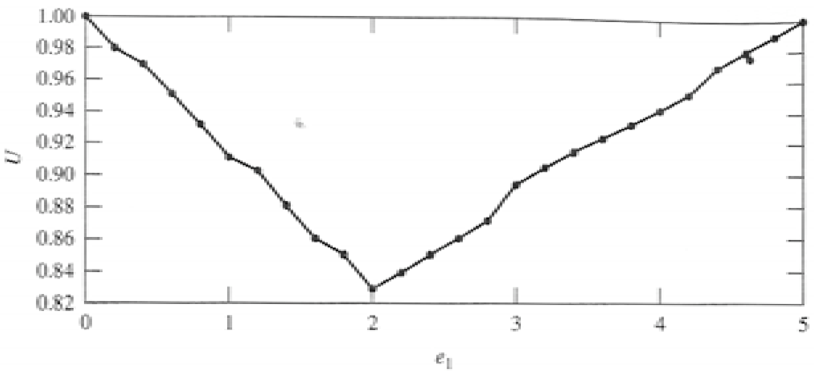

FIGURE 3.11
Processor utilization when the processor is fully utilized.

From this, we find the minimum value of U under full utilization to be $2(\sqrt{2}-1)$. This is the least upper bound. Q.E.D.

Example 3.12. Consider a two-task set with $P_1 = 5$, and $P_2 = 7$. In Figure 3.11, we plot the utilization as a function of e_1 when the processor is fully utilized (e_2 is chosen for full utilization).

Let us now extend this to more than two tasks. We do so in two steps. First we will consider the case $P_n < 2P_1$. In this case, the longest period, P_n, contains only two releases of each higher-priority task; this will greatly simplify our analysis. We will show that the least upper bound for schedulability in this case is given by $n(2^{1/n}-1)$. Then we will consider the $P_n \geq 2P_1$ case. For each set S of n tasks for which $P_n \geq 2P_1$, we will construct a set S' of n tasks for which $P'_n < 2P'_1$ (where P'_i is the period of the ith task in set S') with the property that if S fully utilizes the processor, so does S'. We will show that the utilization of the processor under S' is no greater than the utilization under S. As a result, the least upper bound of the utilization for the $P_n \geq 2P_1$ case cannot be less than that for the $P_n < 2P_1$ case. The overall least upper bound for schedulability is therefore $n(2^{1/n}-1)$.

Lemma 3.2. Given n tasks in the task set S with execution times e_i for task T_i, if $e_i = P_{i+1} - P_i$ for $i = 1, \ldots, n-1$, and $e_n = 2P_1 - P_n$, with $P_n < 2P_1$, then under the RM algorithm

- the task set fully utilizes the processor,
- there does not exist any other task set that also fully utilizes the processor and that has a lower processor utilization, and
- the processor utilization is at least $U = n(2^{(1/n)}-1)$.

Proof. As before, assume that the tasks are numbered so that $P_1 \leq P_2 \leq \ldots \leq P_n$. Let U denote the processor utilization under this task set. The task set fully utilizes the processor.

We will show that the utilization is minimized when $e_1 = P_2 - P_1$ by checking out the cases when $e_1 > P_2 - P_1$ and $e_1 < P_2 - P_1$. A similar argument yields the best values for the execution time of the other tasks.

Consider the case where $e_1 > P_2 - P_1$, that is,

$$e_1 = P_2 - P_1 + \Delta \qquad \Delta > 0 \tag{3.20}$$

Figure 3.12a illustrates this situation for the tasks T_1, T_2. The first release of task T_2 must complete before time P_1 (since the interval $[P_1, P_2]$ will be fully occupied by the second release of task T_1).

Now, define another task set S' with task execution times

$$\begin{aligned} e'_1 &= e_1 - \Delta \\ e'_2 &= e_2 + \Delta \\ e'_3 &= e_3 \\ &\vdots \\ e'_n &= e_n \end{aligned}$$

Task set S' will also fully utilize the processor. The additional slack created in the interval $[0, P_1]$ by reducing the T_1 execution time is cancelled out by increasing the execution time of task T_2. See Figure 3.12a.

If U' denotes the processor utilization under task set S', we have

$$U - U' = \frac{\Delta}{P_1} - \frac{\Delta}{P_2} > 0 \tag{3.21}$$

Now, suppose that instead of Equation (3.20), we have

$$e_1 = P_2 - P_1 - \Delta \qquad \Delta > 0 \tag{3.22}$$

In such a case, to fully utilize the processor, tasks $T_2, T_3, \ldots$, must fill the intervals $[P_1, P_2]$ and $[P_1 + e_1, P_2]$.

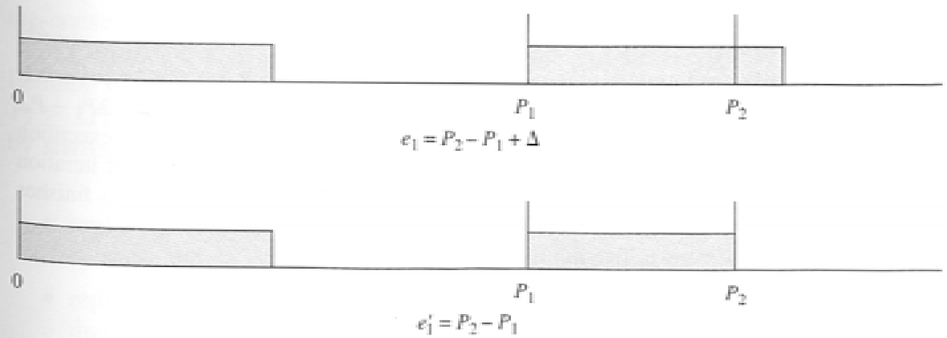

FIGURE 3.12a
Lemma 3.2; the shaded portion indicates when T_1 is executing.

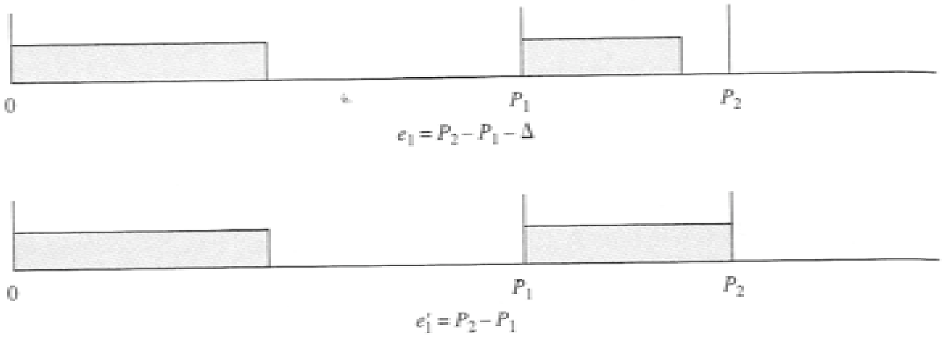

FIGURE 3.12b
Lemma 3.2; the shaded portion indicates when T_1 is executing.

