4

PERIODIC TASK SCHEDULING

4.1 INTRODUCTION

In many real-time control applications, periodic activities represent the major computational demand in the system. Periodic tasks typically arise from sensory data acquisition, low-level servoing, control loops, action planning, and system monitoring. Such activities need to be cyclically executed at specific rates, which can be derived from the application requirements. Some specific examples of real-time applications are illustrated in Chapter 10.

When a control application consists of several concurrent periodic tasks with individual timing constraints, the operating system has to guarantee that each periodic instance is regularly activated at its proper rate and is completed within its deadline (which, in general, could be different than its period).

In this chapter three basic algorithms for handling periodic tasks are described in detail: Rate Monotonic, Earliest Deadline First, and Deadline Monotonic. Schedulability analysis is performed for each algorithm in order to derive a guarantee test for generic task sets. To facilitate the description of the scheduling results presented in this chapter, the following notation is introduced:

Γ denotes a set of periodic tasks;

τ_i denotes a generic periodic task;

$\tau_{i,j}$ denotes the jth instance of task τ_i;

$r_{i,j}$ denotes the release time of the jth instance of task τ_i;

Φ_i denotes the *phase* of task τ_i; that is, the release time of its first instance ($\Phi_i = r_{i,1}$);

D_i denotes the relative deadline of task τ_i;

$d_{i,j}$ denotes the absolute deadline of the jth instance of task τ_i ($d_{i,j} = \Phi_i + (j-1)T_i + D_i$).

$s_{i,j}$ denotes the start time of the jth instance of task τ_i; that is, the time at which it starts executing.

$f_{i,j}$ denotes the finishing time of the jth instance of task τ_i; that is, the time at which it completes the execution.

Moreover, in order to simplify the schedulability analysis, the following hypotheses are assumed on the tasks:

A1. The instances of a periodic task τ_i are regularly activated at a constant rate. The interval T_i between two consecutive activations is the *period* of the task.

A2. All instances of a periodic task τ_i have the same worst case execution time C_i.

A3. All instances of a periodic task τ_i have the same relative deadline D_i, which is equal to the period T_i.

A4. All tasks in Γ are independent; that is, there are no precedence relations and no resource constraints.

In addition, the following assumptions are implicitly made:

A5. No task can suspend itself, for example on I/O operations.

A6. All tasks are released as soon as they arrive.

A7. All overheads in the kernel are assumed to be zero.

Notice that assumptions A1 and A2 are not restrictive because in many control applications each periodic activity requires the execution of the same routine at regular intervals; therefore, both T_i and C_i are constant for every instance. On the other hand, assumptions A3 and A4 could be too tight for practical

applications. However, the four assumptions are initially considered to derive some important results on periodic task scheduling, then such results are extended to deal with more realistic cases, in which assumptions A3 and A4 are relaxed. In particular, the problem of scheduling a set of tasks under resource constraints is considered in detail in Chapter 7.

In those cases in which the assumptions A1, A2, A3, and A4 hold, a periodic task τ_i can be completely characterized by the following three parameters: its phase Φ_i, its period T_i and its worst-case computation time C_i. Thus, a set of periodic tasks can be denoted by

$$\Gamma = \{\tau_i(\Phi_i, T_i, C_i),\ i = 1, \ldots, n\}.$$

The release time $r_{i,k}$ and the absolute deadline $d_{i,k}$ of the generic kth instance can then be computed as

$$\begin{aligned} r_{i,k} &= \Phi_i + (k-1)T_i \\ d_{i,k} &= r_{i,k} + T_i = \Phi_i + kT_i. \end{aligned}$$

Other parameters that are typically defined on a periodic task are described below.

- **Response time** of an instance. It is the time (measured from the release time) at which the instance is terminated:

$$R_{i,k} = f_{i,k} - r_{i,k}.$$

- **Critical instant** of a task. It is the time at which the release of a task will produce the largest response time.

- **Critical time zone** of a task. It is the interval between the critical instant and the response time of the corresponding request of the task.

- **Relative Release Jitter** of a task. It is the maximum deviation of the start time of two consecutive instances:

$$RRJ_i = \max_k |(s_{i,k} - r_{i,k}) - (s_{i,k-1} - r_{i,k-1})|.$$

- **Absolute Release Jitter** of a task. It is the maximum deviation of the start time among all instances:

$$ARJ_i = \max_k (s_{i,k} - r_{i,k}) - \min_k (s_{i,k} - r_{i,k}).$$

- **Relative Finishing Jitter** of a task. It is the maximum deviation of the finishing time of two consecutive instances:

$$RFJ_i = \max_k |(f_{i,k} - r_{i,k}) - (f_{i,k-1} - r_{i,k-1})|.$$

- **Absolute Finishing Jitter** of a task. It is the maximum deviation of the finishing time among all instances:

$$AFJ_i = \max_k (f_{i,k} - r_{i,k}) - \min_k (f_{i,k} - r_{i,k}).$$

In this context, a periodic task τ_i is said to be *feasible* if all its instances finish within their deadlines. A task set Γ is said to be *schedulable* (or *feasible*) if all tasks in Γ are feasible.

4.1.1 Processor utilization factor

Given a set Γ of n periodic tasks, the *processor utilization factor* U is the fraction of processor time spent in the execution of the task set [LL73]. Since C_i/T_i is the fraction of processor time spent in executing task τ_i, the utilization factor for n tasks is given by

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}.$$

The processor utilization factor provides a measure of the computational load on the CPU due to the periodic task set. Although the CPU utilization can be improved by increasing tasks' computation times or by decreasing their periods, there exists a maximum value of U below which Γ is schedulable and above which Γ is not schedulable. Such a limit depends on the task set (that is, on the particular relations among tasks' periods) and on the algorithm used to schedule the tasks. Let $U_{ub}(\Gamma, A)$ be the upper bound of the processor utilization factor for a task set Γ under a given algorithm A.

When $U = U_{ub}(\Gamma, A)$, the set Γ is said to *fully utilize* the processor. In this situation, Γ is schedulable by A, but an increase in the computation time in any of the tasks will make the set infeasible. For a given algorithm A, the *least upper bound* $U_{lub}(A)$ of the processor utilization factor is the minimum of the utilization factors over all task sets that fully utilize the processor:

$$U_{lub}(A) = \min_{\Gamma} U_{ub}(\Gamma, A).$$

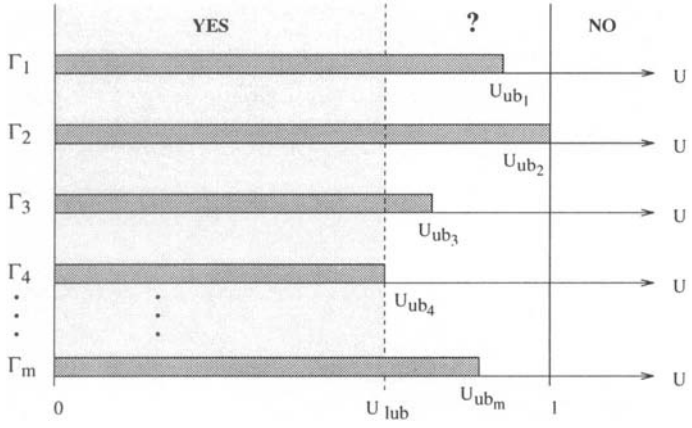

Figure 4.1 Meaning of the least upper bound of the processor utilization factor.

