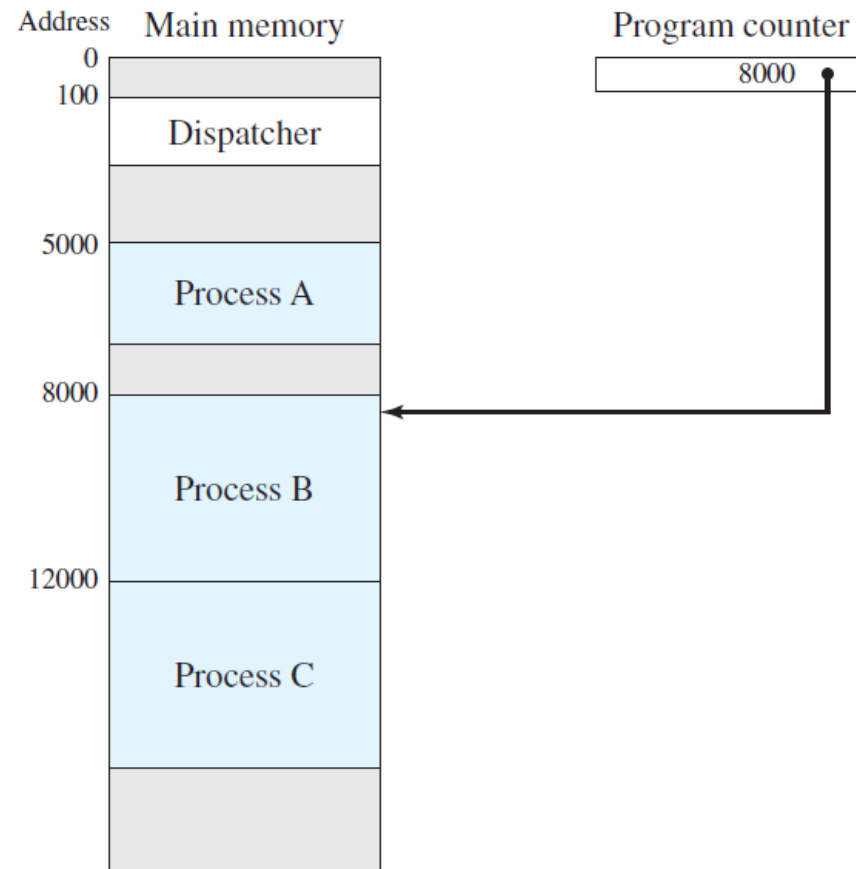


Process States

106

→ 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

- 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. 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.



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

- Next figure 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.

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	

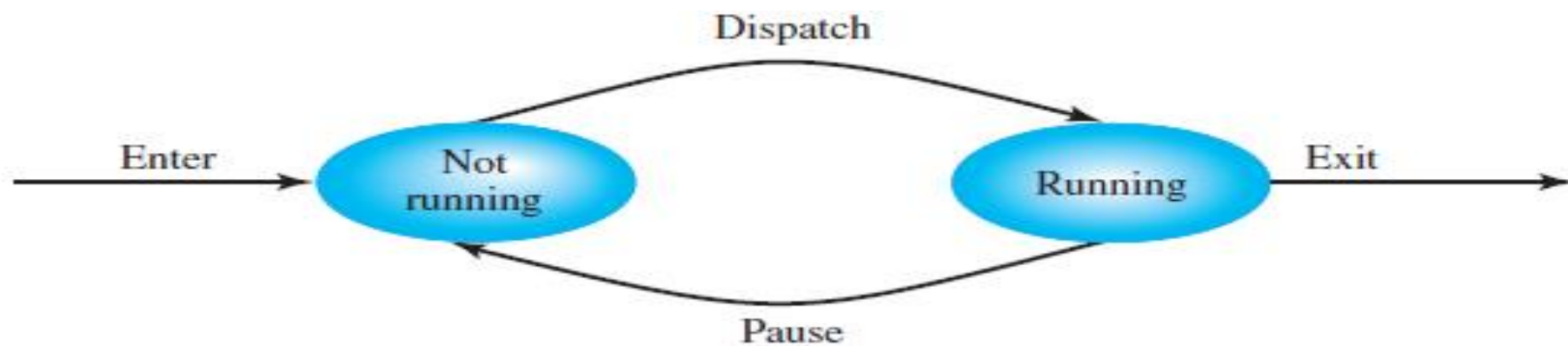
100 = Starting address of dispatcher program