Define another task set S'' with task execution times

$$e_1'' = e_1 + \Delta$$
$$e_2'' = e_2 - 2\Delta$$
$$e_3'' = e_3$$
$$\vdots$$
$$e_n'' = e_n$$

Task set S'' will also fully utilize the processor. Task T_1 will consume an extra Δ time units in the interval $[0, P_1]$, and an additional Δ time units in $[P_1, P_1 + e_1'']$. To make up for this, we reduce the execution time of task T_2 by 2Δ; otherwise T_2 cannot meet its deadline. See Figure 3.12b.

If U' denotes the processor utilization under task set S', we have

$$U - U' = -\frac{\Delta}{P_1} + \frac{(2\Delta)}{P_2} > 0 \qquad (\text{since } P_2 < 2P_1) \tag{3.23}$$

From Equations (3.21) and (3.23), we have that if task set S minimizes the utilization factor,

$$e_1 = P_2 - P_1 \tag{3.24}$$

Using an identical argument, we can show that $e_i = P_{i+1} - P_i$ for $i = 2, \ldots, n-1$. To ensure that the set fully utilizes the processor, we must also have $e_n = 2P_1 - P_n$. To see why, prepare a diagram showing the schedule that results when the execution times are as chosen here. You will see that the only place left for the first iteration of T_n in the schedule is the interval between when the first iteration of T_{n-1} finishes and the second iteration of T_n begins.

The processor utilization under task set S is given by

$$\begin{aligned} U &= \frac{P_2 - P_1}{P_1} + \frac{P_3 - P_2}{P_2} + \cdots + \frac{P_n - P_{n-1}}{P_{n-1}} + \frac{2P_1 - P_n}{P_n} \\ &= \frac{P_2}{P_1} + \frac{P_3}{P_2} + \cdots + \frac{P_n}{P_{n-1}} + \frac{2P_1}{P_n} - n \end{aligned} \tag{3.25}$$

To obtain the minimum possible U so that S completely utilizes the processor, we must therefore choose $P_1, P_2, \ldots, P_n$ to minimize

$$u = \frac{P_2}{P_1} + \frac{P_3}{P_2} + \cdots + \frac{P_n}{P_{n-1}} + \frac{2P_1}{P_n} \tag{3.26}$$

subject to the constraint that $P_n < 2P_1$.

Constrained minimization is quite easy, but unconstrained minimization is even simpler. Let us carry out the unconstrained minimization (i.e., by ignoring the constraint that $P_n < 2P_1$), and see if the result that we obtain satisfies the constraint $P_n < 2P_1$.

To find the unconstrained minimum of u, we must solve the equations

$$\frac{\partial u}{\partial P_1} = \frac{\partial u}{\partial P_2} = \cdots = \frac{\partial u}{\partial P_n} = 0 \tag{3.27}$$

This results in the following equations:

$$P_i = \frac{P_2^{(i-1)}}{P_1^{(i-2)}} \qquad \text{if } 3 \le i \le n \tag{3.28}$$

$$P_n^2 = 2P_1 P_{n-1} \tag{3.29}$$

These equations yield

$$P_i = 2^{(i-1)/n} P_1 \tag{3.30}$$

In particular, if we set $i = n$ in Equation (3.30), we have $P_n = 2^{(n-1)/n} P_1 \Rightarrow P_n < 2P_1$, which satisfies the constraint.

After a little algebra, the corresponding utilization can be shown to be equal to

$$U = n(2^{1/n} - 1) \tag{3.31}$$

Q.E.D.

In fact, we can do better than Lemma 3.2. We can prove the same result without the constraint that $P_n < 2P_1$.

Consider a task set S that satisfies all the conditions in the statement of Lemma 3.2, except the one that $P_n < 2P_1$. Then, there exist tasks T_i such that $P_n \ge 2P_i$. Let $Q \subset S$ denote the set of such tasks. Construct another task set S' by starting with T and

- replacing every task $T_i \in Q$ with a task T_i' that has $P_i' = \lfloor P_n/P_i \rfloor P_i$ and $e_i' = e_i$, and
- replacing task T_n with T_n', which has its execution time increased over that of T_n by the amount required to fully utilize the processor. Let Δ_i be the amount of the increase that compensates for the replacement of T_i by T_i'.

It is easy to see that $\Delta_i \leq (\lfloor P_n/P_i \rfloor - 1)e_i$. Therefore, if U' denotes the utilization of task set T', we have

$$
\begin{aligned}
U' &= U + \sum_{i \in Q} \left\{ \frac{\Delta_i}{P_n} + \frac{e_i}{P_i'} - \frac{e_i}{P_i} \right\} \\
&\leq U + \sum_{i \in Q} \left\{ \left(\frac{\lfloor P_n}{P_i \rfloor} - 1 \right) \frac{e_i}{P_n} + \frac{e_i}{P_i'} - \frac{e_i}{P_i} \right\} \\
&= U + \sum_{i \in Q} \left\{ \left(\left\lfloor \frac{P_n}{P_i} \right\rfloor - 1 \right) e_i \left[\left(\frac{1}{P_n} \right) - \left(\frac{1}{P_i'} \right) \right] \right\} \qquad (3.32)
\end{aligned}
$$

But, $P_n \geq P_i'$. We therefore have from Equation (3.32) that

$$U' \leq U \qquad (3.33)$$

So, task set S', which satisfies the condition that $P_n' < 2P_1'$ in Lemma 3.2, has a lower utilization than S, which has the condition $P_n \geq 2P_1$. However, we know from Lemma 3.2 that $U' \geq n(2^{1/n} - 1)$. It therefore follows that $U \geq n(2^{1/n} - 1)$ for any periodic task set S that fully utilizes the processor. Hence, we have proved the following theorem.

Theorem 3.3. Any set of n periodic tasks that fully utilizes the processor under RM must have a processor utilization of at least $n(2^{(1/n)} - 1)$.

The necessary and sufficient conditions for schedulability are proved below.

Theorem 3.4. 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 $L_i \leq 1$.

Proof. If $L_i \leq 1$, then there exists $t \in [0, P_i]$, such that $W_i(t)/t \leq 1$, that is, $W_i(t) \leq t$. Since $I_i = 0$ for all $i = 1, \ldots, n$ (recall that we have shown that we only need to check the case $I_1 = \ldots = I_n = 0$), $W_i(t) \leq t$ implies that by time t, the computational needs of tasks T_1 to T_i have been met. As $t \leq P_i$ task T_i meets its deadline.

