Module 6

Embedded System Software

Lesson 30

Real-Time Task Scheduling – Part 2

Specific Instructional Objectives

At the end of this lesson, the student would be able to:

- Get an introduction to event-driven schedulers
- Understand the basics of Foreground-Background schedulers
- Get an overview of Earliest Deadline First (EDF) Algorithm
- Work out solutions to problems based on EDF
- Know the shortcomings of EDF
- Get an overview of Rate Monotonic Algorithm (RMA)
- Know the necessary and sufficient conditions for a set of real-time tasks to be RMA-schedulable
- Work out solutions to problems based on EDF
- Infer the maximum achievable CPU utilization
- Understand the Advantages and Disadvantages of RMA
- Get an overview of Deadline Monotonic Algorithm (DMA)
- Understand the phenomenon of Context-Switching and Self-Suspension

1. Event-driven Scheduling – An Introduction

In this lesson, we shall discuss the various algorithms for event-driven scheduling. From the previous lesson, we may recollect the following points:
The clock-driven schedulers are those in which the scheduling points are determined by the interrupts received from a clock. In the event-driven ones, the scheduling points are defined by certain events which precludes clock interrupts. The hybrid ones use both clock interrupts as well as event occurrences to define their scheduling points

Cyclic schedulers are very efficient. However, a prominent shortcoming of the cyclic schedulers is that it becomes very complex to determine a suitable frame size as well as a feasible schedule when the number of tasks increases. Further, in almost every frame some processing time is wasted (as the frame size is larger than all task execution times) resulting in sub-optimal schedules. Event-driven schedulers overcome these shortcomings. Further, event-driven schedulers can handle aperiodic and sporadic tasks more proficiently. On the flip side, event-driven schedulers are less efficient as they deploy more complex scheduling algorithms. Therefore, event-driven schedulers are less suitable for embedded applications as these are required to be of small size, low cost, and consume minimal amount of power.

It should now be clear why event-driven schedulers are invariably used in all moderate and large-sized applications having many tasks, whereas cyclic schedulers are predominantly used in small applications. In event-driven scheduling, the scheduling points are defined by task completion and task arrival events. This class of schedulers is normally preemptive, i.e., when a higher priority task becomes ready, it preempts any lower priority task that may be running.

1.1. Types of Event Driven Schedulers

We discuss three important types of event-driven schedulers:

- Simple priority-based
- Rate Monotonic Analysis (RMA)
- Earliest Deadline First (EDF)

The simplest of these is the foreground-background scheduler, which we discuss next. In section 3.4, we discuss EDF and in section 3.5, we discuss RMA.

1.2. Foreground-Background Scheduler

A foreground-background scheduler is possibly the simplest priority-driven preemptive scheduler. In foreground-background scheduling, the real-time tasks in an application are run as fore- ground tasks. The sporadic, aperiodic, and non-real-time tasks are run as background tasks. Among the foreground tasks, at every scheduling point the highest priority task is taken up for scheduling. A background task can run when none of the foreground tasks is ready. In other words, the background tasks run at the lowest priority.

Let us assume that in a certain real-time system, there are n foreground tasks which are denoted as: T1,T2,...,Tn. As already mentioned, the foreground tasks are all periodic. Let T_B be the only background task. Let e_B be the processing time requirement of T_B. In this case, the completion time (ct_B) for the background task is given by:

$$ct_B = e_B / (1 - \sum_{i=1}^{n} e_i / p_i) \quad \ldots \quad (3.1/2.7)$$

This expression is easy to interpret. When any foreground task is executing, the background task waits. The average CPU utilization due to the foreground task T_i is e_i/p_i, since e_i amount of processing time is required over every p_i period. It follows that all foreground tasks together would result in CPU utilization of $\sum_{i=1}^{n} e_i / p_i$. Therefore, the average time available for execution of the background tasks in every unit of time is $1 - \sum_{i=1}^{n} e_i / p_i$. Hence, Expr. 2.7 follows easily. We now illustrate the applicability of Expr. 2.7 through the following three simple examples.

1.3. Examples

Example 1: Consider a real-time system in which tasks are scheduled using foreground-background scheduling. There is only one periodic foreground task T_f: (ϕ_f =0, p_f =50 msec, e_f =100 msec, d_f =100 msec) and the background task be T_B = (e_B =1000 msec). Compute the completion time for background task.

Solution: By using the expression (2.7) to compute the task completion time, we have

$$ct_B = 1000 / (1 - 50/100) = 2000 \text{ msec}$$

So, the background task T_B would take 2000 milliseconds to complete.

Example 2: In a simple priority-driven preemptive scheduler, two periodic tasks T1 and T2 and a background task are scheduled. The periodic task T1 has the highest priority and executes once every 20 milliseconds and requires 10 milliseconds of execution time each time. T2 requires 20 milliseconds of processing every 50 milliseconds. T3 is a background task and requires 100 milliseconds to complete. Assuming that all the tasks start at time 0, determine the time at which T3 will complete.

Solution: The total utilization due to the foreground tasks: $\sum_{i=1}^{2} e_i / p_i = 10/20 + 20/50 = 90/100$.

This implies that the fraction of time remaining for the background task to execute is given by:

$$1 - \sum_{i=1}^{2} e_i / p_i = 10/100.$$

Therefore, the background task gets 1 millisecond every 10 milliseconds. Thus, the background task would take $10*(100/1) = 1000$ milliseconds to complete.

Example 3: Suppose in Example 1, an overhead of 1 msec on account of every context switch is to be taken into account. Compute the completion time of T_B.

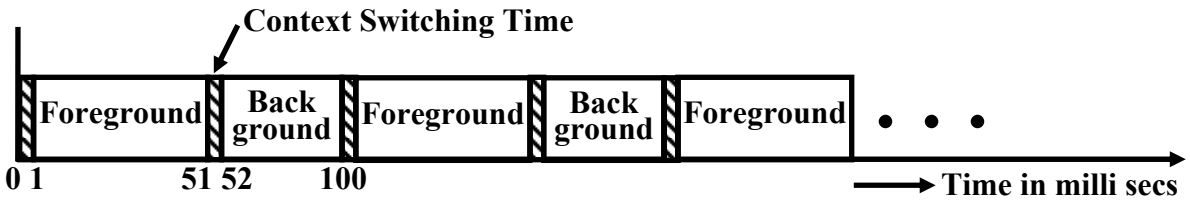


Fig. 30.1 Task Schedule for Example 3

Solution: The very first time the foreground task runs (at time 0), it incurs a context switching overhead of 1 msec. This has been shown as a shaded rectangle in Fig. 30.1. Subsequently each time the foreground task runs, it preempts the background task and incurs one context switch. On completion of each instance of the foreground task, the background task runs and incurs another context switch. With this observation, to simplify our computation of the actual completion time of T_B, we can imagine that the execution time of every foreground task is increased by two context switch times (one due to itself and the other due to the background task running after each time it completes). Thus, the net effect of context switches can be imagined to be causing the execution time of the foreground task to increase by 2 context switch times, i.e. to 52 milliseconds from 50 milliseconds. This has pictorially been shown in Fig. 30.1.

Now, using Expr. 2.7, we get the time required by the background task to complete:

$$1000/(1-52/100) = 2083.4 \text{ milliseconds}$$

In the following two sections, we examine two important event-driven schedulers: EDF (Earliest Deadline First) and RMA (Rate Monotonic Algorithm). EDF is the optimal dynamic priority real-time task scheduling algorithm and RMA is the optimal static priority real-time task scheduling algorithm.

