Chapter 4

INTER –PROCESS SYNCHRONIZATION

4.1 Background

In Chapter 2, we developed a model of a system consisting of a number of cooperating sequential processes, all running asynchronously and possibly sharing data. We have illustrated this model with the bounded-buffer scheme, which is representative of operating systems.

Let us return to the shared-memory solution to the bounded-buffer problem that we presented in Section 2.4. As we pointed out, our solution allows at most n - 1 items in the buffer at the same time. Suppose that we wanted to modify the algorithm to remedy this deficiency. One possibility is to add an integer variable counter, initialized to 0. Counter is incremented every time we add a new item to the buffer, and is decremented every time we remove one item from the buffer. The code for the producer process can be modified as follows:

repeat
…
produce an item in *nextp*
…
while *counter = n* **do** *no-op;*
buffer[in] := *nextp;*
in := *in + 1* **mod** *n;*
counter := *counter + 1;*
until *false;*

The code for the consumer process can be modified as follows:

repeat
while *counter = 0* **do** *no-op;*
nextc := *buffer[out];*
out := *out + 1* **mod** *n;*
counter := *counter - 1;*
…
consume the item in *nextc*
…
until *false;*

Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently. As an illustration, suppose that the value of the variable *counter* is currently *5,* and that the producer and consumer processes execute the statements *"counter* := *counter* + **1"** and *"counter* := *counter* - **1"** concurrently. Following the execution of these two statements, the value of the variable *counter* may be 4,5, or **6!** The only correct result is *counter* = *5,* which is generated correctly if the producer and consumer execute separately.

We can show that the value of *counter* may be incorrect, as follows. Note that the statement *"counter* := *counter+lW* may be implemented in machine language (on a typical machine) as

registerl := *counter;*
registerl := *registerl+l;*
counter := *register1*

where *registerl* is a local CPU register. Similarly, the statement *"counter* := *counter - 1"* is implemented as follows:

register2 := *counter*
register2 := *register2 - 1;*
counter := *register2*

where again ***register2*** is a local **CPU** register. Even though ***registerl*** and ***register2*** may be the same physical registers (an accumulator, say), remember that the contents of this register will be saved and restored by the interrupt handler.

The concurrent execution of the statements ***"counter*** := ***counter*** + 1" and ***"counter*** := ***counter*** - 1" is equivalent to a sequential execution where the lowerlevel statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such inter leaving is

T0: producer* execute *register, :=counter {register, = 5)
T1: producer* execute *register,* := *registerl* + 1 *{registerl = 6)
T2: consumer* execute *register2* := *counter {register2 = 5)
T3: consumer* execute *register2* := *register2* - 1 *(register2 = 4)
T4: *producer* execute *counter* := *register1 {counter = 6)*
T5: consumer* execute *counter* := *register2 {counter = 4)

Notice that we have arrived at the incorrect state ***"counter = 4,"*** recording that there are four full buffers, when, in fact, there are five full buffers. If we reversed the order of the statements at ***T4*** and T5, we would arrive at the incorrect state ***"counter = 6."***

We would arrive at this incorrect state because we allowed both processes to manipulate the variable ***counter*** concurrently. A situation like this, where several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place, is called a ***race condition.*** To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable ***counter.***

To make such a guarantee, we require some form ofsynchronization of the processes. Such situations occur frequently in operating systems as different parts of the system manipulate resources and we want the changes not to interfere with one another. A major portion of this chapter is concerned with the issue of process synchronization and coordination.

4.2 Classical Problems of Synchronization

In this section, we present a number of different synchronization problems that are important mainly because they are examples for a large class of concurrency-control problems. These problems are used for testing nearly every newly proposed synchronization scheme. Semaphores are used for synchronization in our solutions.

4.2.1 The Bounded-Buffer Problem

The bounded-buffer problem is commonly used to illustrate the power of synchronization primitives. We present here a general structure of this scheme, without committing ourselves to any particular implementation. We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.

repeat

…

produce an item in nextp

```
...
wait(empty);
wait(mutex);

...
add nextp to buffi
...
signal(mutex);
signal(ful1);
until false;
```

Figure 4.1 The structure of the producer process.

The empty and fill semaphores count the number of empty and full buffers, respectively. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The code for the producer process is shown in Figure 4.1: the code for the consumer process is shown in Figure 4.2. Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer, or as the consumer producing empty buffers for the producer.

