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

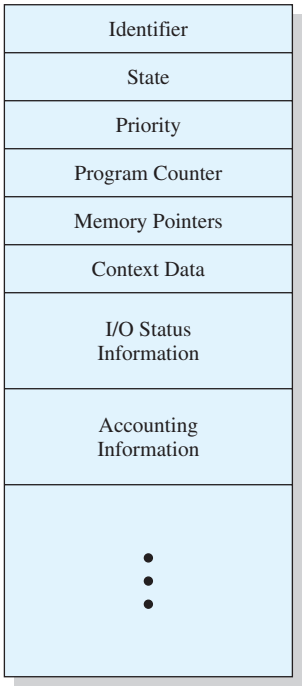


Figure 3.1 Simplified Process Control Block

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

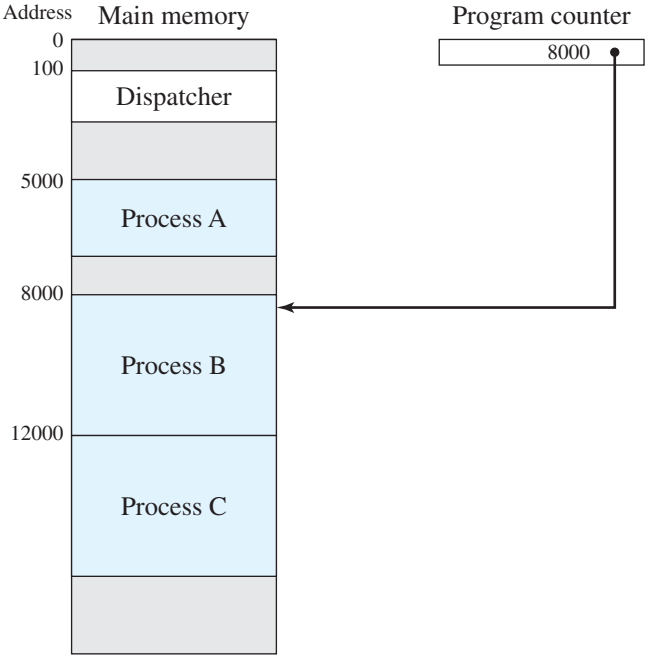


Figure 3.2 **Snapshot of Example Execution (Figure 3.4) at Instruction Cycle 13**

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**

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	

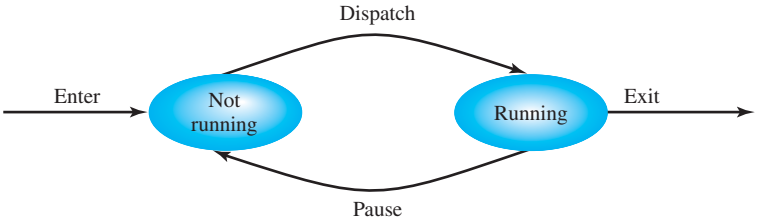
100 = Starting address of dispatcher program

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

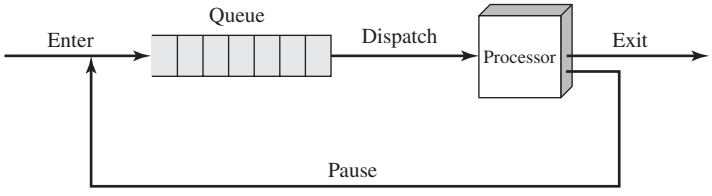
Figure 3.4 Combined Trace of Processes of Figure 3.2

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.

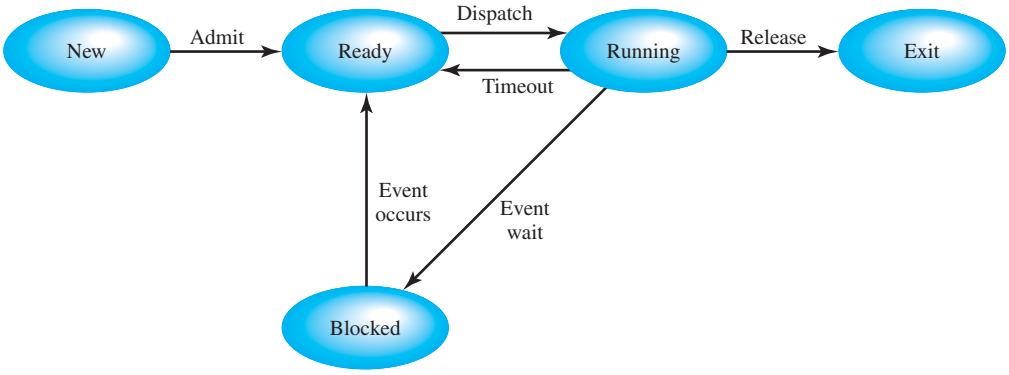


Figure 3.6 **Five-State Process Model**

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.

