UNIT-1

Typical Real-Time Applications

Introduction:

System completes its work and delivers its services on a timely basis. Examples of real-time systems include digital control, command and control, signal processing, and telecommunication systems.

Real-time applications:

Digital control, optimal control, command and control, signal processing, tracking, real-time databases, and multimedia

DIGITAL CONTROL:

They are the simplest and the most deterministic real-time applications. Many real-time systems are embedded in sensors and actuators and function as digital controllers. Figure 1–1 shows such a system. The term plant in the block diagram refers to a controlled system, for example, an engine, a brake, an aircraft, a patient. The state of the plant is monitored by sensors and can be changed by actuators.

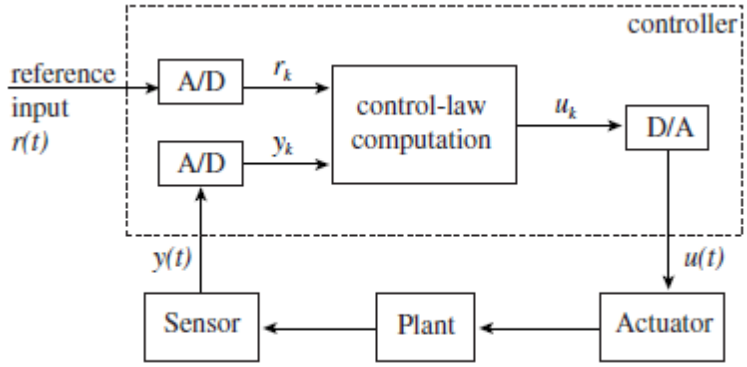


FIGURE 1–1 A digital controller.

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.

Sampled Data Systems:

Acommon approach to designing a digital controller is to start with an analog controller that has the desired behavior. The analog version is then transformed into a digital (i.e., discrete-time and discrete-state) version. The resultant controller is a *sampled data system*.

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 .

Selection of Sampling Period:

The length *T* of time between any two consecutive instants at which *y(t)* and *r (t)* are sampled is called the *sampling period*. 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.

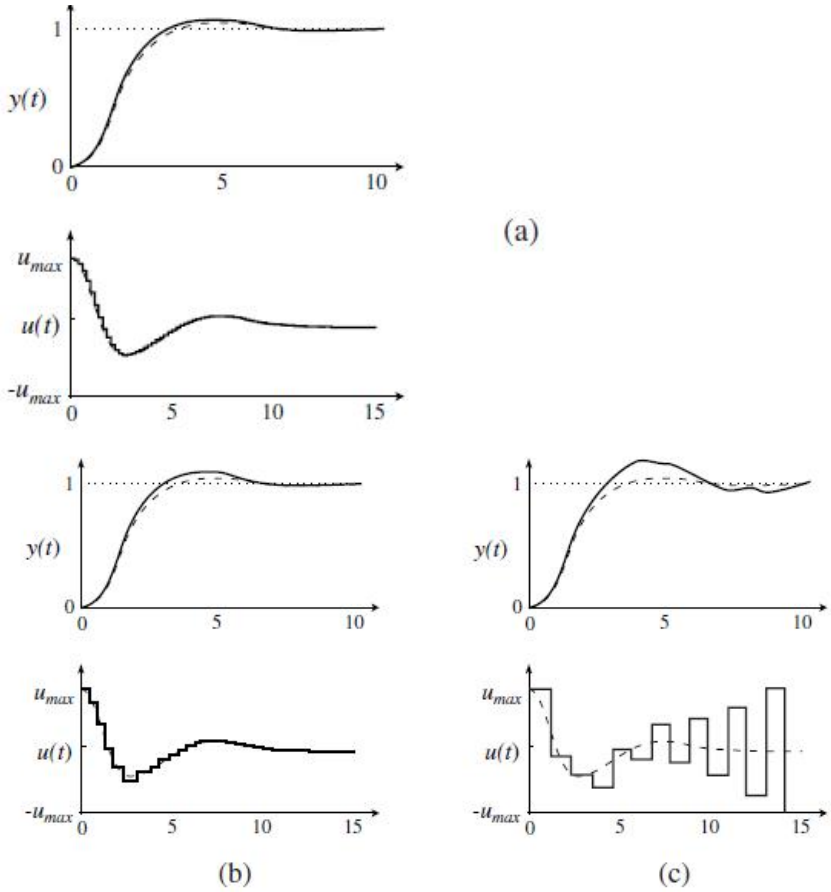


FIGURE 1–2 Effect of sampling period.

