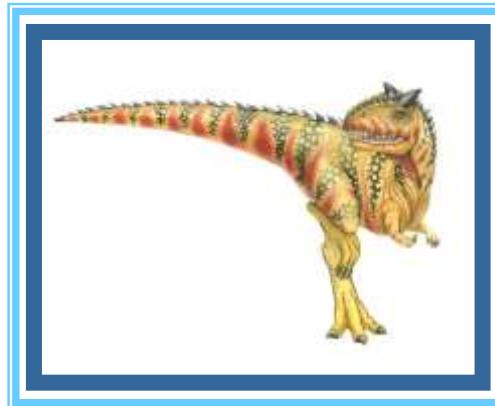


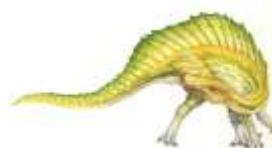
Chapter 1: Introduction





Chapter 1: Introduction

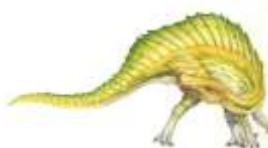
- What is an Operating System?
- Computer-System Organization
- Operating-System Structure
- Operating-System Operations
- Process Management
- Memory Management
- Storage Management
- Protection and Security
- Kernel Data Structures
- Computing Environments
- Open-Source Operating Systems
- Computer-System Architecture





Objectives

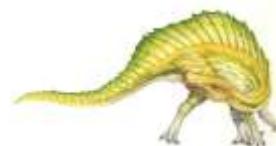
- To describe the basic organization of computer systems
- To provide a grand tour of the major components of operating systems
- To give an overview of the many types of computing environments
- To explore several open-source operating systems





What is an Operating System?

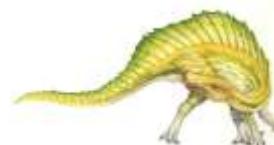
- A program that acts as an intermediary between a user of a computer and the computer hardware
- Operating system goals:
 - Execute user programs and make solving user problems easier
 - Make the computer system convenient to use
 - Use the computer hardware in an efficient manner





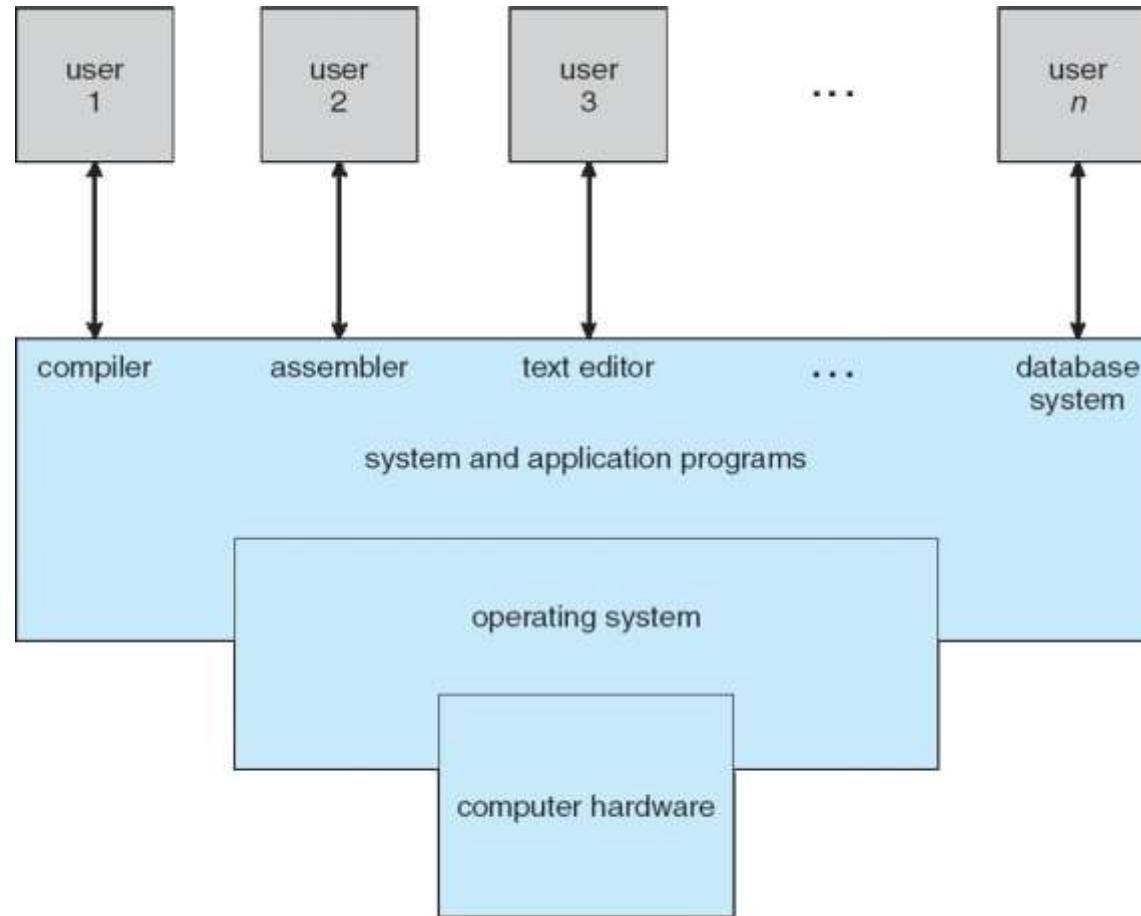
Computer System Structure

- Computer system can be divided into four components:
 - Hardware – provides basic computing resources
 - ▶ CPU, memory, I/O devices
 - Operating system
 - ▶ Controls and coordinates use of hardware among various applications and users
 - Application programs – define the ways in which the system resources are used to solve the computing problems of the users
 - ▶ Word processors, compilers, web browsers, database systems, video games
 - Users
 - ▶ People, machines, other computers





Four Components of a Computer System





What Operating Systems Do

- The operating system controls the hardware and coordinates its use among the various application programs for the various users.
- We can also view a computer system as consisting of hardware, software, and data.
- The operating system provides the means for proper use of these resources in the operation of the computer system.
- An operating system is similar to a government. Like a government, it performs no useful function by itself. It simply provides an **environment** within which other programs can do useful work.
- To understand more fully the operating system's role, we explore operating systems from two viewpoints:
 - The user
 - The system.





User View

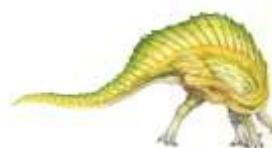
The user's view of the computer varies according to the interface being used

- **Single user computers** (e.g., PC, workstations). Such systems are designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. the operating system is designed mostly for **ease of use** and **good performance**.
- **Multi user computers** (e.g., mainframes, computing servers). These users share resources and may exchange information. The operating system in such cases is designed to maximize resource utilization -- to assure that all available CPU time, memory, and I/O are used efficiently and that no individual users takes more than their air share.



User View (Cont.)

- **Handheld computers** (e.g., smartphones and tablets). The user interface for mobile computers generally features a **touch screen**. The systems are resource poor, optimized for usability and battery life.
- **Embedded computers** (e.g., computers in home devices and automobiles) The user interface may have numeric keypads and may turn indicator lights on or off to show status. The operating systems are designed primarily to run without user intervention.





System View

From the computer's point of view, the operating system is the program most intimately involved with the hardware. There are two different views:

- The operating system is a **resource allocator**
 - Manages all resources
 - Decides between conflicting requests for efficient and fair resource use
- The operating systems is a **control program**
 - Controls execution of programs to prevent errors and improper use of the computer

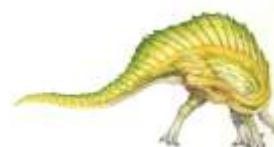




Defining Operating System

No universally accepted definition of **what** an OS:

- Operating systems exist to offer a reasonable way to solve the problem of creating a usable computing system.
- The fundamental goal of computer systems is to execute user programs and to make solving user problems easier.
- Since bare hardware alone is not particularly easy to use, application programs are developed.
 - These programs require certain common operations, such as those controlling the I/O devices.
 - The common functions of controlling and allocating resources are brought together into one piece of software: the **operating system**.

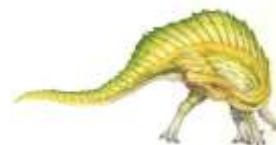




Defining Operating System (Cont.)

No universally accepted definition of what is **part** of the OS:

- A simple viewpoint is that it includes everything a vendor ships when you order the operating system. The features that are included vary greatly across systems:
 - Some systems take up less than a megabyte of space and lack even a full-screen editor,
 - Some systems require gigabytes of space and are based entirely on graphical windowing systems.

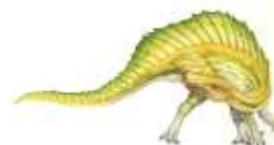




Defining Operating System (Cont.)

No universally accepted definition of what is **part** of the OS:

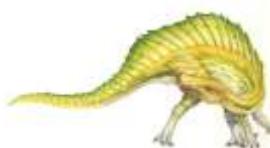
- A more common definition, and the one that we usually follow, is that the operating system is the one program running at all times on the computer -- usually called the **kernel**.
- Along with the kernel, there are two other types of programs:
 - System programs, which are associated with the operating system but are not necessarily part of the kernel.
 - Application programs, which include all programs not associated with the operation of the system.





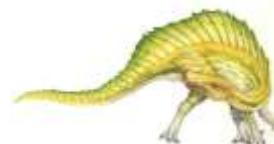
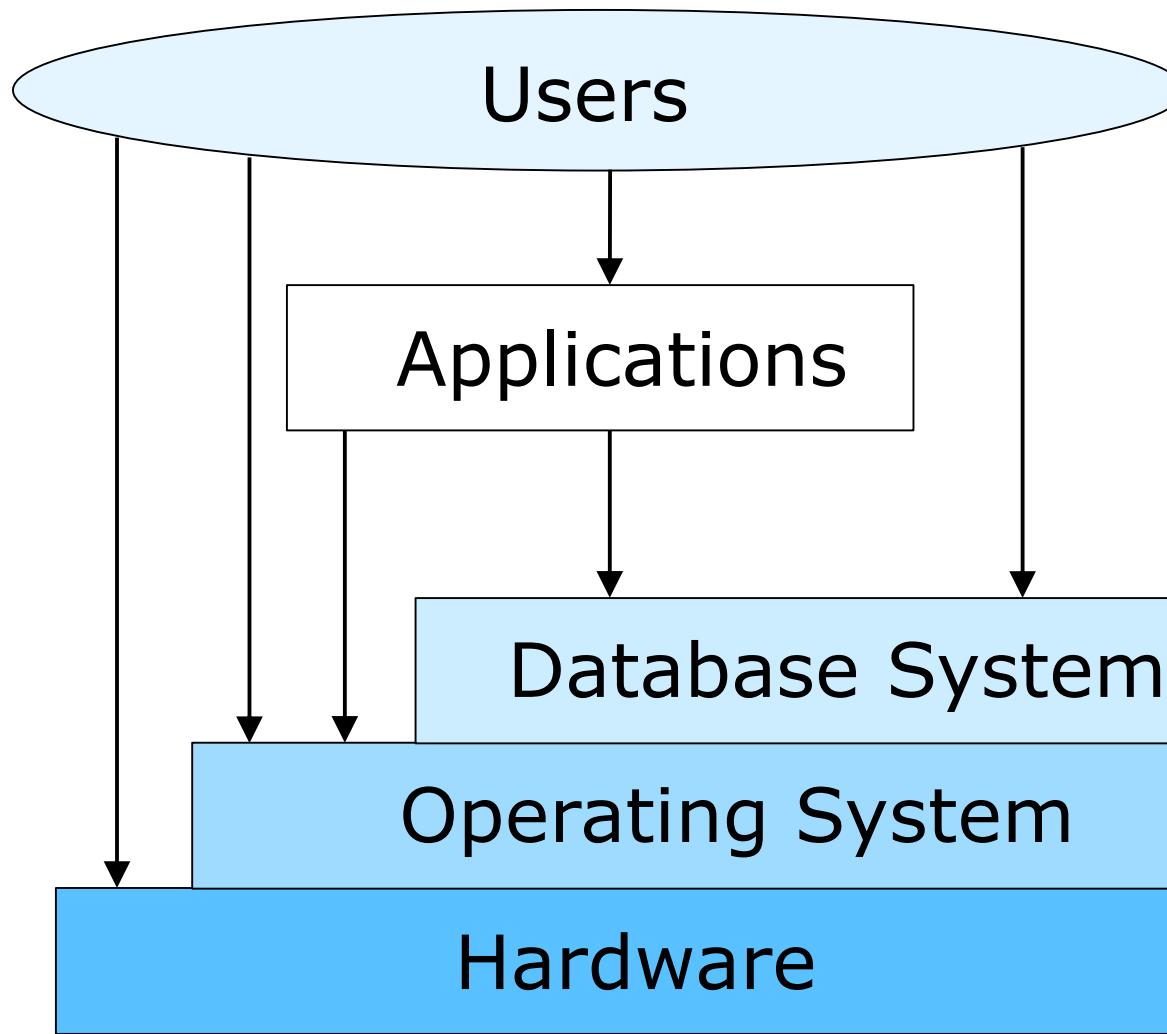
Defining Operating System (Cont.)

- The emergence of mobile devices, have resulted in an increase in the number of features that constituting the operating system.
- Mobile operating systems often include not only a core kernel but also **middleware** -- a set of software frameworks that provide additional services to application developers.
- For example, each of the two most prominent mobile operating systems -- Apple's iOS and Google's Android -- feature a core kernel along with middleware that supports databases, multimedia, and graphics (to name only a few).





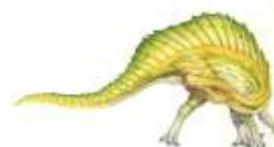
Evolution of Computer Systems





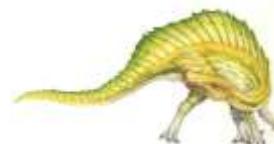
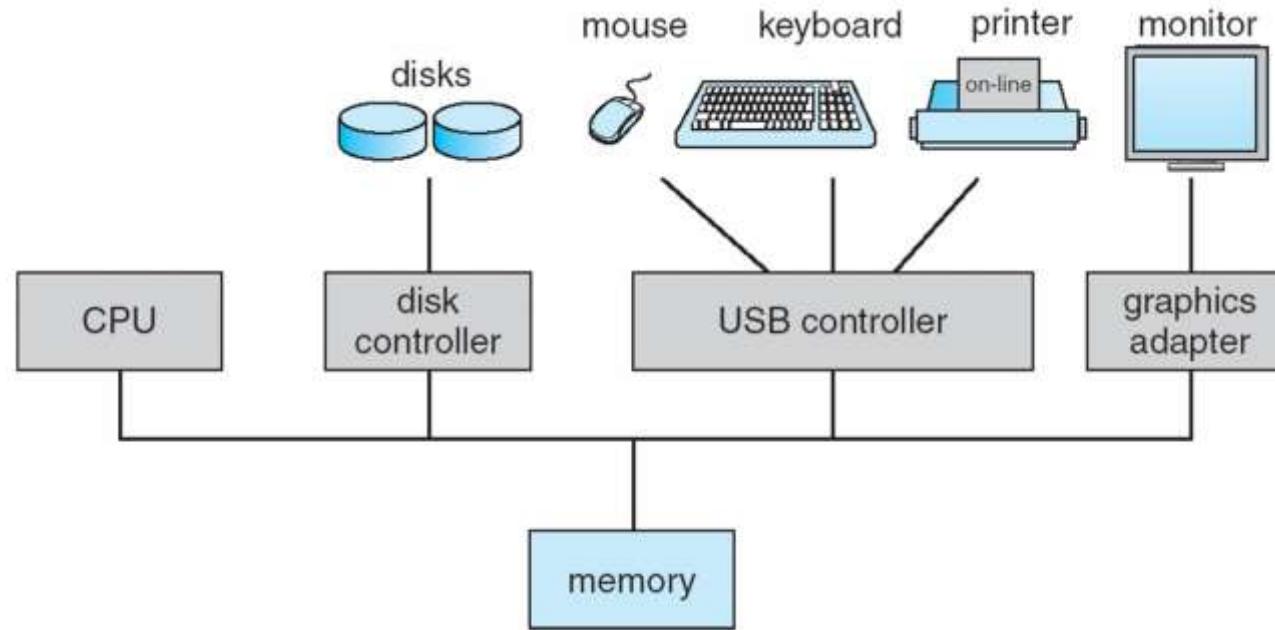
Computer-System Organization

- A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access to shared memory.
- Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, or video displays). Each device controller has a local buffer.
- CPU moves data from/to main memory to/from local buffers.
- The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.





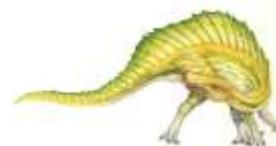
Modern Computer System





Computer Startup

- **Bootstrap program** is loaded at power-up or reboot
 - Typically stored in ROM or EPROM, generally known as **firmware**
 - Initializes all aspects of system
 - Loads operating system kernel and starts execution





Computer-System Operation

- Once the kernel is loaded and executing, it can start providing services to the system and its users.
- Some services are provided outside of the kernel, by system programs that are loaded into memory at boot time to become **system processes**, or **system daemons** that run the entire time the kernel is running.
- On UNIX, the first system process is **init** and it starts many other daemons. Once this phase is complete, the system is fully booted, and the system waits for some event to occur.
- The occurrence of an event is usually signaled by an **interrupt**.





Interrupts

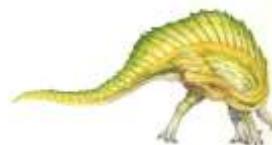
- There are two types of interrupts:
 - **Hardware** -- a device may trigger an interrupt by sending a signal to the CPU, usually by way of the system bus.
 - **Software** -- a program may trigger an interrupt by executing a special operation called a **system call**.
- A software-generated interrupt (sometimes called **trap** or **exception**) is caused either by an error (e.g., divide by zero) or a user request (e.g., an I/O request).
- An operating system is **interrupt driven**.





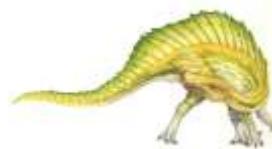
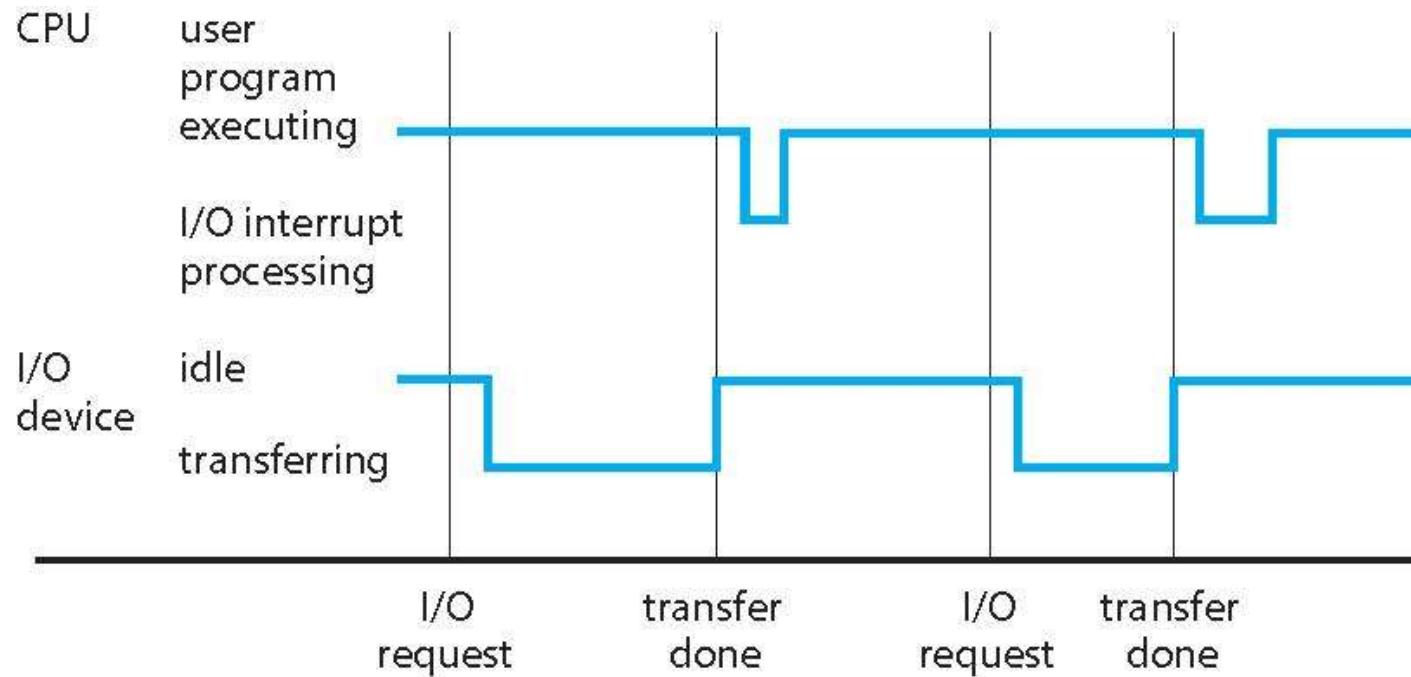
Common Functions of Interrupts

- When an interrupt occurs, the operating system preserves the state of the CPU by storing the registers and the program counter
- Determines which type of interrupt has occurred and transfers control to the interrupt-service routine.
- An interrupt-service routine is a collection of routines (modules), each of which is responsible for handling one particular interrupt (e.g., from a printer, from a disk)
- The transfer is generally through the **interrupt vector**, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction.



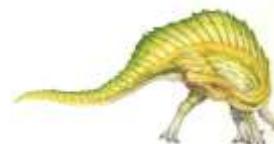
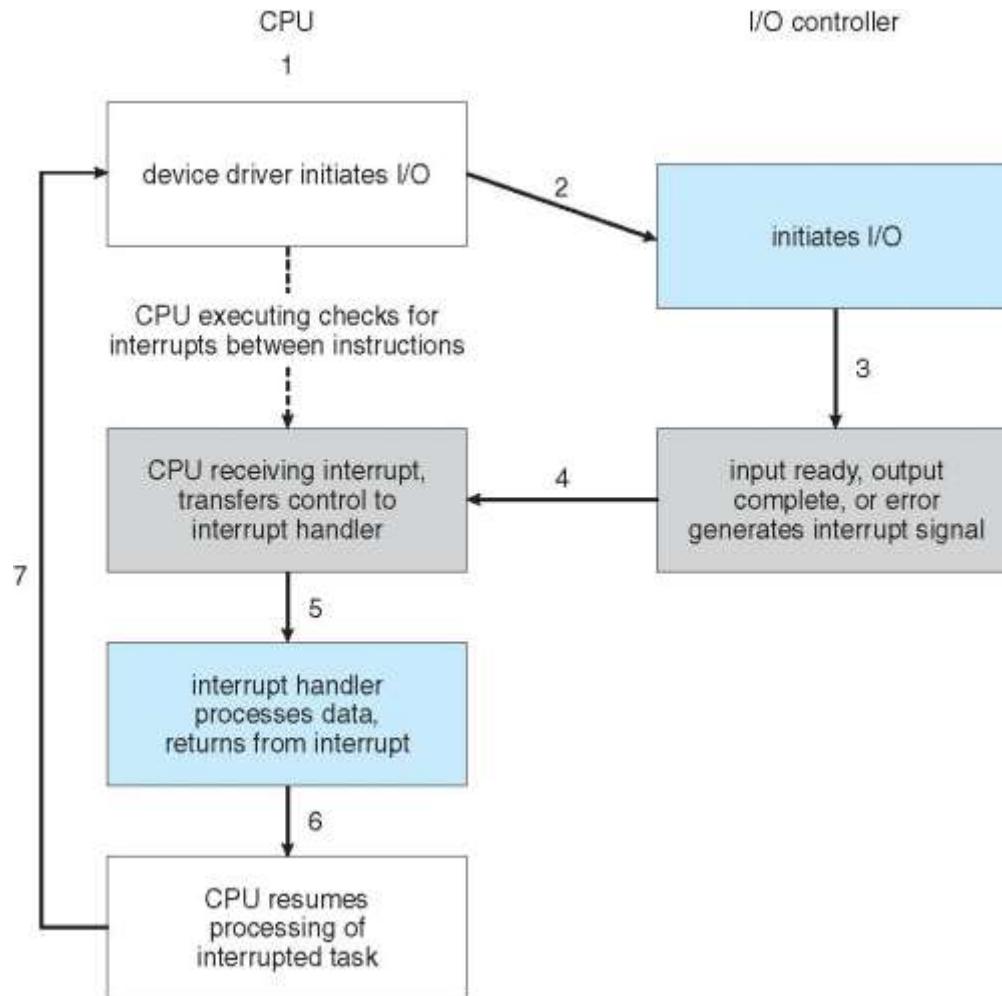


Interrupt Timeline





Interrupt-driven I/O cycle.





Intel Pentium processor event-vector table

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts





Storage Structure

- Main memory – the only large storage media that the CPU can access directly
 - Random access
 - Typically volatile
- Secondary storage – extension of main memory that provides large **nonvolatile** storage capacity
 - Hard disks – rigid metal or glass platters covered with magnetic recording material
 - ▶ Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
 - ▶ The **disk controller** determines the logical interaction between the device and the computer
 - **Solid-state disks** – faster than hard disks, nonvolatile
 - ▶ Various technologies
 - ▶ Becoming more popular
- Tertiary storage





Storage Definition

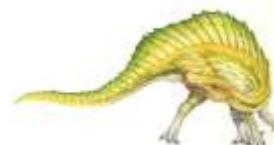
- The basic unit of computer storage is the **bit**. A bit can contain one of two values, 0 and 1. All other storage in a computer is based on collections of bits.
- A **byte** is 8 bits, and on most computers it is the smallest convenient chunk of storage.
- A less common term is **word**, which is a given computer architecture's native unit of data. A word is made up of one or more bytes.





Storage Definition (Cont.)

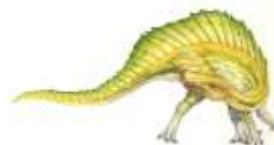
- Computer storage, along with most computer throughput, is generally measured and manipulated in bytes and collections of bytes.
 - A **kilobyte**, or **KB**, is 1,024 bytes
 - a **megabyte**, or **MB**, is $1,024^2$ bytes
 - a **gigabyte**, or **GB**, is $1,024^3$ bytes
 - a **terabyte**, or **TB**, is $1,024^4$ bytes
 - a **petabyte**, or **PB**, is $1,024^5$ bytes
 - exabyte, zettabyte, yottabyte
- Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes. Networking measurements are an exception to this general rule; they are given in bits (because networks move data a bit at a time).





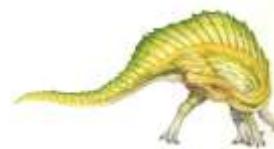
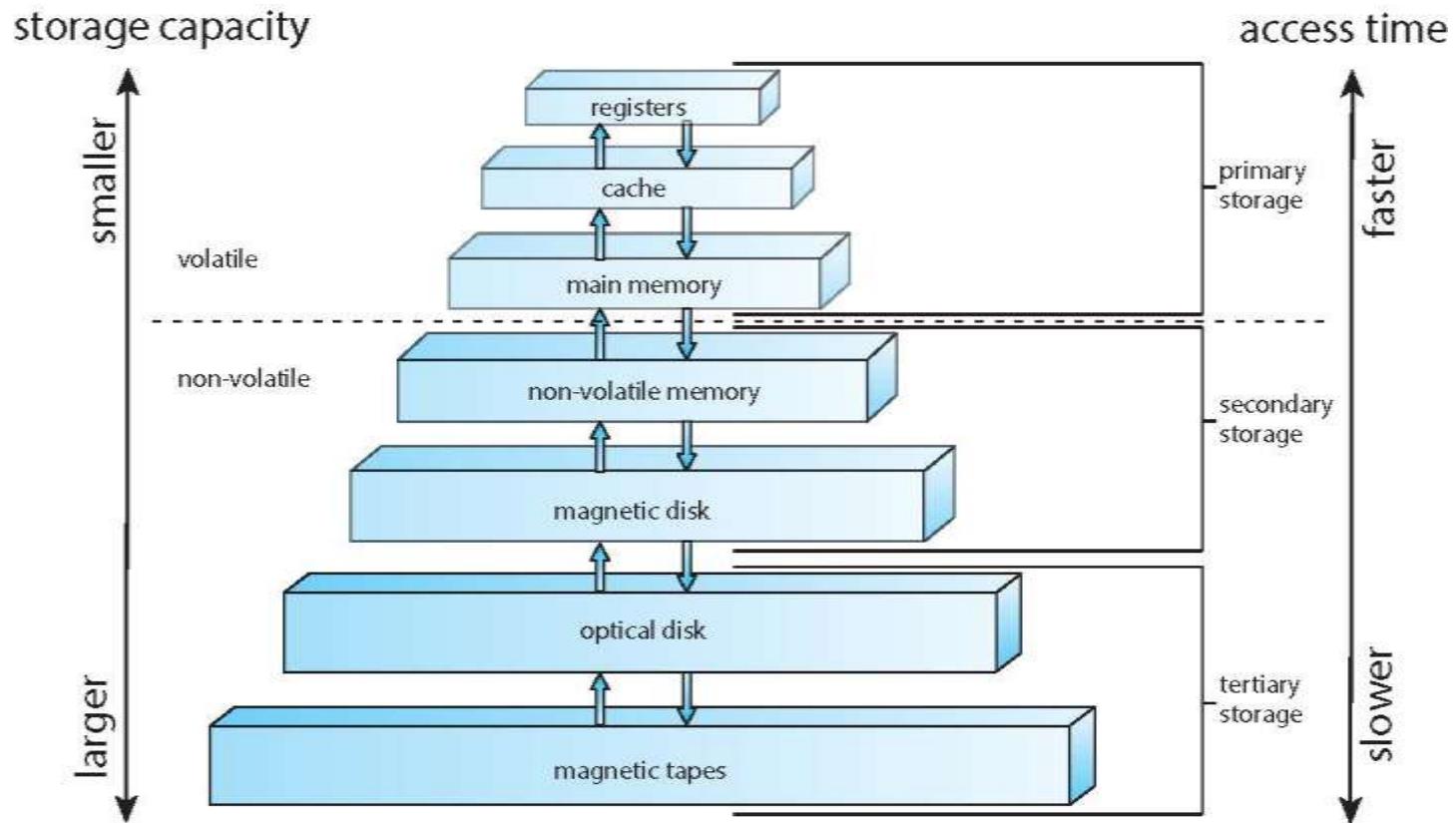
Storage Hierarchy

- Storage systems organized in hierarchy
 - Speed
 - Cost
 - Volatility
- **Caching** – copying information from “slow” storage into faster storage system;
 - Main memory can be viewed as a cache for secondary storage
- **Device Driver** for each device controller to manage I/O
 - Provides uniform interface between controller and kernel





Storage-device hierarchy





I/O Structure

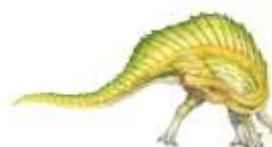
- A general-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus.
- Each device controller is in charge of a specific type of device. More than one device may be attached. For instance, seven or more devices can be attached to the **small computer-systems interface (SCSI)** controller.
- A device controller maintains some local buffer storage and a set of special-purpose registers.
- The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage.
- Typically, operating systems have a **device driver** for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device.





I/O Structure (Cont.)

- To start an I/O operation, the device driver loads the appropriate registers within the device controller.
- The device controller, in turn, examines the contents of these registers to determine what action to take (such as “read” a character from the keyboard).
- The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver via an interrupt that it has finished its operation.
- The device driver then returns control to the operating system, possibly returning the data or a pointer to the data if the operation was a read.
- For other operations, the device driver returns status information.





Direct Memory Access Structure

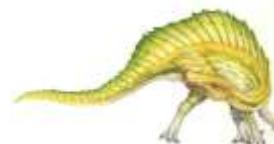
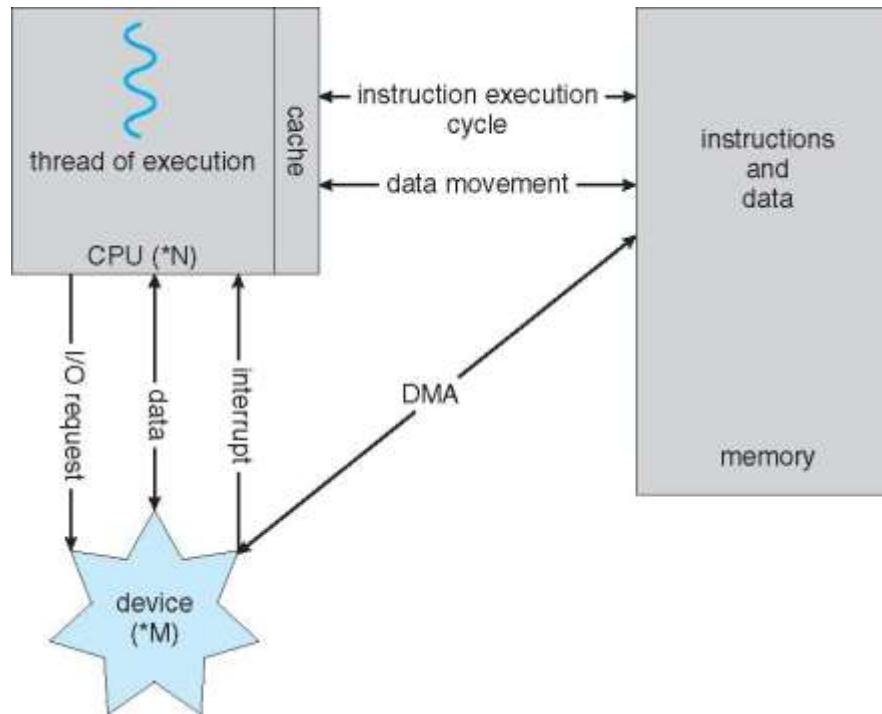
- Interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as disk I/O.
- To solve this problem, **direct memory access** (DMA) is used.
 - After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU.
 - Only one interrupt is generated per block, to tell the device driver that the operation has completed. While the device controller is performing these operations, the CPU is available to accomplish other work.
- Some high-end systems use switch rather than bus architecture. On these systems, multiple components can talk to other components concurrently, rather than competing for cycles on a shared bus. In this case, DMA is even more effective. The figure in next slide shows the interplay of all components of a computer system.





How a Modern Computer Works

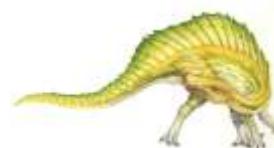
A von Neumann architecture and a depiction of the interplay of all components of a computer system.





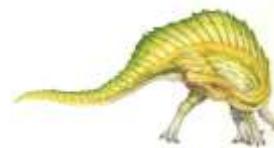
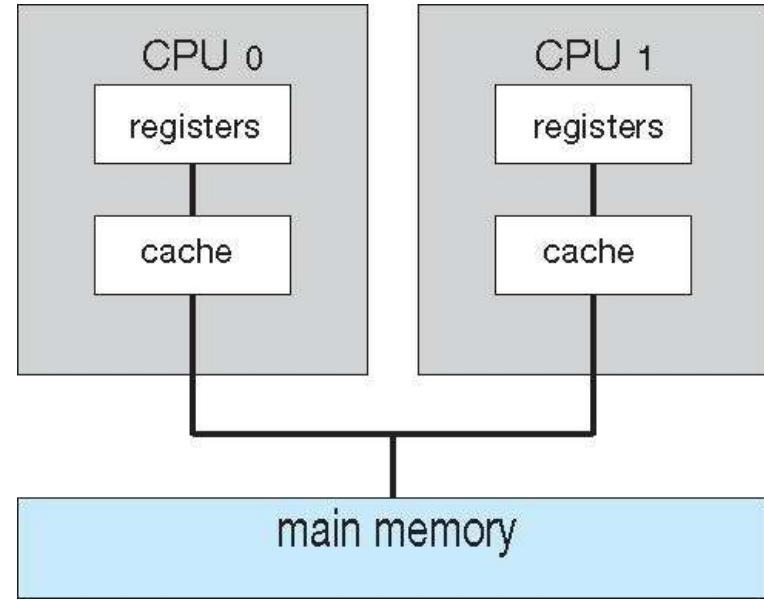
Computer-System Architecture

- **Single general-purpose processor**
 - Most systems have special-purpose processors as well
- **Multiprocessors** systems
 - Also known as **parallel systems, tightly-coupled systems**
 - Advantages include:
 - ▶ **Increased throughput**
 - ▶ **Economy of scale**
 - ▶ **Increased reliability** – graceful-degradation/fault-tolerance
 - Two types:
 - ▶ **Symmetric Multiprocessing** – each processor performs all tasks
 - ▶ **Asymmetric Multiprocessing** – each processor is assigned a specific task.





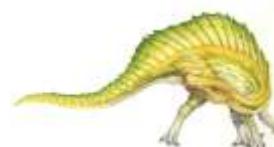
Symmetric Multiprocessing Architecture





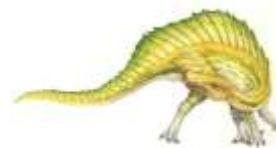
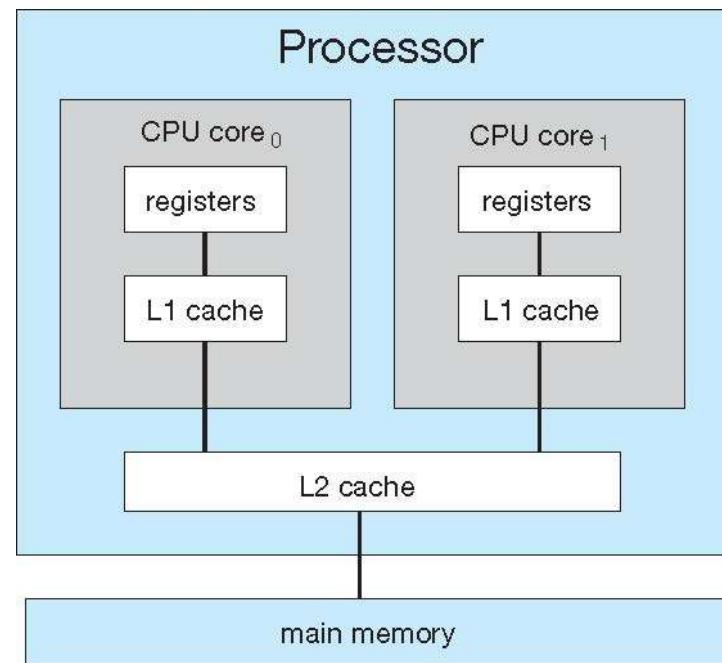
Multicore Systems

- Most CPU design now includes multiple computing cores on a single chip. Such multiprocessor systems are termed **multicore**.
- Multicore systems can be more efficient than multiple chips with single cores because:
 - On-chip communication is faster than between-chip communication.
 - One chip with multiple cores uses significantly less power than multiple single-core chips, an important issue for laptops as well as mobile devices.
- Note -- while multicore systems are multiprocessor systems, not all multiprocessor systems are multicore.





A dual-core with two cores placed on the same chip





Clustered Systems

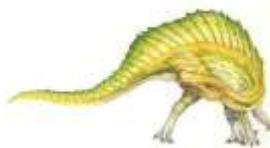
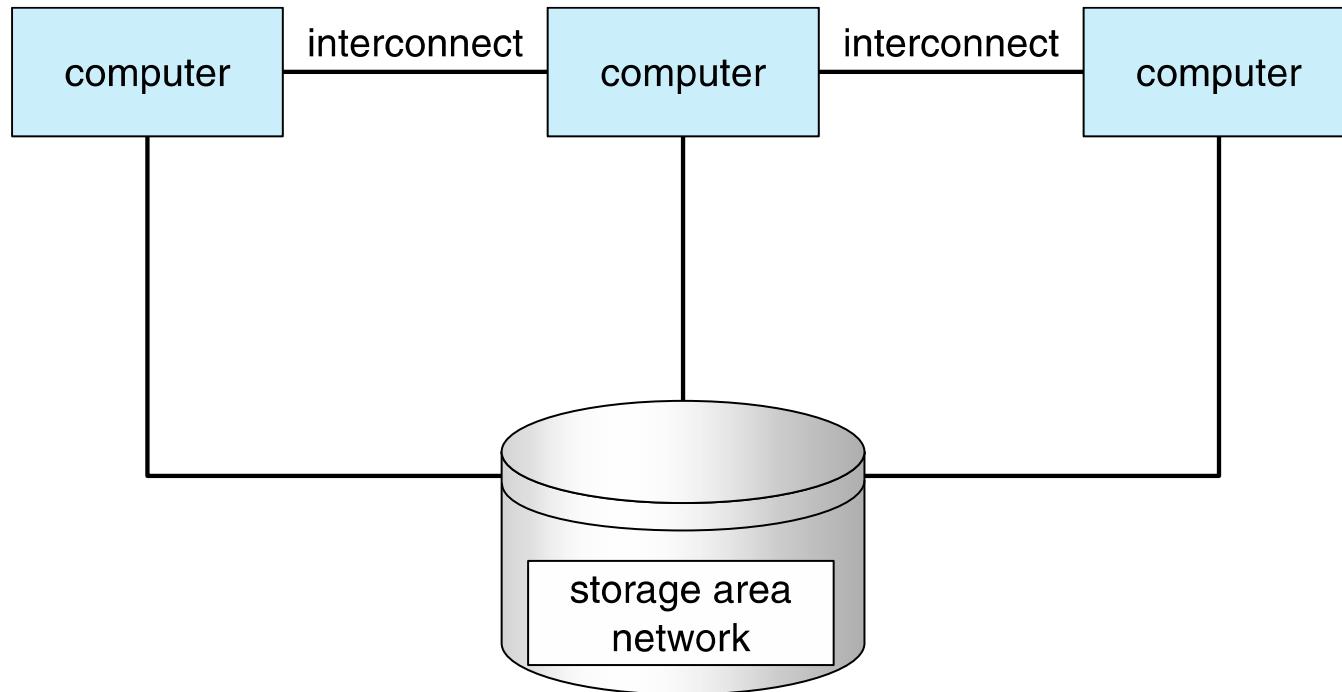
Like multiprocessor systems, but multiple systems working together

- Usually sharing storage via a **storage-area network (SAN)**
- Provides a **high-availability** service which survives failures
 - **Asymmetric clustering** has one machine in hot-standby mode
 - **Symmetric clustering** has multiple nodes running applications, monitoring each other
- Some clusters are for **high-performance computing (HPC)**
 - Applications must be written to use **parallelization**
- Some have **distributed lock manager (DLM)** to avoid conflicting operations





Clustered Systems





Multiprogrammed System

- Single user cannot keep CPU and I/O devices busy at all times
- **Multiprogramming** organizes jobs (code and data) so CPU always has one to execute
- A subset of total jobs in system is kept in memory
- Batch systems:
 - One job selected and run via **job scheduling**
 - When it has to wait (for I/O for example), OS switches to another job
- Interactive systems:
 - Logical extension of batch systems -- CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing





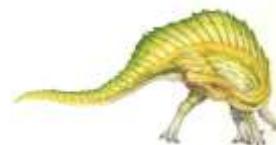
Interactive Systems

- **Response time** should be < 1 second
- Each user has at least one program executing in memory.
Such a program is referred to as a **process**
- If several processes are ready to run at the same time, we need to have **CPU scheduling**.
- If processes do not fit in memory, **swapping** moves them in and out to run
- **Virtual memory** allows execution of processes not completely in memory





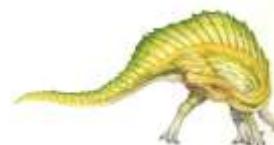
Memory Layout for Multiprogrammed System





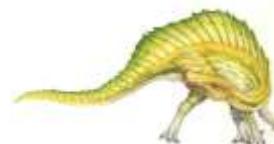
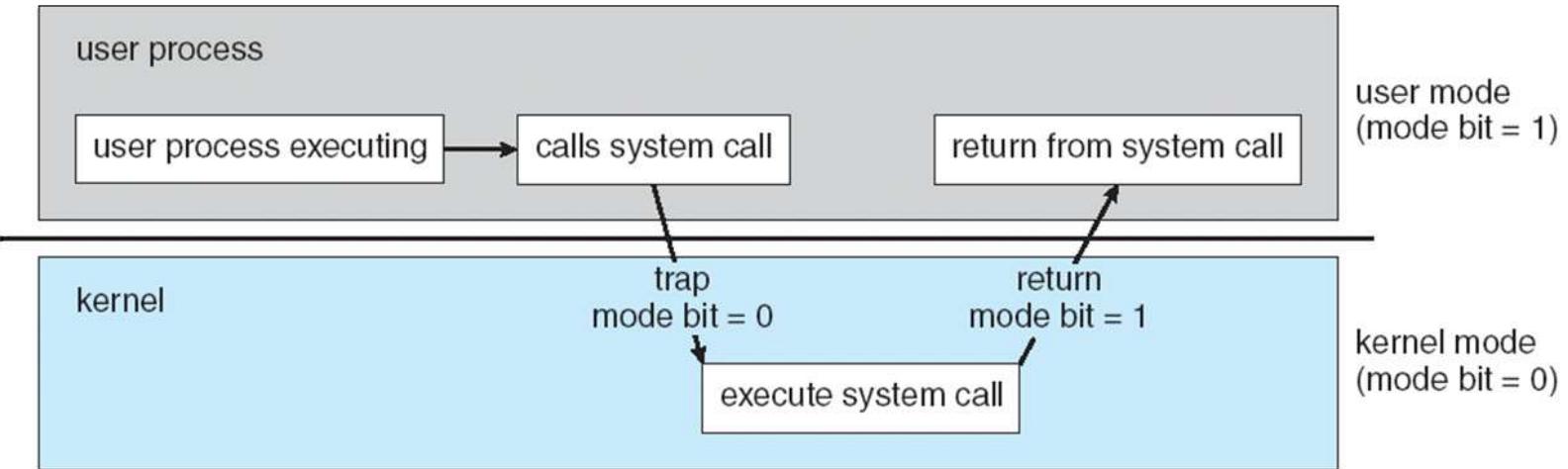
Modes of Operation

- A mechanism that allows the OS to protect itself and other system components
- Two modes:
 - **User mode**
 - **Kernel mode**
- Mode bit (0 or 1) provided by hardware
 - Provides ability to distinguish when system is running user code or kernel code
 - Some instructions designated as **privileged**, only executable in kernel mode
 - Systems call by a user asking the OS to perform some function changes from user mode to kernel mode.
 - Return from a system call resets the mode to user mode.





Transition from User to Kernel Mode

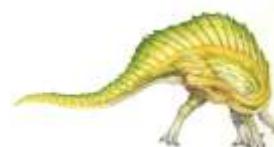




Timer

To prevent process to be in infinite loop (process hogging resources), a **timer** is used, which is a hardware device.

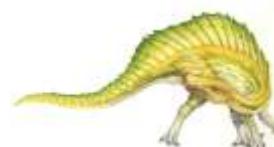
- Timer is a counter that is decremented by the physical clock.
- Timer is set to interrupt the computer after some time period
- Operating system sets the counter (privileged instruction)
- When counter reaches the value zero, and interrupt is generated.
- The OS sets up the value of the counter before scheduling a process to regain control or terminate program that exceeds allotted time





Process Management

- A process is a program in execution. It is a unit of work within the system. Program is a ***passive entity***, process is an ***active entity***.
- Process needs resources to accomplish its task
 - CPU, memory, I/O, files, etc.
 - Initialization data
- Process termination requires reclaim of any reusable resources
- A thread is a basic unit of CPU utilization within a process.
 - Single-threaded process. Instructions are executed sequentially, one at a time, until completion
 - Process has one **program counter** specifying location of next instruction to execute
- Multi-threaded process has one program counter per thread
- Typically, a system has many processes, some user, some operating system running concurrently on one or more CPUs
 - Concurrency by multiplexing the CPUs among the threads





Process Management Activities

The operating system is responsible for the following activities in connection with process management:

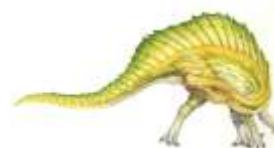
- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling





Memory Management

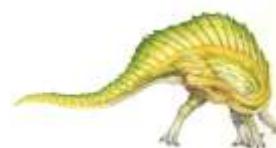
- To execute a program all (or part) of the instructions must be in memory
- All (or part) of the data that is needed by the program must be in memory.
- Memory management determines what is in memory and when
 - Optimizing CPU utilization and computer response to users
- Memory management activities
 - Keeping track of which parts of memory are currently being used and by whom
 - Deciding which processes (or parts thereof) and data to move into and out of memory
 - Allocating and deallocating memory space as needed





Storage Management

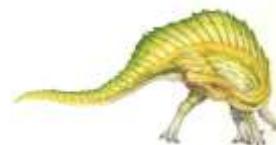
- OS provides uniform, logical view of information storage
- Abstracts physical properties to logical storage unit - **file**
- Files are stored in a number of different storage medium.
 - Disk
 - Flash Memory
 - Tape
- Each medium is controlled by device drivers (i.e., disk drive, tape drive)
 - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)





File System Management

- Files usually organized into directories
- Access control on most systems to determine who can access what
- OS activities include
 - Creating and deleting files and directories
 - Primitives to manipulate files and directories
 - Mapping files onto secondary storage
 - Backup files onto stable (non-volatile) storage media





Secondary-Storage Management

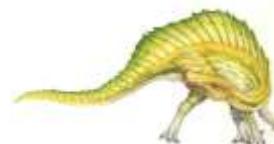
- Usually disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time
- Proper management is of central importance
- Entire speed of computer operation hinges on disk subsystem and its algorithms
- OS activities
 - Free-space management
 - Storage allocation
 - Disk scheduling
- Some storage need not be fast
 - Tertiary storage includes optical storage, magnetic tape
 - Still must be managed – by OS or applications





Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
 - If it is, information used directly from the cache (fast)
 - If not, data copied to cache and used there
- Cache are smaller (size-wise) than storage being cached
 - Cache management important design problem
 - Cache size and replacement policy

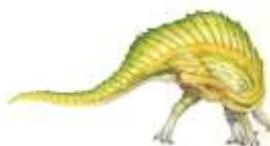




Performance of Various Levels of Storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

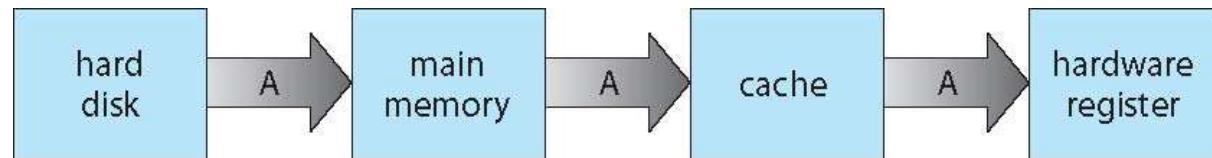
Movement between levels of storage hierarchy can be explicit or implicit





Migration of data “A” from Disk to Register

- Multitasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy



- Multiprocessor environment must provide **cache coherency** in hardware such that all CPUs have the most recent value in their cache
- Distributed environment situation even more complex
 - Several copies of a datum can exist
 - Various solutions covered in Chapter 17





I/O Subsystem

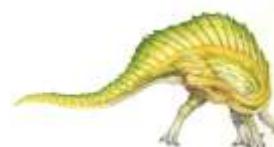
- One purpose of an operating system is to hide peculiarities of hardware devices from the user
- I/O subsystem responsible for
 - Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
 - General device-driver interface
 - Drivers for specific hardware devices





Protection and Security

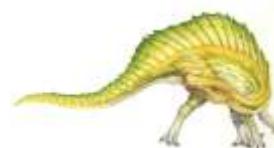
- **Protection** – A mechanism for controlling access of processes (or users) to resources defined by the OS
- **Security** – A defense of the system against internal and external attacks
 - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
 - User identities (**user IDs**, security IDs) include name and associated number, one per user
 - User ID is associated with all files and processes of that user to determine access control
 - Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file
 - **Privilege escalation** allows user to change to effective ID with more rights





Virtualization

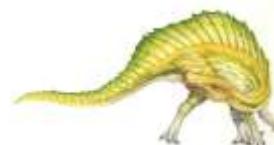
- Allows operating systems to run applications within other OSes
 - Vast and growing industry
- **Emulation** used when the source CPU type is different from the target type (i.e., PowerPC to Intel x86)
 - Generally slowest method
 - When computer language not compiled to native code – **Interpretation**
- **Virtualization** – OS natively compiled for CPU, running **guest** OSes also natively compiled
 - Consider VMware running WinXP guests, each running applications, all on native WinXP **host** OS
 - **VMM** (virtual machine Manager) provides virtualization services





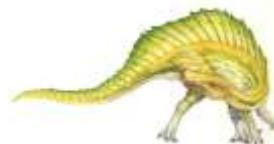
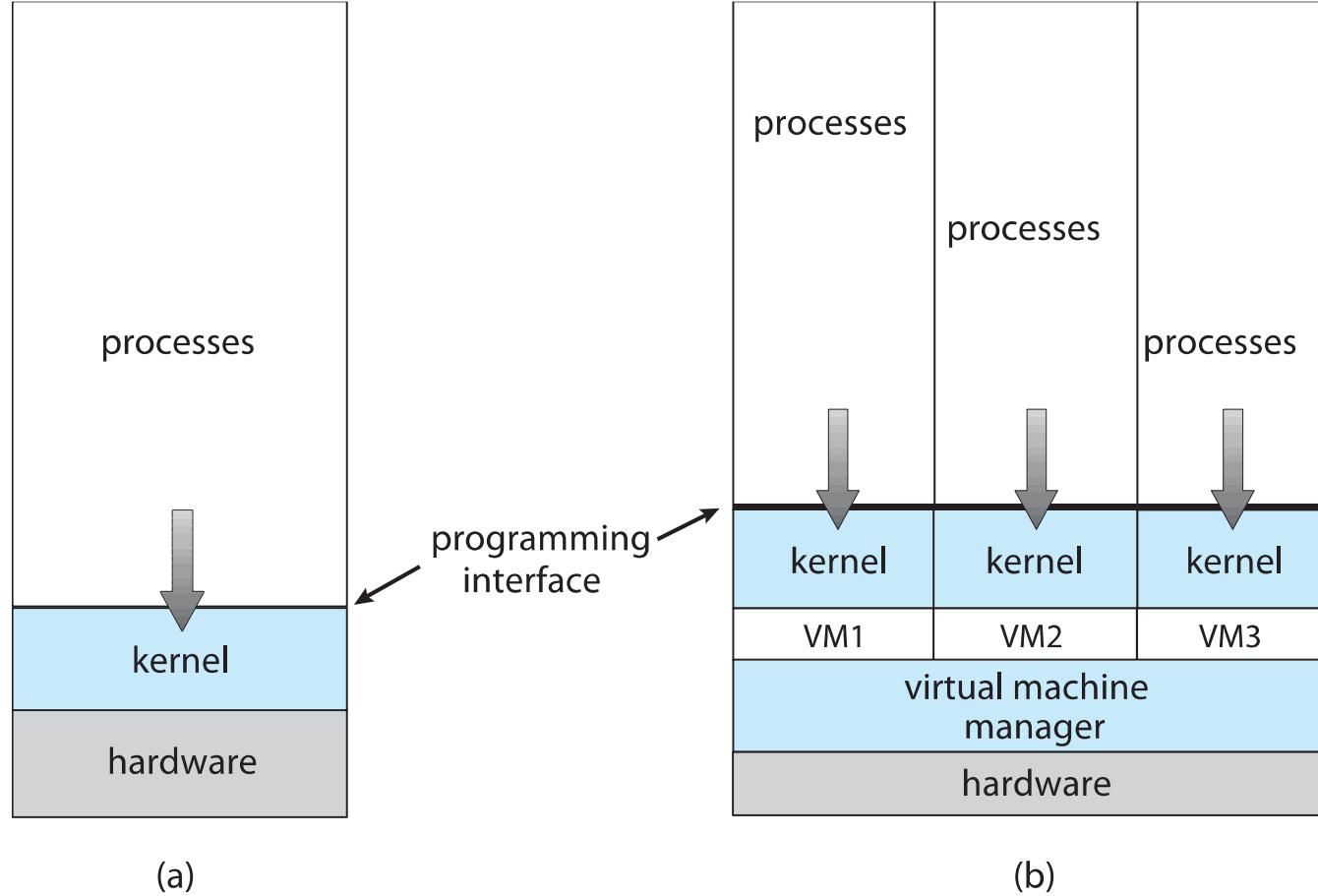
Virtualization on Laptops and Destops

