Real-Time Systems and Programming Languages

Ada 95, Real-Time Java and Real-Time POSIX

Third Edition

Alan Burns and Andy Wellings

ISBN 0 201 72988 1

ADDISON-WESLEY

An imprint of Pearson Education

Harlow, England · London · New York · Reading, Massachusetts · San Francisco · Toronto · Don Mills, Ontario · Sydney · Tokyo · Singapore · Hong Kong · Seoul · Taipei · Cape Town · Madrid · Mexico City · Amsterdam · Munich · Paris · Milan

Chapter 13
Scheduling

In a concurrent program, it is not necessary to specify the exact order in which processes execute. Synchronization primitives are used to enforce the local ordering constraints, such as mutual exclusion, but the general behaviour of the program exhibits significant non-determinism. If the program is correct then its functional outputs will be the same regardless of internal behaviour or implementation details. For example, five independent processes can be executed non-preemptively in 120 different ways on a single processor. With a multiprocessor system or preemptive behaviour, there are infinitely more interleavings.

While the program's outputs will be identical with all these possible interleavings, the timing behaviour will vary considerably. If one of the five processes has a tight deadline then perhaps only interleavings in which it is executed first will meet the program's temporal requirements. A real-time system needs to restrict the non-determinism found within concurrent systems. This process is known as scheduling. In general, a scheduling scheme provides two features:

- An algorithm for ordering the use of system resources (in particular the CPUs).
- A means of predicting the worst-case behaviour of the system when the scheduling algorithm is applied.

The predictions can then be used to confirm that the temporal requirements of the system are satisfied.

A scheduling scheme can be static (if the predictions are undertaken before execution) or dynamic (if run-time decisions are used). This chapter will concentrate mainly on static schemes. Most attention will be given to preemptive priority-based schemes. Here, processes are assigned priorities such that at all times the process with the highest priority is executing (if it is not delayed or otherwise suspended). A scheduling scheme will therefore involve a priority assignment algorithm and a schedulability test.

13.1 Simple process model

An arbitrarily complex concurrent program cannot easily be analyzed to predict its worst-case behaviour. Hence it is necessary to impose some restrictions on the structure of real-time concurrent programs. This section will present a very simple model in order to describe some standard scheduling schemes. The model is generalized in later sections of this chapter (and is further examined in Chapters 14 and 16). The basic model has the following characteristics:

- The application is assumed to consist of a fixed set of processes.
- All processes are periodic, with known periods.
- The processes are completely independent of each other.
- All system's overheads, context-switching times and so on are ignored (that is, assumed to have zero cost).
- All processes have deadlines equal to their periods (that is, each process must complete before it is next released).
- All processes have fixed worst-case execution times.

One consequence of the process's independence is that it can be assumed that at some point in time all processes will be released together. This represents the maximum load on the processor and is known as a **critical instant**.

Table 13.1 gives a standard set of notations for process characteristics.

13.2 The cyclic executive approach

With a fixed set of purely periodic processes, it is possible to lay out a complete schedule such that the repeated execution of this schedule will cause all processes to run at their

Notation	Description
B	Worst-case blocking time for the process (if applicable)
C	Worst-case computation time (WCET) of the process
D	Deadline of the process
I	The interference time of the process
J	Release jitter of the process
N	Number of processes in the system
P	Priority assigned to the process (if applicable)
R	Worst-case response time of the process
T	Minimum time between process releases (process period)
U	The utilization of each process (equal to C/T)
$a - z$	The name of a process

Table 13.1 Standard notation.

correct rate. The cyclic executive is, essentially, a table of procedure calls, where each procedure represents part of the code for a 'process'. The complete table is known as the **major cycle**; it typically consists of a number of **minor cycles** each of fixed duration. So, for example, four minor cycles of 25 ms duration would make up a 100 ms major cycle. During execution, a clock interrupt every 25 ms will enable the scheduler to loop through the four minor cycles. Table 13.2 provides a process set that must be implemented via a simple four-slot major cycle. A possible mapping onto the cyclic executive is shown in Figure 13.1 which illustrates the job that the processor is executing at any particular time. The code for such a system would have a simple form:

```
loop
  wait_for_interrupt;
  procedure_for_a;
  procedure_for_b;
  procedure_for_c;
  wait_for_interrupt;
  procedure_for_a;
  procedure_for_b;
  procedure_for_d;
  procedure_for_e;
  wait_for_interrupt;
  procedure_for_a;
  procedure_for_b;
  procedure_for_c;
  wait_for_interrupt;
  procedure_for_a;
  procedure_for_b;
  procedure_for_d;
end loop;
```

Even this simple example illustrates some important features of this approach:

Process	Period, T	Computation time, C
a	25	10
b	25	8
c	50	5
d	50	4
e	100	2

Table 13.2 Cyclic executive process set.

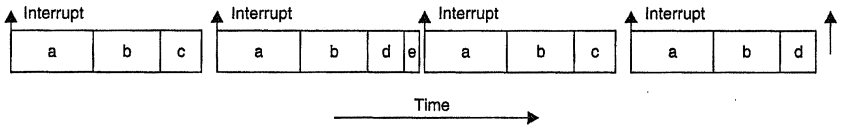

Figure 13.1 Time-line for process set.

- No actual processes exist at run-time; each minor cycle is just a sequence of procedure calls.
- The procedures share a common address space and can thus pass data between themselves. This data does not need to be protected (via a semaphore, for example) because concurrent access is not possible.
- All 'process' periods must be a multiple of the minor cycle time.

This final property represents one of the major drawbacks of the cyclic executive approach; others include (Locke, 1992):

- the difficulty of incorporating sporadic processes;
- the difficulty of incorporating processes with long periods; the major cycle time is the maximum period that can be accommodated without secondary schedules (that is, a procedure in a major cycle that will call a secondary procedure every N major cycles);
- the difficulty of actually constructing the cyclic executive;
- any 'process' with a sizable computation time will need to be split into a fixed number of fixed sized procedures (this may cut across the structure of the code from a software engineering perspective, and hence may be error-prone).

If it is possible to construct a cyclic executive then no further schedulability test is needed (the scheme is 'proof by construction'). However, for systems with high utilization, the building of the executive is problematic. An analogy with the classical bin

packing problem can be made. With that problem, items of varying sizes (in just one dimension) have to be placed in the minimum number of bins such that no bin is overfull. The bin packing problem is known to be NP-hard and hence is computationally infeasible for sizable problems (a typical realistic system will contain perhaps 40 minor cycles and 400 entries). Heuristic sub-optimal schemes must therefore be used.

Although for simple periodic systems, the cyclic executive will remain an appropriate implementation strategy, a more flexible and accommodating approach is furnished by the process-based scheduling schemes. These approaches will therefore be the focus in the remainder of this chapter.

13.3 Process-based scheduling

With the cyclic executive approach, at run-time, only a sequence of procedure calls are executed. The notion of process (thread) is not preserved during execution. An alternative approach is to support process execution directly (as is the norm in general-purpose operating systems) and to determine which process should execute at any one time by the use of one or more scheduling attributes. With this approach, a process is deemed to be in one of a number of *states* (assuming no interprocess communication):

- runnable
- suspended waiting for a timing event – appropriate for periodic processes
- suspended waiting for a non-timing event – appropriate for sporadic processes

13.3.1 Scheduling approaches

There are, in general, a large number of different scheduling approaches. In this book we will consider three.

- Fixed-Priority Scheduling (FPS) – this is the most widely used approach and is the main focus of this chapter. Each process has a fixed, **static**, priority which is computed pre-run-time. The runnable processes are executed in the order determined by their priority. *In real-time systems, the 'priority' of a process is derived from its temporal requirements, not its importance to the correct functioning of the system or its integrity.*
- Earliest Deadline First (EDF) Scheduling. Here the runnable processes are executed in the order determined by the absolute deadlines of the processes; the next process to run being the one with the shortest (nearest) deadline. Although it is usual to know the relative deadlines of each process (e.g. 25 ms after release), the absolute deadlines are computed at run-time, and hence the scheme is described as **dynamic**.
- Value-Based Scheduling (VBS). If a system can become overloaded then the use of simple static priorities or deadlines is not sufficient; a more **adaptive** scheme

is needed. This often takes the form of assigning a *value* to each process and employing an online value-based scheduling algorithm to decide which process to run next.

As indicated earlier, the bulk of this chapter is concerned with FPS as it is supported by various real-time languages and operating system standards. The use of EDF is also important and some consideration of its analytical basis is given in the following discussions. A short description of the use of VBS is given towards the end of the chapter in section 13.13.

13.3.2 Preemption and non-preemption

With priority-based scheduling, a high-priority process may be released during the execution of a lower priority one. In a **preemptive** scheme, there will be an immediate switch to the higher-priority process. Alternatively, with **non-preemption**, the lower-priority process will be allowed to complete before the other executes. In general, preemptive schemes enable higher-priority processes to be more reactive, and hence they are preferred. Between the extremes of preemption and non-preemption, there are alternative strategies that allow a lower priority process to continue to execute for a bounded time (but not necessarily to completion). These schemes are known as **deferred preemption** or **cooperative dispatching**. These will be considered again in Section 13.12.1. Before then, dispatching will be assumed to be preemptive. Schemes such as EDF and VBS can also take on a preemptive or non-preemptive form.

13.3.3 FPS and rate monotonic priority assignment

With the straightforward model outlined in Section 13.1, there exists a simple optimal priority assignment scheme known as **rate monotonic** priority assignment. Each process is assigned a (unique) priority based on its period: the shorter the period, the higher the priority (that is, for two processes i and j, $T_i < T_j \Rightarrow P_i > P_j$). This assignment is optimal in the sense that if any process set can be scheduled (using preemptive priority-based scheduling) with a fixed-priority assignment scheme, then the given process set can also be scheduled with a rate monotonic assignment scheme. Table 13.3 illustrates a five process set and shows what the relative priorities must be for optimal temporal behaviour. Note that priorities are represented by integers, and that the higher the integer, the greater the priority. Care must be taken when reading other books and papers on priority-based scheduling, as often priorities are ordered the other way; that is, priority 1 is the highest. In this book, *priority 1 is the lowest*, as this is the normal usage in most programming languages and operating systems.

Process	Period, T	Priority, P
a	25	5
b	60	3
c	42	4
d	105	1
e	75	2

Table 13.3 Example of priority assignment.

N	Utilization bound
1	100.0%
2	82.8%
3	78.0%
4	75.7%
5	74.3%
10	71.8%

Table 13.4 Utilization bounds.

13.4 Utilization-based schedulability tests

This section describes a very simple schedulability test for FPS which, although not exact, is attractive because of its simplicity.

Liu and Layland (1973) showed that by considering only the utilization of the process set, a test for schedulability can be obtained (when the rate monotonic priority ordering is used). If the following condition is true then all N processes will meet their deadlines (note that the summation calculates the total utilization of the process set):

$$\sum_{i=1}^{N} \left(\frac{C_i}{T_i} \right) \leq N(2^{1/N} - 1) \qquad (13.1)$$

Table 13.4 shows the utilization bound (as a percentage) for small values of N. For large N, the bound asymptotically approaches 69.3%. Hence any process set with a combined utilization of less than 69.3% will always be schedulable by a preemptive priority-based scheduling scheme, with priorities assigned by the rate monotonic algorithm.

Three simple examples will now be given to illustrate the use of this test. In these examples, the units (absolute magnitudes) of the time values are not defined. As long as all the values (Ts, Cs and so on) are in the same units, the tests can be applied. So in these (and later examples), the unit of time is just considered to be a *tick* of some notional time base.

Process	Period, T	Computation time, C	Priority, P	Utilization, U
a	50	12	1	0.24
b	40	10	2	0.25
c	30	10	3	0.33

Table 13.5 Process set A.

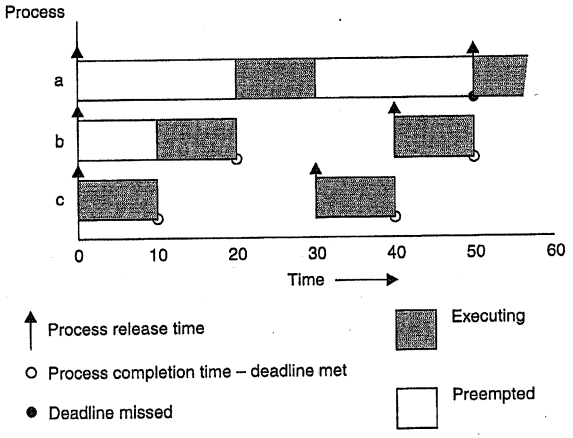

Figure 13.2 Time-line for process set A.

Table 13.5 contains three processes that have been allocated priorities via the rate monotonic algorithm (hence process c has the highest priority and process a the lowest). Their combined utilization is 0.82 (or 82%). This is above the threshold for three processes (0.78), and hence this process set fails the utilization test.

The actual behaviour of this process set can be illustrated by drawing out a **time-line**. Figure 13.2 shows how the three processes would execute if they all started their executions at time 0. Note that, at time 50, process a has consumed only 10 ticks of execution, whereas it needed 12, and hence it has missed its first deadline.

Time-lines are a useful way of illustrating execution patterns. For illustration, Figure 13.2 is drawn as a **Gantt chart** in Figure 13.3.

The second example is contained in Table 13.6. Now the combined utilization is 0.775, which is below the bound, and hence this process set is guaranteed to meet all its deadlines. If a time-line for this set is drawn, all deadlines would be satisfied.

Although cumbersome, time-lines can actually be used to test for schedulability. But how far must the line be drawn before one can conclude that the future holds no

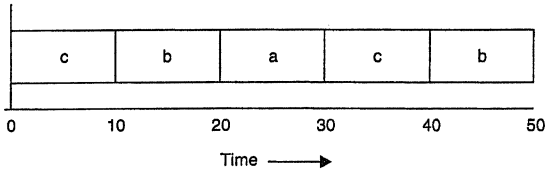

Figure 13.3 Gantt chart for process set A.

