Real-Time Task Scheduling

In the last chapter we defined a real-time task as one that has some constraints associated with it. Out of the three broad classes of time constraints we discussed, deadline constraint on tasks is the most common. In all subsequent discussions we therefore implicitly assume only deadline constraints on real-time tasks, unless we mention otherwise.

Real-time tasks get generated in response to some events that may either be external or internal to the system. For example, a task might get generated due to an internal event such as a clock interrupt occurring every few milli seconds to periodically poll the temperature of a chemical plant. Another task might get generated due to an external event such as the user pressing a switch. When a task gets generated, it is said to have *arrived* or *got released*. Every real-time system usually consists of a number of real-time tasks. The time bounds on different tasks may be different. We had already pointed out that the consequences of a task missing its time bounds may also vary from task to task. This is often expressed as the criticality of a task.

In the last chapter, we had pointed out that appropriate scheduling of tasks is the basic mechanism adopted by a real-time operating system to meet the time constraints of a task. Therefore, selection of an appropriate task scheduling algorithm is central to the proper functioning of a real-time system. In this chapter we discuss some fundamental task scheduling techniques that are available. An understanding of these techniques would help us not only to satisfactorily design a real-time application, but also understand and appreciate the features of modern commercial real-time operating systems discussed in later chapters.

This chapter is organized as follows. We first introduce some basic concepts and terminologies associated with task scheduling. Subsequently, we discuss two major classes of task schedulers: clock-driven and event-driven. Finally, we explain some important issues that must be considered while developing practical applications.

1 Some Important Concepts

In this section we introduce a few important concepts and terminologies which would be useful in understanding the rest of this chapter.

Task Instance: Each time an event occurs, it triggers the task that handles this event to run. In other words, a task is generated when some specific event occurs. Real-time tasks therefore normally recur a large number of times at different instants of time depending on the event occurrence times. It is possible that real-time tasks recur at random instants. However, most real-time tasks recur with certain fixed periods. For example, a temperature sensing task in a chemical plant might recur indefinitely with a certain period because the temperature is sampled periodically, whereas a task handling a device interrupt might recur at random instants. Each time a task recurs, it is called an *instance* of the task. The first time a task occurs, it is called the first instance of the task. The next occurrence of the task is called its second instance, and so on. The jth instance of a task T_i would be denoted as $T_i(j)$. Each instance of a real-time task is associated with a deadline by which it needs to complete and produce results. We shall at times refer to task instances as processes and use these two terms interchangeably when no confusion arises.

Relative Deadline versus Absolute Deadline: The absolute deadline of a task is the absolute time value (counted from time 0) by which the results from the task are expected. Thus, absolute deadline is equal to the

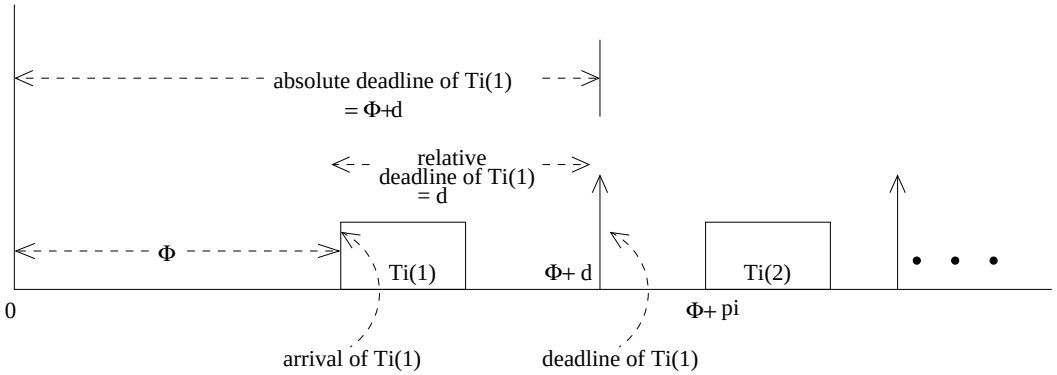


Figure 1: Relative and Absolute Deadlines of a Task

interval of time between the time 0 and the actual instant at which the deadline occurs as measured by some physical clock. Whereas, relative deadline is the time interval between the start of the task and the instant at which deadline occurs. In other words, relative deadline is the time interval between the arrival of a task and the corresponding deadline. The difference between relative and absolute deadlines is illustrated in Fig. 1. It can be observed from Fig. 1 that the relative deadline of the task Ti(1) is d, whereas its absolute deadline is $\phi + d$.

Response Time: The response time of a task is the time it takes (as measured from the task arrival time) for the task to produce its results. As already remarked, task instances get generated due to occurrence of events. These events may be internal to the system, such as clock interrupts, or external to the system such as a robot encountering an obstacle.

The response time is the time duration from the occurrence of the event generating the task to the time the task produces its results.

For hard real-time tasks, as long as all their deadlines are met, there is no special advantage of completing the tasks early. However, for soft real-time tasks, average response time of tasks is an important metric to measure the performance of a scheduler. A scheduler for soft real-time tasks should try to execute the tasks in an order that minimizes the average response time of tasks.

Task Precedence. A task is said to precede another task, if the first task must complete before the second task can start. When a task T_i precedes another task T_j, then each instance of T_i precedes the corresponding instance of T_j. That is, if T_1 precedes T_2, then $T_1(1)$ precedes $T_2(1)$, $T_1(2)$ precedes $T_2(2)$, and so on. A precedence order defines a partial order among tasks. Recollect from a first course on discrete mathematics that a partial order relation is reflexive, antisymmetric, and transitive. An example partial ordering among tasks is shown in Fig. 2. Here T_1 precedes T_2, but we cannot relate T_1 with either T_3 or T_4. We shall later use task precedence relation to develop appropriate task scheduling algorithms.

Data Sharing: Tasks often need to share their results among each other when one task needs to share the results produced by another task; clearly, the second task must precede the first task. In fact, precedence relation between two tasks sometimes implies data sharing between the two tasks (e.g. first task passing some results to the second task). However, this is not always true. A task may be required to precede another even when there is no data sharing. For example, in a chemical plant it may be required that the reaction chamber must be filled with water before chemicals are introduced. In this case, the task handling filling up the reaction chamber with water must complete, before the task handling introduction of the chemicals is activated. It is therefore not appropriate to

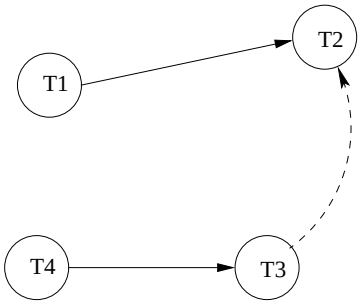


Figure 2: Precedence Relation Among Tasks

represent data sharing using precedence relation. Further, data sharing may occur not only when one task precedes the other, but might occur among truly concurrent tasks, and overlapping tasks. In other words, data sharing among tasks does not necessarily impose any particular ordering among tasks. Therefore, data sharing relation among tasks needs to be represented using a different symbol. We shall represent data sharing among two tasks using a dashed arrow. In the example of data sharing among tasks represented in Fig. 2, T_2 uses the results of T_3, but T_2 and T_3 may execute concurrently. T_2 may even start executing first, after sometimes it may receive some data from T_3, and continue its execution, and so on.

2 Types of Real-Time Tasks and Their Characteristics

Based on the way real-time tasks recur over a period of time, it is possible to classify them into three main categories: periodic, sporadic, and aperiodic tasks. In the following, we discuss the important characteristics of these three major categories of real-time tasks.

Periodic Tasks: A periodic task is one that repeats after a certain fixed time interval. The precise time instants at which periodic tasks recur are usually demarcated by clock interrupts. For this reason, periodic tasks are sometimes referred to as clock-driven tasks. The fixed time interval after which a task repeats is called the *period* of the task. If T_i is a periodic task, then the time from 0 till the occurrence of the first instance of T_i (i.e. $T_i(1)$) is denoted by ϕ_i; and is called the phase of the task. The second instance (i.e. $T_i(2)$) occurs at $\phi_i + p_i$. The third instance (i.e. $T_i(3)$) occurs at $\phi_i + 2 * p_i$ and so on. Formally, a periodic task T_i can be represented by a 4 tuple (ϕ_i, p_i, e_i, d_i) where p_i is the period of task, e_i is the worst case execution time of the task, and d_i is the relative deadline of the task. We shall use this notation extensively in the remainder of this book.

To illustrate the above notation to represent real-time periodic tasks, let us consider the track correction task typically found in a rocket control software. Assume the following characteristics of the track correction task. The track correction task starts 2000 milli Seconds after the launch of the rocket, and it periodically recurs every 50 milli Seconds then on. Each instance of the task requires a processing time of 8 milli Seconds and its relative deadline is 50 milli Seconds. Recall that the phase of a task is defined by the occurrence time of the first instance of the task. Therefore, the phase of this task is 2000 milli seconds. This task can formally be represented as (2000 mSec, 50 mSec, 8 mSec, 50 mSec). This task is pictorially shown in Fig. 3. When the deadline of a task equals its period (i.e. p_i=d_i), we can omit the fourth tuple. In this case, we can represent the task as T_i= (2000 mSec, 50 mSec, 8 mSec). This would automatically mean p_i=d_i=50 mSec. Similarly, when $\phi_i = 0$, it can be omitted when no confusion arises. So, $T_i = (20mSec, 100mSec)$ would indicate a task with $\phi_i = 0$, p_i=100mSec, e_i=20mSec, and d_i=100mSec. Whenever there is any scope for confusion, we shall explicitly write out the parameters T_i=(p_i=50 milli Secs, e_i =8 milli Secs, d_i=40 milli Secs), etc.

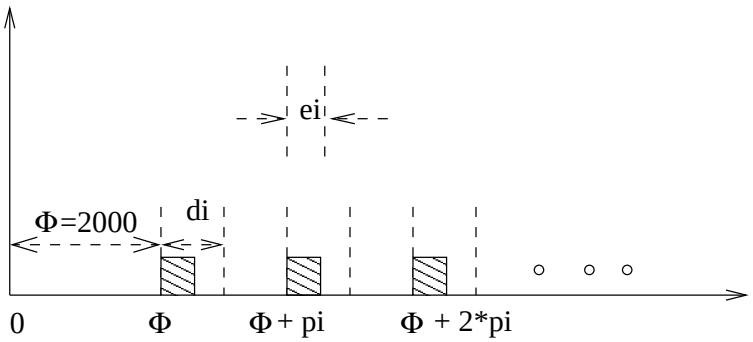


Figure 3: Track Correction Task $(2000mSec, p_i, e_i, d_i)$ of a Rocket

A vast majority of the tasks present in a typical real-time system are periodic. The reason for this is that many activities carried out by real-time systems are periodic in nature, for example monitoring certain conditions, polling information from sensors at regular intervals to carry out certain action at regular intervals (such as drive some actuators). We shall consider examples of such tasks found in a typical chemical plant. In a chemical plant several temperature monitors, pressure monitors, and chemical concentration monitors periodically sample the current temperature, pressure, and chemical concentration values which are then communicated to the plant controller. The instances of the temperature, pressure, and chemical concentration monitoring tasks are normally generated through the interrupts received from a periodic timer. These inputs are used to compute corrective actions required to maintain the chemical reaction at a certain rate. The corrective actions are then carried out through actuators.

The periodic task in the above example exists from the time of system initialization. However, periodic tasks can also come into existence dynamically. The computation that occurs in ait traffic monitors, once a flight is detected by the radar till the radar exits the radar signal zone is an example of a dynamically created periodic task.

Sporadic Task. A sporadic task is one that recurs at random instants. A sporadic task T_i can be is represented by a three tuple:

$$T_i = (e_i, g_i, d_i)$$

where e_i is the worst case execution time of an instance of the task, g_i denotes the minimum separation between two consecutive instances of the task, d_i is the relative deadline. The minimum separation (g_i) between two consecutive instances of the task implies that once an instance of a sporadic task occurs, the next instance cannot occur before g_i time units have elapsed. That is, g_i restricts the rate at which sporadic tasks can arise. As done for periodic tasks, we shall use the convention that the first instance of a sporadic task T_i is denoted by $T_i(1)$ and the successive instances by $T_i(2)$, $T_i(3)$, etc.

Many sporadic tasks such as emergency message arrivals are highly critical in nature. For example, in a robot a task that gets generated to handle an obstacle that suddenly appears is a sporadic task. In a factory, the task that handles fire conditions is a sporadic task. The time of occurrence of these tasks can not be predicted.

The criticality of sporadic tasks varies from highly critical to moderately critical. For example, an I/O device interrupt, or a DMA interrupt is moderately critical. However, a task handling the reporting of fire conditions is highly critical.