1.4. Earliest Deadline First (EDF) Scheduling

In Earliest Deadline First (EDF) scheduling, at every scheduling point the task having the shortest deadline is taken up for scheduling. This basic principles of this algorithm is very intuitive and simple to understand. The schedulability test for EDF is also simple. A task set is schedulable under EDF, if and only if it satisfies the condition that the total processor utilization due to the task set is less than 1. For a set of periodic real-time tasks $\{T_1, T_2, \ldots, T_n\}$, EDF schedulability criterion can be expressed as:

$$\sum_{i=1}^{n} e_i / p_i = \sum_{i=1}^{n} u_i \leq 1$$

… (3.2/2.8)

where u_i is average utilization due to the task T_i and n is the total number of tasks in the task set. Expr. 3.2 is both a necessary and a sufficient condition for a set of tasks to be EDF schedulable.

EDF has been proven to be an optimal uniprocessor scheduling algorithm. This means that, if a set of tasks is not schedulable under EDF, then no other scheduling algorithm can feasibly schedule this task set. In the simple schedulability test for EDF (Expr. 3.2), we assumed that the period of each task is the same as its deadline. However, in practical problems the period of a task may at times be different from its deadline. In such cases, the schedulability test needs to be changed. If $p_i > d_i$, then each task needs e_i amount of computing time every $min(p_i, d_i)$ duration of time. Therefore, we can rewrite Expr. 3.2 as:

$$\sum_{i=1}^{n} e_i / min(p_i, d_i) \leq 1 \quad \ldots \quad (3.3/2.9)$$

However, if $p_i < d_i$, it is possible that a set of tasks is EDF schedulable, even when the task set fails to meet the Expr 3.3. Therefore, Expr 3.3 is conservative when $p_i < d_i$, and is not a necessary condition, but only a sufficient condition for a given task set to be EDF schedulable.

Example 4: Consider the following three periodic real-time tasks to be scheduled using EDF on a uniprocessor: $T_1 = (e_1=10, p_1=20)$, $T_2 = (e_2=5, p_2=50)$, $T_3 = (e_3=10, p_3=35)$. Determine whether the task set is schedulable.

Solution: The total utilization due to the three tasks is given by:

$$\sum_{i=1}^{3} e_i / p_i = 10/20 + 5/50 + 10/35 = 0.89$$

This is less than 1. Therefore, the task set is EDF schedulable.

Though EDF is a simple as well as an optimal algorithm, it has a few shortcomings which render it almost unusable in practical applications. The main problems with EDF are discussed in Sec. 3.4.3. Next, we discuss the concept of task priority in EDF and then discuss how EDF can be practically implemented.

1.4.1. Is EDF Really a Dynamic Priority Scheduling Algorithm?

We stated in Sec 3.3 that EDF is a dynamic priority scheduling algorithm. Was it after all correct on our part to assert that EDF is a dynamic priority task scheduling algorithm? If EDF were to be considered a dynamic priority algorithm, we should be able determine the precise priority value of a task at any point of time and also be able to show how it changes with time. If we reflect on our discussions of EDF in this section, EDF scheduling does not require any priority value to be computed for any task at any time. In fact, EDF has no notion of a priority value for a task. Tasks are scheduled solely based on the proximity of their deadline. However, the longer a task waits in a ready queue, the higher is the chance (probability) of being taken up for scheduling. So, we can imagine that a virtual priority value associated with a task keeps increasing with time until the task is taken up for scheduling. However, it is important to understand that in EDF the tasks neither have any priority value associated with them, nor does the scheduler perform any priority computations to determine the schedulability of a task at either run time or compile time.

1.4.2. Implementation of EDF

A naive implementation of EDF would be to maintain all tasks that are ready for execution in a queue. Any freshly arriving task would be inserted at the end of the queue. Every node in the

queue would contain the absolute deadline of the task. At every preemption point, the entire queue would be scanned from the beginning to determine the task having the shortest deadline. However, this implementation would be very inefficient. Let us analyze the complexity of this scheme. Each task insertion will be achieved in O(1) or constant time, but task selection (to run next) and its deletion would require O(n) time, where n is the number of tasks in the queue.

A more efficient implementation of EDF would be as follows. EDF can be implemented by maintaining all ready tasks in a sorted priority queue. A sorted priority queue can efficiently be implemented by using a heap data structure. In the priority queue, the tasks are always kept sorted according to the proximity of their deadline. When a task arrives, a record for it can be inserted into the heap in $O(\log_2 n)$ time where n is the total number of tasks in the priority queue. At every scheduling point, the next task to be run can be found at the top of the heap. When a task is taken up for scheduling, it needs to be removed from the priority queue. This can be achieved in O(1) time.

A still more efficient implementation of the EDF can be achieved as follows under the assumption that the number of distinct deadlines that tasks in an application can have are restricted. In this approach, whenever task arrives, its absolute deadline is computed from its release time and its relative deadline. A separate FIFO queue is maintained for each distinct relative deadline that tasks can have. The scheduler inserts a newly arrived task at the end of the corresponding relative deadline queue. Clearly, tasks in each queue are ordered according to their absolute deadlines.

To find a task with the earliest absolute deadline, the scheduler only needs to search among the threads of all FIFO queues. If the number of priority queues maintained by the scheduler is Q, then the order of searching would be O(1). The time to insert a task would also be O(1).

1.4.3. Shortcomings of EDF

In this subsection, we highlight some of the important shortcomings of EDF when used for scheduling real-time tasks in practical applications.

Transient Overload Problem: Transient overload denotes the overload of a system for a very short time. Transient overload occurs when some task takes more time to complete than what was originally planned during the design time. A task may take longer to complete due to many reasons. For example, it might enter an infinite loop or encounter an unusual condition and enter a rarely used branch due to some abnormal input values. When EDF is used to schedule a set of periodic real-time tasks, a task overshooting its completion time can cause some other task(s) to miss their deadlines. It is usually very difficult to predict during program design which task might miss its deadline when a transient overload occurs in the system due to a low priority task overshooting its deadline. The only prediction that can be made is that the task (tasks) that would run immediately after the task causing the transient overload would get delayed and might miss its (their) respective deadline(s). However, at different times a task might be followed by different tasks in execution. However, this lead does not help us to find which task might miss its deadline. Even the most critical task might miss its deadline due to a very low priority task overshooting its planned completion time. So, it should be clear that under EDF any amount of careful design will not guarantee that the most critical task would not miss its deadline under transient overload. This is a serious drawback of the EDF scheduling algorithm.

Resource Sharing Problem: When EDF is used to schedule a set of real-time tasks, unacceptably high overheads might have to be incurred to support resource sharing among the tasks without making tasks to miss their respective deadlines. We examine this issue in some detail in the next lesson.

Efficient Implementation Problem: The efficient implementation that we discussed in Sec. 3.4.2 is often not practicable as it is difficult to restrict the number of tasks with distinct deadlines to a reasonable number. The efficient implementation that achieves O(1) overhead assumes that the number of relative deadlines is restricted. This may be unacceptable in some situations. For a more flexible EDF algorithm, we need to keep the tasks ordered in terms of their deadlines using a priority queue. Whenever a task arrives, it is inserted into the priority queue. The complexity of insertion of an element into a priority queue is of the order $log_2 n$, where n is the number of tasks to be scheduled. This represents a high runtime overhead, since most real-time tasks are periodic with small periods and strict deadlines.

1.5. Rate Monotonic Algorithm(RMA)

We had already pointed out that RMA is an important event-driven scheduling algorithm. This is a static priority algorithm and is extensively used in practical applications. RMA assigns priorities to tasks based on their rates of occurrence. The lower the occurrence rate of a task, the lower is the priority assigned to it. A task having the highest occurrence rate (lowest period) is accorded the highest priority. RMA has been proved to be the optimal static priority real-time task scheduling algorithm.

In RMA, the priority of a task is directly proportional to its rate (or, inversely proportional to its period). That is, the priority of any task T_i is computed as: $priority = k / p_i$, where p_i is the period of the task T_i and k is a constant. Using this simple expression, plots of priority values of tasks under RMA for tasks of different periods can be easily obtained. These plots have been shown in Fig. 30.10(a) and Fig. 30.10(b). It can be observed from these figures that the priority of a task increases linearly with the arrival rate of the task and inversely with its period.

