Multiprocessor priority driven system is either dynamic or static. → previous unit.

Unit 6

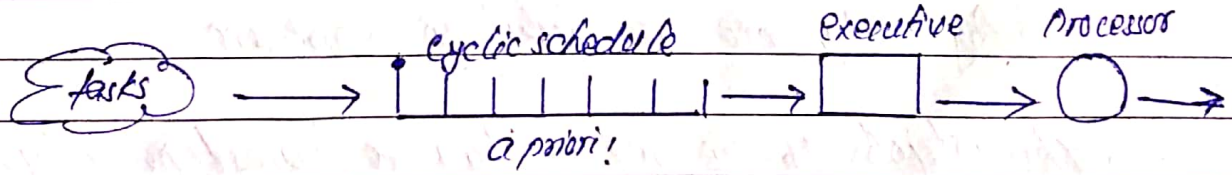
Priority Driven scheduling of Periodic Tasks

static Assumptions

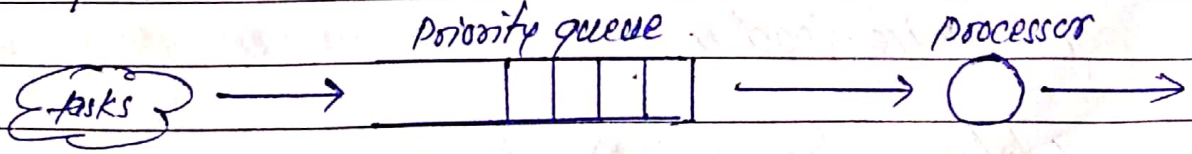
1. A fixed number of independent periodic tasks exist.
2. Jobs are ready for execution as soon as they are released.
3. Jobs can be preempted at any time.
4. Jobs never suspend themselves.
5. New task only admitted after an acceptance test.
6. The period of task is the minimum inter release time of jobs.
7. There are no aperiodic or sporadic task.
8. scheduling decisions are made immediately upon the job release and complete.
9. Algorithms are event driven, not clock driven.
10. The context switched overload are negligibly small.
11. Schedulable utilization is always less or equal to 1.

Priority driven vs. clock driven scheduling:

clock-driven:



Priority-driven:



Example showing poor performance of priority driven systems:

- Consider $m+1$ independent periodic task.
- For first m tasks (1 to m) T_i:
  - Periods = 1
  - Execution times = 2ϵ where ϵ is a small number

  $T_i = (1, 2\epsilon)$ for $i = 1, 2, \ldots, m$
- For $m+1^{th}$ task
  - Period = $1+\epsilon$
  - Execution time = 1

  $T_{m+1} = (1+\epsilon, 1)$
- Here $D_i = P_i$ and phase = 0.
- Priorities are assigned in EDF basis.
- Jobs are dispatched and scheduled dynamically on m processor.

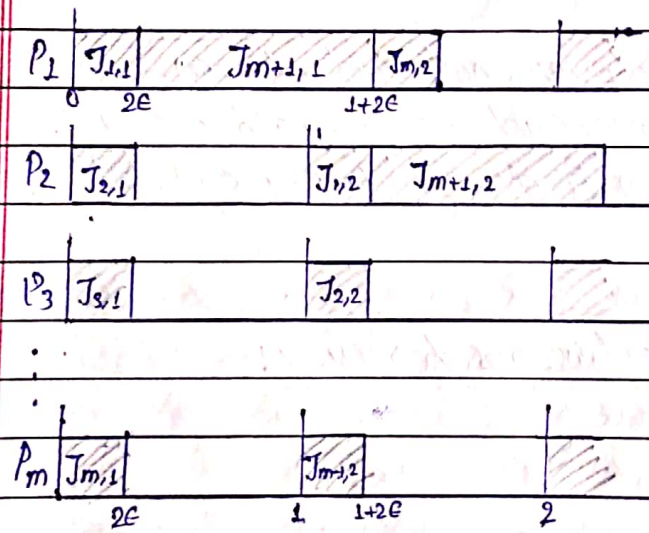


Fig: A dynamic EDF schedule on m processor

→ The 1st job $J_{m+1,1}$ in T_{m+1} has the lowest priority because it has the latest deadline.

→ Clearly $J_{m+1,1}$ does not complete until $1+2\epsilon$ and hence misses its deadline.

$$\text{Total utilization} = m\left(\frac{2\epsilon}{1}\right) + \frac{1}{(1+\epsilon)} = 2m\epsilon + \frac{1}{(1+\epsilon)} = \frac{2m\epsilon + 1}{(1+\epsilon)}$$

In the limit as ϵ approaches zero, U approaches 1, and yet the system remains unschedulable.
We would get the same infeasible schedule if we assigned the same priority to all jobs in each task according to the period of the task; the shorter the period, the higher the priority.
→ On the other hand, this system can be feasibly scheduled statically.

As long as the total utilization of the first n task, one is equal to or less than 1, this system can be feasibly schedule d on two processor if we put T_{n+1} on one processor and the other tasks on other processer and schedule ~~task~~ the task(s) on each processor according to either of these priority-driven algorithms.

Fixed-Priority versus Dynamic-Priority Algorithms

A priority driven scheduling is an online scheduler in which the priority of each periodic task is fixed relative to other task. This scheduler doesn't pre-computed a schedule of the tasks, but assigns priority to the jobs when they are released and places them on a run queue in priority order. When preemption is allowed, a scheduling decision is made whenever a job is released or completed. At each scheduling decision time, the scheduler updates the run queues and executes the jobs at the head of the queue.

