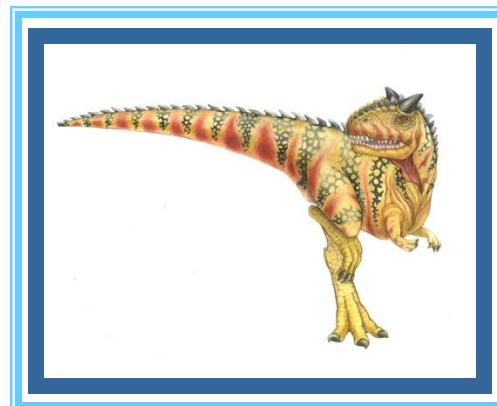


Week 1

Introduction to OS and Computer System Structures





Teaching Team

Name of Course/Module	Operating Systems
Course Code	CSN6214
Name of Course Coordinator(s)	Sharaf Horani

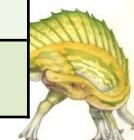
Name	Room Number	Email Address	Tel. No.
Mr Sharaf Horani	BR4007	Sharaf.horani@mmu.edu.my	03-83125232
Dr Ng Hu	BR3010	nghu@mmu.edu.my	03-83125411
Dr Tan Saw Chin	BR2010	sctan1@mmu.edu.my	03-83125346
Dr Syed Qamrun Nisa	BR2016	qamrunnisa@mmu.edu.my	03-83125162
Ms Wan Nor Al-Ashekin Wan Husin	BR3028	wanashekin@mmu.edu.my	-
Mr Wong Kim Ming	Part time	wkmwong@outlook.com	-





Teaching Plan (1)

WEEK	TOPICS/ACTIVITIES	ASSESSMENT METHOD & WEIGHTAGE	IMPORTANT NOTES
1	Introduction to OS	L	
2	OS structures	L	
3	Processes and Threads	T + L	
4	Process Scheduling	T Group assignment release (20%)	
5	Process Scheduling	T + L Quiz 1 (10%)	Convo 26th week (6-8 th)
6	Process Synchronization	T + L	Thursday 11th December Sultan's Birthday
7	Deadlocks	T + L	
8	Memory Management	T Quiz 2 (10%)	Thursday 25 th December (Xmas)
9	Virtual Memory and Memory Management	T + L Midterm (20%)	Thursday 1 st Jan (New Year) Midterm on Saturday the 3 rd of January 2026
10	Virtual Memory	T + L	
11	File Systems	T + L	
12	File Systems Implementation	T + L	
13	Storage Management	T Quiz 3 (10%)	
14	OS case study (BSD)	L Group assignment submission (8 th February – before 5:00 PM)	Monday 2 nd February (replace Sunday – Thaipusam)
15	Assignment Interviews + publishing coursework marks		





Teaching Plan (2)

- Replacement classes will be announced later by individual lecturers and tutors not later than end of week 2.
- T : Theoretical and discussion questions (1 or the full 2 Hours)
- L : Practical exercises – Linux and C programming (1 or the full 2 Hours)
- T + L: Mixture of Theoretical questions and Practical Exercises.
- Quizzes 20%: Select the highest 2 quizzes out of the three.
 - No replacement for the quizzes.
 - Missing a quiz will be given a 0.
 - During lecture (multiple forms paper-based quiz).
 - Quiz 1: Covers Lecture contents of week 1 until the end of week 4
 - Quiz 2: Covers Lecture contents of week 5 until the end of week 7
 - Quiz 3: Covers Lecture contents of week 8 until the end of week 10
- Midterm 20%: Covers all the topics of week 1 to the end of week 7 (Saturday 3rd January 2026).
- Assignment 20%: to be released during week 4 and must be submitted by the end of week 14 before 5:00 PM on the 8th of February 2025.
- Final Exam 40%: Covers all the topics from week 1 until the end of week 13 (Lecture material + Practical).





Textbook and References

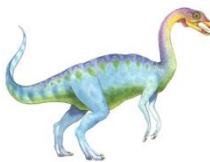
Textbook

- Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, "Operating System Concepts", 10 Edition, John Wiley & Sons, 2018.

References

- William Stallings, "Operating Systems: Internals and Design Principles", 9th Edition, Prentice Hall, 2018 .
- Andrew S. Tanenbaum, "Modern Operating Systems", 4th Edition, Prentice Hall, 2015.
- Andrew S. Tanenbaum, "Operating Systems Design and Implementation", 4th Edition, Prentice Hall, 2006.





Chapter 1: Introduction

- What Operating Systems Do
- Computer-System Organization
- Computer-System Architecture
- Operating-System Operations
- Resource Management
- Security and Protection
- Virtualization
- Distributed Systems
- Kernel Data Structures
- Computing Environments
- Free/Libre and Open-Source Operating Systems





Objectives

- Describe the general organization of a computer system and the role of interrupts
- Describe the components in a modern, multiprocessor computer system
- Illustrate the transition from user mode to kernel mode
- Discuss how operating systems are used in various computing environments
- Provide examples of free and open-source operating systems

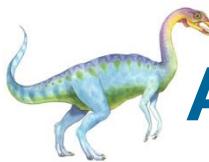




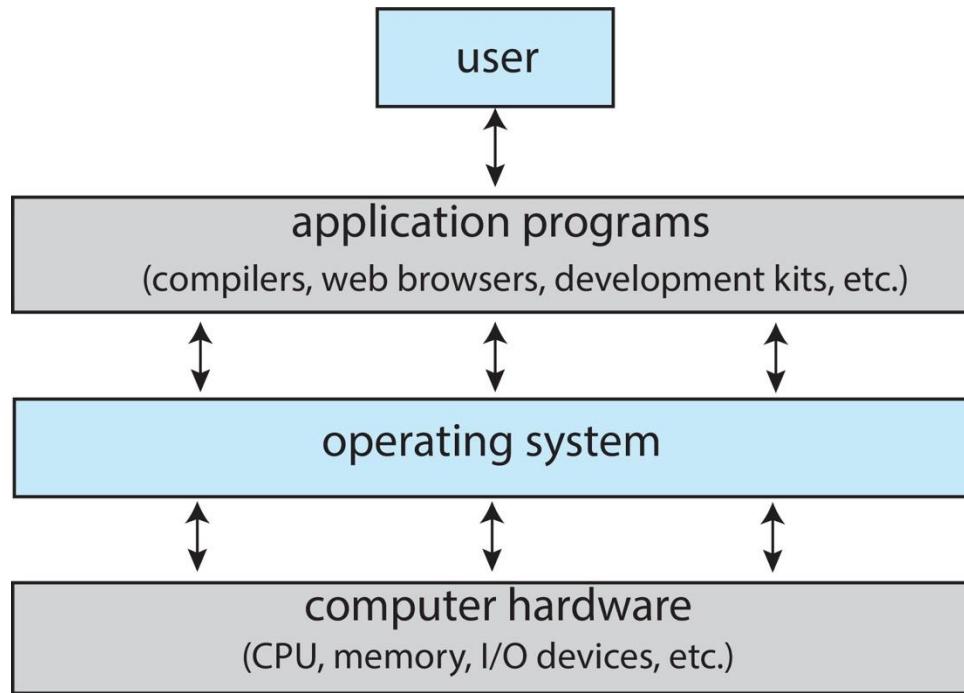
Computer System Structure

- 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





Abstract View of Components of Computer





What Operating Systems Do

- Depends on the point of view
- Users want convenience, **ease of use** and **good performance**
 - Don't care about **resource utilization**
- But shared computer such as **mainframe** or **minicomputer** must keep all users happy
 - Operating system is a **resource allocator** and **control program** making efficient use of HW and managing execution of user programs
- Users of dedicated systems such as **workstations** have dedicated resources but frequently use shared resources from **servers**
- Mobile devices like smartphones and tablets are resource poor, optimized for usability and battery life
 - Mobile user interfaces such as touch screens, voice recognition
- Some computers have little or no user interface, such as embedded computers in devices and automobiles
 - Run primarily without user intervention





Defining Operating Systems

- Term OS covers many roles
 - Because of myriad designs and uses of OSes
 - Present in toasters through ships, spacecraft, game machines, TVs and industrial control systems
 - Born when fixed use computers for military became more general purpose and needed resource management and program control





Operating System Definition (Cont.)

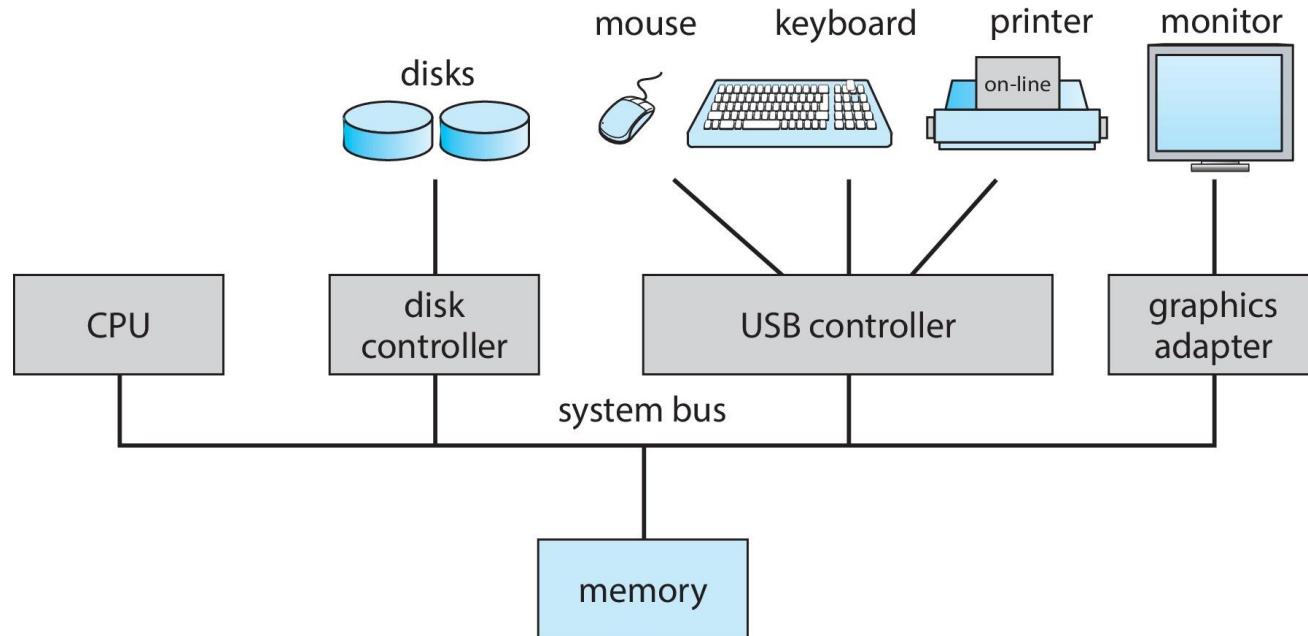
- No universally accepted definition
- “Everything a vendor ships when you order an operating system” is a good approximation
 - But varies wildly
- “The one program running at all times on the computer” is the **kernel**, part of the operating system
- Everything else is either
 - a **system program** (ships with the operating system, but not part of the kernel) , or
 - an **application program**, all programs not associated with the operating system
- Today’s OSes for general purpose and mobile computing also include **middleware** – a set of software frameworks that provide addition services to application developers such as databases, multimedia, graphics





Computer System Organization

- Computer-system operation
 - One or more CPUs, device controllers connect through common **bus** providing access to shared memory
 - Concurrent execution of CPUs and devices competing for memory cycles





Computer-System Operation

- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- Each device controller type has an operating system **device driver** to manage it
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an **interrupt**





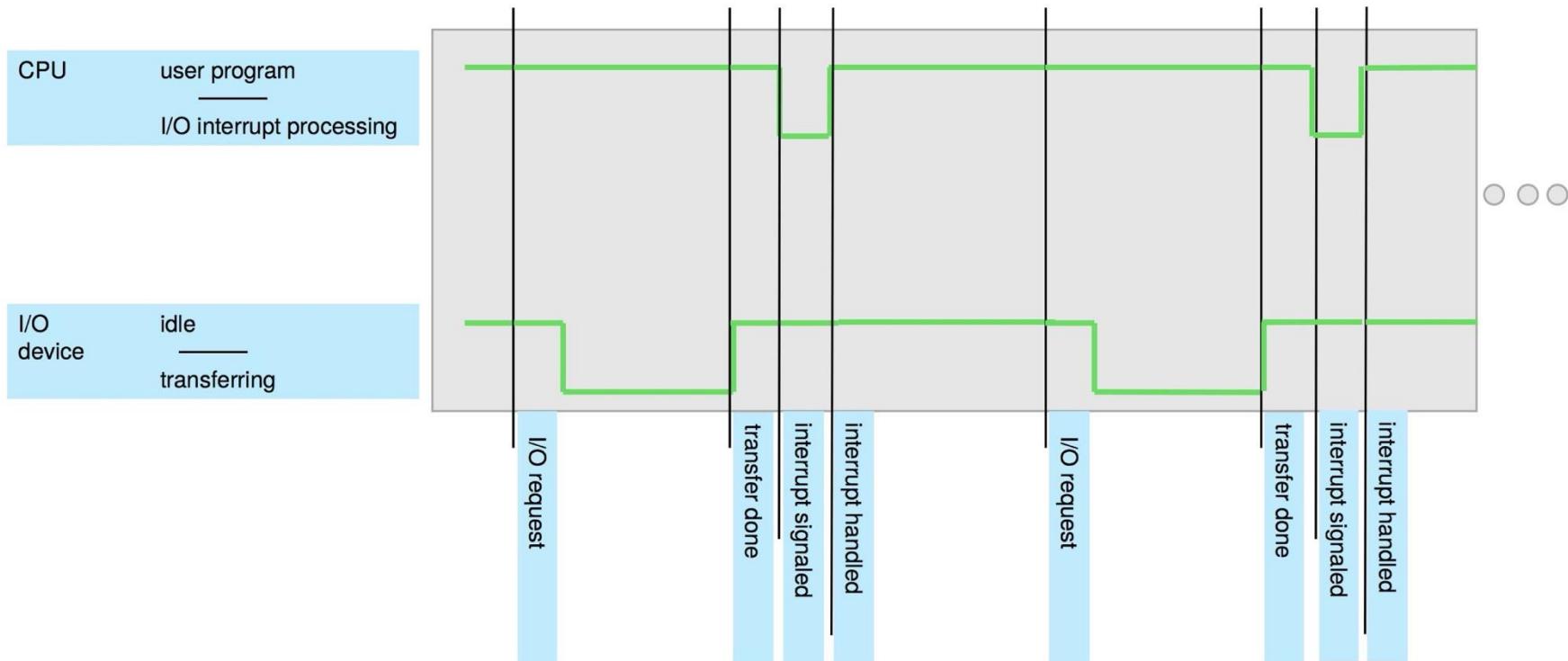
Common Functions of Interrupts

- Interrupt transfers control to the interrupt service routine generally, through the **interrupt vector**, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction
- A **trap** or **exception** is a software-generated interrupt caused either by an error or a user request
- An operating system is **interrupt driven**





Interrupt Timeline





Interrupt Handling

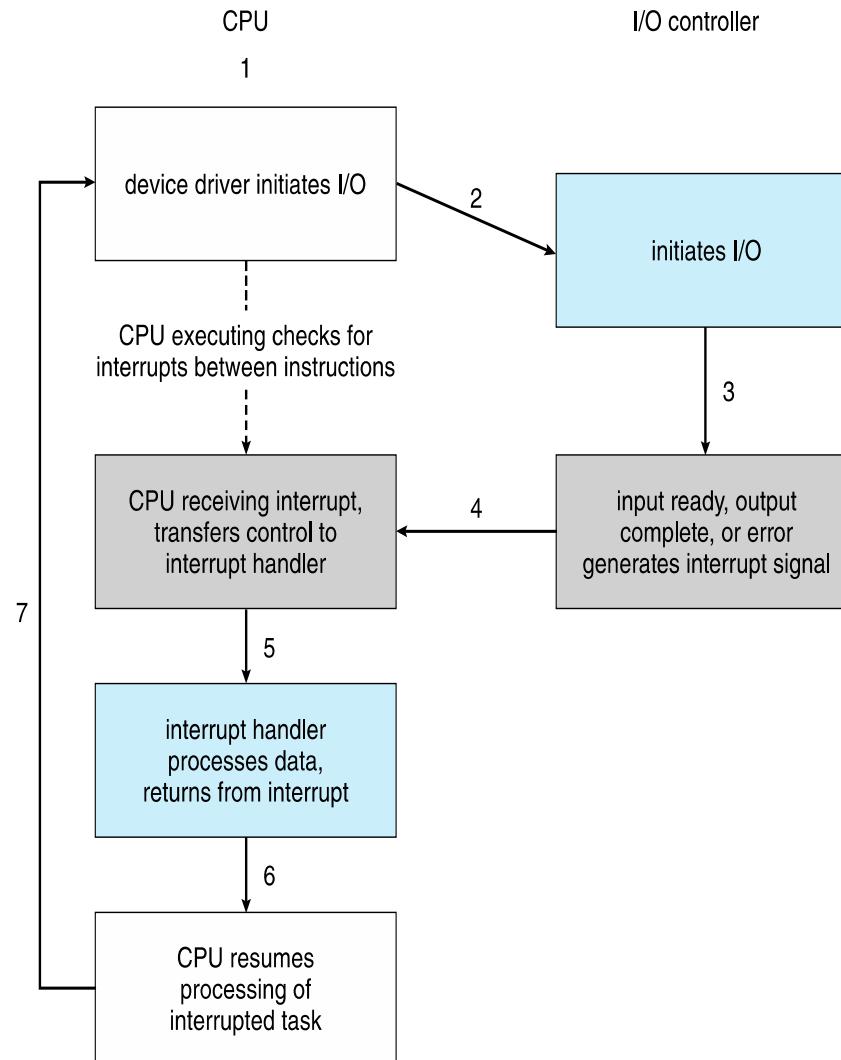
- The operating system preserves the state of the CPU by storing registers and the program counter
- Determines which type of interrupt has occurred:
 - **polling**
 - **vectored** interrupt system
- Separate segments of code determine what action should be taken for each type of interrupt

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





Interrupt-drive I/O Cycle

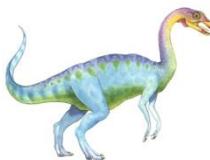




I/O Structure

- After I/O starts, control returns to user program only upon I/O completion
 - Wait instruction idles the CPU until the next interrupt
 - Wait loop (contention for memory access)
 - At most one I/O request is outstanding at a time, no simultaneous I/O processing
- After I/O starts, control returns to user program without waiting for I/O completion
 - **System call** – request to the OS to allow user to wait for I/O completion
 - **Device-status table** contains entry for each I/O device indicating its type, address, and state
 - OS indexes into I/O device table to determine device status and to modify table entry to include interrupt





Storage Structure

- Main memory – only large storage media that the CPU can access directly
 - Random access
 - Typically volatile
 - Typically random-access memory in the form of Dynamic Random-access Memory (DRAM)
- Secondary storage – extension of main memory that provides large nonvolatile storage capacity
- Hard Disk Drives (HDD) – 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
- Non-volatile memory (NVM) devices – faster than hard disks, nonvolatile
 - Various technologies
 - Becoming more popular as capacity and performance increases, price drops





Storage Definitions and Notation Review

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. Given enough bits, it is amazing how many things a computer can represent: numbers, letters, images, movies, sounds, documents, and programs, to name a few. A **byte** is 8 bits, and on most computers it is the smallest convenient chunk of storage. For example, most computers don't have an instruction to move a bit but do have one to move a byte. 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. For example, a computer that has 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words. A computer executes many operations in its native word size rather than a byte at a time.

