6 Elements for Realtime Programming

Introduction

In this final chapter, we will look at topics that are concerned with the issues of realtime programming. We start by giving a description of realtime systems and by identifying their special characteristics and requirements and proceed by examining how our three languages allow a programmer to meet these requirements. This includes some considerations of "low level" topics such as the impact of priorities (including priority inversion), interrupts, queuing policies, scheduling, and dispatching mechanisms.

Following that, we will deal with the issue of time in a concurrent program: *runtime estimation, temporal arrangements for actions, periodic activities,* and *deadline specification* will be of paramount interest here.

The last part of this chapter gives an introduction to PEARL 90, the Process Experiment Automation Real Time Language. This language was defined (and has been mainly used) in Germany.

A note on terminology: Should we ever omit mentioning the fact that a realtime system may well consist of several processors/computers, any statement concerning processors and actors shall be preceded by *for each processor,* .

6.1 Description of Realtime Systems

"There are many interpretations of the exact nature of a realtime system; however, they all have in common the notion of *response time*—the time taken for the system to generate output from some associated input." (Burns, 1989, [p. 2])

(Laplante, 1993, [p. 10]) gives the following: "A realtime system is a system that must satisfy explicit (bounded) response time constraints or risk severe consequences, including failure."

(Ben-Ari, 1990, [p. 164]) states: "Realtime programs are programs that must execute within strict constraints on response time."

Finally, the Oxford Dictionary of Computing gives the following definition: "Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness."

We are not going to assess these definitions, but it should be clear that *time* is of crucial importance in a realtime system. A realtime program is considered to be *erroneous* not only if its result is logically incorrect, but also if this result (be it logically correct or not) is delivered too late. In such systems, it would be better to not respond at all than to provide too late a response!

From this point of view, it can be argued that virtually all systems are realtime since we expect every system to react to input within a certain time. Even a text editor should respond to commands quickly (say, within one second) or it will become a torture to work with. However, the notion of failure can be used to discriminate realtime systems. Clearly, the inability of a text editor to respond quickly to user input cannot be regarded as fatal whereas for a nuclear reactor problem, failure to respond swiftly could result in a melt-down, which is apparently a disaster.

Most of the researchers in the field of realtime systems, therefore, distinguish between hard and soft realtime systems. "*Hard realtime systems* are those where it is absolutely imperative that responses occur within the specified *deadline*. *Soft realtime systems* are those where response times are important but the system will still function correctly if deadlines

are occasionally missed." (Burns, 1989, [p. 2]) This chapter will be mainly concerned with hard realtime elements.

The requirement that realtime systems be complex and of high performance is perhaps a common misconception. Checking a nuclear reactor's temperature (at given time rates) may require only the simplest computations and "a system does not have to process data in microseconds to be considered realtime; it must simply have response times that are constrained and thus predictable." (Laplante, 1993, [p. 11]) Of course, high efficiency is advantageous for meeting deadlines and should not be despised.

Examples of hard realtime systems include: control systems for nuclear power plants, avionics systems, navigation systems for spacecrafts or vessels, industrial production control systems (assembly pipelines), airline reservation systems, bank transaction systems, simulation systems, and the like.

A soft realtime system (besides, perhaps, the text editor), could be a network packet router. It is more appropriate to delay the transmission of packets due to high traffic than to ignore them completely.

"In a hard or soft realtime system, the computer is usually interfaced directly to some physical equipment and is dedicated to monitoring or controlling the operations of that equipment. A key feature of all these applications is the role of the computer as an information processing component within a larger engineering system. It is for this reason that such applications have become known as *embedded computer systems*." (Burns, 1989, [p. 3]) The terms *realtime* and *embedded* can be used interchangeably.

6.2 Special Requirements for Realtime Systems

From the preceding discussion, it should be clear that the concept of time in realtime systems is of utmost importance. We also note that virtually every realtime system is inherently concurrent (which is why we address realtime elements in this paper). This stems from the fact that a realtime system "will tend to consist of computers and several coexisting external devices with which the computer programs must simultaneously interact." (Burns, 1989, [p. 10])

The overall requirement is timeliness, especially deadline handling. The second major requirement is "controlling concurrency", i.e., to synchronize or coordinate actors (or, their concurrent behaviour) in order to meet these stringent timing requirements. Clearly, a realtime system will consist of many actors; some of which have hard deadlines while others have none. We require that it be possible for actors having hard deadlines to be handled in such a way that meeting these deadlines is possible. It is this handling that we would like to refer to as "concurrency control" or, to use more appropriate terms, *scheduling* and *dispatching*. In other words, after we have identified hard-deadline-actors and soft-deadline-actors in our system, it must be possible for the hard-deadline-actors to execute in preference to soft-deadline-actors in order to meet the hard deadlines.[1]

Before we actually go further into details, let us take a look at *scheduling* and *dispatching*. For that purpose, consider a realtime system with several actors—for example, the controlling system of a nuclear power plant: The *scheduler* of a realtime system is the component of the realtime system that manages the state transitions of actors. Remember from Chapter 2 that an actor, during its lifetime, can be in many states (although in only one at a given point in time): running, runnable/ready, exiting, blocked, etc. Transitions between the individual states were discussed in Chapter 2. We can imagine each state to have associated with it a *queue*. This queue actually contains the actors of the respective state. For example, the runnable queue, or ready queue, contains all actors that are ready to run, i.e., that could immediately use a processor were one available. The queue of the blocked actors contains all

[1] Note that we do not require the functional behaviour of the system to depend upon the order of the execution of actors. The *timing* behaviour—of course—is affected.

blocked actors, and so forth. Given an actor, should a transition from one state into another be necessary, it is the scheduler that performs this transition; i.e., the scheduler takes the actor from one queue and puts it into another.

When a blocked actor becomes ready, it is removed from the blocked queue and inserted into the ready queue by the scheduler.

The *dispatcher*, then, is the component of a realtime system that is solely concerned with the runnable queue and the processor(s). Each time it is appropriate for a new actor to be *dispatched* to one of the processors, the dispatcher selects one of the actors from the ready queue to become the currently running actor on that processor. The previously running actor is removed from the processor (by the dispatcher) and placed into a particular queue (by the scheduler), depending on its new state.[2] The dispatcher can thus be seen as a special purpose scheduler removing an actor from the ready queue and inserting it into the running queue, i.e., the processor. In essence, the dispatcher performs what is known as a *context switch*.

"To distinguish scheduling programs from the input queue for entry into the computing system from the problem of allocating the processor among the active processes already in the system, the term *scheduler* is reserved for the former and *dispatcher* for the latter." (Cupper, 1997, [p. 1681])

It should be noted, however, that schedulers and dispatchers are not exclusively bound to realtime systems. Any computer system (concurrent or not) has some kind of these components; there are, however, special requirements imposed upon them in a realtime system.

We can now reformulate the two major requirements for realtime systems: *How can actors be scheduled and dispatched so that deadlines can be met and how is it possible to specify deadlines (and other time related things like periodic activities, temporal arrangements for actions, and so on)?*

"In a non-realtime system, it is acceptable for any task that is executable to use the available processors. With realtime systems, control must be exercised over the use of system resources." (Burns, 1998, [p. 300]) This is because the system resources are scarce and deadlines must be met.

The first step is to identify hard-deadline-actors in our system, their respective memory and CPU-time requirements, and the collective workload of all actors that constitute the system—this is clearly a design issue and thus beyond the scope of this paper. Then, after having identified hard-deadline-actors, we must somehow be able to map the urgency of a hard-deadline-actor onto the actor as it is represented in the program. That is, we must give hard-deadline-actors *priority* over soft-deadline-actors. This is fairly simple since most concurrent programming languages support the concept of actor priorities, which accomplish exactly that. With these priorities, the programmer has the opportunity to state that one actor should execute in preference to others. Priorities can be used to express urgency of actors.

Unlike more complicated, however, is to ensure that the scheduler and the dispatcher actually take advantage of the priorities given by the programmer. It is *not* enough to specify urgency via priorities. The scheduling and dispatching algorithms must take priorities into consideration and, respectively, schedule and dispatch actors with high priorities in preference to actors with low priorities. This, at least, is a necessary prerequisite for hard-deadline-actors (provided they are assigned high priorities) to be able to execute and thus to meet their deadlines.

This, in essence, leads to the question as to how to ensure that actors with higher priorities are scheduled and dispatched in preference to actors having a lower priority.

[2] If this actor is still ready, then it is put back to the ready queue and it is said to have been *preempted* by the (now) currently running actor.

In the following sections, we will consider how our three languages support the programmer in its strive to assure this and the timing issues mentioned above.[3]

6.3 Elements for Realtime Programming in Ada

The facilities Ada provides for the implementation of realtime systems are not part of the core language but are defined in the optional Realtime Systems Annex, Annex D. According to the Rationale for Ada 95, this is largely due to the fact that Ada is a general-purpose programming language, which implies that not all the capabilities required to build applications can be sensibly put into the core language without prohibitively increasing its size and hurting other application domains. It was mentioned earlier (in Chapter 1) that implementing Annex D does not incur serious additional effort to the implementation of the mandatory parts of the language.

Note that this section is written under the assumption that an Ada implementation supports the Realtime Systems Annex.

6.3.1 Priorities, Scheduling and Dispatching

Setting priorities, both statically and dynamically, has already been discussed in Chapter 1 and will not be repeated here. We will, however, reiterate Ada's notion of a task priority as it is of utmost importance for the rest of this subsection: "A task priority is an integer value that indicates a degree of urgency and is the basis for resolving competing demands of tasks for resources. Unless otherwise specified, whenever tasks compete for processors or other implementation-defined resources, the resources are allocated to the task with the highest priority value. The *base priority* of a task is the priority with which it was created, or to which it was later set by `Dynamic_Priorities.Set_Priority` (see D.5). At all times, a task also has an *active priority*, which generally reflects its base priority as well as any priority it inherits from other sources. Priority inheritance is the process by which the priority of a task or other entity (e.g., a protected object; see D.3) is used in the evaluation of another task's active priority." (ARM, 1995, [D.1[15]]) We thus see that Ada supports expressing the urgency of tasks via priorities. "The task's active priority is used when the task competes for processors." (ARM, 1995, [D.1[19]]) Similarly, the task's active priority is used to determine the task's position in an entry queue if *Priority Queuing* (see below) is used.

