

PROCESS CONCEPTS

The background features several light gray, wavy, horizontal lines that sweep across the lower right portion of the slide, creating a sense of motion and flow.

3.1 Introduction

- Computers perform operations concurrently
 - For example, compiling a program, sending a file to a printer, rendering a Web page, playing music and receiving e-mail
 - Processes enable systems to perform and track simultaneous activities
 - Processes transition between process states
 - Operating systems perform operations on processes such as creating, destroying, suspending, resuming and waking

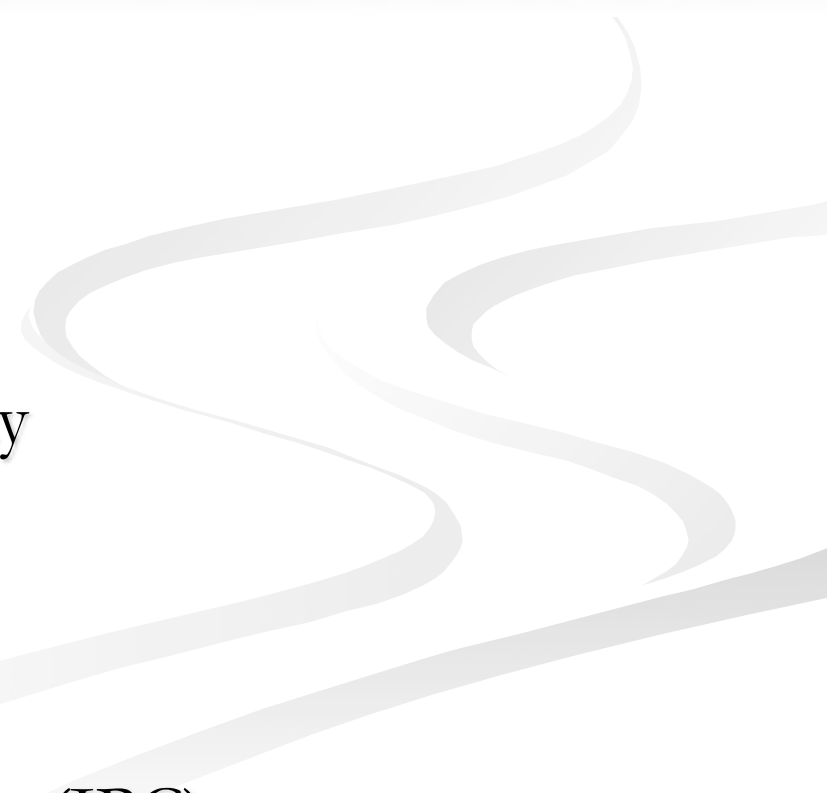
3.1.1 Definition of Process

- A program in execution
 - A process has its own address space consisting of:
 - Text region
 - Stores the code that the processor executes
 - Data region
 - Stores variables and dynamically allocated memory
 - Stack region
 - Stores instructions and local variables for active procedure calls

3.2 Process States: Life Cycle of a Process

- A process moves through a series of discrete process states:
 - *Running* state
 - The process is executing on a processor
 - *Ready* state
 - The process could execute on a processor if one were available
 - *Blocked* state
 - The process is waiting for some event to happen before it can proceed
- The OS maintains a *ready* list and a *blocked* list

