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-

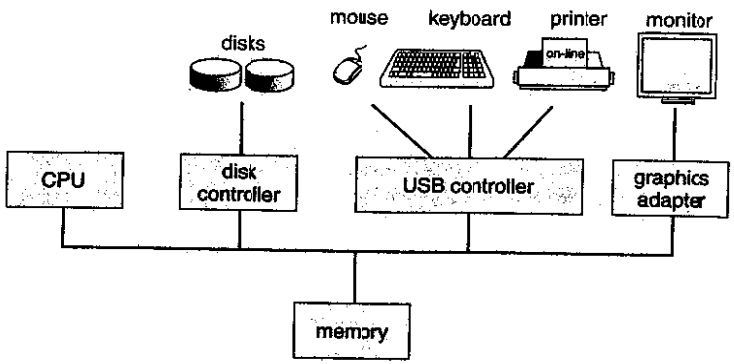


Figure 19.1 Bus-oriented organization.

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

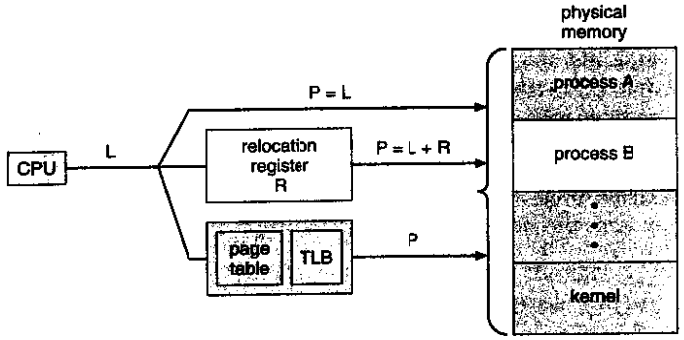


Figure 19.2 Address translation in real-time systems.

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 approaches 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.

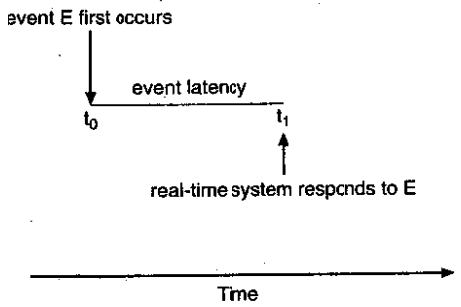


Figure 19.3 Event latency.

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

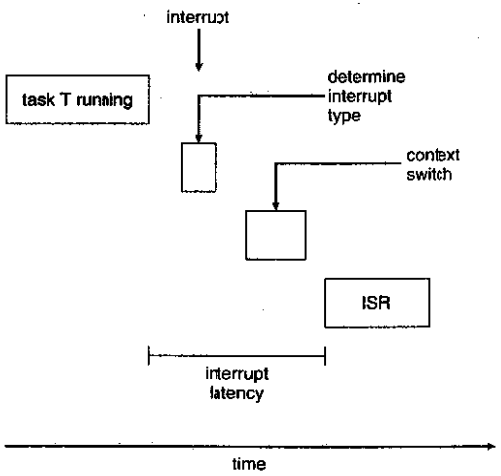


Figure 19.4 Interrupt latency.

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

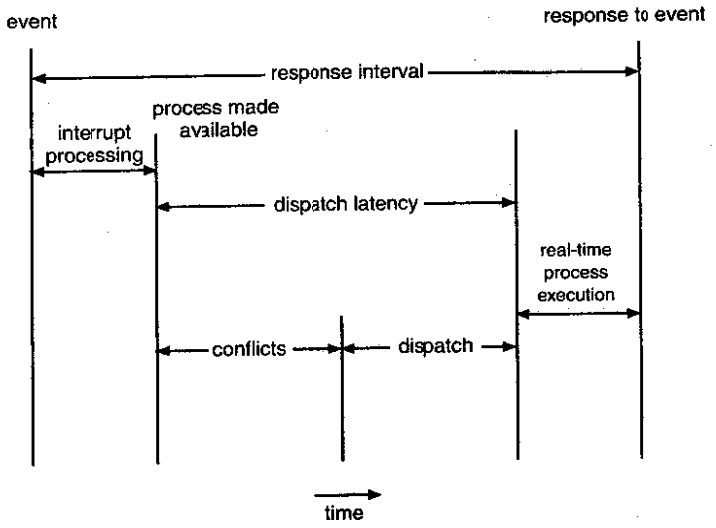


Figure 19.5 Dispatch latency.

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.

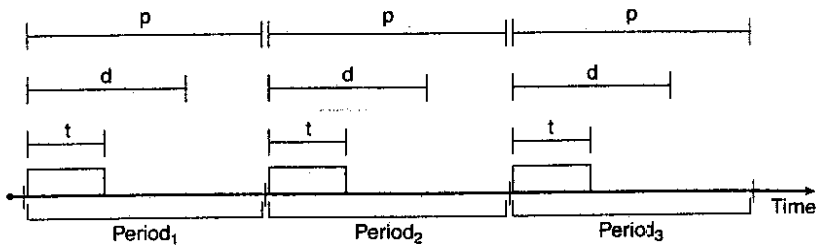


Figure 19.6 Periodic task.

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.

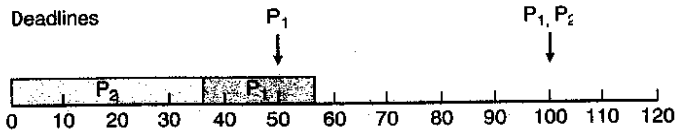


Figure 19.7 Scheduling of tasks when P_2 has a higher priority than P_1.

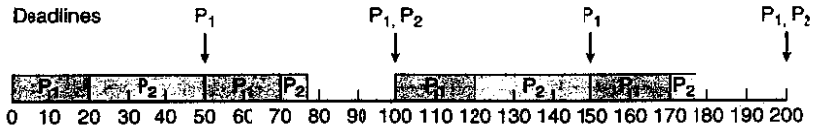


Figure 19.8 Rate-monotonic scheduling.

