

1

Basic Concepts

1.1 Real-Time Applications

1.1.1 Real-time applications issues

In real-time applications, the timing requirements are the main constraints and their mastering is the predominant factor for assessing the quality of service. Timing constraints span many application areas, such as industrial plant automation, embedded systems, vehicle control, nuclear plant monitoring, scientific experiment guidance, robotics, multimedia audio and video stream conditioning, surgical operation monitoring, and stock exchange orders follow-up.

Applications trigger periodic or random events and require that the associated computer system reacts before a given delay or a fixed time. The timing latitude to react is limited since transient data must be caught, actions have a constraint on both start and finish times, and responses or commands must be sent on time.

The time scale may vary largely, its magnitude being a microsecond in a radar, a second in a human-machine interface, a minute in an assembly line, or an hour in a chemical reaction.

The source of timing constraints leads to classifying them as hard or soft. A real-time system has *hard timing constraints* when a timing fault (missing a deadline, delivering a message too late, sampling data irregularly, too large a scatter in data supposed to be collected simultaneously) may cause some human, economic or ecological disaster. A real-time system has *soft timing constraints* when timing faults can be dealt with to a certain extent.

A real-time computer system is a computer system whose behaviour is fixed by the dynamics of the application. Therefore, a real-time application consists of two connected parts: the controlling real-time computer system and the controlled process (Figure 1.1).

Time mastery is a serious challenge for real-time computer systems, and it is often misunderstood. The correctness of system reactions depends not only on the logical results of the computations, but also on the time at which the results are produced. Correct data which are available too late are useless; this is a timing fault (Burns and Wellings, 1997; Lelann, 1990; Stankovic, 1988).

A controlling real-time computer system may be built as:

- a cyclic generator, which periodically samples the state of the controlled process, computes the measured data and sends orders to the actuators (this is also called synchronous control);

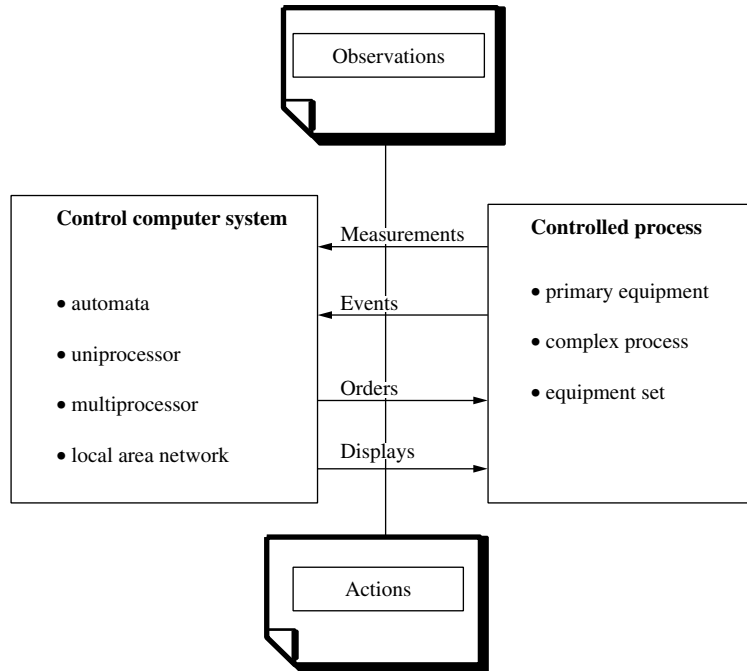


Figure 1.1 Scheme of a real-time application

- a reactive system, which responds instantaneously to the stimuli originating in the controlled process and thus is triggered by its dynamics;
- a union of both aspects, which schedules periodic and aperiodic tasks; this results in an asynchronous system.

1.1.2 Physical and logical architecture, operating systems

Software design of a real-time application

Several steps are usually identified to analyse and implement real-time applications. Some of them are:

- requirements analysis and functional and timing specifications, which result in a functional view (the question to answer is: what should the system do?).
- preliminary design, which performs an operational analysis (the question is: how to do it?) and leads to the choice of logical components of a logical architecture.
- specific hardware and software development. They are often developed concurrently with similar design processes. The hardware analysis (the question is: with which hardware units?) leads to a physical architecture, to the choice of commercial

off-the-shelf components and to the detailed design and development of special hardware. The conceptual analysis (the question is: with which software modules?) leads to a software architecture, to the choice of standard software components and to the implementation of customized ones. These acquisition and realization steps end with unit testing.

- integration testing, which involves combining all the software and hardware components, standard ones as well as specific ones, and performing global testing.
- user validation, which is carried out by measurements, sometimes combined with formal methods, and which is done prior to acceptance of the system.

These steps are summarized in Figure 1.2, which gives an overview of the main design and implementation steps of real-time applications. Once the logical and hardware architecture is defined, an allocation policy assigns the software modules to the hardware units. In distributed fault-tolerant real-time systems, the allocation may be undertaken dynamically and tasks may migrate. The operational analysis must define the basic logical units to map the requirements and to express concurrency in the system, which is our concern. The operational behaviour of the application is produced by their concurrent execution.

The major computing units are often classified as:

- passive objects such as physical resources (devices, sensors, actuators) or logical resources (memory buffers, files, basic software modules);
- communication objects such as messages or shared variables, ports, channels, network connections;
- synchronization objects such as events, semaphores, conditions, monitors (as in Modula), rendezvous and protected objects (as in Ada);

