Chapter 2
Reliability and fault tolerance

Reliability and safety requirements are usually much more stringent for real-time and embedded systems than for other computer systems. For example, if an application which computes the solution to some scientific problem fails then it may be reasonable to abort the program, as only computer time has been lost. However, in the case of an embedded system, this may not be an acceptable action. A process control computer, for instance, responsible for the operation of a large gas furnace, cannot afford to close down the furnace as soon as a fault occurs. Instead, it must try to provide a degraded service and prevent a costly shutdown operation. More importantly, real-time computer systems may endanger human lives if they abandon control of their application. An embedded computer controlling a nuclear reactor must not let the reactor run out of control, as this may result in a core meltdown and an emission of radiation. A military avionics system should at least allow the pilot to eject before permitting the plane to crash!

It is now widely accepted that the society in which we live is totally dependent on the use of computer-based systems to support its vital functions. It is, therefore, imperative that these systems do not fail. Without wishing to define precisely what is meant by a system failure or a fault (at the moment), there are, in general, four sources of faults which can result in an embedded system failure.

(1) Inadequate specification. It has been suggested that the great majority of software faults stem from inadequate specification (Leveson, 1986).

Included in this category are those faults that stem from misunderstanding the interactions between the program and the environment.

(2) Faults introduced from design errors in software components.

(3) Faults introduced by failure of one or more hardware components of the embedded system (including processors).

(4) Faults introduced by transient or permanent interference in the supporting communication subsystem.

It is these last three types of fault which impinge on the programming language used in the implementation of an embedded system. The errors introduced by design faults are, in general, unanticipated (in terms of their consequences), whereas those from processor and network failure are, in some senses, predictable. One of the main requirements, therefore, for any real-time programming language, is that it must facilitate the construction of highly dependable systems. In this chapter, some of the general design techniques that can be used to improve the overall reliability of embedded computer systems are considered. Chapter 3 will show how **exception-handling** facilities can be used to help implement some of these design philosophies, particularly those based on **fault tolerance**.

2.1 Reliability, failure and faults

Before proceeding, more precise definitions of reliability, failures and faults are necessary. Randell et al. (1978) define the **reliability** of a system to be:

> a measure of the success with which the system conforms to some authoritative specification of its behaviour.

Ideally, this specification should be complete, consistent, comprehensible and unambiguous. It should also be noted that the *response times* of the system are an important part of the specification, although discussion of the meeting of deadlines will be postponed until Chapter 11. The above definition of reliability can now be used to define a system **failure**. Again, quoting from Randell et al.:

> When the behaviour of a system deviates from that which is specified for it, this is called a failure.

Section 2.9 will deal with the metrics of reliability; for the time being, *highly reliable* will be considered synonymous with a *low failure rate*.

The alert reader will have noticed that our definitions, so far, have been concerned with the *behaviour* of a system; that is, its *external* appearance. Failures result from unexpected problems internal to the system which eventually manifest themselves in the system's external behaviour. These problems are called **errors** and their mechanical or algorithmic causes are termed **faults**. A faulty component of a system is, therefore, a component which, under a particular set of circumstances during the lifetime of the system, will result in an error. Viewed in terms of state transitions, a system can be considered as a number of *external* and *internal* states. An external state which is not

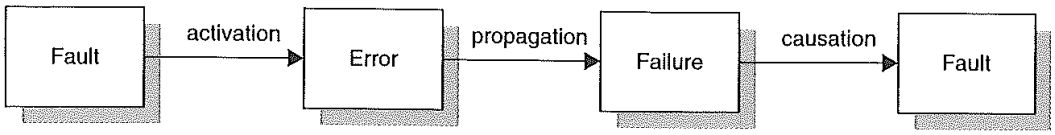

Figure 2.1 Fault, error, failure, fault chain.

specified in the behaviour of the system is regarded as a failure of the system. The system itself consists of a number of components, each with their own states, all of which contribute to the system's external behaviour. The combined states of these components are termed the internal state of the system. An internal state which is not specified is called an error and the component which produced the illegal state transition is said to be faulty.

A fault is **active** when it produces an error, and until this point it is **dormant**. Once produced, the error can be transformed into other errors via the computational process as it propagates through the system. Eventually, the error manifests itself at the boundaries of the system causing a service delivery to fail (Avizienis et al., 2004).

Of course, a system is usually composed of components; each of these may be considered as a system in its own right. Hence a failure in one system will lead to a fault in another which will result in an error and potential failure of that system. This in turn will introduce a fault into any surrounding system and so on (as illustrated in Figure 2.1).

There are many different classifications of fault types depending on the aspect of interest. For example, whether they are created during development or during operations, whether they are intentionally or accidentally created, whether they are hardware or software in origin, etc. From a real-time perspective, the duration of the fault is one of the most important aspects. Three types of fault can be distinguished.

(1) **Transient faults** – a transient fault occurs at a particular time, remains in the system for some period and then disappears. It will initially be dormant but can become active at any time. Examples of such faults occur in hardware components which have an adverse reaction to some external interference, such as electrical fields or radioactivity. After the disturbance disappears so does the fault (although not necessarily the induced error). Many faults in communication systems are transient.

(2) **Permanent faults** – permanent faults start at a particular time and remain in the system until they are repaired; for example, a broken wire or a software design error.

(3) **Intermittent faults** – transient faults that occur from time to time. An example is a hardware component that is heat sensitive: it works for a time, stops working, cools down and then starts to work again.

Software faults are usually called **bugs** and it can be notoriously difficult to isolate and identify them. Over the years, particular types of bugs have been given names in an informal classification. Originally two types of software bugs were identified (Gray, 1986).[1]

[1]The names come from analogies with physics. The assertion that most production software bugs are ephemeral – Heisenbugs that go away when you look at them – is well known to systems programmers. Bohrbugs, like the Bohr atom, are solid, easily detected by standard techniques.

- **Bohrbugs** – these bugs are reproducible and usually identifiable. Hence they can easily be removed during testing. If they cannot be removed, then design diversity techniques can be employed during operation (see Section 2.4).
- **Heisenbugs** – these are software bugs that only activate under certain rare circumstances. A good example is code shared between concurrent tasks that is not properly synchronized. Only when two tasks happen to execute the code concurrently will the fault activate and even then the error may propagate a long way from its source before it is detected. Because of this, they often disappear when investigated – hence their name.

A particular type of Heisenbug is one that results from 'software aging' (Parnas, 1994). In one sense, software can be thought of as not deteriorating with age (unlike hardware). Whilst this is true, faults can remain dormant for a long time, and only become active after significant continual use of the software. These faults are normally related to resources: for example in a dynamic application where memory is constantly allocated and freed, a fault that doesn't free unused memory will result in a **memory leak**. If this is small, the program may run for a significant period of time before memory becomes exhausted.

A good example of the effects of software ageing can be found with the use of the US Patriot missile defence system in the Gulf War in 1991 (see GAO/IMTEC-92-26 Patriot Missile Software Problem at http://www.fas.org/spp/starwars/gao/im92026.htm). The Patriot system was originally designed for mobile operations in Europe. The design assumed that it would only operate for a few hours at one location. During the Gulf War it was used continuously for many hours. Its main battery could last for 100 hours. After the Patriot's radar detects an airborne object that has the characteristics of a Scud missile, the range gate (an electronic detection device within the radar system) calculates an area in the air space where the system should next look for the detected missile. The range gate filters out information about airborne objects outside its calculated area and only processes the information needed for tracking, targeting and intercepting Scuds. Finding an object within the calculated range gate area confirms that it is a Scud missile. In February 1991, a Patriot missile defence system failed to track and intercept an incoming Scud. This Scud subsequently hit an Army barracks, killing 28 people.

The reason for the failure of the Patriot's systems is explained by considering the range gate's prediction software, which used the Scud's velocity and the time of the last radar detection. Time is kept continuously by the system's internal clock in tenths of seconds held as an integer variable. The longer the system has been running, the larger the number representing time. To predict where the Scud will next appear, both time and velocity must be expressed as real numbers. The registers in the Patriot computer are only 24 bits long, and the conversion of time results in a loss of precision causing a less accurate time calculation. The effect of this inaccuracy on the range gate's calculation is directly proportional to the target's velocity and the length of time the system has been running. Consequently, performing the conversion after the Patriot has been running continuously for extended periods causes the range gate to shift away from the centre of the target, making it less likely that the target missile will be successfully intercepted. Table 2.1 shows the effect of this inaccuracy. After 20 hours, the target becomes outside the range gate. As with all software ageing problems, restarting the system (in this case before 20 hours of continual operational time) would clear the problem.

Hours	Seconds	Calculated time (seconds)	Inaccuracy (seconds)	Approximate shift in range gate (meters)
0	0	0	0	0
1	3600	3599.9966	.0034	7
8	28800	28799.9725	.0025	55
20	72000	71999.9313	.0687	137
48	172800	172799.8352	.1648	330
72	259200	259199.7528	.2472	494
100	360000	359999.6667	.3433	687

Table 2.1 Effect of extended run-time on Patriot operation (taken from http://www.fas.org/spp.starwars/gao/im92026.htm).

To create reliable systems, all types of fault must be prevented from causing erroneous system behaviour (that is failure). The difficulty this presents is compounded by the indirect use of computers in the *construction* of safety-critical systems. For example, in 1979 an error was discovered in a program used to design nuclear reactors and their supporting cooling systems. The fault that this caused in the reactor design had not been found during installation tests as it concerned the strength and structural support of pipes and valves. The program had supposedly guaranteed the attainment of earthquake safety standards in operating reactors. The discovery of the bug led to the shutting down of five nuclear power plants (Leveson, 1986).

2.2 Failure modes

A system can fail in many different ways. A designer who is using system X to implement another system, Y, usually makes some assumptions about X's expected failure modes. If X fails differently from that which was expected then system Y may fail as a result.

A system provides services. It is, therefore, possible to classify a system's failure modes according to the impact they have on the services it delivers. Two general domains of failure modes can be identified:

- **value failure** – the value associated with the service is in error;
- **time failure** – the service is delivered at the wrong time.

Combinations of value and timing failures are often termed **arbitrary**.

In general, a value error might still be within the correct range of values or be outside the range expected from the service. The latter is equivalent to a typing error in programming languages and is called a **constraint error**. It is usually easy to recognize this type of failure but its consequence can still be devastating. (Witness the cause of the Arianne 5 disaster where an exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer – see 'ARIANE 5, Flight 501 Failure, Report by the Inquiry Board' at http://klabs.org/richcontent/Reports/Failure_Reports/ariane/ariane501.htm.)

Failures in the time domain can result in the service being delivered:

- too early – the service is delivered earlier than required;
- too late – the service is delivered later than required (often called a **performance error**);
- infinitely late – the service is never delivered (often called an **omission failure**).

One further failure mode should be identified, which is where a service is delivered that is not expected. This is often called a **commission** or **impromptu** failure. It is, of course, often difficult to distinguish a failure in both the value and the time domain from a commission failure followed by an omission failure. Figure 2.2 illustrates the failure mode classification.

Given the above classification of failure modes, it is now possible to make some assumptions about how a system might fail.

- **Fail uncontrolled** – a system which can produce arbitrary errors in both the value and the time domains (including impromptu errors).
- **Fail late** – a system which produces correct services in the value domain but may suffer from a 'late' timing error.
- **Fail silent** – a system which produces correct services in both the value and time domains until it fails; the only failure possible is an omission failure and when this occurs all following services will also suffer an omission failure.
- **Fail stop** – a system which has all the properties of fail silent, but also permits other systems to detect that it has entered the fail-silent state.
- **Fail controlled** – a system which fails in a specified controlled manner.
- **Fail never** – a system which always produces correct services in both the value and the time domain.

Other assumptions and classifications are clearly possible, but the above list will suffice for this book.

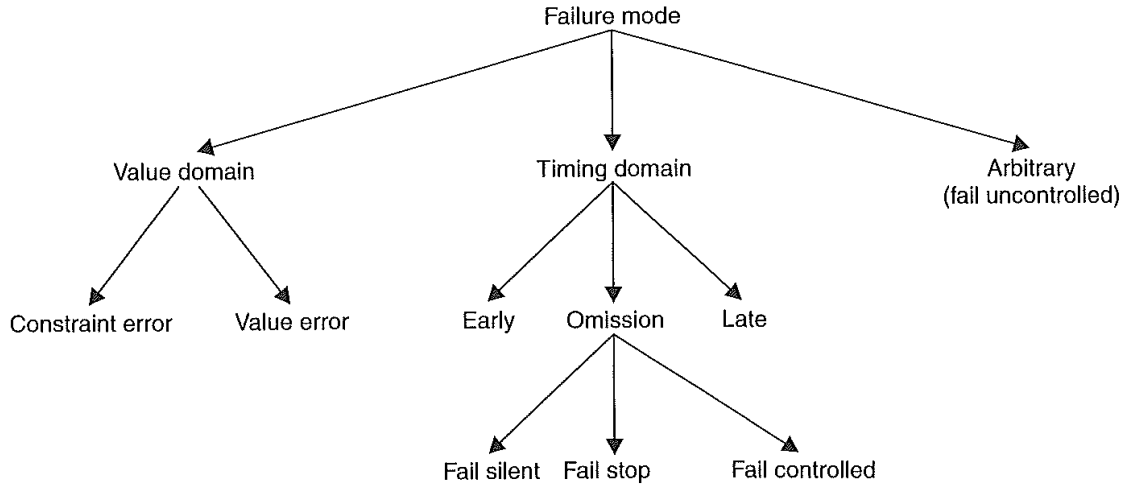

Figure 2.2 Failure mode classification.

2.3 Fault prevention and fault tolerance

Two approaches that can help designers improve the reliability of their systems can be distinguished (Anderson and Lee, 1990). The first is known as **fault prevention**; this attempts to eliminate any possibility of faults creeping into a system before it goes operational. The second is **fault tolerance**; this enables a system to continue functioning even in the presence of faults. Both approaches attempt to produce systems which have well-defined failure modes.

2.3.1 Fault prevention

There are two stages to fault prevention: **fault avoidance** and **fault removal**.

Fault avoidance attempts to limit the introduction of potentially faulty components during the construction of the system. For hardware this may entail (Randell et al., 1978):

- the use of the most reliable components within the given cost and performance constraints;
- the use of thoroughly-refined techniques for the interconnection of components and the assembly of subsystems;
- packaging the hardware to screen out expected forms of interference.

The software components of large embedded systems are nowadays much more complex than their hardware counterparts. It is virtually impossible in all cases to write fault-free programs. However the quality of software can be improved by:

- rigorous, if not formal, specification of requirements (for example, B or Z);
- the use of proven design methodologies (for example, those based on UML, such as Real-Time UML (Douglass, 1999));
- the use of analysis tools to verify key program properties (such as model checkers or proof checkers to ensure multitask programs are free from deadlock);
- the use of languages with facilities for data abstraction and modularity (for example, Ada or Java);
- the use of software engineering tools to help manipulate software components and thereby manage complexity (for example, configuration management tools such as CVS).

In spite of fault avoidance techniques, faults will inevitably be present in the system after its construction. In particular, there may be design errors in both hardware and software components. The second stage of fault prevention, therefore, is *fault removal*. This normally consists of procedures for finding and then removing the causes of errors. Although techniques such as design reviews, program verification and code inspections may be used, emphasis is usually placed on system testing. Unfortunately, system testing can never be exhaustive and remove all potential faults. In particular, the following problems exist.

- A test can only be used to show the presence of faults, not their absence.

- It is sometimes impossible to test under realistic conditions – one of the major causes for concern over the American Strategic Defense Initiative (SDI)[2] was the impossibility of testing any system realistically except under battle conditions. Most tests are done with the system in simulation mode and it is difficult to guarantee that the simulation is accurate. The last French nuclear testing at Mururao in the Pacific during 1995 was allegedly to allow data to be collected so that future tests would not be necessary but could be simulated accurately.
- Errors that have been introduced at the requirements stage of the system's development may not manifest themselves until the system goes operational. For example, in the design of the F18 aircraft an erroneous assumption was made concerning the length of time taken to release a wing-mounted missile. The problem was discovered only during operation when the missile failed to separate from the launcher after ignition, causing the aircraft to go violently out of control (Leveson, 1986).

In spite of all the testing and verification techniques, hardware components will fail; the fault prevention approach will, therefore, be unsuccessful when either the frequency or duration of repair times are unacceptable, or the system is inaccessible for maintenance and repair activities. An extreme example of the latter is the crewless spacecraft Voyager.

2.3.2 Fault tolerance

Because of the inevitable limitations of the fault prevention approach, designers of embedded systems must consider the use of fault tolerance. Of course, this does not mean that attempts at preventing faulty systems from becoming operational should be abandoned. However, this book will focus on fault tolerance rather than fault prevention.

Several different levels of fault tolerance can be provided by a system.

- **Full fault tolerance** – the system continues to operate in the presence of faults, albeit for a limited period, with no significant loss of functionality or performance.
- **Graceful degradation** (or fail soft) – the system continues to operate in the presence of errors, accepting a partial degradation of functionality or performance during recovery or repair.
- **Fail safe** – the system maintains its integrity while accepting a temporary halt in its operation.

The level of fault tolerance required will depend on the application. Although in theory most safety-critical systems require full fault tolerance, in practice many settle for graceful degradation. In particular, those systems which can suffer physical damage, such as combat aircraft, may provide several degrees of graceful degradation. Also, with highly complex applications which have to operate on a continuous basis (they have *high availability* requirements) graceful degradation is a necessity, as full fault tolerance is

[2]This was proposed by President Reagan in the 1980s. Its goal was to use ground-based and space-based systems to protect the US from attacks by ballistic missiles. It was never fully developed or deployed.

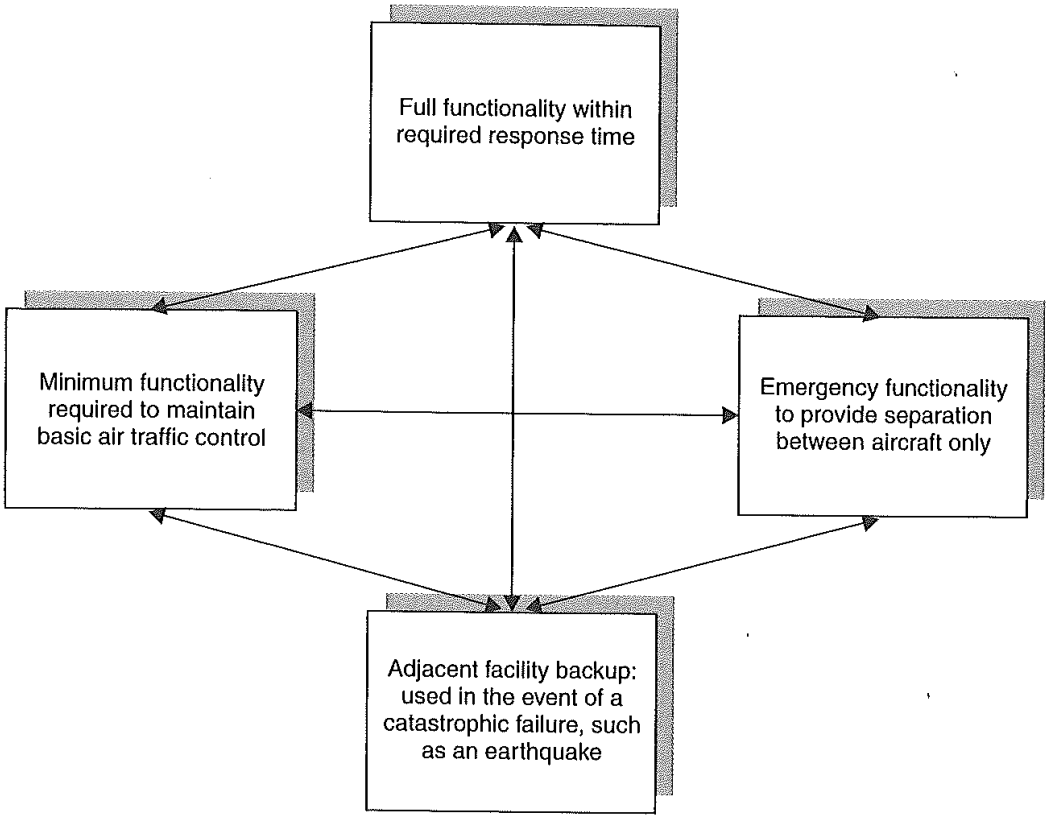

Figure 2.3 Graceful degradation and recovery in an air traffic control system.

not achievable for indefinite periods. For example, the Federal Aviation Administration's Advanced Automation System, which provides automated services to both *en route* and terminal air traffic controllers throughout the USA, has three levels of graceful degradation for its area control computer couplers (Avizienis and Ball, 1987). This is illustrated in Figure 2.3.

In some situations, it may simply be necessary to shut down the system in a safe state. These fail-safe systems attempt to limit the amount of damage caused by a failure. For example, the A310 Airbus's slat and flap control computers, on detecting an error on landing, restore the system to a safe state and then shut down. In this situation, a safe state is having both wings with the same settings; only asymmetric settings are hazardous in landing (Martin, 1982).

Early approaches to the design of fault-tolerant systems made three assumptions.

(1) The algorithms of the system have been correctly designed.
(2) All possible failure modes of the components are known.
(3) All possible interactions between the system and the environment have been foreseen.

However, the increasing complexity of computer software and the introduction of multi-core hardware components mean that it is no longer possible to make these assumptions (if it ever was). Consequently, both anticipated and unanticipated faults must be catered for. The latter include both hardware and software design faults.

2.3.3 Redundancy

All techniques for achieving fault tolerance rely on extra elements introduced into the system to detect and recover from faults. These components are redundant in the sense that they are not required for the system's normal mode of operation. This is often called **protective redundancy**. The aim of fault tolerance is to minimize redundancy while maximizing the reliability provided, subject to the cost, size and power constraints of the system. Care must be taken in structuring fault-tolerant systems because the added components inevitably increase the complexity of the overall system. This itself can lead to *less* reliable systems. For example, the first launch of the Space Shuttle was aborted because of a synchronization difficulty with the replicated computer systems (Garman, 1981). To help reduce problems associated with the interaction between redundant components, it is therefore advisable to separate out the fault-tolerant components from the rest of the system.

There are several different classifications of redundancy, depending on which system components are under consideration and which terminology is being used. Software fault tolerance is the main focus of this chapter and therefore only passing reference will be made to hardware redundancy techniques. For hardware, Anderson and Lee (1990) distinguish between **static** (or masking) and **dynamic** redundancy. With static redundancy, redundant components are used inside a system (or subsystem) to hide the effects of faults. An example of static redundancy is **Triple Modular Redundancy** (TMR). TMR consists of three identical subcomponents and majority voting circuits. The circuits compare the output of all the components, and if one differs from the other two that output is masked out. The assumption here is that the fault is not due to a common aspect of the subcomponents (such as a design error), but is either transient or due to component deterioration. Clearly, to mask faults from more than one component requires more redundancy. The general term N **Modular Redundancy** (NMR) is therefore used to characterize this approach.

Dynamic redundancy is the redundancy supplied inside a component which indicates explicitly or implicitly that the output is in error. It therefore provides an **error detection** facility rather than an error-masking facility; recovery must be provided by another component. Examples of dynamic redundancy are checksums on communication transmissions and parity bits on memories.

For fault tolerance of software design errors, two general approaches can be identified. The first is analogous to hardware masking redundancy and is called *N*-version programming. The second is based on error detection and recovery; it is analogous to dynamic redundancy in the sense that the recovery procedures are brought into action only after an error has been detected.

2.4 *N*-version programming

The success of hardware TMR and NMR has motivated a similar approach to software fault tolerance. Here, the approach is used to focus on detecting design faults. In fact, this approach (which is now known as *N*-version programming) was first advocated by Babbage in 1837 (Randell, 1982):

> When the formula is very complicated, it may be algebraically arranged for computation in two or more distinct ways, and two or more sets of cards

> may be made. If the same constants are now employed with each set, and if under these circumstances the results agree, we may then be quite secure of the accuracy of them all.

N-version programming is defined as the independent generation of *N* (where *N* is greater than or equal to 2) functionally equivalent programs from the same initial specification (Chen and Avizienis, 1978). The independent generation of *N* programs means that *N* individuals or groups produce the required *N* versions of the software *without interaction* (for this reason *N*-version programming is often called **design diversity**). Once designed and written, the programs execute concurrently with the same inputs and their results are compared by a **driver process**. In principle, the results should be identical, but in practice there may be some difference, in which case the consensus result, assuming there is one, is taken to be correct.

N-version programming is based on the assumptions that a program can be completely, consistently and unambiguously specified, and that programs which have been developed independently will fail independently. That is, there is no relationship between the faults in one version and the faults in another. This assumption may be invalidated if each version is written in the same programming language, because errors associated with the implementation of the language may be common between versions. Consequently, different programming languages and different development environments should be used. Alternatively, if the same language is used, compilers and support environments from different manufacturers should be employed. Furthermore, in either case, to protect against physical faults, the *N* versions must be distributed to separate machines which have fault-tolerant communication lines. On the Boeing 777 flight control system, a single Ada program was produced but three different processors and three distinct compilers were used to obtain diversity.

The *N*-version program is controlled by a driver process which is responsible for:

- invoking each of the versions;
- waiting for the versions to complete;
- comparing and acting on the results.

So far it has been implicitly assumed that the programs or processes run to completion before the results are compared, but for embedded systems this often will not be the case; such processes may never complete. The driver and *N* versions must, therefore, communicate during the course of their executions.

It follows that these versions, although independent, must interact with the driver program. This interaction is specified in the requirements for the versions. It consists of three components (Chen and Avizienis, 1978):

(1) comparison vectors;
(2) comparison status indicators;
(3) comparison points.

How the versions communicate and synchronize with the driver will depend on the programming language used and its model of concurrency (see Chapters 4, 5 and 6). If different languages are used for different versions, then a real-time operating system

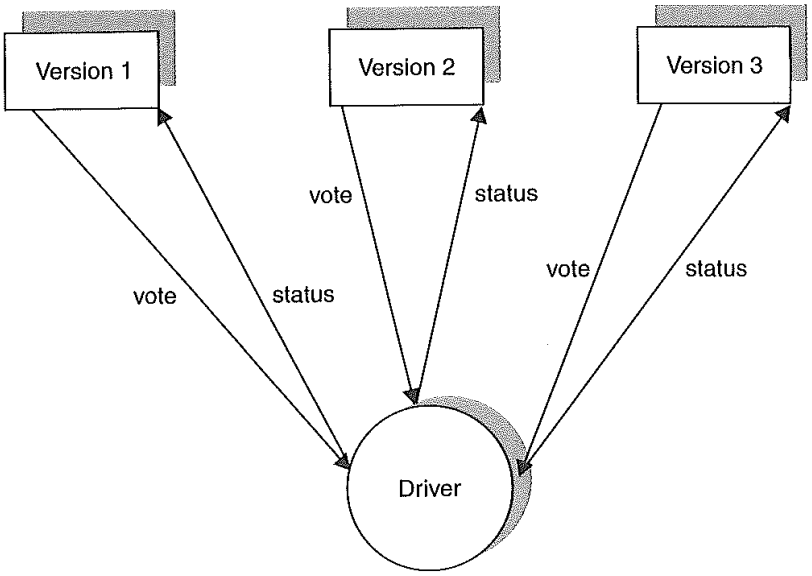

Figure 2.4 *N*-version programming.

will usually provide the means of communication and synchronization. The relationship between the N versions and the driver for an $N=3$ version system is shown diagrammatically in Figure 2.4.

Comparison vectors are the data structures which represent the outputs, or votes, produced by the versions plus any attributes associated with their calculation; these must be compared by the driver. For example, in an air traffic control system, if the values being compared are the positions of aircraft, an attribute may indicate whether the values were the result of a recent radar reading or calculated on the basis of old readings.

The comparison status indicators are communicated from the driver to the versions; they indicate the actions that each version must perform as a result of the driver's comparison. Such actions will depend on the outcome of the comparison: whether the votes agreed and whether they were delivered on time. Possible outcomes include:

- continuation;
- termination of one or more versions;
- continuation after changing one or more votes to the majority value.

The comparison points are the points in the versions where they must communicate their votes to the driver process. As Hecht and Hecht (1986) point out, an important design decision is the frequency with which the comparisons are made. This is the **granularity** of the fault tolerance provision. Fault tolerance of large granularity, that is, infrequent comparisons, will minimize the performance penalties inherent in the comparison strategies and permit a large measure of independence in the version design. However, a large granularity will probably produce a wide divergence in the results obtained because of the greater number of steps carried out between comparisons. The problems of vote comparison or voting (as it is often called) are considered in the next subsection. Fault tolerance of a fine granularity requires commonality of program structures at a detailed level, and therefore reduces the degree of independence between

versions. A frequent number of comparisons also increase the overheads associated with this technique.

2.4.1 Vote comparison

Crucial to N-version programming is the efficiency and the ease with which the driver program can compare votes and decide whether there is any disagreement. For applications which manipulate text or perform integer arithmetic there will normally be a single correct result; the driver can easily compare votes from different versions and choose the majority decision.

Unfortunately, not all results are of an exact nature. In particular, where votes require the calculation of real numbers, it will be unlikely that different versions will produce exactly the same result. This might be due to the inexact hardware representation of real numbers or the data sensitivity of a particular algorithm. The techniques used for comparing these types of results are called **inexact voting**. One simple technique is to conduct a range check using a previous estimation or a median value taken from all N results. However, it can be difficult to find a general inexact voting approach.

Another difficulty associated with finite-precision arithmetic is the so-called **consistent comparison problem** (Brilliant et al., 1987). The trouble occurs when an application has to perform a comparison based on a finite value given in the specification; the result of the comparison then determines the course of action to be taken. As an example, consider a process control system which monitors temperature and pressure sensors and then takes appropriate actions according to their values to ensure the integrity of the system. Suppose that when either of these readings passes a threshold value some corrective course of action must be taken. Now consider a 3-version software system (V_1, V_2, V_3) each of which must read both sensors, decide on some action and then vote on the outcome (there is no communication between the versions until they vote). As a result of finite-precision arithmetic, each version will calculate different values (say T_1, T_2, T_3 for the temperature sensor and P_1, P_2, P_3 for the pressure sensor). Assuming that the threshold value for temperature is T_{th} and for pressure P_{th}, the consistent comparison problem occurs when both readings are around their threshold values.

The situation might occur where T_1 and T_2 are just below T_{th} and T_3 just above; consequently V_1 and V_2 will follow their normal execution paths and V_3 will take some corrective action. Now if versions V_1 and V_2 proceed to another comparison point, this time with the pressure sensor, then it is possible that P_1 could be just below and P_2 just above P_{th}. The overall result will be that all three versions will have followed different execution paths, and therefore produce different results, each of which is valid. This process is represented diagrammatically in Figure 2.5.

At first sight, it might seem appropriate to use inexact comparison techniques and assume that the values are equal if they differ by a tolerance Δ, but as Brilliant et al. (1987) point out, the problem reappears when the values are close to the threshold value $\pm\Delta$.

Still further problems exist with vote comparison when multiple solutions to the same problem naturally exist. For example, a quadratic equation may have more than one solution. Once again disagreement is possible, even though no fault has occurred (Anderson and Lee, 1990).

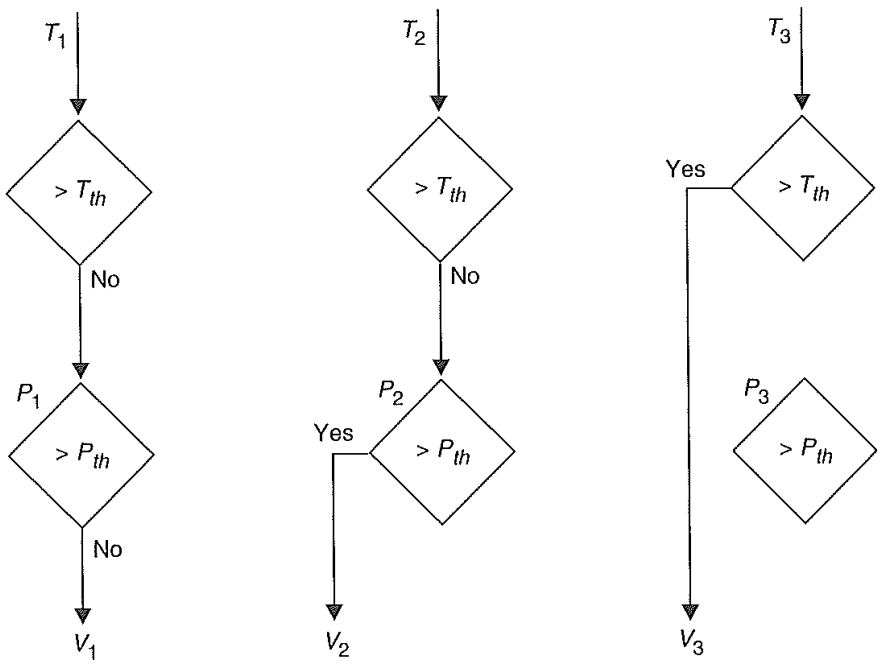

Figure 2.5 Consistent comparison problem with three versions.

2.4.2 Principal issues in *N*-version programming

It has been shown that the success of N-version programming depends on several issues, which are now briefly reviewed.

(1) **Initial specification** – it has been suggested that the great majority of software faults stem from inadequate specification (Leveson, 1986). Current techniques are a long way from producing complete, consistent, comprehensible and unambiguous specifications, although formal specification methods are proving a fruitful line of research. Clearly a specification error will manifest itself in all N versions of the implementation.

(2) **Independence of design effort** – some experiments (Knight et al., 1985; Avizienis et al., 1988; Brilliant et al., 1990; Eckhardt et al., 1991; Hatton, 1997) have been undertaken to test the hypothesis that independently produced software will display distinct failures; however, they produce conflicting results. Knight et al. (1985) have shown that for a particular problem with a thoroughly refined specification, the hypothesis had to be rejected at the far from adequate 99% confidence level. In contrast, Avizienis et al. (1988) found that it was very rare for identical faults to be found in two versions of a six-version system. In comparing their results and those produced by Knight et al., they concluded that the problem addressed by Knight et al. had limited potential for diversity, the programming process was rather informally formulated, testing was limited, and the acceptance test was totally inadequate according to common industrial standards. Avizienis et al. claim that the rigorous application of the N-version programming paradigm would have led to the elimination of all of the errors reported by Knight et al. before the acceptance of the system. However, there is concern that where part of a specification is complex this will inevitably lead to a lack of understanding of the requirements by

all the independent teams. If these requirements also refer to rarely occurring input data, then common design errors may not be caught during system testing. In more recent years, studies by Hatton (1997) found that a three-version system is still around five to nine times more reliable than a single-version high-quality system.

(3) **Adequate budget** – with most embedded systems, the predominant cost is software. A three-version system will therefore almost triple the budget requirement and cause problems for maintenance personnel. In a competitive environment, it is unlikely that a potential contractor will propose an *N*-version technique unless it is mandatory. Furthermore, it is unclear whether a more reliable system would be produced if the resources potentially available for constructing *N* versions were instead used to produce a single version.

It has also been shown that in some instances it is difficult to find inexact voting algorithms, and that unless care is taken with the consistent comparison problem, votes will differ even in the absence of faults.

Although *N*-version programming may have a role in producing reliable software it should be used with care and in conjunction with other techniques; for example, those discussed below.

2.5 Software dynamic redundancy

N-version programming is the software equivalent of static or masking redundancy, where faults inside a component are hidden from the outside. It is static because each version of the software has a fixed relationship with every other version and the driver; and because it operates whether or not faults have occurred. With **dynamic** redundancy, the redundant components only come into operation *when* an error has been detected.

This technique of fault tolerance has four constituent phases (Anderson and Lee, 1990).

(1) **Error detection** – most faults will eventually manifest themselves in the form of an error; no fault tolerance scheme can be utilized until that error is detected.

(2) **Damage confinement and assessment** – when an error has been detected, it must be decided to what extent the system has been corrupted (this is often called *error diagnosis*); the delay between a fault occurring and the manifestation of the associated error means that erroneous information could have spread throughout the system.

(3) **Error recovery** – this is one of the most important aspects of fault tolerance. Error recovery techniques should aim to transform the corrupted system into a state from which it can continue its normal operation (perhaps with degraded functionality).

(4) **Fault treatment and continued service** – an error is a symptom of a fault; although the damage may have been repaired, the fault may still exist, and therefore the error may recur unless some form of maintenance is undertaken.

Although these four phases of fault tolerance are discussed under software dynamic redundancy techniques, they can clearly be applied to *N*-version programming. As

Anderson and Lee (1990) have noted: error detection is provided by the driver which does the vote checking; damage assessment is not required because the versions are independent; error recovery involves discarding the results in error, and fault treatment is simply ignoring the version determined to have produced the erroneous value. However, if all versions have produced differing votes then error detection takes place, but there are *no* recovery facilities.

The next sections briefly cover the above phases of fault tolerance. For a fuller discussion, the reader is referred to Anderson and Lee (1990).

2.5.1 Error detection

The effectiveness of any fault-tolerant system depends on the effectiveness of its error detection techniques. Two classes of error detection techniques can be identified.

- **Environmental detection** – these are the errors which are detected in the environment in which the program executes. They include those that are detected by the hardware, such as 'illegal instruction executed', 'arithmetic overflow' and 'protection violation'. They also include errors detected by the run-time support system for the real-time programming language; for example, 'array bounds error', 'null pointer referenced' and 'value out of range'. These types of error will be considered in the context of the Ada and Java programming languages in Chapter 3.
- **Application detection** – these are the errors that are detected by the application itself. The majority of techniques that can be used by the application fall into the following broad categories.
  - **Replication checks** – it has been shown that N-version programming can be used to tolerate software faults and that the technique can be used to provide error detection (by using two-version redundancy).
  - **Timing checks** – two types of timing check can be identified. The first involves a **watchdog timer** process that, if not reset within a certain period by a component, assumes that the component is in error. The software component must continually reset the timer to indicate that it is functioning correctly.
    In embedded systems, where timely responses are important, a second type of check is required. These enable the detection of faults associated with missed deadlines. Where deadline scheduling is performed by the underlying run-time support system, the detection of missed deadlines can be considered to be part of the environment. For example, with the Real-Time Specification for Java it is the real-time JVM that detects deadline misses. However, an Ada programmer must detect such an error in the application. The issue of tolerating timing faults is covered in detail in Chapter 13.
    Of course, timing checks do *not* ensure that a component is functioning correctly, only that it is functioning on time! Time checks should therefore be used in conjunction with other error detection techniques.
  - **Reversal checks** – these are feasible in components where there is a one-to-one (isomorphic) relationship between the input and the output. Such a check takes the output, calculates what the input should be, and then compares the value with the actual input. For example, for a component which finds the square root of a number, the reversal check is simply to square the output

and compare it with the input. (Note that inexact comparison techniques may have to be used when dealing with real numbers.)