3.3 Process Management

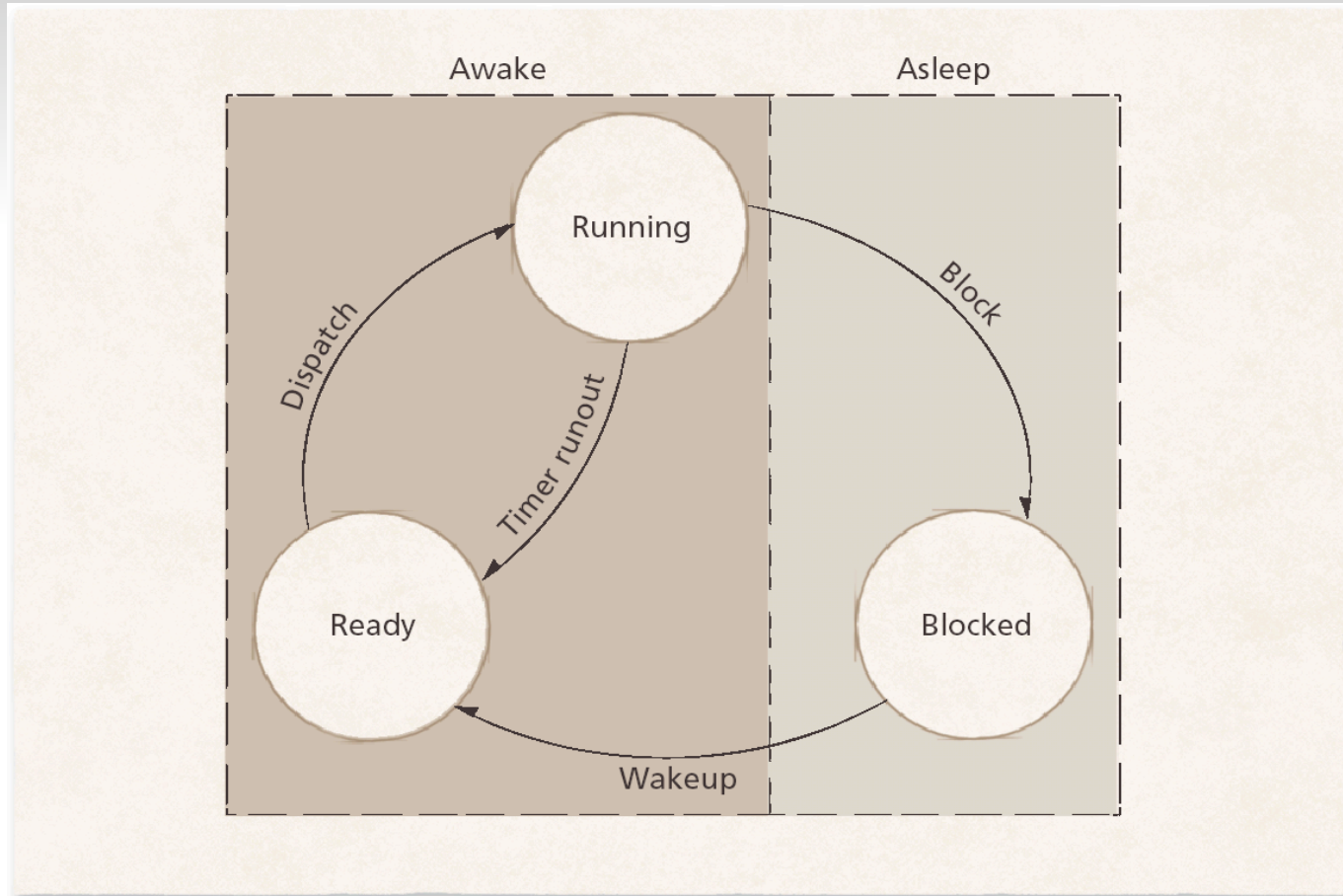
- Operating systems provide fundamental services to processes including:
 - Creating processes
 - Destroying processes
 - Suspending processes
 - Resuming processes
 - Changing a process's priority
 - Blocking processes
 - Waking up processes
 - Dispatching processes
 - Interprocess communication (IPC)
- 
- A decorative graphic consisting of several overlapping, wavy, light gray lines that flow from the right side of the slide towards the left, creating a sense of movement and depth.

3.3.1 Process States and State Transitions

- Process states
 - The act of assigning a processor to the first process on the ready list is called dispatching
 - The OS may use an interval timer to allow a process to run for a specific time interval or quantum
 - Cooperative multitasking lets each process run to completion
- State Transitions
 - At this point, there are four possible state transitions
 - When a process is dispatched, it transitions from *ready* to *running*

3.3.1 Process States and State Transitions

Figure 3.1 Process state transitions.



