Chapter 11
Scheduling real-time systems

In a concurrent program, it is not necessary to specify the exact order in which tasks 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 tasks 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 tasks 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 activity 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 on a single processor system. Here, tasks are assigned priorities such that at all times the task 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. Other scheduling approaches, such as EDF, and multiprocessor and energy issues are also covered in this chapter. The first approach to be review, however, will be the traditional scheme involving the production of a cyclic executive. All issues concerned with programming schedulable systems are covered in the next chapter.

11.1 The cyclic executive approach

With a fixed set of purely periodic tasks, it is possible to lay out a complete schedule such that the repeated execution of this schedule will cause all tasks to run at their correct rate. The cyclic executive is, essentially, a table of procedure calls, where each procedure represents part of the code for a 'task'. 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 11.1 provides a task set that must be implemented via a simple four-slot major cycle. A possible mapping onto the cyclic executive is shown in Figure 11.1, which illustrates the job that the processor is executing at any particular time.

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

Table 11.1 Cyclic executive task set.

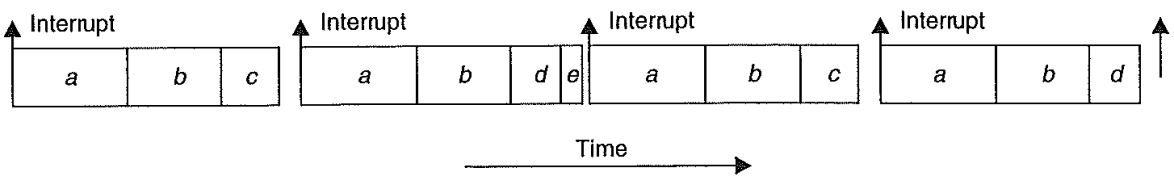

Figure 11.1 Time-line for task set.

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

- No actual tasks 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 'task' 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 tasks;
- the difficulty of incorporating tasks 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 'task' with a sizeable 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 over-full. The bin packing problem is known to be NP-hard and hence is computationally infeasible for sizeable 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 task-based scheduling schemes. These approaches will therefore be the focus in the remainder of this chapter.

11.2 Task-based scheduling

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

- runnable;
- suspended waiting for a timing event – appropriate for periodic tasks;
- suspended waiting for a non-timing event – appropriate for sporadic tasks.

11.2.1 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 task has a fixed, **static**, priority which is computed pre-run-time. The runnable tasks are executed in the order determined by their priority. *In real-time systems, the 'priority' of a task 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 tasks are executed in the order determined by the absolute deadlines of the tasks; the next task to run being the one with the shortest (nearest) deadline. Although it is usual to know the relative deadlines of each task (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 (current utilization greater than 100%) 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 task and employing an online value-based scheduling algorithm to decide which task 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 11.12.

11.2.2 Scheduling characteristics

There are a number of important characteristics that can be ascribed to a scheduling test. The two most important are **sufficiency** and **necessity**.

- A schedulability test is defined to be **sufficient** if a positive outcome guarantees that all deadlines are always met.
- A test can also be labelled as **necessary** if failure of the test will indeed lead to a deadline miss at some point during the execution of the system.

A **sufficient and necessary** test is **exact** and hence is in some sense optimal; a sufficient but not necessary test is pessimistic, but for many situations an exact test is intractable. From an engineering point of view, a tractable sufficient test with low pessimism is ideal.

A scheduling test is usually applied to the worst-case behavioural description of the application. A system is schedulable with respect to a specified scheduling policy if it will meet all its timing requirements when executed on its target platform with that

scheduling policy. A scheduling test is said to be **sustainable** if it correctly predicts that a schedulable system will be remain schedulable when its operational parameters 'improve' – for example, if a system is schedulable it should remain so if some of its tasks have their periods or deadlines increased, or their resource requirement reduced; or if the application is moved to a faster processor.

11.2.3 Preemption and non-preemption

With priority-based scheduling, a high-priority task 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 task. Alternatively, with **non-preemption**, the lower-priority task will be allowed to complete before the other executes. In general, preemptive schemes enable higher-priority tasks 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 task 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 11.10.3. 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.

11.2.4 Simple task model

An arbitrarily complex concurrent program cannot easily be analysed 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. The basic model has the following characteristics.

- The application is assumed to consist of a fixed set of tasks.
- All tasks are periodic, with known periods.
- The tasks are completely independent of each other.
- All system overheads, context-switching times and so on are ignored (that is, assumed to have zero cost).
- All tasks have deadlines equal to their periods (that is, each task must complete before it is next released).[1]
- All tasks have fixed worst-case execution times.
- No task contains any internal suspension points (e.g. an internal delay statement or a blocking I/O request).
- All tasks execute on a single processor (CPU).

[1] As the deadline is derived from the task's period it is sometimes referred to as an *implicit* deadline. It the deadline value is different from the period then the deadline is *explicit*.

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

Table 11.2 Standard notation.

One consequence of the task's independence is that it can be assumed that at some point in time all tasks will be released together. This represents the maximum load on the processor and is known as a **critical instant**. Table 11.2 gives a standard set of notations for task characteristics.

Each task is assumed to give rise to a (potentially) infinite series of executions. Each execution is known as an **invocation** (release) of the task or simply as a **job**.

11.3 Fixed-priority scheduling (FPS)

With the straightforward model outlined above, there exists a simple optimal priority assignment scheme for FPS known as **rate monotonic** priority assignment. Each task is assigned a (unique) priority based on its period: the shorter the period, the higher the priority (that is, for two tasks i and j, $T_i < T_j \Rightarrow P_i > P_j$). This assignment is optimal in the sense that if any task set can be scheduled (using preemptive priority-based scheduling) with a fixed-priority assignment scheme, then the given task set can also be scheduled with a rate monotonic assignment scheme. Table 11.3 illustrates a five task 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

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

Table 11.3 Example of priority assignment.

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.

11.4 Utilization-based schedulability tests for FPS

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 task 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 tasks will meet their deadlines (note that the summation calculates the total utilization of the task set):

$$\sum_{i=1}^{N} \left(\frac{C_i}{T_i} \right) \leq N(2^{1/N} - 1) \tag{11.1}$$

Table 11.4 shows the utilization bound (as a percentage) for small values of N. For large N, the bound asymptotically approaches 69.3%. Hence any task 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.

Table 11.5 contains three tasks that have been allocated priorities via the rate monotonic algorithm (hence task c has the highest priority and task a the lowest). Their

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

Table 11.4 Utilization bounds.

Task	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 11.5 Task set A.

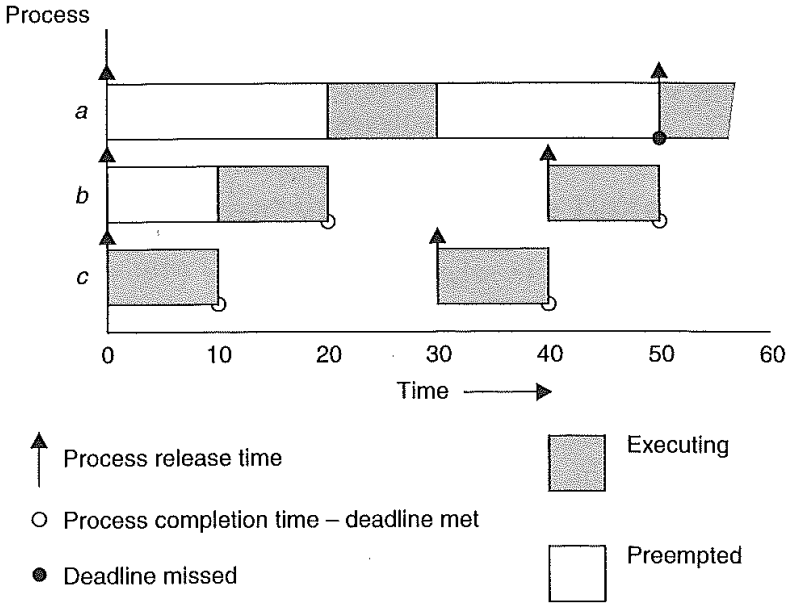

Figure 11.2 Time-line for task set A.

combined utilization is 0.82 (or 82%). This is above the threshold for three tasks (0.78), and hence this task set fails the utilization test.

The actual behaviour of this task set can be illustrated by drawing out a **time-line**. Figure 11.2 shows how the three tasks would execute if they all started their executions at time 0. Note that, at time 50, task *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 11.2 is drawn as a **Gantt chart** in Figure 11.3.

The second example is contained in Table 11.6. Now the combined utilization is 0.775, which is below the bound, and hence this task 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 surprises? For task 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 tasks meet their first deadline then they will meet all future ones.

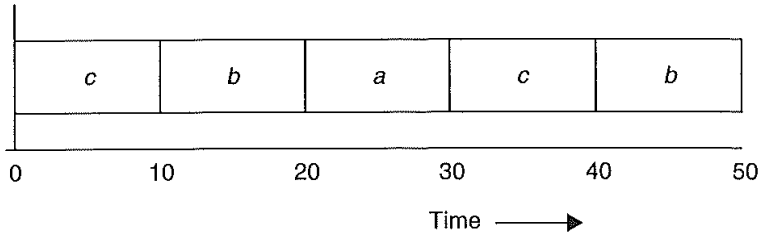

Figure 11.3 Gantt chart for task set A.

Task	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 11.6 Task set B.

Task	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 11.7 Task set C.

A final example in given in Table 11.7. This is again a three-task 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 11.4). Hence the task set fails the test, but at run-time does not miss a deadline. Therefore, the test is *sufficient* but not *necessary*. If a task 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 tasks. This is remedied in the response time approach described in Section 11.5.

11.4.1 Improved utilization-based tests for FPS

Since the publication of the Lui and Layland utilization bound a number of improvements have been developed. Here two alternative schemes are considered. First a simple re-interpretation of Equation (11.1) can be employed. Rather than the N standing for the

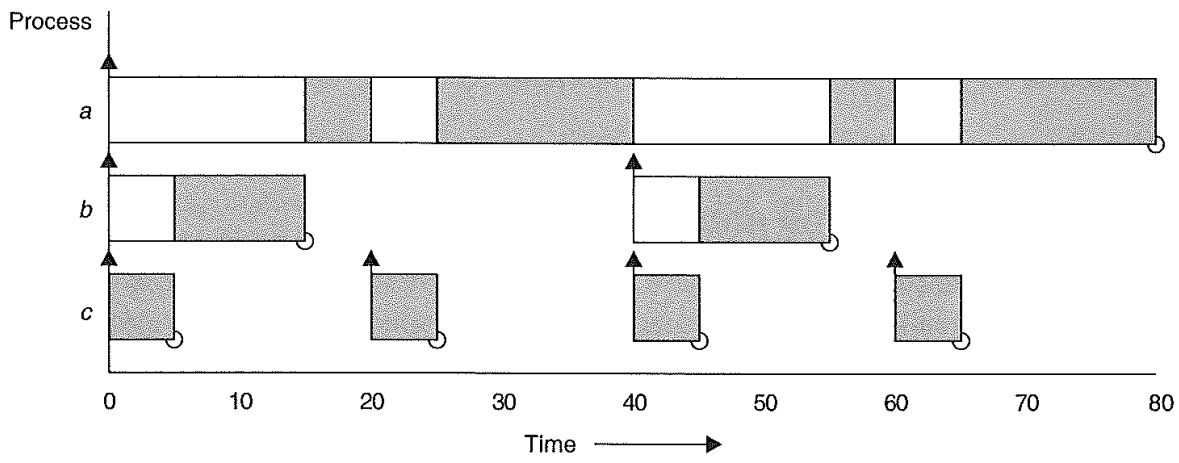

Figure 11.4 Time-line for task set C.

number of tasks, it can be defined to be the number of distinct task families in the application. A family of tasks have periods that are multiples of a common value (for example 8, 16, 64 and 128).

Consider the task sets defined earlier. For task set B (Table 11.6) there are three tasks but only two families (the 80 and 40 periods imply a single family). So the bound for this system is now 0.828 (not 0.78). The utilization of task set B is 0.775 so is below both bounds. However, if the period of task c is shortened to 14 (from 16) then the utilization of the task set rises to 0.81 (approximately) – this is above the Lui and Layland bound but below the new bound and hence this new task set is correctly deemed schedulable by this new test.

