PART TWO

Processes

The fundamental task of any modern operating system is process management. The operating system must allocate resources to processes, enable processes to share and exchange information, protect the resources of each process from other processes, and enable synchronization among processes. To meet these requirements, the operating system must maintain a data structure for each process that describes the state and resource ownership of that process and that enables the operating system to exert process control.

On a multiprogramming uniprocessor, the execution of multiple processes can be interleaved in time. On a multiprocessor, not only may process execution be interleaved, but also multiple processes can execute simultaneously. Both interleaved and simultaneous execution are types of concurrency and lead to a host of difficult problems, both for the application programmer and the operating system.

In many contemporary operating systems, the difficulties of process management are compounded by the introduction of the concept of thread. In a multithreaded system, the process retains the attributes of resource ownership, while the attribute of multiple, concurrent execution streams is a property of threads running within a process.

ROAD MAP FOR PART TWO

Chapter 3 Process Description and Control

The focus of a traditional operating system is the management of processes. Each process is, at any time, in one of a number of execution states, including Ready, Running, and Blocked. The operating system keeps track of these execution states and manages the movement of processes among the states. For this purpose the operating system maintains rather elaborate data structures describing each process. The operating system must perform the scheduling function and provide facilities for process sharing and synchronization. Chapter 3 looks at the data structures and techniques used in a typical operating system for process management.

Chapter 4 Threads, SMP, and Microkernels

Chapter 4 covers three areas that characterize many contemporary operating systems and that represent advances over traditional operating system design. In many operating systems, the traditional concept of process has been split into two parts: one dealing with resource ownership (process) and one dealing with the stream of instruction execution (thread). A single process may contain multiple threads. A multithreaded organization has advantages both in the structuring of applications and in performance. Chapter 4 also examines the symmetric multiprocessor (SMP), which is a computer system with multiple processors, each of which is able to execute all application and system code. SMP organization enhances performance and reliability. SMP is often used in conjunction with multithreading but can have powerful performance benefits even without multithreading. Finally, Chapter 4 examines the microkernel, which is a style of operating system design that minimizes the amount of system code that runs in kernel mode. The advantages of this approach are analyzed.

Chapter 5 Concurrency: Mutual Exclusion and Synchronization

The two central themes of modern operating systems are multiprogramming and distributed processing. Fundamental to both these themes, and fundamental to the technology of operating system design, is concurrency. Chapter 5 looks at two aspects of concurrency control: mutual exclusion and synchronization. Mutual exclusion refers to the ability of multiple processes (or threads) to share code, resources, or data in such a way that only one process has access to the shared object at a time. Related to mutual exclusion is synchronization: the ability of multiple processes to coordinate their activities by the exchange of information. Chapter 5 provides a broad treatment of issues related to concurrency, beginning with a discussion of the design issues involved. The chapter provides a discussion of hardware support for concurrency and then looks at the most important mechanisms to support concurrency: semaphores, monitors, and message passing.

Chapter 6 Concurrency: Deadlock and Starvation

Chapter 6 looks at two additional aspects of concurrency control. *Deadlock* refers to a situation in which a set of two or more processes are waiting for other members of the set to complete an operation in order to proceed, but none of the members is able to proceed. Deadlock is a difficult phenomenon to anticipate, and there are no easy general solutions to this problem. Chapter 6 looks at the three major approaches to dealing with deadlock: prevention, avoidance, and detection. *Starvation* refers to a situation in which a process is ready to execute but is continuously denied access to a processor in deference to other processes. In large part, starvation is dealt with as a scheduling issue and is therefore treated in Part Four. Although Chapter 6 focuses on deadlock, starvation is addressed in the context that solutions to deadlock need to avoid the problem of starvation.

CHAPTER 3

PROCESS DESCRIPTION AND CONTROL

The design of an operating system (OS) reflects certain general requirements. All multiprogramming operating systems, from single-user systems such as Windows 98 to mainframe systems such as IBM's mainframe operating system, z/OS, which can support thousands of users, are built around the concept of the process. Most requirements that the OS must meet can be expressed with reference to processes:

- The OS must interleave the execution of multiple processes, to maximize processor utilization while providing reasonable response time.
- The OS must allocate resources to processes in conformance with a specific policy (e.g., certain functions or applications are of higher priority) while at the same time avoiding deadlock.[1]
- The OS may be required to support interprocess communication and user creation of processes, both of which may aid in the structuring of applications.

We begin our detailed study of operating systems with an examination of the way in which they represent and control processes. After an introduction to the concept of a process, the chapter discusses process states, which characterize the behavior of processes. Then we look at the data structures that the OS uses to manage processes. These include data structures to represent the state of each process and data structures that record other characteristics of processes that the OS needs to achieve its objectives. Next, we look at the ways in which the OS uses these data structures to control process execution. Finally, we discuss process management in UNIX SVR4. Chapter 4 provides more modern examples of process management, namely Solaris, Windows, and Linux.

Note: In this chapter, reference is occasionally made to virtual memory. Much of the time, we can ignore this concept in dealing with processes, but at certain points in the discussion, virtual memory considerations are pertinent. Virtual memory is not discussed in detail until Chapter 8; a brief overview is provided in Chapter 2.

3.1 WHAT IS A PROCESS?

Background

Before defining the term *process*, it is useful to summarize some of the concepts introduced in Chapters 1 and 2:

1. A computer platform consists of a collection of hardware resources, such as the processor, main memory, I/O modules, timers, disk drives, and so on.
2. Computer applications are developed to perform some task. Typically, they accept input from the outside world, perform some processing, and generate output.
3. It is inefficient for applications to be written directly for a given hardware platform. The principal reasons for this are as follows:

[1]Deadlock is examined in Chapter 6. As a simple example, deadlock occurs if two processes need the same two resources to continue and each has ownership of one. Unless some action is taken, each process will wait indefinitely for the missing resource.

a. Numerous applications can be developed for the same platform. Thus, it makes sense to develop common routines for accessing the computer's resources.
b. The processor itself provides only limited support for multiprogramming. Software is needed to manage the sharing of the processor and other resources by multiple applications at the same time.
c. When multiple applications are active at the same time, it is necessary to protect the data, I/O use, and other resource use of each application from the others.

4. The OS was developed to provide a convenient, feature-rich, secure, and consistent interface for applications to use. The OS is a layer of software between the applications and the computer hardware (Figure 2.1) that supports applications and utilities.
5. We can think of the OS as providing a uniform, abstract representation of resources that can be requested and accessed by applications. Resources include main memory, network interfaces, file systems, and so on. Once the OS has created these resource abstractions for applications to use, it must also manage their use. For example, an OS may permit resource sharing and resource protection.

Now that we have the concepts of applications, system software, and resources, we are in a position to discuss how the OS can, in an orderly fashion, manage the execution of applications so that

- Resources are made available to multiple applications.
- The physical processor is switched among multiple applications so all will appear to be progressing.
- The processor and I/O devices can be used efficiently.

The approach taken by all modern operating systems is to rely on a model in which the execution of an application corresponds to the existence of one or more processes.

Processes and Process Control Blocks

Recall from Chapter 2 that we suggested several definitions of the term *process*, including

- A program in execution
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system resources

We can also think of a process as an entity that consists of a number of elements. Two essential elements of a process are **program code** (which may be shared with other processes that are executing the same program) and a **set of data** associated with that code. Let us suppose that the processor begins to execute this program

code, and we refer to this executing entity as a process. At any given point in time, *while the program is executing*, this process can be uniquely characterized by a number of elements, including the following:

- **Identifier:** A unique identifier associated with this process, to distinguish it from all other processes.
- **State:** If the process is currently executing, it is in the running state.
- **Priority:** Priority level relative to other processes.
- **Program counter:** The address of the next instruction in the program to be executed.
- **Memory pointers:** Includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes.
- **Context data:** These are data that are present in registers in the processor while the process is executing.
- **I/O status information:** Includes outstanding I/O requests, I/O devices (e.g., tape drives) assigned to this process, a list of files in use by the process, and so on.
- **Accounting information:** May include the amount of processor time and clock time used, time limits, account numbers, and so on.

The information in the preceding list is stored in a data structure, typically called a **process control block** (Figure 3.1), that is created and managed by the OS. The significant point about the process control block is that it contains sufficient

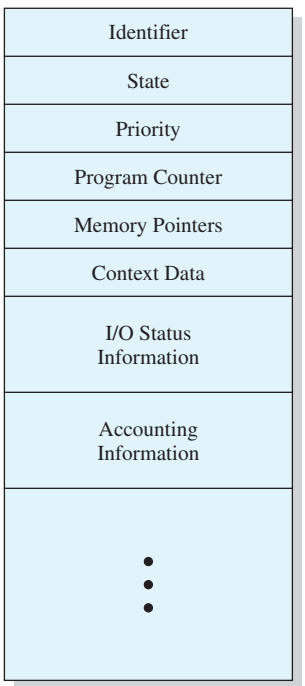


Figure 3.1 Simplified Process Control Block

information so that it is possible to interrupt a running process and later resume execution as if the interruption had not occurred. The process control block is the key tool that enables the OS to support multiple processes and to provide for multiprocessing. When a process is interrupted, the current values of the program counter and the processor registers (context data) are saved in the appropriate fields of the corresponding process control block, and the state of the process is changed to some other value, such as *blocked* or *ready* (described subsequently). The OS is now free to put some other process in the running state. The program counter and context data for this process are loaded into the processor registers and this process now begins to execute.

Thus, we can say that a process consists of program code and associated data plus a process control block. For a single-processor computer, at any given time, at most one process is executing and that process is in the *running* state.

3.2 PROCESS STATES

As just discussed, for a program to be executed, a process, or task, is created for that program. From the processor's point of view, it executes instructions from its repertoire in some sequence dictated by the changing values in the program counter register. Over time, the program counter may refer to code in different programs that are part of different processes. From the point of view of an individual program, its execution involves a sequence of instructions within that program.

We can characterize the behavior of an individual process by listing the sequence of instructions that execute for that process. Such a listing is referred to as a **trace** of the process. We can characterize behavior of the processor by showing how the traces of the various processes are interleaved.

Let us consider a very simple example. Figure 3.2 shows a memory layout of three processes. To simplify the discussion, we assume no use of virtual memory; thus all three processes are represented by programs that are fully loaded in main memory. In addition, there is a small **dispatcher** program that switches the processor from one process to another. Figure 3.3 shows the traces of each of the processes during the early part of their execution. The first 12 instructions executed in processes A and C are shown. Process B executes four instructions, and we assume that the fourth instruction invokes an I/O operation for which the process must wait.

Now let us view these traces from the processor's point of view. Figure 3.4 shows the interleaved traces resulting from the first 52 instruction cycles (for convenience, the instruction cycles are numbered). In this figure, the shaded areas represent code executed by the dispatcher. The same sequence of instructions is executed by the dispatcher in each instance because the same functionality of the dispatcher is being executed. We assume that the OS only allows a process to continue execution for a maximum of six instruction cycles, after which it is interrupted; this prevents any single process from monopolizing processor time. As Figure 3.4 shows, the first six instructions of process A are executed, followed by a time-out and the execution of some code in the dispatcher, which executes six instructions before

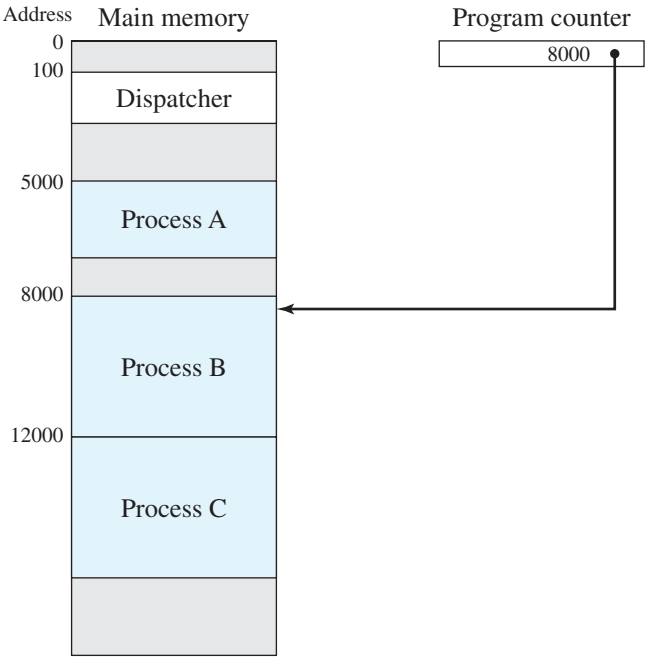


Figure 3.2 **Snapshot of Example Execution (Figure 3.4) at Instruction Cycle 13**

(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

Figure 3.3 **Traces of Processes of Figure 3.2**

1	5000	27	12004
2	5001	28	12005
3	5002	----------Timeout	
4	5003	29	100
5	5004	30	101
6	5005	31	102
----------Timeout		32	103
7	100	33	104
8	101	34	105
9	102	35	5006
10	103	36	5007
11	104	37	5008
12	105	38	5009
13	8000	39	5010
14	8001	40	5011
15	8002	----------Timeout	
16	8003	41	100
----------I/O Request		42	101
17	100	43	102
18	101	44	103
19	102	45	104
20	103	46	105
21	104	47	12006
22	105	48	12007
23	12000	49	12008
24	12001	50	12009
25	12002	51	12010
26	12003	52	12011
		----------Timeout	

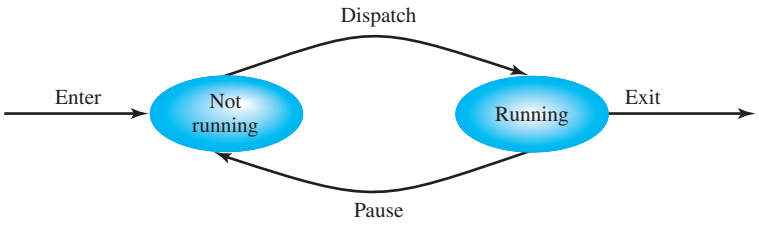
100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

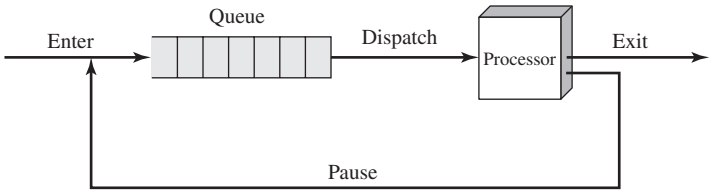
Figure 3.4 Combined Trace of Processes of Figure 3.2

turning control to process B.[2] After four instructions are executed, process B requests an I/O action for which it must wait. Therefore, the processor stops executing process B and moves on, via the dispatcher, to process C. After a time-out, the processor moves back to process A. When this process times out, process B is still waiting for the I/O operation to complete, so the dispatcher moves on to process C again.

