Module 6

Embedded System Software

Lesson 29

Real-Time Task Scheduling – Part 1

Specific Instructional Objectives

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

- Understand the basic terminologies associated with Real-Time task scheduling
- Classify the Real-Time tasks with respect to their recurrence
- Get an overview of the different types of schedulers
- Get an overview of the various ways of classifying scheduling algorithms
- Understand the logic of clock-driven scheduling
- Get an overview of table-driven schedulers
- Get an overview of cyclic schedulers
- Work out problems related to table-driven and cyclic schedulers
- Understand how a generalized task scheduler would be
- Compare table-driven and cyclic schedulers

1. 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 milliseconds 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.

1.1. Basic Terminologies

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 Ti would be denoted as Ti(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.

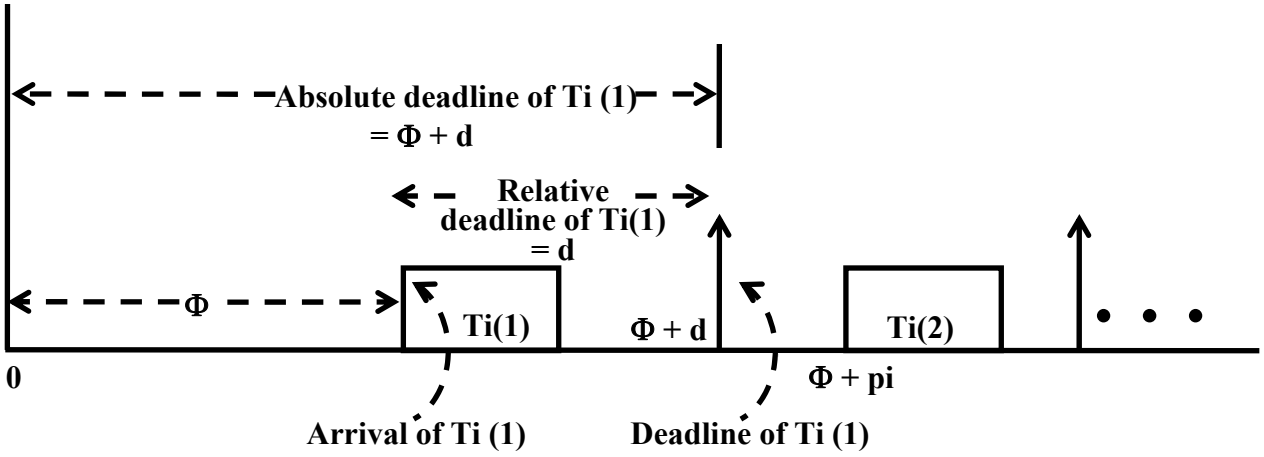

Fig. 29.1 Relative and Absolute Deadlines of a Task

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 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. 29.1. It can be observed from Fig. 29.1 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 Ti precedes another task Tj, then each instance of Ti precedes the corresponding instance of Tj. That is, if T1 precedes T2, then T1(1) precedes T2(1), T1(2) precedes T2(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. 29.2. Here T1 precedes T2, but we cannot relate T1 with either T3 or T4. We shall later use task precedence relation to develop appropriate task scheduling algorithms.

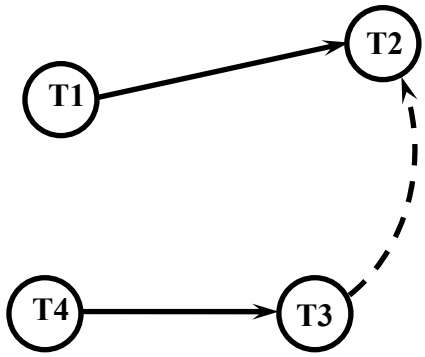

Fig. 29.2 Precedence Relation among Tasks

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 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. 29.2, T2 uses the results of T3, but T2 and T3 may execute concurrently. T2 may even

start executing first, after sometimes it may receive some data from T3, and continue its execution, and so on.

1.2. Types of Real-Time Tasks

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 Task: 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 Ti is a periodic task, then the time from 0 till the occurrence of the first instance of Ti (i.e. Ti(1)) is denoted by ϕ_i, and is called the phase of the task. The second instance (i.e. Ti(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 Ti 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 future discussions.

