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 11.

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, four basic algorithms for handling periodic tasks are described in detail: Timeline Scheduling, 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;

G.C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Real-Time Systems Series 24, DOI 10.1007/978-1-4614-0676-1_4,

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.

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.

- **Hyperperiod**. It is the minimum interval of time after which the schedule repeats itself. If H is the length of such an interval, then the schedule in $[0, H]$ is the same as that in $[kK, (k+1)K]$ for any integer $k > 0$. For a set of periodic tasks synchronously activated at time $t = 0$, the hyperperiod is given by the least common multiple of the periods:

$$H = lcm(T_1, \ldots, T_n).$$

- **Job response time**. It is the time (measured from the release time) at which the job is terminated:

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

- **Task response time**. It is the maximum response time among all the jobs:

$$R_i = \max_k R_{i,k}.$$

- **Critical instant** of a task. It is the arrival time that produces the largest task 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 Start Time 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 Start Time 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.

Figure 4.1 shows an example of two tasks (where τ_1 has higher priority than τ_2) in which $U_{ub} = 5/6 \simeq 0.833$. In fact, if any execution time is increased by epsilon, the

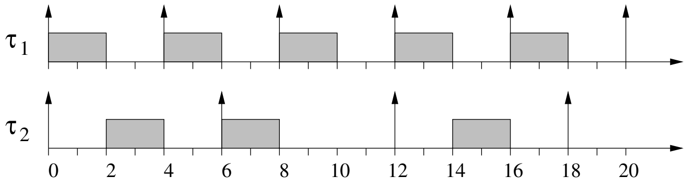

Figure 4.1 A task set with $U_{ub} = 5/6$.

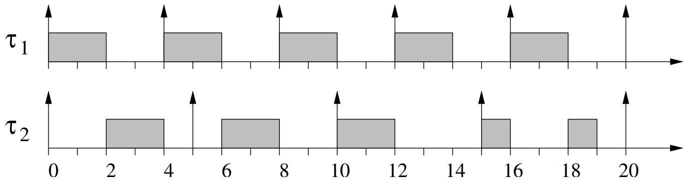

Figure 4.2 A task set with $U_{ub} = 0.9$.

task set becomes infeasible, since the first job of τ_2 misses its deadline. Figure 4.2 shows another example in which $U_{ub} = 0.9$. Notice that setting $T_1 = 4$ and $T_2 = 8$, U_{ub} becomes 1.0.

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 4.3 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 useful for easily verifying the schedulability of a task set. In fact, any task set whose processor utilization factor is less than or equal to this bound is schedulable by the algorithm. On the other hand, when $U_{lub} < U \leq 1.0$, the schedulability can be achieved only if the task periods are suitably related.

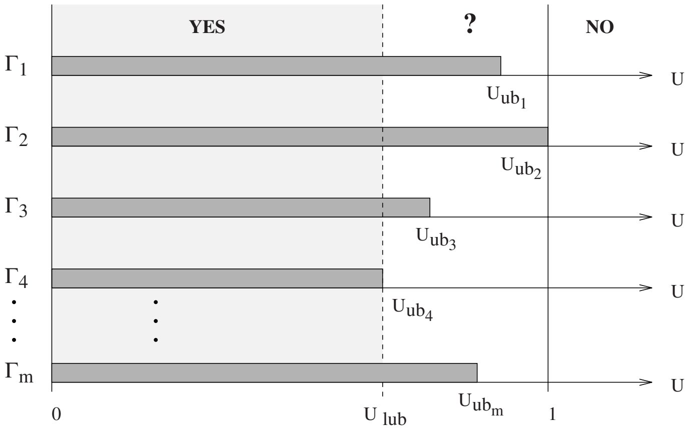

Figure 4.3 Meaning of the least upper bound of the processor utilization factor.

If the utilization factor of a task set is greater than 1.0, the task set cannot be scheduled by any algorithm. To show this result, let H be the hyperperiod of the task set. If $U > 1$, we also have $UH > H$, which can be written as

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

The factor (H/T_i) represents the (integer) number of times τ_i is executed in the hyperperiod, whereas the quantity $(H/T_i)C_i$ is the total computation time requested by τ_i in the hyperperiod. Hence, the sum on the left hand side represents the total computational demand requested by the task set in $[0, H)$. Clearly, if the total demand exceeds the available processor time, there is no feasible schedule for the task set.

4.2 TIMELINE SCHEDULING

Timeline Scheduling (TS), also known as a *Cyclic Executive*, is one of the most used approaches to handle periodic tasks in defense military systems and traffic control systems. The method consists of dividing the temporal axis into slots of equal length, in which one or more tasks can be allocated for execution, in such a way to respect the frequencies derived from the application requirements. A timer synchronizes the activation of the tasks at the beginning of each time slot. In order to illustrate this method, consider the following example, in which three tasks, A, B and C, need to be executed with a frequency of 40, 20 and 10 Hz, respectively. By analyzing the

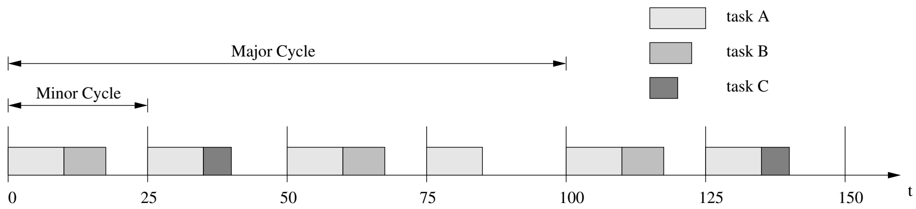

Figure 4.4 Example of timeline scheduling.

task periods ($T_A = 25$ ms, $T_B = 50$ ms, $T_C = 100$ ms), it is easy to verify that the optimal length for the time slot is 25 ms, which is the Greatest Common Divisor of the periods. Hence, to meet the required frequencies, task A needs to be executed in every time slot, task B every two slots, and task C every four slots. A possible scheduling solution for this task set is illustrated in Figure 4.4.

