deerwalk
DWIT College

Real Time System

Roshan Pokhrel

1 Introduction

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

This syllabus is all about the real-time systems i.e. computing, communication, and information. A real-time system is required to complete its work and deliver its services on a timely basis. Examples of real-time systems include **digital control**, **command and control, signal processing**, and **telecommunication** systems. Every day these systems provide us with important services. When we drive, they control the engine and brakes of our car and regulate traffic lights. When we fly, they schedule and monitor the takeoff and landing of our plane, make it fly, maintain its flight path, and keep it out of harm's way. When we are sick, they may monitor and regulate our blood pressure and heart-beats. When we are well, they can entertain us with electronic games and joy rides.

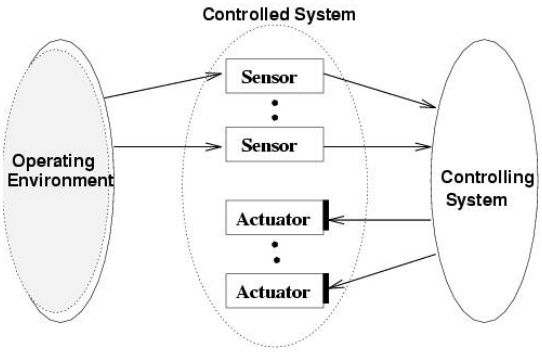
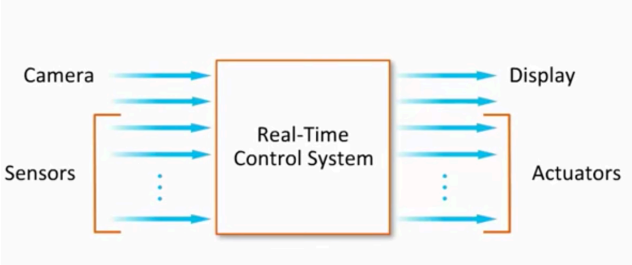


Figure 1.1: A Typical Real-Time System

From the examples above, you can see that malfunctions of some real-time systems can have serious consequences. We not only want such systems to work correctly and responsively but also want to be able to show that they indeed do. For this reason, a major emphasis of this syllabus is on techniques for validating real-time systems. By validation, we mean a rigorous demonstration that the system has the intended timing behavior.

Real-time systems are found in many everyday applications, from appliances, to automotive systems and even toys.

What all of these real-time systems have in common?



Any system, whether biological, electrical, mechanical, software or some combination of these, can be modeled as mapping of a set of inputs into a set of outputs. Typical real-time systems have inputs from sensors and imaging devices, from other systems, or from devices controlled by human operators. These inputs are processed so that the system can produce appropriate control signals to various actuators, motors and displays.

There are several important concepts related to real-time systems.

- Response time

The notion of the time needed by a system to respond to some input. That is, the time between the presentation of a set of inputs to a system and the realization of the required behavior, including the availability of all associated outputs is called the response time of the system. The set of inputs are often called the "stimulus" and the outputs are called the "response".

- Real-time system

A real-time system is a system that must satisfy explicit (and bounded) response time constraints or risk severe consequences. A real-time system is one whose logical correctness is based on both the correctness of the outputs and their timeliness. Therefore, any definition of a real-time system must

incorporate some notion of temporal constraint satisfaction. A real-time systems fails if it cannot satisfy one or more of the requirements stipulated in the system requirements specification.

With these concepts in mind, we can finally define a real-time system as one in which logical correctness is based on both the correctness of the outputs and their timeliness. But this definition of "real-time", while theoretically sound, is too broad to be useful most of the time. Practically speaking, every system has to produce its results within some reasonable deadline, whether that deadline is in microseconds, milliseconds, seconds, minutes, days or months. Therefore we need to specialize the definition of a real-time system.

- Failed system

We define a failed system as one in which performance is destroyed by failure to meet response-time constraints.

- Soft real-time system

For example, we define a soft real-time system as one in which performance is degraded but not destroyed by failure to meet response-time constraints. Many systems fit this definition from airline reservation systems to banking systems to even many computer games. In fact, every practical system is at least, a soft real-time system.

- Firm real-time system

A firm real-time system is one in which a few missed deadlines can be tolerated, but missing more than "a few" may lead to complete and catastrophic system failure. Many process control and telecommunications systems fall in this category.

- Hard real-time system

Finally, we define a hard real-time system as one in which failure to meet a single deadline may lead to complete and catastrophic system failure. For example, you can imagine the consequences of failing to meet a critical deadline in the control of a nuclear plant or vehicular control system.

System	Type	Effect of Missing Interrupts
Avionics weapons missile delivery	Hard	Missing target -- catastrophe
Navigation control for weed-killing robot	Firm	Damaged crops
Console hockey game	Soft	Degraded performance

Challenge to Real Time Systems Engineer?

One of the challenges for the real-time systems engineer is to find ways to transform hard real-time systems into firm or even soft real-time systems. That is, we want to find ways that allow critical systems to occasionally miss an important deadline, but not fail.

Consider the software applications shown in the table. The weapons delivery capability, say firing an offensive missile, for some fighter aircraft, would require a hard real-time system because missing the deadline between pressing the fire missile button and actually launching the missile could cause the target to be missed. On the other hand, the navigation control system for some robotic weed killer might be considered to be firm. Here missing an occasional deadline might lead to a few weeds being missed, or a cash crop being accidentally killed. But if the number of these misfires is small enough, these might be acceptable. Finally, in many applications, say, a simulated hockey game for amusement, missing more than a few deadlines might lead to user frustration, but not serious adverse consequences.

One important question to consider is, if real-time systems correctness is defined through meeting deadlines, where do deadlines come from?

They come from a variety of sources including: The physical laws underlying the dynamics of the system, Standards, Regulations, statues, and laws and Customer requirements. Sometimes the deadlines are arbitrary. Whatever the case the source of deadlines should be documented to help with tradeoff engineering and constraint relaxation during the systems engineering process and, later , in maintenance engineering.

1.1 Digital Control

Many real-time systems are embedded in sensors and actuators and function as digital controllers. Figure 1.2 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.

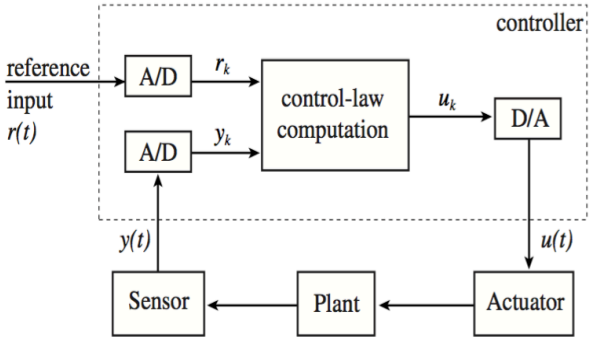


FIGURE 1–1 A digital controller.

Figure 1.2: A Digital Controller

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).

In the sampled data version, the inputs to the control-law computation are the sampled values y_k and r_k, for k = 0, 1, 2,…, which analog-to-digital converters produce by sampling and digitizing y(t) and r(t) periodically every T units of time. $e_k = r_k - y_k$ is the *kth* sample value of e(t). There are many ways to discretize the derivative and integral of e(t). For example, we can approximate the derivative of e(t)

for $(k-1)T \le t \le kT$ by $(e_k - e_{k-1})/T$ and use the trapezoidal rule of numerical integration to transform a continuous integral into a discrete form. The result is the following incremental expression of the kth output u_k:

P: *accounts for present values of the error. For example, if the error is large and positive, the control output will also be large and positive.*

I: *accounts for past values of the error. For example, if the current output is not sufficiently strong, error will accumulate over time, and the controller will respond by applying a stronger action.*

D: *accounts for possible future values of the error, based on its current rate of change.*

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt}$$

$$u_k = u_{k-2} + \alpha e_k + \beta e_{k-1} + \gamma e_{k-2} \qquad 1.1$$

α, β, and γ are proportional constants; they are chosen at design time. During the kth sampling period, the real-time system computes the output of the controller according to this expression.

From Eq.(1.1), we can see that during any sampling period (say kth), the control output u_k depends on the current and past measured values y_i for $i \le k$. The future measured values y_i's for $i > k$ in turn depend on u_k. Such a system is called a (feedback) control loop or simply a loop. We can implement it as an infinite timed loop:

```
set timer to interrupt periodically with period T ;
at each timer interrupt, do
     do analog-to-digital conversion to get y;
     compute control output u;
     output u and do digital-to-analog conversion;
end do;
```

Here, we assume that the system provides a timer. Once set by the program, the timer generates an interrupt every T units of time until its setting is cancelled.

The length *T* of time between any two consecutive instants at which *y*(*t*) and *r*(*t*) are sampled is called the *sampling period*. *T* is a key design choice. The behavior of the resultant digital controller critically depends on this parameter. Ideally we want the sampled data version to behave like the analog version. This can be done by making the sampling period small. However, a *small sampling period means more frequent control-law computation and higher processor-time demand*. We want a sampling period *T* that achieves a good compromise.

In making this selection, we need to consider two factors. The first is the perceived responsiveness of the overall system (i.e., the plant and the controller). Oftentimes, the system is operated by a person (e.g., a driver or a pilot). The operator may issue a command at any time, say at t. The consequent change in the reference input is read and reacted to by the digital controller at the next sampling instant. This instant can be as late as t + T. Thus, sampling introduces a delay in the system response. The operator will feel the system sluggish when the delay exceeds a tenth of a second. Therefore, the sampling period of any manual input should be under this limit.

The second factor is the dynamic behavior of the plant. We want to keep the oscillation in its response small and the system under control.

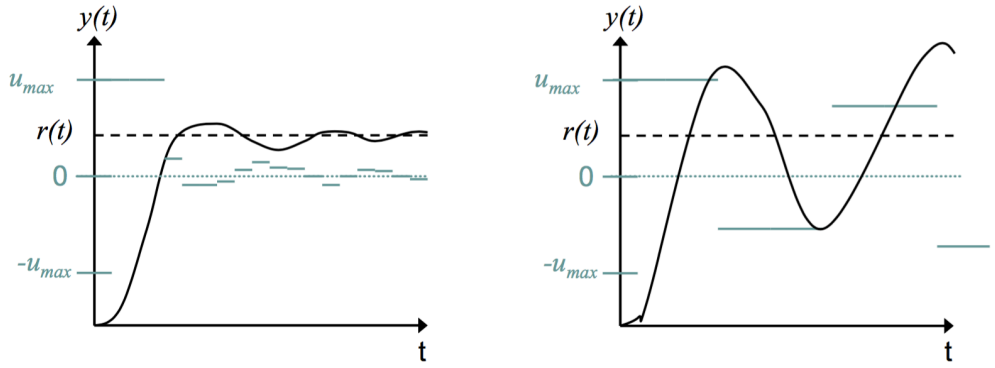


Figure 1.3: a. Short Sampling Period. b. Long Sampling Period

For short sampling period controller reaches r(t) while for large sampling period oscillation will result as system tries to adapt.

In general, the faster a plant can and must respond to changes in the reference input, the faster the input to its actuator varies, and the shorter the sampling period should be. We can measure the responsiveness of the overall system by its *rise time R*. Rise Time is usually defined as the amount of time that it takes the plant to go from 10% to 90% of the height of the step

Rule of thumb for signal processing applications:

Sampling period should be chosen in such a way that the rise time R is 10 to 20 times larger than the sample period T.

The above rule is also commonly stated in terms of the bandwidth, ω, of the system. The bandwidth of the overall system is approximately equal to $1/2R$ Hz. So the sampling rate (i.e., the inverse of sampling period) recommended above is 20 to 40 times the system bandwidth ω. The theoretical lower limit of sampling rate is dictated by *Nyquist sampling theorem.*

Any time-continuous signal of bandwidth ω *can be reproduced faithfully from its sampled values if and only if the sampling rate is* 2ω *or higher*.

Multirate System

A plant typically has more than one degree of freedom (*each of a number of independently variable factors affecting the range of states in which a system may exist, in particular any of the directions in which independent motion can occur.*). Its state is defined by multiple state variables (e.g., the rotation speed, temperature, etc. of an engine or the tension and position of a video tape). Therefore, it is monitored by multiple sensors and controlled by multiple actuators.

Because different state variables may have different dynamics, the sampling periods required to achieve smooth responses from the perspective of different state variables may be different. [*For example, because the rotation speed of a engine changes faster than its temperature, the required sampling rate for RPM (Rotation Per Minute) control is higher than that for the temperature control.*]. Therefore, multivariate digital controllers usually use multiple rates and therefore called multirate systems. In such systems periods are usually related in a harmonic way, i.e. each longer period is an integer multiple of the shortest period.

Note: *We can use the highest of all required sampling rates. This choice simplifies the controller software since all control laws are computed at the same repetition rate. However, some control-law computations are done more frequently than necessary; some processor time is wasted. To prevent this waste, multivariate digital controllers usually use multiple rates and are therefore called multirate systems.*

Note: Control-theoretical reason for choice of harmonic way: *some degree of coupling among individual single-output controllers in a system is inevitable. Consequently, the sampling periods of the controllers cannot be selected independently. A method for the design and analysis of multirate systems is the successive loop closure method. According to this method, the designer begins by selecting the sampling period of the controller that should have the fastest sampling rate among all the controllers. In this selection, the controller is assumed to be independent of the others in the system. After a digital version is designed, it is converted back into an analog form. The analog model is then integrated with the slower portion of the plant and is treated as a part of the plant. This step is then repeated for the controller that should have the fastest sampling rate among the controllers whose sampling periods remain to be selected. The iteration process continues until the slowest digital controller is designed. Each step uses the model obtained during the previous step as the plant. When the chosen sampling periods are harmonic, the analog models of the digital controllers used in this iterative process are exact. The only approximation arises from the assumption made in the first step that the fastest controller is independent, and the error due to this approximation can be corrected to some extent by incorporating the effect of the slower controllers in the plant model and then repeating the entire iterative design process.*

1.1.1 More Complex Control-Law Computations

The simplicity of a PID or similar digital controller follows from three assumptions

1. **sensor data give accurate estimates of the state variables that are monitored and controlled**
   This assumption is not valid when noise and disturbances inside or outside the plant prevent accurate observations of its state.
2. **sensor data give state of the plant (sometimes values are measured indirectly)**
   In general, sensors monitor some observable attributes of the plant. The values of the state variables must be computed from the measured values (i.e., digitized sensor readings).
3. **all parameters representing the dynamics of the plant are known**
   This assumption is not valid for some plants. (An example is a flexible robot arm. Even the parameters of typical manipulators used in automated factories are not known accurately.)

When any of the simplifying assumptions is not valid, the simple feedback loop discussed above no longer suffices. Since these assumptions are often not valid, often a digital controllers implemented as follows.

```
set timer to interrupt periodically with period T ;
at each timer interrupt, do
  sample and digitize sensor readings to get measured values;
  compute control output from measured and state-variable values;
  convert control output to analog form; estimate and update plant parameter;
  compute and update state variables;
end do;
```

The last two steps in the loop can increase the processor time demand of the controller significantly.

Two examples where the state update step is needed.

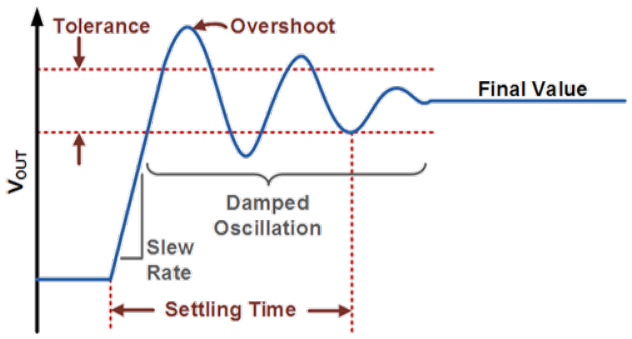
- **Deadbeat Control**

A discrete-time control scheme that has no continuous-time equivalence is deadbeat control. In response to a step change in the reference input, a dead-beat controller brings the plant to the desired state by exerting on the plant a fixed number (say n) of control commands. A command is generated every T seconds. (T is still called a sampling period.) Hence, the plant reaches its desired state in nT second.

In principle, the control-law computation of a deadbeat controller is also simple. The output produced by the controller during the kth sampling period is given by

$$u_k = \alpha \sum_{i=0}^{k} (r_i - y_i) + \sum_{i=0}^{k} \beta_i x_i \qquad 1.2$$

The constants α and β_i 's are chosen at design time. x_i is the value of the state variable in the i th sampling period. During each sampling period, the controller must compute an estimate of x_k from measured values y_i for $i \leq k$. In other words, the state update step in the above do loop is needed.



The deadbeat response has the following characteristics:

- Zero steady-state error
- Minimum rise time
- Minimum settling time
- Less than 2% overshoot/undershoot
- Very high control signal output

- **Kalman Filter**

Kalman filtering is a commonly used means to improve the accuracy of measurements and to estimate model parameters in the presence of noise and uncertainty.

To illustrate, we consider a simple monitor system that takes a measured value y_k every sampling period k in order to estimate the value x_k of a state variable. Suppose that starting from time 0, the value of this state variable is equal to a constant x. Because of noise, the measured value y_k is equal to $x + \varepsilon_k$, where ε_k is a random variable whose average value is 0 and standard deviation is σ_k. The Kalman filter starts with the initial estimate $\tilde{x}_1 = y_1$ and computes a new estimate each sampling period. Specifically, for $k > 1$, the filter computes the estimate $\tilde{x}_k$ as follows:

$$\tilde{x}_k = \tilde{x}_{k-1} + K_k(y_k - \tilde{x}_{k-1}) \qquad 1.3$$

In this expression,

$$K_k = \frac{P_k}{\sigma_k^2 + P_k} \tag{1.4}$$

is called the Kalman gain and P_k is the variance of the estimation error $\tilde{x}_k - x$; the latter is given by

$$P_k = E[(\tilde{x}_k - x)^2] = (1 - K_{k-1})P_{k-1} \tag{1.5}$$

This value of the Kalman gain gives the best compromise between the rate at which P_k decreases with k and the steady-state variance, that is, P_k for large k.

In a multivariate system, the state variable $\mathbf{x}_k$ is an n-dimensional vector, where n is the number of variables whose values define the state of the plant. The measured value $\mathbf{y}_k$ is an n'-dimensional vector, if during each sampling period, the readings of n' sensors are taken. We let $\mathbf{A}$ denote the measurement matrix; it is an $n \times n'$ matrix that relates the n' measured variables to the n state variables. In other words,

$$\mathbf{y}_k = \mathbf{A}\mathbf{x}_k + \mathbf{e}_k \tag{1.6}$$

The vector $\mathbf{e}_k$ gives the additive noise in each of the n' measured values. Eq. (1.3) becomes an n-dimensional vector equation

$$\tilde{\mathbf{x}}_k = \tilde{\mathbf{x}}_{k-1} + \mathbf{K}_k(\mathbf{y}_k - \mathbf{A}\tilde{\mathbf{x}}_{k-1}) \tag{1.7}$$

Similarly, Kalman gain $\mathbf{K}_k$ and variance $\mathbf{P}_k$ are given by the matrix version of Eqs. (1.4) and (1.5). So, the computation in each sampling period involves a few matrix multiplications and additions and one matrix inversion.

To know more about Kalman Filter: http://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/

1.2 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).

1.2.1 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.

Figure 1–4 shows a more complex example: the hierarchy of flight control, avionics, and air traffic control systems. It shows that some sensor data to both on-board controllers come from an air-data

system. This is a system of sensors and a computer. The computer computes flight and environment parameters (e.g., wind speed, true airspeed, static-air temperature, Mach number, altitude hold and rate) from aerodynamic and thermodynamic sensor data. These parameters are used by the controllers as well as being displayed for the pilot.

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 fix (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.

In general, there may be several higher levels of control. Take a control system of robots that perform assembly tasks in a factory for example. Path and trajectory planners at the second level determine the trajectory to be followed by each industrial robot. These planners typically take as an input the plan generated by a task planner, which chooses the sequence of assembly steps to be performed. In a space robot control system, there may be a scenario planner, which determines how a repair or rendezvous function should be performed. The plan generated by this planner is an input of the task planner.

1.2.2 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).

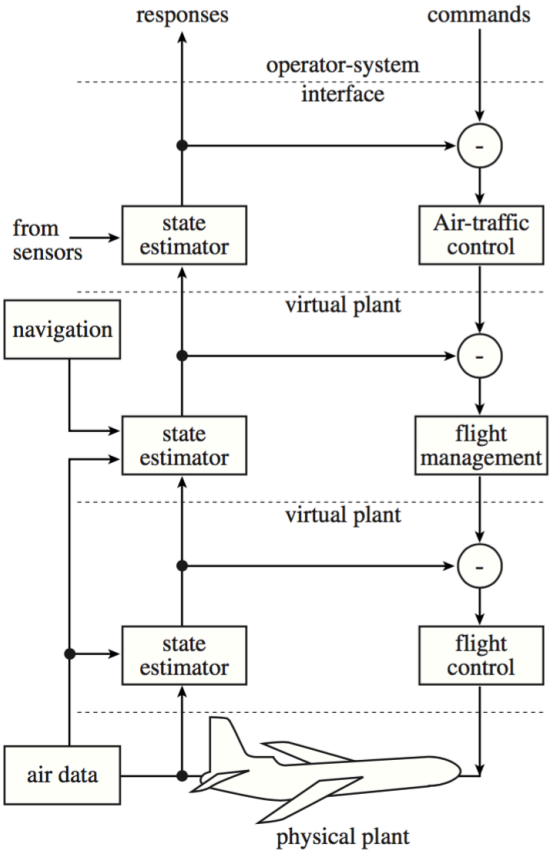


Figure 1.4: Air traffic/flight control hierarchy.

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.

Complexity and Timing Requirements.

The constrained optimization problems that a guidance (or path planning) system must solve are typically nonlinear. In principle, these problems can be solved using dynamic programming and mathematical programming techniques. In practice, however, optimal algorithms are rarely used because most of them are not only very compute intensive but also do not guarantee to find a usable solution. Heuristic algorithm used for guidance and control purposes typically consider one constraint at a time, rather than all the constraints at the same time. They usually start with an initial condition (e.g., in the case of a flight management systems, the initial condition includes the initial position, speed, and heading of the aircraft) and some initial solution and adjust the value of one solution parameter at a time until a satisfactory solution is found.

Fortunately, a guidance system does not need to compute its control laws as frequently as a digital controller. Often, this computation can be done off-line. In the case of a flight management system, for example, it needs to compute and store a climb speed schedule for use during takeoff, an optimum cruise trajectory for use en route, and a descent trajectory for landing. This computation can be done before takeoff and hence is not time-critical. While in-flight, the system still needs to compute some control laws to monitor and control the transitions between different flight phases (i.e., from climb to cruise and cruise to descent) as well as algorithms for estimating and predicting times to waypoints, and so forth. These time- critical computations tend to be simpler and more deterministic and have periods in order of seconds and minutes. When the pre-computed flight plan needs to be updated or a new one computed in-flight, the system has minutes to compute and can accept suboptimal solutions when there is no time.

Other Capabilities

The complexity of a higher-level control system arises for many other reasons in addition to its complicated control algorithms. It often interfaces with the operator and other systems. To interact with the operator, it updates displays and reacts to operator commands. By other systems, we mean those outside the control hierarchy. An example is a voice, telemetry, or multimedia communication system that supports operator interactions. Other examples are radar and navigation devices. The control system may use the information provided by these devices and partially control these devices.

An avionic or flight management system has these capabilities. One of its functions is to update the display of radar, flight path, and air-data information. Like keyboard monitoring, the display updates must done no less frequently than once every 100 milliseconds to achieve a satisfactory performance. Similarly, it periodically updates navigation data provided by inertial and radio navigation aids.

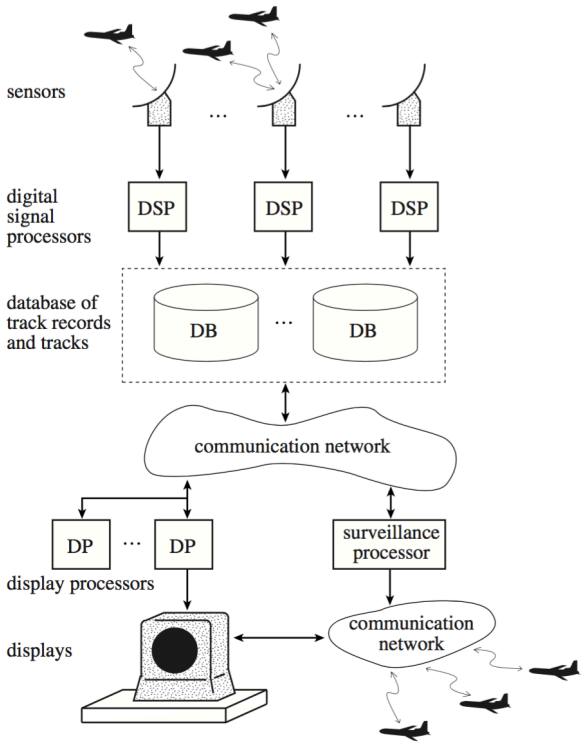


FIGURE 1–5 An architecture of air traffic control system.

An avionics system for a military aircraft also does tracking and ballistic computations and coordinates radar and weapon control systems, and it does them with repetition periods of a few to a few hundred milliseconds. The workload due to these functions is demanding even for today's fast processors and data links.

1.2.3 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.

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. 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 aircraft responds by sending to the ATC system its "state variables": identifier, position, altitude, heading, and so on.

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.

1.3 Signal processing

Most signal processing applications have some kind of real-time requirements. Digital filtering, video and voice compressing/decompression, and radar signal processing must have response times of under a few milliseconds to a few seconds

1.3.1 Processing Bandwidth Demands

A real-time signal processing application computes in each sampling period one or more outputs. Each output $x(k)$ is a weighted sum of n inputs $y(i)$'s:

$$x(k) = \sum_{i=1}^{n} a(k, i)y(i) \qquad 1.8$$

In the simplest case, the weights, a(k, i)'s, are known and fixed. However, in the case of adaptive filtering applications (e.g., echo suppression), each weight changes with time and must be updated. The update of each weight typically takes one multiplication and one addition each sampling period.

In essence, this computation transforms the given representation of an object (e.g., a voice, an image or a radar signal) in terms of the inputs, $y(i)$'s, into another representation in terms of the outputs, $x(k)$'s. Different sets of weights, $a(k,i)$'s, give different kinds of transforms. This expression tells us that the time required to produce an output is $O(n)$.

The processor time demand of an application also depends on the number of outputs it is required to produce in each sampling period. At one extreme, a digital filtering application (e.g., a filter that suppresses noise and interferences in speech and audio) produces one output each sampling period. The sampling rates of such applications range from a few kHz to tens of kHz.

Some other signal processing applications are more computationally intensive. The number of outputs may also be of order n, and the complexity of the computation is $O(n^2)$ in general.

An example is image compression. Most image compression methods have a transform step. This step transforms the space representation of each image into a transform representation (e.g., a hologram). To illustrate the computational demand of a compression process, let us consider an $m \times m$ pixel, 30 frames per second video. Suppose that we were to compress each frame by first computing its transform. The number of inputs is $n = m^2$. The transformation of each frame takes m^4 multiplications and additions. If m is 100, the transformation of the video takes 3×10^9 multiplications and additions per second! One way to reduce the computational demand at the expense of the compression ratio is to divide each image into smaller squares and perform the transform on each square. This indeed is what the video compression standard does. Each image is divided into squares of 8×8 pixels. In this way, the number of multiplications and additions performed in the transform stage is reduced to $64m^2$ per frame (in the case of our example, to 1.92×10^7). Today, there is a broad spectrum of Digital Signal Processors (DSPs) designed specifically for signal processing applications. Computationally intensive signal processing applications run on one or more DSPs. In this way, the compression process can keep pace with the rate at which video frames are captured.

1.3.2 Radar System

Figure 1.6 shows a block diagram of a radar signal processing and tracking system. The 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

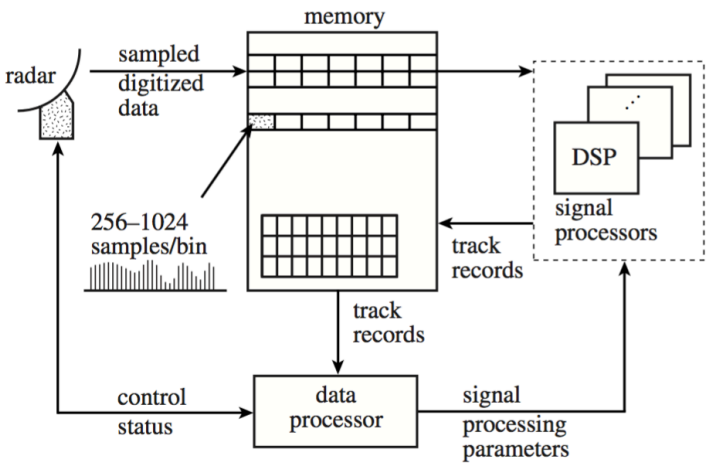


FIGURE 1–6 Radar signal processing and tracking system.

and analysis.

Radar Signal Processing. 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 $c = 3 \times 10^8$ 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.

Specifically, the system divides the time during which the antenna dwells to collect the echo signal into small disjoint intervals. The digital sampled values of the echo signal collected during each range interval are placed in a buffer, called a bin in Figure 1.6. The sampled values in each bin are the inputs used by a digital signal processor to produce outputs of the form given by Eq. (1.8). These outputs represent a discrete Fourier transform of the corresponding segment of the echo signal. Based on the characteristics of the transform, the signal processor decides whether there is an object in that distance range. If there is an object, it generates a *track record* containing the position and velocity of the object and places the record in the shared memory.

The time required for signal processing is dominated by the time required to produce the Fourier transforms, and this time is nearly deterministic. The time complexity of Fast Fourier Transform (FFT) is $O(n\log n)$, where n is the number of sampled values in each range bin. n is typically in the range from 128 to a few thousand. So, it takes roughly 10^3 to 10^5 multiplications and additions to generate a Fourier transform.

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 nonexisting 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. In the example in Figure 1.6, the tracker runs on one or more data processors which communicate with the signal processors via the shared memory.

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. 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.

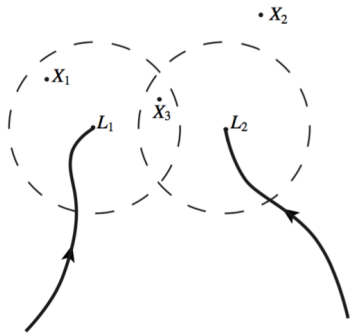


FIGURE 1–7 Gating process.

Figure 1–7 illustrates this process. At the start, the tracker computes the predicted position (and velocity) of the object on each established trajectory. In this example, there are two established trajectories, L_1 and L_2. We also call the predicted positions of the objects on these tracks L_1 and L_2. X_1, X_2, and X_3 are the measured values given by three track records. X_1 is assigned to L_1 because it is within distance G from L_1. X_3 is assigned to both L_1 and L_2 for the same reason. On the other hand, X_2 is not assigned to any of the trajectories. It represents either a false return or a new object. Since it is not possible to distinguish between these two cases, the tracker hypothesizes that X_2 is the position of a new object. Subsequent radar data will allow the tracker to either validate or invalidate this hypothesis. In the latter case, the tracker will discard this trajectory from further consideration.

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

- the radar signal is strong and interference is low (and hence false returns are few) and
- 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.

There are many data association algorithms. One of the most intuitive is the nearest neighbor algorithm. This algorithm works as follows:

1. Examine the tentative assignments produced by the gating step.
    a. For each trajectory that is tentatively assigned a single unique measured value, assign the measured value to the trajectory. Discard from further examination the trajectory and the measured value, together with all tentative assignments involving them.
    b. For each measured value that is tentatively assigned to a single trajectory, discard the tentative assignments of those measured values that are tentatively assigned to this trajectory if the values are also assigned to some other trajectories.
2. Sort the remaining tentative assignments in order of non-decreasing distance.
3. Assign the measured value given by the first tentative assignment in the list to the corresponding trajectory and discard the measured value and trajectory.
4. Repeat step (3) until the list of tentative assignments is empty.

In the example in Figure 1.7, the tentative assignment produced by the gating step is ambiguous. Step (1a) does not eliminate any tentative assignment. However, step (1b) finds that X_1 is assigned to only L_1, while X_3 is assigned to both L_1 and L_2. Hence, the assignment of X_3 to L_1 is discarded from further consideration. After step (1), there still are two tentative assignments, X1 to L1 and X3 to L2. Step (2)

leaves them in this order, and the subsequent steps make these assignments. X2 initiates a new trajectory. If during subsequent scans, no measured values are assigned to the new trajectory, it will be discarded from further consideration.

Complexity and Timing Requirements.

In contrast to signal processing, the amounts of processor time and memory space required by the tracker are data dependent and can vary widely. When there are n established trajectories and m measured values, the time complexity of gating is $O(nm \log m)$. (This can be done by first sorting the m measured values according to their distances from the predicted value for each of the established trajectories and then comparing the distances with the track gate G.) In the worst case, all m measured values are tentatively assigned to all n trajectories in the gating step. The nearest neighbor algorithm must sort all nm tentative assignments and hence has time complexity $O(nm \log nm)$

1.4 Real time applications

The focus of this topic is on the characteristics and requirements of two most common real-time applications. They are real-time databases and multimedia applications.

1.4.1 Real-Time Database

A real-time database is a database system which uses real-time processing to handle workloads whose state is constantly changing. This differs from traditional databases containing persistent data, mostly unaffected by time.

For example, a stock market changes very rapidly and is dynamic. The graphs of the different markets appear to be very unstable and yet a database has to keep track of current values for all of the markets of the Stock Exchange. Real-time processing means that a transaction is processed fast enough for the result to come back and be acted on right away. Real-time databases are useful for accounting, banking, law, medical records, multi-media, process control, reservation systems, and scientific data analysis.

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. In contrast, an underlying assumption of nonreal-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).

TABLE 1–1 Requirements of typical real-time databases

Applications	Size	Ave. Resp. Time	Max Resp. Time	Abs. Cons.	Rel. Cons.	Permanence
Air traffic control	20,000	0.50 ms	5.00 ms	3.00 sec.	6.00 sec.	12 hours
Aircraft mission	3,000	0.05 ms	1.00 ms	0.05 sec.	0.20 sec.	4 hours
Spacecraft control	5,000	0.05 ms	1.00 ms	0.20 sec.	1.00 sec.	25 years
Process control		0.80 ms	5.00 sec	1.00 sec.	2.00 sec	24 hours

- **Absolute Temporal Consistency**

The temporal quality of real-time data is often quantified by parameters such as age and temporal dispersion. The age of a data object measures how up-to-date the information provided by the object is. Intuitively, the *age of an image object* at any time is the length of time since the instant of the last update, that is, when its value is made equal to that of the real-world object it represents. The age of a data object whose value is computed from the values of other objects is equal to the oldest of the ages of those objects.

A set of data objects is said to be *absolutely (temporally) consistent* if the maximum age of the objects in the set is no greater than a certain threshold.

As an example, "aircraft mission" listed in the table refers to the kind of database used to support combat missions of military aircraft. A fighter jet and the targets it tracks move at supersonic speeds. Hence the information on where they are must be less than 50 milliseconds old. On the other hand, an air traffic control system monitors commercial aircraft at subsonic speeds; this is why the absolute temporal consistency threshold for air traffic control is much larger.

- **Relative Temporal Consistency**

A set of data objects is said to be *relatively consistent* if the maximum difference in ages of the objects in the set is no greater than the relative consistency threshold used by the application. For some applications the absolute age of data may not be as important as the differences in their ages. An example is a planning system that correlates traffic densities along a highway with the flow rates of vehicles entering and exiting the highway. The system does not require the most up-to-date flow rates at all interchanges and hence can tolerate a relatively large age (e.g., two minutes). However, if the difference in the ages of flow rates is large (e.g., one minute), the flow rates no longer give a valid snapshot of the traffic scenario and can lead the system to wrong conclusions.

- **Consistency Models**

Concurrency control mechanisms, such as two-phase locking, have traditionally been used to ensure the serializability of read and update transactions and maintain data integrity of nonreal-time databases. These mechanisms often make it more difficult for updates to complete in time. Late updates may cause the data to become temporally inconsistent. Yet temporal consistency of real-time data is often as important as, or even more important than, data integrity. Concurrency control mechanisms required to maintain a weaker sense of consistency tend to improve the timeliness of updates and reads.

1.4.2 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. Therefore, a video stream, as well as the associated audio stream, is invariably compressed as soon as it is captured.

Example: 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, 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.

MPEG Compression/Decompression

A video compression standard is MPEG-2. The standard makes use of three techniques.

1. motion compensation for reducing temporal redundancy,
2. discrete cosine transform for reducing spatial redundancy, and
3. entropy encoding for reducing the number of bits required to encode all the information.

Depending on the application, the compressed bit rate ranges from 1.5 Mbits/sec to 35 Mbits/sec. However, the achievable compression ratio depends on the content of the video.

- ***Motion Estimation***

The first step of compression is motion analysis and estimation. Because consecutive video frames are not independent, significant compression can be achieved by exploiting inter-frame dependency. This is the rationale behind the motion estimation step. The motion-compensation techniques used in this step assume that most small pieces of the image in the current frame can be obtained either by translating in space corresponding small pieces of the image in some previous frame or by interpolating some small pieces in some previous and subsequent frames. For this reason, each image is divided into 16 × 16-pixel square pieces; they are called *major blocks*. The luminance component of each major block consists of four 8 × 8 pixel blocks. Each of the chrominance components has only a quarter of this resolution. Hence, each chrominance component of a major block is an 8 × 8 pixel block.

- ***Discrete Cosine Transform and Encoding***

In the second step, a cosine transform is performed on each of the 8 × 8 pixel blocks produced by the coder after motion estimation. By carefully quantizing the elements of an 8×8 transform matrix obtained from transforming the original matrix that gives the 8×8 values of a pixel block, encoding the entries in the transform matrix as 2-tuples (run length, value), and using a combination of variable-length and fixed-length codes to reduce the bit rate, significant compression is achieved.

- ***Decompression***

During decompression, the decoder first produces a close approximation of the original matrix (i.e., an 8 × 8 pixel block) by performing an inverse transform on each stored transform matrix. (The computation of an inverse transform is the essentially the same as the cosine transform.) It then reconstruct the images in all the frames from the major and difference blocks.

Real-Time Characteristics

As we can see from the above description, video compression is a computational-intensive process. For batch applications such as video on demand, compression is done in batch and off-line, while it must be an on-line process for interactive applications (e.g., teleconferencing). Decompression should be done just before the time the video and audio are presented, in other words, on the just-in-time basis. Today, compression and decompression functions are often handled by an affordable special-purpose processor (e.g., the mmx), rather than by general-purpose processors.

To a great extent, the timing requirements of a multimedia application follow from the required video and audio quality. From the user's point of view, the quality of a video is partially characterized by its frame rate and resolution. A video of standard television quality consists of 30 frames per second. High-definition television uses 60 frames per second to give the picture even less flicker. On the other hand, much lower frame rates (e.g., 10–20) are tolerable for other applications, such as teleconferencing.

The term resolution roughly refers to the number of pixels in each frame (i.e., the size of the frame) and the number of bits used to encode each pixel (i.e., intensity and color resolution). Together, the resolution and frame rate of a video tell us roughly the amount of time required to compress/decompress it and the amounts of storage and transmission bandwidth required to store and transmit it.

Similarly, the quality of an audio component depends on the sampling rate and granularity used to digitize the audio signal. The total bit rate of an audio ranges from 16 Kbits per second for telephone speech quality to 128 Kbits per second for CD quality. Some loss of audio data is unavoidable, because the system may discard data during congestion and some data may arrive too late to be presented to the user, and so on. The quality of speech is usually tolerable when the loss rate is under one percent.

Another dimension of quality of a multimedia application is ***lip synchronization***. This term refers to the temporal synchronization of the video frames and related audio data units. In the case where the video is that of a speaker, the speaker's lips should appear to move to make the accompanied speech. Experimental results indicate that the time interval between the display of each frame and the presentation of the corresponding audio segment should ideally be no more than 80 msec and should definitely be no more than 160 msec for sake of achieving lip synchronization.

For batch applications, a system can often provide the desired quality by trading between real-time performance and space usage. For example, we want to present the audio to the user without pauses. This can clearly be achieved if there is little or no jitter (i.e., variation) in the delay suffered by audio data packets as they are transmitted over the network. However, the system can nevertheless deliver good audio despite large jitter by providing a sufficiently large amount of buffer to smooth out the jitter.

Finally, our ears are extremely sensitive to glitches and pauses in audio, and an end-to-end delay in the order of a few hundred milliseconds significantly decreases the quality of a conversation. Therefore, both end-to-end response time and response time jitter are important for interactive applications.

1.5 SUMMARY

As a summary, we divide real-time applications into the following four types according to their timing attributes.

Purely cyclic: Every task in a purely cyclic application executes periodically. Even I/O operations are polled. Moreover, its demands in (computing, communication, and storage) resources do not vary significantly from period to period. Most digital controllers, exemplified by the flight control system and real-time monitors are of this type.

Mostly cyclic*:* Most tasks in a mostly cyclic system execute periodically. The system must also respond to some external events (fault recovery and external commands) asynchronously. Examples are modern avionics and process control systems.

Asynchronous and somewhat predictable*:* In applications such as multimedia communication, radar signal processing, and tracking, most tasks are not periodic. The duration between consecutive executions of a task may vary considerably, or the variations in the amounts of resources demanded in different periods may be large. However, these variations have either bounded ranges or known statistics.

Asynchronous and unpredictable*:* Applications that react to asynchronous events and have tasks with high run-time complexity belong to this type. An example is intelligent real-time control systems.

An orthogonal dimension is the size of the application. Like nonreal-time applications, some of them run on one or a few microprocessors, even on hand-held devices, while others run on tens and hundreds of computers. They are commonly labeled as uniprocessor, multi- processor, or distributed systems. As you will see shortly, we will not emphasize this aspect. Of course, a solution suited for a system containing a few microprocessors (e.g., an automotive control system) may not be applicable to a large distributed system (e.g., air traffic control system) and vice versa. The subsequent chapters will try to make which ones are which clear to you.

2 Hard versus Soft Real-Time Systems

What are the characteristics that distinguish RTS from non-RTS applications?

2.1 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 is called 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. We call them a control-law computation, a FFT computation, a packet transmission, and so on, only when we want to be specific about the kinds of work, that is, the types of jobs.

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. We call all these resources *processors.*

2.2 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 $J_0, J_1, \ldots, J_k, \ldots$. The release time of the job J_k 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.

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.

In this example, as in many others, it is more natural to state the timing requirement of a job in terms of its *response time*, that is, the length of time from the release time of the job to the instant when it completes. We call the maximum allowable response time of a job its *relative deadline*. Hence the relative deadline of every control-law computation job mentioned above is 100 or 50 msec. The deadline of a job, sometimes called its *absolute deadline*, is equal to its release time plus its relative deadline.

In general, 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 deadlines. We will discuss the parameters needed to specify those constraints when they arise, but in most of this book, we are concerned primarily with this simple form.

2.3 Hard and soft timing constraints

Timing constraints is divided into two types: hard and soft.

2.3.1 Common Definitions

According to a commonly used definition, 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 stated in terms of the usefulness of results (and therefore the over-all 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 dead-line 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.

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. An example is that the system recovery job or a point-of-sales transaction completes within one minute 99.999 percent of the time. In other words, the probability of failure to meet the one-minute relative deadline is 10^{-5}. This definition ignores completely the consequence of a timing failure. In our example, if the failure of an on-time recovery could cause loss of life and property, we would require a rigorous demonstration that the timing failure probability is indeed never more than 10^{-5}.

2.3.2 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 meet the timing constraint. *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 meet some *statistical constraint* (i.e., a timing constraint specified in terms of statistical averages) suffices, then the timing constraint of the job is soft. The satisfaction of statistical constraints (e.g., the average number of missed deadlines per minute is two or less) can usually be demonstrated with a performance profile somewhat more thorough than those used to demonstrate the performance of general interactive systems.

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

2.4 Hard real-time systems

The requirement that all hard timing constraints must be validated invariably places many restrictions on the design and implementation of hard real-time applications as well as on the architectures of hardware and system software used to support them.

2.4.1 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.

Example: 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.

Jobs in some non-embedded systems may also have hard deadlines. An example is a critical information system that must be highly available: The system must never be down for more than a minute. Because of this requirement, reconfiguration and recovery of database servers and network connections in the system must complete within a few seconds or tens of seconds, and this relative deadline is hard.

In general, if safety or property loss is involved, the designer/builder of the system has the burden of proof that bad things will never happen. Whenever it is not possible to prove without doubt that a few timing constraint violations will not jeopardize the safety of users or availability of some critical infrastructure, we take the safe approach and insist on the satisfaction of all timing constraints, even though the requirement may be unnecessarily stringent.

2.4.2 More on Hard Timing Constraints

In principle, our definition of hard and soft timing constraints allows a hard timing constraint to be specified in any terms. Examples are

- deterministic constraints (e.g., the relative deadline of every control-law computation is 50 msec or the response time of at most one out of five consecutive control-law computations exceeds 50 msec);
- probabilistic constraints, that is, constraints defined in terms of tails of some probability distributions (e.g., the probability of the response time exceeding 50 milliseconds is less than 0.2); and
- constraints in terms of some usefulness function (e.g., the usefulness of every control- law computation is 0.8 or more).

In practice, hard timing constraints are rarely specified in the latter two ways. Mostly deterministic hard timing constraints is used. Why?

Because it is much easier to validate deterministic timing constraints than probabilistic constraints and those expressed in terms of usefulness functions.

2.5 Soft real-time constraints.

A system in which jobs have soft deadlines is a *soft real-time system*. The developer of a soft real-time system is rarely required to prove rigorously that the system surely meet its real- time performance objective. Examples of such systems include on-line transaction systems and telephone switches, as well as electronic games. The less rigorous validation required of the system and, often, more relaxed timing constraints allow the developer to consider other performance metrics equally seriously. Meeting all deadlines is not the only consideration, sometimes, not even the primary consideration. An occasional missed deadline or aborted execution is usually considered tolerable; it may be more important for such a system to have a small average response time and high throughput.

A system may have critical timing requirements but is nevertheless considered to be a soft real-time system. An example is a stock price quotation system. It should update the price of each stock each time the price changes. Here, a late update may be highly undesirable, because the usefulness of a late price quotation decreases rapidly with time. However, in a volatile market when prices fluctuate at unusually high rates, we expect that the system cannot keep up with every price change but does its best according to some criteria. Occasional late or missed updates are tolerated as a trade-off for other factors, such as cost and availability of the system and the number of users the system can serve.

The timing requirements of soft real-time systems are often specified in probabilistic terms. Take a telephone network for example. In response to our dialing a telephone number, a sequence of jobs executes in turn, each routes the control signal from one switch to another in order to set up a connection through the network on our behalf. We expect that our call will be put through in a short time. To ensure that this expectation is met most of the time by the network, a timing constraint may be imposed on this sequence of jobs as a design objective (e.g., the sequence must complete in no more than 10 seconds for 95 percent of the time and in no more than 20 seconds for 99.95 percent of the time). The users are usually satisfied if after extensive simulation and trial use, the system indeed appears to meet this requirement.

As a final example, let us consider multimedia systems that provide the user with services of "guaranteed" quality. For example, a frame of a movie must be delivered every thirtieth of a second, and the difference in the times when each video frame is displayed and when the accompanied speech is presented should be no more than 80 msec. In fact, it is common to subject each new video stream to be transmitted by a network to an acceptance test. If the network cannot guarantee the satisfaction of timing constraints of the stream without violating the constraints of existing streams, the new stream is rejected, and its admission is requested again at some later time. However, the users are often willing to tolerate a few glitches, as long as the glitches occur rarely and for short lengths of time. At the same time, they are not willing to pay the cost of eliminating the glitches completely. For this reason, we often see timing constraints of multimedia systems guaranteed on a statistical basis, (e.g., the average number of late/lost frames per minute is less than 2). Moreover, users of such systems rarely demand any proof that the system indeed honor its guarantees. The quality-of-service guarantee is soft, the validation requirement is soft, and the timing constraints defining the quality are soft.

3 Reference Model of Real-Time Systems

When we study how given applications should be scheduled on a system of processors and resources and how we can determine whether the resultant system meets all its timing requirements, many specific details about each application (e.g., whether it is a Kalman filter computation or a sorting routine, whether it is implemented in Ada or C++) are not relevant. We do not want to keep track of these details and let them cloud relevant issues. Similarly, we do not want to keep track of irrelevant properties of system resources (e.g., whether the processor is by Intel or Motorola or whether the transmission medium is cable or fiber.) A good model abstracts the irrelevant details. It allows us to focus on the timing properties and resource requirements of system components and the way the operating system allocates the available system resources among them. By focusing on the relevant characteristics of the system, we can reason better about the timing behavior of each component and the overall system. By describing the algorithms used to schedule the applications and the methods for validating their timing constraints abstractly, rather than in implementation specific terms, we can appreciate their general applicability better.

In this chapter, we describe a reference model of real-time systems. According to this model, each system is characterized by three elements:

a. a workload model that describes the applications supported by the system,
b. a resource model that describes the system resources available to the applications, and
c. algorithms that define how the application system uses the resources at all times.

The model aims at being good. It has a sufficiently rich set of features in terms of which we can describe the relevant characteristics of a wide spectrum of real-time applications and the properties of the underlying platform. We can describe a system in a sufficiently faithful manner in terms of the model so we can analyze, simulate, and even emulate the system based on its description. In particular, analysis and simulation tools can use such a system description as input to produce accurate estimates of the real-time performance and the associated overhead of the system.

3.1 Processor and resources

All the system resources are divided into two major types: processors and resources.

Processors

Processors are often called servers and active resources; computers, transmission links, disks, and database server are examples of processors. They carry out machine instructions, move data from one place to another, retrieve files, process queries, and so on. Every job must have one or more processors in order to execute and make progress toward completion. Processor is denoted by P and m processor in the system is denoted by $P_1, P_2, P_3, \dots, P_m$

Distinction between types of processors:

Two processors are of the same type if they are functionally identical and can be used interchangeably. Hence two transmission links with the same transmission rate between a pair of sender and receiver are processors of the same type; processors in a Symmetrical Multiprocessor (SMP) system are of the same type, and so on. Processors that are functionally different, or for some other reason cannot be used interchangeably, are of different types. CPUs, transmission links, and disks are of different types, because they are functionally different. A transmission link connecting an on-board flight management system to the ground controller is a different type of processor from the

link connecting two air traffic control centers even when the links have the same characteristics, because they cannot be used interchangeably.

When we want focus on how the jobs on each processor are scheduled, how the jobs on different processors are synchronized, and how well the processors are utilized, there is no need to be concerned with whether the processors are identical or different.

Resources

By resources, we will specifically mean passive resources. Examples of resources are memory, sequence numbers, mutexes (*In computer programming, a mutex is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously. When a program is started, a mutex is created with a unique name*.), and database locks.

A job may need some resources in addition to the processor in order to make progress. One of the attributes of a processor is its speed. We assume that the rate of progress a job makes toward its completion depends on the speed of the processor on which it executes. We can explicitly model this dependency by making the amount of time a job requires to complete a function of the processor speed. In contrast, we do not associate speed with a resource. In other words, how long a job takes to complete does not depend on the speed of any resource it uses during execution.

For example, a computation job may share data with other computations, and the data may be guarded by semaphores. We model (the lock of) each semaphore as a resource. When a job wants to access the shared data guarded by a semaphore R, it must first lock the semaphore, and then it enters the critical section of the code where it accesses the shared data. In this case, we say that the job requires the resource R for the duration of this critical section.

As another example, we consider a data link that uses the sliding-window scheme to regulate message transmission. Only a maximum number of messages are allowed to be in transit (i.e., they have been transmitted but their reception has not yet been positively acknowledged). One way to implement this scheme is for the sender to maintain a window of valid sequence numbers. The window is moved forward as messages transmitted earlier are acknowledged by the receiver. A message waiting to be transmitted must first be given one of the valid sequence numbers before it can be transmitted. We model the transmission of a message as a job; the job executes when the message is being transmitted. This job needs the data link, as well as a valid sequence number. The data link is a processor. The sequence numbers are units of the sequence-number resource.

Types of Resources:

a. Reusable

The resources that are not consumed during use are called reusable resources. Like above examples.

b. Non-Reusable

The resources that do not exists after use are called non-reusable resources. A message produced by a process and consumed by another process is not reusable because it no longer exists after use.

R is used to denote resources. In our discussion, a “resource” almost always mean a reusable resource. The statement that the system contains ρ resources means that there are ρ types of serially reusable resources, each resource may have one or more units, and each unit is used in a mutually exclusive manner. For example, if the sliding window of a data link has eight valid sequence numbers, then there are eight units of the sequence-number resource (type). The write lock on a file is a resource that has only one unit because only one job can have the lock at a time. A resource that can be shared by a finite number x of jobs is modeled as a resource (type) that has x units, each of which can be used by only one job at a time.

Plentiful Resource

To prevent our model from being cluttered by irrelevant details, we typically omit the resources that are plentiful.

A resource is ***plentiful*** if no job is ever prevented from execution by the lack of this resource. A resource that can be shared by an infinite number of jobs (e.g., a file that is readable simultaneously by all) need not be explicitly modeled and hence never appears in our model.

Example:

- **The memory resource**: Clearly, memory is an essential type of resource. All computing and communication systems have this resource. We omit it from our model whenever we can account for the speed of the memory by the speed of the processor and memory is not a bottleneck of the system. By memory not being the bottleneck, we mean that whenever a job is scheduled to execute, it always has a sufficient amount of memory. Many real-time systems are designed so that this assumption is valid. When this assumption is not valid, the timing behavior of the system critically depends on how memory is allocated to jobs. Clearly, the model of the system must include the memory resource in this case.
- **I/O bus**: Every computation job must have the I/O bus in addition to the processor to execute. Most of the time, we model the I/O bus as a resource, oftentimes a plentiful and therefore ignored resource. However, when we want to study how long I/O activities may delay the completion of computation jobs that share the I/O bus or when we want to determine the real-time performance of an I/O bus arbitration scheme, we want to model the bus as a resource or a processor.

