CHAPTER 6

Priority-Driven Scheduling of Periodic Tasks

This chapter describes well-known priority-driven algorithms for scheduling periodic tasks on a processor and examines the merits and limitations of these algorithms. It is the first of three chapters devoted to priority-driven scheduling on one processor of applications characterizable by the periodic task model. The simplifying assumptions made in this chapter are that

1. the tasks are independent and
2. there are no aperiodic and sporadic tasks.

In the next chapter we will describe ways to integrate the scheduling of aperiodic and sporadic tasks with periodic tasks. In Chapter 8, we will introduce other resources and discuss the effects of resource contention. We will also describe resource access-control protocols designed to keep bounded the delay in job completion caused by resource contentions.

In most of this chapter, we will confine our attention to the case where every job is ready for execution as soon as it is released, can be preempted at any time, and never suspends itself. Scheduling decisions are made immediately upon job releases and completions. Moreover, the context switch overhead is negligibly small compared with execution times of the tasks, and the number of priority levels is unlimited. At the end of the chapter, we will remove these restrictions and discuss the effects of these and other practical factors.

Since there will be no ambiguity in this chapter, we often refer to periodic tasks simply as tasks. We now remove the restrictive assumption on fixed interrelease times which we made in Chapter 5. Hereafter we again use the term *period* to mean *the minimum interrelease time* of jobs in a task, as the term was defined in Chapter 3.

Most of our discussion is in terms of a fixed number of periodic tasks. The assumption here is that some protocol is used to regulate changes in the number or parameters of periodic tasks. In particular, when an application creates a new task, the application first requests the scheduler to add the new task by providing the scheduler with relevant parameters of the task, including its period, execution time, and relative deadline. Based on these parameters, the scheduler does an acceptance test on the new periodic task. In this test, the scheduler uses one of the methods described in this chapter to determine whether the new task can be

feasibly scheduled with all the other existing tasks in the system. It accepts and adds the new task to the system only if the new task and all other existing tasks can be feasibly scheduled. Otherwise, the scheduler rejects the new task. (We assume that the application system deals with the rejections of new task requests and recovers in an application-dependent way.) When tasks are independent, the scheduler can delete any task and add an acceptable task at any time without causing any missed deadline. Later in Chapter 8 and 9, we will consider periodic tasks that share resources and have precedence constraints. Such tasks cannot be added and deleted at any time. We will describe a mode change protocol that tries to complete task additions and deletions as soon as possible.

6.1 STATIC ASSUMPTION

Again, these three chapters focus on uniprocessor systems. You may question why we examine the problems of uniprocessor scheduling and synchronization in so much detail when most real-time systems today and in the future contain more than one processor. To answer this question, we recall that a multiprocessor priority-driven system is either dynamic or static. In a static system, all the tasks are partitioned into subsystems. Each subsystem is assigned to a processor, and tasks on each processor are scheduled by themselves. In contrast, in a dynamic system, jobs ready for execution are placed in one common priority queue and dispatched to processors for execution as the processors become available.

The dynamic approach should allow the processors to be more fully utilized on average as the workload fluctuates. Indeed, it may perform well most of the time. However, in the worst case, the performance of priority-driven algorithms can be unacceptably poor. A simple example [DhLi] demonstrates this fact. The application system contains $m + 1$ independent periodic tasks. The first m tasks T_i, for $i = 1, 2, \ldots m$, are identical. Their periods are equal to 1, and their execution times are equal to 2ε, where ε is a small number. The period of the last task T_{m+1} is $1 + \varepsilon$, and its execution time is 1. The tasks are in phase. Their relative deadlines are equal to their periods. Suppose that the priorities of jobs in these tasks are assigned on an EDF basis. The first job $J_{m+1,1}$ in T_{m+1} has the lowest priority because it has the latest deadline. Figure 6–1 shows an EDF schedule of these jobs if the jobs are dispatched and scheduled dynamically on m processors. We see that $J_{m+1,1}$ does not complete until $1 + 2\varepsilon$ and, hence, misses its deadline. The total utilization U of these $m + 1$ periodic tasks is $2m\varepsilon + 1/(1 + \varepsilon)$. In the limit as ε approaches zero, U approaches 1, and yet the system remains unschedulable. We would get the same infeasible schedule if we assigned the same priority to all the jobs in each task according to the period of the task: the shorter the period, the higher the priority. On the other hand, this system can be feasibly scheduled statically. As long as the total utilization of the first m tasks, $2m\varepsilon$, is equal to or less than 1, this system can be feasibly scheduled on two processors if we put T_{m+1} on one processor and the other tasks on the other processor and schedule the task(s) on each processor according to either of these priority-driven algorithms.

6.2

It is arguable that the poor behavior of dynamic systems occurs only for some pathological system configurations, and some other algorithms may perform well even for the pathological cases. In most cases, the performance of dynamic systems is superior to static systems. The more troublesome problem with dynamic systems is the fact that we often do not know how to determine their worst-case and best-case performance. The theories and algorithms

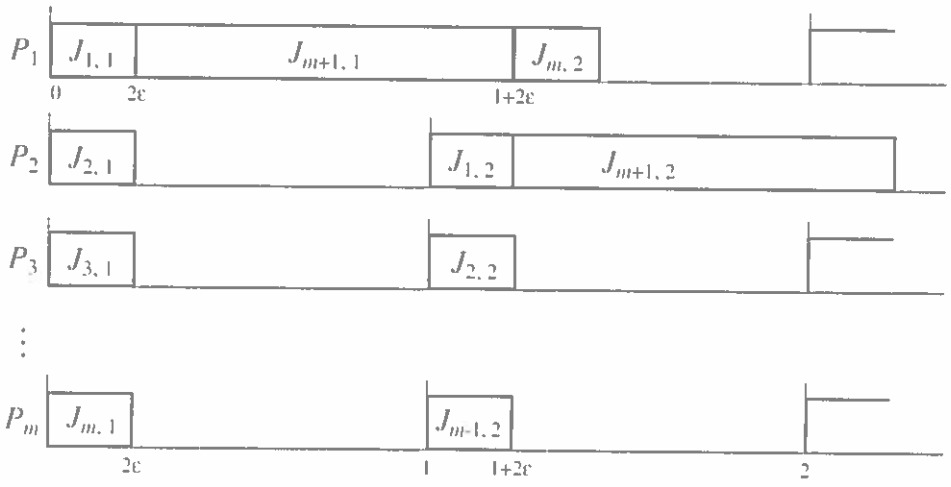

FIGURE 6–1 A dynamic EDF schedule on m processors.

presented in this and subsequent chapters make it possible for us to validate efficiently, robustly, and accurately the timing constraints of static real-time systems characterizable by the periodic task model. There are very few similar theories and algorithms for dynamic systems to date, as we discussed in Section 4.8. Until they become available, the only way to validate a dynamic system is by simulating and testing the system. This is prohibitively time consuming if done exhaustively, or unreliable if the coverage of the test is incomplete.

For these reasons, most hard real-time systems built and in use to date and in the near future are static. In the special case when tasks in a static system are independent, we can consider the tasks on each processor independently of the tasks on the other processors. The problem of scheduling in multiprocessor and distributed systems is reduced to that of uniprocessor scheduling. In general, tasks may have data and control dependencies and may share resources on different processors. As you will see in Chapter 9, uniprocessor algorithms and protocols can easily be extended to synchronize tasks and control their accesses to global resources in multiprocessor and distributed environments.

6.2 FIXED-PRIORITY VERSUS DYNAMIC-PRIORITY ALGORITHMS

As stated in Section 4.9, a priority-driven scheduler (i.e., a scheduler which schedules jobs according to some priority-driven algorithm) is an on-line scheduler. It does not precompute a schedule of the tasks. Rather, it assigns priorities to jobs after they are released and places the jobs in a ready job queue in priority order. When preemption is allowed at any time, a scheduling decision is made whenever a job is released or completed. At each scheduling decision time, the scheduler updates the ready job queue and then schedules and executes the job at the head of the queue.

Priority-driven algorithms differ from each other in how priorities are assigned to jobs. We classify algorithms for scheduling periodic tasks into two types: fixed priority and dynamic priority. A *fixed-priority* algorithm assigns the same priority to all the jobs in each task. In

other words, the priority of each periodic task is fixed relative to other tasks. In contrast, a *dynamic-priority* algorithm assigns different priorities to the individual jobs in each task. Hence the priority of the task with respect to that of the other tasks changes as jobs are released and completed. This is why this type of algorithm is said to be "dynamic."

As you will see later, most real-time scheduling algorithms of practical interest assign fixed priorities to individual jobs. The priority of each job is assigned upon its release when it is inserted into the ready job queue. Once assigned, the priority of the job relative to other jobs in the ready job queue does not change. In other words, at the level of individual jobs, the priorities are fixed, even though the priorities at the task level are variable. Indeed, we have three categories of algorithms: fixed-priority algorithms, task-level dynamic-priority (and job-level fixed-priority) algorithms, and job-level (and task-level) dynamic algorithms. Except where stated otherwise, by dynamic-priority algorithms, we mean task-level dynamic-priority (and job-level fixed-priority) algorithms.

6.2.1 Rate-Monotonic and Deadline-Monotonic Algorithms

A well-known fixed-priority algorithm is the *rate-monotonic* algorithm [LiLa]. This algorithm assigns priorities to tasks based on their periods: the shorter the period, the higher the priority. The *rate* (of job releases) of a task is the inverse of its period. Hence, the higher its rate, the higher its priority. We will refer to this algorithm as the RM algorithm for short and a schedule produced by the algorithm as an RM schedule.

Figure 6–2 gives two examples. Figure 6–2(a) shows the RM schedule of the system whose cyclic schedule is in Figure 5-8. This system contains three tasks: $T_1 = (4, 1)$, $T_2 = (5, 2)$, and $T_3 = (20, 5)$. The priority of T_1 is the highest because its rate is the highest (or equivalently, its period is the shortest). Each job in this task is placed at the head of the priority

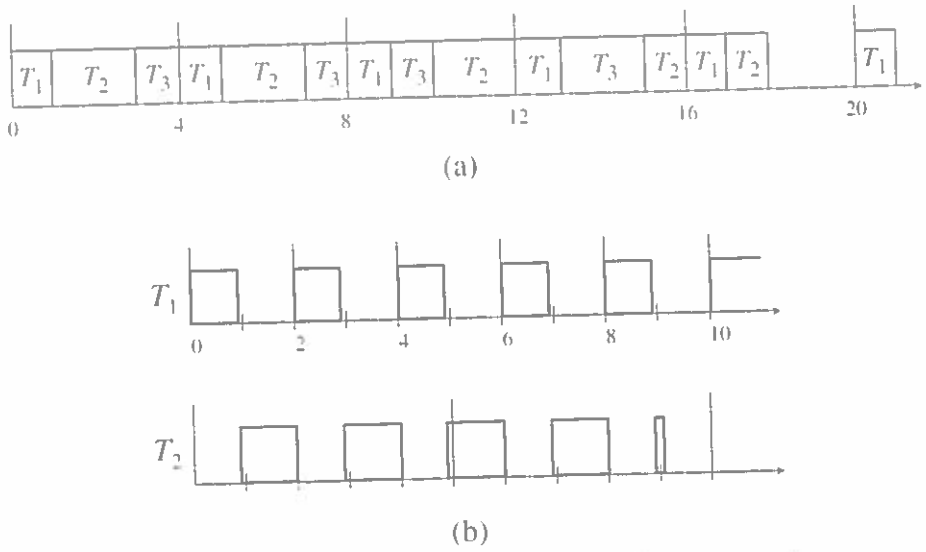

FIGURE 6–2 Examples of RM schedules. (a) RM schedule of $T_1 = (4, 1)$, $T_2 = (5, 2)$, and $T_3 = (20, 5)$. (b) RM schedule of $T_1 = (2, 0.9)$ and $T_2 = (5, 2.3)$.

queue and is executed as soon as the job is released. T_2 has the next highest priority. Its jobs execute in the background of T_1. For this reason, the execution of the first job in T_2 is delayed until the first job in T_1 completes, and the fourth job in T_2 is preempted at time 16 when the fifth job in T_1 is released. Similarly, T_3 executes in the background of T_1 and T_2; the jobs in T_3 execute only when there is no job in the higher-priority tasks ready for execution. Since there is always at least one job ready for execution until time 18, the processor never idles until that time.

The schedule in Figure 6–2(b) is for the tasks $T_1 = (2, 0.9)$ and $T_2 = (5, 2.3)$. The tasks are in phase. Here, we represent the schedule in a different form. Instead of using one time line, sometimes called a Gantt chart, to represent a schedule on a processor as we have done thus far, we use a time line for each task. Each time line is labeled at the left by the name of a task; the time line shows the intervals during which the task executes. According to the RM algorithm, task T_1 has a higher-priority than task T_2. Consequently, every job in T_1 is scheduled and executed as soon as it is released. The jobs in T_2 are executed in the background of T_1.

Another well-known fixed-priority algorithm is the *deadline-monotonic algorithm*, called the DM algorithm hereafter. This algorithm assigns priorities to tasks according their relative deadlines: the shorter the relative deadline, the higher the priority. Figure 6–3 gives an example. The system consists of three tasks. They are $T_1 = (50, 50, 25, 100)$, $T_2 = (0, 62.5, 10, 20)$, and $T_3 = (0, 125, 25, 50)$. Their utilizations are 0.5, 0.16, and 0.2, respectively. The total utilization is 0.86. According to the DM algorithm, T_2 has the highest priority because its relative deadline 20 is the shortest among the tasks. T_1, with a relative deadline of 100, has the lowest priority. The resultant DM schedule is shown in Figure 6–3(a). According to this schedule, all the tasks can meet their deadlines.

Clearly, when the relative deadline of every task is proportional to its period, the RM and DM algorithms are identical. When the relative deadlines are arbitrary, the DM algorithm performs better in the sense that it can sometimes produce a feasible schedule when the RM algorithm fails, while the RM algorithm always fails when the DM algorithm fails. The example above illustrates this fact. Figure 6–3(b) shows the RM schedule of the three tasks which Figure 6–3(a) has shown to be feasible when scheduled deadline-monotonically. According to the RM algorithm, T_1 has the highest priority, and T_3 has the lowest priority. We see that because the priorities of the tasks with short relative deadlines are too low, these tasks cannot meet all their deadlines.

6.2.2 Well-Known Dynamic Algorithms

The EDF algorithm assigns priorities to individual jobs in the tasks according to their absolute deadlines; it is a dynamic-priority algorithm. Figure 6–4 shows the EDF schedule of the two tasks T_1 and T_2 whose RM schedule is depicted in Figure 6–2(b).

- At time 0, the first jobs $J_{1,1}$ and $J_{2,1}$ of both tasks are ready. The (absolute) deadline of $J_{1,1}$ is 2 while the deadline of $J_{2,1}$ is 5. Consequently, $J_{1,1}$ has a higher priority and executes. When $J_{1,1}$ completes, $J_{2,1}$ begins to execute.
- At time 2, $J_{1,2}$ is released, and its deadline is 4, earlier than the deadline of $J_{2,1}$. Hence, $J_{1,2}$ is placed ahead of $J_{2,1}$ in the ready job queue. $J_{1,2}$ preempts $J_{2,1}$ and executes.
- At time 2.9, $J_{1,2}$ completes. The processor then executes $J_{2,1}$.

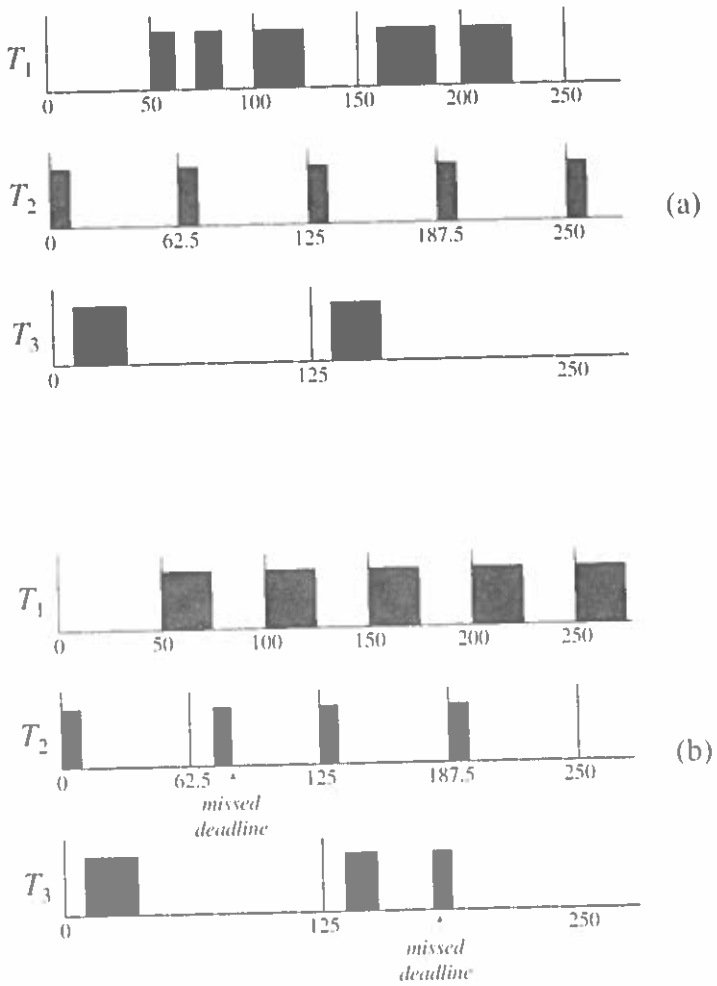

FIGURE 6–3 Fixed-priority schedules of $T_1 = (50, 50, 25, 100)$, $T_2 = (0, 62.5, 10, 20)$ and $T_3 = (0, 125, 25, 50)$.

