Real-Time Systems

Our coverage of operating-system issues thus far has focused mainly on general-purpose computing systems (for example, desktop and server systems). In this chapter, we turn our attention to real-time computing systems. The requirements of real-time systems differ from those of many of the systems we have described, largely because real-time systems must produce results within certain deadlines. In this chapter we provide an overview of real-time computer systems and describe how real-time operating systems must be constructed to meet the stringent timing requirements of these systems.

CHAPTER OBJECTIVES

- To explain the timing requirements of real-time systems.
- To distinguish between hard and soft real-time systems.
- To discuss the defining characteristics of real-time systems.
- To describe scheduling algorithms for hard real-time systems.

19.1 Overview

A **real-time system** is a computer system that requires not only that the computing results be "correct" but also that the results be produced within a specified deadline period. Results produced after the deadline has passed—even if correct—may be of no real value. To illustrate, consider an autonomous robot that delivers mail in an office complex. If its vision-control system identifies a wall *after* the robot has walked into it, despite correctly identifying the wall, the system has not met its requirement. Contrast this timing requirement with the much less strict demands of other systems. In an interactive desktop computer system, it is desirable to provide a quick response time to the interactive user, but it is not mandatory to do so. Some systems — such as a batch-processing system— may have no timing requirements whatsoever.

Real-time systems executing on traditional computer hardware are used in a wide range of applications. In addition, many real-time systems are

embedded in "specialized devices," such as ordinary home appliances (for example, microwave ovens and dishwashers), consumer digital devices (for example, cameras and MP3 players), and communication devices (for example, cellular telephones and Blackberry handheld devices). They are also present in larger entities, such as automobiles and airplanes. An **embedded system** is a computing device that is part of a larger system in which the presence of a computing device is often not obvious to the user.

To illustrate, consider an embedded system for controlling a home dishwasher. The embedded system may allow various options for scheduling the operation of the dishwasher—the water temperature, the type of cleaning (light or heavy), even a timer indicating when the dishwasher is to start. Most likely, the user of the dishwasher is unaware that there is in fact a computer embedded in the appliance. As another example, consider an embedded system controlling antilock brakes in an automobile. Each wheel in the automobile has a sensor detecting how much sliding and traction are occurring, and each sensor continually sends its data to the system controller. Taking the results from these sensors, the controller tells the braking mechanism in each wheel how much braking pressure to apply. Again, to the user (in this instance, the driver of the automobile), the presence of an embedded computer system may not be apparent. It is important to note, however, that not all embedded systems are real-time. For example, an embedded system controlling a home furnace may have no real-time requirements whatsoever.

Some real-time systems are identified as **safety-critical systems**. In a safety-critical system, incorrect operation—usually due to a missed deadline—results in some sort of "catastrophe." Examples of safety-critical systems include weapons systems, antilock brake systems, flight-management systems, and health-related embedded systems, such as pacemakers. In these scenarios, the real-time system *must* respond to events by the specified deadlines; otherwise, serious injury—or worse—might occur. However, a significant majority of embedded systems do not qualify as safety-critical, including FAX machines, microwave ovens, wristwatches, and networking devices such as switches and routers. For these devices, missing deadline requirements results in nothing more than perhaps an unhappy user.

Real-time computing is of two types: hard and soft. A **hard real-time system** has the most stringent requirements, guaranteeing that critical real-time tasks be completed within their deadlines. Safety-critical systems are typically hard real-time systems. A **soft real-time system** is less restrictive, simply providing that a critical real-time task will receive priority over other tasks and that it will retain that priority until it completes. Many commercial operating systems—as well as Linux—provide soft real-time support.

19.2 System Characteristics

In this section, we explore the characteristics of real-time systems and address issues related to designing both soft and hard real-time operating systems.

The following characteristics are typical of many real-time systems:

- Single purpose
- Small size

- Inexpensively mass-produced
- Specific timing requirements

We next examine each of these characteristics.

Unlike PCs, which are put to many uses, a real-time system typically serves only a single purpose, such as controlling antilock brakes or delivering music on an MP3 player. It is unlikely that a real-time system controlling an airliner's navigation system will also play DVDs! The design of a real-time operating system reflects its single-purpose nature and is often quite simple.

Many real-time systems exist in environments where physical space is constrained. Consider the amount of space available in a wristwatch or a microwave oven—it is considerably less than what is available in a desktop computer. As a result of space constraints, most real-time systems lack both the CPU processing power and the amount of memory available in standard desktop PCs. Whereas most contemporary desktop and server systems use 32- or 64-bit processors, many real-time systems run on 8- or 16-bit processors. Similarly, a desktop PC might have several gigabytes of physical memory, whereas a real-time system might have less than a megabyte. We refer to the **footprint** of a system as the amount of memory required to run the operating system and its applications. Because the amount of memory is limited, most real-time operating systems must have small footprints.

Next, consider where many real-time systems are implemented: They are often found in home appliances and consumer devices. Devices such as digital cameras, microwave ovens, and thermostats are mass-produced in very cost-conscious environments. Thus, the microprocessors for real-time systems must also be inexpensively mass-produced.