For task set C (see Table 11.7) there is an even more impressive improvement. Now there is only one family (as the periods are 80, 40 and 20). So the utilization bound is 1.0 and hence this system is schedulable by this test. Although this result shows the effectiveness of this approach there is a drawback with this test – it is not sustainable. Consider a minor change to the characteristics of this task set; let the period of task a move from 80 to 81. This alteration should make the system easier to schedule; a period has been extended and hence the overall utilization has been reduced (though only by a small amount from 1 to 0.994). But the move from 80 to 81 results in there now being two families and not just one, so the bound drops from 1 to 0.82. The new system cannot be proven to be schedulable (although it clearly is if the original task set was schedulable).

Another improved utilization-based test was developed by Bini et al. (2007) and has a different form:

$$\prod_{i=1}^{N} \left(\frac{C_i}{T_i} + 1 \right) \leq 2 \tag{11.2}$$

To give a simple example of the use of this formulation, consider again task set B (Table 11.6) with the minor modification that the period of task a is now 76 (rather than 80). The total utilization of this new system is .796 which is above the bound for three tasks, and hence schedulability is unproven. Note there are now three families so no improvement from the other approach. Applying Equation (11.2)

$$1.421 * 1.125 * 1.25 = 1.998 < 2$$

indicates that the system is schedulable by this test and, indeed, a time-line for this revised task set would show that all deadlines have been met.

11.5 Response time analysis (RTA) 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 task 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 (R) of each task. These values are then compared, trivially, with the task deadlines. This requires each task to be analysed individually.

For the highest-priority task, its worst-case response time will equal its own computation time (that is, $R = C$). Other tasks will suffer **interference** from higher-priority

tasks; this is the time spent executing higher-priority tasks when a low-priority task is runnable. So for a general task i:

$$R_i = C_i + I_i \tag{11.3}$$

where I_i is the maximum interference that task i can experience in any time interval $(t, t + R_i)$.[2] The maximum interference obviously occurs when all higher-priority tasks are released at the same time as task i (that is, at a critical instant). Without loss of generality, it can be assumed that all tasks are released at time 0. Consider one task (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. Later in this chapter floor functions are employed; they compute the largest integer smaller than the fractional part meaning that the floor of 1/3 is 0, of 6/5 is 1 and of 6/3 is again 2.

So, if R_i is 15 and T_j is 6 then there are three releases of task j (at times 0, 6 and 12). Each release of task 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 task of higher priority is interfering with task 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 tasks (than i). Substituting this value back into Equation (11.3) gives (Joseph and Pandya, 1986):

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{11.4}$$

Although the formulation of the interference equation is exact, the actual amount of interference is unknown as R_i is unknown (it is the value being calculated). Equation (11.4) 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 (11.4). The smallest such value of R_i represents the worst-case response time for the task. The simplest way of solving Equation (11.4) is to

[2]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 task can complete executing on the same tick as a higher-priority task is released.

form a recurrence relationship (Audsley et al., 1993a):

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \tag{11.5}$$

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 task if the full set has a utilization greater than 100%). Once they get bigger than the task's period, T, it can be assumed that the task will not meet its deadline. The starting value for the process, w_i^0, must not be greater than the final (unknown) solution R_i. As $R_i \geq C_i$ a safe starting point is C_i – there are, however, more efficient starting values (Davis et al., 2008).

