PREV
12. Consensus

NEXT
A. The Pseudocode Notation

Chapter 13. Real-Time Systems

Introduction

A *reactive system* is one that runs continuously, receiving inputs from and sending outputs to hardware components. If a reactive system must operate according to strict constraints on its response time, it is called a *real-time system*. Modern computer systems are more likely than not to be reactive systems: a word processor can be considered to be a reactive system because it waits for input from hardware components (the keyboard and the mouse) and produces output on other hardware components (the screen and the hard disk). Nevertheless, the study of reactive systems in general, and real-time systems in particular, primarily concerns *embedded computer systems*, which have a computer as only one of a set of components that together comprise a system. Examples are: aircraft and spacecraft flight control systems, industrial plant controllers, medical imaging systems and monitors, and communications networks. Most of these systems are real-time systems, because given an input they must produce an output within a precisely-defined period of time, ranging from a few milliseconds for aerospace systems, to perhaps a few seconds for the controller of an industrial plant.

The difficulty in designing real-time systems lies in satisfying the strict requirement to meet processing deadlines. Here are two examples of challenging, complex computer systems which are *not* real-time:

- Simulation of the weather to develop a forecast. This requires the use of powerful supercomputers, but it is not a critical failure if the answers are received ten seconds or even ten minutes late.
- An airline reservation system. A large network of computers, databases and communications channels is needed to

We don't want to imply that these systems shouldn't be designed to meet their deadlines all the time; however, there can be some flexibility in satisfying response-time requirements if it would be prohibitively expensive to guarantee that all responses are produced within the target deadlines. A real-time system, on the other hand, need not be complex or high-performance. Checking the radioactivity levels in a nuclear power plant or monitoring the vital signs of a patient may require only the simplest processing, but a delay in decreasing the power level of the plant or in raising an alarm if the patient's vital signs deteriorate can literally be fatal.

This chapter surveys a range of topics concerning real-time systems, with emphasis on those related to synchronization issues. Detailed treatments of real-time systems can be found in the textbooks by Liu [47] and Burns and Wellings [19]. We start with a section that defines the temporal characteristics of real-time systems, followed by a description of two famous examples of problems that arose in real-time systems. Sections 13.4–13.6 present three different approaches to writing software for real-time systems: synchronous, asynchronous and interrupt driven. Almost invariably, some tasks performed by a real-time system are more important or more urgent than others. Priority assignments enable the designer to specify these constraints, but the interaction between priority and synchronization leads to a difficult problem called priority inversion. Section 13.7 discusses priority inversion, together with techniques to overcome it, priority inheritance and priority ceiling locking; Section 13.8 shows how priority inversion and inheritance can be modeled in Spin.

Real-time systems have requirements for moving data from processes reading sensors to processes computing algorithms and from there to processes controlling actuators. These are classical producer–consumer problems, but with the added requirement that blocking synchronization constructs like semaphores cannot be used. Section 13.9 presents Simpson's four-slot algorithm for ensuring wait-free atomic communication of data from a producer to a consumer.

The topic of languages and tools for specifying and implementing real-time systems is a current topic of research. Section 13.10 presents an overview of the Ravenscar profile, which is the specification of a subset of the Ada language designed for the performance requirements of real-time systems. Section 13.11 briefly introduces UPPAAL, a tool for verifying real-time algorithms specified as timed automata. The final section is a short introduction to scheduling algorithms.

Software problems occurring in spacecraft have a particularly high profile and are widely reported, and the danger and expense of exploring space ensure that such problems are carefully analyzed. We bring several examples of "bugs in space" that we hope will motivate the study of concurrency.

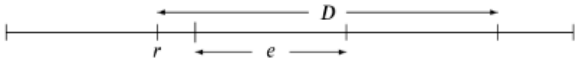
The Ada Real-Time Annex[L]

The Ada language specification contains several annexes that describe features of the language that go beyond a core common to all implementations. Throughout this chapter, we shall frequently mention the Real-Time Annex, which specifies constructs relevant to real-time programming. This makes Ada an excellent platform for building real-time systems, because real-time issues can be handled within the framework of a portable, standard language, rather than a specific operating system. For details, see [7, Chapter 16] and [19].

Definitions

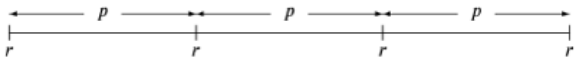
Processes in real-time systems are called *tasks*.

With each task in a real-time system is associated a set of parameters specifying its timing requirements. The *release time r* is the time when the task joins the ready queue of tasks ready to execute. The *execution time e* is the maximum duration of the execution of the task. The *response time* of a task is the duration of time from its release time *r* until it has completed its execution; the maximum response time allowed for a task is called its *(relative) deadline D*:

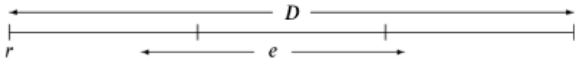


The response time for this execution of the task is measured from the release time *r* until the right end of the arrow for *e* indicating that the task is completed. Clearly, the response time is less than the deadline in this case.

Many tasks in a real-time system are *periodic*, meaning that there is a fixed interval between release times called the *period p*:



Real-time systems can be designed so that all tasks are released at the beginning of a period. A further simplification occurs when the deadline of a task is a multiple of its period, and if the periods of all tasks are multiples of the smallest period:



In addition to periodic tasks, a real-time system may have *sporadic* and *aperiodic* tasks which are released at arbitrary times. The difference between the two is that sporadic tasks have hard deadlines while aperiodic tasks have soft deadlines.

A task in a real-time system is categorized as having a *hard* or *soft* deadline, depending on the importance attached by the designer to meeting the deadline. (The terminology is extended so that a real-time system is hard if it contains tasks with hard deadlines.) The classification of a task is not simply a binary decision between hard and soft. Some tasks like those for flight control of an aircraft must meet their deadlines every time or almost every time. Of course it makes no sense to have a task that never need meet its deadline, but there are tasks like the self-test of a computer that can be significantly delayed without affecting the usefulness of a system. In the middle are tasks that can fail to meet their deadlines occasionally, or fail to meet them with a specified probability, for example, the task may fail to meet its deadline at most 5% of the time. An example would be the telemetry task of a spacecraft: an occasional delay in the transmission of telemetry data would not endanger the mission.

The designer of a real-time system has to decide what happens if a deadline is not met. Should the task continue executing, thereby delaying other tasks? Or should the task be terminated if it has not completed by its deadline (thereby freeing up CPU time for other tasks) because a late result is of no use?

Reliability and Repeatability

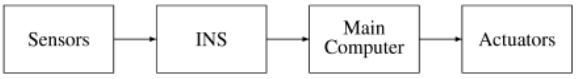
We have become tolerant of unreliable desktop software that crashes frequently. This attitude is simply unacceptable for embedded systems that control critical tasks, especially for systems

that cannot be debugged. I call a computer system for a rocket launcher a “disposable computer” because you run the software once and then “throw away” the computer! There is no analog to pressing `ctrl-alt-del`. Two famous examples highlight the problems involved. The first launch of the Ariane 5 rocket failed because of a chain of mistakes related to issues of testing, verification and reliability. The first launch of the space shuttle was delayed because of a minor problem that had never occurred in countless hours of testing.