[2]The small numbers of instructions executed for the processes and the dispatcher are unrealistically low; they are used in this simplified example to clarify the discussion.



(a) State transition diagram



(b) Queuing diagram

Figure 3.5 Two-State Process Model

A Two-State Process Model

The operating system's principal responsibility is controlling the execution of processes; this includes determining the interleaving pattern for execution and allocating resources to processes. The first step in designing an OS to control processes is to describe the behavior that we would like the processes to exhibit.

We can construct the simplest possible model by observing that, at any time, a process is either being executed by a processor or not. In this model, a process may be in one of two states: Running or Not Running, as shown in Figure 3.5a. When the OS creates a new process, it creates a process control block for the process and enters that process into the system in the Not Running state. The process exists, is known to the OS, and is waiting for an opportunity to execute. From time to time, the currently running process will be interrupted and the dispatcher portion of the OS will select some other process to run. The former process moves from the Running state to the Not Running state, and one of the other processes moves to the Running state.

From this simple model, we can already begin to appreciate some of the design elements of the OS. Each process must be represented in some way so that the OS can keep track of it. That is, there must be some information relating to each process, including current state and location in memory; this is the process control block. Processes that are not running must be kept in some sort of queue, waiting their turn to execute. Figure 3.5b suggests a structure. There is a single queue in which each entry is a pointer to the process control block of a particular process. Alternatively, the queue may consist of a linked list of data blocks, in which each block represents one process; we will explore this latter implementation subsequently.

Table 3.1 Reasons for Process Creation

New batch job	The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands.
Interactive logon	A user at a terminal logs on to the system.
Created by OS to provide a service	The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

We can describe the behavior of the dispatcher in terms of this queuing diagram. A process that is interrupted is transferred to the queue of waiting processes. Alternatively, if the process has completed or aborted, it is discarded (exits the system). In either case, the dispatcher takes another process from the queue to execute.

The Creation and Termination of Processes

Before refining our simple two-state model, it will be useful to discuss the creation and termination of processes; ultimately, and regardless of the model of process behavior that is used, the life of a process is bounded by its creation and termination.

Process Creation When a new process is to be added to those currently being managed, the OS builds the data structures that are used to manage the process and allocates address space in main memory to the process. We describe these data structures in Section 3.3. These actions constitute the creation of a new process.

Four common events lead to the creation of a process, as indicated in Table 3.1. In a batch environment, a process is created in response to the submission of a job. In an interactive environment, a process is created when a new user attempts to log on. In both cases, the OS is responsible for the creation of the new process. An OS may also create a process on behalf of an application. For example, if a user requests that a file be printed, the OS can create a process that will manage the printing. The requesting process can thus proceed independently of the time required to complete the printing task.

Traditionally, the OS created all processes in a way that was transparent to the user or application program, and this is still commonly found with many contemporary operating systems. However, it can be useful to allow one process to cause the creation of another. For example, an application process may generate another process to receive data that the application is generating and to organize those data into a form suitable for later analysis. The new process runs in parallel to the original process and is activated from time to time when new data are available. This arrangement can be very useful in structuring the application. As another example, a server process (e.g., print server, file server) may generate a new process for each request that it handles. When the OS creates a process at the explicit request of another process, the action is referred to as **process spawning**.

Table 3.2 Reasons for Process Termination

Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (for example, if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

When one process spawns another, the former is referred to as the **parent process**, and the spawned process is referred to as the **child process**. Typically, the "related" processes need to communicate and cooperate with each other. Achieving this cooperation is a difficult task for the programmer; this topic is discussed in Chapter 5.

Process Termination Table 3.2 summarizes typical reasons for process termination. Any computer system must provide a means for a process to indicate its completion. A batch job should include a Halt instruction or an explicit OS service call for termination. In the former case, the Halt instruction will generate an interrupt to alert the OS that a process has completed. For an interactive application, the action of the user will indicate when the process is completed. For example, in a time-sharing system, the process for a particular user is to be terminated when the user logs off or turns off his or her terminal. On a personal computer or workstation, a user may quit an application (e.g., word processing or spreadsheet). All of these actions ultimately result in a service request to the OS to terminate the requesting process.

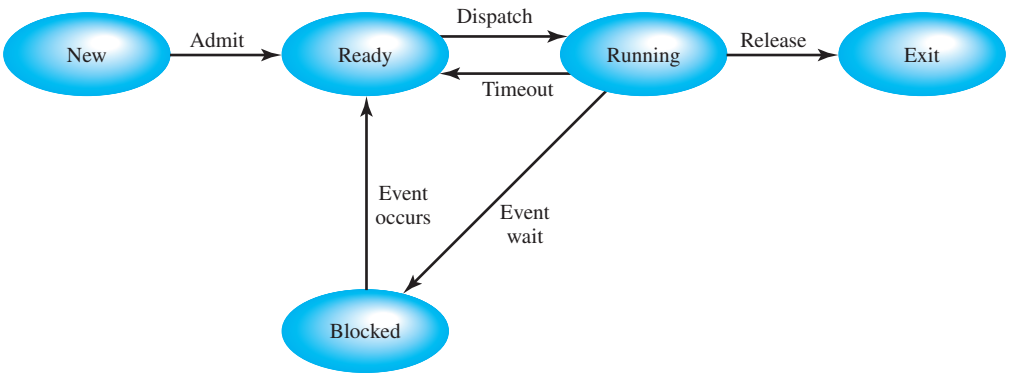


Figure 3.6 **Five-State Process Model**

Additionally, a number of error and fault conditions can lead to the termination of a process. Table 3.2 lists some of the more commonly recognized conditions.[3]

Finally, in some operating systems, a process may be terminated by the process that created it or when the parent process is itself terminated.

A Five-State Model

If all processes were always ready to execute, then the queuing discipline suggested by Figure 3.5b would be effective. The queue is a first-in-first-out list and the processor operates in **round-robin** fashion on the available processes (each process in the queue is given a certain amount of time, in turn, to execute and then returned to the queue, unless blocked). However, even with the simple example that we have described, this implementation is inadequate: some processes in the Not Running state are ready to execute, while others are blocked, waiting for an I/O operation to complete. Thus, using a single queue, the dispatcher could not just select the process at the oldest end of the queue. Rather, the dispatcher would have to scan the list looking for the process that is not blocked and that has been in the queue the longest.

A more natural way to handle this situation is to split the Not Running state into two states: Ready and Blocked. This is shown in Figure 3.6. For good measure, we have added two additional states that will prove useful. The five states in this new diagram are as follows:

- **Running:** The process that is currently being executed. For this chapter, we will assume a computer with a single processor, so at most one process at a time can be in this state.
- **Ready:** A process that is prepared to execute when given the opportunity.
- **Blocked/Waiting:**[4] A process that cannot execute until some event occurs, such as the completion of an I/O operation.

[3] A forgiving operating system might, in some cases, allow the user to recover from a fault without terminating the process. For example, if a user requests access to a file and that access is denied, the operating system might simply inform the user that access is denied and allow the process to proceed.

[4] *Waiting* is a frequently used alternative term for *Blocked* as a process state. Generally, we will use *Blocked*, but the terms are interchangeable.

- **New:** A process that has just been created but has not yet been admitted to the pool of executable processes by the OS. Typically, a new process has not yet been loaded into main memory, although its process control block has been created.
- **Exit:** A process that has been released from the pool of executable processes by the OS, either because it halted or because it aborted for some reason.

The New and Exit states are useful constructs for process management. The New state corresponds to a process that has just been defined. For example, if a new user attempts to log onto a time-sharing system or a new batch job is submitted for execution, the OS can define a new process in two stages. First, the OS performs the necessary housekeeping chores. An identifier is associated with the process. Any tables that will be needed to manage the process are allocated and built. At this point, the process is in the New state. This means that the OS has performed the necessary actions to create the process but has not committed itself to the execution of the process. For example, the OS may limit the number of processes that may be in the system for reasons of performance or main memory limitation. While a process is in the new state, information concerning the process that is needed by the OS is maintained in control tables in main memory. However, the process itself is not in main memory. That is, the code of the program to be executed is not in main memory, and no space has been allocated for the data associated with that program. While the process is in the New state, the program remains in secondary storage, typically disk storage.[5]

Similarly, a process exits a system in two stages. First, a process is terminated when it reaches a natural completion point, when it aborts due to an unrecoverable error, or when another process with the appropriate authority causes the process to abort. Termination moves the process to the exit state. At this point, the process is no longer eligible for execution. The tables and other information associated with the job are temporarily preserved by the OS, which provides time for auxiliary or support programs to extract any needed information. For example, an accounting program may need to record the processor time and other resources utilized by the process for billing purposes. A utility program may need to extract information about the history of the process for purposes related to performance or utilization analysis. Once these programs have extracted the needed information, the OS no longer needs to maintain any data relating to the process and the process is deleted from the system.

Figure 3.6 indicates the types of events that lead to each state transition for a process; the possible transitions are as follows:

- **Null → New:** A new process is created to execute a program. This event occurs for any of the reasons listed in Table 3.1.
- **New → Ready:** The OS A will move a process from the New state to the Ready state when it is prepared to take on an additional process. Most systems set some limit based on the number of existing processes or the amount of virtual

[5]In the discussion in this paragraph, we ignore the concept of virtual memory. In systems that support virtual memory, when a process moves from New to Ready, its program code and data are loaded into virtual memory. Virtual memory was briefly discussed in Chapter 2 and is examined in detail in Chapter 8.

memory committed to existing processes. This limit assures that there are not so many active processes as to degrade performance.

- **Ready → Running:** When it is time to select a process to run, the OS chooses one of the processes in the Ready state. This is the job of the scheduler or dispatcher. Scheduling is explored in Part Four.
- **Running → Exit:** The currently running process is terminated by the OS if the process indicates that it has completed, or if it aborts. See Table 3.2.
- **Running → Ready:** The most common reason for this transition is that the running process has reached the maximum allowable time for uninterrupted execution; virtually all multiprogramming operating systems impose this type of time discipline. There are several other alternative causes for this transition, which are not implemented in all operating systems. Of particular importance is the case in which the OS assigns different levels of priority to different processes. Suppose, for example, that process A is running at a given priority level, and process B, at a higher priority level, is blocked. If the OS learns that the event upon which process B has been waiting has occurred, moving B to a ready state, then it can interrupt process A and dispatch process B. We say that the OS has **preempted** process A.[6] Finally, a process may voluntarily release control of the processor. An example is a background process that performs some accounting or maintenance function periodically.
- **Running → Blocked:** A process is put in the Blocked state if it requests something for which it must wait. A request to the OS is usually in the form of a system service call; that is, a call from the running program to a procedure that is part of the operating system code. For example, a process may request a service from the OS that the OS is not prepared to perform immediately. It can request a resource, such as a file or a shared section of virtual memory, that is not immediately available. Or the process may initiate an action, such as an I/O operation, that must be completed before the process can continue. When processes communicate with each other, a process may be blocked when it is waiting for another process to provide data or waiting for a message from another process.
- **Blocked → Ready:** A process in the Blocked state is moved to the Ready state when the event for which it has been waiting occurs.
- **Ready → Exit:** For clarity, this transition is not shown on the state diagram. In some systems, a parent may terminate a child process at any time. Also, if a parent terminates, all child processes associated with that parent may be terminated.
- **Blocked → Exit:** The comments under the preceding item apply.

Returning to our simple example, Figure 3.7 shows the transition of each process among the states. Figure 3.8a suggests the way in which a queuing discipline

[6]In general, the term *preemption* is defined to be the reclaiming of a resource from a process before the process is finished using it. In this case, the resource is the processor itself. The process is executing and could continue to execute, but is preempted so that another process can be executed.

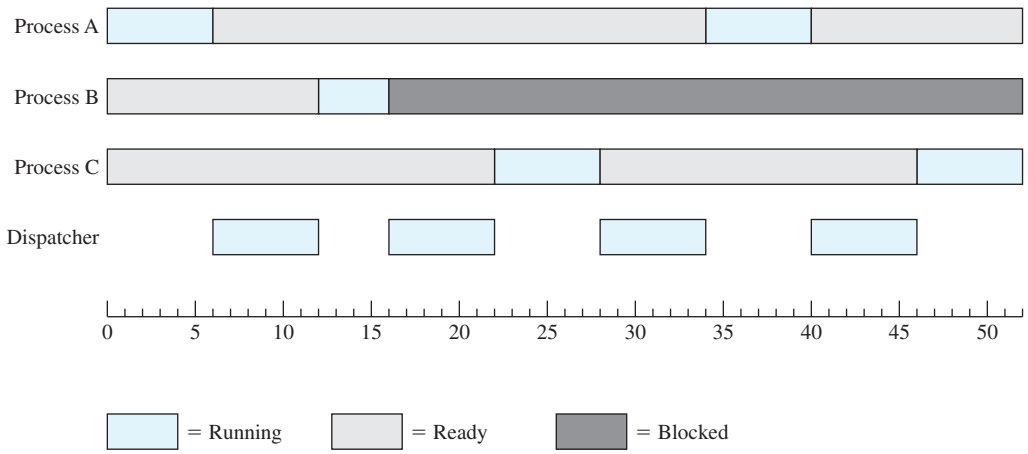
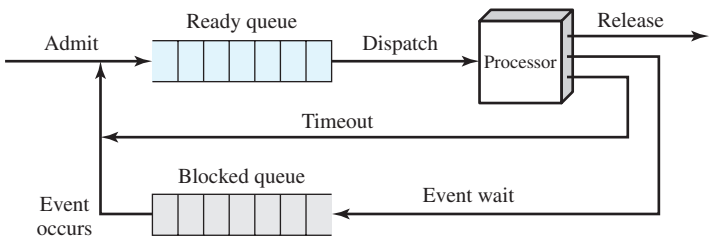
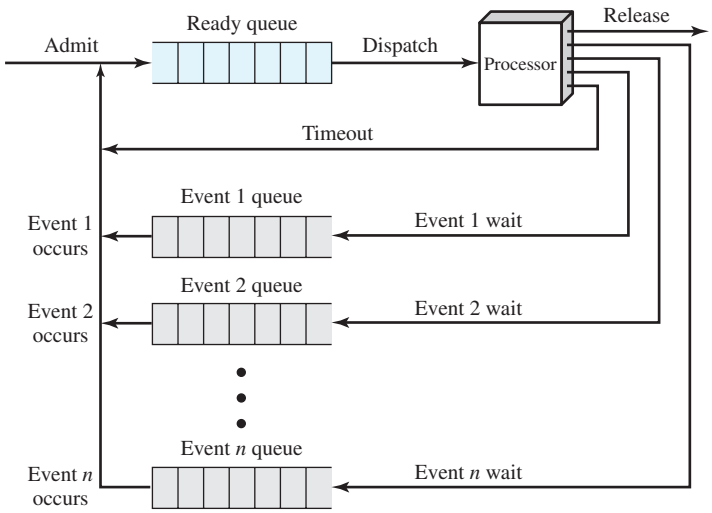


Figure 3.7 **Process States for the Trace of Figure 3.4**



(a) Single blocked queue



(b) Multiple blocked queues

Figure 3.8 **Queuing Model for Figure 3.6**

might be implemented with two queues: a Ready queue and a Blocked queue. As each process is admitted to the system, it is placed in the Ready queue. When it is time for the OS to choose another process to run, it selects one from the Ready queue. In the absence of any priority scheme, this can be a simple first-in-first-out queue. When a running process is removed from execution, it is either terminated or placed in the Ready or Blocked queue, depending on the circumstances. Finally, when an event occurs, any process in the Blocked queue that has been waiting on that event only is moved to the Ready queue.

