Real-Time Task Scheduling:-

→ 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 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 consist of no. of tasks. The time bounds on different tasks may be different. The consequences of a task missing its time bounds may also vary from task to task. This is often expressed as the criticality of the task.

Some Important Concepts :—

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 occurance times.

⇒ It is possible that real-time tasks recur at random instants. However, most real-time tasks recur with certain fixed periods.

e.g—

A temperature sensing task in a chemical plant might recur indefinitely with a certain period because the temperature is sampled periodically, where as a task handling a device interrupt might recur at random instants.

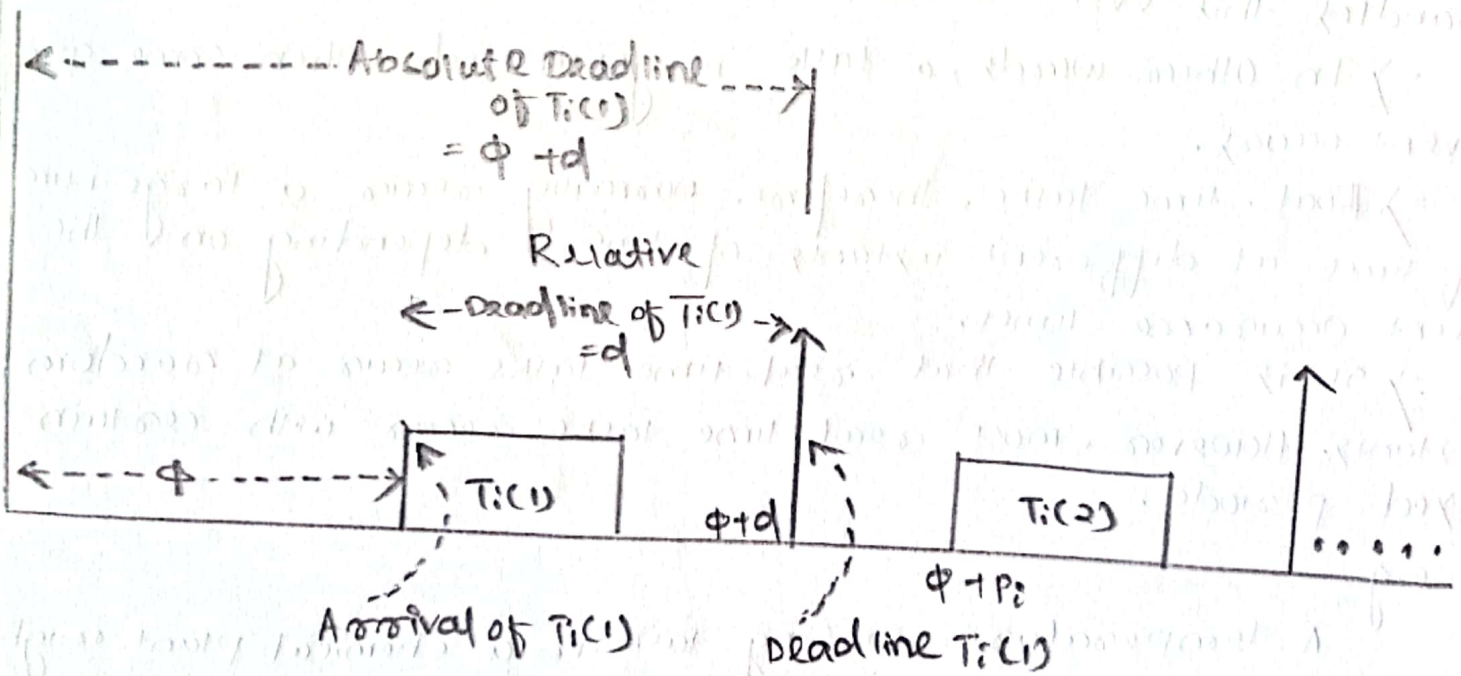
Each time a task recurs, it is called an instance of the task. The first time a task occurs, it is called the first instance of the task. The next occurrence of the task is called its second instance, and so on.

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 tasks are expected.

→ Absolute deadline is equal to the interval of time betn the time 0 and the actual instant at which the deadline occurs as measured by some physical clock.

