

# Operating Systems

**Operating system:** An operating system is a program that controls the execution of application program and acts as an interface between applications and the computer hardware.

**Processes:** Many definitions have been given for the term process; one of them is “A program in execution”.

**Multiprogramming:** With multiprogramming several processes are kept in memory at one time, the CPU is switched back and forth among them to increase CPU utilization while also keeping peripheral devices in use. Multiprogramming provides an environment where the various system resources (for example, CPU, memory, peripheral devices) are utilized effectively.

**Time Sharing:** Time sharing or multitasking is a logical extension of multiprogramming. The CPU executes multiple processes by switching among them. Time shared operating systems allow many users (from one to several hundreds) to use a computer system interactively at the same time.

## References

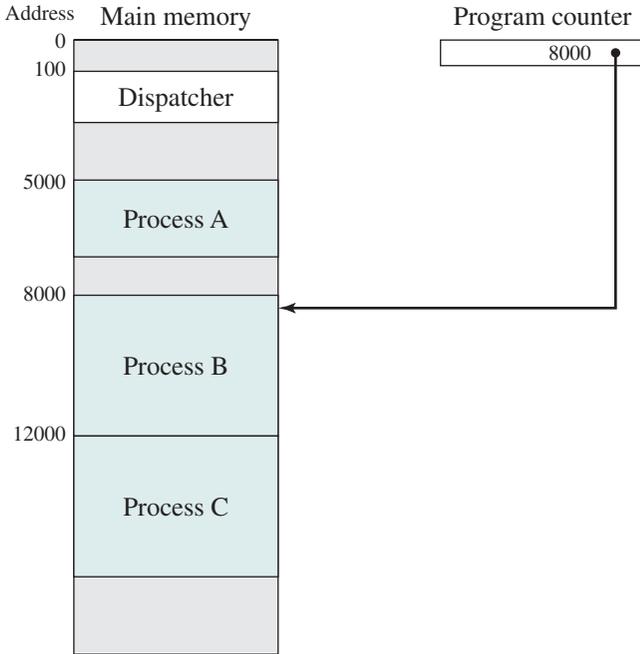
- 1- Operating Systems by William Stallings
- 2- Operating Systems Concepts by Abraham Silberschatz, Peter B. Galvan.
- 3- Operating Systems by H. M. Deitel.

## 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



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

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

(a) Trace of process A    (b) Trace of process B    (c) Trace of process C

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

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

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

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

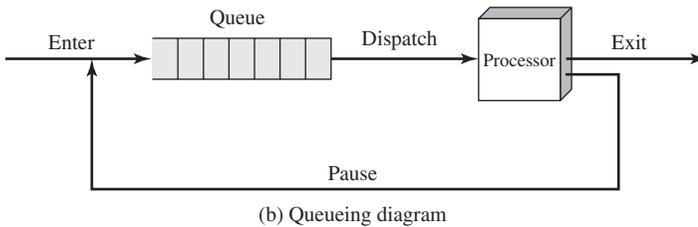
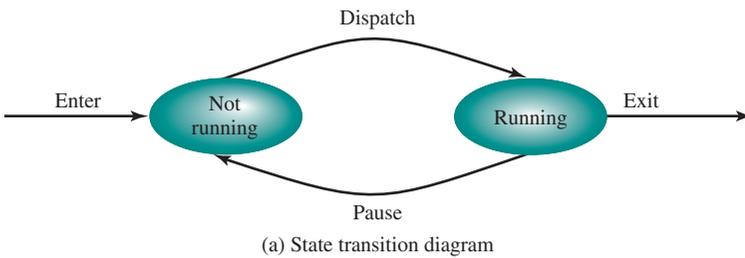
**Figure 3.4** Combined Trace of Processes of Figure 3.2

and the execution of some code in the dispatcher, which executes six instructions before turning control to process B.<sup>2</sup> 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.

### 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.



**Figure 3.5 Two-State Process Model**

<sup>2</sup>The small number of instructions executed for the processes and the dispatcher are unrealistically low; they are used in this simplified example to clarify the discussion.

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.