The duration of the time slot is also called a Minor Cycle, whereas the minimum interval of time after which the schedule repeats itself (the hyperperiod) is also called a Major Cycle. In general, the major cycle is equal to the least common multiple of all the periods (in the example, it is equal to 100 ms). In order to guarantee a priori that a schedule is feasible on a particular processor, it is sufficient to know the task worst-case execution times and verify that the sum of the executions within each time slot is less than or equal to the minor cycle. In the example shown in Figure 4.4, if C_A, C_B and C_C denote the execution times of the tasks, it is sufficient to verify that

$$\begin{cases} C_A + C_B \leq 25ms \\ C_A + C_C \leq 25ms \end{cases}$$

The main advantage of timeline scheduling is its simplicity. The method can be implemented by programming a timer to interrupt with a period equal to the minor cycle and by writing a main program that calls the tasks in the order given in the major cycle, inserting a time synchronization point at the beginning of each minor cycle. Since the task sequence is not decided by a scheduling algorithm in the kernel, but it is triggered by the calls made by the main program, there are no context switches, so the runtime overhead is very low. Moreover, the sequence of tasks in the schedule is always the same, can be easily visualized, and it is not affected by jitter (i.e., task start times and response times are not subject to large variations).

In spite of these advantages, timeline scheduling has some problems. For example, it is very fragile during overload conditions. If a task does not terminate at the minor cycle boundary, it can either be continued or aborted. In both cases, however, the system may run into a critical situation. In fact, if the failing task is left in execution, it can cause a domino effect on the other tasks, breaking the entire schedule (timeline break). On the other hand, if the failing task is aborted while updating some shared data, the system may be left in an inconsistent state, jeopardizing the correct system behavior.

Another big problem of the timeline scheduling technique is its sensitivity to application changes. If updating a task requires an increase of its computation time or its activation frequency, the entire scheduling sequence may need to be reconstructed from scratch. Considering the previous example, if task B is updated to B' and the code change is such that $C_A + C_{B'} > 25ms$, then task B' must be split in two or more pieces to be allocated in the available intervals of the timeline. Changing the task frequencies may cause even more radical changes in the schedule. For example, if the frequency of task B changes from 20 Hz to 25 Hz (that is T_B changes from 50 to 40 ms), the previous schedule is not valid any more, because the new Minor Cycle is equal to 5 ms and the new Major Cycle is equal to 200 ms. Note that after this change, since the Minor cycle is much shorter than before, all the tasks may need to be split into small pieces to fit in the new time slots.

Finally, another limitation of the timeline scheduling is that it is difficult to handle aperiodic activities efficiently without changing the task sequence. The problems outlined above can be solved by using priority-based scheduling algorithms.

4.3 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: a priority P_i is assigned to the task before execution and does 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.

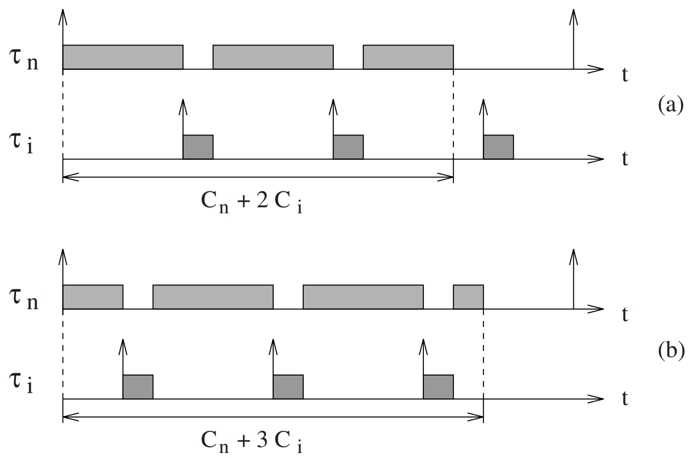

Figure 4.5 **a.** The response time of task τ_n is delayed by the interference of τ_i with higher priority. **b.** The interference may increase advancing the release time of τ_i.

4.3.1 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 4.5a, the response time of task τ_n is delayed by the interference of τ_i with higher priority. Moreover, from Figure 4.5b, 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 $\tau_i,\ i = 2, \ldots, n-1$, we prove 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 τ_2 will receive the highest priority. This situation

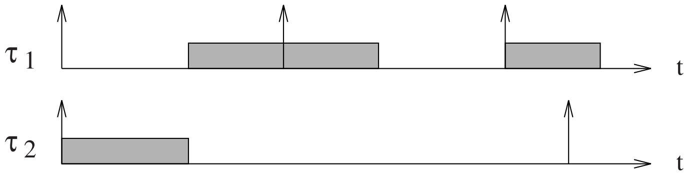

Figure 4.6 Tasks scheduled by an algorithm different from RM.

is depicted in Figure 4.6, 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 4.7, 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.

Case 1. The computation time of τ_1 (synchronously activated with τ_2) is short enough that all its requests are completed before the second request of τ_2. That is, $C_1 < T_2 - FT_1$.

In this case, from Figure 4.7a, 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 $C_1 < T_2 - FT_1$, we have

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

which satisfies (4.2).

[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.

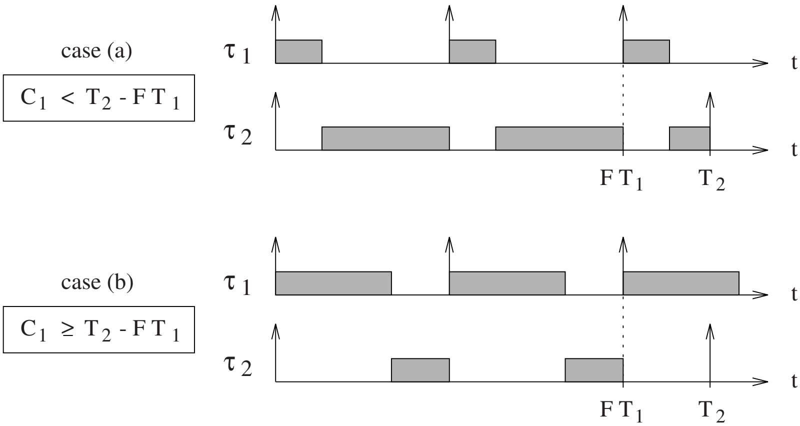

Figure 4.7 Schedule produced by RM in two different conditions.

