2

BASIC CONCEPTS

2.1 INTRODUCTION

Over the last few years, several algorithms and methodologies have been proposed in the literature to improve the predictability of real-time systems. In order to present these results we need to define some basic concepts that will be used throughout the book. We begin with the most important software entity treated by any operating system, the *process*. A process is a computation that is executed by the CPU in a sequential fashion. In this text, the terms *process* and *task* are used as synonyms. However, it is worth saying that some authors prefer to distinguish them and define a task as a sequential execution of code that does not suspend itself during execution, whereas a process is a more complex computational activity, that can be composed by many tasks.

When a single processor has to execute a set of concurrent tasks – that is, tasks that can overlap in time – the CPU has to be assigned to the various tasks according to a predefined criterion, called a *scheduling policy*. The set of rules that, at any time, determines the order in which tasks are executed is called a *scheduling algorithm*. The specific operation of allocating the CPU to a task selected by the scheduling algorithm is referred as *dispatching*.

Thus, a task that could potentially execute on the CPU can be either in execution if it has been selected by the scheduling algorithm or waiting for the CPU if another task is executing. A task that can potentially execute on the processor, independently on its actual availability, is called an *active* task. A task waiting for the processor is called a *ready* task, whereas the task in execution is called a *running* task. All ready tasks waiting for the processor are kept in

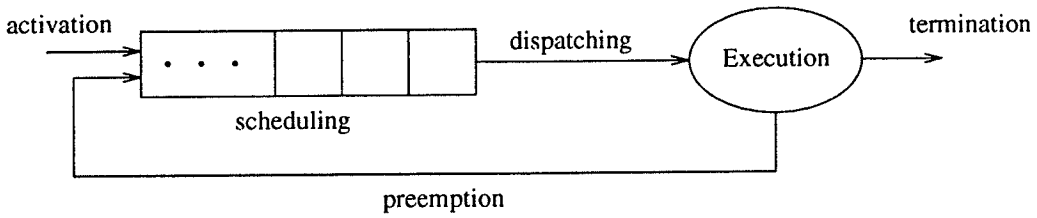

Figure 2.1 Queue of ready tasks waiting for execution.

a queue, called *ready queue*. Operating systems that handles different types of tasks, may have more than one ready queue.

In many operating systems that allow dynamic task activation, the running task can be interrupted at any point, so that a more important task that arrives in the system can immediately gain the processor and does not need to wait in the ready queue. In this case, the running task is interrupted and inserted in the ready queue, while the CPU is assigned to the most important ready task which just arrived. The operation of suspending the running task and inserting it into the ready queue is called *preemption*. Figure 2.1 schematically illustrates the concepts presented above. In dynamic real-time systems, preemption is important for three reasons [SZ92]:

- Tasks performing exception handling may need to preempt existing tasks so that responses to exceptions may be issued in a timely fashion.
- When application tasks have different levels of criticalness expressing task importance, preemption permits to anticipate the execution of the most critical activities.
- More efficient schedules can be produced to improve system responsiveness.

Given a set of tasks, $J = \{J_1, \ldots, J_n\}$, a *schedule* is an assignment of tasks to the processor, so that each task is executed until completion. More formally, a schedule can be defined as a function $\sigma : \mathbf{R}^+ \rightarrow \mathbf{N}$ such that $\forall t \in \mathbf{R}^+$, $\exists t_1, t_2$ such that $t \in [t_1, t_2)$ and $\forall t' \in [t_1, t_2)$ $\sigma(t) = \sigma(t')$. In other words, $\sigma(t)$ is an integer step function and $\sigma(t) = k$, with $k > 0$, means that task J_k is executing at time t, while $\sigma(t) = 0$ means that the CPU is idle. Figure 2.2 shows an example of schedule obtained by executing three tasks: J_1, J_2, J_3.

- At times t_1, t_2, t_3, and t_4, the processor performs a *context switch*.