4.2.2 The Readers and Writers Problem

A data object (such as a file or record) is to be shared among several concurrent processes. Some of these processes may want only to read the content of the shared object, whereas others may want to update (that is, to read and write) the shared object.

```
repeat
wait(ful1);
wait(mutex);
...
remove an item from buffer to nextc
...
signal(mutex);
signal (empty);
...
consume the item in nextc
...
```

until false;

Figure 4.2 The structure of the consumer process.

We distinguish between these two types of processes by referring to those processes that are interested in only reading as readers, and to the rest as writers. Obviously, if two readers access the shared data object simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the shared object simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared object. This synchronization problem is referred to as the readers-writers problem. Since it was originally stated, it has been used to test nearly every new synchronization primitive. The readers writers problem has several variations, all involving priorities.

The simplest one, referred to as the first readers-writers problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting.

The second readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

We note that a solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. **In** this section, we present a solution to the first readers-writers problem. Refer to the Bibliographic
Notes for relevant references on starvation-free solutions to the readers-writers problem.

In the solution to the first readers-writers problem, the reader processes share the following data structures:

```
var mutex, wrt: semaphore;
readcount: integer;
```

The semaphores mutex and wrt are initialized to 1; readcount is initialized to 0. The semaphore ***wrt*** is common to both the reader and writer processes.

The mutex semaphore is used to ensure mutual exclusion when the ariable readcount is updated. Readcount keeps track of how many processes are currently reading the object. The semaphore wrt functions as a mutual exclusion semaphore for the writers.
It also is used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

The code for a writer process is shown in Figure 4.3; the code for a reader process is shown in Figure 4.4. Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on wrt, and n - 1 readers are queued on mutex. Also observe that, when a writer executes signal(wrt), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

```
wait(wrt);
....
writing is performed
...
signal(wrt);
```

Figure 4.3 The structure of a writer process.

4.2.3 The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table there is a bowl of rice, and the table is laid with five single chopsticks (Figure 6.16). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).

A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

The dining-philosophers problem is considered a classic synchronization

problem, neither because of its practical importance, nor because computer scientists dislike philosophers, but because it is an example for a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock and starvation free manner.

```
wait(mutex);
readcount := readcount + 1;
if readcount = 1 then wait(wrt);
signal(mutex);
...
reading is performed
...
wait(mutex);
readcount := readcount - 1;
if readcount = 0 then signal(wrt);
signal(mutex);
```

Figure 4.4 The structure of a reader process.

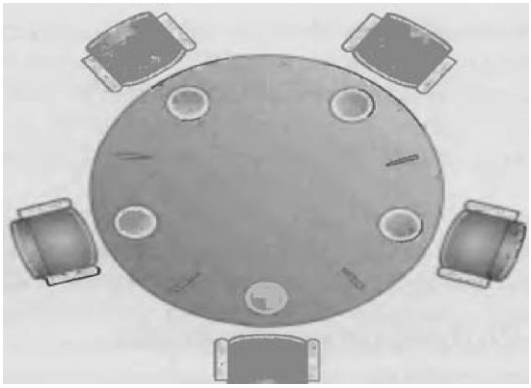


Figure 4.5 The situation of the dining philosophers.

One simple solution is to represent each chopstick by a semaphore. A philosopher tries to grab the chopstick by executing a ***waif*** operation on that semaphore; she releases her chopsticks by executing the ***signal*** operation on the appropriate semaphores. Thus, the shared data are var ***chopstick:*** array **[0..4] of *semaphore;*** where all the elements of ***chopstick*** are initialized to ***1.*** The structure of philosopher ***i*** is shown in Figure 4***.17.*** Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it has the possibility of creating a deadlock. Suppose that all five philosophers become hungry

simultaneously, and each grabs her left chopstick. All the elements of ***chopstick*** will now be

```
repeat
wait(chopstick[i]);
wait(chopstick[i+1 mod 51);
..
eat
..
signal(chopstick[i]);
signal(chopstick[i+1 mod 51);
...
think
...
until false;
```

Figure 4.17 The structure of philosopher ***i.***

equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever. Several possible remedies to the deadlock problem are listed next. In Section 4.7, we present a solution to the dining-philosophers problem that ensures freedom from deadlocks.

- Allow at most four philosophers to be sitting simultaneously at the table.

- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (note that she must pick them up in a critical section).

- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Finally, any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death. A deadlock-free solution does not necessarily eliminate the possibility of starvation.