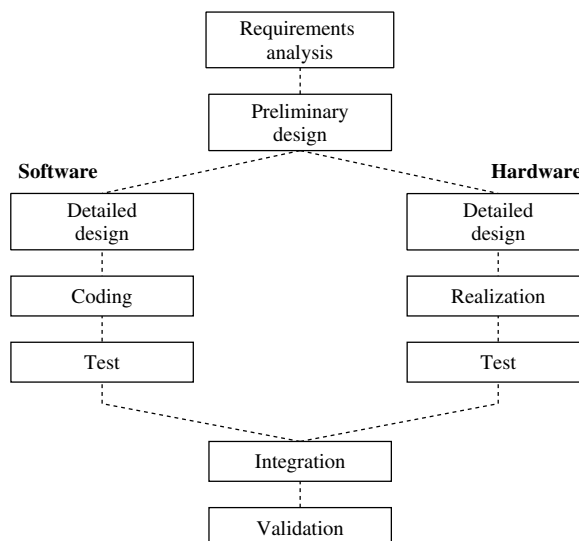


Figure 1.2 Joint hardware and software development

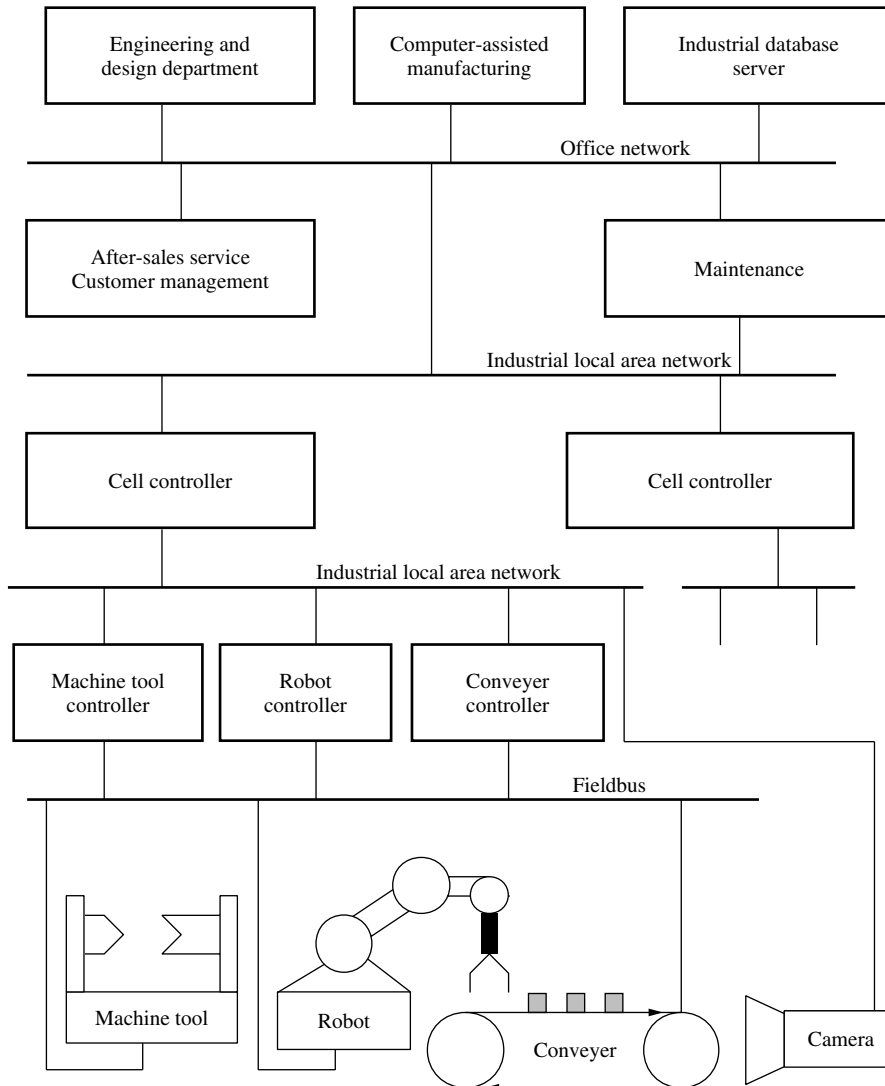


Figure 1.4 Example of a distributed architecture of real-time application

Logical architecture and real-time computing systems

Operating systems In order to locate real-time systems, let us briefly recall that computing systems may be classified, as shown by Figure 1.5, into transformational, interactive and reactive systems, which include asynchronous real-time systems.

The transformational aspect refers to systems where the results are computed with data available right from the program start and usable when required at any moment. The relational aspect between programming entities makes reference to systems where the environment-produced data are expected by programs already started; the results

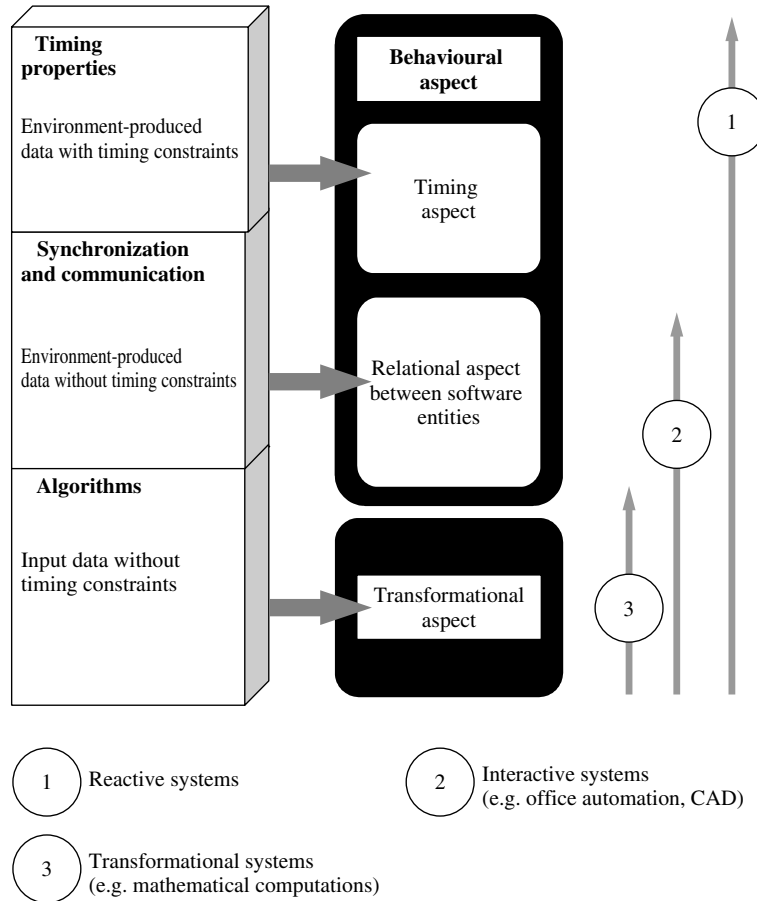


Figure 1.5 Classes of computing systems

of these programs are input to other programs. The timing aspect refers to systems where the results must be given at times fixed by the controlled process dynamics.