Process	Period, T	Computation time, C	Priority, P	Utilization, U
a	80	32	1	0.400
b	40	5	2	0.125
c	16	4	3	0.250

Table 13.6 Process set B.

surprises? For process sets that share a common release time (that is, they share a *critical instant*), it can be shown that a time-line equal to the size of the longest period is sufficient (Liu and Layland, 1973). So if all processes meet their first deadline then they will meet all future ones.

A final example in given in Table 13.7. This is again a three-process system, but the combined utility is now 100%, so it clearly fails the test. At run-time however, the behaviour seems correct, all deadlines are met up to time 80 (see Figure 13.4). Hence the process set fails the test, but at run-time does not miss a deadline. Therefore, the test is said to be **sufficient** but not **necessary**. If a process set passes the test, it *will* meet all deadlines; if it fails the test, it *may* or *may not* fail at run-time. A final point to note about this utilization-based test is that it only supplies a simple yes/no answer. It does not give any indication of the actual response times of the processes. This is remedied in the response time approach described in Section 13.5.

Process	Period, T	Computation time, C	Priority, P	Utilization, U
a	80	40	1	0.50
b	40	10	2	0.25
c	20	5	3	0.25

Table 13.7 Process set C.

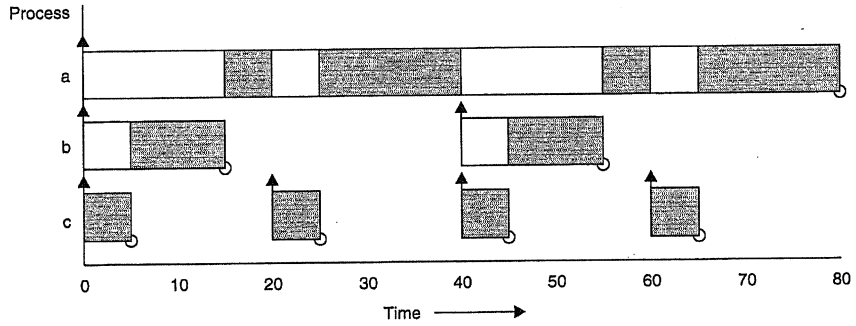

Figure 13.4 Time-line for process set C.

13.4.1 Utilization-based schedulability tests for EDF

Not only did the seminal paper of Liu and Layland (1973) introduce a utilization-based test for fixed priority scheduling but it also gave one for EDF:

$$\sum_{i=1}^{N} \left(\frac{C_i}{T_i} \right) \leq 1 \tag{13.2}$$

Clearly this is a much simpler test. As long as the utilization of the process set is less than the total capacity of the processor then all deadlines will be met (for the simple process model). In this sense EDF is superior to FPS; it can always schedule any process set that FPS can, but not all process sets that are passed by the EDF test can be scheduled using fixed priorities. Given this advantage it is reasonable to ask why EDF is not the preferred process-based scheduling method? The reason is that FPS has a number of advantages over EDF:

- FPS is easier to implement, as the scheduling attribute (*priority*) is static; EDF is dynamic and hence requires a more complex run-time system which will have higher overhead.
- It is easier to incorporate processes without deadlines into FPS (by merely assigning them a priority); giving a process an arbitrary deadline is more artificial.
- The deadline attribute is not the only parameter of importance; again it is easier to incorporate other factors into the notion of priority than it is into the notion of deadline.
- During overload situations (which may be a fault condition) the behaviour of FPS is more predictable (the lower priority processes are those that will miss their deadlines first); EDF is unpredictable under overload and can experience a domino effect in which a large number of processes miss deadlines. This is consider again in Section 13.13.
- The utilization-based test, for the simple model, is misleading as it is necessary and sufficient for EDF but only sufficient for FPS. Hence higher utilizations can, in general, be achieved for FPS.

Notwithstanding this final point, EDF does have an advantage over FPS because of its higher utilization, and hence it continues to be studied and used in some experimental systems.

13.5 Response time analysis for FPS

The utilization-based tests for FPS have two significant drawbacks: they are not exact, and they are not really applicable to a more general process model. This section provides a different form of test. The test is in two stages. First, an analytical approach is used to predict the worst-case response time of each process. These values are then compared, trivially, with the process deadlines. This requires each process to be analyzed individually.

For the highest-priority process, its worst-case response time will equal its own computation time (that is, $R = C$). Other processes will suffer **interference** from higher-priority processes; this is the time spent executing higher-priority processes when a low-priority process is runnable. So for a general process i:

$$R_i = C_i + I_i \tag{13.3}$$

where I_i is the maximum interference that process i can experience in any time interval $[t, t + R_i)$.[1] The maximum interference obviously occurs when all higher-priority processes are released at the same time as process i (that is, at a critical instant). Without loss of generality, it can be assumed that all processes are released at time 0. Consider one process (j) of higher priority than i. Within the interval $[0, R_i)$, it will be released a number of times (at least one). A simple expression for this number of releases is obtained using a ceiling function:

$$Number\_Of\_Releases = \left\lceil \frac{R_i}{T_j} \right\rceil$$

The ceiling function ($\lceil\ \rceil$) gives the smallest integer greater than the fractional number on which it acts. So the ceiling of 1/3 is 1, of 6/5 is 2, and of 6/3 is 2. The definitions of the ceilings of negative values need not be considered.

[1] Note that as a discrete time model is used in this analysis, all time intervals must be closed at the beginning (denoted by '[') and open at the end (denoted by a ')'). Thus a process can complete executing on the same tick as a higher-priority process is released.

So, if R_i is 15 and T_j is 6 then there are 3 releases of process j (at times 0, 6 and 12). Each release of process j will impose an interference of C_j. Hence

$$Maximum\_Interference = \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

If $C_j = 2$ then in the interval [0, 15) there are 6 units of interference. Each process of higher priority is interfering with process i, and hence:

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

where $hp(i)$ is the set of higher-priority processes (than i). Substituting this value back into Equation (13.3) gives (Joseph and Pandya, 1986):

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \quad \textbf{(13.4)}$$

Although the formulation of the interference equation is exact, the actual amounts of interference is unknown as R_i is unknown (it is the value being calculated). Equation (13.4) has R_i on both sides, but is difficult to solve due to the ceiling functions. It is actually an example of a fixed-point equation. In general, there will be many values of R_i that form solutions to Equation (13.4). The smallest such value of R_i represents the worst-case response time for the process. The simplest way of solving Equation (13.4) is to form a recurrence relationship (Audsley et al., 1993a):

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \quad \textbf{(13.5)}$$

The set of values $\{w_i^0, w_i^1, w_i^2, ..., w_i^n, ...\}$ is, clearly, monotonically non-decreasing. When $w_i^n = w_i^{n+1}$, the solution to the equation has been found. If $w_i^0 < R_i$ then w_i^n is the smallest solution and hence is the value required. If the equation does not have a solution then the w values will continue to rise (this will occur for a low-priority process if the full set has a utilization greater than 100%). Once they get bigger than the process's period, T, it can be assumed that the process will not meet its deadline. The above analysis gives rise to the following algorithm for calculation response times:

```
for i in 1..N loop -- for each process in turn
  n := 0
  w_i^n := C_i
  loop
    calculate new w_i^(n+1) from Equation (13.5)
    if w_i^(n+1) = w_i^n then
      R_i := w_i^n
      exit {value found}
    end if
    if w_i^(n+1) > T_i then
      exit {value not found}
    end if
    n := n + 1
  end loop
end loop
```


By implication, if a response time is found it will be less than T_i, and hence less than D_i, its deadline (remember with the simple process model $D_i = T_i$).

In the above discussion, w_i has been used merely as a mathematical entity for solving a fixed-point equation. It is, however, possible to get an intuition for w_i from the problem domain. Consider the point of release of process i. From that point, until the process completes, the processor will be executing processes with priority P_i or higher. The processor is said to be executing a P_i-**busy period**. Consider w_i to be a time window that is moving down the busy period. At time 0 (the notional release time of process i), all higher priority processes are assumed to have also been released, and hence

$$w_i^1 = C_i + \sum_{j \in hp(i)} C_j$$

This will be the end of the busy period unless some higher-priority process is released a second time. If it is, then the window will need to be pushed out further. This continues with the window expanding and, as a result, more computation time falling into the window. If this continues indefinitely then the busy period is unbounded (that is, there is no solution). However, if at any point, an expanding window does not suffer an extra 'hit' from a higher-priority process then the busy period has been completed, and the size of the busy period is the response time of the process.

To illustrate how the response time analysis is used, consider process set D given in Table 13.8.

The highest-priority process, a, will have a response time equal to its computation time (for example, $R_a = 3$). The next process will need to have its response time calculated. Let w_b^0 equal the computation time of process a, which is 3. Equation (13.5)

Process	Period, T	Computation time, C	Priority, P
a	7	3	3
b	12	3	2
c	20	5	1

Table 13.8 Process set D

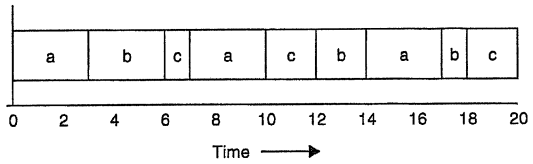

Figure 13.5 Gantt chart for process set D.

is used to derive the next value of w:

$$w_b^1 = 3 + \left\lceil \frac{3}{7} \right\rceil 3$$

that is, $w_b^1 = 6$. This value now balances the Equation ($w_b^2 = w_b^1 = 6$) and the response time of process b has been found (that is, $R_b = 6$).

The final process will give rise to the following calculations:

$$w_c^0 = 5$$

$$w_c^1 = 5 + \left\lceil \frac{5}{7} \right\rceil 3 + \left\lceil \frac{5}{12} \right\rceil 3 = 11$$

$$w_c^2 = 5 + \left\lceil \frac{11}{7} \right\rceil 3 + \left\lceil \frac{11}{12} \right\rceil 3 = 14$$

$$w_c^3 = 5 + \left\lceil \frac{14}{7} \right\rceil 3 + \left\lceil \frac{14}{12} \right\rceil 3 = 17$$

$$w_c^4 = 5 + \left\lceil \frac{17}{7} \right\rceil 3 + \left\lceil \frac{17}{12} \right\rceil 3 = 20$$

$$w_c^5 = 5 + \left\lceil \frac{20}{7} \right\rceil 3 + \left\lceil \frac{20}{12} \right\rceil 3 = 20$$

Hence R_c has a worst-case response time of 20, which means that it will just meet its deadline. This behaviour is illustrated in the Gantt chart shown in Figure 13.5.

Consider again the process set C. This set failed the utilization-based test but was observed to meet all its deadlines up to time 80. Table 13.9 shows the response times calculated by the above method for this collection. Note that all processes are now predicted to complete before their deadlines.

The response time calculations have the advantage that they are sufficient and necessary – if the process set passes the test they will meet all their deadlines; if they fail the test, then, at run-time, a process will miss its deadline (unless the computation time estimations, C, themselves turn out to be pessimistic). As these tests are superior to the utilization-based ones, this chapter will concentrate on extending the applicability of the response time method.

Process	Period, T	Computation time, C	Priority, P	Response time, R
a	80	40	1	80
b	40	10	2	15
c	20	5	3	5

Table 13.9 Response time for process set C.

Process	T(=D)	C
a	4	1
b	12	3
c	16	8

Table 13.10 A process set for EDF.

13.6 Response time analysis for EDF

One of the disadvantages of the EDF scheme is that the worst-case response time for each process does not occur when all processes are released at a critical instant. In that situation only processes with a shorter relative deadline will interfere. But later there may exist a position in which all (or at least more) processes have a shorter absolute deadline. For example, consider a three process system as depicted in Table 13.10. The behaviour of process b illustrates the problem. At time 0, a critical instant, b only gets interference from process a (once) and has a response time of 4. But at its next release (at time 12) process c is still active and has a shorted deadline (16 versus 24) and hence c takes precedence; the response time for this second release of b is 8, twice the value obtained at the critical instant. Later releases may give an even larger value, although it is bounded at 12 as the system is schedulable by EDF (utilization is 1). Hence to find the worst case is much more complex. It is necessary to consider all process releases to see which one suffers the maximum interference from other processes with shorter deadlines.

In the simple model with all periodic processes, the full process set will repeat its execution every *hyper-period*; that is, the least common multiple (LCM) of the process periods. For example, in a small system with only four processes but periods of 24, 50, 73 and 101 time units, the LCM is 4 423 800. To find the worst-case response time for

EDF requires each release within 4 423 800 to be considered – remember with FPS only the first release needs be analyzed (that is, the maximum time to consider is 101 time units).

Although more releases must be considered it is possible to derive a formula for computing each response time in a manner similar to that given above for FPS. We will not give that derivation here, but interested readers can find this (and other results relating to EDF scheduling) in the book on EDF included in the further reading section at the end of the chapter.

13.7 Worst-case execution time

In all the scheduling approaches described so far (that is, cyclic executives, FPS and EDF), it is assumed that the worst-case execution time of each process is known. This is the maximum any process invocation could require.

Worst-case execution time estimation (represented by the symbol C) can be obtained by either measurement or analysis. The problem with measurement is that it is difficult to be sure when the worst case has been observed. The drawback of analysis is that an effective model of the processor (including caches, pipelines, memory wait states and so on) must be available.

Most analysis techniques involve two distinct activities. The first takes the process and decomposes its code into a directed graph of basic blocks. These basic blocks represent straightline code. The second component of the analysis takes the machine code corresponding to a basic block and uses the processor model to estimate its worst-case execution time.

