To my parents, Wira and Mietek, my wife, Haya, and my children, Lemor, Sivan and Aaron.

Avi Silberschatz

To my wife, Jeanne, and my children, Jennifer and Kathryn.

Jim Peterson

Mark Dalton: Sponsoring Editor

Hugh Crawford: Manufacturing Supervisor
Karen Guardino: Production Manager
Thomas A. Philbrook and **Barbara Atkinson:** Cover Designers
Susan E. Vicenti: Art Editor
Natasha Wei: Production Editor

This book is in the Addison-Wesley series in Computer Science.

Michael A. Harrison: Consulting Editor

Library of Congress Cataloging in Publication Data

Peterson, James Lyle.
Operating system concepts.

Bibliography: p.
Includes index.
1. Operating systems (Computers) I. Silberschatz, Abraham. II. Title.
QA76.6.P475 1985 001.64'2 84-21637
ISBN 0-201-06198-8

Reproduced by Addison-Wesley from camera-ready copy prepared by the authors.

BCDEFGHIJ-AL-898765

Operating System Concepts

Second Edition

James L. Peterson
Abraham Silberschatz

University of Texas at Austin

Addison-Wesley Publishing Company

Reading, Massachusetts • Menlo Park, California
Don Mills, Ontario • Wokingham, England • Amsterdam
Sydney • Singapore • Tokyo • Mexico City • Bogotá
Santiago • San Juan

Minoura [1982]. The issue of testing for deadlock-freedom in a computer system was discussed by Kameda [1980].

Survey papers have been written by Coffman et al. [1971] and Isloor et al. [1980]. A formal treatment of deadlocks was presented by Shaw [1974, Chapter 8] and Coffman and Denning [1973, Section 2.3].

9

Concurrent Processes

Until now we have only considered the issue of concurrency as it relates to hardware components or user processes. In this chapter we examine the general issue of concurrency as it relates to arbitrary algorithms. In particular, we exploit concurrency both within a single process and concurrency across processes.

9.1 Precedence Graphs

Consider the following program segment, Program 1, which performs some simple arithmetic operations.

$$a := x + y;$$
$$b := z + 1;$$
$$c := a - b;$$
$$w := c + 1;$$

Suppose we want to execute some of these statements concurrently. We may have multiple functional units (such as adders) in our processor, or multiple cpus. The "addition" and "subtraction" might be operations on matrices or vectors, or on very large sets (union and intersection), or on files (concatenation and difference). With multiple cpus we may be able to execute some statements concurrently with others, reducing our total execution time.

Clearly, the statement $c := a - b$ cannot be executed before both a and b have been assigned values. Similarly, $w := c + 1$ cannot be executed before the new value of c has been computed. On the other hand, the statements $a := x + y$ and $b := z + 1$ could be executed concurrently since neither depends upon the other.

The point of this example is that, within a single program, there are *precedence constraints* among the various statements. In the following sections, we formalize these ideas.

9.1.1 Definition

A *precedence graph* is a directed acyclic graph whose nodes correspond to individual statements. An edge from node S_i to node S_j means that statement S_j can be executed only after statement S_i has completed execution.

For example, in the precedence graph depicted in Figure 9.1, the following precedence relations exist:

- S_2 and S_3 can be executed after S_1 completes.
- S_4 can be executed after S_2 completes.
- S_5 and S_6 can be executed after S_4 completes.
- S_7 can execute only after S_5, S_6, and S_3 complete.

Note that S_3 can be executed concurrently with S_2, S_4, S_5, and S_6.

The precedence graph must be acyclic. The graph in Figure 9.2 has the following precedence constraints: S_3 can be executed only after S_2 has completed and S_2 can be executed only after S_3 has completed. Obviously, these constraints cannot both be satisfied at the same time.

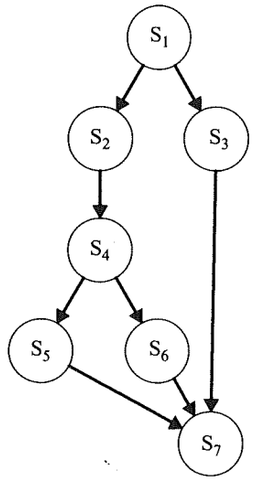

Figure 9.1 Precedence graph

9.1.2 Concurrency Conditions

When can two statements in a program be executed concurrently and still produce the same results? That is, should there be an edge from S_1 to S_2 in the precedence graph corresponding to this program? Before we answer this question, let us first define some notation.

