
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 term *process* is used as synonym of *task* and *thread*. However, it is worth saying that some authors prefer to distinguish them and define a process as a more complex entity that can be composed by many concurrent tasks (or threads) sharing a common memory space.

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

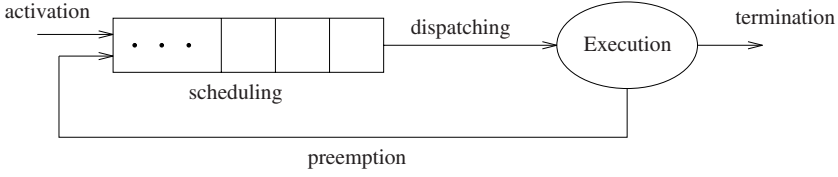


Figure 2.1 Queue of ready tasks waiting for execution.

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 that 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 tasks have different levels of criticality (expressing task importance), preemption permits executing the most critical tasks, as soon as they arrive.
- Preemptive scheduling typically allows higher efficiency, in the sense that it allows executing a real-time task sets with higher processor utilization.

On the other hand, preemption destroys program locality and introduces a runtime overhead that inflates the execution time of tasks. As a consequence, limiting preemptions in real-time schedules can have beneficial effects in terms of schedulability. This issue will be investigated in Chapter 8.

Given a set of tasks, $J = \{J_1, \dots, J_n\}$, 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 .

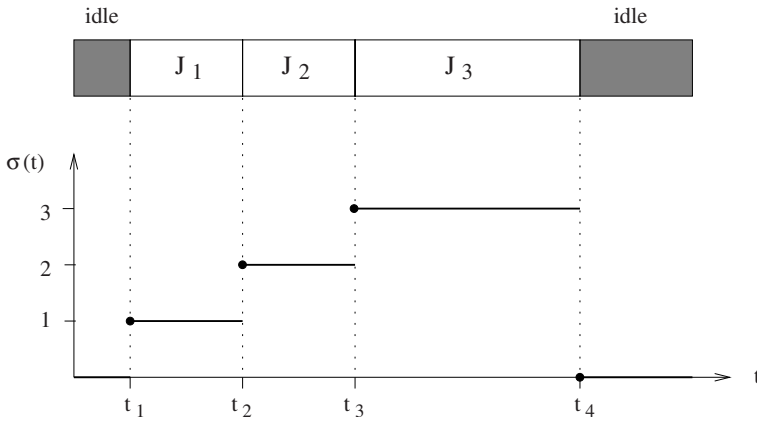


Figure 2.2 Schedule obtained by executing three tasks J_1 , J_2 , and J_3 .

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

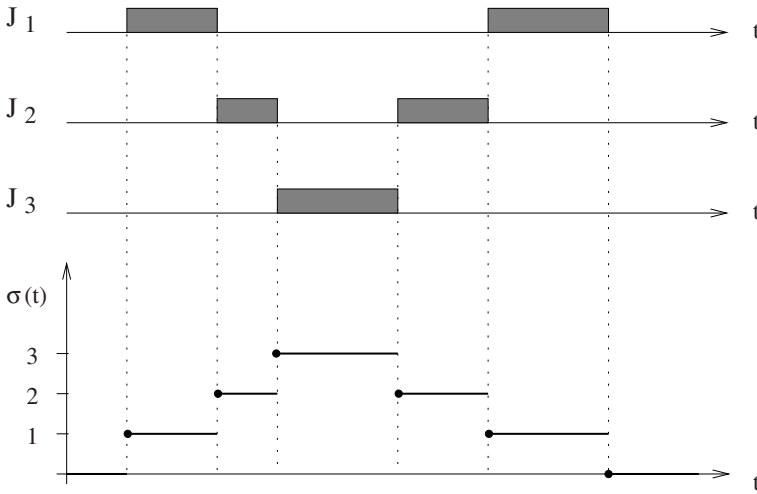


Figure 2.3 Example of a preemptive schedule.

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. If a deadline is specified with respect to the task arrival time, it is called a *relative deadline*, whereas if it is specified with respect to time zero, it is called an *absolute deadline*. Depending on the consequences of a missed deadline, real-time tasks are usually distinguished in three categories:

- **Hard:** A real-time task is said to be *hard* if missing its deadline may cause catastrophic consequences on the system under control.
- **Firm:** A real-time task is said to be *firm* if missing its deadline does not cause any damage to the system, but the output has no value.
- **Soft:** A real-time task is said to be *soft* if missing its deadline has still some utility for the system, although causing a performance degradation.

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

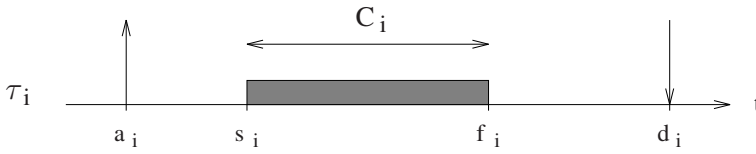


Figure 2.4 Typical parameters of a real-time task.

- **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 ;
- **Computation time** C_i is the time necessary to the processor for executing the task without interruption;
- **Absolute Deadline** d_i is the time before which a task should be completed to avoid damage to the system;
- **Relative Deadline** D_i is the difference between the absolute deadline and the request time: $D_i = d_i - r_i$;
- **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;
- **Response time** R_i is the difference between the finishing time and the request time: $R_i = f_i - r_i$;
- **Criticality** is a parameter related to the consequences of missing the deadline (typically, it can be hard, firm, 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.

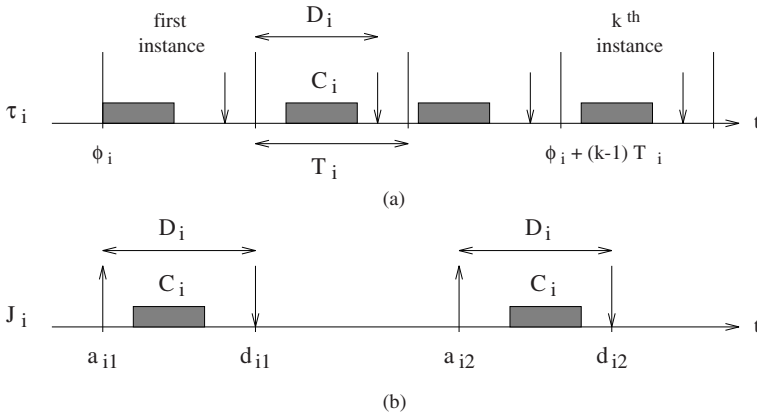


Figure 2.5 Sequence of instances for a periodic task (a) and an aperiodic job (b).

