1

A GENERAL VIEW

1.1 INTRODUCTION

Real-time systems are computing systems that must react within precise time constraints to events in the environment. As a consequence, the correct behavior of these systems depends not only on the value of the computation but also on the time at which the results are produced [SR88]. A reaction that occurs too late could be useless or even dangerous. Today, real-time computing plays a crucial role in our society, since an increasing number of complex systems rely, in part or completely, on computer control. Examples of applications that require real-time computing include

- Chemical and nuclear plant control,
- Control of complex production processes,
- Railway switching systems,
- Automotive applications,
- Flight control systems,
- Environmental acquisition and monitoring,
- Telecommunication systems,
- Industrial automation,
- Robotics,
- Military systems,

- Space missions,
- Multimedia systems, and
- Virtual reality.

Despite this large application domain, many researchers, developers, and technical managers have serious misconceptions about real-time computing [Sta88], and most of today's real-time control systems are still designed using ad hoc techniques and heuristic approaches. Very often, control applications with stringent time constraints are implemented by writing large portions of code in assembly language, programming timers, writing low-level drivers for device handling, and manipulating task and interrupt priorities. Although the code produced by these techniques can be optimized to run very efficiently, this approach has the following disadvantages:

- **Tedious programming**. The implementation of large and complex applications in assembly language is much more difficult and time consuming than high-level programming. Moreover, the efficiency of the code strongly depends on the programmer's ability.
- **Difficult code understanding**. Except for the programmers who develop the application, very few people can fully understand the functionality of the software produced. Clever hand-coding introduces additional complexity and makes a program more difficult to comprehend.
- **Difficult software maintainability**. As the the complexity of the program increases, the modification of large assembly programs becomes difficult even for the original programmer.
- **Difficult verification of time constraints**. Without the support of specific tools and methodologies for code and schedulability analysis, the verification of time constraints becomes practically impossible.

The major consequence of this approach is that the control software produced by empirical techniques can be highly unpredictable. If all critical time constraints cannot be verified a priori and the operating system does not include specific features for handling real-time tasks, the system could apparently work well for a period of time, but it could collapse in certain rare, but possible, situations. The consequences of a failure can sometimes be catastrophic and may injure people or cause serious damage to the environment.

A high percentage of accidents that occur in nuclear power plants, in space missions, or in defensive systems are often caused by software bugs in the control system. In some cases, these accidents have caused huge economic losses or even catastrophic consequences including the loss of human lives.

As an example, the first flight of the space shuttle was delayed, at considerable cost, because of a timing bug that arose from a transient CPU overload during system initialization on one of the redundant processors dedicated to the control of the aircraft [Sta88]. Although the shuttle control system was intensively tested, the timing error was never discovered before. Later, by analyzing the code of the processes, it has been found that there was only a 1 in 67 probability (about 1.5 percent) that a transient overload during initialization could push the redundant processor out of synchronization.

Another software bug was discovered on the real-time control system of the Patriot missiles, used to protect Saudi Arabia during the Gulf War.[1] When a Patriot radar sights a flying object, the on-board computer calculates its trajectory and, to ensure that no missiles are launched in vain, it performs a verification. If the flying object passes through a specific location, computed based on the predicted trajectory, then the Patriot is launched against the target, otherwise the phenomenon is classified as a false alarm.

On February 25, 1991, the radar sighted a Scud missile directed at Saudi Arabia, and the on-board computer predicted its trajectory, performed the verification, but classified the event as a false alarm. A few minutes later, the Scud fell on the city of Dhahran, causing victims and enormous economic damage. Later on, it was discovered that, because of a subtle software bug, the real-time clock of the on-board computer was accumulating a delay of about 57 microseconds per minute. The day of the accident, the computer had been working for about 100 hours (an exceptional condition that was never experienced before), thus accumulating a total delay of 343 milliseconds. This delay caused a prediction error in the verification phase of 687 meters! The bug was corrected on February 26, the day after the accident.

The examples of failures described above show that software testing, although important, does not represent a solution for achieving predictability in real-time systems. This is mainly due to the fact that, in real-time control applications, the program flow depends on input sensory data and environmental conditions, which cannot be fully replicated during the testing phase. As a consequence, the testing phase can provide only a *partial* verification of the software behavior, relative to the particular subset of data provided as input.

[1] *L'Espresso*, Vol. XXXVIII, No. 14, 5 April 1992, p. 167.

A more robust guarantee of the performance of a real-time system under all possible operating conditions can be achieved only by using more sophisticated design methodologies, combined with a static analysis of the source code and specific operating systems mechanisms, purposely designed to support computation under time constraints. Moreover, in critical applications, the control system must be capable of handling all anticipated scenarios, including peak load situations, and its design must be driven by pessimistic assumptions on the events generated by the environment.

In 1949, an aeronautical engineer of the U.S. Air Force, Captain Ed Murphy, observed the evolution of his experiments and said: "If something can go wrong, it will go wrong." Several years later, Captain Ed Murphy became famous around the world, not for his work in avionics but for his phrase, simple but ineluctable, today known as *Murphy's Law* [Blo77, Blo80, Blo88]. Since that time, many other laws on existential pessimism have been formulated to describe unfortunate events in a humorous fashion. Due to the relevance that pessimistic assumptions have on the design of real-time systems, Table 1.1 lists the most significant laws on the topic, which a software engineer should always keep in mind.

1.2 WHAT DOES REAL TIME MEAN?

1.2.1 THE CONCEPT OF TIME

The main characteristic that distinguishes real-time computing from other types of computation is time.

The word *time* means that the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced.

The word *real* indicates that the reaction of the systems to external events must occur *during* their evolution. As a consequence, the system time (internal time) must be measured using the same time scale used for measuring the time in the controlled environment (external time).

Although the term *real time* is frequently used in many application fields, it is subject to different interpretations, not always correct. Often, people say that a control system operates in real time if it is able to *quickly* react to external events. According to this interpretation, a system is considered to be real-time if it is fast. The term *fast*, however, has a relative meaning and does not capture the main properties that characterize these types of systems.

Murphy's General Law

If something can go wrong, it will go wrong.

Murphy's Constant

Damage to an object is proportional to its value.

Naeser's Law

One can make something bomb-proof, not jinx-proof.

Troutman Postulates

1. *Any software bug will tend to maximize the damage.*
2. *The worst software bug will be discovered six months after the field test.*

Green's Law

If a system is designed to be tolerant to a set of faults, there will always exist an idiot so skilled to cause a nontolerated fault.

Corollary

Dummies are always more skilled than measures taken to keep them from harm.

Johnson's First Law

If a system stops working, it will do it at the worst possible time.

Sodd's Second Law

Sooner or later, the worst possible combination of circumstances will happen.

Corollary

A system must always be designed to resist the worst possible combination of circumstances.

Table 1.1 Murphy's laws on real-time systems.

In nature, living beings act in real time in their habitat independently of their speed. For example, the reactions of a turtle to external stimuli coming from its natural habitat are as effective as those of a cat with respect to its habitat. In fact, although the turtle is much slower than a cat, in terms of absolute speed, the events that it has to deal with are proportional to the actions it can coordinate, and this is a necessary condition for any animal to survive within an environment.

On the contrary, if the environment in which a biological system lives is modified by introducing events that evolve more rapidly than it can handle, its actions will no longer be as effective, and the survival of the animal is compromised. Thus, a quick fly can still be caught by a fly-swatter, a mouse can be captured by a trap, or a cat can be run down by a speeding car. In these examples, the fly-swatter, the trap, and the car represent unusual and anomalous events for the animals, out of their range of capabilities, which can seriously jeopardize their survival. The cartoons in Figure 1.1 schematically illustrate the concept expressed above.

