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.

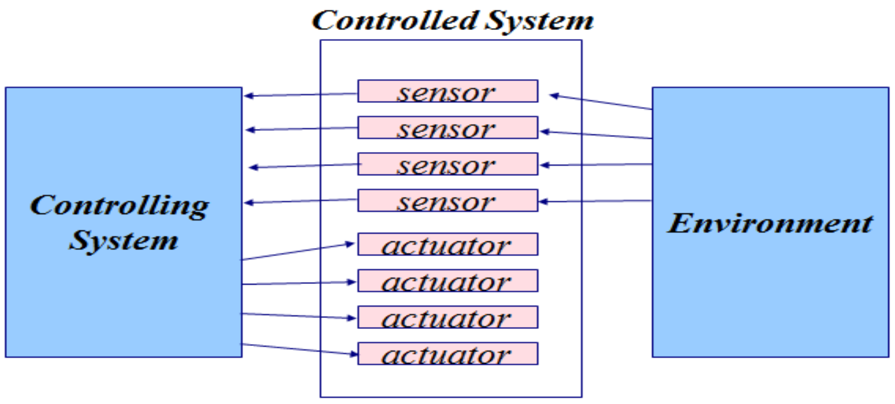


Fig Typical Real time System

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)*.

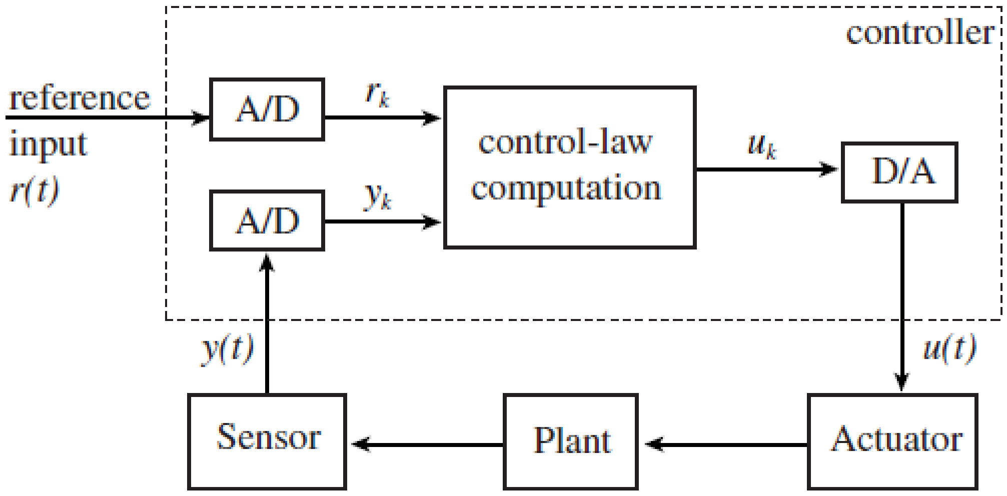


FIGURE 1–1 A digital controller.

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:

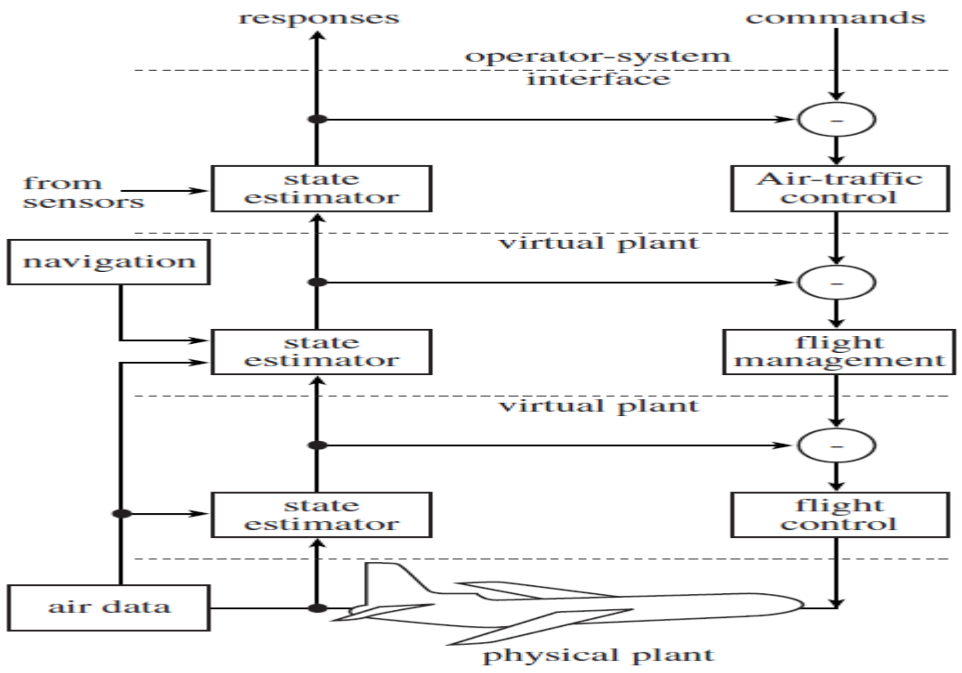


FIGURE 1—4 Air traffic/flight control hierarchy.

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.

