CHAPTER 2

SYNCHRONIZATION MECHANISMS

2.1 INTRODUCTION

Processes that interact with each other often need to be synchronized. The synchronization of a process is normally achieved by regulating the flow of its execution. In this chapter, various mechanisms for process synchronization are presented. First, the notion of a process is presented, then the issue of synchronizing processes and mechanisms for synchronization are introduced.

2.2 CONCEPT OF A PROCESS

The notion of a *process* is fundamental to operating systems. Although there are many accepted definitions of a process, here we define the concept of process in the context of this book. A process is a program whose execution has started but is not yet complete (i.e., a program in execution). A process can be in any of the following three basic states:

Running. The processor is executing the instructions of the corresponding process.

Ready. The process is ready to be executed, but the processor is not available for the execution of this process.

Blocked. The process is waiting for an event to occur. Examples of events are an I/O operation waiting to be completed, memory to be made available, a message to be received, etc.

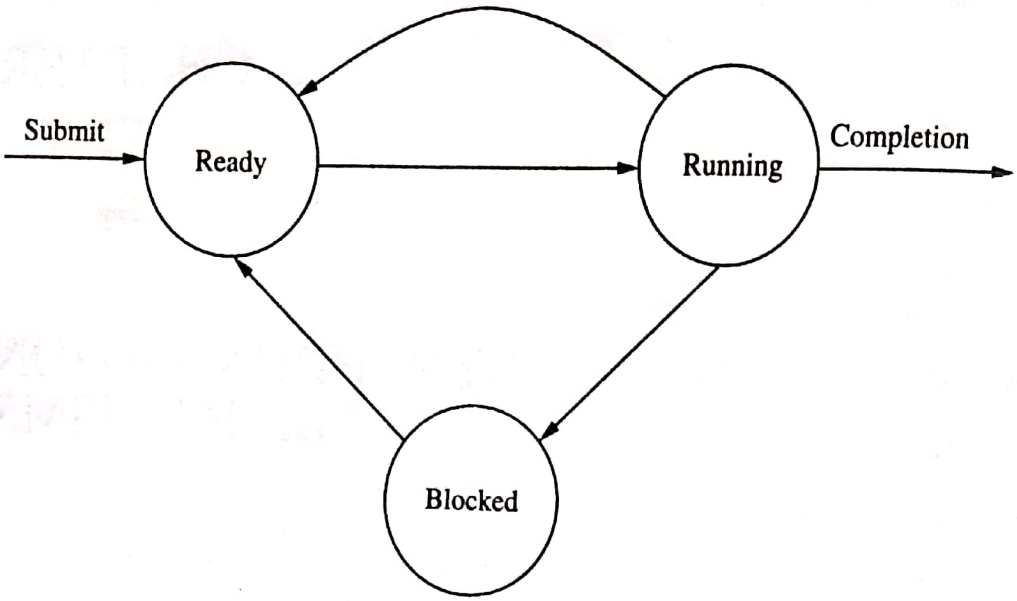

FIGURE 2.1
State transition diagram of a process.

Figure 2.1 depicts transitions among these states during the life cycle of a process. A running process gets blocked because a requested resource is not available or can become ready because the CPU decided to execute another process. A blocked process becomes ready when the needed resource becomes available to it. A ready process starts running when the CPU becomes available to it.

A data structure commonly referred to as the *Process Control Block* (PCB), stores complete information about a process, such as id, process state, priority, privileges, virtual memory address translation maps, etc. The operating system as well as other processes can perform operations on a process. Examples of such operations are create, kill, signal, suspend, schedule, change-priority, resume, etc. A detailed treatment of these topics is beyond the scope of this book and can be found in [22].

2.3 CONCURRENT PROCESSES

Two processes are concurrent if their execution can overlap in time; that is, the execution of the second process starts before the first process completes. In multiprocessor systems, since CPUs can simultaneously execute different processes, the concept of concurrency is concrete and easy to visualize. In a single CPU system, physical concurrency can be due to concurrent execution of the CPU and an I/O. If a CPU interleaves the execution of several processes, logical concurrency is obtained (as opposed to the physical concurrency of a multiprocessor system).

