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 paral- lel-or, as we will say, concurrently. Respiration, blood circulation, thinking, walk- ing 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 con- currently (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 pro- cess. 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 Multicast system in the 1960s (sce the Mini Case Study, CTSS and Multicast and the Biographical Note, Fernando J. Corbató). 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 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 uses during execution.

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 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 pro- cess is said to be blocked (i.e., in the blocked state) if it is waiting for some event to happen (such as an VO 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 pro- cesses and a blocked list of blocked processes. The ready list is maintained in prior- ity 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 funda- mental services to processes-these include creating processes, destroying pro- cesses, suspending processes, resuming processes, changing a process's priority, blocking processes, waking up processes, dispatching processes, enabling processes to interact via interposes 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 tran- sition from the ready state to the running state (Fig. 3.1). The act of assigning a pro- cessor 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 mali- ciously, the operating system sets a hardware interrupting clock (also called an inter- val 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 con- trol 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 ini- tiates 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

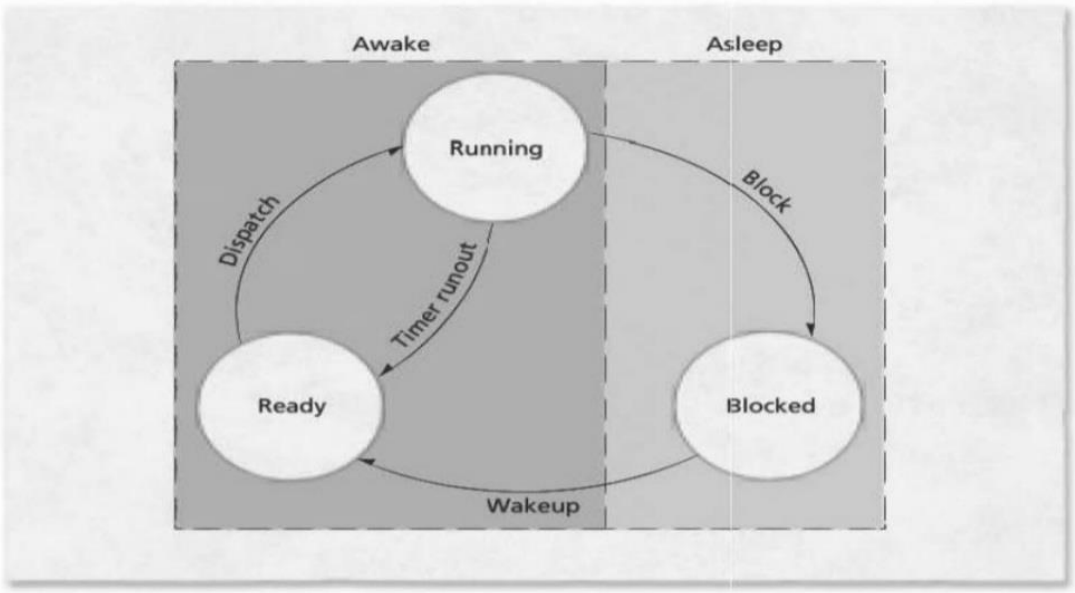

Figure 3.1 | Process state transitions.

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 transi- tions 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 voluntar- ily 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 Proces Control Blocks (PCBS)/Process Descriptors

The operating system typically performs several operations when it creates a pro- cess. 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 proces- sor 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 pro- cess) 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 run- ning 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 operat- ing 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 main- tains 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 oper- ating system data structures we discuss in this text (see the Operating Systems Thinking feature, Data Structures in Operating Systems). When a process is termi-