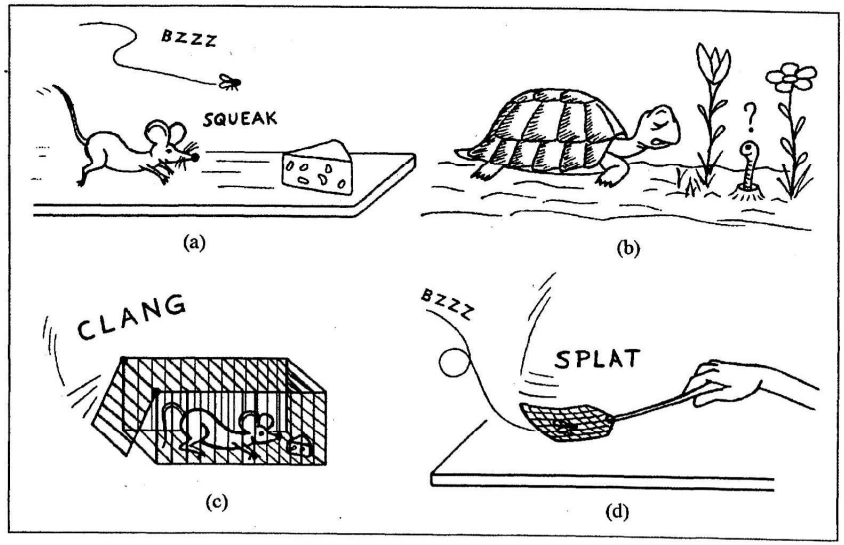

Figure 1.1 Both the mouse (a) and the turtle (b) behave in real time with respect to their natural habitat. Nevertheless, the survival of fast animals such as a mouse or a fly can be jeopardized by events (c and d) quicker than their reactive capabilities.

The previous examples show that the concept of time is not an intrinsic property of a control system, either natural or artificial, but that it is strictly related to the environment in which the system operates. It does not make sense to design a real-time computing system for flight control without considering the timing characteristics of the aircraft.

Hence, the environment is always an essential component of any real-time system. Figure 1.2 shows a block diagram of a typical real-time architecture for controlling a physical system.

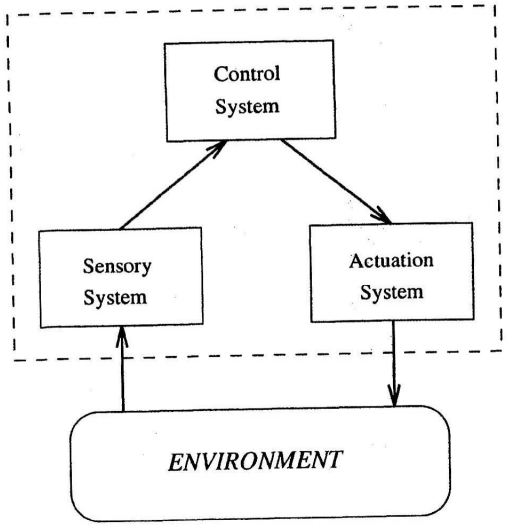

Figure 1.2 Block diagram of a generic real-time control system.

Some people erroneously believe that it is not worth investing in real-time research because advances in computer hardware will take care of any real-time requirements. Although advances in computer hardware technology will improve system throughput and will increase the computational speed in terms of millions of instructions per second (MIPS), this does not mean that the timing constraints of an application will be met automatically. In fact, whereas the objective of fast computing is to minimize the average response time of a given set of tasks, the objective of real-time computing is to meet the individual timing requirement of each task [Sta88].

However short the average response time can be, without a scientific methodology we will never be able to guarantee the individual timing requirements of each task in all possible circumstances. When several computational activities have different timing constraints, average performance has little significance for the correct behavior of the system. To better understand this issue, it is worth thinking about this little story[2]:

[2] From John Stankovic's notes.

There was a man who drowned crossing a stream with an average depth of six inches.

Hence, rather than being fast, a real-time computing system should be predictable. And one safe way to achieve predictability is to investigate and employ new methodologies at every stage of the development of an application, from design to testing.

At the process level, the main difference between a real-time and a non-real-time task is that a real-time task is characterized by a *deadline*, which is the maximum time within which it must complete its execution. In critical applications, a result produced after the deadline is not only late, but wrong! Depending on the consequences that may occur because of a missed deadline, real-time tasks are usually distinguished in two classes, *hard* and *soft*:

- A real-time task is said to be *hard* if missing its deadline may cause catastrophic consequences on the environment under control.
- A real-time task is said to be *soft* if meeting its deadline is desirable for performance reasons, but missing its deadline does not cause serious damage to the environment and does not jeopardize correct system behavior.

A real-time operating system that is able to handle hard real-time tasks is called a *hard real-time system*. Typically, real-world applications include hard and soft activities, and therefore a hard real-time system should be designed to handle both hard and soft tasks using two different strategies. In general, when an application consists of a hybrid task set, the objective of the operating system should be to guarantee the individual timing constraints of the hard tasks while minimizing the average response time of the soft activities.

Examples of hard activities that may be present in a control application include

- Sensory data acquisition,
- Detection of critical conditions,
- Actuator servoing,
- Low-level control of critical system components, and
- Planning sensory-motor actions that tightly interact with the environment.

Examples of soft activities include

- The command interpreter of the user interface,
- Handling input data from the keyboard,
- Displaying messages on the screen,
- Representation of system state variables,
- Graphical activities, and
- Saving report data.

1.2.2 LIMITS OF CURRENT REAL-TIME SYSTEMS

Most of the real-time computing systems used to support control applications are based on kernels [AL86, Rea86, HHPD87, SBG86], which are modified versions of time-sharing operating systems. As a consequence, they have the same basic features found in timesharing systems, which are not suited to support real-time activities. The main characteristics of such real-time systems include

- **Multitasking**. A support for concurrent programming is provided through a set of system calls for process management (such as *create*, *activate*, *terminate*, *delay*, *suspend*, and *resume*). Many of these primitives do not take time into account and, even worse, introduce unbounded delays on tasks' execution time that may cause hard tasks to miss their deadlines in an unpredictable way.
- **Priority-based scheduling**. Such a scheduling mechanism is quite flexible, since it allows the implementation of several strategies for process management just by changing the rule for assigning priorities to tasks. Nevertheless, when application tasks have explicit time requirements, mapping timing constraints into a set of priorities may not be simple, especially in dynamic environments. The major problem comes from the fact that these kernels have a limited number of priority levels (typically 128 or 256), whereas task deadlines can vary in a much wider range. Moreover, in dynamic environments, the arrival of a new task may require remapping the entire set of priorities.
- **Ability to quickly respond to external interrupts**. This feature is usually obtained by setting interrupt priorities higher than process priorities and by reducing the portions of code executed with interrupts disabled. Note that, although this

approach increases the reactivity of the system to external events, it introduces unbounded delays on processes' execution. In fact, an application process will be always interrupted by a driver, even though it is more important than the device that is going to be served. Moreover, in the general case, the number of interrupts that a process can experience during its execution cannot be bounded in advance, since it depends on the particular environmental conditions.

- **Basic mechanisms for process communication and synchronization**. Binary semaphores are typically used to synchronize tasks and achieve mutual exclusion on shared resources. However, if no access protocols are used to enter critical sections, classical semaphores can cause a number of undesirable phenomena, such as priority inversion, chained blocking, and deadlock, which again introduce unbounded delays on real-time activities.

- **Small kernel and fast context switch**. This feature reduces system overhead, thus improving the average response time of the task set. However, a small average response time on the task set does not provide any guarantee on the individual task deadlines. On the other hand, a small kernel implies limited functionality, which affects the predictability of the system.