Two processes are serial if the execution of one must be complete before the execution of the other can start. Normally, two processes are said to be concurrent if they are not serial. Concurrent processes generally interact through either of the following mechanisms:

Shared variables. The processes access (read or write) a common variable or common data.

Message passing. The processes exchange information with each other by sending and receiving messages.

If two processes do not interact, then their execution is transparent to each other (i.e., their concurrent execution is the same as their serial execution).

2.3.1 Threads

Traditionally, a process has a single address space and a single thread of control with which to execute a program within that address space. To execute a program, a process has to initialize and maintain state information. The state information typically is comprised of page tables, swap images, file descriptors, outstanding I/O requests, saved register values, etc. This information is maintained on a per program basis, and thus, a per process basis. The volume of this state information makes it expensive to create and maintain processes as well as to switch between them.

To handle situations where creating, maintaining, and switching between processes occur frequently (e.g., parallel applications), *threads* or *lightweight processes* have been proposed.

Threads separate the notion of execution from the rest of the definition of the process [1]. A single thread executes a portion of a program, cooperating with other threads concurrently executing within the same address space. Each thread makes use of a separate program counter and a stack of activation records (that describe the state of the execution), and a control block. The control block contains the state information necessary for thread management, such as for putting a thread into a ready list and for synchronizing with other threads. Most of the information that is part of a process is common to all the threads executing within a single address space, and hence maintenance is common to all the threads. By sharing common information, the overhead incurred in creating and maintaining information, and the amount of information that needs to be saved when switching between threads of the same program, is reduced significantly. Threads are treated in more detail in Sec. 17.4.

2.4 THE CRITICAL SECTION PROBLEM

When concurrent processes (or threads) interact through a shared variable, the integrity of the variable may be violated if access to the variable is not coordinated. Examples of integrity violations are (1) the variable does not record all changes, (2) a process may read inconsistent values, and (3) the final value of the variable may be inconsistent.

A solution to this problem requires that processes be synchronized such that only one process can access the variable at any one time. This is why this problem is widely referred to as the problem of *mutual exclusion*. A *critical section* is a code segment in a process in which a shared resource is accessed. A solution to the problem of mutual exclusion must satisfy the following requirements:

- Only one process can execute its critical section at any one time.
- When no process is executing in its critical section, any process that requests entry to its critical section must be permitted to enter without delay.

- When two or more processes compete to enter their respective critical sections, the selection cannot be postponed indefinitely.
- No process can prevent any other process from entering its critical section indefinitely; that is, every process should be given a fair chance to access the shared resource.

2.4.1 Early Mechanisms for Mutual Exclusion

Various versions of *busy waiting* [3, 6, 13, 14, 19] were some of the first mechanisms to achieve mutual exclusion. In this mechanism, a process that cannot enter its critical section continuously tests the value of a status variable to find if the shared resource is free. The status variable records the status of the shared resource. The main problems with this approach are the wastage of CPU cycles and the memory access bandwidth.

Disabling interrupts, another mechanism that achieves mutual exclusion, is a mechanism where a process disables interrupts before entering the critical section and enables the interrupts immediately after exiting the critical section. Mutual exclusion is achieved because a process is not interrupted during the execution of its critical section and thus excludes all other processes from entering their critical section. The problems with this method are that it is applicable to only uniprocessor systems and important input-output events may be mishandled.

In multiprocessor systems, a special instruction called the *test-and-set instruction* is used to achieve mutual exclusion. This instruction (typically completed in one clock cycle) performs a single indivisible operation on a designated/specific memory location. When this instruction is executed, a specified memory location is checked for a particular value; if they match, the memory location's contents are altered. This instruction can be used as a building block for busy waiting or can be incorporated into schemes that relinquish the CPU when the instruction fails (i.e., a match is not found).

2.4.2 Semaphores

A semaphore is a high-level construct used to synchronize concurrent processes. A semaphore S is an integer variable on which processes can perform two indivisible operations, $P(S)$ and $V(S)$ [3]. Each semaphore has a queue associated with it, where processes that are blocked on that semaphore wait. The P and V operations are defined as follows:

$P(S)$: if $S \geq 1$ then $S := S - 1$
else block the process on the semaphore queue;

$V(S)$: if some processes are blocked on the semaphore S
then unblock a process
else $S := S + 1$;

When a $V(S)$ operation is performed, a blocked process is picked up for execution. The queueing discipline of a semaphore queue depends upon the implementation.

