CHAPTER 2

# Basic Concepts: Processes and Their Interactions



## 2.1 THE PROCESS NOTION

When developing a large and complex system, it is of utmost importance to use some structuring method to guide and manage the design. The system must be decomposed into manageable software parts and subsystems that perform well-defined functions and interact with one another via well-defined interfaces. Many such systems, particularly operating systems (OSs), have additional challenges due to their high degree of nondeterminism and their logical and physical parallelism.

Consider first the presence of nondeterminism: Many functions or services are invoked in response to particular events. These may occur at unpredictable times and with a varying degree of frequency. We can order the events informally according to their typical frequency of occurrence. At the highest level are requests from communication and storage devices needing attention from the CPU. The next frequency level consists of requests for resources such as physical devices, blocks of memory, or software components. Finally, at the lowest of the three levels, we have commands entered by interactive users or the operator via terminals, and possible hardware or software errors. All these events typically arrive at unpredictable times and may require that different modules or subsystems of the OS be invoked.

An OS also must handle a high degree of parallelism. The parallel activities causing the nondeterministic events, the parallel execution of user and OS programs, and the parallel operation of computer components.

The notion of a **sequential process** is introduced as a way to cope elegantly with the structuring problem, the highly nondeterministic nature of the environment, and the parallel activities. Informally, a sequential process (sometimes also called **task**) is the activity resulting from the execution of a program by a sequential processor (CPU). A process consists of a **program** and a **data area**, both of which reside in main memory. It also contains a **thread of execution**, which is represented by the program counter and the stack. The program counter points at the currently executing instruction and the

stack captures the sequence of nested function invocations. Some systems allow multiple threads of execution to exist within a single process. Each such thread is represented by its own program counter and stack.

Conceptually, each thread of execution has its own CPU and main memory. In reality, the number of physical CPUs is much smaller than the number of processes or threads ready to run. Many computers are equipped with only a single CPU. Thus, many processes are forced to share the same CPU. To treat each process as an autonomous unit of execution, the details of CPU sharing must be invisible to the processes. This is accomplished by the lowest level of the operating system, usually referred to as the **kernel**; a principal task of the kernel is to "virtualize" the CPU, i.e., to create the illusion of a separate CPU for each running process. The kernel also may provide a separate storage—a virtual memory—for each process. Under these assumptions, each process may be viewed in isolation; it interacts with other processes only via a limited number of primitives provided by the kernel.

In systems with only one processor, the achieved concurrency among processes is a *logical* one since only one process may be executing at any given time. In the case of multiprocessors or multicomputers where more than one processing element is dedicated to general computation, or in systems equipped with specialized I/O coprocessors, *physical* concurrency is also possible.

Regardless of whether the concurrency is logical or physical, the OS and the user applications are viewed as a collection of processes, all running concurrently. These processes operate almost independently of one another, *cooperate* by sharing memory or by sending messages and synchronization signals to each other, and *compete* for resources. Each process is dedicated to a specific function, and its interactions with other processes are limited to only a few well-defined interfaces.

The process notion is invaluable for addressing the problems of distributing computation to the available processors. In both uniprocessor and multiprocessor environments, each process is treated as an autonomous entity, and each of its threads may be scheduled for execution on a CPU.

By examining the logic of processes and ignoring the number of physical processors and the details of physical memory allocation, it is possible to develop hardware-independent solutions to several systems and application problems. The solutions will ensure that a system of processes cooperate correctly, regardless of whether or not they share physical resources. The process model has several other implications in OSs. It has permitted the isolation and specification of many primitive OS tasks, has simplified the study of the organization and dynamics of an OS, and has led to the development of useful design methodologies.

In summary, the process concept is one of the fundamental notions in OSs. Using the idea of processes and its variants, both user applications and OSs consist of logically and physically concurrent programs. This chapter covers the basic methods, operations, and issues of concurrent programming with processes, with special emphasis on applications to OSs. For the purposes of this chapter, we will use the term *process* to refer primarily to its thread of execution. Thus creating a new process means starting a new thread of execution, with its own program counter and stack. Depending on the implementation, the new thread may share the memory with its creator, or it may get its own copy of the program and data areas. The details of implementing processes and threads—which is one of the main functions of an OS kernel—are treated in Chapter 4.

## 2.2 DEFINING AND INSTANTIATING PROCESSES

Traditionally, only the OS itself was permitted (and able) to create new processes. It was assumed that application programmers had neither the need nor the expertise to handle concurrency. This view has changed in modern computer systems. Typically, users may now employ sophisticated process mechanisms to create their own subsystems consisting of many concurrent processes.

Depending on the sophistication of the system, process creation may be done either *statically* or *dynamically*. In the first case, all processes are predeclared and activated when the system begins execution. More flexibility is attained when processes may be spawned (and terminated) dynamically during execution. The most advanced systems treat processes as first-class programming elements, similar to functions, procedures, modules, data types, objects, and classes.

### 2.2.1 Precedence Relations Among Processes

Programming constructs for creating and terminating processes should be able to implement a variety of precedence relations. These relations define when processes start and stop executing relative to one another. For example, each interactive user in a simple system might start a session with an initialization or login process followed in sequence by a command interpreter process; during the session a mail process might run concurrently with a date and time process, and with the command interpreter.

Figure 2-1 illustrates some of the precedence constraints that are possible among processes. The execution of a process $p_i$ is represented by a directed edge of a graph. Each graph in the figure denotes an execution-time trace of a set of processes, and the graph connectivity describes the start and the finish precedence constraints on the processes. These graphs will be called **process flow graphs**. Any directed acyclic graph may be interpreted as a process flow graph.

An important class of process flow graphs are those that are properly nested. Let $S(p_1, \ldots, p_n)$ denote the serial execution of processes $p_1$ through $p_n$ and let $P(p_1, \ldots, p_n)$ denote the parallel execution of processes $p_1$ through $p_n$. Then a process flow graph is *properly nested* if it can be described by the functions $S$ and $P$, and only function composition.[1]

---

**EXAMPLES: Process Flow Graphs**

1. The graphs in Figure 2-1a, b, and c, respectively, can be described by the following expressions:

$$S(p_1, p_2, p_3, p_4),$$
$$P(p_1, p_2, p_3, p_4), \text{ and}$$
$$S(p_1, P(p_2, S(p_3, P(p_4, p_5)), p_6), P(p_7, p_8)).$$

2. *Evaluation of arithmetic expressions*. Many subexpressions of arithmetic expressions are independent and can be evaluated in parallel; the amount of parallelism

[1] This property is very similar to the "proper nesting" of block structure in programming languages and of parentheses within expressions.

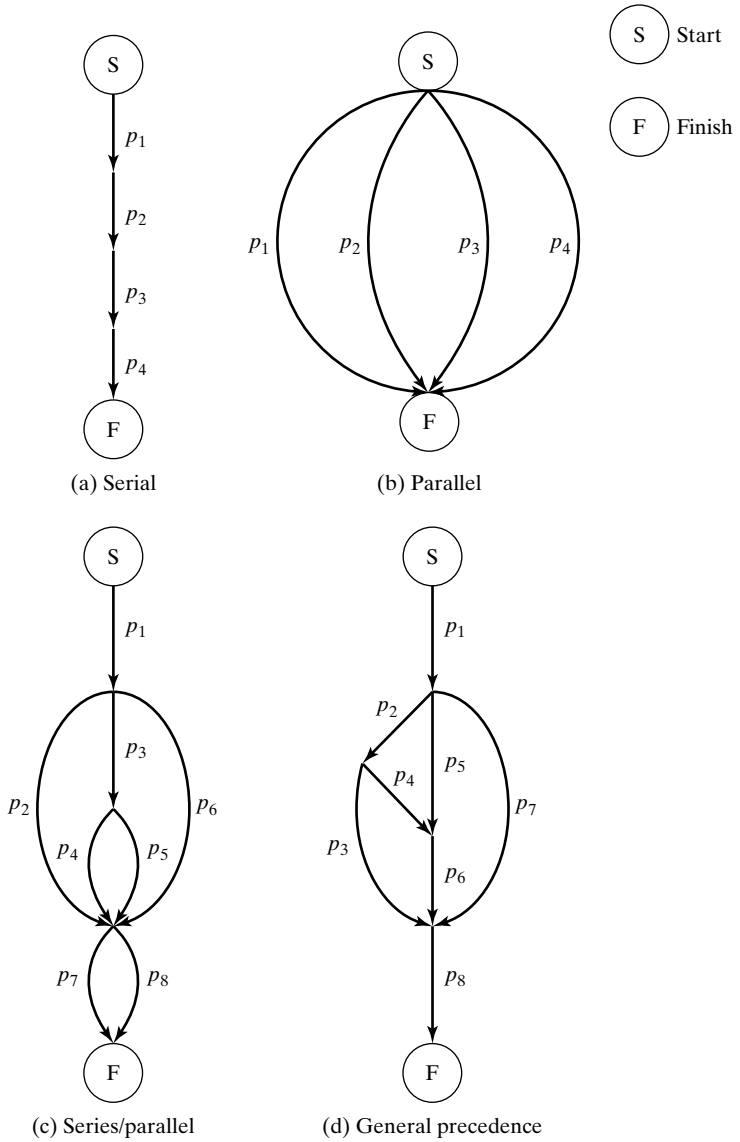


FIGURE 2-1. Precedence relations among processes: (a) serial; (b) parallel; (c) serial/parallel; and (d) general precedence.

that can occur is limited by the depth of the expression tree. Figure 2-2 shows an example of an arithmetic expression and the corresponding expression tree. (Edges are labeled by the code executed by the corresponding process). Many problems in which the primary data structure is a tree can be logically described in terms of parallel computations.

