Chapter 4

Semaphores

4.1 Introduction

The algorithms in the previous chapter can be run on a *bare machine*. That is, they use only the machine language instructions that the computer provides. Though bare-machine instructions can be used to implement correct solutions to mutual exclusion and other concurrent programming problems, they are too low level to be efficient and reliable. In this chapter, we will study the *semaphore* which provides a concurrent programming primitive on a higher level than machine instructions. Semaphores are usually implemented by an underlying operating system, but for now we will investigate them by defining the required behavior and assuming that this behavior can be efficiently implemented.

A semaphore is an integer-valued variable which can take only non-negative values. Exactly two operations are defined on a semaphore S:

Wait(S) If $S > 0$ then $S := S - 1$ else suspend the execution of this process. The process is said to be suspended *on* the semaphore S.

Signal(S) If there are processes that have been suspended on this semaphore, wake one of them else $S := S + 1$.

The semaphore has the following properties:

1. `Wait(S)` and `Signal(S)` are atomic instructions.[1] In particular, no instructions can be interleaved between the test that $S > 0$ and the decrement of S or the suspension of the calling process.
2. A semaphore must be given a non-negative initial value.
3. The `Signal(S)` operation must waken one of the suspended processes. The definition does *not* specify which process will be awakened.

A semaphore which can take any non-negative value is called a *general semaphore*. A semaphore which takes only the values 0 and 1 is called a *binary semaphore* in which case `Signal(S)` is defined by: if ... else $S := 1$.

[1] The original notation is `P(S)` for `Wait(S)` and `V(S)` for `Signal(S)`, the letters P and V taken from corresponding words in Dutch.

4.2 Semaphore Invariants

A semaphore satisfies the following invariants:

$$S \geq 0 \tag{4.1}$$

$$S = S0 + \#Signals - \#Waits \tag{4.2}$$

where $S0$ is the initial value of the semaphore, $\#Signals$ is the number of signals executed on S, and $\#Waits$ is the number of completed waits executed on S. These invariants follow directly from the definition of semaphores, i.e. if you write a program and one of these formulas is not invariant, you have been given a defective implementation of semaphores.

The only non-trivial part to prove is the case of a signal which wakes a suspended process. But then $\#Signals$ and $\#Waits$ both increase by one so their difference remains invariant as does the value of S.

In the next section we give a solution to the mutual exclusion problem using semaphores and prove its correctness by appealing to the semaphore invariants.

4.3 Mutual Exclusion

Figure 4.1 is a solution to the mutual exclusion problem for two processes using semaphores. A process that wishes to enter its critical section, say `P1`, executes a pre-protocol that consists only of the `Wait(S)` instruction. If $S = 1$ then S can be decremented and `P1` enters its critical section. When `P1` exits its critical section

```
  S: Semaphore := 1;

task body P1 is
begin
  loop
    Non_Critical_Section_1;
    Wait(S);
    Critical_Section_1;
    Signal(S);
  end loop;
end P1;

task body P2 is
begin
  loop
    Non_Critical_Section_2;
    Wait(S);
    Critical_Section_2;
    Signal(S);
  end loop;
end P2;
```

Figure 4.1 Mutual exclusion with semaphores

and executes the post-protocol consisting only of the `Signal(S)` instruction, the value S will once more be 1. However, if `P2` attempts to enter its critical section before `P1` has left, $S = 0$ and `P2` will suspend on S. When `P1` finally leaves, the `Signal(S)` will wake `P2`.

The solution is similar to the second attempt of the previous chapter, except that the atomic implementation of the semaphore instruction prevents interleaving between the test of S and the assignment to S. It differs from the test and set instruction in that a process suspended on a semaphore no longer executes instructions checking variables in a busy-wait loop.

Theorem 4.3.1 *The mutual exclusion property is satisfied.*

Proof: Let $\#CS$ be the number of processes in their critical sections. We will prove that

$$\#CS + S = 1 \tag{4.3}$$

is invariant. Since $S \geq 0$ by invariant (4.1), simple arithmetic shows that $\#CS \leq 1$ which proves the mutual exclusion property.

