Chapter 13
Scheduling

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

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

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

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

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

13.1 Simple process model

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

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

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

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

13.2 The cyclic executive approach

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

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

Table 13.1 Standard notation.

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

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

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

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

Table 13.2 Cyclic executive process set.

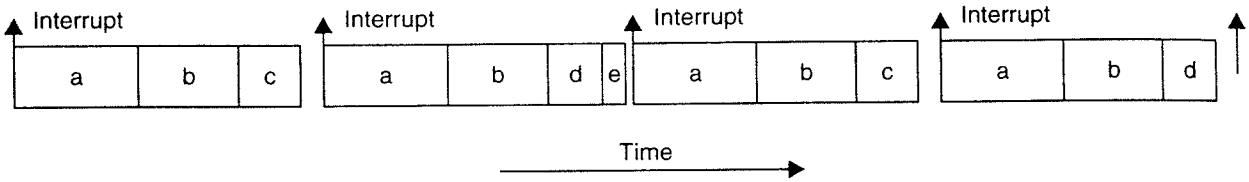

Figure 13.1 Time-line for process set.

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

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

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

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

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

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

13.3 Process-based scheduling

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

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

13.3.1 Scheduling approaches

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

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

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

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

13.3.2 Preemption and non-preemption

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

13.3.3 FPS and rate monotonic priority assignment

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

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

Table 13.3 Example of priority assignment.

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

Table 13.4 Utilization bounds.

13.4 Utilization-based schedulability tests

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

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

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

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

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

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

Table 13.5 Process set A.

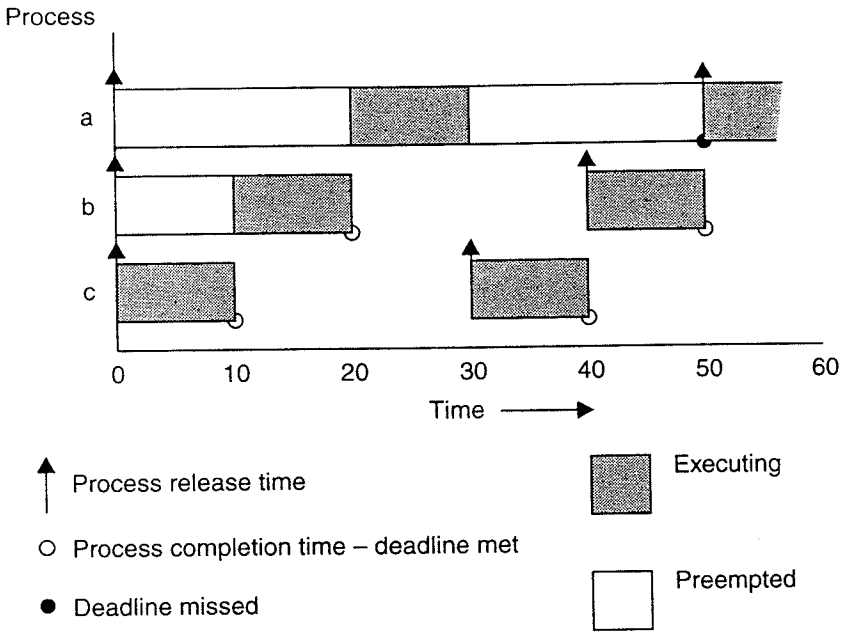

Figure 13.2 Time-line for process set A.

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

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

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

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

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

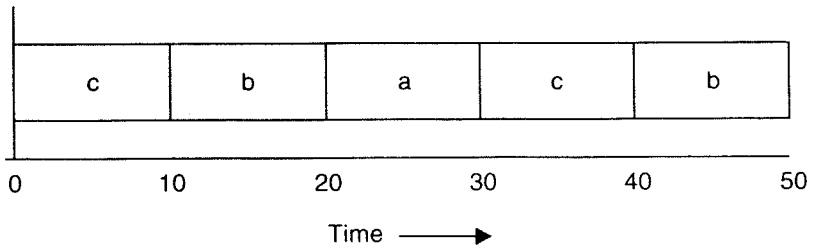

Figure 13.3 Gantt chart for process set A.

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

Table 13.6 Process set B.

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

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

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

Table 13.7 Process set C.