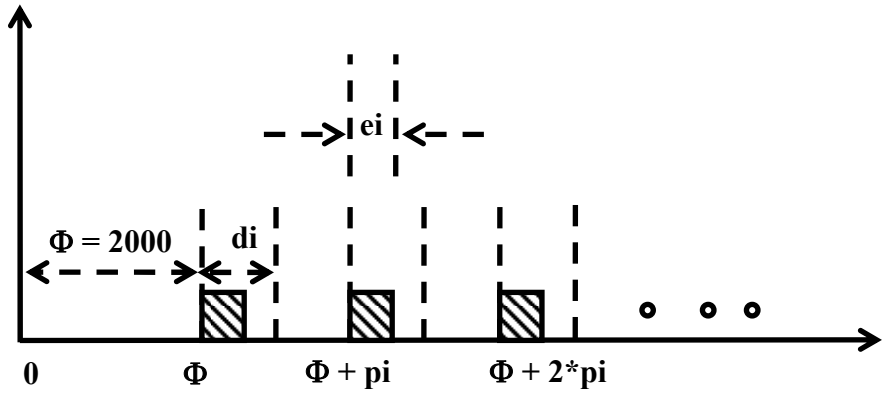

Fig. 29.3 Track Correction Task (2000mSec; p_i; e_i; d_i) of a Rocket

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 milliseconds after the launch of the rocket, and recurs periodically every 50 milliseconds then on. Each instance of the task requires a processing time of 8 milliseconds and its relative deadline is 50 milliseconds. 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 milliseconds. This task can formally be represented as (2000 mSec, 50 mSec, 8 mSec, 50 mSec). This task is pictorially shown in Fig. 29.3. When the deadline of a task equals its period (i.e. pi=di), we can omit the fourth tuple. In this case, we can represent the task as Ti= (2000 mSec, 50 mSec, 8 mSec). This would automatically mean pi=di=50 mSec. Similarly, when $\phi_i = 0$, it can be omitted when no confusion arises. So, Ti = (20mSec; 100mSec) would indicate a task with $\phi_i = 0$, pi=100mSec, ei=20mSec, and di=100mSec. Whenever there is any scope for confusion, we shall explicitly write out the parameters Ti = (pi=50 mSecs, ei = 8 mSecs, di = 40 mSecs), etc.

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 normally get 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.

Sporadic Task: A sporadic task is one that recurs at random instants. A sporadic task Ti 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 gi 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.

1.3. 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 grueling 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.

1.3.1. A Few Basic Concepts

Before focusing 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 determined by occurrence of certain events.

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 $u_i = e_i/p_i$, where e_i is the execution time and p_i is the period of T_i. For a set of periodic tasks {Ti}: the total utilization due to all tasks $U = \sum_{i=1}^{n} e_i/p_i$. 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.

1.4. 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. 2.5.

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 dead- lines, 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 up-to-date state information available in distributed systems. This makes uniprocessor scheduling algorithms that assume 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 over- head. So carefully designed distributed algorithms are normally considered suitable for use in a distributed system. In the following sections, we study the different classes of schedulers in more detail.

1.5. 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 is also called static scheduler.

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

1.5.1. Table-Driven Scheduling

Table-driven schedulers usually pre-compute 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 T1 would be taken up for execution at time instant 0, T_2 would start execution 3 milliseconds afterwards, 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, ..., T_n. For example, if we have the following three tasks: (e1=5 msecs, p1=20 msecs), (e2=20 msecs, p2=100 msecs), (e3=30 msecs, p3=250 msecs); then, the schedule will repeat after every 1000 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 millisecs***
T_1	*0*
T_2	*3*
T_3	*10*
T_4	*12*
T_5	*17*

Table 29.1 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 result of an investigation into this issue has been given as Theorem 2.1.

1.5.2. Theorem 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 phasing.

Proof: As per our definition of a major cycle, even when tasks have non-zero phasing, task instances would repeat the same way in each major cycle. Let us consider an example in which the occurrences of a task Ti in a major cycle be as shown in Fig. 29.4. As shown in the example of Fig. 29.4, there are k-1 occurrences of the task Ti during a major cycle. The first occurrence of Ti 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 Ti in the major cycle. Of course, this must be the same in each major cycle.