- $\mathbf{R}(S_i) = \{a_1, a_2, \ldots, a_m\}$, the *read set* for S_i, is the set of all variables whose values are referenced in statement S_i during its execution.
- $\mathbf{W}(S_i) = \{b_1, b_2, \ldots, b_n\}$, the *write set* for S_i, is the set of all variables whose values are changed (written) by the execution of statement S_i.

To illustrate this notation, consider the statement $c := a - b$. The values of the variables a and b are used to compute the new value of c. Hence a and b are in the read set. The (old) value of c is not used in the statement, but a new value is defined as a result of the execution of the statement. Hence c is in the write set, but not the read set.

$$\mathbf{R}(c := a - b) = \{a,b\}$$
$$\mathbf{W}(c := a - b) = \{c\}$$

For the statement, $w := c + 1$ the read and write sets are:

$$\mathbf{R}(w := c + 1) = \{c\}$$
$$\mathbf{W}(w := c + 1) = \{w\}$$

The intersection of $\mathbf{R}(S_i)$ and $\mathbf{W}(S_i)$ need not be null. For example, in the statement $x := x + 2$, $\mathbf{R}(x := x + 2) = \mathbf{W}(x := x + 2) = \{x\}$.

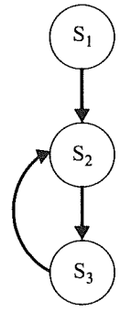

Figure 9.2 Precedence graph with cycles

As another example, consider the statement *read(a)*. Notice that a is being read into, thus its value is changing. The read and write sets are:

$$\mathbf{R}(read(a)) = \{\}$$
$$\mathbf{W}(read(a)) = \{a\}$$

The following three conditions must be satisfied for two successive statements S_1 and S_2 to be executed concurrently and still produce the same result. These conditions were first stated by Bernstein [1966] and are commonly known as *Bernstein's conditions*.

1. $\mathbf{R}(S_1) \cap \mathbf{W}(S_2) = \{\}$.
2. $\mathbf{W}(S_1) \cap \mathbf{R}(S_2) = \{\}$.
3. $\mathbf{W}(S_1) \cap \mathbf{W}(S_2) = \{\}$.

As an illustration, consider S_1: $a := x + y$ and S_2: $b := z + 1$. These two statements can be executed concurrently because

$$\mathbf{R}(S_1) = \{x,y\}$$
$$\mathbf{R}(S_2) = \{z\}$$
$$\mathbf{W}(S_1) = \{a\}$$
$$\mathbf{W}(S_2) = \{b\}$$

However, S_2 cannot be executed concurrently with S_3: $c := a - b$, since

$$\mathbf{W}(S_2) \cap \mathbf{R}(S_3) = \{b\}.$$

9.2 Specification

The precedence graph is a useful device for defining the precedence constraints of the parts of a computation. However, a precedence graph would be difficult to use in a programming language, since it is a two-dimensional object. Other means must be provided to allow the programmer to specify the precedence relations among the various statements in a program.

9.2.1 The Fork and Join Constructs

The **fork** and **join** instructions were introduced by Conway [1963] and Dennis and Van Horn [1966]. They were one of the first language notations for specifying concurrency.

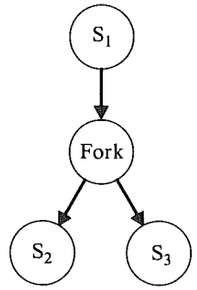

Figure 9.3 Precedence graph for the **fork** construct

The **fork** L instruction produces two concurrent executions in a program. One execution starts at the statement labeled L, while the other is the continuation of the execution at the statement following the **fork** instruction.

To illustrate this concept, consider the following program segment:

```
    S1;
    fork L;
    S2;
    .
    .
    .
L:  S3;
```

Part of the precedence graph corresponding to this program is presented in Figure 9.3. When the **fork** L statement is executed, a new computation is started at S_3. This new computation executes concurrently with the old computation, which continues at S_2.

Note that the execution of a **fork** statement splits one single computation into two independent computations; hence the name **fork**.

The **join** instruction provides the means to recombine two concurrent computations into one. Each of the two computations must request to be joined with the other. Since computations may execute at different speeds, one may execute the **join** before the other. In this case, the computation which executes the join first is terminated, while the second computation is allowed to continue. If there were three computations to be joined, the first two to execute the join are terminated, while the third is allowed to continue.

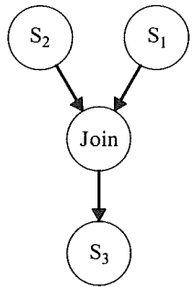

Figure 9.4 Precedence graph for the **join** construct

We need to know the number of computations which are to be joined, so that we can terminate all but the last one. The **join** instruction has a parameter to specify the number of computations to join. Thus the execution of the **join** instruction with a parameter *count* has the following effect:

```
count := count − 1;
if count ≠ 0 then quit;
```

where *count* is a non-negative integer variable, and *quit* is an instruction which results in the termination of the execution. For 2 computations, the variable *count* would be initialized to 2.

