CHAPTER 3

TASK ASSIGNMENT AND SCHEDULING

3.1 INTRODUCTION

The purpose of real-time computing is to execute, by the appropriate deadlines, its critical control tasks. In this chapter, we will look at techniques for allocating and scheduling tasks on processors to ensure that deadlines are met.

There is an enormous literature on scheduling: indeed, the field appears to have grown exponentially since the mid-1970s! We present here the subset of the results that relates to the meeting of deadlines.

The allocation/scheduling problem can be stated as follows. Given a set of tasks, task precedence constraints, resource requirements, task characteristics, and deadlines, we are asked to devise a feasible allocation/schedule on a given computer. Let us define some of these terms.

Formally speaking, a *task* consumes resources (e.g., processor time, memory, and input data), and puts out one or more results. Tasks may have *precedence constraints*, which specify if any task(s) needs to precede other tasks. If task T_i's output is needed as input by task T_j, then task T_j is constrained to be preceded by task T_i. The precedence constraints can be most conveniently represented by means of a *precedence graph*. We show an example of this in Figure 3.1. The arrows indicate which task has precedence over which other task. We denote the precedent-task set of task T by $\prec(T)$; that is, $\prec(T)$ indicates which tasks must be completed before T can begin.

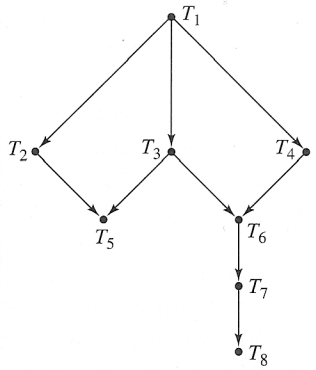

FIGURE 3.1
Example of a precedence graph.

Example 3.1. In Figure 3.1, we have

$$\begin{aligned}
&\prec(1) = \emptyset \\
&\prec(2) = \{1\} \\
&\prec(3) = \{1\} \\
&\prec(4) = \{1\} \\
&\prec(5) = \{1, 2, 3\} \\
&\prec(6) = \{1, 3, 4\} \\
&\prec(7) = \{1, 3, 4, 6\} \\
&\prec(8) = \{1, 3, 4, 6, 7\}
\end{aligned} \tag{3.1}$$

We can also write $i \prec j$ to indicate that task T_i must precede task T_j. (We can also express this by $j \succ i$). Commonly, for economy of representation, we only list the immediate ancestors in the precedence set; for example, we can write $\prec(5) = \{2, 3\}$ since $\prec(2) = \{1\}$. The precedence operator is transitive. That is,

$$i \prec j \quad \text{and} \quad j \prec k \quad \Rightarrow i \prec k$$

In some cases, $\succ$ and $\prec$ are used to denote which task has higher priority; that is, $i \succ j$ can mean that T_i has higher priority than T_j. The meaning of these symbols should be clear from context.

Each task has *resource requirements*. All tasks require some execution time on a processor. Also, a task may require a certain amount of memory or access to a bus. Sometimes, a resource must be *exclusively* held by a task (i.e., the task must have sole possession of it). In other cases, resources are nonexclusive. The same physical resource may be exclusive or nonexclusive depending on the operation to be performed on it. For instance, a memory object (or anything else to which writing is not atomic) that is being read is nonexclusive. However, while it is being written into, it must be held exclusively by the writing task.

The *release time* of a task is the time at which all the data that are required to begin executing the task are available, and the *deadline* is the time by which the task must complete its execution. The deadline may be hard or soft, depending on the nature of the corresponding task. The *relative deadline* of a task is the absolute deadline minus the release time. That is, if task T_i has a relative deadline

d_i and is released at time τ, it must be executed by time $\tau + d_i$. The *absolute deadline* is the time by which the task must be completed. In this example, the absolute deadline of T_i is $\tau + d_i$.

A task may be periodic, sporadic, or aperiodic. A task T_i is *periodic* if it is released periodically, say every P_i seconds. P_i is called the *period* of task T_i. The periodicity constraint requires the task to run exactly once every period; it does not require that the tasks be run exactly one period apart. Quite commonly, the period of a task is also its deadline. The task is *sporadic* if it is not periodic, but may be invoked at irregular intervals. Sporadic tasks are characterized by an upper bound on the rate at which they may be invoked. This is commonly specified by requiring that successive invocations of a sporadic task T_i be separated in time by at least $t(i)$ seconds. Sporadic tasks are sometimes also called aperiodic. However, some people define *aperiodic* tasks to be those tasks which are not periodic and which also have no upper bound on their invocation rate.

Example 3.2. Consider a pressure vessel whose pressure is measured every 10 ms. If the pressure exceeds a critical value, an alarm is sounded. The task that measures the pressure is periodic, with a period of 10 ms. The task that turns on the alarm is sporadic. What is the maximum rate at which this task can be invoked?

A task assignment[1]/schedule is said to be *feasible* if all the tasks start after their release times and complete before their deadlines. We say that a set of tasks is A-feasible if an assignment/scheduling algorithm A, when run on that set of tasks, results in a feasible schedule. The bulk of the work in real-time scheduling deals with obtaining feasible schedules.

Given these task characteristics and the execution times and deadlines associated with the tasks, the tasks are allocated or assigned to the various processors and scheduled on them. The schedule can be formally defined as a function

$$S: \text{Set of processors} \times \text{Time} \rightarrow \text{Set of tasks} \tag{3.2}$$

$S(i, t)$ is the task scheduled to be running on processor i at time t. Most of the time, we will depict schedules graphically.

A schedule may be precomputed (offline scheduling), or obtained dynamically (online scheduling). *Offline scheduling* involves scheduling in advance of the operation, with specifications of when the periodic tasks will be run and slots for the sporadic/aperiodic tasks in the event that they are invoked. In *online scheduling*, the tasks are scheduled as they arrive in the system. The algorithms used in online scheduling must be fast; an online scheduling algorithm that takes so long that it leaves insufficient time for tasks to meet their deadlines is clearly useless.

The relative priorities of tasks are a function of the nature of the tasks themselves and the current state of the controlled process. For example, a task that controls stability in an aircraft and that has to be run at a high frequency

[1] In this book, we will use the terms "task allocate" and "task assign" interchangeably.

can reasonably be assigned a higher priority than one that controls cabin pressure. Also, a mission can consist of different phases or modes and the priority of the same task can vary from one phase to another.

Example 3.3. The flight of an aircraft can be broken down into phases such as takeoff, climb, cruise, descend, and land. In each phase, the task mix, task priorities, and task deadlines may be different.

