CHAPTER 10

Multiprocessor and Real-Time Scheduling

10.1 Multiprocessor Scheduling

Granularity
Granularity Example: Valve Game Software
Design Issues
Process Scheduling
Thread Scheduling

10.2 Real-Time Scheduling

Background
Characteristics of Real-Time Operating Systems
Real-Time Scheduling
Deadline Scheduling
Rate Monotonic Scheduling
Priority Inversion

10.3 Linux Scheduling

Real-Time Scheduling
Non-Real-Time Scheduling

10.4 Unix SVR4 Scheduling

10.5 Windows Scheduling

Process and Thread Priorities
Multiprocessor Scheduling

10.6 Summary

10.7 Recommended Reading

10.8 Key Terms, Review Questions, and Problems

This chapter continues our survey of process and thread scheduling. We begin with an examination of issues raised by the availability of more than one processor. A number of design issues are explored. This is followed by a look at the scheduling of processes on a multiprocessor system. Then the somewhat different design considerations for multiprocessor thread scheduling are examined. The second section of this chapter covers real-time scheduling. The section begins with a discussion of the characteristics of real-time processes and then looks at the nature of the scheduling process. Two approaches to real-time scheduling, deadline scheduling and rate monotonic scheduling, are examined.

10.1 MULTIPROCESSOR SCHEDULING

When a computer system contains more than a single processor, several new issues are introduced into the design of the scheduling function. We begin with a brief overview of multiprocessors and then look at the rather different considerations when scheduling is done at the process level and the thread level.

We can classify multiprocessor systems as follows:

- **Loosely coupled or distributed multiprocessor, or cluster:** Consists of a collection of relatively autonomous systems, each processor having its own main memory and I/O channels. We address this type of configuration in Chapter 16.
- **Functionally specialized processors:** An example is an I/O processor. In this case, there is a master, general-purpose processor; specialized processors are controlled by the master processor and provide services to it. Issues relating to I/O processors are addressed in Chapter 11.
- **Tightly coupled multiprocessing:** Consists of a set of processors that share a common main memory and are under the integrated control of an operating system.

Our concern in this section is with the last category, and specifically with issues relating to scheduling.

Granularity

A good way of characterizing multiprocessors and placing them in context with other architectures is to consider the synchronization granularity, or frequency of synchronization, between processes in a system. We can distinguish five categories of parallelism that differ in the degree of granularity. These are summarized in Table 10.1, which is adapted from [GEHR87] and [WOOD89].

Independent Parallelism With independent parallelism, there is no explicit synchronization among processes. Each represents a separate, independent application or job. A typical use of this type of parallelism is in a time-sharing system. Each user is performing a particular application, such as word processing or using a spreadsheet. The multiprocessor provides the same service as a multiprogrammed uniprocessor. Because more than one processor is available, average response time to the users will be less.

Table 10.1 Synchronization Granularity and Processes

Grain Size	Description	Synchronization Interval (Instructions)
Fine	Parallelism inherent in a single instruction stream.	<20
Medium	Parallel processing or multitasking within a single application	20–200
Coarse	Multiprocessing of concurrent processes in a multiprogramming environment	200–2000
Very Coarse	Distributed processing across network nodes to form a single computing environment	2000–1M
Independent	Multiple unrelated processes	not applicable

It is possible to achieve a similar performance gain by providing each user with a personal computer or workstation. If any files or information are to be shared, then the individual systems must be hooked together into a distributed system supported by a network. This approach is examined in Chapter 16. On the other hand, a single, multiprocessor shared system in many instances is more cost-effective than a distributed system, allowing economies of scale in disks and other peripherals.

Coarse and Very Coarse-Grained Parallelism With coarse and very coarse-grained parallelism, there is synchronization among processes, but at a very gross level. This kind of situation is easily handled as a set of concurrent processes running on a multiprogrammed uniprocessor and can be supported on a multiprocessor with little or no change to user software.

A simple example of an application that can exploit the existence of a multiprocessor is given in [WOOD89]. The authors have developed a program that takes a specification of files needing recompilation to rebuild a piece of software and determines which of these compiles (usually all of them) can be run simultaneously. The program then spawns one process for each parallel compile. The authors report that the speedup on a multiprocessor actually exceeds what would be expected by simply adding up the number of processors in use, due to synergies in the disk buffer caches (a topic explored in Chapter 11) and sharing of compiler code, which is loaded into memory only once.

In general, any collection of concurrent processes that need to communicate or synchronize can benefit from the use of a multiprocessor architecture. In the case of very infrequent interaction among processes, a distributed system can provide good support. However, if the interaction is somewhat more frequent, then the overhead of communication across the network may negate some of the potential speedup. In that case, the multiprocessor organization provides the most effective support.

Medium-Grained Parallelism We saw in Chapter 4 that a single application can be effectively implemented as a collection of threads within a single process. In this case, the programmer must explicitly specify the potential parallelism of an application. Typically, there will need to be rather a high degree of coordination and interaction among the threads of an application, leading to a medium-grain level of synchronization.

Whereas independent, very coarse, and coarse-grained parallelism can be supported on either a multiprogrammed uniprocessor or a multiprocessor with little or no impact on the scheduling function, we need to reexamine scheduling when dealing with the scheduling of threads. Because the various threads of an application interact so frequently, scheduling decisions concerning one thread may affect the performance of the entire application. We return to this issue later in this section.

Fine-Grained Parallelism Fine-grained parallelism represents a much more complex use of parallelism than is found in the use of threads. Although much work has been done on highly parallel applications, this is so far a specialized and fragmented area, with many different approaches.

Granularity Example: Valve Game Software

Valve is an entertainment and technology company that has developed a number of popular games, as well as the Source engine, one of the most widely played game engines available. Source is an animation engine used by Valve for its games and licensed for other game developers.

In recent years, Valve has reprogrammed the Source engine software to use multithreading to exploit the power of multicore processor chips from Intel and AMD. Multicore refers to the placement of multiple processors on a single chip, typically 2 or 4 processors. An SMP system can consist of a single chip or multiple chips, providing extensive opportunity for parallel multithreaded execution. The revised Source engine code provides more powerful support for Valve games such as Half Life 2.

From Valve's perspective, threading granularity options are defined as follows [HARR06]:

- **Coarse threading:** Individual modules, called systems, are assigned to individual processors. In the Source engine case, this would mean putting rendering on one processor, AI (artificial intelligence) on another, physics on another, and so on. This is straightforward. In essence, each major module is single threaded and the principal coordination involves synchronizing all the threads with a timeline thread.
- **Fine-grained threading:** Many similar or identical tasks are spread across multiple processors. For example, a loop that iterates over an array of data can be split up into a number of smaller parallel loops in individual threads that can be scheduled in parallel.
- **Hybrid threading:** This involves the selective use of fine-grain threading for some systems and single threading for other systems.

Valve found that through coarse threading, it could achieve up to twice the performance across two processors compared to executing on a single processor. But this performance gain could only be achieved with contrived cases. For real-world gameplay, the improvement was on the order of a factor of 1.2. Valve also found that effective use of fine-grain threading was difficult. The time per work unit can be variable, and managing the timeline of outcomes and consequences involved complex programming.

Valve found that a hybrid threading approach was the most promising and would scale the best as multiprocessors with eight or sixteen processors became available. Valve identified systems that operate very effectively being permanently assigned to a single processor. An example is sound mixing, which has little user interaction, is not constrained by the frame configuration of windows, and works on its own set of data. Other modules, such as scene rendering, can be organized into a number of threads so that the module can execute on a single processor but achieve greater performance as it is spread out over more and more processors.

Figure 10.1 illustrates the thread structure for the rendering module. In this hierarchical structure, higher-level threads spawn lower-level threads as needed. The rendering module relies on a critical part of the Source engine, the world list, which is a database representation of the visual elements in the game's world. The first task is to determine what are the areas of the world that need to be rendered. The next task is to determine what objects are in the scene as viewed from multiple angles. Then comes the processor-intensive work. The rendering module has to work out the rendering of each object from multiple points of view, such as the player's view, the view of TV monitors, and the point of view of reflections in water.

Some of the key elements of the threading strategy for the rendering module are listed in [LEON07] and include

- Construct scene rendering lists for multiple scenes in parallel (e.g., the world and its reflection in water).
- Overlap graphics simulation.

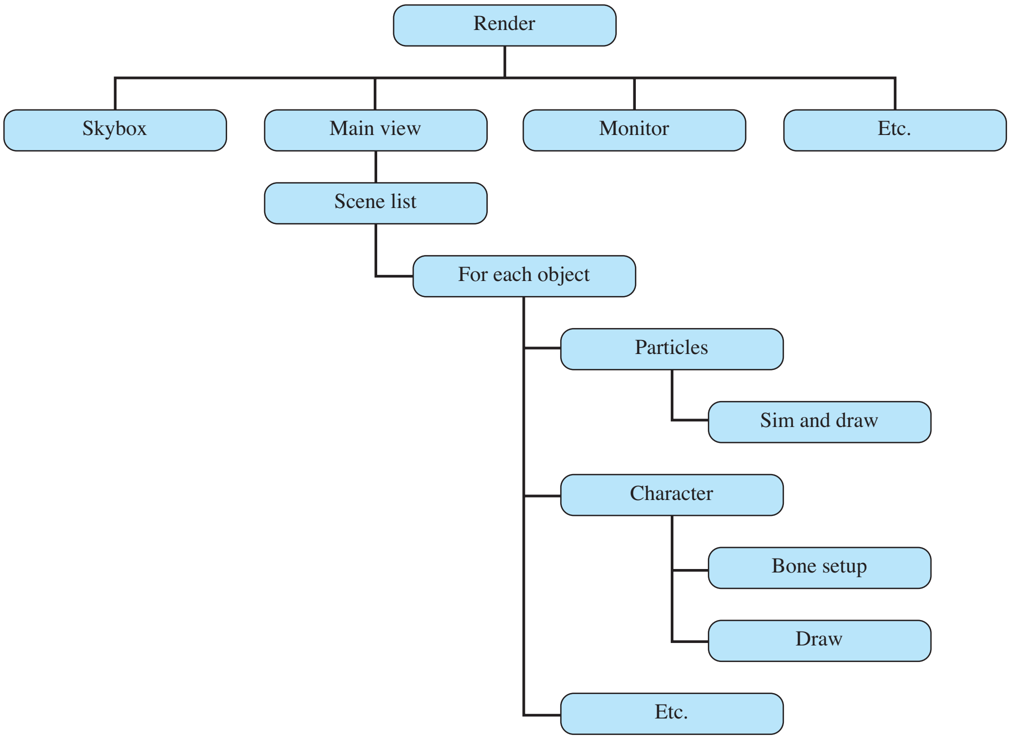


Figure 10.1 **Hybrid Threading for Rendering Module**

- Compute character bone transformations for all characters in all scenes in parallel.
- Allow multiple threads to draw in parallel.