The **join** instruction must be executed atomically; that is, the concurrent execution of two **join** statements is equivalent to the serial execution of these two statements, in some undefined order. The importance of this statement will be demonstrated in Section 9.5.

To illustrate this concept, consider the following program segment:

```
        count := 2;
        fork L1;
        .
        .
        .
        S1;
        go to L2;
L1:     S2;
L2:     join count;
```


Part of the precedence graph corresponding to this program is presented in Figure 9.4.

Note that the execution of the **join** statement merges several concurrent executions; hence the name **join**.

Let us further illustrate these concepts through additional examples. Consider again Program 1. To allow the concurrent execution of the first two statements, this program could be rewritten using **fork** and **join** instructions:

```
        count := 2;
        fork L1;
        a := x + y;
        go to L2;
L1:     b := z + 1;
L2:     join count;
        c := a − b;
        w := c + 1;
```

Now return to the precedence graph of Figure 9.1. The corresponding program using the **fork** and **join** instructions is:

```
        S1;
        count := 3;
        fork L1;
        S2;
        S4;
        fork L2;
        S5;
        go to L3;
L2:     S6;
        go to L3;
L1:     S3;
L3:     join count;
        S7;
```

Note that in Figure 9.1 there exists only one **join** node S_7, which has an in-degree of 3. Hence, only one **join** statement is needed. The counter for this **join** is initialized to 3.

As a final example, consider a program that copies from a sequential file f to another file g. By using double-buffering with r and s, this program can read from f concurrently with writing g.

```
var f, g: file of T;
    r, s: T;
    count: integer;
    begin
        reset(f);
        read(f,r);
        while not eof(f)
            do begin
                    count := 2;
                    s := r;
                    fork L1;
                    write(g,s);
                    go to L2;
                L1: read(f,r);
                L2: join count;
                    end;
        write(g,r);
    end.
```

The **fork** and **join** instructions are a powerful means of writing concurrent programs. Unfortunately, programs that use these statements have an awkward control structure. The **fork** instruction is similar to the **go-to** statement in its effect on the point of execution. Much has been said about the undesirable effects of the **go-to** statement. Rather than repeating these, we refer the interested reader to Dijkstra's [1968a] letter.

9.2.2 The Concurrent Statement

A higher-level language construct for specifying concurrency is the **parbegin/parend** statement of Dijkstra [1965a], which has the following form:

parbegin S_1; S_2; ...; S_n **parend**;

Each S_i is a single statement. All statements enclosed between **parbegin** and **parend** can be executed concurrently. Thus the precedence graph which corresponds to the statement above is depicted in Figure 9.5, where S_0 and S_{n+1} are the statements appearing just before and after the **parbegin/parend** statement, respectively. Note that statement S_{n+1} can be executed only after *all* S_i, $i = 1, 2, \ldots, n$, have completed.

Let us illustrate these concepts through additional examples. Consider again Program 1. To allow the concurrent execution of the first

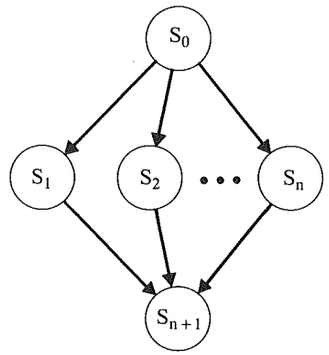

Figure 9.5 Precedence graph for the concurrent statement

two statements, this program could be rewritten using the **parbegin/parend** construct:

```
parbegin
    a := x + y;
    b := z + 1;
parend;
c := a - b;
w := c + 1;
```

We can write the following program to correspond to the precedence graph in Figure 9.1:

```
S1;
parbegin
    S3;
    begin
        S2;
        S4;
        parbegin
            S5;
            S6;
        parend;
    end;
parend;
S7;
```

Finally, let us rewrite the program that copies a file f to another file g, using the concurrent statement.

```
var f, g: file of T;
    r, s: T;
begin
   reset(f);
   read(f,r);
   while not eof(f)
      do begin
            s := r;
            parbegin
               write(g,s);
               read(f,r);
            parend;
         end;
   write(g,r);
end.
```

The concurrent statement is easily added to a modern block-structured higher-level language and exhibits many of the advantages of other structured control statements.

