CHAPTER 1

OVERVIEW

1.1 INTRODUCTION

The use of a bare hardware machine is cumbersome and inefficient because a large number of chores must be manually performed, such as entering programs and data at appropriate locations in the main memory, addressing and activating appropriate input-output devices, etc. When a machine is used by several users simultaneously, numerous other issues arise, such as the protection of user data and the time- and space-multiplexing of shared resources among them. An operating system relieves users of these cumbersome chores and increases efficiency by managing the system's resources.

1.2 FUNCTIONS OF AN OPERATING SYSTEM

An operating system is a layer of software on a bare hardware machine that performs two basic functions:

Resource management. A user program accesses several hardware and software resources during its execution. Examples of resources are the CPU, main memory, input-output devices, and various types of software (compiler, linker-loader, files, etc.). It is the operating system that manages the resources and allocates them to users in an efficient and fair manner. Resource management encompasses the following functions:

- Time management (CPU and disk scheduling).
- Space management (main and secondary storages).

- Process synchronization and deadlock handling.
- Accounting and status information.

User friendliness. An operating system hides the unpleasant, low-level details and idiosyncrasies of a bare hardware machine and provides users with a much friendlier interface to the machine. To load, manipulate, print, and execute programs, high-level commands can be used without the inconvenience of worrying about low-level details. The layer of operating system transforms a bare hardware machine into a virtual or abstract machine with added functionality (such as automatic resource management). Moreover, users of the virtual machine have the illusion that each one of them is the only user of the machine, even though the machine may be operating in a multiuser environment.

User friendliness issues encompass the following tasks:

- Execution environment (process management—creation, control, and termination—file manipulation, interrupt handling, support for I/O operations, language support).
- Error detection and handling.
- Protection and security.
- Fault tolerance and failure recovery.

1.3 DESIGN APPROACHES

An operating system could be designed as a huge, jumbled collection of processes without any structure. Any process could call any other process to request a service from it. The execution of a user command would usually involve the activation of a series of processes. While an implementation of this kind could be acceptable for small operating systems, it would not be suitable for large operating systems as the lack of a proper structure would make it extremely hard to specify, code, test, and debug a large operating system.

The design of general purpose operating systems has matured over the last two and a half decades and today's operating systems are generally enormous and complex. A typical operating system that supports a multiprogramming environment can easily be tens of megabytes in length and its design, implementation, and testing amounts to the undertaking of a huge software project. In this section, we discuss design approaches intended to handle the complexities of today's large operating systems. However, before we discuss these approaches, we first need to make the distinction between *what* should be done and *how* it should be done, in the context of operating system design.

Separation of Policies and Mechanisms

Policies refer to *what* should be done and mechanisms refer to *how* it should be done. For example, in CPU scheduling, mechanisms provide the means to implement various scheduling disciplines, and policy decides which CPU scheduling discipline (such as FCFS, SJTF, priority, etc.) will be used [14, 24] .

A good operating system design must separate policies from mechanisms. Since policies make use of underlying mechanisms, the separation of policies from mechanisms greatly contributes to flexibility, as policy decisions can be made at a higher level. Note that policies are likely to change with time, application, and users. If mechanisms are separated from policies, then a change in policies will not require changes in the mechanisms, and vice-versa. Otherwise, a change in policies may require a complete redesign.

1.3.1 Layered Approach

Dijkstra advocated the layered approach to lessen the design and implementation complexities of an operating system. The layered approach divides the operating system into several layers. The functions of an operating system are then vertically apportioned into these layers. Each layer has well-defined functionality and input-output interfaces with the two adjacent layers. Typically, the bottom layer interfaces with machine hardware and the top layer interfaces with users (or operators). The idea behind the layered approach is the same as in the seven-layer architecture of the Open System Interconnection (OSI) model of the International Standards Organization (ISO).