One technique for reducing the cost of an embedded controller is to use an alternative technique for organizing the components of the computer system. Rather than organizing the computer around the structure shown in Figure 19.1, where buses provide the interconnection mechanism to individual components, many embedded system controllers use a strategy known as **system-on-chip (SOC)**. Here, the CPU, memory (including cache), memory-

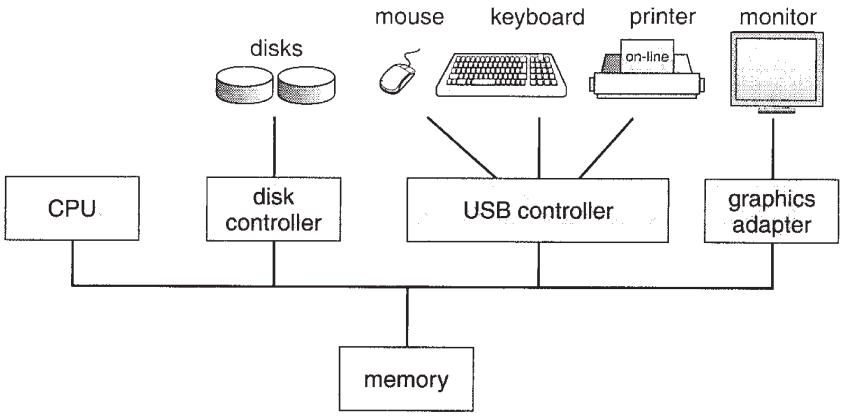

Figure 19.1 Bus-oriented organization.

management-unit (MMU), and any attached peripheral ports, such as USB ports, are contained in a single integrated circuit. The SOC strategy is typically less expensive than the bus-oriented organization of Figure 19.1.

We turn now to the final characteristic identified above for real-time systems: specific timing requirements. It is, in fact, the defining characteristic of such systems. Accordingly, the defining characteristic of both hard and soft real-time operating systems is to support the timing requirements of real-time tasks, and the remainder of this chapter focuses on this issue. Real-time operating systems meet timing requirements by using scheduling algorithms that give real-time processes the highest scheduling priorities. Furthermore, schedulers must ensure that the priority of a real-time task does not degrade over time. A second, somewhat related, technique for addressing timing requirements is by minimizing the response time to events such as interrupts.

19.3 Features of Real-Time Kernels

In this section, we discuss the features necessary for designing an operating system that supports real-time processes. Before we begin, though, let's consider what is typically *not* needed for a real-time system. We begin by examining several features provided in many of the operating systems discussed so far in this text, including Linux, UNIX, and the various versions of Windows. These systems typically provide support for the following:

- A variety of peripheral devices such as graphical displays, CD, and DVD drives
- Protection and security mechanisms
- Multiple users

Supporting these features often results in a sophisticated—and large—kernel. For example, Windows XP has over forty million lines of source code. In contrast, a typical real-time operating system usually has a very simple design, often written in thousands rather than millions of lines of source code. We would not expect these simple systems to include the features listed above.

But why don't real-time systems provide these features, which are crucial to standard desktop and server systems? There are several reasons, but three are most prominent. First, because most real-time systems serve a single purpose, they simply do not require many of the features found in a desktop PC. Consider a digital wristwatch: It obviously has no need to support a disk drive or DVD, let alone virtual memory. Furthermore, a typical real-time system does not include the notion of a user: The system simply supports a small number of tasks, which often await input from hardware devices (sensors, vision identification, and so forth). Second, the features supported by standard desktop operating systems are impossible to provide without fast processors and large amounts of memory. Both of these are unavailable in real-time systems due to space constraints, as explained earlier. In addition, many real-time systems lack sufficient space to support peripheral disk drives or graphical displays, although some systems may support file systems using nonvolatile memory (NVRAM). Third, supporting features common in standard

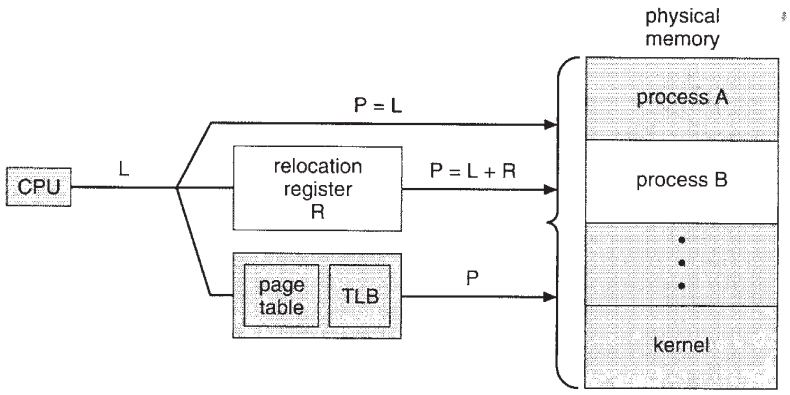

Figure 19.2 Address translation in real-time systems.

desktop computing environments would greatly increase the cost of real-time systems, which could make such systems economically impractical.

Additional considerations apply when considering virtual memory in a real-time system. Providing virtual memory features as described in Chapter 9 require the system include a memory management unit (MMU) for translating logical to physical addresses. However, MMUs typically increase the cost and power consumption of the system. In addition, the time required to translate logical addresses to physical addresses—especially in the case of a translation look-aside buffer (TLB) miss—may be prohibitive in a hard real-time environment. In the following we examine several appraoches for translating addresses in real-time systems.

Figure 19.2 illustrates three different strategies for managing address translation available to designers of real-time operating systems. In this scenario, the CPU generates logical address L that must be mapped to physical address P. The first approach is to bypass logical addresses and have the CPU generate physical addresses directly. This technique—known as **real-addressing mode**—does not employ virtual memory techniques and is effectively stating that P equals L. One problem with real-addressing mode is the absence of memory protection between processes. Real-addressing mode may also require that programmers specify the physical location where their programs are loaded into memory. However, the benefit of this approach is that the system is quite fast, as no time is spent on address translation. Real-addressing mode is quite common in embedded systems with hard real-time constraints. In fact, some real-time operating systems running on microprocessors containing an MMU actually disable the MMU to gain the performance benefit of referencing physical addresses directly.