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

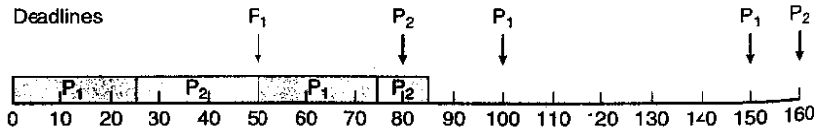


Figure 19.9 Missing deadlines with rate-monotonic scheduling.

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.

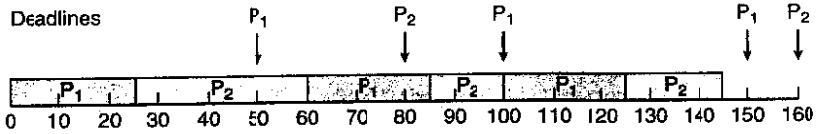


Figure 19.10 Earliest-deadline-first scheduling.

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 queueas outlined in Section5.3.1. 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 Pthreac 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**.

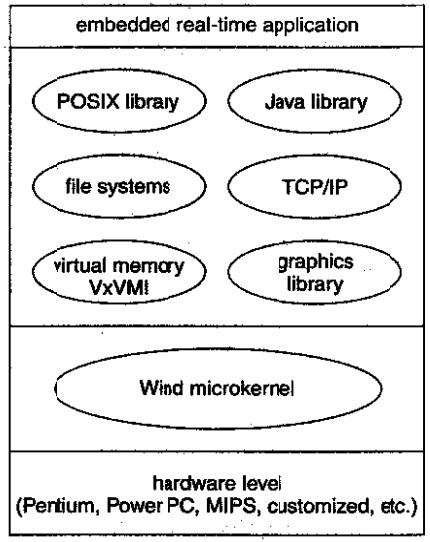


Figure 19.12 The organization of VxWorks.

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.

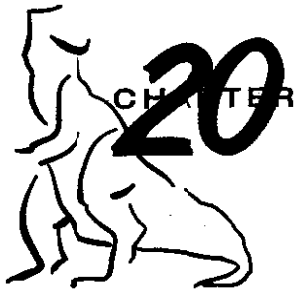
19.6 What are the various components of interrupt and dispatch latency?

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].

Multimedia Systems



In earlier chapters, we generally concerned ourselves with how operating systems handle conventional data, such as text files, programs, binaries, word-processing documents, and spreadsheets. However, operating systems may have to handle other kinds of data as well. A recent trend in technology is the incorporation of **multimedia data** into computer systems Multimedia data consist of continuous-media (audio and video) data as well as conventional files. Continuous-media data differ from conventional data in that continuous-media data—such as frames of video—must be delivered (streamed) according to certain time restrictions (for example, 30 frames per second). In this chapter, we explore the demands of continuous-media data. We also discuss in more detail how such data differ from conventional data and how these differences affect the design of operating systems that support the requirements of multimedia systems.

CHAPTER OBJECTIVES

- To identify the characteristics of multimedia data.
- To examine several algorithms used to compress multimedia data.
- To explore the operating-system requirements of multimedia data, including CPU and disk scheduling and network management.

20.1 What Is Multimedia?

The term *multimedia* describes a wide range of applications that are in popular use today. These include audio and video files such as MP3 audio files, DVD movies, and short video clips of movie previews or news stories downloaded over the Internet. Multimedia applications also include live webcasts (broadcast over the World Wide Web) of speeches or sporting events and even live webcams that allow a viewer in Manhattan to observe customers at a cafe in Paris. Multimedia applications need not be either audio or video; rather, a multimedia application often includes a combination of both. For example, a movie may consist of separate audio and video tracks.

Nor must multimedia applications be delivered only to desktop personal computers. Increasingly, they are being directed toward smaller devices, including personal digital assistants (PDAs) and cellular telephones. For example, a stock trader may have stock quotes delivered in real time to her PDA.

In this section, we explore several characteristics of multimedia systems and examine how multimedia files can be delivered from a server to a client system. We also look at common standards for representing multimedia video and audio files.

20.1.1 Media Delivery

Multimedia data are stored in the file system just like any other data. The major difference between a regular file and a multimedia file is that the multimedia file must be accessed at a specific rate, whereas accessing the regular file requires no special timing. Let's use video as an example of what we mean by "rate." Video is represented by a series of images, formally known as **frames**, that are displayed in rapid succession. The faster the frames are displayed, the smoother the video appears. In general, a rate of 24 to 30 frames per second is necessary for video to appear smooth to human eyes. (The eye retains the image of each frame for a short time after it has been presented, a characteristic known as **persistence of vision**. A rate of 24 to 30 frames per second is fast enough to appear continuous.) A rate lower than 24 frames per second will result in a choppy-looking presentation. The video file must be accessed from the file system at a rate consistent with the rate at which the video is being displayed. We refer to data with associated rate requirements as **continuous-media data**.

Multimedia data may be delivered to a client either from the local file system or from a remote server. When the data are delivered from the local file system, we refer to the delivery as **local playback**. Examples include watching a DVD on a laptop computer or listening to an MP3 audio file on a handheld MP3 player. In these cases, the data comprise a regular file that is stored on the local file system and played back (that is, viewed or listened to) from that system.

Multimedia files may also be stored on a remote server and delivered to a client across a network using a technique known as **streaming**. A client may be a personal computer or a smaller device such as a handheld computer, PDA, or cellular telephone. Data from live continuous media—such as live webcams—are also streamed from a server to clients.

There are two types of streaming techniques: progressive download and real-time streaming. With a **progressive download**, a media file containing audio or video is downloaded and stored on the clients local file system. As the file is being downloaded, the client is able to play back the media file without having to wait for the file to be downloaded in its entirety. Because the media file is ultimately stored on the client system, progressive download is most useful for relatively small media files, such as short video clips.

Real-time streaming differs from progressive download in that the media file is streamed to the client but is only played—and not stored—by the client. Because the media file is not stored on the client system, real-time streaming is preferable to progressive download for media files that might be too large for storage on the system, such as long videos and Internet radio and TV broadcasts.

Both progressive download and real-time streaming may allow a client to move to different points in the stream, just as you can use the fast-forward and rewind operations on a VCR controller to move to different points in the VCR tape. For example, we could move to the end of a 5-minute streaming video or replay a certain section of a movie clip. The ability to move around within the media stream is known as **random access**.