In Figure 1–2, these positions are represented by 0 and 1, respectively, and the time origin is the instant when the step in *r (t)* occurs. The dashed lines in Figure 1–2(a) give the output *u(t)* of the analog is said to controller and the observed position *y(t)* of the arm as a function of time. The solid lines in the lower and upper graphs give, respectively, the analog control signal constructed from the digital outputs of the controller and the resultant observed position *y(t)* of the arm. At the sampling rate shown here, the analog and digital versions are essentially the same. The solid lines in Figure 1–2(b) give the behavior of the digital version when the sampling period is increased by 2.5 times. The oscillatory motion of the arm is more pronounced but remains small enough to be acceptable. Figure 1–2(c), the arm requires larger and larger control to stay in the desired position; when this occurs, the system have become unstable.

Multirate Systems. A plant typically has more than one degree of freedom. Its stateis 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 ontrolled by multiple actuators

Example:

Do the following in each 1/180-second cycle:
- Validate sensor data and select data source; in the presence of failures, reconfigure the system.
- Do the following 30-Hz avionics tasks, each once every six cycles:
  - keyboard input and mode selection
  - data normalization and coordinate transformation
  - tracking reference update
- Do the following 30-Hz computations, each once every six cycles:
  - control laws of the outer pitch-control loop
  - control laws of the outer roll-control loop
  - control laws of the outer yaw- and collective-control loop
- Do each of the following 90-Hz computations once every two cycles, using outputs produced by 30-Hz computations and avionics tasks as input:
  - control laws of the inner pitch-control loop
  - control laws of the inner roll- and collective-control loop
- Compute the control laws of the inner yaw-control loop, using outputs produced by 90-Hz control-law computations as input.
- Output commands.
- Carry out built-in-test.
- Wait until the beginning of the next cycle.

Timing Characteristics

The workload generated by each multivariate, multirate digital controller consists of a few periodic control-law computations. Their periods range from a few milliseconds to a few seconds. A control system may contain numerous digital controllers, each of which deals with some attribute of the plant.

The control laws of each multirate controller may have harmonic periods. They typically use the data produced by each other as inputs and are said to be a rate group. On the other hand, no control theoretical reason to make sampling periods of different rate groups related in a harmonic there is way.

More Complex Control-Law Computations:

A discrete-time control scheme that has no continuous-time equivalence is **deadbeat control**:

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

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

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

The Kalman filter starts with the initial estimate ˜ *x*1 = *y*1 and computes a new estimate each sampling period. Specifically, for *k* > 1, the filter computes the estimate ˜ *xk* as follows:

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

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.

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.

Figure1.4 shows a more complex example: the hierarchy of flight control, avionics, and air traffic control systems.5 The Air Traffic Control (ATC) system is at the highest level. It regulates the flow of flights to each destination airport. It does so by assigning to each aircraft an arrival time at each metering fix6 (or waypoint) en route to the destination: The aircraft is supposed to arrive at the metering fix at the assigned arrival time.

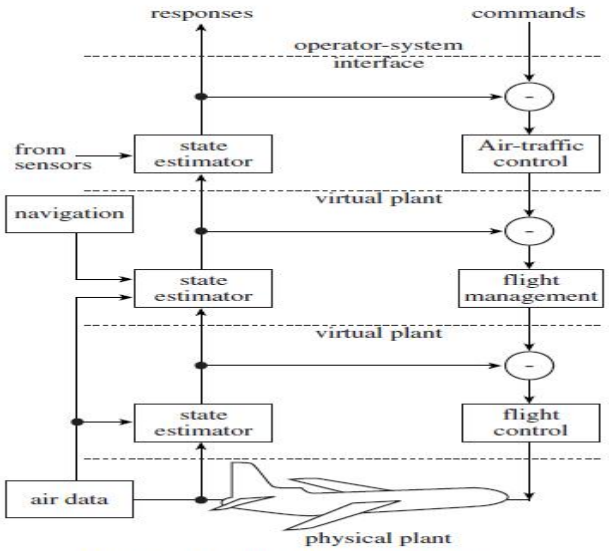


FIGURE 1–4 Air traffic/flight control hierarchy.