3.3.2 Process Control Blocks (PCBs)/Process Descriptors

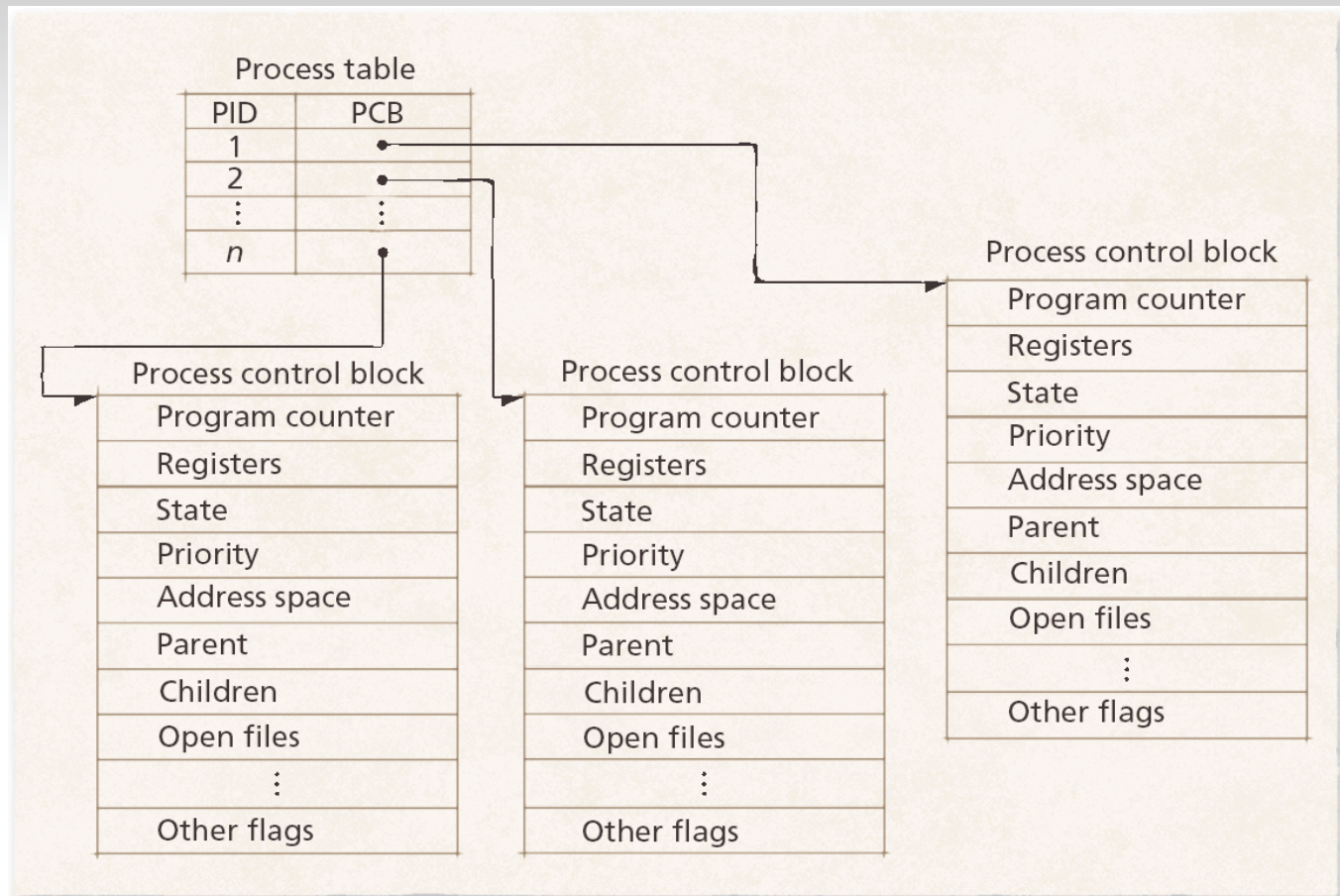
- PCBs maintain information that the OS needs to manage the process
 - Typically include information such as
 - Process identification number (PID)
 - Process state
 - Program counter
 - Scheduling priority
 - Credentials
 - A pointer to the process's parent process
 - Pointers to the process's child processes
 - Pointers to locate the process's data and instructions in memory

3.3.2 Process Control Blocks (PCBs)/Process Descriptors

- Process table
 - The OS maintains pointers to each process's PCB in a system-wide or per-user process table
 - Allows for quick access to PCBs
 - When a process is terminated, the OS removes the process from the process table and frees all of the process's resources

3.3.2 Process Control Blocks (PCBs)/Process Descriptors

Figure 3.2 Process table and process control blocks.