There are algorithms that assume that the task priority does not change within a mode; these are called *static-priority* algorithms. By contrast, *dynamic-priority* algorithms assume that priority can change with time. The best known examples of static- and dynamic-priority algorithms are the rate-monotonic (RM) algorithm and the earliest deadline first (EDF) algorithm, respectively. We shall discuss each in considerable detail.

The schedule may be preemptive or nonpreemptive. A schedule is *preemptive* if tasks can be interrupted by other tasks (and then resumed). By contrast, once a task is begun in a *nonpreemptive* schedule, it must be run to completion or until it gets blocked over a resource. We shall mostly be concerned with preemptive schedules in this book—wherever possible, critical tasks must be allowed to interrupt less critical ones when it is necessary to meet deadlines.

Preemption allows us the flexibility of not committing the processor to run a task through to completion once we start executing it. Committing the processor in a nonpreemptive schedule can cause anomalies.

Example 3.4. Consider a two-task system. Let the release times of tasks T_1 and T_2 be 1 and 2, respectively; the deadlines be 9 and 6; and the execution times be 3.25 and 2. Schedule S_1 in Figure 3.2 meets both deadlines. However, suppose we follow the perfectly sensible policy of not keeping the processor idle whenever there is a task waiting to be run; then, we will have schedule S_2.

This results in task T_2 missing its deadline!

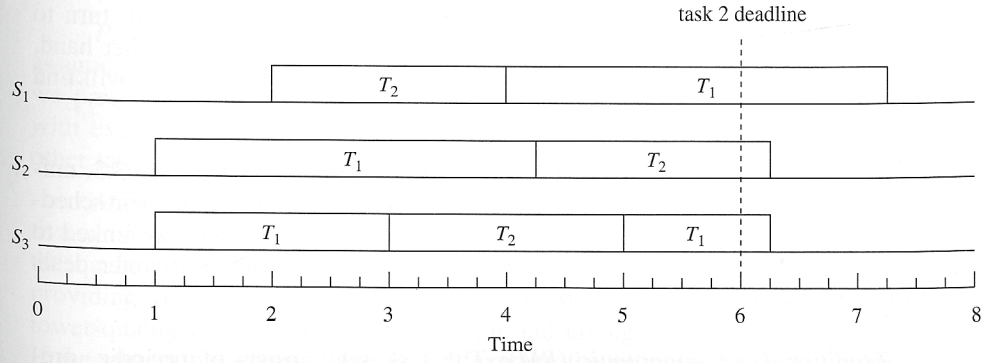

FIGURE 3.2
Preemptive and nonpreemptive schedules.

Notice that for schedule S_1 to have been implemented in the first place, we must have had some notion when T_1 arrived at time 1 that task T_2 would be on its way, and that its deadline would be too tight to allow task T_1 to complete ahead of it.

By contrast, S_3 is a preemptive schedule. When T_1 is released, it starts executing. When T_2 arrives, T_1 is preempted and run to completion, thus meeting its deadline. Then T_1 resumes from where it left off and also meets its deadline.

There is, however, a penalty associated with preemption. In order to allow a task to resume correctly, some housekeeping chores must be done by the system. The register values must be saved when a task is preempted and then restored when it resumes. Preemption is not always possible. For example, consider a disk system in the middle of writing a sector. It cannot simply abort its current operation—it must carry through to completion or the affected sector will not be consistent.

The vast majority of assignment/scheduling problems on systems with more than two processors are NP-complete. We must therefore make do with heuristics. Most heuristics are motivated by the fact that the uniprocessor scheduling (i.e., scheduling a set of tasks on a single processor) problem is usually tractable. The task of developing a multiprocessor schedule is therefore divided into two steps: first we assign tasks to processors, and second, we run a uniprocessor scheduling algorithm to schedule the tasks allocated to each processor. If one or more of the schedules turn out to be infeasible, then we must either return to the allocation step and change the allocation, or declare that a schedule cannot be found and stop. One example of this process is summarized in Figure 3.3. Many variations of this approach are possible; for example, one can check for schedulability after the allocation of each task.

3.1.1 How to Read This Chapter

This chapter is long and potentially intimidating. Many of the algorithms we cover have fairly involved proofs. For those of you interested only in checking if this chapter contains an algorithm that meets your immediate needs, and not interested in all the associated mathematical trappings, we summarize below the key features of the algorithms covered here. Simply go through the list, turn to the detailed algorithm description, and skip all the proofs. If, on the other hand, you are interested in the proof techniques used in scheduling theory, you will find many good examples here.

UNIPROCESSOR SCHEDULING ALGORITHMS.* As shown in Figure 3.3, uniprocessor scheduling is part of the process of developing a multiprocessor schedule. Our ability to obtain a feasible multiprocessor schedule is therefore linked to our ability to obtain feasible uniprocessor schedules. Most of this chapter deals with this problem.

Traditional rate-monotonic (RM): The task set consists of periodic, preemptible tasks whose deadlines equal the task period. A task set of n tasks is schedulable under RM if its total processor utilization is no greater than $n(2^{1/n}-1)$.

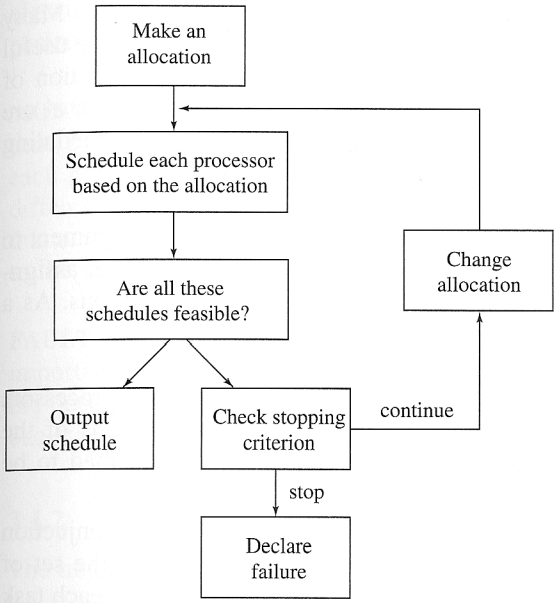

FIGURE 3.3
Developing a multiprocessor schedule.

Task priorities are static and inversely related to their periods. RM is an optimal static-priority uniprocessor scheduling algorithm and is very popular. Some results are also available for the case where a task deadline does not equal its period. See Section 3.2.1.