Case 2. The computation time of τ_1 (synchronously activated with τ_2) is long enough to overlap with the second request of τ_2.
That is, $C_1 \geq T_2 - FT_1$.

In this case, from Figure 4.7b, 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.

4.3.2 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 task set by inflating computation times to fully utilize the processor; and
- minimize the upper bound U_{ub} with respect to all the 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 of τ_1 (synchronously activated with τ_2) is short enough that all its requests are completed before the second request of τ_2. That is, $C_1 \leq T_2 - FT_1$.

In this situation, as depicted in Figure 4.8, 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

$$C_1 = T_2 - FT_1.$$

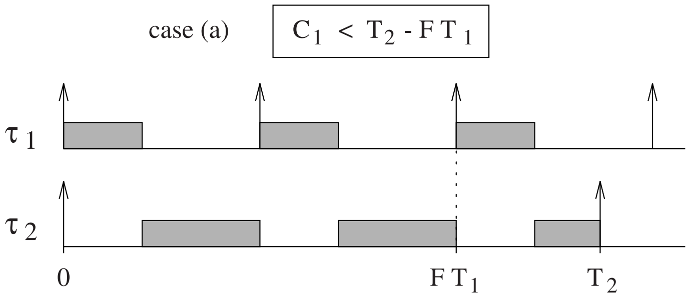

Figure 4.8 The second request of τ_2 is released when τ_1 is idle.

Case 2. The computation time of τ_1 (synchronously activated with τ_2) is long enough to overlap with the second request of τ_2.
That is, $C_1 \geq T_2 - FT_1$.

In this situation, depicted in Figure 4.9, the largest possible value of C_2 is

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

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

$$C_1 = T_2 - T_1 F.$$

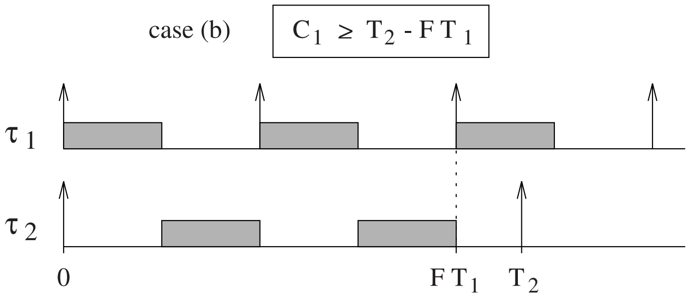

Figure 4.9 The second request of τ_2 is released when τ_1 is active.

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 $G = G_2$:

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

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 4.10 shows the upper bound of U as a function of k.

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.

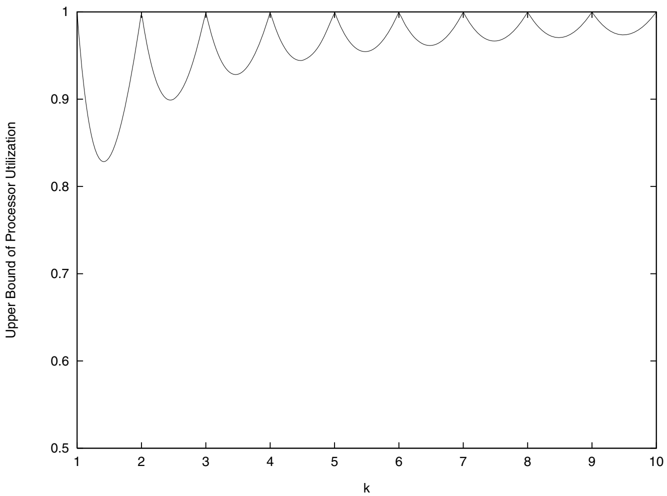

Figure 4.10 Upper bound of the processor utilization factor as a function of the ratio $k = T_2/T_1$.

4.3.3 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

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

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

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

Thus, the processor utilization factor becomes

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

Defining

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

and noting that

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

the utilization factor may be written as

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

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

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

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.

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

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

That is, when all R_i have the same value:

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

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

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

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

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

4.3.4 HYPERBOLIC BOUND FOR RM

The feasibility analysis of the RM algorithm can also be performed using a different approach, called the Hyperbolic Bound [BBB01, BBB03]. The test has the same complexity as the original Liu and Layland bound but it is less pessimistic, as it accepts task sets that would be rejected using the original approach. Instead of minimizing the processor utilization with respect to task periods, the feasibility condition can be manipulated in order to find a tighter sufficient schedulability test as a function of the individual task utilizations.

The following theorem provides a sufficient condition for testing the schedulability of a task set under the RM algorithm.

Theorem 4.1 *Let $\Gamma = \{\tau_1, \ldots, \tau_n\}$ be a set of n periodic tasks, where each task τ_i is characterized by a processor utilization U_i. Then, Γ is schedulable with the RM algorithm if*

$$\prod_{i=1}^{n}(U_i + 1) \leq 2. \tag{4.10}$$

Proof. Without loss of generality, we may assume that tasks are ordered by increasing periods, so that τ_1 is the task with the highest priority and τ_n is the task with the lowest priority. In [LL73], as well as in [DG00], it has been shown that the worst-case scenario for a set on n periodic tasks occurs when all the tasks start simultaneously (e.g., at time $t = 0$) and periods are such that

$$\forall i = 2, \ldots, n \quad T_1 < T_i < 2T_1.$$

Moreover, the total utilization factor is minimized when computation times have the following relations:

$$\begin{cases} C_1 \;=\; T_2 - T_1 \\ C_2 \;=\; T_3 - T_2 \\ \ldots \\ C_{n-1} \;=\; T_n - T_{n-1} \end{cases} \tag{4.11}$$

and the schedulability condition is given by:

$$\sum_{i=1}^{n} C_i \;\leq\; T_1. \tag{4.12}$$

From Equations (4.11), the schedulability condition can also be written as

$$C_n \;\leq\; 2T_1 - T_n \tag{4.13}$$

Now, defining

$$R_i = \frac{T_{i+1}}{T_i} \quad \text{and} \quad U_i = \frac{C_i}{T_i}.$$

Equations (4.11) can be written as follows:

$$\begin{cases} U_1 = R_1 - 1 \\ U_2 = R_2 - 1 \\ \dots \\ U_{n-1} = R_{n-1} - 1. \end{cases} \tag{4.14}$$