**3.** *Sorting*. During the $i$th pass in a standard two-way merge sort, pairs of sorted lists of length $2^{i-1}$ are merged into lists of length $2^i$; each of the merges can be

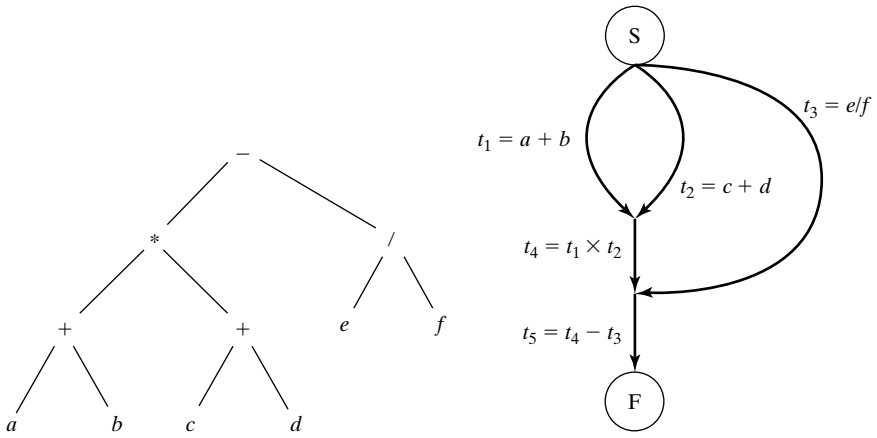


FIGURE 2-2. Process flow graph.

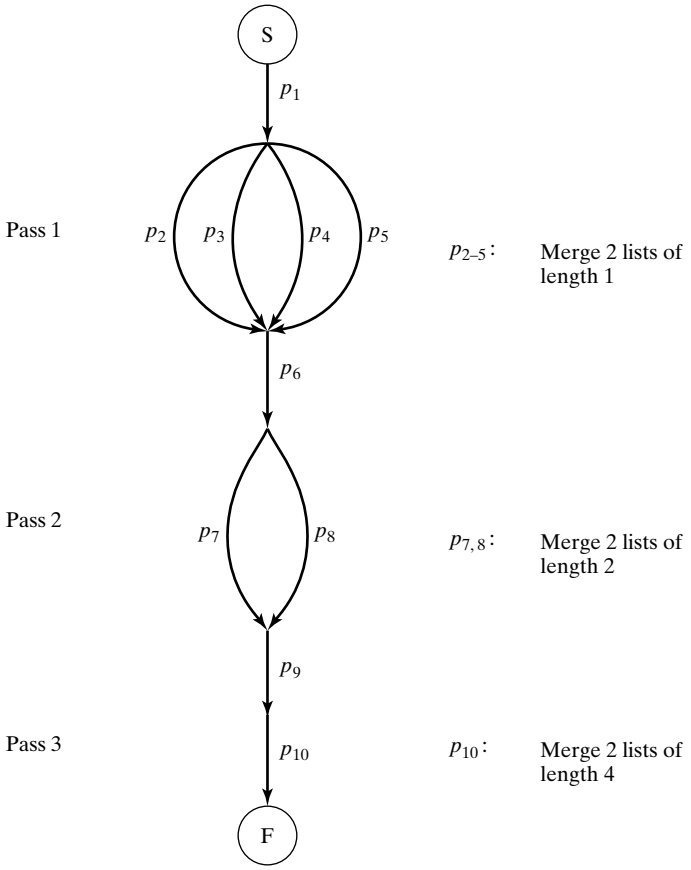


FIGURE 2-3. Merge-sort of a list of eight elements.

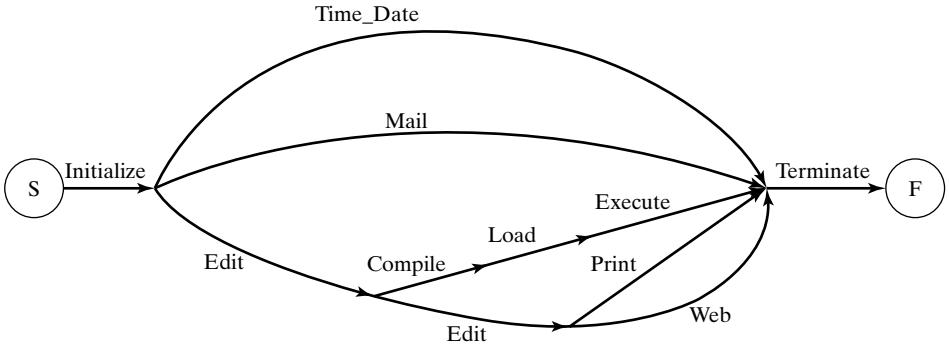


FIGURE 2-4. User session at a workstation.

performed in parallel within a pass (Fig. 2-3). This kind of parallelism where each of the processes executes the same code but on a different portion of the data is termed **data parallelism** (see example 1 on page 46).

4. *Workstation interactions*. Figure 2-4 illustrates the relations among a set of processes created during a typical user session. After initialization, three processes are established in parallel: one that continually displays the time and date, a second that permits sending and reading mail, and a third for editing. On termination of the first edit process, another edit process is created in parallel with a sequence of three processes: compile, load, and execute; the second edit is followed by a print and Web search.

---

The general precedence graph of Figure 2-1d is *not* properly nested. We prove this by first observing that any description by function composition must include at the innermost level an expression of the form $S(p_{i_1}, \ldots, p_{i_n})$ or $P(p_{i_1}, \ldots, p_{i_n})$ for $n \geq 2$ and $1 \leq i_j \leq 8$ ($1 \leq j \leq n$). The expression $P(p_{i_1}, \ldots, p_{i_n})$ cannot appear, since Figure 2-1d does not contain any subgraph of this form (i.e., a graph similar to Fig. 2-1b.) The expression $S(p_{i_1}, \ldots, p_{i_n})$ also cannot appear for the following reason. All serially connected processes $p_i$ and $p_j$ have at least one other process $p_k$ that starts or finishes at the node between $p_i$ and $p_j$; thus for any $S(\ldots, p_i, p_j, \ldots)$ in the expression, the connection of $p_k$ to $p_i$ or $p_j$ could not be described without repeating $p_i$ or $p_j$ in a separate expression. This is not allowed, since the process would have to appear in the graph twice. Therefore, $S(p_{i_1}, \ldots, p_{i_n})$ cannot appear in the expression, and a properly nested description is not possible. Consequently, to describe this more general interaction pattern among processes, another mechanism is necessary.

### 2.2.2 Implicit Process Creation

By convention, program statements that are separated by semicolons or controlled by a loop iterator are executed in sequence. To specify possible parallel execution of blocks of code, we must introduce other separator or controlling symbols. Each sequential code section within a parallel construct then can be treated naturally as a process.

### Specifying Parallelism with the *cobegin // coend* Constructs

The *cobegin* and *coend* primitives, originally called *parbegin* and *parend* (Dijkstra 68), specify a set of program segments that may be executed concurrently. They are used in conjunction with the separator "//" to describe concurrency. The general form is:

```
cobegin C1 // C2 // ...// Cn coend
```

where each *Ci* is a block of code. It results in the creation of a separate concurrent computation for each *Ci*, executing independently of all other processes within the *cobegin-coend* construct. Nesting of sequential and parallel blocks is normally permitted so that any *Ci* could contain other *cobegin-coend* statements.

Note that, although the *specification* of the concurrent computations is *explicit*, the actual *creation* of the corresponding underlying processes is *implicit*; it is up to the system to decide how and when to create these.