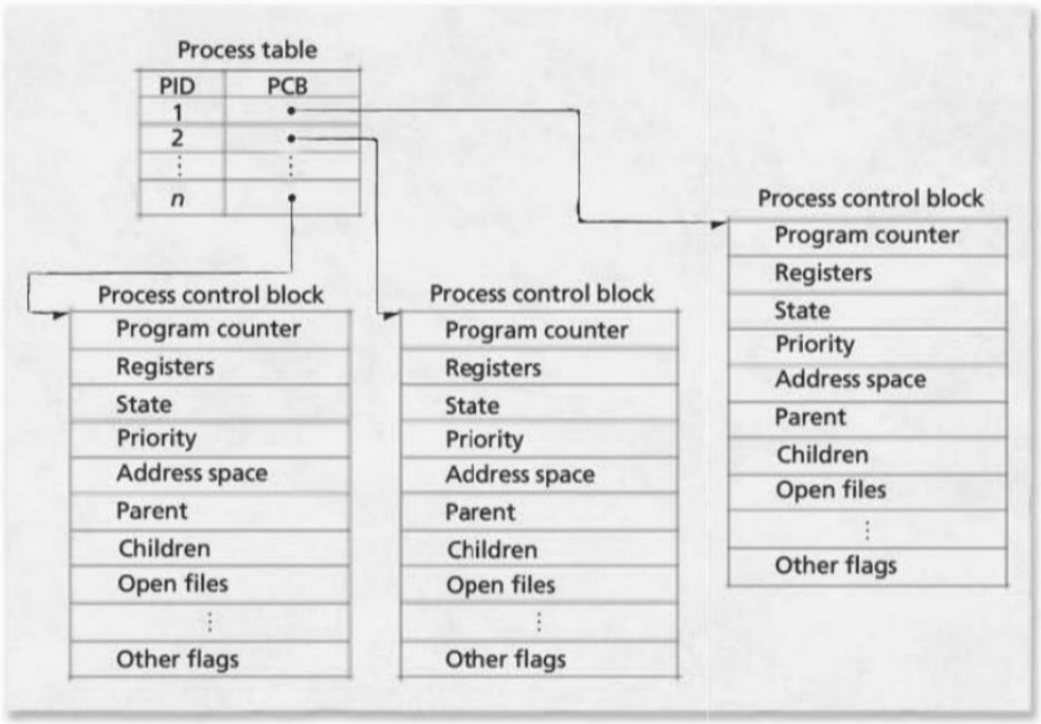

Figure 3.2 | Process table and process control blocks.

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 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 inter- process communication).

A process may spawn a new process. If it does, the creating process is called the parent process and the created process is called the child process. Each child process is created by exactly one parent process. Such creation yields a hierarchical process structure similar to Fig. 3.3, 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 pro- cess, which is created when the kernel loads (Fig. 3.4). In Linux, such processes 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 sys- tem 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 operat- ing 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.

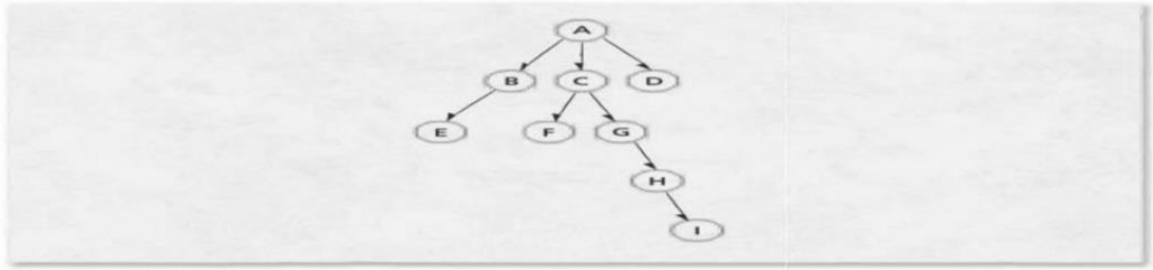

Figure 3.3 | Process creation hierarchy.

Changing the priority of a process normally involves modifying the priority value in the process's control block. Depending on how the operating system imple- ments 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.

3,3.4 Suspend and Resume

Many operating systems allow administrators, users or processes to suspend a pro- cess. 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.

7.7.4 Suspend and Resume

Many operating systems allow administrators, users or processes to suspend a pro- cess. 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, suspend- already and suspendedblocked. Above the dashed line in the figure are the active states; below it are the suspended states.

