Lecture Three
Process Concept

Objectives After reading this chapter, you should understand:

- The concept of process
- Process life cycle
- Process states and state transition
- Process control block (PCB)
- Context Switch

3.1 Introduction

Early computers were batch systems that executed jobs, followed by the emergence of time-shared systems that ran user programs, or tasks. Even on a single-user system, a user may be able to run several programs at one time: a word processor, a web browser, and an e-mail package. And even if a computer can execute only one program at a time, such as on an embedded device that does not support multitasking, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them processes.

Process Detention

Informally, as mentioned earlier, a process is a program in execution. The status of the current activity of a process is represented by the value of the program counter and the contents of the processor's registers. The memory layout of a process is typically divided into multiple sections, and is shown in Figure 3.1. These sections include:

- Text section— the executable code
- Data section—global variables
- Heap section—memory that is dynamically allocated during program run time
- Stack section— temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

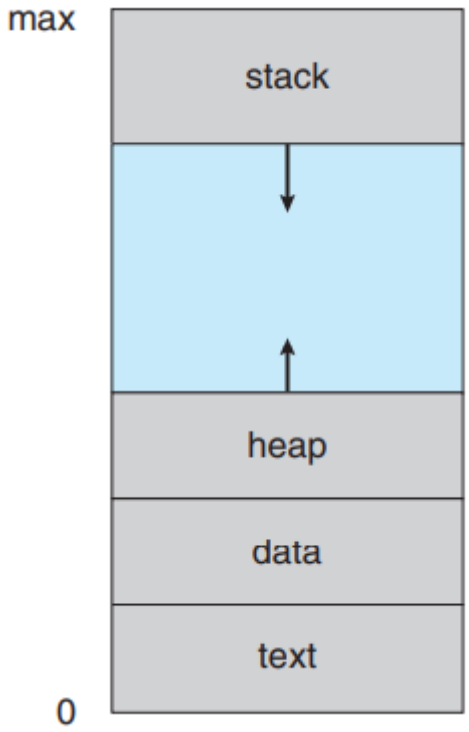

Figure 3.1 Layout of a process in memory.

3.2 Process States: Life cycle of Process

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

- New. The process is being created.
- Running. Instructions are being executed.
- Waiting. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- Ready. The process is waiting to be assigned to a processor.
- Terminated. The process has finished execution.

These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is important

to realize that only one process can be running on any processor core at any instant. Many processes may be ready and waiting, however. The state diagram corresponding to these states is presented in Figure 3.2.

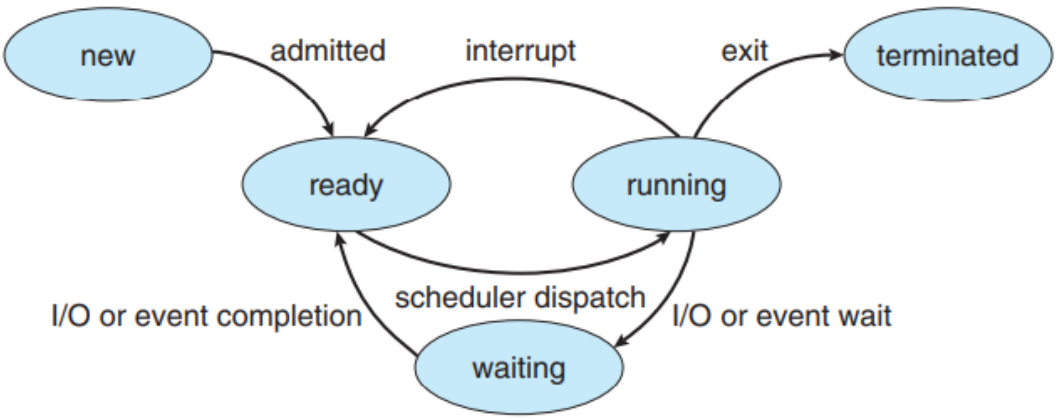

Figure 3.2 Diagram of process state.

Self Review

1. (T/F) at any given time, only one process can be executing instructions on a computer
2. A process enters the blocked state when it is waiting for event accrue. Name a several events that might cause a process to enter the block state.