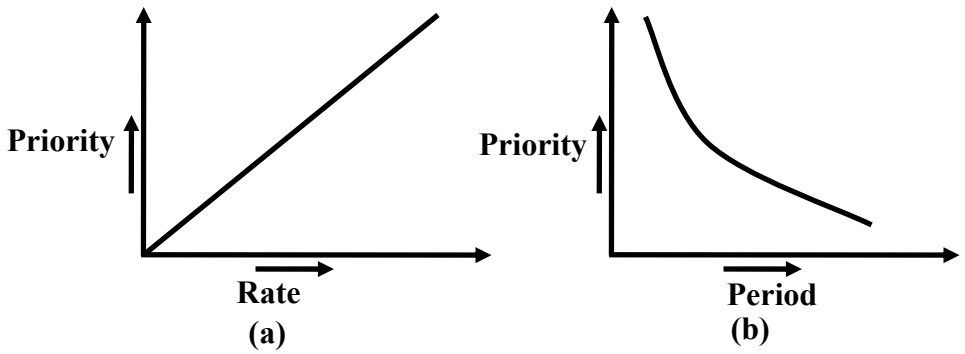


Fig. 30.2 Priority Assignment to Tasks in RMA

1.5.1. Schedulability Test for RMA

An important problem that is addressed during the design of a uniprocessor-based real-time system is to check whether a set of periodic real-time tasks can feasibly be scheduled under RMA. Schedulability of a task set under RMA can be determined from a knowledge of the

worst-case execution times and periods of the tasks. A pertinent question at this point is how can a system developer determine the worst-case execution time of a task even before the system is developed. The worst-case execution times are usually determined experimentally or through simulation studies.

The following are some important criteria that can be used to check the schedulability of a set of tasks set under RMA.

1.5.1.1 Necessary Condition

A set of periodic real-time tasks would not be RMA schedulable unless they satisfy the following necessary condition:

$$\sum_{i=1}^{n} e_i / p_i = \sum_{i=1}^{n} u_i \leq 1$$

where e_i is the worst case execution time and pi is the period of the task T_i, n is the number of tasks to be scheduled, and ui is the CPU utilization due to the task T_i. This test simply expresses the fact that the total CPU utilization due to all the tasks in the task set should be less than 1.

1.5.1.2 Sufficient Condition

The derivation of the sufficiency condition for RMA schedulability is an important result and was obtained by Liu and Layland in 1973. A formal derivation of the Liu and Layland's results from first principles is beyond the scope of this discussion. We would subsequently refer to the sufficiency as the Liu and Layland's condition. A set of n real-time periodic tasks are schedulable under RMA, if

$$\sum_{i=1}^{n} u_i \leq n(2^{1/n} - 1) \qquad (3.4/2.10)$$

where u_i is the utilization due to task T_i. Let us now examine the implications of this result. If a set of tasks satisfies the sufficient condition, then it is guaranteed that the set of tasks would be RMA schedulable.

Consider the case where there is only one task in the system, i.e. n = 1.

Substituting n = 1 in Expr. 3.4, we get,

$$\sum_{i=1}^{1} u_i \leq 1(2^{1/1} - 1) \text{ or } \sum_{i=1}^{1} u_i \leq 1$$

Similarly for n = 2, we get,

$$\sum_{i=1}^{2} u_i \leq 2(2^{1/2} - 1) \text{ or } \sum_{i=1}^{2} u_i \leq 0.828$$

For n = 3, we get,

$$\sum_{i=1}^{3} u_i \leq 3(2^{1/3} - 1) \text{ or } \sum_{i=1}^{3} u_i \leq 0.78$$

For $n \rightarrow \infty$, we get,

$$\sum_{i=1}^{\infty} u_i \leq 3(2^{1/\infty} - 1) \text{ or } \sum_{i=1}^{\infty} u_i \leq \infty.0$$

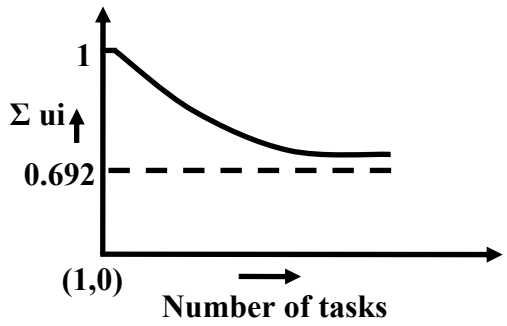


Fig. 30.3 Achievable Utilization with the Number of Tasks under RMA

Evaluation of Expr. 3.4 when $n \rightarrow \infty$ involves an indeterminate expression of the type $\infty.0$. By applying L'Hospital's rule, we can verify that the right hand side of the expression evaluates to $\log_e 2 = 0.692$. From the above computations, it is clear that the maximum CPU utilization that can be achieved under RMA is 1. This is achieved when there is only a single task in the system. As the number of tasks increases, the achievable CPU utilization falls and as $n \rightarrow \infty$, the achievable utilization stabilizes at $\log_e 2$, which is approximately 0.692. This is pictorially shown in Fig. 30.3. We now illustrate the applicability of the RMA schedulability criteria through a few examples.

1.5.2. Examples

Example 5: Check whether the following set of periodic real-time tasks is schedulable under RMA on a uniprocessor: $T_1 = (e_1=20, p_1=100)$, $T_2 = (e_2=30, p_2=150)$, $T_3 = (e_3=60, p_3=200)$.

Solution: Let us first compute the total CPU utilization achieved due to the three given tasks.

$$\sum_{i=1}^{3} u_i = 20/100 + 30/150 + 60/200 = 0.7$$

This is less than 1; therefore the necessary condition for schedulability of the tasks is satisfied. Now checking for the sufficiency condition, the task set is schedulable under RMA if Liu and Layland's condition given by Expr. 3.4 is satisfied Checking for satisfaction of Expr. 3.4, the maximum achievable utilization is given by:

$$3(2^{1/3} - 1) = 0.78$$

The total utilization has already been found to be 0.7. Now substituting these in Liu and Layland's criterion:

$$\sum_{i=1}^{3} u_i \leq 3(2^{1/3} - 1)$$

Therefore, we get $0.7 < 0.78$.
Expr. 3.4, a sufficient condition for RMA schedulability, is satisfied. Therefore, the task set is RMA-schedulable

Example 6: Check whether the following set of three periodic real-time tasks is schedulable under RMA on a uniprocessor: $T_1 = (e_1=20, p_1=100)$, $T_2 = (e_2=30, p_2=150)$, $T_3 = (e_3=90, p_3=200)$.

Solution: Let us first compute the total CPU utilization due to the given task set:

$$\sum_{i=1}^{3} u_i = 20/100 + 30/150 + 90/200 = 0.7$$

Now checking for Liu and Layland criterion:

$$\sum_{i=1}^{3} u_i \leq 0.78$$

Since 0.85 is not ≤ 0.78, the task set is not RMA-schedulable.

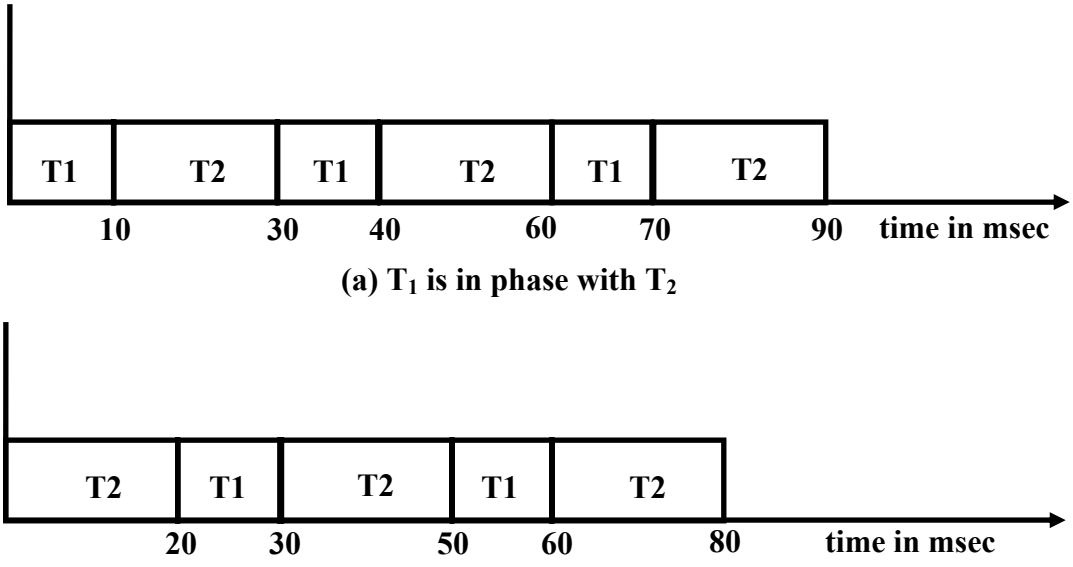
Liu and Layland test (Expr. 2.10) is pessimistic in the following sense.

