unit 1.pdf

unit 2.pdf

unit 3.pdf

unit 4.pdf

unit 5.pdf

unit 6.pdf

unit 7.pdf

unit 8.pdf

Notes By: Lecturer Ganesh

Chapter 1

Introduction

REAL-TIME SYSTEM

Real-time systems have been defined as: "those systems in which the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced".

REAL-TIME CHARACTERISTICS

It's convenient to divide a real-time system into three components: the *controlling* system , *controlled* system and *environment*.

- avionics computer (controller) controls aircraft system (controlled) to maintain desirable flight (environment) characteristics
- desktop computer (controller) buffers, decodes and displays video streams (controlled) and adapts in response to unreliable data delivery of encoded data (environment)

Timing constraints are placed on the controlling system and its interactions with the controlled system, in response to changes in the environment. Hard and soft constraints.

- Hard constraints: missing a deadline is considered catastrophic
- Soft constraints: missing deadline is not considered to be fatal, result may have diminishing value with time.

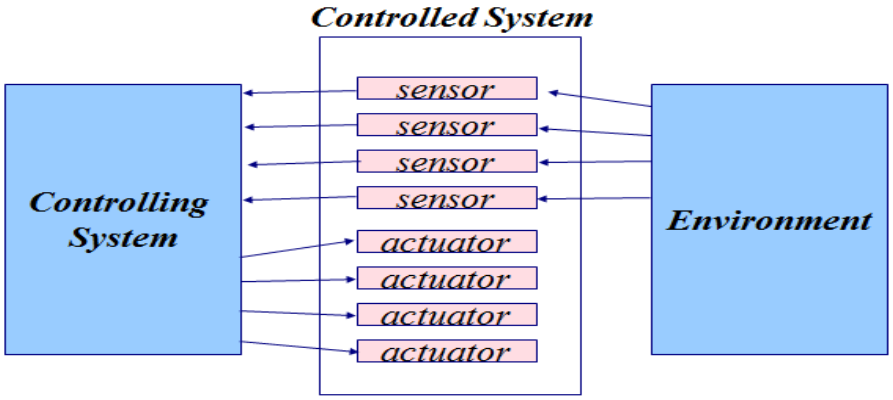


Fig Typical Real time System

REAL-TIME SYSTEM APPLICATION DOMAINS

Potential uses for real-time systems include but are not limited to:

- Telecommunication systems
- Automotive control
- Multimedia servers and workstations
- Signal processing systems
- Radar systems
- Consumer electronics
- Process control
- Automated manufacturing systems
- Supervisory control and data acquisition (SCADA) systems
- Electrical utilities
- Semiconductor fabrication systems
- Defense systems
- Avionics
- Air traffic control
- Autonomous navigation systems
- Vehicle control systems
- Transportation and traffic control systems
- Satellite systems
- Nuclear power control systems

DIGITAL CONTROL

They are the simplest and the most deterministic real-time applications. They also have the most stringent timing requirements. Many real-time systems are embedded in sensors and actuators and function as digital controllers. Figure 1–1 shows such a system. The term plant in the block diagram refers to a controlled system, for example, an engine, a brake, an aircraft, a patient. The state of the plant is monitored by sensors and can be changed by actuators. The real-time (computing) system estimates from the sensor readings the current state of the plant and computes a control output based on the difference between the current state and the desired state (called reference input in the figure). We call this computation the *control-law computation* of

the controller. The output thus generated activates the actuators, which bring the plant closer to the desired state.

A simple example: As an example, we consider an analog single-input/single-output PID (Proportional, Integral, and Derivative) controller. This simple kind of controller is commonly used in practice. The analog sensor reading *y(t)* gives the measured state of the plant at time *t*. Let $e(t) = r(t) - y(t)$ denote the difference between the desired state $r(t)$ and the measured state $y(t)$ at time t . The output $u(t)$ of the controller consists of three terms: a term that is proportional to $e(t)$, a term that is proportional to the integral of $e(t)$ and a term that is proportional to the derivative of $e(t)$.

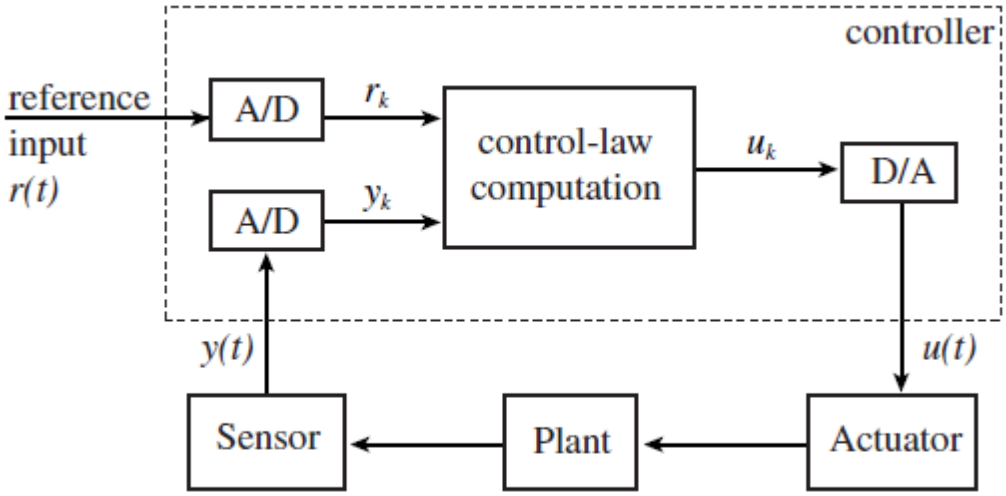


FIGURE 1–1 A digital controller.

HIGH LEVEL CONTROLS:

Controllers in a complex monitor and control system are typically organized hierarchically. One or more digital controllers at the lowest level directly control the physical plant. Each output of a higher-level controller is a reference input of one or more lower-level controllers. With few exceptions, one or more of the higher-level controllers interfaces with the operator(s).

Examples of Control Hierarchy:

For example, a patient care system may consist of microprocessor-based controllers that monitor and control the patient's blood pressure, respiration, glucose, and so forth. There may be a higher-level controller (e.g., an expert system) which interacts with the operator (a nurse or doctor) and chooses the desired values of these health indicators. While the computation done by each digital controller is simple and nearly deterministic, the computation of a high level controller is likely to be far more complex and variable. While the period of a low level control-law computation ranges from milliseconds to seconds, the periods of high-level control-law computations may be minutes, even hours.

Next example is as:

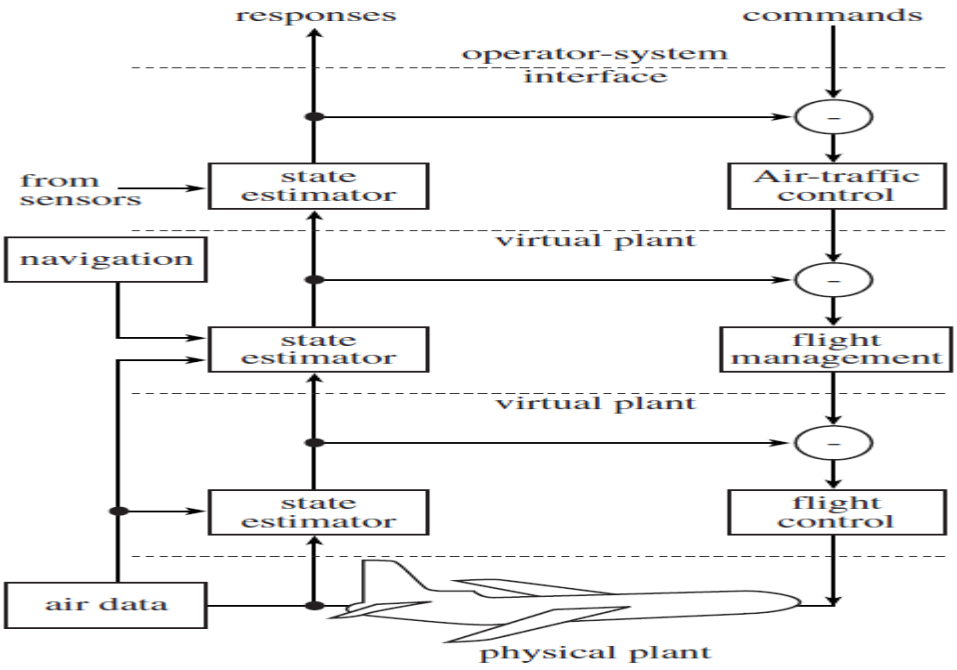


FIGURE 1–4 Air traffic/flight control hierarchy.

shows a more complex example: the hierarchy of flight control, avionics, and air traffic control systems.5 The Air Traffic Control (ATC) system is at the highest level. It regulates the flow of flights to each destination airport. It does so by assigning to each aircraft an arrival time at each metering fix6 (or waypoint) en route to the destination: The aircraft is supposed to arrive at the metering fix at the assigned arrival time. At any time while in flight, the assigned arrival time to the next metering fix is a reference input to the on-board flight management system. The flight management system chooses a time-referenced flight path that brings the aircraft to the next metering fix at the assigned arrival time. The cruise speed, turn radius, decent/accent rates, and

so forth required to follow the chosen time-referenced flight path are the reference inputs to the flight controller at the lowest level of the control hierarchy.

GUIDANCE AND CONTROL:

While a digital controller deals with some dynamical behavior of the physical plant, a second level controller typically performs guidance and path planning functions to achieve a higher level goal. In particular, it tries to find one of the most desirable trajectories among all trajectories that meet the constraints of the system. The trajectory is most desirable because it optimizes some cost function(s). The algorithm(s) used for this purpose is the solution(s) of some constrained optimization problem(s).

As an example, we look again at a flight management system. The constraints that must be satisfied by the chosen flight path include the ones imposed by the characteristics of the aircraft, such as the maximum and minimum allowed cruise speeds and decent/accent rates, as well as constraints imposed by external factors, such as the ground track and altitude profile specified by the ATC system and weather conditions. A cost function is fuel consumption: A most desirable flight path is a most fuel efficient among all paths that meet all the constraints and will bring the aircraft to the next metering fix at the assigned arrival time. This problem is known as the **constrained fixed-time, minimum-fuel** problem. When the flight is late, the flight management system may try to bring the aircraft to the next metering fix in the shortest time. In that case, it will use an algorithm that solves the time-optimal problem.

REAL-TIME COMMAND AND CONTROL

The controller at the highest level of a control hierarchy is a command and control system. An Air Traffic Control (ATC) system is an excellent example. Figure 1–5 shows a possible architecture. The ATC system monitors the aircraft in its coverage area and the environment (e.g, weather condition) and generates and presents the information needed by the operators (i.e., the air traffic controllers). Outputs from the ATC system include the assigned arrival times to metering fixes for individual aircraft. As stated earlier, these outputs are reference inputs to on-board flight management systems. Thus, the ATC system indirectly controls the embedded components in low levels of the control hierarchy. In addition, the ATC system provides voice

and telemetry links to on-board avionics. Thus it supports the communication among the operators at both levels (i.e., the pilots and air traffic controllers).

The ATC system gathers information on the "state" of each aircraft via one or more active radars. Such a radar interrogates each aircraft periodically. When interrogated, an air- craft responds by sending to the ATC system its "state variables": identifier, position, altitude, heading, and so on. (In Figure 1–5, these variables are referred to collectively as a track record, and the current trajectory of the aircraft is a track.) The ATC system processes messages from aircraft and stores the state information thus obtained in a database. This information is picked up and processed by display processors. At the same time, a surveillance system continuously analyzes the scenario and alerts the operators whenever it detects any potential hazard (e.g., a possible collision). Again, the rates at which human interfaces (e.g., keyboards and displays) operate must be at least 10 Hz. The other response times can be considerably larger. For example, the allowed response time from radar inputs is one to two seconds, and the period of weather updates is in the order of ten seconds. From this example, we can see that a command and control system bears little resemblance to low-level controllers. In contrast to a low-level controller whose workload is either purely or mostly periodic, a command and control system also computes and communicates in response to sporadic events and operators' commands. Furthermore, it may process image and speech, query and update databases, simulate various scenarios, and the like. The resource and processing time demands of these tasks can be large and varied. Fortunately, most of the timing requirements of a command and control system are less stringent. Whereas a low-level control system typically runs on one computer or a few computers connected by a small network or dedicated links, a command and control system is often a large distributed system containing tens and hundreds of computers and many different kinds of networks. In this respect, it resembles interactive, on-line transaction systems (e.g., a stock price quotation system) which are also sometimes called real-time systems.

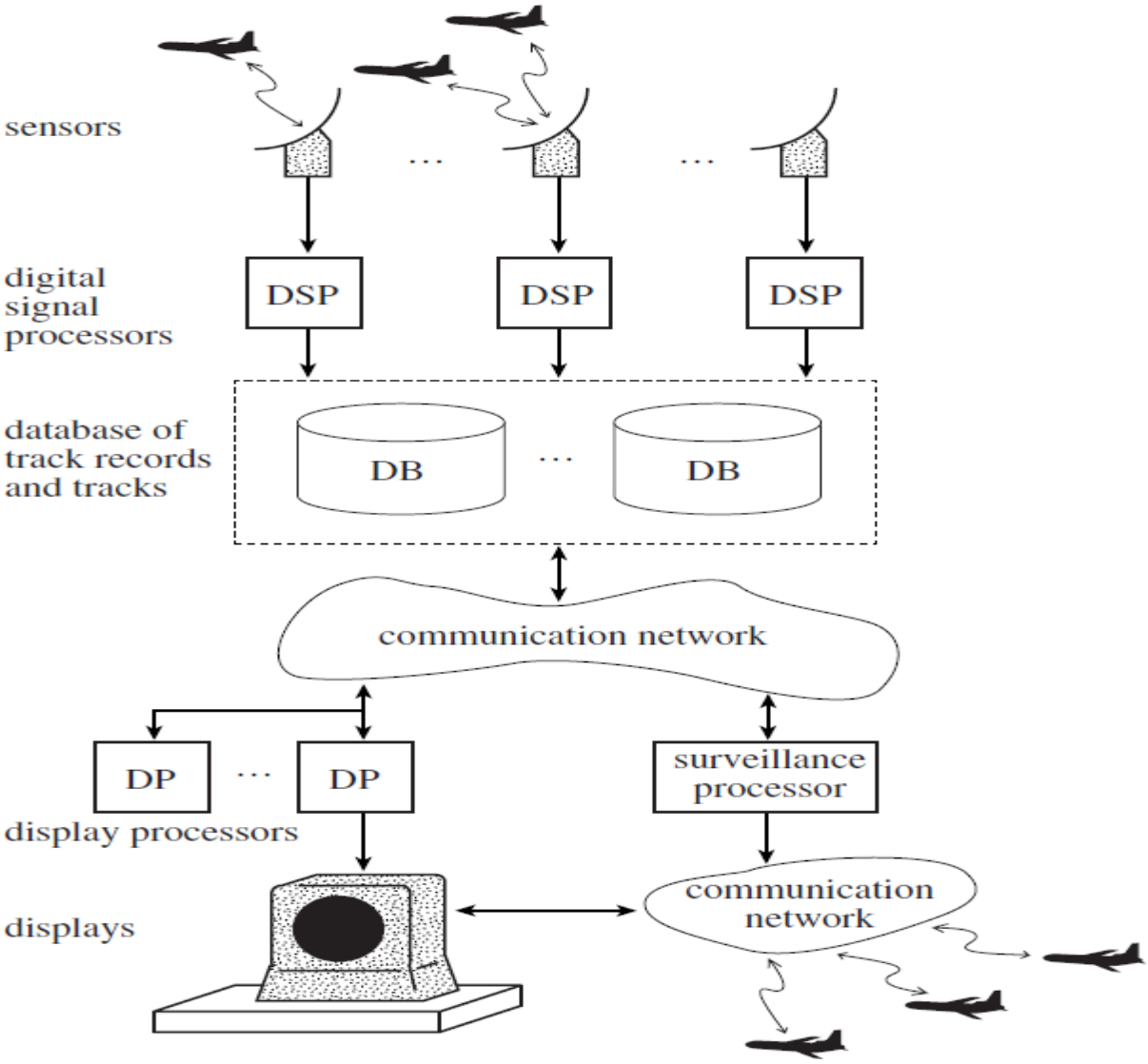


Fig. Architecture of Air Traffic Control System

Radar Signal processing

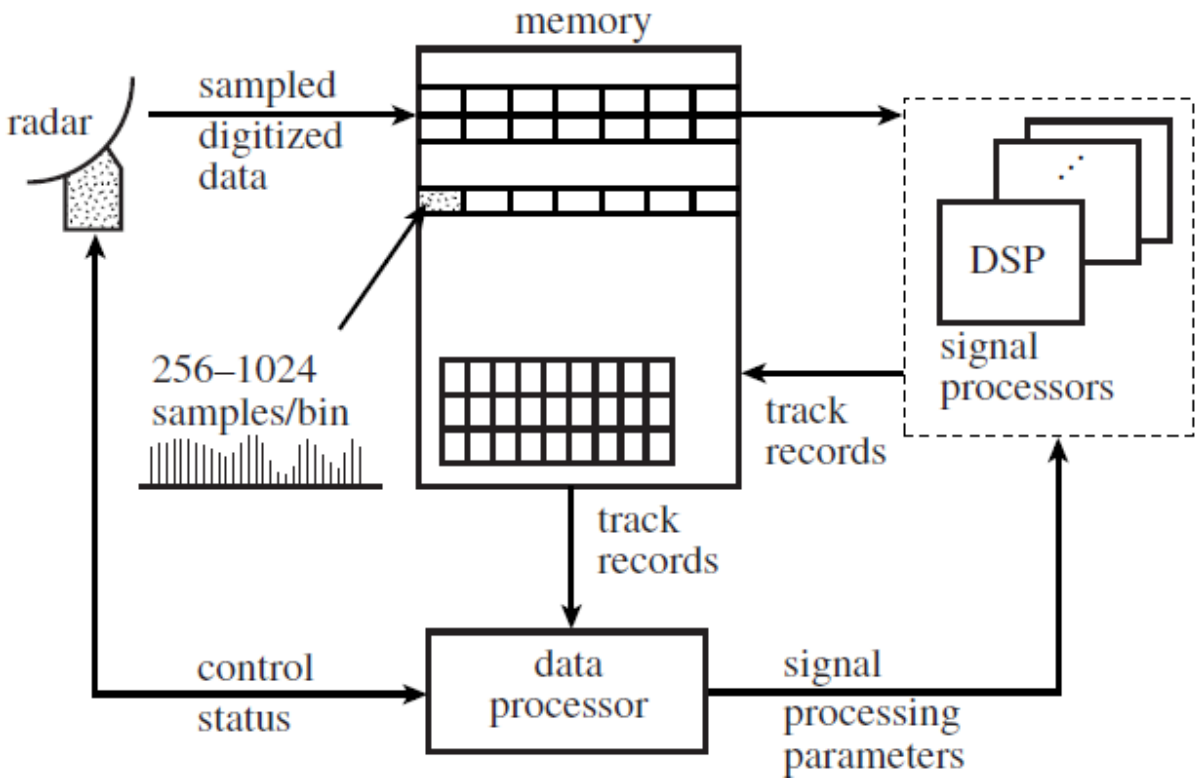


Fig . Radar Signal processing and Tracking System

The Radar signal processing system consists of an Input/Output (I/O) subsystem that samples and digitizes the echo signal from the radar and places the sampled values in a shared memory. An array of digital signal processors processes these sampled values. The data thus produced are analyzed by one or more data processors, which not only interface with the display system, but also generate commands to control the radar and select parameters to be used by signal processors in the next cycle of data collection and analysis.

How signal processing is done?

To search for objects of interest in its coverage area, the radar scans the area by pointing its antenna in one direction at a time. During the time the antenna dwells in a direction, it first sends a short radio frequency pulse. It then collects and examines the echo signal returning to the antenna. The echo signal consists solely of background noise if the transmitted pulse does not hit

any object. On the other hand, if there is a reflective object (e.g., an airplane or storm cloud) at a distance *x* meters from the antenna, the echo signal reflected by the object returns to the antenna

at approximately $2x/c$ seconds after the transmitted pulse, where $c = 3 \times 108$ meters per second is the speed of light. The echo signal collected at this time should be stronger than when there is no reflected signal. If the object is moving, the frequency of the reflected signal is no longer equal to that of the transmitted pulse. The amount of frequency shift (called Doppler shift) is proportional to the velocity of the object. Therefore, by examining the strength and frequency spectrum of the echo signal, the system can determine whether there are objects in the direction pointed at by the antenna and if there are objects, what their positions and velocities are.

BASIC TERMS USED IN RADAR SIGNAL PROCESSING:

TRACKING

Strong noise and man-made interferences, including electronic counter measure (i.e., jamming), can lead the signal processing and detection process to wrong conclusions about the presence of objects. A track record on a non existing object is called a false return.

An application that examines all the track records in order to sort out false returns from real ones and update the trajectories of detected objects is called a *tracker*. Using the jargon of the subject area, we say that the tracker assigns each measured value (i.e., the tuple of position and velocity contained in each of the track records generated in a scan) to a trajectory.

If the trajectory is an existing one, the measured value assigned to it gives the current position and velocity of the object moving along the trajectory. If the trajectory is new, the measured value gives the position and velocity of a possible new object.

GATING

Typically, tracking is carried out in two steps: gating and data association. *Gating* is the process of putting each measured value into one of two categories depending on whether it can or cannot be tentatively assigned to one or more established trajectories.

The gating process tentatively assigns a measured value to an established trajectory if it is within a threshold distance G away from the predicted current position and velocity of the object moving along the trajectory. (Below, we call the distance between the measured and predicted values the distance of the assignment.) The threshold G is called the track gate. It is chosen so

that the probability of a valid measured value falling in the region bounded by a sphere of radius *G* centered around a predicted value is a desired constant.

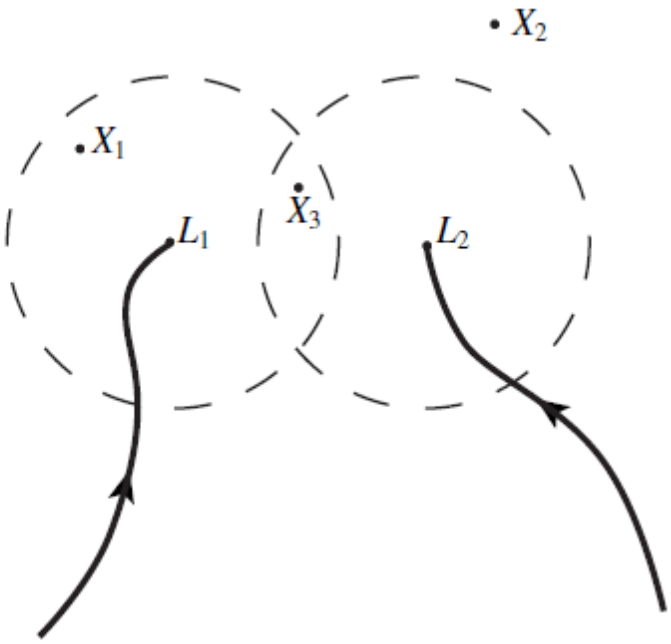


Fig . Gating Process

DATA ASSOCIATION:

The tracking process completes if, after gating, every measured value is assigned to at most one trajectory and every trajectory is assigned at most one measured value. This is likely to be case when (1) the radar signal is strong and interference is low (and hence false returns are few) and (2) the density of objects is low. Under adverse conditions, the assignment produced by gating may be ambiguous, that is, some measured value is assigned to more than one trajectory or a trajectory is assigned more than one measured value. The data association step is then carried out to complete the assignments and resolve ambiguities.

OTHER REAL-TIME APPLICATIONS:

Two most common real-time applications are real-time databases and multimedia applications.

REAL-TIME DATABASES:

The term real-time database systems refers to a diverse spectrum of information systems, ranging from stock price quotation systems, to track records databases, to real-time file systems. What distinguish these databases from non real-time databases is the perishable nature of the data maintained by them. Specifically, a real-time database contains data objects, called *image objects* that represent real-world objects. The attributes of an image object are those of the represented real world object.

For example, an air traffic control database contains image objects that represent aircraft in the coverage area. The attributes of such an image object include the position and heading of the aircraft. The values of these attributes are updated periodically based on the measured values of the actual position and heading provided by the radar system. Without this update, the stored position and heading will deviate more and more from the actual position and heading. In this sense, the quality of stored data degrades. This is why we say that real-time data are perishable. In contrast, an underlying assumption of non real-time databases (e.g., a payroll database) is that in the absence of updates the data contained in them remain good (i.e., the database remains in some consistent state satisfying all the data integrity constraints of the database).