Hence, priorities do have an impact on scheduling and dispatching; this influence will be scrutinized in the following. We are now going to cite quite large portions of the Realtime Systems Annex and intersperse them, when we feel it is appropriate, with our comments. To provide for visual distinction, we use horizontal rules (as we did in Chapter 4) to mark the beginning and the end of the quotations.

This clause describes the rules that determine which task is selected for execution when more than one task is ready (see 9.2). The rules have two parts: the task dispatching model (see D.2.1), and a specific task dispatching policy (see D.2.2).

The task dispatching model specifies preemptive scheduling, based on conceptual priority-ordered ready queues.

An alternative term often found in the literature is *priority-based preemptive scheduling.*

[3] There are, of course, other requirements for realtime systems such as reliability, robustness, fault tolerance, conformance to specification, and the like. But they apply to "normal" programs as well and are not specific to realtime systems.

A task runs (that is, it becomes a running task) only when it is ready (see 9.2) and the execution resources required by that task are available. Processors are allocated to tasks based on each task's active priority.

It is implementation defined whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy.

Task dispatching is the process by which one ready task is selected for execution on a processor. This selection is done at certain points during the execution of a task called *task dispatching points.*

An alternative notion for a task dispatching point is *scheduling event.*

A task reaches a task dispatching point whenever it becomes blocked, and whenever it becomes ready. In addition, the completion of an accept_statement (see 9.5.2), and task termination are task dispatching points for the executing task. Other task dispatching points are defined throughout this Annex.

Task dispatching policies are specified in terms of conceptual ready queues, task states, and task preemption. A ready queue is an ordered list of ready tasks. The first position in a queue is called the head of the queue, and the last position is called the tail of the queue. A task is ready if it is in a ready queue, or if it is running. Each processor has one ready queue for each priority value. At any instant, each ready queue of a processor contains exactly the set of tasks of that priority that are ready for execution on that processor, but are not running on any processor; that is, those tasks that are ready, are not running on any processor, and can be executed using that processor and other available resources. A task can be on the ready queues of more than one processor.

Note that it is somewhat confusing to regard a task to be ready if it is in a ready queue *or* if it is running. When a task is running it does not only have the potential to execute—it *executes.* Note further that, since an Annex D complying implementation is required to support at least 30 priority values, each processor has at least 30 distinct ready queues; one for each priority value. However, as these ready queues are purely notional, it is immaterial for the implementation whether there are distinct queues or whether there is just one queue for all priority values (or even for all processors and all priority values). We can, then, imagine this single queue to be priority-ordered. Of course, from the standpoint of comprehension, distinct queues are clearly more beneficial.

Each processor also has one *running* task, which is the task currently being executed by that processor. [If there is no task that actually needs a processor, then (ARM, 1995, [D.11[4]]) applies: "For each processor, there is a conceptual idle task, which is always ready. The base priority of the idle task is below `System.Any_Priority'First`.", cmnt. by author] Whenever a task running on a processor reaches a task dispatching point, one task is selected to run on that processor. The task selected is the one at the head of the highest priority nonempty ready queue; this task is then removed from all ready queues to which it belongs.

Ada *guarantees* that for each processor, the currently running task is always one with a priority higher than or equal to the priority of any other ready task (*equal to* applies if the highest priority nonempty ready queue has more than one queued task). However, the full evidence can only be seen after we have fully understood what actually constitutes a task

dispatching point. Some were given above but there are more to come. They will be defined in due course.

In addition, we would like to remark that the fact that "this [the currently running, cmnt. by author] task is removed from all ready queues to which it belongs" is slightly in contrast to Annex D's (previous) concept of regarding a task to be ready if it is in a ready queue or if it is running (which we find, as stated above, is confusing). We, personally, prefer to consider a task to be ready if it has the potential to run (but is not running due to the lack of a processor) and to consider it running if it is executing on a processor. That is, a task is ready if and only if it is in a ready queue. The mixing introduced by the Annex D slightly complicates the whole manner. Oh, and if there is a task at the head of the highest priority ready queue, does not that imply that this queue is nonempty?

A preemptible resource is a resource that while allocated to one task can be allocated (temporarily) to another instead. Processors are preemptible resources. Access to a protected object (see 9.5.1) is a nonpreemptible resource. When a higher-priority task is dispatched to the processor, and the previously running task is placed on the appropriate ready queue, the latter task is said to be *preempted*.

A new running task is also selected whenever there is a nonempty ready queue with a higher priority than the priority of the running task, or when the task dispatching policy requires a running task to go back to a ready queue. These are also task dispatching points.

We thus see that high priority tasks are capable of preempting low priority tasks. This is clearly a fateful issue in realtime systems since it enables hard-deadline-actors (provided they are assigned high enough a priority) to gain access to the processor(s) whenever they need it—clearly a prerequisite for their associated deadlines to be met. Of course, since the number of distinct priority values is limited and the number of hard-deadline-actors may well rise above that limit, we face a problem if there are too many hard-deadline-actors. It is beyond the scope of the chapter (and the paper) to discuss this thoroughly, but it must be understood that one of the preliminary steps in developing a realtime systems is what is called *schedulability analysis*, i.e., to determine whether all hard-deadline-actors, given the finite set of distinct priority values, can reach their deadlines. If this question cannot be answered in the affirmative, then a change/refinement of the system's design is necessary (for example, employing additional processors, trimming the runtime systems, reducing I/O, ...). This, in fact, is what we understand by *careful application of realtime facilities*. One cannot design a realtime system without paying heed to this subject.

We now turn our attention to the task dispatching policy. Ada defines a standard task dispatching policy, FIFO_Within_Priorities, which can be requested by using the following pragma:

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
```

The pragma is a configuration pragma—hence, it is valid for the whole program/partition. Note that when FIFO_Within_Priorities is in effect, then the Ceiling_Locking policy (see below) shall also be specified for the partition. The language defines only one task dispatching policy, FIFO_Within_Priorities, but an implementation is allowed to define additional ones (and is then, of course, required to document these policies).

A task dispatching policy specifies the details of task dispatching that are not covered by the basic task dispatching model. These rules govern when tasks are inserted into and deleted from the ready queues, and whether a task is inserted at the head or the tail of the queue for

its active priority. The task dispatching policy is specified by a `Task_Dispatching_Policy` configuration pragma. If no such pragma appears in any of the program units comprising a partition, the task dispatching policy for that partition is unspecified.

To be precise, we are talking about scheduling. But since dispatching is a special form of scheduling, the difference is only marginal. Note that *unspecified* is the same as *arbitrary* (it must not be confused with *implementation-defined*, however).

The language defines only one task dispatching policy, FIFO_Within_Priorities; when this policy is in effect, modifications to the ready queues occur only as follows:

- When a blocked task becomes ready, it is added at the tail of the ready queue for its active priority.
- When the active priority of a ready task that is not running changes, or the setting of its base priority takes effect, the task is removed from the ready queue for its old active priority and is added at the tail of the ready queue for its new active priority, except in the case where the active priority is lowered due to the loss of inherited priority, in which case the task is added at the head of the ready queue for its new active priority.
- When the setting of the base priority of a running task takes effect, the task is added to the tail of the ready queue for its active priority.

This is true regardless of whether the priority actually changes. The setting of a task's base priority as a result of a call to `Set_Priority` does not always take effect immediately when `Set_Priority` is called. The effect is deferred while the affected task performs a protected action. If the active priority of a running task is lowered due to loss of inherited priority (as it is upon completion of a protected operation) and there is a ready (i.e., not running) task of the same active priority, the running task continues to run (provided that there is no higher priority task).

For convenience, we give a survey of the situations in which the priority of a task changes (see (ARM, 1995, [D.1])):

- the base priority of a task is established when the task is created; the task being created inherits the active priority of its activator
- during a rendezvous, the accepting task inherits the active priority of the caller (but executes the rendezvous with its active priority, which is the maximum of all inherited priorities [including its own base priority])
- during a protected action, the caller inherits the ceiling priority (to be discussed shortly) of the target protected object
- a call to `Ada.Dynamic_Priorities.Set_Priority` sets the base priority of the specified task

- When a task executes a delay_statement that does not result in blocking, it is added to the tail of the ready queue for its active priority.

This is the reason why a `delay 0.0` or `delay until Clock` cannot be optimized away! Using such delays, it is possible for a task to yield the processor to another (ready) task of

equal priority.[4] Hence, other scheduling and dispatching mechanisms, most notably *voluntary round robin of equal priority tasks*, can be easily achieved.

Each of the events specified above is a task dispatching point (see D.2.1).

In addition, when a task is preempted, it is added at the head of the ready queue for its active priority.

The name FIFO_Within_Priorities should be clear now: tasks sharing the same priority are queued in FIFO order.

Clearly, no one scheduling/dispatching policy can be acceptable to all users or suitable for all applications. Ada addresses this issue by permitting other policies to be implemented and used. Since it is not the intention of this chapter to provide a detailed introduction to scheduling theory and alternative scheduling algorithms, the inclined reader is referred to the rich set of literature pertinent to this subject.

Avoiding Priority Inversion—Ceiling Priorities

Clearly, the rules stated above try to ensure that for each processor, the currently running task is always one with a priority higher than or equal to the priority of any other ready task in the system. However, there are situations in which this attempt is thwarted, i.e., lower priority tasks are preferred to higher priority tasks. Consider two examples:

Assume that task entry queues are FIFO ordered (this is the default in Ada, see below) and suppose that a task `T1` with priority 10 issues an entry call `S.E` (let `S` be a server task with priority 8) which is queued. Now let `T2` with priority 15 issue `S.E` and be queued as well. `S.E`'s entry queue is thus given by Figure 6.1.

Figure 6.1: The entry queue of `S.E`

If `S` now reaches an accept statement corresponding to the entry `E`, then the rendezvous is started with `T1` since FIFO is in effect. Thus, a lower priority task is preferred to a higher priority task.[5] It can be argued that in the case of a rendezvous, the caller does not execute on the processor (it voluntarily suspended itself), i.e., that it does not need the processor. But since `S` executes on behalf of `T1` with priority 10 and would execute on behalf of `T2` with priority 15 if `T2` were first in the ready queue (note that this—as long as FIFO is in effect—introduces a race condition!), the priority based preference of tasks is violated. One might argue, however, that this example is a trifle peculiar since FIFO queuing contradicts entry calls issued with priority, and that one should use priority queuing instead of FIFO. Granted, but note that this might not be obvious at first glance, and Ada 83 (it is conceivable that many systems have not yet been updated) offers only FIFO (which is why this is the default in Ada 95). Anyway, this example once more illustrates that realtime facilities cannot be applied rashly.

---

[4] in Java, the method `java.lang.Thread.yield()` does the same

[5] We should add, for completeness, that a rendezvous is always executed with the priority that is the maximum of the priorities of the caller and the callee. This, admittedly, is only revealed by very careful study of (ARM, 1995, [D.1]).

The second example is taken from (Burns, 1998, [p. 280]): "Consider, for illustration, a three-task system. The tasks have high, medium, and low priority, and will be identified by the labels *H*, *M*, and *L*. Assume that *H* and *L* share data which is encapsulated in a protected object, *P*. The following execution sequence is possible:

1. *L* is released, executes, and enters *P*.
2. *M* is released and preempts *L* while it is executing in *P*.
3. *H* is released and preempts *M*.
4. *H* executes a call on *P*.

Now, *H* cannot gain access to *P* as *L* has the mutual exclusion lock. Hence, *H* is suspended. The next highest priority task is *M* and, hence, *M* will continue. As a result, *H* must wait for *M* to finish before it can execute again." The situation is shown in Figure 6.2.

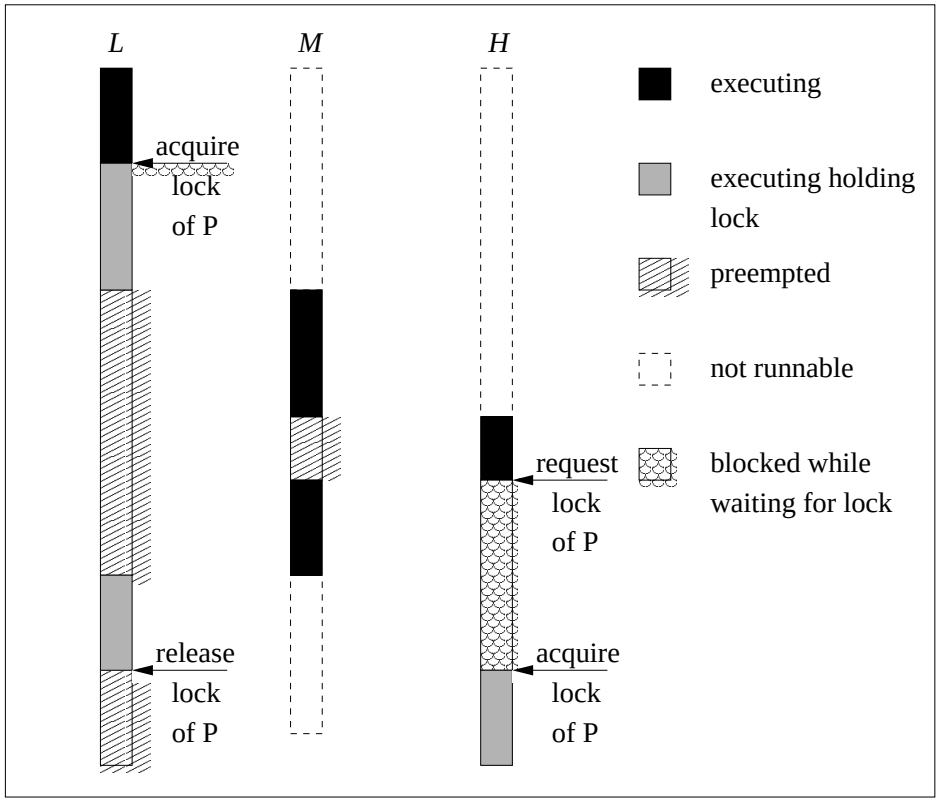

Figure 6.2: Priority Inversion

Here, we have clearly the case of all the three tasks contending for the processor. There is no "working on behalf of others".

The phenomenon we came across in the two examples is called *priority inversion* as the rules of priority based preference seem to have been inverted. As we have seen, priority inversion bears the risk of preferring lower priority tasks to higher priority tasks and, thus, has the potential of undermining the priority model and can, therefore, cause deadlines of hard-deadline-actors to be missed. Hence, it is worth paying attention to it (and especially to means that help prevent or, at least, limit the detrimental effect of priority inversion).

An implementation shall document:

- The maximum priority inversion a user task can experience due to activity of the implementation (on behalf of lower priority tasks), and

- whether execution of a task can be preempted by the implementation processing of delay expirations for lower priority tasks, and if so, for how long.

The first example can be ameliorated by having the entry queues be ordered by the priority of the arriving calls (it is, of course, not possible, in general, to arrange for `T2` to have its call issued before `T1`). Note that this will be addressed below.

For the prevention/bounding of priority inversion, Ada integrates the concept of a *ceiling priority* for a protected object into the general facilities provided by base and active priorities. The ceiling priority of a protected object is an upper bound (hence the name) on the active priority a task can have when it calls protected operations of that protected object. Furthermore, the notion of *priority inheritance* is used to describe the semantics of this mechanism. By requiring that a task's active priority be *raised* to the ceiling priority of a protected object, P, while the task executes a protected operation of P, priority inversion can be effectively eliminated. This, of course, necessitates that P's ceiling priority is higher than the active priority of any task that calls a protected operation of P. Since, at runtime, a check is made to verify this condition, and since a task that has an active priority higher than P's ceiling receives `Program_Error` upon calling one of P's operations, we can indeed say that for each task that executes a protected operation of P, the active priority of that task is *raised* to P's ceiling. The *ceiling locking policy*, which enables ceiling locking, can be employed by using the following pragma:

```
pragma Locking_Policy(Ceiling_Locking);
```

The details follow:

This clause specifies the interactions between priority task scheduling and protected object ceilings. This interaction is based on the concept of the ceiling priority of a protected object.

A locking policy specifies the details of protected object locking. These rules specify whether or not protected objects have priorities, and the relationships between these priorities and task priorities. In addition, the policy specifies the state of a task when it executes a protected action, and how its active priority is affected by the locking. The locking policy is specified by a `Locking_Policy` pragma. For implementation-defined locking policies, the effect of a `Priority` or `Interrupt_Priority` pragma on a protected object is implementation defined. If no `Locking_Policy` pragma appears in any of the program units comprising a partition, the locking policy for that partition, as well as the effect of specifying either a `Priority` or `Interrupt_Priority` pragma for a protected object, are implementation defined.

Note that this is not *unspecified* as was the case when no task dispatching policy had been requested.

There is one predefined locking policy, Ceiling_Locking; this policy is defined as follows:

- Every protected object has a ceiling priority, which is determined by either a `Priority` or `Interrupt_Priority` pragma as defined in D.1. The *ceiling priority* of a protected object (or ceiling, for short) is an upper bound on the active priority a task can have when it calls protected operations of that protected object.
- The expression of a `Priority` or `Interrupt_Priority` pragma is evaluated as part of the creation of the corresponding protected object and converted to the

subtype `System.Any_Priority` or `System.Interrupt_Priority`, respectively. The value of the expression is the ceiling priority of the corresponding protected object.

- If an `Interrupt_Handler` or `Attach_Handler` pragma (see C.3.1) appears in a protected_definition without an `Interrupt_Priority` pragma, the ceiling priority of protected objects of that type is implementation defined, but in the range of the subtype `System.Interrupt_Priority`.
- If no pragma `Priority`, `Interrupt_Priority`, `Interrupt_Handler`, or `Attach_Handler` is specified in the protected_definition, then the ceiling priority of the corresponding protected object is `System.Priority'Last`.
- While a task executes a protected action, it inherits the ceiling priority of the corresponding protected object.

Note that the task's base priority is unaffected—its active priority is changed. And it is this active priority that is used in all sorts of priority based contention. A couple of thoughts on `System.Any_Priority` vs. `System.Interrupt_Priority`:

```
subtype Any_Priority is Integer range <implementation-defined>;
subtype Priority is Any_Priority range
  Any_Priority'First .. <implementation-defined>;
subtype Interrupt_Priority is Any_Priority range
  Priority'Last + 1 .. Any_Priority'Last;
```

This approach ensures that task priorities and interrupt priorities are non-overlapping, i.e., a protected operation (of a protected object having an interrupt priority) used as an interrupt handler is able, when called, to preempt any task. This seems to be wise.

- When a task calls a protected operation, a check is made that its active priority is not higher than the ceiling of the corresponding protected object; `Program_Error` is raised if this check fails.

We continue by citing the NOTES section of (ARM, 1995, [D.3]). These notes are not mandatory but provide for further insight:

While a task executes in a protected action, it can be preempted only by tasks whose active priorities are higher than the ceiling priority of the protected object (the task can only loose the processor, not the mutual exclusion lock on the protected object, however [cmnt. by author]).

If a protected object has a ceiling priority in the range of `Interrupt_Priority`, certain interrupts are blocked while protected actions of that object execute. In the extreme, if the ceiling is `Interrupt_Priority'Last`, all blockable interrupts are blocked during that time.

The ceiling priority of a protected object has to be in the `Interrupt_Priority` range if one of its procedures is to be used as an interrupt handler (see C.3).

When specifying the ceiling of a protected object, one should choose a value that is at least as high as the highest active priority at which tasks can be executing when they call protected operations of that object. In determining this value, the following factors, which

can affect active priority, should be considered: the effect of `Set_Priority`, nested protected operations, entry calls, task activation, and other implementation-defined factors.

Attaching a protected procedure whose ceiling is below the interrupt hardware priority to an interrupt causes the execution of the program to be erroneous (see C.3.1).

On a single processor implementation, the ceiling priority rules guarantee that there is no possibility of deadlock involving only protected subprograms (excluding the case where a protected operation calls another protected operation on the same protected object) [this is defined to be a *bounded error* as it is a *potentially blocking operation*, see Chapter 5, cmnt. by author].

As with task dispatching policies, an implementation is allowed to define additional implementation-defined locking schemes.

With these tools in hand, our simple three-task system will behave as follows (Burns, 1998, [p. 280]):

1. L is released, executes, and enters P; its priority is raised to that of H (at least).
2. M is released but does not execute as the priority of M is less than the current priority of L.
3. H is released but does not execute as its priority is not higher than that of L at this time.
4. L exists P and has its priority lowered.
5. H can now execute and will enter and leave P when required.

The situation is depicted in Figure 6.3.

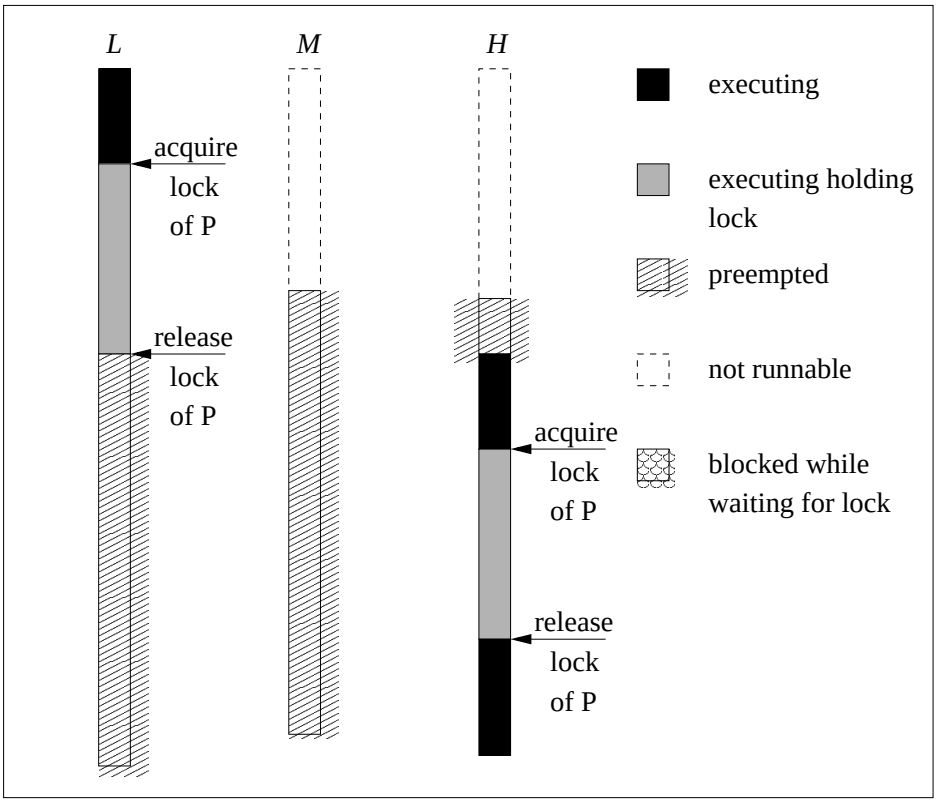

Figure 6.3: Ceiling Priority Inheritance

Note that by using FIFO_Within_Priorities, Ceiling_Locking, and priority ordered entry queues,[6] priority inversion can be avoided. FIFO_Within_Priorities guarantees that for processor contention, one of the highest priority task is the winner in that competition (it is the task at the head of the highest priority ready queue). Ceiling_Locking assures that a task executing a protected operation *is* the highest priority task for the duration of that operation (provided, of course, ceiling locking is applied with a ceiling priority as least as high as the highest active priority of all other tasks in the system). Hence, we can revert to FIFO_Within_Priorities. Last but not least, priority ordered entry queues provide for requests to be serviced in priority order. It is this issue that we will focus our attention on in the following subsubsection.

Entry Queuing Policies

There is no need for yet another motivation since we already pointed out why it might be beneficial to have entry queues be ordered by a certain criterion.[7] For upward compatibility with Ada 83, FIFO queuing (which was mandatory in Ada 83) is still the default queuing policy in Ada 95. But alternative policies can be defined and employed.

Here are the facts:

This clause specifies a mechanism for a user to choose an entry queuing policy. It also defines one such policy. Other policies are implementation defined.

The form of a pragma `Queuing_Policy` is as follows:

```
pragma Queuing_Policy(policy_identifier);
```

The policy_identifier shall be either FIFO_Queuing, Priority_Queuing or an implementation-defined identifier.

A `Queuing_Policy` pragma is a configuration pragma.

A queuing policy governs the order in which tasks are queued for entry service, and the order in which different entry queues are considered for service. The queuing policy is specified by a `Queuing_Policy` pragma.

Two queuing policies, FIFO_Queuing and Priority_Queuing, are language defined. If no `Queuing_Policy` pragma appears in any of the program units comprising the partition, the queuing policy for that partition is FIFO_Queuing. The rules for this policy are specified in 9.5.3 and 9.7.1.

Compare this to the cases of omitting the pragmas `Task_Dispatching_Policy` and `Locking_Policy`. In the former, the dispatching policy is unspecified (arbitrary) whereas in the latter, locking is implementation-defined. Here, now, entry queuing is FIFO_Queuing, the Ada 83 default. Additionally, this does not only affect queuing. It also means that the choice of open alternatives of a selective accept is arbitrary (Ada 83 style).

The Priority_Queuing policy is defined as follows:

- The calls to an entry (including a member of an entry family) are queued in an order consistent with the priorities of the calls. The priority of an entry call is

[6] remember that FIFO_Within_Priorities and Ceiling_Locking must be used in conjunction

[7] In addition, priority based scheduling and dispatching is facilitated by the use of priority ordered entry queues, for example.

initialized from the active priority of the calling task at the time the call is made, but can change later. Within the same priority, the order is consistent with the calling (or requeuing, or priority setting) time (that is, a FIFO order).

Corresponding to FIFO_Within_Priorities for task scheduling and dispatching—this is reasonable since, after all, we are dealing with queues as well here.

- After a call is first queued, changes to the active priority of a task do not affect the priority of the call, unless the base priority of the task is set.
- When the base priority of a task is set (see D.5), if the task is blocked on an entry call, and the call is queued, the priority of the call is updated to the new active priority of the calling task. This causes the call to be removed from and then reinserted in the queue at the new active priority.
- When more than one condition of an entry_barrier of a protected object becomes `True`, and more than one of the respective queues is nonempty, the call with the highest priority is selected. If more than one such call has the same priority, the call that is queued on the entry whose declaration is first in textual order in the protected_definition is selected. For members of the same entry family, the one with the lower family index is selected.
- If the expiration time of two or more open delay_alternatives is the same and no other accept_alternatives are open, the sequence_of_statements of the delay_alternative that is first in textual order in the selective_accept is executed.
- When more than one alternative of a selective_accept is open and has queued calls, an alternative whose queue has the highest-priority call at its head is selected. If two or more open alternatives have equal-priority queued calls, then a call on the entry in the accept_alternative that is first in textual order in the selective_accept is selected.

The last three items are essentially the heart of the priority queuing as they provide for what is desired by the developer using this queuing scheme. Note that the issue regarding the textual order could be understood as *FWFS*—first written first served—thus fitting smoothly into the FIFO_Within_Priorities and the like concepts.

As with the other policies, an implementation is permitted to define additional queuing disciplines.

Neglected Topics

We have identified, at the beginning of this chapter, two major concerns that are pertinent to realtime systems: meeting deadlines (including specifying such) and (as a prerequisite for the former) scheduling and dispatching. There are, of course, other important aspects such as reliability, fault tolerance, robustness, efficiency, careful design, etc. We have not addressed them here since they are not exclusively bound to realtime systems. They are important for "normal" programs, too. We, furthermore, believe that the general (not necessarily concurrent) facilities of the Ada programming language are adequate for coping with these requirements.

Note that we are done now with scheduling and dispatching. The next subsection will deal with deadline-related issues and for the remainder of this subsubsection, we briefly

discuss further provisions made by the Realtime Systems Annex of Ada. For complete reference, we encourage the reader to consult that annex.

Interrupts: To be precise, Ada's interrupt handling facilities are not defined in Annex D but in Annex C, Systems Programming. But since an Annex D complying implementation is required to support Annex C as well, this is immaterial here. Interrupt handling, though not solely bound to realtime systems, is of significant importance since a realtime system will often consist of external devices, such as sensors, connected to computers. These devices, then, typically create interrupts upon which the realtime systems is required to react. Task entries and protected procedures can be used as interrupt handlers, they can be given a special interrupt priority (which is higher than any active priority of a "normal" task), and implementations are encouraged to execute the code of the interrupt handler by utilizing special hardware features (such as special call protocols for interrupts). It is beyond the scope of this chapter (even of this paper) to discuss Ada's interrupt handling facilities in full detail. The inclined reader is referred to Annex C.

Tasking Restrictions: These define restrictions (via the pragma `Restrictions`) that can be used to construct highly efficient tailored tasking runtime systems. For example, if a realtime system requires the number of tasks to be limited, the restriction `Max_Tasks` can be used, or if no abort statements are to be allowed, `No_Abort_Statements` can be used. More restrictions are defined in Annex D.

Synchronous/Asynchronous Task Control: This was discussed in Chapter 5.

Dynamic Priorities: These were briefly mentioned in Chapter 1.

6.3.2 Deadlines and Other Time-Related Issues

It has been mentioned several times already in this chapter that the concept of time is of paramount importance in realtime systems. We will, in this subsection, consider the following topics and analyze how Ada supports them:

- Access to a clock so that the passage of time can be measured
- Delaying a task until some future time
- Timeouts
- Temporal arrangements
- Runtime estimation (not a priori as part of the schedulability analysis but by asking the runtime system to do so (i.e., at runtime))

Access to a clock: Ada gives access to a hardware clock "by providing two packages. The main section of the ARM (Ada Reference Manual) defines a compulsory library package `Ada.Calendar` that provides an abstraction for a "wall clock" time that recognises leap years, leap seconds, and other adjustments. In the Real-Time Systems Annex, a second representation is given that defines a monotonic (that is, non-decreasing) regular clock (package `Ada.Real_Time`). Both these representations should map down to the same hardware clock but cater for different application needs. ... The current time is returned by the function `Ada.Calendar.Clock`. ... In addition, some arithmetic and boolean operations are specified." (Burns, 1998, [p. 34]) The package `Ada.Real_Time` is structurally equivalent to `Ada.Calendar`; it defines additional requirements for the realtime clock such as: there shall be no backward jumps, the amount of time that is allowed to elapse during which consecutive calls to `Ada.Real_Time.Clock` deliver the same result (the "tick", which is required to be no greater than one millisecond), or the range of time supported—at least fifty years. For a more complete description, we refer the reader to the Realtime Systems Annex.

Note that when doing time measurements of the form