3.2 Temporal parameter of real-time workload

The workload on processors consists of jobs, each of which is a unit of work to be allocated processor time and other resources. A set of related jobs that execute to support a function of the system is a task. We typically assume that many parameters of hard real-time jobs and tasks are known at all times; otherwise, it would not be possible to ensure that the system meet its hard real-time requirements. The number of tasks (or jobs) in the system is one such parameter. In many embedded systems, the number of tasks is fixed as long as the system remains in an operation mode. The number of tasks may change when the system operation mode changes, and the number of tasks in the new mode is also known. Moreover, these numbers are known a priori before the system begins execution.

Example: Flight control system (Figure 1–3). During cruise mode, the system has 12 tasks (i.e., 3 30-Hz avionics tasks, 3 30-Hz computations and 2 90-Hz computations, plus 180-Hz computation, validation, output and built-in-test tasks). If the system triply replicates all control-law computations during landing, the number of tasks increases to 24 when it operates in the landing mode.

In some other systems, however, the number of tasks may change as tasks are added and deleted while the system executes. As an example, in an air traffic control system, each surveillance task monitors an aircraft. The number of such tasks changes as tasks are added and deleted when aircraft enter and leave the coverage area. Nevertheless, the number of tasks with hard timing constraints is known at all times. This assumption is valid. When the satisfaction of their timing constraints is to be guaranteed, the admission and deletion of hard real-time tasks are usually done under the control of the run-time system (e.g., by having the application system request admission and deletion of tasks). For this purpose, the system must maintain information on all existing hard real-time tasks, including the number of such tasks.

Each job J_i is characterized by its:

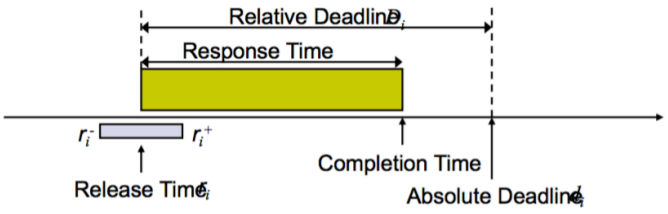
- Temporal parameters
- Functional parameters
- Resource parameters and
- Interconnection parameters

Its temporal parameters tell us its timing constraints and behavior. Its interconnection parameters describe how it depends on other jobs and how other jobs depend on it. Its functional parameters specify the intrinsic properties of the job. Finally, its resource parameters give us its resource requirements.

The release time, absolute deadline, and relative deadline of a job J_i; these are temporal parameters.

r_i, d_i, and D_i, respectively, denotes temporal parameters and the time interval $(r_i, d_i]$ between the release time and absolute deadline of the job J_i its *feasible interval*. d_i and D_i are usually derived from the timing requirements of J_i, jobs in the same task as J_i, and the overall system.

3.2.1 Fixed, Jittered, and Sporadic Release Times



In many systems, we do not know exactly when each job will be released. In other words, we do not know the actual release time r_i of each job J_i; only that r_i is in a range $[r_i^-, r_i^+]$. r_i can be as early as the earliest release time r_i^- and as late as the latest release time r_i^+. Indeed, some models assume that only the range of r_i is known and call this range the *jitter* in r_i, or *release-time jitter*. Sometimes, the jitter is negligibly small compared with the values of other temporal parameters. If, for all practical purposes, we can approximate the actual release time of each job by its earliest or latest release time, then we say that the job has a fixed release time.

Almost every real-time system is required to respond to external events which occur at random instants of time. When such an event occurs, the system executes a set of jobs in response. The release times of these 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 time instants. For example, the pilot may disengage the autopilot system at any time. When this occurs, the autopilot system changes from cruise mode to standby mode. The jobs that execute to accomplish this mode change are sporadic jobs.

The release times of sporadic and aperiodic jobs are random variables. The model of the system gives the probability distribution $A(x)$ of the release time of such a job, or when there is a stream of similar sporadic or aperiodic jobs, the probability distribution of *interrelease time* (i.e., the length of the time interval between the release times of two consecutive jobs in the stream). $A(x)$ gives us the probability that the release time of the job is at or earlier than x (or the interrelease time of the stream of jobs is equal to or less than x) for all valid values of x. Rather than speaking of release times of aperiodic jobs, we sometimes use the term *arrival times* (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*).

3.2.2 Execution Time

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

The actual amount of time required by a job to complete its execution may vary for many reasons. **Examples**: 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 the computation job depend on the input data. If the underlying system has performance enhancing features (e.g., cache memory and pipeline), the amount of time a computation takes to complete may vary each time it executes even when it has no conditional branches. For these reasons, the actual execution time of a computation job is unknown until it completes. Similarly, the actual amount of time to transmit each frame of a MPEG-compressed video is different from frame to frame because the numbers of bits used to encode different frames are different. The actual execution time of the job modeling the transmission of a frame is unknown a priori. What can be determined a priori through analysis and measurement are the maximum and minimum amounts of time required to complete each job. In other words, we know that the execution time e_i of the job J_i is in the range $[e_i^-, e_i^+]$, where e_i^- and e_i^+ are the *minimum execution time* and the *maximum execution time* of J_i, respectively. We usually assume that we know e_i^- and e_i^+ of every hard real-time job J_i but the actual execution time of the job is unknown.

For the purpose of determining whether each job can always complete by its deadline, knowing the maximum execution time of each job often suffices. For this reason, 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. We will also use this term in this sense most of the time. However, except where we state otherwise, we never mean that the actual execution time of the job is fixed and known, only that it never exceeds e_i.

What about the accuracy of deterministic models, which assume that every job takes its maximum execution time to complete? If we design our system based on above assumption and allocate this much time to each job, the processor(s) will surely be underutilized.

This statement is clearly true sometimes. We will encounter applications where the variations in job execution times are so large that working with their maximum values indeed yields unacceptably conservative designs. We should not model such applications deterministically. More importantly, in some systems the response times of some jobs may be larger when the actual execution times of some jobs are smaller than their maximum values. In these cases, we will have to deal with the variations in execution times explicitly.

Reasons to Deterministic Approach

However, there are two good reasons for the common use of the deterministic approach.

a. Many hard real-time systems are safety-critical. These systems are typically designed and implemented in a such a way that the variations in job execution times are kept as small as possible. The need to have relatively deterministic execution times places many restrictions on implementation choices. (For example, the programs cannot use dynamic structures that can lead to variable execution time and memory usage; performance-enhancing features are not used.) By working with these restrictions and making the execution times of jobs almost

deterministic, the designer can model more accurately the application system deterministically. In return, the deterministic approach makes the validation of the resultant system easier.

b. The other reason for the common use of the deterministic approach is that the hard real-time portion of the system is often small. The timing requirements of the rest of the system are soft. In this case, an option is to design the hard real-time subsystem based on its worst-case processor time and resource requirements even though their actual requirements may be much smaller. We can then use the methods and tools supported by the deterministic models to ensure that the hard timing constraints will surely be met at all times. We also can safely reclaim the time and resources allocated to but not used by hard real-time jobs and make the reclaimed time and resources available to soft real-time jobs and nonreal-time jobs. In this way, the system will not be overdesigned with underutilized resources.

3.3 Periodic task model

The *periodic task model* is a well-known deterministic workload model. With its various extensions, the model characterizes accurately many traditional hard real-time applications, such as digital control, real-time monitoring, and constant bit-rate voice/video transmission. Many scheduling algorithms based on this model have good performance and well-understood behavior. There are now methods and tools to support the design, analysis, and validation of real-time systems that can be accurately characterized by the model.

3.3.1 Periods, Execution Times, and Phases of Periodic Tasks

In the periodic task model, each computation or data transmission that is executed repeatedly at regular or semi-regular time intervals in order to provide a function of the system on a continuing basis is modeled as a ***period task***. Specifically, each periodic task, denoted by T_i, is a sequence of jobs. The *period* p_i of the periodic task T_i is the minimum length of all time intervals between release times of consecutive jobs in T_i. Its ***execution time*** is the maximum execution time of all the jobs in it. We use e_i to denote the execution time of the periodic task T_i, as well as that of all the jobs in it. At all times, the period and execution time of every periodic task in the system are known.

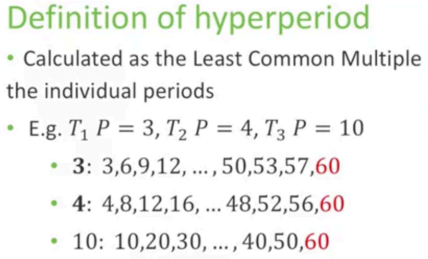
Accuracy of Periodic Task Model

The accuracy of the periodic task model decreases with increasing jitter in release times and variations in execution times. So, a periodic task is an inaccurate model of the transmission of a variable bit-rate video, because of the large variation in the execution times of jobs (i.e., transmission times of individual frames). A periodic task is also an inaccurate model of the transmission of cells on a real-time connection through a switched network that does not do traffic shaping at every switch, because large release-time jitters are possible.

Notation

We call the tasks in the system $T_1, T_2, \ldots, T_n$. When it is necessary to refer to the individual jobs in a task T_i, we call them $J_{i,1}$, $J_{i,2}$ and so on, $J_{i,k}$ being the kth job in T_i. When we want to talk about properties of individual jobs but are not interested in the tasks to which they belong, we also call the jobs J_1, J_2, and so on.

The release time $r_{i,1}$ of the first job $J_{i,1}$ in each task T_i is called the ***phase*** of T_i. For the sake of convenience, we use φ_i to denote the phase of T_i, that is, $\varphi_i = r_{i,1}$. In general, different tasks may have



different phases. Some tasks are *in phase*, meaning that they have the same phase.

We use H to denote the least common multiple of p_i for i = 1,2,...n. A time interval of length H is called a ***hyperperiod*** of the periodic tasks. The (maximum) number N of jobs in each hyperperiod is equal to $\sum_{i=1}^{n} H/p_i$. The length of a hyperperiod of three periodic i=1 tasks with periods 3, 4, and 10 is 60. The total number N of jobs in the hyperperiod is 41. We call the ratio $u_i = e_i/p_i$ the *utilization* of the task T_i. u_i is equal to the fraction of time a truly periodic task with period p_i and execution time e_i keeps a processor busy. It is an upper bound to the utilization of any task modeled by T_i. The *total utilization* U of all the tasks in the system is the sum of the utilizations of the individual tasks in it. So, if the execution times of the three periodic tasks are 1, 1, and 3, and their periods are 3, 4, and 10, respectively, then their utilizations are 0.33, 0.25 and 0.3. The total utilization of the tasks is 0.88; these tasks can keep a processor busy at most 88 percent of the time.

3.3.2 Aperiodic and Sporadic Tasks

A real-time system is invariably required to respond to external events, and to respond, it executes aperiodic or sporadic jobs whose release times are not known a priori. An operator's adjustment of the sensitivity setting of a radar surveillance system is an example. The radar system must continue to operate, but in addition, it also must respond to the operator's command. Similarly, when a pilot changes the autopilot from cruise mode to standby mode, the system must respond by reconfiguring itself, while continuing to execute the control tasks that fly the airplane. A command and control system must process sporadic data messages, in addition to the continuous voice and video traffic.

In the periodic task model, the workload generated in response to these unexpected events is captured by aperiodic and sporadic tasks. Each *aperiodic* or *sporadic task* is a stream of aperiodic or sporadic jobs, respectively. The inter-arrival times between consecutive jobs in such a task may vary widely and, in particular, can be arbitrarily small. The jobs in each task model the work done by the system in response to events of the same type. For example, the jobs that execute to change the detection threshold of the radar system are in one task; the jobs that change the operation mode of the autopilot are in one task; and the jobs that process sporadic data messages are in one task, and so on.

Specifically, the jobs in each aperiodic task are similar in the sense that they have the same statistical behavior and the same timing requirement. Their inter-arrival times are identically distributed random variables with some probability distribution $A(x)$. Similarly, the execution times of jobs in each aperiodic (or sporadic) task are identically distributed random variables, each distributed according to the probability distribution $B(x)$. These assumptions mean that the statistical behavior of the system and its environment do not change with time, that is, the system is stationary. That the system is stationary is usually valid in time intervals of length on the order of H, in particular, within any hyperperiod of the periodic tasks during which no periodic tasks are added or deleted.

We say that a task is *aperiodic* if the jobs in it have either soft deadlines or no deadlines. The task to adjust radar's sensitivity is an example. We want the system to be responsive, that is, to complete each adjustment as soon as possible. On the other hand, a late response is annoying but tolerable. We therefore want to optimize the responsiveness of the system for the aperiodic jobs, but never at the expense of hard real-time tasks whose deadlines must be met at all times.

In contrast, an autopilot system is required to respond to a pilot's command to disengage the autopilot and take over the control manually within a specific time. Similarly, when a transient fault occurs, a fault-tolerant system may be required to detect the fault and recover from it in time. The jobs that execute in response to these events have hard deadlines. Tasks containing jobs that are released at random time instants and have hard deadlines are *sporadic tasks*. We treat them as hard real-time

tasks. Our primary concern is to ensure that their deadlines are always met; minimizing their response times is of secondary importance.

3.4 Precedence constraints and data dependency

Data and control dependencies among jobs may constrain the order in which they can execute. In classical scheduling theory, the jobs are said to have *precedence constraints* if they are constrained to execute in some order. Otherwise, if the jobs can execute in any order, they are said to be *independent*.

For example, in a radar surveillance system, the signal-processing task is the producer of track records, while the tracker task is the consumer. In particular, each tracker job processes the track records generated by a signal-processing job. The designer may choose to synchronize the tasks so that the execution of the *k*th tracker job does not begin until the *k*th signal-processing job completes. The tracker job is precedence constrained.

As another example, we consider queries to an information server. Suppose that before each query is processed and the requested information retrieved, its authorization to access the requested information is first checked. The retrieval job cannot begin execution until the authentication job completes. The communication job that forwards the information to the requester cannot begin until the retrieval job completes. Similarly, in a communication system, the jobs that generate an acknowledgement of a message and transmit the acknowledgement message cannot begin until the job that receives and processes the message completes.

3.4.1 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 a *successor* of J_i) if J_k cannot begin execution until the execution of J_i completes. A short- hand notation to state this fact is $J_i < J_k$. J_i is an *immediate predecessor* of J_k (and J_k 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.

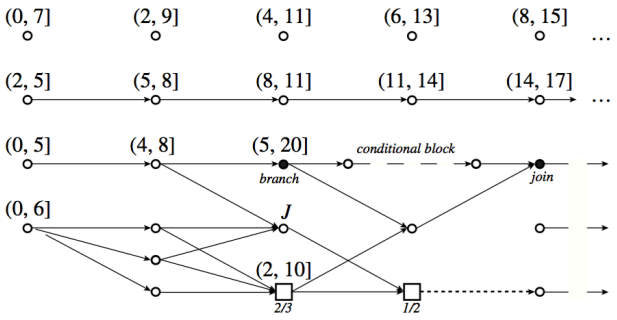


FIGURE 3–1 Example of task graphs.

A classical way to represent the precedence constraints among jobs in a set **J** is by a directed graph $G = (\mathbf{J}, <)$. Each vertex in this graph represents a job in **J**. We will call each vertex by the name of the job represented by it. There is a directed edge from the vertex J_i to the vertex J_k when the job J_i is an immediate predecessor of the job J_k. This graph is called a *precedence graph*.

A *task graph*, which gives us a general way to describe the application system, is an extended precedence graph. Figure 3–1 shows a task graph. As in a precedence graph, the vertices in a task graph represent jobs. The numbers in the bracket above each job give 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 .

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 may contain different types of edges that represent different types of dependencies. The type(s) of dependency represented

by an edge is given by the type(s) of the edge. The types of an edge connecting two vertices and other parameters of the edge are *interconnection parameters* of the jobs represented by the vertices.

3.4.2 Data Dependency

As an example, data dependency cannot be captured by a precedence graph. In many real-time systems, jobs communicate via shared data. Oftentimes, the designer chooses not to synchronize producer and consumer jobs. Rather, each producer places the data generated by it in a shared address space to be used by the consumer at any time. In this case, the classical precedence graph should show that the producer and consumer are independent because they are not explicitly constrained to execute in turn.

As an example, in an avionics system, the navigation job updates the location of the air- plane periodically. These data are placed in a shared space. Whenever the flight management job needs navigation data, it reads the most current data produced by the navigation job. There is no precedence constraint between the navigation job and the flight management job.

In a task graph, data dependencies among jobs are represented explicitly by data-dependency edges among jobs. There is a data-dependency edge from a 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.

3.5 Other dependencies

Like non-real-time applications, real-time applications sometimes contain redundant modules, carry out heuristic searches, use multiple versions, execute some job conditionally, and so forth. We add other extensions to the classical precedence graphs in order to model such jobs and dependencies. These extensions include temporal distance, OR jobs, conditional branches, and pipe (or pipeline).

3.5.1 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.

As an example, we consider the display of video frames and the accompanying audio when the video is that of a person speaking. To have lip synchronization, the time between the display of each frame and the generation of the corresponding audio segment must be no more than 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 J_i to a vertex J_k if the job J_k must be completed within a certain time after J_i completes. The temporal distance between the jobs is given by the temporal distance parameter of the edge. The value of this parameter is infinite if the jobs have no temporal distance constraint, or equivalently, there is no temporal-dependency edge between the jobs.

3.5.2 AND/OR Precedence Constraints

In the classical model, a job with more than one immediate predecessor must wait until all its immediate predecessors have been completed before its execution can begin. Whenever it is necessary to be specific, we call such jobs *AND jobs* and dependencies among them *AND precedence constraints*.

AND jobs are represented by unfilled circles in the task graph in Figure 3–1. An example is the job labeled *J* in this figure. All three of its immediate predecessors must be completed before *J* can begin execution. An *AND* job such as *J* may be the transmission of a message to a user. Its immediate predecessors are the jobs that set up a connection for the message, encrypt the message to safeguard privacy, and check the user's account for the requested quality of the delivery service. These predecessors may execute in any order relative to each other, but they must all be completed before the message transmission job can begin.

In contrast, an *OR job* is one which can begin execution at or after its release time provided one or some of its immediate predecessors has been completed. In Figure 3–1, we represent *OR* jobs by square vertices, as exemplified by the two square vertices at the bottom of the graph. The one labeled *2/3* can begin execution as soon as two out of its three immediate predecessors complete.

In the task graph, the *in-type* of job (i.e., the vertex representing the job) tells us whether all its immediate predecessors must complete before its execution can begin. By default, the value of this job parameter is *AND*. It can have the value *OR*, if only one of its immediate predecessors must be completed, or *k-out-of-l*, if only *k* out *l* of its immediate predecessor must be completed before its execution can begin.

3.5.3 Conditional Branches

Similarly, in the classical model, all the immediate successors of a job must be executed; an outgoing edge from every vertex expresses an *AND constraint*. This convention makes it inconvenient for us to represent conditional execution of jobs as follows:

```
For every second do the following:
   Process radar returns.
   Generate track records.
   Perform track association.
   For the target T on each of the established tracks do:
      If the target T is within distance D from self,
         Do the following:
            Analyze the trajectory of T.
            If T is on collision course with self, sound alarm.
         Enddo
         Else
            Compute the current distance of T from self.
            If the current distance is larger than previous distance,
               drop the track of T.
            Endif
      Endif
Endfor
```

This system can easily be modeled by a task graph that has edges expressing *OR constraints*. Only one of all the immediate successors of a job whose outgoing edges express OR constraints is to be executed. Such a job is called a *branch* job. In a meaningful task graph, there is a *join* job associated with each branch job. In Figure 3–1, these jobs are represented by filled circles. The sub-graph that begins from a vertex representing a branch job and ends at the vertex representing the associated join job is called a *conditional block*. Only one conditional branch in each conditional block is to be executed. The conditional block in Figure 3–1 has two conditional branches: Either the upper conditional branch, containing a chain of jobs, or the lower conditional branch, containing only one job, is to be executed.

3.5.4 Pipeline Relationship

A dependency between a pair of producer-consumer jobs that are pipelined can theoretically be represented by a precedence graph. In this graph, the vertices are the granules of the producer and the consumer. Each granule of the consumer can begin execution when the previous granule of this job and the corresponding granule of the producer job have completed.

In the task graph, we represent a pipeline relationship between jobs by a pipeline edge, as exemplified by the dotted edge between the jobs in the right-bottom corner of the graph in Figure 3–1. There is an edge from J_i to J_k if the output of J_i is piped into J_k and the execution of J_k can proceed as long as there are data for it to process.

3.6 Functional parameters

While scheduling and resource access-control decisions are made disregarding most functional characteristics of jobs, several functional properties do affect these decisions. The work- load model must explicitly describe these relevant properties, and this is done by the values of functional parameters. Among them are pre-emptivity, criticality, optional interval, and laxity type.

3.6.1 Preemptivity of Jobs

Executions of jobs can often be interleaved. The scheduler may suspend the execution of a less urgent job and give the processor to a more urgent job. Later when the more urgent job completes, the scheduler returns the processor to the less urgent job so the job can resume 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, later on, can be resumed from the point of suspension. Computation jobs that execute on CPUs are examples of preemptable jobs. In nonreal-time systems, such jobs are typically scheduled in a round-robin manner; this is possible because they are preemptable.

A job is *nonpreemptable* if it must be executed from start to completion without inter- ruption. This constraint may be imposed because its execution, if suspended and the processor given to other jobs, must be executed again from the beginning. As an example, we consider jobs that model the transmissions of data frames in a token ring (or bus). If transmission of a frame is interrupted before it completes, the partially transmitted frame is discarded by the receiver. The entire frame must be retransmitted from the start. To avoid wasting bandwidth in this way, we make the execution of this job on the ring (or bus) nonpreemptable.

Sometimes, a job may be preemptable everywhere except for a small portion which is constrained to be nonpreemptable. An example is an interrupt handling job. An interrupt handling job usually begins by saving the state of the processor (i.e., storing the processor status register, the stack pointer, the program counter, and so on). This small portion of the job is nonpreemptable because suspending the execution of this portion and giving the CPU to another job may cause serious errors in the data structures shared by the jobs.

During preemption, the system must first save the state of the preempted job at the time of preemption so it can resume the job from that state. Then, the system must prepare the execution environment for the preempting job before starting the job. For example, in the case of CPU jobs, the state of the preempted job includes the contents of its program counter, processor status register, and registers containing temporary results. After saving the contents of these registers in memory and before the preempting job can start, the operating system must load the new processor status register, clear

pipelines, and so on. In operating system literature, these actions are collectively called a *context switch*. The amount of time required to accomplish a context switch is called a *context-switch time*.

3.6.2 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 the job is with respect to other jobs; the more critical the job, the larger its importance. In literature, the terms priority and weight are often used to refer to importance; the more important a job, the higher its priority or the larger its weight.

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 can meet their deadlines. For this reason, some scheduling and resource access-control algorithms try to optimize weighted performance measures such as weighted average response time (i.e., the average of response time multipled by importance) or weighted average tardiness (i.e., the average of tardiness multipled by importance) over all jobs. If the system uses one of these algorithms, the information concerning the criticality of jobs must be given to the scheduler. Assigning importance to each job is a natural way to do so.

For example, in a flight control and management system, the job that controls the flight of the aircraft is more critical than the navigation job that determines the current position relative to the chosen course. The navigation job is more critical than the job that adjusts the course and cruise speed in order to minimize fuel consumption. The cabin air flow and temperature control jobs are more critical than the job that runs the in-flight movies, and so on. In the model of this system, the designer may give these jobs different importance values. In this way, the different degrees of criticality of the jobs are explicitly specified.

3.6.3 Optional Executions

It is often possible to structure an application so that some jobs or portions of jobs are optional. If an *optional job* or an *optional portion* of a job completes late or is not executed at all, the system performance may degrade, but nevertheless function satisfactorily. In contrast, jobs and portions of jobs that are not optional are *mandatory*; they must be executed to completion. Therefore, during a transient overload when it is not possible to complete all the jobs in time, we may choose to discard optional jobs (i.e, leave them unexecuted or partially executed) so that the mandatory jobs can complete in time. In this way, the system can trade the quality of the results it produces and the services it delivers for timeliness of its results and services.

As an example, in a collision avoidance system, we may consider the job that computes the correct evasive action and informs the operator of this action optional. Normally, we want the system to help the operator by choosing the correct action. However, in the presence of a failure and when the system is operating in a degraded mode, it is not possible to complete this computation in time. The collision avoidance system may still function satisfactorily if it skips this computation as long as it generates a warning and displays the course of the object about to collide with it in time.

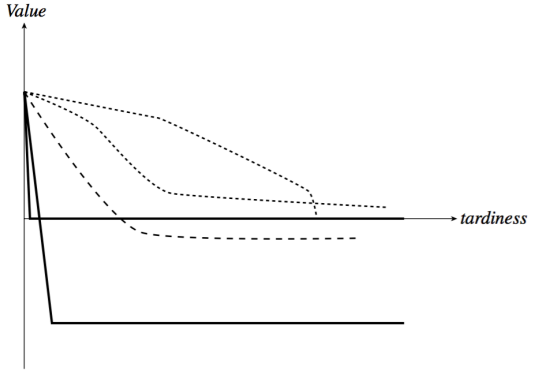


FIGURE 3–3 Examples of usefulness functions.

3.6.4 Laxity Type and Laxity Function

The laxity type of a job indicates whether its timing constraints are soft or hard. The laxity type of a job is sometimes supplemented by a *usefulness function*. This

function gives the usefulness of the result produced by the job as a function of its tardiness.

Figure 3–3 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. In the latter case, it is better not to execute the job than to execute it and complete it late. In other words, it is "better never than late." The transmission and execution of a command to release a bomb on a target is an example of jobs with this laxity type.

The dashed and dotted lines in Figure 3–3 show two other usefulness functions. In particular, the dotted ones may be that of a point-of-sales transaction, for example, one that exe- cutes 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. Eventually, you are likely to give up and walk away. At that point, the usefulness of the result becomes zero. The dashed line shows a function that decreases faster and be- comes negative. An example of jobs that have this kind of usefulness function is a stock price update transaction. 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.

We can use these usefulness functions to describe qualitatively the real-time performance objectives of the system. They can guide the choice and implementation of scheduling strategies. However, their utility as the means to specify timing constraints is small, especially if the timing constraints are hard. The only exception are those exemplified by the solid step functions in Figure 3–3, but such constraints can easily be specified without the usefulness functions.

3.7 Resource parameters of jobs and parameters of resources

The resource parameters of jobs give us a partial view of the processors and resources from the perspective of the applications that execute on them. We sometimes want to describe the characteristics of processors and resources independent of the application and can do so using parameters of resources. A resource parameter is *preemptivity*. A resource is nonpreemptable if each unit of the resource is constrained to be used serially. In other words, once a unit of a nonpreemptable resource is allocated to a job, other jobs needing the unit must wait until the job completes its use. Otherwise, if jobs can use every unit of a resource in an interleaved fashion, the resource is preemptable.

The lock on a data object in a database is an example of nonpreemptable resource. When a job modeling a transaction that reads and writes the data object has acquired the lock, other jobs that also require this lock at the time must wait. The lock is a nonpreemptable resource and, consequently, every transaction job is nonpreemptable in its use of the lock. This does not mean that the job is nonpreemptable on other resources or on the processor. In fact, the transaction may process the data it has already retrieved and, for this purpose, it requires the processor while it holds the lock. The transaction can be preempted on the processor by other transactions that are not waiting for the locks held by it.

3.7.1 Preemptivity of Resources

The resource parameters of jobs give us a partial view of the processors and resources from the perspective of the applications that execute on them. We sometimes want to describe the characteristics of processors and resources independent of the application and can do so using parameters of resources. A resource parameter is *preemptivity*. A resource is nonpreemptable if each unit of the resource is constrained to be used serially.

In other words, once a unit of a nonpreemptable resource is allocated to a job, other jobs needing the unit must wait until the job completes its use. Otherwise, if jobs can use every unit of a resource in an interleaved fashion, the resource is preemptable. The lock on a data object in a database is an example of nonpreemptable resource. When a job modeling a transaction that reads and writes the data object has acquired the lock, other jobs that also require this lock at the time must wait. The lock is a nonpreemptable resource and, consequently, every transaction job is nonpreemptable in its use of the lock. This does not mean that the job is nonpreemptable on other resources or on the processor. In fact, the transaction may process the data it has already retrieved and, for this purpose, it requires the processor while it holds the lock. The transaction can be preempted on the processor by other transactions that are not waiting for the locks held by it.

3.8 Scheduling hierarchy

Figure 3–4 shows the three elements of our model of real-time systems together. The application system is represented by a task graph, exemplified by the graph on the top of the diagram. This graph gives the processor time and resource requirements of jobs, the timing constraints of each job, and the dependencies of jobs. The resource graph describes the amounts of the resources available to execute the application system, the attributes of the resources, and the rules governing their usage. Between them are the scheduling and resource access-control algorithms used by the operating system.

3.8.1 Scheduler and Schedules

Jobs are scheduled and allocated resources according to a chosen set of scheduling algorithms and resource access-control protocols. The module which implements these algorithms is called the *scheduler*.

Specifically, the scheduler assigns processors to jobs, or equivalently, assigns jobs to processors. We say that a job is scheduled in a time interval on a processor if the processor is assigned to the job, and hence the job executes on the processor, in the interval. The total amount of (processor) time assigned to a job according to a schedule is the total length of all the time intervals during which the job is scheduled on some processor.

By a *schedule*, we mean an assignment of all the jobs in the system on the available processors produced by the scheduler. We do not question the correctness of the scheduler; rather, we assume that the scheduler works correctly. By correctness, we mean that the scheduler produces only *valid schedules*; a valid schedule satisfies the following conditions:

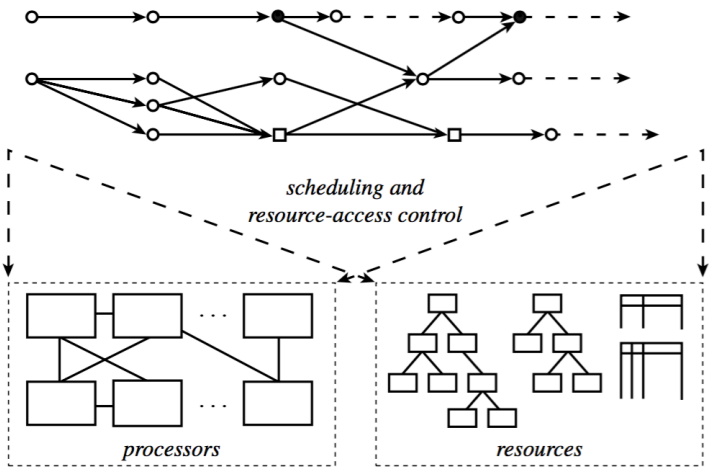


FIGURE 3–4 Model of real-time systems.

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 algorithm(s) 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.

3.8.2 Feasibility, Optimality, and Performance Measures

The criterion we use most of the time to measure the performance of scheduling algorithms for hard real-time applications is their ability to find feasible schedules of the given application system whenever such schedules exist. Hence, we say that 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. Conversely, if an optimal algorithm fails to find a feasible schedule, we can conclude that the given set of jobs cannot feasibly be scheduled by any algorithm.

In addition to the criterion based on feasibility, other commonly used performance measures include the maximum and average tardiness, lateness, and response time and the miss, loss, and invalid rates. The right choice of performance measure depends on the objective of scheduling. As an example, when a set of jobs is not schedulable by any algorithm, we may settle for a schedule according to which the number of jobs failing to complete in time is the smallest. Hence, an algorithm performs better if it can produce a schedule with a smaller number of late jobs than others. Alternatively, we may not care how many jobs are late, pro- vided that their tardiness is small. In this case, we want to use algorithms that give us small maximum or average tardiness.

The *lateness* of a job is the difference between its completion time and its deadline. Unlike the tardiness of a job which never has negative values, the lateness of a job which completes early is negative, while the lateness of a job which completes late is positive. Some- times, we want to keep jitters in the completion times small; we can do so by using scheduling algorithms that try to minimize the average absolute lateness of jobs. As an example, suppose that the jobs model the transmission of packets in a packet-switched network. The packets in each message are buffered at the switch at the receiving end, assembled into a message, and then forwarded to the receiving host. The early arrivals must be held in the buffer waiting for the late arrivals. A larger arrival-time jitter means a larger buffer occupancy time and consequently, a larger total demand for buffer space of all messages arriving at the switch. In this example, the average lateness of jobs is a meaningful performance measure, because minimizing it indirectly minimizes the average buffer occupancy time and buffer space requirement.

In the case where all the jobs have the same release time and deadline, the problem of scheduling the jobs to meet their deadline is in essence the same as that of scheduling to minimize the completion time of the job which completes last among all jobs. The response time of this job is the response time of the set of jobs as a whole and is often called the *makespan* of the schedule. This is a performance criterion commonly used to compare scheduling algorithms in classical scheduling literature: an algorithm that produces a schedule with a shorter makespan is better.

By far, the most frequently used performance measure for jobs that have soft deadlines is their average response times. The smaller the average response time, the better the algorithm, just as for general-purpose, interactive systems. In a system that has a mixture of jobs with hard and soft deadlines, the objective of scheduling is typically to minimize the average response time of jobs with soft deadlines while ensuring that all jobs with hard deadlines complete in time. Since there is no advantage in completing jobs with hard deadlines early, we may delay their execution in order to improve the response time of jobs with soft deadlines.

For many soft real-time applications, it is acceptable to complete some jobs late or to discard late jobs. For such an application, suitable performance measures include the *miss rate* and *loss rate*. The former gives the percentage of jobs that are executed but completed too late, and the latter give the percentage of jobs that are discarded, that is, not executed at all. When it is impossible to complete all the jobs on time, a scheduler may choose to discard some jobs. By doing so, the scheduler increases the loss rate but completes more jobs in time. Thus, it reduces the miss rate. Similarly, reducing the

loss rate may lead to an increase in miss rate. For this reason when we talk about minimization of the miss rate, we mean that the miss rate is reduced as much as possible subjected to the constraint that the loss rate is below some acceptable threshold. Alternatively, we may want to minimize the lost rate provided the miss rate is below some threshold. A performance measure that captures this trade-off is the *invalid rate*, which is the sum of the miss and loss rates and gives the percentage of all jobs that do not produce a useful result. We want to minimize the invalid rate. If the jobs are transmissions of real-time packets, the miss rate gives the percentage of packets arriving at the receiver too late, the loss rate gives the percentage of packets discarded en route, and the invalid rate gives the percentage of packets that are not useful to the application.

3.8.3 Interaction among Schedulers

A system typically has a hierarchy of schedulers. This scheduling hierarchy arises for two reasons.

- First, some processors and resources used by the application system are not physical entities; they are logical resources. Logical resources must be scheduled on physical resources. The algorithms used for this purpose are typically different from the ones used to schedule the application system. A scheduler that schedules a logical resource may be different from the scheduler that schedules the application system using the resource.
- Second, a job may model a server that executes on behalf of its client jobs. The time and resources allocated to the server job must in turn be allocated to its client jobs. Again, the algorithm used by the server to schedule its clients may be different from the algorithm used by the operating system to schedule the server with other servers.

Earlier, we treated database locks as resources. In fact, these resources are implemented by a database management system whose execution must be scheduled on one or more processors. The scheduler that schedules the database management system may be different from the scheduler that schedules the application system using the locks. The schedulers most likely use different algorithms. Now we have two levels of scheduling. In the higher level, the application system is scheduled on the resources. In the lower level, the jobs that execute in order to implement the resources are scheduled on the processors and resources needed by them.

As an example of servers, we consider an application system containing periodic tasks and aperiodic jobs on one processor. All the aperiodic jobs are placed in a queue when they are released. There is a poller. Together with the periodic tasks, the poller is scheduled to execute periodically. When the poller executes, it checks the aperiodic job queue. If there are aperiodic jobs waiting, it chooses an aperiodic job from the queue and executes the job. Hence, the aperiodic jobs are the clients of the poller. We again have two levels of scheduling. In the lower level the scheduler provided by the operating system schedules the poller and the periodic tasks. In the higher level, the poller schedules its clients.

In every level of the scheduling hierarchy, we can represent the workload by a task graph and the processors and resources required by it by a resource graph. In this way all levels of the scheduling hierarchy can be represented in a uniform way.

3.9 Summary

3.9.1 Characterization of Application Systems

According to this model, the basic components of any real-time application system are jobs. The operating system treats each job as a unit of work and allocates processor and resources to it. For the purpose of scheduling and validation, it suffices to define each job by its temporal, resource, interconnection and functional parameters.

- *Release time* (or *arrival time*) r_i: r_i is the instant of time at which the job becomes eligible for execution. The release (arrival) time of the job may be jittery (sporadic), meaning that r_i is in the range $[r_i^-, r_i^+]$ and that only the range of r_i is known but not the actual value of r_i.
- *Absolute deadline* d_i: d_i is the instant of time by which the job must complete.
- *Relative deadline* D_i: D_i is the maximum allowable response time of the job.
- *Laxity type*: The deadline (or timing constraint in general) of the job can be hard or soft.
- *Execution time* e_i: e_i is the amount of time required to complete the execution of J_i when it executes alone and has all the resources it requires. The execution time of J_i may vary, meaning that e_i is in the range $[e_i^-, e_i^+]$ and that this range is known but not the actual value of e_i. Some models assume that J_i always executes for the maximum amount e_i^+ of time; when there is no possible ambiguity, we also use e_i to mean the maximum execution time e_i^+.
- *Preemptivity*: The job may be constrained to be nonpreemptable.
- *Resource requirements*: This parameter gives the processors and resources required by the job in order to execute and the time interval(s) during which each resource is required.

A set of related jobs is called a task. Jobs in a task may be precedence constrained to execute in a certain order. Sometimes jobs may be constrained to complete within a certain time from one another (i.e., they are temporal distance constrained). Jobs may have data dependencies even when they are not precedence constrained.

A periodic task T_i is a sequence of jobs with identical parameters. In addition to the parameters of its jobs, a periodic task is defined by the following task parameters:

- *Period* p_i : p_i is the minimum length of the intervals between the release times of consecutive jobs (i.e., interrelease interval).
- *Execution time* e_i : e_i of T_i is the maximum execution time of all the jobs in the task.
- *Phase* φ_i: φ_i is the release time of the first job in T_i.

When the interrelease intervals of a sequence of jobs vary widely and are arbitrarily small, we model the interrelease intervals as random variables. Similarly, the execution times of jobs are sometimes modeled as random variables. An aperiodic task or a sporadic task is a sequence of jobs whose interrelease times are identically distributed according to the probability distribution $A(x)$ and whose execution times are identically distributed according to the probability distribution $B(x)$. The deadlines of jobs in an aperiodic task are soft, while the deadlines of jobs in a sporadic task are hard.

3.9.2 Characterization of the Underlying Systems

The resources available to the application system are processors and resources. Jobs and tasks require them in order to execute. The scheduler decides when each job (or task) has its required processor and resources.

We usually model CPUs, disks, buses, network links, and so on, that compute, retrieve, transmit, and so on, as processors. Every job must have a processor in order to execute. Pro- cessors that can be

used interchangeably are of the same type, while processors that cannot be used interchangeably are of different types. In addition to its type, a processor is characterized by the following parameters:

- *Preemptivity*: The execution of jobs on the processor may or may not be preemptable.
- *Context-Switch Time*: This parameter gives the cost in time of each preemption.

We model shared data objects, buffers, sequence numbers, and the like, entities some- times called passive resources, as resources. The amount of time a job requires a resource usually depends on the speed of the processor on which the job executes, not on some parameter of the resource. The resources considered here are reusable, because they are not consumed during use. Each unit of each resource (type) can be used by one job at the time, and the use of every resource (unit) is nonpreemptable. A shared resource that can be used simultaneously by at most x jobs at the same time is modeled as a resource with x units, each of which can be used by at most one job at a time.

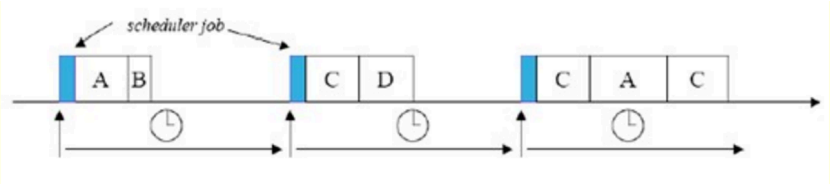
4 Approaches to Real-Time Scheduling

This chapter provides a brief overview of three commonly used approaches to scheduling real-time systems:

- clock-driven,
- weighted round-robin and
- priority-driven.

4.1 Clock-driven approach

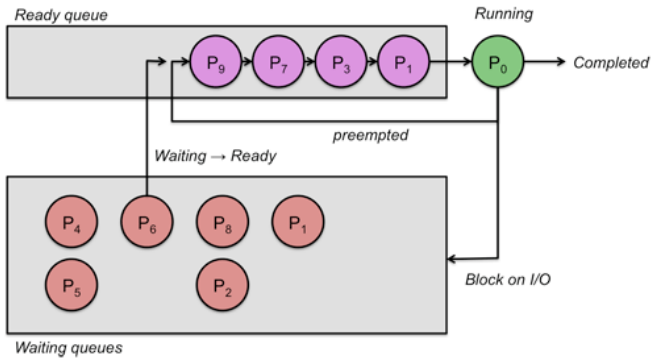
When scheduling is *clock-driven* (also called *time-driven*), decisions on what jobs execute at what times are made at specific time instants. These instants are chosen a priori before the system begins execution. Typically, in a system that uses clock-driven scheduling, all the parameters of hard real-time jobs are fixed and known. A schedule of the jobs is computed off-line and is stored for use at run time. The scheduler schedules the jobs according to this schedule at each scheduling decision time. In this way, scheduling overhead during run-time can be minimized.



A frequently adopted choice is to make scheduling decisions at regularly spaced time instants. One way to implement a scheduler that makes scheduling decisions periodically is to use a hardware timer. The timer is set to expire periodically without the intervention of the scheduler. When the system is initialized, the scheduler selects and schedules the job(s) 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 awakes and repeats these actions.

4.2 Weighted round-robin approach

The round-robin approach is commonly used for scheduling time-shared applications. When jobs are scheduled on a round-robin basis, every job joins a First-in-first-out (FIFO) queue when it becomes ready for execution. The job at the head of the queue executes for at most one time slice. (A time slice is the basic granule of time that is allocated to jobs. In a time-shared environment, a time slice is typically in the order of tens of milliseconds.) 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 every n time slices, that is, every *round*. Because the length of the time slice is relatively short, the execution of every job 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 called the processor-sharing algorithm.



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

By giving each job a fraction of the processor, a round-robin scheduler delays the completion of every job. If it is used to schedule precedence constrained jobs, the response time of a chain of jobs can be unduly large. For this reason, the weighted round-robin approach is not suitable for scheduling such jobs. On the other hand, a successor job may be able to incrementally consume what is produced by a predecessor. In this case, weighted round-robin scheduling is a reasonable approach, since a job and its successors can execute concurrently in a pipelined fashion.

As an example, we consider the two sets of jobs, $\mathbf{J}_1 = \{J_{1,1}, J_{1,2}\}$ and $\mathbf{J}_2 = \{J_{2,1}, J_{2,2}\}$, shown in Figure 4–1. The release times of all jobs are 0, and their execution times are 1. $J_{1,1}$ and $J_{2,1}$ execute on processor P_1, and $J_{1,2}$ and $J_{2,2}$ execute on processor P_2. Suppose that $J_{1,1}$ is the predecessor of $J_{1,2}$, and $J_{2,1}$ is the predecessor of $J_{2,2}$. Figure 4–1(a) shows that both sets of jobs (i.e., the second jobs $J_{1,2}$ and $J_{2,2}$ in the sets) complete approximately at time 4 if the jobs are scheduled in a weighted round-robin manner. (We get this completion time when the length of the time slice is small compared with 1 and the jobs have the same weight.) In contrast, the schedule in Figure 4–1(b) shows that if the jobs on each processor are executed one after the other, one of the chains can complete at time 2, while the other can complete at time 3. On the other hand, suppose that the result of the first job in each set is piped to the second job in the set. The latter can execute after each one or a few time slices of the former complete. Then it is better to schedule the jobs on the round-robin basis because both sets can complete a few time slices after time 2.

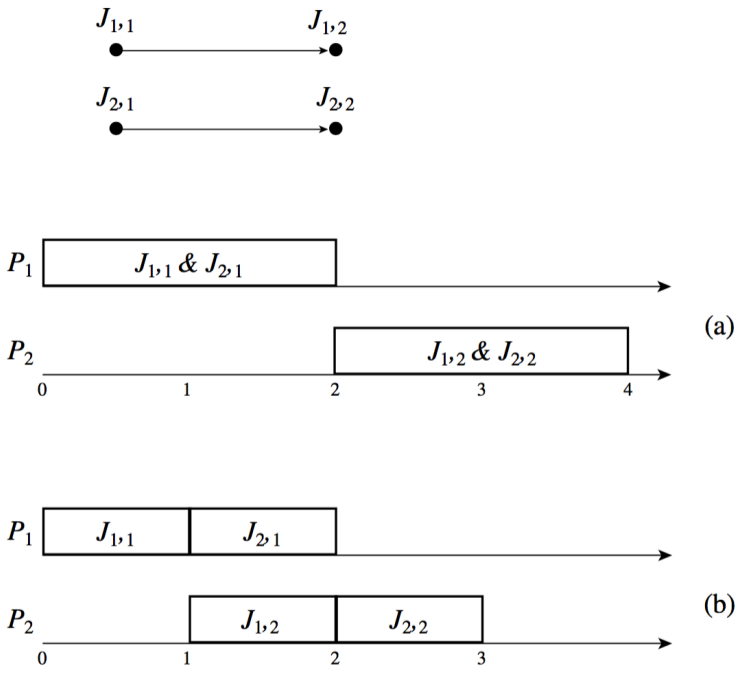


FIGURE 4–1 Example illustrating round-robin scheduling of precedence-constrained jobs.

Indeed, the transmission of each message is carried out by switches en route in a pipeline fashion. A switch downstream can begin to transmit an earlier portion of the message as soon as it receives the portion without having to wait for the arrival of the later portion of the message. 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 priority queues with the required speed are expensive.

4.3 Priority-driven Approach

The term *priority-driven* algorithms refers to a large class of scheduling algorithms that never leave any resource idle intentionally. Stated in another way, a resource idles only when no job requiring the

resource is ready for execution. Scheduling decisions are made when events such as releases and completions of jobs occur. Hence, priority-driven algorithms are *event-driven*.

Other commonly used names 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. So when a processor or resource is available and some job can use it to make progress, such an algorithm never makes the job wait. However, sometimes 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 descriptive 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 to a great extent 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 scheduling algorithms used in nonreal-time systems are priority-driven. Examples include the FIFO (First-In-First-Out) and LIFO (Last-In-First-Out) algorithms, which assign priorities to jobs according their release times, and the SETF (Shortest-Execution-Time-First) and LETF (Longest-Execution-Time-First) algorithms, which assign priorities on the basis of 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.

Figure 4–2 gives an example. The task graph shown here is a classical precedence graph; all its edges represent 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. They communicate via a shared memory. Hence the costs of communication among jobs are negligible no matter where they are executed. The schedulers of the processors keep one common priority queue of ready jobs. The priority list is given next to the graph: J_i has a higher priority than J_k if $i < k$. All the jobs are preemptable; scheduling decisions are made whenever some job becomes ready for execution or some job completes.

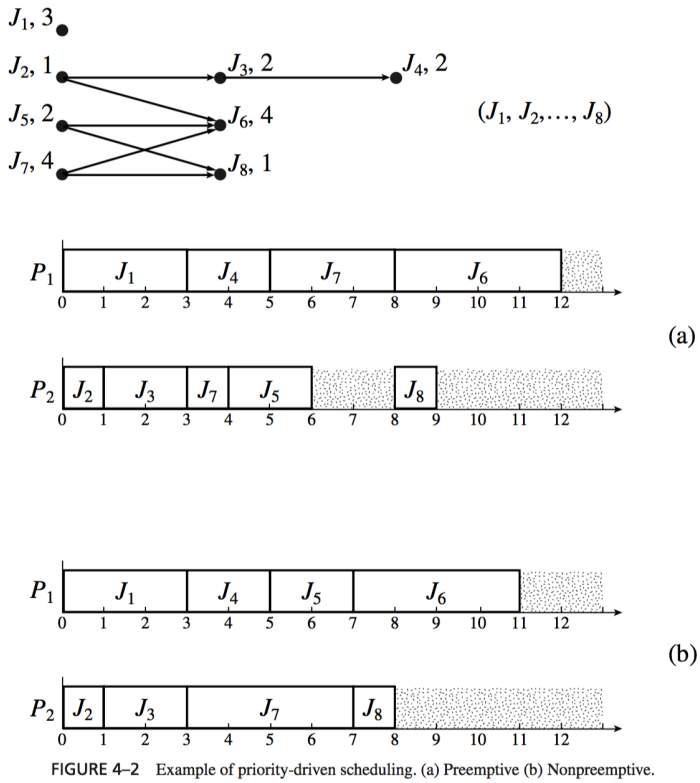


FIGURE 4–2 Example of priority-driven scheduling. (a) Preemptive (b) Nonpreemptive.

Figure 4–2(a) shows the schedule of the 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 common 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. The processors continue to execute the jobs scheduled on them except when the following events occur and new scheduling decisions are made.

- 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. Consequently, it is preempted. 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 ready for execution. Consequently, processor P_2 becomes idle.
- J_7 finally completes at time 8. J_6 and J_8 can now be scheduled and they are.

Figure 4–2(b) shows a nonpreemptive schedule according to the same priority assignment. Before time 4, this schedule is the same as the preemptive schedule. However, at time 4 when J_5 is released, both processors are busy. It has to wait until J_4 completes (at time 5) before it can begin execution. It turns out that for this system this postponement of the higher priority job benefits the set of jobs as a whole. The entire set completes 1 unit of time earlier according to the nonpreemptive schedule.

In general, however, nonpreemptive scheduling is not better than preemptive scheduling.

A fundamental question is, when is preemptive scheduling better than nonpreemptive scheduling and vice versa?

It would be good if we had some rule with which we could determine from the given parameters of the jobs whether to schedule them preemptively or nonpreemptively. Unfortunately, there is no known answer to this question in general. In the special case when jobs have the same release time, preemptive scheduling is better when the cost of preemption is ignored. Specifically, in a multiprocessor system, the minimum makespan (i.e., the response time of the job that completes last among all jobs) achievable by an optimal preemptive algorithm is shorter than the makespan achievable by an optimal nonpreemptive algorithm. A natural question here is whether the difference in the minimum makespans achievable by the two classes of algorithms is significant, in particular, whether the theoretical gain in makespan achievable by preemption is enough to compensate for the context switch overhead of preemption.

4.4 Dynamic versus static system

In the above example, jobs that are ready for execution are placed in a priority queue common to all processors. When a processor is available, the job at the head of the queue executes on the processor. We will refer to such a multiprocessor system as a *dynamic system*, because jobs are *dynamically dispatched* to processors. In the example in Figure 4–2, we allowed each preempted job to resume on any processor and hence, jobs are migratable. We say that a job *migrates* if it starts execution on a processor, is preempted, and later resumes on a different processor.

Another approach to scheduling in multiprocessor and distributed systems is to partition the jobs in the system into subsystems and assign and bind the subsystems statically to the processors. Jobs are moved among processors only when the system must be reconfigured, that is, when the operation mode of the system changes or some processor fails. Such a sys- tem is called a *static system*, 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. Except for the constraints thus imposed, the jobs on each processor are scheduled by themselves.

As an example, a partition and assignment of the jobs in Figure 4–2 put J_1, J_2, J_3, and 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) and (J_5, J_6, J_7, J_8). The scheduler of processor P_1 uses the former while the scheduler of processor P_2 uses the latter. 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. Moreover, J_2 completes by time 4 while J_6 starts at time 6. Therefore, the precedence constraint between them is satisfied.

4.5 Effective release times and deadlines

The given release times and deadlines of jobs are sometimes inconsistent with the precedence constraints of the jobs. By this, we mean that 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. Therefore, 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. The derived timing constraints are consistent with the precedence constraints.

When there is only one processor, we can compute the derived constraints according to the following rules:

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

Effective Deadline: The effective deadline of a job without 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.

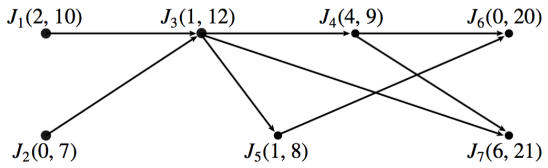


FIGURE 4–3 Example of effective timing constraints.

As an example, we look at the set of jobs in Figure 4–3. The numbers in the parentheses next to the name of each job are its given release times and deadlines. Because J_1 and J_2 have no predecessors, their effective release times are the given release times, that is, 2 and 0, respectively. The given release time of J_3 is 1, but the latest effective release time of its predecessors is 2, that of J_1. Hence, the effective release time of J_3 is 2. The effective release times of the rest of the jobs are 4, 2, 4, and 6, respectively. Similarly, J_6 and J_7 have no successors, and their effective deadlines are equal to their given deadlines, 20 and 21, respectively. Since the effective deadlines of the successors of J_4 and J_5 are larger than the given deadlines of J_4 and J_5, the effective deadlines of J_4 and J_5 are equal to their given deadlines. On the other hand, the given deadline of J_3 is equal to 12, which is larger than the minimum value of 8 among the effective deadlines of its successors. Hence, the effective deadline of J_3 is 8. In a similar way, we find that the effective deadlines of J_1 and J_2 are 8 and 7, respectively.

The calculation of effective release times and deadlines does not take into account the execution times of jobs. More accurately, the effective deadline of a job should be as early as the deadline of each of its successors minus the execution time of the successor. The effective release time of a job is that of its predecessor plus the execution time of the predecessor.

It is feasible to schedule any set of jobs on a processor according to their given release times and deadlines if and only if it is feasible to schedule the set according to their effective release times and deadlines defined above. When there is only one processor and jobs are preemptable, working with the effective release times and deadlines allows us to temporarily ignore the precedence constraints and treat all the jobs as if they are independent.