- **Support of a real-time clock as an internal time reference**. This is an essential feature for any real-time kernel that handles time-critical activities that interact with the environment. Nevertheless, in most commercial kernels this is the only mechanism for time management. In many cases, there are no primitives for explicitly specifying timing constraints (such as deadlines) on tasks, and there is no mechanism for automatic activation of periodic tasks.

From the above features, it is easy to see that those types of real-time kernels are developed under the same basic assumptions made in timesharing systems, where tasks are considered as unknown activities activated at random instants. Except for the priority, no other parameters are provided to the system. As a consequence, computation times, timing constraints, shared resources, or possible precedence relations among tasks are not considered in the scheduling algorithm, and hence no guarantee can be performed.

The only objectives that can be pursued with these systems is a quick reaction to external events and a "small" average response time for the other tasks. Although this may be acceptable for some soft applications, the lack of any form of guarantee precludes the use of these systems for those control applications that require stringent timing constraints that must be met to ensure safe behavior of the system.

1.2.3 DESIRABLE FEATURES OF REAL-TIME SYSTEMS

Complex control applications that require hard timing constraints on tasks' execution need to be supported by highly predictable operating systems. Predictability can be achieved only by introducing radical changes in the basic design paradigms found in classical timesharing systems.

For example, in any real-time control system, the code of each task is known a priori and hence can be analyzed to determine its characteristics in terms of computation time, resources, and precedence relations with other tasks. Therefore, there is no need to consider a task as an unknown processing entity; rather, its parameters can be used by the operating system to verify its schedulability within the specified timing requirements. Moreover, all hard tasks should be handled by the scheduler to meet their individual deadlines, not to reduce their average response time.

In addition, in any typical real-time application, the various control activities can be seen as members of a team acting together to accomplish one common goal, which can be the control of a nuclear power plant or an aircraft. This means that tasks are not all independent and it is not strictly necessary to support independent address spaces.

In summary, there are some very important basic properties that real-time systems must have to support critical applications. They include

- **Timeliness**. Results have to be correct not only in their value but also in the time domain. As a consequence, the operating system must provide specific kernel mechanisms for time management and for handling tasks with explicit time constraints and different criticality.

- **Design for peak load**. Real-time systems must not collapse when they are subject to peak-load conditions, so they must be designed to manage all anticipated scenarios.

- **Predictability**. To guarantee a minimum level of performance, the system must be able to predict the consequences of any scheduling decision. If some task cannot be guaranteed within its time constraints, the system must notify this fact in advance, so that alternative actions can be planned in time to cope with the event.

- **Fault tolerance**. Single hardware and software failures should not cause the system to crash. Therefore, critical components of the real-time system have to be designed to be fault tolerant.

- **Maintainability**. The architecture of a real-time system should be designed according to a modular structure to ensure that possible system modifications are easy to perform.

1.3 ACHIEVING PREDICTABILITY

One of the most important properties that a hard real-time system should have is predictability [SR90]. That is, based on the kernel features and on the information associated with each task, the system should be able to predict the evolution of the tasks and guarantee in advance that all critical timing constraints will be met. The reliability of the guarantee, however, depends on a range of factors, which involve the architectural features of the hardware and the mechanisms and policies adopted in the kernel, up to the programming language used to implement the application.

The first component that affects the predictability of the scheduling is the processor itself. The internal characteristics of the processor, such as instruction prefetch, pipelining, cache memory, and direct memory access (DMA) mechanisms, are the first cause of nondeterminism. In fact, although these features improve the average performance of the processor, they introduce nondeterministic factors that prevent a precise analysis of the worst-case execution times. Other important components that influence the execution of the task set are the internal characteristics of the real-time kernel, such as the scheduling algorithm, the synchronization mechanism, the types of semaphores, the memory management policy, the communication semantics, and the interrupt handling mechanism.

In the rest of this chapter, the main sources of nondeterminism are considered in more detail, from the physical level up to the programming level.

1.3.1 DMA

Direct memory access (DMA) is a technique used by many peripheral devices to transfer data between the device and the main memory. The purpose of DMA is to relieve the central processing unit (CPU) of the task of controlling the input/output (I/O) transfer. Since both the CPU and the I/O device share the same bus, the CPU has to be blocked when the DMA device is performing a data transfer. Several different transfer methods exist.

Problemas con el robo de ciclo.

One of the most common methods is called *cycle stealing*, according to which the DMA device steals a CPU memory cycle in order to execute a data transfer. During the DMA operation, the I/O transfer and the CPU program execution run in parallel. However, if the CPU and the DMA device require a memory cycle at the same time, the bus is assigned to the DMA device and the CPU waits until the DMA cycle is completed. Using the cycle stealing method, there is no way of predicting how many times the CPU will have to wait for DMA during the execution of a task; hence the response time of a task cannot be precisely determined.

A possible solution to this problem is to adopt a different technique, which requires the DMA device to use the memory *time-slice method* [SR88]. According to this method, each memory cycle is split into two adjacent time slots: one reserved for the CPU and the other for the DMA device. This solution is more expensive than cycle stealing but more predictable. In fact, since the CPU and DMA device do not conflict, the response time of the tasks do not increase due to DMA operations and hence can be predicted with higher accuracy.

2 de memoria. el ciclo está dividido en 2 slots adyacentes, 1 para la CPU.

1.3.2 CACHE

The cache is a fast memory that is inserted as a buffer between the CPU and the random access memory (RAM) to speed up processes' execution. It is physically located after the memory management unit (MMU) and is not visible at the software programming level. Once the physical address of a memory location is determined, the hardware checks whether the requested information is stored in the cache: if it is, data are read from the cache; otherwise the information is taken from the RAM, and the content of the accessed location is copied into the cache along with a set of adjacent locations. In this way, if the next memory access is done to one of these locations, the requested data can be read from the cache, without having to access the memory.

This buffering technique is motivated by the fact that statistically the most frequent accesses to the main memory are limited to a small address space, a phenomenon called *program locality*. For example, it has been observed that with a 1 Mb memory and a 8 Kbyte cache, the data requested from a program are found in the cache 80 percent of the time (*hit ratio*).

The need for having a fast cache appeared when memory was much slower. Today, however, since memory has an access time almost comparable to that of the cache, the main motivation for having a cache is not only to speed up process execution but also to reduce conflicts with other devices. In any case, the cache is considered as a processor attribute that speeds up the activities of a computer.

In real-time systems, the cache introduces some degree of nondeterminism. In fact, although statistically the requested data are found in the cache 80 percent of the time, it is also true that in the other 20 percent of the cases the performance degrades. This happens because, when data is not found in the cache (cache fault or miss), the access time to memory is longer, due to the additional data transfer from RAM to cache. Furthermore, when performing write operations into memory, the use of the cache is even more expensive in terms of access time, because any modification made on the cache must be copied to the memory in order to maintain data consistency. Statistical observations show that 90 percent of the memory accesses are for read operations, whereas only 10 percent are for writes.

Statistical observations, however, can provide only an estimation of the average behavior of an application but cannot be used for deriving worst-case bounds. To perform worst-case analysis, in fact, we should assume a cache fault for each memory access. The consequence of this is that, to obtain a higher degree of predictability at the low level, it would be more efficient to have processors without cache or with the cache disabled. In other approaches, the influence of the cache on the task execution time is taken into account by a multiplicative factor, which depends on an estimated percentage of cache faults. A more precise estimation of the cache behavior can be achieved by analyzing the code of the tasks and estimating the execution times by using a mathematical model of the cache.