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; and a **petabyte**, or PB, is $1,024^5$ bytes. 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).

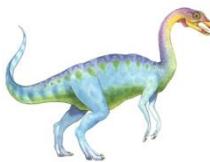




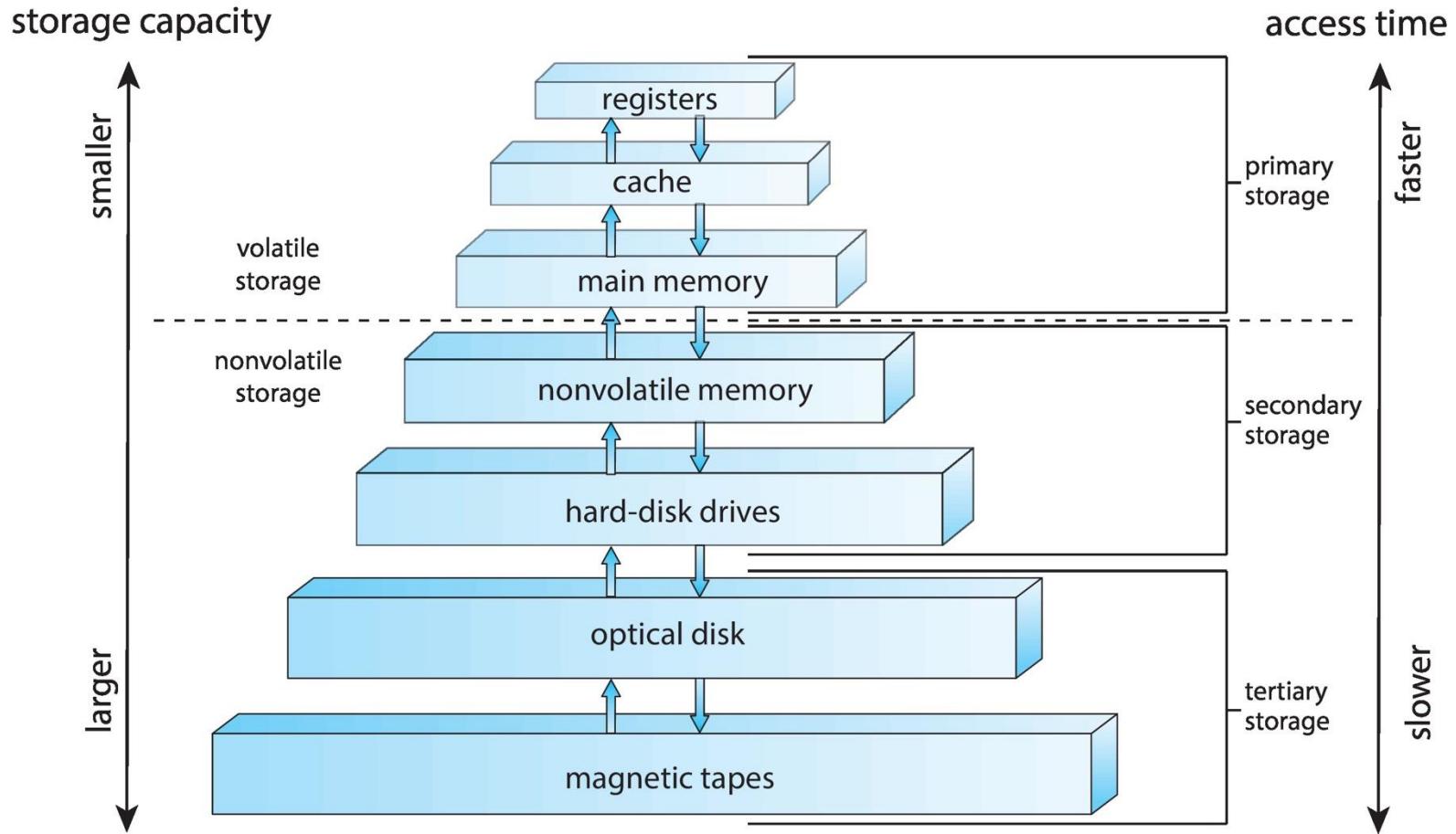
Storage Hierarchy

- Storage systems organized in hierarchy
 - Speed
 - Cost
 - Volatility
- **Caching** – copying information 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



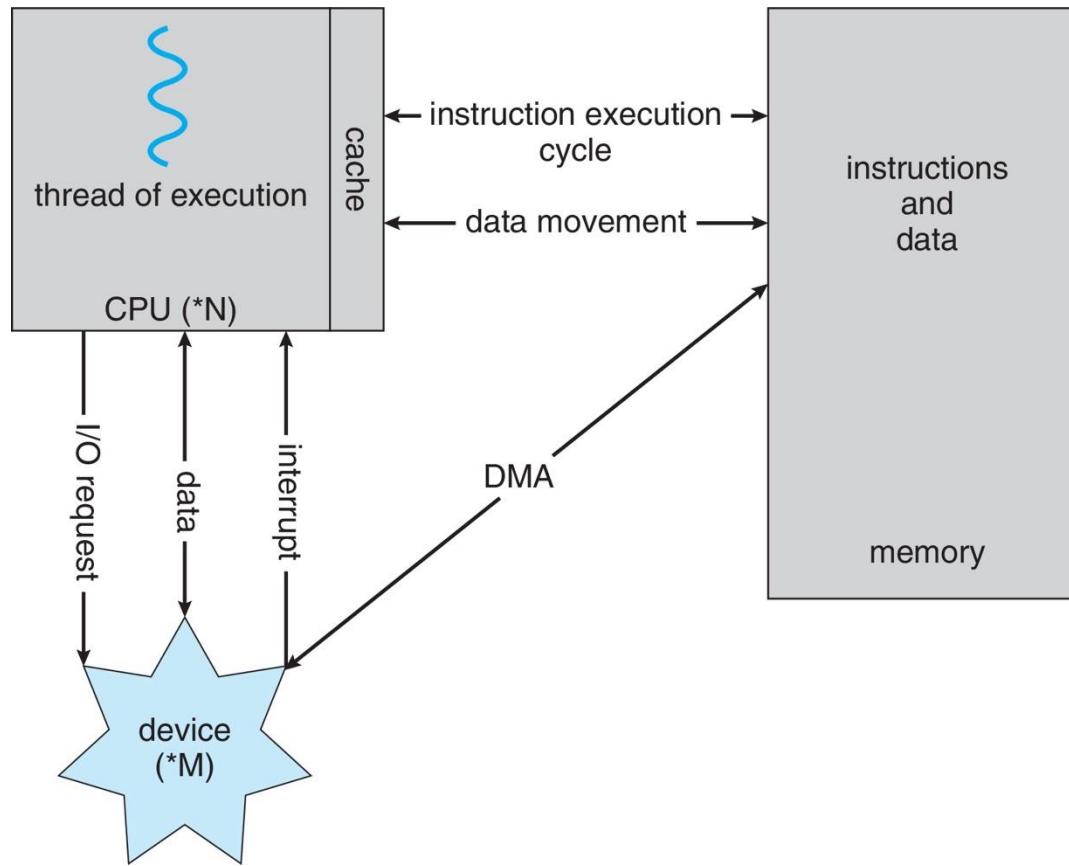


Storage-Device Hierarchy



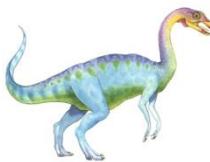


How a Modern Computer Works



A von Neumann architecture





Direct Memory Access Structure

- Used for high-speed I/O devices able to transmit information at close to memory speeds
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block, rather than the one interrupt per byte

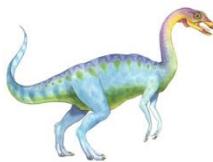




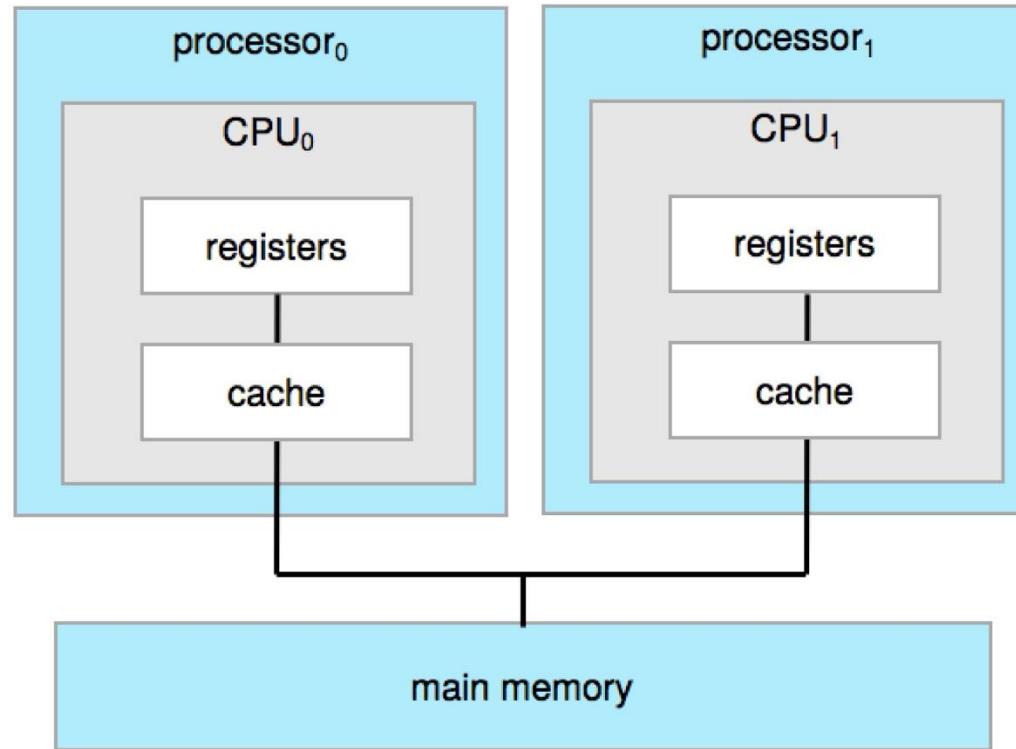
Computer-System Architecture

- Most systems use a single general-purpose processor
 - Most systems have special-purpose processors as well
- **Multiprocessors** systems growing in use and importance
 - Also known as **parallel systems, tightly-coupled systems**
 - Advantages include:
 1. **Increased throughput**
 2. **Economy of scale**
 3. **Increased reliability** – graceful degradation or fault tolerance
 - Two types:
 1. **Asymmetric Multiprocessing** – each processor is assigned a specific task.
 2. **Symmetric Multiprocessing** – each processor performs all tasks





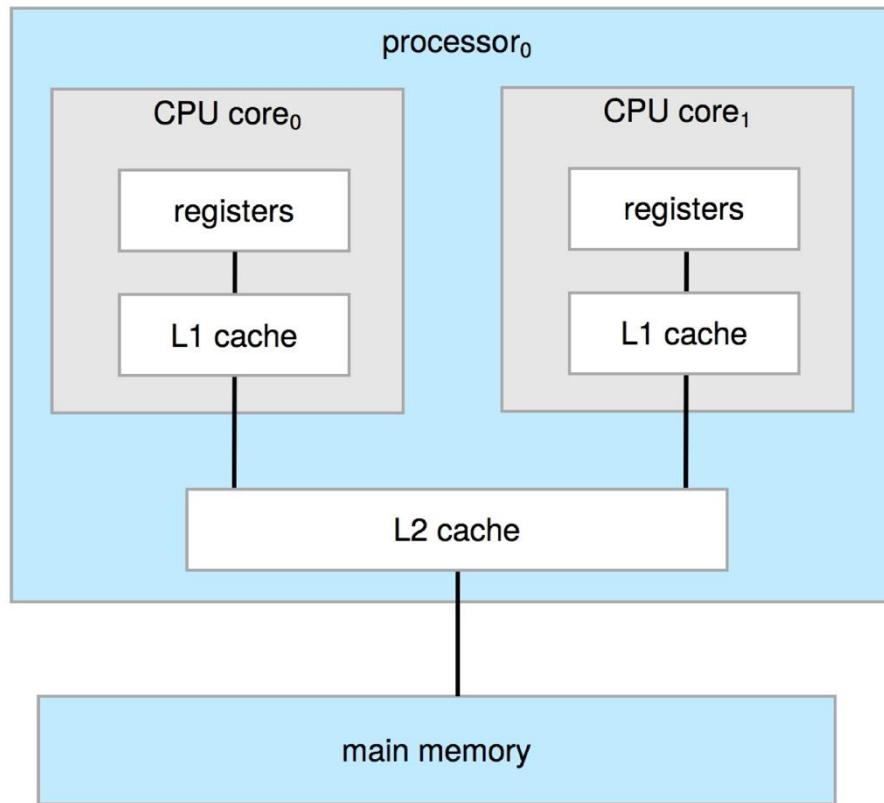
Symmetric Multiprocessing Architecture

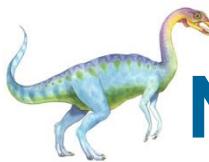




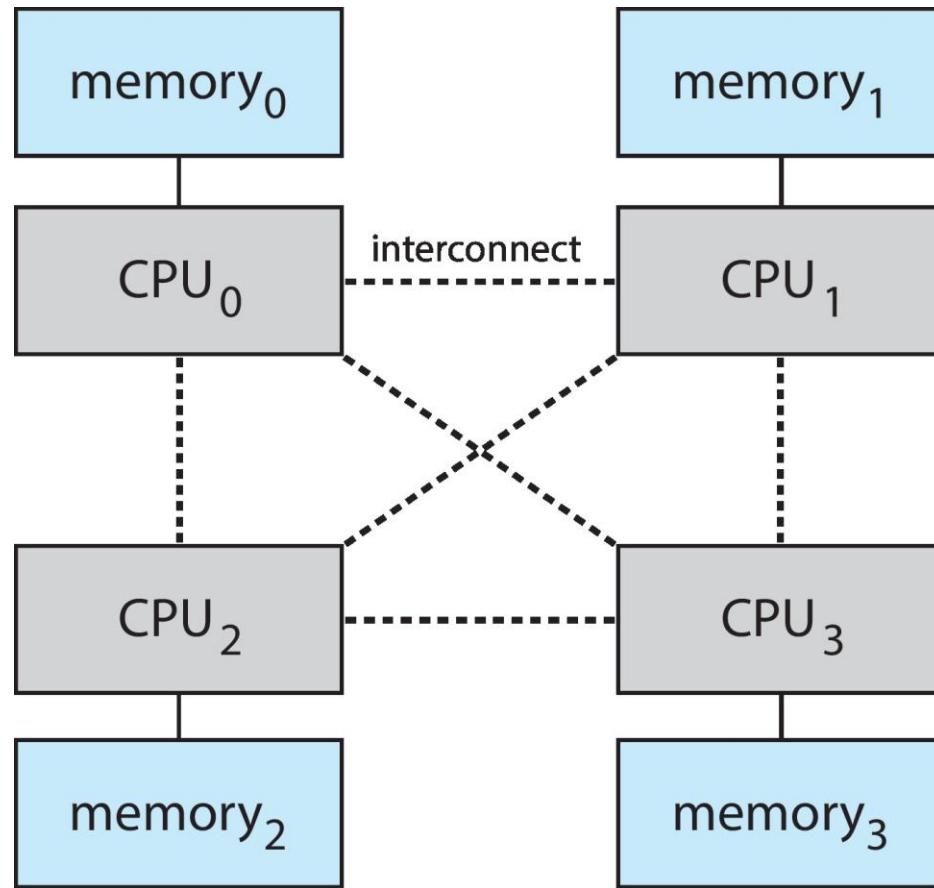
A Dual-Core Design

- Multi-chip and **multicore**
- Systems containing all chips
 - Chassis containing multiple separate systems