To prove that (4.3) is invariant, we use the semaphore invariant (4.2).

1. $\#CS = \#Wait(S) - \#Signal(S)$. An invariant easily proven from the program text.
2. $S = 1 + \#Signal(S) - \#Wait(S)$. The semaphore invariant.
3. $S = 1 - \#CS$. From (1) and (2).
4. $\#CS + S = 1$. Immediate from (3). □

Theorem 4.3.2 *The program cannot deadlock.*

Proof: For the program to deadlock, both process must be suspended on `Wait(S)`. Then $S = 0$ because they are suspended and $\#CS = 0$ since neither is in the critical section. By the critical section invariant (4.3), $0 + 0 = 1$ which is impossible. □

Theorem 4.3.3 *There is no individual starvation.*

Proof: If `P1` is suspended, the semaphore must be 0. By the semaphore invariant, `P2` is in the critical section. When `P2` exits the critical section, it will execute `Signal(S)` which will wake some process suspended on S. Since `P1` is the only process suspended on S, it will be awakened and enter its critical section. □

Finally, it should be obvious that in the absence of contention, $S = 1$ and no single process will be delayed.

4.4 Semaphore Definitions

There are many definitions of semaphores in the literature. It is important to be able to distinguish between the various definitions because the correctness of a

Chapter 5

Monitors

5.1 Introduction

The semaphore was introduced to provide a synchronization primitive that does not require busy waiting. Using semaphores, we have given solutions to common concurrent programming problems. However, the semaphore is still a low-level primitive because it is unstructured. If we were to build a large system using semaphores alone, the responsibility for the correct use of the semaphores is diffused among all the implementers of the system. If one of them forgets to call `Signal(S)` after a critical section, the program can deadlock and the cause of the failure will be difficult to isolate.

Monitors provide a structured concurrent programming primitive that concentrates the responsibility for correctness into a few modules. Monitors are a generalization of the monolithic monitor (or *kernel* or *supervisor*) found in operating systems. Critical sections such as allocation of I/O devices and memory, queuing requests for I/O, and so on, are centralized in a privileged program. Ordinary programs request *services* which are performed by the central monitor. These programs are run in a hardware mode that ensures that they cannot be interfered with by ordinary programs. Because of the separation between the system and its applications programs, it is usually clear who is at fault if the system crashes (though it may be extremely difficult to diagnose the exact reason).

The monitors discussed in this chapter are decentralized versions of the monolithic monitor. Rather then have one system program handle all requests for services involving shared devices or data structures, we can define a separate monitor for each object or related group of objects. Processes request services from the various monitors. If the same monitor is called by two processes, the implementation ensures that these are processed serially to preserve mutual exclusion. If different monitors are called, their executions can be interleaved.

The syntax of monitors is based on *encapsulating* items of data and the procedures that operate upon them in a single module. The interface to a monitor will consist of a set of procedures. These procedures operate on data that are hidden within the module. The difference between a monitor and an ordinary module such as an Ada package is that a monitor not only protects internal data from

unrestricted access but also synchronizes calls to the interface procedures. The implementation ensures that the procedures are executed under mutual exclusion.

We will define a synchronization primitive that will allow a process to suspend itself if necessary. For example, in the producer–consumer problem:

- The only operations permitted on a buffer are append and remove an item.
- Append and remove exclude each other.
- A producer will suspend on a full buffer and a consumer on an empty buffer.

5.2 Producer–Consumer Problem

We will define the monitor construct in parallel with the solution of the producer–consumer problem (Figure 5.1).[1] Note that the monitor is not a process (Ada task), but a static module of data and procedure declarations. The actual producer and consumer processes have to be programmed separately (Figure 5.2).

```
monitor Producer_Consumer_Monitor is
  B: array(0..N-1) of Integer;
  In_Ptr, Out_Ptr: Integer := 0;
  Count: Integer := 0;
  Not_Full, Not_Empty: Condition;

procedure Append(I: in Integer) is
begin
    if Count = N then Wait(Not_Full); end if;
    B(In_Ptr) := I;
    In_Ptr := (In_Ptr + 1) mod N;
    Signal(Not_Empty);
end Append;

procedure Take(I: out Integer) is
begin
    if Count = 0 then Wait(Not_Empty); end if;
    I := B(Out_Ptr);
    Out_Ptr := (Out_Ptr + 1) mod N;
    Signal(Not_Full);
end Take;

end Producer_Consumer_Monitor;
```