- A VMM allow the user to install multiple operating systems to run application written for operating systems other than the native host.
 - Apple laptop running Mac OS X host Windows as a guest
 - Developing apps for multiple OSes without having multiple systems
 - Testing applications without having multiple systems
 - Executing and managing compute environments within data centers





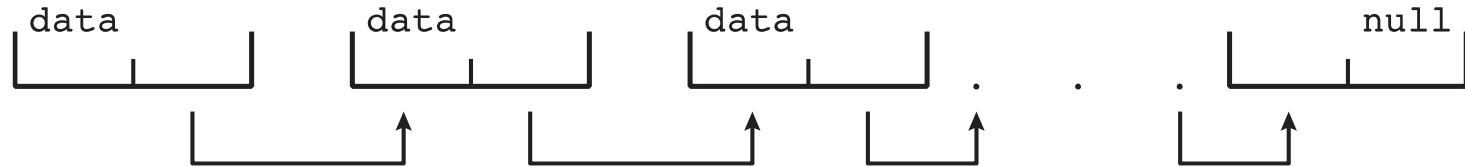
Virtualization Architecture Structure



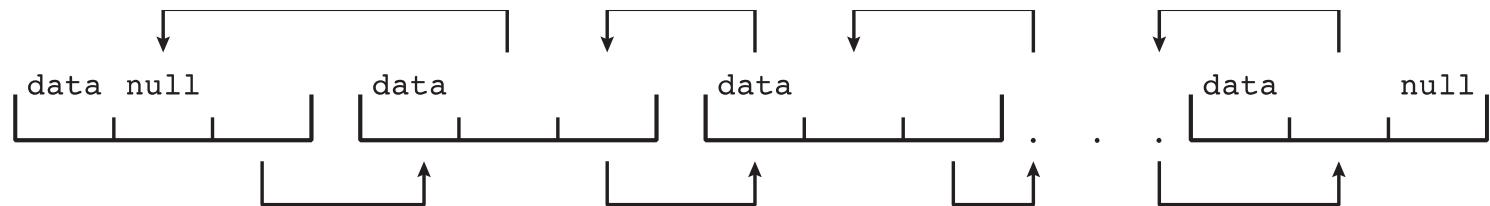


Kernel Data Structures

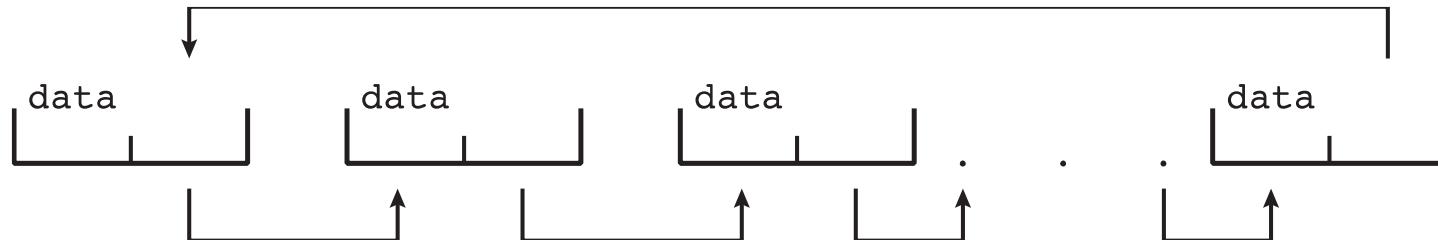
- Many -- similar to standard programming data structures
- ***Singly linked list***



- ***Doubly linked list***

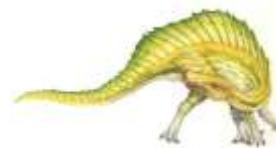
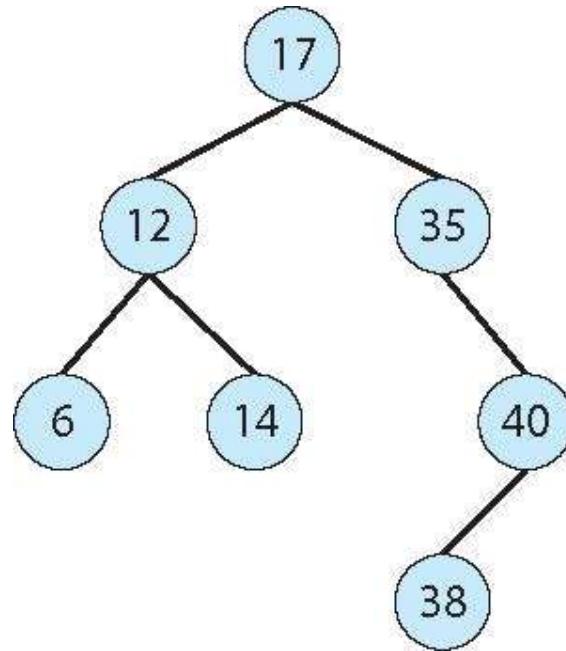


- ***Circular linked list***





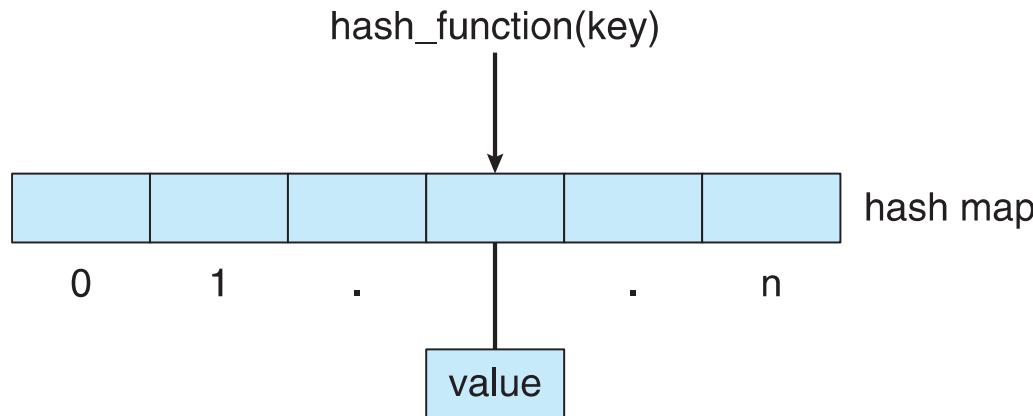
Binary search tree





Kernel Data Structures

- Hash function can create a hash map



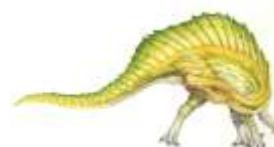
- Bitmap – string of n binary digits representing the status of n items
- Linux data structures defined in
 - include** files `<linux/list.h>`, `<linux/kfifo.h>`,
`<linux/rbtree.h>`





Computing Environments - Traditional

- Stand-alone general purpose machines
- But blurred as most systems interconnect with others (i.e., the Internet)
- **Portals** provide web access to internal systems
- **Network computers (thin clients)** are like Web terminals
- Mobile computers interconnect via **wireless networks**
- Networking becoming ubiquitous – even home systems use **firewalls** to protect home computers from Internet attacks





Computing Environments - Mobile

- Handheld smartphones, tablets, etc
- What is the functional difference between them and a “traditional” laptop?
- Extra features – more OS features (GPS -- Waze)
- Allows new types of apps like ***augmented reality***
- Use IEEE 802.11 wireless, or cellular data networks for connectivity
- Leaders are **Apple iOS** and **Google Android**





Computing Environments – Distributed

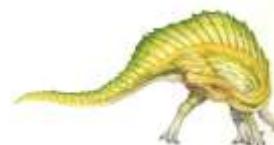
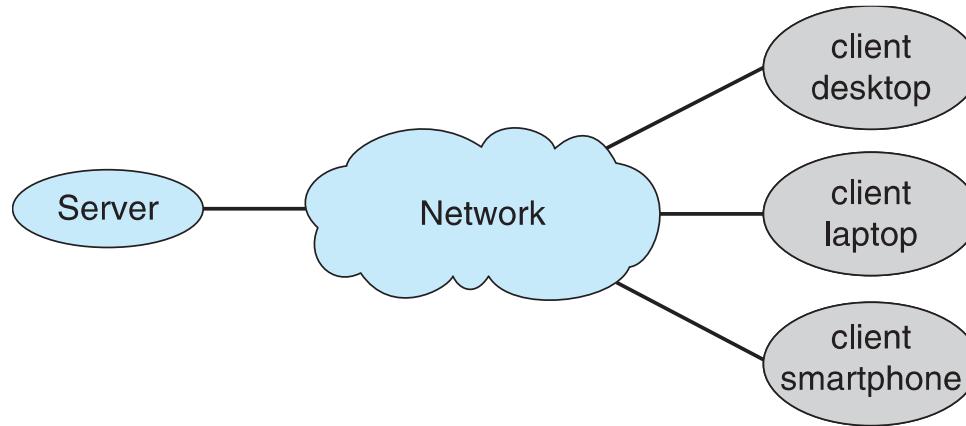
- Collection of separate, possibly heterogeneous, systems networked together
 - **Network** is a communications path, **TCP/IP** most common
 - ▶ **Local Area Network (LAN)**
 - ▶ **Wide Area Network (WAN)**
 - ▶ **Metropolitan Area Network (MAN)**
 - ▶ **Personal Area Network (PAN)**
- **Network Operating System** provides features to allow sharing of data between systems across a network.
 - Communication scheme allows systems to exchange messages
 - Illusion of a single system





Computing Environments – Client-Server

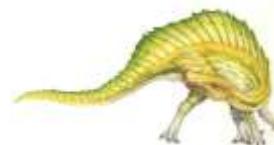
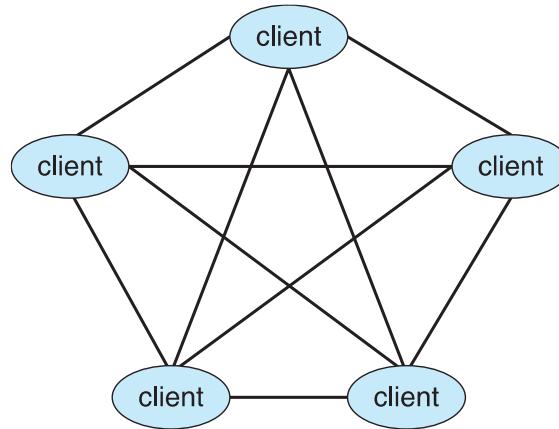
- Dumb terminals supplanted by smart PCs
- Many systems now **servers**, responding to requests generated by **clients**
 - **Compute-server system** provides an interface to client to request services (i.e., database)
 - **File-server system** provides interface for clients to store and retrieve files





Computing Environments - Peer-to-Peer

- Another model of distributed system. P2P does not distinguish clients and servers
 - Instead all nodes are considered peers
 - Each node may act as client, server, or both
 - Node must join P2P network
 - ▶ Registers its service with central lookup service on network, or
 - ▶ Broadcast request for service and respond to requests for service via ***discovery protocol***
 - Examples include Napster and Gnutella, **Voice over IP (VoIP)** such as Skype





Computing Environments – Cloud Computing

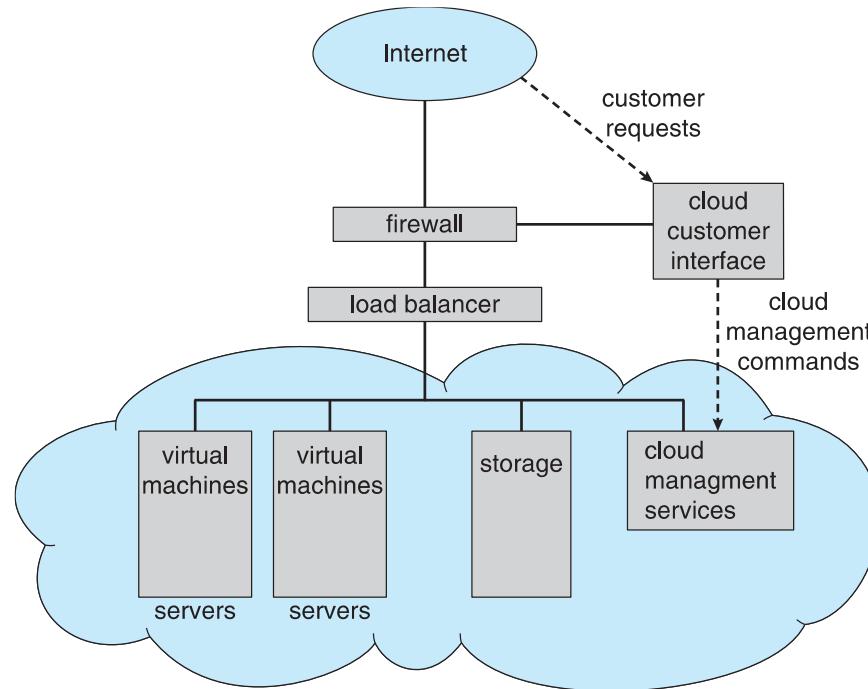
- Delivers computing, storage, even apps as a service across a network
- Logical extension of virtualization because it uses virtualization as the base for its functionality.
 - Amazon **EC2** has thousands of servers, millions of virtual machines, petabytes of storage available across the Internet, pay based on usage
- Many types
 - **Public cloud** – available via Internet to anyone willing to pay
 - **Private cloud** – run by a company for the company's own use
 - **Hybrid cloud** – includes both public and private cloud components
 - Software as a Service (**SaaS**) – one or more applications available via the Internet (i.e., word processor)
 - Platform as a Service (**PaaS**) – software stack ready for application use via the Internet (i.e., a database server)
 - Infrastructure as a Service (**IaaS**) – servers or storage available over Internet (i.e., storage available for backup use)





Computing Environments – Cloud Computing

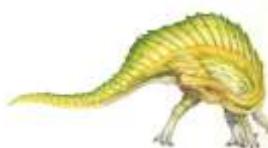
- Cloud computing environments composed of traditional OSes, plus VMs, plus cloud management tools
 - Internet connectivity requires security like firewalls
 - Load balancers spread traffic across multiple applications





Computing Environments – Real-Time Systems

- Real-time embedded systems most prevalent form of computers
 - Vary considerable, special purpose, limited purpose OS,
real-time OS
 - Use expanding
- Many other special computing environments as well
 - Some have OSes, some perform tasks without an OS
- Real-time OS has well-defined fixed time constraints
 - Processing **must** be done within constraint
 - Correct operation only if constraints met



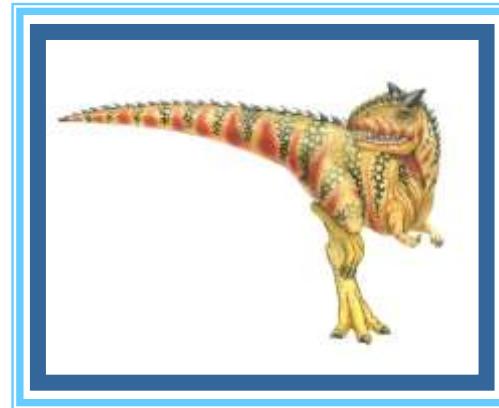


Open-Source Operating Systems

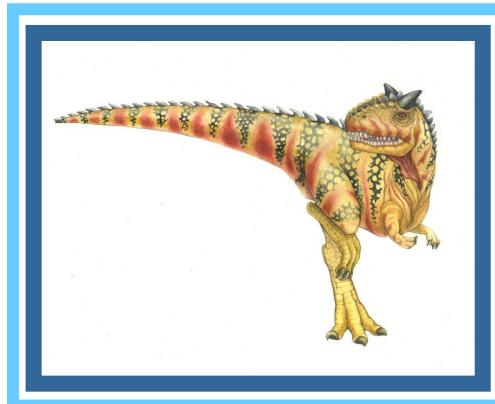
- Operating systems made available in source-code format rather than just binary **closed-source**
- Counter to the **copy protection** and **Digital Rights Management (DRM)** movement
- Started by **Free Software Foundation (FSF)**, which has “copyleft” **GNU Public License (GPL)**
- Examples include **GNU/Linux** and **BSD UNIX** (including core of **Mac OS X**), and many more
- Can use VMM like VMware Player (Free on Windows), Virtualbox (open source and free on many platforms - <http://www.virtualbox.com>)
 - Use to run guest operating systems for exploration



End of Chapter 1



Chapter 2: Operating-System Structures





Objectives

- To describe the services an operating system provides to users, processes, and other systems
- To discuss the various ways of structuring an operating system
- To explain how operating systems are installed and customized and how they boot

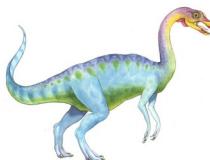




Operating System Services

- Operating systems provide an environment for the execution of programs.
- Operating systems provides certain services to:
 - Programs
 - Users of those programs
- Basically two types of services:
 - Set of operating-system services provides functions that are helpful to the user:
 - Set of operating-system functions for ensuring the efficient operation of the system itself via resource sharing

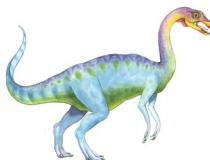




OS Services Helpful to the User

- **User interface** - Almost all operating systems have a **user interface (UI)**. This interface can take several forms:
 - **Command-Line (CLI)** -- uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options).
 - **Graphics User Interface (GUI)** -- the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text..
 - **Batch Interface** -- commands and directives to control those commands are entered into files, and those files are executed
- Some systems provide two or all three of these variations.
- **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
- **I/O operations** - A running program may require I/O, which may involve a file or an I/O device

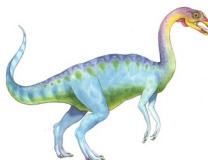




OS Services Helpful to the User (Cont.)

- **File-system manipulation** - Programs need to read and write files and directories, create and delete them, search them, list file information, permission management.
- **Communications** – Processes may exchange information, on the same computer or between computers over a network
 - Communications may be via shared memory or through message passing (packets moved by the OS)
- **Error detection** – OS needs to be constantly aware of possible errors
 - May occur in the CPU and memory hardware, in I/O devices, in user program
 - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
 - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system





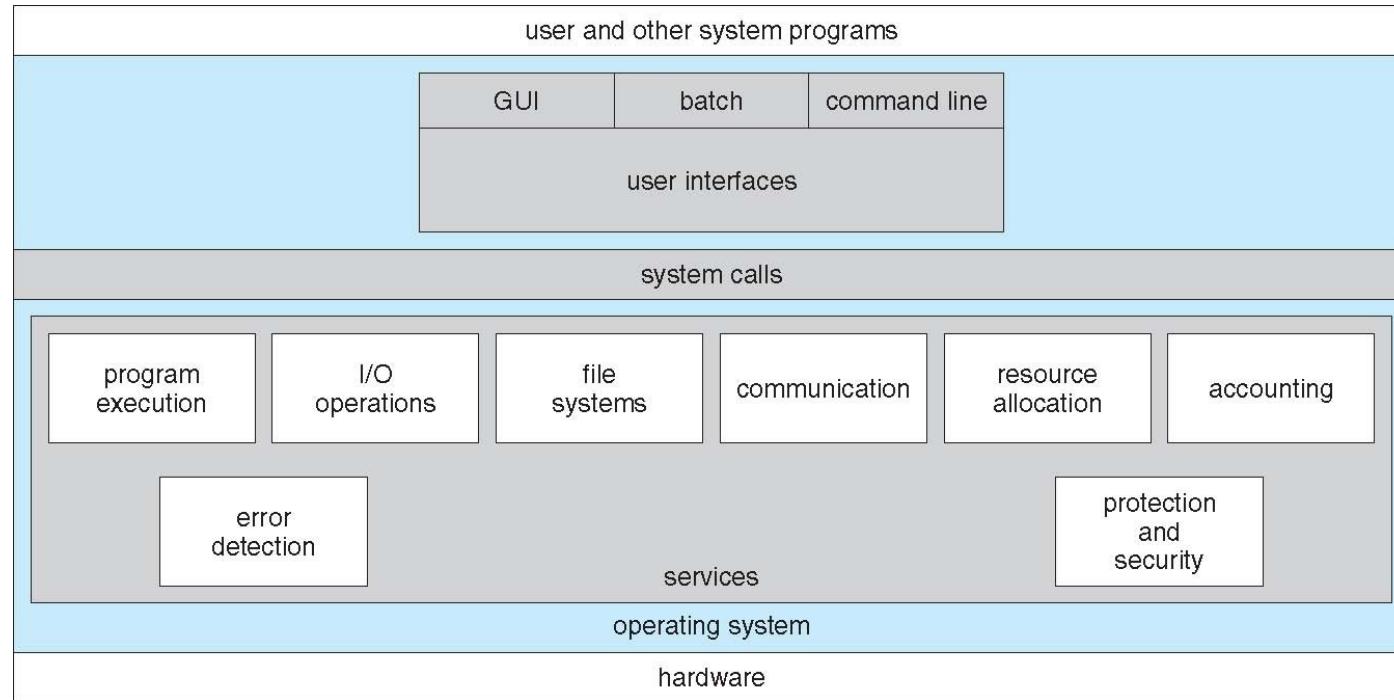
OS Services for Ensuring Efficient Operation

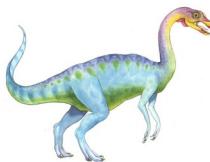
- **Resource allocation** - When multiple users or multiple jobs are running concurrently, resources must be allocated to each of them
 - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
- **Accounting** - To keep track of which users use how much and what kinds of computer resources
- **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
 - **Protection** involves ensuring that all access to system resources is controlled
 - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts





A View of Operating System Services



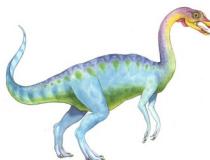


Command Interpreters (CLI)

CLI allows users to directly enter commands to be performed by the operating system.

- Some operating systems include the command interpreter in the kernel.
- Some operating systems, such as Windows and UNIX, treat the command interpreter as a special program that is running when a job is initiated or when a user first logs on.
- On systems with multiple command interpreters to choose from, the interpreters are known as **shells**.
- The main function of the command interpreter is to get and execute the next user-specified command.
- Sometimes commands built-in, sometimes just names of programs
 - If the latter, adding new features doesn't require shell modification





The Bourne shell command interpreter in Solaris

```
1. root@r6181-d5-us01:~ (ssh)
× root@r6181-d5-u... ● %1 × ssh ⚡ %2 × root@r6181-d5-us01... %3
Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbgs$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
 06:57:48 up 16 days, 10:52,  3 users,  load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem      Size  Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
                  50G   19G   28G  41% /
tmpfs           127G  520K  127G   1% /dev/shm
/dev/sda1        477M   71M   381M  16% /boot
/dev/dssd0000    1.0T  480G  545G  47% /dssd_xfs
tcp://192.168.150.1:3334/orangefs
                  12T  5.7T  6.4T  47% /mnt/orangefs
/dev/gpfs-test   23T  1.1T   22T   5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root     97653 11.2  6.6 42665344 17520636 ?  S<Ll Jul13 166:23 /usr/lpp/mmfs/bin/mmfsd
root     69849  6.6  0.0     0     0 ?      S    Jul12 181:54 [vpthread-1-1]
root     69850  6.4  0.0     0     0 ?      S    Jul12 177:42 [vpthread-1-2]
root     3829  3.0  0.0     0     0 ?      S    Jun27 730:04 [rp_thread 7:0]
root     3826  3.0  0.0     0     0 ?      S    Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfsd
-r-x----- 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfsd
[root@r6181-d5-us01 ~]#
```





Bourne Shell Command Interpreter

Default

New Info Close Execute Bookmarks

Default Default

```
PBG-Mac-Pro:~ pbgs$ w
15:24 up 56 mins, 2 users, load averages: 1.51 1.53 1.65
USER    TTY    FROM          LOGIN@ IDLE WHAT
pbgs    console -           14:34      50 -
pbgs    s000   -           15:05      - w

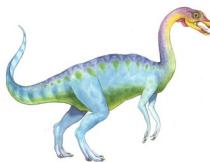
PBG-Mac-Pro:~ pbgs$ iostat 5
              disk0      disk1      disk10     cpu      load average
              KB/t tps MB/s    KB/t tps MB/s    KB/t tps MB/s us sy id 1m 5m 15m
 33.75 343 11.30  64.31 14 0.88  39.67 0 0.02 11 5 84 1.51 1.53 1.65
  5.27 320 1.65  0.00 0 0.00  0.00 0 0.00 4 2 94 1.39 1.51 1.65
  4.28 329 1.37  0.00 0 0.00  0.00 0 0.00 5 3 92 1.44 1.51 1.65

^C
PBG-Mac-Pro:~ pbgs$ ls
Applications           Music           WebEx
Applications (Parallel) Packages       config.log
Desktop                Pictures        getsmartdata.txt
Documents               Public          imp
Downloads              Sites           log
Dropbox                 Thumbs.db      panda-dist
Library                Virtual Machines prob.txt
Movies                 Volumes         scripts

PBG-Mac-Pro:~ pbgs$ pwd
/Users/pbgs

PBG-Mac-Pro:~ pbgs$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=2.257 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=1.262 ms
^C
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 1.262/1.760/2.257/0.498 ms
PBG-Mac-Pro:~ pbgs$
```





Graphical User Interfaces (GUI)

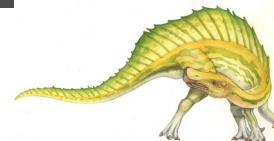
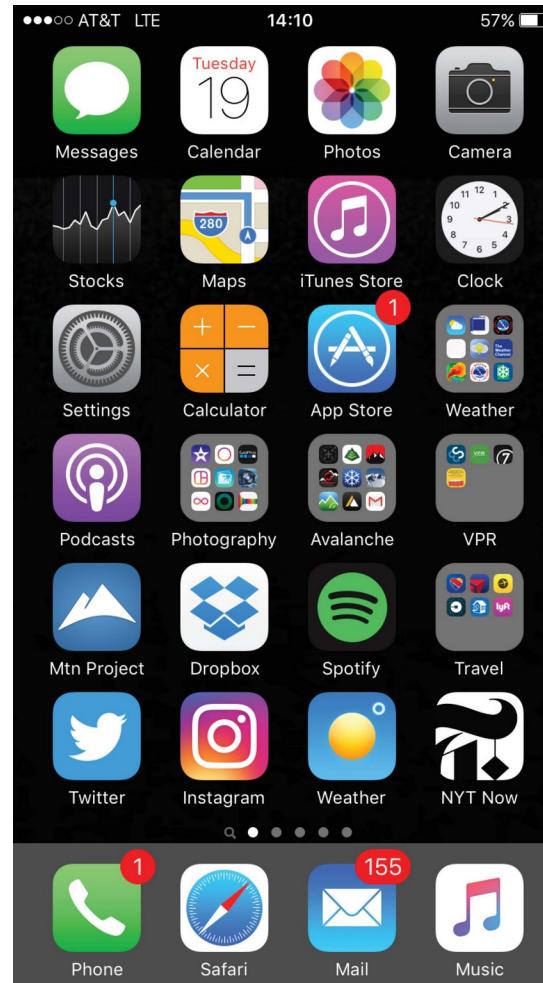
- User-friendly **desktop** metaphor interface
 - Usually mouse, keyboard, and monitor
 - **Icons** represent files, programs, actions, etc
 - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
 - Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
 - Microsoft Windows is GUI with CLI “command” shell
 - Apple Mac OS X is “Aqua” GUI interface with UNIX kernel underneath and shells available
 - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)





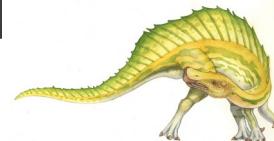
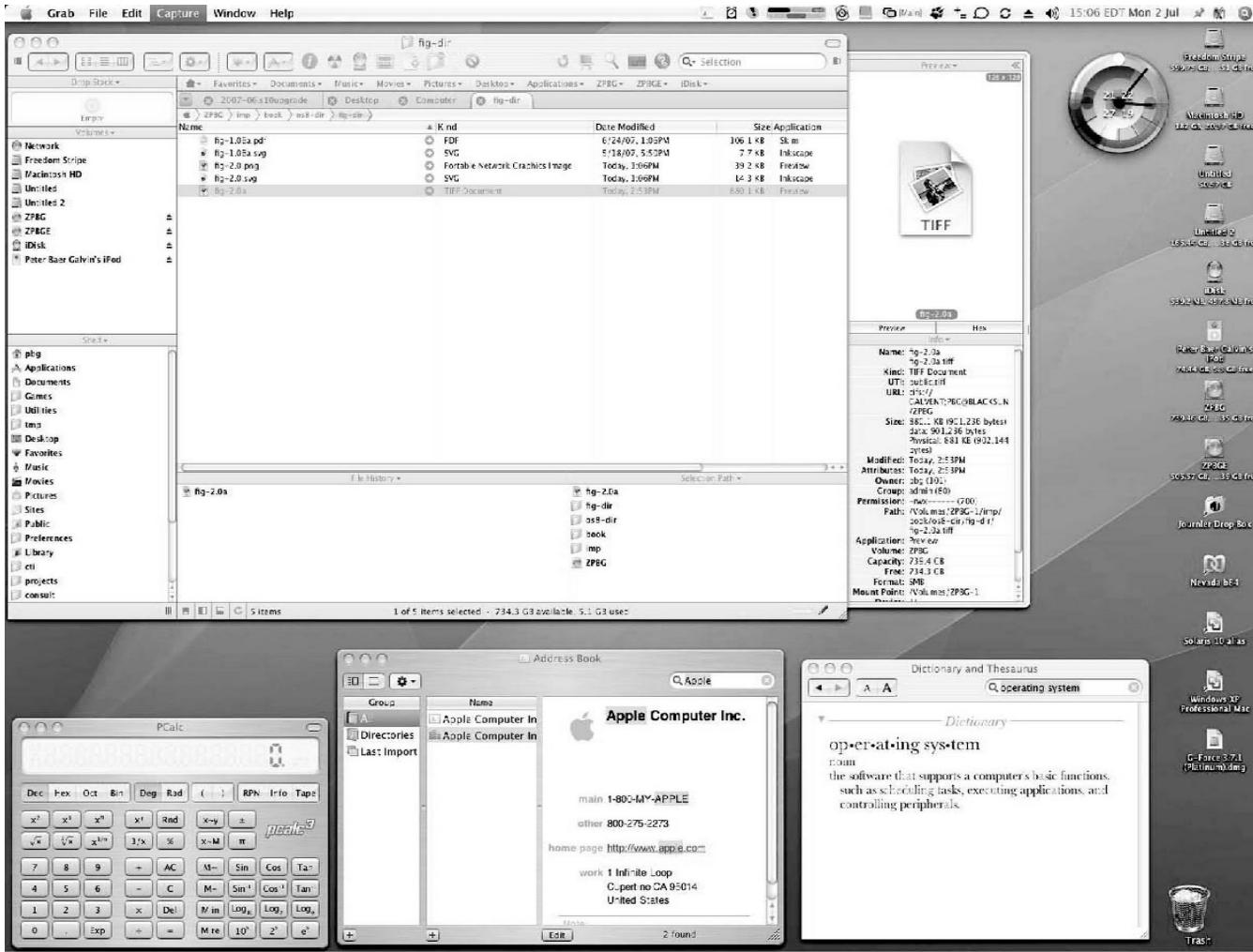
Touchscreen Interfaces

- Touchscreen devices require new interfaces
 - Mouse not possible or not desired
 - Actions and selection based on gestures
 - Virtual keyboard for text entry
- Voice commands.





The Mac OS X GUI



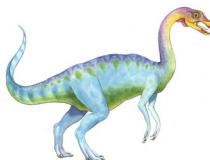


System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call
- Three most common APIs are:
 - Win32 API for Windows,
 - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X),
 - Java API for the Java virtual machine (JVM)

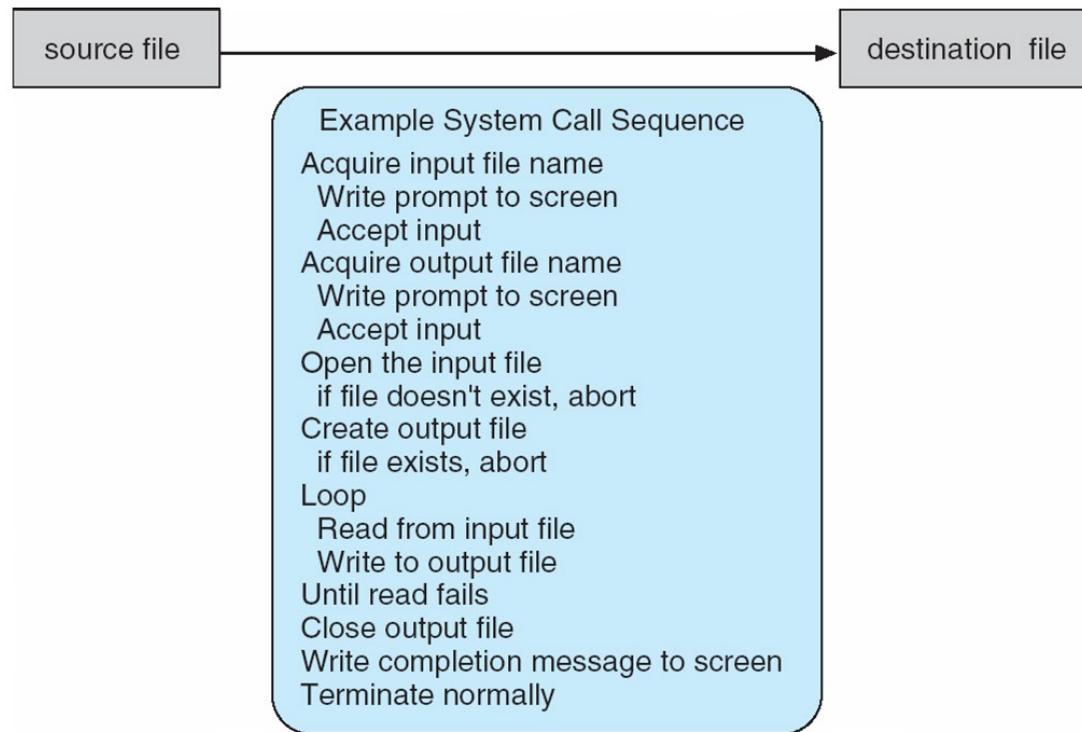
Note that the system-call names used throughout this text are generic

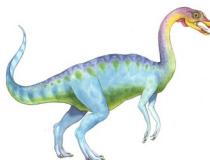




Example of System Calls

- System call sequence to copy the contents of one file to another file





Example of Standard API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t      read(int fd, void *buf, size_t count)
```

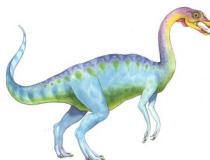
return function parameters
value name

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns -1.





System Call Implementation

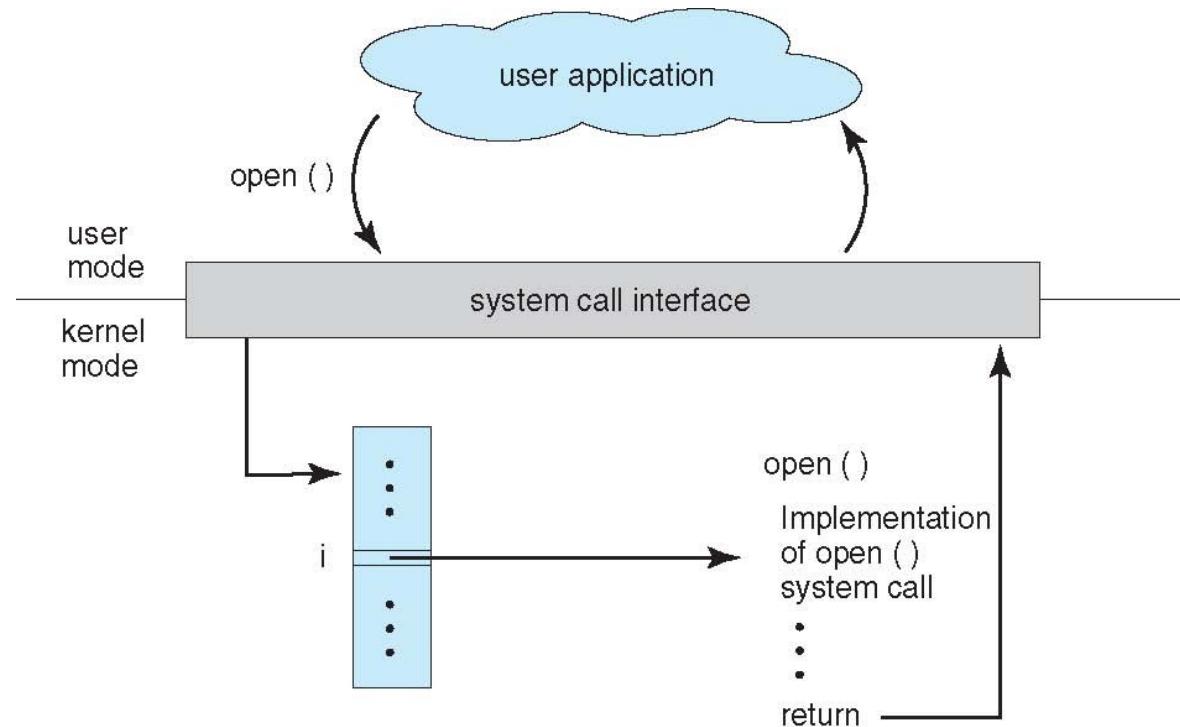
- Typically, a number is associated with each system call
 - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need not know a thing about how the system call is implemented
 - Just needs to obey the API and understand what the OS will do as a result call
 - Most details of OS interface hidden from programmer by API
 - ▶ Managed by run-time support library (set of functions built into libraries included with compiler)

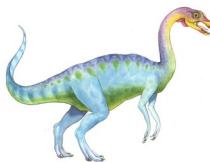




System Call -- OS Relationship

The handling of a user application invoking the open() system call

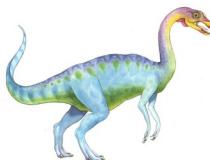




System Call Parameter Passing

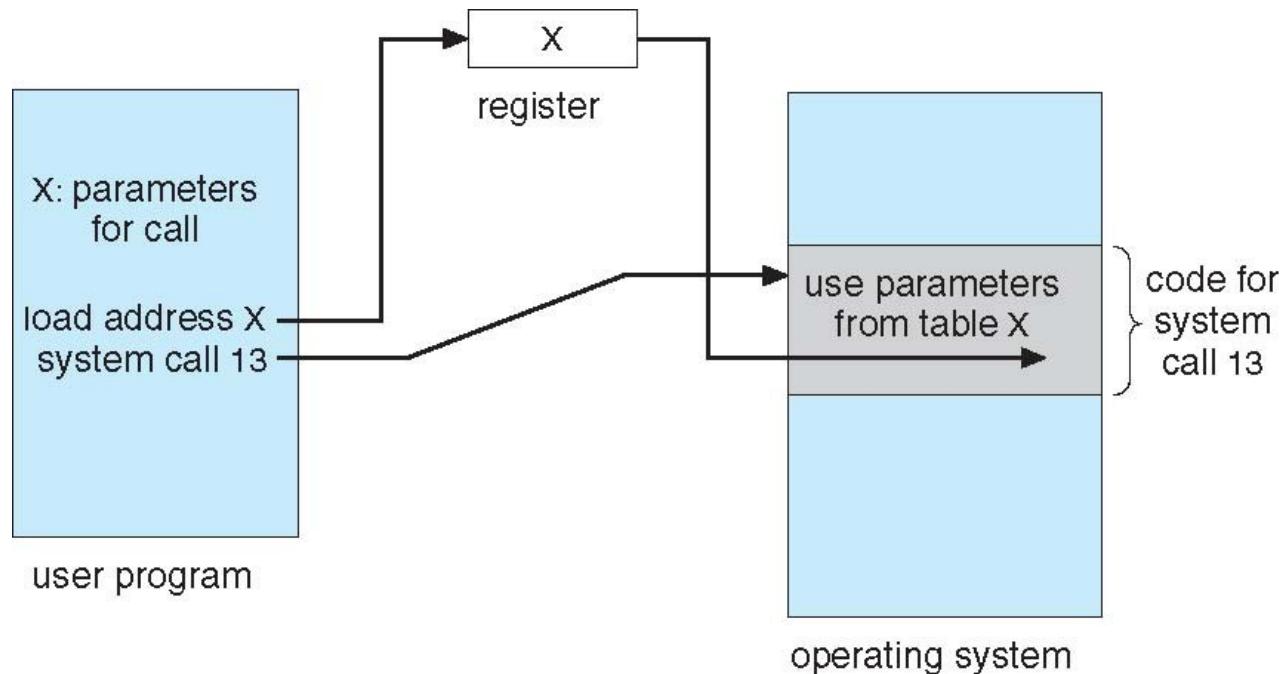
- Often, more information is required than simply identity of desired system call
 - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
 - Simplest: pass the parameters in registers
 - ▶ In some cases, may be more parameters than registers
 - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
 - ▶ This approach taken by Linux and Solaris
 - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
 - Block and stack methods do not limit the number or length of parameters being passed

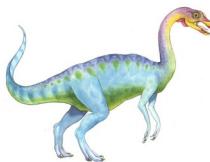




Parameter Passing via Table

x points to a block of parameters. x is loaded into a register

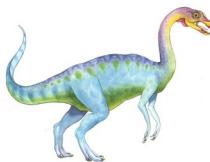




Types of System Calls

- System calls can be grouped roughly into six major categories:
 - Process control,
 - File manipulation,
 - Device manipulation,
 - Information maintenance,
 - Communications,
 - Protection.
- The figure in the slide # 28 summarizes the types of system calls normally provided by an operating system.

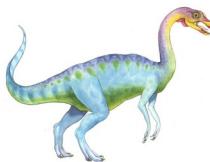




System Calls – Process Control

- create process, terminate process
- end, abort
- load, execute
- get process attributes, set process attributes
- wait for time
- wait event, signal event
- allocate and free memory
- Dump memory if error
- **Debugger** for determining **bugs, single step** execution
- **Locks** for managing access to shared data between processes

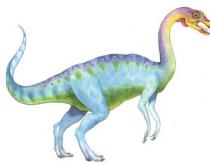




System Calls – File Management

- Create file
- Delete file
- Open and Close file
- Read, Write, Reposition
- Get and Set file attributes





System Calls – Device Management

- request device, release device
- read, write, reposition
- get device attributes, set device attributes
- logically attach or detach devices





System Calls -- Information Maintenance

- get time or date,
- set time or date
- get system data,
- set system data
- get and set process, file, or device attributes





System Calls – Communications

- create, delete communication connection
- if **message passing model**:
 - send, receive message
 - ▶ To **host name** or **process name**
 - ▶ From **client** to **server**
- If **shared-memory model**:
 - create and gain access to memory regions
- transfer status information
- attach and detach remote devices





System Calls -- Protection

- Control access to resources
- Get and set permissions
- Allow and deny user access

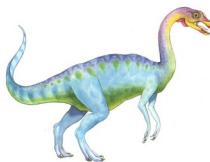




Examples of Windows and Unix System Calls

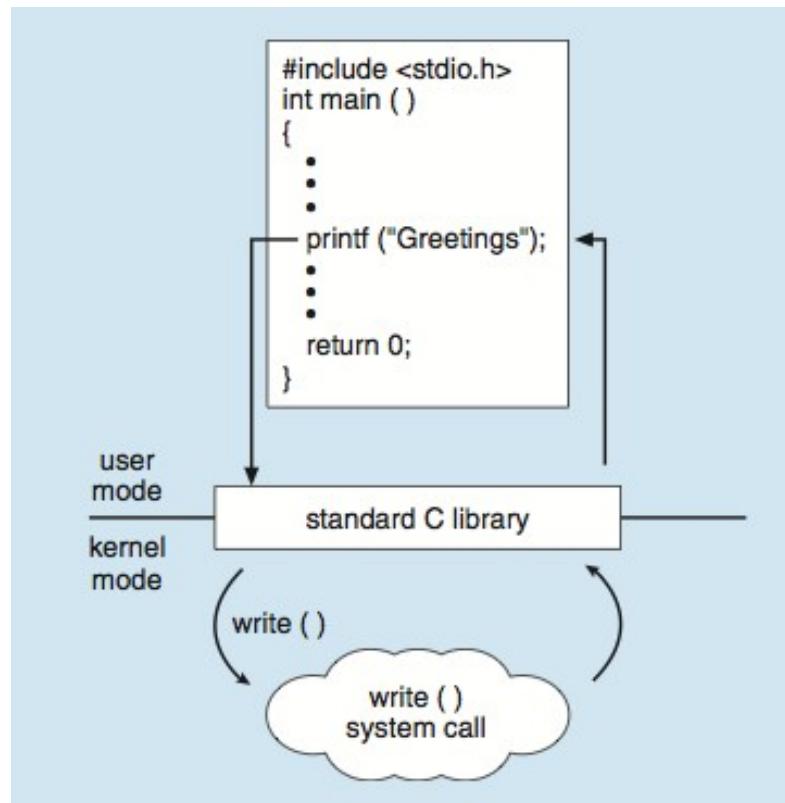
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

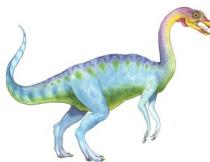




Example -- Standard C Library

C program invoking printf() library call, which calls write() system call

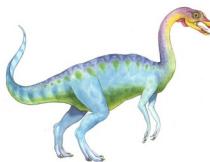




System Programs

- System programs provide a convenient environment for program development and execution.
- Some of them are simply user interfaces to system calls. Others are considerably more complex.
- They can be divided into:
 - File manipulation
 - Status information sometimes stored in a File modification
 - Programming language support
 - Program loading and execution
 - Communications
 - Background services
 - Application programs
- Most users' view of the operation system is defined by system programs, not the actual system calls





System Programs

■ File management

- Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories

■ Status information

- Some programs ask the system for information - date, time, amount of available memory, disk space, number of users
- Others programs provide detailed performance, logging, and debugging information
- Typically, these programs format and print the output to the terminal or other output devices
- Some systems implement a **registry** - used to store and retrieve configuration information





System Programs (Cont.)

■ File modification

- Text editors to create and modify files
- Special commands to search contents of files or perform transformations of the text

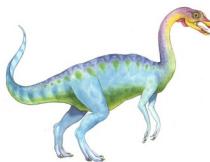
■ Programming-language support - Compilers, assemblers, debuggers and interpreters sometimes provided

■ Program loading and execution- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language

■ Communications - Provide the mechanism for creating virtual connections among processes, users, and computer systems

- Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another





System Programs (Cont.)

■ Background Services

- Launch at boot time
 - ▶ Some for system startup, then terminate
 - ▶ Some from system boot to shutdown
- Provide facilities like disk checking, process scheduling, error logging, printing
- Run in user context not kernel context
- Known as **services, subsystems, daemons**

■ Application programs

- Don't pertain to system
- Run by users
- Not typically considered part of OS
- Launched by command line, mouse click, finger poke





Operating System Design and Implementation

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system – batch, time sharing, single user, multiuser, distributed, real-time
- Two groups in terms of defining goals:
 - **User goals** – operating system should be convenient to use, easy to learn, reliable, safe, and fast
 - **System goals** – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- Specifying and designing an OS is highly creative task of **software engineering**





Mechanisms and Polices

- Important principle to separate
 - Policy:** *What* will be done?
 - Mechanism:** *How* to do it?
- Mechanisms determine how to do something, policies decide what will be done
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)

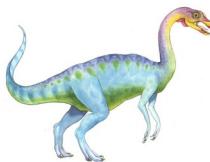




Implementation

- Much variation
 - Early Operating Systems were written in assembly language
 - Then with system programming languages like Algol, PL/1
 - Now C, C++
- Actually usually a mix of languages
 - Lowest levels in assembly
 - Main body in C
 - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- High-level language easier to **port** to other hardware
 - But slower
- **Emulation** can allow an OS to run on non-native hardware





Operating System Structure

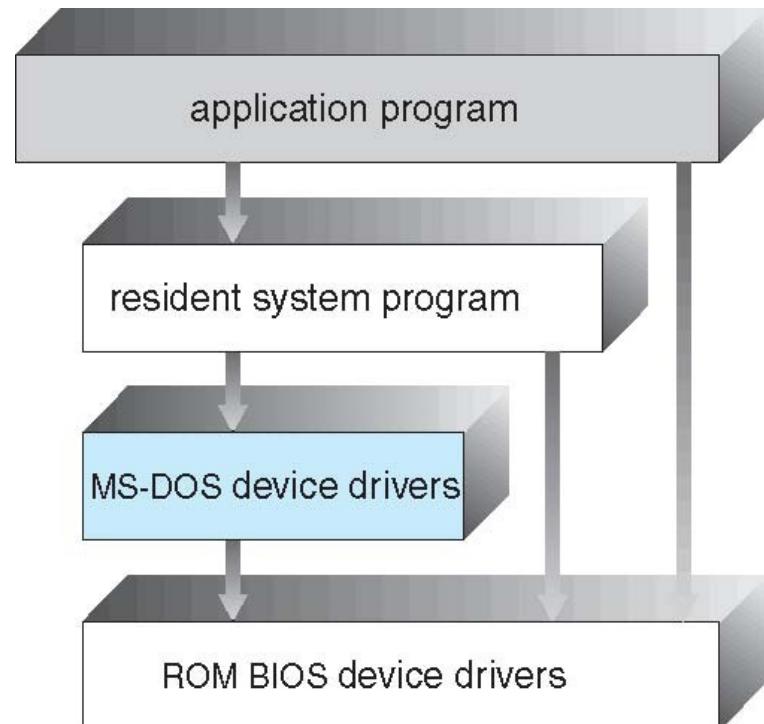
- Various ways to structure an operating system:
 - Monolithic structure
 - ▶ Simple structure – MS-DOS
 - ▶ More complex – UNIX
 - ▶ More complex – Linux
 - Layered – An abstraction
 - Microkernel - Mach

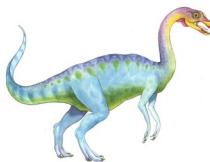




MS-DOS

- MS-DOS – written to provide the most functionality in the least amount of space
- MS-DOS was limited by hardware functionality.
 - Not divided into modules
 - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated





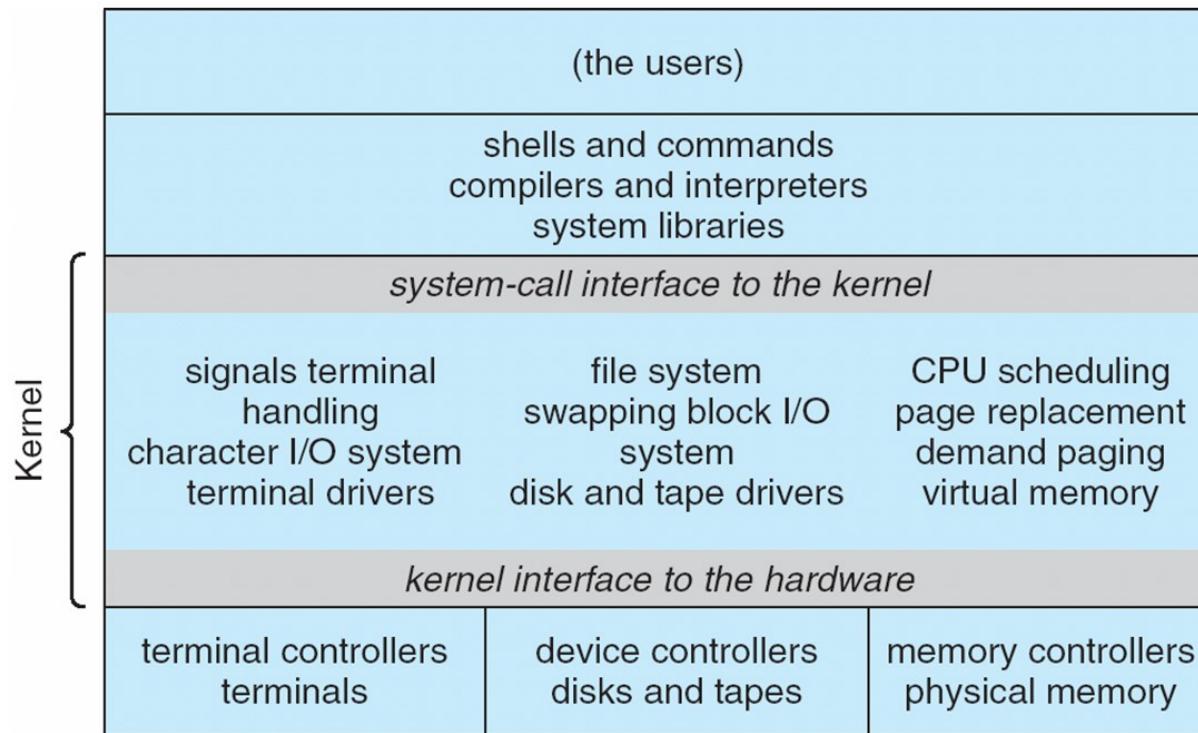
UNIX

- UNIX – the original UNIX operating system had limited structuring and was limited by hardware functionality.
- The UNIX OS consists of two separable parts
 - Systems programs
 - The kernel
 - ▶ Consists of everything below the system-call interface and above the physical hardware
 - ▶ Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level



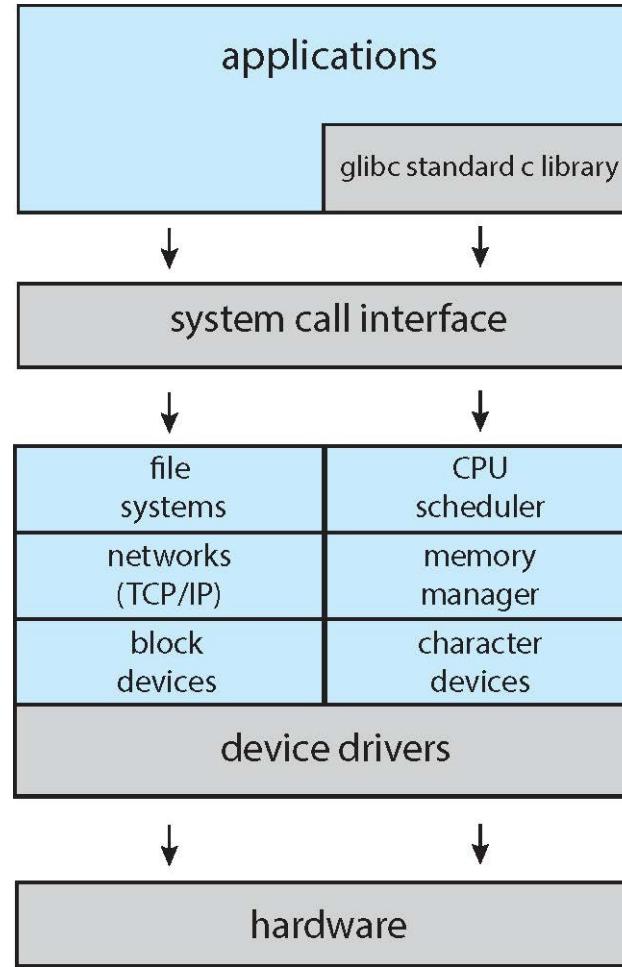


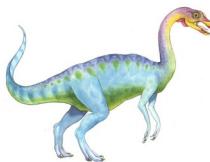
Traditional UNIX System Structure





Linux System Structure





Modularity

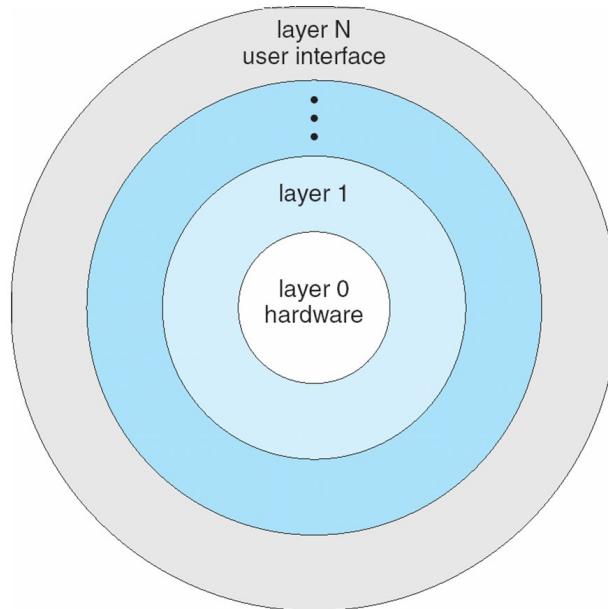
- The monolithic approach results in a situation where changes to one part of the system can have wide-ranging effects to other parts.
- Alternatively, we could design system where the operating system is divided into separate, smaller components that have specific and limited functionality. The sum of all these components comprises the kernel.
- The advantage of this modular approach is that changes in one component only affect that component, and no others, allowing system implementers more freedom when changing the inner workings of the system and in creating modular operating systems.





Layered Approach

- A system can be made modular in many ways. One method is the **layered approach**, in which the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.

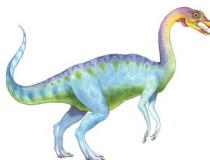




Layered Approach (Cont.)

- A typical operating-system layer -- say, layer M -- consists of data structures and a set of routines that can be invoked by higher-level layers. Layer M in turn, can invoke operations on lower-level layers.
- Each layer is implemented only with operations provided by lower-level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do.
- The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers.
- This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer.

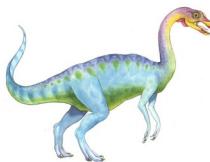




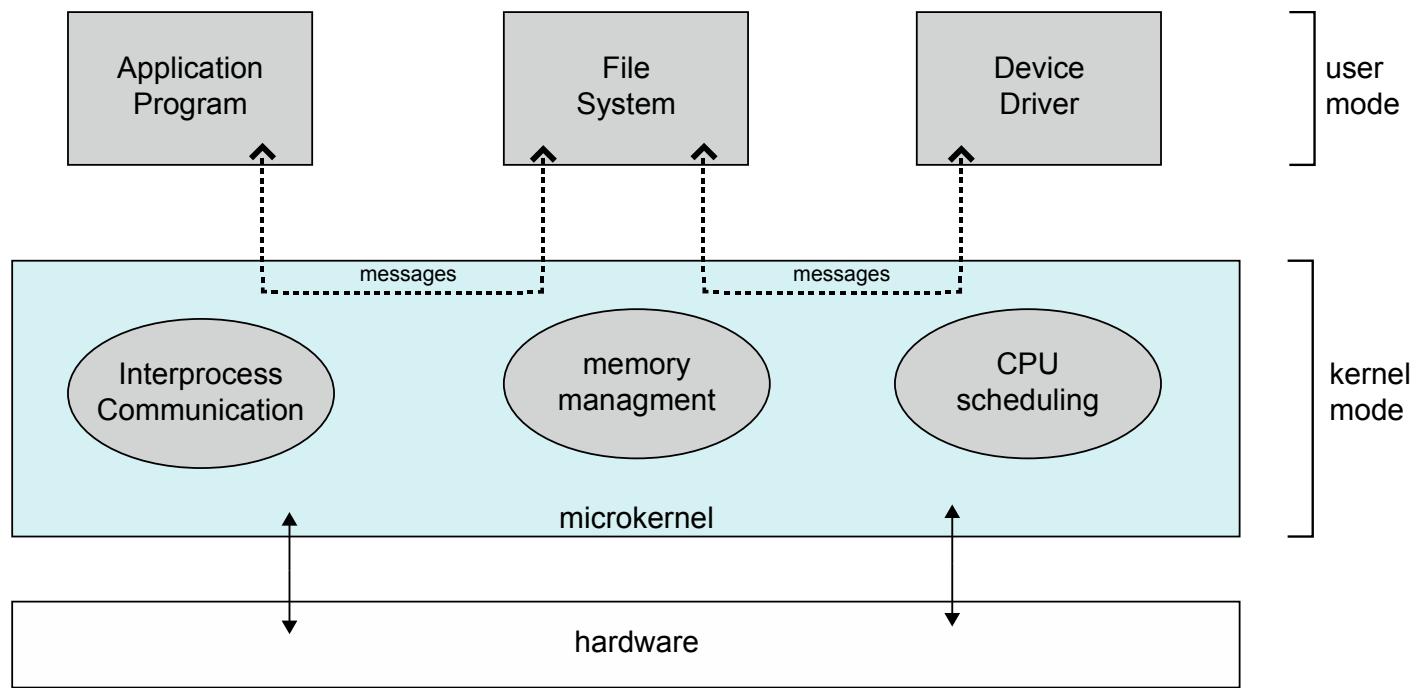
Microkernel System Structure

- Moves as much from the kernel into user space
- **Mach** example of **microkernel**
 - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between user modules using **message passing**
- Benefits:
 - Easier to extend a microkernel
 - Easier to port the operating system to new architectures
 - More reliable (less code is running in kernel mode)
 - More secure
- Detriments:
 - Performance overhead of user space to kernel space communication





Microkernel System Structure





Modules

- Many modern operating systems implement **loadable kernel modules**
 - Uses object-oriented approach
 - Each core component is separate
 - Each talks to the others over known interfaces
 - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible
 - Linux, Solaris, etc





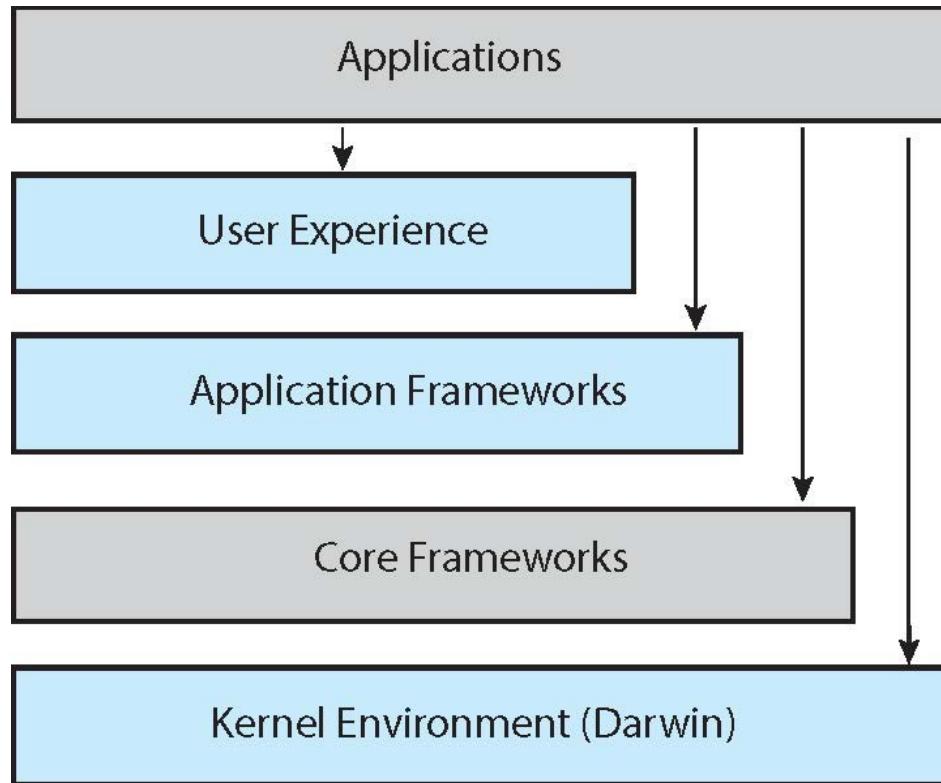
Hybrid Systems

- In practice, very few operating systems adopt a single, strictly defined structure.
- Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues.
 - For example, Linux is monolithic, because having the operating system in a single address space provides very efficient performance. However, Linux are also modular, so that new functionality can be dynamically added to the kernel.
- We cover the structure of three hybrid systems:
 - Apple Mac operating system (laptop)
 - iOS (mobile operating systems)
 - Android (mobile operating systems)





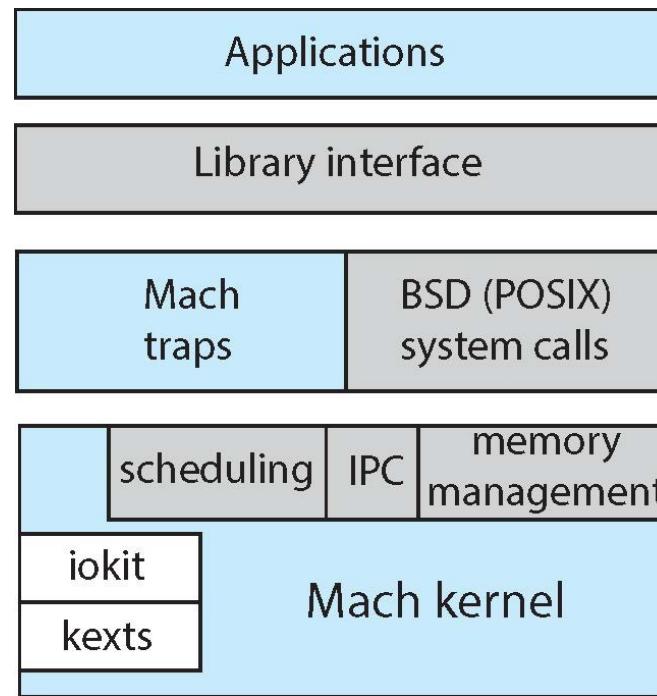
Architecture of Mac OS X and iOS





Darwin

- Darwin uses a hybrid structure, and is shown Darwin is a layered system which consists primarily of the Mach microkernel and the BSD UNIX kernel.

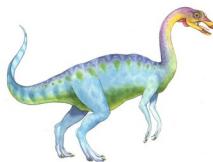




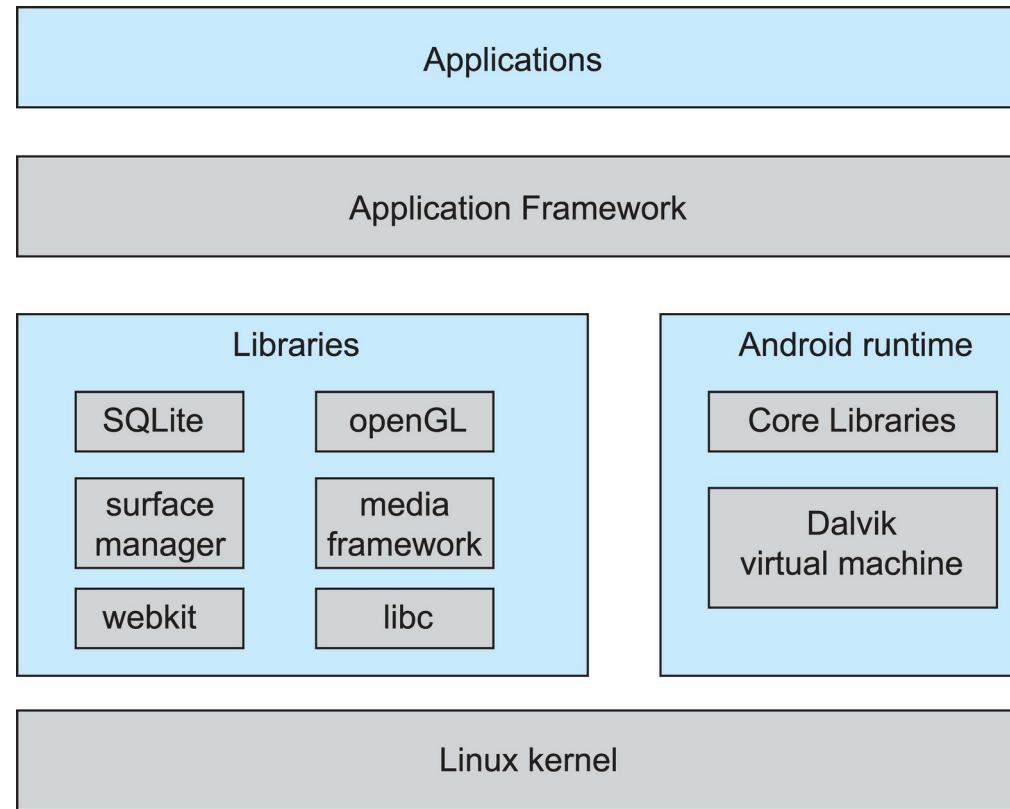
Android

- Developed by Open Handset Alliance (mostly Google)
 - Open Source
- Similar stack to iOS
- Based on Linux kernel but modified
 - Provides process, memory, device-driver management
 - Adds power management
- Runtime environment includes core set of libraries and Dalvik virtual machine
 - Apps developed in Java plus Android API
 - ▶ Java class files compiled to Java bytecode then translated to executable than runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc





Android Architecture



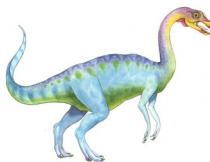


Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
- OS generate **log files** containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- Operating system failure can generate **crash dump** file containing kernel memory
- Beyond crashes, performance tuning can optimize system performance
 - Sometimes using **trace listings** of activities, recorded for analysis
 - **Profiling** is periodic sampling of instruction pointer to look for statistical trends

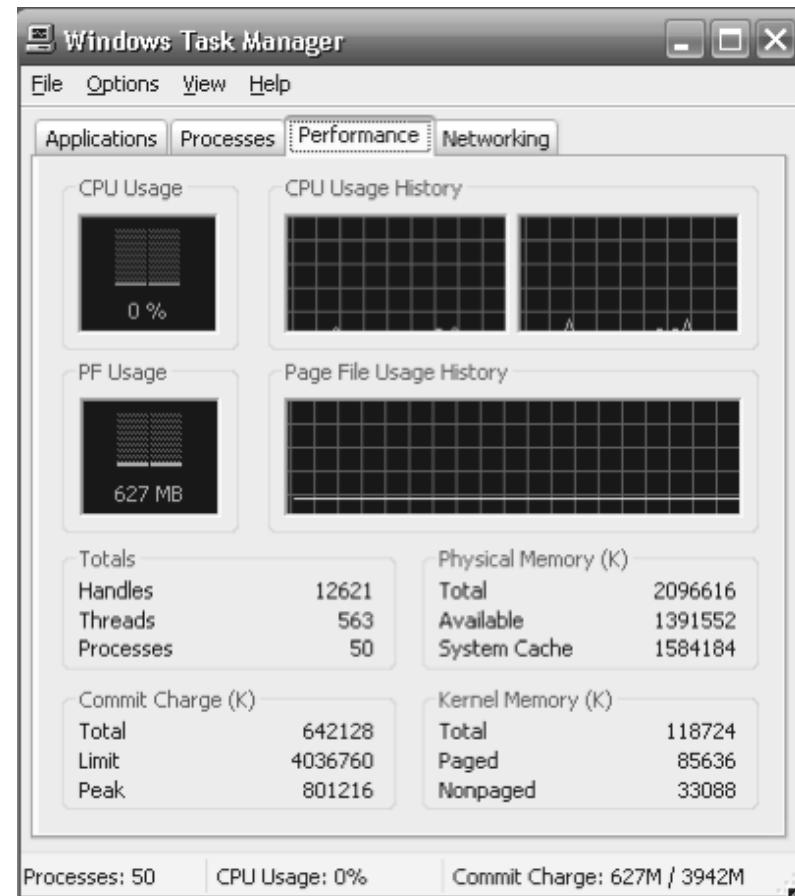
Kernighan's Law: Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”





Performance Tuning

- Improve performance by removing bottlenecks
- OS must provide means of computing and displaying measures of system behavior
- For example, “top” program or Windows Task Manager





DTrace

- DTrace tool in Solaris, FreeBSD, Mac OS X allows live instrumentation on production systems
- **Probes** fire when code is executed within a **provider**, capturing state data and sending it to **consumers** of those probes
- Example of following XEventsQueued system call move from libc library to kernel and back