- **Coding checks** – coding checks are used to test for the corruption of data. They are based on redundant information contained within the data. For example, a value (checksum) may be calculated and sent with the actual data to be transmitted over a communication network. When the data is received, the value can be recalculated and compared with the checksum.
- **Reasonableness checks** – these are based on knowledge of the internal design and construction of the system. They check that the state of data or value of an object is reasonable, based on its intended use. Typically with modern real-time languages, much of the information necessary to perform these checks can be supplied by programmers, as type information associated with data objects. For example, in Ada integer objects which are constrained to be within certain values can be represented by subtypes of integers which have explicit ranges. Range violation can then be detected by the run-time support system.

  Sometimes explicit reasonableness checks are included in software components; these are commonly called **assertions** and take a logical expression which evaluates at run-time to true if no error is detected.
- **Structural checks** – structural checks are used to check the integrity of data objects such as lists or queues. They might consist of counts of the number of elements in the object, redundant pointers or extra status information.
- **Dynamic reasonableness checks** – with output emitted from some digital controllers, there is usually a relationship between any two consecutive outputs. Hence an error can be assumed if a new output is too different from the previous value.

Note that many of the above techniques may be applied also at the hardware level and therefore may result in 'environmental errors'.

2.5.2 Damage confinement and assessment

As there can be some delay between a fault occurring and an error being detected, it is necessary to assess any damage that may have occurred. While the type of error that was detected will give the error-handling routine some idea of the damage, erroneous information could have spread throughout the system and into its environment. Thus damage assessment will be closely related to the damage confinement precautions that were taken by the system's designers. Damage confinement is concerned with structuring the system so as to minimize the damage caused by a faulty component. It is also known as **firewalling**.

There are two techniques that can be used for structuring systems which will aid damage confinement: **modular decomposition** and **atomic actions**. With modular decomposition the emphasis is simply that the system should be broken down into components where each component is represented by one or more modules. Interaction between components then occurs through well-defined interfaces, and the internal details of the modules are hidden and not directly accessible from the outside. This makes it more difficult for an error in one component to be indiscriminately passed to another.

Modular decomposition provides a *static* structure to the software system in that most of that structure is lost at run-time. Equally important to damage confinement is the *dynamic* structure of the system as it facilitates reasoning about the run-time behaviour of the software. One important dynamic structuring technique is based on the use of atomic actions.

> The activity of a component is said to be atomic if there are *no* interactions between the activity and the system for the duration of the action.

That is, to the rest of the system an atomic action appears to be *indivisible* and takes place *instantaneously*. No information can be passed from within the atomic action to the rest of the system and vice versa. Atomic actions are often called **transactions** or **atomic transactions**. They are used to move the system from one consistent state to another and constrain the flow of information between components. Where two or more components share a resource then damage confinement will involve constraining access to that resource. The implementation of this aspect of atomic actions, using the communication and synchronization primitives found in modern real-time languages, will be considered in Chapter 7.

Other techniques which attempt to restrict access to resources are based on **protection mechanisms**, some of which may be supported by hardware. For example, each resource may have one or more modes of operation each with an associated access list (for example, read, write and execute). An activity of a component, or process, will also have an associated mode. Every time a process accesses a resource, the intended operation can be compared against its **access permissions** and, if necessary, access is denied.

In the time domain, damage confinement techniques focus on resource reservation techniques. Budgets can be given to processes that can be policed at run-time. This topic is covered in detail in Chapter 13.

2.5.3 Error recovery

Once an error situation has been detected and the damage assessed, error recovery procedures must be initiated. This is probably one of the most important phases of any fault-tolerance technique. It must transform an erroneous system state into one which can continue its normal operation, although perhaps with a degraded service. Two approaches to error recovery have been proposed: **forward** and **backward** recovery.

Forward error recovery attempts to continue from an erroneous state by making selective corrections to the system state. For embedded systems, this may involve making safe any aspect of the controlled environment which may be hazardous or damaged because of the failure. Although forward error recovery can be efficient, it is system specific and depends on accurate predictions of the location and cause of errors (that is, damage assessment). Examples of forward recovery techniques include redundant pointers in data structures and the use of self-correcting codes, such as Hamming Codes. An abort, or asynchronous exception, facility may also be required during the recovery action if more than one process is involved in providing the service when the error occurred.

Backward error recovery relies on restoring the system to a safe state previous to that in which the error occurred. An alternative section of the program is then executed. This has the same functionality as the fault-producing section, but uses a different algorithm. As with N-version programming, it is hoped that this alternative approach

will *not* result in the same fault recurring. The point to which a process is restored is called a **recovery point** and the act of establishing it is usually termed **checkpointing**. To establish a recovery point, it is necessary to save appropriate system state information at run-time.

State restoration has the advantage that the erroneous state has been cleared and that it does not rely on finding the location or cause of the fault. Backward error recovery can therefore be used to recover from unanticipated faults including design errors. However, its disadvantage is that it cannot undo any effects that the fault may have had in the environment of the embedded system; it is difficult to undo a missile launch, for example. Furthermore, backward error recovery can be time-consuming in execution, which may preclude its use in some real-time applications. For instance, operations involving sensor information may be time dependent, therefore costly state restoration techniques may simply not be feasible. Consequently, to improve performance **incremental checkpointing** approaches have been considered. The **recovery cache** is an example of such a system (Anderson and Lee, 1990). Other approaches include audit trails or logs; in these cases, the underlying support system must undo the effects of the process by reversing the actions indicated in the log.

With concurrent processes that interact with each other, state restoration is not as simple as so far portrayed. Consider two processes depicted in Figure 2.6. Process P_1 establishes recovery points R_{11}, R_{12} and R_{13}. Process P_2 establishes recovery points R_{21} and R_{22}. Also, the two processes communicate and synchronize their actions via IPC_1, IPC_2, IPC_3 and IPC_4. The abbreviation *IPC* is used to indicate Inter-Process Communication.

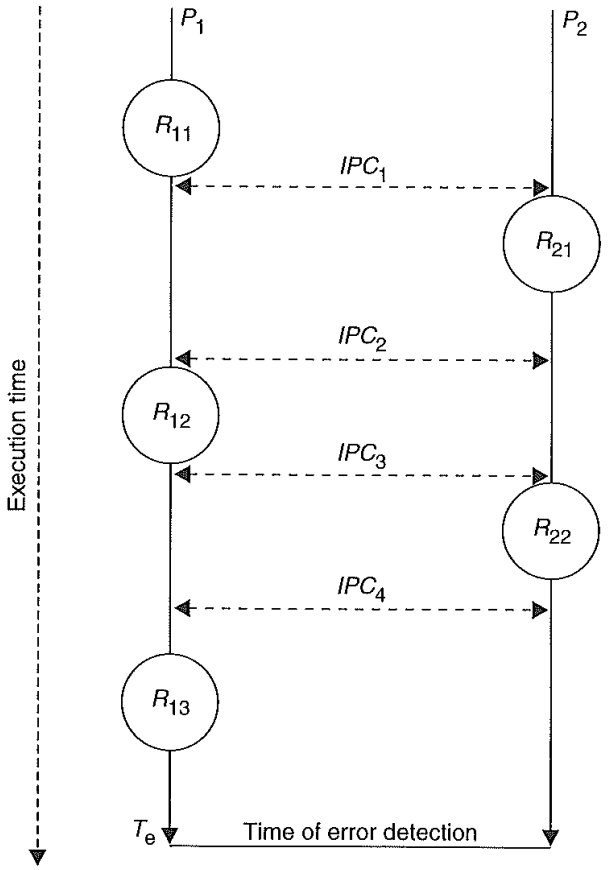

Figure 2.6 The domino effect.

If P_1 detects an error at T_e then it is simply rolled back to recovery point R_{13}. However, consider the case where P_2 detects an error at T_e. If P_2 is rolled back to R_{22} then it must undo the communication IPC_4 with P_1; this requires P_1 to be rolled back to R_{12}. But if this is done, P_2 must be rolled back to R_{21} to undo communication IPC_3, and so on. The result will be that both processes will be rolled back to the beginning of their interaction with each other. In many cases, this may be equivalent to aborting both processes! This phenomenon is known as the **domino effect**.

Obviously, if the two processes do not interact with each other then there will be no domino effect. When more than two processes interact, the possibility of the effect occurring increases. In this case, consistent recovery points must be designed into the system so that an error detected in one process will not result in a total rollback of all the processes with which it interacts; instead, the processes can be restarted from a consistent set of recovery points. These **recovery lines**, as they are often called, are closely linked with the notion of atomic actions, introduced earlier in this section. The issue of error recovery in concurrent processes will be revisited in Chapter 7. For the remainder of this chapter, sequential systems only will be considered.

The concepts of forward and backward error recovery have been introduced; each has its advantages and disadvantages. Not only do embedded systems have to be able to recover from unanticipated errors but they also must be able to respond in finite time; they may therefore require *both* forward and backward error recovery techniques. The expression of backward error recovery in sequential experimental programming languages will be considered in the next section. Mechanisms for forward error recovery will not be considered further in this chapter because it is difficult to provide in an application-independent manner. However, in the next chapter the implementation of both forms of error recovery is considered within the common framework of exception handling.

2.5.4 Fault treatment and continued service

An error is a manifestation of a fault, and although the error recovery phase may have returned the system to an error-free state, the error may recur. Therefore the final phase of fault tolerance is to eradicate the fault from the system so that normal service can be continued.

The automatic treatment of faults is difficult to implement and tends to be system-specific. Consequently, some systems make no provision for fault treatment, assuming that all faults are transient; others assume that error recovery techniques are sufficiently powerful to cope with recurring faults.

Fault treatment can be divided into two stages: fault location and system repair. Error detection techniques can help to trace the fault to a component. For a hardware component this may be accurate enough and the component can simply be replaced. A software fault can be removed in a new version of the code. However, in most non-stop applications it will be necessary to modify the program while it is executing. This presents a significant technical problem, but will not be considered further here.

2.6 The recovery block approach to software fault tolerance

Recovery blocks (Horning et al., 1974) are **blocks** in the normal programming language sense except that at the entrance to the block is an automatic **recovery point** and at

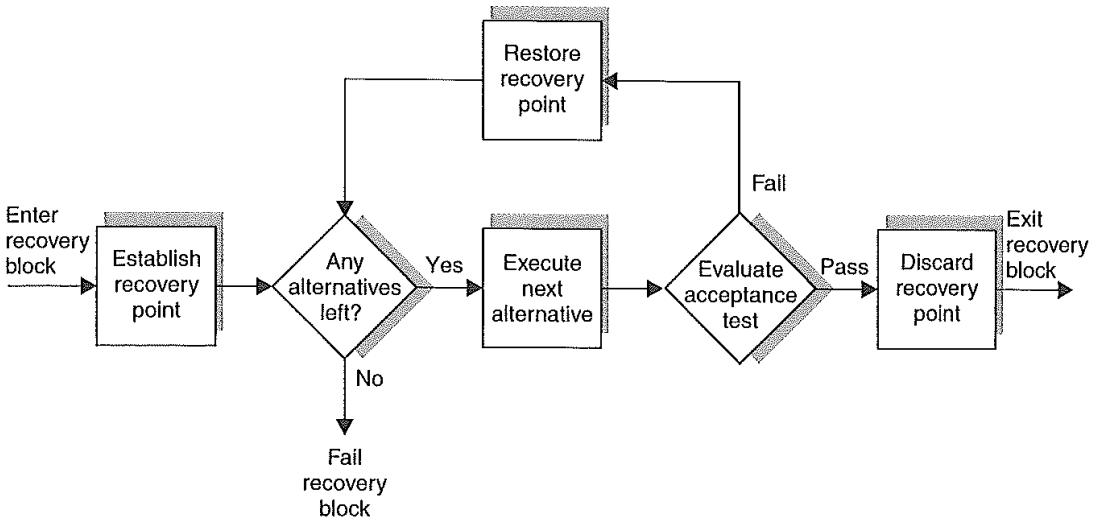

Figure 2.7 Recovery block mechanism.

the exit an **acceptance test**. The acceptance test is used to test that the system is in an acceptable state after the execution of the block (or **primary module** as it is often called). The failure of the acceptance test results in the program being restored to the recovery point at the beginning of the block and an **alternative module** being executed. If the alternative module also fails the acceptance test then again the program is restored to the recovery point and yet another module is executed, and so on. If all modules fail then the block fails and recovery must take place at a higher level. The execution of a recovery block is illustrated in Figure 2.7.

In terms of the four phases of software fault tolerance: error detection is achieved by the acceptance test, damage assessment is not needed as backward error recovery is assumed to clear all erroneous states, and fault treatment is achieved by use of a stand-by spare.

Although no commercially available real-time programming language has language features for exploiting recovery blocks, some experimental systems have been developed (Shrivastava, 1978; Purtilo and Jalote, 1991). A possible syntax for recovery blocks is illustrated below:

```
ensure <acceptance test>
by
  <primary module>
else by
  <alternative module>
else by
  <alternative module>
  ...
else by
  <alternative module>
else error
```

Like ordinary blocks, recovery blocks can be nested. If a block in a nested recovery block fails its acceptance tests and all its alternatives also fail, then the outer level recovery point will be restored and an alternative module to that block executed.

To show the use of recovery blocks, the various methods used to find the numerical solution of a system of differential equations are considered. As such methods do not give exact solutions, but are subject to various errors, it may be found that some approaches will perform better for certain classes of equations than for others. Unfortunately, methods which give accurate results across a wide range of equations are expensive to implement (in terms of the time needed to complete the method's execution). For example, an **explicit Kutta** method will be more efficient than an **implicit Kutta** method. However, it will only give an acceptable error tolerance for particular problems. There is a class of equations called **stiff** equations whose solution using an explicit Kutta leads to an accumulation of rounding errors; the more expensive implicit Kutta method can more adequately deal with this problem. The following illustrates an approach using recovery blocks which enables the cheaper method to be employed for non-stiff equations but which does not fail when stiff equations are given.

```
ensure rounding_error_within_acceptable_tolerance
by
  Explicit Kutta Method
else by
  Implicit Kutta Method
else error
```

In this example, the cheaper explicit method is usually used; however, when it fails the more expensive implicit method is employed. Although this error is anticipated, this approach also gives tolerance to an error in the design of the explicit algorithm. If the algorithm itself is in error and the acceptance test is general enough to detect both types of error result, the implicit algorithm will be used. When the acceptance test cannot be made general enough, nested recovery blocks can be used. In the following, full design redundancy is provided; at the same time the cheaper algorithm is always used if possible.

```
ensure rounding_error_within_acceptable_tolerance
by
  ensure sensible_value
  by
    Explicit Kutta Method
  else by
    Predictor-Corrector K-step Method
  else error
else by
  ensure sensible_value
  by
    Implicit Kutta Method
  else by
    Variable Order K-Step Method
  else error
else error
```

In the above, two explicit methods are given; when both methods fail to produce a sensible result, the implicit Kutta method is executed. The implicit Kutta method will, of course, also be executed if the value produced by the explicit methods is sensible but not within the required tolerance. Only if all four methods fail will the equations remain unsolved.

The recovery block could have been nested the other way around as shown below. In this case, different behaviour will occur when a non-sensible result is also not within acceptable tolerance. In the first case, after executing the explicit Kutta algorithm, the Predictor Corrector method would be attempted. In the second, the implicit Kutta algorithm would be executed.

```
ensure sensible_value
by
  ensure rounding_error_within_acceptable_margin
  by
    Explicit Kutta Method
  else by
    Implicit Kutta Method
  else error
else by
  ensure rounding_error_within_acceptable_margin
  by
    Predictor-Corrector K-step Method
  else by
    Variable Order K-Step Method
  else error
else error
```

2.6.1 The acceptance test

The acceptance test provides the error detection mechanism which then enables the redundancy in the system to be exploited. The design of the acceptance test is crucial to the efficacy of the recovery block scheme. As with all error detection mechanisms, there is a trade-off between providing comprehensive acceptance tests and keeping the overhead this entails to a minimum, so that normal fault-free execution is affected as little as possible. Note that the term used is **acceptance** not **correctness**; this allows a component to provide a degraded service.

All the error detection techniques discussed in Section 2.5.1 can be used to form the acceptance tests. However, care must be taken in their design as a faulty acceptance test may lead to residual errors going undetected.

2.7 A comparison between *N*-version programming and recovery blocks

Two approaches to providing fault-tolerant software have been described: *N*-version programming and recovery blocks. They clearly share some aspects of their basic philosophy, and yet at the same time they are quite different. This section briefly reviews and compares the two.

- **Static versus dynamic redundancy** – *N*-version programming is based on static redundancy; all versions run in parallel irrespective of whether or not a fault occurs. In contrast, recovery blocks are dynamic in that alternative modules only execute when an error has been detected.
- **Associated overheads** – both *N*-version programming and recovery blocks incur extra development cost, as both require alternative algorithms to be developed.

In addition, for N-version programming, the driver process must be designed and recovery blocks require the design of the acceptance test.

At run-time, N-version programming in general requires N times the resources of a single version. Although recovery blocks only require a single set of resources at any one time, the establishment of recovery points and the process of state restoration is expensive. However, it is possible to provide hardware support for the establishment of recovery points (Lee et al., 1980), and state restoration is only required when a fault occurs.

- **Diversity of design** – both approaches exploit diversity in design to achieve tolerance of unanticipated errors. Both are, therefore, susceptible to errors that originate from the requirements specification.
- **Error detection** – N-version programming uses vote comparison to detect errors whereas recovery blocks use an acceptance test. Where exact or inexact voting is possible there is probably less associated overhead than with acceptance tests. However, where it is difficult to find an inexact voting technique, where multiple solutions exist or where there is a consistent comparison problem, acceptance tests may provide more flexibility.
- **Atomicity** – backward error recovery is criticized because it cannot undo any damage which may have occurred in the environment. N-version programming avoids this problem because all versions are assumed not to interfere with each other: they are atomic. This requires each version to communicate with the driver process rather than directly with the environment. However, it is entirely possible to structure a program such that unrecoverable operations do not appear in recovery blocks.

It perhaps should be stressed that although N-version programming and recovery blocks have been described as competing approaches, they also can be considered as complementary ones. For example, there is nothing to stop a designer using recovery blocks within each version of an N-version system.

2.8 Dynamic redundancy and exceptions

In this section, a framework for implementing software fault tolerance is introduced which is based on dynamic redundancy and the notion of exceptions and exception handlers.

So far in this chapter, the term 'error' has been used to indicate the manifestation of a fault, where a fault is a deviation from the specification of a component. These errors can be either anticipated, as in the case of an out of range sensor reading due to hardware malfunction, or unanticipated, as in the case of a design error in the component. An **exception** can be defined as the occurrence of an error. Bringing an exception condition to the attention of the invoker of the operation which caused the exception is called **raising** (or **signalling** or **throwing**) the exception and the invoker's response is called **handling** (or **catching**) the exception. Exception handling can be considered a *forward error recovery* mechanism, as when an exception has been raised the system is not rolled back to a previous state; instead, control is passed to the handler so that recovery procedures can be initiated. However, as will be shown

in Section 3.4, the exception-handling facility can be used to provide backward error recovery.

Although an exception has been defined as the occurrence of an error, there is some controversy as to the true nature of exceptions and when they should be used. For example, consider a software component or module which maintains a compiler symbol table. One of the operations it provides is to look up a symbol. This has two possible outcomes: *symbol present* and *symbol absent.* Either outcome is an anticipated response *and* may or may not represent an error condition. If the *lookup* operation is used to determine the interpretation of a symbol in a program body, *symbol absent* corresponds to 'undeclared identifier', which is an error condition. If, however, the *lookup* operation is used during the declaration process, the outcome *symbol absent* is probably the normal case and *symbol present*, that is 'duplicate definition', the exception. What constitutes an error, therefore, depends on the context in which the event occurs. However, in either of the above cases it could be argued that the error is not an error of the symbol table component or of the compiler, in that either outcome is an anticipated result and forms part of the functionality of the symbol table module. Therefore neither outcome should be represented as an exception.

Exception-handling facilities were *not* incorporated into programming languages to cater for programmer design errors; however, it will be shown in Section 3.4 how they can be used to do just that. The original motivation for exceptions came from the requirement to handle abnormal conditions arising in the environment in which a program executes. These exceptions could be termed rare events in the functioning of the environment, and it may or may not be possible to recover from them within the program. A faulty valve or a temperature alarm might cause an exception. These are rare events which, given enough time, might well occur and must be tolerated.

Despite the above, exceptions and their handlers will inevitably be used as a general purpose error-handling mechanism. To conclude, exceptions and exception handling can be used to:

- cope with abnormal conditions arising in the environment;
- enable program design faults to be tolerated;
- provide a general-purpose error-detection and recovery facility.

Exceptions are considered in more detail in Chapter 3.

2.8.1 Ideal fault-tolerant system components

Figure 2.8 shows the ideal component from which to build fault-tolerant systems (Anderson and Lee, 1990). The component accepts service requests and, if necessary, calls upon the services of other components before yielding a response. This may be a normal response or an exception response. Two types of fault can occur in the ideal component: those due to an illegal service request, called **interface exceptions**, and those due to a malfunction in the component itself, or in the components required to service the original request. Where the component cannot tolerate these faults, either by forward or backward error recovery, it raises **failure exceptions** in the calling component. Before raising any exceptions, the component must return itself to a consistent state, if possible, in order that it may service any future request.

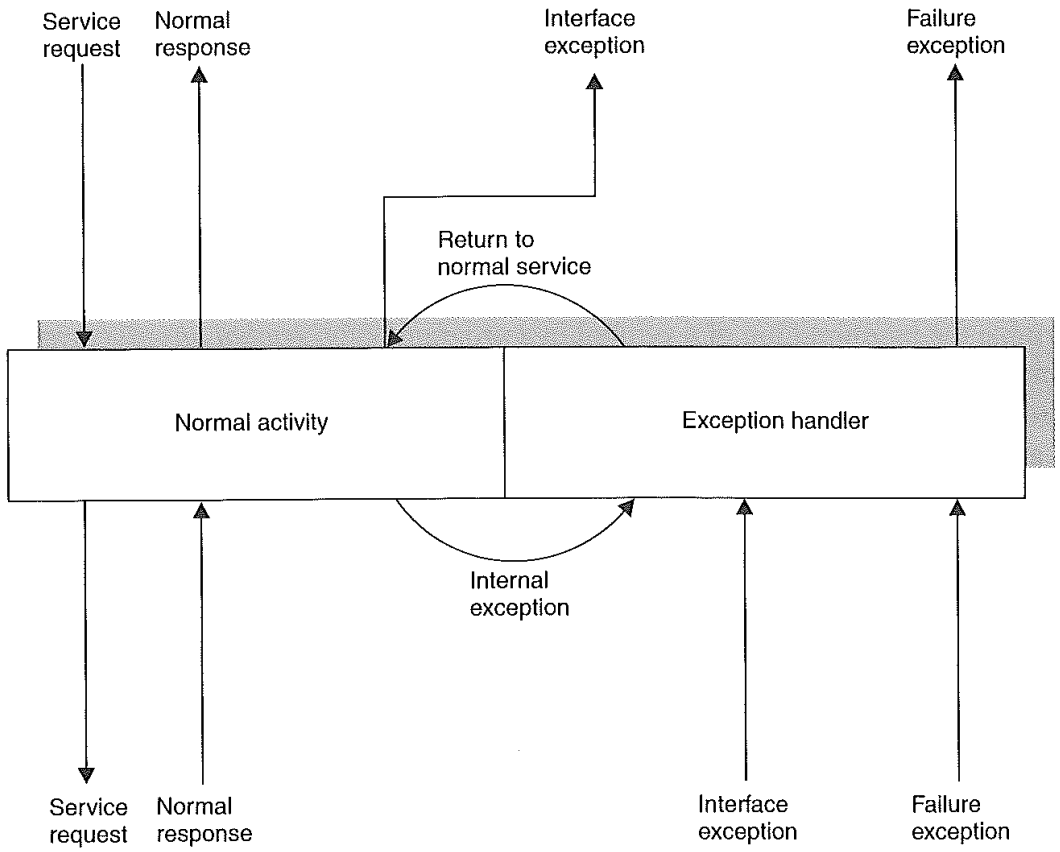

Figure 2.8 An ideal fault-tolerant component.

2.9 Measuring and predicting the reliability of software

Reliability metrics for hardware components have long been established. Traditionally, each component is regarded as a representative of a population of identical members whose reliability is estimated from the proportion of a sample that fail during a specified interval of time, e.g. during testing. Software reliability prediction and measurement, however, is not as well established a discipline. It was ignored for many years by those industries requiring extremely reliable systems because software is assumed not to deteriorate with use; software was regarded as either correct or incorrect – a binary property.

Also, in the past, particular software components were used once only, in the systems for which they were originally intended; consequently, although any errors found during testing were removed, this did not lead to the development of more reliable components which could be used elsewhere. This can be contrasted with hardware components, which are mass produced; any errors found in the design can be corrected, making the next batch more reliable.

The view that software is either correct or not correct is still commonly held. If it is not correct, program testing or program proving will indicate the location of faults which can then be corrected. This chapter has tried to illustrate that the traditional approach of software testing, although indispensable, can never ensure that programs are fault-free, especially with very large and complicated systems where there may be residual specification or design errors. Furthermore, in spite of the continual advances made in the field of proof of correctness, the application of these techniques to non-trivial systems, particularly those involving the concept of time, is still beyond the *state of the art*. Indeed

it may always be beyond the capability of such techniques due to the tendency to make systems and programs ever larger and more complex.

It is for all these reasons that methods of improving reliability through the use of redundancy have been advocated. Unfortunately, even with this approach, it cannot be guaranteed that systems containing software will not fail. It is therefore essential that techniques for assessing software reliability are developed.

As hardware is deemed to be subject to *random* failures it is natural to use a probabilistic approach for reliability assessment. It is perhaps less clear why *systematic* software failures should be characterized similarly. Although systematic in nature, the process by which any particular demand on the system will give rise to a failure is essentially non-deterministic (Littlewood et al., 2001). Software reliability can therefore be considered as *the probability that a given program will operate correctly in a specified environment for a specified length of time.*

Several models have been proposed which attempt to estimate software reliability. These can be broadly classified as (Goel and Bastini, 1985):

- software reliability growth models;
- statistical models.

Growth models attempt to predict the reliability of a program on the basis of its error history (e.g. when faults are identified and repaired). Other statistical models, attempt to estimate the reliability of a program by determining its success or failure response to a random sample of test cases, without correcting any errors found. Unfortunately, Littlewood and Strigini (1993) have argued that testing alone can only provide effective evidence for reliability estimates of at best 10^{-4} (that is 10^{-4} failures per hour of operation). This should be compared with the often quoted reliability requirement of 10^{-9} for avionics systems. To increase the assessment of reliability by an order of magnitude to 10^{-5} would required the observation of 460 000 hours (over 50 years) of fault-free operation (Littlewood et al., 2001).

To estimate the reliability of N-version components is even more difficult as the level of correlation between the versions is, as indicated earlier, very difficult to estimate. Even strong advocates of the approach would not argue that two 10^{-4} versions would combine to give a 10^{-8} service.

2.10 Safety, reliability and dependability

Safety can be defined as *freedom from those conditions that can cause death, injury, occupational illness, damage to (or loss of) equipment (or property), or environmental harm* (Leveson, 1986). However, as this definition would consider most systems which have an element of risk associated with their use as unsafe, software safety is often considered in terms of **mishaps** (Leveson, 1986). A mishap is an **unplanned event** or **series of events** that can result in death, injury, occupational illness, damage to (or loss of) equipment (or property), or environmental harm.

Although reliability and safety are often considered as synonymous, there is a difference in their emphasis. Reliability has been defined as a measure of the success with which a system conforms to some authoritative specification of its behaviour. This is usually expressed in terms of probability. Safety, however, is the probability that

conditions that can lead to mishaps do not occur *whether or not the intended function is performed*. These two definitions can conflict with each other. For example, measures which increase the likelihood of a weapon firing when required may well increase the possibility of its accidental detonation. In many ways, the only safe aeroplane is one that never takes off; however, it is not very reliable. Nevertheless, any system (or subsystem) whose primary role is to provide safety must itself be *sufficiently* reliable. For example, a secondary Nuclear Reactor Protection System (NRPS) is only required to act when other systems have failed. It provided additional safety and hence is of value if its own reliability is assessed as being no more than 10^{-4} failures per demand. The primary NRPS may be assessed to have reliability of only 10^{-3}; as long as the primary and secondary systems are independent this provides an overall reliability of at least 10^{-7}. Plant safety is only compromised if these two systems fail *and* the plant controller itself suffers a 'meltdown' failure – an exceedingly rare event in itself.

As with reliability, to ensure the safety requirements of an embedded system, system safety analysis must be performed throughout all stages of its life cycle development. It is beyond the scope of this book to enter into details of safety analysis; for a general discussion of reliability and safety issues, the reader is referred to the Further Reading section at the end of this chapter.

2.10.1 Dependability

The dependability of a system is that property of the system which allows reliance to be justifiably placed on the service it delivers. Dependability, therefore, includes as special cases the notions of reliability, safety and security (Laprie, 1995). Figure 2.9, based on that given by Laprie (1995), illustrates these and other aspects of dependability (where security is viewed in terms of integrity and confidentiality). In this figure, the term 'reliability' is used as a measure of the continuous delivery of a proper service; availability is a measure of the frequency of periods of improper service.

Dependability itself can be described in terms of three components (Laprie, 1995):

- **threats** – circumstances causing or resulting in non-dependability;
- **means** – the methods, tools and solutions required to deliver a dependable service with the required confidence;

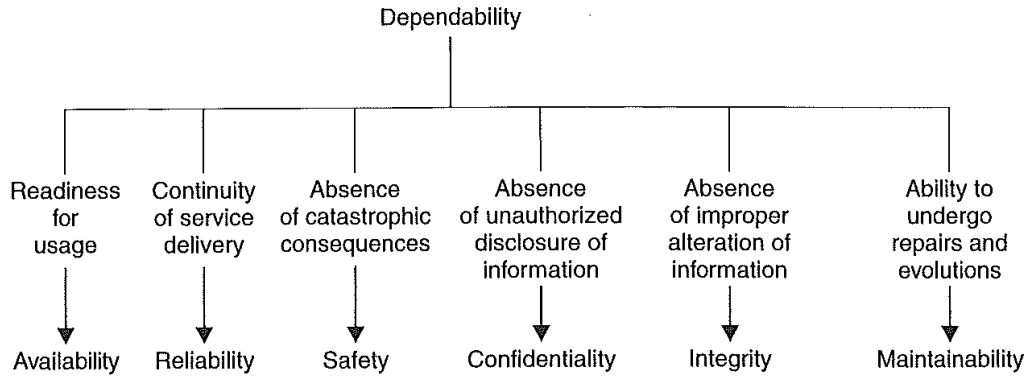

Figure 2.9 Aspects of dependability.

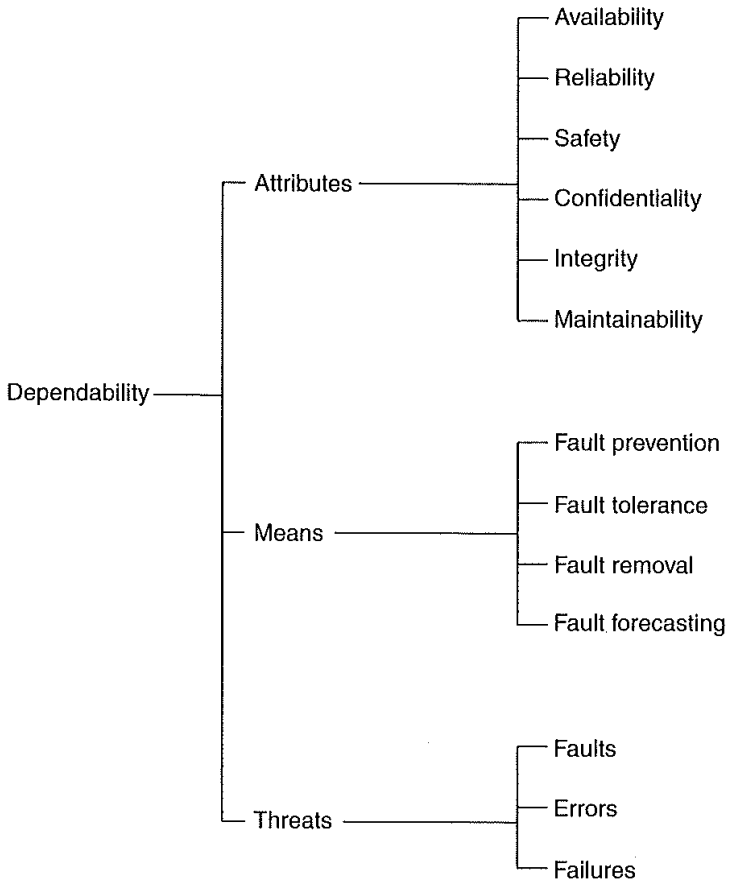

Figure 2.10 Dependability terminology.

- **attributes** – the way and measures by which the quality of a dependable service can be appraised.

Figure 2.10 summarizes the concept of dependability in terms of these three components.

Summary

This chapter has identified reliability as a major requirement for any real-time system. The reliability of a system has been defined as a measure of the success with which the system conforms to some authoritative specification of its behaviour. When the behaviour of a system deviates from that which is specified for it, this is called a failure. Failures result from faults. Faults can be accidentally or intentionally introduced into a system. They can be transient, permanent or intermittent.

There are two approaches to system design which help ensure that potential faults do not cause system failure: fault prevention and fault tolerance. Fault prevention consists of fault avoidance (attempting to limit the introduction of faulty components into the system) and fault removal (the process of finding and removing faults). Fault tolerance involves the introduction of redundant components into a system so that faults can be detected and tolerated. In general, a system will provide either full fault tolerance, graceful degradation or fail-safe behaviour.

Two general approaches to software fault tolerance have been discussed: *N*-version programming (static redundancy) and dynamic redundancy using forward and backward error recovery. *N*-version programming is defined as the independent generation of *N* (where 2 or more) functionally equivalent programs from the same initial specification. Once designed and written, the programs execute concurrently with the same inputs and their results are compared. In principle, the results should be identical, but in practice there may be some difference, in which case the consensus result, assuming there is one, is taken to be correct. *N*-version programming is based on the assumptions that a program can be completely, consistently and unambiguously specified, and that programs which have been developed independently will fail independently. These assumptions may not always be valid, and although *N*-version programming may have a role in producing reliable software it should be used with care and in conjunction with techniques based on dynamic redundancy.

Dynamic redundancy techniques have four constituent phases: error detection, damage confinement and assessment, error recovery, and fault treatment and continued service. One of the most important phases is error recovery for which two approaches have been proposed: backward and forward. With backward error recovery, it is necessary for communicating processes to reach consistent recovery points to avoid the domino effect. For sequential systems, the recovery block has been introduced as an appropriate language concept for expressing backward error recovery. Recovery blocks are blocks in the normal programming language sense except that at the entrance to the block is an automatic recovery point and at the exit an acceptance test. The acceptance test is used to test that the system is in an acceptable state after the execution of the primary module. The failure of the acceptance test results in the program being restored to the recovery point at the beginning of the block and an alternative module being executed. If the alternative module also fails the acceptance test, the program is restored to the recovery point again and yet another module is executed, and so on. If all modules fail then the block fails. A comparison between *N*-version programming and recovery blocks illustrated the similarities and differences between the approaches.

Although forward error recovery is system-specific, exception handling has been identified as an appropriate framework for its implementation. The concept of an ideal fault-tolerant component was introduced which used exceptions.

Finally in this chapter, the notions of software safety and dependability were introduced.

Further reading

Anderson, T. and Lee, P. A. (1990) *Fault Tolerance, Principles and Practice*, 2nd edn. Englewood Cliffs, NJ: Prentice Hall.

Andrews, J. D. and Moss, T. R. (2002) *Reliability and Risk Assessment*, 2nd edn. Chichester: Wiley.

De Florio, V. and Blondia, C. (2008) A survey of linguistic structures for application-level fault tolerance, *ACM Computer Surveys*, **40**(2).

Herrmann, D. S. (1999) *Software Safety and Reliability*. Los Alamitos, CA: IEEE Computer Society.

Kritzinger, D. (2006) *Aircraft System Safety – Military and Civil Aeronautical Applications*. Cambridge: Woodhead Publishing.

Laprie J.-C. et al. (1995) *Dependability Handbook*. Toulouse: Cépaduès (in French).

Leveson, N. G. (1995) *Safeware: System Safety and Computers*. Reading, MA: Addison-Wesley.

Mili, A. (1990) *An Introduction to Program Fault Tolerance*. New York: Prentice Hall.

Neumann, P. G. (1995) *Computer-Related Risks*. Reading, MA: Addison-Wesley.

Redmill, F. and Rajan, J. (eds) (1997) *Human Factors in Safety-Critical Systems*. Oxford: Butterworth-Heinemann.

Storey, N. (1996) *Safety-Critical Computer Systems*. Reading, MA: Addison-Wesley.

Exercises

2.1 Is a program reliable if it conforms to an erroneous specification of its behaviour?

2.2 What would be the appropriate levels of degraded service for a computer-controlled automobile?

2.3 Write a recovery block for sorting an array of integers.

2.4 To what extent is it possible to detect recovery lines at run-time? (See Anderson and Lee, 1990, Chapter 7.)

2.5 Figure 2.11 illustrates the concurrent execution of four communicating processes (P_1, P_2, P_3 and P_4) and their associated recovery points (for example, R_{11} is the first recovery point for process P_1).

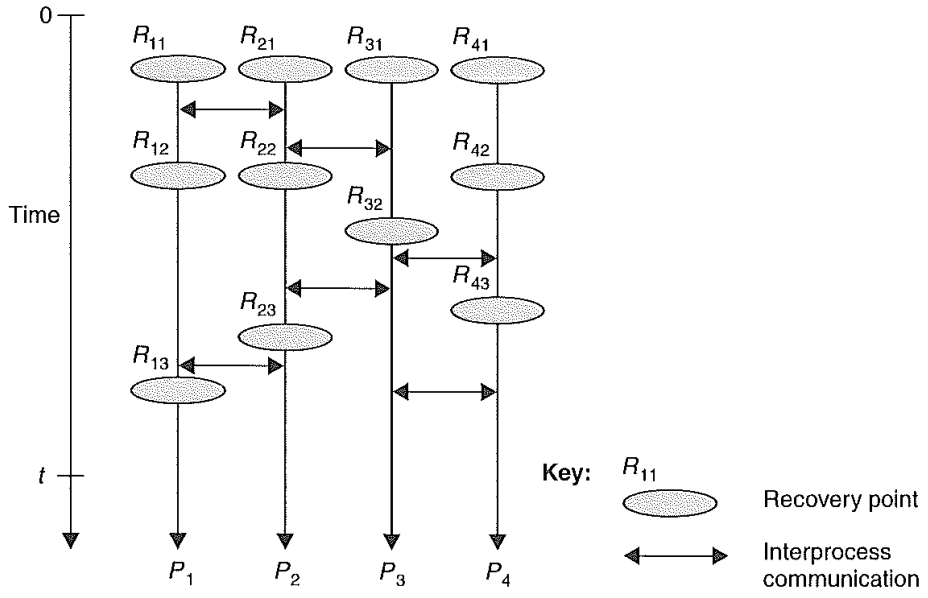

Figure 2.11 Concurrent execution of four processes for Exercise 2.5.