```
-- assuming Ada.Calendar is with'ed and use'd
declare
   Before, After : Time;
   Elapsed : Duration;
begin
   Before := Clock;
   Action;
   After := Clock;
   Elapsed := After - Before;
end;
```

`Elapsed` will hold the amount of time it took `Action` to execute plus the amount of time (if any) the executing task was descheduled between the calls to `Clock`. That is, we can only measure the "wall time" that has elapsed between the calls to `Clock`.

Delaying a task: This can be achieved by using a delay statement (either a relative or an absolute one). Executing a delay simply blocks the task for the specified duration or, in case an absolute delay is employed, until the future time specified. It must be understood, however, that delay is only an approximate time construct. It only guarantees that the executing task, `T`, will be delayed for (or until) **at least** the time given. For when a higher priority task, `S`, is executing at the time of the wakeup, `S` takes precedence (remember, expiration of a delay is a scheduling event/task dispatching point) over the woken up task `T`. "The task executing a delay_statement is blocked until the expiration time is reached, at which point it becomes ready again." (ARM, 1995, [9.6]) Thus, `T` is only inserted into the ready queue for its active priority (by the scheduler)—there is no guarantee for it to be actually dispatched to a processor. Therefore, the execution of `T`'s body might be stalled considerably longer than intended. The task is ready (it is no longer blocked since the delay has expired) but since it is inserted at the tail of its ready queue and since there might be other tasks "in front of" it, it might not be able to get hold of a processor immediately. Given this, `T` becomes overdue. An overdue task misses at least one of its deadlines. The delay cannot be less than the expiration time specified, however.

"Real-time applications require that a task of sufficiently high priority be able to delay itself for a period of time with the assurance that it will resume execution immediately when the delay expires—i.e., that the duration of the interval between the start of the delay and the time the task resumes execution must be equal to the requested duration, within a predictable tolerance.

[RM95 9.6] only requires that execution of the task that executes the delay be blocked for at least the duration specified. It is not, in general, possible to require an upper bound on the duration of the execution of any statement, due to possible interleaved operations of other tasks on the same processor. However, it is both possible and necessary to have an upper bound on the duration of the interval between the start of a delay and the time the expiration of the delay is detected. It is also possible to guarantee that if the task whose delay has expired has higher priority than all the other tasks, it will resume execution as soon as the expiration of the delay is detected." (Rat, 1995, [D.9])

Note that an implementation is required (ARM, 1995, [D.9[12, 13]]) to document, among other things,

---

- An upper bound on the *lateness* of a delay_relative_statement, for a positive value of the delay expression, in a situation where the task has sufficient priority to preempt the processor as soon as it becomes ready, and does not need to wait for any other execution resources. The upper bound is expressed as a function

of the value of the delay expression. The lateness is obtained by subtracting the value of the delay expression from the actual duration. The actual duration is measured from a point immediately before a task executes the delay_statement to a point immediately after the task resumes execution following this statement.

- An upper bound on the *lateness* of a delay_until_statement, in a situation where the value of the requested expiration time is after the time the task begins executing the statement, the task has sufficient priority to preempt the processor as soon as it becomes ready, and it does not need to wait for any other execution resources. The upper bound is expressed as a function of the difference between the requested expiration time and the clock value at the time the statement begins execution. The lateness of a delay_until_statement is obtained by subtracting the requested expiration time from the real time that the task resumes execution following this statement.

Given the bare definition of the verb *preempt*, "preempt the processor" is possible. But, in our context, we would rather think of "one actor preempting another".

Of course, if the woken up task does not have sufficient priority and there is a higher priority task competing for the processor, the documented lateness is of little value. The programmer, therefore, has to ensure that hard-deadline-actors are appropriately represented (i.e., they must bear a distinctively high priority).

But even this is not entirely foolproof. We can, of course, imagine a "realtime system" in which the number of hard-deadline-actors exceeds the number of processors available (we can, in fact, restrict ourselves to a single processor). Now suppose that all hard-deadline-actors are designed by careful application of realtime facilities. Since the number of these actors is conceivably larger than the range of priority values supported, it is inevitable that several (perhaps all) tasks share the same priority (let it be sufficiently high). Now assume that all hard-deadline-actors wake up at once. Since there is only one processor (or, there are too few), contention results. Since several hard-deadline-actors have the same highest priority, competition is amongst them. As the winner of this competition is uniquely identified (it is the task at the head of the highest priority ready queue, which, of course, is unique), exactly one task is dispatched. All other tasks are held in the ready queue and are thus delayed longer than intended. This, albeit somewhat contrived, discussion can clearly be generalized to any positive number of processors less than the number of hard-deadline-actors. It indicates that the one of the preliminary steps in developing a realtime system has not been carried out with due attention—a priori schedulability analysis. If we cannot guarantee schedulability, we must change the system design. After all, we are possibly talking about people's lives. This is yet another indication for the notorious complications that arise in connection with realtime systems design and implementation. Among researchers engaged in the study of realtime systems, it is unanimously agreed that it is the stringent time constraints that set realtime system apart from "normal" computation systems.

Timeouts: Timeouts are used so that the non-occurrence of some event can be recognized and reacted upon. As an example, reconsider the DBMS server from Chapter 4 which was to reorganize the database if no requests were arriving within a certain period of time. Timeouts in Ada are programmed by using a select statement with a delay alternative (or else part, for immediate timeout) and a timed (or conditional) entry call for, respectively, a server and a client. The semantics and examples were already given in Chapter 4.

Temporal arrangements: By temporal arrangements, we mean that it be possible to attach timing constraints to an actor. For example, when an actor is being defined, it should be possible to state (as part of the definition) timing requirements such as start time/ event, periodicity, the deadline—the time by which this actor's execution must be finished, minimum and maximum delay before start, maximum execution time, and the like. This list

is based on that given in (Burns, 1989, [p. 334]). A notion that has emerged in the literature for an actor with associated time constraints is *temporal scope*. "Such scopes identify a collection of statements with an associated timing constraint." (Burns, 1996, [p. 378]) An apparent benefit of a temporal scope is that a realtime systems developer is given assistance in the process of *white box testing*. This process might then even be done using tool support.

By taking into account that an actor has a deadline, the scheduler can, although not in the general case, prefer this actor by, for example, increasing its priority as its deadline becomes closer and closer. This is called *shortest deadline first* scheduling. *Rate monotonic* scheduling could be used to adjust an actor's priority based on its periodicity.

Unfortunately, Ada does not support any deadline specification or any other temporal arrangement specification (by the way, none of our languages provides support for this; but PEARL does, see the last section of this chapter). A task (type) definition does not enable a programmer to state any timing constraints for the task being defined. Hence, the scheduler cannot be aware of deadlines at all!

The good news is that they can be quite easily programmed using Ada's facilities. But be aware, deadline handling is generally (i.e., whether or not temporal scopes are supported by a programming language) complicated by the risk of actors being delayed for too long. Also, we now thrust into the topic of *expressive power* versus *ease of use* of language provided constructs. "If a language does not support a particular notion or concept, those that use the language cannot apply that notion and may even be totally unaware of its existence" outside the language. (Burns, 1998, [p. 21]) If such a concept is needed, then workarounds must be used to achieve *expressive power* but often at the expense of *ease of use*.

Following is a list of how the various timing requirements can be effectively simulated in Ada.

Start time/event: This is somewhat complicated be the fact that Ada tasks are started by the scope rules of the language (see Chapter 1) and the programmer is given only indirect control. However, to simulate, a task is started as usual and then waits for either of the conditions. A delay can be used for the "start" time and an accept statement for an "event". To wait for either of these two conditions, a selective accept with a delay alternative is used:

```
select
   accept Start_Event;
or
   delay until Start_Time;
end select;
```

This code fragment is sensibly put at the very beginning of a task body. Another possibility is to create actors dynamically (using access values and the `new` allocator). As we recall from Chapter 1, such actors are activated immediately (more precisely, before the access value is returned). The programmer is thus given more control. A bit of a problem might be the fact that creation and start are interwoven (what if creating an actor takes too long an amount of time; what if the runtime system is, owing to a requirement in the specification of the realtime system, not allowed to created tasks dynamically, ...). For an absolute start time, the "delay until" is clearly more beneficial (Ada 83 had been much criticized for its lack of an absolute delay since using a relative delay, in this case, is cumbersome).

Periodic tasks/activities: The naive approach

```
Period : constant Duration := ...;

loop
   Some_Action;
   delay Period;
end loop;
```

is not adequate for `Some_Action` to be executed every `Period` seconds. First, the time to execute `Some_Action` is neglected and second, we have not considered the time it takes to execute the loop jump and to evaluate `Period`. As a consequence, between two consecutive executions of `Some_Action`, there might be a delay considerably longer than intended. It would be appropriate to suspend the task only for the "rest" of the period:

```
declare
   Next : Time;
   Period : constant Duration := ...;
begin
   Next := Clock + Period;
   loop
      Some_Action;
      delay Next - Clock;
      Next := Next + Period;
   end loop;
end;
```

compensates for that but is not entirely satisfactory as it introduces a race condition; between evaluating `Next - Clock` and starting the delay, there is the risk of the executing task being preempted by a (higher priority) task. It would be desirable to ensure that evaluating the expiration time and starting the delay are indivisible. This can be accomplished by using a "delay until' statement:

```
declare
   Interval : constant Duration := ...;
   Next_Time : Time := Clock;
begin
   Next_Time := Next_Time + Interval;
   loop
      Action;
      dealy until Next_Time;
      Next_Time := Next_Time + Interval;
   end loop;
end;
```

The deadline/maximum execution time: Of course, a task's execution can be forced to come to an abrupt end (by a certain time) by having a watchdog abort the task. This, however, is not really what we want for it is often way too disruptive and heavy-weight. Furthermore, it does not at all address the case where only a fragment of the code of the task has to meet the deadline. There is a cleaner way in Ada to achieve this: ATC using a delay trigger (an absolute one for a deadline and a relative one for maximum execution time):