A system is centralized when information representing decisions, resource sharing, algorithms and data consistency is present in a shared memory and is directly accessible by all tasks of the system. This definition is independent of the hardware architecture. It refers to a uniprocessor or a shared memory multiprocessor architecture as well as to a distributed architecture where all decisions are only taken by one site. A system is distributed when the decisions are the result of a consensus among sites exchanging messages.

Distributed programming has to cope with uncertainty resulting from the lack of a common memory and common clock, from the variations of message transfer delays from one site to another as well as from one message to another, and from the existence of an important fault rate. Thus, identical information can never be captured simultaneously at all sites. As the time is one of these pieces of information, the sites are not able to read a common clock simultaneously and define instantaneously whether or not 'they have the same time'.

Computing systems are structured in layers. They all contain an operating system kernel as shown in Figure 1.6. This kernel includes mechanisms for the basic management of the processor, the virtual memory, interrupt handling and communication. More elaborate management policies for these resources and for other resources appear in the higher layers.

Conventional operating systems provide resource allocation and task scheduling, applying global policies in order to optimize the use of resources or to favour the response time of some tasks such as interactive tasks. All tasks are considered as aperiodic: neither their arrival times nor their execution times are known and they have no deadline.

In conventional operating systems the shared resources dynamically allocated to tasks are the main memory and the processor. Program behaviour investigations have indicated that the main memory is the sensitive resource (the most sensitive are demand paging systems with swapping between main memory and disk). Thus memory is allocated first according to allocation algorithms, which are often complicated, and the processor is allocated last. This simplifies processor scheduling since it concerns only the small subset of tasks already granted enough memory (Bawn, 1997; Silberscharz and Galvin, 1998; Tanenbaum, 1994; Tanenbaum and Woodhull, 1997). Conventional operating systems tend to optimize resource utilization, principally the main memory, and they do not give priority to deadline observances. This is a great difference with real-time operating systems.

Real-time operating systems In real-time systems, resources other than the processor are often statically allocated to tasks at their creation. In particular, time should not be wasted in dynamic memory allocation. Real-time files and databases are not stored on disks but reside in main memory; this avoids the non-deterministic disk track seeking and data access. Input–output management is important since the connections with the controlled process are various. Therefore, the main allocation parameter is processor time and this gives importance to the kernel and leads to it being named

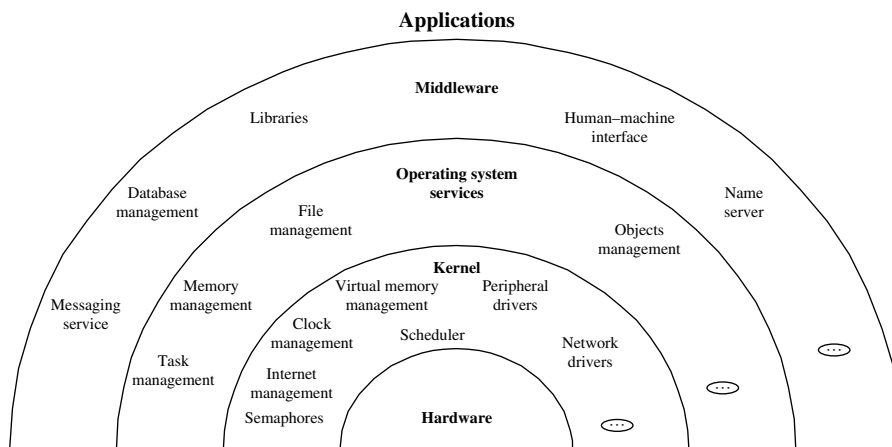


Figure 1.6 Structure of a conventional system

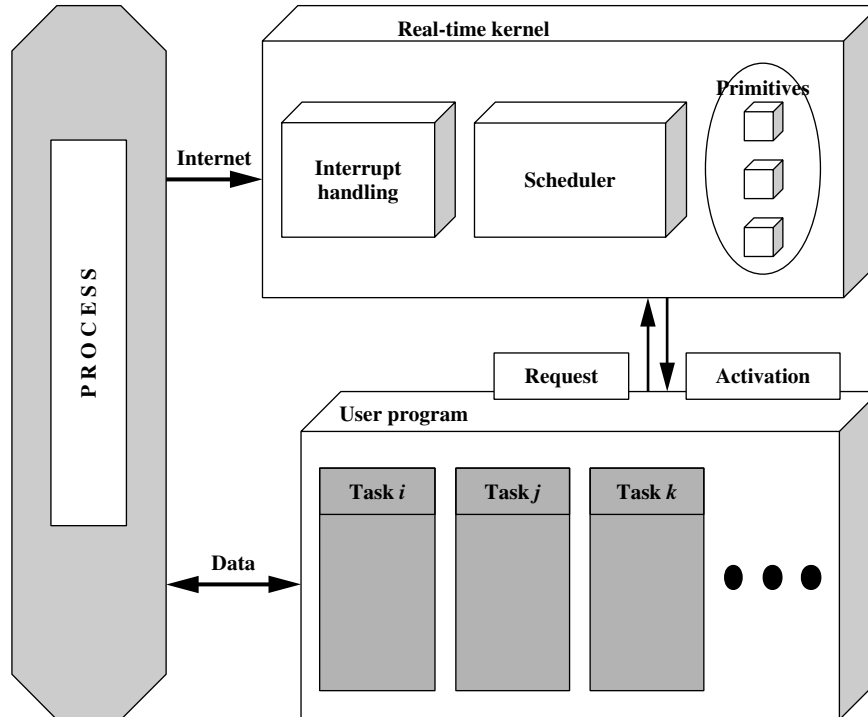


Figure 1.7 Schema of a real-time application

the real-time operating system (Figure 1.7). Nevertheless, conventional operating system services are needed by real-time applications that have additional requirements such as, for example, management of large data sets, storing and implementing programs on the computer also used for process control or management of local network interconnection. Thus, some of these conventional operating systems have been reengineered in order to provide a reentrant and interruptible kernel and to lighten the task structure and communication. This has led to real-time Unix implementations. The market seems to be showing a trend towards real-time systems proposing a Posix standard interface (Portable Operating System Interface for Computer Environments; international standardization for Unix-like systems).

