CHAPTER 3

Higher-Level Synchronization and Communication

3.1 SHARED MEMORY METHODS
3.2 DISTRIBUTED SYNCHRONIZATION AND COMMUNICATION
3.3 OTHER CLASSIC SYNCHRONIZATION PROBLEMS

The main objections to semaphores and events—the synchronization mechanisms introduced in the last chapter—are that they are too low level and do not support the elegant structuring of concurrent programs. For example, they do not permit a segment of code to be designated explicitly as a critical section (CS). Rather, the effect of a CS must be enforced by correctly using the semaphore operations, i.e., by enclosing the desired segment between a pair of $P$ and $V$ operations and presetting the corresponding semaphore to 1, or by employing a $mutex.lock$ and $mutex.unlock$ pair correctly. Violating any of these rules will destroy the desired effect.

Since $P$ and $V$ operations may be used anywhere in a program, the task of understanding and verifying the desired behavior of programs becomes very difficult. One common error that is usually extremely difficult to detect is the omission of a required $V$ operation from a program or its mere bypassing when execution follows some unexpected path; this could result in a situation where processes are blocked forever—a situation commonly referred to as deadlock. Equally dangerous and common is an unintentional execution of a $V$ operation, permitting, e.g., more than one process to enter a CS.

The first part of the chapter presents several constructs that are used as alternatives to the low-level semaphore and event operations. These are based on ideas from abstract data types and objects. The aim is to concentrate and encapsulate all accesses to a shared resource, including any required synchronization. These mechanisms assume that processes share parts of main memory.

However, it is not always desirable or possible that processes share some portion of memory. For example, it is sound software engineering practice to encapsulate program entities, including processes, as a way to improve understanding, analysis, and reusability. Shared memory is in conflict with this principle. For security reasons, processes often must run in isolation, each in its own logical space, with all interactions under strict control of the participating process. Another reason for alternatives to shared memory constructs is the increasing importance and use of distributed systems. In such environments, each processor may have only its own local memory; consequently, there can be no direct data sharing among processes running on different processors. For these reasons, distributed schemes for interprocess communication (IPC) and synchronization are commonly available.
The most common distributed approaches, message passing and procedure-based interactions, are the subjects of the second part of this chapter. The distributed schemes we discuss are usable and frequently used on shared memory architectures; conversely, shared memory methods are often simulated on distributed architectures. The final sections of this chapter describe and solve several famous and classical synchronization problems.

### 3.1 SHARED MEMORY METHODS

The two most interesting, useful, and popular constructs for higher-level, shared memory synchronization are monitors and protected types.

#### 3.1.1 Monitors

**Hoare Monitors**

The monitor concept (Hoare 1974; Brinch Hansen 1973b) follows the principles of abstract data types. For any distinct data type, there is a well-defined set of operations through which, and only through which, any instance of that data type is manipulated. Following this idea, a monitor is defined as a collection of data representing the state of the resource controlled by the monitor, and a set of procedures to manipulate that resource data.

The implementation of the monitor construct must guarantee the following:

1. Access to the resource is possible only via one of the monitor procedures.
2. Procedures are mutually exclusive; i.e., at any given time, only one process may be executing a procedure within a given monitor. The jargon is that only one process or thread may be inside a given monitor at a time. During that time, other processes calling a procedure of this same monitor are delayed until the process leaves the monitor.

A monitor, as defined above, is sufficient to implement a CS by preventing simultaneous access to a resource. However, it does not provide any means for processes to communicate or synchronize with one another. For this purpose, monitors introduce a special type of variable called a condition variable. Two operations, wait and signal, operate on condition variables and can only be used inside monitor procedures.

An operation wait on a condition variable \( c \) will be denoted as:

\[
\text{c.wait.}
\]

It causes the executing process to be suspended (blocked) and placed on a queue associated with the condition variable \( c \). The blocked process releases the mutual exclusion lock of the monitor so that some other process can now execute a procedure of that monitor, i.e., “enter” the monitor. Performing the signal operation on \( c \), written as:

\[
\text{c.signal,}
\]

wakes up one (if any) of the processes waiting on \( c \), placing it on a queue of processes wanting to reenter the monitor and resume execution after the wait. Here, we assume that the process that has waited the longest will be awakened. If no processes are waiting, a signal acts as a null operation, much like a memoryless post of an event.
Condition variables are not variables in the classical sense, i.e., there is no value associated with one. Rather, each condition variable may be viewed as a name chosen by the programmer to refer to a specific event, state of a computation, or assertion. To illustrate this idea, assume that a process may proceed only when some variable, e.g., X, has a value greater than zero. We can define a condition variable, e.g., X_is_positive, and use it to exchange information about the truth value of the expression “X > 0” among different processes. Whenever a process, e.g., p, finds this value to be false, it performs the operation X_is_positive.wait, which suspends that process on a queue associated with X_is_positive. Figure 3-1a illustrates the effect of wait. When another process, e.g., q, changes the content of X to a positive value, it may inform the suspended process of this event by performing the operation X_is_positive.signal. Figure 3-1b shows this situation. Note that the condition variable X_is_positive does not actually contain the value true or false of the expression X > 0; the logic of the monitor code must explicitly test for the desired condition.

There is one important issue surrounding the meaning or semantics of the signal operation. Since it is used to indicate that a condition associated with some condition variable c on which some process p may be suspended, is now satisfied, there may be

![Diagram](image_url)

**FIGURE 3-1.** A monitor (a) effect of wait; and (b) the effect of signal.
two processes eligible to run inside the monitor after a signal is issued: (1) the process executing the signal (process \( q \) in Fig. 3-1), and (2) process \( p \) that previously issued the wait and is now selected for reactivation. By the monitor definition, only one process may be active inside a monitor at any one time; thus, one of the two processes must wait. In the original proposal, Hoare defined the semantics of the signal operation such that the process executing signal is suspended and \( p \) immediately reenters the monitor where it last left off (i.e., immediately after the wait). The suspended process \( q \) is then assigned the highest priority and reenters the monitor as soon as the previously reactivated process \( p \) leaves the monitor, either through a normal exit or by blocking itself with a wait operation.

The rationale behind the above choice is that \( p \) is waiting because some condition \( B \) or assertion, such as \( X > 0 \), is not satisfied, and the signaling process sets \( B \) to true immediately before issuing the signal. Consequently, \( p \) can be assured that nothing will change \( B \) between the time of the signal and the time it regains control of the monitor. Many other alternatives to implementing condition signaling exist. We will examine one popular choice in the next section.

**EXAMPLE: Monitor Solution to the Bounded-Buffer Problem**

Consider the bounded-buffer problem described in the last chapter. Let each element of the buffer be a single character, and implement the shared data area as an array of characters. The monitor, called Bounded Buffer, exports two operations, deposit and remove, which insert and remove, respectively, one character. The synchronization conditions are represented by the condition variables notempty and notfull; these refer to the state of the buffer: not empty and not full, respectively. Thus, for example, a deposit will not be permitted to proceed until the buffer is not full.

```c
monitor Bounded_Buffer {
    char buffer[n];
    int nextin=0, nextout=0, full_cnt=0;
    condition notempty, notfull;

    deposit(char c) {
        if (full_cnt==n) notfull.wait;
        buffer[nextin] = c;
        nextin = (nextin + 1) % n;
        full_cnt = full_cnt+1;
        notempty.signal;
    }

    remove(char c) {
        if (full_cnt==0) notempty.wait;
        c = buffer[nextout];
        nextout = (nextout + 1) % n;
        full_cnt = full_cnt - 1;
        notfull.signal;
    }
}
```
Chapter 3 Higher-Level Synchronization and Communication

The two monitor procedures may be invoked by any process through the calls `Bounded_Buffer.deposit(data)` and `Bounded_Buffer.remove(data)`, respectively, where `data` is a variable of type `char`. If `full_cnt` is zero when `remove` is called, the process will block on the `notempty.wait`, and remain blocked until another process deposits a character and issues a `notempty.signal`. Similarly, a process calling `deposit` when `full_cnt` equals `n` will wait on `notfull.wait` until another process later does a `remove` thereby issuing `notfull.signal`. (The percent sign denotes a modulo `n` operation.)

---

Priority Waits

When more than one process is waiting on the same condition, a `signal` will cause the longest waiting process to be resumed. (This is the usual implementation.) There are many cases where simple first-come/first-served scheduling is not adequate. To give a closer control over scheduling strategy, Hoare (1974) introduced a *conditional* (or scheduled) `wait`, which includes a priority for the waiting process. This has the form:

```
c.wait(p)
```

where `c` is the condition variable on which the process is to be suspended, and `p` is an integer expression defining a priority. When the condition `c` is signaled and there is more than one process waiting, the one which specified the *lowest* value of `p` is resumed.

