Learn to labor and to wait.

—Henry Wadsworth Longfellow—

Many shall run to and fro, and knowledge shall be increased.

—Daniel 12:2—

You will wake, and remember, and understand.

—Robert Browning—

It was surprising that Nature had gone tranquilly on with her golden process in the midst of so much devilment.

—Stephen Crane—

Chapter 3

Process Concepts

Objectives

After reading this chapter, you should understand:

- *the concept of a process.*
- *the process life cycle.*
- *process states and state transitions.*
- *process control blocks (PCBs)/process descriptors.*
- *how processors transition between processes via context switching.*
- *how interrupts enable hardware to communicate with software.*
- *how processes converse with one another via interprocess communication (IPC).*
- *UNIX processes.*

Chapter Outline

3.1 Introduction

Many systems in nature have the ability to perform multiple actions at the same time. For example, the human body performs a great variety of operations in parallel—or, as we will say, **concurrently**. Respiration, blood circulation, thinking, walking and digestion, for example, can occur concurrently, as can the senses—sight, touch, smell, taste and hearing. Computers, too, perform operations concurrently. It is common for desktop computers to be compiling a program, sending a file to a printer, rendering a Web page, playing a digital video clip and receiving e-mail concurrently (see the Operating Systems Thinking feature, Customers Ultimately Want Applications).

In this chapter we formally introduce the notion of a **process**, which is central to understanding how today's computer systems perform and keep track of many simultaneous activities. We introduce some of the more popular definitions of process. We present the concept of discrete **process states** and discuss how and why processes make transitions between these states. We also discuss various operations that operating systems perform to service processes, such as creating, destroying, suspending, resuming and waking up processes.

3.1.1 Definition of Process

The term "process" in the context of operating systems was first used by and the designers of the Multics system in the 1960s (see the Mini Case Study, CTSS and Multics and the Biographical Note, Fernando J. Corbató).[1] Since that time, process, used somewhat interchangeably with **task**, has been given many definitions, such as: a program in execution, an asynchronous activity, the "animated spirit" of a procedure, the "locus of control" of a procedure in execution, that which is manifested by

Operating Systems Thinking

Customers Ultimately Want Applications

Ultimately, computers exist to run useful applications. Operating systems designers can lose sight of this because they tend to be concerned with complex technical issues of operating systems architecture and engineering. But they cannot operate in a void; they must know their user community; the kinds of applications those users will be running and what results the users really want from those applications. Hardware stores sell many tools to help you perform household chores. The tool designer needs to be aware that few people are interested in simply purchasing tools; rather they ultimately buy the tools for the tasks they perform. Customers do not really want saws, hammers and drills—they want cuts, nails in wood and holes.

the existence of a data structure called a "process descriptor" or a "process control block" in the operating system, that entity to which processors are assigned and the "dispatchable" unit. A program is to a process as sheet music is to a symphony orchestra playing the music.

Two key concepts are presented by these definitions. First, a process is an entity. Each process has its own address space, which typically consists of a **text region**, **data region** and **stack region**. The text region stores the code that the processor executes. The data region stores variables and dynamically allocated memory that the process

Mini Case Study

CTSS and Multics

In the early 1960s, a team of programmers at MIT's Project MAC, led by Professor Fernando Corbató, developed the Compatible Time-Sharing System (CTSS) which allowed users to command the computing power of an IBM 7090 (which eventually became an IBM 7094) with typewriterlike terminals.[2, 3] CTSS ran a conventional batch stream to keep the computer working while giving fast responses to interactive users editing and debugging programs. The computing capabilities provided by CTSS resembled those provided to personal computer users today—namely, a highly interactive environment in which the computer gave rapid responses to large numbers of relatively trivial requests.

In 1965 the same MIT group, in cooperation with Bell Labs and GE, began working on the Multics (Multiplexed Information and Computing Service) operating system, the successor to CTSS. Multics was a large and complex system; the designers envisioned a general-purpose computer utility that could be "all things to all people." Although it did not achieve commercial success, it was used by various research centers until the last system was shut down in 2000.[4]

A variety of Multics features influenced the development of future operating systems, including UNIX, TSS/360, TENEX and TOPS-20.[5] Multics used a combination of segmentation and paging for its virtual memory system, with paging controlled only by the operating system, while segments were manipulated by user programs as well.[6] It was one of the first operating systems to be written in a high-level systems-programming language, IBM's PL/I.[7, 8] Its designers coined the term "process" as it is currently used in operating systems. Multics was built for security. It included a discretionary access mechanism called **ACL** (Access Control List), which was a list of permissions on a memory segment which would look familiar to UNIX users. Later versions included a mandatory access control, **AIM** (Access Isolation Mechanism), an enhancement to ACL where every user and object was assigned a security classification, which helped Multics become the first operating system to get a B2 security rating from the U.S. government.[9, 10, 11] In 1976 the first commercial relational database system was written, the **Multics Relational Data Store**.[12]

uses during execution. The stack region stores instructions and local variables for active procedure calls. The contents of the stack grow as a process issues nested procedure calls and shrink as called procedures return.[13] Second, a process is a "program in execution." A program is an inanimate entity; only when a processor "breathes life" into a program does it become the active entity we call a process.

Self Review

1. Why is a process's address space divided into multiple regions?
2. (T/F) The terms "process" and "program" are synonymous.

Ans: **1)** Each region of an address space typically contains information that is accessed in a similar way. For example, most processes read and execute instructions, but do not modify their instructions. Processes read from and write to the stack, but in last-in-first-out order. Processes read and write data in any order. Separating a process's address space into different regions enables the operating system to enforce such access rules. **2)** False. A process is a program in execution; a program is an inanimate entity.

3.2 Process States: Life Cycle of a Process

The operating system must ensure that each process receives a sufficient amount of processor time. For any system, there can be only as many truly concurrently executing processes as there are processors. Normally, there are many more processes

Biographical Note

Fernando J. Corbató

Fernando Jose Corbató received his Ph.D. in Physics from MIT in 1956. Corbató was a professor at MIT from 1965 to 1996, retiring as a Professor Emeritus in the Department of Electrical Engineering and Computer Science.[14] He was a founding member of MIT's Project MAC and led the development of the CTSS and Multics project.[15, 16] He coauthored technical papers on Multics and on the project management issues of that large cooperation among MIT Project MAC, General Electric, and Bell Labs.

Corbató received the 1990 Turing Award for his work on CTSS and Multics.[17] His award lecture was "On Building Systems That Will Fail," in which he describes how the intrinsic complexity of large and innovative projects will always lead to mistakes, drawing largely from his Multics experiences. In his lecture he advised developers to assume that any given error will occur and that they should therefore plan how to handle it.[18]

than processors in a system. Thus, at any given time, some processes can execute and some cannot.

During its lifetime, a process moves through a series of discrete **process states**. Various events can cause a process to change state. A process is said to be ***running*** (i.e., in the ***running* state**) if it is executing on a processor. A process is said to be ***ready*** (i.e., in the ***ready* state**) if it could execute on a processor if one were available. A process is said to be ***blocked*** (i.e., in the ***blocked* state**) if it is waiting for some event to happen (such as an **I/O completion event,** for example) before it can proceed. There are other process states, but for now we will concentrate on these three.

For simplicity, let us consider a uniprocessor system, although the extension to multiprocessing (see Chapter 15, Multiprocessor Management) is not difficult. In a uniprocessor system only one process may be *running* at a time, but several may be *ready* and several *blocked*. The operating system maintains a **ready list** of *ready* processes and a **blocked list** of *blocked* processes. The ready list is maintained in priority order, so that the next process to receive a processor is the first one in the list (i.e., the process with the highest priority). The blocked list is typically unordered—processes do not become **unblocked** (i.e., *ready*) in priority order; rather, they unblock in the order in which the events they are waiting for occur. As we will see later, there are situations in which several processes may block awaiting the same event; in these cases it is common to prioritize the waiting processes.

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 an event to occur. Name several events that might cause a process to enter the *blocked* 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.

3.3 Process Management

As the operating system interleaves the execution of its processes, it must carefully manage them to ensure that no errors occur as the processes are interrupted and resumed. Processes should be able to communicate with the operating system to perform simple tasks such as starting a new process or signaling the end of process execution. In this section, we discuss how operating systems provide certain fundamental services to processes—these include creating processes, destroying processes, suspending processes, resuming processes, changing a process's priority, blocking processes, waking up processes, dispatching processes, enabling processes to interact via interprocess communication (IPC) and more. We also discuss how operating systems manage process resources to allow multiple processes to actively contend for processor time at once.

3.3.1 Process States and State Transitions

When a user runs a program, processes are created and inserted into the ready list. A process moves toward the head of the list as other processes complete their turns using a processor. When a process reaches the head of the list, and when a processor becomes available, that process is given a processor and is said to make a **state transition** from the *ready* state to the *running* state (Fig. 3.1). The act of assigning a processor to the first process on the ready list is called **dispatching** and is performed by a system entity called the **dispatcher**. Processes that are in the *ready* or *running* states are said to be awake, because they are actively contending for processor time. The operating system manages state transitions to best serve processes in the system. To prevent any one process from monopolizing the system, either accidentally or maliciously, the operating system sets a hardware **interrupting clock** (also called an **interval timer**) to allow a process to run for a specific time interval or **quantum**. If the process does not voluntarily yield the processor before the time interval expires, the interrupting clock generates an interrupt, causing the operating system to gain control of the processor (see Section 3.4, Interrupts). The operating system then changes the state of the previously *running* process to *ready* and dispatches the first process on the ready list, changing its state from *ready* to *running*. If a *running* process initiates an input/output operation before its quantum expires, and therefore must wait for the I/O operation to complete before it can use a processor again, the *running* process voluntarily relinquishes the processor. In this case, the process

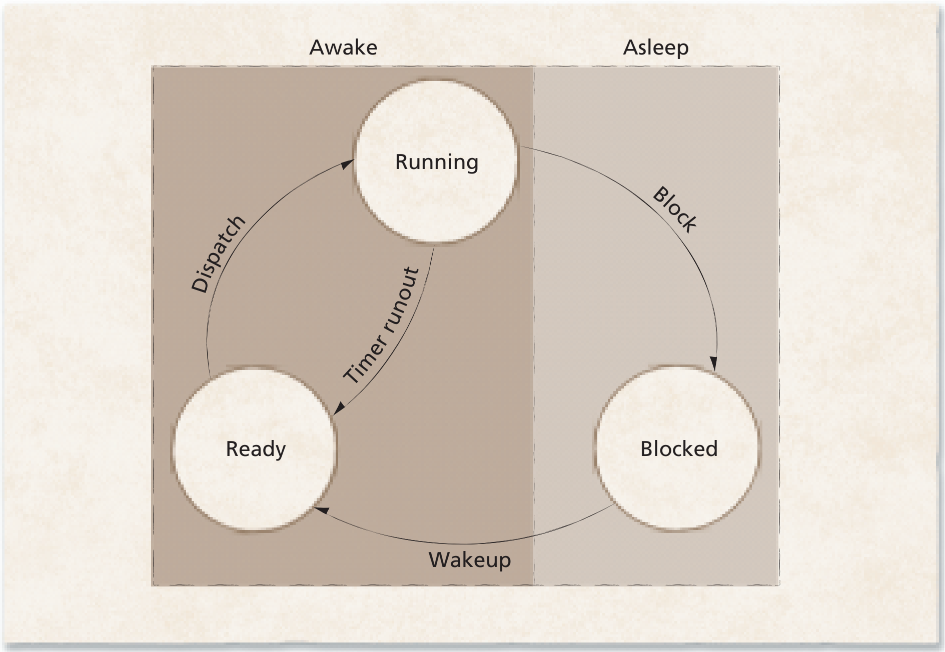


Figure 3.1 | Process state transitions.

is said to **block** itself, pending the completion of the I/O operation. Processes in the *blocked* state are said to be asleep, because they cannot execute even if a processor becomes available. The only other allowable state transition in our three-state model occurs when an I/O operation (or some other event the process is waiting for) completes. In this case, the operating system transitions the process from the *blocked* to the *ready* state.