Figure 5.1 Monitor for producer-consumer

Despite the syntactic similarity to an ordinary module (Ada package), the semantics of a monitor are different because only one process is allowed to execute

[1] Unlike most of the examples in this book, this one is not executable in Ada without modification. See Appendix B for details.

```
task body Producer is
  I: Integer;
begin
  loop
    Produce(I);
    Append(I);
  end loop;
end Producer;

task body Consumer is
  I: Integer;
begin
  loop
    Take(I);
    Consume(I);
  end loop;
end Consumer;
```

Figure 5.2 Producer and consumer processes

a monitor procedure at any time. In this case, the producer can be executing `Append` or the consumer `Take`, but not both. This ensures the mutual exclusion on the global variables, in particular, on the variable `Count` which is updated by both procedures.

The solution is more structured than the semaphore solution both because the data and procedures are encapsulated in a single module and because the mutual exclusion is provided automatically by the implementation. The producer and consumer processes see only abstract `Append` and `Take` operations and do not have to be concerned with correctly programming semaphores.

The solution that used binary semaphores, used three of them: `S` for mutual exclusion, and `Not_Empty` and `Not_Full` for synchronization. The mutual exclusion requirement is now satisfied by the definition of monitors. For synchronization, we define a structure called *condition variables*. A condition variable `C` has three operations defined upon it:[2]

Wait(C) The process that called the monitor procedure containing this statement is suspended on a FIFO queue associated with `C`. The mutual exclusion on the monitor is released.

Signal(C) If the queue for `C` is non-empty then wake the process at the head of the queue.

Non_Empty(C) A boolean function that returns true if the queue for `C` is non-empty.

The `Wait` operation allows a process to suspend itself. Conventionally, the name of the condition variable is chosen so that `Wait(C)` can be read: 'I am waiting

[2] We are using the same names *Wait* and *Signal* that were used for the semaphore operations, but there is no relation between the two primitives.

5.3 Emulation of Semaphores by Monitors

In this section and the next one, we will show how to emulate a semaphore using monitors and conversely. This will not only show that the two primitives are of similar power and expressibility, but will also show that the monitor is a higher-level abstraction than a semaphore because the emulation in one direction will be so much easier. These emulations can also be used to port a program from a system supplying one primitive to a system supplying the other one.

```
monitor Semaphore_Emulation is
  S: Integer := S0;
  Not_Zero: Condition;

  procedure Semaphore_Wait is
  begin
    if S=0 then Wait(Not_Zero); end if;
    S := S - 1;
  end Semaphore_Wait;

  procedure Semaphore_Signal is
  begin
    S := S + 1;
    Signal(Not_Zero);
  end Semaphore_Signal;
end monitor;
```

Figure 5.3 Emulation of semaphores by monitors

The monitor in Figure 5.3 emulates a semaphore. The variable `S` holds the value of the semaphore and is initialized to some non-negative value `S0`. (We could also have defined another procedure to initialize `S`.) The condition variable `Not_Zero` maintains the queue of processes waiting for the semaphore to be non-zero.

Theorem 5.3.1 *The semaphore invariants hold:*

$$S \geq 0 \tag{5.1}$$

$$S = S0 + \#waits - \#signals \tag{5.2}$$

Proof: As usual, the proof is by induction on the execution sequence. Since each monitor procedure is executed under mutual exclusion with no possibility of interleaving, we can relax the proof rules. It is sufficient to prove that the formulas are invariant in any interleaving where every execution of a monitor procedure is a single atomic instruction. The fact that the variables may temporarily have values that falsify the invariant may be ignored since no other process can see these values. Remember that a `Wait(C)` instruction is considered to cause a process to leave the monitor, so the invariants must be checked there too.