Conversely, if $W_i(t) > t$ for all $t \in [0, P_i]$, there is insufficient time to execute task T_i before its deadline, P_i. Q.E.D.

WHEN A TASK DEADLINE IS NOT EQUAL TO ITS PERIOD.* We have so far assumed that the relative deadline of a task is equal to its period. Let us relax this assumption. If we do so, the RM algorithm is no longer an optimum static-priority scheduling algorithm. Consider first the case where the relative deadline is less than the period. Then, a moment's reflection shows that the necessary and sufficient condition for task T_i to be RM-schedulable is

$$W_i(t) = t \qquad \text{for some } t \in [0, d_i] \qquad (3.34)$$

The case $d_i > P_i$ is much harder. Let us begin by considering again our result that the worst-case response time of a task occurs when the task phasings are all

zero. When $d_i \leq P_i$, at most one initiation of the same task can be alive at any one time. As a result, to check for schedulability it is sufficient to set the phasings of all members of the task set to zero and to check that the first initiation of each task meets its deadline. That is, in fact, the origin of RM-scheduling conditions **RM1** and **RM2**. When $d_i > P_i$, however, it is possible for multiple initiations of the same task to be alive simultaneously, and we might have to check a number of initiations to obtain the worst-case response time. To clarify this, consider the Example 3.13.

Example 3.13. Consider a case where $n = 2$, $e_1 = 28$, $e_2 = 71$, $P_1 = 80$, and $P_2 = 110$. Set all task deadlines to infinity. The following table shows the response times of task T_2.

Initiation	Completion time	Response time
0	127	127
110	226	116
220	353	133
330	452	122
440	551	111
550	678	128
660	777	117
770	876	106

As we can see, the worst response time is not for the first initiation, but for the third. This indicates that it is not sufficient just to consider the first initiation of all the tasks.

We must do some additional work before we can write down the schedulability condition for the $d_i > P_i$ case. In this case, more than one iteration of a task can be alive at any one time. As before, we assume that T_i has priority over T_j iff $P_i < P_j$; indeed, we number the tasks in the ascending order of their periods (and thus in the descending order of their priorities).

Let $S_i = \{T_1, \ldots, T_i\}$. We define the *level-i busy period* as the interval $[a, b]$ such that

- $b > a$
- only tasks in S_i are run in $[a, b]$,
- the processor is busy throughout $[a, b]$, and
- the processor is not executing any task from S_i just prior to a or just after b.

Example 3.14. Define $S_i = \{T_1, \ldots, T_i\}$ for $i = 1, \ldots, 5$. In the schedule in Figure 3.13 shows the five busy-period levels.

Theorem 3.5. Task T_i experiences its greatest response time during a level-i busy period, initiated with phasings $I_1 = \ldots = I_i = 0$.

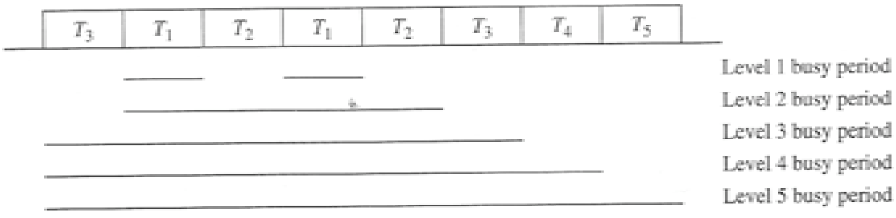

FIGURE 3.13
Busy periods.

Proof. The proof is immediate for the highest-priority task, task T_1. So, consider tasks T_i, for $i > 1$ and, without loss of generality, assume that $I_1 = 0$. Suppose $[0, b)$ is a level-i busy period and $I_i > 0$. By the definition of busy period, only tasks of higher priority than T_i execute in the interval $[0, I_i)$. Decreasing I_i will not change the times at which these higher-priority tasks finish; all it will do is to increase the response time of T_i. Similarly, if $I_k > 0$ for some $k < i$, reducing I_k will either increase or leave unchanged the processing demands of T_k over the interval $[0, b)$. That is, reducing I_k will either increase or leave unchanged the finishing time of T_k. This completes the proof. Q.E.D.

Thus, to determine RM-schedulability, we can continue to concentrate on the case where all phasings are zero. However, to ensure that task T_i meets its deadline, we must check that all its initiations in a level-i busy period beginning at time 0 meet their deadlines.

Let $t(k, i)$ be when the kth initiation (within the busy period) of task T_i completes execution. We leave for the reader to show that $t(k, i)$ is the minimum t for which the following expression holds:

$$t = \sum_{j=1}^{i-1} e_j \left\lceil \frac{t}{P_j} \right\rceil + k e_i \tag{3.35}$$

This kth initiation will meet its deadline if

$$t(k, i) < (k-1)P_i + d_i \tag{3.36}$$

To what value of k should we check that the above condition holds to ensure that all iterations of T_i meet their deadline? We leave for the reader to show that it is sufficient to check that iterations 1 to ℓ_i meet their deadlines, where

$$\ell_i = \min\{m \mid m P_i > t(m, i)\} \tag{3.37}$$

Task T_i is thus RM-schedulable iff

$$t(k, i) < (k-1)P_i + d_i, \quad \forall k \le \ell_i \tag{3.38}$$

and the entire task set is RM-schedulable iff all the tasks in it are RM-schedulable.

Can we obtain a results similar to Theorem 3.3 for the case $d_i \neq P_i$? It is surprisingly difficult to do this and few results are known. We will state some of

these without proof. Suppose we have a task set for which there exists a γ such that $d_i = \gamma P_i$, for all the tasks. Then it is possible to show the following result.

Theorem 3.6. Any set of n periodic tasks that fully utilizes the processor under RM must have a processor utilization of at least

$$U = \begin{cases} n\left(2^{1/n} - 1\right) & \text{if } \gamma = 1 \\ \gamma(n-1)\left[\left(\dfrac{\gamma+1}{\gamma}\right)^{1/(n-1)} - 1\right] & \text{if } \gamma = 2, 3, \ldots \\ \gamma & \text{if } 0 \le \gamma \le 0.5 \\ \log_e(2\gamma) + 1 - \gamma & \text{if } 0.5 \le \gamma \le 1 \end{cases} \tag{3.39}$$

HANDLING CRITICAL SECTIONS. In our discussions so far, we have assumed that all tasks can be preempted at any point of their execution. However, sometimes tasks may need to access resources that cannot be shared. For example, a task may be writing to a block in memory. Until this is completed, no other task can access that block, either for reading or for writing. A task that is currently holding the unsharable resource is said to be in the *critical section* associated with the resource.