Aperiodic Task. An aperiodic task is in many ways similar to a sporadic task. An aperiodic task can arise at random instants. However, in case of an aperiodic task, the minimum separation g_i between two consecutive instances can be 0. That is, two or more instances of an aperiodic task might occur at the same time instant. Also, the deadline for an aperiodic tasks is expressed as either an average value or is expressed statistically. Aperiodic tasks

are generally soft real-time tasks.

It is easy to realize why aperiodic tasks need to be soft real-time tasks. Aperiodic tasks can recur in quick succession. It therefore becomes very difficult to meet the deadlines of all instances of an aperiodic task. When several aperiodic tasks recur in a quick succession, there is a bunching of the task instances and it might lead to a few deadline misses. As already discussed, soft real-time tasks can tolerate a few deadline misses. An example of an aperiodic task is a logging task in a distributed system. The logging task can be started by different tasks running on different nodes. The logging requests from different tasks may arrive at the logger almost at the same time, or the requests may be spaced out in time. Other examples of aperiodic tasks include operator requests, keyboard presses, mouse movements, etc. In fact, all interactive commands issued by users are handled by aperiodic tasks.

3 Task Scheduling

Real-time task scheduling essentially refers to determining the order in which the various tasks are to be taken up for execution by the operating system. Every operating system relies on one or more task schedulers to prepare the schedule of execution of various tasks it needs to run. Each task scheduler is characterized by the scheduling algorithm it employs. A large number of algorithms for scheduling real-time tasks have so far been developed. Real-time task scheduling on uniprocessors is a mature discipline now with most of the important results having been worked out in the early 1970's. The research results available at present in the literature are very extensive and it would indeed be gruelling to study them exhaustively. In this text, we therefore classify the available scheduling algorithms into a few broad classes and study the characteristics of a few important ones in each class.

3.1 A Few Basic Concepts and Terminologies

Before focussing on the different classes of schedulers more closely, let us first introduce a few important concepts and terminologies which would be used in our later discussions.

Valid Schedule. A valid schedule for a set of tasks is one where at most one task is assigned to a processor at a time, no task is scheduled before its arrival time, and the precedence and resource constraints of all tasks are satisfied.

Feasible Schedule. A valid schedule is called a feasible schedule, only if all tasks meet their respective time constraints in the schedule.

Proficient Scheduler. A task scheduler `sch1` is said to be *more proficient* than another scheduler `sch2`, if `sch1` can feasibly schedule all task sets that `sch2` can feasibly schedule, but not *vice versa*. That is, `sch1` can feasibly schedule all task sets that `sch2` can, but there exists at least one task set that `sch2` can not feasibly schedule, whereas `sch1` can. If `sch1` can feasibly schedule all task sets that `sch2` can feasibly schedule and *vice versa*, then `sch1` and `sch2` are called *equally proficient schedulers*.

Optimal Scheduler. A real-time task scheduler is called *optimal*, if it can feasibly schedule any task set that can be feasibly scheduled by any other scheduler. In other words, it would not be possible to find a more proficient scheduling algorithm than an optimal scheduler. If an optimal scheduler can not schedule some task set, then no other scheduler should be able to produce a feasible schedule for that task set.

Scheduling Points: The scheduling points of a scheduler are the points on time line at which the scheduler makes decisions regarding which task is to be run next. It is important to note that a task scheduler does not need to run continuously, it is activated by the operating system only at the scheduling points to make the scheduling decision as to which task to be run next. In a clock-driven scheduler, the scheduling points are defined at the time instants marked by interrupts generated by a periodic timer. The scheduling points in an event-driven scheduler are deter-

mined by occurrence of certain events. This topic is discussed more elaborately in Sec. 2.6.

Preemptive Scheduler: A preemptive scheduler is one which when a higher priority task arrives, suspends any lower priority task that may be executing and takes up the higher priority task for execution. Thus, in a preemptive scheduler, it can not be the case that a higher priority task is ready and waiting for execution, and the lower priority task is executing. A preempted lower priority task can resume its execution only when no higher priority task is ready.

Utilization: The processor utilization (or simply utilization) of a task is the average time for which it executes per unit time interval. In notations: for a periodic task T_i, the utilization $u_i = \frac{e_i}{p_i}$, where e_i is the execution time and p_i is the period of T_i. For a set of periodic tasks $\{T_i\}$: the total utilization due to all tasks U=$\sum_{i=1}^{n} \frac{e_i}{p_i}$. It is the objective of any good scheduling algorithm to feasibly schedule even those task sets that have very high utilization, i.e. utilization approaching 1. Of course, on a uniprocessor it is not possible to schedule task sets having utilization more than 1.

Jitter: Jitter is the deviation of a periodic task from its strict periodic behavior. The arrival time jitter is the deviation of the task from arriving at the precise periodic time of arrival. It may be caused by imprecise clocks, or other factors such as network congestions. Similarly, completion time jitter is the deviation of the completion of a task from precise periodic points. The completion time jitter may be caused by the specific scheduling algorithm employed which takes up a task for scheduling as per convenience and the load at an instant, rather than scheduling at some strict time instants. Jitters are undesirable for some applications. More discussions on this later.

3.2 Classification of Real-Time Task Scheduling Algorithms

Several schemes of classification of real-time task scheduling algorithms exist. A popular scheme classifies the real-time task scheduling algorithms based on how the scheduling points are defined. The three main types of schedulers according to this classification scheme are: clock-driven, event-driven, and hybrid.

> 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.

A few important members of each of these three broad classes of scheduling algorithms are the following:

1. Clock Driven:
   - Table-driven
   - Cyclic
2. Event Driven:
   - Simple priority-based
   - Rate Monotonic Analysis (RMA)
   - Earliest Deadline First (EDF)
3. Hybrid:
   - Round-robin

Important members of clock-driven schedulers that we discuss in this text are table-driven and cyclic schedulers. Clock-driven schedulers are simple and efficient. Therefore, these are frequently used in embedded applications. We investigate these two schedulers in some detail in Sec. 2.4.

Important examples of event-driven schedulers are Earliest Deadline First (EDF) and Rate Monotonic Analysis (RMA). Event-driven schedulers are more sophisticated than clock-driven schedulers and usually are more proficient and flexible than clock-driven schedulers. These are more proficient because they can feasibly schedule some task sets which clock-driven schedulers cannot. These are more flexible because they can feasibly schedule sporadic and aperiodic tasks in addition to periodic tasks, whereas clock-driven schedulers can satisfactorily handle only periodic tasks. Event-driven scheduling of real-time tasks in a uniprocessor environment was a subject of intense research during early 1970's, leading to publication of a large number of research results. Out of the large number of research results that were published, the following two popular algorithms are the essence of all those results: Earliest Deadline First (EDF), and Rate Monotonic Analysis (RMA). If we understand these two schedulers well, we would get a good grip on real-time task scheduling on uniprocessors. Several variations to these two basic algorithms exist.

Another classification of real-time task scheduling algorithms can be made based upon the type of task acceptance test that a scheduler carries out before it takes up a task for scheduling. The acceptance test is used to decide whether a newly arrived task would at all be taken up for scheduling or be rejected. Based on the task acceptance test used, there are two broad categories of task schedulers:

- Planning-based
- Best effort

In planning-based schedulers, when a task arrives the scheduler first determines whether the task can meet its deadlines, if it is taken up for execution. If not, it is rejected. If the task can meet its deadline and does not cause other already scheduled tasks to miss their respective deadlines, then the task is accepted for scheduling. Otherwise, it is rejected. In best effort schedulers, no acceptance test is applied. All tasks that arrive are taken up for scheduling and best effort is made to meet its deadlines. But, no guarantee is given as to whether a task's deadline would be met.

A third type of classification of real-time tasks is based on the target platform on which the tasks are to be run. The different classes of scheduling algorithms according to this scheme are:

- Uniprocessor
- Multiprocessor
- Distributed

Uniprocessor scheduling algorithms are possibly the simplest of the three classes of algorithms. In contrast to uniprocessor algorithms, in multiprocessor and distributed scheduling algorithms first a decision has to be made regarding which task needs to run on which processor and then these tasks are scheduled. In contrast to multiprocessors, the processors in a distributed system do not possess shared memory. Also in contrast to multiprocessors, there is no global uptodate state information available in distributed systems. This makes uniprocessor scheduling algorithms that assume a central state information of all tasks and processors to exist unsuitable for use in distributed systems. Further in distributed systems, the communication among tasks is through message passing. Communication through message passing is costly. This means that a scheduling algorithm should not incur too much communication overhead. So carefully designed distributed algorithms are normally considered suitable for use in a distributed system. We study multiprocessor and distributed scheduling algorithms in chapter 4.

In the following sections, we study the different classes of schedulers in more detail.

4 Clock-Driven Scheduling

Clock-driven schedulers make their scheduling decisions regarding which task to run next only at the clock interrupt points. Clock-driven schedulers are those for which the scheduling points are determined by timer interrupts. Clock-driven schedulers are also called off-line schedulers because these schedulers fix the schedule before the system starts

to run. That is, the scheduler pre-determines which task will run when. Therefore, these schedulers incur very little run time overhead. However, a prominent shortcoming of this class of schedulers is that they can not satisfactorily handle aperiodic and sporadic tasks since the exact time of occurrence of these tasks can not be predicted. For this reason, this type of schedulers are also called a static scheduler.

In this section, we study the basic features of two important clock-driven schedulers: table-driven and cyclic schedulers.

4.1 Table-Driven Scheduling

Table-driven schedulers usually precompute which task would run when and store this schedule in a table at the time the system is designed or configured. Rather than automatic computation of the schedule by the scheduler, the application programmer can be given the freedom to select his own schedule for the set of tasks in the application and store the schedule in a table (called *schedule table*) to be used by the scheduler at run time.

An example of a schedule table is shown in Table. 1. Table 1 shows that task T_1 would be taken up for execution at time instant 0, T_2 would start execution 3 milli seconds after wards, and so on. An important question that needs to be addressed at this point is what would be the size of the schedule table that would be required for some given set of periodic real-time tasks to be run on a system? An answer to this question can be given as follows: if a set ST=$\{T_i\}$ of n tasks is to be scheduled, then the entries in the table will replicate themselves after LCM$(p_1, p_2, ..., p_n)$ time units, where $p_1, p_2, ..., p_n$ are the periods of $T_1, T_2, ...$ For example, if we have the following three tasks: (e_1=5 msecs, p_1=20 msecs), (e_2=20 msecs, p_2=100 msecs), (e_3=30 msecs, p_3=250 msecs). Then, the schedule will repeat after every 500 msecs. So, for any given task set it is sufficient to store entries only for LCM$(p_1, p_2, ..., p_n)$ duration in the schedule table. LCM$(p_1, p_2, ..., p_n)$ is called the *major cycle* of the set of tasks ST.

A major cycle of a set of tasks is an interval of time on the time line such that in each major cycle, the different tasks recur identically.

In the reasoning we presented above for the computation of the size of a schedule table, one assumption that we implicitly made is that $\phi_i = 0$. That is, all tasks are in phase.

Task	Start Time in milli Seconds
T_1	0
T_2	3
T_3	10
T_4	12
T_5	17

Table. 1: An Example of a Table-Driven Schedule

However, tasks often do have non-zero phase. It would be interesting to determine what would be the major cycle when tasks have non-zero phase. The results of an investigation into this issue has been given as Theorem 2.1.

Theorem 2. 1. *The major cycle of a set of tasks ST=$\{T_1, T_2, ..., T_n\}$ is LCM($\{p_1, p_2, ..., p_n\}$) even when the tasks have arbitrary phasings.*

Proof: As per our definition of a major cycle, even when tasks have non-zero phasings, task instances would repeat the same way in each major cycle. Let us consider an example in which the occurrences of a task T_i in a major cycle be as shown in Fig. 4. As shown in the example of Fig. 4, there are k-1 occurrences of the task T_i during a major cycle. The first occurrence of T_i starts ϕ time units from the start of the major cycle. The major cycle ends x time units after the last (i.e. (k-1)th) occurrence of the task T_i in the major cycle. Of course, this must be the same in

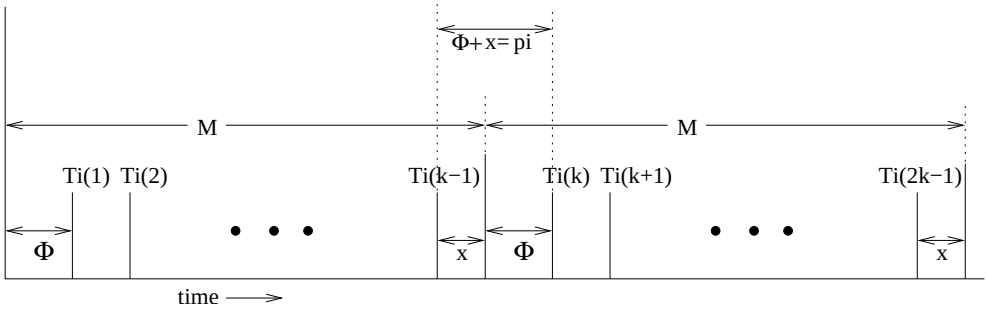


Figure 4: Major Cycle When A Task T_i Has Non-Zero Phasing

each major cycle.