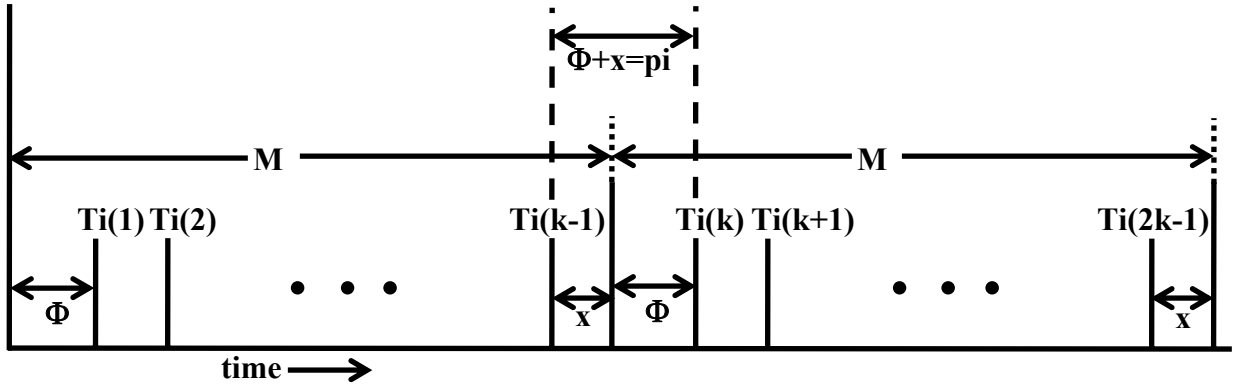

Fig. 29.4 Major Cycle When a Task T_i has Non-Zero Phasing

Assume that the size of each major cycle is M. Then, from an inspection of Fig. 29.4, for the task to repeat identically in each major cycle:

$$M = (k\text{-}1)p_i + \phi + x \qquad \ldots(2.1)$$

Now, for the task T_i to have identical occurrence times in each major cycle, $\phi + x$ must equal to p_i (see Fig. 29.4).

Substituting this in Expr. 2.1, we get, $M = (k\text{-}1) * p_i + p_i = k * p_i$ …(2.2)

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}).

1.5.3. 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.

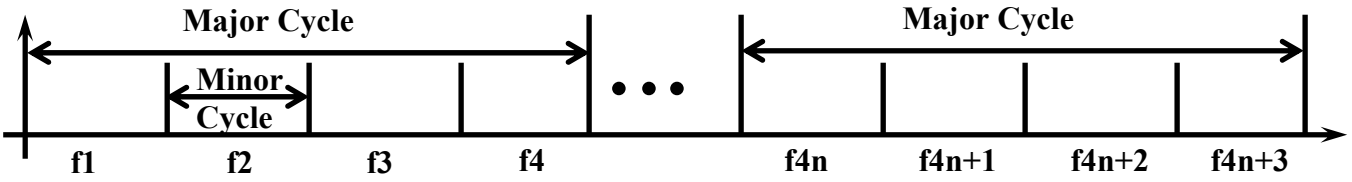

Fig. 29.5 Major and Minor Cycles in a Cyclic Scheduler

A cyclic scheduler repeats a pre-computed schedule. The pre-computed schedule needs to be stored only for one major cycle. 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. 29.5). Each minor cycle is also sometimes called a frame. In the example shown in Fig. 29.5, 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 Figure 29.6.

Task Number	***Frame Number***
T_3	f_1
T_1	f_2
T_3	f_3
T_4	f_4

Fig. 29.6 An Example Schedule Table for 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: $max(\{e_i\}) < F$ 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., 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 a very small storage capacity. So, this constraint is important to the commercial success of a product. The number of entries to be stored in the schedule table can be minimized 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:

$$\lfloor M/F \rfloor = M/F \qquad \ldots(2.3)$$

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

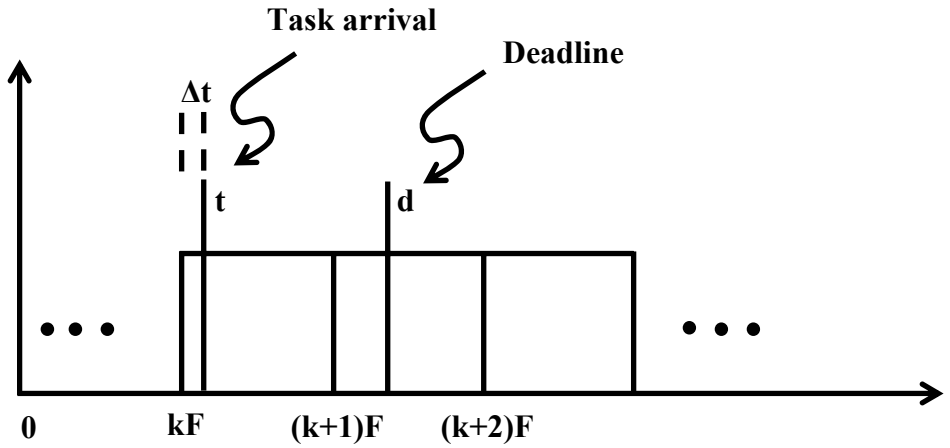

Fig. 29.7 Satisfaction of a Task Deadline

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. 29.7 might arise. In this case, the task arrives sometimes 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+1th frame. But, then it may be too late to meet its deadline since the execution time of a task can be up to 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 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. 29.8, so that task deadlines could be met.