Once the times for all the basic blocks are known, the directed graph can be collapsed. For example, a simple choice construct between two basic blocks will be collapsed to a single value (that is, the largest of the two values for the alternative blocks). Loops are collapsed using knowledge about maximum bounds.

More sophisticated graph reduction techniques can be used if sufficient semantic information is available. To give just a simple example of this, consider the following code:

```
for I in 1.. 10 loop
  if Cond then
    -- basic block of cost 100
  else
    -- basic block of cost 10
  end if;
end loop;
```

With no further information, the total 'cost' of this construct would be $10 \times 100 +$ the cost of the loop construct itself, giving a total of, say, 1005 time units. It may, however, be possible to deduce (via static analysis of the code) that the condition Cond can only be true on at most three occasions. Hence a less pessimistic cost value would be 375 time units.

Other relationships within the code may reduce the number of feasible paths by eliminating those that cannot possibly occur; for instance, when the 'if' branch in one conditional statement precludes a later 'else' branch. Techniques that undertake this sort of semantic analysis usually require annotations to be added to the code. The graph reduction process can then make use of tools such as ILP (integer linear programming) to produce a tight estimate of worst-case execution time. They can also advise on the input data needed to drive the program down the path that gives rise to this estimation.

Clearly, if a process is to be analyzed for its worst-case execution time, the code itself needs to be restricted. For example, all loops and recursion must be bounded, otherwise it would be impossible to predict offline when the code terminates. Furthermore, the code generated by the compiler must also be analyzable.

The biggest challenge facing worst-case execution time analysis comes from the use of modern processors with on-chip caches, pipelines, branch predictors and so on. All of these features aim to reduce average execution time, but their impact on worst-case behaviour can be hard to predict. If one ignores these features the resulting estimates can be very pessimistic, but to include them is not always straightforward. One approach is to assume non-preemptive execution, and hence all the benefits from caching and so on can be taken into account. At a later phase of the analysis, the number of actual preemptions is calculated and a penalty applied for the resulting cache misses and pipeline refills.

To model in detail the temporal behaviour of a modern processor is non-trivial and may need proprietary information that can be hard to obtain. For real-time systems one is left with the choice of either using simpler (but less powerful) processor architectures or to put more effort into measurement. Given that all high-integrity real-time systems will be subject to considerable testing, an approach that combines testing and measurement for code units (basic blocks) but path analysis for complete components seems appropriate with today's technology.

13.8 Sporadic and aperiodic processes

To expand the simple model of Section 13.1 to include sporadic (and aperiodic) process requirements, the value T is interpreted as the minimum (or average) inter-arrival interval (Audsley et al., 1993a). A sporadic process with a T value of 20 ms is guaranteed not to arrive more than once in any 20 ms interval. In reality, it may arrive much less frequently than once every 20 ms, but the response time test will ensure that the maximum rate can be sustained (if the test is passed!).

The other requirement that the inclusion of sporadic processes demands concerns the definition of the deadline. The simple model assumes that $D = T$. For sporadic processes, this is unreasonable. Often a sporadic is used to encapsulate an error-handling routine or to respond to a warning signal. The fault model of the system may state that the error routine will be invoked very infrequently – but when it is, it is urgent and hence it has a short deadline. Our model must therefore distinguish between D and T, and allow $D < T$. Indeed, for many periodic processes, it will be useful to allow the application to define deadline values less than period.

An inspection of the response time algorithm for fixed priority scheduling, described in Section 13.5 reveals that:

- it works perfectly for values of D less than T as long as the stopping criterion becomes $w_i^{n+1} > D_i$,
- it works perfectly well with any priority ordering – $hp(i)$ always gives the set of higher-priority processes.

Although some priority orderings are better than others, the test will provide the worst-case response times for the given priority ordering.

In the Section 13.9, an optimal priority ordering for $D < T$ is defined (and proved). A later section will consider an extended algorithm and optimal priority ordering for the general case of $D < T, D = T$ or $D > T$.

13.8.1 Hard and soft processes

For sporadic processes, average and maximum arrival rates may be defined. Unfortunately, in many situations the worst-case figure is considerably higher than the average. Interrupts often arrive in bursts and an abnormal sensor reading may lead to significant additional computation. It follows that measuring schedulability with worst-case figures may lead to very low processor utilizations being observed in the actual running system. As a guideline for the minimum requirement, the following two rules should always be complied with:

- Rule 1 – all processes should be schedulable using average execution times and average arrival rates.
- Rule 2 – all hard real-time processes should be schedulable using worst-case execution times and worst-case arrival rates of all processes (including soft).

A consequent of Rule 1 is that there may be situations in which it is not possible to meet all current deadlines. This condition is known as a **transient overload**; Rule 2, however, ensures that no hard real-time process will miss its deadline. If Rule 2 gives rise to unacceptably low utilizations for 'normal execution', direct action should be taken to try and reduce the worst-case execution times (or arrival rates).

13.8.2 Aperiodic processes and fixed priority servers

One simple way of scheduling aperiodic processes, within a priority-based scheme, is to run such processes at a priority below the priorities assigned to hard processes. In effect, the aperiodic processes run as background activities, and therefore cannot steal, in a preemptive system, resources from the hard processes. Although a safe scheme, this does not provide adequate support to soft processes which will often miss their deadlines if they only run as background activities. To improve the situation for soft

processes, a **server** can be employed. Servers protect the processing resources needed by hard processes, but otherwise allow soft processes to run as soon as possible.

Since they were first introduced in 1987, a number of server methods have been defined. Here only two will be considered: the Deferrable Server (DS) and the Sporadic Server (SS) (Lehoczky et al., 1987).

With the DS, an analysis is undertaken (using, for example, the response time approach) that enables a new process to be introduced at the highest priority.[2] This process, the server, thus has a period, T_s and a capacity C_s. These values are chosen so that all the hard processes in the system remain schedulable even if the server executes periodically with period T_s and execution time C_s. At run-time, whenever an aperiodic process arrives, and there is capacity available, it starts executing immediately and continues until either it finishes or the capacity is exhausted. In the latter case, the aperiodic process is suspended (or transferred to a background priority). With the DS model, the capacity is replenished every T_s time units.

The operation of the SS differs from DS in its replenishment policy. With SS, if a process arrives at time t and uses c capacity then the server has this c capacity replenished T_s time units after t. In general, SS can furnish higher capacity than DS but has increased implementational overheads. Section 13.14.2 describes how SS is supported by POSIX; DS and SS can be analyzed using response time analyses (Bernat and Burns, 1999).

As all servers limit the capacity that is available to aperiodic soft processes, they can also be used to ensure that sporadic processes do not execute more often than expected. If a sporadic process with interarrival interval of T_i and worst-case execution time of C_i is implemented not directly as a process, but via a server with $T_s = T_i$ and $C_s = C_i$, then its impact (interference) on lower-priority processes is bounded even if the sporadic process arrives too quickly (which would be an error condition).

All servers (DS, SS and others) can be described as *bandwidth preserving* in that they attempt to

- make CPU resources available immediately to aperiodic processes (if there is a capacity);
- retain the capacity for as long as possible if there are currently no aperiodic processes (by allowing the hard processes to execute).

Another bandwidth preserving scheme, which often performs better than the server techniques is **dual-priority scheduling** (Davis and Wellings, 1995). Here, the range of priorities is split into three bands: high, medium and low. All aperiodic processes run in the middle band. Hard processes, when they are released, run in the low band, but they are promoted to the top band in time to meet their deadlines. Hence in the first stage of execution they will give way to aperiodic activities (but will execute if there is no such activity). In the second phase they will move to a higher priority and then have precedence over the aperiodic work. In the high band, priorities are assigned

[2] Servers at other priorities are possible, but the description is more straightforward if the server is given a higher priority than all the hard processes.

according to the deadline monotonic approach (see below). Promotion to this band occurs at time $D - R$. To implement the dual-priority scheme requires a dynamic priority provision.

13.8.3 Aperiodic processes and EDF servers

Following the development of server technology for fixed priority systems, most of the common approaches have been reinterpreted within the context of dynamic EDF systems. For example there is a Dynamic Sporadic Server (Spuri and Buttazzo, 1994). Whereas the static system needs a priority to be assigned (which is done pre-run-time), the dynamic version needs to compute a deadline each time it needs to execute. In essence, the run-time algorithm assigns the server the shortest current deadline if (and only if) there is an outstanding aperiodic process to serve and there is capacity outstanding. Once the capacity is exhausted, the server is suspended until it is replenished.

A different scheme, that has a number of similarities with the dual-priority scheme of FPS, is the *earliest deadline last* (EDL) server defined by Chetto and Chetto (1989). Here hard processes switch between executing under EDF (if there are no aperiodic processes) to EDL (if there is aperiodic work to do). The EDL scheme ensures that every hard process will meet its deadline but postpones the release of the process as long as possible. Hence capacity is made available to aperiodic processes immediately (if there is spare capacity). A hard process will preempt the soft process when it is promoted and will compete its execution just at its deadline.

13.9 Process systems with $D < T$

In the above discussion on sporadic processes is was argued that, in general, it must be possible for a process to define a deadline that is less than its inter-arrival interval (or period). It was also noted earlier that for $D = T$ the rate monotonic priority ordering was optimal for a fixed priority scheme. Leung and Whitehead (1982) showed that for $D < T$, a similar formulation could be defined – the deadline monotonic priority ordering (DMPO). Here, the fixed priority of a process is inversely proportional to its deadline: $(D_i < D_j \Rightarrow P_i > P_j)$. Table 13.11 gives the appropriate priority assignments for a simple process set. It also includes the worst-case response time – as calculated by the algorithm in Section 13.5. Note that a rate monotonic priority ordering would not schedule these processes.

In the following subsection, the optimality of DMPO is proven. Given this result and the direct applicability of response time analysis to this process model, it is clear that fixed priority scheduling can adequately deal with this more general set of scheduling requirements. The same is not true for EDF scheduling. Once processes can have $D < T$ then the simple utilization test (total utilization less than one) cannot be applied. Moreover, the response time analysis, discussed in Section 13.6, is considerable more complex for EDF than it is for FPS.

Having raised this difficulty with EDF is must be remembered that EDF is the more effective scheduling scheme. Hence any process set that passes an FPS schedu-

Process	Period, T	Deadline, D	Computation time, C	Priority, P	Response time, R
a	20	5	3	4	3
b	15	7	3	3	6
c	10	10	4	2	10
d	20	20	3	1	20

Table 13.11 Example process set for DMPO.

lability test *will* also always meet its timing requirements if executed under EDF. The necessary and sufficient tests for FPS can thus be seen as sufficient tests for EDF.

13.9.1 Proof that DMPO is optimal

Deadline monotonic priority ordering (DMPO) is optimal if any process set, Q, that is schedulable by priority scheme, W, is also schedulable by DMPO. The proof of optimality of DMPO will involve transforming the priorities of Q (as assigned by W) until the ordering is DMPO. Each step of the transformation will preserve schedulability.

Let i and j be two processes (with adjacent priorities) in Q such that under W: $P_i > P_j$ and $D_i > D_j$. Define scheme W′ to be identical to W except that processes i and j are swapped. Consider the schedulability of Q under W′:

- All processes with priorities greater than P_i will be unaffected by this change to lower-priority processes.
- All processes with priorities lower than P_j will be unaffected. They will all experience the same interference from i and j.
- Process j, which was schedulable under W, now has a higher priority, suffers less interference, and hence must be schedulable under W′.

All that is left is the need to show that process i, which has had its priority lowered, is still schedulable.

Under W, $R_j \leq D_j$, $D_j < D_i$ and $D_i \leq T_i$ and hence process i only interferes once during the execution of j.

Once the processes have been switched, the new response time of i becomes equal to the old response time of j. This is true because under both priority orderings $C_j + C_i$ amount of computation time has been completed with the same level of interference from higher-priority processes. Process j was released only once during R_j, and hence interferes only once during the execution of i under W′. It follows that:

$$R'_i = R_j \leq D_j < D_i$$

It can be concluded that process i is schedulable after the switch.

Priority scheme W' can now be transformed (to W'') by choosing two more processes 'that are in the wrong order for DMPO' and switching them. Each such switch preserves schedulability. Eventually there will be no more processes to switch; the ordering will be exactly that required by DMPO and the process set will still be schedulable. Hence, DMPO is optimal.

Note that for the special case of $D = T$, the above proof can be used to show that, in this circumstance, rate monotonic ordering is also optimal.

13.10 Process interactions and blocking

One of the simplistic assumptions embodied in the system model, described in Section 13.1, is the need for processes to be independent. This is clearly unreasonable, as process interaction will be needed in almost all meaningful applications. In Chapters 8 and 9, it was noted that processes can interact safely by either some form of protected shared data (using, for example, semaphores, monitors or protected objects) or directly (using some form of rendezvous). All of these language features lead to the possibility of a process being suspended until some necessary future event has occurred (for example, waiting to gain a lock on a semaphore, or entry to a monitor, or until some other process is in a position to accept a rendezvous request). In general, synchronous communication leads to more pessimistic analysis as it is harder to define the real worst case when there are many dependencies between process executions. The following analysis is therefore more accurate when related to asynchronous communication where processes exchange data via protected shared resources. The majority of the material in the next two sections is concerned with fixed-priority scheduling. At the end of this discussion, the applicability of the results to EDF scheduling will be considered.

If a process is suspended waiting for a lower-priority process to complete some required computation then the priority model is, in some sense, being undermined. In an ideal world, such **priority inversion** (Lauer and Satterwaite, 1979) (that is, a high-priority process having to wait for a lower-priority process) should not exist. However, it cannot, in general, be totally eliminated. Nevertheless, its adverse effects can be minimized. If a process is waiting for a lower-priority process, it is said to be **blocked**. In order to test for schedulability, blocking must be bounded and measurable; it should also be small.