Two types of real-time streaming are available: live streaming and on-demand streaming. **Live streaming** is used to deliver an event, such as a concert or a lecture, live as it is actually occurring. A radio program broadcast over the Internet is an example of a live real-time stream. In fact, one of the authors of this text regularly listens to a favorite radio station from Vermont while at his home in Utah as it is streamed live over the Internet. Live real-time streaming is also used for applications such as live webcams and video conferencing. Due to its live delivery, this type of real-time streaming does not allow clients random access to different points in the media stream. In addition, live delivery means that a client who wishes to view (or listen to) a particular live stream already in progress will "join" the session "late," thereby missing earlier portions of the stream. The same thing happens with a live TV or radio broadcast. If you start watching the 7:00 P.M. news at 7:10 P.M., you will have missed the first 10 minutes of the broadcast.

On-demand streaming is used to deliver media streams such as full-length movies and archived lectures. The difference between live and on-demand streaming is that on-demand streaming does not take place as the event is occurring. Thus, for example, whereas watching a live stream is like watching a news broadcast on TV, watching an on-demand stream is like viewing a movie on a DVD player at some convenient time—there is no notion of arriving late. Depending on the type of on-demand streaming, a client may or may not have random access to the stream.

Examples of well-known streaming media products include RealPlayer, Apple QuickTime, and Windows Media Player. These products include both servers that stream the media and client media players that are used for playback.

20.1.2 Characteristics of Multimedia Systems

The demands of multimedia systems are unlike the demands of traditional applications. In general, multimedia systems may have the following characteristics:

1. Multimedia files can be quite large. For example, a 100-minute MPEG-1 video file requires approximately 1.125 GB of storage space; 100 minutes of high-definition television (HDTV) requires approximately 15 GB of storage. A server storing hundreds or thousands of digital video files may thus require several terabytes of storage.
2. Continuous media may require very high data rates. Consider digital video, in which a frame of color video is displayed at a resolution of 800×600. If we use 24 bits to represent the color of each pixel (which allows us to have 2^{24}, or roughly 16 million, different colors), a single

frame requires 800 × 600 × 24 = 11,520,000 bits of data. If the frames are displayed at a rate of 30 frames per second, a bandwidth in excess of 345 Mbps is required.

3. Multimedia applications are sensitive to timing delays during playback. Once a continuous-media file is delivered to a client, delivery must continue at a certain rate during playback of the media; otherwise, the listener or viewer will be subjected to pauses during the presentation.

20.1.3 Operating-System Issues

For a computer system to deliver continuous-media data, it must guarantee the specific rate and timing requirements—also known as **quality of service**, or QoS, requirements—of continuous media.

Providing these QoS guarantees affects several components in a computer system and influences such operating-system issues as CPU scheduling, disk scheduling, and network management. Specific examples include the following:

1. Compression and decoding may require significant CPU processing.
2. Multimedia tasks must be scheduled with certain priorities to ensure meeting the deadline requirements of continuous media.
3. Similarly, file systems must be efficient to meet the rate requirements of continuous media.
4. Network protocols must support bandwidth requirements while minimizing delay and jitter.

In later sections, we explore these and several other issues related to QoS. First, however, we provide an overview of various techniques for compressing multimedia data. As suggested above, compression makes significant demands on the CPU.

20.2 Compression

Because of the size and rate requirements of multimedia systems, multimedia files are often compressed from their original form to a much smaller form. Once a file has been compressed, it takes up less space for storage and can be delivered to a client more quickly. Compression is particularly important when the content is being streamed across a network connection. In discussing file compression, we often refer to the **compression ratio**, which is the ratio of the original file size to the size of the compressed file. For example, an 800-KB file that is compressed to 100 KB has a compression ratio of 8:1.

Once a file has been compressed (**encoded**), it must be decompressed (**decoded**) before it can be accessed. A feature of the algorithm used to compress the file affects the later decompression. Compression algorithms are classified as either **lossy** or **lossless**. With lossy compression, some of the original data are lost when the file is decoded, whereas lossless compression ensures that the compressed file can always be restored back to its original form. In general, lossy techniques provide much higher compression ratios. Obviously, though, only certain types of data can tolerate lossy compression—namely, images, audio, and video. Lossy compression algorithms often work by eliminating certain data, such as very high or low frequencies that a human ear cannot detect. Some lossy compression algorithms used on video operate by storing only the differences between successive frames. Lossless algorithms are used for compressing text files, such as computer programs (for example, **zipping** files), because we want to restore these compressed files to their original state.

A number of different lossy compression schemes for continuous-media data are commercially available. In this section, we cover one used by the Moving Picture Experts Group, better known as MPEG.

MPEG refers to a set of file formats and compression standards for digital video. Because digital video often contains an audio portion as well, each of the standards is divided into three layers. Layers 3 and 2 apply to the audio and video portions of the media file. Layer 1 is known as the **systems layer** and contains timing information to allow the MPEG player to multiplex the audio and video portions so that they are synchronized during playback. There are three major MPEG standards: MPEG-1, MPEG-2, and MPEG-4.

MPEG-1 is used for digital video and its associated audio stream. The resolution of MPEG-1 is 352 × 240 at 30 frames per second with a bit rate of up to 1.5 Mbps. This provides a quality slightly lower than that of conventional VCR videos. MP3 audio files (a popular medium for storing music) use the audio layer (layer 3) of MPEG-1. For video, MPEG-1 can achieve a compression ratio of up to 200:1, although in practice compression ratios are much lower. Because MPEG-1 does not require high data rates, it is often used to download short video clips over the Internet.

MPEG-2 provides better quality than MPEG-1 and is used for compressing DVD movies and digital television (including high-definition television, or HDTV). MPEG-2 identifies a number of **levels** and **profiles** of video compression. The level refers to the resolution of the video; the profile characterizes the video's quality. In general, the higher the level of resolution and the better the quality of the video, the higher the required data rate. Typical bit rates for MPEG-2 encoded files are 1.5 Mbps to 15 Mbps. Because MPEG-2 requires higher rates, it is often unsuitable for delivery of video across a network and is generally used for local playback.