This latter arrangement means that, when an event occurs, the OS must scan the entire blocked queue, searching for those processes waiting on that event. In a large OS, there could be hundreds or even thousands of processes in that queue. Therefore, it would be more efficient to have a number of queues, one for each event. Then, when the event occurs, the entire list of processes in the appropriate queue can be moved to the Ready state (Figure 3.8b).

One final refinement: If the dispatching of processes is dictated by a priority scheme, then it would be convenient to have a number of Ready queues, one for each priority level. The OS could then readily determine which is the highest-priority ready process that has been waiting the longest.

Suspended Processes

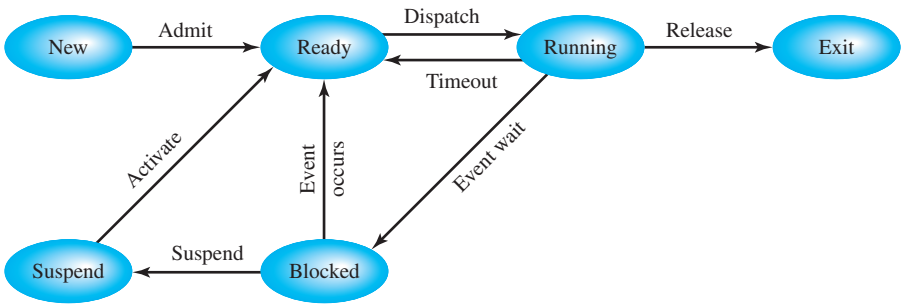
The Need for Swapping The three principal states just described (Ready, Running, Blocked) provide a systematic way of modeling the behavior of processes and guide the implementation of the OS. Some operating systems are constructed using just these three states.

However, there is good justification for adding other states to the model. To see the benefit of these new states, consider a system that does not employ virtual memory. Each process to be executed must be loaded fully into main memory. Thus, in Figure 3.8b, all of the processes in all of the queues must be resident in main memory.

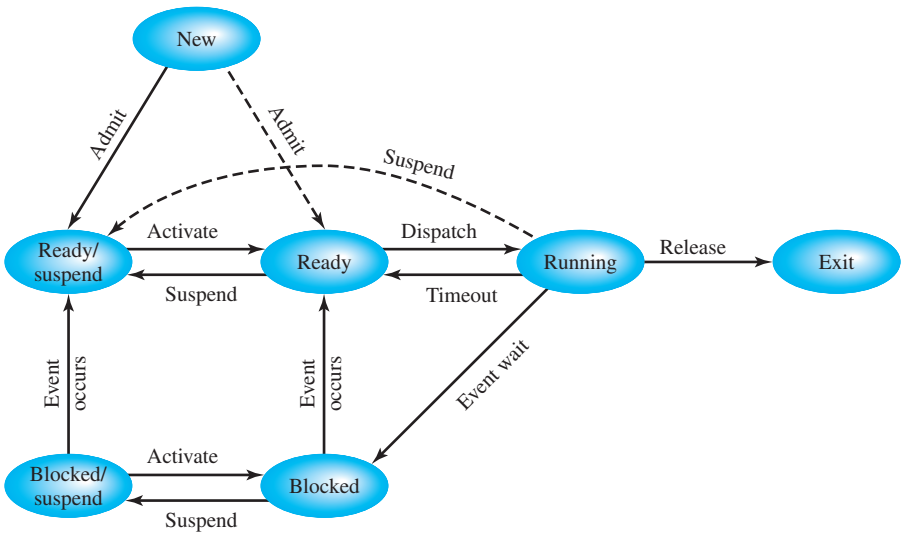
Recall that the reason for all of this elaborate machinery is that I/O activities are much slower than computation and therefore the processor in a uniprogramming system is idle most of the time. But the arrangement of Figure 3.8b does not entirely solve the problem. It is true that, in this case, memory holds multiple processes and that the processor can move to another process when one process is blocked. But the processor is so much faster than I/O that it will be common for all of the processes in memory to be waiting for I/O. Thus, even with multiprogramming, a processor could be idle most of the time.

What to do? Main memory could be expanded to accommodate more processes. But there are two flaws in this approach. First, there is a cost associated with main memory, which, though small on a per-byte basis, begins to add up as we get into the gigabytes of storage. Second, the appetite of programs for memory has grown as fast as the cost of memory has dropped. So larger memory results in larger processes, not more processes.

Another solution is swapping, which involves moving part or all of a process from main memory to disk. When none of the processes in main memory is in the Ready state, the OS swaps one of the blocked processes out onto disk into a suspend queue. This is a queue of existing processes that have been temporarily kicked out of



(a) With one suspend state



(b) With two suspend states

Figure 3.9 Process State Transition Diagram with Suspend States

main memory, or suspended. The OS then brings in another process from the suspend queue, or it honors a new-process request. Execution then continues with the newly arrived process.

Swapping, however, is an I/O operation, and therefore there is the potential for making the problem worse, not better. But because disk I/O is generally the fastest I/O on a system (e.g., compared to tape or printer I/O), swapping will usually enhance performance.

With the use of swapping as just described, one other state must be added to our process behavior model (Figure 3.9a): the Suspend state. When all of the processes in main memory are in the Blocked state, the OS can suspend one process by putting it in the Suspend state and transferring it to disk. The space that is freed in main memory can then be used to bring in another process.

When the OS has performed a swapping-out operation, it has two choices for selecting a process to bring into main memory: It can admit a newly created process or it can bring in a previously suspended process. It would appear that the preference should be to bring in a previously suspended process, to provide it with service rather than increasing the total load on the system.

But this line of reasoning presents a difficulty. All of the processes that have been suspended were in the Blocked state at the time of suspension. It clearly would not do any good to bring a blocked process back into main memory, because it is still not ready for execution. Recognize, however, that each process in the Suspend state was originally blocked on a particular event. When that event occurs, the process is not blocked and is potentially available for execution.

Therefore, we need to rethink this aspect of the design. There are two independent concepts here: whether a process is waiting on an event (blocked or not) and whether a process has been swapped out of main memory (suspended or not). To accommodate this 2×2 combination, we need four states:

- **Ready:** The process is in main memory and available for execution.
- **Blocked:** The process is in main memory and awaiting an event.
- **Blocked/Suspend:** The process is in secondary memory and awaiting an event.
- **Ready/Suspend:** The process is in secondary memory but is available for execution as soon as it is loaded into main memory.

Before looking at a state transition diagram that encompasses the two new suspend states, one other point should be mentioned. The discussion so far has assumed that virtual memory is not in use and that a process is either all in main memory or all out of main memory. With a virtual memory scheme, it is possible to execute a process that is only partially in main memory. If reference is made to a process address that is not in main memory, then the appropriate portion of the process can be brought in. The use of virtual memory would appear to eliminate the need for explicit swapping, because any desired address in any desired process can be moved in or out of main memory by the memory management hardware of the processor. However, as we shall see in Chapter 8, the performance of a virtual memory system can collapse if there is a sufficiently large number of active processes, all of which are partially in main memory. Therefore, even in a virtual memory system, the OS will need to swap out processes explicitly and completely from time to time in the interests of performance.

Let us look now, in Figure 3.9b, at the state transition model that we have developed. (The dashed lines in the figure indicate possible but not necessary transitions.) Important new transitions are the following:

- **Blocked → Blocked/Suspend:** If there are no ready processes, then at least one blocked process is swapped out to make room for another process that is not blocked. This transition can be made even if there are ready processes available, if the OS determines that the currently running process or a ready process that it would like to dispatch requires more main memory to maintain adequate performance.
- **Blocked/Suspend → Ready/Suspend:** A process in the Blocked/Suspend state is moved to the Ready/Suspend state when the event for which it has been

waiting occurs. Note that this requires that the state information concerning suspended processes must be accessible to the OS.

- **Ready/Suspend → Ready:** When there are no ready processes in main memory, the OS will need to bring one in to continue execution. In addition, it might be the case that a process in the Ready/Suspend state has higher priority than any of the processes in the Ready state. In that case, the OS designer may dictate that it is more important to get at the higher-priority process than to minimize swapping.
- **Ready → Ready/Suspend:** Normally, the OS would prefer to suspend a blocked process rather than a ready one, because the ready process can now be executed, whereas the blocked process is taking up main memory space and cannot be executed. However, it may be necessary to suspend a ready process if that is the only way to free up a sufficiently large block of main memory. Also, the OS may choose to suspend a lower-priority ready process rather than a higher-priority blocked process if it believes that the blocked process will be ready soon.

Several other transitions that are worth considering are the following:

- **New → Ready/Suspend and New → Ready:** When a new process is created, it can either be added to the Ready queue or the Ready/Suspend queue. In either case, the OS must create a process control block and allocate an address space to the process. It might be preferable for the OS to perform these housekeeping duties at an early time, so that it can maintain a large pool of processes that are not blocked. With this strategy, there would often be insufficient room in main memory for a new process; hence the use of the (New → Ready/Suspend) transition. On the other hand, we could argue that a just-in-time philosophy of creating processes as late as possible reduces OS overhead and allows that OS to perform the process-creation duties at a time when the system is clogged with blocked processes anyway.
- **Blocked/Suspend → Blocked:** Inclusion of this transition may seem to be poor design. After all, if a process is not ready to execute and is not already in main memory, what is the point of bringing it in? But consider the following scenario: A process terminates, freeing up some main memory. There is a process in the (Blocked/Suspend) queue with a higher priority than any of the processes in the (Ready/Suspend) queue and the OS has reason to believe that the blocking event for that process will occur soon. Under these circumstances, it would seem reasonable to bring a blocked process into main memory in preference to a ready process.
- **Running → Ready/Suspend:** Normally, a running process is moved to the Ready state when its time allocation expires. If, however, the OS is preempting the process because a higher-priority process on the Blocked/Suspend queue has just become unblocked, the OS could move the running process directly to the (Ready/Suspend) queue and free some main memory.
- **Any State → Exit:** Typically, a process terminates while it is running, either because it has completed or because of some fatal fault condition. However, in some operating systems, a process may be terminated by the process that created it or when the parent process is itself terminated. If this is allowed, then a process in any state can be moved to the Exit state.

Table 3.3 Reasons for Process Suspension

Swapping	The OS needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS reason	The OS may suspend a background or utility process or a process that is suspected of causing a problem.
Interactive user request	A user may wish to suspend execution of a program for purposes of debugging or in connection with the use of a resource.
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time interval.
Parent process request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendants.

Other Uses of Suspension So far, we have equated the concept of a suspended process with that of a process that is not in main memory. A process that is not in main memory is not immediately available for execution, whether or not it is awaiting an event.

We can generalize the concept of a suspended process. Let us define a suspended process as having the following characteristics:

1. The process is not immediately available for execution.
2. The process may or may not be waiting on an event. If it is, this blocked condition is independent of the suspend condition, and occurrence of the blocking event does not enable the process to be executed immediately.
3. The process was placed in a suspended state by an agent: either itself, a parent process, or the OS, for the purpose of preventing its execution.
4. The process may not be removed from this state until the agent explicitly orders the removal.

Table 3.3 lists some reasons for the suspension of a process. One reason that we have discussed is to provide memory space either to bring in a Ready/Suspended process or to increase the memory allocated to other Ready processes. The OS may have other motivations for suspending a process. For example, an auditing or tracing process may be employed to monitor activity on the system; the process may be used to record the level of utilization of various resources (processor, memory, channels) and the rate of progress of the user processes in the system. The OS, under operator control, may turn this process on and off from time to time. If the OS detects or suspects a problem, it may suspend a process. One example of this is deadlock, which is discussed in Chapter 6. As another example, a problem is detected on a communications line, and the operator has the OS suspend the process that is using the line while some tests are run.

Another set of reasons concerns the actions of an interactive user. For example, if a user suspects a bug in the program, he or she may debug the program by suspending its execution, examining and modifying the program or data, and resuming execution. Or there may be a background process that is collecting trace or accounting statistics, which the user may wish to be able to turn on and off.

Timing considerations may also lead to a swapping decision. For example, if a process is to be activated periodically but is idle most of the time, then it should be swapped out between uses. A program that monitors utilization or user activity is an example.

Finally, a parent process may wish to suspend a descendent process. For example, process A may spawn process B to perform a file read. Subsequently, process B encounters an error in the file read procedure and reports this to process A. Process A suspends process B to investigate the cause.

In all of these cases, the activation of a suspended process is requested by the agent that initially requested the suspension.

3.3 PROCESS DESCRIPTION

The OS controls events within the computer system. It schedules and dispatches processes for execution by the processor, allocates resources to processes, and responds to requests by user processes for basic services. Fundamentally, we can think of the OS as that entity that manages the use of system resources by processes.

This concept is illustrated in Figure 3.10. In a multiprogramming environment, there are a number of processes ($P_1, \ldots, P_n$,) that have been created and exist in virtual memory. Each process, during the course of its execution, needs access to certain system resources, including the processor, I/O devices, and main memory. In the figure, process P_1 is running; at least part of the process is in main memory, and it has control of two I/O devices. Process P_2 is also in main memory but is blocked waiting for an I/O device allocated to P_1. Process P_n has been swapped out and is therefore suspended.

We explore the details of the management of these resources by the OS on behalf of the processes in later chapters. Here we are concerned with a more fundamental question: What information does the OS need to control processes and manage resources for them?