   Ans:
   1) False. On a multiprocessor computer, there can be as many processes executing instructions as there are processors.
   2) A process may enter the blocked state if it issues a request for data located on a high-latency device such as a hard disk or requests a resource that is allocated to another process and is currently unavailable (e.g., a printer). A process may also block until an event occurs, such as a user pressing a key or moving a mouse.

3.3Process State Transition

Applications that have strict real-time constraints might need to prevent processes from being swapped or paged out to secondary memory. A simplified overview of UNIX process states and the transitions between states is shown in the following figure 3.3

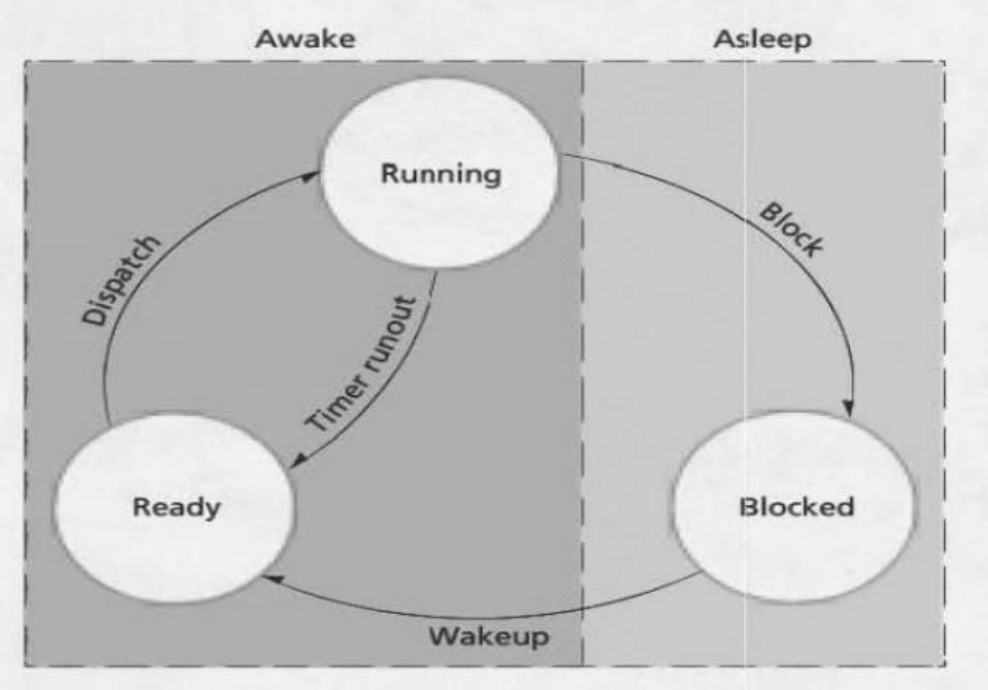

Figure 3.3 Process State Transition Diagram

An active process is normally in one of the five states in the diagram. The arrows show how the process changes states.

- A process is running if the process is assigned to a CPU. A process is removed from the running state by the scheduler if a process with a higher priority becomes runnable. A process is also preempted if a process of equal priority is runnable when the original process consumes its entire time slice.
- A process is runnable in memory if the process is in primary memory and ready to run, but is not assigned to a CPU.
- A process is sleeping in memory if the process is in primary memory but is waiting for a specific event before continuing execution. For example, a process sleeps while waiting for an I/O operation to complete, for a locked resource to be unlocked, or for a timer to expire. When the event occurs, a wakeup call is sent to the process. If the reason for its sleep is gone, the process becomes runnable.

- When a process' address space has been written to secondary memory, and that process is not waiting for a specific event, the process is runnable and swapped.
- If a process is waiting for a specific event and has had its whole address space written to secondary memory, the process is sleeping and swapped.

  If a machine does not have enough primary memory to hold all its active processes, that machine must page or swap some address space to secondary memory.

- When the system is short of primary memory, the system writes individual pages of some processes to secondary memory but leaves those processes runnable. When a running process, accesses those pages, the process sleeps while the pages are read back into primary memory.