Now we notice that:

$$\prod_{i=1}^{n-1} R_i = \frac{T_2}{T_1}\frac{T_3}{T_2}\cdots\frac{T_n}{T_{n-1}} = \frac{T_n}{T_1}.$$

If we divide both sides of the feasibility condition (4.13) by T_n, we get:

$$U_n \leq \frac{2T_1}{T_n} - 1.$$

Hence, the feasibility condition for a task set that fully utilizes the processor can be written as

$$U_n + 1 \leq \frac{2}{\prod_{i=1}^{n-1} R_i}.$$

Since $R_i = U_i + 1$ for all $i = 1, \dots, n-1$, we have

$$(U_n + 1)\prod_{i=1}^{n-1}(U_i + 1) \leq 2$$

and finally

$$\prod_{i=1}^{n}(U_i + 1) \leq 2,$$

which proves the theorem. □

The new test can be compared with the Liu and Layland one in the task utilization space, denoted as the U-space. Here, the Liu and Layland bound for RM is represented by a n-dimensional plane that intersects each axis in $U_{\mathrm{lub}}(n) = n(2^{1/n} - 1)$. All points below the RM surface represent periodic task sets that are feasible by RM. The new bound expressed by equation (4.10) is represented by a n-dimensional hyperbolic surface tangent to the RM plane and intersecting the axes for $U_i = 1$ (this is the reason why it is referred to as the hyperbolic bound). Figure 4.11 illustrates such bounds for $n = 2$. Notice that the asymptotes of the hyperbole are at $U_i = -1$. From the plots, we can clearly see that the feasibility region below the H-bound is larger than that below the LL-bound, and the gain is given by the dark gray area.

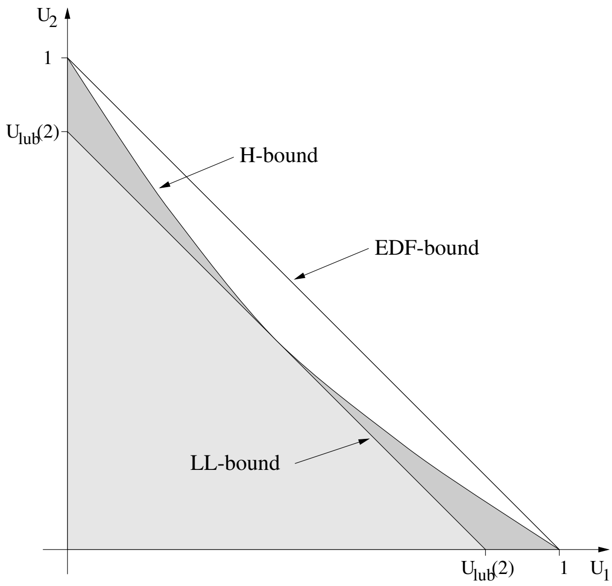

Figure 4.11 Schedulability bounds for RM and EDF in the utilization space.

It has been shown [BBB03] that the hyperbolic bound is tight, meaning that, if not satisfied, it is always possible to construct an unfeasible task set with those utilizations. Hence, the hyperbolic bound is the best possible test that can be found using the individual utilization factors U_i as a task set knowledge.

Moreover, the gain (in terms of schedulability) achieved by the hyperbolic test over the classical Liu and Layland test increases as a function of the number of tasks, and tends to $\sqrt{2}$ for n tending to infinity.

4.4 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 typically executed in preemptive mode, thus the currently executing task is preempted whenever another periodic instance with earlier deadline becomes active.

Note 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.

4.4.1 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 4.2 *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.

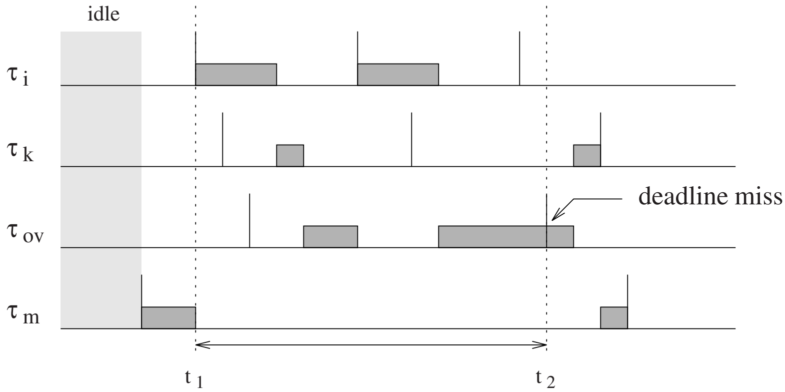

Figure 4.12 Interval of continuous utilization in an EDF schedule before a deadline miss.

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 first instant at which a deadline is missed and let $[t_1, t_2]$ be the longest interval of continuous utilization, before t_2, such that only instances with deadline less than or equal to t_2 are executed in $[t_1, t_2]$ (see Figure 4.12 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

$$C_p(t_1, t_2) \;=\; \sum_{r_k \geq t_1, d_k \leq t_2} C_k \;=\; \sum_{i=1}^{n} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i. \tag{4.15}$$

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. $\square$

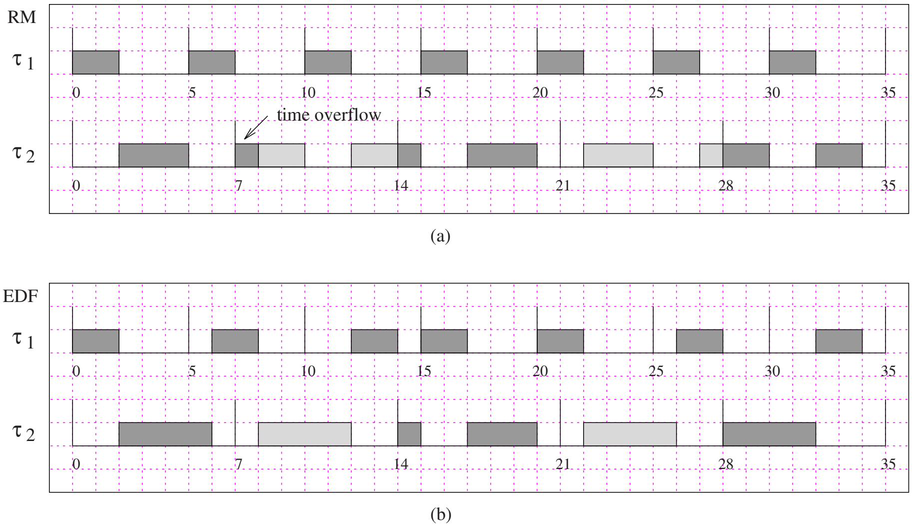

Figure 4.13 Schedule produced by RM (a) and EDF (b) on the same set of periodic tasks.