```
# ./all.d `pgrep xclock` XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                                U
  0  -> _XEventsQueued                            U
  0  -> _X11TransBytesReadable                     U
  0  <- _X11TransBytesReadable                     U
  0  -> _X11TransSocketBytesReadable              U
  0  <- _X11TransSocketBytesreadable             U
  0  -> ioctl                                      U
  0    -> ioctl                                    K
  0    -> getf                                     K
  0      -> set_active_fd                         K
  0      <- set_active_fd                         K
  0    <- getf                                    K
  0    -> get_udatamodel                         K
  0    <- get_udatamodel                         K
...
  0    -> releasef                               K
  0      -> clear_active_fd                      K
  0      <- clear_active_fd                      K
  0      -> cv_broadcast                          K
  0      <- cv_broadcast                         K
  0      <- releasef                            K
  0    <- ioctl                                 K
  0    <- ioctl                                 U
  0  <- _XEventsQueued                         U
  0 <- XEventsQueued                           U
```





Dtrace (Cont.)

- DTrace code to record amount of time each process with UserID 101 is in running mode (on CPU) in nanoseconds

```
sched:::on-cpu
uid == 101
{
    self->ts = timestamp;
}

sched:::off-cpu
self->ts
{
    @time[execname] = sum(timestamp - self->ts);
    self->ts = 0;
}
```

```
# dtrace -s sched.d
dtrace: script 'sched.d' matched 6 probes
^C
      gnome-settings-d          142354
      gnome-vfs-daemon          158243
      dsdm                      189804
      wnck-applet                200030
      gnome-panel                 277864
      clock-applet                374916
      mapping-daemon              385475
      xscreensaver                514177
      metacity                     539281
      Xorg                         2579646
      gnome-terminal                5007269
      mixer_applet2                7388447
      java                        10769137
```

Figure 2.21 Output of the D code.





Operating System Generation

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site
- **SYSGEN** program obtains information concerning the specific configuration of the hardware system
 - Used to build system-specific compiled kernel or system-tuned
 - Can generate more efficient code than one general kernel



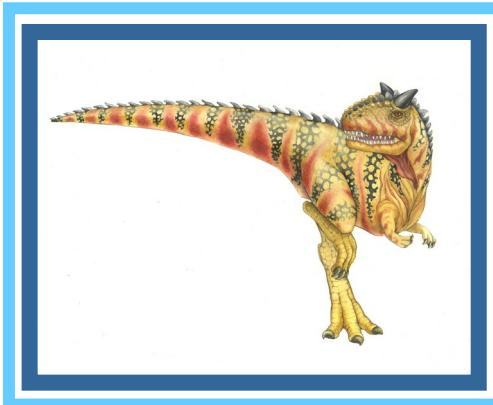


System Boot

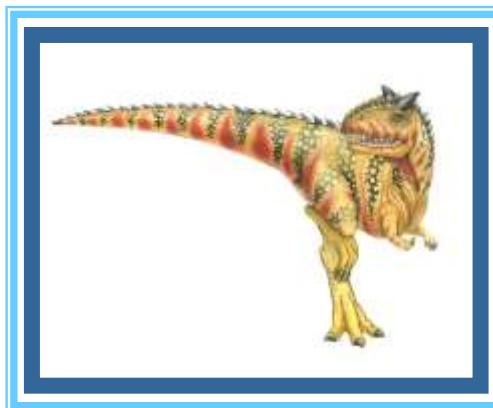
- When power initialized on system, execution starts at a fixed memory location
 - Firmware ROM used to hold initial boot code
- Operating system must be made available to hardware so hardware can start it
 - Small piece of code – **bootstrap loader**, stored in **ROM** or **EEPROM** locates the kernel, loads it into memory, and starts it
 - Sometimes two-step process where **boot block** at fixed location loaded by ROM code, which loads bootstrap loader from disk
- Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then **running**



End of Chapter 2



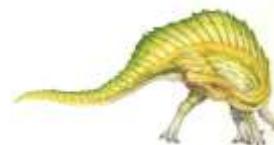
Chapter 3: Processes





Chapter 3: Processes

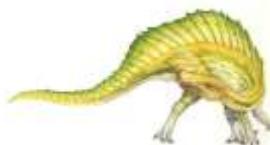
- Process Concept
- Process Scheduling
- Operations on Processes
- Inter-process Communication (IPC)
- Examples of IPC Systems
- Communication in Client-Server Systems





Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore interprocess communication using shared memory and message passing
- To describe communication in client-server systems





Process Concept

- **Process** – a program in execution; process execution must progress in sequential fashion
- Program is **passive** entity stored on disk (**executable file**), process is **active**
 - Program becomes process when executable file loaded into memory
- Execution of program started via:
 - GUI mouse clicks,
 - command line entry of its name,
 - etc
- One program can be several processes
 - Consider multiple users executing the same program





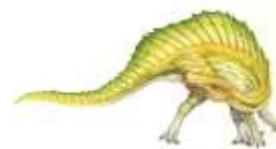
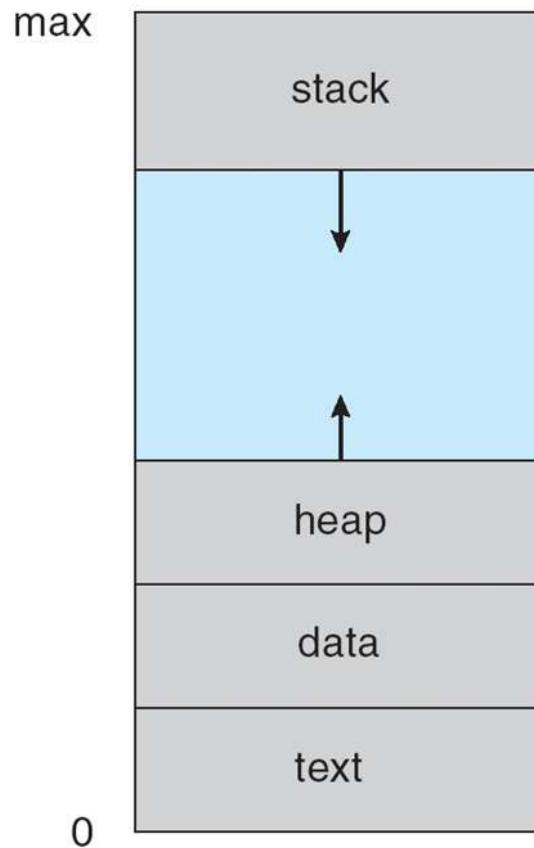
Process Structure

- A process is more than the program code, which is sometimes known as the **text** section.
- It also includes the current activity:
 - The value of the **program counter**
 - The contents of the **processor's registers**.
- It also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables)
- It also includes the **data section**, which contains global variables.
- It may also include a **heap**, which is memory that is dynamically allocated during process run time.





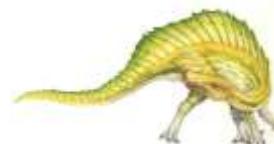
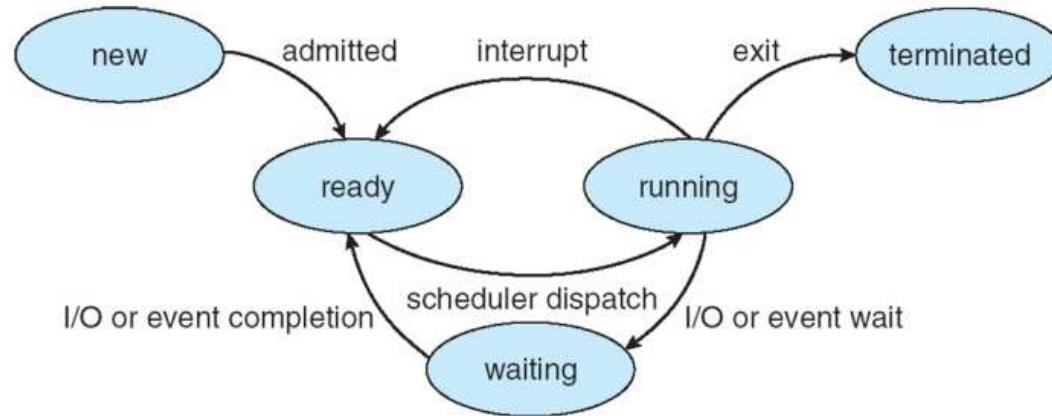
Process in Memory





Process State

- As a process executes, it changes **state**
 - **new**: The process is being created
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event to occur
 - **ready**: The process is waiting to be assigned to a processor
 - **terminated**: The process has finished execution
- Diagram of Process State





Process Control Block (PCB)

Information associated with each process
(also called **task control block**)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files





Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
 - Multiple locations can execute at once
 - ▶ Multiple threads of control -> **threads**
- Need storage for thread details, multiple program counters in PCB
- Covered in the next chapter

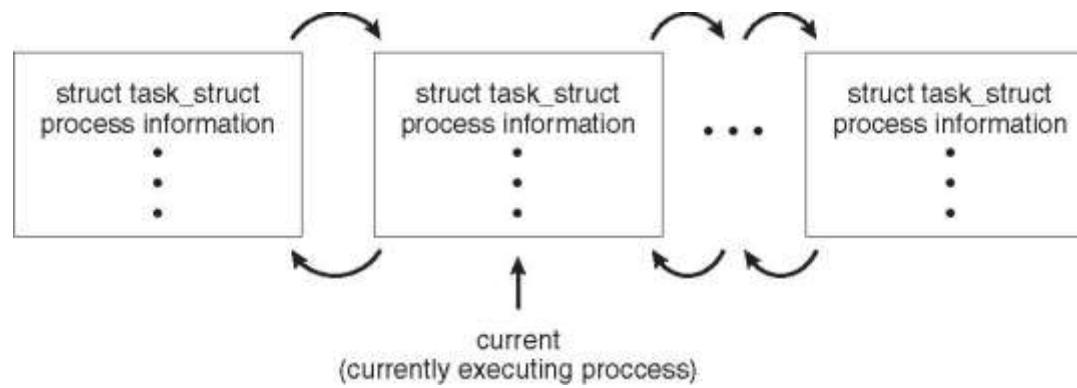




Process Representation in Linux

Represented by the C structure `task_struct`

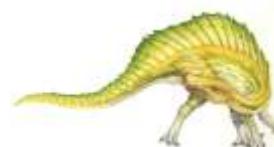
```
pid t_pid; /* process identifier */  
long state; /* state of the process */  
unsigned int time_slice /* scheduling information */  
struct task_struct *parent; /* this process's parent */  
struct list_head children; /* this process's children */  
struct files_struct *files; /* list of open files */  
struct mm_struct *mm; /* address space of this process */
```





Process Scheduling

- Maximize CPU use
 - Quickly switch processes onto CPU for time sharing
- Process “gives” up then CPU under two conditions:
 - I/O request
 - After N units of time have elapsed (need a timer)
- Once a process gives up the CPU it is added to the “ready queue”
- **Process scheduler** selects among available processes in the ready queue for next execution on CPU





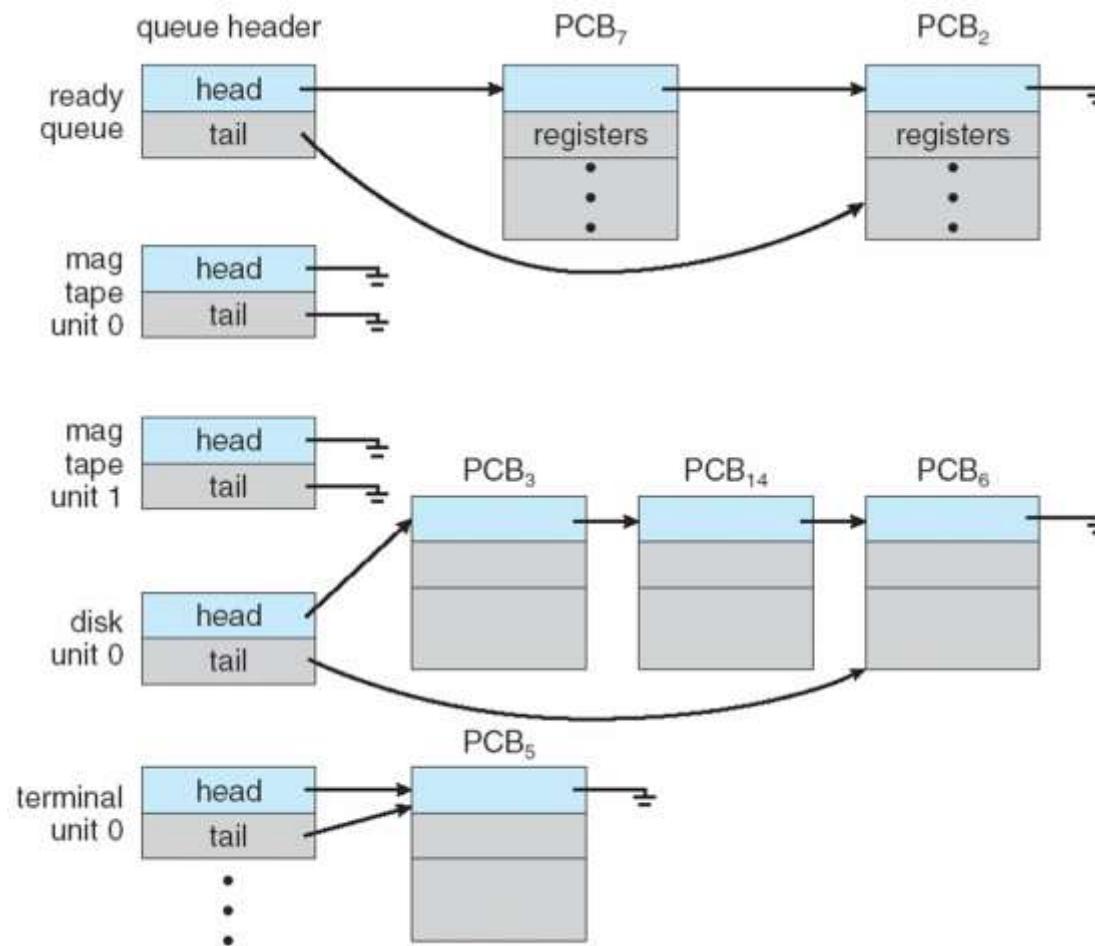
Scheduling Queues

- OS Maintains **scheduling queues** of processes
 - **Job queue** – set of all processes in the system
 - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
 - **Device queues** – set of processes waiting for an I/O device
 - Processes migrate among the various queues





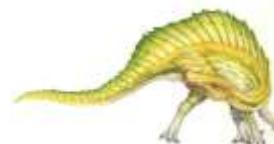
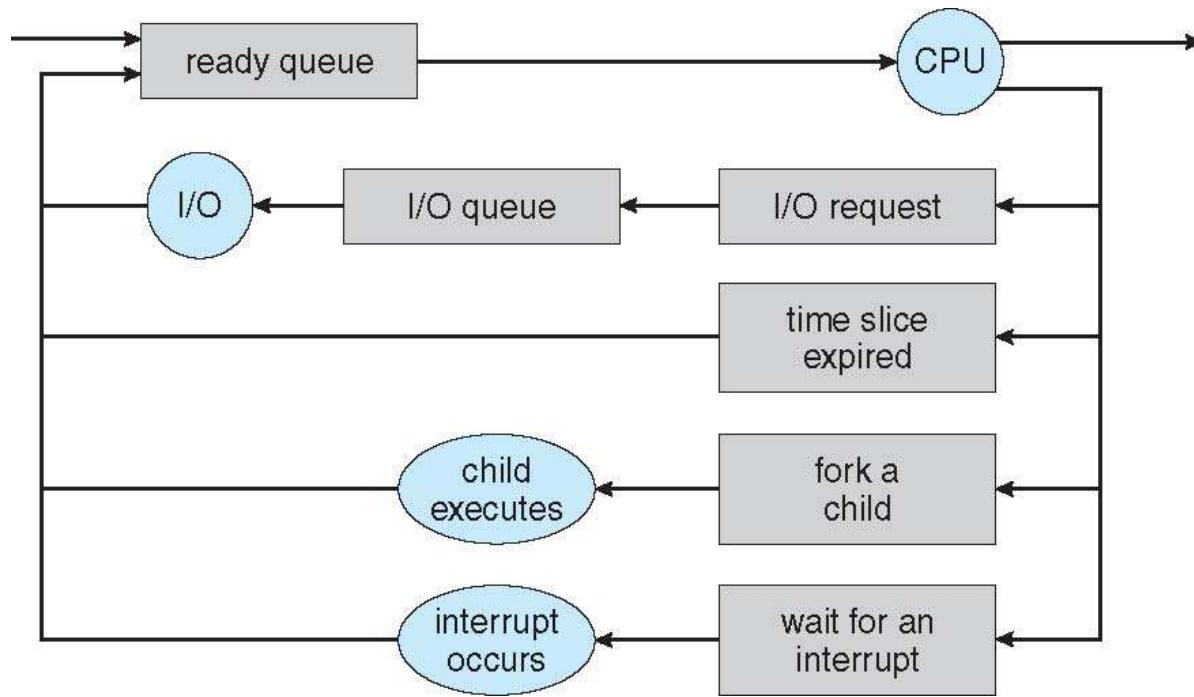
Ready Queue And Various I/O Device Queues





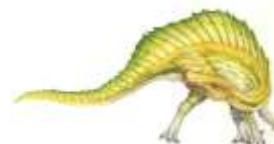
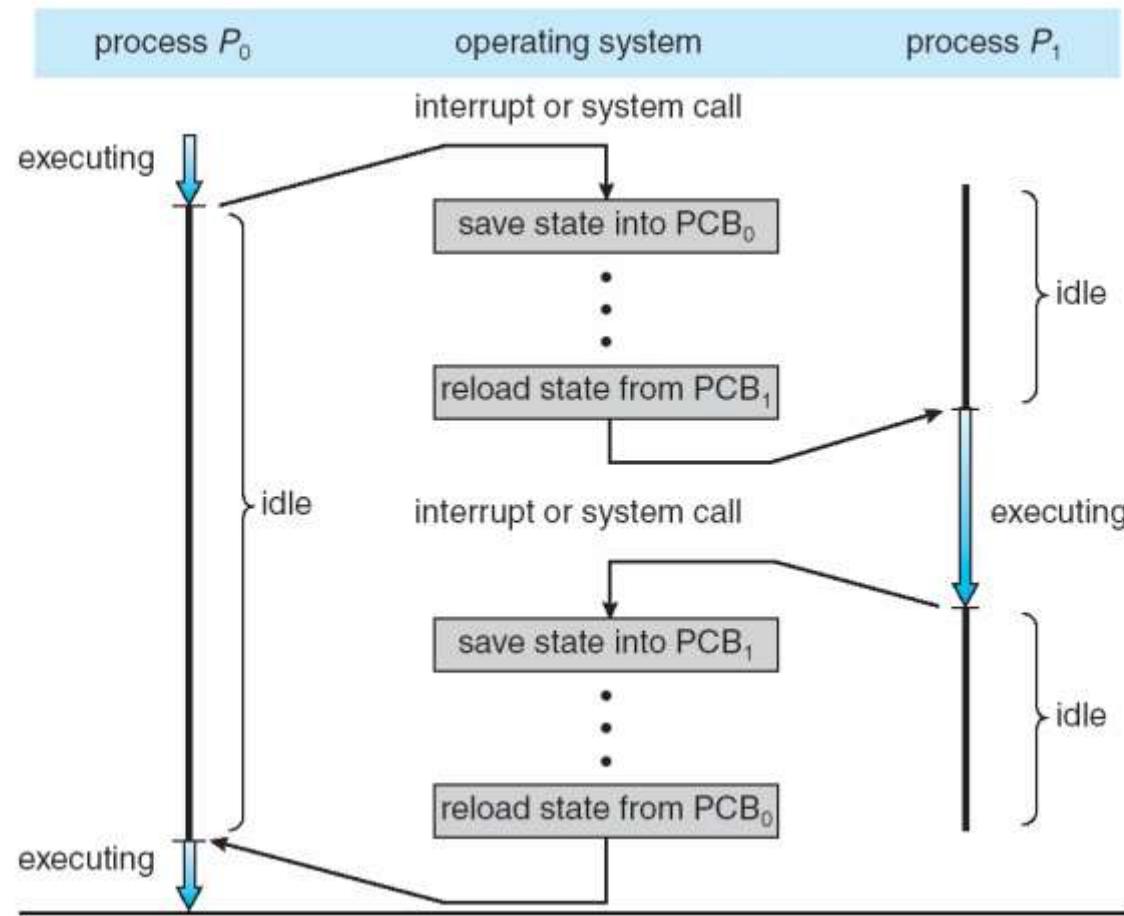
Representation of Process Scheduling

- Queuing diagram represents queues, resources, flows





CPU Switch From Process to Process





Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates a CPU
 - Sometimes the only scheduler in a system
 - Short-term scheduler is invoked frequently (milliseconds) \Rightarrow (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
 - Long-term scheduler is invoked infrequently (seconds, minutes) \Rightarrow (may be slow)
 - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
 - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
 - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***





Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Starting with iOS 4, it provides for a
 - Single **foreground** process – controlled via user interface
 - Multiple **background** processes – in memory, running, but not on the display, and with limits
 - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
 - Background process uses a **service** to perform tasks
 - Service can keep running even if background process is suspended
 - Service has no user interface, small memory use





Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is pure overhead; the system does no useful work while switching
 - The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support
 - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once





Operations on Processes

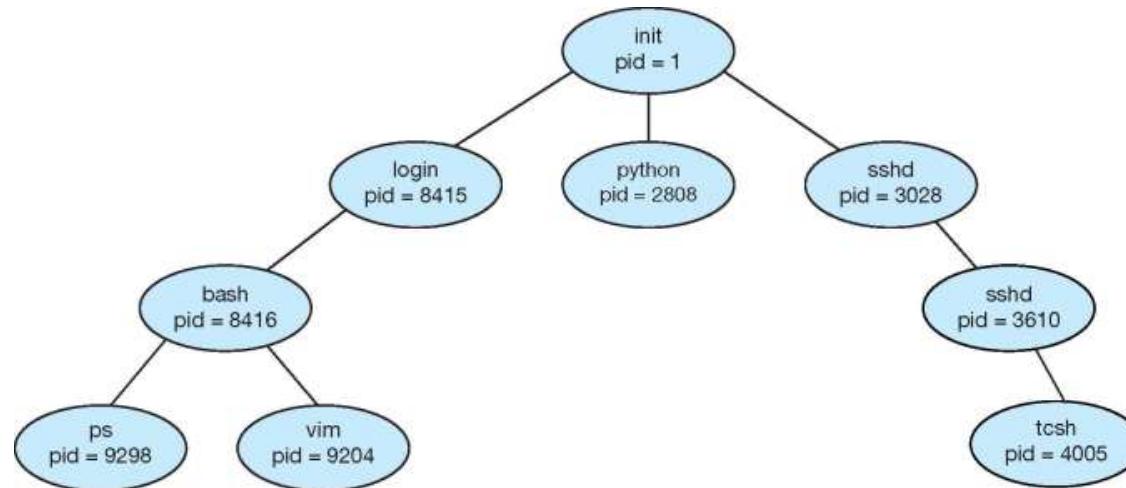
- System must provide mechanisms for:
 - process creation,
 - process termination,
 - and so on as detailed next





Process Creation

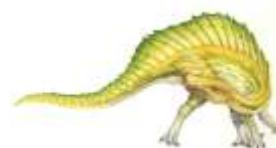
- A **process** may create other processes.
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, a process is identified and managed via a **process identifier (pid)**
- A Tree of Processes in UNIX





Process Creation (Cont.)

- Resource sharing among parents and children options
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution options
 - Parent and children execute concurrently
 - Parent waits until children terminate





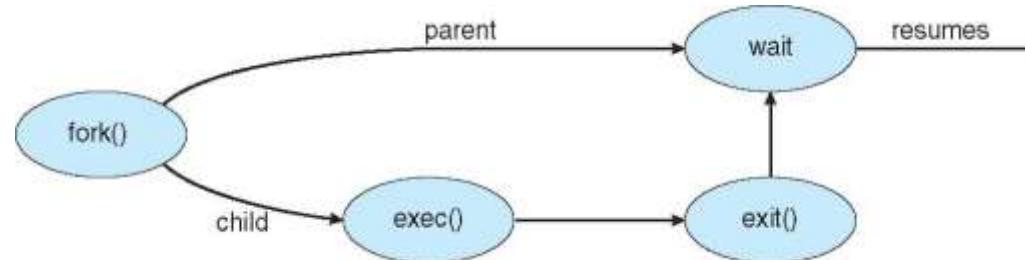
Process Creation (Cont.)

■ Address space

- A child is a duplicate of the parent address space.
- A child loads a program into the address space.

■ UNIX examples

- **fork()** system call creates new process
- **exec()** system call used after a **fork()** replaces the process' memory space with a new program





C program to create a separate process in UNIX

```
int main()
{
pit.t pid;
    /*fork a child process */
    pid = fork();

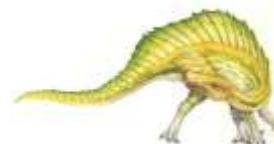
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /*child process */
        execvp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }
    return 0;
}
```





C Program Forking Separate Process

- The C program illustrates how to create a new process UNIX.
- After **fork()** there are two different processes running copies of the same program. The only difference is that the value of pid for the child process is zero, while that for the parent is an integer value greater than zero
- The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files.
- The child process then overlays its address space with the UNIX command “ls” (used to get a directory listing) using the `execlp()` (a version of the `exec()` system call).
- The parent waits for the child process to complete with the `wait()` system call.
- When the child process completes, the parent process resumes from the call to `wait()`, where it completes using the `exit()` system call.



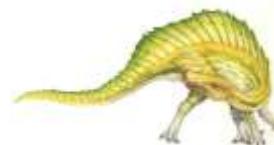


Creating a Separate Process via Windows API

```
int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

/* allocate memory */
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));

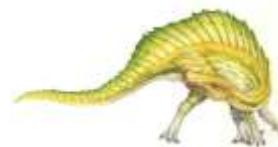
/* create child process */
if (!CreateProcess(NULL, /* use command line */
"C:\\\\WINDOWS\\\\system32\\\\mspaint.exe",/* command */
NULL, /* don't inherit process handle */
NULL, /* don't inherit threat handle */
FALSE, /* disable handle inheritance */
0, /* no creation flags */
NULL, /* use parent's environment block */
NULL, /* use parent's existing directory */
&si,
&pi))
{
```





Creating a Separate Process via Windows API (Cont.)

```
{  
fprintf(stderr, "Create Process Failed");  
return -1;  
}  
/* parent will wait for the child to complete */  
WaitForSingleObject(pi.hProcess, INFINITE);  
printf("Child Complete");  
  
/* close handles */  
CloseHandle(pi.hProcess);  
CloseHandle(pi.hThread);  
}
```





Process Termination

- A process terminates when it finishes executing its final statement and it asks the operating system to delete it by using the **exit()** system call.
 - At that point, the process may return a status value (typically an integer) to its parent process (via the `wait()` system call).
 - All the resources of the process are deallocated by the operating system.
- A parent may terminate the execution of children processes using the **abort()** system call. Some reasons for doing so:
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates





Process Termination (Cont.)

- Some operating systems do not allow a child process to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
 - **cascading termination.** All children, grandchildren, etc. are terminated.
 - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- If no parent waiting (did not invoke `wait()`) process is **zombie**
- If parent terminated without invoking `wait`, process is **orphan**





Interprocess Communication

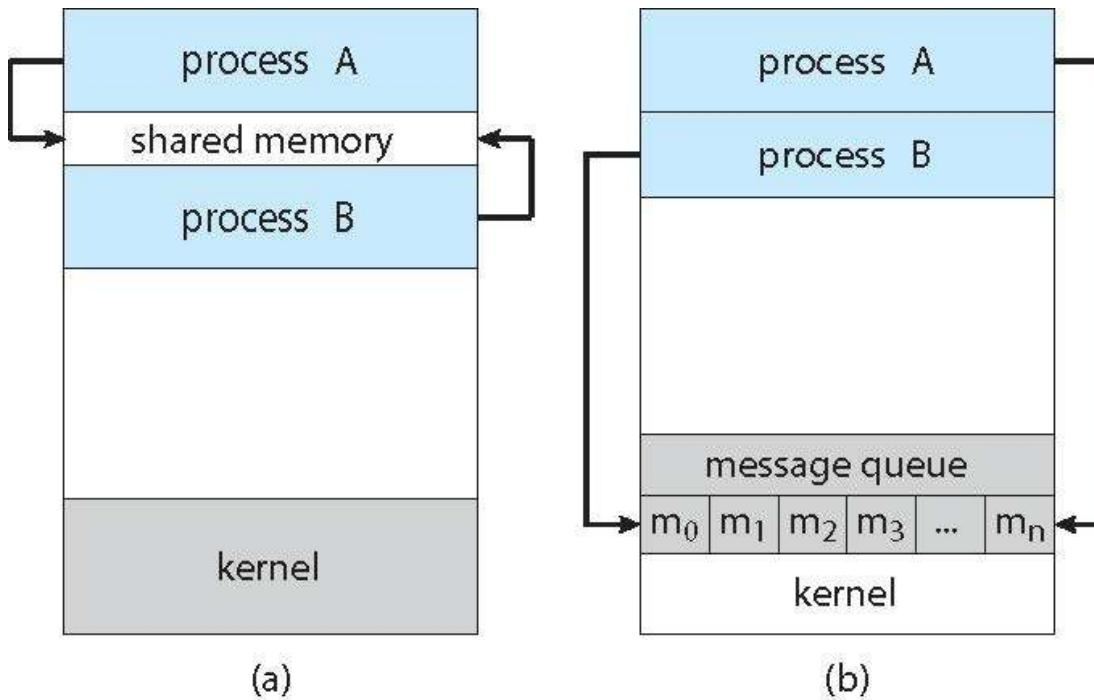
- Processes within a system may be *independent* or *cooperating*
 - Cooperating processes can affect or be affected by other processes, including sharing data
 - Independent processes cannot affect other processes
- Reasons for having cooperating processes:
 - Information sharing
 - Computation speedup (multiple processes running in parallel)
 - Modularity
 - Convenience
- Cooperating processes need **interposes communication (IPC)**





Communications Models

- Two models of IPC
 - Shared memory
 - Message passing





Shared Memory Systems

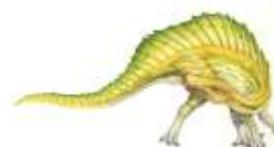
- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the user processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.





Synchronization

- Cooperating processes that access shared data need to synchronize their actions to ensure data consistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem – The producer-Consumer problem
 - Producer process produces information that is consumed by a Consumer process.
 - The information is passed from the Producer to the Consumer via a buffer.
 - Two types of buffers can be used:
 - ▶ **unbounded-buffer** places no practical limit on the size of the buffer
 - ▶ **bounded-buffer** assumes that a fixed buffer size





Bounded-Buffer Solution

- Shared data

```
#define BUFFER_SIZE 10  
  
typedef struct {  
  
    . . .  
  