1.3.3 INTERRUPTS

Interrupts generated by I/O peripheral devices represent a big problem for the predictability of a real-time system because, if not properly handled, they can introduce unbounded delays during process execution. In almost any operating system, the arrival of an interrupt signal causes the execution of a service routine (*driver*), dedicated to the management of its associated device. The advantage of this method is to encapsulate all hardware details of the device inside the driver, which acts as a server for the application tasks. For example, in order to get data from an I/O device, each task must enable the hardware to generate interrupts, wait for the interrupt, and read the data from a memory buffer shared with the driver, according to the following protocol:

<enable device interrupts>
<wait for interrupt>
<get the result>

In many operating systems, interrupts are served using a fixed priority scheme, according to which each driver is scheduled based on a static priority, higher than process priorities. This assignment rule is motivated by the fact that interrupt handling routines usually deal with I/O devices that have real-time constraints, whereas most application programs do not. In the context of real-time systems, however, this assumption is certainly not valid, because a control process could be more urgent than an interrupt handling routine. Since, in general, it is very difficult to bound a priori the number of interrupts that a task may experience, the delay introduced by the interrupt mechanism on tasks' execution becomes unpredictable.

In order to reduce the interference of the drivers on the application tasks and still perform I/O operations with the external world, the peripheral devices must be handled in a different way. In the following, three possible techniques are illustrated.

APPROACH A

The most radical solution to eliminate interrupt interference is to disable all external interrupts, except the one from the timer (necessary for basic system operations). In this case, all peripheral devices must be handled by the application tasks, which have direct access to the registers of the interfacing boards. Since no interrupt is generated, data transfer takes place through polling.

The direct access to I/O devices allows great programming flexibility and eliminates the delays caused by the drivers' execution. As a result, the time needed for transferring data can be precisely evaluated and charged to the task that performs the operation. Another advantage of this approach is that the kernel does not need to be modified as the I/O devices are replaced or added.

The main disadvantage of this solution is a low processor efficiency on I/O operations, due to the busy wait of the tasks while accessing the device registers. Another minor problem is that the application tasks must have the knowledge of all low-level details of the devices that they want to handle. However, this can be easily solved by encapsulating all device-dependent routines in a set of library functions that can be called by the application tasks. This approach is adopted in RK, a research hard real-time kernel designed to support multisensory robotics applications [LKP88].

APPROACH B

As in the previous approach, all interrupts from external devices are disabled, except the one from the timer. Unlike the previous solution, however, the devices are not

directly handled by the application tasks but are managed in turn by dedicated kernel routines, periodically activated by the timer.

This approach eliminates the unbounded delays due to the execution of interrupt drivers and confines all I/O operations to one or more periodic kernel tasks, whose computational load can be computed once for all and taken into account through a specific utilization factor. In some real-time systems, I/O devices are subdivided into two classes based on their speed: slow devices are multiplexed and served by a single cyclical I/O process running at a low rate, whereas fast devices are served by dedicated periodic system tasks, running at higher frequencies. The advantage of this approach with respect to the previous one is that all hardware details of the peripheral devices can be encapsulated into kernel procedures and do not need to be known by the application tasks.

Because the interrupts are disabled, the major problem of this approach is due to the busy wait of the kernel I/O handling routines, which makes the system less efficient during I/O operations. With respect to the previous approach, this case is characterized by a higher system overhead, due to the communication required among the application tasks and the I/O kernel routines for exchanging I/O data. Finally, since the device handling routines are part of the kernel, it has to be modified when some device is replaced or added. This type of solution is adopted in the MARS system [DRSK89, KDK+89].

APPROACH C

A third approach that can be adopted in real-time systems to deal with the I/O devices is to leave all external interrupts enabled, while reducing the drivers to the least possible size. According to this method, the only purpose of each driver is to activate a proper task that will take care of the device management. Once activated, the device manager task executes under the direct control of the operating system, and it is guaranteed and scheduled just like any other application task. In this way, the priority that can be assigned to the device handling task is completely independent from other priorities and can be set according to the application requirements. Thus a control task can have a higher priority than a device handling task.

The idea behind this approach is schematically illustrated in Figure 1.3. The occurrence of event E generates an interrupt, which causes the execution of a driver associated with that interrupt. Unlike the traditional approach, this driver does not handle the device directly but only activates a dedicated task, J_E, which is the actual device manager.

The major advantage of this approach with respect to the previous ones is to eliminate the busy wait during I/O operations. Moreover, compared to the traditional technique,

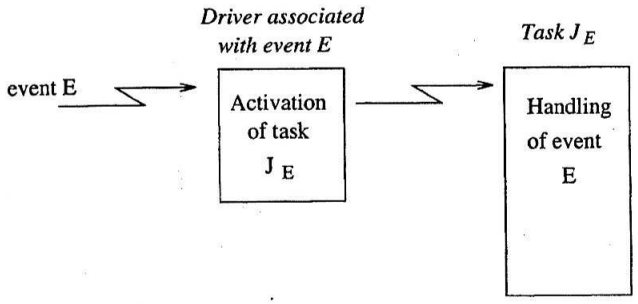

Figure 1.3 Activation of a device-handling task.

the unbounded delays introduced by the drivers during tasks' execution are also drastically reduced (although not completely removed), so the task execution times become more predictable. As a matter of fact, a little unbounded overhead due to the execution of the small drivers still remains in the system, and it should be taken into account in the guarantee mechanism. However, it can be neglected in most practical cases. This type of solution is adopted in the ARTS system [TK88, TM89], in HARTIK [BDN93, But93], and in SPRING [SR91].

1.3.4 SYSTEM CALLS

System predictability also depends on how the kernel primitives are implemented. In order to precisely evaluate the worst-case execution time of each task, all kernel calls should be characterized by a bounded execution time, used by the guarantee mechanism while performing the schedulability analysis of the application. In addition, in order to simplify this analysis, it would be desirable that each kernel primitive be preemptable. In fact, any nonpreemptable section could possibly delay the activation or the execution of critical activities, causing a timing fault to hard deadlines.

1.3.5 SEMAPHORES

The typical semaphore mechanism used in traditional operating systems is not suited for implementing real-time applications because it is subject to the priority inversion phenomenon, which occurs when a high-priority task is blocked by a low-priority task for an unbounded interval of time. Priority inversion must absolutely be avoided in real-time systems, since it introduces nondeterministic delays on the execution of critical tasks.

For the mutual exclusion problem, priority inversion can be avoided by adopting particular protocols that must be used every time a task wants to enter a critical section. For instance, efficient solutions are provided by *Priority Inheritance* [SRL90], *Priority Ceiling* [SRL90], and *Stack Resource Policy* [Bak91]. These protocols will be described and analyzed in Chapter 7. The basic idea behind these protocols is to modify the priority of the tasks based on the current resource usage and control the resource assignment through a test executed at the entrance of each critical section. The aim of the test is to bound the maximum blocking time of the tasks that share critical sections.

The implementation of such protocols may requires a substantial modification of the kernel, which concerns not only the *wait* and *signal* calls but also some data structures and mechanisms for task management.

1.3.6 MEMORY MANAGEMENT

Similarly to other kernel mechanisms, memory management techniques must not introduce nondeterministic delays during the execution of real-time activities. For example, demand paging schemes are not suitable for real-time applications subject to rigid time constraints, because of the large and unpredictable delays caused by page faults and page replacements. Typical solutions adopted in most real-time systems adhere to a memory segmentation rule with a fixed memory management scheme. Static partitioning is particularly efficient when application programs require similar amounts of memory.

In general, static allocation schemes for resources and memory management increase the predictability of the system but reduce its flexibility in dynamic environments. Therefore, depending on the particular application requirements, the system designer has to make the most suitable choices for balancing predictability versus flexibility.