The Ariane 5 Rocket

The first launch of the Ariane 5 rocket was carried out by the European Space Agency in 1996 from Kourou in Guyana. Thirty-seven seconds after launch, the rocket veered on its side and began to break up; it was then destroyed as a safety measure. Investigation revealed that the failure was due to problems with the software that controlled the rocket.

The Ariane rocket uses an *inertial navigation system (INS)* to sense the position of the rocket. Data from motion sensors are processed by the INS and relayed to the main computer which issues commands to actuators that control the nozzles of the engines:



The sequence of events that led to the destruction of the Ariane 5 was as follows. A runtime error occurred during a data conversion operation in the INS, causing it to terminate execution. The error also occurred on the backup computer since both were executing the same program. The error was reported to the main computer which erroneously interpreted it as legal data; the main computer then gave an extreme command to the actuators, causing the rocket to go off course and break up.

The root of the software problem was traced to a decision to reuse the INS of the earlier Ariane 4 rocket. The new, larger rocket had a different trajectory than the Ariane 4, and one sensor value could no longer fit into 16-bits. To save CPU time, the value to be converted from 64- to 16-bits was not checked prior to the conversion, nor was there any provision for exception handling more sophisticated than reporting the error, nor had the omission of the check been justified on physical grounds. Finally, the INS software was not revalidated for the Ariane 5 under the assumption that it was unchanged from the software that had been validated for the Ariane 4 and had worked for many years.

The decision not to revalidate the software was not a simple oversight. Embedded systems cannot be tested as such: you can’t launch thousands of rockets costing hundreds of millions of dollars each just to debug the software. Something has to be simulated somehow in a lab, and the better the fidelity of the simulation, the more it costs and the harder it is to perform. The difficult of testing software for embedded systems emphasizes the importance of both formal approaches to verification and the practices of software engineering. The full report of the investigating committee of the Ariane 5 failure has been published [45]; it is recommended reading for aspiring software engineers.

The Space Shuttle

Just before the first launch of the space shuttle, a fault occurred in the backup computer indicating that it could not receive data from the main computers. While the problem did not seem to be serious, the decision was made not to launch the new spacecraft as long as a

fault was known to exist. The flight was delayed for two days until the problem was diagnosed [62].

The fault was eventually traced back to the scheduling algorithms used in the main computer. Two algorithms were used, one for periodic tasks related to flight control, and one for other, less critical, tasks. The algorithms should have been consistent, but investigation showed that in one case they were not, and this eventually caused the timing errors that were observed. The ultimate cause of the error was what can be called *degradation of assumptions*, where an initially valid assumption used by the programmers became incorrect as the software was modified.

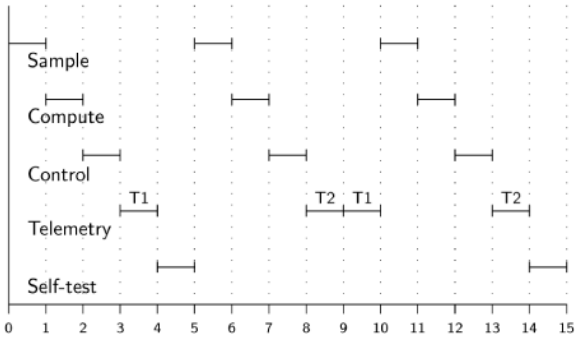
Strangely enough, the fault could only occur if one of the computers was turned on during a single 15 millisecond window of each second. The following diagram will give you an indication of how short that interval is:

The workaround to the fault was simply to turn the computer off and then back on again; there was only a 15 in 1000 (1 in 67) chance of falling into the window again. This also explains why the fault was not discovered in countless hours of testing; normally, you turn on a system and run hours of tests, rather than turning it on and off for each new test.

Synchronous Systems

In a synchronous system, a hardware clock is used to divide the available processor time into intervals called *frames*. The program must then be divided into tasks so that every task can be completed *in the worst case* during a single frame. A scheduling table is constructed which assigns the tasks to frames, so that all the tasks in a frame are completed by the end of the frame. When the clock signals the beginning of the frame, the scheduler invokes the tasks as described in the table. If a task is too long to fit in one frame, it must be artificially split into smaller subtasks that can be individually scheduled.

The following diagram shows a typical, but extremely simple, assignment of tasks from an aircraft system to frames:



There are three critical flight-control tasks: `Sample`, `Compute` and `Control`; these tasks can each be executed within a single frame and they must be executed every five frames. We show two other tasks that are less critical: a `Telemetry` task that takes two frames and must be executed twice every fifteen frames, and a `Self-test` task that takes one frame and must be executed once every ten frames. We have managed to assign tasks to frames, breaking up the `Telemetry` task into two subtasks `Telemetry 1` and `Telemetry 2` that are scheduled separately. This of course requires the programmer to save the state at the end of the first subtask and restore it at the beginning of the second subtask.

Here is a scheduling table assigning tasks to frames:

0	1	2	3	4
Sample	Compute	Control	Telemetry 1	Self-test

5	6	7	8	9
Sample	Compute	Control	Telemetry 2	Telemetry 1

10	11	12	13	14
Sample	Compute	Control	Telemetry 2	Self-test

Algorithm 13.1 for scheduling the tasks is very simple. The assumption underlying this algorithm is that every task can be executed within a single frame. In fact, it has to take a bit less time than that to account for the overhead of the scheduler algorithm. Since the programming of a task may involve `if` and `loop` statements, it may not be possible to precisely calculate the length of time it takes to execute; therefore, it is prudent to ensure that there is time left over in a frame after a task executes. The statement `await beginning of frame` is intended to ensure that each task starts when a frame begins, so that time-dependent algorithms can work as designed. It would normally be implemented as waiting for an interrupt from a hardware clock.

Table 13.1. Synchronous scheduler

```
taskAddressType array[0..numberFrames-1] tasks ←
    [task address,. . . ,task address]
integer currentFrame ← 0
```

```
p1: loop
p2:     await beginning of frame
p3:     invoke tasks[currentFrame]
p4:     increment currentFrame modulo numberFrames
```

The simplicity of the synchronous scheduler is deceiving. All the hard work is hidden in dividing up the computation into tasks and constructing the scheduling table. These systems are very fragile in the sense that it is quite likely that as the system is modified, a task may become too long, and dividing it up may cause difficulties in re-assigning tasks to frames. A further disadvantage is that quite a lot of processor time may be wasted. The worst-case estimate of task duration must be used to ensure that it can be run within a single frame; if the average duration is much smaller, the processor will spend a lot of time waiting for the clock to tick. The frame duration is a basic parameter of the system. If it is made long to accommodate tasks of long duration, the system will not use processor time efficiently. If it is made short, it will become difficult to fit tasks into frames, and they must be divided into subtasks with the accompanying overhead and loss of reliability and maintainability.