Non-Uniform Memory Access System

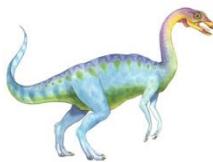




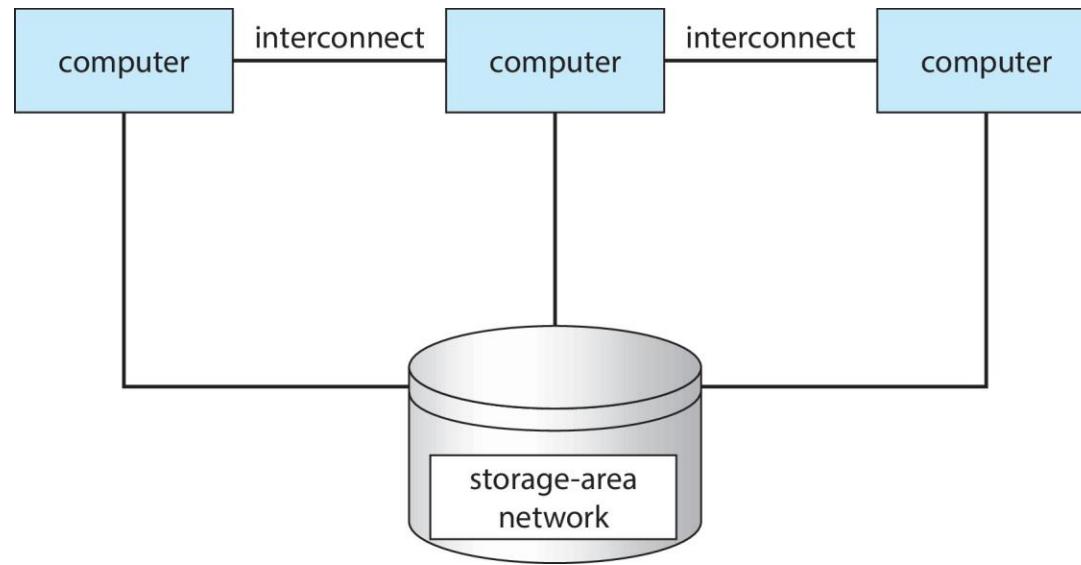
Clustered Systems

- 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





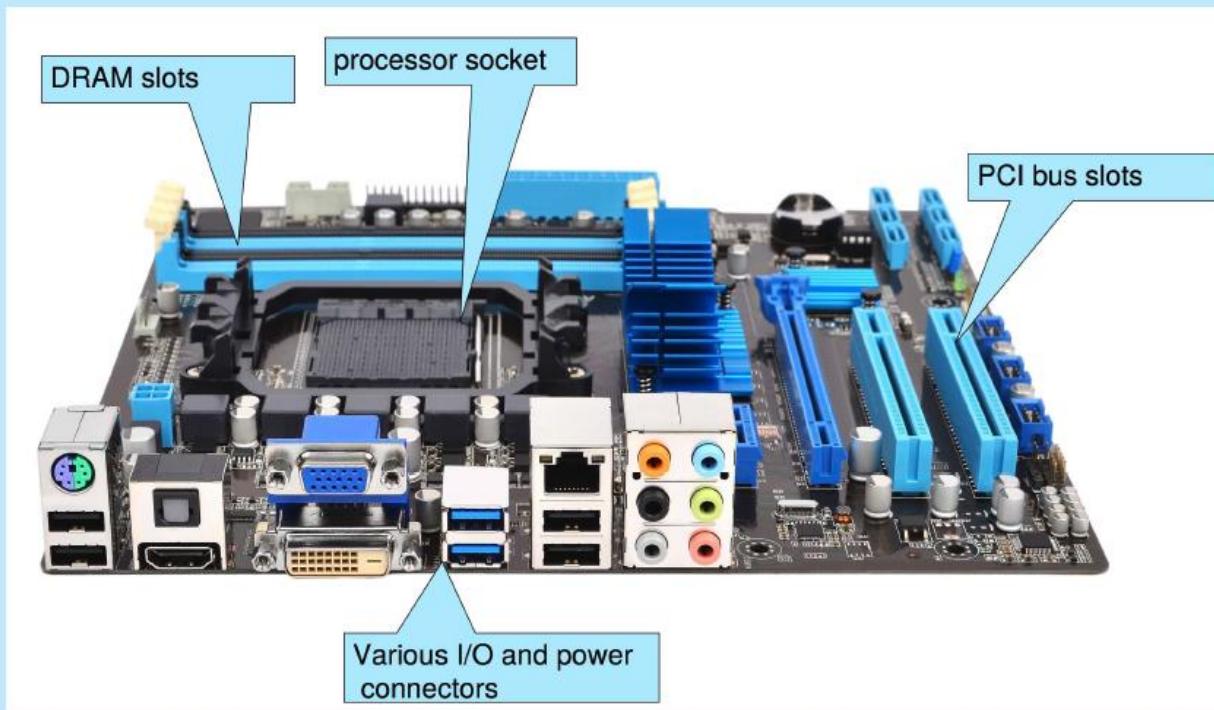
Clustered Systems





PC Motherboard

Consider the desktop PC motherboard with a processor socket shown below:



This board is a fully-functioning computer, once its slots are populated. It consists of a processor socket containing a CPU, DRAM sockets, PCIe bus slots, and I/O connectors of various types. Even the lowest-cost general-purpose CPU contains multiple cores. Some motherboards contain multiple processor sockets. More advanced computers allow more than one system board, creating NUMA systems.





Operating-System Operations

1. Bootstrap program – simple code to initialize the system, load the kernel
2. Kernel loads
3. Starts **system daemons** (services provided outside of the kernel)
4. Kernel **interrupt driven** (hardware and software)
 - Hardware interrupt by one of the devices
 - Software interrupt (**exception** or **trap**):
 - ▶ Software error (e.g., division by zero)
 - ▶ Request for operating system service – **system call**
 - ▶ Other process problems include infinite loop, processes modifying each other or the operating system





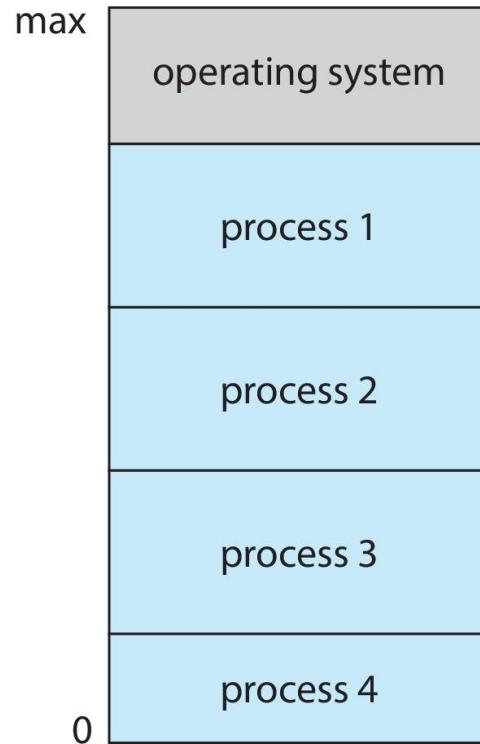
Multiprogramming and Multitasking

- **Multiprogramming (Batch system)** needed for efficiency
 - 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
 - One job selected and run via **job scheduling**
 - When it has to wait (for I/O for example), OS switches to another job
- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
 - **Response time** should be < 1 second
 - Each user has at least one program executing in memory ⇒ **process**
 - If several jobs ready to run at the same time ⇒ **CPU scheduling**
 - If processes don't 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





Dual-mode and Multimode Operation

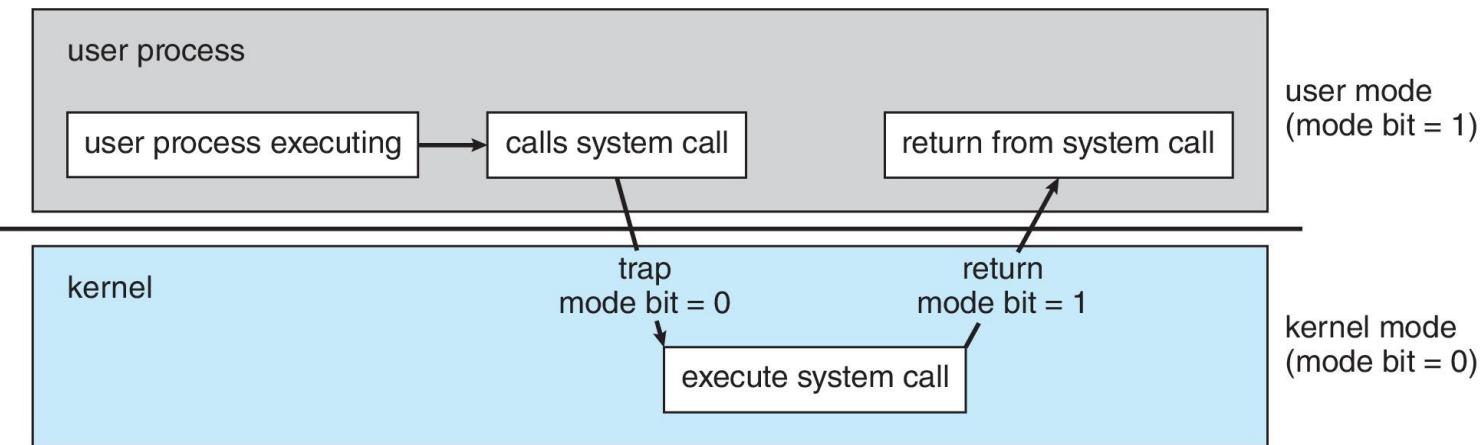
- Dual-mode operation allows OS to protect itself and other system components
 - User mode and kernel mode
 - Mode bit provided by hardware
 - ▶ Provides ability to distinguish when system is running user code or kernel code
 - ▶ Some instructions designated as privileged, only executable in kernel mode
 - ▶ System call changes mode to kernel, return from call resets it to user
- Increasingly CPUs support multi-mode operations
 - i.e. virtual machine manager (VMM) mode for guest VMs





Transition from User to Kernel Mode

- Timer to prevent infinite loop / process hogging resources
 - Timer is set to interrupt the computer after some time period
 - Keep a counter that is decremented by the physical clock
 - Operating system set the counter (privileged instruction)
 - When counter zero generate an interrupt
 - Set up before scheduling 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
 - Initialization data
- Process termination requires reclaim of any reusable resources
- Single-threaded process has one **program counter** specifying location of next instruction to execute
 - Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
 - Concurrency by multiplexing the CPUs among the processes / 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

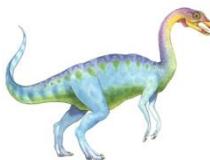




Memory Management

- 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





File-system Management

- OS provides uniform, logical view of information storage
 - Abstracts physical properties to logical storage unit - **file**
 - Each medium is controlled by device (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

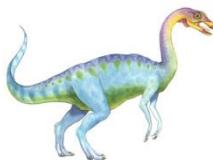




Mass-Storage Management

- 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
 - Mounting and unmounting
 - Free-space management
 - Storage allocation
 - Disk scheduling
 - Partitioning
 - Protection
- 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 smaller than storage being cached
 - Cache management important design problem
 - Cache size and replacement policy





Characteristics of Various Types 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

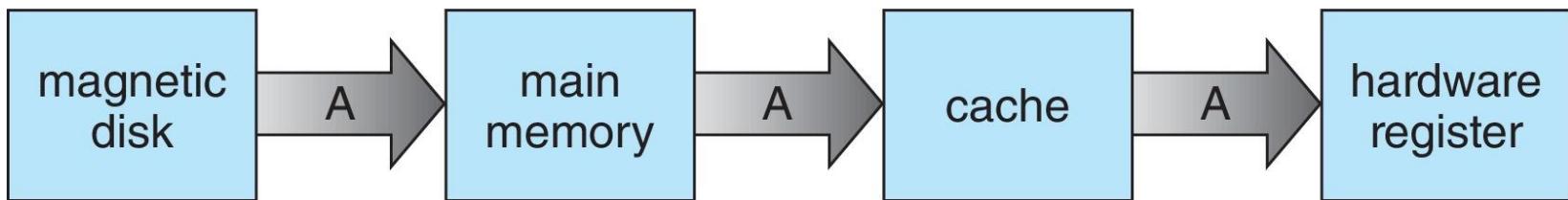
Movement between levels of storage hierarchy can be explicit or implicit





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 19





I/O Subsystem

- One purpose of OS 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** – any mechanism for controlling access of processes or users to resources defined by the OS
- **Security** – 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 then associated with all files, 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 source CPU type different from 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

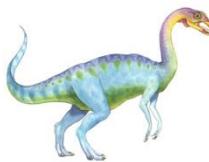




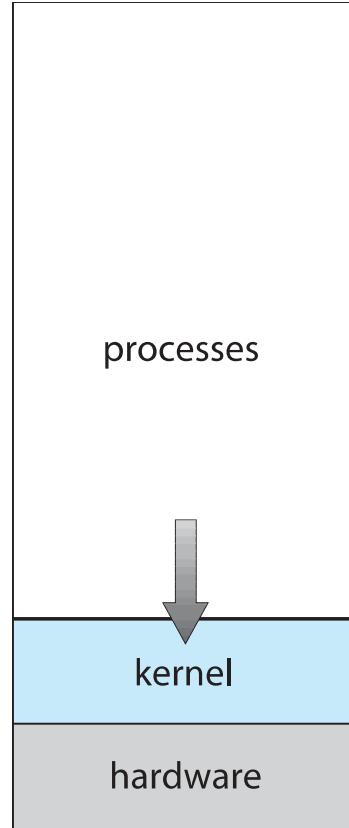
Virtualization (cont.)

- Use cases involve laptops and desktops running multiple OSes for exploration or compatibility
 - Apple laptop running Mac OS X host, Windows as a guest
 - Developing apps for multiple OSes without having multiple systems
 - QA testing applications without having multiple systems
 - Executing and managing compute environments within data centers
- VMM can run natively, in which case they are also the host
 - There is no general purpose host then (VMware ESX and Citrix XenServer)

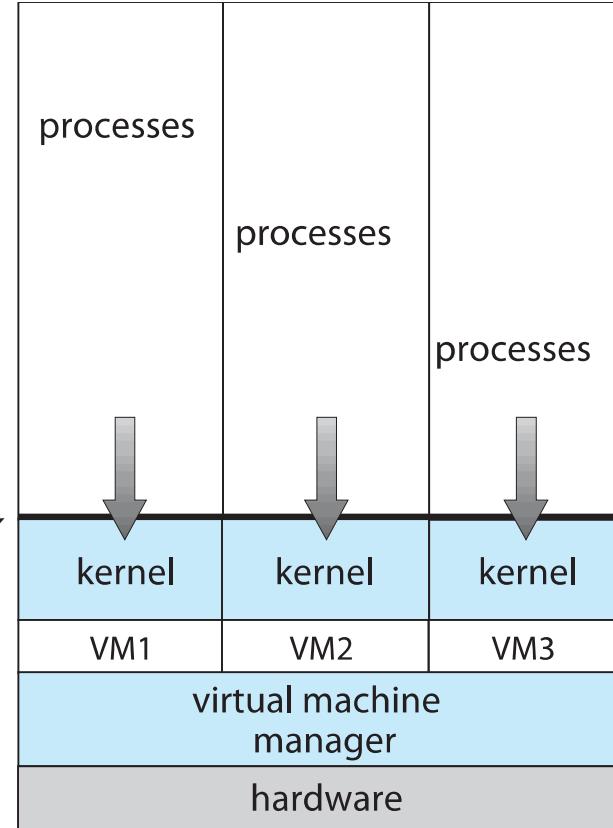




Computing Environments - Virtualization



(a)



(b)





Distributed Systems

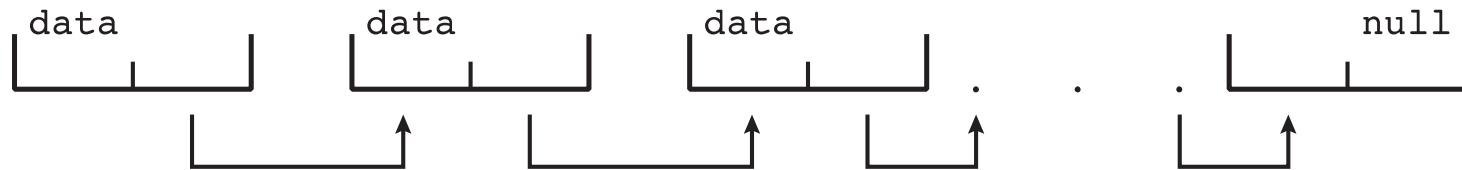
- Distributed computing
 - Collection of separate, possibly heterogeneous, systems networked together
 - ▶ **Network** is a communications path, **TCP/IP** most common
 - **Local Area Network (LAN)**
 - **Wide Area Network (WAN)**
 - **Metropolitan Area Network (MAN)**
 - **Personal Area Network (PAN)**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



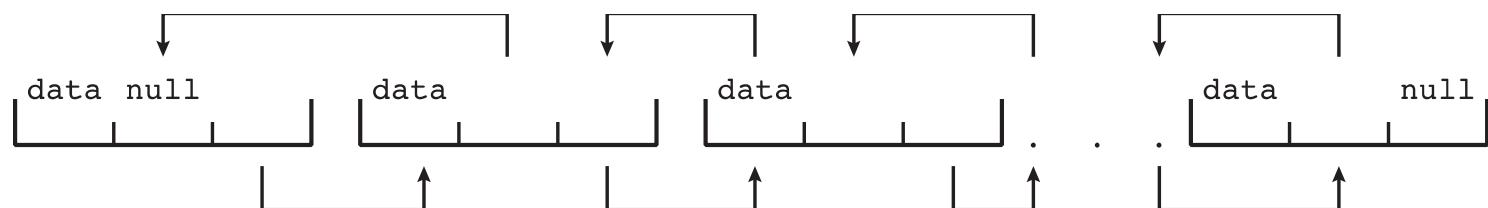


Kernel Data Structures

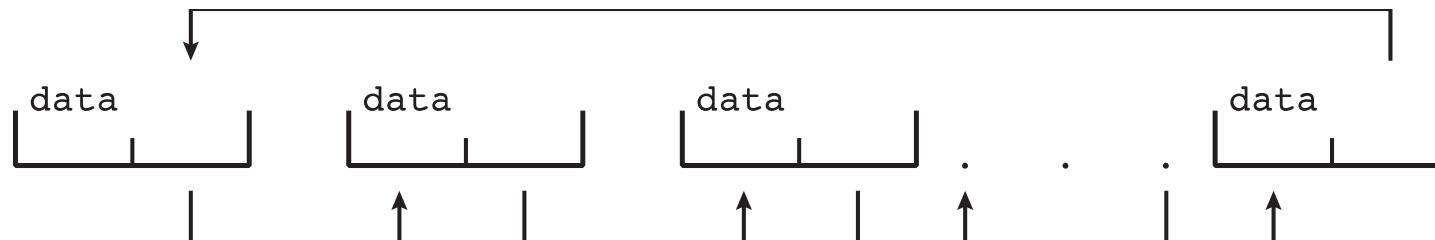
- n Many similar to standard programming data structures
- n ***Singly linked list***

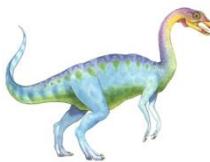


- n ***Doubly linked list***



- n ***Circular linked list***



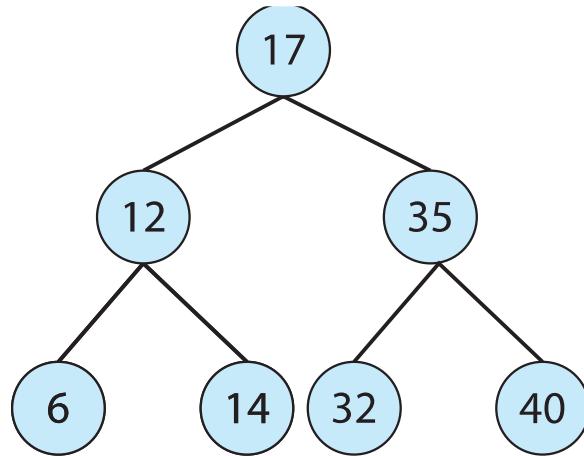


Kernel Data Structures

- **Binary search tree**

left \leq right

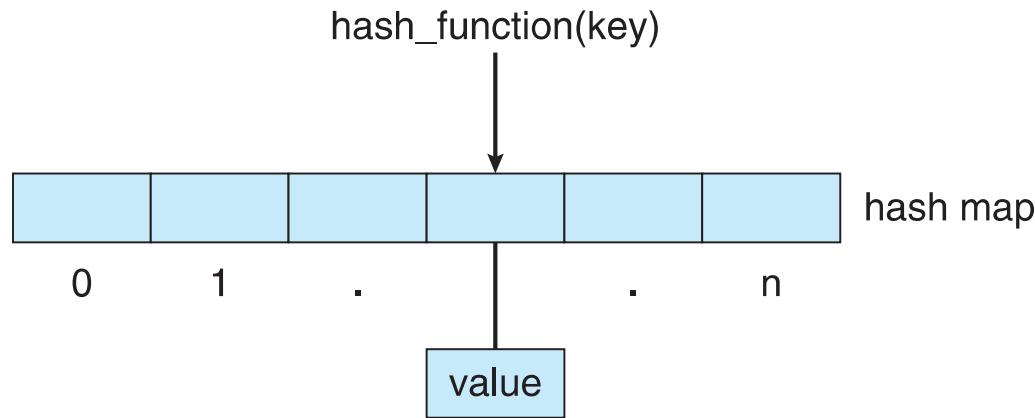
- Search performance is $O(n)$
- **Balanced binary search tree** is $O(\lg n)$