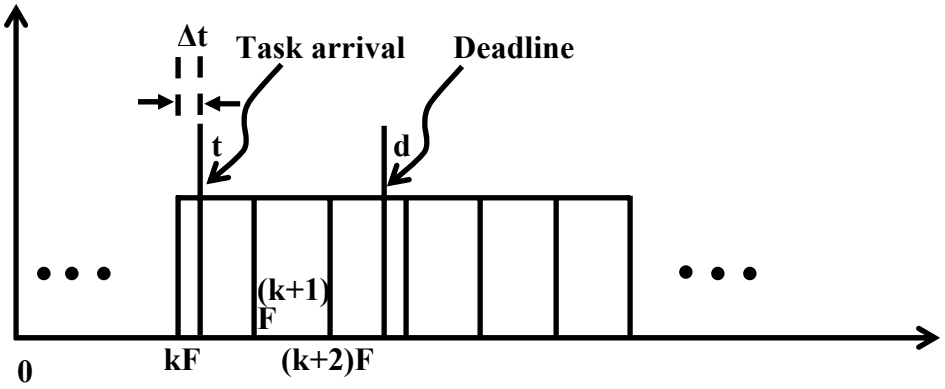

Fig. 29.8 A Full Frame Exists Between the Arrival and Deadline of a Task

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. 29.8). 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* $2F \leq (d_i + \Delta t)$.

…(2.4)

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:

$$\text{for every Ti,} \quad 2F - gcd(F, p_i) \leq d_i \qquad \ldots(2.5)$$

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

1.5.4. Theorem 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 pi. 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 nth task arrival time is less than g. That is, $0 < (m * F - n * p_i) < g$.

Dividing this expression throughout by g, we get:

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

However, F/g and p_i/g are both integers because g is $gcd(F, p_i)$. 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 $0 < m*I_1 - n*I_2 < 1$. 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 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.

1.5.5. Examples

Example 1: A cyclic scheduler is to be used to run the following set of periodic tasks on a uniprocessor: $T_1 = (e_1=1, p_1=4)$, $T_2 = (e_2=, p_2=5)$, $T_3 = (e_3=1, p_3=20)$, $T_4 = (e_4=2, p_4=20)$. 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 {ei, 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: $2F - gcd(F, p_i) < d_i$ for each p_i.

Let us first try frame size 2.

For F = 2 and task T1:

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

Therefore, for p_1 the inequality is satisfied.

Let us try for F = 2 and task T2:

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

Therefore, for p_2 the inequality is satisfied.

Let us try for F = 2 and task T3:

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

Therefore, for p_3 the inequality is satisfied.
For F = 2 and task T4:

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

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 T1:

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

Therefore, for p_1 the inequality is satisfied.
Let us try for F = 4 and task T2:

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

For p2 the inequality is not satisfied. Therefore, we need not 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 T1, we have

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

The inequality is not satisfied for T1. 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 T1, we have

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

The inequality is not satisfied for T1. 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 T1, 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 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: 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 not

schedulable. It is easy to observe that the task T3 has the largest execution time, and consequently due to constraint 1, makes the feasible frame sizes quite large.

We try splitting T3 into two or three tasks. After splitting T3 into three tasks, we have:

$$T_{3.1} = (20, 1, 20), T_{3.2} = (20, 2, 20), T_{3.3} = (20, 2, 20).$$

The possible values of F now are 2 and 4. We can check that now 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. Therefore, this 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 not schedulable by a cyclic scheduler.

1.5.6. 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 pre-computed 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 pre-empted 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.

1.5.7. Comparison of Cyclic with Table-Driven Scheduling

Both table-driven and cyclic schedulers are important clock-driven schedulers. A 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 milliseconds. 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.

1.6. Exercises

1. State whether the following assertions are True or False. 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 pre-computed schedule.

c. The minimum period for which a table-driven scheduler scheduling n periodic tasks needs to pre-store the schedule is given by $\max\{p_1, p_2, \ldots, p_n\}$, where p_i is the period of the task T_i.
d. A cyclic scheduler is more proficient than a pure table-driven scheduler for scheduling a set of hard real-time tasks.
e. 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.
f. Cyclic schedulers are more proficient than table-driven schedulers.
g. 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.
h. When more than one frame satisfies 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.
i. 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 $[0, \max(p_1, p_2, p_3)]$, where p_i is the period of the task T_i.
j. 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.
k. 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.
l. 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.

2. 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.
3. 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.
4. 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 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.
5. Consider a real-time system which consists of three tasks T1, T2, and T3, 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_3	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 pre-computed schedule table of the scheduler.

6. A cyclic real-time scheduler is to be used to schedule three periodic tasks T1, T2, and T3 with the following characteristics:

Task	Phase mSec	Execution Time mSec	Relative Deadline mSec	Period mSec
T_1	0	20	100	100
T_2	0	20	80	80
T_3	0	30	150	150

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

7. 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.