

Chapter 5

Clock Driven Scheduling

Assumptions and notation for clock-driven scheduling

1. There is a constant number n periodic tasks in the system.
2. The parameters of all periodic tasks are known a priori.
 - ☐ Variations in inter-release times of jobs are negligibly small.
 - ☐ Each job in T_i is released p_i units of time after the previous job in T_i .
3. Each job $J_{i,k}$ is ready for execution at its release time $r_{i,k}$.
 - ☐ There are aperiodic jobs released at unexpected time instants.
 - ☐ aperiodic jobs are placed in special queue.
 - ☐ new jobs are added to the queue without need to notify scheduler.
 - ☐ when processor is available aperiodic jobs are scheduled.
 - ☐ There are no sporadic jobs (this assumption will be relaxed later).

Notations:

• The 4-tuple $T_i = (\phi_i, p_i, e_i, D_i)$ refers to a periodic task T_i with phase ϕ_i , period p_i , execution time e_i , and relative deadline D_i .

– Default phase of T_i is $\phi_i = 0$, default relative deadline is the period $D_i = p_i$.

– Omit elements of the tuple that have default values.

Example:

i) $T_1 = (1, 10, 3, 6) \Rightarrow \phi_1 = 1, p_1 = 10, e_1 = 3, D_1 = 6$

$J_{1,1}$ released at 1, deadline 7

$J_{1,2}$ released at 11, deadline 17

ii) $T_2 = (10, 3, 6) \Rightarrow \phi_2 = 0, p_2 = 10, e_2 = 3, D_2 = 6$

$J_{2,1}$ released at 0, deadline 6
 $J_{2,2}$ released at 10 and so on.... deadline 16

$$\text{iii) } T3 = (10, 3) \Rightarrow \phi_3 = 0, p_3 = 10, e_3 = 3, \\ D_3 = 10.$$

$J_{3,1}$ released at 0, deadline 10
 $J_{3,2}$ released at 10, deadline 20

Static, Clock-Driven Scheduler:

- Static schedule can be calculated off-line (all parameters are known at start).
 - Can use complex algorithms can be used. Run-time of the scheduling algorithm irrelevant.
 - Amount of processor time allocated to each job is equal to its maximum execution time.
 - Static schedule guarantees that every job completes by its deadline as long as no job overruns.
 - Scheduler dispatches jobs according to the static schedule, repeating each hyper period.

Example:

Four independent periodic tasks: $T1 = (4,1)$, $T2 = (5, 1.8)$, $T3 = (20, 1)$, $T4 = (20, 2)$

$$\square \text{ Utilization} = 1/4 + 1.8/5 + 1/20 + 2/20 = 0.76$$

$$\square \text{ Hyperperiod} = \text{LCM}(4, 5, 20, 20) = 20$$

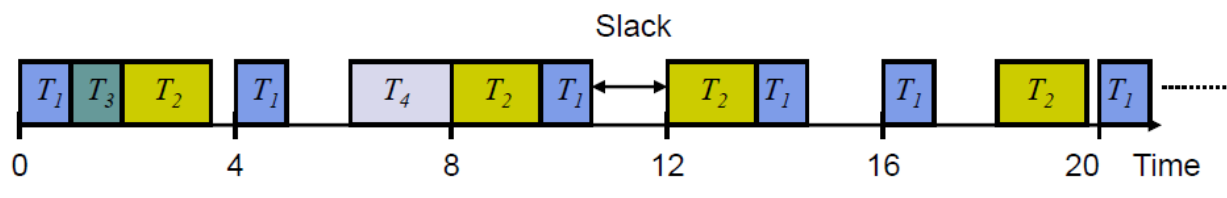


Fig . One possible schedule

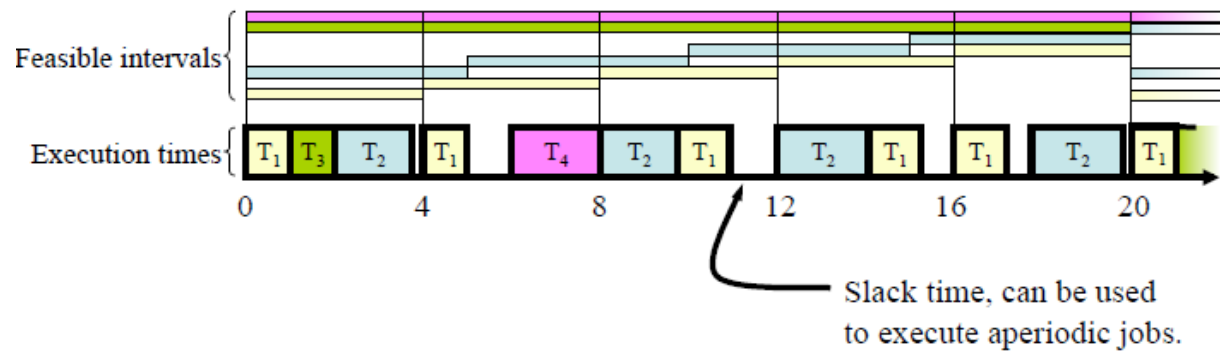
Slack time can be used to execute aperiodic jobs

Another example : Consider a system with 4 independent periodic tasks:

– $T1 = (4, 1.0)$, $T2 = (5, 1.8)$, $T3 = (20, 1.0)$, $T4 = (20, 2.0)$ [Phase and deadline take default values] here first value represents period and next represents execution time.

Hyper-period $H = 20$ (least common multiple of 4, 5, 20, 20)

- Can construct an arbitrary static schedule to meet all deadlines:



Types of Clock Driven Schedules:

- Table driven and
- Cyclic Schedules.

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. Rather than automatic computation of 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 the run time. An example of a schedule table is shown as following fig.

Tasks	Start time in millisecond
T1	0
T2	3
T3	10
T4	12
T5	17

Fig. An example of table driven scheduling

Think ::: What would be the size of schedule table required for a given set of periodic real time tasks to be run on a system?

Cyclic Schedule:

Cyclic schedules are very popular and extensively used in industry. Cyclic schedules are simple, efficient and are easy to program. An example application where cyclic schedule is 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.

Tasks	Frame Number
T3	F1
T1	F2
T3	F3
T4	F2

Fig. Example schedule table for cyclic scheduler

A cyclic scheduler repeats a pre-computed schedule. The pre-computed schedule needs to be stored only for one major cycle.

General structure of cyclic schedules:

Arbitrary table-driven cyclic schedules flexible, but inefficient

- Relies on accurate timer interrupts, based on execution times of tasks
- High scheduling overhead

Easier to implement if structure imposed:

- Make scheduling decisions at periodic intervals (*frames*) of length f
- Execute a fixed list of jobs with each frame, disallowing pre-emption except at frame boundaries
- Require phase of each periodic task to be a non-negative integer multiple of the frame size.
- The first job of every task is released at the beginning of a frame
- $\phi = k \cdot f$ where k is a non-negative integer

Gives two benefits:

- Scheduler can easily check for overruns and missed deadlines at the end of each frame.
- Can use a periodic clock interrupt, rather than programmable timer.

Frame Size Constraints:

How to choose frame length?

- To avoid preemption, want jobs to start and complete execution within a single frame:

$$f \geq \max(e1, e2, \dots, en) \dots\dots\dots (Eq.1)$$

- To minimize the number of entries in the cyclic schedule, the hyper-period should be an integer multiple of the frame size ($\Rightarrow f$ divides evenly into the period of at least one task):

$$\exists i : \text{mod}(pi, f) = 0 \dots\dots\dots (Eq.2)$$

- To allow scheduler to check that jobs complete by their deadline , should be at least one frame boundary between release time of a job and its deadline:

$$2*f - \text{gcd}(pi, f) \leq Di \text{ for } i = 1, 2, \dots, n \dots\dots\dots (Eq.3)$$

All 3 constraints should be satisfied.....!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Frame Size Constraints – Example 1:

Given tasks are T1 = (4, 1.0), T2 = (5, 1.8) T3 = (20, 1.0), T4 = (20, 2.0).

Hyper-period H = lcm (4, 5, 20, 20) = 20

Constraints: Eq.1 $\Rightarrow f \geq \max (1, 1.8, 1, 2) \geq 2$

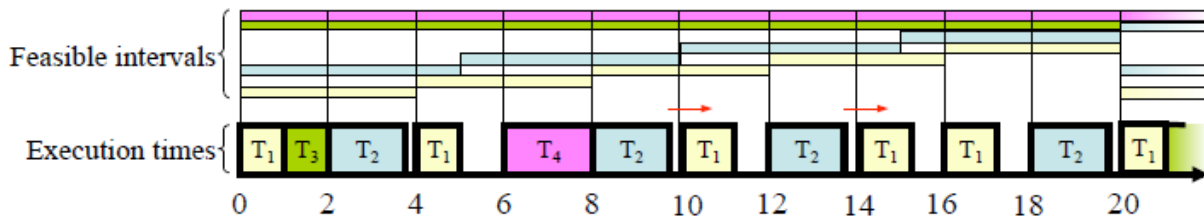
Eq.2 $\Rightarrow f \in \{ 2, 4, 5, 10, 20 \}$

$$\text{Eq.3} \Rightarrow 2f - \text{gcd}(4, f) \leq 4 \quad (T1)$$

$$2f - \text{gcd}(5, f) \leq 5 \quad (T2)$$

$$2f - \text{gcd}(20, f) \leq 20 \quad (T3, T4)$$

The values that satisfies all constraints are $f = 2$ or 4 .



Example 2:

Task set:

(P_i, e_i, D_i)

T1: (15, 1, 14)

T2: (20, 2, 26)

T3: (22, 3, 22)

- (1) $\forall i : f \geq e_i \Rightarrow f \geq 3$
- (2) $f | H \Rightarrow f = 2, 3, 4, 5, 6, 10, \dots$
- (3) $\forall i : 2f - \gcd(p_i, f) \leq D_i \Rightarrow f = 2, 3, 4, 5, 6$

\Rightarrow possible values for $f : 3, 4, 5, 6$

Job Slices:

- Sometimes, a system cannot meet all three frame size constraints simultaneously.
- Can often solve by partitioning a job with large execution time into slices (sub-jobs) with shorter execution times/deadlines. Consider a system with $T1 = (4, 1)$, $T2 = (5, 2, 7)$, $T3 = (20, 5)$
 - Cannot satisfy constraints: $\text{Eq.1} \Rightarrow f \geq 5$ but $\text{Eq.3} \Rightarrow f \leq 4$
 - Solve by splitting $T3$ into $T3,1 = (20, 1)$, $T3,2 = (20, 3)$ and $T3,3 = (20, 1)$
 - Other possible splits exist; pick based on application domain knowledge
 - Result can be scheduled with $f = 4$

A Cyclic Executive

- Modify previous table-driven cyclic scheduler to be frame base schedule all types of jobs in multi-threaded system.
- Table that drives the scheduler has F entries, where $F = H/f$
 - Each corresponding entry $L(k)$ lists the names of the job slices that are scheduled to execute in frame k ; called a scheduling block.
 - Each job slice implemented by a procedure, to be called in turn.
- Cyclic executive executed by the clock interrupt that signals the start of a frame:
 - Determines the appropriate scheduling block for this frame.
 - Executes the jobs in the scheduling block in order.
 - Starts job at head of the aperiodic job queue running for remainder of frame.
- Less overhead than pure table driven cyclic scheduler, since only interrupted on frame boundaries, rather than on each job.

Improving the average response time of Aperiodic Jobs:

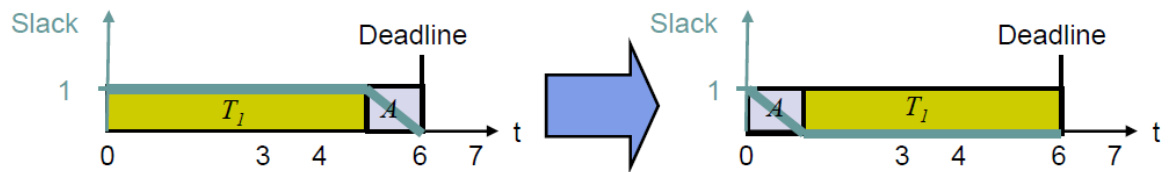
- So far aperiodic jobs have been scheduled in the background after all other job slices have been completed, it causes average response time is long.
- Average response time for aperiodic jobs can be improved by scheduling hard-real time jobs as late as possible without missing the deadline.

Slack Stealing:

A natural way to improve the response times of aperiodic jobs is by executing the aperiodic jobs ahead of the periodic jobs whenever possible. This approach, called *slack stealing*, was originally proposed for priority-driven systems. For the slack-stealing scheme to work, every periodic job slice must be scheduled in a frame that ends no later than its deadline. Let the total amount of time allocated to all the slices scheduled in the frame k be x_k . The *slack* (time) available in the frame is equal to $f - x_k$ at the beginning of the frame. If the aperiodic job queue is nonempty at this time, the cyclic executive can let aperiodic jobs execute for this amount of time without causing any job to miss its deadline.

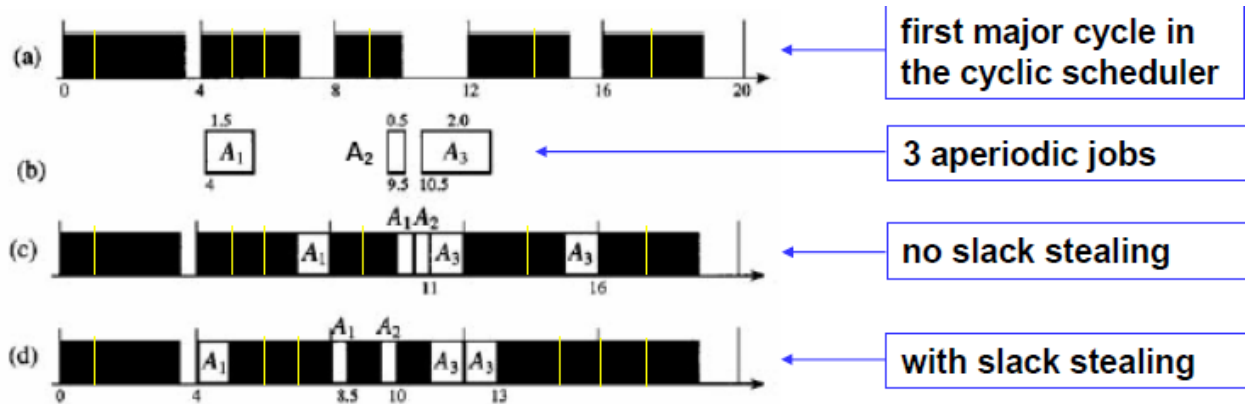
When an aperiodic job executes ahead of slices of periodic tasks, it consumes the slack in the frame. After y units of slack time are used by aperiodic jobs, the available slack is reduced to $f - x_k - y$. The cyclic executive can let aperiodic jobs execute in frame k as long as there is slack, that is, the available slack $f - x_k - y$ in the frame is larger than 0.

When the cyclic executive finds the aperiodic job queue empty, it lets the periodic task server execute the next slice in the current block. The amount of slack remains the same during this execution. As long as there is slack, the cyclic executive returns to examine the aperiodic job queue after each slice completes.



- Interval timer is used.
 - ☐ At beginning of frame timer is set to slack in frame.
 - ☐ whenever an aperiodic job executes slack is reduced.
 - ☐ when timer expires, slack is consumed and aperiodic job is preempted

Example:



With no slack stealing (c):

- job A_1 : released at $t=4$, starts at $t=7$, completes at $t=10.5$; response time = 6.5
- job A_2 : released at $t=9.5$, starts at $t=10.5$, completes at $t=11$; resp. time = 1.5
- job A_3 : released at $t=10.5$, starts at $t=11$, completes at $t=16$; resp. time = 5.5
- average response time = 4.5

With slack stealing (d):

- job A_1 : released at $t=4$, starts at $t=4$, completes at $t=8.5$; response time = 4.5
- job A_2 : released at $t=9.5$, starts at $t=9.5$, completes at $t=10$; resp. time = 0.5
- job A_3 : released at $t=10.5$, starts at $t=11$, completes at $t=13$; resp. time = 2.5
- average response time = 2.5

Fig. example illustrating slack stealing

SCHEDULING SPORADIC JOBS

Like jobs in periodic tasks, sporadic jobs have hard deadlines. On the other hand, their minimum release times and maximum execution times are unknown a priori. Consequently, it is impossible to guarantee a priori that all sporadic jobs can complete in time.

Acceptance Test

A common way to deal with this situation is to have the scheduler perform an acceptance test when each sporadic job is released. During an *acceptance test*, the scheduler checks whether the newly released sporadic job can be feasibly scheduled with all the jobs in the system at the time. Here, by a *job in the system*, we mean either a periodic job, for which time has already been allocated in the precomputed cyclic schedule, or a sporadic job which has been scheduled but not yet completed. If according to the existing schedule, there is a sufficient amount of time in the frames before its deadline to complete the newly released sporadic job without causing any job

in the system to complete too late, the scheduler accepts and schedules the job. Otherwise, the scheduler rejects the new sporadic job.

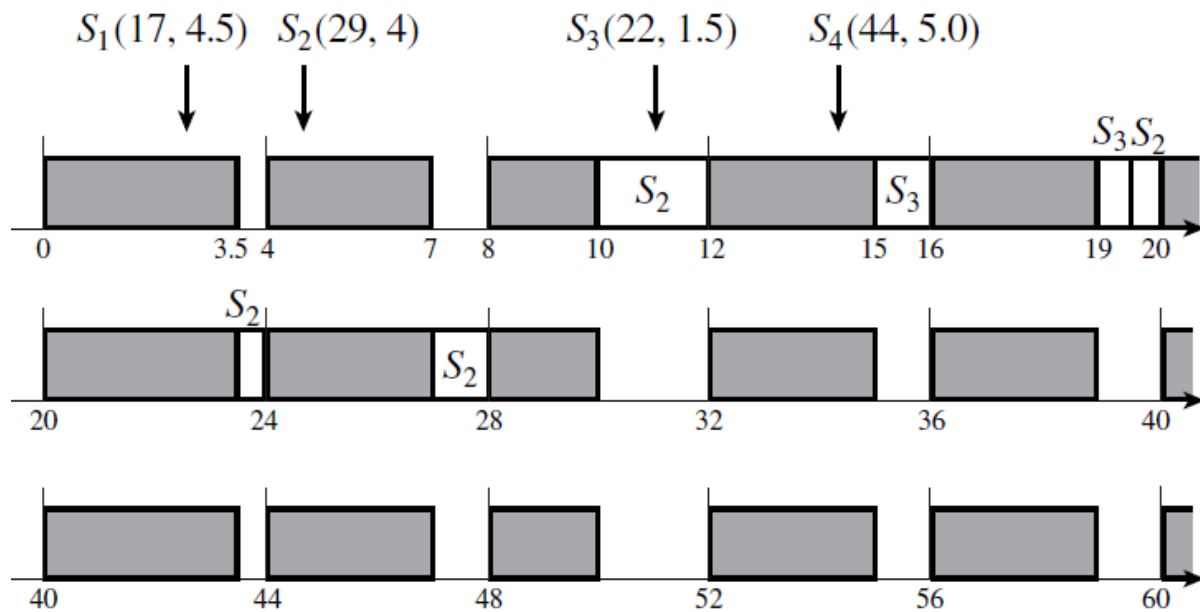
Conceptually, it is quite simple to do an acceptance test. To explain, let us suppose that at the beginning of frame t , an acceptance test is done on a sporadic job $S(d, e)$, with deadline d and (maximum) execution time e . (When it is not necessary to mention the deadline and execution time of the job, we will simply refer to it as S without these parameters.) Suppose that the deadline d of S is in frame $l+1$ (i.e., frame l ends before d but frame $l+1$ ends after d) and $l \geq t$.