MPEG-4 is the most recent of the standards and is used to transmit audio, video, and graphics, including two-dimensional and three-dimensional animation layers. Animation makes it possible for end users to interact with the file during playback. For example, a potential home buyer can download an MPEG-4 file and take a virtual tour through a home she is considering purchasing, moving from room to room as she chooses. Another appealing feature of MPEG-4 is that it provides a scalable level of quality, allowing delivery over relatively slow network connections such as 56-Kbps modems or over high-speed local area networks with rates of several megabits per second. Furthermore, by providing a scalable level of quality, MPEG-4 audio and video files can be delivered to wireless devices, including handheld computers, PDAs, and cell phones.

All three MPEG standards discussed here perform lossy compression to achieve high compression ratios. The fundamental idea behind MPEG compression is to store the differences between successive frames. We do not cover further details of how MPEG performs compression but rather encourage

the interested reader to consult the bibliographical notes at the end of this chapter.

20.3 Requirements of Multimedia Kernels

As a result of the characteristics described in Section 20.1.2, multimedia applications often require levels of service from the operating system that differ from the requirements of traditional applications, such as word processors, compilers, and spreadsheets. Timing and rate requirements are perhaps the issues of foremost concern, as the playback of audio and video data demands that the data be delivered within a certain deadline and at a continuous, fixed rate. Traditional applications typically do not have such time and rate constraints.

Tasks that request data at constant intervals—or **periods**—are known as **periodic processes**. For example, an MPEG-1 video might require a rate of 30 frames per second during playback. Maintaining this rate requires that a frame be delivered approximately every $1/30^{th}$ or 3.34 hundredths of a second. To put this in the context of deadlines, let's assume that frame F_j succeeds frame F_i in the video playback and that frame F_i was displayed at time T_0. The deadline for displaying frame F_j is 3.34 hundredths of a second after time T_0. If the operating system is unable to display the frame by this deadline, the frame will be omitted from the stream.

As mentioned earlier, rate requirements and deadlines are known as quality of service (QoS) requirements. There are three QoS levels:

1. **Best-effort service**. The system makes a best-effort attempt to satisfy the requirements; however, no guarantees are made.
2. **Soft QoS**. This level treats different types of traffic in different ways, giving certain traffic streams higher priority than other streams. However, just as with best-effort service, no guarantees are made.
3. **Hard QoS**. The quality-of-service requirements are guaranteed.

Traditional operating systems—the systems we have discussed in this text so far—typically provide only best-effort service and rely on **overprovisioning**; that is, they simply assume that the total amount of resources available will tend to be larger than a worst-case workload would demand. If demand exceeds resource capacity, manual intervention must take place, and a process (or several processes) must be removed from the system. However next-generation multimedia systems cannot make such assumptions. These systems must provide continuous-media applications with the guarantees made possible by hard QoS. Therefore, in the remainder of this discussion, when we refer to QoS, we mean hard QoS. Next, we explore various techniques that enable multimedia systems to provide such service-level guarantees.

There are a number of parameters defining QoS for multimedia applications, including the following:

- **Throughput**. Throughput is the total amount of work done during a certain interval. For multimedia applications, throughput is the required data rate.

- **Delay**. Delay refers to the elapsed time from when a request is first submitted to when the desired result is produced. For example, the time from when a client requests a media stream to when the stream is delivered is the delay.
- **Jitter**. Jitter is related to delay; but whereas delay refers to the time a client must wait to receive a stream, jitter refers to delays that occur during playback of the stream. Certain multimedia applications, such as on-demand real-time streaming, can tolerate this sort of delay. Jitter is generally considered unacceptable for continuous-media applications, however, because it may mean long pauses—or lost frames—during playback. Clients can often compensate for jitter by buffering a certain amount of data—say, 5 seconds' worth—before beginning playback.
- **Reliability**. Reliability refers to how errors are handled during transmission and processing of continuous media. Errors may occur due to lost packets in the network or processing delays by the CPU. In these—and other—scenarios, errors cannot be corrected, since packets typically arrive too late to be useful.

The quality of service may be **negotiated** between the client and the server. For example, continuous-media data may be compressed at different levels of quality: the higher the quality, the higher the required data rate. A client may negotiate a specific data rate with a server, thus agreeing to a certain level of quality during playback. Furthermore, many media players allow the client to configure the player according to the speed of the client's connection to the network. This allows a client to receive a streaming service at a data rate specific to a particular connection. Thus, the client is negotiating quality of service with the content provider.

To provide QoS guarantees, operating systems often use **admission control**, which is simply the practice of admitting a request for service only if the server has sufficient resources to satisfy the request. We see admission control quite often in our everyday lives. For example, a movie theater only admits as many customers as it has seats in the theater. (There are also many situations in everyday life where admission control is not practiced but would be desirable!) If no admission control policy is used in a multimedia environment, the demands on the system might become so great that the system becomes unable to meet its QoS guarantees.

In Chapter 6, we discussed using semaphores as a method of implementing a simple admission control policy. In this scenario, there exist a finite number of non-shareable resources. When a resource is requested, we will only grant the request if there are sufficient resources available; otherwise the requesting process is forced to wait until a resource becomes available. Semaphores may be used to implement an admission control policy by first initializing a semaphore to the number of resources available. Every request for a resource is made through a `wait()` operation on the semaphore; a resource is released with an invocation of `signal()` on the semaphore. Once all resources are in use, subsequent calls to `wait()` block until there is a corresponding `signal()`.