The designers found that simply locking key databases, such as the world list, for a thread was too inefficient. Over 95% of the time, a thread is trying to read from a data set, and only 5% of the time at most is spent in writing to a data set. Thus, a concurrency mechanism based on the single-writer-multiple-readers model works effectively.

Design Issues

Scheduling on a multiprocessor involves three interrelated issues:

- The assignment of processes to processors
- The use of multiprogramming on individual processors
- The actual dispatching of a process

In looking at these three issues, it is important to keep in mind that the approach taken will depend, in general, on the degree of granularity of the applications and on the number of processors available.

Assignment of Processes to Processors If we assume that the architecture of the multiprocessor is uniform, in the sense that no processor has a particular physical advantage with respect to access to main memory or to I/O devices, then the simplest scheduling approach is to treat the processors as a pooled resource and assign processes to processors on demand. The question then arises as to whether the assignment should be static or dynamic.

If a process is permanently assigned to one processor from activation until its completion, then a dedicated short-term queue is maintained for each processor. An advantage of this approach is that there may be less overhead in the scheduling function, because the processor assignment is made once and for all. Also, the use of dedicated processors allows a strategy known as group or gang scheduling, as discussed later.

A disadvantage of static assignment is that one processor can be idle, with an empty queue, while another processor has a backlog. To prevent this situation, a common queue can be used. All processes go into one global queue and are scheduled to any available processor. Thus, over the life of a process, the process may be executed on different processors at different times. In a tightly coupled shared-memory architecture, the context information for all processes will be available to all processors, and therefore the cost of scheduling a process will be independent of the identity of the processor on which it is scheduled. Yet another option is dynamic load balancing, in which threads are moved for a queue for one processor to a queue for another processor; Linux uses this approach.

Regardless of whether processes are dedicated to processors, some means is needed to assign processes to processors. Two approaches have been used: master/slave and peer. With a master/slave architecture, key kernel functions of the operating system always run on a particular processor. The other processors may only execute user programs. The master is responsible for scheduling jobs. Once a process is active, if the slave needs service (e.g., an I/O call), it must send a request to the master and wait for the service to be performed. This approach is quite simple and requires little enhancement to a uniprocessor multiprogramming operating system. Conflict resolution is simplified because one processor has control of all memory and I/O resources. There are two disadvantages to this approach: (1) A failure of the master brings down the whole system, and (2) the master can become a performance bottleneck.

In a peer architecture, the kernel can execute on any processor, and each processor does self-scheduling from the pool of available processes. This approach complicates the operating system. The operating system must ensure that two processors do not choose the same process and that the processes are not somehow lost from the queue. Techniques must be employed to resolve and synchronize competing claims to resources.

There is, of course, a spectrum of approaches between these two extremes. One approach is to provide a subset of processors dedicated to kernel processing instead of just one. Another approach is simply to manage the difference between the needs of kernel processes and other processes on the basis of priority and execution history.

The Use of Multiprogramming on Individual Processors When each process is statically assigned to a processor for the duration of its lifetime, a new question arises: Should that processor be multiprogrammed? The reader's first reaction may be to wonder why the question needs to be asked; it would appear particularly wasteful to tie up a processor with a single process when that process may frequently be blocked waiting for I/O or because of concurrency/synchronization considerations.

In the traditional multiprocessor, which is dealing with coarse-grained or independent synchronization granularity (see Table 10.1), it is clear that each individual processor should be able to switch among a number of processes to achieve high utilization and therefore better performance. However, for medium-grained applications running on a multiprocessor with many processors, the situation is less clear. When many processors are available, it is no longer paramount that every single processor be busy as much as possible. Rather, we are concerned to provide the best performance, on average, for the applications. An application that consists of a number of threads may run poorly unless all of its threads are available to run simultaneously.

Process Dispatching The final design issue related to multiprocessor scheduling is the actual selection of a process to run. We have seen that, on a multiprogrammed uniprocessor, the use of priorities or of sophisticated scheduling algorithms based on past usage may improve performance over a simple-minded first-come-first-served strategy. When we consider multiprocessors, these complexities may be unnecessary or even counterproductive, and a simpler approach may be more effective with less overhead. In the case of thread scheduling, new issues come into play that may be more important than priorities or execution histories. We address each of these topics in turn.

Process Scheduling

In most traditional multiprocessor systems, processes are not dedicated to processors. Rather there is a single queue for all processors, or if some sort of priority scheme is used, there are multiple queues based on priority, all feeding into the common pool of processors. In any case, we can view the system as being a multiserver queuing architecture.

Consider the case of a dual-processor system in which each processor of the dual-processor system has half the processing rate of a processor in the single-processor system. [SAUE81] reports a queuing analysis that compares FCFS scheduling to round robin and to shortest remaining time. The study is concerned with process service time, which measures the amount of processor time a process needs, either for a total job or the amount of time needed each time the process is ready to use the processor. In the case of round robin, it is assumed that the time quantum is large compared to context-switching overhead and small compared to mean service time. The results depend on the variability that is seen in service times. A common measure of variability is the coefficient of variation, C_s.[1] A value of $C_s = 0$ corresponds to the case where there is no variability: the service times of all processes are equal. Increasing values of C_s correspond to increasing variability among the service times. That is, the larger the value of C_s, the more widely do the values of the services times vary. Values of C_s of 5 or more are not unusual for processor service time distributions.

Figure 10.2a compares round-robin throughput to FCFS throughput as a function of C_s. Note that the difference in scheduling algorithms is much smaller in the dual-processor case. With two processors, a single process with long service time is much less disruptive in the FCFS case; other processes can use the other processor. Similar results are shown in Figure 10.2b.

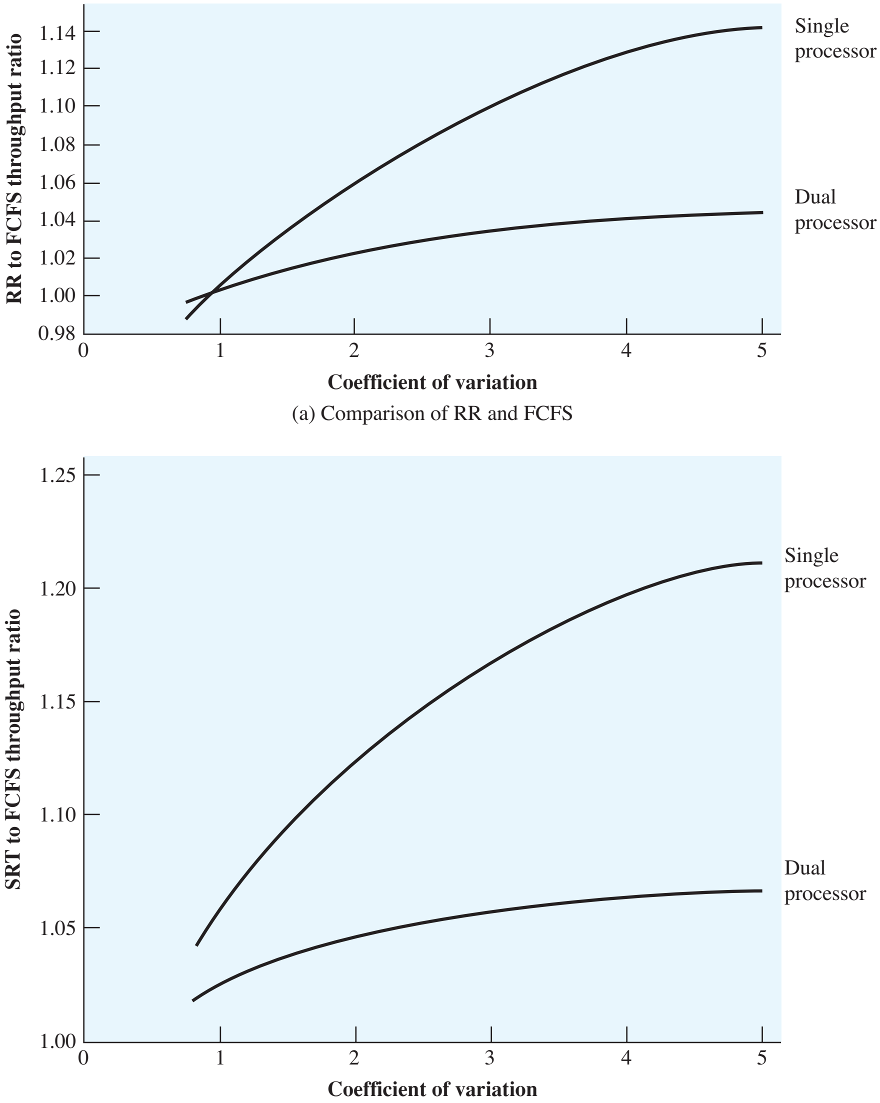
The study in [SAUE81] repeated this analysis under a number of assumptions about degree of multiprogramming, mix of I/O-bound versus CPU-bound processes, and the use of priorities. The general conclusion is that the specific scheduling discipline is much less important with two processors than with one. It should be evident that this conclusion is even stronger as the number of processors increases. Thus, a simple FCFS discipline or the use of FCFS within a static priority scheme may suffice for a multiple-processor system.

Thread Scheduling

As we have seen, with threads, the concept of execution is separated from the rest of the definition of a process. An application can be implemented as a set of threads, which cooperate and execute concurrently in the same address space.

On a uniprocessor, threads can be used as a program structuring aid and to overlap I/O with processing. Because of the minimal penalty in doing a thread switch compared to a process switch, these benefits are realized with little cost. However, the full power of threads becomes evident in a multiprocessor system. In this environment, threads can be used to exploit true parallelism in an application. If the various threads of an application are simultaneously run on separate processors, dramatic gains in performance are possible. However, it can be shown that for applications that require significant interaction among threads (medium-grain parallelism),

[1] The value of C_s is calculated as σ_s/T_s, where σ_s is the standard deviation of service time and T_s is the mean service time. For a further explanation of C_s, see the discussion in the Queuing Analysis document at WilliamStallings.com/StudentSupport.html.