} item;  
  
  
item buffer[BUFFER_SIZE];  
int in = 0;  
int out = 0;
```

- Solution presented in the next two slides is correct, but only 9 out of 10 buffer elements can be used





Bounded-Buffer – Producer

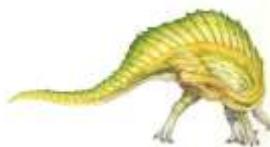
```
item next_produced;  
while (true) {  
    /* produce an item in next produced */  
    while (((in + 1) % BUFFER_SIZE) == out)  
        ; /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
}
```





Bounded Buffer – Consumer

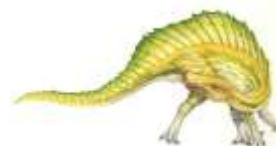
```
item next_consumed;  
  
while (true) {  
    while (in == out)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
  
    /* consume the item in next_consumed */  
}
```





Message Passing Systems

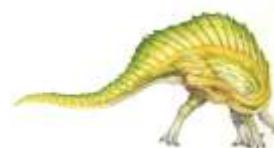
- Mechanism for processes to communicate and to synchronize their actions
 - Without resorting to shared variables
- IPC facility provides two operations:
 - **send(message)**
 - **receive(message)**
- The *message size* is either fixed or variable





Message Passing (Cont.)

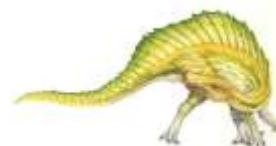
- If processes P and Q wish to communicate, they need to:
 - Establish a ***communication link*** between them
 - Exchange messages via send/receive
- Implementation issues:
 - How are links established?
 - Can a link be associated with more than two processes?
 - How many links can there be between every pair of communicating processes?
 - What is the capacity of a link?
 - Is the size of a message that the link can accommodate fixed or variable?
 - Is a link unidirectional or bi-directional?





Implementation of Communication Link

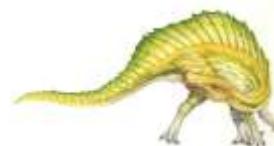
- Physical:
 - Shared memory
 - Hardware bus
 - Network
- Logical:
 - Direct or indirect
 - Synchronous or asynchronous
 - Automatic or explicit buffering





Direct Communication

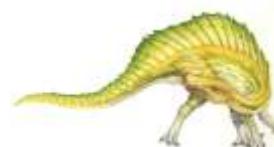
- Processes must name each other explicitly:
 - **send** (P , message) – send a message to process P
 - **receive**(Q , message) – receive a message from process Q
- Properties of communication link
 - Links are established automatically
 - A link is associated with exactly one pair of communicating processes
 - Between each pair there exists exactly one link
 - The link may be unidirectional, but is usually bi-directional





Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
 - Each mailbox has a unique id
 - Processes can communicate only if they share a mailbox
- Operations
 - create a new mailbox (port)
 - send and receive messages through mailbox
 - delete a mailbox
- Primitives are defined as:
 - **send(A, message)** – send a message to mailbox A
 - **receive(A, message)** – receive a message from mailbox A





Indirect Communication (Cont.)

- Properties of communication link
 - Link established only if processes share a common mailbox
 - A link may be associated with many processes
 - Each pair of processes may share several communication links
 - Link may be unidirectional or bi-directional





Indirect Communication Issues

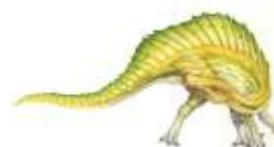
- Mailbox sharing
 - P_1 , P_2 , and P_3 share mailbox A
 - P_1 , sends; P_2 and P_3 receive
 - Who gets the message?
- Solutions
 - Allow a link to be associated with at most two processes
 - Allow only one process at a time to execute a receive operation
 - Allow the system to select arbitrarily the receiver.
Sender is notified who the receiver was.





Blocking and Non-blocking schemes

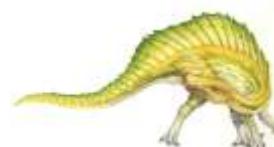
- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
 - **Blocking send** -- the sender is blocked until the message is received
 - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
 - **Non-blocking send** -- the sender sends the message and continue
 - **Non-blocking receive** -- the receiver receives:
 - A valid message, or
 - Null message
- Different combinations possible
 - If both send and receive are blocking, we have a **rendezvous**





Buffering

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.
- Such queues can be implemented in three ways:
 1. Zero capacity – no messages are queued on a link.
Sender must wait for receiver (rendezvous)
 2. Bounded capacity – finite length of n messages
Sender must wait if link full
 3. Unbounded capacity – infinite length
Sender never waits

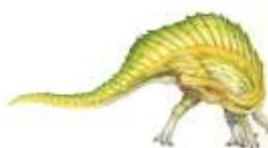




Producer-Consumer with Rendezvous

- Producer-consumer synchronization becomes trivial with rendezvous (blocking send and receive)

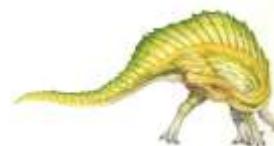
```
message next_produced;  
  
while (true) {  
    /* produce an item in next produced */  
    send(next_produced);  
}  
  
message next_consumed;  
while (true) {  
    receive(next_consumed);  
  
    /* consume the item in next consumed */  
}
```





Example of IPC Systems

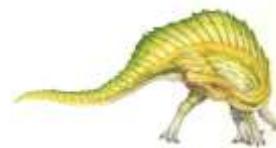
- There are four different IPC systems.
 - POSIX API for shared memory
 - Mach operating system, which uses message passing
 - Windows IPC, which uses shared memory as a mechanism for providing certain types of message passing.
 - Pipes, one of the earliest IPC mechanisms on UNIX systems.





POSIX

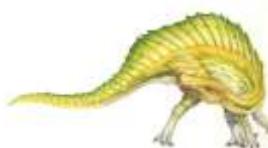
■ POSIX





Mach

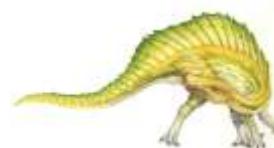
- Mach communication is message based
 - Even system calls are messages
 - Each task gets two mailboxes at creation- Kernel and Notify
 - Only three system calls needed for message transfer
`msg_send()` , `msg_receive()` , `msg_rpc()`
 - Mailboxes needed for communication, created via
`port_allocate()`
 - Send and receive are flexible, for example four options if mailbox full:
 - ▶ Wait indefinitely
 - ▶ Wait at most n milliseconds
 - ▶ Return immediately
 - ▶ Temporarily cache a message





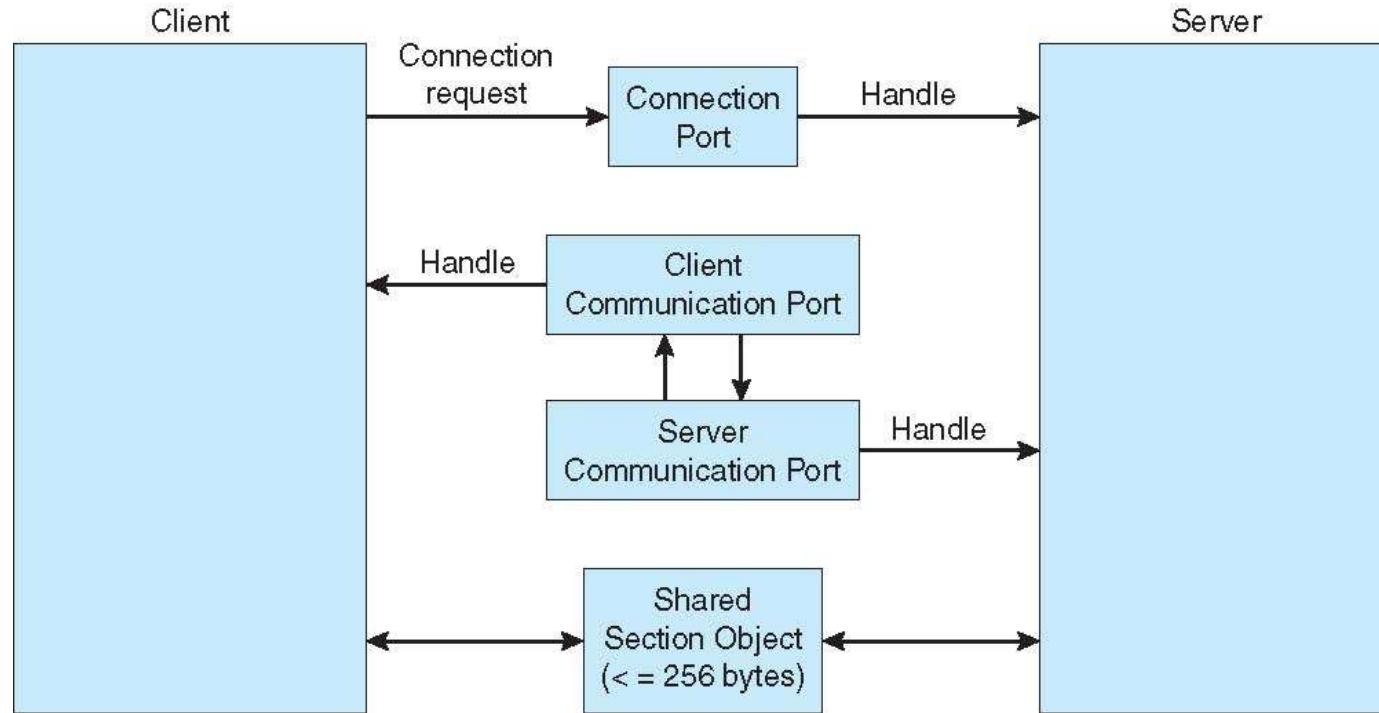
Windows

- Message-passing centric via **advanced local procedure call (LPC)** facility
 - Only works between processes on the same system
 - Uses ports (like mailboxes) to establish and maintain communication channels
 - Communication works as follows:
 - ▶ The client opens a handle (an abstract reference to a resource) to the subsystem's **connection port** object.
 - ▶ The client sends a connection request.
 - ▶ The server creates a private **communication port** and returns the handle to the client.
 - ▶ The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.





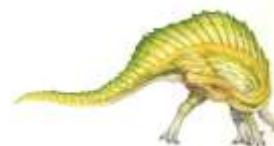
Local Procedure Calls in Windows





Pipes

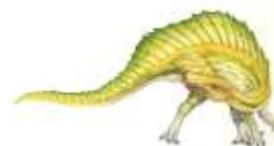
- Acts as a conduit allowing two processes to communicate
- Issues:
 - Is communication unidirectional or bidirectional?
 - In the case of two-way communication, is it half or full-duplex?
 - Must there exist a relationship (i.e., **parent-child**) between the communicating processes?
 - Can the pipes be used over a network?
- Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes – can be accessed without a parent-child relationship.





Ordinary Pipes

- Ordinary Pipes allow two process to communicate in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are unidirectional, allowing only one-way communication.
- If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction.
- Require parent-child relationship between communicating processes





Ordinary Pipes

- On UNIX systems, ordinary pipes are constructed using the function `pipe (int fd[])`
- This function creates a pipe that is accessed through the `int fd[]` file descriptors:
 - `fd[0]` is the read-end of the pipe
 - `fd[1]` is the write-end of the pipe
- UNIX treats a pipe as a special type of file. Thus, pipes can be accessed using ordinary `read()` and `write()` system calls.

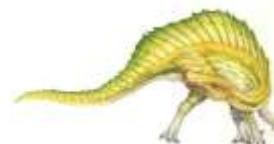
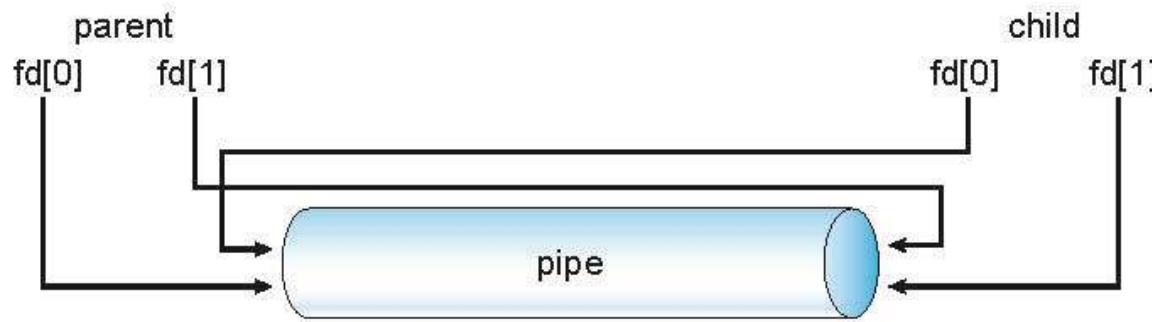
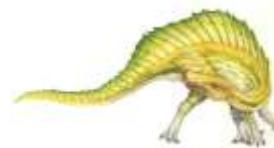
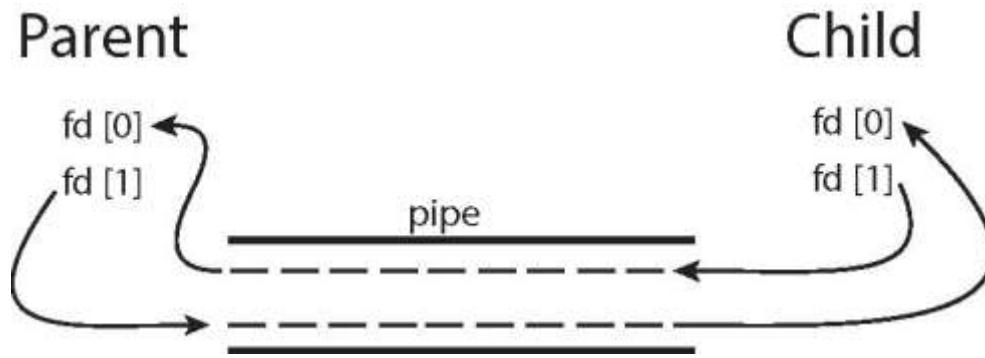




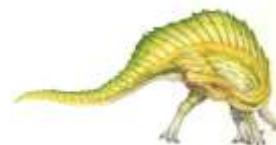
Figure Pipe





Named Pipes

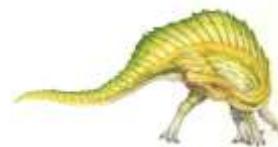
- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems





Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)





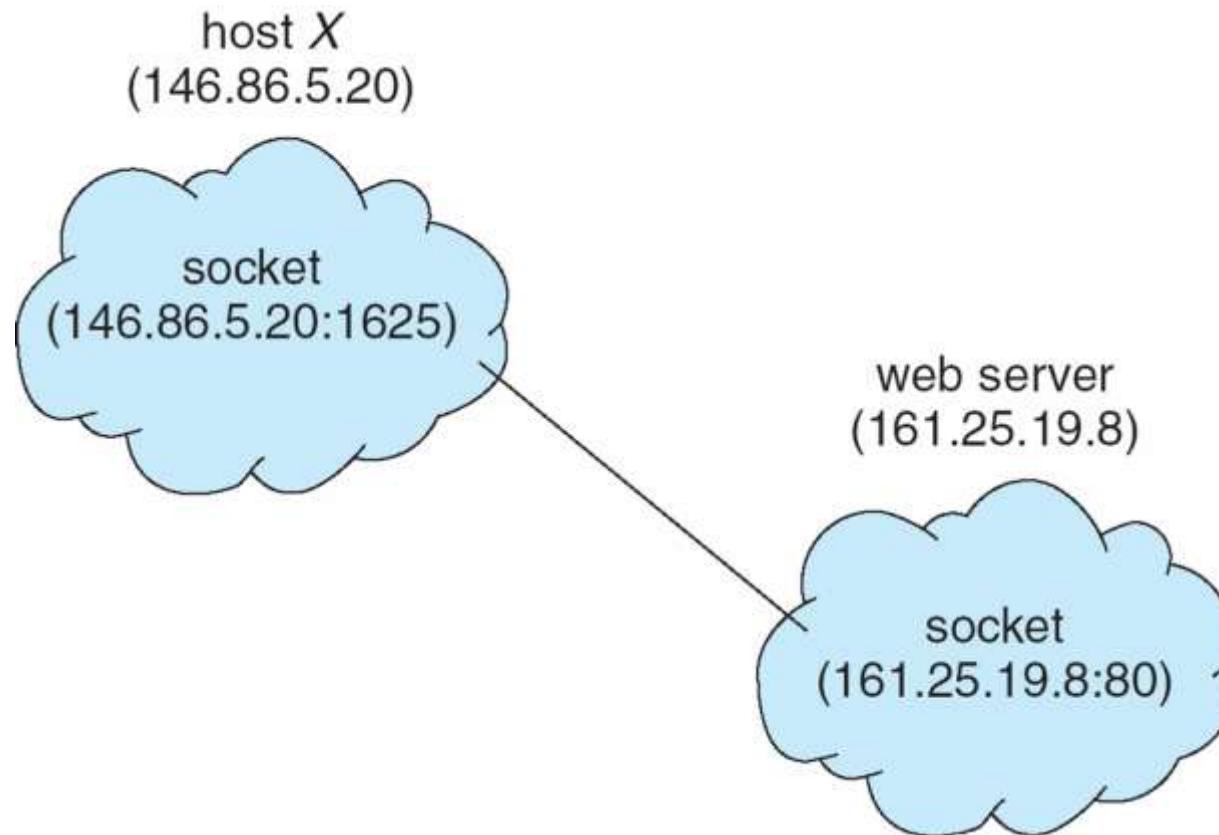
Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are **well known**, used for standard services
- Special IP address 127.0.0.1 (**loopback**) is used to refer to system on which process is running. That is, when a computer refers to address 127.0.0.1, it is referring to itself.





Socket Communication





Sockets in Java

■ Three types of sockets

- **Connection-oriented (TCP)**
- **Connectionless (UDP)**
- **MulticastSocket class**
 - data can be sent to multiple recipients

■ Consider this “Date” server:

```
import java.net.*;
import java.io.*;

public class DateServer
{
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);

            /* now listen for connections */
            while (true) {
                Socket client = sock.accept();

                PrintWriter pout = new
                    PrintWriter(client.getOutputStream(), true);

                /* write the Date to the socket */
                pout.println(new java.util.Date().toString());

                /* close the socket and resume */
                /* listening for connections */
                client.close();
            }
        } catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```





Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
 - Again uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters (marshalling involves packaging the parameters into a form that can be transmitted over a network).
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language (MIDL)**





Remote Procedure Calls (Cont.)

- Data representation handled via **External Data Representation (XDL)** format to account for different architectures.
- Must be dealt with concerns differences in data representation on the client and server machines.
- Consider the representation of 32-bit integers.
 - **Big-endian.** Store the most significant byte first
 - **Little-endian.** Store the least significant byte first.

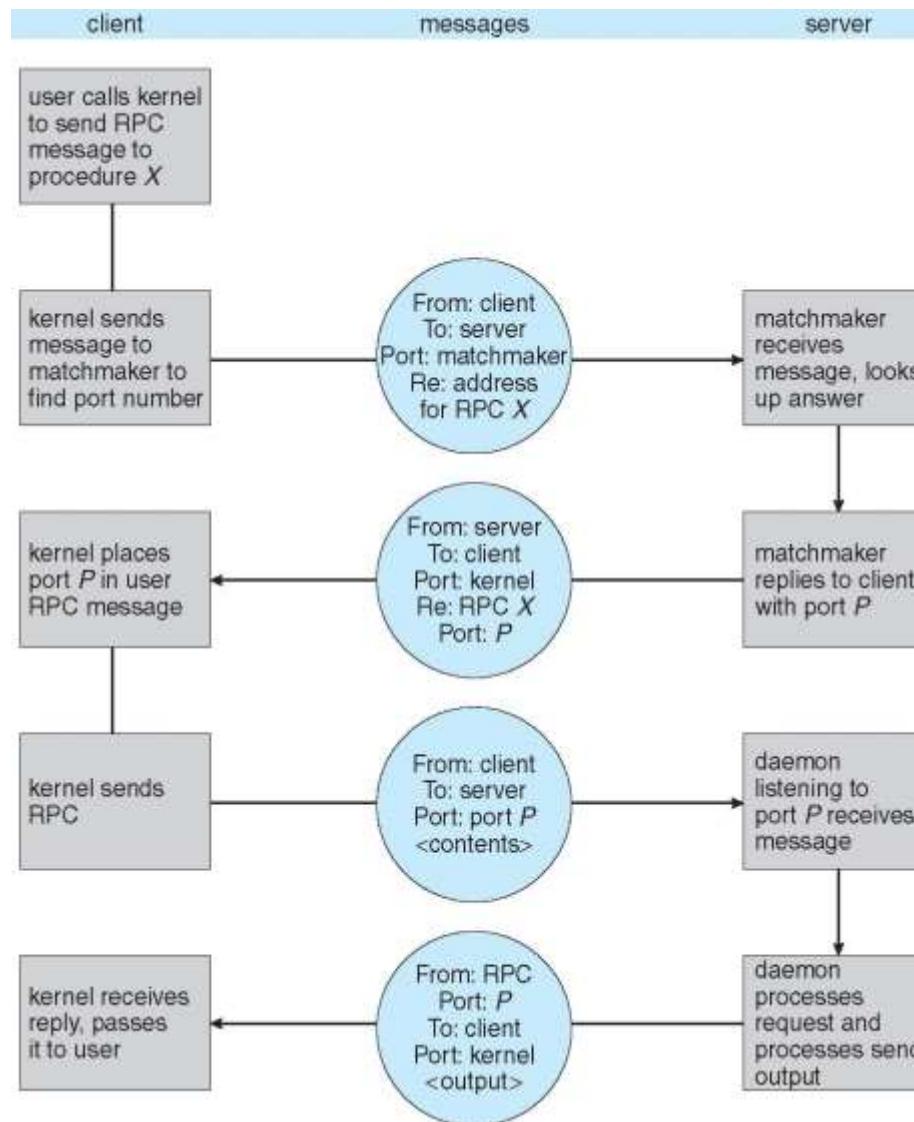
Big-endian is the most common format in data networking . It is also referred to as **network byte order**.

- Remote communication has more failure scenarios than local
 - Messages can be delivered **exactly once** rather than **at most once**
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server



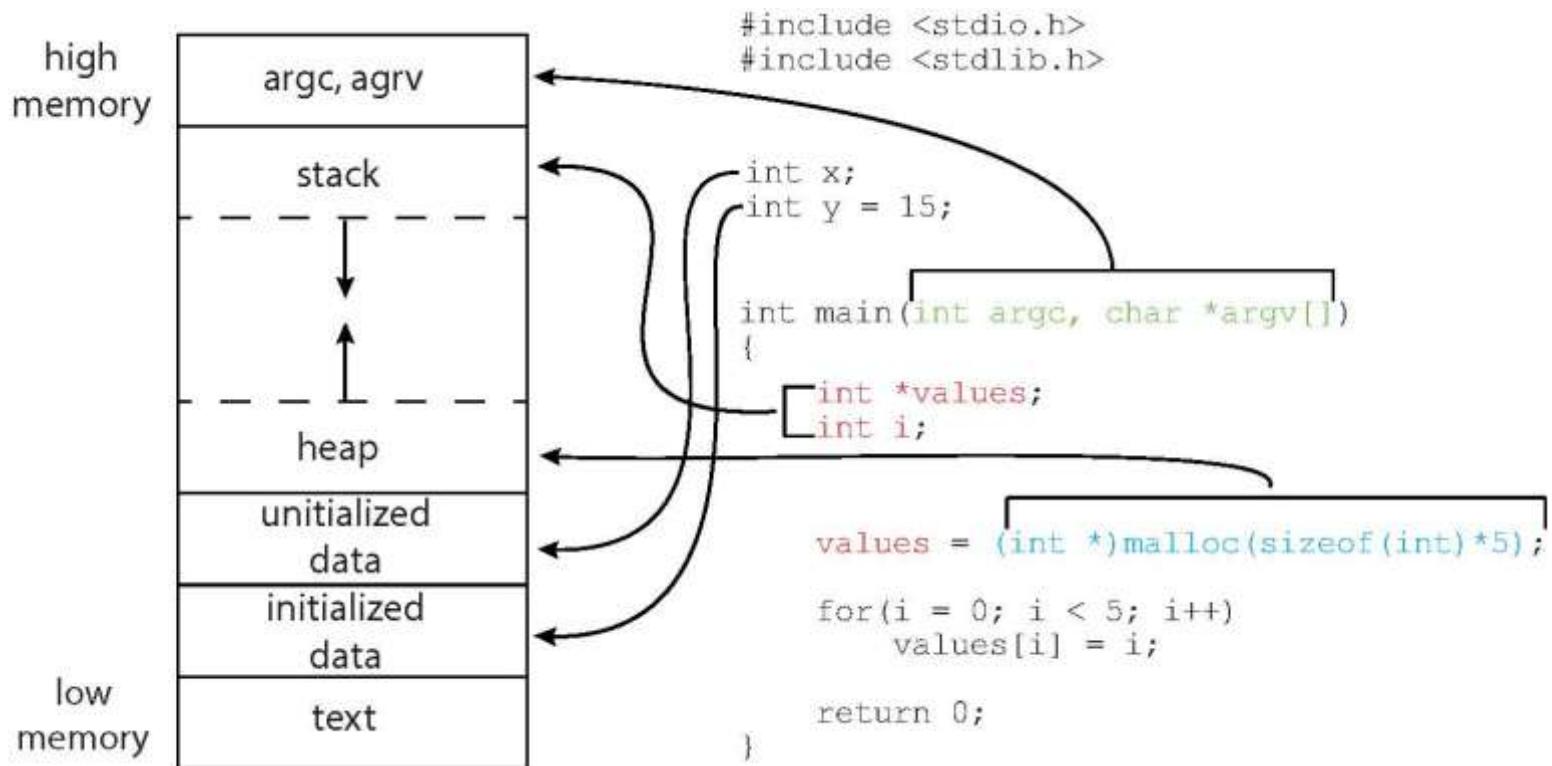


Execution of RPC





Memory Layout



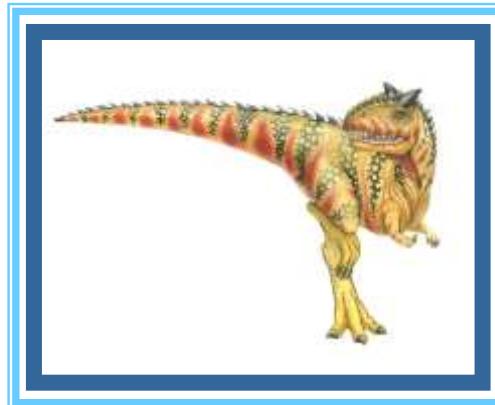


Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
 - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
 - **Browser** process manages user interface, disk and network I/O
 - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
 - ▶ Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
 - **Plug-in** process for each type of plug-in



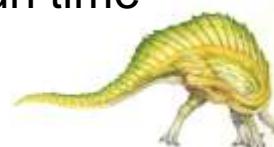
End of Chapter 3





Process Concept

- An operating system executes a variety of programs:
 - Batch system – **jobs**
 - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms **job** and **process** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
 - The program code, also called **text section**
 - Current activity including **program counter**, processor registers
 - **Stack** containing temporary data
 - ▶ Function parameters, return addresses, local variables
 - **Data section** containing global variables
 - **Heap** containing memory dynamically allocated during run time





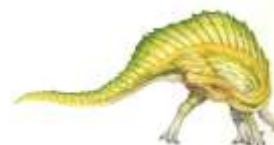
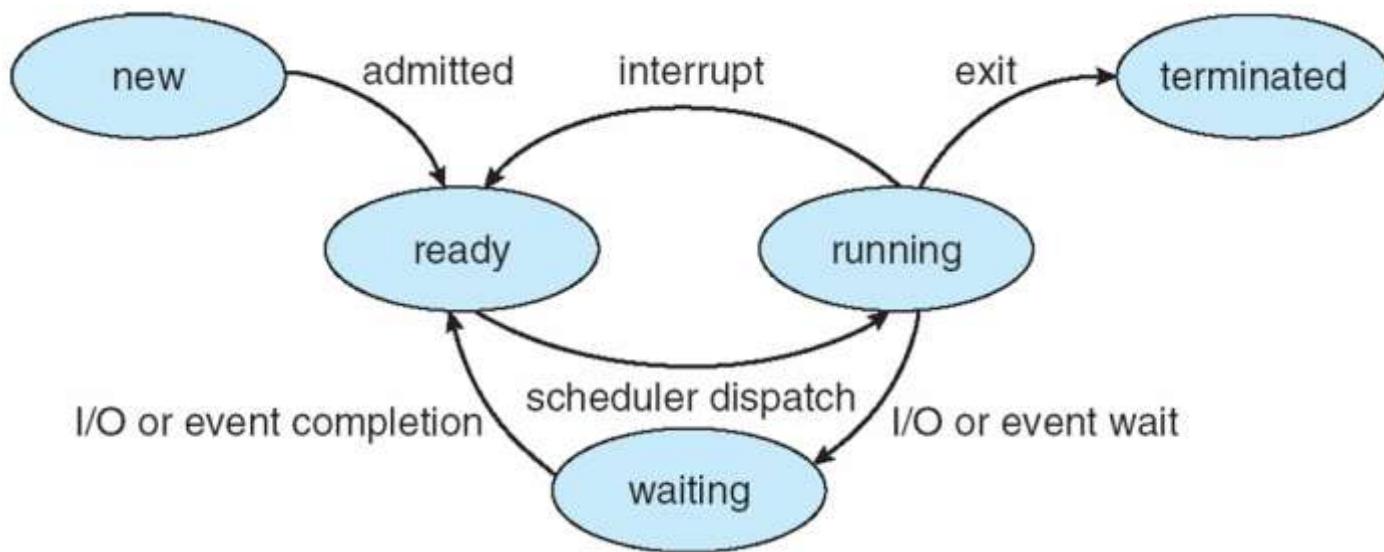
Process Concept

- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
 - The program code, also called **text section**
 - Current activity including **program counter**, processor registers
 - **Stack** containing temporary data
 - ▶ Function parameters, return addresses, local variables
 - **Data section** containing global variables
 - **Heap** containing memory dynamically allocated during run time



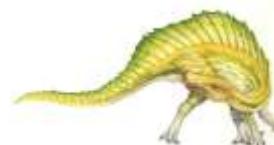
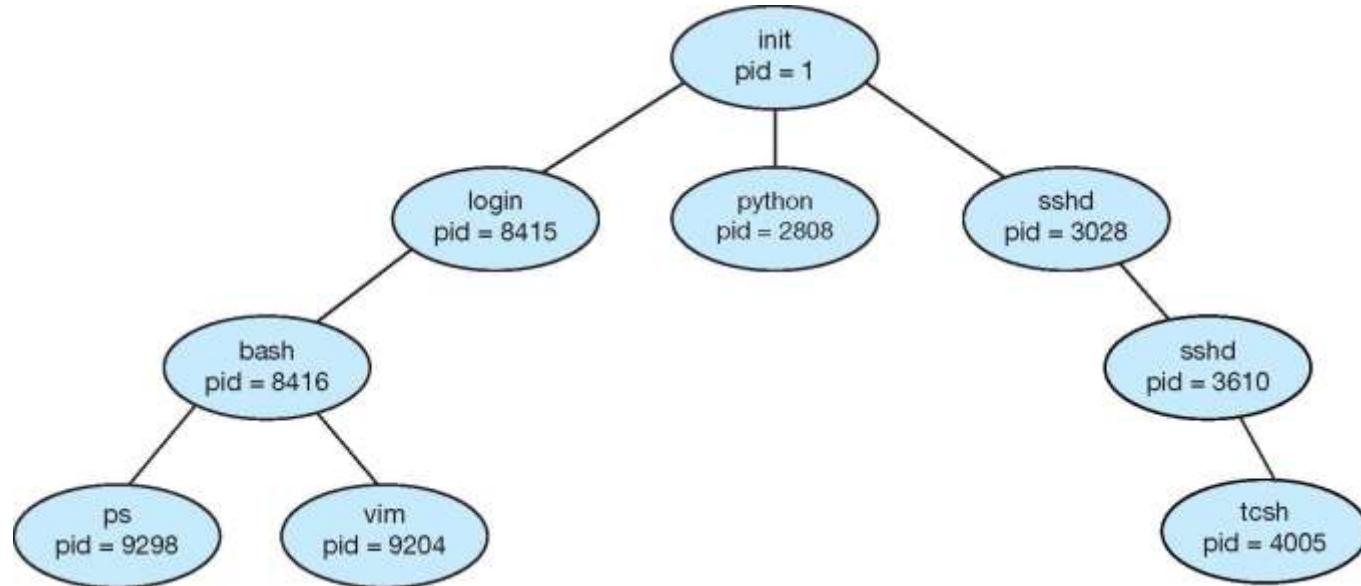


Diagram of Process State





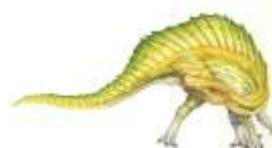
A Tree of Processes in UNIX





Synchronization

- Cooperating process must synchronize their actions
- To illustrate the concept of cooperating processes, we will consider the producer-consumer problem, which is a common paradigm for cooperating processes.
- A producer process produces information that is consumed by a consumer process.
- For example, a compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader.
- The producer-consumer problem also provides a useful metaphor for the client-server paradigm. We generally think of a server as a producer and a client as a consumer. For example, a web server produces (that is, provides) web content such as HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.





Synchronization

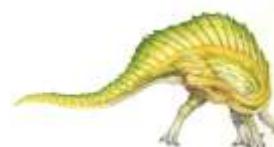
- Cooperating processes Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem – The producer-Consumer problem
 - Producer process produces information that is consumed by a Consumer process.
 - The information is passed from the Producer to the Consumer via a buffer.
 - Two types of buffers can be used:
 - ▶ **unbounded-buffer** places no practical limit on the size of the buffer
 - ▶ **bounded-buffer** assumes that a fixed buffer size





Synchronization

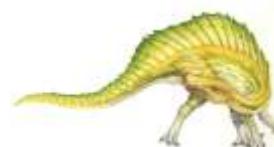
- Cooperating processes Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem – The producer-Consumer problem
 - Producer process produces information that is consumed by a Consumer process.
 - The information is passed from the Producer to the Consumer via a buffer.
 - Two types of buffers can be used:
 - ▶ **unbounded-buffer** places no practical limit on the size of the buffer
 - ▶ **bounded-buffer** assumes that a fixed buffer size





Synchronization

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem – The producer-Consumer problem
 - Producer process produces information that is consumed by a Consumer process.
 - The information is passed from the Producer to the Consumer via a buffer.
 - Two types of buffers can be used:
 - ▶ **unbounded-buffer** places no practical limit on the size of the buffer
 - ▶ **bounded-buffer** assumes that a fixed buffer size





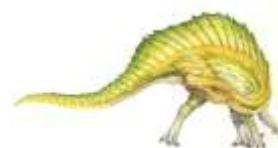
Bounded-Buffer Solution – Shared Memory

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

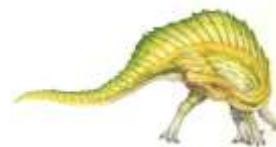
- Solution presented in the next two slides is correct, but only 9 out of 10 buffer elements can be used





Bounded-Buffer – Producer

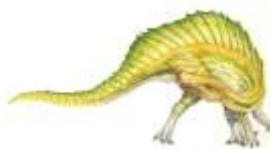
```
item next_produced;  
while (true) {  
    /* produce an item in next produced */  
    while (((in + 1) % BUFFER_SIZE) == out)  
        ; /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
}
```





Bounded Buffer – Consumer

```
item next_consumed;  
  
while (true) {  
    while (in == out)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
  
    /* consume the item in next_consumed */  
}
```



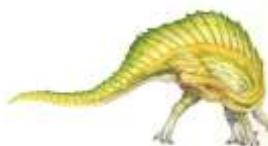


Producer-Consumer with Rendezvous

- Producer-consumer synchronization becomes trivial with rendezvous (blocking send and receive)

```
message next_produced;  
  
while (true) {  
    /* produce an item in next produced */  
    send(next_produced);  
}
```

```
message next_consumed;  
while (true) {  
    receive(next_consumed);  
  
    /* consume the item in next consumed */  
}
```





Examples of IPC Systems - Mach

- Mach communication is message based
 - Even system calls are messages
 - Each task gets two mailboxes at creation- Kernel and Notify
 - Only three system calls needed for message transfer
`msg_send()`, `msg_receive()`, `msg_rpc()`
 - Mailboxes needed for communication, created via
`port_allocate()`
 - Send and receive are flexible, for example four options if mailbox full:
 - ▶ Wait indefinitely
 - ▶ Wait at most n milliseconds
 - ▶ Return immediately
 - ▶ Temporarily cache a message





C program to create a separate process in UNIX

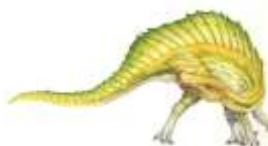
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }
}

return 0;
}
```





Creating a Separate Process via Windows API

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

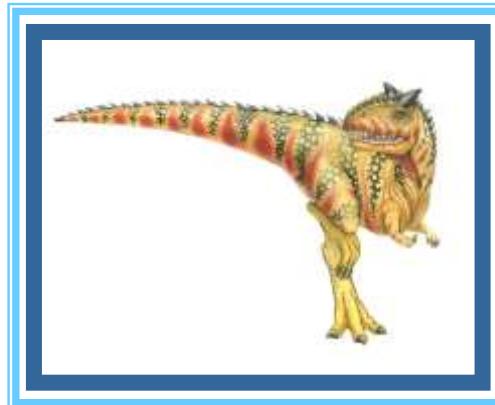
    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
                      "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
                      NULL, /* don't inherit process handle */
                      NULL, /* don't inherit thread handle */
                      FALSE, /* disable handle inheritance */
                      0, /* no creation flags */
                      NULL, /* use parent's environment block */
                      NULL, /* use parent's existing directory */
                      &si,
                      &pi))
    {
        fprintf(stderr, "Create Process Failed");
        return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```



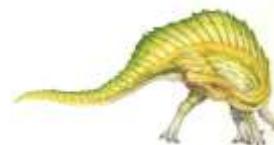
Chapter 4: Threads





Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples





Objectives

- To introduce the notion of a thread—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Windows, and Java thread libraries
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux





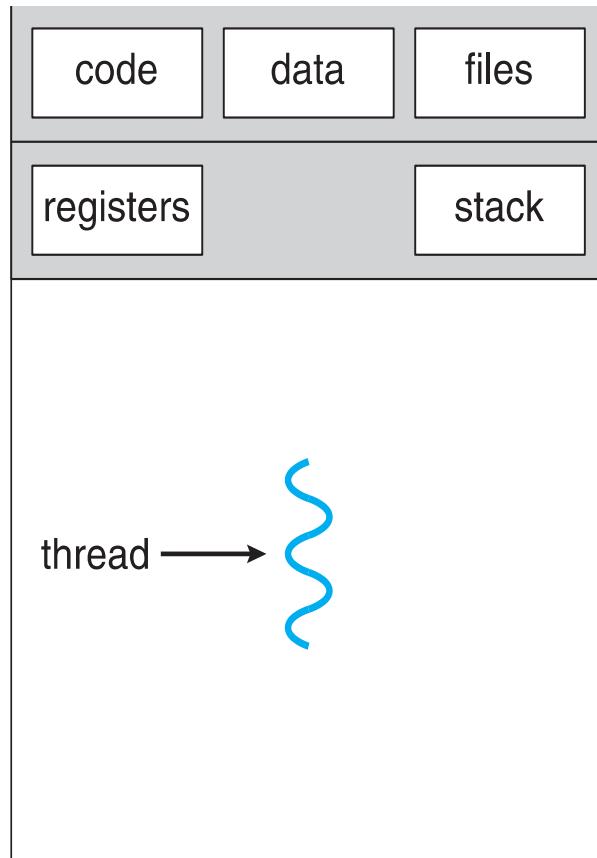
Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks within the application can be implemented by separate threads
 - Update display
 - Fetch data
 - Spell checking
 - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

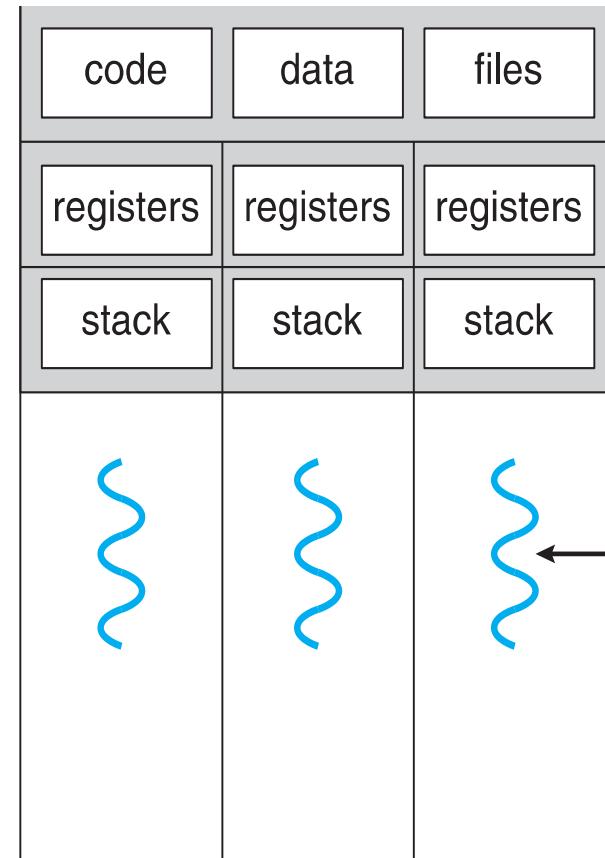




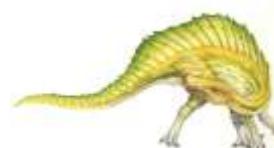
Single and Multithreaded Processes



single-threaded process

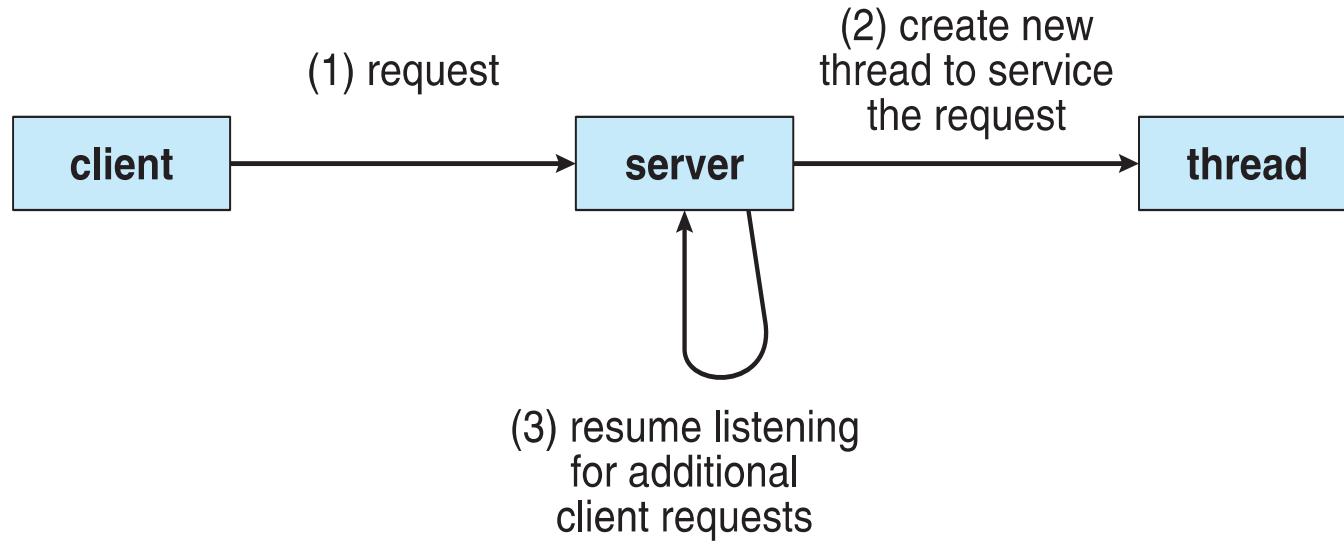


multithreaded process





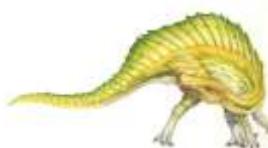
Multithreaded Server Architecture





Benefits

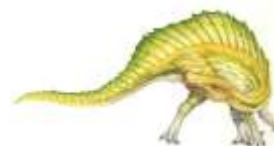
- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures





Multicore Programming

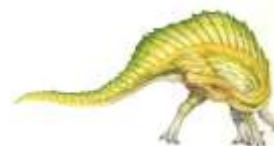
- **Multi-CPU systems.** Multiple CPUs are placed in the computer to provide more computing performance.
- **Multicore systems.** Multiple computing cores are placed on a single processing chip where each core appears as a separate CPU to the operating system
- Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency.
- Consider an application with four threads.
 - On a system with a single computing core, concurrency means that the execution of the threads will be interleaved over time.
 - On a system with multiple cores, however, concurrency means that some threads can run in parallel, because the system can assign a separate thread to each core





Multicore Programming (Cont.)

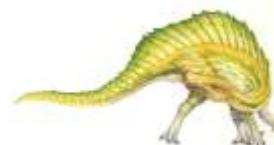
- There is a fine but clear distinction between concurrency and parallelism..
- A concurrent system supports more than one task by allowing all the tasks to make progress.
- In contrast, a system is parallel if it can perform more than one task simultaneously.
- Thus, it is possible to have concurrency without parallelism





Multicore Programming (Cont.)

- Types of parallelism
 - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
 - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As number of threads grows, so does architectural support for threading
 - CPUs have cores as well as ***hardware threads***
 - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core





Multicore Programming

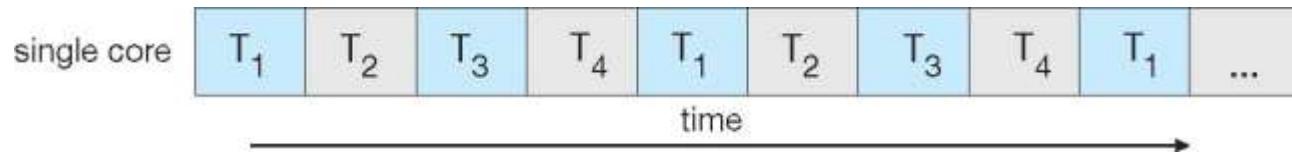
- **Multicore** or **multiprocessor** systems are placing pressure on programmers. Challenges include:
 - **Dividing activities**
 - **Balance**
 - **Data splitting**
 - **Data dependency**
 - **Testing and debugging**
- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
 - Single processor / core, scheduler providing concurrency



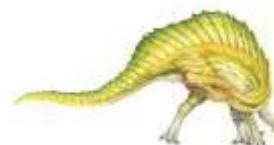
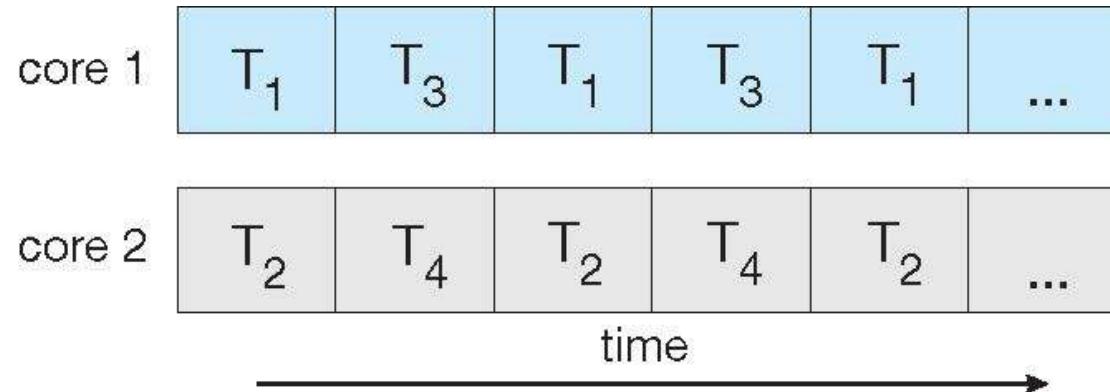


Concurrency vs. Parallelism

- Concurrent execution on single-core system:



- Parallelism on a multi-core system:





Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- N processing cores and S is serial portion

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if an application is 75% parallel and 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As N approaches infinity, speedup approaches $1 / S$
- Serial portion of an application has disproportionate effect on performance gained by adding additional cores**
- But does the law take into account contemporary multicore systems?

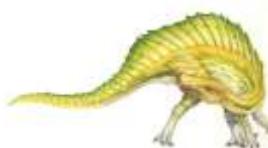
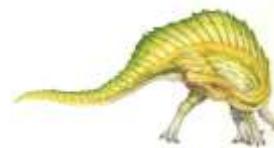
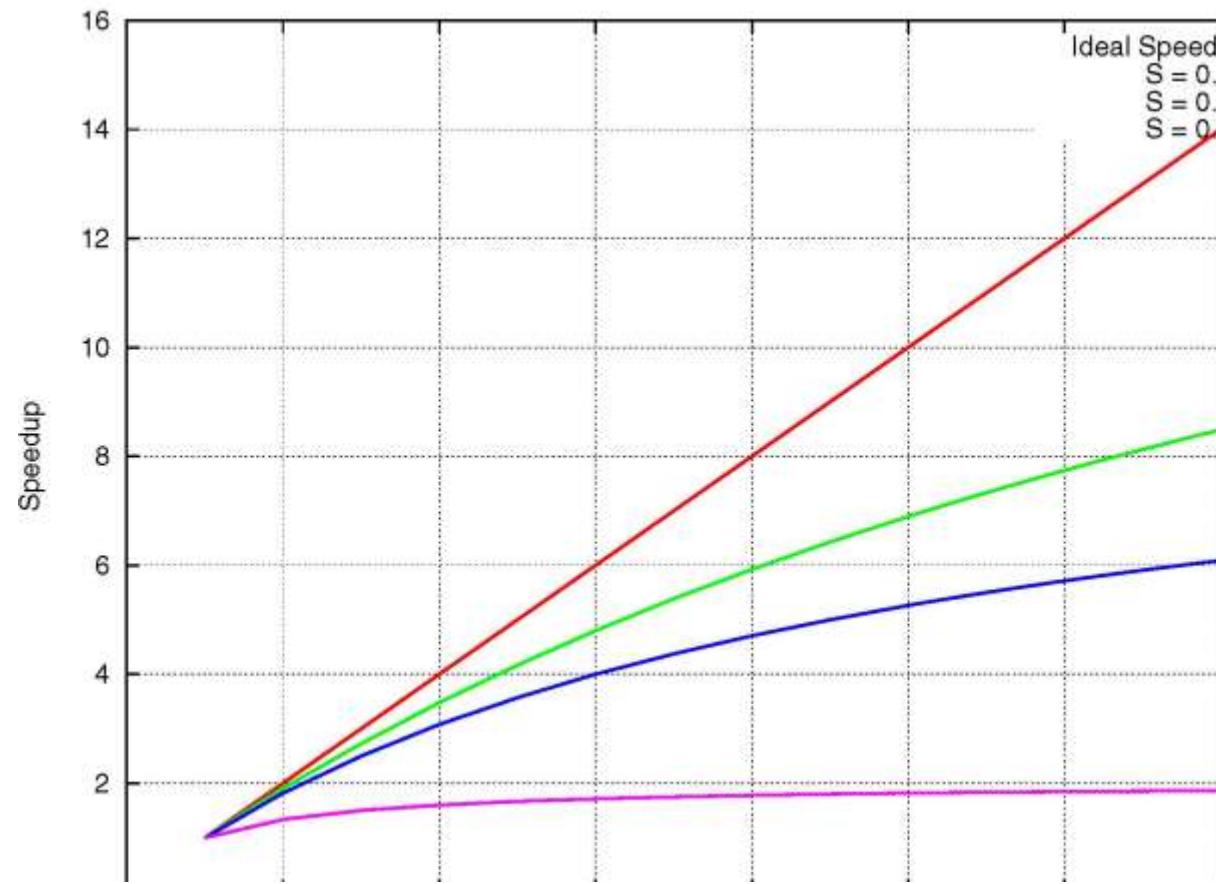




Figure Amdahl





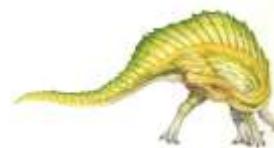
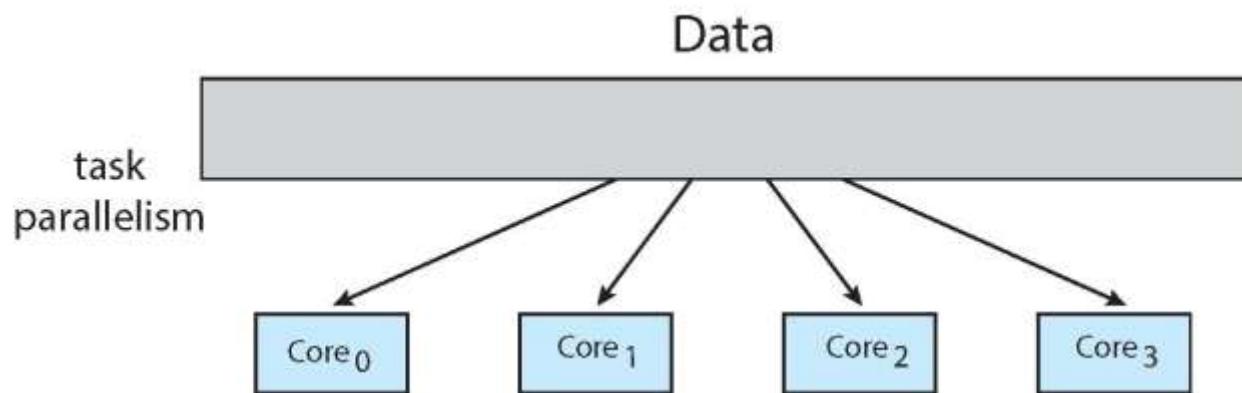
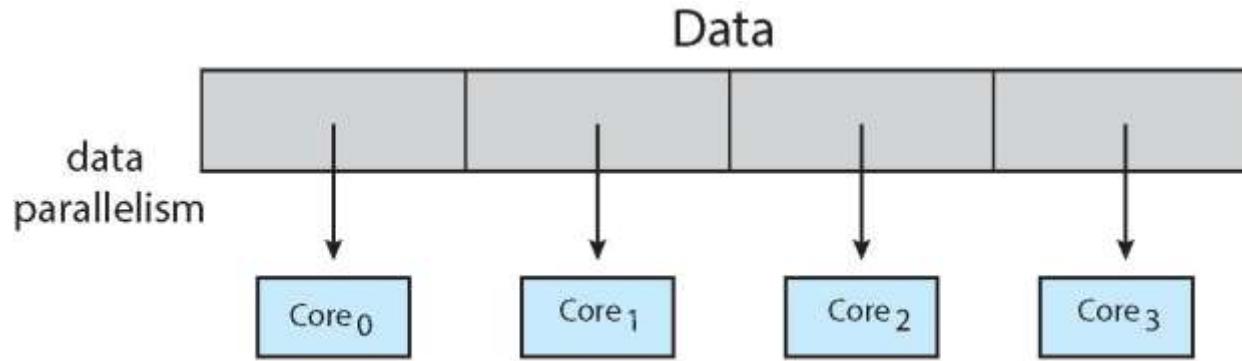
Type of Parallelism

- **Data parallelism.** The focus is on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.
 - Example -- summing the contents of an array of size N . On a single-core system, one thread would sum the elements $0 \dots N-1$. On a dual-core system, however, thread A, running on core 0, could sum the elements $0 \dots N/2$, while thread B, running on core 1, could sum the elements of $N/2 \dots N-1$. The two threads would be running in parallel on separate computing cores.
- **Task parallelism.** involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data.





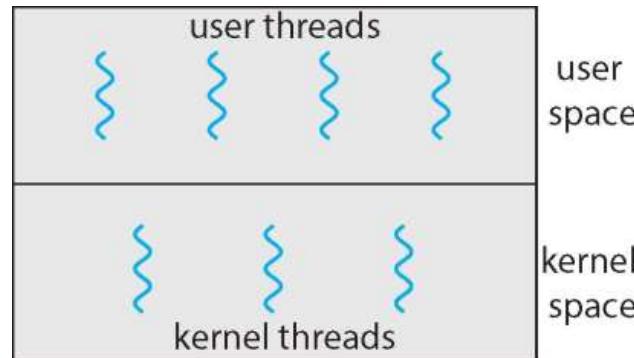
Data and Task Parallelism





User and Kernel Threads

- Support for threads may be provided at two different levels:
 - **User threads** - are supported above the kernel and are managed without kernel support, primarily by user-level threads library.
 - **Kernel threads** - are supported by and managed directly by the operating system.



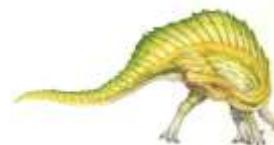
- Virtually all contemporary systems support kernel threads:
 - Windows, Linux, and Mac OS X





Relationship between user and Kernel threads

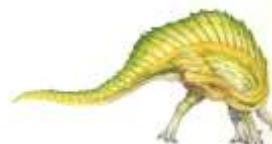
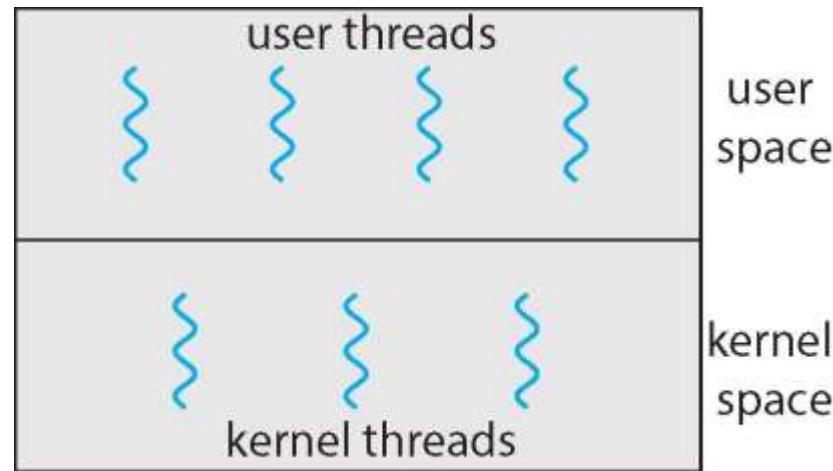
- Three common ways of establishing relationship between user and kernel threads:
 - Many-to-One
 - One-to-One
 - Many-to-Many





One-to-One Model

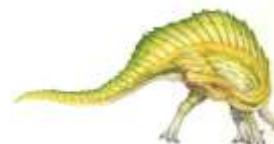
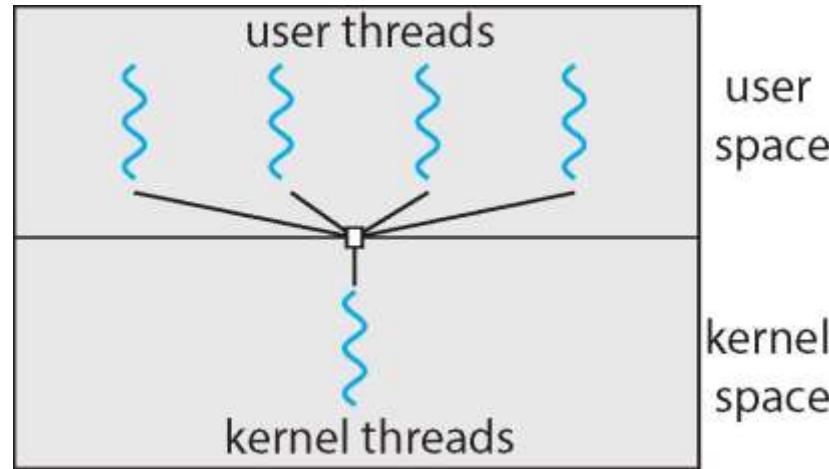
- Each user-level thread maps to a single kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
 - Windows
 - Linux





Many-to-One Model

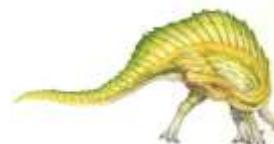
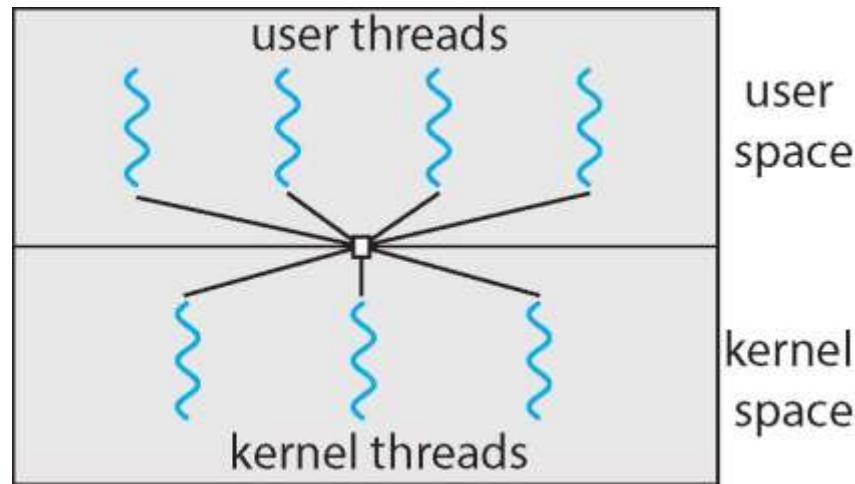
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
 - Solaris Green Threads
 - GNU Portable Threads





Many-to-Many Model

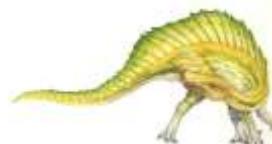
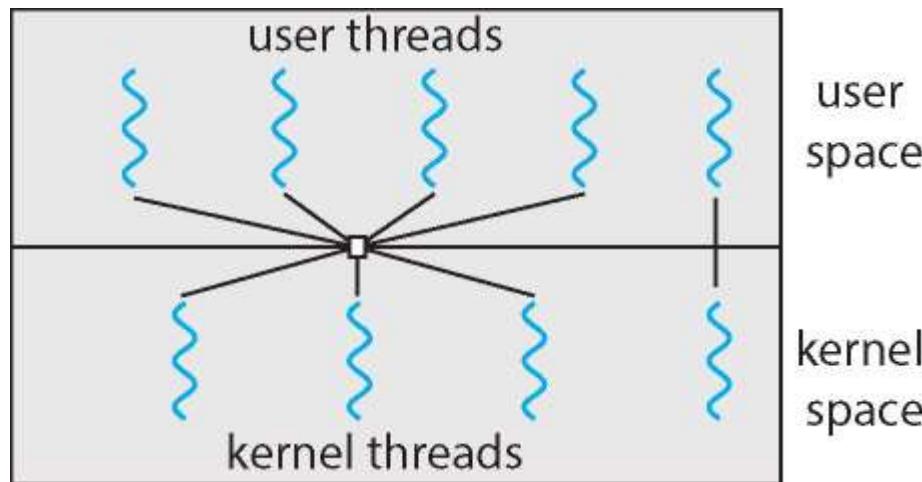
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package





Two-level Model

- Similar to many-to-many, except that it allows a user thread to be **bound** to kernel thread
- Examples
 - IRIX
 - HP-UX
 - Tru64 UNIX
 - Solaris 8 and earlier





Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
 - Library entirely in user space
 - Kernel-level library supported by the OS
- Three primary thread libraries:
 - POSIX **Pthreads**
 - Windows threads
 - Java threads





Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)





Pthreads Example

- Next two slides show a multithreaded C program that calculates the summation of a non-negative integer in a separate thread.
- In a Pthreads program, separate threads begin execution in a specified function. In the program, this is the `runner()` function.
- When this program starts, a single thread of control begins in `main()`. After some initialization, `main()` creates a second thread that begins control in the `runner()` function. Both threads share the global data `sum`.





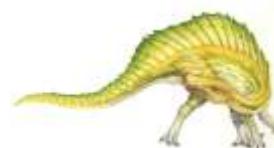
Pthreads Example

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
        return -1;
    }
}
```





Pthreads Example (Cont.)