We can describe the behavior of the dispatcher in terms of this queueing 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

**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 log-on	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.

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**.

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.

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.<sup>3</sup>

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.

---

<sup>3</sup>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.

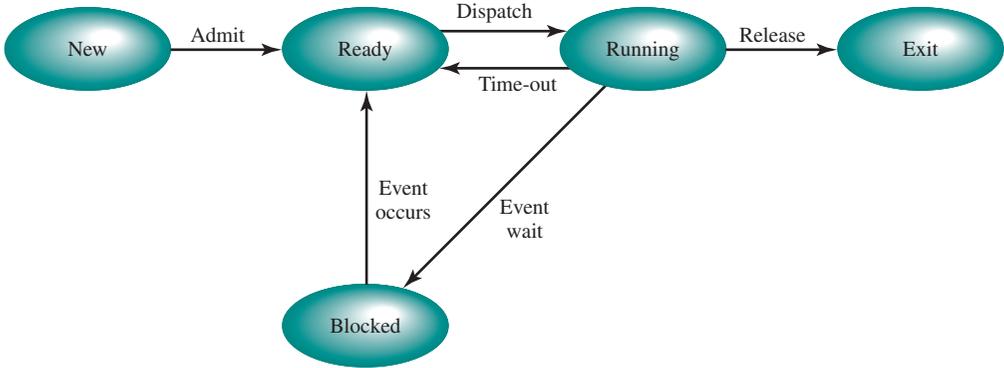
**Table 3.2** Reasons for Process Termination

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 (e.g., 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.

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:**<sup>4</sup> A process that cannot execute until some event occurs, such as the completion of an I/O operation.

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



**Figure 3.6** Five-State Process Model

- **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 on to 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.<sup>5</sup>

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

<sup>5</sup>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.

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 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 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.<sup>6</sup> 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

---

<sup>6</sup>In general, the term *preemption* is defined to be the reclaiming of a resource from a process before the process has 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.

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 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).

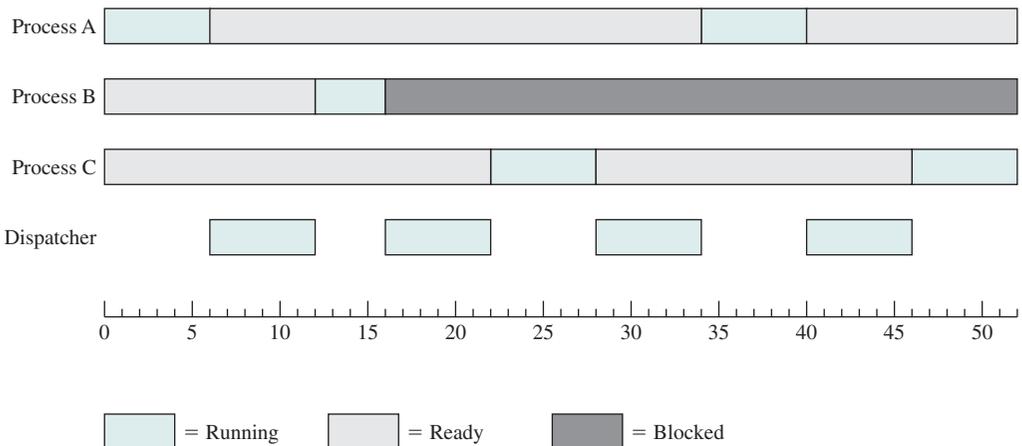
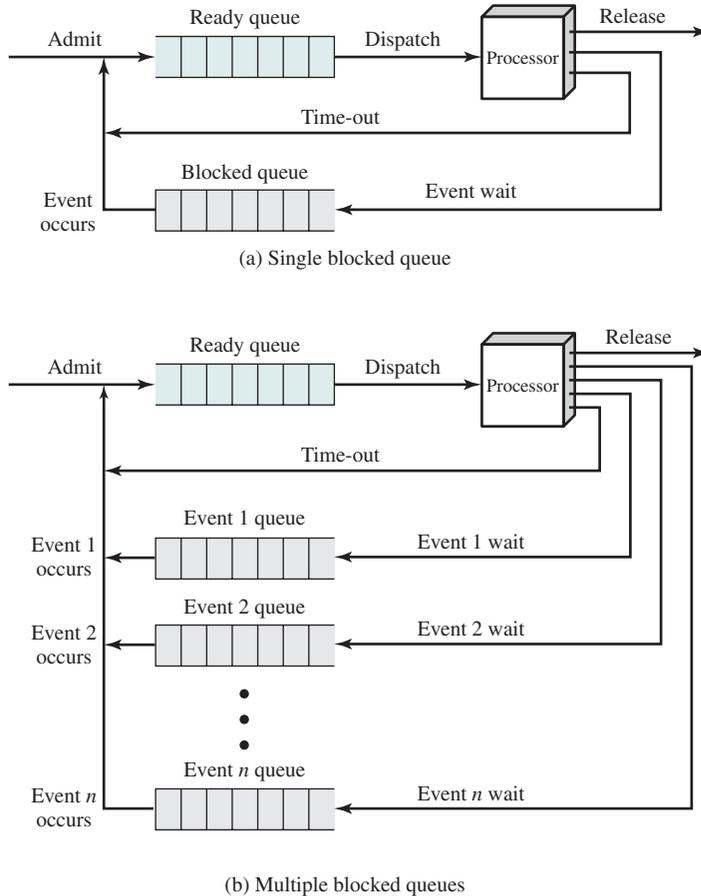