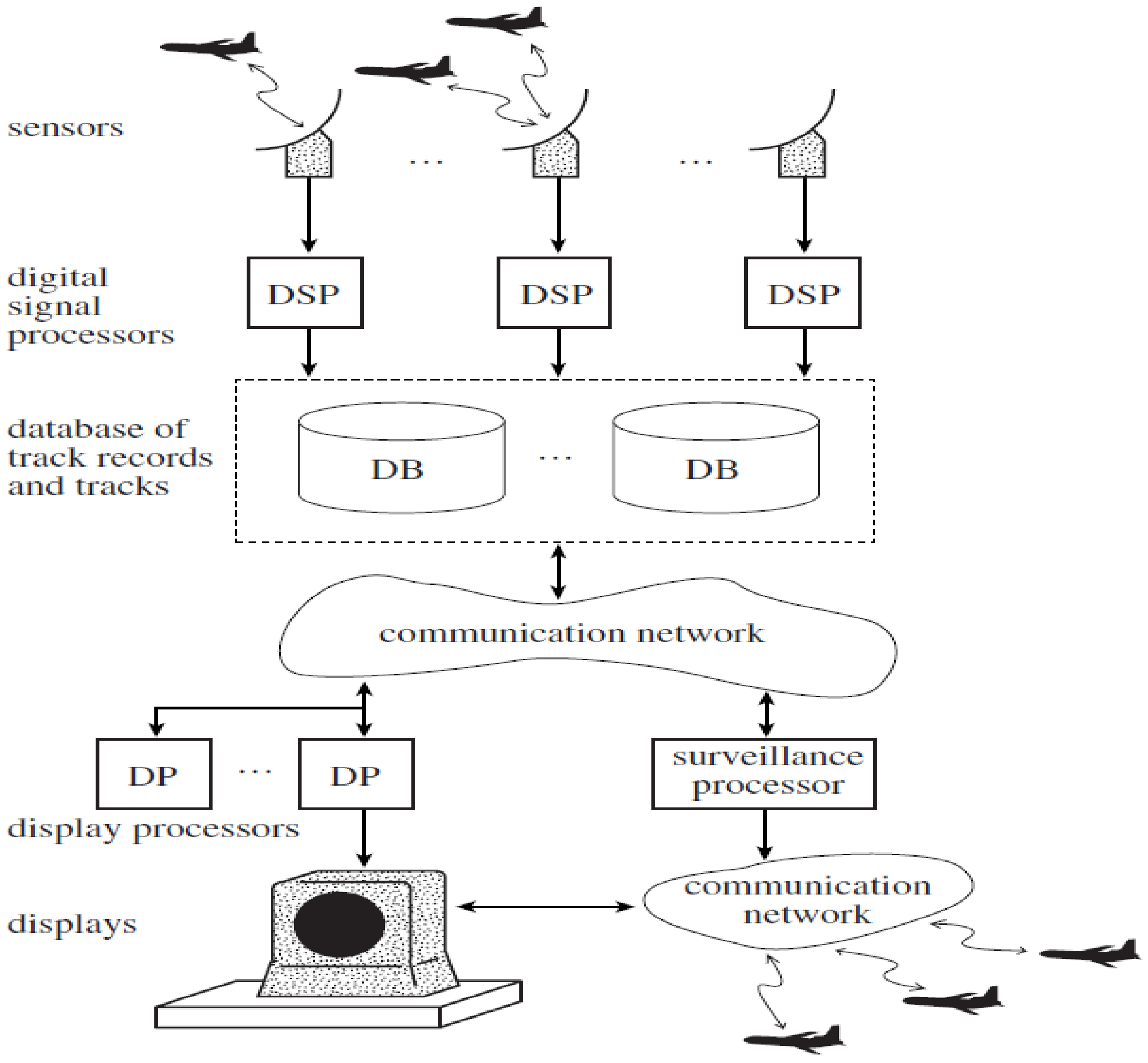


Fig. Architecture of Air Traffic Control System

Radar Signal processing

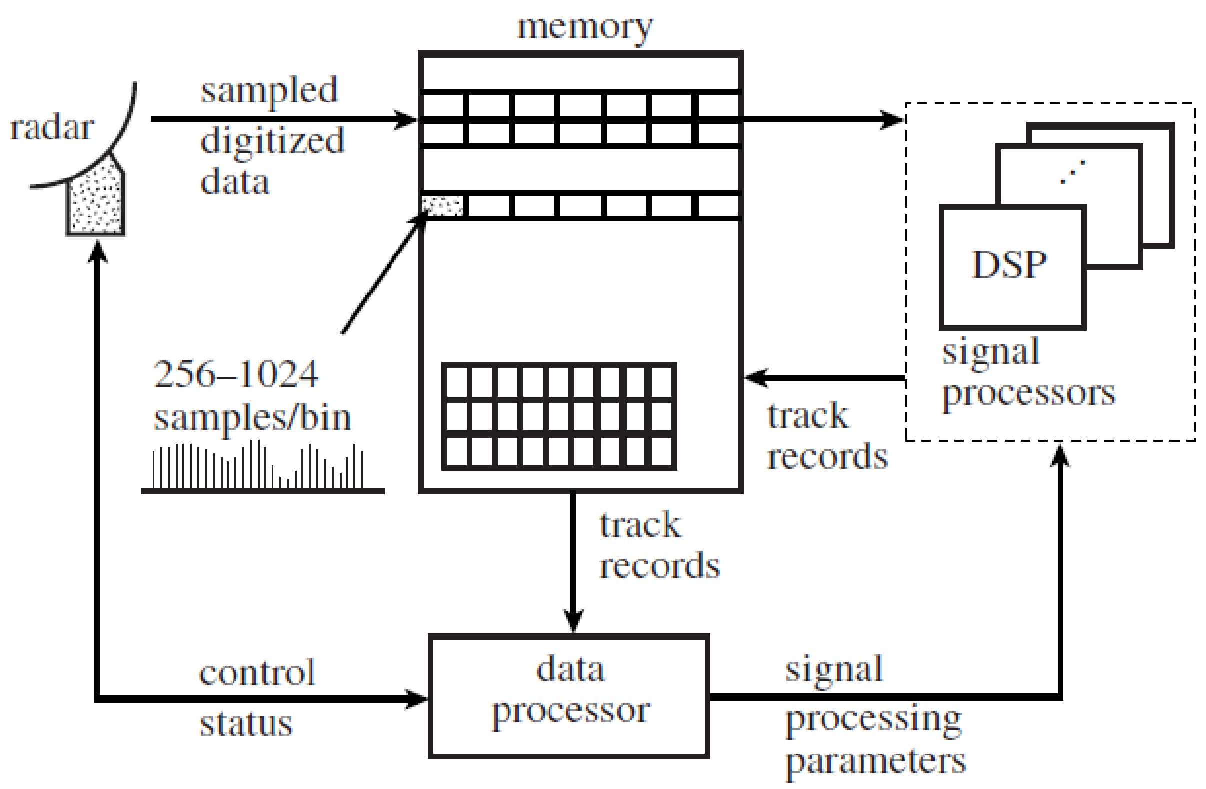


Fig . Radar Signal processing and Tracking System

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 2*x/c* seconds after the transmitted pulse, where *c* = 3×108 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.

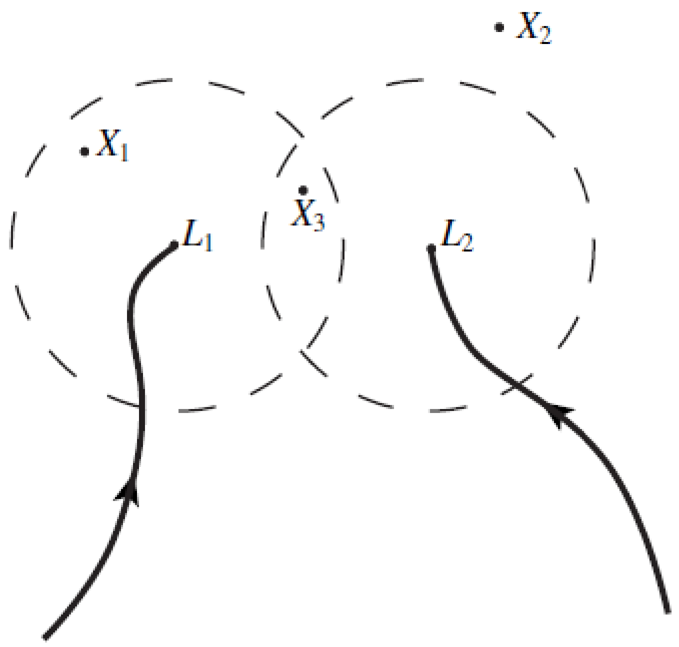


Fig . Gating Process

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.