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 . The generic k^{th} job of a periodic task τ_i will be denoted by $\tau_{i,k}$.

The activation time of the first periodic instance ($\tau_{i,1}$) is called *phase*. If ϕ_i is the phase of task τ_i , the activation time of the k^{th} instance is given by $\phi_i + (k-1)T_i$, where T_i is the activation *period* of the task. In many practical cases, a periodic process can be completely characterized by its phase ϕ_i , its computation time C_i , its period T_i , and its relative deadline D_i .

Aperiodic tasks also consist of an infinite sequence of identical jobs (or instances); however, their activations are not regularly interleaved. An aperiodic task where consecutive jobs are separated by a minimum inter-arrival time is called a *sporadic task*. Figure 2.5 shows an example of task instances for a periodic and 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

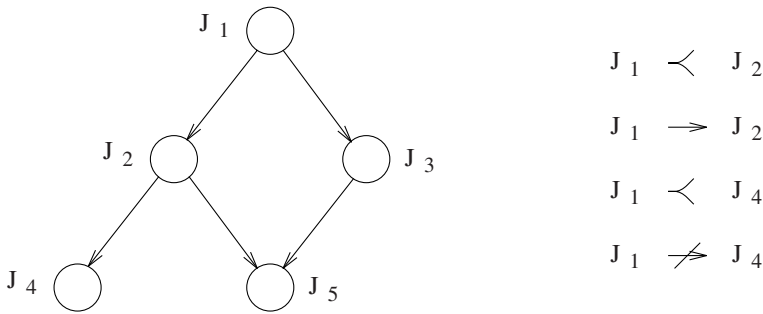


Figure 2.6 Precedence relations among five tasks.

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 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 for 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:

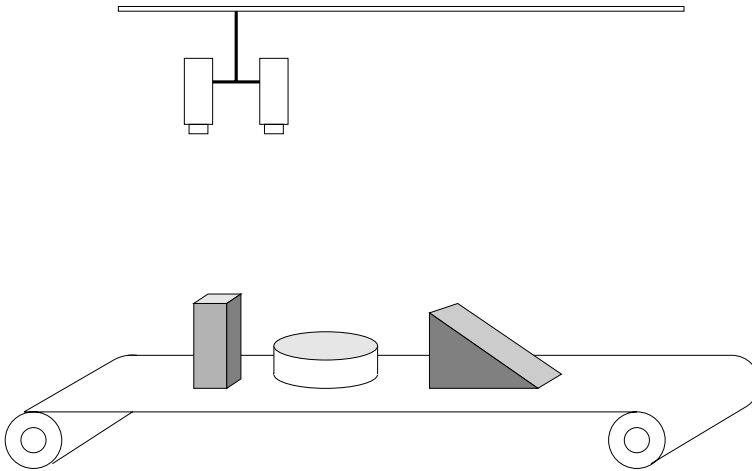


Figure 2.7 Industrial application that requires a visual recognition of objects on a conveyor belt.

- 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*);
- 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

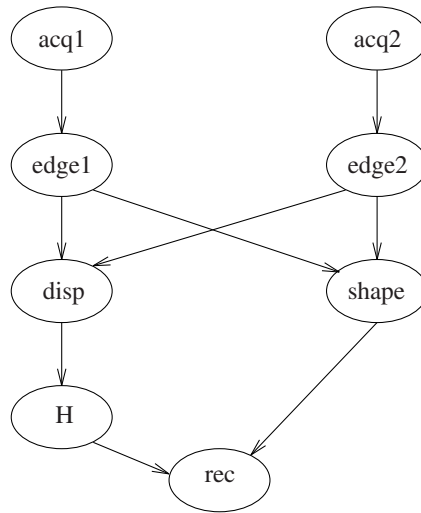


Figure 2.8 Precedence task graph associated with the industrial application illustrated in Figure 2.7.

acquisition task completes. Task *shape* is based on the object contour extracted by the low-level image processing; therefore, it must wait for 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.

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 by competing tasks, but require their mutual exclusion. This means that a task cannot access a resource *R* if another task is inside *R* manipulating its data structures. In this case, *R* is called a *mutually exclusive resource*. A piece of code executed under mutual exclusion constraints is called a *critical section*.

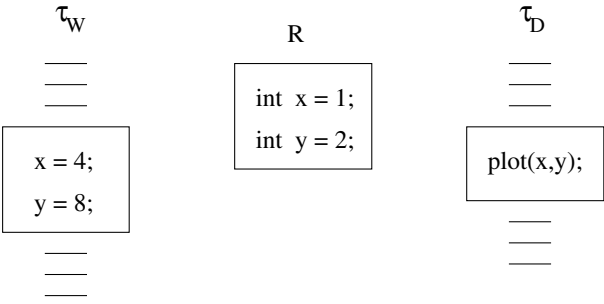


Figure 2.9 Two tasks sharing a buffer with two variables.

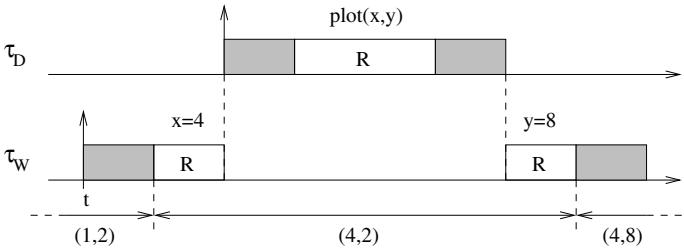


Figure 2.10 Example of schedule creating data inconsistency.

To better understand why mutual exclusion is important for guaranteeing data consistency, consider the application illustrated in Figure 2.9, where two tasks cooperate to track a moving object: task τ_W gets the object coordinates from a sensor and writes them into a shared buffer R , containing two variables (x, y) ; task τ_D reads the variables from the buffer and plots a point on the screen to display the object trajectory.

