CHAPTER 8

Scheduling

Multithreaded programming is a common strategy for embedded software. Partitioning an application into separate threads of execution with careful management of communications among the threads tends to simplify the design of each thread. Once partitioned, it is appropriate to consider how these threads compete for their share of processor time.

8.1 THREAD STATES

We begin by recognizing that only one thread can be running, and that the rest must be waiting their turn, as shown in Figure 8–1. The job of a *scheduler* is to force turn-taking in a manner that meets performance requirements.

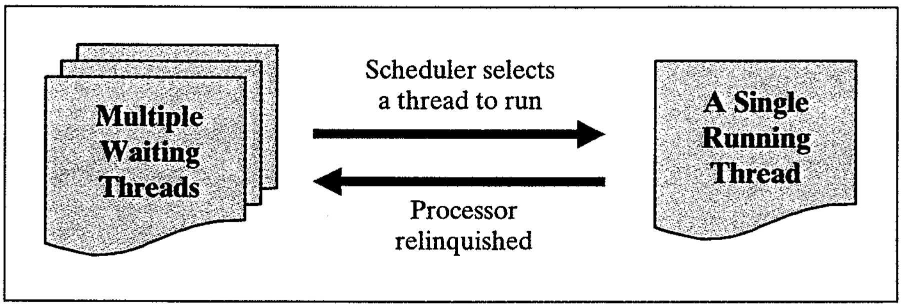

FIGURE 8–1 Scheduler selects thread to Run while others wait.

To compete for processor time, control of each thread must be given to the scheduler by making a call to the kernel. Before that call is made, the thread is unknown to the scheduler and is said to be *inactive*. As shown in Figure 8–2, threads are thus in

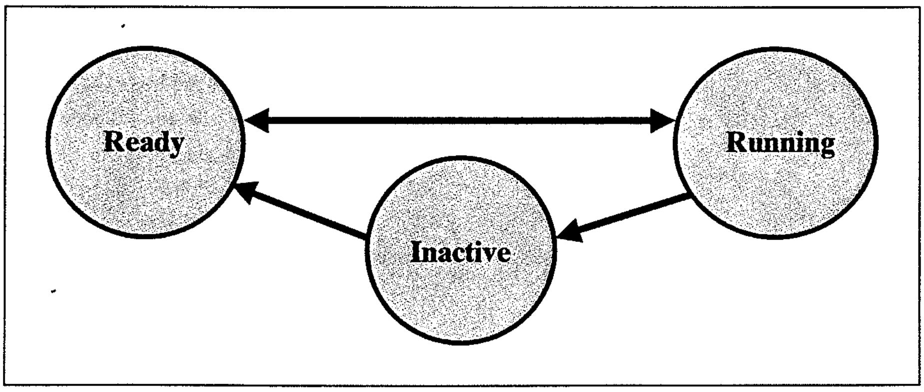

FIGURE 8–2 Initial thread-state diagram.

one of three states (*inactive*, *running*, or *ready*), with the scheduler managing transitions of threads from one state to another.

A simplistic scheduling algorithm called *time-slicing* runs each thread for the same fixed amount of time. When a thread reaches the end of its allocated time slice, the kernel simply gives control to the next thread in the sequence, repeating the sequence over and over again. While time slicing may be an appropriate strategy for regular computing, real-time systems need a kernel that is much more responsive to external events.

8.2 PENDING THREADS

To optimize processor utilization, scheduling must recognize that threads often have to wait for some event to occur or some shared resource to become available. When this happens, the scheduler should run a different thread while the first one waits. Simply moving the delayed task back into the ready state isn't appropriate, however, because the thread may be selected to run again later, only to discover that it must continue to wait, thus adding unnecessary task-switching overhead.

The solution is to introduce a fourth state called *pending*, shown in Figure 8–3. When a running thread reaches a point where it cannot proceed until some specific condition is satisfied (such as the occurrence of an external event or the release of a shared resource), another kernel call is used to move it to the pending state; the scheduler then selects and runs one of the other threads from the ready list. When the condition is satisfied, another kernel call to the scheduler moves the pending thread to the ready state.

Most threads cycle through these three states (ready, running, and pending) as long as the embedded application is running. Some threads, however, are normally inactive; they are occasionally made ready when needed, are run to completion, and are then returned to the inactive state.