1.2 Basic Concepts for Real-Time Task Scheduling

1.2.1 Task description

Real-time task model

Real-time tasks are the basic executable entities that are scheduled; they may be periodic or aperiodic, and have soft or hard real-time constraints. A task model has been

defined with the main timing parameters. A task is defined by chronological parameters denoting delays and by chronometric parameters denoting times. The model includes primary and dynamic parameters. Primary parameters are (Figure 1.8):

- r , task release time, i.e. the triggering time of the task execution request.
- C , task worst-case computation time, when the processor is fully allocated to it.
- D , task relative deadline, i.e. the maximum acceptable delay for its processing.
- T , task period (valid only for periodic tasks).
- when the task has hard real-time constraints, the relative deadline allows computation of the absolute deadline $d = r + D$. Transgression of the absolute deadline causes a timing fault.

The parameter T is absent for an aperiodic task. A periodic task is modelled by the four previous parameters. Each time a task is ready, it releases a periodic request. The successive release times (also called request times, arrival times or ready times) are request release times at $r_k = r_0 + kT$, where r_0 is the first release and r_k the $k + 1$ th release; the successive absolute deadlines are $d_k = r_k + D$. If $D = T$, the periodic task has a relative deadline equal to period. A task is well formed if $0 < C \leq D \leq T$.

The quality of scheduling depends on the exactness of these parameters, so their determination is an important aspect of real-time design. If the durations of operations like task switching, operating system calls, interrupt processing and scheduler execution cannot be neglected, the design analysis must estimate these durations and add them

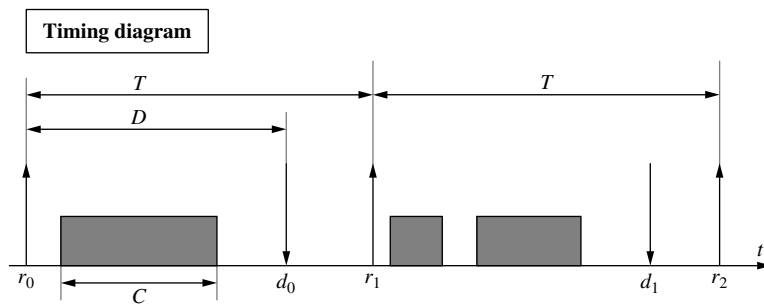
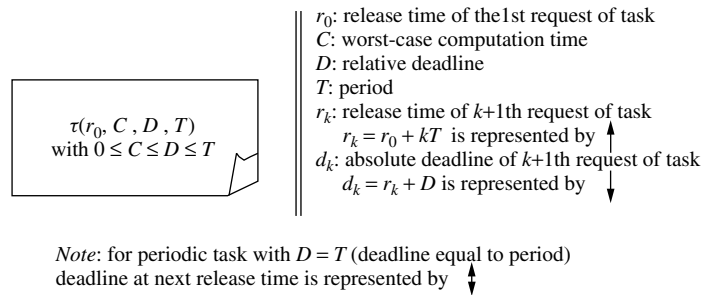


Figure 1.8 Task model

to the task computation times. That is why a deterministic behaviour is required for the kernel, which should guarantee maximum values for these operations.

Other parameters are derived:

- $u = C/T$ is the processor utilization factor of the task; we must have $u \leq 1$.
- $ch = C/D$ is the processor load factor; we must have $ch \leq 1$.

The following dynamic parameters help to follow the task execution:

- s is the start time of task execution.
- e is the finish time of task execution.
- $D(t) = d - t$ is the residual relative deadline at time t : $0 \leq D(t) \leq D$.
- $C(t)$ is the pending execution time at time t : $0 \leq C(t) \leq C$.
- $L = D - C$ is the nominal laxity of the task (it is also called slack time) and it denotes the maximum lag for its start time s when it has sole use of the processor.
- $L(t) = D(t) - C(t)$ is the residual nominal laxity of the task at time t and it denotes the maximum lag for resuming its execution when it has sole use of the processor; we also have $L(t) = D + r - t - C(t)$.
- $TR = e - r$ is the task response time; we have $C \leq TR \leq D$ when there is no time fault.
- $CH(t) = C(t)/D(t)$ is the residual load; $0 \leq CH(t) \leq C/T$ (by definition, if $e = d$, $CH(e) = 0$).

Figure 1.9 shows the evolution of $L(t)$ and $D(t)$ according to time.

Periodic tasks are triggered at successive request release times and return to the passive state once the request is completed. Aperiodic tasks may have the same behaviour if they are triggered more than once; sometimes they are created at release time.

Once created, a task evolves between two states: passive and triggered. Processor and resource sharing introduces several task states (Figure 1.10):

- *elected*: a processor is allocated to the task; $C(t)$ and $D(t)$ decrease, $L(t)$ does not decrease.
- *blocked*: the task waits for a resource, a message or a synchronization signal; $L(t)$ and $D(t)$ decrease.
- *ready*: the task waits for election: in this case, $L(t)$ and $D(t)$ decrease.
- *passive*: the task has no current request.
- *non-existing*: the task is not created.

Other task characteristics

In addition to timing parameters of the task model, tasks are described by other features.

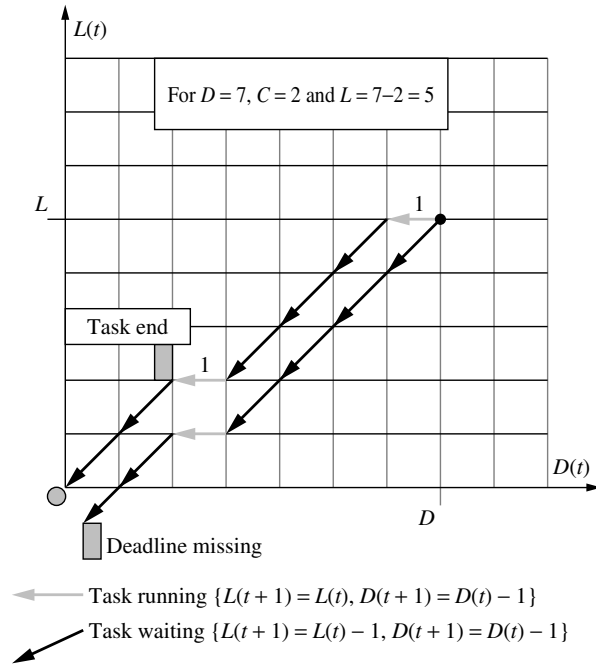
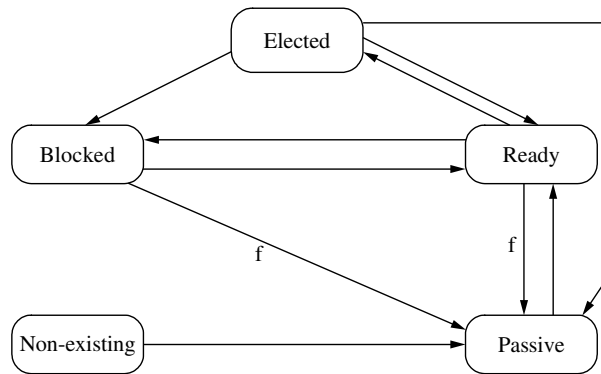