Depending upon the values a semaphore is allowed to take, there are two types of semaphores: a binary semaphore (the initial value is 1) and a resource counting semaphore (the initial value is normally more than 1). A semaphore is initialized by

```
Shared var
mutex: semaphore (= 1);

Process i (i = 1, n);

    begin
    ⋮
    P(mutex);
    execute CS;
    V(mutex);
    ⋮
    end.
```

FIGURE 2.2
Solution to mutual exclusion using a semaphore.

the system. Note that for any semaphore,

$$\text{number of } P \text{ operations} - \text{number of } V \text{ operations} \leq \text{initial value.}$$

Binary semaphores are used to create mutual exclusion, because at any given time only one process can get past the P operation. Resource counting semaphores are primarily used to synchronize access to a shared resource by several concurrent processes. (To control how many processes can concurrently perform an operation.)

Example 2.1. Figure 2.2 shows how we can use a binary semaphore to achieve mutual exclusion. If any process has performed a $P(mutex)$ operation without performing the corresponding $V(mutex)$ operation (i.e., the process is still inside its CS), then all other processes trying to enter the CS will wait on the $P(mutex)$ operation until this process performs the $V(mutex)$ operation (i.e., exits the CS). Therefore, mutual exclusion is achieved.

Although semaphores provide a simple and sufficiently general scheme for all kinds of synchronization problems, they suffer from the following drawbacks:

- A process that uses a semaphore has to know which other processes use the semaphore. It may also have to know how those processes are using the semaphore. This knowledge is required because the code of a process cannot be written in isolation, as the semaphore operations of all the interacting processes have to be coordinated.
- Semaphore operations must be carefully installed in a process. The omission of a P or V operation may result in inconsistencies (i.e., a violation of the integrity of a shared resource) or deadlocks.
- Programs using semaphores can be extremely hard to verify for correctness.

2.5 OTHER SYNCHRONIZATION PROBLEMS

In addition to mutual exclusion, there are many other situations where process synchro-

tion problems. In these problems, the control of concurrent access to shared resources is essential.

2.5.1 The Dining Philosophers Problem

The dining philosophers problem is a classic synchronization problem that has formed the basis for a large class of synchronization problems. In one version of this problem, five philosophers are sitting in a circle, attempting to eat spaghetti with the help of forks. Each philosopher has a bowl of spaghetti but there are only five forks (with one fork placed to the left, and one to the right of each philosopher) to share among them. This creates a dilemma, as both forks (to the left and right) are needed by each philosopher to consume the spaghetti.

A philosopher alternates between two phases: thinking and eating. In the *thinking* mode, a philosopher does not hold a fork. However, when hungry (after staying in the thinking mode for a finite time), a philosopher attempts to pick up both forks on the left and right sides. (At any given moment, only one philosopher can hold a given fork, and a philosopher cannot pick up two forks simultaneously). A philosopher can start eating only after obtaining both forks. Once a philosopher starts eating, the forks are not relinquished until the eating phase is over. When the eating phase concludes (which lasts for finite time), both forks are put back in their original position and the philosopher reenters the thinking phase.

Note that no two neighboring philosophers can eat simultaneously. In any solution to this problem, the act of picking up a fork by a philosopher must be a critical section. Devising a deadlock-free solution to this problem, in which no philosopher starves, is nontrivial.

2.5.2 The Producer-Consumer Problem

In the producer-consumer problem, a set of *producer* processes supplies messages to a set of *consumer* processes. These processes share a common buffer pool where messages are deposited by producers and removed by consumers. All the processes are asynchronous in the sense that producers and consumers may attempt to deposit and remove messages, respectively, at any instant. Since producer processes may outpace consumer processes (or vice versa), two constraints need to be satisfied; no consumer process can remove a message when the buffer pool is empty and no producer process can deposit a message when the buffer pool is full.

Integrity problems may arise if multiple consumers (or multiple producers) try to remove messages (or try to put messages) in the buffer pool simultaneously. For examples, associated data structures (e.g., pointers to buffers) may not be updated consistently, or two producers may try to put messages in the same buffer. Therefore, access to the buffer pool and the associated data structures must constitute a critical section in these processes.