A second strategy for translating addresses is to use an approach similar to the dynamic relocation register shown in Figure 8.4. In this scenario, a relocation register R is set to the memory location where a program is loaded. The physical address P is generated by adding the contents of the relocation register R to L. Some real-time systems configure the MMU to perform this way. The obvious benefit of this strategy is that the MMU can easily translate logical addresses to physical addresses using $P = L + R$. However, this system still suffers from a lack of memory protection between processes.

The last approach is for the real-time system to provide full virtual memory functionality as described in Chapter 9. In this instance, address translation takes place via page tables and a translation look-aside buffer, or TLB. In addition to allowing a program to be loaded at any memory location, this strategy also provides memory protection between processes. For systems without attached disk drives, demand paging and swapping may not be possible. However, systems may provide such features using NVRAM flash memory. The LynxOS and OnCore Systems are examples of real-time operating systems providing full support for virtual memory.

19.4 Implementing Real-Time Operating Systems

Keeping in mind the many possible variations, we now identify the features necessary for implementing a real-time operating system. This list is by no means absolute; some systems provide more features than we list below, while other systems provide fewer.

- Preemptive, priority-based scheduling
- Preemptive kernel
- Minimized latency

One notable feature we omit from this list is networking support. However, deciding whether to support networking protocols such as TCP/IP is simple: If the real-time system must be connected to a network, the operating system must provide networking capabilities. For example, a system that gathers real-time data and transmits it to a server must obviously include networking features. Alternatively, a self-contained embedded system requiring no interaction with other computer systems has no obvious networking requirement.

In the remainder of this section, we examine the basic requirements listed above and identify how they can be implemented in a real-time operating system.

19.4.1 Priority-Based Scheduling

The most important feature of a real-time operating system is to respond immediately to a real-time process as soon as that process requires the CPU. As a result, the scheduler for a real-time operating system must support a priority-based algorithm with preemption. Recall that priority-based scheduling algorithms assign each process a priority based on its importance; more important tasks are assigned higher priorities than those deemed less important. If the scheduler also supports preemption, a process currently running on the CPU will be preempted if a higher-priority process becomes available to run.

Preemptive, priority-based scheduling algorithms are discussed in detail in Chapter 5, where we also present examples of the soft real-time scheduling features of the Solaris, Windows XP, and Linux operating systems. Each of these systems assigns real-time processes the highest scheduling priority. For

example, Windows XP has 32 different priority levels; the highest levels—priority values 16 to 31—are reserved for real-time processes. Solaris and Linux have similar prioritization schemes.

Note, however, that providing a preemptive, priority-based scheduler only guarantees soft real-time functionality. Hard real-time systems must further guarantee that real-time tasks will be serviced in accord with their deadline requirements, and making such guarantees may require additional scheduling features. In Section 19.5, we cover scheduling algorithms appropriate for hard real-time systems.

19.4.2 Preemptive Kernels

Nonpreemptive kernels disallow preemption of a process running in kernel mode; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU. In contrast, a preemptive kernel allows the preemption of a task running in kernel mode. Designing preemptive kernels can be quite difficult; and traditional user-oriented applications such as spreadsheets, word processors, and web browsers typically do not require such quick response times. As a result, some commercial desktop operating systems—such as Windows XP—are nonpreemptive.

However, to meet the timing requirements of real-time systems—in particular, hard real-time systems—preemptive kernels are mandatory. Otherwise, a real-time task might have to wait an arbitrarily long period of time while another task was active in the kernel.

There are various strategies for making a kernel preemptible. One approach is to insert **preemption points** in long-duration system calls. A preemption point checks to see whether a high-priority process needs to be run. If so, a context switch takes place. Then, when the high-priority process terminates, the interrupted process continues with the system call. Preemption points can be placed only at *safe* locations in the kernel—that is, only where kernel data structures are not being modified. A second strategy for making a kernel preemptible is through the use of synchronization mechanisms, which we discussed in Chapter 6. With this method, the kernel can always be preemptible, because any kernel data being updated are protected from modification by the high-priority process.

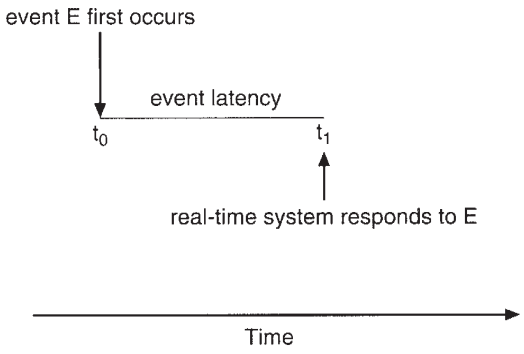

Figure 19.3 Event latency.

19.4.3 Minimizing Latency

Consider the event-driven nature of a real-time system: The system is typically waiting for an event in real time to occur. Events may arise either in software —as when a timer expires—or in hardware—as when a remote-controlled vehicle detects that it is approaching an obstruction. When an event occurs, the system must respond to and service it as quickly as possible. We refer to **event latency** as the amount of time that elapses from when an event occurs to when it is serviced (Figure 19.3).

Usually, different events have different latency requirements. For example, the latency requirement for an antilock brake system might be three to five milliseconds, meaning that from the time a wheel first detects that it is sliding, the system controlling the antilock brakes has three to five milliseconds to respond to and control the situation. Any response that takes longer might result in the automobile's veering out of control. In contrast, an embedded system controlling radar in an airliner might tolerate a latency period of several seconds.