On the surface, it may seem that synchronization is not a problem in synchronous systems, because at any time at most one task is executing and the order of execution of the tasks is known. For example, here is a system with a pair of producer–consumer algorithms:

Table 13.2. Producer–consumer (synchronous system)

```
queue of dataType buffer1, buffer2
```

sample	compute	control

`dataType d`
`p1: d ← sample`
`p2: append(d,buffer1)`
`p3:`
	`dataType d1, d2`
	`q1: d1 ← take(buffer1)`
	`q2: d2 ← compute(d1)`
	`q3: append(d2,buffer2)`
		`dataType d`
		`r1: d ← take(buffer2)`
		`r2: control(d)`
		`r3:`

We assume that each task executes to completion during a frame, so there is no danger that one task might increment the count of elements in the buffer while another is decrementing it. Digging deeper, we find that this is far from true. First, tasks—like the `Telemetry` task in the example—may have to be split between frames. The programmer of the task cannot assume that a global variable retains its value from one subtask to another, because arbitrary tasks may be scheduled between the two. Second, the system is certain to undergo modification as time passes. We might initially assume that there is no need to synchronize the buffer used for passing data between the `Sample` task and the `Compute` task, and between the `Compute` task and the `Control` task, since they always occur one after another. This assumption may become invalid, or, more probably, it may be retained (because you don't want to modify code that has worked reliably for a long time), thus constraining future modifications to the program.

Synchronous systems are primarily used in control systems, where the algorithms require strict adherence to a timing specification. Feedback loops typically involve computations of the form $x_t = f(x_t{-}1)$, where the value of a variable at time t depends on its value at time $t - 1$; the computation of the function f may not be very complex, but it is essential that it be carried out at precise intervals.

Asynchronous Systems

In an asynchronous system tasks are not required to complete execution within fixed time frames. Instead, each task executes to completion and then the scheduler is called to find the next task, if any, ready to execute:

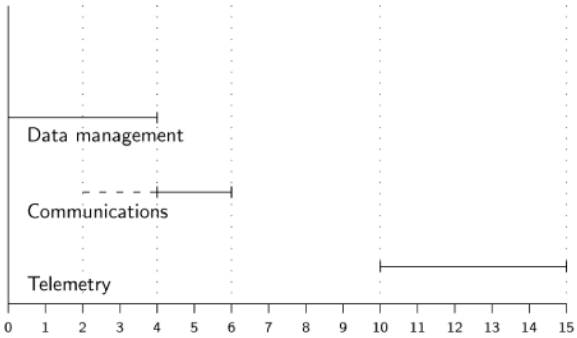
Table 13.3. Asynchronous scheduler

```
                queue of taskAddressType readyQueue ← . . .
                taskAddressType currentTask

-------------------------------------------------------------

      loop forever
p1:     await readyQueue not empty
p2:     currentTask ← take head of readyQueue
p3:     invoke currentTask
```

In the following example, the dashed line starting at time 2 indicates that the `Communications` task is ready to execute, but it must wait until the completion of the `Data management` task in order to begin execution. The `Telemetry` task is released at time 10 and can be executed immediately:



What we haven't shown is how tasks join the queue. One possibility is for one task to request that another be executed by adding it to the queue. This may occur, for example, as the result of receiving input data. Alternatively, a task may request that it be periodically executed, by adding itself to the queue together with an indication of the earliest next time at which it can be run. The scheduler, rather than simply removing and invoking the head of the queue, searches for a task whose time of next execution has passed.

Asynchronous systems are highly efficient, because no processor time is wasted: if a task is ready to run and the processor is free, it will be started immediately. On the other hand, it can be difficult to ensure that deadlines are met in an asynchronous system. For example, if the `Communications` task rescheduled itself to execute at time 11, it would have to wait until time 15 when the execution of the `Telemetry` task is completed.

Priorities and Preemptive Scheduling

Asynchronous systems of the type described are not practical for real-time systems, because a predefined limit would have to be set of the length of a task to ensure that important tasks can be executed when needed. There must be a way to ensure that more important

tasks can be executed at the expense of less important tasks. This can done by defining priorities and using preemptive scheduling.

A priority is a natural number assigned to a task; higher-priority tasks are assumed to be more important or more urgent than lower-priority tasks. In *preemptive scheduling*, a task will not be executed if there is a higher-priority task on the ready queue. This is implemented by defining *scheduling events* that cause the scheduler to be invoked. Whenever a task becomes ready by being appended to the queue, a scheduling event occurs. A scheduling event can also be forced to occur at other times—for example, as the result of a clock interrupt—to ensure a more fair assignment of processor time.

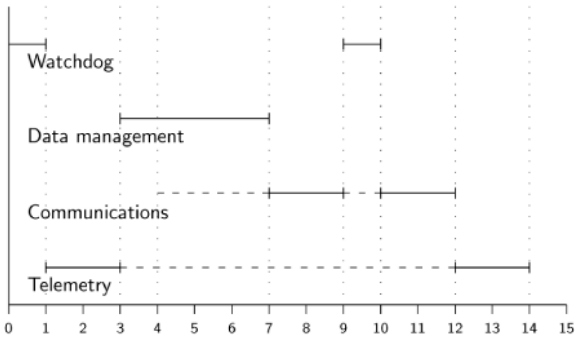
The preemptive scheduler checks the queue of ready tasks; if there is a task on the queue whose priority is higher than the currently running task, it is allowed to run:

Table 13.4. Preemptive scheduler

```
queue of taskAddressType readyQueue ← . . .
taskAddressType currentTask
```

```
    loop forever
p1:    await a scheduling event
p2:    if currentTask.priority < highest priority of a task on readyQueue
p3:       save partial computation of currentTask and place on readyQueue
p4:       currentTask ← take task of highest priority from readyQueue
p5:       invoke currentTask
p6:    else if currentTask's timeslice is past and
              currentTask.priority = priority of some task on readyQueue
p7:       save partial computation of currentTask and place on readyQueue
p8:       currentTask ← take a task of the same priority from readyQueue
p9:       invoke currentTask
p10:   else resume currentTask
```

Here is a timing diagram of the execution of tasks with preemptive scheduling:



In the diagram, higher priority is denoted by a higher position on the *y*-axis. The diagram shows the `Telemetry` task starting its execution at time 1, and then being preempted at time 3 by the `Data management` task. The dashed line starting at time 4 indicates that the `Communications` task is ready to execute, but it will not preempt the higher-priority `Data management` task. Only when it completes at time 7 will the `Communications` task begin executing, and only when it completes at time 12 will the low-priority `Telemetry` task finally be resumed.