Clearly, the job must be scheduled in the l th or earlier frames. The job can complete in time only if the *current (total) amount of slack time* $\sigma c(t, l)$ in frames $t, t+1, \dots, l$ is equal to or greater than its execution time e . Therefore, the scheduler should reject S if $e > \sigma c(t, l)$.

As we will see shortly, the scheduler may let a new sporadic job execute ahead of some previously accepted sporadic jobs. Therefore, the scheduler also checks whether accepting the new job may cause some sporadic jobs in the system to complete late. The scheduler accepts the new job $S(d, e)$ only if $e \leq \sigma c(t, l)$ and no sporadic jobs in system are adversely affected.

EDF Scheduling of the Accepted Jobs:

By virtue of its optimality, the EDF algorithm is a good way to schedule accepted sporadic jobs. For this purpose, the scheduler maintains a queue of accepted sporadic jobs in nondecreasing order of their deadlines and inserts each newly accepted sporadic job into this queue in this order. Whenever all the slices of periodic tasks scheduled in each frame are completed, the cyclic executive lets the jobs in the sporadic job queue execute in the order they appear in the queue. The scheduler allows aperiodic jobs to execute only when the accepted sporadic job queue is empty. Following figure gives an example. The frame size used here is 4. The shaded boxes show where periodic tasks are scheduled.



Description :

Suppose that at time 3, a sporadic job $S_1(17, 4.5)$ with execution time 4.5 and deadline 17 is released. The acceptance test on this job is done at time 4, that is, the beginning of frame 2. S_1 must be scheduled in frames 2, 3, and 4. In these frames, the total amount of slack time is only 4, which is smaller than the execution time of S_1 . Consequently, the scheduler rejects the job.

- At time 5, $S_2(29, 4)$ is released. Frames 3 through 7 end before its deadline. During the acceptance test at 8, the scheduler finds that the total amount of slack in these frames is 5.5. Hence, it accepts S_2 . The first part of S_2 with execution time 2 executes in the current frame.

- At time 11, $S_3(22, 1.5)$ is released. At time 12, the scheduler finds 2 units of slack time in frames 4 and 5, where S_3 can be scheduled. Moreover, there still is enough slack to complete S_2 even though S_3 executes ahead of S_2 . Consequently, the scheduler accepts S_3 . This job executes in frame 4.

- Suppose that at time 14, $S_4(44, 5)$ is released. At time 16 when the acceptance test is done, the scheduler finds only 4.5 units of time available in frames before the deadline of S_4 , after it has

accounted for the slack time that has already been committed to the remaining portions of $S2$ and $S3$. Therefore, it rejects $S4$. When the remaining portion of $S3$ completes in the current frame, $S2$ executes until the beginning of the next frame.

- The last portion of $S2$ executes in frames 6 and 7.

ALGORITHM FOR CONSTRUCTING STATIC SCHEDULES

The general problem of choosing a frame length for a given set of periodic tasks, segmenting the tasks if necessary, and scheduling the tasks to meet all their deadlines is NP-hard.

- In the special case of independent preemptable tasks, a polynomial-time solution is based on the Iterative Network Flow algorithm (INF algorithm).
- A system of independent preemptable periodic tasks whose relative deadlines are not less than their respective periods is schedulable iff the total utilization of the tasks is ≤ 1 .
 - if some relative deadlines are shorter than the period, a feasible schedule may not exist even when $U \leq 1$.

The INF algorithm is performed in 2 steps:

- step 1: find all the possible frame sizes of the system that meet the frame size constraints 2 and 3 but not necessarily constraint 1;
- step 2: apply INF algorithm starting with the largest possible frame.
- Example (slide 11):
 - $T_1 = (4,1)$; $T_2 = (5,2,7)$ and $T_3 = (20,5)$
 - frame sizes 2 and 4 meet constraints 2 and 3, but not 1.
- The INF algorithm iteratively tries to find a feasible cyclic schedule of the system for a possible frame size at a time, starting with the largest value.

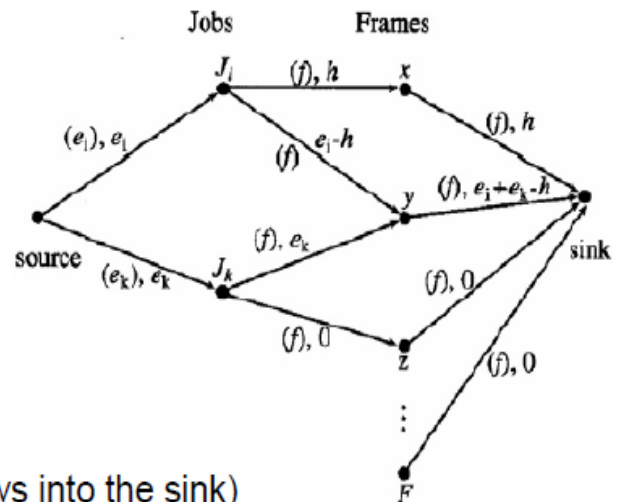
Network-Flow Graph:

The algorithm used during each iteration is based on the wellknown network-flow formulation of the preemptive scheduling problem. In the description of this formulation, it is more convenient to ignore the tasks to which the jobs belong and name the jobs to be scheduled in a major cycle of F frames $J1, J2, \dots, JN$. The constraints on when the jobs can be scheduled are represented by the *network-flow graph* of the system.

This graph contains the following vertices and edges; the capacity of an edge is a nonnegative number associated with the edge.

1. There is a **job vertex** J_i representing each job J_i , for $i = 1, 2, \dots, N$.
2. There is a **frame vertex** named j representing each frame j in the major cycle, for $j = 1, 2, \dots, F$.
3. There are two special vertices named **source** and **sink**.
4. There is a directed edge (J_i, j) from a job vertex J_i to a frame vertex j if the job J_i can be scheduled in the frame j , and the *capacity* of the edge is the frame size f .
5. There is a directed edge from the *source* vertex to every job vertex J_i , and the capacity of this edge is the execution time e_i of the job.
6. There is a directed edge from every frame vertex to the *sink*, and the capacity of this edge is f .

- The flow of an edge (J_i, j) gives the amount of time in frame j allocated to job J_i .
- A flow of an edge is a positive number that satisfies the following constraints:
 - \leq the edge capacity,
 - $\sum(\text{flows into vertex}) = \sum(\text{flows out})$.

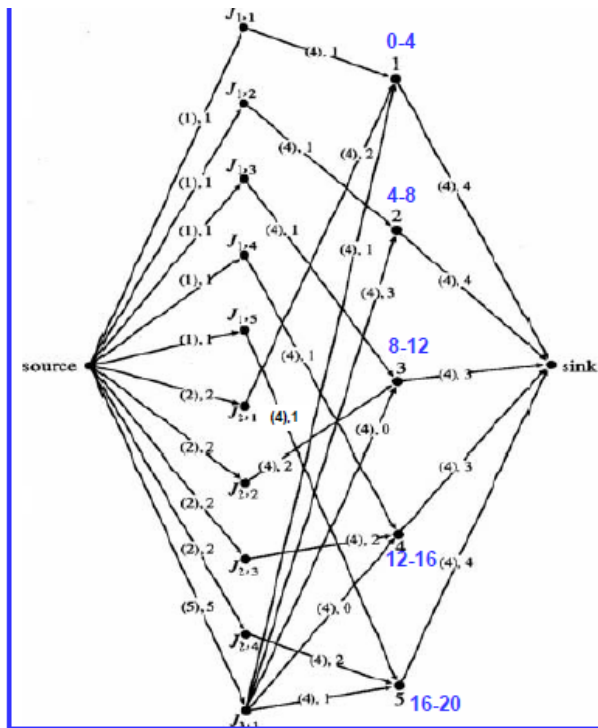


- Flow of a networkflow graph = $\sum(\text{all flows into the sink})$
- **Problem:** Find the maximum flow of network-flow graph: time complexity of algorithm = $((N+F)^3)$.
- Maximum flow $\leq \sum e_j$; if the set of flows of Jobs \rightarrow Frames edges gives maximum flow = e_j , then they represent a feasible preemptive schedule.

41

- Example:
 - $T_1 = (4, 1)$; $T_2 = (5, 2, 7)$; $T_3 = (20, 5)$
 $H = 20$; $U = 0.9$
 - frame sizes 2 and 4 meet constraints 2 and 3, (but not 1).
- Try first frame size 4: in H there are:
 - 5 frames,
 - 5 jobs of T_1 ; 4 jobs of T_2 ; 1 of T_3 .
- Each job (or job slice) is schedulable in a frame that is contained in its feasible interval, i.e. each job (or job slice) is scheduled in a frame that:
 - begins no sooner than its release time,
 - ends no later than its deadline.

job	feasible interval	schedulable in frame(s)
$J_{1,1}$	0 – 4	F_1 (0 – 4)
$J_{1,2}$	4 – 8	F_2 (4 – 8)
$J_{1,3}$	8 – 12	F_3 (8 – 12)
$J_{1,4}$	12 – 16	F_4 (12 – 16)
$J_{1,5}$	16 – 20	F_5 (16 – 20)
$J_{2,1}$	0 – 7	F_1 (0 – 4)
$J_{2,2}$	5 – 12	F_3 (8 – 12)
$J_{2,3}$	10 – 17	F_4 (12 – 16)
$J_{2,4}$	15 – 22	F_5 (16 – 20)
$J_{3,1}$	0 – 20	F_1, F_2, F_3, F_4, F_5



- $T_1 = (4, 1)$; $T_2 = (5, 2, 7)$; $T_3 = (20, 5)$
- edge (J_{ij}, k) from job vertex J_{ij} to frame vertex k is drawn if job J_{ij} is schedulable in frame k ;
- its flow gives the amount of time in frame k allocated to job J_{ij} .
- **INF algorithm:**
 - step 1: possible frame sizes 4, 2
 - step 2: try first with $f = 4$ (figure)
 - maximum flow is $18 = \sum e_i$
 - ⇒ **feasible schedule**
- The flows of the feasible schedule indicate that T_3 is to be partitioned in 3 slices and give their size.
- The time diagram with the job-slice schedule in each frame may now be drawn

Practical Considerations:

• Handling overruns:

- Jobs are scheduled based on maximum execution time, but failures might cause overrun.
- A robust system will handle this by either: 1) killing the job and starting an error recovery task; or 2) preempting the job and scheduling the remainder as an aperiodic job.
- Depends on usefulness of late results, dependencies between jobs, etc.

• Mode changes:

- A cyclic scheduler needs to know all parameters of real-time jobs a priori.
- Switching between modes of operation implies reconfiguring the scheduler and bringing in the code/data for the new jobs.
- This can take a long time: schedule the reconfiguration job as an aperiodic or sporadic task to ensure other deadlines met during mode change.

• Multiple processors:

Can be handled, but off-line scheduling table generation more complex.

Pros and Cons of Clock driven Scheduling:

- Conceptual simplicity
 - Ability to consider complex dependencies, communication delays, and resource contention among jobs when constructing the static schedule , guaranteeing absence of deadlocks and unpredictable delays.
 - Entire schedule is captured in a static table.
 - Different operating modes can be represented by different tables.
 - No concurrency control or synchronization required.
 - If completion time jitter requirements exist, can be captured in the schedule.
- When workload is mostly periodic and the schedule is cyclic, timing constraints can be checked and enforced at each frame boundary.
- Choice of frame size can minimize context switching and communication overheads.
- Relatively easy to validate, test and certify.

Cons:

- Inflexible
 - Pre-compilation of knowledge into scheduling tables means that if anything changes materially, have to redo the table generation.
 - Best suited for systems which are rarely modified once built.
- Other disadvantages:
 - Release times of all jobs must be fixed.
 - All possible combinations of periodic tasks that can execute at the same time must be known a priori, so that the combined schedule can be pre-computed.

CHAPTER 6

Priority-Driven Scheduling of Periodic Tasks:

Assumptions

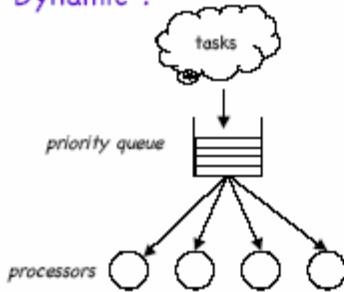
- The tasks are independent and.
- There are no aperiodic and sporadic tasks.
- Every job is ready for execution as soon as it is released,
- Every job can be preempted at any time, and never suspends itself,
- Scheduling decisions are made immediately upon job releases and completions,
- Context switch overhead is negligible
- Fixed number of periodic tasks,
- Scheduling on uniprocessor systems

Multiprocessor Scheduling

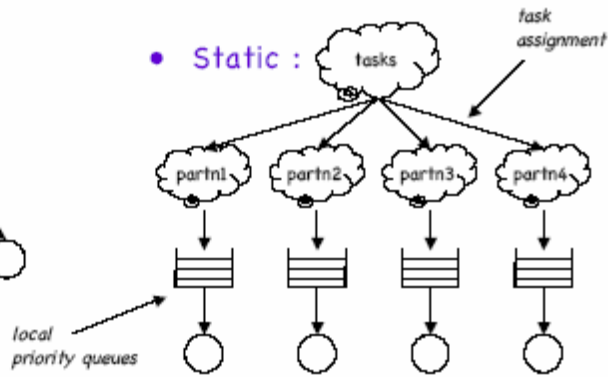
➤ Dynamic vs. Static

A multiprocessor priority-driven system is either dynamic or static. In a static system, all the tasks are partitioned into subsystems. Each subsystem is assigned to a processor, and tasks on each processor are scheduled by themselves. In contrast, in a dynamic system, jobs ready for execution are placed in one common priority queue and dispatched to processors for execution as the processors become available.

- Dynamic :



- Static :



Why not dynamic multiprocessor scheduling?

In most cases, dynamic systems perform better than static systems but we don't know how to determine the worst case performance i.e. the performance of priority driven algorithms can be unacceptably poor. For this reason most hard real time systems are static systems.

Example showing poor performance of priority driven systems:

Consider $m+1$ periodic independent tasks, $T_i = (\text{period or relative deadline, execution time}) = (1, 2\varepsilon)$ for 1 to m tasks (i.e. first tasks) and for $m+1^{\text{th}}$ task $T_{m+1} = (1 + \varepsilon, 1)$. Here $D_i = p_i$ and phase = 0. Priorities are assigned in EDF basis.

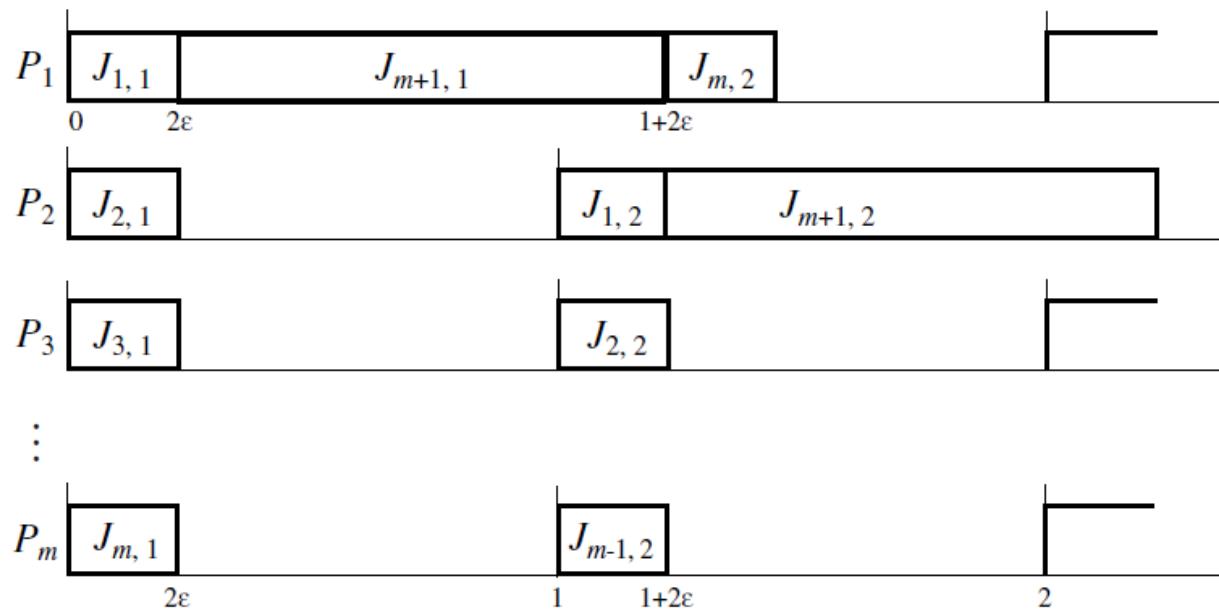


Fig. A dynamic EDF schedule on m processors

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

Here we see that $J_{m+1,1}$ doesnot complete until $1+2\epsilon$ and hence misses its deadline.

$$\text{Total utilization} = m(2\epsilon/1) + 1/(1+\epsilon) = 2m\epsilon/1 + 1/(1+\epsilon) = 2m\epsilon + 1/(1+\epsilon).$$

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 the 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 m tasks, $2m\epsilon$, is equal to or less than 1, this system can be feasibly scheduled on two processors if we put T_{m+1} on one processor and the other tasks on the other processor and schedule the task(s) on each processor according to either of these priority-driven algorithms.

Fixed Priority vs. Dynamic Priority Algorithms:

Priority-driven algorithms differ from each other in how priorities are assigned to jobs. We classify algorithms for scheduling periodic tasks into two types: **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. In contrast, 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 tasks changes as jobs are released and completed. This is why this type of algorithm is said to be “dynamic.”

Indeed, we have three categories of algorithms:

- fixed-priority algorithms, Rate Monotonic (RM) , Deadline Monotonic(DM).
- task-level dynamic-priority (and job level fixed-priority) algorithms EDF, FIFO, LIFO
- job-level (and task-level) dynamic algorithms: LST , RR

Except where stated otherwise, by dynamic-priority algorithms, we mean task-level dynamic-priority (and job-level fixed-priority) algorithms.

Rate Monotonic:

- The rate (of job releases) of a task = $1/\text{period}$.
- The higher its rate, the higher the priority of the task.