The priority driven scheduler are classified into two types based on how the priorities are assigned to the jobs. They are: fixed priority and dynamic priority.

→ A **fixed-priority algorithm** assigns the same priority to all the jobs in each task. In other words, the priority of each

periodic task is fixed relative to other tasks.

→ A dynamic-priority algorithm assigns different priorities to the individual jobs in each task. Hence the priority of the task with respect to that of the other task changes as jobs are released and completed. This is why this type of algorithm is said to be "dynamic".

Fixed-Priority algorithms:
- Rate Monotonic (RM), Deadline Monotonic (DM)

Dynamic-priority algorithms:
- EDF, FIFO, LIFO, LST, RR

Rate-Monotonic (RM) Algorithm

→ This algorithm is a static priority based algorithm.
→ This algorithm assigns priorities to tasks based on their periods; the shorter the period, the higher the priority.
→ The rate (of job releases) of task is the inverse of its period i.e. Rate = 1/Period. Therefore, the jobs with higher rate have higher priority.

There are two important criteria to determine whether the given task set is RMA schedulable.

(i) Necessary condition:

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

$\frac{e_i}{P_i}$ = Processor utilization

(ii) Sufficient condition:

A given task set is RMA schedulable if

$$\sum_{i=1}^{n} \frac{e_i}{P_i} \leq n(2^{1/n} - 1)$$ n: total no. of task sets

It is called LIU-LAYLAND test.

Note:

Even if a task set fails the LIU-LAYLAND test, still then it is possible that the given task is RMA schedulable. A test that can be performed to check whether a task set is RMA schedulable when it fails the LIU-LAYLAND test is the LEHOCZKY'S test.

$$e_i + \sum_{k=1}^{i-1} \left\lceil \frac{P_i}{P_k} \right\rceil \times e_k \leq P_i \quad \text{if } P_i = d_i$$

$$e_i + \sum_{k=1}^{i-1} \left\lceil \frac{d_i}{P_k} \right\rceil \times e_k \leq d_i \quad \text{if } P_i < d_i$$

Q. Determine if the given task set is RMA schedulable.

$T_1 (e_1 = 20, P_1 = 100)$, $T_2 (e_2 = 30, P_2 = 150)$, $T_3 (e_3 = 60, P_3 = 200)$

Soln

Necessary condition :- $\sum_{i=1}^{3} \frac{e_i}{P_i} \leq 1$

$$\frac{20}{100} + \frac{30}{150} + \frac{60}{200} = \frac{7}{10} \leq 1 \text{ (True)}$$

Sufficient condition : $\sum_{i=1}^{3} \frac{e_i}{P_i} \leq 3(2^{1/3} - 1)$

$$\sum_{i=1}^{3} \frac{e_i}{P_i} = \frac{7}{10} = 0.7$$

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

$0.7 \leq 0.78$ (True)

∴ It is RMA schedulable.

(11) $T_1(20, 10)$, $T_2(60, 15)$, $T_3 = (120, 20)$ — P_i, e_i

Necessary:

$$\sum_{i=1}^{3} \frac{e_i}{P_i} = \frac{10}{20} + \frac{15}{60} + \frac{20}{120} = 0.9 \leq 1 \text{ (True)}$$

Sufficient:

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

$0.9 \leq 0.78$ (false)

The given task set fails the LIU-LAYLAND test. {Here this condition is false, so again we cannot say that this condition can be schedulable or not.

LEHOCZKY test — ← Moving to next procedure.

$T_1 (i=1)$

$$e_1 + \sum_{k=1}^{i-1} \left\lceil \frac{P_1}{P_k} \right\rceil \times e_k \leq P_1 \qquad i-1 = 0$$

$\Rightarrow e_1 \leq P_1$

$\Rightarrow 10 \leq 20$ (True)

$T_2 (i=2)$

$$e_2 + \sum_{k=1}^{1} \left\lceil \frac{P_2}{P_k} \right\rceil \times e_k \leq P_2$$

$$\Rightarrow e_2 + \left\lceil \frac{P_2}{P_1} \right\rceil \times e_1 \leq P_2$$

$$\Rightarrow 15 + \frac{60}{20} \times 10 \leq 60$$

$\Rightarrow 45 \leq 60$ (True)

$T_3 (i=3)$

$$e_3 + \sum_{k=1}^{2} \left\lceil \frac{P_3}{P_k} \right\rceil \times e_k \leq P_3$$

$$\Rightarrow e_3 + \left\lceil \frac{P_3}{P_1} \right\rceil * e_1 + \left\lceil \frac{P_3}{P_2} \right\rceil \times e_2 \leq P_3$$

$$\Rightarrow 20 + \left\lceil \frac{120}{20} \right\rceil \times 10 + \left\lceil \frac{120}{60} \right\rceil \times 15 \leq 120$$

$\Rightarrow 110 \leq 120$ (True)

Here all task can complete within deadline. So,

∴ RMA is schedulable.