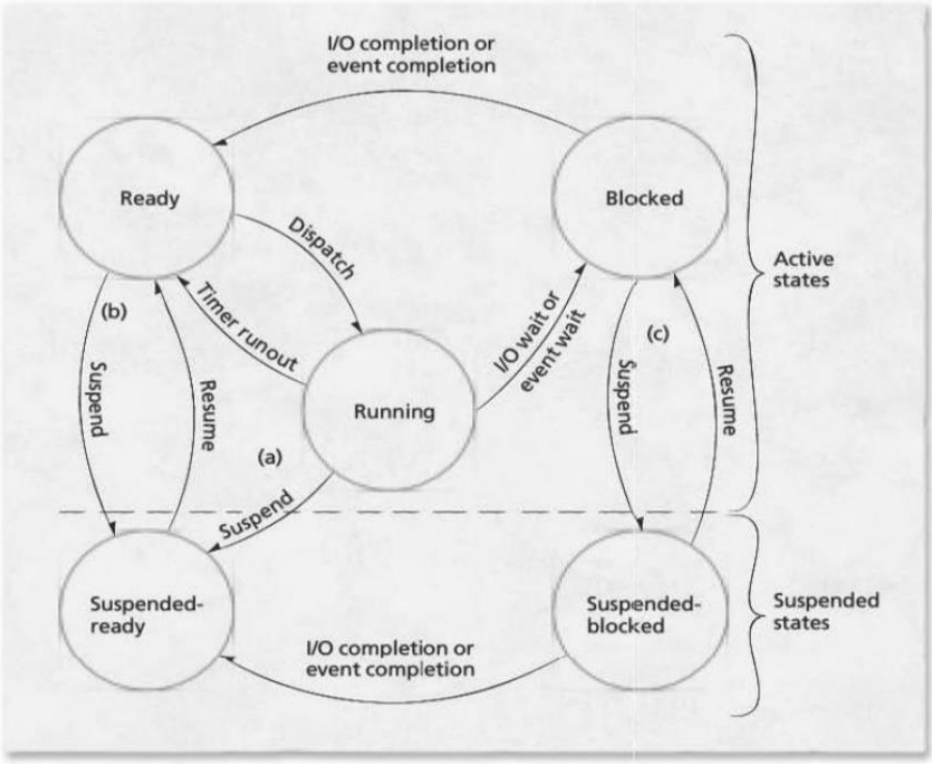

Figure 3.5 | Process state transitions with suspend and resume.

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 suspend- edready. A suspendedready process may be made ready, or resumed, by another pro- cess, causing the first process to transition from suspendedready to ready. A blocked process will make the transition from blocked to suspendedblocked when it is sus- pended 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. Unfortu- nately, 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).

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.

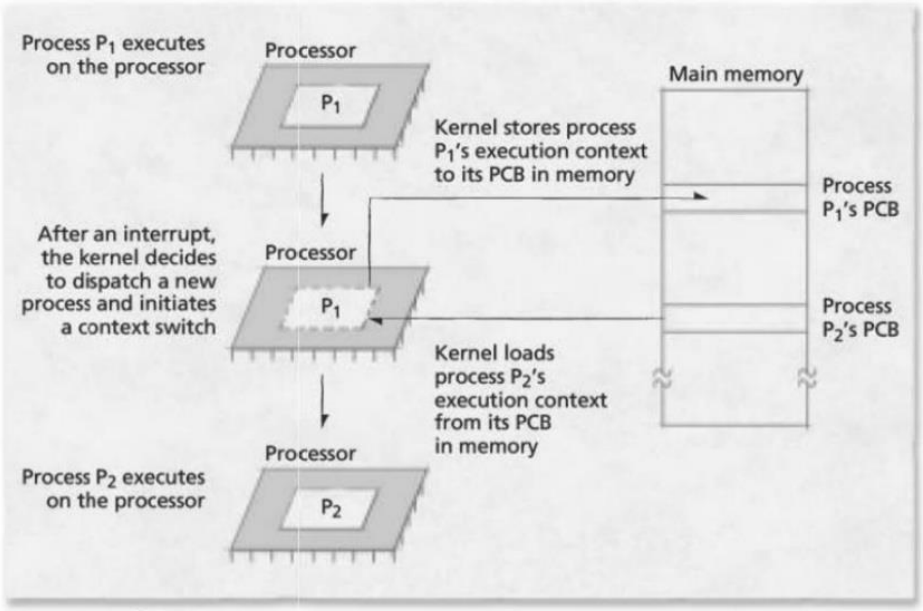

Figure 3.6 | Context switch.

The operating system accesses PCBS often. As a result, many processors con- tain 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 switch- ing by providing instructions that save and restore a process's execution context to and from its PCB, respectively.

Self Review

1. From where does an operating system load the execution context for the process to be dis- patched 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.

3.44 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 ing a protected memory location.

dividing by zero or referenc-

Interrupts provide a low-overhead means of gaining the attention of a proces- sor. 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 appropri- ate 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).