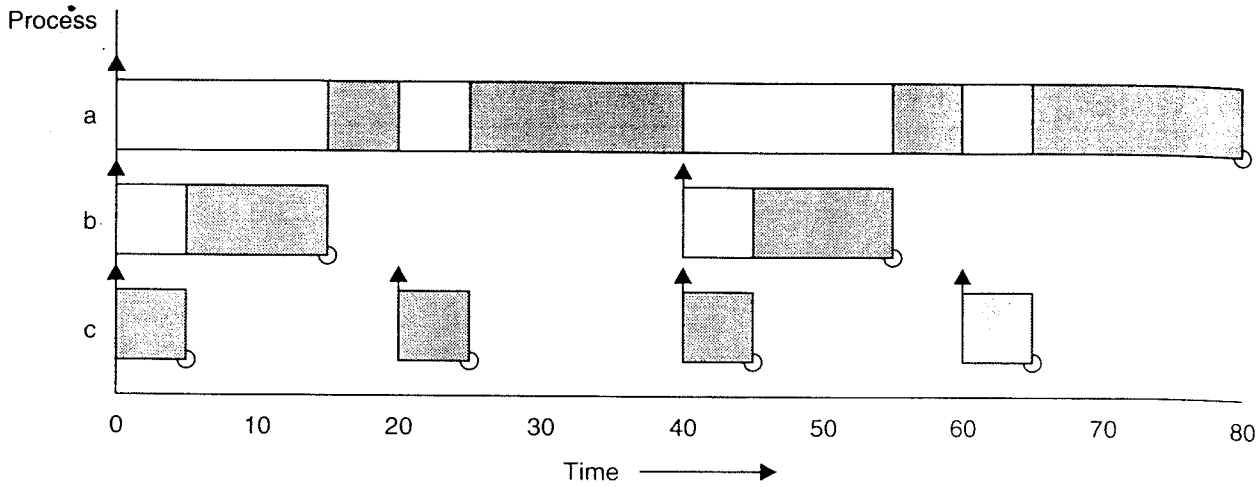

Figure 13.4 Time-line for process set C.

13.4.1 Utilization-based schedulability tests for EDF

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

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

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

- FPS is easier to implement, as the scheduling attribute (*priority*) is static; EDF is dynamic and hence requires a more complex run-time system which will have higher overhead.
- It is easier to incorporate processes without deadlines into FPS (by merely assigning them a priority); giving a process an arbitrary deadline is more artificial.
- The deadline attribute is not the only parameter of importance; again it is easier to incorporate other factors into the notion of priority than it is into the notion of deadline.
- During overload situations (which may be a fault condition) the behaviour of FPS is more predictable (the lower priority processes are those that will miss their deadlines first); EDF is unpredictable under overload and can experience

a domino effect in which a large number of processes miss deadlines. This is consider again in Section 13.13.

- The utilization-based test, for the simple model, is misleading as it is necessary and sufficient for EDF but only sufficient for FPS. Hence higher utilizations can, in general, be achieved for FPS.

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

13.5 Response time analysis for FPS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
for i in 1..N loop -- for each process in turn
  n := 0
  w_i^n := C_i
  loop
    calculate new w_i^(n+1) from Equation (13.5)
    if w_i^(n+1) = w_i^n then
      R_i := w_i^n
      exit {value found}
```

```
    end if
    if w_i^{n+1} > T_i then
      exit {value not found}
    end if
    n := n + 1
  end loop
end loop
```

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

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

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

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

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

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

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

Table 13.8 Process set D

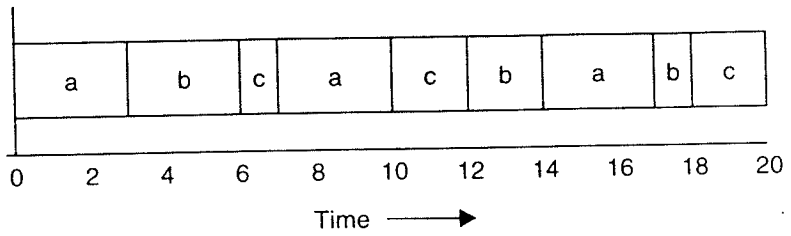

Figure 13.5 Gantt chart for process set D.

is used to derive the next value of w:

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

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

The final process will give rise to the following calculations:

$$w_c^0 = 5$$

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

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

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

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

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

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

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

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

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

Table 13.9 Response time for process set C.

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

Table 13.10 A process set for EDF.

fail the test, then, at run-time, a process will miss its deadline (unless the computation time estimations, C, themselves turn out to be pessimistic). As these tests are superior to the utilization-based ones, this chapter will concentrate on extending the applicability of the response time method.

13.6 Response time analysis for EDF

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

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

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

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

13.7 Worst-case execution time

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

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

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

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

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

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

With no further information, the total 'cost' of this construct would be 10×100 + 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.