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 the following:

- Chemical and nuclear plant control,
- control of complex production processes,
- railway switching systems,
- automotive applications,
- flight control systems,
- environmental acquisition and monitoring,
- telecommunication systems,
- medical systems,
- industrial automation,
- robotics,

G.C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Real-Time Systems Series 24, DOI 10.1007/978-1-4614-0676-1_1,

- military systems,
- space missions,
- consumer electronic devices,
- multimedia systems,
- smart toys, and
- virtual reality.

In many cases, the real-time computer running the application is embedded into the system to be controlled. Embedded systems span from small portable devices (e.g., cellular phones, cameras, navigators, ECG Holter devices, smart toys) to larger systems (e.g., industrial robots, cars, aircrafts).

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 complexity of the application software 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 timing 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 mechanisms 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 damages to the environment.

A high percentage of accidents that occur in nuclear power plants, space missions, or defense 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 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 not discovered. Later, by analyzing the code of the processes, it was 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 onboard 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 onboard 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 injuries and enormous economic damage. Later on, it was discovered that, because of a long interrupt handling routine running with disable interrupts, the real-time clock of the onboard computer was missing some clock interrupts, thus 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 situation never experienced before), thus accumulating a total delay of 343 milliseconds. Such a delay caused a prediction error in the verification phase of 687 meters! The bug was corrected on

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

February 26, the day after the accident, by inserting a few preemption points inside the long interrupt handler.

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.

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 timing 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 in 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. Let us consider the meaning of the words *time* and *real* more closely.

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.

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.

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.

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.

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.

As a matter of fact, the Ariane 5 accident occurred because the characteristics of the launcher were not taken into account in the implementation of the control software [Bab97, JM97]. On June 4, 1996, the Ariane 5 launcher ended in a failure 37 seconds after initiation of the flight sequence. At an altitude of about 3,700 meters, the launcher started deflecting from its correct path, and a few seconds later it was destroyed by its automated self-destruct system. The failure was caused by an operand error originated in a routine called by the Inertial Reference System for converting accelerometric data from 64-bit floating point to 16-bit signed integer.

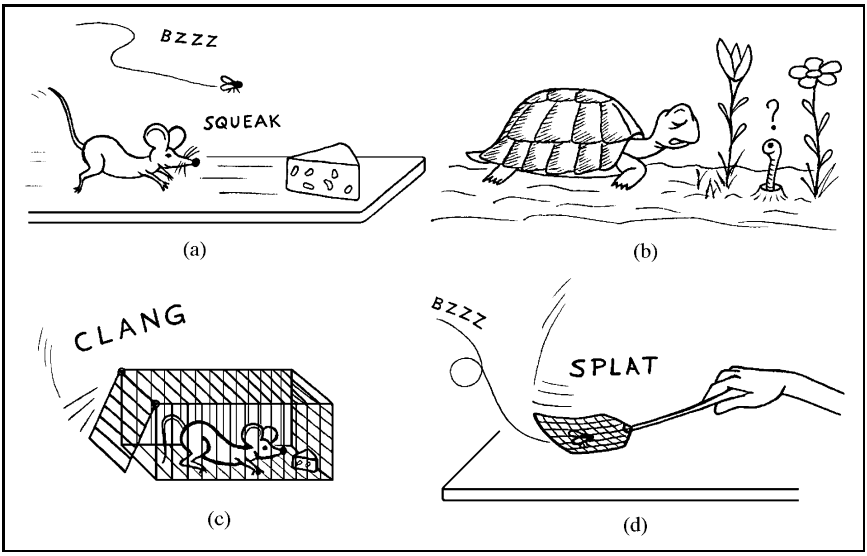

Figure 1.1 Both the mouse (a) and the turtle (b) behave in real time with respect to their natural habitat. Nevertheless, the survival of fast animals such as a mouse or a fly can be jeopardized by events (c and d) quicker than their reactive capabilities.

One value was too large to be converted and the program was not explicitly designed to handle the integer overflow error, so the Inertial Reference System halted, as specified in other requirements, leaving the luncher without inertial guidance. The conversion error occurred because the control software was reused from the Ariane 4 vehicle, whose dynamics was different from that of the Ariane 5. In particular, the variable containing the horizontal velocity of the rocket went out of range (since larger than the maximum value planned for the Ariane 4), thus generating the error that caused the loss of guidance.

The examples considered above indicate that 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.

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.