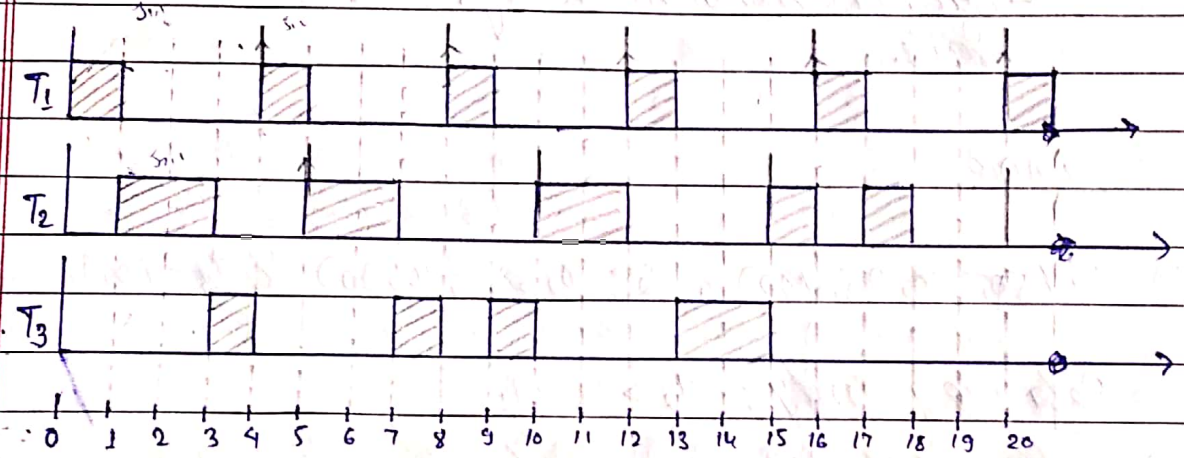
Example:

① $T_1(4,1)$; $T_2(5,2)$; $T_3(20,5)$

Rate of task $T_1 = 1/4$
Rate of task $T_2 = 1/5$
Rate of task $T_3 = 1/20$

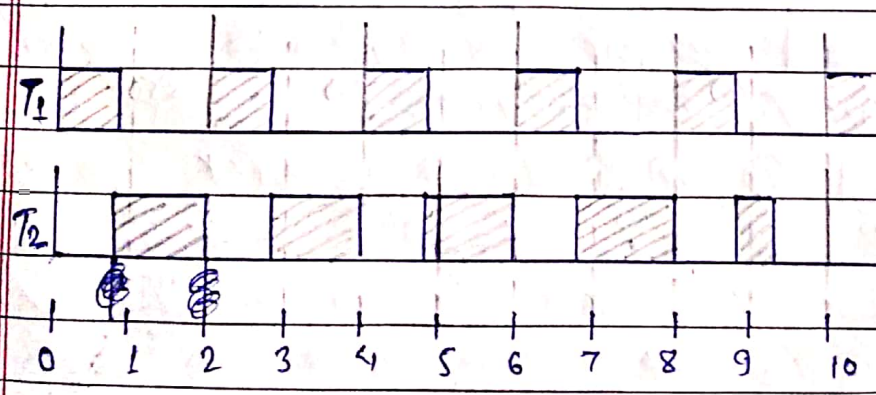
so the priority be assign as $T_1 > T_2 > T_3$

The schedule is:



② $T_1(2, 0.9)$; $T_2 = (5, 2.3)$

Rate of $T_1 = 1/2$
Rate of $T_2 = 1/5$
$T_1 > T_2$



DM preferred to RM for deadline ≠ period.

Deadline-Monotonic Algorithm

→ This algorithm assigns priorities to tasks according to the relative deadlines: the shorter the relative deadline, the higher the priority.

→ When relative deadline of every task matches its period, then rate monotonic and deadline monotonic give identical results.

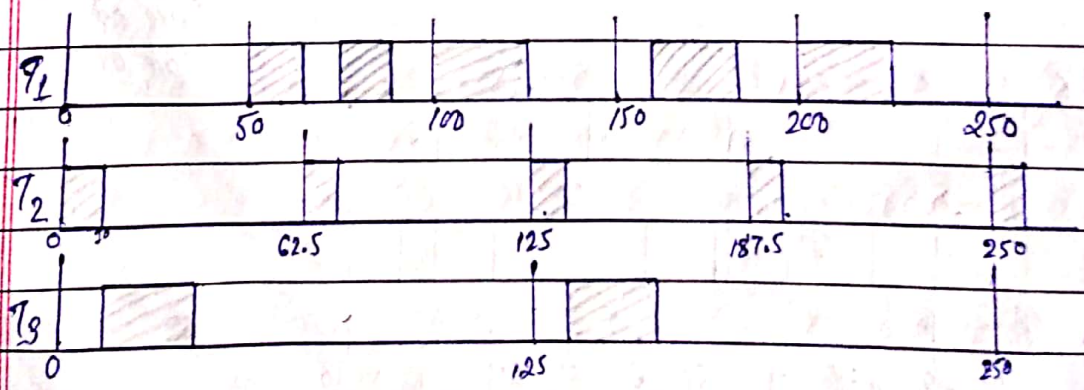
→ When the relative deadlines are arbitary:
- deadline monotonic can sometimes produce a feasible schedule in cases where rate monotonic cannot.
- But, rate monotonic always fails when deadline monotonic fails.