Two types of latencies affect the performance of real-time systems:

1. Interrupt latency
2. Dispatch latency

Interrupt latency refers to the period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt. When an interrupt occurs, the operating system must first complete the instruction it is executing and determine the type of interrupt that occurred. It must then save the state of the current process before servicing the interrupt using the specific interrupt service routine (ISR). The total time required to perform these tasks is the interrupt latency (Figure 19.4). Obviously, it is crucial for real-time

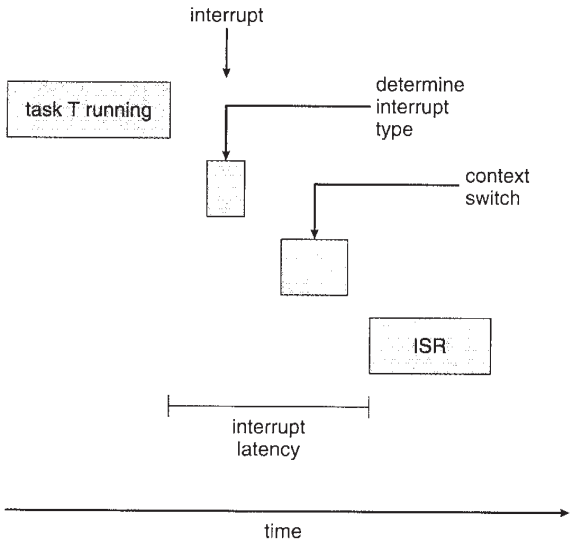

Figure 19.4 Interrupt latency.

operating systems to minimize interrupt latency to ensure that real-time tasks receive immediate attention.

One important factor contributing to interrupt latency is the amount of time interrupts may be disabled while kernel data structures are being updated. Real-time operating systems require that interrupts to be disabled for very short periods of time. However, for hard real-time systems, interrupt latency must not only be minimized, it must in fact be bounded to guarantee the deterministic behavior required of hard real-time kernels.

The amount of time required for the scheduling dispatcher to stop one process and start another is known as **dispatch latency**. Providing real-time tasks with immediate access to the CPU mandates that real-time operating systems minimize this latency. The most effective technique for keeping dispatch latency low is to provide preemptive kernels.

In Figure 19.5, we diagram the makeup of dispatch latency. The **conflict phase** of dispatch latency has two components:

1. Preemption of any process running in the kernel
2. Release by low-priority processes of resources needed by a high-priority process

As an example, in Solaris, the dispatch latency with preemption disabled is over 100 milliseconds. With preemption enabled, it is reduced to less than a millisecond.

One issue that can affect dispatch latency arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process—or a chain of lower-priority processes. As kernel

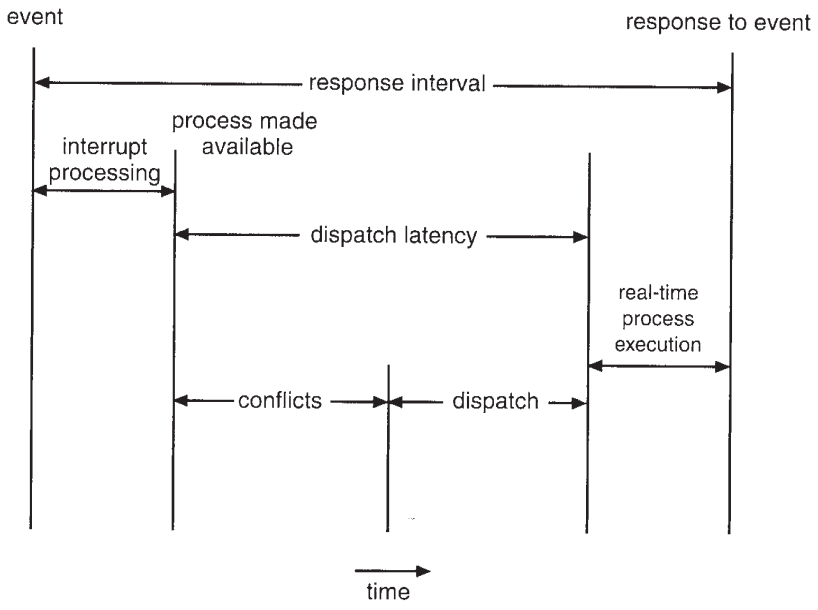

Figure 19.5 Dispatch latency.

data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource. The situation becomes more complicated if the lower-priority process is preempted in favor of another process with a higher priority. As an example, assume we have three processes, L, M, and H, whose priorities follow the order $L < M < H$. Assume that process H requires resource R, which is currently being accessed by process L. Ordinarily, process H would wait for L to finish using resource R. However, now suppose that process M becomes runnable, thereby preempting process L. Indirectly, a process with a lower priority—process M—has affected how long process H must wait for L to relinquish resource R.

This problem, known as **priority inversion**, can be solved by use of the **priority-inheritance protocol**. According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When they are finished, their priorities revert to their original values. In the example above, a priority-inheritance protocol allows process L to temporarily inherit the priority of process H, thereby preventing process M from preempting its execution. When process L has finished using resource R, it relinquishes its inherited priority from H and assumes its original priority. As resource R is now available, process H—not M—will run next.

19.5 Real-Time CPU Scheduling

Our coverage of scheduling so far has focused primarily on soft real-time systems. As mentioned, though, scheduling for such systems provides no guarantee on when a critical process will be scheduled; it guarantees only that the process will be given preference over noncritical processes. Hard real-time systems have stricter requirements. A task must be serviced by its deadline; service after the deadline has expired is the same as no service at all.