(a) Comparison of RR and FCFS

(b) Comparison of SRT and FCFS

Figure 10.2 Comparison of Scheduling Performance for One and Two Processors

small differences in thread management and scheduling can have a significant performance impact [ANDE89].

Among the many proposals for multiprocessor thread scheduling and processor assignment, four general approaches stand out:

- **Load sharing:** Processes are not assigned to a particular processor. A global queue of ready threads is maintained, and each processor, when idle, selects a thread from the queue. The term **load sharing** is used to distinguish this strategy from load-balancing schemes in which work is allocated on a more permanent basis (e.g., see [FEIT90a]).[2]

- **Gang scheduling:** A set of related threads is scheduled to run on a set of processors at the same time, on a one-to-one basis.
- **Dedicated processor assignment:** This is the opposite of the load-sharing approach and provides implicit scheduling defined by the assignment of threads to processors. Each program is allocated a number of processors equal to the number of threads in the program, for the duration of the program execution. When the program terminates, the processors return to the general pool for possible allocation to another program.
- **Dynamic scheduling:** The number of threads in a process can be altered during the course of execution.

Load Sharing

Load sharing is perhaps the simplest approach and the one that carries over most directly from a uniprocessor environment. It has several advantages:

- The load is distributed evenly across the processors, assuring that no processor is idle while work is available to do.
- No centralized scheduler is required; when a processor is available, the scheduling routine of the operating system is run on that processor to select the next thread.
- The global queue can be organized and accessed using any of the schemes discussed in Chapter 9, including priority-based schemes and schemes that consider execution history or anticipated processing demands.

[LEUT90] analyzes three different versions of load sharing:

- **First come first served (FCFS):** When a job arrives, each of its threads is placed consecutively at the end of the shared queue. When a processor becomes idle, it picks the next ready thread, which it executes until completion or blocking.
- **Smallest number of threads first:** The shared ready queue is organized as a priority queue, with highest priority given to threads from jobs with the smallest number of unscheduled threads. Jobs of equal priority are ordered according to which job arrives first. As with FCFS, a scheduled thread is run to completion or blocking.
- **Preemptive smallest number of threads first:** Highest priority is given to jobs with the smallest number of unscheduled threads. An arriving job with a smaller number of threads than an executing job will preempt threads belonging to the scheduled job.

Using simulation models, the authors report that, over a wide range of job characteristics, FCFS is superior to the other two policies in the preceding list. Further, the

[2] Some of the literature on this topic refers to this approach as *self-scheduling*, because each processor schedules itself without regard to other processors. However, this term is also used in the literature to refer to programs written in a language that allows the programmer to specify the scheduling (e.g., see [FOST91]).

authors find that some form of gang scheduling, discussed in the next subsection, is generally superior to load sharing.

There are several disadvantages of load sharing:

- The central queue occupies a region of memory that must be accessed in a manner that enforces mutual exclusion. Thus, it may become a bottleneck if many processors look for work at the same time. When there is only a small number of processors, this is unlikely to be a noticeable problem. However, when the multiprocessor consists of dozens or even hundreds of processors, the potential for bottleneck is real.
- Preempted threads are unlikely to resume execution on the same processor. If each processor is equipped with a local cache, caching becomes less efficient.
- If all threads are treated as a common pool of threads, it is unlikely that all of the threads of a program will gain access to processors at the same time. If a high degree of coordination is required between the threads of a program, the process switches involved may seriously compromise performance.

Despite the potential disadvantages, this is one of the most commonly used schemes in current multiprocessors.

A refinement of the load-sharing technique is used in the Mach operating system [BLAC90, WEND89]. The operating system maintains a local run queue for each processor and a shared global run queue. The local run queue is used by threads that have been temporarily bound to a specific processor. A processor examines the local run queue first to give bound threads absolute preference over unbound threads. As an example of the use of bound threads, one or more processors could be dedicated to running processes that are part of the operating system. Another example is that the threads of a particular application could be distributed among a number of processors; with the proper additional software, this provides support for gang scheduling, discussed next.

Gang Scheduling The concept of scheduling a set of processes simultaneously on a set of processors predates the use of threads. [JONE80] refers to the concept as group scheduling and cites the following benefits:

- If closely related processes execute in parallel, synchronization blocking may be reduced, less process switching may be necessary, and performance will increase.
- Scheduling overhead may be reduced because a single decision affects a number of processors and processes at one time.

On the Cm* multiprocessor, the term *coscheduling* is used [GEHR87]. Coscheduling is based on the concept of scheduling a related set of tasks, called a task force. The individual elements of a task force tend to be quite small and are hence close to the idea of a thread.

The term *gang scheduling* has been applied to the simultaneous scheduling of the threads that make up a single process [FEIT90b]. Gang scheduling is useful for medium-grained to fine-grained parallel applications whose performance severely degrades when any part of the application is not running while other parts are ready to run. It is also beneficial for any parallel application, even one that is not quite so performance sensitive. The need for gang scheduling is widely recognized, and implementations exist on a variety of multiprocessor operating systems.

One obvious way in which gang scheduling improves the performance of a single application is that process switches are minimized. Suppose one thread of a process is executing and reaches a point at which it must synchronize with another thread of the same process. If that other thread is not running, but is in a ready queue, the first thread is hung up until a process switch can be done on some other processor to bring in the needed thread. In an application with tight coordination among threads, such switches will dramatically reduce performance. The simultaneous scheduling of cooperating threads can also save time in resource allocation. For example, multiple gang-scheduled threads can access a file without the additional overhead of locking during a seek, read/write operation.

The use of gang scheduling creates a requirement for processor allocation. One possibility is the following. Suppose that we have *N* processors and *M* applications, each of which has *N* or fewer threads. Then each application could be given 1/*M* of the available time on the *N* processors, using time slicing. [FEIT90a] notes that this strategy can be inefficient. Consider an example in which there are two applications, one with four threads and one with one thread. Using uniform time allocation wastes 37.5% of the processing resource, because when the single-thread application runs, three processors are left idle (see Figure 10.3). If there are several one-thread applications, these could all be fit together to increase processor utilization. If that option is not available, an alternative to uniform scheduling is scheduling that is weighted by the number of threads. Thus, the four-thread application could be given four-fifths of the time and the one-thread application given only one-fifth of the time, reducing the processor waste to 15%.

Dedicated Processor Assignment An extreme form of gang scheduling, suggested in [TUCK89], is to dedicate a group of processors to an application for the duration of the application. That is, when an application is scheduled, each of its threads is assigned a processor that remains dedicated to that thread until the application runs to completion.

This approach would appear to be extremely wasteful of processor time. If a thread of an application is blocked waiting for I/O or for synchronization with

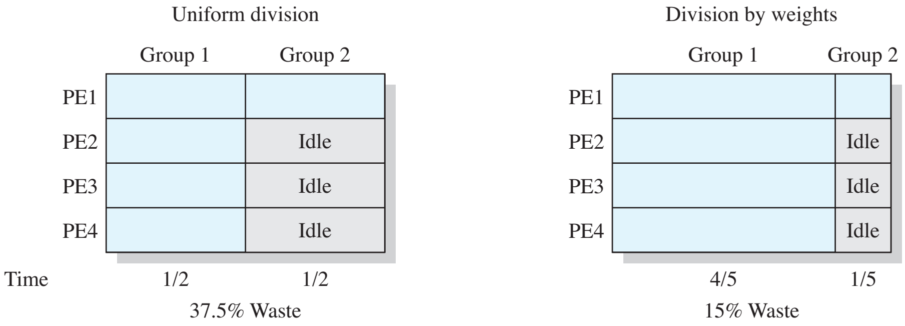


Figure 10.3 Example of Scheduling Groups with Four and One Threads [FEIT90b]

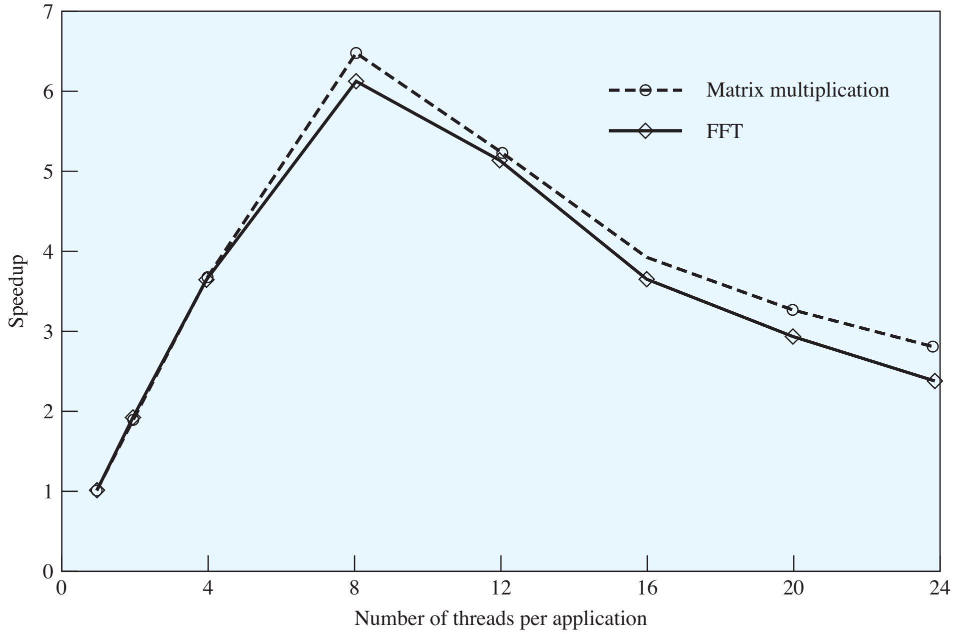


Figure 10.4 **Application Speedup as a Function of Number of Threads [TUCK89]**

another thread, then that thread's processor remains idle: there is no multiprogramming of processors. Two observations can be made in defense of this strategy:

1. In a highly parallel system, with tens or hundreds of processors, each of which represents a small fraction of the cost of the system, processor utilization is no longer so important as a metric for effectiveness or performance.
2. The total avoidance of process switching during the lifetime of a program should result in a substantial speedup of that program.