Figure 4.1 graphically illustrates the meaning of U_{lub} for a scheduling algorithm A. The task sets Γ_i shown in the figure differ for the number of tasks and for the configuration of their periods. When scheduled by the algorithm A, each task set Γ_i fully utilizes the processor when its utilization factor U_i (varied by changing tasks' computation times) reaches a particular upper bound U_{ub_i}. If $U_i \leq U_{ub_i}$, then Γ_i is schedulable, else Γ_i is not schedulable. Notice that each task set may have a different upper bound. Since $U_{lub}(A)$ is the minimum of all upper bounds, any task set having a processor utilization factor below $U_{lub}(A)$ is certainly schedulable by A.

U_{lub} defines an important characteristic of a scheduling algorithm because it allows to easily verify the schedulability of a task set. In fact, any task set whose processor utilization factor is below this bound is schedulable by the algorithm. On the other hand, utilization above this bound can be achieved only if the periods of the tasks are suitably related.

If the utilization factor of a task set is greater than one, the task set cannot be scheduled by any algorithm. To show this result, let T be the product of all the periods: $T = T_1 T_2 \ldots T_n$. If $U > 1$, we also have $UT > T$, which can be written as

$$\sum_{i=1}^{n} \frac{T}{T_i} C_i > T.$$

The factor (T/T_i) represents the number of times that τ_i is executed in the interval T, whereas the quantity $(T/T_i)C_i$ is the total computation time requested by τ_i in the interval T. Hence, the sum on the left hand side represents the total demand of computation time requested by all tasks in T. Clearly, if the total demand exceeds the available processor time, there is no feasible schedule for the task set.

4.2 RATE MONOTONIC SCHEDULING

The Rate Monotonic (RM) scheduling algorithm is a simple rule that assigns priorities to tasks according to their request rates. Specifically, tasks with higher request rates (that is, with shorter periods) will have higher priorities. Since periods are constant, RM is a fixed-priority assignment: priorities are assigned to tasks before execution and do not change over time. Moreover, RM is intrinsically preemptive: the currently executing task is preempted by a newly arrived task with shorter period.

In 1973, Liu and Layland [LL73] showed that RM is optimal among all fixed-priority assignments in the sense that no other fixed-priority algorithms can schedule a task set that cannot be scheduled by RM. Liu and Layland also derived the least upper bound of the processor utilization factor for a generic set of n periodic tasks. These issues are discussed in detail in the following subsections.

4.2.1 Optimality

In order to prove the optimality of the RM algorithm, we first show that a critical instant for any task occurs whenever the task is released simultaneously with all higher-priority tasks. Let $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ be the set of periodic tasks ordered by increasing periods, with τ_n being the task with the longest period. According to RM, τ_n will also be the task with the lowest priority.

As shown in Figure 4.2a, the response time of task τ_n is delayed by the interference of τ_i with higher priority. Moreover, from Figure 4.2b it is clear that advancing the release time of τ_i may increase the completion time of τ_n. As a consequence, the response time of τ_n is largest when it is released simultaneously with τ_i. Repeating the argument for all τ_i, $i = 2, \ldots, n-1$, we prove

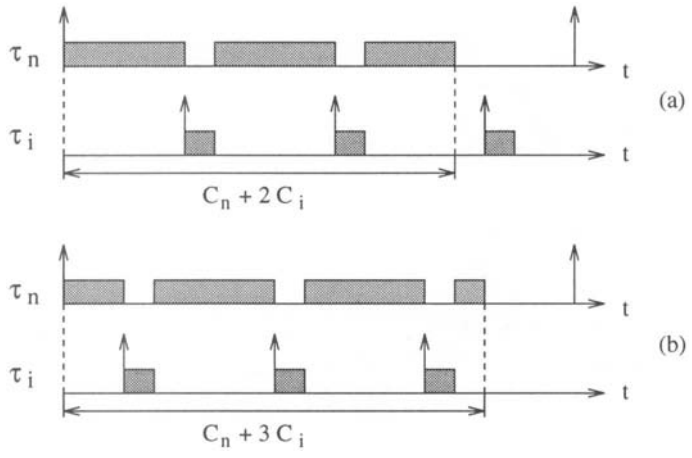

Figure 4.2 **a.** The response time of task τ_n is delayed by the interference of τ_i with higher priority. **b.** The interference may increase advancing the release time of τ_i.

that the worst response time of a task occurs when it is released simultaneously with all higher-priority tasks.

A first consequence of this result is that task schedulability can easily be checked at their critical instants. Specifically, if all tasks are feasible at their critical instants, then the task set is schedulable in any other condition. Based on this result, the optimality of RM is justified by showing that if a task set is schedulable by an arbitrary priority assignment, then it is also schedulable by RM.

Consider a set of two periodic tasks τ_1 and τ_2, with $T_1 < T_2$. If priorities are not assigned according to RM, then task T_2 will receive the highest priority. This situation is depicted in Figure 4.3, from which it is easy to see that, at critical instants, the schedule is feasible if the following inequality is satisfied:

$$C_1 + C_2 \leq T_1. \tag{4.1}$$

On the other hand, if priorities are assigned according to RM, task T_1 will receive the highest priority. In this situation, illustrated in Figure 4.4, in order to guarantee a feasible schedule two cases must be considered. Let $F = \lfloor T_2/T_1 \rfloor$ be the number[1] of periods of τ_1 entirely contained in T_2.

[1] $\lfloor x \rfloor$ denotes the largest integer smaller than or equal to x, whereas $\lceil x \rceil$ denotes the smallest integer greater than or equal to x.

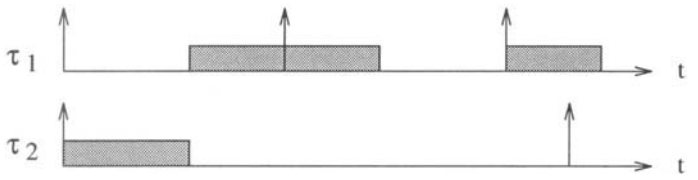

Figure 4.3 Tasks scheduled by an algorithm different from RM.

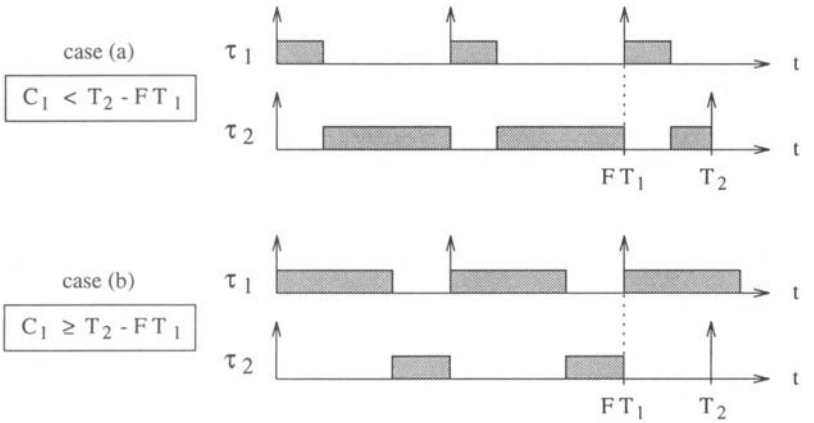

Figure 4.4 Schedule produced by RM in two different conditions.

Case 1. The computation time C_1 is short enough that all requests of τ_1 within the critical time zone of τ_2 are completed before the second request of τ_2. That is, $C_1 \leq T_2 - FT_1$.