4.6 Optimality of the EDF and LST algorithms

A way to assign priorities to jobs is on the basis of their deadlines. In particular, the earlier the deadline, the higher the priority. The priority-driven scheduling algorithm based on this priority assignment is called the *Earliest-Deadline-First* (*EDF*) algorithm. This algorithm is important because it is optimal when used to schedule jobs on a processor as long as preemption is allowed and jobs do not contend for resources. This fact is stated formally below.

THEOREM 4.1. When preemption is allowed and jobs do not contend for resources, the EDF algorithm can produce a feasible schedule of a set **J** of jobs with arbitrary release times and deadlines on a processor if and only if **J** has feasible schedules.

Proof. The proof is based on the following fact: Any feasible schedule of **J** can be systematically transformed into an EDF schedule (i.e., a schedule produced by the EDF algorithm). To see why, suppose that in a schedule, parts of J_i and J_k are scheduled in intervals I_1 and I_2, respectively. Furthermore, the deadline d_i of J_i is later than the deadline d_k of J_k, but I_1 is earlier than I_2 as shown in Figure 4–4.

There are two cases. In the first case, the release time of J_k may be later than the end of I_1. J_k cannot be scheduled in I_1; the two jobs are already scheduled on the EDF basis in these intervals. Hence, we need to consider only the second case where the release time r_k of J_k is before the end of I_1; without loss of generality, we assume that r_k is no later than the beginning of I_1.

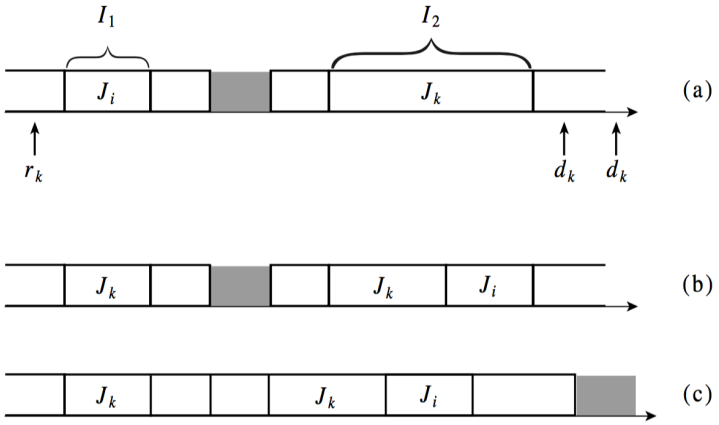


FIGURE 4–4 Transformation of a non-EDF schedule into an EDF schedule.

To transform the given schedule, we swap J_i and J_k. Specifically, if the interval I_1 is shorter than I_2, as shown in Figure 4–4, 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. The result is as shown in Figure 4–4(b). Clearly, this swap is always possible. We can do a similar swap if the interval I_1 is longer than I_2: 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 on the EDF basis. We repeat this transformation for every pair of jobs that are not scheduled on the EDF basis according to the given non-EDF schedule until no such pair exists.

The schedule obtained after this transformation may still not be an EDF schedule if some interval is left idle while there are jobs ready for execution but are scheduled in a later interval (e.g., as in the schedule in Figure 4–4(b).) 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. This is

clearly always possible. We repeat this process if necessary until the processor never idles when there are jobs ready for execution as in Figure 4–4(c).

That the preemptive EDF algorithm can always produce a feasible schedule as long as feasible schedules exist follows straightforwardly from the fact that every feasible schedule can be transformed into a preemptive EDF schedule. If the EDF algorithm fails to produce a feasible schedule, then no feasible schedule exists. (If a feasible schedule were to exist, it could be transformed into an EDF schedule, which contradicts the statement that the EDF algorithm fails to produce a feasible schedule.)

When the goal of scheduling is to meet deadlines, there is no advantage to completing any job sooner than necessary. We may want to postpone the execution of hard real-time jobs for some reason (e.g., to enable soft real-time jobs, whose response times are important, to complete earlier). For this reason, we sometimes also use the *latest release time (LRT)* algorithm (or reverse EDF algorithm). This algorithm treats release times as deadlines and deadlines as release times and schedules the jobs backwards, starting from the latest deadline of all jobs, in "priority-driven" manner, to the current time. In particular, the "priorities" are based on the release times of jobs: the later the release time, the higher the "priority." Because it may leave the processor idle when there are jobs ready for execution, the LRT algorithm is not a priority-driven algorithm.

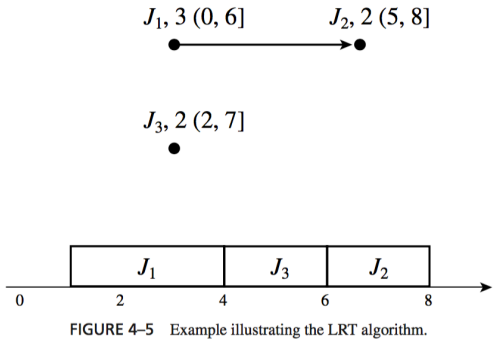


FIGURE 4–5 Example illustrating the LRT algorithm.

Figure 4–5 gives an example. In the precedence graph, the number next to the job name is the execution time of the job. Its feasible interval is given by the range of time next to its execution time. The latest deadline among all jobs is 8. Hence time starts at 8 and goes backwards to 0. At time 8, J_2 is "ready" and is scheduled. At time 7, J_3 is also "ready" to be scheduled, but because J_2 has a later release time, it has a higher priority. Consequently, J_2 is scheduled from 7 to 6. When J_2 "completes" at time 6, J_1 is "ready." However, J_3 has a higher priority and is, therefore, scheduled from 6 to 4. Finally J_1 is scheduled from 4 to 1. The result is a feasible schedule.

The following corollary states that the LRT algorithm is also optimal under the same conditions that the EDF algorithm is optimal. Its proof follows straightforwardly from the proof of Theorem 4.1.

COROLLARY 4.2. When preemption is allowed and jobs do not contend for resources, the LRT algorithm can produce a feasible schedule of a set **J** of jobs with arbitrary release times and deadlines on a processor if and only if feasible schedules of **J** exist.

Another algorithm that is optimal for scheduling preemptive jobs on one processor is the *Least-Slack-Time-First* (*LST*) algorithm (also called the *Minimum-Laxity-First* (*MLF*) algorithm). At any time t, the *slack* (or *laxity*) of a job with deadline at d is equal to $d - t$ minus the time required to complete the remaining portion of the job. Take the job J_1 in Figure 4–5 as an example. It is released at time 0, its deadline is 6, and its execution time is 3. Hence, its slack is equal to 3 at time 0. The job starts to execute at time 0. As long as it executes, its slack remains at 3, because at any time t before its completion, its slack is $6 - t - (3 - t)$. Now suppose that it 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 is1, 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.

THEOREM 4.3. When preemption is allowed and jobs do not contend for resources, the LST (MLF) algorithm can produce a feasible schedule of a set **J** of jobs with arbitrary release times and deadlines on a processor if and only if feasible schedules of **J** exist.

4.7 Nonoptimality of the EDF and LST algorithms

It is natural to ask here whether the EDF and the LST algorithms remain optimal if preemption is not allowed or there is more than one processor. Unfortunately, the answer is no.

The fact that the EDF and the LST algorithms are optimal only when preemption is allowed is illustrated by the example in Figure 4–6. The system shown in this figure has three independent, nonpreemptable jobs J_1, J_2, and J_3. Their release times are 0, 2 and 4, respectively, and are indicated by the arrows above the schedules. Their execution times are 3, 6, and 4; and their deadlines are 10, 14, and 12, respectively. Figure 4–6(a) shows the schedule produced by the EDF algorithm. In particular, when J_1 completes at time 3, J_2 has already been released but not J_3. Hence, J_2 is scheduled. When J_3 is released at time 4, J_2 is executing. Even though J_3 has an earlier deadline and, hence, a higher priority, it must wait until J_2 completes because preemption is not allowed. As a result, J_3 misses its deadline. It is easy to see that the LST algorithm would produce the same infeasible schedule. The fact that these three jobs can meet their deadlines is demonstrated by the feasible schedule in Figure 4–6(b). At time 3 when J_1 completes, the processor is left idle, even though J_2 is ready for execution. Consequently, when J_3 is released at 4, it can be scheduled ahead of J_2, allowing both jobs to meet their deadlines.

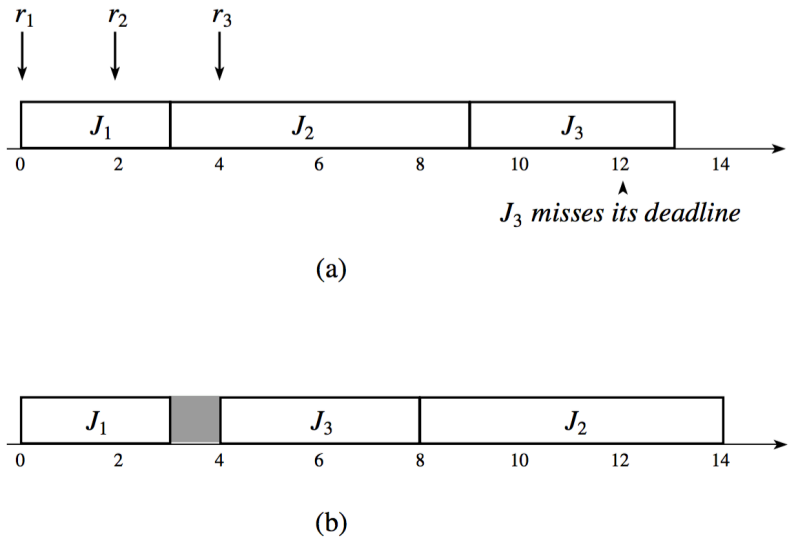


FIGURE 4–6 Example illustrating nonoptimality of the nonpreemptive EDF algorithm. (a) An EDF schedule. (b) A non-EDF schedule.

We note that the schedule in Figure 4–6(b) cannot be produced by any priority-driven scheduling algorithm. By definition, a priority-driven algorithm never leaves a processor idle when there are jobs ready to use the processor. This example illustrates the fact that not only nonpreemptive EDF and LST algorithms are not optimal, but also no nonpreemptive priority- driven algorithm is optimal when jobs have arbitrary release times, execution times, and deadlines.

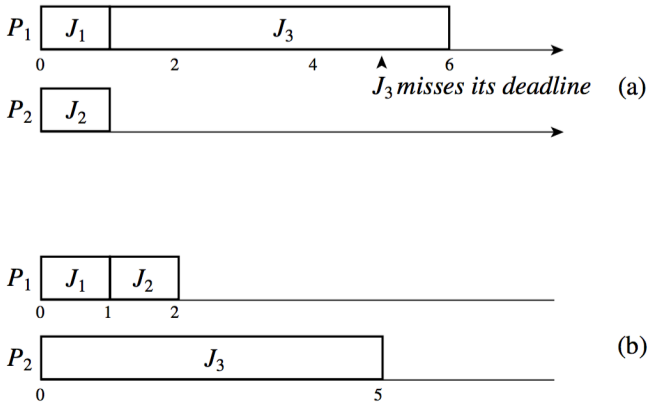


FIGURE 4–7 Example illustrating nonoptimality of the EDF algorithm for multiprocessor scheduling. (a) The EDF schedule. (b) A feasible schedule.

The example in Figure 4–7 shows that the EDF algorithm is not optimal for scheduling preemptable jobs on more than one processor. The system in this figure also contains three jobs, J_1, J_2, and J_3. Their execution times are 1, 1, and 5 and their deadlines are 1, 2, and 5, respectively. The release times of all three jobs are 0. The system has two processors. According to the EDF algorithm, J_1 and J_2 are scheduled on the processors at time 0 because they have higher priorities. The result is the schedule in Figure 4–7(a), and J_3 misses its deadline.

On the other hand, an algorithm which assigns a higher priority to J_3 in this case can feasibly schedule the jobs. An example of such algorithms is the LST algorithm. The slacks of the J_1, J_2, and J_3 in Figure 4–7 are 0, 1, and 0, respectively. Hence, this algorithm would produce the feasible schedule in Figure 4–7(b). Unfortunately, the LST algorithm is also not optimal for scheduling jobs on more than one processor.

4.8 Challenges in validating timing constraints in priority-driven systems

Compared with the clock-driven approach, the priority-driven scheduling approach has many advantages. As examples, you may have noticed that priority-driven schedulers are easy to implement. Many well-known priority-driven algorithms use very simple priority assignments, and for these algorithms, the run-time overhead due to maintaining a priority queue of ready jobs can be made very small. A clock-driven scheduler requires the information on the release times and execution times of the jobs a priori in order to decide when to schedule them. In contrast, a priority-driven scheduler does not require most of this information, making it much better suited for applications with varying time and resource requirements.

Despite its merits, the priority-driven approach has not been widely used in hard real-time systems, especially safety-critical systems, until recently. The major reason is that the timing behavior of a priority-driven system is nondeterministic when job parameters vary. Consequently, it is difficult to validate that the deadlines of all jobs scheduled in a priority-driven manner indeed meet their deadlines when the job parameters vary. In general, this *validation problem* can be stated as follows: 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.

4.8.1 Anomalous Behavior of Priority-Driven Systems

Figure 4–8 gives an example illustrating why the validation problem is difficult when the scheduling algorithm is priority-driven and job parameters may vary. The simple system contains four independent jobs. The jobs are scheduled on two identical processors in a priority-driven manner. The processors maintain a common priority queue, and the priority order is J_1, J_2, J_3, and J_4 with J_1 having the highest priority. In other words, the system is dynamic. The jobs may be preempted but never migrated, meaning that once a job begins execution on a processor, it is constrained to execute on that processor until completion. The execution times of all the jobs are fixed and known, except for J_2. Its execution time can be any value in the range [2, 6].

Suppose that 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 the earliest completion times of the job) is no more than 4. A brute force way to do so is to simulate the system. Suppose that we schedule the jobs according their given priorities, assuming first that the execution time of J_2 has the maximum value 6 and then that it has the minimum value 2. The resultant schedules are shown in Figure 4–8(a) and (b), respectively. Looking at these schedules, we might conclude that all jobs meet

their deadlines, and the completion-time jitters are sufficiently small. This would be an incorrect conclusion, as demonstrated by the schedules in Figure 4–8(c) and (d). As far as J_4 is concerned, the worst-case schedule is the one shown in Figure 4–8(c); it occurs when the execution time of J_2 is 3. According to this schedule, the completion time of J_4 is 21; the job misses its deadline. The best-case schedule for J_4 is shown in Figure 4–8(d); it occurs when the execution time of J_2 is 5. From this schedule, we see that J_4 can complete as early as time 15; its completion-time jitter exceeds the upper limit of 4. To find the worst-case and best-case schedules, we must try all the possible values of e_2.

	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

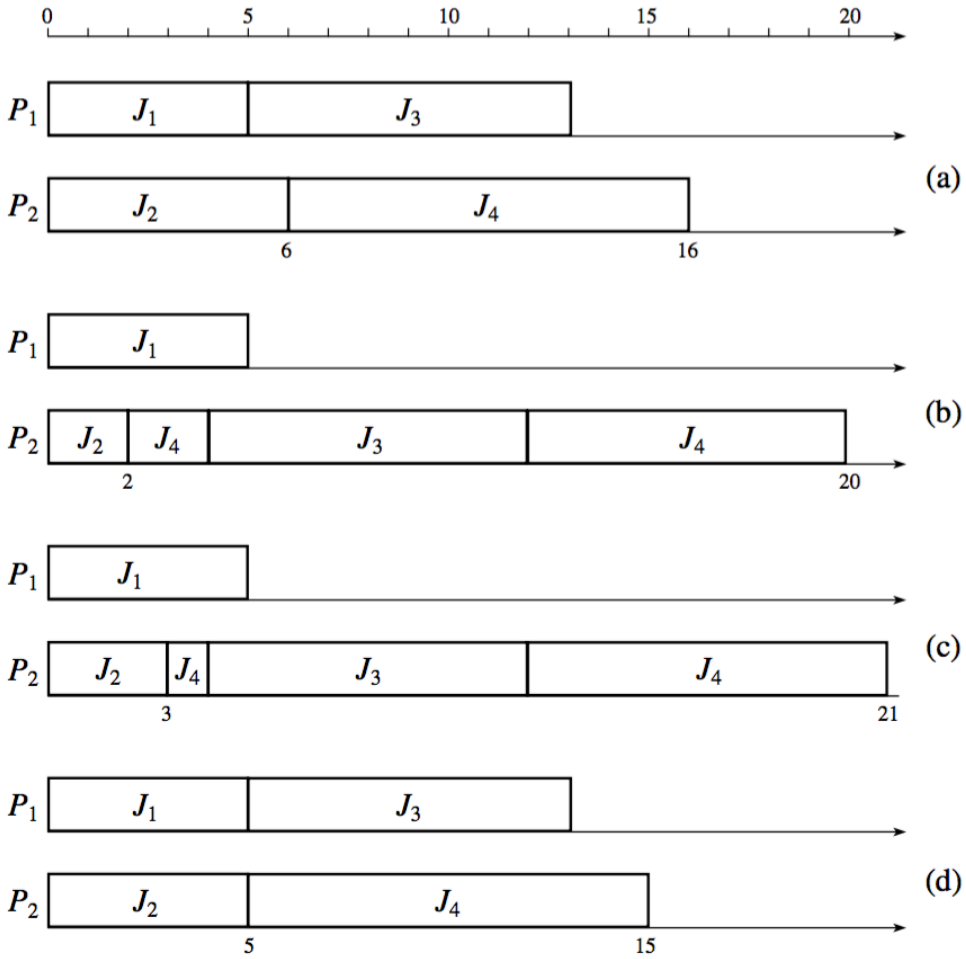


FIGURE 4–8 Example illustrating scheduling anomalies.

The phenomenon illustrated by this example is known as a *scheduling anomaly*, an unexpected timing behavior of priority-driven systems.

Scheduling anomalies make the problem of validating a priority-driven system difficult whenever job parameters may vary. Unfortunately, variations in execution times and release times are often unavoidable. If the maximum range of execution times of all n jobs in a system is X, the time required to find the latest and earliest completion times of all jobs is $O(X^n)$ if we were to find these extrema by exhaustive simulation or testing.

4.9 Off-line versus on-line scheduling

A clock-driven scheduler typically makes use of a precomputed schedule of all hard real-time jobs. 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, scheduling is (done) off-line, and the precomputed schedules are *off-line schedules*.

An obvious disadvantage of off-line scheduling is inflexibility. 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. For a deterministic system, however, off-line scheduling has several advantages, the deterministic timing behavior of the resultant system being one of them. Because the computation of the schedules is done off-line, the complexity of the scheduling algorithm(s) used for this purpose is not important.

Competitiveness of On-Line Scheduling. We say that 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.

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. The price of the flexibility and adaptability is a reduced ability for the scheduler to make the best use of system resources. Without prior knowledge about future jobs, the scheduler cannot make optimal scheduling decisions while a clairvoyant scheduler that knows about all future jobs can.

Example

As a simple example, suppose that at time 0, a nonpreemptive job J_1 with execution time 1 and deadline 2 is released. An on-line scheduler has two options at time 0: It either schedules J_1 to start execution at time 0 or it postpones the execution of J_1 to some later time. Suppose that the on-line scheduler decides to schedule J_1 at time 0. Later at time $x < 1$, a job J_2 with execution time $1 - x$ and deadline 1 is released. J_2 would miss its deadline because it cannot start execution until time 1. In contrast, a clairvoyant scheduler, which knows J_2 at time 0, would schedule J_1 to start execution at time 1 and thus allow both jobs to complete in time. In the second case, the on-line scheduler decides to postpone the execution of J_1 until some later time $x < 1$. Now suppose that at time x, J_3 is released instead of J_2. The execution time of J_3 is 1, and its deadline is 2. It is impossible for the on-line scheduler to schedule both J_1 and J_3 so that they complete in time. Again, a clairvoyant scheduler, knowing the future release of J_3 at time 0, would schedule J_1 to start execution at time 0 so it can complete both J_1 and J_3 on time.

The system is said to be *overloaded* when the jobs offered to the scheduler cannot be feasibly scheduled even by a clairvoyant scheduler. When the system is not overloaded, an optimal on-line scheduling algorithm is one that always produces a feasible schedule of all offered jobs. On the other hand, if all the jobs are preemptable and there is only one processor, optimal on-line algorithms exist, and the EDF and LST algorithms are examples.

During an overload, some jobs must be discarded in order to allow other jobs to complete in time. A reasonable way to measure the performance of a scheduling algorithm during an overload is by the amount of work the scheduler can feasibly schedule according to the algorithm: the larger this amount, the better the algorithm. The competitive factor of an algorithm captures this aspect of performance. To define this performance measure, we say that the *value of a job* is equal to its execution time if the job completes by its deadline according to a given schedule and is equal to zero if the job fails to complete in time according to the schedule. The *value of a schedule* of a sequence of jobs is equal to the sum of the values of all the jobs in the sequence according to the schedule. A scheduling algorithm is optimal if it always produces a schedule of the maximum possible value for every finite set of jobs. An on-line algorithm has a *competitive factor c* if and only if the value of the schedule of any finite sequence of jobs produced by the algorithm is at least *c* times the value of the schedule of the jobs produced by an optimal clairvoyant algorithm.

5 Clock-Driven Scheduling

5.1 Notations and Assumptions

The clock-driven approach to scheduling is applicable only when the system is deterministic, except for a few aperiodic and sporadic jobs to be accommodated in the deterministic framework. For this reason, we assume a restricted periodic task model. The following are the restrictive assumptions:

1. There are *n* periodic tasks in the system. As long as the system stays in an operation mode, *n* is fixed.
2. The parameters of all periodic tasks are known a priori. In particular, variations in the inter-release times of jobs in any periodic task are negligibly small. In other words, for all practical purposes, 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 $r_{i,k}$.

We refer to a periodic task T_i with phase φ_i, period p_i, execution time e_i, and relative deadline D_i by the 4-tuple $(\varphi_i, p_i, e_i, D_i)$. For example, (1,10,3,6) is a periodic task with

- phase is 1,
- period is 10,
- execution time is 3, and
- relative deadline is 6.

Therefore the first job in this task is released and ready at time 1 and must be completed by time 7; the second job is ready at 11 and must be completed by 17, and so on. Each of these jobs executes for at most 3 units of time. The utilization of this task is 0.3. By default, the phase of each task is 0, and its relative deadline is equal to its period. We will omit the elements of the tuple that have their default values. As examples, both (10, 3, 6) and (10, 3) have zero phase. Their relative deadlines are 6 and 10, respectively.

5.2 Static, Timer-driver scheduler

How do we handle Aperiodic Jobs?

For now we assume that there are no sporadic jobs.

We assume that the operating system maintains a queue for aperiodic jobs. When an aperiodic job is released, it is placed in the queue without the attention of the scheduler. We are not concerned with how the aperiodic jobs are ordered in this queue but simply assume that they are ordered in a manner suitable for the applications in the system. Whenever the processor is available for aperiodic jobs, the job at the head of this queue executes.

Job with Hard Deadlines

Whenever the parameters of jobs with hard deadlines are known before the system begins to execute, a straightforward way to ensure that they meet their deadlines is to construct a *static schedule* of the jobs off-line. This schedule specifies exactly when each job executes. According to the schedule, the amount of processor time allocated to every job is equal to its maximum execution time, and every job completes by its deadline. During run time, the scheduler dispatches the jobs according to this schedule. Hence, as long as no job ever *over-runs* (i.e., some rare or erroneous condition causes it to execute longer than its maximum execution time), all deadlines are surely met. Because the schedule is computed off-line, we can afford to use complex, sophisticated algorithms.

Example:

Consider a system that contains four independent periodic tasks. They are $T_1 = (4, 1)$, $T_2 = (5, 1.8)$, $T_3 = (20, 1)$, and $T_4 = (20, 2)$. Their utilizations are 0.25, 0.36, 0.05, and 0.1, respectively, and the total utilization is 0.76. It suffices to construct a static schedule for the first hyperperiod of the tasks. Since the least common multiple of all periods is 20, the length of each hyperperiod is 20. The entire schedule consists of replicated segments of length 20. Figure 5–1 shows such a schedule segment on one processor. We see that T_1 starts execution at time 0, 4, 9.8, 13.8, and so on; T_2 starts execution at 2, 8, 12, 18, and so on. All tasks meet their deadlines.

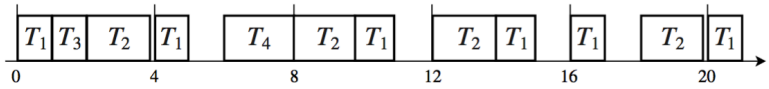


FIGURE 5–1 An arbitrary static schedule.

Some intervals, such as (3.8,4), (5,6), and (10.8,12), are not used by the periodic tasks. These intervals can be used to execute aperiodic jobs.

Use of unused intervals

It may be advantageous to have the unused intervals scattered more or less periodically in the schedule. If no aperiodic jobs are ready for execution during these intervals, we can use the time to execute background nonreal-time jobs whose response times are uncritical to the performance of the system or some built-in self-test job that checks the status and monitors the health of the system.

How is static schedule implemented?

A straightforward way to implement the scheduler is to store the precomputed schedule as a table. Each entry $(t_k, T(t_k))$ in this table gives a *decision time* t_k, which is an instant when a scheduling decision is made, and $T(t_k)$, which is either the name of the task whose job should start at t_k or I. The latter indicates an idle interval during which no periodic task is scheduled. During initialization (say at time 0), the operating system creates all the tasks that are to be executed. The scheduler makes use of a timer. Immediately after all the tasks have been created and initialized and then at every scheduling decision time, the scheduler sets the timer so the timer will expire and request an interrupt at the next decision time. Upon receiving a timer interrupt at t_k, the scheduler sets the timer to expire at t_{k+1} and prepares the task $T(t_k)$ for execution. It then suspends itself, letting the task have the processor and execute. When the timer expires again, the scheduler repeats this operation.

```
Input: Stored schedule (t_k, T(t_k)) for k = 0, 1, ... N − 1.
Task SCHEDULER:
   set the next decision point i and table entry k to 0;
   set the timer to expire at t_k.
   do forever:
      accept timer interrupt;
      if an aperiodic job is executing, preempt the job;
      current task T = T(t_k);
      increment i by 1;
      compute the next table entry k = i mod(N);
      set the timer to expire at ⌊i/N⌋H + t_k;
      if the current task T is I,
         let the job at the head of the aperiodic job queue execute;
         else, let the task T execute;
      sleep;
end SCHEDULER
```

FIGURE 5–2 A clock-driven scheduler.

The pseudocode in Figure 5–2 describes the operation of such a scheduler. H is the length of the hyperperiod of the system. N is the number of entries in the schedule of each hyperperiod. The

description assumes the existence of a timer. The timer, once set to expire at a certain time, will generate an interrupt at that time. This interrupt wakes up the scheduler, which is given the processor with a negligible amount of delay.

In the example in Figure 5–1, the stored table contains 17 entries. They are $(0,T_1)$, $(1, T_3)$, $(2, T_2)$, $(3.8, I)$, $(4, T_1)$, . . . $(19.8, I)$. Hence, the timer is set to expire at 0, 1, 2, 3.8, and so on. At these times, the scheduler schedules the execution of tasks T_1, T_3, T_2, and an aperiodic or background job, respectively. The table is used again during the next hyperperiod, and new decision times 20, 21, 22, 23.8, and so on, can be obtained from the times in the first hyperperiod as described in Fig 5.2.

Cyclic Schedule

We call a periodic static schedule a *cyclic schedule*. Again, this approach to scheduling hard real-time jobs is called the *clock-driven* or *time-driven* approach because each scheduling decision is made at a specific time, independent of events, such as job releases and completions, in the system.

5.3 General structure of cyclic schedules

Rather than using ad hoc cyclic schedules, such as the one in Figure 5–1, we may want to use a schedule that has a certain structure. By making sure that the structure has the desired characteristics, we can ensure that the cyclic schedule and the scheduler have these characteristics.

5.3.1 Frames and Major Cycles

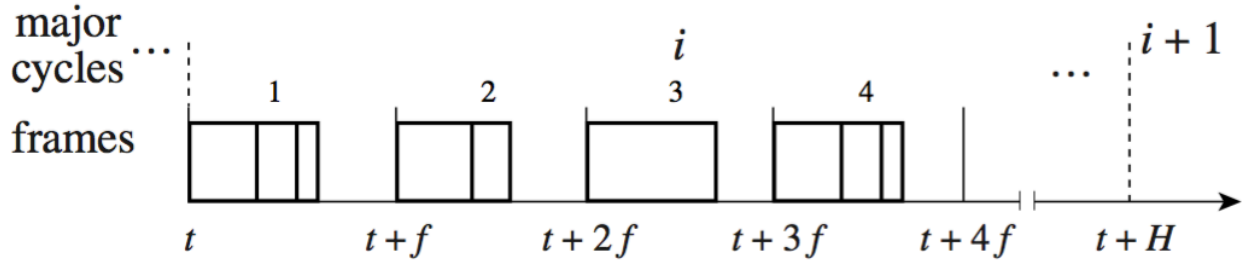


FIGURE 5–3 General structure of a cyclic schedule.

Figure 5–3 shows a good structure of cyclic schedules. A restriction imposed by this structure is that scheduling decisions are made periodically, rather than at arbitrary times. The scheduling decision times partition the time line into intervals called *frames*. Every frame has length f; f is the *frame size*. Because scheduling decisions are made only at the beginning of every frame, there is no preemption within each frame. The phase of each periodic task is a nonnegative integer multiple of the frame size. In other words, the first job of every task is released at the beginning of some frame.

In addition to choosing which job to execute, we want the scheduler to carry out monitoring and enforcement actions at the beginning of each frame. In particular, we want the scheduler to check whether every job scheduled in the frame has indeed been released and is ready for execution. We also want the scheduler to check whether there is any overrun and take the necessary error handling action whenever it finds any erroneous condition. These design objectives make some choices of frame size more desirable than the others.

5.3.2 Frame Size Constraints

Ideally, we want the frames to be sufficiently long so that every job can start and complete its execution within a frame. In this way, no job will be preempted. We can meet this objective if we make the frame size f larger than the execution time e_i of every task T_i. In other words,

$$f \geq \max_{1 \leq i \leq n} (e_i) \tag{5.1}$$

To keep the length of the cyclic schedule as short as possible, the frame size f should be chosen so that it divides H, the length of the hyperperiod of the system. This condition is met when f divides the period pi of at least one task Ti , that is, for at least one i

$$\lfloor p_i/f \rfloor - p_i/f = 0 \tag{5.2}$$

When this condition is met, there is an integer number of frames in each hyperperiod. We let F denote this number and call a hyperperiod that begins at the beginning of the $(kF + 1)$st frame, for any $k = 0,1,...$, a *major cycle*.

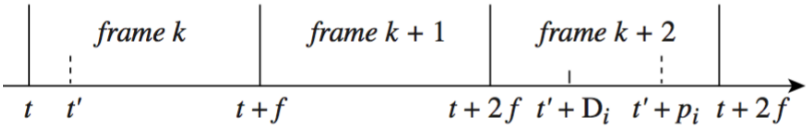


FIGURE 5–4 A constraint on the value of frame size.

On the other hand, to make it possible for the scheduler to determine whether every job completes by its deadline, we want the frame size to be sufficiently small so that between the release time and deadline of every job, there is at least one frame.

Figure 5–4 illustrates the suitable range of f for a task $T_i = (p_i, e_i, D_i)$. When f is in this range, there is at least one frame between the release time and deadline of every job in the task. In this figure, t denotes the beginning of a frame (called the kth frame) in which a job in T_i is released, and t' denotes the release time of this job. We need to consider two cases: $t' > t$ and $t' = t$. If t' is later than t, as shown in this figure, we want the $(k + 1)$st frame to be in the interval between the release time t' and the deadline $t' + D_i$ of this job. For this to be true, we must have $t + 2f$ equal to or earlier than $t' + D_i$, that is, $2f - (t' - t) \leq D_i$. Because the difference $t' - t$ is at least equal to the greatest common divisor $gcd(p_i, f)$ of p_i and f, this condition is met if the following inequality holds:

$$2f - gcd(p_i, f) \leq D_i \tag{5.3}$$

We want the inequality of Eq. (5.3) to hold for all $i = 1,2,...,n$. In the special case when t' is equal to t, it suffices to choose a frame size that is equal to or smaller than D_i. The condition $f \leq D_i$ is satisfied for all values of f that satisfy Eq. (5.3) and, hence, does not need to be considered separately. We refer to Eqs. (5.1), (5.2) and (5.3) as the *frame-size constraints*.

For the four tasks in Figure 5–1, we see that Eq (5.1) constrains the frame size to be no less than 2. Their hyperperiod length is 20; hence, 2, 4, 5, 10 and 20 are possible frame sizes according to Eq. (5.2). However, only 2 satisfies Eq. (5.3). Therefore, we must choose this frame size and can use the cyclic schedule shown in Figure 5–5. As another example, we consider the tasks (15, 1, 14), (20, 2,

26), and (22, 3). Because of Eq.(5.1), we must have $f \geq 3$; because of Eq.(5.2), we must have f =3, 4, 5, 10, 11, 15, 20, and 22; and because of Eq. (5.3), we must have f = 3, 4 or 5. Therefore the possible choices of the frame size are 3, 4, and 5.

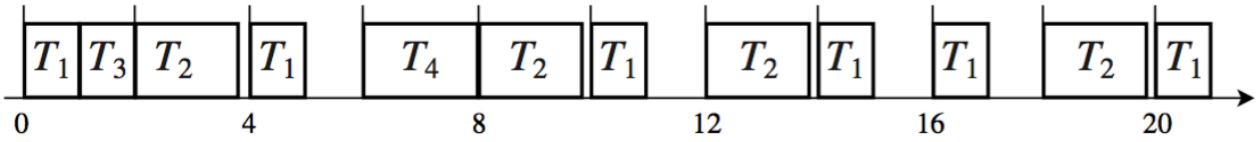


FIGURE 5–5 A cyclic schedule with frame size 2.

5.3.3 Job Slices

Sometimes, the given parameters of some task systems cannot meet all three frame size constraints simultaneously. An example is the system **T** = {(4, 1), (5, 2, 7), (20, 5)}. For Eq. (5.1) to be true, we must have $f \geq 5$, but to satisfy Eq. (5.3) we must have $f \leq 4$. In this situation, we are forced to partition each job in a task that has a large execution time into slices (i.e., subjobs) with smaller execution times. (When the job is a message transmission, we divide the message into several segments. When the job is a computation, we partition the program into procedures, each of which is to be executed nonpreemptively.) In this way, we can reduce the lower bound of f imposed by Eq. (5.1).

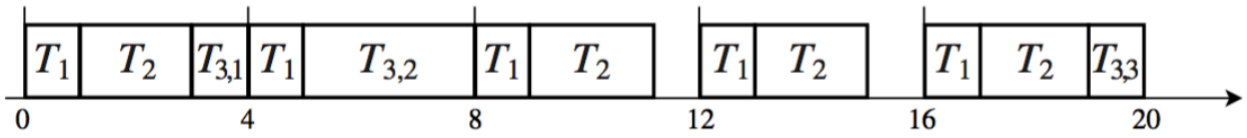


FIGURE 5–6 A preemptive cyclic schedule of $T_1 = (4, 1)$, $T_2 = (5, 2, 7)$ and $T_3 = (20, 5)$.

For **T** = {(4, 1), (5, 2, 7), (20, 5)}, we can divide each job in (20, 5) into a chain of three slices with execution times 1, 3, and 1. In other words, the task (20, 5) now consists of three subtasks (20, 1), (20, 3) and (20, 1). The resultant system has five tasks for which we can choose the frame size 4. Figure 5–6 shows a cyclic schedule for these tasks. The three original tasks are called T_1, T_2 and T_3, respectively, and the three subtasks of T_3 are called $T_{3,1}$, $T_{3,2}$, and $T_{3,3}$.

We choose to decompose (20, 5) into three subtasks. Why? To satisfy Eq. (5.1), it suffices for us to partition each job in the task into two slices, one with execution time 3 and the other with execution time 2. However, a look at the schedule in Figure 5–6 shows the necessity of three slices. It would not be possible to fit the two tasks (20, 3) and (20, 2) together with T_1 and T_2 in five frames of size 4. T_1, with a period of 4, must be scheduled in each frame. T_2, with a period of 5, must be scheduled in four out of the five frames. (The fact that the relative deadline of T_2 is 7 does not help.) This leaves one frame with 3 units of time for T_3. The other frames have only 1 unit of time left for T_3. We can schedule two subtasks each with 1 unit of execution time in these frames, but there is no time in any frame for a subtask with execution time 2.

From this example, we see that in the process of constructing a cyclic schedule, we have to make three kinds of design decisions:

- choosing a frame size,
- partitioning jobs into slices, and
- placing slices in the frames.

In general, these decisions cannot be made independently. The more slices a job is partitioned into, the higher the context switch and communication overhead. Therefore, we want to partition each job into as few slices as necessary to meet the frame-size constraints. Unfortunately, this goal is not always attainable. There may not be any feasible schedule for the choices of frame size and job slices because it is impossible to pack the large job slices into the frames by their deadlines. In contrast, feasible schedules may exist if we choose smaller job slices. For this reason, we sometimes are forced to partition some jobs into more slices than needed to meet the frame size constraints.

5.4 Cyclic executives

The clock-driven scheduler described in Figure 5–1 must be modified to accommodate the restriction that scheduling decisions are made only at frame boundaries. The cyclic executive approach is a way. In real-time systems literature, the term "cyclic executive" refers to a scheduler that deterministically interleaves and sequentializes the execution of periodic-tasks on a CPU according to a given cyclic schedule. Each job slice is a procedure. The cyclic executive executes a single do loop. Starting from the beginning of each frame, it executes in turn the slices scheduled in the frame. The flight controller is an example.

The term *cyclic executive* mean a table-driven cyclic scheduler for all types of jobs in a multithreaded system. It makes scheduling decisions only at the beginning of each frame and deterministically interleaves the execution of periodic tasks. However, it allows aperiodic and sporadic jobs to use the time not used by periodic tasks. The pseudocode in Figure 5–7 describes such a cyclic executive on a CPU. The stored table that gives the precomputed cyclic schedule has F entries, where F is the number of frames per major cycle. Each entry (say the kth) lists the names of the job slices that are scheduled to execute in frame k. In Figure 5–7, the entry is denoted by $L(k)$ and is called a *scheduling block*, or simply a block. The current block refers to the list of periodic job slices that are scheduled in the current frame.

```
Input: Stored schedule: L(k) for k = 0, 1, ..., F − 1;
   Aperiodic job queue
Task CYCLIC_EXECUTIVE:
   the current time t = 0;
   the current frame k = 0;
   do forever
      accept clock interrupt at time tf;
      currentBlock = L(k);
      t = t + 1;
      k = t mod F;
      if the last job is not completed, take appropriate action;
      if any of the slices in currentBlock is not released, take appropriate action;
      wake up the periodic task server to execute the slices in currentBlock;
      sleep until the periodic task server completes;
         while the aperiodic job queue is nonempty;
         wake up the job at the head of the aperiodic job queue;
         sleep until the aperiodic job completes;
         remove the aperiodic job from the queue;
      endwhile;
      sleep until the next clock interrupt;
   enddo;
end CYCLIC_EXECUTIVE
```

FIGURE 5–7 A table-driven cyclic executive.

The cyclic executive takes over the processor and executes at each of the clock interrupts, which occur at the beginning of frames. When it executes, the cyclic executive copies the table entry for the current frame into the current block. It then wakes up a job, called periodic task server, and lets the server execute the job slices in the current block. Upon the completion of the periodic task server, the cyclic executive wakes up the aperiodic jobs in the aperiodic job queue in turn and allows them to use the remaining time in the frame. The assumption here is that whenever the server or a job completes, the cyclic executive wakes up and executes. Alternatively, the system may have an aperiodic task server, which when awaked executes aperiodic jobs in the aperiodic job queue.

In addition to scheduling, the cyclic executive also checks for overruns at the beginning of each frame. If the last job executed in the previous frame is not complete at that time, the cyclic executive preempts the execution of the job if the last job is an aperiodic job. The job remains in the aperiodic job queue and will be resumed whenever there is time again for aperiodic jobs. If the cyclic executive finds the periodic task server still executing at the time of a clock interrupt, a frame overrun occurs; some slice(s) scheduled in the previous frame has executed longer than the time allocated to it by the precomputed cyclic schedule. The cyclic executive takes an appropriate action to recover from this frame overrun.

After checking for overruns, the cyclic executive makes sure that all the job slices scheduled in the current block are ready for execution and then wakes up the periodic task server to execute them. If there is still time after all the slices in the current block are completed and the aperiodic job queue is nonempty, it lets the job at the head of the aperiodic job queue execute.

Two important assumptions that must be valid in order for the cyclic executive to work as intended are:

1. The existence of a timer. The timer generates interrupts periodically without intervention.
2. Each timer interrupt is handled by the cyclic executive within a bounded amount of delay. Specifically, either this delay is negligibly small, or at least we know how large it can be and therefore can take this delay into account when we compute the cyclic schedule.

5.5 Improving the average response time of aperiodic jobs

Thus far we have paid no attention to the performance of the scheduler as far as the aperiodic jobs are concerned. They are scheduled in the background after all the job slices with hard deadlines scheduled in each frame are completed. However, the strategy of delaying the execution, and hence the completion, of aperiodic jobs in preference of periodic tasks is not a good one. There is no advantage to completing a job with a hard deadline early. On the other hand, an aperiodic job is released and executed by the system in response to an event. The sooner an aperiodic job completes, the more responsive the system is. For this reason, minimizing the response time of each aperiodic job or the average response time of all the aperiodic jobs is typically one of the design goals of real-time schedulers.

5.5.1 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*

For the slack-stealing scheme to work, every periodic job slice must be scheduled in a frame that ends no later than its deadline. Let the total amount of time allocated to all the slices scheduled in the frame k be x_k. The *slack* (time) available in the frame is equal to $f - x_k$ at the beginning of the frame. If the aperiodic job queue is nonempty at this time, the cyclic executive can let aperiodic jobs execute for this amount of time without causing any job to miss its deadline.

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

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

Example:

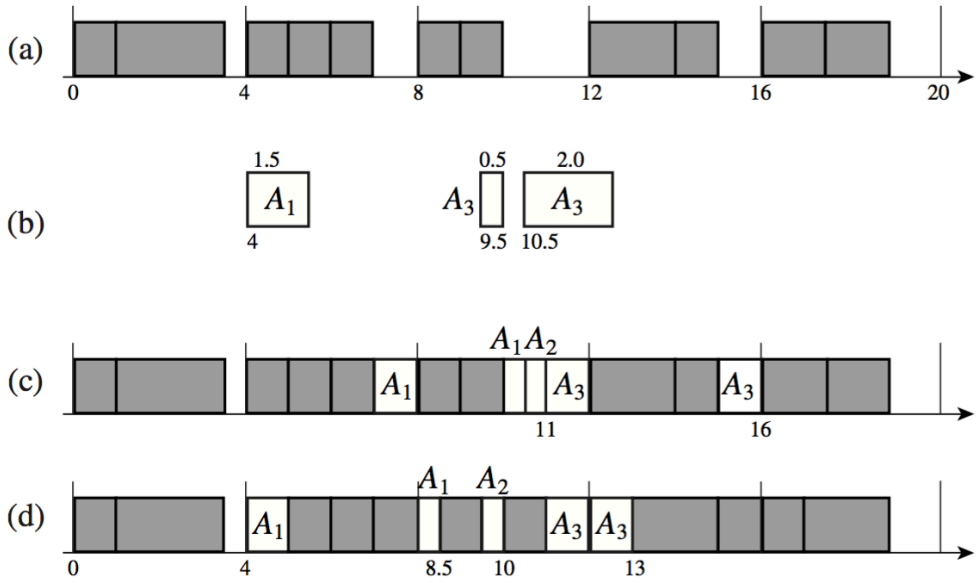


FIGURE 5–8 Example illustrating slack stealing.

Figure 5–8 gives an illustrative example. Figure 5–8(a) shows the first major cycle in the cyclic schedule of the periodic tasks. Figure 5–8(b) shows three aperiodic jobs A_1, A_2, and A_3. Their release times are immediately before 4, 9.5. and 10.5, and their execution times are 1.5, 0.5 and 2, respectively. Figure 5–8(c) shows when the aperiodic jobs execute if we use the cyclic executive shown in Figure 5–7, which schedules aperiodic jobs after the slices of periodic tasks in each frame are completed. The execution of A_1 starts at time 7. It does not complete at time 8 when the frame ends and is, therefore, preempted. It is resumed at time 10 after both slices in the next frame complete. Consequently, its response time is 6.5. A_2 executes after A_1 completes and has a response time equal to 1.5. Similarly, A_3 follows A_2 and is preempted once and completes at the end of the following frame. The response time of A_3 is 5.5. The average response time of these three jobs is 4.5.

Figure 5–8(d) shows what happens if the cyclic executive does slack stealing. At time 4, the cyclic executive finds A_1 in the aperiodic job queue, and there is 1 unit of slack. Hence it lets A_1 execute. At time 5, there is no more slack. It preempts A_1 and lets the periodic task server execute the job slices scheduled in the frame. At the beginning of the next frame, the available slack is 2. It resumes A_1, which completes at time 8.5. At the time, the first slice in the current block is executed, since the aperiodic job queue is empty. Upon completion of the slice at time 9.5, the cyclic executive checks the aperiodic job queue again, finds A_2 ready, and lets A_2 execute. When the job completes at time 10, the cyclic executive finds the aperiodic job queue empty and lets the periodic task server execute the next

job slice in the current block. At time 11, it finds A_3 and lets the job execute during the last unit of time in the frame, as well as in the beginning of the next frame. The job completes by time 13. According to this schedule, the response times of the jobs are 4.5, 0.5, and 2.5, with an average of 2.5.

Note: Response time of a job is the difference between the end time and the release time.

Let us now examine how to implement slack stealing. The initial amount of slack in each frame can be precomputed along with the cyclic schedule and stored in the table defining the schedule. It is necessary for the cyclic executive to keep track of the amount of available slack and update this amount as it consumes the slack. This can be done using an interval timer. At the beginning of each frame, the cyclic executive sets the timer to the value of the initial slack in the frame. The timer counts down whenever an aperiodic job executes ahead of any slice in the current block. When the timer expires, indicating that there is no more slack, the cyclic executive preempts the executing aperiodic job and lets the execution of the next job slice in the current block begin.

5.5.2 Average Response Time

While we are not required to ensure the completion of aperiodic jobs by some specific times, we are often required to guarantee that their average response time is no greater than some value. To give this guarantee, we need to be able estimate the average response time of these jobs.

In general, an accurate estimate of the average response time can be found only by simulation and/or measurement. This process is time consuming and can be done only after a large portion of the system is designed and built. On the other hand, we can apply known results in queuing theory to get a rough estimate of the average response time as soon as we know some statistical behavior of the aperiodic jobs. For example, a requirement of the system may be that it must respond satisfactorily as long as the average rate of arrival (i.e., releases) of aperiodic jobs is within a given limit. We can estimate the average response time of the system for this average arrival rate. We also need to know the mean and mean square values of the execution times of aperiodic jobs. These values can be estimated by analysis, simulation, and/or measurement of the jobs' execution by themselves.

To express average response time in terms of these parameters, let us consider a system in which there are n_a aperiodic tasks. The jobs in each aperiodic task have the same interarrival-time and execution-time distributions and the same response-time requirement. Suppose that the average rate of arrival of aperiodic jobs in the ith aperiodic task is λ_i jobs per unit of time. The sum λ of λ_i over all $i = 1, 2, \ldots a$ is the total number of aperiodic job arrivals per unit of time. The mean and the mean square values of the execution times of jobs in the i th aperiodic task are $E[\beta_i]$ and $E[\beta_i^2]$, respectively. (Here $E[x]$ denotes the mean value of the random variable x. $E[x]$ is also called the expected value of x, hence the choice of the letter E.) Let u_i denote the average utilization of the ith task; it is the average fraction of processor time required by all the jobs in the task. u_i is equal to $\lambda_i E[\beta_i]$. We call the sum U_A of u_i over all aperiodic tasks the *total average utilization of aperiodic tasks*; it is the average fraction of processor time required by all the aperiodic tasks in the system.

Let U be the total utilization of all the periodic tasks. $1 - U$ is the fraction of time that is available for the execution of aperiodic jobs. We call it the *aperiodic* (*processor*) *bandwidth*. If the total average utilization of the aperiodic jobs is larger than or equal to the aperiodic bandwidth of the system (i.e., $U_A \geq 1 - U$), the length of the aperiodic job queue and the average response time will grow without bound. Hence we consider here only the case where $U_A < 1 - U$.

When the jobs in all aperiodic tasks are scheduled on the FIFO basis, we can estimate the average response time W (also known as waiting time) of any aperiodic job by the following expression

$$W = \sum_{i=1}^{n_a} \frac{\lambda_i E[\beta_i]}{\lambda(1-U)} + \frac{W_0}{(1-U)^2[1-U_A/(1-U)]} \qquad (5.4a)$$

where W_0 is given by

$$W_0 = \sum_{i=1}^{n_a} \frac{\lambda_i E[\beta_i{}^2]}{2} \qquad (5.4b)$$

The first term in Eq. (5.4a) gives the average amount of time required by an aperiodic job to complete execution if it does not wait for any aperiodic job. The second term gives us the average queuing time, which is the average amount of time a job waits in the queue.

5.6 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.

5.6.1 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. By rejecting a sporadic job that cannot be scheduled to complete in time immediately after the job is released, the scheduler gives the application system as much time as there is to take any necessary recovery action.

Example:

To illustrate that this approach is a reasonable one, we consider a quality control system. A sporadic job that activates a robotic arm is released when a defective part is detected. The arm, when activated, removes the part from the conveyor belt. This job must complete before the part moves beyond the reach of the arm. When the job cannot be scheduled to complete in time, it is better for the system to have this information as soon as possible. The system can slow down the belt, stop the belt, or alert an operator to manually remove the part. Otherwise, if the sporadic job were scheduled but completed too late, its lateness would not be detected until its deadline. By the time the system attempts a recovery action, the defective part may already have been packed for shipment, too late for simple recovery actions to be effective.

We assume that the maximum execution time of each sporadic job becomes known upon its release. It is impossible for the scheduler to determine which sporadic jobs to admit and which to reject unless this information is available. Therefore, the scheduler must maintain information on the maximum execution times of all types of sporadic jobs that the system may execute in response to the events it is required to handle. We also assume that all sporadic jobs are preemptable. Therefore, each sporadic

job can execute in more than one frame if no frame has a sufficient amount of time to accommodate the entire job.

How Acceptance Test is done? - Algorithm

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 $e > \sigma_c(t,l)$. 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.

In general, more than one sporadic job may be waiting to be tested at the same time. A good way to order them is on the Earliest-Deadline-First (EDF) basis. In other words, newly released sporadic jobs are placed in a waiting queue ordered in nondecreasing order of their deadlines: the earlier the deadline, the earlier in the queue. The scheduler always tests the job at the head of the queue and removes the job from the waiting queue after scheduling it or rejecting it.

5.6.2 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.

Figure 5–11 gives an example. The frame size used here is 4. The shaded boxes show where periodic tasks are scheduled.

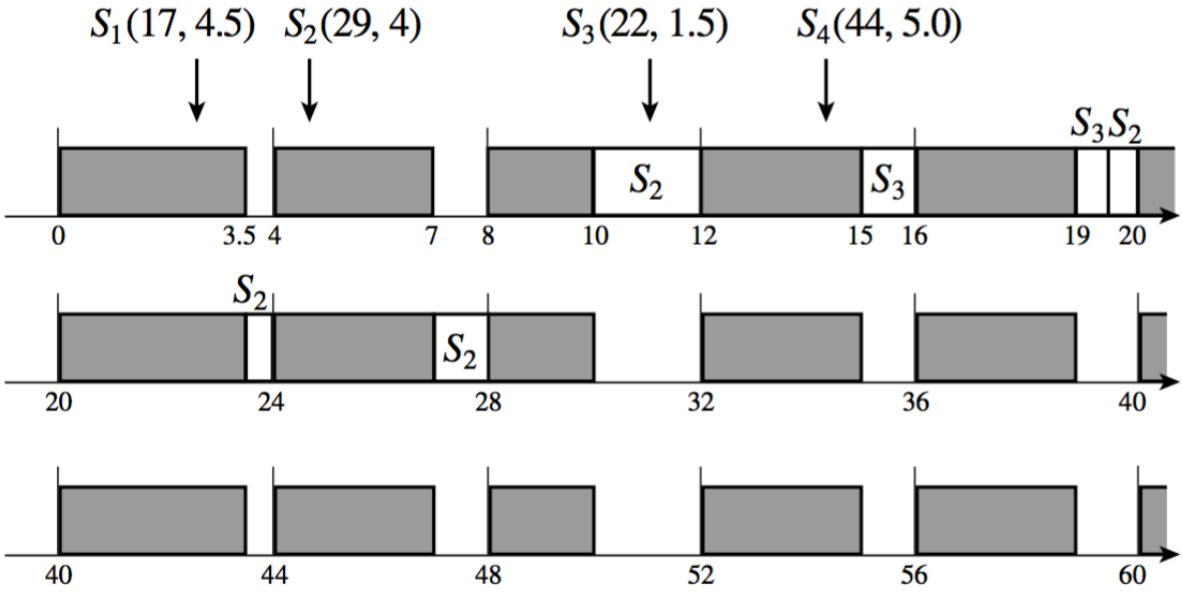


FIGURE 5–11 Example of scheduling sporadic jobs.

- 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 S_2 and S_3. Therefore, it rejects S_4. When the remaining portion of S_3 completes in the current frame, S_2 executes until the beginning of the next frame.
- The last portion of S_2 executes in frames 6 and 7.

5.7 Practical considerations and generalization

Thus far, we have ignored many practical problems, such as how to handle frame overruns, how to do mode changes, and how to schedule tasks on multiprocessor systems. We discuss them now.