We have defined four possible state transitions. When a process is dispatched, it transitions from *ready* to *running*. When a process's quantum expires, it transitions from *running* to *ready*. When a process blocks, it transitions from *running* to *blocked*. Finally, when a process wakes up because of the completion of some event it is awaiting, it transitions from *blocked* to *ready*. Note that the only state transition initiated by the user process itself is block—the other three transitions are initiated by the operating system.

In this section, we have assumed that the operating system assigns each process a quantum. Some early operating systems that ran on processors without interrupting clocks employed **cooperative multitasking**, meaning that each process must voluntarily yield the processor on which it is running before another process can execute. Cooperative multitasking is rarely used in today's systems, however, because it allows processes to accidentally or maliciously monopolize a processor (e.g., by entering an infinite loop or simply refusing to yield the processor in a timely fashion).

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)** 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.

3.3.2 Process Control Blocks (PCBs)/Process Descriptors

The operating system typically performs several operations when it creates a process. First, it must be able to identify each process; therefore, it assigns a **process identification number (PID)** to the process. Next, the operating system creates a **process control block (PCB)**, also called a **process descriptor**, which maintains information that the operating system needs to manage the process. PCBs typically include information such as:

- PID
- process state (e.g., *running*, *ready* or *blocked*)
- **program counter** (i.e., a value that determines which instruction the processor should execute next)
- scheduling priority

- credentials (i.e., data that determines the resources this process can access)
- a pointer to the process's **parent process** (i.e., the process that created this process)
- pointers to the process's **child processes** (i.e., processes created by this process) if any
- pointers to locate the process's data and instructions in memory
- pointers to allocated resources (such as files).

The PCB also stores the register contents, called the **execution context**, of the processor on which the process was last running when it transitioned out of the *running* state. The execution context of a process is architecture specific but typically includes the contents of general-purpose registers (which contain process data that the processor can directly access) in addition to process management registers, such as registers that store pointers to a process's address space. This enables the operating system to restore a process's execution context when the process returns to the *running* state.

When a process transitions from one state to another, the operating system must update information in the process's PCB. The operating system typically maintains pointers to each process's PCB in a systemwide or per-user **process table** so that it can access the PCB quickly (Fig. 3.2). The process table is one of many operating system data structures we discuss in this text (see the Operating Systems Thinking feature, Data Structures in Operating Systems). When a process is termi-

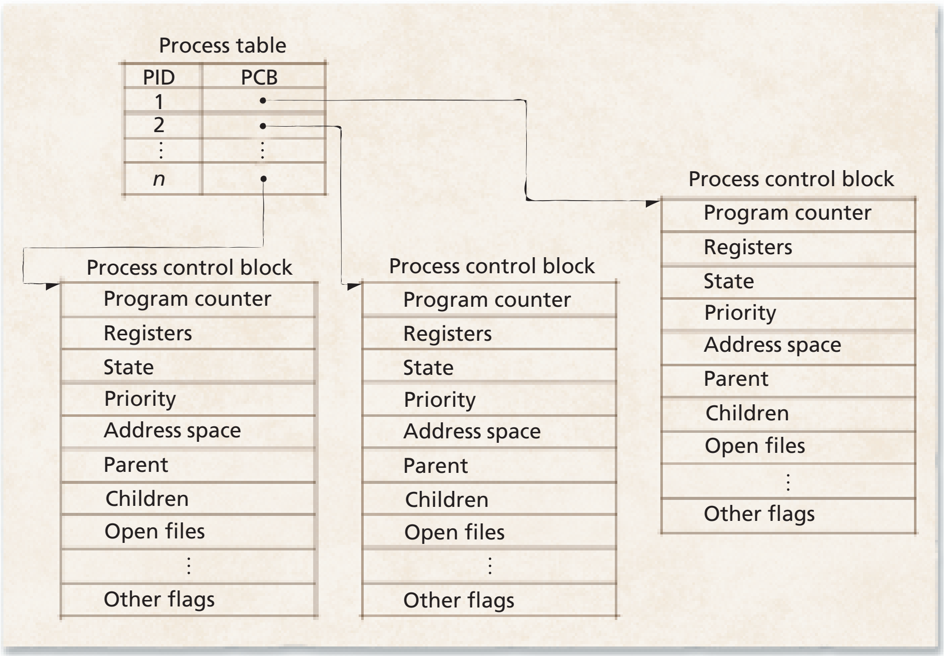


Figure 3.2 | Process table and process control blocks.

nated (either voluntarily or by the operating system), the operating system frees the process's memory and other resources, removes the process from the process table and makes its memory and other resources available to other processes. We discuss other process manipulation functions momentarily.[19]

Self Review

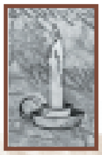
1. What is the purpose of the process table?
2. (T/F) The structure of a PCB is dependent on the operating system implementation.

Ans: **1)** The process table enables the operating system to locate each process's PCB. **2)** True.

3.3.3 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 interprocess communication).



Operating Systems Thinking

Data Structures in Operating Systems

Computer science students generally study data structures, both those on the main topic of a full course and as portions of many upper-level courses, such as compilers, databases, networking and operating systems. Data structures are used abundantly in operating systems. Queues are used wherever entities need to wait—processes waiting for a processor, I/O requests waiting for devices to become available, processes waiting for access to their critical sections and so on. Stacks are used for supporting the function call return mechanism. Trees are used to represent file system directory structures, to keep track of the allocation of disk space to files, to build hierarchical page directory structures in support of virtual address translation, and so on. Graphs are used when studying networking arrangements, deadlock resource allocation graphs, and the like. Hash tables are used to access PCBs quickly (using a PID as the key).

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. 3.3, 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).In UNIX-based systems, such as Linux, many processes are spawned from the *init* process, which is created when the kernel loads (Fig. 3.4). In Linux, such processes include *kswapd*, *xfs* and *khubd*—these processes perform memory, file system and device management operations, respectively. Many of these processes are discussed further in Chapter 20, Case Study: Linux. The *login* process authenticates users to the operating system. This is typically accomplished by requiring a user to enter a valid username and corresponding password. We discuss other means of authentication in Chapter 19, Security. Once the *login* process authenticates the user, it spawns a shell, such as *bash* (*B*ourne-*a*gain *sh*ell), that allows the user to interact with the operating system (Fig. 3.4). The user may then issue commands to the shell to execute programs such as *vi* (a text editor) and *finger* (a utility that displays user information). Destroying a process involves obliterating it from the system. Its memory and other resources are returned to the system, it is purged from any system lists or tables and its process control block is erased, i.e., the PCB's memory space is made available to other processes in the system. Destruction of a process is more complicated when the process has spawned other processes. In some operating systems, each spawned process is destroyed automatically when its parent is destroyed; in others, spawned processes proceed independently of their parents, and the destruction of a parent has no effect on its children.

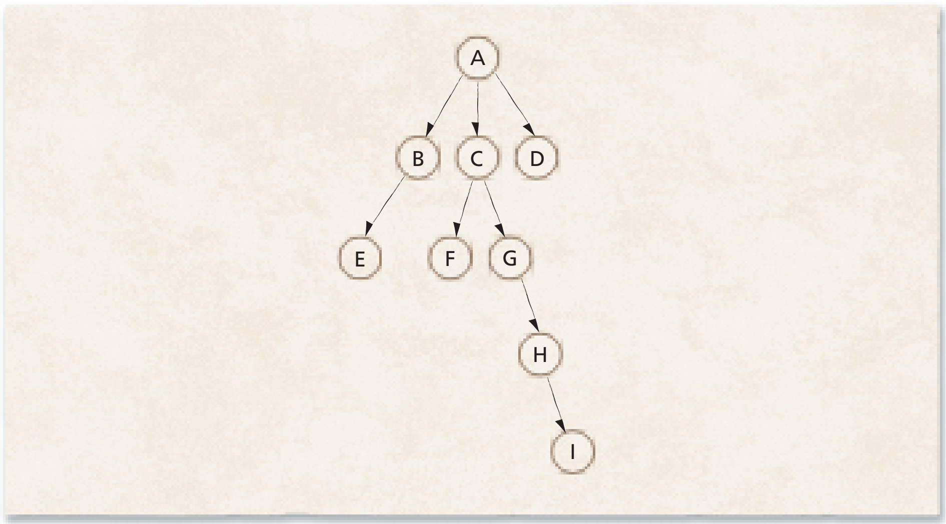


Figure 3.3 | *Process creation hierarchy.*

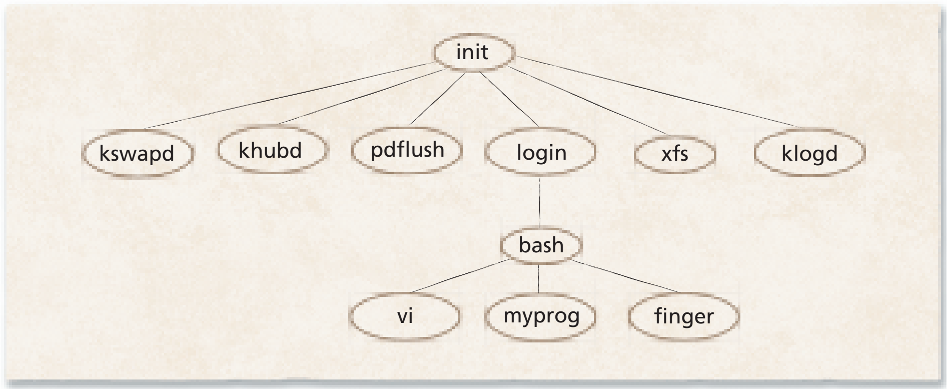


Figure 3.4 | Process hierarchy in Linux.

Changing the priority of a process normally involves modifying the priority value in the process's control block. Depending on how the operating system implements process scheduling, it may need to place a pointer to the PCB in a different priority queue (see Chapter 8, Processor Scheduling). The other operations listed in this section are explained in subsequent sections.

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.

3.3.4 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 3.5 displays the process state-transition diagram of Fig. 3.1 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 are the **suspended states**.