2.5.3 The Readers-Writers Problem

In the readers-writers problem, the shared resource is a file that is accessed by both the reader and writer processes. Reader processes simply read the information in the

file without changing its contents. Writer processes may change the information in the file. The basic synchronization constraint is that any number of readers should be able to concurrently access the file, but only one writer can access the file at a given time. Moreover, readers and writers must always exclude each other.

There are several versions of this problem depending upon whether readers or writers are given priority.

Reader's Priority. In the reader's priority case, arriving readers receive priority over waiting writers. A waiting or an arriving writer gains access to the file only when there are no readers in the system. When a writer is done with the file, all the waiting readers have priority over the waiting writers.

Writer's Priority. In the writer's priority case, an arriving writer receives priority over waiting readers. A waiting or an arriving reader gains access to the file only when there are no writers in the system. When a reader is done with the file, waiting writers have priority over waiting readers to access the file.

In the reader's priority case, writers may *starve* (i.e., writers may wait indefinitely) and vice-versa. To overcome this problem, a *weak reader's priority* case or a *weak writer's priority* case can be used. In a weak reader's priority case, an arriving reader still has priority over waiting writers. However, when a writer departs, both waiting readers and waiting writers have equal priority (that is, a waiting reader or a waiting writer is chosen randomly).

2.5.4 Semaphore Solution to Readers-Writers Problem

Example 2.2. Figure 2.3 shows how we can implement a readers priority solution to the readers-writers problem using semaphores. A reader calls the *reader* procedure and a writer calls the *writer* procedure to read and write the file, respectively. If a reader is reading, then in the *reader* procedure nreaders $\neq$ 0 and consequently, all arriving readers get to immediately read the file. If there are some writers waiting to write, one of them is blocked on the semaphore 'wmutex' and the rest of them are blocked on the semaphore 'srmutex' in the writer procedure. When the last reader finishes reading the file, it unblocks a writer waiting on the 'wmutex' semaphore. If some readers arrive while this writer is writing, the first reader will be blocked on the semaphore 'wmutex' and all subsequent readers will be blocked on the semaphore 'mutex'. When the writer departs, its V(wmutex) operation will unblock a waiting reader (if there is one) which subsequently causes all the waiting readers to unblock, one by one. The departing writer performs a V(srmutex) operation that unblocks a writer (if there is one) waiting on P(srmutex). Clearly, this writer will be blocked at P(wmutex) if nreaders is greater than 0 at this time.

2.6 LANGUAGE MECHANISMS FOR SYNCHRONIZATION

The early synchronization mechanisms described previously (Sec. 2.4.1) and semaphores (Sec. 2.4.2) are too primitive to build large, complex, and reliable systems. The need for reliable and easily maintainable software is even greater when concurrency is

```
shared var
nreaders : integer;
mutex, wmutex, srmutex : semaphore;
procedure reader;

    begin
    P(mutex);
    if nreaders = 0
    then nreaders:=  nreaders + 1; P(wmutex)
    else
            nreaders:=  nreaders + 1;
    V(mutex);
    read(f);
    P(mutex);
    nreaders :=  nreaders − 1;
    if nreaders = 0 then V(wmutex);
    V(mutex);
    end.

procedure writer(d: data);

    begin
    P(srmutex);
    P(wmutex);
    write(f, d);
    V(wmutex);
    V(srmutex);
    end.

begin (* initialization *)
mutex = wmutex = srmutex = 1;
nreaders :=  0;
end.
```

FIGURE 2.3
A semaphore solution to the reader's priority problem.

involved. Parallel programs are more complex than sequential ones because processes interact more often, and time-dependent errors, which are not susceptible to traditional debugging techniques, are much more likely to occur. It is therefore imperative that higher level concepts are integrated into programming languages to ensure that correctness is supported and that underlying hardware implementation is of no concern to the programmer. In the following sections, several high-level mechanisms are discussed.