Figure 3.7 Process States for the Trace of Figure 3.4



**Figure 3.8** Queuing Model for Figure 3.6

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 on to disk into a suspend queue. This is a queue of existing processes that have been temporarily kicked out of 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.

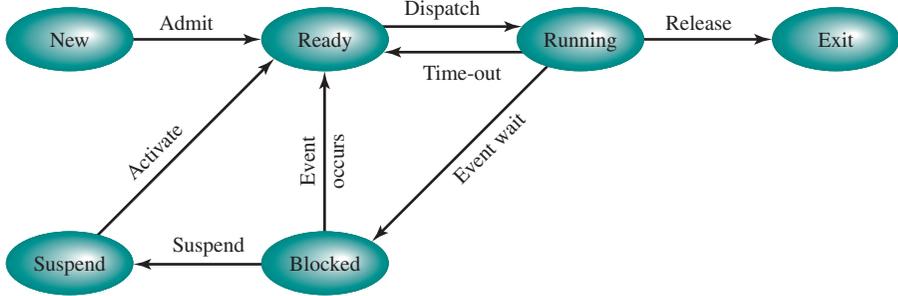
With the use of swapping as just described, one other state must be added to our process behavior model (Figure 3.9a): the Suspend state. When all of the processes in main memory are in the Blocked state, the OS can suspend one process by putting it in the Suspend state and transferring it to disk. The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

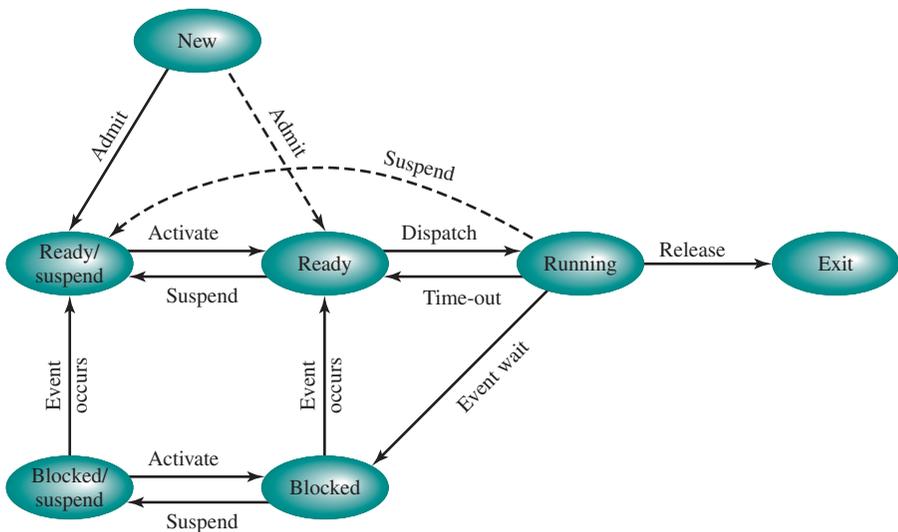
But this line of reasoning presents a difficulty. All of the processes that have been suspended were in the Blocked state at the time of suspension. It clearly would not do any good to bring a blocked process back into main memory, because it is still not ready for execution. Recognize, however, that 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  $2 \times 2$  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.