Rate-monotonic deferred server (DS): This is similar to the RM algorithm, except that it can handle both periodic (with deadlines equal to their periods) and aperiodic tasks. See Section 3.2.1 (in Sporadic Tasks).

Earliest deadline first (EDF): Tasks are preemptible and the task with the earliest deadline has the highest priority. EDF is an optimal uniprocessor algorithm. If a task set is not schedulable on a single processor by EDF, there is no other processor that can successfully schedule that task set. See Section 3.2.2.

Precedence and exclusion conditions: Both the RM and EDF algorithms assume that the tasks are independent and preemptible anytime. In Section 3.2.3, we present algorithms that take precedence conditions into account. Algorithms with exclusion conditions (i.e., certain tasks are not allowed to interrupt certain other tasks, irrespective of priority) are also presented.

Multiple task versions: In some cases, the system has primary and alternative versions of some tasks. These versions vary in their execution time and in the quality of output they provide. Primary versions are the full-fledged tasks, providing top-quality output. Alternative versions are bare-bones tasks, providing lower-quality (but still acceptable) output and taking much less time to execute. If the system has enough time, it will execute the primary; however, under conditions of overload, the alternative may be picked. In Section 3.2.4, an algorithm is provided to do this.

IRIS tasks: IRIS stands for increased reward with increased service. Many algorithms have the property that they can be stopped early and still provide useful output. The quality of the output is a monotonically nondecreasing function of the execution time. Iterative algorithms (e.g., algorithms that compute π or e) are one example of this. In Section 3.3, we provide algorithms suitable for scheduling such tasks.

MULTIPROCESSOR SCHEDULING. Algorithms dealing with task assignment to the processors of a multiprocessor are discussed in Section 3.4. The task assignment problem is NP-hard under any but the most simplifying assumptions. As a result, we must make do with heuristics.

Utilization balancing algorithm: This algorithm assigns tasks to processors one by one in such a way that at the end of each step, the utilizations of the various processors are as nearly balanced as possible. Tasks are assumed to be preemptible.

Next-fit algorithm: The next-fit algorithm is designed to work in conjuction with the rate-monotonic uniprocessor scheduling algorithm. It divides the set of tasks into various classes. A set of processors is exclusively assigned to each task class. Tasks are assumed to be preemptible.

Bin-packing algorithm: The bin-packing algorithm assigns tasks to processors under the constraint that the total processor utilization must not exceed a given threshold. The threshold is set in such a way that the uniprocessor scheduling algorithm is able to schedule the tasks assigned to each processor. Tasks are assumed to be preemptible.

Myopic offline scheduling algorithm: This algorithm can deal with nonpreemptible tasks. It builds up the schedule using a search process.

Focused addressing and bidding algorithm: In this algorithm, tasks are assumed to arrive at the individual processors. A processor that finds itself unable to meet the deadline or other constraints of all its tasks tries to offload some of its workload onto other processors. It does so by announcing which task(s) it would like to offload and waiting for the other processors to offer to take them up.

Buddy strategy: The buddy strategy takes roughly the same approach as the focused addressing algorithm. Processors are divided into three categories: underloaded, fully loaded, and overloaded. Overloaded processors ask the underloaded processors to offer to take over some of their load.

Assignment with precedence constraints: The last task assignment algorithm takes task precedence constraints into account. It does so using a trial-and-error process that tries to assign tasks that communicate heavily with one another to the same processor so that communication costs are minimized.

CRITICAL SECTIONS. Certain anomalous behavior can be exhibited as a result of critical sections. In particular, a lower-priority task can make a higher-priority task wait for it to finish, even if the two are not competing for access to the same critical section. In Section 3.2.1 (in Handling Critical Sections), we present

algorithms to get around this problem and to provide a finite upper bound to the period during which a lower-priority task can block a higher-priority task.

MODE CHANGES. Frequently, task sets change during the operation of a real-time system. We have seen in Chapter 2 that a mission can have multiple phases, each phase characterized by a different set of tasks, or the same task set but with different priorities or arrival rates. In Section 3.5, we discuss the scheduling issues that arise when a mission phase changes. We look at how to delete or add tasks to the task list.

FAULT-TOLERANT SCHEDULNG. The final part of this chapter deals with the important problem of ensuring that deadlines will continue to be met despite the occurrence of faults. In Section 3.6, we describe an algorithm that schedules backups that are activated in the event of failure.

3.1.2 Notation

The notation used in this chapter will be as follows.

n	Number of tasks in the task set.
e_i	Execution time of task T_i.
P_i	Period of task T_i, if it is periodic.
I_i	kth period of (periodic) task T_i begins at time $I_i + (k-1)P_i$, where I_i is called the *phasing* of task T_i.
d_i	Relative deadline of task T_i (relative to release time).
D_i	Absolute deadline of task T_i
r_i	Release time of task T_i
$h_T(t)$	Sum of the execution times of task iterations in task set T that have their absolute deadlines no later than t.

Additional notation will be introduced as appropriate.

3.2 CLASSICAL UNIPROCESSOR SCHEDULING ALGORITHMS

In this section, we will consider two venerable algorithms used for scheduling independent tasks on a single processor, rate-monotonic (RM) and earliest deadline first (EDF). The goal of these algorithms is to meet all task deadlines. Following that, we will deal with precedence and exclusion constraints, and consider situations where multiple versions of software are available for the same task.

The following assumptions are made for both the RM and EDF algorithms.

A1. No task has any nonpreemptable section and the cost of preemption is negligible.

A2. Only processing requirements are significant; memory, I/O, and other resource requirements are negligible.

A3. All tasks are independent; there are no precedence constraints.

These assumptions greatly simplify the analyses of RM and EDF. Assumption **A1** indicates that we can preempt any task at any time and resume it later without penalty. As a result, the number of times that a task is preempted does not change the total workload of the processor. From **A2**, to check for feasibility we only have to ensure that enough processing capacity exists to execute the tasks by their deadlines; there are no memory or other constraints to complicate matters. The absence of precedence constraints, **A3**, means that task release times do not depend on the finishing times of other tasks.

Of course, there are also many systems for which assumptions **A1** to **A3** are not good approximations. Later in this chapter, we will see how to deal with some of these.

3.2.1 Rate-Monotonic Scheduling Algorithm

The rate-monotonic (RM) scheduling algorithm is one of the most widely studied and used in practice. It is a uniprocessor static-priority preemptive scheme. Except where it is otherwise stated, the following assumptions are made in addition to assumptions **A1** to **A3**.