2.6.1 Monitors

Monitors are abstract data types for defining shared objects (or resources) and for scheduling access to these objects in a multiprogramming environment [9]. A monitor

consists of procedures, the shared object (resource), and administrative data. Procedures are the gateway to the shared resource and are called by the processes needing to access the resource. Procedures can also be viewed as a set of operations that can be performed on the resource. The structure of a monitor is illustrated in Fig. 2.4. The execution of a monitor obeys the following constraints:

- Only one process can be active (i.e., executing a procedure) within the monitor at a time. Usually, an implicit process associated with the monitor ensures this. When a process is active within the monitor, processes trying to enter the monitor are placed in the monitor's entry queue (common to the entire monitor). Thus, a monitor, by encapsulating the shared resource, easily guarantees mutual exclusion.
- Procedures of a monitor can only access data local to the monitor; they cannot access an outside variable.
- The variables or data local to a monitor cannot be directly accessed from outside the monitor.

Since the main function of a monitor is to control access to a shared resource, it should be able to delay and resume the execution of the processes calling monitor's procedures. The synchronization of processes is accomplished via two special operations namely, *wait* and *signal*, which are executed within the monitor's procedures. Executing a wait operation suspends the caller process and the caller process thus relinquishes control of the monitor. Executing a signal operation causes exactly one waiting process to immediately regain control of the monitor. The signaling process is suspended on an *urgent queue*. The processes in the urgent queue have a higher priority for regaining control of the monitor than the processes trying to enter the monitor when a process relinquishes it. (Note that at any instant, two types of processes may be trying to gain

```
< Monitor-name>:monitor begin
Declaration of data local to the monitor.
⋮

    procedure < Name>  (< formal parameters>);

        begin
        procedure body
        end;

    Declaration of other procedures
    ⋮

    begin
    Initialization of local data of the monitor
    end;
end;
```

FIGURE 2.4
The structure of a monitor.

the control of the monitor: the processes waiting in the monitor's entry queue to enter the monitor for the first time, and the processes waiting on the urgent queue.) When a waiting process is signaled, it starts execution from the very next statement following the wait statement. If there are no waiting processes, the signal has no effect.

If there are a number of different reasons for the blocking or unblocking of processes, a *condition variable* associated with wait and signal operations helps to distinguish the processes to be blocked or unblocked for different reasons. The condition variable is not a data type in the conventional sense, rather, it is associated with a queue (initially empty) of processes that are currently waiting on that condition. The operation < *condition variable* >.**queue** returns *true* if the queue associated with the condition variable is not empty. Otherwise it returns *false*. The syntax of wait and signal operations associated with a condition is:

<condition variable>.**wait**;
<condition variable>.**signal**;

One major advantage of monitors is the flexibility they allow in scheduling the processes waiting in queues. First-in-first-out discipline is generally used with queues, but priority queues can be implemented by enhancing the wait operation with a parameter. The parameter specifies the priority of the process to be delayed; the smaller the value of the parameter, the higher its priority. When a queue is signaled, the process with the highest priority in that queue is activated. The syntax for the priority wait is:

< condition variable >.**wait** (< parameter >)

Example 2.3. Figure 2.5 gives a solution to the reader's priority problem (see Sec. 2.5.3) using monitors [9]. For proper synchronization, reader processes must call the *startread* procedure before accessing the file (shared resource) and call the *endread* when the read is finished. Likewise, writer processes must call *startwrite* before modifying the file and call *endwrite* when the write is finished. The monitor uses the boolean variable *busy* to indicate whether a writer is active (i.e., accessing the file) and *readercount* to keep track of the number of active readers.

On invoking startread, a reader process is blocked and placed in the queue of the *OKtoread* condition variable if busy is true (i.e., if there is an active writer); otherwise, the reader proceeds and performs the following. The process increments the readercount, and activates a waiting reader, if present, through the OKtoread.**signal** operation. On the completion of access, a reader invokes endread, where readercount is decremented. When there are no active readers (i.e., readercount = 0), the last exiting reader process performs the OKtowrite.**signal** operation to activate any waiting writer.