---

**EXAMPLE: Alarm Clock**

The following example of an alarm clock illustrates the use of priority waits. `Alarm_Clock` enables a process to delay itself for a specified number of clock “ticks,” by calling the procedure `wakeme`. A second operation, `tick`, updates the clock and signals `wakeme`.

```
monitor Alarm_Clock {
    int now=0;
    condition wakeup;

    wakeme(int n) {
        int alarmsetting;
        alarmsetting = now + n;
        while (now<alarmsetting) wakeup.wait(alarmsetting);
        wakeup.signal; /* In case more than one process is to */
        /* wake up at the same time. */
    }

    tick() {
        now = now + 1;
        wakeup.signal;
    }
}
```
Section 3.1 Shared Memory Methods

The \textit{wakeme} procedure presets the wakeup time to $\text{now}+n$ and suspends the calling process until the current time $\text{now}$ is equal to or exceeds $\text{alarmsetting}$. The priority associated with the suspended process provides the basis for the alarm-setting value.

The \textit{tick} function is assumed to be invoked automatically by hardware (e.g., through an interrupt) at regular intervals. It increments the value of the current time $\text{now}$, and wakes up the process with the lowest alarm setting time. The awakened process compares the time with its $\text{alarmsetting}$. If $\text{now}$ is less than $\text{alarmsetting}$, it goes back to sleep by executing the same \textit{wait} operation; otherwise, it performs a \textit{signal} operation to wake up the next process in the queue, which might have its alarm preset to the same time.

\section*{Mesa and Java Monitors}

The \textit{signal} operation as defined above causes the current process to be suspended immediately whenever another process is awakened. A simple variant of \textit{signal} is the \textit{notify} primitive, defined for the programming language Mesa (Lampson and Redell 1980). \textit{Notify} does not suspend the current process (the notifier). It only indicates to a waiting process, without actually activating it, that the corresponding condition has been satisfied; \textit{notify} does this by inserting the waiting process on a queue ready to reenter the monitor. The waiting process is resumed sometime after the current process exits the monitor.

In this scheme, however, a process suspended because of a particular condition $B$ was found \textit{false} cannot be guaranteed that the condition will still be \textit{true} when the process resumes execution. To understand why such a guarantee cannot be given, consider two processes, $p_1$ and $p_2$, both of which are currently blocked as a result of executing the statements:

\begin{verbatim}
if (!B1) c1.wait;
\end{verbatim}

and:

\begin{verbatim}
if (!B2) c2.wait;
\end{verbatim}

respectively. Assume that the process currently executing inside the monitor causes both conditions to become \textit{true} and indicates this fact by executing the statements $c1.\text{notify}$ and $c2.\text{notify}$. When it exits the monitor, one of the waiting processes, e.g., $p_1$, will be permitted to reenter the monitor. During its execution and after the \textit{notify} was issued, $p_1$ may modify the contents of any variable inside the monitor, possibly making the condition $B2$ \textit{false} again. It is also possible that another process enters the monitor after $p_1$ leaves. Consequently, when $p_2$ is permitted to reenter, it cannot assume $B2$ to be \textit{true}. Rather, it must reevaluate $B2$ to determine its current value and, if \textit{false}, suspend itself again.

In general, \textit{wait} statements should be enclosed in a loop of the form:

\begin{verbatim}
while (!B) c.wait;
\end{verbatim}

instead of writing:

\begin{verbatim}
if (!B) c.wait;
\end{verbatim}

as would be done in monitors using \textit{signal}. While apparently more complex than the Hoare \textit{signal} semantics, the \textit{notify} definitions admit to a more efficient implementation,
may result in fewer context switches, and also can be used without explicit condition variables. The cost is decreased performance, due to looping and testing.

Another version of notify appears as part of the monitorlike facility defined in the popular Java language (Lea 1997). To control the sharing of data within an object, the particular methods or operations on the object are declared synchronized. Threads that access an object through “synchronized” methods do so in a mutually exclusive fashion, much like monitor procedures. Within synchronized object methods, wait and notify primitives may be employed for synchronization. However, there are no condition variables in Java; equivalently, each object can be viewed as having one (implicit) condition variable.

Both Mesa and Java also have a broadcast form of notify that awakens all threads that are blocked, instead of just one. Another useful feature available in both languages is a timeout parameter that may be associated with a wait. If a thread fails to be “notified” by the specified time, it still unblocks. Such timeouts are very useful for detecting errors, such as deadlocks or runaway processes.

3.1.2 Protected Types

The synchronization primitives of monitors, wait and signal or notify, may be distributed throughout the procedure code. It is also necessary to insert explicit tests before each wait. However, in almost all applications, the test and waits occur, if at all, in the first executed statement of a monitor procedure. A signal or notify, if issued, occurs as the last-executed statement. Protected types, as defined in the Ada95 language (ADA 1995), are an attractive alternative to monitors that take advantage of this common behavioral pattern. They do so by factoring out the waiting condition at the start of a procedure and providing an implicit signal at the end. The ideas underlying protected types are a combination of those appearing in monitors and conditional critical regions (Hoare 1972; Brinch Hansen 1973).

A protected type is an encapsulated object with public access procedures called entries. Each procedure (entry) may have an associated Boolean guard, which is a condition that must evaluate to true before the procedure is eligible for execution. A condition is reevaluated when a procedure exits a protected type and only if there are one or more tasks queued or blocked on that condition. Like monitor procedures, protected-type procedures are also mutually exclusive; only one procedure at a time can be executed.

EXAMPLE: Bounded Buffer

To illustrate these notions, we use the standard example of a bounded buffer, as specified in the last section. An object is defined in two parts: a specification that declares the exported procedures and data, and an implementation that lists the code or body of the object.

For the bounded buffer, the specification part describes the deposit and remove interfaces, and the internal or private data of the Bounded_Buffer object:

```plaintext
protected Bounded_Buffer {
    entry deposit(char data);
    entry remove(char data);
    private
        char buffer[n];
        int nextin=0,nextout=0,full_cnt=0;
}
```
Section 3.2 Distributed Synchronization and Communication

In the implementation part, each entry starts with the keyword when. For example, the condition for deposit is the expression when (full_cnt < n); thus, a call on deposit will not be permitted to proceed until full_cnt < n. The body of the protected type is:

```plaintext
protected body Bounded_Buffer {
    entry deposit(char c)
        when (full_cnt < n) {
            buffer[nextin] = c;
            nextin = (nextin + 1) % n;
            full_cnt = full_cnt + 1;
        }
    entry remove(char c)
        when (full_cnt > 0) {
            c = buffer[nextout];
            nextout = (nextout + 1) % n;
            full_cnt = full_cnt - 1;
        }
}
```

In addition to procedures, protected types also support functions. The distinction is significant. Procedures may modify the private variables and must be executed under mutual exclusion. Functions, on the other hand, only return values. They have no side effects, i.e., they cannot modify any variables private to the protected type. Consequently, they can be executed in parallel. This constraint—concurrent function execution, one-at-a-time procedure execution, and mutual exclusion between function and procedure executions—is called a readers/writers constraint. (This type of problem is discussed further in Section 3.3.1.)

3.2 DISTRIBUTED SYNCHRONIZATION AND COMMUNICATION

The synchronization mechanisms presented so far assume that processes share some portion of memory. For centralized systems where process isolation and encapsulation is desirable (e.g., for security reasons and for distributed systems where processes may reside on different nodes in a network), IPC typically occurs through either message passing or remote procedure calls. Both techniques are described in the following sections. Section 3.2.3 covers the important application of distributed mutual exclusion.

3.2.1 Message-Based Communication

Probably the most common form of IPC is message passing, with some variant of send and receive operations. Unlike shared-memory primitives, such as the P and V operations on semaphores, there is not a universally accepted definition of send and receive. Depending on a particular system, many different interpretations may be found. A generic form of these two primitives is:

```plaintext
send(p, m)
receive(q, m)
```
Chapter 3  Higher-Level Synchronization and Communication

The operation \texttt{send()} causes the message, stored in or pointed to by the variable \textit{m}, to be transmitted to the process \textit{p}. The \texttt{receive()} operation is used to obtain a message. It names the process \textit{q} from which a message is expected; when the message arrives, it is deposited in the variable \textit{m}. Generally, the message could be arbitrary data, such as a string of binary bits, or it could be a strongly typed object. Some implementations permit the parameters \textit{p} or \textit{q} to be omitted.

One way to extract and understand the many possibilities for defining \texttt{send} and \texttt{receive} is to answer some fundamental questions:

1. When a message is emitted, must the sending process wait until the message has been accepted by the receiver or can it continue processing immediately after emission?
2. What should happen when a \texttt{receive} is issued and there is no message waiting?
3. Must the sender name exactly one receiver to which it wishes to transmit a message or can messages be simultaneously sent to a group of receiver processes?
4. Must the receiver specify exactly one sender from which it wishes to accept a message or can it accept messages arriving from any member of a group of senders?

There are two possible answers to the first question. If the sending process is blocked until the message is accepted, the \texttt{send} primitive is called \textbf{blocking} or \textbf{synchronous}. On the other hand, the process may proceed while the message is being transferred to the destination process, the \texttt{send} is said to be \textbf{nonblocking} or \textbf{asynchronous}.

The second question distinguishes two types of \texttt{receive} operations, analogous to the blocking and nonblocking \texttt{send}. In the first case, if a \texttt{receive} is issued and there is no message waiting, the process is blocked; such a \texttt{receive} is called \textbf{blocking} or \textbf{synchronous}. The second alternative is to permit the receiving process to continue when no messages are waiting; this variant of \texttt{receive} is called \textbf{nonblocking} or \textbf{asynchronous}.