MULTIMEDIA APPLICATIONS:

A multimedia application may process, store, transmit, and display any number of video streams, audio streams, images, graphics, and text. A video stream is a sequence of data frames which encodes a video. An audio stream encodes a voice, sound, or music. Without compression, the storage space and transmission bandwidth required by a video are enormous. (As an example, we consider a small 100 × 100-pixel, 30-frames/second color video. The intensity and color of each pixel is given by the sample values of a luminance and two chrominance signal components,12 respectively, at the location of the pixel. If uncompressed, the video requires a transmission bandwidth of 2.7 Mbits per second when the value of each component at each pixel is encoded with 3 bits.) Therefore, a video stream, as well as the associated audio stream, is invariably compressed as soon as it is captured.

Chapter2

JOBS AND PROCESSORS:

Each unit of work that is scheduled and executed by the system is called a *job* and a set of related jobs which jointly provide some system function a *task*. Hence, the computation of a control law is a job. So is the computation of a FFT (Fast Fourier Transform) of sensor data, or the transmission of a data packet, or the retrieval of a file, and so on.

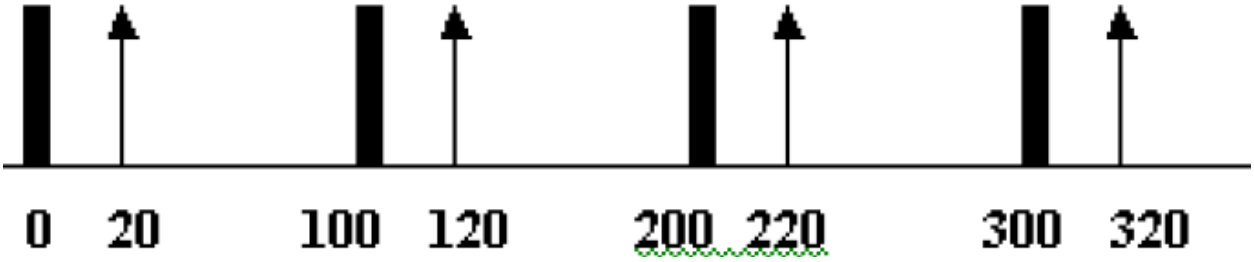
A job executes or is executed by the (operating) system. Every job executes on some resource. For example, the jobs mentioned above execute on a CPU, a network, and a disk, respectively. These resources are called servers in queuing theory literature and, sometimes, active resources in real-time systems literature. To avoid overloading this term, we call all these resources *processors* except occasionally when we want to be specific about what they are.

RELEASE TIMES, DEADLINES, AND TIMING CONSTRAINTS

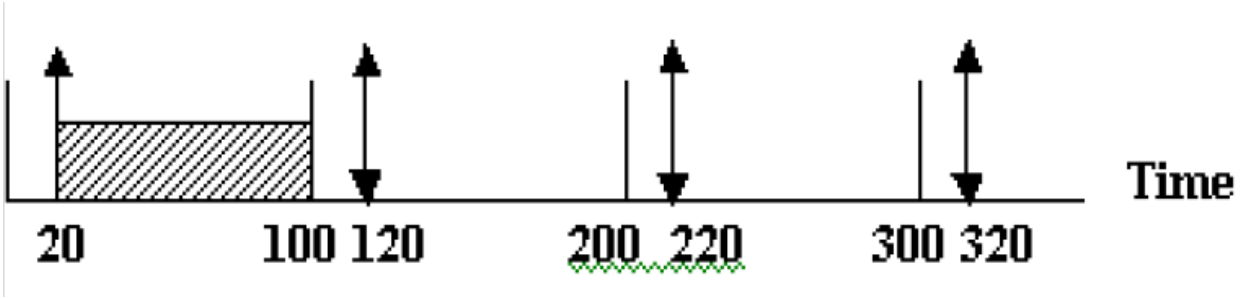
The release time of a job is the instant of time at which the job becomes available for execution. The job can be scheduled and executed at any time at or after its release time whenever its data and control dependency conditions are met.

As an example, we consider a system which monitors and controls several furnaces. After it is initialized and starts execution (say at time 0), the system samples and reads each temperature sensor every 100 msec and places the sampled readings in memory. It also computes the control law of each furnace every 100 msec in order to process the temperature readings and determine flow rates of fuel, air, and coolant. Suppose that the system begins the first control-law computation at time 20 msec. The fact that the control law is computed periodically can be stated in terms of release times of the control-law computation jobs $J0, J1, \ldots, Jk, \ldots$. The release time of the job Jk in this job stream is $20 + k \times 100$ msec, for $k = 0, 1, \ldots$. *We* say that jobs have

no release time if all the jobs are released when the system begins execution. So release times are 20 ms , 120 ms ,220 ms…….



The ***deadline*** of a job is the instant of time by which its execution is required to be completed. Suppose that in the previous example, each control-law computation job must complete by the release time of the subsequent job. Then, their deadlines are 120 msec, 220 msec, and so on, respectively. Alternatively, if the control-law computation jobs must complete sooner, their deadlines may be 70 msec, 170 msec, and so on. We say that a job has no deadline if its deadline is at infinity.



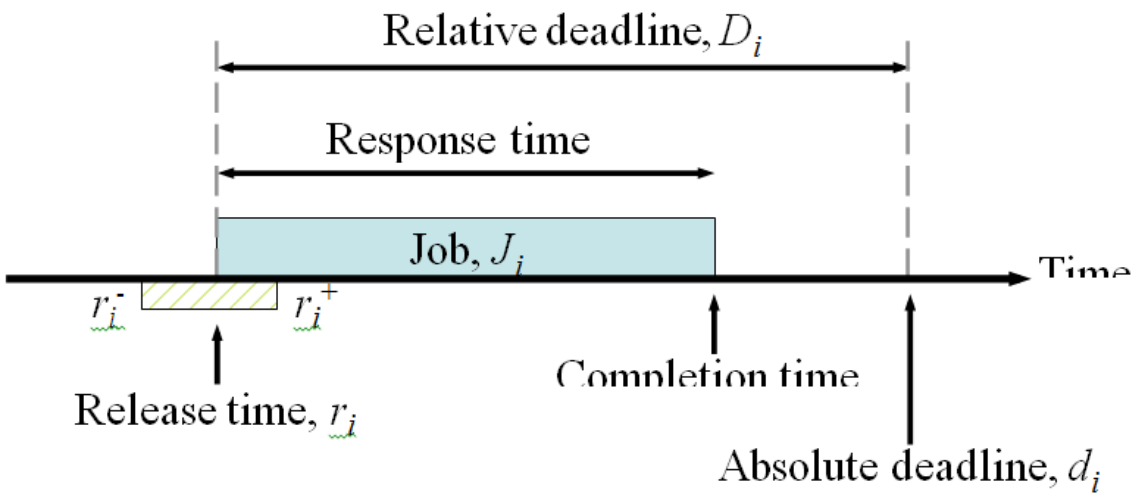
OTHER DEFINITIONS:

RESPONSE TIME: the length of time from the release time of the job to the instant when it completes.
Response time = Job completion time – Job release time
Job's timing requirements can be expressed in terms of "response time.

RELATIVE DEADLINE: the maximum allowable response time of a job its *relative deadline*. The relative deadline of above figure is 100.

ABSOLUTE DEADLINE: is release time plus relative deadline. The absolute deadline of above figure is 120.

These terminologies can be illustrated as following figure



TIMING CONSTRAINT:

We call a constraint imposed on the timing behavior of a job a *timing constraint*. In its simplest form, a timing constraint of a job can be specified in terms of its release time and relative or absolute deadlines, as illustrated by the above example. Some complex timing constraints cannot be specified conveniently in terms of release times and deadline.

HARD AND SOFT TIMING CONSTRAINTS:

They are based on the functional criticality of jobs, usefulness of late results, and deterministic or Probabilistic nature of the constraints.

A timing constraint or deadline is *hard* if the failure to meet it is considered to be a fatal fault. A hard deadline is imposed on a job because a late result produced by the job after the deadline may have disastrous consequences. (As examples, a late command to stop a train may cause a

collision, and a bomb dropped too late may hit a civilian population instead of the intended military target.)

In contrast, the late completion of a job that has a *soft deadline* is undesirable. However, a few misses of soft deadlines do no serious harm; only the system's overall performance becomes poorer and poorer when more and more jobs with soft deadlines complete late.

In real-time systems literature, the distinction between hard and soft timing constraints is sometimes stated quantitatively in terms of the usefulness of results (and therefore the overall system performance) as functions of the tardinesses of jobs. The *tardiness* of a job measures how late it completes respective to its deadline. Its tardiness is zero if the job completes at or before its deadline; otherwise, if the job is late, its tardiness is equal to the difference between its *completion time* (i.e., the time instant at which it completes execution) and its deadline. The usefulness of a result produced by a soft real-time job (i.e, a job with a soft deadline) decreases gradually as the tardiness of the job increases, but the usefulness of a result produced by a hard real-time job (i.e., a job with a hard deadline) falls off abruptly and may even become negative when the tardiness of the job becomes larger than zero. The deadline of a job is softer if the usefulness of its result decreases at a slower rate. By this means, we can define a spectrum of hard/soft timing constraints.

Sometimes, we see this distinction made on the basis of whether the timing constraint is expressed in deterministic or probabilistic terms. If a job must never miss its deadline, then the deadline is hard. On the other hand, if its deadline can be missed occasionally with some acceptably low probability, then its timing constraint is soft.

HARD TIMING CONSTRAINTS AND TEMPORAL QUALITY-OF-SERVICE GUARANTEES:

The timing constraint of a job is hard, and the job is a hard real-time job, if the user requires the validation that the system always meets the timing constraint. By *validation*, we mean a demonstration by a provably correct, efficient procedure or by exhaustive simulation and testing..

On the other hand, if no validation is required, or only a demonstration that the job meets some *statistical constraint* (i.e., a timing constraint specified in terms of statistical averages) suffices, then the timing constraint of the job is soft.

This way to differentiate between hard and soft timing constraints is compatible with the distinction between *guaranteed* and *best-effort* services

If the user wants the temporal quality (e.g., response time and jitter) of the service provided by a task guaranteed and the satisfaction of the timing constraints defining the temporal quality validated, then the timing constraints are hard. On the other hand, if the user demands the best quality of service the system can provide but allows the system to deliver qualities below what is defined by the timing constraints, then the timing constraints are soft.

SOME REASONS FOR REQUIRING TIMING GUARANTEES

Many embedded systems are hard real-time systems. Deadlines of jobs in an embedded system are typically derived from the required responsiveness of the sensors and actuators monitored and controlled by it. As an example, we consider an automatically controlled train. It cannot stop instantaneously. When the signal is red (stop), its braking action must be activated a certain distance away from the signal post at which the train must stop. This braking distance depends on the speed of the train and the safe value of deceleration. From the speed and safe deceleration of the train, the controller can compute the time for the train to travel the braking distance. This time in turn imposes a constraint on the response time of the jobs which sense and process the stop signal and activate the brake. No one would question that this timing constraint should be hard and that its satisfaction must be guaranteed.

Similarly, each control-law computation job of a flight controller must be completed in time so that its command can be issued in time. Otherwise, the plane controlled by it may become oscillatory (and the ride bumpy) or even unstable and uncontrollable. For this reason, we want the timely completion of all control-law computations guaranteed.

HARD REAL-TIME SYSTEMS:

A hard real-time task is one that is constrained to produce its results within certain predefined time bounds. The system is considered to have failed whenever any of its hard real-time tasks does not produce its required results before the specified time bound. An example of a system having hard real-time tasks is a robot. The robot cyclically carries out a number of activities including communication with the host system, logging all completed activities, sensing the environment to detect any obstacles present, tracking the objects of interest, path planning,

effecting next move, etc. Now consider that the robot suddenly encounters an obstacle. The robot must detect it and as soon as possible try to escape colliding with it. If it fails to respond to it quickly (i.e. the concerned tasks are not completed before the required time bound) then it would collide with the obstacle and the robot would be considered to have failed. Therefore detecting obstacles and reacting to it are hard real-time tasks. Another application having hard real-time tasks is an anti-missile system. An anti-missile system consists of the following critical activities (tasks). An anti-missile system must first detect all incoming missiles, properly position the anti-missile gun, and then fire to destroy the incoming missile before the incoming missile can do any damage. All these tasks are hard real- time in nature and the anti-missile system would be considered to have failed, if any of its tasks fails to complete before the corresponding deadlines.

SOFT REAL TIME SYSTEMS:

A system in which jobs have soft deadlines is a *soft real-time system.* Soft real-time tasks also have time bounds associated with them. However, unlike hard real-time tasks, the timing constraints on soft real-time tasks are not expressed as absolute values. Instead, the constraints are expressed either in terms of the average response times required. Examples of such systems include on-line transaction systems and telephone switches, as well as electronic games, multimedia system, web browsing. Normally, after an URL (Uniform Resource Locater) is clicked, the corresponding web page is fetched and displayed within a couple of seconds on the average. However, when it takes several minutes to display a requested page, we still do not consider the system to have failed, but merely express that the performance of the system has degraded. Another example of a soft real-time task is a task handling a request for a seat reservation in a railway reservation application. Once a request for reservation is made, the response should occur within 20 seconds on the average. The response may either be in the form of a printed ticket or an apology message on account of unavailability of seats. Alternatively, we might state the constraint on the ticketing task as: At least in case of 95% of reservation requests, the ticket should be processed and printed in less than 20 seconds. Let us now analyze the impact of the failure of a soft real-time task to meet its deadline, by taking the example of the railway reservation task. If the ticket is printed in about 20 seconds, we feel that the system is working fine and get a feel of having obtained instant results. As already stated, missed deadlines of soft real-time tasks do not result in system failure

Chapter-3

Reference model for real-time systems

Reference model is characterized by:

- A model that describes applications running on the system.
- A model that describes the resources available to those applications.
- A scheduling algorithm that define how the applications execute and use the resources.

Processors and resources

We divide all the system resources into two types:

- processors (sometimes called servers or active resources such as computers, data links, database servers etc.)
- other passive resources (such as memory, sequence numbers, mutual exclusion locks etc.)

  Jobs may need some resources in addition to the processor in order to make progress.

Processors

- Processors carry out machine instructions, move data, retrieve files, process queries etc.
- Every job must have one or more processors in order to make progress towards completion.
- Sometimes we need to distinguish types of processors.

Types of processors

- Two processors are of the same type if they can be used interchangeably and are functionally identical.
  - Two data links with the same transmission rates between the same two nodes are considered processors of the same type. Similarly processors in a symmetric multiprocessor system are of the same type.
- One of the attributes of a processor is its speed. We will assume that the rate of progress a job makes depends on the speed of the processor on which it is running.

Speed

We can explicitly model the dependency of job progression and processor speed by making the amount of time a job requires completing a function of the processor speed. In contrast we do not associate speed with a resource. How long a job takes to complete does not depend on the speed of any resource it uses during execution.

Examples of processors:

Threads scheduled on CPU, data scheduled on a transmission link, read/write requests scheduled to disks, transmission scheduled on database server.

Resources:

A resource R is a passive entity upon which jobs may depend. Resources are of different types and sizes but they do not have a speed attribute. Example includes memory, database locks, mutexes etc. Some Resources are generally reusable and they are not consumed during use whereas other resources are consumed during use and cannot be used again. Some resources are serially reusable. There may be many units of a serial resource, but each can only be used by one job at a time.

A resource is plentiful if no job is ever prevented from running by the lack of this resource.

Processor or Resource?

- We sometimes model some elements of the system as processors and sometimes as resources, depending on how we use the model.
- For example, in a distributed system a computation job may invoke a server on a remote processor.
  - If we want to look at how the response time of this job is affected by the way the job is scheduled on its local processor, we can model the remote server as a resource.
  - We may also model the remote server as a processor.
- There are no fixed rules to guide us in deciding whether to model something as a processor or as a resource, or to guide us in many other modelling choices.
- A good model can give us better insight into the real-time problem we are considering.
- A bad model can confuse us and lead to a poor design and implementation.

- In many ways this is an art which requires some skill but provides great freedom for designing and implementing real-time systems.

Real-time workload parameters

- **The number of tasks or jobs in the system.**
  - In many embedded systems the number of tasks is fixed for each operational mode, and these numbers are known in advance.
  - In some other systems the number of tasks may change as the system executes.
  - Nevertheless, the number of tasks with hard timing constraints is known at all times.
  - When the satisfaction of timing constraints is to be guaranteed, the admission and deletion of hard real-time tasks is usually done under the control of the run-time system.
- **The run-time system**
  - The run-time system must maintain information on all existing hard real-time tasks, including the number of such tasks, and all their real-time constraints and resource requirements.

The job

- Each job J_i is characterized by its temporal parameters, interconnection parameters and functional parameters.
- Its temporal parameters tell us its timing constraints and behaviour.
- Its interconnection parameters tell us how it depends on other jobs and how other jobs depend on it.
- Its functional parameters specify the intrinsic properties of the job.

Job temporal parameters

- For job J_i
  - Release time r_i
  - Absolute deadline d_i
  - Relative deadline D_i
  - Feasible interval $(r_i, d_i]$

- d_i and D_i are usually derived from the timing requirements of J_i, other jobs in the same task as J_i, and the overall system.
- These parameters are part of the system specification.

Release time:

- In many systems we do not know exactly when each job will be released. i.e. we do not know r_i . But We know that r_i is in the range $[r_i^-, r_i^+]$
  - R_i can be as early as r_i^- and as late as r_i^+
  - Some models assume that only the range of r_i is known and call this range, **release time jitter.**
  - If the release time jitter is very small compared with other temporal parameters, we can approximate the actual release time by its earliest r_i^- or latest r_i^+ release time, and say that the job has a fixed release time.

Sporadic jobs

Most real-time systems have to respond to external events which occur randomly. When such an event occurs the system executes a set of jobs in response. The release times of those jobs are not known until the event triggering them occurs. These jobs are called *sporadic jobs* or *aperiodic jobs* because they are released at random times.

Release time of sporadic job

- The release times of sporadic and aperiodic jobs are random variables.
- The system model gives the probability distribution A(x) of the release time of such a job.
- A(x) gives us the probability that the release time of a job is at or earlier than x, or in the case of interrelease time, that it is less than or equal to x.

Arrival time:

Rather than speaking of release times for aperiodic jobs, we sometimes use the term arrival time (or interarrival time) which is commonly used in queueing theory.An aperiodic job arrives when it is released. A(x) is the arrival time distribution or interarrival time distribution.

Execution Time:

Another temporal parameter of a job J_i is its execution time, e_i. e_i is the time required to complete the execution of J_i when it executes alone and has all the resources it requires. The value of e_i depends mainly on the complexity of the job and the speed of the processor used to execute the job. e_i does not depend on how the job is scheduled.

The execution time of a job may vary for many reasons.

- A computation may contain conditional branches and these conditional branches may take different amounts of time to complete.
- The branches taken during the execution of a job depend on input data.
- If the underlying system has performance enhancing features such as caches and pipelines, the execution time can vary each time a job executes, even without conditional branches.

Thus the actual execution time of a computational job may be unknown until it completes.

Characterizing execution time

- What can be determined through analysis and measurement are the maximum and minimum amounts of time required to complete each job.
- We know that the execution time e_i of job J_i is in the range $[e_i^-, e_i^+]$ where e_i^- is the minimum execution time and e_i^+ is the maximum execution time of job J_i.
- We assume that we know e_i^- and e_i^+ of every hard real-time job J_i, even if we don't know e_i.

Maximum execution time

- For the purpose of determining whether each job can always complete by its deadline, it suffices to know its maximum execution time.
- In most deterministic models used to characterize hard real-time applications, the term execution time e_i of each job J_i specifically means its maximum execution time.
- However we don't mean that the actual execution time is fixed and known, only that it never exceeds our e_i (which may actually be e_i^+)

Periodic task model:

It is a deterministic model.

The jobs of a given task are repeated at regular and are modeled as periodic, with period p. Accuracy of model decreases with increasing jitter. Task T_i is a series of periodic Jobs J_{ij} which may be described with the following parameters:

p_i - *period*, minimum inter-release interval between jobs in Task T_i.

e_i - maximum execution time for jobs in task T_i.

r_{ij} - release time of the j^{th} Job in Task i (J_{ij} in T_i).

φ_i - phase of Task T_i, equal to r_{i1}, i.e. it is the release time of the first job in the task T_i.

H - *Hyperperiod* = Least Common Multiple of p_i for all i: H = lcm(pi), for all i. The number of jobs in a hyperperiod is equal to the sum of (H/p_i) over all i.

u_i - utilization of Task T_i and is equal to e_i/p_i.

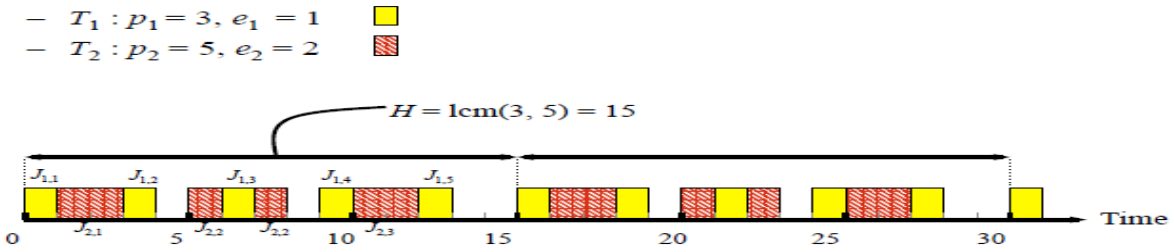
U - Total utilization = Sum over all u_i.

d_i - absolute deadline

D_i - relative deadline

$(r_i, d_i]$ - feasible interval

In other words the periodic task model is a four tuples (**φ_i**, **p_i**, **e_i**, **D_i** **).** E.g consider following periodic task



A system of periodic tasks repeats after the hyper-period, $H = \text{lcm}(p_i)$ for $i = 1, 2, \ldots, n$

The number of jobs in the hyper period = sum (H/p1 +H/p2)

=sum (15/3+15/5)

= 8

Total utilization= sum (e1/p1 +e2/p2)

=sum(1/3+2/5)

=0.733

Functional parameters

- While scheduling and resource control decisions are made independently of most functional characteristics of jobs, there are several functional properties that do affect these decisions.
- The workload model must explicitly describe these properties using functional parameters:
  - Preemptivity
  - Criticality
  - Optional interval
  - Laxity type

Preemptivity of Jobs:

The scheduler may suspend the execution of a less urgent job and give the processor to a more urgent job. Later, the less urgent job can resume its execution. This interruption of job execution is called *preemption.* A job is *preemptable* if its execution can be suspended at any time to allow the execution of other jobs and can later be resumed from the point of suspension. A job is *non-preemptable* if it must be executed from start to completion without interruption. This constraint may be imposed because its execution, if suspended, must be executed again from the beginning. An example is an interrupt handling job. An interrupt handling job usually begins by saving the state of the processor. This small portion of the job is non-preemptable since suspending the execution may cause serious errors in the data structures shared by the jobs.

e.g. in the case of CPU jobs, the state of the pre-empted job includes the contents of the CPU registers. After saving the contents of the registers in memory and before the preempting job can start, the operating system must load the new register values, clear pipelines, perhaps clear the caches, etc. These actions are called a *context switch*. The amount of time required to accomplish a context switch is called a context-switch time.

The terms context switch and context-switch time are used to mean the overhead work done during preemption, and the time required to accomplish this work.

Criticality of Jobs

In any system, jobs are not equally important. The importance (or criticality) of a job is a positive number that indicates how critical a job is with respect to other jobs. The more important a job, the higher its priority or the larger its weight Because priority and weight can have other meanings, During an overload when it is not possible to schedule all the jobs to meet their deadlines, it may make sense to sacrifice the less critical jobs, so that the more critical jobs meet their deadlines. For this reason, some scheduling algorithms try to optimize weighted performance measures, taking into account the importance of jobs.

Optional Executions –

An application may be structured in such a way that some portions are optional while the rest is mandatory. Delay in completion or skipping of an optional job may **degrade performance** but still the system functions are satisfactory. Mandatory jobs must be executed to completion. During transient overload, the mandatory portions must run to completion while the optional portion may be skipped, completely or partially executed.