One way of ensuring exclusive access is to guard the critical sections with *binary semaphores*. These are like locks. When the semaphore is locked (e.g., by setting it to 1), it indicates that there is a task currently in the critical section. When a task seeks to enter a critical section, it checks if the corresponding semaphore is locked. If it is, the task is stopped and cannot proceed further until that semaphore is unlocked. If it is not, the task locks the semaphore and enters the critical section. When a task exits the critical section, it unlocks the corresponding semaphore. For convenience, we shall say that a critical section S is locked (unlocked) when we mean that the semaphore associated with S is locked (unlocked).

We will assume that critical sections are properly nested. That is, if we have sections S_1, S_2 on a single processor, the following sequence is allowed: `Lock` S_1. `Lock` S_2. `Unlock` S_2. `Unlock` S_1, while the following is not: `Lock` S_1. `Lock` S_2. `Unlock` S_1. `Unlock` S_2.

Everything in this section refers to tasks sharing a single processor. We assume that once a task starts, it continues until it (a) finishes, (b) is preempted by some higher-priority task, or (c) is blocked by some lower-priority task that holds the lock on a critical section that it needs. We do not, for example, consider a situation where a task suspends itself when executing I/O operations or when it encounters a page fault. The results of this section can easily be extended for this case, however (see Exercise 3.12).

It is possible for a lower-priority task T_L to block[2] a higher-priority task, T_H. This can happen when T_H needs to access a critical section that is currently

[2]When a lower-priority task is in the way of a higher-priority task, the former is said to *block* the latter.

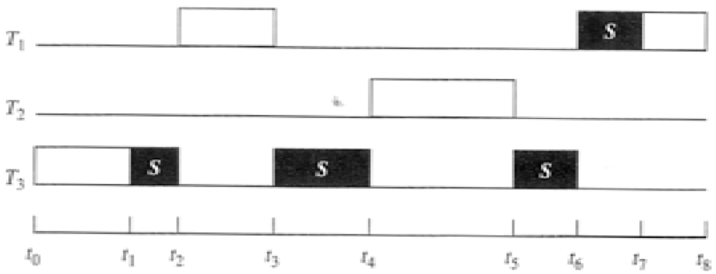

FIGURE 3.14
Priority inversion.

being accessed by T_L. Although T_H has higher priority than T_L, to ensure correct functioning, T_L must be allowed to complete its critical section access before T_H can access it.

Such blocking of a higher-priority task by a lower-priority task can have the unpleasant side effect of priority inversion. This is illustrated in Example 3.15.

Example 3.15. Consider tasks T_1, T_2, T_3, listed in descending order of priority, which share a processor. There is one critical section S that both T_1 and T_3 use. See Figure 3.14. T_3 begins execution at time t_0. At time t_1, it enters its critical section, S. T_1 is released at time t_2 and preempts T_3. It runs until t_3, when it tries to enter the critical section S. However, S is still locked by the suspended task T_3. So, T_1 is suspended and T_3 resumes execution. At time t_4, task T_2 is released. T_2 has higher priority than T_3, and so it preempts T_3. T_2 does not need S and runs to completion at t_5. After T_2 completes execution at t_5, T_3 resumes and exits critical section S at t_6. T_1 can now preempt T_3 and enter the critical section.

Notice that although T_2 is of lower priority than T_1, it was able to delay T_1 indirectly (by preempting T_3, which was blocking T_1). This phenomenon is known as *priority inversion*. Ideally, the system should have noted that T_1 was waiting for access, and so T_2 should not have been allowed to start executing at t_4.

The use of *priority inheritance* allows us to avoid the problem of priority inversion. Under this scheme, if a higher-priority task T_H is blocked by a lower-priority task T_L (because T_L is currently executing a critical section needed by T_H), the lower-priority task temporarily inherits the priority of T_H. When the blocking ceases, T_L resumes its original priority. The protocol is described in Figure 3.15. Example 3.16 shows how this prevents priority inversion from happening.

Example 3.16. Let us return to Example 3.15 to see how priority inheritance prevents priority inversion. At time t_3, when T_3 blocks T_1, T_3 inherits the higher priority of T_1. So, when T_2 is released at t_4, it cannot interrupt T_3. As a result, T_1 is not indirectly blocked by T_2.

1. The highest-priority task T is assigned the processor. T relinquishes the processor whenever it seeks to lock the semaphore guarding a critical section that is already locked by some other job.
2. If a task T_1 is blocked by T_2 (due to contention for a critical section) and $T_1 \succ T_2$, task T_2 inherits the priority of T_1 as long as it blocks T_1. When T_2 exits the critical section that caused the block, it reverts to the priority it had when it entered that section. The operations of priority inheritance and the resumption of previous priority are indivisible.
3. Priority inheritance is transitive. If T_3 blocks T_2, which blocks T_1 (with $T_1 \succ T_2 \succ T_3$), then T_3 inherits the priority of T_1 through T_2.
4. A task T_1 can preempt another task T_2 if T_1 is not blocked and if the current priority of T_1 is greater than that of the current priority of T_2.

FIGURE 3.15
The priority inheritance protocol.

Unfortunately, priority inheritance can lead to deadlock. This is illustrated by Example 3.17.

Example 3.17. Consider two tasks T_1 and T_2, which use two critical sections S_1 and S_2. These tasks require the critical sections in the following sequence:

T_1: Lock S_1. Lock S_2. Unlock S_2. Unlock S_1.
T_2: Lock S_2. Lock S_1. Unlock S_1. Unlock S_2.

Let $T_1 \succ T_2$, and suppose T_2 starts execution at t_0. At time t_1, it locks S_2. At time t_2, T_1 is initiated and it preempts T_2 owing to its higher priority. At time t_3, T_1 locks S_1. At time t_4, T_1 attempts to lock S_2, but is blocked because T_2 has not finished with it. T_2, which now inherits the priority of T_1, starts executing. However, when at time t_5 it tries to lock S_1, it cannot do so since T_1 has a lock on it. Both T_1 and T_2 are now *deadlocked*.

There is another drawback of priority inheritance. It is possible for the highest-priority task to be blocked once by every other task executing on the same processor. (The reader is invited in Exercise 3.8 to construct an example of this.)

To get around both problems, we define the priority ceiling protocol. The *priority ceiling* of a semaphore is the highest priority of any task that may lock it. Let $P(T)$ denote the priority of task T, and $P(S)$ the priority ceiling of the semaphore of critical section S.