A4. All tasks in the task set are periodic.

A5. The relative deadline of a task is equal to its period.

Assumption **A5** simplifies our analysis of RM greatly, since it ensures that there can be at most one iteration of any task alive at any time.

The priority of a task is inversely related to its period; if task T_i has a smaller period than task T_j, T_i has higher priority than T_j. Higher-priority tasks can preempt lower-priority tasks.

Example 3.5. Figure 3.4 contains an example of this algorithm. There are three tasks, with $P_1 = 2$, $P_2 = 6$, $P_3 = 10$. The execution times are $e_1 = 0.5$, $e_2 = 2.0$, $e_3 = 1.75$, and $I_1 = 0$, $I_2 = 1$, $I_3 = 3$. Since $P_1 < P_2 < P_3$, task T_1 has highest priority. Every time it is released, it preempts whatever is running on the processor. Similarly, task T_3 cannot execute when either task T_1 or T_2 is unfinished.

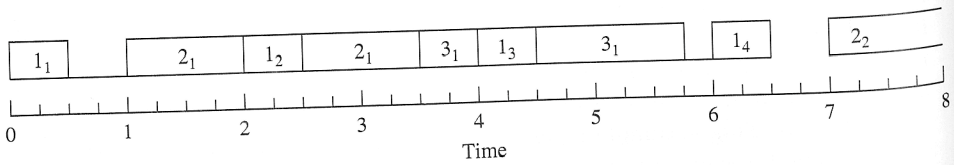

FIGURE 3.4
Example of the RM algorithm; K_j denotes the jth release (or iteration) of Task T_K.

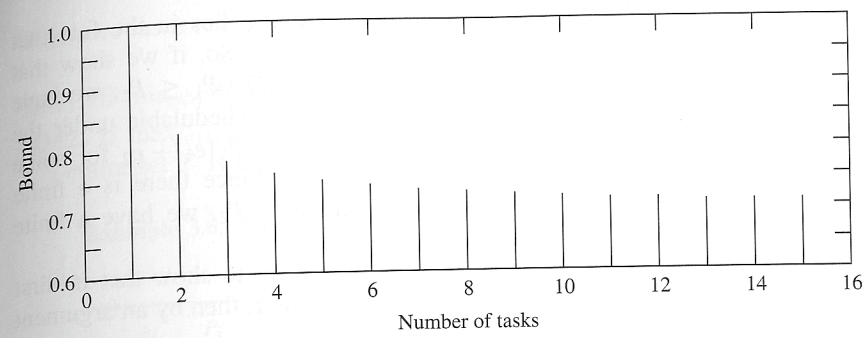

FIGURE 3.5
Utilization bound for the RM algorithm.

There is an easy schedulability test for this algorithm, as follows:

If the total utilization of the tasks is no greater than $n(2^{1/n} - 1)$, where n is the number of tasks to be scheduled, then the RM algorithm will schedule all the tasks to meet their respective deadlines. Note that this is a sufficient, but not necessary, condition. That is, there may be task sets with a utilization greater than $n(2^{1/n} - 1)$ that are schedulable by the RM algorithm.

The $n(2^{1/n} - 1)$ bound is plotted in Figure 3.5.

Let us now turn to determining the necessary and sufficient conditions for RM-schedulability. To gain some intuition into what these conditions are, let us determine them from first principles for the three-task example in Example 3.5.

Assume that the task phasings are all zero (i.e., the first iteration of each task is released at time zero). Observe the first iteration of each task. Let us start with task T_1. This is the highest-priority task, so it will never be delayed by any other task in the system. The moment T_1 is released, the processor will interrupt anything else it is doing and start processing it. As a result, the only condition that must be satisfied to ensure that T_1 can be feasibly scheduled is that $e_1 \leq P_1$. This is clearly a necessary, as well as a sufficient, condition.

Now, turn to task T_2. It will be executed successfully if its first iteration can find enough time over $[0, P_2]$. Suppose T_2 finishes at time t. The total number of iterations of task T_1 that have been released over $[0, t]$ is $\lceil t/P_1 \rceil$. In order for T_2 to finish at t, every one of the iterations of task T_1 released in $[0, t]$ must be completed, and in addition there must be e_2 time available to execute T_2. That is, we must satisfy the condition:

$$t = \left\lceil \frac{t}{P_1} \right\rceil e_1 + e_2$$

If we can find some $t \in [0, P_2]$ satisfying this condition, we are done.

Now comes the practical question of how we check that such a t exists. After all, every interval has an infinite number of points in it, so we can't very

well check exhaustively for every possible t. The solution lies in the fact that $\lceil t/P_1 \rceil$ only changes at multiples of P_1, with jumps of e_1. So, if we show that there exists some integer k such that $kP_1 \geq ke_1 + e_2$ and $kP_1 \leq P_2$, we have met the necessary and sufficient conditions for T_2 to be schedulable under the RM algorithm. That is, we only need to check if $t \geq \lceil t/P_1 \rceil e_1 + e_2$ for some value of t that is a multiple of P_1, such that $t \leq P_2$. Since there is a finite number of multiples of P_1 that are less than or equal to P_2, we have a finite check.

Finally, consider task T_3. Once again, it is sufficient to show that its first iteration completes before P_3. If T_3 completes executing at t, then by an argument identical to that for T_2, we must have:

$$t = \left\lceil \frac{t}{P_1} \right\rceil e_1 + \lceil \frac{t}{P_2} \rceil e_2 + e_3$$

T_3 is schedulable iff there is some $t \in [0, P_3]$ such that the above condition is satisfied. But, the right-hand side (RHS) of the above equation has jumps only at multiples of P_1 and P_2. It is therefore sufficient to check if the inequality

$$t \geq \left\lceil \frac{t}{P_1} \right\rceil e_1 + \left\lceil \frac{t}{P_2} \right\rceil e_2 + e_3$$

is satisfied for some t that is a multiple of P_1 and/or P_2, such that $t \leq P_3$.

We are now ready to present the necessary and sufficient condition in general. We will need the following additional notation:

$$W_i(t) = \sum_{j=1}^{i} e_j \lceil \frac{t}{P_j} \rceil$$

$$L_i(t) = \frac{W_i(t)}{t}$$

$$L_i = \min_{0<t\leq P_i} L_i(t)$$

$$L = \max\{L_i\}$$

$W_i(t)$ is the total amount of work carried by tasks $T_1, T_2, \ldots, T_i$, initiated in the interval $[0, t]$. If all tasks are released at time 0, then task T_i will complete under the RM algorithm at time t', such that $W_i(t') = t'$ (if such a t' exists).