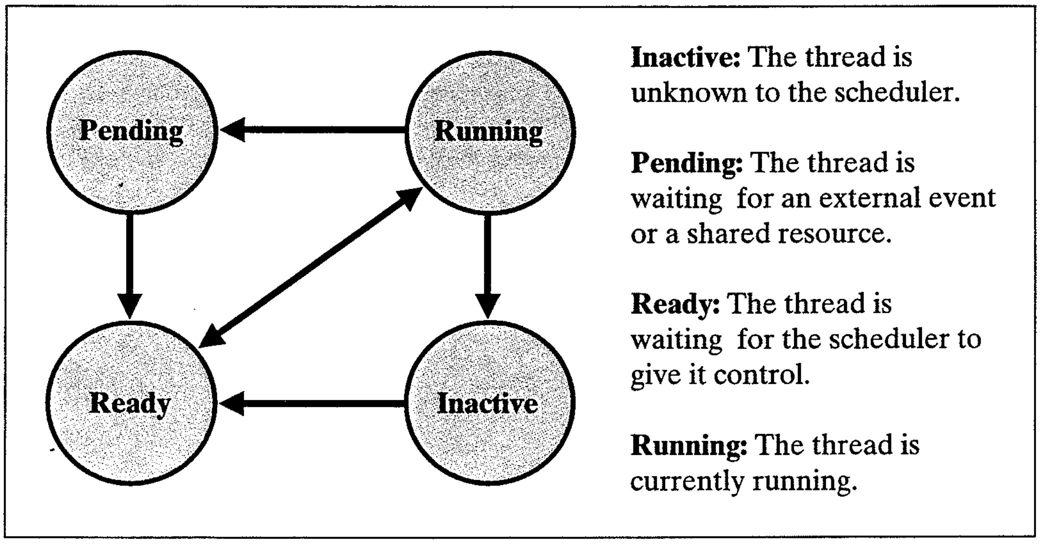

FIGURE 8–3 Revised thread-state diagram.

8.3 CONTEXT SWITCHING

A *nonpreemptive* context switch is shown in Figure 8–4. Initially, thread A is running and thread B is waiting in the ready state. Thread A makes a kernel call, and the scheduler decides to context-switch to thread B. Note that although a hardware interrupt may temporarily suspend a running thread, the interrupt routine still returns to the same thread.

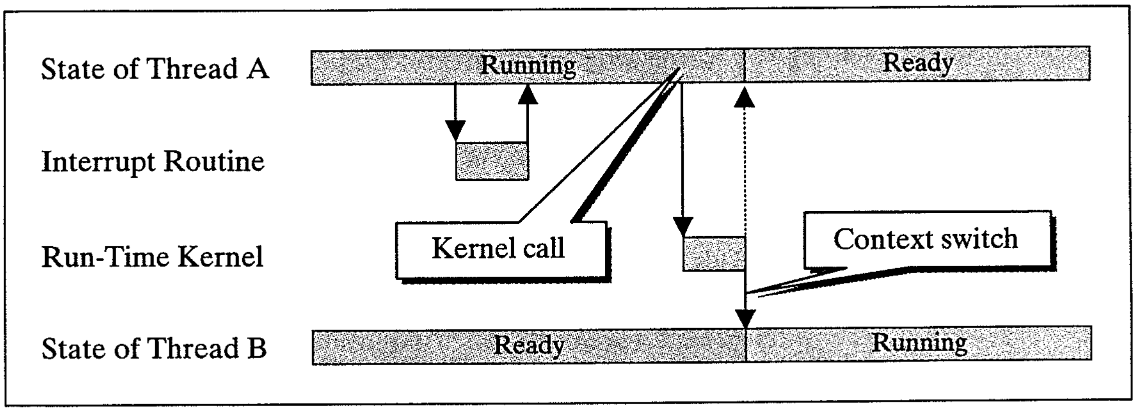

FIGURE 8–4 Nonpreemptive context switch.

A thread may explicitly call the scheduler at any time to give other threads a chance to run; such a call is known as a *yield*. The yield call simply moves the current thread to the ready state and calls the scheduler, as shown in Figure 8–5. However, there are other kinds of kernel calls that may implicitly call the scheduler:

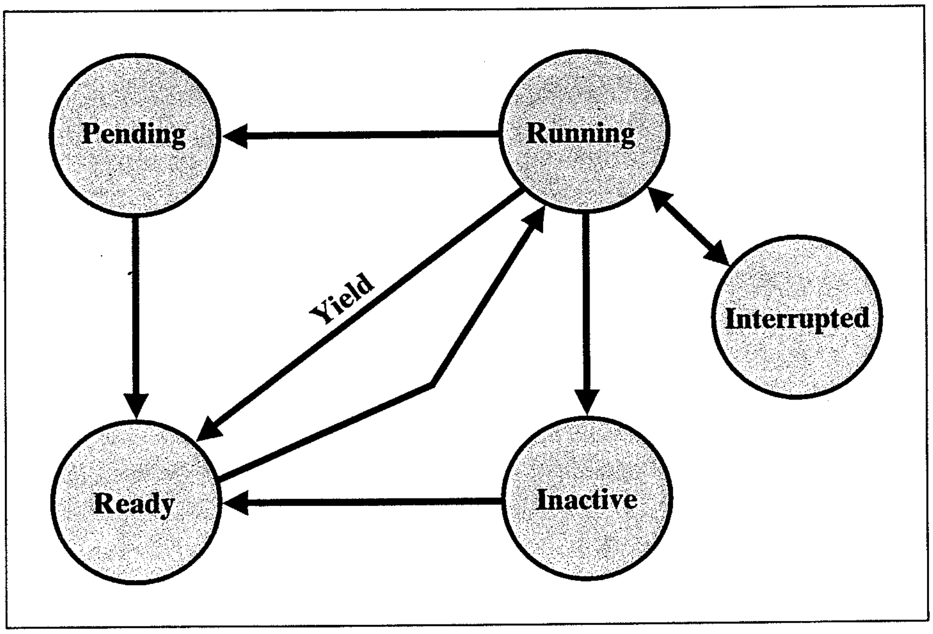

FIGURE 8–5 Thread states in a nonpreemptive kernel.

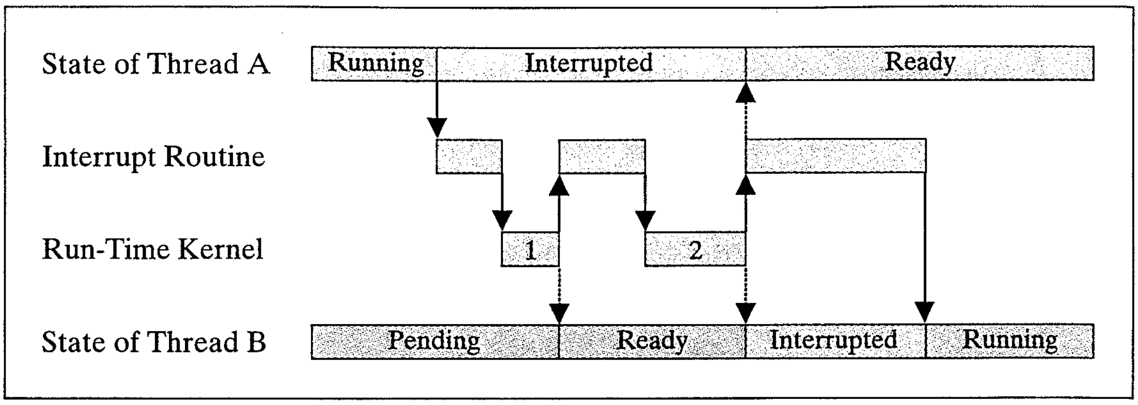

FIGURE 8–6 Interrupt-triggered context-switch in a preemptive kernel.

1. A kernel call to terminate the thread.[1]
2. A kernel call to wait for an external event to occur.
3. A kernel call to wait for a shared resource that is currently in use.
4. A kernel call to "sleep" for some specified amount of time.

Nonpreemptive systems have to make frequent kernel calls in order to achieve good response time. Threads running under a *preemptive* kernel, however, are not *required* to make kernel calls in order to yield. Context-switching within the interrupt routine allows preemptive kernels to immediately activate the thread that processes the event. This provides a drastically improved and predictable response time that is critically important for real-time systems that depend on completion of their tasks within certain deadline constraints.

For example, Figure 8–6 illustrates how a preemptive system context-switches from one thread to another when an interrupt occurs. Assume that thread B is waiting in the pending state for some external event. When that event occurs, it triggers an interrupt that makes a kernel call ('1') to release the pending thread to the ready state. Just before returning, the ISR makes a second kernel call ('2') to the scheduler to check whether or not a context-switch is required. As shown in Figures 8–5 and 8–7, both preemptive and nonpreemptive kernels allow running threads to move temporarily to an "interrupted" state. Unlike nonpreemptive kernels, however, a preemptive kernel allows the interrupted thread to move back to the ready state so that a different thread can resume when the interrupt is completed.

8.4 ROUND-ROBIN SCHEDULING

The simplest algorithm for selecting one of the ready threads to run is called round-robin scheduling. The kernel maintains a queue that holds pointers to threads that are ready to run. When the scheduler needs to select a thread to run, it merely removes the next thread from the front of the queue; threads are entered at the rear of the queue when they are ready to run.

[1] Control implicitly transfers back to the scheduler when the function that implements a thread returns; no kernel call is necessary to terminate the thread.