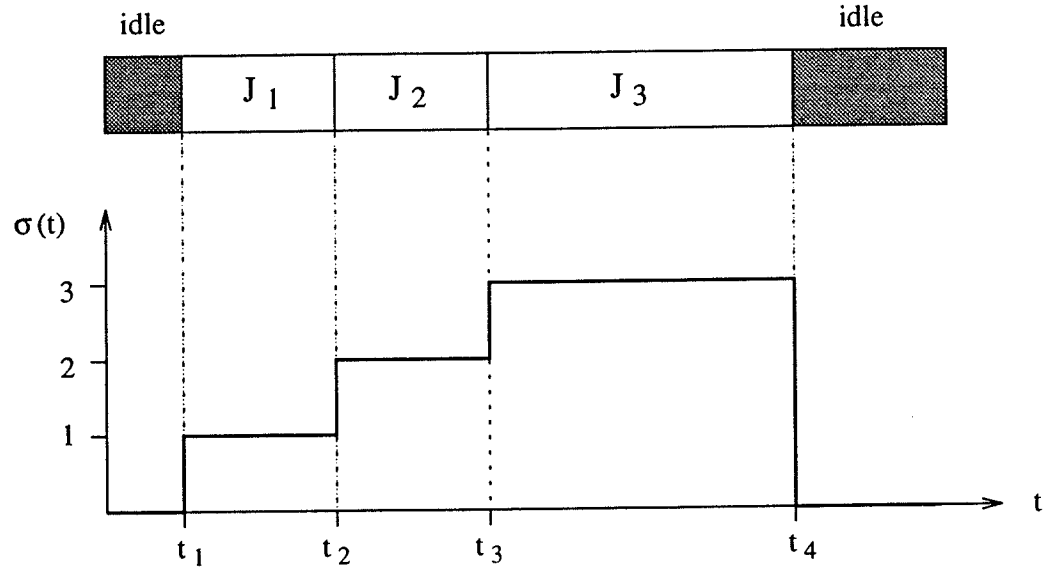

Figure 2.2 Schedule obtained by executing three tasks J_1, J_2, and J_3.

- Each interval $[t_i, t_{i+1})$ in which $\sigma(t)$ is constant is called *time slice*. Interval $[x, y)$ identifies all values of t such that $x \leq t < y$.
- A *preemptive* schedule is a schedule in which the running task can be arbitrarily suspended at any time, to assign the CPU to another task according to a predefined scheduling policy. In preemptive schedules, tasks may be executed in disjointed interval of times.
- A schedule is said to be *feasible* if all tasks can be completed according to a set of specified constraints.
- A set of tasks is said to be *schedulable* if there exists at least one algorithm that can produce a feasible schedule.

An example of preemptive schedule is shown in Figure 2.3.

2.2 TYPES OF TASK CONSTRAINTS

Typical constraints that can be specified on real-time tasks are of three classes: timing constraints, precedence relations, and mutual exclusion constraints on shared resources.

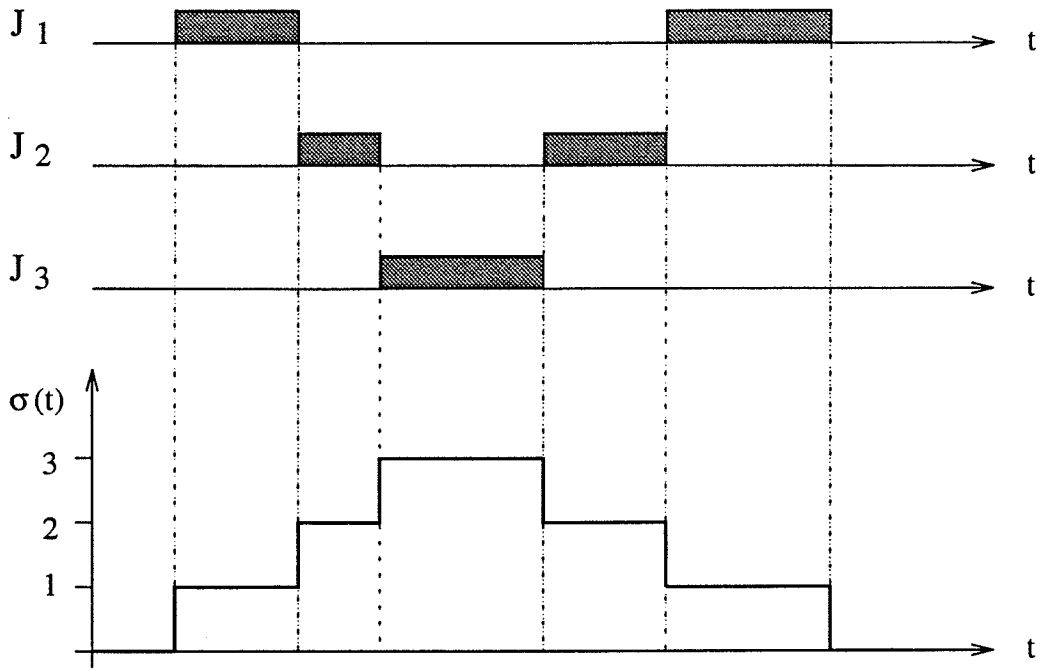

Figure 2.3 Example of a preemptive schedule.

2.2.1 Timing constraints

Real-time systems are characterized by computational activities with stringent timing constraints that must be met in order to achieve the desired behavior. A typical timing constraint on a task is the *deadline*, which represents the time before which a process should complete its execution without causing any damage to the system. Depending on the consequences of a missed deadline, real-time tasks are usually distinguished in two classes:

- **Hard**. A task is said to be hard if a completion after its deadline can cause catastrophic consequences on the system. In this case, any instance of the task should a priori be guaranteed in the worst-case scenario.
- **Soft**. A task is said to be soft if missing its deadline decreases the performance of the system but does not jeopardize its correct behavior.