4.4.2 AN EXAMPLE

Consider the periodic task set illustrated in Figure 4.13, for which the processor utilization factor is

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

This means that 97 percent of the processor time is used to execute the periodic tasks, whereas the CPU is idle in the remaining 3 percent. Being $U > 2(\sqrt{2}-1) \simeq 0.83$, the schedulability of the task set cannot be guaranteed under RM, whereas it is guaranteed under EDF. Indeed, as shown in Figure 4.13a, RM generates a deadline miss at time $t = 7$, whereas EDF completes all tasks within their deadlines (see Figure 4.13b). Another important difference between RM and EDF concerns the number of preemptions occurring in the schedule. As shown in Figure 4.13, 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 the same interval. The smaller 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.

4.5 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 relative deadlines less than or equal to their period (i.e., *constrained deadlines*). 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 4.14 and have the following relationships:

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

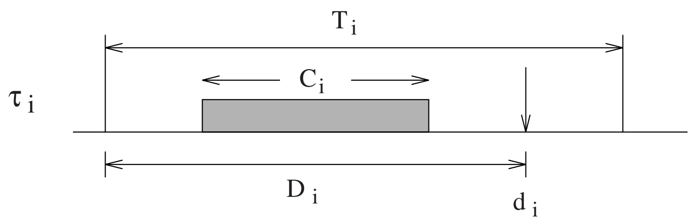

Figure 4.14 Task parameters in Deadline-Monotonic scheduling.

According to the DM algorithm, each task is assigned a fixed priority P_i inversely proportional to its relative deadline D_i. 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 normally used in a fully preemptive mode; 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 a task set is schedulable by some fixed priority assignment, then it is also schedulable by DM.

4.5.1 SCHEDULABILITY ANALYSIS

The feasibility of a task set with constrained deadlines could be guaranteed using the utilization based 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 be quite pessimistic, since 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 can be expressed as follows:

$$\forall i : 1 \leq i \leq n \quad C_i + I_i \leq D_i, \tag{4.16}$$

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:

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

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

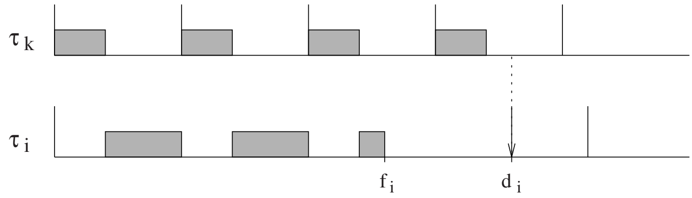

Figure 4.15 More accurate calculation of the interference on τ_i by higher priority tasks.

Note 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 4.15, 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, called Response Time Analysis.

4.5.2 RESPONSE TIME ANALYSIS

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 I_i of the 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,

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \tag{4.17}$$

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 (4.17). Note, however, that only a subset of points in the interval $[0, D_i]$ need

	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.

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 k-th estimate of R_i and let $I_i^{(k)}$ be the interference on task τ_i in the interval $[0, R_i^{(k)}]$:

$$I_i^{(k)} = \sum_{j=1}^{i-1} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j. \tag{4.18}$$

Then the calculation of R_i is performed as follows:

1. Iteration starts with $R_i^{(0)} = \sum_{j=1}^{i} C_j$, 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 (4.18).

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 4.16, and the iteration steps are shown below.

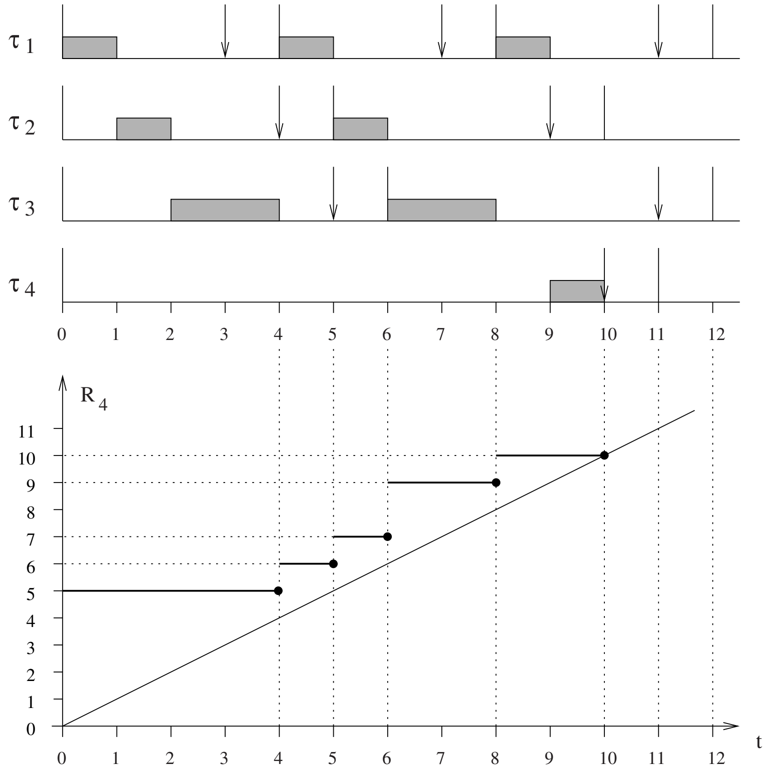

Figure 4.16 Example of schedule produced by DM and response time experienced by τ_4 as a function of the considered interval.