The layered approach has all the advantages of modular design. (In modular design, the system is divided into several modules and each module is designed independently.) Thus, each layer can be designed, coded, and tested independently. Consequently, the layered approach considerably simplifies the design, specification, and implementation (the coding and testing) of an operating system. However, a drawback of the layered approach is that operating system functions must be carefully assigned to various layers because a layer can make use only of the functionality provided by the layers beneath it.

A classic example of the layered approach is the THE operating system [8], which consists of six layers. Figure 1.1 shows these layers with their associated functions. Another classic example of this approach is the MULTICS system [19], which is structured as several concentric layers (rings). This ring structure in MULTICS not only simplifies design and verification, but it also serves as an aid in designing and implementing protection. In MULTICS, privilege decreases from the inner ring to the successive outer rings. The ring structure nicely defines and implements the protection in MULTICS.

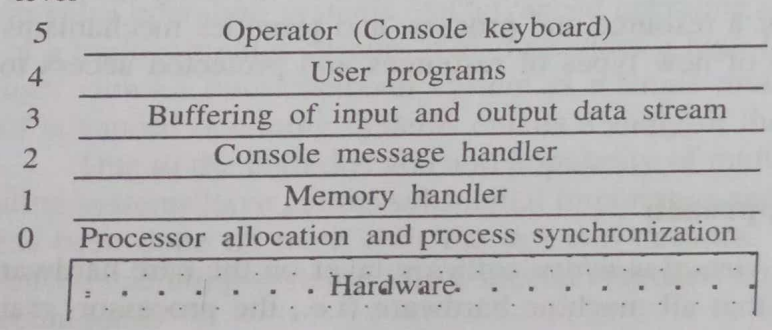

FIGURE 1.1
Structure of the THE operating system.

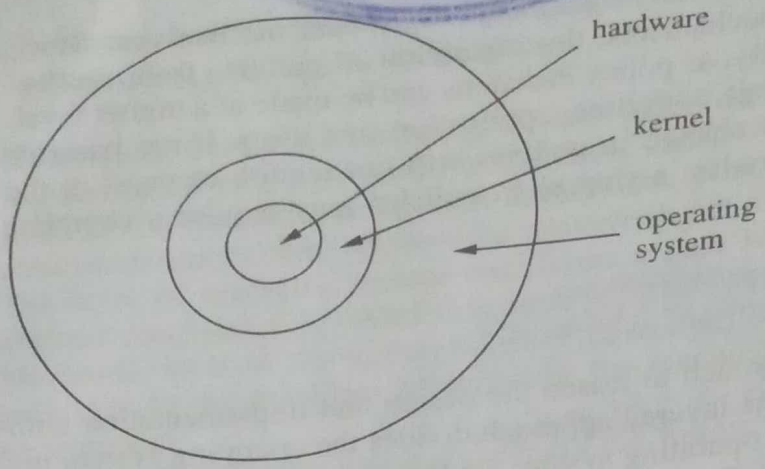

FIGURE 1.2
Structure of a kernel-based operating system.

1.3.2 The Kernel Based Approach

The kernel-based design and structure of operating systems was suggested by Brinch Hansen [12]. The *kernel* (more appropriately called the *nucleus*) is a collection of primitive facilities over which the rest of the operating system is built, using the functions provided by the kernel (see Fig. 1.2). Thus, a kernel provides an environment to build operating systems in which the designer has considerable flexibility because policy and optimization decisions are not made at the kernel level. It follows that a kernel should support only mechanisms and that all policy decisions should be left to the outer layer. An operating system is an orderly growth of software over the kernel where all decisions regarding process scheduling, resource allocation, execution environment, file system, and resource protection, etc. are made.

According to Hansen, a kernel is a fundamental set of primitives that allows the dynamic creation and control of processes, as well as communication among them. Thus, the kernel as advocated by Hansen only supports the notion of a process and does not include the concept of a resource. However, as operating systems have matured in functionality and complexity, more functionality has been relegated to the kernel. A kernel should contain a minimal set of functionality that is adequate to build an operating system with a given set of objectives. Including too much functionality in a kernel results in low flexibility at a higher level, whereas including too little functionality in a kernel results in low functional support at a higher level.