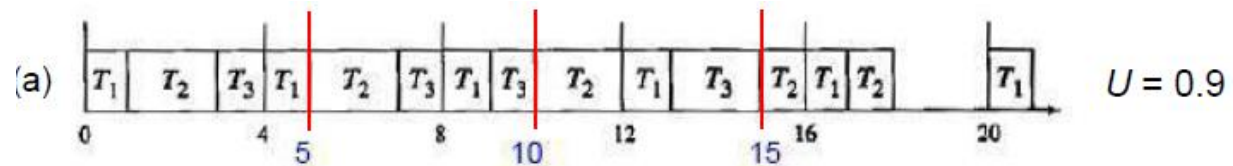
For example: (a): $T1 = (4,1)$; $T2 = (5,2)$; $T3 = (20,5)$.

$$\text{Rate}(T1) = 1/4 = .25$$

$$\text{Rate}(T2) = 1/5 = .2$$

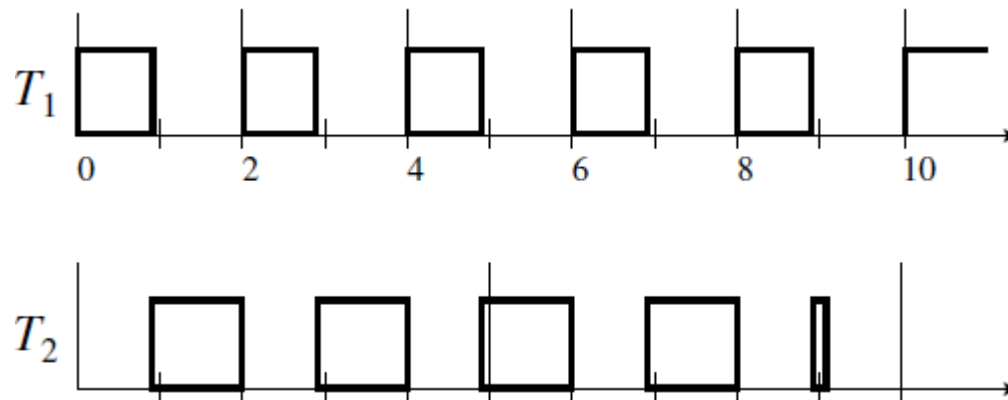
$\text{Rate}(T3) = 1/20 = .05$, Therefore $T1$ has highest priority.

The RM schedule is :



Example (b): $T1 = (2,0.9)$; $T2 = (5,2.3)$

The schedule is :



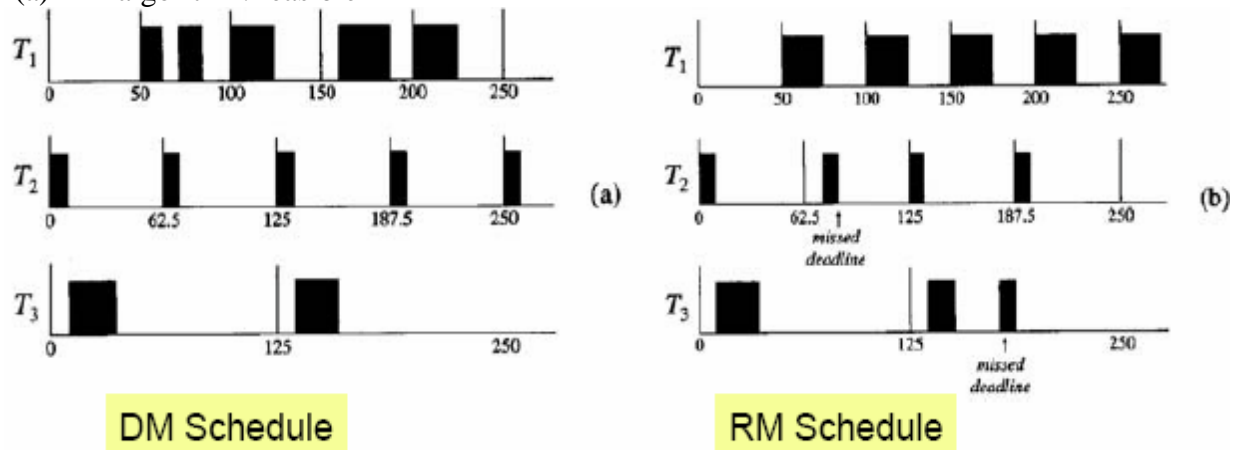
Deadline Monotonic:

- Well known fixed priority algorithm.
- The shorter its relative deadline, the higher the priority of the task.

$T1 = (50,50,25,100)$; $T2 = (0,62.5,10,20)$; $T3 = (0,125,25,50)$

$u1 = 0.5$; $u2 = 0.16$; $u3 = 0.2 \Rightarrow U = 0.86$; $H = 250$.

-(a) DM algorithm: feasible



-(b) RM algorithm: not feasible \Rightarrow schedule not optimal

Earliest-Deadline-First (EDF):

The earlier its absolute deadline, the higher the priority of the job.

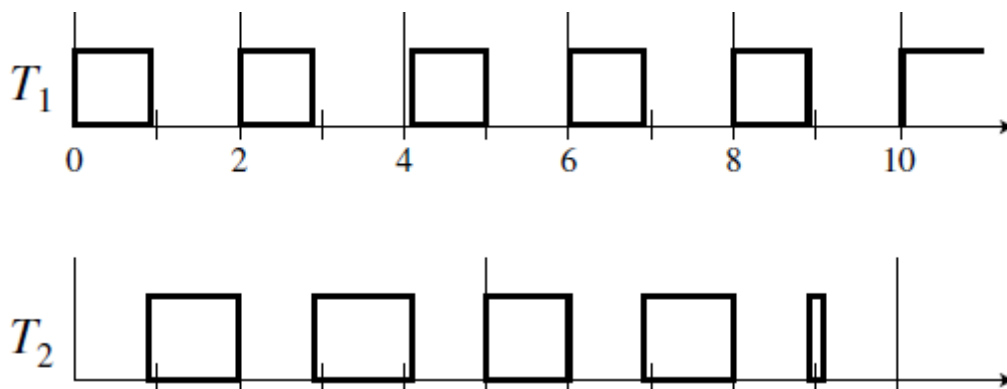


Fig. An earliest-deadline-first schedule of (2, 0.9) and (5, 2.3)

Description:

-At time 0, the first jobs $J_{1,1}$ and $J_{2,1}$ of both tasks are ready. The (absolute) deadline of $J_{1,1}$ is $= r+D=0+2= 2$ while the deadline of $J_{2,1}$ is $= 0+5=5$. Consequently, $J_{1,1}$ has a higher priority and executes. When $J_{1,1}$ completes, $J_{2,1}$ begins to execute.

- At time 2, $J1,2$ is released, and its deadline is 4, earlier than the deadline of $J2,1$. Hence, $J1,2$ is placed ahead of $J2,1$ in the ready job queue. $J1,2$ preempts $J2,1$ and executes.
- At time 2.9, $J1,2$ completes. The processor then executes $J2,1$

At time 4, $J1,3$ is released; its deadline is 6, which is later than the deadline of $J2,1$.

Hence, the processor continues to execute $J2,1$.

- At time 4.1, $J2,1$ completes, the processor starts to execute $J1,3$, and so on.

(Refer hand note for detail)

Hw# Explain when DM and RM are identical and when differ?

MAXIMUM SCHEDULABLE UTILIZATION

we say that 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 schedules of the system exist.

Schedulable Utilizations of the EDF Algorithm:

A system T of independent, preemptable tasks with relative deadlines equal to their respective periods 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 example : for task $T1=(2,0.9), T2=(5, 2.3)$ is feasible but it would not be schedulable if its relative deadlines were 3 instead of 5.

Density:

The ratio of the execution time e_k of a task T_k to the minimum of its relative deadline D_k and period p_k the *density* δ_k of the task.

In other words, the density of T_k is $e_k/\min(D_k, p_k)$. The sum of the densities of all tasks in a system is the *density* of the system and is denoted by Δ when $D_i < p_i$ for some task T_i , $\Delta > U$. If the density of a system is larger than 1, the system may not be feasible. For example,

ex.: $T_1=(2,0.9)$; $T_2=(5,2.3,3)$; $\Delta = 0.9/2+2.3/3 = 7.3/6 > 1$, not feasible.

and the tasks are not schedulable by any algorithm. On the other hand, any system is feasible if its density is equal to or less than 1.

Theorem: A system T of independent, preemptable, periodic tasks can be feasibly scheduled in one processor if its density is less than or equal to 1.

The condition given by this theorem is not necessary for a system to be feasible. A system may nevertheless be feasible when its density is greater than 1. The system consisting of $(2, 0.6, 1)$ and $(5, 2.3)$ is an example. Its density is larger than 1, but it is schedulable according to the EDF algorithm.

A SCHEDULABILITY TEST FOR FIXED-PRIORITY TASKS WITH SHORT RESPONSE TIMES:

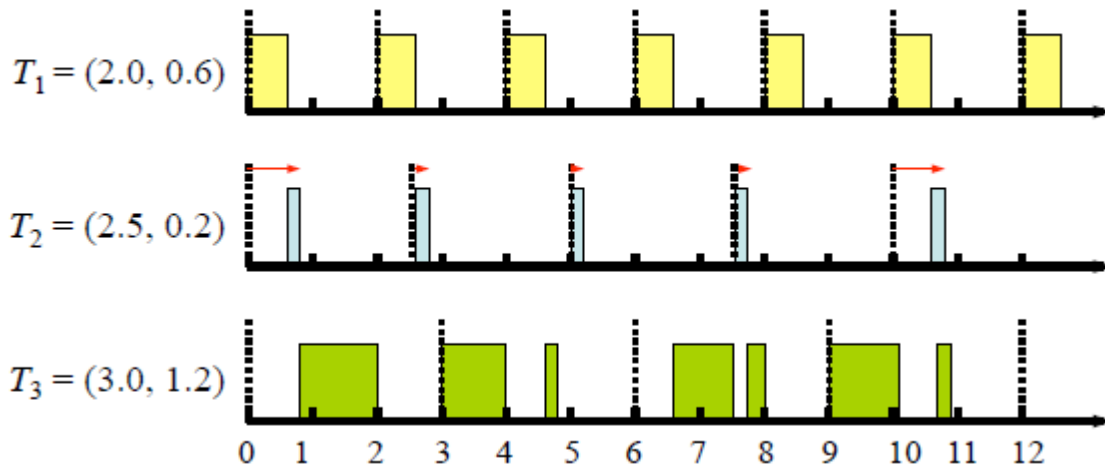
we consider the case where the response times of the jobs are smaller than or equal to their respective periods. In other words, every job completes before the next job in the same task is released.

Critical Instant:

A critical instant for a job is the 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 with higher priority are released, and must wait for all those jobs to complete 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

Example:



3 tasks scheduled using rate-monotonic

- Response times of jobs in T_2 are: $r_{2,1} = 0.8$, $r_{2,3} = 0.3$, $r_{2,3} = 0.2$, $r_{2,4} = 0.3$, $r_{2,5} = 0.8$, ...

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

THEOREM 6.5 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, $r_{i,c} = r_{k,lk}$ for some lk for every $k = 1, 2, \dots, i - 1$.

SCHEDULABILITY TEST FOR FIXED-PRIORITY TASKS WITH ARBITRARY RESPONSE TIMES

This section describes a general time-demand analysis method developed by Lehoczky to determine the schedulability of tasks whose relative deadlines are larger than their 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.

Busy Intervals

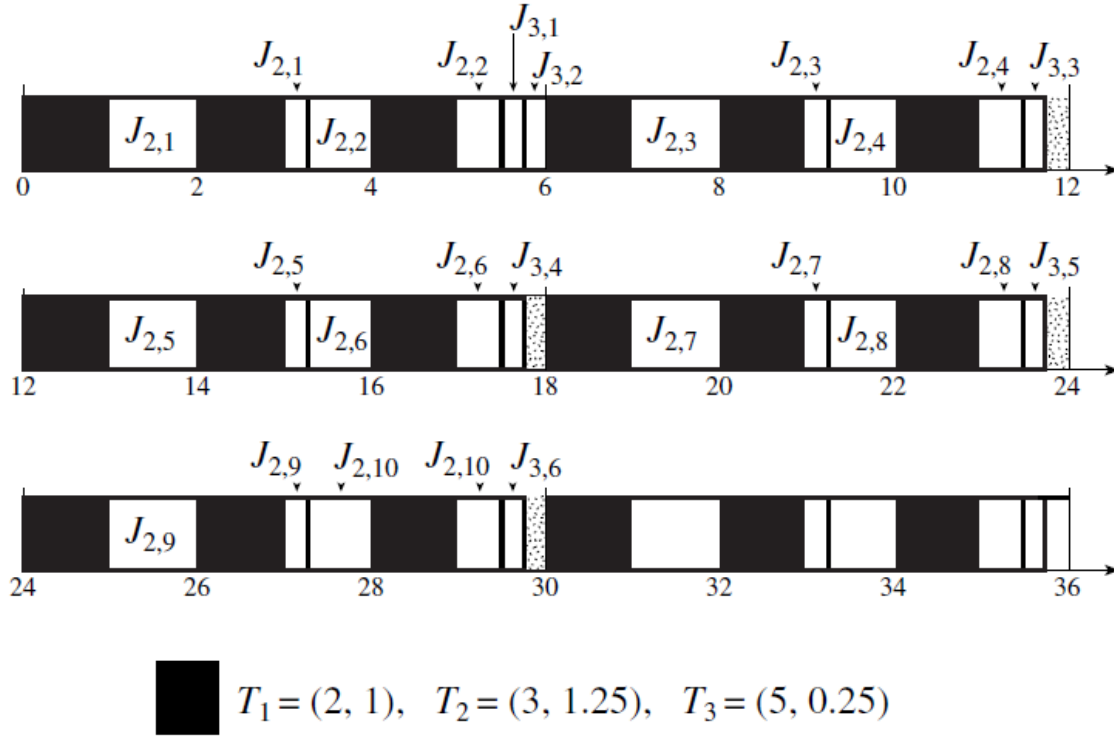


Fig. illustrating Busy Intervals

A *level- π_i busy interval* $(t_0, t]$ begins at an instant t_0 when (1) all jobs in \mathbf{T}_i released before the instant have completed and (2) a job in \mathbf{T}_i is released. The interval ends at the first instant t after t_0 when all the jobs in \mathbf{T}_i released since t_0 are complete. In other words, in the interval $(t_0, t]$, the processor is busy all the time 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. Hence, when computing the response times of jobs in \mathbf{T}_i , we can consider every level- π_i busy interval independently from other level- π_i busy intervals.

With a slight abuse of the term, we say that a level- π_i busy interval is *in phase* if the first jobs 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. Otherwise, we say that the tasks have arbitrary phases in the interval. As an example, Figure shows the schedule of three tasks $T_1 = (2, 1)$, $T_2 = (3, 1.25)$, and $T_3 = (5, 0.25)$ in the first hyperperiod. The filled 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. As expected, 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 which are the least common multiples of the periods of tasks T_1 and T_2 . The lengths of these intervals are all 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 this job completes at 5.75, the second job in T_3 is scheduled. At time 6, all the jobs 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 arbitrary phases.

General Schedulability Test

The general schedulability test described below relies on the fact that when determining the schedulability of a task T_i in a system in which the response times of jobs can be larger than their respective periods, it still suffices to confine our attention to the special case where the tasks are in phase.

General Time-Demand Analysis Method

Test one task at a time starting from the highest priority task T_1 in order of decreasing priority. For the purpose of determining whether a task T_i is schedulable, assume that all the tasks are in phase and the first level- π_i busy interval begins at time 0.

While testing whether all the jobs in T_i can meet their deadlines (i.e., whether T_i is schedulable), consider the subset \mathbf{T}_i of tasks with priorities π_i or higher.

- (i) If the first job of every task in \mathbf{T}_i completes by the end of the first period of the task, check whether the first job $J_{i,1}$ in T_i meets its deadline. T_i is schedulable if $J_{i,1}$ completes in time. Otherwise, T_i is not schedulable.
 - (ii) If the first job of some task in \mathbf{T}_i does not complete by the end of the first period of the task, do the following:
 - (a) Compute the length of the in phase level- π_i busy interval by solving the equation $t = \sum_{k=1}^i \lceil \frac{t}{p_k} \rceil e_k$ iteratively, starting from $t^{(1)} = \sum_{k=1}^i e_k$ until $t^{(l+1)} = t^{(l)}$ for some $l \geq 1$. The solution $t^{(l)}$ is the length of the level- π_i busy interval.
 - (b) Compute the maximum response times of all $\lceil t^{(l)}/p_i \rceil$ jobs of T_i in the in-phase level- π_i busy interval in the manner described below and determine whether they complete in time.
- T_i is schedulable if all these jobs complete in time; otherwise T_i is not schedulable.

SCHEDULABILITY CONDITIONS FOR THE RM AND DM ALGORITHMS:

Consider case when the relative deadline of every task is equal to its period. For such systems, the RM and DM algorithms are identical.

THEOREM 6.11. A system of n independent, preemptable periodic tasks with relative deadlines equal to their respective periods can be feasibly scheduled on a processor according to the RM algorithm if its total utilization U is less than or equal to $U_{RM}(n) = n(2^{1/n} - 1)$

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... how does this affect the system?

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 occur 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 inversions may occur, these may cause a higher priority task to miss its deadline.
 - When attempting to determine if a task meets all of its deadlines, must consider not only all the tasks that have higher priorities, but also nonpreemptable regions of lower-priority tasks.
- Add the blocking time in when calculating if a task is schedulable.

Self-suspension

- A job may invoke an external operation (e.g. request an I/O operation), during which time it is suspended.
- This means the task is no longer strictly periodic... again need to take into account self-suspension time when calculating a schedule.

• Context Switches

- Assume maximum number of context switches K_i for a job in T_i is known; each takes t_{CS} time units.
- Compensate by setting execution time of each job, $e_{actual} = e + 2t_{CS}$
(more if jobs self-suspend, since additional context switches)

Tick Scheduling

- All of our previous discussion of priority-driven scheduling was driven by job release and job completion events
- Alternatively, can perform priority-driven scheduling at periodic events (timer interrupts) generated by a hardware clock i.e. tick (or time-based) scheduling.

- Additional factors to account for in schedulability analysis
 - The fact that a job is ready to execute will not be noticed and acted upon until the next clock interrupt; this will delay the completion of the job
 - A ready job that is yet to be noticed by the scheduler must be held somewhere other than the ready job queue, the *pending job* queue
 - When the scheduler executes, it moves jobs in the pending queue to the ready queue according to their priorities; once in ready queue, the jobs execute in priority order.

Chapter-7

Scheduling aperiodic and sporadic jobs in priority driven systems:

Assumptions and approaches-objectives:

Assumptions:

- The parameters of each sporadic job become known after it is released.
- Sporadic jobs may arrive at any instant, even immediately after each other.
- Moreover, their execution times may vary widely, and
- Their deadlines are arbitrary.

It is impossible for some sporadic jobs to meet their deadlines no matter what algorithm we use to schedule them. The only alternatives are:

- To reject the sporadic jobs that cannot complete in time or
- To accept all sporadic jobs and allow some of them to complete late. Which alternative is appropriate depends on the application.

Objectives, Correctness, and Optimality:

We assume that we are given the parameters $\{p_i\}$ and $\{e_i\}$ of all the periodic tasks and a priority-driven algorithm used to schedule them. Moreover, when the periodic tasks are scheduled according to the given algorithm and there are no aperiodic and sporadic jobs, the periodic tasks meet all their deadlines.

We assume that the operating system maintains the priority queues shown in following fig. The ready periodic jobs are placed in the periodic task queue, ordered by their priorities that are assigned according to the given periodic task scheduling algorithm. Similarly, each accepted sporadic job is assigned a priority and is placed in a priority queue, which may or may not be the same as the periodic task queue. Each newly arrived aperiodic job is placed in the aperiodic job queue. Moreover, aperiodic jobs are inserted in the aperiodic job queue and newly arrived sporadic jobs are inserted into a waiting queue to await acceptance without the intervention of the scheduler.