Example

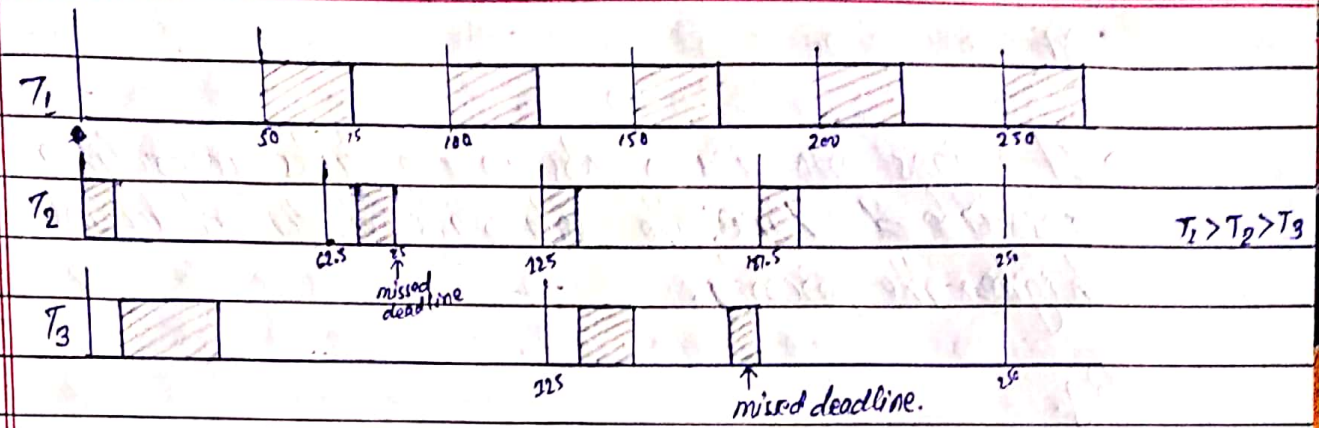
(1) $T_1 = (50, 50, 25, 100)$, $T_2 = (0, 62.5, 10, 20)$ & $T_3 = (0, 125, 25, 50)$

Relative priority: $T_2 > T_3 > T_1$

DM schedule:



(a) DM schedule: feasible

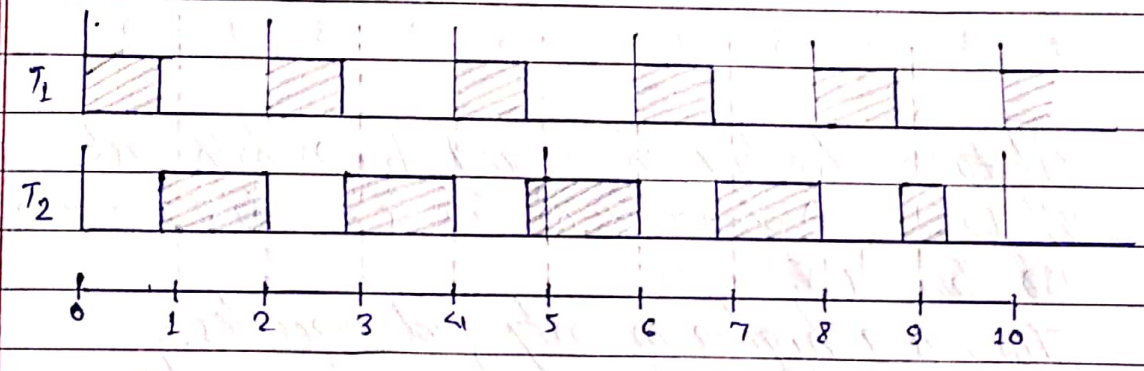


(b) RM schedule : Not feasible

(c) T_1 (2, 0.9) T_2 (5, 2.3)

T_1 (0, 2, 0.9, 2) T_2 (0, 5, 2.3, 5) HP = 10

Relative priority : $T_1 > T_2$



Dynamic Priority Algorithms

Dynamic priority algorithms are:

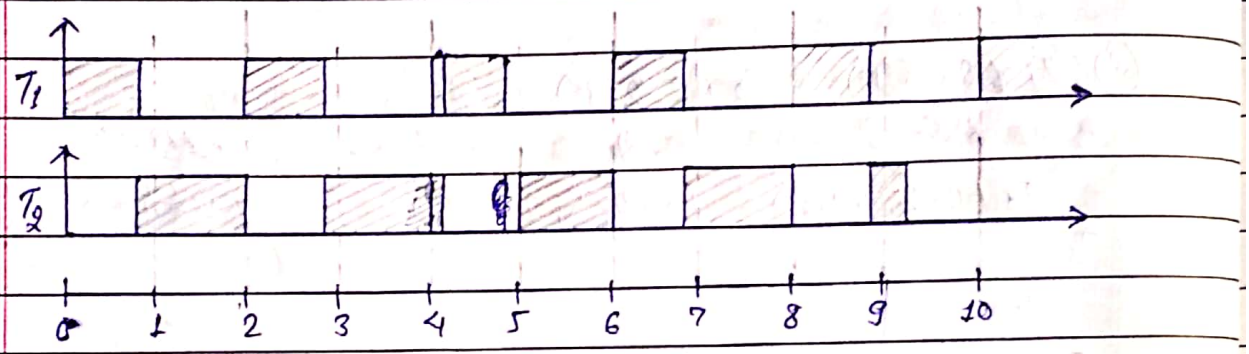
1. Earliest deadline first (EDF)
2. Least slack time (LST)
3. First In First out (FIFO)
4. Last In First out (LIFO)

EDF

→ It assigns priorities to jobs in the tasks according to their absolute deadline: the earlier its absolute deadline, the higher the priority.

e.g.