5.7.1 Handling Frame Overruns

A frame overrun can occur for many reasons. For example, when the execution time of a job is input data dependent, it can become unexpectedly large for some rare combination of input values which is not taken into account in the precomputed schedule. A transient hardware fault in the system may cause some job to execute longer than expected. A software flaw that was undetected during debugging and testing can also cause this problem. There are many ways to handle a frame overrun. Which one is the most appropriate depends on the application and the reason for the overrun.

- A way to handle overruns is to simply abort the overrun job at the beginning of the next frame and log the premature termination of the job. Such a fault can then be handled by some recovery mechanism later when necessary. This way seems attractive for applications where late results are no longer useful.
- However, premature termination of overrun jobs may put the system in some inconsistent state, and the actions required to recover from the state and maintain system integrity may be costly. For this reason, in most real-life systems, a job that overruns its allocated time and is found executing at the end of the frame is preempted immediately, if it is not in a critical section at the time, or as soon as it exits the critical section, if it is. The unfinished portion executes as an aperiodic job during the slack time in the subsequent frame(s) or in the background whenever there is spare time.
- Another way to handle an overrun is to continue to execute the offending job. The start of the next frame and the execution of jobs scheduled in the next frame are then delayed. Letting a late job postpone the execution and completion of jobs scheduled after it can in turn cause these jobs to be late. This way is appropriate only if the late result produced by the job is nevertheless useful, and an occasional late completion of a periodic job is acceptable.

5.7.2 Mode Changes

As stated, the number *n* of periodic tasks in the system and their parameters remain constant as long as the system stays in the same (operation) mode. During a *mode change*, the system is reconfigured. Some periodic tasks are deleted from the system because they will not execute in the new mode. Periodic tasks that execute in the new mode but not in the old mode are created and added to the system. The periodic tasks that execute in both modes continue to execute in a timely fashion. When the mode change completes, the new set of periodic tasks are scheduled and executed.

We assume that the parameters of periodic tasks to be executed in the new mode are also known. The schedule of the tasks executed in the new mode is also precomputed. However, the new schedule table may not be in memory during the old mode. This table must be brought into memory. Similarly, the code of the new tasks must be brought into memory and memory space for data accessed by them must be allocated before their execution begins. The work to configure the system is a mode-change job; the job is released in response to a mode- change command. We need to consider two cases: The mode-change job has either a soft or hard deadline. In both cases, we assume that periodic tasks are independent and hence can be added and deleted independently.

- **Aperiodic and Sporadic Mode Change**

A reasonable way to schedule a mode-change job that has a soft deadline is to treat it just like an ordinary aperiodic job, except that it may be given the highest priority and executed ahead of other aperiodic jobs. Once the job begins to execute, however, it may modify the old schedule in order to speed up the mode change. A periodic task that will not execute in the new mode can be deleted and its memory space and processor time freed as soon as the current job in the task completes. This scheme can be implemented by letting the scheduler or the mode-change job mark each periodic task that is to be deleted. During mode change, the scheduler continues to use the old schedule table. Before the periodic task server begins to execute a periodic job, however, it checks whether the corresponding task is marked and returns immediately if the task is marked. In this way, the schedule of the periodic tasks that execute in both modes remain unchanged during mode change, but the time allocated to the deleted task can be used to execute the mode-change job. Once the new schedule table and code of the new tasks are in memory, the scheduler can switch to use the new table.

A question that remains to be answered is how aperiodic and sporadic jobs in the system should be dealt with during mode changes. It may no longer be safe to execute some of these jobs; they should be deleted as well. Since the deadlines of the remaining aperiodic jobs are soft, their execution can be delayed until after the mode change. On the other hand, the sporadic jobs should not be affected by the mode change, since their on-time completion has been guaranteed. One way to ensure their on-time completion is to defer the switchover from the old schedule table to the new schedule table until all the sporadic jobs in the system complete. Clearly this option can lengthen the response time of the mode change. Another option is have the mode-change job check whether the sporadic jobs in the system can complete in time according to the new schedule. The schedule switchover is deferred only when some sporadic job cannot complete in time according to the new schedule. In the case when the delay thus introduced is unacceptably long, the scheduler may be forced to switchover to the new schedule, let some sporadic jobs complete late, and leave the affected applications to handle the timing faults.

5.7.3 General Workloads and Multiprocessor Scheduling

The clock-driven approach is applicable to workloads that are not characterizable by the periodic task model. As long as the parameters of the jobs are known a priori, a static schedule can be computed off-line. Precedence constraints and other types of dependencies and contentions for resources among

jobs constrain when the jobs may be scheduled relative to each other. The algorithms used to search for a feasible schedule can take these constraints into account.

It is conceptually straightforward to schedule tasks on several processors whenever the workload parameters are known a priori and there is a global clock. We can construct a global schedule which specifies on what processor each job executes and when the job executes. As long as the clock drifts on the processors are sufficiently small, we can use the uniprocessor schedulers described in earlier sections on each processor to enforce the execution of the jobs according to the global schedule.

Sometimes, a precomputed multiprocessor schedule can be found straightforwardly from a precomputed uniprocessor schedule. As an example, Figure 5–13(a) shows a system containing several CPUs connected by a system bus. Each task consists of a chain of jobs, which executes on one of the CPUs and sends or receives data from one of the I/O devices via the system bus. In a system with such an architecture, the system bus is sometimes the bottleneck. By the bus being the bottleneck, we mean that if there is a feasible schedule of all the data transfer activities on the bus, it is always possible to feasibly schedule the jobs that send and receive the data on the respective CPUs. To illustrate, Figure 5–13(b) shows a cyclic schedule of the data-transfer activities on the bus and the schedules of CPUs and I/O device interfaces that produce and consume the data. The shaded boxes on the time lines of CPUS1 and CPUS2 show when the CPUs execute in order to produce and consume the data that occupy the bus in time intervals shown by shaded boxes on the time line of the bus. Similarly, the CPUD is the producer of the data transferred to an I/O device during intervals shown as dotted boxed on the bus time line. We can see that the schedules of the CPUs and I/O devices can be derived directly from the schedule of the bus. Computing the schedule for the entire system is simplified to computing the schedule of the system bus.

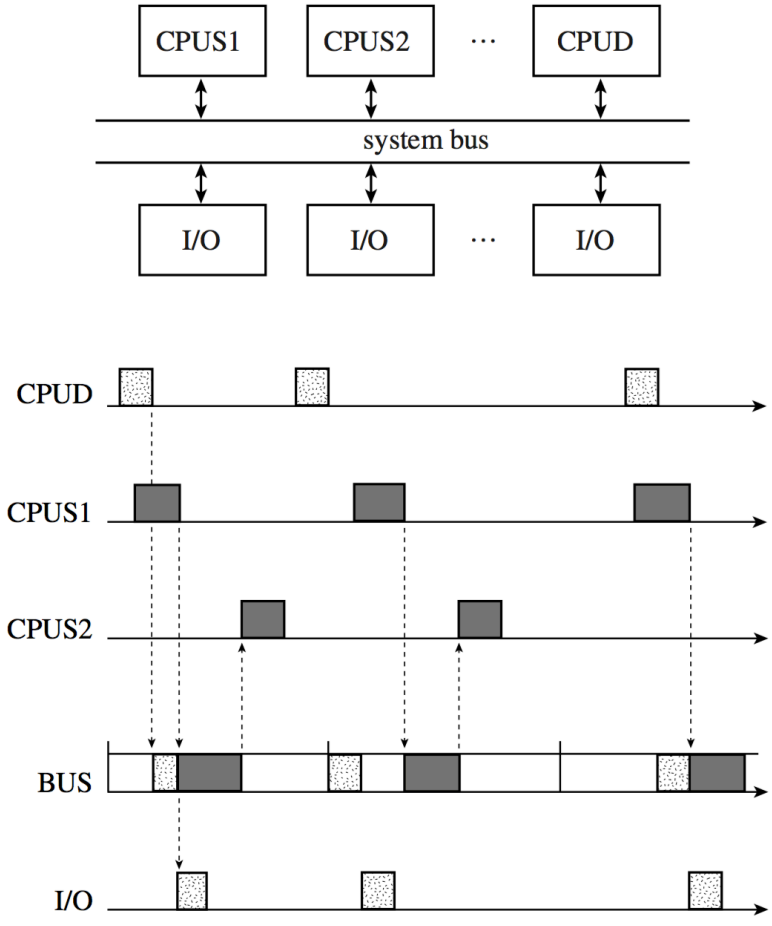


FIGURE 5–13 A simple clock-driven multiprocessor schedule.

This example is based on the Boeing 777 Airplane Information Management System (AIMS). The system uses a table-driven system bus protocol. The protocol controls the timing of all data transfers. The intervals when the bus interface unit of each CPU must execute (and hence when each CPU must execute) are determined by the schedule of the system bus in a manner illustrated by this example.

5.8 Algorithm for constructing static schedules

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

A system of independent, preemptable periodic tasks whose relative deadlines are equal to or greater than their respective periods is schedulable if and only if the total utilization of the tasks is no greater than 1. Because some tasks may have relative deadlines shorter than their periods a feasible schedule may not exist even when this condition is met. The iterative algorithm described below enables us to find a feasible cyclic schedule if one exists. The algorithm is called the *iterative network-flow algorithm*, or the *INF algorithm* for short. Its key assumptions are that tasks can be preempted at any time and are independent.

Before applying the INF algorithm on the given system of periodic tasks, we find all the possible frame sizes of the system: These frame sizes met the constraints of Eqs. (5.2) and (5.3) but not necessarily satisfy Eq. (5.1). [For example, the possible frame sizes of tasks $T_1 = (4,1)$, $T_2 = (5,2,7)$, and $T_3 = (20,5)$ are 2 and 4. They satisfy Eqs. (5.2) and (5.3), but not Eq. (5.1).] The INF algorithm iteratively tries to find a feasible cyclic schedule of the system for a possible frame size at a time, starting from the largest possible frame size in order of decreasing frame size. A feasible schedule thus found tells us how to decompose some tasks into subtasks if their decomposition is necessary. If the algorithm fails to find a feasible schedule after all the possible frame sizes have been tried, the given tasks do not have a feasible cyclic schedule that satisfies the frame size constraints even when tasks can be decomposed into subtasks.

5.8.1 Network-Flow Graph

The algorithm used during each iteration is based on the well known 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 J_1, J_2, . . . , J_N. The constraints on when the jobs can be scheduled are represented by the *network-flow graph* of the system. This graph contains the following vertices and edges; the capacity of an edge is a nonnegative number associated with the edge.

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

A *flow of an edge* is a nonnegative number that satisfies the following constraints:

1. It is no greater than the capacity of the edge and
2. With the exception of the *source* and *sink*, the sum of the flows of all the edges into every vertex is equal to the sum of the flows of all the edges out of the vertex.

Figure 5–14 shows part of a network-flow graph. For simplicity, only job vertices J_i and J_k are shown. The label "(capacity), flow" of the each edge gives its capacity and flow. This graph indicates that job J_i can be scheduled in frames x and y and the job J_k can be scheduled in frames y and z.

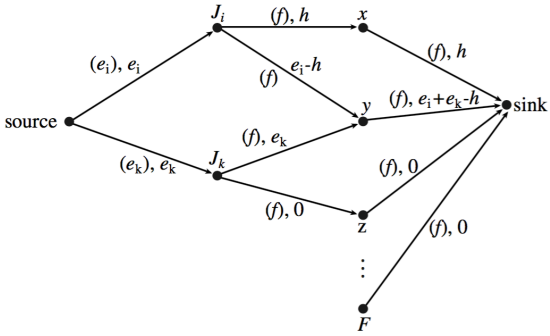


FIGURE 5–14 Part of a network-flow graph.

A *flow of a network-flow graph*, or simply a flow, is the sum of the flows of all the edges from the *source*; it should equal to the sum of the flows of all the edges into the *sink*. There are many algorithms for finding the maximum flows of network-flow graphs.

5.9 Pros and cons of clock-driven scheduling

The clock-driven approach to scheduling has many advantages. The most important one is its conceptual simplicity. We can take into account complex dependencies, communication delays, and resource contentions among jobs in the choice and construction of the static schedule, making sure there will be no deadlock and unpredictable delays. A static schedule can be represented by a table of start times and completion times that is interpreted by the scheduler at run time. By changing the table, jobs can be scheduled according to different static schedules in different operation modes. There is no need for concurrency control. Precedence constraints and other types of dependency can be taken care of by the choice of the schedule. Hence, there is no need for any synchronization mechanism. Some applications have tight completion-time jitter requirements. Again, such requirements can be taken into account in the choice of the cyclic schedule.

When the workload is mostly periodic and the schedule is cyclic, timing constraints can be checked and enforced at each frame boundary. Context switching and communication overhead can be kept low by choosing as large a frame size as possible so as many jobs as possible can execute from start to finish without interruption. This is a good scheduling strategy for applications where variations in time and resource requirements are small. Many traditional real-time applications (e.g., the patient care system and the flight control system mentioned in Sections 1.1 and 1.2) are examples.

In practice, systems based on the clock-driven scheduling paradigm are typically *time-triggered*. In a time-triggered system, interrupts in response to external events are queued and polled periodically. The periods are chosen to be integer multiples of the frame size. This design choice simplifies the construction of the static schedules and eliminates the waste in slack time. The system and its static schedule(s) can be simplified further if the periods of periodic tasks are chosen to be harmonic

Time-triggered systems based on the clock-driven scheduling approach are relatively easy to validate, test, and certify. Because the times when jobs execute are deterministic, a system scheduled in this manner will not exhibit any kind of anomalies. For this reason, it is possible to determine whether the system indeed meets all of its timing requirements by exhaustive simulation and testing.

However, the clock-driven approach also has many disadvantages. The most obvious one is that a system based on this approach is brittle: It is relatively difficult to modify and maintain. For example, an enhancement that leads to changes in execution times of some tasks or the addition of new tasks often requires that a new schedule be constructed. Consequently, the approach is suited only for systems (e.g., small embedded controllers) which are rarely modified once built.

Other disadvantages include the ones listed below.

1. The release times of all jobs must be fixed. In contrast, priority-driven algorithms do not require fixed release times. We can guarantee the timely completion of every job in a priority-driven system as long as the interrelease times of all jobs in each periodic task are never less than the period of the task. This relaxation of the release-time jitter requirement often eliminates the need for global clock synchronization and permits more design choices.
2. In a clock-driven system, all combinations of periodic tasks that might execute at the same time must be known a priori so a schedule for the combination can be precomputed. This restriction is clearly not acceptable for applications which must be reconfigurable on-line and the mix of periodic tasks cannot be predicted in advance. As we will see in the next chapter, a priority-driven system does not have this restriction. We can vary the number and parameters of periodic tasks in the system provided we subject each new periodic task to an on-line acceptance test.
3. The pure clock-driven approach is not suitable for many systems that contain both hard and soft real-time applications. You may have noticed that the methods for scheduling aperiodic and sporadic tasks in the midst of periodic tasks described in earlier sections in fact schedule the aperiodic and sporadic tasks in a priority-driven manner.

6 Priority-Driven Scheduling of Periodic Tasks

This chapter describes well-known priority-driven algorithms for scheduling periodic tasks on a processor and examines the merits and limitations of these algorithms. It is the first of three chapters devoted to priority-driven scheduling on one processor of applications characterizable by the periodic task model. The simplifying assumptions made in this chapter are that

1. the tasks are independent and
2. there are no aperiodic and sporadic tasks.

In most of this chapter, we will confine our attention to the case where every job is ready for execution as soon as it is released, can be preempted at any time, and never suspends itself. Scheduling decisions are made immediately upon job releases and completions. Moreover, the context switch overhead is negligibly small compared with execution times of the tasks, and the number of priority levels is unlimited. At the end of the chapter, we will remove these restrictions and discuss the effects of these and other practical factors.

Most of our discussion is in terms of a fixed number of periodic tasks. The assumption here is that some protocol is used to regulate changes in the number or parameters of periodic tasks. In particular, when an application creates a new task, the application first requests the scheduler to add the new task by providing the scheduler with relevant parameters of the task, including its period, execution time, and relative deadline. Based on these parameters, the scheduler does an acceptance test on the new periodic task. In this test, the scheduler uses one of the methods described in this chapter to determine whether the new task can be feasibly scheduled with all the other existing tasks in the system. It accepts and adds the new task to the system only if the new task and all other existing tasks can be feasibly scheduled. Otherwise, the scheduler rejects the new task. (We assume that the application system deals with the rejections of new task requests and recovers in an application-dependent way.) When tasks are independent, the scheduler can delete any task and add an acceptable task at any time without causing any missed deadline.

6.1 Static assumption

We focus on uniprocessor systems. Why examine the problems of uniprocessor scheduling and synchronization in so much detail when most real-time systems today and in the future contain more than one processor? To answer this question, we recall that 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.

The dynamic approach should allow the processors to be more fully utilized on average as the workload fluctuates. Indeed, it may perform well most of the time. However, in the worst case, the performance of priority-driven algorithms can be unacceptably poor.

Example

The application system contains $m + 1$ independent periodic tasks. The first m tasks T_i, for $i = 1, 2, \ldots m$, are identical. Their periods are equal to 1, and their execution times are equal to 2ε, where ε is a small number. The period of the last task T_{m+1} is $1 + \varepsilon$, and its execution time is 1. The tasks are in phase. Their relative deadlines are equal to their periods. Suppose that the priorities of jobs in these

tasks are assigned on an EDF basis. The first job $J_{m+1,1}$ in T_{m+1} has the lowest priority because it has the latest deadline.

Figure 6–1 shows an EDF schedule of these jobs if the jobs are dispatched and scheduled dynamically on m processors. We see that $J_{m+1,1}$ does not complete until $1 + 2\varepsilon$ and, hence, misses its deadline. The total utilization U of these $m + 1$ periodic tasks is $2m\varepsilon + 1/(1 + \varepsilon)$. 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.

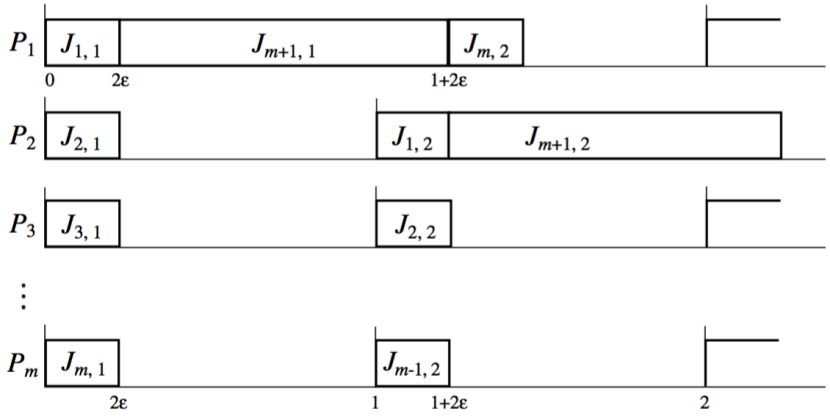


FIGURE 6–1 A dynamic EDF schedule on m processors.

It is arguable that the poor behavior of dynamic systems occurs only for some pathological system configurations, and some other algorithms may perform well even for the pathological cases. In most cases, the performance of dynamic systems is superior to static systems. The more troublesome problem with dynamic systems is the fact that we often do not know how to determine their worst-case and best-case performance.

For these reasons, most hard real-time systems built and in use to date and in the near future are static. In the special case when tasks in a static system are independent, we can consider the tasks on each processor independently of the tasks on the other processors. The problem of scheduling in multiprocessor and distributed systems is reduced to that of uniprocessor scheduling. In general, tasks may have data and control dependencies and may share resources on different processors. Uniprocessor algorithms and protocols can easily be extended to synchronize tasks and control their accesses to global resources in multiprocessor and distributed environments.

6.2 Fixed-priority versus dynamic-priority algorithms

A priority-driven scheduler (i.e., a scheduler which schedules jobs according to some priority-driven algorithm) is an on-line scheduler. It does not precompute a schedule of the tasks. Rather, it assigns priorities to jobs after they are released and places the jobs in a ready job queue in priority order. When preemption is allowed at any time, a scheduling decision is made whenever a job is released or completed. At each scheduling decision time, the scheduler updates the ready job queue and then schedules and executes the job at the head of the queue.

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."

Most real-time scheduling algorithms of practical interest assign fixed priorities to individual jobs. The priority of each job is assigned upon its release when it is inserted into the ready job queue. Once assigned, the priority of the job relative to other jobs in the ready job queue does not change. In other words, at the level of individual jobs, the priorities are fixed, even though the priorities at the task level are variable. We have three categories of algorithms:

- fixed-priority algorithms,
- task-level dynamic-priority (and job- level fixed-priority) algorithms, and
- job-level (and task-level) dynamic algorithms.

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

6.2.1 Rate-Monotonic and Deadline-Monotonic Algorithms

A well-known fixed-priority algorithm is the *rate-monotonic* algorithm. This algorithm assigns priorities to tasks based on their periods: the shorter the period, the higher the priority. The *rate* (of job releases) of a task is the inverse of its period. Hence, the higher its rate, the higher its priority. This algorithm is referred to as the RM algorithm for short and a schedule produced by the algorithm as an RM schedule.

Example:

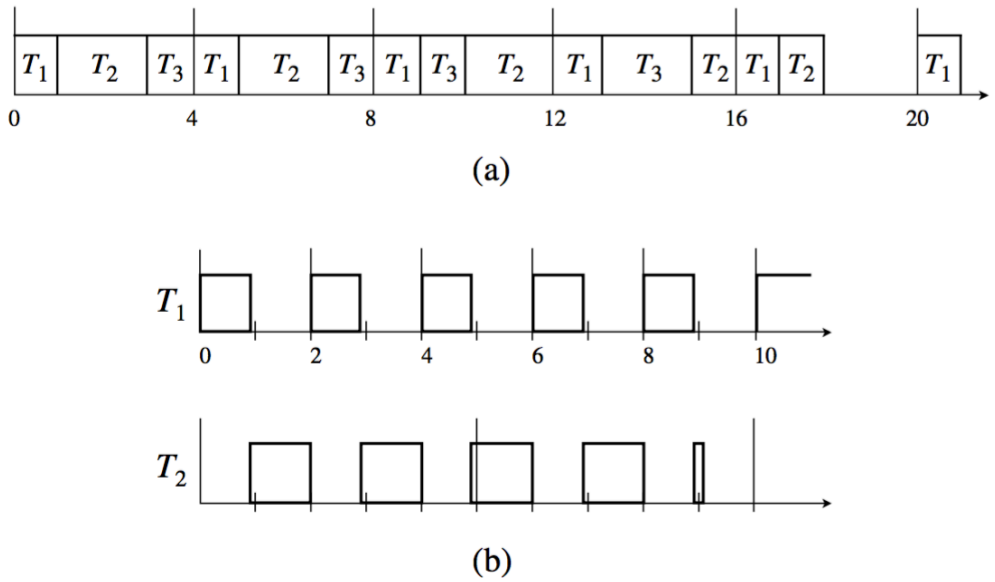


FIGURE 6–2 Examples of RM schedules. (a) RM schedule of $T_1 = (4, 1)$, $T_2 = (5, 2)$, and $T_3 = (20, 5)$. (b) RM schedule of $T_1 = (2, 0.9)$ and $T_2 = (5, 2.3)$.

This system contains three tasks: $T_1 = (4, 1)$, $T_2 = (5, 2)$, and $T_3 = (20, 5)$. The priority of T_1 is the highest because its rate is the highest (or equivalently, its period is the shortest). Each job in this task is placed at the head of the priority queue and is executed as soon as the job is released. T_2 has the next highest priority. Its jobs execute in the background of T_1. For this reason, the execution of the first job in T_2 is delayed until the first job in T_1 completes, and the third job in T_2 is preempted at time 16 when the fourth job in T_1 is released. Similarly, T_3 executes in the background of T_1 and T_2; the jobs in T_3 execute only when there is no job in the higher-priority tasks ready for execution. Since

there is always at least one job ready for execution until time 18, the processor never idles until that time.

The schedule in Figure 6–2(b) is for the tasks $T_1 = (2, 0.9)$ and $T_2 = (5, 2.3)$. The tasks are in phase. Here, we represent the schedule in a different form. Instead of using one time line, sometimes called a Gantt chart, to represent a schedule on a processor as we have done thus far, we use a time line for each task. Each time line is labeled at the left by the name of a task; the time line shows the intervals during which the task executes. According to the RM algorithm, task T_1 has a higher-priority than task T_2. Consequently, every job in T_1 is scheduled and executed as soon as it is released. The jobs in T_2 are executed in the background of T_1.

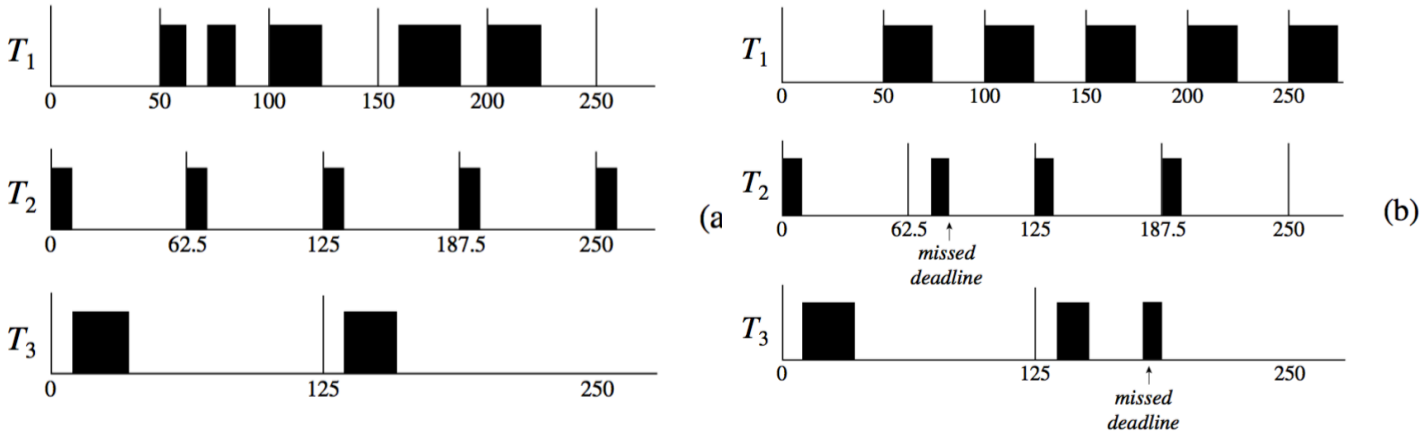


FIGURE 6–3 Fixed-priority schedules of $T_1 = (50, 50, 25, 100)$, $T_2 = (0, 62.5, 10, 20)$ and $T_3 = (0, 125, 25, 50)$.

Another well-known fixed-priority algorithm is the *deadline-monotonic algorithm* (DM). This algorithm assigns priorities to tasks according their relative deadlines: the shorter the relative deadline, the higher the priority. Figure 6–3 gives an example. The system consists of three tasks. They are $T_1 = (50, 50, 25, 100)$, $T_2 = (0, 62.5, 10, 20)$, and $T_3 = (0, 125, 25, 50)$. Their utilizations are 0.5, 0.16, and 0.2, respectively. The total utilization is 0.86. According to the DM algorithm, T_2 has the highest priority because its relative deadline 20 is the shortest among the tasks. T_1, with a relative deadline of 100, has the lowest priority. The resultant DM schedule is shown in Figure 6–3(a). According to this schedule, all the tasks can meet their deadlines.

Clearly, when the relative deadline of every task is proportional to its period, the RM and DM algorithms are identical. When the relative deadlines are arbitrary, the DM algorithm performs better in the sense that it can sometimes produce a feasible schedule when the RM algorithm fails, while the RM algorithm always fails when the DM algorithm fails. The example above illustrates this fact. Figure 6–3(b) shows the RM schedule of the three tasks which Figure 6–3(a) has shown to be feasible when scheduled deadline-monotonically. According to the RM algorithm, T_1 has the highest priority, and T_3 has the lowest priority. We see that because the priorities of the tasks with short relative deadlines are too low, these tasks cannot meet all their deadlines.

6.2.2 Well-Known Dynamic Algorithms

EDF algorithm

The EDF algorithm assigns priorities to individual jobs in the tasks according to their absolute deadlines; it is a dynamic-priority algorithm. Figure 6–4 shows the EDF schedule of the two tasks T_1 and T_2 whose RM schedule is depicted in Figure 6–2(b).

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

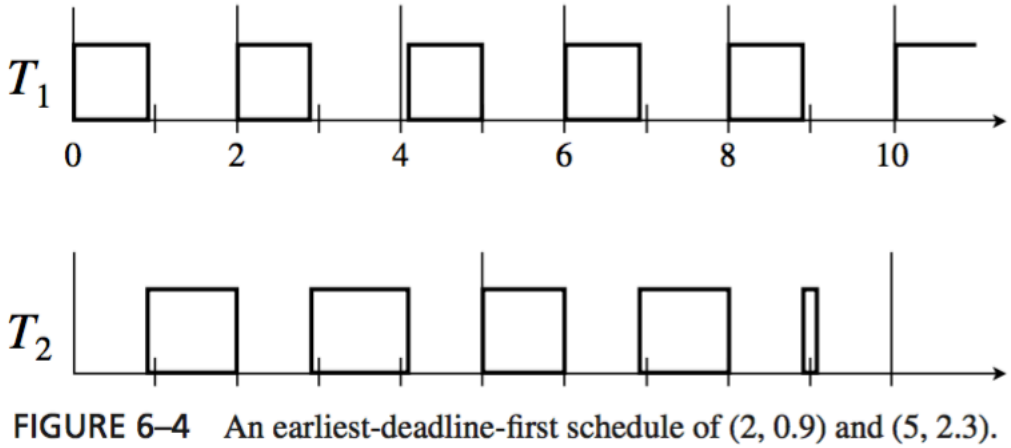


FIGURE 6–4 An earliest-deadline-first schedule of (2, 0.9) and (5, 2.3).

We note that the priority of T_1 is higher than the priority of T_2 from time 0 until time 4.0. T_2 starts to have a higher priority at time 4. When the job $J_{2,2}$ is released, T_2 again has a lower priority. Hence, the EDF algorithm is a task-level dynamic-priority algorithm. On the other hand, once a job is placed in the ready job queue according to the priority assigned to it, its order with respect to other jobs in the queue remains fixed. In other words, the EDF algorithm is a job-level fixed-priority algorithm.

***Least-Slack-Time-First* (LST) algorithm**

Another well-known dynamic-priority algorithm is the *Least-Slack-Time-First* (LST) algorithm. You recall that at time t, the slack of a job whose remaining execution time (i.e., the execution of its remaining portion) is x and whose deadline is d is equal to $d - t - x$. The scheduler checks the slacks of all the ready jobs each time a new job is released and orders the new job and the existing jobs on the basis of their slacks: the smaller the slack, the higher the priority.

Coincidentally, the schedule of T_1 and T_2 in the above example produced by the LST algorithm happens to be identical to the EDF schedule in Figure 6–4. In general, however, the LST schedule of a system may differ in a fundamental way from the EDF schedule.

Example

To illustrate, we consider a more complicated system that consists of three tasks: $T_1 = (2, 0.8)$, $T_2 = (5, 1.5)$, and $T_3 = (5.1, 1.5)$. When the first jobs $J_{1,1}$, $J_{2,1}$ and $J_{3,1}$ are released at time 0, their slacks are

1.2, 3.5, and 3.6, respectively. $J_{1,1}$ has the highest priority, and $J_{3,1}$ has the lowest. At time 0.8, $J_{1,1}$ completes, and $J_{2,1}$ executes. When $J_{1,2}$ is released at time 2, its slack is 1.2, while the slacks of $J_{2,1}$ and $J_{3,1}$ become 2.7 and 1.6, respectively. Hence, $J_{1,2}$ has the highest priority, but now $J_{3,1}$ has a higher-priority than $J_{2,1}$! From this example, we see that the LST algorithm is a job-level dynamic-priority algorithm, while the EDF algorithm is a job-level fixed-priority algorithm.

Strict and non-strict LST algorithm (why is strict LST not desirable?)

Because scheduling decisions are made only at the times when jobs are released or completed, this version of the LST algorithm does not follow the LST rule of priority assignment at all times. If we wish to be specific, we should call this version of the LST algorithm the *nonstrict LST* algorithm. If the scheduler were to follow the LST rule strictly, it would have to monitor the slacks of all ready jobs and compare them with the slack of the executing job. It would reassign priorities to jobs whenever their slacks change relative to each other. As an example, according to the schedule in Figure 6–4, the scheduler would find that at time 2.7, the slack of $J_{2,1}$ becomes $(5 - 2.7 - 1.2) = 1.1$, the same as that of $J_{1,2}$. It would schedule the two ready jobs in a round-robin manner until $J_{1,2}$ completes. The run-time overhead of the strict LST algorithm includes the time required to monitor and compare the slacks of all ready jobs as time progresses. Moreover, by letting jobs with equal slacks execute in a round-robin manner, these jobs suffer extra context switches. For this reason, the strictly LST algorithm is an unattractive alternative.

FIFO and Last-in-First-Out (LIFO) algorithms

According to our classification, FIFO and Last-in-First-Out (LIFO) algorithms are also dynamic-priority algorithms. As an example, suppose that we have three tasks: $T_1 = (0, 3, 1, 3)$, $T_2 = (0.5, 4, 1, 1)$, and $T_3 = (0.75, 7.5, 2, 7.5)$. Suppose that the jobs in them are scheduled on the FIFO basis. Clearly, $J_{1,1}$ has a higher priority than $J_{2,1}$, which in turn has a higher priority than $J_{3,1}$. In other words, T_1 has the highest priority, and T_3 has the lowest priority initially. Later, $J_{1,4}$, $J_{2,3}$, and $J_{3,2}$ are released at the times 9, 8.5, and 8.25, respectively, and T_3 has the highest priority while T_1 has the lowest priority.

6.3 Maximum schedule utilization-theorem

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.

How large the total utilization of a system can be in order for the system to be surely schedulable?

6.3.1 Schedulable Utilizations of the EDF Algorithm

We first focus on the case where the relative deadline of every task is equal to its period. (This choice of the relative deadline arises naturally from throughput considerations. The job in each period completes before the next period starts so there is no backlog of jobs.) The following theorem tells us that any such system can be feasibly scheduled if its total utilization is equal to or less than one, no matter how many tasks there are and what values the periods and execution times of the tasks are.

THEOREM 6.1. 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.

The following facts follow straightforwardly from this theorem.

- A system of independent, preemptable periodic tasks with relative deadlines longer than their periods can be feasibly scheduled on a processor as long as the total utilization is equal to or less than 1.
- The schedulable utilization $U_{EDF}(n)$ of the EDF algorithm for n independent, preemptable periodic tasks with relative deadlines equal to or larger than their periods is equal to 1.

The EDF algorithm is not the only algorithm with this schedulable utilization. In particular, *the schedulable utilization of the LST algorithm is also 1*. This follows straightforwardly from Theorem 4.3 which states that the LST algorithm is also optimal for scheduling independent, preemptable jobs on one processor.

Example of infeasible EDF schedules

When the relative deadlines of some tasks are less than their respective periods, the system may no longer be feasible, even when its total utilization is less than 1. As an example, the task with period 5 and execution time 2.3 in Figure 6–4 would not be schedulable if its relative deadline were 3 instead of 5.

We call the ratio of the execution time e_k of a task T_k to the minimum of its relative deadline D_k and period p_k the *density* δ_k of the task. In other words, the density of T_k is $e_k/\min(D_k, p_k)$. The sum of the densities of all tasks in a system is the *density* of the system and is denoted by Δ. When $D_i < p_i$ for some task T_i, $\Delta > U$. If the density of a system is larger than 1, the system may not be feasible. For example, this sum is larger than 1 for (2, 0.9) and (5, 2.3, 3), 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. We state this fact in the following theorem which generalizes Theorem 6.1; its proof is similar to the proof of Theorem 6.1 and is left to you as an exercise.

THEOREM 6.2. A system **T** of independent, preemptable tasks can be feasibly scheduled on one processor if its density is equal to or less than 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.

6.4 Optimality of the RM and DM algorithms

We assume that fixed-priority scheduling indexes the tasks in decreasing order of their priorities except where stated otherwise. In other words, the task T_i has a higher priority than the task T_k if $i < k$. By indexing the tasks in this manner, our discussion implicitly takes into consideration the scheduling algorithm. Sometimes, we refer to the priority of a task T_i as priority π_i. π_i's are positive integers1,2,...,n, 1 being the highest priority and n being the lowest priority. We denote the subset of tasks with equal or higher priority than T_i by $\mathbf{T}_i$ and its total utilization by $U_i = \sum_{k=1}^{i} u_k$.

Because they assign fixed priorities to tasks, fixed-priority algorithms cannot be optimal: Such an algorithm may fail to schedule some systems for which there are feasible schedules. To demonstrate this fact, we consider a system which consists of two tasks: $T_1 = (2, 1)$ and $T_2 = (5, 2.5)$. Since their total utilization is equal to 1, we know from Theorem 6.1 that the tasks are feasible. $J_{1,1}$ and $J_{1,2}$ can complete in time only if they have a higher priority than $J_{2,1}$. In other words, in the time interval (0,4],

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

While the RM algorithm is not optimal for tasks with arbitrary periods, it is optimal in the special case when the periodic tasks in the system are simply periodic and the deadlines of the tasks are no less than their respective periods. A system of periodic tasks is *simply periodic* if for every pair of tasks T_i and T_k in the system and $p_i < p_k$, p_k is an integer multiple of p_i.

THEOREM 6.3. A system of simply periodic, independent, preemptable tasks whose relative deadlines are equal to or larger than their periods is schedulable on one processor according to the RM algorithm if and only if its total utilization is equal to or less than 1.

To see why this theorem is true, we suppose for now that the tasks are in phase (i.e., the tasks have identical phases) and the processor never idles before the task T_i misses a deadline for the first time at t. t is an integer multiple of p_i. Because the tasks are simply periodic, t is also an integer multiple of the period p_k of every higher-priority task T_k, for $k = 1, 2, \ldots, i-1$. Hence the total time required to complete all the jobs with deadlines before and at t is equal to $\sum_{k=1}^{i}(e_k t/p_k)$, which is equal to t times the total utilization $U_i = \sum_{k=1}^{i} u_k$ of the i highest priority tasks. That T_i misses a deadline at t means that this demand for time exceeds t. In other words, $U_i > 1$.

Despite the fact that fixed-priority scheduling is not optimal in general, we may nevertheless choose to use this approach because it leads to a more predictable and stable system. For this reason, we want to know among all the fixed-priority algorithms, which one(s) is the best. The answer is that the DM algorithm is the optimal fixed-priority algorithm. Theorem 6.4 states this fact more precisely.

THEOREM 6.4. A system **T** of independent, preemptable periodic tasks that are in phase and have relative deadlines equal to or less than their respective periods can be feasibly scheduled on one processor according to the DM algorithm whenever it can be feasibly scheduled according to any fixed-priority algorithm.

This theorem is true because we can always transform a feasible fixed-priority schedule that is not a DM schedule into one that is. Specifically, suppose that a system has a feasible fixed-priority schedule that is not a DM schedule. We scan the tasks, starting from task T_1 with the shortest relative deadline in order of increasing relative deadlines. When we find two tasks T_i and $T_i + 1$ which are such that D_i is less than D_{i+1} but T_i has a lower priority than T_{i+1} according to this schedule, we switch the priorities of these two tasks and modify the schedule of the two tasks accordingly. After the switch, the priorities of the two tasks are assigned on the DM basis relative to one another. By repeating this process, we can transform the given schedule into a DM schedule. The step of showing that when the tasks are in phase, it is always possible to switch the priorities of tasks T_i and T_{i+1} and hence the time intervals in which jobs in these tasks are scheduled without leading to any missed deadline.

6.5 A schedulability test for fixed-priority task with short response times

In this section we confine our attention to 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. We will consider the general case where the response times may be larger than the periods in the next section. Since no system with total utilization greater than 1 is schedulable, we assume hereafter that the total utilization U is equal to or less than 1.

6.5.1 Critical Instants

The schedulability test uses as inputs the given sets $\{p_i\}$ and $\{e_i\}$ of periods and execution times of the tasks in **T** and checks one task T_i at a time to determine whether the response times of all its jobs are equal to or less than its relative deadline D_i. Because we cannot count on any relationship among the release times to hold, we must first identify the worst-case combination of release times of any job $J_{i,c}$ in T_i and all the jobs that have higher priorities than $J_{i,c}$. This combination is the worst because the response time of a job $J_{i,c}$ released under this condition is the largest possible for all combinations of release times. For this purpose, the notion of critical instant was introduced.

A critical instant of a task Ti is a time instant which is such that

1. the job in Ti released at the instant has the maximum response time of all jobs in Ti , if the response time of every job in Ti is equal to or less than the relative deadline Di of Ti , and
2. the response time of the job released at the instant is greater than Di if the response time of some jobs in Ti exceeds Di.

We call the response time of a job in Ti released at a critical instant the maximum (possible) response time of the task and denote it by Wi . The following theorem gives us the condition under which a critical instant of each task Ti occurs.

THEOREM 6.5. In a fixed-priority system where every job completes before the next job in the same task is released, a critical instant of any task T_i occurs when one of its job $J_{i,c}$ is released at the same time with a job in every higher-priority task, that is, $r_{i,c}=r_{k,l_k}$ for some l_k for every k=1,2,...,i−1.

Example

Figure 6–8 gives an illustrative example. Figure 6–8(a) shows an RM schedule of the three jobs, (2, 0.6), (2.5, 0.2), and (3, 1.2) when they are in phase. Time 0 is a critical instant of both lower-priority tasks. The response times of the jobs in (2.5, 0.2) are 0.8, 0.3, 0.2, 0.2, 0.8, and so on. These times never exceed the response time of the first job. Similarly, the response times of the jobs in (3, 1.2) are 2, 1.8, 2, 2, and so on, which never exceed 2, the response time of the first job in (3, 1.2). Figure 6–8(b) shows an RM schedule when the phase of the task with period 2.5 is one, while the phases of the other tasks are 0. We see that 6 is a critical instant of the two lower-priority tasks. The jobs of these tasks released at this instant have the maximum possible response times of 0.8 and 2, respectively.

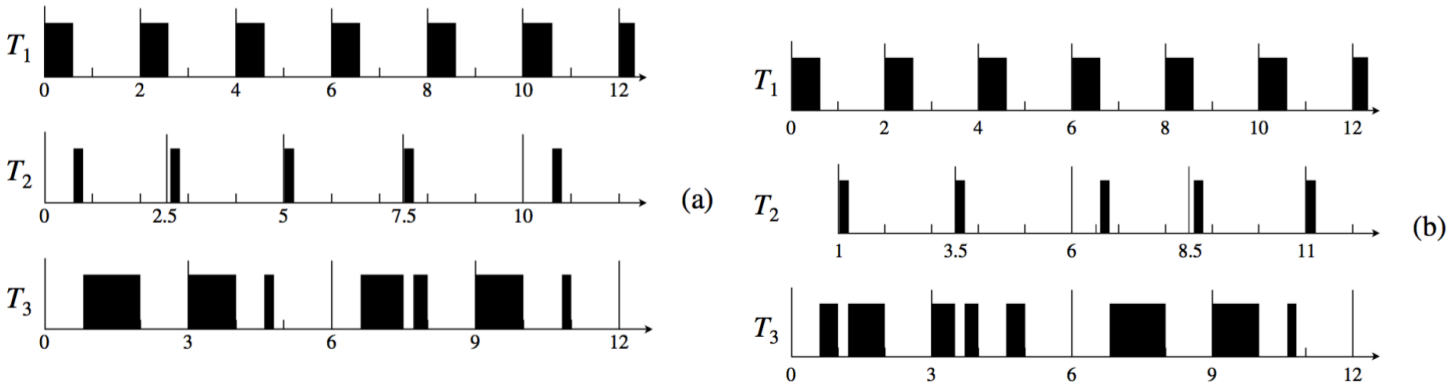


FIGURE 6–8 Example illustrating critical instants ($T_1 = (2, 0.6)$, $T_2 = (2.5, 0.2)$, $T_3 = (3, 1.2)$).

6.6 Schedulability test for fixed-priority task with arbitrary response times

This section describes a general time-demand analysis method 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. We assume here that this policy is used.

6.6.1 Busy intervals

We will use the term level-π_i busy interval. A *level-π_i busy interval* $(t_0,t]$ begins at an instant t_0 when

1. all jobs in $\mathbf{T}_i$ released before the instant have completed and
2. a job in $\mathbf{T}_i$ is released.

The interval ends at the first instant t after t_0 when all the jobs in $\mathbf{T}_i$ released since t_0 are complete. In other words, in the interval $(t_0, t]$, the processor is busy all the time executing jobs with priorities π_i or higher, all the jobs executed in the busy interval are released in the interval, and at the end of the interval there is no backlog of jobs to be executed afterwards. Hence, when computing the response times of jobs in $\mathbf{T}_i$, we can consider every level-π_i busy interval independently from other level-π_i busy intervals.

With a slight abuse of the term, we say that a level-π_i busy interval is *in phase* if the first jobs of all tasks that have priorities equal to or higher than priority π_i and are executed in this interval have the same release time. Otherwise, we say that the tasks have arbitrary phases in the interval.

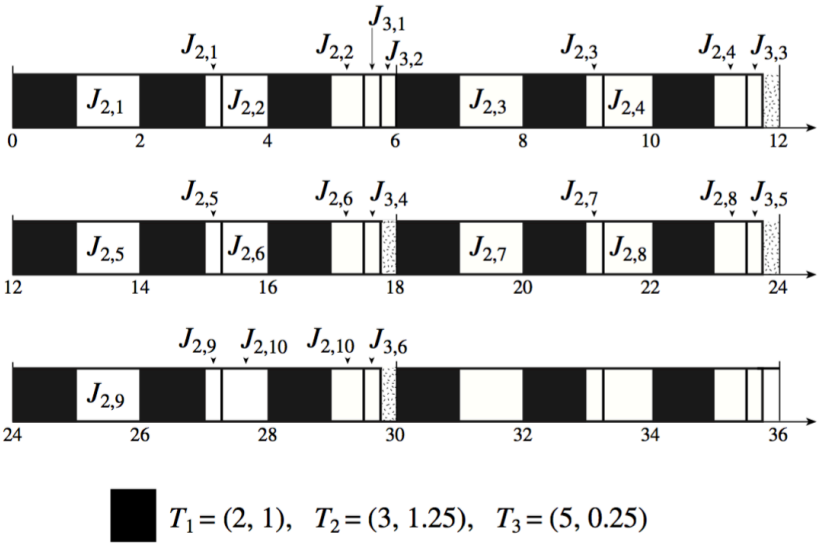


FIGURE 6–12 Example illustrating busy intervals.

Example

As an example, Figure 6–12 shows the schedule of three tasks $T_1 = (2,1)$, $T_2 = (3, 1.25)$, and $T_3 = (5, 0.25)$ in the first hyperperiod. The filled rectangles depict where jobs in T_1 are scheduled. The first busy intervals of all levels are in phase. The priorities of the tasks are $\pi_1 = 1$, $\pi_2 = 2$, and $\pi_3 = 3$, with 1 being the highest priority and 3 being the lowest priority. As expected, every level-1 busy interval always ends 1 unit time after it begins. For this system, all the level-2 busy intervals are in phase. They begin at times 0, 6, and so on which are the least common multiples of the periods of tasks T_1 and T_2. The lengths of these intervals are all equal to 5.5. Before time 5.5, there is at least one job of priority 1 or 2 ready for execution, but immediately after 5.5, there are none. Hence at 5.5, the first job in T_3 is scheduled. When this job completes at 5.75, the second job in T_3 is scheduled. At time 6, all the jobs released before time 6 are completed; hence, the first level-3 busy interval ends at this time. The second level-3 busy interval begins at time 6. This level-3 busy interval is not in phase since the release times of the first higher-priority jobs in this interval are 6, but the first job of T_3 in this interval is not released until time 10. The length of this level-3 busy interval is only 5.75. Similarly, all the subsequent level-3 busy intervals in the hyperperiod have arbitrary phases.

6.6.2 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. However, the first job $J_{i,1}$ may no longer have the largest response time among all jobs in T_i. (Example of two tasks: $T_1 = (70, 26)$ and $T_2 = (100, 52)$. Seven jobs of T_2 execute in the first level-2 busy interval. Their response times are 114, 102, 116, 104, 118, 106, and 94, respectively. The response times of both the third and fifth jobs in T_2 are larger than the response time of the first job.) Consequently, we must examine all the jobs of T_i that are executed in the first level-π_i busy interval. (Obviously, this busy interval is in phase when the tasks are in phase.) If the response times of all these jobs are no greater than the relative deadline of T_i, T_i is schedulable; otherwise, T_i may not be schedulable.

General Time-Demand Analysis Method

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

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

I. If the first job of every task in $\mathbf{T}_i$ completes by the end of the first period of the task, check whether the first job $J_{i,1}$ in T_i meets its deadline. T_i is schedulable if $J_{i,1}$ completes in time. Otherwise, T_i is not schedulable.

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

T_i is schedulable if all these jobs complete in time; otherwise T_i is not schedulable.

6.7 Sufficient schedulability conditions for the RM and DM algorithms

When we know the periods and execution times of all the tasks in an application system, we can use the schedulability test described in the last section to determine whether the system is schedulable according to the given fixed-priority algorithm. However, before we have com- pleted the design of the application system, some of these parameters may not be known. In fact, the design process invariably involves the trading of these parameters against each other. We may want to vary the periods and execution times of some tasks within some range of values for which the system remains feasible in order to improve some aspects of the system. For this purpose, it is desirable to have a schedulability condition similar to the ones given by Theorems 6.1 and 6.2 for the EDF and the LST algorithms. These schedulability conditions give us a flexible design guideline for the choices of the periods and execution times of tasks. The schedulable utilizations presented in this section give us

similar schedulability conditions for systems scheduled according to the RM or DM algorithms. An acceptance test based on such a schedulable utilization can decide whether to accept or reject a new periodic task in constant time. In contrast, the more accurate time-demand analysis test takes $O(nq_{n,1})$ time; moreover, the accurate test is less robust because its result is sensitive to the values of periods and execution times.

6.7.1 Schedulable Utilization of the RM Algorithm for Tasks with $D_i = p_i$

Specifically, the following theorem from gives us a schedulable utilization of the RM algorithm. We again focus on the 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

$$U_{RM}(n) = n\left(2^{\frac{1}{n}} - 1\right) \qquad 6.1$$

$U_{RM}(n)$ is the schedulable utilization of the RM algorithm when $D_i = p_i$ for all $1 \le k \le n$. Figure 6–14 shows its value as a function of the number n of tasks in the set. When n is equal to 2, $U_{rm}(n)$ is equal to 0.828. It approaches ln 2 (0.693), shown by the dashed line, for large n.

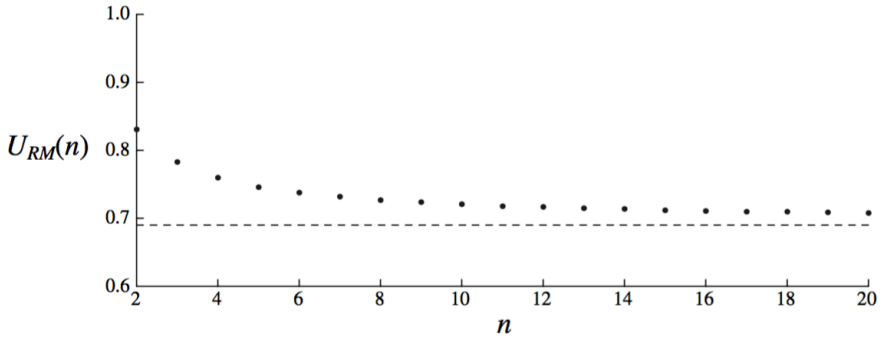


FIGURE 6–14 $U_{RM}(n)$ as a function n.

Specifically, $U(n) \le U_{RM}(n)$ is a sufficient schedulability condition for any system of n independent, preemptable tasks that have relative deadlines equal to their respective periods to be schedulable rate-monotonically. (We use the notation $U(n)$ in place of U in our subsequent discussion whenever we want to bring the number of tasks n to our attention.) As long as the total utilization of such a system satisfies this condition, it will never miss any deadline. In particular, we can reach this conclusion without considering the individual values of the phases, periods, and execution times.

Example

Consider the system **T** of 5 tasks: (1.0, 0.25), (1.25, 0.1), (1.5, 0.3), (1.75, 0.07), and (2.0, 0.1). Their utilizations are 0.25, 0.08, 0.2, 0.04, and 0.05. The total uti- lization is 0.62, which is less than 0.743, the value of $U_{RM}(5)$. Consequently, we can conclude that we can feasibly schedule **T** rate-monotonically. Suppose that the system is later enhanced. As a result, the tasks are modified, and the resultant tasks are (0.3, 1.3, 0.1), (1.0, 1.5, 0.3), (1.75, 0.1), (2.0, 0.1), and (7.0, 2.45). Since their total utilization is 0.737, which is still less than 0.743, we know for sure that the system remains schedulable. There is no need for us to do the more complex time-demand analysis to verify this fact. On the other hand, suppose that to make the above five-task system more modular, we divide the task with period 7.0 into three smaller tasks with periods 5, 6, and 7, while keeping the total utilization of the system at 0.737. We can no longer use this condition to assure ourselves that the system is schedulable because $U_{RM}(7)$ is 0.724 and the total utilization of the system exceeds this bound.

Since $U(n) \leq U_{RM}(n)$ is not a necessary condition, a system of tasks may nevertheless be schedulable even when its total utilization exceeds the schedulable bound. For example, the total utilization of the system with the four tasks (3, 1), (5, 1.5), (7, 1.25), and (9, 0.5) is 0.85, which is larger than $U_{RM}(4) = 0.757$. Earlier in Figure 6–9, we have shown by the time-demand analysis method that this system is schedulable according to the RM algorithm.

6.8 Practical factor

Thus far, we have assumed that every job is preemptable at any time; once a job is released, it never suspends itself and hence is ready for execution until it completes; scheduling and context-switch overhead is negligible; the scheduler is event-driven and acts immediately upon event occurrences; every task (or job) has a distinct priority; and every job in a fixed- priority system is scheduled at a constant priority. These assumptions are often not valid. We now discuss how these factors affect the schedulability of fixed-priority and dynamic-priority systems. In this discussion, we will need two new terms: blocking and priority inversion. A (ready) job J_i is *blocked* when it is prevented from executing by a lower-priority job: The lower-priority job executes while J_i waits. We say that a *priority inversion* occurs whenever a lower-priority job executes while some ready higher-priority job waits.