In general, a real-time task J_i can be characterized by the following parameters:

- **Arrival time** a_i: is the time at which a task becomes ready for execution; it is also referred as *request time* or *release time* and indicated by r_i;

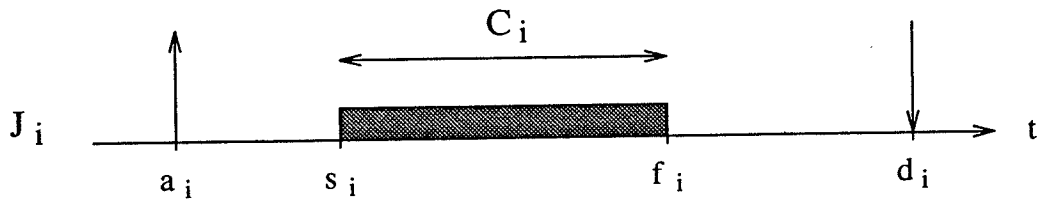

Figure 2.4 Typical parameters of a real-time task.

- **Computation time** C_i: is the time necessary to the processor for executing the task without interruption;
- **Deadline** d_i: is the time before which a task should be complete to avoid damage to the system;
- **Start time** s_i: is the time at which a task starts its execution;
- **Finishing time** f_i: is the time at which a task finishes its execution;
- **Criticalness**: is a parameter related to the consequences of missing the deadline (typically, it can be hard or soft);
- **Value** v_i: represents the relative importance of the task with respect to the other tasks in the system;
- **Lateness** L_i: $L_i = f_i - d_i$ represents the delay of a task completion with respect to its deadline; note that if a task completes before the deadline, its lateness is negative;
- **Tardiness** or *Exceeding time* E_i: $E_i = max(0, L_i)$ is the time a task stays active after its deadline;
- **Laxity** or *Slack time* X_i: $X_i = d_i - a_i - C_i$ is the maximum time a task can be delayed on its activation to complete within its deadline.

Some of the parameters defined above are illustrated in Figure 2.4.

Another timing characteristic that can be specified on a real-time task concerns the regularity of its activation. In particular, tasks can be defined as *periodic* or *aperiodic*. Periodic tasks consist of an infinite sequence of identical activities, called *instances* or *jobs*, that are regularly activated at a constant rate. For the sake of clarity, from now on, a periodic task will be denoted by τ_i, whereas an aperiodic job by J_i.

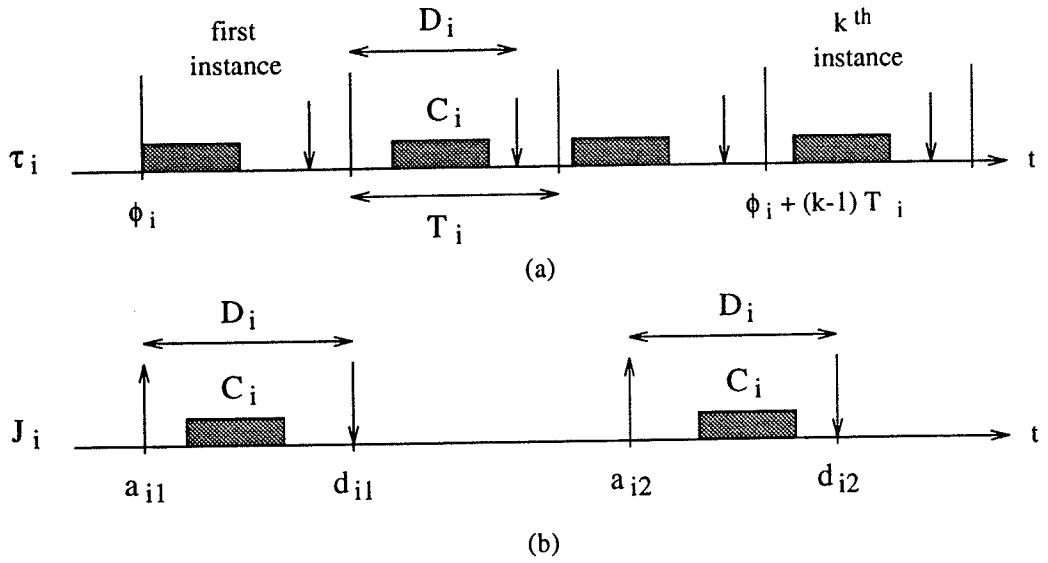

Figure 2.5 Sequence of instances for a periodic and an aperiodic task.

The activation time of the first periodic instance is called *phase*. If ϕ_i is the phase of the periodic task τ_i, the activation time of the kth instance is given by $\phi_i + (k-1)T_i$, where T_i is called *period* of the task. In many practical cases, a periodic process can be completely characterized by its computation time C_i and its relative deadline D_i, which is often considered coincident to the end of the period. Moreover, the parameters C_i, T_i e D_i are considered to be constant for each instance. Aperiodic tasks also consist of an infinite sequence of identical activities (instances); however, their activations are not regular. Figure 2.5 shows an example of task instances for a periodic and for an aperiodic task.