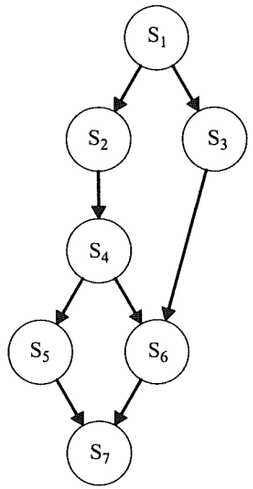

Figure 9.6 Precedence graph with no corresponding concurrent statement

9.2.3 Comparison

Is the concurrent statement powerful enough to model all possible precedence graphs? Unfortunately, the answer is no! To illustrate this point, suppose that we change the graph in Figure 9.1, adding an edge from node S_3 to S_6 (Figure 9.6). Note that the edge from S_3 to S_7 was removed since the addition of an edge from S_3 to S_6 made it redundant. We claim, without proof, that this new graph does not have a corresponding program that uses only the concurrent statement. Try to construct an equivalent program using **parbegin** and **parend**.

In terms of modeling precedence graphs, the **fork/join** construct is more powerful than the concurrent statement. The precedence graph of Figure 9.6, which has no corresponding program using the concurrent statement, has the following program using the **fork/join** construct.

```
        S1;
        count1 := 2;
        fork L1;
        S2;
        S4;
        count2 := 2;
        fork L2;
        S5;
        go to L3;
L1:     S3;
L2:     join count1;
        S6;
L3:     join count2;
        S7;
```

Although the concurrent statement alone is not enough to implement all precedence graphs, other mechanisms (such as the semaphores of Section 9.6) can be added to the concurrent statement to match the power of precedence graphs. In addition, we would not expect *all* possible precedence graphs to need implementing, but only those corresponding to real-world problems. It is not clear that the concurrent statement is insufficient for all real-world problems.

9.3 Review of Process Concept

In the previous section, we introduced precedence graphs and showed how concurrency can be represented within a single program. In this section we formalize the notion of concurrent execution by reintroducing

the sequential process concept. The notion of a process was discussed in more detail in Chapter 4.

Informally, a *sequential process* is a program in execution. We emphasize that a program by itself is not a process; a program is a *passive* entity, while a process is an *active* entity. The execution of a process must progress in a sequential fashion (hence the name *sequential* process). That is, at any point in time at most one instruction is executed on behalf of the process. Thus although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. The relation between the process concept and the precedence graph concept will be explored shortly.

9.3.1 State of a Process

A sequential process may be in one of the following four states:

- **Running**. Instructions are being executed.
- **Blocked**. The process is waiting for some event to occur (such as an I/O completion).
- **Ready**. The process is waiting to be assigned to a processor.
- **Deadlocked**. The process is waiting for some event that will never occur.

The state diagram corresponding to these four states is presented in Figure 9.7. It is essentially the same as the state diagram in Figure 4.7, with the addition of the deadlocked state.

9.3.2 Relation to Precedence Graphs

We have seen that concurrent computation within a single program can be modeled by a precedence graph. We have also presented two different specification notations (**fork/join** and **parbegin/parend**) to describe a concurrent computation. Since we are interested in describing such a computation as a set of sequential processes, we must now relate these two concepts: processes and precedence graphs.

It is convenient to view each node in a precedence graph as a sequential process. In such an environment, processes appear and disappear dynamically during the lifetime of a single program execution. This scheme, however, may result in significant overhead, in terms of the number of processes that need to be created and destroyed. The overhead could be minimized if we collapse those activities that can be carried out sequentially into a single process. This change, however,

needs to be made cautiously, so that we do not reduce the amount of concurrency allowed in a computation. For example, in the precedence graph of Figure 9.1, statements S_2 and S_4 could be collapsed into a single process.

Let us now formally describe the effect of Dijkstra's concurrent statements and the **fork/join** instructions, as they relate to sequential processes. To simplify our discussion, we consider only the **fork/join** constructs. We can always simulate the concurrent statement using the **fork/join** constructs as follows. Let

parbegin S_1; S_2; ...; S_n **parend**;

be a general statement. It can be simulated by:

```
        count := n;
        fork L2;
        fork L3;
             .
             .
             .
        fork Ln;
        S1;
        go to Lj;
L2:     S2;
        go to Lj;
L3:     S3;
        go to Lj;
             .
             .
             .
Ln:     Sn;
Lj:     join count;
```

When process P_i executes the statement **fork** L, a new process P_j is created. P_i and P_j share the same program, as well as any global variables. (It should be clear that in such an environment programs should be written as reentrant code). The main difference between P_i and P_j is that the instruction counter of P_j is set to L and its internal hardware registers are initialized appropriately.