1.3.7 PROGRAMMING LANGUAGE

Besides the hardware characteristics of the physical machine and the internal mechanisms implemented in the kernel, there are other factors that can determine the predictability of a real-time system. One of these factors is certainly the programming language used to develop the application. As the complexity of real-time systems increases, high demand will be placed on the programming abstractions provided by languages.

Unfortunately, current programming languages are not expressive enough to prescribe certain timing behavior and hence are not suited for realizing predictable real-time applications. For example, the Ada language (demanded by the Department of Defense of the United States for implementing embedded real-time concurrent applications) does not allow the definition of explicit time constraints on tasks' execution. The *delay* statement puts only a lower bound on the time the task is suspended, and there is no language support to guarantee that a task cannot be delayed longer than a desired upper bound. The existence of nondeterministic constructs, such as the *select* statement, prevents the performing of a reliable worst-case analysis of the concurrent activities. Moreover, the lack of protocols for accessing shared resources allows a high-priority task to wait for a low-priority task for an unbounded duration. As a consequence, if a real-time application is implemented using the Ada language, the resulting timing behavior of the system is likely to be unpredictable.

Recently, new high-level languages have been proposed to support the development of hard real-time applications. For example, *Real-Time Euclid* [KS86] is a programming language specifically designed to address reliability and guaranteed schedulability issues in real-time systems. To achieve this goal, Real-Time Euclid forces the programmer to specify time bounds and timeout exceptions in all loops, waits, and device accessing statements. Moreover, it imposes several programming restrictions, such as the ones listed below:

- **Absence of dynamic data structures**. Third-generation languages normally permit the use of dynamic arrays, pointers, and arbitrarily long strings. In real-time languages, however, these features must be eliminated because they would prevent a correct evaluation of the time required to allocate and deallocate dynamic structures.
- **Absence of recursion**. If recursive calls were permitted, the schedulability analyzer could not determine the execution time of subprograms involving recursion or how much storage will be required during execution.
- **Time-bounded loops**. In order to estimate the duration of the cycles at compile time, Real-Time Euclid forces the programmer to specify the maximum number of iterations for each loop construct.

Real-Time Euclid also allows the classification of processes as periodic or aperiodic and provides statements for specifying task timing constraints, such as activation time and period, as well as system timing parameters, such as the time resolution.

Another high-level language for programming hard real-time applications is *Real-Time Concurrent C* [GR91]. It extends Concurrent C by providing facilities to specify

periodicity and deadline constraints, seek guarantees that timing constraints will be met, and perform alternative actions when either the timing constraints cannot be met or guarantees are not available. With respect to Real-Time Euclid, which has been designed to support static real-time systems, where guarantees are made at compile time, Real-Time Concurrent C is oriented to dynamic systems, where tasks can be activated at run time. Another important feature of Real-Time Concurrent C is that it permits the association of a deadline with any statement, using the following construct:

within deadline *(d) statement-1*
[**else** *statement-2*]

If the execution of *statement-1* starts at time t and is not completed at time $(t + d)$, then its execution is terminated and *statement-2*, if specified, is executed.

Clearly, any real-time construct introduced in a language must be supported by the operating system through dedicated kernel services, which must be designed to be efficient and analyzable. Among all kernel mechanisms that influence predictability, the scheduling algorithm is certainly the most important factor, since it is responsible for satisfying timing and resource contention requirements.

In the rest of this book, several scheduling algorithms are illustrated and analyzed under different constraints and assumptions. Each algorithm is characterized in terms of performance and complexity to assist a designer in the development of reliable real-time applications.

Exercises

1.1 Explain the difference between fast computing and real-time computing.

1.2 What are the main limitations of the current real-time kernels for the development of critical control applications?

1.3 Discuss the features that a real-time system should have for exhibiting a predictable timing behavior.

1.4 Describe the approches that can be used in a real-time system to handle peripheral I/O devices in a predictable fashion.

1.5 Which programming restrictions should be used in a programming language to permit the analysis of real-time applications? Suggest some extensions that could be included in a language for real-time systems.

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 (or *thread*) 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 to 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.

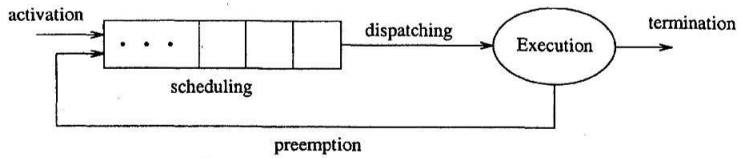

Figure 2.1 Queue of ready tasks waiting for execution.

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 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 criticality (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*.
- 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.

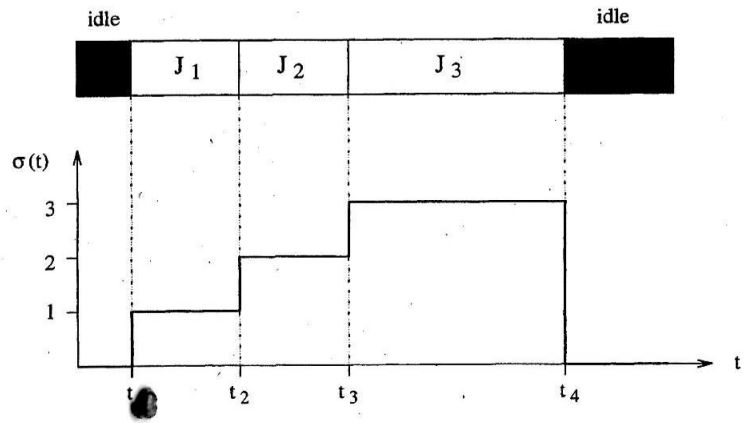

Figure 2.2 Schedule obtained by executing three tasks J_1, J_2, and J_3.

- 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.

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 it should complete its execution without causing any damage to the system. If a deadline is specified with respect to the 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 two classes:

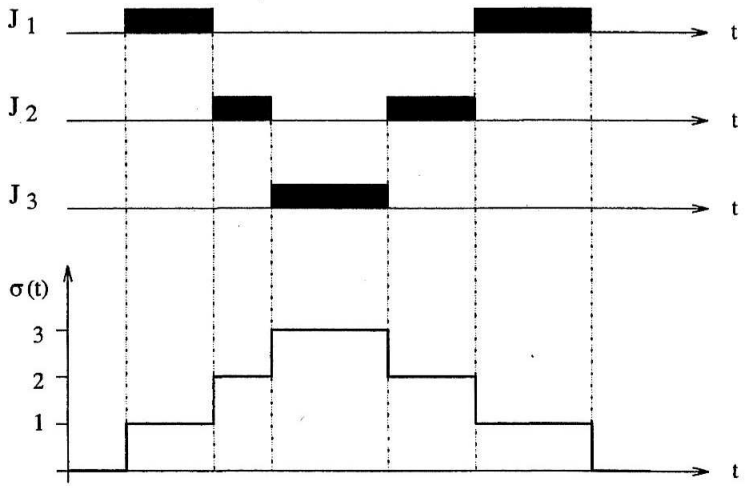

Figure 2.3 Example of a preemptive schedule.

- **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 be guaranteed a priori 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;
- **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 (if hard), or performance degradation (if soft);
- **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;

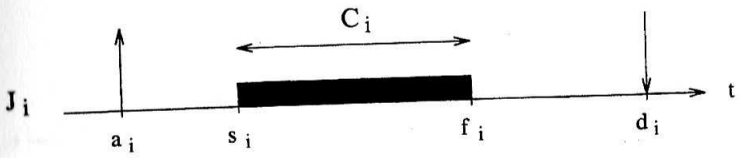

Figure 2.4 Typical parameters of a real-time task.

- **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 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.

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 task can be completely characterized by its computation time C_i and its relative deadline D_i (often set equal to 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 jobs; however, their activation is not regular.