An outstanding example of a kernel is the *Hydra*, the kernel of an operating system for C.mmp, a multiprocessor system developed at Carnegie-Mellon University [28]. The Hydra kernel supports the notion of a resource and process, and provides mechanisms for the creation and representation of new types of resources and protected access to resources.

1.3.3 The Virtual Machine Approach

In the virtual machine approach, a *virtual machine software* layer on the bare hardware of the machine gives the illusion that all machine hardware (i.e., the processor, main

memory, secondary storage, etc.) is at the sole disposal of each user. A user can execute the entire instruction set, including the privileged instructions. The virtual machine software creates this illusion by appropriately time-multiplexing the system resources among all the users of the machine.

A user can also run a single-user operating system on this virtual machine. The design of such a single-user operating system can be very simple and efficient because it does not have to deal with the complications that arise due to multiprogramming and protection. The virtual machine concept provides higher flexibility in that it allows different operating systems to run on different virtual machines. Uniprogrammed operating systems can mix with multiprogrammed operating systems. The virtual machine concept provides a useful test-bed to experiment with new operating systems without interfering with other users of the machine. The efficient implementation of virtual machine software (e.g., VM/370), however, is a very difficult problem because virtual machine software is huge and complex.

A classical example of this system is the IBM 370 system [21] wherein the virtual machine software, VM/370, provides a virtual machine to each user. When a user logs on, VM/370 creates a new virtual machine (i.e., a copy of the bare hardware of the IBM 370 system) for the user. In the IBM 370 system, users traditionally run the CMS (Conversational Monitor System) operating system, which is a single-user, interactive operating system.

1.4 WHY ADVANCED OPERATING SYSTEMS

In the 1960s and 1970s, most efforts in operating system design were largely focused on the so-called traditional operating systems, which ran on stand-alone computers with single processors. Considerable advances in integrated circuit and computer communication technologies over the last two decades have spurred unprecedented interest in multicomputer systems and have resulted in the proliferation of a variety of computer architectures, viz., shared memory multiprocessors to distributed memory distributed systems. These multicomputer systems were prompted by the need for high-speed computing that conventional single processor systems were unable to provide [1].

Multiprocessor systems and distributed systems have many idiosyncrasies not present in traditional single-processor systems. These idiosyncrasies render the design of operating systems for these multicomputer systems extremely difficult and require that nontrivial design issues be addressed. Due to their relative newness and enormous design complexity, operating systems for these multicomputers are referred to as *advanced* or *modern* operating systems. An advanced operating system not only harnesses the power of a multicomputer system; it also provides a high-level coherent view of the system; a user views a multicomputer system as a single monolithic powerful machine. A study of advanced operating systems entails a study of these nontrivial design techniques.

Due to the high demand and popularity of multicomputer systems, advanced operating systems have gained substantial importance and a considerable amount of research has been done on them over the last two decades. This book presents a study of advanced operating systems with a special emphasis on the concepts underlying the design techniques.

1.5 TYPES OF ADVANCED OPERATING SYSTEMS

Figure 1.3 gives a classification of advanced operating systems. The impetus for advanced operating systems has come from two directions. First, it has come from advances in the architecture of multicomputer systems and is now driven by a wide variety of high-speed architectures. Hardware design of extremely fast parallel and distributed systems is fairly well understood. These architectures offer great potential for speed up but they also present a substantial challenge to operating system designers. Operating system designs for two types of multicomputer systems, namely, multiprocessor systems and distributed computing systems, have been well-studied.

A second class of advanced operating systems is driven by applications. There are several important applications that require special operating system support, as a requirement as well as for efficiency. General purpose operating systems are too broad in nature and inefficient and fail to provide adequate support for such applications. Two specific applications, namely, database systems and real-time systems, have received considerable attention in the past and the operating system issues for these systems have been extensively examined. Other applications include graphics systems, surveillance, and process control.