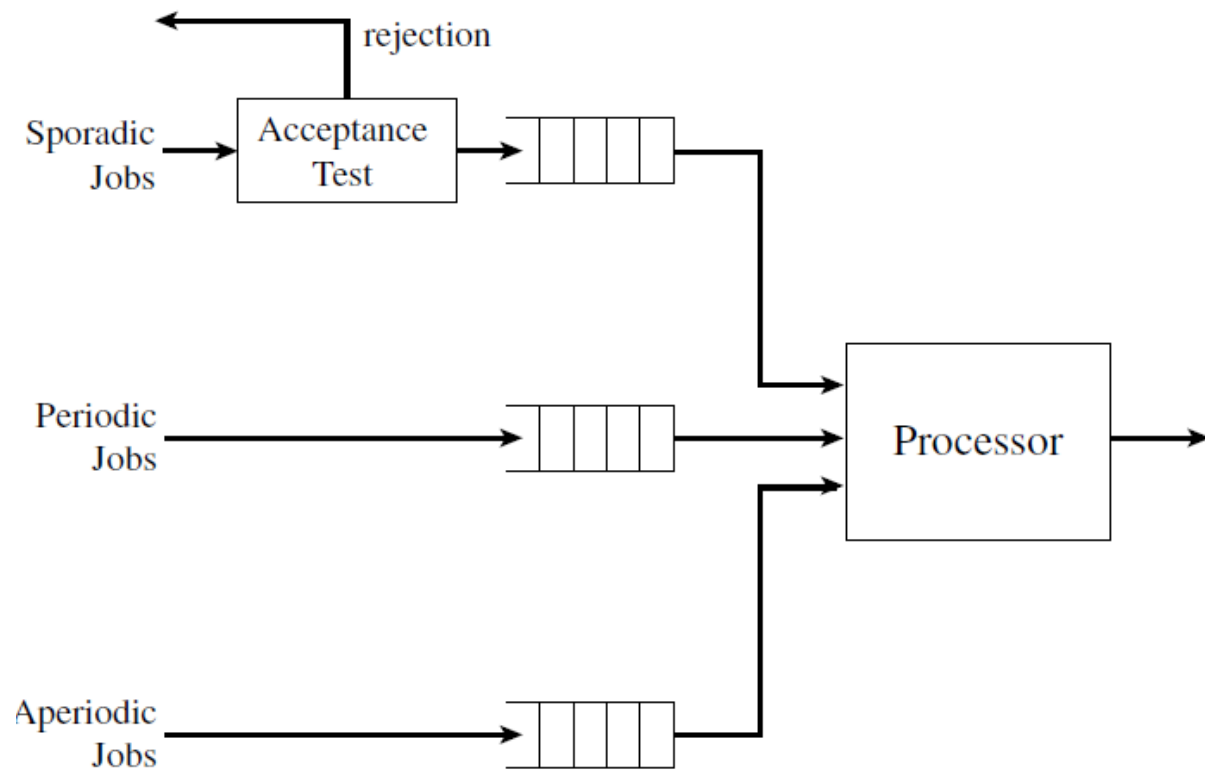


Fig. Priority queues maintained by Operating System

Aperiodic job and sporadic job scheduling algorithms; they are solutions to the following problems:

1. Based on the execution time and deadline of each newly arrived sporadic job, the scheduler decides whether to accept or reject the job. If it accepts the job, it schedules the job so that the job completes in time without causing periodic tasks and previously accepted sporadic jobs to miss their deadlines. The problems are how to do the acceptance test and how to schedule the accepted sporadic jobs.
2. The scheduler tries to complete each aperiodic job as soon as possible. The problem is how to do so without causing periodic tasks and accepted sporadic jobs to miss their deadlines.

Such an algorithm is *correct* if it produces only correct schedules of the system. By a *correct schedule*, we mean one according to which periodic and accepted sporadic jobs never miss their deadlines.

An aperiodic job scheduling algorithm is optimal if it minimizes either the response time of the aperiodic job at the head of the aperiodic job queue or the average response time of all the aperiodic jobs for the given queueing discipline. An algorithm for (accepting and) scheduling sporadic jobs is optimal if it accepts each sporadic job newly offered to the system and schedules the job to complete in time if and only if the new job can be correctly scheduled to complete in time by some means.

Aperiodic job scheduling

- Background:

- Aperiodic job queue has always lowest priority among all queues.
- Periodic tasks (and accepted sporadic jobs) always meet deadlines.
- Simple to implement.
- Problem: execution of aperiodic jobs may be unduly delayed.

- Interrupt-Driven:

- Response time as short as possible.
- Periodic tasks may miss some deadlines.

- Slack-Stealing:

- Postpone execution of periodic tasks only when it is safe to do so:
 - well-suited for clock-driven environments.
 - what about priority-driven environments? (quite complicated).

- Polling Server:

- a periodic task that is scheduled with the other periodic tasks,
- executes the first job in the aperiodic job queue (when not empty).

Polling Server

Polling Server (Poller) $PS = (p_s, e_s)$: scheduled as a periodic task, the polling server takes care of executing aperiodic jobs when there are any waiting;

- p_s : poller becomes ready for execution every p_s time units,
- e_s : is the upper bound on execution time.

- Terminology:

- execution budget: e_s
- size of the server: $u_s = e_s / p_s$ (it's the maximum utilization factor of PS),

- replenishment rule: the budget is set to e_s at the beginning of each period.
- consumption rules:
 - the poller consumes its budget at the rate of 1 per time unit, while it is executing aperiodic jobs;
 - the poller exhausts its budget whenever it finds the aperiodic job queue empty;
- Whenever the budget is exhausted, the scheduler removes the poller from the periodic queue until it is replenished;
- The server is idle when the aperiodic job queue is empty;
- The server is backlogged when the aperiodic job queue is not empty.

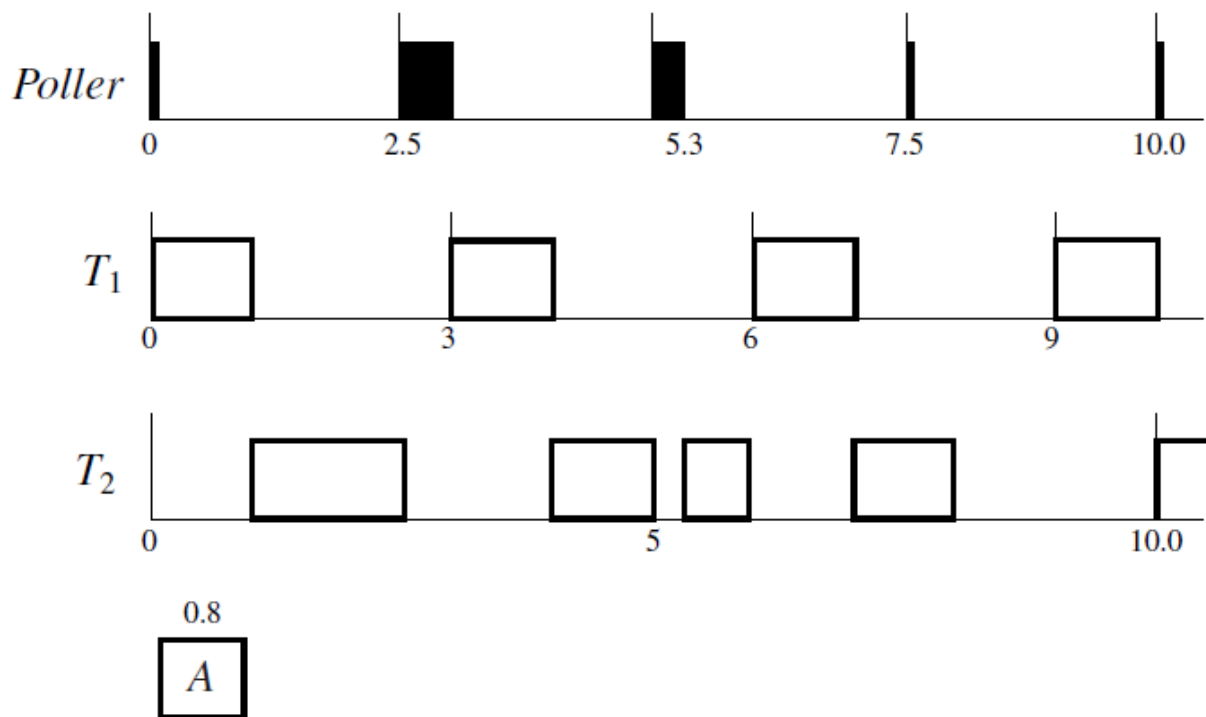


Fig. Polling for $T_1 = (3, 1)$, $T_2 = (10, 4)$, poller = $(2.5, 0.5)$ and A: $r=0.1$, $e=0.8$

Descriptions of above fig.:

At the beginning of the first polling period, the poller's budget is replenished, but when it executes, it finds the aperiodic job queue empty. Its execution budget is consumed instantaneously, and its execution suspended immediately. The aperiodic job A arrives a short time later and must wait in the queue until the beginning of the second polling period when the poller's budget is replenished. The poller finds A at head of the queue at time 2.5 and executes the job until its execution budget is exhausted at time 3.0. Job A remains in the aperiodic job

queue and is again executed when the execution budget of the poller is replenished at 5.0. The job completes at time 5.3, with a response time of 5.2. Since the aperiodic job queue is empty at 5.3, the budget of the poller is exhausted and the poller suspends.

Bandwidth Preserving Servers

Problem with polling servers:

- aperiodic jobs released after the poller has found the queue empty, must wait for the poller to examine the queue again, one polling period later: their response time is unduly longer;
- If the poller could preserve its budget, when it finds the aperiodic job queue empty, and use it later in the period, the response time of some aperiodic jobs would be shortened.

• Bandwidth-preserving server algorithms:

In previous example, if the poller were able to examine the queue again at time 0.1, then job *A* would complete in the second polling period, making its response time significantly shorter.

Algorithms that improve the polling approach in this manner are called ***bandwidth preserving server*** algorithms. Bandwidth-preserving servers are periodic servers. Each type of server is defined by a set of ***consumption and replenishment*** rules. The former give the conditions under which its execution budget is preserved and consumed. The latter specify when and by how much the budget is replenished.

i.e.

- improve in this manner upon polling approach,
- use periodic servers,
- are defined by consumption and replenishment rules.

• 3 types of bandwidth-preserving server algorithms:

- ☐ **deferrable servers,**
- ☐ **sporadic servers,**
- ☐ **constant utilization / total bandwidth / weighted fair queuing - servers.**

DEFERRABLE SERVERS

A *deferrable server* is the simplest of bandwidth-preserving servers. Like a poller, the execution

budget of a deferrable server with period p_s and execution budget e_s is replenished periodically with period p_s . Unlike a poller, however, when a deferrable server finds no aperiodic job ready for execution, it preserves its budget. However Any budget held prior to replenishment is lost (no carry-over).

Operations of Deferrable Servers

Specifically, the consumption and replenishment rules that define a deferrable server (p_s, e_s) are as follows.

Consumption Rule

The execution budget of the server is consumed at the rate of one per unit time whenever the server executes.

Replenishment Rule

The execution budget of the server is set to e_s at time instants $k p_s$, for $k = 0, 1, 2, \dots$.

Note that the server is not allowed to cumulate its budget from period to period. Stated in another way, any budget held by the server immediately before each replenishment time is lost.

Suppose that the task $(2.5, 0.5)$ is a deferrable server. When it finds the aperiodic job queue empty at time 0, it suspends itself, with its execution budget preserved. When aperiodic job A arrives at 0.1, the deferrable server resumes and executes A . At time 0.6, its budget completely consumed, the server is suspended. It executes again at time 2.5 when its budget is replenished. When A completes at time 2.8, the aperiodic job queue becomes empty. The server is suspended, but it still has 0.2 unit of execution budget. If another aperiodic job arrives later, say at time 4.0, the server resumes at that time.

Another example:

shows that the deferrable server $TDS = (3, 1)$ has the highest priority. The periodic tasks $T1 = (2.0, 3.5, 1.5)$ and $T2 = (6.5, 0.5)$ and the server are scheduled rate-monotonically. Suppose that an aperiodic job A with execution time 1.7 arrives at time 2.8.

1. At time 0, the server is given 1 unit of budget. The budget stays at 1 until time 2.8. When A arrives, the deferrable server executes the job. Its budget decreases as it executes.
2. Immediately before the replenishment time 3.0, its budget is equal to 0.8. This 0.8 unit is lost at time 3.0, but the server acquires a new unit of budget. Hence, the server continues

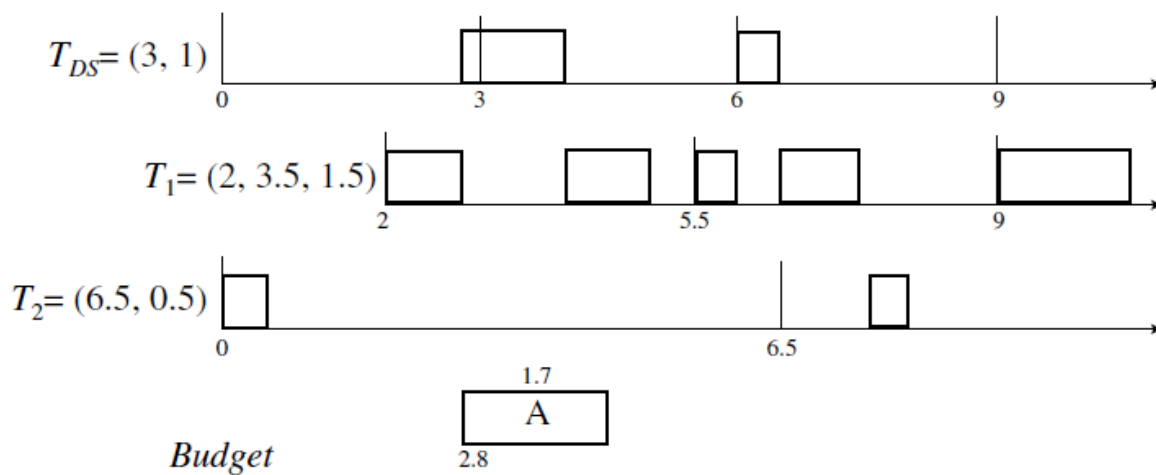
to execute.

3. At time 4.0, its budget is exhausted. The server is suspended, and the aperiodic job *A* waits.

4. At time 6.0, its budget replenished, the server resumes to execute *A*.

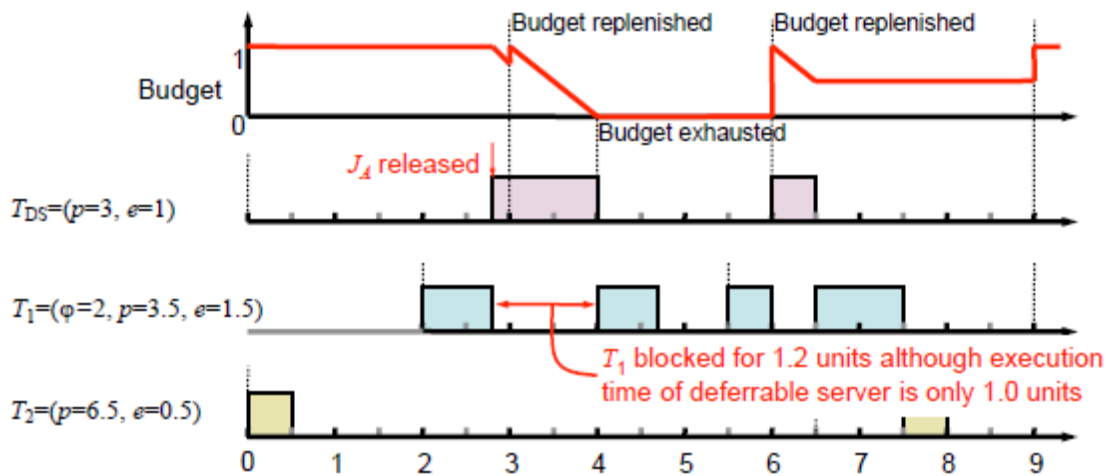
5. At time 6.5, job *A* completes. The server still has 0.5 unit of budget. Since no aperiodic job waits in the queue, the server suspends itself holding this budget.

Section 7.2 Deferrable Servers



Limitations of Deferrable Servers:

Limitation of deferrable servers – they may delay lower-priority tasks for more time than a periodic task with the same period and execution time:



Sporadic Servers

- A sporadic server is designed to eliminate this limitation.
- A different type of bandwidth preserving server: several different subtypes
- More complex consumption and replenishment rules ensure that a sporadic server with period p_s and budget e_s never demands more processor time than a periodic task with the same parameters.

Simple Fixed-Priority Sporadic Server:

System, T , of independent preemptable periodic tasks and a sporadic server with parameters (p_s, e_s)

- Fixed-priority scheduling; system can be scheduled if sporadic server behaves as a periodic task with parameters (p_s, e_s)
- Define:
 - T_H : the periodic tasks with higher priority than the server (may be empty)
 - t_r : the last time the server budget replenished
 - t_f : the first instant after t_r at which the server begins to execute
- At any time t define:
 - *BEGIN* as the start of the earliest busy interval in the most recent contiguous sequence of busy intervals of T_H starting before t (busy intervals are contiguous if the later one starts immediately the earlier one ends)

- END as the end of the latest busy interval in this sequence if this interval ends before t ; define $END = \infty$ if the interval ends after t .

Consumption rule:

- At any time $t \geq t_r$, if the server has budget and if either of the following two conditions is true, the budget is consumed at the rate of 1 per unit time:
 - C1: The server is executing
 - C2: The server has executed since t_r and $END < t$
- When they are not true, the server holds its budget.

That is:

- The server executes for no more time than it has execution budget.
- The server retains its budget if:
 - A higher-priority job is executing, or
 - It has not executed since t_r .
- Otherwise, the budget decreases when the server executes, or if it idles while it has budget.

Replenishment Rules:

- R1: When system begins executing, and each time budget is replenished, set the budget to e_s and $t_r =$ the current time.
- R2: When server begins to execute (defined as time t_f)

if $END = t_f$ then

$t_e = \max(t_r, BEGIN)$ $//t_e$ is effective replenishment time

else if $END < t_f$ then

$t_e = t_f$

The next replenishment time is set to $t_e + p_s$

- R3: budget replenished at the next replenishment time, unless:
 - If $t_e + p_s$ is earlier than t_f the budget is replenished as soon as it is exhausted
- If T becomes idle before $t_e + p_s$, and becomes busy again at t_b , the budget is replenished at $\min(t_b, t_e + p_s)$.

