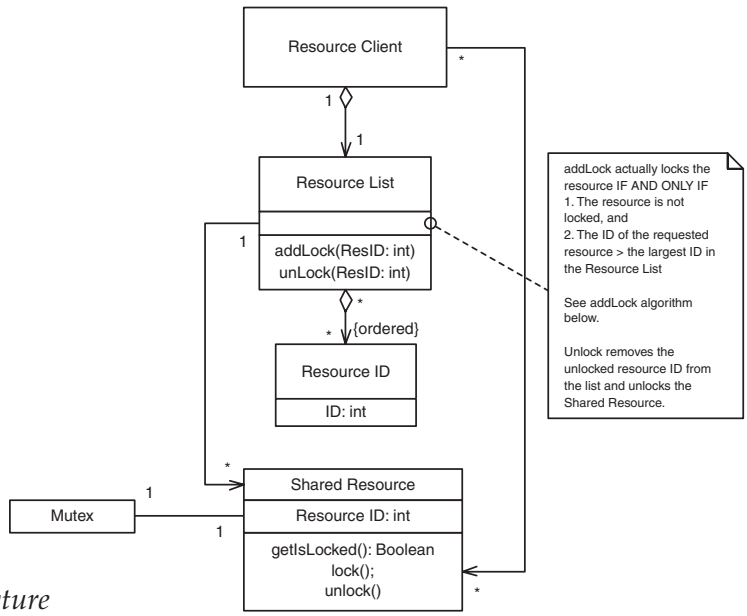
7.7.2 Problem

The Ordered Locking Pattern solely addresses the problem of deadlock elimination, as does the previous Simultaneous Locking Pattern.

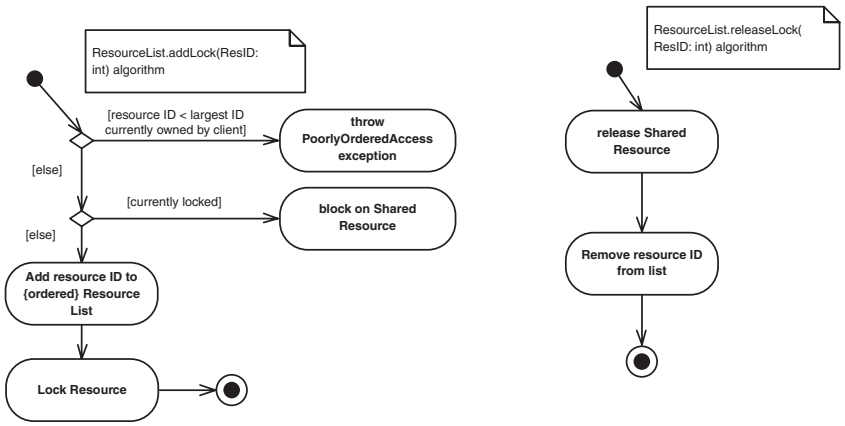
7.7.3 Pattern Structure

Figure 7-17a shows the structural part of the Ordered Locking Pattern. Each *Resource Client* aggregates a *Resource List*, which contains an ordered list of Resource IDs currently locked by the *Thread.*

Figure 7-17b uses UML activity charts to show the algorithms for locking and unlocking the resource. The basic policy of resource locking is that each resource in the entire system has a unique integer-valued identifier, and a *Thread* may *only* lock a resource whose ID is greater than that of the highest resource it currently owns. An attempt to lock a resource with a lower-valued ID than the highest-



a. Structure



b. Access Algorithms

Figure 7-17: *Ordered Locking Pattern*

valued resource you currently own causes the *Resource List* to throw a *PoorlyOrderedAccess* exception, indicating a violation of this policy. Sometimes a *Resource Client* may block on a resource that it needs because it is locked, and that is perfectly fine. But it can never happen that a circular waiting condition can occur (required for deadlock) because it would require that at least one *Resource Client* would have to block waiting for the release of a resource whose ID is lower than its highest-owned resource.[2]