Laxity type or laxity function –

Laxity can be used to indicate the relative importance of a time constraint, for example hard versus soft constraints. May be supplemented with a utility function (for soft constraints) that gives the usefulness of a result versus its degree of tardiness. The following figure gives several usefulness functions as examples. The ones shown as solid step functions are usually associated with hard real-time jobs. The usefulness of the result becomes zero or negative as soon as the job is tardy. The dashed and dotted lines in following figure show two other usefulness functions. In particular, the dotted ones may be that of a point-of-sales transaction, for example, one that executes to check whether you have enough credit for the current purchase. If the job is late, you and the salesperson become more and more impatient. The usefulness of the result decreases gradually. It may be tolerable if the update completes slightly late and the price *x* written into the database is somewhat old. However, if the transaction completes so late that the current price differs significantly from *x*, the result *x* can be misleading. By that time, the usefulness of this update becomes negative.

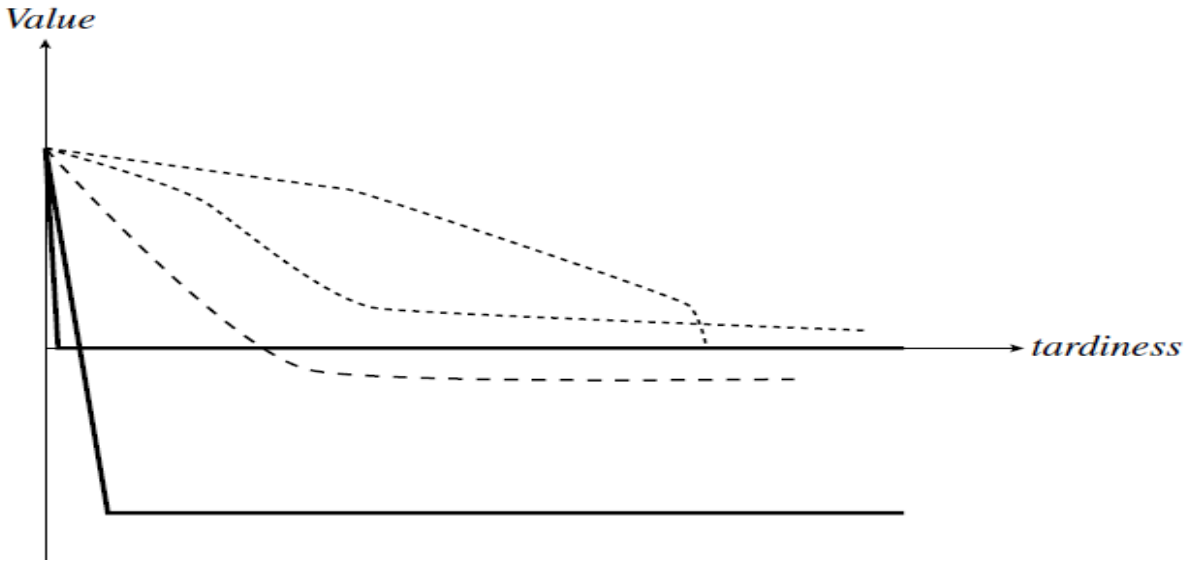


Fig:- Examples of usefulness function

Resource parameters of jobs and parameters of resources:

Every job requires a processor throughout its execution. A job may also require some resources. The resource parameters of each job give us the type of processor and the units of each resource type required by the job and the time intervals during its execution when the resources are required.

Preemptivity of Resources: A resource parameter is *preemptivity*. A resource is *non-preemptable* if each unit of the resource is constrained to be used serially. Once a unit of a non-preemptable resource is allocated to a job, other jobs needing the unit must wait until the job completes its use. If jobs can use every unit of a resource in an interleaved way, the resource is preemptable. **A lock on a data object is an example of a non-preemptable resource**.

This does not mean that the job is non-preemptable on other resources or on the processor.

The transaction can be preempted on the processor by other transactions not waiting for the locks.

ii. Resource Graph:

- A resource graph describes the configuration of resources.
- There is a vertex R_i for every processor or resource R_i in the system.
- We can treat resources and processors similarly for the sake of convenience.
- The attributes of the vertex are the parameters of the resource.

- The *resource type* of a resource tells us whether the resource is a processor or a passive resource, and its *number* gives us the number of available units.
- Edges in resource graphs represent the relationship among resources.
- Using different types of edges we can describe different configurations of the underlying system.
- **There are 2 types of edges in resource graphs**
- An edge from vertex R_i to vertex R_k can mean that R_k is a component of R_i. E.g. a memory is part of a computer and so is a monitor.
- This edge is an ***is-a-part-of edge*.**
- **The subgraph containing all the is-a-part-of edges is a forest.** The root of each tree represents a major component, with subcomponents represented by vertices. E.g. the resource graph of a system containing 2 computers consists of 2 trees. The root of each tree represents a computer with children of this vertex including CPUs etc.
- **The other type of edge in resource graphs are *accessibility edges***.
- Some edges in resource graphs represent connectivity between components.
- These edges are called *accessibility edges*.
- e.g. if there is a connection between two CPUs in the two computers, then each CPU is accessible from the other computer and there is an accessibility edge from each computer to the CPU of the other computer.
- Each accessibility edge may have several parameters. A parameter of an accessibility edge from a processor P_i to another P_k is the cost of sending a unit of data from a job executing on P_i to a job executing on P_k
- Some algorithms use such information to decide how to allocate jobs and resources to processors in a statically configured system.

Precedence constraints:

Data and control dependencies among jobs may constrain the order in which they can execute Such jobs are said to have *precedence constraints*. If jobs can execute in any order, they are said to be *independent.*

Example: Consider an information server. Before a query is processed and the requested information retrieved, its authorization to access the information must first be checked. The retrieval job

cannot begin execution before the authentication job completes. The communication job that forwards the information to the requester cannot begin until the retrieval job completes.

Precedence Graph and Task graph:

We use a partial order relation <, called a precedence relation, over the set of jobs to specify the precedence constraints among jobs.

A job J_i is a predecessor of another job j_k (and j_k is a successor of j_i) if j_k cannot begin execution until the execution of j_i completes. This is represented as $j_i < j_k$

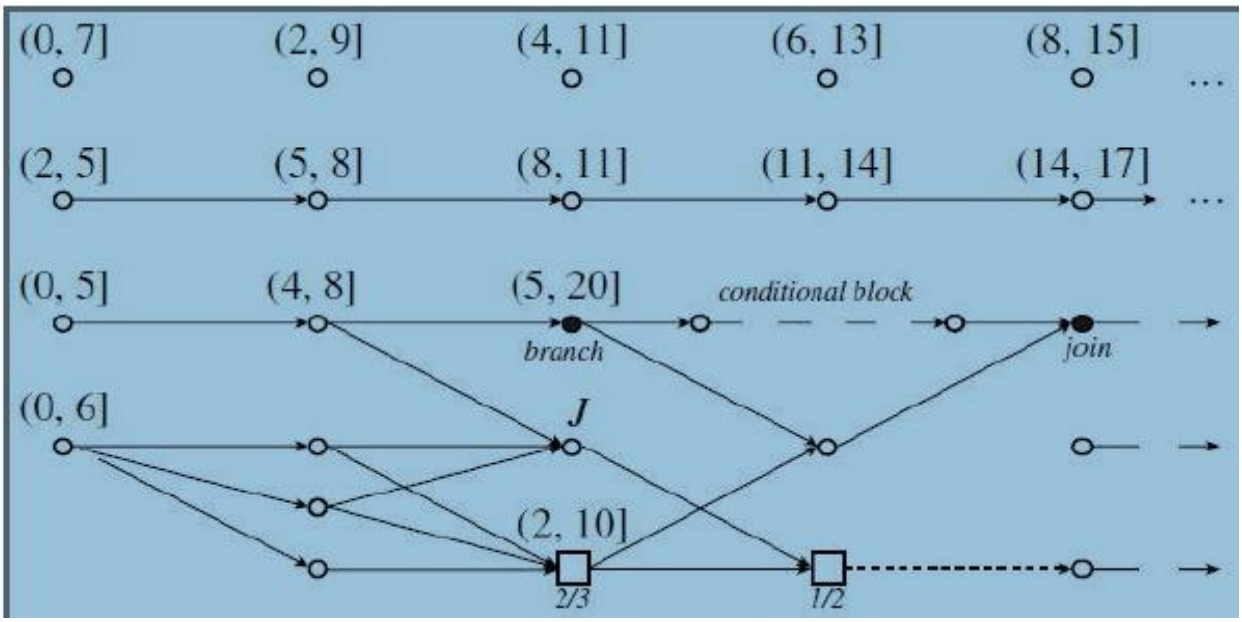
- Ji is an immediate predecessor of j_k (j_k is is an immediate successor of j_i) if $j_i < j_k$ and there is no other job j_j such that $j_i < j_j < j_k$
- Two jobs j_i and j_k are independent when neither $j_i < j_k$ nor $j_k < j_i$
- A job with predecessors is *ready* for execution when the time is at or after its release time and all of its predecessors are completed.

A precedence graph is a directed graph which represents the precedence constraints among a set of jobs J. Each vertex represents a job in J. There is a directed edge from vertex J_i to vertex J_k when the job J_i is an immediate predecessor of job J_k

Task graph:

- A task graph, which gives us a general way to describe an application system, is an extended precedence graph.
- As in a precedence graph, vertices represent jobs. They are shown as circles and squares (the distinction between them will come later).
- The numbers in the bracket above each job gives us its feasible interval.
- The edges in the graph represent dependencies among jobs.
- If all the edges are precedence edges representing precedence constraints then the graph is a precedence graph.

  Example of task graph :



The system shown in the sample task graph includes two periodic tasks.

Job row 1: Periodic task phase = 0, period = 2, relative deadline = 7 Independent jobs.

Job row 2: Periodic task phase = 2, period = 3, relative deadline = 3 Dependent jobs, precedence constraints. Hence the jobs must be executed in serial order.

Why Task Graph?

A task graph like the example is only really necessary when the system contains components which have complex dependencies like the subgraph below the periodic tasks.

Many types of interactions and communication among jobs are not captured by a precedence graph but can be captured by a task graph.

Unlike a precedence graph, a task graph can contain difference types of edges representing different types of dependencies.

Data Dependency:

Data dependency cannot be captured by a precedence graph. In many real-time systems jobs communicate via shared data. Often the designer chooses not to synchronize producer and consumer jobs. Instead the producer places the data in a shared address space to be used by the

consumer at any time. In this case the precedence graph will show the producer and consumer jobs as independent since they are apparently not constrained to run in turn.

- In a task graph, data dependencies are represented explicitly by data dependency edges among jobs.
- There is a data dependency edge from the vertex J_i to vertex J_k in the task graph if the job J_k consumes data generated by J_i or the job J_i sends messages to J_k
- A parameter of an edge from J_i to J_k is the volume of data from J_i to J_k.
- In multiple processor systems the volume of data to be transferred can be used to make decisions about scheduling of jobs on processors.

Other types of dependencies

The other dependencies are as follows.

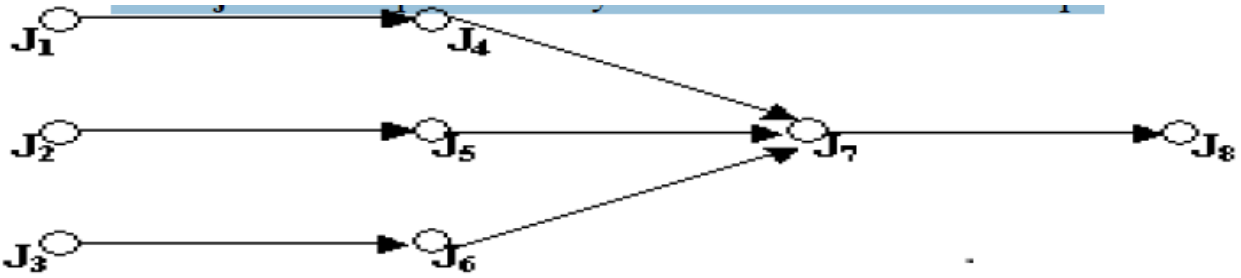
1. Temporal dependency
2. AND/OR precedence constraints
3. Conditional branches
4. Pipeline relationship.

1. Temporal dependency

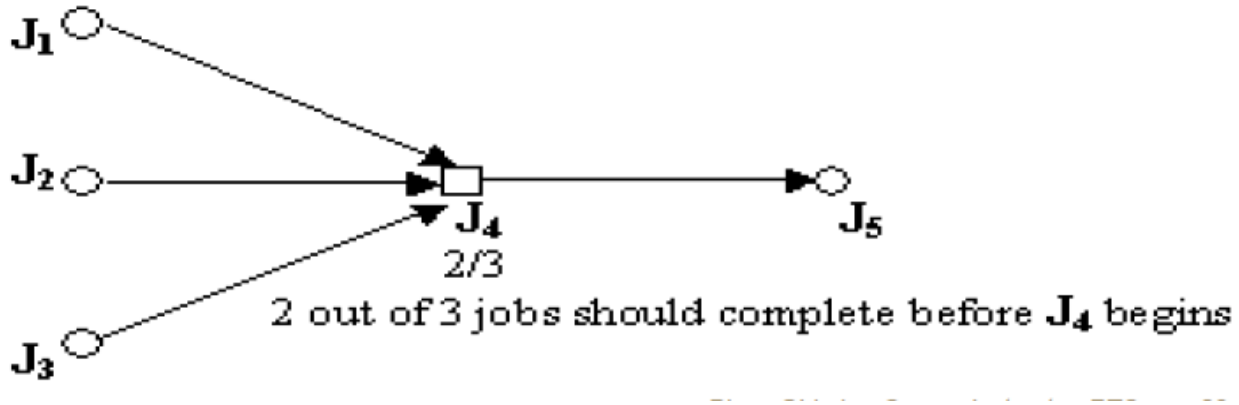
Some jobs may be constrained to complete within a certain amount of time relative to one another. We call the difference in the completion times of two jobs the *temporal distance* between them. Jobs are said to have a *temporal distance constraint* if their temporal distance must be no more than some finite value. Jobs with temporal distance constraints may or may not have deadlines. e.g. in a video frame lip synchronization delay is limited to 160 msec . In a task graph, temporal distance constraints among jobs are represented by temporal dependency edges. There is a temporal-dependency edge from a vertex *Ji* to a vertex *Jk* if the job *Jk* must be completed within a certain time after *Ji* completes.

2. AND/OR precedence constraints

If a job having more than one predecessor job needs that all the immediate predecessor must have been completed before its execution can begin, then such jobs are called AND jobs , having AND precedence constraint dependency among them . **AND jobs** are represented by **hollow circle** in Task Graph.

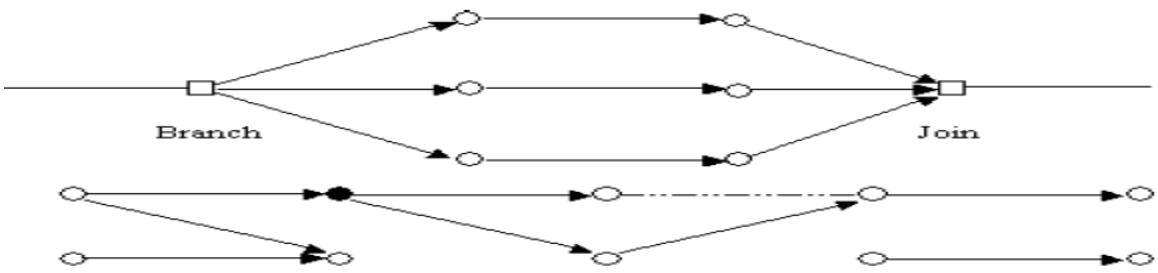


A **square node** is represented by a label **(*x*/*y*)** to indicate that the job can begin execution if **„*x*' out of „*y*'** predecessors complete execution. If job having more than one predecessor jobs can begin execution at or after its release time provided one or more of its predecessor have completed execution Jobs having such kind of dependency on its predecessor is called an **OR job** and is represented by a **square node.**



3. Conditional branches

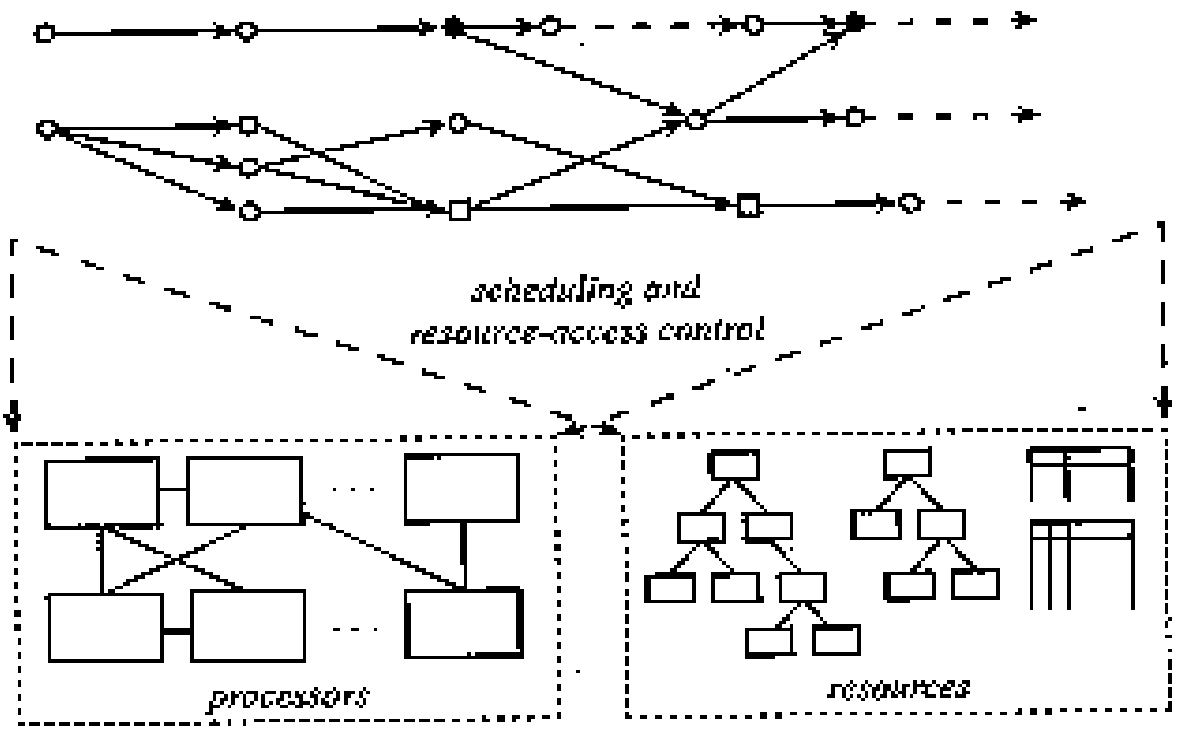
Sub graph beginning with branch job and ending with join job is called **conditional block**. l^k precedence graphs are needed to represent a system with *k* conditional block each having *l* branches .



4. Pipeline relationship

Each consumer granule can begin execution when the previous granule of job and the corresponding producer granule is completed. In the task graph a pipeline relationship among job is shown by pipeline edge , indicated by a **dotted edge.**

Scheduling hierarchy:



- The figure 'model of a real-time system' shows the three elements of our model of real-time systems.
- The application system is represented by
  - a task graph which gives the processor time and resource requirements of jobs, their timing constraints and dependencies
  - A resource graph describing the resources available to execute the application system, their attributes and rules governing their use

- And between these graphs are the scheduling and resource access-control algorithms used by the operating system.

Scheduler and Schedules:

- Jobs are scheduled and allocated resources according to a chosen set of scheduling algorithms and resource access-control protocols.
- The *scheduler* is a module that implements these algorithms.
- The scheduler assigns processors to jobs, or equivalently, assigns jobs to processors.
    - A *schedule* is an assignment by the *scheduler* of all the jobs in the system on the available processors.

A valid Schedule:

A valid schedule is the schedule satisfies the following conditions:

1. Every processor is assigned to at most one job at any time.
2. Every job is assigned at most one processor at any time.
3. No job is scheduled before its release time.
4. Depending on the scheduling algorithms used, the total amount of processor time assigned to every job is equal to its maximum or actual execution time.
5. All the precedence and resource usage constraints are satisfied.

Feasibility:

- A **valid schedule** is a ***feasible schedule*** if every job completes by its deadline and in general meets its timing constraints.
- A set of jobs is *schedulable* according to a scheduling algorithm if when using the algorithm the scheduler always produces a feasible schedule.

Optimality:

- A hard real-time scheduling algorithm is ***optimal*** if using the algorithm the scheduler always produces a feasible schedule if the given set of jobs has feasible schedules.
- If an optimal algorithm cannot find a feasible schedule, we can conclude that a given set of jobs cannot feasibly be scheduled by any algorithm.

Performance measures

The following are the performance measure of real time system.

- ***Miss rate***: percentage of jobs executed but completed late
- ***Loss rate***: percentage of jobs discarded
- ***Invalid rate***: sum of *miss* and *loss* rate.
- ***Makespan***: If all jobs have same release time and deadline then the completion time of last time is the makespan.
- Max or average response times
- Max or average tardiness/lateness
- ***Tardiness***: Zero if completion time <= deadline, otherwise equals (completion time - deadline).
- ***Lateness***: difference between completion time and deadline, can be negative if early.
- **Or another words**
- **Lateness *(L)*:** L = *Completion time* - *deadline*($L > 0$ if deadline is not met)
- **Tardiness *(E)*:** $E = \max\{L, 0\}$ ($E = 0$ if deadline is met)

Chapter-4

Commonly Used Approaches to Real-Time Scheduling

Three commonly used approaches to real-time scheduling are :

- Clock-driven
- Weighted round-robin
- Priority-driven

The weighted round-robin approach is mainly used for scheduling real-time traffic in high-speed switched networks. It is not ideal for scheduling jobs on CPUs.

Clock-Driven Approach

- Clock-driven scheduling is often called time-driven scheduling.
- When scheduling is *clock-driven*, decisions are made at specific time instants on what jobs should execute when.
- Typically in a system using 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 scheduler schedules the jobs according to this schedule at each scheduling decision time.
- Thus scheduling overhead at run-time is minimized.
- Scheduling decisions are usually made at regularly spaced time instants.
- One way to implement this is to use a hardware timer set to expire periodically which causes an interrupt which invokes the scheduler.
- When the system is initialized, the scheduler selects and schedules the jobs that will execute until the next scheduling decision time and then blocks itself waiting for the expiration of the timer.
- When the timer expires, the scheduler repeats these actions.

Round-Robin approach:

- The round-robin approach is commonly used for scheduling time-shared applications.
- When jobs are scheduled in a round-robin system, every job joins a first-in-first-out (FIFO) queue when it becomes ready for execution.
- The job at the head 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.
- When there are *n* ready jobs in the queue, each job gets one time slice in *n*, that is every *round*.
- Because the length of the time slice is relatively short (typically tens of milliseconds) the execution of each jobs begins almost immediately after it becomes ready.
- In essence, each job gets 1/*n*th share of the processor when there are *n* jobs ready for execution.
- This is why the round-robin algorithm is also known as the processor-sharing algorithm.

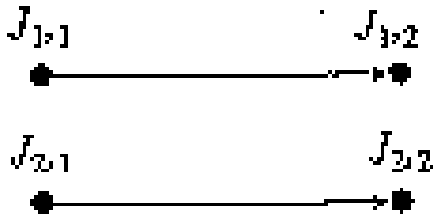
Weighted Round-Robin Approach:

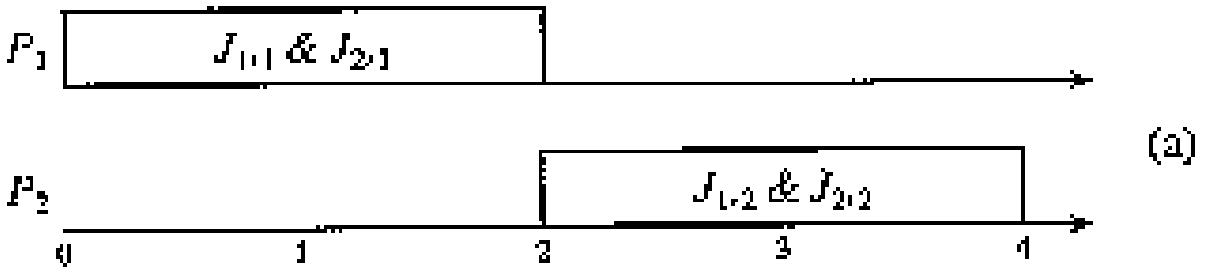
- The *weighted round-robin algorithm* has been used for scheduling real-time traffic in high-speed switched networks.
- Rather than giving all the ready jobs equal shares of the processor, different jobs may be given different *weights*.
- A job with weight *wt* gets *wt* time slices every round.
- The length of the round equals the sum of the weights of all the ready jobs.
- By adjusting the weights of jobs we can speed up or slow down the progress of each job.
- By giving each job a fraction of the processor, a round-robin scheduler delays the completion of every job.
- If round-robin scheduling is used to schedule precedence constrained jobs, the response time of a chain of jobs can get very large.
- For this reason, the weighted round-robin approach is unsuitable for scheduling such jobs.