To illustrate an extreme example of priority inversion, consider the executions of four periodic processes: a, b, c and d. Assume they have been assigned priorities according to the deadline monotonic scheme, so that the priority of process d is the highest and that of process a the lowest. Further, assume that processes d and a (and processes d and c) share a critical section (resource), denoted by the symbol Q (and V), protected by mutual exclusion. Table 13.12 gives the details of the four processes and their execution sequences; in this table 'E' represents a single tick of execution time and 'Q' (or 'V') represent an execution tick with access to the Q (or V) critical section. Thus process c executes for four ticks; the middle two while it has access to critical section V.

Figure 13.6 illustrates the execution sequence for the start times given in the table.

Process	Priority	Execution sequence	Release time
a	1	EQQQQE	0
b	2	EE	2
c	3	EVVE	2
d	4	EEQVE	4

Table 13.12 Execution sequences.

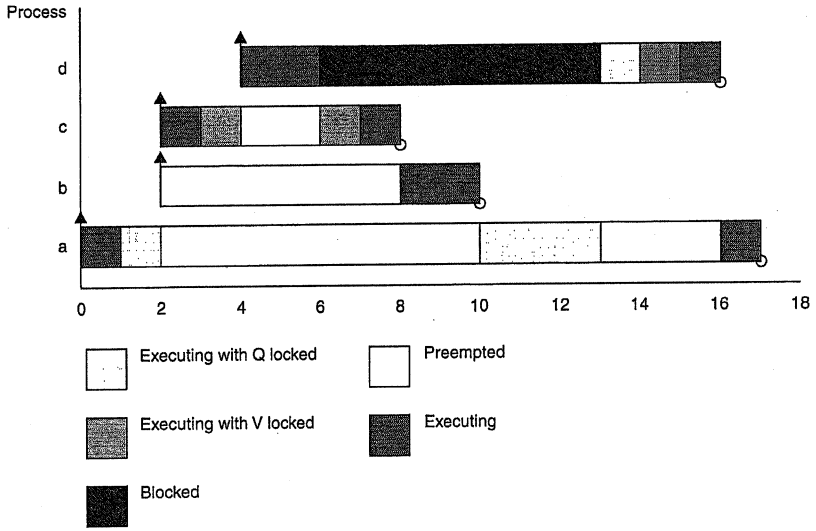

Figure 13.6 Example of priority inversion.

Process a is released first, executes and locks the critical section, Q. It is then preempted by the release of process c which executes for one tick, locks V and is then preempted by the release of process d. The higher-priority process then executes until it also wishes to lock the critical section, Q; it must then be suspended (as the section is already locked by a). At this point, c will regain the processor and continue. Once it has terminated, b will commence and run for its entitlement. Only when b has completed will a be able to execute again; it will then complete its use of the Q and allow d to continue and complete. With this behaviour, d finishes at time 16, and therefore has a response time of 12; c has a value of 6, b a value of 8, and a a value of 17.

An inspection of Figure 13.6 shows that process d suffers considerable priority inversion. Not only is it blocked by process a but also by processes b and c. Some

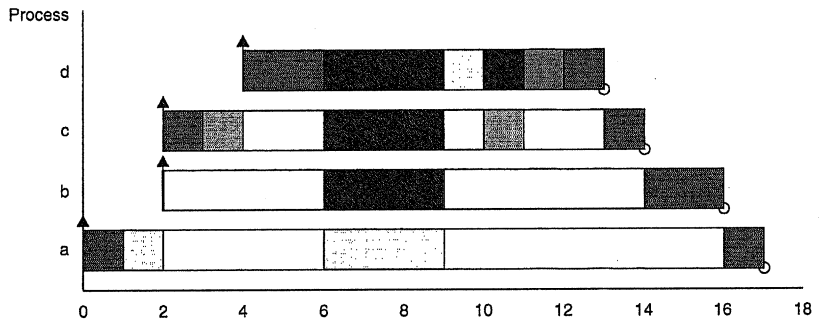

Figure 13.7 Example of priority inheritance.

blocking is inevitable; if the integrity of the critical section (and hence the shared data) is to be maintained then a must run in preference to d (while it has the lock). But the blocking of d by processes c and b is unproductive and will severely affect the schedulability of the system (as the blocking on process d is excessive).

This type of priority inversion is the result of a purely fixed-priority scheme. One method of limiting this effect is to use **priority inheritance** (Cornhill et al., 1987). With priority inheritance, a process's priority is no longer static; if a process p is suspended waiting for process q to undertake some computation then the priority of q becomes equal to the priority of p (if it were lower to start with). In the example just given, process a will be given the priority of process d and will, therefore, run in preference to process c and process b. This is illustrated in Figure 13.7. Note as a consequence of this algorithm, process b will now suffer blocking even though it does not use a shared object. Also note that process d now has a second block, but its response time has been reduced to 9.

With this simple inheritance rule, the priority of a process is the maximum of its own default priority and the priorities of all the other processes that are at that time dependent upon it.

In general, inheritance of priority would not be restricted to a single step. If process d is waiting for process c, but c cannot deal with d because it is waiting for process b then b as well as c would be given d's priority. The implication for the run-time dispatcher is that a process's priorities will often be changing and that it may be better to choose the appropriate process to run (or make runnable) at the time when the action is needed rather than try and manage a queue that is ordered by priority.

In the design of a real-time language, priority inheritance would seem to be of paramount importance. To have the most effective model, however, implies that the concurrency model should have a particular form. With standard semaphores and condition variables, there is no direct link between the act of becoming suspended and the identity of the process that will reverse this action. Inheritance is therefore not easily

implemented. With synchronous message passing, indirect naming (for example, use of the channel in occam2) may also make it difficult to identify the process upon which one is waiting. To maximize the effectiveness of inheritance, direct symmetric naming would be the most appropriate.

Sha et al. (1990) show that with a priority inheritance protocol, there is a bound on the number of times a process can be blocked by lower priority processes. If a process has m critical sections that can lead to it being blocked then the maximum number of times it can be blocked is m. That is, in the worst case, each critical section will be locked by a lower-priority process (this is what happened in Figure 13.7). If there are only n ($n < m$) lower-priority processes then this maximum can be further reduced (to n).

If B_i is the maximum blocking time that process i can suffer then for this simple priority inheritance model, a formula for calculating B can easily be found. Let K be the number of critical sections (resources). Equation (13.6) thus provides an upper bound on B.

$$B_i = \sum_{k=1}^{K} usage(k,i)C(k) \tag{13.6}$$

where $usage$ is a 0/1 function: $usage(k,i) = 1$ if resource k is used by at least one process with a priority less than P_i, and at least one process with a priority greater or equal to P_i. Otherwise it gives the result 0. $C(k)$ is the worst-case execution time of the k critical section.

This algorithm is not optimal for this inheritance protocol, but serves to illustrate the factors that need to be taken into account when calculating B. In Section 13.11, better inheritance protocols will be described and an improved formulae for B will be given.

13.10.1 Response time calculations and blocking

Given that a value for B has been obtained, the response time algorithm can be modified to take the blocking factor into account:[3]

$$R = C + B + I$$

that is,

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{13.7}$$

[3] Blocking can also be incorporated into the utilization-based tests, but now each process must be considered individually.

which can again be solved by constructing a recurrence relationship:

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \qquad (13.8)$$

Note that this formulation may now be pessimistic (that is, not necessarily sufficient and necessary). Whether a process actually suffers its maximum blocking will depend upon process phasings. For example, if all processes are periodic and all have the same period then no preemption will take place and hence no priority inversion will occur. However, in general, Equation (13.7) represents an effective scheduling test for real-time systems containing cooperating processes.

13.11 Priority ceiling protocols

While the standard inheritance protocol gives an upper bound on the number of blocks a high-priority process can encounter, this bound can still lead to an unacceptably pessimistic worst-case calculation. This is compounded by the possibility of chains of blocks developing (transitive blocking), that is, process c being blocked by process b which is blocked by process a and so on. As shared data is a system resource, from a resource management point of view not only should blocking be minimized, but failure conditions such as deadlock should be eliminated. All of these issues are addressed by the ceiling priority protocols (Sha et al., 1990); two of which will be considered in this chapter: the **original ceiling priority protocol** and the **immediate ceiling priority protocol**. The original protocol (OCPP) will be described first, followed by the somewhat more straightforward immediate variant (ICPP). When either of these protocols are used on a single-processor system:

- A high-priority process can be blocked at most once during its execution by lower-priority processes.
- Deadlocks are prevented.
- Transitive blocking is prevented.
- Mutual exclusive access to resources is ensured (by the protocol itself).

The ceiling protocols can best be described in terms of resources protected by critical sections. In essence, the protocol ensures that if a resource is locked, by process a say, and could lead to the blocking of a higher-priority process (b), then no other resource that could block b is allowed to be locked by any process other that a. A process can therefore be delayed by not only attempting to lock a previously locked resource but also when the lock could lead to multiple blocking on higher-priority processes.

The original protocol takes the following form:

(1) Each process has a static default priority assigned (perhaps by the deadline monotonic scheme).

(2) Each resource has a static ceiling value defined, this is the maximum priority of the processes that use it.

(3) A process has a dynamic priority that is the maximum of its own static priority and any it inherits due to it blocking higher-priority processes.

(4) A process can only lock a resource if its dynamic priority is higher than the ceiling of any currently locked resource (excluding any that it has already locked itself).

The locking of a first system resource is allowed. The effect of the protocol is to ensure that a second resource can only be locked if there does not exist a higher-priority process that uses both resources. Consequently, the maximum amount of time a process can be blocked is equal to the execution time of the longest critical section in any of the lower-priority processes that are accessed by higher-priority processes; that is, Equation (13.6) becomes:

$$B_i = \max_{k=1}^{K} usage(k,i)C(k) \qquad (13.9)$$

The benefit of the ceiling protocol is that a high-priority process can only be blocked once (per activation) by any lower-priority process. The cost of this result is that more processes will experience this block.

Not all the features of the algorithm can be illustrated by a single example, but the execution sequence shown in Figure 13.8 does give a good indication of how the algorithm works and provides a comparison with the earlier approaches (that is, this figure illustrates the same process sequence used in Figures 13.7 and 13.6).

In Figure 13.8, process a again locks the first critical section, as no other resources have been locked. It is again preempted by process c, but now the attempt by c to lock the second section (V) is not successful as its priority (3) is not higher than the current ceiling (which is 4, as Q is locked and is used by process d). At time 3, a is blocking c, and hence runs with its priority at the level 3, thereby blocking b. The higher-priority process, d, preempts a at time 4, but is subsequently blocked when it attempts to access Q. Hence a will continue (with priority 4) until it releases its lock on Q and has its priority drop back to 1. Now, d can continue until it completes (with a response time of 7).

The priority ceiling protocols ensure that a process is only blocked once during each invocation. Figure 13.8, however, appears to show process b (and process c) suffering two blocks. What is actually happening is that a single block is being broken in two by the preemption of process d. Equation (13.9) determines that all processes (apart from process a) will suffer a maximum single block of 4. Figure 13.8 shows that for this particular execution sequence process c and process d actually suffer a block of 3 and process d a block of only 2.

13.11.1 Immediate ceiling priority protocol

The immediate ceiling priority algorithm (ICPP) takes a more straightforward approach and raises the priority of a process as soon as it locks a resource (rather than only when

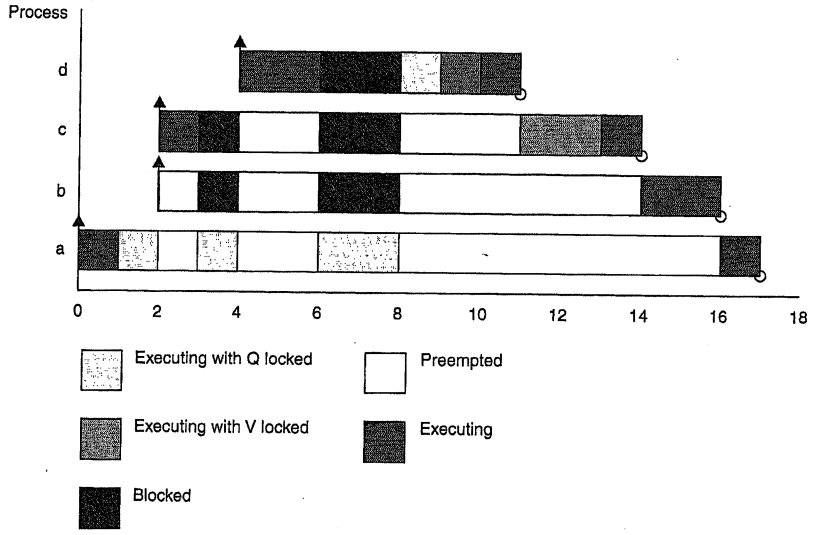

Figure 13.8 Example of priority inheritance – OCPP.

it is actually blocking a higher-priority process). The protocol is thus defined as follows:

- Each process has a static default priority assigned (perhaps by the deadline monotonic scheme).
- Each resource has a static ceiling value defined, this is the maximum priority of the processes that use it.
- A process has a dynamic priority that is the maximum of its own static priority and the ceiling values of any resources it has locked.

As a consequence of this final rule, a process will only suffer a block at the very beginning of its execution. Once the process starts actually executing, all the resources it needs must be free; if they were not, then some process would have an equal or higher priority and the process's execution would be postponed.

The same process set used in earlier illustrations can now be executed under ICPP (see Figure 13.9).

Process a having locked Q at time 1, runs for the next 4 ticks with priority 4. Hence neither process b, process c nor process d can begin. Once a unlocks Q (and has its priority reduced), the other processes execute in priority order. Note that all blocking is before actual execution and that d's response time is now only 6. This is somewhat misleading, however, as the worst-case blocking time for the two protocols is the same (see Equation (13.9)).