The *cobegin-coend* primitives correspond in a straightforward way to the $S$ and $P$ functions defined in Section 2.2.1 for describing properly nested process flow graphs. The above *cobegin-coend* construct corresponds to the following expression:

$$P(C_1, C_2, \ldots C_n)$$

Similarly, a sequential code segment,

C1; C2; . . . ; Cn;

can be expressed as:

$$S(C_1, C_2, \ldots C_n)$$

---

**EXAMPLE: Use of cobegin/coend**

Consider again the workstation example of the last section (Fig. 2-4). Assuming that code segments are represented by their corresponding process names, the system can be specified by the following program:

```
Initialize;
     cobegin
     Time_Date // Mail //
          { Edit; cobegin
                  { Compile; Load; Execute } //
                  { Edit; cobegin Print // Web coend }
                  coend
          }
     coend;
Terminate
```

---

**CASE STUDY: OCCAM/CSP**

A construct semantically similar to *cobegin-coend* is used in Occam, a concurrent language based on the distributed programming language CSP (Hoare 1978). In Occam 2 (e.g., Burns 1988), the keyword *PAR* denotes that the subsequent list of code blocks may be executed in parallel; similarly, *SEQ* precedes a sequential list of statements. Indentation is used for grouping purposes. The arithmetic expression computation in Figure 2-2 can be written as:

```
SEQ
   PAR
      SEQ
         PAR
            t1 = a + b
            t2 = c + d
         t4 = t1 * t2
      t3 = e / f
   t5 = t4 - t3
```

### Data Parallelism: The *forall* Statement

Another natural form of concurrency is a parallel analog of iteration. It occurs when the same body of code is executed on different parts of a data structure, usually an array—hence, the name *data parallelism*. The merge sort example of Section 2.2.1 is typical. This type of concurrency occurs frequently in scientific computation.

We illustrate the idea with a *forall* construct, modeled loosely after the *FORALL* statement defined in the current version of ISO Fortran and used in an earlier version of the high performance Fortran language (HPF) (e.g., Andrews 2000). The syntax of the *forall* statement is:

```
forall ( parameters ) statements
```

where the *parameters* field specifies a set of data elements, typically through a list of indices, each of which is bound to a separate concurrent instantiation of the code given by *statements*—the loop body. Each such code instance corresponds implicitly to a concurrent process.

## EXAMPLES: Merge-Sort and Matrix Multiplication

1. *Merge-Sort*. Consider again the merge sort described in Section 2.2.1 (Fig. 2-3). Let there be $2^k$ elements in the list to be sorted. This gives rise to $k$ passes in a two-way merge sort. During each pass $i$, $2^{k-i}$ merging processes can be created and executed in parallel. Following is the code for the main part of a concurrent program for the sort. It assumes the existence of a function *Merge(s,n)*, which merges two lists of size $s$; the first list starts at index $n$ and the other starts at index $n + s$:

```
for ( i=1; i<=k; i++ ) {
    size = pow(2, i-1 );
```

```
        forall ( j:0..(pow(2, k-i)-1) )
          Merge(size, j*size*2);
    }
```

The notation $(j : 0..(pow(2, k-i)-1)$ binds $j$ to each element in the set $\{0, \ldots, (pow(2, k-i)-1)\}$.

**2.** *Matrix multiplication*. On performing the matrix multiplication $A = B \times C$, all elements of the product $A$ can be computed simultaneously. Let $B$ be an $n$ by $r$ matrix and $C$ be a $r$ by $m$ matrix. Then, the product can be computed concurrently by the code:

```
forall ( i:1..n, j:1..m ) {
       A[i][j] = 0;
       for ( k=1; k<=r; ++k )
           A[i][j] = A[i][j] + B[i][k]*C[k][j] ;
}
```

### 2.2.3 Explicit Process Creation with *fork* and *join*

The primitives *fork* and *join* provide a more general means for explicitly spawning parallel activities in a program. They may also be used in a straightforward way to implement the *cobegin-coend* and *forall* constructs, used to denote independent and thus potentially parallel computations. We will start with the early, low-level form (Conway 1963; Dennis and Van Horn 1966), and then discuss several recent versions supported by contemporary systems.

A process $p$ executing the instruction:

```
fork x
```

creates a new process $q$, which starts executing at the instruction labeled $x$; $p$ and $q$ then execute concurrently. The instruction:

```
join t,y
```

decrements $t$ by 1 and, if $t$ equals zero, transfers control to the location $y$. The following statements describe the effect of the *join* instruction:

```
t = t-1;
if (t==0) goto y;
```

The execution of the *join* instruction must be *atomic*. That is, the two statements must be executed as a single indivisible instruction without interruption and interference from any other process.

In addition to the *fork* and *join* instructions, the instruction *quit* is provided, which allows a process to terminated itself.

To illustrate the use of *fork/join/quit*, consider the following program segment for evaluating the expression of Figure 2-2:

```
        n = 2;
        fork L3;
        m = 2;
        fork L2;
        t1 = a+b; join m,L4; quit;
L2:     t2 = c+d; join m,L4; quit;
L4:     t4 = t1*t2; join n,L5; quit;
L3:     t3 = e/f; join n,L5; quit;
L5:     t5 = t4-t3;
```

This program first forks two child processes, $L3$ and $L2$, to compute the values of $t3$ and $t2$, respectively. In the meantime, the parent process computes the value of $t1$. It then joins with the child process $L2$ as follows. Both execute the same *join* statement. The process that executes the *join* first, finds $n$ equal to 1 (after decrementing it) and terminates itself by executing the *quit* statement following the *join*. The other process finds $n$ equal to 0 (after decrementing it) and continues executing at $L4$. It then joins with $L3$ in an analogous manner, and the survivor of the join executes the final statement at $L5$.

To create private or local copies of variables within parallel processes, variables may be declared as "private" to a process by the declaration:

```
private: x1,x2, ..., xn;
```

The variables $xi$ then only exist for the process executing the private declarations; in addition, any new process created by the latter (using a *fork*) will receive its own copy of the private variables of the parent process.

To illustrate the use of private variables, consider the following example from image processing. We are given an array $A[0..n+1, 0..n+1]$ consisting of zeros and ones representing a digitized image. It is desired to "smooth" the image by replacing each interior point $A[i, j]$ by 1 if the majority of $A[i, j]$ and its immediate eight neighbors are 1, and by 0 otherwise. This procedure is called *local averaging* and is logically a parallel computation. We assume that the smoothed image is stored in a new array, $B$.

```
Local_average( int A[][], int n, int B) {
int t,i,j; private: i,j;
    t = n*n;
    for (i=1; i<=n; ++i)
    for (j=1; j<=n; ++j)
        fork e;
    quit;
e:  if (A[i-1][j-1]+A[i-1][j]+A[i-1][j+1]
       +A[i][j-1]+A[i][j]+A[i][j+1]+A[i+1][j-1]
       +A[i+1][j]+A[i+1][j+1] >= 5) B[i][j] = 1;
    else B[i][j] = 0;
    join t,r;
    quit;
r:  return;
}
```

The main virtue of the statements *fork, join* and *quit* is their high expressive power. They are sufficient to describe any process flow graph. Below is a program for the graph of Figure 2-1d, which is not properly nested:

```
     t1 = 2; t2 = 3;
     p1; fork L2; fork L5; fork L7; quit;
L2:  p2; fork L3; fork L4; quit;
L5:  p5; join t1,L6; quit;
L7:  p7; join t2,L8; quit;
L3:  p3; join t2,L8; quit;
L4:  p4; join t1,L6; quit;
L6:  p6; join t2,L8; quit;
L8:  p8; quit;
```

The main disadvantages of these primitives are that they are low level and may be used indiscriminately anywhere within a program. When invoked inside of loops or other control statements, the program structure may become rather obscure. However, various higher-level forms and derivatives of *fork* and *join* are implemented in many OSs and languages. The following five examples are representative.

● Au: The words 'fork' and 'join' appear both in normal font as well as in computer font in the text. Pl. clarify which of these font need to be used for consistency.

### CASE STUDY: PROCESS CREATION WITH FORK and JOIN

**1.** *UNIX fork*. The UNIX OS employs a variant of the *fork* primitive that has the form:

```
procid = fork()
```

Execution of this statement causes the current process, called the *parent*, to be simply replicated. The only distinction between the parent and the newly created process, called *child*, is the variable *procid*; in the parent it contains the process number of the child as its value, whereas in the child its value is zero. This permits each of the two processes to determine its identity and to proceed accordingly. Typically, the next statement following *fork* will be of the form:

```
if (procid==0) do_child_processing;
else do_parent_processing;
```

One of the processes, say the child, may overlay itself by performing the call *exec()* as part of the *do_child_processing* clause. This function specifies a new program as one of its parameters; the new program then continues executing as the child process.