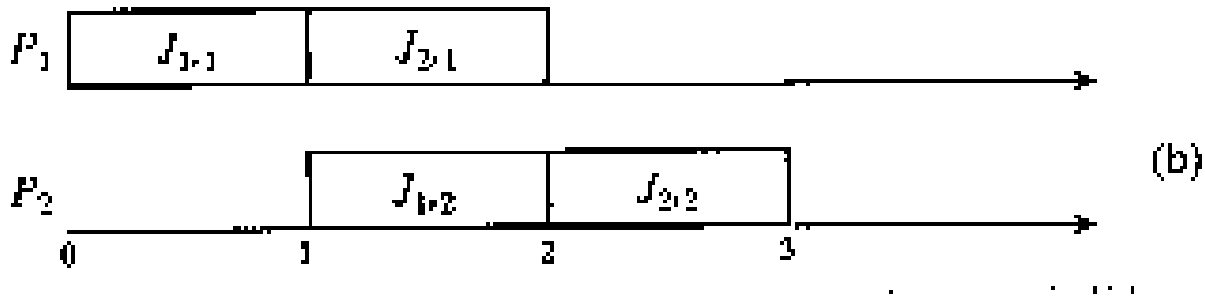
- However if a successor job is able to consume incrementally what is produced by a predecessor job, such as with a Unix pipe, weighted round-robin scheduling may be a reasonable approach.

For example consider two sets of jobs $\mathbf{J_1} = \{J_{1,1}, J_{1,2}\}$ and $\mathbf{J_2} = \{J_{2,1}, J_{2,2}\}$

- The release times of all jobs are 0
- The execution times of all jobs are 1
- $J_{1,1}$ and $J_{2,1}$ execute on processor P_1
- $J_{1,2}$ and $J_{2,2}$ execute on processor P_2







- We can see in the figure ‘round-robin scheduling’ that both sets of jobs complete approximately at time 4 if the jobs are scheduled in a weighted round-robin manner.
- In contrast, we can see that if the jobs on each processor are scheduled one after the other, one of the chains can complete at time 2 and the other at time 3.
- The weighted round-robin approach does not require a sorted priority queue, only a round-robin queue.

  This is a distinct advantage for scheduling message transmissions in ultrahigh-speed networks since fast priority queues are very expensive

Priority- Driven Approach:

- The term *priority-driven* algorithms refer to a class of scheduling algorithms that never leave any resource idle intentionally.
- With a priority-driven algorithm 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.
- Priority-driven algorithms are *event-driven*.
- Other commonly used terms for this approach are *greedy scheduling*, *list scheduling*, and *work-conserving scheduling*.
- A priority-driven algorithm is greedy because it tries to make locally optimal decisions.
- Leaving a resource idle while some job is ready to use the resource is not locally optimal.
- When a processor or resource is available and some job can use it to make progress, a priority-driven algorithm never makes the job wait.
- However there are cases where it is better to have some jobs wait even when they are ready to execute and the resources they require are available.
- The term list scheduling is also used because any priority-driven algorithm can be implemented by assigning priorities to jobs.
- 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.

- Hence a priority-driven scheduling algorithm is defined largely by the list of priorities it assigns to jobs.
- The priority list and other rules such as whether preemption is allowed, define the scheduling algorithm completely.
- Most non real-time scheduling algorithms are priority-driven.

Examples include:

- FIFO (first-in-first-out) and LIFO (last-in-first-out) algorithms which assign priorities to jobs based on their release times.
- SETF (shortest-execution-time-first) and LETF (longest-execution-time-first) algorithms which assign priorities based on job execution times.
- Because we can dynamically change the priorities of jobs, even round-robin scheduling can be thought of as priority-driven.
- The priority of the executing job is lowered to the minimum among all jobs waiting for execution after the job has executed for a time slice.

The following figure shows "examples of priority driven scheduling"

- The task graph is a precedence graph with all edges showing precedence constraints.
- The number next to the name of each job is its execution time.
- J_5 is released at time 4.
- All the other jobs are released at time 0.
- We want to schedule and execute the jobs on two processors P_1 and P_2.
- The schedulers of the processors share a common priority queue of ready jobs.
- Scheduling decisions are made whenever some job becomes ready for execution or some job completes.

The first schedule (a) and (b) shows the schedule of jobs on the two processors generated by the priority-driven algorithm following this priority assignment.

- At time 0, jobs J_1, J_2, and J_7 are ready for execution.
- They are the only jobs in the priority queue at this time.

- Since J_1 and J_2 have higher priorities than J_7 they are ahead of J_7 in the queue and hence are scheduled.
- At time 1, J_2 completes and hence J_3 becomes ready. J_3 is placed in the priority queue ahead of J_7 and is scheduled on P_2, the processor freed by J_2.
- At time 3, both J_1 and J_3 complete. J_5 is still not released. J_4 and J_7 are scheduled.
- At time 4, J_5 is released. Now there are three ready jobs. J_7 has the lowest priority among them so it is preempted and J_4 and J_5 have the processors.
- At time 5, J_4 completes. J_7 resumes on processor P_1.
- At time 6, J_5 completes. Because J_7 is not yet completed, both J_6 and J_8 are not yet ready for execution. Thus processor P_2 becomes idle.
- J_7 finally completes at time 8. J_6 and J_8 can now be scheduled on the processors.

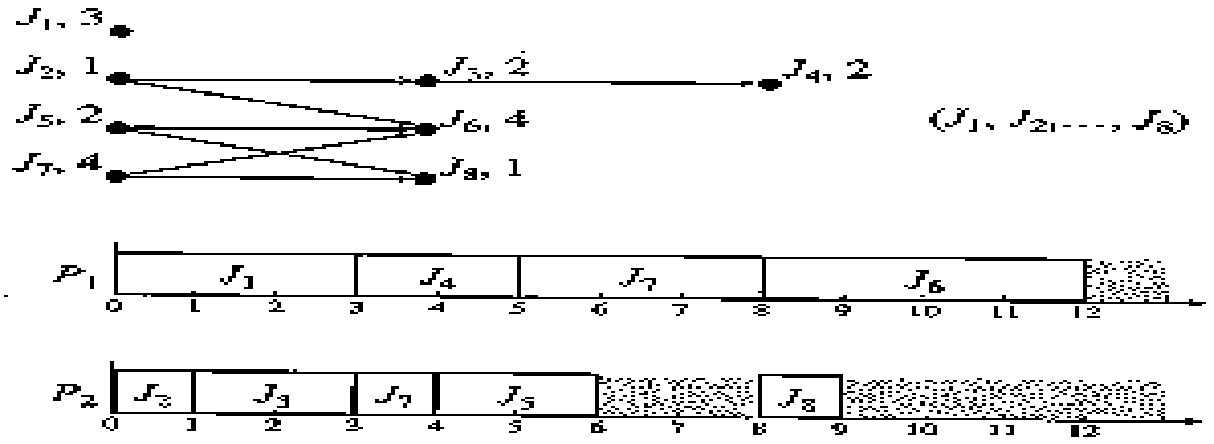


Fig. With preemption

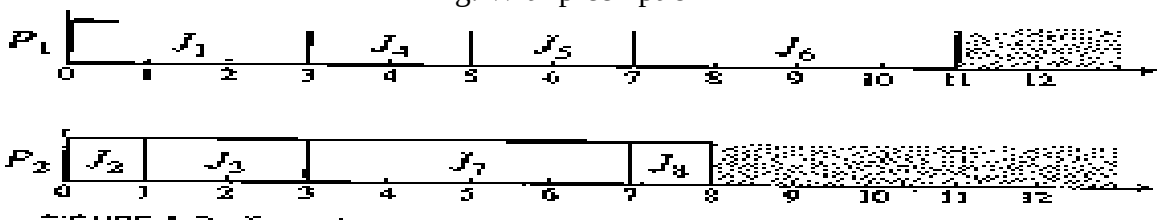


Fig without preemption

The figure "examples of priority driven scheduling

Dynamic vs. Static system:

- We have seen examples of jobs that are ready for execution being 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 a system is called a *dynamic system*, because jobs are *dynamically dispatched* to processors.
- In the example of priority scheduling we allowed each preempted job to resume on any processor.

We say a job *migrates* if it starts execution on a processor, is preempted, and later resumes on a different processor.

- An alternate approach to scheduling in multiprocessor and distributed systems is to partition the jobs in the system into subsystems and to allocate the subsystems statically to particular processors.
- In such systems, jobs are moved between processors only when the system must be reconfigured such as when a processor fails,we call such systems, *static systems*, because the system is *statically configured*.
- If jobs on different processors are dependent the schedulers on the processors must synchronize the jobs according to some synchronization and resource access-control protocol.
- Otherwise, jobs on each processor are scheduled by themselves

For example we could do a static partitioning of the jobs in our priority-driven scheduling example.

- Put J_1, J_2, J_3, J_4 on P_1 and the remaining jobs on P_2.
- The priority list is segmented into two parts:
    - (J_1, J_2, J_3, J_4) used by the scheduler of processor P_1
    - (J_5, J_6, J_7, J_8) used by the scheduler of processor P_2
- It is easy to see that the jobs on P_1 complete by time 8 and the jobs on P_2 complete by time 11.

- Also J_2 completes by time 4 while J_6 starts at time 6, thus the precedence constraint is satisfied.

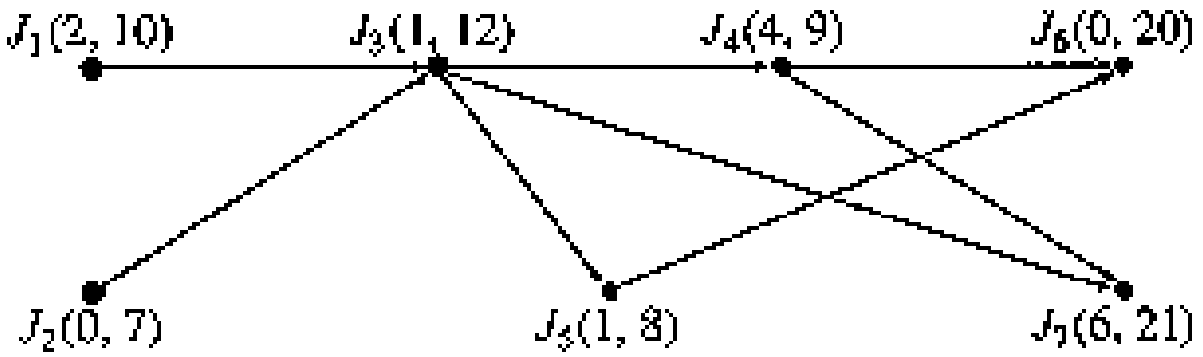
In this example the response of the static system is just as good as that of the dynamic system.

Effective release times and deadlines:

- The given release times and deadlines are sometimes inconsistent with the precedence constraints of the jobs.
  - i.e. the release time of a job may be later than that of its successors, and its deadline may be earlier than that of its predecessors.
- Rather than working with the given release times and deadlines, we first derive a set of effective release times and deadlines from these timing constraints together with the given precedence constraints.
- These derived timing constraints are consistent with the precedence constraints.
  - **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 its predecessors.
  - **Effective Deadline**:
    - The effective deadline of a job without a 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.

The effective release times of all the jobs can be computed in one pass through the precedence graph in $O(n^2)$ time where n is the number of jobs. Similarly for the effective deadlines.

- Consider the following example whose task graph is given in the following figure "example of effective timing constraints"



- The numbers in brackets next to each job are its given release time and deadline.
- Because J_1 and J_2 have no predecessors, their effective release times are their given release times, 2 and 0 respectively.
- Similarly the effective release time of J_3, J_4, J_5, J_6, J_7 ***are 2, 4, 2, 4, 6***

Effective deadlines

- J_6 and J_7 have no successors so their effective deadlines are their given deadlines, 20 and 21 respectively.
- Similarly the effective deadline of J_1, J_2, J_3, J_4, J_5 ***are 8, 7, 8, 9, 8***

Earliest-Deadline-First (EDF) Algorithm:

A way to assign priorities to jobs is on the basis of their deadlines. Earliest-Deadline-First (EDF) algorithm is based on the priority assignment whereby the earlier the deadline, the higher the priority. This algorithm is important because it is optimal when used to schedule jobs on a processor when preemption is allowed and there is no contention for resources.

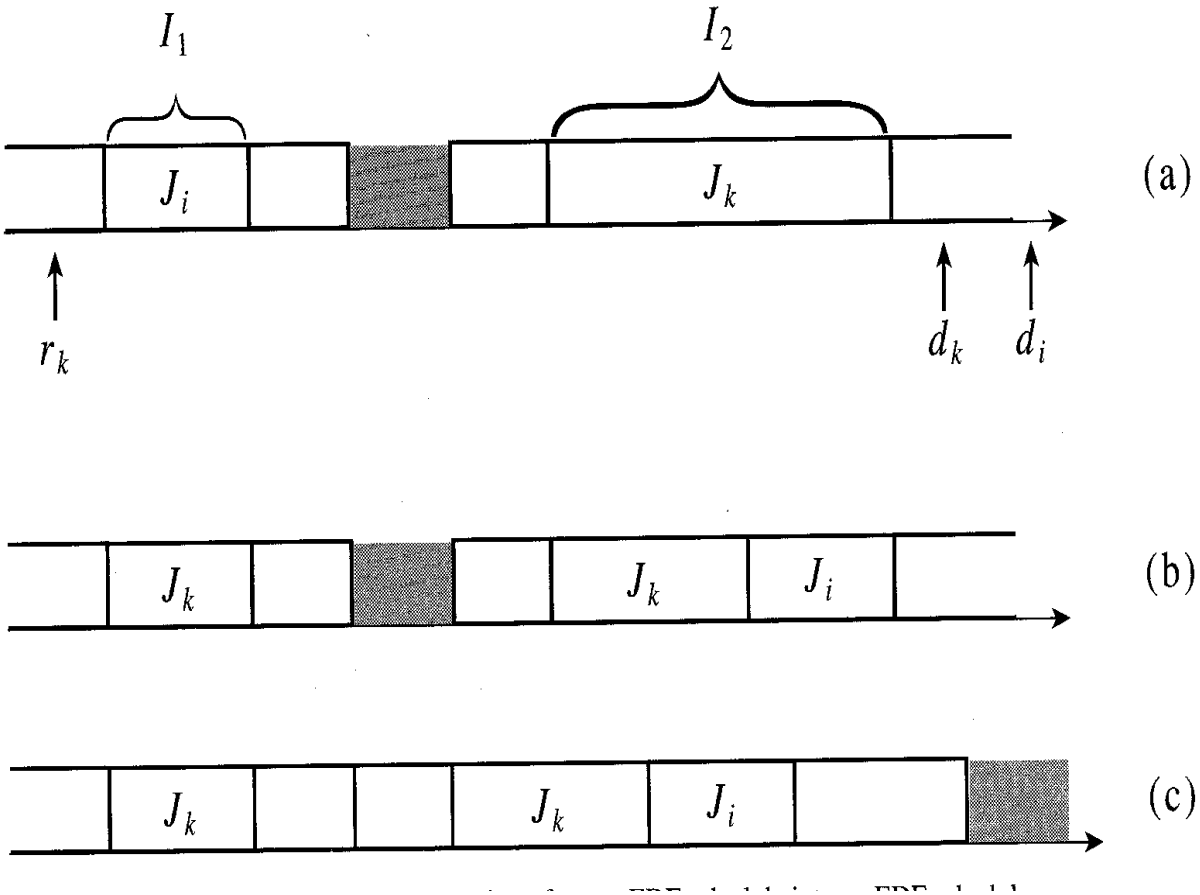
Theorem

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

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

To see how we can look at the figure "Transformation of a non-EDF schedule into an EDF schedule"

- Suppose that in a schedule parts of J_i and J_k are scheduled in intervals I_1 and I_2 respectively, and that deadline d_i of J_i is later than deadline d_k of J_k, but I_1 is earlier than I_2.



There are two cases:

- 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 2 jobs are already scheduled according to EDF.
- Second case
  - The release time r_k of J_k is before the end of I_1.

- Without loss of generality we can assume that r_k is no later than the beginning of I_1.

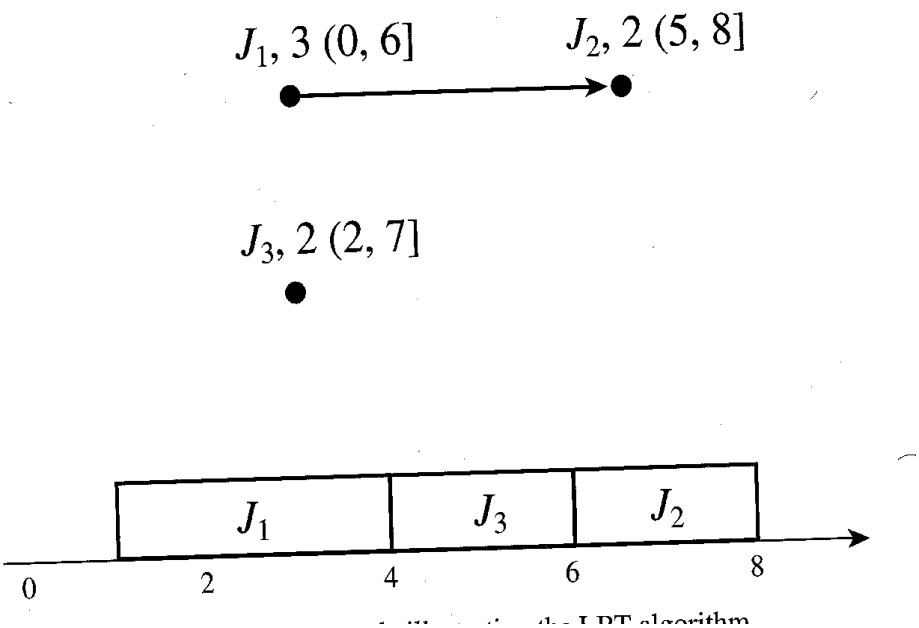
To transform the given schedule we swap J_i and J_k.

- If 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 always possible.
- We can do a similar swap if I_1 is longer than I_2.
- In which case we move the entire portion of J_k scheduled in I_2 to I_1 and place it before J_i and move the portion of J_i that fits in I_2 to the interval.
- The result of this swap is that these two jobs are now scheduled according to EDF.
- We repeat this transformation for every pair of jobs not scheduled according to EDF until no such pair exists.

The schedule so obtained may still not be an EDF schedule if some interval is left idle while there are jobs ready for execution but scheduled later.

- This is illustrated in part (b) of the figure.
- We can eliminate such an idle interval by moving one or more of these jobs forward into the idle interval and leave the interval where the jobs were scheduled idle.
- Clearly this is always possible.
- We repeat this until the processor never idles when there are jobs ready for execution.
- This is illustrated in part (c) of the figure.
- This only works when there is preemption.
- This is because the preemptive EDF algorithm can always produce a feasible schedule if such a schedule exists.

LRT Algorithm Example:



- In the above example, the number next to the job is the execution time and the feasible interval follows it.
- The latest deadline is 8, so time starts at 8 and goes back to 0. At time 8, J2 is "ready" and is scheduled. At time 7, J3 is also "ready" but because J2 has a later release time, it has a higher priority, so J2 is scheduled from 7 to 6.
- When J2 "completes" at time 6, J1 is "ready" however J3 has a higher priority so is scheduled from 6 to 4.

  Finally J1 is scheduled from 4 to 1.

Least-Slack-Time-First (LST) Algorithm:

- Another algorithm optimal for scheduling preemptive jobs on one processor is *Least-Slack-Time-First (LST)* also called *Minimum-Laxity-First (MLF).*
- At any time *t*, the slack (or laxity) of a job with deadline *d*, is equal to *d* - *t* minus the time required to complete the remainder of the job.
- Job J_1 from the LRT example is released at time 0 and has its deadline at time 6 and execution time 3.
- Hence its slack is 3 at time 0.
- The job starts to execute at time 0.

- As long as it executes its slack remains 3 because at any time before its completion its slack is 6 - *t* - (3 - *t*)
- Suppose J_1 is preempted at time 2 by J_3 which executes from time 2 to 4.
- During this interval the slack of J_1 decreases from 3 to 1.
- At time 4 the remaining execution time of J_1 is 1, so its slack is 6 - 4 - 1 = 1.
- The LST algorithm assigns priorities to jobs based on their slacks.
- The smaller the slack, the higher the priority

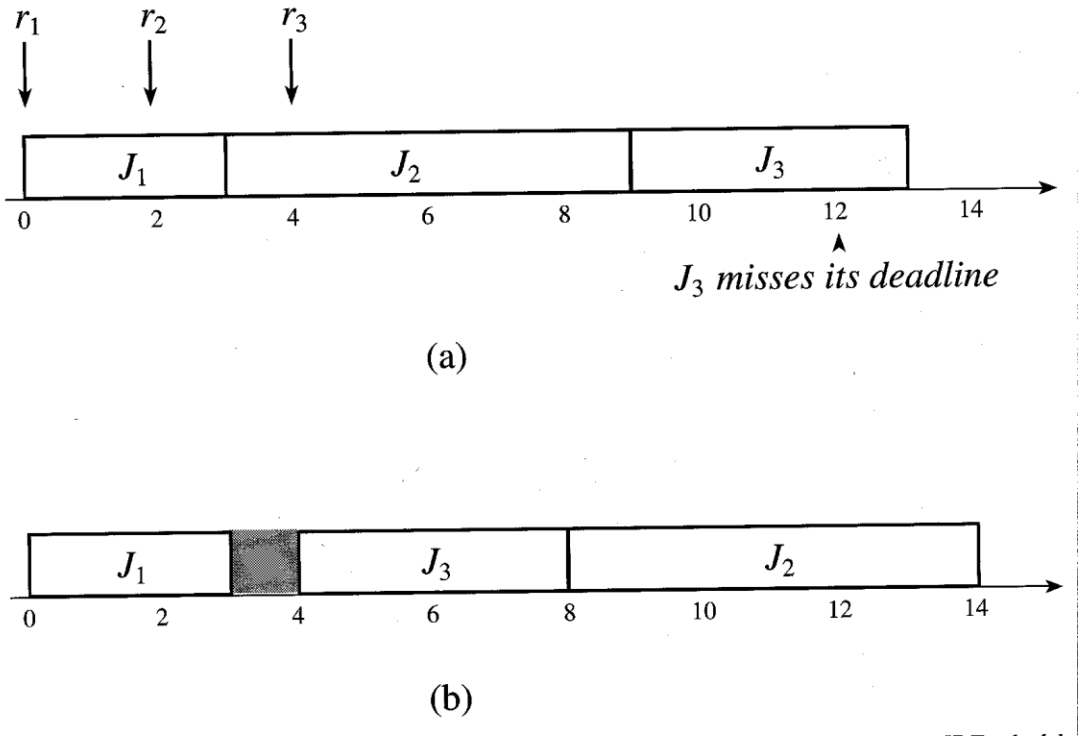
Non-optimality of EDF and LST:

- Do EDF and LST algorithms remain optimal if preemption is not allowed or there is more than one processor?

No!