(a) With one suspend state



(b) With two suspend states

**Figure 3.9** Process State Transition Diagram with Suspend States

- **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. In particular, 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, a blocked process will be suspended.
- **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.

**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

**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.

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.

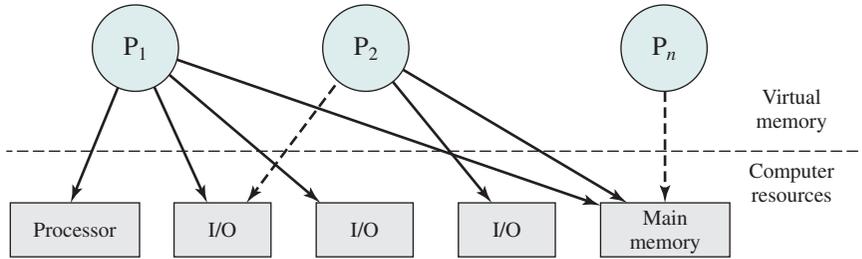
Finally, a parent process may wish to suspend a descendent process. For example, process A may spawn process B to perform a file read. Subsequently, process B encounters an error in the file read procedure and reports this to process A. Process A suspends process B to investigate the cause.

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 ( $P_1, \dots, P_n$ ) 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.



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

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 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.

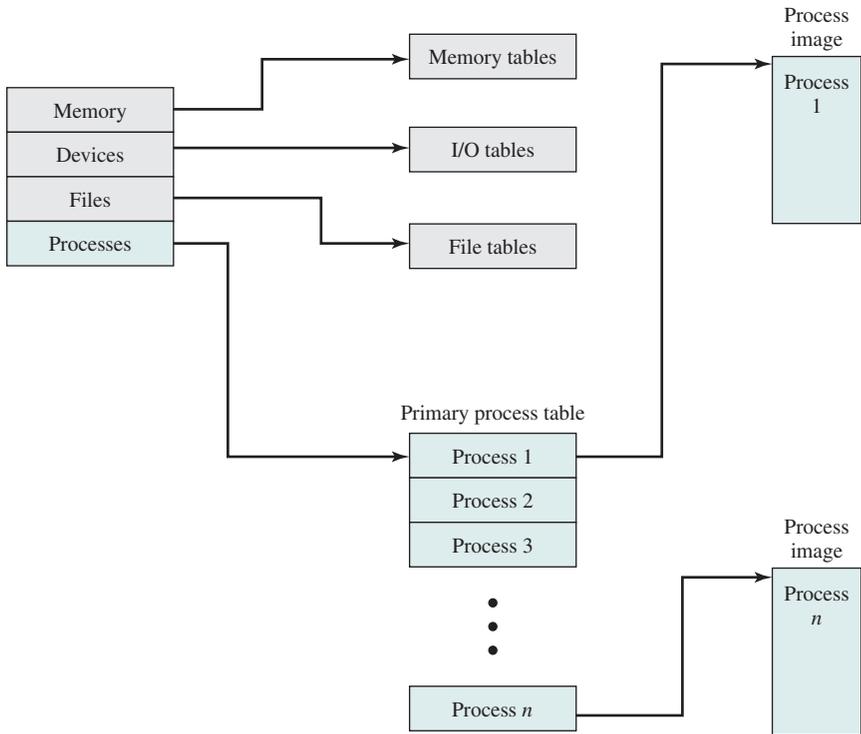
**Memory tables** are used to keep track of both main (real) and secondary (virtual) memory. Some of main memory is reserved for use by the OS; the remainder is available for use by processes. Processes are maintained on secondary memory using some sort of virtual memory or simple swapping mechanism.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



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

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; 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 P) 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 block*.<sup>7</sup> 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.