→ Relative deadline is the time interval between the start of the task and the instant at which the deadline occurs. In other words, relative deadline is the time interval between the arrival of a task and the corresponding dead-line.



(Relative And Absolute Deadline Of a Task)

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.

* Task instances get generated due to occurrence of events. These events may be internal to the system, such a clock interrupts, or external to the system such as a robot encountering an obstracles.

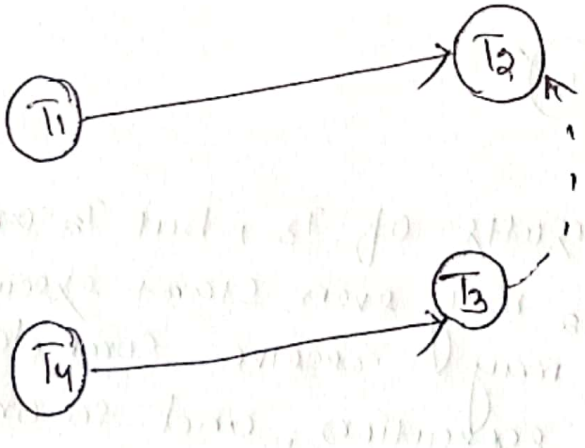
* The response time is the time duration from the occurance of the event generating the task to the time the task produces its results.

Task Precedence :—

→ A task is said to precede another task, if the first task must completed before the second task can start.

→ When the 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.



Here T_1 precedes T_2, but we can't relate T_1 with either T_3 or T_4.

Data Sharing :—

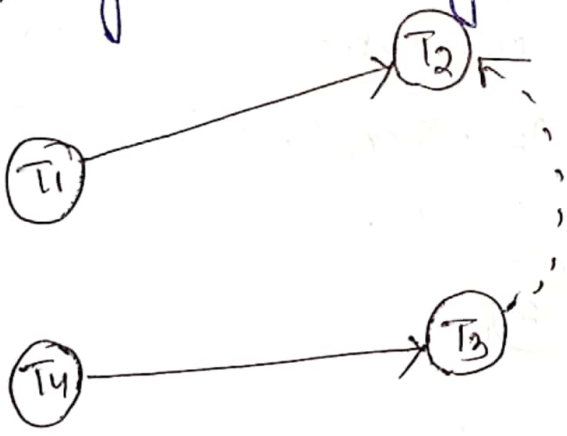
→ Task often need to share their results among each other when one task needs to share the results produced by another task; clearly, the second task must precede the first task.

→ 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).

→ A task may be required to precede that the another even when there is no data sharing.

→ Data sharing may occur not only when one task precedes the other, but might occurs among truly concurrent tasks, and overlapping tasks.

→ In other words, data sharing among tasks does not necessarily impose any particular ordering among tasks.



Here, 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.

TYPES OF REAL-TIME TASKS AND THEIR CHARACTERISTICS :-

→ 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 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 intance (i.e, $T_i(1)$) is denoted by ϕ_i; and is called the phase of the task.

→ The second intance (i.e, $T_i(2)$) occurs at $\phi_i + P_i$.

→ The third intance (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 fourk tuple (ϕ_i, P_i, e_i, d_i) where,