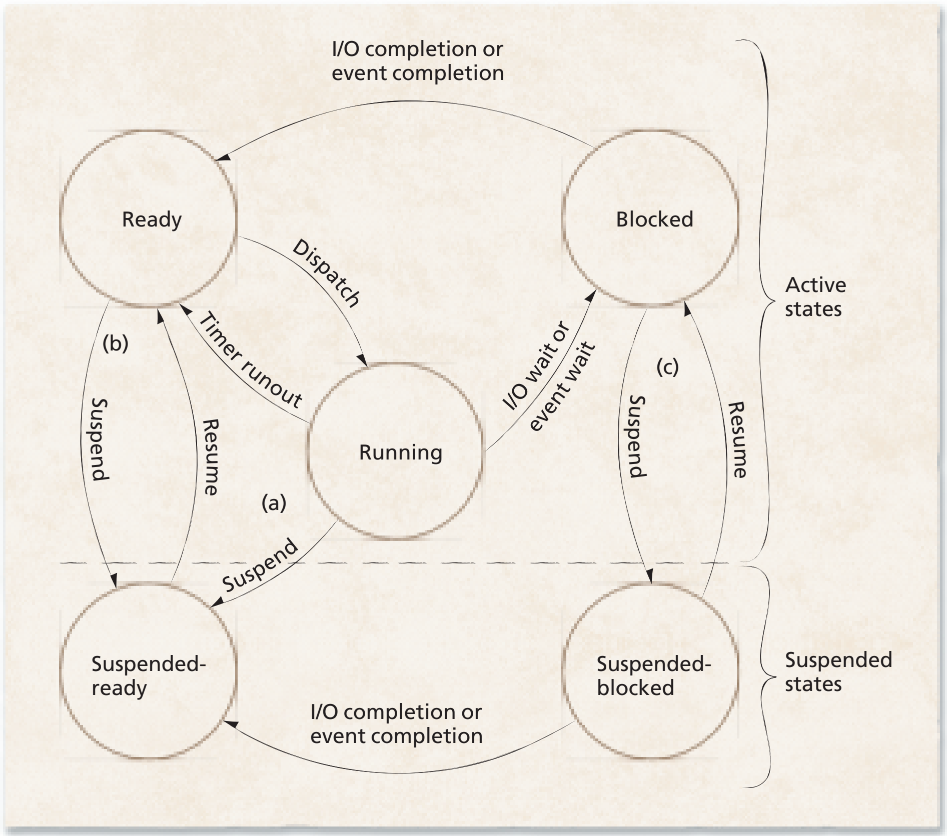


Figure 3.5 | Process state transitions with suspend and resume.

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. 3.5(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. 3.5(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 *suspendedready*. A *suspendedready* process may be made ready, or **resumed**, by another process, causing the first process to transition from *suspendedready* to *ready*. A *blocked* process will make the transition from *blocked* to *suspendedblocked* when it is suspended by another process. A *suspendedblocked* process may be resumed by another process and make the transition from *suspendedblocked* 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 *suspendedready* 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 *suspendedblocked* process makes the transition from *suspendedblocked* to *suspendedready*.

Self Review

1. In what three ways can a process get to the *suspendedready* state?
2. In what scenario is it best to suspend a process rather than abort it?

Ans: **1)** A process can get to the *suspendedready* 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.

3.3.5 Context Switching

The operating system performs a **context switch** to stop executing a *running* process and begin executing a previously *ready* process.[20] To perform a context switch, the kernel must first save the execution context of the *running* process to its PCB, then load the *ready* process's previous execution context from its PCB (Fig. 3.6).

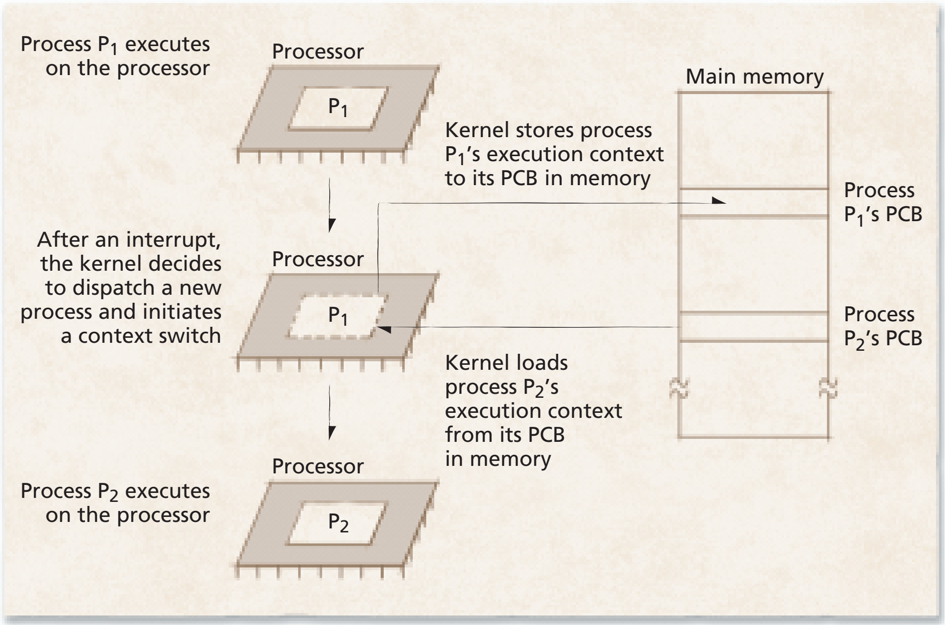

Process P1 executes on the processor
Processor
P1
Main memory
Kernel stores process P1's execution context to its PCB in memory
Process P1's PCB
After an interrupt, the kernel decides to dispatch a new process and initiates a context switch
Processor
P1
Process P2's PCB
Kernel loads process P2's execution context from its PCB in memory
Processor
Process P2 executes on the processor
P2

Figure 3.6 | Context switch.

Context switches, which are essential in a multiprogrammed environment, introduce several operating system design challenges. For one, context switches must be essentially transparent to processes, meaning that the processes are unaware they have been removed from the processor. During a context switch a processor cannot perform any "useful" computation—i.e., it performs tasks that are essential to operating systems but does not execute instructions on behalf of any given process. Context switching is pure overhead and occurs so frequently that operating systems must minimize context-switching time.

The operating system accesses PCBs often. As a result, many processors contain a hardware register that points to the PCB of the currently executing process to facilitate context switching. When the operating system initiates a context switch, the processor safely stores the currently executing process's execution context in the PCB. This prevents the operating system (or other processes) from overwriting the process's register values. Processors further simplify and speed context switching by providing instructions that save and restore a process's execution context to and from its PCB, respectively.

In the IA-32 architecture, the operating system dispatches a new process by specifying the location of its PCB in memory. The processor then performs a context switch by saving the execution context of the previously running process. The IA-32 architecture does not provide instructions to save and restore a process's execution context, because the processor performs these operations without software intervention.[21]

Self Review

1. From where does an operating system load the execution context for the process to be dispatched during a context switch?
2. Why should an operating system minimize the time required to perform a context switch?

Ans: **1)** The process to be dispatched has its context information stored in its PCB. **2)** During a context switch, a processor cannot perform instructions on behalf of processes, which can reduce throughput.

3.4 Interrupts

As discussed in Chapter 2, Hardware and Software Concepts, interrupts enable software to respond to signals from hardware. The operating system may specify a set of instructions, called an **interrupt handler**, to be executed in response to each type of interrupt. This allows the operating system to gain control of the processor to manage system resources.

A processor may generate an interrupt as a result of executing a process's instructions (in which case it is often called a **trap** and is said to be **synchronous** with the operation of the process). For example, synchronous interrupts occur when a process attempts to perform an illegal action, such as dividing by zero or referencing a protected memory location.

Interrupts may also be caused by some event that is unrelated to a process's current instruction (in which case they are said to be **asynchronous** with process execution; see the Operating Systems Thinking feature, Asynchronism vs. Synchronism). Hardware devices issue asynchronous interrupts to communicate a status change to the processor. For example, the keyboard generates an interrupt when a user presses a key; the mouse generates an interrupt when it moves or when one of its buttons is pressed.

Interrupts provide a low-overhead means of gaining the attention of a processor. An alternative to interrupts is for a processor to repeatedly request the status of each device. This approach, called **polling**, increases overhead as the complexity of the computer system increases. Interrupts eliminate the need for a processor to repeatedly poll devices.

A simple example of the difference between polling and interrupts can be seen in microwave ovens. A chef may either set a timer to expire after an appropriate number of minutes (the timer sounding after this interval interrupts the chef), or the chef may regularly peek through the oven's glass door and watch as the roast cooks (this kind of regular monitoring is an example of polling).

Interrupt-oriented systems can become overloaded—if interrupts arrive too quickly, the system may not be able to keep up with them. A human air traffic controller, for example, could easily be overwhelmed by a situation in which too many planes converged in a narrow area.

In networked systems, the network interface contains a small amount of memory in which it stores each packet of data that it receives from other computers.

Operating Systems Thinking

Asynchronism vs. Synchronism

When we say events occur asynchronously with the operation of a process, we mean that they happen independently of what is going on in the process. I/O operations can proceed concurrently and asynchronously with an executing process. Once the process initiates an asynchronous I/O operation, the process can continue executing while the I/O operation proceeds. When the I/O completes, the process is notified. That notification can come at any time. The process can deal with it at that moment or can proceed with other tasks and deal with the I/O-completion interrupt at an appropriate time. So interrupts are often characterized as an asynchronous mechanism. Polling is a synchronous mechanism. The processor repeatedly tests a device until the I/O is complete. Synchronous mechanisms can spend a lot of time waiting or retesting a device until an event occurs. Asynchronous mechanisms can proceed with other work and waste no time testing for events that have not happened, which generally improves performance.

Each time the network interface receives a packet, it generates an interrupt to inform a processor that data is ready for processing. If a processor cannot process data from the network interface before the interface's memory fills, packets might be lost. Systems typically implement queues to hold interrupts to be processed when a processor becomes available. These queues, of course, consume memory that is limited in size. Under heavy load, the system might not be able to enqueue all arriving interrupts, meaning that some could be lost.

Self Review

1. What does it mean for an interrupt to be synchronous?
2. What is an alternative to interrupts and why is it rarely used?

Ans: **1)** A synchronous interrupt occurs due to software execution. **2)** A system can perform polling, in which the processor periodically checks the status of devices. This technique is rarely used, because it creates significant overhead when the processor polls devices whose status has not changed. Interrupts eliminate this overhead by notifying a processor only when a device's status changes.

3.4.1 Interrupt Processing

We now consider how computer systems typically process hardware interrupts. (Note that there are other interrupt schemes.)