**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).

<sup>7</sup>Other commonly used names for this data structure are *task control block*, *process descriptor*, and *task descriptor*.

Modern operating systems presume paging hardware that allows noncontiguous physical memory to support partially resident processes.<sup>8</sup> At any given time, a portion of a process image may be in main memory, with the remainder in secondary memory.<sup>9</sup> 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 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

<sup>8</sup>A brief overview of the concepts of pages, segments, and virtual memory is provided in the subsection on memory management in Section 2.3.

<sup>9</sup>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

<b>Process Identification</b>
<p><b>Identifiers</b></p> <p>Numeric identifiers that may be stored with the process control block include</p> <ul style="list-style-type: none"> <li>• Identifier of this process.</li> <li>• Identifier of the process that created this process (parent process).</li> <li>• User identifier.</li> </ul>
<b>Processor State Information</b>
<p><b>User-Visible Registers</b></p> <p>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.</p> <p><b>Control and Status Registers</b></p> <p>These are a variety of processor registers that are employed to control the operation of the processor. These include</p> <ul style="list-style-type: none"> <li>• <b>Program counter:</b> Contains the address of the next instruction to be fetched.</li> <li>• <b>Condition codes:</b> Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow).</li> <li>• <b>Status information:</b> Includes interrupt enabled/disabled flags, execution mode.</li> </ul> <p><b>Stack Pointers</b></p> <p>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.</p>
<b>Process Control Information</b>
<p><b>Scheduling and State Information</b></p> <p>This is information that is needed by the operating system to perform its scheduling function. Typical items of information:</p> <ul style="list-style-type: none"> <li>• <b>Process state:</b> Defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, halted).</li> <li>• <b>Priority:</b> 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).</li> <li>• <b>Scheduling-related information:</b> 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.</li> <li>• <b>Event:</b> Identity of event the process is awaiting before it can be resumed.</li> </ul> <p><b>Data Structuring</b></p> <p>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.</p> <p><b>Interprocess Communication</b></p> <p>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.</p> <p><b>Process Privileges</b></p> <p>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.</p> <p><b>Memory Management</b></p> <p>This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.</p> <p><b>Resource Ownership and Utilization</b></p> <p>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.</p>

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.

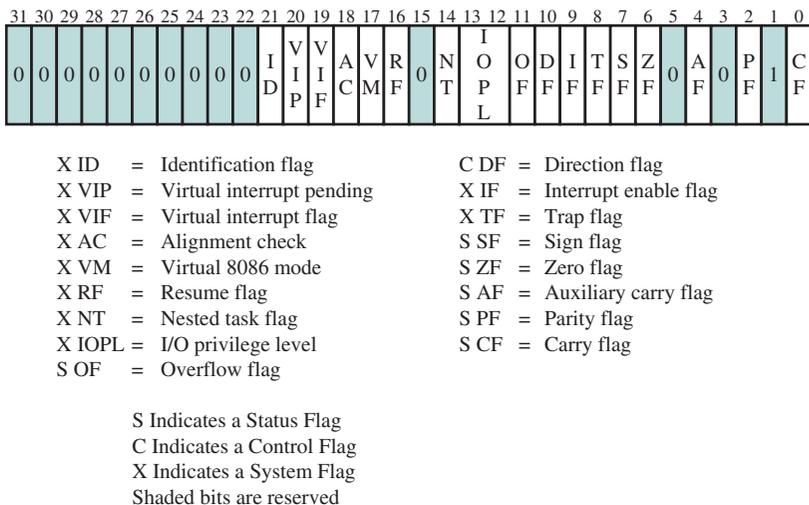
In addition to these process identifiers, a process may be assigned a user identifier that indicates the user responsible for the job.

**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 contains condition codes plus other status information. A good example of a processor status word is that on Intel x86 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 an x86 processor.

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.