P_i = it is the period of task

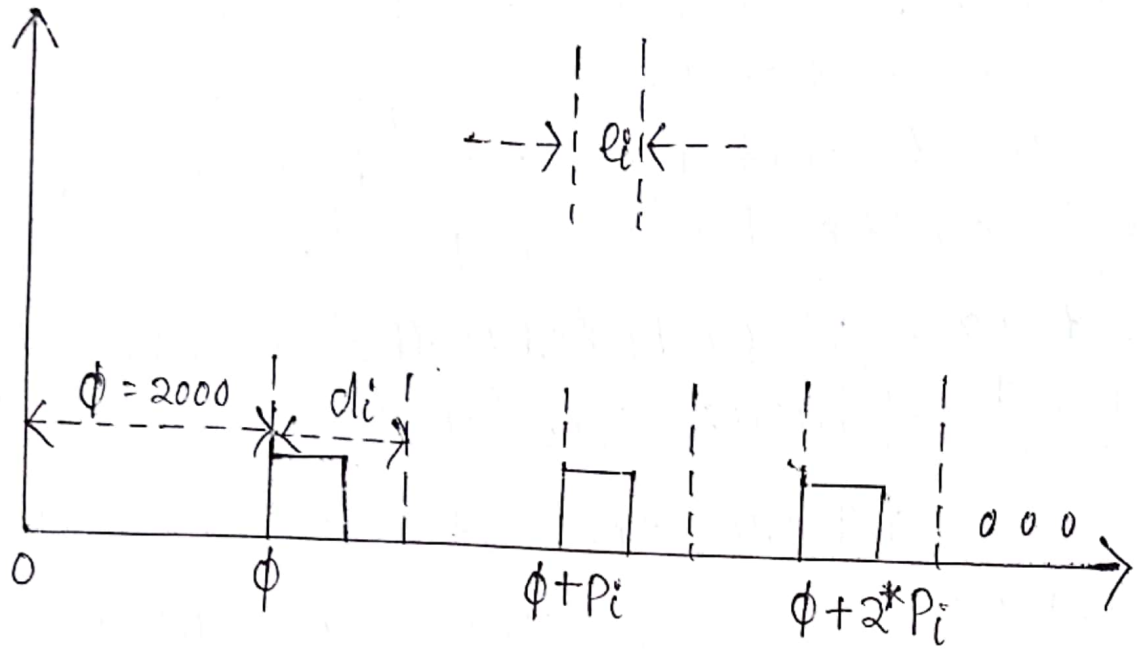
e_i = it is the worst case execution time of the task.

and d_i = it is the relative deadline of the task.

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 2000msec. after the launch of the rocket, and it periodically recurs every 50msec then on. Each instance of the task requires a processing time of 8msec, and its relative deadline is 50msec. 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 2000msec. This task can formally be represented as (2000msec, 50msec, 8msec, 50msec). When the deadline of a task equal its period (i.e., $P_i = d_i$), we can omit the forth tuple. In this case, we can represent the task as T_j = (2000msec, 50msec, 8msec). 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 $\phi_i = 0$, $P_i = 100$ms, $e_i = 20$msec and $d_i = 100$msec. Whenever there is any scope for confusion, we shall explicitly write out the parameters T_i = (P_i = 50msec, e_i = 8msec, d_i = 40msec). etc.



Track correction Task (2000msec, Pi, ei, di) of a Rocket.

SPORADIC TASKS :—

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 = gt is worst case execution time of an instance of the task.

g_i = gt denotes the minimum separation between two consecutive instances of the task.

d_i = gt 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.

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 TASKS :—

→ 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.

→ periodic tasks are generally soft real-time tasks.

It is easy to realize why aperiodic task need to be soft real-time tasks. Aperiodic tasks can occur in quick succession. It, therefore, becomes very difficult to meet the deadlines of all instances of an aperiodic task.

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 request may be spaced out in time. other example of aperiodic tasks include operator request, keyboard presses, mouse movements, etc. In fact all the interactive commands issued by users are handled by aperiodic tasks.

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 Few basic Concepts and Terminologies:

- 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 Schedulers: A task scheduler Sch.2 can said to be more proficient than another scheduler Sch. 2, if Sch. 1 can feasibly schedule all task sets that Sch. 2 can feasibly schedule, but not vice versa.
  That is, Sch. 1 can feasibly schedule all task sets that Sch. 2, but there exists at least one task set that Sch.2

Can, but there exists at least one task set that Sch. 2 can not feasibly schedule, whereas Sch. 1 can. If Sch. 1 can feasibly schedule all task sets that Sch. 2 can feasibly schedule and vice versa, then Sch. 1 and Sch. 2 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 efficient proficient scheduling algorithm than an optimal scheduler.

> If any optimal scheduler can not schedule some task set, then no other schedulers 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 notation:

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 behaviour.

> 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.

> 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.

Classification of Real-Time Task Scheduling Algorithm :-

> The three main types of schedulers are:

i. clock-driven
ii. event-driven
iii. Hybrid

1. **Clock Driven:** The clock-driven schedulers are those in which the scheduling points are determined by the interrupts received from a clock.

> Clock-driven schedulers are simple and efficient. Therefore, these are frequently used in embedded applications.

> The basic features of two clock-driven Schedulers are: i. Table-driven scheduling
ii. Cyclic schedulers.