1. The interrupt line, an electrical connection between the mainboard and a processor, becomes active—devices such as timers, peripheral cards and controllers send signals that activate the interrupt line to inform a processor that an event has occurred (e.g., a period of time has passed or an I/O request has completed). Most processors contain an interrupt controller that orders interrupts according to their priority so that important interrupts are serviced first. Other interrupts are queued until all higher-priority interrupts have been serviced.
2. After the interrupt line becomes active, the processor completes execution of the current instruction, then pauses the execution of the current process. To pause process execution, the processor must save enough information so that the process can be resumed at the correct place and with the correct register information. In early IBM systems, this data was contained in a data structure called the program status word (PSW). In the Intel IA-32 architecture, such process state is referred to as the task state segment (TSS). The TSS is typically stored in a process's PCB.[22]
3. The processor then passes control to the appropriate interrupt handler. Each type of interrupt is assigned a unique value that the processor uses as an index into the **interrupt vector**, which is an array of pointers to interrupt handlers. The interrupt vector is located in memory that processes cannot access, so that errant processes cannot modify its contents.
4. The interrupt handler performs appropriate actions based on the type of interrupt.

5. After the interrupt handler completes, the state of the interrupted process (or some other "next process" if the kernel initiates a context switch) is restored.
6. The interrupted process (or some other "next process") executes. It is the responsibility of the operating system to determine whether the interrupted process or some other "next process" executes. This important decision, which can significantly impact the level of service each application receives, is discussed in Chapter 8, Processor Scheduling. For example, if the interrupt signaled an I/O completion event that caused a high-priority process to transition from *blocked* to *ready*, the operating system might preempt the interrupted process and dispatch the high-priority process.

Let us consider how the operating system and hardware interact in response to clock interrupts (Fig. 3.7). At each timer interval, the interrupting clock generates an interrupt that allows the operating system to execute to perform system management operations such as process scheduling. In this case, the processor is executing process P_1 (1) when the clock issues an interrupt (2). Upon receiving the interrupt, the processor accesses the interrupt vector entry that corresponds to the timer interrupt (3).

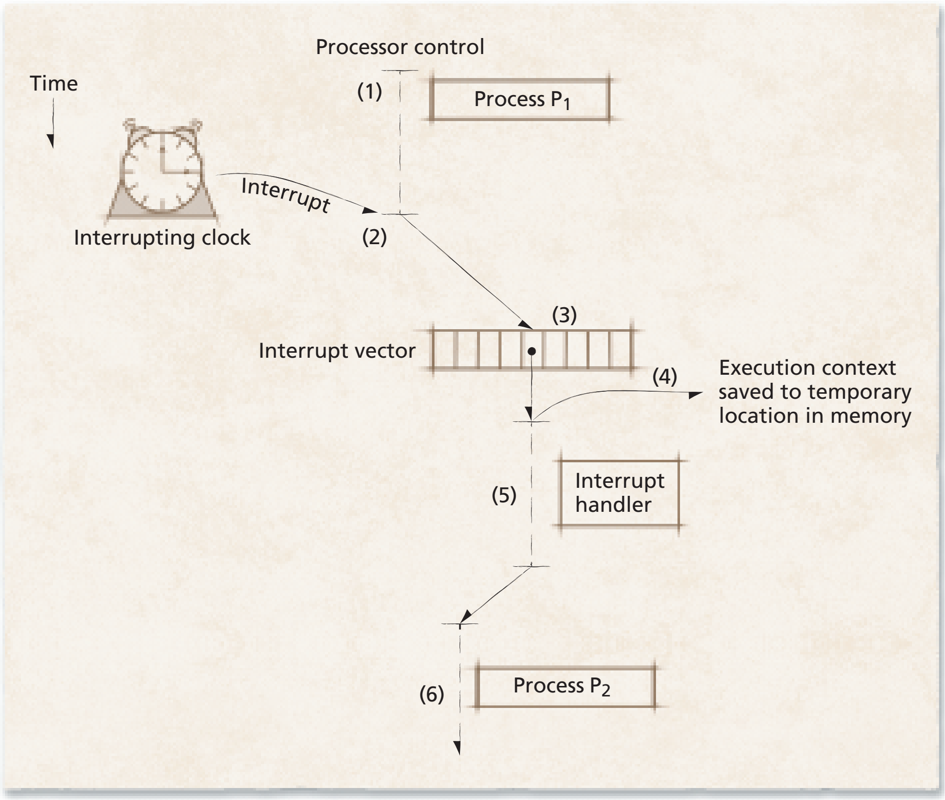


Figure 3.7 | *Handling interrupts.*

The processor then saves the process's execution context to memory (4) so that the P_1's execution context is not lost when the interrupt handler executes.[23] The processor then executes the interrupt handler, which determines how to respond to the interrupt (5). The interrupt handler may then restore the state of the previously executing process (P_1) or call the operating system processor scheduler to determine the "next" process to run. In this case, the handler calls the process scheduler, which decides that process P_2, the highest-priority waiting process, should obtain the processor (6). The context for process P_2 is then loaded from its PCB in main memory, and process P_1's execution context is saved to its PCB in main memory.

Self Review

1. Why are the locations of interrupt handlers generally not stored in a linked list?
2. Why is the process's execution context saved to memory while the interrupt handler executes?

Ans: **1)** To avoid becoming overwhelmed by interrupts, the system must be able to process each interrupt quickly. Traversing a linked list could significantly increase a system's response time if the number of interrupt types were large. Therefore, most systems use an interrupt vector (i.e., an array) to quickly access the location of an interrupt handler. **2)** If the process's execution context is not saved in memory, the interrupt handler could overwrite the process's registers.

3.4.2 Interrupt Classes

The set of interrupts a computer supports is dependent on the system's architecture. Several types of interrupts are common to many architectures; in this section we discuss the interrupt structure supported by the Intel IA-32 specification,[24] which is implemented in Intel® Pentium® processors. (Intel produced over 80 percent of the personal computer processors shipped in 2002.[25])

The IA-32 specification distinguishes between two types of signals a processor may receive: interrupts and exceptions. **Interrupts** notify the processor that an event has occurred (e.g., a timer interval has passed) or that an external device's status has changed (e.g., an I/O completion). The IA-32 architecture also provides **software-generated interrupts**—processes can use these to perform system calls. **Exceptions** indicate that an error has occurred, either in hardware or as a result of a software instruction. The IA-32 architecture also uses exceptions to pause a process when it reaches a breakpoint in code.[26]

Devices that generate interrupts, typically in the form of I/O signals and timer interrupts, are external to a processor. These interrupts are asynchronous with the running process, because they occur independently of instructions being executed by the processor. Software-generated interrupts, such as system calls, are synchronous with the running process, because they are generated in response to an instruction. Figure 3.8 lists several types of interrupts recognized by the IA-32 architecture.

Interrupt Type	Description of Interrupts in Each Type
I/O	These are initiated by the input/output hardware. They notify a processor that the status of a channel or device has changed. I/O interrupts are caused when an I/O operation completes, for example.
Timer	A system may contain devices that generate interrupts periodically. These interrupts can be used for tasks such as timekeeping and performance monitoring. Timers also enable the operating system to determine if a process's quantum has expired.
Interprocessor interrupts	These interrupts allow one processor to send a message to another in a multiprocessor system.

Figure 3.8 | *Common interrupt types recognized in the Intel IA-32 architecture.*

The IA-32 specification classifies exceptions as **faults**, **traps** or **aborts** (Fig. 3.9). Faults and traps are exceptions to which an exception handler can respond to allow processes to continue execution. A fault indicates an error that an exception handler can correct. For example, a page fault occurs when a process attempts to access data that is not in memory (we discuss page faults in Chapter 10, Virtual Memory Organization, and in Chapter 11, Virtual Memory Management). The operating system can correct this error by placing the requested data in main memory. After the problem is corrected, the processor restarts the process that caused the error at the instruction that caused the exception.

Traps do not typically correspond to correctable errors, but rather to conditions such as overflows or breakpoints. For example, as a process instructs a processor to increment the value in an accumulator, the value might exceed the capacity

Exception Class	Description of Exceptions in Each Class
Fault	These are caused by a wide range of problems that may occur as a program's machine-language instructions are executed. These problems include division by zero, data (being operated upon) in the wrong format, attempt to execute an invalid operation code, attempt to reference a memory location beyond the limits of real memory, attempt by a user process to execute a privileged instruction and attempt to reference a protected resource.
Trap	These are generated by exceptions such as overflow (when the value stored by a register exceeds the capacity of the register) and when program control reaches a breakpoint in code.
Abort	This occurs when the processor detects an error from which a process cannot recover. For example, when an exception-handling routine itself causes an exception, the processor may not be able to handle both errors sequentially. This is called a double-fault exception, which terminates the process that initiated it.

Figure 3.9 | *Intel IA-32 exception classes.*

of the accumulator. In this case, the operating system can simply notify the process that an overflow occurred. After executing the trap's exception handler, the processor restarts the process at the next instruction following the one that caused the exception.

Aborts indicate errors from which the process (or perhaps even the system) cannot recover, such as hardware failure. In this case, the processor cannot reliably save the process's execution context. Typically, as a result, the operating system terminates prematurely the process that caused the abort.

Most architectures and operating systems prioritize interrupts, because some require more immediate action than others. For example, responding to a hardware failure is more important than responding to an I/O-completion event. Interrupt priorities can be implemented in both hardware and software simultaneously. For example, a processor might block or queue interrupts of a lower priority than that of the interrupt the processor is currently handling. At times, the kernel can become so overloaded with interrupts that it can no longer respond to them. Rapid response to interrupts and quick return of control to interrupted processes is essential to maximizing resource utilization and achieving a high degree of interactivity. Most processors therefore allow the kernel to **disable** (or **mask**) an interrupt type. The processor may then ignore interrupts of that type or store them in a queue of pending interrupts that are delivered when that type of interrupt is reenabled. In the IA-32 architecture, the processor provides a register that indicates whether interrupts are disabled.[27]

Self Review

1. In the IA-32 architecture, what two types of signals can a processor receive?
2. In the IA-32 architecture, what is the difference between a fault and a trap?

Ans: **1)** A processor can receive interrupts or exceptions. Interrupts indicate that an event has occurred; exceptions indicate that an error has occurred. **2)** A fault restarts a process from the instruction that caused the exception. Faults are generally errors that can be corrected. A trap restarts a process at the next instruction following the one that caused the exception. Traps are usually generated by system calls and by the arrival of program control at breakpoints.

3.5 Interprocess Communication

In multiprogrammed and networked environments, it is common for processes to communicate with one another. Many operating systems provide mechanisms for interprocess communication (IPC) that, for example, enable a text editor to send a document to a print spooler or a Web browser to retrieve data from a distant server. Interprocess communication is also essential for processes that must coordinate (i.e., synchronize) activities to achieve a common goal. The case studies on Linux (see Section 20.10, Interprocess Communication) and Windows XP (see Section 21.10, Interprocess Communication) discuss how IPC is implemented in popular operating systems.

3.5.1 Signals

Signals are software interrupts that notify a process that an event has occurred. Unlike other IPC mechanisms we discuss, signals do not allow processes to specify data to exchange with other processes.[28] A system's signals depend on the operating system and the software-generated interrupts supported by a particular processor. When a signal occurs, the operating system first determines which process should receive the signal and how that process will respond to the signal.