The above analysis gives rise to the following algorithm for calculation response times:

```
for i in 1..N loop -- for each task in turn
  n := 0
  w_i^n := C_i
  loop
    calculate new w_i^(n+1) from Equation (11.5)
    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 task 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 task i. From that point, until the task completes, the processor will be executing tasks 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 task i), all higher-priority tasks 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 task 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

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

Table 11.8 Task set D.

is no solution). However, if at any point, an expanding window does not suffer an extra 'hit' from a higher-priority task then the busy period has been completed, and the size of the busy period is the response time of the task.

To illustrate how the RTA is used, consider task set D given in Table 11.8. The highest-priority task, a, will have a response time equal to its computation time (for example, $R_a = 3$). The next task will need to have its response time calculated. Let w_b^0 equal the computation time of task b, which is 3. Equation (11.5) 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 task b has been found (that is, $R_b = 6$).

The final task 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 11.5.

Consider again the task set C. This set failed the utilization-based test but was observed to meet all its deadlines up to time 80. Table 11.9 shows the response times calculated by the above method for this collection. Note that all tasks are now predicted to complete before their deadlines.

The response time calculations have the advantage that they are sufficient and necessary – if the task set passes the test they will meet all their deadlines; if they fail

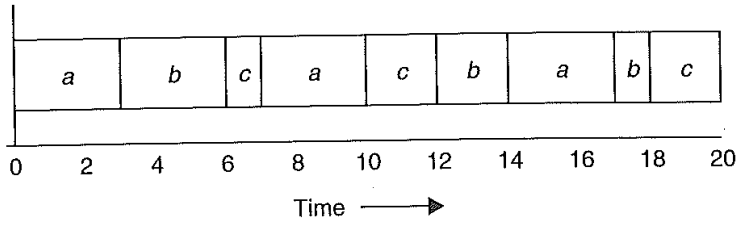

Figure 11.5 Gantt chart for task set D.

Task	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 11.9 Response time for task set C.

the test, then, at run-time, a task 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.

11.6 Sporadic and aperiodic tasks

To expand the simple model of Section 11.2.4 to include sporadic (and aperiodic) task requirements, the value T is interpreted as the minimum (or average) inter-arrival interval (Audsley et al., 1993a). A sporadic task 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 tasks demands concerns the definition of the deadline. The simple model assumes that $D = T$. For sporadic tasks, 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 tasks, it is also useful to allow the application to define deadline values less than period.

An inspection of the response time algorithm for FPS, described in Section 11.5, 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 tasks.

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

In Section 11.7, 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 $D > T$.

11.6.1 Hard and soft tasks

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

A consequence 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 task 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).

11.6.2 Aperiodic tasks and fixed-priority execution-time servers

One simple way of scheduling aperiodic tasks, within a priority-based scheme, is to run such tasks at a priority below the priorities assigned to hard tasks. In effect, the aperiodic tasks run as background activities, and therefore cannot steal, in a preemptive system, resources from the hard tasks. Although a safe scheme, this does not provide adequate support to soft tasks which will often miss their deadlines if they only run as background activities. To improve the situation for soft tasks, a **server** (or **execution-time server**) can be employed. Servers protect the tasking resources needed by hard tasks, but otherwise allow soft tasks 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 task to be introduced at the highest priority.[3] This task, the

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

server, thus has a period, T_s and a capacity C_s. These values are chosen so that all the hard tasks 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 task 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 task 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 task 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 implementation overheads. Section 12.6 describes how SS is supported by C/Real-Time POSIX; DS and SS can be analysed using response time analysis (Bernat and Burns, 1999).

As all servers limit the capacity that is available to aperiodic soft tasks, they can also be used to ensure that sporadic tasks do not execute more often than expected. If a sporadic task with inter-arrival interval of T_i and worst-case execution time of C_i is implemented not directly as a task, but via a server with $T_s = T_i$ and $C_s = C_i$, then its impact (interference) on lower-priority tasks is bounded even if the sporadic task 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 tasks (if there is a capacity);
- retain the capacity for as long as possible if there are currently no aperiodic tasks (by allowing the hard tasks 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 tasks run in the middle band. Hard tasks, 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 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.

11.7 Task systems with $D < T$

In the above discussion on sporadic tasks it was argued that, in general, it must be possible for a task 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

Task	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 11.10 Example task set for DMPO.

for $D < T$, a similar formulation could be defined – the **deadline monotonic** priority ordering (DMPO). Here, the fixed priority of a task is inversely proportional to its relative deadline: $(D_i < D_j \Rightarrow P_i > P_j)$. Table 11.10 gives the appropriate priority assignments for a simple task set. It also includes the worst-case response time – as calculated by the algorithm in Section 11.5. Note that a rate monotonic priority ordering would successfully schedule these tasks.

In the following subsection, the optimality of DMPO is proven. Given this result and the direct applicability of response time analysis to this task model, it is clear that FPS can adequately deal with this more general set of scheduling requirements. The same is not true for EDF scheduling, see Section 11.11. Once tasks can have $D < T$ then the simple utilization test (total utilization less than one) cannot be applied.

Having raised this difficulty with EDF, it must be remembered that EDF is the more effective scheduling scheme. Hence any task set that passes an FPS schedulability 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.

11.7.1 Proof that DMPO is optimal

Deadline monotonic priority ordering is optimal if any task 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 tasks (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 tasks i and j are swapped. Consider the schedulability of Q under W′.

- All tasks with priorities greater than P_i will be unaffected by this change to lower-priority tasks.
- All tasks with priorities lower than P_j will be unaffected. They will all experience the same interference from i and j.
- Task 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 task 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 task i only interferes once during the execution of j.

Once the tasks 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 tasks. Task 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 task i is schedulable after the switch.

Priority scheme W' can now be transformed (to W'') by choosing two more tasks 'that are in the wrong order for DMPO' and switching them. Each such switch preserves schedulability. Eventually there will be no more tasks to switch; the ordering will be exactly that required by DMPO and the task 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.

11.8 Task interactions and blocking

One of the simplistic assumptions embodied in the system model, described in Section 11.2.4, is the need for tasks to be independent. This is clearly unreasonable, as task interaction will be needed in almost all meaningful applications. In Chapters 5 and 6, it was noted that tasks can interact safely either by some form of protected shared data (using, for example, semaphores, monitors, synchronized methods or protected objects) or directly (using some form of rendezvous). All of these language features lead to the possibility of a task 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 task 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 task executions. The following analysis is therefore more accurate when related to asynchronous communication where tasks exchange data via protected shared resources. The majority of the material in the next two sections is concerned with fixed-priority scheduling. The issue of task interactions and EDF scheduling will be considered in Section 11.11.4.

If a task is suspended waiting for a lower-priority task 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 task having to wait for a lower-priority task) should not exist. However, it cannot, in general, be totally eliminated. Nevertheless, its adverse effects can be minimized. If a task is waiting for a lower-priority task, 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 tasks: a, b, c and d. Assume they have been assigned priorities according to the deadline monotonic scheme, so that the priority of task d is the highest and that of task a the lowest. Further, assume that tasks d and a (and tasks d and c) share a critical

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

Table 11.11 Execution sequences.

section (resource), denoted by the symbol Q (and V), protected by mutual exclusion. Table 11.11 gives the details of the four tasks 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 task c executes for four ticks; the middle two while it has access to critical section V.

Figure 11.6 illustrates the execution sequence for the start times given in the table. Task a is released first, executes and locks the critical section, Q. It is then preempted by the release of task c which executes for one tick, locks V and is then preempted by the release of task d. The higher-priority task 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.

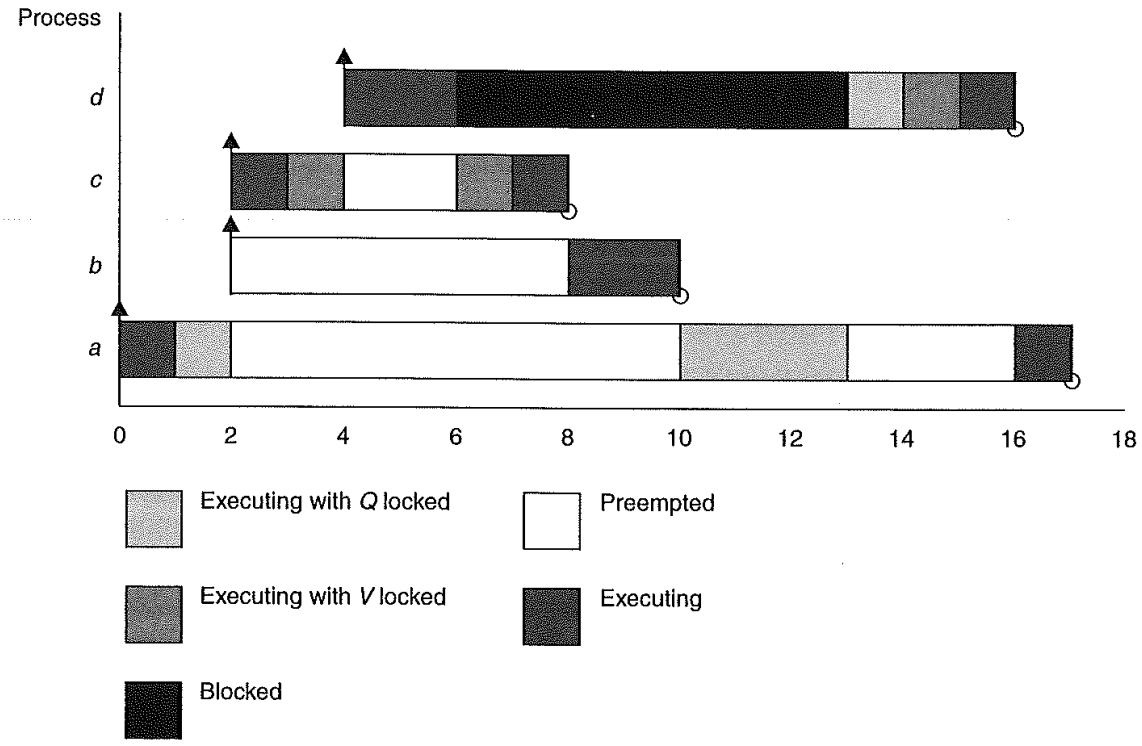

Figure 11.6 Example of priority inversion.

An inspection of Figure 11.6 shows that task d suffers considerable priority inversion. Not only is it blocked by task a but also by tasks b and c. Some 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 tasks c and b is unproductive and will severely affect the schedulability of the system (as the blocking on task d is excessive).

Priority inversion is not just a theoretical problem; real systems have been known to fail due to this phenomenon. A much publicized[4] case was that of the NASA Mars Pathfinder. Although the Sojourner rover successfully survived the bouncy landing on Mars and was able to collect meteorological data, the spacecraft initially experienced a series of total system resets resulting in lost data. Tasks on the Pathfinder spacecraft were executed as fixed-priority threads. The high-priority data bus management thread and a low-priority meteorological data gathering thread shared an 'information bus' protected by a mutex. A communications thread ran with medium priority. At run-time, the release pattern of the threads was such that the high-priority thread was waiting for the mutex to be released on the information bus, but the lower-priority thread which was using the bus and hence held the mutex lock could not make progress as it was preempted by the relatively long-running medium-priority thread. This resulted in a watchdog timer being triggered as the urgent high-priority data bus thread was missing its deadline. The watchdog initiated a total system reset. The situation then repeated itself again and again. The solution to this problem, once it was identified (which was not easy), was to turn on **priority inheritance** that was fortunately supported by the spacecraft's operating system.

With priority inheritance, a task's priority is no longer static; if a task p is suspended waiting for task q to undertake some computation then the priority of q becomes equal to the priority of p (if it was lower to start with). In the example given a little earlier, task a will be given the priority of task d and will, therefore, run in preference to task c and task b. This is illustrated in Figure 11.7. Note that as a consequence of this algorithm, task b will now suffer blocking even though it does not use a shared object. Also note that task d now has a second block, but its response time has been reduced to 9. With the Mars Pathfinder example once priority inheritance was turned on, the lower-priority thread inherited the data bus thread's priority and thus ran in preference to the medium-priority thread.

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

In general, inheritance of priority is not restricted to a single step. If task d is waiting for task c, but c cannot deal with d because it is waiting for task b then b as well as c is given d's priority.

In the design of a real-time language, priority inheritance is 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 task that will reverse this action. Inheritance is therefore not easily implemented. With synchronous

[4] See http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html.

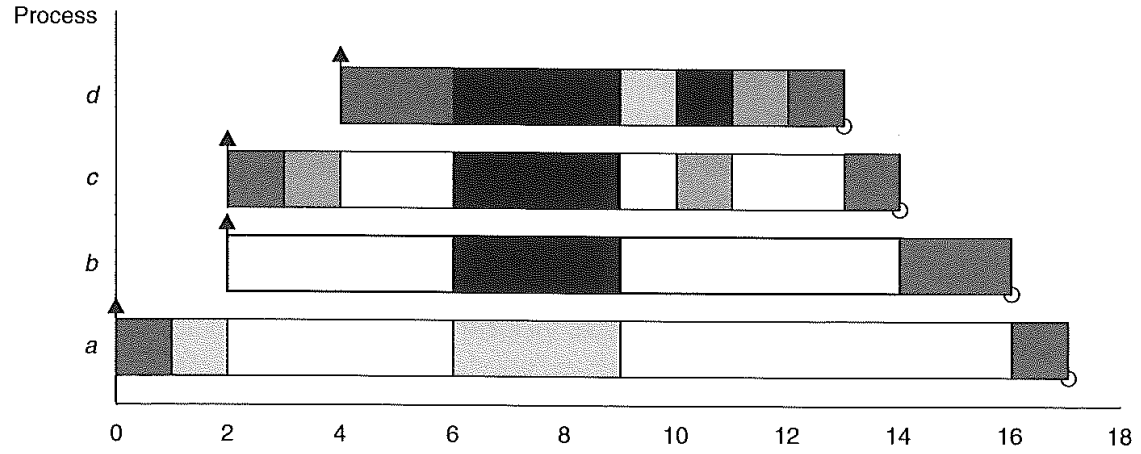

Figure 11.7 Example of priority inheritance.

message passing, indirect naming may also make it difficult to identify the task 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 task can be blocked by lower-priority tasks. If a task 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 task (this is what happened in Figure 11.7). If there are only n ($n < m$) lower-priority tasks then this maximum can be further reduced (to n).

If B_i is the maximum blocking time that task 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) in the system. Equation (11.6) thus provides an upper bound on B:

$$B_i = \sum_{k=1}^{K} usage(k, i)C(k) \tag{11.6}$$

where *usage* is a 0/1 function: $usage(k, i) = 1$ if resource k is used by at least one task with a priority less than P_i, and at least one task 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. Nested resources are not accommodated by this simple formula; they require the *usage* function to track resources that use other resources.

This algorithm is not optimal for this simple inheritance protocol. Firstly, it assumes a single cost for using the resource, it does not try to differentiate between the cost of each task's use of the resource. Secondly, it adds up the blocking from each resource, but this can only happen if each such resource is used by a different lower-priority process. This may not be possible for a particular application. For example, if all k resources are only used by one lower-priority task then there would be just one term to include in the equation for B. Nevertheless, the equation serves to illustrate the factors that need to be taken into account when calculating B. In Section 11.9, better inheritance protocols will be described and an improved formula for B will be given.

11.8.1 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:[5]

$$R = C + B + I$$

that is,

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{11.7}$$

which can again be solved by constructing a recurrence relationship:

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \tag{11.8}$$

Note that this formulation may now be pessimistic (that is, not necessarily sufficient and necessary). Whether a task actually suffers its maximum blocking will depend upon task phasings. For example, if all tasks 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 (11.7) represents an effective scheduling test for real-time systems containing cooperating tasks.

11.9 Priority ceiling protocols

While the standard inheritance protocol gives an upper bound on the number of blocks a high-priority task 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, task c being blocked by task b which is blocked by task 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 is used on a single-processor system:

- a high-priority task can be blocked at most once during its execution by lower-priority tasks;
- deadlocks are prevented;
- transitive blocking is prevented;
- mutual exclusive access to resources is ensured (by the protocol itself).

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

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 task a say, and could lead to the blocking of a higher-priority task (b), then no other resource that could block b is allowed to be locked by any task other than a. A task 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 tasks.

The original protocol takes the following form.

(1) Each task 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 tasks that use it.

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

(4) A task 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 task that uses both resources. Consequently, the maximum amount of time a task can be blocked is equal to the execution time of the longest critical section in any of the lower-priority tasks that are accessed by higher-priority tasks; that is, Equation (11.6) becomes:

$$B_i = \max_{k=1}^{K} usage(k, i)C(k) \qquad \textbf{(11.9)}$$

The benefit of the ceiling protocol is that a high-priority task can only be blocked once (per activation) by any lower-priority task. The penalty of this result is that more tasks 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 11.8 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 task sequence used in Figures 11.6 and 11.7).

In Figure 11.8, task a again locks the first critical section, as no other resources have been locked. It is again preempted by task 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 task d). At time 3, a is blocking c, and hence runs with its priority at the level 3, thereby blocking b. The higher-priority task, d, preempts a at time 4, but is subsequently blocked when it attempts to access Q. Hence a will now 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 task is only blocked once during each invocation. Figure 11.8, however, appears to show task b (and task c) suffering two blocks. What is actually happening is that a single block is being broken in two by the preemption of task d. Equation (11.9) determines that all tasks (apart from task a) will suffer a maximum single block of 4. Figure 11.8 shows that for this particular execution sequence task c and task b actually suffer a block of 3 and task d a block of only 2.