When the **join** *count* instruction is encountered, the value of *count* is decremented by one. If the result is equal to zero, the process continues with its execution. Otherwise, the process terminates.

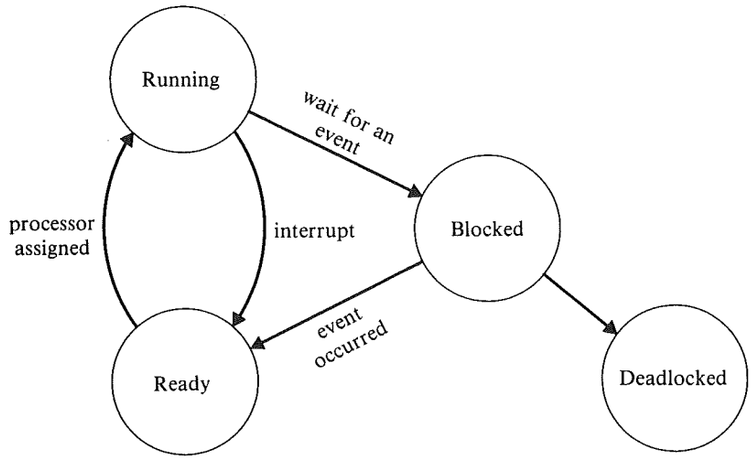

Figure 9.7 State diagram of a process

9.4 Hierarchy of Processes

In this section we shift our attention to the issues of how the various processes are related, and what kind of operations may be invoked on a process. It is convenient to define a new graphical representation of a computation, which we call a process graph.

A *process graph* is a directed rooted tree, whose nodes correspond to processes. An edge from node P_i to node P_j means that P_i created P_j. In this case we shall say that P_i is the parent of P_j, or that P_j is the child of P_i. The graph must be a rooted tree, since each process can have at most one parent, but as many children as it creates (Figure 9.8).

Note the difference between a process graph (which depicts a process creation relation) and a precedence graph (which depicts a precedence relation). In a process graph, an edge from P_i to P_j does not imply that P_j can only execute after P_i, only that P_i created P_j; P_i and P_j may execute concurrently.

9.4.1 Operations on a Process

Given the concept of a process graph, we can now discuss how parent/child relations are formed and the main differences that exist between a parent and a child process.

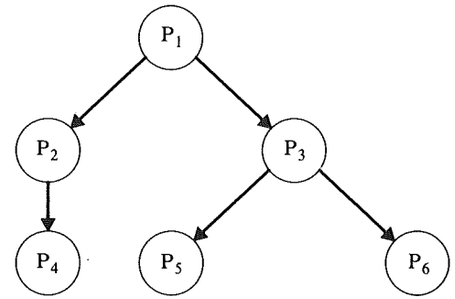

Figure 9.8 Process graph

Process Creation

When one process creates a new process (or processes), by some create operation (such as a **fork**), several possible implementations exist:

1. **Execution**. Concurrent versus sequential.
   a. The parent continues to execute concurrently with its children.
   b. The parent waits until all of its children have terminated.
2. **Sharing**. All versus partial.
   a. The parent and children share all variables in common.
   b. The children share only a subset of their parent's variables.

Let us now briefly elaborate on each of these possibilities.

Option 1a has been adopted in the **fork/join** constructs. When process P_i executes **fork** L, a new process P_j is created, where P_i is the parent of P_j. Both processes continue to execute concurrently.

Option 1b has been adopted in the concurrent statements. When process P_i executes the statement

parbegin S_1; S_2; ...; S_n **parend**;

n new processes are created, all executing concurrently. Process P_i, however, is delayed until all of these processes terminate. Then process

P_i continues with its execution at the statement following the concurrent statement.

Option 2a has been adopted in both the **fork/join** constructs and the concurrent statement. In both schemes, the parent and children share all variables in common.

Option 2b has been adopted in the Unix operating system. When process P_i creates a new process P_j, this new process has an independent memory image of P_i, including access permission to all opened files of P_i. The variables accessible to P_i, however, cannot be accessed by P_j since P_i and P_j have independent memory images. Thus, in Unix, P_i and P_j can only communicate by the use of shared files.

In general, a process will need certain resources (cpu time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, the subprocess may be able to obtain its resources directly from the operating system or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents a process from overloading the system by spawning too many subprocesses.

Process Termination

A process terminates when it finishes executing its last statement. However, there are additional circumstances when termination occurs. A process can cause the termination of another process by issuing the command:

kill *id;*

where *id* is the name of the process to be terminated. The **kill** operation can usually only be invoked by the parent of the process to be terminated. Note that a parent needs to know the identities of its children. Thus when one process creates a new process, the identity of the newly created process is passed to the parent. For example,

id := **fork** *L*

creates a new process (executing at *L*) whose identity is stored in the variable *id*.