The `Watchdog` is a task that is executed periodically with the highest priority in order to check that certain other tasks run sufficiently

often. Suppose we wish to ensure that the important `Data management` task runs at least once every nine units of time. We define a global boolean variable `ran` which is set to true as the last statement of the `Data management` task:

Table 13.5. Watchdog supervision of response time

```
boolean ran ← false
```

data management	**watchdog**
`loop forever`	`loop forever`
`p1: do data management`	`q1: await ninth frame`
`p2: ran ← true`	`q2: if ran is false`
`p3: rejoin readyQueue`	`q3: response-time overflow`
`p4:`	`q4: ran ← false`
`p5:`	`q5: rejoin readyQueue`

If the `watchdog` task executes twice without an intervening execution of the `Data management` task, the variable `ran` will remain false, indicating that the task has either failed to execute or failed to be completed within the required time span.

Algorithm 13.4 also shows how processor time can be shared among tasks of equal priority by time-slicing. Each task is granted a period of time called a *timeslice*; when a task has been computing at least for the duration of its timeslice, another ready task of equal priority (if any) is allowed to execute. As a rule, tasks of equal priority are executed in FIFO order of their joining the ready queue; this is called *round-robin scheduling* and it ensures that no task is ever starved for CPU time (provided, of course, that higher-priority tasks do not monopolize the CPU).

With proper assignment of priorities, preemptive schedulers do a good job of ensuring adequate response time as well as highly efficient use of processor time.

Asynchronous systems have to contend with difficult problems of synchronization. By definition, a preemptive scheduler can interrupt the execution of a task at any point, so the task cannot even assume that two adjacent statements will be executed one after another. In fact, this is an important justification for the definition of the concurrency model in terms of arbitrary interleaving. Since we do not know precisely when an interrupt occurs, it is best to simply assume that it can occur at any time.

Interrupt-Driven Systems

An *interrupt* is the invocation of a software task by the hardware. The software task itself is called an *interrupt handler*. Since many real-time systems are embedded systems, they may be wholly or partially built around interrupts. In fact, it is convenient to consider an interrupt handler to be a normal task whose priority happens to be higher than that of any normal software task. This ensures that interrupt handlers can execute as critical sections with no danger of being interrupted by other tasks. If there are several interrupts, the hardware can be programmed to *mask* interrupts during the execution of an interrupt handler, thus ensuring mutual exclusion.

Alternatively, interrupts themselves may have their own priorities that can be utilized in the design of the system: a higher-priority interrupt can be assured that it will not be preempted by a lower-priority one.

However, the problem remains of how the interrupt handlers synchronize and communicate with the software tasks. For example, suppose that an interrupt signals that data may be read from a temperature sensor; the interrupt handler will read the data and store it in a global buffer for use by another task. This is a producer–consumer problem and we must ensure synchronization between the two tasks. Of course, we could use a synchronization construct like a semaphore to ensure that the consumer does not read from an empty buffer and that the producer does not write to a full buffer. However, there is a lot of overhead in the use of semaphores, and, furthermore, semaphores are a relatively high-level construct that may not be appropriate to use in low-level interrupt handlers.

For these reasons, non-blocking producer–consumer algorithms are preferred for interrupt handling in real-time systems. One possibility is for the producer to throw away the new data if the buffer is full:

Table 13.6. Real-time buffering—throw away new data

```
queue of dataType buffer ← empty queue
```

sample	**compute**

`dataType d`
`loop forever`
`p1:  d ← sample`
`p2:  if buffer is full do nothing`
`p3:  else append(d, buffer)`
	`dataType d`
	`loop forever`
	`q1:  await buffer not empty`
	`q2:  d ← take(buffer)`
	`q3:  compute(d)`

The `compute` task waits for data, either in a busy-wait loop or by being blocked. The `sample` task is invoked by an interrupt whenever data arrives, so there is the possibility that the buffer will be full. If so, it simply throws away the new data.

You should be quite familiar with throwing away new data: if you type on your keyboard so fast that the keyboard buffer fills up, the interrupt handler for the keyboard simply throws the data away (with an annoying beep). This alternative is used in communications systems that depend on acknowledgement of received messages, and retrying in the presence of errors. The system would be able to report that it has successfully received messages `1..n`, but that, sorry, it threw away the later messages that now have to be resent.

Rather than throw away new data, real-time control systems overwrite old data:

Table 13.7. Real-time buffering—overwrite old data

```
queue of dataType buffer ← empty queue
```

sample

```
      dataType d
      loop forever
p1:     d ← sample
p2:     append(d, buffer)
p3:
```

compute

```
      dataType d
      loop forever
q1:     await buffer not empty
q2:     d ← take(buffer)
q3:     compute(d)
```

Interrupt Overflow in the Apollo 11 Lunar Module

The absolutely higher priority given to interrupts over software tasks can cause problems, especially when they are invoked by hardware components over which the designer has little control. This was vividly demonstrated during the first landing on the moon. The radar of the lunar module was programmed to raise interrupts during the descent, but as the module neared the moon's surface, the number of interrupts increased so fast that that they tied up 15% of the resources of the computer system. Figure 13.1 shows what happened. The counter interrupts are unpredictable. The design presumed that they would not occur too often, but in this case, too many interrupts caused a `Main task` to be delayed and not to complete its execution before the check by the `Watchdog`.

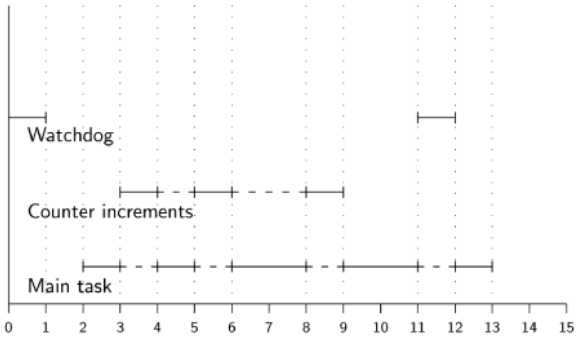


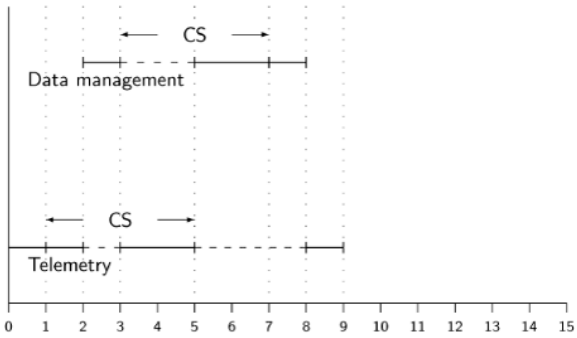
Figure 13.1. Interrupt overflow

In the case of the Apollo 11 landing module, the computer reinitialized itself three times during a 40-second period, causing a warning light to appear. Fortunately, NASA engineers recognized the source of the problem, and knew that it would not affect the success of the landing.

For more information about the Apollo computer system, see [66].