T_1 (2, 0.9) & T_2 (5, 2.3)



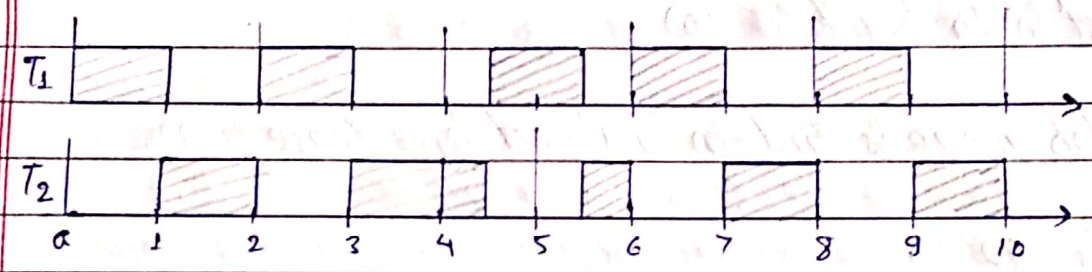
- At time 0, the first job $J_{1,1}$ and $J_{2,1}$ of both tasks are ready.
- The (absolute) deadline of $J_{1,1}$ is 2 while the deadline of $J_{2,1}$ is 5.
- $J_{1,1}$ has a higher priority and executes.
- When $J_{1,1}$ completes $J_{2,1}$ begins to execute.
- At time 2, $J_{1,2}$ is released, and its deadline is 4, earlier than the deadline of $J_{2,1}$.
- Hence, $J_{1,2}$ is placed ahead of $J_{2,1}$ in the ready job queue.
- $J_{1,2}$ preempts $J_{2,1}$ and executes.
- At time 2.9, $J_{1,2}$ completes. The processor then executes $J_{2,1}$.
- At time 4, $J_{1,3}$ is released; its deadline is 6, which is later than the deadline of $J_{2,1}$.
- Hence the processor continues to execute $J_{2,1}$.
- At time 4.1, $J_{2,1}$ completes, the processor starts to execute $J_{1,3}$ and so on.

- It is noted that the priority of T_1 is higher than the priority of T_2 from time 0 until time 4.0.

Least slack Time (LST)

E.g. $T_1(2, 1)$, $T_2(5, 2.5)$

$H = LCM(2, 5) = 10$



Calculating slack time

At $t=0$, slack time for
$T_1 = 2-0-1 = 1$
$T_2 = 5-0-2.5 = 2.5$

At $t=1$,
only T_2 is ready.

At $t=2$,
$T_1 = 4-2-1 = 1$
$T_2 = 5-2-1.5 = 1.5$

At $t=3$
only T_2 is ready

At $t=4.5$
only T_1 is ready.

At $t=4$
$T_1 = 6-4-1 = 1$
$T_2 = 5-4-0.5 = 0.5$

At $t=5$
$T_1 = 6-5-0.5 = 0.5$
$T_2 = 10-5-2.5 = 2.5$

At $t=5.5$
only T_2 is ready.

At $t=6$
$T_1 = 8-6-1 = 1$
$T_2 = 10-6-2 = 2$

At $t=7$
only T_2 is ready.

At $t=8$
$T_1 = 10-8-1 = 1$
$T_2 = 10-8-1 = 1$

At $t=9$
only T_2 is ready

Two variations of LST:

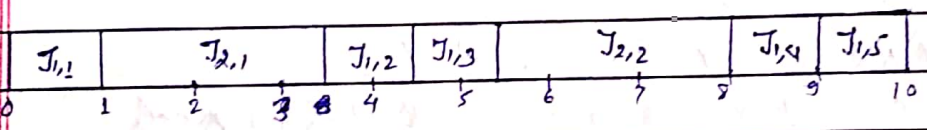
1) strict LST: scheduling decisions are made also whenever a queued job's slack time becomes smaller than the executing job's slack time - huge overhead, not used.
2) Non-strict LST: scheduling decisions made only when job releases or complete.

First in first out (FIFO)

- Job queue is first-in-first-out by release time.

$T_1 = (2, 1)$

$T_2 = (5, 2.5)$



Last in First out (LIFO)

- Job queue is last-in-first-out by release time.

FIFO e.g.

$T_1 = (0, 3, 1, 3)$, $T_2 = (0.5, 4, 1, 1)$ & $T_3 = (0.75, 7.5, 2, 7.5)$

→ $J_{1,1}$ has a higher priority than $J_{2,1}$ and $J_{2,1}$ has higher priority than $J_{3,1}$.
In other words T_1 has the highest priority & T_3 has the lowest priority initially.

Later $J_{1,4}$, $J_{2,3}$ & $J_{3,2}$ are released at the times 9, 8.5 & 8.25
T_3 has the highest priority while T_1 has the lowest priority.

Maximum schedule utilization

- A system is schedulable by an algorithm if the algorithm always produces a feasible schedule of the system.
- A system is schedulable (and feasible) if it is schedulable by some algorithm. that is, feasible schedule of the system exist.
- A scheduling algorithm can feasibly schedule any set of periodic task on a processor if and only if the total utilization of the tasks is equal to or less than the schedulable utilization of the algorithm (U_{ALG}). If $U_{ALG} = 1$, the algo. is optimal.