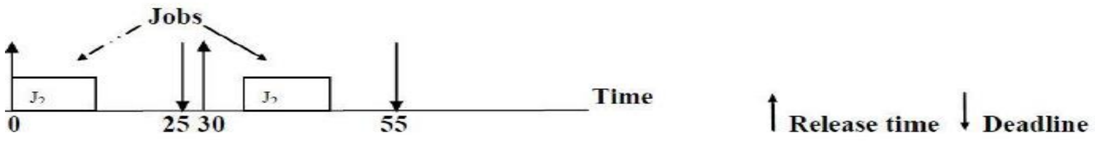
Chapter2

JOBS AND PROCESSORS:

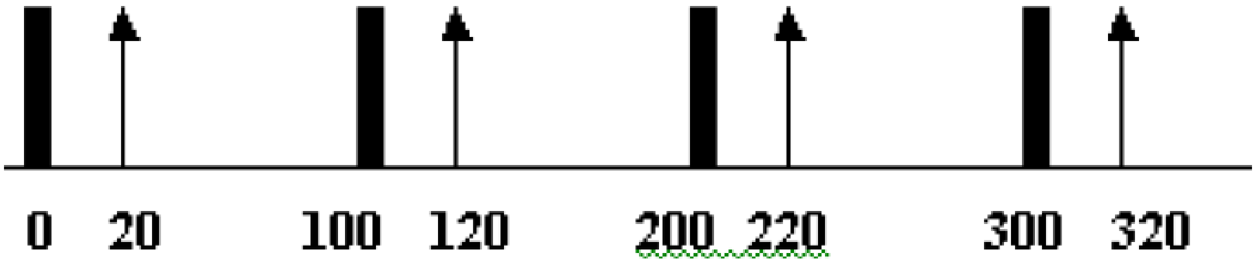
Each unit of work that is scheduled and executed by the system is called a *job* and a set of related jobs which jointly provide some system function a *task*. Hence, the computation of a control law is a job. So is the computation of a FFT (Fast Fourier Transform) of sensor data, or the transmission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

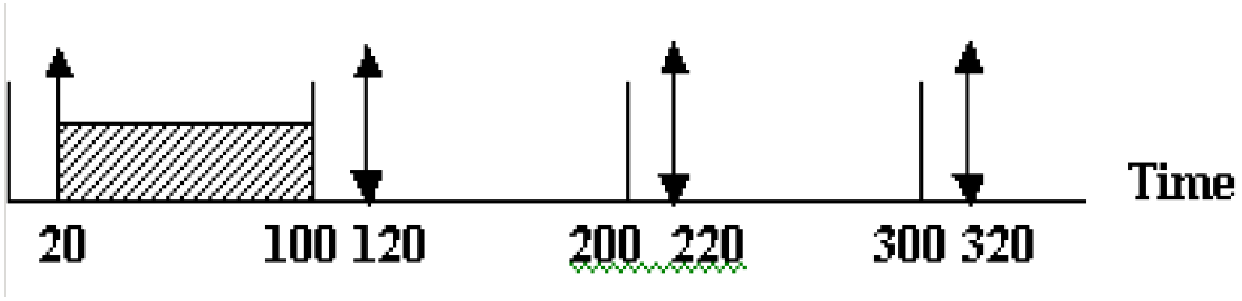
The release time of a job is the instant of time at which the job becomes available for execution. The job can be scheduled and executed at any time at or after its release time whenever its data and control dependency conditions are met.



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 $J0$, $J1$, . . . , Jk, ... The release time of the job Jk in this job stream is 20 + k × 100 msec, for k = 0, 1, ... *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…………………….

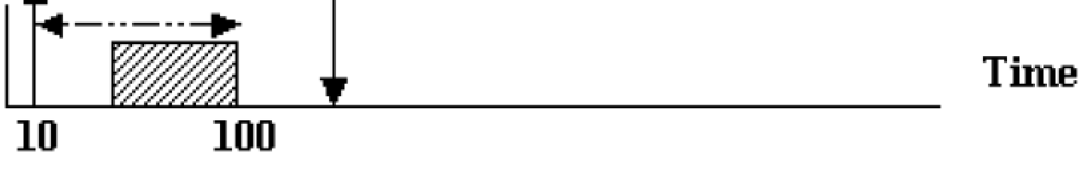


The ***deadline*** of a job is the instant of time by which its execution is required to be completed. Suppose that in the previous example, each control-law computation job must complete by the release time of the subsequent job. Then, their deadlines are 120 msec, 220 msec, and so on, respectively. 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:

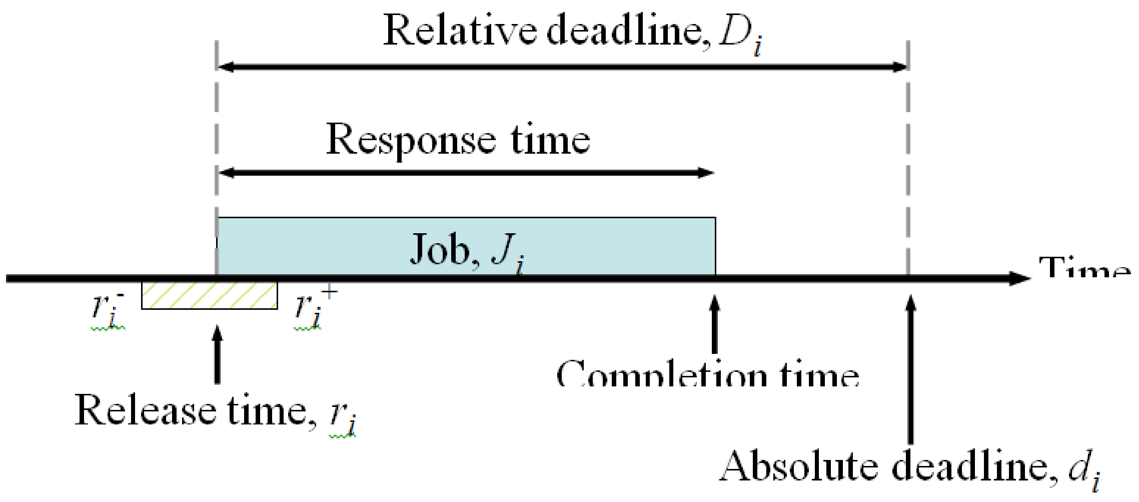
RESPONSE TIME: the length of time from the release time of the job to the instant when it completes.
Response time = Job completion time – Job release time
Job's timing requirements can be expressed in terms of "response time.