2.2.2 Precedence constraints

In certain applications, computational activities cannot be executed in arbitrary order but have to respect some precedence relations defined at the design stage. Such precedence relations are usually described through a directed acyclic graph G, where tasks are represented by nodes and precedence relations by arrows. A precedence graph G induces a partial order on the task set.

- The notation $J_a \prec J_b$ specifies that task J_a is a *predecessor* of task J_b, meaning that G contains a directed path from node J_a to node J_b.
- The notation $J_a \rightarrow J_b$ specifies that task J_a is an *immediate predecessor* of J_b, meaning that G contains an arc directed from node J_a to node J_b.

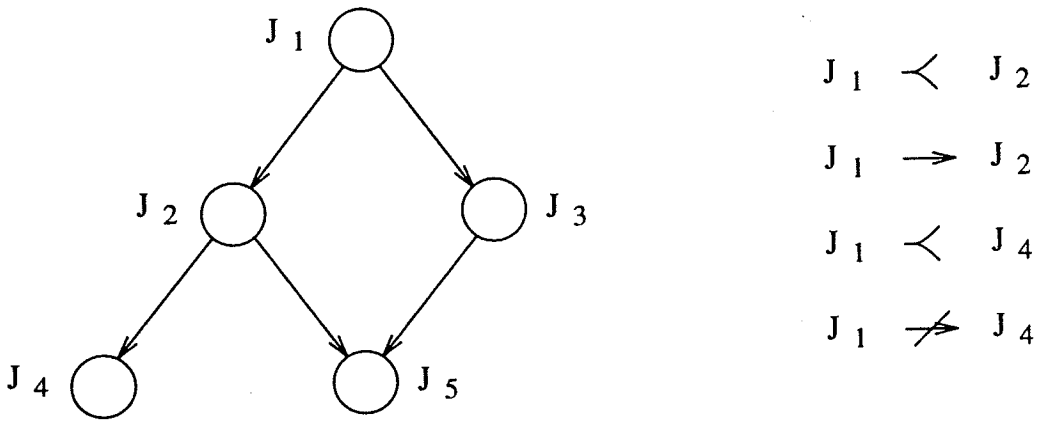

Figure 2.6 Precedence relations among five tasks.

Figure 2.6 illustrates a directed acyclic graph that describes the precedence constraints among five tasks. From the graph structure we observe that task J_1 is the only one that can start executing since it does not have predecessors. Tasks with no predecessors are called *beginning tasks.* As J_1 is completed, either J_2 or J_3 can start. Task J_4 can start only when J_2 is completed, whereas J_5 must wait the completion of J_2 and J_3. Tasks with no successors, as J_4 and J_5, are called *ending tasks.*

In order to understand how precedence graphs can be derived from tasks' relations, let us consider the application illustrated in Figure 2.7. Here, a number of objects moving on a conveyor belt must be recognized and classified using a stereo vision system, consisting of two cameras mounted in a suitable location. Suppose that the recognition process is carried out by integrating the two-dimensional features of the top view of the objects with the height information extracted by the pixel disparity on the two images. As a consequence, the computational activities of the application can be organized by defining the following tasks:

- Two tasks (one for each camera) dedicated to image acquisition, whose objective is to transfer the image from the camera to the processor memory (they are identified by *acq1* and *acq2*);
- Two tasks (one for each camera) dedicated to low-level image processing (typical operations performed at this level include digital filtering for noise reduction and edge detection; we identify these tasks as *edge1* and *edge2*);
- A task for extracting two-dimensional features from the object contours (it is referred as *shape*);