6.8.1 Nonpreemptability

There are many reasons for a job, or a portion of it, to be nonpreemptable. When a job is using a resource (e.g., a critical section) that must be used in a mutual exclusive manner, making the job nonpreemptable while it has the resource is one way to ensure mutual exclusion. Some system calls (or parts of the calls) are nonpreemptable for the same reason; consequently, a job that makes such a call is nonpreemptable during the execution of the call. Sometimes, preemption may be too costly. An example is disk scheduling. If reads and writes of files on a disk were preemptable, it would take an additional seek to resume the read or write of a file each time such an operation were preempted. Since seek and latency times are large on average, a significant amount of bandwidth might be lost if preemption were allowed. Consequently, disk scheduling is typically nonpreemptive.

Blocking Time Due to Nonpreemptivity. A higher-priority job that becomes ready when a nonpreemptable lower-priority job is executing is blocked until the nonpreemptable portion of the lower-priority job completes. The delay due to blocking may cause the higher-priority job to miss its deadline. Consequently, when we want to determine whether a task can meet all its deadlines, we must consider not only all the tasks that have higher priorities than it, but also the nonpreemptable portions of lower-priority tasks.

Example: As an example, we consider a system of three fixed-priority tasks: $T_1 = (\varepsilon,4,1)$, $T_2 = (\varepsilon, 5, 1.5)$, and $T_3 = (9, 2)$, where $\varepsilon > 0$ is very small compared with the other tim- ing parameters of the tasks. The total utilization is 0.77. If all the tasks are preemptable, they are schedulable according to both the schedulable utilization $U_{RM}(3) = 0.78$ and the time-demand analysis.

Now, suppose that T_3 is nonpreemptable. When the jobs $J_{1,1}$ and $J_{2,1}$ become ready at ε, the first job $J_{3,1}$ of T_3 is executing. Because $J_{3,1}$ cannot be preempted, the higher-priority jobs are blocked until time 2 when $J_{3,1}$ completes; there is a priority inversion during (ε, 2). $J_{2,1}$ is forced to wait for the lower-priority job $J_{3,1}$ in addition to the higher-priority task T_1. As a consequence, it misses its deadline at time 5. Because both the schedulable utilization $U_{RM}(n)$ and the time-demand functions do

not take into account the delay suffered by T_2 due to priority inversion, they would mislead us into believing that T_2 can meet all its deadlines.

6.8.2 Self-Suspension

While executing, a job may invoke some external operation, for example, an I/O operation or a remote procedure, that is executed on another processor. *Self-blocking* or *self-suspension* occurs when the job is suspended and waits until such an operation completes before its execution can continue. While it waits, the operating system removes it from the ready queue and places it in a blocked queue. We assume that the maximum amount of time each external operation takes to complete and, hence, the maximum duration of each self-suspension, is known. (This time can be an upper bound on the maximum response time of the external operation obtained by doing a time-demand analysis of all the tasks on the processor where the operation executes.)

In a special case, every job in a task T_i self-suspends for x units of time immediately after it is released (e.g., due to input data transmission). The job is ready for execution x time units after it is released. Hence, the time from the instant when the job is ready to its deadline is only $D_i - x$, not D_i . To determine whether the task T_i is schedulable, we use the shortened deadline $D_i - x$ in the schedulability test; there is no need to modify any of the methods otherwise.

6.8.3 Context Switches

We now confine our attention to job-level fixed-priority assignment, that is, each job is given a fixed priority throughout its execution. In such a system, each job preempts at most one job if there is no self-suspension. Hence, each job suffers at most one context switch when it starts execution and another context switch when it completes. We can account for the context-switch overhead in a schedulability test by including the time spent for the two context switches at the start and completion of each job as part of the execution time of the job. If the job is preempted and later resumed, the time spent for the two context switches is accounted for in the same way: Include the context-switch time in the execution time of the preempting job.

We let $C\ S$ denote the *context-switch time* of the system, that is, the maximum amount of time the system spends per context switch. $C\ S$ includes the time required to maintain the context of the jobs involved in the context switch, as well as the time spent by the scheduler to service the event interrupt that triggers the context switch and to carry out the scheduling action at the context switch. If no job ever self-suspends, we increase the execution time of every task T_i to $e_i + 2CS$, for $i = 1,2,...,n$ when the context-switch overhead is not negligible. If some job self-suspends, the job incurs two more context switches each time it self-suspends. Therefore, if each job in any task T_i can self-suspend a maximum of K_i times after its execution starts, we add $2(K_i + 1)C\ S$ to the execution time e_i .

6.8.4 Tick Scheduling

A way to implement the scheduler is to make it time-driven. By this we mean that the execution of the scheduler is triggered by a timer which is set to expire periodically. Scheduling decisions are made at these time instants, called *clock interrupts*. This method is called *tick scheduling* or time-based scheduling. We now focus on the case when the scheduler executes only at clock interrupts.

Tick scheduling introduces two additional factors that must be accounted for in schedulability analysis. First, the fact that a job is ready may not be noticed and acted upon by the scheduler until the next clock interrupt. The delayed action of the scheduler may delay the completion of the job. Second, a ready job that is yet to be noticed by the scheduler must be held somewhere other than the ready job queue. Let us call this place the pending (job) queue; it holds the jobs that have been released or

unblocked since the last clock interrupt. When the scheduler executes, it moves the jobs in the pending queue to the ready job queue and places them there in order of their priorities. Once in the ready queue, the jobs execute in priority order without intervention by the scheduler. The time the scheduler takes to move the jobs introduces additional scheduling overhead.

Summary: When the scheduler makes scheduling decisions only at clock interrupts that are p_0 units of time apart, a newly released or unblocked job must wait in the pending queue until the next clock interrupt. We need to consider the following two factors: the delay suffered by each job waiting to be noticed by the scheduler and the time taken by the scheduler to move each job from the pending queue to the ready queue.

6.8.5 Varying Priority in Fixed-Priority Systems

In general, each job in a task T_i may have more than one segment and the segments may have different priorities. There are many reasons for doing so. When tasks contend for resources, we sometimes raise the priority of a job segment during which the job holds some nonpreemptable resource in order to speed up the release of the resource.

As an example, the system of two tasks $T_1 = (2, 1)$ and $T_2 = (5, 2.5)$ is not schedulable on a fixed-priority basis. However, if each job in T_2 is divided into two segments whose execution times are 2.0 and 0.5, respectively, and if the first segment has a lower priority than T_1 but the second segment has a higher priority than T_1, then both tasks would be schedulable. (This "fixed-priority assignment" in fact emulates the EDF assignment.)

6.8.6 Schedulability Test of Hierarchically Scheduled Periodic Tasks

Two common hierarchical scheduling schemes are the priority-driven/round-robin scheme and fixed-time partitioning scheme. In a *priority-driven/round-robin* system, the scheduler schedules the subsystems in a priority-driven manner. The tasks in each subsystem are scheduled in a round-robin manner in the intervals assigned to the subsystem. According to the *fixed-time partitioning* scheme, the schedulers schedules the subsystems according to a cyclic schedule. The tasks in each subsystem are scheduled according to the scheduling algorithm chosen for the subsystem. We do not need any new method to determine the schedulability of a priority-driven/round-robin system.

7 Scheduling Aperiodic and Sporadic Jobs in Priority-Driven Systems

This chapter describes algorithms for scheduling aperiodic and sporadic jobs among periodic tasks in a priority-driven system.

7.1 Assumptions and approaches

As in Chapters 5 and 6, we assume that there is only one processor and the periodic tasks in the system are independent. Aperiodic and sporadic jobs are also independent of each other and of the periodic tasks. We assume that every job can be preempted at any time.

We do not make any assumptions on the interrelease-times and execution times of aperiodic jobs. We assume that 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. As we discussed in Chapter 5, in general, 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. This chapter focuses primarily on the first alternative.

7.1.1 Objectives, Correctness, and Optimality

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

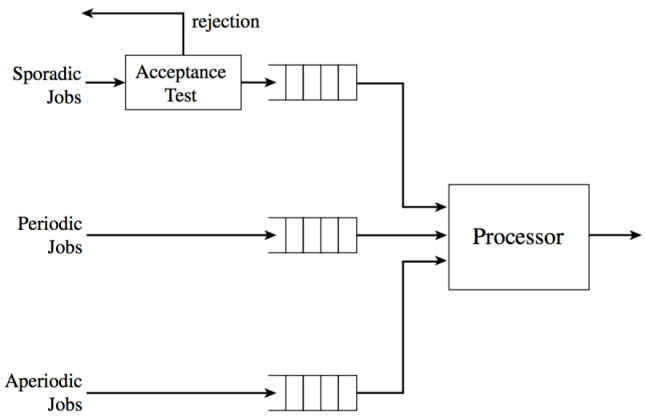


FIGURE 7–1 Priority queues maintained by the operating system.

For the sake of concreteness, we assume that the operating system maintains the priority queues shown in Figure 7–1. 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.

The algorithms described in this chapter determine when aperiodic or sporadic jobs are executed. We call them *aperiodic job and sporadic job scheduling algorithms*; they are solutions to the following problems:

- 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.

- 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.

Hereafter, by algorithm, we mean specifically an aperiodic job or sporadic job scheduling algorithm, except when it is stated otherwise. 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. We consider only correct algorithms.

Finally, we assume that the queueing discipline used to order aperiodic jobs among themselves is given. 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.

7.1.2 Some Terms

Periodic Server:

We call a task that behaves more or less like a periodic task and is created for the purpose of executing aperiodic jobs a *periodic server*. A periodic server (p_s , e_s) is defined partially by its period p_s and execution time e_s . The parameter e_s is called the *execution budget* (or simply *budget*) of the server. The ratio $u_s = e_s/p_s$ is the *size* of the server. At the beginning of each period, the budget of the server is set to e_s . We say that its budget is *replenished* (by e_s units) and call a time instant when the server budget is replenished a *replenishment time*.

Backlogged

We say that the periodic server is *backlogged* whenever the aperiodic job queue is nonempty and, hence, there is at least an aperiodic job to be executed by the server. The server is *idle* when the queue is empty. The server is *eligible* (i.e., ready) *for execution only when it is backlogged and has budget* (i.e., its budget is nonzero). When the server is eligible, the scheduler schedules it with the ready periodic tasks according to the algorithm used for scheduling periodic tasks as if the server is the periodic task (p_s,e_s). When the server is scheduled and executes aperiodic jobs, it *consumes* its budget at the rate of one per unit time. We say that the server budget becomes *exhausted* when the budget becomes zero.

Bandwidth Preserving Server

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.

7.2 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.

7.2.1 Operations of Deferrable Servers

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

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

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

We 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.

Example: Figure 7–3 gives an example. Figure7–3(a) shows that the deferrable server T_{DS} = (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.

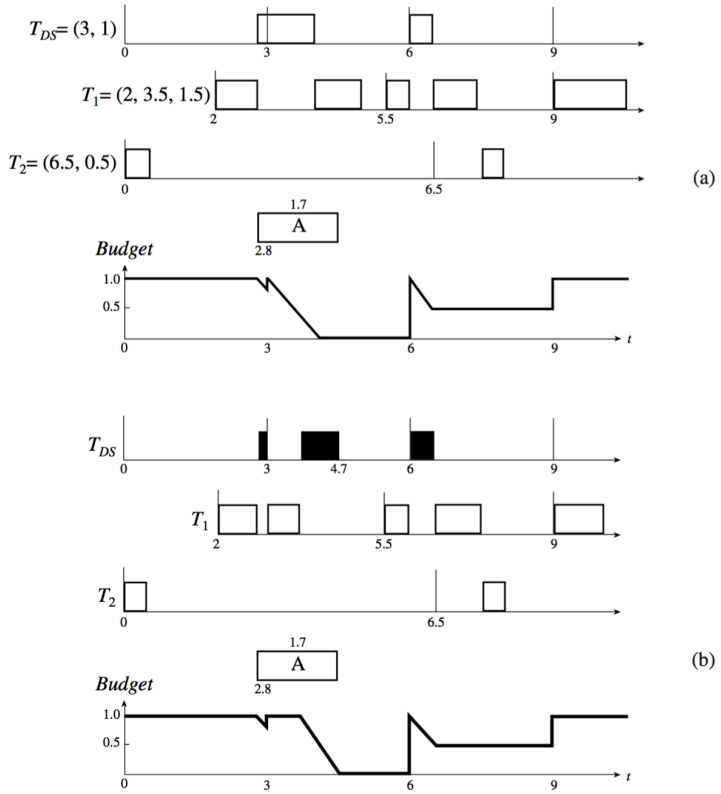


FIGURE 7–3 Example illustrating the operations of a deferrable server: (T_{DS} = (3, 1), T_1 = (2, 3.5, 1.5), and T_2 = (6.5, 0.5). (a) Rate-monotonic schedule. (b) EDF schedule.

Figure 7–3(b) shows the same periodic tasks and the deferrable server scheduled according to the EDF algorithm. At any time, the deadline of the server is equal to the next replenishment time.

1. At time 2.8, the deadline of the deferrable server is 3.0. Consequently, the deferrable server executes at the highest-priority beginning at this time.
2. At time 3.0, when the budget of the deferrable server is replenished, its deadline for consuming this new unit of budget is 6. Since the deadline of $J_{1,1}$ is sooner, this job has a higher priority. The deferrable server is preempted.
3. At time 3.7, $J_{1,1}$ completes. The deferrable server executes until time 4.7 when its budget is exhausted.
4. At time 6 when the server's budget is replenished, its deadline is 9, which is the same as the deadline of the job $J_{1,2}$. Hence, $J_{1,2}$ would have the same priority as the server. The figure shows that the tie is broken in favor of the server.

Again, the scheduler treats a deferrable server as a periodic task. You can see from the rules of the deferrable server algorithm, the algorithm is simple. The scheduling overhead of a deferrable server is no higher than that of a poller.

The responsiveness of the system can be further improved if we combine the use of a deferrable server with background execution. In other words, we also use a *background server*. This server is scheduled whenever the budget of the deferrable server has been exhausted and none of the periodic tasks is ready for execution. When the background server is scheduled, it also executes the aperiodic job at the head of the aperiodic job queue. With such a server, job A in Figure 7–3(a) would be executed from time 4.7 and completed by time 5.2, rather than 6.5, as shown in the figure.

7.3 Sporadic servers

We have just seen that a deferrable server may delay lower-priority tasks for more time than a period task with the same period and execution time. This section describes a class of bandwidth-preserving servers, called *sporadic servers*, that are designed to improve over a deferrable server in this respect. The consumption and replenishment rules of sporadic server algorithms ensure that each sporadic server with period p_s and budget e_s never demands more processor time than the periodic task (p_s , e_s) in any time interval. Consequently, we can treat the sporadic server exactly like the periodic task (p_s,e_s) when we check for the schedulability of the system. A system of periodic tasks containing a sporadic server may be schedulable while the same system containing a deferrable server with the same parameters is not.

With slight abuse of the notation, we will sometimes refer to a sporadic server with period p_s and budget e_s as the periodic task T_s = (p_s , e_s) and say that a server job is released when the server becomes eligible. Different kinds of sporadic servers differ in their consumption and replenishment rules. More complicated rules allow a server to preserve its budget for a longer time, replenish the budget more aggressively, and, in a deadline-driven system, execute at a higher priority. By using different rules, you can trade off the responsiveness of the server for the overhead in its implementation.

7.3.1 Sporadic Server in Fixed-Priority Systems

We assume that there is only one sporadic server in a fixed-priority system **T** of n independent, preemptable periodic tasks. The server has an arbitrary priority π_s . (If the server has the same priority as some periodic task, the tie is always broken in favor of the server.) We use $\mathbf{T}_H$ to denote the subset of periodic tasks that have higher priorities than the server. We say that the system **T** of periodic tasks

(or the higher-priority subsystem $\mathbf{T}_H$) idles when no job in $\mathbf{T}$ (or $\mathbf{T}_H$) is ready for execution; $\mathbf{T}$ (or $\mathbf{T}_H$) is busy otherwise. By definition, the higher-priority subsystem remains busy in any busy interval of $\mathbf{T}_H$. Finally, a *server busy interval* is a time interval which be- gins when an aperiodic job arrives at an empty aperiodic job queue and ends when the queue becomes empty again.

Since our focus here is on the consumption and replenishment rules of the server, we assume that we have chosen the parameters p_s and e_s and have validated that the periodic task (p_s , e_s) and the system $\mathbf{T}$ are schedulable according to the fixed-priority algorithm used by the system. When doing the schedulability test, we assume that the relative deadline for consuming the server budget is finite but arbitrary. In particular, we allow this deadline to be larger than p_s. During an interval when the aperiodic job queue is never empty, the server behaves like the periodic task (p_s , e_s) in which some jobs may take longer than one period to complete.

We state below the consumption and replenishment rules that define a simple sporadic server. In the statement, we use the following notations.

- t_r denotes the latest (actual) replenishment time.
- t_f denotes the first instant after t_r at which the server begins to execute.
- t_e denotes the latest *effective replenishment time*.
- At any time t, *BEGIN* is the beginning instant of the earliest busy interval among the latest contiguous sequence of busy intervals of the higher-priority subsystem $\mathbf{T}_H$ that started before t. (Two busy intervals are contiguous if the later one begins immediately after the earlier one ends.)
- *END* is the end of the latest busy interval in the above defined sequence if this interval ends before t and equal to infinity if the interval ends after t.

The scheduler sets t_r to the current time each time it replenishes the server's execution budget. When the server first begins to execute after a replenishment, the scheduler determines the latest effective replenishment time t_e based on the history of the system and sets the next replenishment time to $t_e + p_s$. (In other words, the next replenishment time is p_s units away from t_e, as if the budget was last replenished at t_e and hence the name effective replenishment time.)

Simple Sporadic Server. In its simpliest form, a sporadic server is governed by the following consumption and replenishment rules. We call such a server a *simple sporadic server*. A way to implement the server is to have the scheduler monitor the busy intervals of $\mathbf{T}_H$ and maintain information on *BEGIN* and *END*.

- ***Consumption Rules of Simple Fixed-Priority Sporadic Server*:** At any time t after t_r , the server's execution budget is consumed at the rate of 1 per unit time until the bud- get is exhausted when either one of the following two conditions is true. When these conditions are not true, the server holds its budget. **C1** The server is executing. **C2** The server has executed since t_r and $END < t$.
- ***Replenishment Rules of Simple Fixed-Priority Sporadic Server*:**

R1 Initially when the system begins execution and each time when the budget is replenished, the execution budget=e_s, and t_r =the current time.

R2 At time t_f, if $END=t_f$, t_e =max(t_r,*BEGIN*). If $END<t_f$, $t_e = t_f$. The next replenishment time is set at $t_e + p_s$.

R3 The next replenishment occurs at the next replenishment time, except under the following conditions. Under these conditions, replenishment is done at times stated below.

(a) If the next replenishment time $t_e + p_s$ is earlier than t_f, the budget is replenished as soon as it is exhausted.

(b) If the system **T** becomes idle before the next replenishment time $t_e + p_s$ and becomes busy again at t_b, the budget is replenished at $\min(t_e + p_s, t_b)$.

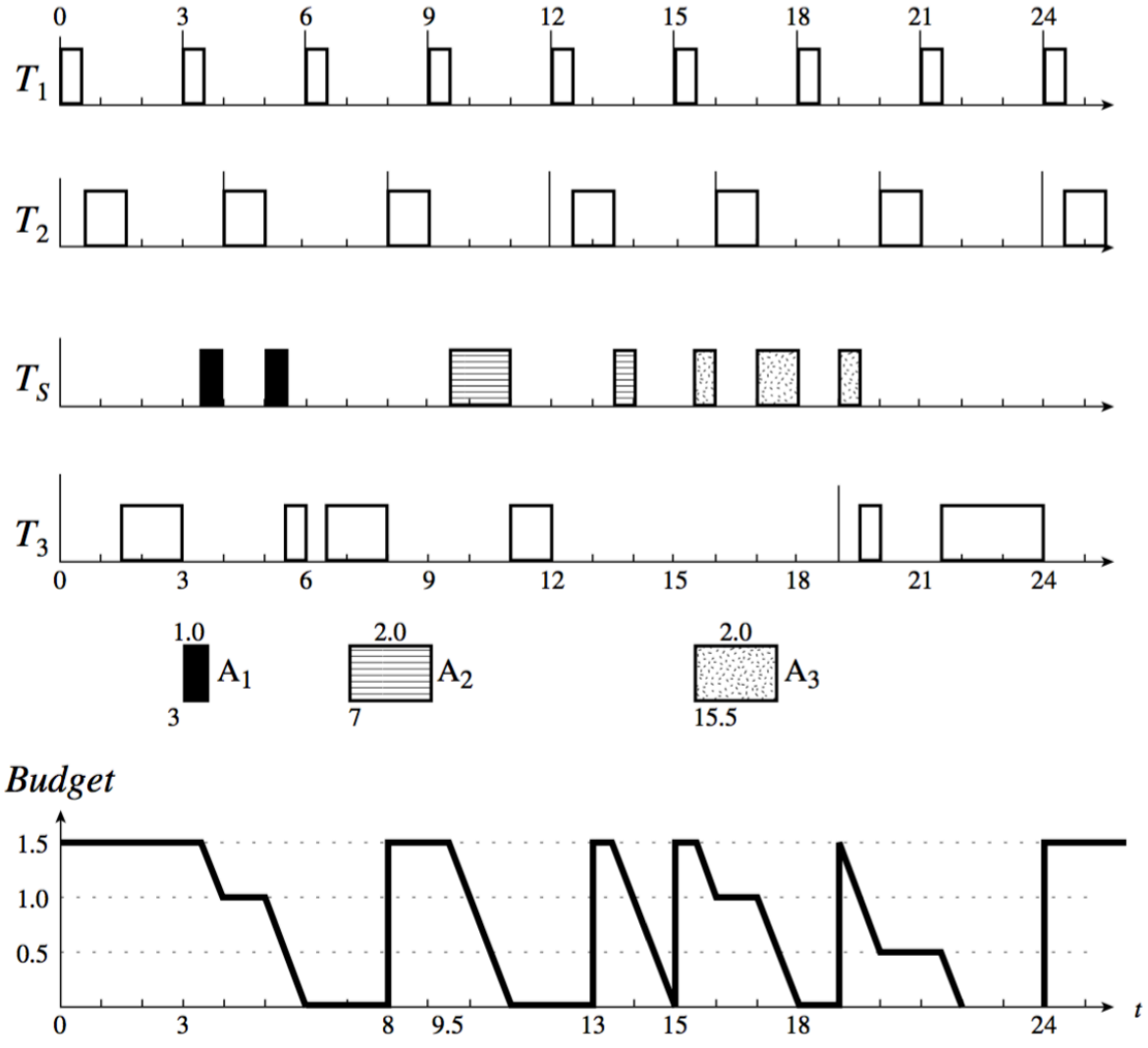


FIGURE 7–8 Example illustrating the operations of a simple sporadic server: $T_1 = (3, 0.5)$, $T_2 = (4, 1.0)$, $T_3 = (19, 4.5)$, $T_s = (5, 1.5)$.

Rules C1 and R1 are self-explanatory. Equivalently, rule C2 says that the server consumes its budget at any time t if it has executed since t_r but at t, it is suspended and the higher-priority subsystem $\mathbf{T}_H$ is idle. Rule R2 says that the next replenishment time is p_s units after t_r (i.e., the effective replenishment time t_e is t_r) only if the higher-priority subsystem $\mathbf{T}_H$ has been busy throughout the interval (t_r, t_f). Otherwise, t_e is later; it is the latest instant at which an equal or lower-priority task executes (or the system is idle) in (t_r, t_f).

Figure 7–8 gives an illustrative example. Initially the budget of the server (5, 1.5) is 1.5. It is scheduled rate-monotonically with three periodic tasks: $T_1 = (3, 0.5)$, $T_2 = (4, 1.0)$, and $T_3 = (19, 4.5)$. They are schedulable even when the aperiodic job queue is busy all the time.

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, t_r is 0, *BEGIN* is equal to 3, and *END* is equal to 3.5. According to rule R2, the effective replenishment time t_e 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 be- cause 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. t_e is equal to the latest replenishment time 8. Hence the next replenishment time is 13. The server execute 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.

7.4 Constant utilization, total bandwidth, and weighted fair-queuing servers

We now describe 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.

7.4.1 Schedulability of Sporadic Jobs in Deadline-Driven Systems

We know that a system of independent, preemptable periodic tasks whose relative deadlines are equal to or larger than their respective periods is schedulable according to the EDF algorithm if the total utilization of the tasks is equal to or less than 1. Also, if the execution times of some jobs in some task are smaller than the execution time of the task and the interrelease times between some jobs are larger than its period, the system is schedulable if it is schedulable according to this criterion.

We now study a more general sufficient schedulability condition for independent, preemptable sporadic jobs. This schedulability condition is in terms of densities of sporadic jobs. The ***density*** of a

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

THEOREM 7.4. 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.

The condition stated in Theorem 7.4 is not necessary: Sporadic jobs may be schedulable on the EDF basis when this condition is not satisfied. As an example, we consider three sporadic jobs each of which has a relative deadline of 2 and execution time of 1. They are released at time instants 0, 0.5, and 1.0. The total density of jobs is 1.5 in (1, 2], yet they are schedulable on the EDF basis.

A sporadic task S_i is a stream of sporadic jobs. Let $S_{i,j}$ denote the jth job in the task S_i (i.e.,the release time of $S_{i,j}$ is later than the release times of $S_{i,1}, S_{i,2}, \ldots, S_{i,j-1}$). Let $e_{i,j}$ denote the execution time of $S_{i,j}$, and $p_{i,j}$ denote the length of time between the re- lease times of $S_{i,j}$ and $S_{i,j+1}$. At the risk of abusing the term, we call $p_{i,j}$ the *period* of the sporadic job $S_{i,j}$ and the ratio $e_{i,j} / p_{i,j}$ the *instantaneous utilization* of the job. The *instantaneous utilization* $\tilde{u}_i$ of a sporadic task is the maximum of the instantaneous utilizations of all the jobs in this task (i.e., $\tilde{u}_i = \max_j(e_{i,j}/p_{i,j})$). As with execution times and periods of periodic tasks, we assume that the instantaneous utilization of a sporadic task is a known parameter of the task.

In a system of n sporadic tasks whose total instantaneous utilization is equal to or less than one, the total density of all active jobs is equal to or less than 1 at all times. Consequently, the following sufficient schedulability condition of sporadic tasks scheduled according to the EDF algorithm follows straightforwardly from Theorem 7.4.

COROLLARY 7.5. 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.

Because the utilization $u_i = \max_j (e_{i,j}) / \min_j (p_{i,j})$ of any task S_i is always larger than its instantaneous utilization $\tilde{u}_i = \max_j(e_{i,j}/p_{i,j})$, we have the following corollary.

COROLLARY7.6. 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.

This corollary gives us theoretical basis of the constant utilization server algorithm

7.4.2 Constant Utilization Server Algorithm

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 $\tilde{u}_s$; 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 and Replenishment Rules. 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*. Such a server never has any budget when there is no aperiodic job ready for execution. Hence the problem of dealing with chunks of budget never arises.

The budget of a basic constant utilization server is replenished and its deadline set according to the following rules. In the description of the rules, $\tilde{u}_s$ 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. We will return later to discuss how to remove this restriction.

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.

In short, a constant utilization server is always given enough budget to complete the job at the head of its queue each time its budget is replenished. Its deadline is set so that its instantaneous utilization is equal to $\tilde{u}_s$.

Example:

Figure 7–13 illustrates how a constant utilization server works. This system of periodic tasks and aperiodic jobs is essentially the same as the system in Figure 7–8. (For this system, the simple fixed-priority and deadline-driven sporadic servers happen to behave the same.) The only difference between them is that in Figure 7–13, aperiodic job A_2 arrives at time 6.9 instead of 7.0. The size of the constant utilization server is 0.25, slightly smaller than the size of the

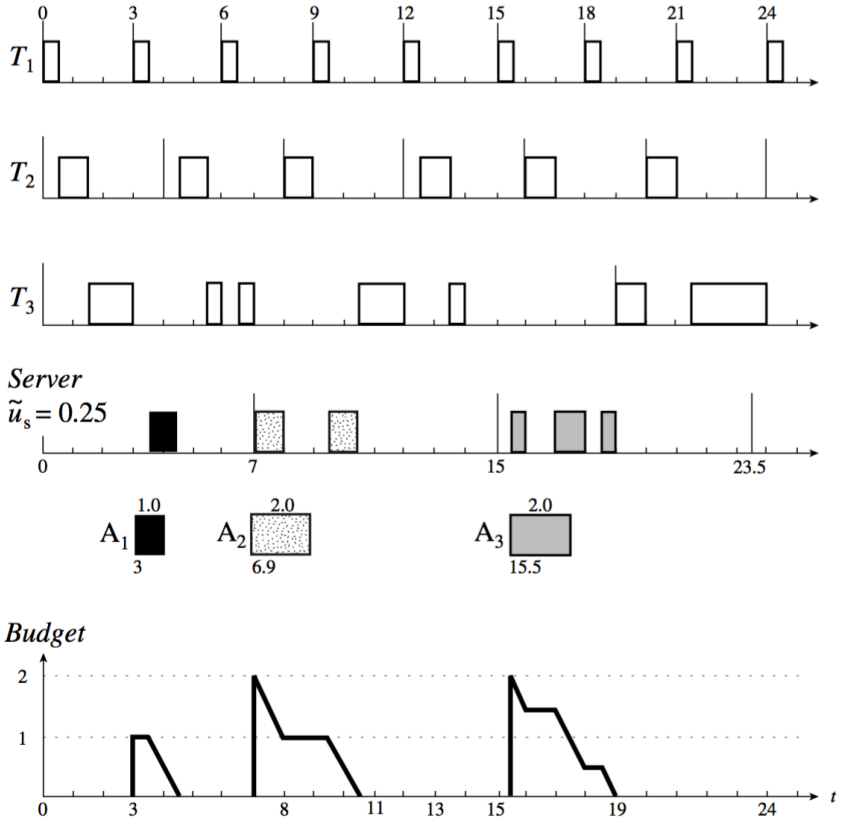


FIGURE 7–13 Example illustrating the operations of constant utilization server: $T_1 = (4, 0.5)$, $T_2 = (4, 1.0)$, $T_3 = (19, 4.5)$.

sporadic server in Figure 7–8.

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.
6. At time 15.5, A_3 arrives. At the time, the deadline of the server is 15. Hence according to rule R2b, its deadline is set at 23.5, and the server budget is set to 2.0. This allows the server to execute A_3 to completion at time 19.

We see that the constant utilization server completes each of the aperiodic jobs earlier than a simple sporadic server.

Scheduling Aperiodic Jobs with Unknown Execution Times. In the description of the constant utilization server algorithm, we assume that the execution times of aperiodic jobs become known upon their arrival. This restrictive assumption can be removed by modifying the replenishment rules of constant utilization (or total bandwidth) servers. One way is to give the server a fixed size budget e_s and fixed period $e_s/\tilde{u}_s$ just like sporadic and deferrable servers. Since the execution time of each aperiodic job can be determined after it is completed with little overhead, we can adjust the server deadline based on this knowledge upon the completion of the job. Specifically, when an aperiodic job with execution time e shorter than e_s completes, we reduce the current deadline of the server by $(e_s - e)/\tilde{u}_s$ units before replenishing the next e_s units of budget and setting the deadline accordingly. This action clearly can improve the performance of the server and does not make the instantaneous utilization of the server larger than $\tilde{u}_s$.

An aperiodic job with execution time larger than e_s is executed in more than one server period. We can treat the last chunk of such a job in the manner described above if the execution time of this chunk is less than e_s.

7.4.3 Total Bandwidth Server Algorithm

Let us return to the example in Figure 7–13. Suppose that A_3 were to arrive at time 14 instead. Since 14 is before the current server deadline 15, the scheduler must wait until time 15 to replenish the budget of the constant utilization server. A_3 waits in the interval from 14 to 15, while the processor idles! Clearly, one way to improve the responsiveness of the server is to replenish its budget at time 14. This is exactly what the total bandwidth server algorithm does.

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.

Comparing a total bandwidth server with a constant utilization server, we see that for a given set of aperiodic jobs and server size, both kinds of servers have the same sequence of deadlines, but the budget of a total bandwidth server may be replenished earlier than that of a constant utilization server. As long as a total bandwidth server is backlogged, it is always ready for execution.

COROLLARY 7.7. When a system of independent, preemptable periodic tasks is scheduled with one or more total bandwidth and constant utilization servers on the EDF basis, every periodic task and every server meets its deadlines if the sum of the total density of periodic tasks and the total size of all servers is no greater than 1.

7.4.4 Weighted Fair Queuing Server

For many applications, it is important that the scheduling algorithm be fair. The total bandwidth server algorithm is not fair; a server can be starved for an arbitrary amount of time. The weighted fair-queueing algorithm does not have this problem and can achieve the same worst-case response time as the total bandwidth server algorithm. The nonpreemptive version of this algorithm is also called packet-by-packet generalized processor sharing algorithm and is for scheduling transmissions of packets in switched networks.

- **Aim of the constant utilization and total bandwidth servers is to assign some fraction of processor capacity to a task**
- **When assigning capacity there is the issue of *fairness:***
  - A scheduling algorithm is *fair* within any particular time interval if the fraction of processor time in the interval attained by each backlogged server is proportional to the server size
    - Not only do all tasks meet their deadline, but they all make continual progress according to their share of the processor, no *starvation*
  - Constant utilization and total bandwidth servers are fair on the long term, but can diverge significantly from fair shares in the short term
    - Total bandwidth server partly by design, since it uses background time, but also has fairness issues when there is no spare background time

7.5 Slack stealing in deadline-driven systems

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.

Example

As an example, we consider again the system of two periodic tasks, $T_1 = (2.0, 3.5, 1.5)$ and $T_2 = (6.5, 0.5)$. 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 A_1 and A_2, 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 A_1 arrives at 2.8, the slack stealer resumes. Because the execution of the last 0.7 units of $J_{1,1}$ 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.

The key step in slack stealing is the computation to determine whether the system has any slack. A slack computation algorithm is *correct* if it never says that the system has slack when the system does not, since doing so may cause a periodic job to complete too late. An *optimal slack computation algorithm* gives the

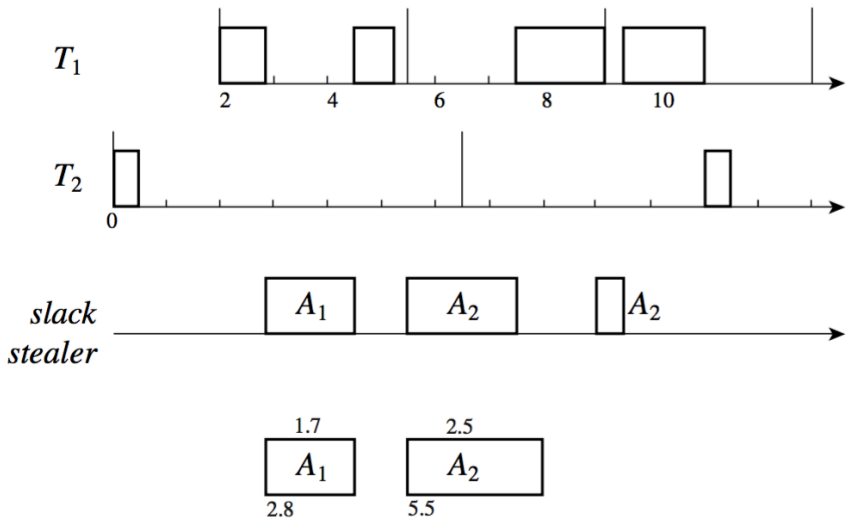


FIGURE 7–15 Example ilustrating a slack stealer in a deadline-driven system: $T_1 = (2.0, 3.5, 1.5)$ and $T_2 = (6.5, 0.5)$.

exact amount of slack the system has at the time of the computation; hence, it is correct. A correct slack computation algorithm that is not optimal gives a lower bound to the available slack.

Approaches to Slack Computation

There are two approaches to slack computation: *static* and *dynamic*. The method used to compute slack in a clock-driven system exemplifies the static approach. According to this approach, the initial slacks of all periodic jobs are computed off-line based on the given parameters of the periodic tasks. The scheduler only needs to update the slack information during run time to keep the information current, rather than having to generate the information from scratch. Consequently, the run-time overhead of the static approach is lower. A serious limitation of this approach is that the jitters in the release times of the periodic jobs must be negligibly small. We will show later that the slack computed based on the precomputed information may become incorrect when the actual release-times of periodic jobs differ from the release times used to generate the information.

According to the *dynamic* approach, the scheduler computes the amount of available slack during run time. When the interrelease times of periodic jobs vary widely, dynamic-slack computation is the only choice. The obvious disadvantage of the dynamic-slack computation is its high run-time overhead. However, it has many advantages. For example, the scheduler can integrate dynamic-slack computation with the reclaiming of processor time not used by periodic tasks and the handling of task overruns. This can be done by keeping track of the cumulative unused processor time and overrun time and taking these factors into account in slack computation.

7.6 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.

7.6.1 Optimality Criterion and Design Consideration

Example:

The system contains three periodic tasks: $T_1 = (3, 1)$, $T_2 = (4, 1)$, and $T_3 = (6, 1)$. 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 $J_{1,2}$ becomes ready, $J_{2,1}$ must wait for it to complete. As a consequence, $J_{2,1}$ must complete by time 3, although

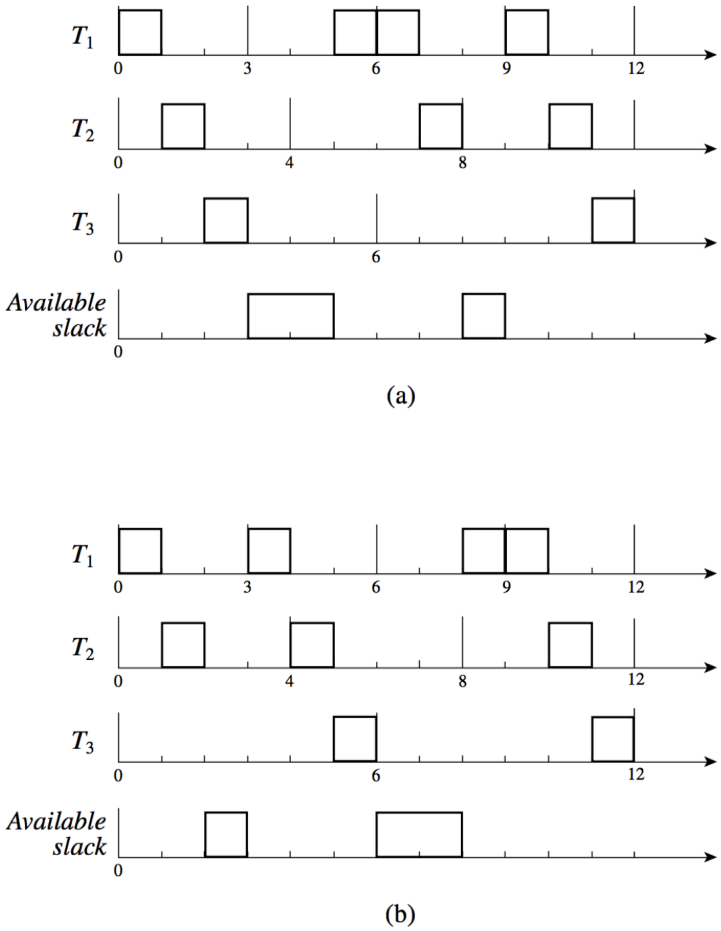


FIGURE 7–19 Example illustrating slack stealing in fixed-priority systems containing $T_1 = (3, 1)$, $T_2 = (4, 1)$, $T_3 = (6, 1)$.

its deadline is 4. In essence, 3 is the effective deadline of $J_{2,1}$, and its slack is determined by the effective deadline.

Figure 7–19(a) shows the schedule for the case when the 1 unit of slack is not used before time 3. At time 3, $J_{3,1}$ has already completed. $J_{1,2}$ and $J_{2,2}$ 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 7–19(b) shows the schedule for the other case: The 1 unit of slack is used before time 3. $J_{3,1}$ is not yet complete at time 3. Consequently, $J_{1,2}$ and $J_{2,2}$ must execute immediately after they are released, even though their deadlines are 6 and 8; otherwise, $J_{3,1}$ cannot complete in time. Under this condition, the system has no more slack until time 6.

Now suppose that aperiodic jobs A_1 and A_2 arrive at times 2 and 3, respectively, and their execution times are equal to 1. If the slack stealer executes A_1 immediately upon arrival, the job can be completed at 3 and have the minimum response time of 1. Since the system has no more slack until time 6, A_2 cannot be completed until time 7 (i.e., its response time is 4). On the other hand, if the scheduler waits until time 3 and then schedules the slack stealer, the aperiodic jobs are completed at times 4 and 5, respectively. A_2 now has the minimum response time of 2. This reduction is achieved at the expense of A_1, whose response time is no longer the minimum possible.

As a variation, suppose that A_1 is the only aperiodic job. We have just seen that when its execution time is 1, its response time is minimum if the job is executed immediately upon its arrival. However, if the execution time of the job were larger than 1 but no greater than 2, the scheduler would do better by waiting until time 3 to start its execution; the response time of the job would be 3 or less but would be more than 4 if the job were executed starting from time 2. If the execution time of A_1 were between 2 and 3, the scheduler would again do better by letting the job be executed immediately upon its arrival; the job would be completed by time 8.

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 is 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.

Because of (1), we use here a weaker optimality criterion: A slack-stealing algorithm for fixed-priority systems is *optimal in the weak sense* if it is correct and it minimizes the response time of the job at the head of the aperiodic job queue. Because of (2), an optimal slack-stealing algorithm for fixed-priority systems does not use available slack greedily. The algorithm is defined by a static-slack computation algorithm used to compute available slack and a scheduling algorithm for deciding when the available slack is used. It is optimal in the weak sense. To achieve optimality in the weak sense, it requires and uses the knowledge of the execution times of aperiodic jobs after their arrivals.

7.7 Scheduling of sporadic jobs

To schedule sporadic jobs that have hard deadlines among periodic tasks the key assumptions are that

1. the scheduler subjects each new sporadic job to an acceptance test and
2. the accepted sporadic jobs are ordered among themselves in the EDF order.

Acceptance Tests for Deadline-Driven Systems. The simplest acceptance test algorithm is based on Theorem 7.4, and it is applicable only when periodic tasks are scheduled on a EDF basis. It works as follows. In a system where the total density of periodic tasks is Δ, the scheduler accepts a new sporadic task $S(t,d,e)$ at the time t of its arrival if the total density of all the existing sporadic jobs whose deadlines are at or before the deadline d of the new job is less than or equal to $1 - \Delta - e/(d - t)$.

The simple acceptance test is not accurate. Its accuracy can be improved if the scheduler also computes and makes use of the slack of the system at each acceptance test. The dynamic-slack computation algorithm can be used for this purpose. When the periodic tasks have fixed release times, the acceptance test based on slack computation can be used. This test uses static-slack computation. Its time complexity is $O(n + N_s)$, where n and N_s are the number of periodic tasks and sporadic jobs, respectively, in the system.

Acceptance Tests for Fixed-Priority Systems. A simple and effective way to schedule sporadic jobs is to use a bandwidth-preserving server (p_s , e_s). The fact that the server is guaranteed to have e_s units of processor time every p_s units of time makes it possible for the scheduler to compute the latest completion times of the new sporadic job and existing sporadic jobs.

7.8 Real-time performance for jobs with soft timing constraints

For many applications, occasional missed deadlines are acceptable; their sporadic jobs have soft deadlines.

7.8.1 Traffic Models

Each sporadic task is a stream of sporadic jobs that have the same interrelease-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. In a system that provides each sporadic task with some kind of performance guarantee, the system subjects each new sporadic task to an acceptance test. 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 interarrival 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.

7.8.2 Performance of Bandwidth-Preserving Server Algorithms

Because no job of any accepted sporadic task is rejected by the scheduler, it is possible for a sporadic task to overload the processor. For this reason, the scheduler must provide firewall protection not only to tasks and jobs that have hard deadlines, but also to each existing sporadic task so the performance guarantee of the task will not be violated when other sporadic tasks misbehave. The bandwidth-preserving server algorithms described in earlier sections are designed to provide such protection and, therefore, are ideally suited for scheduling sporadic tasks with soft deadlines.

We focus here on systems which use a bandwidth-preserving server to execute each sporadic task. During an acceptance test, the scheduler can use an appropriate schedulability test to determine the maximum schedulable size of the server chosen to execute the new task. Hence, whether a new sporadic task is acceptable is reduced to the question of whether its required performance can be

achieved when its server has the maximum schedulable size. This question can be answered by analyzing the new sporadic task alone without regard to other tasks.

7.9 Tow-level scheme for integrated scheduling

This section describes 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. More importantly, the schedulability and real-time performance of each application can be determined independently of other applications executed on the same processor. By emulating an infinitesmally fine-grain time slicing scheme, the two-level scheme creates a slower virtual processor for each applications in the system.

7.9.1 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 and schedules the ready servers on the EDF basis. At the higher level, each server has a *server scheduler*; this scheduler schedules the jobs in the application executed by the server according to the algorithm chosen for the application.

Required Capability

In the description below, we use $\mathbf{T}_i$ for $i = 1, 2, \ldots, n$ 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 $\mathbf{T}_i$, we examine the tasks in it as if the application executes alone on a slower processor whose speed is a fraction s of the speed of the physical processor. In other words, we multiple the execution time of every job in the application by a factor $1/s > 1$ and use the product in place of the execution time in the schedulability test on the application. The minimum fraction of speed at which the application is schedulable is called its *required capacity* s_i. The required capacity of every application must be less than 1.

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 multi- ple the execution time of each task by 2 and the resultant tasks (2, 1.0) and (5, 2) are schedu- lable, but if the multiplication factor were bigger, the resultant tasks would not be schedulable. If these tasks are scheduled on the EDF basis, its required capacity is 0.45.

Predictable versus Nonpredictable Applications

In order to correctly maintain the server of an application that is scheduled according to a preemptive priority-driven algorithm, the OS scheduler needs an estimate of the occurrence time of every event of the application that may trigger a context switch within the application. Such events include the releases and completions of jobs and their requests and releases of resources. At any time t, the *next event* of application $\mathbf{T}_i$ is the one that would have the earliest occurrence time after t among all events of $\mathbf{T}_i$ if the application were to execute alone on a slow processor with speed s_i equal to its required capacity. We call an application that is scheduled according to a preemptive, priority-driven algorithm and contains aperiodic and sporadic tasks and/or periodic tasks with release-time jitters an *unpredictable application*. The reason for this name is that the OS scheduler needs an estimate of its

next event (occurrence) time at each replenishment time of its server, but its server scheduler cannot compute an accurate estimate.

All other types of applications are *predictable*. An application that contains only periodic tasks with fixed release times and known resource request times is predictable because its server scheduler can compute accurately the occurrence times of its future events. (In fact, for such an application, the event occurrence times can be computed a priori before the application requests to execute in the real-time mode and stored for use at run time.) All time-driven applications are predictable because scheduling decisions are triggered by clock interrupts which occur at known time instants.

7.9.2 Scheduling Predictable Applications

An important property of all types of predictable applications is that such an application is schedulable according to the two-level scheme if the size of its server is equal to its required capability and its server is schedulable. In contrast, a nonpredictable application requires a server of a size larger than than its required capability

7.9.3 Scheduling Nonpredictable Applications

In general, the actual release-times of some jobs in a nonpredictable application $\mathbf{T}_i$ may be unknown. It is impossible for its server scheduler to determine the next release-time t' of $\mathbf{T}_i$ precisely. Therefore, some priority inversion due to overreplenishment of the server budget is unavoidable. Fortunately, the bad effect on the schedulability of $\mathbf{T}_i$ can be compensated by making the size $\tilde{u}_i$ of the server larger than the required capacity s_i of the application.

8 Resources and Resource Access Control

In Section 3.1, we briefly mentioned the fact that in addition to a processor, each job may require some other resource in order to execute, but thus far we have ignored this requirement in order to focus on the problems in processor scheduling. We are now ready to take into account the resource requirements.

8.1 Assumptions on resources and their usage

We continue to focus on the case where the system contains only one processor. In addition, the system also contains ρ types of serially reusable resources named R_1 , R_2 , . . . , R_ρ . There are ν_i indistinguishable units of resource (of type) R_i, for $1 \leq i \leq \rho$. Serially reusable resources are typically granted (i.e., allocated) to jobs on a nonpreemptive basis and used in a *mutually exclusive* manner. In other words, when a unit of a resource R_i is granted to a job, this unit is no longer available to other jobs until the job frees the unit. Again, examples of such resources are mutexes, reader/writer locks, connection sockets, printers, and remote servers. A binary semaphore is a resource (type) that has only 1 unit while a counting semaphore has many units. A system containing five printers has 5 units of the printer resource. There is only 1 unit of an exclusive write-lock.

A resource that has an infinite number of units has no effect on the timing behavior of any job since every job can have the resource at any time; there is no need to include the resource in our model. Therefore, we lose no generality by assuming that every resource R_i has a finite number of units.

Some resources can be used by more than one job at the same time. We model such a resource as a resource type that has many units, each used in a mutually exclusive manner. For example, a file that can be read by at most ν users at the same time is modeled as a resource that has ν exclusive units. By modeling shared resources in this manner, we do not need to treat them differently.

8.1.1 Resource Conflicts and Blocking

Two jobs *conflict* with one another, or have a *resource conflict*, if some of the resources they require are of the same type. The jobs *contend* for a resource when one job requests a resource that the other job already has. The scheduler always denies a request if there are not enough free units of the resource to satisfy the request. Sometimes a scheduler may deny a request even when the requested resource units are free in order to prevent some undesirable execution behavior.

When the scheduler does not grant η_i units of resource R_i to the job requesting them, the lock request L (R_i , η_i) of the job fails (or is denied). When its lock request fails, the job is *blocked* and loses the processor. A blocked job is removed from the ready job queue. It stays blocked until the scheduler grants it η_i units of R_i for which the job is waiting. At that time, the job becomes *unblocked*, is moved backed to the ready job queue, and executes when it is scheduled.

The example in Figure 8–2 illustrates the effect of resource contentions. In this example, there are three jobs, J_1 , J_2 , and J_3 , whose feasible intervals are (6, 14], (2, 17] and (0, 18], respectively. The release time and deadline of each job are marked by the vertical bar on each of the time lines. The jobs are scheduled on the processor on the earliest-deadline-first basis. Hence, J_1 has the highest priority and J_3 the lowest. All three jobs require the resource R, which has only 1 unit. In particular, the critical sections in these jobs are [R;2], [R;4], and [R; 4], respectively. Below is a description of this schedule segment. The black boxes in Figure 8–2 show when the jobs are in their critical sections.

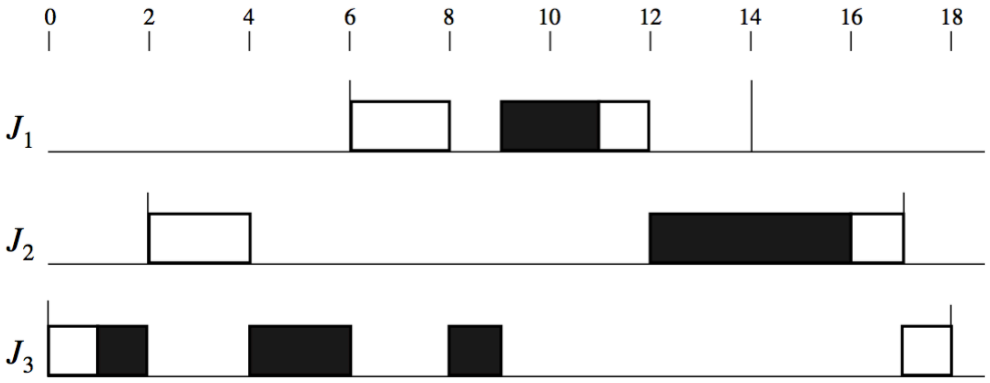


FIGURE 8–2 Example of job interaction due to resource contention.

1. At time 0, only J_3 is ready. It executes.
2. At time 1, J_3 is granted the resource R when it executes $L(R)$.
3. J_2 is released at time 2, preempts J_3, and begins to execute.
4. At time 4, J_2 tries to lock R. Because R is in use by J_3, this lock request fails. J_2 becomes blocked, and J_3 regains the processor and begins to execute.
5. At time 6, J_1 becomes ready, preempts J_3 and begins to execute.
6. J_1 executes until time 8 when it executes a $L(R)$ to request R. J_3 still has the resource. Consequently, J_1 becomes blocked. Only J_3 is ready for execution, and it again has the processor and executes.
7. The critical section of J_3 completes at time 9. The resource R becomes free when J_3 executes $U(R)$. Both J_1 and J_2 are waiting for it. The priority of the former is higher. Therefore, the resource and the processor are allocated to J_1, allowing it to resume execution.
8. J_1 releases the resource R at time 11. J_2 is unblocked. Since J_1 has the highest priority, it continues to execute.
9. J_1 completes at time 12. Since J_2 is no longer blocked and has a higher priority than J_3, it has the processor, holds the resource, and begins to execute. When it completes at time 17, J_3 resumes and executes until completion at 18.

This example illustrates how resource contention can delay the completion of higher-priority jobs. It is easy to see that if J_1 and J_2 do not require the resource, they can complete by times 11 and 14, respectively.

8.2 Effects of resources contention and resource access control

A *resource access-control protocol*, or simply an *access-control protocol*, is a set of rules that govern

1. when and under what conditions each request for resource is granted and
2. how jobs requiring resources are scheduled.

8.2.1 Priority Inversion, Timing Anomalies, and Deadlock

Priority inversion can occur when the execution of some jobs or portions of jobs is nonpreemptable. Resource contentions among jobs can also cause priority inversion. Because resources are allocated to jobs on a nonpreemptive basis, a higher-priority job can be blocked by a lower-priority job if the jobs conflict, even when the execution of both jobs is preemptable.