Assume that the size of each major cycle is M. Then, from an inspection of Fig. 4, for the task to repeat identically in each major cycle.

$$M = (k-1)p_i + \phi + x \quad \ldots \quad (2.1)$$

Now, for the task T_i to have identical occurrence times in each major cycle, $\phi + x$ must equal to p_i (see Fig. 2.4). Substituting this in Expr. 2.1 we get, $M = (k-1) * p_i + p_i = k * p_i$

So, the major cycle M contains an integral multiple of p_i. This argument holds for each task in the task set irrespective of its phase. Therefore, M=LCM($\{p_1, p_2, ..., p_n\}$). □

4.2 Cyclic Schedulers

Cyclic schedulers are very popular and are being extensively used in the industry. A large majority of all small embedded applications being manufactured presently are based on cyclic schedulers. Cyclic schedulers are simple, efficient, and are easy to program. An example application where a cyclic scheduler is normally used is a temperature controller. A temperature controller periodically samples the temperature of a room and maintains it at a preset value. Such temperature controllers are embedded in typical computer-controlled air conditioners.

Task Number	Frame Number
T3	F1
T1	F2
T3	F3
T4	F2

Figure 5: An Example Schedule Table for a Cyclic Scheduler

A cyclic scheduler repeats a precomputed schedule. The precomputed schedule needs to be stored only for one *major cycle* as discussed in Sec. 2.4.1. Each task in the task set to be scheduled repeats identically in every major cycle. The major cycle is divided into one or more minor cycles (see Fig. 6). Each minor cycle is also called a *frame*. In the example shown in Fig. 6, the major cycle has been divided into four minor cycles (frames). The

scheduling points of a cyclic scheduler occur at frame boundaries. This means that a task can start executing only at the beginning of a frame.

The frame boundaries are defined through the interrupts generated by a periodic timer. Each task is assigned to run in one or more frames. The assignment of tasks to frames is stored in a *schedule table.* An example schedule table is shown in 5.

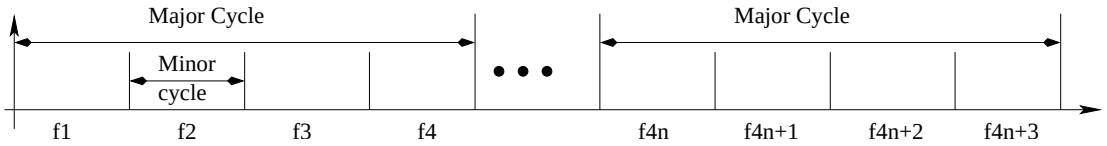


Figure 6: Major and Minor Cycles in a Cyclic Scheduler

The size of the frame to be used by the scheduler is an important design parameter and needs to be chosen very carefully. A selected frame size should satisfy the following three constraints.

1. **Minimum Context Switching.** This constraint is imposed to minimize the number of context switches occurring during task execution. The simplest interpretation of this constraint is that a task instance must complete running within its assigned frame. Unless a task completes within its allocated frame, the task might have to be suspended and restarted in a later frame. This would require a context switch involving some processing overhead. To avoid unnecessary context switches, the selected frame size should be larger than the execution time of each task, so that when a task starts at a frame boundary it should be able to complete within the same frame. Formally, we can state this constraint as: $max(\{e_i\}) \leq F$ where e_i is the execution times of the of task T_i, and F is the frame size. Note that this constraint imposes a lower-bound on frame size, i.e. the frame size F must not be smaller than $max(\{e_i\})$.

2. **Minimization of Table Size.** This constraint requires that the number of entries in the schedule table should be minimum in order to minimize the storage requirement of the schedule table. Remember that cyclic schedulers are used in small embedded applications with very small storage capacity. So, this constraint is important to the commercial success of a product. Minimization of the number of entries to be stored in the schedule table can be achieved when the minor cycle squarely divides the major cycle. When the minor cycle squarely divides the major cycle, the major cycle contains an integral number of minor cycles (no fractional minor cycles). Unless the minor cycle squarely divides the major cycle, storing the schedule for one major cycle would not be sufficient, as the schedules in the major cycle would not repeat and this would make the size of the schedule table large. We can formulate this constraint as:

$$\lfloor \frac{M}{F} \rfloor = \frac{M}{F}$$

   In other words, if the floor of M/F equals M/F, then the major cycle would contain an integral number of frames.

3. **Satisfaction of Task Deadline.** This third constraint on frame size is necessary to meet the task deadlines. This constraint imposes that between the arrival of a task and its deadline, there must exist at least one full frame. This constraint is necessary since a task should not miss its deadline, because by the time it could be taken up for scheduling, the deadline was imminent. Consider this: a task can only be taken up for scheduling at the start of a frame. If between the arrival and completion of a task not even one frame exists, a situation as shown in Fig. 7 might arise. In this case, the task arrives a little after the kth frame has started. Obviously it can not be taken up for scheduling in the kth frame and can only be taken up in the (k+1)th frame. But, then it may be too late to meet its deadline since the execution time of a task can be upto the size of a full frame. This might result in the task missing its deadline since the task might complete only at the end of (k+1)th

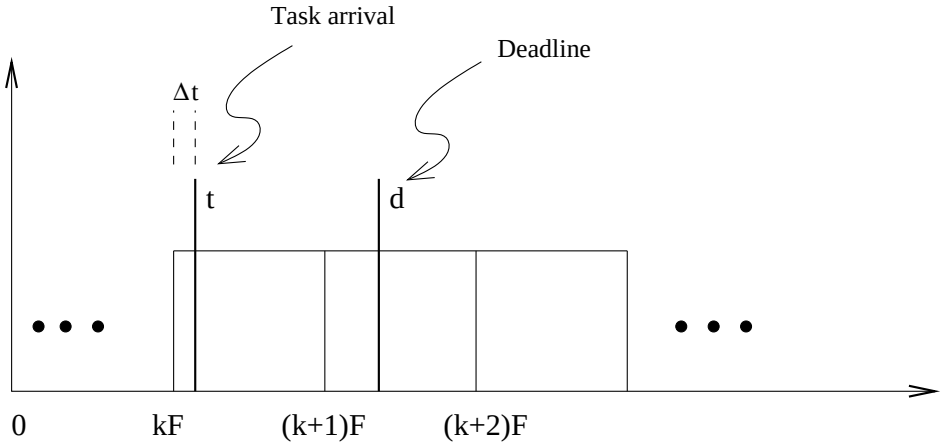


Figure 7: Satisfaction of a Task Deadline

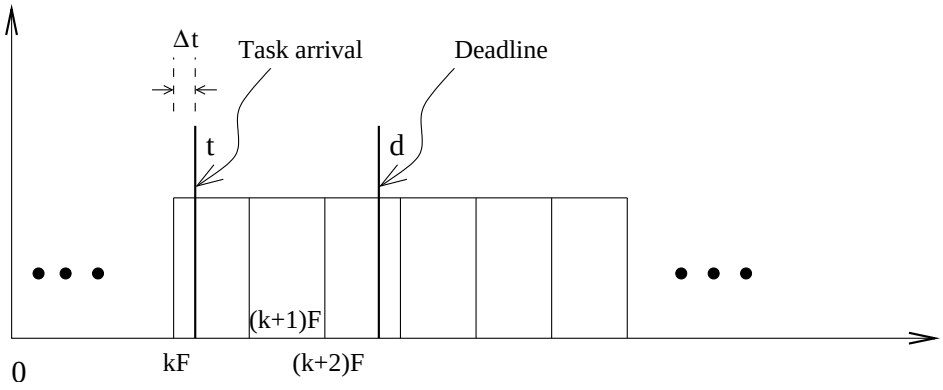


Figure 8: A Full Frame Exists Between the Arrival and Deadline of a Task

frame much after the deadline d has passed. We therefore need a full frame to exist between the arrival of a task and its deadline as shown in Fig. 8, so that task deadlines could be met.

More formally, this constraint can be formulated as follows: Suppose a task arises after Δt time units have passed since the last frame (see Fig. 8). Then, assuming that a single frame is sufficient to complete the task, the task can complete before its deadline iff $2F - \Delta t \leq d_i$ or, $2F \leq d_i + \Delta t$. Remember that the value of Δt might vary from one instance of the task to another. The worst case scenario (where the task is likely to miss its deadline) occurs for the task instance having the minimum value of Δt, such that $\Delta t > 0$. This is the worst case scenario, since under this the task would have to wait the longest before its execution can start.

It should be clear that if a task arrives just after a frame has started, then the task would have to wait for the full duration of the current frame before it can be taken up for execution. If a task at all misses its deadline, then certainly it would be under such situations. In other words, the worst case scenario for a task to meet its deadline occurs for its instance that has the minimum separation from the start of a frame. The determination of the minimum separation value (i.e. $\min(\Delta t)$) for a task among all instances of the task would help in determining a feasible frame size. We show by Theorem 2.2 that $min(\Delta t)$ is equal to $gcd(F, p_i)$ Consequently, this constraint can be written as: for every T_i:

$$2F - gcd(F, p_i) \leq d_i \quad \ldots (2.5)$$

Note that this constraint defines an upper-bound on frame size for a task T_i. That is, if the frame size is any larger than the defined upper-bound, then tasks might miss their deadlines. Expr. 2.5 defines the frame size, from the consideration of one task only. Now considering all tasks, the frame size must be smaller than max(gcd(F,p_i)+d_i)/2.

Theorem 2. 2. *The minimum separation of the task arrival from the corresponding frame start time (min(Δt)) considering all instances of a task T_i is equal to gcd(F, p_i).*

Proof: Let g=gcd(F,p_i), where gcd is the function determining the greatest common divisor of its arguments. It follows from the definition of gcd that g must squarely divide each of F and p_i. Let T_i be a task with zero phasing. Now, assume that this Theorem is violated for certain integers m and n, such that the $T_i(n)$ occurs in the mth frame and the difference between the start time of the mth frame and the arrival time of the nth task is less than g. That is, $0 < (m * F - n * p_i) < g$.
Dividing this expression throughout by g, we get:

$$0 < (m * F/g - n * p_i/g) < 1 \qquad ... \quad (2.6)$$

However, F/g and p_i/g are both integers because g is gcd(p_i,F). Therefore, we can write $F/g = I_1$ and $P_i/g = I_2$ for some integral values I_1 and I_2. Substituting this in Expr 2.6, we get $0 < m * I1 - n * I_2 < 1$. Since $m * I_1$ and $n * I_2$ are both integers, their difference cannot be a fractional value lying between 0 and 1. Therefore, this expression can never be satisfied.

It can therefore be concluded that the minimum time between a frame boundary and the arrival of the corresponding instance of T_i can not be less than gcd(F,p_i). □

For a given task set it is possible that more than one frame size satisfies all the three constraints. In such cases, it is better to choose the shortest frame size. This is because of the fact that the schedulability of a task set increases as more number of frames become available over a major cycle.

It should however be remembered that the mere fact that a suitable frame size can be determined does not mean that a feasible schedule would be found. It may so happen that there is not enough number of frames available in a major cycle to be assigned to all the task instances.

We now illustrate how an appropriate frame size can be selected for cyclic schedulers through a few examples.

Example 1. A cyclic scheduler is to be used to run the following set of periodic tasks on a uniprocessor: T_1 = (e_1=1,p_1=4), T_2 = (e_2=1,p_2=5), T_3 = (e_3=1,p_3=20), T_4 = (e_4=2,p_4=20). Select an appropriate frame size.

Solution. For the given task set, an appropriate frame size is the one that satisfies all the three required constraints. In the following, we determine a suitable frame size F which satisfies all the three required constraints.

Constraint 1. Let F be an appropriate frame size, then $max\{e_i\} \leq F$. From this constraint, we get $F \geq 1.5$

Constraint 2. The major cycle M for the given task set is given by M = LCM(4,5,20) = 20.
M should be an integral multiple of the frame size F, i.e. $M \ mod \ F = 0$.
This consideration implies that F can take on the values 2, 4, 5, 10, 20. Frame size of 1 has been ruled out since it would violate the constraint 1.

Constraint 3. To satisfy this constraint, we need to check whether a selected frame size F satisfies the inequality: $2F - gcd(F, p_i) \leq d_i$ for each p_i.