Priority Inversion and Priority Inheritance

Preemptive scheduling based upon priorities can interact with synchronization constructs in unforeseen ways; in particular, *priority inversion* [59] can arise. Consider the following timing diagram:

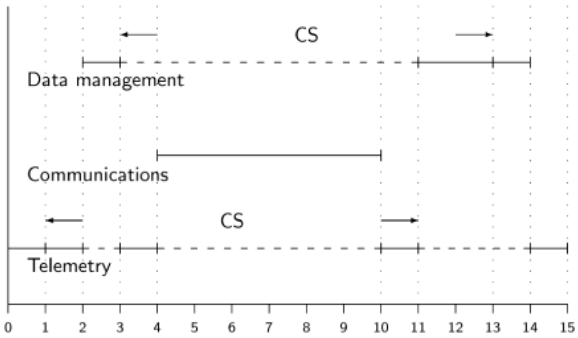


The low-priority `Telemetry` task begins executing and at time 1 enters its critical section. At time 2, it is preempted by the high-priority `Data management` task.

This poses no problem until time 3, when the `Data management` task attempts to enter a critical section that is to be executed under mutual exclusion with the critical section of the `Telemetry` task. The `Data management` task will block at the entry to its critical section, releasing the processor. The `Telemetry` task now resumes its critical section which completes at time 5. The `Data management` task can now preempt it and execute to completion, at which time the `Telemetry` task can be resumed.

At first glance, this seems to be a design defect, because a low-priority task is being executed in preference to a high-priority task, but there is really no other option. By definition, if a task starts a critical section, it must complete it before another task can start its critical section. Real-time systems designers must take to heart the suggestion that critical sections should be as short as possible. If critical sections are short, perhaps only taking a value from a buffer, then their maximum durations are easy to compute and take into account when analyzing if the response time requirements of the system are satisfied.

Priority inversion results if there is a third task whose priority is between those of the other two tasks, and which does not attempt to enter the critical section:



When the `Communications` task becomes ready at time 4, it preempts the `Telemetry` task and executes to completion at time 10; then the `Telemetry` task can resume its critical section. Only at time 11 when the critical section is exited can the high-priority `Data management` task continue. Its completion at time 14 will be way past its deadline and this will be detected by a `Watchdog` task (not shown in the diagram).

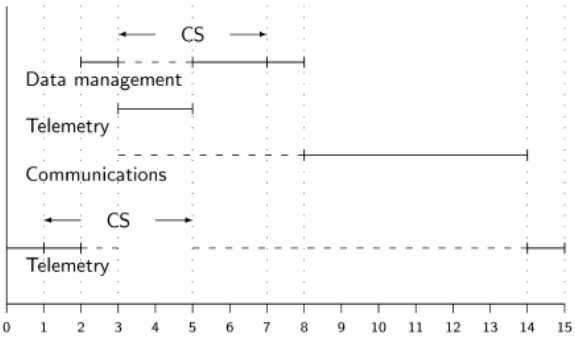
Since the medium-priority task may execute for an arbitrarily long amount of time, the high-priority task is delayed for this amount of time, defeating the concept of preemptive scheduling. Priority inversion must be prevented so that the analysis of the performance of a real-time system can be done independently for each priority

level, considering only tasks with higher priorities and the time spent executing critical sections.

Priority Inheritance

Priority inversion can be solved by a mechanism called *priority inheritance*. Whenever a task `p` is about to be blocked waiting to enter a critical section, a check is made if a lower-priority task `q` is in its critical section; if so, `q`'s priority is temporarily raised to be greater than or equal to the priority of `p`. Task `q` is said to *inherit* the priority of task `p`, enabling `q` to complete its critical section at a high priority, so that `p` will not be unnecessarily delayed.

In the following diagram, when the `Data management` task tries to enter its critical section at time 3, the `Telemetry` task will have its priority raised to be equal to the priority of the `Data management` task:



(For clarity in the diagram, this part of the time line of the `Telemetry` task is shown slightly below that of the `Data management` task.)

During the period from time 3 to time 5, the `Telemetry` task executes its critical section at the higher priority, and therefore is not preempted by the `Communications` task. Upon completion of the critical section at time 5, it relinquishes the higher priority and settles down to wait until time 14 when the other two tasks have been completed.

Priority Inversion from Queues

Priority inversion can also be caused by tasks waiting on queues of monitors or protected objects (Section 7.10). Normally these queues are implemented as FIFO queues. This means that a low-priority task could be blocked on a queue ahead of a high-priority task. The Real-Time Annex of Ada enables you to specify **pragma** `Priority_Queuing`; queues associated with entries of protected objects are then maintained in order of priority. However, if the queues are long, the overhead associated with maintaining these ordered queues may be significant.

Priority Ceiling Locking

For real-time systems with a single CPU, priority inversion can be avoided using an implementation technique called *priority ceiling locking*. This technique is appropriate when synchronization is performed by monitors or protected objects, where the statements that must be executed under mutual exclusion are clearly identified as belonging to a specific software component.

In priority ceiling locking, such component is assigned a *ceiling priority* which is greater than or equal to the highest priority of any task that can call its operations. When a task does call one of these

operations, it inherits the ceiling priority. By definition, no other task that can invoke an operation of the component has the same or a higher priority, so by the normal rules of preemptive scheduling, no other task will ever preempt the task during its execution of the operation. This is sufficient to prevent priority inversion.

Not only will there be no priority inversion, but ceiling priority locking implements mutual exclusion without any need for additional statements! This follows from the same consideration as before: no task can ever preempt a task executing at the ceiling priority.

The Real-Time Annex of the Ada programming language specifies that ceiling priorities must be implemented for protected objects.

The Mars Pathfinder

The description of priority inversion in this section was adapted from a problem that occurred on the Mars Pathfinder mission. This small spacecraft landed on Mars in 1997 and—although designed to last only one month—continued to work for three months, sending back almost ten thousand pictures and large numbers of measurements of scientific data. The computer on the spacecraft used a real-time operating system with preemptive scheduling, and global data areas were protected by semaphores.

The Pathfinder and its computer system were extremely successful, but there was an incident of priority inversion. The symptom of the problem was excessive resets of the software, causing a loss of valuable data. Using a duplicate system on the ground, the problem was finally diagnosed as priority inversion. Fortunately, the operating system implemented priority inheritance and the software was designed to allow remote modification. Once priority inheritance was specified, there were no more resets causing loss of data. For further details, see [35].

The Mars Pathfinder in Spin^{L}

While Spin cannot model the timing aspects of real-time systems, it can nevertheless be used to model qualitative aspects of such systems. Conceptually, the problem that occurred in the Mars Pathfinder is that a “long” computation in a task of medium priority was able to block the completion of a “short” computation in the critical section of a lower-priority task; it is not necessary to define precisely what is meant by “long” and “short.”

Listing 13.1 shows a Promela model of the Pathfinder software. There are three processes: `Data` and `Telem` which alternate between executing critical and noncritical sections, and `Comm` which executes a long computation. The state of each process is stored in the corresponding variables `data`, `telem` and `comm`. The values of these variables are taken from the **mtype** definition and include `idle` and `blocked` which model the scheduler, and three states to model the various computational states: `nonCS`, `CS` and `long`.

The implementation of entering and exiting the critical section is shown in Listing 13.2. The two operations are defined as **atomic** so that no interleaving can occur between the operations on the binary semaphore and the change in the state of the process.