6.6 Critical Sections

Although semaphores provide a convenient and effective mechanism for process synchronization, their incorrect use can still result in timing errors that are

difficult to detect, since these errors happen only if some particular execution sequences take place, and these sequences do not always occur. We have seen an example of such types of errors in the use of counters in our solution to the producer-consumer problem (Section 6.1). In that example, the timing problem happened only rarely, and even then the counter value appeared to be a reasonable value - off by only 1. Nevertheless, this solution is obviously not an acceptable one. It is for this reason that semaphores were introduced in the first place.

Unfortunately, such timing errors can still occur with the use of semaphores. To illustrate how, let us review the solution to the critical section problem using semaphores. All processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait(mutex) before entering the critical section, and signal(mutex) afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously.

Let us examine the various difficulties that may result. Note that these difficulties will arise even if a single process is not well behaved. This situation may be the result of an honest programming error or of an uncooperative programmer.

Suppose that a process interchanges the order in which the ***wait*** and ***signal*** operations on the semaphore ***mutex*** are executed, resulting in the following execution:

```
signal (mutex);
.....
critical section
......
wait ();
```

In this situation, several processes may be executing in their critical section simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

- Suppose that a process replaces ***signal(mutex)*** with ***wait(mutex).*** That is, it executes

wait(mutex);
...
critical section

...
wait(mutex);

In this case, a deadlock will occur.

- Suppose that a process omits the ***wait(mutex),*** or the ***signal(mutex),*** or both.In this case, either mutual exclusion is violated or a deadlock will occur.

These examples illustrate that various types of errors can be generated easily when semaphores are used incorrectly to solve the critical-section problem. Similar problems may arise in the other synchronization models we discussed in Section 4.5.

To deal with the type of errors we have outlined, a number of high-level language constructs have been introduced. In this section, we describe one fundamental high-level synchronization construct - the ***critical region*** (sometimes referred to as ***conditional critical region).*** In Section ***6.7,*** we present another fundamental synchronization construct - the ***monitor.*** In our presentation of these two constructs, we assume that a process consists of some local data, and a sequential program that can operate on the data. The local data can be accessed by only the sequential program that is encapsulated within the same process.That is, one process cannot directly access the local data of another process. Processes can, however, share global data.

The critical-region high-level language synchronization construct requires that a variable v of type *T,* which is to be shared among many processes, be declared as

var v: shared *T;*

The variable v can be accessed only inside a region statement of the following form:

region v when B do S;

This construct means that, while statement S is being executed, no other process can access the variable v. The expression B is a Boolean expression that governs the access to the critical region. When a process tries to enter the critical section region, the Boolean expression B is evaluated. If the expression is true, statement **S** is executed. If it is false, the process relinquishes the mutual exclusion and is delayed until B becomes true and no other process is in the region associated with v. Thus, if the two statements,

region v when true do S1;
region v when true do S2;

are executed concurrently in distinct sequential processes, the result will be equivalent to the sequential execution "S1 followed by S2," or "S2 followed by S1."

The critical-region construct guards against certain simple errors associated with the semaphore solution to the critical-section problem that may be made by a programmer. Note that it does not necessarily eliminate all synchronization errors; rather, it reduces their number. If errors occur in the logic of the program, reproducing a particular sequence of events may not be simple. The critical-region construct can be effectively used to solve certain general synchronization problems. To illustrate, let us code the bounded-buffer scheme.

The buffer space and its pointers are encapsulated in

```
var buffer: shared record
                pool: array [0..n−1] of item;
                count,in,out: integer;
            end;
```

The producer process inserts a new item nextp into the shared buffer by Executing.

```
region buffer when count < n
    do begin
        pool[in] := nextp;
        in := in+1 mod n;
        count := count + 1;
    end;
```

The consumer process removes an item from the shared buffer and puts it in nextc by executing

```
region buffer when count > 0
    do begin
        nextc := pool[out];
        out := out+1 mod n;
        count := count − 1;
    end;
```

Let us illustrate how the conditional critical region could be implemented by a compiler. With each shared variable, the following variables are associated:

```
var mutex, first-delay, second-delay: semaphore;
    first-count, second-count: integer;
```

The semaphore mutex is initialized to 1; the semaphoresfirst-delay and seconddelay are initialized to 0. The integersfirst-count and second-count are initialized to 0.

Mutually exclusive access to the critical section is provided by mutex. If a process cannot enter the critical section because the Boolean condition B is false, it initially waits on thefirst-delay semaphore. A process waiting on the first-delay semaphore is eventually moved to the second-delay semaphore before it is allowed to reevaluate its Boolean condition B. We keep track of the number of processes waiting onfirst-delay and second-delay, withfirst-count and second count respectively.