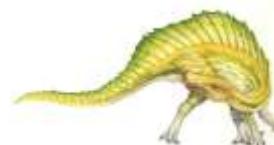
```
/* get the default attributes */
pthread_attr_init(&attr);
/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);
/* wait for the thread to exit */
pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```





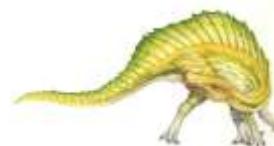
Pthreads Code for Joining Ten Threads

- The summation program in the previous slides creates a single thread.
- With multicore systems, writing programs containing several threads is common.
- Example a Pthreads program, for joining the threads:

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```





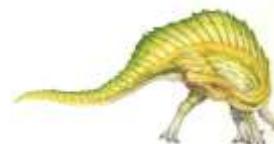
Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
        Sum += i;
    return 0;
}

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

    if (argc != 2) {
        fprintf(stderr,"An integer parameter is required\n");
        return -1;
    }
    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr,"An integer >= 0 is required\n");
        return -1;
    }
}
```





Windows Multithreaded C Program (Cont.)

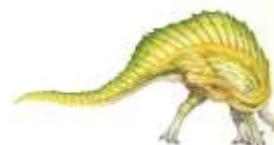
```
/* create the thread */
ThreadHandle = CreateThread(
    NULL, /* default security attributes */
    0, /* default stack size */
    Summation, /* thread function */
    &Param, /* parameter to thread function */
    0, /* default creation flags */
    &ThreadId); /* returns the thread identifier */

if (ThreadHandle != NULL) {
    /* now wait for the thread to finish */
    WaitForSingleObject(ThreadHandle, INFINITE);

    /* close the thread handle */
    CloseHandle(ThreadHandle);

    printf("sum = %d\n", Sum);
}

}
```





Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Two techniques for creating threads in a Java program.
 - One approach is to create a new class that is derived from the Thread class and to override its run() method.
 - An alternative is to define a class that implements the Runnable} interface, as shown below

```
public interface Runnable
{
    public abstract void run();
}
```

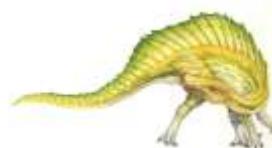
- When a class implements Runnable, it must define a run() method. The code implementing the run() method is what runs as a separate thread.





Java Multithreaded Program

- The Java program in the next slide shows a multithreaded program that determines the summation of a non-negative integer.
- The Summation class implements the Runnable interface.
- Thread creation is performed by creating an object instance of the Thread} class and passing the constructor a Runnable object.





Java Multithreaded Program (Cont.)

```
class Sum
{
    private int sum;

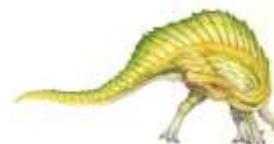
    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
}

class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue) {
        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run() {
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum += i;
        sumValue.setSum(sum);
    }
}
```





Java Multithreaded Program (Cont.)

```
public class Driver
{
    public static void main(String[] args) {
        if (args.length > 0) {
            if (Integer.parseInt(args[0]) < 0)
                System.err.println(args[0] + " must be >= 0.");
            else {
                Sum sumObject = new Sum();
                int upper = Integer.parseInt(args[0]);
                Thread thrd = new Thread(new Summation(upper, sumObject));
                thrd.start();
                try {
                    thrd.join();
                    System.out.println
                        ("The sum of "+upper+" is "+sumObject.getSum());
                } catch (InterruptedException ie) { }
            }
        }
        else
            System.err.println("Usage: Summation <integer value>"); }
    }
}
```





Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods explored
 - Thread Pools
 - Fork Join
 - OpenMP
 - Intel Thread Building Blocks
 - Grand Central Dispatch
- Other methods include Microsoft Threading Building Blocks (TBB), `java.util.concurrent` package

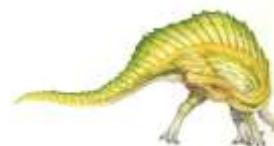




Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
 - Usually slightly faster to service a request with an existing thread than create a new thread
 - Allows the number of threads in the application(s) to be bound to the size of the pool
 - Separating task to be performed from mechanics of creating task allows different strategies for running task
 - ▶ That is, tasks could be scheduled to run periodically
- Windows API supports thread pools:

```
DWORD WINAPI PoolFunction(VOID Param) {  
    /*  
     * this function runs as a separate thread.  
     */  
}
```





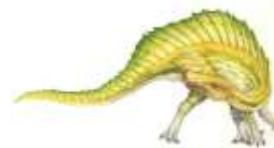
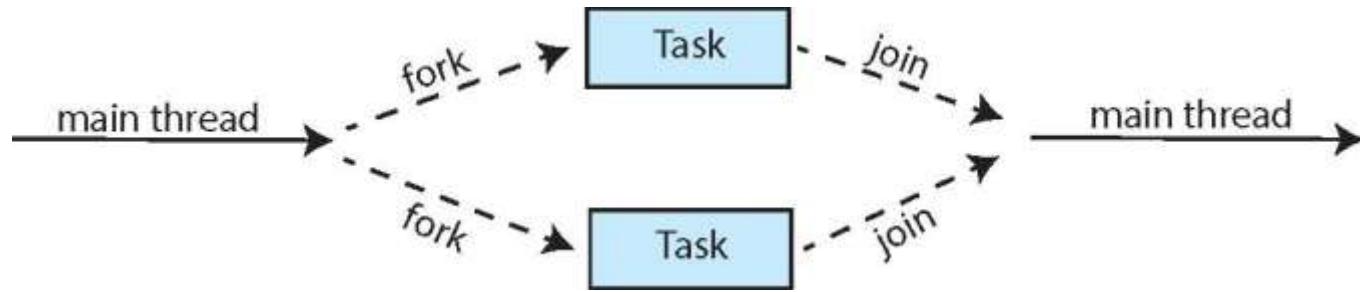
Creating a Thread Pool in Java

```
import java.util.concurrent.*;  
  
public class ThreadPoolExample  
{  
    public static void main(String[] args) {  
        int numTasks = Integer.parseInt(args[0].trim());  
  
        /* Create the thread pool */  
        ExecutorService pool =Executors.newCachedThreadPool();  
  
        /* Run each task using a thread in the pool */  
        for (int i = 0; i < numTasks; i++)  
            pool.execute (new Task());  
  
        /* Shut down the pool once all threads have completed */  
        pool.shutdown();  
    }  
}
```



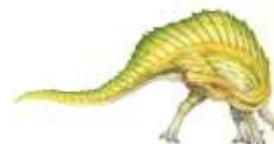
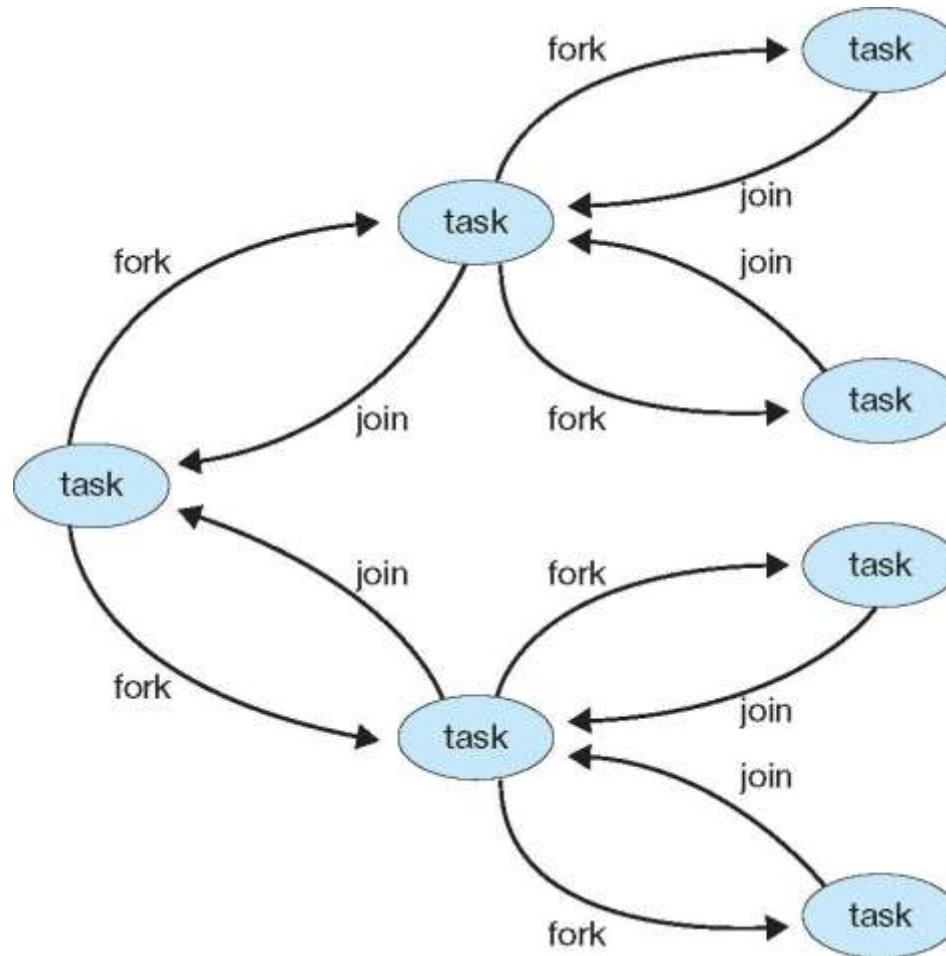


Fork-Join Parallelism





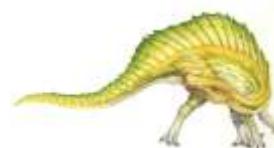
Fork-Join in JAVA





Fork-Join Calculation using the Java API

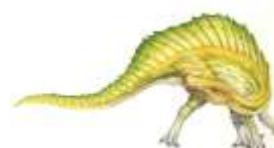
```
import java.util.concurrent.*;
public class SumTask extends RecursiveTask<Integer>
{
    static final int THRESHOLD = 1000;
    private int begin;
    private int end;
    private int[] array;
    public SumTask(int begin, int end, int[] array) {
        this.begin = begin;
        this.end = end;
        this.array = array;
    }
}
```





Fork-Join Calculation using the Java API (Cont.)

```
protected Integer compute() {
    if (end - begin < THRESHOLD) {
        int sum = 0;
        for (int i = begin; i <= end; i++)
            sum += array[i];
        return sum;
    }
    else {
        int mid = begin + (end - begin) / 2;
        int mid = (begin + end) / 2;
        SumTask leftTask = new SumTask(begin, mid, array);
        SumTask rightTask = new SumTask(mid + 1, end, array);
        leftTask.fork();
        rightTask.fork();
        return rightTask.join() + leftTask.join();
    }
}
```





OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

```
#pragma omp parallel
```

Create as many threads as there are cores

```
#pragma omp parallel for
for(i=0;i<N;i++) {
    c[i] = a[i] + b[i];
}
```

Run for loop in parallel

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    /* sequential code */

    #pragma omp parallel
    {
        printf("I am a parallel region.");
    }

    /* sequential code */

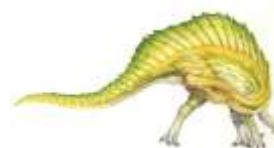
    return 0;
}
```





Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in “^{ }” - ^{ printf("I am a block") ; }
- Blocks placed in dispatch queue
 - Assigned to available thread in thread pool when removed from queue





Grand Central Dispatch

- Two types of dispatch queues:
 - serial – blocks removed in FIFO order, queue is per process, called **main queue**
 - ▶ Programmers can create additional serial queues within program
 - concurrent – removed in FIFO order but several may be removed at a time
 - ▶ Three system wide queues with priorities low, default, high

```
dispatch_queue_t queue = dispatch_get_global_queue
    (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

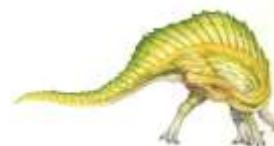
dispatch_async(queue, ^{
    printf("I am a block.");
});
```





Threading Issues

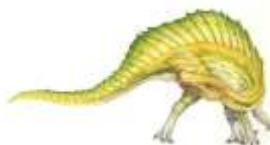
- Semantics of **fork()** and **exec()** system calls
- Signal handling
 - Synchronous and asynchronous
- Thread cancellation of target thread
 - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations





Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?
 - Some UNIX systems have two versions of fork
- **exec()** usually works as normal – replace the running process including all threads





Signal Handling

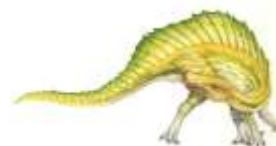
- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
 1. Signal is generated by particular event
 2. Signal is delivered to a process
 3. Signal is handled by one of two signal handlers:
 1. default
 2. user-defined
- Every signal has **default handler** that the kernel runs when handling signal
 - **User-defined signal handler** can override default
 - For single-threaded, signal delivered to process





Signal Handling (Cont.)

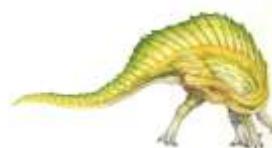
- Where should a signal be delivered for multi-threaded?
 - Deliver the signal to the thread to which the signal applies
 - Deliver the signal to every thread in the process
 - Deliver the signal to certain threads in the process
 - Assign a specific thread to receive all signals for the process





Thread Cancellation

- Terminating a thread before it has finished
- The thread to be canceled is referred to as **target thread**
- Cancelation of a target thread may be handled using two general approaches:
 - **Asynchronous cancellation** terminates the target thread immediately
 - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- The difficulty with cancellation occurs in situations where:
 - Resources have been allocated to a canceled thread.
 - A thread is canceled while in the midst of updating data it is sharing with other threads.





Thread Cancellation

- The operating system usually will reclaim system resources from a canceled thread but will not reclaim all resources.
- Canceling a thread asynchronously does not necessarily free a system-wide resource that is needed by others.
- Deferred cancellation does not suffer from this problem:
 - One thread indicates that a target thread is to be canceled,
 - The cancellation occurs only after the target thread has checked a flag to determine whether or not it should be canceled.
 - The thread can perform this check at a point at which it can be canceled safely.





Pthread Cancellation

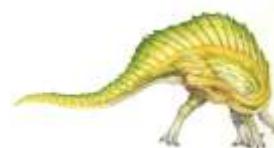
- Pthread cancellation is initiated using the function:

`pthread\cancel()`

The identifier of the target Pthread is passed as a parameter to the function.

- Pthread code to create and cancel a thread:

```
pthread_t tid;  
  
/* create the thread */  
pthread_create(&tid, 0, worker, NULL);  
  
.  
.  
.  
  
/* cancel the thread */  
pthread_cancel(tid);
```



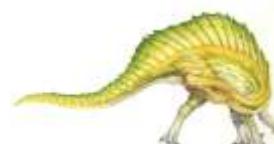


Pthread Cancellation (Cont.)

- Invoking `pthread_cancel()` indicates only a request to cancel the target thread.
- The actual cancellation depends on how the target thread is set up to handle the request.
- Pthread supports three cancellation modes. Each mode is defined as a state and a type. A thread may set its cancellation state and type using an API.

Mode	State	Type
Off	Disabled	-
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it





Thread Cancellation (Cont.)

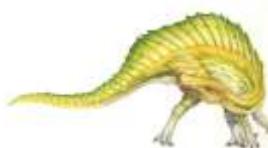
- The default cancellation type is the deferred cancellation
 - Cancellation only occurs when thread reaches **cancellation point**
 - One way for establishing a cancellation point is to invoke the **`pthread_testcancel()`** function.
 - If a cancellation request is found to be pending, a function known as a **cleanup handler** is invoked. This function allows any resources a thread may have acquired to be released before the thread is terminated.
- On Linux systems, thread cancellation is handled through signals





Thread-Local Storage

- **Thread-local storage (TLS)** allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
 - Local variables visible only during single function invocation
 - TLS visible across function invocations
- Similar to **static** data
 - TLS is unique to each thread





Scheduler Activations

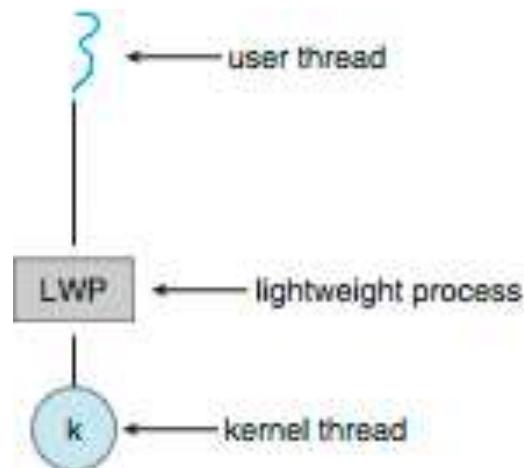
- Both many-to-many and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application.
- Typically uses an intermediate data structure between user and kernel threads
- This data structure is known as a – **lightweight process (LWP)**
 - Appears to be a virtual processor on which process can schedule user thread to run
 - Each LWP is attached to kernel thread.
 - The kernel threads are the ones that the operating system schedules to run on physical processors.





Scheduler Activations (Cont.)

- Lightweight process schema



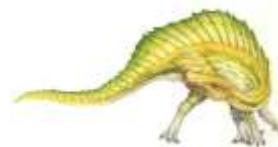
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads





Operating System Examples

- Windows Threads
- Linux Threads





Windows Threads

- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
- Implements the one-to-one mapping, kernel-level
- Each thread contains
 - A thread id
 - Register set representing state of processor
 - Separate user and kernel stacks for when thread runs in user mode or kernel mode
 - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread





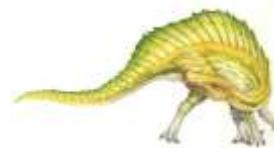
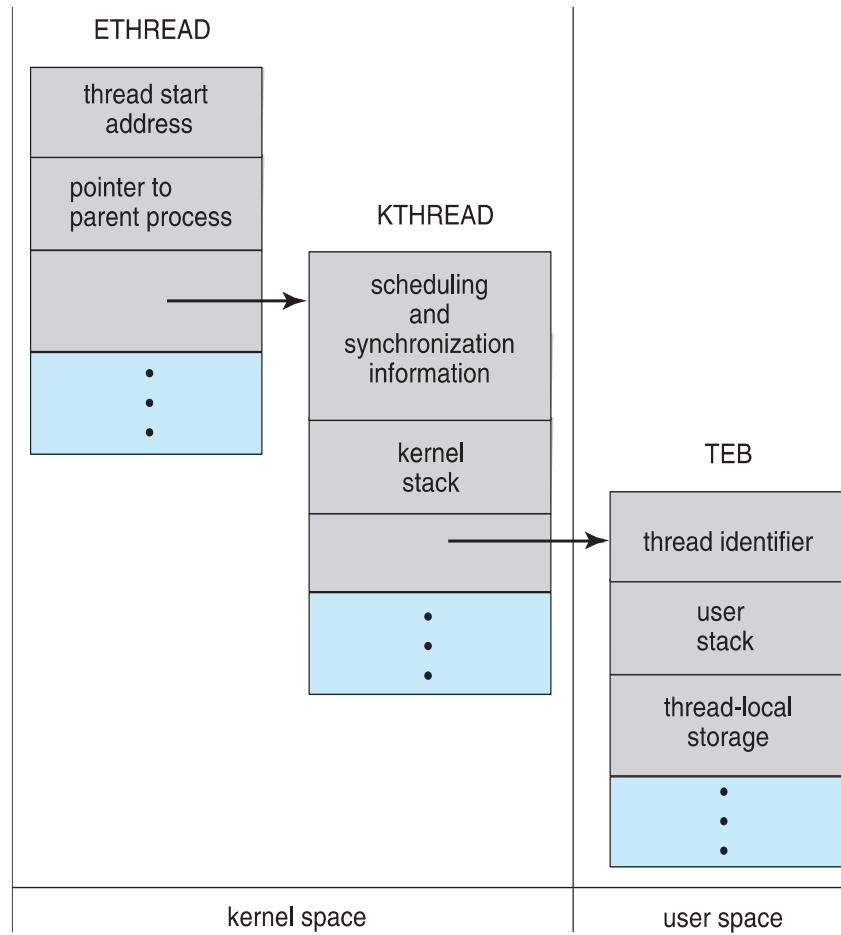
Windows Threads (Cont.)

- The primary data structures of a thread include:
 - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
 - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
 - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space





Windows Threads Data Structures





Linux Threads

- Linux does not distinguish between processes and threads.
 - Linux uses the term **task** than process or thread when referring to a flow of control within a program.
- Linux provides the **fork()** system call with the traditional functionality of duplicating a process,
- Linux also provides the ability to create threads using the **clone()** system call.
- The system **clone()** system call allows a child task to share the address space of the parent task (process).



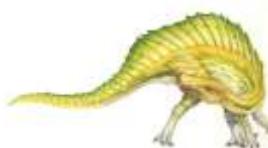


Linux Threads (Cont.)

- When **clone()** is invoked, it is passed a set of flags that determine how much sharing is to take place between the parent and child tasks.
- Flags control behavior

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- For example, suppose that **clone()** is passed the flags CLONE_FS, CLONE_VM, CLONE_SIGHAND and CLONE_FILES.
 - The parent and child tasks will then share the same file-system information, the same memory space, the same signal handlers, and the same set of open files.



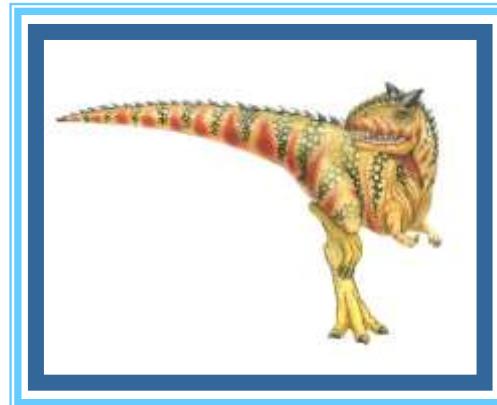


Linux Threads (Cont.)

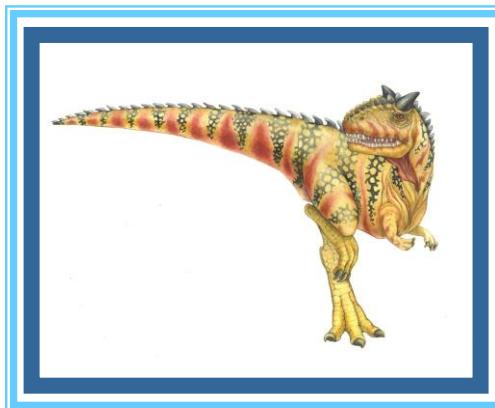
- Using `clone()` with all the flags set (as in the previous slide) is equivalent to creating a thread as described in this chapter, since the parent task shares most of its resources with its child task.
- If none of these flags is set when `clone()` is invoked, then no sharing takes place, resulting in functionality similar to that provided by the `fork()` system call.
- `struct task_struct` points to process data structures (shared or unique)



End of Chapter 4



Chapter 5a: CPU Scheduling





Chapter 5a: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation





Objectives

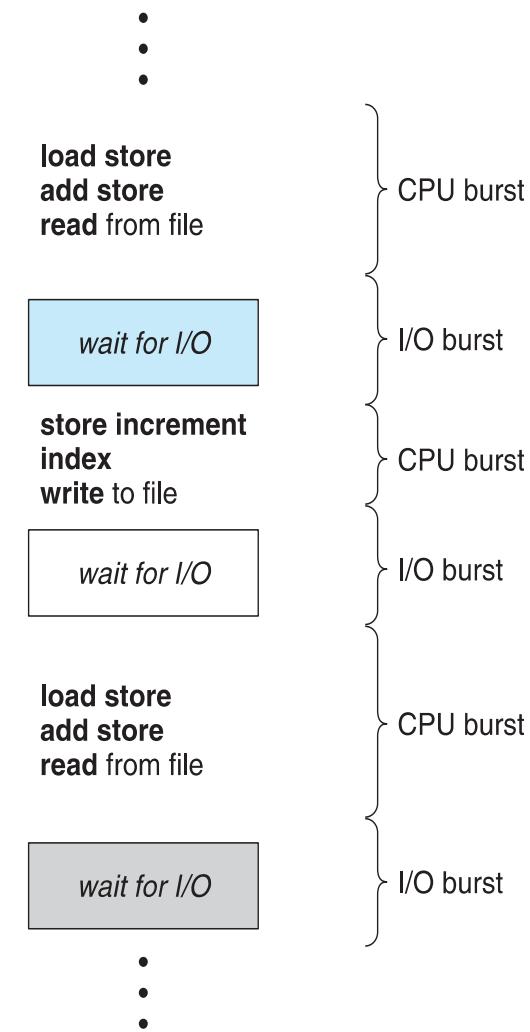
- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems





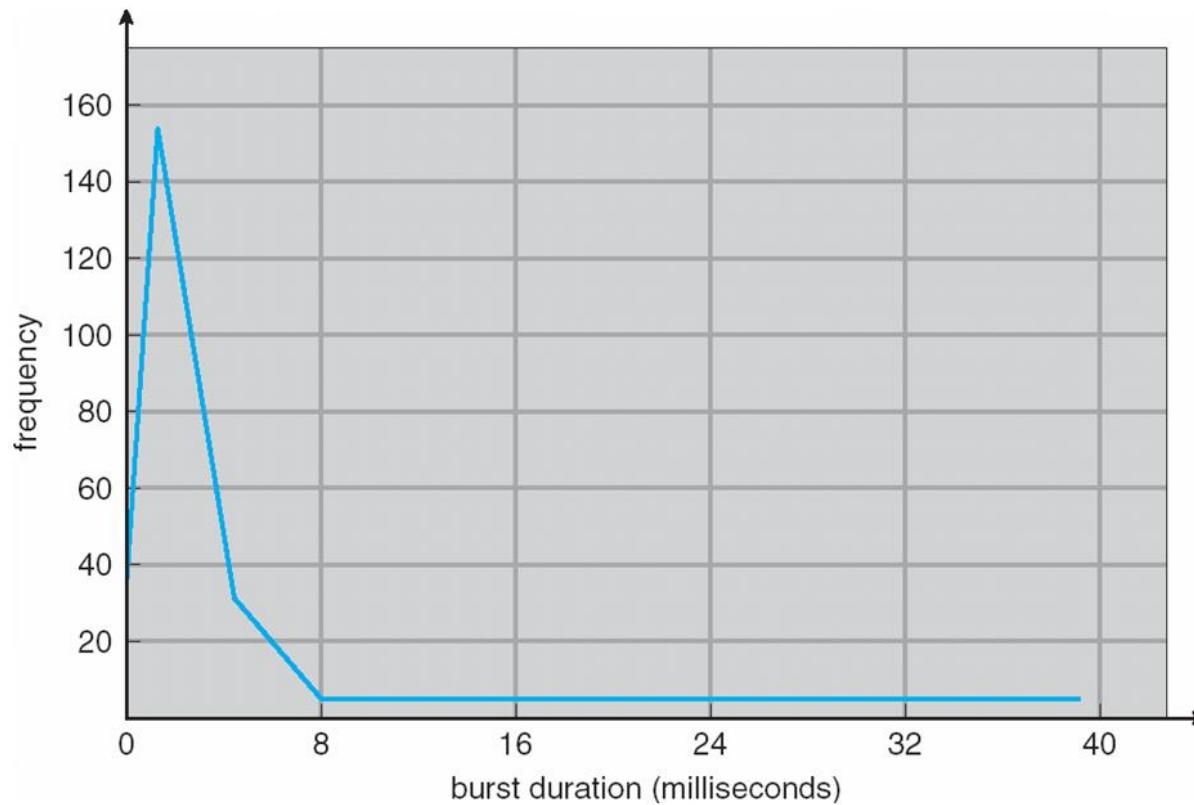
Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- Most processes exhibit the following behavior:
- **CPU burst** followed by **I/O burst**
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- CPU burst distribution is of main concern





Histogram of CPU-burst Times

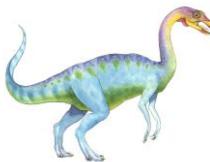




CPU Scheduler

- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.
- The selection process is carried out by the **CPU scheduler**.
- The ready queue may be ordered in various ways.
- CPU scheduling decisions may take place when a process:
 1. Switches from running state to waiting state
 2. Switches from running state to ready state
 3. Switches from waiting state to ready state
 4. When a process terminates
- For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution.
- There is a choice, however, for situations 2 and 3.





Nonpreemptive Scheduling

- Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU:
 - either by terminating
 - or by switching to the waiting state.





Preemptive scheduling

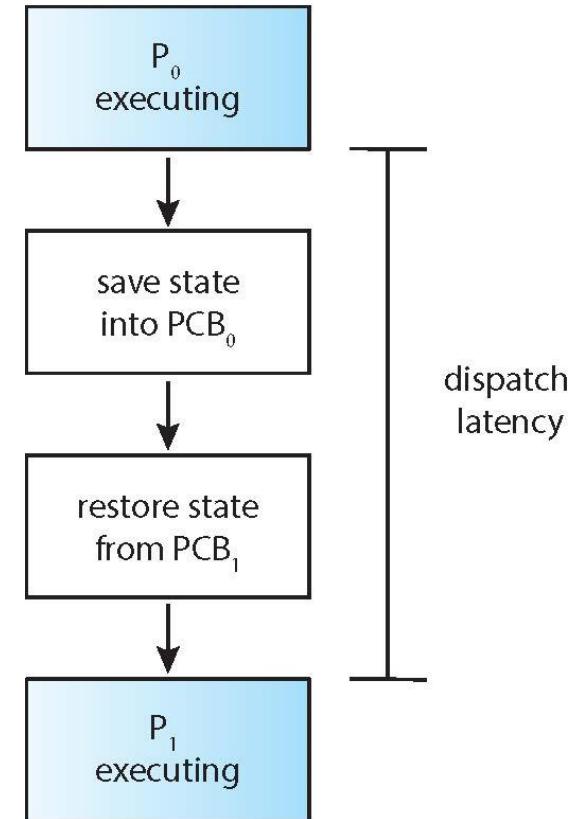
- Preemptive scheduling can result in race conditions when data are shared among several processes.
 - Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.
 - Consider preemption while in kernel mode
 - Consider interrupts occurring during crucial OS activities
- Virtually all modern operating systems including Windows, Mac OS X, Linux, and UNIX use preemptive scheduling algorithms.





Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the CPU scheduler; this involves:
 - Switching context
 - Switching to user mode
 - Jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running





Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – number of processes that complete their execution per time unit (e.g., 5 per second)
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – total amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)





Optimization Criteria for Scheduling

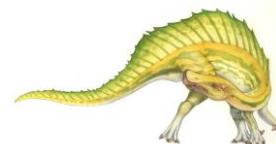
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time





Scheduling Algorithm

- First –come, First-serve (FCFS)
- Shortest-Job-First Scheduling (SJF)
- Round-Robin Scheduling (RR)
- Priority Scheduling
- Multilevel Queue Scheduling





First-Come, First-Served (FCFS) Scheduling

- Consider the following three processes and their burst time

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1, P_2, P_3
- We use [Gantt Chart](#) to illustrate a particular schedule



- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$





FCFS Scheduling (Cont.)

- Suppose that the processes arrive in the order:

P_2, P_3, P_1

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$, $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- **Convoy effect** - short process behind long process
 - Consider one CPU-bound and many I/O-bound processes





Shortest-Job-First (SJF)

- Associate with each process the length of its next CPU burst
 - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
 - How do we know what is the length of the next CPU request
 - Could ask the user
 - ▶ What if the user lies?



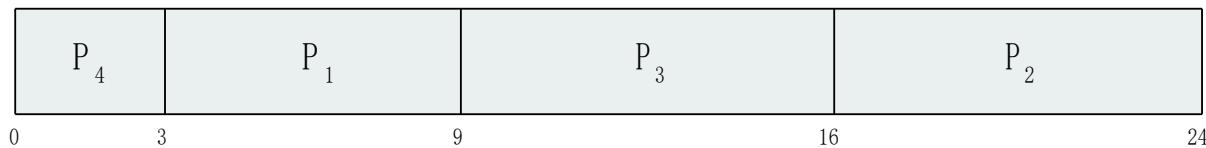


Example of SJF

- Consider the following four processes and their burst time

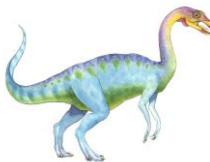
<u>Process</u>	<u>Burst Time</u>
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = $(3 + 16 + 9 + 0) / 4 = 7$





Determining Length of Next CPU Burst

- Can only estimate (predict) the length – in most cases should be similar to the previous CPU burst
 - Pick the process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging
 1. t_n = actual length of n^{th} CPU burst
 2. τ_{n+1} = predicted value for the next CPU burst
 3. $\alpha, 0 \leq \alpha \leq 1$
 4. Define : $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$.
- Commonly, α set to $\frac{1}{2}$





Examples of Exponential Averaging

- $\alpha = 0$
 - $\tau_{n+1} = \tau_n$
 - Recent history does not count
- $\alpha = 1$
 - $\tau_{n+1} = \alpha t_n$
 - Only the actual last CPU burst counts
- If we expand the formula, we get:

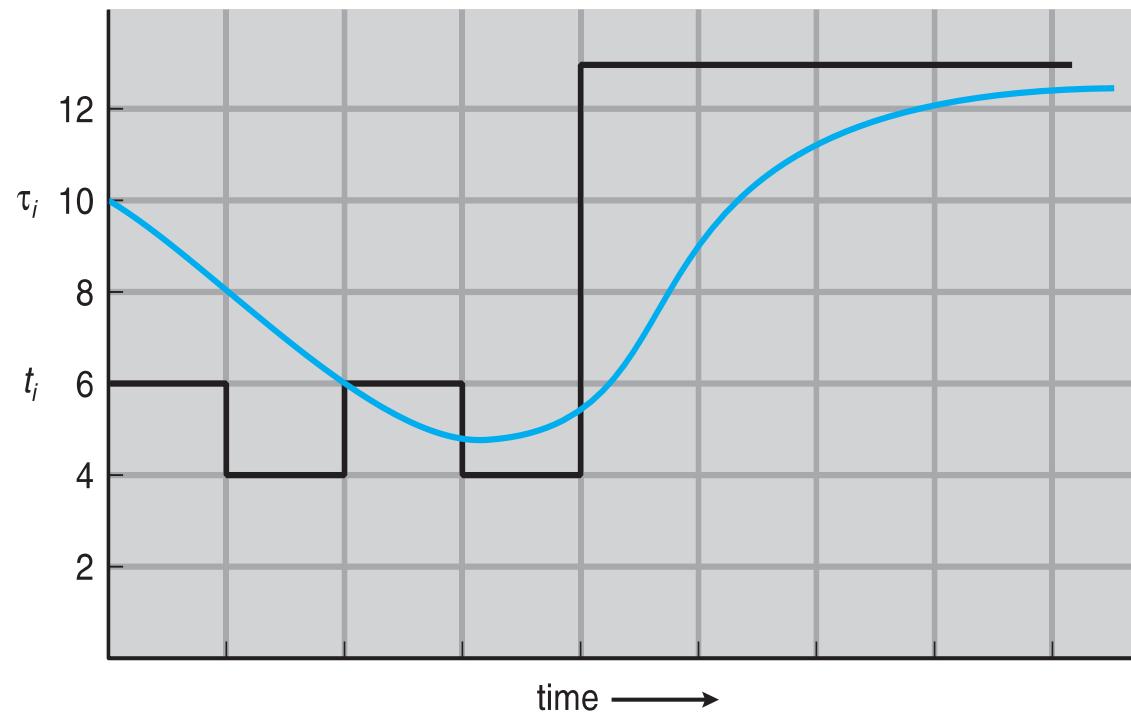
$$\begin{aligned}\tau_{n+1} &= \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots \\ &\quad + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ &\quad + (1 - \alpha)^{n+1} \tau_0\end{aligned}$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor





Prediction of the Length of the Next CPU Burst



CPU burst (t_i)	6	4	6	4	13	13	13	...	
"guess" (τ_i)	10	8	6	6	5	9	11	12	...



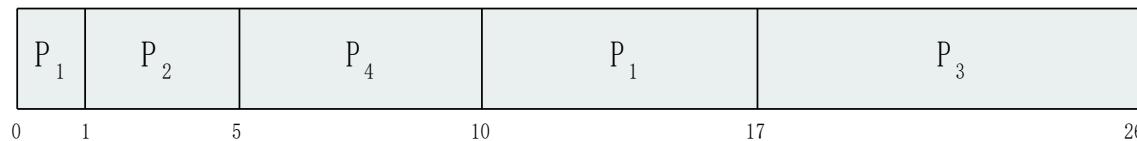


Shortest-remaining-time-first

- Preemptive version of SJF is called **shortest-remaining-time-first**
- Example illustrating the concepts of varying arrival times and preemption.

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive SJF Gantt Chart*



- Average waiting time = $[(10-1)+(1-1)+(17-2)+5-3]/4 = 26/4 = 6.5$ msec





Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** q). After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are N processes in the ready queue and the time quantum is q , then each process gets $1/N$ of the CPU time in chunks of at most q time units at once. No process waits more than $(N-1)*q$ time units.
- Timer interrupts every quantum to schedule next process
- Performance
 - q large \Rightarrow FIFO
 - q small \Rightarrow q must be large with respect to context switch, otherwise overhead is too high



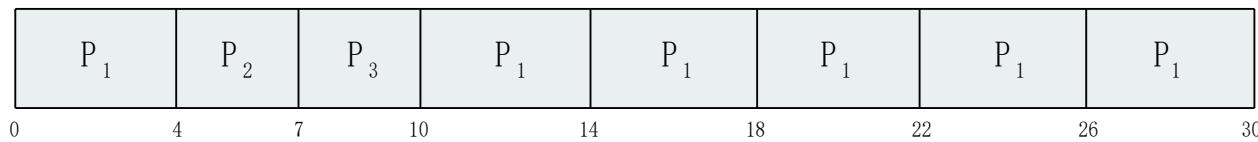


Example of RR with Time Quantum = 4

- Consider the following three processes and their burst time

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



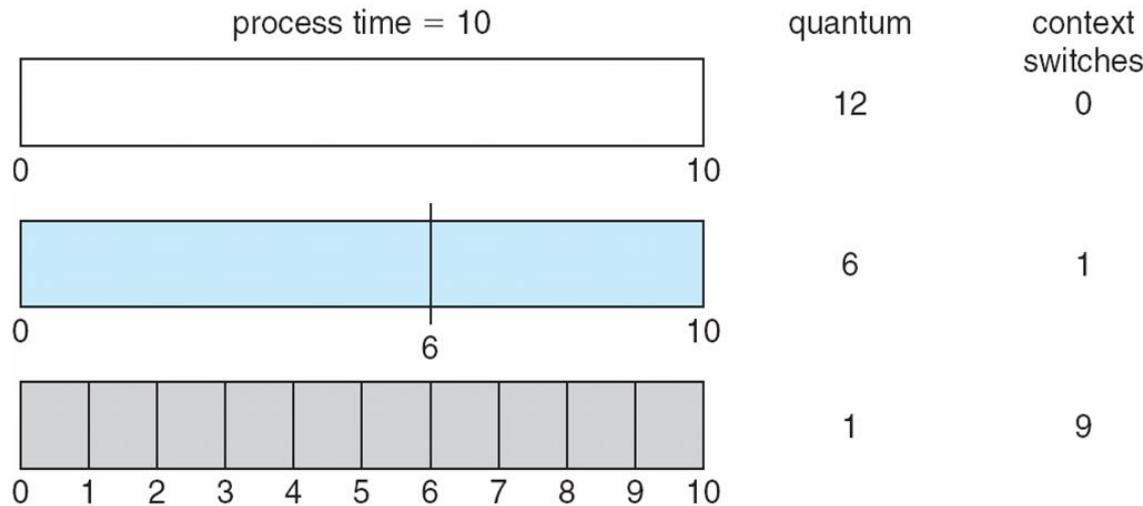
- The average waiting time under the RR policy is often longer
- Typically, higher average turnaround than SJF, but better **response**
- q should be large compared to context switch time
- q is usually 10ms to 100ms, context switch < 10 usec





Time Quantum and Context Switch Time

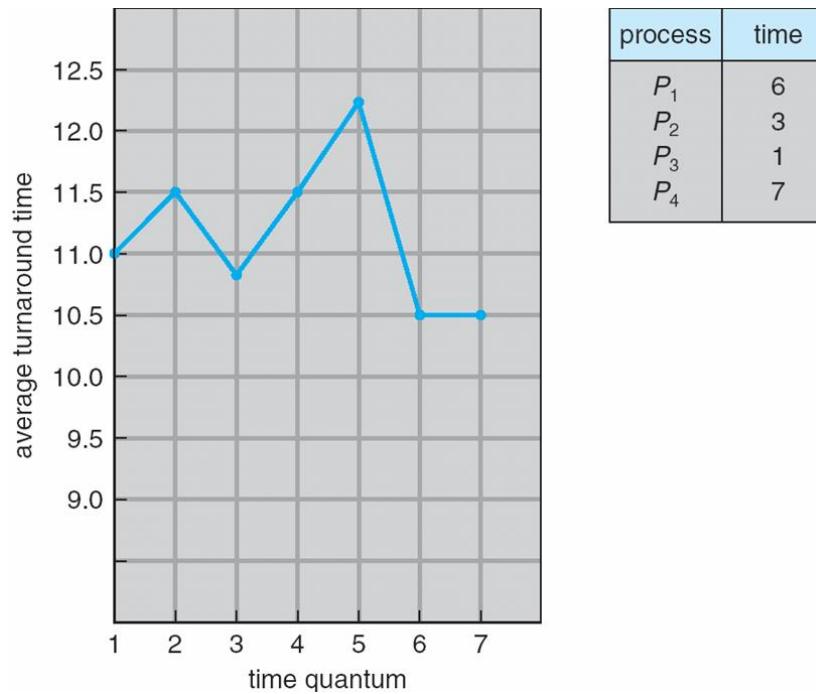
- The performance of the RR algorithm depends on the size of the time quantum. If the time quantum is extremely small (say, 1 millisecond), RR can result in a large number of context switches.





Turnaround Time Varies with the Time Quantum

- The average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.





Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer \equiv highest priority)
 - Preemptive
 - Non-preemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem \equiv **Starvation** – low priority processes may never execute
- Solution \equiv **Aging** – as time progresses increase the priority of the process

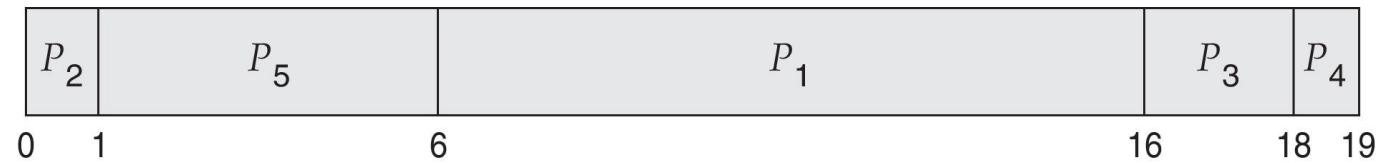




Example of Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec



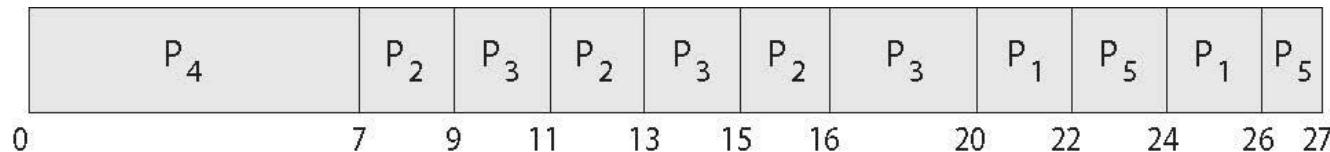


Combining Priority Scheduling and RR

- System executes the highest priority process; processes with the same priority will be run using round-robin.
- Consider the following five processes and their burst time

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

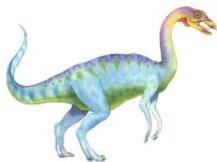




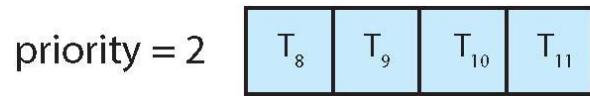
Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
 - **foreground** (interactive)
 - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
 - foreground – RR
 - background – FCFS
- Scheduling must be done between the queues:
 - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
 - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
 - 20% to background in FCFS





Separate Queue For Each Priority



•

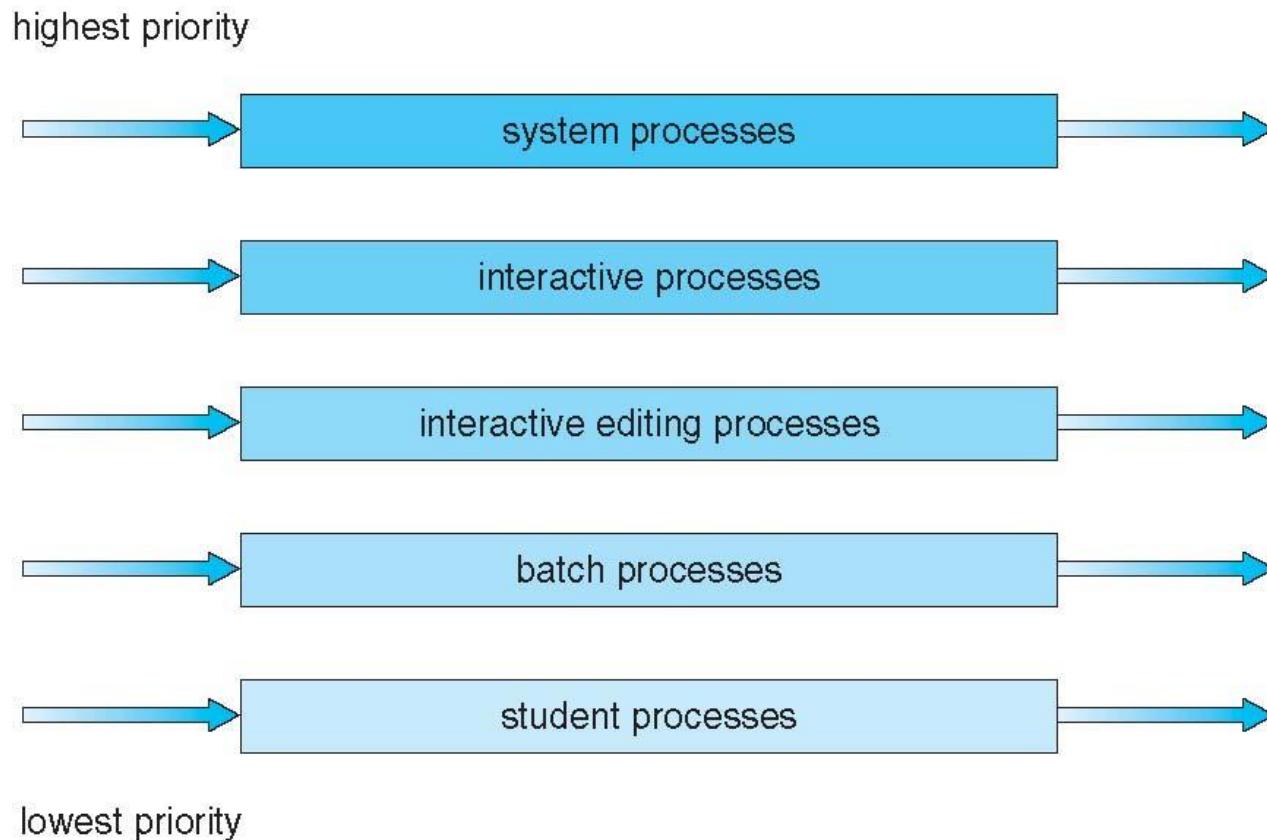
•

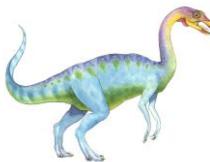
•





Multilevel Queue Scheduling





Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
 - number of queues
 - scheduling algorithms for each queue
 - method used to determine when to upgrade a process
 - method used to determine when to demote a process
 - method used to determine which queue a process will enter when that process needs service





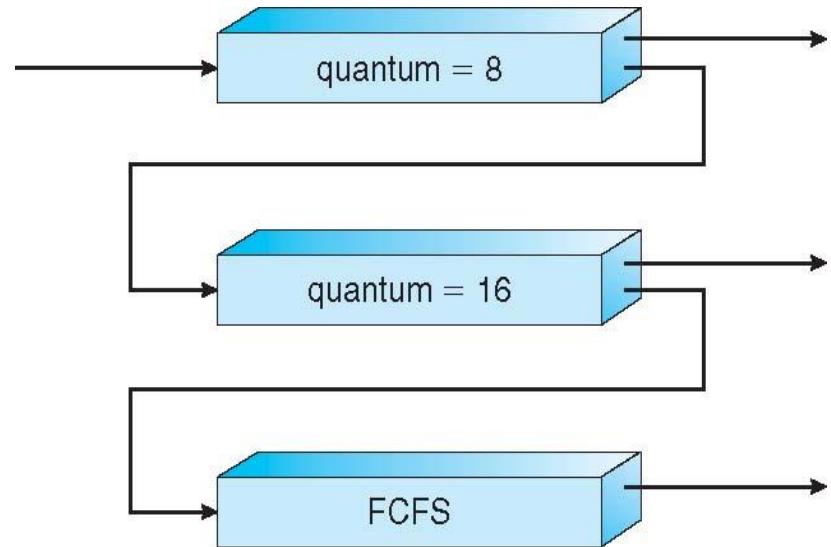
Example of Multilevel Feedback Queue

■ Three queues:

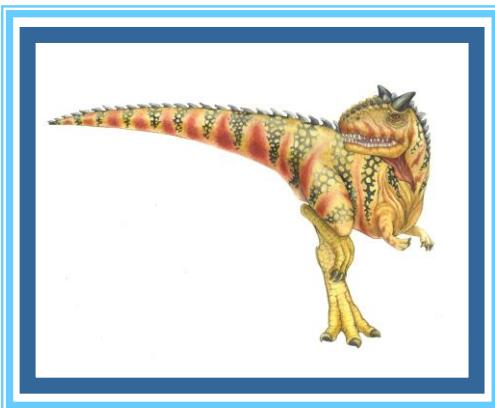
- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR time quantum 16 milliseconds
- Q_2 – FCFS

■ Scheduling

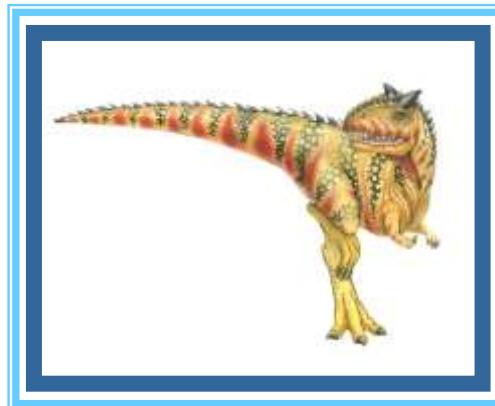
- A new job enters queue Q_0 which is served FCFS
 - When it gains CPU, job receives 8 milliseconds
 - If it does not finish in 8 milliseconds, job is moved to queue Q_1
- At Q_1 job is again served FCFS and receives 16 additional milliseconds
 - If it still does not complete, it is preempted and moved to queue Q_2



End of Chapter 5a



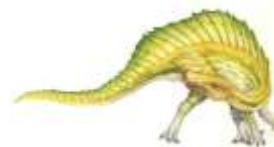
Chapter 5b: Advanced CPU Scheduling





Chapter 5b: CPU Scheduling

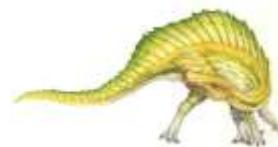
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation





Objectives

- Fill





Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
 - Known as **process-contention scope (PCS)** since scheduling competition is within the process
 - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system





Pthread Scheduling

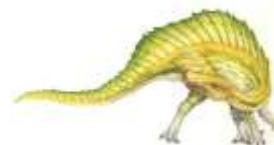
- API allows specifying either PCS or SCS during thread creation
 - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
 - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM





Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }
}
```