```
select
   delay until Deadline; -- or delay Max_Execution_Time_Span
then abort
   -- code of the task
end select;
```

In essence, we have programmed a time supervision.

Minimum/maximum delay before the start of a task/action: Minimum delay is trivial and is guaranteed but maximum delay is not (neither trivial nor guaranteed) for reasons already discussed. It is in the programmer's hands, by a priori schedulability analysis, carefully

applying Ada's facilities, and scrutinizing the documented lateness of delay, to assure this. However, as stated above, this is generally the case with any realtime system written in any language.

For completeness, we note that for all timing considerations, both `Ada.Calendar`-time and `Ada.Real_Time`-time can be used.

Runtime estimation: As stated above, "it is not, in general, possible to require an upper bound on the duration of the execution of any statement, due to possible interleaved operations of other tasks on the same processor". This, again, applies to any realtime system written in any language.

"In summary, realtime developers analyze two quantities to demonstrate compliance with realtime constraints: execution time and blocking time. Performing accurate analysis is quite difficult. Consider the analogy of someone attempting to predict how much time will be required to drive a car across a small town. The "execution time" is how long it takes to drive from the starting point to the ending point assuming there are no delays along the way. If the car gets a flat tire or experiences mechanical difficulties, the times required to respond to these problems must be included in the worst-case execution time. But the typical execution time ignores these possibilities. The person's "blocking time" would be the maximum amount of time needed to wait for red lights at intersections, for railroad crossings, for traffic jams, and for coordination with emergency vehicles that are granted priority access to the public roadways." (Kelvin Nilsen, *"Adding Real-Time Capabilities to Java"*; in *Communications of the ACM*, June 1998, Vol. 41, No. 6) Of course, runtime is the sum of execution time and blocking time.

We conclude this subsection by considering a more elaborate example that will, after the somewhat pessimistic discussion, provide a bit of relief (we hope so) and that tries to combine all timing constraints mentioned above. This example can be found in (Burns, 1989, [p. 345]).

Consider the important realtime activity of making and drinking instant coffee:

```
Get_Cup
Put_Coffee_In_Cup
Boil_Water
Put_Water_In_Cup
Drink_Coffee
Replace_Cup
```

The act of making a cup of coffee should take no more than ten minutes; drinking is more complicated. A delay of three minutes should ensure that the mouth is not burnt; the cup itself should be emptied within 25 minutes (it would then be cold) or before 17:00 (in other words, 5 o'clock and time to go home). Two temporal scopes are required:

```
start elapse 10 do
   Get_Cup
   Put_Coffee_In_Cup
   Boil_Water
   Put_Water_In_Cup
end

start after 3 elapse 25 by 17:00 do
   Drink_Coffee
   Replace_Cup
end
```

For a temporal scope that is executed repetitively, a time loop construct is useful:

```
from <start> to <end> every <period>
```

For example, many software engineers require regular coffee throughout the working day:

```
from 9:00 to 16:15 every 45 do
   Make_And_Drink_Coffee
```

where Make_And_Drink_Coffee could be made up of the two temporal scopes given above (minus the by constraint on the drinking block). Note that if this were done, the maximum elapse time for each iteration of the loop would be 35 minutes; correctly, less than the period of the loop. An implementation of this scheme is given in Example 6.1.

```
task type Software_Engineer;

task body Software_Engineer is

   Now : constant Time := Clock;
   Time_To_Get_Started : constant Time := Time_Of(
     Year => Year(Now),
     Month => Month(Now),
     Day => Day(Now),
     Seconds => Day_Duration(9 * 3_600));  -- 9 a.m.
   Let_Us_Call_It_A_Day : constant Time := Time_Of(
     Year => Year(Now),
     Month => Month(Now),
     Day => Day(Now),
     Seconds => Day_Duration(17 * 3_600));  -- tea time

begin
   delay until Time_To_Get_Started;
   select
      delay until Let_Us_Call_It_A_Day;
   then abort
      Next_Time := Clock + 45 * Minutes;
      loop
         Coffee_Ready := False;
         select
            delay 10 * Minutes;
         then abort
            Get_Cup;
            Put_Coffee_In_Cup;
            Boil_Water;
            Put_Water_In_Cup;
            Coffee_Ready := True;
         end select;

         if Coffee_Ready then
            delay 3 * Minutes;
            select -- nested ATC
               delay 25 * Minutes;
            then abort
```

```
                Drink_Coffee;
                Replace_Cup;
             end select;
          end if;

          delay until Next_Time;
          Next_Time := Next_Time + 45 * Minutes;
       end loop;
    end select;

end Software_Engineer;

My_Company : array(1 .. 10) of Software_Engineer;
```

Example 6.1: The life of a software engineer

Note the ease of writing temporal scopes at the beginning of our example and compare this with the effort needed to realize that expressive power in Ada.

6.4 Elements for Realtime Programming in CHILL

CHILL, too, is aimed at addressing the field of realtime systems. This should not seem a surprise—telecommunication systems are yet another example of realtime systems. Unlike Ada, CHILL consists only of the core language defined in (Z200, 1996), there are no specialized needs annexes (but there is quite large a set of implementation-defined features as we shall see in a moment). CHILL was designed to, among other things, "cater for realtime applications by providing built-in concurrency and time supervision primitives". (Z200, 1996, [1.1]) Incidentally, here, once again, evidence is given for concurrency being a prerequisite for realtime issues.

6.4.1 Priorities, Scheduling and Dispatching

Unfortunately, when it comes to priorities, CHILL is unhelpful. It is not possible for a programmer to specify that a priority be associated with a task or process. Exceptions arise in the special cases of attaching a priority to a thread that is to be delayed on an event, signal, or buffer (see chapters 4 and 5) and when issuing a call to one of a task's simple guarded procedures. But these are no real exceptions for they do not allow (or, at most, only indirectly) these "priorities" to be used for preference specification in processor contention. Furthermore, the priorities cease to be associated with the thread as soon as their context (delaying and reactivation on an event, signal, or buffer, and selection of an outstanding call on a task component) ceases to exist. Thus, priorities are not permanently but only transiently associated with threads (and, of course, as it depends on the flow of control whether or not the above mentioned contexts will ever be entered, not even this can be taken for granted). In other words: priorities do, in general, not exist in CHILL. Thus, although we might be able to, during systems specification and design, identify hard-deadline-actors and soft-deadline-actors, it is impossible to map the intended preference of the hard-deadline-actors onto the actors provided by the language.

Another drawback in CHILL is scheduling and dispatching. All that is said about it in the definition is that it is implementation-defined (Z200, 1996, [13.7]). At least, it is not unspecified. A realtime systems developer has, therefore, to study the vendor's documentation of the actual algorithms used. Note that implementation-defined is not as bad as it might seem at first glance. Portability is, for obvious reasons stemming from the very

nature of realtime systems, not (cannot be!) an overriding concern. It is to be hoped (and there is conviction as several hard realtime systems, written in CHILL, have been successfully employed) that scheduling and dispatching algorithms appropriate to realtime systems' concerns are supported by an implementation.

6.4.2 Deadlines and Other Time-Related Issues

CHILL uses the notion of *time supervision* for what we are concerned with in this subsubsection. Chapter 9 of (Z200, 1996) is dedicated to this topic; so all quotations in the sequel originate from this chapter.

Access to a clock: CHILL provides access to a clock by means of the absolute time built-in routine call `ABSTIME`, the syntax and semantics of which are as follows:

```
<absolute time built-in routine call>::=
  ABSTIME([[[[[[<year expression>,]
    <month expression>,]
    <day expression>,]
    <hour expression>,]
    <minute expression>,]
    <second expression>])
```

Each of the above expressions shall be an integer expression.

The `ABSTIME` built-in routine call delivers an absolute time value denoting the point in time in the Gregorian calendar indicated in the parameter list. The parameters indicate the components of time in the following order: the year, the month, the day, the hour, the minute, and the second. When higher order parameters are omitted, the point in time indicated is the next one that matches the low order parameters present (e.g., `ABSTIME (15, 12, 00, 00)` denotes noon on the 15th in this or the next month. When no parameters are specified, an absolute time value denoting the present point in time is delivered.

Thus, declaring two time mode locations, we can undertake simple time measurements:

```
DCL
   Before, After TIME;
   Elapsed DURATION;

   Before := ABSTIME();
   Action();
   After := ABSTIME();
```

Note that `Elapsed := After - Before` is not allowed in CHILL since arithmetic operators are not defined on values of mode `TIME` (the relational operators, interestingly, are defined on them).

Note further that `ABSTIME` only gives access to a wall clock; it is not possible to directly determine the amount of time `Action()` kept the processor busy. Additionally, this clock might be subject to daylight savings or human operator adjustments.

Delaying a thread: For this purpose, we need to study CHILL's notion of time supervision more elaborately:

It is assumed that a concept of time exists externally to a CHILL program (system). CHILL does not specify the precise properties of time, but provides mechanisms to enable a program to interact with the external world's view of time.

The concept of a *timeoutable* process exists in order to identify the precise points during program execution where a time interrupt may occur, that is, when a time supervision may interfere with the normal execution of a process. A process becomes timeoutable when it reaches a well-defined point in the execution of certain actions. CHILL defines a process to become timeoutable during the execution of specific actions; an implementation may define a process to become timeoutable during the execution of further actions.

<timing action>::=
  <relative timing action>
  | <absolute timing action>
  | <cyclic timing action>

A timing action specifies time supervisions of the executing process. A time supervision may be initiated, it may expire, and it may cease to exist. Several time supervisions may be associated with a single process because of the cyclic timing action and because a timing action can itself contain other actions whose execution can initiate time supervisions. A time interrupt occurs when a process is timeoutable and at least one of its associated time supervisions has expired. The occurrence of a time interrupt implies that the first expired time supervision ceases to exist; furthermore, it leads to the transfer of control associated with that time supervision in the supervised process. If the supervised process was delayed, it becomes re-activated. Time supervisions also cease to exist when control leaves the timing action that initiated them. Note that if the transfer of control causes the process to leave a region, the region will be released (see Section 11.2.1).

<relative timing action>::=
  AFTER <duration primitive value> [DELAY] IN
  <action statement list>
  <timing handler> END

<timing handler>::=
  TIMEOUT <action statement list>

The duration primitive value is evaluated, a time supervision is initiated, and then the action statement list is entered. If `DELAY` is specified, the time supervision is initiated when the executing process becomes timeoutable at the point of execution specified by the action statement in the action statement list; otherwise, it is initiated before the action statement list is entered. If `DELAY` is specified, the time supervision ceases to exist if it has been initiated and the executing process ceases to be timeoutable. The time supervision expires if it has not ceased to exist when the specified period of time has elapsed since initiation. The transfer of control associated with the time supervision is to the action statement list of the timing handler. If `DELAY` is specified, the action statement list must consist of precisely one action statement that may itself cause the executing process to become timeoutable.

<absolute timing action>::=
  AT <absolute time primitive value> IN
  <action statement list>
  <timing handler> END

The absolute time primitive value is evaluated, a time supervision is initiated, and then the action statement list is entered. The time supervision expires if it has not ceased to exist at (or after) the specified point in time. The transfer of control associated with the time supervision is to the action statement list of the timing handler.