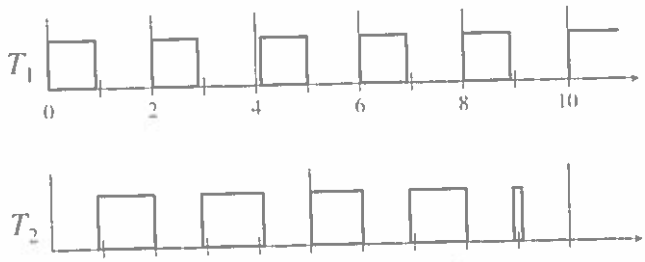

FIGURE 6–4 An earliest-deadline-first schedule of $(2, 0.9)$ and $(5, 2.3)$.

- At time 4, $J_{1,3}$ is released; its deadline is 6, which is later than the deadline of $J_{2,1}$. Hence, the processor continues to execute $J_{2,1}$.
- At time 4.1, $J_{2,1}$ completes, the processor starts to execute $J_{1,3}$, and so on.

We note that the priority of T_1 is higher than the priority of T_2 from time 0 until time 4.0. T_2 starts to have a higher priority at time 4. When the job $J_{2,2}$ is released, T_2 again has a lower priority. Hence, the EDF algorithm is a task-level dynamic-priority algorithm. On the other hand, once a job is placed in the ready job queue according to the priority assigned to it, its order with respect to other jobs in the queue remains fixed. In other words, the EDF algorithm is a job-level fixed-priority algorithm. Later on in Chapter 8, we will make use of this property.

Another well-known dynamic-priority algorithm is the *Least-Slack-Time-First* (LST) algorithm. You recall that at time t, the slack of a job whose remaining execution time (i.e., the execution of its remaining portion) is x and whose deadline is d is equal to $d - t - x$. The scheduler checks the slacks of all the ready jobs each time a new job is released and orders the new job and the existing jobs on the basis of their slacks: the smaller the slack, the higher the priority.

Coincidentally, the schedule of T_1 and T_2 in the above example produced by the LST algorithm happens to be identical to the EDF schedule in Figure 6–4. In general, however, the LST schedule of a system may differ in a fundamental way from the EDF schedule. To illustrate, we consider a more complicated system that consists of three tasks: $T_1 = (2, 0.8)$, $T_2 = (5, 1.5)$, and $T_3 = (5.1, 1.5)$. When the first jobs $J_{1,1}$, $J_{2,1}$ and $J_{3,1}$ are released at time 0, their slacks are 1.2, 3.5, and 3.6, respectively. $J_{1,1}$ has the highest priority, and $J_{3,1}$ has the lowest. At time 0.8, $J_{1,1}$ completes, and $J_{2,1}$ executes. When $J_{1,2}$ is released at time 2, its slack is 1.2, while the slacks of $J_{2,1}$ and $J_{3,1}$ become 2.7 and 1.6, respectively. Hence, $J_{1,2}$ has the highest priority, but now $J_{3,1}$ has a higher-priority than $J_{2,1}$! From this example, we see that the LST algorithm is a job-level dynamic-priority algorithm, while the EDF algorithm is a job-level fixed-priority algorithm. As we will see in Chapter 8, this change in the relative priorities of jobs makes resource access control much more costly.

25, 25, 50).

Because scheduling decisions are made only at the times when jobs are released or completed, this version of the LST algorithm does not follow the LST rule of priority assignment at all times. If we wish to be specific, we should call this version of the LST algorithm the *nonstrict LST* algorithm. If the scheduler were to follow the LST rule strictly, it would have to monitor the slacks of all ready jobs and compare them with the slack of the executing job. It would reassign priorities to jobs whenever their slacks change relative to each other. As an example, according to the schedule in Figure 6–4, the scheduler would find that at time 2.7, the slack of $J_{2,1}$ becomes $(5 - 2.7 - 1.2) = 1.1$, the same as that of $J_{1,2}$. It would schedule the two ready jobs in a round-robin manner until $J_{1,2}$ completes. The run-time overhead of the strict LST algorithm includes the time required to monitor and compare the slacks of all ready jobs as time progresses. Moreover, by letting jobs with equal slacks execute in a round-robin manner, these jobs suffer extra context switches. For this reason, the strictly LST algorithm is an unattractive alternative, and we will not consider it further.

According to our classification, FIFO and Last-in-First-Out (LIFO) algorithms are also dynamic-priority algorithms. As an example, suppose that we have three tasks: $T_1 = (0, 3, 1, 3)$, $T_2 = (0.5, 4, 1, 1)$, and $T_3 = (0.75, 7.5, 2, 7.5)$. Suppose that the jobs in them

are scheduled on the FIFO basis. Clearly, $J_{1,1}$ has a higher priority than $J_{2,1}$, which in turn has a higher priority than $J_{3,1}$. In other words, T_1 has the highest priority, and T_3 has the lowest priority initially. Later, $J_{1,4}$, $J_{2,3}$, and $J_{3,2}$ are released at the times 9, 8.5, and 8.25, respectively, and T_3 has the highest priority while T_1 has the lowest priority.

6.2.3 Relative Merits

Algorithms that do not take into account the urgencies of jobs in priority assignment usually perform poorly. Dynamic-priority algorithms such as FIFO and LIFO are examples. (If the two tasks in Figure 6–4 were scheduled on the FIFO basis, most of the jobs in T_1 would miss their deadlines.) An example of fixed-priority algorithms of this nature is one which assigns priorities to tasks on the basis of their functional criticality: the more critical the task, the higher the priority. (Suppose that T_1 in this example is a video display task while T_2 is a task which monitors and controls a patient's blood pressure. The latter is clearly more functionally critical. If it were given the higher-priority, T_1 would miss most of its deadlines. This sacrifice of T_1 for T_2 is unnecessary since both tasks can be feasibly scheduled.) Hereafter, we confine our attention to algorithms that assign priorities to jobs based on one or more temporal parameters of jobs and have either optimal or reasonably good performance. The RM, DM, EDF, and the LST algorithm are such algorithms.

A criterion we will use to measure the performance of algorithms used to schedule periodic tasks is the schedulable utilization. The *schedulable utilization* of a scheduling algorithm is defined as follows: *A scheduling algorithm can feasibly schedule any set of periodic tasks on a processor if the total utilization of the tasks is equal to or less than the schedulable utilization of the algorithm.*

Clearly, the higher the schedulable utilization of an algorithm, the better the algorithm. Since no algorithm can feasibly schedule a set of tasks with a total utilization greater than 1, an algorithm whose schedulable utilization is equal to 1 is an optimal algorithm. In the subsequent sections, we will show that the EDF algorithm is optimal in this sense as expected, but the RM and DM algorithms are not.

While by the criterion of schedulable utilization, optimal dynamic-priority algorithms outperform fixed-priority algorithms, an advantage of fixed-priority algorithms is predictability. The timing behavior of a system scheduled according to a fixed-priority algorithm is more predictable than that of a system scheduled according to a dynamic-priority algorithm. When tasks have fixed priorities, overruns of jobs in a task can never affect higher-priority tasks. It is possible to predict which tasks will miss their deadlines during an overload.

In contrast, when the tasks are scheduled according to a dynamic algorithm, it is difficult to predict which tasks will miss their deadlines during overloads. This fact is illustrated by the examples in Figure 6–5. The tasks shown in Figure 6–5(a) have a total utilization of 1.1. According to their EDF schedule shown here, the job $J_{1,5}$ in $T_1 = (2, 1)$ is not scheduled until 10 and misses its deadline at 10. Deadlines of all jobs in T_2 are met in the schedule segment. The tasks in Figure 6–5(b) also have a total utilization of 1.1. According to the EDF schedule in Figure 6–5(b), $J_{1,5}$ in $T_1 = (2, 0.8)$, as well as every job in T_2, cannot complete on time. There is no easy test, short of an exhaustive one, that allows us to determine which tasks will miss their deadlines and which tasks will not.

The EDF algorithm has another serious disadvantage. We note that a late job which has already missed its deadline has a higher-priority than a job whose deadline is still in the future.

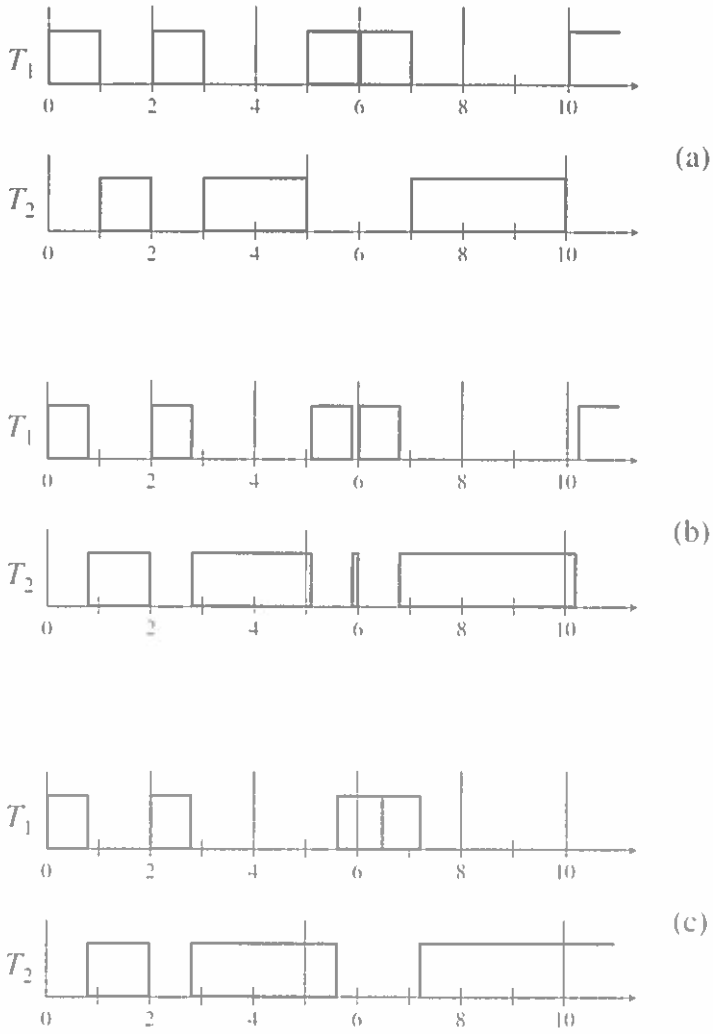

FIGURE 6–5 Unpredictability and instability of the EDF algorithm. (a) An EDF schedule of $T_1 = (2, 1)$ and $T_2 = (5, 3)$ with $U = 1.1$. (b) An EDF schedule of $T_1 = (2, 0.8)$ and $T_2 = (5, 3.5)$ with $U = 1.1$. (c) An EDF schedule of $T_1 = (2, 0.8)$ and $T_2 = (5, 4.0)$ with $U = 1.2$.

Consequently, if the execution of a late job is allowed to continue, it may cause some other jobs to be late. As an example, we examine the schedule shown in Figure 6–5(c). When $J_{2,1}$ becomes late at 5, it continues to execute because its priority is higher than the priority of $J_{1,3}$. As a consequence, $J_{1,3}$ is late also. In fact, after $J_{1,4}$, every job in both tasks will be late. This unstable behavior of the EDF algorithm, that one job being late causes many other jobs to be late, makes the algorithm unsuitable for systems where overload conditions are unavoidable.

A good overrun management strategy is crucial to prevent this kind of instability. An overrun management strategy is to schedule each late job at a lower priority than the jobs that are not late. An alternative is to complete the late job, but schedule some functional noncritical

jobs at the lowest priority until the system recovers from the overload condition. There are many other alternatives. Invariably, the scheduler either lowers the priorities of some or all the late jobs (or portions of jobs), or discards some jobs if they cannot complete by their deadlines and logs this action. Clearly, what alternatives are suitable depends on the application.

6.3 MAXIMUM SCHEDULABLE UTILIZATION

Again, we say that a system is *schedulable* by an algorithm if the algorithm always produces a feasible schedule of the system. A system is schedulable (and *feasible*) if it is schedulable by some algorithm, that is, feasible schedules of the system exist. We now ask how large the total utilization of a system can be in order for the system to be surely schedulable.

6.3.1 Schedulable Utilizations of the EDF Algorithm

We first focus on the case where the relative deadline of every task is equal to its period. (This choice of the relative deadline arises naturally from throughput considerations. The job in each period completes before the next period starts so there is no backlog of jobs.) The following theorem tells us that any such system can be feasibly scheduled if its total utilization is equal to or less than one, no matter how many tasks there are and what values the periods and execution times of the tasks are. In the proof of this and later theorems, we will use the following terms. At any time t, the *current period* of a task is the period that begins before t and ends at or after t. We call the job that is released in the beginning of this period the *current job*.

THEOREM 6.1. A system T of independent, preemptable tasks with relative deadlines equal to their respective periods can be feasibly scheduled on one processor if and only if its total utilization is equal to or less than 1.

Proof. That the system is not feasible if its total utilization is larger than 1 is obvious so we focus on the *if* part of the proof. As stated in Theorem 4.1, the EDF algorithm is optimal in the sense that it can surely produce a feasible schedule of any feasible system. Hence, it suffices for us to prove that the EDF algorithm can surely produce a feasible schedule of any system with a total utilization equal to 1. We prove this statement by showing that if according to an EDF schedule, the system fails to meet some deadlines, then its total utilization is larger than 1. To do so, let us suppose that the system begins to execute at time 0 and at time t, the job $J_{i,c}$ of task T_i misses its deadline.

For the moment, we assume that prior to t the processor never idles. We will remove this assumption at the end of the proof. There are two cases to consider: (1) The current period of every task begins at or after $r_{i,c}$, the release time of the job that misses its deadline, and (2) the current periods of some tasks begin before $r_{i,c}$. The two cases are illustrated in Figure 6–6. In this figure, we see that the current jobs of all tasks T_k, for all $k \neq i$, have equal or lower priorities than $J_{i,c}$ because their deadlines are at or after t.

Case (1): This case is illustrated by the time lines in Figure 6–6(a); each tick on the time line of a task shows the release time of some job in the task. The fact that

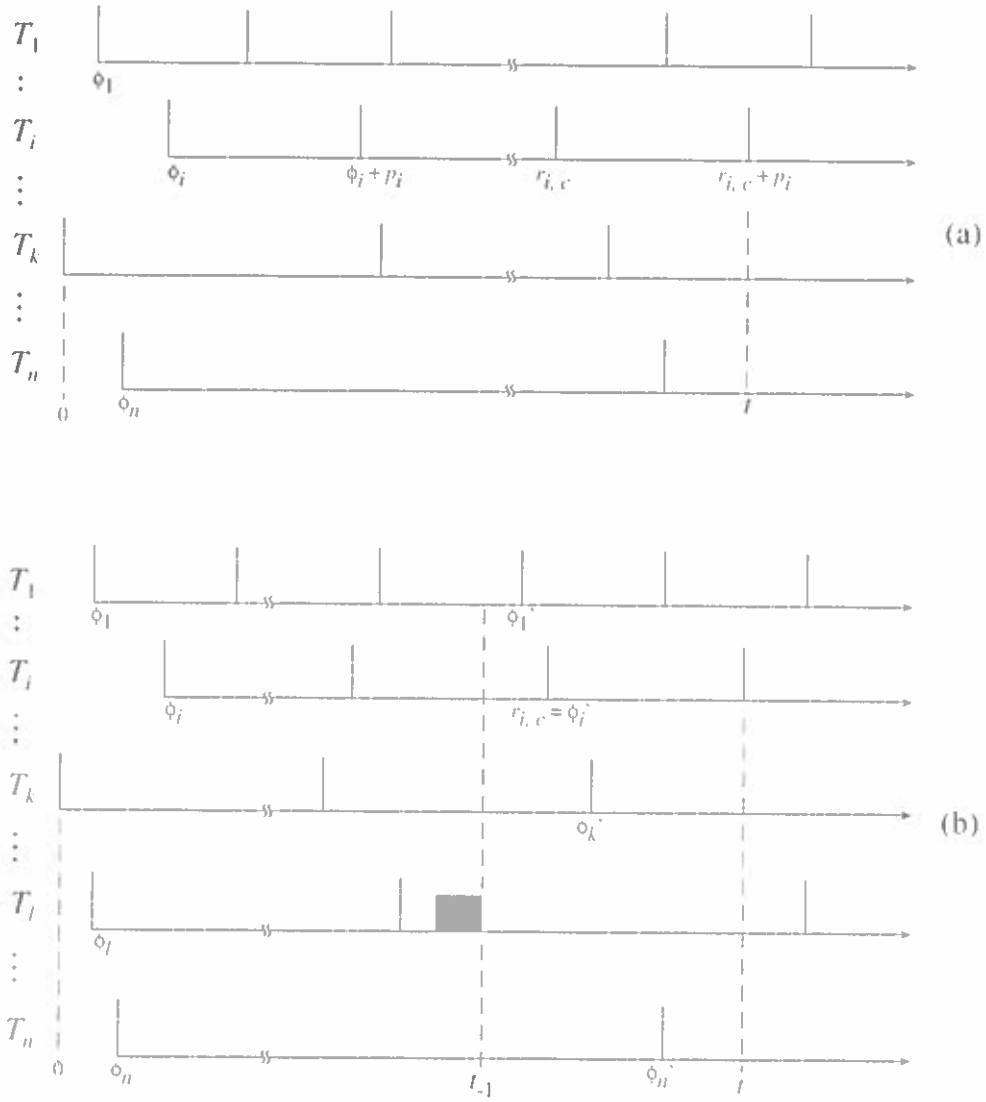

FIGURE 6–6 Infeasible EDF schedules.