Example: $T1 = (3, 0.5)$ $T2 = (4, 1.0)$ $T3 = (19, 4.5)$ $TSS = (5, 1.5)$

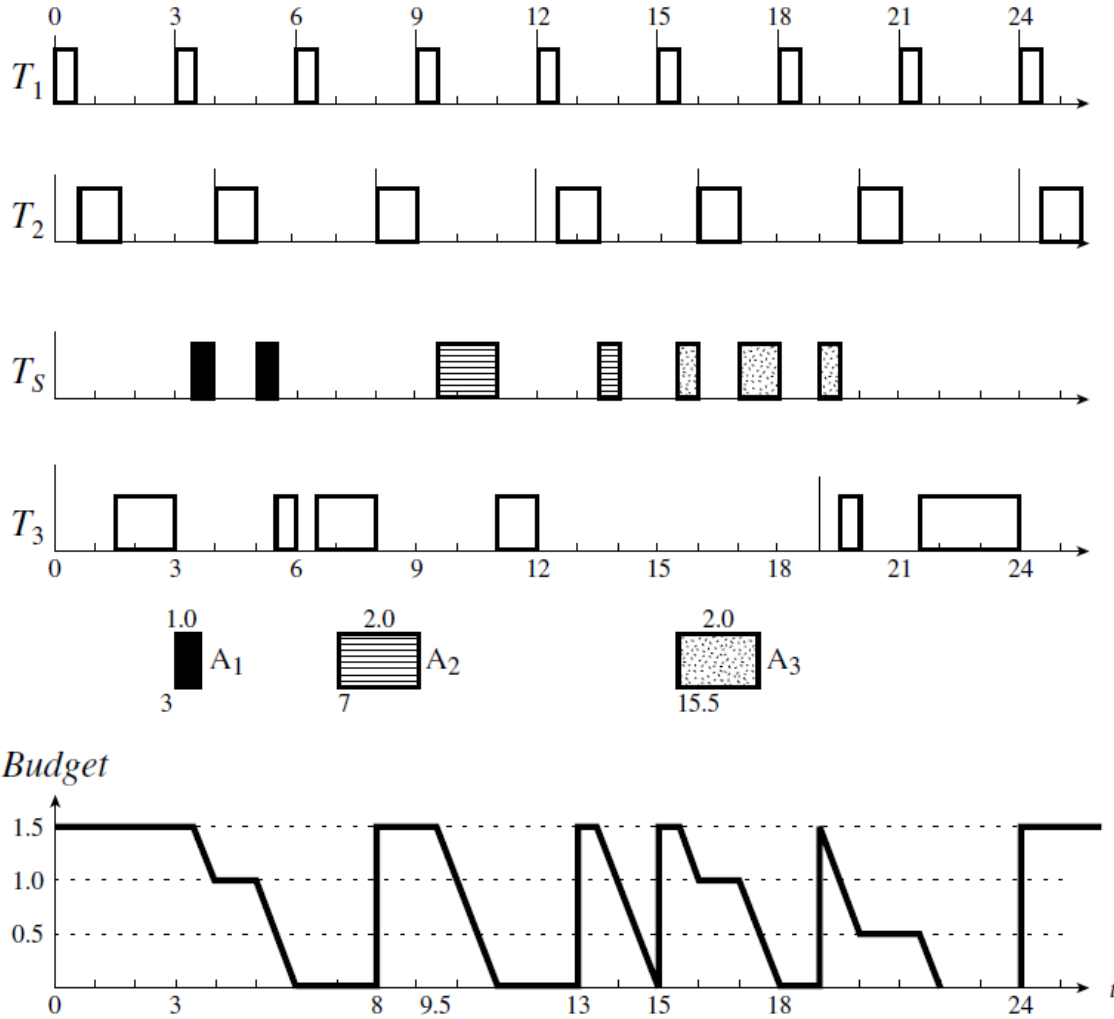


FIGURE 7-8 Example illustrating the operations of a simple sporadic server: $T_1 = (3, 0.5)$, $T_2 = (4, 1.0)$, $T_3 = (19, 4.5)$, $T_s = (5, 1.5)$.

Descriptions:

1. From time 0 to 3, the aperiodic job queue is empty and the server is suspended. Since it has not executed, its budget stays at 1.5. At time 3, the aperiodic job A_1 with execution time 1.0 arrives; the server becomes ready. Since the higher-priority task $(3, 0.5)$ has a job ready for execution, the server and the aperiodic job wait.
2. The server does not begin to execute until time 3.5. At the time, tr is 0, $BEGIN$ is equal to 3, and END is equal to 3.5. According to rule R2, the effective replenishment time te is equal to $\max(0, 3.0) = 3$, and the next replenishment time is set at 8.
3. The server executes until time 4; while it executes, its budget decreases with time.

4. At time 4, the server is preempted by T_2 . While it is preempted, it holds on to its budget.
5. After the server resumes execution at 5, its budget is consumed until exhaustion because first it executes (C_1) and then, when it is suspended again, T_1 and T_2 are idle (or equivalently, END , which is 5.0, is less than the current time) (C_2).
6. When the aperiodic job A_2 arrives at time 7, the budget of the server is exhausted; the job waits in the queue.
7. At time 8, its budget replenished (R_3), the server is ready for execution again.
8. At time 9.5, the server begins to execute for the first time since 8. te is equal to the latest replenishment time 8. Hence the next replenishment time is 13. The server executes until its budget is exhausted at 11; it is suspended and waits for the next replenishment time. In the meantime, A_2 waits in the queue.
9. Its budget replenished at time 13, the server is again scheduled and begins to execute at time 13.5. This time, the next replenishment time is set at 18. However at 13.5, the periodic task system T becomes idle. Rather than 18, the budget is replenished at 15, when a new busy interval of T begins, according to rule R_{3b} .
10. The behavior of the later segment also obeys the above stated rules. In particular, rule R_{3b} allows the server budget to be replenished at 19.

CONSTANT UTILIZATION, TOTAL BANDWIDTH, AND WEIGHTED FAIR-QUEUEING SERVERS

These three bandwidth preserving server algorithms that offer a simple way to schedule aperiodic jobs in deadline-driven systems. They are constant utilization, total bandwidth, and weighted fair-queueing algorithms. These algorithms belong to a class of algorithms that more or less emulate the Generalized Processor Sharing (GPS) algorithm. GPS, sometimes called fluid-flow processor sharing, is an idealized weighted round-robin algorithm; it gives each backlogged server in each round an infinitesimally small time slice of length proportional to the server size. Clearly, infinitesimally fine-grain time slicing cannot be implemented in practice

In particular, each server maintained and scheduled according to any of these algorithms offers timing isolation to the task(s) executed by it; by this statement, we mean that the worst-case response times of jobs in the task are independent the processor-time demands of tasks executed by other servers. While such a server works in a way that is similar to sporadic servers, its

correctness relies on a different principle. The correctness of these algorithms follows from the condition a schedulability condition of sporadic jobs scheduled on the EDF.

Schedulability of Sporadic Jobs in Deadline-Driven Systems

The *density* of a sporadic job J_i that has release time r_i , maximum execution time e_i and deadline d_i is the ratio $e_i / (d_i - r_i)$. A sporadic job is said to be *active* in its feasible interval $(r_i, d_i]$; it is not active outside of this interval. (Note : see on book for detail)

Theorem:

A system of independent, preemptable sporadic jobs is schedulable according to the EDF algorithm if the total density of all active jobs in the system is no greater than 1 at all times.

Corollary:

A system of n independent, preemptable sporadic tasks, which is such that the relative deadline of every job is equal to its period, is schedulable on a processor according to the EDF algorithm if the total instantaneous utilization (i.e

$$\sum_{i=1}^n \tilde{u}_i$$

), is equal to or less than 1.

COROLLARY:

A system of independent, preemptable, periodic and sporadic tasks, which is such that the relative deadline of every job is equal to its period, is schedulable on a processor according to the EDF algorithm if the sum of the total utilization of the periodic tasks and the total instantaneous utilization of the sporadic tasks is equal to or less than 1.

Constant Utilization server

We now return our attention to the problem of scheduling aperiodic jobs amid periodic tasks in a deadline-driven system. For the purpose of executing aperiodic jobs, there is a basic *constant utilization server*. The server is defined by its *size*, which is its instantaneous utilization ; this fraction of processor time is reserved for the execution of aperiodic jobs. As with deferrable

servers, the deadline d of a constant utilization server is always defined. It also has an execution budget which is replenished according to the replenishment rules described below.

The server is eligible and ready for execution only when its budget is nonzero. When the server is ready, it is scheduled with the periodic tasks on the EDF basis. While a sporadic server emulates a periodic task, a constant utilization server emulates a sporadic task with a constant instantaneous utilization, and hence its name.

Consumption Rule:

The consumption rule of a constant utilization server, as well as that of a total bandwidth or weighted fair-queueing server, is quite simple. *A server consumes its budget only when it executes.*

Replenishment Rules:

u_s is the size of the server, e_s is its budget, and d is its deadline. t denotes the current time, and e denotes the execution time of the job at the head the aperiodic job queue. The job at the head of the queue is removed when it completes. The rules assume that the execution time e of each aperiodic job becomes known when the job arrives.

Replenishment Rules of a Constant Utilization Server of Size \bar{u}_s

R1 Initially, $e_s = 0$, and $d = 0$.

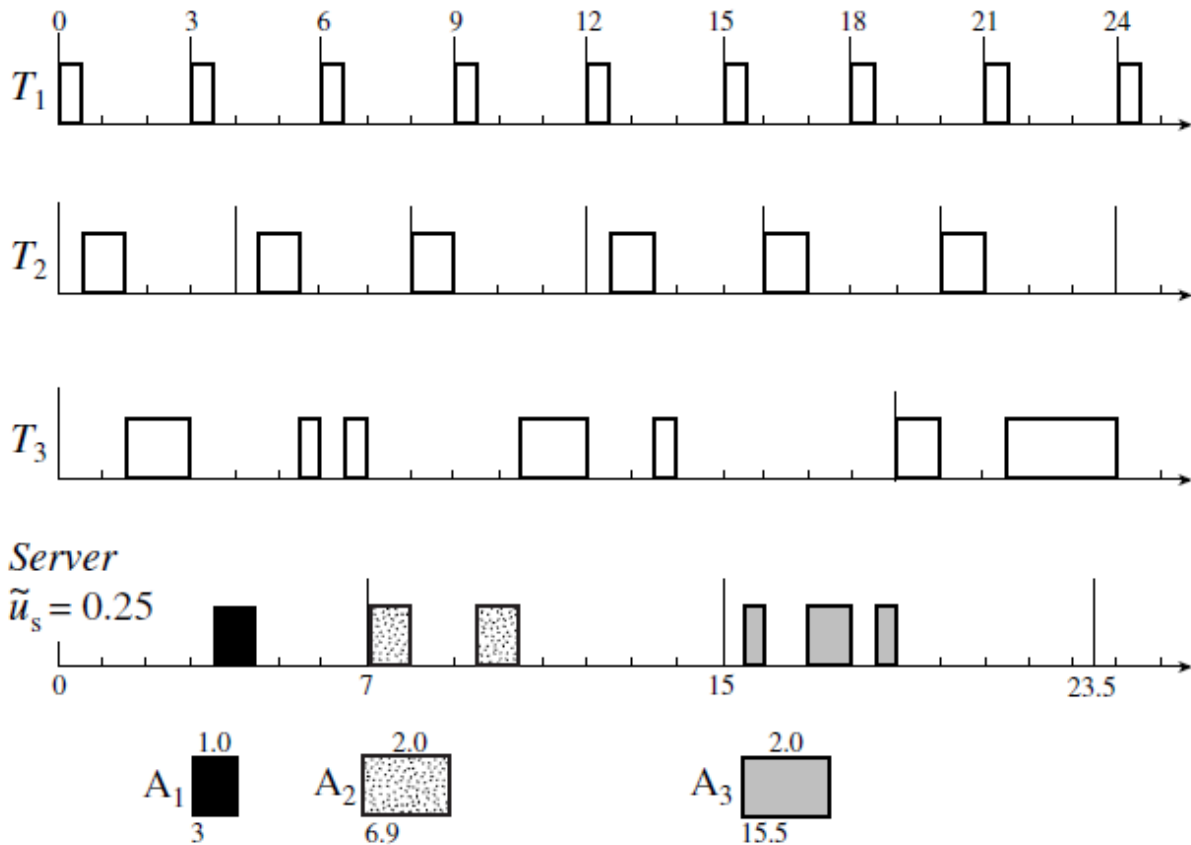
R2 When an aperiodic job with execution time e arrives at time t to an empty aperiodic job queue,

- (a) if $t < d$, do nothing;
- (b) if $t \geq d$, $d = t + e/\bar{u}_s$, and $e_s = e$.

R3 At the deadline d of the server,

- (a) if the server is backlogged, set the server deadline to $d + e/\bar{u}_s$ and $e_s = e$;
- (b) if the server is idle, do nothing.

1. Before time 3.0, the budget of the server is 0. Its deadline is 0. The server does not affect other tasks because it is suspended.
2. At time 3, A_1 arrives. The budget of the server is set to 1.0, the execution time of A_1 , and its deadline is $3 + 1.0/0.25 = 7$ according to R2b. The server is ready for execution. It completes A_1 at time 4.5.
3. When A_2 arrives at time 6.9, the deadline of the server is later than the current time. According to R2a, nothing is done except putting A_2 in the aperiodic job queue.
4. At the next deadline of the server at 7, the aperiodic job queue is checked and A_2 is found waiting. The budget of the server is replenished to 2.0, the execution time of A_2 , and its deadline is $7 + 2.0/0.25 = 15$. The server is scheduled and executes at time 7, is preempted by T_2 at time 8, resumes execution at 9.5 and completes A_2 at time 10.5.
5. At time 15, the aperiodic job queue is found empty. Nothing is done.



TOTAL BANDWIDTH SERVER ALGORITHM

Specifically, the total bandwidth server algorithm improves the responsiveness of a constant utilization server by allowing the server to claim the background time not used by periodic tasks. This is done by having the scheduler replenish the server budget as soon as the budget is exhausted if the server is backlogged at the time or as soon as the server becomes backlogged. We now show that a constant utilization server works correctly if its budget is replenished in this aggressive manner. In particular, we can change the replenishment rules as follows and get a *total bandwidth server*. You can see that the rules of a total bandwidth server are even simpler than the rules of a constant utilization server.

Replenishment Rules of a Total Bandwidth Server of size \bar{u}_s

- R1 Initially, $e_s = 0$ and $d = 0$.
- R2 When an aperiodic job with execution time e arrives at time t to an empty aperiodic job queue, set d to $\max(d, t) + e/\bar{u}_s$ and $e_s = e$.
- R3 When the server completes the current aperiodic job, the job is removed from its queue.
 - (a) If the server is backlogged, the server deadline is set to $d + e/\bar{u}_s$, and $e_s = e$.
 - (b) If the server is idle, do nothing.

Example: same as the example of constant utilization server algorithm

SLACK STEALING IN DEADLINE DRIVEN SYSTEM

We know that slack-stealing algorithms for deadline-driven systems are conceptually simpler than slack stealing algorithms for fixed-priority systems. For this reason, we first focus on systems where periodic tasks are scheduled according to the EDF algorithm. In this part aperiodic jobs are executed by a *slack stealer*. The slack stealer is ready for execution whenever the aperiodic job queue is nonempty and is suspended when the queue is empty. The scheduler monitors the periodic tasks in order to keep track of the amount of available slack. It gives the slack stealer the highest priority whenever there is slack and the lowest priority whenever there is no slack. When the slack stealer executes, it executes the aperiodic job at the head of the aperiodic job queue. This kind of slack-stealing algorithm is said to be *greedy*: The available slack is always used if there is an aperiodic job ready to be executed.

As an example, we consider the system of two periodic tasks, $T1 = (2.0, 3.5, 1.5)$ and $T2 = (6.5, 0.5)$. Suppose that in addition to the aperiodic job that has execution time 1.7 and is released at 2.8, another aperiodic job with execution time 2.5 is released at time 5.5. We call these jobs A1 and A2, respectively. Figure 7–15 shows the operation of a slack stealer.

1. Initially, the slack stealer is suspended because the aperiodic job queue is empty. When A1 arrives at 2.8, the slack stealer resumes. Because the execution of the last 0.7 units of J1,1 can be

postponed until time 4.8 (i.e., $5.5 - 0.7$) and T_2 has no ready job at the time, the system has 2 units of slack. The slack stealer is given the highest priority. It preempts $J_{1,1}$ and starts to execute A_1 . As it executes, the slack of the system is consumed at the rate of 1 per unit time.

2. At time 4.5, A_1 completes. The slack stealer is suspended. The job $J_{1,1}$ resumes and executes to completion on time.

3. At time 5.5, A_2 arrives, and the slack stealer becomes ready again. At this time, the execution of the second job $J_{1,2}$ of T_1 can be postponed until time 7.5, and the second job $J_{2,2}$ of T_2 can be postponed until 12.5. Hence, the system as a whole has 2.0 units of slack. The slack stealer has the highest priority starting from this time. It executes A_2 .

4. At time 7.5, all the slack consumed, the slack stealer is given the lowest priority. $J_{1,2}$ preempts the slack stealer and starts to execute.

5. At time 9, $J_{1,2}$ completes, and the system again has slack. The slack stealer now has the highest priority. It continues to execute A_2 .

6. When A_2 completes, the slack stealer is suspended again. For as long as there is no job in the aperiodic job queue, the periodic tasks execute on the EDF basis.

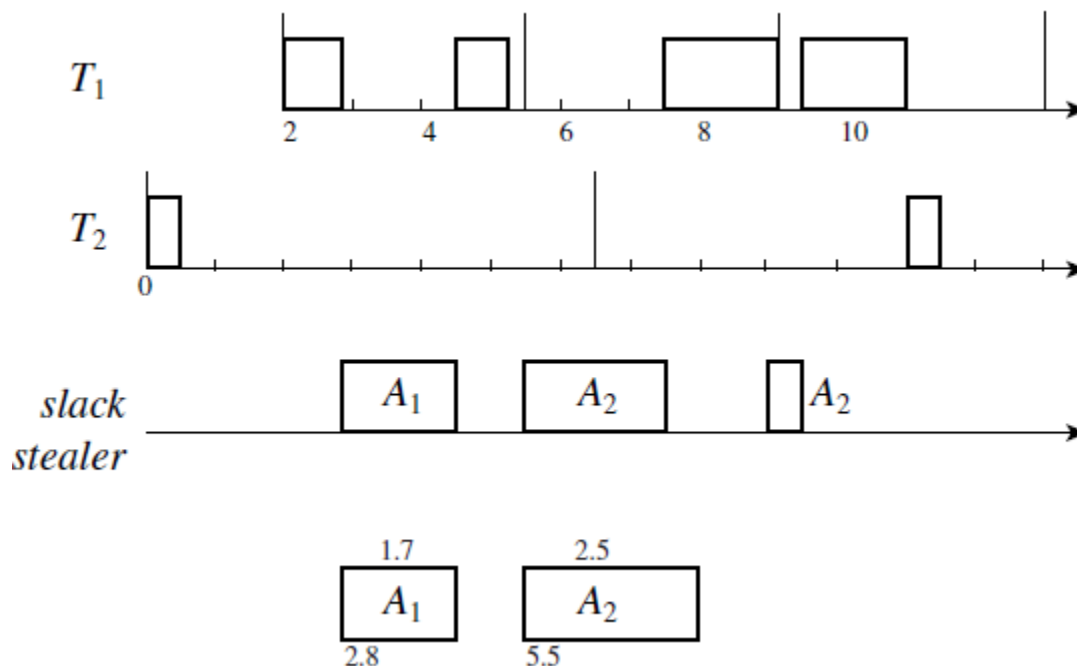


Fig Operation of stack stealer in deadline driven system

SLACK STEALING IN FIXED-PRIORITY SYSTEMS

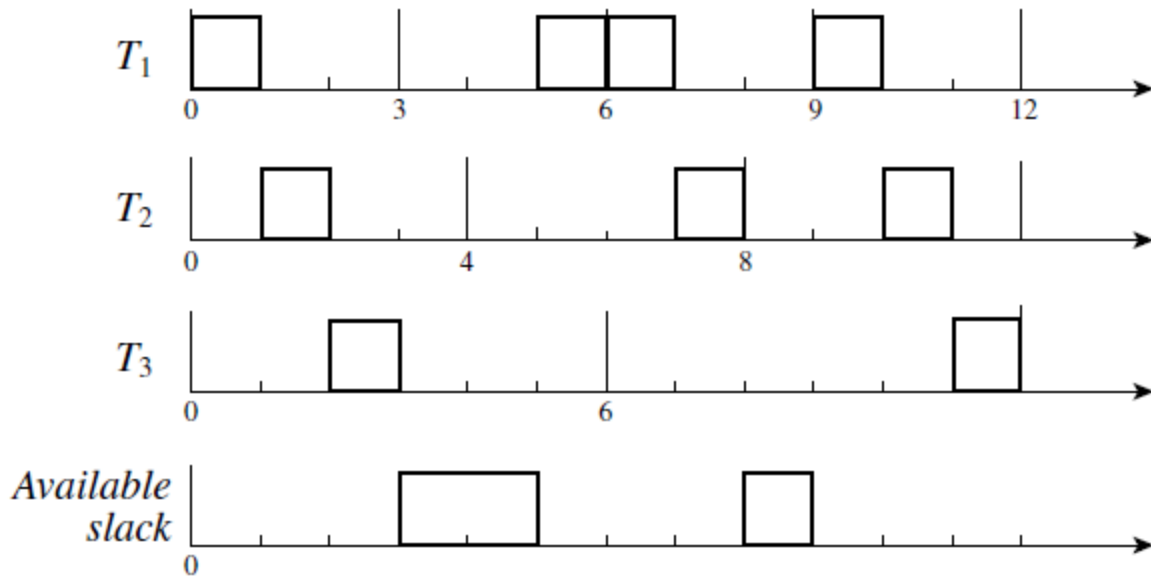
In principle, slack stealing in a fixed-priority system works in the same way as slack stealing in a deadline-driven system. However, both the computation and the usage of the slack are more complicated in fixed-priority systems.