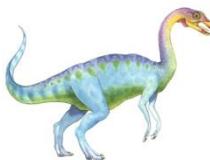
Kernel Data Structures

- 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

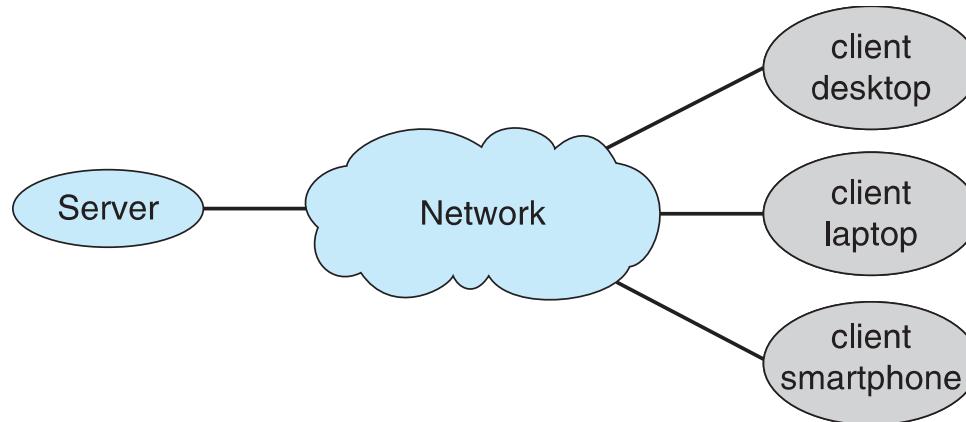
- Handheld smartphones, tablets, etc
- What is the functional difference between them and a “traditional” laptop?
- Extra feature – more OS features (GPS, gyroscope)
- Allows new types of apps like ***augmented reality***
- Use IEEE 802.11 wireless, or cellular data networks for connectivity
- Leaders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





Computing Environments – Client-Server

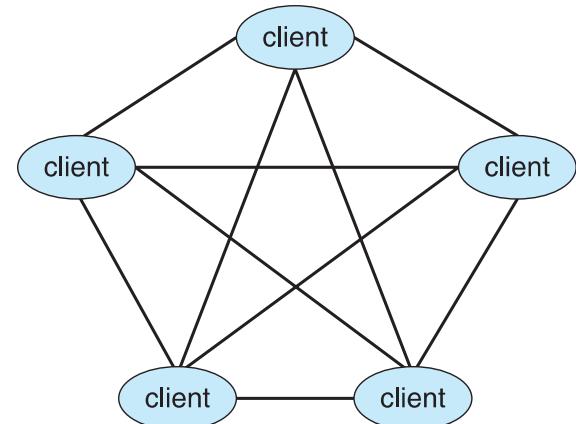
- Client-Server Computing
 - Dumb terminals supplanted by smart PCs
 - Many systems now **servers**, responding to requests generated by **clients**
 - ▶ **Compute-server system** provides an interface to client to request services (i.e., database)
 - ▶ **File-server system** provides interface for clients 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
 - May each 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

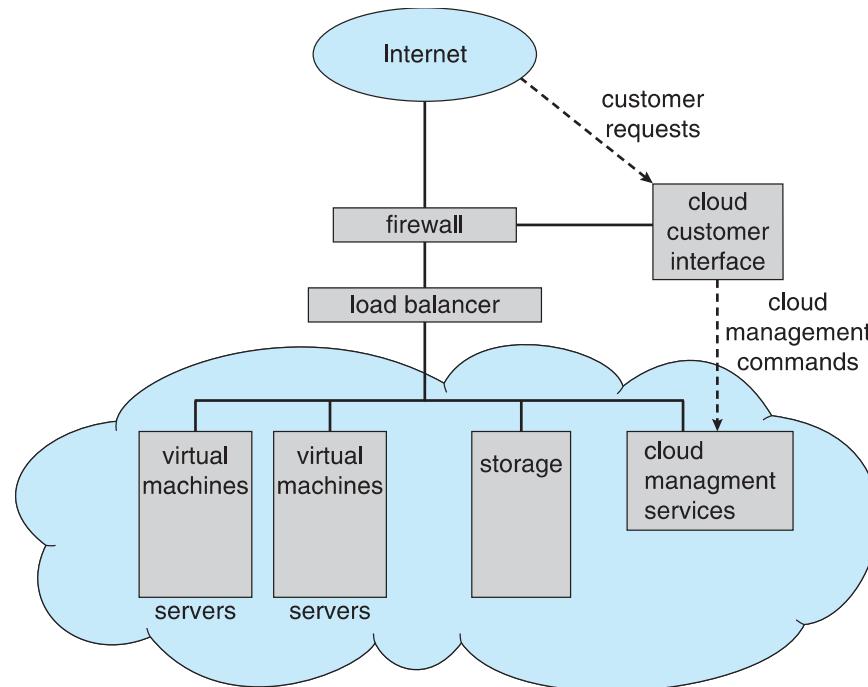
- Delivers computing, storage, even apps as a service across a network
- Logical extension of virtualization because it uses virtualization as the base for its functionality.
 - Amazon **EC2** has thousands of servers, millions of virtual machines, petabytes of storage available across the Internet, 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

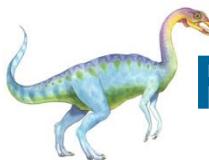




Computing Environments – Real-Time Embedded Systems

- 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

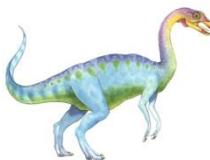




Free and Open-Source Operating Systems

- Operating systems made available in source-code format rather than just binary **closed-source** and **proprietary**
- Counter to the **copy protection** and **Digital Rights Management (DRM)** movement
- Started by **Free Software Foundation (FSF)**, which has “copyleft” **GNU Public License (GPL)**
 - Free software and open-source software are two different ideas championed by different groups of people
 - ▶ <http://gnu.org/philosophy/open-source-misses-the-point.html/>
- 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





The Study of Operating Systems

There has never been a more interesting time to study operating systems, and it has never been easier. The open-source movement has overtaken operating systems, causing many of them to be made available in both source and binary (executable) format. The list of operating systems available in both formats includes Linux, BBSD UNIX, Solaris, and part of macOS. The availability of source code allows us to study operating systems from the inside out. Questions that we could once answer only by looking at documentation or the behavior of an operating system we can now answer by examining the code itself.

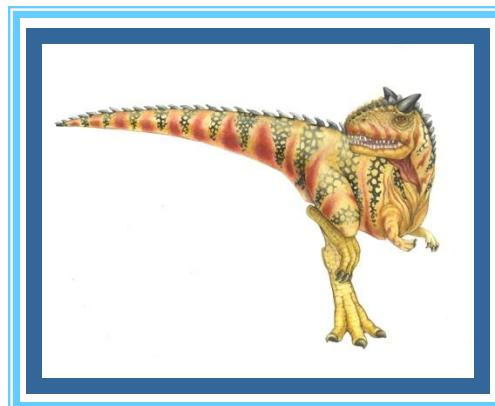
Operating systems that are no longer commercially viable have been open-sourced as well, enabling us to study how systems operated in a time of fewer CPU, memory, and storage resources. An extensive but incomplete list of open-source operating-system projects is available from https://curlie.org/Computers/Software/Operating_Systems/Open_Source/

In addition, the rise of virtualization as a mainstream (and frequently free) computer function makes it possible to run many operating systems on top of one core system. For example, VMware (<http://www.vmware.com>) provides a free “player” for Windows on which hundreds of free “virtual appliances” can run. Virtualbox (<http://www.virtualbox.com>) provides a free, open-source virtual machine manager on many operating systems. Using such tools, students can try out hundreds of operating systems without dedicated hardware.

The advent of open-source operating systems has also made it easier to make the move from student to operating-system developer. With some knowledge, some effort, and an Internet connection, a student can even create a new operating-system distribution. Just a few years ago, it was difficult or impossible to get access to source code. Now, such access is limited only by how much interest, time, and disk space a student has.

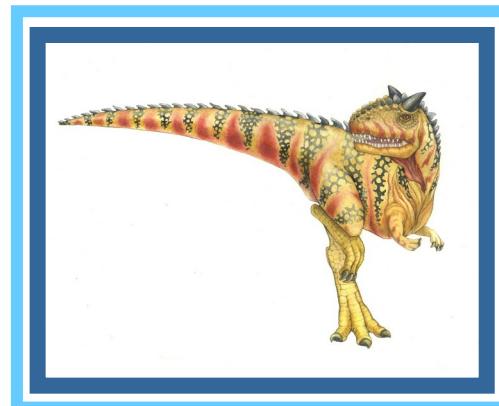


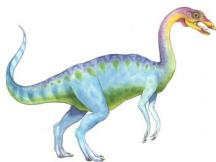
End of Week 1



Week 2

OS Structures





Operating-System Structures

- Operating System Services
- User and Operating System-Interface
- System Calls
- System Services
- Linkers and Loaders
- Why Applications are Operating System Specific
- Operating-System Design and Implementation
- Operating System Structure
- Building and Booting an Operating System
- Operating System Debugging





Operating System Services (1)

- Operating systems provide an **environment** for execution of programs and services to programs and users
- Operating-system services provides functions that are helpful to the user:
 - **User interface** - Almost all operating systems have a user interface (**UI**).
 - ▶ Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **touch-screen**, **Batch**
 - **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

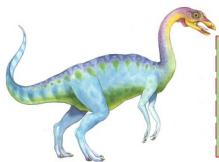




Operating System Services (2)

- **File-system manipulation** - The file system is of particular interest. 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

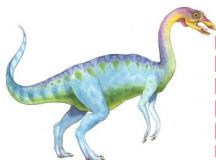




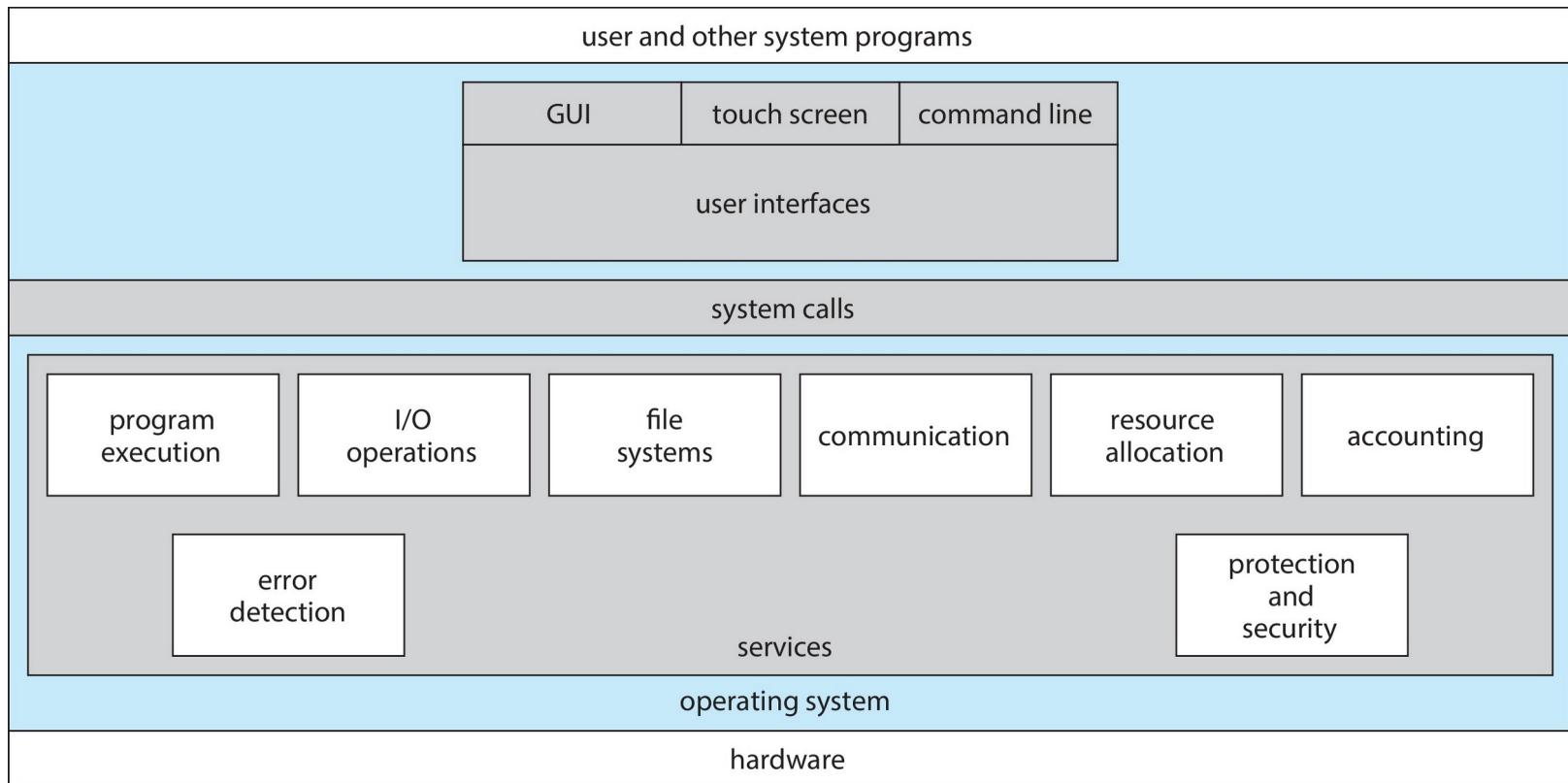
Operating System Services (3)

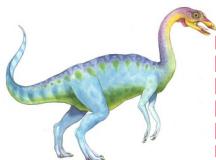
- **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
 - ▶ Many types of resources - CPU cycles, main memory, file storage, I/O devices.
- **Logging** - To keep track of which users use how much and what kinds of computer resources
- **Protection and security** - The owners of information stored in a multi user 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





User Operating System Interface - CLI

CLI or **Command Line Interpreter** allows direct command entry

- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – **shells**
 - ▶ Examples for Unix OS (**sh, csh, ksh, bash, tcsh, zsh**)
- Primarily fetches a command from user and executes it
- Sometimes commands built-in, sometimes just names of programs
 - ▶ If the latter, adding new features doesn't require shell modification

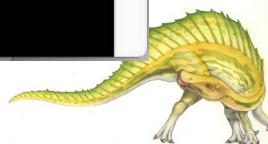




Bourne Shell Command Interpreter

```
1. root@r6181-d5-us01:~ (ssh)
X root@r6181-d5-u... ● %1 X ssh ⚡ %2 X 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 ~]#
```

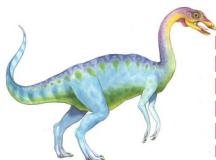




User Operating System Interface - GUI

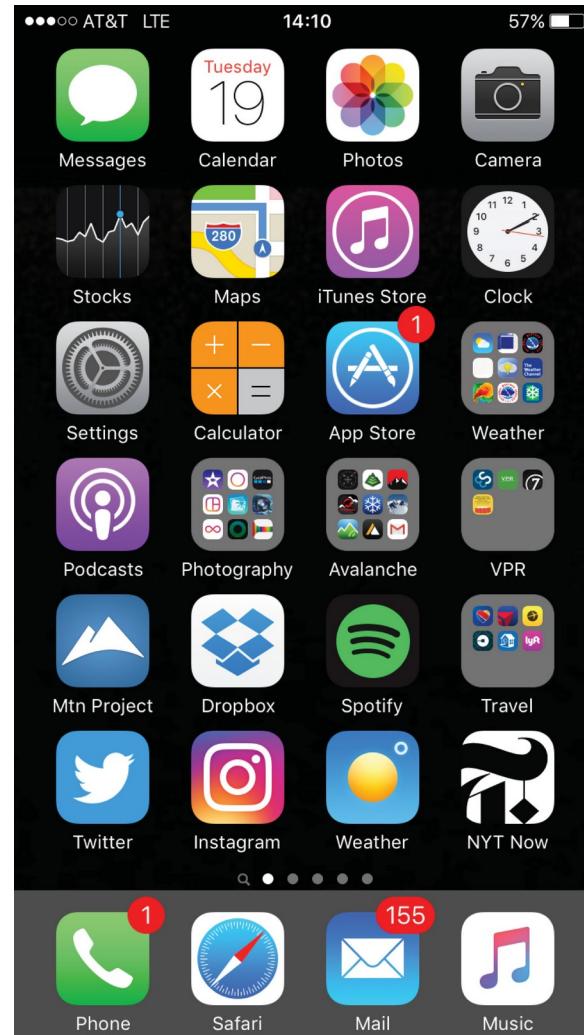
- User-friendly **desktop** metaphor interface
 - Usually mouse, keyboard, and monitor
 - **Icons** represent files, folders, programs, actions, etc
 - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **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



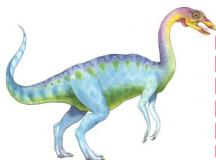


System Calls

- 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 use
- 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)

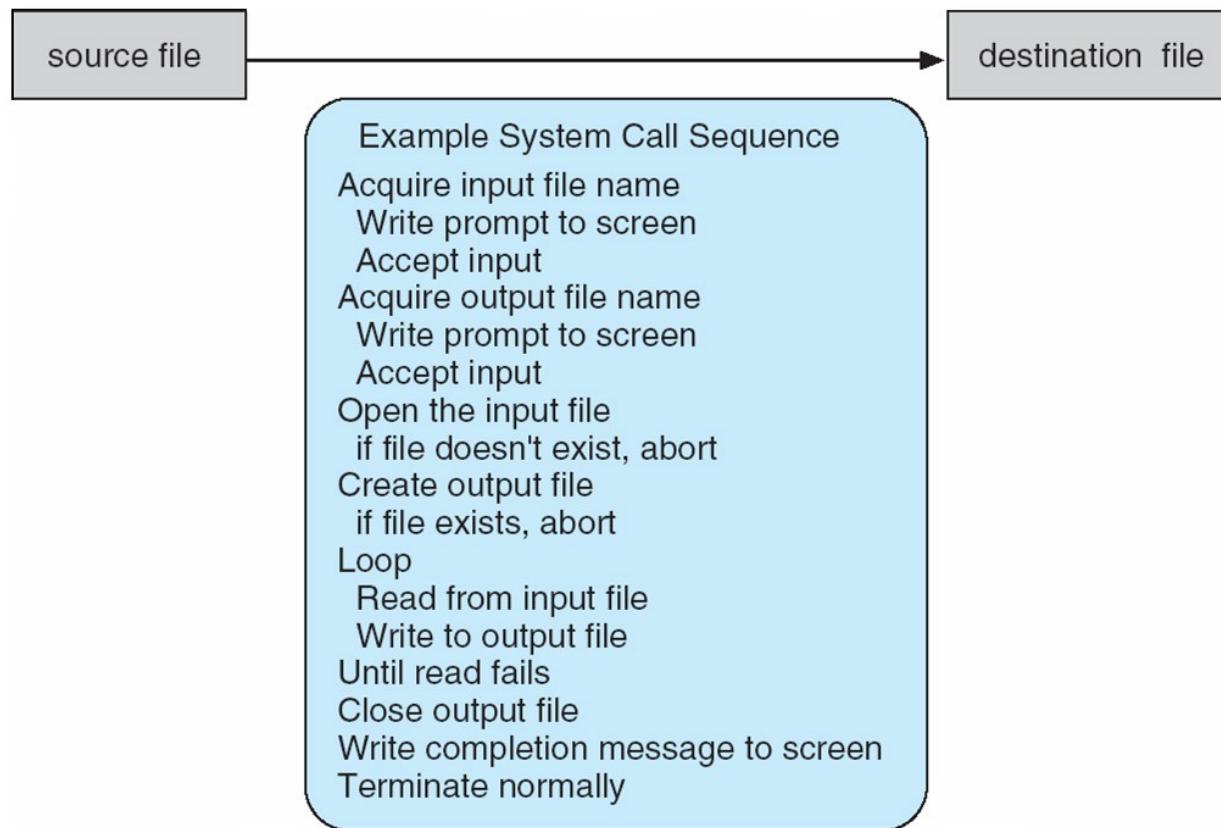
Note that the system-call names used throughout this text are generic