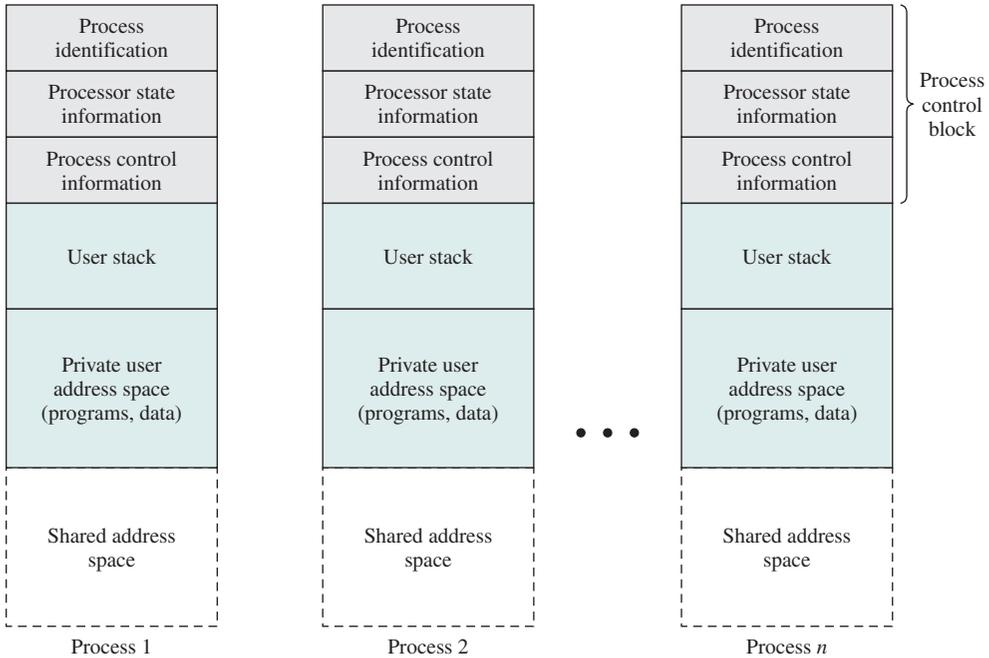
Figure 3.13 suggests the structure of process images in virtual memory. Each process image consists of a process control block, a user stack, the private address space of the process, and any other address space that the process shares with other processes.



**Figure 3.12** x86 EFLAGS Register

**Table 3.6** x86 EFLAGS Register Bits

<b>Status Flags (condition codes)</b>
<p><b>AF (Auxiliary carry flag)</b> Represents carrying or borrowing between half-bytes of an 8-bit arithmetic or logic operation using the AL register.</p>
<p><b>CF (Carry flag)</b> Indicates carrying out or borrowing into the leftmost bit position following an arithmetic operation; also modified by some of the shift and rotate operations.</p>
<p><b>OF (Overflow flag)</b> Indicates an arithmetic overflow after an addition or subtraction.</p>
<p><b>PF (Parity flag)</b> Parity of the result of an arithmetic or logic operation. 1 indicates even parity; 0 indicates odd parity.</p>
<p><b>SF (Sign flag)</b> Indicates the sign of the result of an arithmetic or logic operation.</p>
<p><b>ZF (Zero flag)</b> Indicates that the result of an arithmetic or logic operation is 0.</p>
<b>Control Flag</b>
<p><b>DF (Direction flag)</b> 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).</p>
<b>System Flags (should not be modified by application programs)</b>
<p><b>AC (Alignment check)</b> Set if a word or doubleword is addressed on a nonword or nondoubleword boundary.</p>
<p><b>ID (Identification flag)</b> If this bit can be set and cleared, this processor supports the CPUID instruction. This instruction provides information about the vendor, family, and model.</p>
<p><b>RF (Resume flag)</b> Allows the programmer to disable debug exceptions so that the instruction can be restarted after a debug exception without immediately causing another debug exception.</p>
<p><b>IOPL (I/O privilege level)</b> When set, it causes the processor to generate an exception on all accesses to I/O devices during protected mode operation.</p>
<p><b>IF (Interrupt enable flag)</b> When set, the processor will recognize external interrupts.</p>
<p><b>TF (Trap flag)</b> When set, it causes an interrupt after the execution of each instruction. This is used for debugging.</p>
<p><b>NT (Nested task flag)</b> Indicates that the current task is nested within another task in protected mode operation.</p>
<p><b>VM (Virtual 8086 mode)</b> Allows the programmer to enable or disable virtual 8086 mode, which determines whether the processor runs as an 8086 machine.</p>
<p><b>VIP (Virtual interrupt pending)</b> Used in virtual 8086 mode to indicate that one or more interrupts are awaiting service.</p>
<p><b>VF (Virtual interrupt flag)</b> Used in virtual 8086 mode instead of IF.</p>