Processes may catch, ignore or mask a signal. A process **catches** a signal by specifying a routine that the operating system calls when it delivers the signal.[29] A process may also **ignore** the signal. In this case, the process relies on the operating system's **default action** to handle the signal. A common default action is to **abort**, which causes the process to exit immediately. Another common default action is called a **memory dump**, which is similar to aborting. A memory dump causes a process to exit, but before doing so, the process generates a **core file** that contains the process's execution context and data from its address space, which is useful for debugging. A third default action is to simply ignore the signal. Two other default actions are to suspend and, subsequently, resume a process.[30]

A process can also block a signal by **masking** it. When a process masks a signal of a specific type (e.g., the suspend signal), the operating system does not deliver signals of that type until the process clears the signal mask. Processes typically block a signal type while handling another signal of the same type. Similar to masked interrupts, masked signals may be lost, depending on the operating system implementation.

Self Review

1. What is the major drawback of using signals for IPC?
2. What are the three ways in which a process can respond to a signal?

Ans: **1)** Signals do not support data exchange between processes. **2)** A process can catch, ignore or mask a signal.

3.5.2 Message Passing

With the increasing prominence of distributed systems, there has been a surge of interest in message-based interprocess communication.[31, 32, 33, 34, 35, 36] We discuss message-based communication in this section; particular implementations are discussed in the Linux and Windows XP case studies.[37, 38]

Messages can be passed in one direction at a time—for any given message, one process is the sender and the other is the receiver. Message passing may be bidirectional, meaning that each process can act as either a sender or a receiver while participating in interprocess communication. One model of message passing specifies that processes send and receive messages by making calls such as

```
send( receiverProcess, message );
receive( senderProcess, message );
```

The send and receive calls are normally implemented as system calls accessible from many programming language environments. A **blocking send** must wait for the receiver to receive the message, requiring that the receiver notify the sender when the message is received (this notification is called an acknowledgment). A **nonblocking send** enables the sender to continue with other processing even if the receiver has not yet received (and acknowledged) the message; this requires a message buffering mechanism to hold the message until the receiver receives it. A blocking send is an example of **synchronous communication**; a nonblocking send is an example of **asynchronous communication**. The send call may explicitly name a receiving process, or it may omit the name, indicating that the message is to be **broadcast** to all processes (or to some "working group" with which the sender generally communicates).

Asynchronous communication with nonblocking sends increases throughput by reducing the time that processes spend waiting. For example, a sender may send information to a busy print server; the system will buffer this information until the print server is ready to receive it, and the sender will continue execution without having to wait on the print server.

If no message has been sent, then a blocking receive call forces the receiver to wait; a nonblocking receive call enables the receiver to continue with other processing before it next attempts a receive. A receive call may specify that a message is to be received from a particular sender, or the receive may receive a message from any sender (or from any member of a group of senders).

A popular implementation of message passing is a **pipe**—a region of memory protected by the operating system that serves as a buffer, allowing two or more processes to exchange data. The operating system synchronizes access to the buffer—after a writer completes writing to the buffer (possibly filling it), the operating system pauses the writer's execution and allows a reader to read data from the buffer. As a process reads data, that data is removed from the pipe. When the reader completes reading data from the buffer (possibly emptying it), the operating system pauses the reader's execution and allows the writer to write data to the buffer.[39] Detailed treatments of pipes are provided in the Linux and Windows XP case studies at the end of the book. See Section 20.10.2, Pipes, and Section 21.10.1, Pipes, respectively.

In our discussions of interprocess communication between processes on the same computer, we always assumed flawless transmission. In distributed systems, on the other hand, transmissions can be flawed and even lost. So senders and receivers often cooperate using an **acknowledgment protocol** for confirming that each transmission has been properly received. A timeout mechanism can be used by the sender waiting for an acknowledgment message from the receiver; on timeout, if the acknowledgment has not been received, the sender can retransmit the message. Message passing systems with retransmission capabilities can identify each new message with a sequence number. The receiver can examine these numbers to be sure that it has received every message and to resequence out-of-sequence messages. If an acknowledgment message is lost and the sender decides to retransmit, it assigns the same sequence number to the retransmitted message as to the originally transmitted one. The receiver detecting several messages with the same sequence number knows to keep only one of them.

One complication in distributed systems with send/receive message passing is in naming processes unambiguously so that explicit send and receive calls reference the proper processes. Process creation and destruction can be coordinated through some centralized naming mechanism, but this can introduce considerable transmission overhead as individual machines request permission to use new names. An alternate approach is to have each computer ensure unique process names for its own processes; then processes may be addressed by combining the computer name with the process name. This, of course, requires centralized control in determining a unique name for each computer in a distributed system, which could potentially incur significant overhead if computers are frequently added and removed from the network. In practice, distributed systems pass messages between computers using numbered ports on which processes listen, avoiding the naming problem (see Chapter 16, Introduction to Networking).

As we will see in Chapter 17, Introduction to Distributed Systems, message-based communication in distributed systems presents serious security problems. One of these is the **authentication problem**: How do the senders and receivers know that they are not communicating with imposters who may be trying to steal or corrupt their data? Chapter 19, Security, discusses several authentication approaches.

There are several IPC techniques that we discuss later in the book. In addition to signals and pipes, processes may communicate via shared memory (discussed in Chapter 10, Virtual Memory Organization), sockets (discussed in Chapter 16, Introduction to Networking) and remote procedure calls (discussed in Chapter 17). They also may communicate to synchronize activities using semaphores and monitors., which are discussed in Chapter 5, Asynchronous Concurrent Execution, and Chapter 6, Concurrent Programming, respectively.

Self Review

1. Why do distributed systems rely on message passing instead of signals?
2. When a process performs a blocking send, it must receive an acknowledgment message to unblock. What problem might result from this scheme, and how can it be avoided?

Ans: **1)** Signals are typically architecture specific, meaning that signals supported by one computer may not be compatible with signals supported by another. Also, signals do not allow processes to transmit data, a capability required by most distributed systems. **2)** The sender may never receive an acknowledgment message, meaning that the process could be blocked indefinitely. This can be remedied by a timeout mechanism—if the sender does not receive an acknowledgment after a period of time, the send operation is assumed to have failed and it can be retried.

3.6 Case Study: UNIX Processes

UNIX and UNIX-based operating systems provide an implementation of processes that has been borrowed by many other operating systems (see the Mini Case Study, UNIX Systems). In this section, we describe the structure of UNIX processes, discuss several UNIX features that motivate the discussion in the following chapters and introduce how UNIX allows users to perform process management operations.

Each process must store its code, data and stack in memory during execution. In a real memory system, processes would locate such information by referencing a range of physical addresses. The range of valid main memory addresses for each process is determined by the size of main memory and the memory consumed by other processes. Because UNIX implements virtual memory, all UNIX processes are provided with a set of memory addresses, called a **virtual address space**, in which the process may store information. The virtual address space contains a text region, data region and stack region.[40]

The kernel maintains a process's PCB in a protected region of memory that user processes cannot access. In UNIX systems, a PCB stores information including the contents of processor registers, the process identifier (PID), the program counter and the system stack.[41, 42] The PCBs for all processes are listed in the process table, which allows the operating system to access information (e.g., priority) regarding every process.[43]

UNIX processes interact with the operating system via system calls. Figure 3.10 lists several of these. A process can spawn a child process by using the `fork` system call, which creates a copy of the parent process.[44, 45] The child process receives a copy of the parent process's data and stack segments and any other resources.[46, 47] The text segment, which contains the parent's read-only instructions, is shared with its child. Immediately following the `fork`, the parent and child process contain identical data and instructions. This means that the two processes must perform exactly the same actions unless either the parent or child can determine its identity. The `fork` system call therefore returns different values; the parent process

Mini Case Study

UNIX Systems

In the days before Windows, Macintosh, Linux or even DOS, operating systems typically worked on only one model of computer, managing system resources, running batch streams and little more.[48] From 1965 to 1969, a group of research teams from Bell Laboratories, General Electric and Project MAC at MIT developed the **Multics** operating system—a general-purpose computer utility, designed to be "all things to all people."[49] It was large, expensive and complex. In 1969, Bell Labs withdrew from the project and their own small team, led by Ken Thompson, began designing a more practical operating system to run machines at Bell Labs.

(Continued on the next page.)

Mini Case Study

UNIX Systems (Cont.)

Thompson implemented the basic components of the operating system, which Brian Kernighan named UNICS, a joke on the "multi" aspect of Multics; the spelling eventually changed to UNIX. Over the next few years, UNIX was rewritten in an interpreted implementation of Thompson's language B (based on Martin Richard's BCPL programming language), and soon after in Dennis Ritchie's faster, compiled C language.[50]

Due to a federal anti-trust lawsuit, AT&T (which owned Bell Labs) was not allowed to sell computer products, so they distributed UNIX source code to universities for a small fee to cover just the expense of producing the magnetic tapes. A group of students at the University of California at Berkeley, led by Bill Joy (later a cofounder of Sun Microsystems), modified the UNIX source code, evolving the operating system into what became known as **Berkeley Software Distribution UNIX (BSD UNIX)**.[51]

Industry software developers were drawn to UNIX because it was free, small and customizeable. To work with UNIX, developers had to learn C, and they liked it. Many of these developers also taught in colleges, and C gradually replaced Pascal as the preferred teaching language in college programming courses. Sun Microsystems based its SunOS on BSD UNIX, then later teamed up with AT&T to design the **Solaris** operating system based on AT&T's System V Release 4 UNIX.[52] A group of other UNIX developers, concerned that Sun's association with AT&T would give Sun an unfair business lead over other UNIX developers, formed the **Open Software Foundation (OSF)** to produce their own nonproprietary version of UNIX called **OSF/1**; the fierce competition between OSF and AT&T-backed Sun was dubbed the UNIX Wars.[53]

Several important operating systems are based on UNIX technology. Professor Andrew Tanenbaum of the Vrije Universiteit in Amsterdam built Minix in 1987, a stripped-down version of UNIX that was designed for teaching OS basics and is still used for this purpose in some college courses. Linus Torvalds, a Finnish graduate student, used Minix to begin writing the well-known open-source Linux operating system—now a whole family of systems in its own right (see Chapter 20, Case Study: Linux).[54] Linux is the most popular open-source operating system, and companies including IBM, Hewlett-Packard, Sun Microsystems and Intel all offer Linux versions as an operating system option for their servers. OpenBSD is another open-source project, led by Theo de Raadt, and is recognized as the most secure operating system available (see Chapter 19, Security).[55, 56, 57, 58] FreeBSD is also open-source and is known for its ease of use.[59] Yet another BSD descendant, NetBSD, has focused on portability to a variety of systems.[60, 61] IBM's AIX, based on both System V and BSD,[62] runs on some of IBM's servers. IBM claims AIX has a high degree of source-code compatibility with Linux.[63] Hewlett-Packard's HP-UX is becoming a strong competitor to AIX and Solaris, achieving the highest ratings in all the categories in a 2002 D.H. Brown Associates report, placing ahead of both Solaris and AIX.[64, 65, 66] receives the PID of the child and the child process receives a value of zero. This convention allows the child process to recognize that it is newly created. Application programmers can use this convention to specify new instructions for the child process to execute.