**2.** *Linux clone*. The Linux OS provides a system call:

```
procid = clone(params)
```

This is similar to the UNIX *fork* but the parameters supplied to the function allow the caller to control what is shared between the caller (the parent) and

●Au: The words 'fork' and 'join' appear both in normal font as well as in computer font in the text. Pl. clarify which of these font need to be used for consistency.

**CASE STUDY: PROCESS CREATION WITH `FORK` and `JOIN` (*continued*)**

the child process. In particular, the child can get its own copy of the program and data space. In this case, the *clone* functions as the UNIX *fork*. Alternately, the child can share its program and data space with the parent. In this case, the child becomes only a new thread within the same parent process.

**3.** *UNIX/Linux join*. Both UNIX and Linux provide a construct similar to a *join*:

```
procid = waitpid(pids, *stat_loc, ...)
```

The calling process specifies the identifications of one or more of its children in the parameter *pids*. The *waitpid* call then suspends the calling process until one of the specified children terminates (using the call *exit()*, which corresponds to the statement *quit*.) Upon awakening, the parent process may examine the child's status in the variable **stat_loc*. The difference between *waitpid* and *join* is that the latter is symmetric—two or more processes execute the same *join* statement, and all may continue executing after the *join*. With *waitpid*, only one process—the parent—may wait and then continue, whereas the child must always terminate.

**4.** *Orca*. In the examples 1 and 2 above, the *fork* and *clone* statements were part of the OS interface. Frequently, such primitives are incorporated directly in a programming language. One interesting example is the Orca parallel programming language (Bal, Kaashoek, and Tanenbaum 1992). In Orca, a new process can be generated with the statement:

```
fork pname(params) on cpu;
```

*pname* is the name of a previously declared process, *params* are any parameters associated with the process, and *cpu*, which is optional, identifies the machine on which the new forked process is to run.

**5.** *Java join*. The concurrency facility offered by the Java language (Naughton and Schildt 1997) has a *join* method for synchronizing with the termination of a thread. The call:

```
thread_id.join();
```

causes the caller to wait until the thread named *thread_id* completes before it continues execution.

**6.** *Mesa fork/join*. The Mesa language (Lampson and Redell 1980) is an influential concurrent programming language that has been used for implementing OSs. It permits *any* procedure $q(\ldots)$ to be invoked as a separate process. The two statements used to spawn and coordinate concurrent processes have the following form:

```
p = fork q(...);
var_list = join p;
```

The *fork* statement creates a new process which begins executing the procedure *q* concurrently with the parent process. In Mesa, each process is considered an *object*, and it may be assigned to a variable. In the above statement, the variable *p* represents the child process. Note that, unlike the *procid* variable in UNIX, the variable *p* contains not just a process identifier but also the process object itself. This permits processes to be treated as any other variable; for example, a process may be passed to another procedure as a parameter.

To synchronize the termination of a child process with the parent's computation, the *join* primitive is used. It forces the parent process to wait for the termination of the child process *p*. Each procedure in Mesa must explicitly specify a list of values to be returned as the result of the procedure's computation. When a child process terminates, the results of the corresponding procedure may be transmitted to the parent process by assigning them to variables listed as part of the *join* statement. In the above example, the results returned from the procedure *q* executed by the child process *p* are assigned to the variables listed in *var_list* upon *p*'s termination. At that point, the parent process resumes its execution with the next instruction following the *join*.

### 2.2.4 Process Declarations and Classes

Some languages have combined the virtues of both *fork-join* and *cobegin-coend* by providing mechanisms that designate segments of code to be separate processes or classes of separate processes, but permit their invocation and instantiation to be controlled during execution. A process is declared using the following syntax:

```
process p {
    declarations_for_p;
    executable_code_for_p;
}
```

The keyword *process* designates the segment of code between the curly braces as a separate unit of execution named *p*. The list of declarations preceding the executable code typically contains local variables, functions, procedures, and other process declarations. As soon as the process *p* is activated, all processes defined within the declaration portion of *p* are activated as well. This mechanism represents a *static* creation of processes.

To provide more flexibility, processes may be created *dynamically* by replacing the keyword *process* by the phrase *process type*. This can be interpreted as defining a process class or template, instances of which may be created dynamically during execution by using a special command *new*. The following program skeleton illustrates the distinction between static and dynamic process creation:

```
process p {
    process p1 {
        declarations_for_p1;
        executable_code_for_p1;
    }
```

```
        process type p2 {
            declarations_for_p2;
            executable_code_for_p2;
        }
        other_declarations_for_p;
        ...
        q = new p2;
        ...
}
```

The process *p1* is declared statically within the declaration part of *p* and will be activated as soon as the process *p* begins execution. The process *p2*, on the other hand, is declared as a process type, and an instance of *p2* will be created only as a result of the explicit call *new p2*, performed during the execution of the body of process *p*. The new instance of *p2* will be named *q*.

**CASE STUDY: PROCESS DECLARATIONS AND CLASSES**

1. *Ada*. The Ada language (Ada 1995) supports both static and dynamic process creation as described above. The processes in Ada are called *tasks*.
2. *Java*. The Java language supports concurrency through a class named *Thread*. The Java *join* method, discussed in Example 5 of Section 2.2.3, is part of this class. A Java thread can be created or instantiated using the *new* primitive, which is similar to the one discussed above. To start the actual execution of a created thread—to put the thread on a ready list for the CPU scheduler of the OS—the *start* method may be used, for example:

```
thread_id.start();
```

Process declarations permit any number of processes to be declared and spawned at arbitrary times and in arbitrary order during execution. However, they do not permit processes to interact with one another in any way. For example, a process cannot delay itself to await the termination of some other process. When compared with the *fork-join* primitives, the term *new* acts as a *fork* and the *end* statement is similar to a *quit*; however, there is no construct corresponding to the primitive *join*. This implies that additional mechanisms must be provided if more controlled interactions such as those depicted in Figure 2-1c and 2-1d are to be enforced.

## 2.3 BASIC PROCESS INTERACTIONS

### 2.3.1 Competition: The Critical Section Problem

When several processes may asynchronously access a common data area, it is necessary to protect the data from simultaneous change by two or more processes. If this protection is not provided, the updated area may be left in an inconsistent state.

There is also a *distributed* version of the problem where the goal again is to ensure that only one process at a time can access a data area or resource. However, the processes

and resource may be on different nodes of a distributed network, and do not directly share memory. We will consider the distributed problem is Section 3.2.3. This section assumes the shared memory version.

Consider two processes $p_1$ and $p_2$, both asynchronously incrementing a common variable $x$ representing, say, the number of units of a resource:

```
cobegin
p1:  ...
     x = x + 1;
     ...
     //
p2:  ...
     x = x + 1;
     ...
coend
```

Each of the two high-level instructions `x = x + 1;` is normally translated into several machine-level instructions. Let us assume that these instructions are:

1. load the value of $x$ into some internal register ($R = x$;).
2. increment that register ($R = R + 1$;).
3. store the new value into the variable $x$ ($x = R$;).

Assume that $p_1$ and $p_2$ run on two separate CPUs, each having its own set of registers but sharing the main memory. Given that the two processes are asynchronous and may proceed at arbitrary pace, either of the two execution sequences shown below could occur over time.

*Sequence 1:*

```
p1:  R1 = x; R1 = R1 + 1; x = R1; ...
p2:  ...                          R2 = x; R2 = R2 + 1; x = R2; ...
```

*Sequence 2:*

```
p1:  R1 = x; R1 = R1 + 1; x = R1; ...
p2:  ...     R2 = x; R2 = R2 + 1; x = R2; ...
```

Let $x$ contain the value $v$ at the time execution begins. At the time of completion, $x$ should contain $v + 2$. This will be the case if the execution follows sequence 1, which executes the instructions sequentially. However, the value of $x$ is $v + 1$ if execution follows sequence 2. The reason is that *R1* and *R2* both receive the *same* initial value $v$, and both later store the same value $v + 1$ into $x$.

The same problem can occur even on a uniprocessor system, where $p_1$ and $p_2$ are time-sharing the same CPU with control switching between the processes by means of interrupts. This is true even if both processes use the same register to load and update the value of $x$. The reason is that registers are saved and restored during every process switch, and each process has its own copy of all register contents. For example,

the following interleaved sequence of execution produces the same incorrect result as Sequence 2 above:

*Sequence 3:*

```
p1:  R = x; ...                          R = R + 1; x = R; ...
p2:  ...    R = x; R = R + 1; x = R; ...
```

The loss of one of the updates of the variable $x$ by Sequence 2 and 3 is clearly unacceptable. To solve this problem, we must prevent the interleaving or concurrent execution of the three machine instructions corresponding to the high-level instruction `x = x + 1`. In general, any segment of code involved in reading and writing a shared data area is called a **critical section** (CS).