Schedulable utilization of EDF/LST Algorithm

Maximum schedulable utilization Theorem:

Theorem 1:

A system of independent, preemptable periodic tasks with relative deadlines equal to or greater than their respective periods (i.e. $D_i \geq P_i$) can be feasibly scheduled on one processor if and only if its total utilization is equal to or less than 1.

→ When the relative deadlines of some tasks are less than their period, the system may not be feasible, even when its total utilization is less than 1. For these conditions, we have another test (Theorem 2).

e.g. For task $T_1 = (2, 0.9)$, $T_2 = (5, 2.3)$ is feasible but it would not be schedulable if its relative deadlines were 3 instead of 5.

Theorem 2: (with $D_i < P_i$)

A system of independent, preemptable tasks can be feasibly scheduled on one processor if and only if its density is equal to or less than 1.

→ The ratio of the execution time e_i of a task T_i to the minimum of its relative deadline D_i, and period P_i is called the density δ_i of the task.

i.e. density of the task T_k is $\delta_k = \frac{e_k}{\min(D_k, P_k)}$.

→ The sum of the densities of all periodic task in a system is called the density of the system and is denoted by Δ.

i.e. $\Delta = \delta_1 + \delta_2 + \cdots + \delta_n$.

E.g.

$T_1 = (2, 0.9)$; $T_2 = (5, 2.3, 3)$ then

$\Delta = \frac{0.9}{2} + \frac{2.3}{3} = \frac{7.3}{6} > 1$, not feasible.

→ Neither RM nor DM is optimal.

Optimality of the RM and DM Algorithms

RM and DM algorithms assigns fixed priorities to the tasks. so they cannot be optimal. They may fail to schedule some system, even though feasible schedule exists.

For E.g.

Consider two tasks $T_1 = (2, 1)$ & $T_2 = (5, 2.5)$

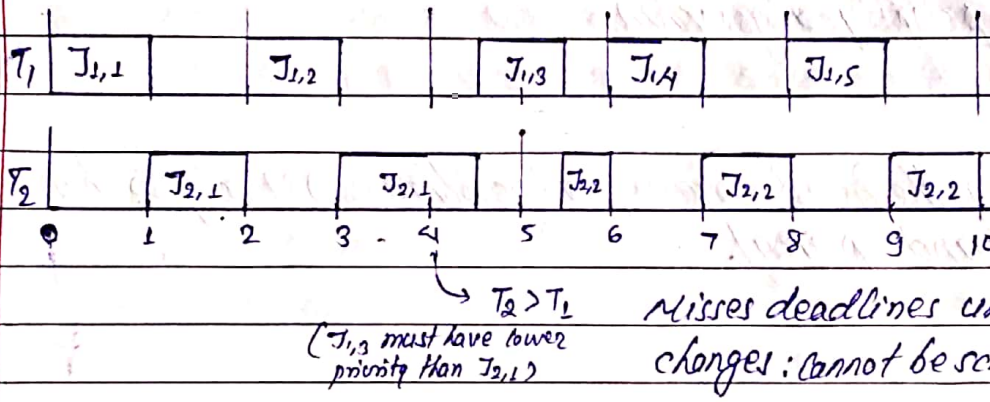
Total utilization $(U) = \frac{1}{2} + \frac{2.5}{5} = 1$

Since $U = 1$, the task are feasible.

- $J_{1,1}$ & $J_{1,2}$ can complete in time only if they have a higher

priority than $J_{2,1}$. In other words, in the time interval 0 to 4, T_1 must have higher priority than T_2.

- However at time 4 when $J_{1,3}$ is released, $J_{2,1}$ can complete in time only if T_2 (i.e. $J_{2,1}$) has higher priority than T_1 (i.e. $J_{1,3}$). This change in the relative priorities of the tasks is not allowed by any fixed priority algorithm.



→ Any fixed priority algorithm can be optimal in restricted system. RM (DM) algorithm is not optimal for tasks with arbitary periods. But it is optimal in a special case when the periodic tasks in the system or simply periodic and deadlines of the tasks are not less than their respective periods.

→ A system of periodic tasks is said to be simply periodic if the period of each is an integer multiple of the periods of the other tasks.

i.e $P_K = n * P_i$, where $P_i < P_K$ & n is a positive integer for all T_i & T_K.

Theorem for optimality of RM algorithm:

A system of simply periodic, independent, preemptable tasks whose relative deadlines are equal to or larger than their period (i.e. $D_i >= P_i$) is schedulable on one processor according to the RM algorithm if and only if its total utilization is equal to or less than 1.

Theorem for optimality of RM algorithm:

A system T of independent, preemptable periodic tasks that are in phase and have relative deadlines equal to or less than their respective periods can be feasibly scheduled on one processor according to the RM algorithm whenever it can be feasibly scheduled according to any fixed-priority algorithm.

→ Among the fixed priority algorithms, RM algorithm is the best and optimal.

Scheduling Test for Fixed priority Tasks with short Response Times - Critical Instants

→ The response time of the task is the difference between its release time and completion time.

→ The response times of the jobs should be smaller than or equal to their respective periods and the total utilization should be less than or equal to 1. It means, every job completes execution before the next job if the same task is released.