The last two questions address the problem of naming. A process may wish to transmit a message nonselectively to any of a number of processes that may wish to receive it. This operation is usually referred to as \textbf{broadcasting} or \texttt{send} with \textbf{implicit naming}, because there may be many potential receiving processes and they are not named explicitly. A restricted but widespread form of broadcast is the \texttt{multicast send} that transmits a message to all members of a named group of processes. Similarly, a process may receive messages from any of a number of possible senders, depending on their order of arrival. A \texttt{receive} with \textbf{implicit naming} is used for this purpose.

The semantics of various \texttt{send/receive} primitives can be summarized as follows (Shatz 1984):

<table>
<thead>
<tr>
<th>send</th>
<th>blocking</th>
<th>nonblocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>explicit naming</td>
<td>Send message m to receiver r. Wait until message is accepted.</td>
<td>Send message m to receiver r.</td>
</tr>
<tr>
<td>implicit naming</td>
<td>Broadcast message m. Wait until message is accepted.</td>
<td>Broadcast message m.</td>
</tr>
</tbody>
</table>
Section 3.2  Distributed Synchronization and Communication

### Table: Receive Primitives

<table>
<thead>
<tr>
<th>Type</th>
<th>Blocking</th>
<th>Nonblocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>explicit naming</td>
<td>Wait for message from sender $s$.</td>
<td>If there is a message from sender $s$, then receive it; otherwise proceed.</td>
</tr>
<tr>
<td>implicit naming</td>
<td>Wait for message from any sender.</td>
<td>If there is a message from any sender, then receive it; otherwise proceed.</td>
</tr>
</tbody>
</table>

Combinations of the above primitives may be provided to implement a variety of different facilities for process interactions. Blocking send/receive primitives are powerful mechanisms for solving a variety of process coordination problems. This is because both the sender and receiver are fully synchronized at the point of communication. The receiving process knows that the sender will continue only after the message has been accepted; the sender knows that at its continuation point, the message has been safely received. In the case of send, the blocking version with implicit naming is of little practical use because of the complexities of synchronizing a broadcasted send with all possible receivers; hence, broadcasting is usually done in the nonblocking form. On the other hand, the blocking versions of receive are both very useful and much more common than their nonblocking forms. For example, consider a server process such as a printer, a stand-alone buffer processes, or a file server, which accepts service request messages from clients. The natural form of receive is blocking with implicit naming, since the server should accept messages from any of a group of clients, perhaps not known in advance.

### CASE STUDY: SYSTEM V

The V distributed system (Cheriton 1988) is one example that includes blocking operations for IPC. Their basic scheme involves a pair of synchronous send/receive operations of short fixed-length messages (32 bytes). A client outputs a message using a blocking send; a server accepts a message with a blocking receive and later transmits a reply message. The client issues a receive operation and remains blocked until it receives the reply. This sequence of events occurs most commonly in the higher-level remote procedure call interaction that we present in the next section. System V also supports prominently a multicast facility.

The nonblocking operations with explicit naming are also important and have many applications. However, they are at a higher level than the blocking ones since they require a form of built-in buffering to hold all sent messages not yet requested by a receiving process. Here, the bounded-buffer problem is solved implicitly within the primitives themselves. Because, by definition, there is no waiting with the nonblocking send and receive, these operations do not have the flavor of synchronization primitives but provide the functions of a more general “mailing” facility. Both types of problem—synchronization and the asynchronous exchange of messages—are important in a general-purpose computing facility; thus many operating systems (OS) provide several different versions of send and receive to satisfy the needs of different users.
Message Channels, Ports, and Mailboxes

Several indirect forms of message passing exist, wherein processes communicate through a named intermediary rather than directly. One common form uses named channels as the intermediary. These are abstractions of wires or other media (e.g., radio waves) that may connect processes. A process communicates with another by sending a message over a particular channel, and a receiver obtains a message from a specific channel. In general, there may be many channels connecting processes; the channels could be unidirectional or bidirectional, and the communications could be synchronous or asynchronous. In the latter case, the channel requires some memory to act as a buffer.

The best known example of a named-channel system is the implementation of Hoare’s communicating sequential processes (CSP) in the Occam language (Hoare 1978, 1984; Inmos 1984, 1988). In CSP/Occam, channels are unidirectional, messages are typed, and communication is synchronous.

For example, suppose that channel ch1 connects process p1 to p2, and can carry messages of type char. p1 could send the character “a” to p2 with the command:

\[
\text{send}(\text{ch1}, \text{’a’});
\]

p2 accepts the message with the command:

\[
\text{receive}(\text{ch1}, x);
\]

using a variable x of type char. p1 will wait until p2 is ready to receive, and p2 waits until p1 is ready to send. Communication occurs instantaneously, if and when it occurs. At that point, x is assigned the value “a”, and both processes continue in execution.

To simplify the use of the communication primitives, receive statements may be implemented in the form of guarded commands (Dijkstra 1975). These are similar to the when clauses defined with protected types (Section 3.1.2). The syntax of the guarded commands is:

\[
\text{when } (C) \ S
\]

where C is an arbitrary Boolean expression and S is a set of executable statements. The expression C is called the guard. As with protected types, the statements in S are executed only when the expression C evaluates to true. The difference is that C also may contain receive statements. These evaluate to true only if the sending process is ready to execute the corresponding send on the same channel. Several guarded input commands may be enabled at the same time and are potentially executable. One is selected arbitrarily; the others have no effect. This feature of CSP is a form of a “selective input,” similar to the selective accept statement of Ada presented in the next section. It permits processes to communicate in a nondeterministic fashion, depending on the availability of input data at run time. The CSP/Occam ideas are illustrated with our bounded-buffer example.

EXAMPLE: Bounded Buffer with CSP/Occam

Since all send and receive primitives are blocking, we have the following problem: When the buffer is completely full, it can only send data to the consumer. Similarly, when the

\[1\text{We use a different syntax than that of CSP or Occam, mainly because their syntax is relatively unusual.}\]
buffer is completely empty, it can only receive data from the producer. The problem occurs when the buffer is partially filled. In this case, its action depends on whether the producer is ready to send more data or the consumer is ready to receive any data. Determining whether the producer is ready can be solved by embedding the receive statement in a guarded clause. Determining whether the consumer is ready can be done only by asking the consumer to first send a request to the buffer, followed by the data-receiving statement. The buffer may then detect the presence of a request by embedding its request-receiving statement in a guarded clause.

The complete protocol is as follows: We define three channels, deposit, request, and remove, named to reflect their purpose. They are used as follows: The producer process inserts an element named data into the buffer by sending it on the channel named deposit to the process BoundedBuffer:

send(deposit, data);

The consumer process requires two communications to perform its function. First, it sends a request to BoundedBuffer on the channel named request and follows this by a receive command on the remove channel.

send(request);
receive(remove, data);

Note that the send has an empty message; it is sufficient that BoundedBuffer just receives a signal on the request channel. The received message is stored in the variable data.

The buffer process takes the following form:

process BoundedBuffer {
    message buffer[n];
    int nextin=0, nextout=0, full_cnt=0;
    while (1) {
        when ( ((full_cnt < n) && receive(deposit, buffer[nextin])) ) {
            nextin = (nextin + 1) % n;
            full_cnt = full_cnt + 1;
        }
        or
        when ( ((full_cnt > 0) && receive(request)) ) {
            send(remove, buffer[nextout]);
            nextout = (nextout + 1) % n;
            full_cnt = full_cnt - 1;
        }
    }
}

The two when clauses separated by or indicate possible nondeterministic selection. If full_cnt is neither 0 nor n, and sends are pending on both the request and deposit channels, then one of the sends is selected arbitrarily, and the code following the guard is executed.
Chapter 3 Higher-Level Synchronization and Communication

In the absence of receive primitives with implicit naming or the guarded input commands used in CSP/Occam, a process is limited to only one potential source of messages at any given time. If the receive is nonblocking, it can be embedded into a “busy-wait” loop to simulate the effect of implicit naming. However, this solution is quite inefficient. A common way to avoid the limitations of explicit naming which does not suffer from the performance problem of busy-waiting is to use indirect communication with queues or buffers. The basic idea is to make visible to the processes those queues holding messages between senders and receivers. That is, a send primitive does not use the name of a process as its destination; rather, it addresses the queue to which the message is to be appended. Similarly, a receiving process names the queue from which a message is to be accepted.

A queue that can be named by more than one sender and more than one receiver process is called a mailbox. This scheme is illustrated in Figure 3-2a. Mailboxes provide the most general communication facility, because any of the $n$ senders may emit messages that may be intercepted by any of the $m$ receivers. Unfortunately, in a distributed environment, the implementation of the receive operation can be costly, because receives referring to the same mailbox may reside on different computers. Therefore, a limited form of a mailbox, usually called a port, is frequently implemented. A port is associated with only one receiver. Messages that come from different processes but address the same port are sent to one central place associated with the receiver, as shown in Figure 3-2b.