Example 3.18. Consider a three-task system T_1, T_2, T_3, with $T_1 \succ T_2 \succ T_3$. There are four critical sections, and the following table indicates which tasks may lock which sections, and the resultant priority ceilings.

Critical section	Accessed by	Priority ceiling
S_1	T_1, T_2	$P(T_1)$
S_2	T_1, T_2, T_3	$P(T_1)$
S_3	T_3	$P(T_3)$
S_4	T_2, T_3	$P(T_2)$

The priority ceiling protocol is the same as the priority inheritance protocol, except that a task can also be blocked from entering a critical section if there exists any semaphore currently held by some other task whose priority ceiling is greater than or equal to the priority of T.

Example 3.19. Consider the tasks and critical sections mentioned in Example 3.18. Suppose that T_2 currently holds a lock on S_2, and task that T_1 is initiated. T_1 will be blocked from entering S_1 because its priority is not greater than the priority ceiling of S_2.

The priority ceiling protocol is specified in Figure 3.16. The key properties of the priority ceiling protocol are as follows:

P1. The priority ceiling protocol prevents deadlocks.

P2. Let B_i be the set of all critical sections that can cause the blocking of task T_i and $t(x)$ be the time taken for section x to be executed. Then, T_i will be blocked for at most $\max_{x \in B_i} t(x)$.

---

1. The highest-priority task, T, is assigned the processor. T relinquishes the processor (i.e., it is blocked) whenever it seeks to lock the semaphore guarding a critical section which is already locked by some other task Q (in which case it is said to be blocked by task Q), or when there exists a semaphore S' locked by some other task, whose priority ceiling is greater than or equal to the priority of T. In the latter case, let S^* be the semaphore with the highest priority among those locked by some other tasks. We say that T is blocked on S^*, and by the task currently holding the lock on S^*.
2. Suppose T blocks one or more tasks. Then, it inherits the priority of the highest-priority task that it is currently blocking. The operations of priority inheritance and resumption of previous priority are indivisible.
3. Priority inheritance is transitive.
4. A task T_1 can preempt another task T_2 if T_2 does not hold a critical section which T_1 currently needs, and if the current priority of T_1 is greater than that of the current priority of T_2.

FIGURE 3.16
The priority ceiling protocol.

Priority ceiling property **P2** allows us to conduct a schedulability analysis on systems using the priority ceiling protocol. Take, for example, the rate-monotonic scheduling algorithm that we discussed earlier in this section. We can revise Theorem 3.6 as follows (here we use T_i to denote both the task and its period—which one it represents is obvious from the context):

Theorem 3.7. Any set of n periodic processes that fully utilizes the processor under RM must have, for each $i \in \{1, \ldots, n\}$

$$\frac{e_1}{T_1} + \frac{e_2}{T_2} + \cdots + \frac{e_i}{T_i} + \frac{b_i}{T_i} \leq i(2^{1/i} - 1)$$

where $b_i = \max_{x \in B_i} t(x)$.

Proof. The proof is left to the reader as an exercise.

As a result of Theorem 3.7, we know that task T_i can be scheduled under the RM algorithm to meet its deadline if

$$\frac{e_1}{T_1} + \frac{e_2}{T_2} + \cdots + \frac{e_i}{T_i} + \frac{b_i}{T_i} \leq i(2^{1/i} - 1).$$

The necessary and sufficient conditions for RM-schedulability can be similarly written.

MATHEMATICAL UNDERPINNINGS OF THE PRIORITY CEILING ALGORITHM.* To prove that priority ceiling properties **P1** and **P2** hold, we will need the following series of results.

Lemma 3.3. Task T_1 can only be blocked by a lower-priority task T_2 if T_2 is in a critical section at the time that T_1 arrives.

Proof. If T_2 is not in a critical section when T_1 arrives, it will be preempted and will never regain the processor until after T_1 leaves. Q.E.D.

Lemma 3.4. Task T_1 can only be blocked by a lower-priority task T_2 if the priority of T_1 is no greater than the greatest priority of all the semaphores currently locked by all lower-priority tasks.

Proof. This follows immediately from the definition of the priority ceiling protocol. Q.E.D.

Lemma 3.5. Suppose that task T_2 is currently executing critical section S_2, and that it is preempted by a higher-priority task T_1 that then executes critical section S_1. It is impossible for T_2 to inherit a priority greater than or equal to that of T_1, until T_1 finishes execution.

Proof. T_1 can only execute S_1 if

$$P(T_1) > \text{ceil}(S_2) \tag{3.40}$$

T_2 can only inherit the priority of some task T if T is being blocked on S_2. But then,

$$\text{ceil}(S_2) \geq P(T) \tag{3.41}$$

It follows from Equations (3.40) and (3.41) that

$$P(T_1) > P(T) \tag{3.42}$$

Q.E.D.

Lemma 3.6. The priority ceiling protocol prevents transitive blocking.

Proof. Again, we prove the result by contradiction. Let the lemma be false. Then there must exist tasks T_1, T_2, T_3 such that $T_1 \succ T_2 \succ T_3$ and where T_3 blocks T_2, which blocks T_1. But this would mean that T_3 would inherit the priority of T_1. This contradicts Lemma 3.5. Q.E.D.

We now have the means to prove property **P1**.

Theorem 3.8. The priority ceiling protocol prevents deadlocks.

Proof. Deadlock can only occur if we have a cycle of n tasks each blocking on the one in front of it; see the example in Figure 3.17. (We are assuming that a task never deadlocks with itself.) Since we have shown in Lemma 3.6 that transitive blocking is impossible, the largest cycle we can have consists of just two tasks (i.e., $n = 2$). Assume that T_2 is preempted by T_1 when T_2 is in a set of critical sections σ_2. Suppose that then T_1 enters some critical section S_1. This can only happen if no member $S_2 \in \sigma_2$ is ever required by T_1 itself (otherwise S_2 would have priority equal to that of T_1 and T_1 would not be allowed to enter any critical section as long as T_2 was holding S_2). Thus there is no possibility of a deadlock. Q.E.D.

In the following, let $T_1 \succ T_2$, and $B_{1,2}$ be the set of critical sections of T_2 that can block T_1. Let $b_{1,2}$ be the critical section in $B_{1,2}$ that takes the longest time to execute.

Lemma 3.7. T_1 can be blocked by T_2 by at most $b_{1,2}$.

Proof. Since $T_1 \succ T_2$, T_1 can only be blocked by T_2 if T_2 is executing a critical section in $B_{1,2}$, deadlock is not possible (by Theorem 3.8). T_2 (which will inherit the priority of T_1) will exit that critical section within at most $b_{1,2}$ unless it is preempted by some task $T \succ T_1$. If such a preemption happens, T_1 will no longer be blocked by T_2. If no such preemption occurs, T_2 will exit T_2 within at most $b_{1,2}$ and not resume execution until T_1 has completed execution. Q.E.D.