Guidance and Control

While a digital controller deals with some dynamical behavior of the physical plant, a econdlevel 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.

Complexity and Timing Requirements:

In principle, these problems can be solved using dynamic programming and mathematical programming techniques.Heuristic algorithms used for guidance and control purposes typically consider one constraint at a time, rather than all the constraints at the same time.

Other Capabilities. 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. 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.

Real-Time Command and Control:

The controller at the highest level of a control hierarchy is a command and control system. An Air Traffic Control (ATC) system is an excellent example.

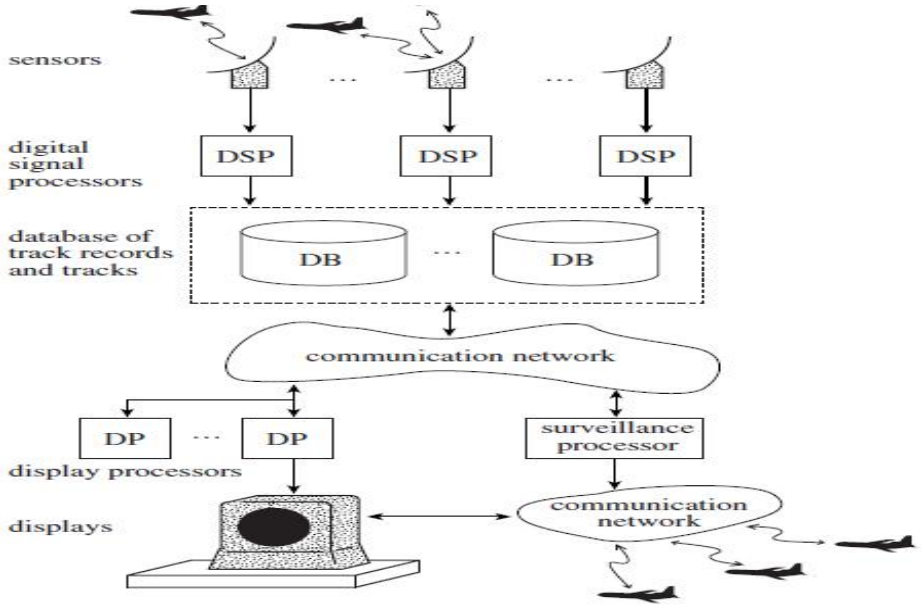


FIGURE 1–5 An architecture of air traffic control system.

Figure 1–5 shows a possible architecture. The ATC system monitors the aircraft in its coverage area and the environment Outputs from the ATC system include the assigned arrival times to metering fixes for individual aircraft. As stated earlier, these outputs are reference inputs to on-board flight management systems. Thus, the ATC system indirectly controls the embedded components in low levels of the control hierarchy. In addition, the ATC system provides voice and telemetry links to on-board avionics.

SIGNAL PROCESSING:

Most signal processing applications have some kind of real-time requirements.We focus here on those whose response times must be under a few milliseconds to a few seconds. Examples are digital filtering, video and voice compressing/decompression, and radar signal processing.

Typically, 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:

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

In the simplest case, the weights, *a(k, i)*'s, are known and fixed.8 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 that the time required to produce an output is *O(n)*.

Radar System:

A signal processing application is typically a part of a larger system. As an example, Figure 1–6 shows a block diagram of a (passive) 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.