A common technique for implementing admission control is to use **resource reservations**. For example, resources on a file server may include the CPU, memory, file system, devices, and network (Figure 20.1). Note that

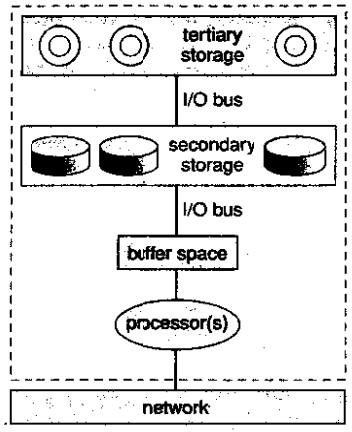


Figure 20.1 Resources on a file server.

resources may be either exclusive or shared and that there may be either single or multiple instances of each resource type. To use a resource, a client must make a reservation request for the resource in advance. If the request cannot be granted, the reservation is denied. An admission control scheme assigns a **resource manager** to each type of resource. Requests for resources have associated QoS requirements—for example, required data rates. When a request for a resource arrives, the resource manager determines if the resource can meet the QoS demands of the request. If not, the request may be rejected, or a lower level of QoS may be negotiated between the client and the server. If the request is accepted, the resource manager reserves the resources for the requesting client, thus assuring the client the desired QoS requirements. In Section 20.7.2, we examine the admission control algorithm used to ensure QoS guarantees in the CineBlitz multimedia storage server.

20.4 CPU Scheduling

In Chapter 19, which covered real-time systems, we distinguished between **soft real-time systems** and **hard real-time systems**. Soft real-time systems simply give scheduling priority to critical processes. A soft real-time system ensures that a critical process will be given preference over a noncritical process but provides no guarantee as to when the critical process will be scheduled. A typical requirement of continuous media, however, is that data must be delivered to a client by a certain deadline; data that do not arrive by the deadline are unusable. Multimedia systems thus require hard real-time scheduling to ensure that a critical task will be serviced within a guaranteed period of time.

Another scheduling issue concerns whether a scheduling algorithm uses **static priority** or **dynamic priority**—a distinction we first discussed in Chapter 5. The difference between the two is that the priority of a process will remain unchanged if the scheduler assigns it a static priority. Scheduling algorithms

that assign dynamic priorities allow priorities to change over time. Most operating systems use dynamic priorities when scheduling non-real-time tasks with the intention of giving higher priority to interactive processes. However, when scheduling real-time tasks, most systems assign static priorities, as the design of the scheduler is less complex.

Several of the real-time scheduling strategies discussed in Section 19.5 can be used to meet the rate and deadline QoS requirements of continuous-media applications.

20.5 Disk Scheduling

We first discussed disk scheduling in Chapter 12. There, we focused primarily on systems that handle conventional data; for these systems, the scheduling goals are fairness and throughput. As a result, most traditional disk schedulers employ some form of the SCAN (Section 12.4.3) or C-SCAN (Section 12.4.4) algorithm.

Continuous-media files, however, have two constraints that conventional data files generally do not have: timing deadlines and rate requirements. These two constraints must be satisfied to preserve QoS guarantees, and disk-scheduling algorithms must be optimized for the constraints. Unfortunately, these two constraints are often in conflict. Continuous-media files typically require very high disk-bandwidth rates to satisfy their data-rate requirements. Because disks have relatively low transfer rates and relatively high latency rates, disk schedulers must reduce the latency times to ensure high bandwidth. However, reducing latency times may result in a scheduling policy that does not prioritize according to deadlines. In this section, we explore two disk-scheduling algorithms that meet the QoS requirements for continuous-media systems.

20.5.1 Earliest-Deadline-First Scheduling

We first saw the earliest-deadline-first (EDF) algorithm in Section 19.5.2 as an example of a CPU-scheduling algorithm that assigns priorities according to deadlines. EDF can also be used as a disk-scheduling algorithm; in this context, EDF uses a queue to order requests according to the time each request must be completed (its deadline). EDF is similar to shortest-seek-time-first (SSTF), which was discussed in 12.4.2, except that instead of servicing the request closest to the current cylinder, we service requests according to deadline—the request with the closest deadline is serviced first.

A problem with this approach is that servicing requests strictly according to deadline may result in higher seek times, since the disk heads may move randomly throughout the disk without any regard to their current position. For example, suppose a disk head is currently at cylinder 75 and the queue of cylinders (ordered according to deadlines) is 98, 183, 105. Under strict EDF scheduling, the disk head will move from 75, to 98, to 183, and then back to 105. Note that the head passes over cylinder 105 as it travels from 98 to 183. It is possible that the disk scheduler could have serviced the request for cylinder 105 en route to cylinder 183 and still preserved the deadline requirement for cylinder 183.

20.5.2 SCAN-EDF Scheduling

The fundamental problem with strict EDF scheduling is that it ignores the position of the read-write heads of the disk; it is possible that the movement of the heads will swing wildly to and fro across the disk, leading to unacceptable seek times that negatively affect disk throughput. Recall that this is the same issue faced with FCFS scheduling (Section 12.4.1). We ultimately addressed this issue by adopting SCAN scheduling, wherein the disk arm moves in one direction across the disk, servicing requests according to their proximity to the current cylinder. Once the disk arm reaches the end of the disk, it begins moving in the reverse direction. This strategy optimizes seek times.

SCAN-EDF is a hybrid algorithm that combines EDF with SCAN scheduling. SCAN-EDF starts with EDF ordering but services requests with the same deadline using SCAN order. What if several requests have different deadlines that are relatively close together? In this case, SCAN-EDF may batch requests, using SCAN ordering to service requests in the same batch. There are many techniques for batching requests with similar deadlines; the only requirement is that reordering requests within a batch must not prevent a request from being serviced by its deadline. If deadlines are equally distributed, batches can be organized in groups of a certain size—say, 10 requests per batch.

Another approach is to batch requests whose deadlines fall within a given time threshold—say, 100 milliseconds. Let's consider an example in which we batch requests in this way. Assume we have the following requests, each with a specified deadline (in milliseconds) and the cylinder being requested:

request	deadline	cylinder
A	150	25
B	201	112
C	399	95
D	94	31
E	295	185
F	78	85
G	165	150
H	125	101
I	300	85
J	210	90

Suppose we are at $time_0$, the cylinder currently being serviced is 50, and the disk head is moving toward cylinder 51. According to our batching scheme, requests D and F will be in the first batch; A, G, and H in batch 2; B, E, and J in batch 3; and C and I in the last batch. Requests within each batch will be ordered according to SCAN order. Thus, in batch 1, we will first service request F and then request D. Note that we are moving downward in cylinder numbers, from 85 to 31. In batch 2, we first service request A; then the heads begin moving upward in cylinders, servicing requests H and then G. Batch 3 is serviced in the order E, B, J. Requests I and C are serviced in the final batch.