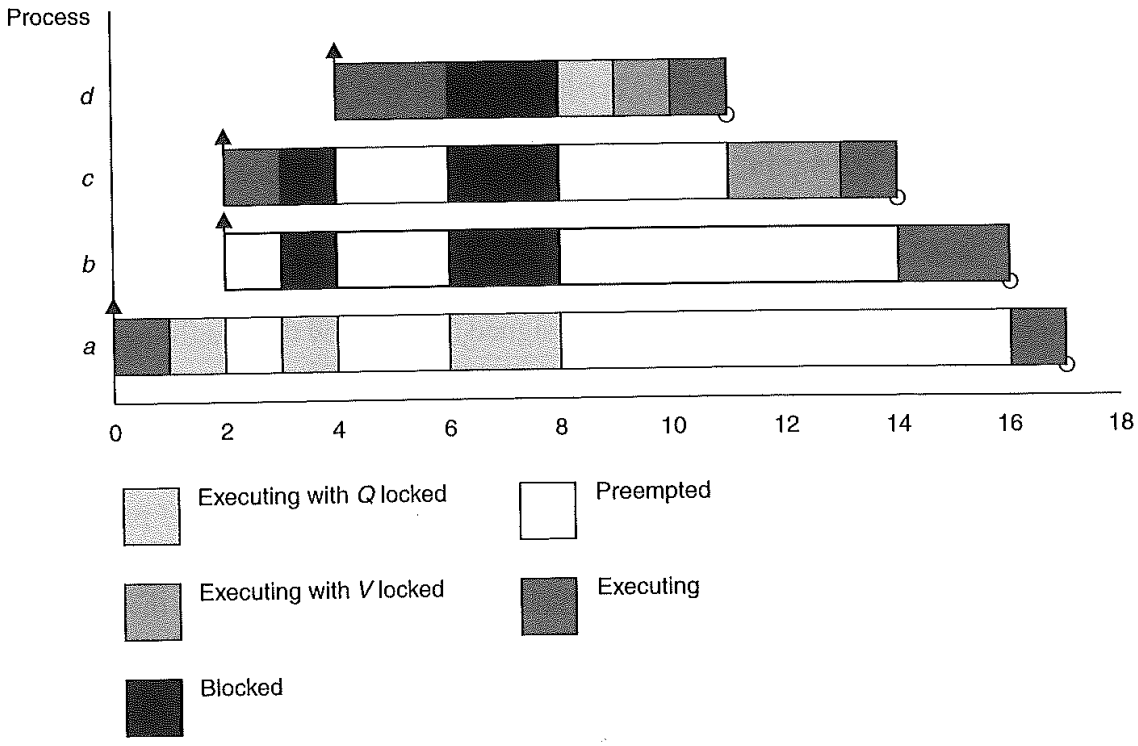

Figure 11.8 Example of priority inheritance – OCPP.

11.9.1 Immediate ceiling priority protocol

The immediate ceiling priority algorithm (ICPP) takes a more straightforward approach and raises the priority of a task as soon as it locks a resource (rather than only when it is actually blocking a higher-priority task). The protocol is thus defined as follows.

- Each task 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 tasks that use it.
- A task 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 task will only suffer a block at the very beginning of its execution. Once the task starts actually executing, all the resources it needs must be free; if they were not, then some task would have an equal or higher priority and the task's execution would be postponed. The same task set used in earlier illustrations can now be executed under ICPP (see Figure 11.9).

Task a having locked Q at time 1, runs for the next four ticks with priority 4. Hence neither task b, task c nor task d can begin. Once a unlocks Q (and has its priority reduced), the other tasks 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 (11.9)).

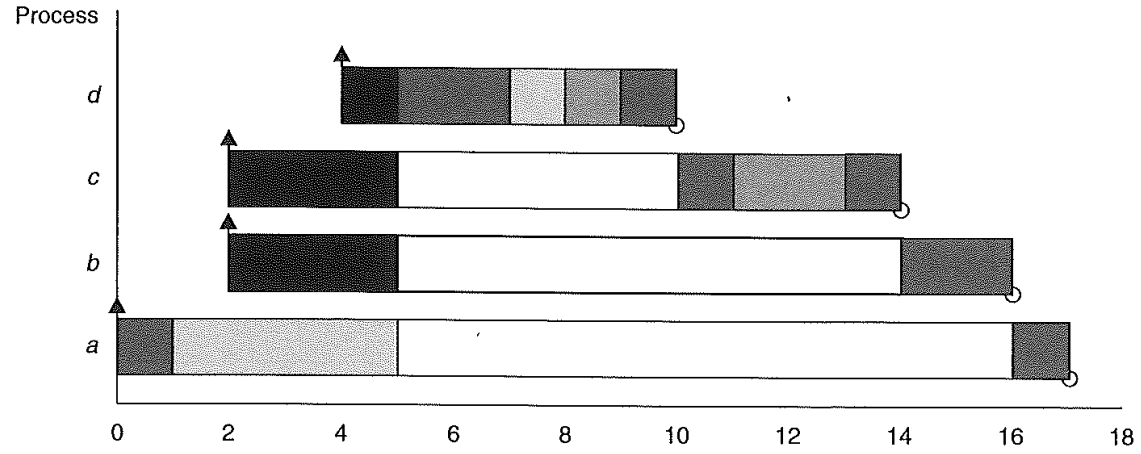

Figure 11.9 Example of priority inheritance – ICPP.

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 fewer 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 ICPP is called the Priority Protect protocol in C/Real-Time POSIX and Priority Ceiling Emulation in Real-Time Java.

11.9.2 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 and assuming the tasks do not suspend whilst holding a lock). Consider ICPP; if a task has access to some resource then it will be running with the ceiling value. No other task that uses that resource can have a higher priority, and hence the executing task will either execute unimpeded while using the resource, or, if it is preempted, the new task 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 and non-self-suspension) is that they are deadlock-free. In Section 8.7, the issue of deadlock-free resource usage was considered. The ceiling protocols are a form of deadlock prevention. If a task 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 tasks) then their ceilings must be identical. As one task is not preempted by another with merely the same priority, it follows that once

a task 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.

11.10 An extendible task model for FPS

It was noted earlier that the model outlined in Section 11.2.4 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 tasks, as well as periodic tasks, can be supported.
- Task interactions are possible, with the resulting blocking being factored into the response time equations.

Within this section, five further generalizations will be given. The section will conclude with a general-purpose priority assignment algorithm.

11.10.1 Release jitter

In the simple model, all tasks are assumed to be periodic and to be released with perfect periodicity; that is, if task l has period T_l then it is released with exactly that frequency. Sporadic tasks 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 task s being released by a periodic task l (on another processor). The period of the first task is T_l and the sporadic task will have the same rate, but it is incorrect to assume that the maximum load (interference) s exerts on low-priority tasks can be represented in Equation (11.4) or (11.5) as a periodic task with period $T_s = T_l$.

To understand why this is insufficient, consider two consecutive executions of task l. Assume that the event that releases task s occurs at the very end of the periodic task's execution. On the first execution of task l, assume that the task does not complete until its latest possible time, that is, R_l. However, on the next invocation assume there is no interference on task l so it completes within C_l. As this value could be arbitrarily small, let it equal zero. The two executions of the sporadic task are not separated by T_l but by $T_l - R_l$. Figure 11.10 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 task within 6 time units). Note that this phenomenon is of interest only if task l is remote. If this was not the case then the variations in the release of task 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 tasks have upon other tasks, the recurrence relationship must be modified. The maximum variation in a task's release is termed its **release jitter** (and is represented by J). For example, in the above, task s would have a jitter value of 15. In terms of its maximum impact on lower-priority tasks, this sporadic task 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 task i will suffer one interference from task s if R_i is between 0 and $T - J$,

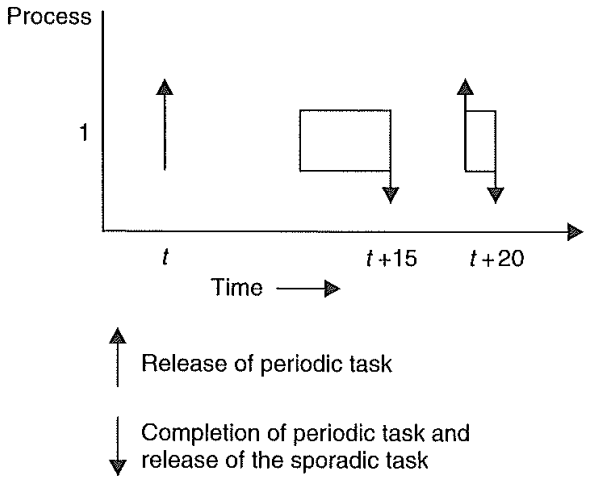

Figure 11.10 Releases of sporadic tasks.

that is $R_i \in [0, T - J)$, two if $R_i \in [T - J, 2T - J)$, 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., 1993b):

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \qquad \textbf{(11.10)}$$

In general, periodic tasks do not suffer release jitter. An implementation may, however, restrict the granularity of the system timer (which releases periodic tasks). In this situation, a periodic task 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 task 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:

$$R_i^{periodic} = R_i + J_i \qquad \textbf{(11.11)}$$

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

11.10.2 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 task. The critical instant, when all higher-priority tasks 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 task will be delayed until any previous releases of the same task have completed.*

If a task executes into the next period then both releases must be analysed 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 (11.5) can be extended to have the following form (ignoring release jitter) (Tindell et al., 1994):

$$w_i^{n+1}(q) = B_i + (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j \tag{11.12}$$

For example, with q equal to 2, three releases of the task will occur in the window. For each value of q, a stable value of $w(q)$ can be found by iteration – as in Equation (11.5). The response time is then given as:

$$R_i(q) = w_i^n(q) - qT_i \tag{11.13}$$

For example, with $q = 2$ the task 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:

$$R_i(q) \leq T_i \tag{11.14}$$

At this point, the task 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:

$$R_i = \max_{q=0,1,2,...} R_i(q) \tag{11.15}$$

Note that for $D \leq T$, the relation in Equation (11.14) is true for $q = 0$ (if the task can be guaranteed), in which case Equations (11.12) and (11.13) simplify back to the original equation. If any $R > D$, then the task 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 tasks suffer release jitter:

$$w_i^{n+1}(q) = B_i + (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q) + J_j}{T_j} \right\rceil C_j \tag{11.16}$$

The other change involves the task 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 (11.13) must be altered:

$$R_i(q) = w_i^n(q) - qT_i + J_i \tag{11.17}$$

11.10.3 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 analysed by the scheduling technique based on response time analysis. In Equation (11.7), for example, there is a blocking term B that accounts for the time a lower-priority task may be executing while a higher-priority task is runnable. In the application domain, this may be caused by the existence of data that is shared (under mutual exclusion) by tasks 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 task being delayed by the time it takes to context switch to a lower-priority task – even though an immediate context switch to the higher-priority task 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 task cannot be blocked both by an application task and a kernel routine – only one could actually be happening when the higher-priority task 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 task is now runnable the kernel will instigate a context switch; if not, the currently running task will continue into the next non-preemptive block.

The normal execution of the application code is thus totally cooperative. A task 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 task if any non-preemptive block lasts longer than B_{MAX} (see Chapter 13).

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 (11.4), as the value of w is being extended, new releases of higher-priority tasks 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 task has consumed $C_i - F_i$ time units, the last block has (just) started. Equation (11.4) is now solved for $C_i - F_i$ rather than C_i:

$$w_i^{n+1} = B_{\text{MAX}} + C_i - F_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \qquad \textbf{(11.18)}$$

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

$$R_i = w_i^n + F_i \qquad \textbf{(11.19)}$$

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

This straightforward application of response time analysis is, however, misleading and may in certain circumstances lead to errors – that is, the analysis is not sufficient. Consider as a simple example a two task system with each task having deadline equal to period. The first task has a period of 6 and a computation time of 2 which is executed as a single non-preemptive block. The other task has a period of 8 and an execution time of 6 split into two 3 unit blocks. The longer period for this task means that it has the lower priority. The first task has a blocking term of 3 which, with its own computation time of 2, gives a response time of 5. The second task is first analysed to see when its first block will complete. This has a computation time of 3 and suffers 2 units of interference and so w_i^n converges simple to the value 5. To this is added the F_i value of 3 to give an overall response time of 8. This appears to imply that the system is schedulable. But this is impossible – the overall utilization of these two tasks is greater than 1 (1/3 + 3/4) which is indisputable evidence of unschedulability.

So why does the analysis fail on this example? There is a constraint on using Equations (11.18) and (11.19) that is hidden and this example highlights the problem because it does not satisfy this constraint. For these equations to apply, the worst-case response time for each task *with preemption* must be less than the task's period. If this is not the case then it is possible for the second (or third ...) release of the task to be the worst. If releases overlap in this way then the analysis used in the previous section for deadline greater than period must be used.

For the example, the preemptive worst-case response time of the second task is 10 (two interferences plus execution time of 6) which is greater than 8 and hence the second release must be analysed. The easiest method for computing this is to look at the worst-case response time of a task made up of two serial executions of the second task. Now this new task has a computation time of 12 made up of four 3 unit blocks. Applying Equations (11.18) and (11.19) gives a value of w_i^n of 15; when the final 3 is added in this gives a response time of 18 which breaks the deadline value of 16 (for the second invocation).

It must be emphasized that for most systems with utilization not greater than 1, releases will not overlap and the straightforward use of these equations will provide the correct result – but the constraint must always be checked.

The other advantage of deferred preemption comes from predicting more accurately the execution times of a task's non-preemptable basic blocks. Modern processors have caches, prefetch queues and pipelines that all significantly reduce the execution times of 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 postponing a context switch is high, this will militate against these advantages.

11.10.4 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 known as the **fault model**. If C_i^f is the extra

computation time that results from an error in task i, then the response time equation can easily be changed:

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} C_k^f \tag{11.20}$$

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

Here, the fault model defines a maximum of one fault and there is an assumption that a task will execute its recovery action at the same priority as its ordinary computation. Equation (11.20) is easily changed to increase the number of allowed faults (F):

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} FC_k^f \tag{11.21}$$

Indeed, a system can be analysed 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:

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} \left(\left\lceil \frac{R_i}{T_f} \right\rceil C_k^f \right) \tag{11.22}$$

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

In Equations (11.21) and (11.22), the assumption is made that in the worst case, the fault will always occur in the task that has the longest recovery time.

11.10.5 Introducing offsets

In the scheduling analysis presented so far in this chapter, it has been assumed that all tasks share a common release time. This critical instant is when all tasks are released simultaneously (this is usually taken to occur at time 0). For fixed-priority scheduling, this is a safe assumption; if all tasks meet their timing requirements when released together then they will always be schedulable. There are, however, sets of periodic tasks that can benefit from explicitly choosing their release times so that they do not share a critical instant. This may result in improved schedulability. One task is said to have an **offset** with respect to the others. Consider for illustration the three tasks defined in Table 11.12.

If a critical instant is assumed then task a has response time of 4 and task b has a response time of 8, but the third task has a worst-case response time of 16, which is

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

Table 11.12 Example of a task set.

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

Table 11.13 Response time analysis of the task set.

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

Table 11.14 Response time analysis of the transformed task set.

beyond its deadline. For task c the interference from task b is sufficient to force a further interference from a, and this is crucial. However, if task 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 task set – see Table 11.13.

Unfortunately, task sets with arbitrary offsets are not amenable to analysis. It is a strongly NP-hard problem to choose offsets so that a task set is optimally schedulable. Indeed, it is far from trivial to even check if a set of tasks with offsets share a critical instant.[6]

Notwithstanding this theoretical result, there are task sets that can be analysed in a relatively straightforward (although not necessarily optimal) way. In most realistic systems, task periods are not arbitrary but are likely to be related to one another. As in the example just illustrated, two tasks have a common period. In these situations it is easy to give one an offset (of $T/2$) and to analyse the resulting system using a transformation technique that removes the offset – and hence critical instant analysis applies. In the example, tasks b and c (c having the offset of 10) are replaced by a single notional task with period 10, computation time 4, deadline 10 but no offset. This notional task has two important properties.

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

These properties follow from the observation that the notional task always uses more (or equal) CPU time than the two real tasks. Table 11.14 shows the analysis that would apply to the transformed task set. The notional task is given the name 'n' in this table.

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

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

$$T_n = T_a/2 \quad (\text{or } T_b/2 \text{ as } T_a = T_b)$$

$$C_n = \text{Max}(C_a, C_b)$$

$$D_n = \text{Min}(D_a, D_b)$$

$$P_n = \text{Max}(P_a, P_b)$$

where P denotes priority.

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

In Section 10.5 offsets are used to control input and output jitter. Typically the input and output activities involve much less computation time than the 'middle' task that implements whatever algorithms are necessary to convert the input value to an output setting. To analyse this program structure it is acceptable to ignore offsets. As noted earlier a system that is schedulable when offsets are ignored remains schedulable when they are added to the implementation scheme.

11.10.6 Other characteristics

In addition to the characteristics discussed in the last few sections (e.g. release jitter, non-preemption, fault tolerance, arbitrary deadlines and offsets) there are many other task attributes that have been analysed in the fixed-priority scheduling literature. For example, tasks with precedence, tasks that must meet N in M deadlines (e.g. 4 in 5) but not every deadline, and tasks that have a set of C values (not just a single maximum). It is not necessary, however, to cover all these topics (and more) in order to complete this treatment of RTA. The key property of RTA is that it is extendable and configurable. New characteristics can be easily accommodated into the theory.

11.10.7 Priority assignment

The formulation given for arbitrary deadlines has the property that no simple algorithm (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 task (Audsley et al., 1993b).

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

The proof of this theorem comes from considering the schedulability equations – for example, Equation (11.12). If a task has the lowest priority, it suffers interference from all

higher-priority tasks. This interference is not dependent upon the actual ordering of these higher priorities. Hence if any task 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 task set. Hence through successive reapplication, a complete priority ordering is obtained (if one exists).

The following code in Ada implements the priority assignment algorithm;[7] `Set` is an array of tasks that is notionally ordered by priority; `Set(N)` being the highest priority, `Set(1)` being the lowest. The procedure `Task_Test` tests to see whether task `K` is feasible at that place in the array. The double loop works by first swapping tasks into the lowest position until a feasible result is found; this task 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 task, the whole procedure is abandoned. Note that a concise algorithm is possible if an extra swap is undertaken.