Let us first try frame size 2.
For F = 2 and task T_1:

$$2 * 2 - gcd(2, 4) \leq 4 \equiv 4 - 2 \leq 4$$

Therefore, for p_1 the inequality is satisfied.
Let us try for F = 2 and task T_2:

$$2 * 2 - gcd(2, 5) \leq 4 \equiv 4 - 1 \leq 5$$

Therefore, for p_2 the inequality is satisfied.
Let us try for F = 2 and task T_3:

$$2 * 2 - gcd(2, 20) \leq 4 \equiv 4 - 2 \leq 20$$

For p_3 the inequality is satisfied.
Therefore, for F = 2 and task T_4:

$$2 * 2 - gcd(2, 20) \leq 4 \equiv 4 - 2 \leq 20$$

For p_4 the inequality is satisfied.
Thus, constraint 3 is satisfied by all tasks for frame size 2.

So, frame size 2 satisfies all the three constraints. Hence, 2 is a feasible frame size.

Let us try frame size 4.
For F = 4 and task T_1:

$$2 * 4 - gcd(4, 4) \leq 4 \equiv 8 - 4 \leq 4$$

Therefore, for p_1 the inequality is satisfied.
Let us try for F = 4 and task T_2:

$$2 * 4 - gcd(4, 5) \leq 5 \equiv 8 - 1 \leq 5$$

For p_2 the inequality is not satisfied. We need not therefore look any further. Clearly, F = 4 is not a suitable frame size.

Let us now try frame size 5, to check if that is also feasible.
For F = 5 and task T_1 , we have

$$2 * 5 - gcd(5, 4) \leq 4 \equiv 10 - 1 \leq 4$$

The inequality is not satisfied for T_1. We need not look any further. Clearly, F = 5 is not a suitable frame size

Let us now try frame size 10.
For F = 10 and task T_1, we have

$$2 * 10 - gcd(10, 4) \leq 4 \equiv 20 - 2 \leq 4$$

The inequality is not satisfied for T_1. We need not look any further. Clearly, F = 10 is not a suitable frame size

Let us try if 20 is a feasible frame size.
For F = 20 and task T_1, we have

$$2 * 20 - gcd(20, 4) \leq 4 \equiv 40 - 4 \leq 4$$

Therefore, F = 20 is also not suitable.

So, only the frame size 2 is suitable for scheduling. □

Even though for Example 2.1 we could successfully find a suitable frame size that satisfies all the three constraints, it is quite probable that a suitable frame size may not exist for many problems. In such cases, to find a feasible frame size we might have to split the task (or a few tasks) that is (are) causing violation of the constraints into smaller sub-tasks that can be scheduled in different frames.

Example 2.2: Consider the following set of periodic real-time tasks to be scheduled by a cyclic scheduler: T_1 = (e_1=1,p_1=4), T_2 = (e_2=2,p_2=5), T_3 = (e_3=5,p_3=20). Determine a suitable frame size for the task set.

Solution Using the first constraint, we have $F \geq 5$.
Using the second constraint, we have the major cycle $M = LCM(4, 5, 20) = 20$. So, the permissible values of F are 5, 10 and 20. Checking for a frame size that satisfies the third constraint, we can find that no value of F is suitable. To overcome this problem, we need to split the task that is making the task set unschedulable. It is easy to observe that the task T_3 has the largest execution time and consequently due to the constraint 1 makes the feasible frame sizes quite large.

We try splitting T_3 into two or three tasks. After splitting T_3 into three tasks, we have: $T_{3.1}$ = (20,1,20), $T_{3.2}$ = (20,2,20), $T_{3.3}$ = (20,2,20).

Now the possible values of F are 2 and 4. We can check that after splitting the tasks, F=2 and F=4 become feasible frame sizes. □

It is very difficult to come up with a clear set of guidelines to identify the exact task that is to be split, and the parts into which it needs to be split. This therefore needs to be done by trial and error. Further, as the number of tasks to be scheduled increases, this method of trial and error becomes impractical since each task needs to be checked separately. However, when the task set consists of only a few tasks we can easily apply this technique to find a feasible frame size for a set of tasks otherwise unschedulable by a cyclic scheduler.

4.3 A Generalized Task Scheduler

We have already stated that cyclic schedulers are overwhelmingly popular in low-cost real-time applications. However, our discussion on cyclic schedulers was so far restricted to scheduling periodic real-time tasks. On the other hand, many practical applications typically consist of a mixture of several periodic, aperiodic, and sporadic tasks. In this section, we discuss how aperiodic and sporadic tasks can be accommodated by cyclic schedulers.

Recall that the arrival times of aperiodic and sporadic tasks are expressed statistically. Therefore, there is no way to assign aperiodic and sporadic tasks to frames without significantly lowering the overall achievable utilization of the system. In a generalized scheduler, initially a schedule (assignment of tasks to frames) for only periodic tasks is prepared. The sporadic and aperiodic tasks are scheduled in the slack times that may be available in the frames. Slack time in a frame is the time left in the frame after a periodic task allocated to the frame completes its execution. Non-zero slack time in a frame can exist only when the execution time of the task allocated to it is smaller than the frame size.

A **sporadic** task is taken up for scheduling only if enough slack time is available for the arriving sporadic task to complete before its deadline. Therefore, a sporadic task on its arrival is subjected to an acceptance test. The acceptance test checks whether the task is likely to be completed within its deadline when executed in the available slack times. If it is not possible to meet the task's deadline, then the scheduler rejects it and the corresponding recovery routines for the task are run. Since aperiodic tasks do not have strict deadlines, they can be taken up for scheduling without any acceptance test and best effort can be made to schedule them in the slack times available. Though for aperiodic tasks no acceptance test is done, but no guarantee is given for a task's completion time and best effort is made to complete the task as early as possible.

An efficient implementation of this scheme is that the slack times are stored in a table and during acceptance test this table is used to check the schedulability of the arriving tasks.

Another popular alternative is that the aperiodic and sporadic tasks are accepted without any acceptance test, and best effort is made to meet their respective deadlines.

Pseudo-code for a Generalized Scheduler. The following is the pseudo-code for a generalized cyclic scheduler we discussed which schedules periodic, aperiodic, and sporadic tasks. It is assumed that the precomputed schedule for periodic tasks is stored in a schedule table, and if required the sporadic tasks have already been subjected to an acceptance test and only those which have passed the test are available for scheduling.

```
cyclic-scheduler() {
current-task T = Schedule-Table[k];
k = k + 1;
k = k mod N;                // N is the total number of tasks
                            //in the schedule table
dispatch-current-Task(T);
schedule-sporadic-tasks();  //Current task T completed early,
                            //sporadic tasks can be taken up.
schedule-aperiodic-tasks(); //At the end of the frame, the running
                            //task is preempted, if not complete.
idle(),                     // No task to run, idle.
}
```

The cyclic scheduler routine `cyclic-scheduler()` is activated at the end of every frame by a periodic timer. If the current task is not complete by the end of the frame, then it is suspended and the task to be run in the next frame is dispatched by invoking the routine `cyclic-scheduler()`. If the task scheduled in a frame completes early, then any existing sporadic or aperiodic task is taken up for execution.

4.4 Comparison of Cyclic with Table-Driven Scheduling

Both table-driven and cyclic schedulers are important clock-driven schedulers. A cyclic scheduler needs to set a periodic timer only once at the application initialization time. This timer continues to give an interrupt exactly at every frame boundary. But in table-driven scheduling, a timer has to be set every time a task starts to run. The execution time of a typical real-time task is usually of the order of a few milli Seconds. Therefore, a call to a timer is made every few mill Seconds. This represents a significant overhead and results in degraded system performance. Therefore, a cyclic scheduler is more efficient than a table-driven scheduler. This probably is a reason why cyclic schedulers are so overwhelmingly popular especially in embedded applications. However, if the overhead of setting a timer can be ignored, a table-driven scheduler is more proficient than a cyclic scheduler because the size of the frame that needs to be chosen should be at least as long as the size of the largest execution time of a task in the task set. This is a source of inefficiency, since this results in processor time being wasted in case of those tasks whose execution times are smaller than the chosen frame size.

5 Hybrid Schedulers

We had seen that for clock-driven schedulers, the scheduling points are defined through clock interrupts and in case of event-driven schedulers these are defined by events such as arrival and completion of tasks. In hybrid schedulers, the scheduling points are defined both through the clock interrupts and the event occurrences. In the following, we discuss time-sliced round-robin scheduling — a popular hybrid scheduler.

Time-Sliced Round Robin Scheduling:
Time-sliced round robin schedulers are very commonly used in the traditional operating systems, and are profusely discussed in the standard operating systems books and in the available literature. We therefore keep our discussion on time-sliced round robin scheduling to the minimum. Time-sliced round robin scheduling is a preemptive scheduling method. In round robin scheduling, the ready tasks are held in a circular queue. The tasks are taken up one after

the other in a sequence from the queue. Once a task is taken up, it runs for a certain fixed interval of time called its *time slice*. If a task does not complete within its allocated time slice, it is inserted back into the circular queue. A time-sliced round-robin scheduler is less proficient than table-driven or cyclic scheduler for scheduling real-time tasks. It is rather easy to see why this is so. A time-sliced round robin scheduler treats all tasks equally, and all tasks are assigned identical time slices irrespective of their priority, criticality, or closeness of deadline. So, tasks with short deadlines might fail to complete on time.

However, it is possible to consider task priorities in the time-sliced round-robin schedulers through a minor extension to the basic round robin scheme. The scheduler can assign larger time slices to higher priority tasks. In fact, the number of slices allocated to a task can be made proportional to the priority of the task. Even with this modification, time-sliced round-robin scheduling is far from satisfactory for real-time task scheduling (Can you identify the reasons?). In this case, the higher priority tasks are made to complete as early as possible. However, proficient real-time schedulers should try to meet the deadlines of as many tasks as possible, rather than completing the higher priority tasks in the shortest time.

6 Event-driven Scheduling

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 are normally preemptive. That is, a higher priority task when becomes ready, preempts any lower priority task that may be running. We discuss three important examples of event-driven schedulers. The simplest of these is the foreground-background scheduler, which we discuss next. In section 2.7 we discuss EDF and in section 2.8 we discuss RMA.

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 foreground 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: T_1,T_2,...,T_n. 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:

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

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 $\sum_{i=1}^{n} \frac{e_i}{p_i}$. Therefore,

the average time available for execution of the background tasks in every unit of time is $1 - \sum_{i=1}^{n} \frac{e_i}{p_i}$. From this, Expr. 2.7 follows easily. We now illustrate the applicability of Expr. 2.7 through the following three simple examples.

Example 2.3: 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 $T_B = (e_B = 1000msec)$. Compute the completion time for background task.

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

$$ct_B = \frac{1000}{1 - \frac{50}{100}} msec = 2000msec$$

So, the background task T_B would take 2000 milli Seconds to complete. □

Example 2.4: In a simple priority-driven preemptive scheduler, two periodic tasks T_1 and T_2 and a background task are scheduled. The periodic task T_1 has the highest priority and executes once every 20 milli seconds and requires 10 milli seconds of execution time each time. T_2 requires 20 milli seconds of processing every 50 milli seconds. T_3 is a background task and requires 100milli seconds to complete. Assuming that all the tasks start at time 0, determine the time at which T_3 will complete.

Solution: The total utilization due to the foreground tasks: $\sum_{i=1}^{2} \frac{e_i}{p_i} = \frac{10}{20} + \frac{20}{50} = \frac{90}{100}$.
This implies that the fraction of time remaining for the background task to execute is given by $1 - \sum_{i=1}^{2} \frac{e_i}{p_i} = \frac{10}{100}$
Therefore, the background task gets 1 milli second every 10 milli seconds. Thus, the background task would take 100*(10/1)=1000 milli seconds to complete. □

Example 2.5: Suppose in Example 2.3, an overhead of 1 msec on account of every context switch is to be taken into account. Compute the completion time of T_B.

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. 9. 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 milli Sec from 50 milli Secs. This has pictorially been shown in Fig. 9. □

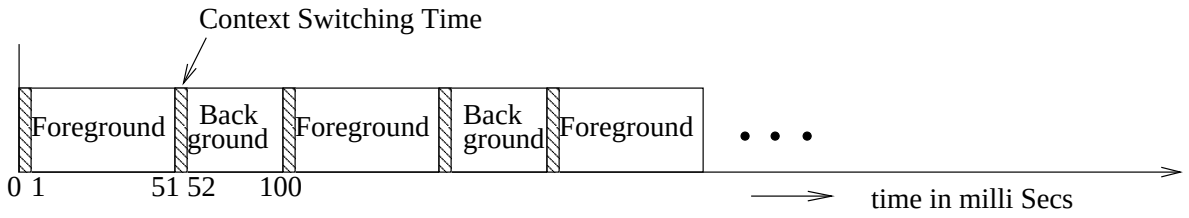


Figure 9: Task Schedule for Example 2.3

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

$$\frac{1000}{1 - \frac{52}{100}} = 2083.4 \; milli \; seconds$$

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.