The necessary and sufficient condition for schedulability is as follows.

> Given a set of n periodic tasks (with $P_1 \leq P_2 < \ldots \leq P_n$). Task T_i can be feasibly scheduled using RM iff $L_i \leq 1$.

As in our previous example, the practical question of how to check for $W_i(t) \leq t$ is easily answered by examining the defining equation $W_i(t) = \sum_{j=1}^{i} e_j \lceil t/P_j \rceil$. We see that $W_i(t)$ is constant, except at a finite number of points when tasks are released. We only need to compute $W_i(t)$ at the times

$$\tau_i = \{\ell P_j | j = 1, \cdots, i; \ell = 1, \cdots, \lfloor P_i/P_j \rfloor\} \tag{3.3}$$

Then, we have two RM-scheduling conditions:

RM1. If $\min_{t\in\tau_i} W_i(t) \leq$, task T_i is RM-schedulable.

RM2. If $\max_{i\in\{1,\ldots,n\}} \{\min_{t\in\tau_i} W_i(t)/t\} \leq 1$ for $i \in \{1, \ldots, n\}, t \in T_i$, then the entire set T is RM-schedulable.

Example 3.6. Consider the set of four tasks where

i	e_i	P_i	i	e_i	P_i
1	20	100	3	80	210
2	30	150	4	100	400

Then,

$$\tau_1 = \{100\}$$

$$\tau_2 = \{100, 150\}$$

$$\tau_3 = \{100, 150, 200, 210\}$$

$$\tau_4 = \{100, 150, 200, 210, 300, 400\}$$

Let us check the RM-schedulability of each task. Figure 3.6 contains plots of $W_i(t)$ for $i = 1, 2, 3, 4$. Task T_i is RM-schedulable iff any part of the plot of $W_i(t)$ falls on or below the $W_i(t) = t$ line.

In algebraic terms, we have:

- task T_1 is RM-schedulable iff $e_1 \leq 100$
- task T_2 is RM-schedulable iff

$$\begin{array}{ll} e_1 + e_2 \leq 100 & \text{OR} \\ 2e_1 + e_2 \leq 150 & \end{array} \tag{3.4}$$

- task T_3 is RM-schedulable iff

$$\begin{array}{ll} e_1 + e_2 + e_3 \leq 100 & \text{OR} \\ 2e_1 + e_2 + e_3 \leq 150 & \text{OR} \\ 2e_1 + 2e_2 + e_3 \leq 200 & \text{OR} \\ 3e_1 + 2e_2 + e_3 \leq 210 & \end{array} \tag{3.5}$$

- task T_4 is RM-schedulable iff

$$\begin{array}{ll} e_1 + e_2 + e_3 + e_4 \leq 100 & \text{OR} \\ 2e_1 + e_2 + e_3 + e_4 \leq 150 & \text{OR} \\ 2e_1 + 2e_2 + e_3 + e_4 \leq 200 & \text{OR} \\ 3e_1 + 2e_2 + e_3 + e_4 \leq 210 & \text{OR} \\ 3e_1 + 2e_2 + 2e_3 + e_4 \leq 300 & \text{OR} \\ 4e_1 + 3e_2 + 2e_3 + e_4 \leq 400 & \end{array} \tag{3.6}$$

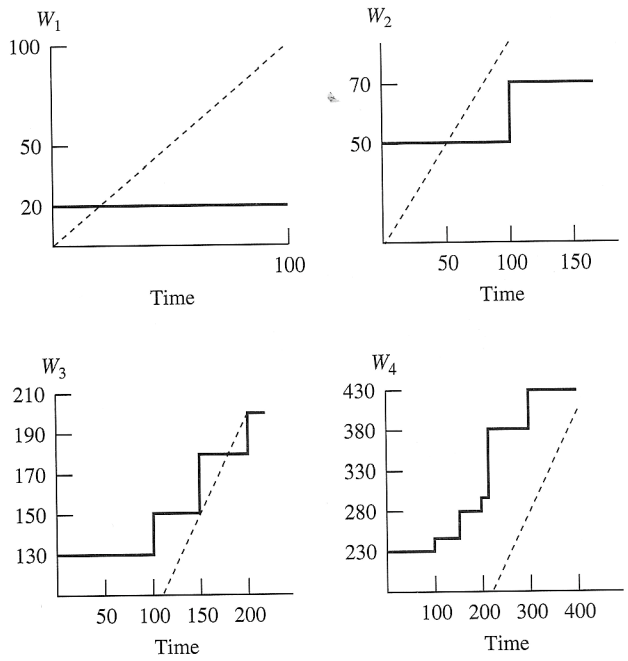

FIGURE 3.6
$W_i(t)$ for Example 3.6; the dotted line indicates the locus of $W_i(t) = t$.

From Figure 3.6 and the above equations, we can see that tasks T_1, T_2, and T_3 are RM-schedulable, but task T_4 is not.

SPORADIC TASKS. Thus far, we have only considered periodic tasks. Let us now introduce sporadic tasks. These are released irregularly, often in response to some event in the operating environment. While sporadic tasks do not have periods associated with them, there must be some maximum rate at which they can be released. That is, we must have some minimum interarrival time between the release of successive iterations of sporadic tasks. Otherwise, there is no limit to the amount of workload that sporadic tasks can add to the system and it will be impossible to guarantee that deadlines are met.

One way of dealing with sporadic tasks is to simply consider them as periodic tasks with a period equal to their minimum interarrival time. Two other approaches are outlined below.

Perhaps the simplest way to incorporate sporadic tasks is to define a fictitious periodic task of highest priority and of some chosen fictitious execution period. During the time that this task is scheduled to run on the processor, the processor is available to run any sporadic tasks that may be awaiting service. Outside this time, the processor attends to the periodic tasks.

Example 3.7. Figure 3.7 provides an illustration. We have here a fictitious highest-priority task with period of 10 and execution time of 2.5. This task occupies the

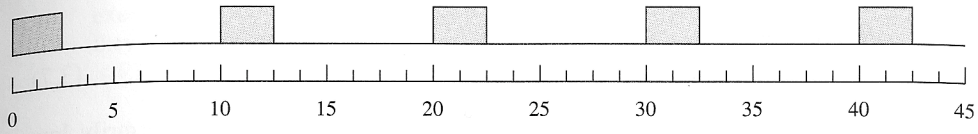

FIGURE 3.7
Incorporating sporadic tasks: method 1.

processor during the time shown by the shaded portion, which has been set aside to execute any pending sporadic tasks—every 10 time units, the processor can execute up to 2.5 units of sporadic tasks. If, during that time, there is no sporadic task awaiting service, the processor is idle. The processor cannot execute sporadic tasks outside the shaded intervals.