We are now ready to prove that property **P2** holds. To facilitate this, define β_i as the set of all critical sections used by tasks T_j such that $T_i \succ T_j$. Define b_i as the greatest execution time of any critical section in β_i.

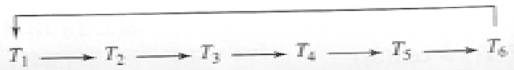

FIGURE 3.17
Six-task deadlocked system; the arrows indicate a "waiting-for" relationship.

Theorem 3.9. Task T_i can be blocked by at most one lower-priority task, and for a duration of at most b_i.

Proof. We prove this result by contradiction. Suppose task T_i can be blocked by more than b_i. This can only happen if it is blocked by $n > 1$ distinct tasks (since we know from Lemma 3.7 that T_i can be blocked at most once by any one lower-priority task).

Suppose that T_i is blocked by T_1 and T_2. Assume, without loss of generality, that $T_1 \succ T_2$. Of course, $T_i \succ T_1$ and $T_i \succ T_2$. Suppose T_i is blocked by T_1 in S_1 and by T_2 in S_2. (If either or both of these tasks is in a nested set of semaphores, focus on the outermost one). Then, T_1 and T_2 must have been in S_1 and S_2, respectively, when T_i arrived. Furthermore, T_2 must have been in S_2 when T_1 arrived.

Since T_1 enters S_1 with T_2 in S_2, we must have

$$P(T_1) > \text{ceil}(S_2) \tag{3.43}$$

Since T_i is blocked by T_1 on S_1, we must have

$$P(T_i) \leq \text{ceil}(S_1) \tag{3.44}$$

Similarly, since T_i is blocked by T_2 on S_2,

$$P(T_i) \leq \text{ceil}(S_2) \tag{3.45}$$

But, this implies that

$$P(T_1) > P(T_i) \tag{3.46}$$

which is a contradiction. Q.E.D.

3.2.2 Preemptive Earliest Deadline First (EDF) Algorithm

A processor following the EDF algorithm always executes the task whose absolute deadline is the earliest. EDF is a *dynamic-priority* scheduling algorithm; the task priorities are not fixed but change depending on the closeness of their absolute deadline. EDF is also called the deadline-monotonic scheduling algorithm.

Example 3.20. Consider the following set of (aperiodic) task arrivals to a system.

Task	Arrival time	Execution time	Absolute deadline
T_1	0	10	30
T_2	4	3	10
T_3	5	10	25

When T_1 arrives, it is the only task waiting to run, and so starts executing immediately. T_2 arrives at time 4; since $d_2 < d_1$, it has higher priority than T_1 and preempts it. T_3 arrives at time 5; however, since $d_3 > d_2$, it has lower priority than T_2 and must wait for T_2 to finish. When T_2 finishes (at time 7), T_3 starts (since it has higher priority than T_1). T_3 runs until 15, at which point T_1 can resume and run to completion. 17

In our treatment of the EDF algorithm, we will make all the assumptions we made for the RM algorithm, except that the tasks do not have to be periodic.

EDF is an optimal uniprocessor scheduling algorithm. That is, if EDF cannot feasibly schedule a task set on a uniprocessor, there is no other scheduling algorithm that can.

If all the tasks are periodic and have relative deadlines equal to their periods, the test for task-set schedulability is particularly simple:

> If the total utilization of the task set is no greater than 1, the task set can be feasibly scheduled on a single processor by the EDF algorithm.

There is no simple schedulability test corresponding to the case where the relative deadlines do not all equal the periods; in such a case, we actually have to develop a schedule using the EDF algorithm to see if all deadlines are met over a given interval of time. The following is a schedulability test for EDF under this case.

Define $u = \sum_{i=1}^{n}(e_i/P_i)$, $d_{max} = \max_{1\le i\le n}\{d_i\}$ and $P = \text{lcm}(P_1, \ldots P_n)$. (Here "lcm" stands for least common multiple.) Define $h_T(t)$ to be the sum of the execution times of all tasks in set T whose absolute deadlines are less than t. A task set of n tasks is not EDF-feasible iff

- $u > 1$ or
- there exists

$$t < \min\left\{P + d_{max}, \quad \frac{u}{1-u}\max_{1\le i\le n}\{P_i - d_i\}\right\}$$

such that $h_T(t) > t$.

MATHEMATICAL UNDERPINNINGS.* As we said earlier, EDF is an optimal uniprocessor scheduling algorithm (i.e., if a set of tasks cannot be feasibly scheduled under EDF, there is no other uniprocessor algorithm that can feasibly schedule them).

Theorem 3.10. EDF is optimal for uniprocessors.

Proof. The proof is by contradiction. Assume that the theorem is not true and that there is some other algorithm Σ that is optimal. Then, there must exist some set of tasks S such that S is Σ-schedulable but not EDF-schedulable. Let us focus on this set S.

Suppose that t_2 is the earliest absolute deadline that is missed by the EDF algorithm. Define t_1 as the last instant, prior to t_2, at which EDF had the processor working on a task whose absolute deadline exceeded t_2. If no such instant exists, set $t_1 = 0$. Since only tasks with absolute deadlines $\le t_2$ are scheduled by EDF in the interval $[t_1, t_2]$, any task executing in that interval must have been released at or after t_1. The reason is that at t_1, the processor was executing a task with an absolute deadline $> t_2$, which would only have been possible under EDF if there was no pending task at t_1 with an absolute deadline $\le t_2$.

Define

$$A = \{T_i | T_i \text{ is released in } [t_1, t_2] \text{ and } D_i < t_2\}$$

$$B = \{T_i | T_i \text{ is released in } [t_1, t_2] \text{ and } D_i \ge t_2\}$$

By the definition of t_2, B is nonempty. Also by the definition of the t_2, all the deadlines of the tasks in A are met by both EDF and Σ. We have two cases.

Case 1. Under EDF, the processor is continuously busy over $[t_1, t_2]$.