$J_{i,c}$ misses its deadline at t tells us that any current job whose deadline is after t is not given any processor time to execute before t and that the total processor time required to complete $J_{i,c}$ and all the jobs with deadlines at or before t exceeds the total available time t. In other words,

$$t < \frac{(t - \phi_i)e_i}{p_i} + \sum_{k \neq i} \left\lfloor \frac{t - \phi_k}{p_k} \right\rfloor e_k \tag{6.1}$$

(You recall that ϕ_k, p_k and e_k are the phase, period, and execution time of task T_k, respectively.) $\lfloor x \rfloor$ ($x \geq 0$) denotes the largest integer less than or equal to x. The first term on the right-hand side of the inequality is the time required to complete all the jobs in T_i with deadlines before t and the job $J_{i,c}$. Each term in the sum gives the total amount of time before t required to complete jobs that are in a task T_k other than T_i and have deadlines at or before t. Since $\phi_k \geq 0$ and $e_k/p_k = u_k$ for all k, and $\lfloor x \rfloor \leq x$ for any $x \geq 0$,

$$\frac{(t-\phi_i)e_i}{p_i} + \sum_{k \neq i} \left\lfloor \frac{t-\phi_k}{p_k} \right\rfloor e_k \leq t\frac{e_i}{p_i} + t\sum_{k \neq i} \frac{e_k}{p_k} = t\sum_{k=1}^{n} u_k = tU$$

Combining this inequality with the one in Eq. (6.1), we have $U > 1$.

Case (2): The time lines in of Figure 6–6(b) illustrate case (2). Let $\mathbf{T}'$ denote the subset of $\mathbf{T}$ containing all the tasks whose current jobs were released before $r_{i,c}$ and have deadlines after t. It is possible that some processor time before $r_{i,c}$ was given to the current jobs of some tasks in $\mathbf{T}'$. In the figure, T_j is such a task.

Let t_{-1} be the end of the latest time interval I (shown as a black box in Figure 6–6(b)) that is used to execute some current job in $\mathbf{T}'$. We now look at the segment of the schedule starting from t_{-1}. In this segment, none of the current jobs with deadlines after t are given any processor time. Let ϕ_k' denote the release time of the first job of task T_k in $\mathbf{T} - \mathbf{T}'$ in this segment. Because $J_{i,c}$ misses its deadline at t, we must have

$$t - t_{-1} < \frac{(t-t_{-1}-\phi_i')e_i}{p_i} + \sum_{T_k \in \mathbf{T}-\mathbf{T}'} \left\lfloor \frac{t-t_{-1}-\phi_k'}{p_k} \right\rfloor e_k$$

(Tasks in $\mathbf{T}'$ are not included the sum; by definition of t_{-1}, these tasks are not given any processor time after t_{-1}.) This inequality is the same as the one in Eq. (6.1) except that t is replaced by $t - t_{-1}$ and ϕ_k is replaced by ϕ_k'. We can used the same argument used above to prove that $\sum_{T_k \in \mathbf{T}-\mathbf{T}'} u_k > 1$, which in turn implies that $U > 1$.

Now we consider the case where the processor idles for some time before t. Let t_{-2} be the latest instant at which the processor idles. In other words, from t_{-2} to the time t when $J_{i,c}$ misses its deadline, the processor never idles. For the same reason that Eq. (6.1) is true, the total time required to complete all the jobs that are released at and after t_{-2} and must be completed by t exceeds the total available time $t - t_{-2}$. In other words, $U > 1$. □

The following facts follow straightforwardly from this theorem.

1. *A system of independent, preemptable periodic tasks with relative deadlines longer than their periods can be feasibly scheduled on a processor as long as the total utilization is equal to or less than 1.*
2. *The schedulable utilization $U_{EDF}(n)$ of the EDF algorithm for n independent, preemptable periodic tasks with relative deadlines equal to or larger than their periods is equal to 1.*

The EDF algorithm is not the only algorithm with this schedulable utilization. In particular, *the schedulable utilization of the LST algorithm is also 1*. This follows straightforwardly from Theorem 4.3 which states that the LST algorithm is also optimal for scheduling independent, preemptable jobs on one processor.

When the relative deadlines of some tasks are less than their respective periods, the system may no longer be feasible, even when its total utilization is less than 1. As an example, the task with period 5 and execution time 2.3 in Figure 6–4 would not be schedulable if its relative deadline were 3 instead of 5.

We call the ratio of the execution time e_k of a task T_k to the minimum of its relative deadline D_k and period p_k the *density* of the task. In other words, the density of T_k is $e_k/\min(D_k, p_k)$. The sum of the densities of all tasks in a system is the *density* of the system and is denoted by Δ. When $D_i < p_i$ for some task T_i, $\Delta > U$. If the density of a system is larger than 1, the system may not be feasible. For example, this sum is larger than 1 for (2, 0.9) and (5, 2.3, 3), and the tasks are not schedulable by any algorithm. On the other hand, any system is feasible if its density is equal to or less than 1. We state this fact in the following theorem which generalizes Theorem 6.1; its proof is similar to the proof of Theorem 6.1 and is left to you as an exercise.

THEOREM 6.2. A system **T** of independent, preemptable tasks can be feasibly scheduled on one processor if its density is equal to or less than 1.

The condition given by this theorem is not necessary for a system to be feasible. A system may nevertheless be feasible when its density is greater than 1. The system consisting of (2, 0.6, 1) and (5, 2.3) is an example. Its density is larger than 1, but it is schedulable according to the EDF algorithm.

6.3.2 Schedulability Test for the EDF Algorithm

Hereafter, we call a test for the purpose of validating that the given application system can indeed meet all its hard deadlines when scheduled according to the chosen scheduling algorithm a *schedulability test*. If a schedulability test is efficient, it can be used as an on-line acceptance test.

Checking whether a set of periodic tasks meet all their deadlines is a special case of the validation problem that can be stated as follows: We are given

1. the period p_i, execution time e_i, and relative deadline D_i of every task T_i in a system $\mathbf{T} = \{T_1, T_2, \ldots T_n\}$ of independent periodic tasks, and
2. a priority-driven algorithm used to schedule the tasks in **T** preemptively on one processor.

We are asked to determine whether all the deadlines of every task T_i, for every $1 \leq i \leq n$, are always met. We note that in the above statement, the phases of tasks are not given. If we were also given the phases and if the values of all the given parameters would never vary, this problem could be solved by simulation. We simply construct a segment of the schedule of these tasks according to the given scheduling algorithm. A segment of length $2H + \max_i p_i +$

$\max_i D_i$ suffices [BaHR]. T_i meets all its deadlines if we observe no missed deadline in this schedule segment. However, this method does not work whenever the parameters of the tasks do vary; some of the reasons were discussed in Section 4.8.

When the scheduling algorithm is the EDF algorithm, Theorems 6.1 and 6.2 give us the theoretical basis of a very simple schedulability test. To determine whether the given system of n independent periodic tasks surely meets all the deadlines when scheduled according to the preemptive EDF algorithm on one processor, we check whether the inequality

$$\sum_{k=1}^{n} \frac{e_k}{\min(D_k, p_k)} \leq 1 \tag{6.2}$$

is satisfied. We call this inequality the *schedulability condition* of the EDF algorithm. If it is satisfied, the system is schedulable according to the EDF algorithm. When Eq. (6.2) is not satisfied, the conclusion we may draw from this fact depends on the relative deadlines of the tasks. If $D_k \geq p_k$ for all k from 1 to n, then Eq. (6.2) reduces to $U \leq 1$, which is both a necessary and sufficient condition for a system to be feasible. On the other hand, if $D_k < p_k$ for some k, Eq. (6.2) is only a sufficient condition; therefore we can only say that the system may not be schedulable when the condition is not satisfied.

This schedulability test is not only simple but also robust. Theorem 4.4 states that the execution of independent jobs on one processor according to a preemptive schedule is predictable. This theorem allows us to conclude that the system remains schedulable if some jobs execute for less time than their respective (maximum) execution times. The proof of Theorems 6.1 (and similarly the proof of Theorem 6.2) never makes use of the actual values of the periods; it only makes use of the total demand for processor time by the jobs of each task T_k in any interval of length t. If the actual interrelease times are sometimes longer than p_k, the total demand will in fact be smaller. Hence if according to Eq. (6.2) the system is schedulable on the EDF basis, it remains schedulable when the interrelease times of jobs in some tasks are sometimes longer than the respective periods of tasks. We will return to state this fact more formally in Section 7.4 when we discuss sporadic tasks.

We can also use Eq. (6.2) as a rule to guide the choices of the periods and execution times of the tasks while we design the system. As a simple example, we consider a digital robot controller. Its control-law computation takes no more than 8 milliseconds on the chosen processor to complete. The desired sampling rate is 100 Hz, that is, the control-law task executes once every 10 milliseconds. Clearly the task is feasible if the control-law task is the only task on the processor and its relative deadline is equal to the period. Suppose that we also need to use the processor for a Built-In Self-Test (BIST) task. The maximum execution time of this task is 50 milliseconds. We want to do this test as frequently as possible without causing the control-law task to miss any deadline, but never less frequently than once a second. Equation (6.2) tells us that if we schedule the two tasks according to the EDF algorithm, we can execute the BIST task as frequently as once every 250 milliseconds. Now suppose that we need to add a telemetry task. Specifically, to send or receive each message, the telemetry task must execute 15 milliseconds. Although the interarrival times of messages are very large, we want the relative deadline of message processing jobs to be as small as feasible. Equation (6.2) tells us that if we are willing to reduce the frequency of the BIST task to once a second, we can make the relative deadline of the telemetry task as short as 100 milliseconds. If

this guaranteed maximum response time of the telemetry task is not acceptable and we must redesign the three tasks. Eq. (6.2) can guide us in the process of trading off among the task parameters in order to keep the entire system feasible.

6.4 OPTIMALITY OF THE RM AND DM ALGORITHMS

Hereafter in our discussion on fixed-priority scheduling, we index the tasks in decreasing order of their priorities except where stated otherwise. In other words, the task T_i has a higher priority than the task T_k if $i < k$. By indexing the tasks in this manner, our discussion implicitly takes into consideration the scheduling algorithm. Sometimes, we refer to the priority of a task T_i as priority π_i. π_i's are positive integers $1, 2, \ldots, n$, 1 being the highest priority and n being the lowest priority. We denote the subset of tasks with equal or higher priority than T_i by $\mathbf{T}_i$ and its total utilization by $U_i = \sum_{k=1}^{i} u_k$. You may have noticed that we implicitly assume here that the tasks have distinct priorities. At the end of the chapter we will return to remove this assumption, as well as to discuss the effects of a limited number of priority levels when it is not possible to give tasks distinct priorities.

Because they assign fixed priorities to tasks, fixed-priority algorithms cannot be optimal: Such an algorithm may fail to schedule some systems for which there are feasible schedules. To demonstrate this fact, we consider a system which consists of two tasks: $T_1 = (2, 1)$ and $T_2 = (5, 2.5)$. Since their total utilization is equal to 1, we know from Theorem 6.1 that the tasks are feasible. $J_{1,1}$ and $J_{1,2}$ can complete in time only if they have a higher priority than $J_{2,1}$. In other words, in the time interval $(0, 4]$, T_1 must have a higher-priority than T_2. However, at time 4 when $J_{1,3}$ is released, $J_{2,1}$ can complete in time only if T_2 (i.e., $J_{2,1}$) has a higher priority than T_1 (i.e., $J_{1,3}$). This change in the relative priorities of the tasks is not allowed by any fixed priority algorithm.

While the RM algorithm is not optimal for tasks with arbitrary periods, it is optimal in the special case when the periodic tasks in the system are simply periodic and the deadlines of the tasks are no less than their respective periods. A system of periodic tasks is *simply periodic* if for every pair of tasks T_i and T_k in the system and $p_i < p_k$, p_k is an integer multiple of p_i. An example of a simply periodic task system is the flight control system in Figure 1-3. In that system, the shortest period is 1/180 seconds. The other two periods are two times and six times 1/180 seconds. The following theorem states that the RM algorithm is optimal for scheduling such a system.

> **THEOREM 6.3.** A system of simply periodic, independent, preemptable tasks whose relative deadlines are equal to or larger than their periods is schedulable on one processor according to the RM algorithm if and only if its total utilization is equal to or less than 1.

To see why this theorem is true, we suppose for now that the tasks are in phase (i.e., the tasks have identical phases) and the processor never idles before the task T_i misses a deadline for the first time at t. (In the next section we will justify why it suffices for us to consider the case where the tasks are in phase and the processor never idles before t.) t is an integer multiple of p_i. Because the tasks are simply periodic, t is also an integer multiple of the period p_k of every higher-priority task T_k, for $k = 1, 2, \ldots, i - 1$. Hence the total time required to

complete all the jobs with deadlines before and at t is equal to $\sum_{k=1}^{i}(e_k t/p_k)$, which is equal to t times the total utilization $U_i = \sum_{k=1}^{i} u_k$ of the i highest priority tasks. That T_i misses a deadline at t means that this demand for time exceeds t. In other words, $U_i > 1$.

Despite the fact that fixed-priority scheduling is not optimal in general, we may nevertheless choose to use this approach because it leads to a more predictable and stable system. For this reason, we want to know among all the fixed-priority algorithms, which one(s) is the best. The answer is that the DM algorithm is the optimal fixed-priority algorithm. Theorem 6.4 states this fact more precisely. We will return shortly to clarify the sense in which this theorem supports the claim that the DM algorithm is optimal even though the theorem is stated for tasks with identical phase, that is, in phase.

> **THEOREM 6.4.** A system T of independent, preemptable periodic tasks that are in phase and have relative deadlines equal to or less than their respective periods can be feasibly scheduled on one processor according to the DM algorithm whenever it can be feasibly scheduled according to any fixed-priority algorithm.

This theorem is true because we can always transform a feasible fixed-priority schedule that is not a DM schedule into one that is. Specifically, suppose that a system has a feasible fixed-priority schedule that is not a DM schedule. We scan the tasks, starting from task T_1 with the shortest relative deadline in order of increasing relative deadlines. When we find two tasks T_i and $T_i + 1$ which are such that D_i is less than D_{i+1} but T_i has a lower priority than T_{i+1} according to this schedule, we switch the priorities of these two tasks and modify the schedule of the two tasks accordingly. After the switch, the priorities of the two tasks are assigned on the DM basis relative to one another. By repeating this process, we can transform the given schedule into a DM schedule. The step of showing that when the tasks are in phase, it is always possible to switch the priorities of tasks T_i and T_{i+1} and hence the time intervals in which jobs in these tasks are scheduled without leading to any missed deadline is left to you as an exercise.

In some systems, the relative deadline of every task T_i is equal to δp_i for some constant $0 < \delta$. For such systems, the DM and RM algorithms are the same. As a corollary to the above theorem, *the RM algorithm is optimal among all fixed-priority algorithms whenever the relative deadlines of the tasks are proportional to their periods.*

6.5 A SCHEDULABILITY TEST FOR FIXED-PRIORITY TASKS WITH SHORT RESPONSE TIMES

We now describe a pseudopolynomial time schedulability test developed by Lehoczky, *et al.* [LeSD] and Audsley, *et al.* [ABTR] for tasks scheduled according to a fixed-priority algorithm. In this section we confine our attention to the case where the response times of the jobs are smaller than or equal to their respective periods. In other words, every job completes before the next job in the same task is released. We will consider the general case where the response times may be larger than the periods in the next section. Since no system with total utilization greater than 1 is schedulable, we assume hereafter that the total utilization U is equal to or less than 1.

6.5.1 Critical Instants

The schedulability test uses as inputs the given sets $\{p_i\}$ and $\{e_i\}$ of periods and execution times of the tasks in $\mathbf{T}$ and checks one task T_i at a time to determine whether the response times of all its jobs are equal to or less than its relative deadline D_i. Because we cannot count on any relationship among the release times to hold, we must first identify the worst-case combination of release times of any job $J_{i,c}$ in T_i and all the jobs that have higher priorities than $J_{i,c}$. This combination is the worst because the response time of a job $J_{i,c}$ released under this condition is the largest possible for all combinations of release times. For this purpose, the notion of critical instant was first introduced in [LiLa] and subsequently refined by others (e.g., [Bake] pointed out the need to consider case (2) below.) A *critical instant* of a task T_i is a time instant which is such that

1. the job in T_i released at the instant has the maximum response time of all jobs in T_i, if the response time of every job in T_i is equal to or less than the relative deadline D_i of T_i, and
2. the response time of the job released at the instant is greater than D_i if the response time of some jobs in T_i exceeds D_i.

We call the response time of a job in T_i released at a critical instant the *maximum (possible) response time* of the task and denote it by W_i. The following theorem gives us the condition under which a critical instant of each task T_i occurs.

THEOREM 6.5. In a fixed-priority system where every job completes before the next job in the same task is released, a critical instant of any task T_i occurs when one of its job $J_{i,c}$ is released at the same time with a job in every higher-priority task, that is, $r_{i,c} = r_{k,l_k}$ for some l_k for every $k = 1, 2, \ldots, i-1$.

**Proof.* We first show that the response time of the first job $J_{i,1}$ is the largest when the tasks in the subset $\mathbf{T}_i$ of i highest priority tasks are in phase, that is, when the condition stated in the theorem is met. To do so, we take as the time origin the minimum of all the phases of tasks in $\mathbf{T}_i$. It suffices for us to consider the case where the processor remains busy executing jobs with higher priorities than $J_{i,1}$ before $J_{i,1}$ is released at ϕ_i.[1]

Let $W_{i,1}$ denote the response time of $J_{i,1}$. From the release time ϕ_k of the first job in T_k to the instant $\phi_i + W_{i,1}$ when the first job $J_{i,1}$ in T_i completes, at most $\lceil (W_{i,1} + \phi_i - \phi_k)/p_k \rceil$ jobs in T_k become ready for execution. Each of these jobs demands e_k units of processor time. Hence the total amount of processor time demanded by $J_{i,1}$ and all the jobs that must be completed before $J_{i,1}$ is given by

$$e_i + \sum_{k=1}^{i-1} \left\lceil \frac{W_{i,1} + \phi_i - \phi_k}{p_k} \right\rceil e_k$$

[1] If during some intervals before ϕ_i the processor idles or executes lower priority jobs, we can ignore the segment of the schedule before the end of the latest of such intervals, take this time instant as the time origin, and call the first job in every higher-priority task released after this instant the first job of the task.