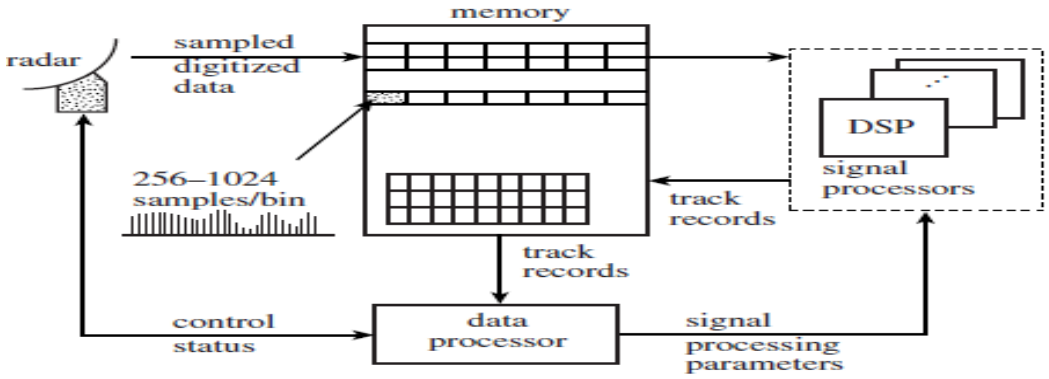


FIGURE 1–6 Radar signal processing and tracking system.

An array of digital signal processors processes these sampled values. The data thus produced are analyzed by one or more data processors, which not only interface with the display system, but also generate commands to control the radar and select parameters to be used by signal processors in the next cycle of data collection and analysis.

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 = 3×108 meters.

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.

Tracking: 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*

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 an or cannot be tentatively assigned to one or more established trajectories.

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.

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 $O(nm \log m)$.

OTHER REAL-TIME APPLICATIONS:

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

Real-Time Databases

Specifically, a real-time database contains data objects, called *image objects* that represent real-world objects. The attributes of an image object are those of the represented real world object. For example, an air traffic control database contains image objects that represent aircraft in the coverage area. The attributes of such an image object include the position and heading of the aircraft. The values of these attributes are updated periodically based on the measured values of the actual position and heading provided by the radar system.

Absolute Temporal Consistency

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.

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

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.

MPEG Compression/Decompression. A video compression standard is MPEG-2 [ISO94]. The standard makes use of three techniques. They are motion compensation for reducing temporal redundancy, discrete cosine transform for reducing spatial redundancy, and entropy encoding for reducing the number of bits required to encode all the information

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 blocks in I-frames and difference blocks in P- and B-frames.

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 de-mand, 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 ge

neral-purpose processor.

Hard versus Soft Real-Time Systems

JOBS AND PROCESSORS:

We call each unit of work that is scheduled and executed by the system a *job* and a set of lated jobs which jointly provide some system function a *task.* 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.

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.
The ***deadline*** of a job is the instant of time by which its execution is required to be completed. control-law computation job by the release must complete time of the subsequent job.
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

HARD AND SOFT TIMING CONSTRAINTS:

They are based on the functional criticality of jobs, usefulness of late results, and deterministic or probabilistic nature of the constraints.a timing constraint or deadline is *hard* if the failure to meet it is considered to be a fatal fault. A hard deadline is imposed on a job because a late result produced by the job after the deadline may have disastrous consequences. (As examples,a late command to stop a train may cause a collision, and a bomb dropped too late may hit a civilian population instead of the intended military target.)

In contrast, the late completion of job that has a *soft deadline* is undesirable In real-time systems literature, the distinction between hard and soft timing constraints is sometimes stated quantitatively in terms of the usefulness of results (and therefore the overall system performance) as functions of the tardinesses of jobs. The *tardiness* of a job measures how late it completes respective to its deadline. Its tardiness is zero if the job completes at or before its deadline; otherwise, if the job is late, its tardiness is equal to the difference between its *completion time.*

The deadline of a job is softer if the usefulness of its result decreases at a slower rate. By this means, we can define a spectrum of hard/soft timing constraints. Sometimes, we see this distinction made on the basis of whether the timing constraint is expressed in deterministic or probabilistic terms. If a job must never miss its deadline, then the deadline is

hard. On the other hand, if its deadline can be missed occasionally with some acceptably low probability, then its timing constraint is soft

Hard Timing Constraints and Temporal Quality-of-Service Guarantees

The timing constraint of a job is hard, and the job is a hard real-time job, if the user requires the validation that the system By validation, we mean a demonstration by a provably correct, efficient procedure or by exhaustive simulation and testing We call an application (task) with hard timing constraints a hard real-time application and a system containing mostly hard real-time applications a hard real-time system. For many traditional hard real-time applications (e.g., digital controllers)

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.