When a process leaves the critical section, it may have changed the value of some Boolean condition B that prevented another process from entering the critical section. Accordingly, we must trace through the queue of processes waiting on first-delay and second-delay (in that order) allowing each process to test its Boolean condition. When a process tests its Boolean condition (during this trace), it may discover that the latter now evaluates to the value true. In this case, the process enters its critical section. Otherwise, the process must wait again on the first-delay and second-delay semaphores, as described previously

Accordingly, for a shared variable x, the statement

region x **when B do S;**

```
wait(mutex);
while not B
    do begin
            first-count:=first-count + 1;
            if second-count ⩾ 0
                then signal(second-delay)
                else signal(mutex);
            wait(first-delay);
            first-count:=first-count – 1;
            second-count:= second-count + 1;
            iffirst-count⩾ 0
                then signal(first-delay)
                else signal(second-delay);
            wait(second-delay);
            second-count := second-count – 1;
       end;
S;
iffirst-count> 0
    then signal(first-delay);
    else if second-count ⩾ 0
                then signal(second-delay);
                else signal(mutex);
```

Figure 4.18 Implementation of the conditional-region construct.

can be implemented as shown in Figure 4.18. Note that this implementation requires the reevaluation of the expression B for any waiting processes every time a process leaves the critical region. If several processes are delayed, waiting for their respective Boolean expressions to become true, this reevaluation overhead may result in inefficient code. There are various optimization methods that we can use to reduce this overhead. Refer to the Bibliographic Notes for relevant references.

4.2 The Critical-Section Problem

Consider a system consisting of ***n*** processes ***{Po, PI,*** …, ***Pn-l).*** Each process has a segment of code, called a ***critical section,*** in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section,no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is ***mutually exclusive*** in time. The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section.

The section of code implementing this request is the ***entry*** section. The critical section may be followed by an ***exit*** section. The remaining code is the ***remainder*** section.

A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual Exclusion: If process ***Pi*** is executing in its critical section, then no other processes can be executing in their critical sections.
2. Progress: If no process is executing in its critical section and there exist some processes that wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision of which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. Bounded Waiting: There exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the ***relative*** speed of the ***n*** processes. In Sections 4.2.1 and 4.2.2, we work up to solutions to the critical section problem that satisfy these three requirements.

The solutions do not rely on any assumptions concerning the hardware instructions or the number of processors that the hardware supports. We do, however, assume that the basic machine language instructions (the primitive instructions such as load, store, and test) are executed atomically. That is, if two such instructions are executed concurrently, the result is equivalent to their sequential execution in some unknown order. Thus, if a load and a store are executed concurrently, the load will get either the old value or the new value, but not some combination of the two.

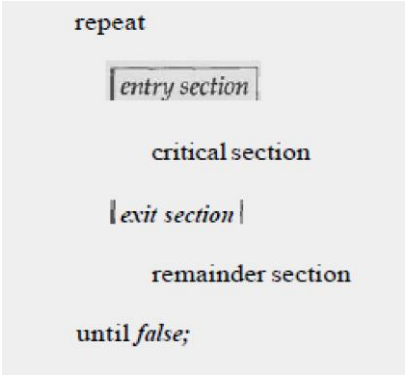


Figure 4.1 General structure of a typical process *Pi*.

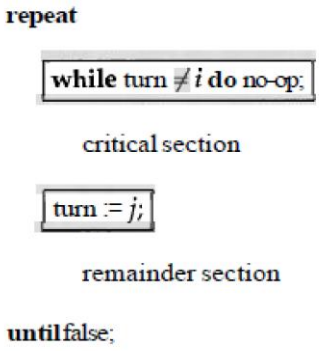


Figure 4.2 The structure of process *Pi* in algorithm 1.

When presenting an algorithm, we define only the variables used for synchronization purposes, and describe only a typical process ***Pi*** whose general structure is shown in Figure 6.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

4.2.1 Two-Process Solutions (Peterson's Algorithm)

In this section, we restrict our attention to algorithms that are applicable to only two processes at a time. The processes are numbered ***Po*** and ***P1.*** For convenience, when presenting ***Pi,*** we use ***Pj*** to denote the other process; that is, j= l-i .

4.2.1.1 Algorithm 1

Our fist approach is to let the processes share a common integer variable turn initialized to 0 (or 1). If turn = i, then process ***Pi*** is allowed to execute in its critical section. The structure of process ***Pi*** is shown in Figure 4.2.