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.

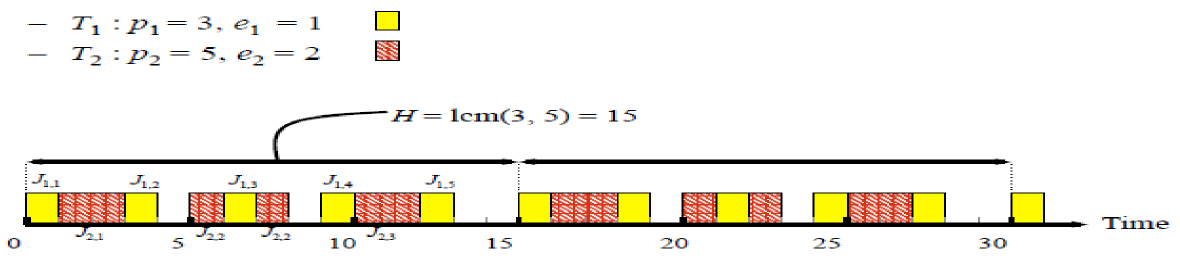
U - Total utilization = Sum over all u_i.

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, $H = \mathrm{lcm}(p_i)$ 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.

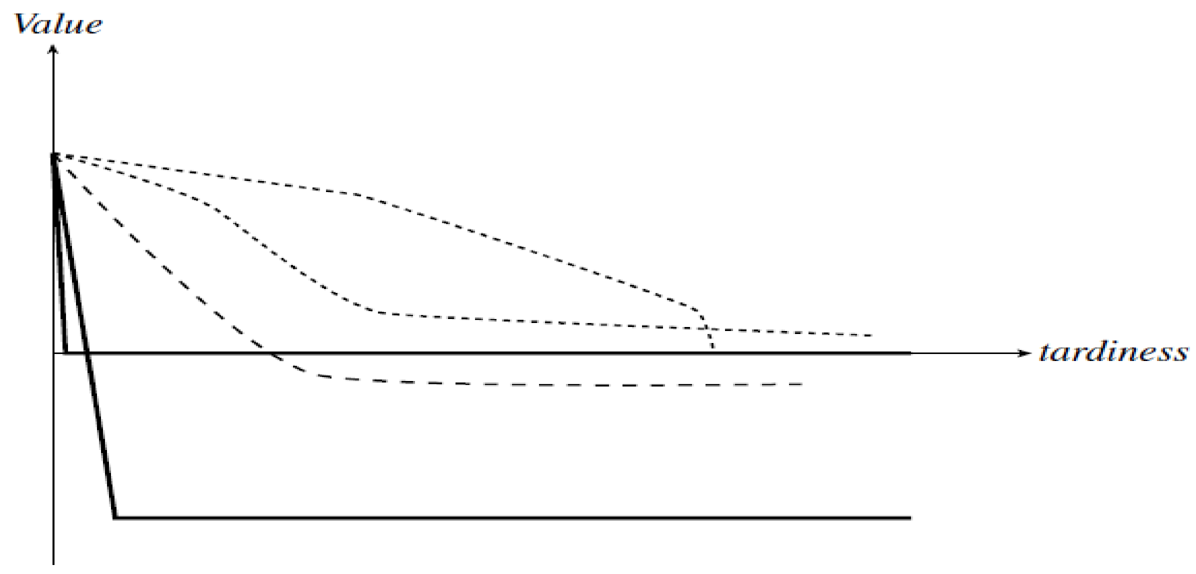


Fig:- Examples of usefulness function

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 $j_i < j_k$

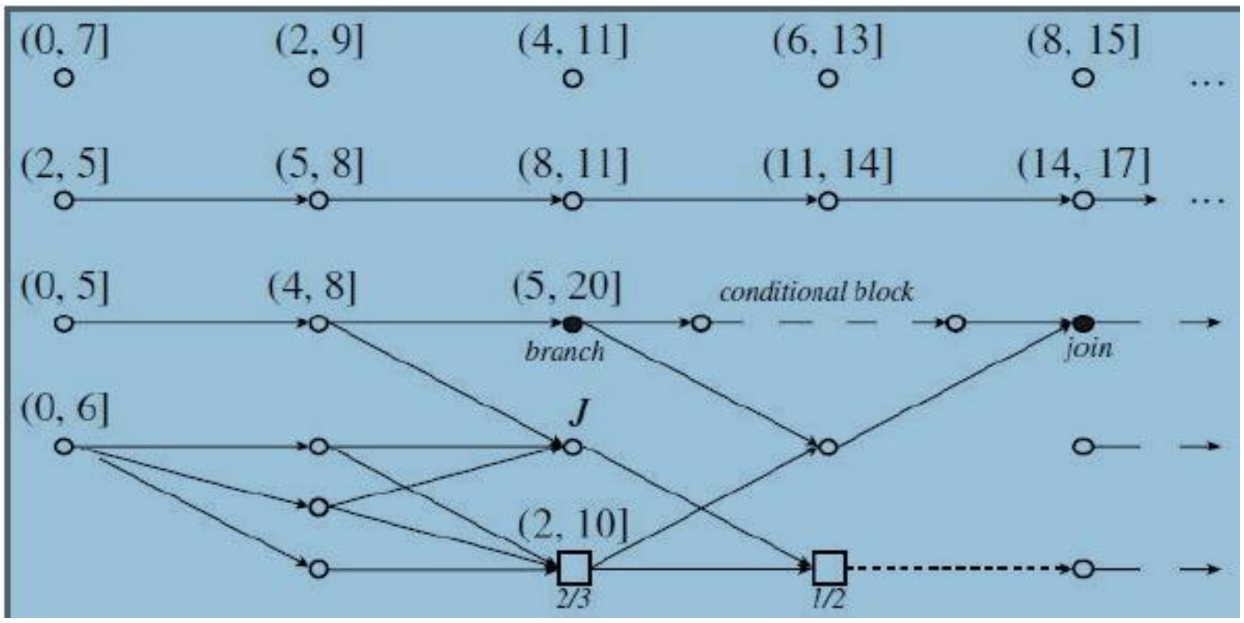
- Ji is an immediate predecessor of j_k (and j_k is is an immediate successor of j_i) if $j_i < j_k$ and there is no other job j_j such that $j_i < j_j < j_k$
- Two jobs j_i and j_k are independent when neither $j_i < j_k$ nor $j_k < j_i$
- A job with predecessors is *ready* for execution when the time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