Optimality Criterion and Design Consideration

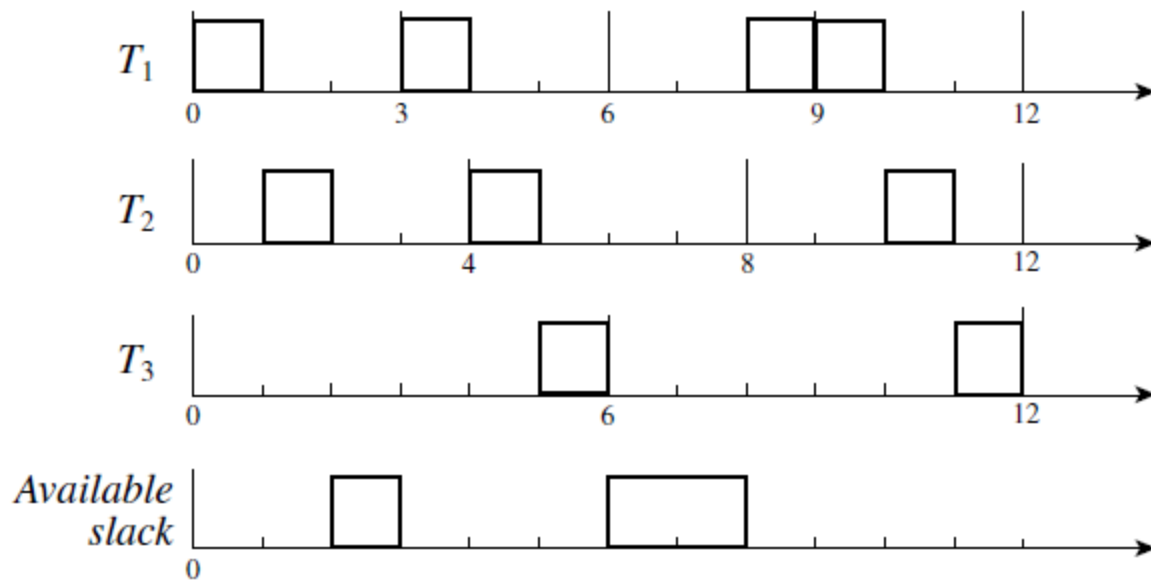
The system contains three periodic tasks: $T1 = (3, 1)$, $T2 = (4, 1)$, and $T3 = (6, 1)$. They are scheduled rate monotonically. If the system were deadline-driven, it would have 2 units of slack in the interval $(0, 3]$, but this system has only 1 unit. The reason is that once $J1,2$ becomes ready, $J2,1$ must wait for it to complete. As a consequence, $J2,1$ must complete by time 3, although its deadline is 4. In essence, 3 is the effective deadline of $J2,1$, and its slack is determined by the effective deadline. Figure (a) shows the schedule for the case when the 1 unit of slack is not used before time 3. At time 3, $J3,1$ has already completed. $J1,2$ and $J2,2$ can start as late as time 5 and 7, respectively, and still complete in time. Therefore, the system has two units of slack at time 3. Figure (b) shows the schedule for the other case: The 1 unit of slack is used before time 3. $J3,1$ is not yet complete at time 3. Consequently, $J1,2$ and $J2,2$ must execute immediately after they are released, even though their deadlines are 6 and 8; otherwise, $J3,1$ cannot complete in time. Under this condition, the system has no more slack until time 6.

This example points out the following important facts. These facts provide the rationales for the slack-stealing algorithm described below.

1. *No slack-stealing algorithm can minimize the response time of every aperiodic job in a fixed-priority system even when prior knowledge on the arrival times and execution times of aperiodic jobs are available.*
2. *The amount of slack a fixed-priority system has in a time interval may depend on when the slack is used. To minimize the response time of an aperiodic job, the decision on when to schedule the job must take into account the execution time of the job.*



(a)



(b)

Fig: Slack Stealing in fixed priority system

REAL-TIME PERFORMANCE FOR JOBS WITH SOFT TIMING CONSTRAINTS

For many applications, occasional missed deadlines are acceptable; their sporadic jobs have soft deadlines by a sporadic job, we mean one whose deadline is soft.

Traffic Models

Each sporadic task is a stream of sporadic jobs that have the same inter-release time and execution-time distributions and the same real-time performance requirements. The real-time performance experienced by each sporadic task is typically measured in terms of such criteria as the maximum tardiness and miss rate of jobs in it. Once a sporadic task is admitted into the system, the scheduler accepts and schedules every job in it. Specifically, when requesting admission into the system, each sporadic task presents to the scheduler its *traffic parameters*. These parameters define the constraints on the inter arrival times and execution times of jobs in the task. The performance guarantee provided by the system to each task is *conditional*, meaning that the system delivers the guaranteed performance conditioned on the fact that the task meets the constraints defined by its traffic parameters. Different traffic models use different traffic parameters to specify the behavior of a sporadic task.

FeVe and (λ, β) Models. According to the FeVe model, each sporadic task is characterized by a 4-tuple (e, p, \bar{p}, I) : e is the maximum execution time of all jobs in the task; p is the minimum interarrival time of the jobs; \bar{p} is their average interarrival time, and this average is taken over a time interval of length I . The (λ, β) model, characterizes each sporadic task by a rate parameter λ and a burst parameter β . The total execution time of all jobs that are released in any time interval of length x is no more than $\beta + \lambda x$.

Leaky Bucket Model. To define the *leaky bucket model*, we first introduce the notion of a (λ, E) leaky bucket filter. Such a filter is specified by its (input) rate λ and size E : The filter can hold at most E tokens at any time and it is being filled with tokens at a constant rate of λ tokens per unit time. A token is lost if it arrives at the filter when the filter already contains E tokens. We can think of a sporadic task that meets the (λ, E) leaky bucket constraint as if its jobs were generated by the filter in the following manner.

A TWO-LEVEL SCHEME FOR INTEGRATED SCHEDULING

A two-level scheduling scheme that provides timing isolation to individual applications executed on one processor. Each application contains an arbitrary number and types of tasks. By design, the two-level scheme allows different applications to use different scheduling algorithms (e.g., some may be scheduled in a clock-driven manner while the others in a priority-driven manner). Hence, each application can be scheduled in a way best for the application.

Overview and Terminology

According to the two-level scheme, each application is executed by a server. The scheduler at the lower level is called the *OS scheduler*. It replenishes the budget and sets the deadline of each server in the manners described below and schedules the ready servers on the EDF basis. At the higher level, each server has a *server scheduler*; this scheduler schedules the jobs in the application executed by the server according to the algorithm chosen for the application.

Required Capability. In the description below, we use \mathbf{T}_i for $i = 1, 2, \dots, n$ to denote n real time applications on a processor; each of these applications is executed by a server. To determine the schedulability and performance of each application \mathbf{T}_i , we examine the tasks in it as if the application executes alone on a slower processor whose speed is a fraction s of the speed of the physical processor. In other words, we multiple the execution time of every job in the application by a factor $1/s > 1$ and use the product in place of the execution time in the schedulability test on the application.

For example, the required capacity of an application that contains two periodic tasks $(2, 0.5)$ and $(5, 1)$ is 0.5 if it is scheduled rate-monotonically. The reason is that we can multiple the execution time of each task by 2 and the resultant tasks $(2, 1.0)$ and $(5, 2)$ are schedulable, but if the multiplication factor were bigger, the resultant tasks would not be schedulable. If these tasks are scheduled on the EDF basis, its required capacity is 0.45.

Predictable versus Nonpredictable Applications. In order to correctly maintain the server of an application that is scheduled according to a preemptive priority-driven algorithm, the OS scheduler needs an estimate of the occurrence time of every event of the application that may trigger a context switch within the application. Such events include the releases and completions of jobs and their requests and releases of resources. At any time t , the *next event* of application \mathbf{T}_i is the one that would have the earliest occurrence time after t among all events of \mathbf{T}_i if the application were to execute alone on a slow processor with speed s_i equal to its required capacity. We call an application that is scheduled according to a preemptive, priority-driven algorithm and contains aperiodic and sporadic tasks and/or periodic tasks with release-time jitters an *unpredictable application*. The reason for this name is that the OS scheduler needs an estimate of its next event (occurrence) time at each replenishment time of its server, but its server scheduler cannot compute an accurate estimate. All other types of applications are *predictable*.

CHAPTER-8

RESOURCES AND RESOURCE ACCESS CONTROL

ASSUMPTIONS ON RESOURCES AND THEIR USAGE:

- System contains only one processor
 - ☐ System contains ρ types of serially reusable resources $R1, R2, \dots, R\rho$.
 - ☐ There are v_i (v = “upsilon”) indistinguishable units of a resource of type R_i .
 - ☐ plentiful resources are ignored
 - ☐ binary semaphore has one unit
 - ☐ counting semaphore has n units.
- Serially reusable resources are allocated to jobs on a *non-preemptive* basis and used in a *mutually exclusive* manner
 - ☐ If a resource can be used by more than one jobs at the same time, this is modeled as a resource with several units, each used in a mutual exclusive manner.

Enforcement of Mutual Exclusion

- Mutual Exclusion is a lock-based concurrency control mechanism assumed to be used to enforce mutual exclusive access to resources.
- when a job wants to use v_i units of a resource R_i , it executes a *lock* $L(R_i, \eta_i)$ (η = “eta”) to request them.
- When the job no longer needs the resources, it releases them by executing an *unlock* $U(R_i, \eta_i)$.
- When a lock request fails, the requesting job is blocked and loses the processor
- It stays blocked until the scheduler grants the resources the job is waiting for.
- If a resource has only 1 unit the simpler notations $L(R_i)$ and $U(R_i)$ are used for lock and unlock resp.

Critical Section

- A segment of a job that begins with a lock and ends at a matching unlock is called a *critical section*.
- Resources are released in last-in-first-out order.
- A critical section that is not included in other critical sections is called an *outermost critical section*.
- Critical sections are denoted by $[R, \eta; e]$, where R gives the name and η the number of units of a resource and e the (maximum) execution time of the critical section.
- If there is only one unit of a resource the simpler notation $[R; e]$ is used.

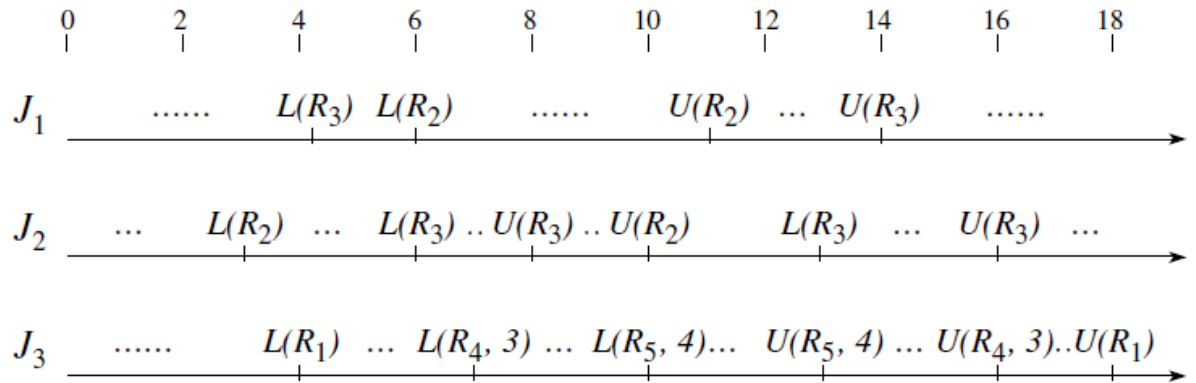


Fig. Example of Critical Section

As an example, above fig shows three jobs, J_1 , J_2 , and J_3 , and the time instants when locks and unlocks are executed if each job executes alone starting from time 0. Resources R_1 , R_2 , and R_3 have only 1 unit each, while resources R_4 and R_5 have many units. Job J_3 has three overlapping critical sections that are properly nested. A critical section that is not included in other critical sections is an *outermost critical section*; the critical section delimited by $L(R_1)$ and $U(R_1)$ in J_3 is an example. Other examples are the critical sections delimited by $L(R_2)$ and $U(R_2)$ in J_2 , the second pair of $L(R_3)$ and $U(R_3)$ in J_2 and $L(R_3)$ and $U(R_3)$ in J_1 . the critical section in J_3 that begins at $L(R_5, 4)$ is $[R_5, 4; 3]$ because in this critical section, the job uses 4 units of R_5 and the execution time of this critical section is 3.

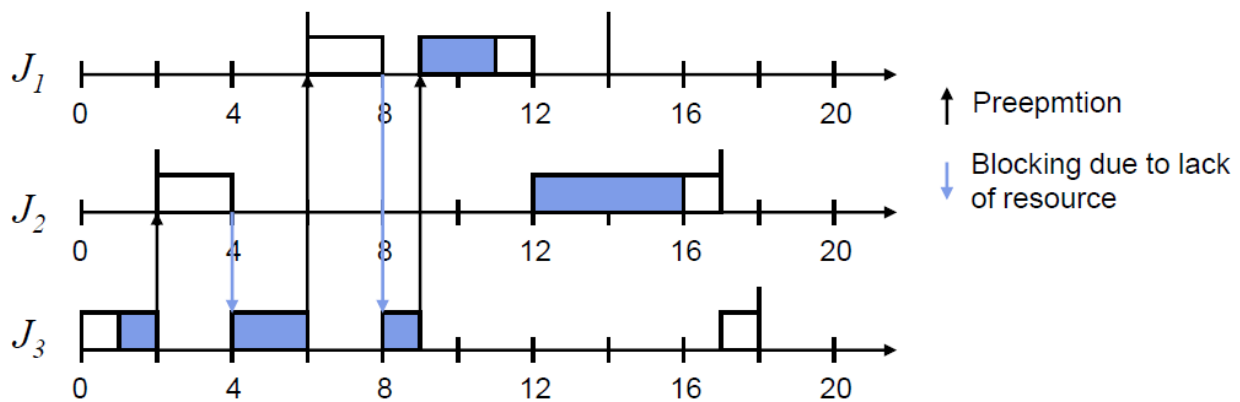
Resource Conflicts

- Two jobs *conflict* with each other, if some of the resources they require are of the same type.
- They *contend* for a resource when one job requests a resource that the other job already has.
- These terms are used interchangeably.

□ Jobs J_1 , J_2 , and J_3 with feasible intervals $(6, 14]$, $(2, 17]$, $(0, 18]$

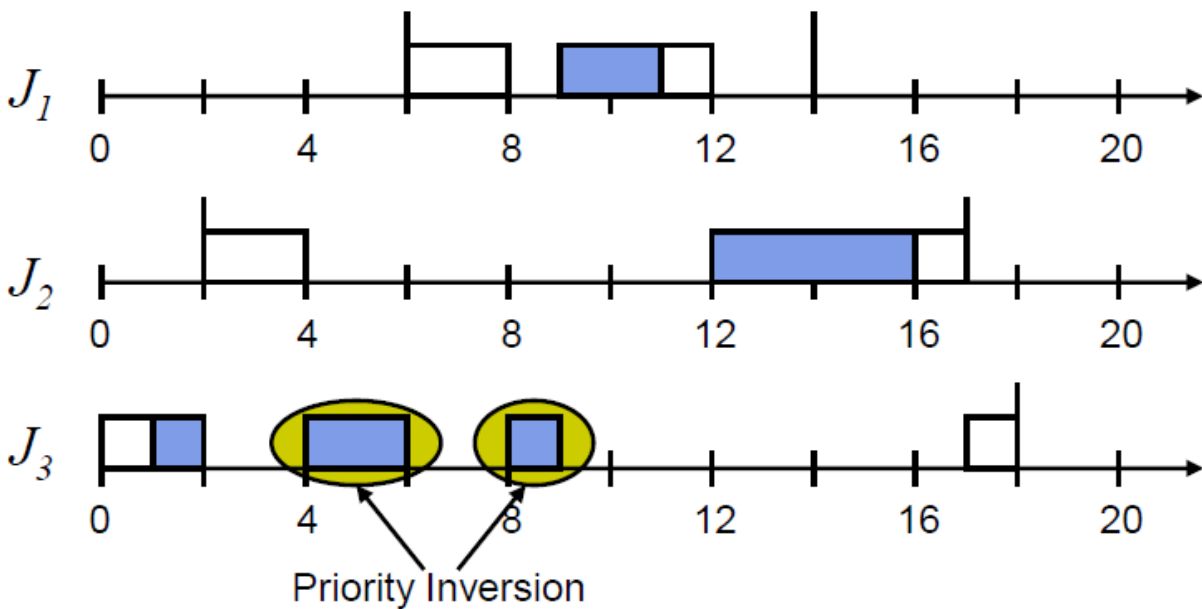
□ Critical sections: $[R; 2]$, $[R; 4]$, $[R; 4]$ for jobs 1, 2, 3

□ EDF-schedule.