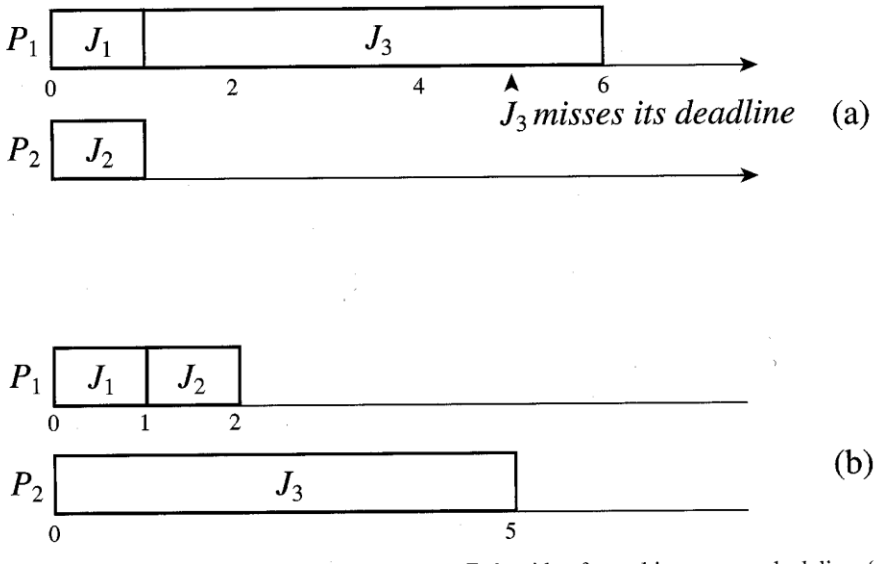
Consider the following 3 independent non-preemptable jobs, J_1, J_2, J_3, with release times 0, 2, 4 and execution times 3, 6, 4, and deadlines 10, 14, 12 respectively.

- Both EDF and LST would produce the infeasible schedule (a) whereas a feasible schedule is possible (b).
- Note that (b) cannot be produced by a priority driven algorithm



Non-optimality of EDF for multiprocessors:

- This time we have two processors and three jobs J_1, J_2, J_3, with execution times 1, 1, 5 and deadlines 1, 2, 5 respectively. All with release time 0.
- EDF gives the infeasible schedule (a) whereas LST gives a feasible schedule (b) but in general LST is also non-optimal for multiprocessors.



VALIDATING TIMING CONSTRAINTS IN PRIORITY-DRIVEN SYSTEMS

- Compared with the clock-driven approach, the priority-driven scheduling approach has many advantages:
  - Easy to implement
  - Often have simple priority assignment rules
  - If the priority assignment rules are simple, the run-time overheads of maintaining priority queues can be small
  - Does not require information about deadlines and release times in advance so it is suited to applications with varying time and resource requirements
- On the other hand a clock-driven scheduler:

  Requires information about release times and deadlines of jobs in advance in order to schedule them

Despite its merits, the priority-driven approach has not been widely used in hard real-time systems, especially safety-critical systems, until recently.

- The main reason for this is that the timing behaviour of a priority-driven system is not deterministic when job parameters vary.
- Thus it is difficult to validate that the deadlines of all jobs scheduled in a priority-driven manner indeed meet their deadlines when job parameters vary.

What is the *validation problem ?*

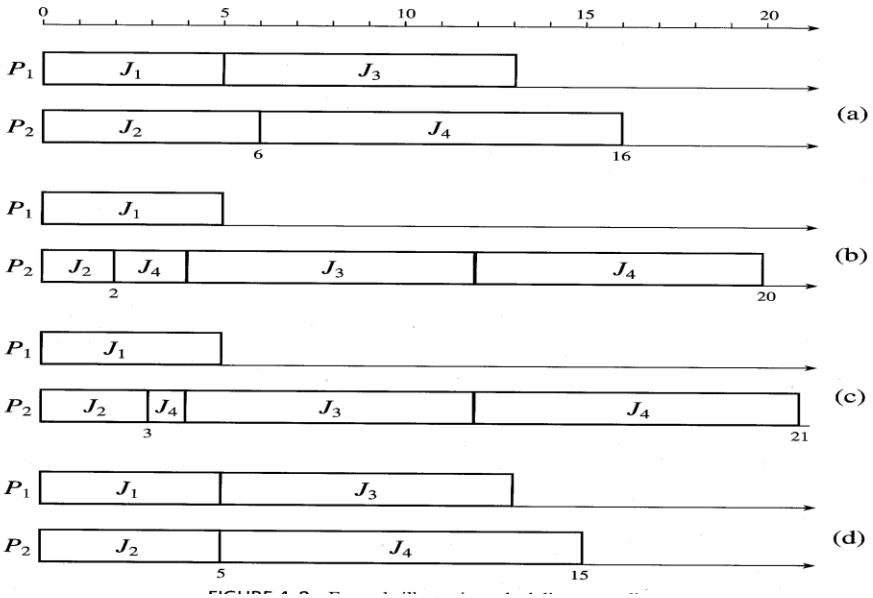
- Given a set of jobs, the set of resources available to the jobs, and the scheduling (and resource access-control) algorithm to allocate processors and resources to jobs, determine whether all the jobs meet their deadlines.

This is a very difficult problem to solve when you don't have all the information about all the jobs available in advance and can verify the complete schedule as in a clock-driven system

ANOMALOUS BEHAVIOUR OF PRIORITY-DRIVEN SYSTEMS

Consider the example in figure "illustrating scheduling anomalies"

	r_i	d_i	$[e_i^-, e_i^+]$
J_1	0	10	5
J_2	0	10	[2, 6]
J_3	4	15	8
J_4	0	20	10



This example illustrates why the validation problem is difficult with priority-driven scheduling and varying job parameters.

- Consider a simple system of four independent jobs scheduled on two processors in a priority-driven manner.
- There is a common priority queue and the priority order of jobs is J_1, J_2, J_3, J_4, with J_1 being of highest priority.
- It is a dynamic system.
- Jobs may be preempted but never migrated to another processor. (a common characteristic)
- The release times, deadlines and execution times or the jobs are provided in the table.
- The execution times of the jobs are fixed except for J_2 whose execution time is somewhere in the range [2,6]

Suppose we want to determine whether the system meets all the deadlines and whether the completion-time jitter of every job (i.e. the difference between the latest and earliest completion times of the job) is <= 4.

- Let us simulate the possible schedules to see what can happen.
- Suppose we schedule the jobs according to their priorities and try out J_2 with its maximum execution time 6 and also with its minimum execution time 2. (see the figure (a),(b))
- It seems OK for deadlines and for completion-time jitter.

Wrong! Have a look at cases (c) and (d)

- As far as J_4 is concerned, the worst-case schedule is (c) when execution time of J_2 is 3 and J_4 completes at 21 missing the deadline.
- The best-case schedule for J_4 is (d) when J_2 has execution time 5 and J_4 completes at time 15, however the completion time jitter exceeds its limit of 4.
- This is known as a scheduling anomaly, an unexpected timing behaviour of priority-driven systems.

PREDICTABILITY OF EXECUTIONS

To define predictability more formally, we call the schedule of **J** produced by the given scheduling algorithm when the execution time of every job has its maximum value the *maximal schedule* of **J**. Similarly, the schedule of **J** produced by the given scheduling algorithm when the execution time of every job has its minimum value is the *minimal schedule*. When the execution time of every job has its actual value, the resultant schedule is the *actual schedule* of **J**. So, the schedules in above figure (a) and (b) are the maximal and minimal schedules, respectively, of the jobs in that system, and all the schedules shown in the figure are possible actual schedules.

Let $s(Ji)$ is the (actual) start time of *Ji*. Let $s^+(Ji)$ and $s^-(Ji)$ be the start times of *Ji* according to the maximal schedule and minimal schedule of **J**, respectively. We say that *Ji* is *start-time predictable* if $s^-(Ji) \leq s(Ji) \leq s^+(Ji)$. As an example, for the job *J*4 in Figure above $s^-(J4)$ is 2. $s^+(J4)$ is 6. Its actual start time is in the range [2, 6]. Therefore, *J*4 is start-time predictable. Similarly, let $f(Ji)$ be the actual completion time (also called finishing time) of *Ji* according to the actual schedule of **J**. Let $f^+(Ji)$ and $f^-(Ji)$ be the completion times of *Ji* according to the maximal schedule and minimal schedule of **J**, respectively. We say that *Ji* is *completion-time predictable* if $f^-(Ji) \leq f(Ji) \leq f^+(Ji)$. The execution of *Ji* is *predictable*, or simply *Ji* is predictable, if *Ji* is both start-time and completion-time predictable. The execution behavior of the entire set **J** is predictable if every job in **J** is predictable. From figure above , we see that *f* –

($J4$) is 20, but $f + (J4)$ is 16. It is impossible for the inequality $20 \leq f(J4) \leq 16$ to hold. Therefore, $J4$ is not completion-time predictable, and the system is not predictable.

Priority-Driven Systems and Priority Inversion:

- Note that in our example of anomalous behaviour, it is possible to obtain valid schedules. It is just that our priority-driven algorithms could not do it.
- For instance, if one assumed that J4 was not preemptable, then case © would meet its deadline, but you would be violating the rule that says that the highest priority runnable job should be run.
- In this case J4 would continue running even though the higher priority J3 was able to run.
- This is called a *priority-inversion*.
- This happens when a lower priority job runs in preference to a higher priority job which has to wait.

OFF-LINE VERSUS ON-LINE SCHEDULING

This schedule is computed off-line before the system begins to execute, and the computation is based on the knowledge of the release times and processor-time/resource requirements of all the jobs for all times. When the operation mode of the system changes, the new schedule specifying when each job in the new mode executes is also precomputed and stored for use. In this case, we say that scheduling is (done) off-line, and the precomputed schedules are *off-line schedules*. This approach is possible only when the system is deterministic, meaning that the system provides some fixed set(s) of functions and that the release times and processor-time/resource demands of all its jobs are known and do not vary or vary only slightly.

Scheduling is done *on-line*, or that we use an *on-line scheduling algorithm*, if the scheduler makes each scheduling decision without knowledge about the jobs that will be released in the future; the parameters of each job become known to the on-line scheduler only after the job is released. The priority-driven algorithms described earlier and in subsequent chapters are on-line algorithms. Clearly, on-line scheduling is the only option in a system whose future workload is unpredictable. An on-line scheduler can accommodate dynamic variations in user demands and resource availability. Without prior knowledge about future jobs, the scheduler cannot make optimal scheduling decisions while a clairvoyant scheduler that knows about all future jobs can.

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 $T_i = (\varphi_i, p_i, e_i, D_i)$ 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 $\varphi_i = 0$, default relative deadline is the period $D_i = p_i$.

– Omit elements of the tuple that have default values.

Example:

i) $T_1 = (1, 10, 3, 6) \Rightarrow \varphi_1 = 1 , p_1 = 10, e_1 = 3 , D_1 = 6$

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

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

ii) $T_2 = (10, 3, 6) \Rightarrow \varphi_2 = 0 , p_2 = 10 , e_2 = 3 , D_2 = 6$

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

iii) $T3 = (10, 3) \Rightarrow \varphi 3 = 0, p3 = 10, e3 = 3, D3 = 10.$

$J_{3,1}$ 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)

□ Utilization = 1/4 + 1.8/5 + 1/20 + 2/20 = 0.76

□ Hyperperiod = LCM (4, 5, 20, 20) = 20

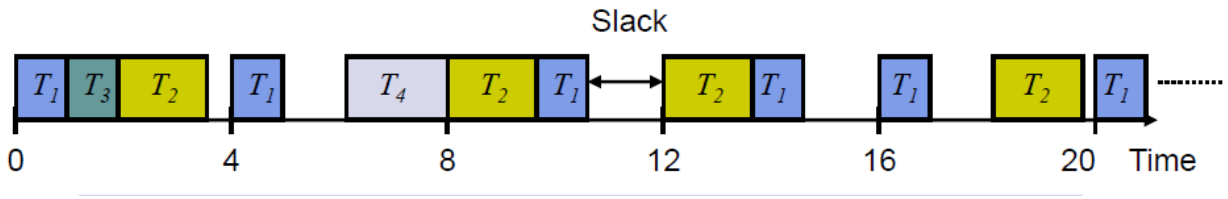


Fig . One possible schedule

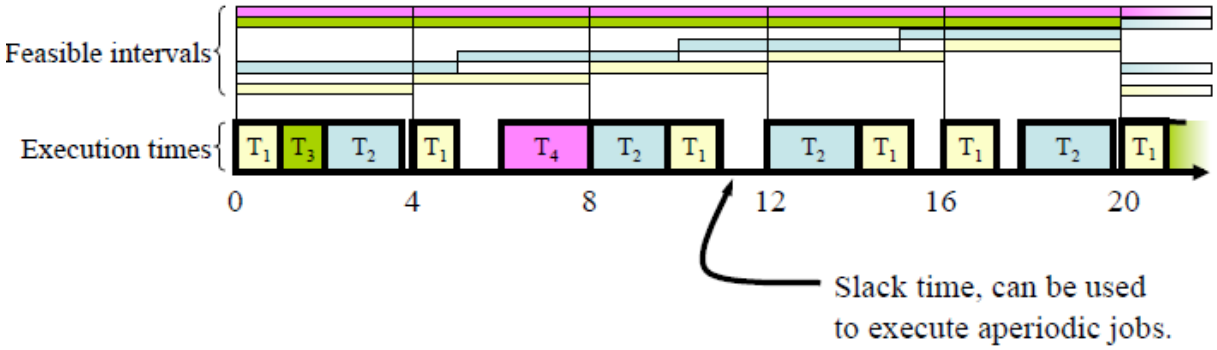
Slack time can be used to execute aperiodic jobs

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

• $\varphi = k \cdot f$ 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:

$f \geq \max(e1, e2, \ldots, en)$ ……………… (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):

$\exists\ i : \mathrm{mod}(pi, f) = 0$ ……………………………………………….(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:

$2*f - \gcd(pi, f) \leq Di$ for $i = 1, 2, \ldots, n$………………………………………………. (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 $\Rightarrow f \geq \max(1, 1.8, 1, 2) \geq 2$

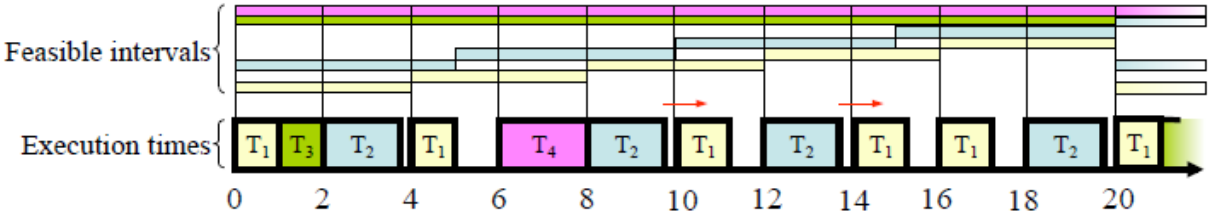
Eq.2 $\Rightarrow f \in \{ 2, 4, 5, 10, 20 \}$

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

$2f - \gcd(5, f) \leq 5$ (T2)

$2f - \gcd(20, f) \leq 20$ (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)

$$(1) \quad \forall i : f \geq e_i \qquad \Rightarrow f \geq 3$$
$$(2) \quad f \mid H \qquad \Rightarrow f = 2,3,4,5,6,10,..$$
$$(3) \quad \forall i : 2f - \gcd(pi, f) \leq Di \qquad \Rightarrow f = 2,3,4,5,6$$

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

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: Eq.1 $\Rightarrow f \geq 5$ but 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:

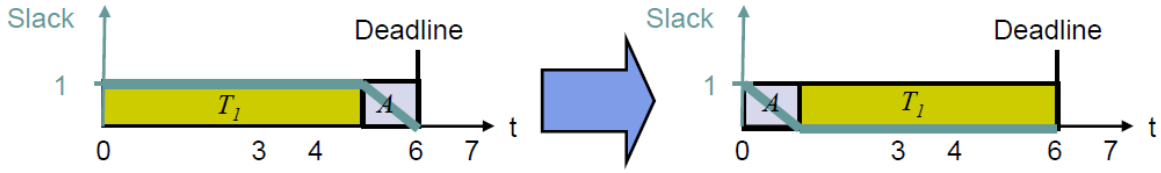
- So far aperiodic jobs have been scheduled in the background after all other job slices have been completed, it causes average response time is long.
- Average response time for aperiodic jobs can be improved by scheduling hard-real time jobs as late as possible without missing the deadline.

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 $\boldsymbol{k}$ be $\boldsymbol{x_k}$. The *slack* (time) available in the frame is equal to $\boldsymbol{f - x_k}$ 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.

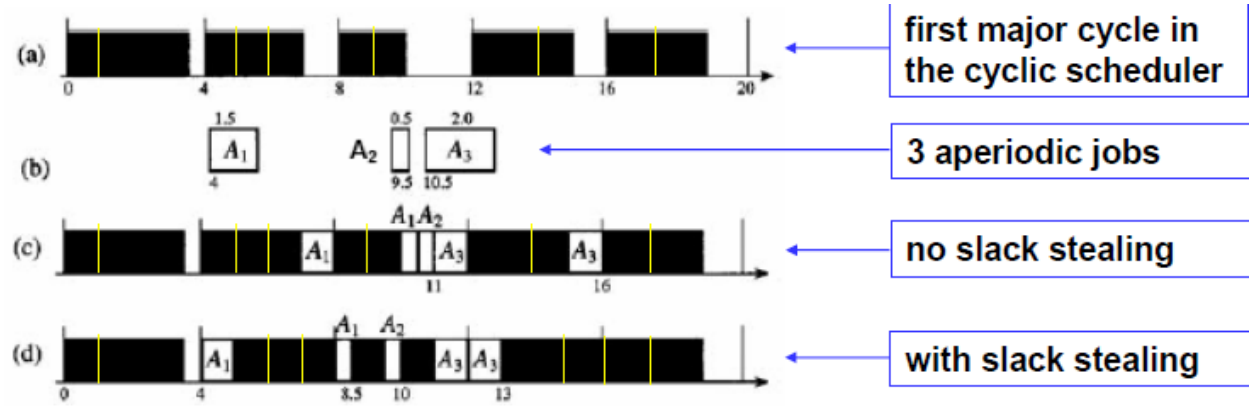
When an aperiodic job executes ahead of slices of periodic tasks, it consumes the slack in the frame. After *y* units of slack time are used by aperiodic jobs, the available slack is reduced to $\boldsymbol{f - x_k - y}$. The cyclic executive can let aperiodic jobs execute in frame $\boldsymbol{k}$ as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

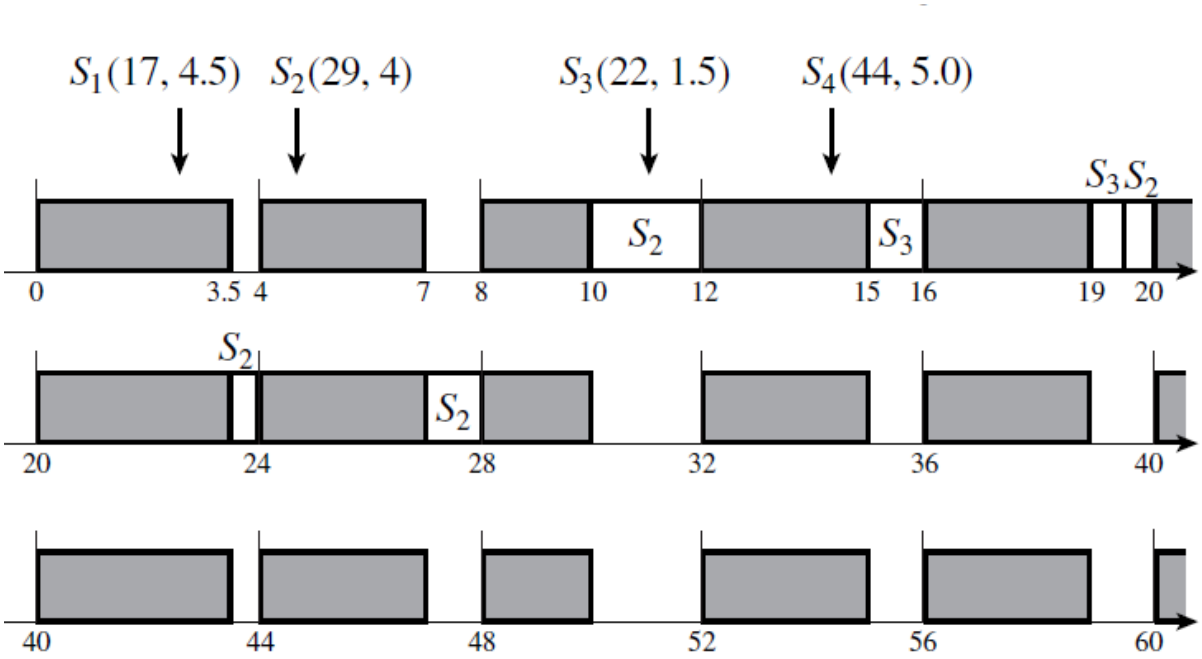
Conceptually, it is quite simple to do an acceptance test. To explain, let us suppose that at the beginning of frame t , an acceptance test is done on a sporadic job $S(d, e)$, with deadline d and (maximum) execution time e. (When it is not necessary to mention the deadline and execution time of the 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 lth 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, . . . 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.

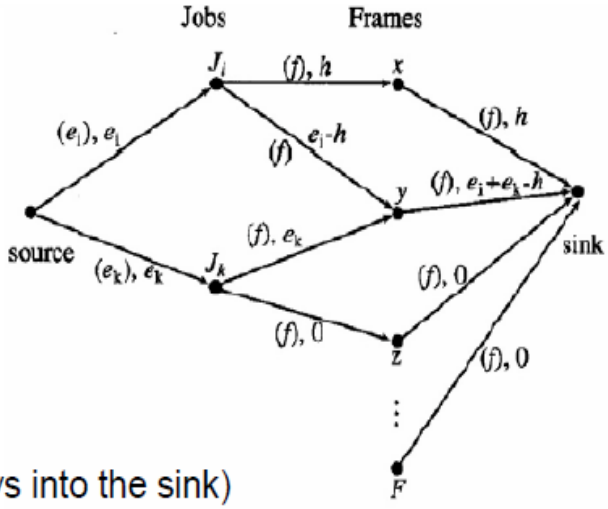
Network-Flow Graph:

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 $J1, J2, \ldots, JN$. 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 Ji*** representing each job *Ji*, for $i = 1, 2, \ldots, N$.
2. There is a ***frame vertex*** named *j* representing each frame *j* in the major cycle, for $j = 1, 2, \ldots, F$.
3. There are two special vertices named ***source*** and ***sink***.
4. There is a directed edge *(Ji , j)* from a job vertex *Ji* to a frame vertex *j* if the job *Ji* 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 *Ji* , and the capacity of this edge is the execution time *ei* 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:
  - ≤ the edge capacity,
  - ∑(flows into vertex) = ∑(flows out).

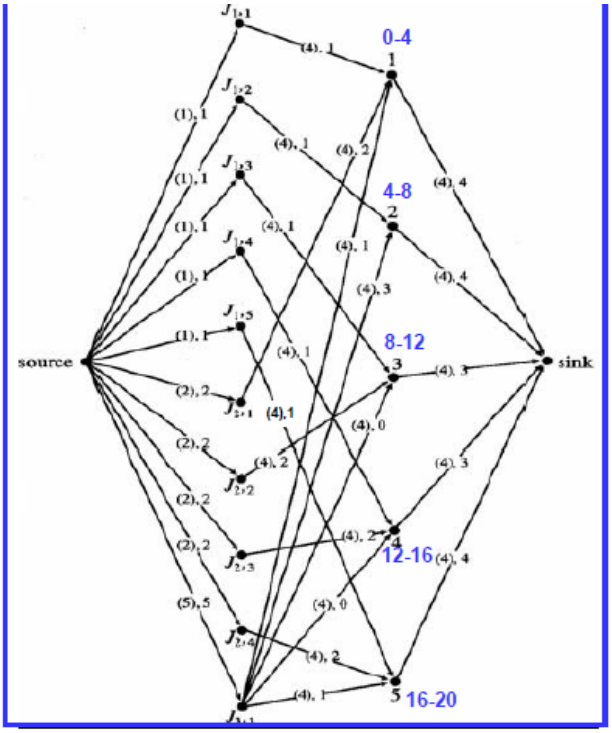


- Flow of a networkflow graph = ∑(all flows into the sink)
- **Problem**: Find the maximum flow of network-flow graph: time complexity of algorithm = $((N+F)^3)$.
- Maximum flow ≤ $\sum e_i$; if the set of flows of Jobs→Frames edges gives maximum flow = e_i, then they represent a feasible preemptive 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 ($J_{i,j}$,k) from job vertex $J_{i,j}$ to frame vertex k is drawn if job $J_{i,j}$ is scheduleable in frame k;
- its flow gives the amount of time in frame k allocated to job $J_{i,j}$.
- 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:

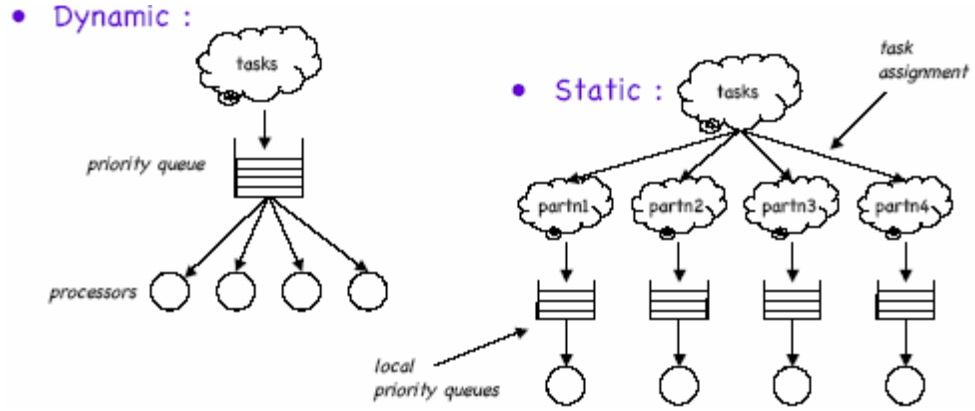
Assumptions

- 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.