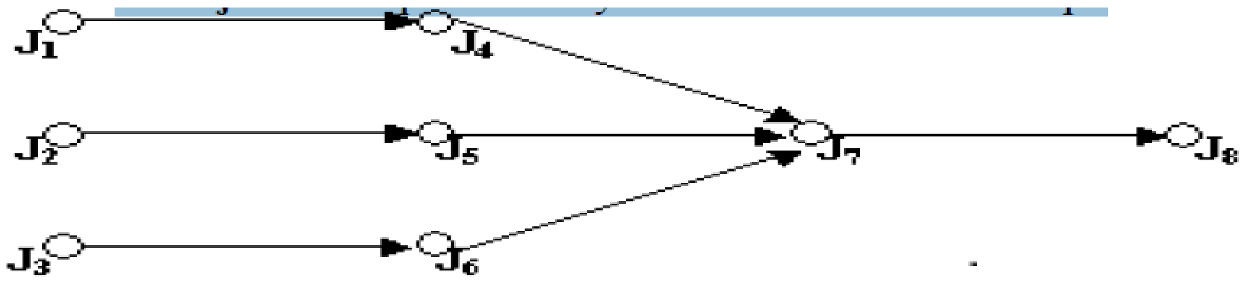
1. Temporal dependency
2. AND/OR precedence constraints
3. Conditional branches
4. Pipeline relationship.

1. Temporal dependency

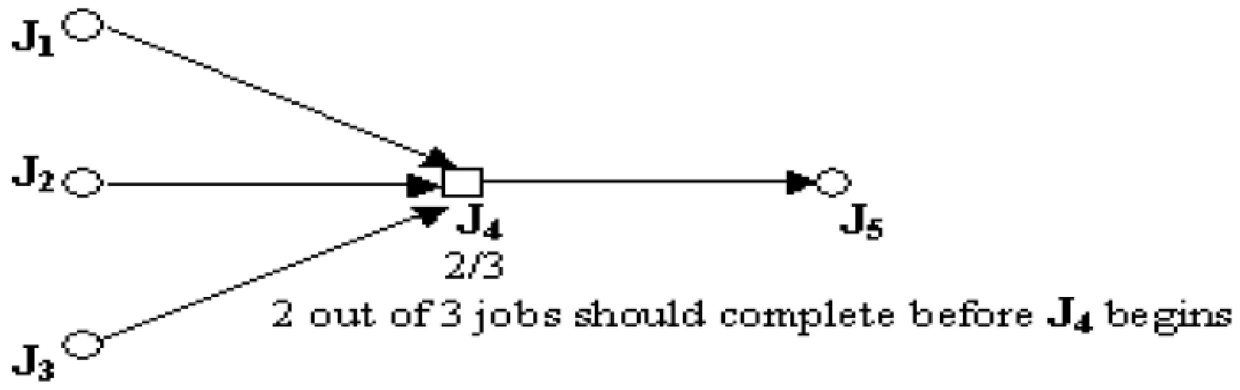
Some jobs may be constrained to complete within a certain amount of time relative to one another. We call the difference in the completion times of two jobs the *temporal distance* between them. Jobs are said to have a *temporal distance constraint* if their temporal distance must be no more than some 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.

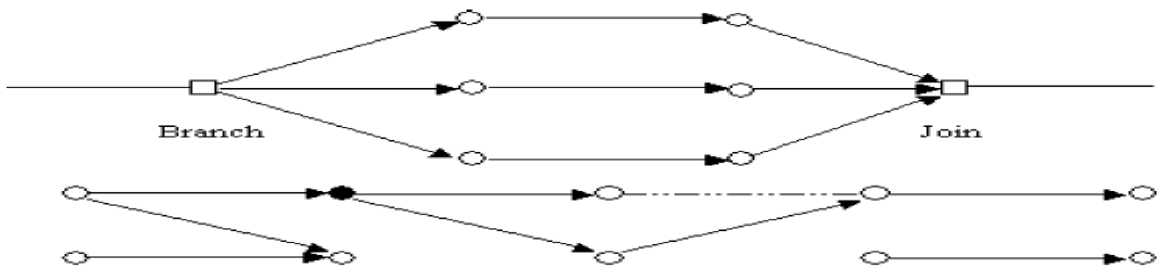


A **square node** is represented by a label **(*x/y*)** to indicate that the job can begin execution if ***„x' out of „y'*** predecessors complete execution. If job having more than one predecessor jobs can begin execution at or after its release time provided one or more of its predecessor have completed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

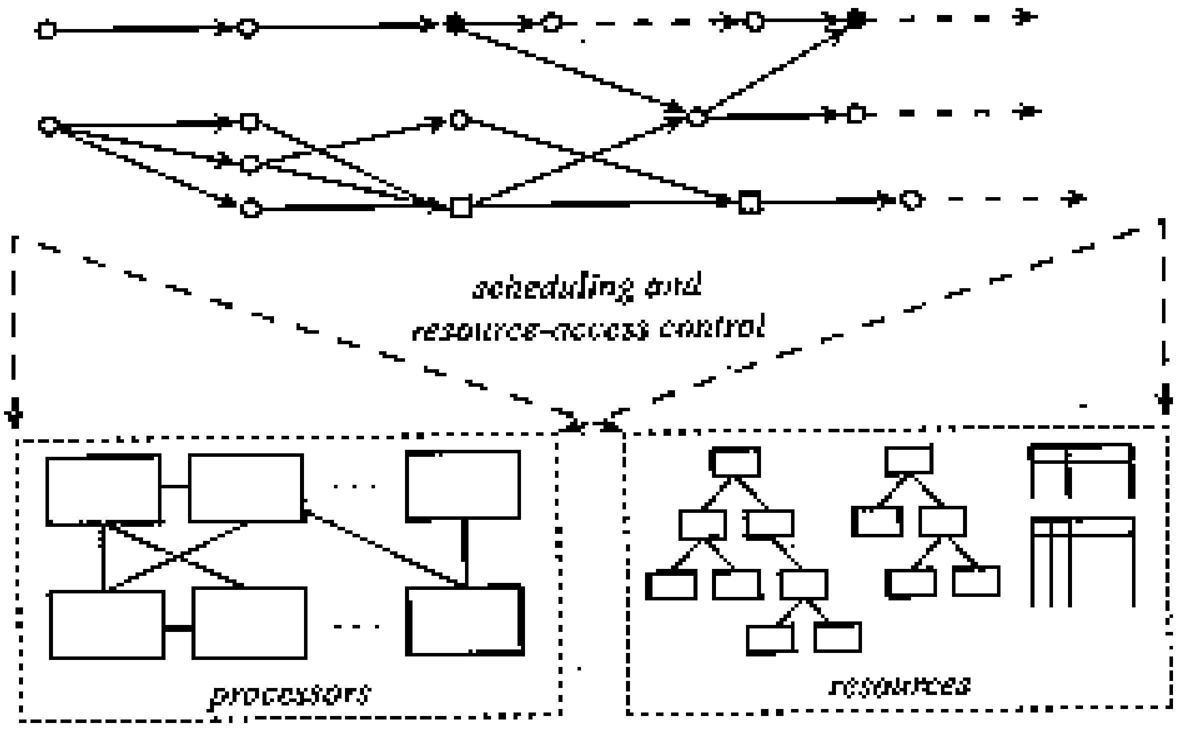
Sub graph beginning with branch job and ending with join job is called **conditional block**. l^k precedence graphs are needed to represent a system with *k* conditional block each having *l* branches .