In this case, from Figure 4.4a we can see that the task set is schedulable if

$$(F + 1)C_1 + C_2 \leq T_2. \tag{4.2}$$

We now show that inequality (4.1) implies (4.2). In fact, by multiplying both sides of (4.1) by F we obtain

$$FC_1 + FC_2 \leq FT_1,$$

and, since $F \geq 1$, we can write

$$FC_1 + C_2 \leq FC_1 + FC_2 \leq FT_1.$$

Adding C_1 to each member we get

$$(F+1)C_1 + C_2 \leq FT_1 + C_1.$$

Since we assumed that $C_1 \leq T_2 - FT_1$, we have

$$(F+1)C_1 + C_2 \leq FT_1 + C_1 \leq T_2,$$

which satisfies (4.2).

Case 2. The execution of the last request of τ_1 in the critical time zone of τ_2 overlaps the second request of τ_2. That is, $C_1 \geq T_2 - FT_1$.

In this case, from Figure 4.4b we can see that the task set is schedulable if

$$FC_1 + C_2 \leq FT_1. \tag{4.3}$$

Again, inequality (4.1) implies (4.3). In fact, by multiplying both sides of (4.1) by F we obtain

$$FC_1 + FC_2 \leq FT_1,$$

and, since $F \geq 1$, we can write

$$FC_1 + C_2 \leq FC_1 + FC_2 \leq FT_1,$$

which satisfies (4.3).

Basically, it has been shown that, given two periodic tasks τ_1 and τ_2, with $T_1 < T_2$, if the schedule is feasible by an arbitrary priority assignment, then it is also feasible by RM. That is, RM is optimal. This result can easily be extended to a set of n periodic tasks. We now show how to compute the least upper bound U_{lub} of the processor utilization factor for the RM algorithm. The bound is first determined for two tasks and then extended for an arbitrary number of tasks.

4.2.2 Calculation of U_{lub} for two tasks

Consider a set of two periodic tasks τ_1 and τ_2, with $T_1 < T_2$. In order to compute U_{lub} for RM, we have to

- Assign priorities to tasks according to RM, so that τ_1 is the task with the highest priority;

- Compute the upper bound U_{ub} for the set by setting tasks' computation times to fully utilize the processor;
- Minimize the upper bound U_{ub} with respect to all other task parameters.

As before, let $F = \lfloor T_2/T_1 \rfloor$ be the number of periods of τ_1 entirely contained in T_2. Without loss of generality, the computation time C_2 is adjusted to fully utilize the processor. Again two cases must be considered.

Case 1. The computation time C_1 is short enough that all requests of τ_1 within the critical time zone of τ_2 are completed before the second request of τ_2. That is, $C_1 \leq T_2 - FT_1$.

In this situation, depicted in Figure 4.5, the largest possible value of C_2 is

$$C_2 = T_2 - C_1(F+1),$$

and the corresponding upper bound U_{ub} is

$$\begin{aligned} U_{ub} &= \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{C_1}{T_1} + \frac{T_2 - C_1(F+1)}{T_2} = \\ &= 1 + \frac{C_1}{T_1} - \frac{C_1}{T_2}(F+1) = \\ &= 1 + \frac{C_1}{T_2}\left[\frac{T_2}{T_1} - (F+1)\right]. \end{aligned}$$

Since the quantity in square brackets is negative, U_{ub} is monotonically decreasing in C_1, and, being $C_1 \leq T_2 - FT_1$, the minimum of U_{ub} occurs for

$$C_1 = T_2 - FT_1.$$

Case 2. The execution of the last request of τ_1 in the critical time zone of τ_2 overlaps the second request of τ_2. That is, $C_1 \geq T_2 - FT_1$.

In this situation, depicted in Figure 4.6, the largest possible value of C_2 is

$$C_2 = (T_1 - C_1)F,$$

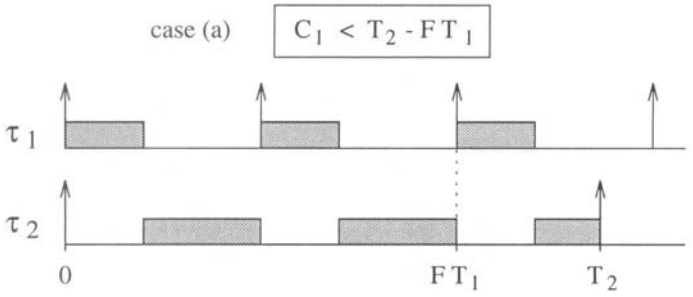

Figure 4.5 The second request of τ_2 is released when τ_1 is idle.

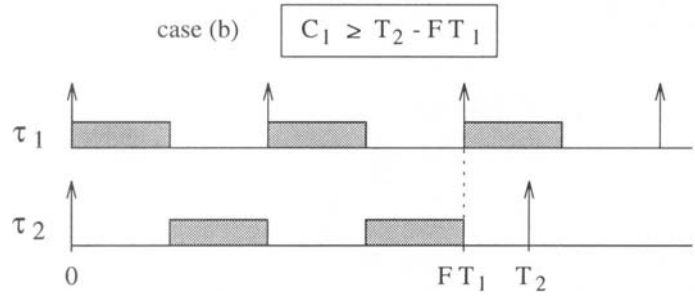

Figure 4.6 The second request of τ_2 is released when τ_1 is active.

and the corresponding upper bound U_{ub} is

$$\begin{aligned} U_{ub} &= \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{C_1}{T_1} + \frac{(T_1 - C_1)F}{T_2} = \\ &= \frac{T_1}{T_2}F + \frac{C_1}{T_1} - \frac{C_1}{T_2}F = \\ &= \frac{T_1}{T_2}F + \frac{C_1}{T_2}\left[\frac{T_2}{T_1} - F\right]. \end{aligned} \tag{4.4}$$

Since the quantity in square brackets is positive, U_{ub} is monotonically increasing in C_1 and, being $C_1 \geq T_2 - FT_1$, the minimum of U_{ub} occurs for

$$C_1 = T_2 - FT_1.$$

In both cases, the minimum value of U_{ub} occurs for

$$C_1 = T_2 - T_1F.$$

Hence, using the minimum value of C_1, from equation (4.4) we have

$$\begin{aligned} U &= \frac{T_1}{T_2}F + \frac{C_1}{T_2}\left(\frac{T_2}{T_1} - F\right) = \\ &= \frac{T_1}{T_2}F + \frac{(T_2 - T_1F)}{T_2}\left(\frac{T_2}{T_1} - F\right) = \\ &= \frac{T_1}{T_2}\left[F + \left(\frac{T_2}{T_1} - F\right)\left(\frac{T_2}{T_1} - F\right)\right]. \end{aligned} \tag{4.5}$$

To simplify the notation, let $G = T_2/T_1 - F$. Thus,

$$\begin{aligned} U &= \frac{T_1}{T_2}(F + G^2) = \frac{(F + G^2)}{T_2/T_1} = \\ &= \frac{(F + G^2)}{(T_2/T_1 - F) + F} = \frac{F + G^2}{F + G} = \\ &= \frac{(F + G) - (G - G^2)}{F + G} = 1 - \frac{G(1 - G)}{F + G}. \end{aligned} \tag{4.6}$$