FIGURE 3-2. Indirect process communication: (a) through a mailbox; and (b) through a port.
Section 3.2   Distributed Synchronization and Communication  83

CASE STUDY: UNIX IPC

Pipes are the standard IPC scheme on uniprocessors in UNIX systems (e.g., [Lefler et al. 1989]). A pipe may be used to connect two or more processes into a sequence of the form:

\texttt{p1 | p2 | \ldots | pn}

where each vertical bar represents a pipe. It connects the standard input of the process on the left to the standard output of the process on the right. Thus, a pipe can be interpreted as an unnamed communication channel with memory that handles variable-length messages of byte streams.

The more elaborate facilities that are defined for distributed UNIX systems employ sockets. Sockets are named endpoints of a two-way communication channel that may be established between two processes running on different machines. To establish a channel, one of the processes acts as a server, whereas the other acts as a client. The server first binds its socket to the IP address of its machine and a port number; this becomes the incoming communication channel on which the server listens. By issuing an \texttt{accept} statement, the server blocks until the client issues a corresponding \texttt{connect} statement. The \texttt{connect} statement supplies the client’s address and port number to complete the two-way communication channel. Both processes may then send data to each other via this dynamically established connection.

3.2.2 Procedure-Based Communication

One problem with the \texttt{send/receive} primitives is their low level. The programmer is forced to abandon the structuring concepts of procedures and other high-level modules, and use these message passing primitives in much the same way as one uses \texttt{P} and \texttt{V} operations on semaphores. \textbf{Remote Procedure Calls} (RPC) and \textbf{rendezvous} have been proposed as higher-level mechanisms for process communication to overcome this problem. They also provide a natural way to program client/server interactions, which are the basis of many distributed applications. A client requests a service by invoking an RPC or performing a rendezvous with a service procedure, which may reside on another machine or node of a network.

As far as the calling process is concerned, both RPC and rendezvous have the same effect as regular procedure calls. Each transfers control to another procedure while suspending the caller; a return statement executed by the called procedure then transfers control back to the caller, where execution continues with the instruction following the call. Consequently, the called procedure may be viewed as a service invoked by the caller, i.e., the client.

Ideally, the caller should not be aware of the fact that the called procedure is remote. However, the fact that the caller and the procedure are on different machines cannot be made completely transparent. The main distinction between regular procedures and remote procedures is the inability to pass pointers as parameters to the remote procedure. This is because the client and the server do not share any memory. To exchange information, the calling procedure must pass copies of all input parameters to the called procedure. Similarly, results are returned to the caller as explicit copies of the values.
Chapter 3 Higher-Level Synchronization and Communication

At the implementation level, remote procedures are quite different from regular procedures. Since a remote procedure executes in a separate address space on a different computer, it cannot become part of the calling process. Rather, a separate process must execute the called procedure. This process may be created dynamically at each call or statically as a dedicated service.

Assume that a client issues the call:

\[ \text{res} = f(\text{params}) \]

where \( f \) is a remote procedure. In the simplest implementation, a permanent process, e.g., \( \text{RP}_\text{server} \), contains the code for the remote procedure \( f \) and repeatedly executes this embedded code for each call. Coordination of the caller and callee processes can be accomplished using the low-level \text{send/receive} primitives. Each remote call by a client is implemented as a pair of synchronous \text{send} and \text{receive} statements. The client initiates the call by transmitting the function name and the necessary input parameters (\text{params}) to the server; the server executes the specified function, \( f \), locally, and returns the results (\text{res}) to the client. This implementation is illustrated by the following code skeleton:

**Calling Process:**

```
...-
send(RP_server, f, params);
...
```

**Server Process:**

```
process RP_server {
    while (1) {
        receive(caller, f, params);
        res = f(params);
        send(caller, res);
    }
}
```

Now consider how such procedure-based communications can be incorporated into a high-level language. Doing so greatly expands the applicability of remote procedures. With the RPC scheme as described so far, the interaction is always asymmetric: a dedicated server must be set up, whose sole purpose is to accept calls from clients and to execute their remote procedure requests. In contrast, allowing any process to accept remote calls from other processes results in a general IPC and coordination mechanism. The receiving process is able to perform arbitrary computations on its own, prior to or following any calls. It may also maintain its own state between any calls. Finally, the mechanism is fully symmetrical: Any process may play the role of a client by invoking a remote procedure in another process, and any process may play the role of a server by accepting remote calls.

To implement such a scheme, generally referred to as rendezvous, we must provide mechanisms for clients to issue remote procedure calls and for servers to accept them. We describe the approach taken in Ada (Ada 1995), which pioneered the rendezvous concept. From the calling process perspective, calling a remote procedure is similar to calling a local procedure; the main differences are the inability to pass pointers to a different machine and the possibility of remote machine failure. However, the receiving process must be able to express its willingness to accept a call. In the simplest case, an \text{accept} statement is used. This statement defines a procedure that may be invoked...
Section 3.2  Distributed Synchronization and Communication  85

remotely by another process. It has the following form:

\[
\text{accept } f(\text{params}) S
\]

where \( f \) is the procedure name, \( \text{params} \) are its parameters, and \( S \) is the procedure body. Its counterpart in the calling program is a procedure call of the form:

\[
q. f(\text{params})
\]

where \( q \) is the name of the remote process, and \( \text{params} \) are the actual parameters passed to the \text{accept} statement.

To understand the semantics of a rendezvous, assume that \( p \) is the process issuing the call, and \( q \) is the process executing the corresponding \text{accept} statement.

1. If the process \( p \) issues the call before process \( q \) has reached the corresponding \text{accept}, \( p \) becomes blocked. When \( q \) executes the \text{accept} statement, it proceeds by executing the procedure body, \( S \). After completion of this rendezvous, both processes continue their execution concurrently. This is illustrated in Figure 3-3a.

2. If process \( q \) reaches the \text{accept} statement before the procedure call has been issued by \( p \), \( q \) becomes blocked. As soon as a call is made, the rendezvous takes place and \( q \) resumes execution. While \( q \) executes the procedure body \( S \), process \( p \) is suspended. As in the first case, both processes continue concurrently upon termination of \( S \). Figure 3-3b illustrates this case.

In this fully exposed form, the \text{accept} statement suffers from the same problem as a blocking \text{receive} with explicit naming. It does not permit the process executing this statement to wait selectively for the arrival of one of several possible requests. This would make sharing the code limited to a predetermined order of arrivals.

![Diagram](image)

**FIGURE 3-3.** Execution of a rendezvous: (a) calling process is delayed; and (b) called process is delayed.
CSP/Occam solved this problem by introducing nondeterministic selective input. Similarly, Ada provides a `select` statement that permits several `accepts` to be active simultaneously. A Boolean guard also may be associated with each `accept`, which prevents its execution when its value is `false` at the time the `select` statement is executed. The `accept` statements are embedded into a `select` statement as follows:

\[
\text{select} \{ \\
\text{[when B1:] accept E1(...) S1 ;} \\
\text{or [when B2:] accept E2(...) S2 ;} \\
\text{or} \\
\text{...} \\
\text{[when Bn:] accept En(...) Sn ;} \\
\text{[else R]} \\
\}\n\]

The construct has the following semantics: The `when` clause, associated with each `accept` statement, is optional, as indicated by the square brackets. When omitted, its value is assumed to be `true`.

Execution of the `select` statement causes one of the embedded `accepts` to be executed. To be selected, the corresponding `when` clause must yield the value `true`, and there must be at least one pending procedure call to that `accept` statement performed by another (currently blocked) process. If there is more than one such eligible `accept` statement, the system chooses among these according to a fair internal policy. On the other hand, if none of the `accept` statements is eligible, the `else` clause containing the statement `R` is executed.

The `else` clause is optional. If it is omitted, the `select` statement behaves as follows: If none of the `when` clauses evaluates to `true`, an error is generated. Otherwise, the process is suspended until a call to one of the `accept` statements is performed for which the `when` clause evaluates to `true`. At this time, the body `Si` of that `accept` statement is executed and both processes continue concurrently.

**EXAMPLE: Bounded Buffer**
We present a solution to the Bounded Buffer problem using a selective `accept` statement. The buffering system is implemented as a separate server process:

```
process Bounded_Buffer {
    ... 
    while(1) {
        select {
            when (full_cnt < n):
                accept deposit(char c) {
                    buffer[nextin] = c;
                }
        }
    }
```

\(^2\)We have modified the original syntax using a C-like approximation.
Section 3.2 Distributed Synchronization and Communication

\[
\text{nextin} = (\text{nextin} + 1) \mod n; \\
\text{full}_\text{cnt} = \text{full}_\text{cnt} + 1;
\]

or

\[
\text{or}
\]

\[
\text{when} (\text{full}_\text{cnt} > 0):
\]

\[
\text{accept remove(char c)} \{ \\
\text{c} = \text{buffer}[\text{nextout}]; \\
\text{nextout} = (\text{nextout} + 1) \mod n; \\
\text{full}_\text{cnt} = \text{full}_\text{cnt} - 1;
\}
\]

The two operations on the buffer may be invoked by other client processes issuing the procedure calls:

\[
\text{Bounded\_Buffer.deposit(data)}
\]

and

\[
\text{Bounded\_Buffer.remove(data)},
\]

respectively. This code is at the same high level as the analogous version used in protected types and is slightly more abstract than the CSP/Occam version, both described in previous sections.

