* Commonly used approaches to Real time scheduling *

* Clock driven approach,
* Weighted Round-Robin approach,
* priority driven approach.

* Clock driven approach *

when scheduling is clock-driven (also called time-driven), decisions on what jobs execute at what times are made at specific time instants. These instants are chosen a priori before the system begins execution.

Typically, in a system that uses clock-driven scheduling, all the parameters of hard real-time jobs are fixed and known. A schedule of the jobs is computed off-line and is stored for use at run-time.

The schedular schedules the jobs according to this schedule at each scheduling decision time. Thus the scheduling overhead during runtime can be minimized.

One way to implement a schedular that makes scheduling decisions periodically is to use a hardware time. The timer is set to expire periodically.

when the system is initialized, the schedular selects and schedule the job(s) that will execute until the next scheduling decision time.

* weighted Round-Robin approach *

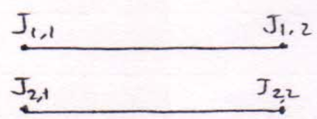
The round-robin approach is commonly used for scheduling time-shared applications. When jobs are scheduled on a round-robin basis, every Job joins a First-in-first-out (FIFO) queue when it becomes ready for execution.

The job at the of the queue executes for at most one time slice. If the job does not complete by the end of the time slice, it is preempted and placed at the end of the queue to wait for its next turn.

The weighted round-robin algorithm has been used for scheduling real-time traffic in high-speed switched networks. Rather than giving all the ready jobs equal shares of the processor, different jobs may be given different weights. Here, the weight of a job refers to the fraction of processor time allocated to the job. A job with weight wt get wt time slices every round, and the length of a round is equal to the sum of the weights of all the ready jobs. By adjusting the weights of jobs, we can speed up or retard the progess of each job toward its completion.

For precedence constrainted jobs the weighted round-robin approach is not suitable.

Example: we consider the two set of jobs, $J_1 = \{J_{1,1}, J_{1,2}\}$ and $J_2 = \{J_{2,1}, J_{2,2}\}$ shown in Fig.