```
<cyclic timing action>::=
  CYCLE <duration primitive value> IN
  <action statement list> END
```

The cyclic timing action is intended to ensure that the executing process enters the action statement list at precise intervals without cumulated drifts (this implies that the execution time for the action statement list on average should be less than the specified duration value). The duration primitive value is evaluated, a relative time supervision is initiated, and then the action statement list is entered. The time supervision expires if it has not ceased to exist when the specified period of time has elapsed since initiation. Indivisibly with the expiration, a new time supervision with the same duration value is initiated. The transfer of control associated with the time supervision is to the beginning of the action statement list. Note that the cyclic timing action can only terminate by a transfer of control out of it. The executing process becomes timeoutable if and when control reaches the end of the action statement list.

```
<duration built-in routine call>::=
  MILLISECS(<integer expression>)
  | SECS(<integer expression>)
  | MINUTES(<integer expression>)
  | HOURS(<integer expression>)
  | DAYS(<integer expression>)
```

A duration built-in routine call delivers a duration value with implementation defined and possibly varying precision (i.e., `MILLISECS(1000)` and `SECS(1)` may deliver different duration values); this value is the closest approximation in the chosen precision to the indicated period of time. The argument of `MILLISECS`, `SECS`, `MINUTES`, `HOURS`, and `DAYS` indicate a point in time expressed in milliseconds, seconds, minutes, hours, and days respectively.

With the help of timing built-in routine calls, a delay can be easily achieved. But first things first:

```
<timing simple built-in routine call>::=
  WAIT()
  | EXPIRED()
  | INTTIME(<absolute time primitive value>,
    [[[[<year location>
    <month location>,]
    <day location>,]
    <hour location>,]
    <minute location>,]
    <second location>)
```

Each of the above mentioned locations shall be an integer location.
`WAIT` unconditionally makes the executing process timeoutable: its execution can only terminate by a time interrupt. (Note that the process remains active in the CHILL sense). `EXPIRED` makes the executing process timeoutable if one of its associated time supervisions

has expired; otherwise, it has no effect. INTTIME assigns to the specified integer locations an integer representation of the point in time in the Gregorian calendar specified by the absolute time primitive value. The locations passed as arguments receive the components of time in the following order: the year, the month, the day, the hour, the minute, and the second.

Note that there is a somewhat ambiguous formulation regarding WAIT, "WAIT unconditionally makes the executing process timeoutable: its execution can only terminate by a time interrupt". Whose execution can only terminate by a time interrupt? The process' or WAIT's? We **assume** WAIT's.

With these tools in hand, a relative delay is just as easy as

```
AFTER SECS(5) IN
   WAIT();
TIMEOUT
   -- Statements_After_The_Break
END;
```

while an absolute delay is expressed using

```
AT ABSTIME(12, 24, 18, 00, 00) IN  -- X-mas!
   WAIT();
TIMEOUT
   Deliver_Packages();
END;
```

Timeouts: These can be specified by employing essentially the same mechanism used in the specification of delays. We simply replace WAIT by the event we would like to supervise with a timeout. Not that it is not straightforward to impose a timeout on a call to a task component owing to the asynchronous nature of such a call.

Temporal arrangements: As already briefly mentioned, this is not feasible in CHILL but, again, the good news is that deadlines can be programmed; again, of course, with the same caveats concerning expressive power versus ease of use (see the corresponding Ada part). We briefly touch the various deadline-related topics:

Start time/event: For processes, this is easy. A programmer has full control over when and under what preconditions, for the event, to issue a START expression. This is straightforward. For a task, the scope rules of the language determine when and if a task is started. Of course, by deferring the dynamic creation of a task object or by making its creation dependent upon some event, the intended effect can be simulated. But here again, creation and start are interwoven so that the same caveats as in the case of Ada can be attributed to this approach. Additionally, creating a task object simply enables it to receive requests. A CHILL task object does not execute voluntarily—see Chapter 1.

Periodic threads: The cyclic timing action is clearly the candidate for this type of application for it provides essentially what is needed. Note that no explicit loop is required as the transfer of control caters for the turnaround. Perhaps a bit of a flaw, the cyclic timing action allows only a duration primitive value to be specified—that is, the action statement list is only entered at intervals relative to a given start point. Thus, for example, if a process is to perform some action each day at noon, this must be programmed directly:

```
DO FOR EVER;
   AT Noon IN
      WAIT();
   TIMEOUT
      Action();
   END;
OD;
```

Note that if a task object is to execute a cyclic timing action, a trigger, i.e., a call from a client, is required.

The deadline/maximum execution time: Clearly, the absolute timing action is appropriate for the former whereas the relative one is for the latter. Straightforward.

Minimum/maximum delay before start of a thread/action: Again, minimum delay is trivial and guaranteed but, again, maximum delay is not (neither trivial nor guaranteed) for, again, reasons already discussed.

Runtime estimation: Again, the pessimistic statement made earlier regarding this issue applies to CHILL as well—runtime estimation is complicated by the fact that both execution time *and* blocking time must be taken into account.

It would be tedious to repeat the example of coffee making here. In fact, the tools described in this subsubsection allow for easy realization. It would merely be a paraphrasing from Ada to CHILL with some minor changes. It thus provides no further insight and is left as an exercise (the only one in this paper) for the reader.

6.5 Elements for Realtime Programming in Java

"Java was originally called Oak, and designed for use in embedded consumer-electronic applications by James Gosling. After several years of experience with the language, and significant contributions by Ed Frank, Patrick Naughton, Jonathan Payne, and Chris Warth, it was retargeted to the Internet, renamed Java, and substantially revised to be the language specified here." (JLS, 1996, [Preface])

The revision process—besides contributing Internet capabilities—must have removed many of the realtime facilities that we assume were present in Oak. As we shall see in a moment, Java is not suitable for the development of realtime systems (well, it is—as we just have learnt—not aimed at realtime systems). We might as well skip Java and turn our attention to PEARL instead. But for the sake of completeness (and for further insight, hopefully), let us examine our list of topics with respect to Java.

6.5.1 Priorities, Scheduling and Dispatching

As we have seen in Chapter 1, priorities and the usual operations on them are available in Java. Ten values constitute the range of priorities supported. Java's notion of a priority is as follows:

"Every thread has a *priority*. When there is competition for processing resources, threads with higher priority are generally executed in preference to threads with lower priority. Such preference is not, however, a guarantee that the highest priority thread will always be running, and thread priorities cannot be used to reliably implement mutual exclusion."

This is quite vague a description for it weakens the impact of priorities considerably. For example, if an implementation chose to dispatch *any* thread—a malicious scheduler/dispatcher—whenever there is competition, then this implementation would comply to (JLS, 1996) (note that dispatching the lowest priority thread is acceptable if this is clearly documented). But this implementation would be of little benefit for realtime systems development. Note that *generally* is not *always* and that Java cannot guarantee that, for each processor, the currently running thread is one of the threads with the highest priority. Priority inversion is thus possible, that is, Java provides no means for the prevention/bounding of this detrimental phenomenon. Not only are Java's priorities useless for realtime systems (can be ignored), but they are also of little avail for general applications of concurrency. Java's priorities beg the question. The problem, clearly, is that too much of the details are left unspecified by (JLS, 1996).

As a consequence, the impact of priorities on scheduling/dispatching is undefined (no wonder as, after all, the meaning of priorities is). Moreover, neither (JLS, 1996) nor (JVM,

1996) define the scheduling/dispatching model to be used (nor do they require an implementation to document the chosen approach). Hence, we must assume that it is *arbitrary*, which might be acceptable for general application of concurrency—but **not** for realtime systems. Of interest is the static method `java.lang.Thread.yield()` which causes the current thread to yield, allowing the thread scheduler to choose another runnable thread for execution.

These two issues impose severe restrictions upon Java being used for the development of realtime systems.

6.5.2 Deadlines and Other Time-Related Issues

The class `java.util.Date` provides a system-independent abstraction of dates and times, to a millisecond precision.

Access to a clock: Java gives access to a clock by the method `getTime()`, which is defined in `Date`:

```
public long getTime()
```

"This method returns the time represented by this `Date` object, represented as the distance, measured in milliseconds, of that time from the *epoch* (00:00:00 GMT on January 1, 1970)."

For convenience, several other methods like `getSeconds()`, `getMinutes()`, etc. are defined. To interrogate the *current* time, a call to `java.lang.System.currentTimeMillis()` can be used. It is essentially equivalent to `new Date().getTime()`. Note that Java's clock is a wall clock, too.

Delaying a thread: A thread can sleep for a while by executing one of the following methods:

```
public static void sleep(long millis) throws InterruptedException
```

"This method causes the current thread to yield and not to be scheduled for further execution until a certain amount of real time has elapsed, more or less.

The amount of real time, measured in milliseconds, is given by `millis`.

If the current thread is interrupted by another thread while it is waiting, then the sleep is ended and an `InterruptedException` is thrown.