In summary, procedure-based communication can be considered as closing the gap between procedure-oriented synchronization schemes, such as monitors and protected types, and distributed methods based on message passing. From the programmer’s point of view, remote procedures are high-level objects that fit well into the philosophy of block-structured programming languages. On the other hand, their implementation in terms of simple message-passing primitives makes them suitable to distributed environments.

3.2.3 Distributed Mutual Exclusion

The CS problem, discussed in the last chapter within the context of a shared memory architecture, also has a distributed version. In this case, we have a set of processes that share a resource in a mutually exclusive fashion, but the processes can communicate only through message passing or remote procedures. There are no shared memory, no shared clock, and possibly significant message delays. Consequently, different solution techniques are required.

The distributed CS problem arises in different situations. Several processes or nodes in a network could share some hardware resource, such as a printer, a satellite or GPS link, a special-purpose processor, or another expensive or rare piece of equipment. Similarly, some expensive software resource, such as a database or filing system, may be used by several distributed processes. The processes require mutually exclusive access to the resource. One reason is to preserve the integrity of the shared data. A less-obvious reason arises in systems where software resources, especially files, are replicated over a
network so that each user process can efficiently access a nearby copy. A read access may be done using any replica, but a write request must be propagated to all replicas. One way to ensure consistency of the replicas is to enforce the mutual exclusion constraints on all write accesses.

There are several approaches to ensure distributed mutual exclusion. Perhaps the most straightforward involves a central controller. Whenever a process requires the resource, i.e., wants to enter a CS, it sends a request to the controller and waits for its request to be granted. After using the resource, it informs the controller, which then may allocate it to another waiting process. One disadvantage of this approach is that it relies on the correct operation of the controller. If the controller fails, the resource allocation fails, thus eliminating one of the benefits of distribution—resilience on failures of individual nodes. Another disadvantage of a centralized allocator is that it is potentially a performance bottleneck. Simultaneous requests by many processes can slow down the entire system.

The other extreme is a fully distributed approach where a process that needs the resource must engage in a lengthy arbitration involving all processes. The problems with this scheme include the large amount of message passing, a requirement for accurate time stamps on messages, and the difficulty of managing node or process failures. A practical and elegant compromise is the token ring method that we now develop in detail (LeLann 1977).

Assume \( n \) processes, \( p_0 \) to \( p_{n-1} \), where each process \( p_i \) continuously loops through a critical section \( CS_i \) and a noncritical part, \( program_i \). The idea of the token ring is to define a separate controller process, \( controller_i \), associated with each \( p_i \). The controllers are arranged in a circle and repeatedly pass a token from neighbor to neighbor (Fig. 3-4). The token represents the opportunity for a controlled process to enter its CS. If a process \( p_i \) wishes to enter its \( CS_i \), it sends a request message (Request\( _{CS} \)) to \( controller_i \); the controller grants the request when it receives the token, and it keeps the token while \( p_i \)

![FIGURE 3-4. Distributed mutual exclusion with a token ring.](image-url)
Section 3.2  Distributed Synchronization and Communication  89

is in $C_S_i$. The process informs its controller that it has left $C_S_i$ using a second message ($Release_{CS}$). The controller then passes the token on to its neighbor.

The controller algorithm and $p_i$ skeleton are described using the selective accept statements as follows:

```plaintext
process controller[i] {
    while(1) {
        accept Token;
        select {
            accept Request_{CS}() {busy=1;}
            else null;
        }
        if (busy) accept Release_{CS}() {busy=0;}
        controller[(i+1) % n].Token;
    }
}

process p[i] {
    while(1) {
        controller[i].Request_{CS}();
        CSi;
        controller[i].Release_{CS}();
        programi;
    }
}
```

The system is represented by two arrays of processes, $p[n]$ and $controller[n]$. The code for the process is straightforward. Prior to entering its CS, it issues the blocking call $Request_{CS}()$ to its controller; after exiting the CS, it informs the controller using the call $Release_{CS}()$. Note that neither call has any parameters or executable body of code. Their sole purpose is to serve a synchronization signals between the process and the controller.

The passing of the token around the ring is implemented using the pairs of statements $accept Token$ and $controller[(i+1) % n].Token$. Note than no real data item (token) is passed between the controllers. Instead, the token is represented by the current locus of control. At any moment in time, only one controller is active; all others are blocked on their $accept Token$ calls.

When the controller receives the token, it checks if its process wishes to enter the CS. This is accomplished using the selective $accept$ of $Request_{CS}$. If no call is pending, the controller executes the else clause, which is empty. The controller then immediately passes the token to its neighbor. If a call is pending, the controller accepts it and sets its $busy$ flag to 1. As a result, it blocks itself by executing the statement $accept Release_{CS}()$ until the process leaves the CS.

Mutual exclusion is clearly satisfied, because only the controller currently holding the token may grant CS entry; all others are blocked. Starvation is also not possible; once a process has issued the request, it waits for at most one complete round of the token through the other controllers, but then its controller will accept this request. The
principal disadvantages are the overhead resulting from the additional controller processes and the need for continuous message passing around the token ring, even when there is no demand for a CS. Note also that a failure of a single controller will make the system inoperable. Extensions to the algorithm exist that make it more resilient against such failures.

3.3 OTHER CLASSIC SYNCHRONIZATION PROBLEMS

In the previous sections, we have used the CS problem and the bounded-buffer problem to study various approaches to process synchronization and communication, as well as the capabilities of different synchronization mechanisms. These problems are also important in their own right in concurrent programming and in OSs. In this section, we present four well-known problems that have been used extensively in the literature and are abstractions of important OS applications.

3.3.1 The Readers/Writers Problem

This problem, first formulated and solved by Courtois, Heymans, and Parnas (1971), arises when two types of processes, referred to as readers and writers, must share a common resource, such as a file. Writers are permitted to modify the state of the resource and must have exclusive access. On the other hand, readers only can interrogate the resource state and, consequently, may share the resource concurrently with an unlimited number of other readers. In addition, fairness policies must be included, for example, to prevent the indefinite exclusion or starvation of readers, writers, or both. Depending on fairness requirements, there are several different variants of the basic readers/writers problem.

In the original paper, two solutions using semaphore $P$ and $V$ operations were given. The first adopts the policy that readers have priority over writers. That is, no reader should be kept waiting unless a writer has already obtained the permission to use the resource. Note that this policy will result in an indefinite postponement (starvation) of writers if there is a continuous stream of read requests (see exercise 12).

CASE STUDY: READERS AND WRITERS IN LINUX

The read/write spin locks of Linux, discussed in Section 2.4.2, implement this policy. The read_lock macro is allowed to proceed concurrently with others, unless the resource is already locked for writing. Thus, new readers are admitted indefinitely, whereas the write_lock macro remains blocked until the number of readers drops to zero.

The second policy gives priority to writers. When a writer arrives, only those readers already granted permission to read will be allowed to complete their operation. All new readers arriving after the writer will be postponed until the writer’s completion. This policy will result in the indefinite postponement of read requests when there is an uninterrupted stream of writers arriving at the resource.

With these two priority policies, one of the process types has absolute priority over the other. To prevent this from occurring, Hoare (1974) proposed a fairer policy, defined by the following rules:
Section 3.3 Other Classic Synchronization Problems

1. A new reader should not be permitted to start during a read sequence if there is a writer waiting.
2. All readers waiting at the end of a write operation should have priority over the next writer.

We show this solution in the form of a monitor. For this problem and many others, it is convenient to add yet another function to our monitor primitives. If \( c \) is a condition variable, then the Boolean function \( \text{empty}(c) \) returns \( true \) whenever there are one or more processes waiting on the \( c \) queue as a result of a \( c\.wait \); it returns \( false \) otherwise.

The monitor provides four procedures, used by reader and writer processes as follows:

1. \textit{start\_read}: Called by a reader that wishes to read.
2. \textit{end\_read}: Called by a reader that has finished reading.
3. \textit{start\_write}: Called by a writer that wishes to write.
4. \textit{end\_write}: Called by a writer that has finished writing.

The resource itself is not part of the monitor. The monitor only guarantees that it can be accessed safely by a reader between the calls to \textit{start\_read} and \textit{end\_read}, and by a writer between the calls \textit{start\_write} and \textit{end\_write}. The monitor is implemented as follows:

```
monitor readers/writers {
    int read\_cnt=0, writing=0;
    condition OK\_to\_read, OK\_to\_write;

    start\_read() {
        if (writing || !empty(OK\_to\_write)) OK\_to\_read.wait;
        read\_cnt = read\_cnt + 1;
        OK\_to\_read.signal;
    }

    end\_read() {
        read\_cnt = read\_cnt - 1;
        if (read\_cnt == 0) OK\_to\_write.signal;
    }

    start\_write() {
        if ((read\_cnt != 0) || writing) OK\_to\_write.wait;
        writing = 1;
    }

    end\_write() {
        writing = 0;
        if (!empty(OK\_to\_read)) OK\_to\_read.signal;
        else OK\_to\_write.signal;
    }
}
```
The variable \texttt{read\_cnt} keeps track of the number of processes currently reading the resource data, whereas \texttt{writing} indicates whether a writer is currently using the resource. The Boolean condition in the first statement of \texttt{start\_read} guarantees that rule 1 of the priority policy is obeyed. That is, a new reader is not admitted if there is a writer currently writing or waiting to write. The reader is blocked on the \texttt{OK\_to\_read} queue. The \texttt{OK\_to\_read.signal} in \texttt{end\_write} wakes up the first waiting reader on this queue.