Self Review

1. How does the operating system prevent a process from monopolizing a processor?

2. What is the difference between processes that are awake and those that are asleep?

Ans:

1) An interrupting clock generates an interrupt after a specified time quantum, and the operating system dispatches another process to execute. The interrupted process will run again when it gets to the head of the ready list and a processor again becomes available.

2) 2) A process that is awake is in active contention for a processor; a process that is asleep cannot use a processor even if one becomes available.

3.4 **Process Control Block**

Each process is represented in the operating system by a process control block (PCB)—also called a task control block. Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique Process Identification number(or PID), which is typically an integer number. The PID provides a unique value for each process in the

system, and it can be used as an index to access various attributes of a process within the kernel.

A PCB is shown in Figure 3.4.and Figure 3.5. It contains many pieces of information associated with a specific process, including these:

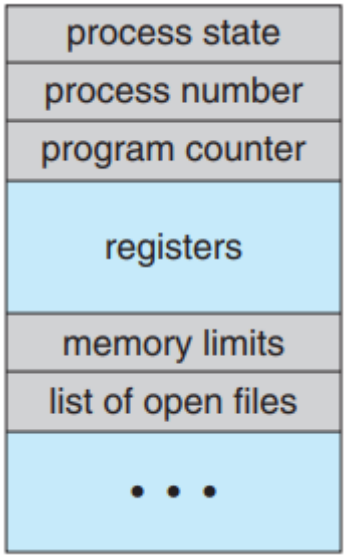

Figure 3.4 Process control block (PCB).

- PID
- Process state. The state may be new, ready, running, waiting, halted, and so on.
- Program counter. The counter indicates the address of the next instruction to be executed for this process
- CPU registers. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.
- CPU-scheduling information. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. (describes process scheduling.)
- Memory-management information. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.
- Accounting information. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- I/O status information. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