Explain what happens when an error is detected by:

- Process P_1 at time t;
- Process P_2 at time t.

2.6 Should the end of file condition that occurs when sequentially reading a file be signalled to the programmer as an exception?

2.7 Data diversity is a fault-tolerance strategy that complements design diversity. Under what conditions might data diversity be more appropriate than design diversity? (Hint: see Ammann and Knight, 1988.)

2.8 Should the dependability of a system be judged by an independent assessor?

Chapter 5
Shared variable-based synchronization and communication

The major difficulties associated with concurrent programming arise from task interactions. Rarely are tasks as independent of one another as they were in the simple example at the end of Chapter 4. The correct behaviour of a concurrent program is critically dependent on synchronization and communication between tasks. In its widest sense, synchronization is the satisfaction of constraints on the interleaving of the actions of different tasks (for example, a particular action by one task only occurring after a specific action by another task). The term is also used in the narrower sense of bringing two tasks simultaneously into predefined states. Communication is the passing of information from one task to another. The two concepts are linked, since some forms of communication require synchronization, and synchronization can be considered as contentless communication.

Inter-task communication is usually based upon either **shared variables** or **message passing**. Shared variables are objects to which more than one task has access; communication can therefore proceed by each task referencing these variables when appropriate. Message passing involves the explicit exchange of data between two tasks by means of a message that passes from one task to another via some agency. Note that the choice between shared variables and message passing is one for the language or operating systems designers; it does not imply that any particular implementation method should be used. Shared variables are easy to support if there is shared memory between the tasks and, hence, they are an ideal mechanism for communication between tasks in a shared memory multiprocessor system. However, they can

still be used even if the hardware incorporates a communication medium. Similarly, a message-passing primitive is an ideal abstraction for a distributed system where there is potentially no shared physical memory but, again, it can also be supported via shared memory. Furthermore, an application can arguably be programmed in either style and obtain the same functionality (Lauer and Needham, 1978).

This chapter will concentrate on shared variable-based communication and synchronization primitives. In particular, busy waiting, semaphores, conditional critical regions, monitors, protected types and synchronized methods are discussed. The impact of shared memory multiprocessors is also considered. Message-based synchronization and communication are discussed in Chapter 6.

5.1 Mutual exclusion and condition synchronization

Although shared variables appear to be a straightforward way of passing information between tasks, their unrestricted use is unreliable and unsafe due to multiple update problems. Consider two tasks updating a shared variable, `X`, with the assignment:

```
X:= X+1
```

On most hardware this will not be executed as an **indivisible** (atomic) operation, but will be implemented in three distinct instructions:

(1) load the value of `X` into some register (or to the top of the stack);

(2) increment the value in the register by `1`; and

(3) store the value in the register back to `X`.

As the three operations are not indivisible, two tasks simultaneously updating the variable could follow an interleaving that would produce an incorrect result. For example, if `X` was originally `5`, the two tasks could each load `5` into their registers, increment and then store `6`.

A sequence of statements that must appear to be executed indivisibly is called a **critical section**. The synchronization required to protect a critical section is known as **mutual exclusion**. Atomicity, although absent from the assignment operation, is assumed to be present at the memory level. Thus, if one task is executing `X := 5`, simultaneously with another executing `X := 6`, the result will be either `5` or `6` (not some other value). If this were not true, it would be difficult to reason about concurrent programs or implement higher levels of atomicity, such as mutual exclusion synchronization. Clearly, however, if two tasks are updating a structured object, this atomicity will only apply at the single word element level.

The mutual exclusion problem itself was first described by Dijkstra (1965). It lies at the heart of most concurrent task synchronizations and is of great theoretical as well as practical interest. Mutual exclusion is not, however, the only synchronization of importance; indeed, if two tasks do not share variables then there is no need for mutual exclusion. Condition synchronization is another significant requirement and is needed

when a task wishes to perform an operation that can only sensibly, or safely, be performed if another task has itself taken some action or is in some defined state.

An example of condition synchronization comes with the use of buffers. Two tasks that exchange data may perform better if communication is not direct but via a buffer. This has the advantage of de-coupling the tasks and allows for small fluctuations in the speed at which the two tasks are working. For example, an input task may receive data in bursts that must be buffered for the appropriate user task. The use of a buffer to link two tasks is common in concurrent programs and is known as a **producer–consumer** system.

Two condition synchronizations are necessary if a finite (bounded) buffer is used. Firstly, the producer task must not attempt to deposit data into the buffer if the buffer is full. Secondly, the consumer task cannot be allowed to extract objects from the buffer if the buffer is empty. Moreover, if simultaneous deposits or extractions are possible, mutual exclusion must be ensured so that two producers, for example, do not corrupt the 'next free slot' pointer of the buffer.

The implementation of any form of synchronization implies that tasks must at times be held back until it is appropriate for them to proceed. In Section 5.2, mutual exclusion and condition synchronization will be programmed (in pseudo code with explicit task declaration) using **busy-wait** loops and **flags**. From this analysis, it should be clear that further primitives are needed to ease the coding of algorithms that require synchronization.

5.2 Busy waiting

One way to implement synchronization is to have tasks set and check shared variables that are acting as flags. This approach works reasonably well for implementing condition synchronization, but no simple method for mutual exclusion exists. To signal a condition, a task sets the value of a flag; to wait for this condition, another task checks this flag and proceeds only when the appropriate value is read.

```
task P1;  -- pseudo code for waiting task
  ...
  while flag = down do
    null
  end;
  ...
end P1;

task P2;  -- signalling task
  ...
  flag := up;
  ...
end P2;
```

If the condition is not yet set (that is, flag is still down) then `P1` has no choice but to loop round and recheck the flag. This is **busy waiting**; also known as **spinning** (with the flag variables called **spin locks**).

Busy-wait algorithms are in general inefficient; they involve tasks using up processing cycles when they cannot perform useful work. Even on a multiprocessor system,

they can give rise to excessive traffic on the memory bus or network (if distributed). Moreover, it is not possible to impose queuing disciplines easily if there is more than one task waiting on a condition (that is, checking the value of a flag). More seriously, they can leave to **livelock**. This is an error condition where tasks get stuck in their busy-wait loops and are unable to make progress.

Mutual exclusion presents even more difficulties as the algorithms required are more complex. Consider two tasks (`P1` and `P2` again) that have mutual critical sections. In order to protect access to these critical sections, it can be assumed that each task executes an entry protocol before the critical section and an exit protocol afterwards. Each task can therefore be considered to have the following form.

```
task P; -- pseudo code
  loop
    entry protocol
      critical section
    exit protocol
    non-critical section
  end
end P;
```

An algorithm is presented below that provides mutual exclusion and absence of livelock. It was first presented by Peterson (1981). The approach of Peterson is to have two flags (`flag1` and `flag2`) that are manipulated by the task that 'owns' them and a `turn` variable that is only used if there is contention for entry to the critical sections.

```
task P1; -- pseudo code
  loop
    flag1:= up;     -- announce intent to enter
    turn:= 2;       -- give priority to other task
    while flag2 = up and turn = 2 do
      null;
    end;
    <critical section>
    flag1:= down;
    <non-critical section>
  end
end P1;

task P2;
  loop
    flag2:= up;     -- announce intent to enter
    turn:= 1;       -- give priority to other task
    while flag1 = up and turn = 1 do
      null;
    end;
    <critical section>
    flag2:= down;
    <non-critical section>
  end
end P2;
```

If only one task wishes to enter its critical section then the other task's flag will be down and entry will be immediate. However, if both flags have been raised then the value of `turn` becomes significant. Let us say that it has the initial value `1`; then there are four possible interleavings, depending on the order in which each task assigns a value to `turn` and then checks its value in the while statement:

```
First Possibility -- P1 first then P2
P1 sets turn to 2
P1 checks turn and enters busy loop
P2 sets turn to 1 (turn will now stay with that value)
P2 checks turn and enters busy loop
P1 loops around rechecks turn and enters critical section

Second Possibility -- P2 first then P1
P2 sets turn to 1
P2 checks turn and enters busy loop
P1 sets turn to 2 (turn will now stay with that value)
P1 checks turn and enters busy loop
P2 loops around rechecks turn and enters critical section

Third Possibility -- interleaved P1 and P2
P1 sets turn to 2
P2 sets turn to 1 (turn will stay with this value)
P2 enters busy loop
P1 enters critical section

Fourth Possibility -- interleaved P2 and P1
P2 sets turn to 1
P1 sets turn to 2 (turn will stay with this value)
P1 enters busy loop
P2 enters critical section
```

All four possibilities lead to one task in its critical section and one task in a busy loop.

In general, although a single interleaving can only illustrate the failure of a system to meet its specification, it is not possible to show easily that all possible interleavings lead to compliance with the specification. Normally, proof methods (including model checking) are needed to show such compliance.

Interestingly, the above algorithm is fair in the sense that if there is contention for access (to their critical sections) and, say, `P1` was successful (via either the first or third possible interleaving) then `P2` is bound to enter next. When `P1` exits its critical section, it lowers `flag1`. This could let `P2` into its critical section, but even if it does not (because `P2` was not actually executing at that time) then `P1` would proceed, enter and leave its non-critical section, raise `flag1`, set `turn` to `2` and then be placed in a busy loop. There it would remain until `P2` had entered and left its critical section and reset `flag2` as its exit protocol.

In terms of reliability, the failure of a task in its non-critical section will not affect the other task. This is not the case with failure in the protocols or critical section. Here, premature termination of a task would lead to livelock difficulties for the remaining program.

This discussion has been given at length to illustrate the difficulties of implementing synchronization between tasks with only shared variables and no additional primitives

other than those found in sequential languages. These difficulties can be summarized as follows.

- Protocols that use busy loops are difficult to design, understand and prove correct. (The reader might like to consider generalizing Peterson's algorithm for n tasks.)
- Testing programs may not examine rare interleavings that break mutual exclusion or lead to livelock.
- Busy-wait loops are inefficient.
- An unreliable (rogue) task that misuses shared variables will corrupt the entire system.

No concurrent programming language relies entirely on busy waiting and shared variables; other methods and primitives have been introduced. For shared-variable systems, semaphores and monitors are the most significant constructs and are described in Sections 5.4 and 5.6.

5.3 Suspend and resume

One of the problems with busy-wait loops is that they waste valuable processor time. An alternative approach is to suspend (that is, remove from the set of runnable tasks) the calling task if the condition for which it is waiting does not hold. Consider, for example, simple condition synchronization using a `flag`. One task sets the flag, and another task waits until the flag is set and then clears it. A simple suspend and resume mechanism could be used as follows:

```
task P1;  -- pseudo code for waiting task
  ...
  if flag = down do
    suspend;
  end;
  flag := down;
  ...
end P1;

task P2;  -- signalling task
  ...
  flag := up;
  resume P1; -- has no effect, if P1 is not suspended
  ...
end P2;
```

An early version of the Java `Thread` class provided the following methods in support of this approach.

```
public final void suspend();
                  // throws SecurityException;
public final void resume();
                  // throws SecurityException;
```

Thus the above example would be represented in Java.

```
boolean flag;
final boolean up = true;
final boolean down = false;

class FirstThread extends Thread {

  public void run() {
    ...
    if(flag == down) {
      suspend();
    };
    flag = down;
    ...
  }
}

class SecondThread extends Thread { // T2

  FirstThread T1;

  public SecondThread(FirstThread T) {
    super();
    T1 = T;
  }

  public void run() {
    ...
    flag = up;
    T1.resume();
    ...
  }
}
```

Unfortunately, this approach suffers from what is called a **data race condition**.

> *A data race condition is a fault in the design of the interactions between two or more tasks whereby the result is unexpected and critically dependent on the sequence or timing of accesses to shared data.*

In this case, thread `T1` could test the `flag`, and then the underlying run-time support system (or operating system) could decide to preempt it and run `T2`. `T2` sets the `flag` and `resumes` `T1`. `T1` is, of course, not suspended, so the `resume` has no effect. Now, when `T1` next runs, it thinks the `flag` is `down` and therefore suspends itself.

The reason for this problem is that the `flag` is a shared resource which is being tested and an action is being taken which depends on its status (the thread is suspending itself). This testing and suspending is not an atomic operation, and therefore interference can occur from other threads. It is for this reason that the most recent version of Java has made these methods obsolete.

There are several well-known solutions to this race condition problem, all of which provide a form of **two-stage suspend** operation. `P1` essentially has to announce that it

Program 5.1 Synchronous task control.

```
package Ada.Synchronous_Task_Control is
  type Suspension_Object is limited private;
  procedure Set_True(S : in out Suspension_Object);
  procedure Set_False(S : in out Suspension_Object);
  function Current_State(S : Suspension_Object) return Boolean;
  procedure Suspend_Until_True(S: in out Suspension_Object);
    -- raises Program_Error if more than one task tries
    -- to suspend on S at once.
private
  -- not specified by the language
end Ada.Synchronous_Task_Control;
```

is planning to suspend in the near future; any resume operation which finds that `P1` is not suspended will have a deferred effect. When `P1` does suspend, it will immediately be resumed; that is, the suspend operation itself will have no effect.

Although suspend and resume is a low-level facility, which can be error-prone in its use, it is an efficient mechanism which can be used to construct higher-level synchronization primitives. For this reason, Ada provides, as part of its Real-Time Annex, a safe version of this mechanism. It is based around the concept of a **suspension** object, which can hold the value `True` or `False`. Program 5.1 gives the package specification.

All four subprograms defined by the package are atomic with respect to each other. On return from the `Suspend_Until_True` procedure, the referenced suspension object is reset to `False`.

The simple condition synchronization problem, given earlier in this section, can, therefore, be easily solved.

```
with Ada.Synchronous_Task_Control;
use Ada.Synchronous_Task_Control;

...
Flag : Suspension_Object;
...
task body P1 is
begin
  ...
  Suspend_Until_True(Flag);
  ...
end P1;

task body P2 is
begin
  ...
  Set_True(Flag);
  ...
end P2;
```

Suspension objects behave in much the same way as binary semaphores, which are discussed in Section 5.4.4.

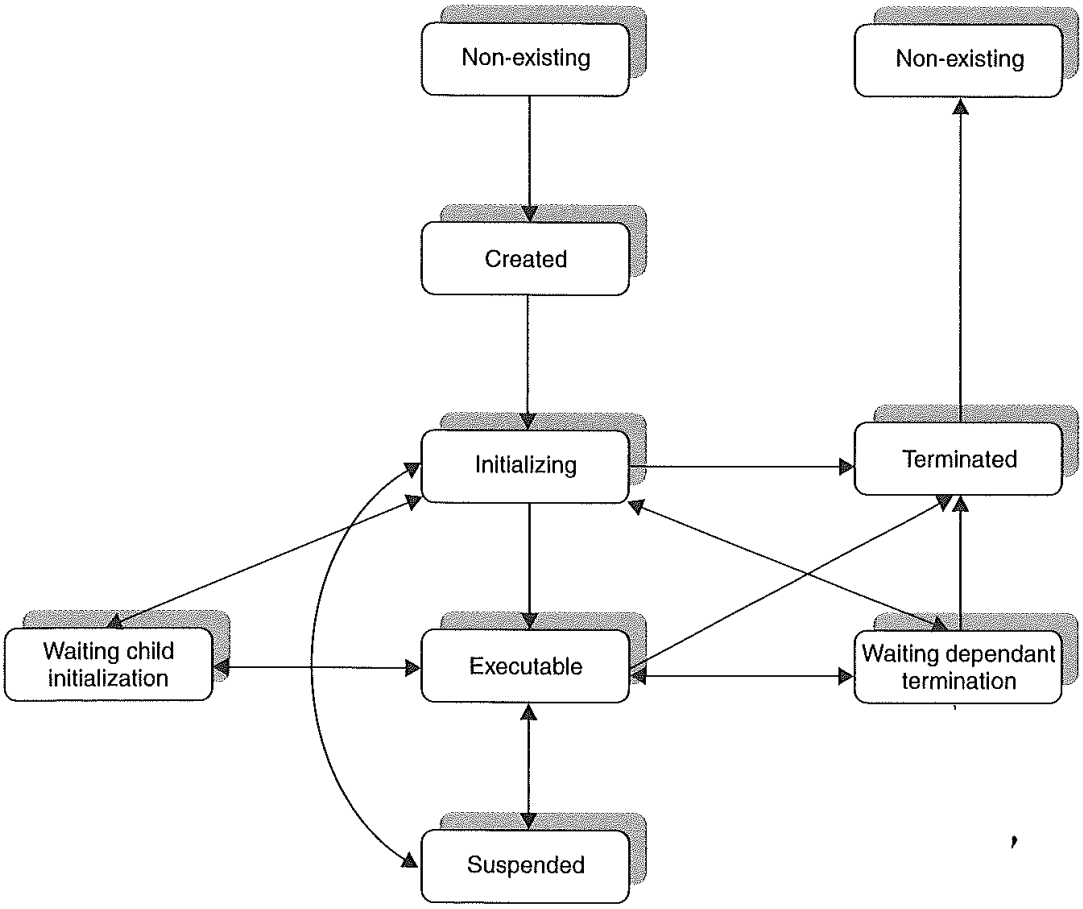

Figure 5.1 State diagram for a task.

Although `suspend` and `resume` are useful low-level primitives, no operating system or language relies solely on these mechanisms for mutual exclusion and condition synchronization. If present, they clearly introduce a new state into the state transition diagram introduced in Chapter 4. The general state diagram for a task, therefore, is extended in Figure 5.1.

5.4 Semaphores

Semaphores are a simple mechanism for programming mutual exclusion and condition synchronization. They were originally designed by Dijkstra (1968) and have the following two benefits.

(1) They simplify the protocols for synchronization.
(2) They remove the need for busy-wait loops.

A **semaphore** is a non-negative integer variable that, apart from initialization, can only be acted upon by two procedures. These procedures are called `wait` and `signal` in this book. The semantics of `wait` and `signal` are as follows.

(1) `wait(S)` – If the value of the semaphore, `S`, is greater than zero then decrement its value by one; otherwise delay the task until `S` is greater than zero (and then decrement its value).

(2) `signal(S)` – Increment the value of the semaphore, `S`, by one.

General semaphores are often called **counting semaphores**, as their operations increment and decrement an integer count. The additional important property of `wait` and `signal` is that their actions are atomic (indivisible). Two tasks, both executing `wait` operations on the same semaphore, cannot interfere with each other. Moreover, a task cannot fail during the execution of a semaphore operation.

Condition synchronization and mutual exclusion can be programmed easily with semaphores. First, consider condition synchronization:

```
-- pseudo code for condition synchronization
consyn : semaphore; -- initially 0
task P1; -- waiting task
  ...
  wait(consyn);
  ...
end P1;

task P2; -- signalling task
  ...
  signal(consyn);
  ...
end P2;
```

When `P1` executes the `wait` on a `0` semaphore, it will be delayed until `P2` executes the `signal`. This will set `consyn` to `1` and hence the `wait` can now succeed; `P1` will continue and `consyn` will be decremented to `0`. Note that if `P2` executes the `signal` first, the semaphore will be set to `1`, so `P1` will not be delayed by the action of the `wait`.

Mutual exclusion is similarly straightforward:

```
-- pseudo code for mutual exclusion

mutex : semaphore; -- initially 1
task P1;
  loop
    wait(mutex);
      <critical section>
    signal(mutex);
    <non-critical section>
  end
end P1;

task P2;
  loop
    wait (mutex);
      <critical section>
    signal (mutex);
    <non-critical section>
  end
end P2;
```

If P1 and P2 are in contention then they will execute their wait statements simultaneously. However, as wait is atomic, one task will complete execution of this statement before the other begins. One task will execute a wait(mutex) with mutex=1, which will allow the task to proceed into its critical section and set mutex to 0; the other task will execute wait(mutex) with mutex=0, and be delayed. Once the first task has exited its critical section, it will signal(mutex). This will cause the semaphore to become 1 again and allow the second task to enter its critical section (and set mutex to 0 again).

With a wait/signal bracket around a section of code, the initial value of the semaphore will restrict the maximum amount of concurrent execution of the code. If the initial value is 0, no task will ever enter; if it is 1 then a single task may enter (that is, mutual exclusion); for values greater than one, the given number of concurrent executions of the code is allowed.

5.4.1 Suspended tasks

In the definition of wait it is clear that if the semaphore is zero then the calling task is delayed. One method of delay (busy waiting) has already been introduced and criticized. A more efficient mechanism, that of suspending the task, was introduced in Section 5.3. In fact, all synchronization primitives deal with delay by some form of suspension; the task is removed from the set of executable tasks.

When a task executes a wait on a zero semaphore, the RTSS (run-time support system) is invoked, the task is removed from the processor, and placed in a queue of suspended tasks (that is a queue of tasks suspended on that particular semaphore). The RTSS must then select another task to run. Eventually, if the program is correct, another task will execute a signal on that semaphore. As a result, the RTSS will pick out one of the suspended tasks awaiting a signal on that semaphore and make it executable again.

From these considerations, a slightly different definition of wait and signal can be given. This definition is closer to what an implementation would do:

```
-- pseudo code for wait(S)
if S > 0 then
  S:= S-1;
else
  number_suspended := number_suspended + 1
  suspend_calling_task;

-- pseudo code signal(S)
if number_suspended > 0 then
  number_suspended := number_suspended - 1;
  make_one_suspended_task_executable_again;
else
  S:= S+1;
end if;
```

With this definition, the increment of a semaphore immediately followed by its decrement is avoided.

Note that the above algorithm does not define the order in which tasks are released from the suspended state. Usually, they are released in a FIFO order, although arguably

with a true concurrent language, the programmer should assume a non-deterministic order (see Section 6.6). However, for a real-time programming language, the priority of the tasks has an important role to play (see Chapter 11).

5.4.2 Implementation

The above algorithm for implementing a semaphore is quite straightforward, although it involves the support of a queue mechanism. Where difficulty could arise is in the requirement for indivisibility in the execution of the `wait` and `signal` operations. Indivisibility means that once a task has started to execute one of these procedures it will continue to execute until the operation has been completed. With the aid of the RTSS, this is easily achieved; the scheduler is programmed so that it does not swap out a task while it is executing a `wait` or a `signal`; they are **non-preemptible** operations.

Unfortunately, the RTSS is not always in full control of scheduling events. Although all internal actions are under its influence, external actions happen asynchronously and could disturb the atomicity of the semaphore operations. To prohibit this, the RTSS will typically disable interrupts for the duration of the execution of the indivisible sequence of statements. In this way, no external events can interfere.

This disabling of interrupts is adequate for a single processor system but not for a multiprocessor one. With a shared-memory system, two parallel tasks may be executing a `wait` or `signal` (on the same semaphore) and the RTSS is powerless to prevent it. In these circumstances, a 'lock' mechanism is need to protect access to the operations. Two such mechanisms are used.

On some processors, a 'test and set' instruction is provided. This allows a task to access a bit in the following way.

(1) If the bit is zero then set it to one and return zero.

(2) If the bit is one return one.

These actions are themselves indivisible. Two parallel tasks, both wishing to operate a `wait` (for example), will do a test and set operation on the same lock bit (which is initially zero). One task will succeed and set the bit to one; the other task will have returned a one and will, therefore, have to loop round and retest the lock. When the first task has completed the wait operation, it will assign the bit to zero (that is, unlock the semaphore) and the other task will proceed to execute its `wait` operation.

If no test and set instruction is available then a similar effect can be obtained by a swap instruction. Again, the lock is associated with a bit that is initially zero. A task wishing to execute a semaphore operation will swap a one with the lock bit. If it gets a zero back from the lock then it can proceed; if it gets back a one then some other task is active with the semaphore and it must retest.

As was indicated in Section 5.1, a software primitive such as a semaphore cannot conjure up mutual exclusion out of 'fresh air'. It is necessary for memory locations to exhibit the essence of mutual exclusion in order for higher-level structures to be built. Similarly, although busy-wait loops are removed from the programmer's domain by the use of semaphores, it may be necessary to use busy waits (as above) to implement the wait and signal operations. *It should be noted, however, that the latter use of busy-waits is only short-lived (the time it takes to execute a wait or signal operation), whereas their*

use for delaying access to the program's critical sections could involve many seconds of looping.

5.4.3 Liveness provision

In Section 5.2, the error condition livelock was illustrated. Unfortunately (but inevitably), the use of synchronization primitives introduces other error conditions. **Deadlock** is the most serious such condition and entails a set of tasks being in a state from which it is impossible for any of them to proceed. This is similar to livelock but the tasks are suspended. To illustrate this condition, consider two tasks `P1` and `P2` wishing to gain access to two non-concurrent resources (that is, resources that can only be accessed by one task at a time) that are protected by two semaphores `S1` and `S2`. If both tasks access the resource in the same order then no problem arises:

```
  P1                              P2
wait (S1);                      wait (S1);
  wait (S2);                      wait (S2);
  .                               .
  .                               .
  .                               .
  signal (S2);                    signal (S2);
signal (S1);                    signal (S1);
```

The first task to execute the `wait` on `S1` successfully will also successfully undertake the `wait` on `S2` and subsequently `signal` the two semaphores and allow the other task in. A problem occurs, however, if one of the tasks wishes to use the resources in the reverse order, for example:

```
  P1                              P2
wait (S1);                      wait (S2);
  wait (S2);                      wait (S1);
  .                               .
  .                               .
  .                               .
  signal (S2);                    signal (S1);
signal (S1);                    signal (S2);
```

In this case, an interleaving could allow `P1` and `P2` to execute successfully the `wait` on `S1` and `S2`, respectively, but then inevitably both tasks will be suspended waiting on the other semaphore which is now zero.

It is in the nature of an interdependent concurrent program that usually once a subset of the tasks becomes deadlocked all the other tasks will eventually become part of the deadlocked set.

The testing of software rarely removes other than the most obvious deadlocks; they can occur infrequently but with devastating results. This error is not isolated to the use of semaphores and is possible in all concurrent programming languages. The design of languages that prohibit the programming of deadlocks is a desirable, but not yet attainable, goal. Issues relating to deadlock avoidance, detection and recovery will be considered in Chapters 8 and 11.

Indefinite postponement (sometimes called **lockout** or **starvation**) is a less severe error condition whereby a task that wishes to gain access to a resource, via a critical section, is never allowed to do so because there are always other tasks gaining access before it. With a semaphore system, a task may remain indefinitely suspended (that is, queued on the semaphore) due to the way the RTSS picks tasks from this queue when a signal arrives. Even if the delay is not in fact indefinite, but merely open ended (indeterminate), this may give rise to an error in a real-time system.

If a task is free from livelocks, deadlocks and indefinite postponements then it is said to possess **liveness**. Informally, the liveness property implies that if a task wishes to perform some action then it will, eventually, be allowed to do so. In particular, if a task requests access to a critical section it will gain access within a finite time.

5.4.4 Binary and quantity semaphores

The definition of a (general) semaphore is a non-negative integer; by implication its actual value can rise to any supported positive number. However, in all the examples given so far in this chapter (that is, for condition synchronization and mutual exclusion), only the values 0 and 1 have been used. A simple form of semaphore, known as a **binary semaphore**, can be implemented that takes only these values; that is, the signalling of a semaphore which has the value 1 has no effect – the semaphore retains the value 1. The construction of a general semaphore from two binary semaphores and an integer can then be achieved, if the general form is required.

Another variation on the normal definition of a semaphore is the **quantity semaphore**. With this structure, the amount to be decremented by the `wait` (and incremented by the `signal`) is not fixed as 1, but is given as a parameter to the procedures:

```
wait(S, i) :-   if S >= i then
                   S := S-i
                else
                   delay
                   S := S-i
signal(S, i) :- S := S+i
```

5.4.5 Example semaphore programs in Ada

Algol-68 was the first language to introduce semaphores. It provided a type `sema` that was manipulated by the operators `up` and `down`. To illustrate some simple programs that use semaphores, an abstract data type for semaphores, in Ada, will be used.

```
package Semaphore_Package is
  type Semaphore(Initial : Natural := 1) is limited private;
  procedure Wait (S : in out Semaphore);
  procedure Signal (S : in out Semaphore);
private
  type Semaphore is ...
end Semaphore_Package;
```

Ada does not directly support semaphores, but the `Wait` and `Signal` procedures can, however, be constructed from the Ada synchronization primitives; these have not yet

been discussed, so the full definition of the type semaphore and the body of the package will not be given here (see Section 5.8). The essence of abstract data types is, however, that they can be used without knowledge of their implementation.

The first example is the producer/consumer system that uses a bounded buffer to pass integers between the two tasks:

```
procedure Main is
  package Buffer is
    procedure Append (I : Integer);
    procedure Take (I : out Integer);
  end Buffer;
  task Producer;
  task Consumer;

  package body Buffer is separate;
  use Buffer;

  task body Producer is
    Item : Integer;
  begin
    loop
      -- produce item
      Append (Item);
    end loop;
  end Producer;

  task body Consumer is
    Item : Integer;
  begin
    loop
      Take (Item);
      -- consume item
    end loop;
  end Consumer;
begin
  null;
end Main;
```

The buffer itself must protect against concurrent access, appending to a full buffer and taking from an empty one. This it does by the use of three semaphores:

```
with Semaphore_Package; use Semaphore_Package;
separate (Main)
package body Buffer is
  Size : constant Natural := 32;
  type Buffer_Range is mod Size;
  Buf : array (Buffer_Range) of Integer;
  Top, Base : Buffer_Range := 0;

   Mutex : Semaphore; -- default is 1
   Item_Available : Semaphore(0);
   Space_Available : Semaphore(Initial => Size);

   procedure Append (I : Integer) is
   begin
```

```
    Wait(Space_Available);
    Wait(Mutex);
      Buf(Top) := I;
      Top := Top + 1;
    Signal(Mutex);
    Signal(Item_Available);
  end Append;

  procedure Take (I : out Integer) is
  begin
    Wait(Item_Available);
    Wait(Mutex);
      I := Buf(Base);
      Base := Base + 1;
    Signal(Mutex);
    Signal(Space_Available);
  end Take;
end Buffer;
```

The initial values of the three semaphores are different. `Mutex` is an ordinary mutual exclusion semaphore and is given the default initial value of 1; `Item_Available` protects against taking from an empty buffer and has the initial value 0; and `Space_Available` (initially `Size`) is used to prevent `Append` operations to a full buffer.

When the program starts, any consumer task that calls `Take` will be suspended on `Wait(Item_Available)`; only after a producer task has called `Append`, and in doing so `Signal(Item_Available)`, will the consumer task continue.

5.4.6 Semaphore programming using Java

Although the Java language supports a monitor-like communication and synchronization model (see Section 5.9), the Java platform provides several standard packages that support concurrency utilities. One of these provides general-purpose classes to support different synchronization approaches. Semaphores are included in this package.

5.4.7 Semaphore programming using C/Real-Time POSIX

Although few modern programming languages support semaphores directly, many operating systems do. The POSIX API, for example, provides counting semaphores to enable processes running in separate address spaces (or threads within the same address space) to synchronize and communicate using shared memory. Note, however, that it is more efficient to use mutexes and condition variables to synchronize and communicate in the same address space – see Section 5.7. Program 5.2 defines the C/Real-Time POSIX interface for semaphores (functions for naming a semaphore by a character string are also provided but have been omitted here). The standard semaphore operations *initialize*, *wait* and *signal* are called `sem_init`, `sem_wait` and `sem_post` in C/Real-Time POSIX. A non-blocking wait (`sem_trywait`) and a timed-version (`sem_timedwait`) are also provided, as is a routine to determine the current value of a semaphore (`sem_getvalue`).

Consider an example of a resource controller which appears in many forms in real-time programs. For simplicity, the example will use threads rather than processes. Two functions are provided: `allocate` and `deallocate`; each takes a parameter

Program 5.2 The C/Real-Time POSIX interface to semaphores.

```
#include <time.h>
typedef ... sem_t;
int sem_init(sem_t *sem_location, int pshared, unsigned int value);
  /* initializes the semaphore at location sem_location to value */
  /* if pshared is 1, the semaphore can be used between processes */
  /*    or threads */
  /* if pshared is 0, the semaphore can only be used between threads */
  /* of the same process */

int sem_destroy(sem_t *sem_location);
  /* remove the unnamed semaphore at location sem_location */

int sem_wait(sem_t *sem_location);
  /* a standard wait operation on a semaphore */

int sem_trywait(sem_t *sem_location);
 /* attempts to decrement the semaphore */
 /* returns -1 if the call might block the calling process */

int sem_timedwait(sem_t *sem, const struct timespec *abstime);
 /* returns -1 if the semaphore could not be locked */
 /* by abstime */

int sem_post(sem_t *sem_location);
 /* a standard signal operation on a semaphore */

int sem_getvalue(sem_t *sem_location, int *value);
 /* gets the current value of the semaphore to a location */
 /* pointed at by value; negative value indicates the number */
 /* of threads waiting */


/* All the above functions return 0 if successful, otherwise -1. */
/* When an error condition is returned by any of the above */
/* functions, a shared variable errno contains the reason for */
/* the error */
```

which indicates a priority level associated with the request. It is assumed that the calling thread deallocates the resource at the same priority with which it requested allocation. For ease of presentation, the example does not consider how the resource itself is transferred. Moreover, the solution does not protect itself against race conditions (see Exercise 5.21).

```
#include <semaphore.h>

typedef enum {high, medium, low} priority_t;
typedef enum {false, true} boolean;

 sem_t mutex;    /* used for mutual exclusive
                    access to waiting and busy */
 sem_t cond[3]; /* used for condition synchronization */
```

```
int waiting;    /* count of number of threads
                   waiting at a priority level */
int busy; /* indicates whether the resource is in use*/

void allocate(priority_t P)
{
  SEM_WAIT(&mutex); /* lock mutex */
  if(busy) {
    SEM_POST(&mutex); /* release mutex */
    SEM_WAIT(&cond[P]); /* wait at correct priority level */
    /* resource has been allocated */
  }
  busy = true;
  SEM_POST(&mutex); /* release mutex */
}
```

A single semaphore, `mutex`, is used to ensure that all allocation and deallocation requests are handled in mutual exclusion. Three condition synchronization semaphores, `cond[3]`, are used to queue the waiting threads at three priority levels (`high`, `medium` and `low`). The `allocate` function allocates the resource if it is not already in use (indicated by the `busy` flag).

The deallocation function simply signals the semaphore of the highest priority waiter.

```
int deallocate(priority_t P)
{
  SEM_WAIT(&mutex); /* lock mutex */
  if(busy) {
    busy = false;
    /* release highest priority waiting thread */
    SEM_GETVALUE(&cond[high],&waiting);
    if (waiting < 0)  {
      SEM_POST(&cond[high]);
    }
    else {
      SEM_GETVALUE(&cond[medium],&waiting);
      if (waiting < 0)  {
         SEM_POST(&cond[medium]);
      }
      else {
        SEM_GETVALUE(&cond[low],&waiting);
        if (waiting < 0)  {
          SEM_POST(&cond[low]);
        }
        else SEM_POST(&mutex);
        /* no one waiting, release lock */
      }
    }
    /* resource and lock passed on to */
    /* highest priority waiting thread */
    return 0;
  }
  else return -1; /* error return */
}
```

An initialization routine sets the `busy` flag to `false` and creates the four semaphores used by `allocate` and `deallocate`.

```
void initialize() {
  priority_t i;

  busy = false;
  SEM_INIT(&mutex, 0, 1);
  for (i = high; i<= low; i++) {
    SEM_INIT(&cond[i], 0, 0);
  };
}
```

Remember that, as the C binding to Real-Time POSIX uses non-zero return values to indicate an error has occurred, it is necessary to encapsulate every POSIX call in an `if` statement. This makes the code more difficult to understand (an Ada or C++ binding to POSIX would allow exceptions to be raised when errors occur). Consequently, as with the other C examples used in this book, `SYS_CALL` is used to represent a call to `sys_call` and any appropriate error recovery (see Section 3.1.1). For `SEM_INIT` this might include a retry.

A thread wishing to use the resource would make the following calls:

```
priority_t my_priority;

...
allocate(my_priority); /* wait for resource */
  /* use resource */
if(deallocate(my_priority) <= 0) {
    /* cannot deallocate resource, */
    /* undertake some recovery operation */
}
```

5.4.8 Criticisms of semaphores

Although the semaphore is an elegant low-level synchronization primitive, a real-time program built only upon the use of semaphores is again error-prone. It needs just one occurrence of a semaphore to be omitted or misplaced for the entire program to collapse at run-time. Mutual exclusion may not be assured and deadlock may appear just when the software is dealing with a rare but critical event. What is required is a more structured synchronization primitive.

What the semaphore provides is a means to program mutual exclusion over a critical section. A more structured approach would give mutual exclusion directly. This is precisely what is provided for by the constructs discussed in Sections 5.5 to 5.9.

The examples shown in Section 5.4.5 showed that an abstract data type for semaphores can be constructed in Ada. However, no high-level concurrent programming language relies entirely on semaphores. They are important historically but are arguably not adequate for the real-time domain.

5.5 Conditional critical regions

Conditional critical regions (CCRs) are an attempt to overcome some of the problems associated with semaphores. A critical region is a section of code that is guaranteed to be executed in mutual exclusion. This must be compared with the concept of a critical section that should be executed under mutual exclusion (but in error may not be). Clearly, the programming of a critical section as a critical region immediately meets the requirement for mutual exclusion.

Variables that must be protected from concurrent usage are grouped together into named regions and are tagged as being resources. Processes are prohibited from entering a region in which another task is already active. Condition synchronization is provided by guards on the regions. When a task wishes to enter a critical region, it evaluates the guard (under mutual exclusion); if the guard evaluates true, it may enter, but if it is false, the task is delayed. As with semaphores, the programmer should not assume any order of access if more than one task is delayed attempting to enter the same critical region (for whatever reason).

To illustrate the use of CCRs, an outline of the bounded buffer program is given below.

```
-- pseudo code
program buffer_eg;
  type buffer_t is record
    slots       : array(1..N) of character;
    size        : integer range 0..N;
    head, tail  : integer range 1..N;
  end record;

  buffer : buffer_t;

  resource buf : buffer;

  task producer;
    ...
    loop
      region buf when buffer.size < N do
        -- place char in buffer etc
      end region;
      ...
    end loop;
  end,

  task consumer;
    ...
    loop
      region buf when buffer.size > 0 do
        -- take char from buffer etc
      end region
      ...
    end loop;
  end;
end;
```

One potential performance problem with CCRs is that tasks must re-evaluate their guards every time a CCR naming that resource is left. A suspended task must become executable again in order to test the guard; if it is still false, it must return to the suspended state.