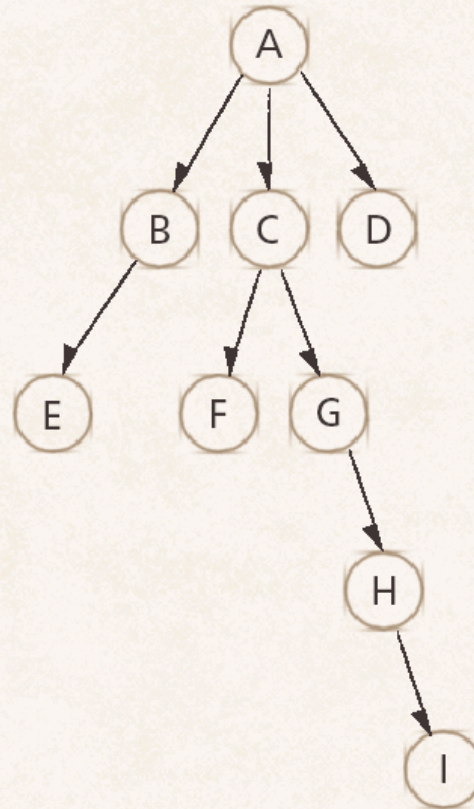


3.3.3 Process Operations

- A process may spawn a new process
 - The creating process is called the parent process
 - The created process is called the child process
 - Exactly one parent process creates a child
 - When a parent process is destroyed, operating systems typically respond in one of two ways:
 - Destroy all child processes of that parent
 - Allow child processes to proceed independently of their parents