A parent may terminate the execution of one of its children for a variety of reasons, such as:

a. The child has exceeded its usage of some of the resources it has been allocated.

b. The task assigned to the child is no longer required.

In order to determine (a), a mechanism must be available to allow the parent to inspect the state of its children.

Many systems do not allow a child to exist if its parent has terminated. In such systems, if process P_j terminates (either normally or abnormally), then all of its children must also be terminated. This phenomenon is referred to as *cascading termination* and is normally initiated by the operating system.

9.4.2 Static and Dynamic Processes

A process that does not terminate while the operating system is functioning is called *static*; a process that may terminate is called *dynamic*. If a system consists of only a bounded number of static processes, then its corresponding process graph is also static; that is, it never changes. Obviously, we must be careful when we define our terms, since initially the graph has no nodes. Rather than worrying about this technicality, let us agree that by *static process graph* we mean one that reaches a static state after some initial "short" period of time.

What are the main advantages and disadvantages of each of these schemes? The difference between the two schemes is similar in many respects to differences between block-structured languages, (such as Algol, PL/1, or Pascal) and languages with static memory allocation (such as Fortran). In the former, blocks/processes appear and disappear dynamically, while in the latter, blocks/processes are fixed at declaration time. The dynamic scheme is more flexible than the static scheme, but it requires more overhead, since process creation and deletion can be quite expensive.

9.5 The Critical Section Problem

In the last section, we developed a model of a system consisting of a number of cooperating sequential processes, all running asynchronously and sharing some data in common. Let us illustrate this model with a simple example that is representative of operating systems.

Producer/consumer processes are quite common in operating systems. A *producer* process produces information that is consumed by a *consumer* process. For example, a line printer driver produces characters

which are consumed by the line printer. A compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce load modules, which are consumed by the loader.

To allow producer and consumer processes to run concurrently, we must create a pool of buffers that can be filled by the producer and emptied by the consumer. A producer can produce into one buffer while the consumer is consuming from another buffer. The producer and consumer must be synchronized, so that the consumer does not try to consume items which have not yet been produced. In this situation, the consumer must wait until an item is produced.

The *unbounded-buffer* producer/consumer problem places no limit on the number of buffers. The consumer may have to wait for new items, but the producer can always produce new items; there are always empty buffers. The *bounded-buffer* producer/consumer problem assumes that there is a fixed number, n, of buffers. In this case, the consumer must wait if all the buffers are empty and the producer must wait if all the buffers are full.

In the following solution to the bounded buffer problem, the shared pool of buffers is implemented as a circular array with two logical pointers: *in* and *out*. The variable *in* points to the next free buffer, while *out* points to the first full buffer. The pool is empty when *in* = *out*; the pool is full when *in*+1 **mod** n = *out*.

The *skip* is a do-nothing instruction. Thus, **while** *condition* **do** *skip* simply tests the condition repetitively until it becomes false.

```
type item = ... ;
   var buffer: array [0..n−1] of item;
       in, out: 0..n−1;
       nextp, nextc: item;
       in := 0;
       out := 0;
       parbegin

          producer: begin
                       repeat
                          ...
                          produce an item in nextp
                          ...
                          while in+1 mod n = out do skip;
                          buffer[in] := nextp;
                          in := in+1 mod n;
                       until false;
                    end;
```



```
          consumer: begin
                       repeat
                          while in = out do skip;
                          nextc := buffer[out];
                          out := out+1 mod n;
                          ...
                          consume the item in nextc
                          ...
                       until false;
                    end;
       parend;
```