Operating System Control Structures

If the OS is to manage processes and resources, it must have information about the current status of each process and resource. The universal approach to providing this information is straightforward: The OS constructs and maintains tables of

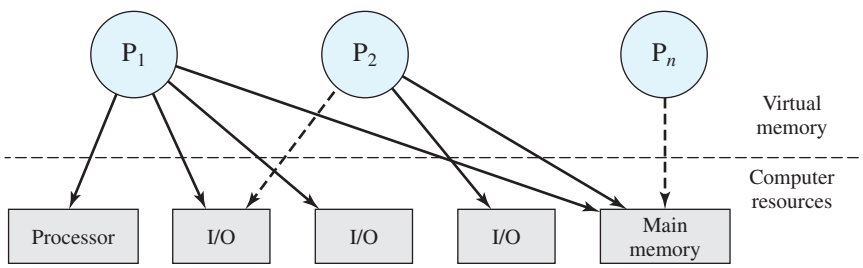


Figure 3.10 Processes and Resources (resource allocation at one snapshot in time)

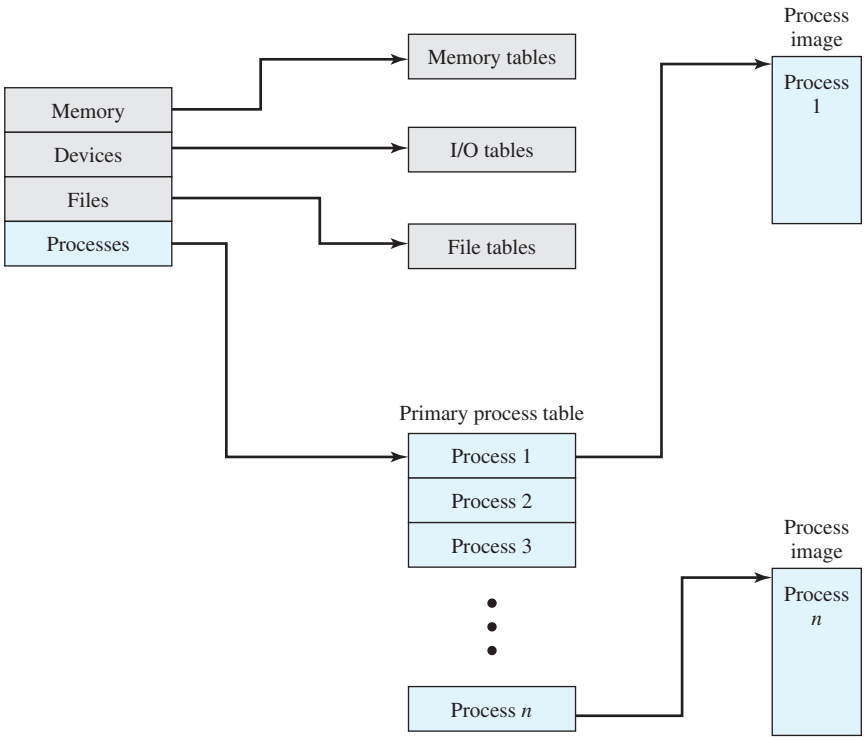


Figure 3.11 **General Structure of Operating System Control Tables**

information about each entity that it is managing. A general idea of the scope of this effort is indicated in Figure 3.11, which shows four different types of tables maintained by the OS: memory, I/O, file, and process. Although the details will differ from one OS to another, fundamentally, all operating systems maintain information in these four categories.

Memory tables are used to keep track of both main (real) and secondary (virtual) memory. Some of main memory is reserved for use by the OS; the remainder is available for use by processes. Processes are maintained on secondary memory using some sort of virtual memory or simple swapping mechanism. The memory tables must include the following information:

- The allocation of main memory to processes
- The allocation of secondary memory to processes
- Any protection attributes of blocks of main or virtual memory, such as which processes may access certain shared memory regions
- Any information needed to manage virtual memory

We examine the information structures for memory management in detail in Part Three.

I/O tables are used by the OS to manage the I/O devices and channels of the computer system. At any given time, an I/O device may be available or assigned to a particular process. If an I/O operation is in progress, the OS needs to know the status of the I/O operation and the location in main memory being used as the source or destination of the I/O transfer. I/O management is examined in Chapter 11.

The OS may also maintain **file tables**. These tables provide information about the existence of files, their location on secondary memory, their current status, and other attributes. Much, if not all, of this information may be maintained and used by a file management system, in which case the OS has little or no knowledge of files. In other operating systems, much of the detail of file management is managed by the OS itself. This topic is explored in Chapter 12.

Finally, the OS must maintain **process tables** to manage processes. The remainder of this section is devoted to an examination of the required process tables. Before proceeding to this discussion, two additional points should be made. First, although Figure 3.11 shows four distinct sets of tables, it should be clear that these tables must be linked or cross-referenced in some fashion. Memory, I/O, and files are managed on behalf of processes, so there must be some reference to these resources, directly or indirectly, in the process tables. The files referred to in the file tables are accessible via an I/O device and will, at some times, be in main or virtual memory. The tables themselves must be accessible by the OS and therefore are subject to memory management.

Second, how does the OS know to create the tables in the first place? Clearly, the OS must have some knowledge of the basic environment, such as how much main memory exists, what are the I/O devices and what are their identifiers, and so on. This is an issue of configuration. That is, when the OS is initialized, it must have access to some configuration data that define the basic environment, and these data must be created outside the OS, with human assistance or by some autoconfiguration software.

Process Control Structures

Consider what the OS must know if it is to manage and control a process. First, it must know where the process is located, and second, it must know the attributes of the process that are necessary for its management (e.g., process ID and process state).

Process Location Before we can deal with the questions of where a process is located or what its attributes are, we need to address an even more fundamental question: What is the physical manifestation of a process? At a minimum, a process must include a program or set of programs to be executed. Associated with these programs is a set of data locations for local and global variables and any defined constants. Thus, a process will consist of at least sufficient memory to hold the programs and data of that process. In addition, the execution of a program typically involves a stack (see Appendix 1B) that is used to keep track of procedure calls and parameter passing between procedures. Finally, each process has associated with it a number of attributes that are used by the OS for process control. Typically, the collection of attributes is referred to as a **process control**

Table 3.4 Typical Elements of a Process Image

User Data The modifiable part of the user space. May include program data, a user stack area, and programs that may be modified.
User Program The program to be executed.
Stack Each process has one or more last-in-first-out (LIFO) stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls.
Process Control Block Data needed by the OS to control the process (see Table 3.5).

block.[7] We can refer to this collection of program, data, stack, and attributes as the **process image** (Table 3.4).

The location of a process image will depend on the memory management scheme being used. In the simplest case, the process image is maintained as a contiguous, or continuous, block of memory. This block is maintained in secondary memory, usually disk. So that the OS can manage the process, at least a small portion of its image must be maintained in main memory. To execute the process, the entire process image must be loaded into main memory or at least virtual memory. Thus, the OS needs to know the location of each process on disk and, for each such process that is in main memory, the location of that process in main memory. We saw a slightly more complex variation on this scheme with the CTSS OS, in Chapter 2. With CTSS, when a process is swapped out, part of the process image may remain in main memory. Thus, the OS must keep track of which portions of the image of each process are still in main memory.

Modern operating systems presume paging hardware that allows noncontiguous physical memory to support partially resident processes.[8] At any given time, a portion of a process image may be in main memory, with the remainder in secondary memory.[9] Therefore, process tables maintained by the OS must show the location of each page of each process image.

Figure 3.11 depicts the structure of the location information in the following way. There is a primary process table with one entry for each process. Each entry contains, at least, a pointer to a process image. If the process image contains multiple blocks, this information is contained directly in the primary process table or is available by

[7]Other commonly used names for this data structure are task control block, process descriptor, and task descriptor.

[8]A brief overview of the concepts of pages, segments, and virtual memory is provided in the subsection on memory management in Section 2.3.

[9]This brief discussion slides over some details. In particular, in a system that uses virtual memory, all of the process image for an active process is always in secondary memory. When a portion of the image is loaded into main memory, it is copied rather than moved. Thus, the secondary memory retains a copy of all segments and/or pages. However, if the main memory portion of the image is modified, the secondary copy will be out of date until the main memory portion is copied back onto disk.

Table 3.5 Typical Elements of a Process Control Block

Process Identification

Identifiers
Numeric identifiers that may be stored with the process control block include

- Identifier of this process
- Identifier of the process that created this process (parent process)
- User identifier

Processor State Information

User-Visible Registers
A user-visible register is one that may be referenced by means of the machine language that the processor executes while in user mode. Typically, there are from 8 to 32 of these registers, although some RISC implementations have over 100.

Control and Status Registers
These are a variety of processor registers that are employed to control the operation of the processor. These include

- *Program counter:* Contains the address of the next instruction to be fetched
- *Condition codes:* Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow)
- *Status information:* Includes interrupt enabled/disabled flags, execution mode

Stack Pointers
Each process has one or more last-in-first-out (LIFO) system stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls. The stack pointer points to the top of the stack.

Process Control Information

Scheduling and State Information
This is information that is needed by the operating system to perform its scheduling function. Typical items of information:

- *Process state:* Defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, halted).
- *Priority:* One or more fields may be used to describe the scheduling priority of the process. In some systems, several values are required (e.g., default, current, highest-allowable).
- *Scheduling-related information:* This will depend on the scheduling algorithm used. Examples are the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running.
- *Event:* Identity of event the process is awaiting before it can be resumed.

Data Structuring
A process may be linked to other process in a queue, ring, or some other structure. For example, all processes in a waiting state for a particular priority level may be linked in a queue. A process may exhibit a parent-child (creator-created) relationship with another process. The process control block may contain pointers to other processes to support these structures.

Interprocess Communication
Various flags, signals, and messages may be associated with communication between two independent processes. Some or all of this information may be maintained in the process control block.

Process Privileges
Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed. In addition, privileges may apply to the use of system utilities and services.

Memory Management
This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.

Resource Ownership and Utilization
Resources controlled by the process may be indicated, such as opened files. A history of utilization of the processor or other resources may also be included; this information may be needed by the scheduler.

cross-reference to entries in memory tables. Of course, this depiction is generic; a particular OS will have its own way of organizing the location information.

Process Attributes A sophisticated multiprogramming system requires a great deal of information about each process. As was explained, this information can be considered to reside in a process control block. Different systems will organize this information in different ways, and several examples of this appear at the end of this chapter and the next. For now, let us simply explore the type of information that might be of use to an OS without considering in any detail how that information is organized.

Table 3.5 lists the typical categories of information required by the OS for each process. You may be somewhat surprised at the quantity of information required. As you gain a greater appreciation of the responsibilities of the OS, this list should appear more reasonable.

We can group the process control block information into three general categories:

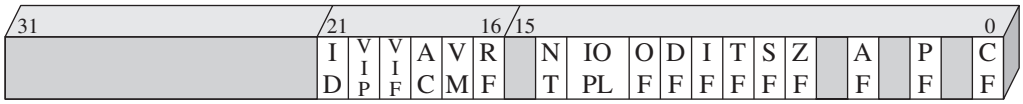
- Process identification
- Processor state information
- Process control information

With respect to **process identification**, in virtually all operating systems, each process is assigned a unique numeric identifier, which may simply be an index into the primary process table (Figure 3.11); otherwise there must be a mapping that allows the OS to locate the appropriate tables based on the process identifier. This identifier is useful in several ways. Many of the other tables controlled by the OS may use process identifiers to cross-reference process tables. For example, the memory tables may be organized so as to provide a map of main memory with an indication of which process is assigned to each region. Similar references will appear in I/O and file tables. When processes communicate with one another, the process identifier informs the OS of the destination of a particular communication. When processes are allowed to create other processes, identifiers indicate the parent and descendents of each process.

In addition to these process identifiers, a process may be assigned a user identifier that indicates the user responsible for the job.

Processor state information consists of the contents of processor registers. While a process is running, of course, the information is in the registers. When a process is interrupted, all of this register information must be saved so that it can be restored when the process resumes execution. The nature and number of registers involved depend on the design of the processor. Typically, the register set will include user-visible registers, control and status registers, and stack pointers. These are described in Chapter 1.

Of particular note, all processor designs include a register or set of registers, often known as the program status word (PSW), that contains status information. The PSW typically contain condition codes plus other status information. A good example of a processor status word is that on Pentium processors, referred to as the EFLAGS register (shown in Figure 3.12 and Table 3.6). This structure is used by any OS (including UNIX and Windows) running on a Pentium processor.



ID = Identification flag
VIP = Virtual interrupt pending
VIF = Virtual interrupt flag
AC = Alignment check
VM = Virtual 8086 mode
RF = Resume flag
NT = Nested task flag
IOPL = I/O privilege level
OF = Overflow flag
DF = Direction flag
IF = Interrupt enable flag
TF = Trap flag
SF = Sign flag
ZF = Zero flag
AF = Auxiliary carry flag
PF = Parity flag
CF = Carry flag

Figure 3.12 **Pentium II EFLAGS Register**

Table 3.6 Pentium EFLAGS Register Bits

Control Bits
AC (Alignment check) Set if a word or doubleword is addressed on a nonword or nondoubleword boundary.
ID (Identification flag) If this bit can be set and cleared, this processor supports the CPUID instruction. This instruction provides information about the vendor, family, and model.
RF (Resume flag) Allows the programmer to disable debug exceptions so that the instruction can be restarted after a debug exception without immediately causing another debug exception.
IOPL (I/O privilege level) When set, causes the processor to generate an exception on all accesses to I/O devices during protected mode operation.
DF (Direction flag) Determines whether string processing instructions increment or decrement the 16-bit half-registers SI and DI (for 16-bit operations) or the 32-bit registers ESI and EDI (for 32-bit operations).
IF (Interrupt enable flag) When set, the processor will recognize external interrupts.
TF (Trap flag) When set, causes an interrupt after the execution of each instruction. This is used for debugging.
Operating Mode Bits
NT (Nested task flag) Indicates that the current task is nested within another task in protected mode operation.
VM (Virtual 8086 mode) Allows the programmer to enable or disable virtual 8086 mode, which determines whether the processor runs as an 8086 machine.
VIP (Virtual interrupt pending) Used in virtual 8086 mode to indicate that one or more interrupts are awaiting service.
VIF (Virtual interrupt flag) Used in virtual 8086 mode instead of IF.

Condition Codes

AF (Auxiliary carry flag)
Represents carrying or borrowing between half-bytes of an 8-bit arithmetic or logic operation using the AL register.