Both [TUCK89] and [ZAHO90] report analyses that support statement 2. Figure 10.4 shows the results of one experiment [TUCK89]. The authors ran two applications simultaneously (executing concurrently), a matrix multiplication and a fast Fourier transform (FFT) calculation, on a system with 16 processors. Each application breaks its problem into a number of tasks, which are mapped onto the threads executing that application. The programs are written in such a way as to allow the number of threads to be used to vary. In essence, a number of tasks are defined and queued by an application. Tasks are taken from the queue and mapped onto the available threads by the application. If there are fewer threads than tasks, then leftover tasks remain queued and are picked up by threads as they complete their assigned tasks. Clearly, not all applications can be structured in this way, but many numerical problems and some other applications can be dealt with in this fashion.

Figure 10.4 shows the speedup for the applications as the number of threads executing the tasks in each application is varied from 1 to 24. For example, we see that when both applications are started simultaneously with 24 threads each, the speedup obtained, compared to using a single thread for each application, is 2.8 for matrix multiplication and 2.4 for FFT. The figure shows that the performance of both applications worsens considerably when the number of threads in each application exceeds 8 and thus the total number of processes in the system exceeds the number of processors. Furthermore, the larger the number of threads the worse the performance gets, because there is a greater frequency of thread preemption and rescheduling. This excessive preemption results in inefficiency from many sources, including time spent waiting for a suspended thread to leave a critical section, time wasted in process switching, and inefficient cache behavior.

The authors conclude that an effective strategy is to limit the number of active threads to the number of processors in the system. If most of the applications are either single thread or can use the task-queue structure, this will provide an effective and reasonably efficient use of the processor resources.

Both dedicated processor assignment and gang scheduling attack the scheduling problem by addressing the issue of processor allocation. One can observe that the processor allocation problem on a multiprocessor more closely resembles the memory allocation problem on a uniprocessor than the scheduling problem on a uniprocessor. The issue is how many processors to assign to a program at any given time, which is analogous to how many page frames to assign to a given process at any time. [GEHR87] proposes the term *activity working set*, analogous to a virtual memory working set, as the minimum number of activities (threads) that must be scheduled simultaneously on processors for the application to make acceptable progress. As with memory management schemes, the failure to schedule all of the elements of an activity working set can lead to processor thrashing. This occurs when the scheduling of threads whose services are required induces the descheduling of other threads whose services will soon be needed. Similarly, processor fragmentation refers to a situation in which some processors are left over when others are allocated, and the leftover processors are either insufficient in number or unsuitably organized to support the requirements of waiting applications. Gang scheduling and dedicated processor allocation are meant to avoid these problems.

Dynamic Scheduling For some applications, it is possible to provide language and system tools that permit the number of threads in the process to be altered dynamically. This would allow the operating system to adjust the load to improve utilization.

[ZAHO90] proposes an approach in which both the operating system and the application are involved in making scheduling decisions. The operating system is responsible for partitioning the processors among the jobs. Each job uses the processors currently in its partition to execute some subset of its runnable tasks by mapping these tasks to threads. An appropriate decision about which subset to run, as well as which thread to suspend when a process is preempted, is left to the individual applications (perhaps through a set of run-time library routines). This approach may not be suitable for all applications. However, some applications could default to a single thread while others could be programmed to take advantage of this particular feature of the operating system.

In this approach, the scheduling responsibility of the operating system is primarily limited to processor allocation and proceeds according to the following policy. When a job requests one or more processors (either when the job arrives for the first time or because its requirements change),

1. If there are idle processors, use them to satisfy the request.
2. Otherwise, if the job making the request is a new arrival, allocate it a single processor by taking one away from any job currently allocated more than one processor.
3. If any portion of the request cannot be satisfied, it remains outstanding until either a processor becomes available for it or the job rescinds the request (e.g., if there is no longer a need for the extra processors).

Upon release of one or more processors (including job departure),

4. Scan the current queue of unsatisfied requests for processors. Assign a single processor to each job in the list that currently has no processors (i.e., to all waiting new arrivals). Then scan the list again, allocating the rest of the processors on an FCFS basis.

Analyses reported in [ZAHO90] and [MAJU88] suggest that for applications that can take advantage of dynamic scheduling, this approach is superior to gang scheduling or dedicated processor assignment. However, the overhead of this approach may negate this apparent performance advantage. Experience with actual systems is needed to prove the worth of dynamic scheduling.

10.2 REAL-TIME SCHEDULING

Background

Real-time computing is becoming an increasingly important discipline. The operating system, and in particular the scheduler, is perhaps the most important component of a real-time system. Examples of current applications of real-time systems include control of laboratory experiments, process control in industrial plants, robotics, air traffic control, telecommunications, and military command and control systems. Next-generation systems will include the autonomous land rover, controllers of robots with elastic joints, systems found in intelligent manufacturing, the space station, and undersea exploration.

Real-time computing may be defined as that type of computing in which the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced. We can define a real-time system by defining what is meant by a real-time process, or task.[3] In general, in a real-time system,

[3] As usual, terminology poses a problem, because various words are used in the literature with varying meanings. It is common for a particular process to operate under real-time constraints of a repetitive nature. That is, the process lasts for a long time and, during that time, performs some repetitive function in response to real-time events. Let us, for this section, refer to an individual function as a task. Thus, the process can be viewed as progressing through a sequence of tasks. At any given time, the process is engaged in a single task, and it is the process/task that must be scheduled.

some of the tasks are real-time tasks, and these have a certain degree of urgency to them. Such tasks are attempting to control or react to events that take place in the outside world. Because these events occur in "real time," a real-time task must be able to keep up with the events with which it is concerned. Thus, it is usually possible to associate a deadline with a particular task, where the deadline specifies either a start time or a completion time. Such a task may be classified as hard or soft. A **hard real-time task** is one that must meet its deadline; otherwise it will cause unacceptable damage or a fatal error to the system. A **soft real-time task** has an associated deadline that is desirable but not mandatory; it still makes sense to schedule and complete the task even if it has passed its deadline.

Another characteristic of real-time tasks is whether they are periodic or aperiodic. An **aperiodic task** has a deadline by which it must finish or start, or it may have a constraint on both start and finish time. In the case of a **periodic task**, the requirement may be stated as "once per period *T*" or "exactly *T* units apart."

Characteristics of Real-Time Operating Systems

Real-time operating systems can be characterized as having unique requirements in five general areas [MORG92]:

- Determinism
- Responsiveness
- User control
- Reliability
- Fail-soft operation

An operating system is **deterministic** to the extent that it performs operations at fixed, predetermined times or within predetermined time intervals. When multiple processes are competing for resources and processor time, no system will be fully deterministic. In a real-time operating system, process requests for service are dictated by external events and timings. The extent to which an operating system can deterministically satisfy requests depends first on the speed with which it can respond to interrupts and, second, on whether the system has sufficient capacity to handle all requests within the required time.

One useful measure of the ability of an operating system to function deterministically is the maximum delay from the arrival of a high-priority device interrupt to when servicing begins. In non-real-time operating systems, this delay may be in the range of tens to hundreds of milliseconds, while in real-time operating systems that delay may have an upper bound of anywhere from a few microseconds to a millisecond.

A related but distinct characteristic is **responsiveness**. Determinism is concerned with how long an operating system delays before acknowledging an interrupt. Responsiveness is concerned with how long, after acknowledgment, it takes an operating system to service the interrupt. Aspects of responsiveness include the following:

1. The amount of time required to initially handle the interrupt and begin execution of the interrupt service routine (ISR). If execution of the ISR requires a process switch, then the delay will be longer than if the ISR can be executed within the context of the current process.

2. The amount of time required to perform the ISR. This generally is dependent on the hardware platform.
3. The effect of interrupt nesting. If an ISR can be interrupted by the arrival of another interrupt, then the service will be delayed.

Determinism and responsiveness together make up the response time to external events. Response time requirements are critical for real-time systems, because such systems must meet timing requirements imposed by individuals, devices, and data flows external to the system.

User control is generally much broader in a real-time operating system than in ordinary operating systems. In a typical non-real-time operating system, the user either has no control over the scheduling function of the operating system or can only provide broad guidance, such as grouping users into more than one priority class. In a real-time system, however, it is essential to allow the user fine-grained control over task priority. The user should be able to distinguish between hard and soft tasks and to specify relative priorities within each class. A real-time system may also allow the user to specify such characteristics as the use of paging or process swapping, what processes must always be resident in main memory, what disk transfer algorithms are to be used, what rights the processes in various priority bands have, and so on.

Reliability is typically far more important for real-time systems than non-real-time systems. A transient failure in a non-real-time system may be solved by simply rebooting the system. A processor failure in a multiprocessor non-real-time system may result in a reduced level of service until the failed processor is repaired or replaced. But a real-time system is responding to and controlling events in real time. Loss or degradation of performance may have catastrophic consequences, ranging from financial loss to major equipment damage and even loss of life.

As in other areas the difference between a real-time and a non-real-time operating system is one of degree. Even a real-time system must be designed to respond to various failure modes. **Fail-soft operation** is a characteristic that refers to the ability of a system to fail in such a way as to preserve as much capability and data as possible. For example, a typical traditional UNIX system, when it detects a corruption of data within the kernel, issues a failure message on the system console, dumps the memory contents to disk for later failure analysis, and terminates execution of the system. In contrast, a real-time system will attempt either to correct the problem or minimize its effects while continuing to run. Typically, the system notifies a user or user process that it should attempt corrective action and then continues operation perhaps at a reduced level of service. In the event a shutdown is necessary, an attempt is made to maintain file and data consistency.

An important aspect of fail-soft operation is referred to as stability. A real-time system is stable if, in cases where it is impossible to meet all task deadlines, the system will meet the deadlines of its most critical, highest-priority tasks, even if some less critical task deadlines are not always met.