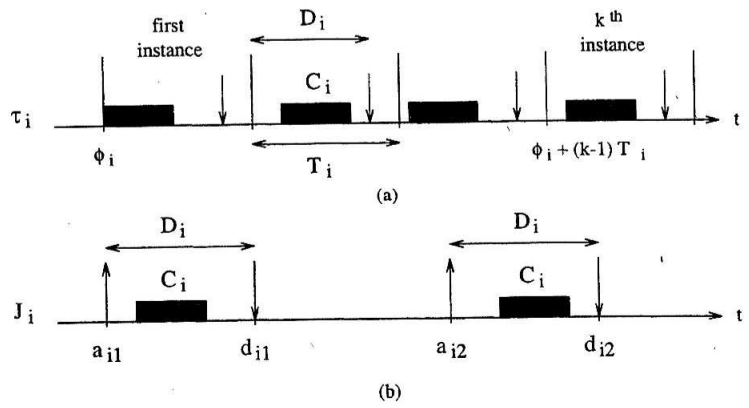

Figure 2.5 Sequence of instances for a periodic (a) and an aperiodic task (b).

An aperiodic task where consecutive jobs are separated by a minimum interarrival time is called a *sporadic* task. Figure 2.5 shows an example of 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 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, because 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

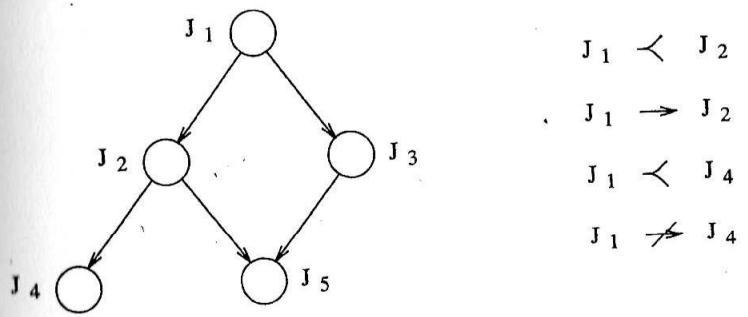

Figure 2.6 Precedence relations among five tasks.

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 denoted as *shape*);
- A task for computing the pixel disparities from the two images (it is denoted as *disp*);
- A task for determining the object height from the results achieved by the *disp* task (it is denoted as *H*);

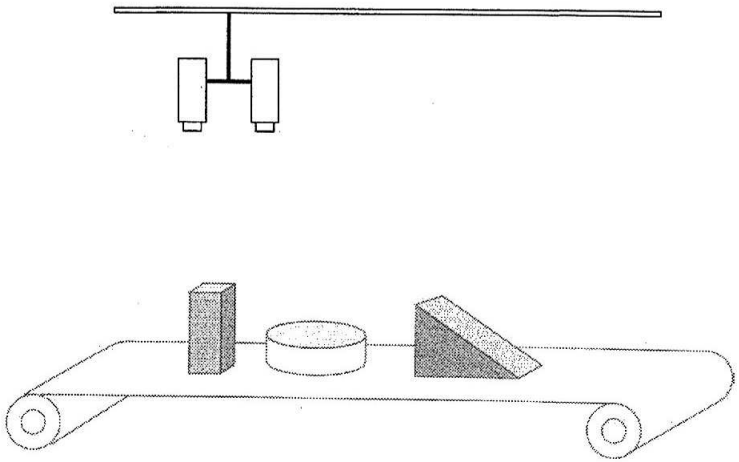

Figure 2.7 Industrial application which requires a visual recognition of objects on a conveyor belt.

- 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 a data base; it is denoted 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.

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*.

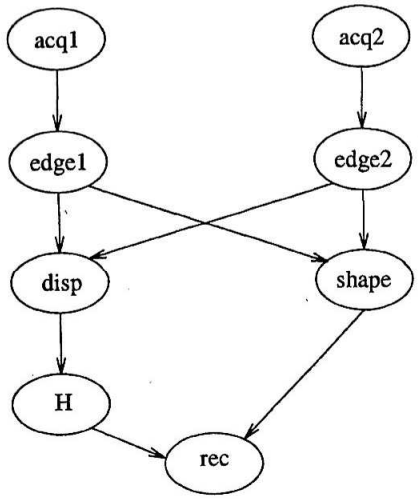

Figure 2.8 Precedence graph associated with the robotic application illustrated in Figure 2.7.

To maintain data consistency, many shared resources do not allow simultaneous accesses but require mutual exclusion among competing tasks. They are called *mutually exclusive resources*. Let R be a mutually 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 mutually exclusive resources, operating systems usually provide a synchronization mechanism (such as semaphores) that can be used by tasks to create critical sections. Hence, when we say that two or more tasks have resource constraints, we mean that they share mutually exclusive resources, and hence they have to be synchronized.

Consider two tasks J_1 and J_2 that share a mutually 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).

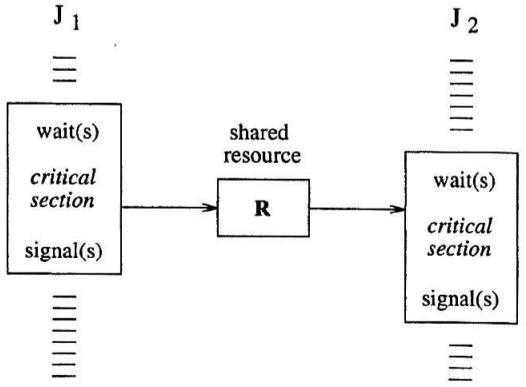

Figure 2.9 Structure of two tasks that share a mutually exclusive resource.

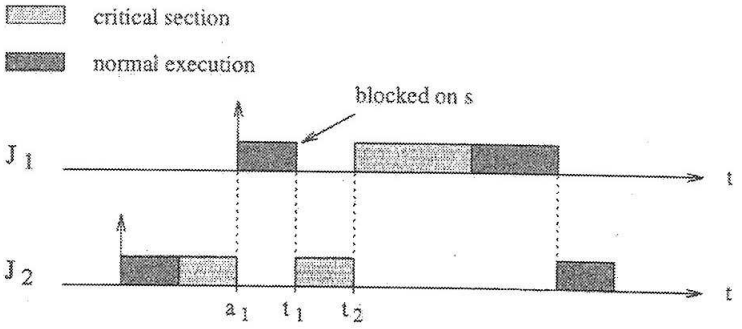

Figure 2.10 Example of blocking on a mutually exclusive resource.

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 its critical section and locks the semaphore. While J_2 is executing its 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 a mutually 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 that protects the resource. When a running task executes a *wait* primitive on a locked

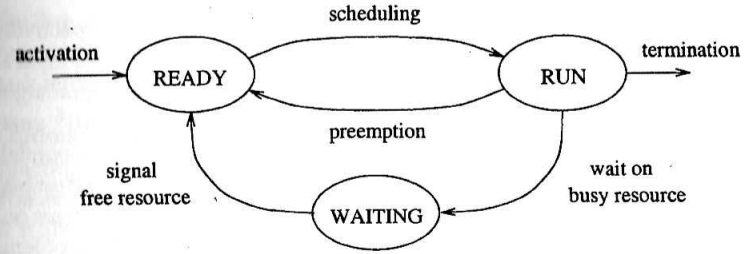

Figure 2.11 Waiting state caused by resource constraints.

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.

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 r types of resources $R = \{R_1, R_2, \ldots, R_r\}$. 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 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 $Q \in$ **NP** and, for every $Q' \in$ **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 $Q \in$ **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 always finds a feasible schedule whenever there exists one.
- **Heuristic**. An algorithm is said to be heuristic if it searches for a feasible schedule using an objective function (*heuristic function*). Heuristic algorithms do not guarantee to find the optimal schedule, even if there exists one.