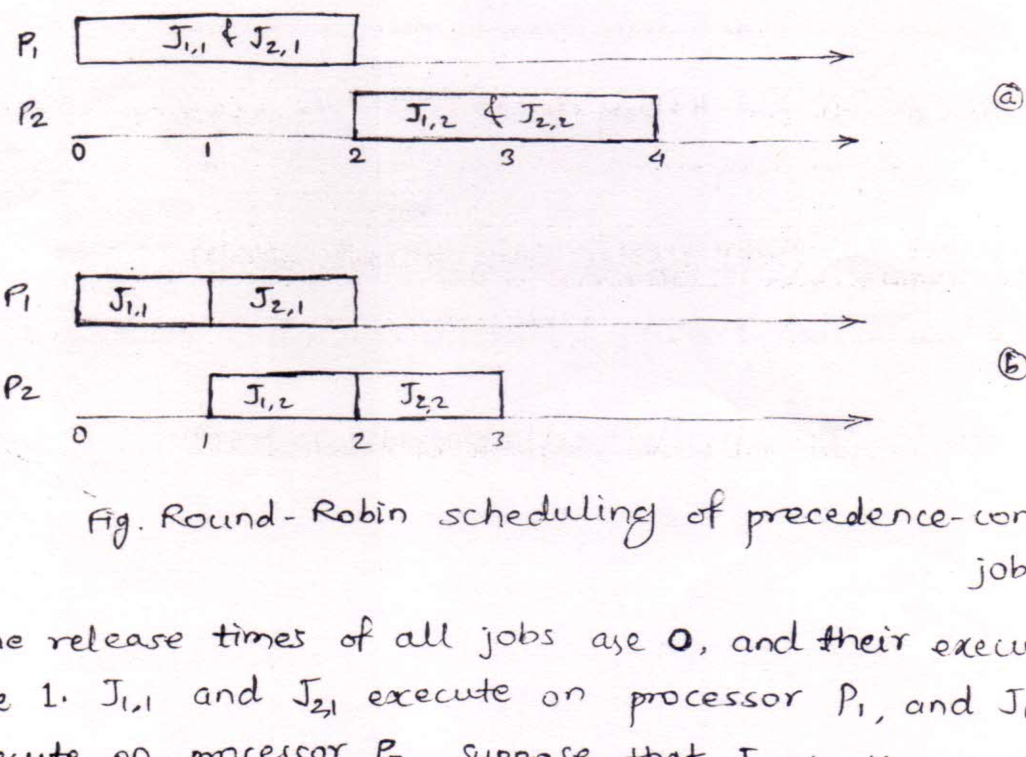


Fig. Round-Robin scheduling of precedence-constrained jobs

The release times of all jobs are 0, and their execution times are 1. $J_{1,1}$ and $J_{2,1}$ execute on processor P_1, and $J_{1,2}$ and $J_{2,2}$ execute on processor P_2. Suppose that $J_{1,1}$ is the predecessor of $J_{1,2}$ and $J_{2,1}$ is the predecessor of $J_{2,2}$.

Fig (a) shows that both sets of jobs complete approximately at time 4 if the jobs are scheduled in a weighted round-robin manner.

In contrast, the schedule in Fig (b) shows that if the jobs on each processor are executed one after the other, one of the chains can complete at time 2, while the other can complete at time 3.

* Priority Driven approach *

In this type of algorithms no algorithm never leave any resource idle intentionally.
A resource idles only when no job requiring the resource is ready for execution.
Scheduling decisions are made when events such as releases and completions of jobs occur. Hence, priority-driven algorithms are event-driven.
It is also called as Greedy scheduling, list scheduling and work-conserving scheduling.
It is greedy because it tries to make locally optimal. Leaving a resource idle while some job is ready to use the resource is not locally optimal.
Jobs ready for execution are placed in one or more queues ordered by the priorities of the jobs. At any scheduling decision time, the jobs with the highest priorities are scheduled and executed on the available processors.
It assigns priorities to jobs; the priority list and other rules, such as whether preemption is allowed, define the scheduling algorithm completely.
Most scheduling algorithms used in Real-time systems are priority-driven.

Examples: FIFO (First-In-First-Out) and LIFO (Last-In-First-Out) algorithms, which assign priorities to jobs according their release times, and the SETF (Shortest-Execution-Time-First) and ~~LET~~ LETF (Longest-Execution-Time-First) algorithms, which assign priorities on the basis of job execution times.

* Dynamic versus Static Systems *

Jobs ~~are~~ that are ready for execution are placed in a priority queue common to all processors. When a processor is available, the job at the head of the queue executes on the processor. Such type of system as a dynamic system, because jobs are dynamically dispatched to processors.

Another approach to scheduling in multiprocessor and distributed systems is to partition the jobs in the system into subsystems and assign and bind the subsystems statically to the processors. Jobs are moved among processors only when the system must be reconfigured, that is, when the operation mode of the system changes or some processor fails. Such a system is called a static system, because the system is statically configured.

Effective Release time: The effective release time of a job without predecessors is equal to its given release time. The effective release time of a job with predecessors is equal to the maximum value among its given release time and the effective release times of all of its predecessors.

Effective deadline: The effective deadline of a job without successor is equal to its given deadline. The effective deadline of a job with successors is equal to the minimum value among its given deadline and the effective deadlines of all of its successors.

Optimality of the EDF and LST algorithms :

Priority-driven scheduling algorithm based on this priority assignment is called the Earliest-Deadline-First (EDF) algorithm. This algorithm is important because it is ~~equal~~ optimal when used to schedule jobs on a processor as long as preemption is allowed and jobs do not contend for resources.

Theorem: when preemption is allowed and jobs do not contend for resources, the EDF algorithm can produce a feasible schedule of a set J of jobs with arbitrary release times and deadlines on a processor if and only if J has feasible schedules.

Proof: Any feasible schedule of J can be systematically transformed into an EDF schedule.

For this, suppose that in a schedule, parts of J_i and J_k are scheduled in intervals I_1 and I_2 resp. Further, the deadline d_i of J_i is later than the deadline d_k of J_k, but I_1 is earlier than I_2. as given in Fig.