If the access to the buffer is not mutually exclusive, task τ_W (having lower priority than τ_D) may be preempted while updating the variables, so leaving the buffer in an inconsistent state. The situation is illustrated in Figure 2.10, where, at time t , the (x, y) variables have values $(1, 2)$. If τ_W is preempted after updating x and before updating y , τ_D will display the object in $(4, 2)$, which is neither the old nor the new position.

To ensure a correct access to exclusive resources, operating systems provide a synchronization mechanism (e.g., semaphores) that can be used to create critical sections of code. In the following, when we say that two or more tasks have resource constraints, we mean that they share resources that are accessed through a synchronization mechanism.

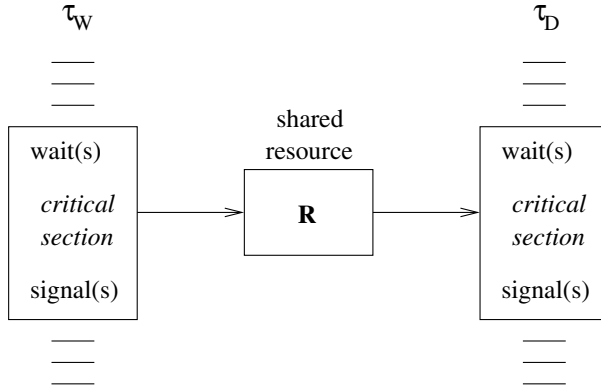


Figure 2.11 Structure of two tasks that share a mutually exclusive resource protected by a semaphore.

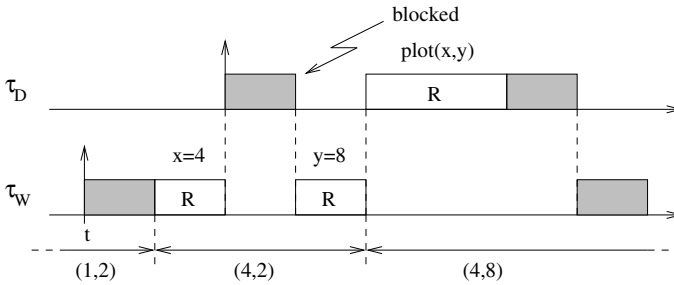


Figure 2.12 Example of schedule when the resource is protected by a semaphore.

To avoid the problem illustrated in Figure 2.10, both tasks have to encapsulate the instructions that manipulate the shared variables into a critical section. If a binary semaphore s is used for this purpose, then each critical section must begin with a $\text{wait}(s)$ primitive and must end with a $\text{signal}(s)$ primitive, as shown in Figure 2.11.

If the resource is free, the $\text{wait}(s)$ primitive executed by τ_W notifies that a task is using the resource, which becomes locked until the task executes the $\text{signal}(s)$. Hence, if τ_D preempts τ_W inside the critical section, it is blocked as soon as it executes $\text{wait}(s)$, and the processor is given back to τ_W . When τ_W exits its critical section by executing $\text{signal}(s)$, then τ_D is resumed and the processor is given to the ready task with the highest priority. The resulting schedule is shown in Figure 2.12.

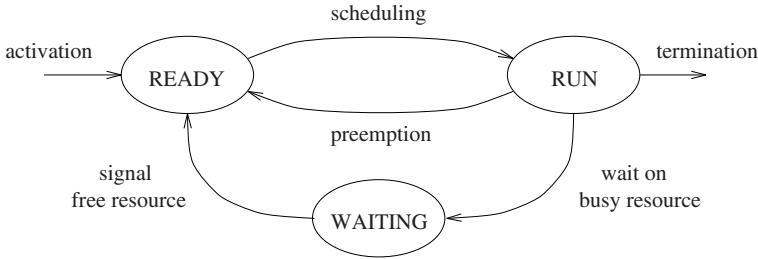


Figure 2.13 Waiting state caused by resource constraints.

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 protecting 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. Note that 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 2.13.

2.3 DEFINITION OF SCHEDULING PROBLEMS

In general, to define a scheduling problem we need to specify three sets: a set of n tasks $\Gamma = \{\tau_1, \tau_2, \dots, \tau_n\}$, a set of m processors $P = \{P_1, P_2, \dots, P_m\}$ and a set of s types of resources $R = \{R_1, R_2, \dots, R_s\}$. 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 assigning processors from P and resources from R to tasks from Γ in order to complete all tasks under the specified 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 *nondeterministic* 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 \mathbf{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 $1 \mu s$. 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 $30 \mu s$, 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 in real-time scheduling is to identify simpler, but still practical, problems that can be solved in polynomial time.

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, the following main classes can be identified:

■ Preemptive vs. Non-preemptive.

- In 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.
- In non-preemptive algorithms, a task, once started, is executed by the processor until completion. In this case, all scheduling decisions are taken as the task terminates its execution.

■ **Static vs. Dynamic.**

- Static algorithms are those in which scheduling decisions are based on fixed parameters, assigned to tasks before their activation.
- Dynamic algorithms are those in which scheduling decisions are based on dynamic parameters that may change during system evolution.

■ **Off-line vs. Online.**

- A scheduling algorithm is used off line if it is executed on the entire task set before tasks activation. The schedule generated in this way is stored in a table and later executed by a dispatcher.
- A scheduling algorithm is used online if scheduling decisions are taken at runtime every time a new task enters the system or when a running task terminates.

■ **Optimal vs. Heuristic.**

- 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 is able to find a feasible schedule, if one exists.
- An algorithm is said to be heuristic if it is guided by a heuristic function in taking its scheduling decisions. A heuristic algorithm tends toward the optimal schedule, but does not guarantee finding it.

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 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 runtime 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 (typically consisting of firm tasks), tasks can be created at runtime; hence the guarantee must be done *online* every time a new task is created. A scheme of the guarantee mechanism typically adopted in dynamic real-time systems is illustrated in Figure 2.14.

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

It is worth noting that since the guarantee mechanism is based on worst-case assumptions a task could unnecessarily be rejected. This means that the guarantee of firm tasks is achieved at the cost of a lower efficiency. 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 2.15, where five jobs are scheduled based on their absolute deadlines.