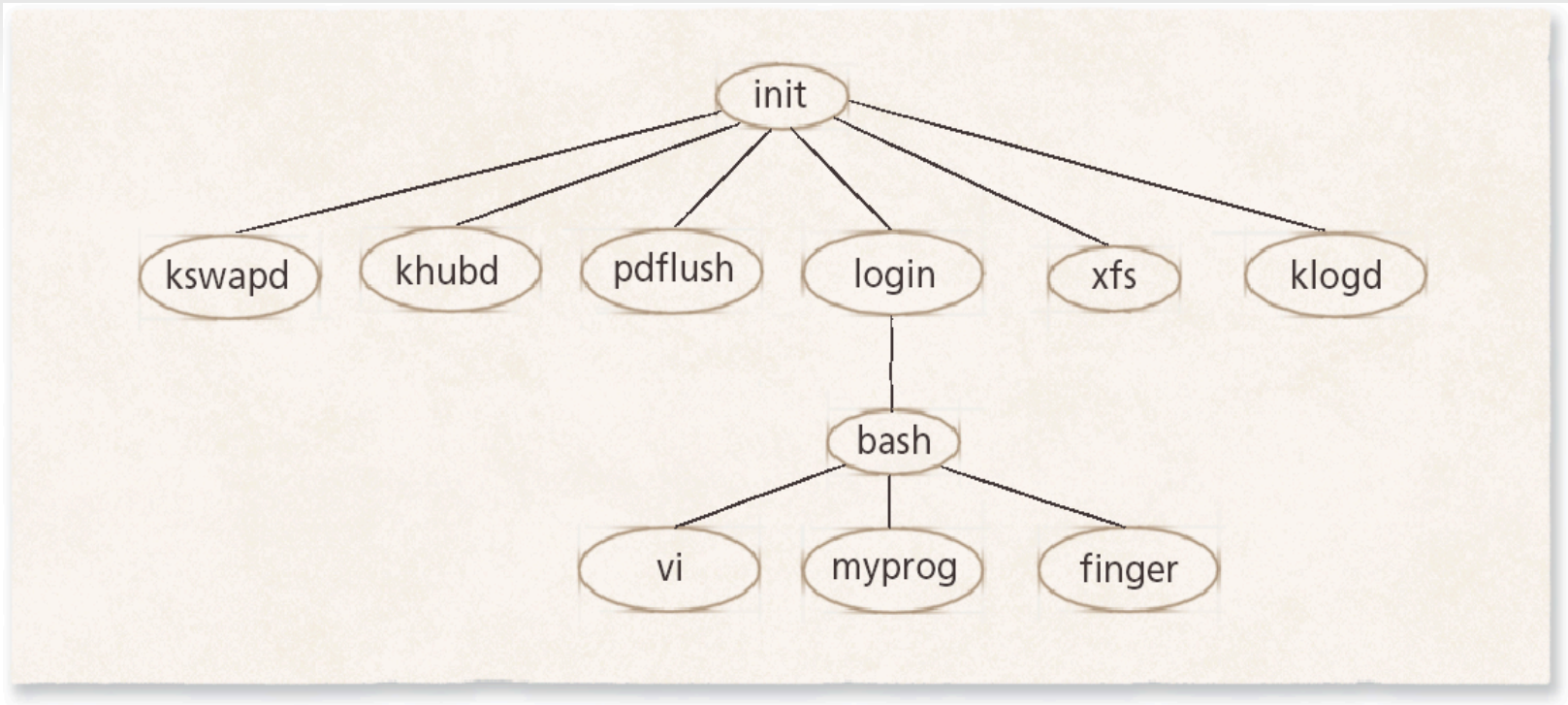
3.3.3 Process Operations

Figure 3.3 Process creation hierarchy.



3.3.3 Process Operations

Figure 3.4 Process hierarchy in Linux.

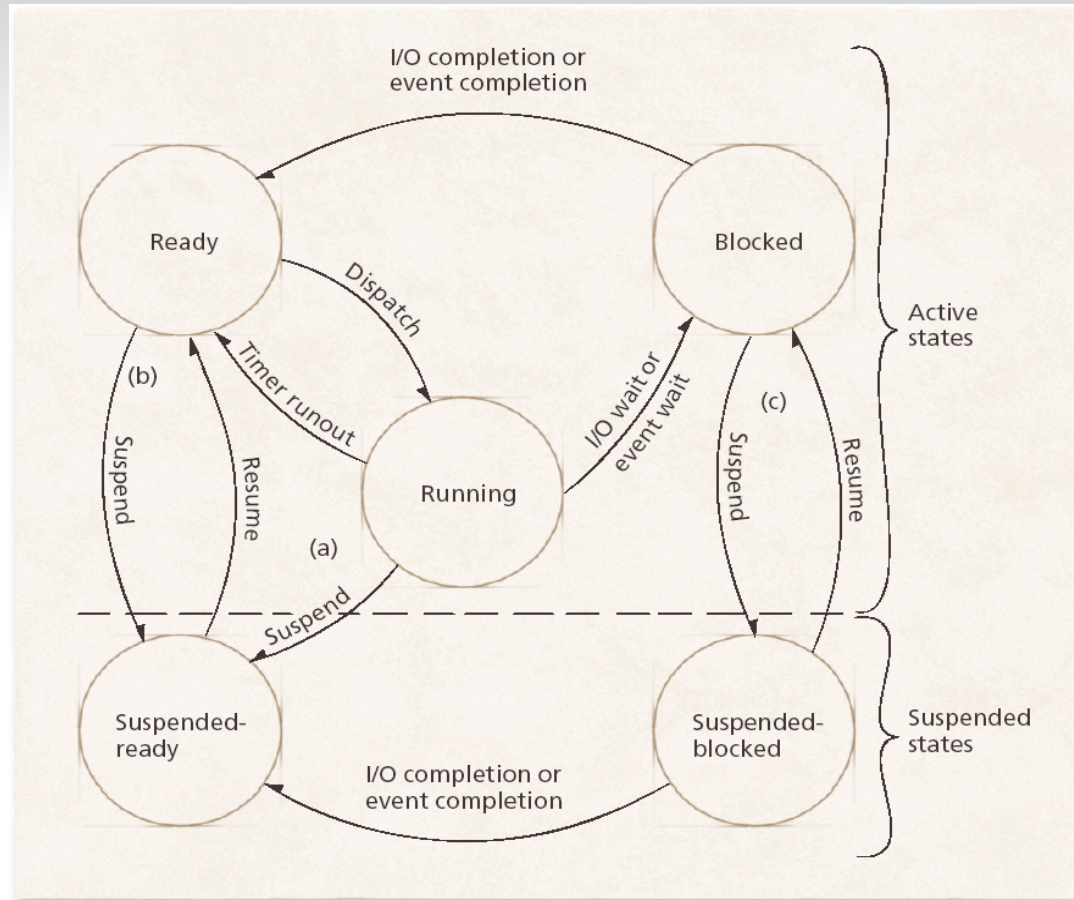


3.3.4 Suspend and Resume

- Suspending a process
 - Indefinitely removes it from contention for time on a processor without being destroyed
 - Useful for detecting security threats and for software debugging purposes
 - A suspension may be initiated by the process being suspended or by another process
 - A suspended process must be resumed by another process
 - Two suspended states:
 - *suspendedready*
 - *suspendedblocked*

3.3.4 Suspend and Resume

Figure 3.5 Process state transitions with suspend and resume.