To meet the foregoing requirements, real-time operating systems typically include the following features [STAN89]:

- Fast process or thread switch
- Small size (with its associated minimal functionality)
- Ability to respond to external interrupts quickly

- Multitasking with interprocess communication tools such as semaphores, signals, and events
- Use of special sequential files that can accumulate data at a fast rate
- Preemptive scheduling based on priority
- Minimization of intervals during which interrupts are disabled
- Primitives to delay tasks for a fixed amount of time and to pause/resume tasks
- Special alarms and timeouts

The heart of a real-time system is the short-term task scheduler. In designing such a scheduler, fairness and minimizing average response time are not paramount. What is important is that all hard real-time tasks complete (or start) by their deadline and that as many as possible soft real-time tasks also complete (or start) by their deadline.

Most contemporary real-time operating systems are unable to deal directly with deadlines. Instead, they are designed to be as responsive as possible to real-time tasks so that, when a deadline approaches, a task can be quickly scheduled. From this point of view, real-time applications typically require deterministic response times in the several-millisecond to submillisecond span under a broad set of conditions; leading-edge applications—in simulators for military aircraft, for example—often have constraints in the range of 10 to 100 μs [ATLA89].

Figure 10.5 illustrates a spectrum of possibilities. In a preemptive scheduler that uses simple round-robin scheduling, a real-time task would be added to the ready queue to await its next time slice, as illustrated in Figure 10.5a. In this case, the scheduling time will generally be unacceptable for real-time applications. Alternatively, in a nonpreemptive scheduler, we could use a priority scheduling mechanism, giving real-time tasks higher priority. In this case, a real-time task that is ready would be scheduled as soon as the current process blocks or runs to completion (Figure 10.5b). This could lead to a delay of several seconds if a slow, low-priority task were executing at a critical time. Again, this approach is not acceptable. A more promising approach is to combine priorities with clock-based interrupts. Preemption points occur at regular intervals. When a preemption point occurs, the currently running task is preempted if a higher-priority task is waiting. This would include the preemption of tasks that are part of the operating system kernel. Such a delay may be on the order of several milliseconds (Figure 10.5c). While this last approach may be adequate for some real-time applications, it will not suffice for more demanding applications. In those cases, the approach that has been taken is sometimes referred to as immediate preemption. In this case, the operating system responds to an interrupt almost immediately, unless the system is in a critical-code lockout section. Scheduling delays for a real-time task can then be reduced to 100 μs or less.

Real-Time Scheduling

Real-time scheduling is one of the most active areas of research in computer science. In this subsection, we provide an overview of the various approaches to real-time scheduling and look at two popular classes of scheduling algorithms.

In a survey of real-time scheduling algorithms, [RAMA94] observes that the various scheduling approaches depend on (1) whether a system performs schedulability

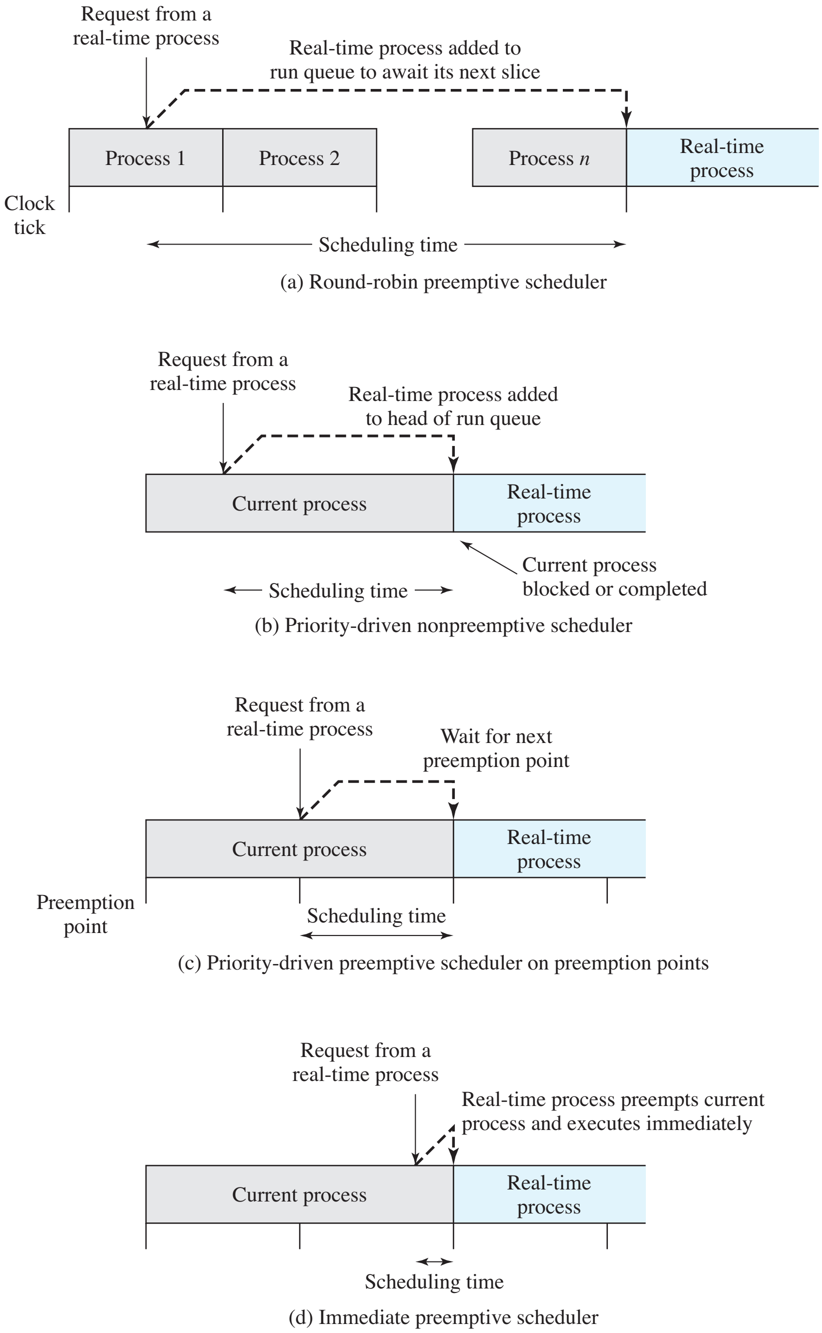

Request from a real-time process
Real-time process added to run queue to await its next slice
Process 1
Process 2
Process *n*
Real-time process
Clock tick
Scheduling time
(a) Round-robin preemptive scheduler
Request from a real-time process
Real-time process added to head of run queue
Current process
Real-time process
Current process blocked or completed
Scheduling time
(b) Priority-driven nonpreemptive scheduler
Request from a real-time process
Wait for next preemption point
Current process
Real-time process
Preemption point
Scheduling time
(c) Priority-driven preemptive scheduler on preemption points
Request from a real-time process
Real-time process preempts current process and executes immediately
Current process
Real-time process
Scheduling time
(d) Immediate preemptive scheduler

Figure 10.5 Scheduling of Real-Time Process

analysis; (2) if it does, whether it is done statically or dynamically; and (3) whether the result of the analysis itself produces a schedule or plan according to which tasks are dispatched at run time. Based on these considerations, the authors identify the following classes of algorithms:

- **Static table-driven approaches:** These perform a static analysis of feasible schedules of dispatching. The result of the analysis is a schedule that determines, at run time, when a task must begin execution.
- **Static priority-driven preemptive approaches:** Again, a static analysis is performed, but no schedule is drawn up. Rather, the analysis is used to assign priorities to tasks, so that a traditional priority-driven preemptive scheduler can be used.
- **Dynamic planning-based approaches:** Feasibility is determined at run time (dynamically) rather than offline prior to the start of execution (statically). An arriving task is accepted for execution only if it is feasible to meet its time constraints. One of the results of the feasibility analysis is a schedule or plan that is used to decide when to dispatch this task.
- **Dynamic best effort approaches:** No feasibility analysis is performed. The system tries to meet all deadlines and aborts any started process whose deadline is missed.

Static table-driven scheduling is applicable to tasks that are periodic. Input to the analysis consists of the periodic arrival time, execution time, periodic ending deadline, and relative priority of each task. The scheduler attempts to develop a schedule that enables it to meet the requirements of all periodic tasks. This is a predictable approach but one that is inflexible, because any change to any task requirements requires that the schedule be redone. Earliest-deadline-first or other periodic deadline techniques (discussed subsequently) are typical of this category of scheduling algorithms.

Static priority-driven preemptive scheduling makes use of the priority-driven preemptive scheduling mechanism common to most non-real-time multiprogramming systems. In a non-real-time system, a variety of factors might be used to determine priority. For example, in a time-sharing system, the priority of a process changes depending on whether it is processor bound or I/O bound. In a real-time system, priority assignment is related to the time constraints associated with each task. One example of this approach is the rate monotonic algorithm (discussed subsequently), which assigns static priorities to tasks based on the length of their periods.

With **dynamic planning-based scheduling**, after a task arrives, but before its execution begins, an attempt is made to create a schedule that contains the previously scheduled tasks as well as the new arrival. If the new arrival can be scheduled in such a way that its deadlines are satisfied and that no currently scheduled task misses a deadline, then the schedule is revised to accommodate the new task.

Dynamic best effort scheduling is the approach used by many real-time systems that are currently commercially available. When a task arrives, the system assigns a priority based on the characteristics of the task. Some form of deadline scheduling, such as earliest-deadline scheduling, is typically used. Typically, the tasks are aperiodic and so no static scheduling analysis is possible. With this type of scheduling, until a deadline arrives or until the task completes, we do not know whether a timing constraint will be met. This is the major disadvantage of this form of scheduling. Its advantage is that it is easy to implement.

Deadline Scheduling

Most contemporary real-time operating systems are designed with the objective of starting real-time tasks as rapidly as possible, and hence emphasize rapid interrupt handling and task dispatching. In fact, this is not a particularly useful metric in evaluating real-time operating systems. Real-time applications are generally not concerned with sheer speed but rather with completing (or starting) tasks at the most valuable times, neither too early nor too late, despite dynamic resource demands and conflicts, processing overloads, and hardware or software faults. It follows that priorities provide a crude tool and do not capture the requirement of completion (or initiation) at the most valuable time.