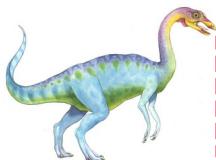




Example of System Calls

- 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)
```

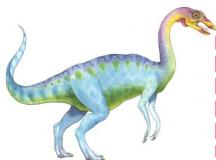
return function parameters
value name

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 into which the data will be read
- `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.





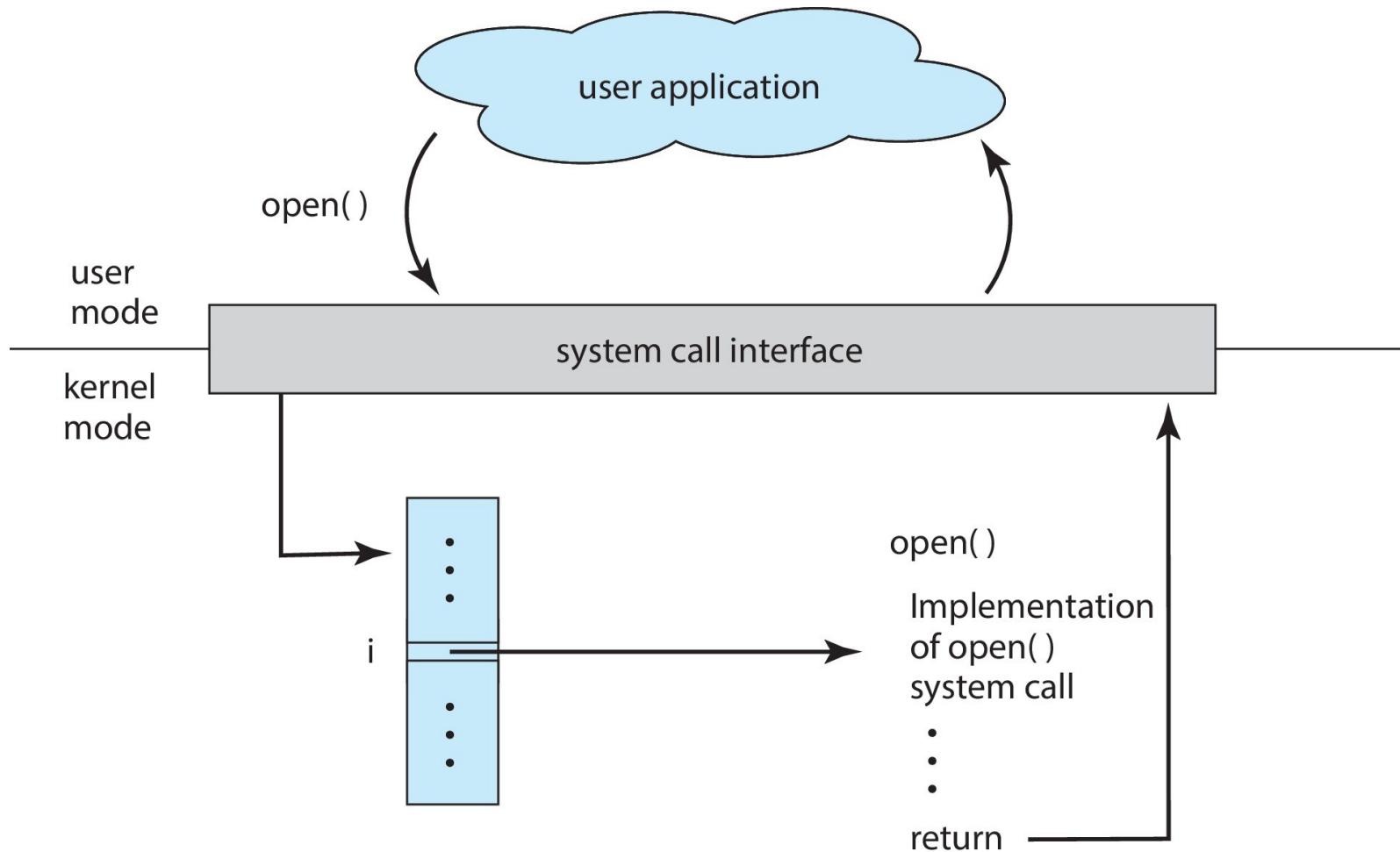
System Call Implementation

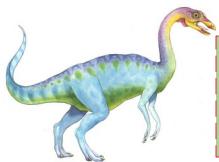
- Typically, a number associated with each system call
 - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
 - Just needs to obey API and understand what 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)





API – System Call – OS Relationship

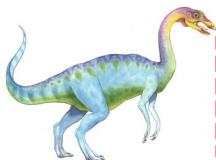




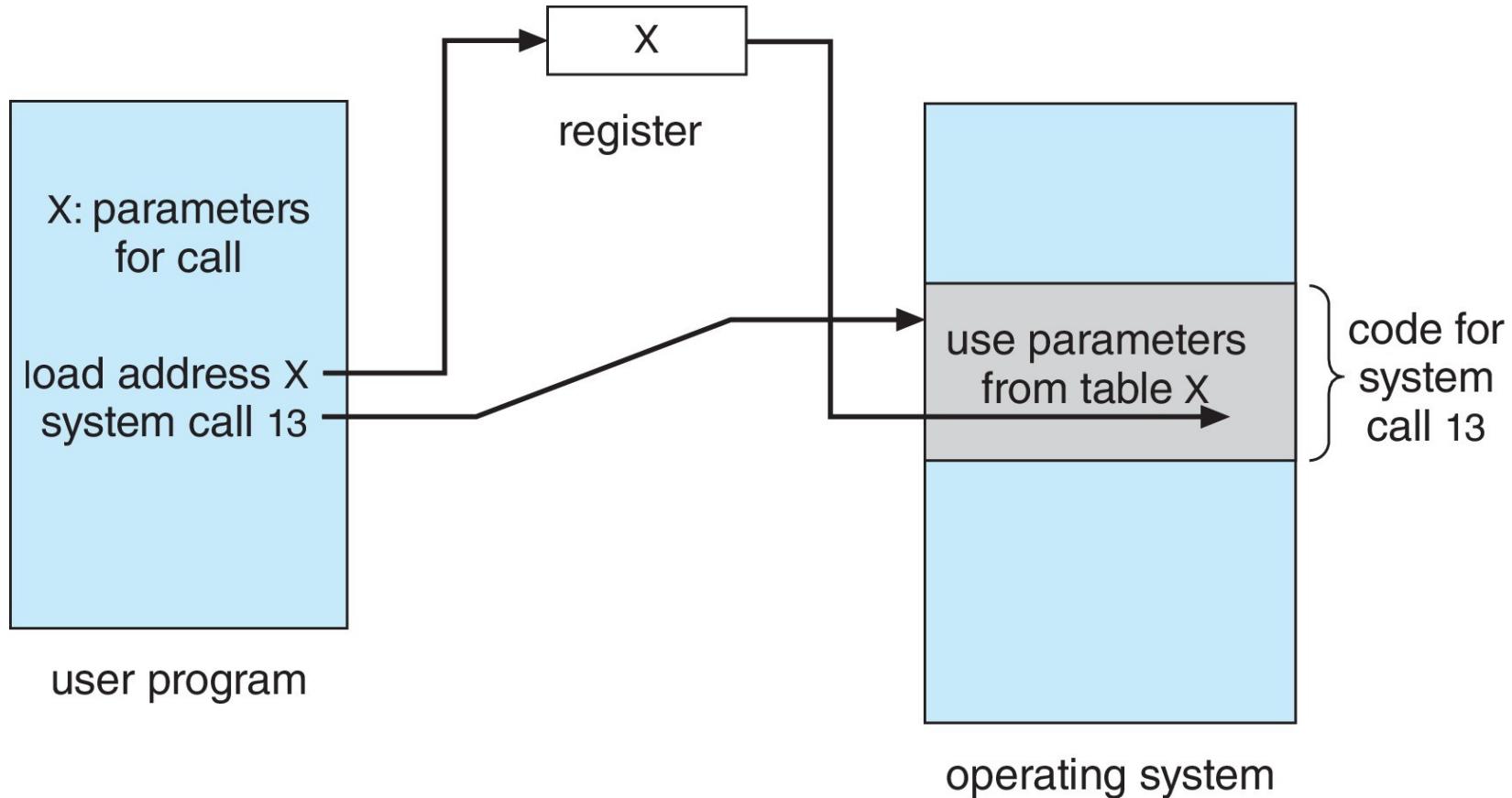
System Call Parameter Passing

- 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





Types of 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





Types of System Calls (cont.)

- File management
 - create file, delete file
 - open, close file
 - read, write, reposition
 - get and set file attributes
- Device management
 - request device, release device
 - read, write, reposition
 - get device attributes, set device attributes
 - logically attach or detach devices





Types of System Calls (Cont.)

- Information maintenance
 - get time or date, set time or date
 - get system data, set system data
 - get and set process, file, or device attributes
- Communications
 - create, delete communication connection
 - send, receive messages if **message passing model** to **host name** or **process name**
 - ▶ From **client** to **server**
 - **Shared-memory model** create and gain access to memory regions
 - transfer status information
 - attach and detach remote devices
- Protection
 - Control access to resources
 - Get and set permissions
 - Allow and deny user access





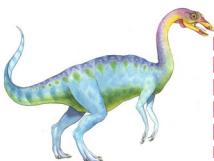
Windows and Unix System Calls

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

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



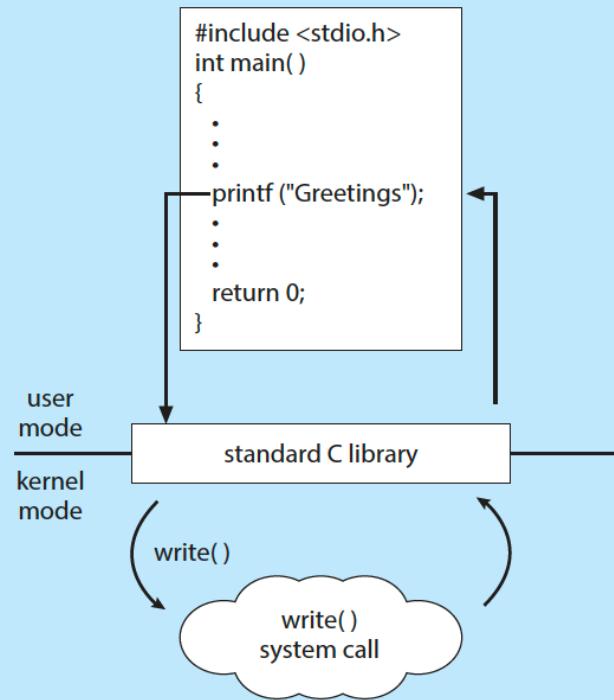


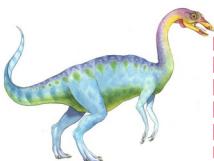
Standard C Library Example

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

THE STANDARD C LIBRARY

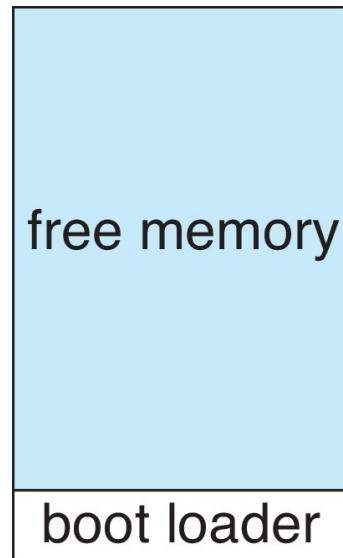
The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:





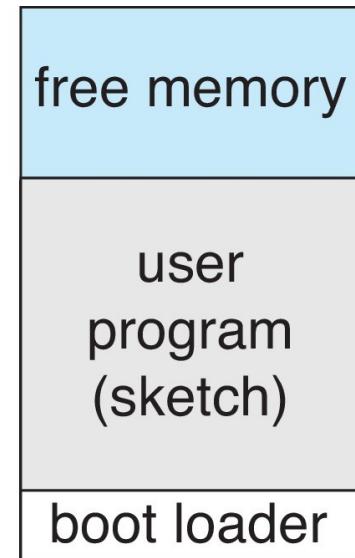
Example: Arduino

- Single-tasking
- No operating system
- Programs (sketch) loaded via USB into flash memory
- Single memory space
- Boot loader loads program
- Program exit -> shell reloaded



(a)

At system startup



(b)

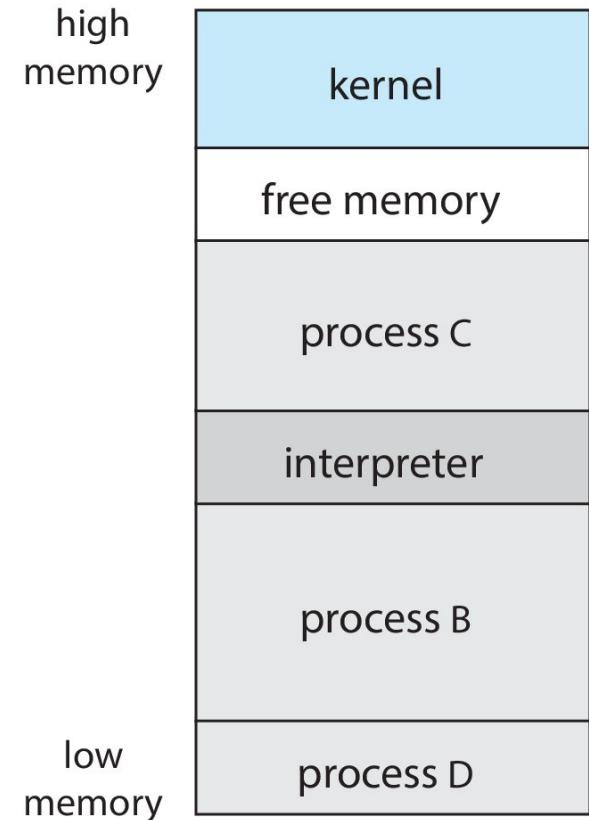
running a program





Example: FreeBSD

- Unix variant
- Multitasking
- User login -> invoke user's choice of shell
- Shell executes fork() system call to create process
 - Executes exec() to load program into process
 - Shell waits for process to terminate or continues with user commands
- Process exits with:
 - code = 0 – no error
 - code > 0 – error code





System Services

- System programs provide a convenient environment for program development and execution. They can be divided into:
 - File manipulation
 - Status information sometimes stored in a file
 - 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 Services (cont.)

- Provide a convenient environment for program development and execution
 - Some of them are simply user interfaces to system calls; others are considerably more complex
- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
 - Some ask the system for info - date, time, amount of available memory, disk space, number of users
 - Others 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 Services (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 Services (Cont.)

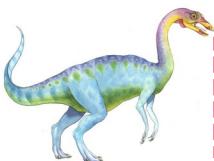
■ Background Services

- 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

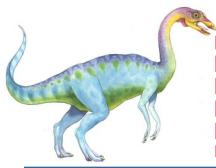




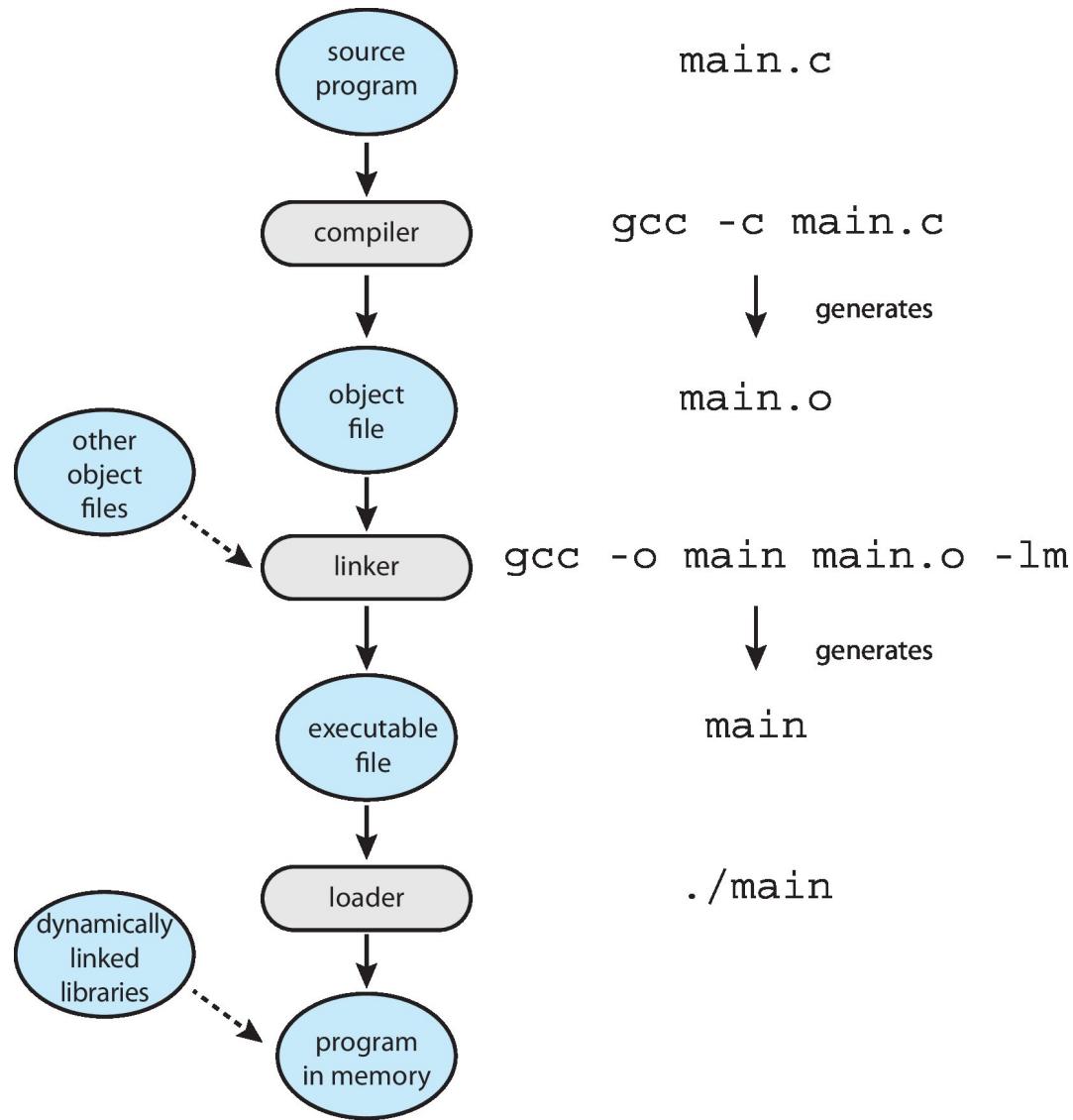
Linkers and Loaders

- Source code compiled into object files designed to be loaded into any physical memory location – **relocatable object file**
- **Linker** combines these into single binary **executable** file
 - Also brings in libraries
- Program resides on secondary storage as binary executable
- Must be brought into memory by **loader** to be executed
 - **Relocation** assigns final addresses to program parts and adjusts code and data in program to match those addresses