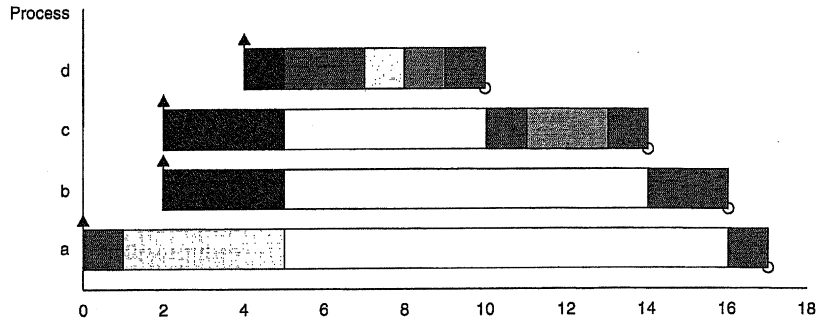

Figure 13.9 Example of priority inheritance – ICPP.

Although the worst-case behaviour of the two ceiling schemes is identical (from a scheduling view point), there are some points of difference:

- ICCP is easier to implement than the original (OCPP) as blocking relationships need not be monitored.
- ICPP leads to less context switches as blocking is prior to first execution.
- ICPP requires more priority movements as this happens with all resource usages; OCPP changes priority only if an actual block has occurred.

Finally, note that Protocol ICPP is called Priority Protect Protocol in POSIX and Priority Ceiling Emulation in Real-Time Java.

13.11.2 Ceiling protocols, mutual exclusion and deadlock

Although the above algorithms for the two ceiling protocols were defined in terms of locks on resources, it must be emphasized that the protocols themselves rather than some other synchronization primitive provided the mutual exclusion access to the resource (at least on a single processor system). Consider ICPP; if a process has access to some resource then it will be running with the ceiling value. No other process that uses that resource can have a higher priority, and hence the executing process will either execute unimpeded while using the resource, or, if it is preempted, the new process will not use this particular resource. Either way, mutual exclusion is ensured.

The other major property of the ceiling protocols (again for single-processor systems) is that they are deadlock-free. In Chapter 11, the issue of deadlock-free resource usage was considered in detail. The ceiling protocols are a form of deadlock prevention. If a process holds one resource while claiming another, then the ceiling of the second resource cannot be lower than the ceiling of the first. Indeed, if two resources are used

in different orders (by different processes) then their ceilings must be identical. As one process is not preempted by another with merely the same priority, it follows that once a process has gained access to a resource then all other resources will be free when needed. There is no possibility of circular waits and deadlock is prevented.

13.11.3 Blocking and EDF

When considering shared resources and blocking, there is a direct analogy between EDF and FPS. Where FPS suffers priority inversion, EDF suffers deadline inversion. This is when a process requires a resource that is currently locked by another process with a longer deadline. Not surprisingly inheritance and ceiling protocols have been developed for EDF, but as with earlier comparisons, the EDF scheme is somewhat more complex.

Because priorities are static, it is easy to determine which processes can block the process currently being analyzed. With EDF, this relationship is dynamic; it depends on which processes (with longer deadlines) are active when the process is released. And this varies from one release to another through the hyper-period.

Probably the best scheme for EDF is the *Stack Resource Policy* (SRP) of Baker (1991). This works in a very similar way to the immediate ceiling priority protocol for FPS (indeed SRP influenced the development of ICPP). Each process, under SRP, is assigned a preemption level. Preemption levels reflect the relative deadlines of the processes, the shorter the deadline the higher the preemption level; so they actual designate the static priority of the process as assigned by the deadline monotonic scheme. At run-time, resources are given ceiling values based on the maximum preemption level of the processes that use the resource. When a process is released it can only preempt the currently executing process if its absolute deadline is shorter *and* its preemption level is higher than the highest ceiling of currently locked resources. The result of this protocol is identical to ICPP. Processes suffer only a single block, it is as they are released, deadlocks are prevented and a simple formulae is available for calculating the blocking time.

13.12 An extendible process model

It was noted earlier that the model outlined in Section 13.1 was too simplistic for practical use. In subsequent sections, three important restrictions were removed:

- Deadlines can be less than period ($D < T$).
- Sporadic and aperiodic processes, as well as periodic processes, can be supported.
- Process interactions are possible, with the resulting blocking being factored into the response time equations.

Within this section, five further generalizations will be given. Only fixed priority scheduling is considered, as EDF analysis is not as mature in these areas. The section will conclude with a general-purpose priority assignment algorithm.

13.12.1 Cooperative scheduling

The models described above have all required true preemptive dispatching. In this section, an alternative scheme is outlined (the use of deferred preemption). This has a number of advantages, but can still be analyzed by the scheduling technique embodied in Equation (13.7). In Equation (13.7), there is a blocking term B that accounts for the time a lower-priority process may be executing while a higher-priority process is runnable. In the application domain, this may be caused by the existence of data that is shared (under mutual exclusion) by processes of different priority. Blocking can, however, also be caused by the run-time system or kernel. Many systems will have the non-preemptable context switch as the longest blocking time (for example, the release of a higher-priority process being delayed by the time it takes to context switch to a lower-priority process – even though an immediate context switch to the higher-priority process will then ensue).

One of the advantages of using the immediate ceiling priority protocol (to calculate and bound B) is that blocking is not cumulative. A process cannot be blocked both by an application process and a kernel routine – only one could actually be happening when the higher-priority process is released.

Cooperative scheduling exploits this non-cumulative property by increasing the situation in which blocking can occur. Let B_{MAX} be the maximum blocking time in the system (using a conventional approach). The application code is then split into non-preemptive blocks, the execution times of which are bounded by B_{MAX}. At the end of each of these blocks, the application code offers a 'de-scheduling' request to the kernel. If a high-priority process is now runnable the kernel will instigate a context switch; if not, the currently running process will continue into the next non-preemptive block.

The normal execution of the application code is thus totally cooperative. A process will continue to execute until it offers to de-schedule. Hence, as long as any critical section is fully contained between de-scheduling calls, mutual exclusion is assured. This method does, therefore, require the careful placement of de-scheduling calls.

To give some level of protection over corrupted (or incorrect) software, a kernel could use an asynchronous signal, or abort, to remove the application process if any non-preemptive block lasts longer than B_{MAX}.

The use of deferred preemption has two important advantages. It increases the schedulability of the system, and it can lead to lower values of C. In the solution of Equation (13.4), as the value of w is being extended, new releases of higher-priority processes are possible that will further increase the value of w. With deferred preemption, no interference can occur during the last block of execution. Let F_i be the execution time of the final block, such that when the process has consumed $C_i - F_i$ time units, the last block has (just) started. Equation (13.4) is now solved for $C_i - F_i$ rather than C_i:

$$w_i^{n+1} = B_{MAX} + C_i - F_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \qquad \textbf{(13.10)}$$

When this converges (that is, $w_i^{n+1} = w_i^n$), the response time is given by:

$$R_i = w_i^n + F_i \qquad \textbf{(13.11)}$$

In effect, the last block of the process has executed with a higher priority (the highest) than the rest of the processes.

The other advantage of deferred preemption comes from predicting more accurately the execution times of a process's non-preemptable basic blocks. Modern processors have caches, prefetch queues and pipelines that all significantly reduce the execution times of straightline code. Typically, simple estimations of worst-case execution time are forced to ignore these advantages and obtain very pessimistic results because preemption will invalidate caches and pipelines. Knowledge of non-preemption can be used to predict the speed up that will occur in practice. However, if the cost of offering a context switch is high, this will militate against these advantages. See Section 16.3.4 for a discussion on how to model the impact of cache effects on the schedulability analysis.

13.12.2 Release jitter

In the simple model, all processes are assumed to be periodic and to be released with perfect periodicity; that is, if process l has period T_l then it is released with exactly that frequency. Sporadic processes are incorporated into the model by assuming that their minimum inter-arrival interval is T. This is not, however, always a realistic assumption. Consider a sporadic process s being released by a periodic process l (on another processor). The period of the first process is T_l and the sporadic process will have the same rate, but it is incorrect to assume that the maximum load (interference) s exerts on low-priority processes can be represented in Equations (13.4) or (13.5) as a periodic process with period $T_s = T_l$.

To understand why this is insufficient, consider two consecutive executions of process l. Assume that the event that releases process s occurs at the very end of the periodic process's execution. On the first execution of process l, assume that the process does not complete until its latest possible time, that is, R_l. However, on the next invocation assume there is no interference on process l so it completes within C_l. As this value could be arbitrarily small, let it equal zero. The two executions of the sporadic process are not separated by T_l but by $T_l - R_l$. Figure 13.10 illustrates this behaviour for T_l equal to 20, R_l equal to 15 and minimum C_l equal to 1 (that is, two releases of the sporadic process within 6 time units). Note that this phenomenon is of interest only if process l is remote. If this was not the case then the variations in the release of process s would be accounted for by the standard equations, where a critical instant can be assumed between the releaser and the released.

To capture correctly the interference sporadic processes have upon other processes, the recurrence relationship must be modified. The maximum variation in a process's release is termed its *jitter* (and is represented by J). For example, in the above, process s would have a jitter value of 15. In terms of its maximum impact on lower-priority processes, this sporadic process will be released at time 0, 5, 25, 45 and so on. That is, at times $0, T-J, 2T-J, 3T-J$, and so on. Examination of the derivation of the schedulability equation implies that process i will suffer one interference from process s if R_i is between 0 and $T-J$, that is $R_i \in [0, T-J)$, two if $R_i \in [T-J, 2T-J)$,

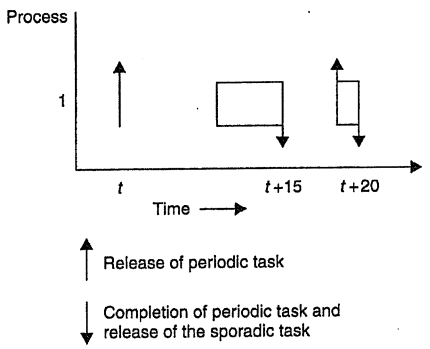

Figure 13.10 Releases of sporadic processes.

three if $R_i \in [2T-J, 3T-J)$ and so on. A slight rearrangement of these conditions shows a single hit if $R_i + J \in [0, T)$, a double hit if $R_i + J \in [T, 2T)$ and so on. This can be represented in the same form as the previous response time equations as follows (Audsley et al., 1993a):

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \qquad (13.12)$$

In general, periodic processes do not suffer release jitter. An implementation may, however, restrict the granularity of the system timer (which releases periodic processes). In this situation, a periodic process may also suffer release jitter. For example, a T value of 10 but a system granularity of 8 will imply a jitter value of 6 – at time 16 the periodic process will be released for its time '10' invocation. If response time (now denoted as $R_i^{periodic}$) is to be measured relative to the real release time then the jitter value must be added to that previously calculated:

$$R_i^{periodic} = R_i + J_i \qquad (13.13)$$

If this new value is greater than T_i then the following analysis must be used.

13.12.3 Arbitrary deadlines

To cater for situations where D_i (and hence potentially R_i) can be greater than T_i, the analysis must again be adapted. When deadline is less than (or equal) to period, it is necessary to consider only a single release of each process. The critical instant, when

all higher-priority processes are released at the same time, represents the maximum interference and hence the response time following a release at the critical instant must be the worst case. However, when deadline is greater than period, a number of releases must be considered. The following assumes that the release of a process will be delayed until any previous releases of the same process have completed.

If a process executes into the next period then both releases must be analyzed to see which gives rise to the longest response time. Moreover, if the second release is not completed before a third occurs than this new release must also be considered, and so on.

For each potentially overlapping release, a separate window $w(q)$ is defined, where q is just an integer identifying a particular window (that is, $q = 0, 1, 2, ...$). Equation (13.5) can be extended to have the following form (ignoring jitter) (Tindell et al., 1994) :

$$w_i^{n+1}(q) = B_i + (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j \quad (13.14)$$

For example, with q equal to 2, three releases of the process will occur in the window. For each value of q, a stable value of $w(q)$ can be found by iteration – as in Equation (13.5). The response time is then given as

$$R_i(q) = w_i^n(q) - qT_i \quad (13.15)$$

for example, with $q = 2$ the process started $2T_i$ into the window and hence the response time is the size of the window minus $2T_i$.

The number of releases that need to be considered is bounded by the lowest value of q for which the following relation is true:

$$R_i(q) \leq T_i \quad (13.16)$$

At this point, the process completes before the next release and hence subsequent windows do not overlap. The worst-case response time is then the maximum value found for each q:

$$R_i = \max_{q=0,1,2,...} R_i(q) \quad (13.17)$$

Note that for $D \leq T$, the relation in Equation (13.16) is true for $q = 0$ (if the process can be guaranteed), in which case Equations (13.14) and (13.15) simplify back to the original equation. If any $R > D$, then the process is not schedulable.

When this arbitrary deadline formulation is combined with the effect of release jitter, two alterations to the above analysis must be made. First, as before, the interference factor must be increased if any higher priority processes suffers release jitter:

$$w_i^{n+1}(q) = B_i + (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q) + J_j}{T_j} \right\rceil C_j \quad (13.18)$$

The other change involves the process itself. If it can suffer release jitter then two consecutive windows could overlap if response time plus jitter is greater than period. To accommodate this, Equation (13.15) must be altered:

$$R_i(q) = w_i^n(q) - qT_i + J_i \quad (13.19)$$

13.12.4 Fault tolerance

Fault tolerance via either forward or backward error recovery always results in extra computation. This could be an exception handler or a recovery block. In a real-time fault tolerant system, deadlines should still be met even when a certain level of faults occur. This level of fault tolerance is know as the **fault model**. If C_i^f is the extra computation time that results from an error in process i, then the response time equation can easily be changed.