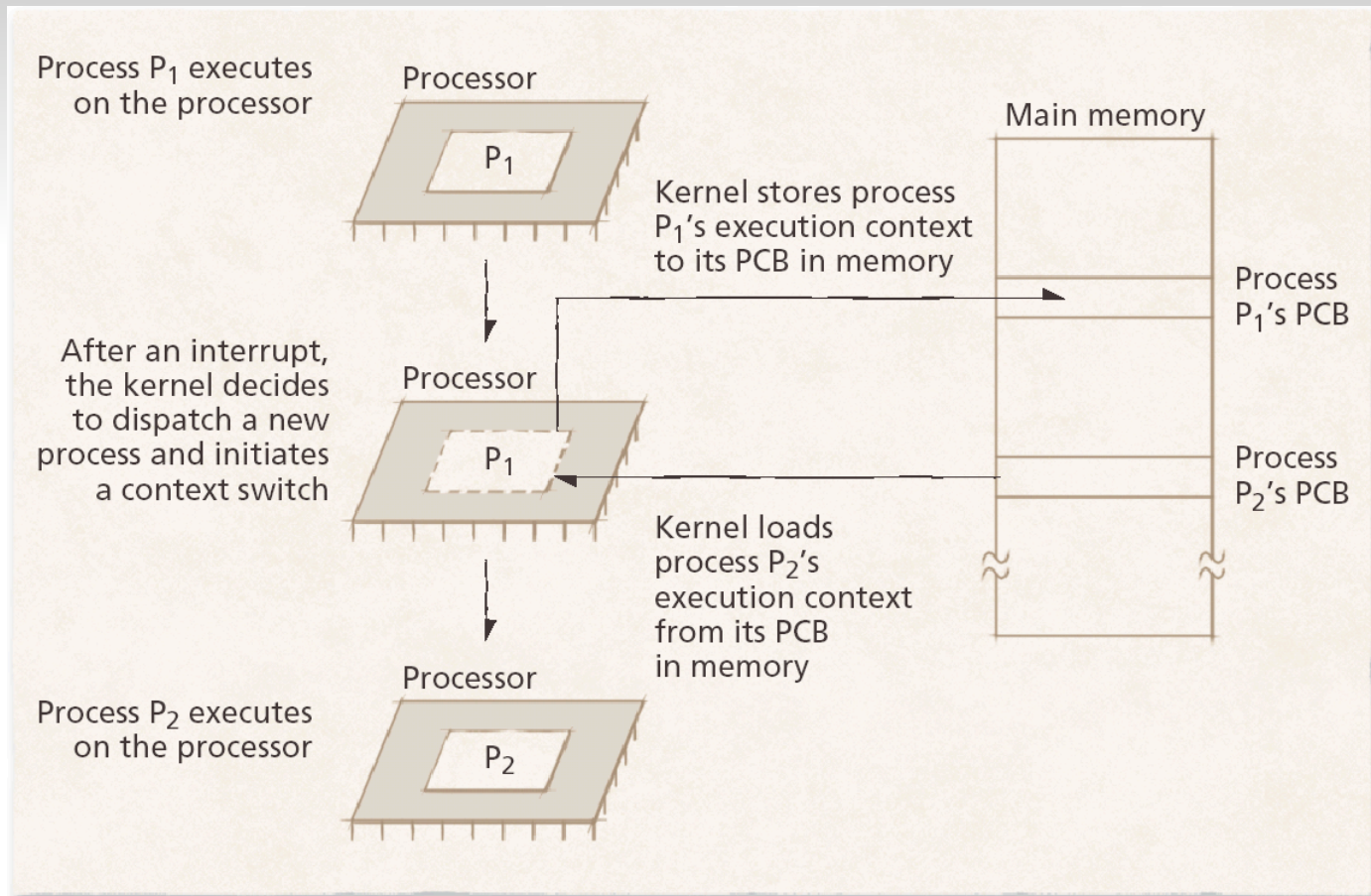


3.3.5 Context Switching

- Context switches
 - Performed by the OS to stop executing a *running* process and begin executing a previously *ready* process
 - Save the execution context of the *running* process to its PCB
 - Load the *ready* process's execution context from its PCB
 - Must be transparent to processes
 - Require the processor to not perform any “useful” computation

3.3.5 Context Switching

Figure 3.6 Context switch.