→ Schedulability test of fixed priority tasks with short response time includes:

1. Find the critical instant when the system is most loaded and has its worst response time.
2. Use time demand analysis to determine whether the system is schedulable at that instant.
3. If a fixed priority system is schedulable at the critical instant then it is always schedulable.

Critical Instants

A critical instant for a job is a worst case release time for that job, taking into account all jobs that have higher priority. i.e. a job released at the same instant as all jobs that have higher priority are released and must wait for all those to completed before it executes.

→ The response time of a job in T_i released at a critical instant is called the maximum (possible) response time and is denoted by W_i.

Critical instant of a task T_i is a time instant such that:

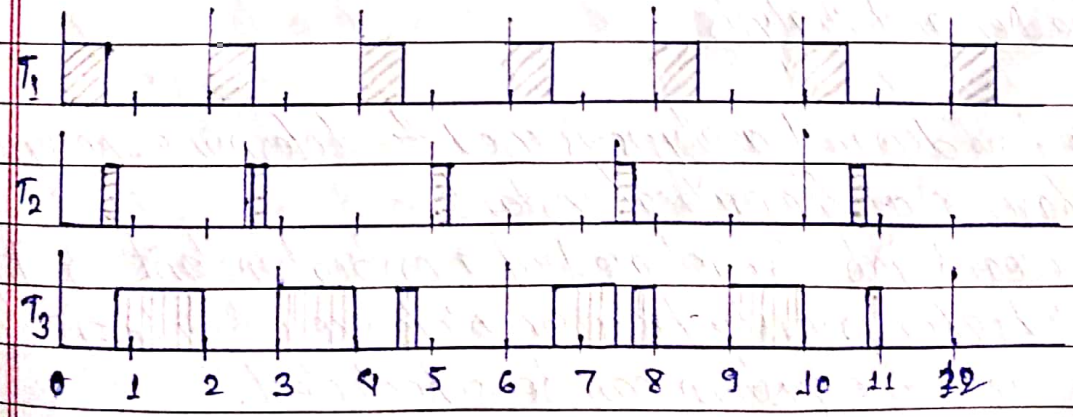
- The job of T_i released at this instant has maximum response time of all jobs in T_i, if the response time of every job of T_i is less than or equal to relative deadline (D_i) of T_i and
- The response time of the job released at this instant is greater than D_i if the response time of some jobs in T_i exceeds D_i.

E.g.

Consider three tasks $T_1 = (2, 0.6)$, $T_2 = (2.5, 0.2)$ & $T_3 = (3, 1.2)$

RM schedule of these 3 tasks:

$T_1 > T_2 > T_3$



Response time for all jobs of Task $T_1 = 0.6$

Response times of jobs in T_2 are:

$r_{2,1} = 0.8$, $r_{2,2} = 2.8 - 2.5 = 0.3$, $r_{2,3} = 5.2 - 5 = 0.2$, $r_{2,4} = 7.8 - 7.5 = 0.3$

$r_{2,5} = 10.8 - 10 = 0.8$ and so on.

∴ critical instants of T_2 are $t = 0$ and $t = 10$

Response times of jobs in T_3 are

$r_{3,1} = 2$, $r_{3,2} = 1.8$, $r_{3,3} = 2$, $r_{3,4} = 2$

∴ critical instants of T_3 are $t = 0$, $t = 6$ and $t = 9$.

Theorem:

In a fixed-priority system where every job completes before the next job in the same task is released, a critical instant of any task T_i occurs when one of its job $J_{i,c}$ is released at the same time with a job in every higher-priority task, that is, $r_{i,c} = r_{k,l_k}$ for some l_k for every $k = 1, 2, \ldots i-1$.

Time Demand Analysis

- The time demand analysis is used to determine the system behaviour at the critical instant.
- For each job $J_{i,c}$ released at a critical instant if $J_{i,c}$ and all higher priority tasks complete executing before their D_i, then the system can be scheduled.

Schedulability test

- Calculate the worst case response time R for each task with deadline D.
- If R<=D, the task is schedulable / feasible.
- Repeat the same checks for all tasks. If all tasks pass the test, the task set is schedulable.

Schedulability test for fixed priority tasks with arbitary response times - Busy Interval:

It is the schedulability test for tasks with relative deadlines greater than respective periods. Since the response time of a task may be larger than its period, it may have more than one job ready for execution at any time. Ready jobs in the same task are usually scheduled on the FIFO basis. This policy is used here named as busy interval.

Busy Interval

π_i denotes the priority of periodic task T_i.

- Busy interval represents the time interval at which the processor is busy all the time executing some periodic jobs.
→ level-π_i busy interval represents the interval in which jobs with equal or higher priority than π_i are executed.