> If a task set passes the Liu and Layland test, then it is guaranteed to be RMA schedulable. On the other hand, even if a task set fails the Liu and Layland test, it may still be RMA schedulable.

It follows from this that even when a task set fails Liu and Layland's test, we should not conclude that it is not schedulable under RMA. We need to test further to check if the task set is RMA schedulable. A test that can be per- formed to check whether a task set is RMA schedulable when it fails the Liu and Layland test is the Lehoczky's test. Lehoczky's test has been expressed as Theorem 3.

1.5.3. Theorem 3

A set of periodic real-time tasks is RMA schedulable under any task phasing, iff all the tasks meet their respective first deadlines under zero phasing.



(a) T_1 is in phase with T_2

(b) T_1 has a 20 msec phase with respect to T_2

Fig. 30.4 Worst Case Response Time for a Task Occurs When It is in Phase with Its Higher Priority Tasks

A formal proof of this Theorem is beyond the scope of this discussion. However, we provide an intuitive reasoning as to why Theorem 3 must be true. Intuitively, we can understand this result from the following reasoning. First let us try to understand the following fact.

The worst case response time for a task occurs when it is in phase with its higher

To see why this statement must be true, consider the following statement. Under RMA whenever a higher priority task is ready, the lower priority tasks can not execute and have to wait. This implies that, a lower priority task will have to wait for the entire duration of execution of each higher priority task that arises during the execution of the lower priority task. More number of instances of a higher priority task will occur, when a task is in phase with it, when it is in phase with it rather than out of phase with it. This has been illustrated through a simple example in Fig. 30.4. In Fig. 30.4(a), a higher priority task T1=(10,30) is in phase with a lower priority task T2=(60,120), the response time of T2 is 90 msec. However, in Fig. 30.4(b), when T1 has a 20 msec phase, the response time of T2 becomes 80. Therefore, if a task meets its first deadline under zero phasing, then they it will meet all its deadlines.

Example 7: Check whether the task set of Example 6 is actually schedulable under RMA.

Solution: Though the results of Liu and Layland's test were negative as per the results of Example 6, we can apply the Lehoczky test and observe the following:

For the task T1: $e_1 < p_1$ holds since 20 msec < 100 msec. Therefore, it would meet its first deadline (it does not have any tasks that have higher priority).

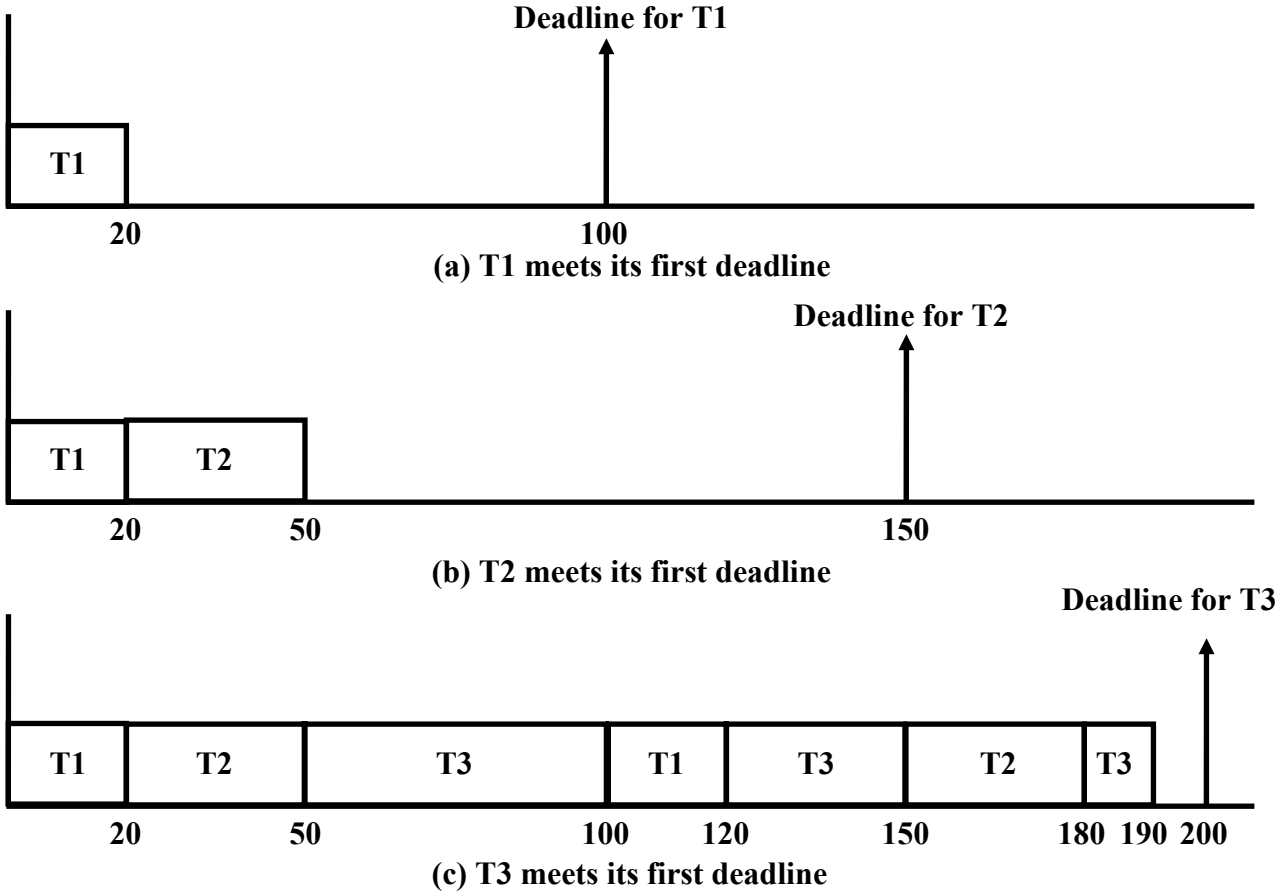


Fig. 30.5 Checking Lehoczky's Criterion for Tasks of Example 7

For the task T_2: T_1 is its higher priority task and considering 0 phasing, it would occur once before the deadline of T_2. Therefore, $(e_1 + e_2) < p_2$ holds, since 20 + 30 = 50 msec < 150 msec. Therefore, T_2 meets its first deadline.

For the task T_3: $(2e_1 + 2e_2 + e_3) < p_3$ holds, since 2∗20 + 2∗30 + 90 = 190msec < 200 msec.

We have considered 2∗e1 and 2∗e2 since T_1 and T_2 occur twice within the first deadline of T_3. Therefore, T_3 meets its first deadline. So, the given task set is schedulable under RMA. The schedulability test for T_3 has pictorially been shown in Fig. 30.5. Since all the tasks meet their first deadlines under zero phasing, they are RMA schedulable according to Lehoczky's results.