The CS problem now can be stated more precisely as follows. We are given $n$ sequential processes ($n \geq 2$), which can communicate with each other through a shared data area. Each process contains a code section—its *CS*—in which read and write accesses to the common data are made. The processes are cyclic, each executing in an infinite loop. During each iteration, a process $p_i$ needs to execute its CS, $CS_i$. It also executes other code, denoted as $program_i$, which does not involve any of the shared data. The following code skeleton shows the overall organization of the competing processes:

```
cobegin
p1: while (1) {CS1; program1;}
        //
p2: while (1) {CS2; program2;}
        //
        ⋮
        //
pn: while (1) {CSn; programn;}
coend
```

We make the following additional assumptions about the system:

1. Writing into a variable and reading from a variable are each indivisible operations. Thus simultaneous access to the same memory location by more than one process results in a sequential access in an unknown order. For example, when two or more processes attempt to simultaneously write to the same location, one of the values (chosen at random) is stored; the others are lost.
2. CSs do not have priorities associated with them.
3. The relative speeds of the processes are unknown. Thus processors of different speeds may be used and the interleaving of instructions on a single processor may be arbitrary.
4. A program may halt only outside of its CS.

The CS problem is then to implement the processes so that, at any point in time, at most one of them is in its CS; once a process $p$ enters its CS, no other process may

do the same until $p$ has left its CS. This property is called **mutual exclusion**, since CSs must be executed in a mutually exclusive fashion with respect to each other.

At the same time, **mutual blocking** of processes must be avoided. We can distinguish the following types of blocking:

1. A process operating well outside its CS, i.e., a process that is not attempting to enter its CS, must not be preventing another process from entering its CS.
2. It must not be possible for one of the processes to repeatedly enter its CS while the other process never gets its chance; i.e., the latter process cannot **starve**.
3. Two processes about to enter their CS must not enter infinite waiting loops or blocking statements that would cause them to wait forever. Such a condition is referred to as a **deadlock**
4. Two processes about to enter their CS must not repeatedly yield to each other and indefinitely postpone the decision on which one actually enters. Such a condition is referred to as a **livelock**.

### Software Solution

In this section, we will present a solution to the CS problem that does not require any special machine instructions or other hardware. Before presenting this solution, we will illustrate some of the difficulties and pitfalls that exist when solving this seemingly simple problem. In fact, solving CS and other synchronization problems are among the most difficult and error-prone tasks in concurrent programming.

We restrict the system initially to only two processes as illustrated in Figure 2-5. The primary goal is to prevent $p_1$ and $p_2$ from executing in their respective CS together (mutual exclusion) while avoiding all forms of mutual blocking.

The problem is easily solved if we insist that $p_1$ and $p_2$ enter their CSs alternately; one common variable, *turn*, can keep track of whose turn it is. This idea is implemented in the first algorithm below.

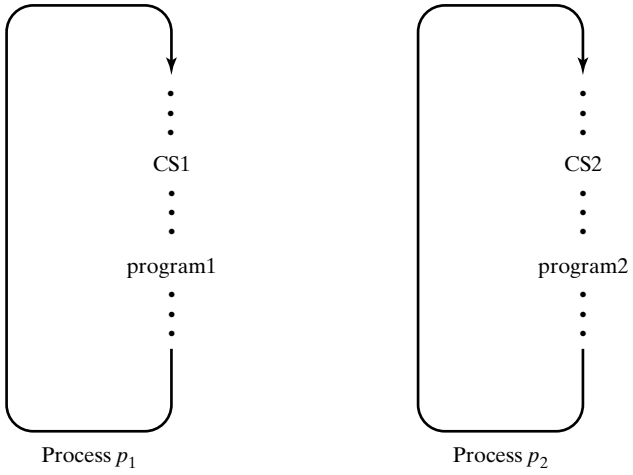


FIGURE 2-5. Two processes with critical sections.

```
/* CS Algorithm: Try #1 */
int turn = 1;
cobegin
p1:  while (1) {
         while (turn==2) ; /*wait loop*/
         CS1; turn = 2; program1;
     }
     //
p2:  while (1) {
         while (turn==1) ; /*wait loop*/
         CS2; turn = 1; program2;
     }
coend
```

Initially, *turn* is set to 1, which allows $p_1$ to enter its CS. After exiting, the process sets *turn* to 2, which now allows $p_2$ to enter its CS, and so on.

Unfortunately, if *program1* were much longer than *program2* or if $p_1$ halted in *program*1, this solution would hardly be satisfactory. One process well outside its CS can prevent the other from entering its CS, violating the requirement 1 stated above.

To avoid this type of blocking, two common variables, *c1* and *c2*, may be used as flags to indicate whether a process is inside or outside of its CS. A process $p_i$ wishing to enter its CS indicates its intent by setting the flag $c_i$ to 1. It then waits for the other flag to become 0 and enters the CS. Upon termination of the CS it resets its flag to 0, thus permitting the other process to continue. Our second try below follows this logic.

```
/* CS Algorithm: Try #2 */
int c1 = 0, c2 = 0;
cobegin
p1:  while (1) {
         c1 = 1;
         while (c2) ; /*wait loop*/
         CS1; c1 = 0; program1;
     }
     //
p2: ... /*analogous to p1*/
coend
```

Mutual exclusion is guaranteed with this solution but mutual blocking of the third type is now possible. Both *c1* and *c2* may be set to 1 at the same time and the processes would loop forever in their *while* statements, effectively deadlocked. The obvious way to rectify this is to reset *c1* and *c2* to 0 after testing whether they are 1, leading to our third try at an algorithm for solving the CS problem.

```
/* CS Algorithm: Try #3 */
int c1 = 0, c2 = 0;
cobegin
p1:  while (1) {
         c1 = 1;
```

```
            if (c2) c1 = 0;
                  else {
                      CS1; c1 = 0; program1;
                  }
        }
        //
p2: ...
coend
```

Unfortunately, this solution may lead to both the second and fourth type of blocking. When the execution timing is such that one process, say $p_2$, always tests the *c1* flag just after it was set to 1 by $p_1$ and, as a result, sets *c2* to 0 before it is tested by $p_1$, $p_1$ will always succeed in entering its critical section, while $p_2$ is forced to wait.

The fourth type of blocking occurs when both processes begin execution at the same time and proceed at exactly the same speed. They will keep setting and testing their mutual flags indefinitely, without ever entering their CSs. Note, however, that such indefinite postponement is much less likely to occur here than with the previous solution, since both processes must maintain the same speed indefinitely; if one becomes slower than the other, the mutual blocking is resolved. With the previous solution (Try #2), both processes remained in their respective *while*-loops forever, regardless of speed.

The above three attempts at solving the CS problem illustrate some of the subtleties underlying process synchronization. In 1981, G. L. Peterson proposed a simple and elegant algorithm, which we now present as the final complete software solution to the CS problem.

```
/* CS Algorithm: Peterson Solution */
int c1 = 0, c2 = 0, will_wait;
cobegin
p1:  while (1) {
         c1 = 1;
         will_wait = 1;
         while (c2 && (will_wait==1)) ; /*wait loop*/
         CS1; c1 = 0; program1;
     }
     //
p2:  while (1) {
         c2 = 1;
         will_wait = 2;
         while (c1 && (will_wait==2)) ; /*wait loop*/
         CS2; c2 = 0; program2;
     }
coend
```

A process indicates its interest to enter the CS by setting its flag $c_i$ to 1. To break possible race conditions, the algorithm uses the variable *will_wait*; by setting this variable to its own identifier, a process indicates its willingness to wait if both processes are trying to enter their CS at the same time. Since *will_wait* can contain the identifier of only one process at any time, the one who sets it last will be forced to wait.

The solution guarantees mutual exclusion and prevent all forms of blocking. Consider first the problem of mutual blocking. If it were possible, at least one of the processes, e.g., $p_1$, must somehow circle through its *while* loop forever. This is not possible for the following reason. When $p_1$ is in its loop, the second process $p_2$ may be doing one of three general things: (1) not trying to enter its CS, (2) waiting in its own *while* loop, or (3) repeatedly executing its own complete loop. In the first case, $p_1$ detects that *c2* is 0 and proceeds into its CS. The second case is impossible because *will_wait* is either 1 or 2, which permits one of the processes to proceed. Similarly, the third case is impossible since $p_2$ will set *will_wait* to 2, and, consequently, not pass through its *while* statement test. It will not be able to proceed until $p_1$ has executed its CS.