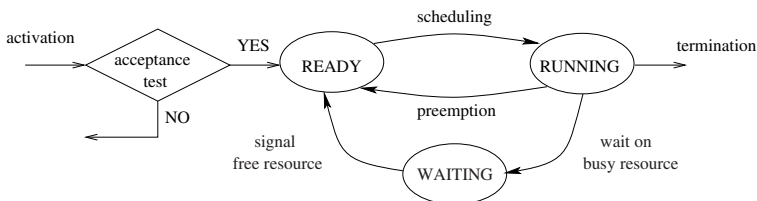


Figure 2.14 Scheme of the guarantee mechanism used in dynamic real-time systems.

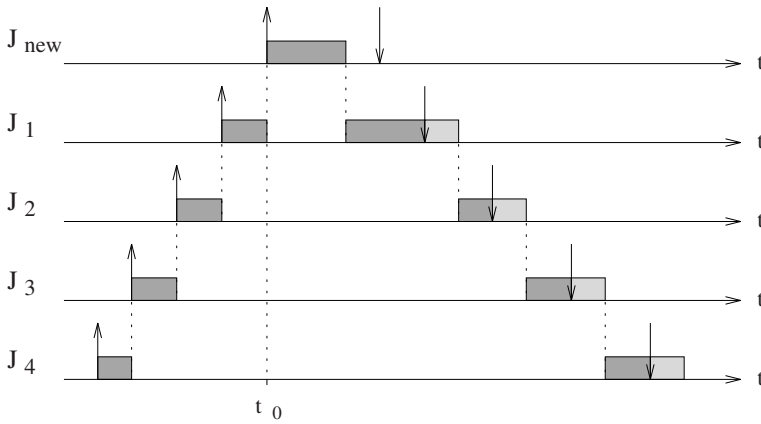


Figure 2.15 Example of domino effect.

At time t_0 , if job J_{new} were accepted, all the other jobs (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 job 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.

BEST-EFFORT ALGORITHMS

In certain real-time applications, computational activities have soft timing constraints that should be met whenever possible to satisfy system requirements. In these systems, missing soft deadlines do not cause catastrophic consequences, but only a performance degradation.

For example, in typical multimedia applications, the objective of the computing system is to handle different types of information (such as text, graphics, images, and sound) in order to achieve a certain quality of service for the users. In this case, the timing constraints associated with the computational activities depend on the quality of service requested by the users; hence, missing a deadline may only affect the performance of the system.

To efficiently support soft real-time applications that do not have hard timing requirements, a *best-effort* approach may be adopted for scheduling.

A best-effort scheduling algorithm tries to “do its best” to meet deadlines, but there is no guarantee of finding a feasible schedule. In a best-effort approach, tasks may be enqueued according to policies that take time constraints into account; however, since feasibility is not checked, a task may be aborted during its execution. On the other hand, best-effort algorithms perform much better than guarantee-based schemes in the average case. In fact, whereas the pessimistic assumptions made in the guarantee mechanism may unnecessarily cause task rejections, in best-effort algorithms a task is aborted only under real overload conditions.

2.3.2 METRICS FOR PERFORMANCE EVALUATION

The performance of scheduling algorithms is typically evaluated through a cost function defined over the task set. For example, classical scheduling algorithms try to minimize the average response time, the total completion time, the weighted sum of completion times, or the maximum lateness. When deadlines are considered, they are usually added as constraints, imposing that all tasks must meet their deadlines. If some deadlines cannot be met with an algorithm *A*, the schedule is said to be infeasible by *A*. Table 2.1 shows some common cost functions used for evaluating the performance of a scheduling algorithm.

The metrics adopted in the scheduling algorithm has strong implications on the performance of the real-time system [SSDNB95], and it must be carefully chosen according to the specific application to be developed. For example, the average response time is generally not of interest for real-time applications, because there is not direct assessment of individual timing properties such as periods or deadlines. The same is true for minimizing the total completion time. The weighted sum of completion times is relevant when tasks have different importance values that they impart to the system on completion. Minimizing the maximum lateness can be useful at design time when resources can be added until the maximum lateness achieved on the task set is less than or equal to zero. In that case, no task misses its deadline. In general, however, minimizing the maximum lateness does not minimize the number of tasks that miss their deadlines and does not necessarily prevent one or more tasks from missing their deadline.

Let us consider, for example, the case depicted in Figure 2.16. The schedule shown in Figure 2.16a minimizes the maximum lateness, but all tasks miss their deadline. On the other hand, the schedule shown in Figure 2.16b has a greater maximum lateness, but four tasks out of five complete before their deadline.

Average response time:

$$\overline{t_r} = \frac{1}{n} \sum_{i=1}^n (f_i - a_i)$$

Total completion time:

$$t_c = \max_i(f_i) - \min_i(a_i)$$

Weighted sum of completion times:

$$t_w = \sum_{i=1}^n w_i f_i$$

Maximum lateness:

$$L_{max} = \max_i(f_i - d_i)$$

Maximum number of late tasks:

$$N_{late} = \sum_{i=1}^n miss(f_i)$$

where

$$miss(f_i) = \begin{cases} 0 & \text{if } f_i \leq d_i \\ 1 & \text{otherwise} \end{cases}$$

Table 2.1 Example of cost functions.

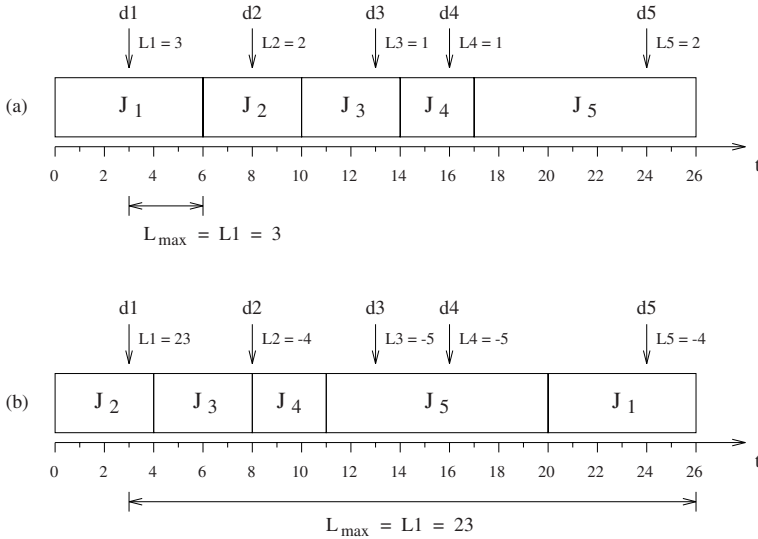


Figure 2.16 The schedule in (a) minimizes the maximum lateness, but all tasks miss their deadline. The schedule in (b) has a greater maximum lateness, but four tasks out of five complete before their deadline.