In the example in Figure 8–2, the lowest priority job J_3 first blocks J_2 and then blocks J_1 while it holds the resource R. As a result, priority inversion occurs in intervals (4, 6] and (8, 9].

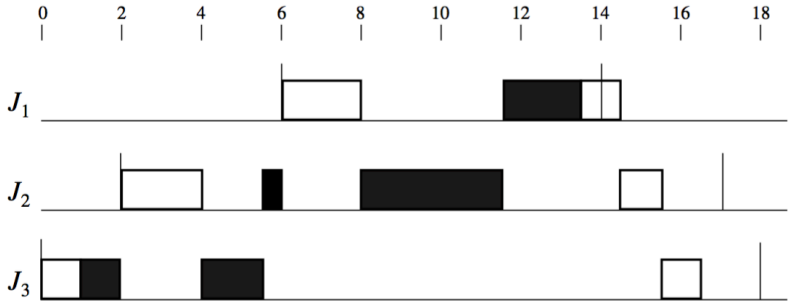


FIGURE 8–3 Example illustrating timing anomaly.

When priority inversion occurs, timing anomalies invariably follow. Figure 8–3 gives an example. In this example, there are three jobs, J_1 , J_2 , and J_3 , whose feasible intervals are (6, 14], (2, 17] and (0, 18], respectively. The release time and deadline of each job are marked by the vertical bar on each of the time lines. The jobs are scheduled on the processor on the earliest-deadline-first basis. Hence, J_1 has the highest priority and J_3 the lowest. All three jobs require the resource R, which has only 1 unit. In particular, the critical sections in these jobs are [R;2], [R;4], and [R; 2.5], respectively.

In other words, the execution time of the critical section in J_3 is shortened by 1.5. Intuitively a consequence of this reduction in J_3's execution time, all jobs should complete sooner. Indeed, this reduction does allow jobs J_2 and J_3 to complete sooner. Unfortunately, rather than meeting its deadline at 14, J_1 misses its deadline because it does not complete until 14.5.

More seriously, without good resource access control, the duration of a priority inversion can be unbounded. The example in Figure 8–4 illustrates this fact. Here, jobs J_1 and J_3 have the highest priority and lowest priority, respectively. At time 0, J_3 becomes ready and executes. It acquires the resource R shortly afterwards and continues to execute. After R is allocated to J_3, J_1 becomes ready. It preempts J_3 and executes until it requests resource R at time 3. Because the resource is in use, J_1 becomes blocked, and a priority inversion begins. While J_3 is holding the resource and executes, a job J_2 with a priority higher than J_3 but lower than J_1 is released. Moreover, J_2 does not require the resource R. This job preempts J_3 and executes to completion. Thus, J_2 lengthens the duration of this priority inversion. In this situation, the priority inversion is said to be uncontrolled. There can be an arbitrary number of jobs with priorities lower than J_1 and higher than J_3 released in the meantime. They can further lengthen the duration of the priority inversion. Indeed, when priority inversion is uncontrolled, a job can be blocked for an infinitely long time.

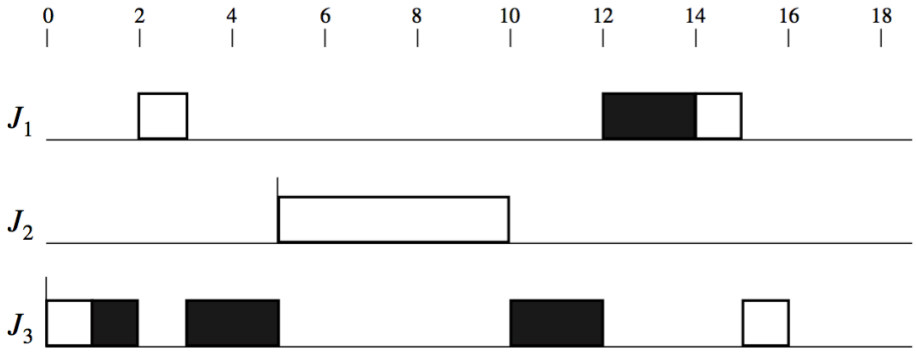


FIGURE 8–4 Uncontrolled priority inversion.

Nonpreemptivity of resource allocation can also lead to deadlocks. The classic example is one where there are two jobs that both require resources X and Y . The jobs are in deadlock when one of them holds X and requests for Y, while the other holds Y and requests for X. The conditions that allow this circular wait of jobs for each other (i.e., a deadlock) to occur are well-known.

From these examples, we see that no resource access-control protocol can eliminate the priority inversion and anomalous behavior caused by resource contention. A more realistic goal of such a protocol is that it keeps the delays thus incurred as short as possible. For this reason, a criterion we use to measure the performance of a resource access-control protocol is the blocking time of each job. A

good resource access-control protocol should control priority inversion and prevent deadlock and, thus, keep the blocking time of every job bounded from the above.

8.3 Nonpreemptive critical sections

The simplest way to control access of resources is to schedule all critical sections on the processor nonpreemptively. (In other words, when a job requests a resource, it is always allocated the resource. When a job holds any resource, it executes at a priority higher than the priorities of all jobs.) This protocol is called the *Nonpreemptive Critical Section* (*NPCS*) *protocol*. Because no job is ever preempted when it holds any resource, deadlock can never occur.

Take the jobs in Figure 8–4 for example. Figure 8–7(a) shows the schedule of these jobs when their critical sections are scheduled nonpreemptively on the processor. According to this schedule, J_1 is forced to wait for J_3 when J_3 holds the resource. However, as soon as J_3 releases the resource, J_1 becomes unblocked and executes to completion. Because J_1 is not delayed by J_2, it completes at time 10, rather than 15 according to the schedule in Figure 8–4.

In general, uncontrolled priority inversion illustrated by Figure 8–4 can never occur. The reason is that a job J_h can be blocked only if it is released when some lower-priority job is in a critical section. If it is blocked, once the blocking critical section completes, all resources are free. No lower-priority job can get the processor and acquire any resource until J_h completes. Hence, J_h can be blocked only once, and its blocking time due to resource conflict is at most equal to the maximum execution time of the critical sections of all lower-priority jobs.

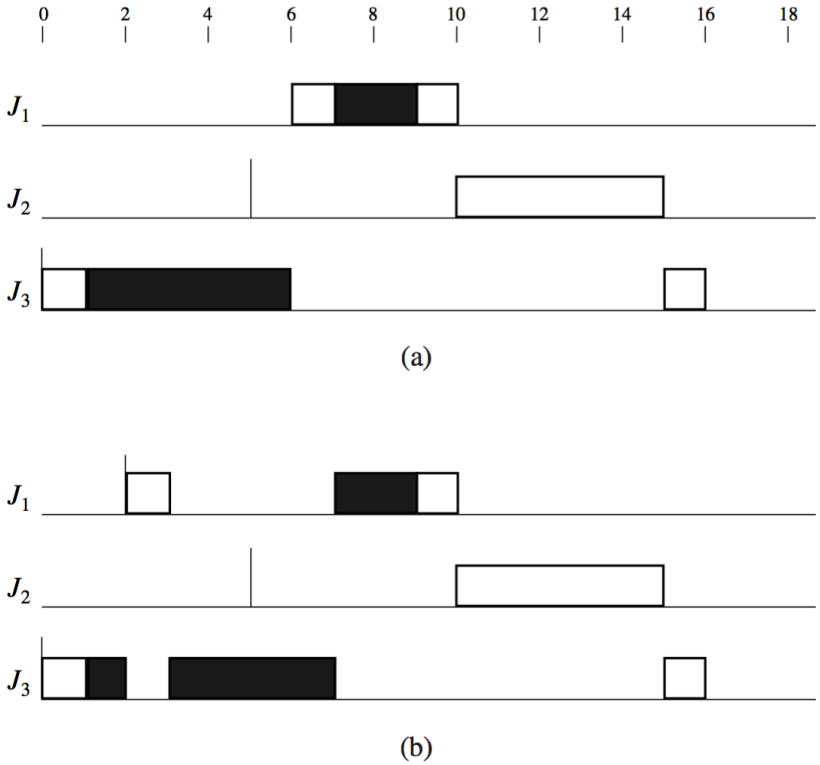


FIGURE 8–7 Example to illustrate two simple protocols. (a) Controlling priority inversion by disallowing preemption of critical section. (b) Controlling priority inversion by using priority inheritance.

Specifically, the blocking time $b_i(rc)$ due to resource conflict of a periodic task T_i in a fixed-priority system of n periodic tasks is equal to

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

when the tasks are indexed in order of nonincreasing priority. In a system where periodic tasks are scheduled on the EDF basis, a job in task T_i with relative deadline D_i can be blocked only by jobs in tasks with relative deadlines longer than D_i. Therefore, the blocking time $b_i(rc)$ of T_i is again given by above equation if we index the periodic tasks according to their relative deadlines so that $i < j$ if $D_i < D_j$.

As an example, suppose that the tasks in the system in Figure 8–6(b) are scheduled on a fixed-priority basis. The blocking time $b_1(rc)$ of the highest priority task T_1 is 8, the execution time of the (outermost) critical section of T_3. Similarly, $b_2(rc)$ is 8, while $b_3(rc)$ is 2, the execution time of the critical section of T_4. No task blocks T_4 since it has the lowest priority. Suppose that the relative deadlines of the tasks are $D_1 < D_2 < D_3 < D_4$ and the tasks are scheduled on an EDF basis. Then the blocking times $b_i(r\,c)$ for $i = 1, 2, 3$, and 4 are also 8, 8, 2, and 0, respectively.

The most important advantage of the NPCS protocol is its simplicity, especially when the numbers of resource units are arbitrary. The protocol does not need any prior knowledge about resource requirements of jobs. It is simple to implement and can be used in both fixed-priority and dynamic-priority systems. It is clearly a good protocol when all the critical sections are short and when most of the jobs conflict with each other.

An obvious shortcoming of this protocol is that every job can be blocked by every lower-priority job that requires some resource even when there is no resource conflict between them. When the resource requirements of all jobs are known, an improvement is to let a job holding any resource execute at the highest priority of all jobs requiring the resource.

8.4 Basic priority-inheritance protocol

The priority-inheritance protocol is a simple protocol. It works with any preemptive, priority-driven scheduling algorithm. Like the NPCS protocol, it does not require prior knowledge on resource requirements of jobs. The priority-inheritance protocol does not prevent deadlock. When there is no deadlock (i.e., when some other method is used to prevent deadlock), the protocol ensures that no job is ever blocked for an indefinitely long time because uncontrolled priority inversion cannot occur.

8.4.1 Definition of Basic Priority-Inheritance Protocol

In the definition of this protocol we call the priority that is assigned to a job according to the scheduling algorithm 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 may vary with time. In particular, the current priority $\pi_l(t)$ of a job J_l may be raised to the higher priority $\pi_h(t)$ of another job J_h. When this happens, we say that the lower-priority job J_l *inherits* the priority of the higher priority job J_h and that J_l executes at its *inherited priority* $\pi_h(t)$.

In its simplest form, the *priority-inheritance protocol* is defined by the following rules. These rules govern the ways current priorities of jobs are set and jobs are scheduled when some of them contend for resources. Again, this version assumes that every resource has only 1 unit.

Rules of the Basic Priority-Inheritance Protocol

1. ***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 rule 3.

2. ***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.
3. ***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 acquires the resource R.

According to this protocol, a job J is denied a resource only when the resource requested by it is held by another job. At time t when it requests the resource, J has the highest priority among all ready jobs. The current priority $\pi_l(t)$ of the job J_l directly blocking J is never higher than the priority $\pi(t)$ of J. Rule 3 relies on this fact.

The simple example in Figure 8–7(b) illustrates how priority inheritance affects the way jobs are scheduled and executed. The three jobs in this figure are the same as the ones in Figure 8–7(a). When J_1 requests resource R and becomes blocked by J_3 at time 3, job J_3 inherits the priority π_1 of J_1. When job J_2 becomes ready at 5, it cannot preempt J_3 because its priority π_2 is lower than the inherited priority π_1 of J_3. As a consequence, J_3 completes its critical section as soon as possible. In this way, the protocol ensures that the duration of priority inversion is never longer than the duration of an outermost critical section each time a job is blocked.

Figure 8–8 gives a more complex example. In this example, there are five jobs and two resources *Black* and *Shaded*. The parameters of the jobs and their critical sections are listed in part (a). As usual, jobs are indexed in decreasing order of their priorities: The priority π_i of J_i is i, and the smaller the integer, the higher the priority. In the schedule in part (b) of this figure, black boxes show the critical sections when the jobs are holding *Black*. Shaded boxes show the critical sections when the jobs are holding *Shaded*.

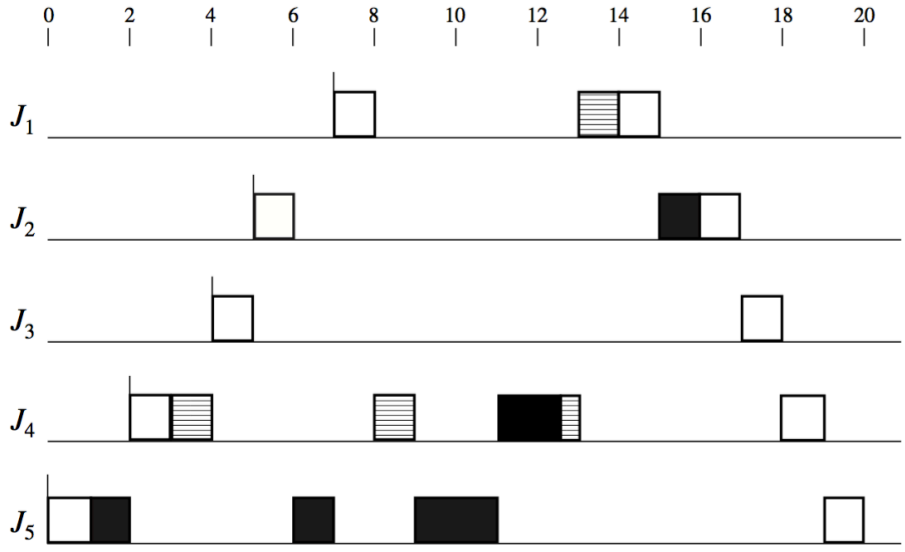
1. At time 0, job J_5 becomes ready and executes at its assigned priority 5. At time 1, it is granted the resource *Black*.
2. At time 2, J_4 is released. It preempts J_5 and starts to execute.
3. At time 3, J_4 requests *Shaded*. *Shaded*, being free, is granted to the job. The job continues to execute.
4. At time 4, J_3 is released and preempts J_4. At time 5, J_2 is released and preempts J_3.
5. At time 6, J_2 executes L(*Black*) to request *Black*; L(*Black*) fails because *Black* is in use by J_5. J_2 is now directly blocked by J_5. According to rule 3, J_5 inherits the priority 2 of J_2. Because J_5's priority is now the highest among all ready jobs, J_5 starts to execute.
6. J_1 is released at time 7. Having the highest priority 1, it preempts J_5 and starts to execute.
7. At time 8, J_1 executes L(*Shaded*), which fails, and becomes blocked. Since J_4 has *Shaded* at the time, it directly blocks J_1 and, consequently, inherits J_1's priority 1. J_4 now has the highest priority among the ready jobs J_3, J_4, and J_5. Therefore, it starts to execute.
8. At time 9, J_4 requests the resource *Black* and becomes directly blocked by J_5. At this time the current priority of J_4 is 1, the priority it has inherited from J_1 since time 8. Therefore, J_5 inherits priority 1 and begins to execute.
9. At time 11, J_5 releases the resource *Black*. Its priority returns to 5, which was its priority when it acquired *Black*. The job with the highest priority among all unblocked jobs is J_4.

Consequently, J_4 enters its inner critical section and proceeds to complete this and the outer critical section.

10. At time 13, J_4 releases *Shaded*. The job no longer holds any resource; its priority re- turns 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.

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*; 4 [*Black*; 1.5]]
J_5	0	6	5	[*Black*; 4]

(a)



(b)

FIGURE 8–8 Example illustrating transitive inheritance of priority inheritance. (a) Parameters of jobs. (b) Schedule under priority inheritance.

8.5 Basic priority- ceiling protocol

The *priority-ceiling protocol* extends the priority-inheritance protocol to prevent deadlocks and to further reduce the blocking time. This protocol makes two key assumptions:

1. The assigned priorities of all jobs are fixed.
2. The resources required by all jobs are known a priori before the execution of any job begins.

To define the protocol, we need two additional terms. The protocol makes use of a parameter, called priority ceiling, of every resource. The *priority ceiling* of any resource R_i is the highest priority of all

the jobs that require R_i and is denoted by $\prod(R_i)$. For example, the priority ceiling (*Black*) of the resource *Black* in the example in Figure 8–8 is 2 because J_2 is the highest priority job among the jobs requiring it. Similarly, (*Shaded*) is 1. Because of assumption 2, the priority ceilings of all resources are known a priori. We note that if the resource access control protocol includes the priority-inheritance rule, then a job can inherit a priority as high as x during its execution if it requires a resource with priority ceiling x.

At any time t, the *current priority ceiling* (or simply the *ceiling*) $\prod'(t)$ 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. If all the resources are free at the time, the current ceiling $\prod'(t)$ is equal to Ω, a nonexisting priority level that is lower than the lowest priority of all jobs. As an example, we again look at the system in Figure 8–8. In the interval [0, 1) when both resources in the system are free, the current ceiling of the system is equal to Ω, lower than 5, the priority of the lowest priority job J_5. In (1,3], *Black* is held by J_5; hence, the current ceiling of the system is 2. In (3, 13] when *Shaded* is also in use, the current ceiling of the system is 1, and so it is in (13, 14].

8.5.1 Definition of the Basic Priority-Ceiling Protocol

We now define the priority-ceiling protocol for the case when there is only 1 unit of every resource.

Rules of Basic Priority-Ceiling Protocol

1. *Scheduling Rule*:
   a. 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 rule 3.
   b. Every ready job J is scheduled preemptively and in a priority-driven manner at its current priority $\pi(t)$.
2. *Allocation Rule*: Whenever a job J requests a resource R at time t, one of the following two conditions occurs:
   a. R is held by another job. J 's request fails and J becomes blocked.
   b. R is free.
      i. If J's priority $\pi(t)$ is higher than the current priority ceiling $\prod'(t)$, R is allocated to J.
      ii. If J's priority $\pi(t)$ is not higher than the ceiling $\prod'(t)$ of the system, R is allocated to J only if J is the job holding the resource(s) whose priority ceiling is equal to $\prod'(t)$; otherwise, J 's request is denied, and J becomes blocked.
3. *Priority-Inheritance Rule*: When J becomes blocked, the job J_l which blocks J inherits the current priority $\pi(t)$ of J. J_l executes at its inherited priority until the time when it releases every resource whose priority ceiling is equal to or higher than $\pi(t)$; at that time, the priority of J_l returns to its priority $\pi_l(t')$ at the time t' when it was granted the resource(s).

We note that (ii) in rule 2 assumes that only one job holds all the resources with priority ceiling equal to $\prod'(t)$. Similarly, rule 3 assumes that only one job is responsible for J 's request being denied, because it holds either the requested resource or a resource with priority ceiling $\prod'(t)$.

Figure 8–10 shows the schedule of the system of jobs whose parameters are listed in Figure 8–8(a) when their accesses to resources are controlled by the priority-ceiling protocol. As stated earlier, the priority ceilings of the resources *Black* and *Shaded* are 2 and 1, respectively.

1. In the interval $(0, 3]$, this schedule is the same as the schedule shown in Figure 8–8, which is produced under the basic priority-inheritance protocol. In particular, the ceiling of the system at time 1 is Ω. When J_5 requests *Black*, it is allocated the resource according to (i) in part (b) of rule 2. After *Black* is allocated, the ceiling of the system is raised to 2, the priority ceiling of *Black*.
2. At time 3, J_4 requests *Shaded*. *Shaded* is free; however, because the ceiling $\hat{\Pi}(3)$ $(= 2)$ of the system is higher than the priority of J_4, J_4's request is denied according to (ii) in part (b) of rule 2. J_4 is blocked, and J_5 inherits J_4's priority and executes at priority 4.
3. At time 4, J_3 preempts J_5, and at time 5, J_2 preempts J_3. At time 6, J_2 requests *Black* and becomes directly blocked by J_5. Consequently, J_5 inherits the priority 2; it executes until J_1 becomes ready and preempts it. During all this time, the ceiling of the system remains at 2.
4. When J_1 requests *Shaded* at time 8, its priority is higher than the ceiling of the system. Hence, its request is granted according to (i) in part (b) of rule 2, allowing it to enter its critical section and complete by the time 10. At time 10, J_3 and J_5 are ready. The latter has a higher priority (i.e., 2); it resumes.
5. At 11, when J_5 releases *Black*, its priority returns to 5, and the ceiling of the system drops to Ω. J_2 becomes unblocked, is allocated *Black* [according to (i) in part (b) of rule 2], and starts to execute.
6. At time 14, after J_2 and J_3 complete, J_4 has the processor and is granted the resource *Shaded* because its priority is higher than Ω, the ceiling of the system at the time. It starts to execute. The ceiling of the system is raised to 1, the priority ceiling of *Shaded*.
7. At time 16, J_4 requests *Black*, which is free. The priority of J_4 is lower than $\hat{\Pi}(16)$, but J_4 is the job holding the resource (i.e., *Shaded*) whose priority ceiling is equal to $\hat{\Pi}(16)$. Hence, according to (ii) of part (b) of rule 2, J_4 is granted *Black*. It continues to execute. The rest of the schedule is self-explanatory.

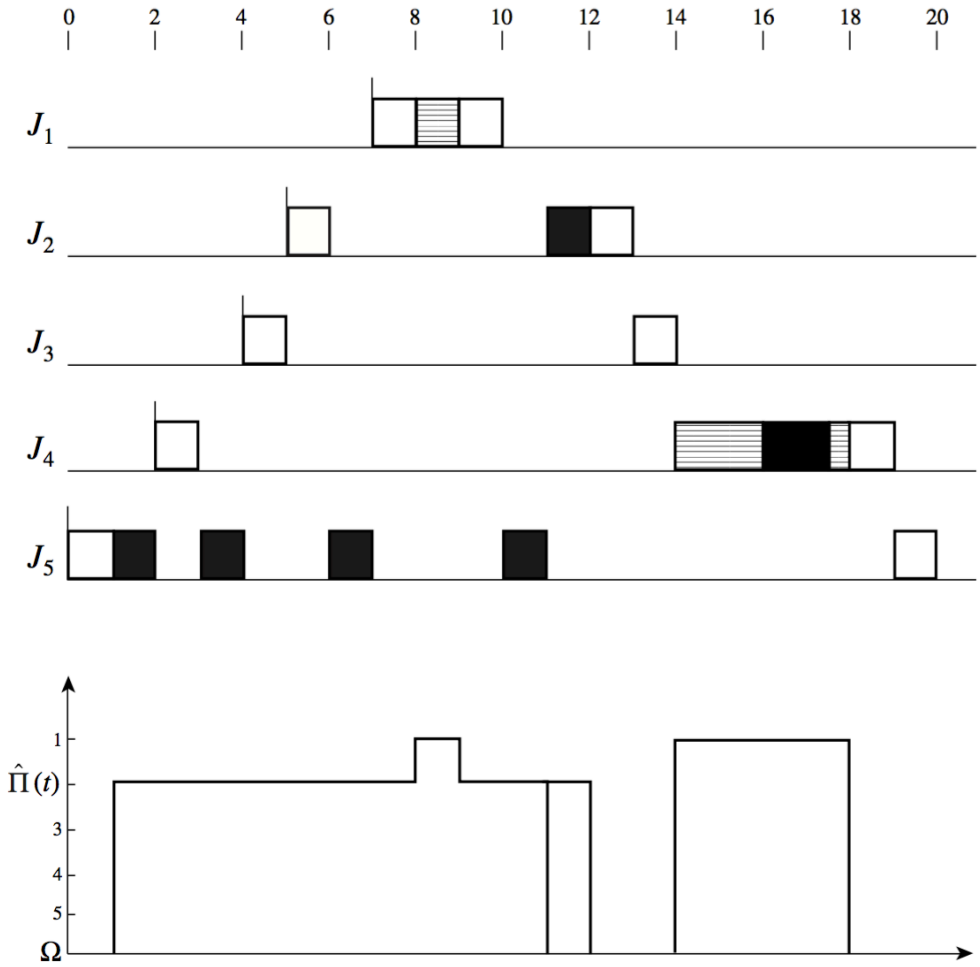


FIGURE 8–10 A schedule illustrating priority-ceiling protocol.

8.6 Stack-based, priority-ceiling (ceiling-priority) protocol

In this section, we give two different definitions of a protocol that is simpler than the priority- ceiling protocol but has the same worst-case performance as the priority-ceiling protocol. The different definitions arise from two different motivations: to provide stack-sharing capability and to simplify the priority-ceiling protocol. They led to the two different names of the same protocol.

8.6.1 Motivation and Definition of Stack-Sharing Priority-Ceiling Protocol

A resource in the system is the run-time stack. Thus far, we have assumed that each job has its own run-time stack. Sometimes, especially in systems where the number of jobs is large, it may be necessary for the jobs to share a common run-time stack, in order to reduce overall memory demand. Space in the (shared) stack is allocated to jobs contiguously in the last-in-first-out manner. When a job J executes, its stack space is on the top of the stack. The space is freed when the job completes. When J is preempted, the preempting job has the stack space above J's. J can resume execution only after all the jobs holding stack space above its space complete, free their stack spaces, and leave J 's stack space on the top of the stack again.

To ensure deadlock-free sharing of the run-time stack among jobs, we must ensure that no job is ever blocked because it is denied some resource once its execution begins. This observation leads to the following priority-ceiling protocol, called the *stack-based, priority-ceiling protocol*. *This protocol allows jobs to share the run-time stack if they never self-suspend*.

Rules Defining Basic Stack-Based, Priority-Ceiling Protocol

0. *Update of the Current Ceiling*: Whenever all the resources are free, the ceiling of the system is Ω. The ceiling $\hat{\Pi}(t)$ is updated each time a resource is allocated or freed.

1. *Scheduling Rule*: After a job is released, it is blocked from starting execution until its assigned priority is higher than the current ceiling $\hat{\Pi}(t)$ of the system. At all times, jobs that are not blocked are scheduled on the processor in a priority-driven, preemptive manner according to their assigned priorities.

2. *Allocation Rule*: Whenever a job requests a resource, it is allocated the resource.

Advantage of this rule

- According to the scheduling rule, when a job begins to execute, all the resources it will ever need during its execution are free. (Otherwise, if one of the resources it will need is not free, the ceiling of the system is equal to or higher than its priority.)
- *No job is ever blocked once its execution begins*.
- Likewise, when a job J is preempted, all the resources the preempting job will require are free, ensuring that the preempting job can always complete so J can resume. Consequently, *deadlock can never occur*.

Job	r_i	e_i	π_i
J_1	7	3	1
J_2	5	3	2
J_3	4	2	3
J_4	2	6	4
J_5	0	6	5

The schedule in Figure 8–17 shows how the system of jobs would be scheduled if the stack-based, priority-ceiling protocol were used instead of the basic priority- ceiling protocol. To better illustrate the stack-based protocol, we let J_2 be released at 4.8 and the execution time of the critical section of J_2 be 1.2. At time 2 when J_4 is released, it is blocked from starting because its priority is not higher than the ceiling of the system, which is equal to 2 at the time. This allows J_5 to continue execution. For the same reason, J_3 does not start execution when it is released. When J_2 is released at time 4.8, it cannot start execution because the ceiling of the system is 2. At time 5, the resource held by J_5 becomes free and the ceiling of the system is at Ω. Consequently, J_2 starts to execute since it has the highest priority among all the jobs ready at the time. As expected, when it requests the resource *Black* at time 6, the resource is free. It acquires the resource and continues to execute. At time 7 when J_1 is released, its priority is higher than the ceiling of the system, which is 2 at the time. (Again, this fact indicates that the resource *Shaded*, which it will require later, is free.) J_1, therefore, preempts J_2 and holds the space on the top of the stack until it completes at time 10. J_2 then resumes and completes at 11. Afterwards, J_3, J_4, and J_5 complete in the order of their priorities.

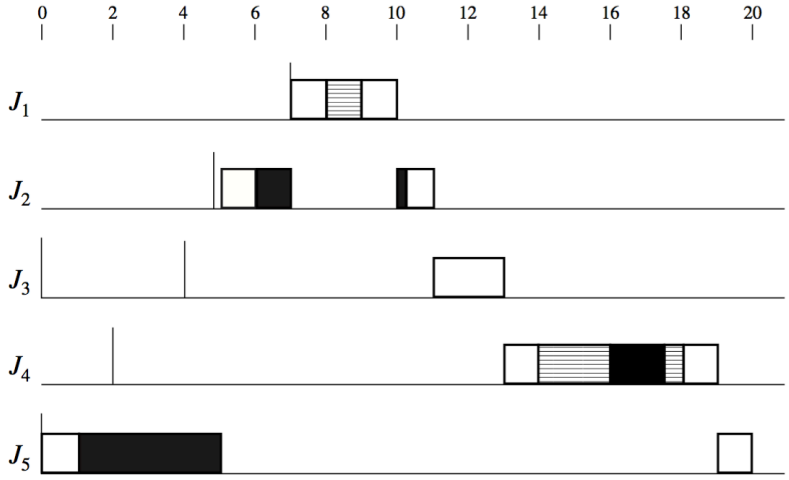


FIGURE 8–17 Schedule illustrating the stack-based, priority-ceiling protocol.

8.7 Use of priority-ceiling protocol in 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.

8.7.1 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: $T_1 = (0.5, 2.0, 0.2; [Black; 0.2])$, $T_2 = (3.0, 1.5; [Shaded; 0.7])$, and $T_3 = (5.0, 1.2; [Black; 1.0\ [Shaded; 0.4])$. 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. We use consecutive positive integers to denote the priorities of all the ready jobs, the highest priority being 1.

To emphasize that the priority ceiling of a resource R_i may change with time, we denote it by $\prod t(R_i)$

1. At time 0, there are only two ready jobs, $J_{2,1}$ and $J_{3,1}$. $J_{2,1}$ (and hence T_2) has priority 1 while T_3 has priority 2, the priority of $J_{3,1}$. The priority ceilings of *Black* and *Shaded* are 2 and 1, respectively. Since $J_{2,1}$ has a higher priority, it begins to execute. Because no resource is in use, the ceiling of the system is Ω. At time 0.3, $J_{2,1}$ acquires *Shaded*, and the ceiling of the system rises from Ω to 1, the priority ceiling of *Shaded*.
2. At time 0.5, $J_{1,1}$ is released, and it has a higher priority than $J_{2,1}$ and $J_{3,1}$. Now the priorities of T_1, T_2, and T_3 become 1, 2, and 3, respectively. The priority ceiling Π_t (*Black*) of *Black* is 1, the priority of $J_{1,1}$ and T_1. The priority ceiling Π_t (*Shaded*) of *Shaded* becomes 2 because the priority of $J_{2,1}$ and T_2 is now 2. The ceiling of the system based on these updated values is 2. For this reason, $J_{1,1}$ is granted the resource *Black*. The ceiling of the system is 1 until $J_{1,1}$ releases *Black* and completes at time 0.7. Afterwards, $J_{2,1}$ continues to execute, and the ceiling of the system is again 2. When $J_{2,1}$ completes at time 1.7, $J_{3,1}$ commences to execute and later acquires the resources as shown.
3. At time 2.5, $J_{1,2}$ is released. It has priority 1, while $J_{3,1}$ 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, $J_{1,2}$ becomes blocked at 2.5. At time 2.9, $J_{3,1}$ releases *Black*, and $J_{1,2}$ commences execution.
4. At time 3.0, only T_1 and T_2 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 $J_{1,3}$ of T_1 has a later deadline than $J_{2,2}$. (Again, T_3 has no ready job.) Hence, the priority of T_1 is 2 while the priority of T_2 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.

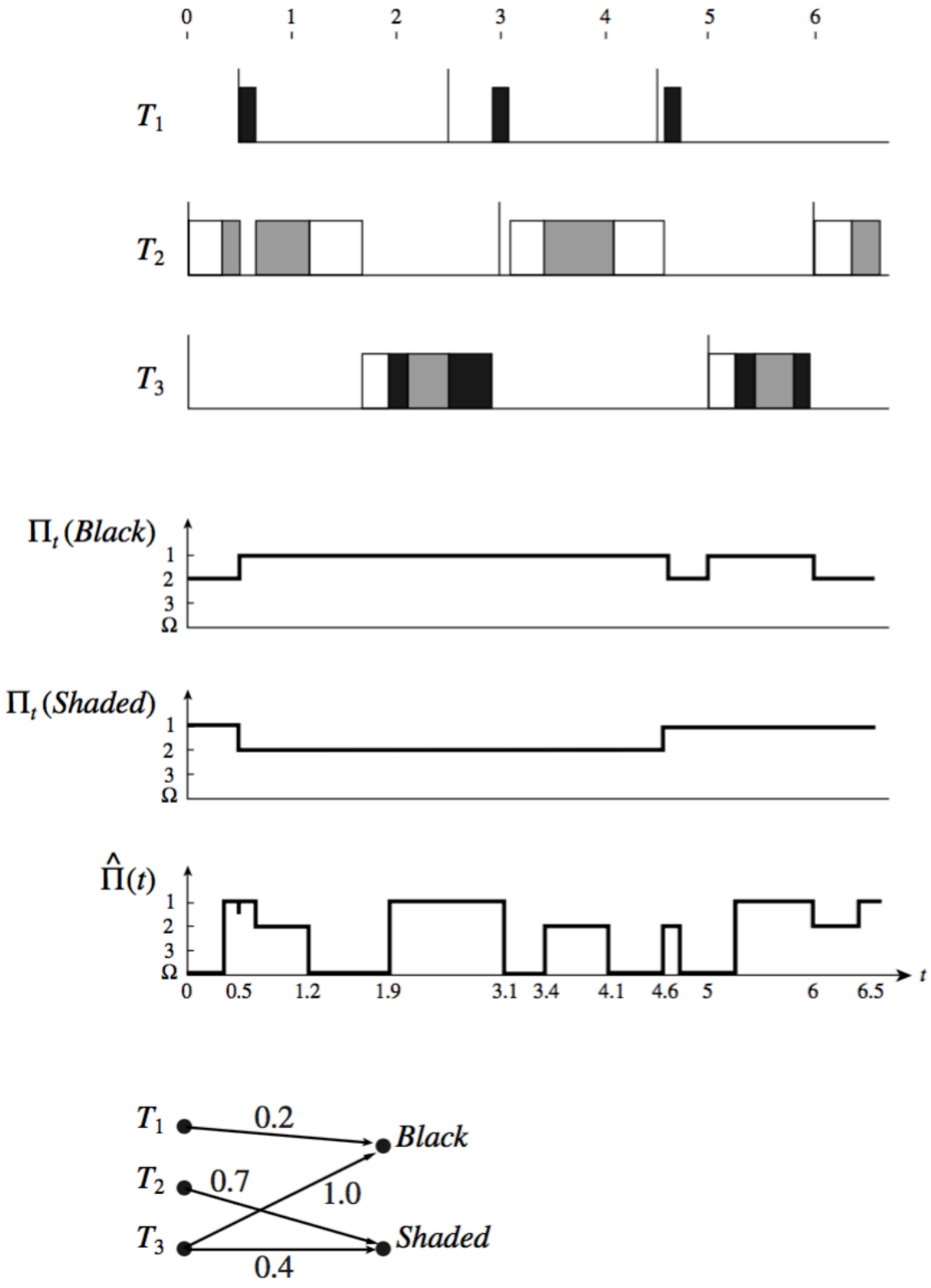


FIGURE 8–18 Example illustrating the use of the basic priority-ceiling protocol in an EDF system.

In a system with ρ resources, each of which is required by n periodic tasks on average, the time required to update the priority ceilings of all resources is $O(\rho)$ each time a new job is released. This is a significant amount of overhead. For systems where the release-time jitters are negligible, we can save this run-time overhead by precomputing and storing the priority ceilings of all the resources for every time interval between consecutive releases of all N jobs in the entire hyperperiod. Each time a

new job is released, we use the precomputed priority ceilings of the interval beginning from the release time of the job. The storage overhead for this purpose is $O(N\rho)$.

8.8 Preemption-ceiling protocol

For a fixed preemption-level system, there is a simpler approach to control resource accesses. The approach is based on the clever observation that the potentials of resource contentions in such a dynamic-priority system do not change with time, just as in fixed-priority systems, and hence can be analyzed statically. The observation is supported by the following facts:

1. The fact that a job J_h has a higher priority than another job J_l and they both require some resource does not imply that J_l can directly block J_h. This blocking can occur only when it is possible for J_h to preempt J_l.
2. For some dynamic priority assignments, it is possible to determine a priori the possibility that jobs in each periodic task will preempt the jobs in other periodic tasks.

Because of fact 1, when determining whether a free resource can be granted to a job, it is not necessary to be concerned with the resource requirements of all higher-priority jobs; only those that can preempt the job. Fact 2 means that for some dynamic priority systems, the possibility that each periodic task will preempt every other periodic task does not change with time, just as in fixed-priority systems. no job in a periodic task with a smaller relative deadline is ever preempted by jobs in periodic tasks with identical or larger relative deadlines, despite the fact that some jobs in the latter may have higher priorities.

8.8.1 Preemption Levels of Jobs and Periodic Tasks

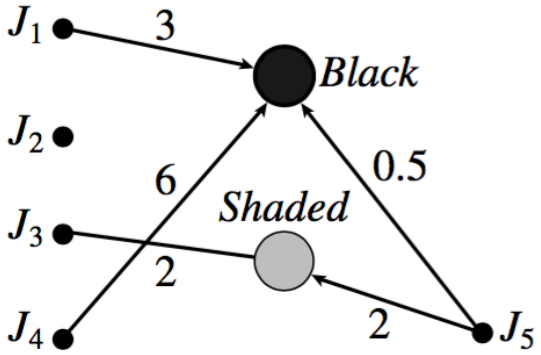
The possibility that a job J_i will preempt another job is captured by the parameter *preemption level* ψ_i of J_i. The preemption levels of jobs are functions of their priorities and release times. According to a *valid preemption-level assignment*, for every pair of jobs J_i and J_k, the preemption level ψ_i of J_i being equal to or higher than the preemption level ψ_k of J_k implies that it is never possible for J_k to preempt J_i. Stated in another way,

Validity Condition: If π_i is higher than π_k and $r_i > r_k$, then ψ_i is higher than ψ_k.

Given the priorities and release times of all jobs, this condition gives us a partial assignment of preemption levels, that is, the preemption levels of a subset of all jobs. The preemption levels of jobs that are not given by the above rule are valid as long as the linear order over all jobs defined by the preemption-level assignment does not violate the validity condition.

Figure 8–19 gives another example. As usual, the five jobs are indexed in decreasing priorities. Their release times are such that $r_4 < r_5 < r_3 < r_1 < r_2$. We note that J_1, the job with the highest priority, has a later release time than J_3, J_4, and J_5. Hence, J_1 should have a higher preemption level than these three jobs. However, it is never possible for J_1 to preempt J_2 because J_1 has an earlier release time, and it is never possible for J_2 to preempt J_1, because J_2 has a lower priority. We therefore give these two jobs the same preemption level. Similarly, J_3 should have a higher preemption level than J_4 and J_5, and we can give J_4 and J_5 the same preemption level. In summary, we can assign ψ_i for $i = 1, 2, 3, 4$, and 5 the values 1, 1, 2, 3, and 3, respectively; it is easy see that this is a valid preemption level assignment. (Again, a smaller integer represents a higher preemption level.) Alternatively, we can assign preemption levels according to the release times of the jobs: the earlier the release time, the lower the

preemption level. This assignment also satisfies the validity condition. The resultant preemption levels are 2, 1, 3, 5, and 4, respectively.



Jobs	r_i	e_i	π_i	ψ_i
J_1	5	5	1	1
J_2	7	3	2	1
J_3	4	4	3	2
J_4	0	9	4	3
J_5	3	4	5	3

FIGURE 8–19 A schedule according to the preeption-ceiling protocol.

Let us now return to periodic tasks. When periodic tasks are scheduled on the EDF basis, a valid preemption-level assignment is according to the relative deadlines of jobs: the smaller the relative deadline, the higher the preemption level. (An assumption here is that

either release-time jitters are negligible or the relative deadlines of all jobs remain fixed despite release-time jitters.) For this preemption-level assignment, all the jobs in every periodic task in a deadline-driven system have the same preemption level. This is an example of fixed preemption-level systems. A system of periodic tasks is a *fixed preemption-level system* if there is a valid assignment of preemption levels to all jobs such that all the jobs in every task have the same preemption level. Clearly, all fixed-priority systems are also fixed preemption- level systems. Indeed, an obvious preemption-level assignment in a fixed-priority system is to make the preemption level of each job equal to its priority.

When there is no chance of confusion, we call the preemption level of all the jobs in a fixed preemption-level task T_i the *preemption level of the task* and denote it by ψ_i. We index periodic tasks in a fixed preemption-level system according to their preemption levels: the higher the level, the smaller the index.

8.8.2 Definitions of Protocols and Duration of Blocking

A *preemption-ceiling protocol* makes decisions on whether to grant a free resource to any job based on the preemption level of the job in a manner similar to the priority-ceiling protocol. This protocol also assumes that the resource requirements of all the jobs are known a priori. After assigning preemption levels to all the jobs, we determine the preemption ceiling of each resource. Specifically, when there is only 1 unit of each resource, which we assume is the case here, the *preemption ceiling* (R) of a resource R is the highest preemption level of all the jobs that require the resource. For the example in Figure 8–19, the preemption ceiling of *Black* is 1, while the preemption ceiling of *Shaded* is 2.

The (*preemption*) *ceiling of the system* $\hat{\psi}(t)$ at any time t is the highest preemption ceiling of all the resources that are in use at t. When the context is clear and there is no chance of confusion, we will simply refer to $\hat{\psi}(t)$ as the ceiling of the system. We again use Ω to denote a preemption level that is lower than the lowest preemption level among all jobs since there is no possibility of confusion. When all the resources are free, we say that the ceiling of the system is Ω.

Like the priority-ceiling protocol, the preemption-ceiling protocol also has a basic version and a stack-based version. The former assumes that each job has its own stack and the latter allows the jobs to share a common stack. Basic versions of priority-ceiling and preemption-ceiling protocols differ mainly in their allocation rules. For this reason, only the allocation rule of the basic preemption-ceiling protocol is given below. You can see that the principle of this rule for both protocols is the same, the only difference being the parameters (i.e., priority or preemption levels and ceilings) used by the rule.

Rules of Basic Preemption-Ceiling Protocol

1 and 3. The *scheduling rule* (i.e., rule 1) and *priority inheritance rule* (i.e., rule 3) are the same as the corresponding rules of the priority-ceiling protocol.

2. *Allocation Rule*: Whenever a job J requests resource R at time t, one of the following two conditions occurs:

(a) R is held by another job. J's request fails, and J becomes blocked.

(b) R is free.

(i) If J's preemption level $\psi(t)$ is higher than the current preemption ceiling $\hat{\Psi}(t)$ of the system, R is allocated to J.

(ii) If J's preemption level $\psi(t)$ is not higher than the ceiling $\hat{\Psi}(t)$ of the system, R is allocated to J only if J is the job holding the resource(s) whose preemption ceiling is equal to $\hat{\Psi}(t)$; otherwise, J's request is denied, and J becomes blocked.

The stack-based preemption-ceiling protocol is called the Stack-Based Protocol (SBP) and is defined by the following rules.

Rules of Basic Stack-Based, Preemption-Ceiling Protocol

0. *Update of the Current Ceiling*: Whenever all the resources are free, the preemption ceiling of the system is Ω. The preemption ceiling $\hat{\Psi}(t)$ is updated each time a resource is allocated or freed.

1. *Scheduling Rule*: After a job is released, it is blocked from starting execution until its preemption level is higher than the current ceiling $\hat{\Psi}(t)$ of the system and the preemption level of the executing job. At any time t, jobs that are not blocked are scheduled on the processor in a priority-driven, preemptive manner according to their assigned priorities.

2. *Allocation Rule*: Whenever a job J requests for a resource R, it is allocated the resource.

3. *Priority-Inheritance Rule*: When some job is blocked from starting, the blocking job inherits the highest priority of all the blocked jobs.

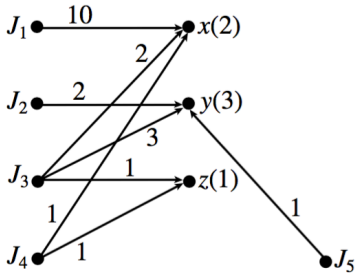
8.9 Controlling accesses to multiple-unit resources

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).

8.9.1 Priority (Preemption) Ceilings of Multiple-Unit Resources

The first step in extending the priority-ceiling protocol is to modify the definition of the priority ceilings of resources. We let (R_i, k), for $k \leq \nu_i$, denote the priority ceiling of a resource R_i when k out of the $\nu_i (\geq 1)$ units of R_i are free. If one or more jobs in the system

require more than k units of R_i , (R_i , k) is the highest priority of all these jobs. If no job requires more than k units of R_i , (R_i , k) is equal to , the nonexisting lowest priority. In this notation, the priority ceiling (R_j) of a resource R_j that has only 1 unit is $(R_j,0)$.



	Resources	$x(2)$	$y(3)$	$z(1)$
Units Required by	J_1	1	0	0
	J_2	0	2	0
	J_3	2	3	1
	J_4	1	0	1
	J_5	0	1	0
$\Pi(*, 0)$		π_1	π_2	π_3
$\Pi(*, 1)$		π_3	π_2	Ω
$\Pi(*, 2)$		Ω	π_3	Ω
$\Pi(*, 3)$		Ω	Ω	Ω

FIGURE 8–21 Example of priority ceilings of multiple-unit resources.

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

Let $k_i(t)$ denote the number of units of R_i that are free at time t. Because this number changes with time, the priority ceiling of R_i changes with time. The (current) priority ceiling of the system at time t is equal to the highest priority ceiling of all the resources at the time.

Figure 8–21 gives an example. The resource requirement graph gives the numbers of units of the resources X, Y, and Z required by the five jobs that are indexed in decreasing order of their priorities. The table below the graph gives the priority ceilings of each resource for different numbers of free resource units. For example, there are 2 units of X. When 1 unit of X is used, only J_3 is directly blocked. Therefore, $\Pi(X, 1)$ is π_3. J_1 is also directly blocked when both units of X are in use. For this reason, $\Pi(X, 0)$ is π_1, the higher priority between π_1 and π_3. When both units of X are free, the ceiling of the resource is Ω. Similarly, since J_2, J_3, and J_5 require 2, 3, and 1 unit of Y, which has 3 units, $\Pi(Y, 0)$, $\Pi(Y, 1)$, and $\Pi(Y, 2)$ are equal to π_2, π_2, and π_3, respectively. Suppose that at time t, 1 unit of each of X, Y, and Z is free. The priority ceilings of the resources are π_3, π_2, and Ω, respectively, and the priority ceiling of the system is π_2.

The preemption ceilings of resources that have multiple units can be defined in a similar manner: The preemption ceiling $\Psi(R_i, k)$ of the resource R_i when k units of R_i are free is the highest preemption level of all the jobs that require more than k units of R_i. Hence, if the jobs in Figure 8–21 were indexed in decreasing order according to their preemption levels and we replaced π_i and $\Pi(*, k)$ in the table by ψ_i and $\Psi(*, k)$, respectively, we would get the preemption ceilings of the three resources. The preemption ceiling of the system at time t is equal to the highest preemption ceiling of all the resources at the time.

8.10 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).

8.10.1 Convex-Ceiling Protocol

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

Motivation and Assumptions

A well-known way to ensure serializability is Two-Phase Locking (2PL). According to the 2PL protocol, a job never requests any lock once it releases some lock.

We can easily get concurrency-control protocols that not only ensure serializability but also prevent deadlock and transitive blocking by augmenting the protocols described in earlier sections with the two-phase locking rule. As a result, we have the NPCS-2PL and the PCP- 2PL protocols. The augmented protocols have an obvious shortcoming: prolonged blocking. Following the 2PL rule, a job may hold a data object even when it no longer require the object. As a consequence, it may block other jobs for a longer duration.

The convex-(priority-) ceiling protocol is another extension of the priority-ceiling protocol. It is an improvement over the PCP-2PL protocol because it reduces the duration of blocking.

Definition and Capability. As with the priority-ceiling protocol, at any time t when the scheduler receives a request to access an object R for the first time from any job J, it computes the system ceiling $\hat{\Pi}(t)$. $\hat{\Pi}(t)$ is equal to the highest priority of the priority-ceiling functions of all the jobs in the system. The convex-ceiling protocol defined by the following rules.

Rules of Convex-Ceiling Protocol

1. *Scheduling Rule*: At any time, jobs that are not suspended are scheduled on the processor in a preemptive, priority-driven manner. Upon its release, the current priority of every job J_i is its assigned priority π_i. It executes at this priority except when the inheritance rule is applied.
2. *Allocation Rule*: When a job J_i requests to access a data object R for the first time,
   - **(a)** if J_i's priority is higher than the system ceiling $\hat{\Pi}(t)$, J is allowed to continue execution and access R;
   - **(b)** if J_i's priority is not higher than $\hat{\Pi}(t)$,
     - **i.** if $\hat{\Pi}(t)$ is equal to $\Pi(J_i, t)$, J_i is allowed to access R;
     - **ii.** otherwise, J is suspended.
3. *Priority-Inheritance Rule*: When J_i becomes suspended, the job J_l whose priority-ceiling function is equal to the system ceiling at the time inherits the current priority $\pi_i(t)$ of J_i

9 Multiprocessor Scheduling, Resource Access Control and Synchronization

Thus far, we have ignored several realistic facts: Almost every system contains more than one processor, control and data dependencies impose precedence constraints among jobs, and timing constraints of jobs are usually not independent.

9.1 Model of multiprocessor and distributed systems

The systems contains more than one processors. Some systems are known as multiprocessor systems while the others are known as distributed systems. A multiprocessor system is *tightly coupled* so that global status and workload information on all processors can be kept current at a low cost. The system uses a centralized dispatcher or scheduler. When each processor has its own scheduler, the actions and the decisions of the scheduler of all processors are coherent. In contrast, a distributed system is *loosely coupled* in such a system that it is costly to keep global status and workload information current. The schedulers on the different processors may make scheduling and resource access control decisions independently. As a result, their decisions may be incoherent as a whole.

It is assumed that each processor has its own scheduler in this chapter. Each scheduling, resource access control or synchronization algorithm will be evaluated to see how much the algorithm relies on the current global information, how much coordination among schedulers is required and therefore how suitable the algorithm is for loosely coupled systems.

9.1.1 IDENTICAL VERSUS HETEROGENEOUS PROCESSORS

It is said that the processors are of the same type or identical, if the processors can be used interchangeably. For example, in a parallel machine, each of the CPUs can execute every computation job in the system, so the CPUs are identical. If any message from a source to a destination can be sent on any of the data links connecting them, then the links are identical.

Different types of processors cannot be used interchangeably. Different types of processors may have different functions. As an example, CPUs, file disk, and transmission links are functionally different. So, they cannot be used interchangeably. Processors can be of different types for many reasons. For example, if the designer decides to use some CPUs for only some components of the system but not others, then the CPUs are divided into different types according to the components that can execute them. In a static system, the application system is partitioned into μ components and jobs in each component execute on a fixed CPU. CPUs are viewed as μ different processors.

The model of heterogeneous processors used here is known as the unrelated processors model in scheduling theory literature. According to this model, each job can execute on some types of processors but, in general, not all types. Different types of processors may have different speeds. The execution times of each job on different types of processors are unrelated hence the name of the model. For example, the execution time of a computation intensive job is 1 second in CPU1 but is 5 seconds on a less powerful CPU2. Because CPU2 has better interrupt handling and I/O capabilities, the execution time of an I/O intensive job is 10 seconds on CPU1 but is only 3 seconds in CPU2. Both jobs cannot execute on a transmission link and signal processor, so their execution time on these kinds of processors are infinite. The unrelated models allows us to characterize all system.

Processor is denoted by the letter P. the system contains μ types of processors. There are a total of m_1 processors of type i i.e P_i for i = 1, 2, 3, . . . , m.

9.1.2 END-TO-END JOBS AND TASKS

In practice, a system function is often provided by a set of related jobs, that is, a task. The jobs in each task may have precedence constraints.

A task in real time of a monitor system consists of three jobs: sampling, encoding, and processing the reading of a sensor on a field processors; sending the sensor data by a communication processor to the central control processor; and correlating and displaying the data with other sensor data on the control processor.

9.1.2.1 JOB SHOPS AND FLOW SHOPS

Concurrency in a multiprocessor system arises as jobs of different tasks sequencing through different processors in a pipeline manner. The classical jobs shop and flow shop model captures this types of concurrency. According to the job shop model, each task $T_1$is a chain of n(i) jobs denoted by $J_{i,k}$ for k = 1, 2, 3, . . . n(i) for all $1 \leq k \leq n(i)$. The adjacent jobs $J_{i,k}$ and $J_{i,k+1}$ on the chain execute on different processors $J_{i,k+1}$ become ready for execution only when $J_{i,k}$ completes. For example, the real-time monitor task is a chain of three jobs.

We can specify the processors on which n(i) jobs in each task T_i execute by the visit sequence $V_i = (V_{i,1}, V_{i,2}, \ldots, V_{i,n(i)})$ of the task. The k^{th} entry $V_{i,k}$ in this sequence gives the name of the processors on which the job $J_{i,k}$ executes. Therefore, the visit sequence of the real time monitor task is (field processor, communication processor, and control processor).

For example, visit sequence $V_i = (P_1, P_2)$, $V_2 = (P_2, P_3, P_4, P_3, P_1)$ of tasks T_1 and T_2 in a system means that T_1 has two jobs $J_{1,1}$ on P_1 followed by $J_{1,2}$ on P_2. T_2 has five jobs and they execute in turn on P_2 then on P_3, P_4, P_3 and finally on P_1.