7 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. The basic principle 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 $\{T_1, T_2, ..., T_n\}$, EDF schedulability criterion can be expressed as:

$$\sum_{i=1}^{n} \frac{e_i}{p_i} = \sum_{i=1}^{n} u_i \leq 1 \qquad .... \quad (2.8)$$

where u_i is average utilization due to the task Ti and n is the total number of tasks in the task set. Expr 2.8 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 [2]. This means that if a set of tasks is unschedulable under EDF, then no other scheduling algorithm can feasibly schedule this task set. In the simple schedulability test for EDF (Expr. 2.8), 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. 2.8 as:

$$\sum_{i=1}^{n} \frac{e_i}{min(p_i, d_i)} \leq 1 \qquad ... \quad (2.9)$$

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 2.9. Therefore, Expr 2.9 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 2.6: 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: $\sum_{i=1}^{3} \frac{e_i}{p_i} = \frac{10}{20} + \frac{5}{50} + \frac{10}{35} = 0.89$
This is less than 1. Therefore, the task set is EDF schedulable. □

A variant of EDF scheduling is **Minimum Laxity First (MLF)** Scheduling. In MLF, at every scheduling point, a laxity value is computed for every task in the system, and the task having the minimum laxity is executed first. Laxity of a task measures the amount of time that would remain if the task is taken up for execution next. Essentially, laxity is ameasure of the flexibility available for scheduling a task. The main difference between MLF and EDF is that unlike EDF, MLF takes into consideration the execution time of a task.

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. 2.7.3. Next, we discuss the concept of task priority in EDF and then discuss how EDF can be practically implemented.

7.1 Is EDF Really a Dynamic Priority Scheduling Algorithm?

We stated in Sec 2.6 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 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.

7.2 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).

7.3 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 chapter.

Efficient Implementation Problem: The efficient implementation that we discussed in Sec. 2.7.2 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.

8 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. The interested reader may see [2] for a proof.

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: $priority = \frac{k}{p_i}$, 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. 10(a) and Fig. 10(b). It can be observed from Fig. 10 (a) and (b) that the priority of a task increases linearly with the arrival rate of the task and inversely with its period.

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.

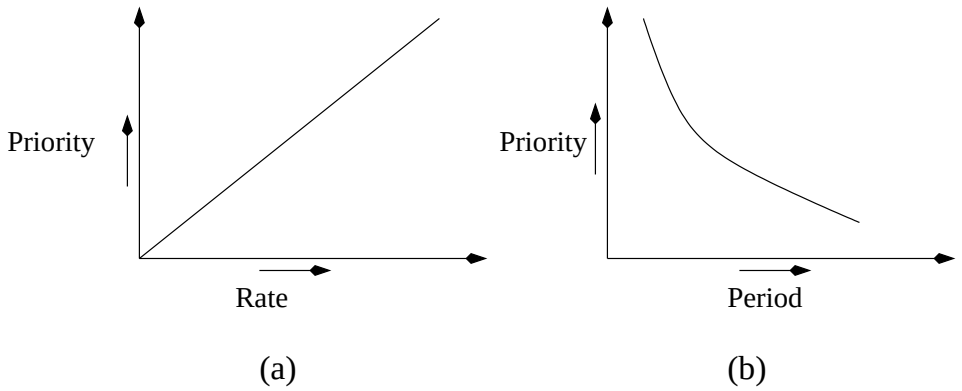


Figure 10: Priority Assignment to Tasks in RMA

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

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

where e_i is the worst case execution time and p_i is the period of the task T_i, n is the number of tasks to be scheduled, and u_i 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.

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 [1]. A formal derivation of the Liu and Layland's results from first principles is beyond the scope of this book. The interested reader is referred to [2] for a formal treatment of this important result. 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

$$\sum_{i=1}^{n} u_i \leq n(2^{\frac{1}{n}} - 1) \qquad \ldots \quad (2.10)$$

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 (2.10), we get

$$\sum_{i=1}^{1} u_i \leq 1(2^{\frac{1}{1}} - 1), \quad or \quad \sum_{i=1}^{1} u_i \leq 1$$

Similarly, for n=2 we get

$$\sum_{i=1}^{2} u_i \leq 2(2^{\frac{1}{2}} - 1), \quad or \quad \sum_{i=1}^{2} u_i \leq 0.824$$

For n=3 we get

$$\sum_{i=1}^{3} u_i \leq 3(2^{\frac{1}{3}} - 1), \quad or \quad \sum_{i=1}^{3} u_i \leq 0.78$$

When n$\rightarrow \infty$ we get

$$\sum_{i=1}^{\infty} u_i \leq \infty(2^{\frac{1}{\infty}} - 1), or \quad \sum_{i=1}^{\infty} u_i \leq = \infty.0$$

Evaluation of Expr. 2.10 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 11. We now illustrate the applicability of the RMA schedulability criteria through a few examples.

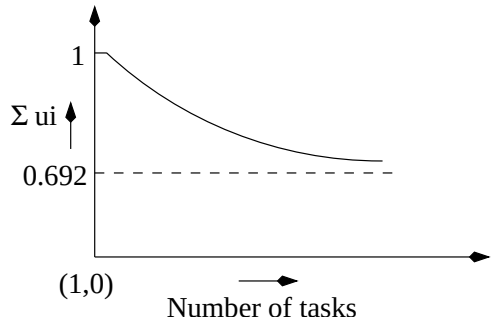


Figure 11: Achievable Utilization with the Number of Tasks under RMA

Example 2.7: 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.

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

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. 2.10 is satisfied. Checking for satisfaction of Expr. 2.10, the maximum achievable utilization is given by: $3(2^{\frac{1}{3}} - 1) = 0.78$. The total utilization has already been found to be 0.7. Now substituting these in the Liu and Layland's criterion: $\sum_{i=1}^{3} u_i \leq 3(2^{\frac{1}{3}} - 1)$, we get $0.7 < 0.78$.
Expr. 2.10 which is a sufficient condition for RMA schedulability is satisfied. Therefore, the task set is RMA schedulable. □

Example 2.8: 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:

$$\sum_{i=1}^{3} u_i = \frac{20}{100} + \frac{30}{150} + \frac{90}{200} = 0.85$$

Now checking for Liu and Layland criterion: $\sum_{i=1}^{3} u_i \leq 0.78$; since $0.85 \not\leq 0.78$, the task set is not RMA schedulable.

□

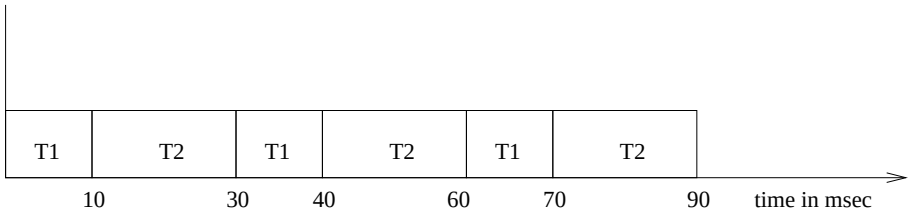
Liu and Layland test (Expr. 2.10) is pessimistic in the following sense.

> 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.

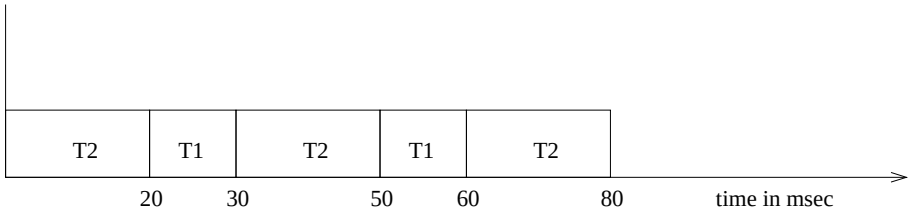
It follows from this that even when a task set fails Liu and Layland's test, we should not conclude that it is not schedulable under RMA. We need to test further to check if the task set is RMA schedulable. A test that can be performed to check whether a task set is RMA schedulable when it fails the Liu and Layland test is the Lehoczky's test [3].

Lehoczky test has been expressed as Theorem 2.3.

Theorem 2. 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) T1 is in phase with T2



(b) T1 has a 20 msec phase with respect to T2

Figure 12: 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 book. However, we provide an intuitive reasoning as to why Theorem 2.3 must be true. For a formal proof of the Lehoczky' results the reader is referred to [3]. Intuitively, we can understand this result from the following reasonings. 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 priority tasks.

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. 12. In Fig. 12(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. 12(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.

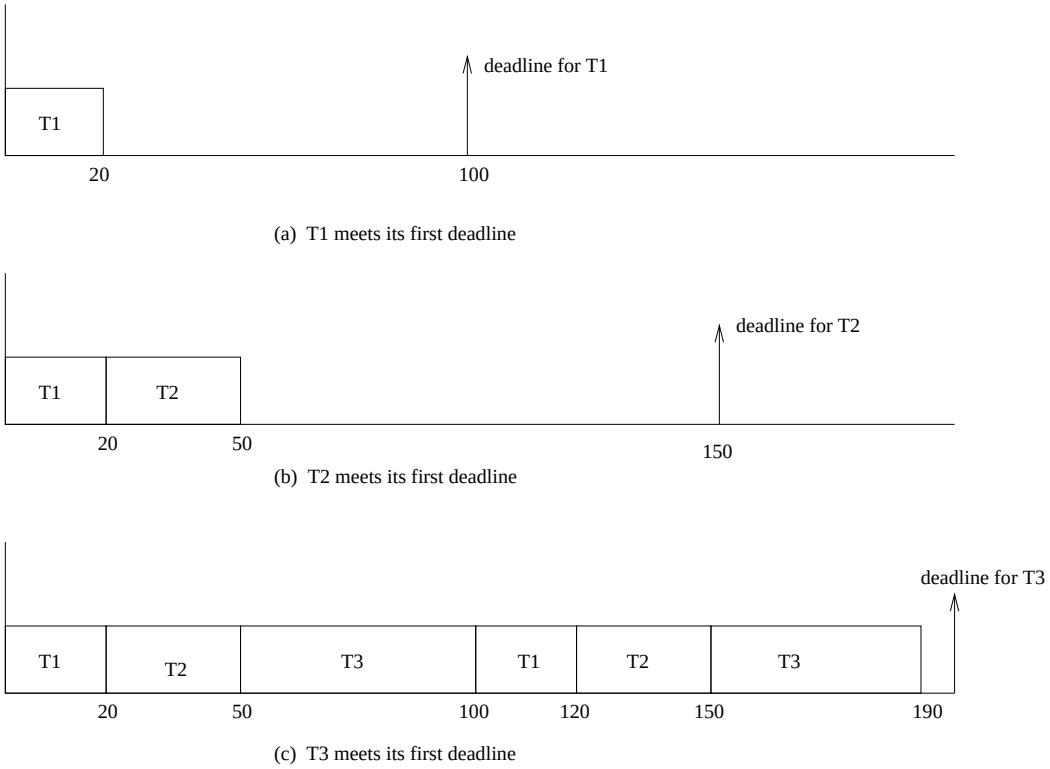


Figure 13: Checking Lehoczky's Criterion for Tasks of Example 2.9

Example 2.9: Check whether the task set of Example 2.8 is actually schedulable under RMA.

Solution: Though the results of Liu and Layland's test were negative, as per the results of Example 2.8 we can apply the Lehoczky test and observe the following:
For the task T_1: $e_1 < p_1$ holds since $20msec < 100msec$. Therefore, it would meet its first deadline (it does not have any tasks that have higher priority).
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 $20 + 30 = 50msec < 150msec$. Therefore, T_2 meets its first deadline.

For the task T_3: $(2e_1 + 2e_2 + e_3) < p_3$ holds, since $2 * 20 + 2 * 30 + 90 = 190msec < 200msec$. We have considered $2*e_1$ and $2*e_2$ 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. 13. Since all the tasks meet their first deadlines under zero phasing, they are RMA schedulable according to Lehoczky's results.