```
procedure Assign_Pri (Set : in out Task_Set; N : Natural;
                      Ok : out Boolean) is
begin
  for K in 1..N loop
    for Next in K..N loop
      Swap(Set, K, Next);
      Task_Test(Set, K, Ok);
      exit when Ok;
    end loop;
    exit when not Ok;  -- failed to find a schedulable task
  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. Where there is blocking, the priority ceiling protocols ensure that blockings are relatively small and, therefore, the above algorithm produces adequate near optimal results.

11.10.8 Insufficient priorities

In all of the analysis presented in this chapter it has been assumed that each task has a distinct priority. Unfortunately it is not always possible to accommodate this 'one priority per task' ideal. If there are insufficient priorities then two or more tasks must share the same priority. Fortunately, to check the schedulability of shared-priority tasks requires only a minor modification to the response time test. Consider the basis Equation (11.4) derived earlier in this chapter, which has a summation over all the higher-priority tasks. If tasks share priorities then this summation must be over all higher- *or equal*-priority tasks:

$$R_i = C_i + \sum_{j \in hep(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{11.23}$$

where $hep(i)$ is the set of higher- or equal-priority tasks (than i).

[7] This algorithm has become known as *Audsley's algorithm*.

So, if tasks a and b share priority i then a assumes it is getting interference from b and b assumes it is getting interference from a. Clearly if a and b are schedulable when they share priority i then they will remain schedulable if they are assigned distinct but adjacent priorities. The converse is, however, not true.

One way to reduce the number of priority levels required for a specific system is to first make sure the system is schedulable with distinct priorities. Then, starting from the lowest priority, tasks are grouped together until the addition of an extra task breaks schedulability. A new group is then started with this task, and the process continues until all tasks are in groups (although some groups may contain only have a single task). A minor variant of the priority assignment algorithm given above can easily implement this scheme.

Reducing the number of priority levels inevitably reduces schedulability. Tests have shown (Klein et al., 1993) that 92% of systems that are schedulable with distinct priorities will remain schedulable if only 32 levels are available. For 128 priority values this rises to 99%. In will be noted in the next chapter that Ada requires a minimum of 31 distinct priorities, Real-Time POSIX a minimum of 32 and Real-Time Java a minimum of 28.

11.10.9 Execution-time servers

Finally, in the description of fixed-priority scheduling the topic of execution-time servers is revisited. As applications and hardware platforms become more complicated it is useful to employ a virtual resource layer between the set of applications and the processor (or processors) they execute on. An execution-time server both guarantees a certain level of service and ensures that no more resource is allocated than is implied by the 'service contract'. So, for example, two multithreaded applications may co-exist on the one processor. One application receives 4 ms every 10 ms, the other 6 ms. These levels are guaranteed and policed. The first application will definitely get 4 ms, but it will not be allocated more than 4 ms in a 10 ms interval even if it has a runnable high-priority task.

There have been a number of execution-time servers proposed for FPS (see Section 11.6.2). Here three common ones are described: the **Periodic Server**, the **Deferrable Server** and the **Sporadic Server**. The simple Periodic Server has a budget (capacity) and a replenishment period. Following replenishment, client tasks can execute until either the budget is exhausted or there are no longer any runnable client tasks. The server is then suspended until the next replenishment time. The Deferrable Server is similar except that the budget remains available even after clients have been satisfied – a client arriving late will be serviced if there is budget available. Both Periodic and Deferrable Servers are replenished periodically and the budget still available at replenishment is lost. The Sporadic Server behaves a little differently. The budget remains indefinitely. When a client arrives (at time t, say) it can use up the available budget which is then replenished at time t + the replenishment period of the server.

A complete system can contain a number of servers of these three types. The Periodic Server is ideally suited for supporting periodic tasks, the Sporadic Server is exactly what is required for sporadic tasks and the Deferrable Server is a good match for handling aperiodic work. In the latter case, aperiodic tasks can be handled quickly if

there is budget available – but once this is exhausted then the aperiodic tasks will not be serviced and hence an unbounded load on the server will have no detrimental effects on other parts of the system.

Scheduling these three server types on a fixed-priority system is relatively straightforward. Each server, of whichever type, is allocated a distinct priority. Response time analysis is then used to verify that all servers can guarantee their budget and replenishment period. Fortunately Periodic and Sporadic Servers behave exactly the same as periodic tasks and hence the straightforward analysis for these servers is directly applicable. For Deferrable Servers, the worst-case impact such a server can have on lower-priority tasks occurs when its budget is used at the very end of one period and then again at the start of the next. Conveniently this behaviour is identical to a periodic task suffering release jitter and hence can be analysed using the formulation given in Section 11.10.1.

It follows from this brief discussion that the schedulability test for a task running on a server involves two steps; first to verify that the server's parameters are valid and second that the response time of the task on that server is bounded by the task's deadline. The worst-case response time for a task executing on a server can be computed in a number of ways. For example, a server that guarantees 2 ms every 10 ms is equivalent to a processor running at 1/5 of its original speed. If all task computation times are multiplied by 5 then standard RTA can be applied (using these new C values).[8] A similar approach is taken with variable speed processors – an example of this analysis is given in Section 11.15.

11.11 Earliest deadline first (EDF) scheduling

FPS is undoubtedly the most popular scheduling approach available to the implementors of real-time systems. The next chapter will show how it is supported in a number of languages and operating systems. However, as discussed at the beginning of this chapter, it is not the only approach studied in the real-time scheduling community. This section focuses on an alternative approach, EDF, that has a number of properties that make it almost as important as FPS. Unfortunately, it is currently less supported by languages and operating systems (again see the next chapter). For this reason EDF analysis is not covered here to the same level of detail afforded to FPS analysis.

11.11.1 Utilization-based schedulability tests for EDF

Not only did the seminal paper of Liu and Layland introduce a utilization-based test for FPS but it also gave one for EDF. The following equation is for the simple task model introduced in Section 11.2.4 – in particular, $D = T$ for all tasks:

$$\sum_{i=1}^{N} \left(\frac{C_i}{T_i} \right) \leq 1 \tag{11.24}$$

[8] With this example, the computed response times may need to have the value 8 added to take into account the 'dead time' before the sever can respond to requests from its client tasks.

Clearly this is a much simpler test than the corresponding test for FPS (Equation (11.1)). As long as the utilization of the task set is less than the total capacity of the processor then all deadlines will be met (for the simple task model). In this sense EDF is superior to FPS; it can always schedule any task set that FPS can, but not all task 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 task-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 tasks without deadlines into FPS (by merely assigning them a priority); giving a task 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, for example, the criticality of the task.
- During overload situations (which may be a fault condition) the behaviour of FPS is more predictable (the lower-priority tasks 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 tasks miss deadlines. This is considered again in Section 11.12.
- 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. Indeed it is easy to show that if a task set, with restrictions such as deadline equal to period removed, is schedulable by any scheme then it will also be schedulable by EDF. The proof of this property follows the pattern used for proving that DMPO is optimal (see Section 11.7.1). Starting with the feasible schedule it is always possible to transform the schedule to one that becomes identical with the one EDF would produce – and at each transformation schedulability is preserved.

11.11.2 Processor demand criteria for EDF

One of the disadvantages of the EDF scheme is that the worst-case response time for each task does *not* occur when all tasks are released at a critical instant. In this situation only tasks with a shorter relative deadline will interfere. However, later there may exist a position in which all (or at least more) tasks have a shorter absolute deadline. In situations where the simple utilization-based test cannot be applied (for example when there is release jitter or when deadlines are shorter then periods) then a more sophisticated scheduling test must be used. In FPS this takes the form of RTA (calculate the worst-case response time for each task and then check that this is less than the related deadline). For EDF this approach can again be used, but it is much more complicated to calculate these response time values and hence it will not be described here. There is, however,

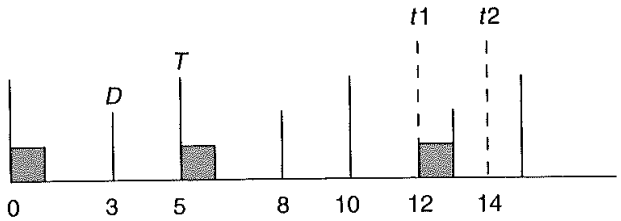

Figure 11.11 PDC example.

an alternative scheme that checks for schedulability directly rather than via response times. This method, called PDC (Processor Demand Criteria) (Baruah et al., 1990a, b), is defined as follows.

Assuming a system starts at time 0 and all tasks arrive at their maximum frequency, at any future time, t, it is possible to calculate the load on the system, $h(t)$. This is the amount of work that must be completed before t, in other words, all jobs that had absolute deadlines before (or at) t. It is easy to give a formula for $h(t)$:

$$h(t) = \sum_{i=1}^{N} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor C_i \tag{11.25}$$

To illustrate this formula, consider a single task with $T = 5$, $D = 3$ and $C = 1$, and two possible values of t: $t1 = 12$ and $t2 = 14$. Figure 11.11 illustrates these parameters. The third deadline of the task is at 13 and hence is after $t1$ so $h(t1)$ for this task should be 2. But at $t2 = 14$ another execution of the task must be completed and so $h(t2) = 3$. To compute these values easily, $T - D$ is added to t; if this results in a value after the next period then the floor function in Equation (11.25) will correctly add an extra C to the total. So, in the example, $T - D = 2$, $t1 + 2 = 14$ and hence $\lfloor 14/5 \rfloor = 2$. But $t2 + 2 = 16$ and so $\lfloor 16/5 \rfloor = 3$.

The requirement for schedulability is that the load must never exceed the time available to satisfy that level of load:

$$\forall t > 0 \;\; h(t) \leq t \tag{11.26}$$

PDC involves applying this equation to a limited number of t values. The number of points is limited by two factors:

- only values of t that correspond to deadlines of tasks need be checked;
- there is an upper bound (L) on the values of t that must be checked – this means that an unschedulable system will have $h(t) > t$ for some value of $t < L$.

The first reduction comes from the fact that $h(t)$ is constant between deadlines and hence the worst case occurs at a deadline. To calculate the upper bound (L) on the interval that must be checked, two formulae have been developed. The first one comes from the need to check at least the first deadline of each task, and a bound based on utilization (the derivation of all the equations given in the section can be found in the

literature referenced in the Further Reading section at the end of this chapter):

$$L_a = \max\left\{D_1, ..., D_N, \frac{\sum_{i=1}^{N}(T_i - D_i)C_i/T_i}{1-U}\right\}$$

The second bound is derived from the busy period of the system (that is, the time from start-up at time 0 to the first null or background tick where no task is executing – at this time, by definition, the load has been satisfied). This is easily obtained from a recurrence relation similar to that used in FPS:

$$w^0 = \sum_{i=1}^{N} C_i$$

$$w^{j+1} = \sum_{i=1}^{N} \left\lceil \frac{w^j}{T_i} \right\rceil C_i$$

When $w^{j+1} = w^j$ then this is the end of the busy period and $L_b = w^j$. Note this busy period is bounded if the utilization of the task set is not greater then 1 (so this is always checked first).