We now consider scheduling for hard real-time systems. Before we proceed with the details of the individual schedulers, however, we must define certain characteristics of the processes that are to be scheduled. First, the processes are considered **periodic**. That is, they require the CPU at constant intervals (periods). Each periodic process has a fixed processing time t once it acquires the CPU, a deadline d when it must be serviced by the CPU, and a period p. The relationship of the processing time, the deadline, and the period can be expressed as $0 \leq t \leq d \leq p$. The **rate** of a periodic task is $1/p$. Figure 19.6 illustrates the execution of a periodic process over time. Schedulers can take advantage of this relationship and assign priorities according to the deadline or rate requirements of a periodic process.

What is unusual about this form of scheduling is that a process may have to announce its deadline requirements to the scheduler. Then, using a technique known as an **admission-control** algorithm, the scheduler either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible if it cannot guarantee that the task will be serviced by its deadline.

In the following sections, we explore scheduling algorithms that address the deadline requirements of hard real-time systems.

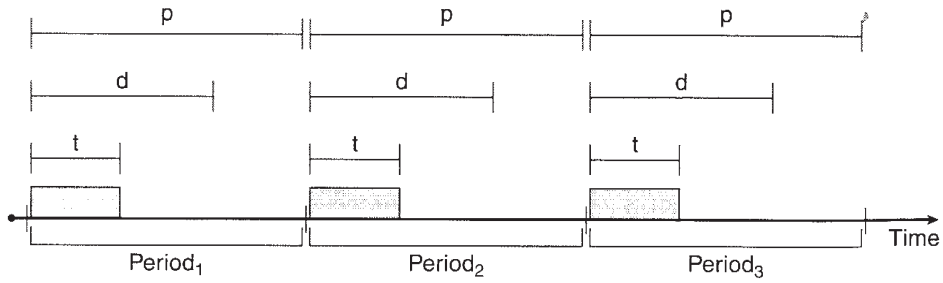

Figure 19.6 Periodic task.

19.5.1 Rate-Monotonic Scheduling

The **rate-monotonic** scheduling algorithm schedules periodic tasks using a static priority policy with preemption. If a lower-priority process is running and a higher-priority process becomes available to run, it will preempt the lower-priority process. Upon entering the system, each periodic task is assigned a priority inversely based on its period: The shorter the period, the higher the priority; the longer the period, the lower the priority. The rationale behind this policy is to assign a higher priority to tasks that require the CPU more often. Furthermore, rate-monotonic scheduling assumes that the processing time of a periodic process is the same for each CPU burst. That is, every time a process acquires the CPU, the duration of its CPU burst is the same.

Let's consider an example. We have two processes P_1 and P_2. The periods for P_1 and P_2 are 50 and 100, respectively—that is, $p_1 = 50$ and $p_2 = 100$. The processing times are $t_1 = 20$ for P_1 and $t_2 = 35$ for P_2. The deadline for each process requires that it complete its CPU burst by the start of its next period.

We must first ask ourselves whether it is possible to schedule these tasks so that each meets its deadlines. If we measure the CPU utilization of a process P_i as the ratio of its burst to its period—t_i/p_i—the CPU utilization of P_1 is $20/50 = 0.40$ and that of P_2 is $35/100 = 0.35$, for a total CPU utilization of 75 percent. Therefore, it seems we can schedule these tasks in such a way that both meet their deadlines and still leave the CPU with available cycles.

First, suppose we assign P_2 a higher priority than P_1. The execution of P_1 and P_2 is shown in Figure 19.7. As we can see, P_2 starts execution first and completes at time 35. At this point, P_1 starts; it completes its CPU burst at time 55. However, the first deadline for P_1 was at time 50, so the scheduler has caused P_1 to miss its deadline.

Now suppose we use rate-monotonic scheduling, in which we assign P_1 a higher priority than P_2, since the period of P_1 is shorter than that of P_2.

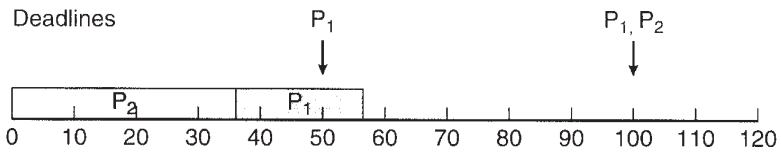

Figure 19.7 Scheduling of tasks when P_2 has a higher priority than P_1.

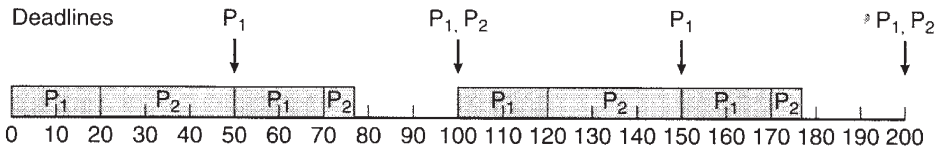

Figure 19.8 Rate-monotonic scheduling.

The execution of these processes is shown in Figure 19.8. P_1 starts first and completes its CPU burst at time 20, thereby meeting its first deadline. P_2 starts running at this point and runs until time 50. At this time, it is preempted by P_1, although it still has 5 milliseconds remaining in its CPU burst. P_1 completes its CPU burst at time 70, at which point the scheduler resumes P_2. P_2 completes its CPU burst at time 75, also meeting its first deadline. The system is idle until time 100, when P_1 is scheduled again.