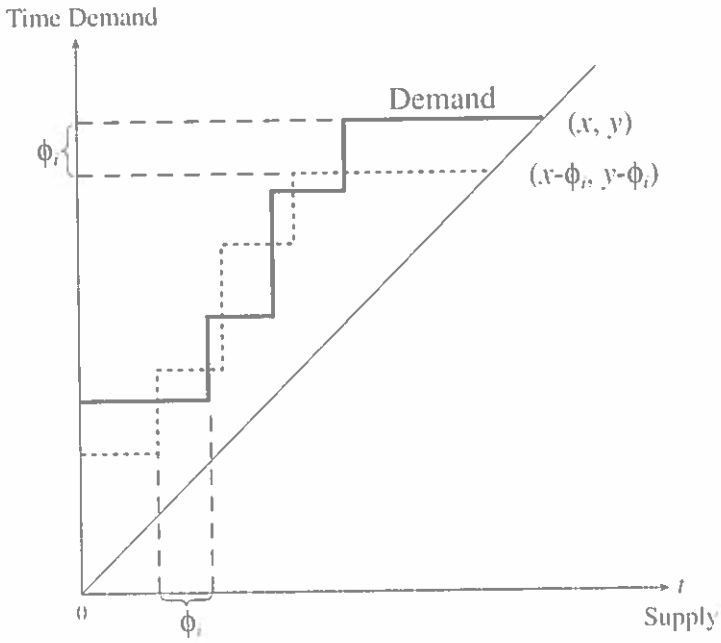

FIGURE 6–7 Dependency of response time on phases.

At time $W_{i,1} + \phi_i$ when $J_{i,1}$ completes, the supply of processor time becomes sufficient to meet this total demand for the processor time for first time since time 0. In other words, $W_{i,1}$ is equal to the smallest of all solutions of

$$W_{i,1} = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{W_{i,1} + \phi_i - \phi_k}{p_k} \right\rceil e_k - \phi_i \tag{6.3}$$

if this equation has solutions in the range $(0, p_i]$. If the equation does not have a solution in this range, then $J_{i,1}$ cannot complete in time and misses its deadline.

To see how $W_{i,1}$ depends on the phase ϕ_k of each higher-priority task, we note that the expression in the right-hand side of this equation is a staircase function of $W_{i,1}$. It lies above the 45° straight line $y(W_{i,1}) = W_{i,1}$ until it intersects the straight line; the value of $W_{i,1}$ at this intersection is the solution of Eq. (6.3). From Eq. (6.3), it is clear that the staircase function has the largest value, and hence its intersection with the 45° straight line has the largest value, when ϕ_k is 0 for every $k = 1, 2, \ldots, i-1$. In other words, the job $J_{i,1}$ has the largest response time when all the higher-priority tasks are in phase.

We now let ϕ_k be 0 for all $k < i$. The time instant when the processor first completes all the ready jobs in higher-priority tasks released since time 0 is independent when $J_{i,1}$ is released, and this is the first instant when $J_{i,1}$ can begin execution. Therefore, the sooner $J_{i,1}$ is released, the longer it must wait to start execution and the larger

its response time. This observation leads us to conclude that the response time $W_{i,1}$ of $J_{i,1}$ has the largest possible value when ϕ_i is also equal to zero.[2]

To show that an arbitrary job $J_{i,c}$ in T_i has the maximum possible response time whenever it is released at the same time with a job in every higher-priority task, we let the release time of this job be ϕ'_i and the release time of the job in T_k that is current at time ϕ'_i be ϕ'_k. Because every job completes in the period in which it is released, there is no need for us to consider the jobs in each task T_k that are released before ϕ'_k. Hence we can take the minimum among the release times ϕ'_k, for $k = 1, 2, \ldots, i$, of the current jobs as the time origin and consider the segment of the schedule starting from this new origin independently from the earlier segment. The above argument can be repeated and allows us to reach the conclusion that the response time of $J_{i,c}$ has the maximum value when the release times of all the current jobs in $\mathbf{T}_i$ are the same.

It follows from the above arguments that the maximum possible response time W_i of all jobs in T_i is, therefore, equal to the smallest value of t that satisfies the equation

$$t = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k \tag{6.4}$$

It is easy to see that if Eq. (6.3) does not have a solution equal to or less than D_i, neither does this equation. □

Figure 6–8 gives an illustrative example. Figure 6–8(a) shows an RM schedule of the three jobs, (2, 0.6), (2.5, 0.2), and (3, 1.2) when they are in phase. Time 0 is a critical instant of both lower-priority tasks. The response times of the jobs in (2.5, 0.2) are 0.8, 0.3, 0.2, 0.2, 0.8, and so on. These times never exceed the response time of the first job. Similarly, the response times of the jobs in (3, 1.2) are 2, 1.8, 2, 2, and so on, which never exceed 2, the response time of the first job in (3, 1.2). Figure 6–8(b) shows an RM schedule when the phase of the task with period 2.5 is one, while the phases of the other tasks are 0. We see that 6 is a critical instant of the two lower-priority tasks. The jobs of these tasks released at this instant have the maximum possible response times of 0.8 and 2, respectively.

We are now ready to clarify a point that was left fuzzy at the end of the last section. Although Theorem 6.4 is stated in terms of tasks that are in phase, we claimed that the DM algorithm is an optimal fixed-priority algorithm. Theorem 6.5 gives us justification. Whenever release-time jitters are not negligible, the information on release times cannot be used to determine whether any algorithm can feasibly schedule the given system of tasks. Under this

[2]Rather than relying on this intuitive argument, we can arrive at this conclusion by examining how the solution in Eq. (6.3) depends on ϕ_i when ϕ_k equal 0 for every $k = 1, 2, \ldots, i-1$. The solid graph in Figure 6–7 shows the general behavior of the staircase function when ϕ_i is also equal to 0. Each rise occurs when $W_{i,1}$ is an integer multiple of some p_k. When ϕ_i is nonzero, the staircase function behaves as indicated by the dotted line. Each rise in the dotted line occurs when $W_{i,1} + \phi_i$ is an integer multiple of some period p_k. Hence, corresponding to each rise in the solid staircase function at some value z of $W_{i,1}$, there is a rise in the dotted function at $z - \phi_i$. The step sizes at the corresponding rises of both staircase functions are the same. However, every plateau in the dotted function is ϕ_i units lower than the corresponding plateau in the solid function. In short, we can obtain the dotted staircase function by shifting the solid staircase function to the right by ϕ_i units and down for ϕ_i units. From this, we can see that if the solid staircase function intersects the 45° straight line for the first time at some point (x, y), as shown in Figure 6–7, the dotted staircase function intersects the straight line at $(x - \phi_i, x - \phi_i)$. Therefore, the response time of $J_{i,c}$ for an arbitrary ϕ_i is equal to its response time for zero phase minus ϕ_i.

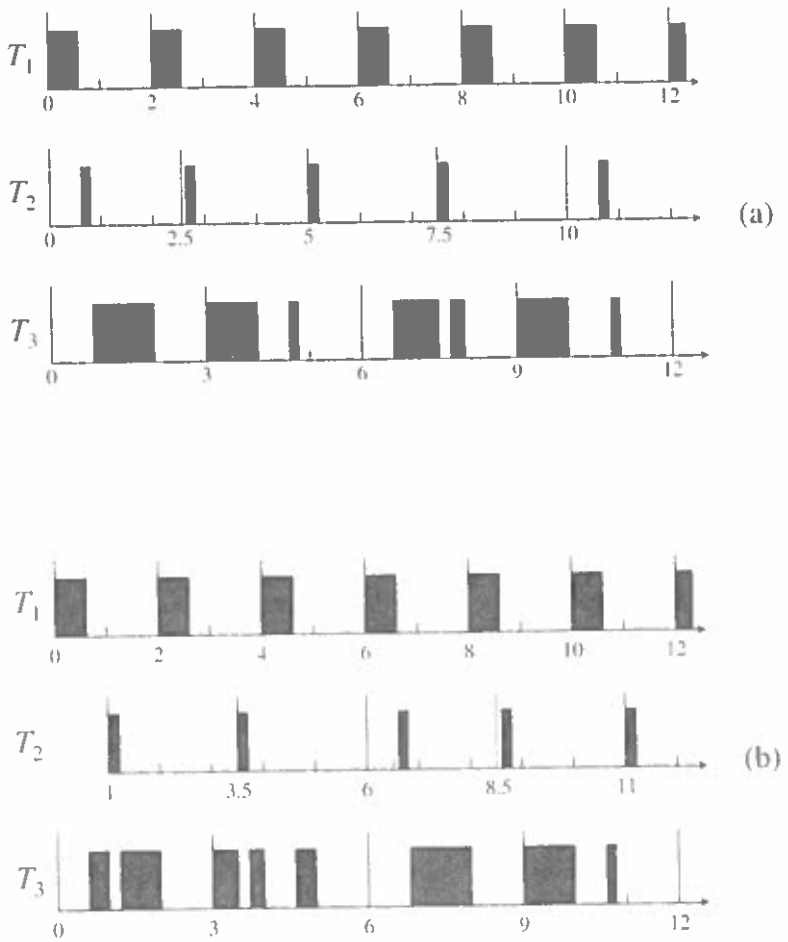

FIGURE 6–8 Example illustrating critical instants ($T_1 = (2, 0.6)$, $T_2 = (2.5, 0.2)$, $T_3 = (3, 1.2)$).

circumstance, we have no choice but to judge a fixed-priority algorithm according to its performance for tasks that are in phase because all fixed-priority algorithms have their worst-case performance for this combination of phases.

6.5.2 Time-Demand Analysis

The schedulability test [LeSD, ABTR] described below makes use of the fact stated in Theorem 6.5. To determine whether a task can meet all its deadlines, we first compute the total demand for processor time by a job released at a critical instant of the task and by all the higher-priority tasks as a function of time from the critical instant. We then check whether this demand can be met before the deadline of the job. For this reason, we name this test a *time-demand analysis*.

To carry out the time-demand analysis on T, we consider one task at a time, starting from the task T_1 with the highest priority in order of decreasing priority. To determine whether

a task T_i is schedulable after finding that all the tasks with higher priorities are schedulable, we focus on a job in T_i, supposing that the release time t_0 of the job is a critical instant of T_i. At time $t_0 + t$ for $t \geq 0$, the total (processor) time demand $w_i(t)$ of this job and all the higher-priority jobs released in $[t_0, t]$ is given by

$$w_i(t) = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k, \qquad \text{for } 0 < t \leq p_i \tag{6.5}$$

This job of T_i can meet its deadline $t_0 + D_i$ if at some time $t_0 + t$ at or before its deadline, the supply of processor time, which is equal to t, becomes equal to or greater than the demand $w_i(t)$ for processor time. In other words, $w_i(t) \leq t$ for some $t \leq D_i$, where D_i is equal to or less than p_i. Because this job has the maximum possible response time of all jobs in T_i, we can conclude that all jobs in T_i can complete by their deadlines if this job can meet its deadline.

If $w_i(t) > t$ for all $0 < t \leq D_i$, this job cannot complete by its deadline; T_i, and hence the given system of tasks, cannot be feasibly scheduled by the given fixed-priority algorithm. More specifically, as long as the variations in the interrelease times of tasks are not negligibly small, we have to make this conclusion because a critical instant of T_i can occur. However, if the given tasks have known phases and periods and the jitters in release times are negligibly small, it is possible to determine whether any job of T_i is ever released at the same time as a job of every higher priority task. If this condition can never occur, a critical instant of T_i can never occur; no job in T_i can ever have the maximum possible response time. T_i may nevertheless be schedulable even though the time-demand analysis test indicates that it is not.

Because of Theorem 6.5, we can say that $w_i(t)$ given by the Eq. (6.5) is the maximum time demanded by any job in T_i, plus all the jobs that must be completed before this job, at any time $t < p_i$ since its release. We call $w_i(t)$ the *time-demand function* of the task T_i.

As an example, we consider a system containing four tasks: $T_1 = (\phi_1, 3, 1)$, $T_2 = (\phi_2, 5, 1.5)$, $T_3 = (\phi_3, 7, 1.25)$, and $T_4 = (\phi_4, 9, 0.5)$, where the phases of the tasks are arbitrary. Their total utilization is 0.87. The given scheduling algorithm is the RM algorithm. In Figure 6–9, the solid lines show the time-demand functions of these four tasks. The dotted lines show the total contributions of higher-priority tasks to the time-demand function of each lower-priority task. For example, the job in T_2 being considered is released at t equal to 0. In the time interval $(0, 5)$, it contributes 1.5 units of time to its time-demand function $w_2(t)$. T_1 contributes a total of 1 unit from time 0 to 3 and 2 units from time 3 to 6, and so on. Because at time $t = 2.5$ the total demand $w_2(t)$ of 2.5 units is met, the maximum possible response time of jobs in T_2 is 2.5. Since the relative deadline of T_2 is 5, the task is schedulable. From Figure 6–9, we can see that every task in this system is schedulable. In particular, the dot at the intersection of $w_i(t)$ and the straight line $y(t) = t$ marks the time instant from a critical instant of T_i at which the job in T_i released at the critical instant completes. From this graph, we can tell that the maximum possible response times of the tasks are 1, 2.5, 4.75, and 9, respectively.

Suppose that in addition to these four tasks, there is a fifth task with period 10 and execution time 1. It is easy to see that the time-demand function of this task lies entirely above the supply function t from 0 to 10. For this reason, we can conclude that this task cannot be feasibly scheduled by the given scheduling algorithm.

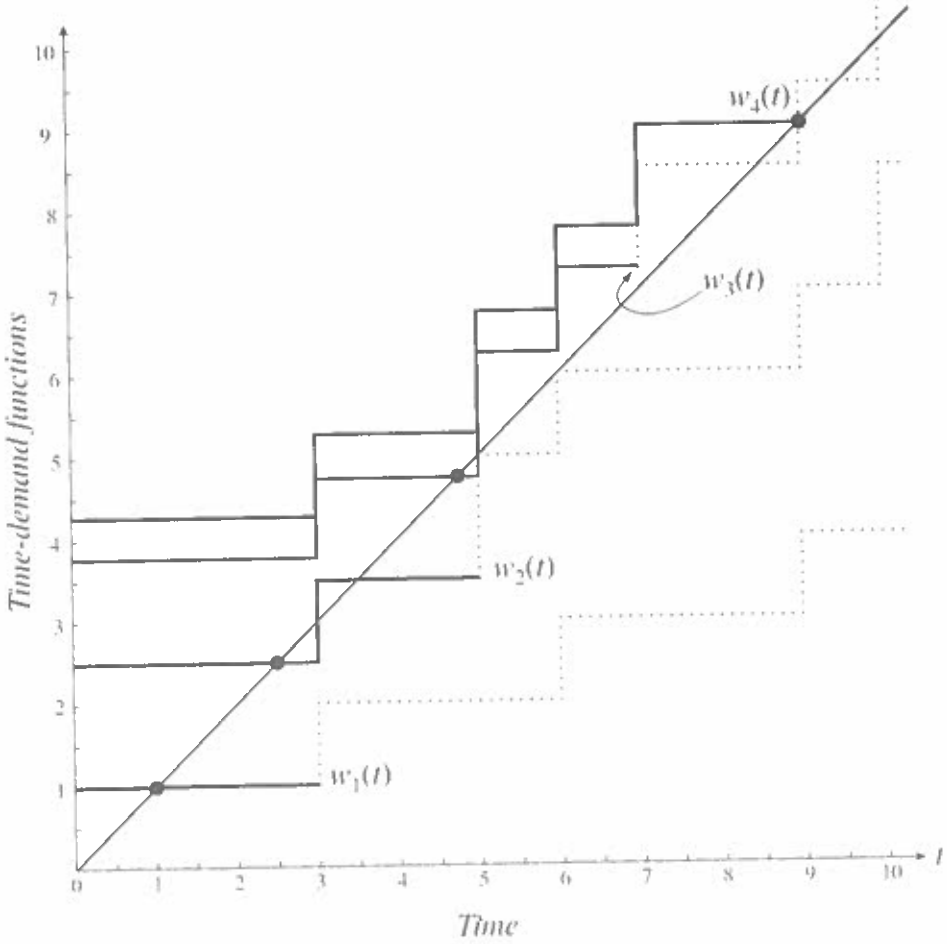

FIGURE 6-9 Time-demand analysis (3,1), (5, 1.5), (7, 1.25), and (9, 0.5).

This example illustrates the fact that the time-demand function of any task T_i is a staircase function. The rises in the function occur at time instants which are integer multiples of periods of higher-priority tasks. Immediately before each rise in each function $w_i(t)$, the shortage $w_i(t) - t$ between the processor-time demand and the supply is the smallest for all t in the interval from the previous rise. (In this example, in the range of t from 0 to 3, $w_2(t) - t$ is the smallest at t equal to 3; in the range from 3 to 5, this difference is the smallest at t equal to 5.) Hence, if we are not interested in the value of its maximum possible response time, only whether a task is schedulable, it suffices for us to check whether the time-demand function of the task is equal to or less than the supply at these instants.

In summary, to determine whether T_i can be feasibly scheduled by the given scheduling algorithm using the *time-demand analysis method* proposed by Lehoczky, *et al.* [LeSD], we

1. compute the time-demand function $w_i(t)$ according to Eq. (6.5), and

2. check whether the inequality

$$w_i(t) \le t \tag{6.6a}$$

is satisfied for values of t that are equal to

$$t = jp_k: \qquad k = 1, 2, \ldots i;\ j = 1, 2, \ldots, \lfloor \min(p_i, D_i)/p_k \rfloor \tag{6.6b}$$

If this inequality is satisfied at any of these instants, T_i is schedulable.

Let $q_{n,1}$ denote the *period ratio* of the system, that is, the ratio of the largest period p_n to the smallest period p_1. The time complexity of the time-demand analysis for each task is $O(nq_{n,1})$.

For task T_3 in this example, we need to check whether its time-demand function $w_3(t)$ is equal to or less than t at t equal to 3, 5, 6, and 7. In doing so, we find that the inequality is not satisfied at 3 but is satisfied at 5. Hence, we can conclude that the maximum response time of T_3 is between 3 and 5 and the task is schedulable. As a more complicated example, suppose that we are given a set of tasks with periods 2, 3, 5, and 11 and we want to determine whether the task with period 11 is schedulable. We check whether Eq. (6.6a) is satisfied at 2, 4, 6, 8, and 10, which are integer multiples of 2 less than 11; 3, 6, and 9, which are integer multiples of 3 less than 11; 5 and 10, which are integer multiples of 5 less than 11, and finally 11.