7.7.4 Collaboration Roles

- *Mutex*
  The *Mutex* is a mutual exclusion semaphore object that associates with *Shared Resource.* If a Shared Resource is currently locked when requested by a *Resource Client* (via its *Resource List*), then the *Resource Client* blocks on that resource.
- *Resource Client*
  A *Resource Client* is an object (which may be «active») that owns and locks resources. It aggregates a *Resource List* to manage the locking and unlocking of those resources.
- *Resource ID*
  The *Resource ID* is a simple part objected aggregated by *Resource List*. It merely contains the ID of a corresponding *Shared Resource* currently locked by the *Thread*. When the *Shared Resource* is unlocked by the *Resource List*, its ID is removed from the list.
- *Resource List*
  The *Resource List* manages the locking and unlocking of *Shared Resources* according to the algorithm shown in Figure 7-17b. When a *Resource Client* wants to lock a resource, it makes the request of the *Resource List.* If the ID of the required resource is greater than any currently owned resource, and then if it is unlocked, the *Resource List* locks it and adds it to the list. If it is locked, then the *Thread* blocks on the resource.
- *Shared Resource*
  A resource is an object shared by one or more *Resource Client.* In this pattern, each *Shared Resource* has a unique integer-valued

2. The proof is left as an exercise for the reader.

identifier. This identifier is used to control the order in which *Shared Resources* may be locked. If a *Shared Resource* itself uses other *Shared Resources,* then it may *only* do so if the called *Shared Resource* identifiers are of higher value than its own.

7.7.5 Consequences

This pattern effectively removes the possibility of resource-based deadlocks by removing the possibility of condition 4—circular waiting. For the algorithm to work any ordering of *Shared Resources* will do provided that this ordering is global. However, some orderings are better than others and will result is less blocking overall. This may take some analysis at design time to identify the best ordering of the *Shared Resources.* As mentioned above, if *Shared Resources* are themselves *Resource Clients* (a reasonable possibility), then they should *only* invoke services of *Shared Resources* that have higher-valued IDs than they do. If they invoke a lower-valued *Shared Resource,* then they are in effect violating the ordered locking protocol by the transitive property of locking (if A locks B and then B locks C, then A is in effect locking C).

While draconian, one solution to the potential problem of transitive violation of the ordering policy is to enforce the rule that a *Shared Resource* may never invoke services or lock other *Shared Resources.* If your system design does allow such transitive locking, then each transitive path must be examined to ensure that the ordering policy is not violated. The Ordered Locking Pattern does not address the issue of bounding priority inversion as do some other patterns here.

7.7.6 Implementation Strategies

One memory-efficient implementation for *Resource List* is to use an array of integers to hold the *Resource IDs.* The array only needs to be as large as the maximum number of resources held at any one time. For an even more memory-efficient implementation (but at the cost of some computational complexity), a bit set can be used. The bit set must have the same number of bits as maximum *Resource ID* value. Setting and unsetting the bit is computationally lightweight, but

checking to see if there is a greater bit set is a little more computationally intensive.