Pthread Scheduling API

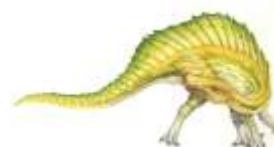
```
/* set the scheduling algorithm to PCS or SCS */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);
/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```





Multiprocessor CPU Scheduling

- CPU scheduling is more complex when multiple CPUs are available
- We assume that we are dealing with **homogeneous processors** within a multiprocessor system. This is, all processors are identical
- Two approaches to multiprocessor scheduling
 - **Asymmetric multiprocessing** – all scheduling decisions, I/O processing, and other system activities are handled by a single processor—the master server. The other processors execute only user code
 - **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
 - ▶ Currently, most common





Multicore Processors

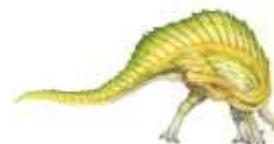
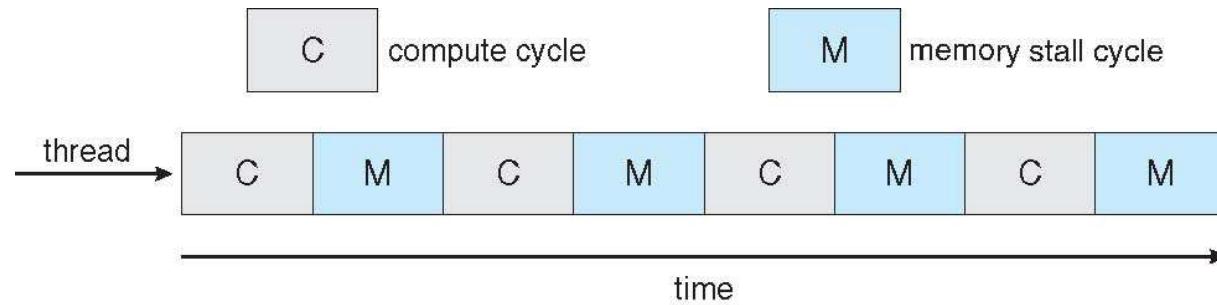
- SMP systems allow several threads to run concurrently by providing multiple physical processors.
- Multicore processor hardware places multiple computing cores on the same physical chip.
 - Each core maintains its architectural state and thus appears to the operating system to be a separate CPU.
- SMP systems that use multicore processors are faster and consume less power than systems in which each processor has its own physical chip.
- Multicore processors complicate scheduling issues





Memory Stall in Multicore Systems

- When a processor accesses memory, it spends a significant amount of time waiting for the data to become available. This situation, known as a memory stall
- Stall occurs because modern processors operate at much faster speeds than memory.





Multithreaded Multicore System

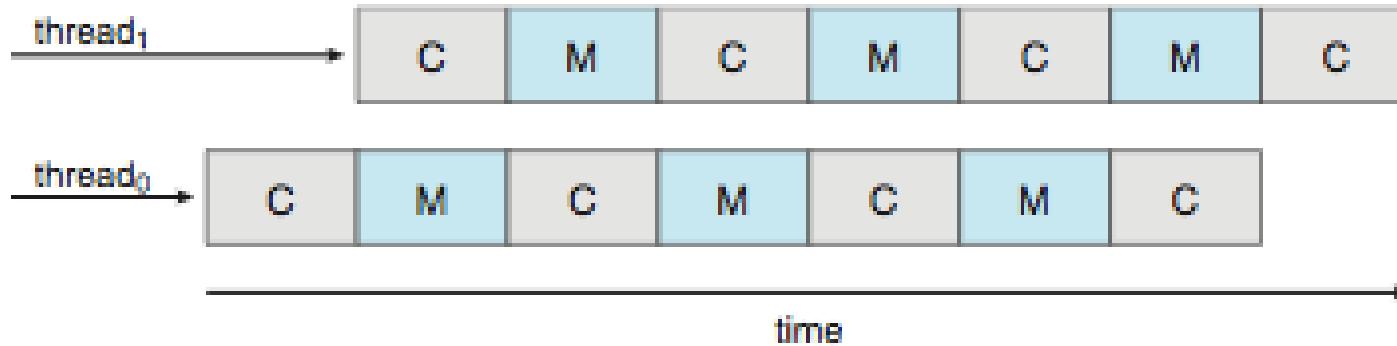




Figure 5.14

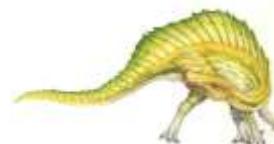
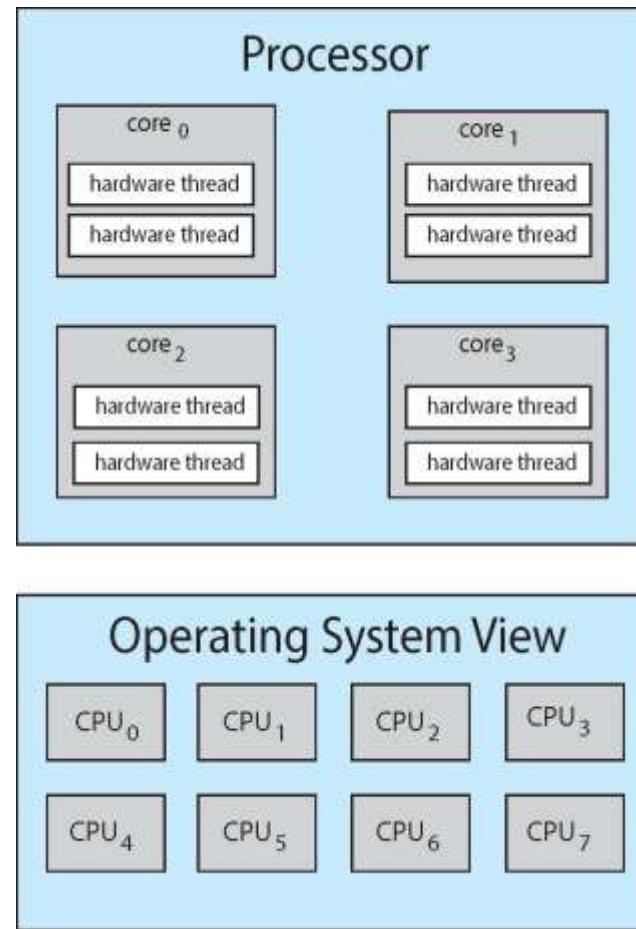
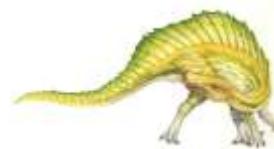
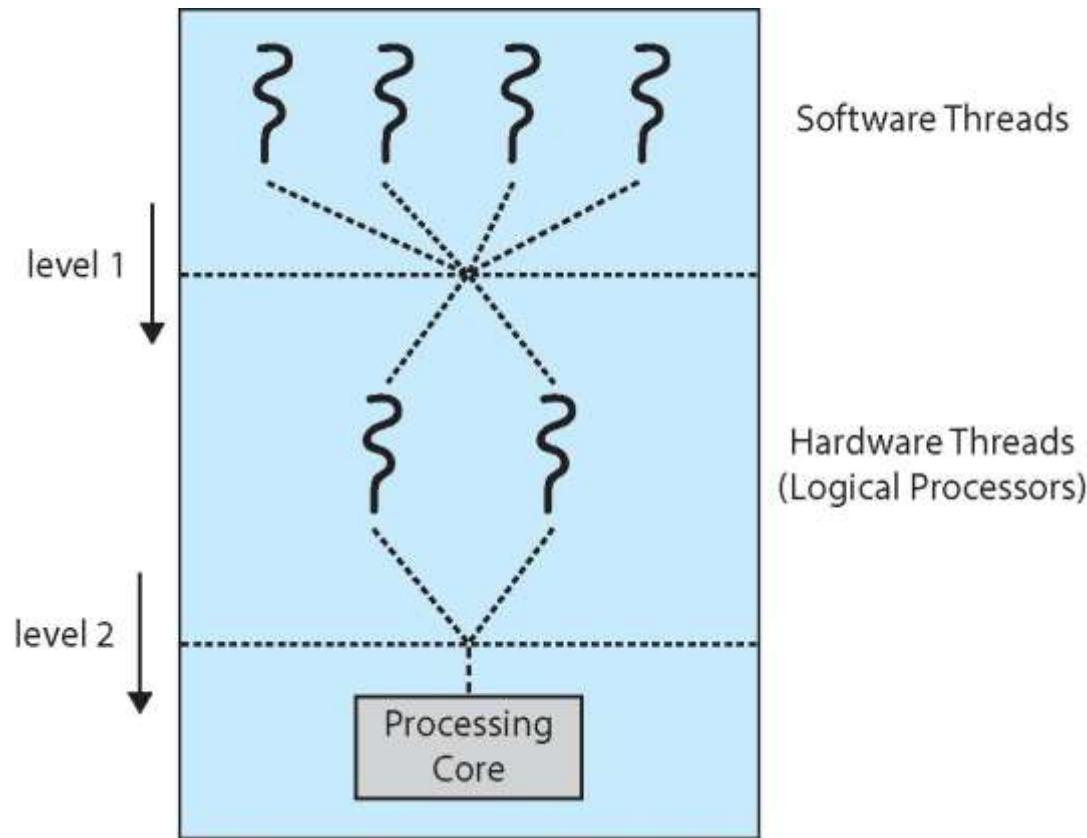


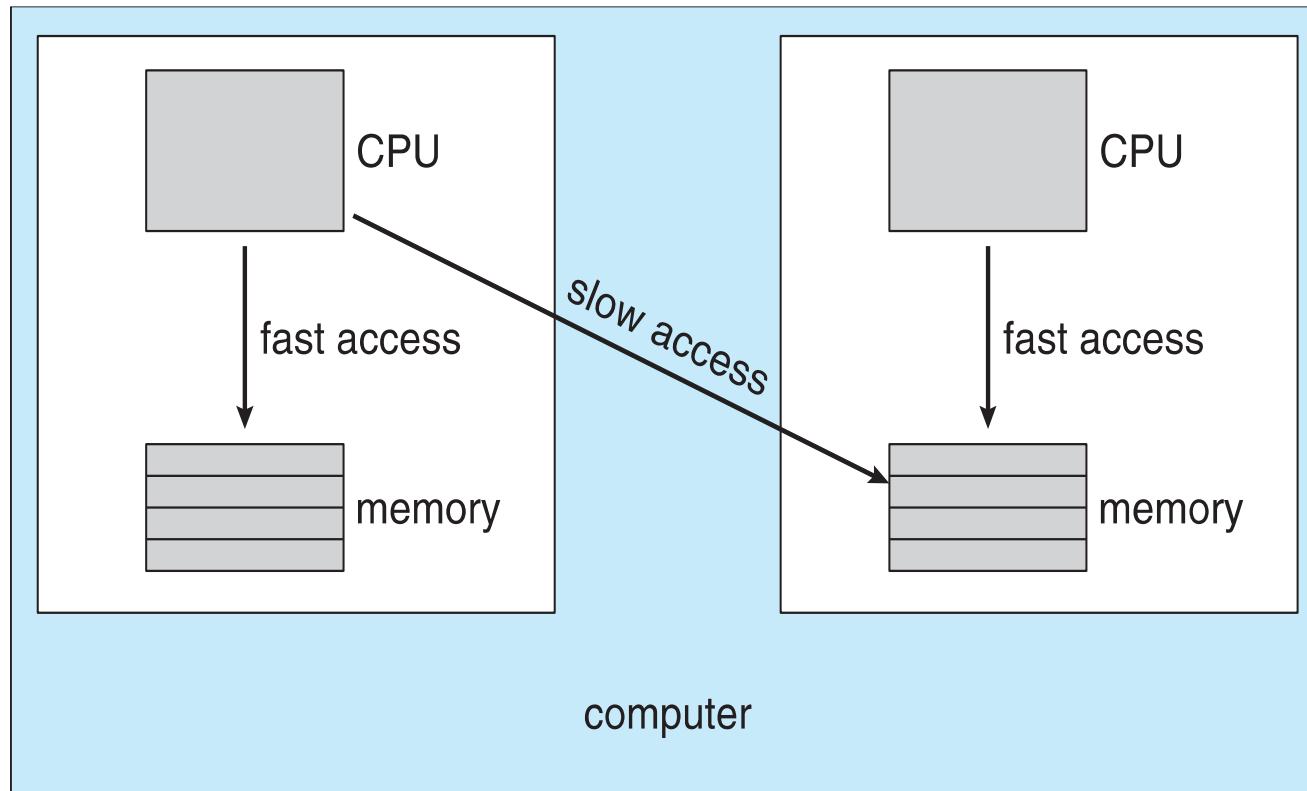


Figure 5.13





NUMA and CPU Scheduling

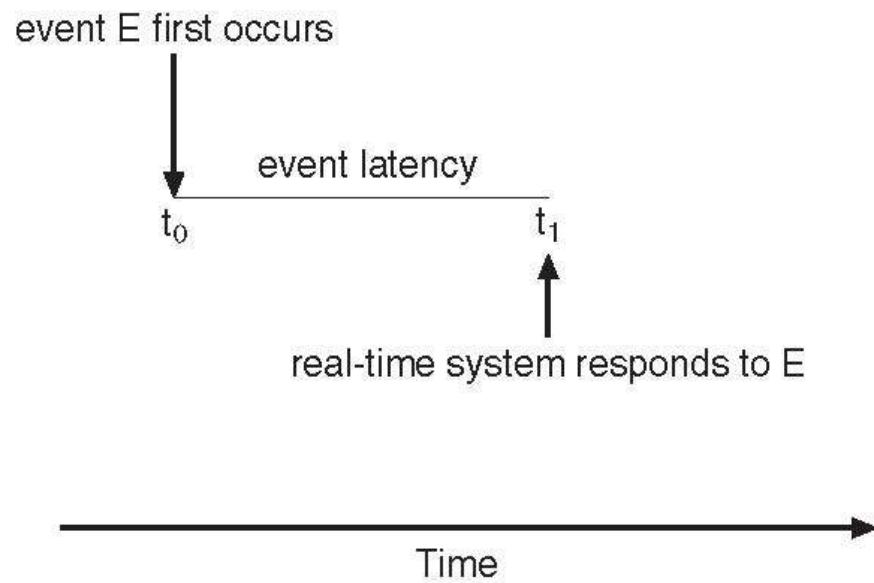


Note that memory-placement algorithms can also consider affinity





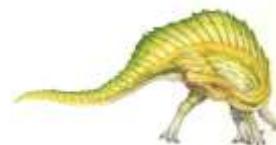
Figure 5.17





Multiple-Processor Scheduling – Load Balancing

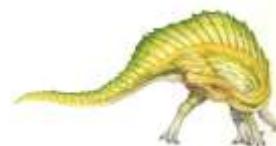
- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor





Multicore Processors

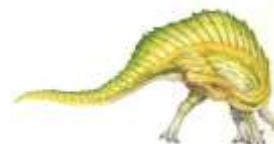
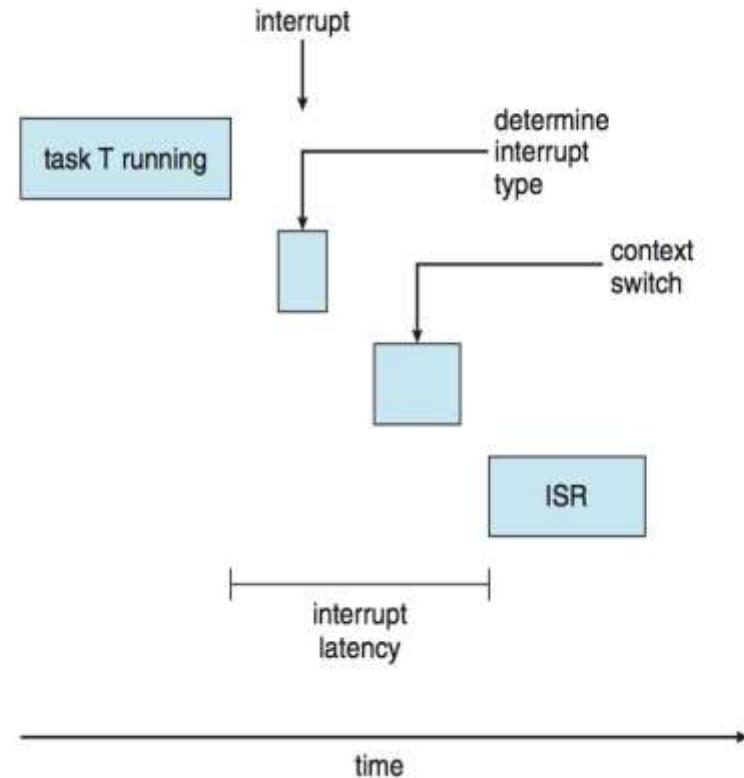
- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
 - Takes advantage of memory stall to make progress on another thread while memory retrieve happens





Real-Time CPU Scheduling

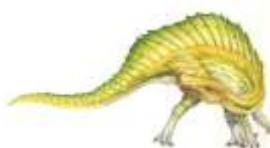
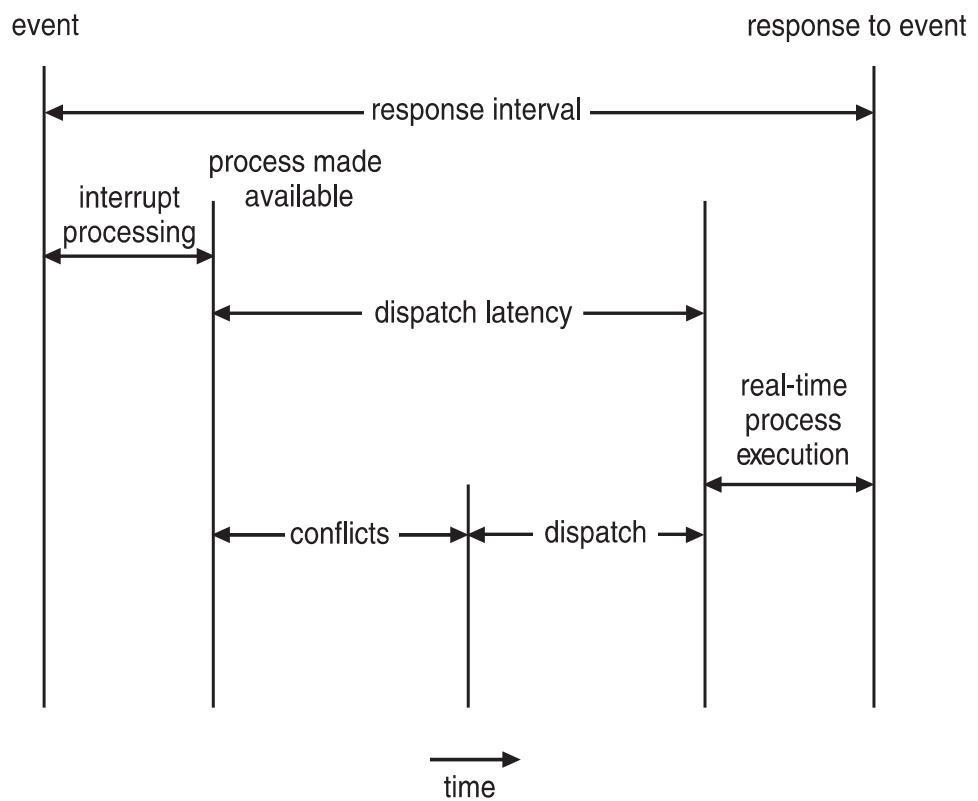
- Can present obvious challenges
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- Two types of latencies affect performance
 1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
 2. Dispatch latency – time for scheduler to take current process off CPU and switch to another





Real-Time CPU Scheduling (Cont.)

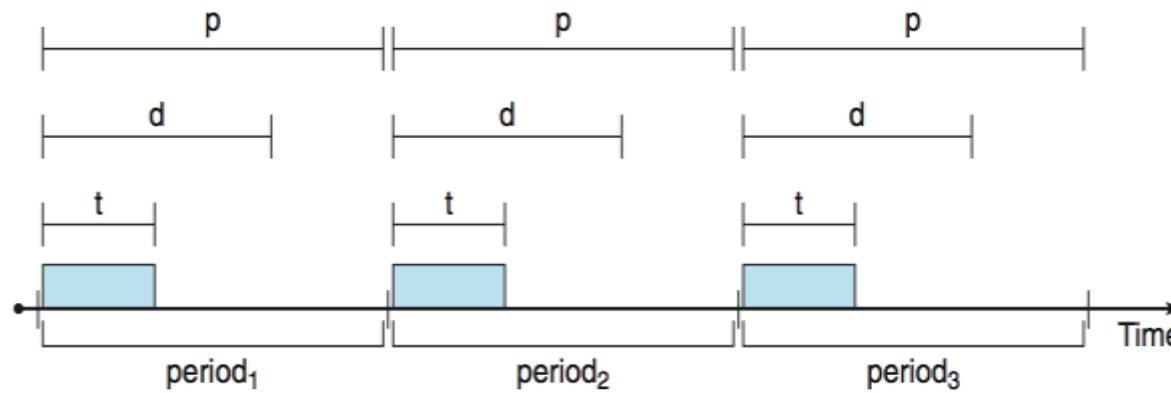
- Conflict phase of dispatch latency:
 1. Preemption of any process running in kernel mode
 2. Release by low-priority process of resources needed by high-priority processes





Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
 - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
 - Has processing time t , deadline d , period p
 - $0 \leq t \leq d \leq p$
 - **Rate** of periodic task is $1/p$





Virtualization and Scheduling

- Virtualization software schedules multiple guests onto CPU(s)
- Each guest doing its own scheduling
 - Not knowing it doesn't own the CPUs
 - Can result in poor response time
 - Can effect time-of-day clocks in guests
- Can undo good scheduling algorithm efforts of guests

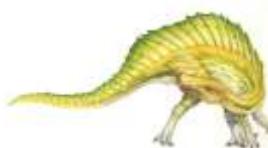




Figure 5.21

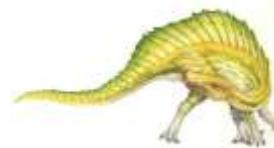
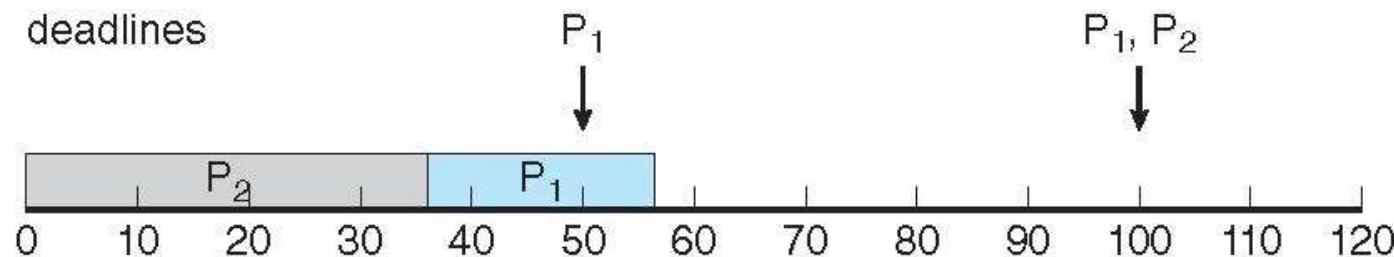




Figure 5.22

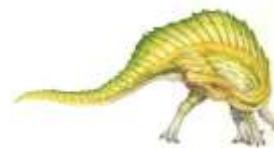
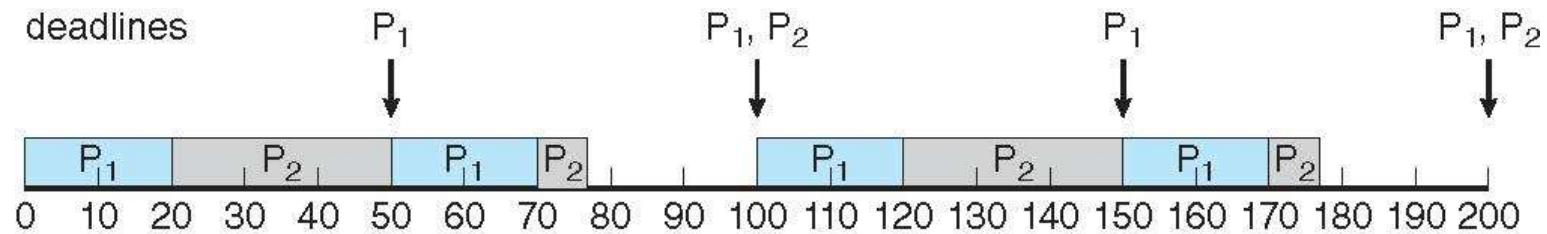
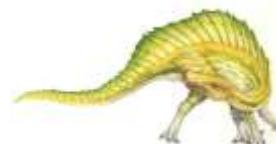
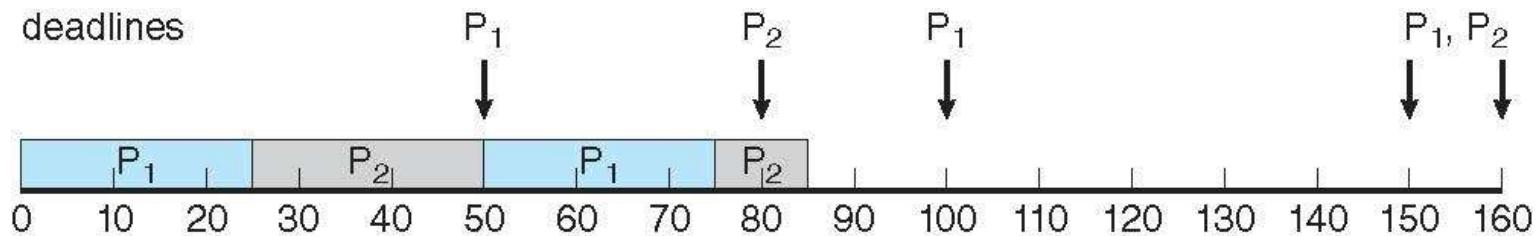


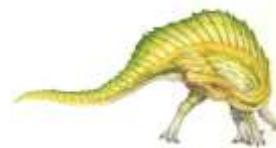
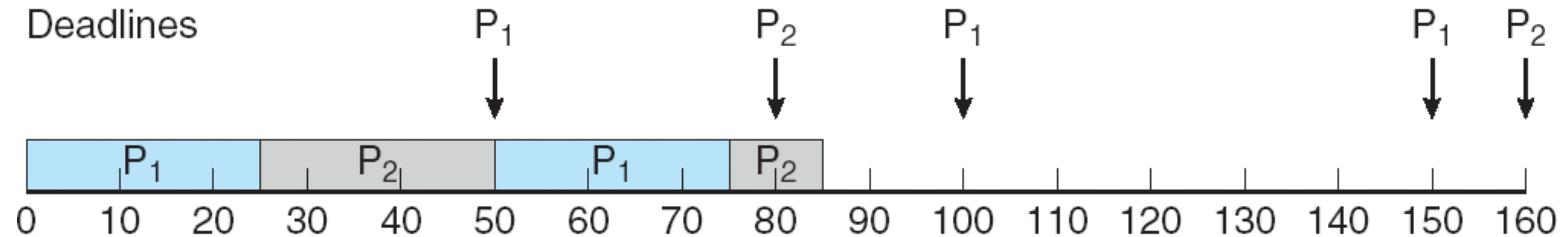


Figure 5.23





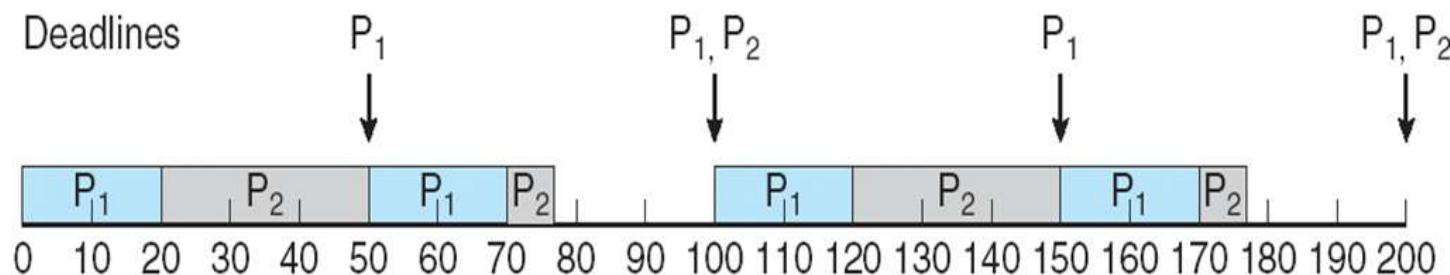
Missed Deadlines with Rate Monotonic Scheduling





Rate Montonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P_1 is assigned a higher priority than P_2 .

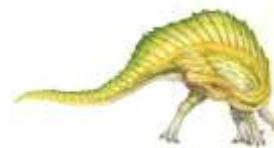
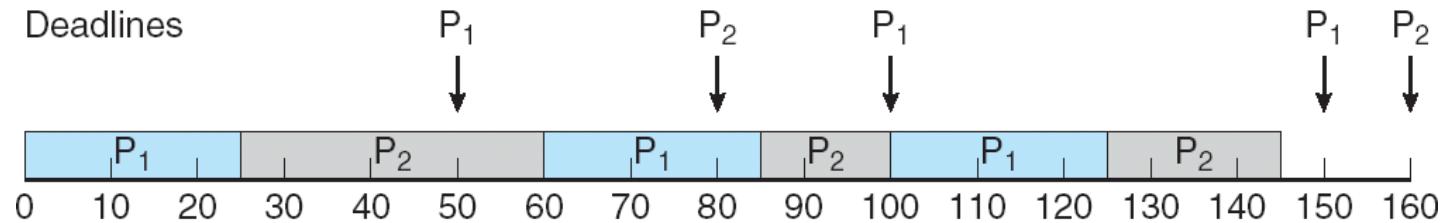




Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:

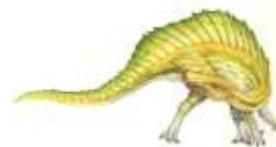
the earlier the deadline, the higher the priority;
the later the deadline, the lower the priority





Proportional Share Scheduling

- T shares are allocated among all processes in the system
- An application receives N shares where $N < T$
- This ensures each application will receive N / T of the total processor time





POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads:
 1. SCHED_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
 2. SCHED_RR - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority
- Defines two functions for getting and setting scheduling policy:
 1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
 2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`





POSIX Real-Time Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
    int i, policy;
    pthread_t_tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* get the current scheduling policy */
    if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
        fprintf(stderr, "Unable to get policy.\n");
    else {
        if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
        else if (policy == SCHED_RR) printf("SCHED_RR\n");
        else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
    }
}
```