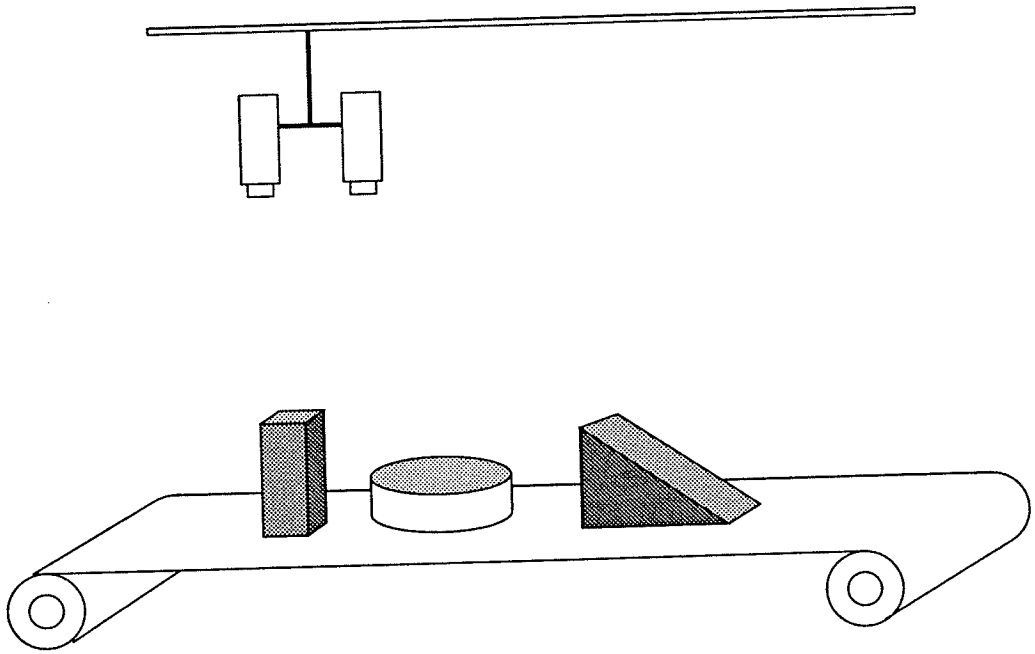

Figure 2.7 Industrial application which requires a visual recognition of objects on a conveyor belt.

- A task for computing the pixel disparities from the two images (it is referred as *disp*);
- A task for determining the object height from the results achieved by the *disp* task (it is referred as *H*);
- A task performing the final recognition (this task integrates the geometrical features of the object contour with the height information and tries to match these data with those stored in the data base; it is referred as *rec*).

From the logic relations existing among the computations, it is easy to see that tasks *acq1* and *acq2* can be executed in parallel before any other activity. Tasks *edge1* and *edge2* can also be executed in parallel, but each task cannot start before the associated acquisition task completes. Task *shape* is based on the object contour extracted by the low-level image processing, therefore it must wait the termination of both *edge1* and *edge2*. The same is true for task *disp*, which however can be executed in parallel with task *shape*. Then, task *H* can only start as *disp* completes and, finally, task *rec* must wait the completion of *H* and *shape*. The resulting precedence graph is shown in Figure 2.8.

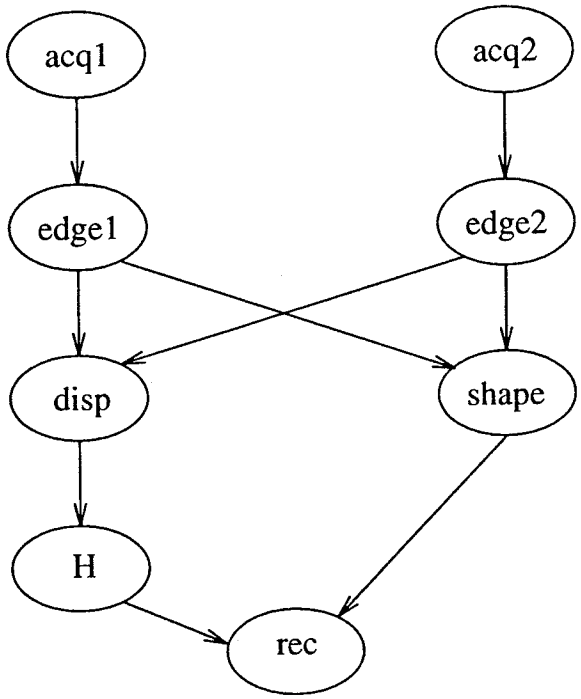

Figure 2.8 Precedence graph associated with the robotic application.

2.2.3 Resource constraints

From a process point of view, a *resource* is any software structure that can be used by the process to advance its execution. Typically, a resource can be a data structure, a set of variables, a main memory area, a file, a piece of program, or a set of registers of a peripheral device. A resource dedicated to a particular process is said to be *private*, whereas a resource that can be used by more tasks is called a *shared resource.*

To maintain data consistency, many shared resources do not allow simultaneous accesses but require mutual exclusion among competing tasks. They are called *exclusive resources.* Let R be an exclusive resource shared by tasks J_a and J_b. If A is the operation performed on R by J_a, and B is the operation performed on R by J_b, then A and B must never be executed at the same time. A piece of code executed under mutual exclusion constraints is called a *critical section.*

To ensure sequential accesses to exclusive resources, operating systems usually provide a synchronization mechanism (such as semaphores) that can be used by tasks to create critical sections of code. Hence, when we say that two or more tasks have resource constraints, we mean that they have to be synchronized since they share exclusive resources.

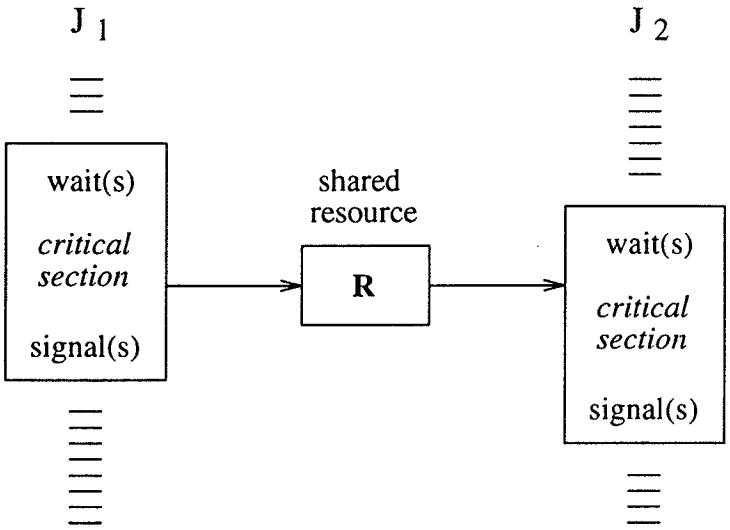

Figure 2.9 Structure of two tasks that share an exclusive resource.