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 for 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 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.

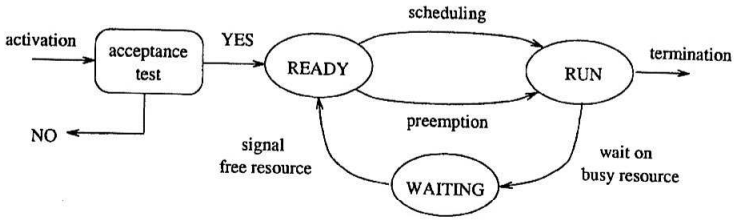

Figure 2.12 Scheme of the guarantee mechanism used in dynamic hard real-time systems.

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.

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 to be schedulable. If J' is not schedulable, then task J_{new} is rejected to preserve the feasibility of the current task set.

It is worth noticing that, since the guarantee mechanism is based on worst-case assumptions, a task could be unnecessarily 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 the *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} is accepted, all other tasks (previously schedulable) will 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.

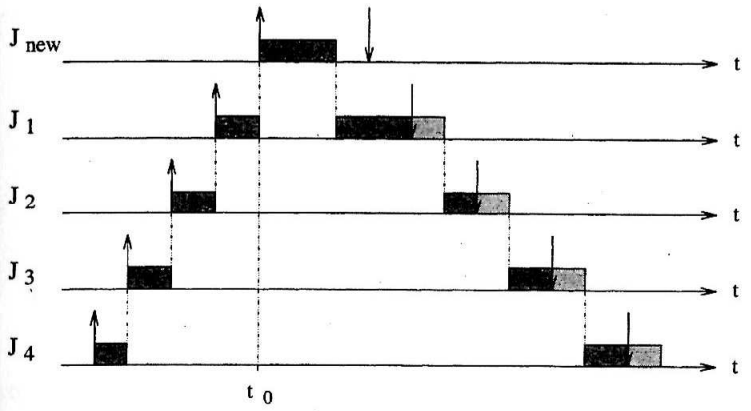

Figure 2.13 Example of domino effect.

BEST-EFFORT ALGORITHMS

In certain real-time applications, computational activities have soft timing constraints that should be met whenever possible to satisfy system requirements. However, no catastrophic events occur if one or more tasks miss their deadlines. The only consequence associated with a timing fault is a performance degradation of the system.

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 queued 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.

Average response time:

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

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.

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.

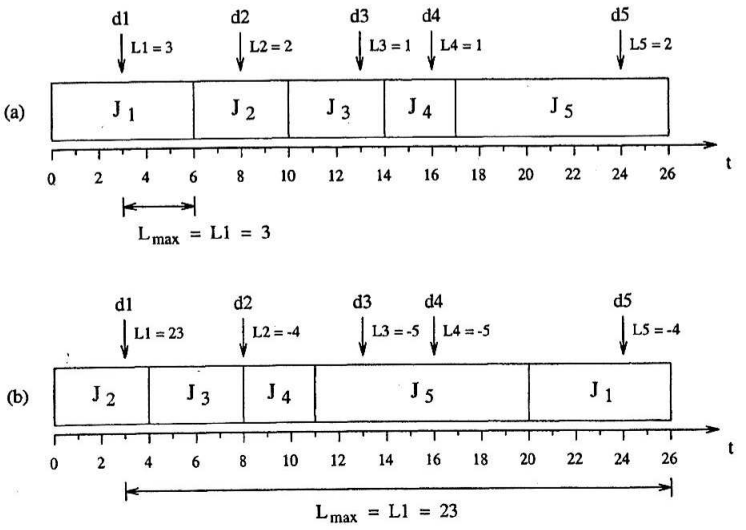

Figure 2.14 The schedule in a minimizes the maximum lateness, but all tasks miss their deadline. The schedule in b has a greater maximum lateness, but four tasks out of five complete before their deadline.

The cost function selected for evaluating the performance of a 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 2.14. The schedule shown in Figure 2.14a minimizes the maximum lateness, but all tasks miss their deadline. On the other hand, the schedule shown in Figure 2.14b has a greater maximum lateness, but four tasks out of five complete before their deadline.

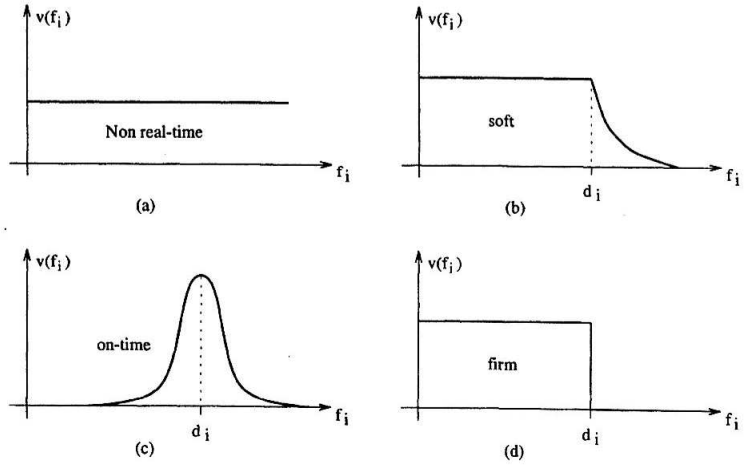

Figure 2.15 Example of cost functions for different types of tasks.

When tasks have soft deadlines and the application concern 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 2.15 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. 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. These types of tasks are denoted as *firm*.

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 8.

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, and an increase of computational power in the supporting hardware does not always cause an improvement on the performance of a task set. These particular situations, called Richard's anomalies, have been described by Graham in 1976 and refer to task sets with precedence relations executed in a multiprocessor environment. Designers should be aware of such insidious anomalies, so that they can avoid them. The most important anomalies are expressed by the following theorem [Gra76, SSDNB95]:

Theorem 2.1 (Graham) *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 better illustrate why this theorem is true.

Let us consider a task set composed by nine tasks $J = \{J_1, J_2, \ldots, J_9\}$, sorted by decreasing priorities, so that J_i priority is greater than J_j priority if and only if $i < j$. Moreover, tasks are subject to precedence constraints that are described through the graph shown in Figure 2.16. Computation times are indicated in parentheses.

If the above set is executed on a parallel machine with three processors, we obtain the optimal schedule σ^* illustrated in Figure 2.17, where the global completion time is $t_e = 12$ units of time.

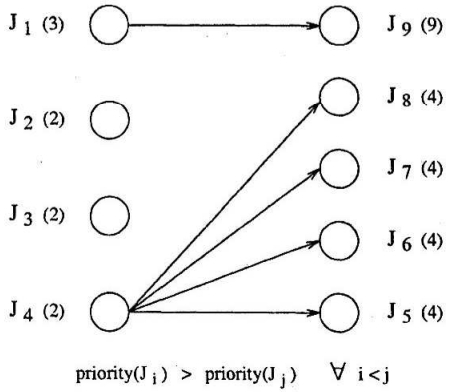

Figure 2.16 Precedence graph of the task set J; numbers in parentheses indicate computation times.

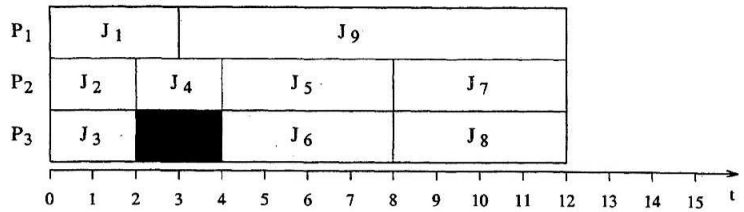

Figure 2.17 Optimal schedule of task set J on a three-processor machine.