20.6 Network Management

Perhaps the foremost QoS issue with multimedia systems concerns preserving rate requirements. For example, if a client wishes to view a video compressed with MPEG-1, the quality of service greatly depends on the system's ability to deliver the frames at the required rate..

Our coverage of issues such as CPU- and disk-scheduling algorithms has focused on how these techniques can be used to better meet the quality-of-service requirements of multimedia applications. However, if the media file is being streamed over a network—perhaps the Internet—issues relating to how the network delivers the multimedia data can also significantly affect how QoS demands are met. In this section, we explore several network issues related to the unique demands of continuous media.

Before we proceed, it is worth noting that computer networks in general —and the Internet in particular— currently do not provide network protocols that can ensure the delivery of data with timing requirements. (There are some proprietary protocols—notably those running on Cisco routers—that do allow certain network traffic to be prioritized to meet QoS requirements. Such proprietary protocols are not generalized for use across the Internet and therefore do not apply to our discussion.)

When data are routed across a network, it is likely that the transmission will encounter congestion, delays, and other network traffic issues—issues that are beyond the control of the originator of the data. For multimedia data with timing requirements, any timing issues must be synchronized between the end hosts: the server delivering the content and the client playing it back.

One protocol that addresses timing issues is the **real-time transport protocol (RTP)**. RTP is an Internet standard for delivering real-time data, including audio and video. It can be used for transporting media formats such as MP3 audio files and video files compressed using MPEG. RTP does not provide any QoS guarantees; rather, it provides features that allow a receiver to remove jitter introduced by delays and congestion in the network.

In following sections, we consider two other approaches for handling the unique requirements of continuous media.

20.6.1 Unicasting and Multicasting

In general, there are three methods for delivering content from a server to a client across a network:

- **Unicasting.** The server delivers the content to a single client. If the content is being delivered to more than one client, the server must establish a separate unicast for each client.
- **Broadcasting.** The server delivers the content to all clients, regardless of whether they wish to receive the content or not.
- **Multicasting.** The server delivers the content to a group of receivers who indicate they wish to receive the content; this method lies somewhere between unicasting and broadcasting.

An issue with unicast delivery is that the server must establish a separate unicast session for each client. This seems especially wasteful for live real-time

streaming, where the server must make several copies of the same content, one for each client. Obviously, broadcasting is not always appropriate, as not all clients may wish to receive the stream. (Suffice to say that broadcasting is typically only used across local area networks and is not possible across the public Internet.)

Multicasting appears to be a reasonable compromise, since it allows the server to deliver a single copy of the content to all clients indicating that they wish to receive it. The difficulty with multicasting from a practical standpoint is that the clients must be physically close to the server or to intermediate routers that relay the content from the originating server. If the route from the server to the client must cross intermediate routers, the routers must also support multicasting. If these conditions are not met, the delays incurred during routing may result in violation of the timing requirements of the continuous media. In the worst case, if a client is connected to an intermediate router that does not support multicasting, the client will be unable to receive the multicast stream at all!

Currently, most streaming media are delivered across unicast channels; however, multicasting is used in various areas where the organization of the server and clients is known in advance. For example, a corporation with several sites across a country may be able to ensure that all sites are connected to multicasting routers and are within reasonable physical proximity to the routers. The organization will then be able to deliver a presentation from the chief executive officer using multicasting.

20.6.2 Real-Time Streaming Protocol

In Section 20.1.1, we described some features of streaming media. As we noted there, users may be able to randomly access a media stream, perhaps rewinding or pausing, as they would with a VCR controller. How is this possible?

To answer this question, let's consider how streaming media are delivered to clients. One approach is to stream the media from a standard web server using the hypertext transport protocol, or HTTP—the protocol used to deliver

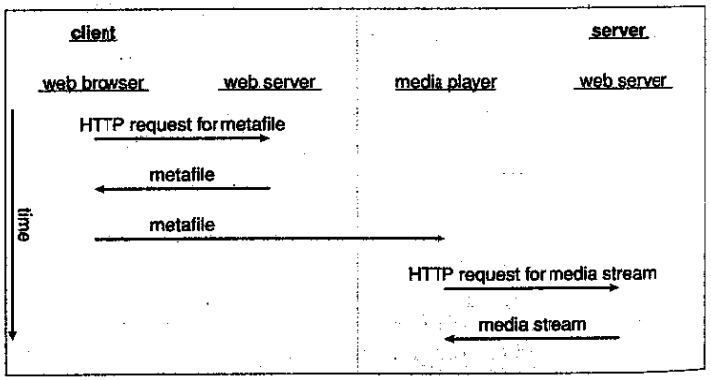


Figure 20.2 Streaming media from a conventional web server.

documents from a web server. Quite often, clients use a **media player**, such as QuickTime, RealPlayer, or Windows Media Player, to play back media streamed from a standard web server. Typically, the client first requests a **metafile**, which contains the location (possibly identified by a uniform resource locator, or URL) of the streaming media file. This metafile is delivered to the client's web browser, and the browser then starts the appropriate media player according to the type of media specified by the metafile. For example, a Real Audio stream would require the RealPlayer, while the Windows Media Player would be used to play back streaming Windows media. The media player then contacts the web server and requests the streaming media. The stream is delivered from the web server to the media player using standard HTTP requests. This process is outlined in Figure 20.2.

The problem with delivering streaming media from a standard web server is that HTTP is considered a **stateless** protocol; thus, a web server does not maintain the state (or status) of its connection with a client. As a result, it is difficult for a client to pause during the delivery of streaming media content, since pausing would require the web server to know where in the stream to begin when the client wished to resume playback.

An alternative strategy is to use a specialized streaming server that is designed specifically for streaming media. One protocol designed for communication between streaming servers and media players is known as the real-time streaming protocol, or RTSP. The significant advantage RTSP provides over HTTP is a stateful connection between the client and the server, which allows the client to pause or seek to random positions in the stream during playback. Delivery of streaming media using RTSP is similar to delivery using HTTP (Figure 20.2) in that the meta file is delivered using a conventional web server. However, rather than using a web server, the streaming media is delivered from a streaming server using the RTSP protocol. The operation of RTSP is shown in Figure 20.3.