This solution ensures that only one process at a time can be in its critical section. However, it does not satisfy the progress requirement, since it requires strict alternation of processes in the execution of the critical section. For example, if turn = 0 and ***P1*** is ready to enter its critical section, ***P1*** cannot do so, even though ***Po*** may be in its remainder section.

4.2.1.2 Algorithm 2

The problem with algorithm 1 is that it does not retain sufficient information about the state of each process; it remembers only which process is allowed to enter that process' critical section. To remedy this problem, we can replace the variable turn with the following array:

var flag: **array** [0..l] of boolean;

The elements of the array are initialized to false. If flag[i] is true, this value indicates that ***Pi*** is ready to enter the critical section. The structure of process ***Pi*** is shown in Figure 4.3.

In this algorithm, process ***Pi*** first sets flag[i] to be true, signalling that it is ready to enter its critical section. Then, ***Pi*** checks to verify that process ***Pj*** is also not ready to enter its critical section. If ***Pj*** were ready, then ***Pi*** would wait until ***Pj*** had indicated that it no longer needed to be in the critical section (that is, untilJag[j] was false). At this point, ***Pi*** would enter the critical section. On exiting the critical section, ***Pi*** would set its Jag to be false, allowing the other process (if it is waiting) to enter its critical section.

In this solution, the mutual-exclusion requirement is satisfied. Unfortunately, the progress requirement is not met. To illustrate this problem, we consider the following execution sequence:

To: ***Po*** sets flag[O] = true
T1: ***P1*** sets flag[l] = true

Now ***Po*** and ***P1*** are looping forever in their respective while statements.

This algorithm is crucially dependent on the exact timing of the two processes. The sequence could have been derived in an environment where there are several processors executing concurrently, or where an interrupt (such as a timer interrupt) occurs immediately after step To is executed, and the **CPU** is switched from one process to another.

Note that switching the order of the instructions for setting flag[i] and testing the value of a flag[j], will not solve our problem. Rather, we will have a situation where it is possible for both processes to be in the critical section at the same time, violating the mutual-exclusion requirement.

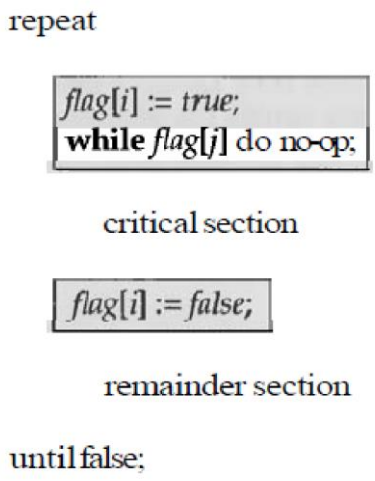


Figure 4.3 The structure of process *Pi* in algorithm 2 .

solution to the critical-section problem, where all three requirements are met. The processes share two variables:

var flag: array [O..l] of boolean;
turn: O..l;

Initially flag[0] =flag[l] =false, and the value of turn is immaterial (but is either 0 or 1). The structure of process ***Pi*** is shown in Figure 4.4. To enter the critical section, process ***Pi*** first sets flag[i] to be true, and then asserts that it is the other process' turn to enter if appropriate (turn = ***1).*** If both processes try to enter at the same time, turn will be set to both ***i*** and j at roughly the same time. Only one of these assignments will last; the other will occur, but will be overwritten

immediately. The eventual value of turn decides which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved,
2. The progress requirement is satisfied,
3. The bounded-waiting requirement is met.

To prove property 1, we note that each ***Pi*** enters its critical section only if either flag[j] =false or turn = i. Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] =flag[l] = true. These two observations imply that ***P0*** and ***P1*** could not have executed successfully their

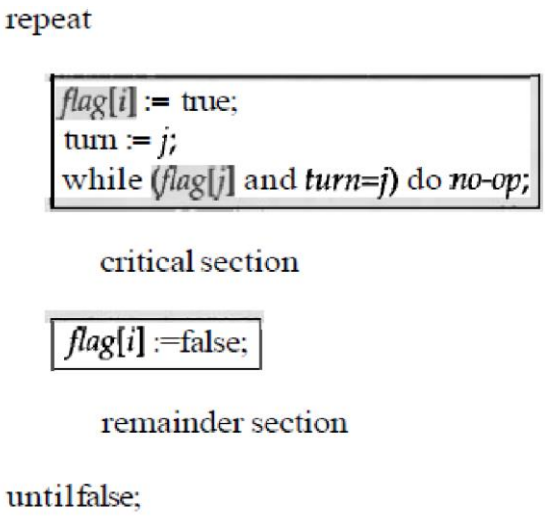


Figure 4.4 The structure of process *Pi* in algorithm 3 .

while statements at about the same time, since the value of turn can be either **0** or 1, but cannot be both. Hence, one of the processes - say ***Pi*** - must have executed successfully the while statement, whereas ***Pi*** had to execute at least one additional statement ("turn = j"). However, since, at that time, flag[j] = true, and turn = i, and this condition will persist as long as ***Pi*** is in its critical section, the result follows: Mutual exclusion is preserved.