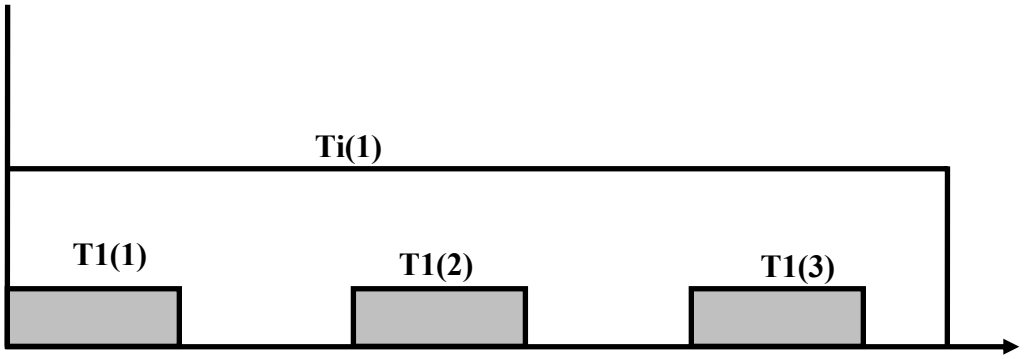


Fig. 30.6 Instances of T_1 over a single instance of T_i

Let us now try to derive a formal expression for this important result of Lehoczky. Let $\{T_1, T_2, \ldots, T_i\}$ be the set of tasks to be scheduled. Let us also assume that the tasks have been ordered in descending order of their priority. That is, task priorities are related as: $pr(T_1) > pr(T_2) > \ldots > pr(T_i)$, where $pr(T_i)$ denotes the priority of the task T_i. Observe that the task T_1 has the highest priority and task T_i has the least priority. This priority ordering can be assumed without any loss of generalization since the required priority ordering among an arbitrary collection of tasks can always be achieved by a simple renaming of the tasks. Consider that the task T_i arrives at the time instant 0. Consider the example shown in Fig. 30.6. During the first instance of the task T_i, three instances of the task T_1 have occurred. Each time T_1 occurs, T_i has to wait since T_1 has higher priority than T_i.

Let us now determine the exact number of times that T_1 occurs within a single instance of T_i. This is given by $\lceil p_i / p_1 \rceil$. Since T_1's execution time is e_1, then the total execution time required due to task T_1 before the deadline of T_i is $\lceil p_i / p_1 \rceil * e_1$. This expression can easily be generalized to consider the execution times all tasks having higher priority than T_i (i.e. T_1, T_2, …, T_{i-1}). Therefore, the time for which T_i will have to wait due to all its higher priority tasks can be expressed as:

$$\sum_{k=1}^{i-1} \lceil p_i / p_k \rceil * e_k \qquad \ldots(3.5/2.11)$$

Expression 3.5 gives the total time required to execute T_i's higher priority tasks for which Ti would have to wait. So, the task T_i would meet its first deadline, iff

$$e_i + \sum_{k=1}^{i-1} \lceil p_i / p_k \rceil * e_k \leq p_i \qquad \ldots(3.6/2.12)$$

That is, if the sum of the execution times of all higher priority tasks occurring before T_i's first deadline, and the execution time of the task itself is less than its period p_i, then T_i would complete before its first deadline. Note that in Expr. 3.6, we have implicitly assumed that the

task periods equal their respective deadlines, i.e. $p_i = d_i$. If $p_i < d_i$, then the Expr. 3.6 would need modifications as follows.

$$e_i + \sum_{k=1}^{i-1} \lceil d_i / p_k \rceil * e_k \leq d_i \quad \ldots(3.7/2.13)$$

Note that even if Expr. 3.7 is not satisfied, there is some possibility that the task set may still be schedulable. This might happen because in Expr. 3.7 we have considered zero phasing among all the tasks, which is the worst case. In a given problem, some tasks may have non-zero phasing. Therefore, even when a task set narrowly fails to meet Expr 3.7, there is some chance that it may in fact be schedulable under RMA. To understand why this is so, consider a task set where one particular task Ti fails Expr. 3.7, making the task set not schedulable. The task misses its deadline when it is in phase with all its higher priority task. However, when the task has non-zero phasing with at least some of its higher priority tasks, the task might actually meet its first deadline contrary to any negative results of the expression 3.7.

Let us now consider two examples to illustrate the applicability of the Lehoczky's results.

Example 8: Consider the following set of three periodic real-time tasks: T_1=(10,20), T_2=(15,60), T_3=(20,120) to be run on a uniprocessor. Determine whether the task set is schedulable under RMA.

Solution: First let us try the sufficiency test for RMA schedulability. By Expr. 3.4 (Liu and Layland test), the task set is schedulable if $\sum u_i \leq 0.78$.

$$\sum u_i = 10/20 + 15/60 + 20/120 = 0.91$$

This is greater than 0.78. Therefore, the given task set fails Liu and Layland test. Since Expr. 3.4 is a pessimistic test, we need to test further.

Let us now try Lehoczky's test. All the tasks T_1, T_2, T_3 are already ordered in decreasing order of their priorities.

Testing for task T_1:

Since e1 (10 msec) is less than d1 (20 msec), T_1 would meet its first deadline.

Testing for task T_2:

$$15 + \lceil 60/20 \rceil * 10 \leq 60 \text{ or } 15 + 30 = 45 \leq 60 \text{ msec}$$

The condition is satisfied. Therefore, T_2 would meet its first deadline.

Testing for Task T3:

$$20 + \lceil 120/20 \rceil * 10 + \lceil 120/60 \rceil * 15 = 20 + 60 + 30 = 110 \text{ msec}$$

This is less than T3's deadline of 120. Therefore T_3 would meet its first deadline.

Since all the three tasks meet their respective first deadlines, the task set is RMA schedulable according to Lehoczky's results.

Example 9: RMA is used to schedule a set of periodic hard real-time tasks in a system. Is it possible in this system that a higher priority task misses its deadline, whereas a lower priority task meets its deadlines? If your answer is negative, prove your denial. If your answer is affirmative, give an example involving two or three tasks scheduled using RMA where the lower priority task meets all its deadlines whereas the higher priority task misses its deadline.

Solution: Yes. It is possible that under RMA a higher priority task misses its deadline where as a lower priority task meets its deadline. We show this by constructing an example. Consider the following task set: T_1 = (e_1=15, p_1=20), T_2 = (e_2=6, p_2=35), T_3 = (e_3=3, p_3=100). For the given task set, it is easy to observe that $pr(T_1) > pr(T_2) > pr(T_3)$. That is, T1, T2, T3 are ordered in decreasing order of their priorities.

For this task set, T3 meets its deadline according to Lehoczky's test since

$$e_3 + \lceil p_3 / p_2 \rceil * e_2 + \lceil p_3 / p_1 \rceil * e_1 = 3 + (\lceil 100/35 \rceil * 6) + (\lceil 100/20 \rceil * 15)$$

$$= 3 + (3 * 6) + (5 * 15) = 96 \leq 100 \text{ msec.}$$

But, T_2 does not meet its deadline since

$$e_2 + \lceil p_2 / p_1 \rceil * e_1 = 6 + (\lceil 35/20 \rceil * 15) = 6 + (2 * 15) = 36 \text{ msec.}$$

This is greater than the deadline of T_2 (35 msec).

As a consequence of the results of Example 9, by observing that the lowest priority task of a given task set meets its first deadline, we can not conclude that the entire task set is RMA schedulable. On the contrary, it is necessary to check each task individually as to whether it meets its first deadline under zero phasing. If one finds that the lowest priority task meets its deadline, and concludes that the entire task set would be feasibly scheduled under RMA, he is likely to be flawed.

1.5.4. Achievable CPU Utilization

Liu and Layland's results (Expr. 3.4) bounded the CPU utilization below which a task set would be schedulable. It is clear from Expr. 3.4 and Fig. 30.10 that the Liu and Layland schedulability criterion is conservative and restricts the maximum achievable utilization due to any task set which can be feasibly scheduled under RMA to 0.69 when the number of tasks in the task set is large. However, (as you might have already guessed) this is a pessimistic figure. In fact, it has been found experimentally that for a large collection of tasks with independent periods, the maximum utilization below which a task set can feasibly be scheduled is on the average close to 88%.