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} C_k^f \quad (13.20)$$

where $hep(i)$ is the set of processes with a priority equal or higher than i.

Here, the fault model defines a maximum of one fault and there is an assumption that a process will execute its recovery action at the same priority as its ordinary computation. Equation (13.20) is easily changed to increase the number of allowed faults (F):

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} FC_k^f \quad (13.21)$$

Indeed, a system can be analyzed for increasing values of F to see what number of faults (arriving in a burst) can be tolerated. Alternatively, the fault model may indicate a minimum arrival interval for faults, in this case the equation becomes:

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} \left(\left\lceil \frac{R_i}{T_f} \right\rceil C_k^f \right) \quad (13.22)$$

where T_f is the minimum inter-arrival time between faults.

In Equations (13.21) and (13.22), the assumption is made that in the worst case, the fault will always occur in the process that has the longest recovery time.

13.12.5 Introducing offsets

In the scheduling analysis presented so far in this chapter, it has been assumed that all processes share a common release time, This critical instant is when all processes are released simultaneously (this is usually taken to occur at time 0). For fixed priority

Process	T	D	C
a	8	5	4
b	20	10	4
c	20	12	4

Table 13.13 Example of a process set.

Process	T	D	C	O	R
a	8	5	4	0	4
b	20	10	4	0	8
c	20	12	4	10	8

Table 13.14 Response time analysis of the process set.

scheduling, this is a safe assumption; if all processes meet their timing requirements when released together then they will always be schedulable. There are, however, sets of periodic processes that can benefit from explicitly choosing their release times so that they do not share a critical instant. One process is said to have an **offset** with respect to the others. Consider for illustration the three processes defined in Table 13.13.

If a critical instant is assumed then process a has response time of 4, process b has a response time of 8, but the third process has a worst-case response time of 16, which is beyond its deadline. For process c the interference from process b is sufficient to force a further interference from a, and this is crucial. However, if process c is given an offset (O) of 10 (that is, retain the same period and relative deadline, but have its first release at time 10) then it will never execute at the same time as b. The result is a schedulable process set – see Table 13.14.

Unfortunately, process sets with arbitrary offsets are not amenable to analysis. It is a strongly NP-hard problem to choose offsets so that a process set is optimally schedulable. Indeed, it is far from trivial to even check if a set of processes with offsets share a critical instant.[4]

Notwithstanding this theoretical result, there are process sets that can be analyzed in a relatively straightforward (although not necessarily optimal) way. In most realistic systems, process periods are not arbitrary but are likely to be related to one another. As in the example just illustrated, two processes have a common period. In these situations it is ease to give one an offset (of $T/2$) and to analyze the resulting system using a transformation technique that removes the offset – and hence critical instant analysis applies. In the example, processes b and c (c having the offset of 10) are replaced by a

[4] One interesting result is that a process set with co-prime periods will always have a critical instant no matter what offsets are chosen (Audsley and Burns, 1998).

Process	T	D	C	O	R
a	8	5	4	0	4
n	10	10	4	0	8

Table 13.15 Response time analysis of the transformed process set.

single notional process with period 10, computation time 4, deadline 10 but no offset. This notional process has two important properties.

- If it is schedulable (when sharing a critical instant with all other processes), the two real process will meet their deadlines when one is given the half period offset.
- If all lower priority processes are schedulable when suffering interference from the notional process (and all other high-priority processes), they will remain schedulable when the notional process is replaced by the two real process (one with the offset).

These properties follow from the observation that the notional process always uses more (or equal) CPU time than the two real process. Table 13.15 shows the analysis that would apply to the transformed process set. The notional process is given the name 'n' in this table.

More generally the parameters of the notional process are calculated from the real processes a and b as follows:

$$
\begin{aligned}
T_n &= T_a/2 \quad (or\ T_b/2\ as\ T_a = T_b) \\
C_n &= Max(C_a, C_b) \\
D_n &= Min(D_a, D_b) \\
P_n &= Max(P_a, P_b)
\end{aligned}
$$

where P denotes priority.

Clearly, what is possible for two processes is also applicable to three or more processes. A fuller description of these techniques is given in Bate and Burns (1997). In summary, although arbitrary offsets are effectively impossible to analyze, the judicious use of offsets and the transformation technique can return the analysis problem to one of a simple process set that shares a critical instant. All the analysis given in earlier sections of this chapter, therefore, applies.

13.12.6 Priority assignment

The formulation given for arbitrary deadlines has the property that no simple algorithms (such as rate or deadline monotonic) gives the optimal priority ordering. In this section,

a theorem and algorithm for assigning priorities in arbitrary situations is given. The theorem considers the behaviour of the lowest priority process (Audsley et al., 1993b).

> **Theorem** *If process p is assigned the lowest priority and is feasible, then, if a feasible priority ordering exists for the complete process set, an ordering exists with process p assigned the lowest priority.*

The proof of this theorem comes from considering the schedulability equations – for example, Equation (13.14). If a process has the lowest priority, it suffers interference from all higher-priority processes. This interference is not dependent upon the actual ordering of these higher priorities. Hence if any process is schedulable at the bottom value it can be assigned that place, and all that is required is to assign the other $N - 1$ priorities. Fortunately, the theorem can be reapplied to the reduced process set. Hence through successive reapplication, a complete priority ordering is obtained (if one exists).

The following code in Ada implements the priority assignment algorithm; `Set` is an array of processes that is notionally ordered by priority; `Set(N)` being the highest priority, `Set(1)` being the lowest. The procedure `Process_Test` tests to see whether process `K` is feasible at that place in the array. The double loop works by first swapping processes into the lowest position until a feasible result is found, this process is then fixed at that position. The next priority position is then considered. If at any time the inner loop fails to find a feasible process, the whole procedure is abandoned. Note that a concise algorithm is possible if an extra swap is undertaken.

```
procedure Assign_Pri (Set : in out Process_Set; N : Natural;
                      Ok : out Boolean) is
begin
  for K in 1..N loop
    for Next in K..N loop
      Swap(Set, K, Next);
      Process_Test(Set, K, Ok);
      exit when Ok;
    end loop;
    exit when not Ok;  -- failed to find a schedulable process
  end loop;
end Assign_Pri;
```

If the test of feasibility is exact (necessary and sufficient) then the priority ordering is optimal. Thus for arbitrary deadlines (without blocking), an optimal ordering is found.

13.13 Dynamic systems and online analysis

Earlier in this chapter it was noted that there is a wide variety of scheduling schemes that have been developed for different application requirements. For hard real-time systems, offline analysis is desirable (indeed it is often mandatory). To undertake such analysis requires:

- arrival patterns of incoming work to be known and bounded (this leads to a fixed set of processes with known periods or worst-case arrival intervals);
- bounded computation times; and
- a scheduling scheme that leads to predictable execution of the application processes.

This chapter has shown how fixed-priority scheduling (and to a certain extent, EDF) can provide a predictable execution environment.

In contrast to hard systems, there are dynamic soft real-time applications in which arrival patterns and computation times are not known *a priori*. Although some level of offline analysis may still be applicable, this can no longer be complete and hence some form of online analysis is required.

The main task of an online scheduling scheme is to manage any overload that is likely to occur due to the dynamics of the system's environment. It was noted earlier that EDF is a dynamic scheme that is an optimal scheduling discipline. Unfortunately, EDF also has the property that during transient overloads it performs very badly. It is possible to get a cascade effect in which each process misses its deadline but uses sufficient resources to result in the next process also missing its deadline.

To counter this detrimental domino effect many online schemes have two mechanisms:

(1) an admissions control module that limits the number of processes that are allowed to compete for the processors, and

(2) an EDF dispatching routine for those processes that are admitted.

An ideal admissions algorithm prevents the processors getting overloaded so that the EDF routine works effectively.

If some processes are to be admitted, while others rejected, the relative importance of each process must be known. This is usually achieved by assigning *value*. Values can be classified as follows.

- Static – the process always has the same value whenever it is released.
- Dynamic – the process's value can only be computed at the time the process is released (because it is dependent on either environmental factors or the current state of the system).
- Adaptive – here the dynamic nature of the system is such that the value of the process will change during its execution.

To assign static values (or to construct the algorithm and define the input parameters for the dynamic or adaptive schemes) requires the domain specialists to articulate their understanding of the desirable behaviour of the system. As with other areas of computing, knowledge elicitation is not without its problems. But these issues will not be considered here (see Burns et al. (2000)).

One of the fundamental problems with online analysis is the trade-off that has to be made between the quality of the scheduling decision and the resources and time

needed to make the decision. At one extreme, every time a new process arrives, the complete set of processes could be subject to an exact offline test such as those described in this chapter. If the process set is not schedulable, the lowest value process is dropped and the test repeated (until a schedulable set is obtained). This approach (which is known as *best-effort*) is optimal for static or dynamic value assignment - but only if the overheads of the tests are ignored. Once the overheads are factored in, the effectiveness of the approach is seriously compromised. In general, heuristics have to be used for online scheduling and it is unlikely that any single approach will work for all applications. This is still an active research area. It is clear, however, that what is required is not a single policy defined in a language or OS standard, but mechanisms from which applications can program their own schemes to meet their particular requirements.

The final topic to note in this section is hybrid systems that contain both hard and dynamic components. It is likely that these will become the norm in many application areas. Even in quite static systems, value-added computations, in the form of soft or firm processes that improve the quality of the hard processes, are an attractive way of structuring systems. In these circumstances, as was noted in Section 13.8.1, the hard processes must be protected from any overload induced by the behaviour of the non-hard processes. One way of achieving this is to use fixed priority scheduling for the hard processes and servers for the remaining work. The servers can embody whatever admissions policy is desirable and service the incoming dynamic work using EDF.

13.14 Programming priority-based systems

Few programming languages explicitly define priorities as part of their concurrency facilities. Those that do, often provide only a rudimentary model. For example, occam2 has a variation of the PAR construct that indicates that static priorities should be assigned to the designated processes:

```
PRI PAR
  P1
  P2
  PAR
    P3
    P4
  P5
```

Here, relative priorities are used, with the textual order of the processes in the PRI PAR being significant. Hence P1 has the highest priority, P2 the second highest; P3 and P4 share the next priority level and P5 has the lowest priority. No minimum range of priorities need be supported by an implementation. There is no support for priority inheritance.

The one language that does attempt to give a more complete provision is Ada – this is, therefore, discussed in detail in the next subsection. Traditionally, priority-based scheduling has been more an issue for operating systems than languages. Hence, after a discussion of Ada, the facilities of POSIX are reviewed. Ada and POSIX assume that any schedulability analysis has been performed offline. More recently, Real-Time Java has attempted to provide the same facilities as Ada and POSIX but with the option of supporting online feasibility analysis. This is considered in Section 13.14.3.

It should, perhaps, be noted at this point that although most general-purpose operating systems provide the notion of process or thread priority, their facilities are often inadequate for hard real-time programming.

13.14.1 Ada

As indicated in the Preface, Ada is defined as a core language plus a number of annexes for specialized application domains. These annexes do not contain any new language features (in the way of new syntax) but define pragmas and library packages that must be supported if that particular annex is to be adhered to. This section considers some of the provisions of the Real-Time Systems Annex. In particular, those that allow priorities to be assigned to tasks (and protected objects).[5] In package `System`, there are the following declarations:

```
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;

Default_Priority : constant Priority :=
                   (Priority'First + Priority'Last)/2;
```

An integer range is split between standard priorities and (the higher) interrupt priority range. An implementation must support a range for `System.Priority` of at least 30 values and at least one distinct `System.Interrupt_Priority` value.

A task has its initial priority set by including a pragma in its specification:

```
task Controller is
   pragma Priority(10);
end Controller;
```

If a task-type definition contains such a pragma, then all tasks of that type will have the same priority unless a discriminant is used:

```
task type Servers(Task_Priority : System.Priority) is
   entry Service1(...);
   entry Service2(...);
   pragma Priority(Task_Priority);
end Servers;
```

For entities acting as interrupt handlers, a special pragma is defined:

[5] Priorities can also be given to entry queues and the operation of the select statement. This section will, however, focus on task priorities and protected object ceiling priorities.

```
pragma Interrupt_Priority(Expression);
```

or simply

```
pragma Interrupt_Priority;
```

The definition, and use, of a different pragma for interrupt levels improves the readability of programs and helps to remove errors that can occur if task and interrupt priority levels are confused. However, the expression used in `Interrupt_Priority` evaluates down to `Any_Priority`, and hence it is possible to give a relatively low priority to an interrupt handler. If the expression is actually missing, the highest possible priority is assigned.

A priority assigned using one of these pragmas is called a **base priority**. A task may also have an **active priority** that is higher – this will be explained in due course.

The main program, which is executed by a notional environmental task, can have its priority set by placing the `Priority` pragma in the main subprogram. If this is not done, the default value, defined in `System`, is used. Any other task that fails to use the pragma has a default base priority equal to the base priority of the task that created it.

In order to make use of ICPP, an Ada program must include the following pragma:

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

An implementation may define other locking policies; only `Ceiling_Locking` is required by the Real-Time Systems Annex. The default policy, if the pragma is missing, is implementation defined. To specify the ceiling priority for each protected object, the `Priority` and `Interrupt_Priority` pragmas defined in the previous section are used. If the pragma is missing, a ceiling of `System.Priority'Last` is assumed.

The exception `Program_Error` is raised if a task calls a protected object with a priority greater than the defined ceiling. If such a call were allowed, then this could result in the mutually exclusive protection of the object being violated. If it is an interrupt handler that calls in with an inappropriate priority, then the program becomes erroneous. This must eventually be prevented through adequate testing and/or static analysis of the program.