A brief introduction of four advanced operating systems follows.

Distributed Operating Systems

Distributed operating systems are operating systems for a network of autonomous computers connected by a communication network. A distributed operating system controls and manages the hardware and software resources of a distributed system such that its users view the entire system as a powerful monolithic computer system. When a program is executed in a distributed system, the user is not aware of where the program is executed or of the location of the resources accessed.

The basic issues in the design of a distributed operating system are the same as in a traditional operating system, viz., process synchronization, deadlocks, scheduling, file systems, interprocess communication, memory and buffer management, failure recovery, etc. However, several idiosyncrasies of a distributed system, namely, the lack of both

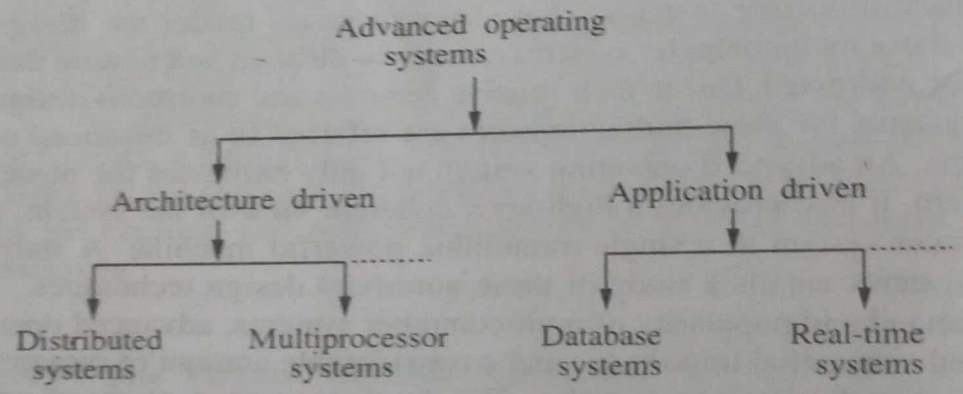

FIGURE 1.3
A classification of advanced operating systems.

shared memory and a physical global clock, and unpredictable communication delays, make the design of distributed operating systems much more difficult.

Multiprocessor Operating Systems

A typical multiprocessor system consists of a set of processors that share a set of physical memory blocks over an interconnection network. Thus, a multiprocessor system is a tightly coupled system where processors share an address space. A multiprocessor operating system controls and manages the hardware and software resources such that users view the entire system as a powerful uniprocessor system; a user is not aware of the presence of multiple processors and the interconnection network.

The basic issues in the design of a multiprocessor operating system are the same as in a traditional operating system. However, the issues of process synchronization, task scheduling, memory management, and protection and security, become more complex because the main memory is shared by many physical processors.

Database Operating Systems

Database systems place special requirements on operating systems. These requirements have their roots in the specific environment that database systems support. A database system must support: the concept of a transaction; operations to store, retrieve, and manipulate a large volume of data efficiently; primitives for concurrency control, and system failure recovery. To store temporary data and data retrieved from secondary storage, it must have a buffer management scheme.

In this book, we primarily focus on concurrency control aspects of database operating systems. Concurrency control, one of the most challenging problems in the design of database operating systems, has been actively studied over the last one and a half decades. An elegant theory of concurrency control exists and a rich set of algorithms to solve the problem have been developed. Recovery and fault tolerance are covered in Chaps. 12 and 13.

Real-time Operating Systems

Real-time systems also place special requirements on operating systems, which have their roots in the specific application that the real-time system is supporting. A distinct feature of real-time systems is that jobs have completion deadlines. A job should be completed before its deadline to be of use (in *soft* real-time systems) or to avert a disaster (in *hard* real-time systems). The major issue in the design of real-time operating systems is the scheduling of jobs in such a way that a maximum number of jobs satisfy their deadlines. Other issues include designing languages and primitives to effectively prepare and execute a job schedule.