2. **Event Driven:** In the event-driven ones, the scheduling points are defined by certain events which precedes clock interrupts.

> Event-driven schedulers are more sophisticated and causes usually more proficient and flexible than clock-driven schedulers.

> These are more proficient because they can feasibly schedule some task sets which clock-driven schedulers can not.

> Event-driven are flexible because they can feasibly schedule sporadic & aperiodic tasks in addition to periodic tasks, whereas clock-driven schedulers can satisfactorily handle only periodic tasks.

Examples: Earliest Deadline First (EDF) and Rate Monotonic Analysis (RMA).

> The basic features of three important event-driven schedulers are: i. Simple priority-based
ii. Rate monotonic Analysis (RMA)
iii. Earliest Deadline First (EDF)

3. Hybrid: In hybrid schedulers, the scheduling points are defined both through the clock interrupts and the event occurrences.

A popular hybrid scheduler: Round-Robin.

- Based on the task acceptance test used, there are broad categories of task schedulers: i. planning-based ii. Best effort.

i. Planning-based: In planning based schedulers, when 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 & does not cause other already scheduled tasks to miss their respective deadlines, then the task is accepted for scheduling. Other-wise, it is rejected.

ii. Best effort: In best effort schedulers, no acceptance test is applied. All tasks that arrive are taken up for scheduling & best effort is made to meet its deadlines. But, no guarantee is given as to whether a task's dead-line would be met.

- The different classes of scheduling algorithms are:

i. Uni Processor
ii. Multi Processor
iii. Distributed

i. Uni Processor: Uni processor scheduling algorithm's are possibly the simplest of the three classes of algorithm's.

> In contrast to uni processor algorithms, in multiprocessor and distributed scheduling algorithms first made in which task needs to run on which processor and then these tasks are scheduled.

> Uni Processor scheduling algorithms that assume a central state information of all tasks and processors to exist conveitable for use in distributed system.

i. Multi Processor:

> In contrast to multi processors, 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 system.

iii. Distributed: In distributed systems, the communicati-on among tasks is through message passing which is costly. This means that a good distributed scheduling algorithm should not incur too much communication overhead.

> So, carefully designed distributed algorithms are normally considered suitable for use in a distributed system.

CLOCK-DRIVEN SCHEDULING

- These are those scheduler in which the Scheduling points are defined by the interrupts, generated by a periodic timer.
- These schedulers predetermine which task will run when and accordingly fix the Schedule before the system starts to run. Therefore those schedulars are also known as off-line schedular / static schedular.
- This Schedulers can't handle sporadic and aperiodic task satisfactorily, since the exact occurance of this task can't be predicted.
- In this section, we study the basic features of two important clock-driven schedulers: table driven and cyclic schedulers.

Table-Driven Scheduling

- These schedulers precompates which task will run when and store this information in a scheduled table during the time the system is designed or configured.
- An example of a schedule table shows that task T_1 would be taken up for execution at time instant 0, T_2 would start execution 3msec

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 $(P_1, P_2, \cdots, P_n)$ time units, where $P_1, P_2, \cdots P_n$ are the periods of $T_1, T_2, \cdots$. For example, if we have the following three tasks: $(e_1 = 5\,msec, P_1 = 20\,msec)$, $(e_2 = 20\,msec, P_2 = 100\,msec)$, $(e_3 = 30\,msec, P_3 = 250\,msec)$. Then, the schedule will repeat after every 500 msec. So, for any given task set it is sufficient to store entries only for LCM $(P_1, P_2, \cdots, P_n)$ duration in the schedule table. LCM $(P_1, P_2, \cdots P_n)$ is called the major cycle of the set of tasks ST.

An Example of a Table-Driven Schedule

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

Theorem The major cycle of a set of tasks $ST = \{T_1, T_2, \ldots T_n\}$ is LCM $(\{P_1, P_2, \ldots P_n\})$ even when the tasks have arbitrary phasings.

PROOF

As per our defenation of a major cycle, even when tasks have non-zero phasings, task in-stances would repeat the same way in each major cycle. Let us consider an example in which the occurences of a task T_i in a major cycle be as shown in Fig. 2.4. In Fig. 2.4 there are k-1 occurences of task Ti 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 n time units after the last (i.e, (k-1)th) occurence of the task Ti in the major cycle. of course, this must be the same in each major cycle.