$$
\begin{array}{lll}
\text{Step 0:} & R_4^{(0)} = \sum_{i=1}^{4} C_i = 5, & \text{but } I_4^{(0)} = 5 \text{ and } I_4^{(0)} + C_4 > R_4^{(0)} \\
& & \text{hence } \tau_4 \text{ does not finish at } R_4^{(0)}. \\[1em]
\text{Step 1:} & R_4^{(1)} = I_4^{(0)} + C_4 = 6, & \text{but } I_4^{(1)} = 6 \text{ and } I_4^{(1)} + C_4 > R_4^{(1)} \\
& & \text{hence } \tau_4 \text{ does not finish at } R_4^{(1)}. \\[1em]
\text{Step 2:} & R_4^{(2)} = I_4^{(1)} + C_4 = 7, & \text{but } I_4^{(2)} = 8 \text{ and } I_4^{(2)} + C_4 > R_4^{(2)} \\
& & \text{hence } \tau_4 \text{ does not finish at } R_4^{(2)}. \\[1em]
\text{Step 3:} & R_4^{(3)} = I_4^{(2)} + C_4 = 9, & \text{but } I_4^{(3)} = 9 \text{ and } I_4^{(3)} + C_4 > R_4^{(3)} \\
& & \text{hence } \tau_4 \text{ does not finish at } R_4^{(3)}. \\[1em]
\text{Step 4:} & R_4^{(4)} = I_4^{(3)} + C_4 = 10, & \text{but } I_4^{(4)} = 9 \text{ and } I_4^{(4)} + C_4 = R_4^{(4)} \\
& & \text{hence } \tau_4 \text{ finishes at } R_4 = R_4^{(4)} = 10.
\end{array}
$$

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 4.17.

DM_guarantee (Γ) $\{$
    for (each $\tau_i \in \Gamma$) $\{$
        $I_i = \sum_{k=1}^{i-1} C_k$;
        do $\{$
            $R_i = I_i + C_i$;
            if $(R_i > D_i)$ return(UNSCHEDULABLE);
            $I_i = \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$;
        $\}$ **while** $(I_i + C_i > R_i)$;
    $\}$
    return(SCHEDULABLE);
$\}$

Figure 4.17 Algorithm for testing the schedulability of a periodic task set Γ under Deadline Monotonic.

Note that the algorithm in Figure 4.17 has a pseudo-polynomial complexity. In fact, the guarantee of the entire task set requires $O(nN)$ steps, where n is the number of tasks and N is the number of iterations in the inner loop, which does not depend directly on n, but on the period relations.

4.5.3 WORKLOAD ANALYSIS

Another necessary and sufficient test for checking the schedulability of fixed priority systems with constrained deadlines was proposed by Lehoczky, Sha, and Ding [LSD89]. The test is based on the concept of Level-i workload, defined as follows.

Definition 4.1 *The Level-i workload $W_i(t)$ is the cumulative computation time requested in the interval $(0, t]$ by task τ_i and all the tasks with priority higher than P_i.*

For a set of synchronous periodic tasks, the Level-i workload can be computed as follows:

$$W_i(t) = C_i + \sum_{h:P_h>P_i} \left\lceil \frac{t}{T_h} \right\rceil C_h. \tag{4.19}$$

Then, the test can be expressed by the following theorem:

Theorem 4.3 (Lehoczky, Sha, Ding, 1989) *A set of fully preemptive periodic tasks can be scheduled by a fixed priority algorithm if and only if*

$$\forall i = 1, \ldots, n \quad \exists t \in (0, D_i] : W_i(t) \leq t. \tag{4.20}$$

Later, Bini and Buttazzo [BB04] restricted the number of points in which condition (4.20) has to be checked to the following *Testing Set*:

$$\mathcal{TS}_i \stackrel{\text{def}}{=} \mathcal{P}_{i-1}(D_i) \tag{4.21}$$

where $\mathcal{P}_i(t)$ is defined by the following recurrent expression:

$$\begin{cases} \mathcal{P}_0(t) = \{t\} \\ \mathcal{P}_i(t) = \mathcal{P}_{i-1}\left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i\right) \cup \mathcal{P}_{i-1}(t). \end{cases} \tag{4.22}$$

Thus, the schedulability test can be expressed by the following theorem:

Theorem 4.4 (Bini and Buttazzo, 2004) *A set of fully preemptive periodic tasks can be scheduled by a fixed priority algorithm if and only if*

$$\forall i = 1, \ldots, n \quad \exists t \in \mathcal{TS}_i : W_i(t) \leq t. \tag{4.23}$$

An advantage of Equation (4.23) is that it can be formulated as the union of a set of simpler conditions, leading to a more efficient guarantee test, named the Hyperplanes test [BB04]. The test has still a pseudo-polynomial complexity, but runs much quicker than the response time analysis in the average case. Moreover, a novel feature of this test is that it can be tuned using a parameter to balance acceptance ratio versus complexity. Such a tunability property is important in those cases in which the performance of a polynomial time test is not sufficient for achieving high processor utilization, and the overhead introduced by exact tests is too high for an online admission control.

Another advantage of this formulation is that Equation (4.23) can be manipulated to describe the feasibility region of the task set in a desired space of design parameters, so enabling sensitivity analysis [BDNB08], which determines how to change task set parameters when the schedule is infeasible.

4.6 EDF WITH CONSTRAINED DEADLINES

Under EDF, the analysis of periodic tasks with deadlines less than or equal to periods can be performed using the *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.

4.6.1 THE PROCESSOR DEMAND APPROACH

In general, the processor demand of a task τ_i in an interval $[t_1, t_2]$ is the amount of processing time $g_i(t_1, t_2)$ requested by those instances of τ_i activated in $[t_1, t_2]$ that must be completed in $[t_1, t_2]$. That is,

$$g_i(t_1, t_2) \;=\; \sum_{r_{i,k} \geq t_1, d_{i,k} \leq t_2} C_i.$$

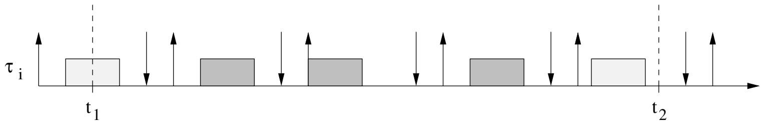

Figure 4.18 The instances in dark gray are those contributing to the processor demand in $[t_1, t_2]$.