With `Ceiling_Locking`, an effective implementation will use the thread of the calling task to execute not only the code of the protected call, but also the code of any other task that happens to have been released by the actions of the original call. For example, consider the following simple protected object:

```
protected Gate_Control is
   pragma Priority(28);
   entry Stop_And_Close;
   procedure Open;
private
   Gate: Boolean := False;
end Gate_Control;

protected body Gate_Control is
   entry Stop_And_Close when Gate is
   begin
     Gate := False;
   end Stop_And_Close;

   procedure Open is
   begin
     Gate := True;
   end Open;
end Gate_Control;
```


Assume a task `T`, priority 20, calls `Stop_And_Close` and is blocked. Later, task `S` (priority 27) calls `Open`. The thread that implements `S` will undertake the following actions:

(1) Execute the code of `Open` for `S`.

(2) Evaluate the barrier on the entry and note that `T` can now proceed.

(3) Execute the code of `Stop_And_Close` for `T`.

(4) Evaluate the barrier again.

(5) Continue with the execution of `S` after its call on the protected object.

As a result, there has been no context switch. The alternative is for `S` to make `T` runnable at point (2); `T` now has a higher priority (28) than `S` (27) and hence the system must switch to `T` to complete its execution within `Gate_Control`. As `T` leaves, a switch back to `S` is required. This is much more expensive.

As a task enters a protected object, its priority may rise above the base priority level defined by the `Priority` or `Interrupt_Priority` pragmas. The priority used to determine the order of dispatching is the **active priority** of a task. This active priority is the maximum of the task's base priority and any priority it has inherited.

The use of a protected object is one way in which a task can inherit a higher active priority. There are others, for example:

- During activation – a task will inherit the active priority of the parent task that created it; remember the parent task is blocked waiting for its child task to complete, and this could be a source of priority inversion without this inheritance rule.
- During a rendezvous – the task executing the accept statement will inherit the active priority of the task making the entry call (if it is greater than its own priority).

Note that the last case does not necessarily remove all possible cases of priority inversion. Consider a server task, `S`, with entry `E` and base priority `L` (low). A high-priority task makes a call on `E`. Once the rendezvous has started, `S` will execute with the higher priority. But before `S` reaches the accept statement for `E`, it will execute with priority `L` (even though the high-priority task is blocked). This, and other candidates for priority inheritance, can be supported by an implementation. The implementation must, however, provide a pragma that the user can employ to select the additional conditions explicitly.

The Real-Time Systems Annex attempts to provide flexible and extensible features. Clearly, this is not easy. Ada 83 suffered from being too prescriptive. However, the lack of a defined dispatching policy would be unfortunate, as it would not assist software development or portability. If base priorities have been defined, then it is assumed that preemptive priority-based scheduling is to be employed. On a multiprocessor system, it is implementation defined whether this is on a per-processor basis or across the entire processor cluster.

To give extensibility, the dispatching policy can be selected by using the following pragma:

```
pragma Task_Dispatching_Policy(Policy_Identifier);
```

The Real-Time Systems Annex defines one possible policy: `FIFO_Within_Priority`. Where tasks share the same priority, then they are queued in FIFO order. Hence, as tasks become runnable, they are placed at the back of a notional run queue for that priority level. One exception to this case is when a task is preempted; here the task is placed at the front of the notional run queue for that priority level.

If a program specifies the `Fifo_Within_Priority` option, then it must also pick the `Ceiling_Locking` policy defined earlier. Together, they represent a consistent and usable model for building, implementing and analyzing real-time programs.

Other Ada facilities

Ada also provides other facilities which are useful for programming a wide variety of systems. For example, dynamic priorities, prioritized entry queues, task attributes, asynchronous task control facilities, and so on. The reader is referred to the Systems Programming and Real-Time Annexes of the Ada Reference Manual or Burns and Wellings (1998) for further details.

Currently, there is no support in Ada for sporadic servers. However, see Harbour et al. (1998) for a discussion on how they can be approximated.

13.14.2 POSIX

POSIX supports priority-based scheduling, and has options to support priority inheritance and ceiling protocols. Priorities may be set dynamically. Within the priority-based facilities, there are four policies:

- FIFO – a process/thread runs until it completes or it is blocked; if a process/thread is preempted by a higher-priority process/thread then it is placed at the head of the run queue for its priority.
- Round-Robin – a process/thread runs until it completes or it is blocked or its time quantum has expired; if a process/thread is preempted by a higher-priority process then it is placed at the head of the run queue for its priority; however, if its quantum expires it is placed at the back.

- Sporadic Server – a process/thread runs as a sporadic server (see below).
- OTHER – an implementation-defined policy (which must be documented).

For each scheduling policy, there is a minimum range of priorities that must be supported; for FIFO and round-robin, this must be at least 32. The scheduling policy can be set on a per process and a per thread basis.

Threads may be created with a 'system contention' option, in which case they compete with other system threads according to their policy and priority. Alternatively, threads can be created with a 'process contention' option; in this case they must compete with other threads (created with a process contention) in the parent process. It is unspecified how such threads are scheduled relative to threads in other processes or to threads with global contention. A specific implementation must decide whether to supports 'system contention' or 'process contention' or both.

As discusses in Section 13.8.2, a sporadic server assigns a limited amount of CPU capacity to handle events, has a replenishment period, a budget, and two priorities. The server runs at a high priority when it has some budget left and a low one when its budget is exhausted. When a server runs at the high priority, the amount of execution time it consumes is subtracted from its budget. The amount of budget consumed is replenished at the time the server was activated plus the replenishment period. When its budget reaches zero, the server's priority is set to the low value.

Program 13.1 illustrates a C interface to the POSIX scheduling facilities. The functions are divided into those which manipulate a process's scheduling policy and parameters, and those that manipulate a thread's scheduling policy and parameters. If a thread modifies the policy and parameters of its owning process, the effect on the thread will depend upon its contention scope (or level). If it is contending at a system level, the change will not affect the thread. If, however, it is contending at the process level, there will be an impact on the thread.

In order to prevent priority inversion, POSIX allows inheritance protocols to be associated with mutex variables. As well as catering for basic priority inheritance, the immediate ceiling priority protocol (called priority protect protocol by POSIX) is also supported.

Other POSIX facilities

POSIX provides other facilities that are useful for real-time systems. For example, it allows:

- message queues to be priority ordered;
- functions for dynamically getting and setting a thread's priority;
- threads to indicate whether their attributes should be inherited by any child thread they create.

Program 13.1 A typical C interface to some of the POSIX scheduling facilities.

```
#define SCHED_FIFO ...      /* preemptive priority scheduling */
#define SCHED_RR ...        /* preemptive priority with quantum */
#define SCHED_SPORADIC ...  /* sporadic server */
#define SCHED_OTHER ...     /* implementation-defined scheduler */
#define PTHREAD_SCOPE_SYSTEM ... /* system-wide contention */
#define PTHREAD_SCOPE_PROCESS ...  /* local contention */
#define PTHREAD_PRIO_NONE ... /* no priority inheritance */
#define PTHREAD_PRIO_INHERIT ... /* basic priority inheritance */
#define PTHREAD_PRIO_PROTECT ... /* ICPP */

typedef ... pid_t;
struct sched_param {
  ...
  int sched_priority; /* used for SCHED_FIFO and SCHED_RR */
  int sched_ss_low_priority
  timespec sched_ss_repl_period
  timespec sched_ss_init_budget
  int  sched_ss_max_repl
  ... };

int sched_setparam(pid_t pid, const struct sched_param *param);
/* set the scheduling parameters of process pid */

int sched_getparam(pid_t pid, struct sched_param *param);
/* get the scheduling parameters of process pid */

int sched_setscheduler(pid_t pid, int policy,
                       const struct sched_param *param);
/* set the scheduling policy and parameters of process pid */

int sched_getscheduler(pid_t pid);
/* returns the scheduling policy of process pid */

int sched_yield(void);
/* causes the current thread/process to be placed at the back */
/* of the run queue */

int sched_get_priority_max(int policy);
/* returns the maximum priority for the policy specified */

int sched_get_priority_min(int policy);
/* returns the minimum priority for the policy specified */

int sched_rr_get_interval(pid_t pid, struct timespec *t);
/* if pid /= 0, the time quantum for the calling process/thread is *
 * set in the structure referenced by t                            *
 * if pid = 0, the calling process/threads time quantum is set     *
 * in the structure pointed to by t                                */

int pthread_attr_setscope(pthread_attr_t *attr,
                          int contentionscope);
```



```
/* set the contention scope attribute for a thread attribute object */

int pthread_attr_getscope(const pthread_attr_t *attr,
                          int *contentionscope);
/* get the contention scope attribute for a thread attribute object */

int pthread_attr_setschedpolicy(pthread_attr_t *attr,
                          int policy);
/* set the scheduling policy attribute for a thread attribute object */

int pthread_attr_getschedpolicy(const pthread_attr_t *attr,
                          int *policy);
/* get the scheduling policy attribute for a thread attribute object */

int pthread_attr_setschedparam(pthread_attr_t *attr,
                          const struct sched_param *param);
/* set the scheduling policy attribute for a thread attribute object */

int pthread_attr_getschedparam(const pthread_attr_t *attr,
                          struct sched_param *param);
/* get the scheduling policy attribute for a thread attribute object */

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
                          int protocol);
/* set the priority inheritance protocol */
int pthread_mutexattr_getprotocol(pthread_mutexattr_t *attr,
                          int *protocol);
/* get the priority inheritance protocol */

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
                          int prioceiling);
/* set the priority ceiling */
int pthread_mutexattr_getprioceiling(pthread_mutexattr_t *attr,
                          int *prioceiling);
/* get the priority ceiling */

/* All the above integer functions except sched_get_priority_max  */
/* and sched_get_priority_min return 0 if successful */
```

13.14.3 Real-Time Java

Real-Time Java has the notion of a schedulable object. This is any object that supports the `Schedulable` interface given in Program 13.2. The classes `RealtimeThread`, `NoHeapRealTimeThread` and `AsyncEventHandler` all support this interface. Objects of these classes all have scheduling parameters (see Program 13.3). Real-Time Java implementations are required to support at least 28 real-time priority levels. As with Ada and POSIX, the larger the integer value, the higher the priority (and, therefore, the greater the execution eligibility). Non-real-time threads are given priority levels below

Program 13.2 The Real-Time Java `Schedulable` interface.

```
public interface Schedulable extends java.lang.Runnable
{

  public void addToFeasibility();
  public void removeFromFeasibility();

  public MemoryParameters getMemoryParameters();
  public void setMemoryParameters(MemoryParameters memory);

  public ReleaseParameters getReleaseParameters();
  public void setReleaseParameters(ReleaseParameters release);

  public SchedulingParameters getSchedulingParameters();
  public void setSchedulingParameters(SchedulingParameters scheduling);

  public Scheduler getScheduler();
  public void setScheduler(Scheduler scheduler);
}
```

Program 13.3 The Real-Time Java `SchedulingParameters` class and its subclasses.

```
public abstract class SchedulingParameters
{
  public SchedulingParameters();
}

public class PriorityParameters extends SchedulingParameters
{
  public PriorityParameters(int priority);

  public int getPriority();
  public void setPriority(int priority) throws
                          IllegalArgumentException;
  ...
}

public class ImportanceParameters extends PriorityParameters
{
  public ImportanceParameters(int priority, int importance);
  public int getImportance();
  public void setImportance(int importance);
  ...
}
```


Program 13.4 The Real-Time Java `Scheduler` class.

```
public abstract class Scheduler
{
  public Scheduler();

  protected abstract void addToFeasibility(Schedulable schedulable);
  protected abstract void removeFromFeasibility(Schedulable schedulable);

  public abstract boolean isFeasible();
  // checks the current set of schedulable objects

  public boolean changeIfFeasible(Schedulable schedulable,
                ReleaseParameters release, MemoryParameters memory);

  public static Scheduler getDefaultScheduler();
  public static void setDefaultScheduler(Scheduler scheduler);

  public abstract java.lang.String getPolicyName();
}
```

the minimum real-time priority. Note, scheduling parameters are bound to threads at thread creation time (see Program 12.11). If the parameter objects are changed, they have an immediate impact on the associated threads.

In common with Ada and Real-Time POSIX, Real-Time Java supports a preemptive priority-based dispatching policy. However, unlike Ada and Real-Time POSIX, Real-Time Java does not require a preempted thread to be placed at the head of the run queue associated with its priority level. Real-Time Java also supports a high-level scheduler whose goals are:

- to decide whether to admit new schedulable objects according to the resources available and a feasibility algorithm, and
- to set the priority of the schedulable objects according to the priority assignment algorithm associated with the feasibility algorithm.

Hence, while Ada and Real-Time POSIX focus on static offline schedulability analysis, Real-Time Java addresses more dynamic systems with the potential for online analysis.

The `Scheduler` abstract class is given in Program 13.4. The `isFeasible` method considers only the set of schedulable objects that have been added to its feasibility list (via the `addToFeasibility` and `removeFromFeasibility` methods). The method `changeIfFeasible` checks to see if its set of objects is still feasible if the given object has its release and memory parameters changed. If it is, the parameters are changed. Static methods allow the default scheduler to be queried or set.

One defined subclass of the `Scheduler` class is the `PriorityScheduler` class (defined in Program 13.5) which implements standard preemptive priority-based scheduling.

Program 13.5 The Real-Time Java `PriorityScheduler` class.

```
class PriorityScheduler extends Scheduler
{
  public PriorityScheduler()

  protected void addToFeasibility(Schedulable s);
  protected void removeFromFeasibility(Schedulable s);

  public boolean isFeasible();
  // checks the current set of schedulable objects

  public boolean changeIfFeasible(Schedulable schedulable,
               ReleaseParameters release, MemoryParameters memory);

  protected void addToFeasibility(Schedulable s);
  protected void removeFromFeasibility(Schedulable s);

  public abstract boolean isFeasible();
  // checks the current set of schedulable objects

  public void fireSchedulable(Schedulable schedulable);

  public int getMaxPriority();
  public int getMinPriority();
  public int getNormPriority();
  public java.lang.String getPolicyName();

  public static PriorityScheduler instance();

  ...
}
```

Again, it should be stressed that Real-Time Java does not require an implementation to provide an online feasibility algorithm. It is adequate to simply return false to the methods which test for feasibility.