3.4 Interrupts

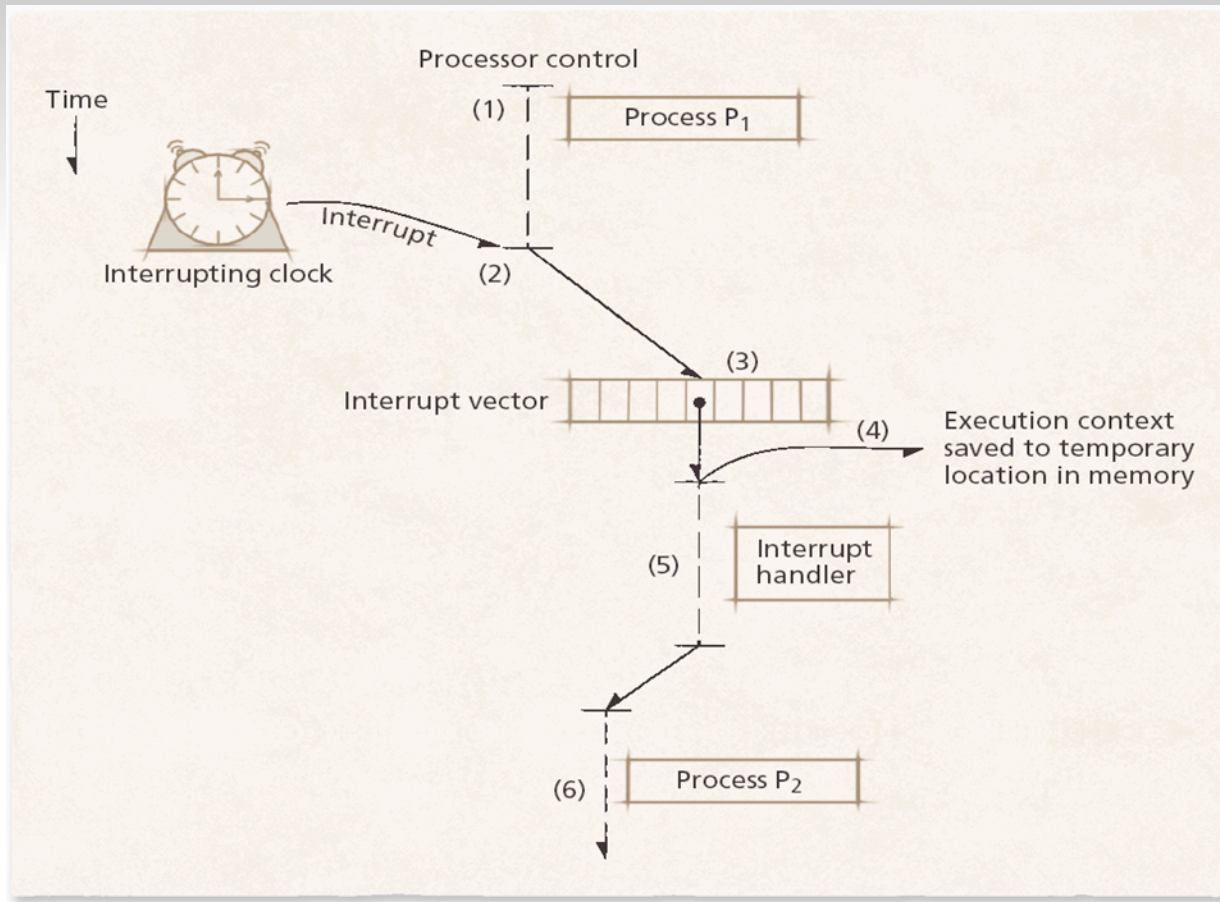
- Interrupts enable software to respond to signals from hardware
 - May be initiated by a running process
 - Interrupt is called a trap
 - Synchronous with the operation of the process
 - For example, dividing by zero or referencing protected memory
 - May be initiated by some event that may or may not be related to the running process
 - Asynchronous with the operation of the process
 - For example, a key is pressed on a keyboard or a mouse is moved
 - Low overhead
- Polling is an alternative approach
 - Processor repeatedly requests the status of each device
 - Increases in overhead as the complexity of the system increases