For the whole task set, the processor demand in $[t_1, t_2]$ is given by

$$g(t_1, t_2) = \sum_{i=1}^{n} g_i(t_1, t_2).$$

Then, the feasibility of a task set is guaranteed if and only if *in any interval of time* the processor demand does not exceed the available time; that is, if and only if

$$\forall t_1, t_2 \quad g(t_1, t_2) \leq (t_2 - t_1).$$

Referring to Figure 4.18, the number of instances of task τ_i that contribute to the demand in $[t_1, t_2]$ can be expressed as

$$\eta_i(t_1, t_2) = max \left\{ 0, \left\lfloor \frac{t_2 + T_i - D_i - \Phi_i}{T_i} \right\rfloor - \left\lceil \frac{t_1 - \Phi_i}{T_i} \right\rceil \right\}$$

and the processor demand in $[t_1, t_2]$ can be computed as

$$g(t_1, t_2) = \sum_{i=1}^{n} \eta_i(t_1, t_2) C_i. \tag{4.24}$$

If relatives deadlines are no larger than periods and periodic tasks are simultaneously activated at time $t = 0$ (i.e., $\Phi_i = 0$ for all the tasks), then the number of instances contributing to the demand in an interval $[0, L]$ can be expressed as:

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

Thus, the processor demand in $[0, L]$ can be computed as

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

Function $g(0, L)$ is also referred to as Demand Bound Function:

$$\mathsf{dbf}(t) \stackrel{\text{def}}{=} \sum_{i=1}^{n} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor C_i. \tag{4.25}$$

Therefore, a synchronous set of periodic tasks with relative deadlines less than or equal to periods is schedulable by EDF if and only if

$$\forall t > 0 \quad \mathsf{dbf}(t) \ \leq \ t. \tag{4.26}$$

It is worth observing that, for the special case of tasks with relative deadlines equal to periods, the test based on the processor demand criterion is equivalent to the one based on the processor utilization. This result is formally expressed by the following theorem [JS93].

Theorem 4.5 (Jeffay and Stone, 1993) *A set of periodic tasks with relative deadlines equal to periods is schedulable by EDF if and only if*

$$\forall L > 0 \quad \sum_{i=1}^{n} \left\lfloor \frac{L}{T_i} \right\rfloor C_i \ \leq \ L. \tag{4.27}$$

Proof. The theorem is proved by showing that equation (4.27) is equivalent to the classical Liu and Layland's condition

$$U \ = \ \sum_{i=1}^{n} \frac{C_i}{T_i} \ \leq \ 1. \tag{4.28}$$

(4.28) $\Rightarrow$ (4.27). 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 (4.28) $\Leftarrow$ (4.27) we show that $\neg$(4.28) $\Rightarrow$ $\neg$(4.27). That is, we assume $U > 1$ and prove that there exist an $L \geq 0$ for which (4.27) does not hold. If $U > 1$, then for $L = H = lcm(T_1, \ldots, T_n)$,

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

□

4.6.2 REDUCING TEST INTERVALS

In this section we show that the feasibility test expressed by condition (4.26) can be simplified by reducing the number of intervals in which it has to be verified. We first observe that

1. if tasks are periodic and are simultaneously activated at time $t = 0$, then the schedule repeats itself every hyperperiod H; thus condition (4.26) needs to be verified only for values of L less than or equal to H.

2. $g(0, L)$ is a step function whose value increases when L crosses a deadline d_k and remains constant until the next deadline d_{k+1}. This means that if condition $g(0, L) < L$ holds for $L = d_k$, then it also holds for all L such that $d_k \leq L < d_{k+1}$. As a consequence, condition (4.26) needs to be verified only for values of L equal to absolute deadlines.

The number of testing points can be reduced further by noting that

$$\left\lfloor \frac{L + T_i - D_i}{T_i} \right\rfloor \;\leq\; \left(\frac{L + T_i - D_i}{T_i} \right).$$

and defining

$$G(0, L) \;=\; \sum_{i=1}^{n} \frac{L + T_i - D_i}{T_i} C_i \;=\; \sum_{i=1}^{n} \frac{T_i - D_i}{T_i} C_i + \frac{L}{T_i} C_i$$

we have that

$$\forall L > 0, \quad g(0, L) \;\leq\; G(0, L),$$

where

$$G(0, L) \;=\; \sum_{i=1}^{n} (T_i - D_i) U_i + LU.$$

From Figure 4.19, we can note that $G(0, L)$ is a function of L increasing as a straight line with slope U. Hence, if $U < 1$, there exists an $L = L^*$ for which $G(0, L) = L$. Clearly, for all $L \geq L^*$, we have that $g(0, L) \leq G(0, L) \leq L$, meaning that the schedulability of the task set is guaranteed. As a consequence, there is no need to verify condition (4.26) for values of $L \geq L^*$.

The value of L^* is the time at which $G(0, L^*) = L^*$; that is,

$$\sum_{i=1}^{n} (T_i - D_i) U_i + L^* U \;=\; L^*,$$

which gives

$$L^* \;=\; \frac{\sum_{i=1}^{n} (T_i - D_i) U_i}{1 - U}.$$

Considering that the task set must be tested at least until the largest relative deadline D_{max}, the results of the previous observations can be combined in the following theorem.

Theorem 4.6 *A set of synchronous periodic tasks with relative deadlines less than or equal to periods can be scheduled by EDF if and only if $U < 1$ and*

$$\forall t \in \mathcal{D} \quad \mathsf{dbf}(t) \ \leq \ t. \tag{4.29}$$

where

$$\mathcal{D} = \{d_k \mid d_k \leq \min[H, \max(D_{max}, L^*)]\}$$

and

$$L^* \ = \ \frac{\sum_{i=1}^{n}(T_i - D_i)U_i}{1 - U}.$$

EXAMPLE

To illustrate the processor demand criterion, consider the task set shown in Table 4.4, where three periodic tasks with deadlines less than periods need to be guaranteed under EDF. From the specified parameters it is easy to compute that

$$\begin{aligned} U &= \frac{2}{6} + \frac{2}{8} + \frac{3}{9} = \frac{11}{12} \\ L^* &= \frac{\sum_{i=1}^{n}(T_i - D_i)U_i}{1 - U} = 25 \\ H &= \text{lcm}(6, 8, 9) = 72. \end{aligned}$$