The \texttt{OK\_to\_read.signal} in \texttt{start\_read} ensures that once one reader is permitted entry, all other waiting readers will follow immediately one after the other. Thus, rule 2 of the priority policy also is satisfied.

### 3.3.2 The Dining Philosophers Problem

To illustrate the subtleties of deadlock and indefinite postponement in process synchronization, Dijkstra (1968) formulated and proposed a solution to a toy problem that became known as the dining philosophers problem.

Five philosophers, \(p_i\) (1 \(\leq\) \(i\) \(\leq\) 5), sit around a table in the middle of which is a bowl of spaghetti. There is a plate in front of each philosopher and there are five forks, \(f_i\) (1 \(\leq\) \(i\) \(\leq\) 5), on the table, one between each two plates, as illustrated in Figure 3-5. At unspecified times, each of the philosophers may wish to eat. To do that, a philosopher \(p_i\) must first pick up the two forks \(f_i\) and \(f_{(i+1)\%5}\) next to \(p_i\)’s plate. Only then is \(p_i\) allowed to start eating. When \(p_i\) is finished (after a finite time), \(p_i\) places the forks back on the table. The problem is to develop a fork acquisition protocol so that none of the philosophers starve because of the unavailability of forks.

There are three main concerns when developing a solution to this problem:

1. **Deadlock:** A situation must be prevented where each philosopher obtains one of the forks and is blocked forever waiting for the other to be available.

2. **Fairness:** It should not be possible for one or more philosophers to conspire in such a way that another philosopher is prevented indefinitely from acquiring its fork.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{fig3-5.png}
\caption{Five dining philosophers.}
\end{figure}
3. Concurrency: When one philosopher, e.g., \( p_1 \), is eating, only its two immediate neighbors (\( p_5 \) and \( p_2 \)) should be prevented from eating. The others (\( p_3 \) and \( p_4 \)) should not be blocked; one of these must be able to eat concurrently with \( p_1 \).

Assume that each philosopher is a process and each fork is a resource. A process \( p(i) \) (\( 1 \leq i \leq 5 \)) loops continuously through two phases, a thinking phase \( \text{think}(i) \) and an eating phase \( \text{eat}(i) \), as shown in the following code skeleton:

```c
p(i) {
    while (1) {
        think(i);
        grab_forks(i);
        eat(i);
        return_forks(i);
    }
}
```

The action is in the entry and exit protocols for eating, \( \text{grab}_{\text{forks}}(i) \) and \( \text{return}_{\text{forks}}(i) \), respectively. Let us develop a solution using semaphores. The forks are represented by an array of binary semaphores \( f[1], f[2], \ldots, f[5] \). The most obvious scheme has each process acquiring its left fork and then its right fork in sequence:

```c
\text{grab}_{\text{forks}}(i): P(f[i]); P(f[(i + 1) \% 5]);
\text{return}_{\text{forks}}(i): V[f[i]]; V(f[(i + 1) \% 5]);
```

This solution guarantees fairness as long as processes blocked on the same semaphore are serviced in an unbiased order, i.e., using first-in/first-out (FIFO) or a random order. Under this assumption, a philosopher blocked on a semaphore will eventually acquire this semaphore and may proceed. Concurrent execution of nonneighboring philosophers is also possible. Unfortunately, this solution can easily lead to a deadlock situation. Suppose that each process moves from its “think” phase to its “eat” phase at approximately the same time and obtains its left fork, i.e., successfully passes through its \( P(f[i]) \). Then they each attempt to acquire their right fork (\( P(f[(i + 1) \% 5]) \)); they will all block, waiting for their right neighbor (also blocked) to release its fork. The release will never happen, because for a process to release a fork, it must first acquire both. The processes are blocked forever; deadlock has occurred.

To avoid this deadlock, we must ensure that the circular wait described above can never occur. One easy solution is to implement a global counter (e.g., a semaphore) that allows at most \( n - 1 \) philosophers (i.e., 4 in the above example) to compete for their forks.

Another solution, which does not require a global semaphore, is to have one process, e.g., \( p(1) \), pick up its right fork first instead of the left fork first. This breaks the circular condition, since two of the philosophers (\( p(1) \) and \( p(2) \)) will be competing for the same fork \( f(2) \) first, before attempting to acquire the other. Since only one will win this competition, the other will not be able to continue, and hence its other fork will remain free. Unfortunately, this solution violates the concurrency requirement: As one philosopher, e.g., \( p(1) \) is eating, all others could be blocked in a chain of requests to get their respective left forks. A better solution that avoids this problem is to divide
all philosophers into odd- and even-numbered ones. One group, e.g., the odd-numbered philosophers, pick up their right fork before the left; the even-numbered ones pick up their left fork first. This way, pairs of neighboring philosophers compete for the same fork, which avoids the unbounded chain of blocked processes waiting for each other.

3.3.3 The Elevator Algorithm

Consider an elevator in a multistory building. We present a monitor that governs the motion of the elevator in response to requests by people wishing to travel to other floors. (In Chapter 11, we will see how the same algorithm may be used to schedule the motion of the read/write head of a magnetic disk.)

Figure 3-6 shows the basic organization of the elevator. There are \( n \) floors, 1 being the lowest and \( n \) the highest. There are two types of control buttons. At each floor, a
person may press a call button to indicate she wishes to use the elevator. (We ignore the fact that most modern elevators provide two such call buttons, one to go up and one to go down; see exercise 17.) Inside the elevator cage, a panel with buttons numbered 1 through \( n \) allows the person to specify which floor she wishes to move to.

The algorithm controlling the elevator is as follows. At any point in time, the elevator has a current direction of motion (up or down). It maintains this direction for as long as there are requests for floors in that direction. When there are no such requests, the current direction is reversed, and the elevator begins to service (stop at) floors in the opposite direction.

We implement this algorithm as a monitor, which accepts the following calls from the elevator hardware:

- Pressing the call button at floor \( i \) or pressing one of the buttons labeled \( i \) inside the elevator both invoke the function \( \text{request}(i) \);
- Whenever the elevator door closes, the function \( \text{release()} \) is automatically invoked.

The monitor code is then as follows:

```c
monitor elevator {
    int direction=1, up=1, down=-1, position=1, busy=0;
    condition upsweep, downsweep;

    request(int dest) {
        if (busy) {
            if ((position<dest) ||
                ((position==dest) && (direction==up)))
                upsweep.wait(dest);
            else downsweep.wait(-dest);
        }
        busy = 1;
        position = dest;
    }

    release() {
        busy = 0;
        if (direction==up) {
            if (!empty(upsweep)) upsweep.signal;
            else {
                direction = down;
                downsweep.signal;
            }
        } else if (!empty(downsweep)) downsweep.signal;
        else {
            direction = up;
            upsweep.signal;
        }
    }
}
```
Chapter 3 Higher-Level Synchronization and Communication

The monitor uses the following variables:

- **position** records the current floor of the elevator (1 through \( n \));
- **direction** records the current direction in which the elevator is moving (up or down);
- **busy** records the current state of the elevator; \( \text{busy}=0 \) means the elevator is not moving, and there are no pending requests (no buttons have been pressed); \( \text{busy}=1 \) means the elevator is servicing a request (the door is open), or it is moving to a new floor.

Whenever any button is pressed and the elevator is not busy, the **busy** flag is set to 1 and **position** is set to the requested destination floor, \( \text{dest} \). The function then exits, which is interpreted by the elevator hardware as a permission to move to the new floor. If the elevator is busy when a button is pressed, the request is placed on one of two queues, **upsweep** or **downsweep**. The decision on which queue the request is placed depends on the current elevator position, the destination, and the direction of the current sweep. The following rules apply:

- When the elevator is below the requested destination, then, regardless of the current direction of motion, the request will be placed on the **upsweep** queue and served during an upsweep.
- Analogously, when the elevator is above the requested destination, the request is placed on the **downsweep** queue;
- When the elevator is at the requested destination, it should be serviced immediately, i.e., during the current stop. For example, when a person presses the button \( i \) when the elevator is on floor \( i \), the door should open immediately, rather than waiting for the elevator to return to the same floor during the next sweep. To guarantee this, the request must be placed on the queue currently being serviced—if the direction is up, the request is placed on the **upsweep** queue; if the direction is down, it is placed on the **downsweep** queue.