To obtain the least upper bound, the simple minimum of these two values is used:

$$L = \min(L_a, L_b) \quad (11.27)$$

For example, consider a three task system as depicted in Table 11.15. The utilization of this task set is 0.92. The computed values of L_a and L_b are 30.37 and 15 respectively; hence the least upper bound is 15. In the time period from 0 to 15 there are five deadlines to check – task a at times 4, 8 and 12, task b at time 10 and task c at time 14. At all of these points Equation (11.26) is satisfied and the system is determined to be schedulable. For example $h(15)$ is 14 and $h(12)$ is 6.

If the example is now modified to increase the computation time of b to 4 then the utilization is still acceptable (0.987), but it is not schedulable. At time 14, $h(t)$ has the value 15, so $h(14) > 14$.

11.11.3 The QPA test

For non-trivial systems, L can be large and the number of deadlines between 0 and L that need to be checked becomes excessive. Fortunately an efficient scheme has recently been developed that can significantly reduce the number of time points that need to be tested. This scheme, known as QPA (Quick Processor-demand Analysis), exploits the

Task	T	D	C
a	4	4	1
b	15	10	3
c	17	14	8

Table 11.15 A task set for EDF.

Task	T	D	C
a	4	4	1
b	8	6	1
c	10	10	1
d	12	8	2
e	15	12	2
f	21	20	3

Table 11.16 A task set for EDF.

following property (Zhang and Burns, 2008): rather than progress from 0 to L checking each deadline, QPA starts at L and moves backwards towards 0 checking only a necessary subset of the deadlines.

Let $h(L) = s$. If $s > L$ then the system is unschedulable. If this is not the case ($s \leq L$) then $h(t) < t$ for all values of t: $s < t < L$. Hence there is no need to check the deadlines within the interval $s..L$. To verify this property assume (in order to construct a counterexample) a value t within the range has $h(t) > t$. Now $t > s$ so $h(t) > s$. Also $h(t) < h(L)$ as $t < L$ (the function h is monotonic in t). We must conclude that $s > L$ which contradicts the assumption that $s < L$.

Having jumped from L back to $h(L)$, the procedure is repeated from $h(L)$ to $h(h(L))$, etc. At each step the essential test of $h(t) < t$ is undertaken. Of course if $h(t) = t$ then no progress can be made and it is necessary to force progress by moving from t to the largest absolute deadline (d) in the system such as $d < t$.

The QPA test looks at only a small fraction of the number of points that would need to be analysed if all deadlines were checked. An example of the approach is as follows. Six tasks have the characteristics given in Table 11.16. The utilization of this task set is 0.965. The value of L is 59 and there are 34 deadlines that need to be checked in this interval using PDC.

Applying QPA results in just 14 points that need to be considered, and these correspond to the following values of t: 59, 53, 46, 43, 40, 33, 29, 24, 21, 19, 12, 9, 5 and 1. In other examples (Zhang and Burns, 2008), QPA typically requires only 1% of the effort of the original processor demand analysis scheme.

11.11.4 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 task requires a resource that is currently locked by another task with a longer deadline. Not surprisingly inheritance and ceiling protocols have been developed for EDF but, as with earlier comparisons, the EDF schemes are somewhat more complex (Baruah, 2006).

As priorities are static, it is easy to determine which tasks can block the task currently being analysed. With EDF, this relationship is dynamic; it depends on which tasks (with longer deadlines) are active when the task is released. And this varies from one release to another throughout 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 task, under SRP, is assigned a **preemption level**. *Preemption levels reflect the relative deadlines of the tasks, the shorter the deadline the higher the preemption level;* so they actually designate the static priority of the task as assigned by the deadline monotonic scheme. At run-time, resources are given ceiling values based on the maximum preemption level of the tasks that use the resource.

> *When a task is released, it can only preempt the currently executing task if its absolute deadline is shorter and its preemption level is higher than the highest ceiling of the currently locked resources.*

The result of applying this protocol is identical to applying ICPP (on a single processor). Tasks suffer only a single block (it is as they are released), deadlocks are prevented and a simple formula is available for calculating the blocking time. The blocking term, once calculated, can be incorporated into PDC and QPA.

11.11.5 Aperiodic tasks and EDF execution-time 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 and a Dynamic Deferrable Server. 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 addition to these common forms of servers there are also a number that are EDF-specific. These take the form of virtual (but slow) processors that can guarantee C in T. So an aperiodic task that requires to execute for $3C$ will be guaranteed to complete in $3T$ if the server has no other work to do. To find more about EDF servers and EDF scheduling in general, the reader is referred to the books by Liu and Buttazzo in the Further Reading section at the end of this chapter.

11.12 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 tasks with known periods or worst-case arrival intervals);
- bounded computation times;
- a scheduling scheme that leads to predictable execution of the application tasks.

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 objective 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 task misses its deadline but uses sufficient resources to result in the next task also missing its deadline.

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

(1) an admissions control procedure that limits the number of tasks that are allowed to compete for the processors;

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

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

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

- **Static** – the task always has the same value whenever it is released.
- **Dynamic** – the task's value can only be computed at the time the task 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 task 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 task arrives, the complete set of tasks could be subject to an exact test such as those described in this chapter. If the task set is not schedulable, the lowest value task 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 consider 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 essentially static systems, value-added computations, in the form of soft or firm tasks that improve the quality of the hard tasks, are an attractive way of structuring systems. In these circumstances, as was noted in Section 11.6.1, the hard tasks must be protected from any overload induced by the behaviour of the non-hard tasks. One way of achieving this is to use FPS for the hard tasks and execution-time servers for the remaining work. The servers can be executed at a given priority level, but can embody whatever admissions policy is desirable and service the incoming dynamic work using EDF.

11.13 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 task is known. This is the maximum any task invocation/release (i.e. job) could require.

Worst-case execution time estimation (represented by the symbol C but also known by the acronym WCET) 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, branch prediction, out-of-order execution, memory wait states and so on) must be available.

Most analysis techniques involve two distinct activities. The first takes the task 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 task is to be analysed 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 analysable.

The biggest challenge facing worst-case execution time analysis comes from the use of modern processors with multicores, 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 putting 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.

This brief discussion has only addressed a few of the issues involved with WCET estimation. A comprehensive coverage would perhaps triple the size of this chapter. Interested readers are referred to the Further Reading section of this chapter.

11.14 Multiprocessor scheduling

All the analysis presented so far in this chapter has been concerned with the scheduling of concurrent tasks on to a single computer. As processors have become more powerful, the size and complexity of the applications that can fit on to a single processor have increased substantially. However, for a significant class of system, there is the need to move to a multiprocessor execution platform. This can take the form of a relatively simple dual processor or may involve a large multicore chip. The processors may all be of an identical type (homogeneous) or may have quite different characteristics (heterogeneous). Also the links between the processors may be a shared memory bus or an

Task	T	D	C
a	10	10	5
b	10	10	5
c	12	12	8

Table 11.17 Example task set.

independent network. Whatever the architecture, the scheduling problem for a multiprocessor system is significantly more complicated than the single processor case. Here three issues are addressed, the placement of tasks to processors, the scheduling of any shared network and the implementation of locks in shared-memory multiprocessor architectures.

11.14.1 Global or partitioned placement

The first new issue that must be addressed with a multiprocessor platform is placement. This is the mapping of tasks to processors. Two schemes are possible: **global** and **partitioned**. As the names imply, a partitioned scheme is a pre-run-time allocation; the dynamic global alternative allocates tasks as they become runnable, and even during execution a task may move from one processor to another. Both placement schemes have their advantages and disadvantages. The partitioned approach has the benefit of not requiring any run-time support and of being able to cater for certain types of systems that the global scheme has difficulty with. Consider a simple three task system that is to be implemented on two identical processors (see Table 11.17).

The total utilization of the system is 1.66 so it is capable of being scheduled (as $1.66 < 2$). A simple global placement scheme using either fixed priority or EDF would allocate a and b to the two processors. They would then execute until time 5. Now c can be placed on either processor as they are both free, but there are only 7 units of time left before the deadline at time 12. Hence c will fail to meet its deadline.

The partitioned approach would allocate a and b to one processor (where they will completely utilize its capacity), leaving the other processor to c which can then easily meet its deadline. A different example, however, will show the benefits of the global scheme. Table 11.18 also has a three task / two processor configuration. No partitioned scheme can schedule this system. However, a global scheme that starts with d and f for one tick and then executes d and e for the next eight ticks will be able to run f and e for the 10th tick and satisfy all requirements.

Task	T	D	C
d	10	10	9
e	10	10	9
f	10	10	2

Table 11.18 A further example task set.

The only way to fit this example into a partitioned scheme is to split f into two identical length parts. This artificial decomposition of a software model was criticized earlier within the context of cyclic executives and cannot be recommended.

The main challenge for the partitioned approach is to generate a valid allocation. Once an allocation has been formed, it is straightforward to analyse each processor in turn using the analysis already available for single processors. Each processor can be scheduled via fixed priorities or EDF – it would even be possible to mix the schemes with EDF on some processors and FPS on others. An allocation must not 'overfill' any processor and hence task mapping is similar to the classic bin-packing problem. Optimal schemes for large numbers of tasks and processors are therefore not possible – heuristics must be employed.

The challenge for the global scheme is to identify the optimal scheduling policy. For a single processor, EDF is optimal, but no optical scheme is known for general task models executing on multiprocessor platforms. The earlier example in Table 11.17 shows that EDF scheduling would lead to c missing its deadline, but the system is schedulable as evident from the partitioned approach. Not only are there no optimal schemes available, but what policies are available tend not to be sustainable – a schedulable system may become unschedulable if, for example, an extra processor is added. For these reasons, most current systems use a partitioned approach, which also has the advantage, as noted earlier, of run-time efficiency. Nevertheless, for the future, global schemes have the potential to deliver better schedulability.

Notwithstanding the lack of maturity in this area, it is possible to report some useful results. It was noted at the beginning of this chapter that single processor analysis started from a simple task model (in particular, tasks are independent and periodic, and have deadline equal to period). For uniprocessors, the model has been generalized significantly, but for multiprocessor scheduling, useful results are only really known for the simple task model. Here a number of such results are stated.

(1) There is a global scheduling algorithm called *pfair* that is able to schedule any periodic system with utilization $\leq M$ on M identical processors (Baruah et al., 1996). However, schedules generated by the pfair algorithm tend to have a large number of preemptions, and tasks are frequently required to move from one processor to another.

(2) For FP scheduling with partitioned placement, the following utilization-based sufficient schedulability test has been derived for a first-fit (using decreasing task utilization) placement strategy (Oh and Baker, 1998):

$$U \leq M(\sqrt{2} - 1) \tag{11.28}$$

(3) A sufficient utilization-based schedulability test is also known for EDF with partitioned first-fit placement. Let $U_{\max}$ denote the largest utilization of any task in the system: $U_{\max} = \max_{i=1}^{N} \left(\frac{C_i}{T_i}\right)$. Letting β denote $\lfloor 1/U_{\max} \rfloor$, this test is as follows (Lopez et al., 2004):

$$U \leq \frac{\beta M + 1}{\beta + 1} \tag{11.29}$$

(4) A sufficient schedulability test is known for global EDF (Goossens et al., 2003), that depends only upon the utilization of the task system and $U_{\max}$.

$$U \leq M - (M-1)U_{\max} \tag{11.30}$$

(5) A variant of EDF with global placement, called fpEDF, has been proposed (Baruah, 2004) that

(a) assigns greatest (fixed) priority to tasks with utilization $> \frac{1}{2}$ (if there are any such tasks), and

(b) schedules the remaining tasks according to EDF.

A couple of schedulability tests have been derived for fpEDF. One uses the utilization of the task system:

$$U \leq \left(\frac{M+1}{2}\right) \tag{11.31}$$

Another, superior, test for fpEDF uses both the utilization of the system and the largest utilization of any task in the system:

$$U \leq \max\left[M - (M-1)U_{\max}, \frac{M}{2} + U_{\max}\right] \tag{11.32}$$

A direct comparison between all these bounds would be misleading as the bounds are only sufficient. Tighter bounds are possible; indeed a number have been proposed in the literature for some of the above schemes. However, to illustrate the results that these test do provide, a couple of scenarios are defined and the tests applied. Case I has 10 processors ($M = 10$) with the highest utilization task having a capacity of 0.1. Case II has $M = 4$ and $U_{\max} = 2/3$. Table 11.19 gives the maximum utilization that is guaranteed to lead to a schedulable system for each of the scheduling schemes. To compute the average 'per-processor' utilization these values need to be divided by M (which is 10 in the first scenario and 4 in the other).