Figure 1.9 Dynamic parameter evolution



f: evolution when a request is aborted after a timing fault (missing deadline)

Figure 1.10 Task states

Preemptive or non-preemptive task Some tasks, once elected, should not be stopped before the end of their execution; they are called *non-preemptive* tasks. For example, a non-preemptive task is necessary to handle direct memory access (DMA) input–output or to run in interrupt mode. Non-preemptive tasks are often called *immediate* tasks. On the contrary, when an elected task may be stopped and reset to the ready state in order to allocate the processor to another task, it is called a *preemptive* task.

Dependency of tasks Tasks may interact according to a partial order that is fixed or caused by a message transmission or by explicit synchronization. This creates precedence relationships among tasks. Precedence relationships are known before execution, i.e. they are static, and can be represented by a static precedence graph (Figure 1.11). Tasks may share other resources than the processor and some resources may be exclusive or critical, i.e. they must be used in mutual exclusion. The sequence of instructions that a task has to execute in mutual exclusion is called a critical section. Thus, only one task is allowed to run its critical section for a given resource (Figure 1.12).

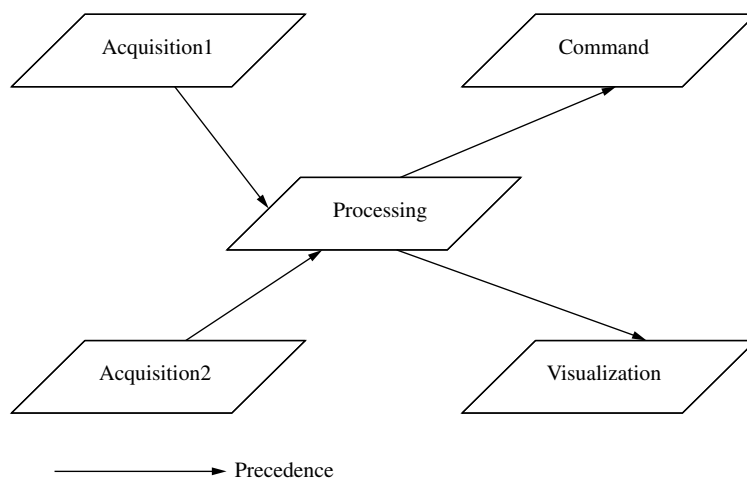


Figure 1.11 A precedence graph with five tasks

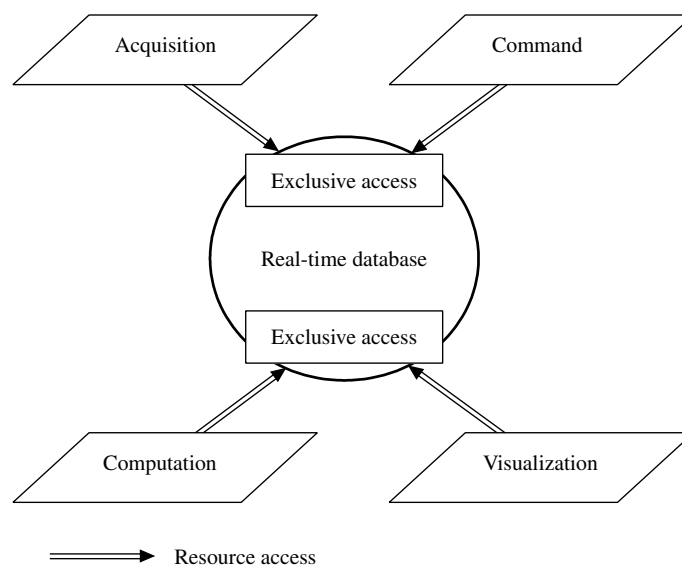


Figure 1.12 Example of a critical resource shared by four tasks

Resource sharing induces a dynamic relationship when the resource use order depends on the task election order. The relationships can be represented by an allocation graph. When the tasks have static and dynamic dependencies which may serialize them, the notion of global response time, or end-to-end delay, is used. This is the time elapsed between the release time of the task reactive to the process stimulus and the finish time of the last task that commands the actuators in answer to the stimulus. Tasks are independent when they have no precedence relationships and do not share critical resources.

Maximum jitter Sometimes, periodic requests must have regular start times or response times. This is the case of periodic data sampling, a proportional integral derivative (PID) control loop or continuous emission of audio and video streams. The difference between the start times of two consecutive requests, s_i and s_{i+1} , is the start time jitter. A maximum jitter, or absolute jitter, is defined as $|s_{i+1} - (s_i + T)| \leq Gmax$. The maximum response time jitter is similarly defined.

Urgency The task deadline allows the specification of the urgency of data provided by this task. Two tasks with equal urgency are given the same deadline.

Importance (criticality) When some tasks of a set are able to overcome timing faults and avoid their propagation, the control system may suppress the execution of some tasks. The latter must be aware of which tasks to suppress first or, on the other hand, which tasks are essential for the application and should not be suppressed. An importance parameter is introduced to specify the criticality of a task. Two tasks with equal urgency (thus having the same deadline) can be distinguished by different importance values.

External priority The designer may fix a constant priority, called external priority. In this simplified form, all scheduling decisions are taken by an off-line scheduler or by *a priori* rules (for example, the clock management task or the backup task in the event of power failure must run immediately).