- Modern general purpose systems don't link libraries into executables
 - Rather, **dynamically linked libraries** (in Windows, **DLLs**) are loaded as needed, shared by all that use the same version of that same library (loaded once)
- Object, executable files have standard formats, so operating system knows how to load and start them





The Role of the Linker and Loader





Why Applications are Operating System Specific?

- Apps compiled on one system usually not executable on other operating systems
- Each operating system provides its own unique system calls
 - Own file formats, etc
- Apps can be multi-operating system
 - Written in interpreted language like Python, Ruby, and interpreter available on multiple operating systems
 - App written in language that includes a VM containing the running app (like Java)
 - Use standard language (like C), compile separately on each operating system to run on each
- **Application Binary Interface (ABI)** is architecture equivalent of API, defines how different components of binary code can interface for a given operating system on a given architecture, CPU, etc

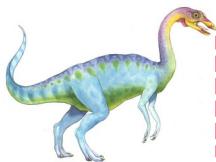




Operating System Design and Implementation (1)

- 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
- **User** goals and **System** 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





Operating System Design and Implementation (2)

- 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)
- Specifying and designing an OS is highly creative task of **software engineering**





Implementation

- Much variation
 - Early OSes in assembly language
 - Then 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
- More 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

- General-purpose OS is very large program
- Various ways to structure ones
 - Simple structure – MS-DOS
 - More complex -- UNIX
 - Layered – an abstraction
 - Microkernel -Mach





Monolithic Structure – Original UNIX

UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts

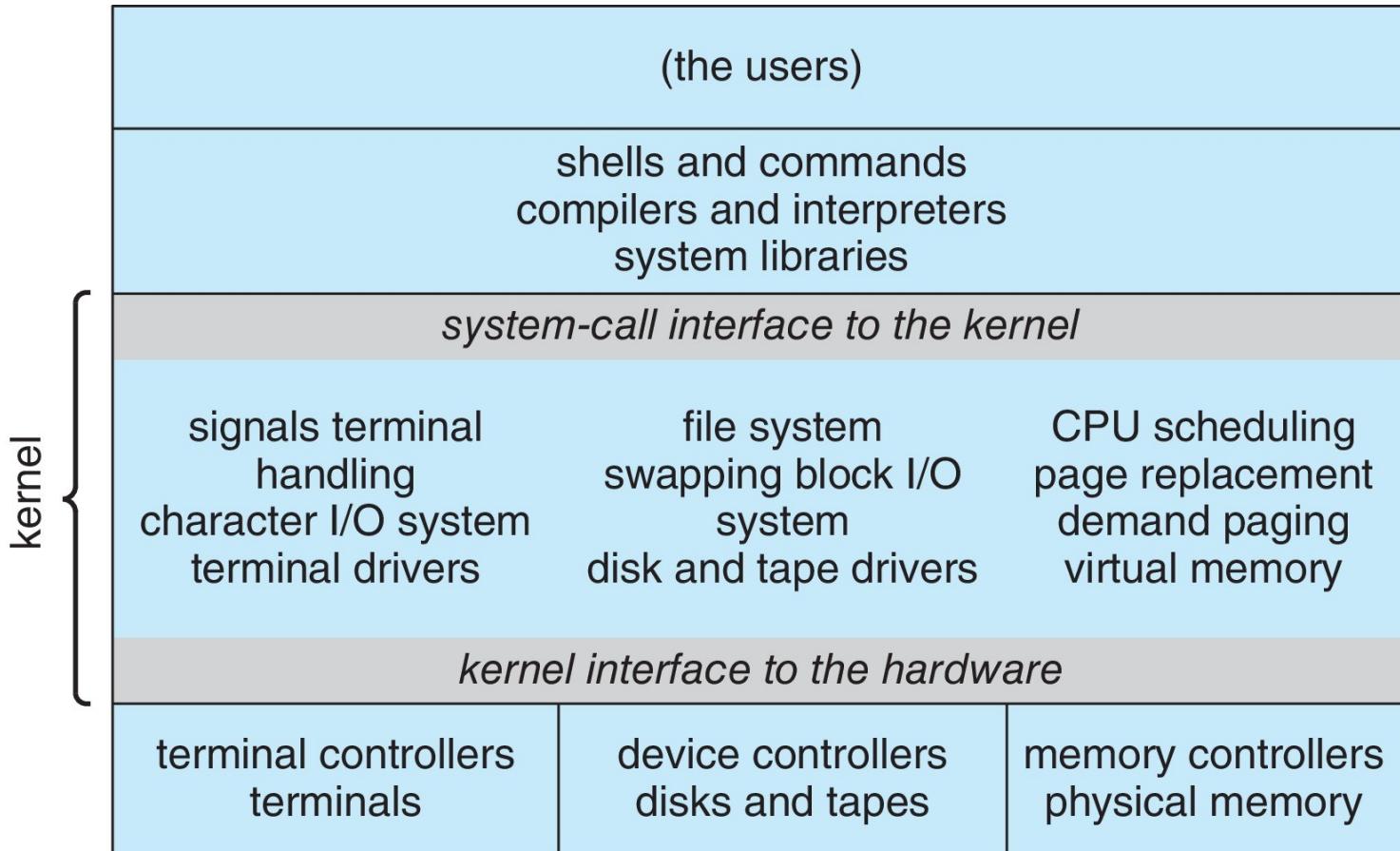
- Systems programs
- The kernel
 - ▶ Consists of everything below the system-call interface and above the physical hardware
 - ▶ Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

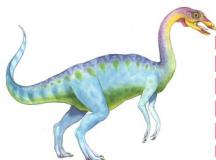




Traditional UNIX System Structure

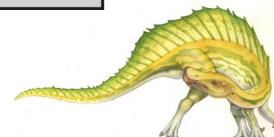
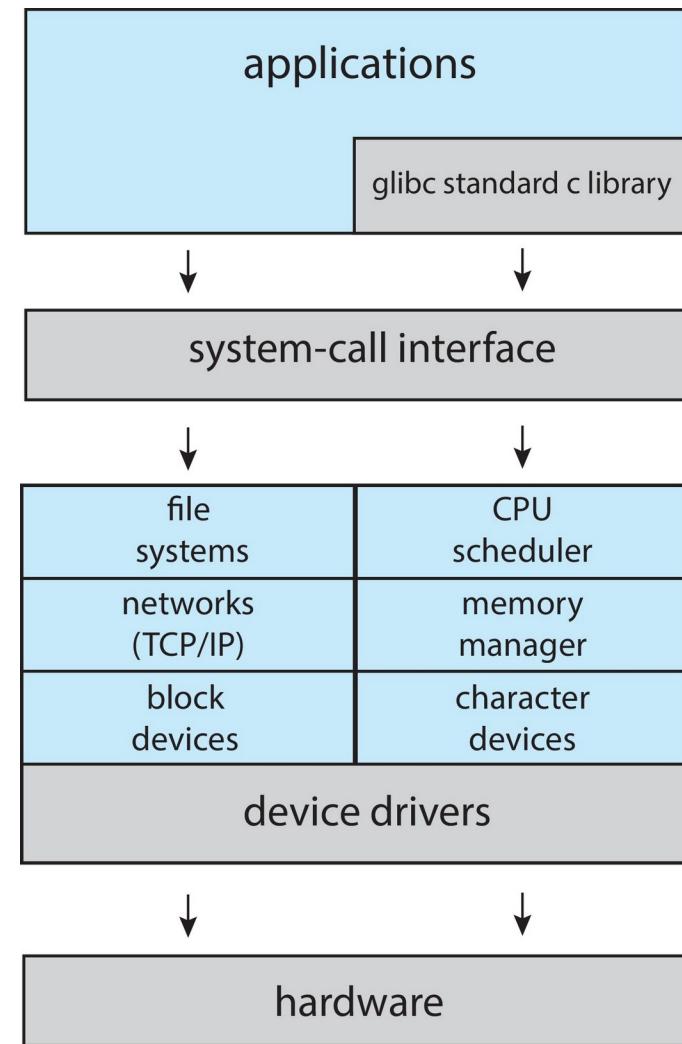
Beyond simple but not fully layered

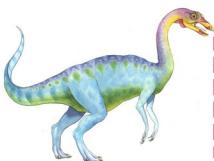




Linux System Structure

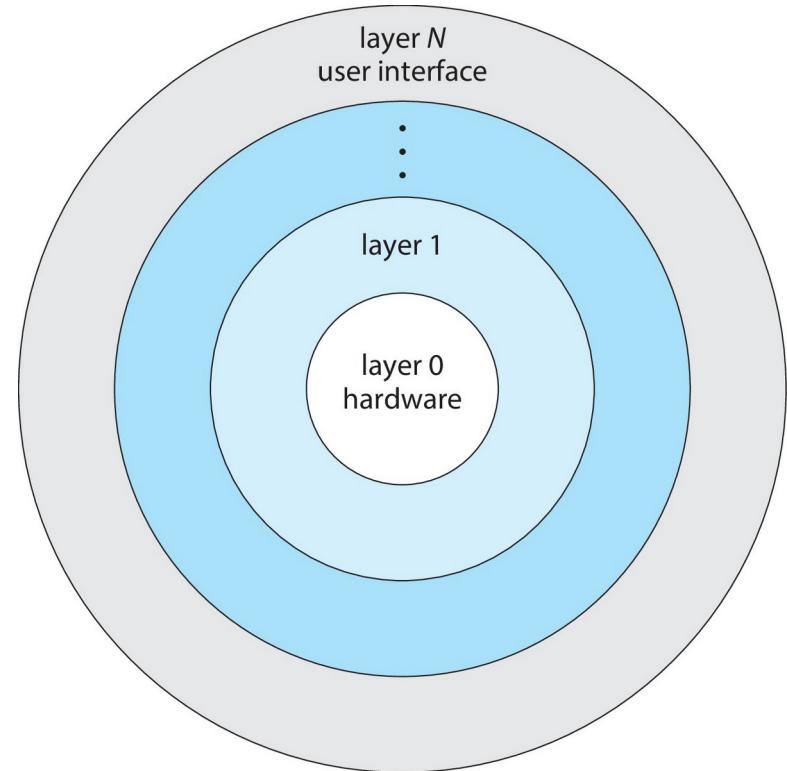
Monolithic plus modular design





Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers





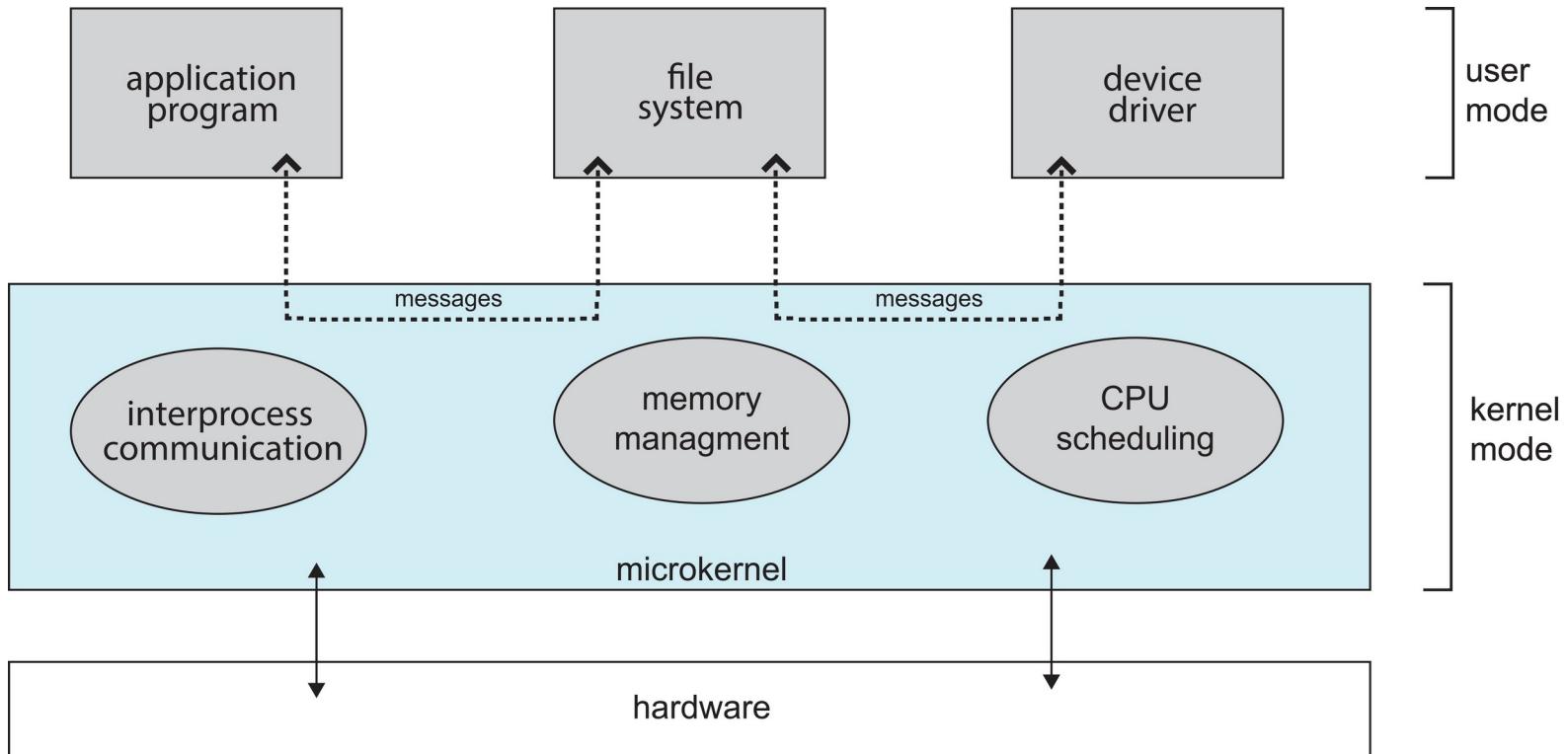
Microkernels

- 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





Micro kernel System Structure





Modules

- Many modern operating systems implement **loadable kernel modules (LKMs)**
 - 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

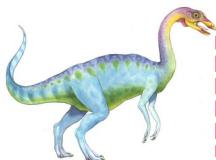




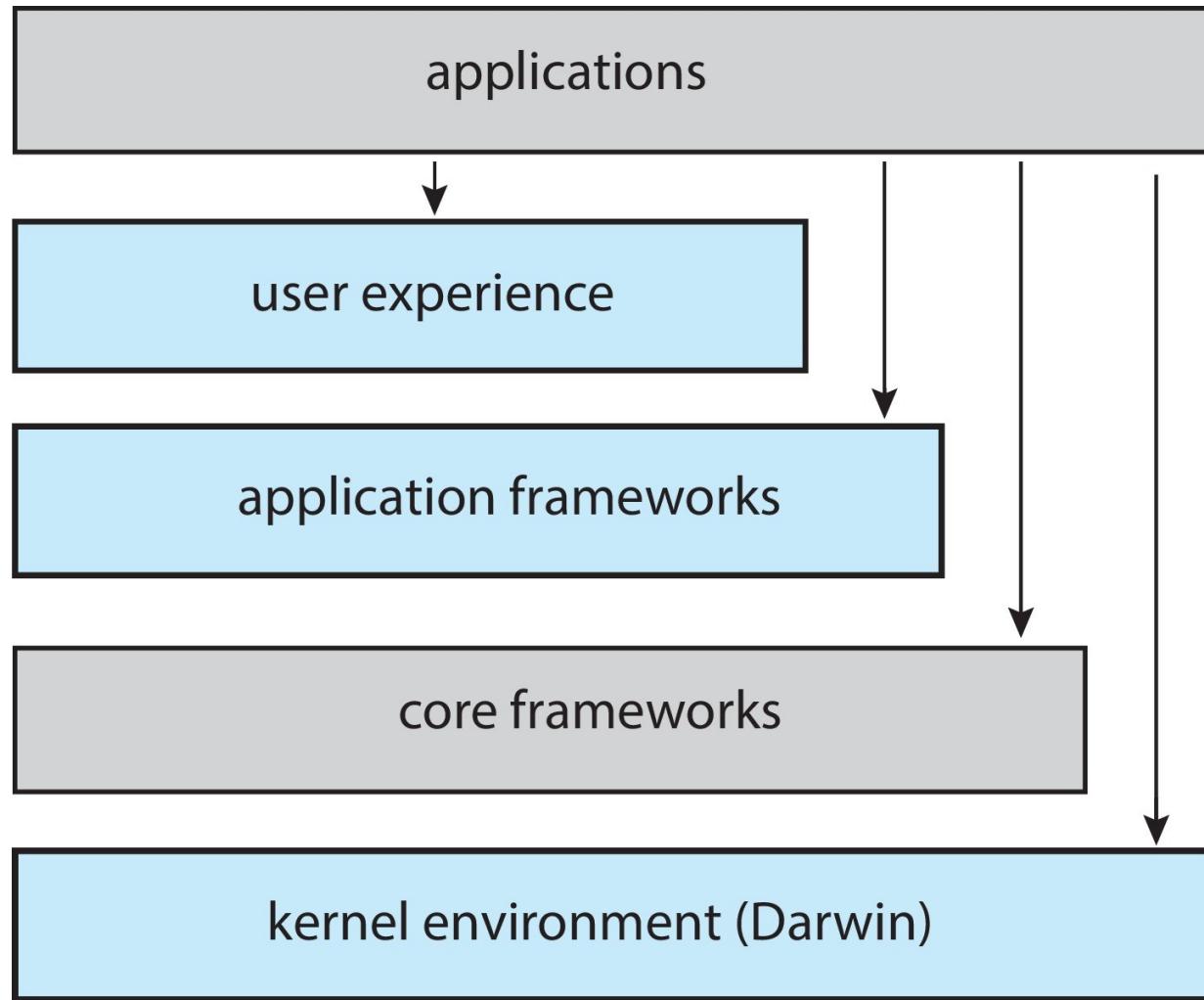
Hybrid Systems

- Most modern operating systems are actually not one pure model
 - Hybrid combines multiple approaches to address performance, security, usability needs
 - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
 - Windows mostly monolithic, plus microkernel for different subsystem **personalities**
- Apple Mac OS X hybrid, layered, **Aqua** UI plus **Cocoa** programming environment
 - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)



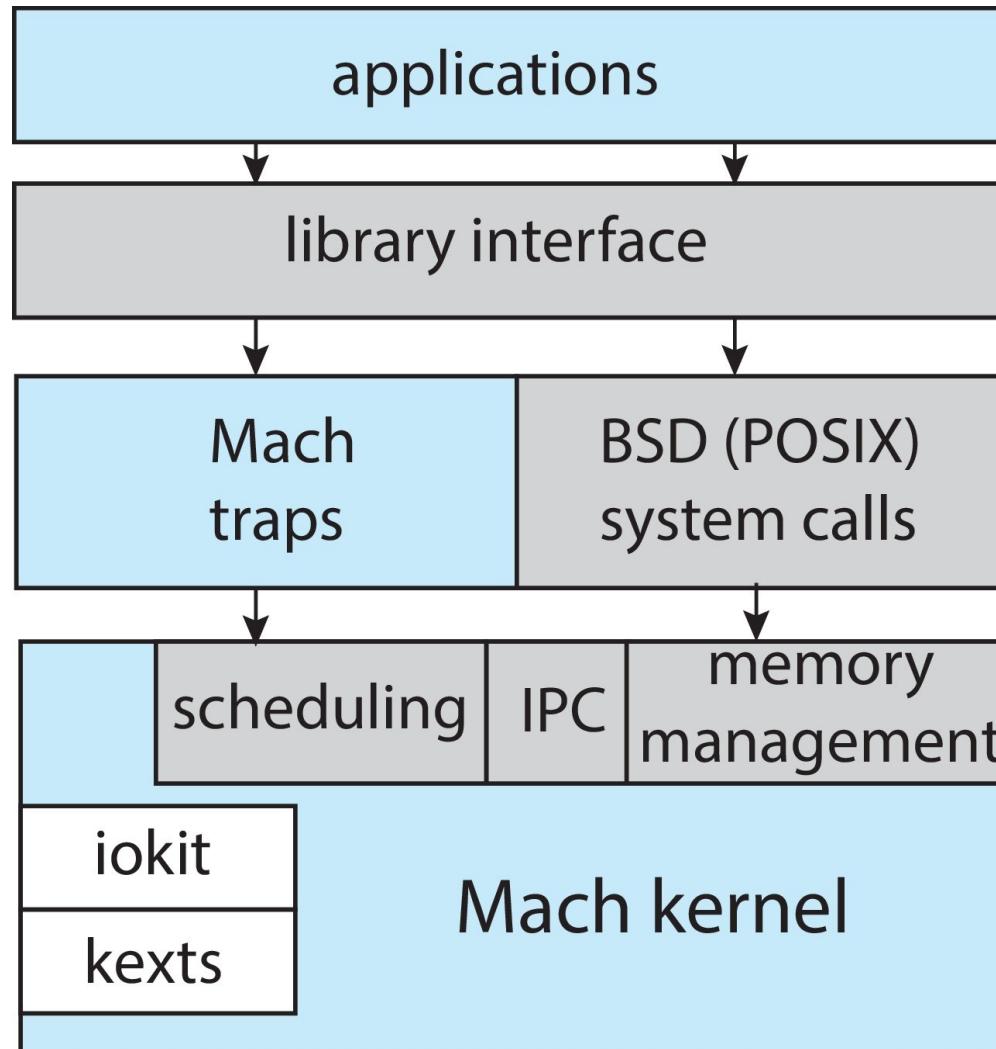


macOS and iOS Structure





Darwin





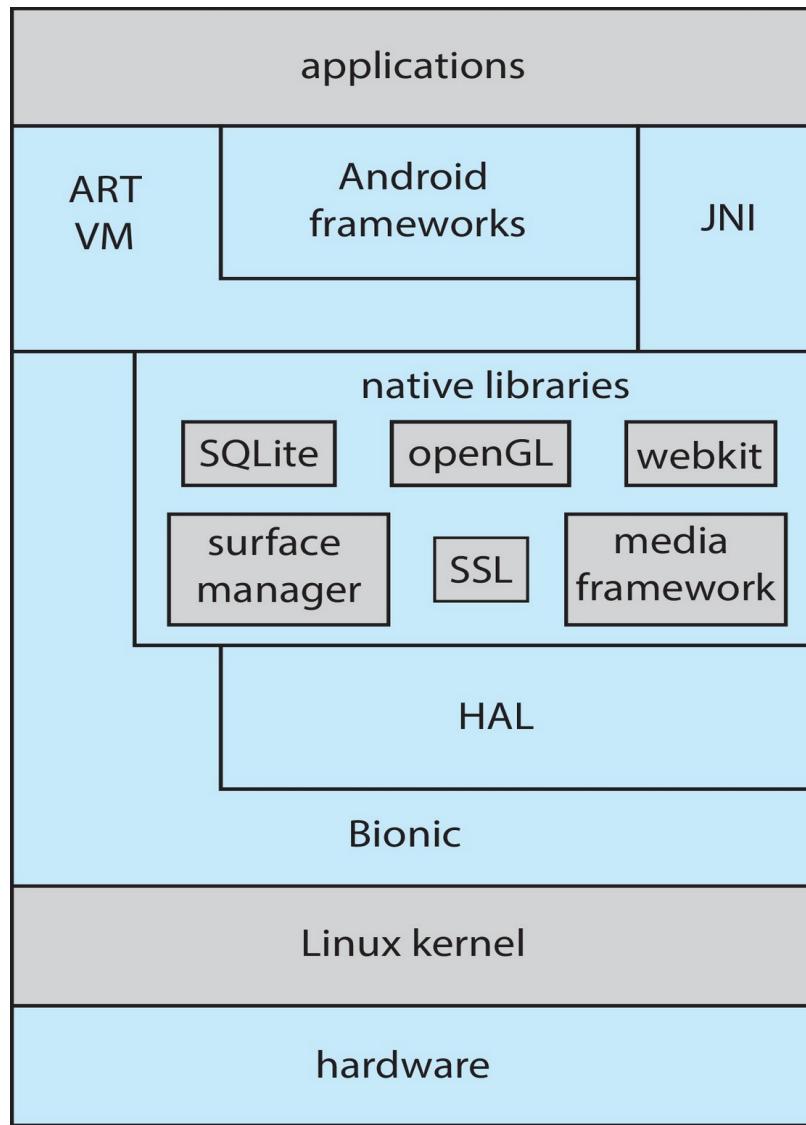
Android

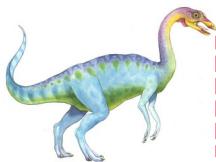
- Developed by Open Handset Alliance (mostly Google)
 - Open Source
- Similar stack to IOS
- Based on **Linux** kernel but modified
 - Provides process, memory, device-driver management
 - Adds power management
- Runtime environment includes core set of libraries and Dalvik virtual machine
 - Apps developed 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





Building and Booting an Operating System

- Operating systems generally designed to run on a class of systems with variety of peripherals
- Commonly, operating system already installed on purchased computer
 - But can build and install some other operating systems
 - If generating an operating system from scratch
 - ▶ Write the operating system source code