Assume that the size of each major cycle is M. Then, from an inspection of Fig. 2.4 for the task to repeat identically in each major cycle.

$$M = (k-1)p_i + \phi + n$$

Now, for the task Ti to have identical occurence time in each major cycle, $\phi + x$ must equal to Pi (see fig 2.4).

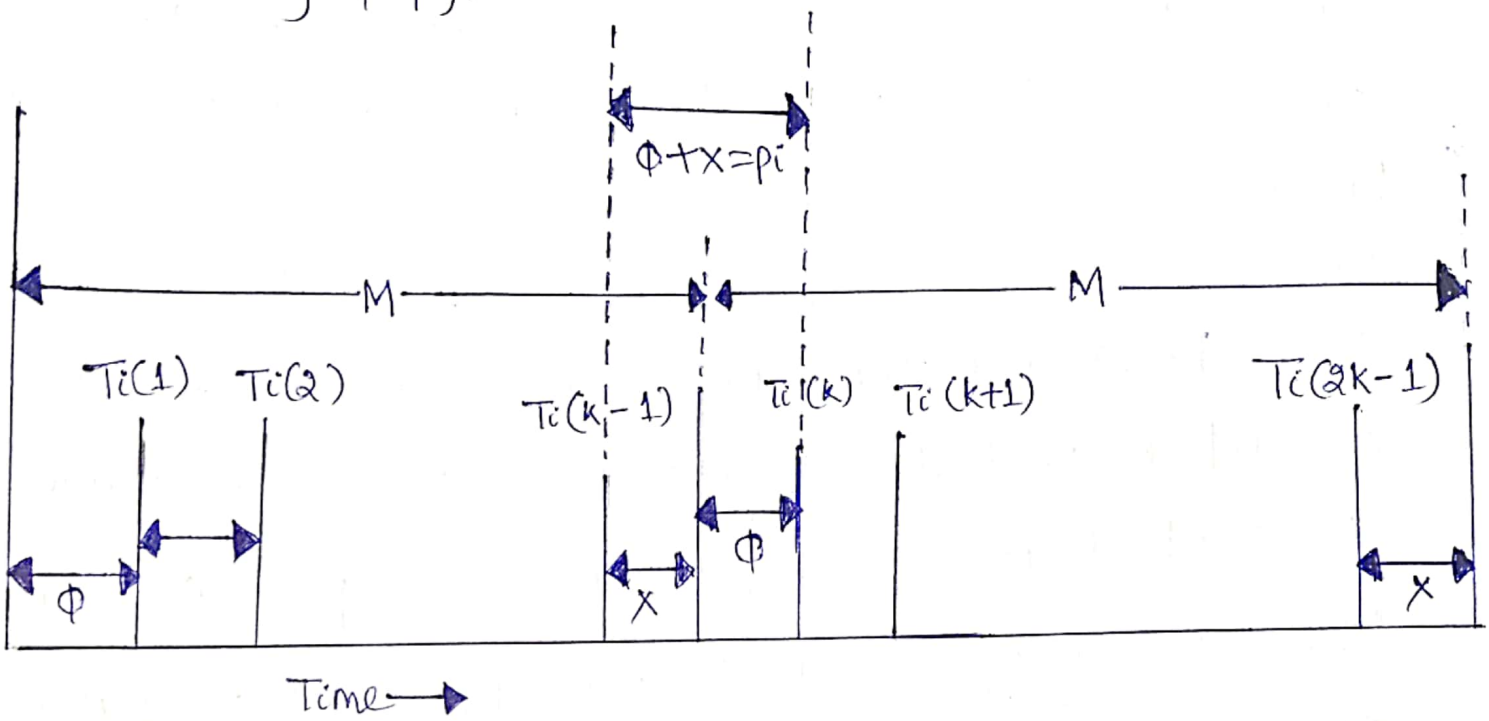


FIGURE 2.4

Major cycle when a Task Ti has Non-Zero phasing Substituting this in Expr. 2.1 we get, $M = (k-1) * Pi + Pi = k * Pi$

So, the major cycle M contains an integral multiple of Pi. This argument holds for each task in the task set irrespective of its phase. Therefore, $M = LCM(\{P_1, P_2, \ldots P_n\})$.