1.6 AN OVERVIEW OF THE BOOK

In this book, we study three types of advanced operating systems, namely, distributed operating systems, multiprocessor operating systems, and database operating systems. Based on these topics, the book is divided into seven parts.

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.

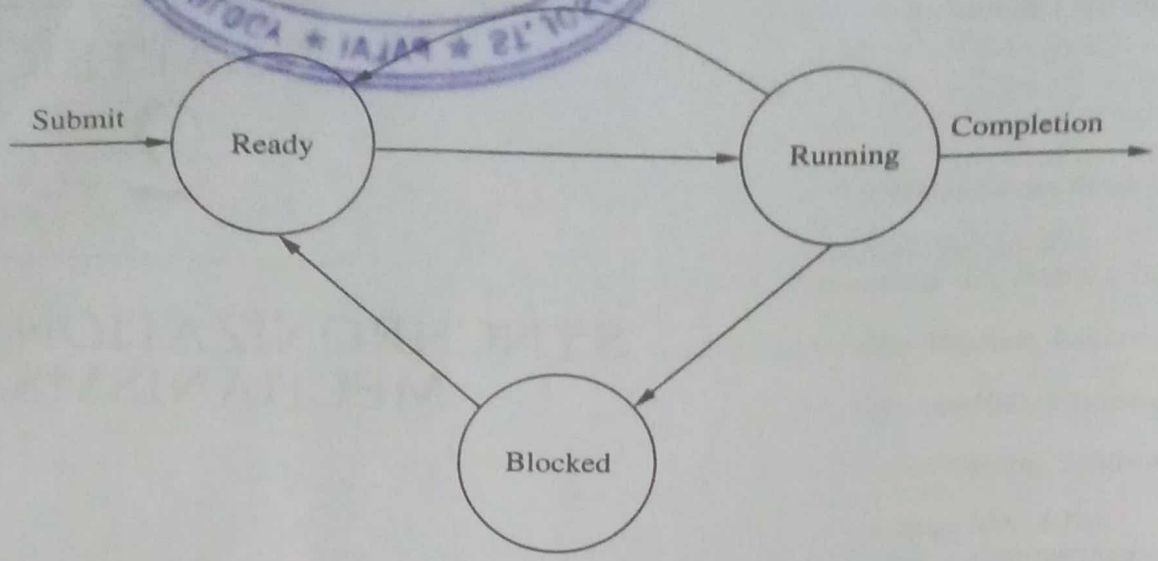

FIGURE 2.1
State transition diagram of a process.

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 synchronization is necessary. In the following sections, we discuss some common synchroniza-

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).