Rate-monotonic scheduling is considered optimal in the sense that if a set of processes cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that assigns static priorities. Let's next examine a set of processes that cannot be scheduled using the rate-monotonic algorithm. Assume that process P_1 has a period of $p_1 = 50$ and a CPU burst of $t_1 = 25$. For P_2, the corresponding values are $p_2 = 80$ and $t_2 = 35$. Rate-monotonic scheduling would assign process P_1 a higher priority, as it has the shorter period. The total CPU utilization of the two processes is $(25/50)+(35/80) = 0.94$, and it therefore seems logical that the two processes could be scheduled and still leave the CPU with 6 percent available time. The Gantt chart showing the scheduling of processes P_1 and P_2 is depicted in Figure 19.9. Initially, P_1 runs until it completes its CPU burst at time 25. Process P_2 then begins running and runs until time 50, when it is preempted by P_1. At this point, P_2 still has 10 milliseconds remaining in its CPU burst. Process P_1 runs until time 75; however, P_2 misses the deadline for completion of its CPU burst at time 80.

Despite being optimal, then, rate-monotonic scheduling has a limitation: CPU utilization is bounded, and it is not always possible to fully maximize CPU resources. The worst-case CPU utilization for scheduling N processes is

$$2(2^{1/n} - 1).$$

With one process in the system, CPU utilization is 100 percent; but it falls to approximately 69 percent as the number of processes approaches infinity. With two processes, CPU utilization is bounded at about 83 percent. Combined CPU utilization for the two processes scheduled in Figures 19.7 and 19.8 is 75 percent; and therefore, the rate-monotonic scheduling algorithm is guaranteed

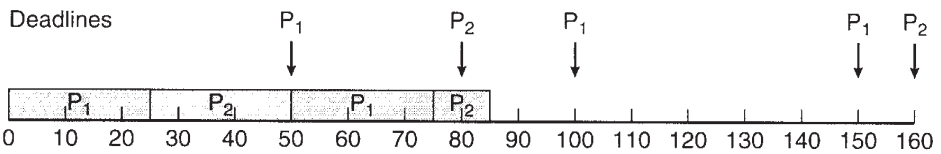

Figure 19.9 Missing deadlines with rate-monotonic scheduling.

to schedule them so that they can meet their deadlines. For the two processes scheduled in Figure 19.9, combined CPU utilization is approximately 94 percent; therefore, rate-monotonic scheduling cannot guarantee that they can be scheduled so that they meet their deadlines.

19.5.2 Earliest-Deadline-First Scheduling

Earliest-deadline-first (EDF) scheduling dynamically assigns priorities according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority. Under the EDF policy, when a process becomes runnable, it must announce its deadline requirements to the system. Priorities may have to be adjusted to reflect the deadline of the newly runnable process. Note how this differs from rate-monotonic scheduling, where priorities are fixed.

To illustrate EDF scheduling, we again schedule the processes shown in Figure 19.9, which failed to meet deadline requirements under rate-monotonic scheduling. Recall that P_1 has values of $p_1 = 50$ and $t_1 = 25$ and that P_2 has values $p_2 = 80$ and $t_2 = 35$. The EDF scheduling of these processes is shown in Figure 19.10. Process P_1 has the earliest deadline, so its initial priority is higher than that of process P_2. Process P_2 begins running at the end of the CPU burst for P_1. However, whereas rate-monotonic scheduling allows P_1 to preempt P_2 at the beginning of its next period at time 50, EDF scheduling allows process P_2 to continue running. P_2 now has a higher priority than P_1 because its next deadline (at time 80) is earlier than that of P_1 (at time 100). Thus, both P_1 and P_2 have met their first deadlines. Process P_1 again begins running at time 60 and completes its second CPU burst at time 85, also meeting its second deadline at time 100. P_2 begins running at this point, only to be preempted by P_1 at the start of its next period at time 100. P_2 is preempted because P_1 has an earlier deadline (time 150) than P_2 (time 160). At time 125, P_1 completes its CPU burst and P_2 resumes execution, finishing at time 145 and meeting its deadline as well. The system is idle until time 150, when P_1 is scheduled to run once again.

Unlike the rate-monotonic algorithm, EDF scheduling does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst. The only requirement is that a process announce its deadline to the scheduler when it becomes runnable. The appeal of EDF scheduling is that it is theoretically optimal—theoretically, it can schedule processes so that each process can meet its deadline requirements and CPU utilization will be 100 percent. In practice, however, it is impossible to achieve this level of CPU utilization due to the cost of context switching between processes and interrupt handling.

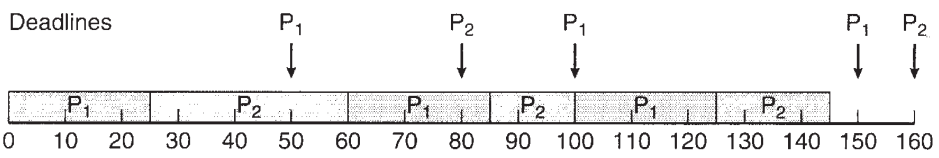

Figure 19.10 Earliest-deadline-first scheduling.

19.5.3 Proportional Share Scheduling

Proportional share schedulers operate by allocating T shares among all applications. An application can receive N shares of time, thus ensuring that the application will have N/T of the total processor time. As an example, assume that there is a total of $T = 100$ shares to be divided among three processes, A, B, and C. A is assigned 50 shares, B is assigned 15 shares, and C is assigned 20 shares. This scheme ensures that A will have 50 percent of total processor time, B will have 15 percent, and C will have 20 percent.

Proportional share schedulers must work in conjunction with an admission control policy to guarantee that an application receives its allocated shares of time. An admission control policy will only admit a client requesting a particular number of shares if there are sufficient shares available. In our current example, we have allocated $50 + 15 + 20 = 75$ shares of the total of 100 shares. If a new process D requested 30 shares, the admission controller would deny D entry into the system.