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.

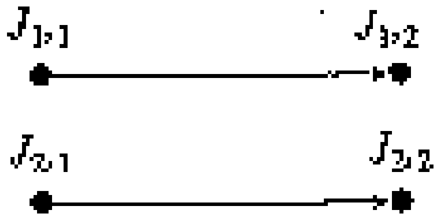
Weighted Round-Robin Approach:

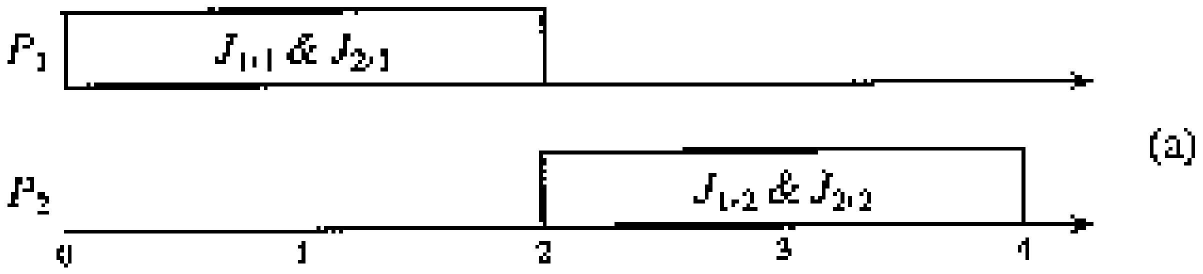
- The *weighted round-robin algorithm* has been used for scheduling real-time traffic in high-speed switched networks.
- Rather than giving all the ready jobs equal shares of the processor, different jobs may be given different *weights*.
- A job with weight *wt* gets *wt* time slices every round.
- 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.

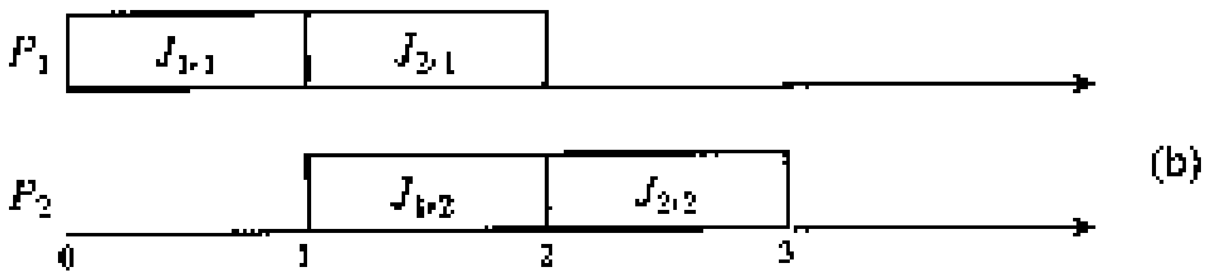
- However if a successor job is able to consume incrementally what is produced by a predecessor job, such as with a Unix pipe, weighted round-robin scheduling may be a reasonable approach.

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.

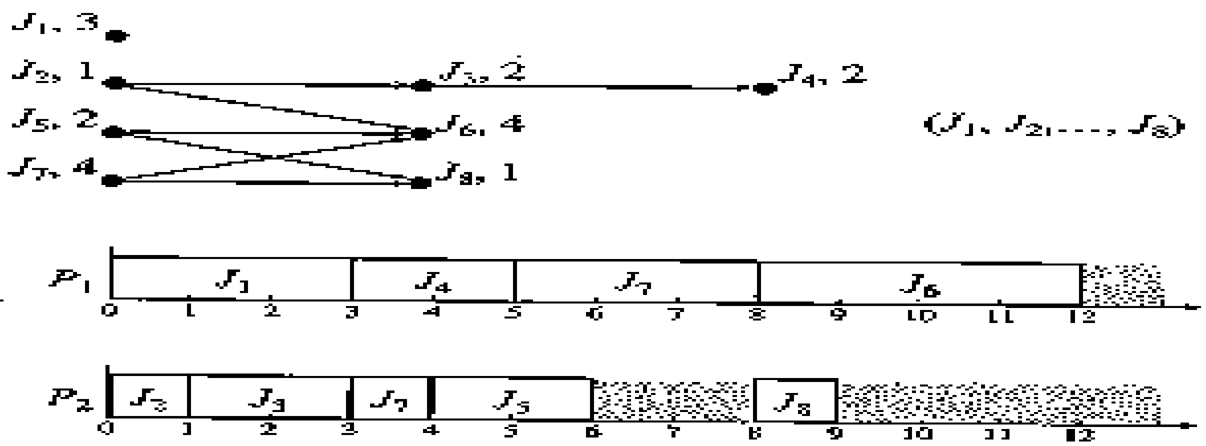


Fig. With preemption

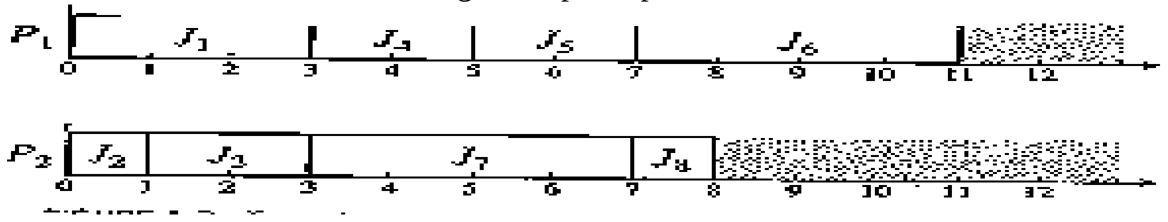


Fig without preemption

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.