Let $E^{EDF}(A)$, $E^{EDF}(B)$, $E^{\Sigma}(A)$, $E^{\Sigma}(B)$ be the execution time over $(t_1, t_2]$ allocated by EDF and Σ to the tasks in A and B, respectively. Then,

$$E^{EDF}(A) + E^{EDF}(B) = t_2 - t_1 \tag{3.47}$$

Since all the deadlines of the tasks in A are met by both EDF and Σ,

$$E^{EDF}(A) = E^{\Sigma}(A) \tag{3.48}$$

However, since at least one task in B misses its deadline under EDF, we must have

$$E^{EDF}(B) < E^{\Sigma}(B) \tag{3.49}$$

Hence, in the interval $(t_1, t_2]$, under Σ, the processor is used for

$$\begin{aligned} E^{\Sigma}(A) + E^{\Sigma}(B) &> E^{EDF}(A) + E^{EDF}(B) \quad &&\text{[from (3.48) and (3.49)]} \\ &= t_2 - t_1 &&\text{[from (3.47)]} \end{aligned} \tag{3.50}$$

But that is plainly impossible, and we have a contradiction.

Case 2. Under EDF, the processor is idle over some part of $(t_1, t_2]$.

Let t_3 be the last instant in $(t_1, t_2]$ at which the processor is idle under the EDF discipline. Since EDF causes a deadline to be missed at t_2, $t_3 < t_2$.

The processor can only be idle at t_3 if there are no pending requests for execution, that is, if every task released prior to t_3 has been executed. The argument we made in Case 1 over the interval $(t_1, t_2]$ now applies over the interval $(t_3, t_2]$, and so here too we have a contradiction. This completes the proof. Q.E.D.

Let us now turn to periodic task sets. We will first consider the case where for every task the relative deadline equals the task period, and show that the necessary and sufficient condition for a task set to be schedulable is $\sum_{i=1}^{n} e_i/P_i \le 1$. To begin with, as with the RM algorithm, it is sufficient to consider the case where all the task phasings are zero. We then proceed in two steps. First, we show that if a deadline is indeed missed under EDF, the processor will be continuously busy from time 0 to when it missed the deadline. This tells us the amount of work that the processor has completed up to that time. We can then use this information to show that if $\sum_{i=1}^{n} e_i/P_i \leq 1$, all deadlines will indeed be satisfied. In what follows, we will assume that all task phasings are zero.

Lemma 3.8. If a deadline is missed for the first time at t_f, the processor is continuously busy throughout the interval $[0, t_f]$.

Proof. We proceed by contradiction. Suppose this lemma is not true and that the processor was idle at some time within the interval $[0, t_f]$ when all the task phasings are 0 (i.e., the first iteration of each task is released at time 0). Let t_1 be the last such time; that is, the processor was idle at t_1, but busy in the interval $(t_1, t_f]$.

If the processor was idle at t_1, it must be that all the tasks released prior to t_1 have been completed by t_1. Therefore, all tasks executing in the interval $(t_1, t_f]$ must have been released in that interval and no task released prior to t_1 will affect the scheduling in $(t_1, t_f]$ (since all such tasks have completed and left the system). Now, construct a new task phasing so that every task has one iteration released at t_1. (See Figure 3.18). Under this case, also, a deadline will be missed; in particular,

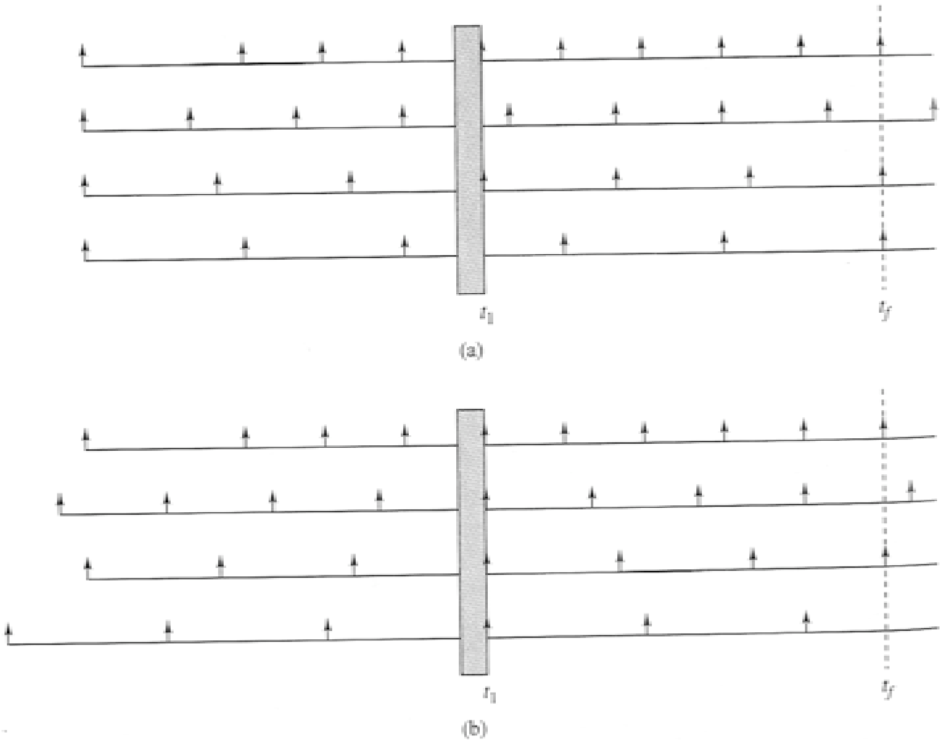

FIGURE 3.18
Lemma 3.8: (a) schedule with zero phasing; (b) schedule with new phasing. The processor is idle over the shaded portions.

it will be missed at time $t' \leq t_f$. Also, the processor cannot be idle at any time in $(t_1, t']$ under this new task phasing. (Why?)

Now, compare the situation under the new task phasing over the interval $(t_1, t_f]$ with the situation under the zero task phasing over the interval $(0, t_1]$. The load presented to the processor at time t_1 is the same under the new task phasing as it was at time 0 under the zero task phasing. But, we have argued that under the new phasing, the processor will be busy throughout $(t_1, t']$, and miss a deadline at t'. Therefore, under the zero task phasing the processor cannot have been idle before the deadline was missed. We therefore have a contradiction and the lemma is proved. Q.E.D.

Now, we are ready to prove Theorem 3.11, which contains the necessary and sufficient condition:

Theorem 3.11. Suppose we have a set of n periodic tasks, each of whose relative deadline equals its period. They can be feasibly scheduled by EDF iff

$$\sum_{i=1}^{n}(e_i/P_i) \leq 1$$

Proof. Proving that scheduling is impossible if $\sum_{i=1}^{n}(e_i/P_i) > 1$ is the easy part—we simply show that the processor utilization would have to exceed 1, which is impossible. Suppose that $\sum_{i=1}^{n}(e_i/P_i) > 1$. Let P be the least common multiple (lcm) of $\{P_1, \ldots, P_n\}$ and $\ell_i = P/P_i$. Then, over the interval $[kP, (k+1)P]$, $k = 0, 1, \ldots$, the processor will receive requests for a total of

$$\sum_{i=1}^{n} \ell_i e_i = P\left\{\sum_{i=1}^{n} \frac{e_i}{P_i}\right\} > P \tag{3.51}$$

units of work. Requests for processor time thus arrive at a higher rate than they can be met and the unfinished work will pile up without limit as time goes on. Hence, the task set cannot be feasibly scheduled if $\sum_{i=1}^{n}(e_i/P_i) > 1$.