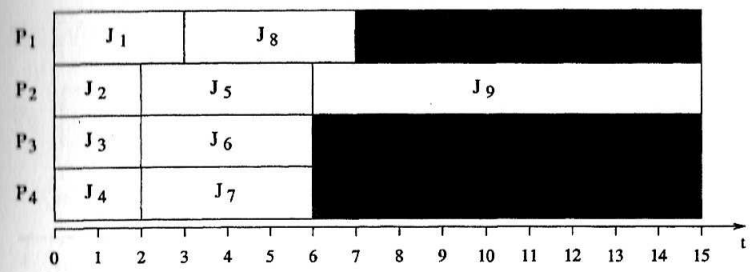

Figure 2.18 Schedule of task set J on a four-processor machine.

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 2.18, which is characterized by a global completion time of $t_c = 15$ units of time. In these examples, it is implicitly assumed that tasks are allocated to the first available processor.

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 if we reduce 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 2.19.

PRECEDENCE CONSTRAINTS WEAKENED

Scheduling anomalies can also arise if we remove precedence constraints from the directed acyclic graph depicted in Figure 2.16. For instance, if we remove the precedence relations between task J_4 and tasks J_5 and J_6 (see Figure 2.20a), we obtain the schedule shown in Figure 2.20b, which is characterized by a global completion time of $t_c = 16$ units of time.

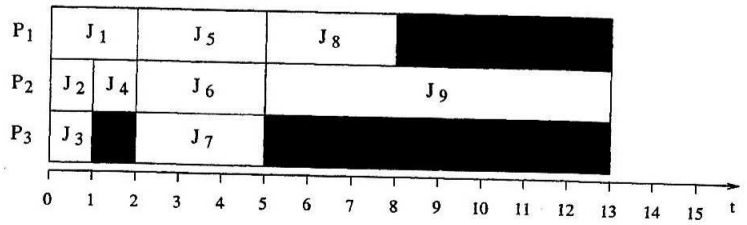

Figure 2.19 Schedule of task set J on three processors, with computation times reduced by one unit of time.

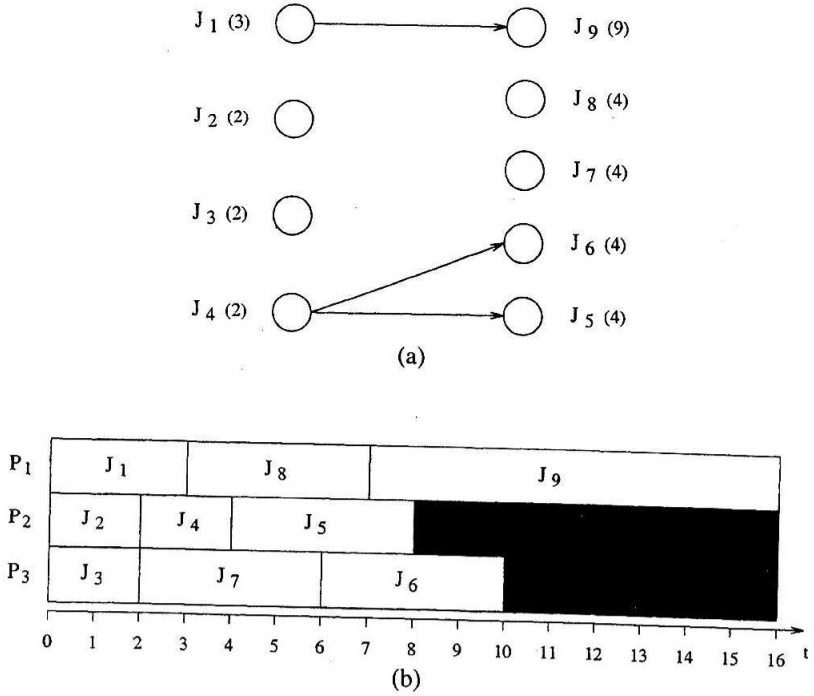

Figure 2.20 **a.** Precedence graph of task set J obtained by removing the constraints on tasks J_5 and J_6. **b.** Schedule of task set J on three processors, with precedence constraints weakened.

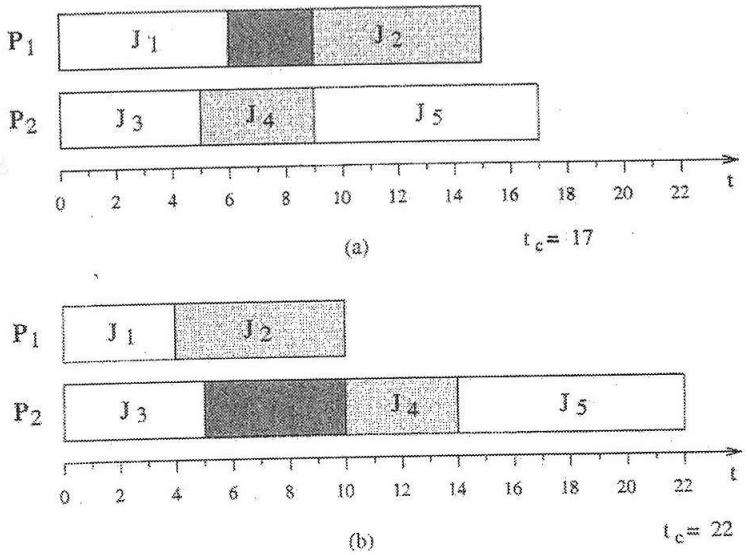

Figure 2.21 Example of anomaly under resource constraints. If J_2 and J_4 share the same resource in exclusive mode, the optimal schedule length (a) increases if the computation time of task J_1 is reduced (b). Task are statically allocated on the processors.

ANOMALIES UNDER RESOURCE CONSTRAINTS

As a last example of scheduling anomalies, we will show how the schedule length of a task set can increase when reducing tasks' computation times in the presence of shared resources. Consider the case illustrated in Figure 2.21, where five tasks are statically allocated on two processors: tasks J_1 and J_2 on processor P_1, and tasks J_3, J_4 and J_5 on processor P_2. Moreover, tasks 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 2.21a, where the total completion time is $t_c = 17$.

If we now reduce the computation time of task J_1 on the first processor, then J_2 can begin earlier and take the resource before task J_4. As a consequence, task J_4 must now block over the shared resource and possibly miss its deadline. This situation is illustrated in Figure 2.21b. 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 this idle time, while addressing the anomalies so that they will not occur.

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, on-line 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.

3

APERIODIC TASK SCHEDULING

3.1 INTRODUCTION

In this chapter we present a variety of algorithms for scheduling real-time aperiodic tasks on a single machine environment. Each algorithm represents a solution for a particular scheduling problem, which is expressed through a set of assumptions on the task set and by an optimality criterion to be used on the schedule. The restrictions made on the task set are aimed at simplifying the algorithm in terms of time complexity. When no restrictions are applied on the application tasks, the complexity can be reduced by employing heuristic approaches, which do not guarantee to find the optimal solution to a problem, but can still guarantee a feasible schedule in a wide range of situations.

Although the algorithms described in this chapter are presented for scheduling aperiodic tasks on uniprocessor systems, many of them can be extended to work on multiprocessor or distributed architectures and deal with more complex task models.

To facilitate the description of the scheduling problems presented in this chapter we introduce a systematic notation that could serve as a basis for a classification scheme. Such a notation, proposed by Graham et al. [GLLK79], classifies all algorithms using three fields $\alpha \mid \beta \mid \gamma$, having the following meaning:

- The first field α describes the machine environment on which the task set has to be scheduled (uniprocessor, multiprocessor, distributed architecture, and so on).
- The second field β describes task and resource characteristics (preemptive, independent versus precedence constrained, synchronous activations, and so on).