3.4.1 Interrupt Processing

- Handling interrupts
 - After receiving an interrupt, the processor completes execution of the current instruction, then pauses the current process
 - The processor will then execute one of the kernel's interrupt-handling functions
 - The interrupt handler determines how the system should respond
 - Interrupt handlers are stored in an array of pointers called the interrupt vector
 - After the interrupt handler completes, the interrupted process is restored and executed or the

3.4.1 Interrupt Processing

Figure 3.7 Handling interrupts.



3.4.2 Interrupt Classes

- Supported interrupts depend on a system's architecture
 - The IA-32 specification distinguishes between two types of signals a processor may receive:
 - Interrupts
 - Notify the processor that an event has occurred or that an external device's status has changed
 - Generated by devices external to a processor
 - Exceptions
 - Indicate that an error has occurred, either in hardware or as a result of a software instruction
 - Classified as faults, traps or aborts

3.4.2 Interrupt Classes

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

<i>Interrupt Type</i>	<i>Description of Interrupts in Each Type</i>
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.

3.4.2 Interrupt Classes

Figure 3.9 Intel IA-32 exception classes.

<i>Exception Class</i>	<i>Description of Exceptions in Each Class</i>
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 Interprocess Communication

- Many operating systems provide mechanisms for interprocess communication (IPC)
 - Processes must communicate with one another in multiprogrammed and networked environments
 - For example, a Web browser retrieving data from a distant server
 - Essential for processes that must coordinate activities to achieve a common goal

3.5.1 Signals

- Software interrupts that notify a process that an event has occurred
 - Do not allow processes to specify data to exchange with other processes
 - Processes may catch, ignore or mask a signal
 - Catching a signal involves specifying a routine that the OS calls when it delivers the signal
 - Ignoring a signal relies on the operating system's default action to handle the signal
 - Masking a signal instructs the OS to not deliver signals of that type until the process clears the signal mask

3.5.2 Message Passing

- Message-based interprocess communication
 - Messages can be passed in one direction at a time
 - One process is the sender and the other is the receiver
 - Message passing can be bidirectional
 - Each process can act as either a sender or a receiver
 - Messages can be blocking or nonblocking
 - Blocking requires the receiver to notify the sender when the message is received
 - Nonblocking enables the sender to continue with other processing
 - Popular implementation is a pipe
 - A region of memory protected by the OS that serves as a buffer, allowing two or more processes to exchange data

3.5.2 Message Passing

- IPC in distributed systems
 - Transmitted messages can be flawed or lost
 - Acknowledgement protocols confirm that transmissions have been properly received
 - Timeout mechanisms retransmit messages if acknowledgements are not received
 - Ambiguously named processes lead to incorrect message referencing
 - Messages are passed between computers using numbered ports on which processes listen, avoiding this problem
 - Security is a significant problem
 - Ensuring authentication

3.6 Case Study: UNIX Processes

- UNIX processes
 - All processes are provided with a set of memory addresses, called a virtual address space
 - A process's PCB is maintained by the kernel in a protected region of memory that user processes cannot access
 - A UNIX PCB stores:
 - The contents of the processor registers
 - PID
 - The program counter
 - The system stack
 - All processes are listed in the process table

3.6 Case Study: UNIX Processes

- UNIX processes continued
 - All processes interact with the OS via system calls
 - A process can spawn a child process by using the fork system call, which creates a copy of the parent process
 - Child receives a copy of the parent's resources as well
 - Process priorities are integers between -20 and 19 (inclusive)
 - A lower numerical priority value indicates a higher scheduling priority
 - UNIX provides IPC mechanisms, such as pipes, to allow unrelated processes to transfer data

3.6 Case Study: UNIX Processes

Figure 3.10 UNIX system calls.

<i>System Call</i>	<i>Description</i>
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.