To prove properties 2 and 3, we note that a process ***Pi*** can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] = true and turn = *j;* this loop is the only one. If ***Pj*** is not ready to enter the critical section, then flag[j] =false, and ***Pi*** can enter its critical section.

If ***Pi*** has set flag[j] = true and is also executing in its while statement, then either turn = i or turn = *j.* If turn = i, then ***Pi*** will enter the critical section. If turn = j, then ***Pi*** will enter the critical section. However, once ***Pi*** exits its critical section, it will reset flag[j] to false, allowing ***Pi*** to enter its critical section. If ***Pi*** resets flag[j] to true, it must also set turn = i. Thus, since ***Pi*** does not change the value of the variable turn while executing the while statement, ***Pi*** will enter the critical section (progress) after at most one entry by ***Pi*** (bounded waiting).

4.3 Synchronization Hardware

As with other aspects of software, features of the hardware can make the programming task easier and improve system efficiency. In this section, we present some simple hardware instructions that are available on many systems, and show how they can be used effectively in solving the critical section problem.

The critical-section problem could be solved simply in a uniprocessor environment if we could disallow interrupts to occur while a shared variable is being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.

Unfortunately, this solution is not feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time-consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also, consider the effect on a system's clock, if the clock is kept updated by interrupts.

Many machines therefore provide special hardware instructions that allow us either to test and modify the content of a word, or to swap the contents of two words, atomically. We can use these special instructions to solve the critical section problem in a relatively simple manner. Rather than discussing one specific instruction for one specific machine, let us abstract the main concepts behind these types of instructions.
The Test-and-Set instruction can be defined as shown in Figure 4.6. The important characteristic is that this instruction is executed atomically - that is, as one uninterruptible unit. Thus, if two Test-and-Set instructions are executed simultaneously (each on a different **CPU),** they will be executed sequentially in some arbitrary order.

If the machine supports the Test-and-Set instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of process ***Pi*** is shown in Figure 4.7.

The Swap instruction, defined as as shown in Figure 6.8, operates on the contents of two words; like theTest-and-Set instruction, it is executed atomically. If the machine supports the Swap instruction, then mutual exclusion can be provided as follows. A global Boolean variable lock is declared and is initialized

```
function Test-and-Set (var target: boolean): boolean;
    begin
        Test-and-Set := target;
        target := true;
    end;
```

Figure 4.6 The definition of the Test-and-% instruction.

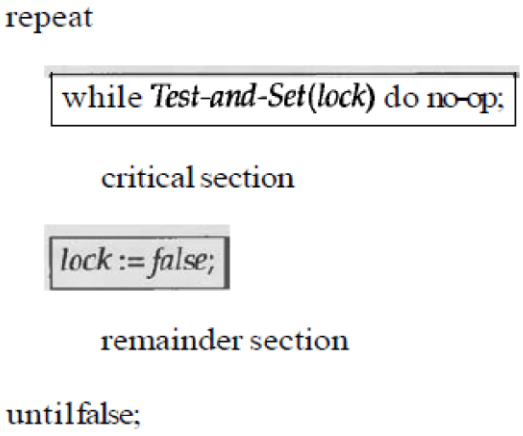


Figure 4.7 Mutual-exclusion implementation with Test-and-Set.

to false. In addition, each process also has a local Boolean variable key. The structure of process ***Pi*** is shown in Figure 4.9.

These algorithms do not satisfy the bounded-waiting requirement. We present algorithm that uses the Test-and-Set instruction in Figure 4.10. This algorithm satisfies all the critical-section requirements.

The common data structures are

```
var waiting: array [0..n – 1] of boolean
    lock: boolean
```

These data structures are initialized to false.

To prove that the mutual-exclusion requirement is met, we note that process ***Pi*** can enter its critical section only if either waiting[i] =false or key =false. Key can become false only if the Test-and-Set is executed. The first process to execute the Test-and-Set will find key =false; all others must wait. The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual-exclusion requirement.