```
public static void sleep(long millis, int nanos) throws
   InterruptedException
```

This method causes the current thread to yield and not to be scheduled for further execution until a certain amount of real time has elapsed, more or less.

The amount of real time, measured in nanoseconds, is given by:

```
1000000 * millis + nanos
```

In all other respects, this method does the same thing as the method `sleep` of one argument. In particular, `sleep(0, 0)` means the same thing as `sleep(0)`.

If the current thread is interrupted by another thread while it is waiting, then the sleep is ended and an `InterruptedException` is thrown."

Note that, since a thread may be interrupted anytime, Java cannot guarantee that a thread executing `sleep` is delayed for at least the duration given. On the other hand, an upper bound on the delay cannot be guaranteed either—but for reasons beyond Java (in fact, beyond every programming language as already discussed). The lower bound on the delay *could* be enforced by executing, in the exception handler corresponding to the

`InterruptedException`, a further delay if necessary. But since Java only supports relative delays, determining whether there is a necessity (for a further delay) is awkward and error-prone for there are race conditions.[8] For this purpose, an absolute delay would be more favourable.

Using `suspend/resume` for delaying purposes is rather bad practice for it requires a *watchdog* to issue the `resume` and this watchdog would be required to keep track of the wakeup time.

Timeouts: Java provides no language constructs that allow for the recognition of the non-occurrence of some event.

Temporal arrangements: Similarly to Ada and CHILL, temporal arrangements cannot be attached to a thread. Even worse, simulating the various kinds of temporal arrangements is complicated by the fact that not even minimum delay can be guaranteed. But let us consider our list:

Start time/event: For the event, this is fairly easy since starting a thread is completely subject to the whim of the programmer. In the case of start time, the drawback of `sleep` is clearly a disadvantage.

Periodic threads: We could not find a satisfactory solution. Influencing factors include: general lack of minimum delay, lack of absolute delay (so that, for example, cumulative drifts have to be eliminated by hand—this introduces a race condition: between determining the remaining time the thread must sleep and actually being put to sleep, the thread might be descheduled).

The deadline/maximum execution time: Java provides no constructs that allow for either to be programmed.

Minimum/maximum delay before start of an action: Minimum delay is trivial but not guaranteed, and maximum delay is neither trivial nor guaranteed (again, we cannot blame Java).

Runtime estimation: We can only repeat the statements regarding this issue made earlier in the Ada and CHILL sections.

Not only would it be tedious to repeat the coffee making example, but it is also—or, *first of all*—virtually impossible. As we were incapable to provide workarounds for the deadline issues, it is not clear how to achieve a satisfactory implementation. Of course, we could use a brute-force approach by heavily making use of watchdogs and other agent threads, but we are loath to adopt this strategy. In short, this approach is too low-level, suffers from race conditions, and baffles sound principles of software engineering. We do not encourage the reader to work it as an exercise!

Instead, we would like to draw the reader's attention to the following article, which was already mentioned in this chapter:

Kelvin Nilsen, *Adding Real-Time Capabilities to Java*; in *Communications of the ACM*, June 1998/Vol. 41, No. 6

Nilsen, in this article, mentions further aspects of Java that are disadvantageous for its use in the development of realtime systems. He provides hints what could be ameliorated and has also undertaken practical work in the design of PERC—a realtime variant of the Java language.

This article is worth reading.

This completes this section and our journey into the consideration of the realtime facilities supported by our three languages. The next section is devoted to a brief introduction

[8] We have assumed that this additional `sleep` is not interrupted—there is, however, no guarantee.

to PEARL. It mainly focuses on the two topics we have laid emphasis upon in the previous parts of this chapter: scheduling/dispatching and deadline issues.

6.6 PEARL 90

6.6.1 Priorities and Dispatching of Tasks

Each PEARL 90 task has a priority that is represented by a number in the range of 1 to 255. The number 1 corresponds to the highest priority whereas the number 255 corresponds to the lowest priority. If no priority is specified in the definition of a task T, then T implicitely gets the lowest priority (number 255).

Processors and other resources are allocated to tasks according to the priority of each task. Tasks with higher priorities take precedence over tasks with lower priorities. Tasks that have the same priority are treated according to the round robin strategy.

A new priority based allocation of the processor takes place each time one of the processor's operations concerning the operating system is executed. Such operations are, for instance, the occurrence of an interrupt, statements for controlling tasks, statements for synchronizing tasks, and input/output operations.

If there are more tasks than processors, the tasks have to compete for the processors. The same holds for other resources that are used by several tasks (for instance, input/output-devices). If there is only one processor, P, that is allocated to a task `T`, and `T` wants to access another system resource, then P is taken away from `T` and allocated to one task (from the set of tasks competing for P) with the highest priority.

6.6.2 Scheduling of Tasks

In PEARL 90, it is possible to schedule a given task to be executed at certain points in time. These points in time are explicitely specified or they are given by the occurrence of interrupts. This scheduling is done by a Task_Start statement, the syntax of which reads

```
Task_Start ::= [ Start_Condition ] ACTIVATE Task_Name [ Priority ];
Priority ::= {PRIORITY | PRIO} Number
Start_Condition ::= AT Time [ Frequency ]
    | AFTER Duration [ Frequency ]
    | WHEN Interrupt [ AFTER Duration ] [ Frequency ]
    | Frequency
Frequency ::= ALL Duration [ {UNTIL Time} | DURING Duration ]
```

The task given by Task_Name becomes runnable at the point in time specified by the Start_Condition. If the Start_Condition is omitted, then the given task becomes runnable immediately.

If AT Time is specified, then the task will become runnable at the point in time given by Time.

If AFTER Duration is specified, then the task will become runnable after the period of time given by Duration has expired.

If WHEN Interrupt is specified, then the task becomes runnable whenever the interrupt given by Interrupt occurs. Should AFTER Duration be specified as well, then the task becomes runnable after the interrupt given by Interrupt has occurred and then the period of time given by Duration has expired.

If AT, AFTER, and WHEN are not specified, then the task becomes runnable immediately.

Periodic execution of a task `T` can be achieved by specifying Frequency. ALL Duration defines the time between two activations of `T`.

If UNTIL Time is specified, then no further activation of T is possible after reaching the point in time given by Time.

If DURING Duration is specified, then no further activation of T is possible after the period of time given by Duration has elapsed.

The Start_Condition has no effect (and the associated task T is not activated) if

- the Start_Condition contains UNTIL Time and the point in time given by Time has already been reached
- the Start_Condition contains DURING Duration and the period of time given by Duration has already elapsed
- the Start_Condition has the form AFTER Duration and the period of time given by Duration has already elapsed
- a new Task_Start statement concerning T is executed and the new Start_Condition replaces the old one
- a PREVENT statement with T as Task_Name is executed (see below).

The PREVENT statement is used to make a Start_Condition inoperative. The syntax is

PREVENT [Task_Name];

If Task_Name is not specified, then the PREVENT statement only effects the Start_Condition of the task executing this statement. The PREVENT statement has no effect on the current execution of the task given by Task_Name.

We conclude this section with some examples illustrating the use of the scheduling facilities of PEARL 90.

```
AT 20:0:0 ACTIVATE T1;
```
The task T1 will become runnable at the next time when it is twenty o'clock.

```
ACTIVATE T1;
```
T1 becomes runnable immediately.

```
ALL 2 HRS ACTIVATE T1;
```
T1 becomes runnable immediately and it will become runnable every two hours.

```
WHEN Alarm ACTIVATE T1;
```
Whenever the interrupt Alarm occurs, T1 becomes runnable.

Summary and Comparison

This chapter is special in that it tries to portray a field of concurrent programming that poses quite subtle problems and grand challenges to its adepts—the field of realtime programming. After a brief introductory explanation, two overriding concerns regarding realtime systems were identified: the aspect of stringent deadlines and scheduling/dispatching. We claim that it is these two requirements that actually set realtime systems apart from "normal" computation systems. A prerequisite for realtime concerns is concurrency as virtually all realtime systems are inherently concurrent. We then turned our attention to how our three languages support a developer in her/his strive to cope adequately with the above mentioned requirements. Note that only Ada and CHILL were designed with the issue of realtime systems in mind. Note further that in the general application of concurrency, there is no compelling need to be this much concerned with things like priorities, scheduling/dispatching,

stringent deadlines, or the number of processors available. But, as stated, realtime systems somehow are the grand challenges of concurrency for they incur all the "usual" problems of concurrency *and* introduce additional ones. A realtime systems developer has to deal with this bulk of additional problems. We have programming, we have concurrent programming, and then we have realtime programming.

Priorities, understood as a means for preference specification in resource contention, were considered first. Only Ada could be identified as having an appropriate priority model. CHILL generally lacks priorities while Java's priority model is not clearly defined and, thus, futile for the intended aim. Ada guarantees that for each processor, one of the highest priority tasks is the winner in resource contention and provides means to avoid/limit the effect of priority inversion. Neither CHILL nor Java do address either topic at all.

The priority model is expected to fit seamlessly into the mechanisms of *scheduling/ dispatching*—that is, the algorithms used for the latter should take advantage of the former. Again, only Ada can fulfill this requirement since it supports a priority based preemptive scheduling/dispatching policy, which warrants for higher priority tasks to be able to gain access to a processor whenever necessary by preempting lower priority tasks. This is regarded as being a fateful prerequisite for hard-deadline-actors to meet their deadlines. CHILL's scheduling/dispatching algorithm is implementation-defined, which, to some extent, is tolerable. The developer is required to scrutinize the compiler vendor's documentation and trace down the actual algorithm used. Java's approach, however—leaving the actual procedure unspecified—, is thwarting the analysis of schedulability of threads and, therefore, makes impossible the programmer's chore of determining whether all hard-deadline-actors a capable of meeting their deadlines.

Temporal arrangements, most notably deadlines, then, were identified as timing constraints that are associated with an actor (as part of its definition). Unfortunately, in none of our three languages, it it is feasible to associate deadlines with actors. A clear benefit of having actors with deadlines (*temporal scopes*) is that the scheduler/dispatcher can be made aware of the deadlines so as to try to prefer hard-deadline-actors by techniques such as *earliest deadline first scheduling* or *rate monotonic scheduling*. We mentioned various forms of timing constraints. In order to achieve *expressive power*, we tried to provide workarounds. As regards *ease of use*, Ada and CHILL posed only little problems since the language provided constructs allow for easy realization. Java lacks many of the needed (basic) constructs and, consequently, satisfactory solutions could not be found in all cases.

Java, in its current form, is neither targeted to nor appropriate for the development of realtime systems. As a consequence of the former, it falls short in virtually all aspects pertaining to this kind of concurrent programming. Ada 83, originally designed to—in particular—address the sector of embedded systems, had been much criticized for providing too few (or inappropriate) means to tackle the problems of realtime systems. Its successor, Ada 95, has been equipped with a rich set of additional tools that, at least, help us approach the desired aim. CHILL is as expressive a language as Ada and should be included into the consideration when it comes to the choice of the language to be used for the implementation.

Clearly, no one language is perfect—general purpose languages, the set of which all our three language must be regarded as being a member of, cannot address all application areas while special purpose languages are limited in their usability. PEARL is is an exception—a general purpose language designed to address the field of realtime systems development. From the brief introduction given in Section 6.6, it should be obvious that PEARL deals adequately with the requirements of realtime systems. It has an appropriate priority model that is taken advantage of by the scheduler/dispatcher and, how gratifying, supports the specification of deadlines—*expressive power* can be achieved with *ease of use* here.