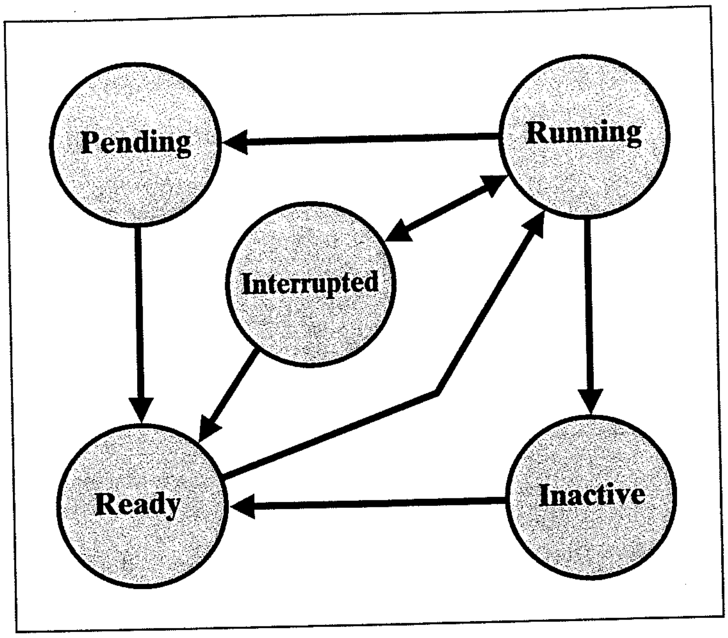

FIGURE 8–7 Thread states in a preemptive kernel.

Round-robin scheduling simply treats all threads as having equal importance. In most real-time applications, however, certain threads are more important than others. For example, a task that updates a display can afford to be delayed, but a task that modulates brake pressure in an antilock braking system must certainly be given preference over other threads.

8.5 PRIORITY-BASED SCHEDULING

An important consideration for real-time systems is *how* the scheduler selects one of the ready threads to run. The most common solution by far is to assign each thread a priority number. *Static* priorities are essentially constants that never change during the execution of the application; we'll see later, however, that *dynamic* priorities are necessary in order to make system response time fast and predictable.

8.5.1 Priority Inversion

High-priority threads are intended to have precedence over low-priority threads. However, a situation called *priority inversion* can temporarily defeat this intention, and is said to exist whenever the completion of one task is delayed by a second task of lower priority.

Priority inversion can occur when two threads use the same (shared) resource. For example, let's assume that the low-priority thread in Figure 8–8 had already acquired exclusive access to such a resource before the interrupt occurred. Even though the high-priority thread is now running, if it now attempts to acquire that same resource (kernel call 'C'), it will have to go back to the pending state and wait until the low-priority thread relinquishes it (kernel call 'D'). This simple form of priority inversion is considered *bounded* because its duration is no longer than that of the critical section where the lower priority thread owns the resource.

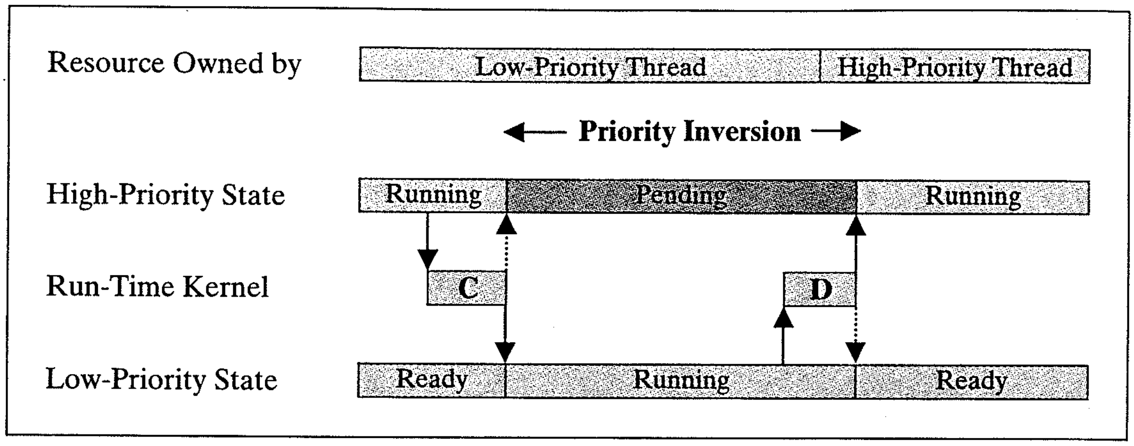

FIGURE 8–8 Example of a priority inversion.