7.7.7 Related Patterns

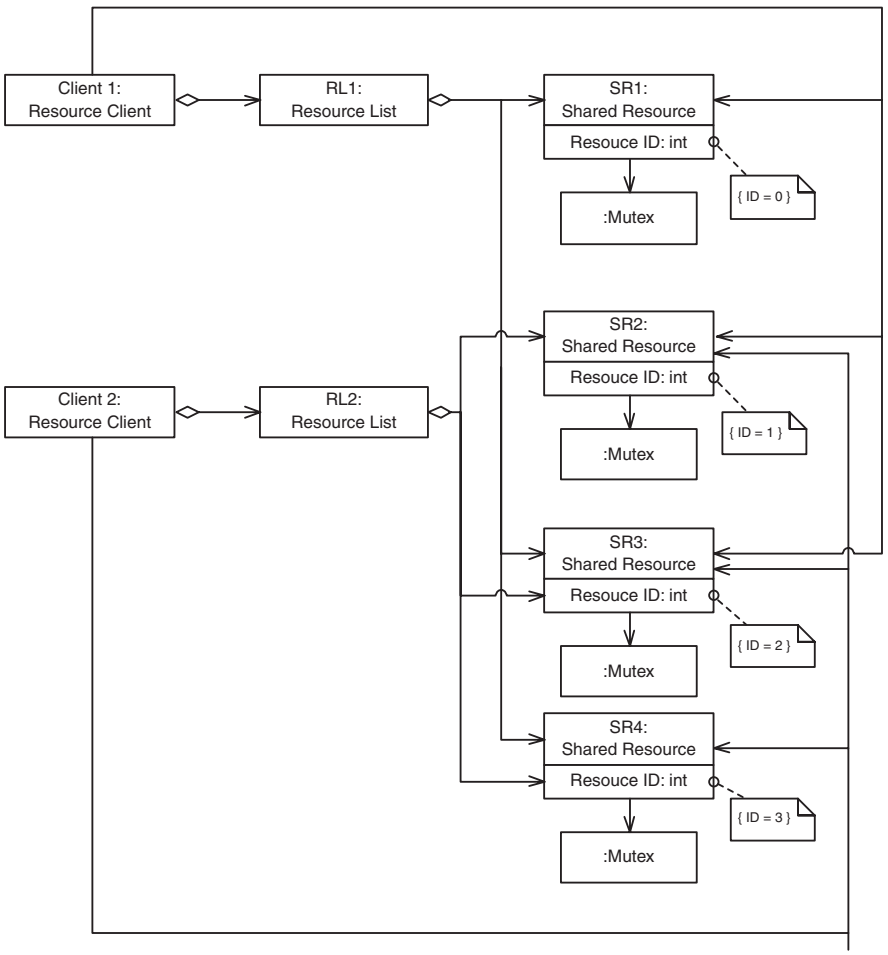
There are two other patterns here that prevent deadlock. The Simultaneous Locking Pattern locks all the needed resources in a single critical section; other *Resource Clients* that need to run can do so as long as they don't request any of the same *Shared Resources.* If a small subset of the resources need to be locked at any given time or if the sets needed for different *Resource Clients* overlap only slightly, then the Simultaneous Locking Pattern works well.

The Priority Ceiling Pattern solves the deadlock problem as well, although the algorithm is significantly more complex. For that added sophistication, the Priority Ceiling Pattern also bounds priority inversion to a single level.