Within each queue, all requests must be ordered according to the direction of travel. This is accomplished by specifying a priority for each queued request as follows:

- On the **upsweep** queue, lower-numbered floors have precedence over higher-numbered ones because they are closer to the current position; thus, the priority is given by the destination, i.e., **downsweep.wait(dest)**;
- On the downsweep, higher-numbered floors have precedence over lower-numbered ones; thus, the priority is given by the inverse of the destination \( \text{dest} \), i.e., **downsweep.wait(-dest)**.

The function **release** signals one of the waiting requests (if any) to proceed. If the elevator is moving up and the **upsweep** queue is not empty, the process with the smallest priority on that queue is enabled. Similarly, on a downsweep, the smallest priority process from the **downsweep** queue is enabled. The direction of the sweep is reversed whenever the corresponding queue is empty. The Boolean function **empty**(\( c \)) is used to test whether the queue associated with the condition variable \( c \) is empty.
3.3.4 Event Ordering with Logical Clocks

A clock service is a fundamental component of any computer. One major difference between centralized and distributed systems is that there is no single global clock in the latter. Generally, each node in a distributed system has its own clock. These clocks may be synchronized so that their values are reasonably close to each other, but they will rarely have identical values because of various systematic errors that appear in all clock hardware, notably errors resulting from clock drift.

A particularly important application of clocks is the determination of the causality of events by time-stamping, i.e., labeling each event with its time of occurrence. An event could be, for example, the start of the execution of a program segment, allocation of a resource, sending or receiving a message, an interrupt, updating of a file, or the activation of a process. The problem is to label the events of interests so that for any two events $e_1$ and $e_2$, it is possible to tell whether $e_1$ precedes $e_2$, or vice versa, or neither.

Such a labeling is easy to do in a centralized system. We can simply attach the current clock value to the event of interest; then, if $C(e)$ is the time of event $e$, $C(e_1) < C(e_2)$ is true whenever $e_1$ precedes $e_2$. We wish to provide such a labeling to events in a distributed system. The following example shows that a straightforward use of the clocks at each node does not always produce a correct labeling.

**EXAMPLE: Labeling with Physical Clocks**

Suppose that a file user $U$ and a file server $FS$ each reside on a separate node. The user process has a clock $C_U$ and the server has a (different) clock $C_{FS}$. $U$ updates a file by sending changes $delta_1$ and $delta_2$ to $FS$ using an asynchronous send command, such as:

```
send(FS, delta1)
```

$FS$ employs a blocking receive, such as:

```
receive(U, changes)
```

These processes and a set of hypothetical clock values are illustrated in Figure 3-7.

---

3Note that the clock is assumed to be monotonically increasing; thus, it is never set back (resetting from daylight saving to standard time would destroy this monotonicity) and is never called twice in a row before it has a chance to tick at least once.
Difficulties occur because the clock values differ on the two nodes. In particular, assume that $C_U(\text{send}(FS, \delta_1)) = 10$ and $C_{FS}(\text{receive}(U, \delta_1)) = 5$. Thus, from the perspective of an external observer, the receive event $e_3$ seems to have happened before the send event $e_1$, which is not only counterintuitive, but can easily lead to errors. For example, suppose that the system was being debugged or it crashed, and a rerun or recreation was necessary. A common debugging or recovery technique assumes that a history or log of all significant events with their clock values is kept. Then, one essentially executes this log starting from a previous correct version of the system. Using the clock values for ordering events, our log would contain the event sequence $<e_3, e_1, e_2, e_4>$, instead of either $<e_1, e_3, e_2, e_4>$ or $<e_1, e_2, e_3, e_4>$, making it impossible or extremely difficult to execute correctly.

A very simple but clever solution was developed by Lamport (1978). Time-stamping is still employed, but not using the real clocks at each node. Instead, integer counters, called logical clocks, provide the event-ordering labels. Associated with each process $p$ is a logical clock $L_p$ that is incremented when events occur. More precisely, the clock is incremented between any two successive events $e_i$ and $e_{i+1}$ within a given process $p$, thus assuring that $L_p(e_i) < L_p(e_{i+1})$. Usually, the chosen increment value is one. Thus, incrementing the clock between the two events follows the rule:

$$L_p(e_{i+1}) = L_p(e_i) + 1$$

The tricky part is to handle events that are associated with the interaction of processes on different nodes. We assume that processes interact only through message-passing using the send and receive primitives, defined as in the example above. A send by a process $p$ to another process $q$ is treated as an event, say $e_s$, within $p$. Its timestamp is:

$$L_p(e_s) = L_p(e_i) + 1$$

where $L_p(e_i)$ is the time-stamp of the last event within $p$.

The corresponding receive is an event, say $e_r$, within the process $q$. To guarantee that this event has a greater time-stamp than $e_s$, process $p$ attaches its local time-stamp $L_p(e_s)$ to the message. The receiving process, instead of just incrementing the value of its last event, derives the time-stamp for the receive event using the following rule:

$$L_q(e_r) = \max(L_p(e_s), L_q(e_i)) + 1$$

where $L_q(e_i)$ is the time-stamp of the last event within $q$.

This scheme yields a partial ordering of events with respect to a relation called happened-before. This relation, denoted by “$\rightarrow$”, captures the flow of information through the system. In particular, given two events $e_i$ and $e_j$, the relation $e_i \rightarrow e_j$ holds if the following two conditions are satisfied:

1. $e_i$ and $e_j$ belong to the same process and $e_i$ occurred before $e_j$ (in real time);
2. $e_i$ is the event of sending a message and $e_j$ is the event of receiving the same message.
EXAMPLE: Using Logical Clocks

Figure 3-8 shows how the logical clocks change for three processes, \( p_1 \), \( p_2 \), and \( p_3 \), as events occur and messages are passed. Events are denoted by \( \times \)-marks on a vertical logical clock line. The arrows indicate message communications. The clocks of all three processes start at 0. After 3 internal events, \( p_1 \) sends a message to process \( p_2 \) with the time-stamp of \( L_{p_1}(u) = 4 \). \( p_2 \)'s clock is still at 1 (event \( j \)) when the message arrives. Thus \( p_2 \) advances its clock for the receive event to \( L_{p_2}(v) = \max(4, 1) + 1 = 5 \).

When \( p_2 \) sends a message to \( p_3 \) at time 6, \( p_3 \)'s clock is already at 12. Thus the new time-stamp assigned to the receive event in \( p_3 \) is \( L_{p_3}(x) = \max(6, 12) + 1 = 13 \).

FIGURE 3-8. Logical clocks in action.
Chapter 3  Higher-Level Synchronization and Communication

Time-stamps of several events could be identical in the scheme described so far. For example, in Figure 3-8, \( L_{p_1(i)} = L_{p_2(j)} = L_{p_3(k)} = 1 \). If nodes and/or processes have unique identifiers, we can obtain a unique global time-stamp for each event by just concatenating this identifier with the logical clock value; the causality ordering will not be changed. For example, assume the unique IDs of processes \( p_1, p_2, \) and \( p_3 \) are 1, 2, and 3, respectively. Then \( L_{p_1(i)}(i) = 1.1, L_{p_2(j)}(j) = 1.2, \) and \( L_{p_3(k)}(k) = 1.3 \) would yield three unique time-stamps.

There are many applications of Lamport’s logical clocks. The time-stamping of events allows the simulation or replay of a distributed set of interacting programs, as was illustrated by the file server example above. Some algorithms for solving the distributed mutual exclusion problem, for implementing distributed semaphores, for ordering messages broadcast (multicast) by different processes, and for detecting deadlocks in distributed systems also rely on logical clocks in a central way.

Yet another application area is transaction processing, an important part of database systems (e.g., [Bernstein and Lewis 1993]). A transaction is a unit of execution that accesses and possibly updates a database using a sequence of read, write, and computational operations. A standard example is a banking system transaction where a customer accesses and withdraws money from one account and transfers it to another; the customer and each of the accounts may reside at different computer sites. The entire sequence of operations starting from the access of the first account to the completion of the update of the second is defined as a single transaction. The banking system must service many of these transactions at the same time for reasons of cost and speed.

Generally, there is a possibility that the transaction may crash or deadlock before completion, leaving the database in an inconsistent state unless properly handled. Consequently, an “all-or-nothing” approach is required: either all the operations of a transaction are completed correctly or none of them may have any effect. This property of transactions is called atomicity. In addition, concurrent or interleaved execution of transactions must have the same effect as if all transactions had been executed sequentially one at a time. This requirement is referred to as serializability of transactions.

Note that the problem of concurrent or interleaved execution is similar to the CS problem discussed in Chapter 2; however, unlike CS, transactions must be allowed to proceed concurrently for reasons of speed and efficiency. The system must detect inconsistent interleavings of operations at runtime and abort, undo, and redo all affected transactions.

One set of techniques for providing atomicity and serializability in a distributed environment involves the use of unique time-stamps, such as those definable with logical clocks, for each transaction. Transactions may then be ordered by their relative age. This information may be used to decide which transaction should be aborted and redone, or which transactions depend causally on other transactions and result in additional abort, undo, and redo operations.