We now use the example in Figure 6–9 to explain intuitively why the time-demand analysis method is robust: The conclusion that a task T_i is schedulable remains correct when the execution times of jobs may be less than their maximum execution times and interrelease times of jobs may be larger than their respective periods. We replotted the time-demand functions for the two highest priority tasks in Figure 6–10(a). Suppose that the execution times of jobs in T_1 are in the range [0.9, 1] and those of jobs in T_2 are in the range [1.25, 1.5]. $w_2(t)$, calculated according to Eq. (6.5), is the maximum time-demand function of T_2. Its actual time-demand function can be anywhere in the shaded area. Clearly, the intersection of the actual time-demand function with the time supply line t is a point at most equal to 2.5. Similarly, the dotted lines in Figure 6–10(b) show the time-demand function $w_3(t)$ of T_3, which is calculated according to Eq. (6.5), as well as the contribution of T_1 and T_2 to the function. The dashed lines show the contribution of T_1 and T_2 to the actual time-demand function of T_3 when the actual interrelease times of the jobs in T_1 and T_2 are larger than their periods. We see again that the actual time-demand function of T_3, depicted by the solid line, lies below the time-demand function depicted by the dotted line. T_3 remains schedulable despite these variations in release times.

6.5.3 Alternatives to Time-Demand Analysis

For the simple case considered here, Theorem 6.5 lets us identify the worst-case condition for the schedulability of a fixed-priority system. Instead of carrying out a time-demand analysis, we can also determine whether a system of independent preemptable tasks is schedulable by simply simulating this condition and observing whether the system is then schedulable. In other words, a way to test the schedulability of such a system is to construct a schedule of it according to the given scheduling algorithm. In this construction, we assume that the tasks

are in phase and the actual execution times and interrelease times of jobs in each task T_i are equal to e_i and p_i, respectively. As long as Theorem 6.5 holds, it suffices for us to construct only the initial segment of length equal to the largest period of the tasks. If there is no missed deadline in this segment, then all tasks are feasibly scheduled by the given algorithm. Figure 6–11 shows the worst-case initial segment of the RM schedule of the system in Figure 6–9.

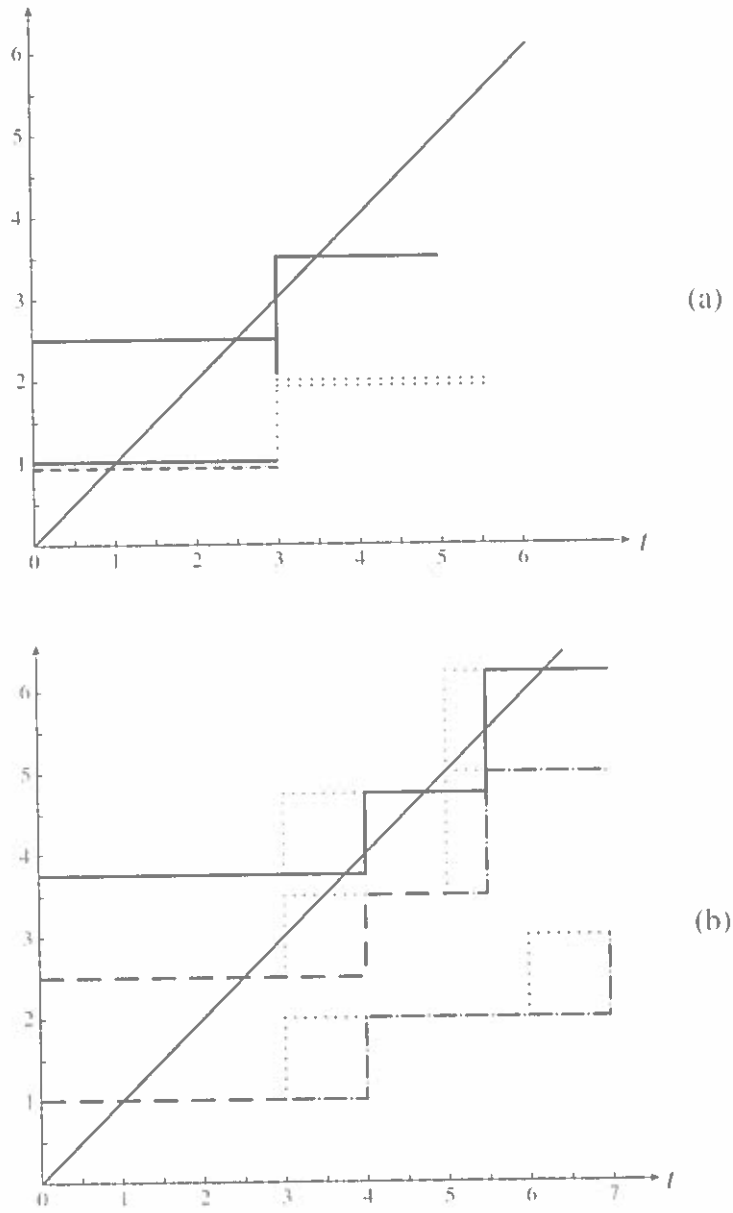

FIGURE 6–10 Actual time-demand functions.

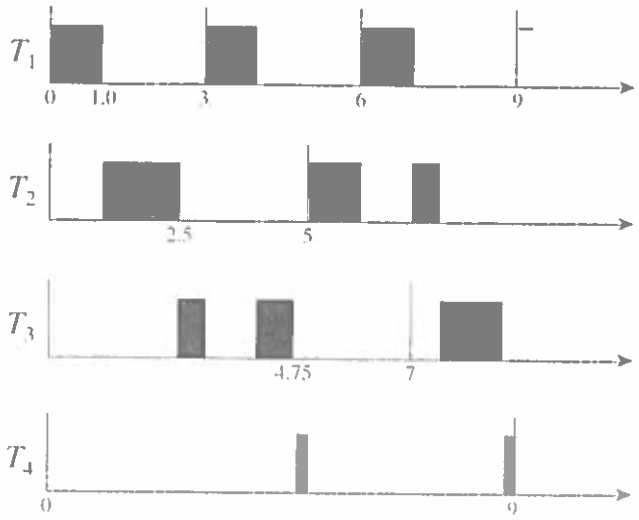

FIGURE 6–11 A worst-case schedule of (3, 1), (5, 1.5), (7, 1.25), and (9, 0.5).

This schedule allows us to draw the same conclusion about the maximum response times of the four tasks. We refer to this alternative as the *worst-case simulation method*. It is easy to see that the time complexity of this method is also $O(nq_{n,1})$, where $q_{n,1}$ is the ratio p_n/p_1.

For now, it appears that the more conceptually complex time-demand analysis method has no advantage over the simulation method. Later in the chapter, it will become evident that we can easily extend the time-demand analysis method to deal with other factors, such as nonpreemptivity and self-suspension, that affect the schedulability of a system, but we cannot easily extend the simulation method. When these factors must be taken into account, either we no longer know the worst-case condition for each task or the worst-case conditions for different tasks are different. In either case, the need to construct a large number of worst-case conditions greatly increases the complexity of the simulation method and makes it impractical for large systems.

If we want to know the maximum possible response time W_i of each task T_i, we must solve Eq. (6.4). This can be done in an iterative manner [ABTR], starting from an initial guess $t^{(1)}$ of W_i. (Since the response time of a job is at least equal to its execution time, we can always use e_i as the initial value.) During the lth iteration for $l \geq 1$, we compute the value $t^{(l+1)}$ on the right-hand side of Eq. (6.4) according to

$$t^{(l+1)} = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t^{(l)}}{p_k} \right\rceil e_k \tag{6.7}$$

We terminate the iteration either when $t^{(l+1)}$ is equal to $t^{(l)}$ and $t^{(l)} \leq p_i$ for some l or when $t^{(l+1)}$ becomes larger than p_i, whichever occurs sooner. In the former case, W_i is equal to $t^{(l)}$, and T_i is schedulable. In the latter case, we fail to find W_i, and T_i is not schedulable.

For example, we consider the task $T_3 = (7, 1.25)$ in Figure 6–9. Substituting the parameters of this task and the higher-priority tasks (3, 1) and (5, 1.5) into Eq. (6.4), we have

$$t = 1.25 + 1\left\lceil \frac{t}{3} \right\rceil + 1.5\left\lceil \frac{t}{5} \right\rceil$$

As an initial guess $t^{(1)}$ of W_3, we use its execution time 1.25. Evaluating the right-hand side of this equation, we find $t^{(2)}$ equal to 3.75. Substituting this value into the right-hand side again, we find $t^{(3)}$ equal to 4.75. $t^{(4)}$ calculated from $t^{(3)} = 4.75$ is equal to 4.75. Hence, W_3 is 4.75.

6.6 SCHEDULABILITY TEST FOR FIXED-PRIORITY TASKS WITH ARBITRARY RESPONSE TIMES

This section describes a general time-demand analysis method developed by Lehoczky [Leho] to determine the schedulability of tasks whose relative deadlines are larger than their respective periods. Since the response time of a task may be larger than its period, it may have more than one job ready for execution at any time. Ready jobs in the same task are usually scheduled on the FIFO basis. We assume here that this policy is used.

6.6.1 Busy Intervals

We will use the term level-π_i busy interval. A *level-π_i busy interval* $(t_0, t]$ begins at an instant t_0 when (1) all jobs in $\mathbf{T}_i$ released before the instant have completed and (2) a job in $\mathbf{T}_i$ is released. The interval ends at the first instant t after t_0 when all the jobs in $\mathbf{T}_i$ released since t_0 are complete. In other words, in the interval $(t_0, t]$, the processor is busy all the time executing jobs with priorities π_i or higher, all the jobs executed in the busy interval are released in the interval, and at the end of the interval there is no backlog of jobs to be executed afterwards. Hence, when computing the response times of jobs in $\mathbf{T}_i$, we can consider every level-π_i busy interval independently from other level-π_i busy intervals.

With a slight abuse of the term, we say that a level-π_i busy interval is *in phase* if the first jobs of all tasks that have priorities equal to or higher than priority π_i and are executed in this interval have the same release time. Otherwise, we say that the tasks have arbitrary phases in the interval.

As an example, Figure 6–12 shows the schedule of three tasks $T_1 = (2, 1)$, $T_2 = (3, 1.25)$, and $T_3 = (5, 0.25)$ in the first hyperperiod. The filled rectangles depict where jobs in T_1 are scheduled. The first busy intervals of all levels are in phase. The priorities of the tasks are $\pi_1 = 1$, $\pi_2 = 2$, and $\pi_3 = 3$, with 1 being the highest priority and 3 being the lowest priority. As expected, every level-1 busy interval always ends 1 unit time after it begins. For this system, all the level-2 busy intervals are in phase. They begin at times 0, 6, and so on which are the least common multiples of the periods of tasks T_1 and T_2. The lengths of these intervals are all equal to 5.5. Before time 5.5, there is at least one job of priority 1 or 2 ready for execution, but immediately after 5.5, there are none. Hence at 5.5, the first job in T_3 is scheduled. When this job completes at 5.75, the second job in T_3 is scheduled. At time 6, all the jobs released before time 6 are completed; hence, the first level-3 busy interval ends at this time. The second level-3 busy interval begins at time 6. This level-3 busy interval is not in phase since the release times of the first higher-priority jobs in this interval are 6, but the first job of T_3 in this interval is not released until time 10. The length of this level-3 busy

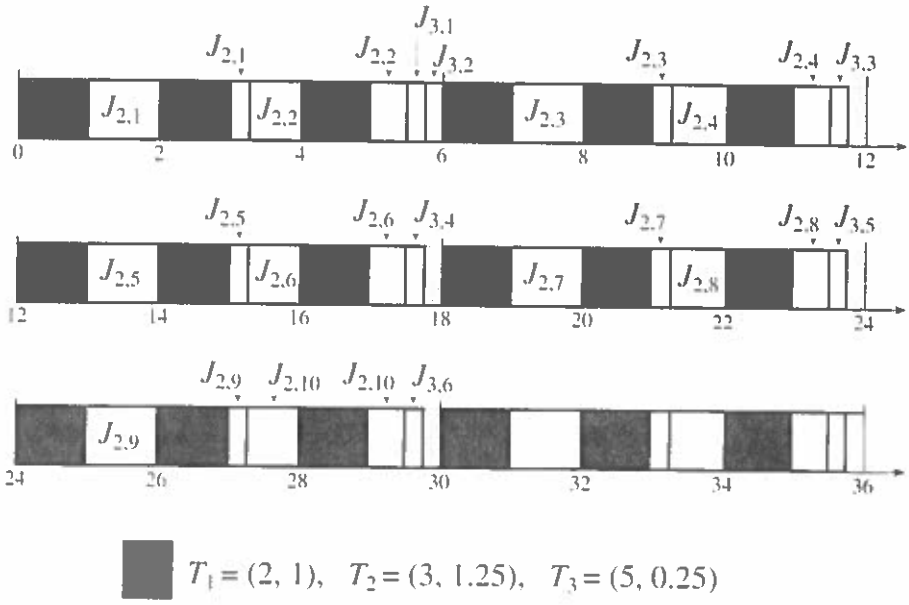

FIGURE 6–12 Example illustrating busy intervals.

interval is only 5.75. Similarly, all the subsequent level-3 busy intervals in the hyperperiod have arbitrary phases.

6.6.2 General Schedulability Test

The general schedulability test described below relies on the fact that when determining the schedulability of a task T_i in a system in which the response times of jobs can be larger than their respective periods, it still suffices to confine our attention to the special case where the tasks are in phase. However, the first job $J_{i,1}$ may no longer have the largest response time among all jobs in T_i. (This fact was illustrated by Lehoczky, *et al.* [LSST] by the example of two tasks: $T_1 = (70, 26)$ and $T_2 = (100, 62)$. Seven jobs of T_2 execute in the first level-2 busy interval. Their response times are 114, 102, 116, 104, 118, 106, and 94, respectively. The response times of both the third and fifth jobs in T_2 are larger than the response time of the first job.) Consequently, we must examine all the jobs of T_i that are executed in the first level-π_i busy interval. (Obviously, this busy interval is in phase when the tasks are in phase.) If the response times of all these jobs are no greater than the relative deadline of T_i, T_i is schedulable; otherwise, T_i may not be schedulable.

Specifically, the following general schedulability test developed by Lehoczky [Leho] for tasks with arbitrary relative deadlines works in this manner.

General Time-Demand Analysis Method

Test one task at a time starting from the highest priority task T_1 in order of decreasing priority. For the purpose of determining whether a task T_i is schedulable, assume that all the tasks are in phase and the first level-π_i busy interval begins at time 0.

While testing whether all the jobs in T_i can meet their deadlines (i.e., whether T_i is schedulable), consider the subset $\mathbf{T}_i$ of tasks with priorities π_i or higher.

(i) If the first job of every task in $\mathbf{T}_i$ completes by the end of the first period of the task, check whether the first job $J_{i,1}$ in T_i meets its deadline. T_i is schedulable if $J_{i,1}$ completes in time. Otherwise, T_i is not schedulable.

(ii) If the first job of some task in $\mathbf{T}_i$ does not complete by the end of the first period of the task, do the following:

(a) Compute the length of the in phase level-π_i busy interval by solving the equation $t = \sum_{k=1}^{i} \lceil \frac{t}{p_k} \rceil e_k$ iteratively, starting from $t^{(1)} = \sum_{k=1}^{i} e_k$ until $t^{(l+1)} = t^{(l)}$ for some $l \geq 1$. The solution $t^{(l)}$ is the length of the level-π_i busy interval.

(b) Compute the maximum response times of all $\lceil t^{(l)}/p_i \rceil$ jobs of T_i in the in-phase level-π_i busy interval in the manner described below and determine whether they complete in time.

T_i is schedulable if all these jobs complete in time; otherwise T_i is not schedulable.

It is easy to compute the response time of the first job $J_{i,1}$ of T_i in the first in-phase level-π_i busy interval. The time-demand function $w_{i,1}(t)$ is still given by the expression in the right-hand side of Eq. (6.5). An important difference is that the expression remains valid for all $t > 0$ before the end of the level-π_i busy interval. For the sake of convenience, we copy the expression here.

$$w_{i,1}(t) = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k, \qquad \text{for } 0 < t \leq w_{i,1}(t) \tag{6.8}$$

The maximum possible response time $W_{i,1}$ of $J_{i,1}$ is equal to the smallest value of t that satisfies the equation $t = w_{i,1}(t)$. To obtain $W_{i,1}$, we solve the equation iteratively and terminate the iteration only when we find $t^{(l+1)}$ equal to $t^{(l)}$. Because U_i is no greater than 1, this equation always has a finite solution, and the solution can be found after a finite number of iterations.

Let $W_{i,j}$ denote the maximum possible response time of the jth job in a level-π_i busy interval. The following lemma tells us how to compute $W_{i,j}$.

Lemma 6.6. The maximum response time $W_{i,j}$ of the jth job of T_i in an in-phase level-π_i busy period is equal to the smallest value of t that satisfies the equation

$$t = w_{i,j}(t + (j-1)p_i) - (j-1)p_i \tag{6.9a}$$

where $w_{i,j}(\,)$ is given by

$$w_{i,j}(t) = je_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k, \qquad \text{for } (j-1)p_i < t \leq w_{i,j}(t) \tag{6.9b}$$

As an example, Figure 6–13(a) shows the time-demand functions $w_{1,1}(t)$, $w_{2,1}(t)$ and $w_{3,1}(t)$ of the first jobs $J_{1,1}$, $J_{2,1}$ and $J_{3,1}$ in the system in Figure 6–12. Since $w_{2,1}(t)$ lies entirely above the supply t line from 0 to 3, $J_{2,1}$ does not complete at 3. Solving for the response time $W_{2,1}$ of this job, we find that $W_{2,1}$ equals 3.25. Similarly, the first intersection of $w_{3,1}(t)$ with the supply line t is at 5.75; this is the response time of $J_{3,1}$.