7.7.8 Sample Model

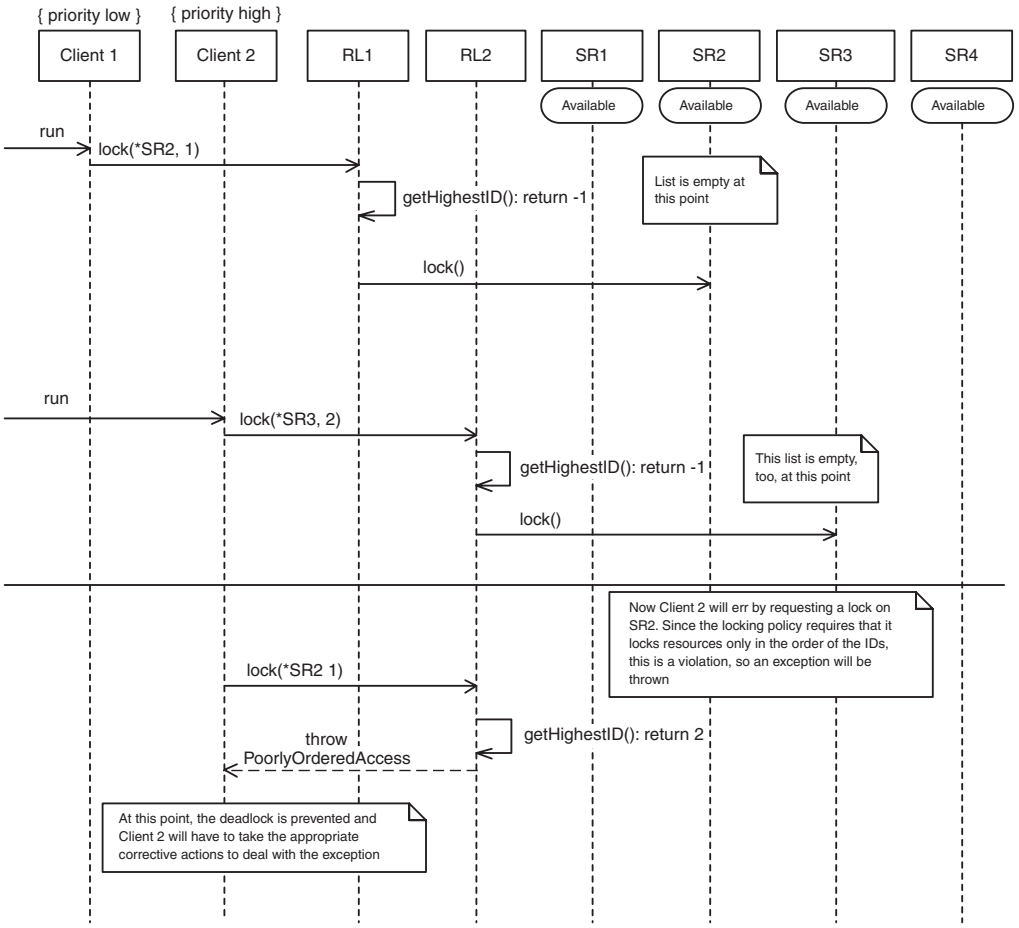
The example shown in Figure 7-18a provides a simple illustration of how the pattern works. *Client 1* uses three *Shared Resources: SR1* (ID=0), *SR3* (ID=2), and *SR4* (ID=3). *Client 2* uses three *Shared Resources: SR2* (ID=1), *SR3* (ID=2), and *SR4* (ID=3). They both, therefore, share SR2, SR3, and SR4.

Note that in the absence of some scheme to prevent deadlock (such as the use of the Ordered Locking Pattern), deadlock is easily possible in this configuration. Suppose *Client 1* ran and locked SR2, and when it was just about to lock SR3, *Client 2* (running in a higher-priority thread) preempts *Client 1. Client 2* now locks SR3 and tries to lock SR2. It cannot, of course, because it is already locked (by *Client 1*), and so it must block and allow *Client 1* to run until it releases the resource. However, now *Client 1* cannot successfully run because it needs SR3, and it is locked by *Client 2.* A classic deadlock situation. This particular scenario is not allowed with the Ordered Locking Pattern. Figure 7-18b shows what happens when this scenario is attempted.



a. Ordered Locking Pattern Example

Figure 7-18: *Ordered Locking Pattern*



b. Scenario

Figure 7-18: *Ordered Locking Pattern (continued)*

7.8 References

[1] *Response to the OMG RFP for Schedulability, Performance, and Time, Revised Submission,* Boston, MA: Object Management Group OMG Document Number: ad/2001-06-14, 2001.

[2] Klein, M., T. Ralya, B. Pollak, R. Obenza, and M. Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems,* Norwell, MA: Kluwer Academic Press, 1993.

[3] Douglass, Bruce Powel. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns,* Reading, MA: Addison-Wesley, 1999.

[4] Douglass, Bruce Powel. *Real-Time UML 2nd Edition: Developing Efficient Objects for Embedded Systems,* Boston, MA: Addison-Wesley, 2000.