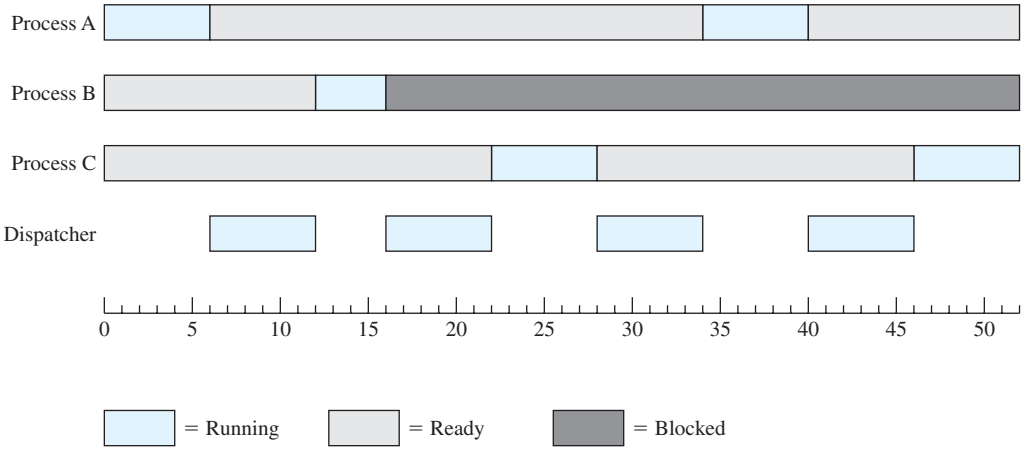
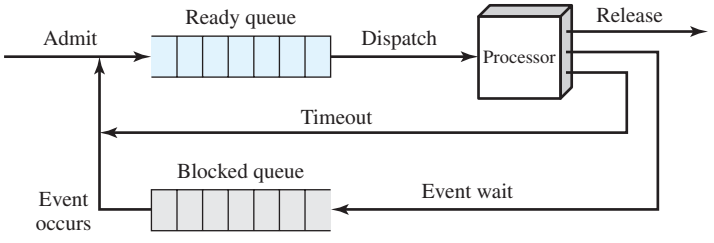
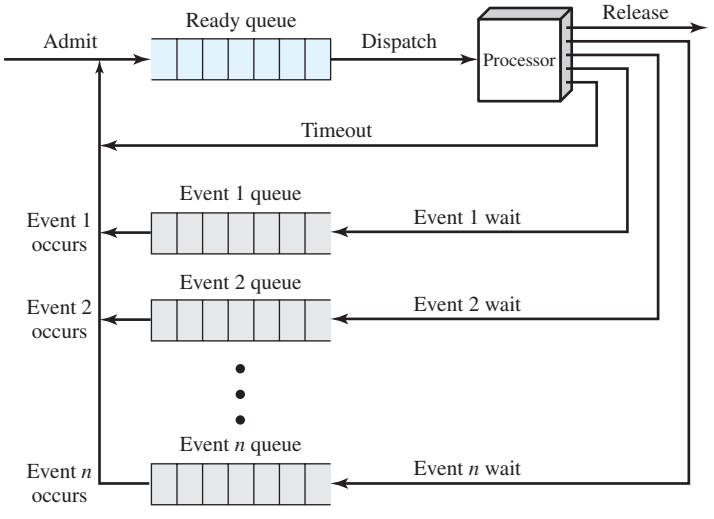


Figure 3.7 **Process States for the Trace of Figure 3.4**



(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

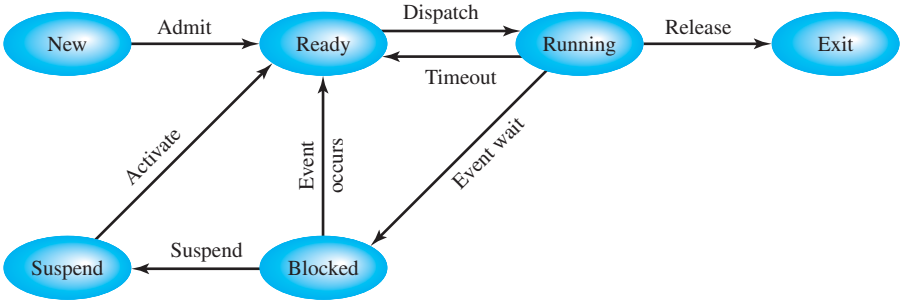
The Need for Swapping The three principal states just described (Ready, Running, Blocked) provide a systematic way of modeling the behavior of processes and guide the implementation of the OS. Some operating systems are constructed using just these three states.

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.

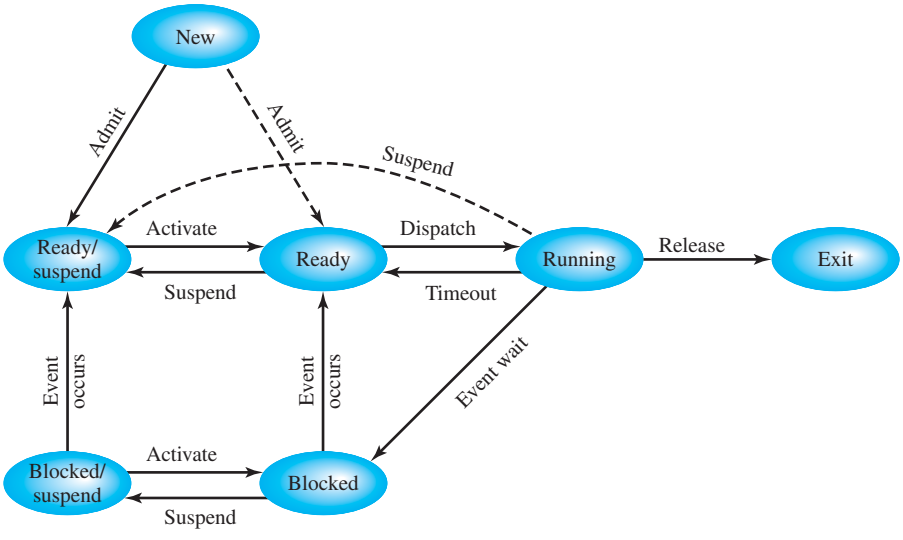
Recall that the reason for all of this elaborate machinery is that I/O activities are much slower than computation and therefore the processor in a uniprogramming system is idle most of the time. But the arrangement of Figure 3.8b does not entirely solve the problem. It is true that, in this case, memory 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