A flow shop is a special job shop in which all the task have the same visit sequence. For example, suppose that all the tasks in a real-time monitor system are similar to the monitor task mentioned above: each task samples, processes, and displays the reading of a different sensor. The visit sequences of all the monitor tasks are (field processor, communication processor, control processor) if the system has only one processor of each the three type.

9.1.2.2 END TO END TIMING CONSTRAINTS

The job shop model of multiprocessor systems we adopt here differs from the classical model in two ways. The first difference is a substantive one. The classical job shop model assumes that all tasks are ready for execution at the same time. Typically, the objective of classical job shop scheduling is to maximize the throughput (i.e. the number of tasks completed per unit time) of the system or to minimize the average response time of the tasks. In contrast, our tasks have arbitrary release times and deadlines, and some tasks have hard deadlines. Meeting hard deadlines is always our primary objective.

The timing constraints that can be derived directly from the high-level requirement of the applications are typically end to end in nature. They give the release time and deadline of each task as a whole.

Formally, we let the release time r_i of a task T_i in a job shop be the release time of the first job $J_{i,1}$ in the task. The deadline d_i of the task is the deadline of its last job $J_{i,n(i)}$. As long as the last job completes by the task's deadline, it is not important when the other jobs in the task complete. The executions of these jobs are constrained only by the dependencies between them and by the fact that they must complete sufficiently early to allow the on-time completion of the last job. Because the timing constraints of such a task are imposed on the jobs at the two ends of the task, we call them end-

to-end release time and end-to-end deadline when we want to emphasize this point. A task that has an end-to-end release time and deadline is an end-to-end task.

9.1.2.3 PERIODIC END TO END TASK

The second difference between the classical job shop model and ours is mostly a matter of notation. Just like periodic tasks in uniprocessor environment, end-to-end tasks in a multiprocessor system may be periodic. An End to end task T_i is periodic with period p_i if a chain of n(i) jobs is released every p_i or more units of time and the jobs in the chain executes in turn on processors according to the visit sequence ($V_{i,1}$, $V_{i,2}$, ... , $V_{i,n(i)}$). We call the infinite sequence of jobs that executes on processor $V_{i,k}$ the kth subtask $T_{i,k}$, for k = 1, 2, …, n(i) of T_i.

To be exact, the first subtask $T_{i,1}$ of an end-to-end periodic task T_i is a periodic task with period p_i. Following the completion of the jth job of the first subtask $T_{i,1}$ on processor $V_{i,1}$ the jth job of the second subtask $T_{i,2}$ can be released on processor $V_{i,2}$. Similarly, following the completion of the jth job of the kth subtask $T_{i,k}$ on processor $V_{i,k}$ the jth job in the (k+1)st subtask $T_{i,k+1}$ can be released on processor $V_{i,k+1}$.

The end to end periodic task T_i is called parent task n(i) subtask and the subtask in the same parent task are called sibling/(sub) task. The period of an end to end periodic task is the period of its first subtask. In this, the phase Φ_i of the task T_i means the release time of first job of first sub-task. Similarly, execution time $e_{i,k}$ means the maximum amount of time required to complete any job in $T_{i,k}$.

9.1.2.4 Parallelism

In addition to pipelined executions of jobs on different types of processors, parallel executions are possible whenever there is more than one processor of the same type.

Example: in an ATC system there may be an array of signal processors making it possible to execute many FFTs in parallel. Similarly, multiple links and switches between a sender and receiver pair provide parallel paths that can be used to increase the throughput or reduce delays of messages between them. The traditional multiprocessor models used in studies on parallel and distributed scheduling, task assignment and load balancing capture this parallelism.

9.1.3 Local versus Remote Resources

We assume that each resource resides on a processor. By a resource R residing on a processor we mean that the scheduler of the processor controls the access to the resource R and when a job uses the resource, its critical section executes on the processor.

- Resource models:
  - MPCP (Multiprocessor Priority-Ceiling Protocol) Model
  - End-to-end model

9.1.3.1 MPCP Resource Model

An extension of the priority-ceiling protocol to control resource access in multiprocessor systems is MPCP. This model calls the processor on which each resource resides its *synchronization processor*. The processor on which each job is released and become ready for execution is the *local processor* of the job. From the perspective of a job, a resource that also resides on the local processor of the job is a local resource, and resource that resides on another processor is a *remote resource*. A global resource is required by jobs that have different local processors.

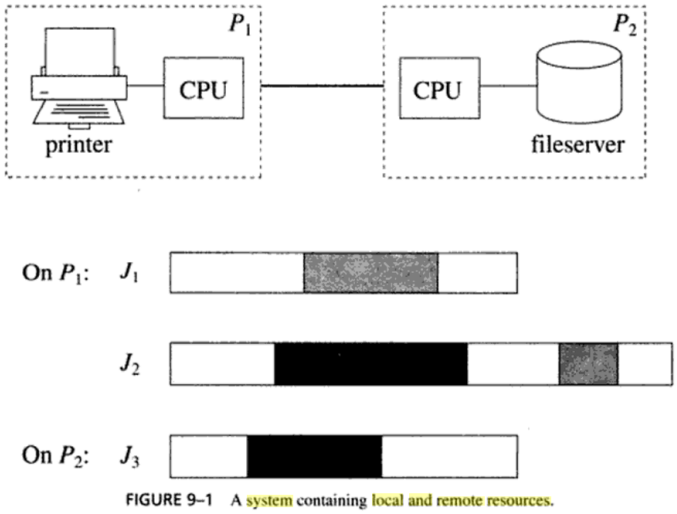


FIGURE 9–1 A system containing local and remote resources.

Figure 9-1 gives an example. The system shown here has three jobs, two processors, Pi and P2, and two resources, printer and fileServer. Processor P1 is the synchronization processor of printer, and processor P2 is the synchronization processor fileServer. The jobs J1 and J2 are local to P1, while J3 is a local job on P2. Suppose that J1 is a text formating and printing job. It executes on P1 and uses only the local resource printer. In other words, it has only a local critical section, which is shown as a shaded box in the figure. Similarly, the backup job J3 requires only fileServer. Since fileServer resides on its local processor, this resource is local to J3. (Critical sections during which the fileServer is used are shown as black boxes.) On the other hand, during the execution of J2, it requires a file maintained by the fileServer. fileServer is a global resource. During the remote global critical section while J2 uses this resource, the file server executes on the behalf of J2 on P2. (Later, we will simply say that the global critical section of J2 executes on P2.) After the global critical section of J2 completes, it returns to execute on P1 and uses the printer before it completes.

In summary, according to MPCP model, a job may require both local and remote resource. The access to each resource is controlled by the scheduler of the synchronization processor of resource. The critical section during which the job uses a resource executes on the synchronization processor of the resource. Hence, a job that has k_r (outermost) remote critical sections gives up its local processor k_r times and become eligible for execution on its local processor k_r+1 times. Requests for resources on each processor may be from local jobs and remote jobs and the resource access-control protocol used on the processor may handle these requests in different ways.

In general, local and remote critical sections of a job may be nested. A job may hold a resource on one processor and then request resources on another processor. The MPCP model allows such nested critical sections.

9.1.3.2 End-to-End Resource Model

A reasonable restriction is that no job makes nested requests for resources that reside on different processors. Stated another way, all the resources used by every job during every nested critical section reside on the same processor. In a system that satisfies this assumption, we can view each job that requires resources on more than one processor as an end-to-end job.

In Fig 9.1 job J2 consists of three component jobs.

- The first one executes on its local processor and does not require any resource
- Its remote critical section is the second component, and this component executes on the remote processor P2.
- The third component job is the portion after the remote critical section and executes on P1.

Since J_1 and J_3 do not require any remote resource, each of them consists of only one component.

- Job that require only local resources has only one component.

In general, each job that requires only local resources has only one component which executes on local processor. Each remote outermost critical section of a job is a component job which executes on

the synchronization processor of the remote resources guarded by the critical section and the critical sections nested in it. A portion of a job is a component job on its local processor if the portion

1. is before or after a remote critical section and
2. either requires no resource or only local resources

By this definition, every component job requires only resources on the processor on which the component job executes.

The scheduler of each processor can treat all requests for resources controlled by it as local requests.

9.1.4 Interprocessor Communication

We seem to ignore the time required to synchronize jobs and transmit data among them. The reason is that it is not necessary to account for the cost of interprocessor communication separately in an ad hoc manner. The task and resource models used here allow us to take this cost into account in several ways.

A special-case of practical interest is where interprocessor communication is via shared memory. Sometimes, we treat shared memory as a plentiful resource and do not include this resource in our model. The implication is that the execution of a job is never adversely affected by memory contention; therefore, the cost of interprocessor communication is negligible. This assumption is clearly not true in general. When the assumption is not true, we can model shared memory explicitly either as a resource or as a processor.

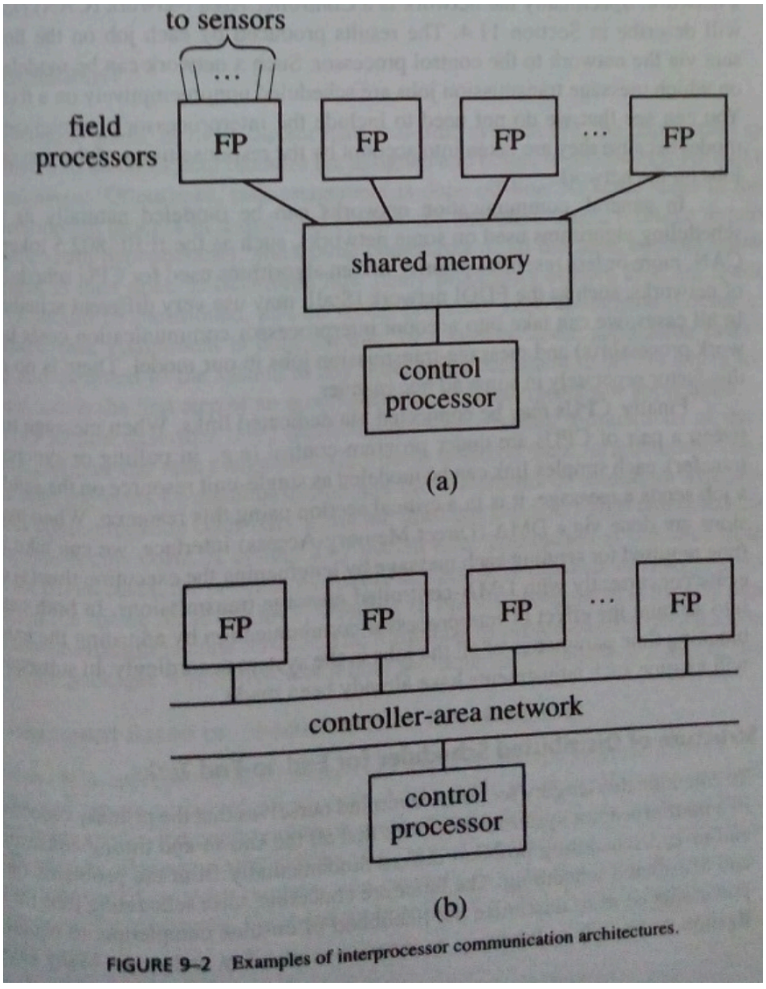


FIGURE 9–2 Examples of interprocessor communication architectures.

As an example, fig 9.2(a) shows a possible configuration of a real-time monitor system that has many field processors. The producer jobs that collect and process sensor data execute on these processors. Consumer jobs that correlate and display the data execute on the control processor. The jobs communicate via a shared dual-port memory. Each field processor is connected via a dedicated link to a port of the memory that is shared by all field processors. To take into account the effect of memory contention, we can model the shared memory as a processor. The system contains three types of processors: field processors, shared-memory processor and the control processor. The workload consists of end-to-end jobs, each containing a memory-access job. (Specifically, each producer is an end-to-end job whose first component executes on a field processor and whose memory-access component executes on the shared memory. Each consumer job is an end-to-end job which first executes on the shared memory and then on the control processor.) In this way, the delay in the completion of each job caused by memory contention is taken into account by the response time of the

memory-access component of the job. Alternatively, you can model the shared memory as a global resource whose synchronization processor is the shared-memory "processor," and both the producer and consumer jobs require this resource.

Figure 9-2(b) shows another configuration where all the processors are connected via a network. Specifically the network is a Controller Area Network (CAN). The results produced by each job on the field processor are sent via the network to the control processor. Such a network can be modeled as a processor on which message transmission jobs are scheduled nonpreemptively on a fixed-priority basis. We do not need to include the interprocessor communication costs in our model because they are taken into account by the response times of the message-transmission jobs on the network.

In general, communication networks can be modeled naturally as processors. The scheduling algorithms used on some networks, such as the IEEE 802.5 token ring and CAN, more or less resemble priority-driven algorithms used for CPU scheduling. Other types of networks, such as the FDDI network, may use very different scheduling algorithms. In all cases, we can take into account interprocessor communication costs by including net-work processor(s) and message-transmission jobs in our model. There is no need to consider this factor separately in some ad hoc manner.

Finally, CPUs may be connected via dedicated links. When message transmissions between a pair of CPUs are under program control (e.g., in polling or synchronized message transfer), each simplex link can be modeled as single-unit resource on the sending CPU. When a job sends a message, it is in a critical section using this resource. When message transmissions are done via a DMA (Direct Memory-Access) interface, we can take into account the time required for sending each message by lengthening the execution time(s) of job(s) that execute concurrently with DMA-controlled message transmissions. In both cases, we can take into account the effect of interprocessor communication by adjusting the execution time and blocking time parameters of all the jobs in the system accordingly.

9.1.5 Structure of Distributed Scheduler for End-to-End Taks

The primary objective of scheduling in a multiprocessor system is to ensure that all the end-to-end timing constraints are met. The end-to-end scheduling problem differs fundamentally from the problems on multiprocessor and distributed scheduling. The latter are concerned with scheduling jobs on interchangeable processors so as to maximize the likelihood of on-time completion, to equalize resource utilization, to provide redundancy, to increase availability, and so on. Many excellent multiprocessor and distributed scheduling algorithms can be found in literature. They can be used with end-to-end scheduling algorithms when appropriate to improve the overall performance of a multiprocessor system, but by themselves, are not solutions to the end-to-end scheduling problem.

Assumptions made in this chapter:

- Each processor has its own scheduler.
- The scheduler uses a uniprocessor scheduling algorithm or an extension of such an algorithm to schedule the jobs on the processor.
- Schedulers on different processors may use different scheduling algorithms.

9.2 Task assignment

Most of the hard real-time systems are static in nature. To execute such hard real-time system in a multiprocessor environment, the application system is partitioned into modules or sub tasks and each module assigned and bound to different individual processors. This process is called **task assignment**. Generally, the task assignment is done offline but if the execution time, response requirement, data and control dependencies and the timing constraints are known then the task assignment can be done online. In this case, it is necessary to perform an acceptance test to decide whether to execute each

new task or not. There are three different types of task assignment methods. They are Simple Bin-Packing formulation, Variable Size-Bin-Packing formulation and RMFF Algorithm. The first method ignores both the cost of communication and the placement of resources. The second one considers communication cost only and the third one considers both communication cost and resource accessed cost.

9.2.1 TASK ASSIGNMENT BASED ON EXECUTION TIME REQUIREMENTS

Sometimes, it is appropriate to consider only the processing time requirements of the jobs and tasks and ignore communication costs. An example is when all tasks communicate via a shared memory; so, the communication costs of individual tasks are independent of where the task execute. For some applications (e.g., signal processing), it is possible to provide a sufficient number of memory modules and carefully lay out the address spaces of tasks to minimize memory contention. For these applications, the cost of communication can be made negligibly small.

9.2.1.1 SIMPLE BIN-PACKING FORMULATION

The task assignment method that ignores the communication cost and resource access cost and only considers the execution time requirement is called simple bin packing formulation. It is used to determine whether the number and kinds of processors planned for the systems are adequate. It is used for online acceptance test and load balancing.

Consider that the utilization of n periodic task is given and it is asked that the system should be portioned into modules in such a way that the tasks in each module are schedulable in a process according to a uniprocessor scheduling algorithm. A task assignment can be defined by the subset of task in every module. We can say that the assignment requires 'm' processors if it partitions the 'n' task into 'm' schedulable modules. The quality of the task assignment is measured by the number of processors required by the assignment. The smaller the number of processors required by an assignment, the better the assignment.

In order to schedule 'n' independent preemptable periodic task with relative deadlines equal to their periods based on EDF algorithm, we need to consider the utilization. In this case, the utilization of all the task is more than 1, therefore it is not possible to schedule feasibly on a single processor. To determine how to partition the task so that individual modules can be scheduled on minimum number of processors it suffices to know the utilization u_i for i = 1, 2, 3, . . . , n of all the tasks. We know the individual task of module is schedulable if their total utilization is less than or equal to one. However, it is constrained that the total utilization of all the tasks in each module should be less than or equal to some value U<1. In this way, we leave some spare capacity on each processor for sporadic and aperiodic tasks and future extensions. From this, we can formulate a task assignment problem as the simple bin packing problem in which the sizes of all the bins are equal to U and the sizes of items to be packed into the bins are u_i = 1, 2, . . . , n. the number of bins required to pack all the item is the number of processors required to feasibly schedule all the n tasks. The tasks represented by items packed in each bin are in a module assigned on a processor.

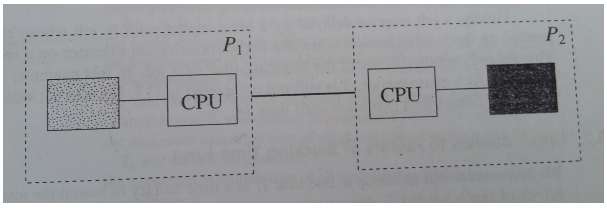
- Sometimes, we are given a fixed number *m* of processors.
- In this situation, a more meaningful question is how large can the total utilization of the periodic tasks be for the existence of a feasible assignment, that is, an assignment according to which the tasks on every processor are schedulable?
- Ans: It is shown that in a system containing *m* identical processors, each scheduled on a fixed priority basis, the first fit algorithm can always produce a feasible assignment if the total utilization U of the independent, preemptable periodic tasks is no greater than

$$U_{FF} = m(2^{1/2} - 1) = 0.414m$$

Note: In First fit algorithm tasks are assigned in processor one by one in an arbitrary order. The first task is assigned to processor P_1. After *i-1* tasks have been assigned , the ith task T_i is assigned to processor P_k if the total utilization of T_i and the tasks already assigned to P_k is equal to or less than U, but assigning T_i to any of the processors $P_1,P_2,\ldots,P_{k-1}$ would make the total utilization of tasks on the processor larger than U. The number of processors required by the assignment is the number of processors on which some task is assigned.

9.3 Multiprocessor Priority Ceiling Protocol

The multiprocessor priority ceiling assumes that the tasks and resource have been assigned and statically bound to processors and also the scheduler of every synchronization processor knows the priorities and resource requirements of all the tasks requiring the global resources managed by the processor. It is also assumed that the resources used by every job during nested critical sections lies on the same processor.



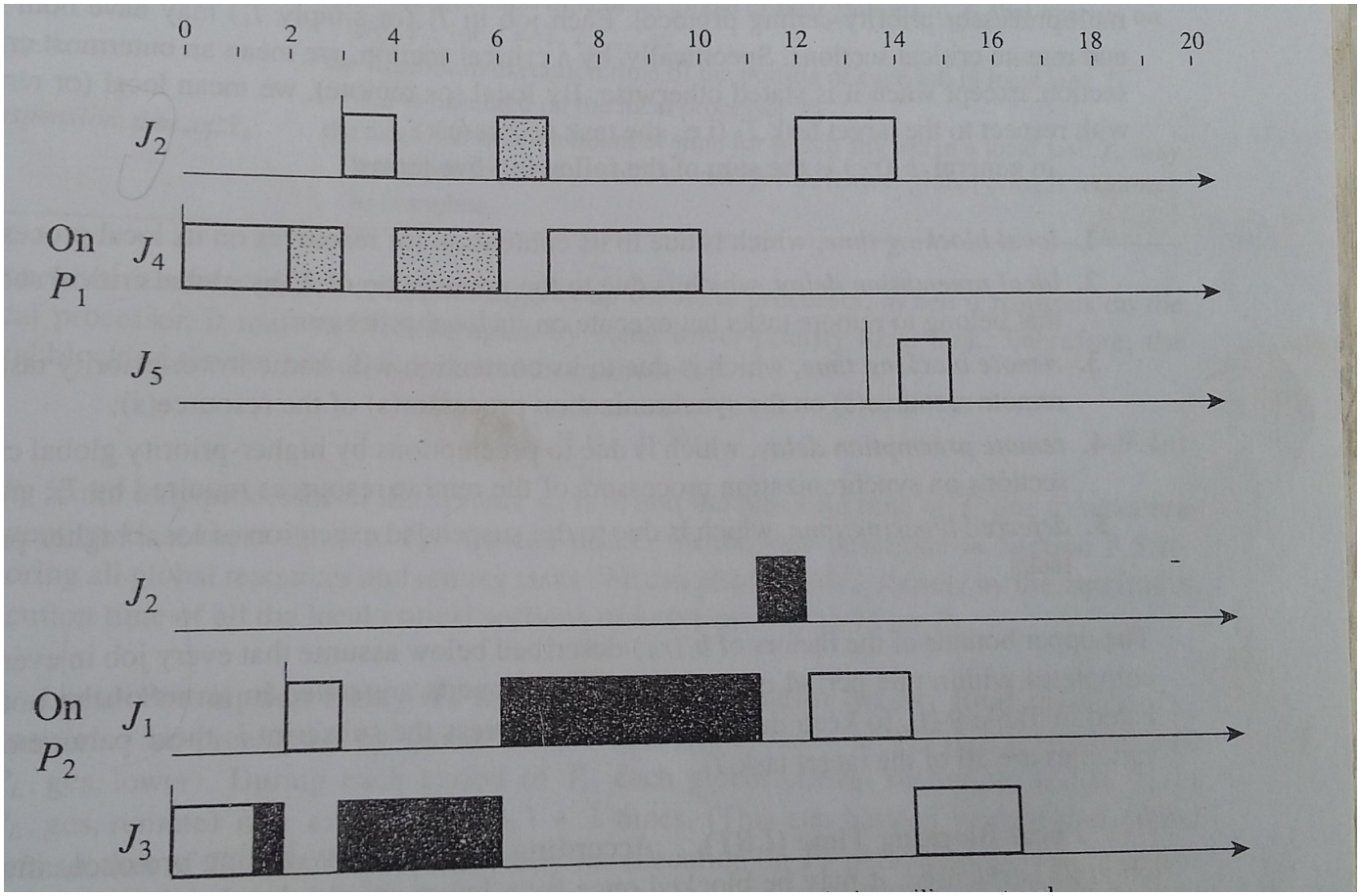
According to this protocol, the scheduler of each processor schedules all the local tasks and global critical sections on the processor on a fixed priority basis and controls their resource accesses according to the basic priority ceiling protocol. According to the MPCP model, when a task uses a global resource, its global critical section executes on the synchronization processor of the resource. If the global section of a remote task were to have a lower priority than some local task on the synchronization processor, these local tasks could delay the completion of the global critical section and prolong the blocking time of the remote task. For preventing this, the multiprocessor priority ceiling protocol schedules all the global critical sections at higher priorities than all the local tasks on every synchronization processor. This can be implemented in a system where the lowest priority π_{lowest} all the tasks is known. The scheduler of each synchronization processor schedules the global critical sections of a task with priority π_i at priority $\pi_i - \pi_{lowest}$.

As an example, consider a system where tasks have priorities 1 through 5, π_{lowest} is 5. A global critical section of a task with priority 5 is scheduled at priority 0, which is higher that priority 1. Also, the priority of a global critical section of a task with priority 1 is – 4 which is the highest priority in the system.

9.3.1 BLOCKING TIME DUE TO RESOURCE CONTENTION

The figure below shows the illustration that the types of blocking a job may suffer under the multiprocessor priority ceiling protocol.

The system has two processors P1 and P2. Jobs J2, J4 and J5 are local to processor P1, which is the synchronization processor of the resource dotted. Dotted is a local resource since it is only required by local jobs J2 and J4. Jobs J1 and J3 are local to processor P2, which is the synchronization processor of the resource black. Black is a global resource and is required by Jobs J1, J2 and J3. The jobs are arranged in decreasing order. The short vertical bar on the timeline of each jobs marks the release time of the job.



Here, J2 is directly blocked by J4 when J2 requests dotted at time 4. J1 is directly blocked by J3 when J1 requests black at time 3. The global critical section of J2 on P2 is delayed by the global critical section of the higher priority J1 in the time interval (7, 11]. The preemption delay thus suffered by J2 must be taken into account when the schedulability of J2 is to be determined. This delay is treated as J2's blocking time.

At time 11, J1 exits from its critical section. Its priority is lower than the priority of the global critical section of J2. As a result, J2 preempts J1 on P2 in the interval (11, 12]. The total delay experienced by a job as a result of preemption by global critical sections of lower priority jobs is also a factor in the total blocking time of the job.

Finally, a job can be delayed by a local higher priority job whose execution is suspended on the local processor when the priority job executes on a remote processor. This delay is another factor of the blocking time of the job. Here in this example, J2 is suspended at time 7. As a result, it is still not completed in (13, 14] and J5 released at time 13.2 cannot start until time 14.

9.4 Elements of Scheduling Algorithms for End-to-End Periodic Tasks

End to end scheduling has two essential components:

1. Protocols for synchronizing the execution of siblings subtasks on different processors so that precedence constraints among subtask are maintained.
2. Algorithms for scheduling subtasks on each processor.

According to the end to end scheduling approach, the fact that no subtasks in the system ever requires remote resource makes it possible for the scheduler on each processor to use any on uniprocessor scheduling algorithms and resource access protocols to schedule subtasks on the processor and control

their accesses to resources. It is even possible for the system to use a mixture of more than one scheduling strategy. For example, using dynamic priority schemes on some processors while using fixed priority schemes on the others, or make some processors priority driven and others clock driven.

Most of the real life systems use mixed scheduling strategies on different processors. For example, an end to end task whose subtasks compress video images periodically on a computer transmits video stream across a network whereas decompresses and displays the video on other computer. The computers at the two ends schedules the compression and decompression subtask differently from the way the network schedules the transmission of the video.

9.4.1 INTERPROCESSOR SYNCHRONIZATION PROTOCOLS

Just like a periodic task in a uniprocessor system, the jobs of the first subtask $T_{i,1}$ of every end-to-end periodic task T_i are released no less than p_i units of time apart.

When should jobs in subsequent sibling subtasks be released?

This question doesn't arise in uniprocessor systems, but is important in multiprocessor systems because how jobs in sibling subtasks are released critically affects in multiprocessor systems because how jobs in sibling subtasks are released critically affects the schedulability, completion-time jitter, and average response time of end-to-end tasks. We call a protocol that governs when the schedulers on different processors release the jobs of sibling subtasks an (interprocessor) execution synchronization protocol. Since there is no ambiguity, we call it simply a synchronization protocol.

A synchronization protocol is correct if it,

1. Never releases jobs in any first subtask before the end to end release times of the jobs, and
2. Never allows the violation of any precedence constraint among sibling subtasks.

There are two types of synchronization protocols:

1. Greedy Synchronization Protocol (work-conserving)
2. Non-Greedy Synchronization Protocol (nonwork-conserving)

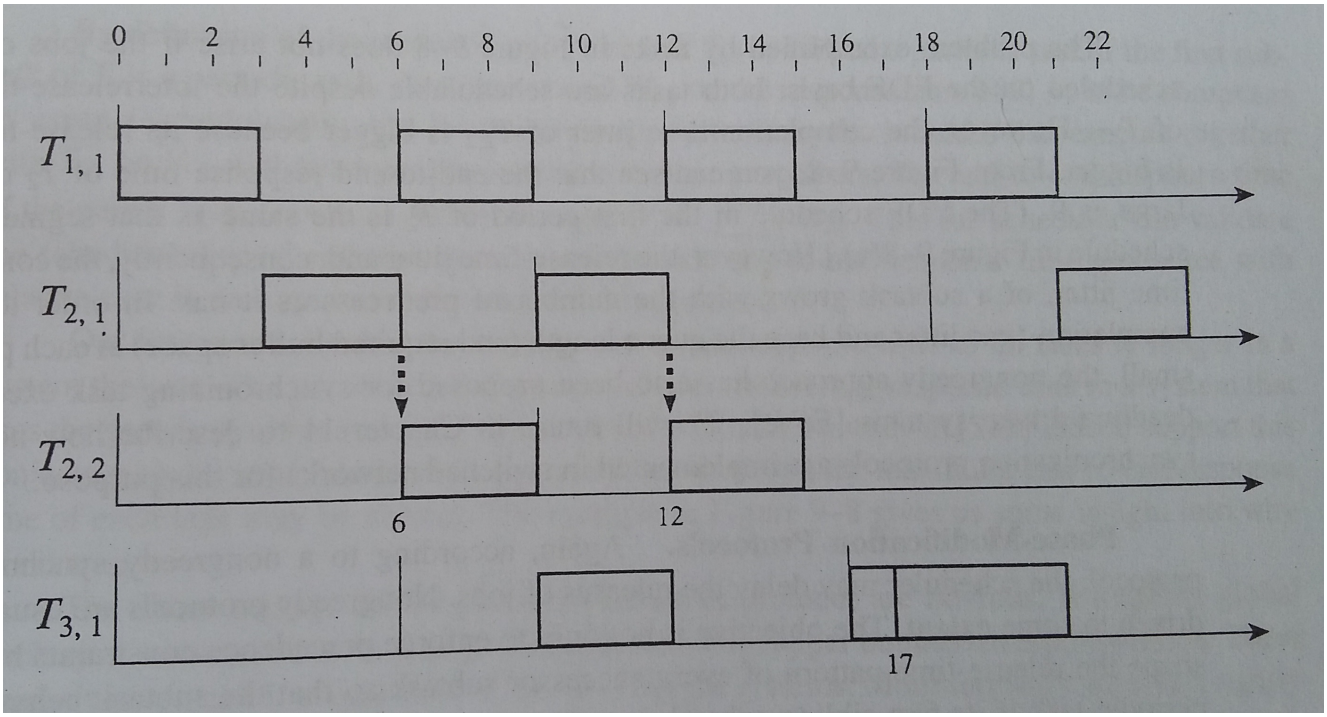
When sibling subtasks are synchronized according to the *greedy synchronization protocol*, the jth job $j_{i,k+1;j}$ of a subtask $T_{i,k+1}$ is released on $V_{i,k+1}$ as soon as its immediate predecessor $J_{i,k;j}$ completes on $V_{i,k}$, for every k=1, 2, …, n(i)-1. On the other hand, if the subtasks are synchronized according to a *non greedy synchronization protocol*, the completion time of $J_{i,k;j}$ is the earliest possible release time of $J_{i,k+1;j}$.

9.4.2 GREEDY SYNCHRONIZATION PROTOCOL

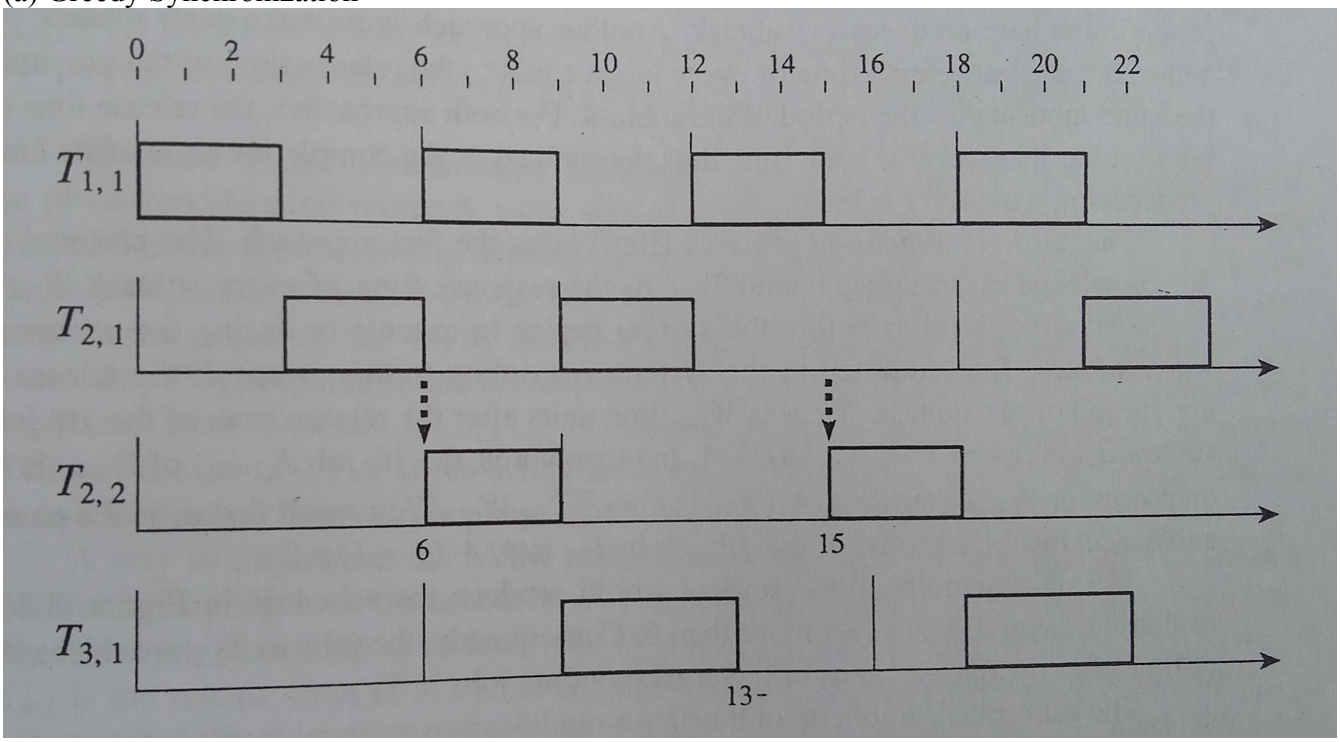
It is a commonly used synchronization protocol, especially in real time system. It can be implemented in many different ways. It can be implemented as follows: when the j^{th} job of $T_{i,k}$ completes on $V_{i,k}$, the scheduler of $V_{i,k}$ sends a synchronization signal to the scheduler of $V_{i,k+1}$ on which the successor subtask $T_{i,k+1}$ is executed. After receiving the synchronization signal, the scheduler of $V_{i,k+1}$ releases the corresponding job $J_{i,k+1;j}$immediately. Since, the multiprocessor system model used here takes into account interprocessor communication costs, the delay in synchronization signal delivery id let to be 0.

The greedy synchronization protocol is a simple protocol. It does not require any global clock synchronization. The only global information it requires is the identities of the immediate upstream processor and the next downstream processor in the visit sequence of each task which is executed on the processor. The greedy synchronization protocol yields the shortest average end to end response time of all tasks compared with non-greedy protocols. But the inter-release intervals of consecutive

jobs in a later subtask can be shorter than the period of the subtask. This can by illustrated by the following example.



(a) Greedy Synchronization



(b)nongreedy synchronization

Consider a system with two processor and three tasks: T_1 = (6, 3), T_3 = (6, 10, 4) do not have subtasks, and they are executed on P_1 and P_2. T_2 has Subtasks: $T_{2,1}$ = (9,3) executes on P_1 and $T_{2,2}$ = (9, 3) executes on P_2. The relative end to end deadlines of the tasks are equal to their respective periods. The tasks are given fixed priorities on both the processors. T_1 has higher priority on P_1 and $T_{2,2}$ has a higher priority on P_2. Figure (a) shows the schedule of the tasks when they are synchronized

on the basis of greedy protocol. The dashed vertical arrows at times 6 and 12 represents synchronization signal between processors. Since, the inter-release times of jobs in $T_{2,2}$ can be as short as 6, T_3cannot meet its deadlines. Figure (b) shows that if the jobs in $T_{2,2}$ are released periodically. T_3 always meets its deadlines.

The greedy synchronization protocol is unsuitable for hard real-time applications because tasks thus synchronized may have extremely large end-to-end response times when scheduled on a fixed-priority basis. Futhermore, there is yet no schedulability analysis method that can give sufficiently tight upper bounds to end-to-end response times of greedily synchronized, fixed-priority tasks. As a consequence, response time guarantees in fixed-priority systems are sometimes unduly pessismistic.

9.5 End-to-End Tasks in Heterogeneous Systems

Thus far, our discussion on end-to-end scheduling has assumed that a priority-driven scheme is used to schedule every processor. We now remove this restrictive assumption. It is not valid for most real-life systems. All but the simplest embedded systems contain different types of processors (e.g. CPUs, disks, and networks). Typically, a variety of approaches are taken to scheduling tasks on these processors.

Exactly which scheduling algorithms are used on individual processors and how these algorithms work are unimportant here. Whatever the choice is, it is essential that the scheduling algorithm used on every processor keeps the response time of every subtask on the processor bounded from above as long as the processor time demands (i.e. release-time patterns and execution times of jobs) of the subtasks on the processor satisfy the design constraint of the algorithm.

Hereafter, by a real-time scheduling algorithm, we mean specifically one that meets this requirement. When the processor time demand of every subtask $T_{i,k}$ on a processor meets the design constraint of the scheduling algorithm used by the processor, an upper bound $W_{i,k}$ to the subtask's response time is obtained on the processor without considering tasks that execute on other processors. Without loss of generality, it is assumed that the scheduler on every processor uses a real-time scheduling algorithm.

In a system of end to end periodic tasks, one of the way to keep the worst case end to end response time of every periodic task small and end to end schedulability analysis simple is to use a non-greedy interprocessor synchronization protocol that reshapes the release time pattern of every subtask to make the subtask periodic. The modified phase protocol (MPM) and release guard (RG) protocol are examples of such protocols. They do not require priority driven scheduling to be used on individual processors.

Corollary

In a system where a real time algorithm is used to schedule sub-tasks on every processor, an upper bound W_i to the end to end response time of any periodic task T_i synchronized to the MPM protocol or the RG protocol is given by

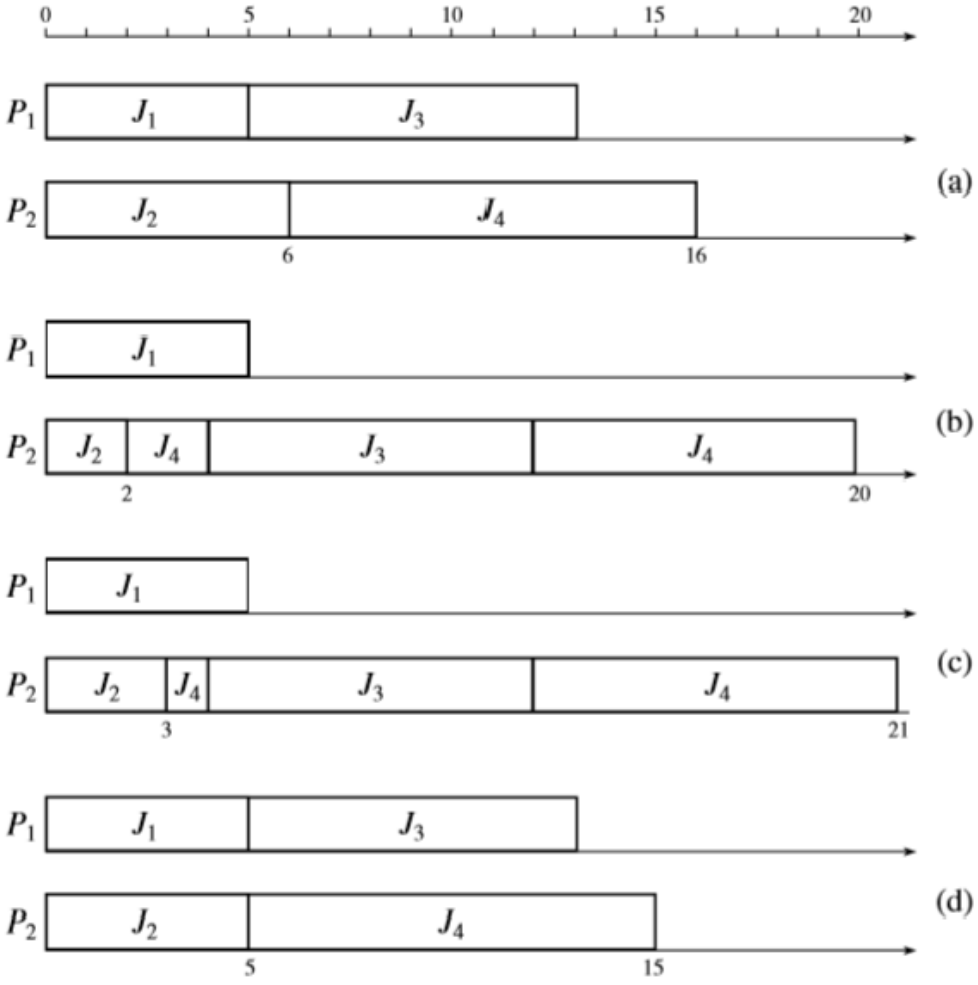
$$W_i = \sum_{k=1}^{n(i)} W_{i,k}$$

Where n(i) is the number of subtasks in T_i and the upper bound $W_{i,k}$ to the response time of every subtask $T_{i,k}$ is obtained by considering only subtasks on the same processor as $T_{i,k}$ and by treating every subtask $T_{j,l}$ as a periodic task whose period is equal to the period P_j of the parent task T_j.

9.6 Predictability and Validation of Dynamic Multiprocessor Systems

Predictable execution of a set **J** of jobs is defined in terms of 3 possible schedules of **J** according to the given scheduling algorithm: the maximum, minimal and actual schedules. If every job in **J** were to execute for as long as its maximum execution time (or short as its minimum execution time), the resultant schedule of **J** would be the maximal (or minimal) schedule. These schedules can be easily be constructed as long as the range of execution time of every hard real time job is known. The actual execution times of jobs are unknown, and therefore, the actual schedule of **J** is unknown.

	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



Example illustrating scheduling anomalies

The figure above illustrates that in a dynamic system, the actual start time $s(J_i)$ of a job J_i according to the actual schedule of **J** can be later or earlier than its start time $s^+(J_i)$ or $s^-(J_i)$ according to the maximal schedule or minimal schedule. The start time of J_i is said to be unpredictable if this condition

occurs. Similarly, the actual completion time $f(J_i)$ according to the actual schedule of J can be later than its completion time $f^+(J_i)$ of J_i according to the maximum schedule. The completion time of J_i is unpredictable if this is true. On the other hand, if $s^-(J_i) \leq s(j_i) \leq s^+(J_i)$ then J_i is start time predictable and if $f^-(J_i) \leq f(j_i) \leq f^+(J_i)$, then J_i is completion time predictable. The execution of J_i is predictable if J_i 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.

The above example shows that the execution of jobs in dynamic, priority driven multiprocessor is unpredictable in general, but this system can have predictable execution under several conditions. According to the maximal and minimal schedules, when one of these conditions are satisfied, the response time of each job gives the upper and lower bounds of the response time.

9.6.1 VALIDATION OF PREEMPTABLE/MIGRATABLE SYSTEM

A dynamic system in which jobs are scheduled on a priority driven manner on 'm' processors is called as a preemptable/migratable system. Every job that can be dispatched to execute on any processor can be preempted at any time and when preempted, they can be resumed on any processor. That is, jobs can be migrated among processors. A condition for predictability of a preemptable or migratable system is: the jobs have no precedence constraints and do not contend for resources.

Theorem

The execution of a system of preemptable/migratable jobs is predictable if all the jobs have fixed release times, are independent, and do not contend for resources.

9.6.2 OTHER CONDITIONS FOR PREDICTABILITY

The system shown in above figure is a preemptable or migratable system. In such system, jobs are scheduled in priority driven manner and every job can be dispatched to execute on any processor. However, once a job starts on a processor, it is constrained to execute on that processor. Job migration is too costly in most multiprocessor and distributed systems so, this scheduling strategy is commonly used. Under the theorem stated below, the execution of preemptable or migratable, independent jobs is predictable.

Theorem 1

If in a system of preemptable/nonmigratable, independent jobs, preemption can never occur, then the execution of the jobs is predictable.

Theorem 2

If according to the maximal schedule of a system of preemptable/nonmigratable independent jobs, no job is preemptable and the jobs start in the same sequence according to the maximal and minimal schedules, then their execution is predictable.

10 Real-time Communication

10.1 Model of real-time communication

The figure below shows the well-known model of distributed systems. The hosts are connected by a communication network or several interconnected networks. The top layers are simplified and all the entities above all the transport layer are called applications.

10.1.1 ARCHITECTURAL OVERVIEW

We focus on message exchanged among applications on different hosts. The source and destinations of every message are application tasks (or end users) residing on different hosts. The network interface of each host contains an input queue and an output queue which can also be referred to as input/output buffers or simply buffers. For the sake of correctness, it is assumed that these queues are jointly maintained by two local servers: the transport protocol (TP) handler and Network Access Control (NAC) handler. The TP handler interfaces with local applications and provides them with message transport services. The NAC handler interfaces with the network below and provides network access and messages transmission services to the TP handler.

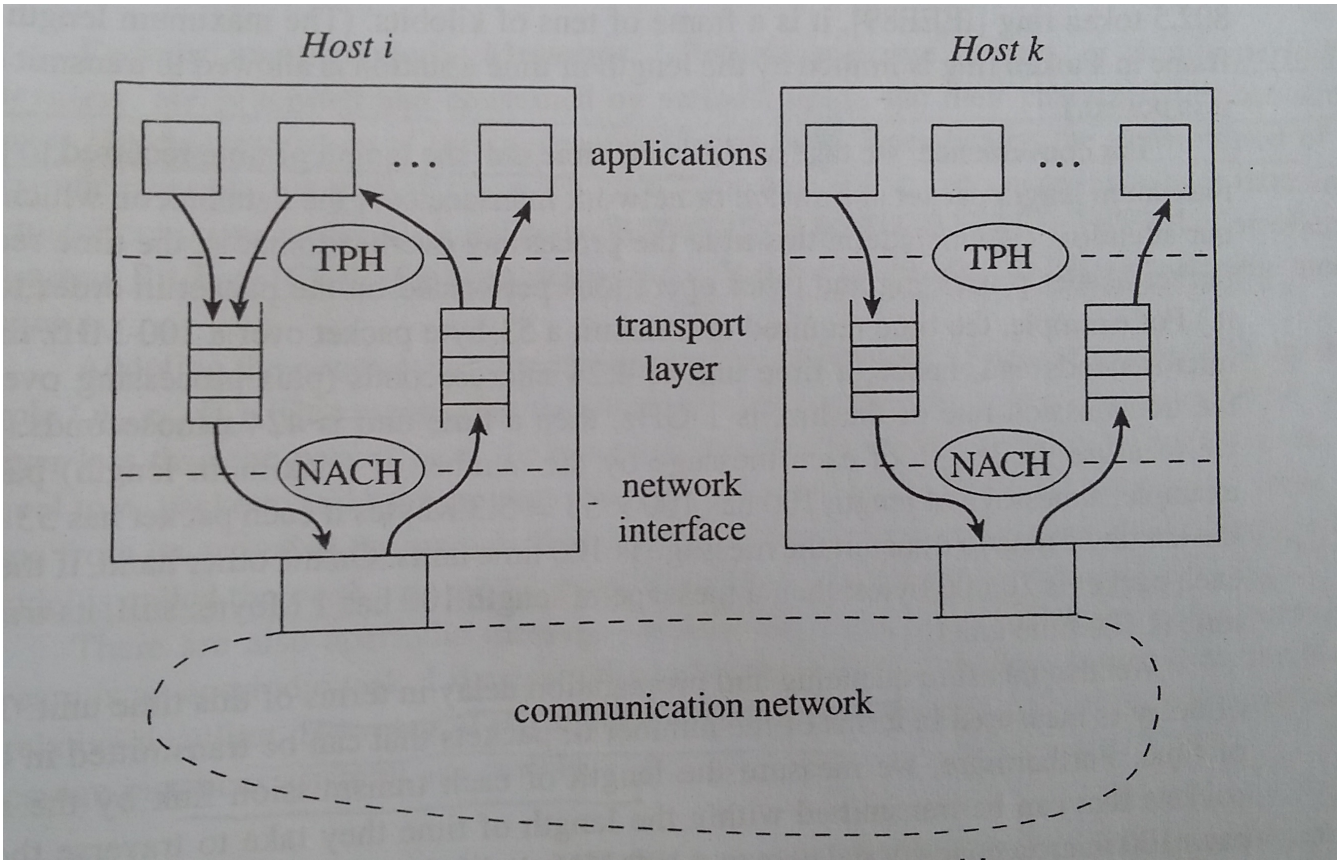


fig. A real time communication model

Figure shows the data paths (indicated by heavy arrows) traversed by messages in and out of two hosts. The circles marked TPH and NACH are TP and NAC handlers, respectively. When requested to send a message by a local application task, the source TP handler places the message in that output queue. From there, each outgoing message is delivered to the network under the control of the source NAC handler. After the message has traversed to the network, the destination NAC handler places the message in the input queue and notices the destination TP handler. Then, the destination TP handler

moves the message to the address space of the destination application task and notifies the application of the arrival of the message.

When there is no possibility of ambiguity or when there is no need to be specific; we use the terms source and destinations endpoints (or simply source and destination) to mean either the source and destination applications tasks, or the TP handlers, or the NAC handlers, or the entry and exit points of the network. The activities of sending a message can naturally be represented in the end-to-end model by a chain of jobs. The source and destination applications are the predecessor and successor of this chain, respectively. At the beginning and end of the chain are the source and destination TP processing jobs. In between them, each job that accesses the network or transmits the message becomes ready for execution after its immediately predecessor completes, possibly with some delay in its readiness introduced by the execution synchronization protocol used along the path.

10.1.2 Packets, Network Bandwidth, and Physical Size

Messages are typically fragmented into segments for their transmission through the communication network. Each segment is handled by the network as a basic transmission unit, and the transmission of the unit is nonpreemptable. Such a unit is called a frame, a packet, a cell, and the like, in different types of networks. We use the name packet to refer to all these basic transmission units. As an example, a packet is a 53-byte cell in an ATM network. In an IEEE 802.5 token ring [IEEE89], it is a frame of tens of kilobits. (The maximum length of a data frame in a token ring is limited by the length of time a station is allowed to transmit each time it is polled.)

For convenience, we take as the basic time unit the length of time required to transmit a maximum length packet at a switch or network interface over the data link on which we focus our attention. (We include in this time the processing overhead, that is, the time required for packet header processing and other operations performed on the packet in order to transmit it.) For example, the time required to transmit a 53-byte packet over a 100-MHz link is 4.24 microseconds and, hence, a time unit is 4.24 microseconds (plus processing overhead). If the transmission rate of the link is 1 GHz, then a time unit is 424 nanoseconds. Similarly, we measure the length of each message by the number of (maximum length) packets. For example, a message of length 100 has 100 x 53 = 5.3 Kbytes if each packet has 53 bytes, and the length of time to transmit the message is 100 time units. On the other hand, if the length of each packet is 10,000 bytes, then a message of length 100 has 1 Mbyte; still, its transmission time is 100 time units.

We also measure queueing and propagation delay in terms of this time unit: The length of delay is measured in terms of the number of packets that can be transmitted in that length of time. Furthermore, we measure the length of each transmission link by the number of packets that can be transmitted within the length of time they take to traverse the link. So, each 100 microseconds of delay over a 100-MHz link where each packet is 53 bytes in length is 100/4.24 = 23.6 time units because in 100 microseconds, we can transmit this many 53-byte packets over the link. but this delay is only one time unit if the packet size is 10,000 bits. Over a 1 GHz link. the same 100 microseconds is 236 or 10 time units, respectively. The use of this basic time unit allows us to take into account the transmission rate and packet size of each network implicitly and eliminates the need to have separate units for distance and time. Similarly, when our discussion focuses on one input or output link, the absolute value of its bandwidth is irrelevant; only how this bandwidth is allocated among the message streams on the link is. We therefore say that the link bandwidth is one (one packet per unit time). When we say that a message stream is allocated a bandwidth of $\hat{u}$, we always mean that the message is given the fraction $\hat{u}$ of the link bandwidth.

10.1.3 REAL TIME TRAFFIC MODELS

In real-time communications literature, the term real-time traffic typically means isochronous (or synchronous) traffic, consisting of message streams that are generated by their sources on a continuing basis and delivered to their respective destinations on a continuing basis. Such traffic includes periodic and sporadic messages. which require some degree of guarantee for on-time delivery. In addition, there are also aperiodic (or asynchronous) messages. Aperiodic messages have soft timing constraints and expect the system to deliver them on a best-effort basis.

We also refer to each of these types of messages as a message stream and denote it by M_i for some index i when it is necessary to distinguish it from other message streams. We use interchangeably the expressions that a message instance or packet arrives and that it is released for transmission. A message instance or packet arrives (or departs) at a point in time when the last bit in it arrives (or departs).

Periodic and Aperiodic Messages.

Periodic message streams, or simply periodic messages, are generated and consumed by periodic tasks, and their characteristics are similar to the characteristics of their respective source tasks. Specifically, the transmission of a periodic message is a periodic task. Examples are message streams carrying sensor data and actuator commands generated and consumed by data acquisition tasks and digital controllers. Constant Bit-Rate (CBR) digitized voices and videos are accurately modeled by periodic messages.

We denote a periodic message M, by the tuple (p_i, e_i, D_i). This means that the interarrival (interrelease) times of instances in M, are never less than the period p_i of the message, the maximum length of the instances in M, is equal to e_i packets, and each instance must be delivered to the destination within D_i units of time from its arrival at the source. D_i is the (end-to-end) relative deadline of M. This traffic model is called the peak rate model in real-time communication literature. There are also aperiodic message streams; the transmission of an aperiodic message stream is an aperiodic task. Like aperiodic tasks, an aperiodic message stream does not have a relative deadline. However, it is desirable to keep the average delay suffered by aperiodic message instances as small as possible.

Sporadic Messages.