Unbounded priority inversion occurs when a third (medium-priority) thread preempts the low-priority thread during the inversion, thus delaying the high-priority thread even more. (This is exactly what happened during the Mars Pathfinder[2] mission.) There are two well-known solutions to this particular problem: the Priority Inheritance Protocol and the Priority Ceiling Protocol. Both are based manipulating thread priorities at run time.

Consider a low-priority thread that has locked a mutex to acquire exclusive access to a shared resource. As long as this thread maintains its lock, higher-priority threads that attempt to lock the same mutex will be blocked. The objective of both protocols is to prevent the low-priority thread from being preempted by medium-priority threads until the end of the critical section during which the mutex is locked.

8.5.2 The Priority Inheritance Protocol

When a high-priority thread attempts to lock a mutex already locked by a lower-priority thread, the Priority Inheritance Protocol (PIP) temporarily raises the priority of the low-priority thread to match that of the blocked thread until the low-priority thread unlocks the mutex. The advantage of this solution is that it is transparent to the application, although it adds a moderate amount of complexity to the kernel.

Priority inversion can occur just as easily when critical sections are protected with a semaphore instead of a mutex object. For reasons of efficiency, the internal data structure of a semaphore is little more than a counter; keeping track of all the different threads that have acquired the semaphore simply isn't practical. Thus, most kernels simply do not support PIP with semaphores.

[2] The Mars Pathfinder software included three threads (among others): a high-priority bus management thread designed to run quickly and frequently, a medium-priority communications thread that ran for a much longer time, and a low-priority meteorological thread that ran infrequently. To publish its data the meteorological thread had to acquire the (shared) bus. Then the communications thread woke up and preempted the meteorological thread. Finally, the bus management thread woke up and was blocked because it couldn't acquire the bus; when it couldn't meet its deadline, an alarm went off that reinitialized the computer via a hardware reset.

8.5.3 The Priority Ceiling Protocol

What's different about the Priority Ceiling Protocol (PCP) is that the priority of the low-priority thread is raised immediately when it locks the mutex, rather than waiting for a subsequent lock attempt by a higher-priority thread.

An interesting aspect of PCP is that it can be easily implemented within the application if not already provided by the kernel. All that is required is a kernel call to raise priority before locking the mutex and another to lower priority after the mutex is released. If the kernel *does* implement PCP within the mutex code, the associated priority ceiling value is simply provided as a calling parameter when the mutex is created and kept within the mutex data structure. This approach is preferred because PCP then becomes transparent to the application except for the initial mutex creation. The disadvantage of PCP is that the priority ceiling value must be *predetermined* for use with the mutex; this value must be the highest among all the threads that attempt to lock the same mutex.

Most kernels don't support PCP with semaphores for the same reason they don't support PIP.

8.6 ASSIGNING PRIORITIES

The primary challenge in scheduling is to assign priority numbers in a manner that guarantees that the system meets real-time performance criteria. In real-time systems, faster response times are generally more desirable than slower response times. However, this doesn't always mean that we must compute every output response as fast as possible. Computing all output responses within their corresponding deadline constraints is a somewhat different objective.

In conventional multi-user systems, we usually try to keep users satisfied with system performance by running the shortest computations (tasks) first with the objective of minimizing the average overall response time. For example, consider the case of two people who submit requests at the same moment–one that needs 1 second of CPU time and another that needs 60 seconds. As shown in Figure 8–9 below, if the longer task runs first,[3] the response times will be 60 and 61 seconds and one user will be significantly delayed; however, if the short task runs first, the response times are 1 and 61 seconds and both users are satisfied.

8.6.1 Deadline-Driven Scheduling

The shortest-first policy may not be appropriate for real-time systems with response time deadlines. Associated with every task are two times: (1) the minimum time required to compute the response as determined by CPU speed and the complexity of the computation, and (2) the maximum time allowed by external constraints–i.e., the deadline.

---

[3] In practice, tasks do not run serially as in Figure 8–9; rather the CPU continuously switches back and forth, devoting a percentage of its time to each task. The example shown represents the extreme case of allocating 100% to one task and 0% to the other. Actual allocations would fall somewhere in between the two cases indicated.