The deferred server (DS) approach is less wasteful. Whenever the processor is scheduled to run sporadic tasks and finds no such tasks awaiting service, it starts executing the other (periodic) tasks in order of priority. However, if a sporadic task arrives, it preempts the periodic task and can occupy a total time up to the time allotted for sporadic tasks.

Example 3.8. In Figure 3.8, the occupancy of the processor by sporadic tasks is indicated by shaded rectangles. 2.5 time units are allocated every 10-unit period for sporadic tasks. A sporadic task requiring 5 units arrives at time 5 and takes over the processor. At time 7.5, the processor has given the task its entire quota of 2.5 units over the current period, and so the sporadic task is preempted by some other task. At time 10, the next sporadic-task period begins and the remaining 2.5 units of service are delivered. The next sporadic task, with a total execution time requirement of 7.5 units arrives at time 27.5. It has available to it the 2.5 units from the current period of [20, 30] plus the 2.5 units from the next period of [30, 40]. It therefore occupies the processor over the interval [27.5, 32.5]. At that point, its quota of time on the processor (for the [30, 40] period) is exhausted and it relinquishes the processor. At time 40, a new sporadic-task period begins and the sporadic task receives its last 2.5 units of service, completing at 42.5.

Schedulability criteria can be derived for the DS algorithm in much the same way as for the basic RM algorithm. When the relative deadlines of all tasks equal their periods, and U_s is the processor utilization allocated to the sporadic tasks, we can show that it is possible to schedule periodic tasks if the total task utilization

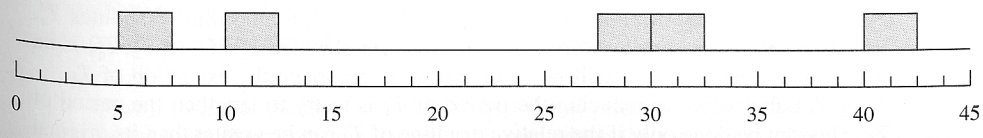

FIGURE 3.8
Incorporating sporadic tasks: method 2 (deferred server).

(including the sporadic contribution) U satisfies the following bound:

$$U \leq \begin{cases} 1 - U_s & \text{if } U_s \leq 0.5 \\ U_s & \text{if } U_s \geq 0.5 \end{cases} \tag{3.7}$$

When $U_s \geq 0.5$, it is possible to construct a periodic task set of arbitrarily low (but positive) utilization that cannot be feasibly scheduled.

Example 3.9. Suppose $P_s = 6$ is the period of the deferred server and $P_1 = 6$ for periodic task T_1. Let the execution time reserved for the sporadic task be $e_s = 3$, that is, $U_s = 3/6 = 0.5$. Then, if the sporadic tasks occupy back-to-back time-slices of 3 each, an entire period of P_1 will pass with no time available for T_1.

Equation 3.7 is a sufficient, though not necessary, condition for schedulability: It is easy to construct feasible periodic task sets even when $U_s \geq 0.5$.

TRANSIENT OVERLOADS. One drawback of the RM algorithm is that task priorities are defined by their periods. Sometimes, we must change the task priorities to ensure that all critical tasks get completed. We motivate this using the following example.

Example 3.10. Suppose that we are given, in addition to the task periods P_i and worst-case execution times e_i for task T_i, average execution times a_i. Consider the four-task set with the following characteristics that we considered in Example 3.6.

i	e_i	a_i	P_i
1	20	10	100
2	30	25	150
3	80	40	210
4	100	20	400

Suppose that tasks T_1, T_2, and T_4 are critical, and that T_3 is noncritical. It is easy to check that if we run the RM algorithm on this task set, we cannot guarantee the schedulability of all four tasks if they each take their worst-case execution times. However, in the average case, they will all be RM-schedulable. The problem is how to arrange matters so that all the critical tasks meet their deadlines under the RM algorithm even in the worst case, while T_3 meets its deadline in many other cases.

The solution is to boost the priority of T_4 by altering its period. We will replace T_4 by task T_4' with the following parameters: $P_4' = P_4/2$, $e_4' = e_4/2$, $a_4' = a_4/2$. It is easy to check that tasks T_1, T_2, and T_4' are RM-schedulable even in the worst case. T_3 now has a lower priority than T_4'. Whenever the algorithm schedules T_4', we can run the code for T_4. Because of the way we obtained e_4', if $\{T_1, T_2, T_4'\}$ is an RM-schedulable set, there will be enough time to complete the execution of T_4.

An alternative to reducing the period of T_4 is to try to lengthen the period of T_3. This can be done only if the relative deadline of T_3 can be greater than its original period. In this case, we can replace T_3 by two tasks T_3' and T_3'', each with period 420 (i.e., 210 × 2), with worst-case execution times $e_3' = e_3'' = 80$ and average-case execution times $a_3' = a_3'' = 40$. The scheduler will have to phase T_3' and T_3'' so that they are released $P_3 = 210$ units apart. If the resultant task set $\{T_1, T_2, T_3', T_3'', T_4\}$ is RM-schedulable, we are done.

In general, if we lengthen the period by a factor of k, we will replace the original task by k tasks, each phased by the appropriate amount. If we reduce the period by a factor of k, we will replace the original task by one whose execution time is also reduced by a factor of k.

This procedure of period transformation ultimately results in two sets of tasks, C and NC, with the following properties:

- C contains all the critical tasks and possibly some noncritical tasks.
- NC contains only noncritical tasks.
- $P_{c,\max} \leq P_{n,\min}$, where $P_{c,\max}$ and $P_{n,\min}$ are the maximum and minimum periods of tasks in C and NC, respectively.
- C is RM-schedulable under worst-case task execution times.

The procedure is to first set C to be the set of critical tasks and NC the set of non-critical tasks. If $P_{c,\max} \leq P_{n,\min}$, we are done. If this is not the case, then we move those noncritical tasks whose periods are less than or equal to $P_{c,\max}$ into the set C. If the new set C is RM-schedulable under worst-case task execution times, we are done. If not, then we try to lengthen the periods of the noncritical tasks in C by as much as possible until C is RM-schedulable. If this is not possible, then we reduce the periods of the higher-priority critical tasks and move back into NC all noncritical tasks in C whose periods are larger than the largest period of any critical task in C. We continue this process until we arrive at C and NC with the above properties.