A periodic task is a poor model for some applications. Similarly, a periodic message is an inaccurate model of any sporadic message stream that some threshold miss rate is never exceeded; some statistics and histograms on miss rate usually suffice. In earlier chapters, we assumed that no job is ever blocked or lost because there is no space in the ready job queue for it when it becomes ready for execution. This assumption is usually valid for computation tasks but often not valid in communication systems. It is not economically sound to put so many input/output buffers in all the network-access interfaces and switches that buffer overflow never occurs and hence this assumption holds. When a queue is full or when the queue length reaches a certain (drop) threshold, some packet(s) destined for the queue are dropped (i.e., discarded). As a consequence, some packets may be lost. The loss rate of a message stream or a set of message streams gives the fraction of all message instances (or packets) in the stream(s) that are dropped en route for flow and congestion control reasons. (Some packets and messages may also be discarded because they are found erroneous.)

Delay Jitter, Buffer Requirement, and Throughput.

While it is important that the end-to-end delay experienced by messages of interactive applications (e.g., teleconferencing) be acceptably small, this performance objective is relative unimportant for some isochronous traffic. For example, the performance of a video-on-demand system is not seriously

affected by the end-to-end delay suffered by packets in each video stream. In contrast, delay jitter and throughput are important.

The *delay jitter* of a message stream is the variation (e.g., the maximum or average absolute difference) in the delays suffered by different message instances or packets in the stream. There is no advantage to completing a job in a periodic or sporadic task early. We gain no advantage by delivering packets in a periodic or sporadic message early. In fact, it may be disadvantageous to do so. A packet that arrives too early to be processed by the destination must be buffered. Hence, a larger delay jitter of a message stream means that more buffers must be provided by the stream. For this reason, many algorithms and protocols for real-time communication are designed to keep not only the worst-case delay small but also delay jitter small. Fortunately, these performance measures can be minimized simultaneously by using some nongreedy synchronization protocol.

Finally, the rate of each message stream measures the throughput of the stream. Most of the scheduling and flow-control algorithms are rate-based, meaning that they are designed to provide each message stream with a guaranteed minimum throughput independent of the demands of the other message streams.

10.1.4 PERFORMANCE OBJECTIVES AND CONSTRAINTS

We want to measure the performance of scheduling, synchronization, and flow control algorithms used for real time communication and the performance of the performance of the resultant communication system along two dimension: from the points of view of the user and the system. The user is concerned with the on time delivery of periodic and sporadic messages and the average response time of aperiodic messages.

Miss rate: the fraction of all message instances or packets that are delivered to their destinations too late.

Loss rate: gives the fraction of all message instances in the stream that are dropped en route for flow and congestion control reasons.

Invalid rate: combination or miss and loss rate (sum of miss and loss rate)

Delay Jitter: the variation in the delays suffered by different message instances or packets in the stream.

Buffer Requirement: a packet that arrives too early to be processed by the destination must be buffered. So, a larger delay jitter of a message stream means that more buffers must be provided by the stream.

Throughput: the rate of each message stream measures the throughput of the stream.

10.1.5 Real Time Connections and Service Discipline

It is common to adopt the connection-oriented approach for real-time traffic. According to the connection oriented approach, a logical simplex connection from the source to the destination is set up for the transmission of each message stream (or set of message stream). All packets on each connection are sent along a fixed route. The chosen route remains in use until the connection is torn down or when some adaptation mechanism is invoked to alter the route.

Admission control and Connection Establishment

There are good reasons for using a fixed route for traffic on each connection. As examples, high speeds of modem networks make it impractical to route individual packets independently. More importantly, the use of a fixed route for each connection allows each switch en route, in addition to the

endpoints, to set aside the required bandwidth and buffer space for the connection so the network can provide some form of performance guarantee. This also enables control over packet transmission (e.g., traffic shaping) to be done on a per connection basis.

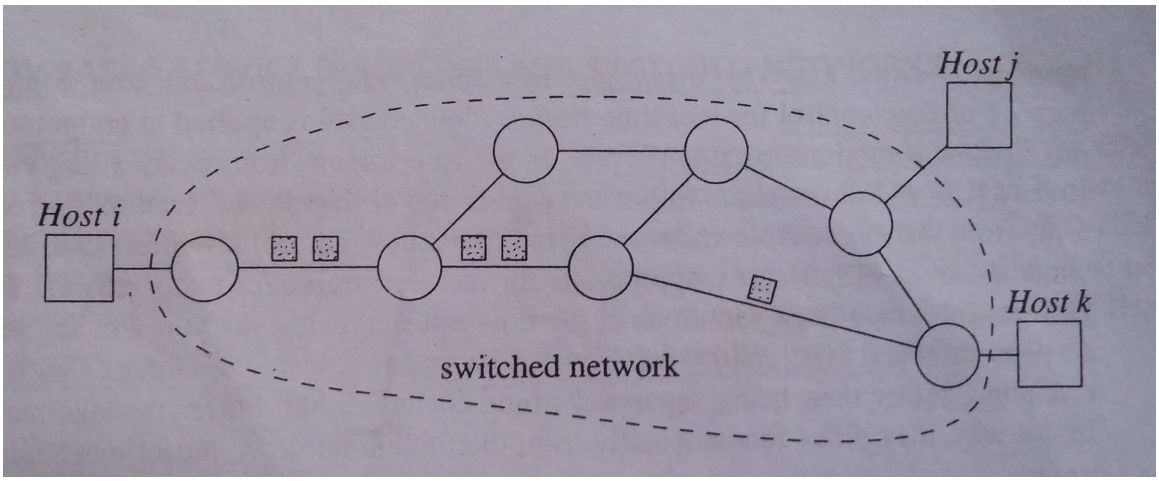
To request a connection, the requesting client (or clients) declares the characteristics of the message stream and the required performance of the connection. The former are defined by parameters of the message stream to be carried on the connection, for example, by the parameters of a leaky bucket filter that constrains the message stream. Collectively, these parameters are called the flow specification. The required performance is stated in terms of quality-of-service parameters such as end-to-end delay, jitter, and so on. The admission controller of each handler and switch along the chosen route uses these parameters as the basis of an acceptance test to determine whether to admit the connection. The connection is admitted if the requested quality of service can be delivered to the new connection without causing any deterioration of the quality of service of existing connections serviced by the handler or switch.

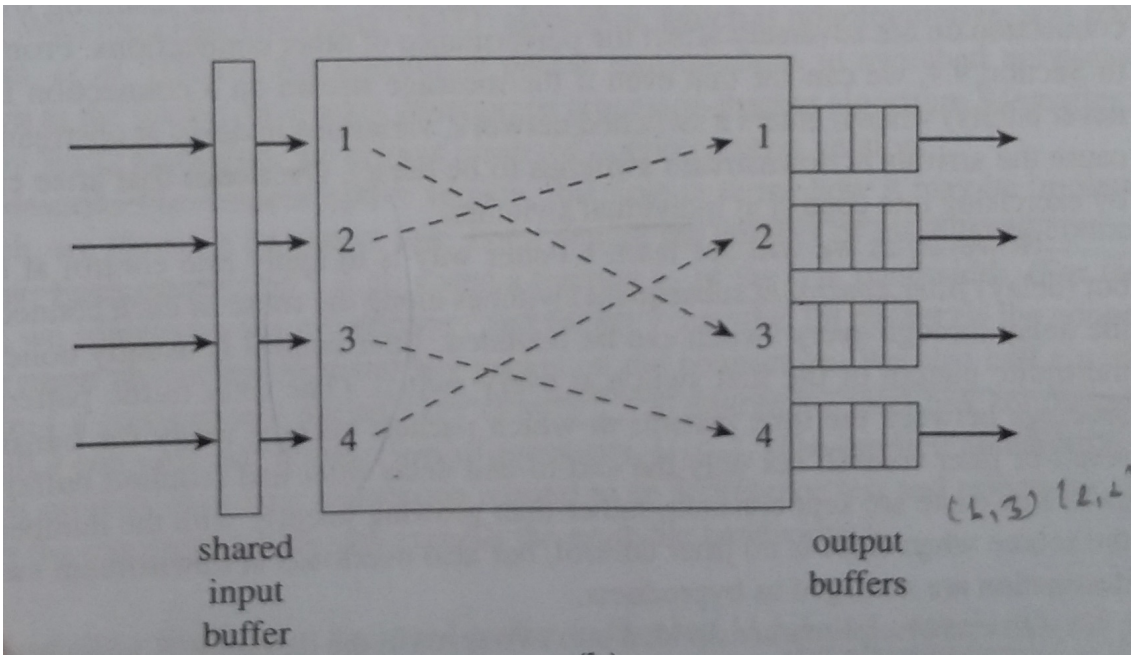
Packet Switched Networks

The diagram in Figure 11-2(a) illustrates such a network. The circles in the diagram represent switches. Figure 11-2(b) shows a m x m switch; it has m input links and m output links, both called links 1, 2, ... , m. The switch routes packets on its input links to its output links.

We can represent every switching (i.e., routing) pattern by a permutation of the m-tuple (1, 2, ... , m). A number i at position k means that the switch is configured to route a packet coming on the input link i to (the queue of) the output link k at the same time when it is routing packets on the other input links to other output links as specified by the permutation. As an example, for a 4 x 4 switch, the 4-tuple (2, 4, 1, 3) means that a packet on input link 2 goes to the queue of output link 1, a packet on input link 4 goes to the queue of output link 2, and so on. This is the switching pattern depicted by Figure 1 1-2(b). The switch is nonblocking, meaning that every permutation represents a possible switching pattern.

Without loss of generality, we assume that switches are output buffered. At each switch, there is a buffer pool for each of its output links, holding packets that are queued for transmission on the link. Once the switch routes a packet to this queue, the packet waits in the queue until the scheduler schedules it for transmission, and then it is transmitted to the next hop at the other end of the output link. The amount of time the switch takes to route packets from its input links to its output links is negligibly small. Consequently, the time a packet takes passing through a switch is essentially equal to its waiting time at the output queue (i.e., its output queueing delay) plus the packet transmission time (i.e., the time required to transmit the packet onto the output link and hence the length of the packet). We call this sum the *per hop delay* of the packet.





Service Discipline

The combination of an acceptance test and admission control protocol, and synchronization protocol and a scheduling algorithm used for the purpose of rate control, jitter control and scheduling of packets transmission is called a service discipline. In essence, rate control and jitter control serve the purpose of flow-control for real-time traffic. Flow control is applied at endpoints and within the communication network to prevent an entity upstream from overloading entities down-stream. (Up- and downstream follow the convention of river flow: The direction of traffic flow away from the source is downstream.) Traditional flow-control schemes, such as the sliding-window protocol are inappropriate for real-time traffic for many reasons. For example, they can introduce large variations in the flow rate and delay. Rate and jitter control mechanisms for real-time traffic are integral parts of a service discipline, rather than being separated from scheduling and buffer management functions. They differ fundamentally from the traditional flow-control mechanisms for data traffic.

Specifically, the term rate control is used in the communications literature to mean load management; the purpose is to ensure that the bursty traffic and resulting overloads on any connection do not adversely affect the performance of other connections. If the message stream on a connection is periodic (i.e., never bursty) when it enters a switched network, variations in delay at upstream switches may cause the arrivals at downstream switches to be bursty. Overloads that arise can be managed by exercising rate control at individual switches. However a better way is to apply rate control at the first switch but (delay) jitter control at subsequent switches along the route of each connection whenever the delay through every switch can be bounded. Jitter control is usually done by preserving the traffic pattern of the first switch at every switch. (The term traffic pattern refers to the spacing between the time instants at which packets become ready for transmission.) As a result of jitter control, not only the end-to-end delay jitter and required buffer space at each switch en route are kept constant, rather than growing linearly with the number of hops from the source when there is no jitter control, but also overloads at downstream switches and the destination are managed as byproducts.

Service disciplines are divided into two types in the literature.

- rate-allocating,
- rate-controlled

Rate-allocating disciplines allow packets on each connection to be transmitted at higher rates than the guaranteed rate (i.e., the minimum rate required to meet the guaranteed delay and throughput of the connection), provided the switch can still meet the guarantees to all other connections. A service discipline is rate-controlled if it ensures each connection the guaranteed rate but never allows packets on any connection to be sent above the guaranteed rate. In general, a bandwidth-preserving server that claims all the background time is rate-allocating. The sporadic/background server and the total bandwidth server are examples. The constant utilization server is rate-controlled. So rate-allocating algorithms are work-conserving (i.e., greedy), while rate-controlled algorithms are nonwork-conserving (i.e., nongreedy).

10.2 Priority-based service disciplines for switched networks

According to the priority based service discipline, the transmission of ready packets are scheduled priority driven manner. Among this, the well-known are Delay Earliest Due-Date (Delay-EDD), weighted fair queuing, jittered EDD and Rate Controlled Static Priority (RCSF).

10.2.1 Weighted Fair Queuing Discipline

It is a packet by packet generalized processor sharing algorithm. It is a type of rate allocating service discipline that provides each flow with at least it's proportional fair share of link capacity & isolates the timing between flow.

Let n denote the number of established connections on the link. Each connection *i* is allocated a fraction $\hat{u}_i$, of the link bandwidth. Let $U = \sum_{i=1}^{n} \hat{u}_i$ denote the total link bandwidth allocated to all n connections. Without loss of generality, U < 1. (The implicit assumption is that the switch subjects each new connection request to an acceptance test and rejects the request whenever the requested bandwidth exceeds the available bandwidth.)

10.2.1.1 SCHEDULING PACKETS

Assume that the switch is output buffered. The buffer space that holds all packets on connection *i* being transmitted or waiting to be transmitted is in essence a FIFO queue for the connection. Each connection-*i* packet is placed at the end of this queue upon arrival without scheduler attention. A packet becomes ready for transmission when it reaches the head of the queue, that is, when all connection-*i* packets that arrived before it have been transmitted. Each packet is removed from the queue when its transmission completes. We say that a connection *i* is idle when no connection-*i* packet is waiting or is being transmitted; otherwise, the connection is back-logged.

For the purpose of scheduling ready packets on backlogged connections, the scheduler keeps a priority queue. Each backlogged connection (say connection *i*) has an entry (fn_i, *i*) in this queue. The entry gives the *finish number* fn_i of the connection, or more precisely, the finish number of the ready packet on the connection. It also gives the ID *i* of the connection. The entries are sorted in order of finish numbers: the smaller the number, the earlier in the queue. Therefore, we call this queue the SFN (Smallest Finish Number first) queue. The only exception to this order is the entry of the packet currently being transmitted. Since scheduling is nonpreemptive, the scheduler keeps this entry at the head of the SFN queue, even when the finish numbers of new entries inserted during the packet's transmission are smaller. Ready packets on backlogged connections are transmitted in the order given by this queue.

Scheduling according to the WFQ algorithm is done as follows:

- When the first packet in a busy interval of the output link arrives, the scheduler computes its finish number and enters this number and connection ID in the SFN queue. It commences the transmission of the packet immediately.
- During a link busy interval, the scheduler computes the finish number of each packet that arrives on an idle connection and inserts the corresponding entry into the SFN queue.
- For as long as the link is busy, whenever the transmission of a packet (say a connection-i packet) on the link completes, the packet is removed from the connection's FIFO queue and the entry containing the finish number of this packet is removed from the head of the SFN queue. The scheduler chooses the-next packet to transmit in the following manner.
  i. If connection i is still backlogged, the scheduler computes the finish number of its new ready packet and inserts this number and connection ID in the SFN queue.
  ii. It then commences the transmission of the ready packet on the connection identified by the first entry in the SFN queue.

10.2.1.2 Computing Finish Numbers

It is necessary for the scheduler to maintain the current values of the total bandwidth U_b of all backlogged connections and the finish number FN of the link. t denotes the current time, and t_{-1} denotes the previous time instant when FN and U_b were updated. There are *n* existing connections on the output link of interest here. By the length of a packet, we mean the time required to transmit the packet at the transmission rate of the output link. Also, we normalize time so that the maximum length of all packets transmitted over the link is 1.

Rules for Computing the First Finish Number of a Link Busy Interval

I1 For as long as the link is idle, $FN = 0$, $U_b = 0$, $t_{-1} = 0$, and finish number fn_i of connection i, for every $i = 1, 2, \ldots, n$, is 0.

I2 When the first packet of length e arrives and starts a busy interval of the link,

(a) set $t_{-1} = t$; and

(b) the packet being a connection-i packet, increment U_b by $\tilde{u}_i$, compute $fn_i = fn_i + e/\tilde{u}_i$, and insert the entry (fn_i, i) in the *SFN* queue.

Rules for Computing Subsequent Finish Numbers during a Link Busy Interval

R1 For every i, when a connection-i packet arrives at t during a link busy interval, if connection i was idle prior to this arrival,

(a) increment FN by $(t - t_{-1})/U_b$, compute $fn_i = \max(FN, fn_i) + e/\tilde{u}_i$, and insert the entry (fn_i, i) in the *SFN* queue; and

(b) set $t_{-1} = t$ and increment U_b by $\tilde{u}_i$.

R2 For every i, when the transmission of a connection-i packet completes,

(a) if the connection remains backlogged, increment fn_i by $e/\tilde{u}_i$, where e is the length of the new ready connection-i packet, and insert the entry (fn_i, i) in the *SFN* queue;

(b) if connection i becomes idle,

i increment the link finish number FN by $(t - t_{-1})/U_b$; and

ii set $t_{-1} = t$, and decrement U_b by $\tilde{u}_i$.

Maximum Delay and Queue Length at the First Switch. The delay between when a packet becomes ready (i.e., when it reaches the head of the FIFO queue of its connection) and when its transmission completes is called its *latency*. Corollary 7.9 allows us to conclude that the latency of a packet on connection i that is allocated the fraction $\tilde{u}_i$ of link bandwidth is bounded from above by

$$L_i = e_i/\tilde{u}_i + 1 \qquad (11.1a)$$

where e_i is the maximum length of all connection-i packets. The first term in the right-hand side gives the amount of time required to transmit the largest connection-i packet on a slower link of bandwidth $\tilde{u}_i$ times that of the physical link. The second term accounts for the effect of nonpreemptivity, and the maximum blocking time due to nonpreemptivity is equal to 1. Because of the rate-control function provided by a WFQ scheduler, this upper bound holds independent of the arrival rates of all other connections.

Maximum End-to-End Delay and Delay Jitter. We now look at a connection i that transverses ρ switches. Suppose that only the traffic into the first switch en route is shaped to meet the (u_i, E_i) leaky bucket constraint. The intermediate switches do not do any traffic shaping.

The end-to-end delay of a packet of length e through ρ homogeneous switches under the WFQ algorithm is given by

$$W_i(\rho) = \frac{E_i + \rho e}{\tilde{u}_i} + \rho$$

10.3 Weighted Round Robin Service Discipline

Weighted Round-Robin (WRR) algorithms are unsuitable for scheduling precedence-constrained jobs. However, messages in a switched network are pipelined through the switches; earlier packets in a message are sent from a switch without having to wait for the arrivals of later packets. For scheduling message transmissions through switches, and for pipelined jobs in general, the WRR scheme offers an excellent alternative to time-driven and priority-driven schemes. Its major advantage over the time-driven scheme when used for end-to-end scheduling in a network or distributed system is that it does not require globally synchronized clocks. The major advantage of the WRR scheme over priority-driven schemes is that it does not require any sorted queue. These advantages make the WRR scheme a good practical choice, especially for constant bit-rate traffic such as uncompressed voices. Indeed, some forms of the WRR scheme have been implemented in ATM networks.

In a round-robin scheduling, the jobs are placed in a FIFO queue. The job at the head of the queue executes for one time slice if it does not complete within the time slice. If it does not complete within a time slice, it is preempted and put at the back of the queue. If there are n jobs in the queue, then each job gets on slice every n time slots.

But in the weighed round robin scheduling, each job gets a weight wt_i. A job with the weight wt_i executes for wt_i time slices each round robin and the length of the round is equal to

10.3.1 Greedy WRR Discipline

One of the most widely used WRR service discipline is called Greedy WRR. In this discipline during connection establishment, the scheduler at each switch assigns to the new connection *i* a weight of wt_i. It means that connection i is allocated wt_i slots in each round during which message packets on all existing connections sharing the same output link are transmitted in turn. Each slot has length 1, the time to transmit a maximum-size packet.

10.3.2 THROUGHPUT AND DELAY GUARANTEES

Consider a constant bit rate periodic message $M_i = (p_i, e_i, D_i)$ where p_i is minimum interarrival time of the message, e_i is size of each message and D_i is maximum acceptable end to end delay.

During each round, if more than wt_i packets on connection i are waiting, wt_i packets are transmitted. If wt_i or fewer packets are waiting, all packets on the connection are transmitted. After the scheduler completes the transmission of packets on connection i, it proceeds to transmit packets on connection i + 1 in the same manner; except, of course, the maximum number of packets on connection (i + 1) transmitted per round is wt_{i+1}. In this way, each connection i is guaranteed wt_i slots in each round.

A design parameter of each switch is the maximum number of slots RL (round length) per round. At all times, the sum of weights of all n connections on the output link is no greater than RL, that is, $\sum_{i=1}^{n} wt_i \leq RL$. Each connection i is guaranteed the throughput rate of wt_i/RL, and this guarantee is never violated due to overloads on other connections.

The upper limit on the round length RL (and consequently the maximum weight as-signed to any connection) is imposed by the delay guarantee that the switch provides to each connection. A packet may have to wait for an entire round even when there is no other packet on the connection waiting at the switch when it arrives. Hence, to guarantee that every instance of every message be transmitted within a period of the message, RL must satisfy

$$RL < P_{min} \qquad 11.7a$$

where P_{min} is the minimum of the periods of messages on all existing connections. Moreover, the weight wt_i of each connection i must satisfy the following constraint.

$$wt_i > \left\lceil \frac{e_i}{\lfloor pi/RL \rfloor} \right\rceil \qquad 11.7b$$

That each instance of M_i is transmitted within one period follows directly from the fact that there are at least $\lfloor pi/RL \rfloor$ rounds within each period of Mi and at least e_i packets are transmitted within this many rounds. On the other hand, if connection i were assigned a smaller weight than the lower bound in $\lfloor pi/RL \rfloor$, the throughput provided to connection i would be lower than the declared peak rate e_i/p_i of its message. The delay of packets on the connection may grow without bound.

Since each message in Mi takes at most ei/wti rounds to complete, the delay of the message through a switch is at most equal to $ei/wti\ RL$. A packet sent in a round at a switch is eligible for transmission at the next downstream switch. Therefore, the message suffers this delay only at the first switch but only one more additional round of delay through each of the subsequent switches. Therefore, the end-to-end delay W_i of a message on a connection i that traverses ρ switches is bounded from above by following equation when the round lengths at all ρ switches en route are RL.

$$W_i \leq (\lceil e_i/wt_i \rceil + \rho - 1)RL \leq p_i + (\rho - 1)RL \qquad (11.8)$$

10.3.3 CONNECTION ESTABLISHMENT

Because weights of all connections on an output link depend on the round length used for the link, a change of the latter may require changes of the weights. It would be too costly for the scheduler to adjust the round length dynamically as a part of acceptance test and admission of new connections. For this reason, the round length used at each switch is chosen a priori based on the characteristics of all types of traffic the switch is designed to carry. In particular, the round length used for each output link satisfies the constraint Eq. (11.7a) for all anticipated connections.

Admission control and connection establishment in a network that uses the WRR discipline are simple. Again, the request for each new connection provides as part of the flow specification the parameters p_i, e_i, and Di of the message stream M_i to be carried by the new connection i. The scheduler of each switch computes the weight wt_i required by the connection according to the expression in the right-hand side of Eq. (11.7b). If the sum of weights assigned to all existing connections is no more than $RL - wti$, the scheduler accepts the connection and allocates wt_i slots per round to the connection. Otherwise, the scheduler rejects the connection. If all ρ switches on the route selected for the connection accept the connection and the end-to-end delay computed according to Eq. (11.8) does not exceed the end-to-end relative deadline Di of Mi, the connection is admitted and established. Otherwise, the connection establishment fails and the schedulers at switches en route free the slots assigned to the connection when informed of the failure.

10.3.4 Delay Jitter and Buffer Requirements

The WRR discipline is greedy and, hence, rate allocating. The actual throughput of a connection can exceed its guaranteed rate. Like all other greedy disciplines, the greedy WRR discipline does not control delay jitters. The end-to-end delay of a message can be as small as $ei + \rho - 1$. (This occurs when every packet is transmitted in 1 unit of time after its arrival at all p switches en route.) Therefore, the end-to-end delay jitter of can be as large as $pi — ei + (\rho — 1)(RL — 1)$.

At the kth switch from the source of a connection i, the end-to-end delay jitter is $pi — ei + (k — 1)(RL — 1)$. Because of this delay jitter, the amount of buffer required at switch k for connection i to prevent packet loss is $(1 + \lceil (k — 1)(RL — 1)/\, pi \rceil)ei$ packets. As expected, this amount grows linearly with the number of hops from the source. Again, the way to fix this problem is to do some form of traffic shaping at each switch, and this is the motivation of the various nongreedy variants of the WRR discipline.

10.4 Medium access-control protocols of broadcast networks

In our terminology, the transmission medium of a broadcast network is a "processor." A Medium Access-Control (MAC) protocol is a discipline for scheduling this type of processor. Scheduling the transmission medium is done distributedly by network interfaces of hosts in the network. This is why MAC protocols typically either look different or are in fact different from the scheduling algorithms described earlier. Some MAC protocols are designed for distributed scheduling by loosely coupled schedulers, and they are fundamentally different from centralized scheduling algorithms.

We call the transmission medium of a network simply the network. A *station* on the network is the network interface of a host on the network. Each basic unit of data is transmitted nonpreemptively over the network a packet. The statement that a station listens means that it monitors the data on the network, and that it hears or sees means that it senses the presence of data or some special bit patterns on the network. Each station maintains its own outgoing and incoming queues as shown in Figure 11-1. Except for where it is stated otherwise, we assume that the header of each packet contains the

address of the destination station (or the ID of the message). When a station is not transmitting, it listens on the network. It copies the packet into its buffer (i.e., it receives the packet) when it hears its own address in the packet header (or the ID of a message it is prepared to receive). We focus here on the real-time aspects of the MAC protocols.

10.4.1 Medium Access Protocols in CAN and IEEE 802.5 Token Ring

We measure the size of a broadcast network in term of the ratio of the network round-trip delay to the transmission time of a maximum length packet. If this ratio is small (say on the order of 10^{-2} or smaller), every station can hear the transmission of every other station almost immediately after the transmission starts. We say that the network is small when this ratio is small. In a small network, circulating control information among the stations takes a small fraction of packet transmission time. The stations on the network can coordinate their decisions and actions without incurring significant performance penalty. By so doing, they can carry out a centralized scheduling algorithm in a distributed manner.

10.4.1.1 FIXED-PRIORITY SCHEDULING IN CAN

Controlled Area Networks are the examples of small networks. CANs are used to connect component of embedded controllers. For example, an automobile control system, whose components control the engine, the brakes, the environment and so on. At the transmission rate of one Mbits per second, the end to end length of a CAN must be no greater than 50 meters. It means, within a fraction of a bit-time after statin starts to transmit, all the stations on the network can hear the transmission. Therefore, the network functionally behaves like a local bus. The output of all the stations are wire-ANDed together by the bus: the bit on the network during a bit-time is a logical 0 if the output of any station is a 0 and a logical 1 only when the outputs of all stations are 1. The MAC protocol for CAN take advantage of this feature.

Each message stream transmitted in a network has a unique message ID. Each packet in the stream begins with this ID, with the most significant bit first. A station on the network determines whether to receive a packet based on the ID number of the packet. Finally, a packet contains 1 to 8 bytes of data.

CAN MAC protocol is a CSMA/CD (Carrier-Sense Multiple Access/Collision Detection) protocol. A station with a packet to send waits until it hears that the network is idle and then commences to transmit the ID number of the packet. At the same time the station listens. Whenever it hears a 0 on the network while it is transmitting a 1, it interrupts its own transmission. This way, network contention is resolved in favor of the packet with smallest ID among all contending packets.

So, the packets in each message stream are given a fixed priority that is equal to the ID of the message. The smaller the ID the higher the priority. Packets are transmitted non-preemptively based on their priorities.

10.4.1.2 PRIORITIZED ACCESS IN IEEE 802.5 TOKEN RINGS

In an IEEE 802.5 token ring network, packets are transmitted along a circular transmission medium in one direction. A station transmits a packet by breaking the network and placing its packets on the output link to the network. As the packet circulates around the network, the station identified by the destination address in the packet header copies the packet. When the packet returns to the source station, the station removes the packet from the network.

Polling

Network contention is resolved by a polling mechanism called token passing. For polling, each packet has an 8-bit Access Control (AC) field in its header. One of the bits in AC field is called the token bit. A station can determine whether the network is busy or free by examining tokenbit. As a polling

packet circulates around the ring, the stations are polled in a round robin manner in the order of their physical locations on the ring. The polling packet is called the free token or simply token when there is no possibility of confusion.

When a free token reaches a station that has outgoing packets waiting, it can seize the token (i.e., stop it from circulating) under the condition. After it seizes the token, it changes the token bit to mark the token busy and transmits packet(s) in its outgoing queue. When the station completes its transmission, it generates a free token and transmits the free token downstream. Afterwards, it continues to relay the token and data packets of other stations as long as its outgoing queue is empty.

Priority Scheduling

Prioritized access is made possible by using the two groups of 3 bits each in the AC field: Their values represent the token priority $\prod_T$ and the reservation priority $\prod_R$. Specifically, the 3 token priority bits give the priority of the token. A station can seize the free token only when its outgoing packet has an equal or higher priority than the token priority $\prod_T$. When a station seizes the token, it leaves the token priority unchanged but sets the reservation priority to the lowest priority of the network. It then marks the token busy and puts the token in the header of the packet and transmits the packet.

Reservation for the next turn to use the network is done distributedly during the trans-mission of a data packet in the following manner. As a data packet circulates around the ring, a station with outgoing packets can make a reservation for future use of the network by setting the reservation priority bits in the AC field to the highest priority π of its outgoing packets, if π is higher than $\prod_R$. When the data packet returns to the source station, the reservation priority $\prod_R$ is equal to the highest priority of all packets waiting at all stations to be transmitted on the network.

When a source station removes its own packet from the network, it saves the reservation priority carried in the packet. Suppose that when the source station transmits a free token, it sets the token priority of the token to this reservation priority or the highest priority of its outgoing packets, whichever is higher. In this case, the priority arbitration mechanism allows the stations to jointly carry out any fixed-priority scheduling algorithm. [As an example, it is common to divide messages into classes, depending on whether they are synchronous (i.e., periodic) or asynchronous (i.e., aperiodic), urgent or not urgent. Packets in messages of any class can be given the same priority.] Even the EDF algorithm is possible with this priority arbitration mechanism if the round-trip delay of the network is small and the clocks of the stations are synchronized.

Schedulability Analysis

The amount of time each packet occupies the network is equal to its transmission time plus the round-trip delay it takes to return to the source station. For the purpose of schedulability analysis, we use this amount as the "execution time" of the packet transmission. The round-trip delay includes the propagation delay of the transmission medium itself, plus the delays introduced by stations on the network. This delay is usually on the order of 10^{-2} or less of the packet transmission time. By including this delay and the time used to transmit the packet header, the checksum, and the delimiting bits in the execution time, the overhead introduced by framing the user data according the MAC protocol can be taken into account. In addition, we need to take into account the following three factors.

- **Context Switching**: A context switching time is equal to the amount of time required to transmit a free token, plus the round-trip delay of the network, which is an upper bound of the time the token takes to reach the station whose outgoing packets has the highest priority among all outgoing packets during the transmission of the latest data packet.

- **Blocking**: Since packets are transmitted non-preemptively, we need to take into account the blocking time due to non-preemptivity. A higher priority packet that arrives at a station just after the header of the current data packet passed the station may need to wait for a lower priority packet. The blocking delay caused by this priority inversion must also be taken into account. Hence, the total blocking time is equal to twice the maximum execution time.
- **Limited priority levels**: Since the network provides only eight priority levels, schedulability loss should also be take into account

10.5 Internet and Resource Reservation Protocol

We know that rate and jitter control is an integral part of good service disciplines for real-time traffic in switched networks. If every switch reserves resources (i.e., link bandwidth and buffer space) for every connection routed through it and exercises rate and jitter control of traffic on the connection, congestion within the network and at the destination can be prevented and real-time performance guaranteed. Similarly, in order to deliver the desired quality of service to users through interconnected networks, hosts and routers must re-serve resources needed to ensure quality. While the onset of congestion in networks carrying only data traffic tends to be gradual, allowing time for congestion-control mechanisms to react and adapt, congestion of real-time (video and audio) traffic can build up quickly in integrated service networks. Traditional congestion-control mechanisms do not work well, whereas admission control and resource reservation provide an effective means to prevent congestion. How hosts and routers (and switches) operate and interact for the purpose of resource-reservation is governed by a resource-reservation protocol.

10.5.1 Issues in Resource Reservation

There are four closely related issues that a resource-reservation protocol must deal with but we have ignored thus far. They are the multicast nature of communication, heterogeneity of the destinations, the dynamics of multicast group membership, and the relationship between resource reservation and routing and admission-control functions.

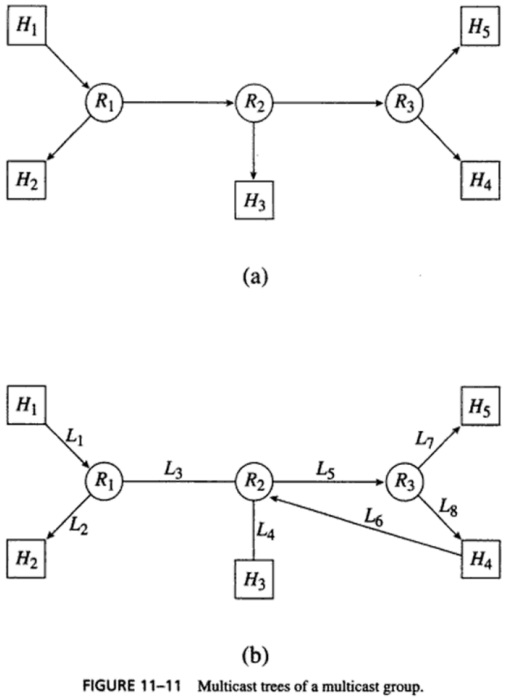


FIGURE 11–11 Multicast trees of a multicast group.

1. Multipoint-to-Multipoint Communication

Thus far, we have confined our attention to unicast communication, that is, communication between a source and destination pair. In applications such as video-on-demand and teleconferencing, messages from a source are broadcast to multiple destinations. The hosts participating in the communication form a multi-cast group. In the former case, there is one source: The communication is one-to-many. In the latter case, there are many sources: The communication is many-to-many. Many distributed applications and fault-tolerance mechanisms also require multicast capabilities among entities.

Giving each source an independent point-to-point connection to each destination is usually an unacceptable approach because of the potential waste in resources. The common approach is to give each source a multicast tree connection to all destinations. A multicast tree connection, or simply a multicast tree, from a source host in a switched network or inter-connected network is a set of point-to-point simplex connections between pairs of routers (or

switches) and hosts that form a spanning tree; the tree is rooted at the source and connects all the destinations. Figure 11-11(a) gives an example. The graph represents a multicast tree from the source host H1. Each vertex represents a router R_i (shown as a circle) or a host H_1 (shown as a square). Each directed edge in the tree represents a simplex connection. At each entity represented by a vertex adjacent to the edge, resources required to carry messages from the source host over the connection have been reserved, and these messages are transmitted according to some connection-oriented service discipline.

When there is more than one source host in a multicast group and all sources may transmit at the same time, the only viable approach is to have a multicast tree for every host. (An example is a distributed monitoring and control system, where every host exchanges commands with all the other hosts in each round of interaction.) Oftentimes, however, only a subset of hosts in the group transmits simultaneously. (For example, in an audio teleconferencing, only one or a few people talk at a time.) One can save a great deal of resources by having the sources share resources whenever possible. As an example, Figure 1 1-11(b) shows a multicast group for which multicast trees from some sources traverse some common routers and transmission links. We use an undirected edge between a pair of endpoints to represent one or more pairs of simplex connections in both directions between the endpoints. So, the graph in Figure 11-1 1(b) represents three multicast trees rooted at H1, 113, and 114. Hosts H2 and 115 receive messages but never send any of their own. The connections to them are simplex. Hi never receives, and 1/4 sends and receives via different routers. The connections to and from them are simplex also. We can sometimes reduce the bandwidth and buffer space reserved for a multicast group at routers and links traversed by more than one multicast tree of the group by using a resource-reservation protocol that considers the aggregate requirements of the multicast group as a whole. Indeed, this is what RSVP protocol does.

2. Heterogeneity of Destinations

In general, some destination hosts may not be able or willing to set aside the communication and computing resources needed to achieve the quality of service provided by the source and demanded by some other destinations. A method to accommodate different quality of service requirements of destinations is to have the source use a layered scheme to encode its messages. A message carrying layered encoded video or audio can be divided into a mandatory part that gives an acceptable quality followed by one or more optional parts that improve the quality. A destination may choose to receive only packets that give it a poorer quality video and audio, rather than all the packets that the source uses to encode high-resolution and -fidelity video and audio. This reduction in the desired quality in turn leads to a reduction in the amounts of transmission bandwidth and buffer space that routers en route must reserve on behalf of the destination and the destination must have itself.

As an example, suppose that the hosts H4 and H5 in Figure 11-11(a) can only receive and process a low-resolution video stream. Then some network bandwidth and buffer space can be saved by reserving downstream to the right of router R2 only the bandwidth and buffer space needed to guarantee the delivery of the low-resolution video stream. The other hosts demand a high-resolution video transmitted by HI, and resources required to guarantee the timely delivery of the high-resolution video must be reserved at entities represented by the subtree at and to the left of R2.

3. Dynamic Multicast Group Membership

Another behavior of many multicast applications is that participants may join and leave a multicast group. Take video-on-demand for example. A viewer may preview or order a movie at any time. A new destination joins the multicast group when this happens, and the destination may leave the group after previewing. When there is more than one source in a multicast group, the destination may choose to receive from different sources and make this choice dynamically. As an example, in a multimedia

teleconference, a destination with limited resources may choose to receive only videos from a subset of the multicast group. The destination may dynamically change the sources in this subset.

4. Relation to Routing and Admission Control

The establishment of a multicast tree connection involves routing, that is, the choices of routers and links spanning the tree, as well as resource reservation. Good utilization of network resources can be achieved only if the resources required by the connection are taken into account in the choice of the route. This is the rationale of some routing and real-time connection establishment protocols.

An alternative is to treat resource reservation as a separate function from routing and admission control. This is the approach taken by RSVP protocol. Modularity is an obvious advantage. The protocol itself is not concerned with the establishment and maintenance of the underlying multicast tree. Rather, it makes use of the routing and admission-control functions provided by the network or interconnected networks. As a result, RSVP protocol is "portable." With this approach, the routers used by each multicast tree are chosen by the routing protocol based on partial knowledge of the resource requirement (e.g., maximum and average demands) of the multicast group. This can lead to poorer resource utilization. However, resources reserved for a multicast group can be reclaimed if they are not needed later, so it is not obvious that modularity can be gained only at the expense of significant performance penalty. There are yet no evaluation and measurement data to tell us how large the penalty is.

Design Objectives.

In summary, a resource-reservation protocol should accommodate heterogeneous destinations in a multicast group whose membership may change dynamically. At any time, the total resource demand of the multicast group as whole is usually smaller than the sum of the demands of individual members because the members may not use their resources at the same time. Ideally, the amounts of resources reserved at any time for the multicast group should approximate the total amounts demanded by the group as a whole.

10.6 Real-time protocol

Since their advent, the transport protocols TCP and UDP and the Internet protocol IP have served nonreal-time applications well. Yet these protocols are unsuitable for real-time applications for many reasons. IP is connectionless. Previously, we have argued that the connection-oriented approach is more advantageous for real-time applications because of the ability to reserve resources, manage quality of service, and provide isolation in an end-to-end manner for each flow. Therefore, one way to enhance the real-time capabilities of the Internet is to have a connection-oriented alternative to IP. ST-II is such an alternative. TCP is a connection-oriented transport protocol. However, its error-control, flow-control, and sequencing mechanisms are designed to provide users with reliable, sequenced data delivery; they can introduce large delay and jitter and severely limit end-to-end throughput. Unlike data, many real-time applications are more tolerant of erroneous and lost data. For these applications, the cost in real-time performance is too high for the gain in reliability.

This section gives a brief overview of the Real-time Protocol (RTP) and Real-Time Control Protocol (RTCP) [SCFJ]. RTP is a data transport protocol for real-time applications. RTCP is the control protocol accompanying RTP.

10.6.1 Data Transport

RTP is designed to support multicast communication in interactive multimedia applications, such as audio and video teleconferencing and distributed simulation. It deals solely with data delivery and

uses services (e.g., multicast routing, resource reservation and quality of service guarantee) provided by lower-layer protocols. In particular, RTP can run on the unreliable datagram protocol UDP. It uses UDP's multiplexing and error-control services and compensates UDP with its own sequencing function. Monitoring, control, and identification functions are provided by RTCP.

RTP is designed to be scalable from a few participants to thousands of participants in multicast communication and to be able to accommodate heterogeity of sources, receivers, and networks. RTP assumes that multicast group membership may change dynamically. In contrast with TCP, which is separate from applications, RTP is often integrated with applications. By providing headers that can be modified and expanded. the protocol can be tailored by applications.

To explain its main features, we look at a multicast group participating in a video teleconferences supported by RTP and UDP. According to RTP, their audio and video data are transmitted separately. Each medium uses a separate multicast network address and a pair of UDP ports, one for data packets and one for control packets. The combination of the network address and a port is a destination transport address, so each medium uses a pair of transport destination addresses. (We assume here that the IP multicast is used and all participants have the same destination transport address pair. In the case where the underlying layer supports only unicast, each participant has its own transport destination address.) Before the start of the teleconference, one of the participants obtains from a resource manager the transport address pairs to be used by the group and distributes this information to all participants. Thus their multicast session is established.

10.6.1.1 RTP Packets

During the conference, each chunk of audio or video data (e.g., a segment of audio-or an image) from each participant is framed by an RTP header to form an RTP packet. Our example assumes that RTP runs on UDP; hence RTP packets are framed in UDP packets and sent by UDP. A participant may have multiple microphones and cameras. These devices are called synchronization sources. Headers of data packets from the same SSRC have the same SSRC identifier: the identifier helps a receiver separate data packets from the SSRC from data packets generated by other SSRCs.

Data packets from each SSRC are sequenced and time stamped consecutively. The header of each RTP packet contains the time stamp and sequence number of the packet. The former gives the sampling instant of the first octet of the packet. The latter gives the order of the packet with respect to other packets in the same data stream. The need for sequence and timing information arises from the fact that RTP packets on a session may be lost and delivered out of order. (UDP packets are routed through the network independently of each other, so, if RTP runs on top of UDP, there is no guarantee against lost and out-of-sequence packets.) Timing information also makes it possible for the destination to synchronize the associated audio and video (to achieve lip synchronization, e.g.).

Another type of information contained in an RTP packet header is the method used to encode the data. Examples of encoding methods are PCM (Pulse Code Modulation) for audio and JPEG for image. Each SSRC can change the encoding method during the conference. The flexibility of using different encoding methods enables the applications to trade off between quality and resource demands. For example by changing to an encoding scheme that requires a lower bandwidth but produces poorer voice, image, or video, a source can lessen its bandwidth demand during network congestion at the expense of the quality of its data. This is also a way for a multicast group to accommodate a new participant who is resource poor and is constrained to use a lower-quality/required bandwidth scheme.

10.6.1.2 Mixers and Translators.

In addition to hosts, there are also mixers and translators; both are RTP relays. Such a relay connects two or more networks (or interconnected networks) at the transport level. To a relay. each network is

defined by its network and transport layer protocols (e.g., IP/UDP), a multicast address, and a transport-layer destination port.

Each translator relays streams of RTP packets from different sources separately, that is, without altering their SSRC identifiers. Some translators may alter the encoding of data in data packets. Translators may be used as filters that forward packets to destinations protected by firewalls for security reasons. Translators are also needed to connect hosts that support different protocols (e.g., some hosts only IP/UDP while other hosts only ST-II) and to replicate from multicast to unicast.

In contrast to translators, a mixer receives data streams from one or more sources, combines the streams in some manner, possibly changes the data encoding, and then forwards the combined stream onward. Because its input data streams are from different sources and the streams are not synchronized, a mixer first adjusts the timing of packets in each stream and generates its own time stamps of packets in the combined stream. The mixer is the SSRC of the combined stream. The SSRCs of original data streams are called contributing sources (CSRC). Each packet in the combined stream contains in its header a CSRC list, which gives the identifiers of the contributing sources of the data contained in the packet. With this information, a receiver can identify the original source of the data even after it is mixed with other data streams by a mixer.

Often, mixers are used to support heterogeity among destinations. For example, some destinations may be connected through a network with a lower bandwidth. To accommodate such a destination without forcing all destinations to accept poorer quality, a mixer may be used to forward the lower-bandwidth audio stream on low bandwidth links. A video mixer may be used to scale images from separate sources and compose them into one video stream (e.g., putting smaller images of participants in an image to simulate a conference scene.)

10.6.2 RTCP Control Protocol

As stated earlier, there is a control port associated with each data port. Control packets carrying control and monitoring information on data transmission to the data port are sent under the control of RTCP to the control port using the same multicast route as the associated data packets. Specifically, reception quality is fed back to every participant in a multicast group, as well as to a network service provider or monitor when there is one. Distributing this feedback information is the primary function of RTCP. This information is needed to support adaptive encoding and congestion control. It also helps in fault diagnosis, since a monitor can easily determine from reports of all participants whether a problem in data distribution is local or global.

10.6.2.1 Transmission of Reception Quality Reports

Each receiver (i.e, a destination) periodically sends RTCP report packets, simply called reports hereafter. Such a report contains a reception block for each of the sources from which the receiver has received data since the previous report. This block provides the values of performance measures (e.g., fraction and cumulative number of packets lost, interarrival jitter, and delay) used to measure the quality of the data from the source. A report may be a receiver report or a sender report. A receiver report contains only reception blocks. A receiver sends a receiver report if it has not sent any data since the transmission of its previous report. Otherwise, if it has sent some data, it sends a sender report, which contains a sender information section, in addition to reception blocks. The sender section provides transmission statistics and source description items.

We note that multimedia data traffic is self-limiting. For example, usually only one or two people talk and only a few images are sent and displayed at a time during a teleconference independent of the group size. In contrast, control traffic can grow linearly with group size if each participant sends its reception report at a rate independent of group size. To constrain the growth in control traffic, RTCP

keeps the total bandwidth consumed by control packets from all participants in a multicast group constant. In particular, the recommended fraction of bandwidth allocated to RTCP for each multicast session is 5 percent of the total RTP bandwidth of the session. This 5 percent is further divided among senders (i.e., those sending sender reports) and nonsenders, with 1.25 percent for senders and the remaining 3.75 percent for nonsenders. In this way, the protocol ensures that sufficient bandwidth is available for the transmission of sender reports which, as we will see shortly, are more time critical.

10.6.2.2 Computation of Reception Report Retransmission Interval

To maintain a constant total rate, each participant must estimate the group size (i.e., the number of participants) of its multicast session and calculate the interval p between consecutive RTCP packet trans-missions based on the estimated number. The information it maintains for the purpose of this computation include the following:

- $\mathbf{t_p}$: the previous transmission time of a RTCP packet;
- $\mathbf{t_n}$: the next scheduled transmission time of a RTCP packet;
- ***members***: the most recent estimate of group size;
- ***senders***: the most recent estimate of the number of senders in the session;
- ***rtcp_bw***: the total bandwidth to be used for RTCP packets by all participants; and
- ***avg _rtcp _size***: the average RTCP packet size, in octets, over all RTCP packets sent and received by participants.

The rules used by each participant to compute its retransmission interval *p* are rather complicated. While the average retransmission interval of each participant is chosen deterministically, the next transmission time t_n is chosen randomly. Thus, control packets from all participants tend to spread out over time, rather than arriving deterministically and possibly bunched together in time. Specifically, the value of the retransmission interval *p* is computed in two steps.

1. The average retransmission interval p_d is computed deterministically according to $pd = max(Pmin, nC)$ with the parameters p_{min} , C and n determined according to the rules given below.
2. The retransmission interval p is computed from p_d by first randomly choosing a number x from a uniform distribution in the range [$0.5p_d$, $1.5p_d$]. The retransmission interval p is equal to x/1.21828.

After a participant computes the retransmission interval p, it sets the next transmission time t_n of its reception report to the previous transmission time t_p plus p.

The parameter p_{min} used in step I is the minimum average retransmission interval. This parameter is normally five seconds. However, it is desirable to enable a participant (an application) to quickly send a sender report when it starts up and begins to send data. For this reason, when a participants just starts, its average minimum retransmission interval is only half the normal minimum value, that is, 2.5. before the participant sends its first report.

The value of the parameter C is a function of *members, senders*, and *avg_rtcp_size*. Depending on the percentage of members who are senders, there are two cases. When $0 < senders < 0.25\, members$, the senders share 25 percent of the RTCP bandwidth $rtcp\_bw$, leaving the rest of this bandwidth to nonsenders. Hence, the average retransmission interval of a participant depends on whether the participant is a sender. If the participant is a sender, the constant C is equal to avg_rtcp_size divided by 25 percent of the RTCP bandwidth, and n is equal to the number of senders. On the other hand, for a participant which is not a sender, the constant C is equal to avg_rtcp.size divided by 75 percent of the RTCP bandwidth, and n is equal to the number of nonsenders. When

$sender > 0.25$ members, all participants are treated equally. C is equal to avg_rtcp_size divided by rtcp_bw, and n is equal to member.

To estimate the number of participants, each participant keeps a member table containing the SSRCs of participants known to it. Each time the participant receives an RTP or RTCP packet from a source whose SSRC is not in the table, it enters the SSRC into the table and increments member. It deletes the SSRC of a participant and decrements member when it receives a BYE packet from the participant. The BYE packet indicates that the participant has left the multicast group. Similarly, each participant keeps a sender table and updates the table and the estimate sender each time it receives an RTP packet from the sender.

Collision Resolution and Intermedia Synchronization

One of the functions of RTCP is to support collision resolution and intermedia synchronization. An SSRC identifier carried in RTP headers of packets from a source and in the associated RTCP packets is a 32-bit number that is unique within an RTP session. This identifier is chosen randomly by the source when the source starts. Because each source makes this choice independently, it is possible for sources to collide (i.e., the SSRC identifiers chosen by them are identical). Although the probability of a collision is small (less than 10^{-4} when there are 1000 sources), the fact that it can occur makes it necessary for all the sources to be able to detect collisions. When a source detects that its own SSRC identifier is the same as that of other source(s), it sends a BYE RTCP packet containing the identifier, indicating that it ceases to participate and then chooses a new SSRC identifier for itself and uses the new identifier in its packets.

In addition to the above reason, the SSRC identifier of a source also changes when the source restarts. The fact that the SSRC identifier of a source may change makes it necessary to provide another means for receivers to identify each source. This function is provided by RTCP. RTCP packets from each source carry a persistent transport-level identifier called CNAME. When a source starts, its CNAME is sent to all participants in its sender reports. This identifier allows receivers to associate multiple data streams from the source. Intermedia synchronization also requires that the network time-stamp and RTP time-stamps be included in the RTCP packets of data senders.

10.7 Communication in Multicomputer Systems

Like packet switched networks, multihop networks used to interconnect processors in massively parallel machines also consist of crossbar switches connected by full duplex links. Unlike packet switched networks, these networks typically adopt a simpler routing and flow control scheme called wormhole routing.

10.7.1 WORMHOLE NETWORKS

In a wormhole network, messages are segmented into very small flow control units called flits. In a simple wormhole network, each switch provides only enough buffer space to hold on flit per input link. The buffer is there in order to decouple the input link from the output links.

10.7.1.1 Routing and Transmission

Only one flit can occupy a link at each time step; while a message is using a link, another message that also needs the link must wait. Therefore, the transmission of a message may be blocked from starting. Specifically, when the header (i.e., the first flit) of a message reaches a switch, the switch selects an output link for the message based on the information provided in the header. If the output link is free at the time, the header moves forward on that link to the next switch, leaving the input link it used to reach the current switch to the second flit in the message. Similarly, the third flit follows,

using the link freed by the second flit, and so on. On the other hand, if an output link chosen for a message is in use, the header is buffered and waits at the switch until the output link becomes free. In the meantime, subsequent flits that have reached upstream switches occupy the flit buffers there, one per switch. The associated input links (i.e., output links of the corresponding upstream switches) are not available to other messages. We call this phase of message transmission the routing phase. The routing of a message starts when its header leaves the source processor and completes when the header reaches the destination processor. Hereafter, the message has all the links along the path between its source and destination. Each link is occupied by one of the message's flits. The flits shift downstream by one link (i.e., one switch) in each time step without intervention of the switches. Thus, the message "worms" its way nonpreemptively through the network without being queued at any switch. Its transmission completes when its last flit is delivered to the destination processor. In short, the message delay through a wormhole network is the sum of its routing time and transmission time. The latter is essentially the total propagation delay of all links on its transmission path. The time required to route a message depends on the overall network traffic and is the nonpredictable component of message delay.

10.7.1.2 Path Selection and Scheduling

To make the network scalable with the number of processors in the system, algorithms used to select paths for messages are typically simple. As an example, in a two-dimensional mesh network, each switch is connected to four neighboring switches. It is common to use one-bend paths for unicast messages. A one-bend path between two processors in a mesh consists of a segment of column links and a segment of row links. If the path of every message traverses a column segment first and then a row segment (or vice versa), there is no deadlock. There are equally simple, deadlock-free routing algorithms for high-dimensional interconnection networks.

Typically, algorithms used to schedule nonreal-time messages do not prioritize messages. They can be divided into two categories: greedy and throttling. According to a greedy algorithm, each message is routed along a deadlock-free path as soon as it is ready for transmission at the source. In contrast, according to an algorithm that does throttling, a message may wait at the source. This intentional delay to start routing serves a purpose similar to traffic shaping in packet-switched networks, that is, to reduce the worst-case delay.