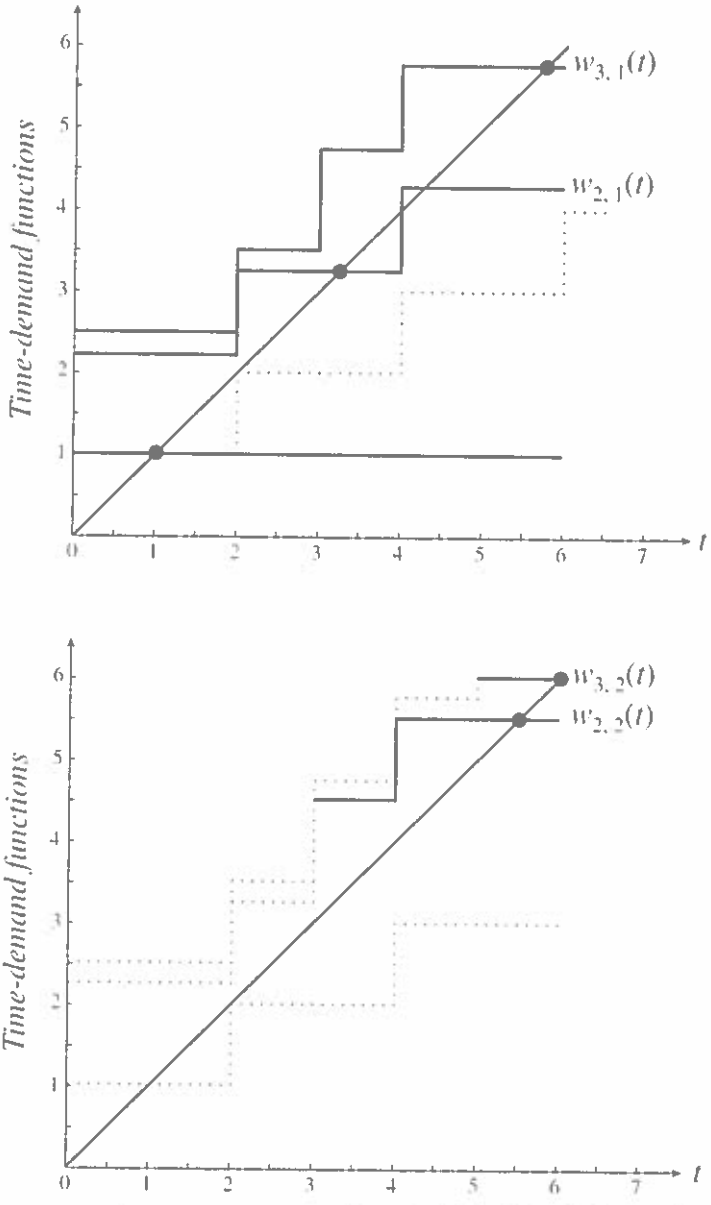

FIGURE 6–13 Time-demand functions of tasks $T_1 = (2, 1)$, $T_2 = (3, 1.25)$ and $T_3 = (5, 0.25)$.

Figure 6–13(b) shows the time-demand functions $w_{2,2}(t)$ and $w_{3,2}(t)$ of the second jobs of T_2 and T_3 in the first busy intervals, respectively. We see that these functions are equal to t when t is equal to 5.5 and 6, respectively. Therefore, $J_{2,2}$ completes at $t = 5.5$, and $J_{3,2}$ completes at $t = 6$. Subtracting their release times, we find that $W_{2,2}$ is 2.5 and $W_{3,2}$ is 1.

Computationally, it is more efficient to find $W_{i,j}$ by solving Eq. (6.9) iteratively in a manner similar to Eq. (6.7). For example, to find $W_{2,2}$, we substitute the parameters of the tasks into Eq. (6.9) to obtain

$$t = 2 \times 1.25 + \lceil (t + 3)/2 \rceil - 3$$

To solve this equation iteratively, we begin with the initial guess $t^{(1)}$ of 1.25, the execution time of $J_{2,2}$. Substituting t on the right-hand side of the equation by this value, we obtain $t^{(2)} = 2.5$. When we substitute t on the right-hand side by 2.5, we obtain $t^{(3)} = 2.5$. This allows us to terminate the iteration and conclude that $W_{2,2}$ is 2.5. Similarly, $W_{3,2} = 1$ is the minimum solution of the equation

$$t = 2 \times 0.25 + \lceil (t + 5)/2 \rceil + 1.25\lceil (t + 5)/3 \rceil - 5$$

Again, an alternative is the worst-case simulation approach. To determine whether T_i is schedulable, we generate the schedule of the tasks in $\mathbf{T}_i$ according to the given fixed-priority algorithm, assuming that the tasks are in phase. (The schedule in Figure 6–12 is an example.) If we do not observe any missed deadlines within the first level-π_i busy interval, we can conclude that all the tasks in $\mathbf{T}_i$ are schedulable. Otherwise, we cannot guarantee that all tasks in $\mathbf{T}_i$ always meet their deadlines.

Finally, to determine whether the jth job in an in-phase level-π_i busy interval completes in time, we can check whether the inequality $w_{i,j}(t) \le t$ is ever satisfied for some instant t in the range from $(j - 1)p_i$ to $(j - 1)p_i + D_i$. As with Eq. (6.6), we only need to check for the satisfiability of this inequality at time instants which are integer multiples of p_k, for $k = 1, 2, \ldots, i$, in this range of time.

*6.6.3 Correctness of the General Schedulability Test

The general schedulability test described above makes a key assumption: The maximum response time of some job $J_{i,j}$ in an in-phase level-π_i busy interval is equal to the maximum possible response time of all jobs in T_i. Therefore, to determine whether T_i is schedulable, we only need to check whether the maximum response times of all jobs in this busy interval are no greater than the relative deadline of T_i. This subsection presents several lemmas as proof that this assumption is valid.

We begin by considering an arbitrary job $J_{i,c}$ of T_i. When computing the maximum possible response time of this job, we must consider the possibility that some jobs released before the release time t_0 of $J_{i,c}$ may remain incomplete at the time. Since some of these jobs are executed before $J_{i,c}$, their processor time demands must be included in the time-demand function of $J_{i,c}$. As it turns out, because of Lemma 6.7, we need not be concerned with the jobs released before t_0 if $J_{i,c}$ is released at the same time as a job in every higher-priority task.

Lemma 6.7. Let t_0 be a time instant at which a job of every task in $\mathbf{T}_i$ is released. All the jobs in $\mathbf{T}_i$ released before t_0 have been completed at t_0.

Intuitively, we can see why this lemma is true by focusing on the latest level-π_i busy interval before t_0. Suppose that this interval begins at t_{-1}. We can use an argument similar to the one used in the proof of Theorem 6.1 to show that the amount of processor time demanded by all the jobs of priorities π_i or higher that are released in the interval (t_{-1}, t_0) cannot exceed $t_0 - t_{-1}$ since the total utilization U_i of $\mathbf{T}_i$ is no greater than 1. Consequently, at t_0 there is no backlog of jobs in $\mathbf{T}_i$, that is, every job in $\mathbf{T}_i$ released before t_0 is completed at t_0.

Lemma 6.7 tells us that any instant which is the release time of a job from every task in $\mathbf{T}_i$ is the beginning of an in-phase level-π_i busy interval. As a consequence, Lemma 6.8 is true.

LEMMA 6.8. When a system of independent, preemptive periodic tasks is scheduled on a processor according to a fixed-priority algorithm, the time-demand function $w_{i,1}(t)$ of a job in T_i released at the same time as a job in every higher-priority task is given by Eq. (6.8).

To see how the response time of an arbitrary job in a busy interval depends on the phases of the tasks for the interval, we again look at the first busy interval, and this time, we assume that each task T_k has an arbitrary phase ϕ_k. The response time of the jth job $J_{i,j}$ in the first level-π_i busy period is the smallest value of t that satisfies the equation

$$t = je_i + \sum_{k=1}^{i-1} \left\lceil \frac{t + \phi_i - \phi_k}{p_k} \right\rceil e_k - \phi_i - (j-1)p_i$$

Again, we take as the time origin the smallest of all the phases. The argument used in the proof of Theorem 6.5 can be used again to show that the response time of this job is the largest when all the tasks $\mathbf{T}_i$ have zero phase. This fact remains true for any arbitrary level-π_i busy interval, not just the first one. We restate this fact below.

LEMMA 6.9. The response time $W_{i,j}$ of the jth job of T_i executed in an in-phase level-π_i busy interval is no less than the response time of the jth job of T_i executed in any level-π_i busy interval.

Finally, the correctness of the general schedulability test depends on the following lemma

LEMMA 6.10. The number of jobs in T_i that are executed in an in-phase level-π_i busy interval is never less than the number of jobs in this task that are executed in a level-π_i busy interval of arbitrary phase.

Figure 6–12 illustrates this fact. Each of the later level-3 busy intervals is not in-phase. Only one job of T_3 executes in such an interval. In contrast, two jobs of T_3 execute in the first interval that is in phase. To see why this lemma is true in general, we note that during the time when each job of T_i is waiting to complete, new jobs with priorities π_i and higher are released and become ready. The rates at which the new jobs are released are independent of the response time of the waiting job. The longer this job of T_i waits, the more new jobs become ready. Hence, this lemma follows directly from Lemma 6.9.

In summary, Lemmas 6.7—6.10 tell us that the busy interval we examine according to the general time-demand analysis method contains the most number of jobs in T_i that can possibly execute in any level-π_i busy interval. Moreover, the response time of each job in the examined busy interval is larger than the response time of the corresponding job executed in any level-π_i busy interval. This is why if we find that none of the examined jobs completes late, we know that no job in T_i will.

6.7 SUFFICIENT SCHEDULABILITY CONDITIONS FOR THE RM AND DM ALGORITHMS

When we know the periods and execution times of all the tasks in an application system, we can use the schedulability test described in the last section to determine whether the system is schedulable according to the given fixed-priority algorithm. However, before we have completed the design of the application system, some of these parameters may not be known. In fact, the design process invariably involves the trading of these parameters against each other. We may want to vary the periods and execution times of some tasks within some range of values for which the system remains feasible in order to improve some aspects of the system. For this purpose, it is desirable to have a schedulability condition similar to the ones given by Theorems 6.1 and 6.2 for the EDF and the LST algorithms. These schedulability conditions give us a flexible design guideline for the choices of the periods and execution times of tasks. The schedulable utilizations presented in this section give us similar schedulability conditions for systems scheduled according to the RM or DM algorithms. An acceptance test based on such a schedulable utilization can decide whether to accept or reject a new periodic task in constant time. In contrast, the more accurate time-demand analysis test takes $O(nq_{n-1})$ time; moreover, the accurate test is less robust because its result is sensitive to the values of periods and execution times.

6.7.1 Schedulable Utilization of the RM Algorithm for Tasks with $D_i = p_i$

Specifically, the following theorem from [LiLa] gives us a schedulable utilization of the RM algorithm. We again focus on the case when the relative deadline of every task is equal to its period. For such systems, the RM and DM algorithms are identical.

THEOREM 6.11. A system of n independent, preemptable periodic tasks with relative deadlines equal to their respective periods can be feasibly scheduled on a processor according to the RM algorithm if its total utilization U is less than or equal to

$$U_{RM}(n) = n(2^{1/n} - 1) \tag{6.10}$$

$U_{RM}(n)$ is the schedulable utilization of the RM algorithm when $D_i = p_i$ for all $1 \le k \le n$. Figure 6–14 shows its value as a function of the number n of tasks in the set. When n is equal to 2, $U_{RM}(n)$ is equal to 0.828. It approaches ln 2 (0.693), shown by the dashed line, for large n.

Specifically, $U(n) \le U_{RM}(n)$ is a sufficient schedulability condition for any system of n independent, preemptable tasks that have relative deadlines equal to their respective periods to be schedulable rate-monotonically. (We use the notation $U(n)$ in place of U in our subsequent discussion whenever we want to bring the number of tasks n to our attention.) As long as

*6.7

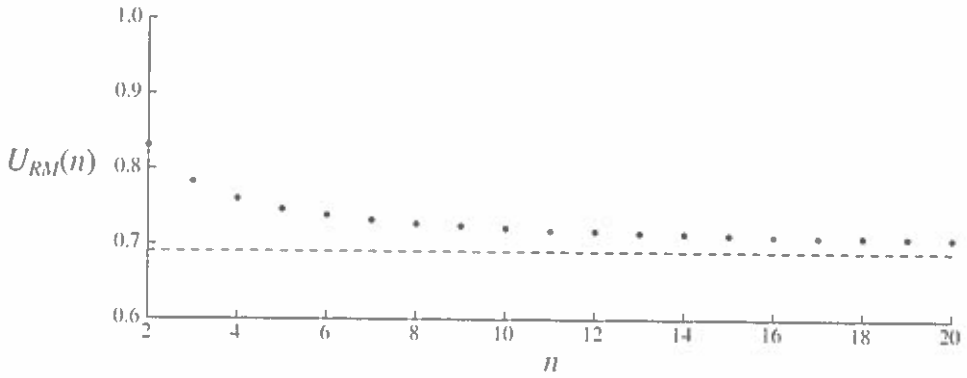

FIGURE 6–14 $U_{RM}(n)$ as a function n.

the total utilization of such a system satisfies this condition, it will never miss any deadline. In particular, we can reach this conclusion without considering the individual values of the phases, periods, and execution times.

As an example, we consider the system **T** of 5 tasks: (1.0, 0.25), (1.25, 0.1), (1.5, 0.3), (1.75, 0.07), and (2.0, 0.1). Their utilizations are 0.25, 0.08, 0.2, 0.04, and 0.05. The total utilization is 0.62, which is less than 0.743, the value of $U_{RM}(5)$. Consequently, we can conclude that we can feasibly schedule **T** rate-monotonically. Suppose that the system is later enhanced. As a result, the tasks are modified, and the resultant tasks are (0.3, 1.3, 0.1), (1.0, 1.5, 0.3), (1.75, 0.1), (2.0, 0.1), and (7.0, 2.45). Since their total utilization is 0.737, which is still less than 0.743, we know for sure that the system remains schedulable. There is no need for us to do the more complex time-demand analysis to verify this fact. On the other hand, suppose that to make the above five-task system more modular, we divide the task with period 7.0 into three smaller tasks with periods 5, 6, and 7, while keeping the total utilization of the system at 0.737. We can no longer use this condition to assure ourselves that the system is schedulable because $U_{RM}(7)$ is 0.724 and the total utilization of the system exceeds this bound.

Since $U(n) \leq U_{RM}(n)$ is not a necessary condition, a system of tasks may nevertheless be schedulable even when its total utilization exceeds the schedulable bound. For example, the total utilization of the system with the four tasks (3, 1), (5, 1.5), (7, 1.25), and (9, 0.5) is 0.85, which is larger than $U_{RM}(4) = 0.757$. Earlier in Figure 6–9, we have shown by the time-demand analysis method that this system is schedulable according to the RM algorithm.

*6.7.2 Proof of Theorem 6.11

While the schedulable utilization of the EDF algorithm given by Theorem 6.1 is intuitively obvious, the schedulable utilization $U_{RM}(n)$ given by Theorem 6.11 is not. We now present an informal proof of this theorem in order to gain some insight into why it is so.

The proof first shows that the theorem is true for the special case where the longest period p_n is less than or equal to two times the shortest period p_1. After the truth of the theorem is established for this special case, we then show that the theorem remains true when

this restriction is removed. As before, we assume that the priorities of all tasks are distinct. Here, this means that $p_1 < p_2 < \cdots < p_n$.

Proof for the Case of $p_n \leq 2p_1$. The proof for the case where $p_n \leq 2p_1$ consists of the four steps that are described below. Their goal is to find the most difficult-to-schedule system of n tasks among all possible combinations of n tasks that are difficult-to-schedule rate-monotonically. We say that a system is *difficult to schedule* if it is schedulable according to the RM algorithm, but it fully utilizes the processor for some interval of time so that any increase in the execution time or decrease in the period of some task will make the system unschedulable. The system sought here is the *most difficult* in the sense that its total utilization is the smallest among all difficult-to-schedule n-task systems. The total utilization of this system is the schedulable utilization of the RM algorithm, and any system with a total utilization smaller than this value is surely schedulable. Each of the following steps leads us closer to this system and the value of its total utilization.

Step 1: In the first step, we identify the phases of the tasks in the most difficult-to-schedule system. For this we rely on Theorem 6.5. You recall that according to that theorem, a job has its maximum possible response time if it is released at the same time as a job in every higher-priority task. The most difficult-to-schedule system must have one or more in-phase busy intervals. Therefore, in the search for this system, we only need to look for it among in-phase systems.

Step 2: In the second step, we choose a relationship among the periods and execution times and hypothesize that the parameters of the most difficult-to-schedule system of n tasks are thus related. In the next step, we will verify that this hypothesis is true. Again from Theorem 6.5, we know that in making this choice, we can confine our attention to the first period of every task. To ensure that the system is schedulable, we only need to make sure that the first job of every task completes by the end of the first period of the task. Moreover, the parameters are such that the tasks keep the processor busy once some task begins execution, say at time 0, until at least p_n, the end of the first period of the lowest priority task T_n.