Consider two tasks J_1 and J_2 that share an exclusive resource R (for instance, a list), on which two operations (such as *insert* and *remove*) are defined. The code implementing such operations is thus a critical section that must be executed in mutual exclusion. If a binary semaphore s is used for this purpose, then each critical section must begin with a *wait(s)* primitive and must end with a *signal(s)* primitive (see Figure 2.9).

If preemption is allowed and J_1 has a higher priority than J_2, then J_1 can block in the situation depicted in Figure 2.10. Here, task J_2 is activated first, and, after a while, it enters the critical section and locks the semaphore. While J_2 is executing the critical section, task J_1 arrives, and, since it has a higher priority, it preempts J_2 and starts executing. However, at time t_1, when attempting to enter its critical section, it is blocked on the semaphore and J_2 is resumed. J_1 is blocked until time t_2, when J_2 releases the critical section by executing the *signal(s)* primitive, which unlocks the semaphore.

A task waiting for an exclusive resource is said to be *blocked* on that resource. All tasks blocked on the same resource are kept in a queue associated with the semaphore, which protects the resource. When a running task executes a *wait* primitive on a locked semaphore, it enters a *waiting* state, until another task executes a *signal* primitive that unlocks the semaphore. When a task leaves the waiting state, it does not go in the running state, but in the ready state, so that the CPU can be assigned to the highest-priority task by the scheduling algorithm. The state transition diagram relative to the situation described above is shown in Figure 2.11.

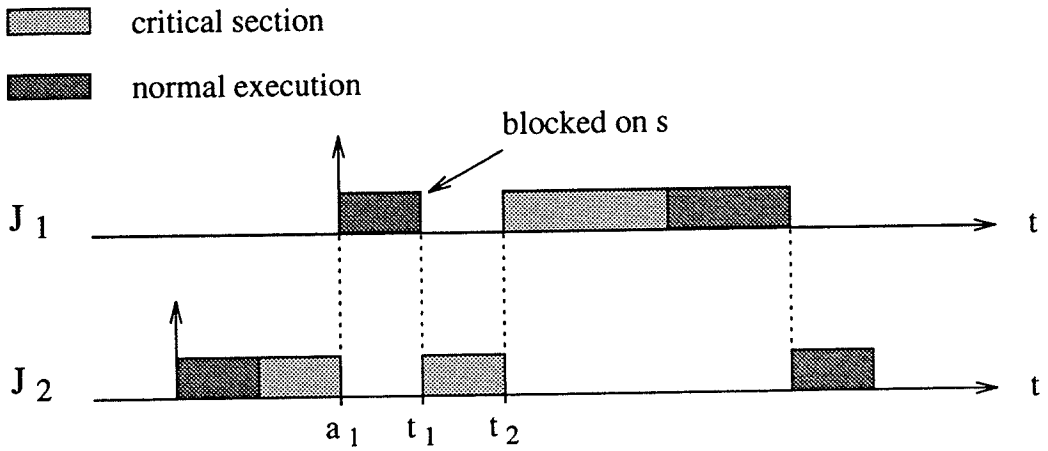

Figure 2.10 Example of blocking on an exclusive resource.

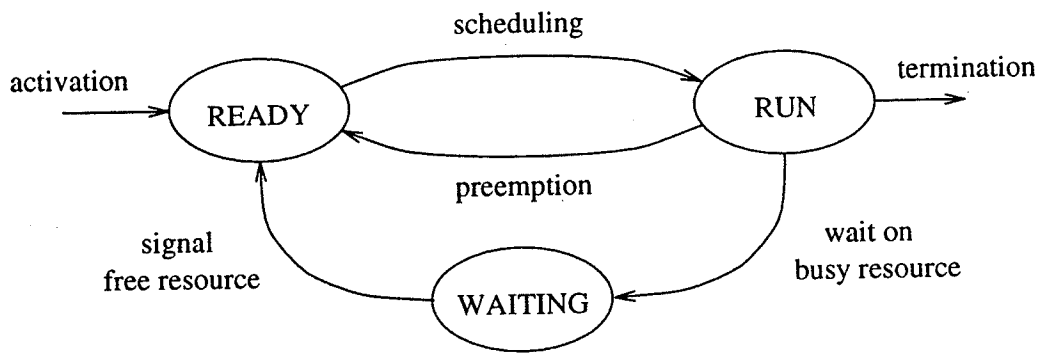

Figure 2.11 Waiting state caused by resource constraints.