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

1. 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);

2. 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);

3. Constraints in terms of some usefulness function (e.g., the usefulness of every control law computation is 0.8 or more).

SOFT REAL-TIME SYSTEMS

A system in which jobs have soft deadlines is a *soft real-time system*. The developer 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.

A Reference Model of Real-Time Systems

PROCESSORS AND RESOURCES:

We divide all the system resources into two major types: processors and resources. Again, 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. Sometimes, we will need to distinguish the *types* of processors. Two processors are of the same type if they are functionally identical and can be used interchangeably.

By resources, we will specifically mean *passive* resources. Examples of resources are memory, sequence numbers, mutexes, 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. Although we will rarely mention this attribute, we will implicitly assume that the rate of progress a job makes toward its completion depends on the speed of the processor on which it executes.

We will use the letter *R* to denote resources. The resources in the examples mentioned above are *reusable*,

TEMPORAL PARAMETERS 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. 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.
Each job *Ji* is characterized by its temporal parameters, functional parameters, resource parameters, and interconnection parameters. Its temporal parameters tell us its timing constraints and behavior.
the release time, absolute deadline, and relative deadline of a job *Ji* ; these are temporal parameters. We will use *ri* , *di*, and *Di* , respectively, to denote them and call the time interval *(ri , di*]1 between the release time and absolute deadline of the job *Ji* its *feasible interval*.

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 *ri* of each job *Ji* ; only that *ri* is in a range [*ri* − *, ri* +]. *Ri* can be as early as the earliest release time *ri* − and as late as the latest release time *ri* +. Indeed, some models assume that only the range of *ri* is known and call this range the *jitter* in *ri*, or *release-*

time jitter 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.

Execution Time:

Another temporal parameter of a job, Ji , is its execution time, ei . ei is the amount of time required to complete the execution of Ji when it executes alone and has all the resources it requires.

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

Periods, Execution Times, and Phases of Periodic Tasks

In the periodic task model, each computation or data transmission that is executed repeatly 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 *Ti* , is a sequence of jobs. The *period pi* of the periodic task *Ti* is the minimum length of all time intervals between release times of consecutive jobs in *Ti*. Its *execution time* is the maximum execution time of all the jobs in it. With a slight abuse of the notation, we use *ei* to denote the execution time of the periodic task *Ti*

Aperiodic and Sporadic Tasks

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.

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.

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*.

There is a directed edge from the vertex *Ji* to the vertex *Jk* when the job *Ji* is an immediate predecessor of the job *Jk* . 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.

OTHER TYPES OF DEPENDENCIES:

These extensions include temporal distance, OR jobs, conditional branches, and pipe (or pipeline).

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.

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***. 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.

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, such as the example in Figure

```
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
```

FUNCTIONAL PARAMETERS:

Among them are preemptivity, criticality, optional interval, and laxity type.

Preemptivity of Jobs:

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. A job is *nonpreemptable* if it must be executed from start to completion without interruption.

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.

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*.

Laxity Type and Laxity Function

The laxity type of a job indicates whether its timing constraints are soft or hard. As mentioned earlier, in real-time systems literature, the laxity type of a job is sometimes supplemented by a *usefulness function*.

RESOURCE PARAMETERS OF JOBS ANDP ARAMETERS OF RESOURCES

Earlier we said that the basic components of the underlying system available to the application system(s) are processors and resources. Every job requires a processor throughout its execution.3 In addition to a processor, a job may also require some resources. The resource parameters of each job give us the type of processor and the units of each resource type required by the job and the time intervals during its execution when the units are required. These parameters provide the information that is needed to support resource management decisions.