To prove the progress requirement, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section

```
procedure Swap (var a, b: boolean);
    var temp: boolean;
    begin
        temp := a;
        a := b;
        b := temp;
    end;
```

Figure 4.8 the definition of the Swap instruction.

```
repeat
    key := true;
    repeat
       Swap(lock,key);
    until key =false;

        critical section

    lock := false;

        remainder section

until false;
```

Figure 4.9 Mutual-exclusion implementation with the *Swap* instruction.

either sets ***lock*** to ***false,*** or sets ***waiting[j]*** to ***false.*** Both allow a process that is waiting to enter its critical section to proceed.

```
var j: 0..n – 1;
    key: boolean;
repeat
    waiting[i] := true;
    key := true;
    while waiting[i] and key do key := Test-and-Set(lock);
    waiting[i] :=false;

        critical section

    j := i+1 mod n;
    while (j ≠ i) and (not waiting[j]) do j := j+1 mod n;
    if j = i then lock :=false
          else waiting[j] :=false;

        remainder section

until false;
```

Figure 4.10 Bounded-waiting mutual exclusion with *Test-and-Set.*

To prove bounded waiting, we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ..., n - 1, 0, ..., i - 1). It designates the first process in this ordering that is in the entry section (waiting[j] = true) as the next one to enter the critical section.

Any process waiting to enter its critical section will thus do so within n – 1 turns. Unfortunately for hardware designers, implementing atomic test-andset instructions on multiprocessors is not a trivial task. Such implementations are discussed in books on computer architecture.

4.4 Semaphores

The solutions to the critical-section problem presented in Section 6.3 are not easy to generalize to more complex problems. To overcome this difficulty, we can use a synchronization tool, called a semaphore. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait and signal. These operations were originally termed P (for wait; from the Dutch proberen, to test) and V (for signal; from verhogen, to increment). The classical definitions of wait and signal are

```
wait(S):  while S ≤ 0 do no-op;
          S := S – 1;

signal(S): S := S + 1;
```

Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of the wait(S), the testing of the integer value of S (S *5* O), and its possible modification (S := S - 1), must also be executed without interruption. We shall see how these operations can be implemented in Section 6.4.2; first, let us see how semaphores can be used.

4.4.1 Usage

We can use semaphores to deal with the n-process critical-section problem. The n processes share a semaphore, mutex (standing for mutual exclusion), initialized to 1. Each process Pi is organized as shown in Figure 4.11.

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P1 with a statement S1, and P2 with a statement S2. Suppose that we require that S2 be executed only after S1 has completed.

We can implement this scheme readily by letting **P1** and PZ share a common semaphore synch, initialized to 0, and by inserting the statements

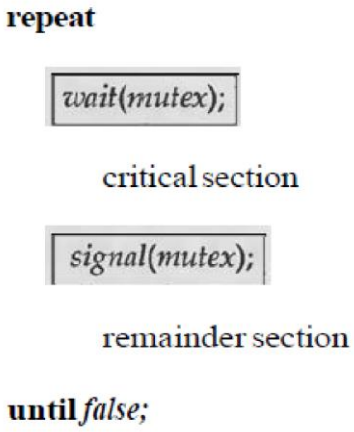
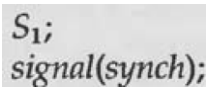
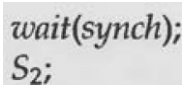


Figure 4.11 Mutual-exclusion implementation with semaphores.



in process PI, and the statements



in process P2. Because ***synch*** is initialized to 0, P2 will execute S2 only after P1 has invoked ***signal(synch),*** which is after S1.

4.4.2 Implementation

The main disadvantage of the mutual-exclusion solutions of Section 6.2, and of the semaphore definition given here, is that they all require ***busy waiting.*** While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively.

This type of semaphore is also called a ***spinlock*** (because the process "spins" while waiting for the lock). Spinlocks are useful in multiprocessor systems. The advantage of a spinlock is that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful.

To overcome the need for busy waiting, we can modify the definition of the ***wait*** and ***signal*** semaphore operations. When a process executes the ***wait*** operation and finds that the semaphore value is not positive, it must wait. However, rather than busy waiting, the process can ***block*** itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then, control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore *S*, should be restarted when some other process executes a *signal* operation. The process is restarted by a *wakeup* operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

To implement semaphores under this definition, we define a semaphore as a record:

type *semaphore* = **record**
value: integer;
L: **list of** *process;*
end;

Each semaphore has an integer value and a list of processes. When a process must wait on a semaphore, it is added to the list of processes. A *signal* operation removes one process from the list of waiting processes, and awakens that process.