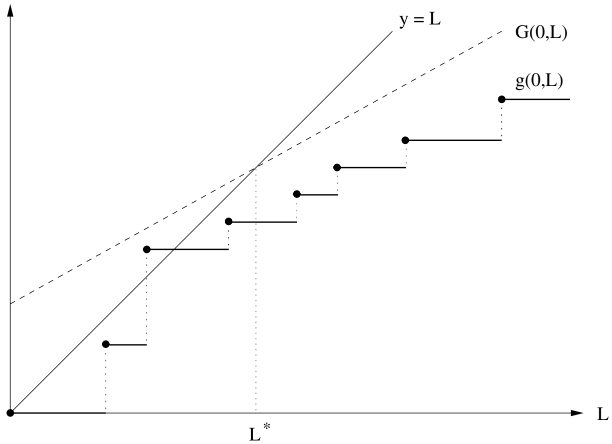

Figure 4.19 Maximum value of L for which the processor demand test has to be verified.

	C_i	D_i	T_i
τ_1	2	4	6
τ_2	2	5	8
τ_3	3	7	9

Table 4.4 A task set with relative deadlines less than periods.

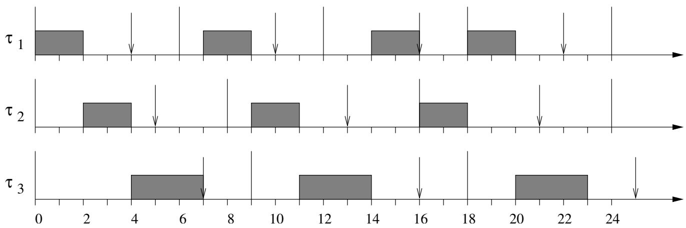

Figure 4.20 Schedule produced by EDF for the task set shown in Table 4.4.

L	$g(0, L)$	result
4	2	OK
5	4	OK
7	7	OK
10	9	OK
13	11	OK
16	16	OK
21	18	OK
22	20	OK

Table 4.5 Testing intervals for the processor demand criterion.

Hence, condition (4.29) has to be tested for any deadline less than 25, and the set of checking points is given by $\mathcal{D} = \{4, 5, 7, 10, 13, 16, 21, 22\}$. Table 4.5 shows the results of the test and Figure 4.20 illustrates the schedule produced by EDF for the task set.

4.7 COMPARISON BETWEEN RM AND EDF

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 major advantage of the fixed priority approach is that it is simpler to implement. In fact, if the ready queue is implemented as a multi-level queue with P priority levels (where P is the number of different priorities in the system), both task insertion and extraction can be achieved in $O(1)$. On the other hand, in a deadline driven scheduler, the best solution for the ready queue is to implement it as a heap (i.e., a balanced binary tree), where task management requires an $O(\log n)$ complexity.

Except for such an implementation issue, which becomes relevant only for very large task sets (consisting of hundreds of tasks), or for very slow processors, a dynamic priority scheme has many advantages with respect to a fixed priority algorithm. A detailed comparison between RM and EDF has been presented by Buttazzo [But03, But05].

In terms of schedulability analysis, an exact guarantee test for RM requires a pseudo-polynomial complexity, even in the simple case of independent tasks with relative deadlines equal to periods, whereas it can be performed in $O(n)$ for EDF. In the general case in which deadlines can be less than or equal to periods, the schedulability analysis becomes pseudo-polynomial for both algorithms. Under fixed-priority assignments, the feasibility of the task set can be tested using the response time analysis, whereas under dynamic priority assignments it can be tested using the processor demand criterion.

As for the processor utilization, EDF is able to exploit the full processor bandwidth, whereas the RM algorithm can only guarantee feasibility for task sets with utilization less than 69%, in the worst case. In the average case, a statistical study performed by Lehoczky, Sha, and Ding [LSD89] showed that for task sets with randomly generated parameters the RM algorithm is able to feasibly schedule task sets with a processor utilization up to about 88%. However, this is only a statistical result and cannot be taken as an absolute bound for performing a precise guarantee test.

In spite of the extra computation needed by EDF for updating the absolute deadline at each job activation, EDF introduces less runtime overhead than RM, when context switches are taken into account. In fact, to enforce the fixed priority order, the number of preemptions that typically occur under RM is much higher than under EDF.

An interesting property of EDF during permanent overloads is that it automatically performs a period rescaling, so tasks start behaving as they were executing at a lower rate. This property has been proved by Cervin in his PhD dissertation [Cer03] and it is formally stated in the following theorem.

Theorem 4.7 (Cervin) *Assume a set of n periodic tasks, where each task is described by a fixed period T_i, a fixed execution time C_i, a relative deadline D_i, and a release offset Φ_i. If $U > 1$ and tasks are scheduled by EDF, then, in stationarity, the average period $\overline{T_i}$ of each task τ_i is given by $\overline{T_i} = T_i U$.*

Note that under fixed priority scheduling, a permanent overload condition causes a complete blocking of the lower priority tasks.

As discussed later in the book, another major advantage of dynamic scheduling with respect to fixed priority scheduling is a better responsiveness in handling aperiodic tasks. This property comes from the higher processor utilization bound of EDF. In fact, the lower schedulability bound of RM limits the maximum utilization ($U_s = C_s/T_s$) that can be assigned to a server for guaranteeing the feasibility of the periodic task set. As a consequence, the spare processor utilization that cannot be assigned to the server is wasted as a background execution. This problem does not occur under EDF, where, if U_p is the processor utilization of the periodic tasks, the full remaining fraction $1-U_p$ can always be allocated to the server for aperiodic execution.

Exercises

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

	C_i	T_i
τ_1	2	6
τ_2	2	8
τ_3	2	12

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

	C_i	T_i
τ_1	3	5
τ_2	1	8
τ_3	1	10

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

	C_i	T_i
τ_1	1	4
τ_2	2	6
τ_3	3	10

4.4 Verify the schedulability under RM of the following task set:

	C_i	T_i
τ_1	1	4
τ_2	2	6
τ_3	3	8

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

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

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

4.7 Verify the schedulability of the task set described in Exercise 4.6 using the Deadline-Monotonic algorithm. Then construct the schedule.