A writer, on invoking startwrite, proceeds only when no other writer or readers are active. The writer process sets busy to true to indicate that a writer is active. On completion of the access, a writer invokes the endwrite procedure. The endwrite procedure sets busy to false, indicating that no writer is active, and checks the OKtoread queue for the presence of waiting readers. If there is a waiting reader, the exiting writer signals it, otherwise it signals the writer queue. If a reader is activated in endwrite procedure, it increments the readercount and executes the OKtoread.**signal**, thereby activating the next waiting reader in the queue. This process continues until all the waiting readers have been activated, during which processes trying to enter the

```
readers-writers : monitor;
begin
readercount : integer;
busy : boolean;
OKtoread, OKtowrite : condition;

procedure startread;

    begin
    if busy then OKtoread.wait;
    readercount := readercount + 1;
    OKtoread.signal;
    (* Once one reader can start, they all can *)
    end startread;
procedure endread;

    begin
    readercount := readercount - 1;
    if readercount = 0 then OKtowrite.signal;
    end endread;

procedure startwrite;

    begin
    if busy OR readercount ≠ 0 then OKtowrite.wait;
    busy := true;
    end startwrite;

procedure endwrite;

    begin
    busy := false;
    if OKtoread.queue then OKtoread.signal
            else OKtowrite.signal;
    end endwrite;

begin (* initialization *)
readercount := 0;
busy := false;
end;

end readers-writers;
```

Reader priority

FIGURE 2.5
A monitor solution for the reader's-priority problem.

monitor are blocked and join the monitor's entry queue. But, after all the readers waiting on the OKtoread condition have been signaled, any newly arrived readers will gain access to the monitor before any waiting writers. In summary, the reader's priority monitor, while not permitting a new writer to start when there is a reader waiting, permits any number of readers to proceed, as long as there is at least one active reader.

Example 2.4. Figure 2.6 illustrates the use of priority wait. Consider a problem where multiple users share a printer. The monitor *smallest-job-first* of Fig. 2.6 synchronizes print requests that may arrive concurrently. To illustrate priority wait, the monitor prints the smallest file first. The printer process has the following execution sequence:

```
loop

    smallest-job-first.start-print-job;
    print the file;
    smallest-job-first.end-print-job;

endloop
```

When a user submits a print command, the procedure *queue-print-job* is invoked. If the printer is busy (i.e., *printer-busy* = true), the print request process is blocked and is queued on the OKtoprint condition. Otherwise, the request proceeds and sets printer-busy to true, spools the file into a buffer, and activates the printer process by executing the next-job-avail.**signal** operation. Note that the priority of a blocked request is dictated by the size of the file it is trying to print. When the printer finishes printing, it invokes the *end-print-job* procedure. This procedure sets printer-busy to false (to indicate that the printer is free) and activates a waiting print request through the OKtoprint.**signal** operation. The signal operation wakes up the process with the highest priority, i.e., the process requesting to print the smallest file among all waiting processes. This continues until there are no more print requests waiting, at which point the printer process is blocked in the *start-print-job* procedure on the *next-job-avail* condition.

DRAWBACKS OF MONITORS. A major weakness of monitors is the absence of concurrency if a monitor encapsulates the resource, since only one process can be active within a monitor at a time. In the example of Fig. 2.5, to allow concurrent access for readers, the resource (a file) is separated from the monitor (it is not local to the monitor). For proper synchronization, procedures of the monitor must be invoked before and after accessing the shared resource. This arrangement, however, allows the possibility of processes improperly accessing the resources without first invoking the monitor's procedures. Further, there is the possibility of deadlocks in the case of nested monitor calls. For example, consider a process calling a monitor procedure that in turn calls another lower level monitor procedure. If a wait is executed in the lower level monitor, the control of the lower level monitor is relinquished, but not the control of the higher level monitor. If the processes entering the higher level monitor can only cause signaling to occur in the lower level monitor, then deadlock occurs.

```
smallest-job-first : monitor;
begin
printer-busy : boolean;
buffer : data;
OKtoprint, next-job-avail : condition;

function filesize (file) : integer;

    begin
    This function returns filesize of the file.
    end;

procedure queue-print-job (file : data);

    begin
    if printer-busy then OKtoprint.wait (filesize(file));
    printer-busy := true;
    buffer := file; (* spooling of user's file *)
    next-job-avail.signal;
    end;

procedure start-print-job (var file : data);

    begin
    if NOT printer-busy then next-job-avail.wait;
    file := buffer; (* copy file into printer's buffer *)
    end;

procedure end-print-job;

    begin
    printer-busy := false;
    OKtoprint.signal;
    end;

begin
printer-busy := false;
end;

end smallest-job-first.
```

FIGURE 2.6
A monitor solution for printing the smallest job first.