In order to model priority scheduling, we use the **provided** clause of Promela. When a process declaration is suffixed with a **provided** clause, the expression in the clause is implicitly added as a constraint on the execution of any statement in the process. We have defined the symbol `ready(p)` for the expression that checks that the state of process `p` is neither `idle` nor `blocked`. The **provided** clause

for the medium-priority process `Comm` ensures that a statement of the process can only execute if the `Data` process is *not* ready, and similarly, the **provided** clause for the low-priority process `Telem` ensures that a statement of the process can only execute if both other processes are *not* ready.

Example 13.1. A Promela program for the Mars Pathfinder

```
1  mtype = { idle, blocked, nonCS, CS, long };
2  mtype data = idle, comm = idle, telem = idle;
3  #define ready(p) (p != idle && p != blocked)
4
5  active proctype Data() {
6      do
7       :: data = nonCS;
8          enterCS(data);
9          exitCS(data);
10         data = idle;
11     od
12 }
13
14 active proctype Comm() provided (!ready(data)) {
15     do
16      :: comm = long;
17         comm = idle;
18     od
19 }
20
21 active proctype Telem()
22         provided (!ready(data) && !ready(comm)) {
23     do
24      :: telem = nonCS;
25         enterCS(telem);
26         exitCS(telem);
27         telem = idle;
28     od
29 }
```

What can this model show about the correctness of the program? Suppose we add the assertion:

```
assert(! (telem == CS));
```

between lines 7 and 8 of the program. This claims that when the high-priority process wants to enter its critical section, the low-priority process cannot be in its critical section. Clearly, this assertion is false, but that is not considered to be a problem, because critical sections are designed to be short.

Example 13.2. Entering and exiting the critical section

```
1  bit sem = 1;
2  inline enterCS(state) {
3       atomic {
4          if
5          :: sem == 0 ->
6               state = blocked;
7               sem != 0;
8          :: else ->
9          fi;
10         sem = 0;
11         state = CS;
12      }
13 }
14
15 inline exitCS(state) {
16      atomic {
17         sem = 1;
18         state = idle
19      }
20 }
```

Unfortunately, however, the following assertion is also false:

```
assert(! (telem == CS && comm == long));
```

Not only is the low-priority process `Telem` in its critical section, but it will not be allowed to execute because the medium-priority process `Comm` is executing a "long" computation. You can check this by running Spin in its interactive simulation mode. Execute a few steps of `Telem` until it completes execution of `enterCS` and then execute one step of `Comm`. A few steps of `Data` lead now to the following system state:

```
data = blocked, comm = long, telem = CS, sem = 0
```

The interactive simulation offers the following two choices (using the format of the jSpin display):

```
choice 1: 1 (Comm) [comm = idle]
choice 2: 0 (Data) [((! sem=0))]
```

Choice 1 represents the medium-priority process `Comm` completing its long computation and returning to the idle state, while Choice 2 represents the high-priority process `Data` evaluating the semaphore and continuing execution if it is non-zero. We have priority inversion: process `Telem`, which is the only process that can set the semaphore to a nonzero value, is not among the processes that can be selected to run because the medium-priority `Comm` is executing a long computation.

The problem can be solved by priority inheritance. This can be modeled by defining the symbol:

```
#define inherit(p) (p == CS)
```

and changing the **provided** clauses to:

```
active proctype Comm()
    provided (!ready(data) && !inherit (telem))

active proctype Telem()
    provided (!ready(data) &&
              (! ready(comm) || inherit (telem)))
```

This will ensure that the low-priority process `Telem` has the highest priority when it is in its critical section. Executing a safety verification run in Spin proves that the assertion

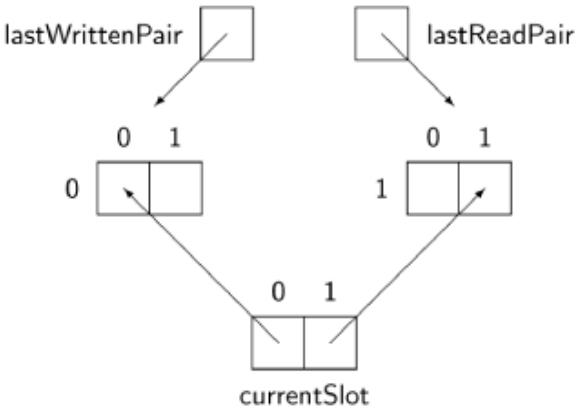
```
assert(! (telem == CS && comm == long));
```

is never violated.

Simpson's Four-Slot Algorithm[A]

In Section 2.9 we discussed *atomic* variables, which are variables that can be read from and written to in a single CPU instruction. On most computer systems, the hardware provides for atomicity at the level of bits or words, so that if we want atomic access to multiword variables (like records) a software mechanism must be provided. For communicating values from interrupts, or more generally in real-time systems, the use of synchronization mechanisms like semaphores and monitors is not appropriate, because they can block the tasks. Here we present Simpson's algorithm [61],[1] which enables consistent access to memory by a reading task and a writing task without blocking either one.

The data structures of the algorithm are illustrated in the following diagram:



At the center are shown four *slots*, variables of the type that must be passed from the writer to the reader. The slots are arranged in two pairs of two slots each. The array `currentSlot` contains the indices of the current slot within each pair; in the diagram, the current slot of pair 0 is 0 and the current slot of pair 1 is 1.

In Algorithm 13.8, the two-dimensional array `data` stores the four slots; the first index is the pair number and second is the slot number within the pair. Note that the indices are bit values and the complement of b is computed as $1 - b$.

`p1`, `p4` and `p11`, `p12` are the normal statements of a solution to the producer–consumer problem. The rest of the algorithm is devoted to computing indices of the pairs and slots.

If a writer task is not active, clearly, the reader tasks will continue to read the freshest data available. If a reader task is not active, the writer task uses the slots of one of the pairs alternately, as shown by the assignment to `writeSlot` in `p3`. The pair index is computed in `p2` as the complement of `lastReadPair` and after the write is stored in `lastWrittenPair`.

Consider now what happens if a write operation commences during the execution of a read operation. `lastReadPair` contains the index of the pair being read from and the writer writes to its complement; therefore, it will not interfere with the read operation and the reader will obtain a consistent value. This holds true even if there is a sequence of write operations during the read operation, because the values will be written to the two slots of `1-lastReadPair`. Although the reader may not read the latest value, it will obtain a consistent value.

What happens if a read operation commences during the execution of a write operation? “During the execution of a write operation” means after executing `p2` but before executing `p6`. It is possible that `lastReadPair` and `lastWrittenPair` have different values, so that after the writer executes `p2` and the reader executes `p7`, both `writePair` and `readPair` contain the same value (exercise). But `p3` and `p9` ensure that the reader and writer access separate slots within the pair so there is no interference.