There have been a number of proposals for more powerful and appropriate approaches to real-time task scheduling. All of these are based on having additional information about each task. In its most general form, the following information about each task might be used:

- **Ready time:** Time at which task becomes ready for execution. In the case of a repetitive or periodic task, this is actually a sequence of times that is known in advance. In the case of an aperiodic task, this time may be known in advance, or the operating system may only be aware when the task is actually ready.
- **Starting deadline:** Time by which a task must begin.
- **Completion deadline:** Time by which task must be completed. The typical real-time application will either have starting deadlines or completion deadlines, but not both.
- **Processing time:** Time required to execute the task to completion. In some cases, this is supplied. In others, the operating system measures an exponential average (as defined in Chapter 9). For still other scheduling systems, this information is not used.
- **Resource requirements:** Set of resources (other than the processor) required by the task while it is executing.
- **Priority:** Measures relative importance of the task. Hard real-time tasks may have an "absolute" priority, with the system failing if a deadline is missed. If the system is to continue to run no matter what, then both hard and soft real-time tasks may be assigned relative priorities as a guide to the scheduler.
- **Subtask structure:** A task may be decomposed into a mandatory subtask and an optional subtask. Only the mandatory subtask possesses a hard deadline.

There are several dimensions to the real-time scheduling function when deadlines are taken into account: which task to schedule next, and what sort of preemption is allowed. It can be shown, for a given preemption strategy and using either starting or completion deadlines, that a policy of scheduling the task with the earliest deadline minimizes the fraction of tasks that miss their deadlines [BUTT99, HONG89, PANW88]. This conclusion holds both for single-processor and multiprocessor configurations.

The other critical design issue is that of preemption. When starting deadlines are specified, then a nonpreemptive scheduler makes sense. In this case, it would be the responsibility of the real-time task to block itself after completing the mandatory or critical portion of its execution, allowing other real-time starting deadlines to be satisfied. This fits the pattern of Figure 10.5b. For a system with completion deadlines, a preemptive strategy (Figure 10.5c or d) is most appropriate. For example, if task X is running and task Y is ready, there may be circumstances in which the only way to allow both X and Y to meet their completion deadlines is to preempt X, execute Y to completion, and then resume X to completion.

Animation: Periodic with Completion Deadline

As an example of scheduling periodic tasks with completion deadlines, consider a system that collects and processes data from two sensors, A and B. The deadline for collecting data from sensor A must be met every 20 ms, and that for B every 50 ms. It takes 10 ms, including operating system overhead, to process each sample of data from A and 25 ms to process each sample of data from B. Table 10.2 summarizes the execution profile of the two tasks. Figure 10.6 compares three scheduling techniques using the execution profile of Table 10.2. The first row of Figure 10.6 repeats the information in Table 10.2; the remaining three rows illustrate three scheduling techniques.

The computer is capable of making a scheduling decision every 10 ms.[4] Suppose that, under these circumstances, we attempted to use a priority scheduling

Table 10.2 Execution Profile of Two Periodic Tasks

Process	Arrival Time	Execution Time	Ending Deadline
A(1)	0	10	20
A(2)	20	10	40
A(3)	40	10	60
A(4)	60	10	80
A(5)	80	10	100
• • •	• • •	• • •	• • •
B(1)	0	25	50
B(2)	50	25	100
• • •	• • •	• • •	• • •

[4] This need not be on a 10-ms boundary if more than 10 ms has elapsed since the last scheduling decision.

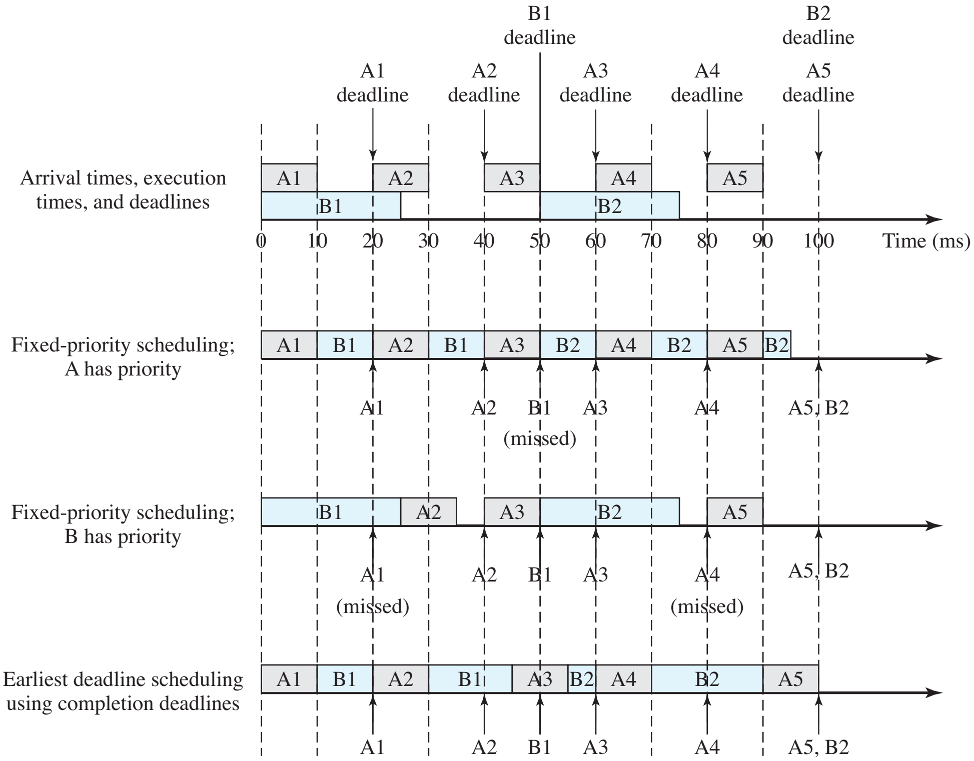

B1 deadline
B2 deadline
A1 deadline
A2 deadline
A3 deadline
A4 deadline
A5 deadline
Arrival times, execution times, and deadlines
A1
A2
A3
A4
A5
B1
B2
0
10
20
30
40
50
60
70
80
90
100
Time (ms)
Fixed-priority scheduling; A has priority
A1 B1 A2 B1 A3 B2 A4 B2 A5 B2
A1
A2
B1 (missed)
A3
A4
A5, B2
Fixed-priority scheduling; B has priority
B1 A2 A3 B2 A5
A1 (missed)
A2
B1
A3
A4 (missed)
A5, B2
Earliest deadline scheduling using completion deadlines
A1 B1 A2 B1 A3 B2 A4 B2 A5
A1
A2
B1
A3
A4
A5, B2

Figure 10.6 **Scheduling of Periodic Real-Time Tasks with Completion Deadlines (based on Table 10.2)**

scheme. The first two timing diagrams in Figure 10.6 show the result. If A has higher priority, the first instance of task B is only given 20 ms of processing time, in two 10-ms chunks, by the time its deadline is reached, and thus fails. If B is given higher priority, then A will miss its first deadline. The final timing diagram shows the use of earliest-deadline scheduling. At time $t = 0$, both A1 and B1 arrive. Because A1 has the earliest deadline, it is scheduled first. When A1 completes, B1 is given the processor. At $t = 20$, A2 arrives. Because A2 has an earlier deadline than B1, B1 is interrupted so that A2 can execute to completion. Then B1 is resumed at $t = 30$. At $t = 40$, A3 arrives. However, B1 has an earlier ending deadline and is allowed to execute to completion at $t = 45$. A3 is then given the processor and finishes at $t = 55$.

Animation: Aperiodic with Starting Deadline

In this example, by scheduling to give priority at any preemption point to the task with the nearest deadline, all system requirements can be met. Because the tasks are periodic and predictable, a static table-driven scheduling approach is used.

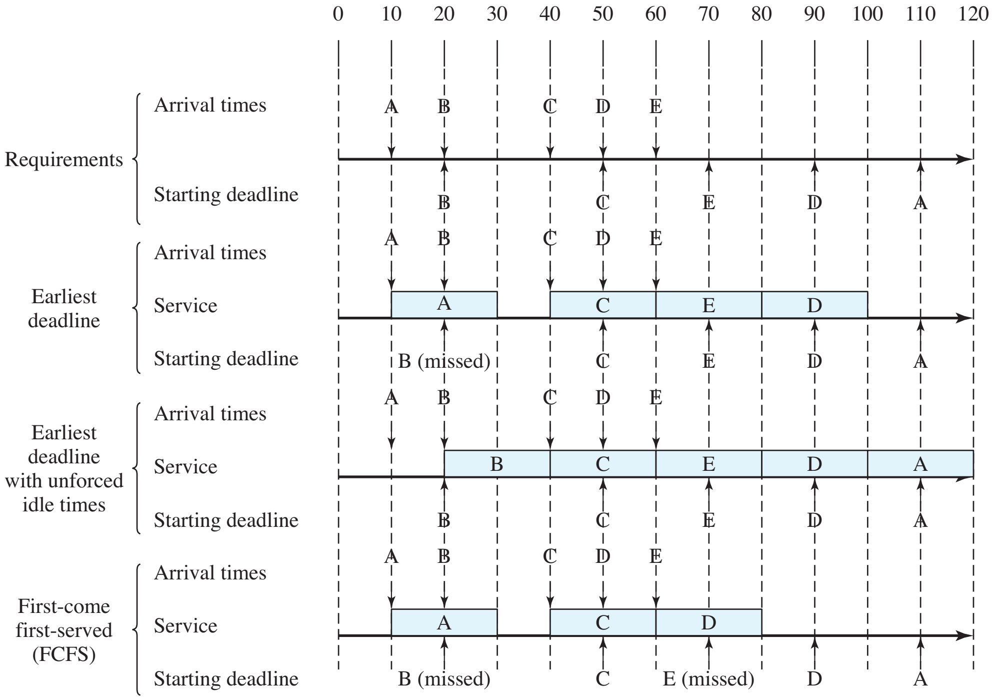


Figure 10.7 **Scheduling of Aperiodic Real-Time Tasks with Starting Deadlines**

Now consider a scheme for dealing with aperiodic tasks with starting deadlines. The top part of Figure 10.7 shows the arrival times and starting deadlines for an example consisting of five tasks each of which has an execution time of 20 ms. Table 10.3 summarizes the execution profile of the five tasks.