CF (Carry flag)
Indicates carrying our or borrowing into the leftmost bit position following an arithmetic operation. Also modified by some of the shift and rotate operations.

OF (Overflow flag)
Indicates an arithmetic overflow after an addition or subtraction.

PF (Parity flag)
Parity of the result of an arithmetic or logic operation. 1 indicates even parity; 0 indicates odd parity.

SF (Sign flag)
Indicates the sign of the result of an arithmetic or logic operation.

ZF (Zero flag)
Indicates that the result of an arithmetic or logic operation is 0.

The third major category of information in the process control block can be called, for want of a better name, **process control information**. This is the additional information needed by the OS to control and coordinate the various active processes. The last part of Table 3.5 indicates the scope of this information. As we examine the details of operating system functionality in succeeding chapters, the need for the various items on this list should become clear.

Figure 3.13 suggests the structure of process images in virtual memory. Each process image consists of a process control block, a user stack, the private address space

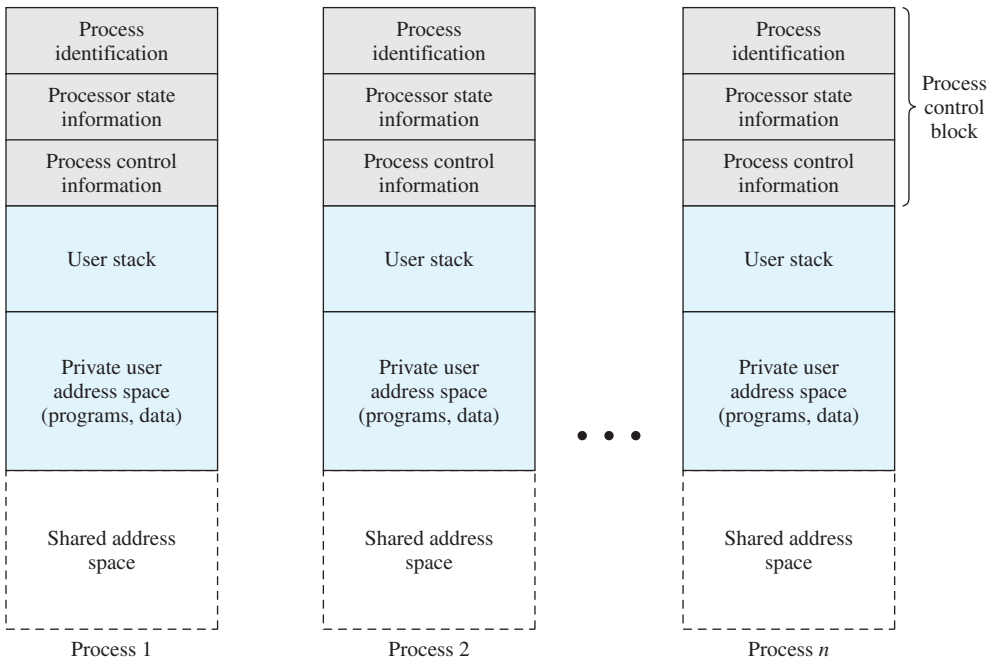


Figure 3.13 User Processes in Virtual Memory

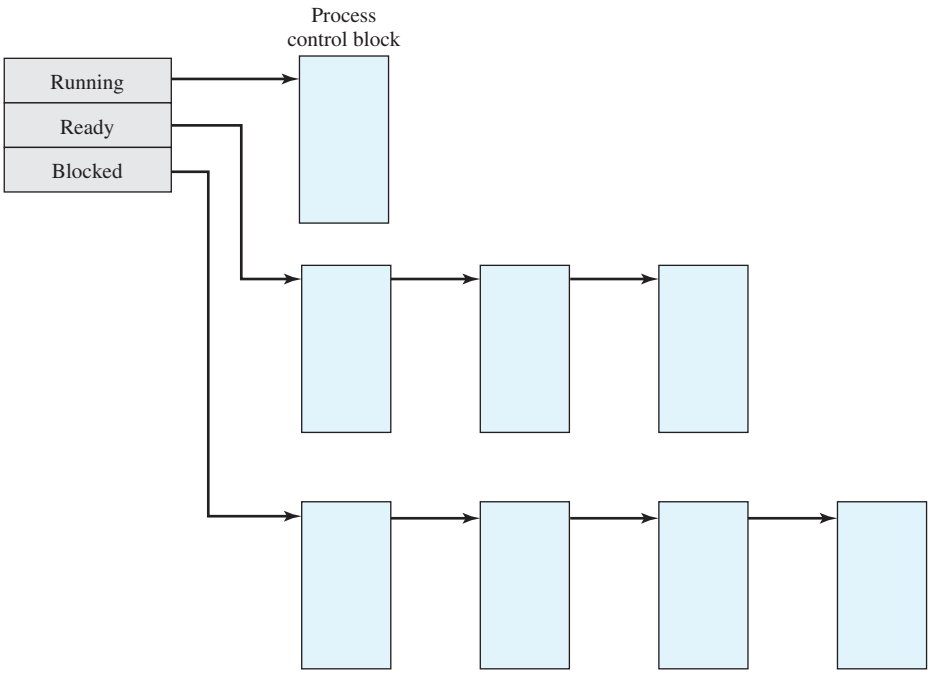


Figure 3.14 **Process List Structures**

of the process, and any other address space that the process shares with other processes. In the figure, each process image appears as a contiguous range of addresses. In an actual implementation, this may not be the case; it will depend on the memory management scheme and the way in which control structures are organized by the OS.

As indicated in Table 3.5, the process control block may contain structuring information, including pointers that allow the linking of process control blocks. Thus, the queues that were described in the preceding section could be implemented as linked lists of process control blocks. For example, the queuing structure of Figure 3.8a could be implemented as suggested in Figure 3.14.

The Role of the Process Control Block The process control block is the most important data structure in an OS. Each process control block contains all of the information about a process that is needed by the OS. The blocks are read and/or modified by virtually every module in the OS, including those involved with scheduling, resource allocation, interrupt processing, and performance monitoring and analysis. One can say that the set of process control blocks defines the state of the OS.

This brings up an important design issue. A number of routines within the OS will need access to information in process control blocks. The provision of direct access to these tables is not difficult. Each process is equipped with a unique ID, and this can be used as an index into a table of pointers to the process control blocks. The difficulty is not access but rather protection. Two problems present themselves:

- A bug in a single routine, such as an interrupt handler, could damage process control blocks, which could destroy the system's ability to manage the affected processes.

- A design change in the structure or semantics of the process control block could affect a number of modules in the OS.

These problems can be addressed by requiring all routines in the OS to go through a handler routine, the only job of which is to protect process control blocks, and which is the sole arbiter for reading and writing these blocks. The tradeoff in the use of such a routine involves performance issues and the degree to which the remainder of the system software can be trusted to be correct.

3.4 PROCESS CONTROL

Modes of Execution

Before continuing with our discussion of the way in which the OS manages processes, we need to distinguish between the mode of processor execution normally associated with the OS and that normally associated with user programs. Most processors support at least two modes of execution. Certain instructions can only be executed in the more-privileged mode. These would include reading or altering a control register, such as the program status word; primitive I/O instructions; and instructions that relate to memory management. In addition, certain regions of memory can only be accessed in the more-privileged mode.

The less-privileged mode is often referred to as the **user mode**, because user programs typically would execute in this mode. The more-privileged mode is referred to as the **system mode**, **control mode**, or **kernel mode**. This last term refers to the kernel of the OS, which is that portion of the OS that encompasses the important system functions. Table 3.7 lists the functions typically found in the kernel of an OS.

Table 3.7 Typical Functions of an Operating System Kernel

Process Management
• Process creation and termination
• Process scheduling and dispatching
• Process switching
• Process synchronization and support for interprocess communication
• Management of process control blocks
Memory Management
• Allocation of address space to processes
• Swapping
• Page and segment management
I/O Management
• Buffer management
• Allocation of I/O channels and devices to processes
Support Functions
• Interrupt handling
• Accounting
• Monitoring

The reason for using two modes should be clear. It is necessary to protect the OS and key operating system tables, such as process control blocks, from interference by user programs. In the kernel mode, the software has complete control of the processor and all its instructions, registers, and memory. This level of control is not necessary and for safety is not desirable for user programs.

Two questions arise: How does the processor know in which mode it is to be executing and how is the mode changed? Regarding the first question, typically there is a bit in the program status word (PSW) that indicates the mode of execution. This bit is changed in response to certain events. Typically, when a user makes a call to an operating system service or when an interrupt triggers execution of an operating system routine, the mode is set to the kernel mode and, upon return from the service to the user process, the mode is set to user mode. As an example, consider the Intel Itanium processor, which implements the 64-bit IA-64 architecture. The processor has a processor status register (psr) that includes a 2-bit cpl (current privilege level) field. Level 0 is the most privileged level, while level 3 is the least privileged level. Most operating systems, such as Linux, use level 0 for the kernel and one other level for user mode. When an interrupt occurs, the processor clears most of the bits in the psr, including the cpl field. This automatically sets the cpl to level 0. At the end of the interrupt-handling routine, the final instruction that is executed is irt (interrup return). This instruction causes the processor to restore the psr of the interrupted program, which restores the privilege level of that program. A similar sequence occurs when an application places a system call. For the Itanium, an application places a system call by placing the system call identifier and the system call arguments in a predefined area and then executing a special instruction that has the effect of interrupting execution at the user level and transferring control to the kernel.

Process Creation

In Section 3.2, we discussed the events that lead to the creation of a new process. Having discussed the data structures associated with a process, we are now in a position to describe briefly the steps involved in actually creating the process.

Once the OS decides, for whatever reason (Table 3.1), to create a new process, it can proceed as follows:

1. **Assign a unique process identifier to the new process.** At this time, a new entry is added to the primary process table, which contains one entry per process.
2. **Allocate space for the process.** This includes all elements of the process image. Thus, the OS must know how much space is needed for the private user address space (programs and data) and the user stack. These values can be assigned by default based on the type of process, or they can be set based on user request at job creation time. If a process is spawned by another process, the parent process can pass the needed values to the OS as part of the process-creation request. If any existing address space is to be shared by this new process, the appropriate linkages must be set up. Finally, space for a process control block must be allocated.
3. **Initialize the process control block.** The process identification portion contains the ID of this process plus other appropriate IDs, such as that of the parent process. The processor state information portion will typically be initialized with

most entries zero, except for the program counter (set to the program entry point) and system stack pointers (set to define the process stack boundaries). The process control information portion is initialized based on standard default values plus attributes that have been requested for this process. For example, the process state would typically be initialized to Ready or Ready/Suspend. The priority may be set by default to the lowest priority unless an explicit request is made for a higher priority. Initially, the process may own no resources (I/O devices, files) unless there is an explicit request for these or unless they are inherited from the parent.

4. **Set the appropriate linkages.** For example, if the OS maintains each scheduling queue as a linked list, then the new process must be put in the Ready or Ready/Suspend list.
5. **Create or expand other data structures.** For example, the OS may maintain an accounting file on each process to be used subsequently for billing and/or performance assessment purposes.

Process Switching

On the face of it, the function of process switching would seem to be straightforward. At some time, a running process is interrupted and the OS assigns another process to the Running state and turns control over to that process. However, several design issues are raised. First, what events trigger a process switch? Another issue is that we must recognize the distinction between mode switching and process switching. Finally, what must the OS do to the various data structures under its control to achieve a process switch?

When to Switch Processes A process switch may occur any time that the OS has gained control from the currently running process. Table 3.8 suggests the possible events that may give control to the OS.

First, let us consider system interrupts. Actually, we can distinguish, as many systems do, two kinds of system interrupts, one of which is simply referred to as an interrupt, and the other as a trap. The former is due to some sort of event that is external to and independent of the currently running process, such as the completion of an I/O operation. The latter relates to an error or exception condition generated within the currently running process, such as an illegal file access attempt. With an ordinary **interrupt**, control is first transferred to an interrupt handler, which does

Table 3.8 Mechanisms for Interrupting the Execution of a Process

Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction	Reaction to an asynchronous external event
Trap	Associated with the execution of the current instruction	Handling of an error or an exception condition
Supervisor call	Explicit request	Call to an operating system function

some basic housekeeping and then branches to an OS routine that is concerned with the particular type of interrupt that has occurred. Examples include the following:

- **Clock interrupt:** The OS determines whether the currently running process has been executing for the maximum allowable unit of time, referred to as a **time slice**. That is, a time slice is the maximum amount of time that a process can execute before being interrupted. If so, this process must be switched to a Ready state and another process dispatched.
- **I/O interrupt:** The OS determines what I/O action has occurred. If the I/O action constitutes an event for which one or more processes are waiting, then the OS moves all of the corresponding blocked processes to the Ready state (and Blocked/Suspend processes to the Ready/Suspend state). The OS must then decide whether to resume execution of the process currently in the Running state or to preempt that process for a higher-priority Ready process.
- **Memory fault:** The processor encounters a virtual memory address reference for a word that is not in main memory. The OS must bring in the block (page or segment) of memory containing the reference from secondary memory to main memory. After the I/O request is issued to bring in the block of memory, the process with the memory fault is placed in a blocked state; the OS then performs a process switch to resume execution of another process. After the desired block is brought into memory, that process is placed in the Ready state.

With a **trap**, the OS determines if the error or exception condition is fatal. If so, then the currently running process is moved to the Exit state and a process switch occurs. If not, then the action of the OS will depend on the nature of the error and the design of the OS. It may attempt some recovery procedure or simply notify the user. It may do a process switch or resume the currently running process.

Finally, the OS may be activated by a **supervisor call** from the program being executed. For example, a user process is running and an instruction is executed that requests an I/O operation, such as a file open. This call results in a transfer to a routine that is part of the operating system code. The use of a system call may place the user process in the Blocked state.

Mode Switching In Chapter 1, we discussed the inclusion of an interrupt stage as part of the instruction cycle. Recall that, in the interrupt stage, the processor checks to see if any interrupts are pending, indicated by the presence of an interrupt signal. If no interrupts are pending, the processor proceeds to the fetch stage and fetches the next instruction of the current program in the current process. If an interrupt is pending, the processor does the following:

1. It sets the program counter to the starting address of an interrupt handler program.
2. It switches from user mode to kernel mode so that the interrupt processing code may include privileged instructions.

The processor now proceeds to the fetch stage and fetches the first instruction of the interrupt handler program, which will service the interrupt. At this point, typically, the context of the process that has been interrupted is saved into that process control block of the interrupted program.