Preemptivity of Resources:

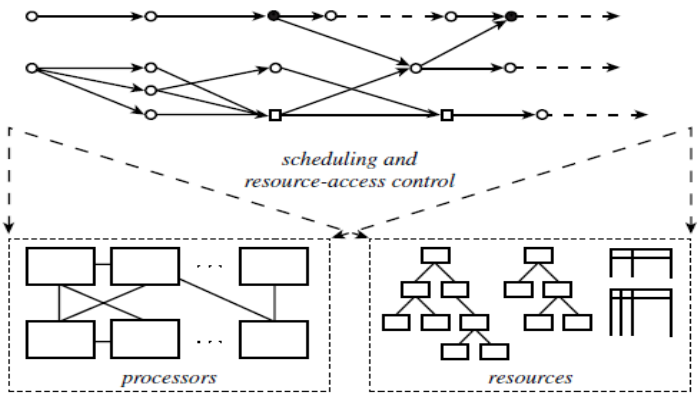
resource parameter is *preemptivity*. A resource is nonpreemptable if each unit of the resource is constrained to be used serially. In other words, once a unit of a nonpreemptable resource is allocated to a job, other jobs needing the unit must wait until the job completes its use. Otherwise, if jobs 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.

Resource Graph:

We can describe the configuration of the resources using a *resource graph*. In a resource graph, there is a vertex *Ri* for every processor or resource *Ri* in the system. While edges in task graphs represent different types of dependencies among jobs, edges in a resource graph represent the relationship among resources.

Scheduling hierarchy:

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



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*

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.

Feasibility, Optimality, and Performance Measures:

A valid schedule is a *feasible schedule* if every job completes by its deadline (or, in general, meets its timing constraints).We say that a set of jobs is *schedulable* according to a scheduling algorithm if when using the algorithm the scheduler always produces a feasible schedule. 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.*

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.

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

Commonly Used Approaches to Real-Time Scheduling

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.

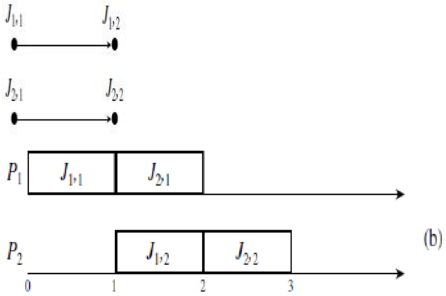
WEIGHTED ROUND-ROBIN APPROACH:

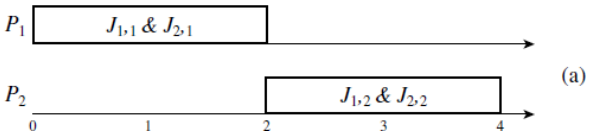
The round-robin approach is commonly used for scheduling time-shared applications. When jobs are scheduled on a round-robin basis, every job joins a First-in-first-out (FIFO) queue when it becomes ready for execution. The job at the head of the queue executes for at most one time slice. (A time slice is the basic granule of time that is allocated to jobs. In a timeshared vironment, 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.

Example,

we consider the two sets of jobs, $\mathbf{J}1 = \{J1,1, J1,2\}$ and $\mathbf{J}2 = \{J2,1, J2,2\}$, shown in Figure . The release times of all jobs are 0, and their execution times are 1. $J1,1$ and $J2,1$ execute on processor $P1$, and $J1,2$ and $J2,2$ execute on processor $P2$. Suppose that $J1,1$ is the predecessor of $J1,2$, and $J2,1$ is the predecessor of $J2,2$. Figure (a) shows that both sets of jobs (i.e., the second jobs $J1,2$ and $J2,2$ 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 (b) shows that if the jobs on each processor are executed one

after the other, one of the chains can complete at time 2, while the other can complete at time 3.





PRIORITY-DRIVEN APPROACH:

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. We will return shortly to illustrate that greed does not always pay; sometimes it is better to have some jobs wait even when they are ready to execute and the resources they require are available.