1.2.2 Scheduling: definitions, algorithms and properties

In a real-time system, tasks have timing constraints and their execution is bounded to a maximum delay that has to be respected imperatively as often as possible. The objective of scheduling is to allow tasks to fulfil these timing constraints when the application runs in a nominal mode. A schedule must be predictable, i.e. it must be *a priori* proven that all the timing constraints are met in a nominal mode. When malfunctions occur in the controlled process, some alarm tasks may be triggered or some execution times may increase, overloading the application and giving rise to timing faults. In an overload situation, the objective of scheduling is to allow some tolerance, i.e. to allow the execution of the tasks that keep the process safe, although at a minimal level of service.

Task sets

A real-time application is specified by means of a set of tasks.

Progressive or simultaneous triggering Application tasks are simultaneously triggered when they have the same first release time, otherwise they are progressively triggered. Tasks simultaneously triggered are also called *in phase* tasks.

Processor utilization factor The processor utilization factor of a set of n periodic tasks is:

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \quad (1.1)$$

Processor load factor The processor load factor of a set of n periodic tasks is:

$$CH = \sum_{i=1}^n \frac{C_i}{D_i} \quad (1.2)$$

Processor laxity Because of deadlines, neither the utilization factor nor the load factor is sufficient to evaluate an overload effect on timing constraints. We introduce $LP(t)$, the processor laxity at t , as the maximal time the processor may remain idle after t without causing a task to miss its deadline. $LP(t)$ varies as a function of t . For all t , we must have $LP(t) \geq 0$. To compute the laxity, the assignment sequence of tasks to the processor must be known, and then the conditional laxity $LC_i(t)$ of each task i must be computed:

$$LC_i(t) = D_i - \sum C_j(t) \quad (1.3)$$

where the sum in j computes the pending execution time of all the tasks (including task i) that are triggered at t and that precede task i in the assignment sequence. The laxity $LP(t)$ is the smallest value of conditional laxity $LC_i(t)$.

Processor idle time The set of time intervals where the processor laxity is strictly positive, i.e. the set of spare intervals, is named the processor idle time. It is a function of the set of tasks and of their schedule.

Task scheduling definitions

Scheduling a task set consists of planning the execution of task requests in order to meet the timing constraints:

- of all tasks when the system runs in the nominal mode;
- of at least the most important tasks (i.e. the tasks that are necessary to keep the controlled process secure), in an abnormal mode.

An abnormal mode may be caused by hardware faults or other unexpected events. In some applications, additional performance criteria are sought, such as minimizing the response time, reducing the jitter, balancing the processor load among several sites, limiting the communication cost, or minimizing the number of late tasks and messages or their cumulative lag.

The scheduling algorithm assigns tasks to the processor and provides an ordered list of tasks, called the planning sequence or the schedule.

Scheduling algorithms taxonomy

On-line or off-line scheduling Off-line scheduling builds a complete planning sequence with all task set parameters. The schedule is known before task execution and can be implemented efficiently. However, this static approach is very rigid; it assumes that all parameters, including release times, are fixed and it cannot adapt to environmental changes.

On-line scheduling allows choosing at any time the next task to be elected and it has knowledge of the parameters of the currently triggered tasks. When a new event occurs the elected task may be changed without necessarily knowing in advance the time of this event occurrence. This dynamic approach provides less precise statements than the static one since it uses less information, and it has higher implementation overhead. However, it manages the unpredictable arrival of tasks and allows progressive creation of the planning sequence. Thus, on-line scheduling is used to cope with aperiodic tasks and abnormal overloading.

Preemptive or non-preemptive scheduling In preemptive scheduling, an elected task may be preempted and the processor allocated to a more urgent task or one with higher priority; the preempted task is moved to the ready state, awaiting later election on some processor. Preemptive scheduling is usable only with preemptive tasks. Non-preemptive scheduling does not stop task execution. One of the drawbacks of non-preemptive scheduling is that it may result in timing faults that a preemptive algorithm can easily avoid. In uniprocessor architecture, critical resource sharing is easier with non-preemptive scheduling since it does not require any concurrent access mechanism for mutual exclusion and task queuing. However, this simplification is not valid in multiprocessor architecture.

Best effort and timing fault intolerance With soft timing constraints, the scheduling uses a best effort strategy and tries to do its best with the available processors. The application may tolerate timing faults. With hard time constraints, the deadlines must be guaranteed and timing faults are not tolerated.

Centralized or distributed scheduling Scheduling is centralized when it is implemented on a centralized architecture or on a privileged site that records the parameters of all the tasks of a distributed architecture. Scheduling is distributed when each site defines a local scheduling after possibly some cooperation between sites leading to a global scheduling strategy. In this context some tasks may be assigned to a site and migrate later.

Scheduling properties

Feasible schedule A scheduling algorithm results in a schedule for a task set. This schedule is feasible if all the tasks meet their timing constraints.

Schedulable task set A task set is schedulable when a scheduling algorithm is able to provide a feasible schedule.

Optimal scheduling algorithm An algorithm is optimal if it is able to produce a feasible schedule for any schedulable task set.

Schedulability test A schedulability test allows checking of whether a periodic task set that is submitted to a given scheduling algorithm might result in a feasible schedule.

Acceptance test On-line scheduling creates and modifies the schedule dynamically as new task requests are triggered or when a deadline is missed. A new request may be accepted if there exists at least a schedule which allows all previously accepted task requests as well as this new candidate to meet their deadlines. The required condition is called an acceptance test. This is often called a guarantee routine since if the tasks respect their worst-case computation time (to which may be added the time waiting for critical resources), the absence of timing faults is guaranteed. In distributed scheduling, the rejection of a request by a site after a negative acceptance test may lead the task to migrate.

Scheduling period (or major cycle or hyper period) The validation of a periodic and aperiodic task set leads to the timing analysis of the execution of this task set. When periodic tasks last indefinitely, the analysis must go through infinity. In fact, the task set behaviour is periodic and it is sufficient to analyse only a validation period or pseudo-period, called the scheduling period, the schedule length or the hyper period (Grolleau and Choquet-Geniet, 2000; Leung and Merrill, 1980). The scheduling period of a task set starts at the earliest release time, i.e. at time $t = \text{Min}\{r_{i,0}\}$, considering all tasks of the set. It ends at a time which is a function of the least common multiple (LCM) of periods (T_i), the first release times of periodic tasks and the deadlines of aperiodic tasks:

$$\text{Max}\{r_{i,0}, (r_{j,0} + D_j)\} + 2 \cdot \text{LCM}(T_i) \quad (1.4)$$

where i varies in the set of periodic task indexes, and j in the set of aperiodic task indexes.