A process can call `exec` to load a new program from a file; `exec` is often performed by a child process immediately after it is spawned.[67] When the parent creates a child process, the parent can issue a `wait` system call, which blocks the parent until the specified child process terminates.[68] After a process has completed its work, it issues the `exit` system call. This tells the kernel that the process has finished; the kernel responds by freeing all of the process's memory and other resources, such as open files. When a parent process exits, its child processes are typically relocated in the process hierarchy to be children of the *init* process.[69, 70] If a parent process is terminated by a `kill` signal from another process, that signal is also sent to its child processes.

UNIX process priorities are integers between –20 and 19 (inclusive) that the system uses to determine which process will run next. A lower numerical priority value indicates a higher scheduling priority.[71] Processes that belong to the operating system, called kernel processes, often have negative integer values and typically have higher scheduling priority than user processes.[72] Operating system processes that perform maintenance operations periodically, called daemons, typically execute with the lowest possible priority.

Many applications require several independent components to communicate during execution, requiring interprocess communication (IPC). UNIX provides several mechanisms to enable processes to exchange data, such as signals and pipes (see Section 3.5, Interprocess Communication, and Section 20.10.2, Pipes).[73]

System Call	Description
`fork`	Spawns a child process and allocates to that process a copy of its parent's resources.
`exec`	Loads a process's instructions and data into its address space from a file.
`wait`	Causes the calling process to block until its child process has terminated.
`signal`	Allows a process to specify a signal handler for a particular signal type.
`exit`	Terminates the calling process.
`nice`	Modifies a process's scheduling priority.

Figure 3.10 | UNIX system calls.

Self Review

1. Why can a parent and its child share the parent's text segment after a `fork` system call?
2. Why must a process use IPC to share data with other processes?

Ans: **1)** The text segment contains instructions that cannot be modified by either process, meaning that both the parent and child process will execute the same instructions regardless of whether the operating system maintains one, or multiple, copies of the segment in memory. Therefore, the operating system can reduce memory consumption by sharing access to the text region between a parent and its child. **2)** The operating system does not allow unrelated processes to share the data segment of their address spaces, meaning that data stored by one process is inaccessible to an unrelated process. Therefore, the operating system must provide some mechanism to make data from one process available to another.

Web Resources

`msdn.microsoft.com/library/en-us/dllproc/base/about_processes_and_threads.asp`
Provides a description of processes in Windows XP.

`www.freebsd.org/handbook/basics-processes.html`
Includes a description of how the FreeBSD operating system handles processes. Mac OS X is based (in part) on FreeBSD.

`www.linux-tutorial.info/cgi-bin/display.pl?83&99980&0&3`
Discusses how Linux handles processes.

`docs.sun.com/db/doc/806-4125/6jd7pe6bg?a=view`
Provides a process state-transition diagram for Sun Microsystem's Solaris operating system.

`www.beyondlogic.org/interrupts/interupt.htm`
Overviews interrupts and provides a detailed description of interrupt handling in the Intel architecture.

`developer.apple.com/documentation/Hardware/DeviceManagers/pci_srvcs/pci_cards_drivers/PCI_BOOK.16d.html`
Documents the interrupt model in Apple Macintosh computers.

Summary

A process, which is a program in execution, is central to understanding how today's computer systems perform and keep track of many simultaneous activities. Each process has its own address space, which may consist of a text region, data region and stack region. A process moves through a series of discrete process states. For example, a process can be in the *running* state, *ready* state or *blocked* state. The ready list and blocked list store references to processes that are not *running*.

When a process reaches the head of the ready list, and when a processor becomes available, that process is given the processor and is said to make a transition from the *ready* state to the *running* state. The act of assigning a processor to the first process on the ready list is called dispatching. To prevent any one process from monopolizing the system, either accidentally or maliciously, the operating system sets a hardware interrupting clock (or interval timer) to allow a process to run for a specific time interval or quantum. If a *running* process initiates an input/output operation before its quantum expires, the process is said to block itself pending the completion of the I/O operation. Alternatively, the operating system can employ cooperative multitasking in which each process runs until completion or until it voluntarily relinquishes its processor. This can be dangerous, because cooperative multitasking does not prevent processes from monopolizing a processor.

The operating system typically performs several operations when it creates a process, including assigning a process identification number (PID) to the process and creating a process control block (PCB), or process descriptor, which stores the program counter (i.e., the pointer to the next instruction the process will execute), PID, scheduling priority and the process's execution context. The operating system maintains pointers to each process's PCB in the process table so that it can access PCBs quickly. When a process terminates (or is terminated by the operating system), the operating system removes the process from the process table and frees all of the process's resources, including its memory.

A process may spawn a new process—the creating process is called the parent process and the created process is called the child process. Exactly one parent process creates a child. Such creation yields a hierarchical process structure. In some systems, a spawned process is destroyed automatically when its parent is destroyed; in other systems, spawned processes proceed independently of their parents, and the destruction of a parent has no effect on the destroyed parent's children.

A suspended process is indefinitely removed from contention for time on the processor without being destroyed. The suspended states are *suspendedready* and *suspendedblocked*. A suspension may be initiated by the process being suspended or by another process; a suspended process must be resumed by another process.

When the operating system dispatches a ready process to a processor, it initiates a context switch. Context switches must be transparent to processes. During a context switch a processor cannot perform any "useful" computation, so operating systems must minimize context-switching time. Some architectures reduce overhead by performing context-switching operations in hardware.

Interrupts enable software to respond to signals from hardware. An interrupt may be specifically initiated by a running process (in which case it is often called a trap and said to be synchronous with the operation of the process), or it may be caused by some event that may or may not be related to the running process (in which case it is said to be asynchronous with the operation of the process). An alternative to interrupts is for the processor to repeatedly request the status of each device, an approach called polling.

Interrupts are essential to maintaining a productive and protected computing environment. When an interrupt occurs, the processor will execute one of the kernel's interrupt-handling functions. The interrupt handler determines how the system should respond to interrupts. The locations of the interrupt handlers are stored in an array of pointers called the interrupt vector. The set of interrupts a computer supports depends on the system's architecture. The IA-32 specification distinguishes between two types of signals a processor may receive: interrupts and exceptions.

Many operating systems provide mechanisms for interprocess communication (IPC) that, for example, enable a Web browser to retrieve data from a distant server. Signals are software interrupts that notify a process that an event has occurred. Signals do not allow processes to specify data to exchange with other processes. Processes may catch, ignore or mask a signal.

Message-based interprocess communication can occur in one direction at a time or it may be bidirectional. One model of message passing specifies that processes send and receive messages by making calls. A popular implementation of message passing is a pipe—a region of memory protected by the operating system that allows two or more processes to exchange data. One complication in distributed systems with send/receive message passing is in naming processes unambiguously so that explicit send and receive calls reference the proper processes.

UNIX processes are provided with a set of memory addresses, called a virtual address space, which contains a text region, data region and stack region. In UNIX systems, a PCB stores information including the contents of processor registers, the process identifier (PID), the program counter and the system stack. All processes are listed in the process table, which allows the operating system to access information regarding every process. UNIX processes interact with the operating system via system calls. A process can spawn a child process by using the `fork` system call, which creates a copy of the parent process. UNIX process priorities are integers between –20 and 19 (inclusive) that the system uses to determine which process will run next; a lower numerical priority value indicates a higher scheduling priority. The kernel also provides IPC mechanisms, such as pipes, to allow unrelated processes to transfer data.

Key Terms

abort—Action that terminates a process prematurely. Also, in the IA-32 specification, an error from which a process cannot recover.

Access Control List (ACL) (Multics)—Multics' discretionary access control implementation.

Access Isolation Mechanism (AIM) (Multics)—Multics' mandatory access control implementation.

address space—Set of memory locations a process can reference.

Berkeley Software Distribution (BSD) UNIX—UNIX version modified and released by a team led by Bill Joy at the University of California at Berkeley. BSD UNIX is the parent of several UNIX variations.

***blocked* state**—Process state in which the process is waiting for the completion of some event, such as an I/O completion, and cannot use a processor even if one is available.

blocked list—Kernel data structure that contains pointers to all *blocked* processes. This list is not maintained in any particular priority order.

child process—Process that has been spawned from a parent process. A child process is one level lower in the process hierarchy than its parent process. In UNIX systems, child processes are created using the `fork` system call.

concurrent program execution—Technique whereby processor time is shared among multiple active processes. On a uniprocessor system, concurrent processes cannot execute simultaneously; on a multiprocessor system, they can.

context switching—Action performed by the operating system to remove a process from a processor and replace it with another. The operating system must save the state of the process that it replaces. Similarly, it must restore the state of the process being dispatched to the processor.

cooperative multitasking—Process scheduling technique in which processes execute on a processor until they voluntarily relinquish control of it.

data region—Section of a process's address space that contains data (as opposed to instructions). This region is modifiable.

disable (mask) interrupts—When a type of interrupt is disabled (masked), interrupts of that type are not delivered to the process that has disabled (masked) the interrupts. The interrupts are either queued to be delivered later or dropped by the processor.

dispatcher—Operating system component that assigns the first process on the ready list to a processor.

exception—Hardware signal generated by an error. In the Intel IA-32 specification, exceptions are classified as traps, faults and aborts.

fault—In the Intel IA-32 specification, an exception as the result of an error such as division by zero or illegal access to memory. Some faults can be corrected by appropriate operating system exception handlers.

I/O completion interrupt—Message issued by a device when it finishes servicing an I/O request.

hierarchical process structure—Organization of processes when parent processes spawn child processes and, in particular, only one parent creates a child.

interrupt—Hardware signal indicating that an event has occurred. Interrupts cause the processor to invoke a set of software instructions called an interrupt handler.

interrupt handler—Kernel code that is executed in response to an interrupt.

interrupt vector—Array in protected memory containing pointers to the locations of interrupt handlers.

interrupting clock (interval timer)—Hardware device that issues an interrupt after a certain amount of time (called a quantum), e.g., to prevent a process from monopolizing a processor.

message passing—Mechanism to allow unrelated processes to communicate by exchanging data.

Multics Relational Data Store (MRDS)—First commercial relational database system, included in Multics.

Open Software Foundation (OSF)—Coalition of UNIX developers that built the OSF/1 UNIX clone to compete with AT&T's and Sun's Solaris. The OSF and the AT&T/Sun partnership were the participants in the UNIX Wars.

OSF/1—UNIX clone built by the Open Software Foundation to compete with Solaris.

parent process—Process that has spawned one or more child processes. In UNIX, this is accomplished by issuing a `fork` system call.

pipe—Message passing mechanism that creates a direct data stream between two processes.

polling—Technique to discover hardware status by repeatedly testing each device. Polling can be implemented in lieu of interrupts but typically reduces performance due to increased overhead.

process—Entity that represents a program in execution.

process control block (PCB)—Data structure containing information that characterizes a process (e.g., PID, address space and state); also called a process descriptor.

process descriptor—See process control block (PCB).

process identification number (PID)—Value that uniquely identifies a process.

process priority—Value that determines the importance of a process relative to other processes. It is often used to determine how a process should be scheduled for execution on a processor relative to other processes.

process state—Status of a process (e.g., *running*, *ready*, *blocked*, etc.).

process table—Data structure that contains pointers to all processes in the system.

program counter—Pointer to the instruction a processor is executing for a running process. After the processor completes the instruction, the program counter is adjusted to point to the next instruction the processor should execute.

quantum—Unit of time during which a process can execute before it is removed from the processor. Helps prevent processes from monopolizing processors.