2.6.1 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;
```

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

2.6.2 Serializers

Hewitt and Atkinson [8] proposed *serializers* as a synchronization mechanism to overcome some of the deficiencies of monitors. Serializers allow concurrency inside and thus the shared resource can be encapsulated in a serializer. Serializers replace explicit signaling required by monitors with automatic signaling. This is achieved by requiring the condition for resuming the execution of a waiting process to be explicitly stated when a process waits.

The basic structure of a serializer is similar to a monitor. Like monitors, serializers are abstract data types defined by a set of procedures (or operations) and can encapsulate the shared resource to form a protected resource object. The operations users invoke to access the resource are actually the operations of the serializers. Only one process has access to the serializer at a time. However, procedures of a serializer may have *hollow* regions wherein multiple processes can be concurrently active (see Fig. 2.7). When a process enters a hollow region, it releases the possession of the serializer and consequently, some other process can gain possession of the serializer. Any number of processes can be active in a hollow region. A hollow region in a procedure is specified by a *join-crowd* operation that allows processes to access the resource while releasing (but not exiting) the serializer, thereby allowing concurrency. The syntax of the join-crowd command is

join-crowd (<crowd>) **then** <body> **end**

On invocation of a join-crowd operation, possession of the serializer is released, the identity of the process invoking the join-crowd is recorded in the *crowd*, and the list of statements in the *body* is executed. At the end of the execution of the body, a *leave-crowd* operation is executed which results in the process regaining the possession of the serializer and the removal of the identity of the process from the crowd.

The queue of a serializer is somewhat different than a monitor queue. Instead of condition variables, a serializer has *queue* variables. An *enqueue* operation, along with

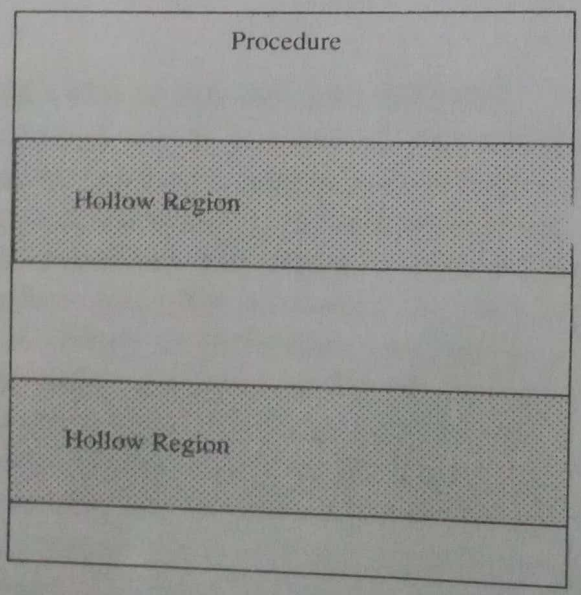

FIGURE 2.7
The structure of a procedure of a serializer.

the condition the process is waiting for, provides a delaying or blocking facility. The syntax of the enqueue command is

enqueue (<priority>, <queue-name>) **until** (<condition>)

where *priority* specifies the priority of the process to be delayed. A process invoking the enqueue is placed at an appropriate position (based on the priority specified) of the specified queue and the condition is not checked until the process reaches the head of the queue. The serializer mechanism automatically restarts the process at the head of the queue when condition for which it is waiting is satisfied and no other process has control of the serializer. No explicit signaling is required, in contrast to monitors.

Serializers derive their name from the fact that all of the events that gain and release possession of the serializer are totally ordered (serial) in time. A typical sequence of events occurring in the use of a protected resource follows. A process gains possession of the serializer as a result of an *entry* event. The process waits (possession of the serializer is released) until a proper condition is established before accessing the resource. An *establish* event regains possession of the serializer as a result of a *guarantee* event, with the proper condition for accessing the resource established to be true. Then, a *join-crowd* event releases the possession of the serializer, records that there is another process in the crowd—an internal data structure of the serializer—that keeps track of which processes are using the resource. The *leave-crowd* event releases the resource, regaining possession of the serializer. An *exit* event releases the serializer. There is also a *timeout* event which regains possession of the serializer as a result of waiting for a condition longer than the specified period.

Example 2.5. Figure 2.8 gives a solution to the readers-priority problem using serializers. Note that *empty(crowd)* and *empty(queue)* operations permit us to check if a crowd or queue is empty. On invoking the procedure *read*, a reader process is blocked if there is an active writer (i.e., wcrowd is not empty); otherwise, reader proceeds and executes join-crowd operation thereby joining the crowd *rcrowd* (i.e., the presence of a reader in the resource (db) is recorded) and releasing the possession of the serializer. Releasing the serializer facilitates concurrent access to the resource by allowing another reader to gain the control of the serializer and execute a join-crowd operation. On completing the read operation, a reader leaves the body of the join-crowd which causes the automatic execution of a leave-crowd operation. The leave-crowd operation results in a reader regaining control of the serializer and its removal from the rcrowd (i.e., the reader is no longer accessing the resource).

A writer, on the other hand, invokes the *write* procedure and proceeds to execute a join-crowd operation, only if there are no active writers (wcrowd is empty), no active readers (rcrowd is empty), and no readers are waiting (readq is empty); otherwise, a writer process is blocked and is queued in the *writeq*. On executing a join-crowd operation, a writer joins the crowd *wcrowd* and releases the possession of the serializer. Reader and writer processes trying to gain access to the resource when a writer is active are blocked and queued in readq and writeq, respectively. On completion of the access, a writer leaves the body of the join-crowd, causing the automatic execution of a leave-crowd operation. The leave-crowd operation results in the writer regaining control of the serializer and in the writer's removal from the wcrowd. Once a writer exits the serializer, wcrowd is empty and the condition for waiting readers becomes true, and they proceed to execute join-crowd one by one. If

there are no waiting readers, then wcrowd, rcrowd, and readq are empty and hence a waiting writer, if present, will proceed to execute the join-crowd operation.

Weak reader's priority solution. A weak reader's priority solution can be obtained by simply replacing the enqueue command in writer procedure by the following command:

```
enqueue (writeq) until (empty(wcrowd) AND empty(rcrowd));
```

```
readerwriter : serializer
var

    readq : queue;
    writeq : queue;
    rcrowd : crowd; (* readers crowd *)
    wcrowd : crowd; (* writers crowd *)
    db : database; (* the shared resource *)

procedure read (k:key; var data : datatype);

    begin
    enqueue (readq) until empty(wcrowd);
    joincrowd (rcrowd) then

     data := read-opn(db[key]);
     end

    return (data);

end read;

procedure write (k:key, data:datatype);

    begin
    enqueue (writeq) until
    (empty(wcrowd) AND empty(rcrowd) AND empty(readq));
    joincrowd (wcrowd) then

     write-opn (db[key], data);
     end

end write;
```

FIGURE 2.8
A serializer solution to the readers-priority problem.

That is, a writer does not have to wait for readq to become empty. Consequently, when a writer departs and both a reader and a writer are waiting, dequeue conditions for both are satisfied and one of them is dequeued randomly.

Writer's priority solution. To obtain a writer's priority solution, we need to replace the enqueue command in writer procedure by the following command:

enqueue (writeq) **until** (empty(wcrowd) **AND** empty(rcrowd));

and also replace the enqueue command in the reader procedure by the following command:

enqueue (readq) **until** (empty(wcrowd) **AND** empty(writeq));

A major drawback of serializers is that they are more complex than monitors and therefore less efficient. For example, crowd is not a simple counter, but a complex data structure that stores the identity of processes. Also, the automatic signaling feature, while simplifying the task of a programmer, comes at a cost of higher overhead. Automatic signaling requires testing conditions waited upon by processes at the head of every queue every time possession of the serializer is relinquished.

2.6.3 Path Expressions

The concept of *path expression* was proposed by Campbell and Habermann [2]. Conceptually, a "path expression" is a quite different approach to process synchronization. A path expression restricts the set of admissible execution histories of the operations on the shared resource so that no incorrect state is ever reached and it indicates the order in which operations on a shared resource can be interleaved. A path expression has the following form

path S **end**;

where S denotes possible execution histories. It is an expression whose variables are the operations on the resource and whose operators are:

Sequencing (;). It defines a sequencing order among operations. For example, **path** open; read; close; **end** means that an open must be performed first, followed by a read and a close in that order. There is no concurrency in the execution of these operations.

Selection (+). It signifies that only one of the operations connected by a + operator can be executed at a time. For example, **path** read + write **end** means that only read or only write can be executed at a given time, but the order of execution of these operations does not matter.

Concurrency ({}). It signifies that any number of instances of the operation delimited by { and } can be in execution at a time. For example, **path** {read} **end** means that any number of read operations can be executed concurrently. The path expression **path** write; {read} **end** allows either several read operations or a single write operation to be executed at any time (read and write operations exclude each other). However, whenever the system is empty after all readers have finished, the writer must execute

first. Between every two write operations, at least one read operation must be executed. The path expression **path** {write; read} **end** means that at any time there can be any number of instantiations of the path write; read. At any instant, the number of read operations executed is less than or equal to the number of write operations executed. The path expression **path** {write + read} **end** is meaningless and does not impose any restriction on the execution of read and write operations.