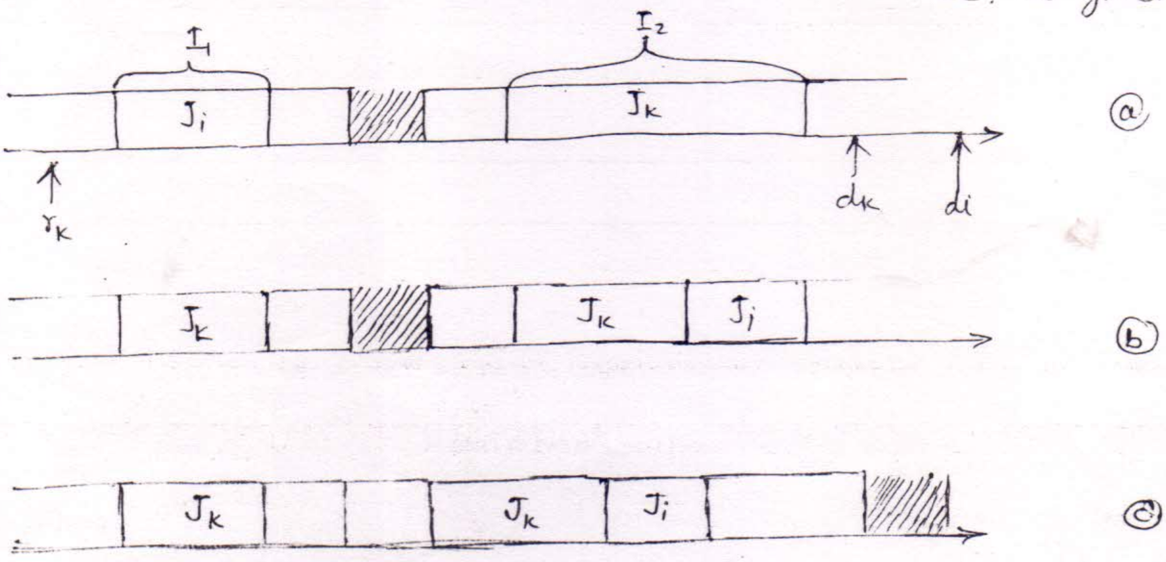


Fig. Transformation of non-EDF into an EDF schedule.

There are two cases. In the first case, the release time of J_k may be later than the end of I_1. J_k cannot be scheduled in I_1; the two jobs are already scheduled on the EDF basis.

Therefore, we have to consider only second case where the release time r_k of J_k is before the end of I_1; without loss of generality; assume that r_k is no later than the beginning of I_1.

To transform the given schedule, swap J_i and J_k. Specifically, if the interval I_1 is shorter than I_2. we move the portion of J_k that fits in I_1 forward to I_1 and move the entire portion of J_i scheduled in I_1 backward to I_2 and place it after J_k. Clearly this swap is possible. we can do a similar swap if the interval I_1 is longer than I_2: we can move the entire portion of J_i that fits in I_2 to the interval.

* Preemptive Earliest Deadline First (EDF) algorithm *

A processor following the EDF algorithm always executes the task whose absolute deadline is the earliest.

It is a dynamic-priority scheduling algorithm; the task priorities are not fixed but change depending on the closeness of their absolute deadline. EDF is also called the deadline-monotonic scheduling algorithm.

Example: Consider a following set of (aperiodic) task arrivals to a system.

Task	Arrival Time	Execution Time	Absolute Deadline
T_1	0	10	30
T_2	4	3	10
T_3	5.	10	25

when T_1 arrives, it is the only task waiting to run, and so starts executing immediately. T_2 arrives at time 4; since $d_2 < d_1$, it has higher priority than T_1 and preempts it. T_3 arrives at time 5; however, since $d_3 > d_2$, it has lower priority than T_2, and must wait for T_2 to finish. When T_2 finishes (at time 7), T_3 starts (since it has higher priority than T_1). T_3 runs until 15, at which point T_1 can resume and run to completion.

It is an optimal uniprocessor scheduling algorithm.

* Offline Versus Online Scheduling *

* validation algorithm *

* Latest Release Time (LRT)

This algorithm treats release times as deadlines and deadlines as a release times and schedules the jobs backwards, starting from the latest deadline of all jobs; in "priority driven" manner, to the current time.

Priorities are based on the release times of jobs; the later the release time, the higher the "priority". Because it may leave the processor ~~[illegible]~~ idle when there are jobs ready for execution, the LRT algorithm is not a priority-driven algorithm.