A straightforward scheme is to always schedule the ready task with the earliest deadline and let that task run to completion. When this approach is used in the example of Figure 10.7, note that although task B requires immediate service, the service is denied. This is the risk in dealing with aperiodic tasks, especially with starting deadlines. A refinement of the policy will improve performance if deadlines can be known in advance of the time that a task is ready. This policy, referred to as earliest deadline with unforced idle times, operates as follows: Always schedule the eligible task with the earliest deadline and let that task run to completion. An eligible

Table 10.3 Execution Profile of Five Aperiodic Tasks

Process	Arrival Time	Execution Time	Starting Deadline
A	10	20	110
B	20	20	20
C	40	20	50
D	50	20	90
E	60	20	70

task may not be ready, and this may result in the processor remaining idle even though there are ready tasks. Note that in our example the system refrains from scheduling task A even though that is the only ready task. The result is that, even though the processor is not used to maximum efficiency, all scheduling requirements are met. Finally, for comparison, the FCFS policy is shown. In this case, tasks B and E do not meet their deadlines.

Animation: Rate Monotonic Scheduling

Rate Monotonic Scheduling

One of the more promising methods of resolving multitask scheduling conflicts for periodic tasks is rate monotonic scheduling (RMS). The scheme was first proposed in [LIU73] but has only recently gained popularity [BRIA99, SHA94]. RMS assigns priorities to tasks on the basis of their periods.

For RMS, the highest-priority task is the one with the shortest period, the second highest-priority task is the one with the second shortest period, and so on. When more than one task is available for execution, the one with the shortest period is serviced first. If we plot the priority of tasks as a function of their rate, the result is a monotonically increasing function (Figure 10.8); hence the name, rate monotonic scheduling.

Figure 10.9 illustrates the relevant parameters for periodic tasks. The task's period, *T*, is the amount of time between the arrival of one instance of the task and the arrival of the next instance of the task. A task's rate (in Hertz) is simply the inverse of its period (in seconds). For example, a task with a period of 50 ms occurs at a rate of 20 Hz. Typically, the end of a task's period is also the task's hard deadline, although some tasks may have earlier deadlines. The execution (or computation) time, *C*, is the amount of processing time required for each occurrence of the task. It should be clear that in a uniprocessor system, the execution time must be no greater than the period (must have $C \leq T$). If a periodic task is always run to completion; that is, if

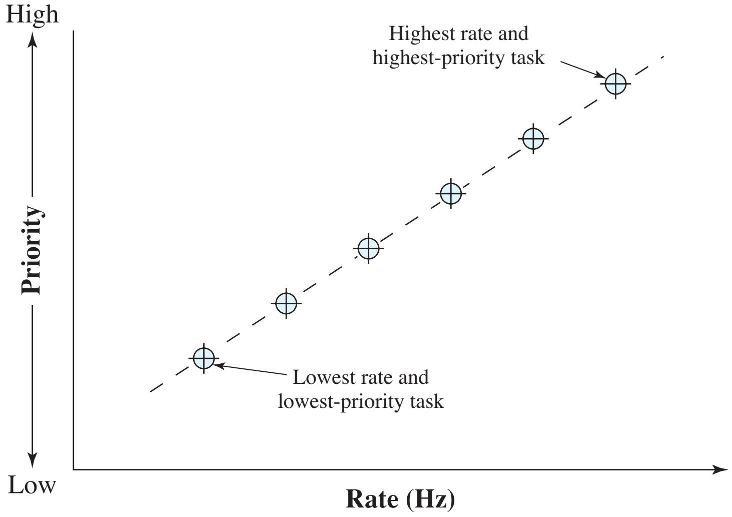


Figure 10.8 **A Task Set with RMS [WARR91]**

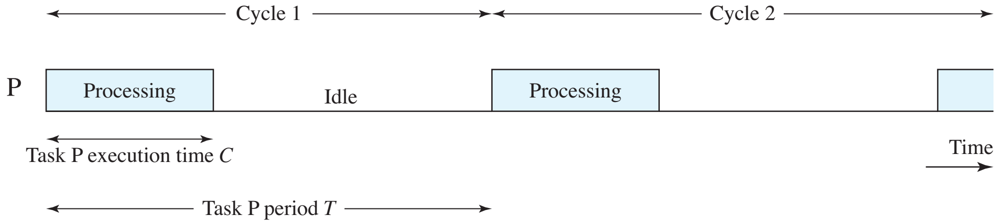


Figure 10.9 Periodic Task Timing Diagram

no instance of the task is ever denied service because of insufficient resources, then the utilization of the processor by this task is $U = C/T$. For example, if a task has a period of 80 ms and an execution time of 55 ms, its processor utilization is $55/80 = 0.6875$.

One measure of the effectiveness of a periodic scheduling algorithm is whether or not it guarantees that all hard deadlines are met. Suppose that we have n tasks, each with a fixed period and execution time. Then for it to be possible to meet all deadlines, the following inequality must hold:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \dots + \frac{C_n}{T_n} \leq 1 \qquad \textbf{(10.1)}$$

The sum of the processor utilizations of the individual tasks cannot exceed a value of 1, which corresponds to total utilization of the processor. Equation (10.1) provides a bound on the number of tasks that a perfect scheduling algorithm can successfully schedule. For any particular algorithm, the bound may be lower. For RMS, it can be shown that the following inequality holds:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \dots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1) \qquad \textbf{(10.2)}$$

Table 10.4 gives some values for this upper bound. As the number of tasks increases, the scheduling bound converges to $\ln 2 \approx 0.693$.

Table 10.4 Value of the RMS Upper Bound

n	$n(2^{1/n} - 1)$
1	1.0
2	0.828
3	0.779
4	0.756
5	0.743
6	0.734
•	•
•	•
•	•
∞	$\ln 2 \approx 0.693$

As an example, consider the case of three periodic tasks, where $U_i = C_i/T_i$:

- **Task P$_1$:** $C_1 = 20$; $T_1 = 100$; $U_1 = 0.2$
- **Task P$_2$:** $C_2 = 40$; $T_2 = 150$; $U_2 = 0.267$
- **Task P$_3$:** $C_3 = 100$; $T_3 = 350$; $U_3 = 0.286$

The total utilization of these three tasks is 0.2 + 0.267 + 0.286 = 0.753. The upper bound for the schedulability of these three tasks using RMS is

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} \leq 3(2^{1/3} - 1) = 0.779$$

Because the total utilization required for the three tasks is less than the upper bound for RMS ($0.753 < 0.779$), we know that if RMS is used, all tasks will be successfully scheduled.

It can also be shown that the upper bound of Equation (10.1) holds for earliest deadline scheduling. Thus, it is possible to achieve greater overall processor utilization and therefore accommodate more periodic tasks with earliest deadline scheduling. Nevertheless, RMS has been widely adopted for use in industrial applications. [SHA91] offers the following explanation:

1. The performance difference is small in practice. The upper bound of Equation (10.2) is a conservative one and, in practice, utilization as high as 90% is often achieved.
2. Most hard real-time systems also have soft real-time components, such as certain noncritical displays and built-in self tests that can execute at lower priority levels to absorb the processor time that is not used with RMS scheduling of hard real-time tasks.
3. Stability is easier to achieve with RMS. When a system cannot meet all deadlines because of overload or transient errors, the deadlines of essential tasks need to be guaranteed provided that this subset of tasks is schedulable. In a static priority assignment approach, one only needs to ensure that essential tasks have relatively high priorities. This can be done in RMS by structuring essential tasks to have short periods or by modifying the RMS priorities to account for essential tasks. With earliest deadline scheduling, a periodic task's priority changes from one period to another. This makes it more difficult to ensure that essential tasks meet their deadlines.

Priority Inversion

Priority inversion is a phenomenon that can occur in any priority-based preemptive scheduling scheme but is particularly relevant in the context of real-time scheduling. The best-known instance of priority inversion involved the Mars Pathfinder mission. This rover robot landed on Mars on July 4, 1997 and began gathering and transmitting voluminous data back to Earth. But a few days into the mission, the lander software began experiencing total system resets, each resulting in losses of data. After much effort by the Jet Propulsion Laboratory (JPL) team that built the Pathfinder, the problem was traced to priority inversion [JONE97].

In any priority scheduling scheme, the system should always be executing the task with the highest priority. **Priority inversion** occurs when circumstances within the system force a higher-priority task to wait for a lower-priority task. A simple example of priority inversion occurs if a lower-priority task has locked a resource (such as a device or a binary semaphore) and a higher-priority task attempts to lock that same resource. The higher-priority task will be put in a blocked state until the resource is available. If the lower-priority task soon finishes with the resource and releases it, the higher-priority task may quickly resume and it is possible that no real-time constraints are violated.

A more serious condition is referred to as an **unbounded priority inversion**, in which the duration of a priority inversion depends not only on the time required to handle a shared resource, but also on the unpredictable actions of other unrelated tasks as well. The priority inversion experienced in the Pathfinder software was unbounded and serves as a good example of the phenomenon. Our discussion follows that of [TIME02]. The Pathfinder software included the following three tasks, in decreasing order of priority:

T_1: Periodically checks the health of the spacecraft systems and software

T_2: Processes image data

T_3: Performs an occasional test on equipment status

After T_1 executes, it reinitializes a timer to its maximum value. If this timer ever expires, it is assumed that the integrity of the lander software has somehow been compromised. The processor is halted, all devices are reset, the software is completely reloaded, the spacecraft systems are tested, and the system starts over. This recovery sequence does not complete until the next day. T_1 and T_3 share a common data structure, protected by a binary semaphore `s`. Figure 10.10a shows the sequence that caused the priority inversion:

t_1: T_3 begins executing.

t_2: T_3 locks semaphore `s` and enters its critical section.

t_3: T_1, which has a higher priority than T_3, preempts T_3 and begins executing.

t_4: T_1 attempts to enter its critical section but is blocked because the semaphore is locked by T_3; T_3 resumes execution in its critical section.

t_5: T_2, which has a higher priority than T_3, preempts T_3 and begins executing.