***ready* state**—Process state from which a process may be dispatched to the processor.

ready list—Kernel data structure that organizes all *ready* processes in the system. The ready list is typically ordered by process scheduling priority.

resume—Remove a process from a suspended state.

***running* state**—Process state in which a process is executing on a processor.

signal—Message sent by software to indicate that an event or error has occurred. Signals cannot pass data to their recipients.

Solaris—UNIX version based on both System V Release 4 and SunOS, designed by AT&T and Sun collaboratively.

spawning a process—A parent process creating a child process.

stack region—Section of process's address space that contains instructions and values for open procedure calls. The contents of the stack grow as a process issues nested procedure calls and shrink as called procedures return.

state transition—Change of a process from one state to another.

***suspended* state**—Process state (either *suspendedblocked* or *suspendedready*) in which a 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.

***suspendedblocked* state**—Process state resulting from the process being suspended while in the *blocked* state. Resuming such a process places it into the *blocked* state.

***suspendedready* state**—Process state resulting from the process being suspended while in the *ready* state. Resuming such a process places it into the *ready* state.

text region—Section of a process's address space that contains instructions that are executed by a processor.

trap—In the IA-32 specification, an exception generated by an error such as overflow (when the value stored by a register exceeds the capacity of the register). Also generated when program control reaches a breakpoint in code.

unblock—Remove a process from the *blocked* state after the event on which it was waiting has completed.

virtual address space—Set of memory addresses that a process can reference. A virtual address space may allow a process to reference more memory than is physically available in the system.

Exercises

3.1 Give several definitions of process. Why, do you suppose, is there no universally accepted definition?

3.2 Sometimes the terms user and process are used interchangeably. Define each of these terms. In what circumstances do they have similar meanings?

3.3 Why does it not make sense to maintain the blocked list in priority order?

3.4 The ability of one process to spawn a new process is an important capability, but it is not without its dangers. Consider the consequences of allowing a user to run the process in Fig. 3.11. Assume that `fork()` is a system call that spawns a child process.

```
int main() {

   while( true ) {
      fork();
   }

}
```

Figure 3.11 | Code for Exercise 3.4.

a. Assuming that a system allowed such a process to run, what would the consequences be?

b. Suppose that you as an operating systems designer have been asked to build in safeguards against such processes. We know (from the "Halting Problem" of computability theory) that it is impossible, in the general case, to predict the path of execution a program will take. What are the consequences of this fundamental result from computer science on your ability to prevent processes like the above from running?

c. Suppose you decide that it is inappropriate to reject certain processes, and that the best approach is to place certain runtime controls on them. What controls might the operating system use to detect processes like the above at runtime?

d. Would the controls you propose hinder a process's ability to spawn new processes?

e. How would the implementation of the controls you propose affect the design of the system's process handling mechanisms?

3.5 In single-user dedicated systems, it is generally obvious when a program goes into an infinite loop. But in multiuser systems running large numbers of processes, it cannot easily be determined that an individual process is not progressing.

a. Can the operating system determine that a process is in an infinite loop?

b. What reasonable safeguards might be built into an operating system to prevent processes in infinite loops from running indefinitely?

3.6 Choosing the correct quantum size is important to the effective operation of an operating system. Later in the text we will consider the issue of quantum determination in depth. For now, let us anticipate some of the problems.

Consider a single-processor timesharing system that supports a large number of interactive users. Each time a process gets the processor, the interrupting clock is set to interrupt after the quantum expires. This allows the operating system to prevent any single process from monopolizing the processor and to provide rapid responses to interactive processes. Assume a single quantum for all processes on the system.

a. What would be the effect of setting the quantum to an extremely large value, say 10 minutes?

b. What if the quantum were set to an extremely small value, say a few processor cycles?

c. Obviously, an appropriate quantum must be between the values in (a) and (b). Suppose you could turn a dial and vary the quantum, starting with a small value and gradually increasing. How would you know when you had chosen the "right" value?

d. What factors make this value right from the user's standpoint?

e. What factors make it right from the system's standpoint?

3.7 In a block/wakeup mechanism, a process blocks itself to wait for an event to occur. Another process must detect that the event has occurred, and wake up the blocked process. It is possible for a process to block itself to wait for an event that will never occur.

a. Can the operating system detect that a blocked process is waiting for an event that will never occur?

b. What reasonable safeguards might be built into an operating system to prevent processes from waiting indefinitely for an event?

3.8 One reason for using a quantum to interrupt a running process after a "reasonable" period is to allow the operating system to regain the processor and dispatch the next process. Suppose that a system does not have an interrupting clock, and the only way a process can lose the processor is to relinquish it voluntarily. Suppose also that no dispatching mechanism is provided in the operating system.

a. Describe how a group of user processes could cooperate among themselves to effect a user-controlled dispatching mechanism.

b. What potential dangers are inherent in this scheme?

c. What are the advantages to the users over a system-controlled dispatching mechanism?

3.9 In some systems, a spawned process is destroyed automatically when its parent is destroyed; in other systems, spawned processes proceed independently of their parents, and the destruction of a parent has no effect on its children.

a. Discuss the advantages and disadvantages of each approach.

b. Give an example of a situation in which destroying a parent should specifically not result in the destruction of its children.

3.10 When interrupts are disabled on most devices, they remain pending until they can be processed when interrupts are again enabled. No further interrupts are allowed. The functioning of the devices themselves is temporarily halted. But in real-time systems, the environment that generates the interrupts is often disassociated from the computer system. When interrupts are disabled on the computer system, the environment keeps on generating interrupts anyway. These interrupts can be lost.

a. Discuss the consequences of lost interrupts.

b. In a real-time system, is it better to lose occasional interrupts or to halt the system temporarily until interrupts are again enabled?

3.11 As we will see repeatedly throughout this text, management of waiting is an essential part of every operating system. In this chapter we have seen several waiting states, namely *ready, blocked, suspendedready* and *suspendedblocked*. For each of these states discuss how a process might get into the state, what the process is waiting for, and the likelihood that the process could get "lost" waiting in the state indefinitely.

What features should operating systems incorporate to deal with the possibility that processes could start to wait for an event that might never happen?

3.12 Waiting processes do consume various system resources. Could someone sabotage a system by repeatedly creating processes and making them wait for events that will never happen? What safeguards could be imposed?

3.13 Can a single-processor system have no processes ready and no process running? Is this a "dead" system? Explain your answer.

3.14 Why might it be useful to add a *dead* state to the state-transition diagram?

3.15 System A runs exactly one process per user. System B can support many processes per user. Discuss the organizational differences between operating systems A and B with regard to support of processes.

3.16 Compare and contrast IPC using signals and message passing.

3.17 As discussed in Section 3.6, Case Study: UNIX Processes, UNIX processes may change their priority using the `nice` system call. What restrictions might UNIX impose on using this system call, and why?

Suggested Projects

3.18 Compare and contrast what information is stored in PCBs in Linux, Microsoft's Windows XP, and Apple's OS X. What process states are defined by each of these operating systems?

3.19 Research the improvements made to context switching over the years. How has the amount of time processors spend on context switches improved? How has hardware helped to make context switching faster?

3.20 The Intel Itanium line of processors, which are designed for high-performance computing, are implemented according to the IA-64 (64-bit) specification. Compare and contrast the IA-32 architecture's method of interrupt processing discussed in this chapter with that of the IA-64 architecture (see `developer.intel.com/design/itanium/manuals/245318.pdf`).

3.21 Discuss an interrupt scheme other than that described in this chapter. Compare the two schemes.

Recommended Reading

Randell and Horning describe basic process concepts as well as management of multiple processes.[74] Lampson considers some basic process concepts, including context switching.[75] Quarterman, Silberschatz and Peterson discuss how BSD UNIX 4.3 manages processes.[76] The case studies in Part 8 describe how Linux and Windows XP implement and manage processes. Information on how processors handle interrupts is available in computer architecture books such as *Computer Organization and Design* by Patterson and Hennessey.[77]

Interprocess communication is a focal point of microkernel (see Section 1.13.3, Microkernel Architecture) and exokernel operating system architectures. Because the performance of such systems relies heavily on the efficient operation of IPC mechanisms, considerable research has been devoted to the topic of improving IPC performance. Early message-based architectures[78] such as Chorus were distributed operating systems, but as early as 1986, microkernels such as Mach[79] were developed. IPC performance improvement to support effective microkernels is reflected in the literature.[80, 81]

IPC mechanisms differ from one operating system to another. Most UNIX systems share the mechanisms presented in this chapter. Descriptions of such implementations can be found in the Linux manual pages, but the reader is encouraged to read Chapter 20, Case Study: Linux, first. *Understanding the Linux Kernel*[82] discusses IPC in Linux and *Inside Windows 2000*[83] discusses IPC in Windows 2000 (almost all of which applies to Windows XP as well). The bibliography for this chapter is located on our Web site at `www.deitel.com/books/os3e/Bibliography.pdf`.