19.5.4 Pthread Scheduling

The POSIX standard also provides extensions for real-time computing—POSIX.1b. In this section, we cover some of the POSIX Pthread API related to scheduling real-time threads. Pthreads defines two scheduling classes for real-time threads:

- SCHED_FIFO
- SCHED_RR

SCHED_FIFO schedules threads according to a first-come, first-served policy using a FIFO queue as outlined in Section 5.3.1. However, there is no time slicing among threads of equal priority. Therefore, the highest-priority real-time thread at the front of the FIFO queue will be granted the CPU until it terminates or blocks. SCHED_RR (for round-robin) is similar to SCHED_FIFO except that it provides time slicing among threads of equal priority. Pthreads provides an additional scheduling class—SCHED_OTHER—but its implementation is undefined and system specific; it may behave differently on different systems.

The Pthread API specifies the following two functions for getting and setting the scheduling policy:

- `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
- `pthread_attr_getsched_policy(pthread_attr_t *attr, int policy)`

The first parameter to both functions is a pointer to the set of attributes for the thread. The second parameter is either a pointer to an integer that is set to the current scheduling policy (for `pthread_attr_getsched_policy()`) or an integer value—SCHED_FIFO, SCHED_RR, or SCHED_OTHER—for the `pthread_attr_getsched_policy()` function. Both functions return non-zero values if an error occurs.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
   int i, policy;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;

   /* get the default attributes */
   pthread_attr_init(&attr);

   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
      fprintf(stderr, "Unable to get policy.\n");
   else {
      if (policy == SCHED_OTHER)
       printf("SCHED_OTHER\n");
      else if (policy == SCHED_RR)
       printf("SCHED_RR\n");
      else if (policy == SCHED_FIFO)
       printf("SCHED_FIFO\n");
   }

   /* set the scheduling policy - FIFO, RR, or OTHER */
   if (pthread_attr_setschedpolicy(&attr, SCHED_OTHER) != 0)
      fprintf(stderr, "Unable to set policy.\n");

   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i],&attr,runner,NULL);