Implementation of schedulers

Scheduling implementation relies on conventional data structures.

Election table When the schedule is fixed before application start, as in static off-line scheduling, this definitive schedule may be stored in a table and used by the scheduler to decide which task to elect next.

Priority queuing list On-line scheduling creates dynamically a planning sequence, the first element of which is the elected task (in a n -processor architecture, the n first elements are concerned). This sequence is an ordered list; the ordering relationship is represented by keys; searching and suppression point out the minimal key element; a new element is inserted in the list according to its key ordering. This structure is usually called a heap sorted list or a priority ordered list (Weiss, 1994).

Constant or varying priority The element key, called priority when elements are tasks, is a timing parameter or a mix of parameters of the task model. It remains constant when the parameter is not variable, such as computation time, relative deadline, period or external priority. It is variable when the parameter changes during task execution, such as pending computation time, residual laxity, or when it is modified from one request to another, such as the release time or absolute deadline. The priority value or

sorting key may be the value of the parameter used or, if the range of values is too large, a one-to-one function from this parameter to a subset of integers. This subset is usually called the priority set. The size of this priority set may be fixed *a priori* by hardware architecture or by the operating system kernel. Coding the priority with a fixed bit-size and using special machine instruction allows the priority list management to be made faster.

Two-level scheduling When scheduling gets complex, it is split into two parts. One elaborates policy (high-level or long-term decisions, facing overload with task suppression, giving preference to some tasks for a while in hierarchical scheduling). The other executes the low-level mechanisms (election of a task in the subset prepared by the high-level scheduler, short-term choices which reorder this subset). A particular case is distributed scheduling, which separates the local scheduling that copes with the tasks allocated to a site and the global scheduling that assigns tasks to sites and migrates them. The order between local and global is another choice whose cost must be appraised: should tasks be settled *a priori* in a site and then migrate if the site becomes overloaded, or should all sites be interrogated about their reception capacity before allocating a triggered task?

1.2.3 Scheduling in classical operating systems

Scheduling objectives in a classical operating system

In a multitasking system, scheduling has two main functions:

- maximizing processor usage, i.e. the ratio between active time and idle time. Theoretically, this ratio may vary from 0% to 100%; in practice, the observed rate varies between 40% and 95%.
- minimizing response time of tasks, i.e. the time between task submission time and the end of execution. At best, response time may be equal to execution time, when a task is elected immediately and executed without preemption.

The success of both functions may be directly appraised by computing the processing ratio and the mean response time, but other evaluation criteria are also used. Some of them are given below:

- evaluating the task waiting time, i.e. the time spent in the ready state;
- evaluating the processor throughput, i.e. the average number of completed tasks during a time interval;
- computing the total execution time of a given set of tasks;
- computing the average response time of a given set of tasks.

Main policies

The scheduling policy decides which ready task is elected. Let us describe below some of the principal policies frequently used in classical operating systems.

First-come-first-served scheduling policy This policy serves the oldest request, without preemption; the processor allocation order is the task arrival order. Tasks with short computation time may be penalized when a task with a long computation time precedes them.

Shortest first scheduling policy This policy aims to correct the drawback mentioned above. The processor is allocated to the shortest computation time task, without preemption. This algorithm is the non-preemptive scheduling algorithm that minimizes the mean response time. It penalizes long computation tasks. It requires estimating the computation time of a task, which is usually unknown. A preemptive version of this policy is called ‘pending computation time first’: the elected task gives back the processor when a task with a shorter pending time becomes ready.

Round-robin scheduling policy A time slice, which may be fixed, for example between 10 ms and 100 ms, is given as a quantum of processor allocation. The processor is allocated in turn to each ready task for a period no longer than the quantum. If the task ends its computation before the end of the quantum, it releases the processor and the next ready task is elected. If the task has not completed its computation before the quantum end, it is preempted and it becomes the last of the ready task set (Figure 1.13). A round-robin policy is commonly used in time-sharing systems. Its performance heavily relies on the quantum size. A large quantum increases response times, while too small a quantum increases task commutations and then their cost may no longer be neglected.

Constant priority scheduling policy A constant priority value is assigned to each task and at any time the elected task is always the highest priority ready task (Figure 1.14). This algorithm can be used with or without preemption. The drawback of this policy is that low-priority tasks may starve forever. A solution is to ‘age’ the priority of waiting ready tasks, i.e. to increase the priority as a function of waiting time. Thus the task priority becomes variable.