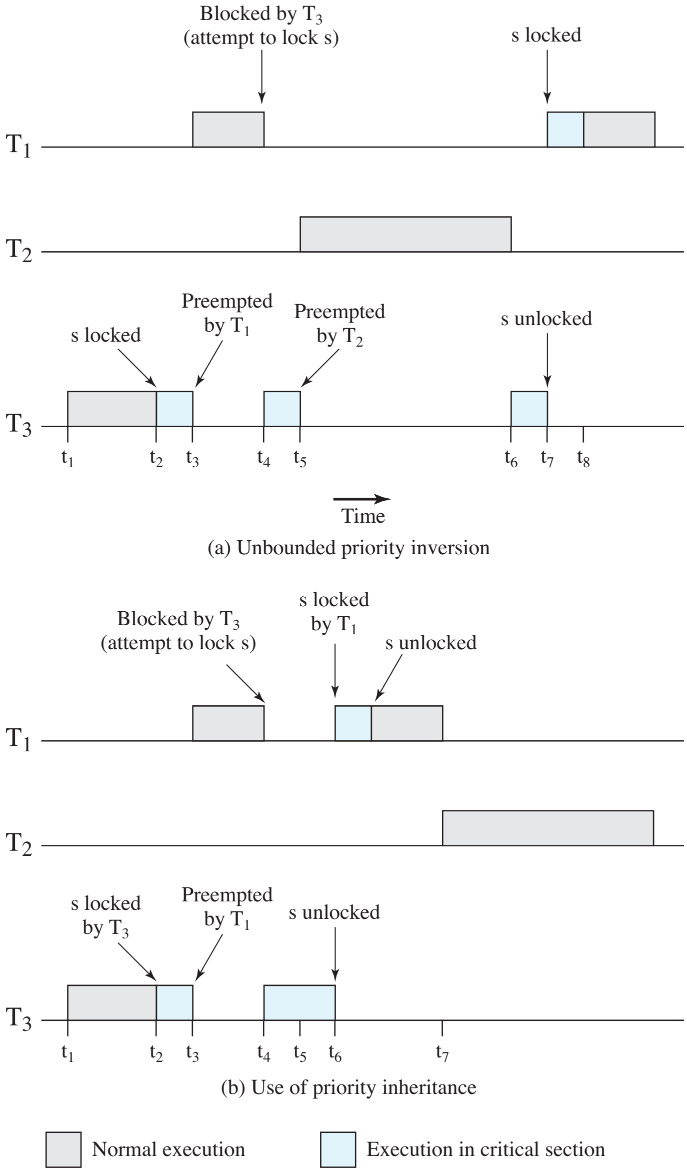
t_6: T_2 is suspended for some reason unrelated to T_1 and T_3; T_3 resumes.

t_7: T_3 leaves its critical section and unlocks the semaphore. T_1 preempts T_3, locks the semaphore and enters its critical section.

In this set of circumstances, T_1 must wait for both T_3 and T_2 to complete and fails to reset the timer before it expires.

In practical systems, two alternative approaches are used to avoid unbounded priority inversion: priority inheritance protocol and priority ceiling protocol.

The basic idea of **priority inheritance** is that a lower-priority task inherits the priority of any higher-priority task pending on a resource they share. This priority change takes place as soon as the higher-priority task blocks on the resource; it should end when the resource is released by the lower-priority task. Figure 10.10b



(a) Unbounded priority inversion

(b) Use of priority inheritance

Figure 10.10 Priority Inversion

shows that priority inheritance resolves the problem of unbounded priority inversion illustrated in Figure 10.10a. The relevant sequence of events is as follows:

t_1: T_3 begins executing.

t_2: T_3 locks semaphore `s` and enters its critical section.

t_3: T_1, which has a higher priority than T_3, preempts T_3 and begins executing.

t_4: T_1 attempts to enter its critical section but is blocked because the semaphore is locked by T_3. T_3 is immediately and temporarily assigned the same priority as T_1. T_3 resumes execution in its critical section.

t_5: T_2 is ready to execute but, because T_3 now has a higher priority, T_2 is unable to preempt T_3.

t_6: T_3 leaves its critical section and unlocks the semaphore: its priority level is downgraded to its previous default level. T_1 preempts T_3, locks the semaphore, and enters its critical section.

t_7: T_1 is suspended for some reason unrelated to T_2, and T_2 begins executing.

This was the approach taken to solving the Pathfinder problem.

In the **priority ceiling** approach, a priority is associated with each resource. The priority assigned to a resource is one level higher than the priority of its highest-priority user. The scheduler then dynamically assigns this priority to any task that accesses the resource. Once the task finishes with the resource, its priority returns to normal.

10.3 LINUX SCHEDULING

For Linux 2.4 and earlier, Linux provided a real-time scheduling capability coupled with a scheduler for non-real-time processes that made use of the traditional UNIX scheduling algorithm described in Section 9.3. Linux 2.6 includes essentially the same real-time scheduling capability as previous releases and a substantially revised scheduler for non-real-time processes. We examine these two areas in turn.

Real-Time Scheduling

The three Linux scheduling classes are

- **SCHED_FIFO:** First-in-first-out real-time threads
- **SCHED_RR:** Round-robin real-time threads
- **SCHED_OTHER:** Other, non-real-time threads

Within each class, multiple priorities may be used, with priorities in the real-time classes higher than the priorities for the SCHED_OTHER class. The default values are as follows: Real-time priority classes range from 0 to 99 inclusively, and SCHED_OTHER classes range from 100 to 139. A lower number equals a higher priority.

For FIFO threads, the following rules apply:

1. The system will not interrupt an executing FIFO thread except in the following cases:
   a. Another FIFO thread of higher priority becomes ready.
   b. The executing FIFO thread becomes blocked waiting for an event, such as I/O.
   c. The executing FIFO thread voluntarily gives up the processor following a call to the primitive sched_yield.
2. When an executing FIFO thread is interrupted, it is placed in the queue associated with its priority.

10.8 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

aperiodic task
deadline scheduling
deterministic operating system
fail-soft operation
gang scheduling
granularity
hard real-time task
load sharing
periodic task
priority inversion
rate monotonic scheduling
real-time operating system
real-time scheduling
responsiveness
soft real-time task
thread scheduling
unbounded priority inversion

Review Questions

10.1 List and briefly define five different categories of synchronization granularity.

10.2 List and briefly define four techniques for thread scheduling.

10.3 List and briefly define three versions of load sharing.

10.4 What is the difference between hard and soft real-time tasks?

10.5 What is the difference between periodic and aperiodic real-time tasks?

10.6 List and briefly define five general areas of requirements for a real-time operating system.

10.7 List and briefly define four classes of real-time scheduling algorithms.

10.8 What items of information about a task might be useful in real-time scheduling?

Problems

10.1 Consider a set of three periodic tasks with the execution profiles of Table 10.5. Develop scheduling diagrams similar to those of Figure 10.6 for this set of tasks.

Table 10.5 Execution Profile for Problem 10.1

Process	Arrival Time	Execution Time	Ending Deadline
A(1)	0	10	20
A(2)	20	10	40
• • •	• • •	• • •	• • •
B(1)	0	10	50
B(2)	50	10	100
• • •	• • •	• • •	• • •
C(1)	0	15	50
C(2)	50	15	100
• • •	• • •	• • •	• • •

10.2 Consider a set of five aperiodic tasks with the execution profiles of Table 10.6. Develop scheduling diagrams similar to those of Figure 10.7 for this set of tasks.

Table 10.6 Execution Profile for Problem 10.2

Process	Arrival Time	Execution Time	Starting Deadline
A	10	20	100
B	20	20	30
C	40	20	60
D	50	20	80
E	60	20	70

10.3 Least laxity first (LLF) is a real-time scheduling algorithm for periodic tasks. Slack time, or laxity, is the amount of time between when a task would complete if it started now and its next deadline. This is the size of the available scheduling window. Laxity can be expressed as

$$\text{laxity} = (\text{deadline time}) - (\text{current time}) - (\text{processor time nedded})$$

LLF assigns selects the task with the minimum laxity to execute next. If two or more tasks have the same minimum laxity value, they are serviced on a FCFS basis.

a. Suppose a task currently has a laxity of *t*. By how long may the scheduler delay starting this task and still meet its deadline?

b. Suppose a task currently has a laxity of 0. What does this mean?

c. What does it mean if a task has negative laxity?

d. Consider a set of three periodic tasks with the execution profiles of Table 10.7a. Develop scheduling diagrams similar to those of Figure 10.5 for this set of tasks that compare rate monotonic, earliest deadline first, and LLF. Assume preemption may occur at 5-ms intervals. Comment on the results.

Table 10.7 Execution Profiles for Problems 10.3 through 10.6

(a) Light load

Task	Period	Execution Time
A	6	2
B	8	2
C	12	3

(b) Heavy load

Task	Period	Execution Time
A	6	2
B	8	5
C	12	3

10.4 Repeat Problem 10.3d for the execution profiles of Table 10.7b. Comment on the results.

10.5 Maximum urgency first (MUF) is a real-time scheduling algorithm for periodic tasks. Each task is assigned an urgency that is defined as a combination of two fixed priorities and one dynamic priority. One of the fixed priorities, the criticality, has precedence over the dynamic priority. Meanwhile, the dynamic priority has precedence over the other fixed priority, called the user priority. The dynamic priority is inversely proportional to the laxity of a task. MUF can be explained as follows. First, tasks are ordered from shortest to longest period. Define the critical task set as the first *N* tasks such that worst-case processor utilization does not exceed 100%. Among critical set tasks that are ready, the scheduler selects the task with the least laxity. If no critical set tasks are ready, the schedule chooses among the noncritical tasks the one with the least laxity. Ties are broken through an optional user priority and then by FCFS. Repeat Problem 10.3d, adding MUF to the diagrams. Assume that user-defined priorities are A highest, B next, C lowest. Comment on the results.

10.6 Repeat Problem 10.4, adding MUF to the diagrams. Comment on the results.

10.7 This problem demonstrates that, although Equation (10.2) for rate monotonic scheduling is a sufficient condition for successful scheduling, it is not a necessary condition [that is, sometimes successful scheduling is possible even if Equation (10.2) is not satisfied].

a. Consider a task set with the following independent periodic tasks:

- Task P_1: $C_1 = 20$; $T_1 = 100$
- Task P_2: $C_2 = 30$; $T_2 = 145$

Can these tasks be successfully scheduled using rate monotonic scheduling?

b. Now add the following task to the set:

- Task P_3: $C_3 = 68$; $T_3 = 150$

Is Equation (10.2) satisfied?

c. Suppose that the first instance of the preceding three tasks arrives at time $t = 0$. Assume that the first deadline for each task is the following:

$$D_1 = 100; \quad D_2 = 145; \quad D_3 = 150$$

Using rate monotonic scheduling, will all three deadlines be met? What about deadlines for future repetitions of each task?

10.8 Draw a diagram similar to that of Figure 10.10b that shows the sequence events for this same example using priority ceiling.