Table 13.8. Simpson’s four-slot algorithm

```
dataType array[0..1,0..1] data ← default initial values
bit array[0..1] currentSlot ← { 0, 0 }
bit lastWrittenPair ← 1, lastReadPair ← 1
```

- **writer**

```
      bit writePair, writeSlot
      dataType item
      loop forever
p1:     item ← produce
p2:     writePair ← 1- lastReadPair
p3:     writeSlot ← 1- currentSlot[writePair]
p4:     data[writePair, writeSlot] ← item
p5:     currentSlot[writePair] ← writeSlot
p6:     lastWrittenPair ← writePair
```

- **reader**

```
      bit readPair, readSlot
      dataType item
      loop forever
p7:     readPair ← lastWrittenPair
p8:     lastReadPair ← readPair
p9:     readSlot ← currentSlot[readPair]
p10:    item ← data[readPair, readSlot]
p11:    consume(item)
```

All the values in the algorithm, except for the data type being read and written, are single bits, which can be read and written atomically. Furthermore, only one task writes to each global variable: `lastReadPair` by the reader and the others by the writer. The overhead of the algorithm is fixed and small: the memory overhead is just four bits and the three extra slots, and the time overhead is four assignments of bit values in the writer and three in the reader. At no time is either the reader or the writer blocked.

A Promela program for verifying Simpson's algorithm is included in the software archive.

The Ravenscar Profile

There is an obvious tradeoff between powerful and flexible constructs for concurrency, and the efficiency and predictability required for building real-time systems. Ada—a language designed for developing high-reliability real-time systems—has a very rich set of constructs for concurrency, some of which are not deterministic and are difficult to implement efficiently. The designer of a hard real-time system must use only constructs known to be efficiently implemented and to have a predictable execution time.

The *Ravenscar profile* [17] is a specification of a set of concurrency constructs in Ada to be used in developing hard real-time systems. The intention is that the profile be accepted as a standard so that programs will be portable across implementations. Note that the portability refers not to the language constructs themselves, which are already standardized, but to the assurance that this set of constructs will be efficiently implemented and that their execution times will be fixed and predictable for any particular selection of hardware and runtime system. This predictability ensures that programs developed according to the profile can be scheduled using real-time scheduling algorithms such as those described in the next section.

Another aim of the Ravenscar profile is to enable verification of programs by static analysis. Static analysis refers to techniques like model checking and SPARK (Appendix B.4) that are designed to verify the correctness of a program without execution and testing. There are also tools that can statically analyze a scheduling algorithm and the timing constraints for a set of tasks (periods, durations and priorities), in order to determine if hard deadlines can be met. The examples of space system given in this chapter show that testing by itself is not sufficient to verify real-time systems.

To make a system amenable to real-time scheduling algorithms, the Ravenscar profile ensures that the number of tasks is fixed and known when the software is written, and further that tasks are non-terminating. Tasks must be statically declared and after initiation must consist of a non-terminating loop. The scheduler of the runtime system is specified to use **pragma** `FIFO_Within_Priorities`, meaning that the running task is always one with the highest priority and that it runs until it is blocked, at which point other tasks with the same priority are run in FIFO order.

The rendezvous construct (Section 8.6) is incompatible with the requirements for predictability, so all synchronization must be carried out using low-level constructs: variables defined with **pragma** `Atomic` (Section 2.9), suspension objects (see below) or a very restricted form of protected objects (Section 7.10).

The requirement for predictability imposes the following restrictions on protected objects: only one entry per protected object, only one task can be blocked upon an entry and a barrier must contain only boolean variables or constants. Interrupt handlers can be written as procedures in protected objects. The runtime system must implement **pragma** `Ceiling_Locking` to ensure that mutual exclusion in access to a protected object is implemented efficiently by priority ceiling locking rather than by blocking. Static analysis can then ensure that deadlines are met because the delay in entering a protected object depends only on the fixed characteristics of tasks with higher priority.

Clearly, these restrictions severely limit the possible ways of writing concurrent programs in Ada (see [17] for examples of programming style according to the Ravenscar profile). But it is precisely these restrictions that make it possible to verify the correct performance of hard real-time systems.

A recent development is the integration of the Ravenscar profile into the SPARK language and toolset [5].

Suspension Objects

Suppose that we just want a task to signal the occurrence of one event in order to unblock another task waiting for this event. Of course we could use a binary semaphore or a protected object:

Table 13.9. Event signaling

```
binary semaphore s ← 0
```

p	q

```
p1: if decision is to wait for event
p2:    wait(s)
```

```
q1: do something to cause event
q2: signal(s)
```

The problem is that a semaphore is associated with a set or queue of blocked tasks, and a protected entry can also have parameters and statements. The overhead of executing a semaphore `wait` statement or the barrier and body of an entry call is inappropriate when the requirement is for a simple signal. Furthermore, the barrier of a protected entry is limited in the kinds of conditions it can check.

The Real-Time Annex of Ada defines a new type called a *suspension object* by the following package declaration:

```
package Ada.Synchronous_Task_Control is
 type Suspension_Object is limited private;
 procedure Set_True(S: in out Suspension_Object);
 procedure Set_False(S: in out Suspension_Object);
 function Current_State(S: Suspension_Object)
   return Boolean;
 procedure Suspend_Until_True(
   S: in out Suspension_Object);
private
 -- not specified by the language
end Ada.Synchronous_Task_Control;
```

The suspension object itself can be implemented with a single bit that is set to false by a task wishing to wait and then set to true by a task that signals.

Table 13.10. Suspension object—event signaling

`Suspension_Object SO ← false (by default)`

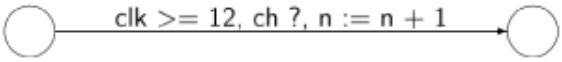
p	q
`p1: if decision is to wait for event` `p2:    Suspend_Until_True(SO)`	`q1: do something to cause event` `q2: Set_True(SO)`

`Suspend_Until_True(SO)` causes the task to block until `SO` is true; it then resets `SO` to false, in preparation for the next time it needs to wait for the event.

UPPAAL[L]

Software tools have been developed to aid in the specification and verification of real-time systems. UPPAAL is similar to Spin, performing simulations and model checking on system specifications. However, it uses *timed automata*, which are finite-state automata that include timing constraints. In UPPAAL, specifications are written directly as automata, rather than as programs in a language like Promela. The automata may be written as text or described using a graphical tool.

A system is specified as a set of tasks, one automaton for each task. There are declarations of global variables for clocks, channels and integer numbers. The automata are built up from states and transitions. Here is an example of a transition:



The first expression is a guard on the transition, which will only be taken when the guard evaluates to true. The guard also includes an input operation on a channel; communication between tasks is synchronous as with rendezvous in Promela. When the transition is taken, the assignment statement (called a reset) is executed.

Correctness specifications are expressed in *branching temporal logic*.

For more details see [43] and the UPPAAL website.