To show that the solution guarantees mutual exclusion, assume that $p_1$ has just passed its test and is about to enter its CS. At this time, *c1* must be 1. Let us examine if there is any way for $p_2$ to enter *CS2*, thus violating the mutual exclusion requirement. There are two cases to consider:

1. $p_1$ has passed its test because *c2* was 0. This implies that $p_2$ is currently not trying to enter its CS, i.e., it is outside of the segment of code delimited by the instructions *c2 = 1;* and *c2 = 0;*. If it now tries to enter, it must first set *will_wait* to 2. Since *c1* is 1, it will fail its test and will have to wait until $p_1$ has left its CS.
2. $p_1$ has passed its test because *will_wait* was equal to 2. This implies that, regardless of where $p_2$ is at this time, it will find *c1* equal to 1 and *will_wait* equal to 2 when it reaches its test, and will not be permitted to proceed.

The CS problem is concerned with the most basic requirements when independent processes may access the same resource or data. Mutual exclusion is also the basis for many other more complex types of process interactions and resource sharing. The solutions to these and other resource allocation scenarios are discussed in the remainder of this chapter and in Chapter 3.

### 2.3.2 Cooperation

The CS problem described above is a situation in which processes *compete* for a resource that may not be accessed by more than one process at any given time. Note that each process could exist without the other—their interaction is needed only to resolve simultaneous access to the resource. A different situation arises when two or more processes **cooperate** in solving a common goal. In this case, each of them is aware of, and usually depends on, the existence of the other. Similar to concurrent processes with CSs, cooperating processes have a need to exchange information with one another. In the simplest cases, only synchronization signals are necessary. In more general scenarios, processes must pass data to each other, which may be done through shared memory or by sending messages.

One general example of cooperation occurs in the implementation of precedence relations among processes (Section 2.2.1). The activation of some processes is synchronized with the termination of others. A similar application, appearing frequently in scientific computations, is *barrier* synchronization, which is required when a group of processes must all arrive at a particular point before continuing.

Processes engaged in what is typically characterized as a **producer/consumer** relationship also fall into this category. Figure 2-6 describes this situation for one *Consumer* and one *Producer* process that communicate via a shared *Buffer* storage that can hold a

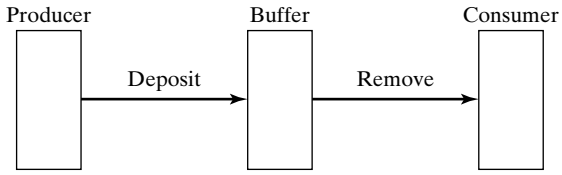


FIGURE 2-6. A producer and a consumer communicating through a buffer.

finite number of data elements. The *Producer*'s task is to generate (produce) one or more new elements of data, which it deposits in the buffer. These are removed (consumed) from the buffer by the *Consumer* process.

Different buffering scenarios occur throughout the I/O system, the file system, and various network components. For example, a main application process may be producing output data and placing them into available buffer slots, while, asynchronously, an output process is removing them from the buffer and printing them. Alternately, the *Producer* could be an input process, and the *Consumer* is the main application process using the input data.

Since both processes run concurrently at varying speeds, we must guarantee that the *Producer* does not overwrite any data in the buffer before the *Consumer* can remove it. Similarly, the *Consumer* must be able to wait for the *Producer* when the latter falls behind and does not fill the buffer on time. Hence the two processes must be able to exchange information about the current state of the shared buffer—specifically, the number of data elements present.

Process cooperation is also necessary in distributed systems, where the processes do not share any memory. For example, OS services such as file or directory servers, are frequently implemented as separate processes running on dedicated machines. Other processes make use of such a service by sending it the necessary input data in the form of messages. Upon performing the desired computations with the received data, the service returns the results to the calling process in the form of reply messages. Such cooperating processes are referred to as **communicating** processes, and this form of interaction is called a **client-server** architecture. It requires a general interprocess communication facility to allow processes to establish logical channels with one another, through which arbitrary messages can be exchanged. We will discuss the principles of message passing in Chapter 3. In the remainder of this chapter, we introduce the principles of *semaphores*. In reality, semaphores are devices for signaling or conveying information by changing the position of a light, flag, or other signal. Semaphores in OSs are an analogy to such signaling devices. They offer an elegant, universal, and popular scheme that can be used as a foundation for programming both process competition and process cooperation in uniprocessor or shared-memory multiprocessor systems.

## 2.4 SEMAPHORES

There are several unappealing features of the software solution to the CS problem that was presented earlier:

1. The solution is difficult to understand and verify. Enforcing mutual exclusion for even two processes results in complex and awkward additions to programs.

Extending the solution to more than two processes requires even more code to be generated.

2. The solution applies to only competition among processes. To address the problem of cooperation, entirely different solutions must be devised.
3. While one process is in its CS, the other process continues running but is making no real progress—it is repeatedly accessing and testing some shared variables. This type of waiting steals memory cycles from the active process; it is also consuming valuable CPU time without really accomplishing anything. The result is a general slowing down of the system by processes that are not doing any useful work.

Dijkstra (1968) introduced two new primitive operations, called $P$ and $V$, that considerably simplified the coordination of concurrent processes. The operations are universally applicable to competition and cooperation among any number of processes. Furthermore, $P$ and $V$ may be implemented in ways that avoid the performance degradation resulting from waiting.

### 2.4.1 Semaphore Operations and Data

The $P$ and $V$ primitives operate on *nonnegative integer* variables called **semaphores**.[2] Let $s$ be a semaphore variable, initialized to some nonnegative integer value. The operations are defined as follows:

- *V(s)*: Increment $s$ by 1 in a single *indivisible* action; i.e, the fetch, increment, and store of $s$ cannot be interrupted, and $s$ cannot be accessed by another process during the operation.
- *P(s)*: If $s > 0$ then decrement $s$ and proceed; otherwise (i.e., when $s == 0$), the process invoking the $P$ operation must wait until $s > 0$. The successful testing and decrementing of $s$ also must be an indivisible operation. In Chapter 4, we discuss in detail how the waiting is accomplished when $s$ is zero. In brief, the invoking process can be suspended or it can execute a "busy-wait" loop continually testing $s$.

In addition to the indivisibility of the $P$ and $V$ operations, we make the following assumptions:

- If several processes simultaneously invoke a $P$ or $V$ operation on the same semaphore, the operations will occur sequentially in an arbitrary order;
- If more than one process is waiting inside a $P$ operation on the same semaphore and the semaphore becomes positive (because of the execution of a $V$), one of the waiting processes is selected arbitrarily to complete the $P$ operation.

The $P$ and $V$ operations are used to synchronize and coordinate processes. The $P$ primitive includes a potential wait of the calling process, whereas the $V$ primitive may possibly activate some waiting process. The indivisibility of $P$ and $V$ assures the integrity of the values of the semaphores and the atomicity of the test and decrement actions within each $P$ operation.

[2] $P$ and $V$ are the first letters of Dutch words that mean "pass" and "release," respectively.

### 2.4.2 Mutual Exclusion with Semaphores

Semaphore operations allow a simple and straightforward solution to the CS problem. Let *mutex* (which stands for *mut*ual *ex*clusion) be a semaphore variable used to protect the CS. A program solution for $n$ processes operating in parallel is as follows:

```
semaphore mutex = 1;
cobegin
p1:  while (1) { P(mutex); CS1; V(mutex); program1; }
     //
pi:  while (1) { P(mutex); CS2; V(mutex); program2; }
     //
     ⋮
     //
pn:  while (1) { P(mutex); CSn; V(mutex); programn; }
coend;
```

The value of *mutex* is 0 when any process is in its CS; otherwise *mutex* is 1. The semaphore has the function of a simple lock. Mutual exclusion is guaranteed, since only one process can decrement *mutex* to zero with the $P$ operation. All other processes attempting to enter their CSs while *mutex* is zero will be forced to wait by *P(mutex)*, and execution of *P(mutex)* by multiple processes simultaneously selects one of them at random. Mutual blocking (through a deadlock or indefinite postponement) is not possible, because simultaneous attempts to enter CSs when *mutex* is 1 must, by our definition, translate into *sequential P* operations. The starvation of a process wishing to enter its CS is, however, possible. This depends on how the semaphores are implemented. Starvation may in fact be desirable in some cases, for example, when processes are serviced on a priority basis; but servicing $P$ operations in a first-come/first-served order eliminates any possibility of starvation.

When a semaphore can take only the values 0 or 1, it is called a **binary** semaphore; otherwise—when the semaphore can take any nonnegative integer value—it is referred to as a **counting** semaphore. An important and widespread application of binary semaphores is the implementation of mutual exclusion locks, such as in the above CS problem: $P$ locks a data structure for exclusive use, and a later $V$ unlocks it. This, and nested versions of it, are such common scenarios that systems provide special operations for just the locking and unlocking of CSs, in addition to counting semaphores.