One question that may now occur to you is, What constitutes the context that is saved? The answer is that it must include any information that may be altered by the execution of the interrupt handler and that will be needed to resume the program that was interrupted. Thus, the portion of the process control block that was referred to as processor state information must be saved. This includes the program counter, other processor registers, and stack information.

Does anything else need to be done? That depends on what happens next. The interrupt handler is typically a short program that performs a few basic tasks related to an interrupt. For example, it resets the flag or indicator that signals the presence of an interrupt. It may send an acknowledgment to the entity that issued the interrupt, such as an I/O module. And it may do some basic housekeeping relating to the effects of the event that caused the interrupt. For example, if the interrupt relates to an I/O event, the interrupt handler will check for an error condition. If an error has occurred, the interrupt handler may send a signal to the process that originally requested the I/O operation. If the interrupt is by the clock, then the handler will hand control over to the dispatcher, which will want to pass control to another process because the time slice allotted to the currently running process has expired.

What about the other information in the process control block? If this interrupt is to be followed by a switch to another process, then some work will need to be done. However, in most operating systems, the occurrence of an interrupt does not necessarily mean a process switch. It is possible that, after the interrupt handler has executed, the currently running process will resume execution. In that case, all that is necessary is to save the processor state information when the interrupt occurs and restore that information when control is returned to the program that was running. Typically, the saving and restoring functions are performed in hardware.

Change of Process State It is clear, then, that the mode switch is a concept distinct from that of the process switch.[10] A mode switch may occur without changing the state of the process that is currently in the Running state. In that case, the context saving and subsequent restoral involve little overhead. However, if the currently running process is to be moved to another state (Ready, Blocked, etc.), then the OS must make substantial changes in its environment. The steps involved in a full process switch are as follows:

1. Save the context of the processor, including program counter and other registers.
2. Update the process control block of the process that is currently in the Running state. This includes changing the state of the process to one of the other states (Ready; Blocked; Ready/Suspend; or Exit). Other relevant fields must also be updated, including the reason for leaving the Running state and accounting information.
3. Move the process control block of this process to the appropriate queue (Ready; Blocked on Event *i*; Ready/Suspend).

[10] The term *context switch* is often found in OS literature and textbooks. Unfortunately, although most of the literature uses this term to mean what is here called a process switch, other sources use it to mean a mode switch or even a thread switch (defined in the next chapter). To avoid ambiguity, the term is not used in this book.

4. Select another process for execution; this topic is explored in Part Four.
5. Update the process control block of the process selected. This includes changing the state of this process to Running.
6. Update memory management data structures. This may be required, depending on how address translation is managed; this topic is explored in Part Three.
7. Restore the context of the processor to that which existed at the time the selected process was last switched out of the Running state, by loading in the previous values of the program counter and other registers.

Thus, the process switch, which involves a state change, requires more effort than a mode switch.

3.5 EXECUTION OF THE OPERATING SYSTEM

In Chapter 2, we pointed out two intriguing facts about operating systems:

- The OS functions in the same way as ordinary computer software in the sense that the OS is a set of programs executed by the processor.
- The OS frequently relinquishes control and depends on the processor to restore control to the OS.

If the OS is just a collection of programs and if it is executed by the processor just like any other program, is the OS a process? If so, how is it controlled? These interesting questions have inspired a number of design approaches. Figure 3.15 illustrates a range of approaches that are found in various contemporary operating systems.

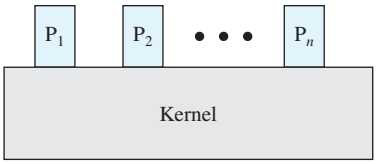
Nonprocess Kernel

One traditional approach, common on many older operating systems, is to execute the kernel of the OS outside of any process (Figure 3.15a). With this approach, when the currently running process is interrupted or issues a supervisor call, the mode context of this process is saved and control is passed to the kernel. The OS has its own region of memory to use and its own system stack for controlling procedure calls and returns. The OS can perform any desired functions and restore the context of the interrupted process, which causes execution to resume in the interrupted user process. Alternatively, the OS can complete the function of saving the environment of the process and proceed to schedule and dispatch another process. Whether this happens depends on the reason for the interruption and the circumstances at the time.

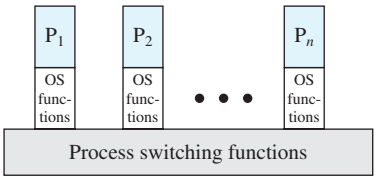
In any case, the key point here is that the concept of process is considered to apply only to user programs. The operating system code is executed as a separate entity that operates in privileged mode.

Execution within User Processes

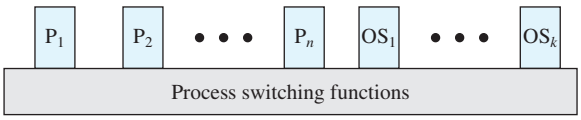
An alternative that is common with operating systems on smaller computers (PCs, workstations) is to execute virtually all OS software in the context of a user process.



(a) Separate kernel



(b) OS functions execute within user processes



(c) OS functions execute as separate processes

Figure 3.15 Relationship between Operating System and User Processes

The view is that the OS is primarily a collection of routines that the user calls to perform various functions, executed within the environment of the user's process. This is illustrated in Figure 3.15b. At any given point, the OS is managing *n* process images. Each image includes not only the regions illustrated in Figure 3.13, but also program, data, and stack areas for kernel programs.

Figure 3.16 suggests a typical process image structure for this strategy. A separate kernel stack is used to manage calls/returns while the process is in kernel mode. Operating system code and data are in the shared address space and are shared by all user processes.

When an interrupt, trap, or supervisor call occurs, the processor is placed in kernel mode and control is passed to the OS. To pass control from a user program to the OS, the mode context is saved and a mode switch takes place to an operating system routine. However, execution continues within the current user process. Thus, a process switch is not performed, just a mode switch within the same process.

If the OS, upon completion of its work, determines that the current process should continue to run, then a mode switch resumes the interrupted program within the current process. This is one of the key advantages of this approach: A user program has been interrupted to employ some operating system routine, and then resumed, and all of this has occurred without incurring the penalty of two process switches. If, however, it is determined that a process switch is to occur rather than returning to the previously executing program, then control is passed to a process-switching routine. This routine may or may not execute in the current process,

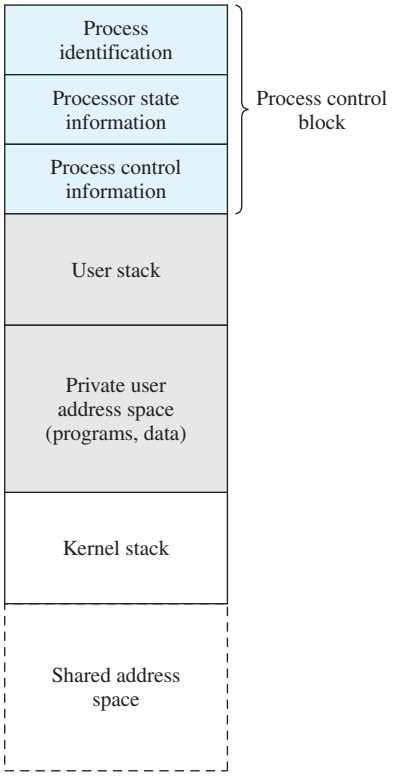


Figure 3.16 **Process Image: Operating System Executes within User Space**

depending on system design. At some point, however, the current process has to be placed in a nonrunning state and another process designated as the running process. During this phase, it is logically most convenient to view execution as taking place outside of all processes.

In a way, this view of the OS is remarkable. Simply put, at certain points in time, a process will save its state information, choose another process to run from among those that are ready, and relinquish control to that process. The reason this is not an arbitrary and indeed chaotic situation is that during the critical time, the code that is executed in the user process is shared operating system code and not user code. Because of the concept of user mode and kernel mode, the user cannot tamper with or interfere with the operating system routines, even though they are executing in the user's process environment. This further reminds us that there is a distinction between the concepts of process and program and that the relationship between the two is not one to one. Within a process, both a user program and operating system programs may execute, and the operating system programs that execute in the various user processes are identical.

Process-Based Operating System

Another alternative, illustrated in Figure 3.15c, is to implement the OS as a collection of system processes. As in the other options, the software that is part of the kernel executes in a kernel mode. In this case, however, major kernel functions are organized as separate processes. Again, there may be a small amount of process-switching code that is executed outside of any process.

This approach has several advantages. It imposes a program design discipline that encourages the use of a modular OS with minimal, clean interfaces between the modules. In addition, some noncritical operating system functions are conveniently implemented as separate processes. For example, we mentioned earlier a monitor program that records the level of utilization of various resources (processor, memory, channels) and the rate of progress of the user processes in the system. Because this program does not provide a particular service to any active process, it can only be invoked by the OS. As a process, the function can run at an assigned priority level and be interleaved with other processes under dispatcher control. Finally, implementing the OS as a set of processes is useful in a multiprocessor or multicomputer environment, in which some of the operating system services can be shipped out to dedicated processors, improving performance.

3.6 SECURITY ISSUES

An OS associates a set of privileges with each process. These privileges dictate what resources the process may access, including regions of memory, files, privileged system instructions, and so on. Typically, a process that executes on behalf of a user has the privileges that the OS recognizes for that user. A system or utility process may have privileges assigned at configuration time.

On a typical system, the highest level of privilege is referred to as administrator, supervisor, or root, access.[11] Root access provides access to all the functions and services of the operating system. With root access, a process has complete control of the system and can add or changes programs and files, monitor other processes, send and receive network traffic, and alter privileges.

A key security issue in the design of any OS is to prevent, or at least detect, attempts by a user or a piece of malicious software (malware) from gaining unauthorized privileges on the system and, in particular, from gaining root access. In this section, we briefly summarize the threats and countermeasures related to this security issue. Part Seven provides more detail.

System Access Threats

System access threats fall into two general categories: intruders and malicious software.

[11]On UNIX systems, the administrator, or *superuser*, account is called root; hence the term *root access*.

Intruders One of the most common threats to security is the intruder (the other is viruses), often referred to as a hacker or cracker. In an important early study of intrusion, Anderson [ANDE80] identified three classes of intruders:

- **Masquerader:** An individual who is not authorized to use the computer and who penetrates a system's access controls to exploit a legitimate user's account
- **Misfeasor:** A legitimate user who accesses data, programs, or resources for which such access is not authorized, or who is authorized for such access but misuses his or her privileges
- **Clandestine user:** An individual who seizes supervisory control of the system and uses this control to evade auditing and access controls or to suppress audit collection

The masquerader is likely to be an outsider; the misfeasor generally is an insider; and the clandestine user can be either an outsider or an insider.

Intruder attacks range from the benign to the serious. At the benign end of the scale, there are many people who simply wish to explore internets and see what is out there. At the serious end are individuals who are attempting to read privileged data, perform unauthorized modifications to data, or disrupt the system.

The objective of the intruder is to gain access to a system or to increase the range of privileges accessible on a system. Most initial attacks use system or software vulnerabilities that allow a user to execute code that opens a back door into the system. Intruders can get access to a system by exploiting attacks such as buffer overflows on a program that runs with certain privileges. We introduce buffer overflow attacks in Chapter 7.

Alternatively, the intruder attempts to acquire information that should have been protected. In some cases, this information is in the form of a user password. With knowledge of some other user's password, an intruder can log in to a system and exercise all the privileges accorded to the legitimate user.

Malicious Software Perhaps the most sophisticated types of threats to computer systems are presented by programs that exploit vulnerabilities in computing systems. Such threats are referred to as **malicious software**, or **malware**. In this context, we are concerned with threats to application programs as well as utility programs, such as editors and compilers, and kernel-level programs.

Malicious software can be divided into two categories: those that need a host program, and those that are independent. The former, referred to as **parasitic**, are essentially fragments of programs that cannot exist independently of some actual application program, utility, or system program. Viruses, logic bombs, and backdoors are examples. The latter are self-contained programs that can be scheduled and run by the operating system. Worms and bot programs are examples.

We can also differentiate between those software threats that do not replicate and those that do. The former are programs or fragments of programs that are activated by a trigger. Examples are logic bombs, backdoors, and bot programs. The latter consist of either a program fragment or an independent program that, when executed, may produce one or more copies of itself to be activated later on the same system or some other system. Viruses and worms are examples.

Malicious software can be relatively harmless or may perform one or more of a number of harmful actions, including destroying files and data in main memory, bypassing controls to gain privileged access, and providing a means for intruders to bypass access controls.

Countermeasures

Intrusion Detection RFC 2828 (*Internet Security Glossary*) defines intrusion detection as follows: A security service that monitors and analyzes system events for the purpose of finding, and providing real-time or near-real-time warning of, attempts to access system resources in an unauthorized manner.

Intrusion detection systems (IDSs) can be classified as follows:

- **Host-based IDS:** Monitors the characteristics of a single host and the events occurring within that host for suspicious activity
- **Network-based IDS:** Monitors network traffic for particular network segments or devices and analyzes network, transport, and application protocols to identify suspicious activity

An IDS comprises three logical components:

- **Sensors:** Sensors are responsible for collecting data. The input for a sensor may be any part of a system that could contain evidence of an intrusion. Types of input to a sensor include network packets, log files, and system call traces. Sensors collect and forward this information to the analyzer.
- **Analyzers:** Analyzers receive input from one or more sensors or from other analyzers. The analyzer is responsible for determining if an intrusion has occurred. The output of this component is an indication that an intrusion has occurred. The output may include evidence supporting the conclusion that an intrusion occurred. The analyzer may provide guidance about what actions to take as a result of the intrusion.
- **User interface:** The user interface to an IDS enables a user to view output from the system or control the behavior of the system. In some systems, the user interface may equate to a manager, director, or console component.

Intrusion detection systems are typically designed to detect human intruder behavior as well as malicious software behavior.

Authentication In most computer security contexts, user authentication is the fundamental building block and the primary line of defense. User authentication is the basis for most types of access control and for user accountability. RFC 2828 defines user authentication as follows:

> The process of verifying an identity claimed by or for a system entity. An authentication process consists of two steps:
>
> - **Identification step:** Presenting an identifier to the security system. (Identifiers should be assigned carefully, because authenticated identities are the basis for other security services, such as access control service.)
> - **Verification step:** Presenting or generating authentication information that corroborates the binding between the entity and the identifier.

For example, user Alice Toklas could have the user identifier ABTOKLAS. This information needs to be stored on any server or computer system that Alice wishes to use and could be known to system administrators and other users. A typical item of authentication information associated with this user ID is a password, which is kept secret (known only to Alice and to the system). If no one is able to obtain or guess Alice's password, then the combination of Alice's user ID and password enables administrators to set up Alice's access permissions and audit her activity. Because Alice's ID is not secret, system users can send her e-mail, but because her password is secret, no one can pretend to be Alice.