□

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: $pri(T_1) > pri(T_2) > \ldots > pri(T_i)$, where pri($T_i$) denotes the priority of the

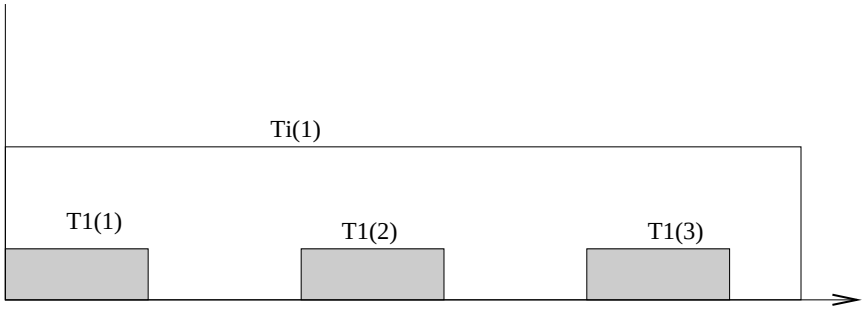


Figure 14: Instances of T_1 Over a Single Instance of T_i

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. 14. 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 $\lceil \frac{p_i}{p_1} \rceil$. 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 $\lceil \frac{p_i}{p_1} \rceil \times e_1$. This expression can easily be generalized to consider the execution times all tasks having higher priority than T_i (i.e. $T_1, T_2, \ldots, T_{i-1}$). Therefore, the time for which T_i will have to wait due to all its higher priority tasks can be expressed as:

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

Expression 2.11 gives the total time required to execute T_i's higher priority tasks for which T_i would have to wait. So, the task T_i would meet its first deadline, iff

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

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. 2.12 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. 2.12 would need modifications as follows.

$$e_i + \sum_{k=1}^{i-1} \lceil \frac{d_i}{p_k} \rceil \times e_k \leq d_i \qquad \ldots \qquad (2.13)$$

Note that even if Expr. 2.13 is not satisfied, there is some possibility that the task set may still be schedulable. This might happen because in Expr. 2.13 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 2.13, 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 T_i fails Expr. 2.13, making the task set unschedulable. 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 2.13.

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

Example 2.10: 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. 2.10 (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. 2.10 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 e_1=10 msec is less than d_1=20 msec. Therefore T_1 would meet its first deadline.
Testing for task T_2: $10 + \lceil \frac{60}{20} \rceil \times 10 \leq 60$ or, $20 + 30 = 50 \leq 60$
This is satisfied. Therefore, T_2 would meet its first deadline.
Testing for Task T_3: $20 + \lceil \frac{120}{20} \rceil \times 10 + \lceil \frac{120}{60} \rceil \times 15 \rceil = 20 + 60 + 30 = 110 msec$
This is less than T_3'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 2.11: 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 assertion. 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: $T_1 = (e_1 = 15msec, p_1 = 20msec)$, $T_2 = (e_2 = 6msec, p_2 = 35msec)$, $T_3 = (e_3 = 3msec, p_3 = 100msec)$. For the given task set, it is easy to observe that $pri(T_1) > pri(T_2) > pri(T_3)$. That is, T_1, T_2, T_3 are ordered in decreasing order of their priorities.

For this task set, T_3 meets its deadline according to Lehoczky's test since $e_3 + (\lceil \frac{p3}{p_2} \rceil \times e_2) + (\lceil \frac{p3}{p_1} \rceil \times e_1) = 3 + 3 * 6 + 5 * 15 = 96 \leq 100$. But, T_2 does not meet its deadline since, $e_2 + \lceil \frac{p2}{p_1} \rceil \times e_1 = 6 + 2 * 15 = 36$, which is greater than the deadline of T_2.

□

As a consequence of the results of Example 11, 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 from this that the entire task set would be feasibly scheduled under RMA is likely to be flawed.

8.1 Can The Achievable CPU Utilization Be Any Better Than What Was Predicted?

Liu and Layland's results (Expr. 2.10) bounded the CPU utilization below which a task set would be schedulable. It is clear from Expr. 2.10 and Fig. 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 $n \times p_k$ 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 msec,30 msec), T_2=(8 msec,120 msec), T_3=(12 msec,60 msec).

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

Theorem 2. 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 $T_1, T_2, \ldots, T_n$ be the tasks in the given task set. Let us further assume that the tasks in the task set $T_1, T_2, \ldots, T_n$ have been arranged in increasing order of their periods. That is, for any i and j, $p_i < p_j$ whenever $i < j$. If this relationship is not satisfied, then a simple renaming of the tasks can achieve this. Now according to Expr. 2.14, a task T_i meets its deadline, if $e_i + \sum_{k=1}^{i-1} \lceil \frac{p_i}{p_k} \rceil * e_k \leq p_i \quad \ldots (2.14)$
However, since the task set is harmonically related, p_i can be written as $m * p_k$ for some m. Using this, $\lceil \frac{p_i}{p_k} \rceil = \frac{p_i}{p_k}$. Now, Expr. 2.12 can be written as: $e_i + \sum_{k=1}^{i-1} \frac{p_i}{p_k} * e_k \leq p_i$. For $T_i = T_n$, we can write, $e_n + \sum_{k=1}^{n} \frac{p_n}{p_k} * e_k \leq p_n$. Dividing both sides of this expression by p_n we get the required result, the task set would be schedulable iff $\sum_{k=1}^{n} \frac{e_k}{p_k} \leq 1$, or $\sum_{i=1}^{n} u_i \leq 1$.

□

9 Some Issues Associated With RMA

In this section, we address some miscellaneous issues associated with RMA scheduling of tasks. We first discuss the advantages and disadvantages of using RMA for scheduling real-time tasks. We then discuss why RMA no longer is optimal when task deadlines differ from the corresponding task periods.

9.1 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. 15). 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. We also discuss in the next chapter why this choice of scheduling among equal priority tasks has an important bearing on the resource sharing protocols.

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, 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

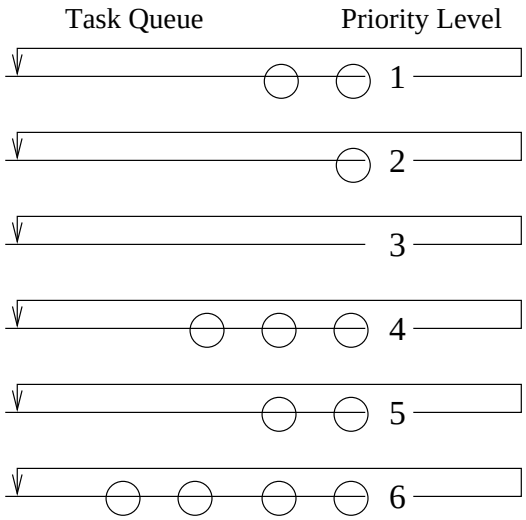


Figure 15: Multi-Level Feedback Queue

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.

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.

9.2 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 2.12: Is the following task set schedulable by DMA? Also check whether it is schedulable using RMA. T_1 = (e_1=10 msec, p_1=50 msec, d_1=35 msec), T_2 = (e_2=15 msec, p_2=100 msec, d_2=20 msec), T_3 = (e_3=20 msec, p_3=200 msec, d_3=200 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 $T_1 : 10msec < 35msec$. Therefore, T_1 would meet its first deadline.
Checking for $T_2 : 10 + 15 \not\leq 20$. Therefore, 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 $T_2 : 10msec < 35msec$. Hence T_2 will meet its first deadline.
Checking for $T_1 : (15 + 20)msec \leq 20msec$, Hence T_1 will meet its first deadline.
Checking for $T_3 : (70 + 30 + 20)msec < 200msec$. Therefore, T_3 will meet its deadline.

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

9.3 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 milli Seconds 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 $e_i + 2c$ for each T_i.

Example 13: Check whether the following set of periodic real-time tasks is schedulable under RMA on a uniprocessor: T_1=(e_1=20 msec,p_1=100 msec), T_2=(e_2=30 msec,p_2=150 msec), T_3=(e_3=90 msec,p_3=200 msec). Assume that context switching overhead does not exceed 1 milli secs 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:

$$\sum_{i=1}^{3} u_i = \frac{22}{100} + \frac{32}{150} + \frac{92}{200} = 0.893$$

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 $T_1 : 22 < 100$. This is satisfied, therefore T_1 meets its first deadline.
Checking for task $T_2 : 22 * 2 + 32 < 150$. This is satisfied, therefore T_2 meets its first deadline.
Checking for task $T_3 : 22 * 2 + 32 * 2 + 90 < 200$. This 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.

□

9.4 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 Sec. 2.3.1. We therefore need to augment our definition of a scheduling point given in Sec. 2.3.1 accordingly.

> 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 $\{T_1, T_2, ..., T_n\}$, 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:

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

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 nonpreemptable, the effect of self suspension is much more severe than that computed by Expr. 2.15. The reason for this 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 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. 2.10) needs to change when we consider the effect of self suspension of tasks. To consider the effect of self suspensions in Expr. 2.10, 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. 2.12) would also have to be generalized:

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

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 the expression 2.16 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 msec, p_1=50 msec), T_2 = (e_2=25 msec, p_2=150 msec). T_3 = (e_3=50 msec, p_3=200 msec). Assume that the self suspension times of T_1,T_2, and T_3 are 3 milli Seconds, 3 milli Seconds, and 5 milli Seconds, 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. 2.16 we get:
For T_1 to be schedulable: $(10 + 3)msec < 50msec$. Therefore T_1 would meet its first deadline.
For T_2 to be schedulable: $(25 + 6 + 10 * 3)msec < 150msec$. Therefore, T_2 meets its first deadline.
For T_3 to be schedulable: $(50 + 11 + (10 * 4 + 25 * 2))msec < 200msec$. 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. □

9.5 Self Suspension with Context Switching Overhead

Let us examine the effect of context switches on the generalized Lehoczky's test (Expr. 2.16) for schedulability of a task set, that 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. 2.16. We can easily extend this argument to consider two, three, or more self suspensions.

10 Issues in Using RMA in Practical Situations

While applying RMA to practical problems, a few interesting issues come up. The first issue that we discuss arises due to the fact that RMA does not consider task criticalities. The other issues that we discuss are coping up with limited number of priority levels that commercial operating systems supports, and handling of aperiodic and sporadic tasks.

10.1 Handling Critical Tasks with Long Periods

A situation that real-time system designers often have to deal with while developing practical applications, is handling applications for which the task criticalities are different from task priorities. Consider the situation where a very critical task has a higher period than some task having a lower criticality. In this case, the highly critical task would be assigned lower priority than a low criticality task. As a result, the critical task might at times miss its deadline due to a simple transient overload of a low critical (but high priority) task. If we simply raised the priority of a critical task to high levels, then the RMA schedulability results would not hold and determining the schedulability of a set of tasks would become extremely difficult.

A solution to this problem was proposed by Sha and Raj Kumar [4]. Sha and Raj Kumar's solution is known as the **period transformation technique.** In this technique, a critical task is logically divided into many small subtasks. Let T_i be a critical task that is split into k subtasks. Let each of these subtasks have period $\frac{p_i}{k}$. Similarly,

the deadline and worst case execution times of these subtasks will be $\frac{d_i}{k}$ and $\frac{e_i}{k}$ respectively. The net effect when a task T_i is split into k subtasks: $\{Ti_1, ..., Ti_k\}$ is to effectively raise the priority of T_i.

Each subtask of the critical task T_i would be represented by $T_{i_j} =< \frac{e_i}{k}, \frac{p_i}{k}, \frac{d_i}{k} >$. Though we talked of a critical task being split up into k parts, this is done virtually at a conceptual level, rather than making any changes physically to the task itself. The period transformation technique effectively raises the priority of a critical task to higher values without making the RMA analysis results totally invalid. However, due to the raising of the priority of the critical task, RMA schedulability results do not hold accurately. The culprit is the fact that the utilization due to all the higher priority tasks is in fact k*(e_i /k)/(p_i/k)=k*e_i /p_i, whereas their actual utilization is e_i /p_i. Therefore, even when tasks appear to be unschedulable after task splitting as indicated by the RMA schedulability analysis results, they may actually be schedulable.

10.2 Handling Aperiodic and Sporadic Tasks

Under RMA it is difficult to assign high priority values to sporadic tasks, since a burst of sporadic task arrivals would overload the system and cause many tasks to miss their deadlines. Also, RMA schedulability analysis results that we discussed in this chapter are not any more applicable when aperiodic and sporadic tasks are assigned high priority values. Therefore aperiodic and sporadic tasks are usually assigned very low priority values. However, in many practical situations tasks such as handling emergency conditions are time bound and critical and may require high priority value to be assigned to a sporadic task. In such situations, the following aperiodic server technique can be used.

Aperiodic Server: An aperiodic server handles aperiodic and sporadic tasks as they arise, selects them at appropriate times, and passes them to an RMA scheduler. It makes an otherwise difficult to analyze aperiodic and sporadic tasks suitable for schedulability analysis. The server deposits a "ticket", which is replenished at the expiration of a certain replenishment period. When an aperiodic event occurs, the server checks to see if any ticket is available. If it is, the system immediately passes on the arriving task to the scheduler. It then creates another ticket based on the specific ticket creation policy it uses. An aperiodic server makes aperiodic tasks more predictable, and hence makes them suitable for analysis with RMA. Based on the ticket creation policies, there are essentially two types of aperiodic servers: *deferrable* and *sporadic*. Of these, the sporadic server results in higher schedulable utilization and lends itself more easily to analysis. However, it is more complex to implement. In the following we briefly discuss these two types of servers.