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, Ti = (period or relative deadline, execution time) = (1,2ε) for 1 to m tasks (i.e. first tasks) and for m+1 th task T_{m+1} = (1+ ε, 1) .Here $D_i = p_i$ and phase = 0. Priorities are assigned in EDF basis.

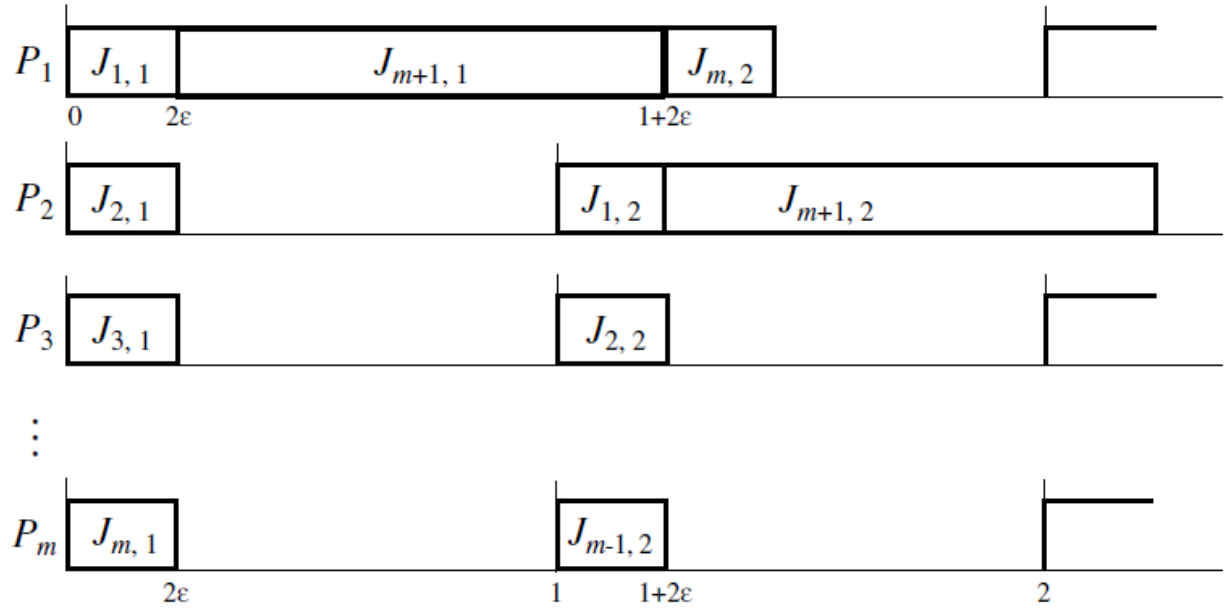


Fig. A dynamic EDF schedule on m processors

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 ε and hence misses its deadline.

Total utilization = m (2 ε/1) + 1/(1+ ε)= 2m ε/1 + 1/(1+ ε) = 2mε +1/(1+ ε).

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, $2m\varepsilon$, 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/period.

- The higher its rate, the higher the priority of the task.

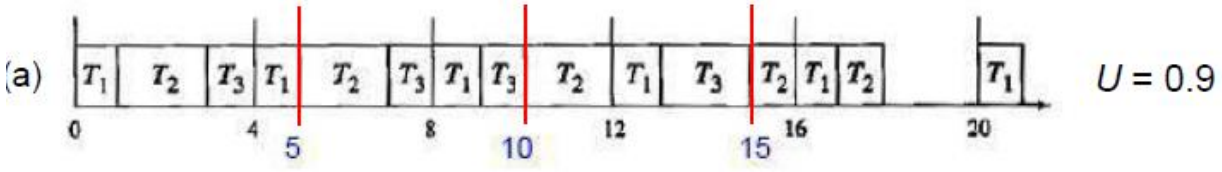
For example: (a): *T1* = (4,1); *T2* = (5,2); *T3* = (20,5) .

Rate(*T1*)=1/4= .25

Rate(T2)= 1/5=.2

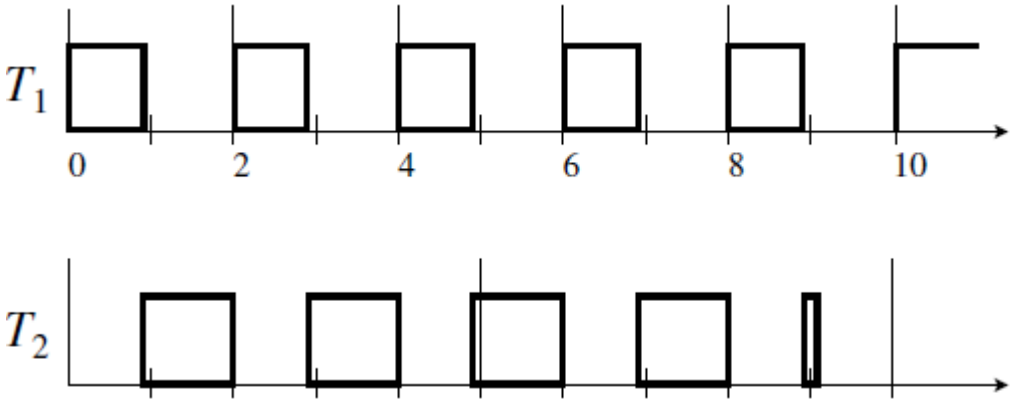
Rate(T3)=1/20=.05 , Therefore T1 has highest priority.

The RM schedule is :



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

The schedule is :

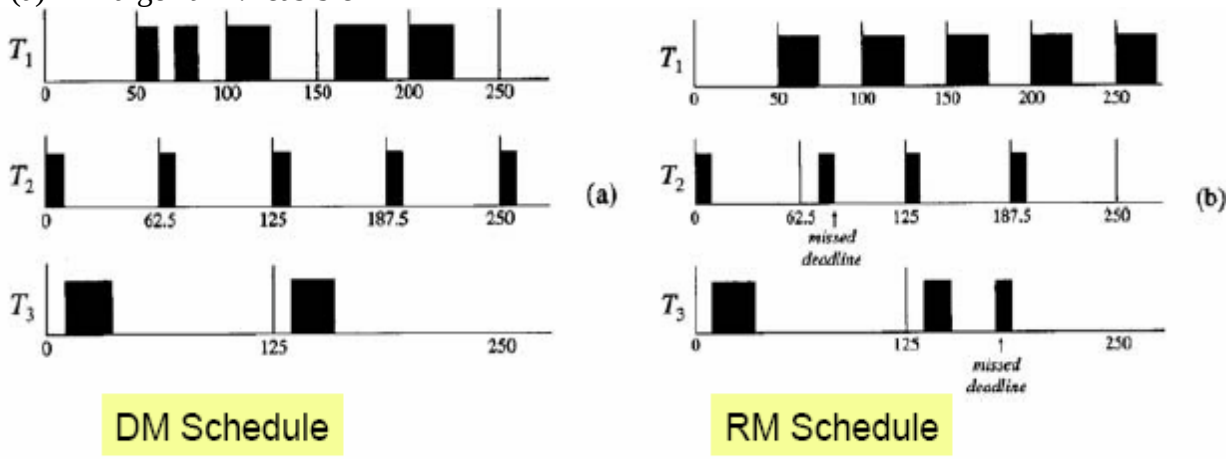


Deadline Monotonic:

- 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 => U = 0.86; H = 250.

-(a) DM algorithm: feasible



- (b) RM algorithm: not feasible => schedule not optimal

Earliest-Deadline-First (EDF):

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

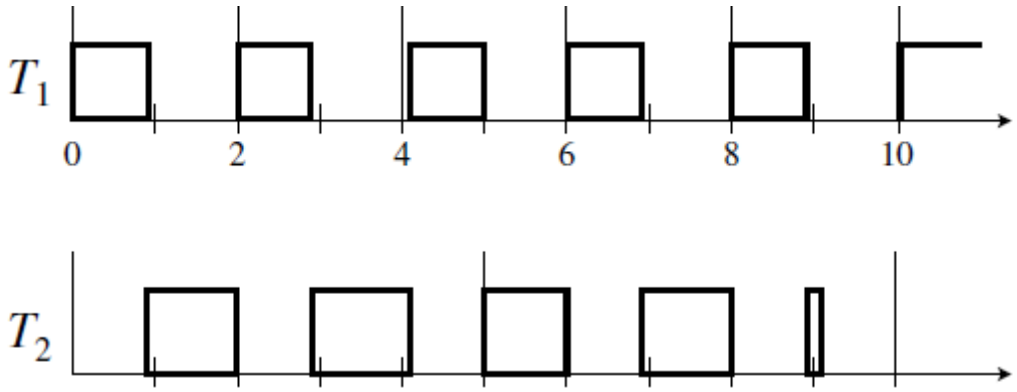


Fig. An earliest-deadline-first schedule of (2, 0.9) and (5, 2.3)

Description:

-At time 0, the first jobs $J1,1$ and $J2,1$ of both tasks are ready. The (absolute) deadline of $J1,1$ is = r+D=0+2= 2 while the deadline of $J2,1$ is= 0+5=5. Consequently, $J1,1$ has a higher priority and executes. When $J1,1$ completes, $J2,1$ 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 $Di < pi$ for some task Ti , $\Delta > U$. 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 feasiblly 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.

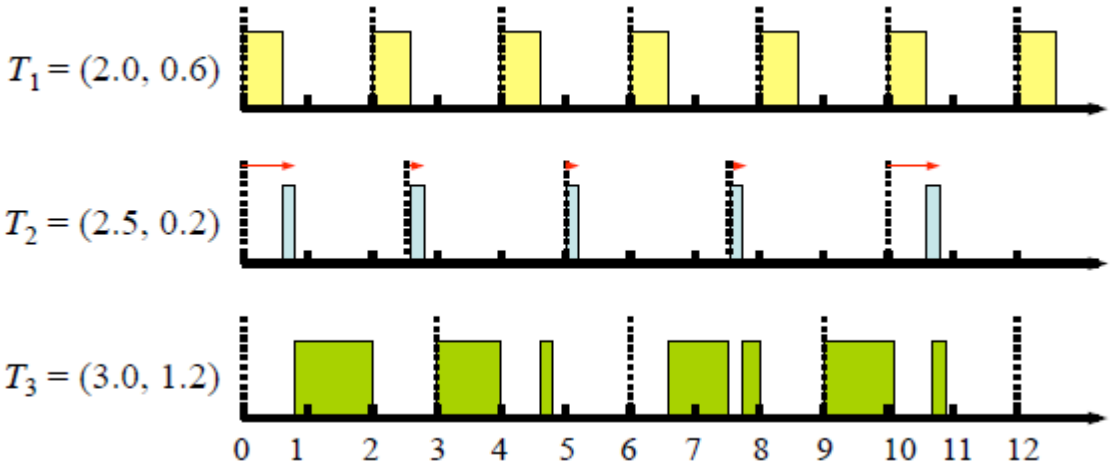
Critical Instant:

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 Ti released at a critical instant is called the maximum (possible) response time, and is denoted by Wi

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 in

stant of any task *Ti* occurs when one of its job *Ji,c* is released at the same time with a job in every higher-priority task, that is, *ri,c* = *rk,lk* for some *lk* for every *k* = 1, 2, . . . , *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

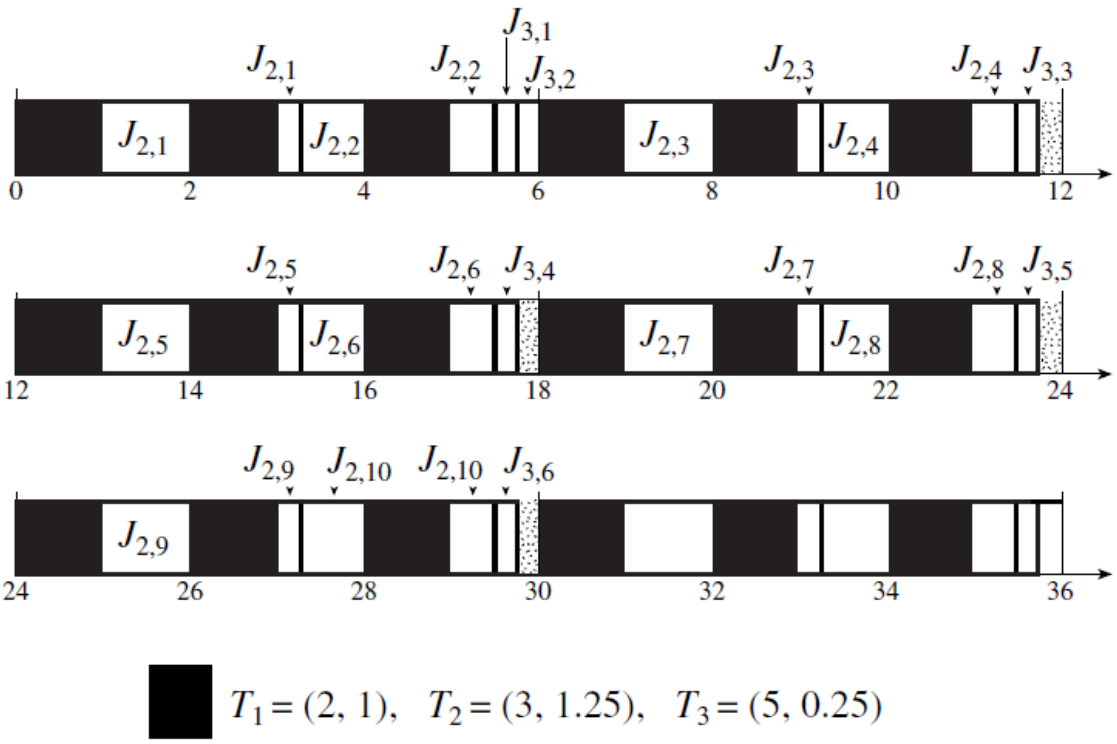


Fig. illustrating Busy Intervals

A *level-π_i busy interval (*$t0$, t] begins at an instant $t0$ 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 $t0$ when all the jobs in $\mathbf{T}i$ released since $t0$ are complete. In other words, in the interval *(*$t0$, 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 $T1$ = *(*2, 1*)*, $T2$ = *(*3, 1.25*)*, and $T3$ = *(*5, 0.25*)* in the first hyperperiod. The filled rectangles depict where jobs in $T1$ are scheduled. The first busy intervals of all levels are in phase. The priorities of the tasks are $\pi1 = 1$, $\pi2 = 2$, and $\pi3 = 3$, 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 $T1$ and $T2$. 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 $T3$ is scheduled. When this job completes at 5.75, the second job in $T3$ 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 $T3$ 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 $T1$ in order of decreasing priority. For the purpose of determining whether a task Ti 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 *Ki* for a job in *Ti* 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 $\{pi\}$ and $\{ei\}$ 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.

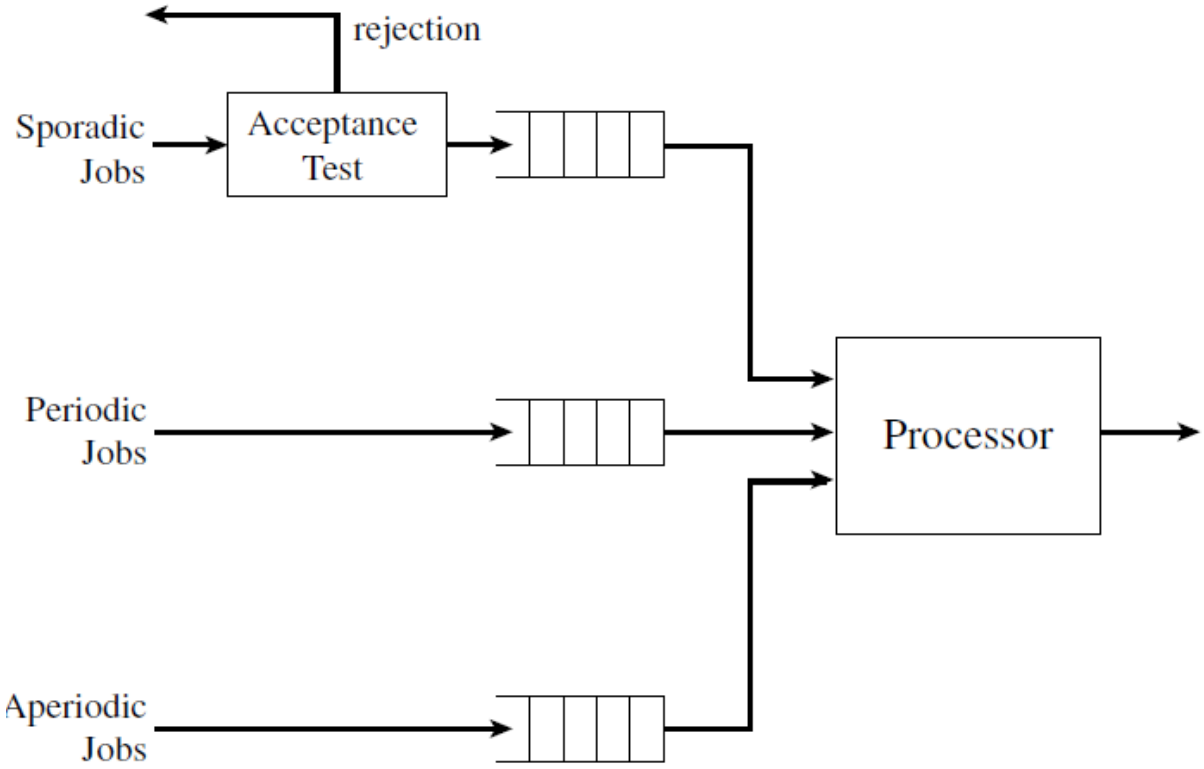


Fig. Priority queues maintained by Operating System

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 empy;
-The server is backlogged when the aperiodic job queue is not empy.

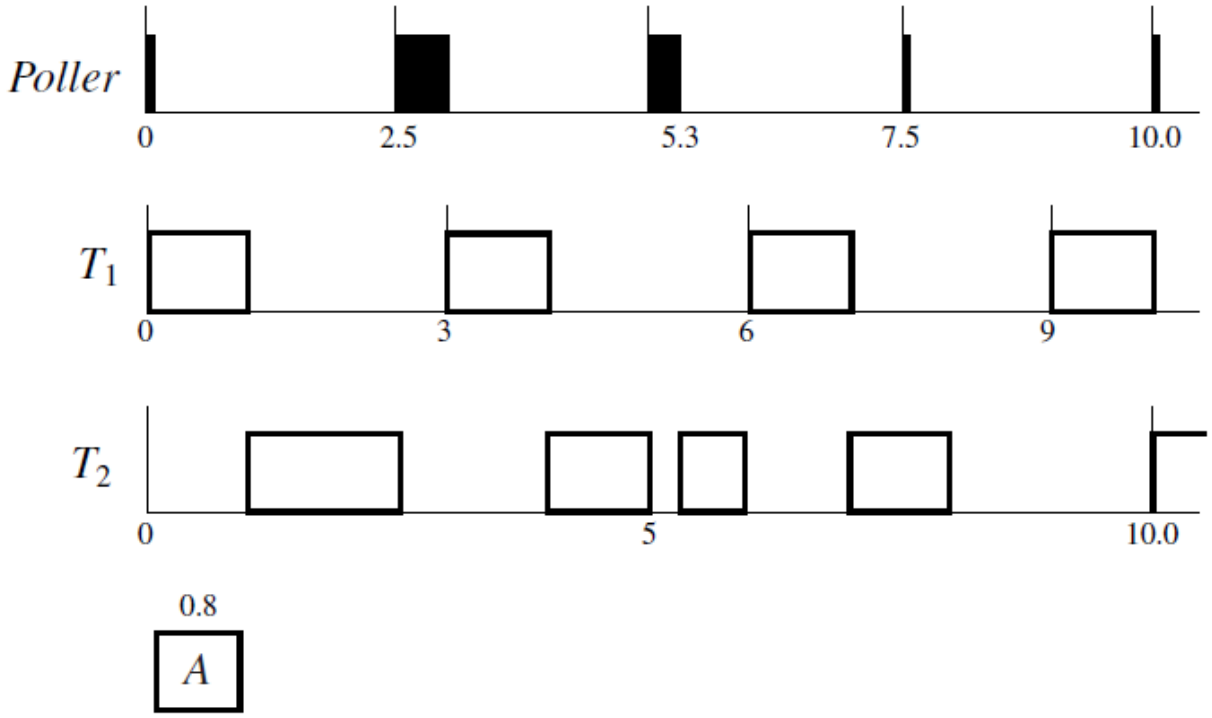


Fig.Polling for $T1 = (3, 1)$, $T2 = (10, 4)$, poller = $(2.5, 0.5)$ and A: r =0.1, e= 0.8

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 empy, 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 *es* at time instants *kpk*, for $k = 0, 1, 2, \ldots$.

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 *T*1 = *(*2.0, 3.5, 1.5*)* and *T*2 = *(*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.

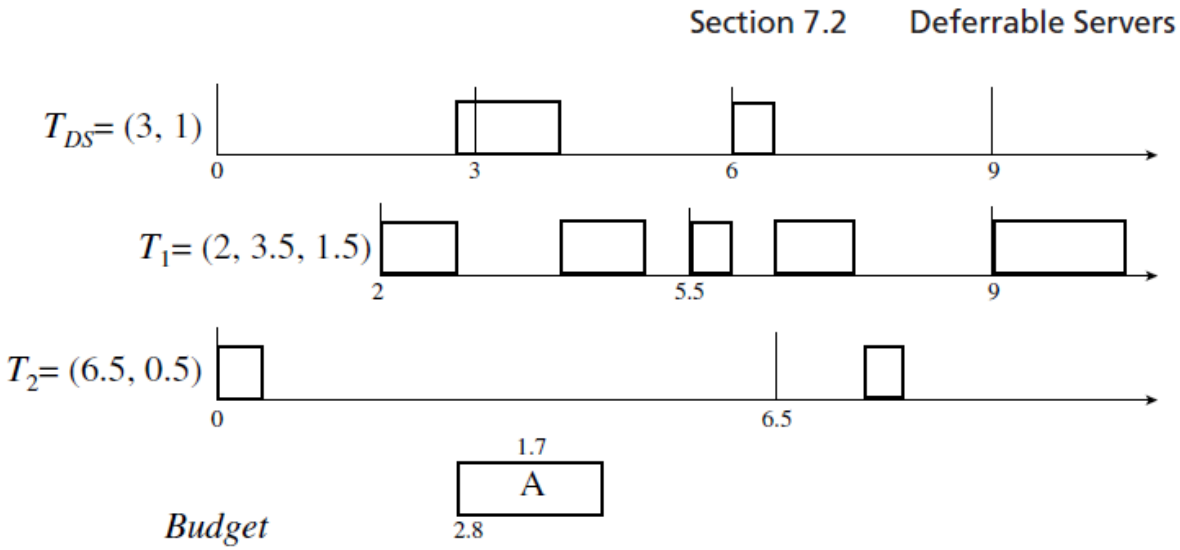
2. Immediately before the replenishment time 3.0, its budget is equal to 0.8. This 0.8 unit is lost at time 3.0, but the server acquires a new unit of budget. Hence, the server continues

to execute.