Next we must prove that if $\sum_{i=1}^{n}(e_i/P_i) \leq 1$, EDF will indeed schedule successfully. This is a little harder and we proceed by contradiction. Suppose that this theorem is not true and that there exists some task set S of n tasks that are not EDF-schedulable, despite $\sum_{i=1}^{n}(e_i/P_i) \leq 1$. Let t_f be the earliest time at which a deadline is missed. Since the set of tasks is finite, such an earliest time does exist and $t_f > 0$. Let S_f denote the set of tasks in S with an absolute deadline equal to t_f.

From Lemma 3.8, we know that the processor must be busy throughout the interval $[0, t_f]$. There are now two cases to consider:

Case 1. None of the tasks executed in $[0, t_f]$ have absolute deadlines beyond t_f. The number of iterations of task T_i that have to be completed in $[0, t_f]$ is $\lfloor t_f/P_i \rfloor$, since all other iterations have absolute deadlines expiring after t_f. But the processor is busy throughout the interval $[0, t_f]$. Hence, it must be that the tasks whose absolute deadlines were less than t_f imposed a demand of an execution time of more than t_f on the processor. In other words, we must have:

$$\left\lfloor \frac{t_f}{P_1} \right\rfloor e_1 + \left\lfloor \frac{t_f}{P_2} \right\rfloor e_2 + \cdots + \left\lfloor \frac{t_f}{P_n} \right\rfloor e_n > t_f$$

$$\Rightarrow \left(\frac{t_f}{P_1}\right) e_1 + \left(\frac{t_f}{P_2}\right) e_2 + \cdots + \left(\frac{t_f}{P_n}\right) e_n > t_f$$

$$\Rightarrow \sum_{i=1}^{n} \frac{e_i}{P_i} > 1 \tag{3.52}$$

which contradicts the assumption that $\sum_{i=1}^{n}(e_i/P_i) \leq 1$.

Case 2. Some tasks executed in the interval $[0, t_f]$ have absolute deadlines beyond t_f. We handle this much as we handled Lemma 3.8. Let τ be the last time before t_f that a task with absolute deadline greater than t_f was executed. Since we are using the EDF algorithm, if such a task was being executed at τ, there must be, at time τ, no tasks awaiting service that have absolute deadlines expiring at or before t_f. Thus, all the tasks that are executed in the interval $[\tau, t_f]$ must be released in that interval. But since a deadline was missed, we must have that the total demand upon the processor during that interval was greater than the length of that interval. In other words, we must have:

$$\left\lfloor \frac{(t_f - \tau)}{P_1} \right\rfloor e_1 + \left\lfloor \frac{(t_f - \tau)}{P_2} \right\rfloor e_2 + \cdots + \left\lfloor \frac{(t_f - \tau)}{P_n} \right\rfloor e_n > t_f - \tau$$

$$\Rightarrow \left[\frac{(t_f - \tau)}{P_1}\right] e_1 + \left[\frac{(t_f - \tau)}{P_2}\right] e_2 + \cdots + \left[\frac{(t_f - \tau)}{P_n}\right] e_n > t_f - \tau$$

$$\Rightarrow \sum_{i=1}^{n} \frac{e_i}{P_i} > 1 \tag{3.53}$$

which is a contradiction.

This completes the proof. Q.E.D.

Theorem 3.11 allows us to quickly check the feasibility of any allocation when the relative task deadlines equal their periods. Unfortunately, there is no efficient way to check feasibility if the relative deadlines do not all equal their periods or if there are sporadic tasks. In order to verify schedulability, we have to actually schedule the task set using EDF and then check if all the deadlines have been satisfied. Since we can't check schedulability for an infinite number of cases, we must obtain a *finiteness* result, which says that if deadlines are ever missed the time of the earliest missed deadline will have a known upper bound. Then we only need to check feasibility up to that point.

Just as with the RM algorithm, it is easy to show that the worst-case execution time of a task occurs when all the task phasings are zero. So, if we verify schedulability for this case, it will hold for all task phasings.

For the finiteness result, we define $u = \sum_{i=1}^{n}(e_i/P_i)$, $d_{\max} = \max_{1\leq i\leq n}\{d_i\}$ and $P = \mathrm{lcm}(P_1, \ldots P_n)$. Define $h_T(t)$ to be the sum of the execution times of all the tasks in set T whose absolute deadlines are less than or equal to t.

Theorem 3.12. A task set of n tasks is not EDF-feasible iff

- $u > 1$ or
- there exists

$$t < \min\left\{P + d_{\max},\ \frac{u}{1-u}\max_{1\leq i\leq n}\{P_i - d_i\}\right\}$$

such that $h_T(t) > t$.

Under this theorem, we only need to check for feasibility up to some finite time. We can build the proof of Theorem 3.12 using the following series of lemmas.

Lemma 3.9. A given set T of periodic tasks is not EDF-schedulable iff there exists some time t such that $h_T(t) > t$.

Proof. This has been left to the reader.

Lemma 3.10. Given a set T of n periodic tasks, if $u \leq 1$,

$$h_T(t + P) > t + P \Rightarrow h_T(t) > t \qquad \text{for all } t \geq d_{\max}$$

Proof

$$\begin{aligned} h_T(t) + P &= \sum_{i=1}^{n} e_i\left(\left\lfloor\frac{t-d_i}{P_i}\right\rfloor + 1\right) + P \\ &\geq \sum_{i=1}^{n} e_i\left(\left\lfloor\frac{t-d_i}{P_i}\right\rfloor + 1\right) + P\sum_{i=1}^{n}\frac{e_i}{P_i} \\ &= \sum_{i=1}^{n} e_i\left(\left\lfloor\frac{t-d_i+P}{P_i}\right\rfloor + 1\right) \qquad \text{since } P \text{ is a multiple of } P_i \\ &= h_T(t+P) \end{aligned}$$

Hence,

$$h_T(t + P) > t + P \Rightarrow h_T(t) > t \tag{3.54}$$

Q.E.D.

Lemma 3.11. If task set T is not EDF-feasible and $u \leq 1$, then there exists $t < P + d_{\max}$ such that $h_T(t) > t$.

Proof. Follows immediately from Lemma 3.10. Q.E.D.