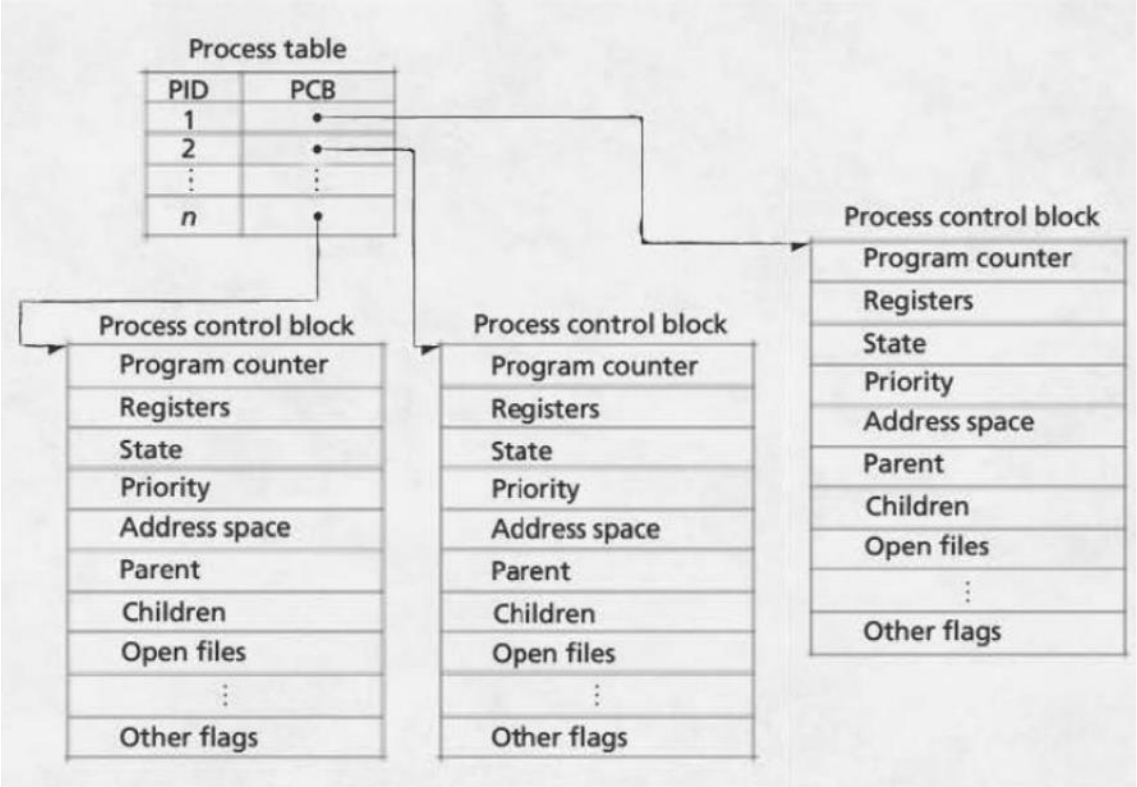

Figure 3.5 Accounting information and PCB for processes

Self Review

1. What is the purpose of the process table?

2. (T/F) The structure of a PCB is dependent on the operating system implementation.

Ans: **1)** The process table enables the operating system to locate each process's PCB.

2) True.

3.5 Process Operations

Operating systems must be able to perform certain process operations, including:

• create a process

• destroy a process

• suspend a process

• resume a process

change a process's priority

• block a process

• wake up a process

• dispatch a process

• enable a process to communicate with another process (this is called inter-process communication).

A process may **spawn** a new process. If it does, the creating process is called the **parent process** and the created process is called the **child process**. Each child process is created by exactly one parent process. Such creation yields a hierarchical process structure similar to Fig. 3.6, in which each child has only one parent (e.g., A is the one parent of C; H is the one parent of I), but each parent may have many children (e.g., B, C, and D are the children of A; F and G are the children of C)

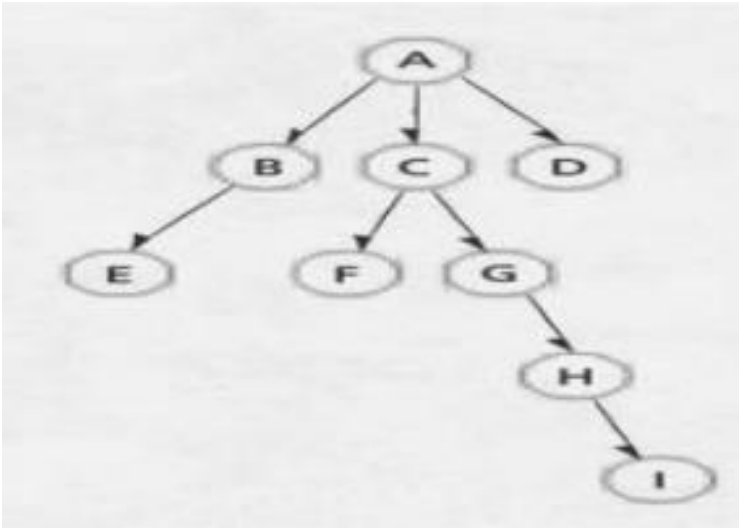

Figure 3.6 : hierarchical process structure

Self Review

1. (T/F) A process may have zero parent processes.

2. Why is it advantageous to create a hierarchy of processes as opposed to a linked list?

Ans:

1) True. The first process that is created, often called *init* in UNIX systems, does not have a parent. Also, in some systems, when a parent process is destroyed, its children proceed independently without their parent.

2) A hierarchy of processes allows the operating system to track parent/child relationships between processes. This simplifies operations such as locating all the child processes of a particular parent process when that parent terminates.

3.6 Suspend and Resume

Many operating systems allow administrators, users or processes to suspend a process. A **suspended** process is indefinitely removed from contention for time on a processor without being destroyed. Historically, this operation allowed a system operator to manually adjust the system load and/or respond to threats of system failure. Most of today's computers execute too quickly to permit such manual adjustments. However, an administrator or a user suspicious of the partial results of a process may suspend it (rather than **aborting** it) until the user can ascertain whether the process is functioning correctly. This is useful for detecting security threats (such as malicious code execution) and for software debugging purposes.

Figure 3.7 displays the process state-transition diagram of Fig. 3.2 modified to include suspend and resume transitions. Two new states have been added, ***suspendedready*** and ***suspendedblocked***. Above the dashed line in the figure are the **active states**; below it is the **suspended states**.