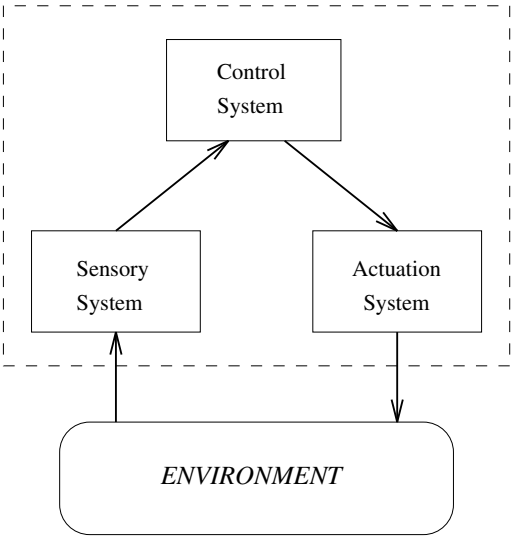

Figure 1.2 Block diagram of a generic real-time control system.

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]

> *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.

[2] From John Stankovic's notes.

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, a real-time task can be distinguished in three categories:

- **Hard**: A real-time task is said to be *hard* if producing the results after its deadline may cause catastrophic consequences on the system under control.
- **Firm**: A real-time task is said to be *firm* if producing the results after its deadline is useless for the system, but does not cause any damage.
- **Soft**: A real-time task is said to be *soft* if producing the results after its deadline has still some utility for the system, although causing a performance degradation.

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, firm, and soft activities; therefore a hard real-time system should be designed to handle all such task categories using different strategies. In general, when an application consists of a hybrid task set, all hard tasks should be guaranteed off line, firm tasks should be guaranteed on line, aborting them if their deadline cannot be met, and soft tasks should be handled to minimize their average response time.

Examples of hard tasks can be found in safety-critical systems, and are typically related to sensing, actuation, and control activities, such as the following:

- Sensory data acquisition;
- data filtering and prediction;
- detection of critical conditions;
- data fusion and image processing;
- actuator servoing;
- low-level control of critical system components; and
- action planning for systems that tightly interact with the environment.

Examples of firm activities can be found in networked applications and multimedia systems, where skipping a packet or a video frame is less critical than processing it with a long delay. Thus, they include the following:

- Video playing;
- audio/video encoding and decoding;
- on-line image processing;
- sensory data transmission in distributed systems.

Soft tasks are typically related to system-user interactions. Thus, they 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 timesharing 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 the following:

- **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**. This 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 the remapping of 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 deadlines of the tasks. 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 real-time 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 a 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 the following:

- **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 timing constraints and different criticality.
- **Predictability**. To achieve a desired level of performance, the system must be analyzable to predict the consequences of any scheduling decision. In safety critical applications, all timing requirements should be guaranteed off line, before putting system in operation. 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 to handle the exception.
- **Efficiency**. Most of real-time systems are embedded into small devices with severe constraints in terms of space, weight, energy, memory, and computational power. In these systems, an efficient management of the available resources by the operating system is essential for achieving a desired performance.

- **Robustness**. Real-time systems must not collapse when they are subject to peak-load conditions, so they must be designed to manage all anticipated load scenarios. Overload management and adaptation behavior are essential features to handle systems with variable resource needs and high load variations.
- **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 non-deterministic factors that prevent a precise estimation of the worst-case execution times (WCETs). 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.

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.

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 in 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 Mbyte 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 in 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.

In preemptive systems, the cache behavior is also affected by the number of preemptions. In fact, preemption destroys program locality and heavily increases the number of cache misses due to the lines evicted by the preempting task. Moreover, the cache-related preemption delay (CRPD) depends on the specific point at which preemption takes place; therefore it is very difficult to precisely estimate [AG08, GA07]. Bui et al. [BCSM08] showed that on a PowerPC MPC7410 with 2 MByte two-way associative L2 cache the WCET increment due to cache interference can be as large as 33 percent of the WCET measured in non-preemptive mode.

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 encap-

sulating 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 and 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 to 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 the 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 will be the actual device manager.

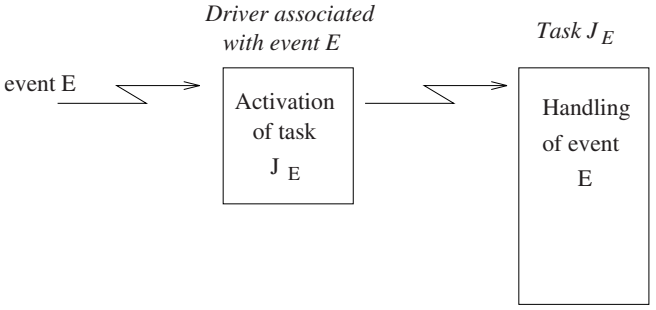

Figure 1.3 Activation of a device-handling task.

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, 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 is desirable that each kernel primitive be preemptable. In fact, any non-preemptable 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 *Basic 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 require 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 to 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 against 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 (required 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 for each loop construct the maximum number of iterations.

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, to seek guarantees that timing constraints will be met, and to 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.