Since $0 \leq G < 1$, the term $G(1 - G)$ is nonnegative. Hence, U is monotonically increasing with F. As a consequence, the minimum of U occurs for the minimum value of F; namely, $F = 1$. Thus,

$$U = \frac{1 + G^2}{1 + G}. \tag{4.7}$$

Minimizing U over G we have

$$\begin{aligned} \frac{dU}{dG} &= \frac{2G(1 + G) - (1 + G^2)}{(1 + G)^2} = \\ &= \frac{G^2 + 2G - 1}{(1 + G)^2}, \end{aligned}$$

and $dU/dG = 0$ for $G^2 + 2G - 1 = 0$, which has two solutions:

$$\begin{cases} G_1 = -1 - \sqrt{2} \\ G_2 = -1 + \sqrt{2}. \end{cases}$$

Since $0 \leq G < 1$, the negative solution $G = G_1$ is discarded. Thus, from equation (4.7), the least upper bound of U is given for $G = G_2$:

$$U_{lub} = \frac{1 + (\sqrt{2} - 1)^2}{1 + (\sqrt{2} - 1)} = \frac{4 - 2\sqrt{2}}{\sqrt{2}} = 2(\sqrt{2} - 1).$$

F	k^*	U^*
1	$\sqrt{2}$	0.828
2	$\sqrt{6}$	0.899
3	$\sqrt{12}$	0.928
4	$\sqrt{20}$	0.944
5	$\sqrt{30}$	0.954

Table 4.1 Values of k_i^* and U_i^* as a function of F.

That is,

$$U_{lub} = 2(2^{1/2} - 1) \simeq 0.83. \tag{4.8}$$

Notice that if T_2 is a multiple of T_1, $G = 0$ and the processor utilization factor becomes 1. In general, the utilization factor for two tasks can be computed as a function of the ratio $k = T_2/T_1$. For a given F, from equation (4.5) we can write

$$U = \frac{F + (k - F)^2}{k} = k - 2F + \frac{F(F+1)}{k}.$$

Minimizing U over k we have

$$\frac{dU}{dk} = 1 - \frac{F(F+1)}{k^2},$$

and $dU/dk = 0$ for $k^* = \sqrt{F(F+1)}$. Hence, for a given F, the minimum value of U is

$$U^* = 2(\sqrt{F(F+1)} - F).$$

Table 4.1 shows some values of k^* and U^* as a function of F, whereas Figure 4.7 shows the upper bound of U as a function of k.

4.2.3 Calculation of U_{lub} for n tasks

From the previous computation, the conditions that allow to compute the least upper bound of the processor utilization factor are

$$\begin{cases} F = 1 \\ C_1 = T_2 - FT_1 \\ C_2 = (T_1 - C_1)F, \end{cases}$$

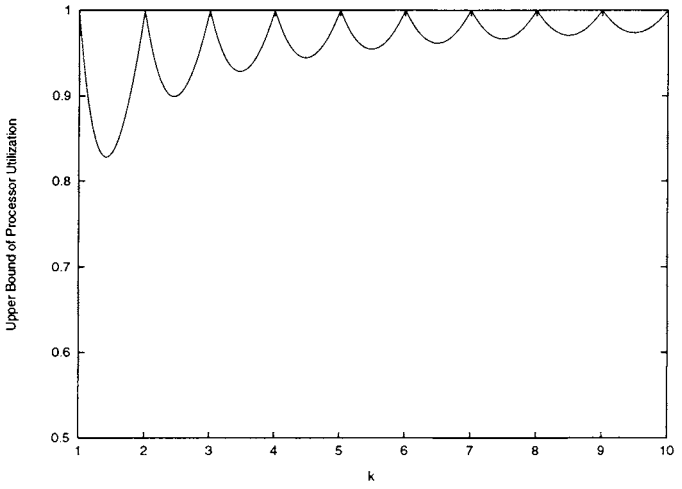

Figure 4.7 Upper bound of the processor utilization factor as a function of the ratio $k = T_2/T_1$.

which can be rewritten as

$$\begin{cases} T_1 < T_2 < 2T_1 \\ C_1 = T_2 - T_1 \\ C_2 = 2T_1 - T_2. \end{cases}$$

Generalizing for an arbitrary set of n tasks, the worst conditions for the schedulability of a task set that fully utilizes the processor are

$$\begin{cases} T_1 < T_n < 2T_1 \\ C_1 = T_2 - T_1 \\ C_2 = T_3 - T_2 \\ \ldots \\ C_{n-1} = T_n - T_{n-1} \\ C_n = T_1 - (C_1 + C_2 + \ldots + C_{n-1}) = 2T_1 - T_n. \end{cases}$$

Thus, the processor utilization factor becomes

$$U = \frac{T_2 - T_1}{T_1} + \frac{T_3 - T_2}{T_2} + \ldots + \frac{T_n - T_{n-1}}{T_{n-1}} + \frac{2T_1 - T_n}{T_n}.$$

Defining

$$R_i = \frac{T_{i+1}}{T_i}$$

and noting that

$$R_1 R_2 \ldots R_{n-1} = \frac{T_n}{T_1},$$

the utilization factor may be written as

$$U = \sum_{i=1}^{n-1} R_i + \frac{2}{R_1 R_2 \ldots R_{n-1}} - n.$$

To minimize U over R_i, $i = 1, \ldots, n-1$, we have

$$\frac{\partial U}{\partial R_k} = 1 - \frac{2}{R_i^2 (\prod_{i \neq k}^{n-1} R_i)}.$$

Thus, defining $P = R_1 R_2 \ldots R_{n-1}$, U is minimum when

$$\begin{cases} R_1 P = 2 \\ R_2 P = 2 \\ \ldots \\ R_{n-1} P = 2. \end{cases}$$

That is, when all R_i have the same value:

$$R_1 = R_2 = \ldots = R_{n-1} = 2^{1/n}.$$

Substituting this value in U we obtain

$$\begin{aligned} U_{lub} &= (n-1)2^{1/n} + \frac{2}{2^{(1-1/n)}} - n = \\ &= n2^{1/n} - 2^{1/n} + 2^{1/n} - n = \\ &= n(2^{1/n} - 1). \end{aligned}$$

Therefore, for an arbitrary set of periodic tasks, the least upper bound of the processor utilization factor under the Rate-Monotonic scheduling algorithm is

$$U_{lub} = n(2^{1/n} - 1). \tag{4.9}$$

This bound decreases with n, and values for some n are shown in Table 4.2.

For high values of n, the least upper bound converges to

$$U_{lub} = \ln 2 \simeq 0.69.$$

In fact, with the substitution $y = (2^{1/n} - 1)$, we obtain $n = \frac{\ln 2}{\ln(y+1)}$, and hence

$$\lim_{n \to \infty} n(2^{1/n} - 1) = (\ln 2) \lim_{y \to 0} \frac{y}{\ln(y+1)}$$

n	U_{lub}
1	1.000
2	0.828
3	0.780
4	0.757
5	0.743

n	U_{lub}
6	0.735
7	0.729
8	0.724
9	0.721
10	0.718

Table 4.2 Values of U_{lub} as a function of n.

and since (by the Hospital's rule)

$$\lim_{y \to 0} \frac{y}{\ln(y+1)} = \lim_{y \to 0} \frac{1}{1/(y+1)} = \lim_{y \to 0} (y+1) = 1,$$

we have that

$$\lim_{n \to \infty} U_{lub}(n) = \ln 2.$$

4.2.4 Concluding remarks on RM

To summarize the most important results derived in this section, the Rate-Monotonic algorithm has been proved to be optimal among all fixed-priority assignments, in the sense that no other fixed-priority algorithms can schedule a task set that cannot be scheduled by RM. Moreover, RM guarantees that an arbitrary set of periodic tasks is schedulable if the total processor utilization U does not exceed a value of 0.69.