This algorithm allows at most $n-1$ buffers to be full 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 a new full buffer is added to the pool and decremented whenever we remove one of the full buffers from the pool. The code for the producer process can be modified as follows:

```
repeat
   ...
   produce an item in nextp
   ...
   while counter = n do skip;
   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 skip;
   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. To illustrate this, suppose that the value of 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 *counter* may be incorrect, as follows. Note that the statement "*counter* := *counter* + 1" is implemented in machine language as:

$$\begin{aligned} register_1 &:= counter; \\ register_1 &:= register_1 + 1; \\ counter &:= register_1 \end{aligned}$$

where $register_1$ is a local cpu register. Similarly, the statement "*counter* := *counter* − 1" is implemented as follows:

$$\begin{aligned} register_2 &:= counter \\ register_2 &:= register_2 - 1; \\ counter &:= register_2 \end{aligned}$$

where again $register_2$ is a local cpu register. Even though $register_1$ and $register_2$ may be the same physical registers (an accumulator, say), remember that the contents of this register will be saved and restored by the cpu scheduling routines: the interrupt handler and the dispatcher (Section 4.2.3).

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

T_0:	*producer*	**execute**	$register_1 := counter$	$\{register_1 = 5\}$
T_1:	*producer*	**execute**	$register_1 := register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	**execute**	$register_2 := counter$	$\{register_2 = 5\}$
T_3:	*consumer*	**execute**	$register_2 := register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	**execute**	$counter := register_1$	$\{counter = 6\}$
T_5:	*consumer*	**execute**	$counter := register_2$	$\{counter = 4\}$

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 reverse the order of the statements at T_4 and T_5 we would arrive at the incorrect state "*counter* = 6."

We may arrive at this incorrect state because we allowed both processes to manipulate the variable *counter* concurrently. (Note that executing these two statements concurrently violates Bernstein's conditions.) In order to remedy this difficulty, we need to ensure that only one process at a time may be manipulating the variable *counter*. This observation leads us to the *critical section problem*.

9.5.1 Problem Definition

Consider a system consisting of n cooperating processes $\{P_1, P_2, \ldots, P_n\}$. Each process has a segment of code, called a *critical section*, in which the process may be reading 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 which the processes may 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 mutual exclusion problem must satisfy the following three requirements:

a. **Mutual Exclusion**. If process P_i is executing in its critical section then no other process can be executing in its critical section.

b. **Progress**. If no process is executing in its critical section and there exists some process that wishes to enter its critical section, then only those processes that are not executing in their remainder section can participate in the decision as to who will enter the critical section next, and this selection cannot be postponed indefinitely.

c. **Bounded Waiting**. There must 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.

It is assumed that each process is executing at a non-zero speed. However, no assumption can be made concerning the *relative* speed of the n processes.

In Sections 9.5.2 and 9.5.3 we work up to solutions to the critical section problem which satisfy the three requirements stated above, and which do not rely on any assumptions concerning the hardware instructions or the number of processors the hardware supports. It is, however, assumed 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 simultaneously, the result is equivalent to their sequential execution in some unknown order. Thus, if a load and a store are executed simultaneously, the load will either get the old value or the new value, but not some combination of the two.

In Section 9.5.4 we present some simple hardware instructions that are available on many systems, and show how they can be effectively utilized in solving the critical section problem.

9.5.2 Two-Process Software Solutions

In this section we trace the initial attempts made in trying to develop algorithms for ensuring mutual exclusion. We restrict our attention to algorithms that are applicable to only two processes at a time. In Section 9.5.3 we deal with the more general problem of n processes. The processes are numbered P_0 and P_1 and the general structure of the problem is:

```
begin
    common variable declarations;
    parbegin
        P0;
        P1;
    parend;
end.
```

For brevity, when presenting the algorithm, we define only the common variables and describe process P_i. For convenience, we use P_j to denote the other process; that is, $j = 1 - i$.

The general structure of process P_i is presented below. The *entry section* and *exit section* are enclosed in boxes to highlight the important segments of code.


```
repeat
    entry section;

        critical section

    exit section;

        remainder section

until false;
```

Algorithm 1

Our first approach is to let the processes share a common integer variable *turn* initialized to 0 (or 1). If $turn = i$, then process P_i is allowed to execute in its critical section.

```
repeat
    while turn ≠ i do skip;

        critical section

    turn := j;