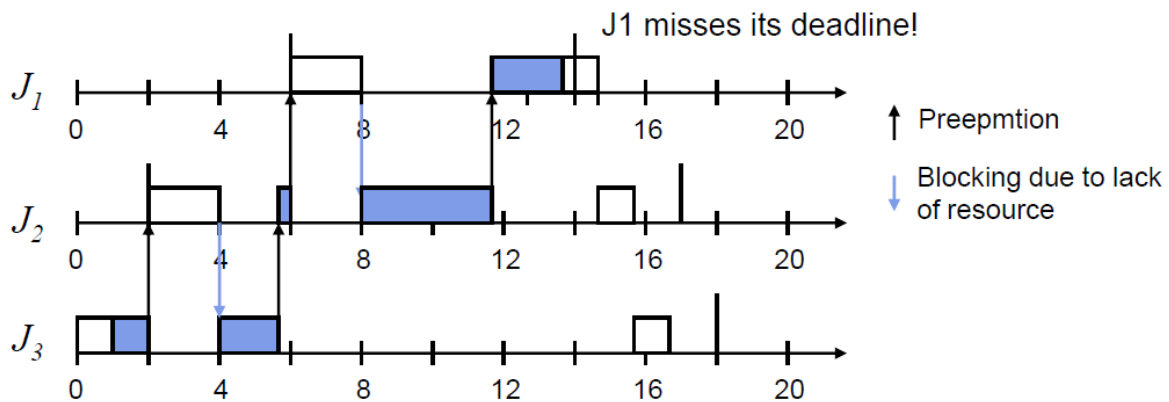
Priority Inversion:

Priority inversion occurs, when a low-priority job executes while a ready higher-priority job waits.



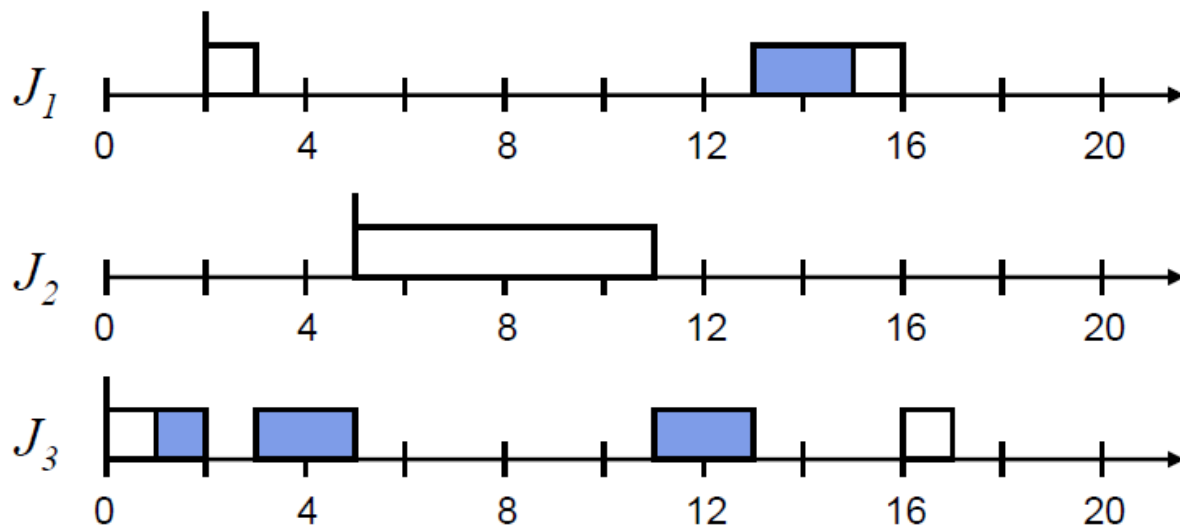
Timing Anomalies

- Timing Anomalies can occur due to priority inversion
- Assume that critical section of job 3 is reduced to $[R; 2.5]$ instead of $[R; 4]$



Resource Conflicts:

In this example J_1 is blocked by J_3 . But J_3 is preempted by J_2 . Thus J_1 has to wait for the lower-priority job J_2 that does not even require the resource that J_1 needs. In this case priority inversion is considered *uncontrolled*.



Need for Protocols?

Protocols are needed to handle priority inversion in a controlled way and to avoid deadlock.

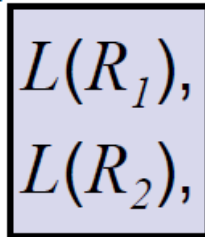
Deadlock:

Deadlock can occur, if jobs block each other from execution.

- **Example:**

- Job 1: $L(R_1), L(R_2), U(R_2), U(R_1)$

- Job 2: $L(R_2), L(R_1), U(R_1), U(R_2)$



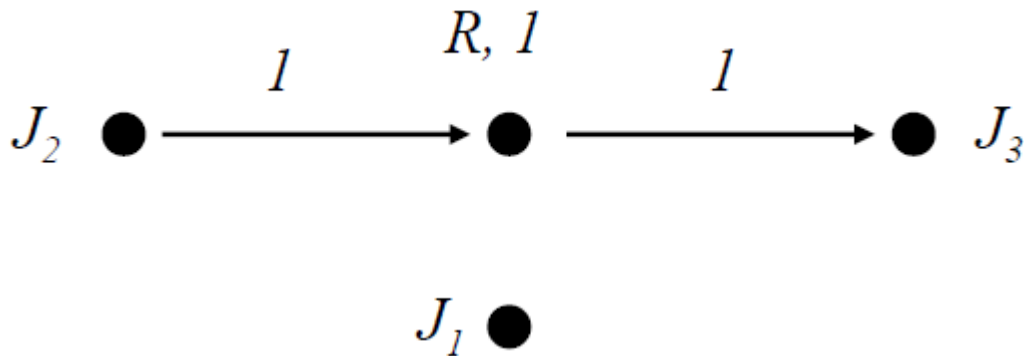
Job 1 and job 2 block each other

Wait-for-Graphs

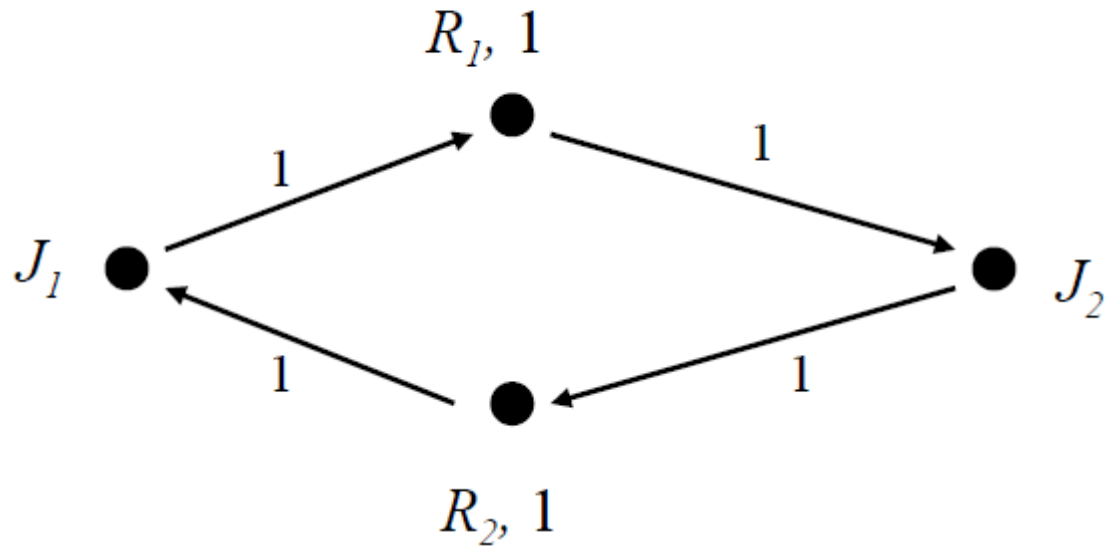
- Wait-for-Graphs are used to describe the dynamic-blocking relationship among jobs
- Vertexes represent jobs and resources.
- Edges can express that
 - a job owns a resource.
 - a job waits for a resource.
- A cyclic path in a wait-for-graph indicates a deadlock.

In the example:

- There is only one unit of resource R
- J2 waits for 1 unit of the resource of R
- J3 holds 1 unit of the resource R (or the resource waits for its release)
- J1 does neither hold or wait for resource.

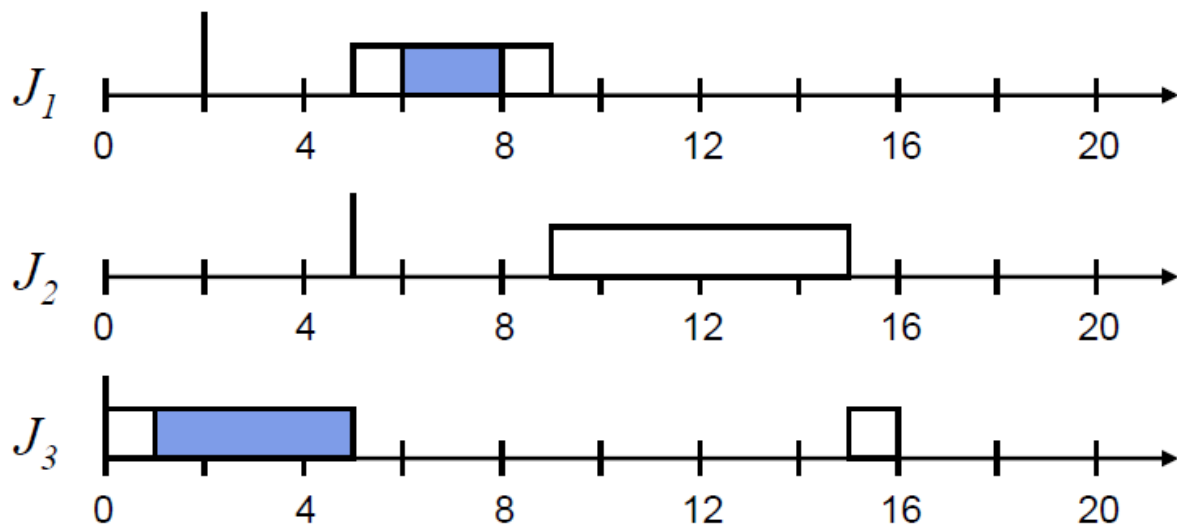


The following system is deadlocked:



Non-preemptive Critical Sections Protocol (NPCS):


- Simplest resource access control protocol .
- All critical sections are scheduled nonpreemptively.



Advantages:

- Simple to implement
- Uncontrolled priority inversion cannot occur
- A high-priority job can only be blocked once because of a low-priority job
- Good protocol when critical sections are short

□ The blocking time due to resource conflict is :

$$b_i(rc) = \max_{i+1 < k < n} (c_k)$$


Execution Time of critical section

Disadvantages:

□ Every job can be blocked by every lower-priority job even if there is no resource conflict between them.

Priority Inheritance Protocol:

- Works with any priority-driven scheduling algorithm.
- Uncontrolled priority inversion cannot occur.
- Protocol does not avoid deadlock.
- External mechanisms needed to avoid deadlock.

Assumption:

□ All resources have only one unit.

Definitions:

□ The priority of a job according to the scheduling algorithm is its *assigned priority*

□ At any time t , each ready job J_l is scheduled and executes at its *current priority* $\pi_l(t)$, which may differ from its assigned priority and vary with time.

Priority Inheritance

- The current priority $\pi_l(t)$ of a job J_l may be raised to the higher priority $\pi_h(t)$ of a job J_h
- When this happens, we say that the lower-priority job J_l *inherits* the priority of higher-priority job J_h , and that J_l executes at its *inherited priority* $\pi_h(t)$

Scheduling Rule

- Ready jobs are scheduled on the processor preemptively in a priority-driven manner according to their current priorities.
- At its release time t , the current priority $\pi(t)$ of every job J is equal to its assigned priority
- The job remains at this priority except under the condition stated in the *priority-inheritance rule*.

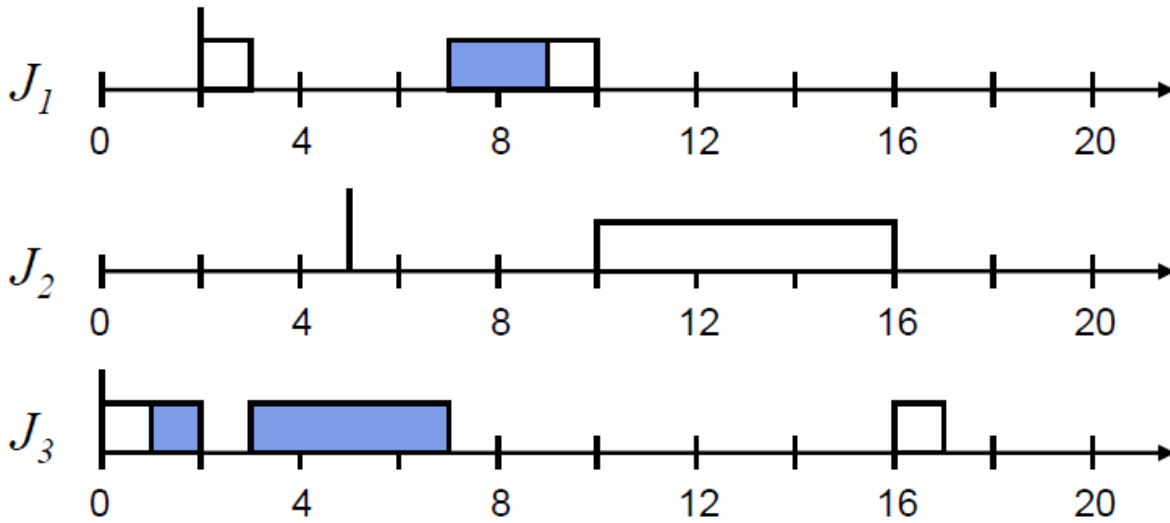
Allocation Rule

- When a job J requests a resource R at time t ,
 - a) if R is free, R is allocated to J until J releases the resource, and
 - b) if R is not free, the request is denied and J is blocked.

Priority-Inheritance Rule:

- When the requesting job J becomes blocked, the job J_l which blocks J inherits the current priority $\pi(t)$ of J .
- The job J_l executes at its inherited priority $\pi(t)$ until it releases R .
- At that time, the priority of J_l returns to its priority $\pi_l(t')$ at the time t' when it acquired the resource R .

Example:

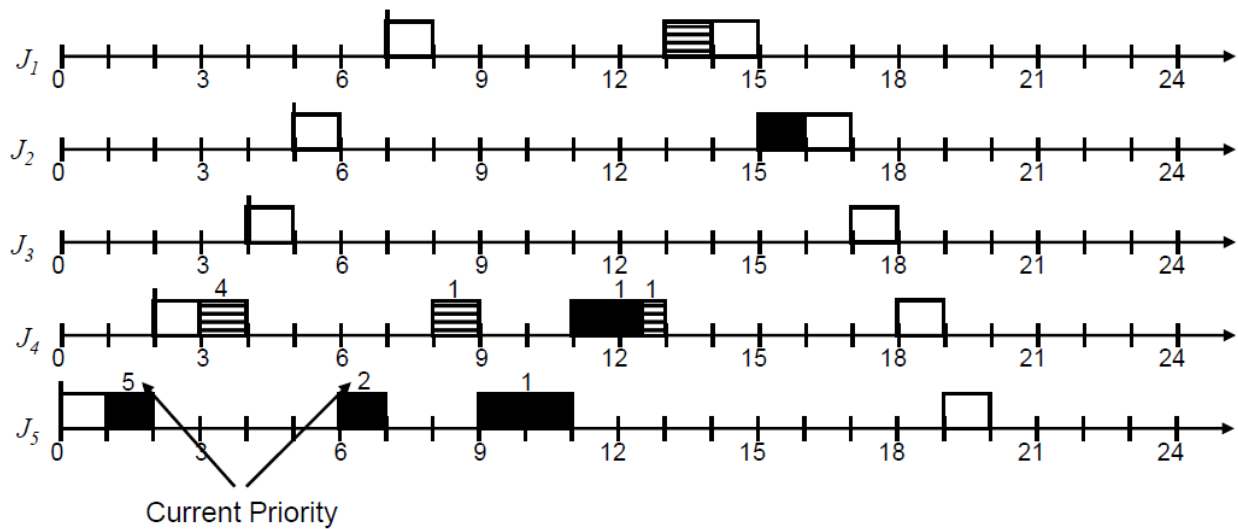


Description:

- When J_1 requests resource R and becomes blocked at time 3, job J_3 inherits the priority π_1 of job J_1 .
- When J_2 becomes ready at time 5, it cannot preempt J_3 because its priority π_2 is lower than the inherited priority π_1 of J_3 .
- As a consequence J_3 completes its critical section as soon as possible.

Next Example:

Job	r_i	e_i	π_i	Critical Sections
J_1	7	3	1	[<i>Shaded</i> ; 1]
J_2	5	3	2	[<i>Black</i> ; 1]
J_3	4	2	3	
J_4	2	6	4	[<i>Shaded</i> ; 2][<i>Black</i> ; 1.5] <i>Shaded</i> ; 0.5]
J_5	0	6	5	[<i>Black</i> ; 4]



Description:

1. At time 0, job J_5 becomes ready and executes at its assigned priority 5. At time 1, it is granted the resource *Black*.
2. At time 2, J_4 is released. It preempts J_5 and starts to execute.
3. At time 3, J_4 requests *Shaded*. *Shaded*, being free, is granted to the job. The job continues to execute.

4. At time 4, J_3 is released and preempts J_4 . At time 5, J_2 is released and preempts J_3 .
5. At time 6, J_2 executes $L(Black)$ to request $Black$; $L(Black)$ fails because $Black$ is in use by J_5 . J_2 is now directly blocked by J_5 . According to rule 3, J_5 inherits the priority 2 of J_2 . Because J_5 's priority is now the highest among all ready jobs, J_5 starts to execute.
6. J_1 is released at time 7. Having the highest priority 1, it preempts J_5 and starts to execute.
7. At time 8, J_1 executes $L(Shaded)$, which fails, and becomes blocked. Since J_4 has $Shaded$ at the time, it directly blocks J_1 and, consequently, inherits J_1 's priority 1. J_4 now has the highest priority among the ready jobs J_3 , J_4 , and J_5 . Therefore, it starts to execute.
8. At time 9, J_4 requests the resource $Black$ and becomes directly blocked by J_5 . At this time the current priority of J_4 is 1, the priority it has inherited from J_1 since time 8. Therefore, J_5 inherits priority 1 and begins to execute.
9. At time 11, J_5 releases the resource $Black$. Its priority returns to 5, which was its priority when it acquired $Black$. The job with the highest priority among all unblocked jobs is J_4 . Consequently, J_4 enters its inner critical section and proceeds to complete this and the outer critical section.
10. At time 13, J_4 releases $Shaded$. The job no longer holds any resource; its priority returns to 4, its assigned priority. J_1 becomes unblocked, acquires $Shaded$, and begins to execute.
11. At time 15, J_1 completes. J_2 is granted the resource $Black$ and is now the job with the highest priority. Consequently, it begins to execute.
12. At time 17, J_2 completes. Afterwards, jobs J_3 , J_4 , and J_5 execute in turn to completion.

Basic Priority Ceiling Protocol

Definitions

- The *priority ceiling* of any resource R_i is the highest priority of all jobs that require R and is denoted $\Pi(R_i)$.

- At any time t , the *current priority ceiling* $\Pi_C(t)$ of the system is equal to the highest priority ceiling of the resources that are in use at the time, if some resources are in use.
- otherwise it is Ω , a non-existing priority that is lower than the priority of any job.