The semaphore operations can now be defined as

wait(S): S.value := S.value - 1;
if *S.value* < 0
then begin
add this process to *S.L;*
block;
end;

signal(S): S.value := *S.value* + 1;
if *S.value* < 0
then begin
remove a process P from *S.L;*
wakeup(P);
end;

The *block* operation suspends the process that invokes it. The w*akeup(P)* operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

Note that, although under the classical definition of semaphores with busy waiting the semaphore value is never negative, this implementation may have negative semaphore values. If the semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact is a result of the switching of the order of the decrement and the test in the implementation of the ***wait*** operation.

The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list, which ensures bounded waiting, would be to use a first-in, first-out (FIFO) queue, where the semaphore contains both head and tail pointers to the queue. In general, however, the list may use ***any*** queueing strategy. Correct usage of semaphores does not depend on a particular queueing strategy for the semaphore lists.

The critical aspect of semaphores is that they are executed atomically. We must guarantee that no two processes can execute ***wait*** and ***signal*** operations on the same semaphore at the same time. This situation is a critical-section problem, and can be solved in either of two ways.

In a uniprocessor environment (that is, where only one CPU exists), we can simply inhibit interrupts during the time the ***wait*** and ***signal*** operations are executing. This scheme works in a uniprocessor environment because, once interrupts are inhibited, instructions from different processes cannot be interleaved. Only the currently running process executes, until interrupts are reenabled and the scheduler can regain control.

In a multiprocessor environment, inhibiting interrupts does not work. Instructions from different processes (running on different processors) may be interleaved in

some arbitrary way. If the hardware does not provide any special instructions, we can employ any of the correct software solutions for the critical section problem (Section 4.2), where the critical sections consist of the ***wait*** and ***signal*** procedures.

It is important to admit that we have not completely eliminated busy waiting with this definition of the ***wait*** and ***signal*** operations. Rather, we have removed busy waiting from the entry to the critical sections of applications programs. Furthermore, we have limited busy waiting to only the critical sections of the ***wait*** and ***signal*** operations, and these sections are short (if properly coded, they should be no more than about 10 instructions). Thus, the critical section is almost never occupied, and busy waiting occurs rarely, and then for only a short time. An entirely different situation exists with applications programs whose critical sections may be long (minutes or even hours) or may be almost always occupied. In this case, busy waiting is extremely inefficient.

4.4.3 Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes. The event in question is the execution of a ***signal*** operation. When such a state is reached, these processes are said to be ***deadlocked.***

To illustrate this, we consider a system consisting of two processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1:

wait (S); wait(Q);
wait(Q); wait(S);
signal (S); signal(Q);
siflal(Q); signal(S);

Suppose that Po executes wait(S), and then PI executes wait(Q). When P0 executes wait(Q), it must wait until P1 executes signal(Q). Similarly, when PI executes wait(S), it must wait until Po executes signal(S). Since these signal operations cannot be executed, Po and P1 are deadlocked.

We say that a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release.

Another problem related to deadlocks is indefinite blocking or starvation, a situation where processes wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO order.

4.4.4 Binary Semaphores

The semaphore construct described in the previous sections is commonly known as a counting semaphore, since its integer value can range over an unrestricted domain. A binary semaphore is a semaphore with an integer value that can range only between 0 and 1. A binary semaphore can be simpler to implement than a counting semaphore, depending on the underlying hardware architecture. We will now show how a counting semaphore can be implemented using binary semaphores.

Let S be a counting semaphore. To implement it in terms of binary semaphores we need the following data structures:

```
var S1: binary-semaphore;
  52: binary-semaphore;
  S3: binary-semaphore;
    C: integer;
```

Initially S1 = 53 = 1, S2 = **0,** and the value of integer **C** is set to the initial value of the counting semaphore S.
The wait operation on the counting semaphore S can be implemented as follows:

```
wait(S3);
wait(S1);
C := C - 1;
ifC<O
then begin
signal(S1);
wait(S2);
end
else signal(S1);
signal(S3);
```

The signal operation on the counting semaphore S can be implemented as follows:

wait(S1);
C := **C** + 1;
if C *5* 0 **then** signal(S2);
signal(S1);

The 53 semaphore has no effect on signal(S), it merely serializes the wait(S) operations.