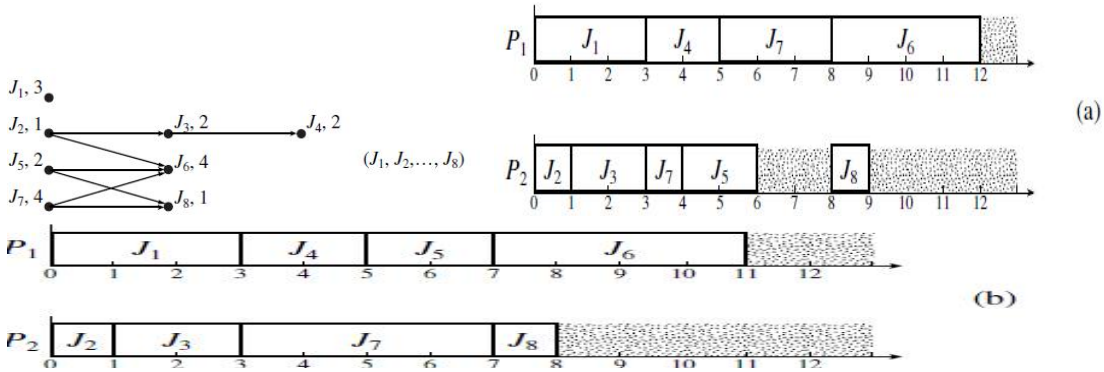
Example:

The given Below Figure 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.

Figure (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 (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.



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.

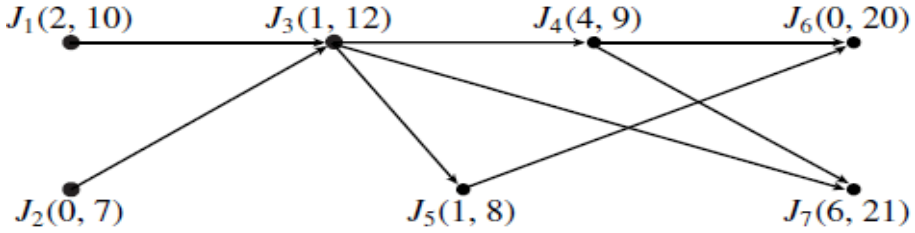
DYNAMIC VERSUS STATIC SYSTEMS:

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

EFFECTIVE RELEASE TIMES AND DEADLINES:

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.



Example: *J*1 and *J*3 in above figure have the same effective release time and deadline.An algorithm which ignores the precedence constraint between them may schedule *J*3 in an earlier interval and *J*1 in a later interval. If this happens, we can always add a step to swap the two jobs, that is, move *J*1 to where *J*3 is scheduled and vice versa. This swapping is always possible, and it transforms an invalid schedule into a valid one.

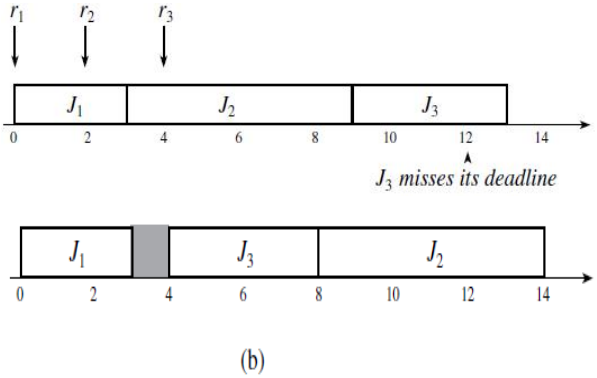
OPTIMALITY OF THE EDF AND LST ALGORITHMS:

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.

NONOPTIMALITY OF THE EDF AND THE 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.



(b)

(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.

(b) Shows 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.

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. You will see in later chapters other advantages of the priority-driven approach which are at least as compelling as these two 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

OFF-LINE VERSUS ON-LINE SCHEDULING:

clock-driven scheduler typically makes use of a pre computed 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 pre computed and stored for use. In this case, we say that scheduling is (done) off-line, and the pre computed 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.

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.

Clearly, on-line scheduling is the only option in a system whose future workload is unpredictable. An on-line scheduler can accommodate dynamic variations in user demands and resource availability. 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