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

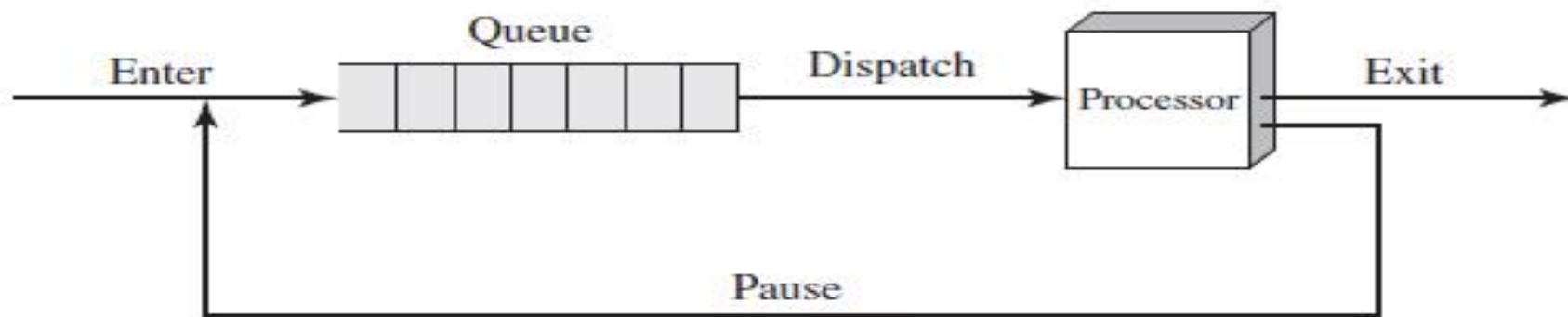
Figure 3.4 Combined Trace of Processes of Figure 3.2

A Two-State Process Model

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. 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.



(a) State transition diagram



(b) Queuing diagram

Figure 3.5 Two-State Process Model

The Creation and Termination of Processes

- **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.

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.

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.

A Five-State Model

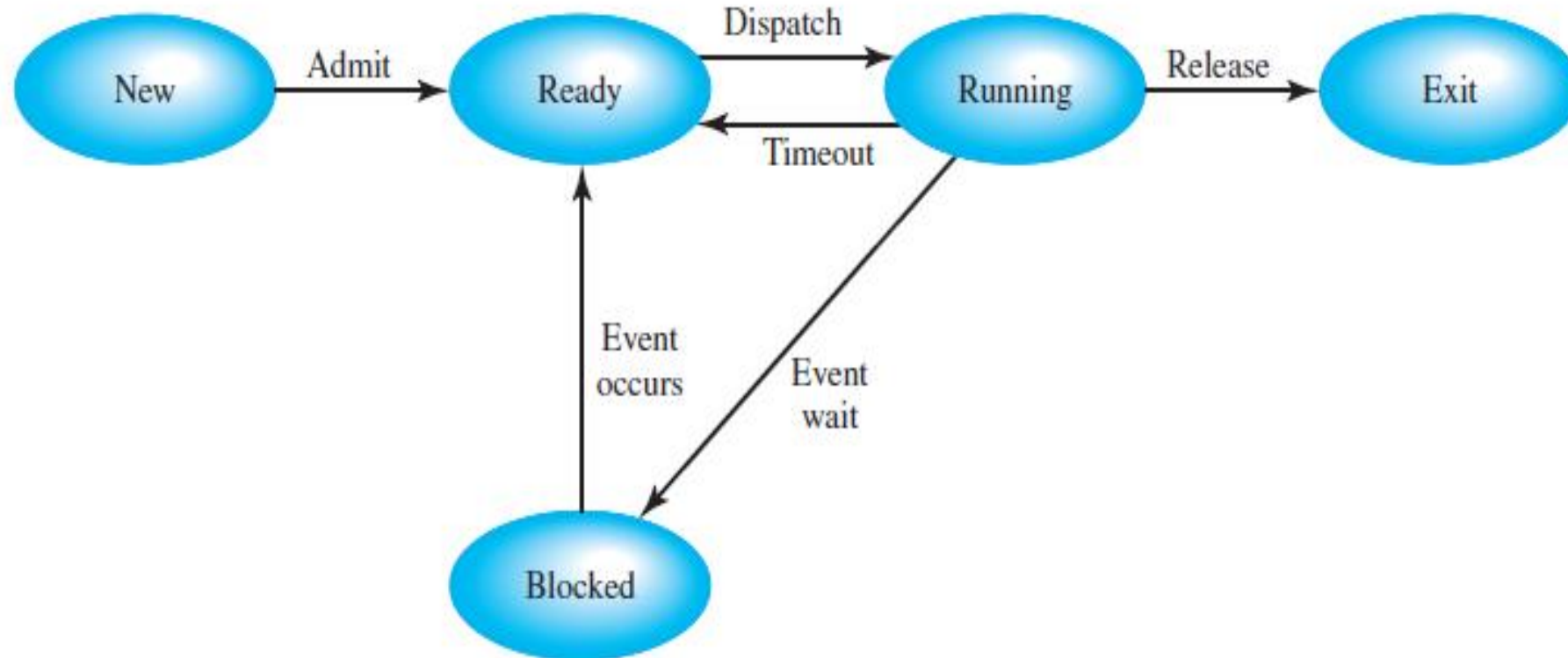


Figure 3.6 Five-State Process Model

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

- **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 following points indicate the types of events that lead to each state transition for a process; the possible transitions are as follows:

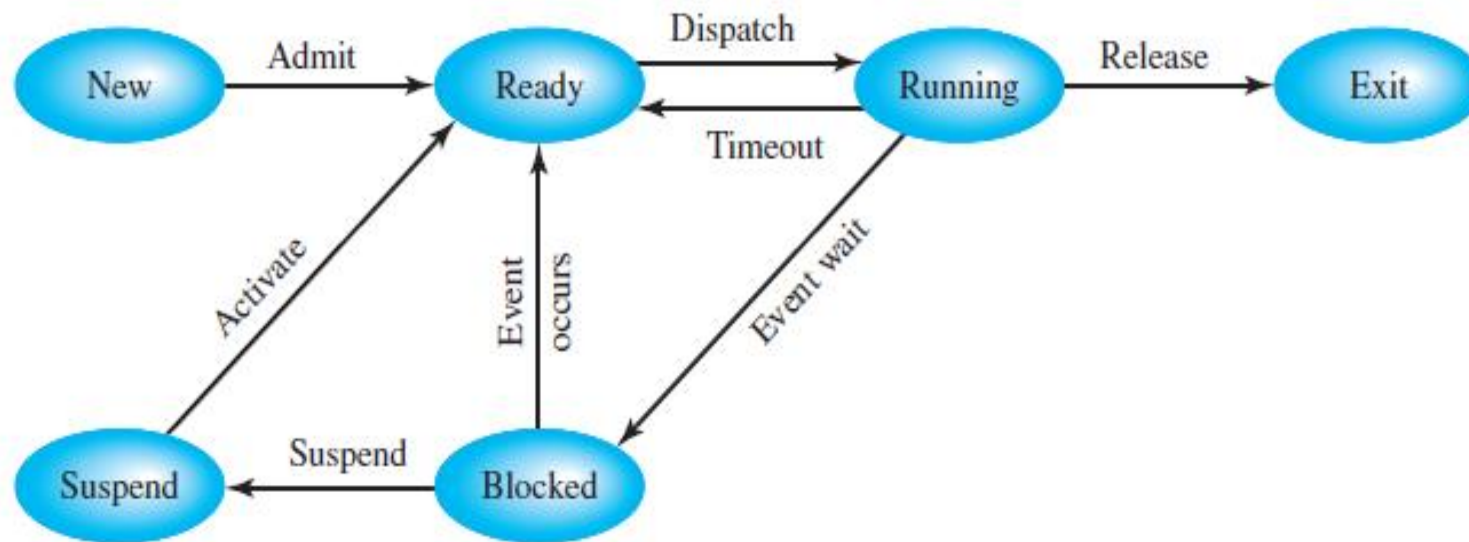
- **Null → New:** A new process is created to execute a program. This event occurs for any of the reasons listed in Table 3.1.
- **New → Ready:** The OS will move a process from the New state to the Ready state when it is prepared to take on an additional process. Most systems set some limit based on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