   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */

   pthread_exit(0);
}
```

Figure 19.11 Pthread scheduling API.

In Figure 19.11, we illustrate a Pthread program using this API. This program first determines the current scheduling policy followed by setting the scheduling algorithm to SCHED_OTHER.

19.6 VxWorks 5.x

In this section, we describe VxWorks, a popular real-time operating system providing hard real-time support. VxWorks, commercially developed by Wind River Systems, is widely used in automobiles, consumer and industrial devices, and networking equipment such as switches and routers. VxWorks is also used to control the two rovers—*Spirit* and *Opportunity*—that began exploring the planet Mars in 2004.

The organization of VxWorks is shown in Figure 19.12. VxWorks is centered around the *Wind* microkernel. Recall from our discussion in Section 2.7.3 that microkernels are designed so that the operating-system kernel provides a bare minimum of features; additional utilities, such as networking, file systems, and graphics, are provided in libraries outside of the kernel. This approach offers many benefits, including minimizing the size of the kernel—a desirable feature for an embedded system requiring a small footprint.

The Wind microkernel supports the following basic features:

- **Processes**. The Wind microkernel provides support for individual processes and threads (using the Pthread API). However, similar to Linux, VxWorks does not distinguish between processes and threads, instead referring to both as **tasks**.

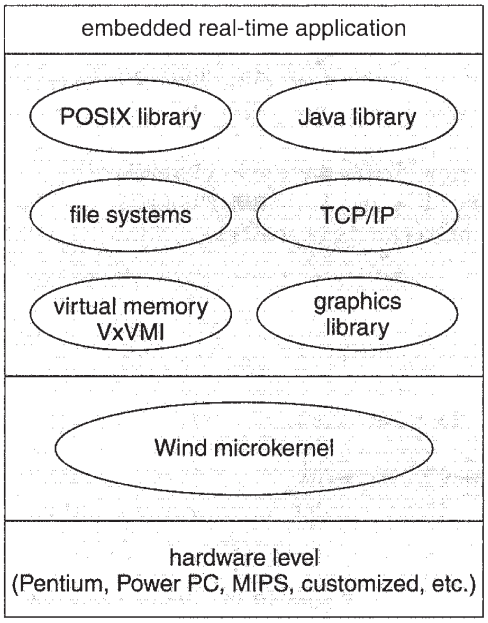

Figure 19.12 The organization of VxWorks.

REAL-TIME LINUX

The Linux operating system is being used increasingly in real-time environments. We have already covered its soft real-time scheduling features (Section 5.6.3), whereby real-time tasks are assigned the highest priority in the system. Additional features in the 2.6 release of the kernel make Linux increasingly suitable for embedded systems. These features include a fully preemptive kernel and a more efficient scheduling algorithm, which runs in $O(1)$ time regardless of the number of tasks active in the system. The 2.6 release also makes it easier to port Linux to different hardware architectures by dividing the kernel into modular components.

Another strategy for integrating Linux into real-time environments involves combining the Linux operating system with a small real-time kernel, thereby providing a system that acts as both a general-purpose and a real-time system. This is the approach taken by the RTLinux operating system. In RTLinux, the standard Linux kernel runs as a task in a small real-time operating system. The real-time kernel handles all interrupts—directing each interrupt to a handler in the standard kernel or to an interrupt handler in the real-time kernel. Furthermore, RTLinux prevents the standard Linux kernel from ever disabling interrupts, thus ensuring that it cannot add latency to the real-time system. RTLinux also provides different scheduling policies, including rate-monotonic scheduling (Section 19.5.1) and earliest-deadline-first scheduling (Section 19.5.2).

- **Scheduling**. Wind provides two separate scheduling models: preemptive and nonpreemptive round-robin scheduling with 256 different priority levels. The scheduler also supports the POSIX API for real-time threads covered in Section 19.5.4.
- **Interrupts**. The Wind microkernel also manages interrupts. To support hard real-time requirements, interrupt and dispatch latency times are bounded.
- **Interprocess communication**. The Wind microkernel provides both shared memory and message passing as mechanisms for communication between separate tasks. Wind also allows tasks to communicate using a technique known as **pipes**—a mechanism that behaves in the same way as a FIFO queue but allows tasks to communicate by writing to a special file, the pipe. To protect data shared by separate tasks, VxWorks provides semaphores and mutex locks with a priority inheritance protocol to prevent priority inversion.

Outside the microkernel, VxWorks includes several component libraries that provide support for POSIX, Java, TCP/IP networking, and the like. All components are optional, allowing the designer of an embedded system to customize the system according to its specific needs. For example, if networking is not required, the TCP/IP library can be excluded from the image of the operating system. Such a strategy allows the operating-system designer to

include only required features, thereby minimizing the size—or footprint—of the operating system.

VxWorks takes an interesting approach to memory management, supporting two levels of virtual memory. The first level, which is quite simple, allows control of the cache on a per-page basis. This policy enables an application to specify certain pages as non-cacheable. When data are being shared by separate tasks running on a multiprocessor architecture, it is possible that shared data can reside in separate caches local to individual processors. Unless an architecture supports a cache-coherency policy to ensure that the same data residing in two caches will not be different, such shared data should not be cached and should instead reside only in main memory so that all tasks maintain a consistent view of the data.

The second level of virtual memory requires the optional virtual memory component VxVMI (Figure 19.12), along with processor support for a memory management unit (MMU). By loading this optional component on systems with an MMU, VxWorks allows a task to mark certain data areas as *private*. A data area marked as private may only be accessed by the task it belongs to. Furthermore, VxWorks allows pages containing kernel code along with the interrupt vector to be declared as read-only. This is useful, as VxWorks does not distinguish between user and kernel modes; all applications run in kernel mode, giving an application access to the entire address space of the system.

19.7 Summary

A real-time system is a computer system requiring that results arrive within a deadline period; results arriving after the deadline has passed are useless. Many real-time systems are embedded in consumer and industrial devices. There are two types of real-time systems: soft and hard real-time systems. Soft real-time systems are the least restrictive, assigning real-time tasks higher scheduling priority than other tasks. Hard real-time systems must guarantee that real-time tasks are serviced within their deadline periods. In addition to strict timing requirements, real-time systems can further be characterized as having only a single purpose and running on small, inexpensive devices.

To meet timing requirements, real-time operating systems must employ various techniques. The scheduler for a real-time operating system must support a priority-based algorithm with preemption. Furthermore, the operating system must allow tasks running in the kernel to be preempted in favor of higher-priority real-time tasks. Real-time operating systems also address specific timing issues by minimizing both interrupt and dispatch latency.

Real-time scheduling algorithms include rate-monotonic and earliest-deadline-first scheduling. Rate-monotonic scheduling assigns tasks that require the CPU more often a higher priority than tasks that require the CPU less often. Earliest-deadline-first scheduling assigns priority according to upcoming deadlines—the earlier the deadline, the higher the priority. Proportional share scheduling uses a technique of dividing up processor time into shares and assigning each process a number of shares, thus guaranteeing each process its proportional share of CPU time. The Pthread API provides various features for scheduling real-time threads as well.

Exercises

19.1 Identify whether hard or soft real-time scheduling is more appropriate in the following environments:

a. Thermostat in a household

b. Control system for a nuclear power plant

c. Fuel economy system in an automobile

d. Landing system in a jet airliner

19.2 Discuss ways in which the priority inversion problem could be addressed in a real-time system. Also discuss whether the solutions could be implemented within the context of a proportional share scheduler.

19.3 The Linux 2.6 kernel can be built with no virtual memory system. Explain how this feature may appeal to designers of real-time systems.

19.4 Under what circumstances is rate-monotonic scheduling inferior to earliest-deadline-first scheduling in meeting the deadlines associated with processes?

19.5 Consider two processes, P_1 and P_2, where $p_1 = 50$, $t_1 = 25$, $p_2 = 75$, and $t_2 = 30$.

a. Can these two processes be scheduled using rate-monotonic scheduling? Illustrate your answer using a Gantt chart.

b. Illustrate the scheduling of these two processes using earliest-deadline-first (EDF) scheduling.

19.6 What are the various components of interrupt and dispatch latency?

19.7 Explain why interrupt and dispatch latency times must be bounded in a hard real-time system.

Bibliographical Notes

The scheduling algorithms for hard real-time systems, such as rate monotonic scheduling and earliest-deadline-first scheduling, were presented in Liu and Layland [1973]. Other scheduling algorithms and extensions to previous algorithms were presented in Jensen et al. [1985], Lehoczky et al. [1989], Audsley et al. [1991], Mok [1983], and Stoica et al. [1996]. Mok [1983] described a dynamic priority-assignment algorithm called least-laxity-first scheduling. Stoica et al. [1996] analyzed the proportional share algorithm. Useful information regarding various popular operating systems used in embedded systems can be obtained from http://rtlinux.org, http://windriver.com, and http://qnx.com. Future directions and important research issues in the field of embedded systems were discussed in a research article by Stankovic [1996].