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.

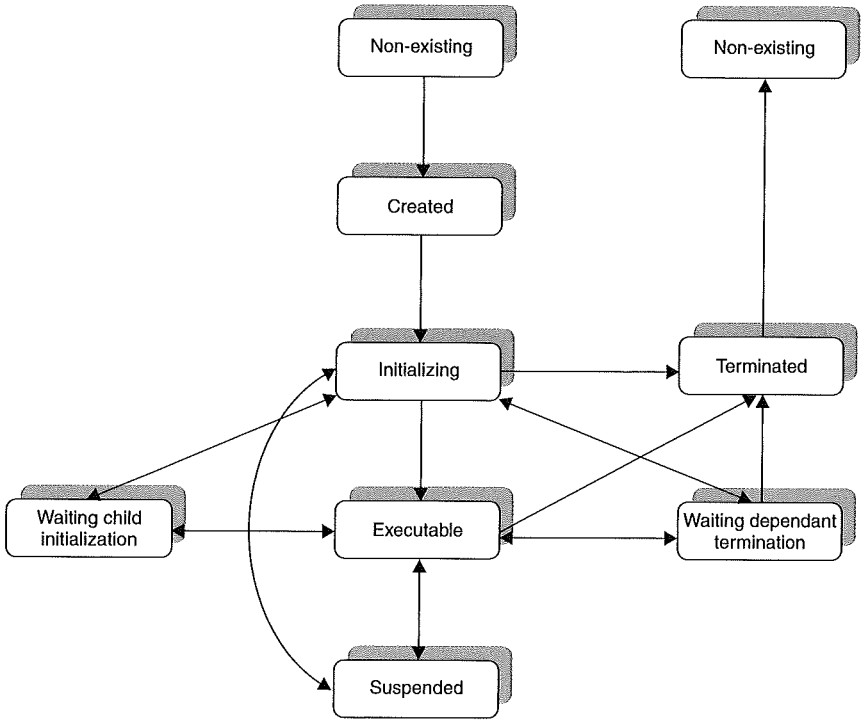

Figure 5.1 State diagram for a task.

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.