In essence, identification is the means by which a user provides a claimed identity to the system; user authentication is the means of establishing the validity of the claim.

There are four general means of authenticating a user's identity, which can be used alone or in combination:

- **Something the individual knows:** Examples include a password, a personal identification number (PIN), or answers to a prearranged set of questions.
- **Something the individual possesses:** Examples include electronic keycards, smart cards, and physical keys. This type of authenticator is referred to as a *token*.
- **Something the individual is (static biometrics):** Examples include recognition by fingerprint, retina, and face.
- **Something the individual does (dynamic biometrics):** Examples include recognition by voice pattern, handwriting characteristics, and typing rhythm.

All of these methods, properly implemented and used, can provide secure user authentication. However, each method has problems. An adversary may be able to guess or steal a password. Similarly, an adversary may be able to forge or steal a token. A user may forget a password or lose a token. Further, there is a significant administrative overhead for managing password and token information on systems and securing such information on systems. With respect to biometric authenticators, there are a variety of problems, including dealing with false positives and false negatives, user acceptance, cost, and convenience.

Access Control Access control implements a security policy that specifies who or what (e.g., in the case of a process) may have access to each specific system resource and the type of access that is permitted in each instance.

An access control mechanism mediates between a user (or a process executing on behalf of a user) and system resources, such as applications, operating systems, firewalls, routers, files, and databases. The system must first authenticate a user seeking access. Typically, the authentication function determines whether the user is permitted to access the system at all. Then the access control function determines if the specific requested access by this user is permitted. A security administrator maintains an authorization database that specifies what type of access to which resources is allowed for this user. The access control function consults this database to determine whether to grant access. An auditing function monitors and keeps a record of user accesses to system resources.

Firewalls Firewalls can be an effective means of protecting a local system or network of systems from network-based security threats while at the same time affording access

to the outside world via wide area networks and the Internet. Traditionally, a firewall is a dedicated computer that interfaces with computers outside a network and has special security precautions built into it in order to protect sensitive files on computers within the network. It is used to service outside network, especially Internet, connections and dial-in lines. Personal firewalls that are implemented in hardware or software, and associated with a single workstation or PC, are also common.

[BELL94] lists the following design goals for a firewall:

1. All traffic from inside to outside, and vice versa, must pass through the firewall. This is achieved by physically blocking all access to the local network except via the firewall. Various configurations are possible, as explained later in this chapter.
2. Only authorized traffic, as defined by the local security policy, will be allowed to pass. Various types of firewalls are used, which implement various types of security policies.
3. The firewall itself is immune to penetration. This implies the use of a hardened system with a secured operating system. Trusted computer systems are suitable for hosting a firewall and often required in government applications.

3.7 UNIX SVR4 PROCESS MANAGEMENT

UNIX System V makes use of a simple but powerful process facility that is highly visible to the user. UNIX follows the model of Figure 3.15b, in which most of the OS executes within the environment of a user process. UNIX uses two categories of processes: system processes and user processes. System processes run in kernel mode and execute operating system code to perform administrative and housekeeping functions, such as allocation of memory and process swapping. User processes operate in user mode to execute user programs and utilities and in kernel mode to execute instructions that belong to the kernel. A user process enters kernel mode by issuing a system call, when an exception (fault) is generated, or when an interrupt occurs.

Process States

A total of nine process states are recognized by the UNIX SVR4 operating system; these are listed in Table 3.9 and a state transition diagram is shown in Figure 3.17 (based on figure in [BACH86]). This figure is similar to Figure 3.9b, with the two UNIX sleeping states corresponding to the two blocked states. The differences are as follows:

- UNIX employs two Running states to indicate whether the process is executing in user mode or kernel mode.
- A distinction is made between the two states: (Ready to Run, in Memory) and (Preempted). These are essentially the same state, as indicated by the dotted line joining them. The distinction is made to emphasize the way in which the preempted state is entered. When a process is running in kernel mode (as a result of a supervisor call, clock interrupt, or I/O interrupt), there will come a time when the kernel

Table 3.9 UNIX Process States

User Running	Executing in user mode.
Kernel Running	Executing in kernel mode.
Ready to Run, in Memory	Ready to run as soon as the kernel schedules it.
Asleep in Memory	Unable to execute until an event occurs; process is in main memory (a blocked state).
Ready to Run, Swapped	Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute.
Sleeping, Swapped	The process is awaiting an event and has been swapped to secondary storage (a blocked state).
Preempted	Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.
Created	Process is newly created and not yet ready to run.
Zombie	Process no longer exists, but it leaves a record for its parent process to collect.

has completed its work and is ready to return control to the user program. At this point, the kernel may decide to preempt the current process in favor of one that is ready and of higher priority. In that case, the current process moves to the preempted state. However, for purposes of dispatching, those processes in the preempted state and those in the Ready to Run, in Memory state form one queue.

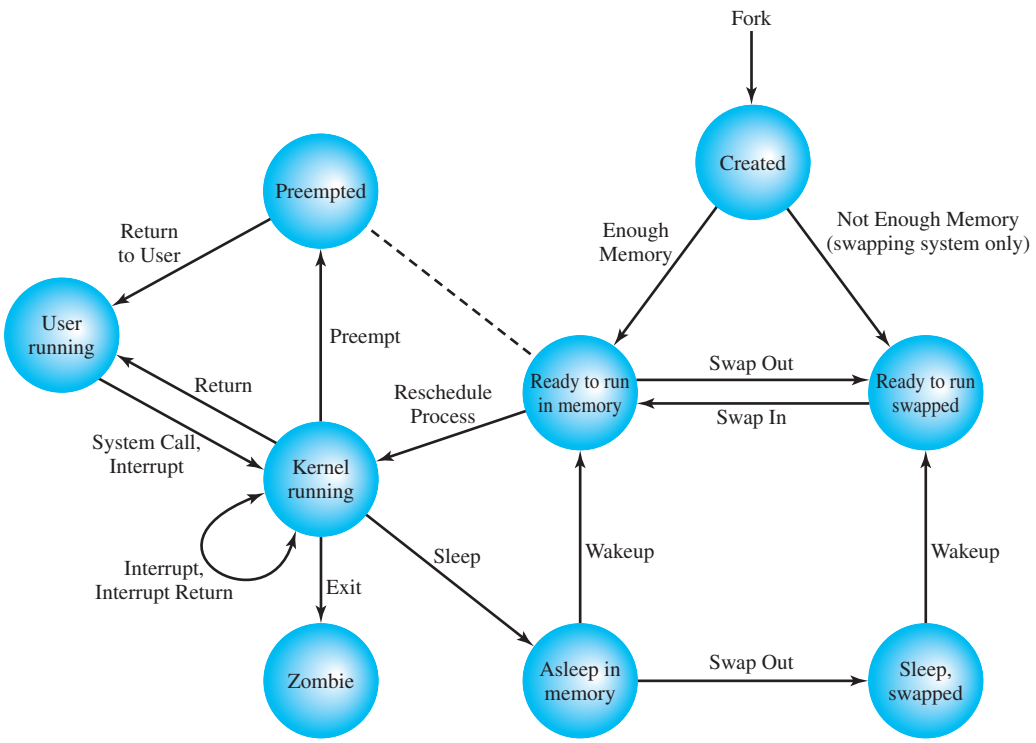


Figure 3.17 **UNIX Process State Transition Diagram**

Preemption can only occur when a process is about to move from kernel mode to user mode. While a process is running in kernel mode, it may not be preempted. This makes UNIX unsuitable for real-time processing. Chapter 10 discusses the requirements for real-time processing.

Two processes are unique in UNIX. Process 0 is a special process that is created when the system boots; in effect, it is predefined as a data structure loaded at boot time. It is the swapper process. In addition, process 0 spawns process 1, referred to as the init process; all other processes in the system have process 1 as an ancestor. When a new interactive user logs onto the system, it is process 1 that creates a user process for that user. Subsequently, the user process can create child processes in a branching tree, so that any particular application can consist of a number of related processes.

Process Description

A process in UNIX is a rather complex set of data structures that provide the OS with all of the information necessary to manage and dispatch processes. Table 3.10 summarizes the elements of the process image, which are organized into three parts: user-level context, register context, and system-level context.

The **user-level context** contains the basic elements of a user's program and can be generated directly from a compiled object file. The user's program is separated

Table 3.10 UNIX Process Image

User-Level Context	
Process text	Executable machine instructions of the program
Process data	Data accessible by the program of this process
User stack	Contains the arguments, local variables, and pointers for functions executing in user mode
Shared memory	Memory shared with other processes, used for interprocess communication
Register Context	
Program counter	Address of next instruction to be executed; may be in kernel or user memory space of this process
Processor status register	Contains the hardware status at the time of preemption; contents and format are hardware dependent
Stack pointer	Points to the top of the kernel or user stack, depending on the mode of operation at the time or preemption
General-purpose registers	Hardware dependent
System-Level Context	
Process table entry	Defines state of a process; this information is always accessible to the operating system
U (user) area	Process control information that needs to be accessed only in the context of the process
Per process region table	Defines the mapping from virtual to physical addresses; also contains a permission field that indicates the type of access allowed the process: read-only, read-write, or read-execute
Kernel stack	Contains the stack frame of kernel procedures as the process executes in kernel mode

into text and data areas; the text area is read-only and is intended to hold the program's instructions. While the process is executing, the processor uses the user stack area for procedure calls and returns and parameter passing. The shared memory area is a data area that is shared with other processes. There is only one physical copy of a shared memory area, but, by the use of virtual memory, it appears to each sharing process that the shared memory region is in its address space. When a process is not running, the processor status information is stored in the **register context** area.

The **system-level context** contains the remaining information that the OS needs to manage the process. It consists of a static part, which is fixed in size and stays with a process throughout its lifetime, and a dynamic part, which varies in size through the life of the process. One element of the static part is the process table entry. This is actually part of the process table maintained by the OS, with one entry per process. The process table entry contains process control information that is accessible to the kernel at all times; hence, in a virtual memory system, all process table entries are maintained in main memory. Table 3.11 lists the contents of a process table entry. The user area, or U area, contains additional process control information that is needed by the kernel when it is executing in the context of this process; it is also used when paging processes to and from memory. Table 3.12 shows the contents of this table.

The distinction between the process table entry and the U area reflects the fact that the UNIX kernel always executes in the context of some process. Much of the time, the kernel will be dealing with the concerns of that process. However, some of the time, such as when the kernel is performing a scheduling algorithm preparatory to dispatching another process, it will need access to information about other

Table 3.11 UNIX Process Table Entry

Process status	Current state of process.
Pointers	To U area and process memory area (text, data, stack).
Process size	Enables the operating system to know how much space to allocate the process.
User identifiers	The **real user ID** identifies the user who is responsible for the running process. The **effective user ID** may be used by a process to gain temporary privileges associated with a particular program; while that program is being executed as part of the process, the process operates with the effective user ID.
Process identifiers	ID of this process; ID of parent process. These are set up when the process enters the Created state during the fork system call.
Event descriptor	Valid when a process is in a sleeping state; when the event occurs, the process is transferred to a ready-to-run state.
Priority	Used for process scheduling.
Signal	Enumerates signals sent to a process but not yet handled.
Timers	Include process execution time, kernel resource utilization, and user-set timer used to send alarm signal to a process.
P_link	Pointer to the next link in the ready queue (valid if process is ready to execute).
Memory status	Indicates whether process image is in main memory or swapped out. If it is in memory, this field also indicates whether it may be swapped out or is temporarily locked into main memory.

Table 3.12 UNIX U Area

Process table pointer	Indicates entry that corresponds to the U area.
User identifiers	Real and effective user IDs. Used to determine user privileges.
Timers	Record time that the process (and its descendants) spent executing in user mode and in kernel mode.
Signal-handler array	For each type of signal defined in the system, indicates how the process will react to receipt of that signal (exit, ignore, execute specified user function).
Control terminal	Indicates login terminal for this process, if one exists.
Error field	Records errors encountered during a system call.
Return value	Contains the result of system calls.
I/O parameters	Describe the amount of data to transfer, the address of the source (or target) data array in user space, and file offsets for I/O.
File parameters	Current directory and current root describe the file system environment of the process.
User file descriptor table	Records the files the process has open.
Limit fields	Restrict the size of the process and the size of a file it can write.
Permission modes fields	Mask mode settings on files the process creates.

processes. The information in a process table can be accessed when the given process is not the current one.

The third static portion of the system-level context is the per process region table, which is used by the memory management system. Finally, the kernel stack is the dynamic portion of the system-level context. This stack is used when the process is executing in kernel mode and contains the information that must be saved and restored as procedure calls and interrupts occur.

Process Control

Process creation in UNIX is made by means of the kernel system call, `fork( )`. When a process issues a fork request, the OS performs the following functions [BACH86]:

1. It allocates a slot in the process table for the new process.
2. It assigns a unique process ID to the child process.
3. It makes a copy of the process image of the parent, with the exception of any shared memory.
4. It increments counters for any files owned by the parent, to reflect that an additional process now also owns those files.
5. It assigns the child process to the Ready to Run state.
6. It returns the ID number of the child to the parent process, and a 0 value to the child process.

All of this work is accomplished in kernel mode in the parent process. When the kernel has completed these functions it can do one of the following, as part of the dispatcher routine:

- Stay in the parent process. Control returns to user mode at the point of the fork call of the parent.
- Transfer control to the child process. The child process begins executing at the same point in the code as the parent, namely at the return from the fork call.
- Transfer control to another process. Both parent and child are left in the Ready to Run state.

It is perhaps difficult to visualize this method of process creation because both parent and child are executing the same passage of code. The difference is this: When the return from the fork occurs, the return parameter is tested. If the value is zero, then this is the child process, and a branch can be executed to the appropriate user program to continue execution. If the value is nonzero, then this is the parent process, and the main line of execution can continue.

3.8 SUMMARY

The most fundamental concept in a modern OS is the process. The principal function of the OS is to create, manage, and terminate processes. While processes are active, the OS must see that each is allocated time for execution by the processor, coordinate their activities, manage conflicting demands, and allocate system resources to processes.

To perform its process management functions, the OS maintains a description of each process, or process image, which includes the address space within which the process executes, and a process control block. The latter contains all of the information that is required by the OS to manage the process, including its current state, resources allocated to it, priority, and other relevant data.