Basic Priority-Ceiling Protocol:

Scheduling Rule

- As its release time t , the current priority $\pi(t)$ of each job J is equal to its assigned priority.
- The job remains at this priority except under the condition stated in the *Priority-Inheritance rule*.
- Every job J is scheduled preemptively and in a priority-driven manner at its current priority $\pi(t)$.

Allocation Rule:

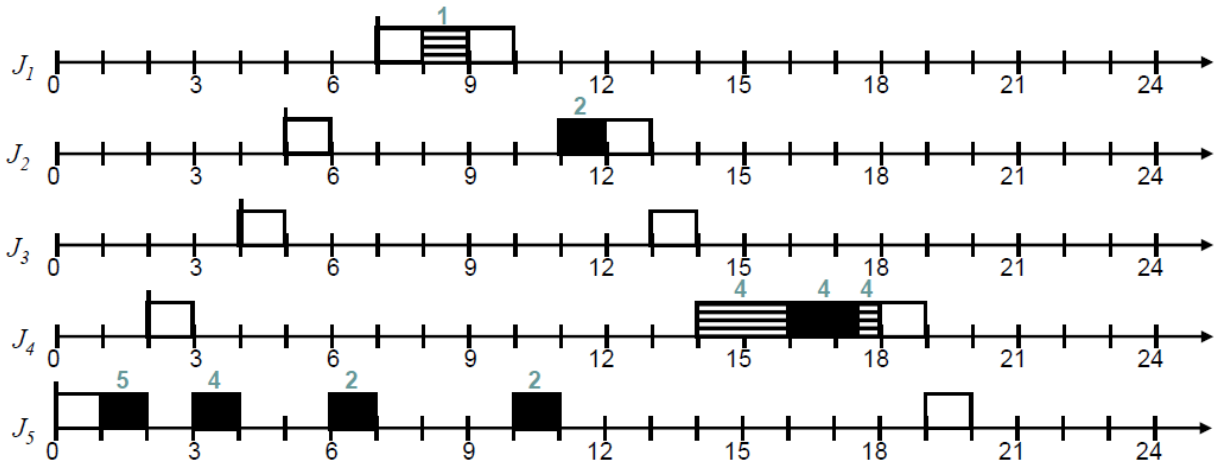
- When a job J requests a resource R at time t ,
 - a) R is held by another job, the request is denied and J is blocked.
 - b) R is free
 - i. If J 's current priority $\pi(t)$ is higher than the current priority ceiling of the system $\Pi_C(t)$, R is allocated to J
 - ii. If J 's current priority $\pi(t)$ is not higher than the current ceiling of the system $\Pi_C(t)$, R is allocated to J only if J is the job holding the resource(s) whose priority ceiling is equal to $\Pi_C(t)$; otherwise J 's request is denied and J becomes blocked.

Priority-Inheritance Rule:

- When J becomes blocked, the job J_l which blocks J inherits the current priority $\pi(t)$ of J .
- J_l executes at its inherited priority $\pi(t)$ until the time where it releases every resource whose priority ceiling is equal to or higher than $\pi(t)$.

□ At that time, the priority of J_l returns to its priority $\pi_l(t')$ at the time t' when it was granted the resource(s).

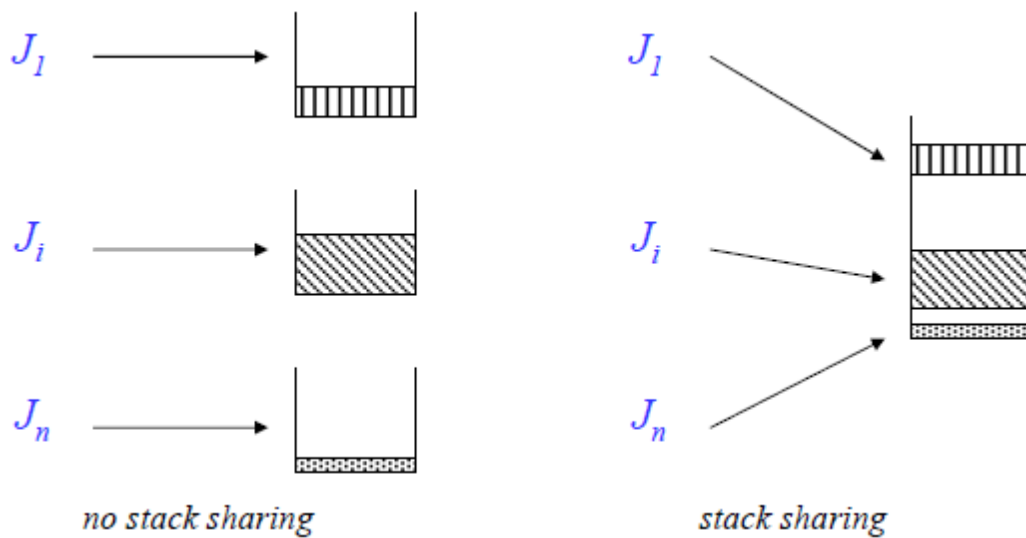
Example:



Description:(see book page no 291 for description).

Stack Sharing:

- Sharing of the stack among jobs eliminates stack space fragmentation and allows for memory savings:
- with no stack sharing, each single job needs some private memory space to allow for its stack growing;
- with stack sharing, there is just one single memory space for all stacks growing.



when a job J executes, its stack frame is on top of the stack;

- The stack frame is freed when J completes;
- When J is preempted, the preempting job has its stack frame on top of J 's frame;
- In a stack sharing system, a job J may resume only when all jobs holding stack space above its frame have completed and leave J 's frame on top of the stack again;
- In a stack sharing system, a job J must never self-suspend
 - in case this happened, no other previously active, lower priority job J_L could proceed, since its stack space would not be on top of the stack: only higher priority jobs could (preempt) and execute; J would keep the processor busy (at its own priority level) while self-suspended, which is the same as saying that in a stack sharing system jobs cannot self-suspend.
- In a stack sharing system, after a job J has begun execution, it must never be blocked because it is denied some resource assigned to a lower priority job J_L :
 - in case this happened, J could proceed only after the (previously active, lower priority) blocking job J_L has been resumed and has freed the requested resource;

but this would not be possible, because the stack space of this (previously active, lower priority) blocking job J_L would not be on top of the stack;

\Rightarrow this would cause deadlock (J needs R held by J_L ; J_L needs the stack held by J);

In a stack sharing system (assumptions):

- jobs may be preempted (by higher priority jobs),
- must never be blocked (by lower priority jobs),
- must never self-suspend.

Stack Based Priority-Ceiling Protocol:

Based on the stack sharing principle (for non self-suspending jobs).

- To avoid deadlocks: the protocol makes sure that once a job begins execution, it will not be blocked due to resource access.

- Define: $\Pi(t)$ = highest priority ceiling of all resources allocated at t . if no resource is allocated,

$\Pi(t) = \Omega$.

SBPC protocol:

1. Update Priority Ceiling: whenever all resources are free, $\Pi(t) = \Omega$. The value of $\Pi(t)$ is updated whenever a resource is allocated or freed.

2. Scheduling Rule: after a job is released, it is blocked from starting execution until its assigned priority is higher than $\Pi(t)$. Jobs that are not blocked are scheduled on the processor in a priority driven, preemptive fashion according to their assigned priorities.

3. Allocation Rule: whenever a job requests a resource, it is allocated the resource.

Example: All the parameters of jobs are same of above example except let J_2 be released at 4.8 and the execution time of the critical section of J_2 be 1.2. The schedule is as follows.

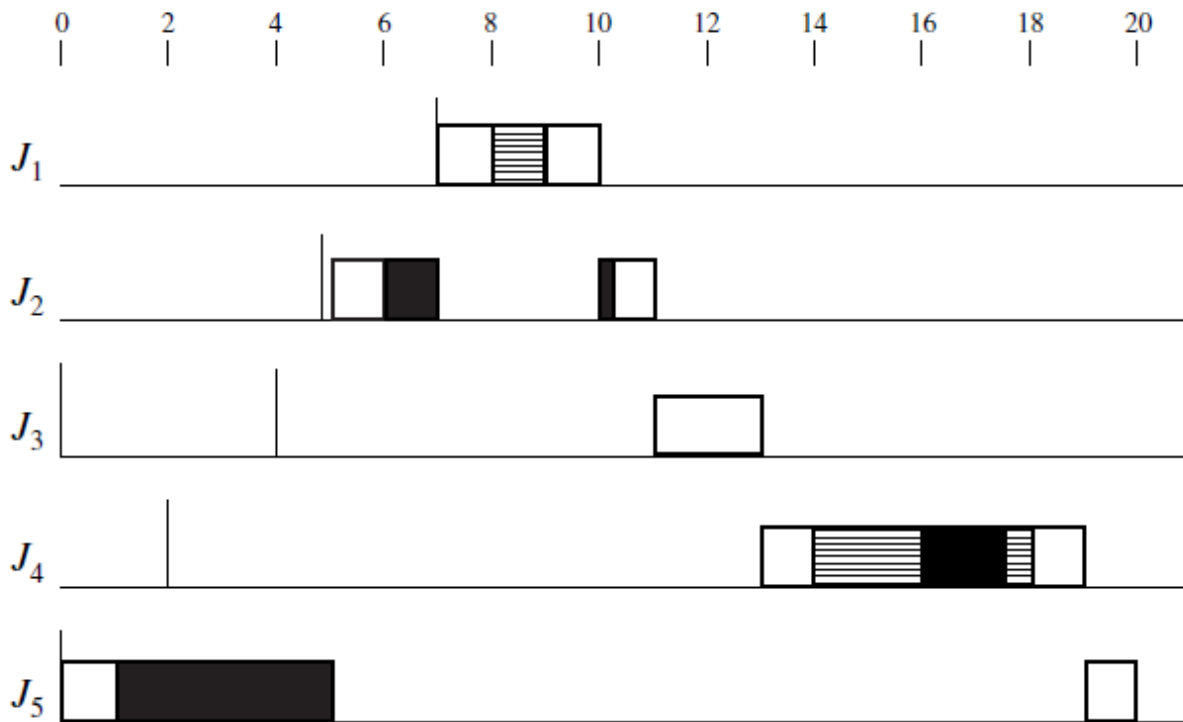


Fig:SBPC

USE OF PRIORITY-CEILING PROTOCOL IN DYNAMIC-PRIORITY SYSTEMS

While both versions of the priority-ceiling protocol are relatively simple to implement and perform well when periodic tasks are scheduled on a fixed-priority basis, it is another matter in a dynamic-priority system. In a dynamic-priority system, the priorities of the periodic tasks change with time while the resources required by each task remain constant. As a consequence, the priority ceilings of the resources may change with time.

As an example, let us look at the EDF schedule of two tasks $T1 = (2, 0.9)$ and $T2 = (5, 2.3)$ in Figure 6–4. In its first two periods (i.e., from time 0 to 4), $T1$ has priority 1 while $T2$ has priority 2, but from time 4 to 5, $T2$ has priority 1 and $T1$ has priority 2. Suppose that the task $T1$ requires a resource X while $T2$ does not. The priority ceiling of X is 1 from time 0 to 4 and becomes 2 from time 4 to 5, and so on.

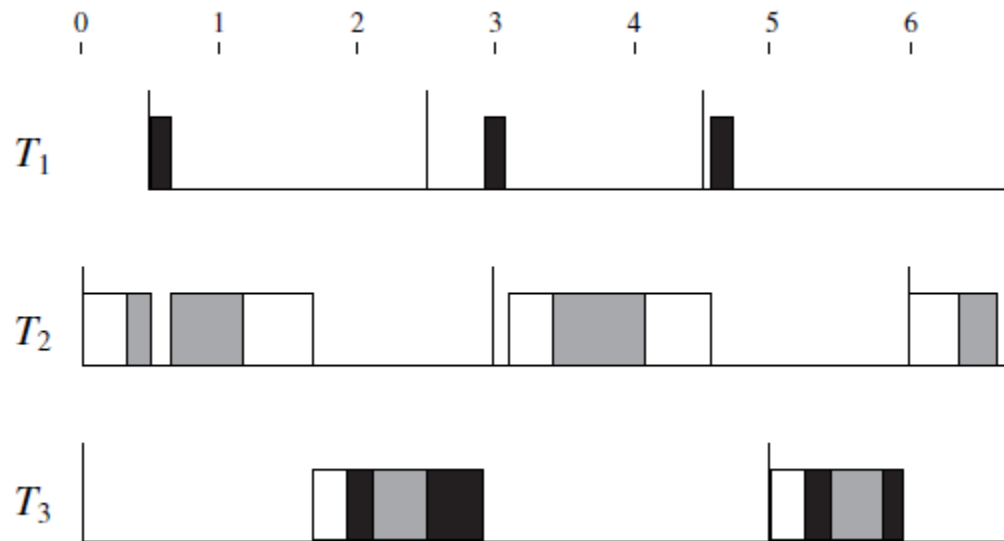
Implementation of Priority-Ceiling Protocol in Dynamic-Priority Systems

One way to implement the basic priority-ceiling protocol in a job-level fixed-priority system is to update the priority ceilings of all resources whenever a new job is released. Specifically, when a new job is released, its priority relative to all the jobs in the ready queue is assigned according to the given dynamic-priority algorithm. Then, the priority ceilings of all the resources are updated based on the new priorities of the tasks, and the ceiling of the system is updated based on the new priority ceilings of the resources. The new priority ceilings are used until they are updated again upon the next job release. The example in Figure 8–18 illustrates the use of this protocol in an EDF system. The system shown here has three tasks: $T1 = (0.5, 2.0, 0.2; [Black; 0.2])$, $T2 = (3.0, 1.5; [Shaded; 0.7])$, and $T3 = (5.0, 1.2; [Black; 1.0] [Shaded; 0.4])$. The priority ceilings of the two resources *Black* and *Shaded* are updated at times 0, 0.5, 2.5, 3, 4.5, 5, 6, and so on.

1. At time 0, there are only two ready jobs, $J2,1$ and $J3,1$. $J2,1$ (and hence $T2$) has priority 1 while $T3$ has priority 2, the priority of $J3,1$. The priority ceilings of *Black* and *Shaded* are 2 and 1, respectively. Since $J2,1$ has a higher priority, it begins to execute. Because no resource is in use, the ceiling of the system is . At time 0.3, $J2,1$ acquires *Shaded*, and the ceiling of the system rises from to 1, the priority ceiling of *Shaded*.
2. At time 0.5, $J1,1$ is released, and it has a higher priority than $J2,1$ and $J3,1$. Now the priorities of $T1$, $T2$, and $T3$ become 1, 2, and 3, respectively. The priority ceiling (*Black*) of *Black* is 1, the priority of $J1,1$ and $T1$. The priority ceiling $_t$ (*Shaded*) of *Shaded* becomes 2 because the priority of $J2,1$ and $T2$ is now 2. The ceiling of the system based on these updated values is 2. For this reason, $J1,1$ is granted the resource *Black*. The ceiling of the system is 1 until $J1,1$ releases *Black* and completes at time 0.7. Afterwards, $J2,1$ continues to execute, and the ceiling of the system is again 2. When $J2,1$ completes at time 1.7, $J3,1$ commences to execute and later acquires the resources as shown.
3. At time 2.5, $J1,2$ is released. It has priority 1, while $J3,1$ has priority 2. This update of task priorities leads to no change in priority ceilings of the resources. Since the ceiling of the system is at 1, $J1,2$ becomes blocked at 2.5. At time 2.9, $J3,1$ releases *Black*, and $J1,2$ commences execution.
4. At time 3.0, only $T1$ and $T2$ have jobs ready for execution. Their priorities are 1 and 2, respectively. The priority ceilings of the resources remain unchanged until time 4.5.
5. At time 4.5, the new job $J1,3$ of $T1$ has a later deadline than $J2,2$. (Again, $T3$ has no ready job.) Hence, the priority of $T1$ is 2 while the priority of $T2$ becomes 1. This change

in task priorities causes the priority ceilings of *Black* and *Shaded* to change to 2 and 1, respectively.

6. At time 5 when $J_{3,2}$ is released, it is the only job ready for execution at the time and hence has the highest priority. The priority ceilings of both resources are 1. These values remain until time 6.



CONTROLLING ACCESSES TO MULTIPLE-UNIT RESOURCES

Both versions of the priority-ceiling protocol and preemption-ceiling protocol described in the previous sections assume that there is only one unit of each resource. We now describe an extension to these protocols so that they can deal with the general case where there may be more than one unit of each resource (type).

Priority (Preemption) Ceilings of Multiple-Unit Resources

The first step in extending the priority-ceiling protocol is to modify the definition of the priority ceilings of resources. We let $_ (R_i, k)$, for $k \leq v_i$, denote the priority ceiling of a resource R_i when k out of the v_i (≥ 1) units of R_i are free. If one or more jobs in the system require more than k units of R_i , $_ (R_i, k)$ is the highest priority of all these jobs. If no job requires more than k units of R_i , $_ (R_i, k)$ is equal to $_$, the nonexistent lowest priority. In this notation, the priority ceiling $_ (R_j)$ of a resource R_j that has only 1 unit is $_ (R_j, 0)$.

CONTROLLING CONCURRENT ACCESSES TO DATA OBJECTS

Data objects are a special type of shared resources. When jobs are scheduled preemptively, their accesses to (i.e., reads and writes) data objects may be interleaved. To ensure data integrity, it is common to require that the reads and writes be serializable. A sequence of reads and writes by a set of jobs is *serializable* if the effect produced by the sequence on all the data objects shared by the jobs is the same as the effect produced by a serial sequence (i.e., the sequence of reads and writes when the jobs execute according to a nonpreemptive schedule).

Convex-Ceiling Protocol

The resource access-control protocols described in earlier sections do not ensure serializability. For example, both the NPCS and PC (Priority- and Preemption-Ceiling) protocols allow a higher-priority job J_h to read and write a data object X between two disjoint critical sections of a lower-priority job J_l during which J_l also reads and writes X . The value of X thus produced may not be the same as the value produced by either of the two possible serial sequences (i.e., all the reads and writes of J_l either proceed or follow that of J_h).

Motivation and Assumptions. A well-known way to ensure serializability is Two-Phase Locking (2PL). According to the 2PL protocol, a job never requests any lock once it releases some lock. We can easily get concurrency-control protocols that not only ensure serializability but also prevent deadlock and transitive blocking by augmenting the protocols. As a result, we have the NPCS-2PL and the PCP- 2PL protocols.

Priority-Ceiling Function. As with the PCP-2PL protocol, the convex-ceiling protocol assumes that the scheduler knows a priori the data objects required by each job and, therefore, the priority ceiling of each data object. In addition, each job notifies the scheduler immediately after it accesses each of its required objects for the last time. We call a notification sent by a job J_i after it accesses R_k for the last time the *last access notification* for R_k by J_i and the time of this notification the *last access time of R_k by J_i* . For example the job J_i requires three data objects: *Dotted*, *Black*, and *Shaded*. Their priority ceilings are 1, 2, and 3, respectively. Figure 8-23(a) shows the time intervals when the job executes and accesses the objects. The two functions of the job are shown in Figure. At time 0, $RP(J_i, 0)$ is 1. The job sends last access notifications at times 4, 6, and 8 when it no longer needs *Dotted*, *Black*, and *Shaded*, respectively. The scheduler updates $RP(J_i, t)$ at these instants; each time, it lowers the remainder priority ceiling to the highest priority ceiling of all objects still required by the job in the future.