**CASE STUDY: SEMAPHORE PRIMITIVES**

1. *Solaris*. The Solaris OS offers both binary and counting semaphores through thread libraries. The binary semaphore is called a *mutex* and may be defined for interprocess or intraprocess synchronization. In the first case, the mutex variable is local to a process and may be used to synchronize threads within that process. The second type allows threads belonging to different processes to synchronize with each other. Mutex synchronization supports the following function:
   - *mutex_init():* creates and initializes a mutex;
   - *mutex_destroy():* destroys an existing mutex;

**CASE STUDY: SEMAPHORE PRIMITIVES (*continued*)**

- *mutex_lock():* locks a mutex; when another thread tries to lock the same mutex, it is blocked until the original locking thread unlocks it;
- *mutex_trylock():* this is a nonblocking version of the *mutex_lock*; it tries to lock the mutex but if the mutex is already locked, it returns an error value, instead of blocking;
- *mutex_unlock():* unlocks a mutex.

Counting semaphores may take on any nonnegative integer value, and function exactly as described in this section. They support five function, analogous to those for mutexes:

- *sema_init():* creates and initializes a semaphore;
- *sema_destroy():* destroys an existing semaphore;
- *sema_wait():* decrements the semaphore if its value is greater than 0; otherwise the operation blocks;
- *sema_trylock():* this is a nonblocking version of the *sema_wait*; it tries to decrement the sempahore but if this is zero, it returns an error value, instead of blocking;
- *sema_post():* increments the semaphore.

**2.** *Linux*. Linux offers two types of synchronization primitives. The first, referred to as *kernel semaphores*, are general counting semaphores, and may be used on both uniprocessor and multiprocessor systems. The two main functions, corresponding to $P$ and $V$, are *down()* and *up()*, respectively.

The other type of primitive are binary semaphores, called *spin locks* (because of their implementation as waiting loops; see Chapter 4). They are controlled through several macros, notably, *spin_lock*, which locks a given binary semaphore, *spin_unlock*, which unlocks it, and *spin_trylock*, which is the nonblocking variant of *spin_lock*. In addition to these exclusive binary semaphores, Linux defines a second type, called *read/write spin locks*. Such locks are controlled using the following macros:

- *read_lock:* locks the semaphore in read-only mode; other threads may apply the same *read_lock* to the semaphore and proceed concurrently; *read_lock* blocks only when the semaphore is already locked using the *write_lock*;
- *write_lock:* locks the semaphore in exclusive (read/write) mode; if the semaphore is already locked (for either reading or writing), the operation blocks;
- *read_unlock:* this removes one of the previously placed read locks, thus decrementing the number of concurrent readers;
- *write_unlock:* this removes a previously placed write lock.

### 2.4.3 Semaphores in Producer/Consumer Situations

Processes in computer systems request and release various resources. Access to nonsharable resources is generally protected by CSs as described above. In addition, processes

may be *producing* resources, such as messages, signals, or data items, which are *consumed* by other processes. Semaphores can be used to maintain resource counts, synchronize processes, and lock out CSs.

For example, a process can block itself by a *P* operation on a semaphore *s* to wait for a certain condition, such as a new data item becoming available. It can be awakened by another process executing *V* on *s* when this condition becomes satisfied. The following code skeleton illustrates this type of interaction among processes:

```
semaphore s = 0;
cobegin
p1:  { ...; P(s); ... }    /* Wait for signal from p2.*/
     //
p2:  { ...; V(s); ... }    /* Send wakeup signal to p1.*/
coend
```

In this scenario, *p1* may also be viewed as consuming units of the resource represented by the semaphore *s* through the instruction *P(s)*, while *p2* produces units of *s* through *V(s)*.

### EXAMPLE: The Bounded-Buffer Problem (Dijkstra 68)

Probably the best known and practical producer/consumer situation is the bounded buffer scenario introduced in Section 2.3.2. A *Producer* process generates data elements and adds them to a shared buffer. Concurrently, a *Consumer* process removes data elements from the buffer and processes them. The buffer is a shared data area. At any point in time, it will contain zero or more full data elements (deposited but not yet removed) and a number of empty slots.

Let the buffer consist of *n* equal-sized slots, each capable of holding one data element. We use two resource counters:

- *e* denotes the number of empty buffer slots; these are currently available to the *Producer*;
- *f* denotes the number of full buffer slots; these are currently available to the *Consumer*.

Incrementing and decrementing *e* and *f* must be indivisible, or their values could be in error as a result of possible race conditions (Section 2.3.1). Thus, instead of implementing *e* and *f* as ordinary variables and treating their changes as CSs, they are implemented as semaphores. The *P* and *V* operations then guarantee that all changes are atomic. Furthermore, the *P(e)* ensures that the *Producer* does not proceed unless an empty buffer slot is available; similarly, *P(f)* guarantees that the *Consumer* waits until at least one full buffer slot exists.

Assume that adding elements to and taking elements from the buffer constitute CSs. We use a binary semaphore, *b*, for mutual exclusion. The processes then may be described as follows:

```
semaphore e = n, f = 0, b = 1;
cobegin
Producer:  while (1) {
               Produce_next_data_element;
```

```
                    /* Deposit data element. */
                    P(e); P(b); Add_to_buffer; V(b); V(f);
                }
                //
Consumer:       while (1) {
                    /* Remove data element. */
                    P(f); P(b); Take_from_buffer; V(b); V(e);
                    Process_data_element;
                }
coend
```

If the buffer was implemented as an array, mutual exclusion, enforced by the semaphore *b*, might not be necessary. The reason is that the *Producer* and the *Consumer* always refer to different buffer elements. However, if linked lists of buffers are employed or if the program is generalized to $m$ producers and $n$ consumers ($m, n \geq 1$), mutual exclusion is necessary. The reason is both processes could be manipulating pointers linking together adjacent list elements or pointers designating the next empty or full slot.

## 2.5 EVENT SYNCHRONIZATION

An **event** designates some change in the state of the system that is of interest to a process. An event is usually considered to take zero time. In computer systems, events are generated principally through hardware interrupts and traps, either directly or indirectly. For example, the end of an I/O operation, the expiration of a timer or clock, a machine error, or a programming error such as overflow or an invalid address, are all considered events that must trigger an action. Events, implemented by software interrupts, also are used for process interactions, e.g., for one process to interrupt and terminate another, or for general interprocess synchronization.

Many systems provide facilities to define and handle specific events. We can differentiate between two type of events: **synchronous** and **asynchronous**. In the first case, a process blocks itself to wait for the occurrence of a specific event or set of events, generated by another process. This is similar to the consumer/producer situation discussed earlier, and, consequently, the primitives to handle synchronous events are similar to binary semaphores.

Generally, support for synchronous events includes ways to define event types, generate or post events, and wait for events. An operation, such as *E.wait*, can be used as an explicit wait for an event, *E*, to occur. An instance of this event is generated by an explicit command, e.g., *E.post*. If there are any processes or threads waiting on *E*, one or more is awakened by the *E.post* operation.

In their simplest form, events can be implemented using binary semaphores. However, there is no consistent definition of events, and a variety of schemes with differing semantics are in use. One common variant that clearly distinguishes events from binary semaphores has no "memory" of posted events. If there are no waiting processes when an event is posted (signaled), then the instance of that event simply disappears. In contrast, the effect of a $P$ or $V$ operation is reflected in the new value of the semaphore. Some implementations of events also allow the *broadcasting* of an event to a group of waiting processes or to wake up only a single waiter.

Asynchronous events also must be defined and explicitly posted using a *E.post* operation. However, a process does not explicitly block itself to await the occurrence of an asynchronous event. Instead, it only indicates its interest in specific types of events and specifies the actions to be taken when a particular event occurs. This is commonly done by providing **event handlers** for different types of events. Each handler is a section of code that is to be executed automatically and asynchronously with the current process whenever the event is posted. Thus, asynchronous events are similar to interrupts.

### CASE STUDY: EVENT SYNCHRONIZATION

1. *UNIX signals*. UNIX allows processes to signal various conditions to each other using asynchronous events. Such signals are sent using the function *kill(pid, sig)*, where *pid* is the receiving process and *sig* is the type of signal sent. The function name, *kill*, is a misnomer—it does not necessary kill the receiving process, even though that is the default.

   There are certain predefined types of signals. For example, *SIGHUP* signals that the phone line the process was using has been hung up, *SIGILL* signals an illegal instruction, *SIGFPE* signals a floating point error, *SIGKILL* signals that the process is to be terminated, and *SIGUSR1* and *SIGUSR2* are intended for application-specific signals defined by the user.

   The receiving process may specify, using the function *sigaction()*, what should happen when a signal arrives. If nothing is specified, the process is killed. If the process specifies that a signal is to be ignored, the signal is simply lost. Alternately, the process may catch a signal by specifying a signal handler, which is invoked automatically whenever a signal of the specified type arrives. Certain types of signals, such as *SIGKILL* cannot be ignored or caught by any process. This is necessary to guarantee that any process can be killed, for example, to prevent runaway processes.