Cyclic Schedulers

⇒ Cyclic schedulers are very popular and are being extensively used in the industry. All small embedded application being manufactured presently are based on cyclic schedulers. It is simple, efficient, and are easy to program.

⇒ Cyclic scheduler repeats a precomputed schedule. The precomputed schedule needs to be stored only for one major cycle. The major cycle is divided into one or more minor cycles. Each minor cycle is called as a frame.

⇒ The scheduling points for a cyclic scheduler are defined only at the frame boundary. This means a task can start its execution only at the begining of the frame.

⇒ The frame boundaries are defined through the interrupts generated by a periodic timer.

⇒ The assignment or allocation of task to its corresponding frame is stored in a scheduled table.

e.g →

Task no.	Frame no.
T_1	F_1
T_3	F_2
T_2	F_4
T_4	F_5
T_5	F_3

Q How to select a suitable frame size for a set of periodic real-time tasks?

Ans:— Selected frame size should satisfy the following three constraints.

1. Minimum Context Switching:

⇒ This constraints is imposed to minimize the number of context switching that occurs during task execution.

⇒ To avoid unnecessary context switching, the frame size should be chosen in such a manner that it will be greater than or equal to the execution time of all the tasks present in the task set.

So that, only one frame will be sufficient to complete the execution of the task.

⇒ This constraint is formulated as $\underline{\underline{\max\{e_i\} \leq F}}$

2. Minimization of Table Size:—

⇒ This constraint is imposed to minimize the number of entries in the schedule table, so that the storage requirement for the schedule table will be minimum.

⇒ This can be achived when minor cycle squarely divides the major cycle.

⇒ This constraint can be formulated as

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

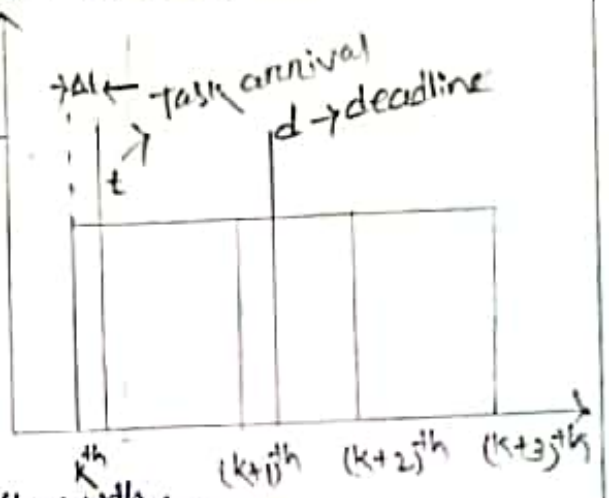
where M = major cycle

f = frame or minor cycle.

3. Satisfaction of Task Deadline:-

⇒ This constraint is imposed, so that each and every task present in the task ~~meet~~ set can meet their respective deadline.

⇒ A task can only be taken up for scheduling at the start of a frame. Consider this figure the task 't' arrived after the kth frame has started. So it can not be taken up for scheduling in the k^{th} frame, only can be taken up in the $(k+1)^{th}$ frame.



⇒ This constraint imposes that a task can meet its deadline, if there exists at least one complete frame between the arrival time of the task and its deadline.

⇒ Suppose a task T_i arrived after Δt time units from the begining of the last frame; assuming that only one frame is sufficient to complete the execution of a task, task T_i can meet its deadline if

$$2F - \Delta t \leq d$$

where $d \rightarrow$ deadline for T_i

$F \rightarrow$ Frame size

$\Delta t \rightarrow$ Time after T_i arrived

$$\Delta t = GCD(F, P_i)$$

where P_i = period of T_i

$$\therefore 2F - GCD(F, P_i) \leq d$$

* Theorem:- The minimum separation of the task arrive from the corresponding frame start time considering all instances of a task T_i is equal to $GCD(F, P_i)$.

Q// T_1 T_2 T_3 T_4

$e_1 = 1$ $e_2 = 1$ $e_3 = 1$ $e_4 = 2$

$P_1 = 4$ $P_2 = 5$ $P_3 = 20$ $P_4 = 20$

Select a suitable frame size for periodic scheduler.