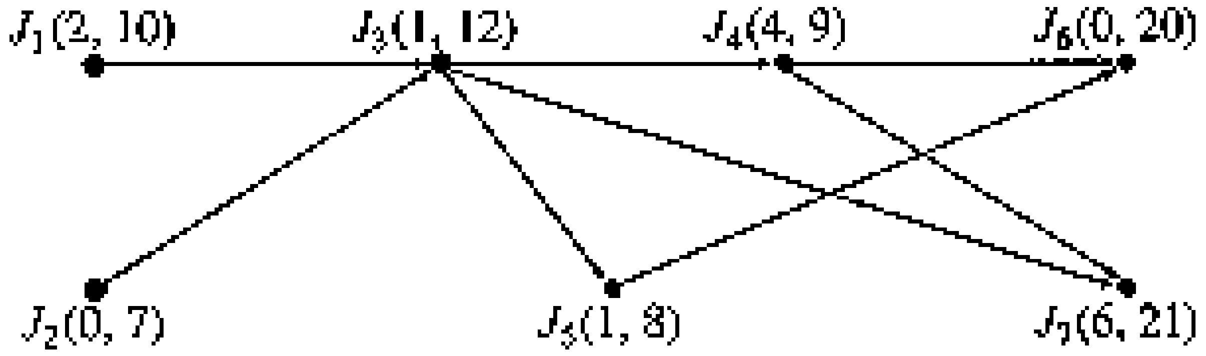
In this example the response of the static system is just as good as that of the dynamic system.

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.

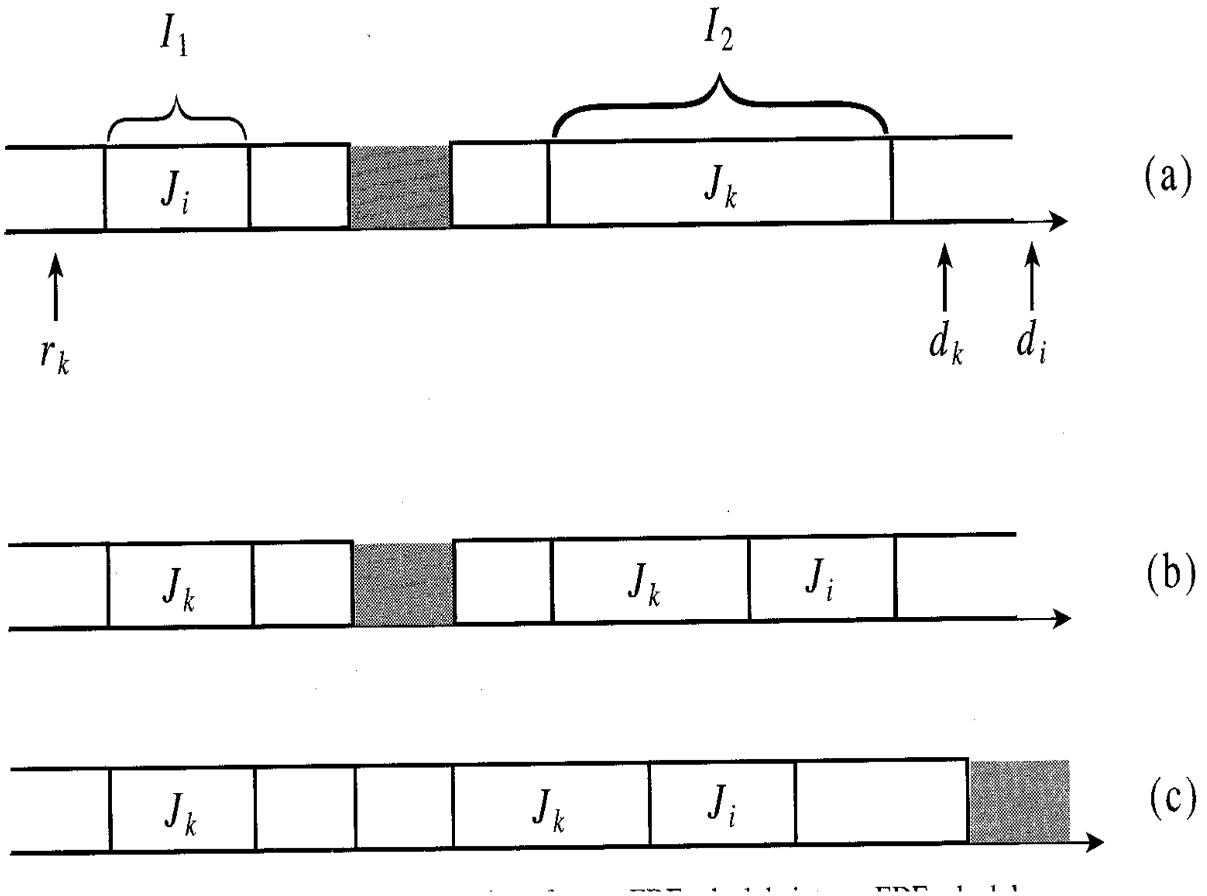
Theorem

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 $\boldsymbol{r_k}$ is no later than the beginning of $\boldsymbol{I_1}$.

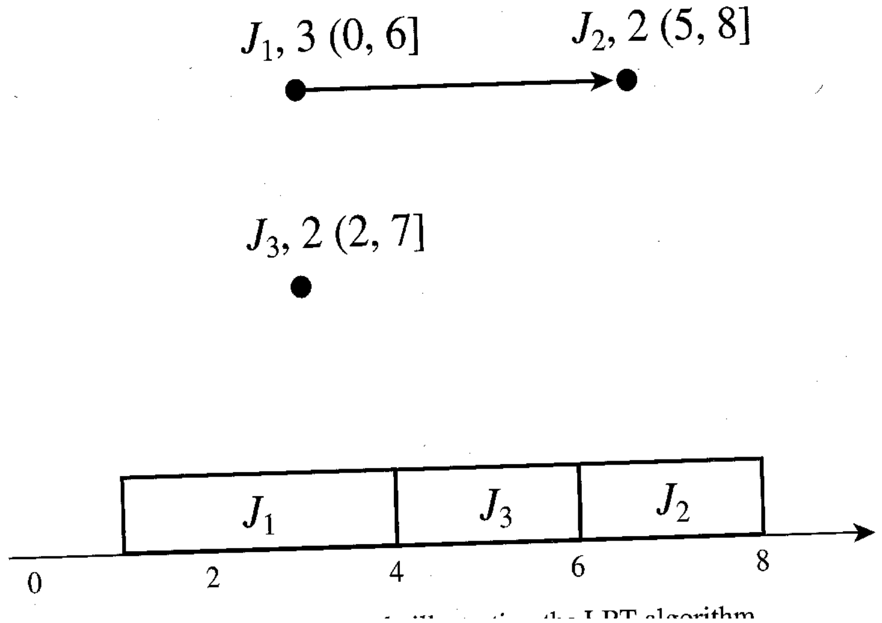
To transform the given schedule we swap $\boldsymbol{J_i}$ and $\boldsymbol{J_k}$.