MATHEMATICAL UNDERPINNINGS.* Let us develop the properties of the RM algorithm. In particular, we are interested in what processor utilizations are possible and how to determine whether a given task set is feasible.

We begin with two definitions. Let $R_i(x)$ be the response time of task T_i if it is released at time x. x^* is said to be a *critical time instant* for task T_i if $R_i(x^*) \geq R_i(x)$ $\forall x$. In other words, x^* is the worst time at which to release T_i, with respect to its response time. A *critical time zone* of task T_i is defined as the interval $[x^*, x^* + R_i(x^*)]$; that is, it is the interval between a critical time instant and when the task (initiated at that time) finishes.

Theorem 3.1. A critical time instant of any task occurs when that task is requested at the same time as all other higher-priority tasks.

Proof. Number the tasks in descending order of priority. The response time of a task is the execution time plus the interval over which the task was waiting to execute.

Let us begin with the case where there are only two tasks, T_1 and T_2. Define the time axis so that $I_2 = 0$. Task T_1 occupies the time intervals $[I_1, I_1 + e_1]$, $[I_1 +$

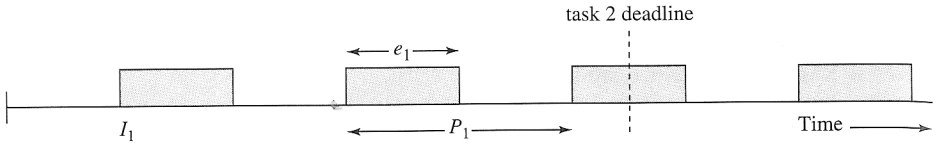

FIGURE 3.9
Theorem 3.1; the shaded portions indicate that the processor is occupied by task T_1.

$P_1, I_1 + P_1 + e_1], \ldots, [I_1 + nP_1, I_1 + nP_1 + e_1], \ldots$. The response time of task T_2 is given by

$$e_2 + n_{p2}(1)e_1$$

where $n_{pi}(j)$ is the number of times task T_i is preempted by task T_j. Task T_2 executes for I_1 before the first iteration of task T_1 arrives. After that, it executes for at most $P_1 - e_1$ between successive iterations of task T_1. Hence,

$$(n_{p2}(1) - 1)(P_1 - e_1) + I_1 < e_2 \leq n_{p2}(1)(P_1 - e_1) + I_1 \tag{3.8}$$

To maximize the response time of task T_2, we must maximize $n_{p2}(1)$, subject to the constraint in (3.8). The only variable is I_1; all the other parameters (P_1, e_1, e_2) are constants. From (3.8), it follows immediately that $n_{p2}(1)$ is maximized when $I_1 = 0$.

This is probably easier to see geometrically as in Figure 3.9. Task T_2 can only execute in the unshaded portions and will complete whenever these intervals have a total duration of e_2. Altering I_1 is tantamount to moving the train of shaded portions. From the diagram, it is apparent that the response time of T_2 is maximized when T_1 has zero phasing, i.e., $I_1 = 0$.

A similar argument holds for an arbitrary number of tasks. **Q.E.D.**

A given set of tasks is said to be *RM-schedulable* if the RM algorithm produces a schedule that meets all the deadlines. It follows from Theorem 3.1 that a set of tasks is RM-schedulable for any values of $I_1, I_2, \ldots$, if it is RM-schedulable for $I_1 = I_2 = \cdots = 0$. Therefore, to check for RM-schedulability, we only need to check for the case where all task phasings are zero. Theorem 3.2 is what has made RM so popular.

Theorem 3.2. The RM algorithm is an optimal static-priority algorithm. That is, if any static-priority algorithm can produce a feasible schedule, so can RM.

Proof. We proceed by contradiction. Suppose there exists some task set and some other static-priority algorithm A such that A generates a feasible schedule, but RM does not.

Since A is a static-priority algorithm, it proceeds by assigning priorities to tasks and then scheduling on the basis of these priorities. Since A is different from RM and optimal (while RM, under the hypothesis, is not), there must be some task set T for which

- algorithm A allocates task priorities differently from algorithm RM, and
- algorithm A successfully schedules the tasks in T, while under RM one or more deadlines are missed.

In particular, there will be tasks T_i, T_j in the set T with the following properties:

1. Priority(T_i) = Priority(T_j) + 1 under algorithm A's priority assignment.
2. $P_i > P_j$, so that under RM T_i has a lower priority than T_j.

Denote by S_A the schedule that is produced by A. Now, consider the schedule, S', obtained by interchanging the priorities (as defined by algorithm A) of T_i and T_j, and keeping all the other priority assignments the same as in algorithm A. If all the task deadlines are met under S_A, all task deadlines will continue to be met under S'. (Can you work out why this should be the case?)

If this interchange leads to the same priority assignment as the RM algorithm, then this shows that T is RM-schedulable, thus contradicting our original hypothesis. If it is still different from the RM priority assignment, we can continue interchanging priorities as before and obtain a feasible schedule each time. This process stops when we get the same priority assignment as for RM and the proof is complete. **Q.E.D.**

Let us now generate some conditions on the processor utilization possible under this algorithm. If there are n tasks, the processor utilization is given by

$$U = \sum_{i=1}^{n} \frac{e_i}{P_i} \tag{3.9}$$

A set of tasks is said to *fully utilize* the processor if

- the RM-schedule meets all deadlines, and
- the task set is no longer RM-schedulable if the execution time of any task is increased.

It is important to keep in mind that the utilization of a processor that is fully utilized, under the above definition, is not necessarily 1 over the entire interval $[0, \infty)$.

Example 3.11. A two-task set with $P_1 = 5$, $P_2 = 7$, $e_1 = 2$, $e_2 = 3$, and $I_1 = I_2 = 0$ fully utilizes the processor; see Figure 3.10. If either e_1 or e_2 is increased, a deadline is missed. The processor utilization is less than 1: It is $2/5 + 3/7 = 0.83$.