When tasks have soft deadlines and the application goal is to meet as many deadlines as possible (without a priori guarantee), then the scheduling algorithm should use a cost function that minimizes the number of late tasks.

In other applications, the benefit of executing a task may depend not only on the task importance but also on the time at which it is completed. This can be described by means of specific *utility functions*, which describe the value associated with the task as a function of its completion time.

Figure 2.17 illustrates some typical utility functions that can be defined on the application tasks. For instance, non-real-time tasks (a) do not have deadlines, thus the value achieved by the system is proportional to the task importance and does not depend on the completion time. Soft tasks (b) have noncritical deadlines; therefore, the value gained by the system is constant if the task finishes before its deadline but decreases with the exceeding time. In some cases (c), it is required to execute a task *on-time*; that is, not too early and not too late with respect to a given deadline. Hence, the value achieved by the system is high if the task is completed around the deadline, but it rapidly decreases with the absolute value of the lateness. Such types of constraints are typical when playing notes, since the human ear is quite sensitive to time jitter.

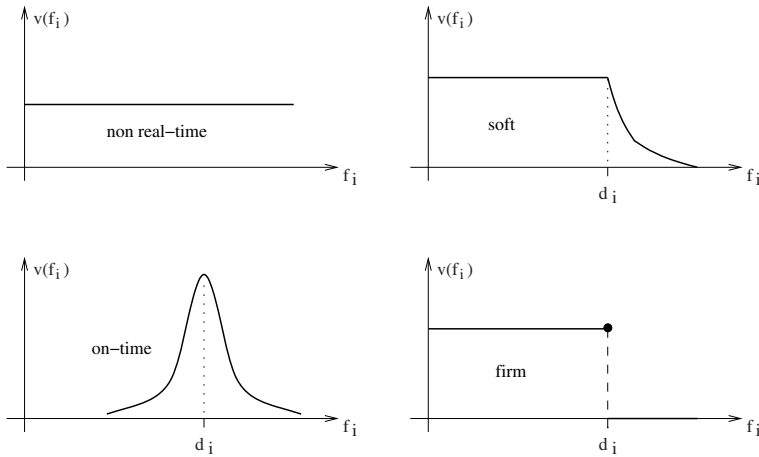


Figure 2.17 Example of cost functions for different types of tasks.

In other cases (d), executing a task after its deadline does not cause catastrophic consequences, but there is no benefit for the system, thus the utility function is zero after the deadline.

When utility functions are defined on the tasks, the performance of a scheduling algorithm can be measured by the *cumulative value*, given by the sum of the utility functions computed at each completion time:

$$Cumulative_value = \sum_{i=1}^n v(f_i).$$

This type of metrics is very useful for evaluating the performance of a system during overload conditions, and it is considered in more detail in Chapter 9.

2.4 SCHEDULING ANOMALIES

In this section we describe some singular examples that clearly illustrate that real-time computing is not equivalent to fast computing, since, for example, an increase of computational power in the supporting hardware does not always cause an improvement of performance. These particular situations, called Richard's anomalies, were described by Graham in 1976 and refer to task sets with precedence relations executed in a multiprocessor environment.

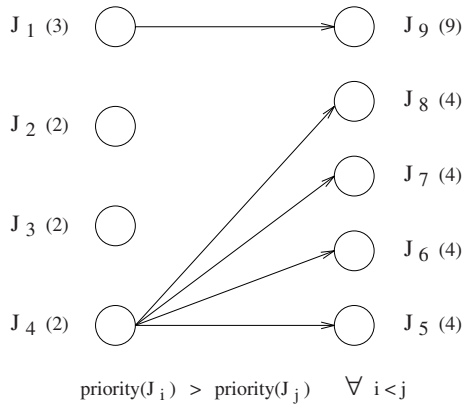


Figure 2.18 Precedence graph of the task set J ; numbers in parentheses indicate computation times.

Designers should be aware of such insidious anomalies to take the proper countermeasures to avoid them. The most important anomalies are expressed by the following theorem [Gra76, SSDNB95]:

Theorem 2.1 (Graham, 1976) *If a task set is optimally scheduled on a multiprocessor with some priority assignment, a fixed number of processors, fixed execution times, and precedence constraints, then increasing the number of processors, reducing execution times, or weakening the precedence constraints can increase the schedule length.*

This result implies that, if tasks have deadlines, then adding resources (for example, an extra processor) or relaxing constraints (less precedence among tasks or fewer execution times requirements) can make things worse. A few examples can best illustrate why this theorem is true.

Let us consider a task set consisting of nine jobs $J = \{J_1, J_2, \dots, J_9\}$, sorted by decreasing priorities, so that J_i priority is greater than J_j priority if and only if $i < j$. Moreover, jobs are subject to precedence constraints that are described through the graph shown in Figure 2.18. Computation times are indicated in parentheses.

If this task set is executed on a parallel machine with three processors, where the highest priority task is assigned to the first available processor, the resulting schedule σ^* is illustrated in Figure 2.19, where the global completion time is $t_c = 12$ units of time.

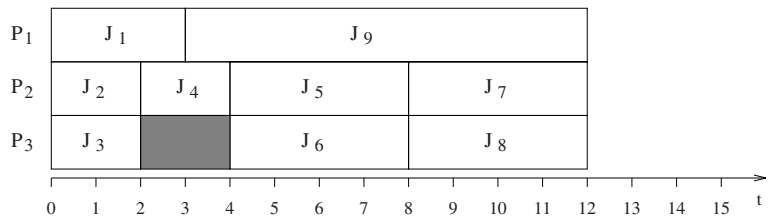


Figure 2.19 Optimal schedule of task set J on a three-processor machine.