In a **deferrable server**, tickets are replenished at regular intervals, completely independent of the actual ticket usage. If an aperiodic task arrives, the system will process it immediately if it has enough tickets, and wait till the tickets are replenished if it does not. Thus, there can be bursts of tasks sent to the scheduler and also when no task is sent over a duration, tickets are accumulated. While a deferrable server is simpler to implement compared to a sporadic server, it deviates from the RMA strict periodic execution model, which leads to conservative system design and low processor utilization.

In a **sporadic server**, the replenishment time depends on exact ticket usage time. As soon as a ticket is used, the system sets a timer that replaces any used tickets when it goes off. A sporadic server therefore guarantees a minimum separation between two instances of a task. Thus, it helps us to consider a sporadic or aperiodic task as a periodic task during schedulability analysis and assign a suitable priority to it. Of course, in some periods the aperiodic or sporadic task may not arise. This leads to some unavoidable CPU idling.

In effect, an aperiodic server helps overcome the nondeterministic nature of aperiodic and sporadic tasks and help consider them to a fast approximation as periodic tasks with period equal to the token generation time.

10.3 Coping With Limited Priority Levels

While developing a real-time system, engineers some times face a situation where the number of real-time tasks in the application exceed the number of distinct priority levels supported by the underlying operating system. This situation often occurs while developing small embedded applications, where the embedded computers have severe limitations on memory size. Even otherwise, real-time operating systems normally do not support too many priority levels (why?). Also, often out of the total number of priority levels supported by an operating system, only a few are earmarked as real-time levels. The number of priority values typically varies from 8 to 256 in commercial operating systems. As the number of priority levels increases, the number of feedback queues to be maintained by the operating system (see Fig. 15) also increases. This not only increases the memory requirement, but also increases the processing overhead at every scheduling point. At every scheduling point, the operating system selects the highest priority task by scanning all the priority queues. Scanning a large number of queues would incur considerable computational overhead. To reduce the operating system overload and to improve the task response times, real-time operating systems support only a restricted number of priority levels.

When there are more real-time tasks in an application than the number of priority levels available from the operating system, more than one task would have to be made to share a single priority value among themselves. This of course would result in lower achievable processor utilization than what was predicted by the RMA schedulability expressions. This would be apparent from the following analysis.

Let us now analyze the effect of sharing a single priority level among a set of tasks. First, let us investigate how the Lehoczky's test (Expr. 2.12) would change when priority sharing among tasks is considered. Let $SP(T_i)$ be the set of all tasks sharing a single priority value with the task T_i. Then, for a set of tasks to be schedulable,

$$e_i + b_{it} + \sum_{T_k \in SP(T_i)} e_k + \sum_{k=1}^{i-1} \lceil \frac{p_i}{p_k} \rceil * e_k \leq p_i \qquad \ldots (2.17)$$

The term $\sum_{T_k \in SP(T_i)} e_k$ is necessary since a round first come first served (FCFS) scheduling policy is assumed among equal priority tasks. In this case, the task T_k can be blocked by an equal priority task only once unlike the higher priority tasks (third term in the expression) which block T_k at their every arrival during the execution of T_k. An equal priority task unlike higher priority tasks can block a task only once.

To understand why a task may get blocked by an equal priority task at most once, let us consider the following example. Suppose T_i and T_j share the same priority level, but T_j has a much shorter period compared T_i. A task instance T_i can get blocked by an equal priority task T_j only once (for the duration of task T_j) even though task T_j might have a shorter period compared to T_i. This is because once T_j completed its execution, T_i would get to execute; T_i would execute to completion since T_j instances that may arrive during the execution can not preempt T_i. A task instance T_i can get blocked by an equal priority task T_j only once (for the duration of task T_j).

Priority Assignment to Tasks:
It is clear that when there are more real-time tasks tasks with distinct relative deadlines than the number of priority levels supported by the operating system being used, some tasks would have to share the same priority value. However, the exact method of assigning priorities to tasks can significantly affect the achievable processor utilization.

Assigning priority to tasks: It is not difficult to reason that randomly selecting tasks for sharing priority levels would severely lower the achievable schedulable utilization. Therefore systematic ways of selecting which tasks need to share a priority value are required.

A large number of schemes are available for assigning priority values to tasks when the number of real-time tasks to be scheduled exceeds the distinct real-time priority levels supported by the underlying operating system. Some of the important schemes that are being used are the following:

- Uniform Scheme.
- Arithmetic Scheme.
- Geometric Scheme.
- Logarithmic Scheme.

We now discuss these schemes in some detail.

1. Uniform Scheme: In this scheme, all the tasks in the application are uniformly divided among the available priority levels. Uniform division of tasks among the available priority levels can easily be achieved when the number of priority levels squarely divides the number of tasks to be scheduled. If uniform division is not possible, then more tasks should be made to share the lower priority levels (i.e. higher priority values are shared by lesser number of tasks) for better schedulability. Accordingly, if there are N number of tasks and n priority levels, then $\lfloor \frac{N}{n} \rfloor$ number of task are assigned to each level and the rest of the tasks are distributed among the lower priority levels. The uniform priority assignment scheme has been illustrated through the following example.

Example 15: In a certain application being developed, there are 10 periodic real-time tasks $T_1, T_2, ..., T_{10}$ whose periods are: 5,6, ...,14 milli secs respectively. These tasks are to be scheduled using RMA. However, only 4 priority levels are supported by the underlying operating system. In this operating system, the lower the priority value, the higher is the priority of the task. Assign suitable priority levels to tasks using the uniform assignment scheme for scheduling the tasks using RMA.

Solution:
Since the number of priority levels does not squarely divide the number of tasks to be scheduled, some priority values have to be assigned more tasks than the others. The tasks are already sorted in ascending order of their periods (i.e. in decreasing order of their priorities). First let us uniformly divide the tasks among the priority levels, each level is assigned two tasks each. The rest of the tasks can now be distributed among the lower priority levels. A possibility is that the two lower priority levels are assigned three tasks each. From this, the following is a suitable task assignment scheme:
Priority Level 1: T_1, T_2.
Priority Level 2: T_3, T_4
Priority Level 3: T_5, T_6, T_7
Priority Level 4: T_8, T_9, T_{10}

□

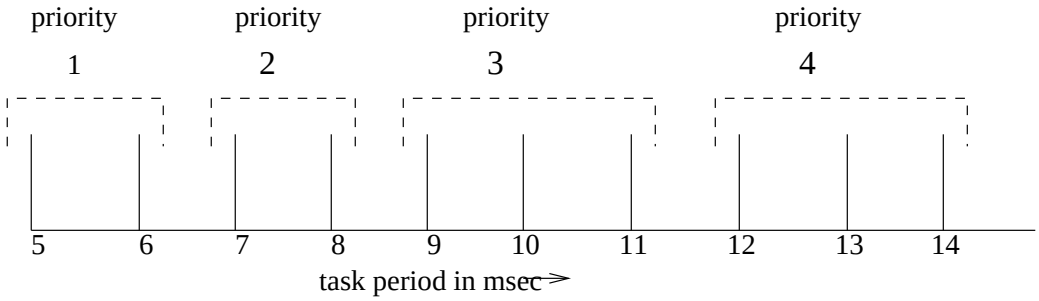


Figure 16: Uniform Priority Assignment to Tasks of Example 15

This priority assignment scheme has been pictorially shown in Fig. 16.

2. Arithmetic Scheme: In this scheme, the number of tasks assigned to different priority levels form an arithmetic progression. A possibility is that "r" tasks having the shortest periods are assigned to the highest priority level, 2r tasks are assigned the next highest priority level, and so on. Let N be the total number of tasks. Then, N=r+2r+3r+4r+....nr, where n is the total number of priority levels.

3. Geometric Scheme: In this scheme, the number of tasks assigned to different priority levels form a geometric progression. This means that if r tasks having the shortest periods are assigned the highest priority, then the next kr^2 tasks are assigned the immediately lower priority, and so on. Therefore, if N is the total number of tasks, and n is the total number of priority levels then $N = r + kr^2 + kr^3 + kr^4 + ...kr^n$.

4. Logarithmic Scheme: The logarithmic scheme is also popularly known as the *logarithmic grid assignment scheme*. The basic idea behind the logarithmic grid assignment scheme is that the shorter period (higher priority) tasks should be allotted distinct priority levels as much as possible. Many lower priority tasks on the other hand, can be clubbed together at the same priority levels without causing any problem to the schedulability of the high priority tasks.

To achieve logarithmic grid assignment, the tasks are first arranged in increasing order of their periods. For priority allocation, the range of task periods are divided into a sequence of logarithmic intervals. The tasks can then be assigned to priority levels based on the logarithmic interval they belong to. In this scheme, if p_{max} is the maximum period among the tasks and p_{min} is the minimum period among the tasks, then r is calculated as $r = (\frac{p_{max}}{p_{\min}})^{\frac{1}{n}}$, where n is the total number of priority levels. Tasks with periods up to r are assigned to the highest priority, tasks with periods in the range r to r^2 are assigned to next highest priority level (assuming k=1 for simplicity), tasks with periods in the range of r^2 to r^3 are assigned to the next highest level, and so on. Simulation experiments have shown that the logarithmic priority assignment works very well for practical problems.

Note that logarithmic task assignment works well only when the task periods are uniformly distributed over an interval. However, if most of the task periods are clustered over a small part of the interval, and the other tasks are sparsely distributed in the rest of the interval, then the logarithmic scheme may yield poor results.

The logarithmic grid assignment scheme has been illustrated in the following Example.

Example 16: Consider an operating system supporting only four priority levels. An application with 10 periodic real-time tasks are to be scheduled on this operating system using RMA. It is also known that of the given tasks, the largest period is 10,000 milli seconds and the shortest period is 1 milli second. Other task periods are distributed uniformly over this interval. Assign the tasks to priority levels using the logarithmic grid assignment scheme.

Solution. $r = (\frac{10000}{1})^{\frac{1}{4}}$=10.
Accordingly, tasks with periods in the range of 1 milli second and 10 milli seconds would be assigned to the highest priority level. Tasks with periods in the range of 11 to 100 milli seconds would be assigned to the the next lower priority level, and so on. □

10.4 Dealing With Task Jitter

We have already defined task jitter as the magnitude of variation in the arrival or completion times of a periodic task. That is, the arrival time jitter is given by the latest arrival time minus the earliest arrival time among all instances of the task. Similarly, the completion time jitter is given by the latest completion time minus the earliest completion time of the task. Presence of small amounts of arrival time jitter is normally unavoidable as all physical clocks show some amount of skew. Completion time jitters are caused by the basic nature of RMA scheduling which schedules a task at the earliest opportunity at which it can run. Thus, the response time of a task depends on how many higher priority tasks arrive (or were waiting) during the execution of the task. Small amounts of jitter normally do not

cause much problems as long as the arrival and completion times of tasks all stay within certain tolerance bounds. However, certain applications might require that jitter be minimized as much as possible.

Real-time programmers commonly handle tasks with tight completion time jitter requirements using any one of the following two techniques:

- If only one or two actions (tasks) have tight jitter requirements, these actions are assigned very high priority. This method works well only when there are a very small number of actions (tasks). When it is used in an application in which the tasks are barely schedulable, it may result in some tasks missing their respective deadlines.
- If jitter must be minimized for an application that is barely schedulable, each task needs to be split into two: one which computes the output but does not pass it on, and one which passes the output on. This method involves setting the second task's priority to very high values and its period to be the same as that of the first task. An action scheduled with this approach will run one cycle behind schedule, but the tasks will have tight completion time jitter.

SUMMARY

- Scheduling of real-time tasks on a uniprocessor was an area of intense research in the 1970s and the underlying theory is now well-developed.
- Uniprocessor real-time task scheduling algorithms can broadly be classified into clock-driven, event-driven, and hybrid algorithms.
- In clock-driven schedulers, the scheduling points are defined by the interrupts generated by the system clock. Important clock-driven schedulers are table-based and cyclic. Cyclic schedulers are being predominantly used in small embedded applications due to their simplicity and low run time overhead.
- Among the large number of results that are available in event-driven scheduling of real-time tasks on a uniprocessor, two algorithms are most significant:

  EDF (Earliest Deadline First): This is an optimal dynamic priority scheduling algorithm.

  RMA (Rate Monotonic Analysis): This is an optimal static priority scheduling algorithm.
- Though EDF is an optimal real-time task scheduling algorithm on a uniprocessor, it suffers from a few shortcomings. It cannot guarantee that the critical tasks meet their respective deadlines under transient overload. Besides, implementation of resource sharing among real-time tasks is extremely difficult. Therefore, EDF-based algorithms are rarely used in practice and RMA-based scheduling algorithms have become popular.
- We discussed the main results concerning how to determine the schedulability of a set of tasks under EDF and RMA.
- We also discussed how to overcome some problems that arise while developing a practical real-time system. In particular, we discussed how to handle situations such as when task priorities differ from task criticalities, and assigning priorities to tasks for RMA scheduling when the priority levels supported by the operating system is less than the number of tasks to be scheduled.