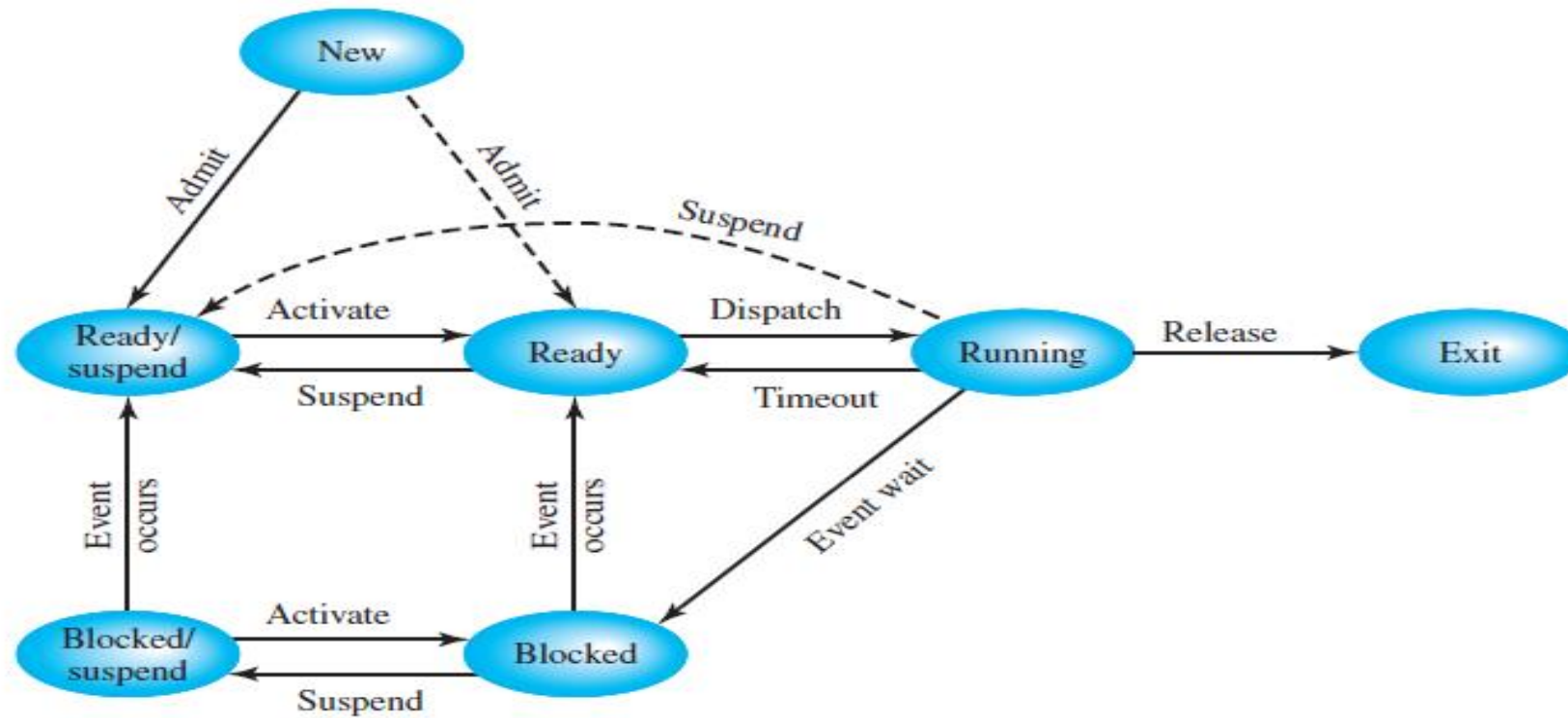
- **Running → Blocked:** A process is put in the Blocked state if it requests something for which it must wait. A request to the OS is usually in the form of a system service call; that is, a call from the running program to a procedure that is part of the operating system code. For example, a process 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.

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.



(a) With one suspend state



(b) With two suspend states

Figure 3.9 Process State Transition Diagram with Suspend States

- 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.

- 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.

- The state transition model that we have developed. (The dashed lines in the figure indicate possible but not necessary transitions.) Important new transitions are the following:
- **Blocked -> Blocked/Suspend:** If there are no ready processes, then at least one blocked process is swapped out to make room for another process that is not blocked. This transition can be made even if there are ready processes available, if the OS determines that the currently running process or a ready process that it would like to dispatch requires more main memory to maintain adequate performance.

- **Blocked/Suspend -> Ready/Suspend:** A process in the Blocked/Suspend state is moved to the Ready/Suspend state when the event for which it has been waiting occurs. Note that this requires that the state information concerning suspended processes must be accessible to the OS.

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

- **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 house-keeping duties at an early time, so that it can maintain a large pool of processes that are not blocked. With this strategy, there would often be insufficient room in main memory for a new process; hence the use of the (New → Ready/Suspend) transition. On the other hand, we could argue that a just-in-time philosophy of creating processes as late as possible reduces OS overhead and allows that OS to perform the process-creation duties at a time when the system is clogged with blocked processes anyway.

- **Blocked/Suspend → Blocked:** Inclusion of this transition may seem to be poor design. After all, if a process is not ready to execute and is not already in main memory, what is the point of bringing it in? But consider the following scenario: A process terminates, freeing up some main memory. There is a process in the (Blocked/Suspend) queue with a higher priority than any of the processes in the (Ready/Suspend) queue and the OS has reason to believe that the blocking event for that process will occur soon. Under these circumstances, it would seem reasonable to bring a blocked process into main memory in preference to a ready process.

...

- **Running → Ready/Suspend:** Normally, a running process is moved to the Ready state when its time allocation expires. If, however, the OS is preempting the process because a higher-priority process on the Blocked/Suspend queue has just become unblocked, the OS could move the running process directly to the (Ready/Suspend) queue and free some main memory.
- **Any State → Exit:** Typically, a process terminates while it is running, either because it has completed or because of some fatal fault condition. However, in some operating systems, a process may be terminated by the process that created it or when the parent process is itself terminated. If this is allowed, then a process in any state can be moved to the Exit state.

Table 3.3 Reasons for Process Suspension

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