The combination of n periods and execution times given by the pattern in Figure 6–15 meets these criteria. By construction, any system of n tasks whose execution times are related to their periods in this way is schedulable. It is easy to see that any increase in execution time of any task makes this system unschedulable. Hence systems whose parameters satisfy this relationship are difficult to schedule. Expressing analytically the dependencies of execution times on the periods of tasks that are given by Figure 6–15, we have

$$e_k = p_{k+1} - p_k \qquad \text{for } k = 1, 2, \ldots, n-1 \tag{6.11a}$$

Since each of the other tasks execute twice from 0 to p_n the execution time of the lowest priority task T_n is

$$e_n = p_n - 2\sum_{k=1}^{n-1} e_k \tag{6.11b}$$

Step 3: We now show that the total utilization of any difficult-to-schedule n-task system whose execution times are not related to their periods according to Eq. (6.11) is larger than or equal to the total utilization of any system whose periods and execution times are thus

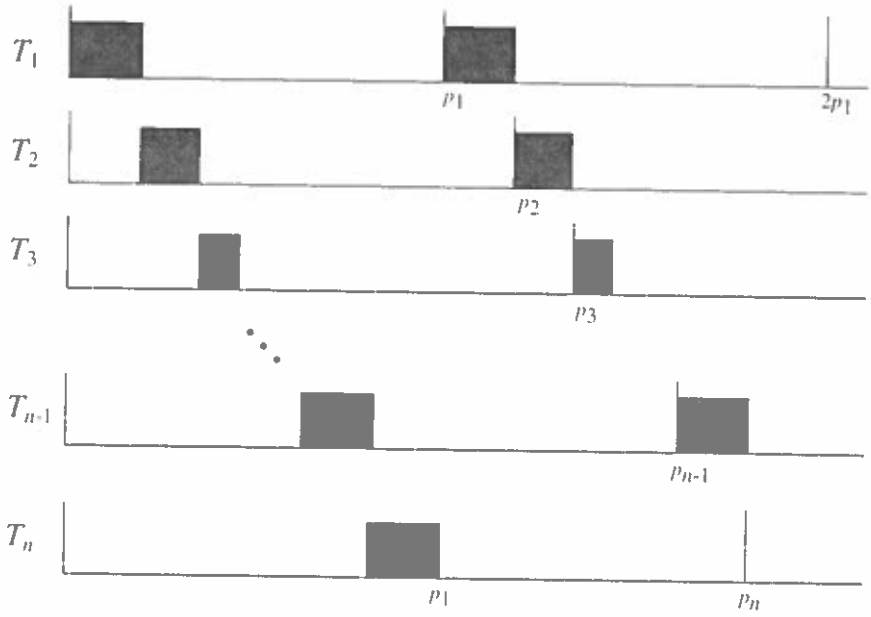

FIGURE 6–15 Relationship among parameters of difficult-to-schedule tasks.

related. Since we are looking for the difficult-to-schedule system with the least total utilization, we need not consider any system whose parameters are not thus related.

To do so, we construct new systems, whose parameters do not satisfy Eq. (6.11), from an original system whose parameters satisfy Eq. (6.11). There are two ways to do this. One way is by increasing the execution time of a higher-priority task from the value given by Eq. (6.11) by a small amount $\varepsilon > 0$. Without loss of generality, let this task be T_1. In other words, in the new system, the execution time e'_1 of T_1 is equal to

$$e'_1 = p_2 - p_1 + \varepsilon = e_1 + \varepsilon$$

For the new system to be schedulable, some other task must have a smaller execution time. From Figure 6–15, we see that the first job in every task can complete in time if we let the execution time of any other task be ε units less than the value given by Eq. (6.11a). Suppose that we choose T_k, for some $k \neq 1$, to be this task and make its new execution time equal to

$$e'_k = e_k - \varepsilon$$

The execution times of the tasks other than T_1 and T_k are still given by Eq. (6.11). The new system still keeps the processor busy in the interval $(0, p_n]$. The difference between the total utilization U' of the new system and the total utilization U of the original system is

$$U' - U = \frac{e'_1}{p_1} + \frac{e'_k}{p_k} - \frac{e_1}{p_1} - \frac{e_k}{p_k} = \frac{\varepsilon}{p_1} - \frac{\varepsilon}{p_k}$$

Since $p_1 < p_k$, this difference is positive, and the total utilization of the new system is larger. (You may want to convince yourself that we would reach the same conclusion if in the construction of the new system, we make the execution time of some task other than T_1 larger by ε units and make the execution times of one or more tasks with priorities lower than this task smaller by a total of ε units.)

Another way to construct a new difficult-to-schedule system from the original one is to let the execution time of a higher-priority task be ε units smaller than the value given by Eq. (6.11). Again, suppose that we choose T_1 to be this task, that is, its new execution time is

$$e_1'' = p_2 - p_1 - \varepsilon$$

From Figure 6–15, we see that if we do not increase the execution time of some other task, the processor will be idle for a total of 2ε units of time in $(0, p_n]$. To keep the processor busy throughout this interval and the system schedulable, we can increase the execution time of any of the other tasks by 2ε units, that is,

$$e_k'' = e_k + 2\varepsilon$$

for some $k \neq 1$. It is easy to see that with this increase accompanying the decrease in the execution time of T_1, the first job of every task in the new system can still complete by its deadline and the processor never idles from 0 to p_n. Comparing the total utilization U'' of this new system with that of the original system, we find that

$$U'' - U = \frac{2\varepsilon}{p_k} - \frac{\varepsilon}{p_1}$$

Since $p_k \leq 2p_1$ for all $k \neq 1$, this difference is never negative. (Again, we could also divide the 2ε units of time arbitrarily among the $n-1$ lower-priority tasks and get a new system with a total utilization larger than or equal to U.)

Step 4: As a result of step 3, we know that the parameters of the most difficult-to-schedule system of tasks must be related according to Eq. (6.11). To express the total utilization of a system whose parameters are given by Eq. (6.11) in terms of periods of the tasks in it, we substitute Eq. (6.11) into the sum $\sum_{k=1}^{n} e_k/p_k$ and thus obtain

$$U(n) = q_{2,1} + q_{3,2} + \cdots + q_{n,(n-1)} + \frac{2}{q_{2,1}q_{3,2}\cdots q_{n,(n-1)}} - n \tag{6.12}$$

where $q_{k,i}$, for $k > i$, is the ratio of the larger period p_k to the smaller period p_i, that is, $q_{k,i} = p_k/p_i$. In particular, the total utilization of any n-task system whose parameters are related according to Eq. (6.11) is a function of the $n-1$ *adjacent period ratios* $q_{k+1,k}$ for $k = 1, 2, \ldots, n-1$.

This equation shows that $U(n)$ is a symmetrical convex function of the adjacent period ratios. It has a unique minimum, and this minimum is the schedulable utilization $U_{RM}(n)$ of the RM algorithm. To find the minimum, we take the partial derivative of $U(n)$ with respect to each adjacent period ratio $q_{k+1,k}$ and set the derivative to 0. This gives us the following $n-1$ equation:

$$1 - \frac{2}{q_{2,1}q_{3,2}\cdots q_{(k+1),k}^2 \cdots q_{n,(n-1)}} = 0$$

for all $k = 1, 2, \ldots, n-1$.

Solving these equations for $q_{k+1,k}$, we find that $U(n)$ is at its minimum when all the $n-1$ adjacent period ratios $q_{k+1,k}$ are equal to $2^{1/n}$. Their product $q_{2,1}q_{3,2}\cdots q_{n,(n-1)}$ is the ratio $q_{n,1}$ of the largest period p_n to the smallest period p_1. This ratio, being equal to $2^{(n-1)/n}$, satisfies the constraint that $p_n \leq 2p_1$. Substituting $q_{k+1,k} = 2^{1/n}$ into the right-hand side of Eq. (6.12), we get the expression of $U_{RM}(n)$ given by Theorem 6.11.

For more insight, let us look at the special case where n is equal to 3. The total utilization of any difficult-to-schedule system whose parameters are related according to Eq. (6.11) is given by

$$U(3) = q_{2,1} + q_{3,2} + + \frac{2}{q_{3,2}q_{2,1}} - 3$$

$U(3)$ is a convex function of $q_{2,1}$ and $q_{3,1}$. Its minimum value occurs at the point $q_{2,1} = q_{3,2} \geq 2^{1/3}$, which is equal to 1.26. In other words, the periods of the tasks in the most difficult-to-schedule three-task system are such that $p_3 = 1.26p_2 = 1.59p_1$.

Generalization to Arbitrary Period Ratios. The ratio $q_{n,1} = p_n/p_1$ is the *period ratio* of the system. To complete the proof of Theorem 6.11, we must show that any n-task system whose total utilization is no greater than $U_{RM}(n)$ is schedulable rate-monotonically, not just systems whose period ratios are less than or equal to 2. We do so by showing that the following two facts are true.

1. Corresponding to every difficult-to-schedule n-task system whose period ratio is larger than 2 there is a difficult-to-schedule n-task system whose period ratio is less than or equal to 2.
2. The total utilization of the system with period ratio larger than 2 is larger than the total utilization of the corresponding system whose period ratio is less than or equal to 2.

Therefore, the restriction of period ratio being equal to or less than 2, which we imposed earlier in steps 1–4, leads to no loss of generality.

We show that fact 1 is true by construction. The construction starts with any difficult-to-schedule n-task system $\{T_i = (p_i, e_i)\}$ whose period ratio is larger than 2 and step-by-step transforms it into a system with a period ratio less than or equal to 2. Specifically, in each step, we find a task T_k whose period is such that $lp_k < p_n \leq (l+1)p_k$ where l is an integer equal to or larger than 2; the transformation completes when no such task can be found. In this step, we modify only this task and the task T_n with the largest period p_n. T_k is transformed into a new task whose period is equal to lp_k and whose execution time is equal to e_k. The period of the task with period p_n is unchanged, but its execution time is increased by $(l-1)e_k$. Figure 6–16 shows the original tasks and the transformed tasks. Clearly, the ratio of p_n and the period of the task transformed from T_k is less than or equal to 2, and the system thus obtained is also a difficult-to-schedule system. By repeating this step until $p_n \leq 2p_k$ for all $k \neq n$, we systematically transform the given system into one in which the ratio of p_n and the period of every other task is less than or equal to 2.

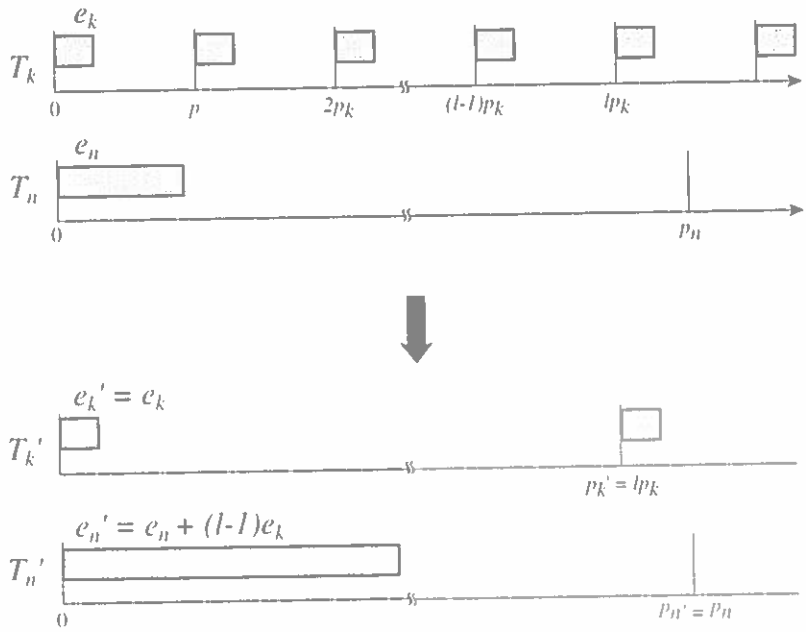

FIGURE 6–16 Transformation of two tasks.

To show fact 2 is true, we compute the difference between the total utilization of the system before the transformation of each task and the total utilization of the system after the transformation. This difference is

$$\frac{e_k}{p_k} - \frac{e_k}{lp_k} - \frac{(l-1)e_k}{p_n} = \left(\frac{1}{lp_k} - \frac{1}{p_n}\right)(l-1)e_k$$

which is larger than 0 because $lp_k < p_n$. This allows us to conclude that the system with a period ratio less than 2 obtained when the transformation completes has a smaller total utilization than the given system.

*6.7.3 Schedulable Utilization of RM Algorithm as Functions of Task Parameters

When some of the task parameters are known, this information allows us to improve the schedulable utilization of the RM algorithm. We now give several schedulable utilizations that are larger than $U_{RM}(n)$ for independent, preemptive periodic tasks whose relative deadlines are equal to their respective periods. These schedulable utilizations are expressed in terms of known parameters of the tasks, for example, the utilizations of individual tasks, the number n_p of disjoint subsets each containing simply periodic tasks, and some functions of the periods of the tasks. The general schedulable utilization $U_{RM}(n)$ of the RM algorithm is the minimum value of these specific schedulable utilizations. Because they are larger than $U_{RM}(n)$, when applicable, these schedulable utilizations are more accurate criteria of schedulability. They are particularly suited for on-line acceptance tests. When checking whether a new periodic task can be scheduled with existing tasks, many of the task parameters are already known.

and computing one of these schedulable utilizations takes a constant amount of time, much less than the time required to do a time-demand analysis.

Schedulable Utilization $U_{RM}(u_1, u_2, \ldots u_n)$ as a Function of Task Utilizations. Rather than replacing the individual periods in Eq. (6.11a) by adjacent period ratios as we did earlier, we rewrite the equation as follows:

$$p_{k+1} = p_k(1 + u_k) \qquad \text{for } k = 1, 2, \ldots, n-1$$

Moreover, from Eq. (6.11b) and the fact that $p_n \leq 2p_1$, we can conclude that

$$p_n(1 + u_n) \leq 2p_1$$

Combining these two expressions, we have the following corollary.

> **COROLLARY 6.12.** *n independent, preemptable periodic tasks with relative deadlines equal to their respective periods are schedulable rate-monotonically if their utilizations $u_1, u_2, \ldots, u_n$ satisfy the inequality*
>
> $$(1 + u_1)(1 + u_2) \cdots (1 + u_n) \leq 2 \tag{6.13}$$

We denote the total utilization of the tasks whose utilizations satisfy the constraint Eq. (6.13) by $U_{RM}(u_1, u_2, \ldots, u_n)$.

As an example, we consider a system of two tasks T_1 and T_2. The schedulable utilization $U_{RM}(u_1, u_2)$ of the system is equal to 0.957, 0.899, 0.861, and 0.828, respectively, when the ratio u_1/U of the utilization of T_1 to the total utilization of both tasks is equal to 0.05, 0.1, 0.25, and 0.5. The minimum of $U(u_1, u_2)$ is at the point $u_1 = 0.5U$ (i.e., when $u_1 = u_2$) and is 0.828, the Liu and Layland bound for n equal to 2.

For arbitrary n, the inequality Eq. (6.13) becomes $(1 + U(n)/n)^n \leq 2$ when the utilizations of all the tasks are equal. For this combination of utilizations, the inequality Eq. (6.13) becomes the same as the Liu and Layland bound $U(n) \leq n(2^{1/n} - 1)$.

Schedulable Utilization of Subsets of Simply Periodic Tasks. We now consider a system of periodic tasks that are not simply periodic but can be partitioned into n_h subsets of simply periodic tasks. For example, we can partition the system $\mathbf{T}$ of tasks with periods 4, 7, 8, 14, 16, 28, 32, 56, and 64 into two subsets $\mathbf{Z}_1$ and $\mathbf{Z}_2$. $\mathbf{Z}_1$ contains the tasks with period 4, 8, 16, 32, and 64; and $\mathbf{Z}_2$ contains tasks with periods 7, 14, 28, and 56. Let $U(\mathbf{Z}_1)$ and $U(\mathbf{Z}_2)$ denote the total utilization of the tasks in $\mathbf{Z}_1$ and $\mathbf{Z}_2$, respectively. Kuo, *et al.* [KuMo91] have shown that if $U(\mathbf{Z}_1) + U(\mathbf{Z}_2) \leq 0.828$ [i.e., $U_{RM}(2)$], all these tasks are schedulable rate-monotonically. In contrast, if we were to treat the tasks separately, we would have to use the bound $U_{RM}(9)$, which is only 0.712.

The following theorem by Kuo, *et al.* [KuMo91] states this fact in general.

> **THEOREM 6.13.** *If a system $\mathbf{T}$ of independent, preemptable periodic tasks, whose relative deadlines are equal to their respective periods, can be partitioned into n_h disjoint*

subsets, $\mathbf{Z}_1, \mathbf{Z}_2, \ldots, \mathbf{Z}_{n_h}$, each of which contains simply periodic tasks, then the system is schedulable rate-monotonically if the total utilizations $U(\mathbf{Z}_i)$, for $i = 1, 2, \ldots, n_h$, of the tasks in the subsets satisfy the inequality

$$(1 + U(\mathbf{Z}_1))(1 + U(\mathbf{Z}_2)) \cdots (1 + U(\mathbf{Z}_{n_h})) \leq 2.$$

It follows that such a system $\mathbf{T}$ is schedulable rate-monotonically if its total utilization is equal to or less than $U_{RM}(n_h)$.

To see intuitively why this theorem is true, we replace each subset $\mathbf{Z}_i$ by a single periodic task T_i'. The period p_i' of this task is equal to the shortest period all the tasks in $\mathbf{Z}_i$, and its execution time is equal to $p_i' U(\mathbf{Z}_i)$. Clearly, the set of n_h new tasks T_i''s are schedulable rate-monotonically if either one of the conditions in Theorem 6.13 is satisfied. The fact that T_i' is schedulable means that within each interval of length p_i', T_i' has $p_i' U(\mathbf{Z}_i)$ units of processor time. Because the tasks in $\mathbf{Z}_i$ are simply periodic, the total amount of time demanded by all the tasks during any period p ($\geq p_i'$) of any task in $\mathbf{Z}_i$ is equal to $pU(\mathbf{Z}_i) = k p_i' U(\mathbf{Z}_i)$ for some integer k. Since this demand is met by any schedule in which all jobs in T_i' complete in time, every job in any task in $\mathbf{Z}_i$ can always complete in time. (This argument is not a proof because it neglects the fact that tasks with longer periods are scheduled at lower priorities. A complete proof of this theorem can be found in [KuMo91].)