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

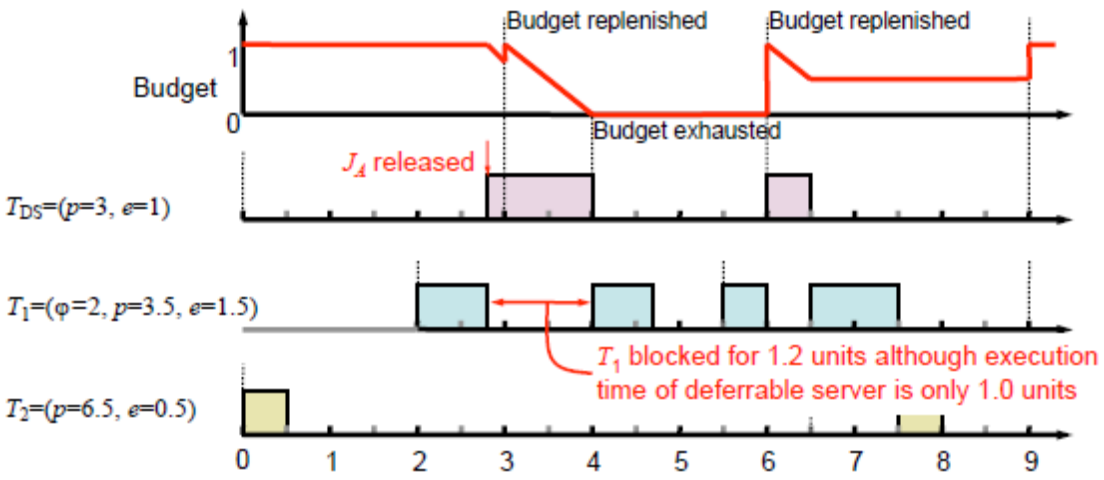
4. At time 6.0, its budget replenished, the server resumes to execute *A*.

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.



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* = ∞ 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)**

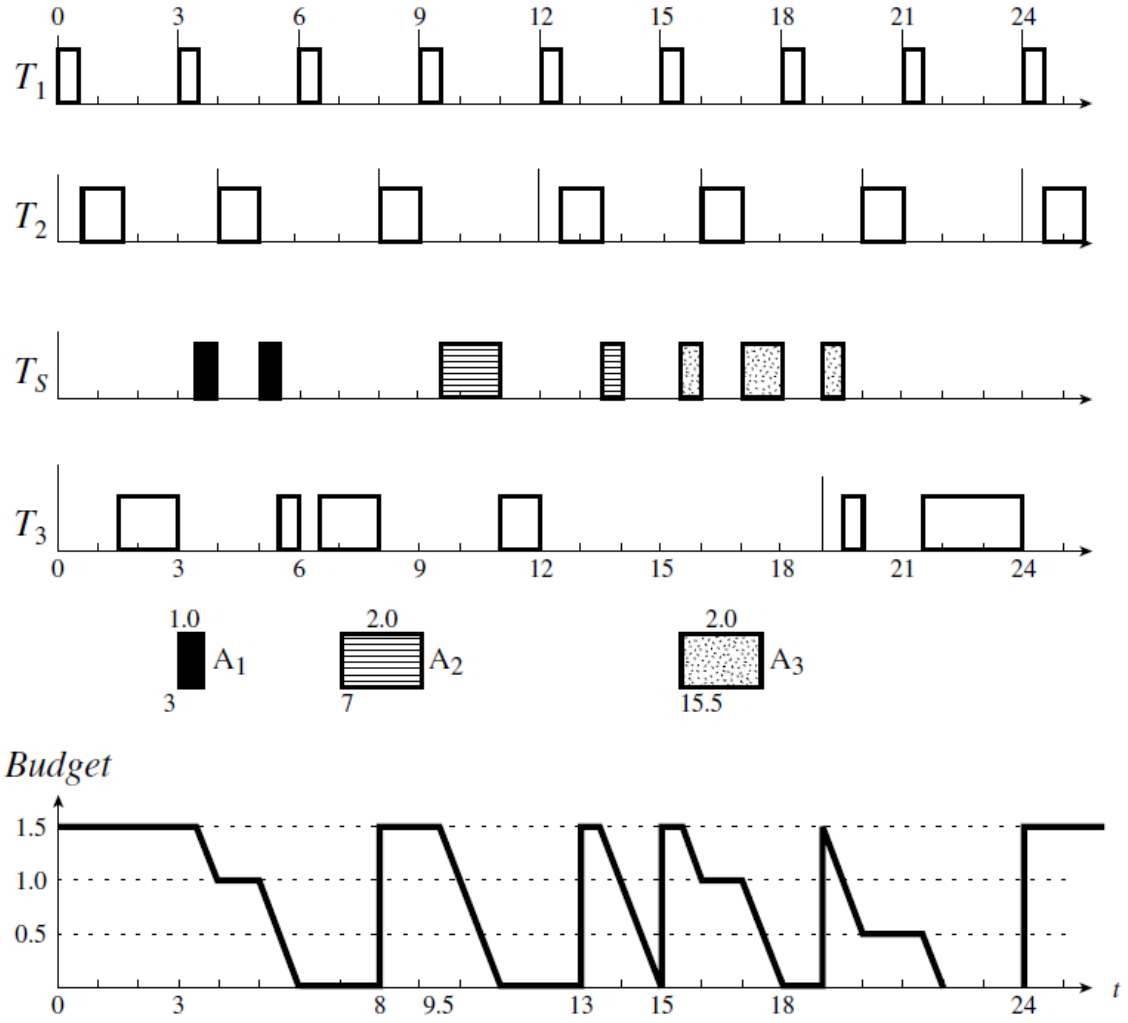


FIGURE 7–8 Example illustrating the operations of a simple sporadic server: $T_1 = (3, 0.5)$, $T_2 = (4, 1.0)$, $T_3 = (19, 4.5)$, $T_s = (5, 1.5)$.

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 (C1) 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) (C2).
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 (R3), 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 R3b.
10. The behavior of the later segment also obeys the above stated rules. In particular, rule R3b 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 infinitesmally small time slice of length proportional to the server size. Clearly, infinitesmally 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 *Ji* that has release time *ri* , maximum execution time *ei* and deadline *di* is the ratio *ei /(di − ri).* A sporadic job is said to be *active* in its feasible interval *(ri , di*]; 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*.

Replinishment 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 $\tilde{u}_s$

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

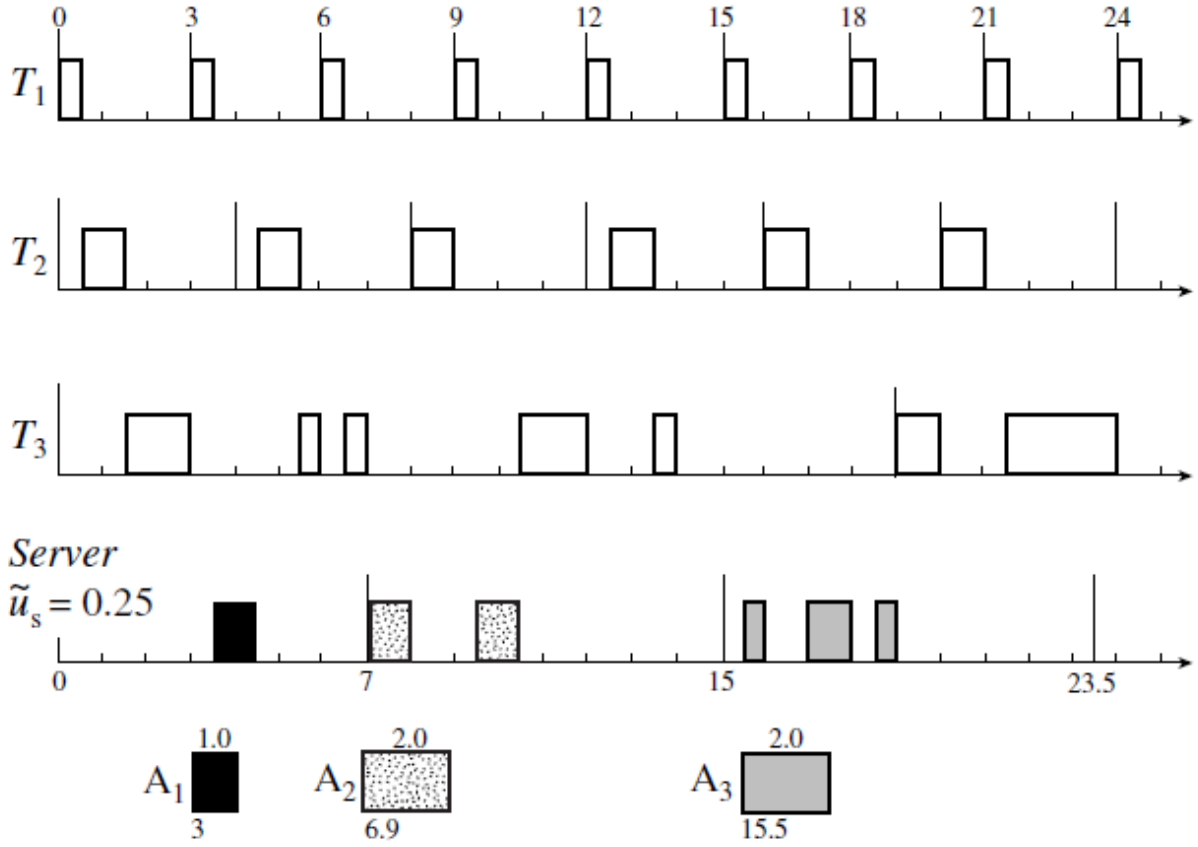
R2 When an aperiodic job with execution time e arrives at time t to an empty aperiodic job queue,

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

R3 At the deadline d of the server,

- **(a)** if the server is backlogged, set the server deadline to $d + e/\tilde{u}_s$ 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 BANDWITH 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 $\tilde{u}_s$

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 $\max(d, t) + e/\tilde{u}_s$ 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 $d + e/\tilde{u}_s$, 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.

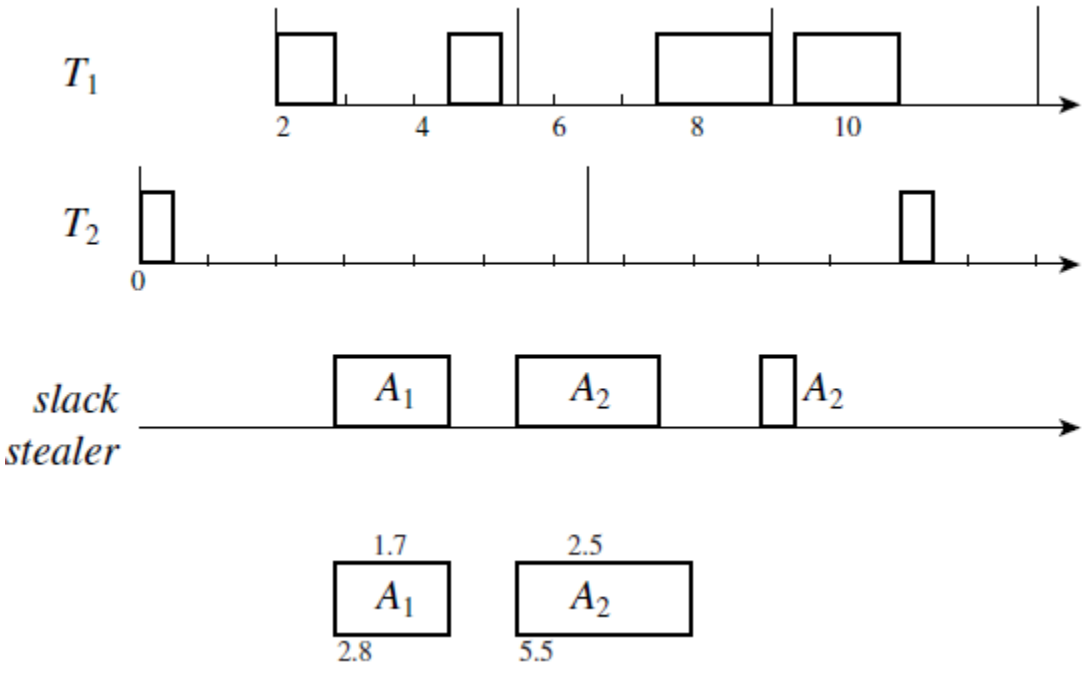


Fig Operation of stack stealer in deadline driven system

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.*

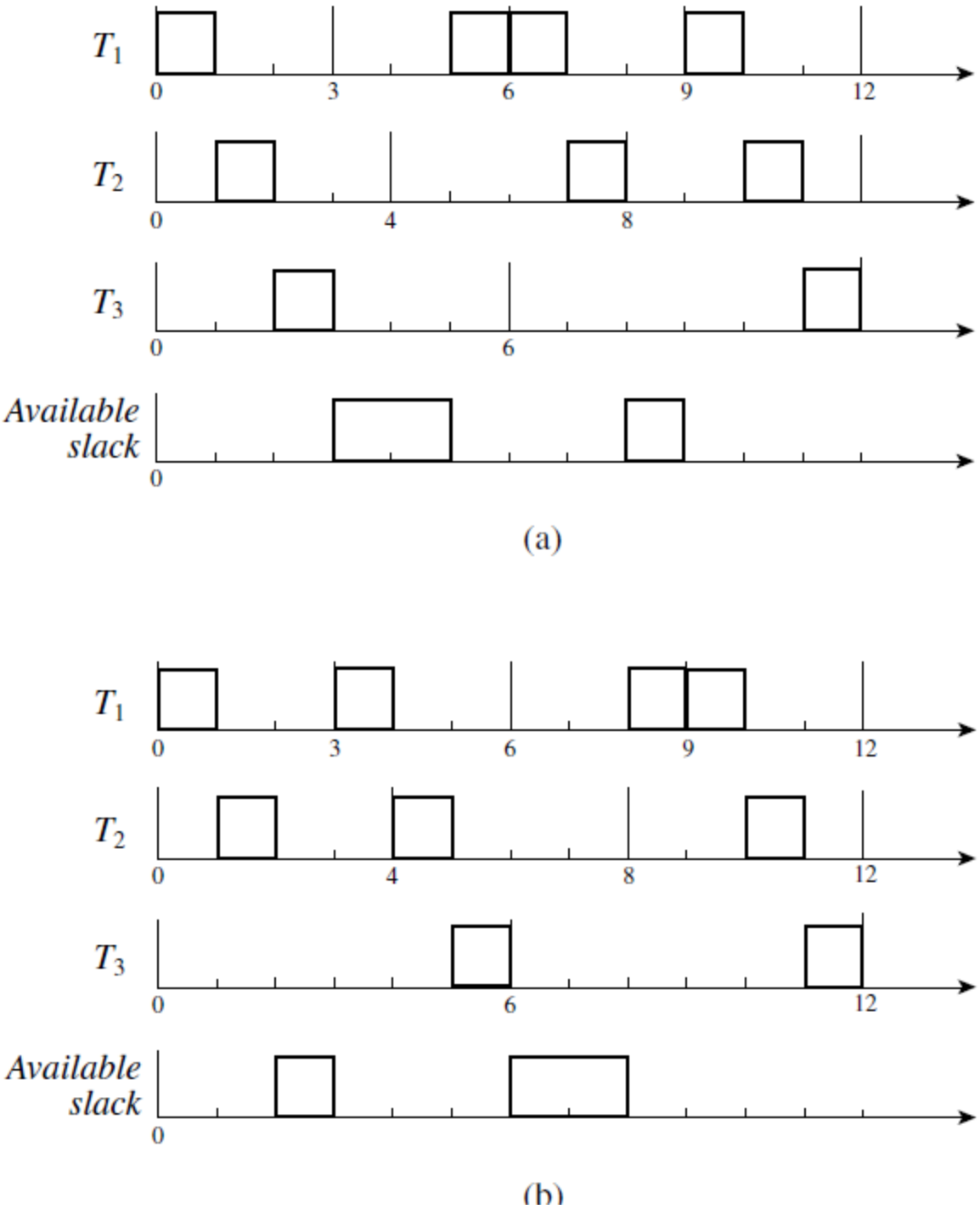


Fig: Slack Stealing in fixed priority system

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, p, I)*: *e* is the maximum execution time of all jobs in the task; *p* is the minimum interarrival time of the jobs; *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 BucketModel. To define the *leaky bucket model*, we first introduce the notion of a *(^, E)* leaky bucket filter. Such a filter is specified by its (input) rate ^ 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 ^ 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 *(^, 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 **T***i* for $i = 1, 2, \ldots, n$ 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 **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 **T***i* is the one that would have the earliest occurrence time after t among all events of **T***i* if the application were to execute alone on a slow processor with speed *si* 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 $R1, R2, \ldots, R\rho$.
  - There are υi (υ = "upsilon") indistinguishable units of a resource of type Ri.
  - 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 υi units of a resource Ri, it executes a *lock* $L(Ri, \eta i)$ (η = "eta") to request them.
- When the job no longer needs the resources, it releases them be executing an *unlock* $U(Ri,\eta i)$.
- 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 $L(Ri)$ and $U(Ri)$ 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, η; 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.

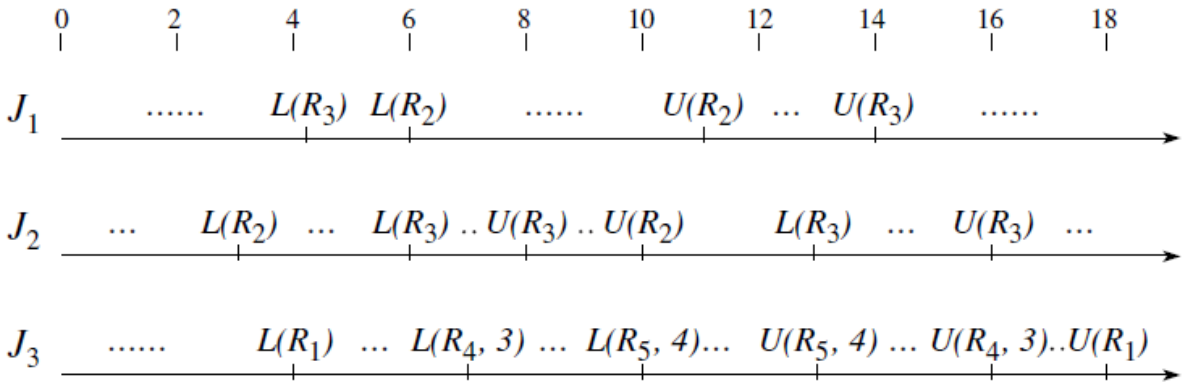


Fig. Example of Critical Section

As an example, above fig shows three jobs, $J1$, $J2$, and $J3$, and the time instants when locks and unlocks are executed if each job executes alone starting from time 0. Resources $R1$, $R2$, and $R3$ have only 1 unit each, while resources $R4$ and $R5$ have many units. Job $J3$ 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(R1)* and *U(R1)* in $J3$ is an example. Other examples are the critical sections delimited by *L (R2)* and *U(R2)* in $J2$, the second pair of *L(R3)* and *U(R3)* in $J2$ and *L(R3)* and *U(R3)* in $J1$. the critical section in $J3$ that begins at *L(R5, 4)* is [$R5$, 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.

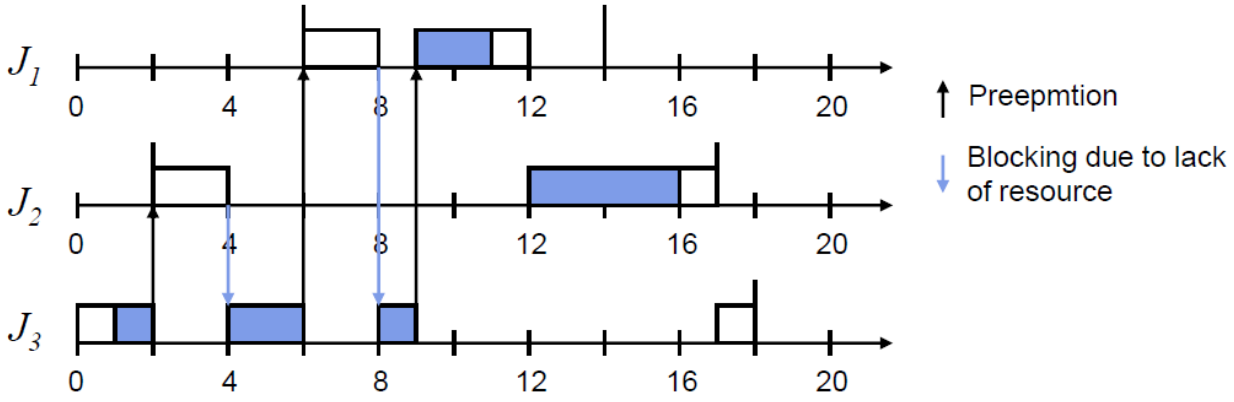
Resource Conflicts

- 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 *J1*, *J2*, and *J3* with feasible intervals (6,14], (2, 17], (0,18]

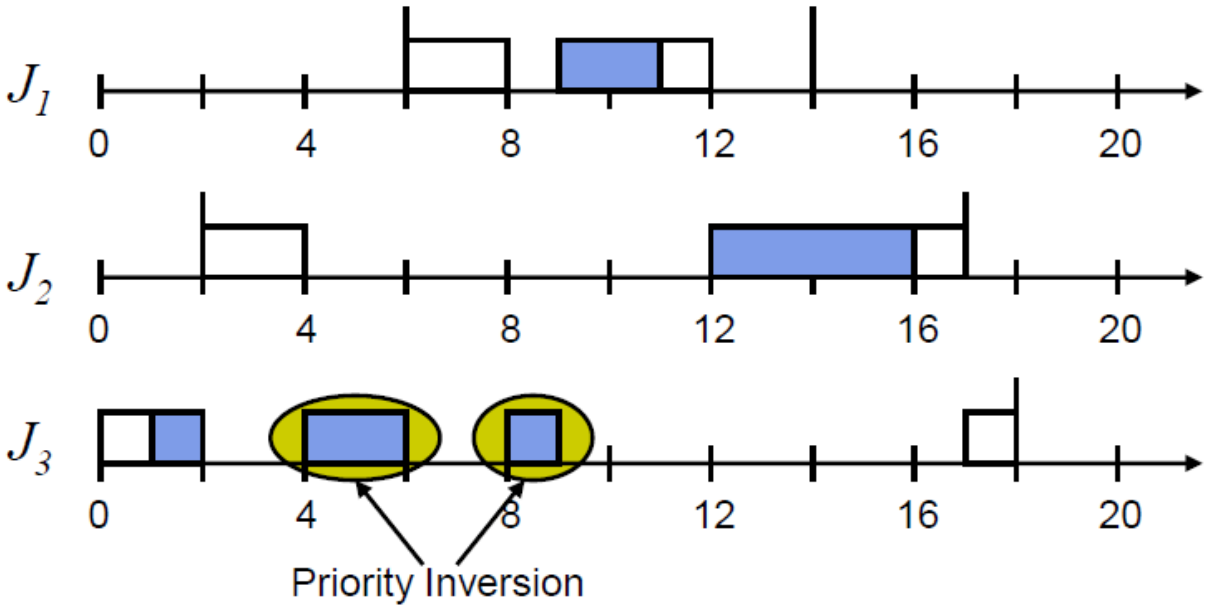
 Critical sections: [*R*; 2], [*R*; 4], [*R*; 4] for jobs 1, 2, 3

 EDF-schedule.