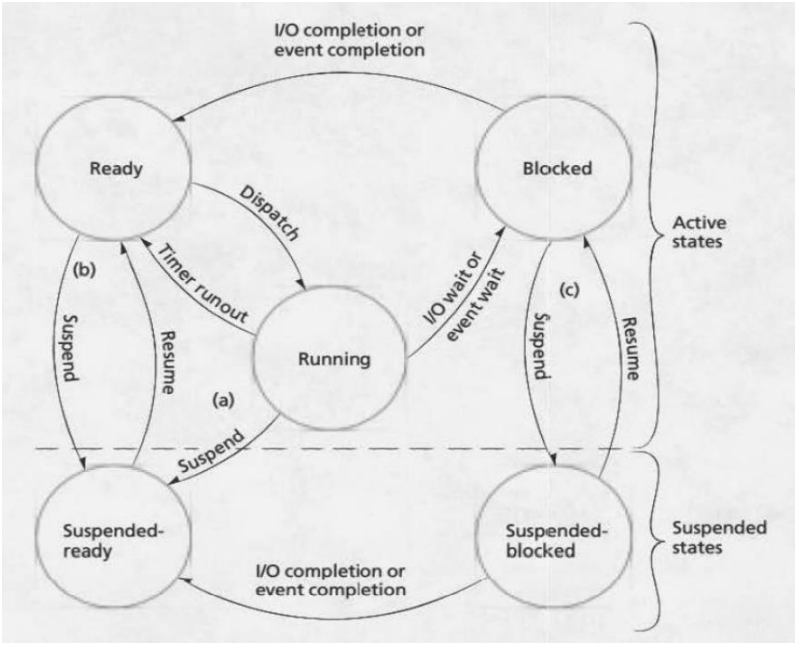

Figure 3.7 : process state-transition diagram with suspend and resume

A suspension may be initiated by the process being suspended or by another process. On a uniprocessor system a *running* process may suspend itself, indicated by Fig. 3.7(a); no other process could be running at the same moment to issue the suspend. A *running* process may also suspend a *ready* process or a *blocked* process, depicted in Fig. 3.7(b) and (c).

On a multiprocessor system, a *running* process may be suspended by another process running at that moment on a different processor.

Clearly, a process suspends itself only when it is in the *running* state, in such a situation, the process makes the transition from *running* to *suspendedready*.

When a process suspends a *ready* process, the *ready* process transitions from *ready* to *suspended ready*.

A *suspended ready* process may be made ready, or resumed, by another process, causing the first process to transition from *suspended ready* to *ready*.

A *blocked* process will make the transition from *blocked* to *suspended blocked* when it is suspended by another process.

A *suspended blocked* process may be resumed by another process and make the transition from *suspended blocked* to *blocked*.

One could argue that instead of suspending a *blocked* process, it is better to wait until the I/O completion or event completion occurs and the process

becomes *ready*; then the process could be suspended to the *suspended ready* state. Unfortunately, the completion may never come, or it may be delayed indefinitely. The designer must choose between performing the suspension of the *blocked* process or creating a mechanism by which the suspension will be made from the *ready* state when the I/O or event completes. Because suspension is typically a high-priority activity, it is performed immediately. When the I/O or event completion finally occurs (if indeed it does), the *suspended blocked* process makes the transition from *suspended blocked* to *suspended ready*.

Self Review

1. In what three ways can a process get to the *suspended ready* state?

2. In what scenario is it best to suspend a process rather than abort it?

Answer:

1) A process can get to the *suspended ready* state if it is suspended from the *running* state, if it is suspended from the *ready* state by a *running* process or if it is in the *suspended- blocked* state and the I/O completion or event completion it is waiting for occurs.

2) When a user or system administrator is suspicious of a process's behavior but does not want to lose the work performed by the process, it is better to suspend the process so that it can be inspected.

3.7 Context Switch

interrupts cause the operating system to change a CPU core from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems.

When an interrupt occurs, the system needs to save the current context of the process running on the CPU core so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process.

It includes the value of the CPU registers, the process state ,and memory-management information. Generically, we perform a state save of the current state of the CPU core, be it in kernel or user mode, and then a state restores to resume operations.

Switching the CPU core to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch and is illustrated in Figure 3.6.

When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context switch time is pure overhead, because the system does no useful

work while switching. Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). A typical speed is a several microseconds.