- If $\boldsymbol{I_1}$ is shorter than $\boldsymbol{I_2}$, we move the portion of $\boldsymbol{J_k}$ that fits in $\boldsymbol{I_1}$ forward to $\boldsymbol{I_1}$ and move the entire portion of $\boldsymbol{J_i}$ scheduled in $\boldsymbol{I_1}$ backward to $\boldsymbol{I_2}$ and place it after $\boldsymbol{J_k}$.
- Clearly, this swap is always possible.
- We can do a similar swap if $\boldsymbol{I_1}$ is longer than $\boldsymbol{I_2}$.
- In which case we move the entire portion of $\boldsymbol{J_k}$ scheduled in $\boldsymbol{I_2}$ to $\boldsymbol{I_1}$ and place it before $\boldsymbol{J_i}$ and move the portion of $\boldsymbol{J_i}$ that fits in $\boldsymbol{I_2}$ 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

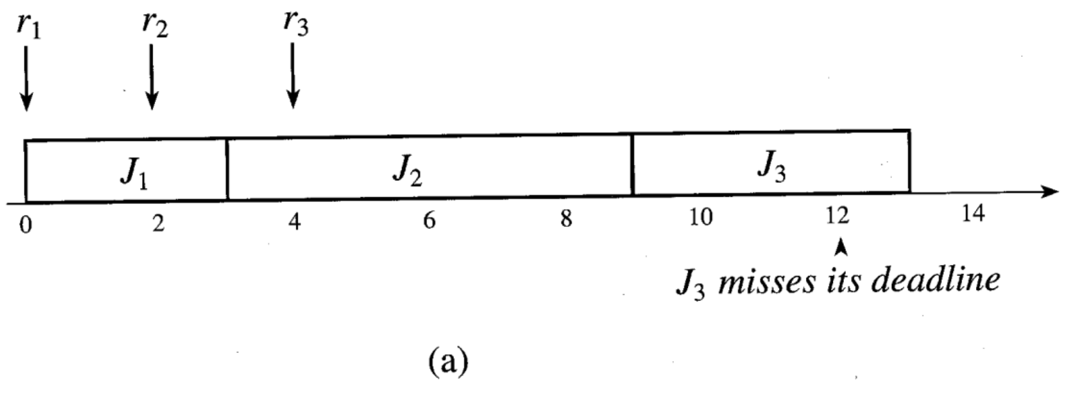
Non-optimality of EDF and LST:

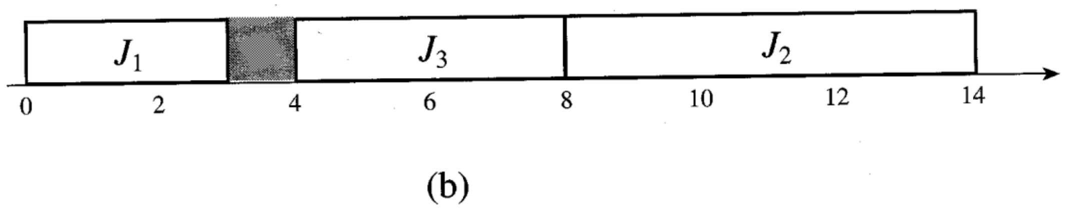
- Do EDF and LST algorithms remain optimal if preemption is not allowed or there is more than one processor?

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.

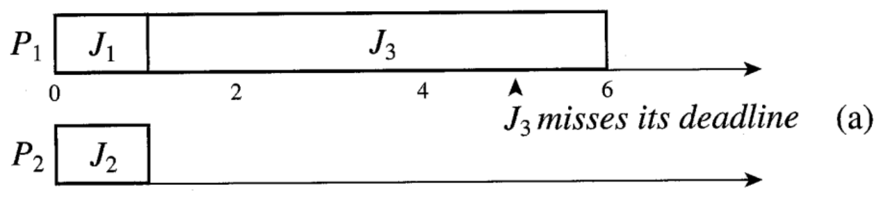
- Both EDF and LST would produce the infeasible schedule (a) whereas a feasible schedule is possible (b).
- Note that (b) cannot be produced by a priority driven algorithm

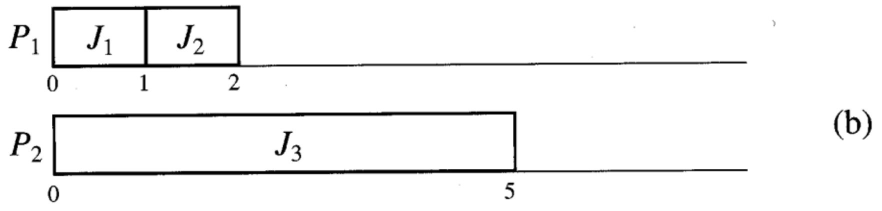




Non-optimality of EDF for multiprocessors:

- This time we have two processors and three jobs *J₁, J₂, J₃,* 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 ?*

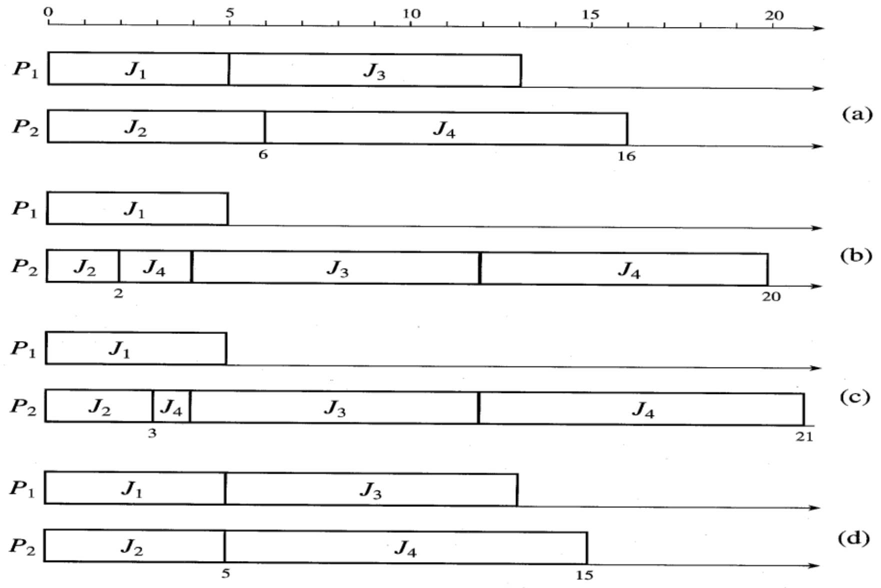
- Given a set of jobs, the set of resources available to the jobs, and the scheduling (and resource access-control) algorithm to allocate processors and resources to jobs, determine whether all the jobs meet their deadlines.

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 *J₁*, *J₂*, *J₃*, *J₄*, with *J₁* 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 *J₂* 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 $J4$ in Figure above $s-(J4)$ is 2. $s+(J4)$ is 6. Its actual start time is in the range [2, 6]. Therefore, $J4$ 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 $f-$

(*J*4*)* is 20, but *f* +*(J*4*)* is 16. It is impossible for the inequality 20 ≤ *f (J*4*)* ≤ 16 to hold. Therefore, *J*4 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.