CONCEPTS, TERMS, AND ABBREVIATIONS

The following concepts have been introduced in this chapter. Test yourself by defining and discussing each keyword or phrase.
Section 3.3  Other Classic Synchronization Problems  101

<table>
<thead>
<tr>
<th>Alarm clock</th>
<th>Monitor, Hoare monitor,</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous communications</td>
<td>Java monitor, Mesa monitor</td>
</tr>
<tr>
<td>Broadcast</td>
<td>Notify</td>
</tr>
<tr>
<td>Condition variable</td>
<td>Priority wait</td>
</tr>
<tr>
<td>Communicating Sequential Processes</td>
<td>Protected type</td>
</tr>
<tr>
<td>programming language (CSP)</td>
<td>Readers/writers problem</td>
</tr>
<tr>
<td>Dining philosophers problem</td>
<td>Remote procedure call (RPC)</td>
</tr>
<tr>
<td>Distributed mutual exclusion</td>
<td>Rendezvous</td>
</tr>
<tr>
<td>Elevator algorithm</td>
<td>Send/Receive operations</td>
</tr>
<tr>
<td>Entry guard or barrier</td>
<td>Signal</td>
</tr>
<tr>
<td>Guarded command</td>
<td>Timeout</td>
</tr>
<tr>
<td>Interprocess communication (IPC)</td>
<td>Time-stamp</td>
</tr>
<tr>
<td>Logical clock</td>
<td>Token ring</td>
</tr>
<tr>
<td>Mailbox</td>
<td>Transaction</td>
</tr>
<tr>
<td>Message port</td>
<td></td>
</tr>
</tbody>
</table>

EXERCISES

1. Consider the following monitor pseudo code:

```plaintext
monitor m() {
    int x=10, y=2;
    condition c;

    A() {
        (1) x++;
        (2) c.signal;
        (3) y = x-2; }

    B() {
        (4) if (x>10)
        (5) x--
        (6) else {c.wait;
        (7)     x--;}
    }
}
```

Assume that after the monitor is initialized, functions A and B are called in the following sequence by various processes:

```plaintext
m.A(); m.A(); m.B(); m.B(); m.B(); m.B(); m.A(); m.A();
```

Using the line numbers in the code, trace the sequence of instruction execution. Show values of x and y at the end of each instruction.

2. Rewrite the code of Exercise 14, Chapter 2 (forall code with arrays A[] and B[]) using a monitor with priority waits to enforce the proper synchronization of the processes. Hint: Define a monitor with two functions. One is called to request permission to update element A[i]; the other is called when the update is done.
3. Write a monitor that implements a bounded stack, \texttt{b\_stack}, i.e., a stack of at most \texttt{max} elements. The push and pop operations behave as follows:

- \texttt{b\_stack.push(x)}: if the stack is not completely full, the operation pushes the new element \(x\) on top of the stack; otherwise, the operation has no effect and the value of \(x\) is discarded.
- \texttt{b\_stack.pop(x)}: if the stack is not completely empty, the operation removes the value currently on top of the stack and returns it in the variable \(x\); if the stack is empty, the operation waits until a value has been pushed on the stack and then completes by popping and returning this value.

4. Let a \texttt{bounded semaphore} \(s\) be a general semaphore that cannot exceed a given value \(s_{max} > 0\). The corresponding operations \texttt{PB} and \texttt{VB} are defined as:

- \texttt{PB(s)}: wait until \(s > 0\); then decrement \(s\) by 1;
- \texttt{VB(s)}: wait until \(s < s_{max}\); then increment \(s\) by 1.

Write a monitor, named \(s\), such that the calls \texttt{s.PB()} and \texttt{s.VB()} emulate the operations on bounded semaphores.

5. Repeat the previous exercise using protected types.

6. Modify and augment the simple batch OS (Exercise 13, Chapter 2) using each of the synchronization mechanisms below:

(a) Monitors
(b) Protected types
(c) Send/Receive primitives
(d) Rendezvous with selective accept statements

7. There are two processes, \(c_1\) and \(c_2\), which at unpredictable times call another process \(p\). The task of \(p\) is to count how many times it has been called by each \(c_i\) process. Once in a while, a fourth process \(q\) calls \(p\) to get the accumulated counts for the \(c_i\) processes. When the counts are reported to \(q\), \(p\) resets both counters to 0. Write the code for process \(p\) using rendezvous with selective accept statements.

8. Consider the following rendezvous code:

```c
while (1) {
    select {
    when a==TRUE :
        accept A() { f1; b=FALSE }
    when b==TRUE :
        accept B() { f2; a=FALSE }
    else { a=TRUE; b=TRUE }
    }
}
```

Assume that there are no outstanding calls to \(A\) or \(B\) when the select statement is executed for the first time. Thereafter, the following calls arrive in the given order:

\[ A(), B(), B(), A(), A(), B() \]

(a) In which order will the calls be accepted (processed)?
(b) Can caller of \(A\) (or \(B\)) starve?

9. Two concurrent processes, \(p\) and \(q\), interact as follows:

- \(p\) and \(q\) start concurrently;
- while \(p\) computes \("x1 = \text{some\_computation}"\) \(q\) computes \("y1 = \text{some\_computation}"\);
- a rendezvous takes place during which \(p\) computes \("y2 = y1 + x1"\);
Section 3.3 Other Classic Synchronization Problems

- While \( p \) computes "\( x_2 = \) some\_computation," \( q \) computes "\( y_3 = y_2 + \) some\_computation";
- A second rendezvous takes place during which \( p \) computes "\( y_4 = y_3 + x_2 \)."

Write the pseudo code for the two processes \( p \) and \( q \) using rendezvous.

10. Consider the solution to the readers/writers problem (Section 3.1). Assume the first request is a writer \( W_1 \). While \( W_1 \) is writing, the following requests arrive in the given order:

\[ W_2, R_1, R_2, W_3, R_3, W_4 \]

(a) In which order will these requests be processed? Which groups of readers will be reading concurrently?

(b) Assume that two new readers \( R_4 \) and \( R_5 \) arrive while \( W_2 \) is writing. For each, indicate when it will be processed (i.e., following which write request)?

11. Consider the solution to the readers/writers problem (Section 3.1). Assume that there is only a single-writer processes. That is, the function `start_write` will never be invoked until the preceding `end_write` has terminated. Simplify the code accordingly.

12. The following solution to the readers/writers problem was proposed by Courtois, Heymans, and Parnas (1971):

\begin{verbatim}
reader() {
    P(mutex); P(w);
    read_cnt++; if (read_cnt==1) P(w); V(mutex);
    V(mutex);
}

writer() {
    P(w); WRITE;
    if(read_cnt==1) P(w); V(w)
    V(mutex);
    READ;
    P(mutex);
    read_cnt--;
    if(read_cnt==0) V(w);
    V(mutex);
}
\end{verbatim}

Initially: mutex=1; read_cnt=0; w=1;

Does this solution satisfy the basic requirements of the readers/writers problem? Is starvation of readers or writers possible?

13. Consider the analogy of a tunnel with only a single lane. To avoid a deadlock, cars must be prevented from entering the tunnel at both ends simultaneously. Once a car enters, other cars from the same direction may follow immediately. Ignoring the problem of starvation, write the code using semaphores to solve this problem. (Hint: Consider the readers/writers code given in the previous exercise. The tunnel problem is a variation of the readers/writers problem where multiple readers or multiple writers are allowed to enter the critical region.)

14. Write a solution for the tunnel problem of the previous exercise, but this time guarantee that cars from neither direction will wait indefinitely (no starvation). (Hint: Write a monitor similar to the readers/writers problem.)

15. Assume that each of the five philosophers, \( i \), in the dining philosophers problem execute the following segment of code:

\begin{verbatim}
P(mutex);
P(fork[i]);
P(fork[i+1 % 5]);
V(mutex);
\end{verbatim}
Chapter 3 Higher-Level Synchronization and Communication

```
eat;
V(fork[i]);
V(fork[i+1 % 5])
```

(a) Does this code satisfy all requirements of the dining philosophers problem?
(b) Would the solution improve or get worse if the V(mutex) statement was moved:
   • after the second V() operation;
   • between the two P() operations.

16. Consider the Elevator algorithm in Section 3.3. Assume we replace the line:
```
if ((position<dest) || ((position==dest) && (direction==up))
```

with either of the following simpler conditions:
(a) if (position<dest)
(b) if (position<=dest)

Will the algorithm still work? In what way will its behavior change?

17. Extend the elevator algorithm in Section 3.3 to have two separate call buttons—UP and DOWN—on each floor. The elevator should stop at the floor only if it is traveling in the requested direction.

18. Consider the three processes in Figure 3-8.
(a) For each of the following pairs of logical clocks (L_i, L_j) indicate whether:
   • (L_i \rightarrow L_j) (L_i happened before L_j);
   • (L_i \leftarrow L_j) (L_i happened after L_j);
   • neither (L_i is concurrent with L_j).

   The subscripts indicate which process the clock value belongs to:
   (1_a, 1_c), (1_a, 10_c), (1_a, 14_c), (5_a, 1_c), (5_a, 14_c), (1_a, 7_b), (16_b, 10_c).

(e) Assume another message was sent from process c to process a. The message was sent at time 10 according to c’s clock and received at time 2 according to a’s clock. Update the values of all the logical clocks affected by the new message in the figure.