RTSP defines several commands as part of its protocol; these commands are sent from a client to an RTSP streaming server. The commands include:

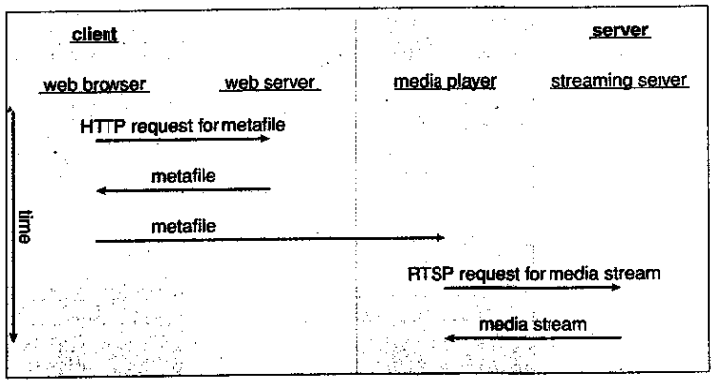


Figure 20.3 Real-time streaming protocol (RTSP).

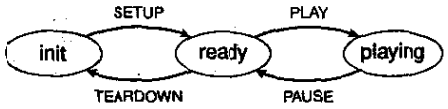


Figure 20.4 Finite-state machine representing RTSP.

- **SETUP.** The server allocates resources for a client session.
- **PLAY.** The server delivers a stream to a client session established from a SETUP command.
- **PAUSE.** The server suspends delivery of a stream but maintains the resources for the session.
- **TEARDOWN.** The server breaks down the connection and frees up resources allocated for the session.

The commands can be illustrated with a state machine for the server, as shown in Figure 20.4. As you can see in the figure, the RTSP server may be in one of three states: **init**, **ready**, and **playing**. Transitions between these three states are triggered when the server receives one of the RTSP commands from the client.

Using RTSP rather than HTTP for streaming media offers several other advantages, but they are primarily related to networking issues and are therefore beyond the scope of this text. We encourage interested readers to consult the bibliographical notes at the end of this chapter for sources of further information.

20.7 An Example: CineBlitz

The CineBlitz multimedia storage server is a high-performance media server that supports both continuous media with rate requirements (such as video and audio) and conventional data with no associated rate requirements (such as text and images). CineBlitz refers to clients with rate requirements as **real-time clients**, whereas **non-real-time clients** have no rate constraints. CineBlitz guarantees to meet the rate requirements of real-time clients by implementing an admission controller, admitting a client only if there are sufficient resources to allow data retrieval at the required rate. In this section, we explore the CineBlitz disk-scheduling and admission-control algorithms.

20.7.1 Disk Scheduling

The CineBlitz disk scheduler services requests in **cycles**. At the beginning of each service cycle, requests are placed in C-SCAN order (Section 12.4.4). Recall from our earlier discussions of C-SCAN that the disk heads move from one end of the disk to the other. However, rather than reversing direction when they reach the end of the disk, as in pure SCAN disk scheduling (Section 12.4.3), the disk heads move back to the beginning of the disk.

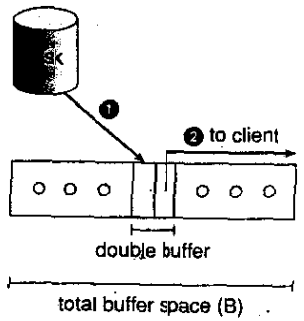


Figure 20.5 Double buffering in CineBlitz.

20.7.2 Admission Control

The admission-control algorithm in CineBlitz must monitor requests from both real-time and non-real-time clients, ensuring that both classes of clients receive service. Furthermore, the admission controller must provide the rate guarantees required by real-time clients. To ensure fairness, only a fraction p of time is reserved for real-time clients, while the remainder, $1 - p$, is set aside for non-real-time clients. Here, we explore the admission controller for real-time clients only; thus, the term *client* refers to a real-time client.

The admission controller in CineBlitz monitors various system resources, such as disk bandwidth and disk latency, while keeping track of available buffer space. The CineBlitz admission controller admits a client only if there is enough available disk bandwidth and buffer space to retrieve data for the client at its required rate.

CineBlitz queues requests R_1, R_2, R_3, ..R_n for continuous media files where r_i is the required data rate for a given request R_i. Requests in the queue are served in cyclic order using a technique known as **double buffering**, wherein a buffer is allocated for each request R_i of size $2 \times T \times r_i$.

During each cycle I, the server must:

1. Retrieve the data from disk to buffer (I mod 2).
2. Transfer data from the ((I + 1) mod 2) buffer to the client.

This process is illustrated in Figure 20.5. For N clients, the total buffer space B required is

$$\sum_{i=1}^{N} 2 \times T \times r_i \leq B. \tag{20.1}$$

The fundamental idea behind the admission controller in CineBlitz is to bound requests for entry into the queue according to the following criteria:

1. The service time for each request is first estimated.
2. A request is admitted only if the sum of the estimated service times for all admitted requests does not exceed the duration of service cycle T.

Let $T \times r_i$ bits be retrieved during a cycle for each real-time client R_i with rate r_i. If $R_1, R_2, \ldots R_n$ are the clients currently active in the system, then the admission controller must ensure that the total times for retrieving $T \times r_1, T \times r_2, \ldots, T \times r_n$ bits for the corresponding real-time clients does not exceed T. We explore the details of this admission policy in the remainder of this section.

If b is the size of a disk block, then the maximum number of disk blocks that can be retrieved for request R_k during each cycle is $\lceil (T \times r_k)/b \rceil + 1$. The 1 in this formula comes from the fact that, if $T \times r_k$ is less than b, then it is possible for $T \times r_k$ bits to span the last portion of one disk block and the beginning of another, causing two blocks to be retrieved. We know that the retrieval of a disk block involves (a) a seek to the track containing the block and (b) the rotational delay as the data in the desired track arrives under the disk head. As described, CineBlitz uses a C-SCAN disk-scheduling algorithm, so disk blocks are retrieved in the sorted order of their positions on the disk.