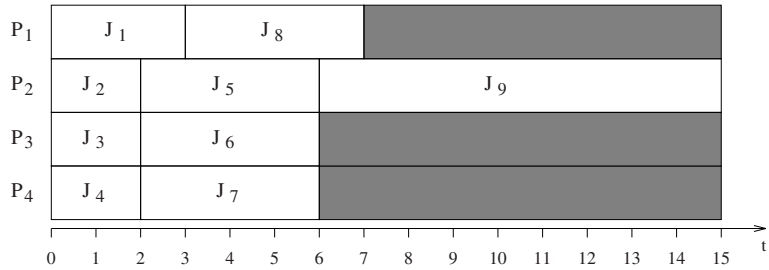


Figure 2.20 Schedule of task set J on a four-processor machine.

Now we will show that adding an extra processor, reducing tasks’ execution times, or weakening precedence constraints will increase the global completion time of the task set.

NUMBER OF PROCESSORS INCREASED

If we execute the task set J on a more powerful machine consisting of four processors, we obtain the schedule illustrated in Figure 2.20, which is characterized by a global completion time of $t_c = 15$ units of time.

COMPUTATION TIMES REDUCED

One could think that the global completion time of the task set J could be improved by reducing tasks’ computation times of each task. However, we can surprisingly see that, reducing the computation time of each task by one unit of time, the schedule length will increase with respect to the optimal schedule σ^* , and the global completion time will be $t_c = 13$, as shown in Figure 2.21.

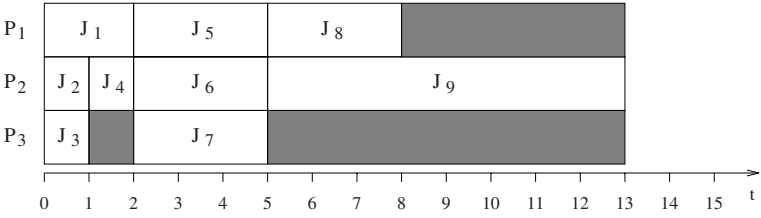


Figure 2.21 Schedule of task set J on three processors, with computation times reduced by one unit of time.

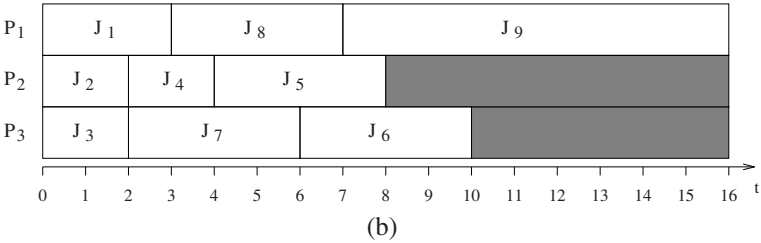
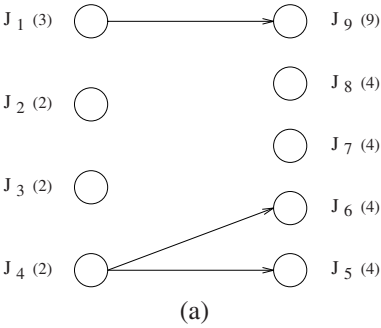


Figure 2.22 a. Precedence graph of task set J obtained by removing the constraints on tasks J_7 and J_8 . b. Schedule of task set J on three processors, with precedence constraints weakened.

PRECEDENCE CONSTRAINTS WEAKENED

Scheduling anomalies can also arise by removing precedence constraints from the directed acyclic graph depicted in Figure 2.18. For instance, removing the precedence relations between job J_4 and jobs J_7 and J_8 (see Figure 2.22a), we obtain the schedule shown in Figure 2.22b, which is characterized by a global completion time of $t_c = 16$ units of time.

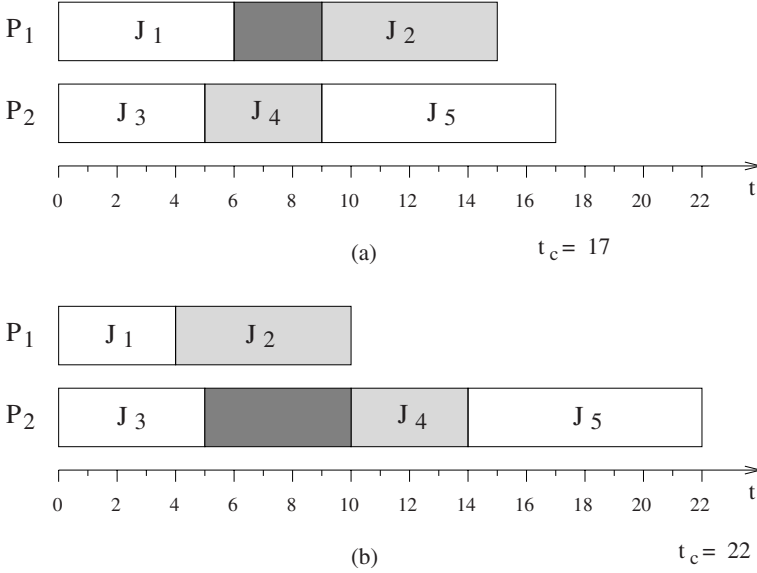


Figure 2.23 Example of anomaly under resource constraints. If J_2 and J_4 share the same resource in exclusive mode, the optimal schedule length (a) increases if the computation time of job J_1 is reduced (b). Jobs are statically allocated on the processors.

ANOMALIES UNDER RESOURCE CONSTRAINTS

The following example shows that, in the presence of shared resources, the schedule length of a task set can increase when reducing tasks' computation times. Consider the case illustrated in Figure 2.23, where five jobs are statically allocated on two processors: jobs J_1 and J_2 on processor P1, and jobs J_3 , J_4 and J_5 on processor P2 (jobs are indexed by decreasing priority). Moreover, jobs J_2 and J_4 share the same resource in exclusive mode; hence their execution cannot overlap in time. A schedule of this task set is shown in Figure 2.23a, where the total completion time is $t_c = 17$.