Works Cited

1. Daley, Robert C., and Jack B. Dennis, "Virtual Memory, Processes, and Sharing in Multics," *Proceedings of the ACM Symposium on Operating System Principles*, January 1967.
2. Corbató, F.; M. Merwin-Daggett, and R.C. Daley, "An Experimental Time-Sharing System," *Proceedings of the Spring Joint Computer Conference (AFIPS)*, Vol. 21, 1962, pp. 335–344.
3. Van Vleck, T., "The IBM 7094 and CTSS," March 3, 2003, <www.multicians.org/thvv/7094.html>.
4. Van Vleck, T., "Multics General Information and FAQ," September 14, 2003, <www.multicians.org/general.html>.
5. Van Vleck, T., "Multics General Information and FAQ," September 14, 2003, <www.multicians.org/general.html>.
6. Green, P., "Multics Virtual Memory: Tutorial and Reflections," 1993, <ftp://ftp.stratus.com/pub/vos/multics/pg/mvm.html>.
7. Van Vleck, T., "Multics General Information and FAQ," September 14, 2003, <www.multicians.org/general.html>.
8. Green, P., "Multics Virtual Memory: Tutorial and Reflections," 1993, <ftp://ftp.stratus.com/pub/vos/multics/pg/mvm.html>.
9. Van Vleck, T., "Multics General Information and FAQ," September 14, 2003, <www.multicians.org/general.html>.
10. Van Vleck, T., "Multics Glossary—A," <www.multicians.org/mga.html>.
11. Van Vleck, T., "Multics Glossary—B," <www.multicians.org/mgb.html>.
12. McJones, P., "Multics Relational Data Store (MRDS)," <www.mcjones.org/System_R/mrds.html>.
13. Peterson, J. L.; J. S. Quarterman; and A. Silbershatz, "4.2BSD and 4.3BSD as Examples of the UNIX System," *ACM Computing Surveys,* Vol. 17, No. 4, December 1985, p. 388.
14. MIT Laboratory for Computer Science, "Prof. F. J. Corbató," April 7, 2003, <www.lcs.mit.edu/people/bioprint.php3?PeopleID=86>.
15. F. Corbató, M. Merwin-Daggett, and R.C. Daley, "An Experimental Time-Sharing System," *Proc. Spring Joint Computer Conference (AFIPS)*, 335–344.
16. MIT Laboratory for Computer Science, "Prof. F. J. Corbató," April 7, 2003, <www.lcs.mit.edu/people/bioprint.php3?PeopleID=86>.
17. MIT Laboratory for Computer Science, "Prof. F. J. Corbató," April 7, 2003, <www.lcs.mit.edu/people/bioprint.php3?PeopleID=86>.
18. F. Corbató, "On Building Systems That Will Fail," *Communications of the ACM*, Vol. 34, No. 9, September 1991, pp. 72–81.
19. "UNIX System Calls Links," <www.softpanorama.org/Internals/unix_system_calls_links.shtml>.
20. Lampson, B. W., "A Scheduling Philosophy for Multiprocessing System," *Communications of the ACM,* Vol. 11, No. 5, 1968, pp. 347–360.
21. *IA-32 Intel Architecture Software Developer's Manual,* Vol. 3*, System Programmer's Guide*, 2002, pp. 6-1–6-15.
22. *IA-32 Intel Architecture Software Developer's Manual,* Vol. 3*, System Programmer's Guide*, 2002.
23. *IA-32 Intel Architecture Software Developer's Manual,* Vol. 3*, System Programmer's Guide*, 2002, pp. 5-16.
24. *IA-32 Intel Architecture Software Developer's Manual,* Vol. 3*, System Programmer's Guide*, 2002.
25. Krazit, T., "Study: Intel's Q2 Market Share Up, AMD's Down," *InfoWorld*, July 31, 2002, <archive.infoworld.com/articles/hn/xml/02/07/31/020731hnstudy.xml>.
26. *IA-32 Intel Architecture Software Developer's Manual,* Vol. 3*, System Programmer's Guide*, 2002, pp. 5–2, 5–5.
27. *IA-32 Intel Architecture Software Developer's Manual,* Vol. 3*, System Programmer's Guide*, 2002, pp. 2-8.
28. Bar, M., "Kernel Korner: The Linux Signals Handling Model," *Linux Journal*, May 2000, <www.linuxjournal.com/article.php?sid=3985>.
29. Bovet, D., and M. Cesati, *Understanding the Linux Kernel,* O'Reilly, 2001, p. 253.
30. Bar, M., "Kernel Korner: The Linux Signals Handling Model," *Linux Journal,* May 2000, <www.linuxjournal.com/article.php?sid=3985>.
31. Gentleman, W. M., "Message Passing Between Sequential Processes: The Reply Primitive and the Administrator Concept," *Software—Practice and Experience*, Vol. 11, 1981, pp. 435–466.
32. Schlichting, R. D., and F. B. Schneider, "Understanding and Using Asynchronous Message Passing Primitives," in *Proceedings of the Symposium on Principles of Distributed Computing*, August 18–20, 1982, Ottawa, Canada, ACM, New York, pp. 141–147.
33. Stankovic, J. A., "Software Communication Mechanisms: Procedure Calls versus Messages," *Computer*, Vol. 15, No. 4, April 1982.
34. Staustrup, J., "Message Passing Communication versus Procedure Call Communication," *Software—Practice and Experience*, Vol. 12, No. 3, March 1982, pp. 223–234.
35. Cheriton, D. R., "An Experiment Using Registers for Fast Message-Based Interprocess Communications," *Operating Systems Review*, Vol. 18, No. 4, October 1984, pp. 12–20.
36. Olson, R., "Parallel Processing in a Message-Based Operating System," *IEEE Software*, Vol. 2, No. 4, July 1985, pp. 39–49.
37. Andrews, G. R., "Synchronizing Resources," *ACM Transactions on Programming Languages and Systems*, Vol. 3, No. 4, October 1981, pp. 405–430.
38. Andrews, G., and F. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, Vol. 15, No. 1, March 1983, pp. 3–44.
39. Bovet, D., and M. Cesati, *Understanding the Linux Kernel,* O'Reilly, 2001, pp. 524-532.

40. Thompson, K., "UNIX Implementation," *UNIX Programer's Manual: 7th ed., Vol. 2b,* January 1979, <cm.bell-labs.com/7thEdMan/bswv7.html>.

41. Thompson, K., "UNIX Implementation," *UNIX Programer's Manual: 7th ed., Vol. 2b,* January 1979, <cm.bell-labs.com/7thEdMan/bswv7.html>.

42. *FreeBSD Handbook: Processes*, 2002, <*www.freebsd.org/handbook/basics-processes.html*>.

43. Thompson, K., "UNIX Implementation," *UNIX Programer's Manual: 7th ed., Vol. 2b,* January 1979, <cm.bell-labs.com/7thEdMan/bswv7.html>.

44. Thompson, K., "UNIX Implementation," *UNIX Programer's Manual: 7th ed., Vol. 2b,* January 1979, <cm.bell-labs.com/7thEdMan/bswv7.html>.

45. Ritchie, D., and K. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM,* July 1974, pp. 370-372.

46. *FreeBSD Handbook: Processes*, 2002, <www.freebsd.org/handbook/basics-processes.html>.

47. Thompson, K., "UNIX Implementation," *UNIX Programer's Manual: 7th ed., Vol. 2b,* January 1979, <cm.bell-labs.com/7thEdMan/bswv7.html>.

48. Lucent Technologies, "The Creation of the UNIX Operating System," 2002, <www.bell-labs.com/history/unix/>.

49. Organick, E., *The Multics System: An Examination of Its Structure*, Cambridge, MA: MIT Press, 1972.

50. Lucent Technologies, "The Creation of the UNIX Operating System," 2002, <www.bell-labs.com/history/unix/>.

51. Sun Microsystems, "Executive Bios: Bill Joy," <www.sun.com/aboutsun/media/ceo/mgt_joy.html>.

52. Calkins, B., "The History of Solaris," December 15, 2001, <unixed.com/Resources/history_of_solaris.pdf>.

53. Lucent Technologies, "The Creation of the UNIX Operating System," 2002, <www.bell-labs.com/history/unix/>.

54. Torvalds, L., "Linux History," July 31, 1992, <www.li.org/linuxhistory.php>.

55. Holland, N., "1—Introduction to OpenBSD," July 23, 2003, <www.openbsd.org/faq/faq1.html>.

56. Howard, J., "Daemon News: The BSD Family Tree," April 2001, <www.daemonnews.org/200104/bsd_family.html>.

57. Jorm, D., "An Overview of OpenBSD Security," August 8, 2000, <www.onlamp.com/pub/a/bsd/2000/08/08/OpenBSD.html>.

58. Security Electronics Magazine, "OpenBSD: Secure by Default," January 2002, <www.semweb.com/jan02/itsecurityjan.htm>.

59. Howard, J., "Daemon News: The BSD Family Tree," April 2001, <www.daemonnews.org/200104/bsd_family.html>.

60. The NetBSD Foundation, Inc., "About the NetBSD Project," July 17, 2003, <www.netbsd.org/Misc/about.html>.

61. Howard, J., "Daemon News: The BSD Family Tree," April 2001, <www.daemonnews.org/200104/bsd_family.html>.

62. Coelho, J., "comp.aix.unix Frequently Asked Questions (Part 1 of 5)," October 10, 2000, <www.faqs.org/faqs/aix-faq/part1/>.

63. IBM, "AIX Affinity with Linux: Technology Paper," <www-1.ibm.com/servers/aix/products/aixos/linux/affinity_linux.pdf>.

64. Springer, I., "comp.sys.hp.hpux FAQ," September 20, 2003, <www.faqs.org/faqs/hp/hpux-faq/>.

65. Hewlett-Packard Company, "HP-UX 11i Operating System," <www.hp.com/products1/unix/operating/>.

66. Hewlett-Packard Company, "Hewlett Packard Receives Top UNIX Ranking from D.H. Brown," May 30, 2002, <www.hp.com/hpinfo/newsroom/press/30may02b.htm>.

67. *FreeBSD Hypertext Man Pages*, 2002 <www.freebsd.org/cgi/man.cgi?query=execve&sektion=2>.

68. Ritchie, D., and K. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM,* July 1974, pp. 370-372.

69. Ritchie, D., and K. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM,* July 1974, pp. 370-372.

70. "Exit," *The Open Group Base Specifications, Issue 6, IEEE Std 1003.1*, 2003 Edition, <www.opengroup.org/onlinepubs/007904975/functions/_Exit.html>.

71. *UNIXhelp for Users, Version 1.3.2: Nice*, <unixhelp.ed.ac.uk/CGI/man-cgi?nice>.

72. Thompson, K., "UNIX Implementation," *UNIX Programer's Manual: 7th ed., Vol. 2b,* January 1979, <cm.bell-labs.com/7thEdMan/bswv7.html>.

73. *The Design and Implementation of 4.4BSD Operating System: Interprocess Communication*, 2002, <www.freebsd.org/doc/en_US.ISO8859-1/books/design-44bsd/x659.html>.

74. Horning, J. J., and B. Randell, "Process Structuring," *ACM Computing Surveys*, Vol. 5, No. 1, March 1973, pp. 5–29.

75. Lampson, B. W., "A Scheduling Philosophy for Multiprocessing System," *Communications of the ACM*, Vol. 11, No. 5, 1968, pp. 347–360.

76. Peterson, J. L.; J. S. Quarterman; and A. Silbershatz, "4.2BSD and 4.3BSD as Examples of the UNIX System," *ACM Computing Surveys,* Vol. 17, No. 4, December 1985, p. 388.

77. Hennessy, J., and D. Patterson, *Computer Organization and Design,* San Francisco: Morgan Kaufmann Publishers, Inc., 1998.

78. Guillemont, M., "The Chorus Distributed Operating System: Design and Implementation," *Proceedings of the ACM International Symposium on Local Computer Networks*, Firenze, April 1982, pp. 207–223.

79. Accetta, M. J.; R. V. Baron; W. Bolosky; D. B. Golub; R. F. Rashid; A. Tevanian; and M. W. Young, "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the Usenix Summer'86 Conference*, Atlanta, Georgia, June 1986, pp. 93–113.

80. Liedtke, Jochen, "Improving IPC by Kernel Design," *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, December 5–8, 1993, pp.175–188.

81. Liedtke, Jochen, "Toward Real Microkernels," *Communications of the ACM*, Vol. 39, No. 9, September 1996, pp. 70–77.

82. Bovet, D., and M. Cesati, *Understanding the Linux Kernel*, O'Reilly, 2001, p. 253.

83. Solomon, D., and M. Russinovich, *Inside Windows 2000,* 3d ed., Redmond: Microsoft Press, 2000.