Notice that this schedulability condition is sufficient to guarantee the feasibility of any task set, but it is not necessary. This means that, if a task set has an utilization factor greater than U_{lub} and less than one, nothing can be said on the feasibility of the set. A sufficient and necessary condition for the schedulability under RM has been derived by Audsley et al. [ABRW91] for the more general case of periodic tasks with relative deadlines less than periods, and it is presented in Section 4.4.

A simulation study carried out by Lehoczky, Sha, and Ding [LSD89] showed that for random task sets the processor utilization bound is approximately 0.88. However, since RM is optimal among all static assignments, an improvement of the processor utilization bound can be achieved only by using dynamic scheduling algorithms.

4.3 EARLIEST DEADLINE FIRST

The Earliest Deadline First (EDF) algorithm is a dynamic scheduling rule that selects tasks according to their absolute deadlines. Specifically, tasks with earlier deadlines will be executed at higher priorities. Since the absolute deadline of a periodic task depends on the current jth instance as

$$d_{i,j} = \Phi_i + (j-1)T_i + D_i,$$

EDF is a dynamic priority assignment. Moreover, it is intrinsically preemptive: the currently executing task is preempted whenever another periodic instance with earlier deadline becomes active.

Notice that EDF does not make any specific assumption on the periodicity of the tasks; hence, it can be used for scheduling periodic as well as aperiodic tasks. For the same reason, the optimality of EDF, proved in Chapter 3 for aperiodic tasks, also holds for periodic tasks.

4.3.1 Schedulability analysis

Under the assumptions A1, A2, A3, and A4, the schedulability of a periodic task set handled by EDF can be verified through the processor utilization factor. In this case, however, the least upper bound is one; therefore, tasks may utilize the processor up to 100% and still be schedulable. In particular, the following theorem holds [LL73, SBS95]:

Theorem 4.1 *A set of periodic tasks is schedulable with EDF if and only if*

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1.$$

Proof. *Only if.* We show that a task set cannot be scheduled if $U > 1$. In fact, by defining $T = T_1 T_2 \ldots T_n$, the total demand of computation time requested by all tasks in T can be calculated as

$$\sum_{i=1}^{n} \frac{T}{T_i} C_i = UT.$$

If $U > 1$ – that is, if the total demand UT exceeds the available processor time T – there is clearly no feasible schedule for the task set.

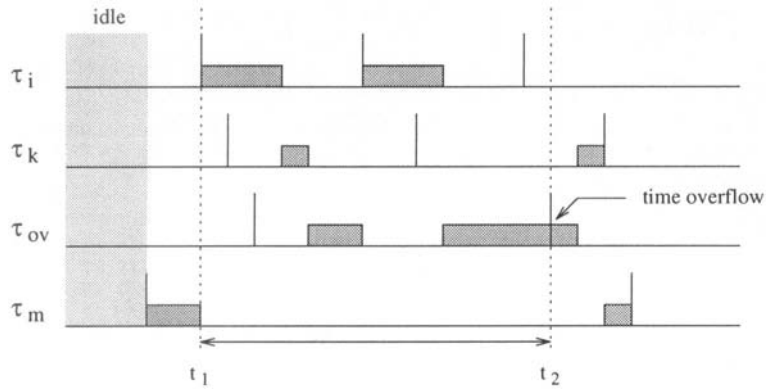

Figure 4.8 Interval of continuous utilization in an EDF schedule before a time-overflow.

If. We show the sufficiency by contradiction. Assume that the condition $U < 1$ is satisfied and yet the task set is not schedulable. Let t_2 be the instant at which the time-overflow occurs and let $[t_1, t_2]$ be the longest interval of continuous utilization, before the overflow, such that only instances with deadline less than or equal to t_2 are executed in $[t_1, t_2]$ (see Figure 4.8 for explanation). Note that t_1 must be the release time of some periodic instance. Let $C_p(t_1, t_2)$ be the total computation time demanded by periodic tasks in $[t_1, t_2]$, which can be computed as

$$C_p(t_1, t_2) = \sum_{r_k \geq t_1, d_k \leq t_2} C_k = \sum_{i=1}^{n} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i. \tag{4.10}$$

Now, observe that

$$C_p(t_1, t_2) = \sum_{i=1}^{n} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i \leq \sum_{i=1}^{n} \frac{t_2 - t_1}{T_i} C_i = (t_2 - t_1)U.$$

Since a deadline is missed at t_2, $C_p(t_1, t_2)$ must be greater than the available processor time $(t_2 - t_1)$; thus, we must have

$$(t_2 - t_1) < C_p(t_1, t_2) \leq (t_2 - t_1)U.$$

That is, $U > 1$, which is a contradiction. □

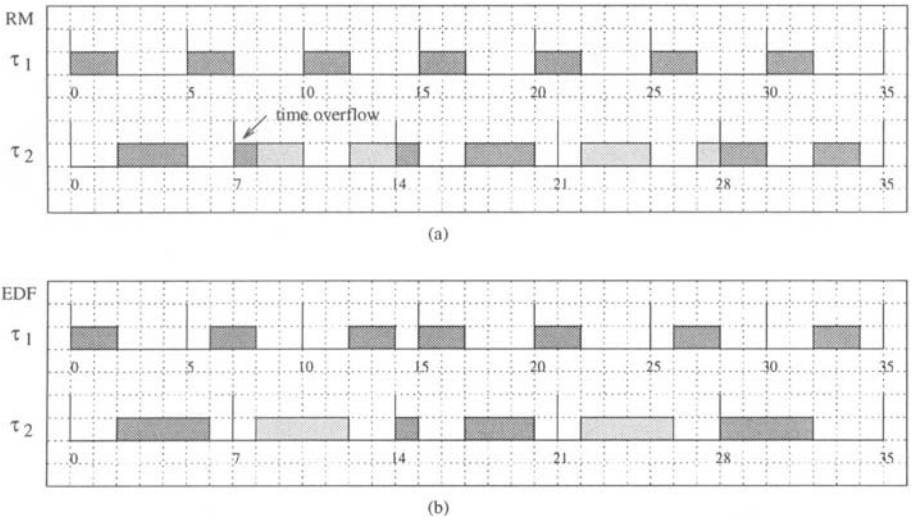

Figure 4.9 Schedule produced by RM (a) and EDF (b) on the same set of periodic tasks.

4.3.2 An example

Consider the periodic task set illustrated in Figure 4.9, for which the processor utilization factor is

$$U = \frac{2}{5} + \frac{4}{7} = \frac{34}{35} \simeq 0.97.$$

This means that 97% of the processor time is used to execute the periodic tasks, whereas the CPU is idle in the remaining 3%. Being $U > \ln 2$, the schedulability of the task set cannot be guaranteed under RM, whereas it is guaranteed under EDF. Indeed, as shown in Figure 4.9a, RM generates a time-overflow at time $t = 7$, whereas EDF completes all tasks within their deadlines (see Figure 4.9b). Another important difference between RM and EDF concerns the number of preemptions occurring in the schedule. As shown in Figure 4.9, under RM every instance of task τ_2 is preempted, for a total number of five preemptions in the interval $T = T_1 T_2$. Under EDF, the same task is preempted only once in T. The small number of preemptions in EDF is a direct consequence of the dynamic priority assignment, which at any instant privileges the task with the earliest deadline, independently of tasks' periods.