We will show that if the task set is such that the processor utilization is no greater than $n(2^{1/n} - 1)$, the task set is RM-schedulable. This will provide us with a simple check on RM-schedulability. Our proof will proceed by showing that the least upper bound of utilization for schedulability is $n(2^{1/n} - 1)$. Hence, if the utilization of any set of tasks is no greater than $n(2^{1/n} - 1)$, we will know

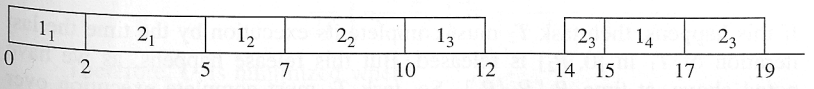

FIGURE 3.10
A task set that fully utilizes the processor.

that it is RM-schedulable. In what follows, assume the tasks are numbered so that $P_1 \leq P_2 \leq \ldots \leq P_n$. Let us begin with a task set consisting of only two tasks (i.e., $n = 2$).

Lemma 3.1. If there are two tasks, T_1, T_2, and

$$\frac{e_1}{P_1} + \frac{e_2}{P_2} \leq 2(\sqrt{2} - 1)$$

then the tasks are RM-schedulable.

Proof. We have assumed that $P_2 \geq P_1$, so that task T_2 has a lower priority than task T_1. During one period of task T_2, there are $\lceil P_2/P_1 \rceil$ releases of task T_1. Let us determine the maximum value of e_2 (as a function of e_1) so that the task set remains RM-schedulable. We have two cases.

Case 1. The processor is not executing task T_1 at time P_2.
Since task T_1 has preemptive priority over task T_2, Case 1 will only happen if every iteration of T_1 that was released in the interval $[0, P_2]$ has been completed by P_2. The last iteration of T_1 within $[0, P_2]$ is released at time $P_1 \lfloor P_2/P_1 \rfloor$. Hence, we must have

$$P_1 \left\lfloor \frac{P_2}{P_1} \right\rfloor + e_1 \leq P_2 \tag{3.10}$$

A total of $\lceil P_2/P_1 \rceil$ executions of task T_1 are completed during one period of T_2, and each consumes e_1 time. Since task T_2 must finish within its period, the maximum possible value of e_2 is

$$e_2 = P_2 - e_1 \left\lceil \frac{P_2}{P_1} \right\rceil \tag{3.11}$$

The processor utilization is then given by

$$\frac{e_1}{P_1} + \frac{e_2}{P_2} = \frac{e_1}{P_1} + 1 - \frac{e_1 \lceil P_2/P_1 \rceil}{P_2} \tag{3.12}$$

Since

$$\frac{1}{P_1} - \frac{\lceil P_2/P_1 \rceil}{P_2} \leq 0$$

the processor utilization is monotonically nonincreasing in e_1 whenever $P_1 \lfloor P_2/P_1 \rfloor + e_1 \leq P_2$.

Case 2. The processor is executing task T_1 at time P_2.
Case 2 occurs when

$$P_1 \left\lfloor \frac{P_2}{P_1} \right\rfloor + e_1 > P_2 \tag{3.13}$$

If this happens, then task T_2 must complete its execution by the time the last iteration of T_1 in $[0, P_2]$ is released. But this release happens, as we have noted above, at time $P_1 \lceil P_2/P_1 \rceil$. So, task T_2 must complete execution over the interval $[0, P_1 \lfloor P_2/P_1 \rfloor]$. Over this interval, however, T_1 occupies the processor for a total time of $\lfloor P_2/P_1 \rfloor e_1$. So, the total time available for T_2 is

$P_1 \lfloor P_2/P_1 \rfloor - \lfloor P_2/P_1 \rfloor e_1$. This means that the maximum possible value for e_2 is as given below:

$$e_2 = \left\lfloor \frac{P_2}{P_1} \right\rfloor \{P_1 - e_1\} \tag{3.14}$$

The corresponding processor utilization is

$$\begin{aligned} \frac{e_1}{P_1} + \frac{e_2}{P_2} &= \frac{e_1}{P_1} + \left\lfloor \frac{P_2}{P_1} \right\rfloor \left\{ \frac{P_1}{P_2} - \frac{e_1}{P_2} \right\} \\ &= \frac{P_1}{P_2} \left\lfloor \frac{P_2}{P_1} \right\rfloor + e_1 \left\{ \frac{1}{P_1} - \frac{\lfloor P_2/P_1 \rfloor}{P_2} \right\} \end{aligned} \tag{3.15}$$

Since

$$\frac{1}{P_1} - \frac{\lfloor P_2/P_1 \rfloor}{P_2} \geq 0$$

we see that when $P_1 \lfloor P_2/P_1 \rfloor + e_1 > P_2$, the processor utilization is monotonically nondecreasing in e_1.

From Equations (3.12) and (3.15), we find that the minimum value that processor utilization can have if the task set fully utilizes the processor occurs when

$$P_1 \left\lfloor \frac{P_2}{P_1} \right\rfloor + e_1 = P_2 \tag{3.16}$$

Denote the integral part of P_2/P_1 by I and its fractional part by f. By definition, $0 \leq f < 1$. Then, $\lfloor P_2/P_1 \rfloor = I$, and

$$\left\lceil \frac{P_2}{P_1} \right\rceil = \begin{cases} I & \text{if } f = 0 \\ I + 1 & \text{otherwise} \end{cases}$$

Using Equations (3.12), (3.16), and a little algebra, we can write an expression for the processor utilization when the processor is fully utilized by the task set:

$$\begin{aligned} U &= 1 + \frac{P_2 - P_1 \lfloor P_2/P_1 \rfloor}{P_1} - \frac{P_2 - P_1 \lfloor P_2/P_1 \rfloor}{P_2} \left\lceil \frac{P_2}{P_1} \right\rceil \\ &= 1 - \left\{ \left\lfloor \frac{P_2}{P_1} \right\rfloor + \left\lceil \frac{P_2}{P_1} \right\rceil - \frac{P_1}{P_2} \left\lfloor \frac{P_2}{P_1} \right\rfloor \left\lceil \frac{P_2}{P_1} \right\rceil - \frac{P_2}{P_1} \right\} \\ &= \begin{cases} 1 & \text{if } f = 0 \\ \dfrac{I + f^2}{I + f} & \text{otherwise} \end{cases} \end{aligned} \tag{3.17}$$

For $f > 0$, utilization is minimized when I is minimized. But $I \geq 1$ (since $P_2 \geq P_1$), so the minimal value of U is attained for $I = 1$. From elementary calculus, we have

$$\frac{d}{df} \left\{ \frac{I + f^2}{I + f} \right\} = \frac{2f}{1 + f} - \frac{1 + f^2}{(1 + f)^2} \tag{3.18}$$

Therefore, U is minimized when

$$\frac{2f}{1 + f} - \frac{1 + f^2}{(1 + f)^2} = 0 \Rightarrow f = \sqrt{2} - 1 \tag{3.19}$$