If we now reduce the computation time of job J_1 on the first processor, then J_2 can begin earlier and take the resource before J_4 . As a consequence, job J_4 must now block over the shared resource and possibly miss its deadline. This situation is illustrated in Figure 2.23b. As we can see, the blocking time experienced by J_4 causes a delay in the execution of J_5 (which may also miss its deadline), increasing the total completion time of the task set from 17 to 22.

Notice that the scheduling anomaly illustrated by the previous example is particularly insidious for hard real-time systems, because tasks are guaranteed based on their worst-case behavior, but they may complete before their worst-case computation time. A simple solution that avoids the anomaly is to keep the processor idle if tasks complete earlier, but this can be very inefficient. There are algorithms, such as the one proposed by Shen [SRS93], that tries to reclaim such an idle time, while addressing the anomalies so that they will not occur.

If tasks share mutually exclusive resources, scheduling anomalies can also occur in a uniprocessor system when changing the processor speed [But06]. In particular, the anomaly can be expressed as follows:

A real-time application that is feasible on a given processor can become infeasible when running on a faster processor.

Figure 2.24 illustrates a simple example where two tasks, τ_1 and τ_2 , share a common resource (critical sections are represented by light grey areas). Task τ_1 has a higher priority, arrives at time $t = 2$ and has a relative deadline $D_1 = 7$. Task τ_2 , having lower priority, arrives at time $t = 0$ and has a relative deadline $D_2 = 23$. Suppose that, when the tasks are executed at a certain speed S_1 , τ_1 has a computation time $C_1 = 6$, (where 2 units of time are spent in the critical section), whereas τ_2 has a computation time $C_2 = 18$ (where 12 units of time are spent in the critical section). As shown in Figure 2.24a, if τ_1 arrives just before τ_2 enters its critical section, it is able to complete before its deadline, without experiencing any blocking. However, if the same task set is executed at a double speed $S_2 = 2S_1$, τ_1 misses its deadline, as clearly illustrated in Figure 2.24b. This happens because, when τ_1 arrives, τ_2 already granted its resource, causing an extra blocking in the execution of τ_1 , due to mutual exclusion.

Although the average response time of the task set is reduced on the faster processor (from 14 to 9.5 units of time), note that the response time of task τ_1 increases when doubling the speed, because of the extra blocking on the shared resource.

ANOMALIES UNDER NON-PREEMPTIVE SCHEDULING

Similar situations can occur in non-preemptive scheduling. Figure 2.25 illustrates an anomalous behavior occurring in a set of three real-time tasks, τ_1 , τ_2 and τ_3 , running in a non-preemptive fashion. Tasks are assigned a fixed priority proportional to their relative deadline, thus τ_1 is the task with the highest priority and τ_3 is the task with the lowest priority. As shown in Figure 2.25a, when tasks are executed at speed S_1 , τ_1 has a computation time $C_1 = 2$ and completes at time $t = 6$. However, if the same

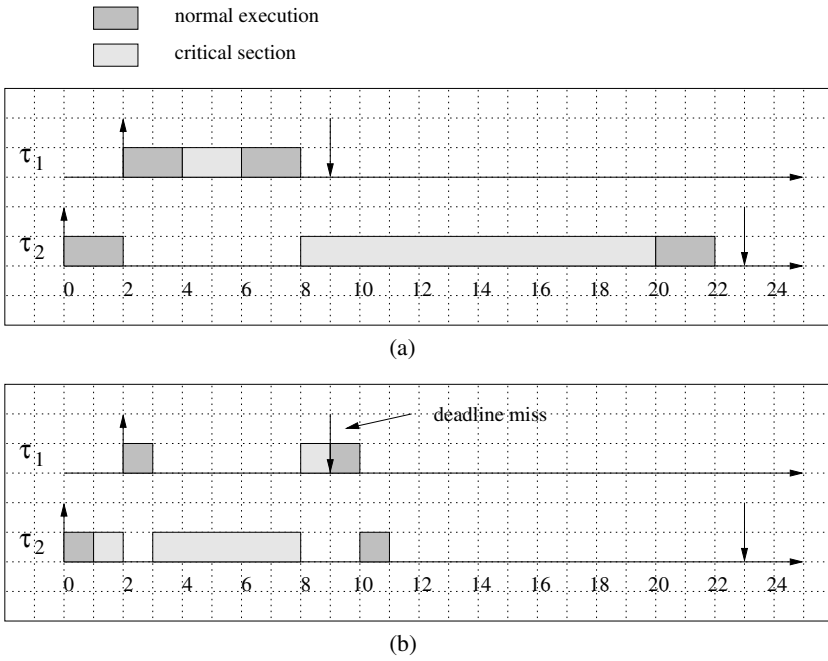


Figure 2.24 Scheduling anomaly in the presence of resource constraints: task τ_1 meets its deadline when the processor is executing at a certain speed S_1 (a), but misses its deadline when the speed is doubled (b).

task set is executed with double speed $S_2 = 2S_1$, τ_1 misses its deadline, as clearly illustrated in Figure 2.25b. This happens because, when τ_1 arrives, τ_3 already started its execution and cannot be preempted (due to the non-preemptive mode).

It is worth observing that a set of non-preemptive tasks can be considered as a special case of a set of tasks sharing a single resource (the processor) for their entire execution. According to this view, each task executes as it were inside a big critical section with a length equal to the task computation time. Once a task starts executing, it behaves as it were locking a common semaphore, thus preventing all the other tasks from taking the processor.