        remainder section

until false;
```

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 P_1 wants to enter its critical section, it cannot do so, even though P_0 may be in its remainder section.

Algorithm 2

The problem with Algorithm 1 is that it fails to remember the state of each process, but remembers only which process is allowed to enter its critical section. To remedy this problem, we can replace the variable *turn* with the following array.

var *flag*: **array** [0..1] **of** *boolean;*

The elements of the array are initialized to *false*. If *flag*[*i*] is *true*, then process P_i is executing in its critical section.

The general structure of process P_i would be:

repeat

while *flag*[*j*] **do** *skip;*
flag[*i*] := *true;*

critical section

flag[*i*] := *false;*

remainder section

until *false;*

Here we first check if the other process is in its critical section (*flag*[*j*] = *true*) and if so, we wait. Then we set our *flag*[*i*] to be *true* and enter our critical section. When we leave our critical section, we reset our *flag* to be *false*, allowing the other process into its critical section if it was waiting.

This algorithm does not ensure that only one process at a time will be executing in its critical section. For example, consider the following execution sequence:

T_0: P_0 enters the **while** statement and finds *flag*[1] = *false*.

T_1: P_1 enters the **while** statement and finds *flag*[0] = *false*.

T_2: P_1 sets *flag*[1] = *true* and enters the critical section.

T_3: P_0 sets *flag*[0] = *true* and enters the critical section.

We now have arrived at a state where P_0 and P_1 are both in their critical sections, violating the mutual-exclusion requirement.

This algorithm is crucially dependent on the exact timing of the two processes. The sequence above could have been derived in an environment where there are several processors executing concurrently, or where an interrupt (such as a timer interrupt) has occurred immediately after step T_0 was executed (and another interrupt after T_2), and the cpu is switched from one process to another.

Algorithm 3

The problem with Algorithm 2 is that process P_i made a decision concerning the state of P_j before P_j had the opportunity to change the state of the variable *flag*[*j*]. We can try to correct this problem. As in Algorithm 2, we still maintain the array *flag*. This time, however, the setting of *flag*[*i*] = *true* indicates only that P_i *wants* to enter the critical section.

repeat

flag[*i*] := *true;*
while *flag*[*j*] **do** *skip;*

critical section

flag[*i*] := *false;*

remainder section

until *false;*

So, in this algorithm, we first set our *flag*[*i*] to be *true*, signaling that we want to enter our critical section. Then we check that the other process does not also want to enter its critical section. We wait if so. Then we enter our critical section. When we exit the critical section, we set our *flag* to be *false*, allowing the other process (if it is waiting) to enter its critical section.

In this solution, unlike Algorithm 2, the mutual-exclusion requirement is satisfied. Unfortunately, the progress requirement is not met. To illustrate this problem, consider the following execution sequence.

T_0: P_0 sets *flag*[0] = *true*.
T_1: P_1 sets *flag*[1] = *true*.

Now P_0 and P_1 are looping forever in their respective **while** statements.

Algorithm 4

By now the reader is probably convinced that there is no simple solution to the critical section problem. It appears that every time we fix one bug in a solution, another bug appears. However, we now (finally) present a correct solution, due to Peterson [1981]. This solution is basically a combination of Algorithm 3 and a slight modification of Algorithm 1.

The processes share two variables in common:

```
var flag: array [0..1] of boolean;
    turn: 0..1;
```

Initially *flag*[0] = *flag*[1] = *false* and the value of *turn* is immaterial (but either 0 or 1). The structure of process P_i is:

```
repeat

    flag[i] := true;
    turn := j;
    while (flag[j] and turn=j) do skip;

        critical section

    flag[i] := false;

        remainder section

until false;
```


To enter our critical section, we first set our *flag*[*i*] to be *true*, and assert that it is the other process' turn to enter if it wants to (*turn* = *j*). 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 be immediately overwritten. The eventual value of *turn* decides which of the two processes is allowed to enter its critical section first.

We now prove that Peterson's solution is correct. To do so we need to show that (a) mutual exclusion is preserved, (b) the progress requirement is satisfied, and (c) the bounded-waiting requirement is met.

To prove property (a) we note that each P_i enters its critical section only if either *flag*[*j*] = *false* or *turn* = *i*. Also note that if both processes could be executing in their critical sections at the same time then *flag*[0] = *flag*[1] = *true*. These two observations imply that P_0 and P_1 could not have successfully executed their *while* statement at about the same time, since the value of *turn* can be either 0 or 1, but not both. Hence, one of the processes, say P_j, must have successfully executed the *while* statement, while P_i had to at least execute one additional statement "*turn* = *j*". However, since at that point in time *flag*[*j*] = *true*, and *turn* = *i*, and this condition will persist as long as P_j is in its critical section, the result follows: mutual exclusion is preserved.

To prove properties (b) and (c), we note that a process P_i 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 is the only loop. If P_j is not interested in entering the critical section, then *flag*[*j*] = *false* and P_i can enter its critical section. If P_j has set *flag*[*j*] = *true* and is also executing in its *while* statement, then either *turn* = *i* or *turn* = *j*. If *turn* = *i*, then P_i will enter the critical section. If *turn* = *j*, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset *flag*[*j*] to *false* allowing P_i to enter its critical section. If P_j should reset *flag*[*j*] to *true*, it must also set *turn* = *i*. Thus, since P_i does not change the value of the variable *turn* while executing the *while* statement, P_i will enter the critical section (progress) after at most one entry by P_j (bounded-waiting).

9.5.3 N-Process Software Solutions

We have seen that Peterson's solution solves the critical section problem for two processes. Now let us develop an algorithm for solving the critical section problem for *n* processes. Algorithm 5 is due to Eisenberg and McGuire [1972] while Algorithm 6 is due to Lamport [1974].