4.4 DEADLINE MONOTONIC

The algorithms and the schedulability bounds illustrated in the previous sections rely on the assumptions A1, A2, A3, and A4 presented at the beginning of this chapter. In particular, assumption A3 imposes a relative deadline equal to the period, allowing an instance to be executed anywhere within its period. This condition could not always be desired in real-time applications. For example, relaxing assumption A3 would provide a more flexible process model, which could be adopted to handle tasks with jitter constraints or activities with short response times compared to their periods.

The Deadline Monotonic (DM) priority assignment weakens the "period equals deadline" constraint within a static priority scheduling scheme. This algorithm was first proposed in 1982 by Leung and Whitehead [LW82] as an extension of Rate Monotonic where tasks can have a relative deadline less than their period. Specifically, each periodic task τ_i is characterized by four parameters:

- A phase Φ_i;
- A worst-case computation time C_i (constant for each instance);
- A relative deadline D_i (constant for each instance);
- A period T_i.

These parameters are illustrated in Figure 4.10 and have the following relationships:

$$\begin{cases} C_i \leq D_i \leq T_i \\ r_{i,k} = \Phi_i + (k-1)T_i \\ d_{i,k} = r_{i,k} + D_i. \end{cases}$$

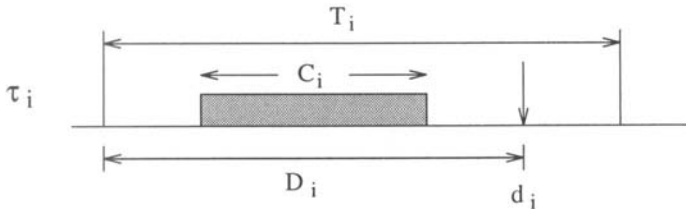

Figure 4.10 Task parameters in Deadline-Monotonic scheduling.

According to the DM algorithm, each task is assigned a priority inversely proportional to its relative deadline. Thus, at any instant, the task with the shortest relative deadline is executed. Since relative deadlines are constant, DM is a static priority assignment. As RM, DM is preemptive; that is, the currently executing task is preempted by a newly arrived task with shorter relative deadline.

The Deadline-Monotonic priority assignment is optimal,[2] meaning that if any static priority scheduling algorithm can schedule a set of tasks with deadlines unequal to their periods, then DM will also schedule that task set.

4.4.1 Schedulability analysis

The feasibility of a set of tasks with deadlines unequal to their periods could be guaranteed using the Rate-Monotonic schedulability test, by reducing tasks' periods to relative deadlines:

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{1/n} - 1).$$

However, such a test would not be optimal as the workload on the processor would be overestimated. A less pessimistic schedulability test can be derived by noting that

- The worst-case processor demand occurs when all tasks are released simultaneously; that is, at their critical instants;
- For each task τ_i, the sum of its processing time and the interference (preemption) imposed by higher priority tasks must be less than or equal to D_i.

Assuming that tasks are ordered by increasing relative deadlines, so that

$$i < j \iff D_i < D_j,$$

such a test is given by

$$\forall i : 1 \leq i \leq n \qquad C_i + I_i \leq D_i, \tag{4.11}$$

[2]The proof of DM optimality is similar to the one done for RM and it can be found in [LW82].

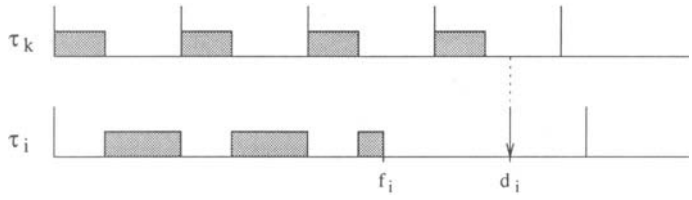

Figure 4.11 More accurate calculation of the interference on τ_i by higher priority tasks.

where I_i is a measure of the interference on τ_i, which can be computed as the sum of the processing times of all higher-priority tasks released before D_i:

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{D_i}{T_j} \right\rceil C_j.$$

Notice that this test is sufficient but not necessary for guaranteeing the schedulability of the task set. This is due to the fact that I_i is calculated by assuming that each higher-priority task τ_j exactly interferes $\lceil \frac{D_i}{T_j} \rceil$ times on τ_i. However, as shown in Figure 4.11, the actual interference can be smaller than I_i, since τ_i may terminate earlier.

To find a sufficient and necessary schedulability test for DM, the exact interleaving of higher-priority tasks must be evaluated for each process. In general, this procedure is quite costly since, for each task τ_i, it requires the construction of the schedule until D_i. Audsley et al. [ABRW92, ABR+93] proposed an efficient method for evaluating the exact interference on periodic tasks and derived a sufficient and necessary schedulability test for DM.

4.4.2 Sufficient and necessary schedulability test

According to the method proposed by Audsley at al., the longest response time R_i of a periodic task τ_i is computed, at the critical instant, as the sum of its computation time and the interference due to preemption by higher-priority tasks:

$$R_i = C_i + I_i,$$

where

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j.$$

Hence,

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \tag{4.12}$$

No simple solution exists for this equation since R_i appears on both sides. Thus, the worst-case response time of task τ_i is given by the smallest value of R_i that satisfies equation (4.12). Notice, however, that only a subset of points in the interval $[0, D_i]$ need to be examined for feasibility. In fact, the interference on τ_i only increases when there is a release of a higher-priority task.

To simplify the notation, let R_i^k be the kth estimate of R_i and let I_i^k be the interference on task τ_i in the interval $[0, R_i^k]$:

$$I_i^k = \sum_{j=1}^{i-1} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j. \tag{4.13}$$

Then the calculation of R_i is performed as follows:

1. Iteration starts with $R_i^0 = C_i$, which is the first point in time that τ_i could possibly complete.
2. The actual interference I_i^k in the interval $[0, R_i^k]$ is computed by equation (4.13).
3. If $I_i^k + C_i = R_i^k$, then R_i^k is the actual worst-case response time of task τ_i; that is, $R_i = R_i^k$. Otherwise, the next estimate is given by

   $$R_i^{k+1} = I_i^k + C_i,$$

   and the iteration continues from step 2.

Once R_i is calculated, the feasibility of task τ_i is guaranteed if and only if $R_i \leq D_i$.

To clarify the schedulability test, consider the set of periodic tasks shown in Table 4.3, simultaneously activated at time $t = 0$. In order to guarantee τ_4, we have to calculate R_4 and verify that $R_4 \leq D_4$. The schedule produced by DM is illustrated in Figure 4.12, and the iteration steps are shown below.

	C_i	T_i	D_i
τ_1	1	4	3
τ_2	1	5	4
τ_3	2	6	5
τ_4	1	11	10

Table 4.3 A set of periodic tasks with deadlines less than periods.