For harmonic tasks, the maximum achievable utilization (for a task set to have a feasible schedule) can still be higher. In fact, if all the task periods are harmonically related, then even a task set having 100% utilization can be feasibly scheduled. Let us first understand when are the periods of a task set said to be harmonically related. The task periods in a task set are said to be harmonically related, iff for any two arbitrary tasks T_i and T_k in the task set, whenever $p_i > p_k$, it should imply that p_i is an integral multiple of p_k. That is, whenever $p_i > p_k$, it should be possible to express p_i as $n * p_k$ for some integer $n > 1$. In other words, p_k should squarely divide p_i. An example of a harmonically related task set is the following: T_1 = (5, 30), T2 = (8, 120), T3 = (12, 60).

It is easy to prove that a harmonically related task set with even 100% utilization can feasibly be scheduled.

1.5.5. Theorem 4

For a set of harmonically related tasks HS = {T_i}, the RMA schedulability criterion is given by $\sum_{i=1}^{n} u_i \leq 1$.

Proof: Let us assume that T_1, T_2, …, T_n be the tasks in the given task set. Let us further assume that the tasks in the task set T_1, T_2, …, T_n have been arranged in increasing order of their periods. That is, for any i and j, $pi < pj$ whenever $i < j$. If this relationship is not satisfied, then a simple renaming of the tasks can achieve this. Now, according to Expr. 3.6, a task T_i meets its deadline, if $e_i + \sum_{k=1}^{i-1} \lceil p_i / p_k \rceil * e_k \leq p_i$.

However, since the task set is harmonically related, p_i can be written as $m * p_k$ for some m. Using this, $\lceil p_i / p_k \rceil = p_i / p_k$. Now, Expr. 3.6 can be written as:

$$e_i + \sum_{k=1}^{i-1} (p_i / p_k) * e_k \leq p_i$$

For $T_i = T_n$, we can write, $e_n + \sum_{k=1}^{n-1} (p_n / p_k) * e_k \leq p_n$.

Dividing both sides of this expression by p_n, we get the required result.

Hence, the task set would be schedulable iff $\sum_{k=1}^{n} e_k / p_k \leq 1$ or $\sum_{i=1}^{n} u_i \leq 1$.

1.5.6. Advantages and Disadvantages of RMA

In this section, we first discuss the important advantages of RMA over EDF. We then point out some disadvantages of using RMA. As we had pointed out earlier, RMA is very commonly used for scheduling real-time tasks in practical applications. Basic support is available in almost all commercial real-time operating systems for developing applications using RMA. RMA is simple and efficient. RMA is also the optimal static priority task scheduling algorithm. Unlike EDF, it requires very few special data structures. Most commercial real-time operating systems support real-time (static) priority levels for tasks. Tasks having real-time priority levels are arranged in multilevel feedback queues (see Fig. 30.7). Among the tasks in a single level, these commercial real-time operating systems generally provide an option of either time-slicing and round-robin scheduling or FIFO scheduling.

RMA Transient Overload Handling: RMA possesses good transient overload handling capability. Good transient overload handling capability essentially means that, when a lower priority task does not complete within its planned completion time, it can not make any higher priority task to miss its deadline. Let us now examine how transient overload would affect a set of tasks scheduled under RMA. Will a delay in completion by a lower priority task affect a higher priority task? The answer is: ‘No’. A lower priority task even when it exceeds its planned execution time cannot make a higher priority task wait according to the basic principles of RMA – whenever a higher priority task is ready, it preempts any executing lower priority task. Thus, RMA is stable under transient overload and a lower priority task overshooting its completion time can not make a higher priority task to miss its deadline.

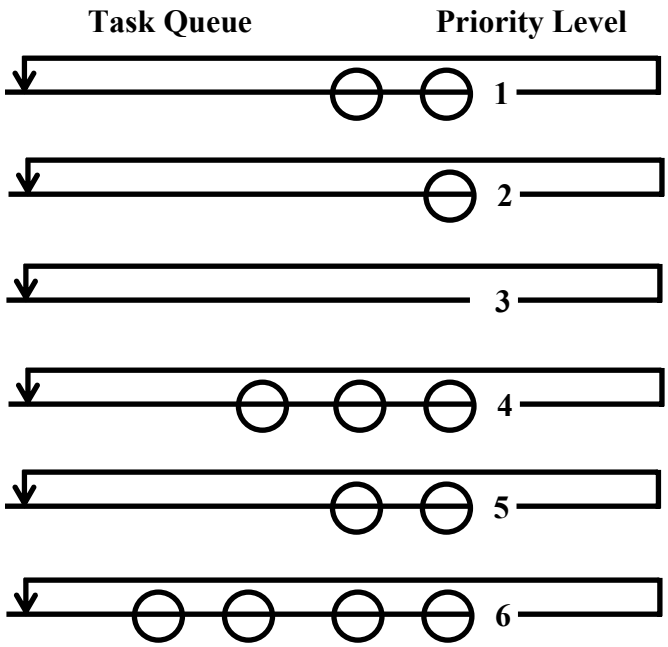


Fig. 30.7 Multi-Level Feedback Queue

The disadvantages of RMA include the following: It is very difficult to support aperiodic and sporadic tasks under RMA. Further, RMA is not optimal when task periods and deadlines differ.

1.6. Deadline Monotonic Algorithm (DMA)

RMA no longer remains an optimal scheduling algorithm for the periodic real-time tasks, when task deadlines and periods differ (i.e. $d_i \neq p_i$) for some tasks in the task set to be scheduled. For such task sets, Deadline Monotonic Algorithm (DMA) turns out to be more proficient than RMA. DMA is essentially a variant of RMA and assigns priorities to tasks based on their deadlines, rather than assigning priorities based on task periods as done in RMA. DMA assigns higher priorities to tasks with shorter deadlines. When the relative deadline of every task is proportional to its period, RMA and DMA produce identical solutions. When the relative deadlines are arbitrary, DMA is more proficient than RMA in the sense that it can sometimes produce a feasible schedule when RMA fails. On the other hand, RMA always fails when DMA fails. We now illustrate our discussions using an example task set that is DMA schedulable but not RMA schedulable.

Example 10: Is the following task set schedulable by DMA? Also check whether it is schedulable using RMA. T_1 = (e_1=10, p_1=50, d_1=35), T_2 = (e_2=15, p_2=100, d_1=20), T_3 = (e_3=20, p_3=200, d_1=200) [time in msec].

Solution: First, let us check RMA schedulability of the given set of tasks, by checking the Lehoczky's criterion. The tasks are already ordered in descending order of their priorities.
Checking for T_1:
10 msec < 35 msec. Hence, T_1 would meet its first deadline.
Checking for T_2:

(10 + 15) > 20 (exceeds deadline)

Thus, T_2 will miss its first deadline. Hence, the given task set can not be feasibly scheduled under RMA.

Now let us check the schedulability using DMA:

Under DMA, the priority ordering of the tasks is as follows: $pr(T_2) > pr(T_1) > pr(T_3)$.

Checking for T_2:

15 msec < 20 msec. Hence, T_2 will meet its first deadline.

Checking for T_1:

$$(15 + 10) < 35$$

Hence T_1 will meet its first deadline.

Checking for T_3:

$$(20 + 30 + 40) < 200$$

Therefore, T_3 will meet its deadline.

Therefore, the given task set is schedulable under DMA but not under RMA.

1.7. Context Switching Overhead

So far, while determining schedulability of a task set, we had ignored the overheads incurred on account of context switching. Let us now investigate the effect of context switching overhead on schedulability of tasks under RMA.