 - ▶ Configure the operating system for the system on which it will run
 - ▶ Compile the operating system
 - ▶ Install the operating system
 - ▶ Boot the computer and its new operating system





Building and Booting Linux

- Download Linux source code (<http://www.kernel.org>)
- Configure kernel via “make menuconfig”
- Compile the kernel using “make”
 - Produces `vmlinuz`, the kernel image
 - Compile kernel modules via “make modules”
 - Install kernel modules into `vmlinuz` via “make modules_install”
 - Install new kernel on the system via “make install”

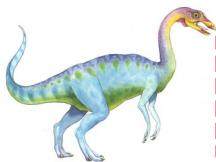




System Boot

- When power initialized on system, execution starts at a fixed memory location
- Operating system must be made available to hardware so hardware can start it
 - Small piece of code – **bootstrap loader**, **BIOS**, 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
 - Modern systems replace BIOS with **Unified Extensible Firmware Interface (UEFI)**
- Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then **running**
- Boot loaders frequently allow various boot states, such as single user mode





Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
- Also **performance tuning**
- 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

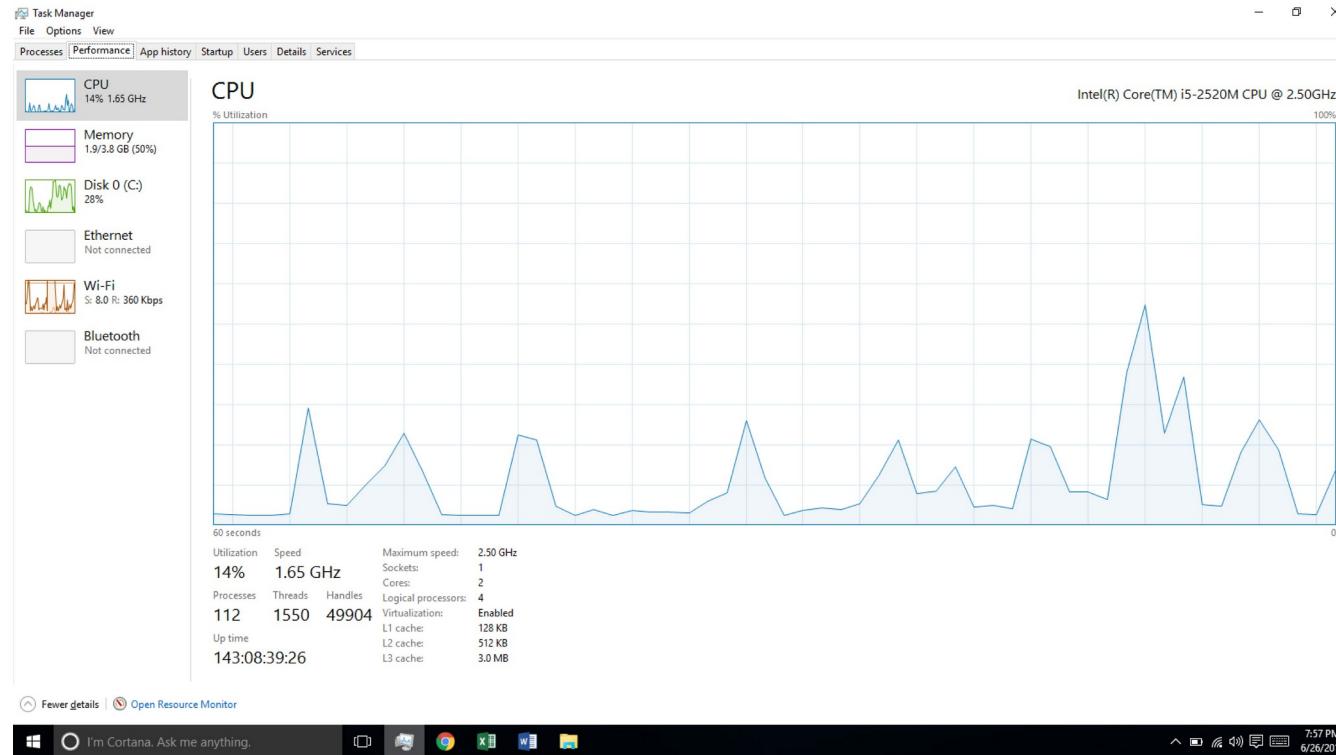
Kernighan's Law: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."





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

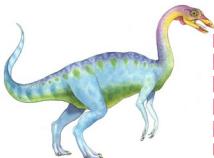




Tracing

- Collects data for a specific event, such as steps involved in a system call invocation
- Tools include
 - strace – trace system calls invoked by a process
 - gdb – source-level debugger
 - perf – collection of Linux performance tools
 - tcpdump – collects network packets



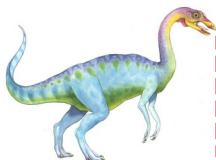


- Debugging interactions between user-level and kernel code nearly impossible without toolset that understands both and an instrument their actions
- BCC (BPF Compiler Collection) is a rich toolkit providing tracing features for Linux
 - See also the original DTrace
- For example, `disknoop nv traces disk I/O activity`

TIME(s)	T	BYTES	LAT(ms)
1946.29186700	R	8	0.27
1946.33965000	R	8	0.26
1948.34585000	W	8192	0.96
1950.43251000	R	4096	0.56
1951.74121000	R	4096	0.35

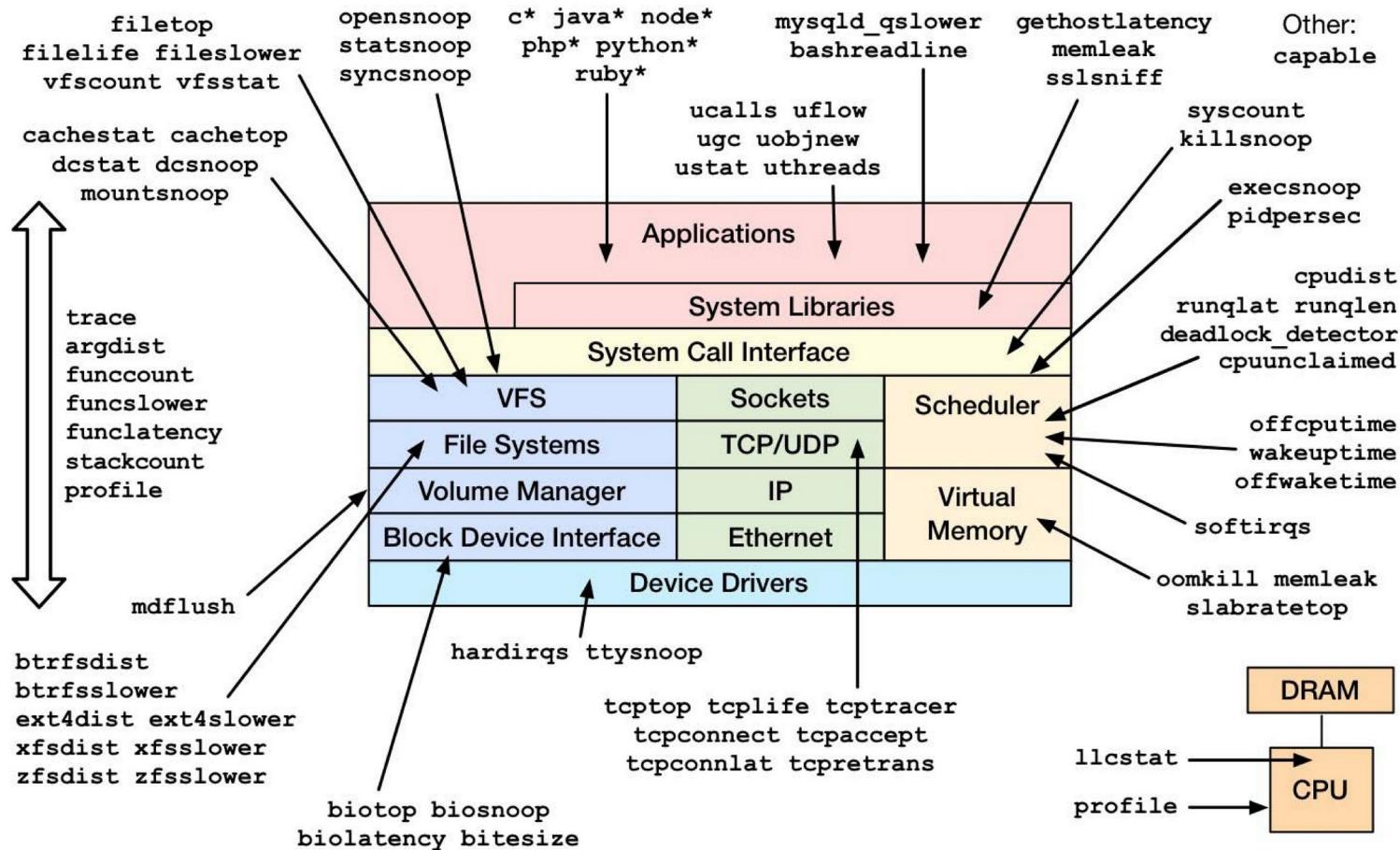
- Many other tools (next slide)





Linux bcc/BPF Tracing Tools

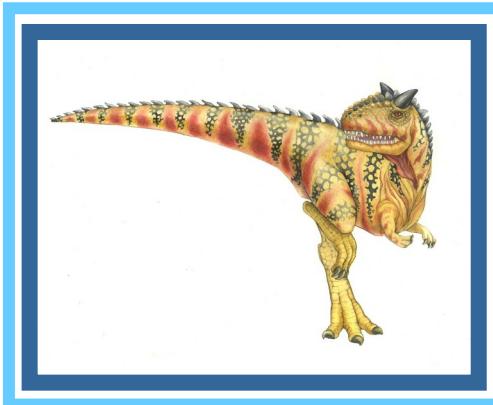
Linux bcc/BPF Tracing Tools



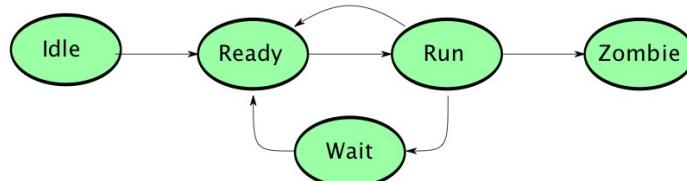
<https://github.com/iovisor/bcc#tools 2017>



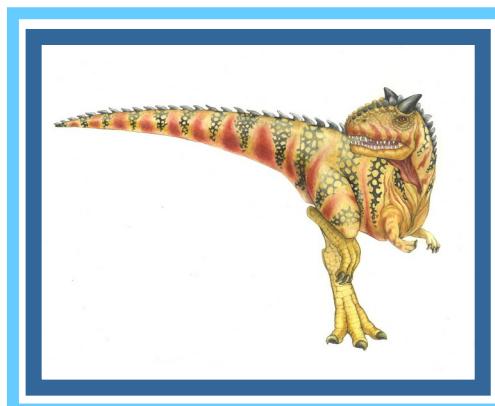
End of Chapter 2



Week 3



Processes





Week 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems





Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that uses pipes and POSIX shared memory to perform interprocess communication.
- Describe client-server communication using sockets and remote procedure calls.
- Design kernel modules that interact with the Linux operating system.





Process Concept

- An operating system executes a variety of programs that run as a process.
- **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

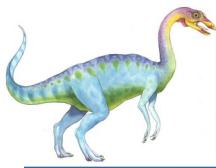




Process Concept (Cont.)

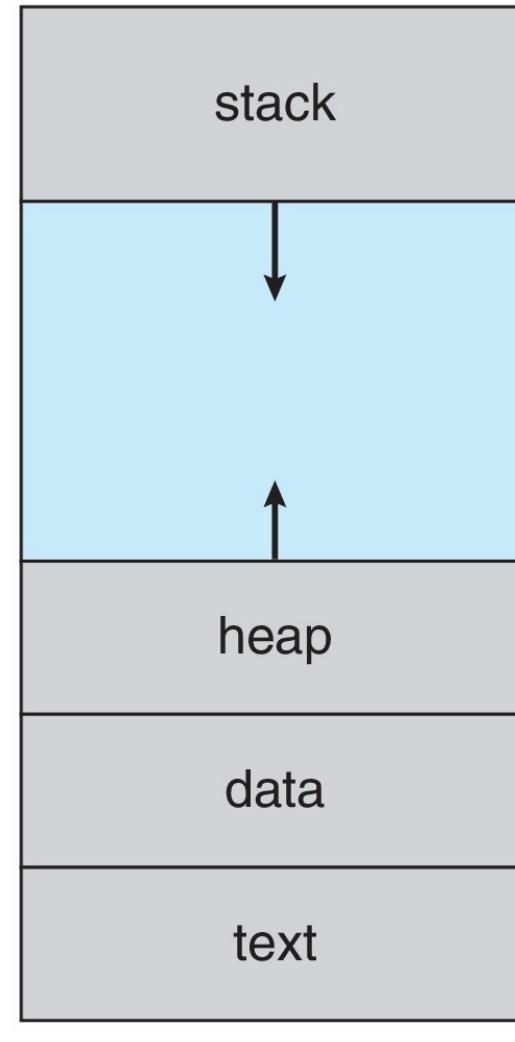
- Program is **passive** entity stored on disk (**executable file**); process is **active**
 - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
 - Consider multiple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





Process in Memory

max

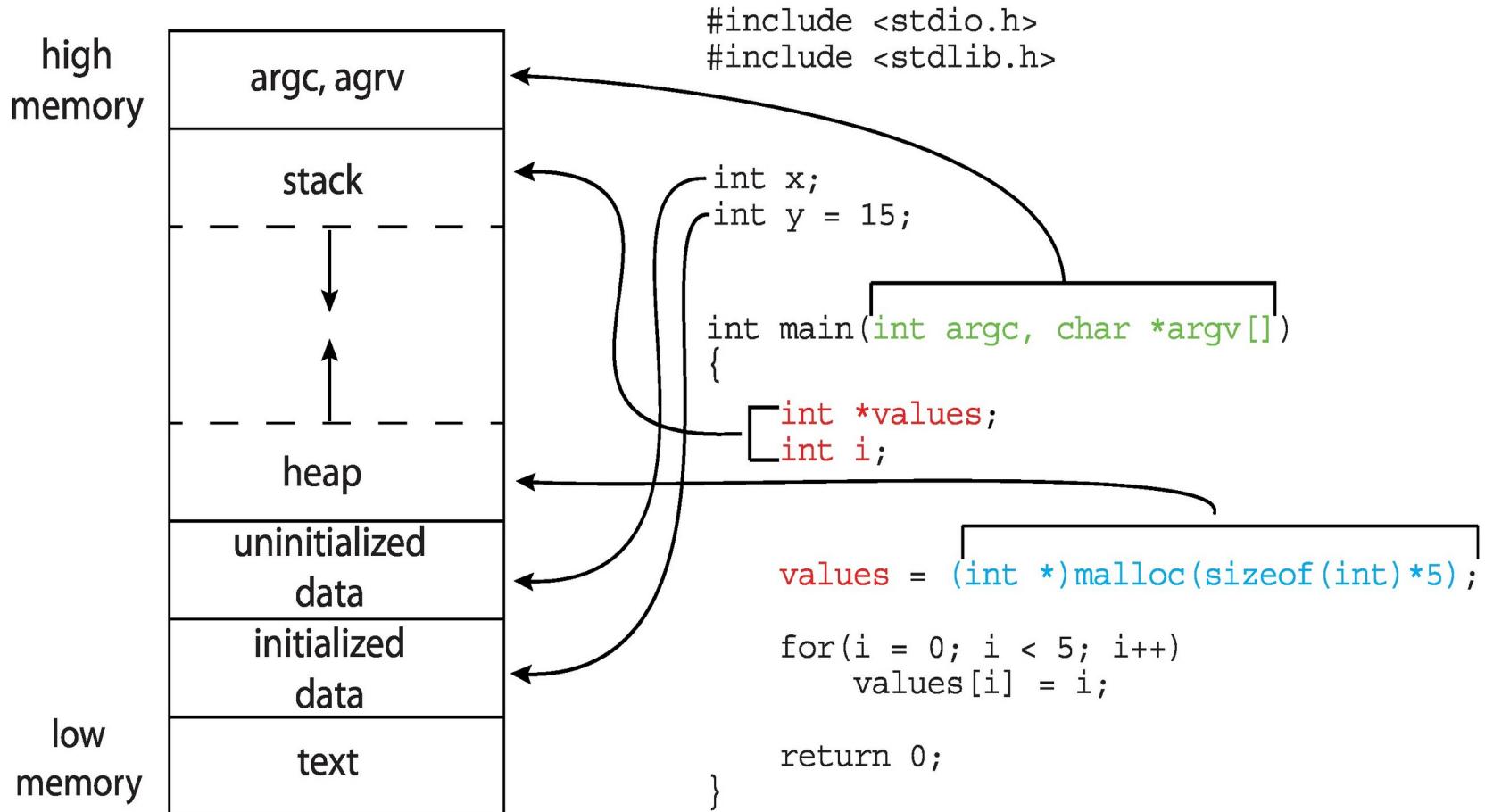


0





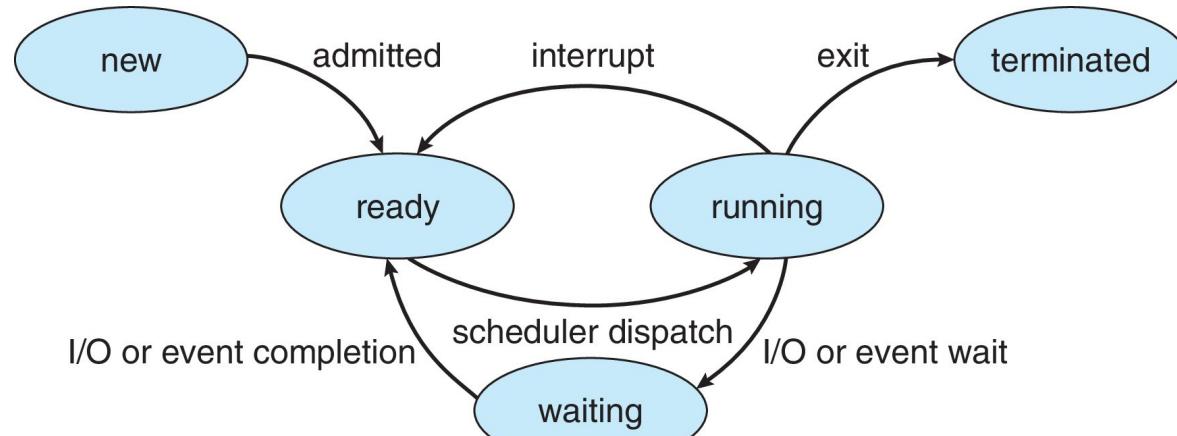
Memory Layout of a C Program





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

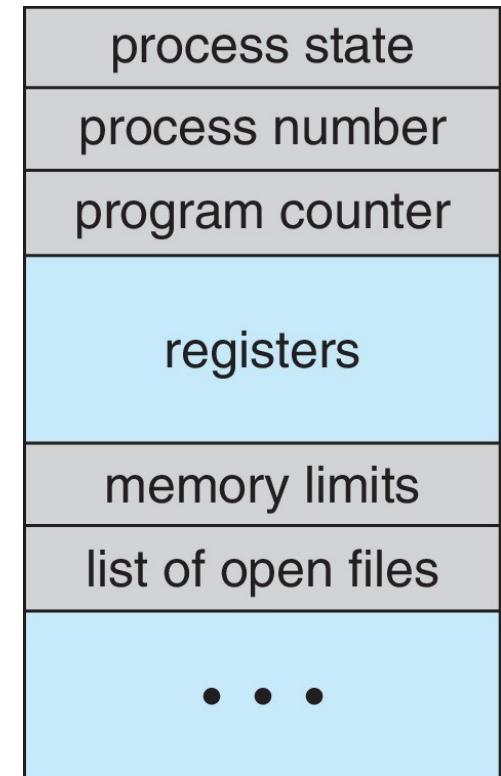




Process Control Block (PCB)

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**
- Must then have storage for thread details, multiple program counters in PCB
- Explore in detail in Chapter 4

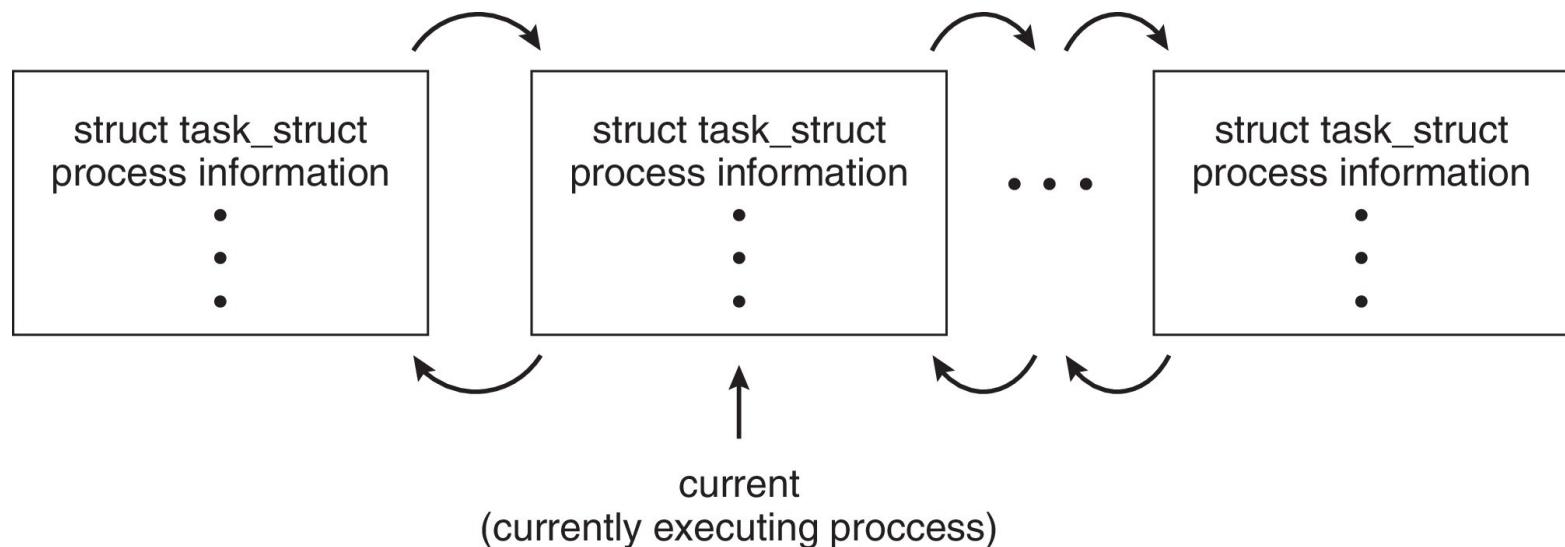




Process Representation in Linux

Represented by the C structure task_struct

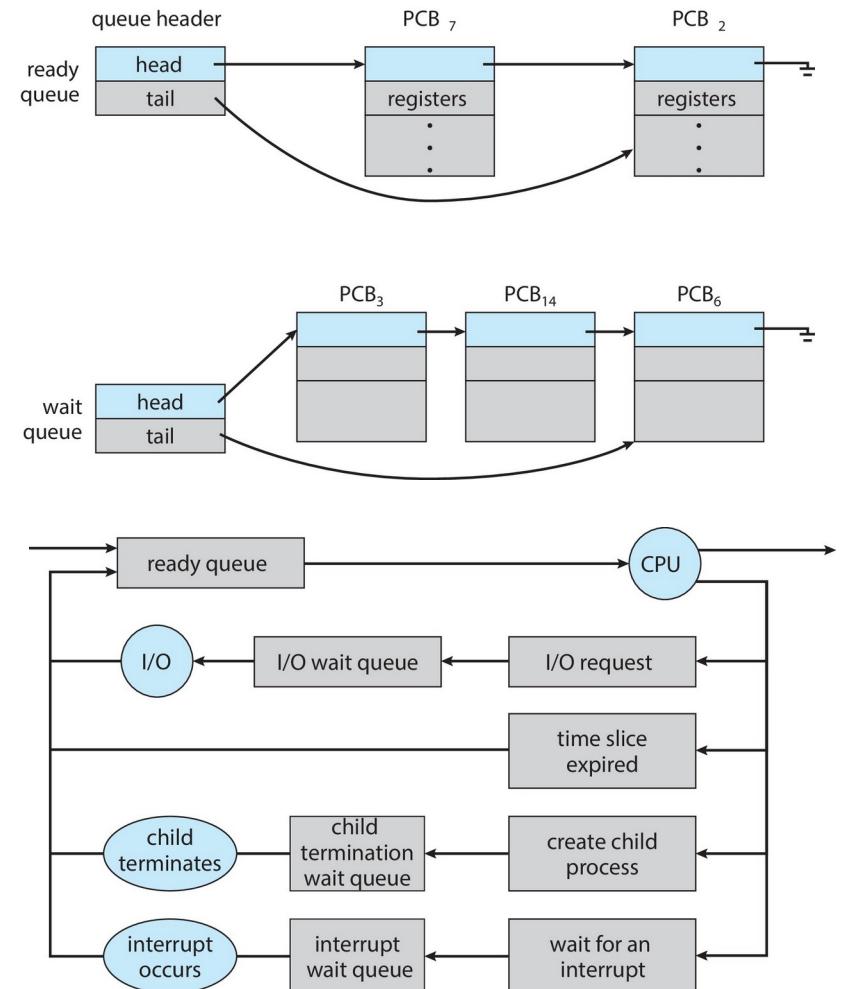
```
pid t_pid;          /* process identifier */  
long state;         /* state of the process */  
unsigned int time_slice; /* scheduling information */  
struct task_struct *parent; /* this process's parent */  
struct list_head children; /* this process's children */  
struct files_struct *files;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