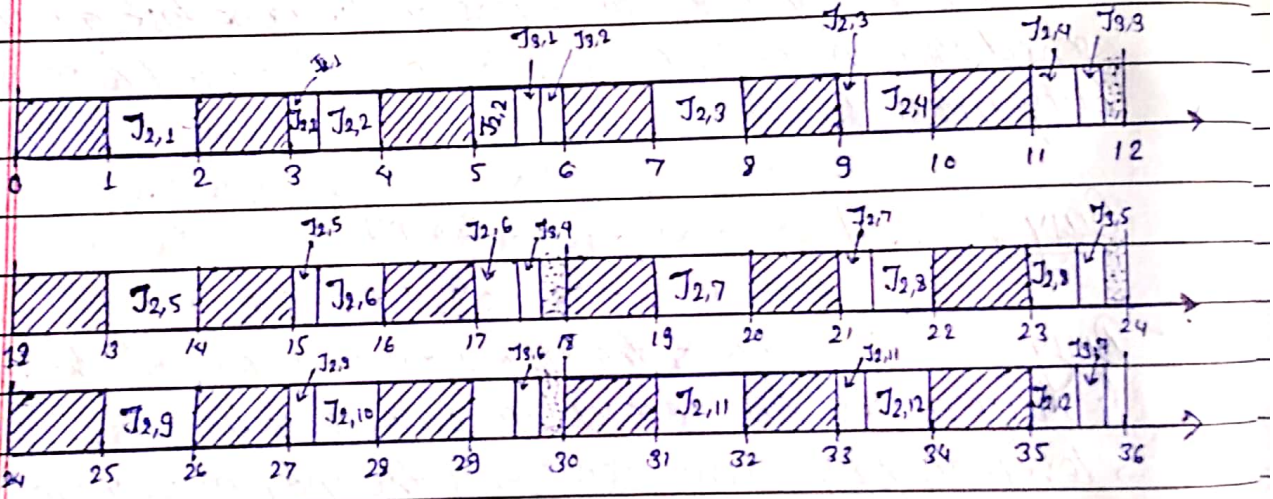
- A level-π_i busy interval is an interval $(t_0, t]$ begins at an instant t_0 when
1) all jobs in T_i released before the instant t_0 have completed execution and
2) A job in T_i is released

The interval ends at the first instant t after t_0 when all jobs in T_i released since t_0 are complete.

[In other words, in the interval $(t_0, t]$, the processor is busy all the times executing jobs with priorities π_i or higher, all the jobs executed in the busy interval are released in the interval, and at the end of the interval there is no backlog of jobs to be executed afterwards.

⇒ A level-π_i busy interval is in phase if the first job of all tasks that have priorities equal to or higher than priority π_i and are executed in this interval have the same release time.

For e.g.
Consider a system with three task $T_1 = (2, 1)$, $T_2 = (3, 1.25)$ & $T_3 = (5, 0.25)$ scheduled as



⇒ The shaded rectangles depict where jobs in T_1 are scheduled. The first busy intervals of all levels are in phase.

∴ The priorities of the tasks are $\pi_1 = 1$, $\pi_2 = 2$ and $\pi_3 = 3$, with 1 being the highest priority and 3 being the lowest priority.

- Every level-1 busy interval always ends 1 unit time after it begins.
- For this system, all the level-2 busy intervals are in phase. They begin at times 0, 6 and so on.

- The length of these intervals are equal to 5.5.
- Before time 5.5, there is at least one job of priority 1 or 2 ready for execution, but immediately after 5.5, there are none.
- Hence at 5.5, the first job in T_3 is scheduled. When the job completed at 5.75, the second job in T_3 is scheduled.
- At time 6, all the job released before time 6 are completed; hence, the first level-3 busy interval ends at this time.
- The second level-3 busy interval begins at time 6. This level-3 busy interval is not in phase since the release times of the first higher priority jobs in this interval are 6, but the first job of T_3 in this interval is not released until time 10.
- The length of this level-3 busy interval is only 5.75. Similarly, all the subsequent level-3 busy intervals in the hyperperiod have arbitary phase.

level-1 busy interval = 1
level-2 " " = 5 ($t=0$ to $t=5.5$)
first level-3 busy interval = 6
second level-3 " " = 5.75

Sufficient schedulability conditions for RM and DM algorithms:

Theorem:

A system of n independent, preemptable periodic task with relative deadline is equal to their respective periods can be feasibly schedule on a ~~one~~ processor according to RM algorithm if its total utilization is less than or equal to $U_{RM}(n) = n(2^{1/n} - 1)$

i.e. $\sum_{i=1}^{n} \frac{e_i}{p_i} \leq n(2^{1/n} - 1)$ n : no. of periodic task.

Practical factors

We have assumed that,
- Jobs are preemptable at any time.
- Jobs never suspend themselves.
- Each job has distinct priority.
- The scheduler is event driven and acts immediately.

These assumptions are often not valid so following practical factors must be considered for priority driven scheduling.

1. Blocking and priority inversion
2. soft-suspensions
3. Context switches
4. Tick scheduling.

Blocking and priority inversion

→ A ready job is blocked when it is prevented from executing by a lower-priority job.
→ A priority inversion is when a lower-priority job executes while a higher priority job is blocked.

These occurs because some jobs cannot be pre-empted:
- Many reasons why a job may have non-preemptable sections:
  - critical section over a resource
  - some system calls are non-preemptable
  - Disk scheduling

- If a job becomes non-preemptable, priority inversion may occur, these may cause a higher priority task to miss its deadline.

self-suspension

- A job may invoke an external operation (e.g. request an I/O operation), during which time it is suspended.

Context switches

- Assume maximum number of context switches K_i for a job T_i is known; each takes t_{cs} time units.
- Compensate by setting execution time of each job,

$$e_{actual} = e + 2t_{cs}$$

Time-Demand Analysis / General scheduling test

The process used to determine whether a task can meet its deadline is called time demand analysis.