Ans:—

constraint 1:

$\text{Max}\{e_i\} \leq F$

$\Rightarrow \text{Max}\{1,1,1,2\} \leq F$

$\Rightarrow 2 \leq F$

$\Rightarrow \boxed{F \geq 2}$

constraint 2:

Major cycle $M = \text{LCM}(4,5,20,20)$

$= 20$

$\boxed{M \bmod F = 0}$

So possible value for F are 2, 4, 5, 10, 20

constraint 3:

$\boxed{2F - \text{GCD}(F, P_i) \leq d_i}$

Frame size is 2

For task T_1: $2 \times 2 - \text{GCD}(2,4) \leq 4 \rightarrow$ satisfied

For task T_2: $2 \times 2 - \text{GCD}(2,5) \leq 5 \rightarrow$ satisfied

For task T_3: $2 \times 2 - \text{GCD}(2,20) \leq 20 \rightarrow$ satisfied

For task T_4: $2 \times 2 - \text{GCD}(2,20) \leq 20 \rightarrow$ satisfied

Frame size is 4:

For task T_1 : $2X4 - GCD(4,4) \leq 4 \rightarrow$ satisfied

For task T_2 : $2X4 - GCD(4,5) \leq 5 \rightarrow$ not satisfied

Frame size is 5:

For task T_1 : $2X5 - GCD(5,4) \leq 4 \rightarrow$ not satisfied

Frame size is 10:

For task T_1 : $2X10 - GCD(10,4) \leq 4 \rightarrow$ not satisfied

Frame size is 20:

For task T_1 : $2X20 - GCD(20,4) \leq 4 \rightarrow$ not satisfied

∴ The suitable frame size is 2.

A Generalized task scheduler :-

→ we have already stated that cyclic schedulers are overwhelming 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 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 ape-riodic and sporadic tasks can be accommodated by cyclic schedulers.

→ Recall that the arrival times of aperiodic and sporadic tasks are expressed statisitically. Therefore, there is an way to assign aperiodic and sporadic tasks to frames without significantly lowering the overall achievable utilization of the system.

→ In a generalized scheduler, initially 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 its Periodic task allocted to the fram completes its execution. Non-zero slck 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 ifonly if enough slack time is available for the arriving sporadic task to complete befor 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.

→ Though for aperiodic tasks no acceptance task 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 a 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, the 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 at an acceptance test and only those which have passed the test are available for scheduling.

→ The cyclic scheduler routine cyclic-scheduler() is activated at the end of every frame by a periodic timer. If the current tasks 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.

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 milliseconds.

→ Therefore, a call to a timer is made every few milliseconds. 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 ~~the~~ a task in the task set.

SOME ISSUES ASSOCIATED WITH RMA

In this Section, we address some miscellaneous issues associated with RMA scheduling of Tasks.

① Advantages and Disadvantages of RMA :-

The important advantages of RMA over EDA :-

→ RMA is simple and efficient and also the optimal Static priority task scheduling algorithm.

→ It is easy to implement.

→ If any static priority assignment algorithm can meet the deadlines then rate Monotonic scheduling can also do the same. It is optimal.

→ It consists of calculated copy of the time periods unlike other time-sharing algorithms as Round robin which neglects the scheduling needs of the processes.

Disadvantages :-

→ It is very difficult to support aperiodic and Sporadic tasks under RMA

→ RMA is not optimal when tasks period and deadline differ.

RMA Transient Overload Handling :-

→ RMA processes good transient overload handling capability. Good Transient overload handling capability essentially means that when a lower priority task does not complete within its planned completion time, it can not make any higher priority task to miss its deadline.

(2) Deadline Monotonic Algorithm (DMA) :-

→ RMA no longer remains an optimal scheduling algorithm for periodic realtime task, 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 then 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 to tasks with shorter deadlines.

→ when the relative deadline of every task is proportional to period, RMA and DMA produce identical solutions.

When the relative deadlines are arbitary, 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 away fails when DMA fails.

Example :-

Is the following task set schedulable by DMA? Check whether it is schedulable using RMA.

$T_1 = (e_1 = 10\ msec, P_1 = 50\ msec, d_1 = 35\ msec)$,

$T_2 = (e_2 = 15\ msec, P_2 = 10\ msec, d_2 = 20\ msec)$,

$T_3 = (e_3 = 20\ msec, P_3 = 200\ msec, d_3 = 200\ msec)$,

Solution.