A version of CCRs has been implemented in Edison (Brinch-Hansen, 1981), a language intended for embedded applications, implemented on multiprocessor systems. Each processor only executes a single task so that it may continually evaluate its guards if necessary. However, this may cause excess traffic on the network.

5.6 Monitors

The main problem with conditional critical regions is that they can be dispersed throughout the program. Monitors are intended to alleviate this problem by providing more structured control regions. They also use a form of condition synchronization that is more efficient to implement.

The intended critical regions are written as procedures and are encapsulated together into a single module called a monitor. As a module, all variables that must be accessed under mutual exclusion are hidden; additionally, as a monitor, all procedure calls into the module are guaranteed to execute with mutual exclusion.

Monitors appeared as a refinement of conditional critical regions. They can be found in numerous programming languages including Modula-1, Concurrent Pascal and Mesa.

To continue, for comparison, with the bounded buffer example, a buffer monitor would have the following structure:

```
monitor buffer; -- pseudo code
  export append, take;
  -- declaration of necessary variables

  procedure append (I : integer);
    ...
  end;

  procedure take (I : integer);
    ...
  end;
begin
  -- initialization of monitor variables
end
```

With languages that support monitors, concurrent calls to append and/or take (in the above example) are serialized – by definition. No mutual exclusion semaphore needs be provided by the programmer. The languages run-time support system will implement the appropriate entry and exit protocols.

Although providing for mutual exclusion, there is still a need for condition synchronization within the monitor. In theory, semaphores could still be used, but normally a simpler synchronization primitive is introduced. In Hoare's monitors (Hoare, 1974), this primitive is called a **condition variable** and is acted upon by two operators which,

because of similarities with the semaphore structure, will again be called `wait` and `signal`. When a task issues a `wait` operation, it is blocked (suspended) and placed on a queue associated with that condition variable (this can be compared with a `wait` on a semaphore with a value of zero; however, note that a `wait` on a condition variable *always* blocks unlike a `wait` on a semaphore). A blocked task then releases its mutually exclusive hold on the monitor, allowing another task to enter. When a task executes a `signal` operation, it will release one blocked task. If no task is blocked on the specified variable then the *signal has no effect.* (Again note the contrast with signal on a semaphore, which always has an effect on the semaphore. Indeed, wait and signal for monitors are more akin to suspend and resume in their semantics.) The bounded buffer example can now be given in full:

```
monitor buffer; -- pseudo code
  export append, take;
  constant size = 32;
  buf : array[0...size-1] of integer;
  top, base : 0..size-1;
  SpaceAvailable, ItemAvailable : condition;
  NumberInBuffer : integer;

  procedure append (I : integer);
  begin
    if NumberInBuffer = size then
      wait(SpaceAvailable);
    buf[top] := I;
    NumberInBuffer := NumberInBuffer+1;
    top := (top+1) mod size;
    signal(ItemAvailable);
  end append;

  procedure take (var I : integer);
  begin
    if NumberInBuffer = 0 then
      wait(ItemAvailable);
    I := buf[base];
    base := (base+1) mod size;
    NumberInBuffer := NumberInBuffer-1;
    signal(SpaceAvailable);
  end take;

begin -- initialization
  NumberInBuffer := 0;
  top := 0;
  base := 0;
end;
```

If a task calls (for example) `take` when there is nothing in the buffer, then it will become suspended on `ItemAvailable`. A task appending an item will, however, signal this suspended task when an item does become available.

The semantics for `wait` and `signal`, given above, are not complete; as they stand, two or more tasks could become active within a monitor. This would occur following a signal operation in which a blocked task was freed. The freed task and the one that

freed it are then both executing inside the monitor. To prohibit this clearly undesirable activity, the semantics of `signal` must be modified. Four different approaches are used in languages.

(1) A signal is allowed only as the last action of a task before it leaves the monitor (this is the case with the buffer example above).
(2) A signal operation has the side-effect of executing a return statement; that is, the task is forced to leave the monitor.
(3) A signal operation which unblocks another task has the effect of blocking itself; this task will only execute again when the monitor is free.
(4) A signal operation which unblocks another task does not block and the freed task must compete for access to the monitor once the signalling task exits.

In case (3), which was proposed by Hoare in his original paper on monitors, the tasks that are blocked because of a signal action are placed on a 'ready queue' and are chosen, when the monitor is free, in preference to tasks blocked on entry. In case (4), it is the freed task which is placed on the 'ready queue'.

C/Real-Time POSIX, Ada and Java all support variation of the monitor approach and will be considered in detail in the Sections 5.7–5.9.

5.6.1 Nested monitor calls

A nested monitor call occurs where a monitor procedure calls a procedure defined within another monitor. This can cause problems when the nested procedure suspends on a condition variable. The mutual exclusion in the last monitor call will be relinquished by the task, due to the semantics of the wait and equivalent operations. However, mutual exclusion will not be relinquished in the monitors from which the nested call has been made. Processes that attempt to invoke procedures in these monitors will become blocked. This can have performance implications, since blockage will decrease the amount of concurrency exhibited by the system.

Various approaches to the nested monitor problem have been suggested. The most popular one, adopted by Java and C/Real-Time POSIX, is to maintain the lock. Other approaches include prohibiting nested procedure calls altogether and providing constructs which specify that certain monitor procedures may release their mutual exclusion lock during remote calls.

5.6.2 Criticisms of monitors

The monitor gives a structured and elegant solution to mutual exclusion problems such as the bounded buffer. It does not, however, deal well with condition synchronizations, resorting to low-level semaphore-like primitives. All the criticisms surrounding the use of semaphores apply equally (if not more so) to condition variables.

In addition, although monitors encapsulate all the entities concerned with a resource, and provide the important mutual exclusion, their internal structure may still be difficult to understand due to the use of condition variables.

5.7 Mutexes and condition variables in C/Real-Time POSIX

In Section 5.4.7, C/Real-Time POSIX semaphores were described as a mechanism for use between processes and between threads. If the threads extension to C/Real-Time POSIX is supported then using semaphores for communication and synchronization between threads in the same address space is expensive as well as being unstructured. **Mutexes** and **condition variables**, when combined, provide the functionality of a monitor but with a procedural interface. Programs 5.3 and 5.4 define the basic C interface. Program 5.3 defines the attributes associated with mutexes and condition variables. As with pthreads, each is defined by a separate object.

Program 5.3 The C/Real-Time POSIX interface to mutexes and condition variable attributes.

```
typedef ... pthread_mutex_t;
typedef ... pthread_mutexattr_t;
typedef ... pthread_cond_t;
typedef ... pthread_condattr_t;

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
  /* destroy the mutex attribute object */
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
  /* initialize a mutex attribute object */

int pthread_mutexattr_getpshared(const pthread_mutexattr_t *
       restrict attr, int *restrict pshared);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
       int pshared);
  /* get and set the attribute that indicates that the mutex */
  /* can between threads in different processes */

int pthread_mutexattr_gettype(
       const pthread_mutexattr_t *restrict attr,
       int *restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
  /* get and set the attribute that defines the amount of */
  /* error detection that is undertaken when mutexes are used. */
  /* e.g unlocking a unlocked mutex */

int pthread_condattr_init();
int pthread_condattr_destroy();
  /* initialize and destroy a condition attribute object */
  /* undefined behaviour if threads are waiting on the */
  /* condition variable when it is destroyed */
int pthread_condattr_getpshared();
int pthread_condattr_setpshared();
  /* get and set the attribute that indicates that the condition */
  /* can between threads in different processes */

  ...
  /* other scheduling related attributes */
```

Program 5.4 The C/Real-Time POSIX interface to mutexes and condition variables.

```
typedef ... pthread_mutex_t;
typedef ... pthread_mutexattr_t;
typedef ... pthread_cond_t;
typedef ... pthread_condattr_t;

int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
  /* initializes a mutex with certain attributes */
int pthread_mutex_destroy(pthread_mutex_t *mutex);
  /* destroys a mutex */
  /* undefined behaviour if the mutex is locked */

int pthread_mutex_lock(pthread_mutex_t *mutex);
  /* lock the mutex; if locked already suspend calling thread */
  /* the owner of the mutex is the thread which locked it */
int pthread_mutex_trylock(pthread_mutex_t *mutex);
  /* as above, but gives an error return if mutex is already locked */
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            const struct timespec *abstime);
  /* as for lock, but return an error if the lock cannot */
  /* be obtained by the timeout */

int pthread_mutex_unlock(pthread_mutex_t *mutex);
  /* unlocks the mutex if called by the owning thread */
  /* when successful, results in a blocked thread being released */

int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
  /* called by thread which owns a locked mutex */
  /* atomically blocks the calling thread on the cond variable and */
  /* releases the lock on mutex */
  /* a successful return indicates that the mutex has been locked */
int pthread_cond_timedwait(pthread_cond_t *cond,
          pthread_mutex_t *mutex, const struct timespec *abstime);
  /* the same as pthread_cond_wait, except that a error is returned */
  /* if the timeout expires */

int pthread_cond_signal(pthread_cond_t *cond);
  /* unblocks at least one blocked thread */
  /* no effect if no threads are blocked */
  /* unblocked threads automatically contend for the associated mutex */
int pthread_cond_broadcast(pthread_cond_t *cond);
  /* unblocks all blocked threads */
  /* no effect if no threads are blocked */
  /* unblocked threads automatically contend for the associated mutex */

/* All the above functions return 0 if successful */
```

Each monitor has an associated (initialized) `mutex` variable, and all operations on the monitor (critical regions) are surrounded by calls to lock (`pthread_mutex_lock`) and unlock (`pthread_mutex_unlock`) the `mutex`.

Condition synchronization is provided by associating condition variables with the `mutex`. Note that, when a thread waits on a condition variable (`pthread_cond_wait`, `pthread_cond_timedwait`), its lock on the associated `mutex` is released. Also, when it successfully returns from the conditional wait, it again holds the lock. However, because more than one thread could be released (even by `pthread_cond_signal`), the program must again test for the condition that caused it to wait initially.

Consider the following integer bounded buffer using mutexes and condition variables. The buffer consists of a `mutex`, two condition variables (`buffer_not_full` and `buffer_not_empty`), a count of the number of items in the buffer, the buffer itself, and the positions of the first and last items in the buffer. The `append` routine locks the buffer and if the buffer is full, waits on the condition variable `buffer_not_full`. When the buffer has space, the integer data item is placed in the buffer, the `mutex` is unlocked and the `buffer_not_empty` signal sent. The `take` routine is similar in structure.

```
#include <pthreads.h>

#define BUFF_SIZE 10

typedef struct {
  pthread_mutex_t mutex;
  pthread_cond_t buffer_not_full;
  pthread_cond_t buffer_not_empty;
  int count, first, last;
  int buf[BUFF_SIZE];
  } buffer;

int append(int item, buffer *B ) {
  PTHREAD_MUTEX_LOCK(&B->mutex);
  while(B->count == BUFF_SIZE)
    PTHREAD_COND_WAIT(&B->buffer_not_full, &B->mutex);
  /* put data in the buffer and update count and last */
  PTHREAD_COND_SIGNAL(&B->buffer_not_empty);
  PTHREAD_MUTEX_UNLOCK(&B->mutex);
  return 0;
}

int take(int *item, buffer *B ) {
  PTHREAD_MUTEX_LOCK(&B->mutex);
  while(B->count == 0)
    PTHREAD_COND_WAIT(&B->buffer_not_empty, &B->mutex);
  /* get data from the buffer and update count and first */
  PTHREAD_COND_SIGNAL(&B->buffer_not_full);
  PTHREAD_MUTEX_UNLOCK(&B->mutex);
  return 0;
}

/* an initialize() function is also required */
```

Although mutexes and condition variables act as a type of monitor, their semantics do differ when a thread is released from a conditional wait and other threads are trying

to gain access to the critical region. With C/Real-Time POSIX, it is unspecified which thread succeeds unless priority-based scheduling is being used (see Section 12.6).

Read/write locks and barriers

Mutexes are mutual exclusion locks that allow threads read and write access to the shared data. On occasions more flexible locking is required. For example, a thread may only require a lock to read the data. Hence, multiple threads that only wish to read the data can access it concurrently. For these occasions, C/Real-Time POSIX provides **read/write locks**. They are similar to mutexes except:

- the lock operation specifies whether a read or a write lock is required (`pthread_rwlock_rdlock`, `pthread_rwlock_wrlock`);
- the full range of attributes are not supported – for example there is no support for priority inversion avoidance (see Section 11.8).

The other useful mechanism that supports C/Real-Time POSIX pthreads are **barriers**. A barrier is a simple mechanism that allows threads to be blocked until a number of them have arrived at the barrier. As with all pthread mechanisms they have attributes and an initialize function. The barrier is initialized with the number of threads required. The threads then call the `pthread_barrier_wait` function and the function does not return until the required number have arrived.

5.8 Protected objects in Ada

The criticism of monitors centres on their use of condition variables. By replacing this approach to synchronization by the use of guards, a more structured abstraction is obtained. This form of monitor will be termed a **protected object**. Ada is the only major language that provides this mechanism, and hence it will be described in terms of Ada.

A protected object in Ada encapsulates data items and allows access to them only via protected subprograms or protected entries. The language guarantees that these subprograms and entries will be executed in a manner that ensures that the data is updated under mutual exclusion. Condition synchronization is provided by having boolean expressions on entries (these are guards but are termed **barriers** in Ada) that must evaluate to `True` before a task is allowed entry. Consequently, protected objects are rather like monitors and conditional critical regions. They provide the structuring facility of monitors with the high-level synchronization mechanism of conditional critical regions.

A protected unit may be declared as a type or as a single instance; it has a specification and a body (hence it is declared in a similar way to a task). Its specification may contain functions, procedures and entries.

The following declaration illustrates how protected types can be used to provide simple mutual exclusion:

```
-- a simple integer
protected type Shared_Integer(Initial_Value : Integer) is
  function Read return  Integer;
  procedure Write(New_Value : Integer);
```

```
  procedure Increment(By : Integer);
private
  The_Data : Integer := Initial_Value;
end Shared_Integer;

My_Data : Shared_Integer(42);
```

The above protected type encapsulates a shared integer. The object declaration `My_Data` declares an instance of the protected type and passes the initial value for the encapsulated data. The encapsulated data can now only be accessed by the three subprograms: `Read`, `Write` and `Increment`.

A protected procedure provides mutually exclusive *read/write* access to the data encapsulated. In this case, concurrent calls to the procedure `Write` or `Increment` will be executed in mutual exclusion; that is, only one can be executing at any one time.

Protected functions provide concurrent *read-only* access to the encapsulated data. In the above example, this means that many calls to `Read` can be executed simultaneously. However, calls to a protected function are still executed mutually exclusively with calls to a protected procedure. A `Read` call cannot be executed if there is a currently executing procedure call; a procedure call cannot be executed if there are one or more concurrently executing function calls. The body of the `Shared_Integer` is simply:

```
protected body Shared_Integer is
  function Read return Integer is
  begin
    return The_Data;
  end Read;

  procedure Write(New_Value : Integer) is
  begin
    The_Data := New_Value;
  end Write;

  procedure Increment(By : Integer) is
  begin
    The_Data := The_Data + By;
  end Increment;
end Shared_Integer;
```

A protected entry is similar to a protected procedure in that it is guaranteed to execute in mutual exclusion and has *read/write* access to the encapsulated data. However, a protected entry is guarded by a boolean expression (the barrier) inside the body of the protected object; if this barrier evaluates to `False` when the entry call is made, the calling task is suspended until the barrier evaluates to `True` and no other tasks are currently active inside the protected object. Hence protected entry calls can be used to implement condition synchronization.

Consider a bounded buffer shared between several tasks. The specification of the buffer is:

```
-- a bounded buffer

Buffer_Size : constant Integer := 10;
```

```
type Index is mod Buffer_Size ;
subtype Count is Natural range 0 .. Buffer_Size;
type Buffer is array (Index) of Data_Item;

protected type Bounded_Buffer is
  entry Get(Item: out Data_Item);
  entry Put(Item: in Data_Item);
private
  First : Index := Index'First;
  Last : Index := Index'Last;
  Number_In_Buffer : Count := 0;
  Buf : Buffer;
end Bounded_Buffer;

My_Buffer : Bounded_Buffer;
```

Two entries have been declared; these represent the public interface of the buffer. The data items declared in the private part are those items which must be accessed under mutual exclusion. In this case, the buffer is an array and is accessed via two indices; there is also a count indicating the number of items in the buffer. The body of this protected type is given below.

```
protected body Bounded_Buffer is
  entry Get(Item: out Data_Item)
      when Number_In_Buffer /= 0 is
  begin
    Item := Buf(First);
    First := First + 1; -- mod types cycle around
    Number_In_Buffer := Number_In_Buffer - 1;
  end Get;

  entry Put(Item: in Data_Item)
      when Number_In_Buffer /= Buffer_Size is
  begin
    Last := Last + 1;
    Buf(Last) := Item;
    Number_In_Buffer := Number_In_Buffer + 1;
  end Put;
end Bounded_Buffer;
```

The `Get` entry is guarded by the barrier '**when** `Number_In_Buffer /= 0`'; only when this evaluates to `True` can a task execute the `Get` entry; similarly with the `Put` entry. Barriers define a precondition; only when they evaluate to `True` can the entry be accepted.

Although calls to a protected object can be delayed because the object is in use (that is, they cannot be executed with the requested read or read/write access), Ada does not view the call as being suspended. Calls which are delayed due to an entry barrier being false are, however, considered suspended and placed on a queue. The reason for this is:

- it is assumed that protected operations are short-lived;

- once started a protected operation cannot suspend its execution – all calls which are potentially suspending are prohibitive and raise exceptions – it can only requeue (see Section 8.4).

Hence a task should not be delayed for a significant period while attempting to access the protected object – other than for reasons associated with the order of scheduling. Once a procedure (or function) call has gained access it will immediately start to execute the subprogram; an entry call will evaluate the barrier and will, of course, be blocked if the barrier is false. In Section 12.3, the implementation strategy required by the Real-Time Systems Annex is considered which guarantees that a task is never delayed when trying to gain access to a protected object.

5.8.1 Entry calls and barriers

To issue a call to a protected object, a task simply names the object and the required subprogram or entry. For example, to place some data into the above bounded buffer requires the calling task to:

```
My_Buffer.Put(Some_Item);
```

At any instant in time, a protected entry is either open or closed. It is open if, when checked, the boolean expression evaluates to `True`; otherwise it is closed. Generally, the protected entry barriers of a protected object are evaluated when:

(1) a task calls one of its protected entries and the associated barrier references a variable or an attribute which might have changed since the barrier was last evaluated;

(2) a task leaves a protected procedure or protected entry and there are tasks queued on entries whose barriers reference variables or attributes which might have changed since the barriers were last evaluated.

Barriers are not evaluated as a result of a protected function call. Note that it is not possible for two tasks to be active within a protected entry or procedure as the barriers are only evaluated when a task leaves the object.

When a task calls a protected entry or a protected subprogram, the protected object may already be locked: if one or more tasks are executing protected functions inside the protected object, the object is said to have an active **read lock**; if a task is executing a protected procedure or a protected entry, the object is said to have an active **read/write lock**.

If more than one task calls the same closed barrier then the calls are queued, by default, in a first-come, first-served fashion. However, this default can be changed (see Section 12.3).

Two more examples will now be given. Consider first the simple resource controller given earlier. When only a single resource is requested (and released) the code is straightforward:

```
protected Resource_Control is
  entry Allocate;
  procedure Deallocate;
```

```
private
  Free : Boolean := True;
end Resource_Control;

protected body Resource_Control is
  entry Allocate when Free is
  begin
    Free := False;
  end Allocate;
  procedure Deallocate is
  begin
    Free := True;
  end Deallocate;
end Resource_Control;
```

The resource is initially available and hence the `Free` flag is true. A call to `Allocate` changes the flag, and therefore closes the barrier; all subsequent calls to `Allocate` will be blocked. When `Deallocate` is called, the barrier is opened. This will allow one of the waiting tasks to proceed by executing the body of `Allocate`. The effect of this execution is to close the barrier again, and hence no further executions of the entry body will be possible (until there is a further call of `Deallocate`).

Interestingly, the general resource controller (where groups of resources are requested and released) is not easy to program using just guards. The reasons for this will be explained in Chapter 8, where resource control is considered in some detail.

Each entry queue has an attribute associated with it that indicates how many tasks are currently queued. This is used in the following example. Assume that a task wishes to broadcast a value (of type `Message`) to a number of waiting tasks. The waiting tasks will call a `Receive` entry which is only open when a new message has arrived. At that time, all waiting tasks are released.

Although all tasks can now proceed, they must pass through the protected object in strict sequence (as only one can ever be active in the object). The last task out must then set the barrier to false again so that subsequent calls to `Receive` are blocked until a new message is broadcast. This explicit setting of the barriers can be compared with the use of condition variables which have no lasting effect (within the monitor) once all tasks have exited. The code for the broadcast example is as follows (note that the attribute `Count` indicates the number of tasks queued on an entry):

```
protected type Broadcast is
  entry Receive(M : out Message);
  procedure Send(M : Message);
private
  New_Message : Message;
  Message_Arrived : Boolean := False;
end Broadcast;

protected body Broadcast is

  entry Receive(M : out Message) when Message_Arrived is
  begin
    M := New_Message;
    if Receive'Count = 0 then
```

```
      Message_Arrived := False;
    end if;
  end Receive;

  procedure Send(M : Message) is
  begin
    if Receive'Count > 0 then
      Message_Arrived := True;
      New_Message := M;
    end if;
  end Send;

end Broadcast;
```

As there may be no tasks waiting for the message, the `send` procedure has to check the `Count` attribute. Only if it is greater than zero will it set the barrier to true (and record the new message).

Finally, this section gives a full Ada implementation of the semaphore package given in Section 5.4.5. This shows that protected objects are not only an excellent structuring abstraction but have the same expressive power as semaphores.

```
package Semaphore_Package is
  type Semaphore(Initial : Natural := 1) is limited private;
  procedure Wait (S : in out Semaphore);
  procedure Signal (S : in out Semaphore);
private
  protected type Semaphore(Initial : Natural := 1) is
    entry Wait_Imp;
    procedure Signal_Imp;
  private
    Value : Natural := Initial;
  end Semaphore;
end Semaphore_Package;

package body Semaphore_Package is
  protected body Semaphore is
    entry Wait_Imp when Value > 0 is
    begin
      Value := Value - 1;
    end Wait_Imp;

    procedure Signal_Imp is
    begin
      Value := Value + 1;
    end Signal_Imp;
  end Semaphore;

  procedure Wait(S : in out Semaphore) is
  begin
    S.Wait_Imp;
  end Wait;

  procedure Signal(S : in out Semaphore) is
  begin
```

```
    S.Signal_Imp;
  end Signal;
end Semaphore_Package;
```

5.8.2 Protected objects and object-oriented programming

As mentioned in Section 4.4.4 Ada 95 did not attempt to integrate the language's support for concurrent programming directly into the OOP model. Instead, the models were orthogonal and paradigms had to be created to allow the benefits of OOP to be available in a concurrent environment. These paradigms had inherent limitations and proposals were developed to add OOP facilities directly into, for example, the protected type mechanism. Unfortunately, these proposals added another layer of complexity to the language and they did not receive widespread support. The introduction of interfaces into Ada 2005 allows the concurrency facilities to provide some limited support of inheritance.

Interfaces in Ada 2005 are classified according to the type of object that can be supported ('implemented' using the Java terminology). For the purpose of this book the following interfaces are relevant.

- **Synchronized** – this specifies a collection of functions and procedures that can be implemented by a task type or a protected type.
- **Protected** – this specifies a collection of functions and procedures that can only be implemented by a protected type.
- **Task** – this specifies a collection of functions and procedures that can only be implemented by a task type.

Synchronized and protected interfaces will be considered in this section; discussion of task interfaces is deferred until Section 6.3.3.

The key idea of a synchronized interface is that there is some implied synchronization between the task that calls an operation from an interface and the object that implements the interface. Synchronization in Ada is achieved via two main mechanisms: a protected action (call of an entry or protected subprogram – a shared variable-based communication mechanism) or the rendezvous (a message-based communication mechanism – see Chapter 6). Hence, a task type or a protected type can implement a synchronized interface. Both protected entries and procedures can implement a synchronized interface procedure. A protected function can implement a synchronized interface function.

Where the programmer is not concerned with the form of synchronization, a synchronized interface is the appropriate abstraction. For situations where the programmer requires a particular form of synchronization, protected interfaces or task interfaces should be used explicitly. For example, there are various communication paradigms that all have at their heart some form of buffer. They, therefore, all have buffer-like operations in common. Some programs will use these paradigms and will not care whether the implementation uses a mailbox, a link or whatever. Some will require a task in the implementations, others will just need a protected object. Synchronized interfaces allow the programmer to defer the commitment to a particular paradigm and its implementation approach.

Consider the operations that can be performed on all integer buffers:

```
package Integer_Buffers is
  type Buffer is synchronized interface;
  procedure Put(Buf : in out Buffer; Item : in Integer)
            is abstract;
  procedure Get(Buf : in out Buffer; Item : out Integer)
            is abstract;
end Integer_Buffers;
```

In the above code, the `Buffer` type declaration indicates that it a synchronized interface, which supports the `Put` and `Get` procedures.

Now consider a protected type that can implement this interface.

```
with Integer_Buffers;
package Integer_Buffers.MailBoxes is
  subtype Capacity_Range is range ...;
  subtype Count is Integer range ...;
  type Buffer_Store is array(Capacity_Range) of Integer;

  protected type Mailbox is new Buffer with
    overriding entry Put(Item : in Integer);
    overriding entry Get(Item : out Integer);
  private
    First : Capacity_Range := Capacity_Range'first;
    Last : Capacity_Range := Capacity_Range'last;
    Number_In_Buffer : Count := 0;
    Box_Store : Buffer_Store;
  end Mailbox;
end Integer_Buffers.Mailboxes;
```

Here, the declaration of the `Mailbox` protected type indicates that it implements the `Buffer` interface by being derived from that type. The 'overriding' keyword on the entries indicates that these entries implement the interface's procedures. The name of the entry and its parameter type must match the interface's. Note, however, because of the way OOP is supported in Ada, the `Buf` parameter is not required as its type is the same as the type that the `Mailbox` is derived from. Hence, it is an implicit parameter that is generated when an instance of the mailbox is used. For example, in

```
with Integer_Buffers.Mailboxes;
use Integer_Buffers.Mailboxes;
...
Mail : Mailbox;
...
Mail.Put(42);
```

the `Mail` object provides the implicit first parameter.

The main limitations with the Ada approach is that you cannot derive one protected type from another.

5.9 Synchronized methods in Java

In many ways, Ada's protected objects are like objects in a class-based object-oriented programming language. The main difference, of course, is that they do not support a full inheritance relationship. Java, having a fully integrated concurrency and object-oriented model, provides a mechanism by which monitors can be implemented in the context of classes and objects.

In Java, there is a lock associated with each object. This lock cannot be accessed directly by the application, but it is affected by:

- the method modifier `synchronized`; and
- block synchronization.

When a method is labelled with the `synchronized` modifier, access to the method can only proceed once the lock associated with the object has been obtained. Hence synchronized methods have mutually exclusive access to the data encapsulated by the object, *if that data is only accessed by other synchronized methods*. Non-synchronized methods do not require the lock, and can therefore be called at any time. Hence to obtain full mutual exclusion, every method has to be labelled synchronized. A simple shared integer is therefore represented by:

```
class SharedInteger
{
  private int theData;

  public SharedInteger(int initialValue) {
    theData = initialValue;
  }

  public synchronized int read() {
    return theData;
  }

  public synchronized void write(int newValue) {
    theData = newValue;
  }

  public synchronized void incrementBy(int by) {
    theData = theData + by;
  }
}

SharedInteger myData = new SharedInteger(42);
```

Block synchronization provides a mechanism whereby a block can be labelled as synchronized. The `synchronized` keyword takes as a parameter an object whose lock it needs to obtain before it can continue. Hence synchronized methods are effectively implementable as:

```
public int read() {
  synchronized(this) {
```

```
    return theData;
  }
}
```

where `this` is the Java mechanism for obtaining the current object.

Used in its full generality, the synchronized block can undermine one of the advantages of monitor-like mechanisms: that of encapsulating synchronization constraints associated with an object into a single place in the program. This is because it is not possible to understand the synchronization associated with a particular object by just looking at the object itself when other objects can name that object in a synchronized statement. However, with careful use this facility augments the basic model and allows more expressive synchronization constraints to be programmed, as will be shown shortly.

Although synchronized methods or blocks allow mutually exclusive access to data in an object, this is not adequate if that data is *static*. Static data is shared between all objects created from the class. To obtain mutually exclusive access to this data requires access to a different lock.

In Java, classes themselves are also objects and therefore there is a lock associated with the class. This lock may be accessed either by labelling a static method with the synchronized modifier or by identifying the class's object in a synchronized block statement. The latter can be obtained from the `Object` class associated with the object. Note, however, that this class-wide lock is not obtained when synchronizing on the object. Hence to obtain mutual exclusion over a static variable requires the following (for example):

```
class StaticSharedVariable
{
  private static int shared;
  ...

  public synchronized static int Read() {
      return shared;
  }

  public synchronized static void Write(int I) {
    shared = I;
  }
}
```

5.9.1 Waiting and notifying

To obtain conditional synchronization requires further support. This again comes from methods provided in the predefined `Object` class as illustrated in Program 5.5. These methods are designed to be used only from within methods that hold the object lock (i.e. they are synchronized). If called without the lock, the exception `IllegalMonitorStateException` is thrown.

The `wait` method always blocks the calling thread and releases the lock associated with the object. If the call is made from within a nested monitor then only the lock associated with the `wait` is released.

Program 5.5 Support for waiting and notifying in the `Object` class.

```
package java.lang;
public class Object {
  ...
  // The following methods all throw the unchecked
  // IllegalMonitorStateException.
  public final void notify();
  public final void notifyAll();
  public final void wait() throws InterruptedException;
}
```

The `notify` method wakes up one waiting thread; the one woken is not defined by the Java language (however, it is defined by Real-Time Java; see Section 12.7). Note that `notify` does not release the lock, and hence the woken thread must still wait until it can obtain the lock before it can continue. To wake up *all* waiting threads requires use of the `notifyAll` method; again this does not release the lock and all the awoken threads must contend for the lock when it becomes free. If no thread is waiting, then `notify` and `notifyAll` have no effect.

A waiting thread can also be awoken if it is interrupted by another thread. In this case the `InterruptedException` is thrown. This situation will be ignored in this chapter (the exception will be allowed to propagate), but discussed fully in Section 7.7.2.

Although it appears that Java provides the equivalent facilities to other languages supporting monitors, there is one important difference. There are no explicit condition variables. Hence, when a thread is awoken, it cannot necessarily assume that its 'condition' is true, as all threads are potentially awoken irrespective of what conditions they were waiting on. For many algorithms this limitation is not a problem, as the conditions under which tasks are waiting are mutually exclusive.

For example, the bounded buffer traditionally has two condition variables: `BufferNotFull` and `BufferNotEmpty`, each associated with the corresponding buffer state. If a thread is waiting for one condition, no other thread can be waiting for the other condition as the buffer cannot be both full and empty at the same time. Hence, one would expect that the thread can assume that when it wakes, the buffer is in the appropriate state. Unfortunately, this is not always the case. Java, in common with other monitor-like approaches (for example, C/Real-Time POSIX mutexes), makes no guarantee that a thread woken from a wait will gain immediate access to the lock. Furthermore, a Java implementation is allowed to generate spurious wake-ups not related to the application.

Consider a thread that is woken after waiting on the `BufferNotFull` condition. Another thread could call the `put` method, find that the buffer has space and insert data into the buffer. When the woken thread eventually gains access to the lock, the buffer will again be full. Hence, it is usually essential for threads to re-evaluate their conditions, as illustrated in the integer bounded buffer example below.

```
public class BoundedBuffer {
  public BoundedBuffer(int length) {
```

```
    size = length;
    buffer = new int[size];
    last = 0;
    first = 0;
  }
  public synchronized void put(int item)
         throws InterruptedException {
    while (numberInBuffer == size) wait();
    last = (last + 1) % size;
    numberInBuffer++;
    buffer[last] = item;
    notifyAll();
  }
  public synchronized int get()
         throws InterruptedException {
    while (numberInBuffer == 0) wait();
    first = (first + 1) % size;
    numberInBuffer--;
    notifyAll();
    return buffer[first];
  }
  private int buffer[];
  private int first;
  private int last;
  private int numberInBuffer = 0;
  private int size;
}
```

Of course, if `notifyAll` is used to wake up threads, then it is more obvious that those threads must always re-evaluate their conditions before proceeding.

In general, many simple synchronization errors can be avoided in Java if all `wait` method calls are enclosed in while loops that evaluate the waiting conditions and the `notifyAll` method is used to signal changes to each object's state. This approach, while safe, is potentially inefficient as spurious wake-ups will occur. To improve performance, the `notify` method may be used when:

- all threads are waiting for the same condition;
- at most one waiting thread can benefit from the state change;
- the JVM does not generate any wake-ups without an associated call to the `notify` and `notifyAll` methods on the corresponding object.

The readers–writers problem

One of the standard concurrency control problems is the **readers–writers** problem. In this, many readers and many writers are attempting to access a large data structure. Readers can read concurrently, as they do not alter the data; however, writers require mutual exclusion over the data both from other writers and from readers. There are different variations on this scheme; the one considered here is where priority is always given to waiting writers. Hence, as soon as a writer is available, all new readers will be blocked until all writers have finished. Of course, in extreme situations this may lead to starvation of readers.

The solution to the readers–writers problem using standard monitors requires four monitor procedures – `startRead`, `stopRead`, `startWrite` and `stopWrite`. The readers are structured:

```
startRead();
  // read data structure
stopRead();
```

Similarly, the writers are structured:

```
startWrite();
  // write data structure
stopWrite();
```

The code inside the monitor provides the necessary synchronization using two condition variables: `OkToRead` and `OkToWrite`. In Java, the approach is as follows.

```
public class ReadersWriters {
  // Preference is given to waiting writers.
  public synchronized void startWrite()
         throws InterruptedException {
    // Wait until it is ok to write.
    while(readers > 0 || writing) {
      waitingWriters++;
      try {
        wait();
      } finally { waitingWriters--; }
    }
    writing = true;
  }
  public synchronized void stopWrite() {
    writing = false;
    notifyAll();
  }
  public synchronized void startRead()
         throws InterruptedException {
    // Wait until it is ok to read.
    while(writing || waitingWriters > 0) wait();
    readers++;
  }
  public synchronized void stopRead() {
    readers--;
    if(readers == 0) notifyAll();
  }
  private int readers = 0;
  private int waitingWriters = 0;
  private boolean writing = false;
}
```

In this solution, on awaking after the wait request, the thread must re-evaluate the conditions under which it can proceed. Although this approach will allow multiple readers or a single writer, arguably it is inefficient, as all threads are woken up every time the data becomes available. Many of these threads, when they finally gain access to the monitor, will find that they still cannot continue and therefore will have to wait again.

Program 5.6 An abridged Java `Lock` interface.

```
package java.util.concurrent.locks;
public interface Lock {
  public void lock();
    // Uninterruptibly wait for the lock to be acquired.
  public void lockInterruptibly()
          throws InterruptedException;
    // As above but interruptible.
  public Condition newCondition();
    // Create a new condition variable for use with the Lock.
  public boolean tryLock();
    // Returns true is lock is available immediately.
  public void unlock();
  ...
}
```

Program 5.7 An abridged Java `Condition` interface.

```
package java.util.concurrent.locks;
public interface Condition {
  public void await() throws InterruptedException;
  /* Atomically releases the associated lock and
   * causes the current thread to wait until
   *  1. another thread invokes the signal method
   *     and the current thread happens to be chosen
   *     as the thread to be awakened; or
   *  2. another thread invokes the signalAll method;
   *  3. another thread interrupts the thread; or
   *  4. a spurious wake-up occurs.
   * When the method returns it is guaranteed to hold the
   * associated lock.
   */

  public void awaitUninterruptible();
   // As for await, but not interruptible.
  public void signal();
   // Wake up one waiting thread.
  public void signalAll();
   // Wake up all waiting threads.
}
```

5.9.2 Synchronizers and locks

The Java concurrency utilities provide a range of support packages aimed at easing the burden of concurrent programming. These expand the built-in facilities that have been discussed above. The package that is most relevant to this section is `java.util.concurrent.locks`. It main goal is to provide efficient support for

Program 5.8 An abridged Java `ReentrantLock` class.

```
package java.util.concurrent.locks;

public class ReentrantLock implements Lock, java.io.Serializable {
  public ReentrantLock();

    ...
  public void lock();
  public void lockInterruptibly() throws InterruptedException;
  public Condition newCondition();
    // Create a new condition variable and associated it
    // with this lock object.
  public boolean tryLock();
  public void unlock();
}
```

various locking approaches including locks that support explicit condition variables. Programs 5.6 and 5.7 show the Java interfaces that support these abstractions. Various types of lock are provided, including mutual exclusion locks (the `ReentrantLock` class shown in Program 5.8) and read/write locks.

Using these facilities it is possible to implement the bounded buffer using the familiar algorithm with two condition variables.

```
import java.util.concurrent.locks.*;
public class BoundedBuffer2 {
  public BoundedBuffer2(int length) {
    size = length;
    buffer = new int[size];
    last = 0;
    first = 0;
    numberInBuffer = 0;
    lock = new ReentrantLock();
    notFull = lock.newCondition();
    notEmpty = lock.newCondition();
  }
  public void put(int item)
         throws InterruptedException {
   lock.lock();
   try {
     while (numberInBuffer == size) notFull.await();
     last = (last + 1) % size;
     numberInBuffer++;
     buffer[last] = item;
     notEmpty.signal();
   } finally {
       lock.unlock();
   }
  }
```

```
public synchronized int get()
       throws InterruptedException {
 lock.lock();
 try {
   while (numberInBuffer == 0) notEmpty.await();
   first = (first + 1) % size ;
   numberInBuffer--;
   notFull.signal();
   return buffer[first];
 } finally {
     lock.unlock();
 }
}
private int buffer[];
private int first;
private int last;
private int numberInBuffer;
private int size;
private Lock lock;
private final Condition notFull;
private final Condition notEmpty;
}
```

Note the use of a finally clause to ensure the `unlock` method is always called before the method exits.