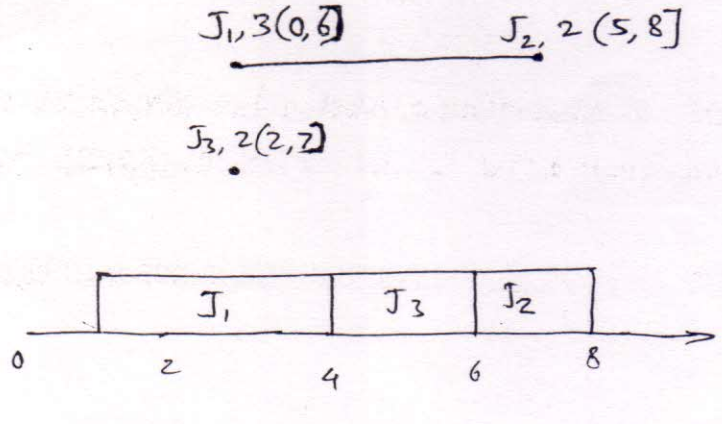


Fig. Example for LRT algorithm

* Scheduling Aperiodic and Sporadic jobs in Priority-Driven Systems *

Objectives, correctness and Optimality:

Correctness:

For correctness, assume that the operating system maintains the priority queue in fig given below.

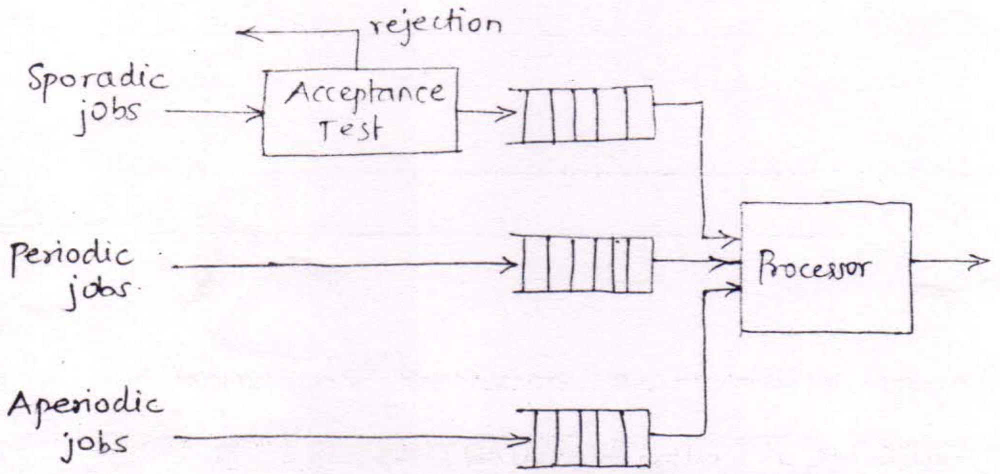


Fig. Priority queues maintained by the operating system.

The ready periodic jobs are placed in periodic task queue, ordered by their priorities that are assigned according to the given periodic task ~~queue~~ scheduling algorithm

Similarly, for sporadic and aperiodic jobs the same thing is done.

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 intervation of the schedular.

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

1. Based on the execution time and deadline of each newly arrived sporadic job, the schedular decides whether to accept or reject the job. The problems are how to do the acceptance test and how to schedule the accepted sporadic jobs
2. The schedular 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.

An algorithm is correct if it produces only correct schedules of the system. By correct schedule it means one according to which periodic and accepted sporadic jobs never miss their deadlines.

Background and Interrupt-Driven Execution versus Slack stealing:
According to the background approach, aperiodic jobs are scheduled and executed only at times when there is ~~not~~ no periodic or sporadic job ready for execution. However, the execution of aperiodic jobs may be delayed and their response times prolonged unnecessarily.

* Deferrable Server
It is the simplest ~~ser~~ of bandwidth-preserving servers.
The execution budget of a deferrable server is ~~the simplest of bandwidth~~ with period ~~P~~ 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.

Operation of deferrable server:

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

Replenishment Rule:
The execution budget of the server is set to e_s at time instants kp_k, for $k = 0, 1, 2, \ldots$

The server is not allowed to cumulate the budget from period to period.