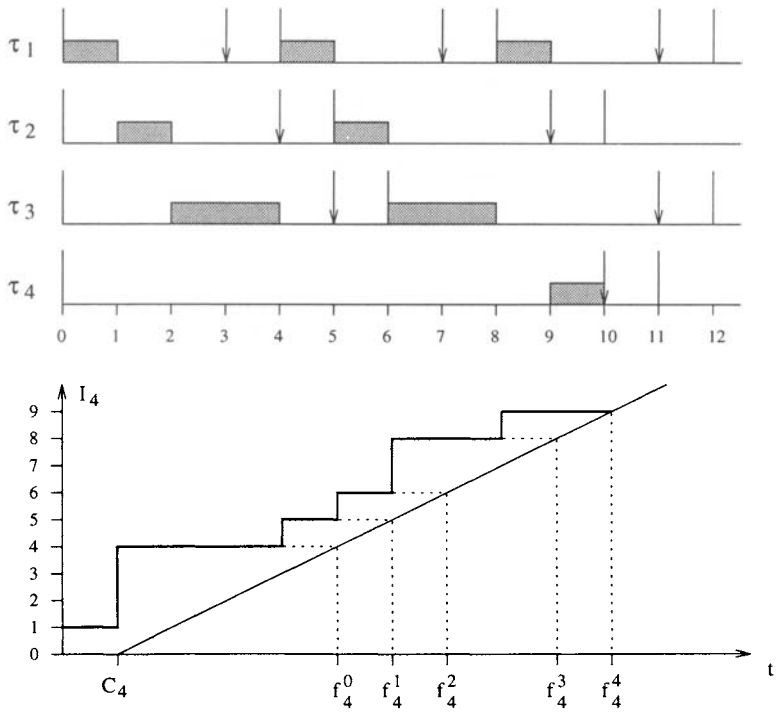

Figure 4.12 Example of schedule produced by DM.

Step 0: $R_4^0 = C_4 = 1$, but $I_4^0 = 4$ and $I_4^0 + C_4 > R_4^0$, hence τ_4 does not finish at R_4^0.

Step 1: $R_4^1 = I_4^0 + C_4 = 5$, but $I_4^1 = 5$ and $I_4^1 + C_4 > R_4^1$ hence τ_4 does not finish at R_4^1.

Step 2: $R_4^2 = I_4^1 + C_4 = 6$, but $I_4^2 = 6$ and $I_4^2 + C_4 > R_4^2$ hence τ_4 does not finish at R_4^2.

Step 3: $R_4^3 = I_4^2 + C_4 = 7$, but $I_4^3 = 7$ and $I_4^3 + C_4 > R_4^3$ hence τ_4 does not finish at R_4^3.

Step 4: $R_4^4 = I_4^3 + C_4 = 9$, but $I_4^4 = 9$ and $I_4^4 + C_4 > R_4^4$ hence τ_4 does not finish at R_4^4.

Step 5: $R_4^5 = I_4^4 + C_4 = 10$, but $I_4^5 = 9$ and $I_4^5 + C_4 = R_5^4$ hence τ_4 finishes at $R_4 = 10$.

Since $R_4 \leq D_4$, τ_4 is schedulable within its deadline. If $R_i \leq D_i$ for all tasks, we conclude that the task set is schedulable by DM. Such a schedulability test can be performed by the algorithm illustrated in Figure 4.13.

```
DM_guarantee (Γ) {
    for (each τ_i ∈ Γ) {
        I = 0;
        do {
            R = I + C_i;
            if (R > D_i) return(UNSCHEDULABLE);
            I = Σ_{j=1}^{i-1} ⌈R/T_j⌉ C_j;
        } while (I + C_i > R);
    }
    return(SCHEDULABLE);
}
```

Figure 4.13 Algorithm for testing the schedulability of a periodic task set Γ under Deadline Monotonic.

4.5 EDF WITH DEADLINES LESS THAN PERIODS

Under EDF, the analysis of periodic tasks with deadlines less than periods can be performed using a *processor demand* criterion. This method has been described by Baruah, Rosier, and Howell in [BRH90] and later used by Jeffay and Stone [JS93] to account for interrupt handling costs under EDF. Here, we first illustrate this approach for the case of deadlines equal to periods and then extend it to more general task models.

4.5.1 The processor demand approach

In general, the processor demand of a task τ_i in any interval $[t, t + L]$ is the amount of processing time required by τ_i in $[t, t + L]$ that has to complete at or before $t + L$. In a deadline-based system, it is the processing time required in $[t, t + L]$ that has to be executed with deadlines less than or equal to $t + L$.

For a set of periodic tasks (with deadlines equal to periods) invoked at time $t = 0$ the cumulative processor demand in any interval $[0, L]$ is the total amount of processing time $C_P(0, L)$ that has to be executed with deadlines less than or equal to L. That is,

$$C_P(0, L) = \sum_{i=1}^{n} \left\lfloor \frac{L}{T_i} \right\rfloor C_i.$$

Given this definition, the schedulability of a periodic task set is guaranteed if and only if the cumulative processor demand in any interval $[0, L]$ is less than the available time; that is, the interval length L. This is stated by the following theorem:

Theorem 4.2 (Jeffay and Stone) *A set of periodic tasks is schedulable by EDF if and only if for all L, $L \geq 0$,*

$$L \geq \sum_{i=1}^{n} \left\lfloor \frac{L}{T_i} \right\rfloor C_i. \tag{4.14}$$

Proof. The theorem is proved by showing that equation (4.14) is equivalent to the classical Liu and Layland's condition

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1. \tag{4.15}$$

(4.15) $\Rightarrow$ (4.14). If $U \leq 1$, then for all L, $L \geq 0$,

$$L \geq UL = \sum_{i=1}^{n} \left(\frac{L}{T_i}\right) C_i \geq \sum_{i=1}^{n} \left\lfloor \frac{L}{T_i} \right\rfloor C_i.$$

To demonstrate (4.15) $\Leftarrow$ (4.14) we show that $\neg$(4.15) $\Rightarrow$ $\neg$(4.14). That is, we assume $U > 1$ and prove that there exist an $L \geq 0$ for which (4.14) does not hold. If $U > 1$, then for $L = lcm(T_1, \ldots, T_n)$,

$$L < LU = \sum_{i=1}^{n} \left(\frac{L}{T_i}\right) C_i = \sum_{i=1}^{n} \left\lfloor \frac{L}{T_i} \right\rfloor C_i.$$

□

Notice that to apply Theorem 4.2 it suffices to test equation (4.14) only for values of L equal to release times less than the hyperperiod H. In fact, if equation (4.14) holds for $L = r_k$, it will also hold for any $L \in [r_k, r_{k+1})$, since

$$\forall L \in [r_k, r_{k+1}), \quad \left\lfloor \frac{L}{T_i} \right\rfloor = \left\lfloor \frac{r_k}{T_i} \right\rfloor.$$

The values of L for which equation (4.14) has to be tested can still be reduced to the set of release times within the busy period. The *busy period* is the smallest interval $[0, L]$ in which the total processing time $W(L)$ requested in $[0, L]$ is completely executed. The quantity $W(L)$ can be computed as

$$W(L) = \sum_{i=1}^{n} \left\lceil \frac{L}{T_i} \right\rceil C_i. \tag{4.16}$$

Thus, the busy period B_p can be defined as

$$B_p = \min\{L \mid W(L) = L\}$$

and computed by the algorithm shown in Figure 4.14.

Notice that, when the system is overloaded, the processor is always busy and the busy period is equal to infinity. On the other hand, if the system is not

```
busy_period {
    L = Σ_{i=1}^{n} C_i;
    L' = W(L);
    H = lcm(T_1, ..., T_n);
    while (L' ≠ L) and (L' ≤ H) {
        L = L';
        L' = W(L);
    }
    if (L' ≤ H) B_p = L;
    else B_p = INFINITY;
}
```

Figure 4.14 Algorithm for computing the busy period.

overloaded, the busy period coincides either with the beginning of an idle time (see Figure 4.15a) or with the release of a periodic instance (see Figure 4.15b).