   Signals also may be used in a synchronous manner. A process may issue the call *pause()*, which will block it until the next signal arrives.

2. *Windows 2000 Synchronization*. Windows 2000 provides a broad spectrum of synchronization options, which it unifies in a common framework of *dispatcher objects*. Each dispatcher object may be in one of two possible states: *signaled* or *nonsignaled*. A process may block itself to wait on an nonsignaled object using the function *WaitForSingleObject*; it is resumed when some other project changes the state of that object to signaled. A process may also block itself on multiple objects using the function *WaitForMultipleObjects*. It is resumed when all the objects have been signaled.

   The possible object types include processes, threads, files, events, semaphores, timers, mutexes, and queues. The change from nonsignaled to signaled depends on the type of object. The following list summarizes the possible causes and actions:

   - A *process* object is signaled when its last thread terminates; all waiting threads are woken up;

**CASE STUDY: EVENT SYNCHRONIZATION (*continued*)**

- A *thread* object is signaled when it terminates; all waiting threads are woken up;
- A *semaphore* object is signaled when the semaphore value is decremented; a single thread is woken up;
- A *mutex* object is signaled when a thread releases the mutex lock; a single thread is woken up;
- An *event* object is signaled when a thread explicitly posts this event; there are two version of this object: one wakes up all waiting threads while the other wakes up only one thread;
- A *timer* object is signaled when a timer expires; similar to an event object, a timer object may wake up all waiting threads or just one;
- A *file* object is signaled when on I/O operation on that file terminates; all waiting threads are woken up;
- A *queue* object is signaled when an item is placed on the queue; a single thread is woken up.

## CONCEPTS, TERMS, AND ABBREVIATIONS

The following concepts have been introduced in this chapter. Test yourself by defining and discussing each keyword or phrase.

| | |
|---|---|
| Bounded buffer | P and V operations |
| Cobegin-coend statements | Process |
| Critical section | Process flow graph |
| Event | Producer-consumer |
| Forall statement | Semaphore, general and binary |
| Fork-join statements | Starvation |
| Mutual exclusion | |

## EXERCISES

**1.** Show the process flow graph for the following expressions:

$$S(p_1, P(p_2, S(P(S(p_3, p_4), p_5), P(p_6, p_7))))$$

$$S(P(S(p_1, P(p_2, p_3)), p_4), p_5)$$

**2.** Rewrite the program below using *cobegin/coend* statements. Make sure that it exploits maximum parallelism but produces the same result as the sequential execution. Hint: Draw first the process flow graph where each line of the code corresponds to an edge. Start with the last line.

```
W = X1 * X2;
V = X3 * X4;
Y = V * X5;
Z = V * X6;
```

```
Y = W * Y;
Z = W * Z;
A = Y + Z;
```

**3.** The following expression describes the serial/parallel precedence relationship among six processes $p_1$ through $p_6$:

$$P(S(P(p_3, S(p_1, P(p_6, p_5))), p_2), p_4)$$

Transform this expression into a program using:
**(a)** *cobegin/coend*
**(b)** *fork, join, quit* primitives

**4.** Rewrite the matrix multiplication example of Section 2.2.2 to reduce the degree of parallelism as follows:
**(a)** The elements of each row $i$ are computed in parallel, but the rows are computed sequentially one at a time
**(b)** The elements of each column $j$ are computed in parallel, but the columns are computed sequentially one at a time
Under what circumstances would either of the above versions of the program be preferable to the one given in the text, where all elements of the matrix are computed in parallel?

**5.** The following function sorts the elements of an array $A[n]$ in place using a recursive two-way merge-sort. Assume that the function *merge()* merges the two subranges $A[lo, mid]$ and $A[mid + 1, hi = 1, hi]$ of the array $A$.

```
void mergesort(int A[], int lo, int hi) {
    int mid;
    if (lo < hi) {
        mid = (lo + hi) / 2;
        mergesort(A, lo, mid);
        mergesort(A, mid+1, hi);
        merge(A, lo, mid, mid+1, hi);
    }
}
```

Rewrite the function using *fork, join, quit* such that the two invocations of *mergesort()* are executed in parallel.

**6.** Why must most of the *join* operation be invisible?

**7.** Consider the CS problem of Section 2.3.1. Show all possible interleavings of the three machine instructions executed on a uniprocessors by the processes $p_1$ and $p_2$ that result in an incorrect value of $x$.

**8.** Consider the last software solution to the mutual exclusion problem (Section 2.3.1).
**(a)** Assume process $p1$ is inside $CS1$. What are the values of $c1$, $c2$, and $turn$ that prevent $p2$ from entering its critical section?
**(b)** Assume that both processes have just entered the while-loop immediately preceding their respective critical sections. What are the values of $c1$, $c2$, and $turn$ at that point? What guarantees that exactly one process will be able to proceed?
**(c)** Assume process $p1$ terminates. What are the values of $c1$, $c2$, and $turn$ that allow $p2$ to continue entering its critical section repeatedly?

**9.** Consider the last software solution to the mutual exclusion problem (Section 2.3.1). Assume that process $p_1$ sets the *will_wait* flag to 2 (instead of 1), indicating that

$p_2$ should wait if necessary. Similarly, $p_2$ sets the *will_wait* flag to 1 (instead of 2), indicating that $p_1$ should wait if necessary. No other changes are made to the program. Will this still prevent mutual exclusion?

**10.** Generalize the last software solution to the mutual exclusion problem (Section 2.3.1) to work with three processes. (Hint: Use two stages. During the first stage, one of the three processes is held back. During the second stage, the remaining two processes compete with each other and one is held back.)

**11.** Use semaphores to describe the synchronization of the eight processes in the general precedence graph of Figure 2-1d.

**12.** What is the effect of interchanging:
**(a)** $P(b)$ and $P(e)$ or
**(b)** $V(b)$ and $V(f)$
in the producer process in the Bounded Buffer example of Section 2.4.3?

**13.** A simple batch OS may be described as a set of three processes interacting as follows:

```
process BATCH_OS;
...
    process reader;
    begin
        while(1)
            read record from input device;
            deposit record in input_buffer
    end;

    process main;
    begin
        while(1)
            fetch record from input_buffer;
            process record and generate output line;
            deposit line in output_buffer
    end;

    process printer;
    begin
        while(1)
            fetch line from output_buffer;
            print line on output device
    end
end
```

Modify and augment the batch OS above using semaphore operations to properly buffer the input data that flows from the reader to the main function, and the output data that flows from main to the printer.

**14.** The following code spawns $n - 1$ concurrent processes. Each process $i$ repeatedly updates the elements $A[i]$ and $B[i]$ of two shared arrays.

```
int A[n], B[n];
A[0] = init();
forall (i:1..(n-1)) {
    while (not_done) {
        A[i] = f(A[i], A[i-1]);
```

```
            B[i] = g(A[i])
        }
    }
```

Since computing $A[i]$ uses $A[i-1]$, each process $i$ must wait for its predecessor process $i-1$ to complete the computation of $A[i-1]$. Insert the necessary semaphore operations into the code to guarantee that processes wait for each other to guarantee the correctness of the program, but allow for as much overlap between the computations as possible. Also show the initial values of all semaphores used. Hint: Declare an array of semaphores $s[n-1]$.

**15.** Consider the following two functions, where $A$ and $B$ are arbitrary computations:

```
f1() {                          f2() {
    P(s1);                          P(s2);
    c1 = c1 + 1;                    c2 = c2 + 1;
    if (c1 == 1) P(d);              if (c2 == 1) P(d);
    V(s1);                          V(s2);
    A;                              B;
    P(s1);                          P(s2);
    c1 = c1 - 1;                    c2 = c2 - 1;
    if (c1 = 0) V(d);               if (c2 = 0) V(d);
    V(s1);                          V(s2);
}                               }
```

Initially: `s1=s2=d=1; c1=c2=0;`

Assume that an unbounded number of processes are invoking either of the functions $f1()$ or $f2()$.

**(a)** How many invocations of the computation $A$ can proceed concurrently? What are the values of $s1$, $c1$, and $d$ at that time?

**(b)** While $A$ is running, how many invocations of $B$ can proceed concurrently? What are the values of $s2$, $c2$, and $d$ at that time?

**(c)** Can $A$ or $B$ starve? Explain why or why not.

**16.** Simulate the traffic at an intersection of two one-way streets using semaphore operations. In particular, the following rules should be satisfied:

- Only one car can be crossing at any given time;
- When a car reaches the intersection and there are no cars approaching from the other street, it should be allowed to cross;
- When cars are arriving from both directions, they should take turns to prevent indefinite postponement of either direction.