It is easy to realize that under RMA, whenever a task arrives, it preempts at most one task – the task that is currently running. From this observation, it can be concluded that in the worst-case, each task incurs at most two context switches under RMA. One when it preempts the currently running task. And the other when it completes possibly the task that was preempted or some other task is dispatched to run. Of course, a task may incur just one context switching overhead, if it does not preempt any task. For example, it arrives when the processor is idle or when a higher priority task was running. However, we need to consider two context switches for every task, if we try to determine the worst-case context switching overhead.

For simplicity we can assume that context switching time is constant, and equals 'c' milliseconds where 'c' is a constant. From this, it follows that the net effect of context switches is to increase the execution time e_i of each task T_i to at most $e_i + 2*c$. It is therefore clear that in order to take context switching time into consideration, in all schedulability computations, we need to replace e_i by $e_i + 2*c$ for each T_i.

Example 11: Check whether the following set of periodic real-time tasks is schedulable under RMA on a uniprocessor: $T_1 = (e_1=20, p_1=100)$, $T_2 = (e_2=30, p_2=150)$, $T_3 = (e_3=90, p_3=200)$. Assume that context switching overhead does not exceed 1 msec, and is to be taken into account in schedulability computations.

Solution: The net effect of context switches is to increase the execution time of each task by two context switching times. Therefore, the utilization due to the task set is:

$$\sum_{i=1}^{3} u_i = 22/100 + 32/150 + 92/200 = 0.89$$

Since $\sum_{i=1}^{3} u_i > 0.78$, the task set is not RMA schedulable according to the Liu and Layland test.

Let us try Lehoczky's test. The tasks are already ordered in descending order of their priorities.

Checking for task T_1:

$$22 < 100$$

The condition is satisfied; therefore T_1 meets its first deadline.
Checking for task T_2:

$$(22*2) + 32 < 150$$

The condition is satisfied; therefore T_2 meets its first deadline.
Checking for task T_3:

$$(22*2) + (32*2) + 90 < 200.$$

The condition is satisfied; therefore T_3 meets its first deadline.
Therefore, the task set can be feasibly scheduled under RMA even when context switching overhead is taken into consideration.

1.8. Self Suspension

A task might cause its self-suspension, when it performs its input/output operations or when it waits for some events/conditions to occur. When a task self suspends itself, the operating system removes it from the ready queue, places it in the blocked queue, and takes up the next eligible task for scheduling. Thus, self-suspension introduces an additional scheduling point, which we did not consider in the earlier sections. Accordingly, we need to augment our definition of a scheduling point given in Sec. 2.3.1 (lesson 2).

> In event-driven scheduling, the scheduling points are defined by task completion, task arrival, and self-suspension events.

Let us now determine the effect of self-suspension on the schedulability of a task set. Let us consider a set of periodic real-time tasks $\{T_1, T_2, \ldots, T_n\}$, which have been arranged in the increasing order of their priorities (or decreasing order of their periods). Let the worst case self-suspension time of a task T_i is b_i. Let the delay that the task T_i might incur due to its own self-suspension and the self-suspension of all higher priority tasks be bt_i. Then, bt_i can be expressed as:

$$bt_i = b_i + \sum_{k=1}^{i-1} min(e_k, b_k) \qquad \ldots(3.8/2.15)$$

Self-suspension of a higher priority task T_k may affect the response time of a lower priority task T_i by as much as its execution time e_k if $e_k < b_k$. This worst case delay might occur when the higher priority task after self-suspension starts its execution exactly at the time instant the lower priority task would have otherwise executed. That is, after self-suspension, the execution of the higher priority task overlaps with the lower priority task, with which it would otherwise not have overlapped. However, if $e_k > b_k$, then the self suspension of a higher priority task can delay a lower priority task by at most b_k, since the maximum overlap period of the execution of a higher priority task due to self-suspension is restricted to b_k.

Note that in a system where some of the tasks are non preemptable, the effect of self suspension is much more severe than that computed by Expr.3.8. The reason is that, every time a processor self suspends itself, it loses the processor. It may be blocked by a non-preemptive lower priority task after the completion of self-suspension. Thus, in a non-preemptable scenario, a task incurs delays due to self-suspension of itself and its higher priority tasks, and also the delay caused due to non-preemptable lower priority tasks. Obviously, a task can not get delayed due to the self-suspension of a lower priority non-preemptable task.

The RMA task schedulability condition of Liu and Layland (Expr. 3.4) needs to change when we consider the effect of self-suspension of tasks. To consider the effect of self-suspension in Expr. 3.4, we need to substitute e_i by $(e_i + bt_i)$. If we consider the effect of self-suspension on task completion time, the Lehoczky criterion (Expr. 3.6) would also have to be generalized:

$$e_i + bt_i + \sum_{k=1}^{i-1} \lceil p_i / p_k \rceil * e_k \leq p_i \quad \ldots (3.9/2.16)$$

We have so far implicitly assumed that a task undergoes at most a single self suspension. However, if a task undergoes multiple self-suspensions, then expression 3.9 we derived above, would need to be changed. We leave this as an exercise for the reader.

Example 14: Consider the following set of periodic real-time tasks: $T_1 = (e_1=10, p_1=50)$, $T_2 = (e_2=25, p_2=150)$, $T_3 = (e_3=50, p_3=200)$ [all in msec]. Assume that the self-suspension times of T_1, T_2, and T_3 are 3 msec, 3 msec, and 5 msec, respectively. Determine whether the tasks would meet their respective deadlines, if scheduled using RMA.

Solution: The tasks are already ordered in descending order of their priorities. By using the generalized Lehoczky's condition given by Expr. 3.9, we get:
For T_1 to be schedulable:

$$(10 + 3) < 50$$

Therefore T_1 would meet its first deadline.
For T_2 to be schedulable:

$$(25 + 6 + 10*3) < 150$$

Therefore, T_2 meets its first deadline.
For T_3 to be schedulable:

$$(50 + 11 + (10*4 + 25*2)) < 200$$

This inequality is also satisfied. Therefore, T_3 would also meet its first deadline.
It can therefore be concluded that the given task set is schedulable under RMA even when self-suspension of tasks is considered.

1.9. Self Suspension with Context Switching Overhead

Let us examine the effect of context switches on the generalized Lehoczky's test (Expr.3.9) for schedulability of a task set, which takes self-suspension by tasks into account. In a fixed priority preemptable system, each task preempts at most one other task if there is no self suspension. Therefore, each task suffers at most two context switches – one context switch when it starts and another when it completes. It is easy to realize that any time when a task self-suspends, it causes at most two additional context switches. Using a similar reasoning, we can determine that when each task is allowed to self-suspend twice, additional four context switching overheads are incurred. Let us denote the maximum context switch time as *c*. The effect of a single self-suspension of tasks is to effectively increase the execution time of each task T_i in the worst case from e_i to $(e_i + 4*c)$. Thus, context switching overhead in the presence of a single self-suspension of tasks can be taken care of by replacing the execution time of a task T_i by $(e_i + 4*c)$ in Expr. 3.9. We can easily extend this argument to consider two, three, or more self-suspensions.

1.10. Exercises

1. State whether the following assertions are True or False. Write one or two sentences to justify your choice in each case.
   a. When RMA is used for scheduling a set of hard real-time periodic tasks, the upper bound on achievable utilization improves as the number in tasks in the system being developed increases.

b. If a set of periodic real-time tasks fails Lehoczky's test, then it can safely be concluded that this task set can not be feasibly scheduled under RMA.
c. A time-sliced round-robin scheduler uses preemptive scheduling.
d. RMA is an optimal static priority scheduling algorithm to schedule a set of periodic real-time tasks on a non-preemptive operating system.
e. Self-suspension of tasks impacts the worst case response times of the individual tasks much more adversely when preemption of tasks is supported by the operating system compared to the case when preemption is not supported.
f. When a set of periodic real-time tasks is being scheduled using RMA, it can not be the case that a lower priority task meets its deadline, whereas some higher priority task does not.
g. EDF (Earliest Deadline First) algorithm possesses good transient overload handling capability.
h. A time-sliced round robin scheduler is an example of a non-preemptive scheduler.
i. EDF algorithm is an optimal algorithm for scheduling hard real-time tasks on a uniprocessor when the task set is a mixture of periodic and aperiodic tasks.
j. In a non-preemptable operating system employing RMA scheduling for a set of real-time periodic tasks, self-suspension of a higher priority task (due to I/O etc.) may increase the response time of a lower priority task.
k. The worst-case response time for a task occurs when it is out of phase with its higher priority tasks.
l. Good real-time task scheduling algorithms ensure fairness to real-time tasks while scheduling.