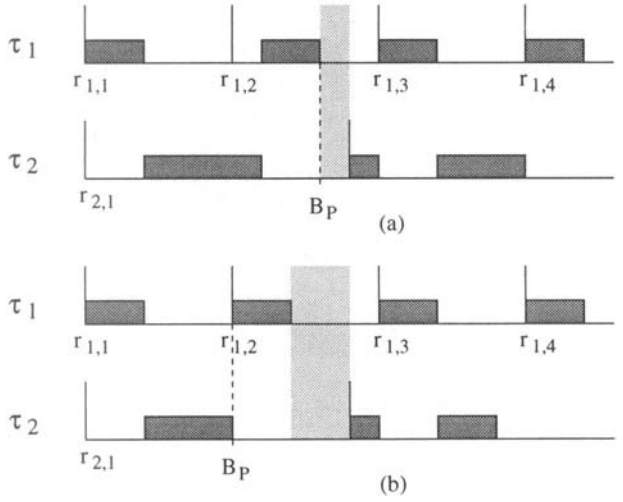

Figure 4.15 Examples of finite busy periods.

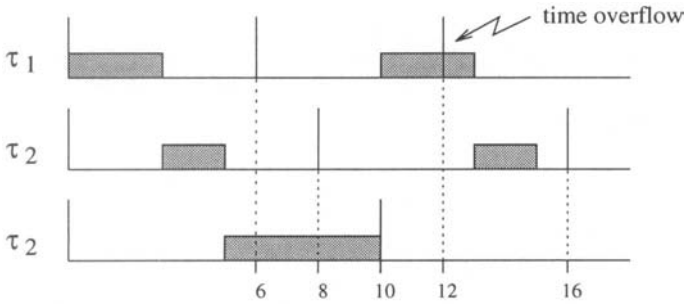

Figure 4.16 Examples of processor demand analysis.

L	$C_P(0, L)$	result
6	3	OK
8	5	OK
10	10	OK
12	13	**NO**

Table 4.4 Results of the processor demand criterion.

Based on the previous observations, to apply Theorem 4.2, equation (4.14) can be tested for all $L \in \mathcal{R}$, where

$$\mathcal{R} = \{r_{i,j} \mid r_{i,j} \leq \min(B_p, H),\ 1 \leq i \leq n,\ j \geq 1\}.$$

Example

To illustrate the processor demand criterion, consider the example shown in Figure 4.16, where three periodic tasks with periods 6, 8, 10, and processing times 3, 2, 5, respectively, are executed under EDF. In this case, the set checking points for equation (4.14) is given by $\mathcal{R} = \{6, 8, 10, 12, 16, \ldots\}$. Applying Theorem 4.2 we have the results shown in Table 4.4.

4.5.2 Deadlines less than periods

The processor demand criterion can easily be extended to deal with tasks with deadlines different than periods. For example, consider the two tasks shown in Figure 4.17. In this case, the processor demands for tasks τ_1 and τ_2 in $[0, L]$ are clearly given by

$$\begin{cases} C_1(0,L) = \left\lfloor \frac{L}{T_1} \right\rfloor C_1 \\ C_2(0,L) = \left(\left\lfloor \frac{L}{T_2} \right\rfloor + 1 \right) C_2. \end{cases}$$

In general, we can write

$$C_i(0,L) = \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i. \tag{4.17}$$

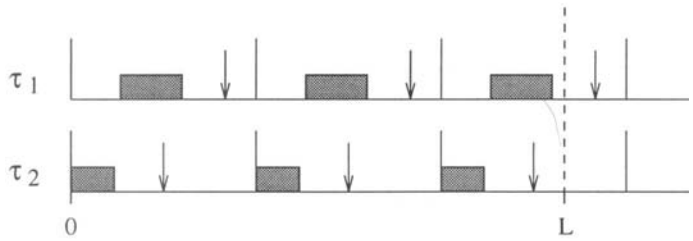

Figure 4.17 Processor demand when deadlines are less than periods.

In summary, the schedulability of a generic task set can be tested by the following theorem [BRH90], whose proof is very similar to the one shown for Theorem 4.2.

Theorem 4.3 *If* $\mathcal{D} = \{d_{i,k} \mid d_{i,k} = kT_i + D_i,\ d_{i,k} \leq \min(B_p, H),\ 1 \leq i \leq n,\ k \geq 0\}$, *then a set of periodic tasks with deadlines less than periods is schedulable by EDF if and only if*

$$\forall L \in \mathcal{D} \quad L \geq \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i. \tag{4.18}$$

4.6 SUMMARY

In conclusion, the problem of scheduling a set of independent and preemptable periodic tasks has been solved both under fixed and dynamic priority assignments. The Rate-Monotonic (RM) algorithm is optimal among all fixed-priority assignments, whereas the Eearliest Deadline First (EDF) algorithm is optimal among all dynamic priority assignments. When deadlines are equal to periods, the guarantee test for both algorithms can be performed in $O(n)$ (being n the number of periodic tasks in the set), using the processor utilization approach. The test for RM, however, provides only a sufficient condition for guaranteeing the feasibility of the schedule.

In the general case in which deadlines can be less or equal to periods, the schedulability analysis becomes more complex and can be performed in pseudo-polynomial time [BRH90]. Under fixed-priority assignments, the feasibility of the task set can be tested using the response time approach, which uses a recurrent formula to calculate the worst-case finishing time of any task. Under dynamic priority assignments, the feasibility can be tested using the processor demand approach. In both cases the test provides a necessary and sufficient condition. The various methods are summarized in Figure 4.18.

	$D_i = T_i$	$D_i \leq T_i$
Static priority	**RM** Processor utilization approach $U \leq n(2^{1/n} - 1)$	**DM** Response time approach $\forall i \quad R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \leq D_i$
Dynamic priority	**EDF** Processor utilization approach $U \leq 1$	**EDF** * Processor demand approach $\forall L > 0 \quad L \geq \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i$

Figure 4.18 Summary of guarantee tests for periodic tasks.

Exercises

4.1 Verify the schedulability and construct the schedule according to the RM algorithm for the following set of periodic tasks:

	τ_1	τ_2
C_i	1	1
T_i	3	4

4.2 Given the following set of periodic tasks

	τ_1	τ_2	τ_3
C_i	1	2	3
T_i	4	6	10

verify the schedulability under RM using the processor utilization approach. Then, perform the worst-case response time analysis and construct the schedule.

4.3 Verify the schedulability under RM and construct the schedule of the following task set:

	τ_1	τ_2	τ_3
C_i	1	2	3
T_i	4	6	8

4.4 Verify the schedulability under EDF of the task set shown in Exercise 4.3, and then construct the corresponding schedule.

4.5 Compute the busy period for the task set described in Exercise 4.2.

4.6 Compute the busy period for the task set described in Exercise 4.3.

4.7 Verify the schedulability under EDF and construct the schedule of the following task set:

	τ_1	τ_2	τ_3
C_i	2	2	4
D_i	5	4	8
T_i	6	8	12

4.8 Verify the schedulability of the task set described in Exercise 4.7 using the Deadline-Monotonic algorithm. Then construct the schedule.