During its lifetime, a process moves among a number of states. The most important of these are Ready, Running, and Blocked. A ready process is one that is not currently executing but that is ready to be executed as soon as the OS dispatches it. The running process is that process that is currently being executed by the processor. In a multiple-processor system, more than one process can be in this state. A blocked process is waiting for the completion of some event, such as an I/O operation.

A running process is interrupted either by an interrupt, which is an event that occurs outside the process and that is recognized by the processor, or by executing a supervisor call to the OS. In either case, the processor performs a mode switch, transferring control to an operating system routine. The OS, after it has completed necessary work, may resume the interrupted process or switch to some other process.

3.9 RECOMMENDED READING

Good descriptions of UNIX process management are found in [GOOD94] and [GRAY97]. [NEHM75] is an interesting discussion of process states and the operating system primitives needed for process dispatching.

GOOD94 Goodheart, B., and Cox, J. *The Magic Garden Explained: The Internals of UNIX System V Release 4.* Englewood Cliffs, NJ: Prentice Hall, 1994.

GRAY97 Gray, J. *Interprocess Communications in UNIX: The Nooks and Crannies.* Upper Saddle River, NJ: Prentice Hall, 1997.

NEHM75 Nehmer, J. "Dispatcher Primitives for the Construction of Operating System Kernels." *Acta Informatica*, vol 5, 1975.

3.10 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

blocked state	privileged mode	round robin
child process	process	running state
exit state	process control block	suspend state
interrupt	process image	swapping
kernel mode	process switch	system mode
mode switch	program status word	task
new state	ready state	trace
parent process		trap
preempt		user mode

Review Questions

3.1 What is an instruction trace?
3.2 What common events lead to the creation of a process?
3.3 For the processing model of Figure 3.6, briefly define each state.
3.4 What does it mean to preempt a process?
3.5 What is swapping and what is its purpose?
3.6 Why does Figure 3.9b have two blocked states?
3.7 List four characteristics of a suspended process.
3.8 For what types of entities does the OS maintain tables of information for management purposes?
3.9 List three general categories of information in a process control block.
3.10 Why are two modes (user and kernel) needed?
3.11 What are the steps performed by an OS to create a new process?
3.12 What is the difference between an interrupt and a trap?

3.13 Give three examples of an interrupt.

3.14 What is the difference between a mode switch and a process switch?

Problems

3.1 Name five major activities of an OS with respect to process management, and briefly describe why each is required.

3.2 Consider a computer with *N* processors in a multiprocessor configuration.

a. How many processes can be in each of the Ready, Running, and Blocked states at one time?

b. What is the minimum number of processes that can be in each of the Ready, Running, and Blocked states at one time?

3.3 Figure 3.9b contains seven states. In principle, one could draw a transition between any two states, for a total of 42 different transitions.

a. List all of the possible transitions and give an example of what could cause each transition.

b. List all of the impossible transitions and explain why.

3.4 In [PINK89], the following states are defined for processes: Execute (running), Active (ready), Blocked, and Suspend. A process is blocked if it is waiting for permission to use a resource, and it is suspended if it is waiting for an operation to be completed on a resource it has already acquired. In many operating systems, these two states are lumped together as the blocked state, and the suspended state has the definition we have used in this chapter. Compare the relative merits of the two sets of definitions.

3.5 For the seven-state process model of Figure 3.9b, draw a queuing diagram similar to that of Figure 3.8b.

3.6 Consider the state transition diagram of Figure 3.9b. Suppose that it is time for the OS to dispatch a process and that there are processes in both the Ready state and the Ready/Suspend state, and that at least one process in the Ready/Suspend state has higher scheduling priority than any of the processes in the Ready state. Two extreme policies are as follows: (1) Always dispatch from a process in the Ready state, to minimize swapping, and (2) always give preference to the highest-priority process, even though that may mean swapping when swapping is not necessary. Suggest an intermediate policy that tries to balance the concerns of priority and performance.

3.7 Table 3.13 shows the process states for the VAX/VMS operating system.

a. Can you provide a justification for the existence of so many distinct wait states?

b. Why do the following states not have resident and swapped-out versions: Page Fault Wait, Collided Page Wait, Common Event Wait, Free Page Wait, and Resource Wait?

c. Draw the state transition diagram and indicate the action or occurrence that causes each transition.

3.8 The VAX/VMS operating system makes use of four processor access modes to facilitate the protection and sharing of system resources among processes. The access mode determines

- **Instruction execution privileges:** What instructions the processor may execute
- **Memory access privileges:** Which locations in virtual memory the current instruction may access

The four modes are as follows:

- **Kernel:** Executes the kernel of the VMS operating system, which includes memory management, interrupt handling, and I/O operations
- **Executive:** Executes many of the OS service calls, including file and record (disk and tape) management routines
- **Supervisor:** Executes other OS services, such as responses to user commands
- **User:** Executes user programs, plus utilities such as compilers, editors, linkers, and debuggers

Table 3.13 VAX/VMS Process States

Process State	Process Condition
Currently Executing	Running process.
Computable (resident)	Ready and resident in main memory.
Computable (outswapped)	Ready, but swapped out of main memory.
Page Fault Wait	Process has referenced a page not in main memory and must wait for the page to be read in.
Collided Page Wait	Process has referenced a shared page that is the cause of an existing page fault wait in another process, or a private page that is in the process of being read in or written out.
Common Event Wait	Waiting for shared event flag (event flags are single-bit interprocess signaling mechanisms).
Free Page Wait	Waiting for a free page in main memory to be added to the collection of pages in main memory devoted to this process (the working set of the process).
Hibernate Wait (resident)	Process puts itself in a wait state.
Hibernate Wait (outswapped)	Hibernating process is swapped out of main memory.
Local Event Wait (resident)	Process in main memory and waiting for local event flag (usually I/O completion).
Local Event Wait (outswapped)	Process in local event wait is swapped out of main memory.
Suspended Wait (resident)	Process is put into a wait state by another process.
Suspended Wait (outswapped)	Suspended process is swapped out of main memory.
Resource Wait	Process waiting for miscellaneous system resource

A process executing in a less-privileged mode often needs to call a procedure that executes in a more-privileged mode; for example, a user program requires an operating system service. This call is achieved by using a change-mode (CHM) instruction, which causes an interrupt that transfers control to a routine at the new access mode. A return is made by executing the REI (return from exception or interrupt) instruction.

a. A number of operating systems have two modes, kernel and user. What are the advantages and disadvantages of providing four modes instead of two?

b. Can you make a case for even more than four modes?

3.9 The VMS scheme discussed in the preceding problem is often referred to as a ring protection structure, as illustrated in Figure 3.18. Indeed, the simple kernel/user scheme, as described in Section 3.3, is a two-ring structure. [SILB04] points out a problem with this approach:

> The main disadvantage of the ring (hierarchical) structure is that it does not allow us to enforce the need-to-know principle. In particular, if an object must be accessible in domain D_j but not accessible in domain D_i , then we must have $j < i$. But this means that every segment accessible in D_i is also accessible in D_j.

a. Explain clearly what the problem is that is referred to in the preceding quote.

b. Suggest a way that a ring-structured OS can deal with this problem.

3.10 Figure 3.8b suggests that a process can only be in one Event queue at a time.

a. Is it possible that you would want to allow a process to wait on more than one event at the same time? Provide an example.

b. In that case, how would you modify the queuing structure of the figure to support this new feature?

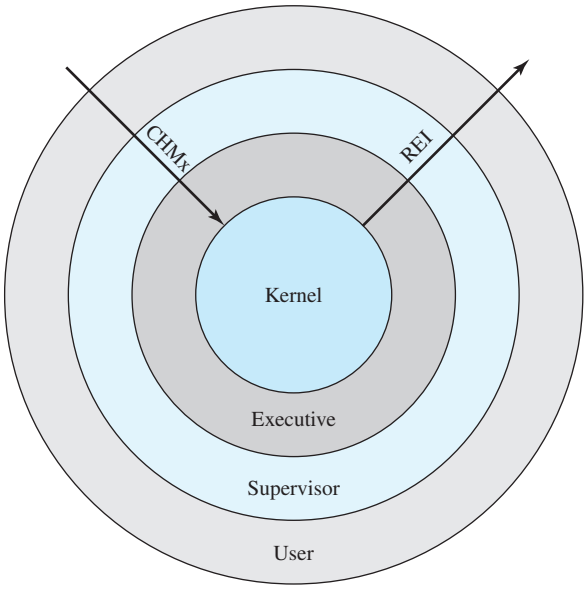


Figure 3.18 VAX/VMS Access Modes

3.11 In a number of early computers, an interrupt caused the register values to be stored in fixed locations associated with the given interrupt signal. Under what circumstances is this a practical technique? Explain why it is inconvenient in general.

3.12 In Section 3.4, it was stated that UNIX is unsuitable for real-time applications because a process executing in kernel mode may not be preempted. Elaborate.

PROGRAMMING PROJECT 1

DEVELOPING A SHELL

The Shell or Command Line Interpreter is the fundamental User interface to an Operating System. Your first project is to write a simple shell - `myshell` - that has the following properties:

1. The shell must support the following internal commands:
   - i. `cd <directory>` - Change the current default directory to `<directory>`. If the `<directory>` argument is not present, report the current directory. If the directory does not exist an appropriate error should be reported. This command should also change the `PWD` environment variable.
   - ii. `clr` - Clear the screen.
   - iii. `dir <directory>` - List the contents of directory `<directory>`.
   - iv. `environ` - List all the environment strings.
   - v. `echo <comment>` - Display `<comment>` on the display followed by a new line (multiple spaces/tabs may be reduced to a single space).
   - vi. `help` - Display the user manual using the `more` filter.
   - vii. `pause` - Pause operation of the shell until '`Enter`' is pressed.
   - viii. `quit` - Quit the shell.
   - ix. The shell environment should contain `shell=<pathname>/myshell` where `<pathname>/myshell` is the full path for the shell executable (not a hardwired path back to your directory, but the one from which it was executed).
2. All other command line input is interpreted as program invocation, which should be done by the shell `fork`ing and `exec`ing the programs as its own child processes. The programs should be executed with an environment that contains the entry: `parent=<pathname>/myshell` where `<pathname>/myshell` is as described in 1.ix. above.
3. The shell must be able to take its command line input from a file. That is, if the shell is invoked with a command line argument:

   ```
   myshell batchfile
   ```

   then `batchfile` is assumed to contain a set of command lines for the shell to process. When the end-of-file is reached, the shell should exit. Obviously, if the shell is invoked without a command line argument, it solicits input from the user via a prompt on the display.
4. The shell must support i/o-redirection on either or both *stdin* and/or *stdout*. That is, the command line

   ```
   programname arg1 arg2 < inputfile > outputfile
   ```

   will execute the program `programname` with arguments `arg1` and `arg2`, the *stdin FILE* stream replaced by `inputfile` and the *stdout FILE stream* replaced by `outputfile`.

stdout redirection should also be possible for the internal commands dir, environ, echo, & help.

With output redirection, if the redirection character is > then the outputfile is created if it does not exist and truncated if it does. If the redirection token is >> then outputfile is created if it does not exist and appended to if it does.

5. The shell must support background execution of programs. An ampersand (&) at the end of the command line indicates that the shell should return to the command line prompt immediately after launching that program.
6. The command line prompt must contain the pathname of the current directory.

Note: You can assume that all command line arguments (including the redirection symbols, <, > & >> and the background execution symbol, &) will be delimited from other command line arguments by white space - one or more spaces and/or tabs (see the command line in 4. above).

Project Requirements

1. Design a simple command line shell that satisfies the above criteria and implement it on the specified UNIX platform.
2. Write a simple manual describing how to use the shell. The manual should contain enough detail for a beginner to UNIX to use it. For example, you should explain the concepts of I/O redirection, the program environment, and background program execution. The manual MUST be named readme and must be a simple text document capable of being read by a standard Text Editor.

   For an example of the sort of depth and type of description required, you should have a look at the online manuals for csh and tcsh (man csh, man tcsh). These shells obviously have much more functionality than yours and thus, your manuals don't have to be quite so large.

   You should NOT include building instructions, included file lists or source code - we can find that out from the other files you submit. This should be an Operator's manual not a Developer's manual.
3. The source code **MUST** be extensively commented and appropriately structured to allow your peers to understand and easily maintain the code. Properly commented and laid out code is much easier to interpret, and it is in your interests to ensure that the person marking your project is able to understand your coding without having to perform mental gymnastics!
4. Details of submission procedures will be supplied well before the deadline.
5. The submission should contain only source code file(s), include file(s), a makefile (all lower case please), and the readme file (all lowercase, please). No executable program should be included. The person marking your project will be automatically rebuilding your shell program from the source code provided. If the submitted code does not compile it cannot be marked!
6. The makefile (all lowercase, please) **MUST** generate the binary file myshell (all lower case please). A sample makefile would be

```
# Joe Citizen, s1234567 - Operating Systems Project 1
# CompLab1/01 tutor: Fred Bloggs
myshell: myshell.c utility.c myshell.h
gcc -Wall myshell.c utility.c -o myshell
```

   The program myshell is then generated by just typing make at the command line prompt.

   Note: The fourth line in the above makefile **MUST** begin with a tab

7. In the instance shown above, the files in the submitted directory would be:

```
makefile
myshell.c
utility.c
myshell.h
readme
```

Submission

A `makefile` is required. All files in your submission will be copied to the same directory, therefore, do not include any paths in your `makefile`. The `makefile` should include all dependencies that build your program. If a library is included, your `makefile` should also build the library.

Do not hand in any binary or object code files. All that is required is your source code, a `makefile` and `readme` file. Test your project by copying the source code only into an empty directory and then compile it by entering the command `make`.

We shall be using a shell script that copies your files to a test directory, deletes any pre-existing `myshell`, `*.a`, and/or `*.o` files, performs a `make`, copies a set of test files to the test directory, and then exercises your shell with a standard set of test scripts through *stdin* and command line arguments. If this sequence fails due to wrong names, wrong case for names, wrong version of source code that fails to compile, nonexistence of files, etc. then the marking sequence will also stop. In this instance, the only marks that can be awarded will be for the tests completed at that point and the source code and manual.

Required Documentation

Your source code will be assessed and marked as well as the `readme` manual. Commenting is definitely required in your source code. The user manual can be presented in a format of your choice (within the limitations of being displayable by a simple Text Editor). Again, the manual should contain enough detail for a beginner to UNIX to use the shell. For example, you should explain the concepts of I/O redirection, the program environment and background execution. The manual **MUST** be named `readme` (all lowercase, please, **NO** `.txt` extension).

DATE \@ "M/d/yy" 8/11/07