2. State whether the following assertions are True or False. Write one or two sentences to justify your choice in each case.
   a. The EDF algorithm is optimal for scheduling real-time tasks in a uniprocessor in a non-preemptive environment.
   b. When RMA is used to schedule a set of hard real-time periodic tasks in a uniprocessor environment, if the processor becomes overloaded any time during system execution due to overrun by the lowest priority task, it would be very difficult to predict which task would miss its deadline.
   c. While scheduling a set of real-time periodic tasks whose task periods are harmonically related, the upper bound on the achievable CPU utilization is the same for both EDF and RMA algorithms.
   d. In a non-preemptive event-driven task scheduler, scheduling decisions are made only at the arrival and completion of tasks.
   e. The following is the correct arrangement of the three major classes of real-time scheduling algorithms in ascending order of their run-time overheads.
      - static priority preemptive scheduling algorithms
      - table-driven algorithms
      - dynamic priority algorithms
   f. While scheduling a set of independent hard real-time periodic tasks on a uniprocessor, RMA can be as proficient as EDF under some constraints on the task set.
   g. RMA should be preferred over the time-sliced round-robin algorithm for scheduling a set of soft real-time tasks on a uniprocessor.

h. Under RMA, the achievable utilization of a set of hard real-time periodic tasks would drop when task periods are multiples of each other compared to the case when they are not.

i. RMA scheduling of a set of real-time periodic tasks using the Liu and Layland criterion might produce infeasible schedules when the task periods are different from the task deadlines.

3. What do you understand by scheduling point of a task scheduling algorithm? How are the scheduling points determined in (i) clock-driven, (ii) event-driven, (iii) hybrid schedulers? How will your definition of scheduling points for the three classes of schedulers change when (a) self-suspension of tasks, and (b) context switching overheads of tasks are taken into account.
4. What do you understand by jitter associated with a periodic task? How are these jitters caused?
5. Is EDF algorithm used for scheduling real-time tasks a dynamic priority scheduling algorithm? Does EDF compute any priority value of tasks any time? If you answer affirmatively, then explain when is the priority computed and how is it computed. If you answer in negative, then explain the concept of priority in EDF.
6. What is the sufficient condition for EDF schedulability of a set of periodic tasks whose period and deadline are different? Construct an example involving a set of three periodic tasks whose period differ from their respective deadlines such that the task set fails the sufficient condition and yet is EDF schedulable. Verify your answer. Show all your intermediate steps.
7. A preemptive static priority real-time task scheduler is used to schedule two periodic tasks T1 and T2 with the following characteristics:

Task	**Phase** mSec	**Execution Time** mSec	**Relative Deadline** mSec	**Period** mSec
T_1	0	10	20	20
T_2	0	20	50	50

Assume that T1 has higher priority than T2. A background task arrives at time 0 and would require 1000mSec to complete. Compute the completion time of the background task assuming that context switching takes no more than 0.5 mSec.

8. Assume that a preemptive priority-based system consists of three periodic foreground tasks T1, T2, and T3 with the following characteristics:

Task	**Phase** mSec	**Execution Time** mSec	**Relative Deadline** mSec	**Period** mSec
T_1	0	20	100	100
T_2	0	30	150	150
T_3	0	30	300	300

T1 has higher priority than T2 and T2 has higher priority than T3. A background task Tb arrives at time 0 and would require 2000mSec to complete. Compute the completion time of the background task Tb assuming that context switching time takes no more than 1 mSec.

9. Consider the following set of four independent real-time periodic tasks.

Task	**Start Time** msec	**Processing Time** msec	**Period** msec
T_1	20	25	150
T_2	40	10	50
T_3	20	15	50
T_4	60	50	200

Assume that task T3 is more critical than task T2. Check whether the task set can be feasibly scheduled using RMA.

10. What is the worst case response time of the background task of a system in which the background task requires 1000 msec to complete? There are two foreground tasks. The higher priority foreground task executes once every 100mSec and each time requires 25mSec to complete. The lower priority foreground task executes once every 50 msec and requires 15 msec to complete. Context switching requires no more than 1 msec.
11. Construct an example involving more than one hard real-time periodic task whose aggregate processor utilization is 1, and yet schedulable under RMA.
12. Determine whether the following set of periodic tasks is schedulable on a uniprocessor using DMA (Deadline Monotonic Algorithm). Show all intermediate steps in your computation.

Task	**Start Time** mSec	**Processing Time** mSec	**Period** mSec	**Deadline** mSec
T_1	20	25	150	140
T_2	60	10	60	40
T_3	40	20	200	120
T_4	25	10	80	25

13. Consider the following set of three independent real-time periodic tasks.

Task	**Start Time** mSec	**Processing Time** mSec	**Period** mSec	**Deadline** mSec
T_1	20	25	150	100
T_2	60	10	50	30
T_3	40	50	200	150

Determine whether the task set is schedulable on a uniprocessor using EDF. Show all intermediate steps in your computation.

14. Determine whether the following set of periodic real-time tasks is schedulable on a uniprocessor using RMA. Show the intermediate steps in your computation. Is RMA optimal when the task deadlines differ from the task periods?

Task	Start Time mSec	Processing Time mSec	Period mSec	Deadline mSec
T_1	20	25	150	100
T_2	40	7	40	40
T_3	60	10	60	50
T_4	25	10	30	20

15. Construct an example involving two periodic real-time tasks which can be feasibly scheduled by both RMA and EDF, but the schedule generated by RMA differs from that generated by EDF. Draw the two schedules on a time line and highlight how the two schedules differ. Consider the two tasks such that for each task:
    a. the period is the same as deadline
    b. period is different from deadline
16. Can multiprocessor real-time task scheduling algorithms be used satisfactorily in distributed systems. Explain the basic difference between the characteristics of a real-time task scheduling algorithm for multiprocessors and a real-time task scheduling algorithm for applications running on distributed systems.
17. Construct an example involving a set of hard real-time periodic tasks that are not schedulable under RMA but could be feasibly scheduled by DMA. Verify your answer, showing all intermediate steps.
18. Three hard real-time periodic tasks T_1 = (50, 100, 100), T_2 = (70, 200, 200), and T_3 = (60, 400, 400) [time in msec] are to be scheduled on a uniprocessor using RMA. Can the task set be feasibly be scheduled? Suppose context switch overhead of 1 millisecond is to be taken into account, determine the schedulability.
19. Consider the following set of three real-time periodic tasks.

Task	Start Time mSec	Processing Time mSec	Period mSec	Deadline mSec
T_1	20	25	150	100
T_2	40	10	50	50
T_3	60	50	200	200

    a. Check whether the three given tasks are schedulable under RMA. Show all intermediate steps in your computation.
    b. Assuming that each context switch incurs an overhead of 1 msec, determine whether the tasks are schedulable under RMA. Also, determine the average context switching overhead per unit of task execution.
    c. Assume that T1, T2, and T3 self-suspend for 10 msec, 20 msec, and 15 msec respectively. Determine whether the task set remains schedulable under RMA. The context switching overhead of 1 msec should be considered in your result. You can assume that each task undergoes self-suspension only once during each of its execution.
    d. Assuming that T1 and T2 are assigned the same priority value, determine the additional delay in response time that T2 would incur compared to the case when they are assigned distinct priorities. Ignore the self-suspension times and the context switch overhead for this part of the question.