5.9.3 Inheritance and synchronization

The combination of the object-oriented paradigm with mechanisms for concurrent programming may give rise to the so-called **inheritance anomaly** (Matsuoka and Yonezawa, 1993). An inheritance anomaly exists if the synchronization between operations of a class is not local but may depend on the whole set of operations present for the class. When a subclass adds new operations, it may become necessary to change the synchronization defined in the parent class to account for these new operations.

For example, consider the bounded buffer presented earlier in this section. With Java, the code that tests the conditions (`BufferNotFull` and `BufferNotEmpty`) is embedded in the methods. It will be shown in Chapter 8 that, in general, this approach has many advantages as it allows the methods to access the parameters to the method as well as the object attributes. However, it does cause some problems when inheritance is considered. Suppose that the bounded buffer is to be subclassed so that all accesses can be prohibited. Two new methods are added, `prohibitAccess` and `allowAccess`. A naive extension might include the following code:

```
public class LockableBoundedBuffer extends BoundedBuffer {
  boolean prohibited ;

  // Incorrect Code

  LockableBoundedBuffer(int length) {
    super(length);
    prohibited = false;
  }
```

```
  public synchronized void prohibitAccess() throws InterruptedException {
    while (prohibited) wait();
    prohibited = true;
  }

  public synchronized void allowAccess() throws AccessError {
    if (!prohibited) throw new AccessError();
    prohibited = false;
    notifyAll();
  }

  public synchronized void put(int item) throws InterruptedException {
    while(prohibited) wait();
    super.put(item);
  }

  public synchronized int get() throws InterruptedException {
    while(prohibited) wait();
    return(super.get());
  }
}
```

Unfortunately, there is a subtle bug with this approach. Consider the case where a producer is attempting to put data into a full buffer. Access to the buffer is not prohibited so `super.put(item)` is invoked where the call is blocked waiting for the `BufferNotEmpty` condition. Now access to the buffer is prohibited. Any further calls to `get` and `put` are held in the overridden subclass methods, as will be further calls to the `prohibitAccess` method. Now a call to the `allowAccess` method is made; this results in all waiting threads being released. Suppose the order in which the released threads acquire the monitor lock is: the consumer thread, the thread attempting to prohibit access to the buffer, the producer thread. The consumer finds that access to the buffer is not prohibited and takes data from the buffer. It issues a `notifyAll` request, but no threads are now currently waiting. The next thread which runs now prohibits access to the buffer. The producer thread runs next and places an item into the buffer although access is prohibited!

Although this example might seem contrived, it does illustrate the subtle bugs that can occur due to the inheritance anomaly.

5.10 Shared memory multiprocessors

Multiprocessor systems are becoming more prevalent. In particular symmetric multiprocessor (SMP) systems, where processors have shared access to the main memory, are often the default platform for large real-time systems rather than a single processor system. From a theoretical concurrent programming viewpoint, a program that is properly synchronized and executes successfully on a single processor systems will execute successfully on a SMP system. Programs that are not properly synchronized may suffer from data race conditions (see Section 5.3). Even properly synchronized programs can suffer from deadlocks. Consequently, a program that *appears* to execute correctly on a single processor cannot be guaranteed to work correctly on a multiprocessor as it is doubtful that all possible interleaving of task executions will have been exercised on

the single processor system. As a result, concurrency-related faults/bugs will remain dormant.

In order to understand how a concurrent program executes on a SMP system it is necessary to understand the memory consistency model provided by the machine. The simplest model is *sequential consistency*. An SMP system is sequentially consistent if the result of any execution of a program is the same as if the instructions of all the processors are executed in some sequential order, and the instructions of any thread within the sequence is the same as that specified by its program logic (Lamport, 1997). Hence, both atomicity of instructions and the maintenance of task instruction sequences are required.

Sequential consistency is a very restrictive property, and if rigidly supported would disallow many optimizations that are typically performed by compilers and modern multiprocessors. For example, a compiler would not be able to reorder the instructions and the hardware would not be able to execute instructions out of order. The presence of hardware caches exacerbates these problems.

To overcome the severe constraints imposed by the requirement of sequential consistency, **relaxed memory models** can be used. These either relax the instruction order or the atomicity requirements. Different SMP architectures adopt different approaches – see Adve and Gharachorloo (1996) for a classification.

From the programmer's perspective, it is crucial to understand what guarantees a programming language provides when a shared variable is updated, in particular, when that update becomes visible to other tasks potentially executing on other processors. The remainder of this section considers the guarantees provided by Java and Ada. It is the compiler and the run-time systems that must implement these guarantees on the underlying architecture's memory model irrespective of the memory model it provides.

5.10.1 The Java memory model

Early versions of the Java language were criticized because its semantics on multiprocessors were ill defined and had serious problems (Pugh, 2000). The Java 5 language has corrected this with a new memory model (the Java Memory Model, JMM) that, on the one hand allows both compiler and hardware optimization, but, on the other hand, gives intuitive semantics to program code.

From a language semantics view point, a concurrent program can be defined using a trace model. A trace model defines the meaning of a thread as the set of sequences of events (traces) that the thread can be observed to perform. Hence, a program's execution is the set of all possible thread traces. The language's **memory model** describes, given a program and an execution trace of that program, whether the execution trace is a legal execution of the program.

The JMM is concerned with **memory actions**, which are defined to be reads and writes to memory locations shared between threads. Local variables, formal method parameters and exception handler parameters are never shared, and consequently fall outside the model. Given two actions A and B, the results of action A is visible to action B if there is a **happens-before** relation between them. The following defines the relation:

- if A and B are in the same thread and A comes before B in the program order then A happens-before B;

- an unlock monitor action happens-before all subsequent lock actions on the same monitor;
- a write to a volatile[1] variable *V* happens-before all subsequent reads from *V* in any thread;
- an action that starts a thread happens-before the first action of the thread it starts;
- the final action of a thread happens-before any action in any other thread that determines that it has terminated (using `isAlive` or the `join` methods in the `Thread` class);
- the interruption of a thread *T* (via the `interrupt` method in the `Thread` class) happens-before any thread that detects that *T* has been interrupted (via the `interrupted` and `isInterrupt` methods in the `Thread` class or by having the `InterruptedException` thrown);
- if *A* happens-before *B* and *B* happens-before *C*, then *A* happens-before *C*.

The formal semantics of the JMM are complex; however, they can be approximated by the following two rules.

- The actions of each thread in isolation are defined by the action of its code (in program order) in isolation, with the exception that the values seen by each read variable action are determined by the JMM.
- A read operation on variable A must return the value written to it by the previous write operation that *happened before* it.

From this, the following points need to be emphasized when accessing variables shared between threads.

- When one thread starts another – changes made by the parent thread before the start requests are visible to the child thread when it executes.
- When one thread waits for the termination of another – changes made by the terminating thread before it terminates are visible to the waiting thread once termination has been detected.
- When one thread interrupts another – changes made by the interrupting thread before the interrupt request are made visible to the interrupted thread when the interruption is detected by the interrupted thread.
- When threads read and write to the same volatile field – changes made by the writer thread to shared data (before it writes to the volatile field) are made visible to a subsequent reader of the same volatile field.

Any implementation of Java on a SMP system must respect the Java Memory Model, and where the architecture potentially performs optimization that might undermine it, code must be executed to ensure that this does not occur (for example memory fences or barriers must be inserted).

[1] A **volatile** variable is one that cannot be held in local registers or caches. All read and write operations go directly to the memory.

5.10.2 Ada and shared variables

Ada has no explicit memory model but, like Java, the Ada language defines the conditions under which it is safe to read and write to shared variables outside the rendezvous or protected objects. Hence, the model is implicit.

The safe conditions are as follows:

- where one task writes a variable before activating another task that reads the variable;
- where the activation of one task writes the variable and the task awaiting completion of the activation reads the variable;
- where one task writes the variable and another task waits for the termination of the task and then reads the variable;
- where one task writes the variable before making an entry call on another task, and the other task reads the variable during the corresponding entry body or accept statement;
- where one task writes a shared variable during an accept statement and the calling task reads the variable after the corresponding entry call has returned;
- where one task writes a variable whilst executing a protected procedure body or entry, and the other task reads the variable, for example as part of a later execution of an entry body of the same protected body.

If the Systems Programming Annex is supported, there are extra facilities that can be used to control shared variables between *unsynchronized* tasks. They come in the form of extra pragmas which can be applied to certain objects or type declarations.

- Pragma `Volatile` – pragma `Volatile` ensures that all reads and writes go directly to memory.
- Pragma `Volatile_Components` – pragma `Volatile_Components` applies to components of an array.
- Pragma `Atomic` and pragma `Atomic_Components` – whilst pragma `Volatile` indicates that all reads and writes must be directed straight to memory, pragma `Atomic` imposes the further restriction that they must be indivisible. That is, if two tasks attempt to read and write the shared variable at the same time, then the result must be internally consistent. An implementation is not required to support atomic operations for all types of variable; however, if not supported for a particular object, the pragma (and hence the program) must be rejected by the compiler.

The language defines accesses to volatile and atomic variables to be interactions with the external environment, and hence compilers must ensure that no reordering of instructions occurs across their use.

Unlike Java, which attempts to define the semantics of a program that is not properly synchronized, Ada simply defines these situations to result in erroneous program execution.

5.11 Simple embedded system revisited

In Section 4.8, a simple embedded system was introduced and a concurrent solution was proposed. The Ada solution is now updated to illustrate communication with the operator console. Recall that the structure of the controller is as below:

```
with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;

procedure Controller is
  task Temp_Controller;
  task Pressure_Controller;

  task body Temp_Controller is
    TR : Temp_Reading; HS : Heater_Setting;
  begin
    loop
      Read(TR);
      Temp_Convert(TR,HS);
      Write(HS);
      Write(TR);
    end loop;
  end Temp_Controller;

  task body Pressure_Controller is
    PR : Pressure_Reading; PS : Pressure_Setting;
  begin
    loop
      Read(PR);
      Pressure_Convert(PR,PS);
      Write(PS);
      Write(PR);
    end loop;
  end Pressure_Controller;

begin
  null;    -- Temp_Controller and Pressure_Controller
           -- have started their executions
end Controller;
```

and that the interfaces to the I/O routines were:

```
with Data_Types; use Data_Types;
package IO is
    -- procedures for data exchange with the environment
  procedure Read(TR : out Temp_Reading); -- from DAC
  procedure Read(PR : out Pressure_Reading); -- from DAC
  procedure Write(HS : Heater_Setting);    -- to switch.
  procedure Write(PS : Pressure_Setting); -- to DAC
  procedure Write(TR : Temp_Reading);      -- to console
  procedure Write(PR : Pressure_Reading); -- to console
end IO;
```

The body of the I/O routines can now be completed. The data to be sent to the console is stored in a monitor (in this case using an Ada protected object). The console task will call the entry to get the new data.

```
package body IO is
  task Console;
  protected Console_Data is
    procedure Write(R : Temp_Reading);
    procedure Write(R : Pressure_Reading);
    entry Read(TR : out Temp_Reading;
               PR : out Pressure_Reading);
  private
    Last_Temperature : Temp_Reading;
    Last_Pressure :Pressure_Reading;
    New_Reading : Boolean := False;
  end Console_Data;

  -- procedures for data exchange with the environment
  procedure Read(TR : out Temp_Reading) is separate; -- from DAC
  procedure Read(PR : out Pressure_Reading) is separate; -- from DAC
  procedure Write(HS : Heater_Setting) is separate;   -- to switch.
  procedure Write(PS : Pressure_Setting) is separate; -- to DAC

  task body Console is
    TR : Temp_Reading;
    PR : Pressure_Reading;
  begin
    loop
       ...
      Console_Data.Read(TR, PR);
      -- Display new readings
     end loop;
  end Console;

  protected body Console_Data is
    procedure Write(R : Temp_Reading) is
    begin
      Last_Temperature := R;
      New_Reading := True;
    end Write;

    procedure Write(R : Pressure_Reading) is
    begin
      Last_Pressure := R;
      New_Reading := True;
    end Write;

    entry Read(TR : out Temp_Reading;
               PR : out Pressure_Reading)
          when New_Reading is
    begin
      TR := Last_Temperature;
      PR := Last_Pressure;
      New_Reading := False;
    end Read;
  end Console_Data;
```

```
  procedure Write(TR : Temp_Reading) is
  begin
    Console_Data.Write(TR);
  end Write;      -- to screen

  procedure Write(PR : Pressure_Reading)  is
  begin
    Console_Data.Write(PR);
  end Write; -- to screen
end IO;
```

Summary

Process interactions require operating systems and concurrent programming languages to support synchronization and inter-task communication. Communication can be based on either shared variables or message passing. This chapter has been concerned with shared variables, the multiple update difficulties they present and the mutual exclusion synchronizations needed to counter these difficulties. In this discussion, the following terms were introduced:

- **critical section** – code that must be executed under mutual exclusion;
- **producer – consumer system** – two or more tasks exchanging data via a finite buffer;
- **busy waiting** – a task continually checking a condition to see if it is now able to proceed;
- **livelock** – an error condition in which one or more tasks are prohibited from progressing whilst using up processing cycles.

Examples were used to show how difficult it is to program mutual exclusion using only shared variables. Semaphores were introduced to simplify these algorithms and to remove busy waiting. A semaphore is a non-negative integer that can only be acted upon by `wait` and `signal` procedures. The executions of these procedures are atomic.

The provision of a semaphore primitive has the consequence of introducing a new state for a task; namely, **suspended**. It also introduces two new error conditions:

- **deadlock** – a collection of suspended tasks that cannot proceed;
- **indefinite postponement** – a task being unable to proceed as resources are not made available for it (also called lockout or starvation).

Semaphores can be criticized as being too low-level and error-prone in use. Following their development, five more structured primitives were introduced:

- conditional critical regions
- monitors
- mutexes

- protected objects
- synchronized methods.

Monitors are an important language feature. They consist of a module, entry to which is assured (by definition) to be under mutual exclusion. Within the body of a monitor, a task can suspend itself if the conditions are not appropriate for it to proceed. This suspension is achieved using a condition variable. When a suspended task is awoken (by a `signal` operation on the condition variable), it is imperative that this does not result in two tasks being active in the module at the same time.

A form of monitor can be implemented using a procedural interface. Such a facility is provided by mutexes and condition variables in C/Real-Time POSIX.

Although monitors provide a high-level structure for mutual exclusion, other synchronizations must be programmed using very low-level condition variables. This gives an unfortunate mix of primitives in the language design. Ada's protected objects give the structuring advantages of monitors and the high-level synchronization mechanisms of conditional critical regions.

Integrating concurrency and OOP is fraught with difficulties. Ada tries to simplify the problem by supporting interfaces but not inheritance with protected types. Java, however, addresses the problem by providing synchronized member methods for classes. This facility (along with the synchronized statement and wait and notify primitives) provides a flexible object-oriented based monitor-like facility. Unfortunately, the inheritance anomaly is present with this approach.

The next chapter considers message-based synchronization and communication primitives. Languages that use these have, in effect, elevated the monitor to an active task in its own right. As a task can only be doing one thing at a time, mutual exclusion is assured. Tasks no longer communicate with shared variables but directly. It is therefore possible to construct a single high-level primitive that combines communication and synchronization. This concept was first considered by Conway (1963) and has been employed in high-level real-time programming languages. It forms the basis of the rendezvous in Ada.

Further reading

Ben-Ari, M. (2005) *Principles of Concurrent and Distributed Programming*. New York: Prentice Hall.

Burns, A. and Wellings, A. J. (2007) *Concurrent and Real-time Programming in Ada*. Cambridge: Cambridge University Press.

Butenhof, D. R. (1997) *Programming With Posix Threads*. Reading, MA: Addison-Wesley.

Goetz, B. (2006) *Java: Concurrency in Practice*. Reading, MA: Addison-Wesley.

Hyde, P. (1999) *Java Thread Programming*. Indianapolis, IN: Sams Publishing.

Lea, D. (1999) *Concurrent Programming in Java: Design Principles and Patterns*. Reading, MA: Addison-Wesley.

Oaks, A. and Wong, H. (1997) *Java Threads*. Sebastopol, CA: O'Reilly.

Nichols, B., Buttlar, D. and Farrell, J. (1996) *POSIX Threads Programming*. Sebastopol, CA: O'Reilly.

Wellings, A.J. (2004) *Concurrent and Real-Time Programming in Java*. Chichester: Wiley.

Exercises

5.1 Show how Peterson's algorithm given in Section 5.2 can be modified to allow a high-priority task to be given preference over a low-priority one when busy waiting.

5.2 Consider a data item which is to be shared between a single producer and a single consumer. The producer is a periodic task which reads a sensor and writes the value to the shared data item. The consumer is a sporadic task which wishes to take the latest value placed in the shared data item by the producer.

The following package claims to provide a generic algorithm with which the producer and consumer can communicate safely, *without* the need for mutual exclusion or busy waiting.

```
generic
  type Data is private;
  Initialvalue : Data;
package Simpsons_Algorithm is
  procedure Write(Item: Data); -- non-blocking
  procedure Read (Item : out Data); -- non-blocking
end Simpsons_Algorithm;

package body Simpsons_Algorithm is

  type Slot is (First, Second);

  Fourslot : array (Slot, Slot) of Data :=
             (First => (Initialvalue,Initialvalue),
              Second => (Initialvalue,Initialvalue));

  Nextslot : array(Slot) of Slot := (First, First);

  Latest : Slot := First;
  Reading : Slot := First;

  procedure Write(Item : Data) is
    Pair, Index : Slot;
  begin
    if Reading = First then
      Pair := Second;
    else
      Pair := First;
    end if;
    if NEXT_SLOT(Pair) = First then
       Index := Second;
    else
```

```
        Index := First;
      end if;
      Fourslot(Pair, Index) := Item;
      Nextslot(Pair) := Index;
      Latest := Pair;
    end Write;

    procedure Read(Item : out Data) is
      Pair, Index : Slot;
    begin
      Pair := Latest;
      Reading := Pair;
      Index := Nextslot(Pair);
      Item := Fourslot(Pair, Index);
    end Read;
  end Simpsons_Algorithm;
```

Describe carefully how this algorithm works by explaining what happens when:

(1) a `Write` is followed by a `Read`
(2) a `Read` is preempted by a `Write`
(3) a `Write` is preempted by a `Read`
(4) a `Read` is preempted by more than one `Write`
(6) a `Write` is preempted by more than one `Read`

Comment on how the algorithm trades off safe communication against data freshness.
Many compilers optimize code so that frequently accessed variables are held in registers local to a task. In this context, comment on whether the above algorithm's claim of safe communication is justifiable. Are there any changes to the algorithm needed to make it work for all implementations of Ada?

5.3 Consider a shared data structure that can be both read from and written to. Show how semaphores can be used to allow many concurrent readers or a single writer, but not both.

5.4 Show how Conditional Critical Region can be implemented using semaphores.

5.5 Show how Hoare's monitors can be implemented using semaphores.

5.6 Show how binary semaphores can be implemented using Hoare's monitors.

5.7 One of the criticisms of monitors is that condition synchronization is too low-level and unstructured. Explain what is meant by this statement. A higher level monitor synchronization primitive might take the form

```
WaitUntil boolean_expression;
```

where the task is delayed until the boolean expression evaluates to true. For example:

```
WaitUntil x < y + 5;
```

would delay the task until `x < y + 5`.

Although this form of condition synchronization is more structured it is not found in most languages which support monitors. Explain why this in the case. Under what circumstances would the objections to the above high-level synchronization facility be invalid? Show how the bounded buffer problem can be solved using the `WaitUntil` synchronization primitive inside a monitor.

5.8 Consider a system of three cigarette smoker tasks and one agent task. Each smoker continuously makes a cigarette and smokes it. To make a cigarette three ingredients are needed: tobacco, paper and matches. One of the tasks has paper, another tobacco and the third has matches. The agent has an infinite supply of all three ingredients. The agent places two ingredients, chosen randomly, on a table. The smoker who has the remaining ingredient can then make and smoke a cigarette. Once it has finished smoking the smoker signals the agent who then puts another two of the three ingredients on the table, and the cycle repeats.
Sketch the structure of a monitor which will synchronize the agent and all three smokers.

5.9 Show how the operations on a semaphore can be implemented in the nucleus of an operating system for a single processor system **without** busy waiting. What hardware facility does your solution require?

5.10 Show how C/Real-Time POSIX mutexes and condition variables can be used to implement a shared data structure that can be both read from and written to. Allow many concurrent readers or a single writer but not both.

5.11 Show how C/Real-Time POSIX mutexes and condition variables can be used to implement a resource controller.

5.12 Complete Exercise 4.8 using Ada's protected objects for task synchronization.

5.13 Compare and contrast the facilities provided by the C/Real-Time POSIX mutexes and condition variables with those provided by Ada's protected objects.

5.14 Redo Exercise 5.8 using protected objects.

5.15 Implement quantity semaphores using protected objects.

5.16 Show how one or more protected objects can be used to implement Hoare's monitors.

5.17 Explain the synchronization imposed by the following Ada protected object:

```
protected type Barrier(Needed : Positive) is
  entry Wait;
private
  Releasing : Boolean := False;
end Barrier;

protected body Barrier is
  entry Wait when Wait'Count = Needed or Releasing is
  begin
    if Wait'Count = 0 then
      Releasing = False;
    else
      Releasing := True;
    end if;
```

```
  end Wait;
end Barrier;
```

The following package provides a simplified Ada binding to Real-Time POSIX Mutexes and Condition Variables. All Mutexes and Condition Variables are initialized with default attributes.

```
package Pthreads is
  type Mutex_T is limited private;
  type Cond_T is limited private;

  procedure Mutex_Initialize (M: in out Mutex_T);
  procedure Mutex_Lock(M: in out Mutex_T);
  procedure Mutex_Trylock(M: in out Mutex_T);
  procedure Mutex_Unlock(M: in out Mutex_T);

  procedure Cond_Initialize(C: in out Cond_T);
  procedure Cond_Wait(C: in out Cond_T;
                      M : in out Mutex_T);

  procedure Cond_Signal(C: in out Cond_T);
  procedure Cond_Broadcast(C: in out Cond_T);
private
   ...
end Pthreads;
```

Show how Barriers, as defined above, can be implemented using this package. Do *not* use any of Ada's communication and synchronization facilities in the solution.

5.18 The following package defines an Ada semaphore abstraction:

```
generic
  Initial : Natural:= 1; -- default initial value of semaphore
package Semaphore_Package is
  type Semaphore is limited private;
  procedure Wait (S : Semaphore);
  procedure Signal (S : Semaphore);
private
  type Semaphore is ...; -- not needed for this question
end Semaphore_Package;
```

Using the `Semaphore_Package`, show how the following communication paradigm (and its associated package specification) can be implemented.
A Multicast is where one task is able to send the same data to several waiting tasks. A package specification for the `Multicast` abstraction is given below:

```
package Multicast is
  procedure Send(I : Integer);
  procedure Receive(I : out Integer);
end Multicast;
```

The receiver tasks indicate their willingness to receive data by calling the procedure `Receive` defined in the package given above (in this example the data is of type `Integer`). The tasks are blocked by this call. A sender task indicates that it wishes to multicast data by calling the `Send` procedure. All tasks which are currently blocked on the receive are released when the sender invokes the `Send` procedure. The data passed by the sender is given to all waiting tasks. Once the `Send` procedure has finished, any new calls to `Receive` must wait for the next sender.
Show how the body of the `Multicast` package can be implemented using semaphores.

5.19 A broadcast is similar to a multicast except that ALL intended recipients must receive the data. A package specification for the `Broadcast` abstraction is given below:

```
package Broadcast is
    -- for 10 tasks
  procedure Send(I : Integer);
  procedure Receive(I : out Integer);
end Broadcast;
```

Consider, for example, a system with 10 recipient tasks. These tasks all call the `Receive` procedure when they are ready to receive the broadcast. The tasks are blocked by this call. A sender task indicates that it wishes to broadcast data by calling the `Send` procedure. The `Send` procedure waits until *all* ten tasks are ready to receive the broadcast before releasing the recipients and passing the data. If more than one call to `Send` occurs, then they are queued.
Show how the body of the `Broadcast` package can be implemented using the semaphore package given in Exercise 5.18.

5.20 It has been suggested that York should put a limit on the number of motorists that can enter the city at any one time. One proposal is to establish monitoring checkpoints at each of the city's Bars (entrances through the city's walls) and, when the city is full, to turn the traffic lights to red for incoming traffic. To indicate to the motorists that the city is full, the red light is set to flashing.
In order to achieve this goal, pressure sensors are placed in the road at the Bars' entry and exit points. Every time a car enters the city, a signal is set to a `BarController` task (as a task entry call); similarly when a car exits:

```
Max_Cars_In_City_For_Red_Light : constant Positive := N;
Min_Cars_In_City_For_Green_Light : constant Positive := N - 10;

type Bar is (Walmgate, Goodramgate, Micklegate,
             Bootham, Barbican);

task type Bar_Controller(G : Bar) is
  entry Car_Entered;
  entry Car_Exited;
end Bar_Controller;
```

```
Walmgate_Bar_Controller : Bar_Controller(Walmgate);
Goodramgate_Bar_Controller : Bar_Controller(Goodramgate);
Micklegate_Bar_Controller : Bar_Controller(Micklegate);
Bootham_Bar_Controller : Bar_Controller(Bootham);
Barbican_Bar_Controller : Bar_Controller(Barbican);
```

Show how the body of these tasks can be coordinated so that ONE of them calls the `CityTrafficLightController` (with the following task specification) to indicate whether more cars are to be allowed in or not.

```
task City_Traffic_Lights_Controller is
  entry City_Is_Full;
  entry City_Has_Space;
end City_Traffic_Lights_Controller;

task body Traffic_Lights_Controller is separate;
--  body of no interest in this question
```

5.21 Explain why the resource controller given in Section 5.4.7 suffers from a race condition. How can the algorithm be modified to remove this condition?

5.22 Show how the reader/writers problem can be implemented in Java where priority is given to readers and where writers are guaranteed to be serviced in a FIFO order.

5.23 Show how Java can be used to implement a resource controller.

5.24 Implement quantity semaphores using Java.

5.25 Consider the following Java class:

```
public class Event {
  public synchronized void highPriorityWait();
  public synchronized void lowPriorityWait();
  public synchronized
  void signalEvent();
}
```

Show how this class be implemented so that `signalEvent` releases one high-priority waiting thread if one is waiting. If there is no high-priority waiting thread, release one low-priority waiting thread. If no thread is waiting, the `signalEvent` has no effect.

Now consider the case where an Id can be associated with the methods. How can the algorithm be modified so that the `signalEvent` wakes up the appropriate blocked thread?

Chapter 7
Atomic actions, concurrent tasks and reliability

Chapter 2 considered how reliable software could be produced in the presence of a variety of errors. Modular decomposition and atomic actions were identified as two techniques essential for damage confinement and assessment. Also, the notions of forward and backward error recovery were introduced as approaches to dynamic error recovery. It was shown that where tasks communicate and synchronize their activities, backward error recovery may lead to the domino effect. In Chapter 3, exception handling was discussed as a mechanism for providing both forward and backward error recovery in sequential tasks. Chapters 4, 5 and 6 then considered the facilities provided by operating systems and real-time languages for concurrent programming. This chapter brings together exception handling and concurrency in order to show how tasks can interact reliably in the presence of other tasks and in the presence of faults. The notion of an atomic action is explored in more detail and the concept of asynchronous notification is introduced.

In Chapter 4, the interaction of tasks was described in terms of three types of behaviour:

- independent
- cooperation
- competition.

Independent tasks do not communicate or synchronize with each other. Consequently, if an error occurs within one task, then recovery procedures can be initiated by that task in isolation from the rest of the system. Recovery blocks and exception handling can be used as described in Chapters 2 and 3.

Cooperating tasks, by comparison, regularly communicate and synchronize their activities in order to perform some common operation. If any error condition occurs, it is necessary for all tasks involved to perform error recovery. The programming of such error recovery is the topic of this chapter.

Competing tasks communicate and synchronize in order to obtain resources; they are, however, essentially, independent. An error in one should have no effect on the others. Unfortunately, this is not always the case, particularly if the error occurred while a task was in the act of being allocated a resource. Reliable resource allocation is considered in Chapter 8.

Where cooperating tasks communicate and synchronize through shared resources, recovery may involve the resource itself. This aspect of resource allocation will also be considered in Chapter 8.

7.1 Atomic actions

One of the main motivations for introducing concurrent tasks into a language is that they enable parallelism in the real world to be reflected in application programs. This enables such programs to be expressed in a more natural way and leads to the production of more reliable and maintainable systems. Disappointingly, however, concurrent tasks create many new problems which did not exist in the purely sequential program. Consequently, the last few chapters have been dedicated to discussing some of the solutions to these problems: in particular, communication and synchronization between tasks using shared variables (correctly) and message passing. This was undertaken in a fairly isolated manner and no consideration has yet been given to the way in which groups of concurrent tasks should be structured in order to coordinate their activities.

The interaction between two tasks has, so far, been expressed in terms of a single communication. In reality, this is not always the case. For example, withdrawal from a bank account may involve a ledger task and a payment task in a sequence of communications to authenticate the drawer, check the balance and pay the money. Furthermore, it may be necessary for more than two tasks to interact in this way to perform the required action. In all such situations, it is imperative that the tasks involved see a consistent system state. With concurrent tasks, it is all too easy for groups of tasks to interfere with one other.

What is required is for each group of tasks to execute their joint activity as an **indivisible** or **atomic action**. Of course, a single task may also want to protect itself from the interference of other tasks (for example, during resource allocation). It follows that an atomic action may involve one or more tasks. Atomic actions have also been called *multiparty* interactions (Evangelist et al., 1989; Yuh-Jzer and Smolka, 1996).

There are several almost equivalent ways of expressing the properties of an atomic action (Lomet, 1977; Randell et al., 1978).

(1) An action is atomic if the tasks performing it are not aware of the existence of any other active task, and no other active task is aware of the activity of the tasks during the time the tasks are performing the action.

(2) An action is atomic if the tasks performing it do not communicate with other tasks while the action is being performed.

(3) An action is atomic if the tasks performing it can detect no state change except those performed by themselves and if they do not reveal their state changes until the action is complete.

(4) Actions are atomic if they can be considered, so far as other tasks are concerned, to be indivisible and instantaneous, such that the effects on the system are as if they were interleaved as opposed to concurrent.

These are not quite all equivalent. For example, consider the second expression: an action is atomic if the tasks performing it communicate only among themselves and not with other tasks in the system. Unlike the other three, this does not really define the true nature of an atomic action. While it will guarantee that the action is indivisible, it is too strong a constraint on the tasks. Interactions between an atomic action and the rest of the system can be allowed as long as they have no impact on the activity of the atomic action and do not provide the rest of the system with any information concerning the progress of the action (Anderson and Lee, 1990). In general, in order to allow such interactions requires detailed knowledge of the atomic action's function and its interface to the rest of the system. As this cannot be supported by a general language implementation, it is tempting, following Anderson and Lee (1990), to adopt the more restrictive (second) definition. This can only be done, however, if the resources necessary to complete an atomic action are acquired by the underlying implementation, not by instructions given in the program. If resources are to be acquired and released when the programmer desires, tasks within atomic actions will have to communicate with general-purpose resource managers.

Although an atomic action is viewed as being indivisible, it can have an internal structure. To allow modular decomposition of atomic actions, the notion of a **nested atomic action** is introduced. The tasks involved in a nested action must be a subset of those involved in the outer level of the action. If this were not the case, a nested action could smuggle information concerning the outer level action to an external task. The outer level action would then no longer be indivisible.

7.1.1 Two-phase atomic actions

Ideally, all tasks involved in an atomic action should obtain the resources they require (for the duration of the action) prior to its commencement. These resources could then be released after the atomic action had terminated. If these rules were followed, there would be no need for an atomic action to interact with any external entity and the stricter definition of atomic action could be adopted.

Unfortunately, this ideal can lead to poor resource utilization, and hence a more pragmatic approach is needed. Firstly, it is necessary to allow an atomic action to start without its full complement of resources. At some point, a task within the action will request a resource allocation; the atomic action must then communicate with the resource manager. This manager may be a server task. If a strict definition of atomic action is adhered to, this server would have to form part of the atomic action, with the effect of serializing all actions involving the server. Clearly, this is undesirable, and hence an atomic action is allowed to communicate externally with resource servers.

Within this context, a resource server is defined to be a custodian of non-sharable system utilities. It protects these utilities against inappropriate access, but does not, itself, perform any actions upon them.

A further improvement in resource allocation can be made if a task is allowed to release a resource prior to completion of the associated atomic action. In order for this premature release to make sense, the state of the resource must be identical to that which would appertain if the resource was retained until completion of the atomic action. Its early release will, however, enhance the concurrency of the whole system.

If resources are to be obtained late and released early it could be possible for an external state change to be affected by a released resource and observed by the acquisition of a new resource. This would break the definition of atomic action. It follows that the only safe policy for resource usage is one that has two distinct phases. In the first 'growing' phase, resources can be requested (only); in the following 'shrinking' phase, resources can be released (but no new allocations can be made). With such a structure, the integrity of the atomic action is assured. However, it should be noted that if resources are released early then it will be more difficult to provide recovery if the atomic action fails. This is because the resource has been updated and another task may have observed the new state of the resource. Any attempt to invoke recovery in the other task may lead to the domino effect (see Section 2.5.3).

In all the following discussions, atomic actions are assumed to be two-phased; recoverable actions do not release any resources until the action successfully completes.

7.1.2 Atomic transactions

Within the theories of operating systems and databases, the term **atomic transaction** is often used. An atomic transaction has all the properties of an atomic action plus the added feature that its execution is allowed either to succeed or to fail. By failure, it is meant that an error has occurred from which the transaction cannot recover; for example, a processor failure. If an atomic action fails then the components of the system, which are being manipulated by the action, may be left in an inconsistent state. With an atomic transaction, this cannot happen because the components are returned to their original state (that is, the state they were *before* the transaction commenced). Atomic transactions are sometimes called **recoverable actions** and, unfortunately, the terms **atomic action** and **atomic transaction** are often interchanged.

The two distinctive properties of atomic transactions are:

- **failure atomicity** – meaning that the transaction must either complete successfully or (in the case of failure) have no effect;
- **synchronization atomicity** (or isolation) – meaning that the transaction is indivisible in the sense that its partial execution cannot be observed by any concurrently executing transaction.

Although atomic transactions are useful for those applications which involve the manipulation of databases, they are not suitable for programming fault-tolerant systems *per se*. This is because they imply that some form of recovery mechanism will be supplied by the system. Such a mechanism would be fixed, with the programmer having no control over

its operation. Although atomic transactions provide a form of backward error recovery, they do not allow recovery procedures to be performed. Notwithstanding these points, atomic transactions do have a role in protecting the integrity of a real-time database system.

7.1.3 Requirements for atomic actions

If a real-time programming language is to be capable of supporting atomic actions, it must be possible to express the requirements necessary for their implementation. These requirements are independent of the notion of a task and the form of intertask communication provided by a language. They are the following.

- **Well-defined boundaries** – each atomic action should have a start, an end and a side boundary. The start boundary is the location in each task involved in the atomic action where the action is deemed to start. The end boundary is the location in each task involved in the atomic action where the action is deemed to end. The side boundary separates those tasks involved in the atomic action from those in the rest of the system.
- **Indivisibility (isolation)** – an atomic action must not allow the exchange of any information between the tasks active inside the action and those outside (resource managers excluded). If two atomic actions do share data then the value of that data after the atomic actions is determined by the strict sequencing of the two actions in some order.
  There is no implied synchronization at the start of an atomic action. Tasks can enter at different times. However, there is an implied synchronization at the end of an atomic action; tasks are not allowed to leave the atomic action until all tasks are willing and able to leave.
- **Nesting** – atomic actions may be nested as long as they do not overlap with other atomic actions. Consequently, in general, only strict nesting is allowed (two structures are strictly nested if one is completely contained within the other).
- **Concurrency** – it should be possible to execute different atomic actions concurrently. One way to enforce indivisibility is to run atomic actions sequentially. However, this could seriously impair the performance of the overall system and therefore should be avoided. Nevertheless, the overall effect of running a collection of atomic actions concurrently must be the same as that which would be obtained from serializing their executions.
- As it is the intention that atomic actions should form the basis of damage confinement, they must allow recovery procedures to be programmed.

Figure 7.1 diagrammatically represents the boundaries of a nested atomic action in a system of six tasks. Action B involves only tasks P_3 and P_4, whereas action A also includes P_2 and P_5. The other tasks (P_1 and P_6) are outside the boundaries of both atomic actions.

It, perhaps, should be noted at this point that some definitions of atomic actions require that all tasks be synchronized on *both* entry and exit of the action.

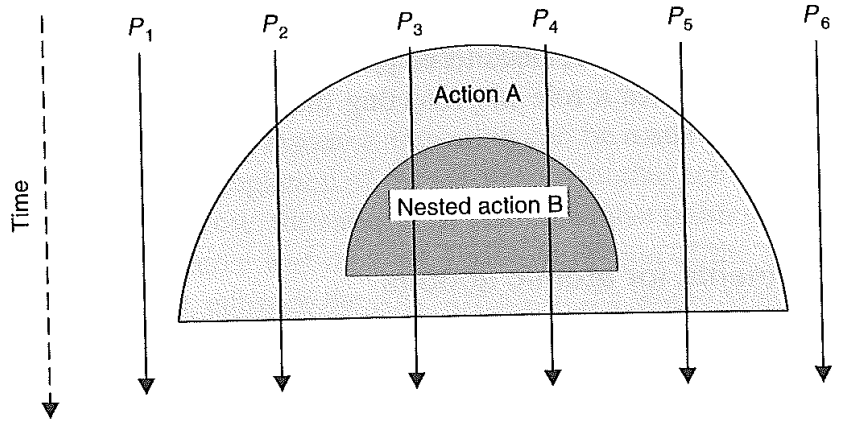

Figure 7.1 Nested atomic actions.

7.2 Atomic actions in C/Real-Time POSIX, Ada and Real-Time Java

Atomic actions provide structuring support for the software of large embedded systems. To get the full benefit of this aid requires the support of the real-time language. Unfortunately, such support is not directly provided by any of the major languages. This section considers the suitability of C/Real-Time POSIX, Ada and Real-Time Java for programming atomic actions. Following this, a possible language framework is given, and then this framework is extended to provide forward and backward error recovery.

The problem of resource allocation is postponed until Chapter 8. For now, it is assumed that resources have two modes of use: sharable and non-sharable, with some resources being amenable to both sharable and non-sharable modes. Furthermore, it is assumed that all actions are two-phased, and that the resource manager will ensure that appropriate usage is made of the resources. Also tasks within an action synchronize their own access to the resource to avoid any interference.