11.14.2 Scheduling the network

The next issue to consider is the communications infrastructure that links the different processors. For bus-based tightly coupled multiprocessors, the behaviour of multilevel

Scheme	Case I	Case II
Partitioned FP (Eqn 11.28)	4.14	1.66
Partitioned EDF (Eqn 11.29)	9.18	2.50
Global EDF (Eqn 11.30)	9.10	2.00
fpEDF (Eqn 11.32)	9.10	2.67

Table 11.19 Utilization bounds.

caches makes worst-case execution time analysis even harder for these platforms. Heterogeneous processors and hierarchical multispeed memories also add significantly to these difficulties. With network-based connections, the messages must themselves be scheduled if end-to-end data flows through the system are to be guaranteed. There are many different network protocols with some being more amenable to timing analysis than others. It is beyond the scope of this book to discuss these different protocols in detail, but two specific schemes are worth noting.

- **Time Division Multiple Access (TDMA)** – here each processor is allocated a fixed time slot within a specified cycle in which tasks hosted on that processor can generate messages.
- **Control Area Network (CAN)** – here each message is given a fixed priority and the network supports priority-based arbitration.

So with TDMA, which is only really applicable to static task allocation, no two processors ever wish to produce messages at the same time. With CAN, competition can occur but priorities are used to order the messages. Being priority-based, the standard RTA presented earlier in this chapter is directly applicable to scheduling CAN. As a message cannot be preempted once it has started to be transmitted, the non-preemptive form of the analysis is the one employed with CAN (see Section 11.10.3), where the parameter C is now the time needed to transmit the message.

The use of a network within a system opens up a number of issues as well as message scheduling. Unless the hardware architecture is fully connected, routing needs to be addressed. Static and dynamic route-finding are possible. Fault tolerance over message corruption is normally dealt with by the transmission protocol, but extra messages and perhaps alternative routes may be employed and these must be accommodated into the scheduling analysis.

11.14.3 Mutual exclusion on multiprocessor platforms

The final issue to address in this short review of the major problems involved in scheduling multiprocessor platforms is the provision of mutual exclusion over shared objects. In networked systems, these objects are typically controlled by a single thread so there are no new problems to solve, but for shared memory systems there is now the need to provide protection from true parallel access. The priority inheritance and priority ceiling protocols no longer work as they depend on an executing high-priority task preventing a lower-priority task from executing. This clearly will not occur if the lower-priority task is on another processor. There are no simple equivalent protocols to those for single processors.

To implement mutual exclusion in a multiprocessor shared memory system usually requires locks that can be contended for globally. When a task holds such a lock it is usual to prevent it from being preempted locally (as this would further delay tasks on other processors waiting for the lock). When a lock is requested but not granted, the task will typically busy-wait on the lock, waiting for it to become free – this is known as **spinning**. Obviously this spinning time will add to the task's execution time and hence has to be bounded if the task's interference on lower-priority tasks is to be calculated. A system with many globally shared objects and nested usage patterns (i.e. accessing one

object whilst holding the lock on others) will be harder to analyse and the analysis itself is likely to be pessimistic. Also error conditions such as deadlocks and livelocks are now possible whereas they were prevented by some of the single processor protocols.

Because of these difficulties with global locks, the use of lock-free algorithms is attractive. Here multiple copies of the shared object are supported and if necessary actions are repeated if conflicts over the copies have occurred. To give a simple example, consider an object that is read by many tasks but updated by only one. While the update is happening all reads are made to an old copy of the object. Once the update is finished a single flag is set to make the new copy available to future read operations. If the timing constraints on the system allow concurrent reads and writes then it must be acceptable for the read operation to get the old value – if it had arrived any earlier (or the writing task any later) then the old copy would have been the 'current' one.

Overall, multiprocessor systems whilst providing more computational power introduce a number of challenges for real-time systems. The move from scheduling a single resource to the coordination of multi-resources is a major one that requires a holistic approach to system scheduling. However, the core of this approach will always be the management of each individual resource.

11.15 Scheduling for power-aware systems

All of the scheduling results presented in this chapter have the common form of: given a set of execution time requirements (the Cs), will all the tasks complete by their deadlines (the Ds)? This assumes a fixed speed processor (or processors) so that the worst-case execution time values can be obtained prior to attempting the system-wide scheduling analysis. There are, however, variable speed processors that can give rise to a difference scheduling question – at what speed must the processor execute in order for the tasks to be schedulable?

Variable speed resources are typically found in power-aware applications, that is in embedded systems that run on batteries. Examples of such systems are mobile devices and nodes in a sensor net. All battery-based systems have the need to preserve energy and thereby extend their operational life or periods between recharges.

To save power, the voltage to the processor is reduced with the result that it runs slower. But the saving is non-linear. Halving the speed of a processor may quadruple its life. Some processors have variable speed, others support just a finite set of speed settings. From the point of view of the scheduling analysis, the verification problem now has two stages.

- With the processor running at its maximum speed (Max), is the system schedulable? This is a standard test.
- If the system is schedulable, by what maximum factor k can all the C values be increased so that the system remains schedulable?

There is no simple way to compute k, rather it needs to be found by a branch and bound search. Consider the fixed priority example given in Table 11.20 where the C values are those that are appropriate for the maximum speed of the processor.

Task	T	D	C
a	70	70	5
b	120	100	7
c	200	200	11

Table 11.20 Example task set with maximum speed.

Task	T	D	C	R
a	70	70	25	25
b	120	100	35	60
c	200	200	55	200

Table 11.21 Example task set with reduced speed.

This is clearly schedulable ($R_a = 5$, $R_b = 12$ and $R_c = 23$). If k is given the value 10 (i.e. execution times are now 50, 70 and 110) then the utilization is greater than 1 so the system is clearly unschedulable. So k must lie between 1 and 10. Using, for illustration, only integer values for k, the value 6 could be tried next (result unschedulable) then 4 (schedulable) and then 5. The results for $k = 5$ are shown in Table 11.21. Note the response time for task c is just on its deadline (200); any increase in any C parameter would cause this task to become unschedulable. Hence $k = 5$ is the optimum value and it is possible to conclude that the task set is schedulable on a processor with speed Max/5.

This discussion has focused on statically fixing the processor speed so that all deadlines are (just) met. In more dynamic systems where the work load fluctuates at run-time it is possible to change the processor's speed whilst continuing to execute the application. For all processors there is a cost (overhead) in making these changes. It is also necessary to take into account the impact on memory performance and I/O devices – savings in processing cost may not lead to overall system economy.

11.16 Incorporating system overheads

In all the analysis presented so far in this chapter, the overheads of actually implementing the multitasking system software have been ignored. Clearly for a real system this is not acceptable and hence the scheduling equations need to be expanded to include terms for the overhead factors. The following characteristics are typical of many operating system kernels or language run-time support systems.

- The cost of a context switch between tasks is not negligible and may not be a single value. The cost of a context switch to a higher-priority periodic task (following, for example, a clock interrupt) may be higher than a context switch from a task to a lower-priority task (at the end of the high-priority task's execution). For systems with a large number of periodic tasks, an additional cost will be incurred for

manipulating the delay queue (for periodic tasks when they execute, say, an Ada 'delay until' statement).

- All context switch operations are non preemptive.
- The cost of handling an interrupt (other than the clock) and releasing an application sporadic task is not insignificant. Furthermore, for DMA and channel-program controlled devices, the impact of shared-memory access can have a non-trivial impact on worst-case performance – such devices are best avoided in hard real-time systems.
- A clock interrupt (say every 10 ms) could result in periodic tasks being moved from a delay queue to the dispatch/ready queue. The cost for this operation varies depending on the number of tasks to be moved.

In addition to the above, the scheduling analysis must take into account the features of the underlying hardware, such as the impact of the cache and pipeline.

11.16.1 Modelling non-trivial context switch times

Most scheduling models ignore context switch times. This approach is, however, too simplistic if the total cost of the context switches is not trivial when compared with the application's own code. Figure 11.12 illustrates a number of significant events in the execution of a typical periodic task.

A – the clock interrupt that designates the notional time at which the task should start (assuming no release jitter or non-preemptive delay – if the interrupts were disabled due to the operation of the context switch then the clock handler would have its execution delayed; this is taken into account in the scheduling equations by the blocking factor B).

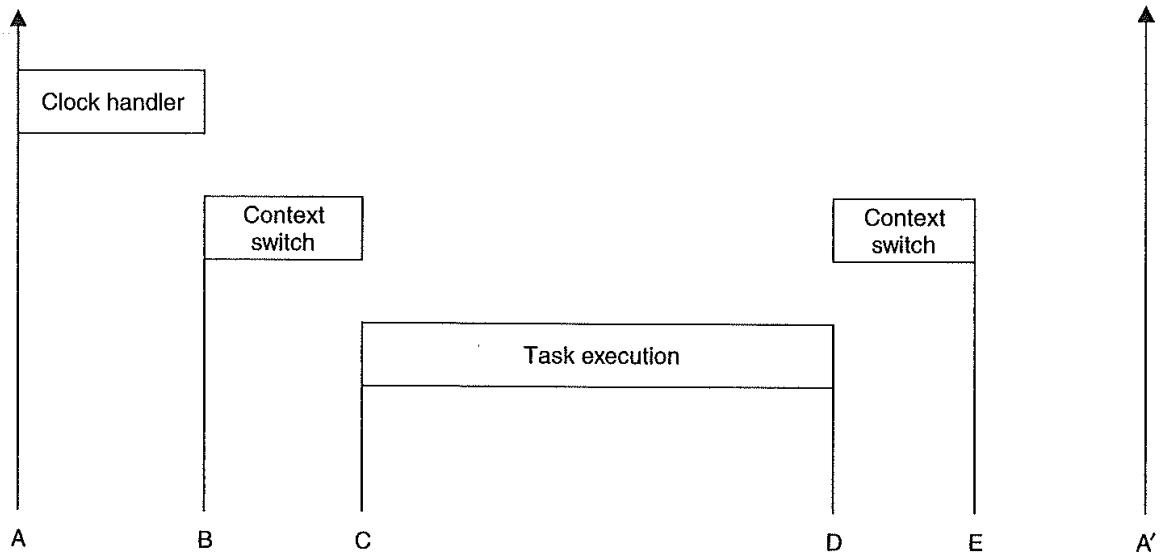

Figure 11.12 Overheads when executing tasks.

B – the earliest time that the clock handler can complete; this signifies the start of the context switch to the task (assume it is the highest priority runnable task).

C – the actual start of the execution of the task.

D – the completion of the task (the task may be preempted a number of times between C and D).

E – the completion of the context switch away from the task.

A′– the next release of the task.

The typical requirement for this task is that it completes before its next release (that is, D $<$ A′), or before some deadline prior to its next release. Either way, D is the significant time, not E. Another form of requirement puts a bound on the time between the start of execution and termination (that is, D–C). This occurs when the first action is an input and the last an output (and there is a deadline requirement between the two). While these factors affect the meaning of the task's own deadline (and hence its response time) they do not affect the interference this task has on lower-priority tasks; here the full cost of both context switches counts. Recall that the basic scheduling equation (11.7) has the form:

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

This now becomes (for periodic tasks only):

$$R_i = CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j) \qquad \textbf{(11.33)}$$

where CS^1 is the cost of the initial context switch (to the task) and CS^2 is the cost of the context switch away from each task at the end of its execution. The cost of putting the task into the delay queue (if it is periodic) is incorporated into C_i. Note that in practice this value may depend on the size of the queue; a maximum value would need to be incorporated into C_i.

This measure of the response time is from point B in Figure 11.12. To measure from point C, the first CS^1 term is removed. To measure from point A (the notional true release time of the task) requires the clock behaviour to be measured (see Section 11.16.3).

For multiprocessor systems the context switch itself may be more complicated if global placement is used as tasks may need to migrate from one processor to another. This will add to the context switch overhead and will make the prediction of worst-case execution time (WCET) more difficult as the cache may not be shared between all the processors.

11.16.2 Modelling sporadic tasks

For sporadic tasks released by other sporadic tasks, or by periodic tasks, Equation (11.33) is a valid model of behaviour. However, the computation time for the task, C_i, must include the overheads of blocking on the agent that controls its release.

When sporadic tasks are released by an interrupt, priority inversion can occur. Even if the sporadic has a low priority (due to its having a long deadline) the interrupt itself will be executed at a high hardware priority level. Let Γs be the set of sporadic tasks released by interrupts. Each interrupt source will be assumed to have the same arrival characteristics as the sporadic that it releases. The additional interference these interrupt handlers have on each application task is given by:

$$\sum_{k \in \Gamma s} \left\lceil \frac{R_i}{T_k} \right\rceil IH$$

where IH is the cost of handling the interrupt (and returning to the running task, having released the sporadic task).