2.3 DEFINITION OF SCHEDULING PROBLEMS

In general, to define a scheduling problem we need to specify three sets: a set of n tasks $J = \{J_1, J_2, \ldots, J_n\}$, a set of m processors $P = \{P_1, P_2, \ldots, P_m\}$ and a set of s types of resources $R = \{R_1, R_2, \ldots, R_s\}$. Moreover, precedence relations among tasks can be specified through a directed acyclic graph, and timing constraints can be associated with each task. In this context, scheduling means to assign processors from P and resources from R to tasks from J in order to complete all tasks under the imposed constraints [B+93]. This problem, in its general form, has been shown to be NP-complete [GJ79] and hence computationally intractable.

Indeed, the complexity of scheduling algorithms is of high relevance in dynamic real-time systems, where scheduling decisions must be taken on-line during task execution. A *polynomial algorithm* is one whose time complexity grows as a polynomial function p of the input length n of an instance. The complexity of such algorithms is denoted by $O(p(n))$. Each algorithm whose complexity function cannot be bounded in that way is called an *exponential time algorithm.* In particular, **NP** is the class of all decision problems that can be solved in polynomial time by a *non*deterministic Turing machine. A problem Q is said to be NP-*complete* if $Q \in \mathbf{NP}$ and, for every $Q' \in \mathbf{NP}$, Q' is polynomially transformable to Q [GJ79]. A decision problem Q is said to be NP-*hard* if all problems in **NP** are polynomially transformable to Q, but we cannot show that $Q \in \mathbf{NP}$.

Let us consider two algorithms with complexity functions n and 5^n, respectively, and let us assume that an elementary step for these algorithms lasts 1 μs. If the input length of the instance is $n = 30$, then it is easy to calculate that the polynomial algorithm can solve the problem in 30 μs, whereas the other needs about $3 \cdot 10^5$ centuries. This example illustrates that the difference between polynomial and exponential time algorithms is large and, hence, it may have a strong influence on the performance of dynamic real-time systems. As a consequence, one of the research objectives on real-time scheduling is to restrict our attention to simpler, but still practical, problems that can be solved in polynomial time complexity.

In order to reduce the complexity of constructing a feasible schedule, one may simplify the computer architecture (for example, by restricting to the case of uniprocessor systems), or one may adopt a preemptive model, use fixed priorities, remove precedence and/or resource constraints, assume simultaneous task

activation, homogeneous task sets (solely periodic or solely aperiodic activities), and so on. The assumptions made on the system or on the tasks are typically used to classify the various scheduling algorithms proposed in the literature.

2.3.1 Classification of scheduling algorithms

Among the great variety of algorithms proposed for scheduling real-time tasks, we can identify the following main classes.

- **Preemptive**. With preemptive algorithms, the running task can be interrupted at any time to assign the processor to another active task, according to a predefined scheduling policy.
- **Non-preemptive**. With non-preemptive algorithms, a task, once started, is executed by the processor until completion. In this case, all scheduling decisions are taken as a task terminates its execution.
- **Static**. Static algorithms are those in which scheduling decisions are based on fixed parameters, assigned to tasks before their activation.
- **Dynamic**. Dynamic algorithms are those in which scheduling decisions are based on dynamic parameters that may change during system evolution.
- **Off-line**. We say that a scheduling algorithm is used off-line if it is executed on the entire task set before actual task activation. The schedule generated in this way is stored in a table and later executed by a dispatcher.
- **On-line**. We say that a scheduling algorithm is used on-line if scheduling decisions are taken at runtime every time a new task enters the system or when a running task terminates.
- **Optimal**. An algorithm is said to be optimal if it minimizes some given cost function defined over the task set. When no cost function is defined and the only concern is to achieve a feasible schedule, then an algorithm is said to be optimal if it may fail to meet a deadline only if no other algorithms of the same class can meet it.
- **Heuristic**. An algorithm is said to be heuristic if it tends toward but does not guarantee to find the optimal schedule.

Moreover, an algorithm is said to be *clairvoyant* if it knows the future; that is, if it knows in advance the arrival times of all the tasks. Although such an

algorithm does not exist in reality, it can be used for comparing the performance of real algorithms against the best possible one.

Guarantee-based algorithms

In hard real-time applications that require highly predictable behavior, the feasibility of the schedule should be guaranteed in advance; that is, before task execution. In this way, if a critical task cannot be scheduled within its deadline, the system is still in time to execute an alternative action, attempting to avoid catastrophic consequences. In order to check the feasibility of the schedule before tasks' execution, the system has to plan its actions by looking ahead in the future and by assuming a worst-case scenario.

In static real-time systems, where the task set is fixed and known a priori, all task activations can be precalculated off-line, and the entire schedule can be stored in a table that contains all guaranteed tasks arranged in the proper order. Then, at runtime, a simple dispatcher simply removes the next task from the table and puts it in the running state. The main advantage of the static approach is that the run-time overhead does not depend on the complexity of the scheduling algorithm. This allows very sophisticated algorithms to be used to solve complex problems or find optimal scheduling sequences. On the other hand, however, the resulting system is quite inflexible to environmental changes; thus, predictability strongly relies on the observance of the hypotheses made on the environment.