Atomic actions could be encapsulated within the monitor-like construct that is found in each of the languages. However, this would not allow any parallelism within an action. Hence, an alternative approach is adopted. In a similar fashion to the solution to the reader/writers problem presented in Section 5.9.1, the entry and exit protocols are implemented with a monitor-like construct and then the application code is surrounded by calls to the routines. Figure 7.2 illustrates an atomic action that requires three tasks (T_1, T_2 and T_3). Each task has its own role to play in the action. These roles and represented by the action procedures: for example, T_1 calls Action Procedure 1, etc. These action procedures then liaise with an action controller to provide the required synchronization. Figure 7.3 shows the structure of the roles.

7.2.1 Atomic actions and C/Real-Time POSIX mutexes

Using C and the Real-Time POSIX API, support for atomic actions following the structure given above can be provided as follows. Here only a two thread action is illustrated, but the approach is easily extended to more threads.

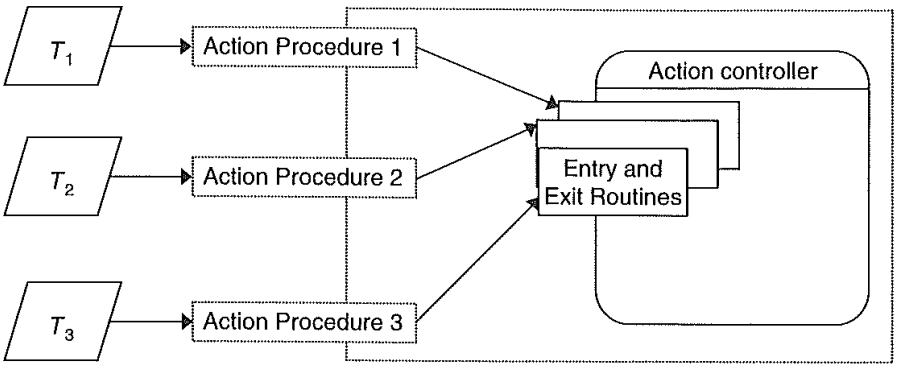

Figure 7.2 The structure of an action controller.

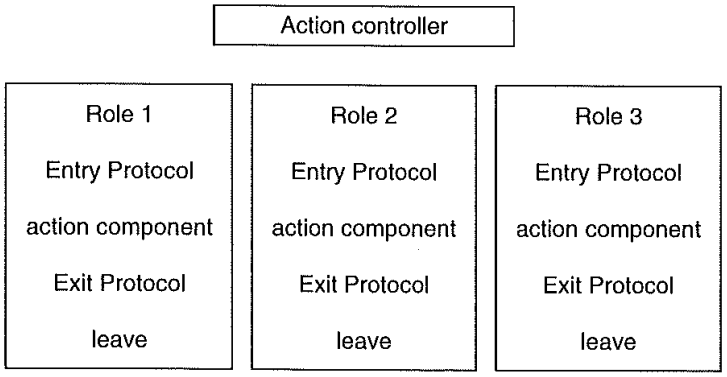

Figure 7.3 Using the action controller.

First, the mutex, condition variables and shared variables needed by the action controller are encapsulated in a `typdef`. Several boolean variables are used to represent the state of the action, and a mutex and four condition variables (and their associated attribute objects) provide the necessary synchronization.

```
#include <pthread.h>

typedef unsigned int boolean;
#define false 0
#define true (!false)

typedef struct {
  boolean first_process_active;
  boolean second_process_active;
  boolean first_process_finished;
  boolean second_process_finished;

  pthread_mutexattr_t mutex_att;
  pthread_mutex_t mutex;

  pthread_condattr_t cond_att;
  pthread_cond_t no_first_process, no_second_process;
  pthread_cond_t atomic_action_ends1, atomic_action_ends2;
} monitor;
```

The entry and exit protocols in support of the first role can now be given:

```
void entry_1(monitor *M) {
  PTHREAD_MUTEX_LOCK(&M->mutex);
    while(M->first_process_active)
      PTHREAD_COND_WAIT(&M->no_first_process, &M->mutex);
    M->first_process_active = true;
}

void exit_1(monitor *M) {
  while(!M->second_process_finished)
    PTHREAD_COND_WAIT(&M->atomic_action_ends2, &M->mutex);
  PTHREAD_COND_SIGNAL(&M->atomic_action_ends1);
  M->first_process_active = false;
  PTHREAD_COND_SIGNAL(&M->no_first_process);
  PTHREAD_MUTEX_UNLOCK(&M->mutex);
}
```

To enter the action, the action controller first obtains a mutual exclusion lock so it knows that its action will not be interfered with. It then checks that there is no other thread executing this role. If there is, the controller forces the calling thread to wait until the role is available. If the role is available, the thread is allowed to continue and a note is made that the role is now active.

When the role is finished, the exit protocol is called. Here, the controller checks to see if the other thread has finished its role. If it hasn't the first thread is made to wait. If it has, the action controller signals that the first thread has finished its role, then returns the data structure to a position from which a new action can start. The second role is similarly structured.

On exit from its role, the thread signals that it has finished and then waits for the other thread to finish. Of course, the data structure must be initialized via some appropriate routine:

```
void init_action(monitor *M) {
 // initialize all mutex and conditions with the appropriate
 // attributes; initialize booleans to false
}
```

The structure of the application code takes the following form:

```
void code_for_first_process(monitor *M) {
  entry_1(M);
    // get resource in non-sharable mode
    // update resource

    // signal second process that it is ok
    // for it to access resource

    // any final processing
  exit_1(M);

}
```

```
void code_for_second_process (monitor *M) {
  entry_2(M);
    // initial processing

    // wait for first process to signal
    // that it is ok to access resource

    // access resource

    // release resource
  exit_2(M);
}
```

Although the above approach has been able to implement the synchronization aspects of an atomic action, there is no way to ensure that the application threads do not communicate with any other threads.

7.2.2 Atomic actions in Ada

The Ada approach to implementing an atomic action is illustrated below for a three task system. It is similar to the C/Real-Time POSIX approach but uses guards instead of condition variables. Here, the action is encapsulated within an Ada package:

```
package Action_X is
  procedure Code_For_First_Task(--params);
  procedure Code_For_Second_Task(--params);
  procedure Code_For_Third_Task(--params);
end Action_X;
```

The body of the package contains the action controller, which is implemented as a protected type.

```
package body Action_X is
  protected Action_Controller is
    entry First;
    entry Second;
    entry Third;
    entry Finished;
  private
    First_Here : Boolean := False;
    Second_Here : Boolean := False;
    Third_Here : Boolean := False;
    Release : Boolean := False;
  end Action_Controller;

  protected body Action_Controller is
    entry First when not First_Here is
    begin
      First_Here := True;
    end First;
```

```
  entry Second when not Second_Here is
  begin
    Second_Here := True;
  end Second;

  entry Third when not Third_Here is
  begin
    Third_Here := True;
  end Third;

  entry Finished when Release or Finished'Count = 3 is
  begin
    if Finished'Count = 0 then
      Release := False;
      First_Here := False;
      Second_Here := False;
      Third_Here := False;
    else
      Release := True;
    end if;
  end Finished;
end Action_Controller;
```

In the above code, the action is synchronized by the `Action_Controller` protected object. This ensures that only three tasks can be active in the action at any one time and that they are synchronized on exit. The boolean `Release` is used to program the required release conditions on `Finished`. The first two calls on `Finished` will be blocked as both parts of the barrier expression are false. When the third call comes, the `Count` attribute will become three; the barrier becomes open and one task will execute the entry body. The `Release` variable ensures that the other two tasks are both released. The last task to exit must ensure that the barrier is closed again.

The application code also has a similar structure to the C/Real-Time POSIX approach.

```
procedure Code_For_First_Task(--params) is
begin
  Action_Controller.First;
  -- acquire resources
  -- the action itself, communicates with tasks executing
  -- inside the action via resources
  Action_Controller.Finished;
  -- release resources
end Code_For_First_Task;

-- similar for second and third task
begin
  -- any initialization of local resources
end Action_X;
```

More details on how to program atomic actions in Ada can be found in Wellings and Burns (1997).

7.2.3 Atomic actions in Java

The previous sections have illustrated the basic structure for programming atomic actions. The Java approach could follow a similar structure. However, this section takes the opportunity to expand on this approach so that the Java support can easily be extended using inheritance.

First, an interface can be defined for a three-way atomic action:

```
public interface ThreeWayAtomicAction {
  public void role1();
  public void role2();
  public void role3();
}
```

Using this interface, it is possible to provide several action controllers that implement a variety of models. Applications can then choose the appropriate controller without having to change their code.

The following action controller implements the same semantics as that given previously for C/Real-Time POSIX and Ada. A synchronized `Controller` class implements the required entry and exit synchronization protocols. The `finished` method is, however, a little more complex than its Ada counterpart. This is due to the semantics of `wait` and `notifyAll`. In particular, it is necessary to count the tasks as they leave the action (using `toExit`) in order to know when to reset the internal data structures for the next action. In Ada this was achieved via the `Count` attribute.

```
public class AtomicActionControl
       implements ThreeWayAtomicAction {
  protected Controller Control;
  public AtomicActionControl()  { // constructor
    Control = new Controller();
  }

  class Controller {
    protected boolean firstHere, secondHere, thirdHere;
    protected int allDone;
    protected int toExit;
    protected int numberOfParticipants;

    Controller() {
      firstHere = false;
      secondHere = false;
      thirdHere = false;
      allDone = 0;
      numberOfParticipants = 3;
      toExit = numberOfParticipants;
    }

    synchronized void first() throws InterruptedException {
      while(firstHere) wait();
      firstHere = true;
    }
```

```
  synchronized void second() throws InterruptedException {
    while(secondHere) wait();
    secondHere = true;
  }

  synchronized void third() throws InterruptedException {
    while(thirdHere) wait();
    thirdHere = true;
  }

  synchronized void finished() throws InterruptedException {
    allDone++;
    if(allDone == numberOfParticipants) {
      notifyAll();
    } else while(allDone != numberOfParticipants) {
        wait();
    };
    toExit--;
    if(toExit == 0) {
      firstHere = false;
      secondHere = false;
      thirdHere = false;
      allDone = 0;
      toExit = numberOfParticipants;
      notifyAll(); // release all waiting for the next action
    }
  }
}

public void role1() {
  boolean done = false;
  while(!done) {
    try {
      Control.first();
      done = true;
    } catch (InterruptedException e) {
      // ignore
    }
  }

  // .... perform action

  done = false;
  while(!done) {
    try {
      Control.finished();
      done = true;
    } catch (InterruptedException e) {
      // ignore
    }
  }
}

public void role2() {
  // similar to role1
}
```

```
  public void role3() {
    // similar to role1
  }
}
```

Given the above framework, it is now possible to extend this to produce, say, a four-way action:

```
public interface FourWayAtomicAction
       extends ThreeWayAtomicAction {
  public void role4();
}
```

and then

```
public class NewAtomicActionControl extends AtomicActionControl
             implements FourWayAtomicAction {
  public NewAtomicActionControl() {
    Control = new RevisedController();
  }

  class RevisedController extends Controller {
    protected boolean fourthHere;

    RevisedController() {
      super();
      fourthHere = false;
      numberOfParticipants = 4;
      toExit = numberOfParticipants;
    }

    synchronized void fourth() throws InterruptedException {
      while(fourthHere) wait();
      fourthHere = true;
    }

    synchronized void finished() throws InterruptedException {
      super.finished();
      if(allDone == 0) {
        fourthHere = false;
        notifyAll();
      }
    }
  }
  public void role4() {
    boolean done = false;
    while(!done) {
      try {
        // As Control is of type Controller, it must first
        // be converted to a RevisedController in order
        // to call the fourth method
        ((RevisedController)Control).fourth();
        done = true;
      } catch (InterruptedException e) {
```

```
        // ignore
      }
    }

    // .... perform action

    done = false;
    while(!done) {
      try {
        Control.finished();
        done = true;
      } catch (InterruptedException e) {
        // ignore
      }
    }
  }
}
```

Note that it has been necessary to override the `finished` method. Care must be taken here because of Java's automatic run-time dispatching of method calls. All calls to `finished` in the original code will dispatch to the overridden method.

7.3 Recoverable atomic actions

Although the various models described above have enabled a simple atomic action to be expressed, they all rely on programmer discipline to ensure that no interactions with external tasks occur (apart from with resource allocators). Moreover, they assume that no task within an atomic action is aborted; if the real-time language supports an abort facility then a task could be asynchronously removed from the action leaving the action in an inconsistent state.

In general, none of the mainstream languages or operating systems directly supports backward or forward error recovery facilities in the context of atomic actions. (However, C/Real-Time POSIX, Ada and Java do provide asynchronous notification mechanisms which can be used to help program recovery – see Sections 7.5, 7.6 and 7.7.) Language mechanisms have been proposed in research-oriented systems. In order to discuss these mechanisms, a simple language framework for atomic actions is introduced. The proposed recovery mechanisms are then discussed in the context of this framework.

To simplify the framework, only static tasks will be considered. Also it will be assumed that all the tasks taking part in an atomic action are known at compile-time. Each task involved in an action declares an action statement which specifies: the action name, the other tasks taking part in the action, and the code to be executed by the declaring task on entry to the action. For example a task P_1 which wishes to enter into an atomic action A with tasks P_2 and P_3 would declare the following action:

```
action A with (P2, P3) do
    -- acquire resources
    -- communicate with P2 and P3
    -- release resources
end A;
```

It is assumed that resource allocators are known and that communication inside the action is restricted to the three P tasks (together with external calls to the resource allocators). These restrictions are checked at compile-time. All other tasks declare similar actions, and nested actions are allowed as long as strict nesting is observed. Note that if the tasks are not known at compile-time, then any communication with a task will be allowed only if both tasks are active in the same atomic action.

The imposed synchronization on the action is as follows. Tasks entering the action are not blocked. A task is blocked inside the action only if it has to wait for a resource to be allocated, or if it attempts to communicate with another task inside the action and that task is either active in the action, but not in a position to accept the communication, or is not as yet active in the action.

Tasks may leave the action only when all tasks active in the action wish to leave. This was not the case in the examples given earlier. There it was assumed that all tasks must enter the action before any could leave. Here it is possible for a subset of the named tasks to enter the action and subsequently leave (without recourse to any interactions with the missing tasks). This facility is deemed to be essential in a real-time system where deadlines are important. It solves the **deserter** problem where all tasks are held in an action because one task has not arrived. This issue will be considered along with error recovery in the next two subsections.

7.3.1 Atomic actions and backward error recovery

Atomic actions are important because they constrain the flow of information around the system to well-defined boundaries and therefore can provide the basis for both damage confinement and error recovery. In this section, backward error recovery between concurrent tasks is described.

In Chapter 2, it was shown that when backward error recovery is applied to groups of communicating tasks, it is possible for all the tasks to be rolled back to the start of their execution. This was the so-called *domino effect*. The problem occurred because there was no consistent set of recovery points or a recovery line. An atomic action provides that recovery line automatically. If an error occurs inside an atomic action then the tasks involved can be rolled back to the start of the action and alternative algorithms executed; the atomic action ensures that tasks have not passed any erroneous values through communication with tasks outside the action. When atomic actions are used in this way they are called **conversations** (Randell, 1975).

With conversations each action statement contains a recovery block. For example:

```
action A with (P2, P3) do
   ensure <acceptance test>
   by
      -- primary module
   else by
      -- alternative module
   else by
      -- alternative module
   else error
end A;
```

Other tasks involved in the conversation declare their part in the action similarly. The basic semantics of a conversation can be summarized as follows.

- On entry to the conversation, the state of a task is saved. The set of entry points forms the recovery line.
- While inside the conversation, a task is allowed only to communicate with other tasks active in the conversation and general resource managers. As conversations are built from atomic actions, this property is inherited.
- In order to leave the conversation, all tasks active in the conversation must have passed their acceptance test. If this is the case, then the conversation is finished and all recovery points are discarded.
- *If any task fails its acceptance test, all tasks have their state restored to that saved at the start of the conversation and they execute their alternative modules.* It is, therefore, assumed that any error recovery to be performed inside a conversation *must* be performed by *all* tasks taking part in the conversation.
- Conversations can be nested, but only strict nesting is allowed.
- If all alternatives in the conversation fail then recovery must be performed at a higher level.

It should be noted that in conversations, as defined by Randell (1975), all tasks taking part in the conversation must have entered the conversation before any of the other tasks can leave. This differs from the semantics described here. If a task does not enter into a conversation, either because of tardiness or because it has failed, then as long as the other tasks active in the conversation do not wish to communicate with it, the conversation can complete successfully. If a task does attempt to communicate with a missing task then either it can block and wait for the task to arrive or it can continue. Adopting this approach has two benefits (Gregory and Knight, 1985).

- It allows conversations to be specified where participation is not compulsory.
- It allows tasks with deadlines to leave the conversation, continue and if necessary take some alternative action.

Although conversations allow groups of tasks to coordinate their recovery, they have been criticized. One important point is that when a conversation fails, all the tasks are restored and all enter their alternative modules. This forces the same tasks to communicate again to achieve the desired effect; a task cannot break out of the conversation. This may be not what is required. Gregory and Knight (1985) point out that in practice when one task fails to achieve its goal in a primary module through communication with one group of tasks, it may wish to communicate with a completely new group of tasks in its secondary module. Furthermore, the acceptance test for this secondary module may be quite different. There is no way to express these requirements using conversations.

7.3.2 Atomic actions and forward error recovery

It was pointed out in Chapter 2 that, although backward error recovery enables recovery from unanticipated errors, it is difficult to undo any operation that may have been

performed in the environment in which the embedded system operates. Consequently forward error recovery and exception handling must be considered. In this section, exception handling between the concurrent tasks involved in an atomic action is discussed.

With backward error recovery, when an error occurs all tasks involved in the atomic action participate in recovery. The same is true with exception handling and forward error recovery. If an exception occurs in one of the tasks active in an atomic action then that exception is raised in *all* tasks active in the action. The exception is said to be **asynchronous** as it originates from another task. The following is a possible Ada-like syntax for an atomic action supporting exception handling:

```
action A with (P2, P3) do
   -- the action
exception
   when exception_a =>
        -- sequence of statements
   when exception_b =>
        -- sequence of statements
   when others =>
      raise atomic_action_failure;
end A;
```

With the termination model of exception handling, if all tasks active in the action have a handler and all handle the exception without raising any further exception, then the atomic action completes normally. If a resumption model is used, when the exception has been handled, the tasks active in the atomic action resume their execution at the point where the exception was raised.

With either model, if there is no exception handler *in any one of the tasks active in the action* or one of the handlers fails then *the atomic action fails* with a standard exception `atomic_action_failure`. This exception is raised in all the involved tasks.

There are two issues which must be considered when exception handling is added to atomic actions: resolution of concurrently raised exceptions and exceptions in nested actions (Campbell and Randell, 1986). These are now briefly reviewed.

Resolution of concurrently raised exceptions

It is possible for more than one task active in an atomic action to raise different exceptions at the same time. As Campbell and Randell (1986) point out, this event is likely if the errors resulting from some fault cannot be uniquely identified by the error-detection facility provided by each component of the atomic action. If two exceptions are simultaneously raised in an atomic action, there may be two separate exception handlers in each task. It may be difficult to decide which one should be chosen. Furthermore, the two exceptions in conjunction constitute a third exception which is the exception that indicates that both the other two exceptional conditions have occurred.

In order to resolve concurrently raised exceptions, Campbell and Randell propose the use of an **exception tree**. If several exceptions are raised concurrently then the exception used to identify the handler is that at the root of the smallest subtree that contains all the exceptions (although it is not clear how to combine any parameters

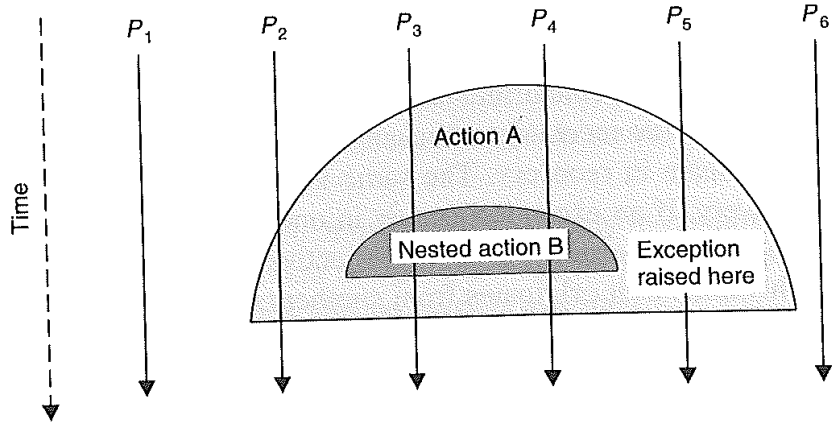

Figure 7.4 An exception in a nested atomic action.

associated with this exception). Each atomic action component can declare its own exception tree; the different tasks involved in an atomic action may well have different exception trees.

Exceptions and internal atomic actions

Where atomic actions are nested, it is possible for one task active in an action to raise an exception when other tasks in the same action are involved in a nested action. Figure 7.4 illustrates the problem.

When the exception is raised, all tasks involved must participate in the recovery action. Unfortunately, the internal action, by definition, is indivisible. To raise the exception in that action would potentially compromise that indivisibility. Furthermore, the internal action may have no knowledge of the possible exceptions that can be raised.

Campbell and Randell (1986) have discussed two possible solutions to this problem. The first solution is to hold back the raising of the exception until the internal action has finished. This they reject because:

- In a real-time system, the exception being raised may be associated with the missing of a deadline. To hold back the recovery procedure may seriously place in jeopardy the action's timely response.
- The error condition detected may indicate that the internal action may never terminate because some deadlock condition has arisen.

For these reasons, Campbell and Randell allow internal actions to have a predefined abortion exception. This exception is raised to indicate to the action that an exception has been raised in a surrounding action and that the preconditions under which the action was invoked are no longer valid. If such an exception is raised, the internal action should invoke fault-tolerant measures to abort itself. Once the action has been aborted, the containing action can handle the original exception.

If the internal action cannot abort itself then it must signal an atomic action failure exception. This then may be combined with the outstanding exception so as to affect the choice of recovery performed by the surrounding action. If no abortion exception is

defined, the surrounding action must wait for the internal action to complete. Alternatively, a default handler could be provided which would raise the atomic action failure exception.

7.4 Asynchronous notification

Although forward and backward error recovery have been discussed separately, in reality they may need to be combined in many real-time systems. Backward error recovery is needed to recover from unanticipated errors, and forward error recovery is needed to undo or ameliorate any interaction with the environment. Indeed, forward error handling can be used to implement a backward error recovery scheme – see Section 7.6.3.

As discussed in Section 7.2, none of the major real-time languages supports atomic actions, and it is necessary to use more primitive language facilities to achieve the same effect. The same is true for recoverable actions. One of the main requirements for a recoverable action is to be able to gain the attention of a task involved in an action and notify it that an error has occurred in another task. Most languages and operating systems support some form of asynchronous notification mechanism. As with exceptions, there are two basic models: resumption and termination.

The resumption model of asynchronous notification handling (often called **event handling**) behaves like a software interrupt. A task indicates which events it is willing to handle; when the event is signalled, the task is interrupted (unless it has temporally inhibited the event from being delivered) and an event handler is executed. The handler responds to the asynchronous event and then the task continues with its execution from the point at which it was interrupted. This, of course, sounds very similar to the resumption model of exception handling given in Section 3.2.4. The main difference is that the event is usually *not* signalled by the affected task (or because of an operation the affected task is performing), but is signalled *asynchronously*. However, many operating systems do not provide a special exception-handling facility for synchronous exception handling, but use the asynchronous events mechanisms instead. The C/Real-Time POSIX signal facility is an example of an asynchronous event model with resumption.

Note that, with the resumption model, the flow of control of a task is only temporarily changed; after the event has been handled the task is resumed. In a multithreaded process, it is possible to associate a distinct thread with the event and to schedule the thread when the event is signalled. Real-Time Java provides support for this model.

With the termination model of asynchronous notification, each task specifies a domain of execution during which it is prepared to receive an asynchronous notification that will cause the domain to be terminated. This form is often called **asynchronous transfer of control** or ATC. If an ATC request occurs outside this domain, it may be ignored or queued. After the ATC has been handled, control is returned to the interrupted task at a location different from that where the ATC was delivered. This, of course, is very similar to the termination model of exception handling. The Ada and Real-Time Java languages support asynchronous transfer of control mechanisms.

An extreme form of asynchronous notification with termination semantics is to abort the task and allow another task to perform some recovery. All operating systems and most concurrent programming languages provide such a facility. However, aborting

a process can be expensive and is often an extreme response to many error conditions. Aborting a thread is less expensive but still potentially dangerous as it can leave resources in an undefined state. Consequently, some form of safe asynchronous notification mechanism is also required.

The inclusion of an asynchronous notification mechanism into a language (or operating system) is controversial, as it complicates the language's semantics and increases the complexity of the run-time support system. This section thus first considers the application requirements which justify the inclusion of such a facility. The C/Real-Time POSIX, Ada and Real-Time Java models of asynchronous notification are then discussed.

7.4.1 The user need for asynchronous notification

The fundamental requirement for an asynchronous notification facility is to enable a task to respond *quickly* to a condition which has been detected by another task. The emphasis here is on a quick response; clearly a task can always respond to an event by simply polling or waiting for that event. The notification of the event could be mapped onto the task's communication and synchronization mechanism. The handling task, when it is ready to receive the event, simply issues the appropriate request.

Unfortunately, there are occasions when polling for events or waiting for the event to occur is inadequate. These include the following.

- **Error recovery** – this chapter has already emphasized that when groups of tasks undertake atomic actions, an error detected in one task requires all other tasks to participate in the recovery. For example, a hardware fault may mean that the task will never finish its planned execution because the preconditions under which it started no longer hold; the task may never reach its polling point. Also, a timing fault might have occurred, which means that the task will no longer meet the deadline for the delivery of its service. In both these situations, the task must be informed that an error has been detected and that it must undertake some error recovery as quickly as possible.
- **Mode changes** – a real-time system often has several modes of operation. For example, a fly-by-wire civil aircraft may have a take-off mode, a cruising mode and a landing mode. On many occasions, changes between modes can be carefully managed and will occur at well-defined points in the system's execution, as in a normal flight plan for a civil aircraft. Unfortunately, in some application areas, mode changes are expected but cannot be planned. For example, a fault may lead to an aircraft abandoning its take-off and entering an emergency mode of operation; or an accident in a manufacturing task may require an immediate mode change to ensure an orderly shutdown of the plant. In these situations, tasks must be quickly and safely informed that the mode in which they are operating has changed, and that they now need to undertake a different set of actions.
- **Scheduling using partial/imprecise computations** – there are many algorithms where the accuracy of the results depends on how much time can be allocated to their calculation. For example, numerical computations, statistical estimations and heuristic searches may all produce an initial estimation of the required result, and then refine that result to a greater accuracy. At run-time, a certain amount of time

can be allocated to an algorithm, and then, when that time has been used, the task must be interrupted to stop further refinement of the result.

- **User interrupts** – in a general interactive computing environment, users often wish to stop the current processing because they have detected an error condition and wish to start again.

7.5 Asynchronous notification in C/Real-Time POSIX

The C/Real-Time POSIX model supports two forms of asynchronous notification: a resumption facility based on signals, and a thread cancellation mechanism.

7.5.1 C/Real-Time POSIX signals

C/Real-Time POSIX signals support the resumption model. They are also used for a class of environment-detected synchronous errors (such as divide by zero, illegal pointer and so on).

There are a number of predefined signals, each of which is allocated an integer value. There are also an implementation-defined number of signals which are available for application use. Each signal has a default handler, which usually terminates the receiving process. Example signals are: `SIGABRT` for abnormal termination, `SIGALARM` for alarm clock expiry, `SIGILL` for illegal instruction exception, `SIGRTMIN` for the identifier of the first real-time application-definable exception, and `SIGRTMAX` for the identifier of the last real-time application-definable exception. Only those signals whose numbers lie between `SIGRTMIN` and `SIGRTMAX` are considered to be real-time by POSIX. A real-time signal is one which can have extra information passed to the handler by the process which generated it; in addition, they are queued.

The program can specify that it wants to receive signals as a result of certain events occurring; for example, when a message is received on a message queue (see the `mq_notify` function in Program 6.1) or a timer expires (see the `timer_create` function in Program 10.7). An application can configure its requirements for generating and receiving signals via the use of several data structures. These are shown in Program 7.1.

The `sigevent` structure is used to pass information to the signal-handling subsystem concerning the generation of a signal. For example, `timer_create` functions take an object of this type to indicate what should happen when the timer expires. The `sigev_signo` indicates the identity of the signal to be generated, and `sigev_value` indicates what data should be passed to the handler. The `sigev_notify` indicates how the program should be notified when the signal is generated. There are three possibilities.

- `SIGEV_NONE` – no notification occurs when the signal is generated.
- `SIGEV_SIGNAL` – a signal is queued with the application-defined data.
- `SIGEV_THREAD` – a new pthread should be created with a start routine (see `pthread_create` function in Program 4.5) defined by `sigev_notify_function` and thread attributes defined by `sigev_notify_attributes`.

Program 7.1 The C/Real-Time POSIX interface to data structures supporting signal generation.

```
/* used with message queue notification, timers etc */

struct sigevent {
  int sigev_notify;
    /* SIGEV_SIGNAL, */
    /* SIGEV_THREAD or SIGEV_NONE */

  int sigev_signo;  /* signal to be generated */

  union sigval sigev_value; /* value to be queued */

  void (*)sigev_notify_function(union sigval s);
  /* function to be treated as thread */
  pthread_attr_t *sigev_notify_attributes;
  /* thread attributes */
};

union sigval {
  int sival_int;
  void *sival_ptr;
};
```

If a signal is to be delivered to the process itself rather than via creating a thread, then further types and functions are available in C/Real-Time POSIX to specify the program's response. These are shown in Program 7.2. Most of the predefined C/Real-Time POSIX signals are delivered in this manner.

There are three ways in which a process can deal with a signal.

- It can **block** the signal and either handle it later or accept it.
- It can **handle** the signal by setting a function to be called whenever it occurs.
- It can **ignore** the signal altogether (in which case the signal is simply lost).

A signal that is not blocked and not ignored is **delivered** as soon as it is **generated**. A signal that is blocked is **pending** delivery, or may be **accepted** by calling one of the `sigwait()` functions.

Blocking a signal

C/Real-Time POSIX maintains the set of signals that have been currently masked by a process. The function `sigprocmask` is used to manipulate this set. The `how` parameter is set to: `SIG_BLOCK` to add signals to the set, `SIG_UNBLOCK` to subtract signals from the set, or `SIG_SETMASK` to replace the set.[1] The other two parameters contain pointers

[1] `SIG_BLOCK`, `SIG_UNBLOCK` and `SIG_SETMASK` are compile-time constants.

Program 7.2 An abridged C/Real-Time POSIX interface to signals.

```
typedef ... sigset_t;

/* the following manipulates the signal mask */
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
  /* how = SIG_BLOCK -> the set is added to the current set */
  /* how = SIG_UNBLOCK -> the set is subtracted from the */
  /*       current set */
  /* how = SIG_SETMASK -> the given set becomes the mask */

/* the following routines allow a signal */
/* set to be created and manipulated */
int sigemptyset(sigset_t *s); /* initialize a set to empty */
int sigfillset(sigset_t *s); /* initialize a set to full */
int sigaddset(sigset_t *s, int signum); /* add a signal */
int sigdelset(sigset_t *s, int signum); /* remove a signal */
int sigismember(const sigset_t *s, int signum);
   /* returns 1 if a member */

/* the following support signal handling */

typedef struct { /* signal parameters */
  int si_signo;
  int si_code;
  union sigval si_value;
} siginfo_t;

struct sigaction {
  void (*sa_handler) (int signum); /* non real-time handler */
  void (*sa_sigaction) (int signum, siginfo_t *data,
                        void *extra); /*real-time handler */
  sigset_t sa_mask; /* signals to mask during handler */
  int sa_flags; /*indicates if signal is to be queued */
};

int sigaction(int sig, const struct sigaction *reaction,
              struct sigaction *old_reaction);
/* sets up a signal handler, reaction, for sig */

/* the following functions allow a  */
/* process to wait for a signal */

int sigsuspend(const sigset_t *sigmask);
int sigwaitinfo(const sigset_t *set, siginfo_t *info);
int sigtimedwait(const sigset_t *set, siginfo_t *info,
                 const struct timespec *timeout);

/* the following functions allow a  */
/* signal to be sent */
int kill (pid_t pid, int sig);
   /* send the signal sig to the process pid */
int sigqueue(pid_t pid, int sig, const union sigval value);
   /* send signal and data */

/* All the above functions return -1 when errors have occurred. */
/* A shared variable errno contains the reason for the error */
```

to the set of signals to be added/subtracted/replaced (`set`) and the returned value of the old set (`oset`). Various functions (`sigemptyset`, `sigfillset`, `sigaddset`, `sigdelset` and `sigismember`) allow a set of signals to be manipulated.

When a signal is blocked, it remains pending until it is unblocked or accepted. When it is unblocked, it is then delivered. Note, some signals *cannot* be blocked.

Handling a signal

A signal handler can be set up using the function `sigaction`. The `sig` parameter indicates which signal is to be handled, `reaction` is a pointer to a structure containing information about the handler, and `old_reaction` points to information about the previous handler. Essentially, the information about a handler contains a pointer to the handler function (`sa_handler` if the signal is a non-real-time signal, or `sa_sigaction` if the signal is a real-time one), the set of signals to be masked during the handler's execution (`sa_mask`), and whether the signal is to be queued (indicated by setting `sa_flags` to the symbolic constant `SA_SIGINFO` – only signals whose value lies between `SIGRTMIN` and `SIGRTMAX` can be queued). The `sa_handler` member indicates the action associated with the signal and can be:

- `SIG_DFL` – default action (usually to terminate the process);
- `SIG_IGN` – ignore the signal;
- pointer to a function – to be called when the signal is delivered.

For non-real-time signals, only an integer parameter can be passed to the handler when the signal is generated. The value of this parameter normally indicates the signal itself (the same handler can be used for more than one signal). However, for the real-time signals, more data can be passed via a pointer to the `siginfo_t` structure. This structure contains the signal number (again), a code which indicates the cause of the signal (for example, a timer signal) and an integer or pointer value.

If more than one *real-time* signal is queued, the one with the lowest value is delivered first (that is, `SIGRTMIN` is delivered before `SIGRTMIN + 1` and so on).

A process can also wait for a signal to arrive using the functions `sigsuspend`, `sigwaitinfo` or `sigtimedwait`. The function `sigsuspend` replaces the mask with that given by the parameter to the call and suspends the process until:

(1) a non-blocked signal is delivered, and

(2) the associated handler is executed.

If the handler terminates the process, the `sigsuspend` function never returns; otherwise it returns with the signal mask reset to the state that existed *prior* to the call of `sigsuspend`.

The `sigwaitinfo` function also suspends the calling process until the signal arrives. However, this time the signal must be blocked, and thus the handler is not called. Instead, the function returns the selected signal number and stores the information about the delivered signal in the `info` argument. The function `sigtimedwait` has the same

semantics as `sigwaitinfo`, but allows a timeout to be specified for the suspension. If no signals are delivered by the timeout, `sigwaitinfo` returns with –1 and `errno` set to `EAGAIN`.

Care must clearly be taken when using signals for condition synchronization. There is a potential race condition between checking to see whether a signal has already arrived and issuing a request which causes suspension. The appropriate protocol is to block the signal first, then test to see if it has occurred and, if not, suspend and unblock the signal using one of the above functions.

Ignoring a signal

A signal can be ignored by simply setting the value of `sa_handler` to `SIG_IGN` in a call to the function `sigaction`.

Generating a signal

There are two ways in which a process can generate a signal to be sent to another process. The first is via the `kill` function and the second is via the `sigqueue` function. The latter can only send real-time signals.

Note, however, that a process can also request that a signal be sent to itself: when a timer expires (for example, `SIGALRM` – see Section 13.2.3), when asynchronous I/O completes, by the arrival of a message on an empty message queue (see Section 6.7), or by using the C raise statement.

A simple example of C/Real-Time POSIX signals and processes

As an illustration of C/Real-Time POSIX signals, consider the program fragment below. A process performs some computation periodically. The actual computation to be performed depends on a system-wide mode of operation. A mode change is propagated to all processes via an application-defined real-time signal `MODE_CHANGE`. The signal handler `change_mode` simply changes a global variable `mode`. The processes access `mode` at the beginning of each iteration. To ensure that the mode does not change while it is accessing it, the `MODE_CHANGE` signal is blocked.

```
#include <signal.h>

#define MODE_A 1
#define MODE_B 2
#define MODE_CHANGE SIGRTMIN + 1

int mode = MODE_A;

void change_mode(int signum, siginfo_t *data, void *extra) {
  /* signal handler */
  mode = data -> si_value.sival_int;
}
```

```
int main() {

  sigset_t mask, omask;
  struct sigaction s, os;
  int local_mode;

  SIGEMPTYSET(&mask);
  SIGADDSET(&mask, MODE_CHANGE);

  s.sa_flags = SA_SIGINFO;
  s.sa_mask = mask;
  s.sa_sigaction = &change_mode;
  s.sa_handler = &change_mode;

  SIGACTION(MODE_CHANGE, &s, &os); /* assign handler */

  while(1) {

    SIGPROCMASK(SIG_BLOCK, &mask, &omask);
      local_mode = mode;
    SIGPROCMASK(SIG_UNBLOCK, &mask, &omask);

    /* periodic operation using mode*/
    switch(local_mode) {
      case MODE_A:
        ...
        break;
      case MODE_B:
        ...
        break;
      default:
        ...
    }
  }
  ...
}
```

Signals and threads

The original C/POSIX signal model came from Unix and was extended to make it more appropriate for real-time when the Real-Time Extensions to POSIX were specified. With the POSIX Thread Extensions, the model has become more complex and is a compromise between a per process model and a per thread model. The following points should be noted.

- Signals which are generated as a result of a synchronous error condition, such as memory violation, are delivered only to the thread that caused the signal.
- Other signals may be sent to the process as a whole; however, each one is only delivered to a single thread in the process.
- The `sigaction` function sets the handler for *all* threads in the process.

- The functions `kill` and `sigqueue` still apply to processes. A new function `pthread_kill`

  ```
  int pthread_kill(pthread_t thread, int sig);
  ```

  allows a process to send a signal to an individual thread.
- If more than one thread is eligible to have a signal delivered to it, it is not defined which thread is chosen.
- If the action specified by a handler for the signal is for termination, *the whole process is terminated*, not just the thread.
- Signals can be blocked on a per thread basis using a function `pthread_sigmask` which has the same set of parameters as `sigprocmask`. The use of the function `sigprocmask` is not specified for a multithreaded process.
- The functions `sigsuspend`, `sigwaitinfo` or `sigtimedwait` operate on the calling thread not the calling process.
- A new function `sigwait`

  ```
  int sigwait(const sigset_t *set, int *sig);
  ```

  allows a thread to wait for one of several blocked signals to occur. It behaves the same as `sigwaitinfo()` except that the information associated with the signal is not returned. The signals are specified in the location referenced `set`. The function returns zero when a successful wait has been performed and the location referenced by `sig` contains the received signal.