Let us check RMA schedulability of the given task, by checking the Lehoczky's criterion.

The tasks are already ordered in decending order of their priorities.

Checking for T_1: 10 m sec < 35 m sec. Therefore, T_1 would meet it's first deadline.

Checking for T_2: $10 + 15 \neq 20$, Therefore, T_2 will miss its first deadline. Hence the given task set can't be feasibly scheduled under RMA.

Now, Check the schedulability using DMA:

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

Checking For T_2: 15 m sec < 20 m sec. Hence, T_2 will meet. it's first dead line.

Checking For T_1: (10 + 15) m sec $\leq$ 35 m sec, Hence T_1 will meet its first deadline.

Checking For T_3: (20 + 40 + 30) m sec < 200 m sec.

Therefore, T_3 will meet its dead line.

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

(3) Context Switching Overhead :-

→ It is easy to realize that under RMA. Whenever a task arrives, at most it preempts one task the task that 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 runs possibly pre-empting the currently running task, and the other when it completes.

A task may incur just one context switching overhead, if it does not preempt any task.

→ For simplicity we can assume that context switching time constant, and equals c milliseconds where c is a constant.

From this, it follows that the net effect of context switches to increases the execution time e_i of each task T_i to at most $e_i + 2 * C$.

There fore, 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:-

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 msec and is to be taken into account in schedulability computations.

Solu^n

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: $22 < 100$. Satisfied, T_1 meets it's first deadline

Checking for task T_2: $22 * 2 + 32 < 150$. satisfied, so T_2 meets it's first deadline.

Checking for task T_3: $22 * 2 + 90 < 200$. 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.

(4) Self Suspension :-

→ A task might causes its self suspension, when it perform its Input/output operations or when it waits for some events to occur.

When a task self suspends it self, 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 did not consider in the earlier section.

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

Let us determine the effect of self suspension on the schedulability of a task set. Let us Consider a set of periodic real time task $\{T_1, T_2, \cdots T_n\}$ which have been arranged in the increasing order of their priorities. Let the worst case self-suspension time task T_i is b_i. Let delay that the task T_i might incur due to its own self-suspension and self suspension of all higher priority task can be b_{ti} can be expressed as :

$$b_{ti} = b_i + \sum_{K=1}^{i-1} \min(e_K, b_K)$$

The RMA task schedulability condition of Liu and Layland needs to change when we consider the effect of self-suspension of tasks. To consider the effect of self suspension. we need to substitute e_i by $(e_i + b_{ti})$. If we consider the effect of self-suspension on task completion time, the lehoczky criterion would also have to be generalized.

$$e_i + b_{ti} + \sum_{K=1}^{i-1} \lceil P_i / P_K \rceil * (e_K \leq P_i) .$$

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

Example-

Consider the following set of periodic realtime tasks $T_1 = (e_1 = 10 \text{ msec}, P_1 = 50 \text{ msec})$, $T_2 = (e_2 = 25 \text{ msec}, P_2 = 150 \text{ msec})$ $T_3 = (e_3 = 50 \text{ msec}, P_3 = 200 \text{ msec})$. Assume that the self suspension times of T_1, T_2 and T_3 are 3 msec, 3 msec, and 5 msec, respectively. Determine whether the task 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 Lehoczkey's condition.

For T_1 to be schedulable: $(10+3)$ m sec < 50 m sec. So, T_1 meet its first deadline.

For T_2 to be schedulable : $(25+6+10*3)$ m sec < 15 m sec.

Therefore, T_2 meets its first deadline.

For T_3 to be schedulable : $(50+11+(10*4+25*2))$ msec ≤ 200 msec.

This inequality is also satisfied. Therefore, T_3 would also meed its first deadline.

(5) Self Suspension with Context Switching Overhead :-

→ In a fixed priority preemptable system, each task preempts at most one other task if there is no 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 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-suspends twice, additional four context switching over heads are incurred.

Let us denote the maximum context switch time as C. The effect of a single self-suspension of task is to effectively increases 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 task can be taken care of by replacing the execution time of a task T_i by $(e_i + 4*C)$.