Support for priority inheritance

Real-Time Java allows priority inheritance algorithms to be used when accessing synchronized classes. To achieve this, three classes are defined (see Program 13.6). Consider the following class:

```
public class SynchronizeClass
{
  public void Method1() { ... };
  public void Method2() { ... };
}
```


Program 13.6 Real-Time Java classes supporting priority inheritance.

```
public abstract class MonitorControl
{
  public MonitorControl();

  public static void setMonitorControl(MonitorControl policy);
  // set the default

  public static void setMonitorControl(java.lang.Object monitor,
                                       MonitorControl policy);
  // sets an individual objects policy
}

public class PriorityCeilingEmulation extends MonitorControl
{
  public PriorityCeilingEmulation(int ceiling);

  public int getDefaultCeiling();
  // get ceiling for this object
}

public class PriorityInheritance extends MonitorControl
{
  public PriorityInheritance();

  public static PriorityInheritance instance();
}
```

An instance of this class can have its control protocol set to immediate priority ceiling inheritance (called priority ceiling emulation by Real-Time Java) with a priority of 10 by the following code

```
SynchronizeClass SC = new SynchronizeClass();
PriorityCeilingEmulation PCI = new PriorityCeilingEmulation(10);
...
MonitorControl.setMonitorControl(SC, PCI);
```

13.14.4 Other Real-Time Java facilities

It is important to note that all queues in Real-Time Java are priority ordered.

One further facility is worth mentioning, Real-Time Java provides support for aperiodic threads in the form of processing groups. A group of aperiodic threads can be linked together and assigned characteristics which aid feasibility analysis. For example, the threads can be defined to consume no more than `cost` CPU time within a given period. The full definition of the class supporting process groups is given in the Appendix.

Summary

A scheduling scheme has two facets: it defines an algorithm for resource sharing; and a means of predicting the worst-case behaviour of an application when that form of resource sharing is used.

Most current periodic real-time systems are implemented using a cyclic executive. With this approach, the application code must be packed into a fixed number of 'minor cycles' such that the cyclic execution of the sequence of minor cycles (called a 'major cycle') will enable all system deadlines to be met. Although an effective implementation strategy for small systems, there are a number of drawbacks with this cyclic approach.

- The packing of the minor cycles becomes increasingly difficult as the system grows.
- Sporadic activities are difficult to accommodate.
- Processes with long periods (that is, longer than the major cycle) are supported inefficiently.
- Processes with large computation times must be split up so that they can be packed into a series of minor cycles.
- The structure of the cyclic executive makes it very difficult to alter to accommodate changing requirements.

Because of these difficulties, this chapter has focused on the use of a priority-based scheduling scheme. Following the description of a simple utilization-based test (which is only applicable to a restricted process model), the response time calculations were derived for a flexible model. This model can accommodate sporadic processes, process interactions, non-preemptive sections, release jitter, aperiodic servers, fault tolerant systems and an arbitrary relationship between process deadline (D) and its minimum arrival interval (T).

Interprocess synchronization, such as mutual exclusive access to shared data, can give rise to priority inversion unless some form of priority inheritance is used. Two particular protocols were described in detail in this chapter: 'original ceiling priority inheritance' and 'immediate ceiling priority inheritance'.

With priority-based scheduling, it is important that the priorities are assigned to reflect the temporal characteristic of the process load. Three algorithms have been described in this chapter:

- Rate monotonic – for $D = T$
- Deadline monotonic – for $D < T$
- Arbitrary – for $D > T$

The chapter concluded with illustrations of how fixed-priority scheduling can be accomplished in Ada, Real-Time Java and Real-Time POSIX.

Further reading

Audsley, N. C., Burns, A., Davis, R., Tindell, K. and Wellings, A. J. (1995) Fixed Priority Preemptive Scheduling: An Historical Perspective. *Real-Time Systems*, 8(3), 173–198.

Buttazzo, G. C. (1997) *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. New York: Kluwer Academic.

Burns, A. and Wellings, A. J. (1995) *Concurrency in Ada*, 2nd edn. Cambridge: Cambridge University Press.

Gallmeister, B. O. (1995), *Programming for the Real World POSIX.4*. Sebastopol, CA: O'Reilly.

Halbwachs, N. (1993) *Synchronous Programming of Reactive Systems*. New York: Kluwer Academic.

Klein, M. H. et al. (1993) *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. New York: Kluwer Academic.

Joseph, M. (ed.) (1996) *Real-time Systems: Specification, Verification and Analysis*. Englewood Cliffs, NJ: Prentice Hall.

Natarajan, S. (ed.) (1995) *Imprecise and Approximate Computation*. New York: Kluwer Academic.

Rajkumar, R. (1993) *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. New York: Kluwer Academic.

Stankovic, J. A. et al. (1998) *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. New York: Kluwer Academic.

Exercises

13.1 Three logical processes P, Q and S have the following characteristics. P: period 3, required execution time 1; Q: period 6, required execution time 2; S: period 18, required execution time 5.
Show how these processes can be scheduled using the rate monotonic scheduling algorithm.
Show how a cyclic executive could be constructed to implement the three logical processes.

13.2 Consider three processes P, Q and S. P has a period of 100 milliseconds in which it requires 30 milliseconds of processing. The corresponding values for Q and S are (5,1) and (25,5) respectively. Assume that P is the most important process in the system, followed by S and then Q.

(1) What is the behaviour of the scheduler if priority is based on importance?
(2) What is the processor utilization of P, Q and S?
(3) How should the processes be scheduled so that all deadlines are met?
(4) Illustrate one of the schemes that allows these processes to be scheduled.

13.3 To the above process set is added a fourth process (R). Failure of this process will not lead to safety being undermined. R has a period of 50 milliseconds, but has a processing requirement that is data dependent and varies from 5 to 25 milliseconds. Discuss how this process should be integrated with P, Q and S.

13.4 Figure 13.11 illustrates the behaviour of four periodic processes w, x, y and z. These processes have priorities determined by the rate monotonic scheme, with the result that priority(w) > priority(x) > priority(y) > priority(z).

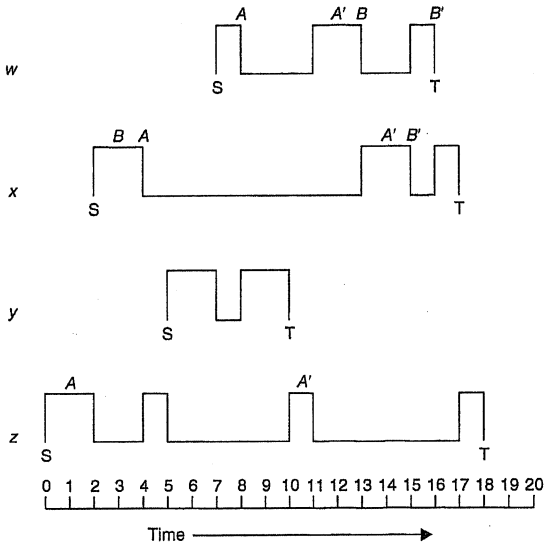

Figure 13.11 The behaviour of four periodic processes in Exercise 13.4.

Each process's period starts at time S and terminates at T. The four processes share 2 resources that are protected by binary semaphores A and B. On the diagram the tag A (and B) implies 'do a wait operation on the semaphore'; the tag A' (and B') implies 'do a signal operation on the semaphore'. Table 13.16 summaries the processes requirements:

The diagram shows the execution histories of the four processes using static priorities. For example, x starts at time 2, executes a successful wait operation on B at time 3 but unsuccessfully waits on A at time 4 (z has already locked A). At time 13 it executes again (that is, it now has lock on A), it releases A at time 14 and B at time 15. It is now preempted by Pw, but executes again at time 16. Finally it terminates at time 17.

Redraw the diagram given in Exercise 4 to illustrate the behaviour of these processes if priority inheritance is employed.

Process	Priority	Start time	Required processor time	Semaphores used
w	10	7	4	A,B
x	8	2	5	A,B
y	6	5	4	–
z	4	0	5	A

Table 13.16 Summary of the process's requirements for Exercise 13.4.

13.5 Redraw the diagram given in Exercise 4 to illustrate the behaviour of these processes if immediate priority ceiling inheritance is employed.

13.6 With the priority ceiling protocol, it is possible to calculate the maximum time any process can be blocked by the operation of a lower priority process. What is the rule for calculating this blocking? Illustrate the answer by calculating the maximum blocking time for each process in the following example. A program consists of five processes, a, b, c, d, e (these are listed in priority order with a having the highest priority), and six resources $R1$, ... $R6$ (protected by semaphores implementing the priority ceiling protocol). The resource accesses have worst case execution times given in Table 13.17.

R1	R2	R3	R4	R5	R6
50 ms	150 ms	75 ms	300 ms	250 ms	175 ms

Table 13.17 Summary of the processes' execution requirements for Exercise 13.6.

Resources are used by the processes according to Table 13.18.

Process	Uses
a	$R3$
b	$R1$,$R2$
c	$R3$,$R4$,$R5$
d	$R1$,$R5$,$R6$
e	$R2$,$R6$

Table 13.18 Summary of the processes' resource requirements for Exercise 13.6.

13.7 Is the process set shown in Table 13.19 schedulable using the simple utilization-based test given in Equation (13.1)? Is the process set schedulable using the response time analysis?

Process	Period	Execution time
a	50	10
b	40	10
c	30	9

Table 13.19 Summary of the processes' attributes for Exercise 13.7.

13.8 The process set shown in Table 13.20 is not schedulable using Equation (13.1) because a must be given the top priority due to its criticality. How can the process set be transformed so that it is schedulable. Note that the computations represented by a must still be given top priority.

Process	Period	Execution time	Criticality
a	60	10	HIGH
b	10	3	LOW
c	8	2	LOW

Table 13.20 Summary of the processes' attributes for Exercise 13.8.

13.9 The process set given in Table 13.21 is not schedulable using Equation (13.1), but does meet all deadlines when scheduled using fixed priorities. Explain why.

Process	Period	Execution time
a	75	35
b	40	10
c	20	5

Table 13.21 Summary of the processes' attributes for Exercise 13.9.

13.10 In Section 13.8, a sporadic process was defined as having a minimum inter-arrival time. Often sporadic processes come in bursts. Update Equation (13.4), to cope with a burst of sporadic activities such that N invocations can appear arbitrarily close together in a period of T.

13.11 Extend the answer given above to cope with sporadic activity which arrives in bursts, where there may be N invocations in a period of T and each invocation must be separated by at least M time units.

13.12 To what extent can the response time equations given in this chapter be applied to resources other than the CPU? For example, can the equations be used to schedule access to a disk?

13.13 In a safety-critical real-time system, a collection of processes can be used to monitor key environmental events. Typically, there will be a deadline defined between the event occurring and some output (which is in response to the event) being produced. Describe how periodic processes can be used to monitor such events.

13.14 Consider the list of events (shown in Table 13.22) together with the computation costs of responding to each event. If a separate process is used for each event (and these processes are implemented by preemptive priority-based scheduling) describe how Rate Monotonic Analysis can be applied to make sure all deadlines are met.

EVENT	Deadline	Computation time
A_Event	36	2
B_Event	24	1
C_Event	10	1
D_Event	48	4
E_Event	12	1

Table 13.22 Summary of events for Exercise 13.14.

13.15 How can the process set shown in Table 13.23 be optimally scheduled (using fixed priority scheduling)? Is this task set schedulable?

Process	T	C	B	D
a	8	4	2	8
b	10	2	2	5
c	30	5	2	30

Table 13.23 Summary of tasks for Exercise 13.15.

13.16 A real-time systems designer wishes to run a mixture of safety-critical, mission-critical and non-critical periodic and sporadic Ada tasks on the same processor. He or she is using preemptive priority-based scheduling and has used the response-time analysis equation to predict that all tasks meet their deadlines. Give reasons why the system might nevertheless fail to meet its deadlines at run-time. What enhancements could be provided to the Ada run-time support systems to help eliminate the problems?

13.17 Show how earliest deadline first scheduling can be implemented in Ada.

13.18 Outline (with code) how Ada supports sporadic tasks. How can the task protect itself from executing more often than its minimum inter-arrival time?

13.19 To what extent can sporadic servers be implemented in Ada?

13.20 Ada allows the base priority of a task to be set dynamically using the following package.

```
with Ada.Task_Identification;
with System;
package Ada.Dynamic_Priorities is

   procedure Set_Priority(Priority : System.Any_Priority;
                T : Ada.Task_Identification.Task_Id :=
                Ada.Task_Identification.Current_Task);
     -- raises Program_Error if T is the Null_Task_Id
     -- has no effect if the task has terminated

   function Get_Priority(T : Ada.Task_Identification.Task_Id :=
            Ada.Task_Identification.Current_Task)
            return System.Any_Priority;
     -- raises Tasking_Error if the task has terminated
     -- or Program_Error if T is the Null_Task_Id
private
  ... -- not required for this question
end Ada.Dynamic_Priorities;
```

Using this package, show how to implement a mode change protocol where a group of tasks must have their priorities changed as a single atomic operation.

13.21 Real-Time POSIX supports dynamic ceiling priorities, but Ada does not. Explain the pros and cons of supporting dynamic ceiling priorities.

13.22 To what extent can `ProcessingGroupParameters` be used by a Real-Time Java scheduler to support sporadic servers?

13.23 Show how earliest deadline first scheduling can be implemented in Real-Time Java with a feasibility test of total process utilization less that 100%.