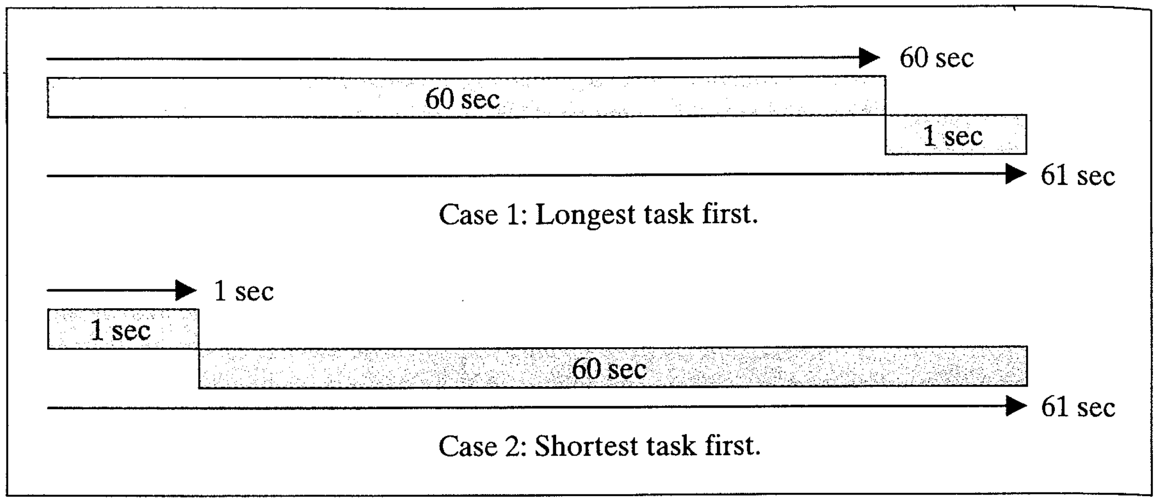

FIGURE 8–9 Scheduling to minimize average response times.

For example, consider two tasks where one requires 10 seconds of CPU time with a deadline of 35 seconds, and the other requires 20 seconds with a deadline of 25 seconds. As Figure 8–10 shows, both deadlines can only be met if the *longer* (20-second) task runs first. This form of scheduling is known as deadline-driven because it schedules the thread with the earliest deadline first.

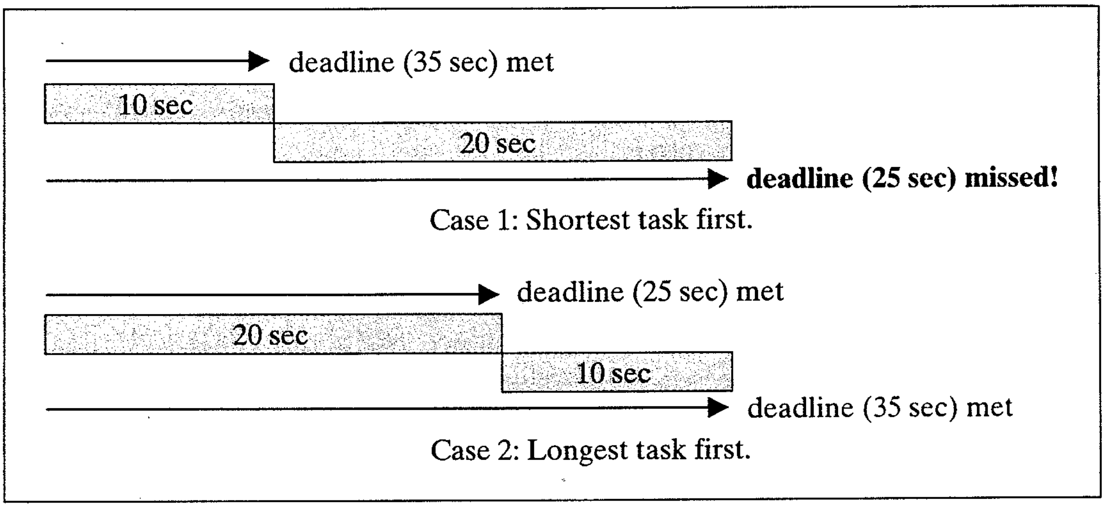

FIGURE 8–10 Scheduling to meet deadlines.

8.6.2 Rate-Monotonic Scheduling

If we restrict ourselves to tasks that are periodic with fixed-length execution times, a popular technique called Rate Monotonic Analysis[4] can be used to analytically determine whether or not a given system will meet its deadlines. A complete treatment of RMA is beyond the scope of this text.

[4] Klein et al., "A Practitioners Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems. Kluwer Academic Publishers, Boston, MA, 1993.

The primary result of using RMA is that tasks with the shortest execution *periods* be assigned the highest priorities, without regard to how critical each task might be. Such an assignment helps to insure that the faster-executing tasks meet their deadlines by deterring preemption by slower tasks.

For example, a task that processes the arrival of serial data packets will have a period that is related to the serial data rate. As the rate increases, complete packets are assembled and become ready for processing more frequently—meaning that the period between successive packets decreases and that (according to RMA) the priority of the task should thus be higher.