If t_{seek} and t_{rot} refer to the worst-case seek and rotational delay times, the maximum latency incurred for servicing N requests is

$$2 \times t_{seek} + \sum_{i=1}^{N} \left(\lceil \frac{T \times r_i}{b} \rceil - 1 \right) \times t_{rot}. \tag{20.2}$$

In this equation, the $2 \times t_{seek}$ component refers to the maximum disk-seek latency incurred in a cycle. The second component reflects the sum of the retrievals of the disk blocks multiplied by the worst-case rotational delay.

If the transfer rate of the disk is r_{disk}, then the time to transfer $T \times r_k$ bits of data for request R_k is $(T \times r_k)/r_{disk}$. As a result, the total time for retrieving $T \times r_1, T \times r_2, \ldots, T \times r_n$ bits for requests $R_1, R_2, \ldots, R_n$ is the sum of equation 20.2 and

$$\sum_{i=1}^{N} \frac{T \times r_i}{r_{disk}} \tag{20.3}$$

Therefore, the admission controller in CineBlitz only admits a new client R_i if at least $2 \times T \times r_i$ bits of free buffer space are available for the client and the following equation is satisfied:

$$2 \times t_{seek} + \sum_{i=1}^{N} \left(\lceil \frac{T \times r_i}{b} \rceil + 1 \right) \times t_{rot} + \sum_{i=1}^{N} \frac{T \times r_i}{r_{disk}} \leq T. \tag{20.4}$$

20.8 Summary

Multimedia applications are in common use in modern computer systems. Multimedia files include video and audio files, which may be delivered to systems such as desktop computers, personal digital assistants, and cell phones. The primary distinction between multimedia data and conventional data is that multimedia data have specific rate and deadline requirements. Because multimedia files have specific timing requirements, the data must often be compressed before delivery to a client for playback. Multimedia data may be delivered either from the local file system or from a multimedia server across a network connection using a technique known as streaming.

The timing requirements of multimedia data are known as quality-of-service requirements, and conventional operating systems often cannot make quality-of-service guarantees. To provide quality of service, multimedia systems must provide a form of admission control whereby a system accepts a request only if it can meet the quality-of-service level specified by the request. Providing quality-of-service guarantees requires evaluating how an operating system performs CPU scheduling, disk scheduling, and network management. Both CPU and disk scheduling typically use the deadline requirements of a continuous-media task as a scheduling criterion. Network management requires the use of protocols that handle delay and jitter caused by the network as well as allowing a client to pause or move to different positions in the stream during playback.

Exercises

20.1 Provide examples of multimedia applications that are delivered over the Internet.

20.2 Distinguish between progressive download and real-time streaming.

20.3 Which of the following types of real-time streaming applications can tolerate delay? Which can tolerate jitter?

- Live real-time streaming
- On-demand real-time streaming

20.4 Discuss what techniques could be used to meet quality-of-service requirements for multimedia applications in the following components of a system:

- Process scheduler
- Disk scheduler
- Memory manager

20.5 Explain why the traditional Internet protocols for transmitting data are not sufficient to provide the quality-of-service guarantees required for a multimedia system. Discuss what changes are required to provide the QoS guarantees.

20.6 Assume that a digital video file is being displayed at a rate of 30 frames per second; the resolution of each frame is 640 × 480, and 24 bits are being used to represent each color. Assuming that no compression is being used, what is the bandwidth necessary to deliver this file? Next, assuming that the file has been compressed at a ratio of 200 : 1, what is the bandwidth necessary to deliver the compressed file?

20.7 A multimedia application consists of a set containing 100 images, 10 minutes of video, and 10 minutes of audio. The compressed sizes of the images, video, and audio are 500 MB, 550 MB, and 8 MB, respectively. The images were compressed at a ratio of 15 : 1, and the video and

audio were compressed at 200 : 1 and 10 : 1, respectively. What were the sizes of the images, video, and audio before compression?

20.8 Assume that we wish to compress a digital video file using MPEG-1 technology. The target bit rate is 1.5 Mbps. If the video is displayed at a resolution of 352 × 240 at 30 frames per second using 24 bits to represent each color, what is the necessary compression ratio to achieve the desired bit rate?

20.9 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.

20.10 The following table contains a number of requests with their associated deadlines and cylinders. Requests with deadlines occurring within 100 milliseconds of each other will be batched. The disk head is currently at cylinder 94 and is moving toward cylinder 95. If SCAN-EDF disk scheduling is used, how are the requests batched together, and what is the order of requests within each batch?

request	deadline	cylinder
R1	57	77
R2	300	95
R3	250	25
R4	88	28
R5	85	100
R6	110	90
R7	299	50
R8	300	77
R9	120	12
R10	212	2

20.11 Repeat the preceding question, but this time batch requests that have deadlines occurring within 75 milliseconds of each other.

20.12 Contrast unicasting, multicasting, and broadcasting as techniques for delivering content across a computer network.

20.13 Describe why HTTP is often insufficient for delivering streaming media.

20.14 What operating principle is used by the CineBlitz system in performing admission control for requests for media files?

Bibliographical Notes

Fuhrt [1994] provides a general overview of multimedia systems. Topics related to the delivery of multimedia through networks can be found in Kurose and Ross [2005]. Operating-system support for multimedia is discussed in Steinmetz [1995] and Leslie et al. [1996]. Resource management for resources such as processing capability and memory buffers are discussed in Mercer et al. [1994] and Druschel and Peterson [1993]. Reddy and Wyllie [1994] give a good overview of issues relating to the use of I/O in a multimedia system. Discussions regarding the appropriate programming model for developing multimedia applications are presented in Regehr et al. [2000] An admission control system for a rate-morotonic scheduler is considered in Lauzac et al. [2003]. Bolosky et a.. [1997] present a system for serving video data and discuss the schedule-management issues that arise in such a system. The details of a real-time streaming protocol can be found at http://www.rtsp.org. Tudor [1995] gives a tutorial on MPEG-2. A tutorial on video compression techniques can be found at http://www.wave-report.com/tutorials/VC.htm.