This representation assumes that all interrupt handlers give rise to the same cost; if this is not the case then IH must be defined for each k. Equation (11.33) now becomes:

$$R_i = CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j) + \sum_{k \in \Gamma s} \left\lceil \frac{R_i}{T_k} \right\rceil IH \quad \textbf{(11.34)}$$

Within Ada, timing events are also used to release sporadic tasks or to undertake short event-handling activities. As timing events are similar to interrupts they can be modelled in the way shown above.

11.16.3 Modelling the real-time clock handler

To support periodic tasks, the execution environment must have access to a real-time clock that will generate interrupts at appropriate times. An ideal system will use an interval timer, and will interrupt only when a periodic task needs to be released. The more common approach, however, is one in which the clock interrupts at a regular rate (say once every 10 ms) and the handler must decide if none, one or a number of periodic tasks must be released. The ideal approach can be modelled in an identical way to that introduced for sporadic tasks (see Section 11.16.2). With the regular clock method, it is necessary to develop a more detailed model as the execution times of the clock handler can vary considerably. Table 11.22 gives possible times for this handler (for a clock period of 10 ms). Note that if the worst case was assumed to occur on all occasions, over

Queue state	Clock handling time, μs
No tasks on queue	16
Tasks on queue but none removed	24
One task removed	88
Two tasks removed	128
Twenty-five tasks removed	1048

Table 11.22 Clock handling overheads.

10% of the processor would have to be assigned to the clock handler. Moreover, all this computation occurs at a high (highest) hardware priority level, and hence considerable priority inversion is occurring. For example, with the figures given in the table, at the LCM (least common multiple) of the 25 periodic tasks 1048 μs of interference would be suffered by the highest priority application task that was released. If the task was released on its own then only 88 μs would be suffered. The time interval is represented by B–A in Figure 11.12.

In general, the cost of moving N periodic tasks from the delay queue to the dispatch queue can be represented by the following formula:

$$C_{clk} = CT^c + CT^s + (N-1)CT^m$$

where CT^c is the constant cost (assuming there is always at least one task on the delay queue), CT^s is the cost of making a single move, and CT^m is the cost of each subsequent move. This model is appropriate due to the observation that the cost of moving just one task is often high when compared with the additional cost of moving extra tasks. With the kernel considered here, these costs were:

CT^c	24 μs
CT^s	64 μs
CT^m	40 μs

To reduce the pessimism of assuming that a computational cost of C_{clk} is consumed on each execution of the clock handler, this load can be spread over a number of clock ticks. This is valid if the shortest period of any application task, $T_{\min}$ is greater than the clock period, T_{clk}. Let M be defined by:

$$M = \left\lceil \frac{T_{\min}}{T_{clk}} \right\rceil$$

If M is greater than 1 then the load from the clock handler can be spread over M executions. In this situation, the clock handler is modelled as a task with period $T_{\min}$ and computation time C'_{clk}:

$$C'_{clk} = M(CT^c + CT^s) + (N-M)CT^m$$

This assumes $M <= N$.

Equation (11.34) now becomes

$$R_i = CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j)$$

$$+ \sum_{k \in \Gamma s} \left\lceil \frac{R_i}{T_k} \right\rceil IH + \left\lceil \frac{R_i}{T_{\min}} \right\rceil C'_{clk} \qquad \textbf{(11.35)}$$

To give further improvements (to the model) requires a more exact representation of the clock handler's actual execution. For example, using just CT^c and CT^s the following

equation can easily be derived:

$$R_i = CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j)$$
$$+ \sum_{k \in \Gamma s} \left\lceil \frac{R_i}{T_k} \right\rceil IH + \left\lceil \frac{R_i}{T_{clk}} \right\rceil CT_c$$
$$+ \sum_{g \in \Gamma p} \left\lceil \frac{R_i}{T_g} \right\rceil CT_s \qquad \textbf{(11.36)}$$

where Γp is the set of periodic tasks.

It is left as an exercise for the reader to incorporate the three-parameter model of clock handling (see Exercise 11.16).

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.

Many 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.
- Tasks with long periods (that is, longer than the major cycle) are supported inefficiently.
- Tasks 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 more dynamic scheduling schemes. The main topic of the chapter has been the fixed priority scheduling of a collection of tasks on a single processor. Following the description of a simple utilization-based test (which is only applicable to a restricted task model), the response time calculations were derived for a more flexible model. This model can accommodate sporadic tasks, task interactions, non-preemptive sections, release jitter, aperiodic servers, fault-tolerant systems and an arbitrary relationship between a task deadline (D) and its minimum arrival interval (T).

Intertask 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 task load. Three algorithms have been described in this chapter:

- rate monotonic – for $D = T$
- deadline monotonic – for $D \leq T$
- arbitrary – for $D > T$.

The other significant scheduling approach is EDF – here the task with the shortest (earliest) deadline is the one to execute first. EDF has the advantage that it optimally allocates the processor. If an application cannot be scheduled by EDF then it cannot be scheduled by any other approach. For simple systems with $D = T$ then a very simple and exact utilization-based test can be used to check schedulability. When $D \leq T$ then Processor Demand Analysis has to be undertaken. An efficient approach to this form of analysis called QPA was introduced and illustrated by examples.

Also in this chapter a number of other scheduling topics have been referred to (even if they have not been dealt with comprehensively). Amongst these are: worst-case execution time estimation, multiprocessor platforms, dynamic open systems and power-aware applications. Finally attention was focused on the overheads of implementing multitasking application code. It was shown how the standard response time equations can be extended to incorporate parameters to characterize the implementation's run-time behaviour.

Further reading

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

Davis, R., Zaboos, A. and Burns, A. (2008) Effective exact schedulability tests for fixed priority real-time systems. *IEEE Transactions on Computing*, **57**(9), 1261–1276.

Ermedahl, A. and Engblom, J. (2007) Execution time analysis for embedded real-time systems, in *Handbook of Real-Time and Embedded Systems*, I. Lee, J. Y-T. Leung and S. H. Son (eds). Boca Raton, FL: Chapman and Hall/CRC.

Klein, M. H., Ralya, T., Pollak, B., Obenza, R. and Gonzalez Harbour, M. (1993) *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. New York: Kluwer Academic.

Liu, J. W. S (2000) *Real-Time Systems*. New York: 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.

Sha, L. et al. (2004) Real time scheduling theory: A historical perspective. *Real-Time Systems*, **28**(2 – 3), 101–155.

Stankovic, J. A., Spuni, M., Ramamritham, K. and Buttazzo., G. C. (1998) *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. New York: Kluwer Academic.

Exercises

11.1 Three logical tasks 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 tasks can be scheduled using the rate monotonic scheduling algorithm.
Show how a cyclic executive could be constructed to implement the three logical tasks.

11.2 Consider three tasks P, Q and S. P has a period of 100 ms in which it requires 30 ms of tasking. The corresponding values for Q and S are (5,1) and (25,5) respectively. Assume that P is the most important task 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 tasks be scheduled so that all deadlines are met?
(4) Illustrate one of the schemes that allows these tasks to be scheduled.

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

11.4 Figure 11.13 illustrates the behaviour of four periodic tasks w, x, y and z. These tasks have priorities determined by the rate monotonic scheme, with the result that priority(w) > priority(x) > priority(y) > priority(z).
Each task's period starts at time S and terminates at T. The four tasks share two 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 11.23 summarizes the task's requirements.
The figure shows the execution histories of the four tasks 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 w, but executes again at time 16. Finally it terminates at time 17.
Redraw Figure 11.13 to illustrate the behaviour of these tasks if priority inheritance is employed.

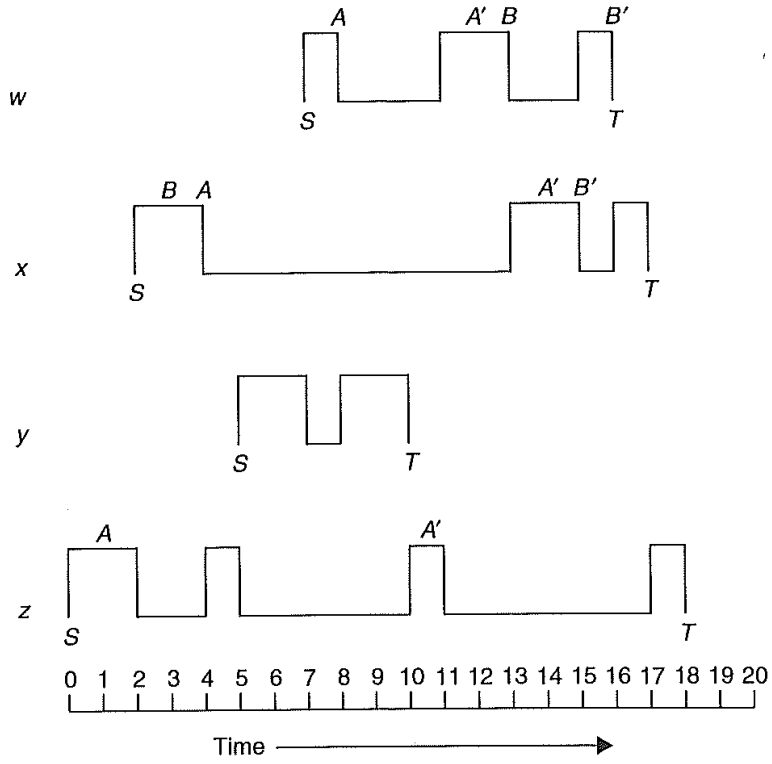

Figure 11.13 The behaviour of four periodic tasks in Exercise 11.4.

Task	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 11.23 Summary of the task's requirements for Exercise 11.4.

11.5 Redraw the figure given in Exercise 11.4 to illustrate the behaviour of these tasks if immediate priority ceiling inheritance is employed.

11.6 With the priority ceiling protocol, it is possible to calculate the maximum time any task can be blocked by the operation of a lower-priority task. What is the rule for calculating this blocking? Illustrate the answer by calculating the maximum blocking time for each task in the following example. A program consists of five tasks, a, b, c, d, e (these are listed in priority order with a having the highest priority), and six resources $R1, \ldots, R6$ (protected by semaphores implementing the priority ceiling protocol). The resource accesses have worst case execution times given in Table 11.24.

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

Table 11.24 Summary of resource access times for Exercise 11.6.

Resources are used by the tasks according to Table 11.25.

Task	Uses
a	*R*3
b	*R*1,*R*2
c	*R*3,*R*4,*R*5
d	*R*1,*R*5,*R*6
e	*R*2,*R*6

Table 11.25 Summary of the tasks' resource requirements for Exercise 11.6.

11.7 Is the task set shown in Table 11.26 schedulable using the simple utilization-based test given in Equation (11.1)? Is the task set schedulable using the response time analysis?

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

Table 11.26 Summary of the tasks' attributes for Exercise 11.7.

11.8 The task set shown in Table 11.27 is not schedulable using Equation (11.1) because *a* must be given the top priority due to its criticality. How can the task set be transformed so that it is schedulable? Note that the computations represented by *a* must still be given top priority.

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

Table 11.27 Summary of the tasks' attributes for Exercise 11.8.

11.9 The task set given in Table 11.28 is not schedulable using Equation (11.1), but does meet all deadlines when scheduled using fixed priorities. Explain why.

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

Table 11.28 Summary of the tasks' attributes for Exercise 11.9.

11.10 In Section 11.6, a sporadic task was defined as having a minimum inter-arrival time. Often sporadic tasks come in bursts. Update Equation (11.4) to cope with a burst of sporadic activities such that N invocations can appear arbitrarily close together in a period of T.

11.11 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.

11.12 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?

11.13 In a safety-critical real-time system, a collection of tasks 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 tasks can be used to monitor such events.

11.14 Consider the list of events (shown in Table 11.29) together with the computation costs of responding to each event. If a separate task is used for each event (and these tasks 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 11.29 Summary of events for Exercise 11.14.

11.15 How can the task set shown in Table 11.30 be optimally scheduled (using fixed-priority scheduling)? Is this task set schedulable?

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

Table 11.30 Summary of tasks for Exercise 11.15.

11.16 Develop a model of clock handling which incorporates the three parameters CT^c, CT^s and CT^m (see Section 11.16.3).

11.17 Rather than using a clock interrupt to schedule periodic tasks, what would be the ramifications of only having access to a real-time clock?

11.18 A periodic task of period 40 ms is controlled by a clock interrupt that has a granularity of 30 ms. How can the worst-case response time of this task be calculated?