RMA also provides the ability to estimate processor utilization. This is a measure of how much of the processor's capability is being used, and indicates whether a faster processor is needed or whether a slower (and less expensive) one would suffice. As you might expect, as processor utilization increases, so does the probability that the tasks will not be able to meet their deadlines. A general rule of thumb is to try to keep utilization below 80%.

8.7 DEADLOCK

Deadlock is a situation in which two or more threads are blocked because they are waiting on each other's resources. It can only occur when these threads each require exclusive access to two or more shared resources simultaneously.

Consider two threads T1 and T2 that share two resources R1 and R2. Since the resources are shared, the threads use mutex objects M1 (for resource R1) and M2 (for resource R2) to protect their critical sections. Suppose that thread T1 has locked mutex M1 because it is using resource R1, and that thread T2 has locked mutex M2 because it is using R2. This corresponds to the beginning of Figure 8–11, where both threads are still active and neither is blocked.

The problem begins when each thread attempts to lock the other mutex without first unlocking the one it already owns. For example, when T2 attempts to lock M1, the kernel puts T2 into the pending state since M1 is already locked by T1. T1 continues to

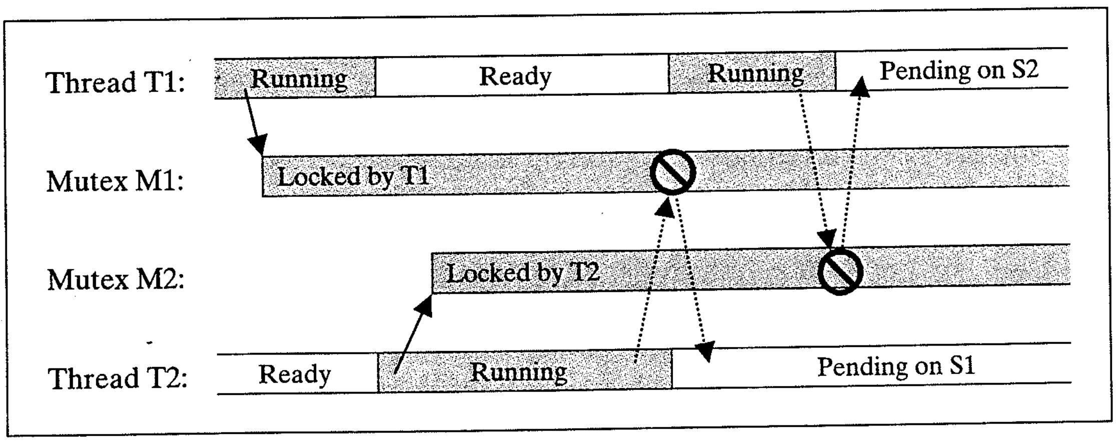

FIGURE 8–11 A sequence of events creating deadlock.

run but then attempts to lock M2; since M2 is already locked by T2, the kernel puts T1 into the pending state too. With both threads now blocked and pending, neither can complete its critical section in order to release its lock.

Our example was based on only two threads for simplicity. After a little thought, you'll soon realize that an N-way deadlock with *more* than two threads is also possible. As N increases, our ability to recognize a potential deadlock situation diminishes rapidly. Fortunately, development tools do exist that can analyze software and detect such situations automatically, but it is still the programmer who has to find a solution to the problem.

It is important to realize that deadlock simply isn't possible when critical sections are protected by turning off task-switching or by disabling interrupts. These techniques are not only effective but are preferable for short critical sections. However, a mutex (or semaphore) is the *only* reasonable way to protect long critical sections without affecting the response time of other tasks.

There are a number of different strategies for dealing with deadlocks. One common approach is to require all threads to acquire resources in the same order. However, even though the specific order is arbitrary, it often causes resources to be locked for longer than necessary. Occasionally there is no order that can be found that is convenient for all threads. When this happens, an alternative strategy is to require a thread to release *all* of its acquired resources and start over if any one resource can't be acquired. There are even algorithms for detecting when a deadlock has occurred, but breaking a deadlock once it has been detected isn't practical in most real-time systems.

8.8 WATCHDOG TIMERS

We've seen how priority inversion and deadlock can cause a real-time system to fail to meet its response-time deadlines. Although strategies exist to avoid such problems, they sometimes aren't implemented, due to expense, complexity, or human nature. Even when they are, the system may still fail because of programming errors, hardware failure, or unanticipated situations.