**Figure 3.13** User Processes in Virtual Memory

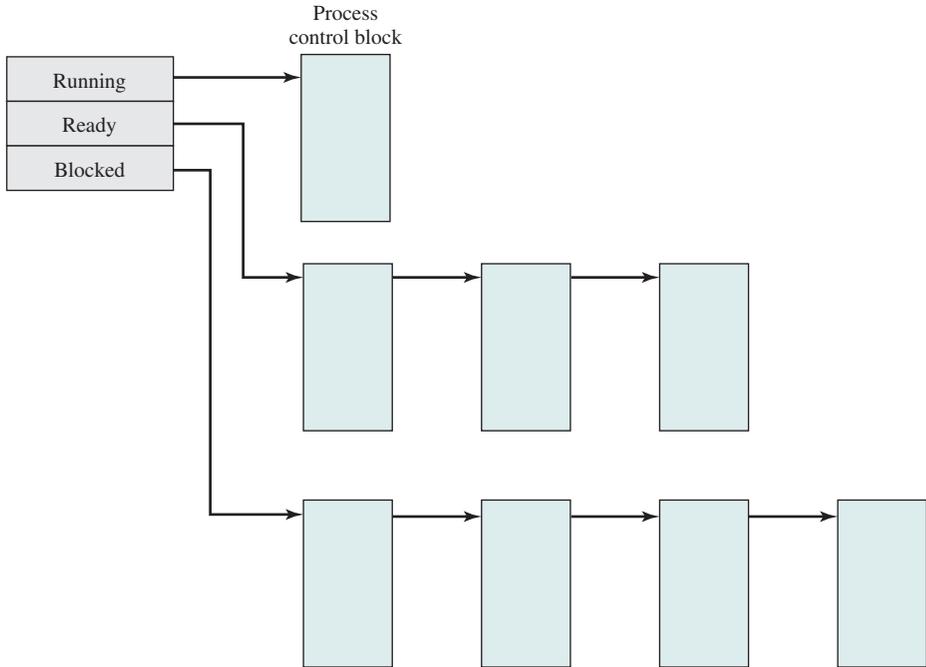
In the figure, each process image appears as a contiguous range of addresses. In an actual implementation, this may not be the case; it will depend on the memory management scheme and the way in which control structures are organized by the OS.

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.



**Figure 3.14** Process List Structures

- 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 trade-off 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.

**Table 3.7** Typical Functions of an Operating System Kernel

<b>Process Management</b>
<ul style="list-style-type: none"> <li>• Process creation and termination</li> <li>• Process scheduling and dispatching</li> <li>• Process switching</li> <li>• Process synchronization and support for interprocess communication</li> <li>• Management of process control blocks</li> </ul>
<b>Memory Management</b>
<ul style="list-style-type: none"> <li>• Allocation of address space to processes</li> <li>• Swapping</li> <li>• Page and segment management</li> </ul>
<b>I/O Management</b>
<ul style="list-style-type: none"> <li>• Buffer management</li> <li>• Allocation of I/O channels and devices to processes</li> </ul>
<b>Support Functions</b>
<ul style="list-style-type: none"> <li>• Interrupt handling</li> <li>• Accounting</li> <li>• Monitoring</li> </ul>

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.

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` (interrupt 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 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

**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

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.<sup>10</sup> 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).
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.

---

<sup>10</sup>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.