POSIX Real-Time Scheduling API (Cont.)

```
/* set the scheduling policy - FIFO, RR, or OTHER */
if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
    fprintf(stderr, "Unable to set policy.\n");

/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);

/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);

}

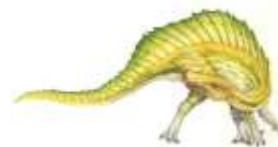
/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```





Operating System Examples

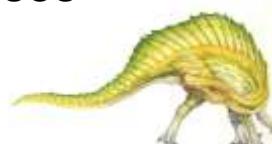
- Linux scheduling
- Windows scheduling
- Solaris scheduling





Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order $O(1)$ scheduling time
 - Preemptive, priority based
 - Two priority ranges: time-sharing and real-time
 - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
 - Map into global priority with numerically lower values indicating higher priority
 - Higher priority gets larger q
 - Task run-able as long as time left in time slice (**active**)
 - If no time left (**expired**), not run-able until all other tasks use their slices
 - All run-able tasks tracked in per-CPU **runqueue** data structure
 - ▶ Two priority arrays (active, expired)
 - ▶ Tasks indexed by priority
 - ▶ When no more active, arrays are exchanged
 - Worked well, but poor response times for interactive processes





Linux Scheduling in Version 2.6.23 +

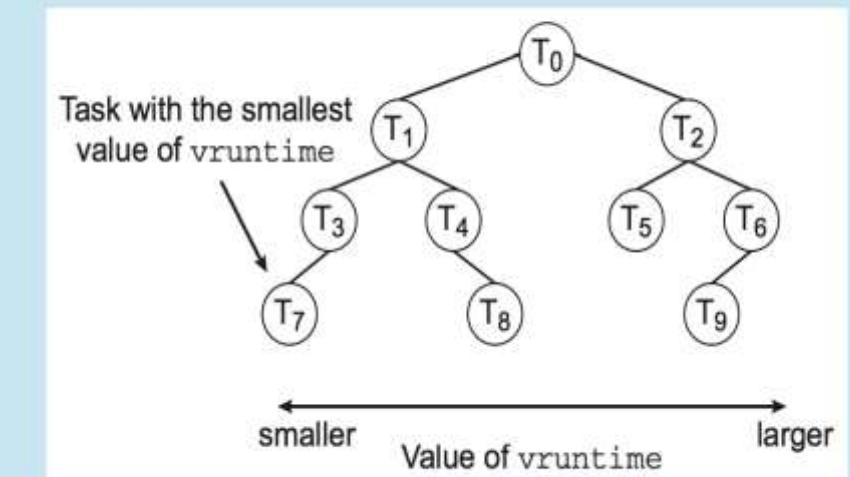
- **Completely Fair Scheduler (CFS)**
- **Scheduling classes**
 - Each has specific priority
 - Scheduler picks highest priority task in highest scheduling class
 - Rather than quantum based on fixed time allotments, based on proportion of CPU time
 - 2 scheduling classes included, others can be added
 1. default
 2. real-time
- Quantum calculated based on **nice value** from -20 to +19
 - Lower value is higher priority
 - Calculates **target latency** – interval of time during which task should run at least once
 - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual run time** in variable **vruntime**
 - Associated with decay factor based on priority of task – lower priority is higher decay rate
 - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time



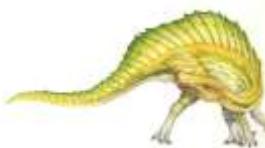


CFS Performance

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.





Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
 - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139

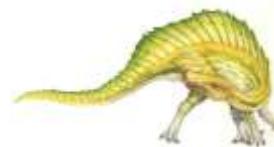
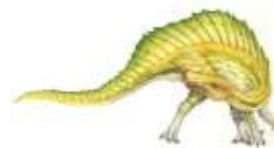
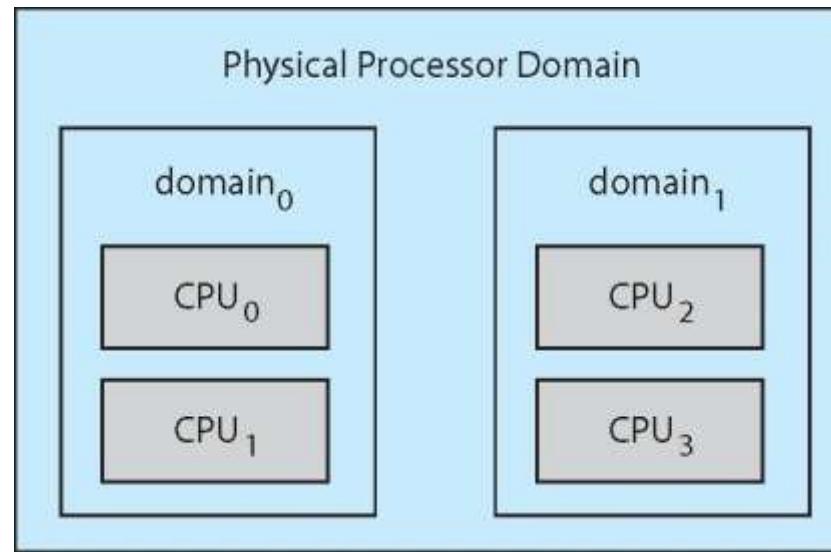




Figure 5.25





Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**





Windows Priority Classes

- Win32 API identifies several priority classes to which a process can belong
 - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS,
ABOVE_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS,
BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
 - All are variable except REALTIME
- A thread within a given priority class has a relative priority
 - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL,
LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base





Windows Priority Classes (Cont.)

- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x priority boost
- Windows 7 added **user-mode scheduling (UMS)**
 - Applications create and manage threads independent of kernel
 - For large number of threads, much more efficient
 - UMS schedulers come from programming language libraries like C++ **Concurrent Runtime** (ConcRT) framework





Windows Priorities

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1





Solaris

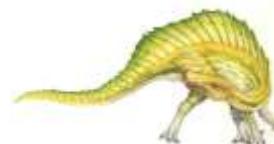
- Priority-based scheduling
- Six classes available
 - Time sharing (default) (TS)
 - Interactive (IA)
 - Real time (RT)
 - System (SYS)
 - Fair Share (FSS)
 - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
 - Loadable table configurable by sysadmin





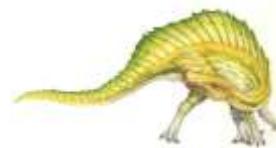
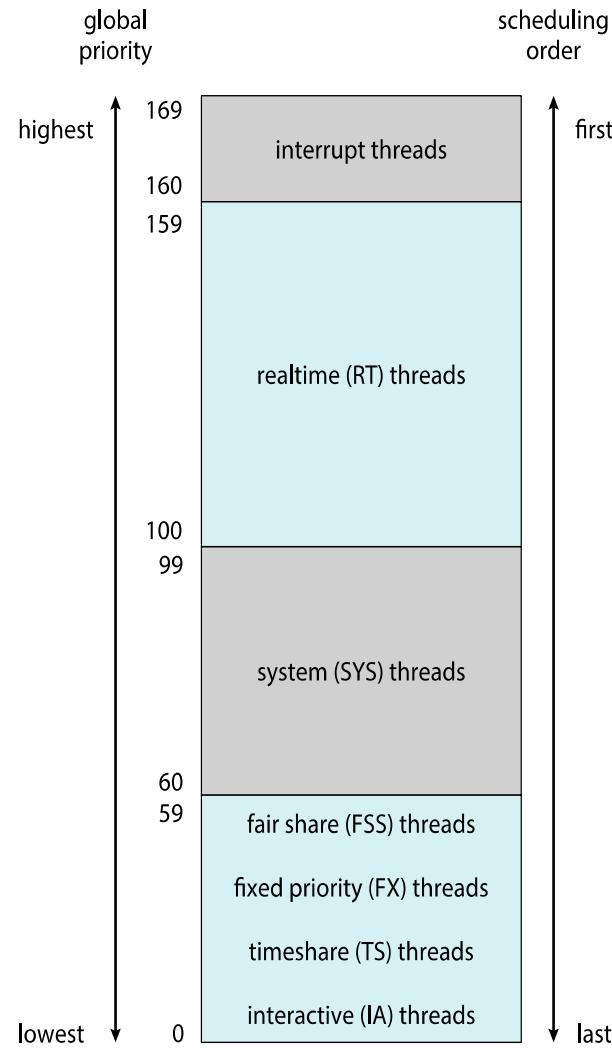
Solaris Dispatch Table

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59





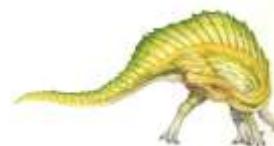
Solaris Scheduling





Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
 - Thread with highest priority runs next
 - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
 - Multiple threads at same priority selected via RR

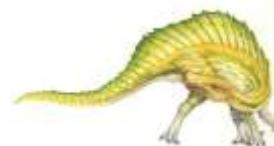




Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
 - Type of **analytic evaluation**
 - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

<u>Process</u>	<u>Burst Time</u>
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12



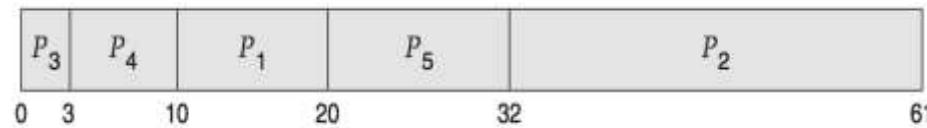


Deterministic Evaluation

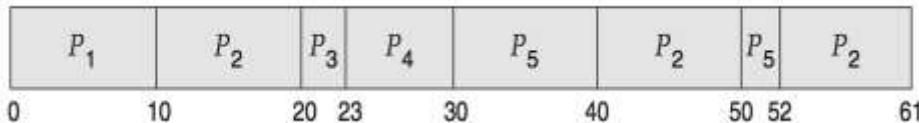
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
 - FCS is 28ms:



- Non-preemptive SJF is 13ms:



- RR is 23ms:





Queueing Models

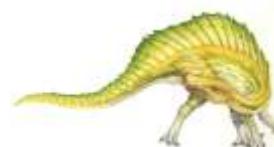
- Describes the arrival of processes, and CPU and I/O bursts probabilistically
 - Commonly exponential, and described by mean
 - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
 - Knowing arrival rates and service rates
 - Computes utilization, average queue length, average wait time, etc





Little's Formula

- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:
$$n = \lambda \times W$$
 - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds





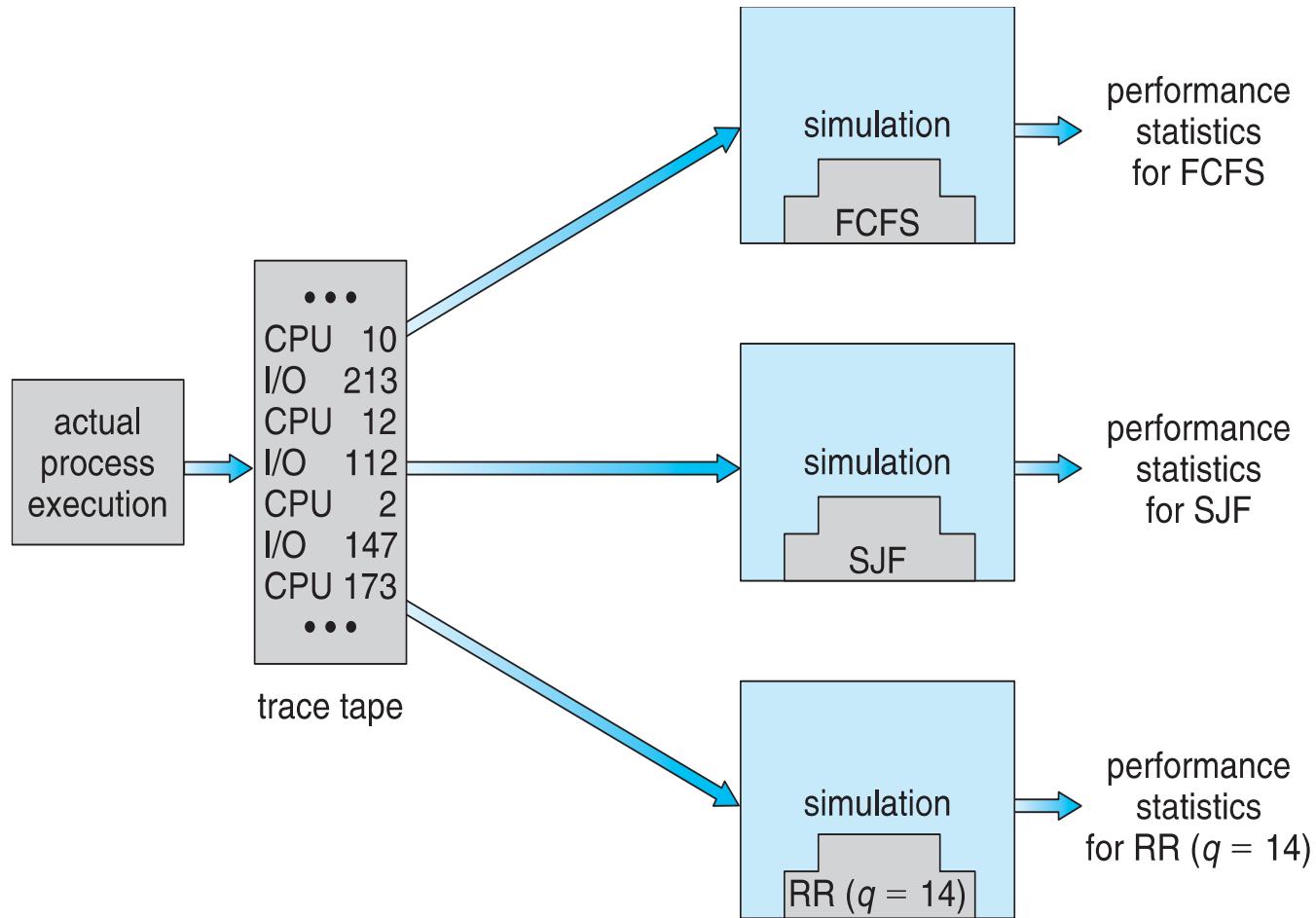
Simulations

- Queueing models limited
- **Simulations** more accurate
 - Programmed model of computer system
 - Clock is a variable
 - Gather statistics indicating algorithm performance
 - Data to drive simulation gathered via
 - ▶ Random number generator according to probabilities
 - ▶ Distributions defined mathematically or empirically
 - ▶ Trace tapes record sequences of real events in real systems





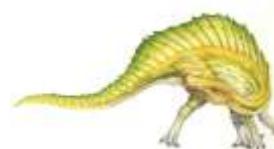
Evaluation of CPU Schedulers by Simulation





Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
 - High cost, high risk
 - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary



End of Chapter 5b

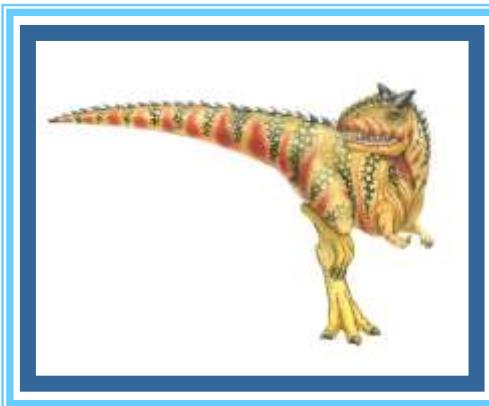




Figure 5.01

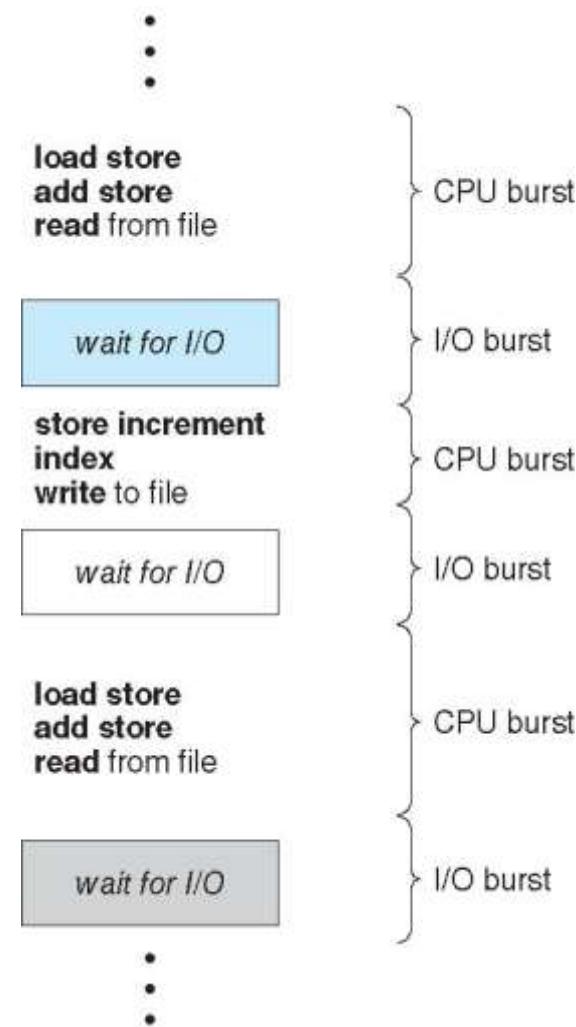




Figure 5.02

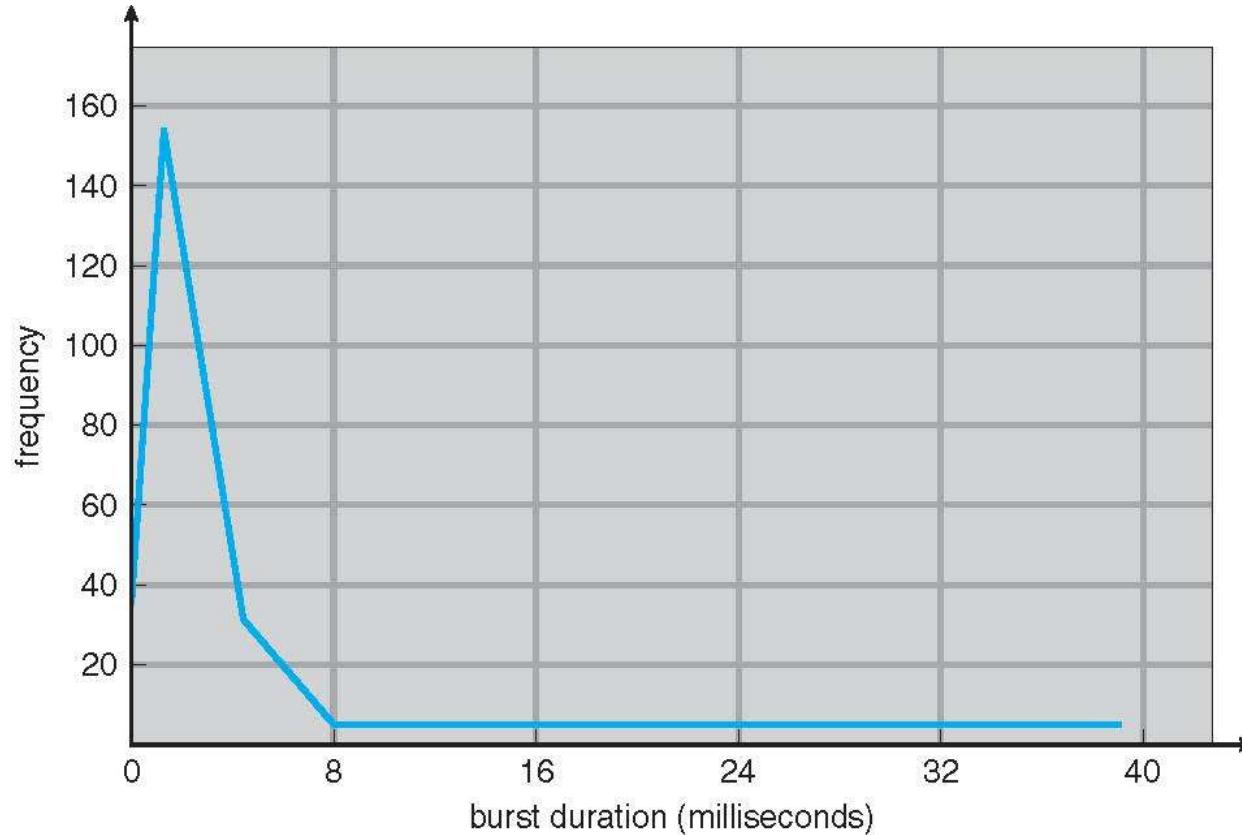
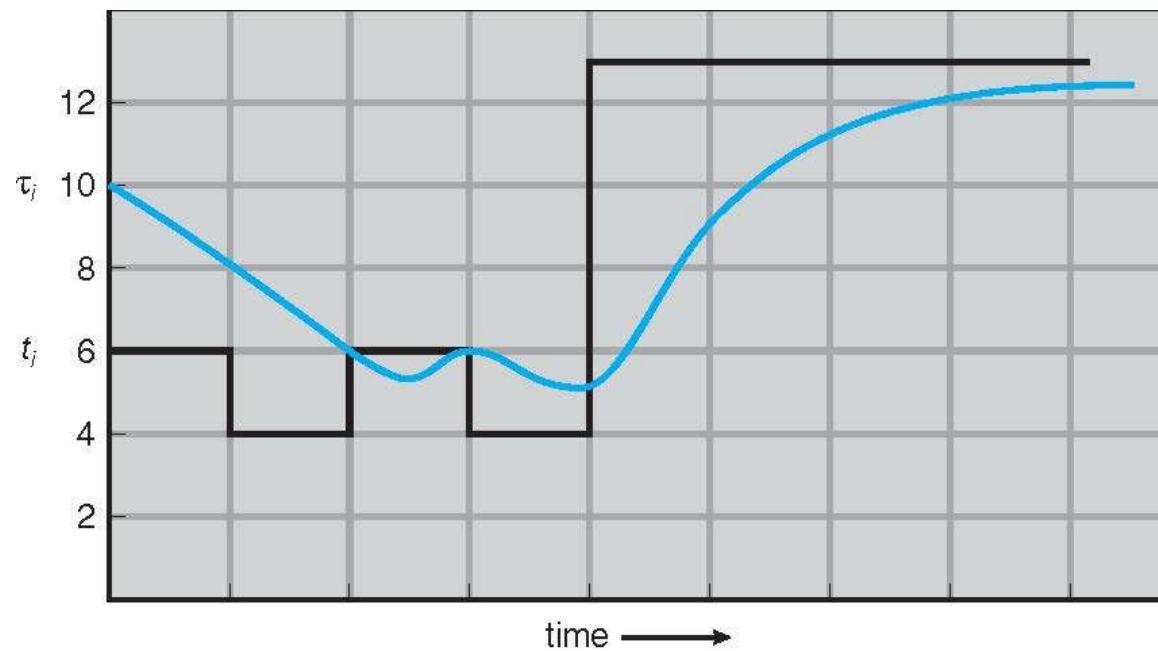




Figure 5.03



CPU burst (t_j)	6	4	6	4	13	13	13	...	
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

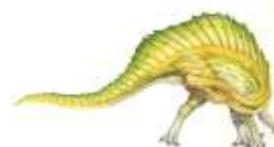




Figure 5.04

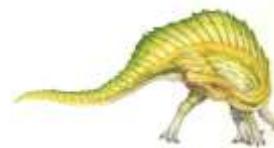
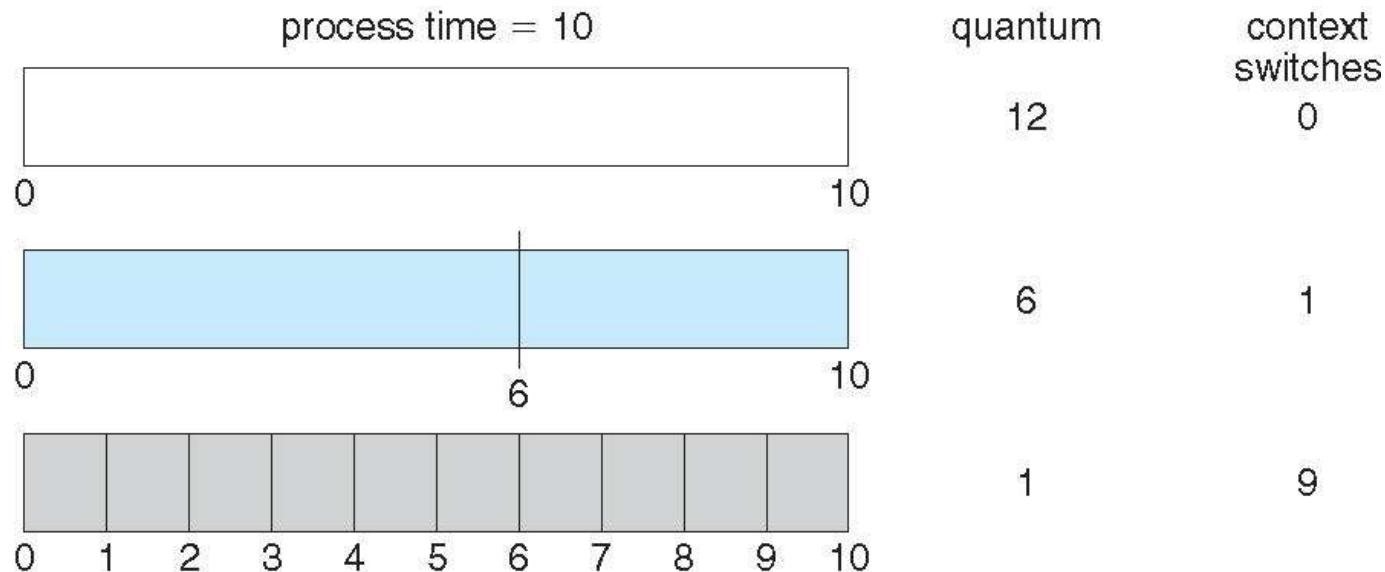
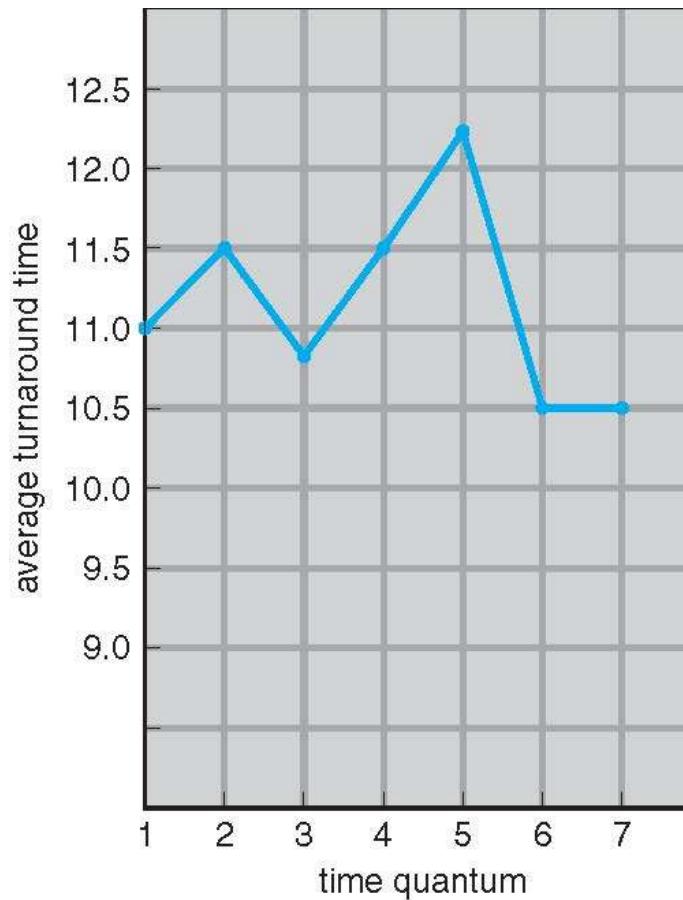




Figure 5.05



process	time
P_1	6
P_2	3
P_3	1
P_4	7

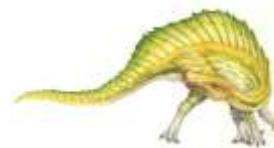




Figure 5.06

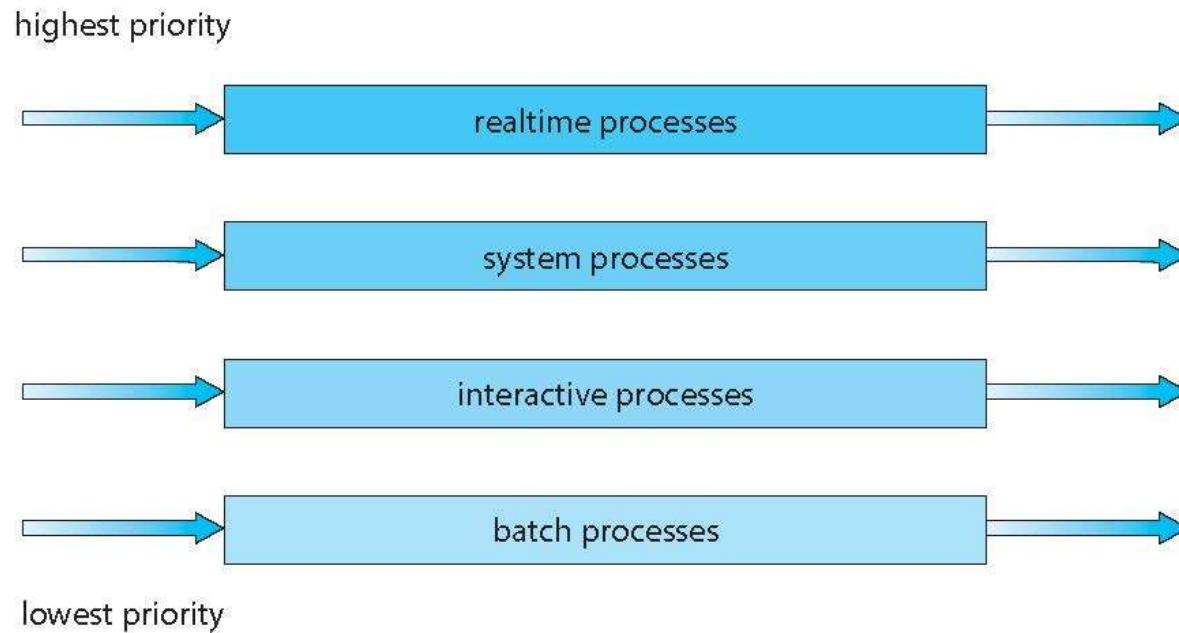




Figure 5.07

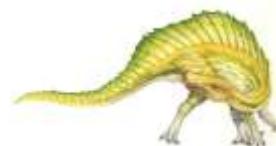
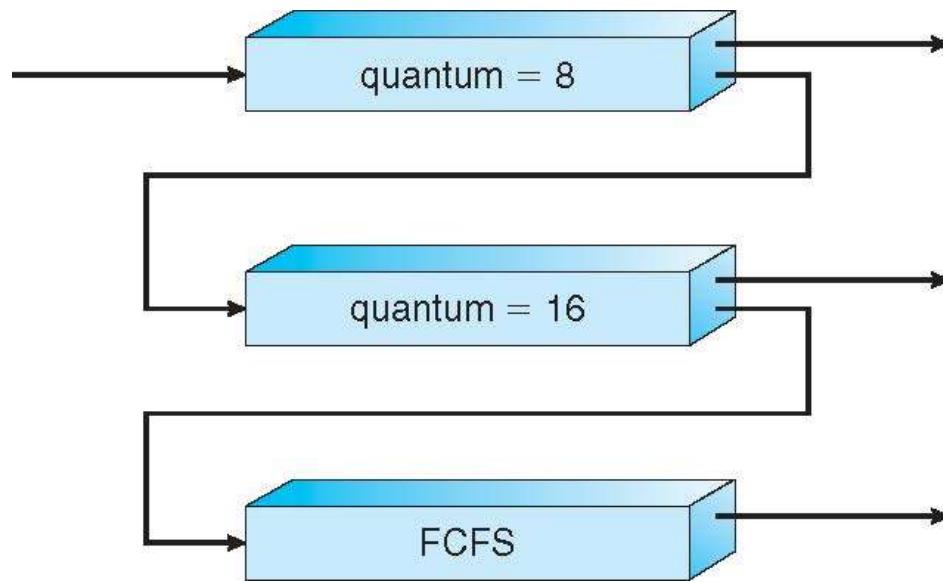




Figure 5.09

Place Holder

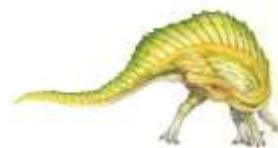




Figure 5.10

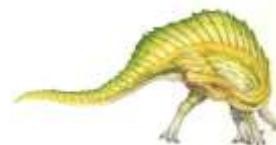
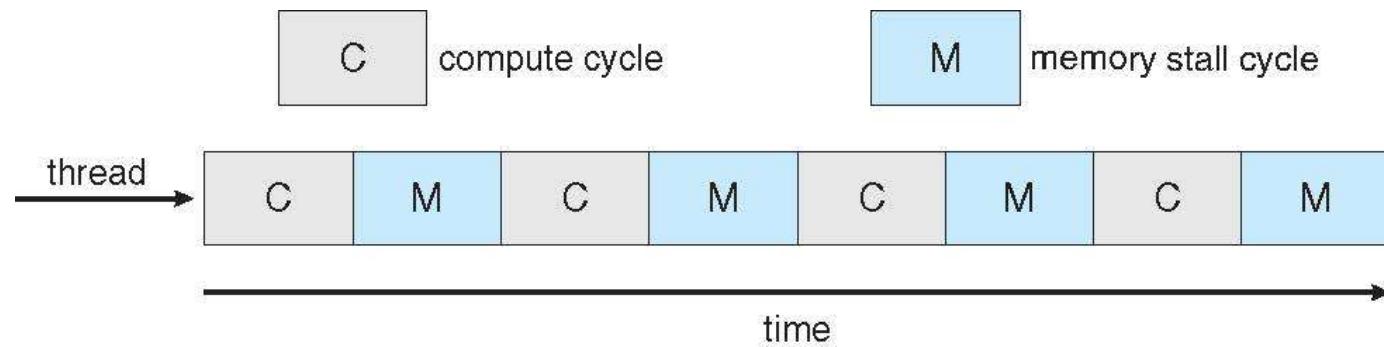




Figure 5.11

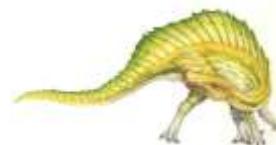
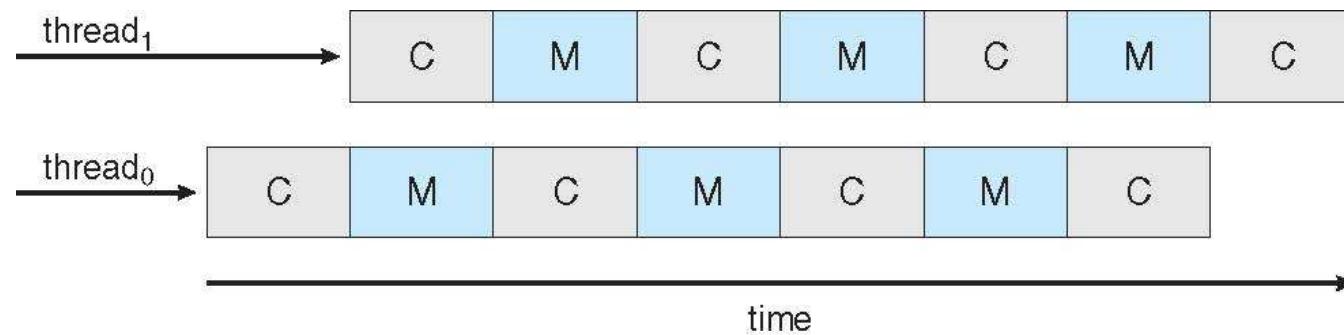




Figure 5.14

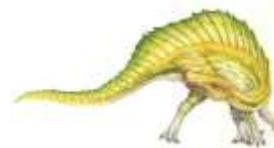
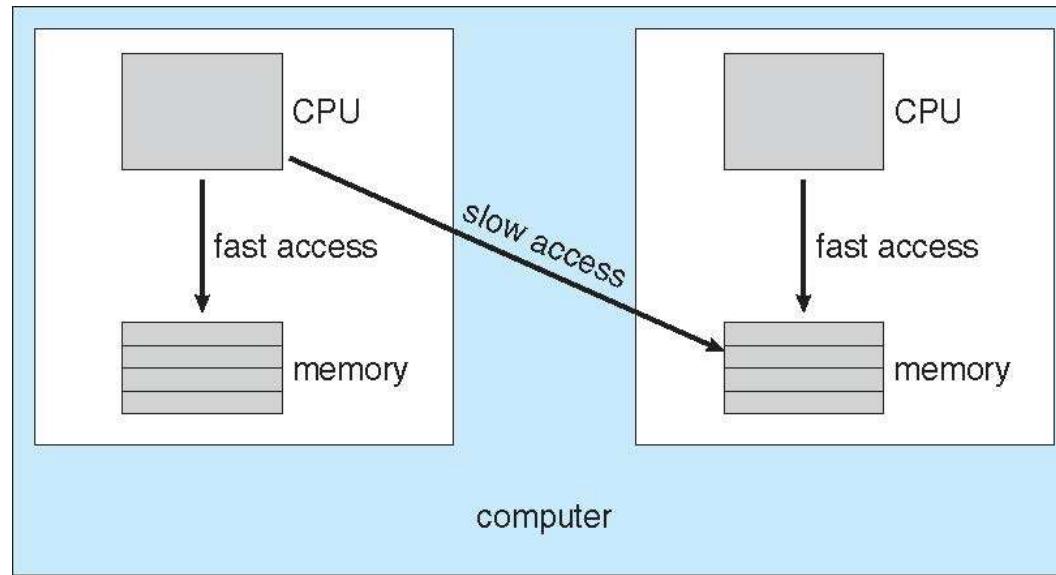




Figure 5.16

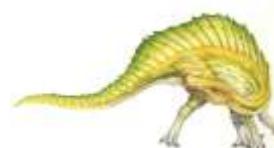
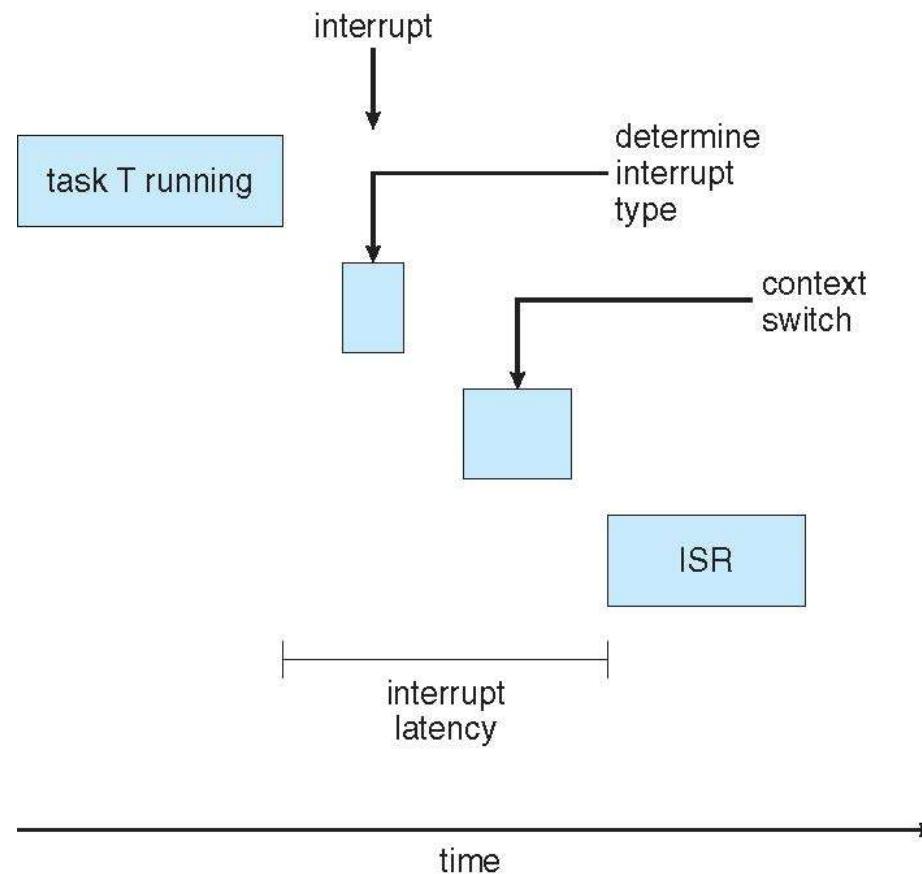




Figure 5.17

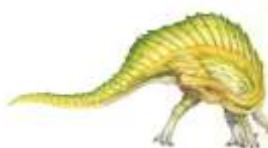
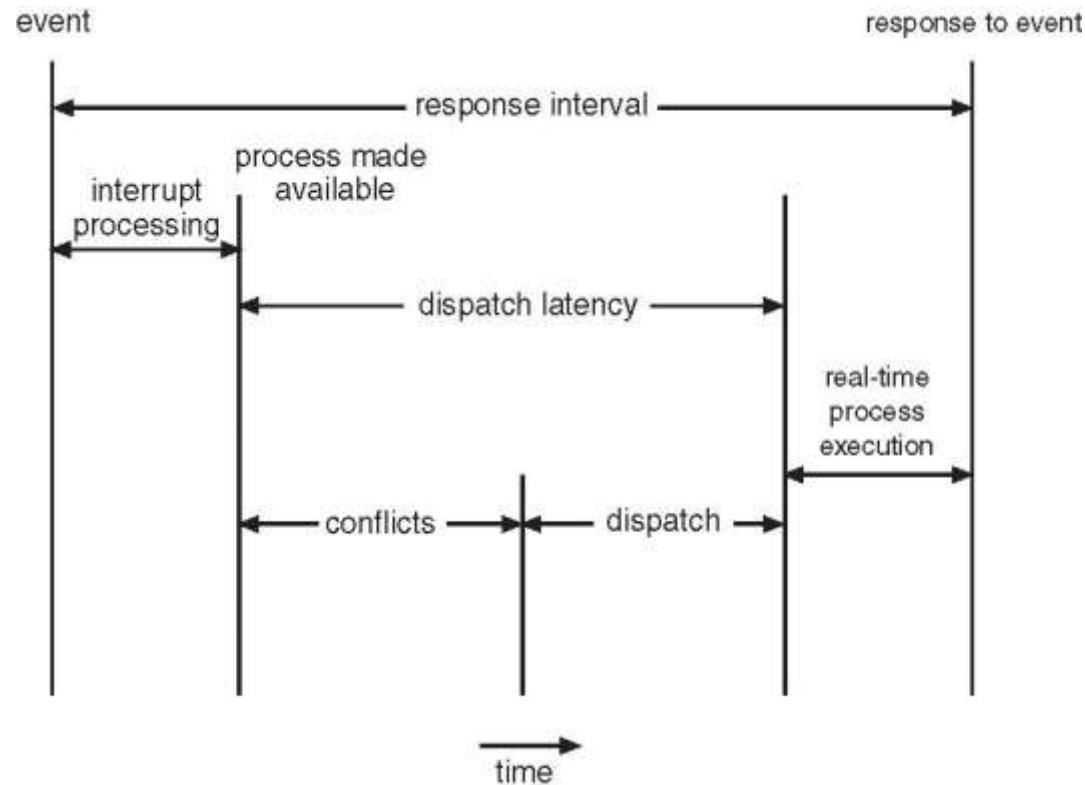




Figure 5.18

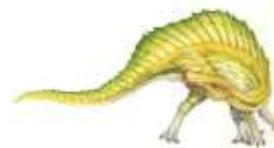
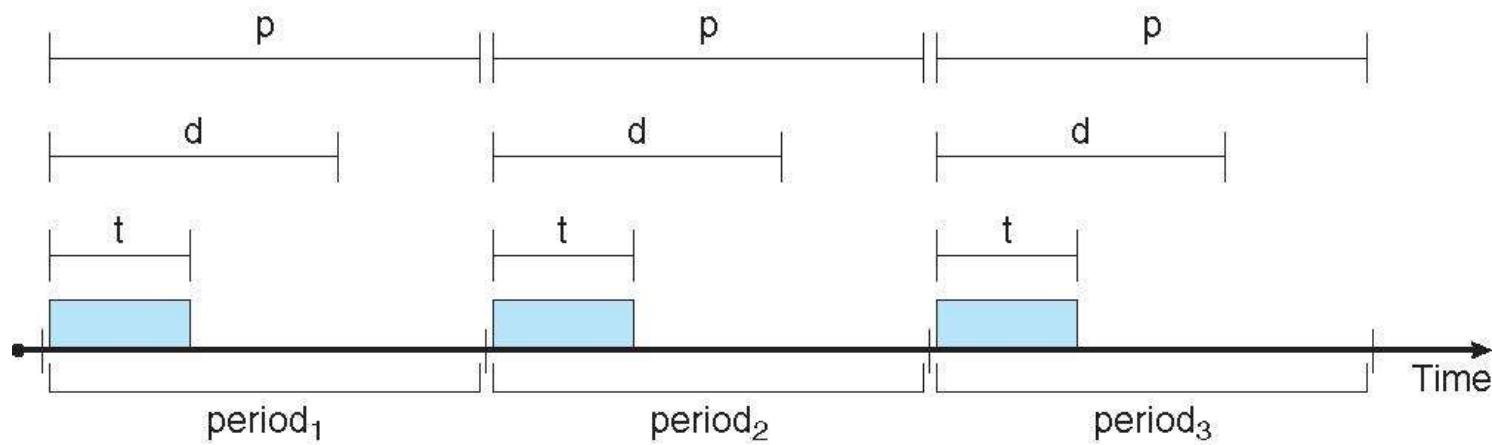




Figure 5.22

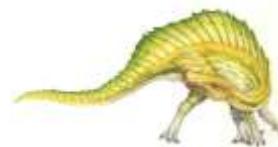
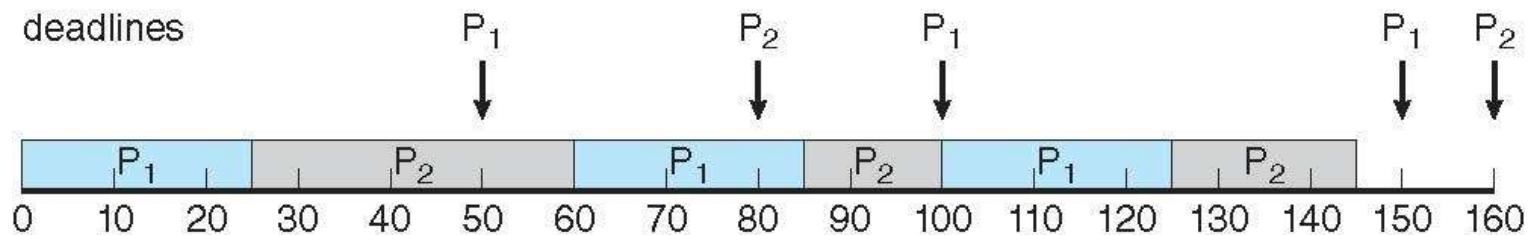




Figure 5.24

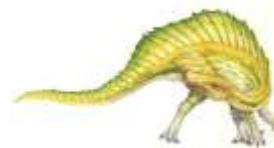
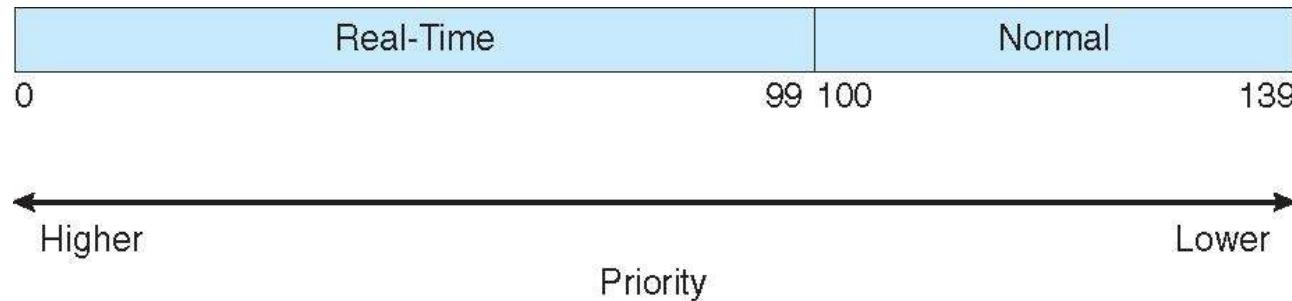




Figure 5.25

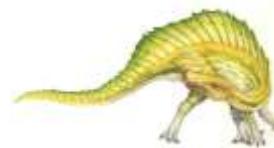
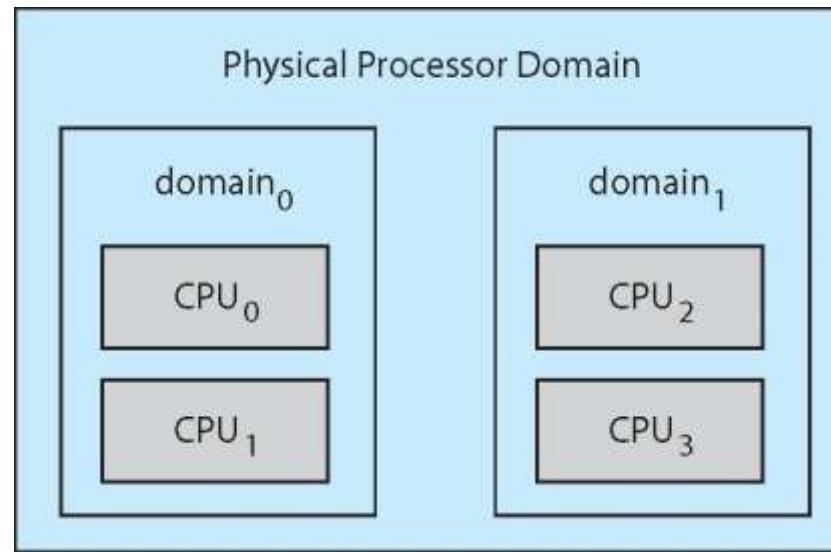




Figure 5.26

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

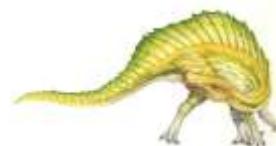




Figure 5.27

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

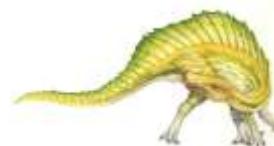




Figure 5.28

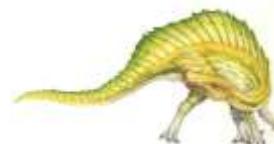
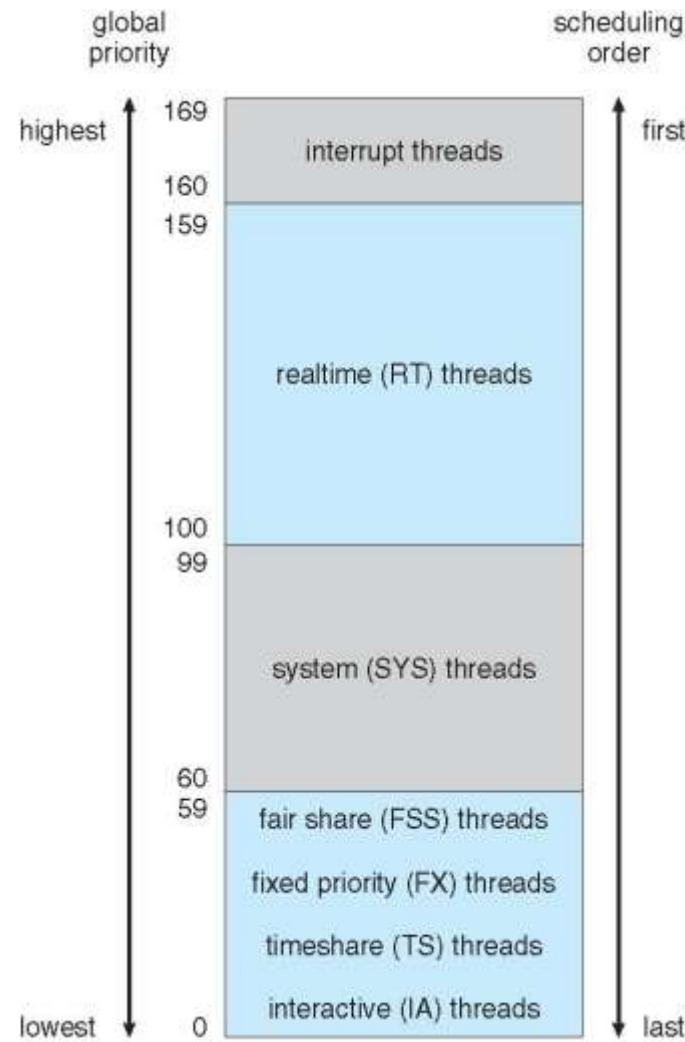




Figure 5.29

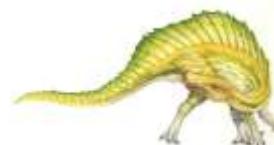
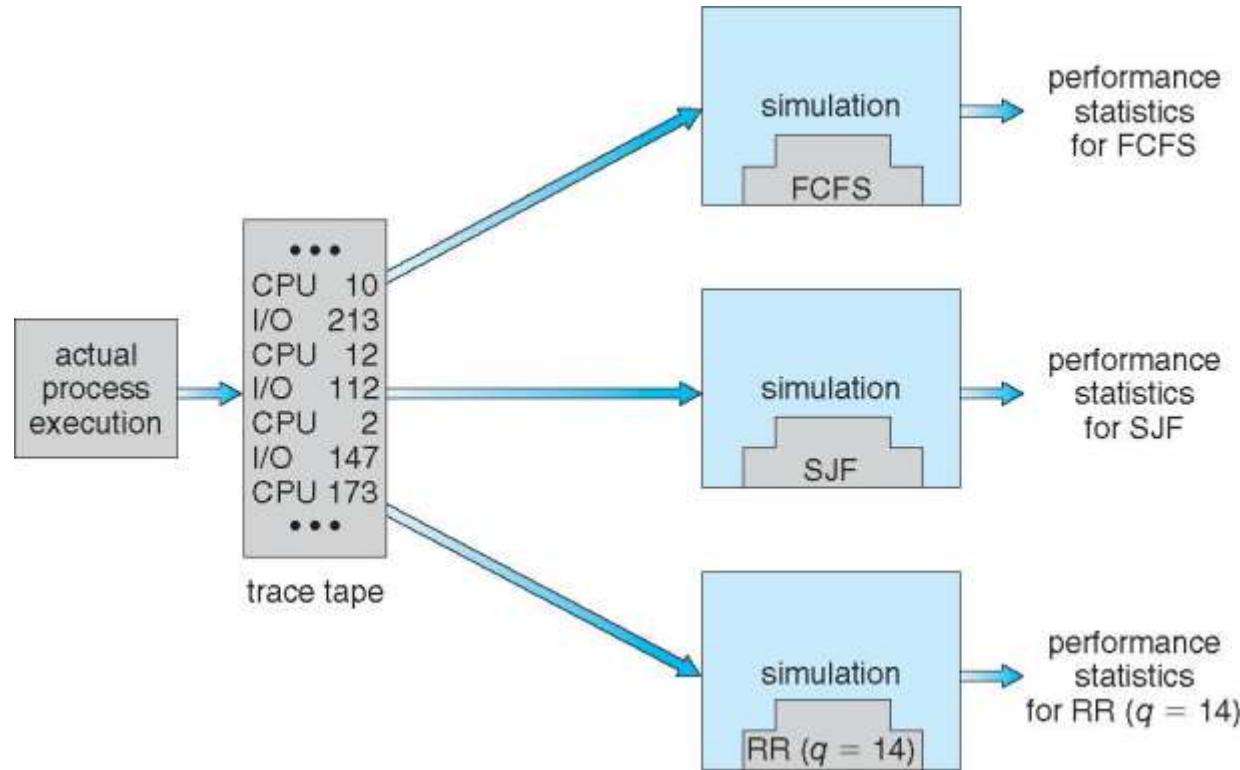




Figure Dispatch Latency

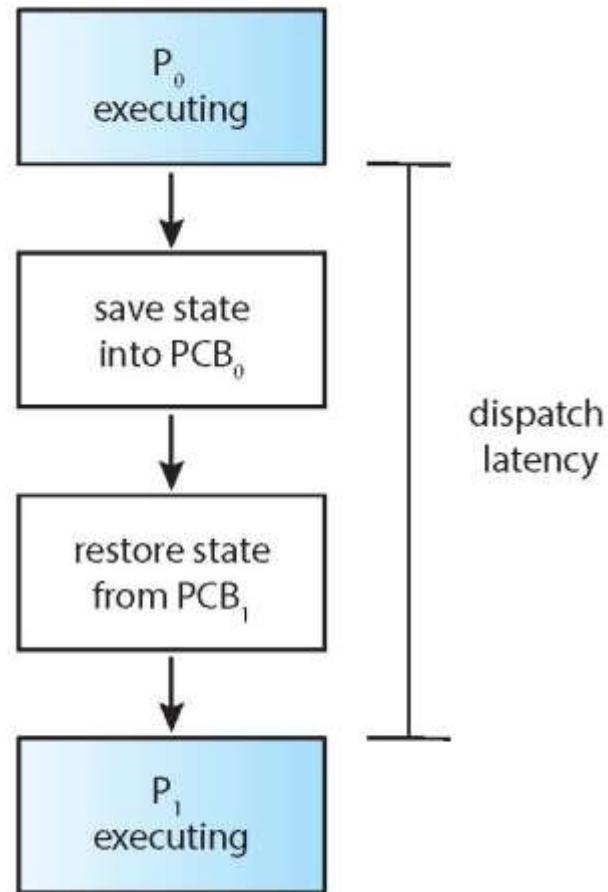




Figure in-5.01

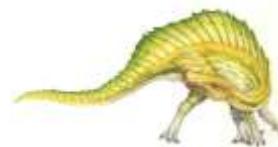
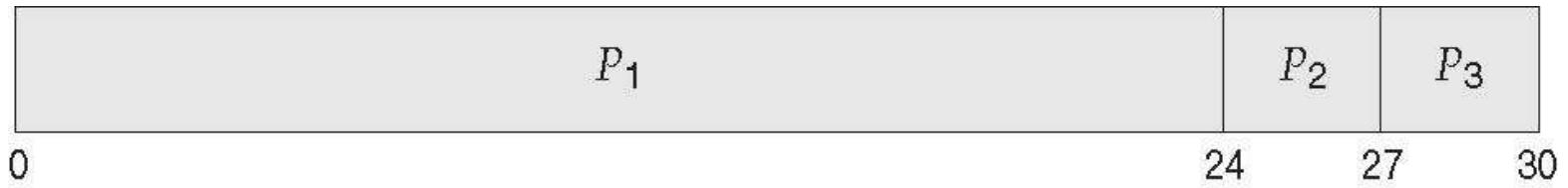




Figure in-5.02

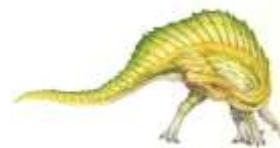
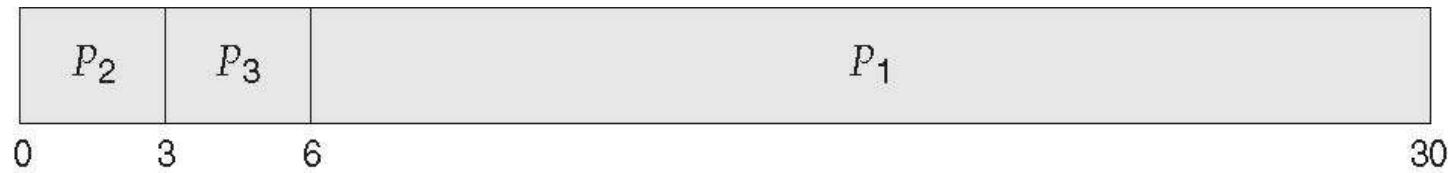




Figure in-5.03

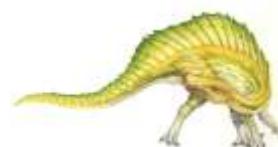
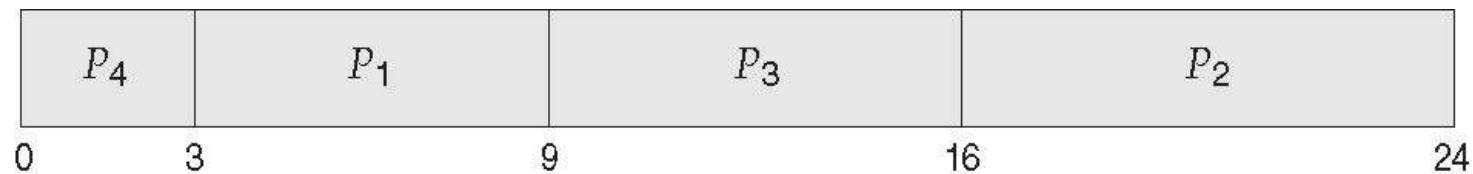




Figure in-5.04

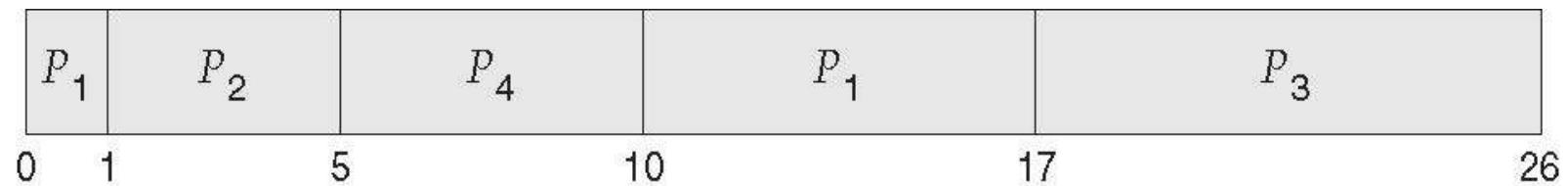




Figure in-5.05

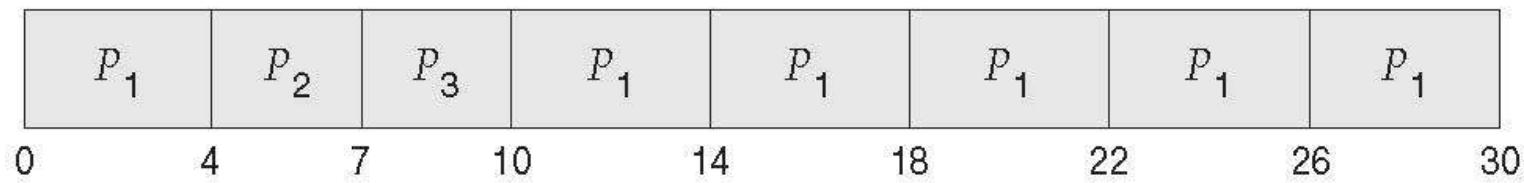




Figure in-5.06

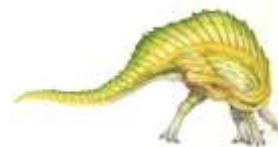




Figure in-5.07

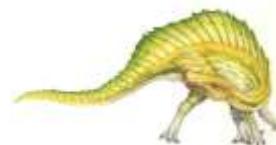
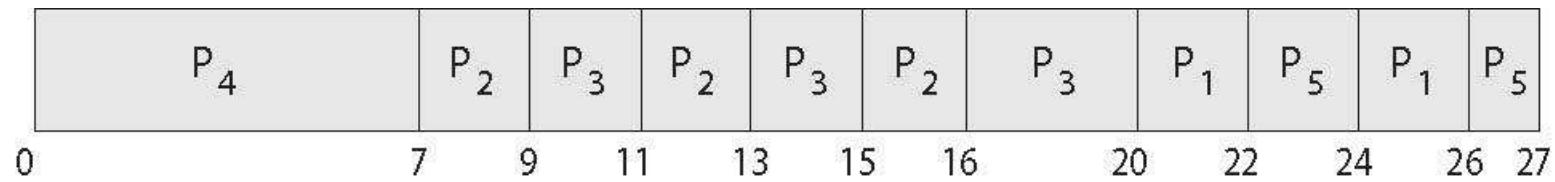




Figure in-5.08

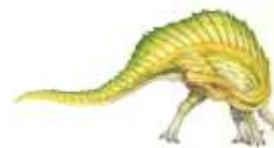
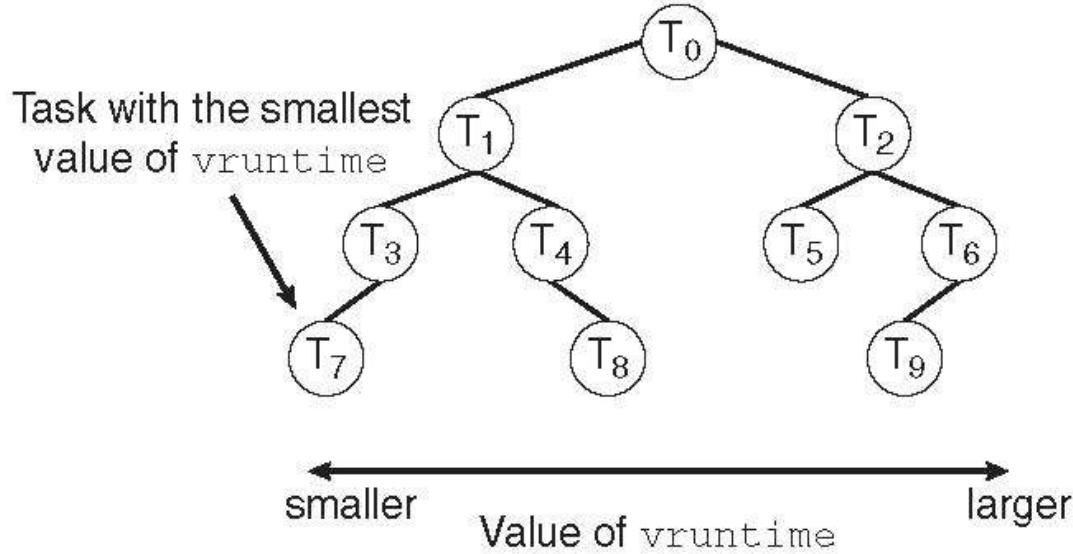




Figure in-5.09

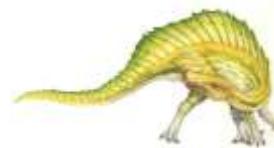
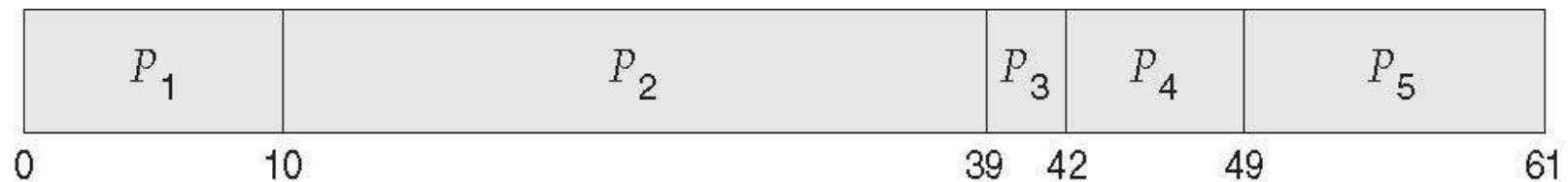




Figure in-5.10

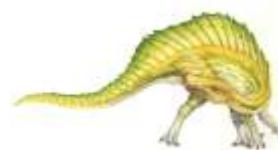
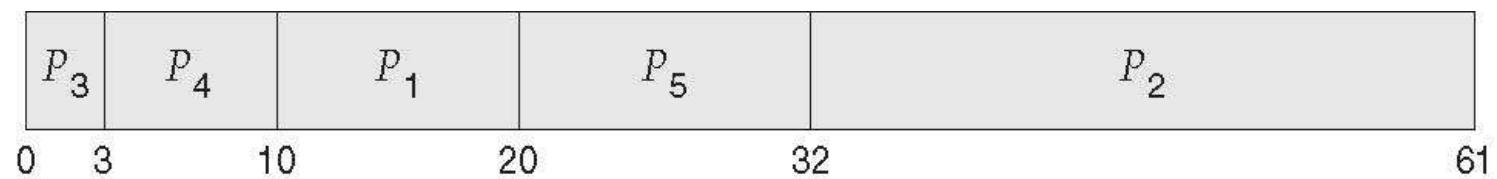




Figure in-5.11

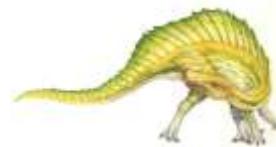
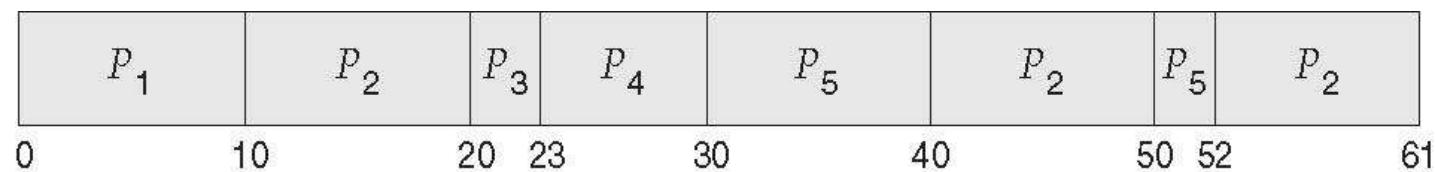
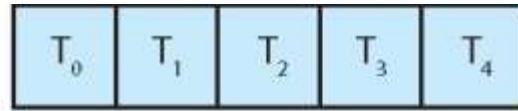


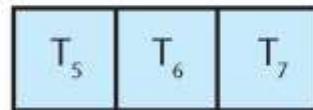


Figure Priority Queues

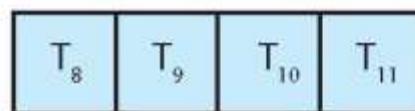
priority = 0



priority = 1



priority = 2

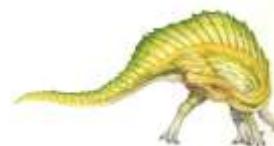


•

•

•

priority = n





CPU Scheduler

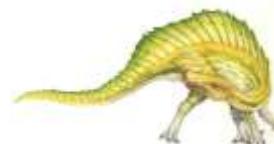
- **CPU scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
 - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
 1. Switches from running to waiting state
 2. Switches from running to ready state
 3. Switches from waiting to ready
 4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
 - Consider access to shared data
 - Consider preemption while in kernel mode
 - Consider interrupts occurring during crucial OS activities





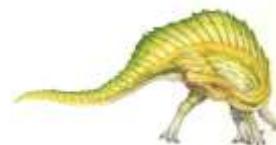
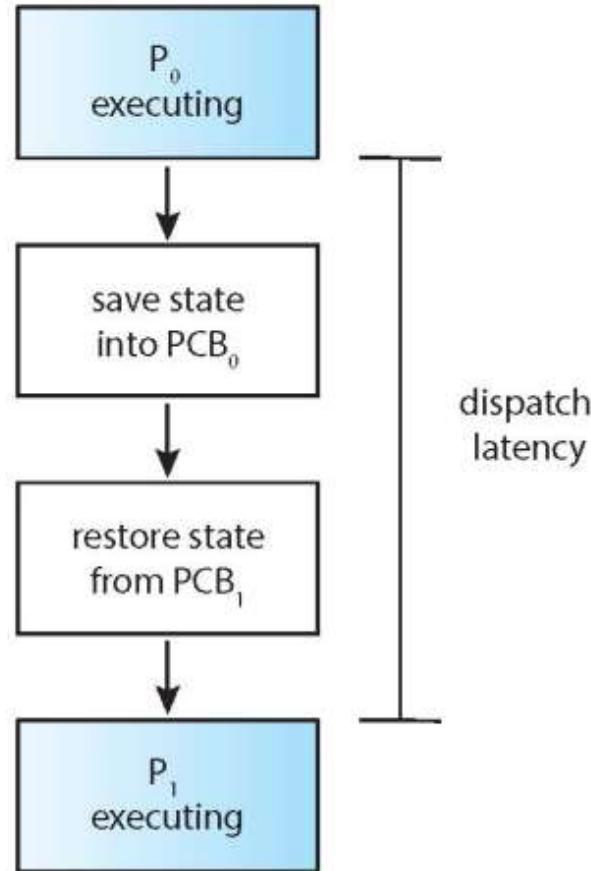
Preemptive and nonpreemptive Scheduling

- **Non-preemptive** – once a CPU is allocated to the process, the process keeps the CPU until it releases the CPU either when it terminates or it switches to the waiting state
- **Preemptive** -- a CPU can be taken away from a process at any time. Issues to consider in preemptive scheduling:
 - Consider access to shared data
 - Consider preemption while in kernel mode
 - Consider interrupts occurring during crucial OS activities





The role of the Dispatcher





First-Come, First-Served (FCFS) Scheduling

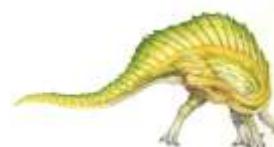
- Consider the following three processes and their burst time

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1, P_2, P_3
- We use **Gantt Chart** to illustrate a particular schedule



- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$





First-Come, First-Served (FCFS) Scheduling

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1, P_2, P_3
The Gantt Chart for the schedule is:



- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$



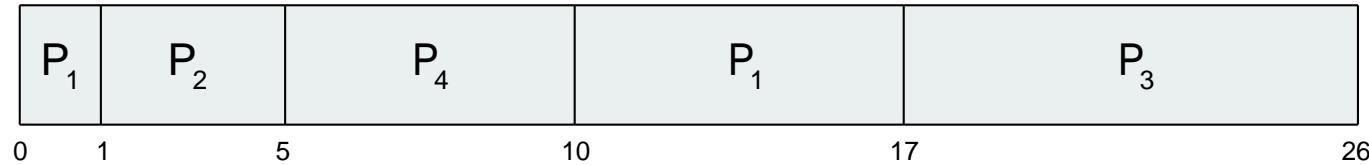


Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- Preemptive SJF Gantt Chart*



- Average waiting time = $[(10-1)+(1-1)+(17-2)+5-3]/4 = 26/4 = 6.5$ msec





Dispatcher

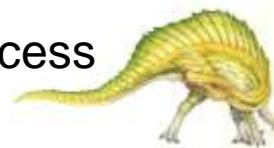
- Dispatcher module gives control of the CPU to the process selected by the CPU scheduler; this involves:
 - switching context
 - switching to user mode
 - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running





Performance of RR Algorithm

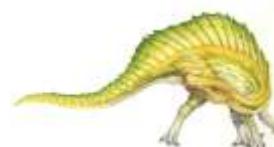
- The performance of the RR algorithm depends on the size of the time quantum. At one extreme, if the time quantum is extremely large, the \RR\ policy is the same as the \FCFS\ policy. In contrast, if the time quantum is extremely small (say, 1 millisecond), the \RR\ approach can result in a large number of context switches. Assume, for example, that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly (\Fref{fig:showing-smaller}).





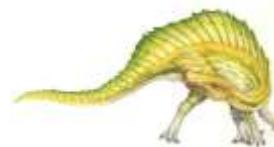
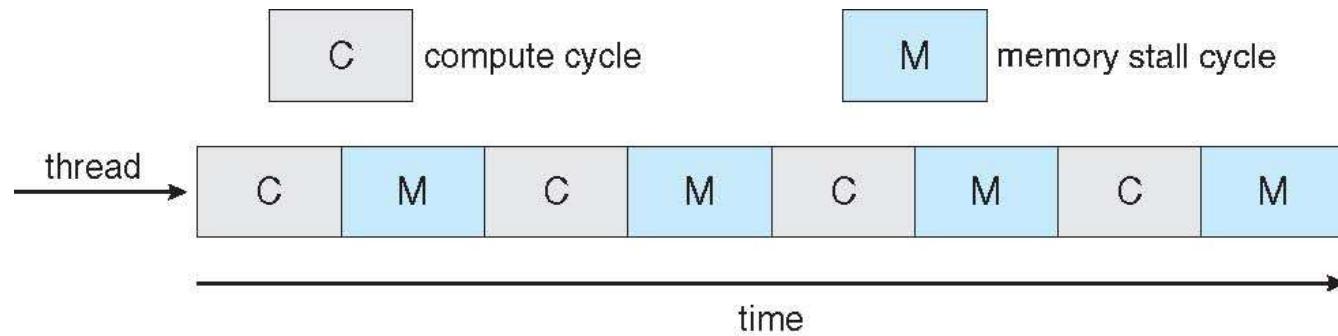
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
 - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running
 - **soft affinity**
 - **hard affinity**
 - Variations including **processor sets**

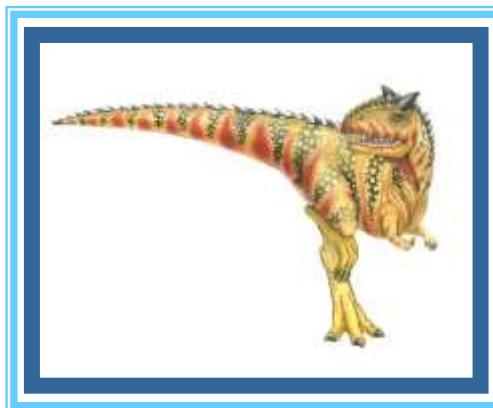




Memory Stall in Multicore Systems



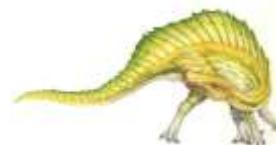
Chapter 6: Synchronization Tools





Chapter 6: Synchronization Tools

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Alternative Approaches





Objectives

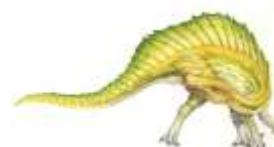
- To present the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems





Synchronization

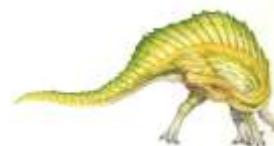
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- The synchronization mechanism is usually provided by both hardware and the operating system
- Illustration of the problem – The producer-Consumer problem, which we introduced in Chapter 3.
- Basic assumption – load and store instructions are atomic.





Producer-Consumer Problem

- The Solution we presented in chapter 3 is correct, but can only use BUFFER_SIZE - 1 elements.
- The methodology used is to allow only a single process to increment/decrement a particular shared variable.
- There is a solution that fills ***all*** the buffers, using the same methodology:
 - The producer process increments the value on the variable “in” (but not “out”) and the consumer process increments the value on the variable “out” (but not “in”)
 - The solution is more complex. Try and see if you can come up with the algorithm.





Producer-Consumer Problem (Cont.)

- Suppose that we wanted to provide a solution that fills **all** the buffers where we allow the producer and consumer processes to increment and decrement the same variable.
- We can do so by adding another integer variable -- **counter** that keeps track of the number of full buffers. Initially, **counter** is set to 0.
- The variable **counter** It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.
- Code is shown in next two slides





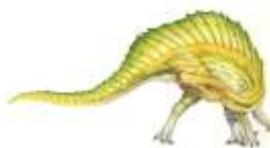
Producer Process

```
while (true) {
    /* produce an item in next produced */

    while (counter == BUFFER_SIZE) ;
        /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter = counter +1;

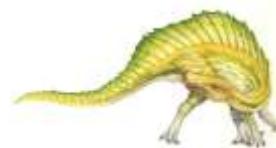
}
```





Consumer Process

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter = counter - 1;  
  
    /* consume the item in next_consumed */  
}
```





Race Condition

- `counter = counter + 1` could be implemented as

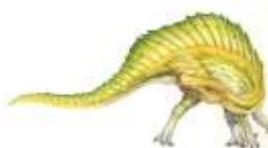
```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

- `counter = counter -1` could be implemented as

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

- Consider this execution interleaving with “count = 5” initially:

S0: producer execute <code>register1 = counter</code>	{register1 = 5}
S1: producer execute <code>register1 = register1 + 1</code>	{register1 = 6}
S2: consumer execute <code>register2 = counter</code>	{register2 = 5}
S3: consumer execute <code>register2 = register2 - 1</code>	{register2 = 4}
S4: producer execute <code>counter = register1</code>	{counter = 6 }
S5: consumer execute <code>counter = register2</code>	{counter = 4}





Race Condition (Cont.)

- How do we solve the race condition?
- We need to make sure that:

- The execution of

```
counter = counter + 1
```

is done as an “atomic” action. That is, while it is being executed, no other instruction can be executed concurrently.

- ▶ actually no other instruction can access **counter**
 - Similarly for

```
counter = counter - 1
```

- The ability to execute an instruction, or a number of instructions, atomically is crucial for being able to solve many of the synchronization problems.





Critical Section Problem

- Consider system of n processes $\{P_0, P_1, \dots P_{n-1}\}$
- Each process has **critical section** segment of code
 - Process may be changing common variables, updating table, writing file, etc
 - When one process in critical section, no other may be in its critical section
- **Critical section problem** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section** code; it then executes in the critical section; once it finishes executing in the critical section it enters the **exit section** code. The process then enters the **remainder section** code.





General structure of Process Entering the Critical Section

- General structure of process P_i

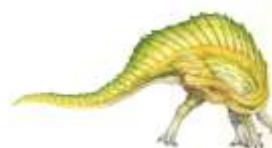
```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```





Hardware Solution

- Entry section – first action is to “disable interrupts.”
- Exit section – last action is to “enable interrupts”.
- Must be done by the OS. Why?
- Implementation issues:
 - Uniprocessor systems
 - ▶ Currently running code would execute without preemption
 - Multiprocessor systems.
 - ▶ Generally too inefficient on multiprocessor systems
 - ▶ Operating systems using this not broadly scalable
- Is this an acceptable solution?
 - This is impractical if the critical section code is taking a long time to execute.





Software Solution for Process P_i

- Keep a variable “turn” to indicate which process is next

```
do {  
    while (turn == j);  
        critical section  
    turn = j;  
        remainder section  
} while (true);
```

- Algorithm is correct. Only one process at a time in the critical section. But:
- Results in “busy waiting”.
- What if $turn = j$; P_i wants to enter the critical section and P_j does not want to enter the critical section?





Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
 - Assume that each process executes at a nonzero speed
 - No assumption concerning **relative speed** of the n processes





Software Solution: Peterson's Algorithm

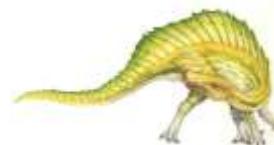
- Good algorithmic software solution
- Two process solution
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
 - **int turn;**
 - **Boolean flag[2]**
- The variable **turn** indicates whose turn it is to enter the critical section
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i] = true** implies that process P_i is ready!





Algorithm for Process P_i

```
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
        critical section  
        flag[i] = false;  
    remainder section  
} while (true);
```





Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
 1. Mutual exclusion is preserved

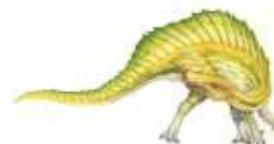
P_i enters CS only if:
either `flag[j] = false` or `turn = i`
 2. Progress requirement is satisfied
 3. Bounded-waiting requirement is met
- What about a solution to $N > 2$ processes





Busy Waiting

- All the software solutions we presented employ “busy waiting”
 - A process interested in entering the critical-section is stuck in a loop asking continuously
“can I get into the critical-section”
- Busy waiting is a pure waste of CPU cycles

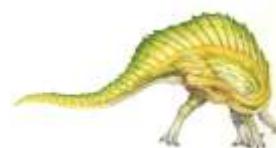




Solution to Critical-section Problem Using Locks

- Many systems provide hardware support for implementing the critical section code.
- All solutions are based on idea of **locking**
 - Two processes can not have a lock simultaneously.
- Code:

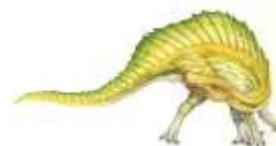
```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (TRUE);
```





Synchronization Hardware

- Modern machines provide special atomic hardware instructions to implement locks
 - **Atomic** = non-interruptible
- Two types instructions:
 - Test memory word and set value
 - Swap contents of two memory words





test_and_set Instruction

■ Definition:

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

■ Properties:

- Executed atomically
- Returns the original value of passed parameter
- Set the new value of passed parameter to “TRUE”.