If continuous operation of the embedded application is critical, and if the failure is only temporary, then a watchdog timer is commonly used to provide a primitive recovery mechanism. A watchdog timer is simply a hardware counter that counts down towards zero at a fixed rate. The software is expected to periodically restart the counter so that it never actually reaches zero. If the counter *does* reach zero, a failure is assumed to have occurred. The watchdog hardware then sends a reset signal to the CPU, causing the system to reinitialize. In safety-critical applications, the watchdog hardware must also be responsible for putting everything into a safe state, since the CPU itself may have failed.

In a real-time multithreaded application, it is important that the watchdog timer verify more than just the fact that the CPU is executing instructions. It's tempting to restart the watchdog using a few lines of code added to the system timer tick's ISR. However, this interrupt and its ISR could easily continue to work properly in the presence of a deadlock or a priority inversion, and thus would *not* verify that tasks are meeting their deadlines.

Task trigger (e.g., ISR or other thread)	*Task to compute output response*

`...`
`deadline[TASK]=current_time()+LIMIT ;`
`busy[TASK]=TRUE ;`
`SemaphorePost(...) ;`
`...`
	`while (TRUE)`
	`{`
	`SemaphorePend(...) ;`
	
	`.../*Compute output response*/`
	
	`if (current_time()> deadline[TASK])`
	`failed[TASK]=TRUE ;`
	
	`busy[TASK]=FALSE ;`
	`}`

FIGURE 8–12 Monitoring the deadline of a task.

When triggered by some event, the time a task requires to compute the corresponding output response must be less than any specified deadline. It doesn't matter whether the task is executed periodically, sporadically, or infrequently. Thus, the part of our software that restarts the watchdog counter must do so only after verifying that all such deadlines have been met.

One effective software strategy is illustrated in Figure 8–12 and Figure 8–13. It associates an integer representation of time ("deadline") and two Boolean flags ("busy" and "failed") with every task. All such flags are cleared to false during system initialization. Whenever an event occurs that triggers the execution of the task, the event computes the future deadline for the task, records it in the integer, sets busy to true, and then makes the kernel call that releases the pending task. Once the task has finished computing the output response, it sets its failed flag to true if it missed its deadline, and it unconditionally clears its busy flag to false.

The watchdog software periodically checks each task, as shown in Figure 8–13. A task has missed a deadline if either its failed flag is true or its busy flag is true and the

```
while (TRUE)
    {
    int now, task ;

    sleep(... /* for less than counter period */) ;

    now = current_time() ;
    for (task = 0; task < tasks; task++)
        {
        if (failed[task]) break ;
        if (busy[task] && now > deadline[task]) break ;
        }
    if (task == tasks) restart_counter() ;
    }
```

FIGURE 8–13 Watchdog task to restart hardware counter.

current time is greater than its posted deadline. If none of the tasks has failed, the watchdog software restarts the hardware counter. These checks are relatively simple and fast to compute, and can usually be run as a relatively infrequent but high-priority task without much impact on the rest of the system.

PROBLEMS

1. Which thread *owns* the shared resource during a bounded priority inversion?
   (a) The high-priority thread
   (b) The low-priority thread

2. A bounded priority inversion *begins* when
   (a) the low-priority thread acquires the shared resource.
   (b) the high-priority thread starts to wait ("pends") for the shared resource.

3. A bounded priority inversion *ends* when
   (a) the low-priority thread releases the shared resource.
   (b) the high-priority thread releases the shared resource.

4. An *unbounded* priority inversion among three threads, one each of low, medium, and high priority, requires a resource shared by
   (a) the low- and medium-priority threads.
   (b) the low- and high-priority threads.
   (c) the medium- and high-priority threads.
   (d) all three threads.

5. Which of the following solutions to unbounded priority inversion minimizes the length of time that the priority of the lower priority thread is raised?
   (a) The Priority Inheritance Protocol
   (b) The Priority Ceiling Protocol

6. Consider a multithreaded application consisting of 3 threads. Initially, a low-priority thread is running and owns a shared resource. A high-priority thread is pending (waiting) on the same resource. Sometime before the low-priority thread releases the resource, a medium-priority thread becomes ready.

   Fill in each of the blanks below with an algebraic expression that is the sum of one or more of the following terms, as appropriate:

   N = no wait
   L = minimum time for the low-priority thread to complete its critical section
   M = minimum time for the medium-priority thread to complete its task
   H = minimum time for the high-priority thread to complete its critical section

   What is the *maximum* time that the *high*-priority thread must wait if using
   (a) Fixed-Priority Protocol? __________
   (b) Priority Ceiling Protocol? __________
   (c) Priority Inheritance Protocol? __________

   What is the *maximum* time that the *medium*-priority thread must wait if using
   (d) Fixed-Priority Protocol? __________
   (e) Priority Ceiling Protocol? __________
   (f) Priority Inheritance Protocol? __________