In dynamic real-time systems, since new tasks can be activated at runtime, the guarantee must be done *on-line* every time a new task enters the system. A scheme of the guarantee mechanism typically adopted in dynamic real-time systems is illustrated in Figure 2.12.

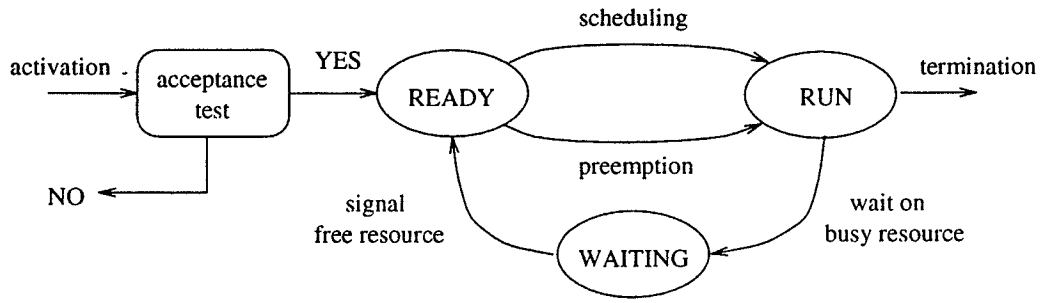

Figure 2.12 Scheme of the guarantee mechanism used in dynamic hard real-time systems.

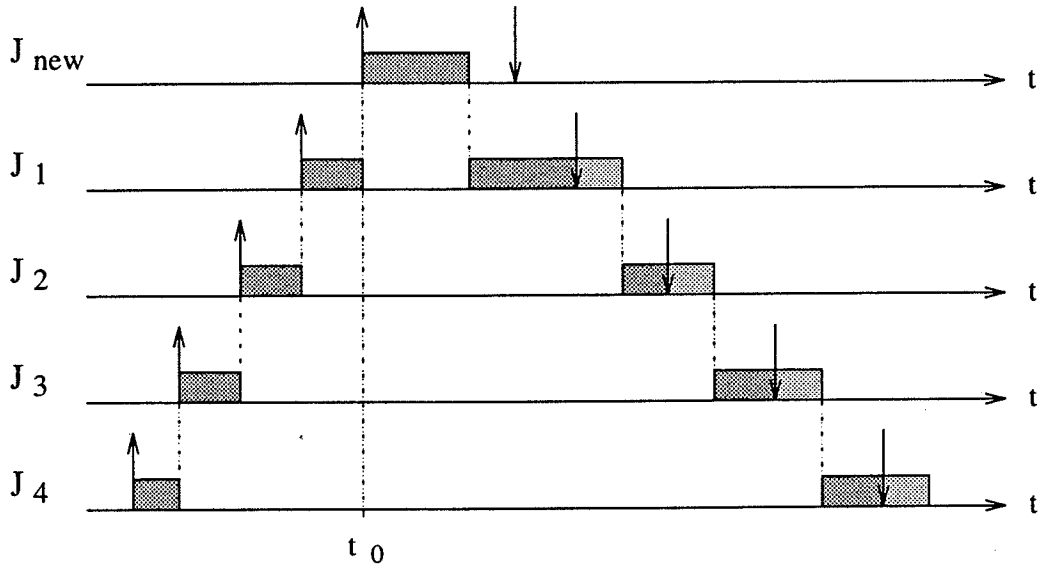

Figure 2.13 Example of domino effect.

If J is the current task set that has been previously guaranteed, a newly arrived task J_{new} is accepted into the system if and only if the task set $J' = J \cup \{J_{new}\}$ is found schedulable. If J' is not schedulable, then task J_{new} is rejected to preserve the feasibility of the current task set.

It is worth to notice that, since the guarantee mechanism is based on worst-case assumptions, a task could unnecessarily be rejected. This means that the guarantee of hard tasks is achieved at the cost of reducing the average performance of the system. On the other hand, the benefit of having a guarantee mechanism is that potential overload situations can be detected in advance to avoid negative effects on the system. One of the most dangerous phenomena caused by a transient overload is called *domino effect*. It refers to the situation in which the arrival of a new task causes *all* previously guaranteed tasks to miss their deadlines. Let us consider for example the situation depicted in Figure 2.13, where tasks are scheduled based on their absolute deadlines.

At time t_0, if task J_{new} was accepted, all other tasks (previously schedulable) would miss their deadlines. In planned-based algorithms, this situation is detected at time t_0, when the guarantee is performed and causes task J_{new} to be rejected.

In summary, the guarantee test ensures that, once a task is accepted, it will complete within its deadline and, moreover, its execution will not jeopardize the feasibility of the tasks that have been previously guaranteed.