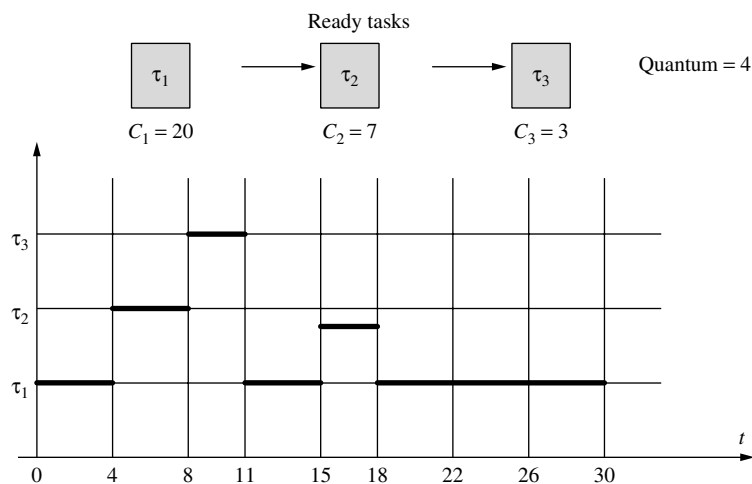


Figure 1.13 Example of Round-Robin scheduling

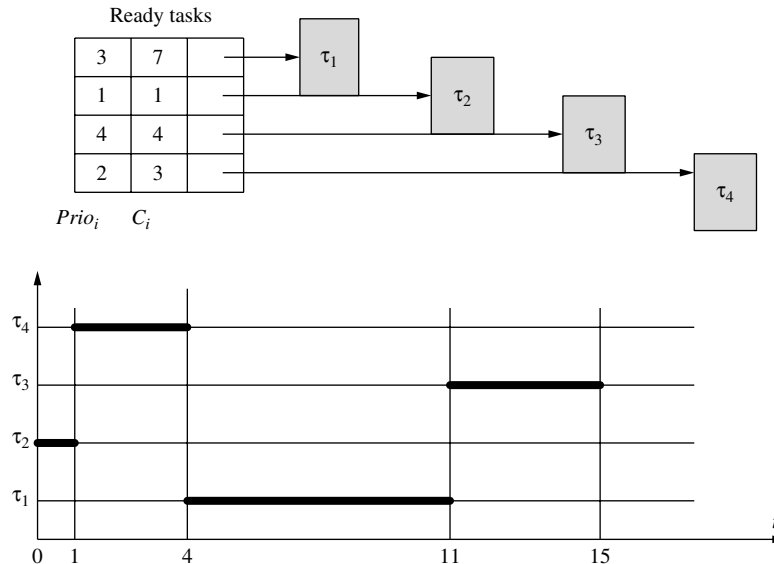


Figure 1.14 Example of priority scheduling (the lower the priority index, the higher is the task priority)

Multilevel priority scheduling policy In the policies above, ready tasks share a single waiting list. We choose now to define several ready task lists, each corresponding to a priority level; this may lead to n different priority lists varying from 0 to $n - 1$. In a given list, all tasks have the same priority and are first-come-first-served without preemption or in a round-robin fashion. The quantum value may be different from one priority list to another. The scheduler serves first all the tasks in list 0, then all the tasks in list 1 as long as list 0 remains empty, and so on. Two variants allow different evolution of the task priorities:

- Task priorities remain constant all the time. At the end of the quantum, a task that is still ready is reentered in the waiting list corresponding to its priority value.
- Task priorities evolve dynamically according to the service time given to the task. Thus a task elected from list x , and which is still ready at the end of its quantum, will not reenter list x , but list $x + 1$ of lower priority, and so on. This policy tries to minimize starvation risks for low-priority tasks by progressively lowering the priority of high-priority tasks (Figure 1.15).

Note: none of the preceding policies fulfils the two objectives of real-time scheduling, especially because none of them integrates the notion of task urgency, which is represented by the relative deadline in the model of real-time tasks.

1.2.4 Illustrating real-time scheduling

Let us introduce the problem of real-time scheduling by a tale inspired by La Fontaine, the famous French fabulist who lived in the 17th century. The problem is to control

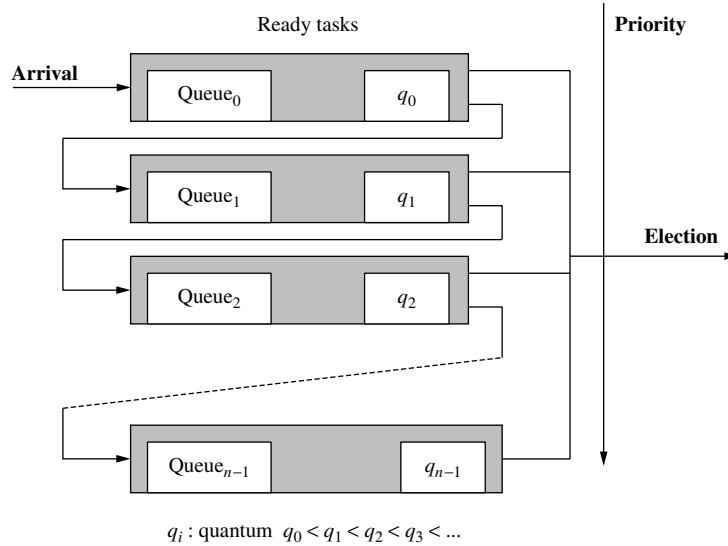


Figure 1.15 Example of multilevel priority scheduling

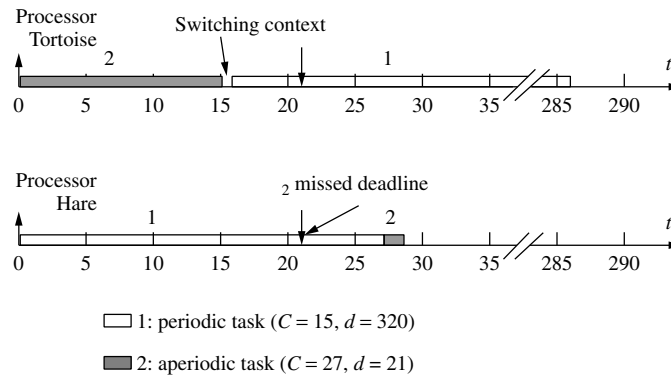


Figure 1.16 Execution sequences with two different scheduling algorithms and two different processors (the Hare and the Tortoise)

a real-time application with two tasks τ_1 and τ_2 . The periodic task τ_1 controls the engine of a mobile vehicle. Its period as well as its relative deadline is 320 seconds. The sporadic task τ_2 has to react to steering commands before a relative deadline of 21 seconds. Two systems are proposed by suppliers.

The Tortoise system has a processor whose speed is 1 Mips, a task switching overhead of 1 second and an earliest deadline scheduler. The periodic task computation is 270 seconds; the sporadic task requires 15 seconds. The Hare system has the advantage of being very efficient and of withdrawing resource-sharing contention. It has a processor whose speed is 10 Mips, a task switching overhead of (almost) 0 and a first-in-first-out non-preemptive scheduler. So, with this processor, the periodic task τ_1 computation is 27 seconds; the sporadic task τ_2 requires 1.5 seconds.

An acceptance trial was made by one of our students as follows. Just after the periodic task starts running, the task is triggered. The Tortoise respects both deadlines while the Hare generates a timing fault for the steering command (Figure 1.16). The explanation is a trivial exercise for the reader of this book and is an illustration that scheduling helps to satisfy timing constraints better than system efficiency.

The first verse of La Fontaine's tale, named the Hare and the Tortoise, is 'It is no use running; it is better to leave on time' (La Fontaine, *Le lièvre et la tortue*, Fables VI, 10, Paris, 17th century).