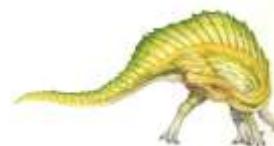


Solution using test_and_set()

- Shared Boolean variable **lock**, initialized to FALSE
- Each process, wishing to execute critical-section code:

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
        /* critical section */  
    lock = false;  
        /* remainder section */  
} while (true);
```

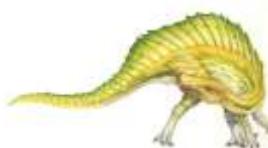
- What about bounded waiting?
- Solution results in busy waiting.





Bounded-waiting Mutual Exclusion with test_and_set

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
} while (true);
```





compare_and_swap Instruction

■ Definition:

```
int compare_and_swap(int *value, int expected, int new_value) {  
    int temp = *value;  
  
    if (*value == expected)  
        *value = new_value;  
    return temp;  
}
```

■ Properties:

- Executed atomically
- Returns the original value of passed parameter “value”
- Set “value” to “new_value ” but only if “value” == “expected”.
That is, the swap takes place only under this condition.



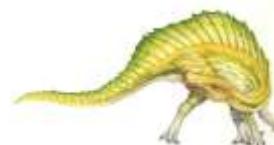


Solution using compare_and_swap

- Shared integer “lock” initialized to 0;
- Solution:

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */
    /* critical section */
    lock = 0;
    /* remainder section */
} while (true);
```

- What about bounded waiting?
- Solution results in busy waiting.

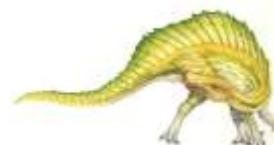




Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest tools is the **Mutex lock**, which has a Boolean variable “available” associated with it to indicate if the lock is available or not.
- Two operations available to access a Mutex Lock:
 - `acquire() {
 while (!available)
 ; /* busy wait */
 available = false;
}

● release() {
 available = true;
}`





Mutex Locks (Cont.)

- Calls to `acquire()` and `release()` are atomic
 - Usually implemented via hardware atomic instructions
- Usage:

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```

- Solution requires **busy waiting**
 - This lock is therefore called a **spinlock**





Semaphores

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations
 - **wait()** and **signal()**
 - ▶ Originally called **P()** and **V()**
- Definition of the **wait()** operation

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S = S - 1;  
}
```

- Definition of the **signal() operation**

```
signal(S) {  
    S = S + 1;  
}
```





Semaphore Usage

Can solve various synchronization problems

- A solution to the CS problem.
 - Create a semaphore “synch” initialized to 1

```
    wait(synch)
```

CS

```
    signal(synch);
```

- Consider P_1 and P_2 that require code segment S_1 to happen before code segment S_2
 - Create a semaphore “synch” initialized to 0

P1 :

```
    S1;
```

```
    signal(synch);
```

P2 :

```
    wait(synch);
```

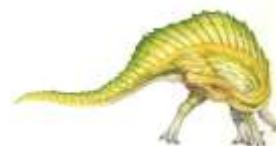
```
    S2;
```





Types of Semaphores

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
 - Same as a **mutex lock**
- Can implement a counting semaphore **S** as a binary semaphore





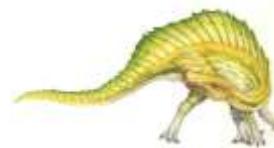
Counting Semaphores Example

- Allow at most two process to execute in the CS.
- Create a semaphore “**synch**” initialized to 2

```
wait(synch)
```

CS

```
signal(synch);
```





Semaphore Implementation

- Must guarantee that no two processes can execute the **wait()** and **signal()** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **wait** and **signal** code are placed in the critical section
- This implementation is based on **busy waiting** in critical section implementation (that is, the code for **wait()** and **signal()**)
 - But implementation code is short
 - Little busy waiting if critical section rarely occupied
- Can we implement semaphores with no busy waiting?



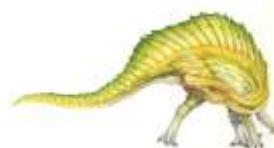


Semaphore Implementation with no Busy Waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
 - value (of type integer)
 - pointer to next record in the list

```
typedef struct{  
    int value;  
    struct process *list;  
} semaphore;
```

- Two operations:
 - **block ()** – place the process invoking the operation on the appropriate waiting queue
 - **wakeup (P)** – remove one of processes in the waiting queue and place it in the ready queue





Implementation with no Busy waiting (Cont.)

- `wait(semaphore *S) {`
 `S->value--;`
 `if (S->value < 0) {`
 `add this process to S->list;`
 `block();`
 `}`
`}`

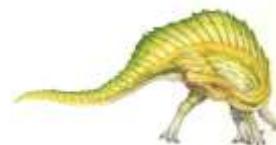
- `signal(semaphore *S) {`
 `S->value++;`
 `if (S->value <= 0) {`
 `remove a process P from S->list;`
 `wakeup(P);`
 `}`
`}`





Implementation with no Busy waiting (Cont.)

- Does the implementation ensure the “progress” requirement?
- Does implementation ensure the “bounded waiting” requirement?





Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
<code>wait(S) ;</code>	<code>wait(Q) ;</code>
<code>wait(Q) ;</code>	<code>wait(S) ;</code>
...	...
<code>signal(S) ;</code>	<code>signal(Q) ;</code>
<code>signal(Q) ;</code>	<code>signal(S) ;</code>

- **Starvation – indefinite blocking**
 - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
 - Solved via **priority-inheritance protocol**





Problems with Semaphores

- Incorrect use of semaphore operations:
 - signal (mutex) wait (mutex)
 - wait (mutex) ... wait (mutex)
 - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.
- Solution – create high-level programming language constructs





Monitors

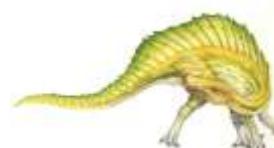
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { ... }

    procedure Pn (...) {.....}

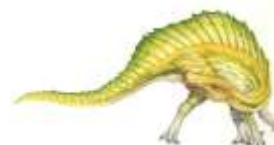
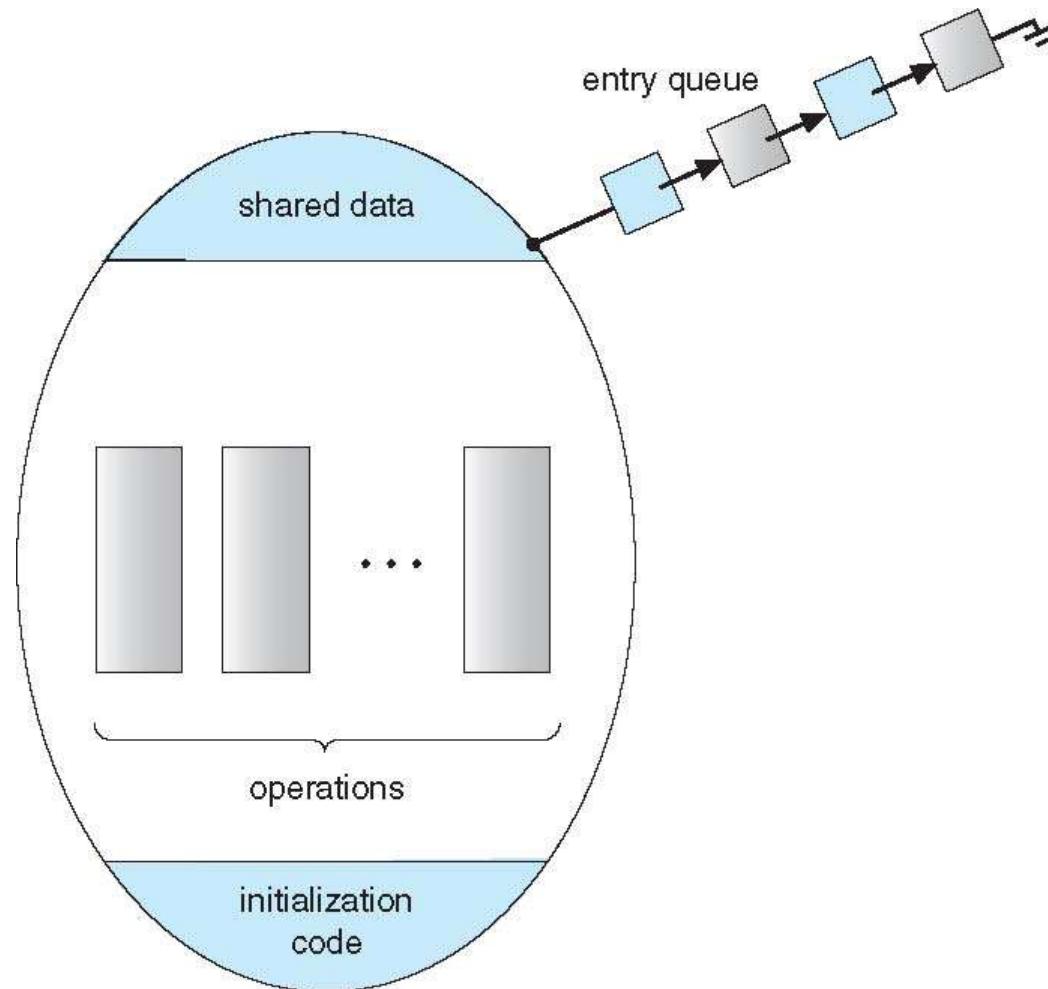
    Initialization code (...) { ... }
}
```

- Mutual exclusion is guaranteed by the compiler.





Schematic view of a Monitor





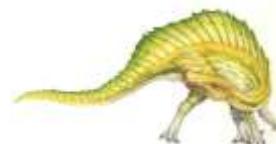
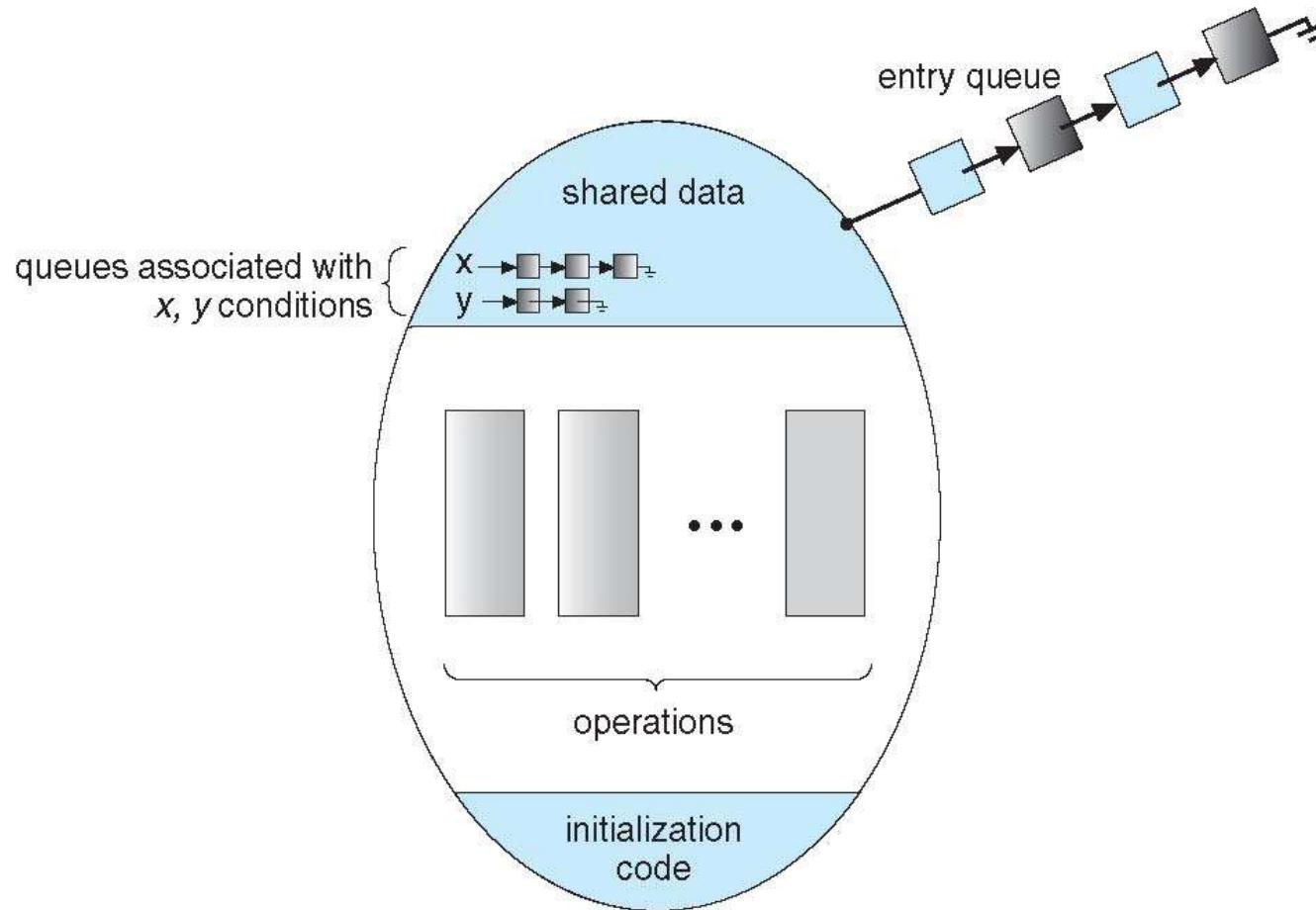
Condition Variables

- Need mechanism to allow a process wait within a monitor
- Provide condition variables.
- A condition variable (say **x**) can be accessed only via two operations:
 - **x.wait()** – a process that invokes the operation is suspended until another process invoked **x.signal()**
 - **x.signal()** – resumes one of processes (if any) that invoked **x.wait()**
 - ▶ If no process is suspended on variable **x** , then it has no effect on the variable





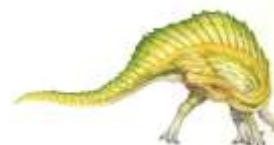
Monitor with Condition Variables





Condition Variables Choices

- If process P invokes **`x.signal()`**, and process Q is suspended in **`x.wait()`**, what should happen next?
 - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include:
 - **Signal and wait** – P either waits until Q leaves the monitor or it waits for another condition
 - **Signal and continue** – Q either waits until P leaves the monitor or it waits for another condition





Condition Variables Choices (Cont.)

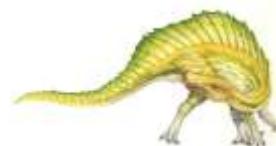
- There are reasonable arguments in favor of adopting either option.
 - Since P was already executing in the monitor, the signal-and-continue method seems more reasonable.
 - However, if we allow P to continue, by the time Q is resumed, the logical condition for which Q was waiting may no longer hold.
- A compromise between these two choices was adopted in the language Concurrent Pascal. When P executes the signal operation, it immediately leaves the monitor. Hence, Q is immediately resumed.





Languages Supporting the Monitor Concept

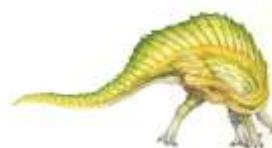
- Many programming languages have incorporated the idea of the monitor as described in this section, including Java and C#.
- Other languages such as Erlang provide concurrency support using a similar mechanism.





Monitor Implementation

- For each monitor, a semaphore **mutex** is provided.
- A process must execute **wait (mutex)** before entering the monitor and must execute **signal (mutex)** after leaving the monitor. This is ensured by the compiler.
- We use the “signal and wait” mechanism to handle the signal operation.
- Since a signaling process must wait until the resumed process either leaves or it waits, an additional semaphore, **next**, is used.
- The signaling processes can use **next** to suspend themselves.
- An integer variable **next_count** is provided to count the number of processes suspended on **next**





Monitor Implementation (Cont.)

- Variables

```
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next_count = 0;
```

- Each procedure *F* will be replaced by

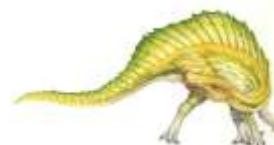
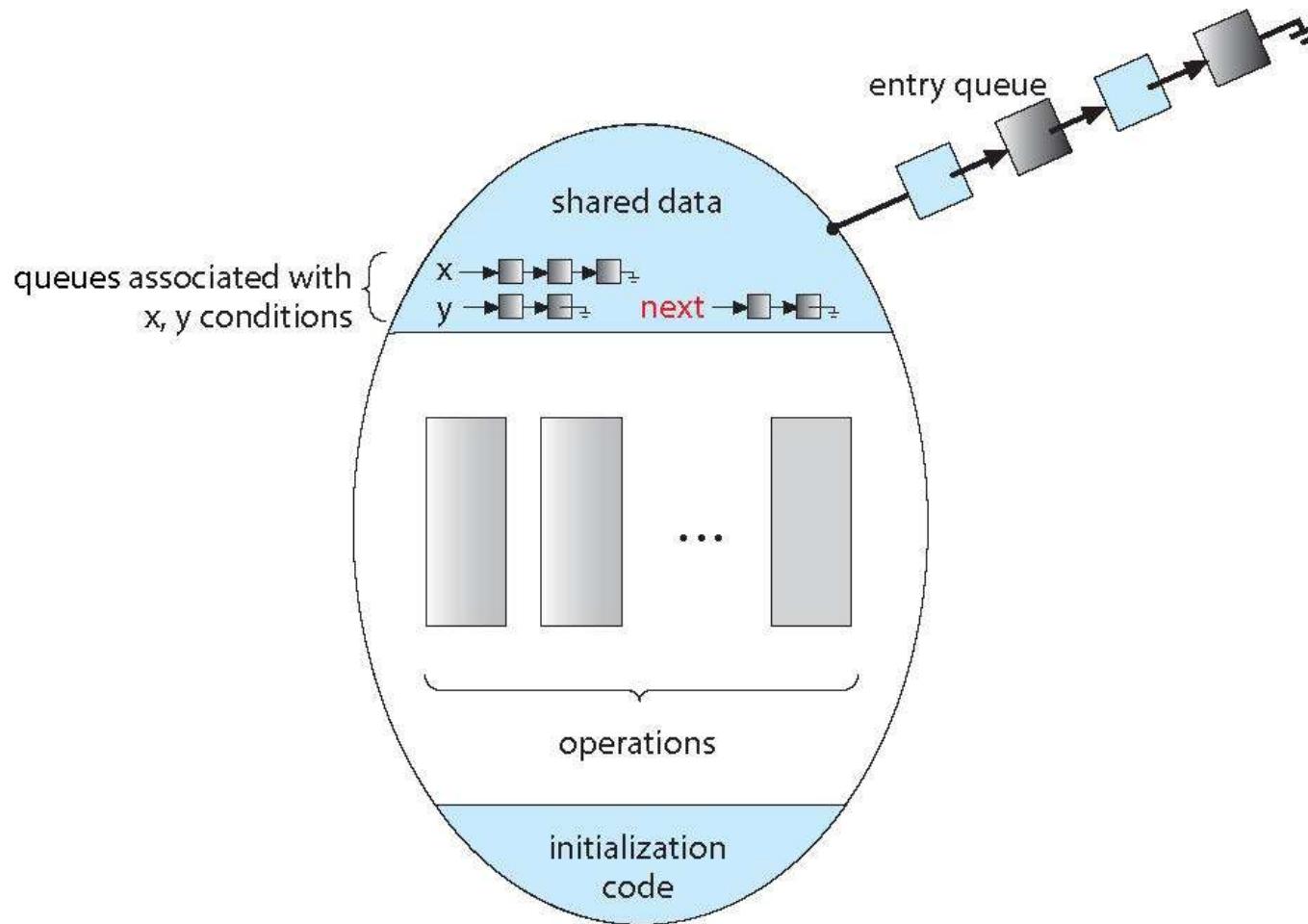
```
wait(mutex);
...
body of F;
...
if (next_count > 0)
    signal(next)
else
    signal(mutex);
```

- Mutual exclusion within a monitor is ensured





Monitor with Next Semaphore





Condition Variables Implementation

- For each condition variable x , we have:

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

- The operation $x.wait$ can be implemented as:

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

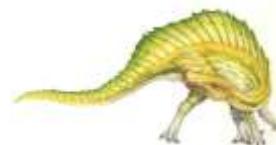




Condition Variables Implementation (Cont.)

- The operation `x.signal` can be implemented as:

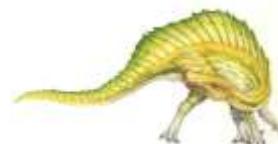
```
if (x_count > 0) {  
    next_count++;  
    signal(x_sem);  
    wait(next);  
    next_count--;  
}
```





Resuming Processes within a Monitor

- If several processes are queued on condition **x**, and **x.signal()** is executed, which one should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form **x.wait(c)**
 - Where **c** is **priority number**
 - Process with lowest number (low number → highest priority) is scheduled next
- Some languages provide a mechanism to find out the PID of the executing process.
 - In C we have **getpid()**, which returns the PID of the calling process





Resource Allocator Monitor Example

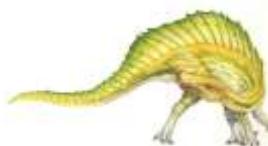
- A monitor to allocate a single resource among competing processes
- Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource. The monitor allocates the resource to the process that has the shortest time-allocation request.

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;

    void acquire(int time) {
        if (busy)
            x.wait(time)
        busy = true;
    }

    void release () {
        busy = false;
        x.signal();
    }

    busy= false;
}
```



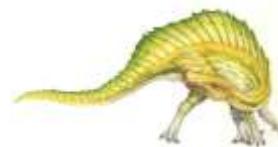


Resource Allocator Monitor Example (Cont.)

- A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);  
    . . .  
    access the resource;  
    . . .  
R.release();
```

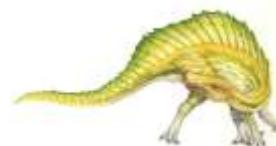
where R is an instance of type ResourceAllocator



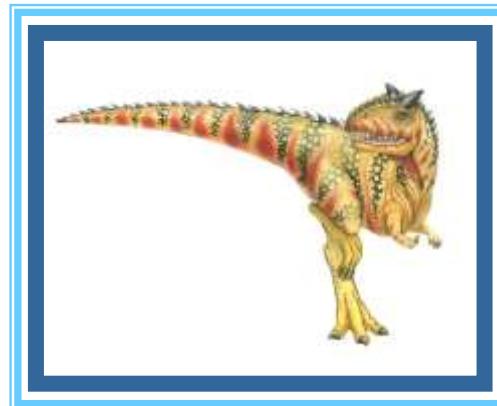


Observation the Resource Allocator Example

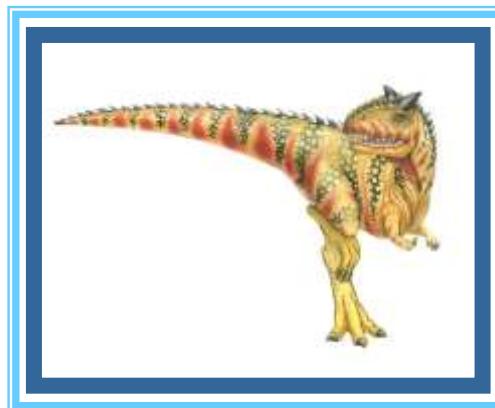
- Incorrect use of the operations:
 - `R.release R.acquire(t)`
 - `R.acquire(t) R.acquire(t)`
 - Omitting of acquire and or release (or both)
- Solution exist but not covered in this course



End of Chapter 6



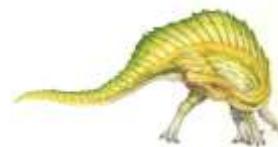
Chapter 7: Synchronization Examples





Chapter 7: Synchronization Examples

- Classic Problems of Synchronization
- Synchronization Examples
- Alternative Approaches





Objectives

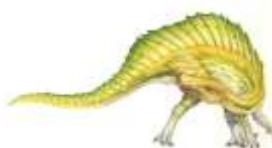
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems





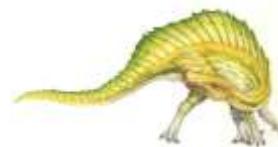
Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
 - Bounded-Buffer Problem
 - Readers and Writers Problem
 - Dining-Philosophers Problem
- We will present solutions using:
 - Semaphores.
 - Monitors
 - Various operating systems





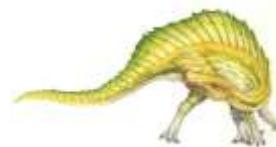
Semaphore Solutions





Bounded-Buffer Problem

- n buffers, each can hold one item
- Semaphore **mutex** initialized to the value 1
- Semaphore **full** initialized to the value 0
- Semaphore **empty** initialized to the value n





Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {  
    ...  
    /* produce an item in next_produced */  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    /* add next produced to the buffer */  
    ...  
    signal(mutex);  
    signal(full);  
} while (true);
```

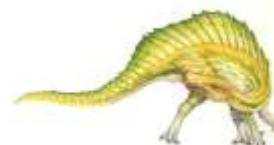




Bounded Buffer Problem (Cont.)

- The structure of the consumer process

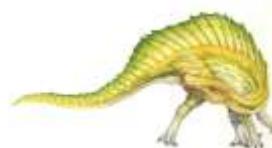
```
Do {  
    wait(full);  
    wait(mutex);  
  
    ...  
    /* remove an item from buffer to next_consumed */  
  
    ...  
    signal(mutex);  
    signal(empty);  
  
    ...  
    /* consume the item in next consumed */  
  
    ...  
} while (true);
```





Readers-Writers Problem

- A data set is shared among a number of concurrent processes
 - Readers – only read the data set; they do **not** perform any updates
 - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
 - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered
 - all involve some form of priorities
- Shared Data
 - Semaphore **rw_mutex** initialized to 1
 - Semaphore **mutex** initialized to 1
 - Integer **read_count** initialized to 0

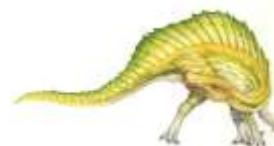




Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {  
    wait(rw_mutex);  
  
    ...  
    /* writing is performed */  
  
    ...  
    signal(rw_mutex);  
} while (true);
```





Readers-Writers Problem (Cont.)

- The structure of a reader process

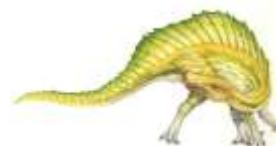
```
do {  
    wait(mutex);  
    read_count++;  
    if (read_count == 1)  
        wait(rw_mutex);  
    signal(mutex);  
  
    ...  
    /* reading is performed */  
  
    ...  
    wait(mutex);  
    read_count--;  
    if (read_count == 0)  
        signal(rw_mutex);  
    signal(mutex);  
} while (true);
```





Readers-Writers Problem Variations

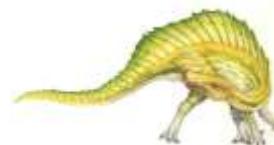
- **First** variation – no reader kept waiting unless writer has permission to use shared object
- **Second** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks





Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating
- They do not interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
 - Need both to eat, then release both when done
- In the case of 5 philosophers, the shared data:
 - Bowl of rice (data set)
 - Semaphore **chopstick [5]** initialized to 1

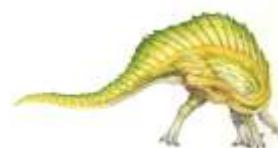




Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {  
    wait (chopstick[i] );  
    wait (chopStick[ (i + 1) % 5] );  
  
        // eat  
  
    signal (chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
  
        // think  
  
} while (TRUE);
```



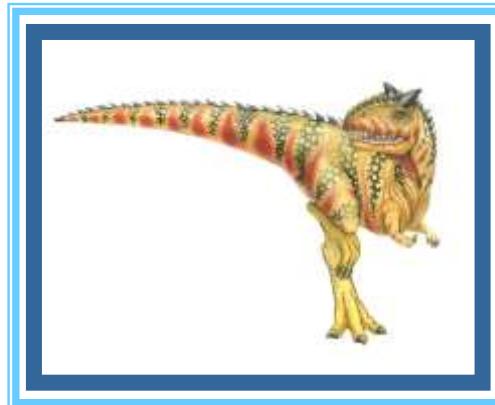


Dining-Philosophers Problem Algorithm (Cont.)

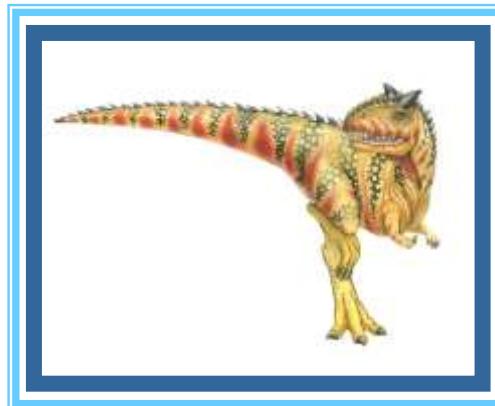
- This solution guarantees that no two neighbors are eating simultaneously.
- Possibility of a deadlock. Suppose that all five philosophers become hungry at the same time and each grabs the left chopstick.
- Solution:
 - Allow at most 4 philosophers to be sitting simultaneously at the table.
 - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section).
 - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.



End of Chapter 7



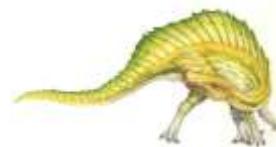
Chapter 8: Deadlocks





Chapter 8: Deadlocks

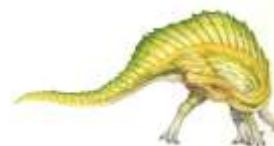
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock





Chapter Objectives

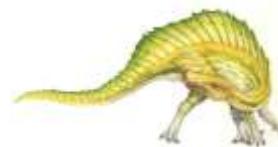
- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system





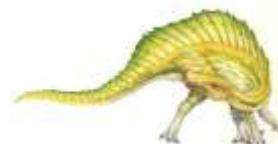
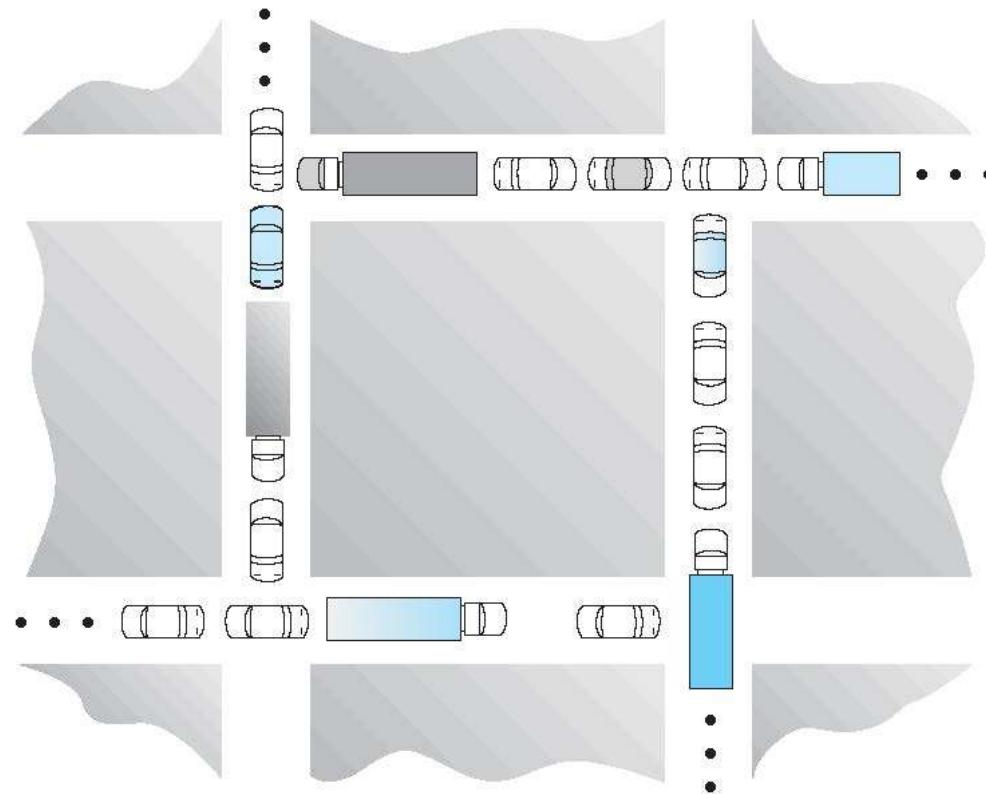
Deadlock Example – One Way Bridge

- Draw a one way bridge
- Can a “far away” can know that it is involved in a deadlock





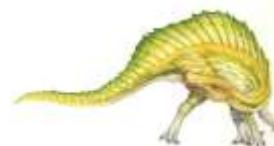
Deadlock Example – Traffic Gridlock





System Model

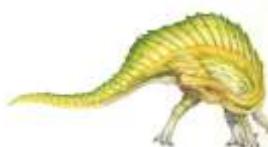
- System consists of resources
- Resource types R_1, R_2, \dots, R_m
CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
 - **request**
 - **use**
 - **release**





Deadlock Example with Mutex locks

- Two mutex locks are created in the following code
 - `pthread_mutex_t first_mutex;`
 - `pthread_mutex_t second_mutex;`
- The two mutex locks are initialized in the following code
 - `pthread_mutex_init (&first_mutex, NULL);`
 - `pthread_mutex_init(&second_mutex, NULL);`
- Two threads-- `thread_one` and `thread_two` are created, and both these threads have access to both mutex locks.

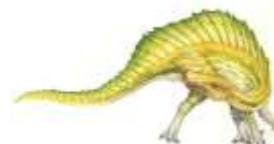




Deadlock Example with Mutex locks (Cont.)

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}

/* thread two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
```

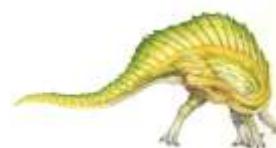




Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0 .

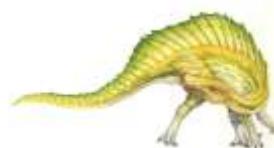




Resource-Allocation Graph

A set of vertices V and a set of edges E .

- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$



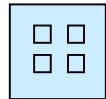


Resource-Allocation Graph (Cont.)

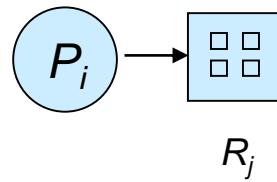
- Process



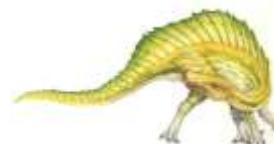
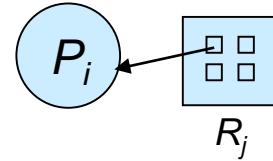
- Resource Type with 4 instances



- P_i requests instance of R_j

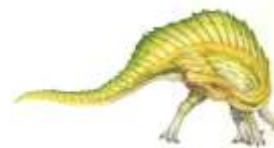
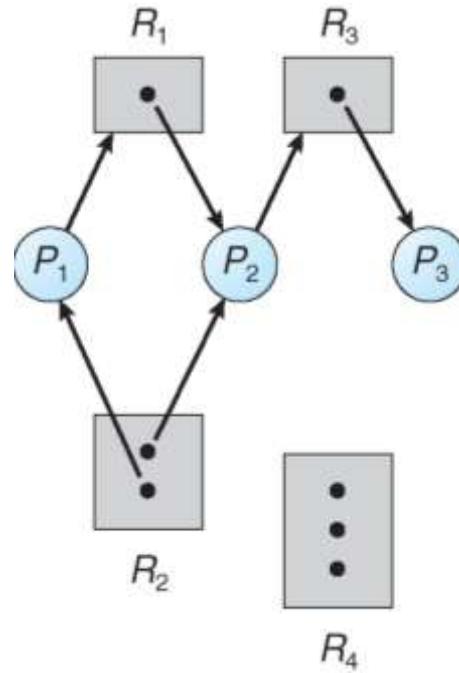


- P_i is holding an instance of R_j





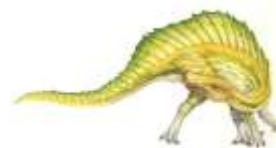
Example of a Resource Allocation Graph





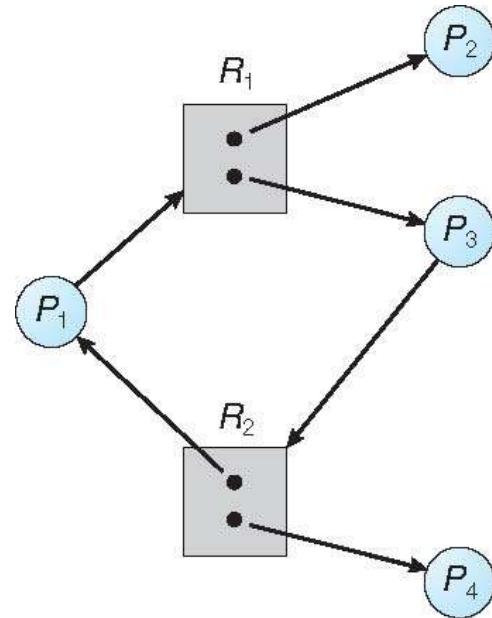
Basic Facts

- If graph contains no cycles \Rightarrow no deadlock
- If graph contains a cycle \Rightarrow
 - If only one instance per resource type, then deadlock exist
 - If several instances per resource type, then possibility of deadlock





Resource Allocation Graph With a Cycle

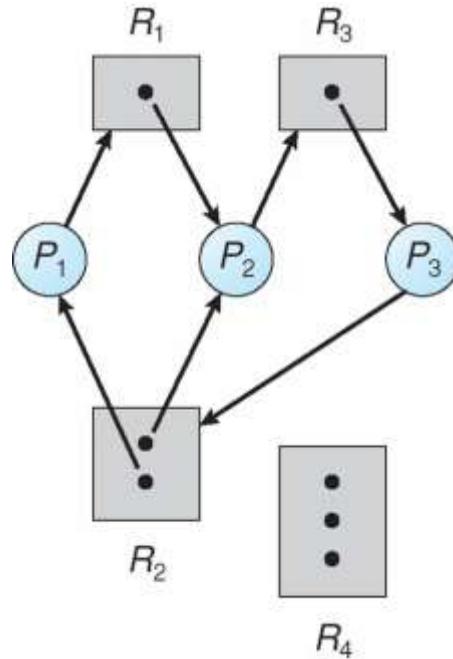


Is there a deadlock?

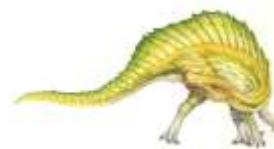




Resource Allocation Graph With a Cycle



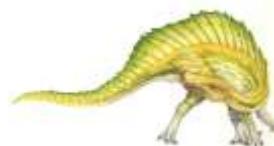
Is there a deadlock?





Methods for Handling Deadlocks

- Ensure that the system will **never** enter a deadlock state:
 - Deadlock prevention
 - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX





Deadlock Prevention

- Ensure that at least one of the necessary condition for deadlocks does not hold. Can be accomplished restraining the ways request can be made
 - **Mutual Exclusion** – Must hold for non-sharable resources that can be accessed simultaneously by various processes. Therefore cannot be used for prevention.
 - **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
 - ▶ Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
 - ▶ Low resource utilization; starvation possible





Deadlock Prevention (Cont.)

- **No Preemption** – not practical for most systems
 - If a process A that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held by A are released
 - Preempted resources are added to the list of resources for which the process is waiting
 - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration. Can be used in practice.





Deadlock Avoidance

- Ensure that the system will **never** enter a deadlock state
- Requires that the system have some additional **a priori** information available on possible resource requests.
 - Simplest and most useful model requires that each process declare the **maximum number** of resources of each type that it may need
 - The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
 - Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes





Safe State

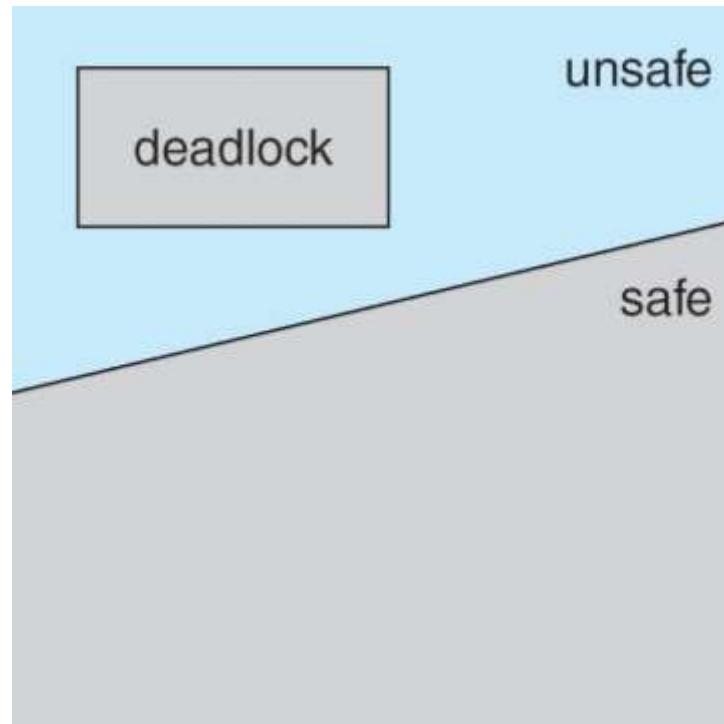
- System is in **safe state** if there exists a sequence $\langle P_1, P_2, \dots, P_n \rangle$ of ALL the processes in the systems such that for each P_i the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j , with $j < i$
- That is:
 - If P_i resource needs are not immediately available, then P_i can wait until all processes P_j ($j < i$) have finished executing.
 - When they have finished executing they release all their resources and then P_i can obtain the needed resources, execute, return allocated resources, and terminate
 - When P_i terminates, P_{i+1} can obtain its needed resources, and so on





Basic Facts

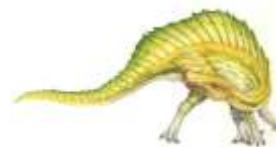
- If a system is in safe state \Rightarrow no deadlocks
- If a system is in unsafe state \Rightarrow possibility of deadlock





Deadlock Avoidance Algorithms

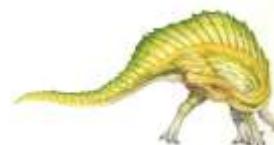
- Deadlock avoidance \Rightarrow ensure that a system will never enter an unsafe state.
- Single instance of a resource type
 - Use a variant of the resource-allocation graph
- Multiple instances of a resource type
 - Use the banker's algorithm





Resource-Allocation Graph Scheme

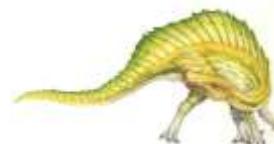
- Single instance of a resource type
- Each process must *a priori* claim maximum resource use
- Use a variant of the resource-allocation graph with claim edges.
- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j **may** request resource R_j ; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system
- A cycle in the graph implies that the system is in unsafe state





Resource-Allocation Graph Scheme

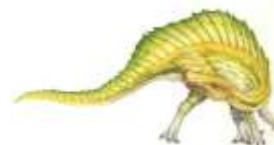
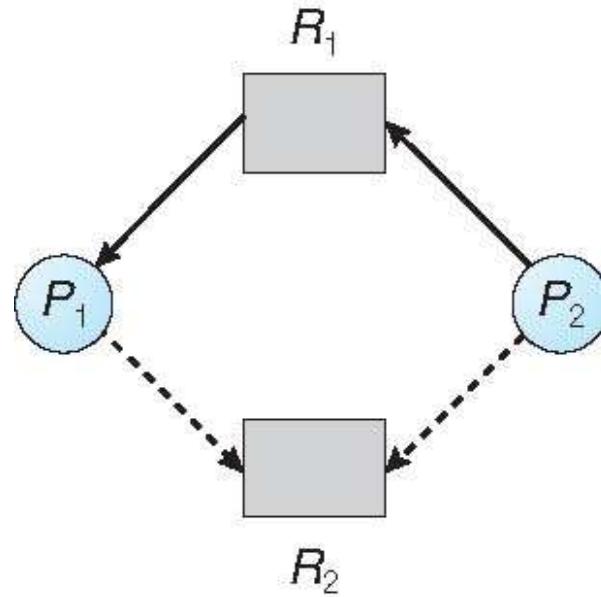
- Single instance of a resource type
- Each process must *a priori* claim maximum resource use
- Use a variant of the resource-allocation graph with claim edges.
- **Claim edge** $P_i \dashrightarrow R_j$ indicated that process P_j **may** request resource R_j ; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system
- A cycle in the graph implies that the system is in unsafe state





Resource-Allocation Graph with claim edges

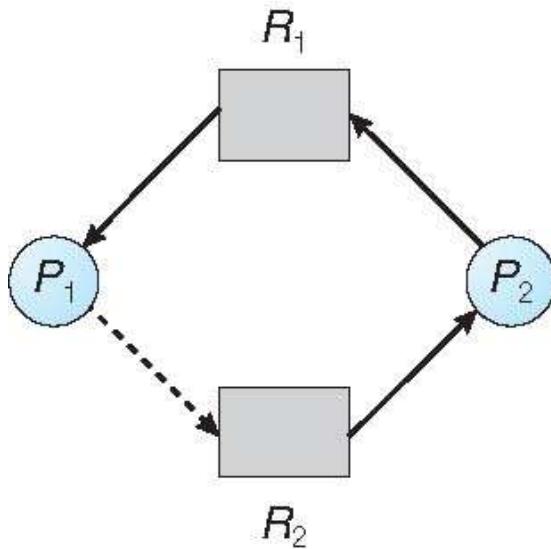
- P1 is holding resource R1 and has a claim on R2
- P2 is requesting R1 and has a claim on R2
- No cycle. So system is in a safe state.





Example of a Resource Allocation Graph

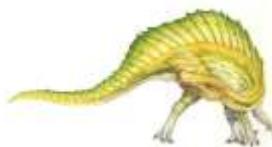
- The graph of slide 8.25 with a claim edge from P2 to R2 is changing to an assignment edge.
- There is a cycle in the graph → unsafe state.
- Is there a deadlock?





Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph
- Otherwise, the process must wait





Banker's Algorithm

- Multiple instances of a resource type
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time.
- Think of an interest-free bank where:
 - A customer establishes a line of credit.
 - Borrows money in chunks that together never exceed the total line of credit.
 - Once it reaches the maximum, the customer must pay back in a finite amount of time.





Data Structures for the Banker' s Algorithm

Let n = number of processes, and m = number of resources types.

- **Available:** Vector of length m .
 - If $\text{available}[j] = k$, then there are k instances of resource type R_j available
- **Max:** $n \times m$ matrix.
 - If $\text{Max}[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation:** $n \times m$ matrix.
 - If $\text{Allocation}[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need:** $n \times m$ matrix.
 - If $\text{Need}[i,j] = k$, then P_i may need at most k more instances of R_j to complete its task.
 - ▶ $\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$





Safety Algorithm

1. Let **Work** and **Finish** be vectors of length m and n , respectively.
Initialize:

Work = Available

Finish [i] = false for $i = 0, 1, \dots, n - 1$

2. Find an i such that both:

(a) **Finish [i] = false**

(b) **Need_i ≤ Work**

If no such i exists, go to step 4

3. **Work = Work + Allocation_i**,

Finish[i] = true

go to step 2

4. If **Finish [i] == true** for all i , then the system is in a safe state.
Otherwise, in an unsafe state.

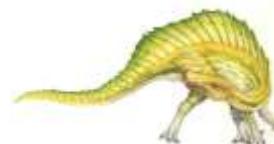




Example of Safety Algorithm

- 5 processes -- $P_0 \dots P_4$;
- 3 resource types:
 A (10 instances), B (5 instances), and C (7 instances)
- Snapshot at time T_0 :

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			





Example of Safety Algorithm (Cont.)

- The content of the matrix **Need** is defined to be **Max – Allocation**

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	3 3 2
P_1	2 0 0	1 2 2	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria





Resource-Request Algorithm for Process P_i

Let $\text{Request}_i[\dots]$ be the request vector for process P_i .

$\text{Request}_i[j] = k$. Process P_i wants k instances of resource type R_j

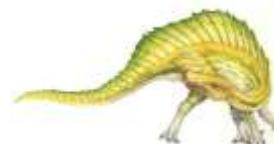
1. If $\text{Request}_i \leq \text{Need}_i$, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\text{Request}_i \leq \text{Available}$, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$\text{Available} = \text{Available} - \text{Request}_i$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i$$

$$\text{Need}_i = \text{Need}_i - \text{Request}_i$$

- If safe \Rightarrow the resources are allocated to P_i
- If unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is restored





Example of Banker' s Algorithm

- 5 processes P_0 through P_4 ;

3 resource types:

A (10 instances), B (5 instances), and C (7 instances)

- Snapshot at time T_0 :

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

- We have shown that the system is in a safe state





Example: P_1 Request (1,0,2)

- Check that Request \leq Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true)
- State of system after resources allocated to P_1

	<u>Allocation</u>			<u>Need</u>			<u>Available</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

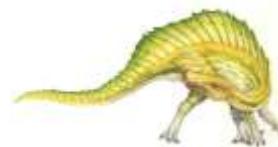
- Executing safety algorithm shows that sequence $< P_1, P_3, P_4, P_0, P_2 >$ satisfies safety requirement
- Given the above state -- can request for (3,3,0) by P_4 be granted?
- Given the above state -- can request for (0,2,0) by P_0 be granted?





Methods for Handling Deadlocks: Detection

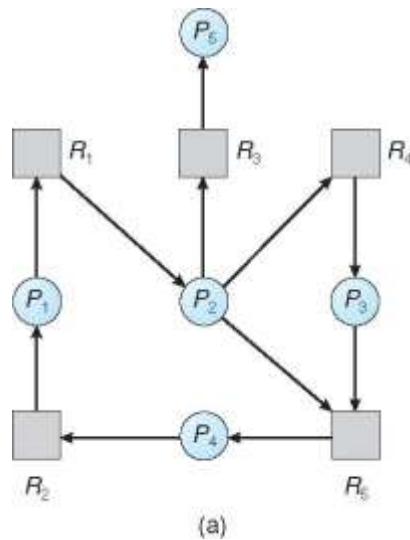
- Allow system to enter deadlock state
- Detection algorithm
 - Single instance of a resource type
 - Multiple instances of a resource type.
- Recovery scheme



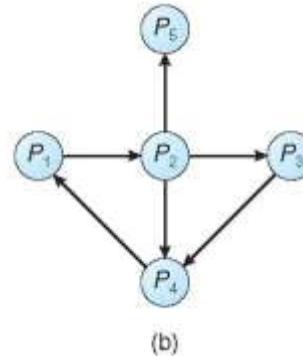


Single Instance of Each Resource Type

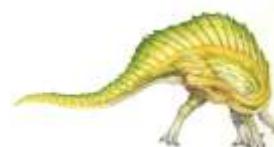
- Maintain a **wait-for** graph
 - Nodes are processes
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Converting a resource-allocation graph to a wait-for graph.



Resource-Allocation Graph



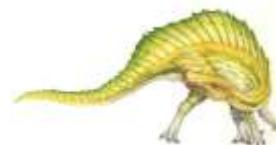
Corresponding wait-for graph





Detection Algorithm for Single Instance

- Maintain a **wait-for** graph
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

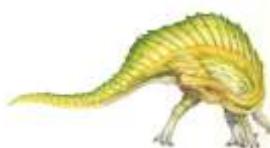




Several Instances of a Resource Type

Let n = number of processes, and m = number of resources types.

- **Available:** Vector of length m . If $\text{available}[j] = k$, then there are k instances of resource type R_j available
- **Allocation:** $n \times m$ matrix. If $\text{Allocation}[i,j] = k$, then P_i is currently allocated k instances of R_j
- **Request:** $n \times m$ matrix that indicates the current request of each process. If $\text{Request}[i,j] = k$, then process P_i is requesting k additional instances of resource type R_j .





Detection Algorithm

Let **Work** and **Finish** be vectors of length m and n , respectively Initialize:

1. Initialization

(a) **Work = Available**

(b) For $i = 1, 2, \dots, n$, if $\text{Allocation}_i \neq 0$, then

Finish[i] = false; otherwise, **Finish[i] = true**

2. Find an index i such that both:

(a) **Finish[i] == false**

(b) **Request_i ≤ Work**

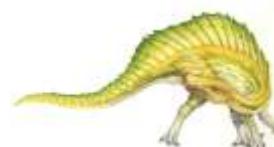
If no such i exists, go to step 4

3. **Work = Work + Allocation_i**,

Finish[i] = true

go to step 2

4. If **Finish[i] == false**, for some i , $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if **Finish[i] == false**, then P_i is deadlocked



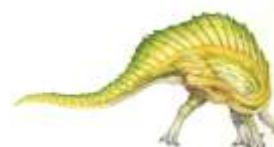


Example of Detection Algorithm

- Five processes P_0 through P_4 ; three resource types A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time T_0 :

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

- Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = \text{true}$ for all i





Example of Detection Algorithm (Cont.)

- P_2 requests one additional instance of type **C**

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- State of system?
 - Can reclaim resources held by process P_0 , but insufficient resources to fulfill other processes requests
 - Deadlock exists, consisting of processes P_1 , P_2 , P_3 , and P_4





Detection-Algorithm Usage

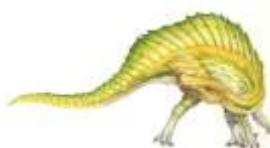
- If a deadlock is detected we must abort (rollback) some of the processes involved in the deadlock (see next slide)
- Need to decide when, and how often, to invoke the deadlock detection algorithm, which depends on:
 - How often a deadlock is likely to occur?
 - How many processes will need to be rolled back?
 - ▶ one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.





Recovery from Deadlock: Process Termination

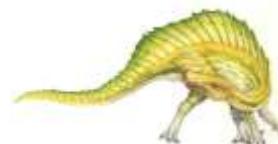
- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
 - Priority of the process
 - How long process has computed, and how much longer to completion
 - Resources the process has used
 - Resources process needs to complete
 - How many processes will need to be terminated
 - Is process interactive or batch?



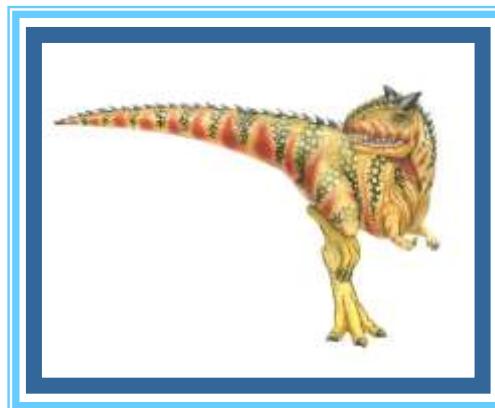


Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor



End of Chapter 8





Deadlock Example with Lock Ordering

- Two bank transactions 1 and 2 execute concurrently.
 - Transaction 1 transfers \$25 from account A to account B
 - Transaction 2 transfers \$50 from account B to account A
- Program

```
void transaction(Account from,
                  Account to, double amount)
{
    mutex lock1, lock2;
    lock1 = get_lock(from);
    lock2 = get_lock(to);
    acquire(lock1);
    acquire(lock2);
    withdraw(from, amount);
    deposit(to, amount);
    release(lock2);
    release(lock1);
}
```