  If one of the signals is already pending when the function is called, the function returns immediately. If more than one is pending, it is not defined which one is chosen unless only real-time signals are pending. In this case, the one with the lowest value is chosen.
- If a signal action is set by a thread to 'ignore', it is unspecified whether the signal is discarded immediately it is generated or remains pending.

Although C/Real-Time POSIX allows a thread or a process to handle an asynchronous event, care must be taken because some of the POSIX system calls are termed **async-signal unsafe** and **async-cancel unsafe**. It is undefined what happens if a signal interrupts an async-unsafe function that was called from a signal-catching function. For example, it is not safe to use the function `pthread_cond_signal` in a signal handler because of the race condition it introduces with the function `pthread_cond_wait`.

7.5.2 Asynchronous transfer of control and thread cancellation

C/Real-Time POSIX supports the C `setjmp` and `longjmp` mechanisms. As discussed in Section 3.3.3 these can be used to implement a termination model of exception handling. There are also versions of these that can be used in conjunction with signals (`sigsetjmp` and `siglongjmp`). These as well as saving the thread's state, save and restore the signal mask respectively. Hence it is conceivable that an ATC could be

Program 7.3 The C/Real-Time POSIX interface to thread cancellation.

```
#define PTHREAD_CANCEL_ASYNCHRONOUS ...
#define PTHREAD_CANCEL_ENABLE ...
#define PTHREAD_CANCEL_DEFERRED ...
#define PTHREAD_CANCEL_DISABLE ...
#define PTHREAD_CANCELED ...

int   pthread_cancel(pthread_t thread);
void  pthread_cleanup_push(void (* routine)(void *), void * arg);
void  pthread_cleanup_pop(int execute);


int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

implemented using a combination of these mechanisms. Unfortunately, these non-local jump functions and all the thread system calls are async-signal unsafe. C/Real-Time POSIX does not define what happens when an async-signal unsafe function is interrupted by a signal handler and the handler calls an async-signal unsafe function.

Threads can also be asynchronously terminated using the C/Real-Time POSIX thread cancellation facilities. However, a thread can control if and when it is cancelled via its cancellation state and cancellation type. The state can be **enabled** or `disabled`. If disabled, any attempt to cancel the thread is held pending. If enabled, cancellation will occur at point determined by the cancellation type. If the type is **asynchronous**, the cancellation may occur at any time. If `deferred`, the cancellation will occur at the next cancellation point that the thread reaches in its execution. The defaults for a newly created thread are enabled and deferred.

The full set of cancellation points is defined in the C/Real-Time POSIX standard. However, it is worth noting that most of the C/Real-Time POSIX pthread functions are *not* cancellation points.

In support of safe cancellation, C/Real-Time POSIX also allows a thread to define one or more routines that will be executed when cancellation is acted upon. These routines are stored on a stack, and the application can push and pop them during its execution. On cancellation (or when `pthread_exit` is called), they are popped off (in LIFO order) and executed.

Program 7.3 gives the full C/Real-Time POSIX API that supports cancellation and cleanup routines. Note that a thread cancels another thread by calling the function `pthread_cancel`, and a thread can introduce a cancellation point into its exection by calling `pthread_testcancel`.

7.5.3 C/Real-Time POSIX and atomic actions

Given the close interaction between activities in an atomic action, it is more appropriate to consider the action to take place between C/Real-Time POSIX threads rather than

C/Real-Time POSIX processes. Atomic actions are easiest to implement if a termination model of asynchronous transfer of control is used.

Given the problems of using a combination of `setjmp/longjmp` and signals, it is best to use thread creation and cancellation to program the required recovery. As C/Real-Time POSIX threads are designed to be efficient, this approach does not have the same performance penalty that would be associated with a more heavyweight process structure.

7.6 Asynchronous notification in Ada

The asynchronous notification facilities in Ada allow an application to respond to:

- events being signalled asynchronous from the external environment – this is in support of interrupt handling and will be considered in detail in Section 14.3;
- events being triggered by the passage of time – the handling for these events is executed at the priority of the clock device and they are considered in detail in Section 10.4.1;
- asynchronous transfer of control (ATC) requests on a task – supporting a termination model;
- task abortion.

There is no generalized mechanisms for a resumption model of asynchronous notification, hence this section will focus on ATC and task abortion.

7.6.1 Asynchronous transfer of control

Ada provides a structured form of asynchronous notification handling called **asynchronous transfer of control** (ATC). To emphasize that ATC is a form of communication and synchronization, the mechanism is built on top of the inter-task communication facility.

The Ada `select` statement was introduced in Chapter 6. It has the following forms:

- a selective accept (to support the server side of the rendezvous) – this was discussed in Section 6.5;
- a timed and a conditional entry call (to either a task or a protected entry) – this is discussed in Section 9.4.2;
- an asynchronous select – discussed here.

The asynchronous select statement provides an asynchronous notification mechanism with termination semantics.

The execution of the asynchronous select begins with the issuing of the triggering entry call or the issuing of the triggering delay. If the triggering statement is an entry call, the parameters are evaluated as normal and the call issued. If the call is queued, then a sequence of statements in an abortable part is executed.

If the triggering statement completes before the execution of the abortable part completes, the abortable part is aborted. When these activities have finished, the optional sequence of statements following the triggering statement is executed.

If the abortable part completes before the completion of the entry call, an attempt is made to cancel the entry call and, if successful, the execution of the asynchronous select statement is finished. The following illustrates the syntax:

```
select
  Trigger.Event;
  -- optional sequence of statements to be
  -- executed after the event has been received
then abort
  -- abortable sequence of statements
end select;
```

Note that the triggering statement can be a delay statement and, therefore, a timeout can be associated with the abortable part (see Section 9.4.3).

If the cancellation of the triggering event fails because the protected action or rendezvous has started, then the asynchronous select statement waits for the triggering statement to complete before executing the optional sequence of statements following the triggering statement.

Clearly, it is possible for the triggering event to occur even before the abortable part has started its execution. In this case the abortable part is not executed and therefore not aborted.

Consider the following example:

```
task Server is
    entry Atc_Event;
end Server;

task To_Be_Interrupted;

task body Server is
begin
  ...
  accept Atc_Event do
    Seq2;
  end Atc_Event;
  ...
end Server;

task body To_Be_Interrupted is
begin
  ...
  select  -- ATC statement
    Server.Atc_Event;
    Seq3;
  then abort
    Seq1;
  end select;
  Seq4;
  ...
end To_Be_Interrupted;
```

When the above ATC statement is executed, the statements which are executed will depend on the order of events that occur:

```
if the rendezvous is available immediately then
   Server.Atc_Event is issued
   Seq2 is executed
   Seq3 is executed
   Seq4 is executed (Seq1 is never started)
elsif no rendezvous starts before  Seq1 finishes then
   Server.Atc_Event is issued
   Seq1 is executed
   Server.Atc_Event is cancelled
   Seq4 is executed
elsif the rendezvous finishes before Seq1 finishes then
   Server.Atc_Event is issued
   partial execution of Seq1 occurs concurrently with Seq2
   Seq1 is aborted and finalized
   Seq3 is executed
   Seq4 is executed
else (the rendezvous finishes after Seq1 finishes)
   Server.Atc_Event is issued
   Seq1 is executed concurrently with partial execution of Seq2
   Server.Atc_Event cancellation is attempted but is unsuccessful
   execution of Seq2 completes
   Seq3 is executed
   Seq4 is executed
end if
```

Note that there is a race condition between `Seq1` finishing and the rendezvous finishing. The situation could occur where `Seq1` does finish but is nevertheless aborted.

Ada allows some operations to be **abort deferred**. If `Seq1` contains an abort-deferred operation, then its cancellation will not occur until the operation is completed. An example of such an operation is a call on a protected object.

The above discussion has concentrated on the concurrent behaviour of `Seq1` and the triggering rendezvous. Indeed, on a multiprocessor implementation it could be the case that `Seq1` and `Seq2` are executing in parallel. However, on a single-processor system, the triggering event will only ever occur if the action that causes it has a higher priority than `Seq1`. The normal behaviour will thus be the preemption of `Seq1` by `Seq2`. When `Seq2` (the triggering rendezvous) completes, `Seq1` will be aborted before it can execute again. And hence the ATC is 'immediate' (unless an abort-deferred operation is in progress).

Exceptions and ATC

With the asynchronous select statement, potentially two activities occur concurrently: the abortable part may execute concurrently with the triggering action (when the action is an entry call). In either one of these activities, exceptions may be raised and unhandled. Therefore, at first sight it may appear that potentially two exceptions can be propagated simultaneously from the select statement. However, this is not the case: one of the exceptions is deemed to be lost (that raised in the abortable part when it is aborted), and hence only one exception is propagated.

7.6.2 Task abortion

Tasks in Ada can be aborted using an abort statement; any task may abort any other named task by executing this statement

Once aborted tasks are said to become *abnormal*, and are prevented from interacting with any other task. Ideally, an abnormal task will stop executing immediately. However, some implementations may not be able to facilitate immediate shut down, and hence all Ada requires is that the task terminate before it next interacts with other tasks. Note that the Real-Time Systems Annex does require 'immediate' to be just that on a single processor system.

After a task has been marked as abnormal, execution of its body is aborted. This means that the execution of every construct in the task body is aborted, unless it is involved in the execution of an *abort-deferred operation*. The execution of an abort-deferred operation is allowed to complete before it is aborted. The same rules for aborting a task body also apply to aborting a sequence of statements in the asynchronous select statement discussed earlier.

If a construct, which has been aborted, is blocked outside an abort-deferred operation (other than at an entry call), the construct becomes abnormal and is immediately completed. Other constructs must complete no later than the next *abort completion point* (if any) that occurs outside an abort-deferred operation.

An abort completion point occurs at:

- the end of activation of a task;
- the point where the execution initiates the activation of another task;
- the start or end of an entry call, accept statement, delay statement or abort statement;
- the start of the execution of a select statement, or of the sequence of statements of an exception handler.

The following operations are defined to be abort-deferred:

- a protected action;
- waiting for an entry call to complete;
- waiting for termination of dependent tasks;
- the execution of an 'initialize' procedure, a 'finalize' procedure, or an assignment operation of an object with a controlled part.

In Ada, the equivalent of C/Real-Time POSIX's cleanup handlers can be programmed using `Controlled` types. Objects of these types must have their finalization routines executed when the task is aborted.

7.6.3 Ada and atomic actions

It was shown in Section 3.4 that backward error recovery in a sequential system could be implemented by exception handling. In this section, the Ada ATC facility and exception handling is used to implement backward and forward error recovery. It is assumed that the underlying Ada implementation and run-time are fault free, and therefore the strong typing provided by Ada will ensure that the Ada program itself remains viable.

Backward error recovery

The following package is a generic version of the one was given in Section 3.4 for saving and restoring a task's state.

```
generic
 type Data is private;
package Recovery_Cache is
  procedure Save(D : in Data);
  procedure Restore(D : out Data);
end Recovery_Cache;
```

Consider three Ada tasks which wish to enter into a recoverable atomic action. Each will call their appropriate procedure in the package given below:

```
package Conversation is

  procedure T1(Params : Param); -- called by task 1
  procedure T2(Params : Param); -- called by task 2
  procedure T3(Params : Param); -- called by task 3

  Atomic_Action_Failure : exception;

end Conversation;
```

The body of the package encapsulates the action and ensures that only communication between the three tasks is allowed during the conversation.[2] The `Controller` protected object is responsible for propagating any error condition noticed in one task to all tasks, saving and restoring any persistent data in the recovery cache, and ensuring that all tasks leave the action at the same time. It contains three protected entries and a protected procedure.

- The `Wait_Abort` entry represents the asynchronous event on which the tasks will wait while performing their part of the action.
- Each task calls `Done` if it has finished its component of the action without error. Only when all three tasks have called `Done` will they be allowed to leave.
- Similarly, each task calls `Cleanup` if it has had to perform any recovery.
- If any task recognizes an error condition (either because of a raised exception or the failure of the acceptance test), it will call `Signal_Abort`. This will set the flag `Killed` to `True`, indicating that the tasks must be recovered.

Note that, as backward error recovery will be performed, the tasks are not concerned with the actual cause of the error. When `Killed` becomes `True`, all tasks in the action receive the asynchronous event. Once the event has been handled, all tasks must wait on the `Cleanup` entry so that they can all terminate the conversation module together.

[2] In practice, this might be difficult to ensure because of Ada's scope rules. One way of increasing the security would be to require that the `Conversation` package is at the library level and its body only references pure (state-free) packages. The solution presented here assumes that the tasks are well behaved. It also assumes, for simplicity, that the correct tasks call `T1`, `T2` and `T3` at the correct times.

```
with Recovery_Cache;
package body Conversation is

  Primary_Failure, Secondary_Failure,
         Tertiary_Failure: exception;
  type Module is (Primary, Secondary, Tertiary);

  protected Controller is
    entry Wait_Abort;
    entry Done;
    entry Cleanup;
    procedure Signal_Abort;
  private
    Killed : Boolean := False;
    Releasing_Done : Boolean := False;
    Releasing_Cleanup : Boolean := False;
    Informed : Integer := 0;
  end Controller;

  -- any local protected objects for communication between actions

  protected body Controller is
    entry Wait_Abort when Killed is
    begin
      Informed := Informed + 1;
      if Informed = 3 then
        Killed := False;
        Informed := 0;
      end if;
    end Wait_Abort;

    procedure Signal_Abort is
    begin
      Killed := True;
    end Signal_Abort;

    entry Done when Done'Count = 3 or Releasing_Done is
    begin
      if Done'Count > 0 then
        Releasing_Done := True;
      else
        Releasing_Done := False;
      end if;
    end Done;

    entry Cleanup when Cleanup'Count = 3 or Releasing_Cleanup is
    begin
      if Cleanup'Count > 0 then
        Releasing_Cleanup := True;
      else
        Releasing_Cleanup := False;
      end if;
    end Cleanup;
  end Controller;
```

```
  procedure T1(Params : Param) is separate;
  procedure T2(Params : Param) is separate;
  procedure T3(Params : Param) is separate;

end Conversation;
```

The code for each task is contained within a single procedure: e.g. `T1`. Within such a procedure, three attempts are made to perform the action. If all attempts fail, the exception `Atomic_Action_Failure` is raised. Each attempt is surrounded by a call that saves the state and restores the state (if the attempt fails). Each attempt is encapsulated in a separate local procedure (`T1_Primary`, etc.), which contains a single 'select and then abort' statement to perform the required protocol with the controller. The recovery cache is used by each task to save its local data.

```
separate(Conversation)
procedure T1(Params : Param) is
  procedure T1_Primary is
  begin
    select
      Controller.Wait_Abort; -- triggering event
      Controller.Cleanup; -- wait for all to finish
      raise Primary_Failure;
    then abort
      begin
        -- code to implement atomic action,
        -- the acceptance test might raise an exception
        if Accept_Test = Failed then
          Controller.Signal_Abort;
        else
          Controller.Done; -- signal completion
        end if;
      exception
        when others =>
          Controller.Signal_Abort;
      end;
    end select;
  end T1_Primary;

  procedure T1_Secondary is ... ;
  procedure T1_Tertiary is ... ;

  package My_Cache is new Recovery_Cache(..); -- for local data

begin
  My_Cache.Save(..);
  for Try in Module loop
    begin
      case Try is
        when Primary => T1_Primary; return;
        when Secondary => T1_Secondary; return;
        when Tertiary => T1_Tertiary;
      end case;
```

```
    exception
      when Primary_Failure =>
        My_Cache.Restore(..);
      when Secondary_Failure =>
        My_Cache.Restore(..);
      when Tertiary_Failure =>
        My_Cache.Restore(..);
        raise Atomic_Action_Failure;
      when others =>
        My_Cache.Restore(..);
        raise Atomic_Action_Failure;
    end;
  end loop;
end T1;

-- similarly for T2 and T3
```

Figure 7.5 illustrates a simple state transition diagram for a participating task in a conversation.

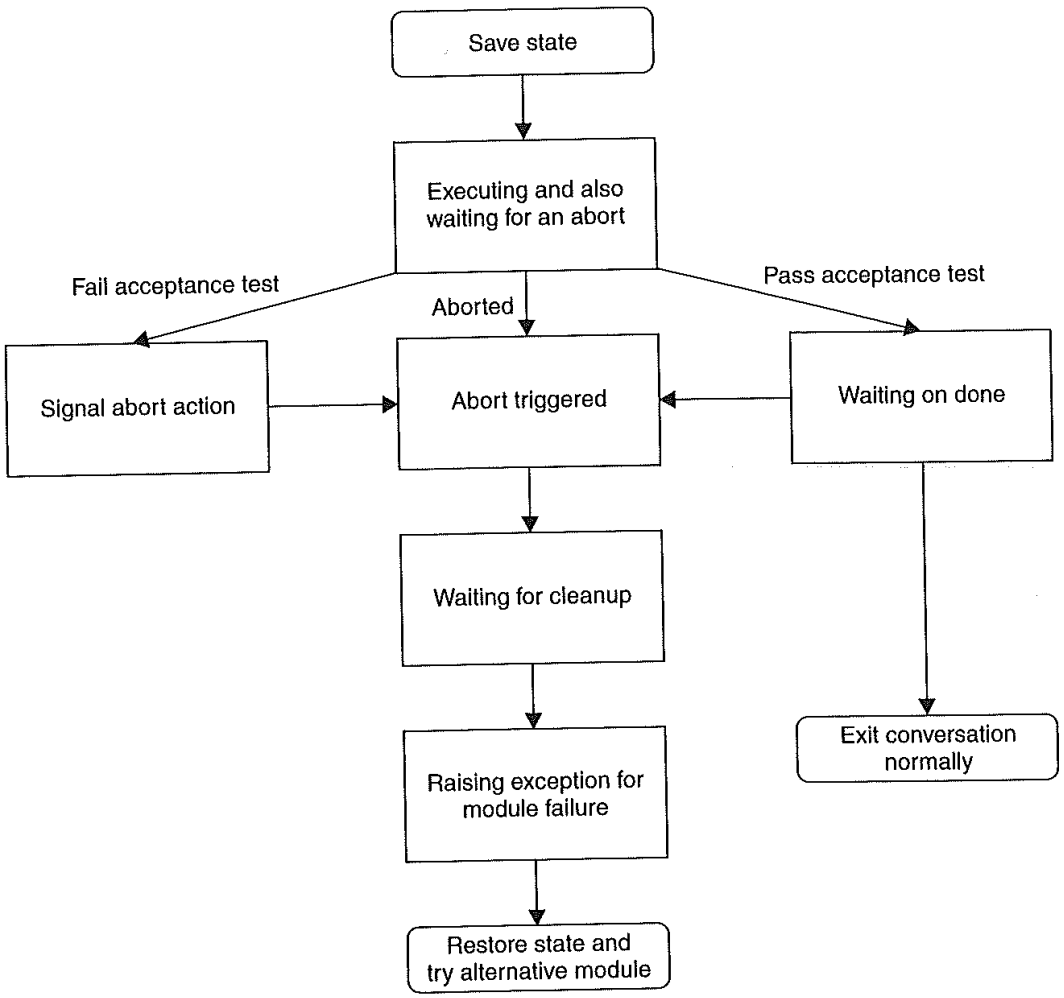

Figure 7.5 Simple state transition diagram for a conversation.

Forward error recovery

Ada's ATC facility can be used with exceptions to implement atomic actions with forward error recovery between concurrently executing tasks. Consider again the following package for implementing an atomic action between three tasks:

```
package Action is
  procedure T1(Params : Param); -- called by task 1
  procedure T2(Params : Param); -- called by task 2
  procedure T3(Params : Param); -- called by task 3

  Atomic_Action_Failure : exception;
end Action;
```

As with backward error recovery, the body of the package encapsulates the action and ensures that only communications between the three tasks are allowed. The `Controller` protected object is responsible for propagating any exception raised in one task to all tasks, and for ensuring that all tasks leave the action at the same time.

```
with Ada.Exceptions;
use Ada.Exceptions;
package body Action is
  type Vote_T is (Commit, Aborted);
  protected Controller is
    entry Wait_Abort(E: out Exception_Id);
    entry Done;
    procedure Cleanup (Vote: Vote_T);
    entry Wait_Cleanup(Result : out Vote_T);
    procedure Signal_Abort(E: Exception_Id);
  private
    Killed : Boolean := False;
    Releasing_Cleanup : Boolean := False;
    Releasing_Done : Boolean := False;
    Reason : Exception_Id;
    Final_Result : Vote_T := Commit;
    Informed : Integer := 0;
  end Controller;

  -- any local protected objects for communication between actions

  protected body Controller is
    entry Wait_Abort(E: out Exception_Id) when Killed is
    begin
      E := Reason;
      Informed := Informed + 1;
      if Informed = 3 then
        Killed := False;
        Informed := 0;
      end if;
    end Wait_Abort;

    entry Done when Done'Count = 3 or Releasing_Done is
    begin
```

```
    if Done'Count > 0 then
      Releasing_Done := True;
    else
      Releasing_Done := False;
    end if;
  end Done;

  procedure Cleanup (Vote: Vote_T) is
  begin
    if Vote = Aborted then
      Final_Result := Aborted;
    end if;
  end Cleanup;

  procedure Signal_Abort(E: Exception_Id) is
  begin
    Killed := True;
    Reason := E;
  end Signal_Abort;

  entry Wait_Cleanup (Result: out Vote_T)
        when Wait_Cleanup'Count = 3 or Releasing_Cleanup is
  begin
    Result := Final_Result;
    if Wait_Cleanup'Count > 0 then
      Releasing_Cleanup := True;
    else
      Releasing_Cleanup := False;
      Final_Result := Commit;
    end if;
  end Wait_Cleanup;
end Controller;

procedure T1(Params: Param) is
  X : Exception_Id;
  Decision : Vote_T;
begin
  select
    Controller.Wait_Abort(X); -- triggering event
    Raise_Exception(X); -- raise common exception
  then abort
    begin
      -- code to implement atomic action
      Controller.Done; --signal completion
    exception
      when E: others =>
        Controller.Signal_Abort(Exception_Identity(E));
    end;
  end select;
exception
    -- if any exception is raised during the action
    -- all tasks must participate in the recovery
    when E: others =>
      -- Exception_Identity(E) has been raised in all tasks
```

```
        -- handle exception
        if Handled_Ok then
          Controller.Cleanup(Commit);
        else
          Controller.Cleanup(Aborted);
        end if;
        Controller.Wait_Cleanup(Decision);
        if Decision = Aborted then
          raise Atomic_Action_Failure;
        end if;
    end T1;

    procedure T2(Params : Param) is ...;

    procedure T3(Params : Param) is ...;
end Action;
```

Each component of the action (`T1`, `T2` and `T3`) has identical structure. The component executes a select statement with an abortable part. The triggering event is signalled by the `Controller` protected object if any component indicates that an exception has been raised and not handled locally in one of the components. The abortable part contains the actual code of the component. If this code executes without incident, the `Controller` is informed that this component is ready to commit the action. If any exceptions are raised during the abortable part, the `Controller` is informed and the identity of the exception passed. Note that, unlike backward error recovery (given earlier), here the cause of the error must be communicated.

If the `Controller` has received notification of an unhandled exception, it releases all tasks waiting on the `Wait_Abort` triggering event (any task late in arriving will receive the event immediately it tries to enter into its select statement). The tasks have their abortable parts aborted (if started), and the exception is raised in each task by the statement after the entry call to the controller. If the exception is successfully handled by the component, the task indicates that it is prepared to commit the action. If not, then it indicates that the action must be aborted. If any task indicates that the action is to be aborted, then all tasks will raise the exception `Atomic_Action_Failure`. Figure 7.6 shows the approach using a simple state transition diagram.

The above example illustrates that it is possible to program atomic actions with forward error recovery in Ada. However, there are two points to note about this example.

- Only the first exception to be passed to the `Controller` will be raised in all tasks. It is not possible to get concurrent raising of exceptions, as any exception raised in an abortable part when it is aborted is lost.
- The approach does not deal with the deserter problem. If one of the participants in the action does not arrive, the others are left waiting at the end of the action. To cope with this situation, it is necessary for each task to log its arrival with the action controller (see Exercise 7.7).

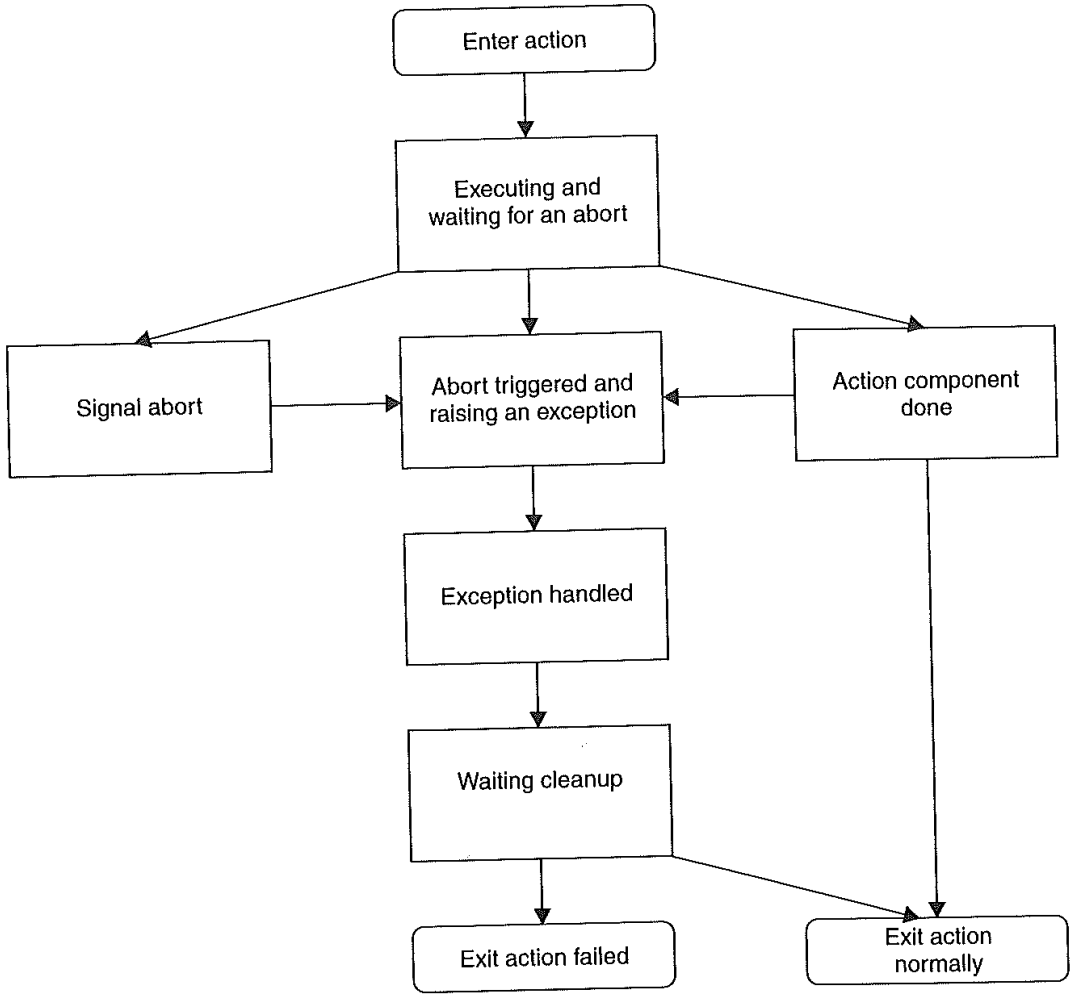

Figure 7.6 Simple state transition diagram illustrating forward error recovery.

7.7 Asynchronous notification in Real-Time Java

Real-Time Java supports both the resumption and termination models of asynchronous notification, but provides no direct support for asynchronous thread termination. The resumption model is provided by an asynchronous event-handling facility, where the handlers are scheduled entities rather than executed in the context of an interrupted thread. The termination model is provided within the context of an asynchronous exception-handling model for real-time threads. The latter can be used to program safe asynchronous termination. Note that facilities discussed apply only to Java systems that support the Real-Time Specification for Java; they cannot be used in a normal Java environment.

7.7.1 Asynchronous event handling

The equivalent of a C/Real-Time POSIX signal in Real-Time Java is an asynchronous event. Indeed, there is even a class `POSIXSignalHandler` which allows POSIX signals to be mapped onto Real-Time Java events for the occasion when Real-Time Java is being implemented on top of a POSIX-compliant operating system (see Section 14.4.2).

Program 7.4 shows the three main classes associated with asynchronous events in Real-Time Java. Each `AsyncEvent` can have one or more `AsyncEventHandlers`.

Program 7.4 The `AsyncEvent`, `AsyncEventHandler` and `BoundAsyncEventHandler` classes.

```
public class AsyncEvent {
  public AsyncEvent();

  public void addHandler(AsyncEventHandler handler);
  public void removeHandler(AsyncEventHandler handler);
  public void setHandler(AsyncEventHandler handler);
  // Associate a new handler with this event,
  // removing all existing handlers.

  public void bindTo(java.lang.String happening);
  // bind to external event
  public void unbindTo(java.lang.String happening);

  public ReleaseParameters createReleaseParameters();
  // creates a ReleaseParameters object representing the
  // characteristics of this event

  public void fire();
  // Schedule for execution the set of handlers for this event.
  public boolean handledBy(AsyncEventHandler handler);
  // Returns true if this event is handled by this handler.
}

public abstract class AsyncEventHandler implements Schedulable {
  public AsyncEventHandler();
  // parameters are inherited from the current thread

  public AsyncEventHandler(SchedulingParameters scheduling,
         ReleaseParameters release, MemoryParameters memory,
         MemoryArea area, ProcessingGroupParameters group);
  ... // other constructors available

  // methods which implement the Schedulable interface,
  // see Chapter 12

  protected int getAndClearPendingFireCount();
  // Atomically set to zero the number of pending executions
  // of this handler and returns the value from before
  // it was cleared.
  protected int getAndDecrementPendingFireCount();
  protected int getAndIncrementPendingFireCount();

  public void handleAsyncEvent();
  // Override this method to define the action to be
  // taken by this handler

  public final void run();
}

public abstract class BoundAsyncEventHandler extends AsyncEventHandler {
  public BoundAsyncEventHandler();
  // other constructors
}
```

When the event occurs (indicated by a call to the `fire` method), all the handlers associated with the event are scheduled for execution according to their `SchedulingParameters` – see Section 12.7. Note that the firing of an event can also be associated with the occurrence of an implementation-dependent external action by using the `bindTo` method.

Each handler is scheduled once for each outstanding event firing. Note, however, that the handler can modify the number of outstanding events by the methods in the `AsyncEventHandler` class.

Although an event handler is a schedulable entity, the goal is that it will not suffer the same overhead as an application thread. Consequently, it cannot be assumed that there is a separate implementation thread for each handler, as more than one handler may be associated with a particular implementation thread. If a dedicated thread is required, the `BoundAsyncEventHandler` should be used.

7.7.2 Asynchronous transfer of control in Real-Time Java

Early versions of Java allowed one thread to asynchronously affect another thread; however, this support has now been deprecated. Standard Java now only supports the following:

```
public void interrupt() throws SecurityException;
public boolean isInterrupted();
```

One thread can signal an interrupt to another thread by calling the `interrupt` method. The result of this depends on the current status of the interrupted thread.

- If the interrupted thread is blocked in the `wait`, `sleep` or `join` methods, it is made runnable and the `InterruptedException` is thrown.
- If the interrupted thread is executing, a flag is set indicating that an interrupt is outstanding. *There is no immediate effect on the interrupted thread.* Instead, the called thread must periodically test to see if it has been 'interrupted' using the `isInterrupted` method.

By itself, this does not meet the user needs outlined in Section 7.4.1.

Real-Time Java provides an alternative approach for interrupting a thread, based on asynchronous transfer of control (ATC). The Real-Time Java ATC model is similar to the Ada one in that it is necessary to indicate which regions of code can receive the ATC request. However, the Real-Time Java model is different in two important respects.

(1) The Real-Time Java model is integrated with the Java exception-handling facility, whereas the Ada model is integrated into the select statement and entry-handling mechanisms.

(2) The Real-Time Java model requires each method to indicate that it is prepared to allow the ATC to occur. ATC is deferred until the thread is executing within such a method. In contrast, Ada's default is to allow the ATC if a subprogram has been called from within the select-then-abort statement; a deferred response must be explicitly handled.

Both languages defer the ATC during interaction with other threads/tasks (for example synchronized statements/methods in Java and protected actions and rendezvous in Ada) or within constructors and finalization (finally) clauses.

The Real-Time Java ATC model brings together the Java exception-handling model and an extension of thread interruption. Essentially the model is that when a real-time thread (or more generally a schedulable object – see Section 12.7) is interrupted, an asynchronous exception (`AsynchronouslyInterruptedException`) is delivered to the real-time thread rather than the thread having to poll for the interruption as would be the case with conventional Java. `AsynchronouslyInterruptedException` is a checked exception.

The notion of an asynchronous exception is not new and has been explored in previous languages. The main problem with them is how to program safely in their presence. As discussed in Chapter 3, most exception-handling mechanisms have exception propagation within a termination model. Consider a thread that has called method A, which has called method B, which has called method C. When an exception is raised within method C, if there is no local handler, the call to method C is terminated and a handler is sought in method B (the exception propagates up the call chain). If no handler is found in B, the exception is propagated to A. When a handler is found, it is executed, and the program continues to execute in the context in which the handler was found. There is no return to the context where the original exception was thrown. This model makes it difficult to write code that is tolerant of an asynchronous exception being thrown at it. Every method would need a handler for the root class of all asynchronous exceptions.

The Real-Time Java solution to this problem is to require that all methods (including constructors) that are prepared to allow the delivery of an asynchronous exception, place the exception in their `throws` lists; Real-Time Java calls such methods **AI-methods** (Asynchronously Interruptible). If a method does not do this, then the asynchronous exception is not delivered but held pending until the real-time thread is in a method that has the asynchronous exception in its throw clause. Hence, code that has been written without being concerned with ATC can execute safely even in an environment where ATCs are being used. Furthermore, to ensure that ATCs can be handled safely, Real-Time Java requires that:

(1) ATCs are deferred during the execution of synchronized methods or statements and static initializers. This is to ensure that any shared data is left in a consistent state; Real-Time Java calls these sections of code and the methods that are not AI methods collectively **ATC-deferred** sections.

(2) An ATC can only be handled from within code that is an ATC-deferred section; this is to avoid the handler for one ATC being interrupted by another ATC being delivered.

The full model is best explained in two stages. The first is the low-level support and overall approach; the second is the use of the high-level support to provide a structured means for handling ATC. Use of the basic ATC facilities requires three activities:

(1) declaring an `AsynchronouslyInterruptedException` (AIE);

(2) identifying methods which can be interrupted;

(3) signalling an `AsynchronouslyInterruptedException` to a thread.

Program 7.5 The Real-Time Java `AsynchronouslyInterruptedException` class.

```
public class AsynchronouslyInterruptedException extends
             java.lang.InterruptedException
{
  public  AsynchronouslyInterruptedException();


  public boolean clear();
  public boolean disable();
  // only valid within a doInterruptible,
  // returns true if successful
  public boolean doInterruptible (Interruptible logic);
  // Only one specific Interruptible can be running per thread
  // at any one time.
  // Returns True, if the Interruptible is executed, false if one
  // is already in progress for this thread.

  public boolean enable();
  public boolean fire();

  public static AsynchronouslyInterruptedException getGeneric();
  // returns the AsynchronouslyInterruptedException which
  // is generated when RealtimeThread.interrupt() is invoked

  public boolean isEnabled();
}
```

Program 7.5 shows the specification of the `AsynchronouslyInterrupted-Exception` class. The methods will be explained in due course, but for now all that is required is to know that for each thread there is an associated generic AIE.

An AIE may be placed in the `throws` list associated with a method. For example, consider the following class which provides an interruptible service using a package which declares non-interruptible services (that is, ones which do not have `throws` lists containing `AsynchronouslyInterruptedExceptions`).

```
import nonInterruptibleServices.*;

public class InterruptibleService {
  public AsynchronouslyInterruptedException stopNow =
         AsynchronouslyInterruptedException.getGeneric();

  public boolean service()
        throws AsynchronouslyInterruptedException {
    //code interspersed with calls to NonInterruptibleServices
  }
}
```

Now assume that a real-time thread `rtThread` has called an instance of this class to provide the `service`:

```
public InterruptibleService IS = new InterruptibleService();

// code of thread, rtThread
if(IS.service()) { ... } else { ... };
```

and that another real-time thread interrupts `rtThread`:

```
rtThread.interrupt();
```

The consequences of this call depend on the current state of `rtThread` when the call is made.

- If `rtThread` is executing within an ATC-deferred section – that is, executing within a synchronized method (or block), a static initializer or within a method that has no `AsynchronouslyInterruptedException` declared in its `throws` list (such as those in the package `nonInterruptibleServices`) – the AIE is marked as pending. The exception is delivered as soon as `rtThread` leaves the ATC-deferred region and is executing in a method with an `AsynchronouslyInterruptedException` declared in its `throws` list (such as the `service` method).
- If `rtThread` is executing within an AI-method (and it is not within a synchronized block), then the method's execution is interrupted, and control is transferred (propagated) up the call chain until it finds a try block in an ATC-deferred region that has a catch clause naming `AsynchronouslyInterruptedException` (or a parent class). Any synchronized methods or statements that are terminated by this propagation have their monitor locks released (and their `finally` clauses executed). The handler is then executed.
- If `rtThread` is blocked inside a `sleep`, `join`, `MemoryArea.join` (or `joinAndEnter`) or `waitForNextPeriodInterruptible` method called from within an AI-method, `rtThread` is rescheduled and the `AsynchronouslyInterruptedException` is delivered.
- If `rtThread` is blocked inside a `wait`, `sleep`, `join`, `MemoryArea.join` (or `joinAndEnter`) or `waitForNextPeriodInterruptible` method called from within an ATC-deferred region, `rtThread` is rescheduled and the `AsynchronouslyInterruptedException` is thrown as a synchronous exception (it is a subclass of the `InterruptedException`) and it is also marked as pending. Even if the synchronous exception is handled, the asynchronous exception is redelivered as soon as `rtThread` enters an AI-method.

Once an ATC has been delivered and control is passed to an appropriate exception handler, it is necessary to ascertain whether the caught ATC is the one expected by the interrupted thread. If it is, the exception can be handled. The `clear` method defined in the class `AsynchronouslyInterruptedException` is used for this purpose. The `clear` method simply resets the pending state if the AIE is current and returns

`true`, indicating that the current AIE is now no longer pending. If the flag is not reset, the AIE will then be redelivered when control next enters an AI-method.