Scheduling Algorithms for Real-Time Systems

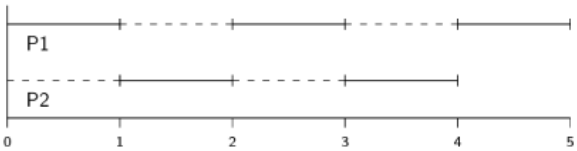
Asynchronous systems have the advantage of flexibility and high utilization of the CPU. In such systems, priorities must be assigned to tasks, and a preemptive scheduler ensures that the processor is assigned to a task with the highest priority. The system designer must be able to assign priorities and to prove that all tasks will meet their deadlines; if so, the assignment is called *feasible*. There are various scheduling algorithms and a large body of theory describing the necessary and sufficient conditions for feasible assignments to exist. In this section, we present two of these algorithms, one for a fixed assignment of priorities and the other for dynamic assignment of priorities.

Let us start with an example. Suppose that there are two tasks `P1` and `P2`, such that

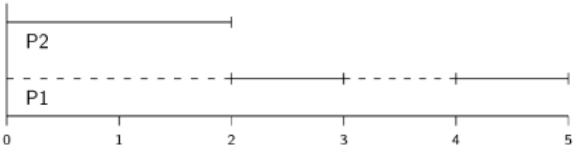
($p_1 = D_1 = 2$, $e_1 = 1$), ($p_2 = D_2 = 5$, $e_2 = 2$).

`P1` needs to be executed every two units of time and it requires one unit during that period, while `P2` requires two units in every five-unit period.

There are two possibilities for assigning priorities: `P1` can have a higher priority than `P2` or vice versa. The first priority assignment is feasible. `P1` completes its execution during the first unit of the first two-unit interval, leaving an an additional unit for `P2`. At the end of two units, the higher-priority task `P1` preempts `P2` for its next interval, after which `P2` continues with its second unit.



However, the second priority assignment is not feasible. If the priority of `P2` is higher than that of `P1`, it will execute for a two-unit period during which `P1` cannot preempt it. Therefore, `P1` has not received its one-unit period during this two-unit interval as required.



The Rate Monotonic Algorithm

The *rate monotonic (RM)* algorithm assigns fixed priorities to tasks in inverse order of the periods p_i, that is, the smaller the period (the

faster a task needs to be executed), the higher its priority, regardless of the duration of the task. In our example, $p_1 < p_2$, so task `P1` receives the higher priority. Under certain assumptions, the RM algorithm is an optimal fixed-priority algorithm, meaning that if there exists a feasible fixed-priority assignment for a set of tasks, then the assignment given by the RM algorithm is also feasible [47, Section 6.4].

The Earliest Deadline First Algorithm

The *earliest deadline first (EDF)* algorithm is an example of an algorithm based upon dynamic modification of task priorities. When a scheduling decision must be made, the EDF assigns the highest priority to the task with the closest deadline. The EDF algorithm is optimal: if it is feasible to schedule a set of tasks, then EDF also gives a feasible schedule. It can be proved that the algorithm is feasible if and only if the processor utilization of the set of tasks, defined as $\Sigma_i(e_i/p_i)$, is less than or equal to one [47, Section 6.3].

EDF is not always applicable because hardware tasks like interrupt routines may have fixed priorities. In addition, the algorithm may impose a serious overhead on the real-time scheduler, because the deadlines have to be recomputed at each scheduling event.

Transition

Concurrency has been a basic property of computer systems since the first multi-tasking operating systems. As the examples in this chapter have shown, even well-engineered and well-tested systems can fall victim to synchronization problems that appear in unexpected scenarios. As computer systems become increasingly integrated into critical aspects of daily life, it becomes more and more important to ensure the safety and reliability of the systems.

Virtually every language or system you are likely to encounter will contain some variant of the constructs you have studied in this book, such as monitors or message passing. I hope that the principles you have studied will enable you to analyze any such construct and use it effectively.

The most important lesson that has been learned from the theory and practice of concurrency is that formal methods are essential for specifying, designing and verifying programs. Until relatively recently, the techniques and tools of formal methods were difficult to use, but the advent of model checking has changed that. I believe that a major challenge for the future of computer science is to facilitate the employment of formal methods in ever larger and more complex concurrent and distributed programs.

Exercises

1. Does the following algorithm ensure that the task `compu` executes once every `period` units of time?

Table 13.11. Periodic task

```
                constant integer period ←
                . . .
```

```
integer next ← currentTime
loop forever
```

```
p1:    delay for (next - currentTime) seconds
p2:    compute
p3:    next ← next + period
```

The function `currentTime` returns the current time and can ignore over-flow of the integer variables.

2. In Algorithm 13.4 for preemptive scheduling, a single rea queue is used; the algorithm searches for tasks of certain priorities on the queue. Modify the algorithm to use sepa ready queues for each level of priority.

3. When a task is preempted, it is returned to the ready qu Should it be placed at the head or tail of the queue (for it priority)? What about a task that is released from being blocked on a semaphore or other synchronization constr

4. In Ada, you can specify that tasks are stored on the queu associated with entries of tasks and protected objects in of ascending priority, rather than in FIFO order:

```
pragma Queuing_Policy(Priority_Queuing)
```

Explain the connection between this and priority inversi

5. Prove that protected objects can be implemented using priorities and ceiling priority locking with no additional locking.

6. Upon completion of a critical section, a low-priority task lose its inherited priority and will likely be preempted b high-priority task whose priority it inherited. Should the priority task be placed at the head or tail of the ready qu (for its priority)?

7. Construct a full scenario of Simpson's algorithm showing `readPair` and `writePair` can get the same value witho there being interference between the reader and the wri

8. Solve the critical section problem using a suspension obj

The following three exercises are taken from Liu [47]. In every case, $D_i = p_i$ and $r_i = 0$ unless otherwise indicated.

9. Let `P1`, `P2` and `P3` be three *nonpreemptable* tasks whose timing constraints are:

($p_1 = 10$, $e_1 = 3$, $r_1 = 0$),

($p_2 = 14$, $e_2 = 6$, $r_2 = 2$),

($p_3 = 12$, $e_3 = 4$, $r_3 = 4$).

Show that the schedule produced by the EDF algorithm i feasible, but that there is a feasible schedule. Can this schedule be generated by a priority-driven scheduling algorithm?

10. Let `P1`, `P2` and `P3` be three tasks whose timing constraint are:

($p_1 = 4$, $e_1 = 1$), ($p_2 = 5$, $e_2 = 2$), ($p_3 = 20$, $e_3 = 5$).

Find the RM schedule for these tasks. At what times is th processor idle?

11. Let `P1`, `P2` and `P3` be three tasks. For which of the three following timing constraints is the RM algorithm feasibl which is the EDF algorithm feasible?

1. ($p_1 = 8$, $e_1 = 3$), ($p_2 = 9$, $e_2 = 3$), ($p_3 = 15$, $e_3 = 3$),
2. ($p_1 = 8$, $e_1 = 4$), ($p_2 = 12$, $e_2 = 4$), ($p_3 = 20$, $e_3 = 4$),
3. ($p_1 = 8$, $e_1 = 4$), ($p_2 = 10$, $e_2 = 2$), ($p_3 = 12$, $e_3 = 3$).

[1] More precisely, this is called Simpson's *asynchronous communication mechanism*.