EXERCISES

1. State whether the following assertions are **T**rue or **F**alse. Write one or two sentences to justify your choice in each case.

   (a) Average response time is an important performance metric for real-time operating systems handling running of hard real-time tasks.

   (b) Unlike table-driven schedulers, cyclic schedulers do not require to store a precomputed schedule.

   (c) 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.

   (d) 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.

   (e) A time-sliced round-robin scheduler uses preemptive scheduling.

   (f) The minimum period for which a table-driven scheduler scheduling n periodic tasks needs to pre-store the schedule is given by $\max\{p_1, p_2, ...p_n\}$, where p_i is the period of the task T_i.

   (g) RMA is an optimal static priority scheduling algorithm to schedule a set of periodic real-time tasks on a non-preemptive operating system.

   (h) A cyclic scheduler is more proficient than a pure table-driven scheduler for scheduling a set of hard real-time tasks.

   (i) RMA is an optimal static priority scheduler in the general case where the task periods and deadlines of a set of hard real-time periodic tasks may differ.

   (j) 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.

   (k) A suitable figure of merit to compare the performance of different hard real-time task scheduling algorithms can be the average task response times resulting from each algorithm.

   (l) 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.

   (m) EDF (Earliest Deadline First) algorithm possesses good transient overload handling capability.

   (n) A time-sliced round robin scheduler is an example of a non-preemptive scheduler.

   (o) RMA (Rate Monotonic Analysis) is an optimal algorithm for scheduling static priority non-preemptive periodic real-time tasks having hard deadlines.

   (p) 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.

   (q) Suppose we have a set of real-time independent tasks to be run on a uniprocessor under RMA scheduling. We observe that a task of period 10 milli Sec has met its deadline. From this observation, it would be safe to conclude that a task having period of 5 milli Sec would definitely have met its deadlines,

   (r) Cyclic schedulers are more proficient than table-driven schedulers.

   (s) In a nonpreemptable operating system employing RMA scheduling of 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.

   (t) The worst-case response time for a task occurs when it is out of phase with its higher priority tasks.

   (u) A sporadic server used to handle aperiodic and sporadic tasks for RMA scheduling achieves higher schedulable utilization compared to a deferrable server.

   (v) While using a cyclic scheduler to schedule a set of real-time tasks on a uniprocessor, when a suitable frame size satisfying all the three required constraints has been found, it is guaranteed that the task set would be feasibly scheduled by the cyclic scheduler.

(w) When more than one frame satisfy all the constraints on frame size while scheduling a set of hard real-time periodic tasks using a cyclic scheduler, the largest of these frame sizes should be chosen.

(x) Good real-time task scheduling algorithms ensure fairness to real-time tasks while scheduling.

2. State whether the following assertions are **T**rue or **F**alse. Write one or two sentences to justify your choice in each case.

(a) In table-driven scheduling of three periodic tasks T_1,T_2,T_3, the scheduling table must have schedules for all tasks drawn up to the time interval [0,max(p1,p2,p3)], where pi is the period of the task T_i.

(b) When a set of hard real-time periodic tasks are being scheduled using a cyclic scheduler, if a certain frame size is found to be not suitable, then any frame size smaller than this would not also be suitable for scheduling the tasks.

(c) When a set of hard real-time periodic tasks are being scheduled using a cyclic scheduler, if a candidate frame size exceeds the execution time of every task and squarely divides the major cycle, then it would be a suitable frame size to schedule the given set of tasks.

(d) Finding an optimal schedule for a set of independent periodic hard real-time tasks without any resource-sharing constraints under static priority conditions is an NP-complete problem.

(e) The EDF algorithm is optimal for scheduling real-time tasks in a uniprocessor in a non-preemptive environment.

(f) 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.

(g) 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

(h) In a non-preemptive event-driven task scheduler, scheduling decisions are made only at the arrival and completion of tasks.

(i) 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

(j) The EDF scheduling algorithm needs to frequently examine the ready queue of the tasks at regular intervals to determine which task should start running next.

(k) In RMA scheduling, if you observe the sequence in which a set of periodic real-time tasks {T_1, T_2, ... Tn} are taken up for execution, the task execution pattern would repeat every LCM($p_1, p_2, ..., p_n$) interval, where p_i is the period of the task T_i.

(l) 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.

(m) For scheduling real-time tasks in practical uniprocessor based real-time systems, sub-optimal heuristic scheduling algorithms are normally used as optimal scheduling algorithms are computationally intractable.

(n) The RMA schedulability of a set of periodic real-time tasks would be higher if their periods are harmonically related compared to the case where their periods are not related.

(o) RMA should be preferred over the time-sliced round-robin algorithm for scheduling a set of soft real-time tasks on a uniprocessor.

(p) 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.

(q) 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.

(r) Assume that a task set can be scheduled under RMA and every task meets its corresponding deadline when no task self suspends. In this system, a task might miss its deadline when a higher priority task self suspends for some finite duration.

(s) In a nonpreemptive scheduler, scheduling decisions are made only at the arrival and completion of tasks.

(t) A time sliced round robin scheduler uses preemptive scheduling.

(u) It is not possible to have a system that is safe and at the same time is unreliable.

(v) Precedence ordering among a set of tasks is essentially determined by the data dependency among them.

(w) When a set of periodic real-time tasks are being scheduled on a uniprocessor using RMA scheduling, all tasks would show similar completion time jitter.

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. Identify the constraints that a set of periodic real-time tasks need to satisfy for RMA to be an optimal scheduler for the set of tasks?

5. Real-time tasks are normally classified into periodic, aperiodic, and sporadic real-time task.

   (a) What are the basic criteria based on which a real-time task can be determined to belong to one of the three categories?

   (b) Identify some characteristics that are unique to each of the three categories of tasks.

   (c) Give examples of tasks in practical systems which belong to each of the three categories.

6. What do you understand by an optimal scheduling algorithm? Is it true that the time complexity of an optimal scheduling algorithm for scheduling a set of real-time tasks in a uniprocessor is prohibitively expensive to be of any practical use? Explain your answer.

7. What do you understand by jitter associated with a periodic task? How are these jitters caused? How can they be overcome?

8. Suppose a set of three periodic tasks is to be scheduled using a cyclic scheduler on a uniprocessor. Assume that the CPU utilization due to the three three tasks is less than 1. Also, assume that for each of the three tasks, the deadlines equals the respective periods. Suppose that we are able to find an appropriate frame size (without having to split any of the tasks) that satisfies the three constraints of minimization of context switches, minimization of schedule table size, and satisfaction of deadlines. Does this imply that it is possible to assert that we can feasibly schedule the three tasks using the cyclic scheduler? If you answer affirmatively, then prove your answer. If you answer negatively, then show an example involving three tasks that disproves the assertion.

9. Classify the existing algorithms for scheduling real-time tasks into a few broad classes. Explain the important features of these broad classes of task scheduling algorithms.

10. Consider a real-time system which consists of three tasks T_1, T_2, and T_3, which have been characterized in the following table.

Task	Phase mSec	Execution time mSec	Relative Deadline mSec	Period mSec
T_1	20	10	20	20
T_2	40	10	50	50
T_2	70	20	80	80

If the tasks are to be scheduled using a table-driven scheduler, what is the length of time for which the schedules have to be stored in the precomputed schedule table of the scheduler.

11. 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 negatively, explain what is the concept of priority in EDF then.

12. 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.

13. A preemptive static priority real-time task scheduler is used to schedule two periodic tasks T_1 and T_2 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 T_1 has higher priority than T_2. 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.

14. Assume that a preemptive priority-based system consists of two periodic foreground tasks T_1, T_2, and T_3 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

T_1 has higher priority than T_2 and T_2 has higher priority than T_3. 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.

15. A cyclic real-time scheduler is to be used to schedule three periodic tasks T_1, T_2, and T_3 with the following characteristics:

Task	Phase mSec	Execution time mSec	Relative Deadline mSec	Period mSec
T_1	0	20	100	100
T_2	0	20	80	80
T_2	0	30	150	150

Suggest a suitable frame size that can be used. Show all intermediate steps in your calculations.

16. 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	40	10	50	30
T_3	60	50	200	150

Suppose a cyclic scheduler is to be used to schedule the task set. What is the major cycle of the task set? Suggest a suitable frame size and provide a feasible schedule (task to frame assignment for a major cycle) for the task set.

17. 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 T_3 is more critical than task T_2. Check whether the task set can be feasibly scheduled using RMA.

18. 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.

19. Construct an example involving more than one hard real-time periodic task whose aggregate processor utilization is 1, and yet schedulable under RMA.

20. Explain the difference between clock-driven, event-driven, and hybrid schedulers for real-time tasks. Which type of scheduler would be preferred for scheduling three periodic tasks in an embedded application. Justify your answer.

21. 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

22. 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	40	10	50	30
T_3	60	50	200	150

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

23. 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

24. Construct an example involving two periodic real-time tasks for 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

25. Briefly explain while scheduling a set of hard real-time periodic tasks, why RMA can not achieve 100% processor utilization without missing task deadlines.

26. 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.

27. Construct an example involving three arbitrary real-time periodic tasks to be scheduled on a uniprocessor, for whom the task schedules worked out by EDF and RMA would be different.

28. Construct an example involving three periodic real-time tasks (for each task, task period should be equal to its deadline) which would be schedulable under EDF but unschedulable under RMA. Justify why your example is correct.

29. 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.

30. Three hard real-time periodic tasks T_1=(50mSec, 100mSec, 100mSec), T_2=(70mSec, 200mSec, 200mSec), and T_3=(60mSec, 400mSec, 400mSec) are to be scheduled on a uniprocessor using RMA. Can the task set be feasibly be scheduled? Suppose context switch overhead of 1 milli Seconds is to be taken into account, determine the schedulability.

31. Consider the following three real-time periodic tasks.

Task	Start-time (mSec)	Processing-time (mSec)	Period (mSec)	Deadline(mSec)
T_1	20	25	150	150
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 T_1, T_2, and T_3 self suspend for 10mSec, 20 mSec, and 15mSec 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 T_1 and T_2 are assigned the same priority value, determine the additional delay in response time that T_2 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.

(e) Assume that T_1, T_2, and T_3, each require certain critical section C during their computation. Would it be correct to assert that T_1 and T_3 would not undergo any priority inversion due to T_2? Justify your answer. Ignore the self suspension times and the context switch overheads for this part of the question.

(f) Assume that T_3 is known to be the most critical of the three tasks. Explain a suitable scheme by which it may be possible to ensure that T_3 does not miss its deadlines under transient overload conditions when the task set is scheduled using RMA.

(g) Assume that you have been asked to implement an EDF task scheduler for an application where efficiency of the scheduling algorithm is a very important concern::

i. Explain the data structure you would use to maintain the ready list. What would be the complexity of inserting and removing tasks from the ready list.

ii. Identify the events which would trigger your scheduler.

(h) Explain how an EDF (Earliest Deadline First) scheduler for scheduling a set of real-time tasks can be implemented most efficiently. What is the complexity of handling a task arrival in your implementation? Explain your answer.

(i) Assume that T_2 and T_3, each require certain critical resource CR1 during their computation. T_2 needs CR1 for 10 mSec and T_3 needs CR1 for 20 mSec. Also, tasks T_1 and T_2, each require critical resource CR2 during their computation. T_1 needs CR1 for 15 mSec and T_2 needs CR2 for 5 mSec. Assume that each task instance tries to get all its required resources before starting to use any of the resources. Also, a task instance returns a resource as soon as it completes computing using it. Once a task instance returns a resource, it does not try to acquire it again. If PCP (priority ceiling protocol) is used for resource arbitration, determine if the task set is schedulable when the context switching overhead of 1mSec is considered (assume no self suspension of tasks).

(j) Typically what is the number of priority levels supported by any commercial real-time operating system that you are aware of? What is the minimum number of priority levels required by RT-POSIX? Why do operating systems restrict the number of priority levels they support?

References

[1] Liu C. and Layland J.W. Scheduling algorithms for multiprogramming in hard real-time environment. *Journal of ACM*, Vol. 20(1):46–61, 1973.

[2] Krishna C.M. and Shin K.G. *Real-Time Systems*. Tata McGraw-Hill, 1997.

[3] Lehoczky John L. and Lui Sha. The rate-monotonic scheduling algorithm: Exact characterization and average case behavior. *Proceedings of Real-Time Systems Symposium*, pages 166–171, December 1989.

[4] Sha L. and Rajkumar R. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, Vol. 1(No. 3):243–265, December 1989.

Source:http://www.nptel.ac.in/courses/Webcourse-contents/IIT%20Kharagpur/Real%20time%20system/pdf/Module2.pdf