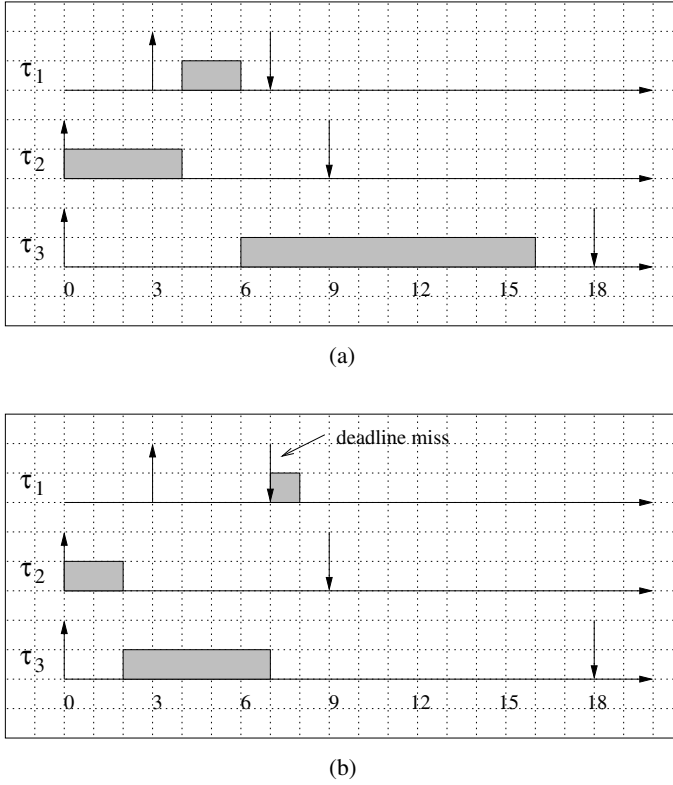


Figure 2.25 Scheduling anomaly in the presence of non-preemptive tasks: task τ_1 meets its deadline when the processor is executing at speed S_1 (a), but misses its deadline when the speed is doubled (b).

ANOMALIES USING A DELAY PRIMITIVE

Another timing anomaly can occur when tasks using shared resources explicitly suspend themselves through a *delay*(T) system call, which suspends the execution of the calling task for T units of time. Figure 2.26a shows a case in which τ_1 is feasible and has a slack time of 6 units when running at the highest priority, suggesting that it could easily tolerate a delay of two units. However, if τ_1 executes a *delay*(2) at time $t = 2$, it gives the opportunity to τ_2 to lock the shared resource. Hence, when τ_1 resumes, it has to block on the semaphore for 7 units, thus missing its deadline, as shown in Figure 2.26b.

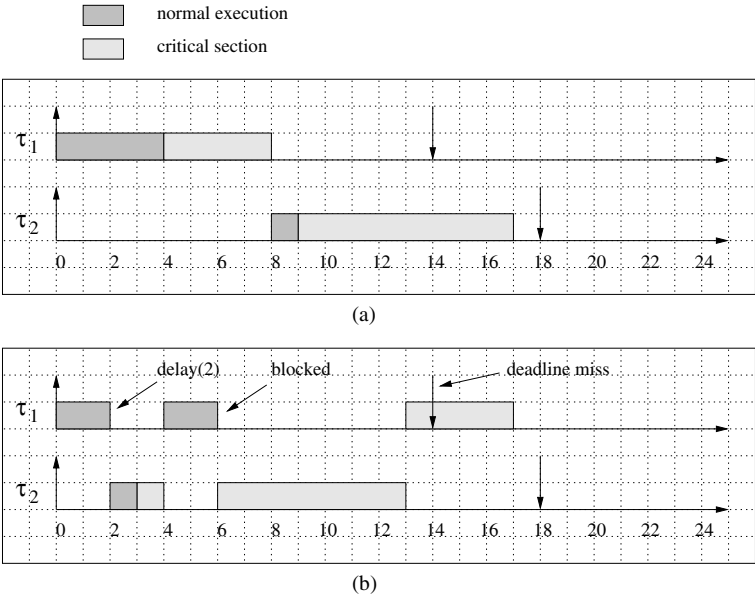


Figure 2.26 Scheduling anomaly in the presence of a delay: τ_1 has a slack of 6 units of time when running at the highest priority (a), but cannot tolerate a self suspension of two units (b).

The example shown in Figure 2.26 illustrates an anomaly in which a task with a large slack cannot tolerate a self suspension of a much smaller value. Figure 2.27 shows another case in which the suspension of a task can also cause a longer delay in a different task, even without sharing any resource. When τ_1 is assigned a higher priority than τ_2 , the resulting schedule shown in Figure 2.27a is feasible, with a slack for τ_1 of 3 units of time. However, if the third instance of τ_1 executes a *delay*(1) after one unit of execution, τ_2 will miss its deadline.

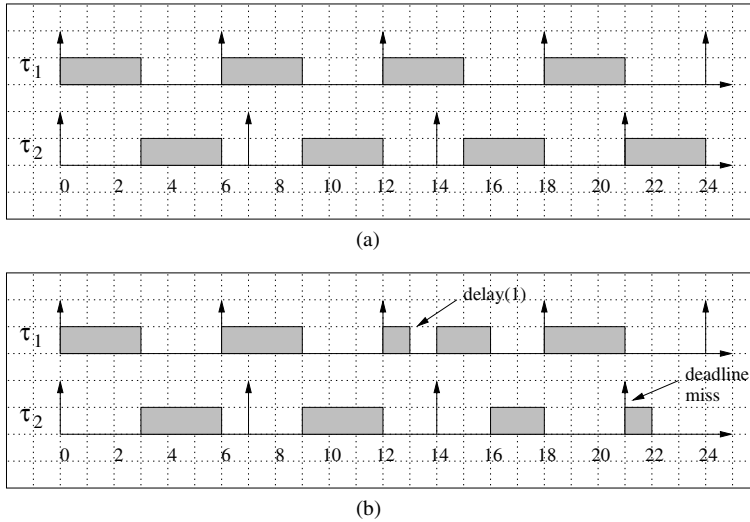


Figure 2.27 Scheduling anomaly in the presence of a delay: two tasks are feasible without delays (a), but a delay in τ_1 causes a deadline miss in τ_2 (b).

Exercises

- 2.1 Give the formal definition of a schedule, explaining the difference between preemptive and non-preemptive scheduling.
- 2.2 Explain the difference between periodic and aperiodic tasks, and describe the main timing parameters that can be defined for a real-time activity.
- 2.3 Describe a real-time application as a number of tasks with precedence relations, and draw the corresponding precedence graph.
- 2.4 Discuss the difference between static and dynamic, online and off-line, optimal, and heuristic scheduling algorithms.
- 2.5 Provide an example of domino effect, caused by the arrival of a task J^* , in a feasible set of three tasks.



<http://www.springer.com/978-1-4614-0675-4>

Hard Real-Time Computing Systems
Predictable Scheduling Algorithms and Applications
Buttazzo, G.
2011, XVI, 524 p., Hardcover
ISBN: 978-1-4614-0675-4