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

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 per- form 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.

3.41 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

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 signalled 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 P1 (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).

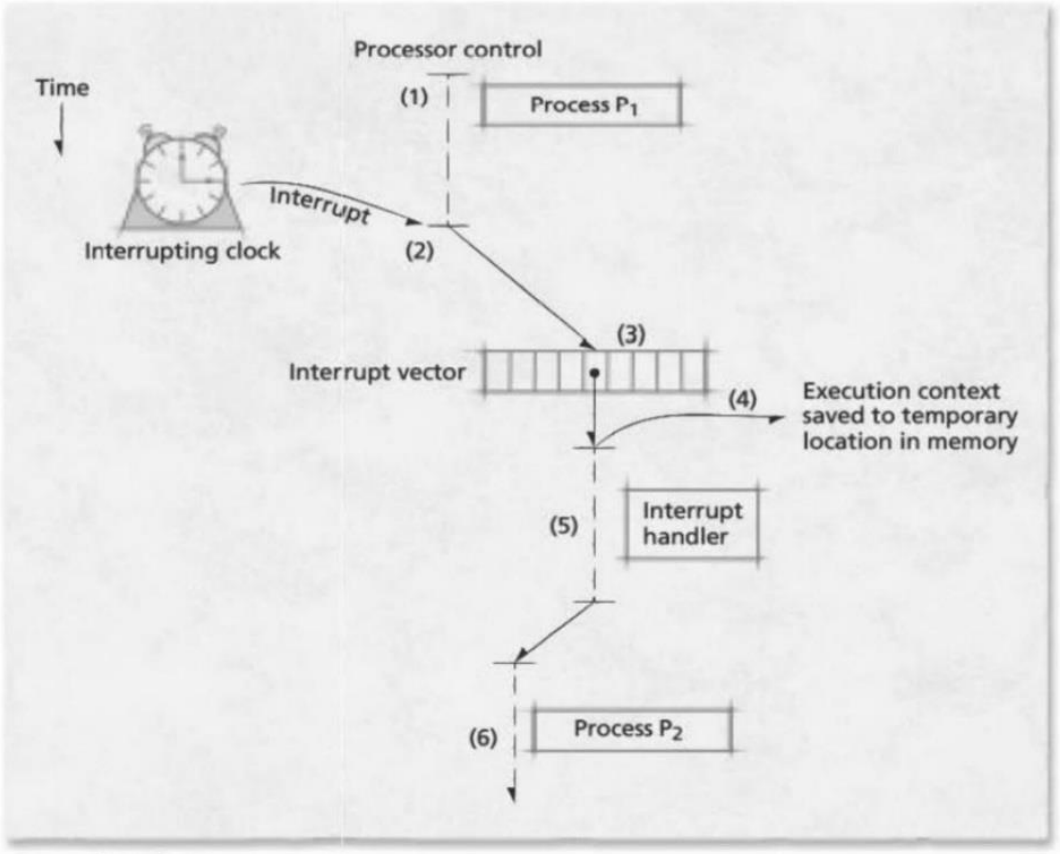

Figure 3.7 | Handling interrupts.

The processor then saves the process's execution context to memory (4) so that the P'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 interupt (5). The interrupt handler may then restore the state of the previously executing process (P1) 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 P2, the highest-priority waiting process, should obtain the processor (6). The context for process P2 is then loaded from its PCB in main memory, and process Pi's execution context is saved to its PCB in main memory.

3.4.2 Interrupt Classes

The set of interrupts a computer supports is dependent on the system's 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.
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.

3,5 Interposes 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.

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.

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

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);

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 pro- cesses 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 treat-

Key Tarms

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.

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.

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

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.

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 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 pro- cesses in the system.

ready state-Process state from which a process may be dis- patched to the processor.

ready list-Kernel data structure that organizes all ready pro- cesses in the system. The ready list is typically order by ed process scheduling priority.

resume-Remove a process from a suspended state.

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

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.

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 sys- tem operator to manually adjust the system load and/or respond to threats of system failure.

suspendedblocked state- Process state resulting from the pro- cess being suspended while in the blocked state. Resum- ing such a process places it into the blocked state.

suspendedready state- Process state resulting from the pro- cess being suspended while in the ready state. Resuming such a process places it into the ready state.

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.