The schedulable utilization given by Theorem 6.13 is particularly useful for any system that contains a small number n of large application modules. If we can make the tasks in each module simply periodic, we do not need to be concerned with the number of tasks in each module. Provided that the total utilization of all the modules is no greater than $U_{RM}(n)$, we are assured of the schedulability of all the tasks in the system. For example, in a system containing a large number of multirate controllers, n is the rate groups (i.e., the number of controllers). We need not be concerned with how many control-law computations per controller the system must perform when the rates in each group are related in a harmonic way.

Dependency of Schedulable Utilization on Periods of Tasks. The schedulable utilization of the RM algorithm depends on the periods of tasks in two ways. First, it increases with the minimum adjacent period ratio and, hence, the period ratio of the system, and second, it increases as the tasks become closer to being simply periodic.

Dependency on Period Ratio. The fact that the schedulable utilization of the RM algorithm increases with the period ratio of the system has been demonstrated statistically in a study by Lehoczky, *et al.* [LeSD] who simulated systems in which task periods are uniformly distributed over a wide range. We expect the schedulable utilization of the RM algorithm to increase with the minimum adjacent period ratio for the following reason. During any period of a task T_i, it is possible for a higher priority task T_k ($k < i$) to consume an extra e_k units of processor time beyond $u_k p_i$. This is why the schedulable utilization $U_{RM}(i)$ is less than 1. However, when the ratio p_i / p_k becomes larger, the extra amount e_k becomes a smaller fraction of $u_k p_i$. In the limit as the adjacent period ratios approach infinity, the amount of time in each period of T_i used by higher-priority tasks approaches $U_{i-1} p_i$, the amount of time available to T_i approaches $1 - U_{i-1}$, and the schedulable utilization of the i tasks approaches 1.

We can also observe from the proof of Theorem 6.11 that the period ratio of the most difficult-to-schedule n-task system is $2^{(n-1)/n}$, which is equal to 1.414 when n is equal to 2 and is approximately equal to 2 when n is large. Moreover in this system, the ratios of all $n-1$ adjacent periods are equal to $2^{1/n}$. The total utilization of a difficult-to-schedule system increases as the minimum adjacent period ratio increases. Hence, the performance of the RM algorithm in terms of its schedulable utilization is the worst when the minimum adjacent period ratio of the system is $2^{1/n}$ and improves as the ratio increases.

Dependency on Values of Periods. Theorem 6.3 tells us that any number of simply periodic, independent, preemptable tasks can be feasibly scheduled rate-monotonically as long as their total utilization is no more than 1. We, therefore, expect that the closer the tasks are to being simply periodic, the larger the total schedulable utilization of the RM algorithm is. The schedulable utilization found by Burchard, *et al.* [BLOS] quantifies this statement.

Specifically, Burchard, *et al.* [BLOS] expresses the schedulable utilization in terms a parameter ζ which measures how far the tasks of a system deviate from being simply periodic. ζ is defined as follows:

$$\zeta = \max_{1 \le i \le n} X_i - \min_{1 \le i \le n} X_i \tag{6.14a}$$

where

$$X_i = log_2 p_i - \lfloor \log_2 p_i \rfloor \tag{6.14b}$$

THEOREM 6.14. The schedulable utilization $U_{RM}(\zeta, n)$ of the RM algorithm as a function of ζ and n is given by

$$U_{RM}(n, \zeta) = \begin{cases} (n-1)(2^{\zeta/(n-1)} - 1) + 2^{1-\zeta} - 1 & \text{for } \zeta < 1 - 1/n \\ n(2^{1/n} - 1) & \text{for } \zeta \ge 1 - 1/n \end{cases} \tag{6.14c}$$

Theorem 6.11 follows straightforwardly from this theorem.

Before we examine the behavior of $U_{RM}(n, \zeta)$ as ζ varies, we first examine how ζ depends on the periods of the tasks. To do so, we write p_i as 2^{x_i} for some $x_i > 0$. If x_i is an integer, the value X_i defined above is 0. X_i increases as x_i increases and deviates more from the integer value and becomes 0 again when it assumes the next larger integer value. For example, X_i is 0 if $p_i = 2^3 = 8$. It has the values 0.322, 0.585, and 0.807 when p_i is equal to 10, 12, and 14, respectively. Similarly, X_i is 0 when p_i is equal to $2^4 = 16$ (or $2^2 = 4$) and is equal to these values when p_i is equal to 20, 24, and 28 (or 5, 6, and 7), respectively. Hence, for a system of n tasks, ζ is equal to 0 when the period p_i of every task T_i is equal to $y2^{x_i}$ for some $y > 0$ independent of i and some positive integer x_i. (In other words, the period of every task is divisible by some power of 2.) In the extreme when ζ approaches one, there must be a task whose period is slightly larger than some power of 2 and some other task whose period is slightly smaller than some power of 2 (i.e., the periods are relatively prime).

Figure 6–17 plots $U_{RM}(n, \zeta)$ as a function of ζ and n. In the region where $\zeta < 1 - 1/n$, $U_{RM}(n, \zeta)$ is larger than the general schedulable utilization $U_{RM}(n)$ given by Eq. (6.10). In particular, as ζ approaches zero, $U_{RM}(n, \zeta)$ approaches one for all n as expected.

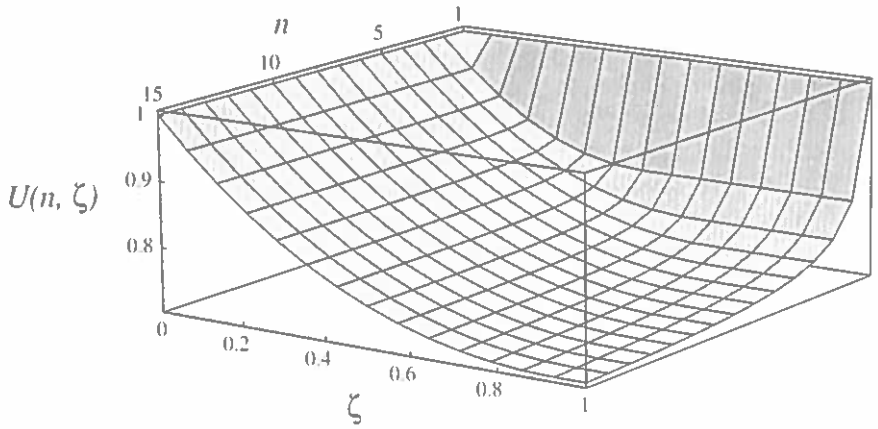

FIGURE 6–17 $U_{RM}(n, \zeta)$ as a function of n and ζ.

Yet Another Schedulability Test Making Use of Task Periods. For a given system $\mathbf{T}$ of tasks, let $\mathbf{T}'$ be a set of tasks which has the following property. There is a task T_i' in $\mathbf{T}'$ if and only if there is task T_i in $\mathbf{T}$. Moreover, the execution time of T_i' is equal to the execution time of T_i, and the period of T_i' is shorter than the period of T_i. $\mathbf{T}'$ is called an *accelerated set* of $\mathbf{T}$.

We now try to make use of the fact demonstrated by the example in Figure 6–9 at the end of Section 6.5.2. It follows from the observation made there that any system $\mathbf{T}$ is schedulable according to a fixed-priority algorithm if it has an accelerated set $\mathbf{T}'$ that is schedulable according to the algorithm. In particular, we have the following theorem regarding the RM algorithm.

> **THEOREM 6.15.** A system $\mathbf{T}$ of independent, preemptive periodic tasks whose relative deadlines are equal to their respective periods is schedulable according the RM algorithm if it has an accelerated set $\mathbf{T}'$ which is simply periodic and has a total utilization equal to or less than 1.

*6.7.5

The theorem follows from the above observation and Theorem 6.3. Han [Han] gives an $O(n^2 \log n)$ algorithm and an $O(n^3)$ algorithm that apply Theorem 6.15 to determine the schedulability of rate-monotonically scheduled systems. The more complex algorithm is more accurate. Both are more accurate than the test based on the schedulable utilizations [Eqs. (6.10) and (6.14)]. Section 10.2.3 will describe an algorithm that can be used for this purpose as well.

6.7.4 Schedulable Utilization of Fixed Priority Tasks with Arbitrary Relative Deadlines

Obviously, a system of n tasks with a total utilization $U_{RM}(n)$ may not be schedulable rate-monotonically when the relative deadlines of some tasks are shorter than their periods. On the other hand, if the relative deadlines of the tasks are larger than the respective task periods, we expect the schedulable utilization of the RM algorithm to be larger than $U_{RM}(n)$. We now consider the case where the relative deadline D_i of every task is equal to δ times its period

p_i for some $0 < \delta$. The following theorem proven by Lehoczky *et al.* [Leho, LeSh, LSST] gives us a schedulable utilization $U_{RM}(n, \delta)$ as a function of δ and the number n of tasks. The schedulable utilization $U_{RM}(n)$ given by Theorem 6.11 is equal to this upper bound in the special case where $\delta = 1$.

THEOREM 6.16. A system of n independent, preemptable periodic tasks with relative deadlines $D_i = \delta p_i$ for all $1 \le i \le n$ is schedulable rate-monotonically if its total utilization is equal to or less than

$$U_{RM}(n, \delta) = \delta(n-1)\left[\left(\frac{\delta+1}{\delta}\right)^{1/n-1} - 1\right], \quad \text{for } \delta = 2, 3, \ldots \tag{6.15}$$

$$n((2\delta)^{1/n} - 1) + 1 - \delta, \qquad 0.5 \le \delta \le 1$$

$$\delta, \qquad 0 \le \delta \le 0.5$$

Figure 6–18 lists the values of $U_{RM}(n, \delta)$ for several values of n and δ. For any n, this schedulable utilization is larger than $U_{RM}(n)$ when δ is an integer larger than 1. In the limit when n is equal to infinity, this upper bound on the total utilization approaches $\delta \ln((\delta+1)/\delta)$, for $\delta = 1, 2, \ldots$. It approaches 1 as δ approaches infinity, as expected; since $U(n)$ is no greater than one, every job eventually completes.

Figure 6–18 also plots $U_{RM}(n, \delta)$ as a function of n and δ in the range $0 \le \delta \le 1$. We see that in this range of δ, $U_{RM}(n, \delta)$ decreases from $U_{RM}(n)$ given by Eq. (6.10) as δ decreases.

As an example, let us consider the system consisting of tasks (3, 0.6), (4, 1.0) and (5, 1). The total utilization of the tasks is 0.65. Since $U_{RM}(3)$ is equal to 0.779, the tasks are schedulable. Now, suppose that as a way to control completion-time jitters, we require that every job completes in half the period; in other words, δ is equal to 0.5. Because $U_{RM}(3, 0.5)$ is 0.5, the schedulable utilization does not ensure us that the tasks are schedulable. In fact, the task (5, 1) is not schedulable. However, if δ is equal to 0.69, $U_{RM}(3, \delta)$ is equal to 0.65 and hence, it guarantees that the tasks are schedulable.

*6.7.5 Schedulable Utilization of the RM Algorithm for Multiframe Tasks

We know that the periodic task model is sometimes inaccurate, and the prediction on schedulability based on the model can be unduly pessimistic. An example is a task that models the transmission of an MPEG compressed video over a network link. Jobs in this task, modeling the transmissions of individual video frames, are released periodically. Because the size of I-frames can be very large compared with that of B- and P-frames, the execution times of the jobs can vary widely. When modeled as a periodic task, the execution time of the task is equal to the transmission time of an I-frame. Hence, if we were to determine whether a system of such tasks is schedulable based on the schedulability tests described above, we would surely underutilize the processor. The *multiframe task model* developed by Mok and Chen [MoCh] is a more accurate model and leads to more accurate schedulability tests.

The example used by Mok and Chen to motivate the multiframe task model is a system of two tasks: T_1 and T_2. T_2 is a task with period 5 and execution time 1. The period of T_1 is 3. The maximum execution time of $J_{1,k}$ is equal to 3, if k is odd and is equal to 1 if k is

n	$\delta = 4.0$	$\delta = 3.0$	$\delta = 2.0$	$\delta = 1.0$	$\delta = 0.9$	$\delta = 0.8$	$\delta = 0.7$	$\delta = 0.6$	$\delta = 0.5$
2	0.944	0.928	0.898	0.828	0.783	0.729	0.666	0.590	0.500
3	0.926	0.906	0.868	0.779	0.749	0.708	0.656	0.588	0.500
4	0.917	0.894	0.853	0.756	0.733	0.698	0.651	0.586	0.500
5	0.912	0.888	0.844	0.743	0.723	0.692	0.648	0.585	0.500
6	0.909	0.884	0.838	0.734	0.717	0.688	0.646	0.585	0.500
7	0.906	0.881	0.834	0.728	0.713	0.686	0.644	0.584	0.500
8	0.905	0.878	0.831	0.724	0.709	0.684	0.643	0.584	0.500
9	0.903	0.876	0.829	0.720	0.707	0.682	0.642	0.584	0.500
∞	0.892	0.863	0.810	0.693	0.687	0.670	0.636	0.582	0.500

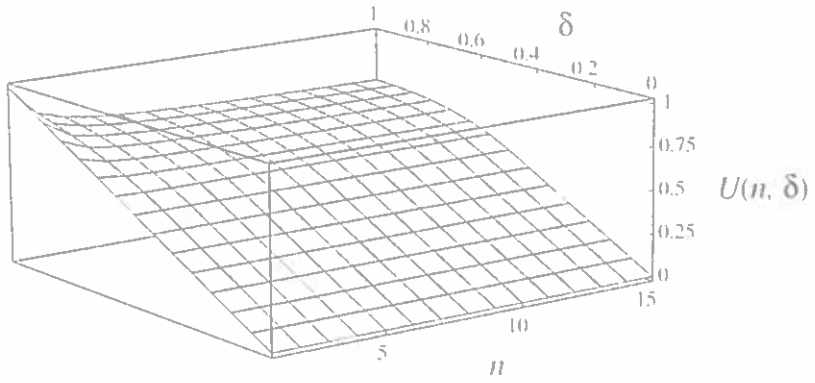

FIGURE 6–18 $U_{RM}(n, \delta)$ as a function of n for $0 \leq \delta \leq 1$. (a) Values of $U_{RM}(n, \delta)$. (b) Behavior of $U_{RM}(n, \delta)$ in the range of $\delta = 1$.

even. The relative deadlines of the tasks are equal to their respective periods. We can treat T_1 as the periodic task (3, 3), but if we were to do so, we would conclude that the system is not schedulable. This conclusion would be too pessimistic because the system is in fact schedulable. Indeed, as we will see shortly, by modeling T_1 more accurately as a multiframe task, we can reach the correct conclusion.

Specifically, in the multiframe task model, each (multiframe) task T_i is characterized by a 4-tuple $(p_i, \xi_i, e_i', e_i'')$. In the 4-tuple, p_i is the period of the task and has the same meaning as the period of a periodic task. Jobs in T_i have either one of two possible maximum execution times: e_i' and e_i'', where $e_i' \geq e_i''$. The former is its *peak execution time*, and the latter is its *normal execution time*. Each period which begins at the release time of a job with the peak execution time is called a *peak frame*, and the other periods are called *normal frames*. Each peak frame is followed by $\xi_i - 1$ normal frames, which in turn are followed by a

6.8

peak frame and so on. The utilization u_i of the multiframe task $T_i = (p_i, \xi_i, e_i^p, e_i^n)$ is equal to $(e_i^p + (\xi_i - 1)e_i^n)/\xi_i p_i$ when ξ_i is larger than 1. A periodic task (p_i, e_i) is the special multiframe task $(p_i, 1, e_i, e_i)$.

According to the multiframe task model, the task T_1 in the example above is (3, 2, 3, 1); its period is 3, peak execution time is 3, normal execution time is 1, and each peak frame is followed by one normal frame. It utilization is equal to $(3 + 1)/6 = 0.667$. The task (33, 6, 1.0, 0.3) can model an MPEG video transmission task. The period of the task is 33 milliseconds. The execution time of the job in each peak frame, which models the transmission of an I-frame in the video stream, is never more than one millisecond. It is followed by five normal frames. The execution times of jobs released in normal frames are never more than 0.3. These jobs model the transmissions of B- and P-frames in the video. They are followed by the transmission of an I-frame, that is, a peak frame, which is in turn followed by five normal frames, and so on.

We observe that the response time of a job $J_{i,k}$ in T_i has the maximum possible value if the kth period, which begins when $J_{i,k}$ is released, is a peak frame and this peak frame begins at the same time as a peak frame in every high-priority task. In other words, a critical instant of a multiframe task T_i occurs under this condition. (The proof of this claim is left to you as an exercise.) Consequently, when the relative deadlines D_i's are such that $D_k \leq p_k$ for all $1 \leq k \leq n$, a task T_i is schedulable if the job in the task released at a critical instant completes in time.

Given a system of n multiframe tasks, the *load variation*, denoted by Ξ, of the system is $\min_{1 \leq i \leq n}(e_i^p/e_i^n)$. We expect that the schedulable utilization of the RM algorithm is an increasing function of the load variation. Mok and Chen [MoCh] found that it depends on Ξ as stated below.

THEOREM 6.17. A system of n independent, preemptable multiframe tasks, whose relative deadlines are equal to the respective periods, is schedulable according to the RM algorithm if their total utilization is no greater than

$$U_{RM}(n, \Xi) = \Xi n \left(\left(\frac{\Xi + 1}{\Xi} \right)^{1/n} - 1 \right) \tag{6.16}$$

Figure 6–19 shows this schedulable utilization as function of Ξ and n. Indeed, as Ξ increases, this upper bound increases. However, we would still declare the tasks $T_1 = (3, 2, 3, 1)$ and $T_2 = (5, 1, 1, 1)$ mentioned earlier unschedulable if we were to use this schedulable utilization as the criterion of schedulability. The load variation of this system is 1. $U_{RM}(n, \Xi = 1)$ is 0.828, but the total utilization of this system is 0.867.

6.8 PRACTICAL FACTORS

Thus far, we have assumed that every job is preemptable at any time; once a job is released, it never suspends itself and hence is ready for execution until it completes; scheduling and context-switch overhead is negligible; the scheduler is event-driven and acts immediately upon event occurrences; every task (or job) has a distinct priority; and every job in a fixed-