```
import NonInterruptibleServices.*;
public class InterruptibleService {
  public AsynchronouslyInterruptedException stopNow =
          new AsynchronouslyInterruptedException();

  public boolean Service()
       throws AsynchronouslyInterruptedException {
    try {
      // code interdispersed with calls to
      // NonInterruptibleServices
    }
    catch AsynchronouslyInterruptedException AIE {
      if(stopNow.clear()) {
        // handle the ATC
       } else {
        // Cleanup and leave the current AIE still pending.
       }
     }
   }
}
```

Here, when the AIE is thrown, control is passed to the catch clause at the end of the try block. A handler is found for `AsynchronouslyInterruptedExceptions`. In order to determine whether the current `AsynchronouslyInterruptedException` is `stopNow`, a call is made to the `clear` method. This returns `true` if `stopNow` is the current exception. If it is not the current exception then, the real-time thread does some cleanup routines before exiting the handler. In the latter case, the AIE is still pending.

The `Interruptible` interface

The above discussion illustrates the basic mechanisms provided by Real-Time Java for handling ATCs. To facilitate their structured use, the language also provides an interface called `Interruptible` – see Program 7.6.

An object which wishes to provide an interruptible method does so by implementing the `Interruptible` interface. The `run` method is the method that is interruptible; the `interruptAction` method is called by the system if the `run` method is interrupted.

Program 7.6 The Real-Time Java `Interruptible` interface.

```
public interface Interruptible {
  public void interruptAction (
          AsynchronouslyInterruptedException exception);

  public void run (AsynchronouslyInterruptedException exception)
       throws AsynchronouslyInterruptedException;
}
```

Once this interface is implemented, the implementation can be passed as a parameter to the `doInterruptible` method in the `AsynchronouslyInterruptedException` class. The method can then be interrupted by calling the `fire` method in the `AsynchronouslyInterruptedException` class. Further control over `AsynchronouslyInterruptedException` is given by the `disable`, `enable` and `isEnabled` methods. A disabled `AsynchronouslyInterruptedException` is deferred until it is enabled. An example of this latter use will be given in Section 7.7.3.

Note that only one `doInterruptible` method for a particular `AsynchronouslyInterruptedException` can be active at one time. If a call is outstanding, the method returns immediately with a `false` value.

Multiple `AsynchronouslyInterruptedExceptions`

Given that `AsynchronouslyInterruptedException` can be deferred, it is possible for multiple ATCs to be deferred. This can happen when the `run` method of one class (which implements the `Interruptible` interface) calls a `doInterruptible` on an AIE. The associated `run` method may also call another `doInterruptible`. Hence it is possible for a thread to be executing nested `doInterruptibles`. Consider the following example:

```
import javax.realtime.*;

public class NestedATC {
  AsynchronouslyInterruptedException AIE1 = new
        AsynchronouslyInterruptedException();
  AsynchronouslyInterruptedException AIE2 = new
        AsynchronouslyInterruptedException();
  AsynchronouslyInterruptedException AIE3 = new
        AsynchronouslyInterruptedException();

 public void method1() {
   // ATC-deferred region
 }

 public void method2() throws AsynchronouslyInterruptedException {
    AIE1.doInterruptible
       (new Interruptible()
         {
            public void run(AsynchronouslyInterruptedException e)
                 throws AsynchronouslyInterruptedException
            {
              method1();
            }
            public void interruptAction(
                 AsynchronouslyInterruptedException e)
            {
               if(AIE1.clear()) {
                 // recovery here
               } else {
                // cleanup
              }
```

```
            }
          }
        );
    }

    public void method3() throws AsynchronouslyInterruptedException {
       AIE2.doInterruptible
          (new Interruptible()
            {
              public void run(AsynchronouslyInterruptedException e)
                    throws AsynchronouslyInterruptedException {
                method2();
              }
              public void interruptAction(
                    AsynchronouslyInterruptedException e){
                 if(AIE2.clear()) {
                   // recovery here
                 } else {
                  // cleanup
                }
              }
            }
          );
    }

    public void method4() throws AsynchronouslyInterruptedException {
       AIE3.doInterruptible
          (new Interruptible()
            {
              public void run(AsynchronouslyInterruptedException e)
                    throws AsynchronouslyInterruptedException {
                method3();
              }
              public void interruptAction(
                    AsynchronouslyInterruptedException e) {
                 if(AIE3.clear()) {
                   // recovery here
                 } else {
                  // cleanup
                }
              }
            }
          );
      }

    }
```

Now suppose that a thread `t` has created an instance of `NestedATC` and has called `method4`, which has called `method3`, which has called `method2`, which has called `method1` which is an ATC-deferred region. Assume that the thread is interrupted by a call to `AIE2.fire()`. This is held pending. If `AIE3` now comes in then `AIE2` is discarded as `AIE3` is at a higher level in the nesting. If `AIE1` comes in then `AIE1` is discarded (because it is at a lower level). Once `method1` has returned, the currently pending AIE is thrown.

7.7.3 Real-Time Java and atomic actions

This section illustrates how Real-Time Java's ATC facilities can be used to implement atomic actions with forward error recovery.

First, an `AtomicActionException` is defined along with an `AtomicActionFailure` exception.

```
import javax.realtime.AsynchronouslyInterruptedException;

public class AtomicActionException extends
          AsynchronouslyInterruptedException {
  public static Exception cause;
  public static boolean wasInterrupted;
}

public class AtomicActionFailure extends Exception;
```

Using a `ThreeWayRecoverableAtomicAction` similar to that defined earlier:

```
public interface ThreeWayRecoverableAtomicAction {
  public void role1() throws AtomicActionFailure;
  public void role2() throws AtomicActionFailure;
  public void role3() throws AtomicActionFailure;
}
```

a `RecoverableAction` class can be implemented with a similar structure to that given for Ada.

```
import javax.realtime.*;

public class RecoverableAction
          implements ThreeWayRecoverableAtomicAction {
  protected RecoverableController Control;
  private final boolean abort = false;
  private final boolean commit = true;

  private AtomicActionException aae1, aae2, aae3;

  public RecoverableAction()  // constructor {
    Control = new RecoverableController();
      // for recovery
      aae1 = new AtomicActionException();
      aae2 = new AtomicActionException();
      aae3 = new AtomicActionException();
  }

  class RecoverableController {
    protected boolean firstHere, secondHere, thirdHere;
    protected int allDone;
    protected int toExit, needed;
    protected int numberOfParticipants;
    private boolean committed = commit;
```

```
RecoverableController() {
  // for synchronization
  firstHere = false;
  secondHere = false;
  thirdHere = false;
  allDone = 0;
  numberOfParticipants = 3;
  toExit = numberOfParticipants;
  needed = numberOfParticipants;
}

synchronized void first() throws InterruptedException {
  while(firstHere) wait();
  firstHere = true;
}

synchronized void second() throws InterruptedException {
  while(secondHere) wait();
  secondHere = true;
}

synchronized void third() throws InterruptedException {
  while(thirdHere) wait();
  thirdHere = true;
}

synchronized void signalAbort(Exception e) {
 allDone = 0;
 AtomicActionException.cause = e;
 AtomicActionException.wasInterrupted = true;
 // raise an AsynchronouslyInterruptedException
 // in all participants
 aae1.fire();
 aae2.fire();
 aae3.fire();
}

private void reset() {
  firstHere = false;
  secondHere = false;
  thirdHere = false;
  allDone = 0;
  toExit = numberOfParticipants;
  needed = numberOfParticipants;
  notifyAll();
}

synchronized void done() throws InterruptedException {
  allDone++;
  if(allDone == needed) {
    notifyAll();
  } else while(allDone != needed) {
      wait();
      if(AtomicActionException.wasInterrupted) {
        allDone--;
        return;
      }
```

```
      }
      toExit--;
       if(toExit == 0) {
         reset();
       }
    }

    synchronized void cleanup(boolean abort) {
      if(abort) { committed = false; };
    }

    synchronized boolean waitCleanup()
             throws InterruptedException {
      allDone++;
      if(allDone == needed) {
        notifyAll();
      } else while (allDone != needed) {
          wait();
      }
      toExit--;
      if(toExit == 0) {
        reset();
      }
      return committed;
    }
}

public void role1() throws AtomicActionFailure,
                    AsynchronouslyInterruptedException {
    boolean Ok;
    // entry protocol
    // no AIE until inside the atomic action
    boolean done = false;
    while(!done) {
      try {
        Control.first();
        done = true;
      } catch(InterruptedException e) {
        // ignore
      }
    }

    // the following defines an interruptible
    // section of code, and a routine to be called
    // if the code is interrupted
    Ok = aae1.doInterruptible
        (new Interruptible()
          {
            public void run(AsynchronouslyInterruptedException e)
                  throws AsynchronouslyInterruptedException {
              try {
               // perform action
               // if necessary call e.disable() and e.enable() to
               // defer AIE
               Control.done();
              }
              catch(Exception x) {
```

```
                Control.signalAbort(x);
              }
            }

            public void interruptAction(
                  AsynchronouslyInterruptedException e) {
              // no action required
            }
          }
        );
    if(!Ok) throw new AtomicActionFailure();

    if(aae1.wasInterrupted) {
      try {
        // try to recover
        Control.cleanup(commit);
        if(Control.waitCleanup() != commit) {
           throw new AtomicActionFailure();
        };
      }
      catch(Exception x) {
        throw new AtomicActionFailure();
      }
    };
  }

  public void role2() throws AtomicActionFailure,
                AsynchronouslyInterruptedException
  {/* similar to role1 */};

  public void role3() throws AtomicActionFailure,
                AsynchronouslyInterruptedException
  {/* similar to role1 */};

}
```

Note that the absence of direct language support for asynchronous transfer of control makes the Java solution appear more complicated than its Ada counterpart. This is mainly due to the requirement to construct `Runnable` objects to pass to the `doInterruptible` method.

Summary

Reliable execution of tasks is essential if real-time embedded systems are to be used in critical applications. When tasks interact, it is necessary to constrain their intertask communication so that recovery procedures can be programmed, if required. Atomic actions have been discussed in this chapter as a mechanism by which programs, consisting of many tasks, can be structured to facilitate damage confinement and error recovery.

Actions are atomic if they can be considered, so far as other tasks are concerned, to be indivisible and instantaneous, such that the effects on the

system are as if they are interleaved as opposed to concurrent. An atomic action has well-defined boundaries and can be nested. Resources used in an atomic action are allocated during an initial *growing phase*, and released either as part of a subsequent *shrinking phase* or at the end of the action (if the action is to be recoverable).

The syntax of an atomic action can be expressed by an action statement. The following statement executed within task P_1 indicates that P_1 wishes to enter into an atomic action with P_2 and P_3:

```
action A with (P2,P3) do
    -- sequence of statements
end A;
```

P_2 and P_3 must execute similar statements.

A *conversation* is an atomic action with backward error recovery facilities (in the form of recovery blocks).

```
action A with (P2,P3) do
   ensure <acceptance test>
   by
      -- primary module
   else by
      -- alternative module
   else error
end A;
```

On entry to the conversation, the state of the task is saved. While inside the conversation, a task is only allowed to communicate with other tasks active in the conversation and general resource managers. In order to leave the conversation, all tasks active in the conversation must have passed their acceptance test. If any task fails its acceptance test, all tasks have their state restored to that saved at the start of the conversation and they execute their alternative modules. Conversations can be nested and if all alternatives in an inner conversation fail then recovery must be performed at an outer level.

Forward error recovery via exception handlers can also be added to atomic actions. If an exception is raised by one task then all tasks active in the action must handle the exception.

```
action A with (P2,P3) do
   -- the action
exception
   when exception_a =>
       -- sequence of statements
   when others =>
      raise atomic_action_failure;
end A;
```

Two issues that must be addressed when using this approach are the resolution of concurrently raised exceptions and exceptions in internal actions.

Few mainstream languages or operating systems directly support the notion of an atomic action or a recoverable atomic action. However, most communication and synchronization primitives allow the isolation property of an action to be programmed. To implement a recoverable action requires an asynchronous notification mechanism. This can either have resumption semantics (in which case it is called an asynchronous event-handling mechanism) or it can have termination semantics (in which case it is called asynchronous transfer of control). C/Real-Time POSIX supports asynchronous events using signals and a thread-cancelling mechanism. A signal can be handled, blocked or ignored. Real-Time Java also supports asynchronous events.

Both Ada and Real-Time Java provide the termination model of asynchronous transfer of control. The Ada mechanism is built on top of the select statement. Real-Time Java's ATC, in contrast, is integrated into its exception and thread interrupt mechanisms. These termination approaches, in combination with exceptions, allow for an elegant implementation of a recoverable action.

Further reading

Anderson, T. and Lee, P. A. (1990) *Fault Tolerance Principles and Practice*, 2nd ed. London: Prentice Hall.

Burns, A. and Wellings, A. J. (2007) *Concurrent and Real-Time Programming in Ada*. Cambridge: Cambridge University Press.

Lea, D. (2000) *Concurrent Programming in Java: Design Principles and Patterns*, 2nd ed. Harlow: Prentice Hall.

Lynch, N. A. (ed.) (1993) *Atomic Transactions*, Morgan Kaufmann Series in Data Management Systems. San Mateo, California: Morgan Kaufmann.

Northcutt, J. D. (1987) *Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel*. Orlando: Academic Press.

Shrivastava, S. K., Mancini, L. and Randell, B. (1987) *On the Duality of Fault Tolerant Structures*, Lecture Notes in Computer Science, Volume 309, pp. 19–37. Springer-Verlag.

Wellings, A. J. (2004) *Concurrent and Real-Time Programming in Java*. Chichester: Wiley.

Exercises

7.1 Distinguish between an atomic *action* and an atomic *transaction*. What is the relationship between an atomic transaction and a conversation?

7.2 Rewrite the `Action_X` Ada package given in Section 7.6.3 so that it becomes a general-purpose package for controlling a three-task conversation. (Hint: use generics.)

7.3 Can your solution to Exercise 7.2 be extended to cope with an arbitrary number of tasks participating in the atomic action?

7.4 What would be the implications of extending Ada to enable one task to raise an exception in another?

7.5 Compare and contrast asynchronous notification and exception handling.

7.6 In Section 7.6.3, backward and forward error recovery is shown between three Ada tasks. Show how they can be combined into a single solution that gives both forward and backward error recovery between the same three tasks.

7.7 Update the solution given in Section 7.6.1 to deal with the deserter problem.

7.8 Consider the following four fragments of code:

```
-- Fragment 1
select
    T.Call; -- an entry call to task T
    Flag := A;
or
    delay 10.0;
    Flag := B;
    -- code taking 2 seconds to execute
end select;

-- Fragment 2
select
    T.Call; -- an entry call to task T
    Flag := A;
else
    delay 10.0;
    Flag := B;
    -- code taking 2 seconds to execute
end select;

-- Fragment 3
select
    T.Call; -- an entry call to task T
    Flag := A;
then abort
    delay 10.0;
    Flag := B;
    -- code taking 2 seconds to execute
end select;

-- Fragment 4
select
    delay 10.0;
    Flag := A;
then abort
    T.Call; -- an entry call to task T
    Flag := B;
    -- code taking 2 seconds to execute
end select;
```

A rendezvous with `T.Call` takes 5 seconds to execute. What is the value of the `Flag` variable after the execution of each of the four fragments in each of the following cases? You may assume a `Flag` assignment statement takes zero execution time.

(1) `T.Call` is available when the select is executed.
(2) `T.Call` is NOT available when the select is executed and does not become available in the next 14 seconds.
(3) `T.Call` is NOT available when the select is executed, but does become available after 2 seconds.
(4) `T.Call` is NOT available when the select is executed but does become available after 8 seconds.

7.9 Consider the following package specification which provides a procedure to search part of a large character array for a *unique* fixed-length string. The procedure returns the position of the start of the string if it is found.

```
package Search_Support is
  type Array_Bounds is range 1 .. 1_000_000_000;
  type Large_Array is array(Array_Bounds) of Character;
  type Pointer is access Large_Array;

  type Search_String is new String(1..10);

  procedure Search(Pt: Pointer;
                   Lower, Upper: Array_Bounds;
                   Looking_For : Search_String;
                   Found : out Boolean;
                   At_Location : out Array_Bounds);
end Search_Support;
```

Three tasks wish to perform a concurrent search of the array for the same string; they are derived from a common task type.

```
task type Searcher(Search_Array: Pointer;
                   Lower, Upper: Array_Bounds) is
  entry Find(Looking_For : Search_String);
  entry Get_Result(At_Location : out Array_Bounds);
end Searcher;
```

The string to be found is passed via an initial rendezvous with the tasks.
Sketch the body of the task type (and any other objects you might need) so that when one task finds the string, all other tasks are *immediately* informed of the string's location so that further fruitless search is avoided. Assume that the `SearchString` will be found by one of the three tasks. Furthermore, all tasks must be prepared to pass back the result via the `GetResult` entry.

7.10 Consider the following Ada code fragment:

```
Error_1, Error_2 : exception;
task Watch;
task Signaller;

protected Atc is
  entry Go;
  procedure Signal;
private
```

```
  Flag : Boolean := False;
end Atc;

protected body Atc is
  entry Go when Flag is
  begin
    raise Error_1;
  end Go;

  procedure Signal is
  begin
    Flag := True;
  end Signal;
end Atc;

task body Watch is
begin
  ...
  select
    Atc.Go;
  then abort
    -- code taking 100 millisecond
    raise Error_2;
  end select;
  ...
exception
  when Error_1 =>
    Put_Line("Error_1 Caught");
  when Error_2 =>
    Put_Line("Error_2 Caught");
  when others =>
    Put_Line("Other Errors Caught");
end Watch;


task body Signaller is
begin
  ...
  Atc.Signal;
  ...
end Signaller;
```

Describe carefully the possible executions of this program fragment assuming that context switches between tasks can happen at any time.

7.11 A particular POSIX-based application consists of several periodic processes and has two modes of operation: MODE A and MODE B. The application has one process which only operates in MODE A. Sketch the design of this process assuming that when the system wishes to undertake a mode change, it sends a signal to all processes indicating the current mode of operation. Assume also the existence of a routine called `WAITNEXTPERIOD`, which will suspend the process until its next period of execution is due. Note that a mode change should only affect the process at the beginning of each period.

7.12 Illustrate how Ada's OOP model can be used to produce extensible atomic actions.

7.13 Compare and contrast the Ada and Java models for asynchronous transfer of control.

7.14 To what extent can standard Java be used to implement atomic actions?

7.15 Why have the Java routines `resume()`, `stop()` and `suspend()` been made obsolete?

7.16 Redo Exercise 7.11 for Real-Time Java.

7.17 Show how Ada's termination model of exception handling can be implemented in response to the receipt of a POSIX signal.

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.

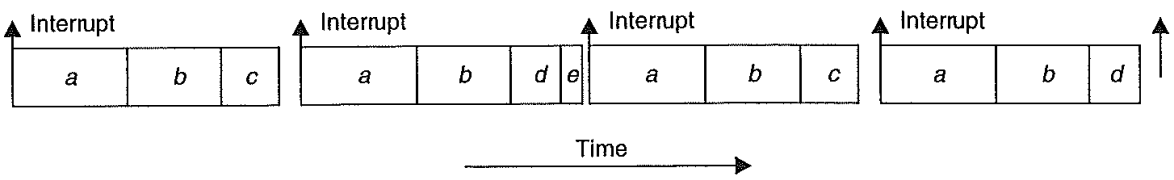

Figure 11.1 Time-line for task set.

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.

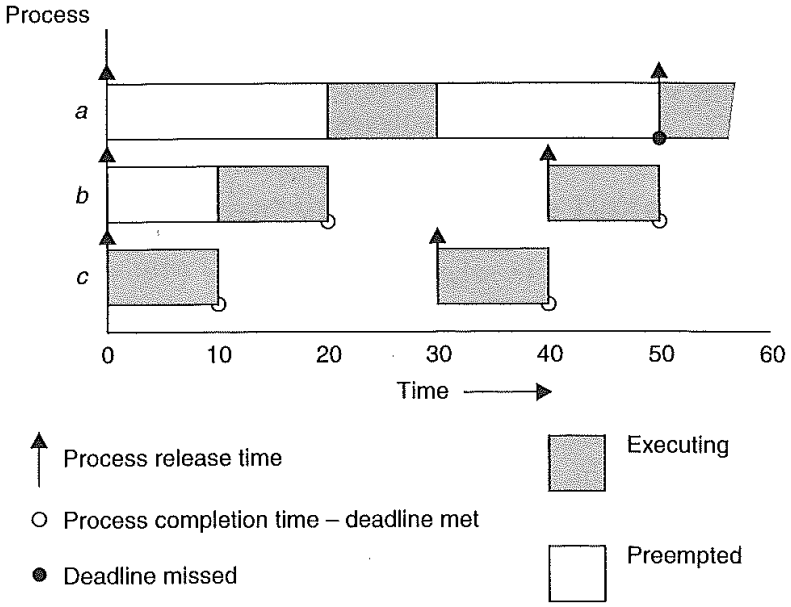

Figure 11.2 Time-line for task set A.

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.

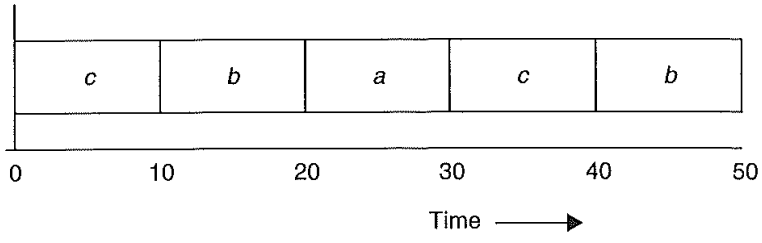

Figure 11.3 Gantt chart for task set A.

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

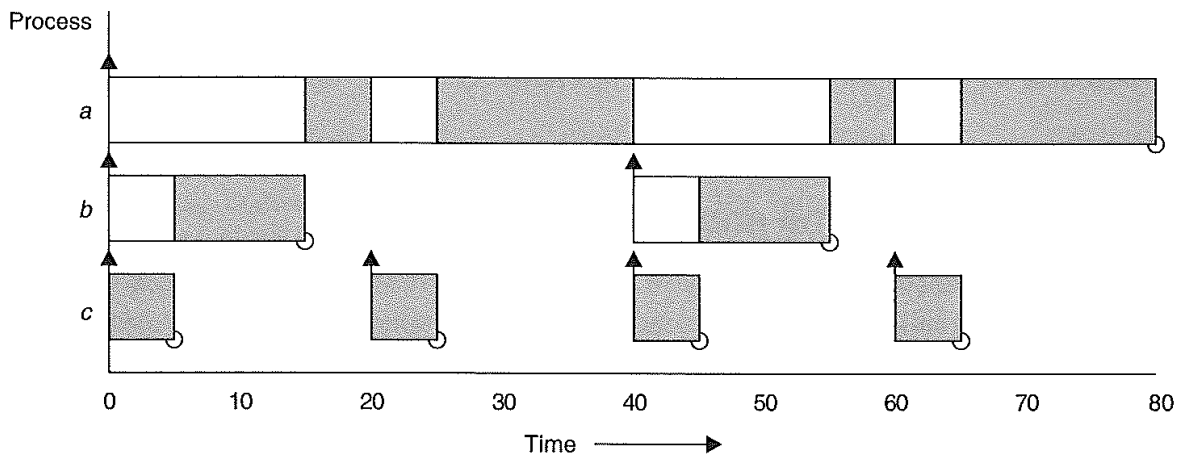

Figure 11.4 Time-line for task set C.

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

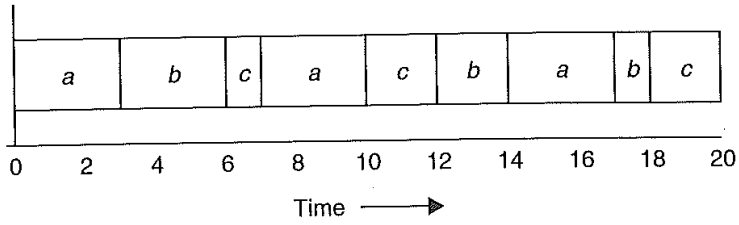

Figure 11.5 Gantt chart for task set D.

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.

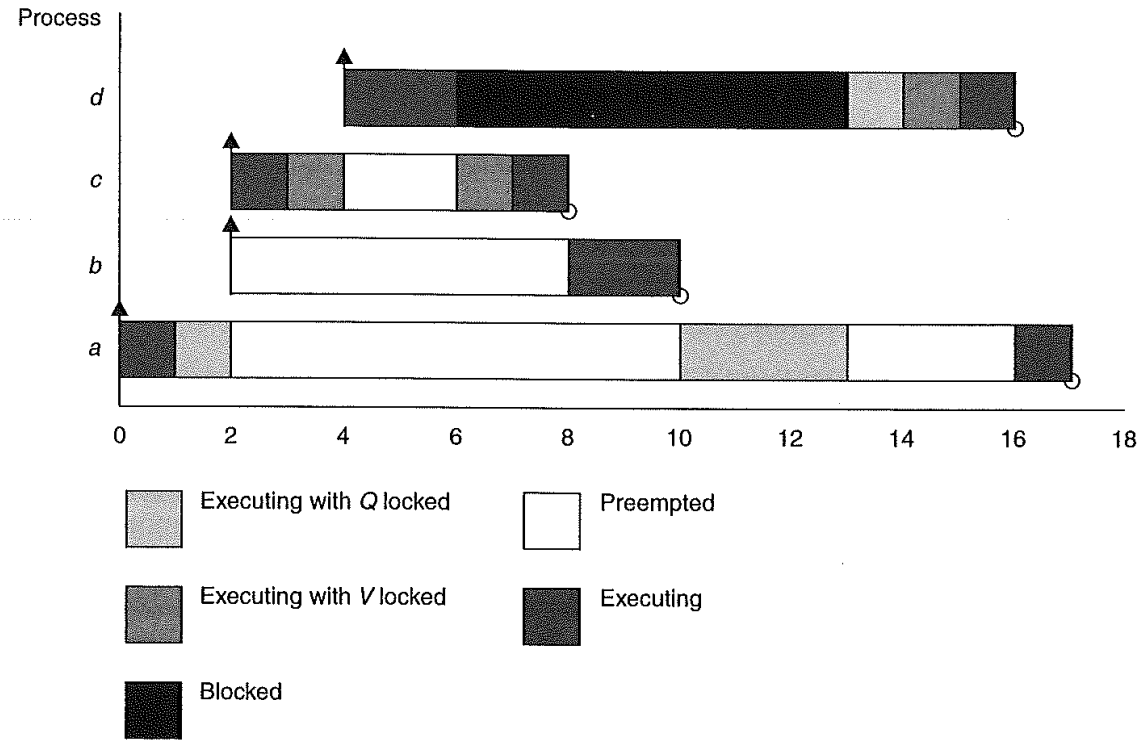

Figure 11.6 Example of priority inversion.

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.

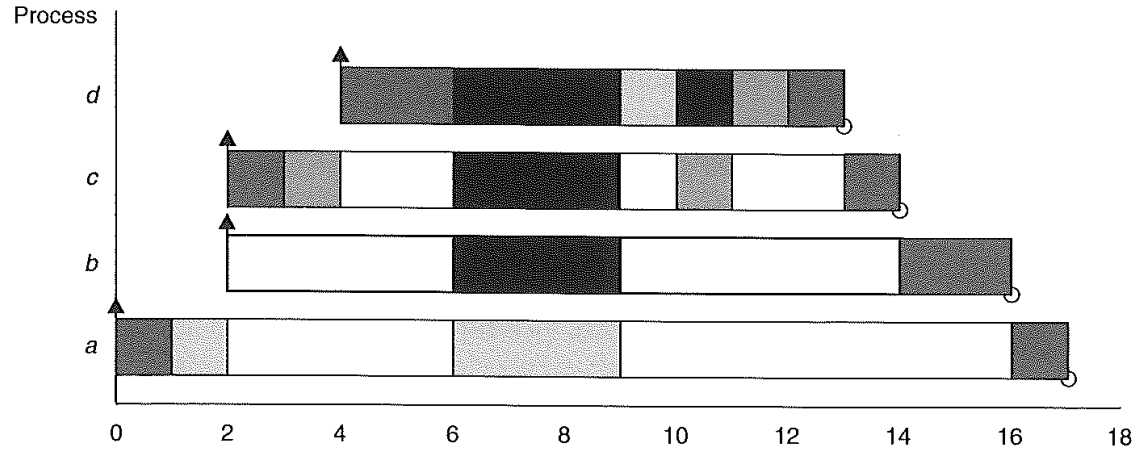

Figure 11.7 Example of priority inheritance.

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.

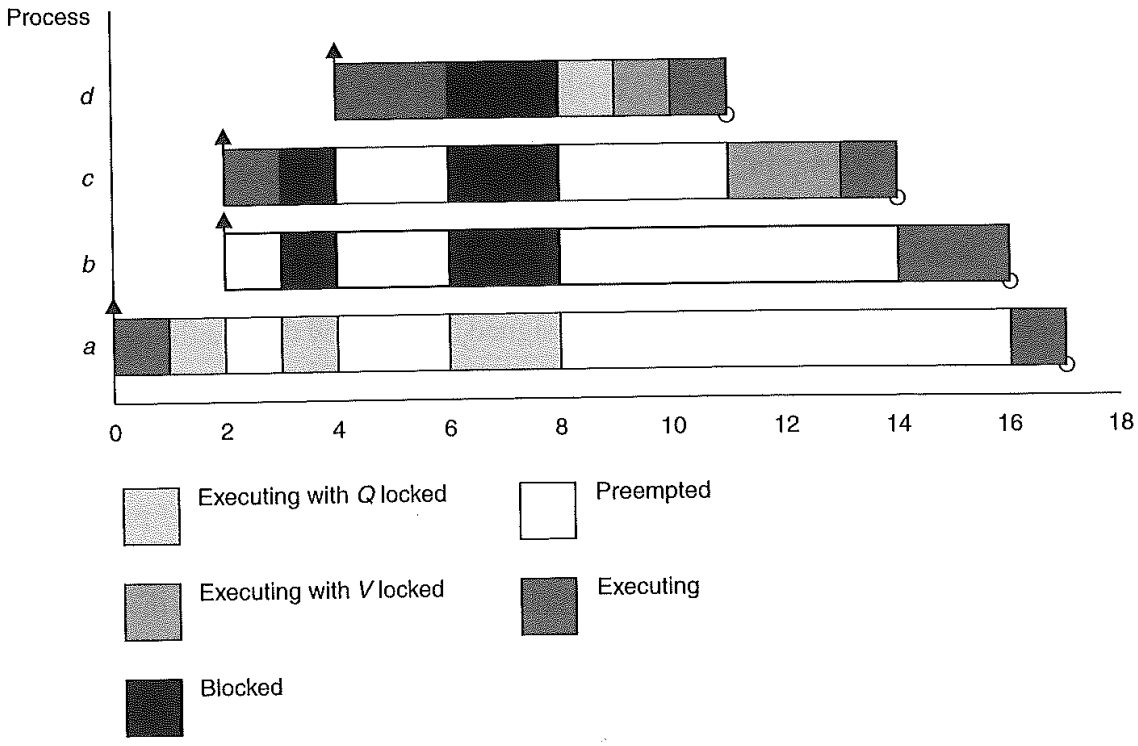

Figure 11.8 Example of priority inheritance – OCPP.

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)).

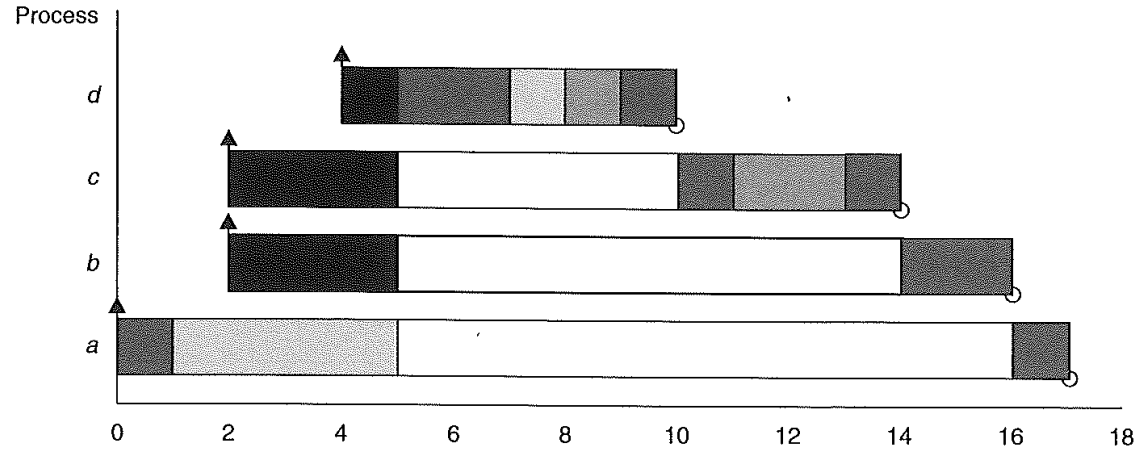

Figure 11.9 Example of priority inheritance – ICPP.

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 $T - J$,

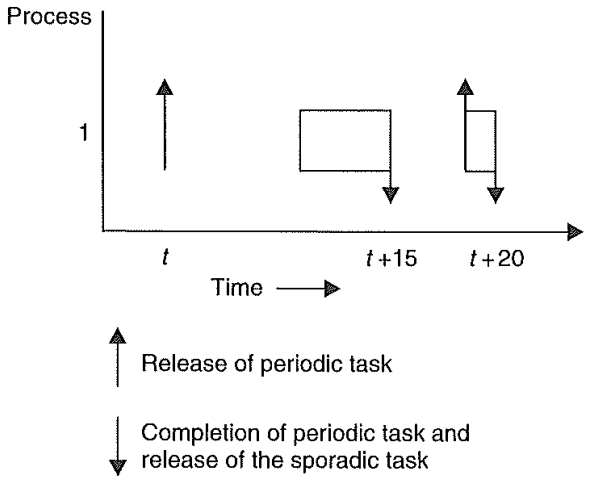

Figure 11.10 Releases of sporadic tasks.

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,

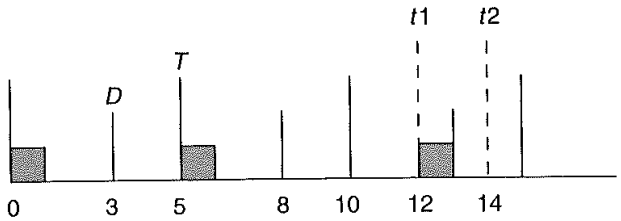

Figure 11.11 PDC example.

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).

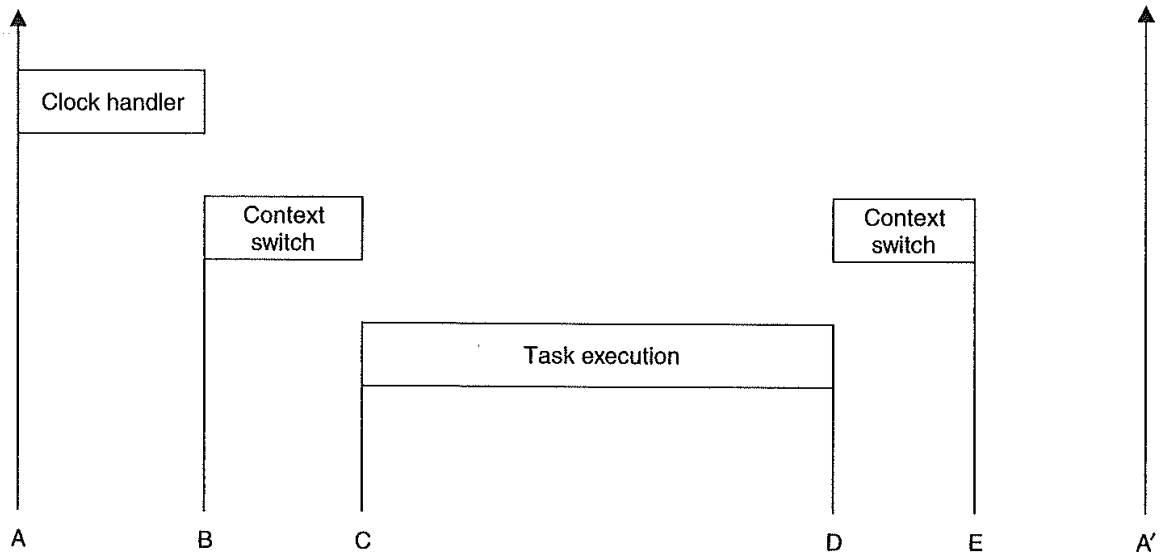

Figure 11.12 Overheads when executing tasks.

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.

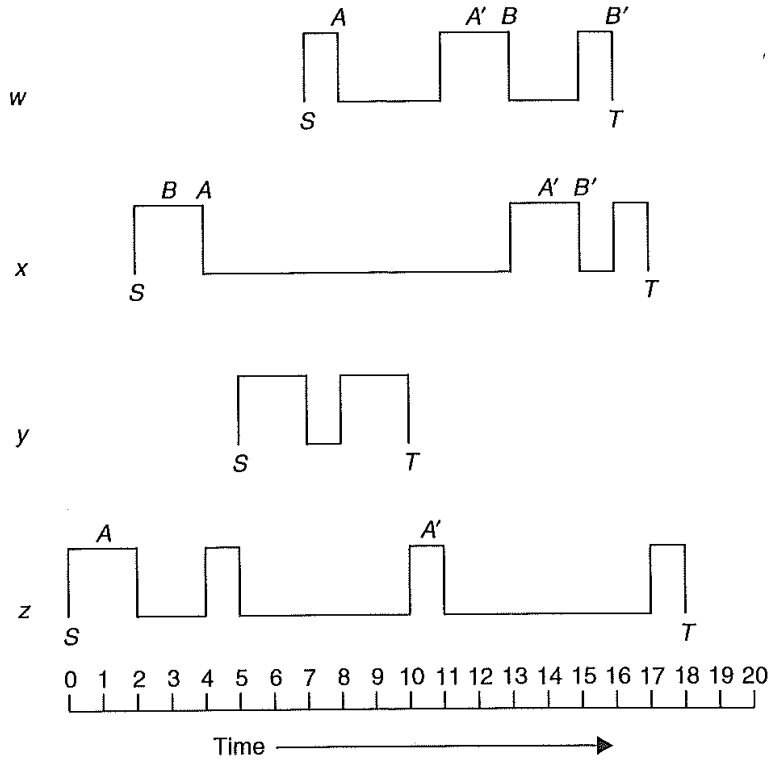

Figure 11.13 The behaviour of four periodic tasks in Exercise 11.4.

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?