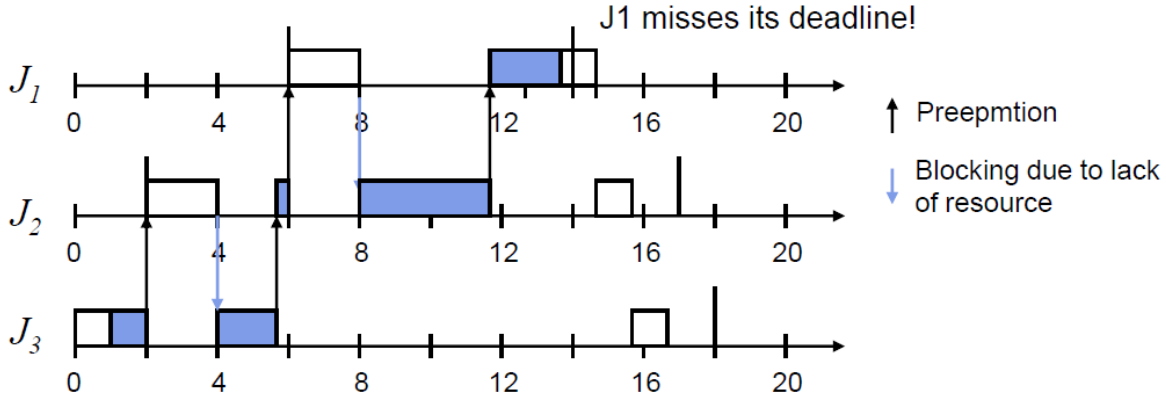
Priority Inversion:

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



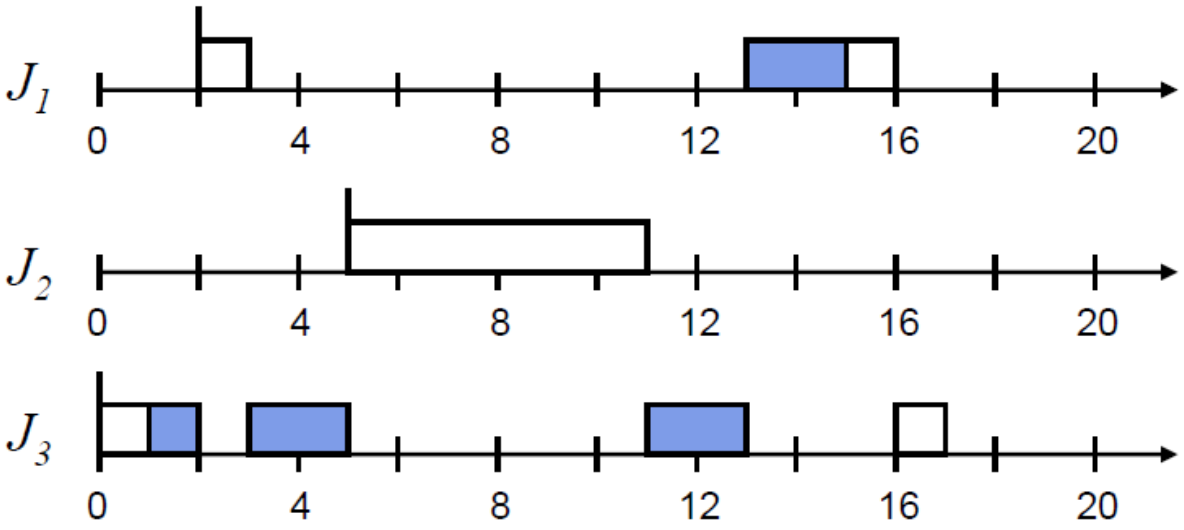
Timing Anomalies

- 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 *J1* is blocked by *J3* But *J3* is preempted by *J2* Thus *J1* has to wait for the lower-priority job *J2* that does not even require the resource that *J1* 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: $L(R_2)$, $L(R_1)$, $U(R_1)$, $U(R_2)$

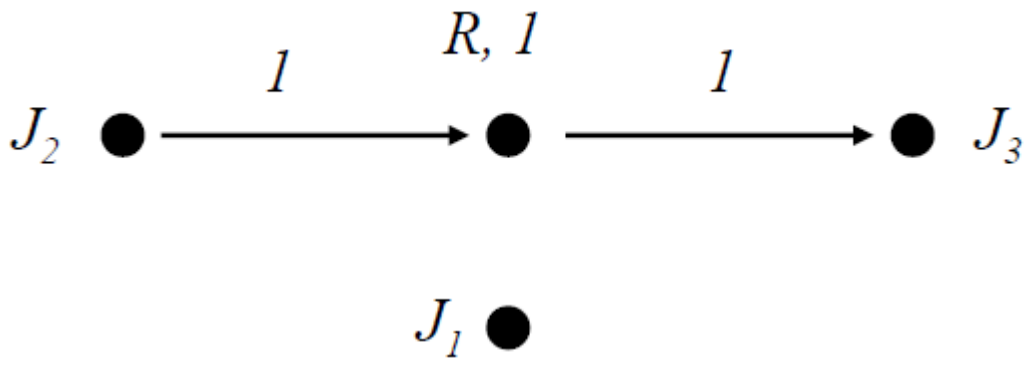
Job 1 and job 2 block each other

Wait-for-Graphs

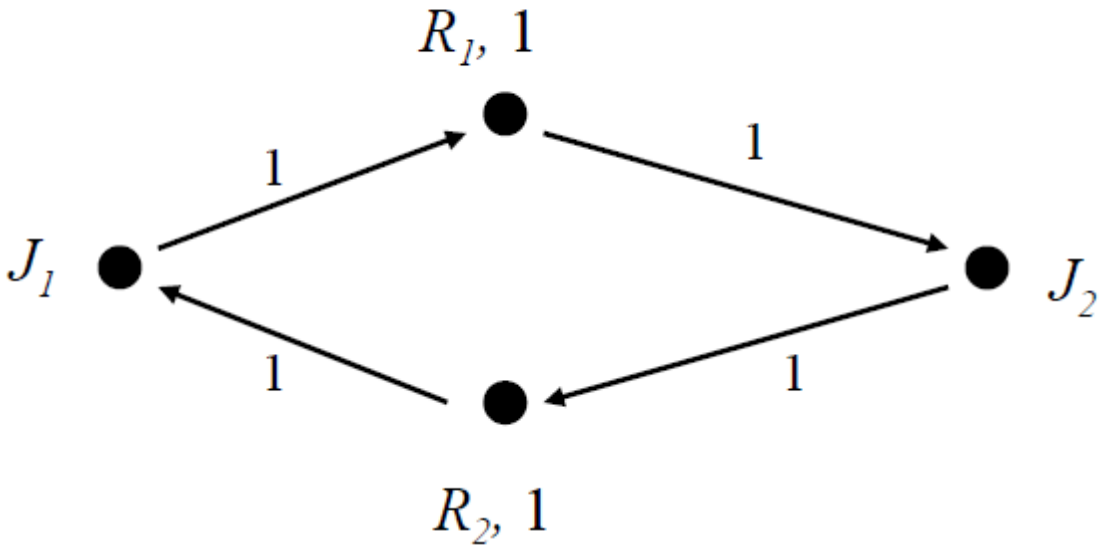
- Wait-for-Graphs are used to describe the dynamic-blocking relationship among jobs
- Vertexes represent jobs and resources.
- Edges can express that
  - a job owns a resource.
  - a job waits for a resource.
- A cyclic path in a wait-for-graph indicates a deadlock.

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.

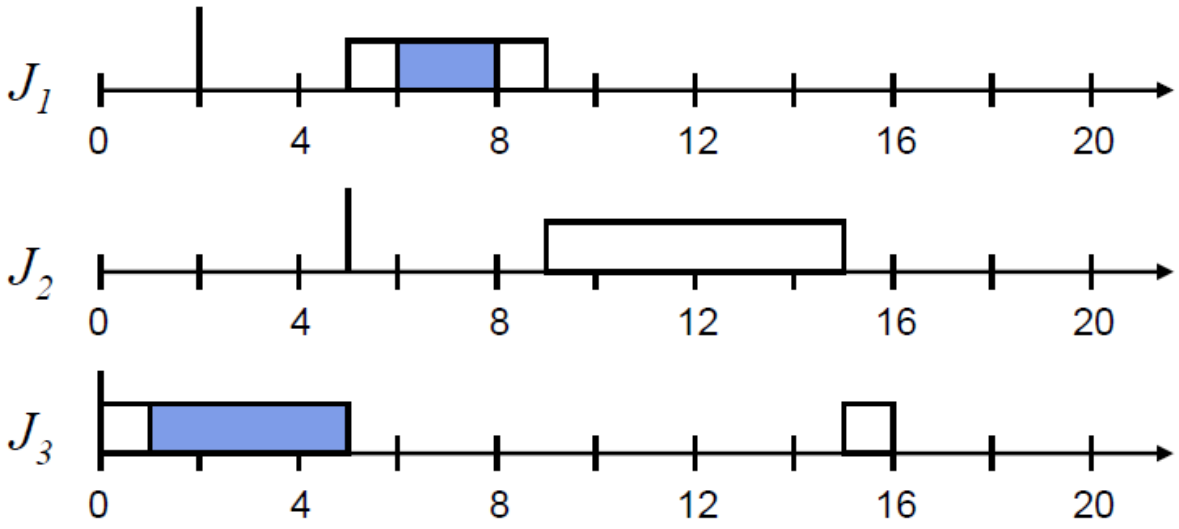


The following system is deadlocked:



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.

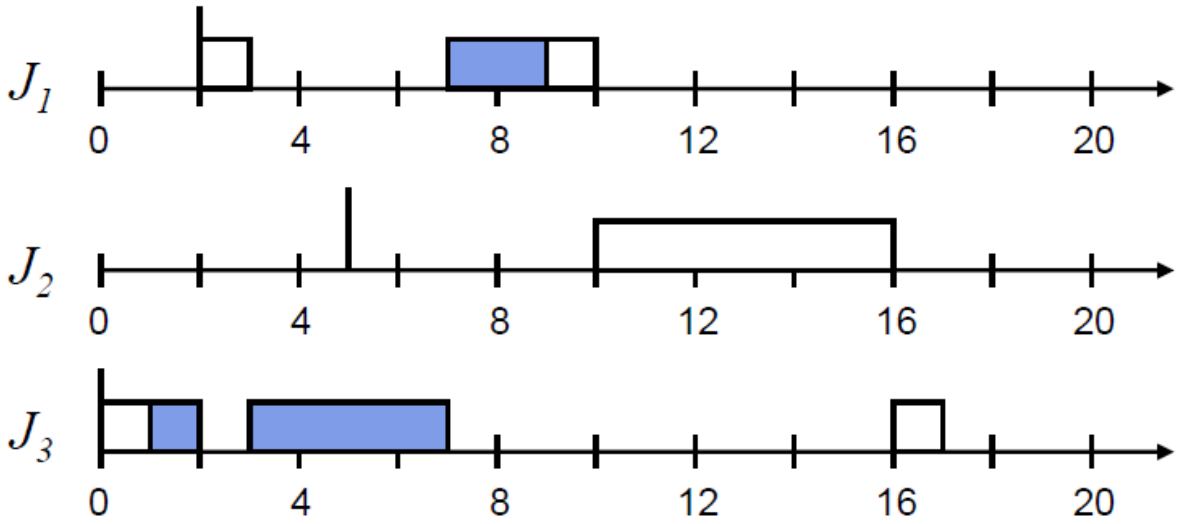
Priority-Inheritance Rule:

 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:

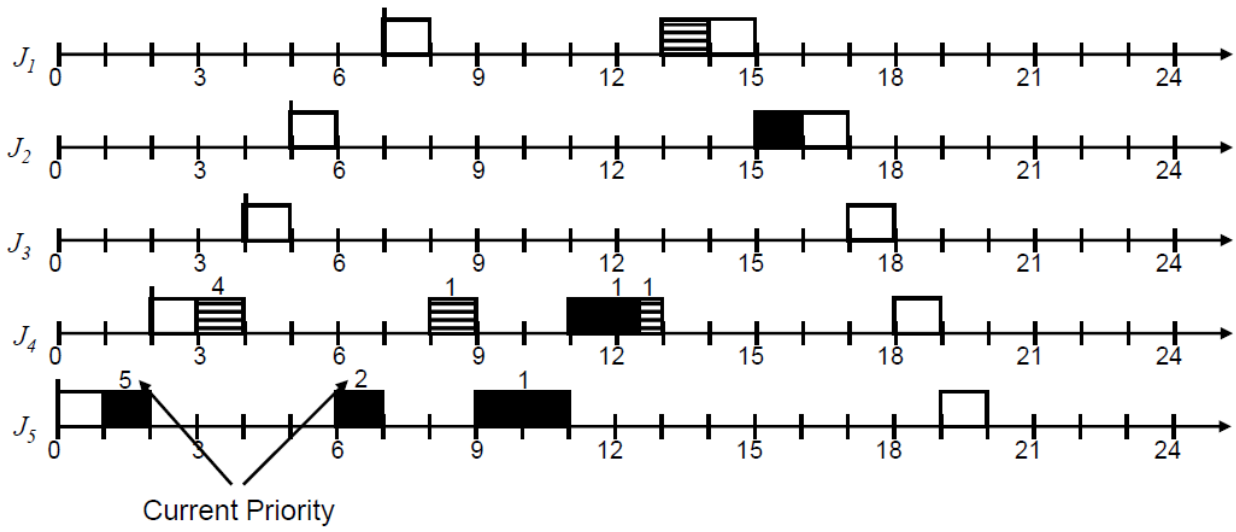


Description:

- When *J1* requests resource *R* and becomes blocked at time 3, job *J3* inherits the priority $\pi 1$ of job *J1*.
- When *J2* becomes ready at time 5, it cannot preempt *J3* because its priority $\pi 2$ is lower than the inherited priority $\pi 1$ of *J3*.
- As a consequence *J3* completes its critical section as soon as possible.

Next Example:

Job	r_i	e_i	π_i	Critical Sections
J_1	7	3	1	[*Shaded*; 1]
J_2	5	3	2	[*Black*;1]
J_3	4	2	3	
J_4	2	6	4	[*Shaded*;2[*Black*;1.5]Shaded;0.5]
J_5	0	6	5	[*Black*;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 *J5*'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 *Ri* is the highest priority of all jobs that require *R* and is denoted Π(*Ri*).

 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 *π(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 *π(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 *π(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 *π(t)* is not higher than the current ceiling of the system Π_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.

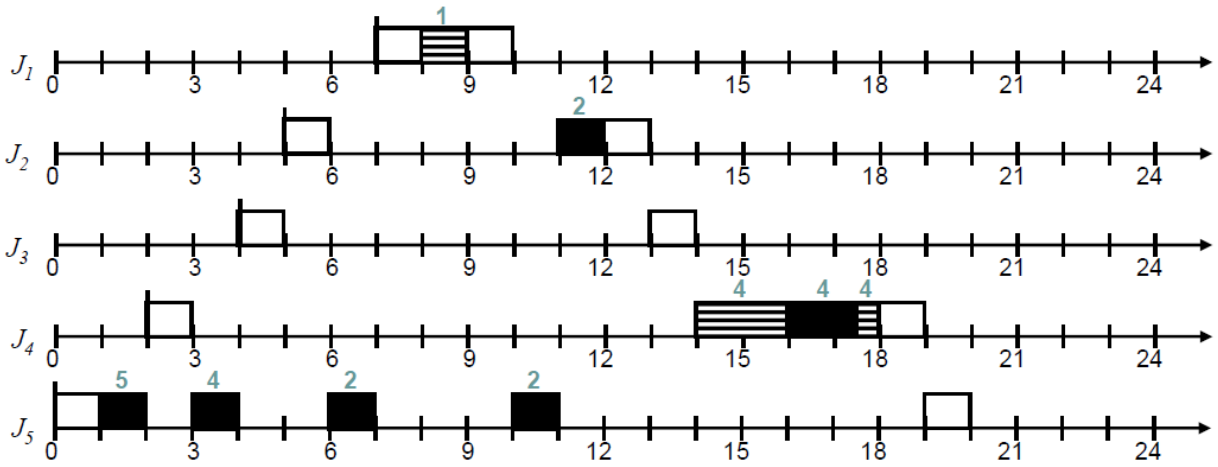
Priority-Inheritance Rule:

 When *J* becomes blocked, the job J_l which blocks *J* inherits the current priority *π(t)* of *J*.

 J_l executes at its inherited priority *π(t)* until the time where it releases every resource whose priority ceiling is equal to or higher than *π(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).

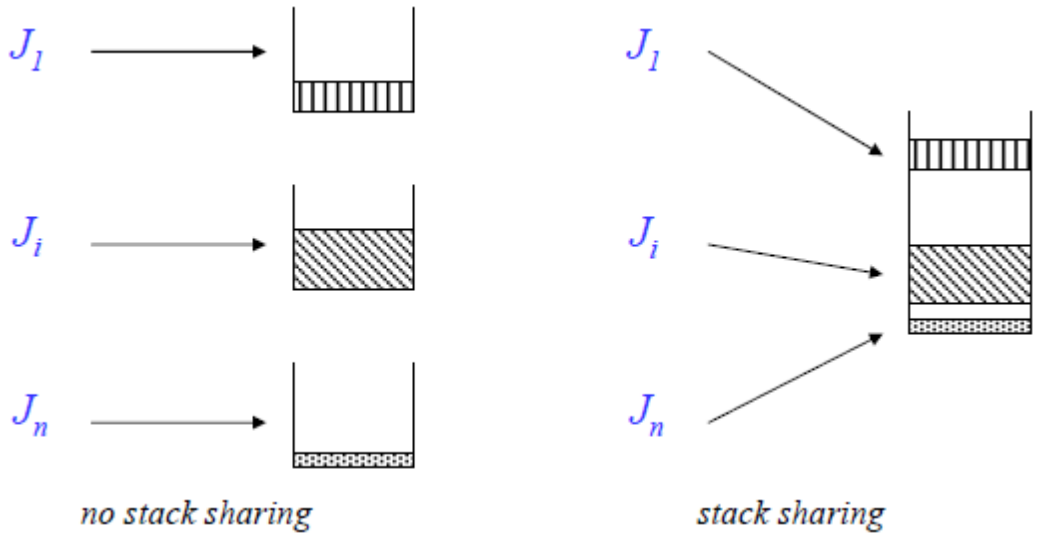
Eample:



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;

•Iin 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 *JL* 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;

⇨ 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 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.

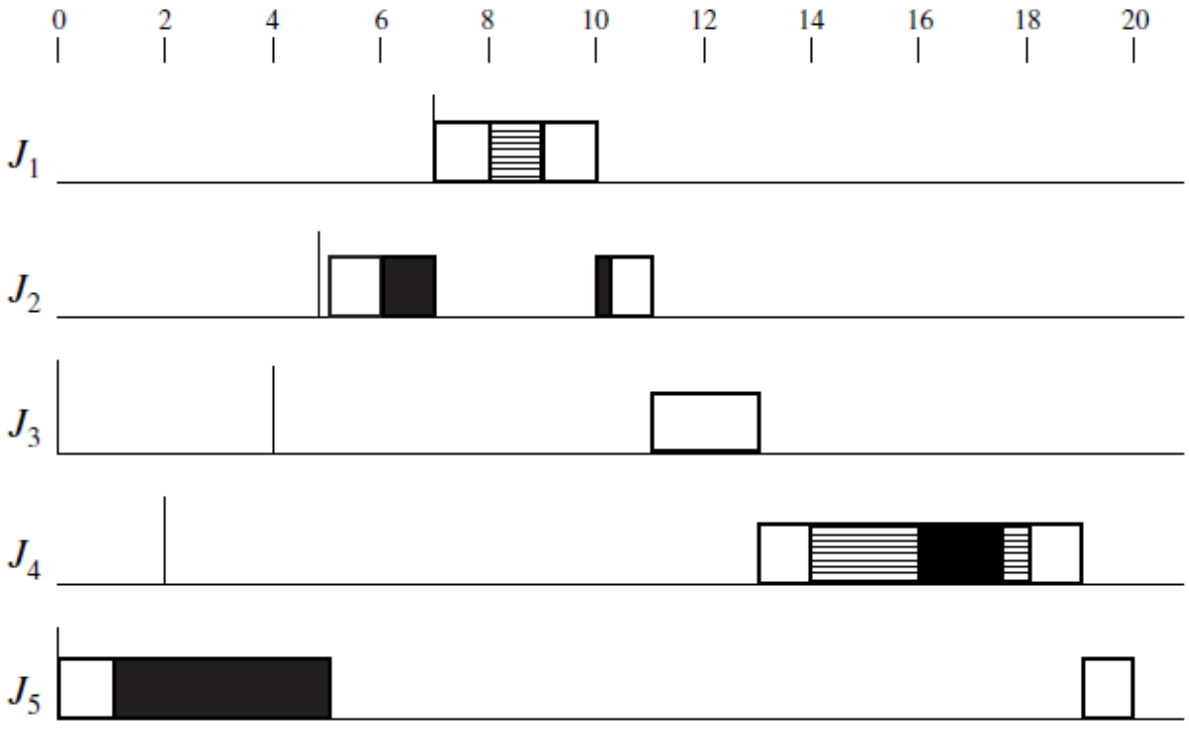


Fig:SBPC

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 *T*1 = *(*2, 0.9*)* and *T*2 = *(*5, 2.3*)* in Figure 6–4. In its first two periods (i.e., from time 0 to 4), *T*1 has priority 1 while *T*2 has priority 2, but from time 4 to 5, *T*2 has priority 1 and *T*1 has priority 2. Suppose that the task *T*1 requires a resource *X* while *T*2 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.

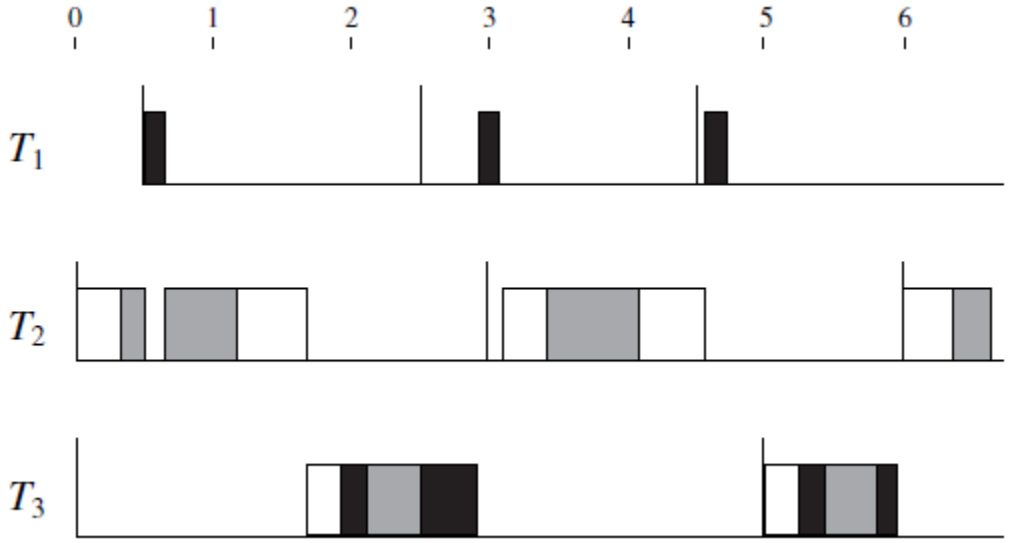
Implementation of Priority-Ceiling Protocol in Dynamic-Priority Systems

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 $T1 = (5.0, 1.2; [Black; 1.0 [Shaded; 0.4])$. 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 ofMultiple-Unit Resources

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

CONTROLLING CONCURRENT ACCESSES TO DATA OBJECTS

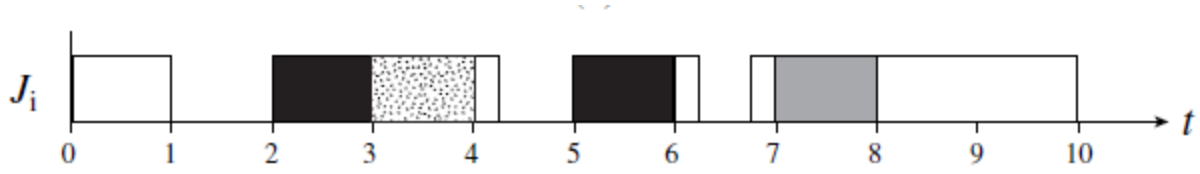
Data objects are a special type of shared resources. When jobs are scheduled preemptively, their accesses to (i.e., reads and writes) data objects may be interleaved. To ensure data integrity, it is common to require that the reads and writes be serializable. A sequence of reads and writes by a set of 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 *Jh* to read and write a data object *X* between two disjoint critical sections of a lower-priority job *Jl* during which *Jl* 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 *Jl* either proceed or follow that of *Jh*).

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 require 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 *Ji* after it accesses *Rk* for the last time the *last access notification* for *Rk* by *Ji* and the time of this notification the *last access time of Rk by Ji. For* example the job *Ji* 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(Ji , 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(Ji , 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.


J_i
0
1
2
3
4
5
6
7
8
9
10
t