Process Scheduling

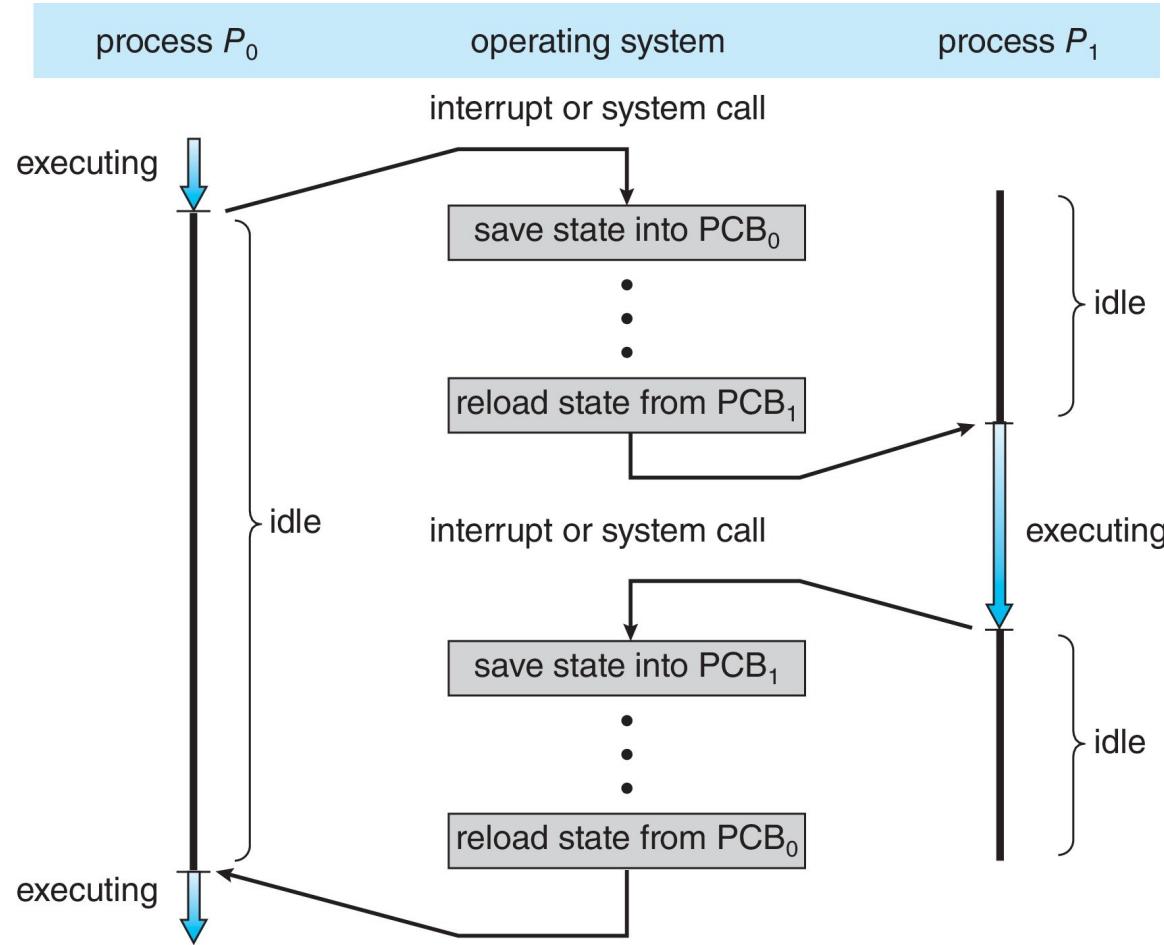
- Maximize CPU use, quickly switch processes onto CPU core
- **Process scheduler** selects among available processes for next execution on CPU core
- Maintains **scheduling queues** of processes
 - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
 - **Wait queues** – set of processes waiting for an event (i.e. I/O)
 - Processes migrate among the various queues

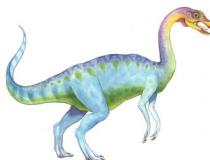




CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.

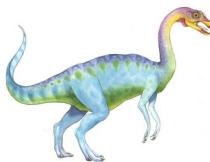




Context Switch

- 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 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





Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS 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





Operations on Processes

- System must provide mechanisms for:
 - process creation
 - process termination

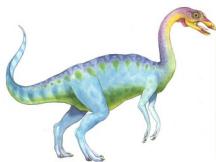




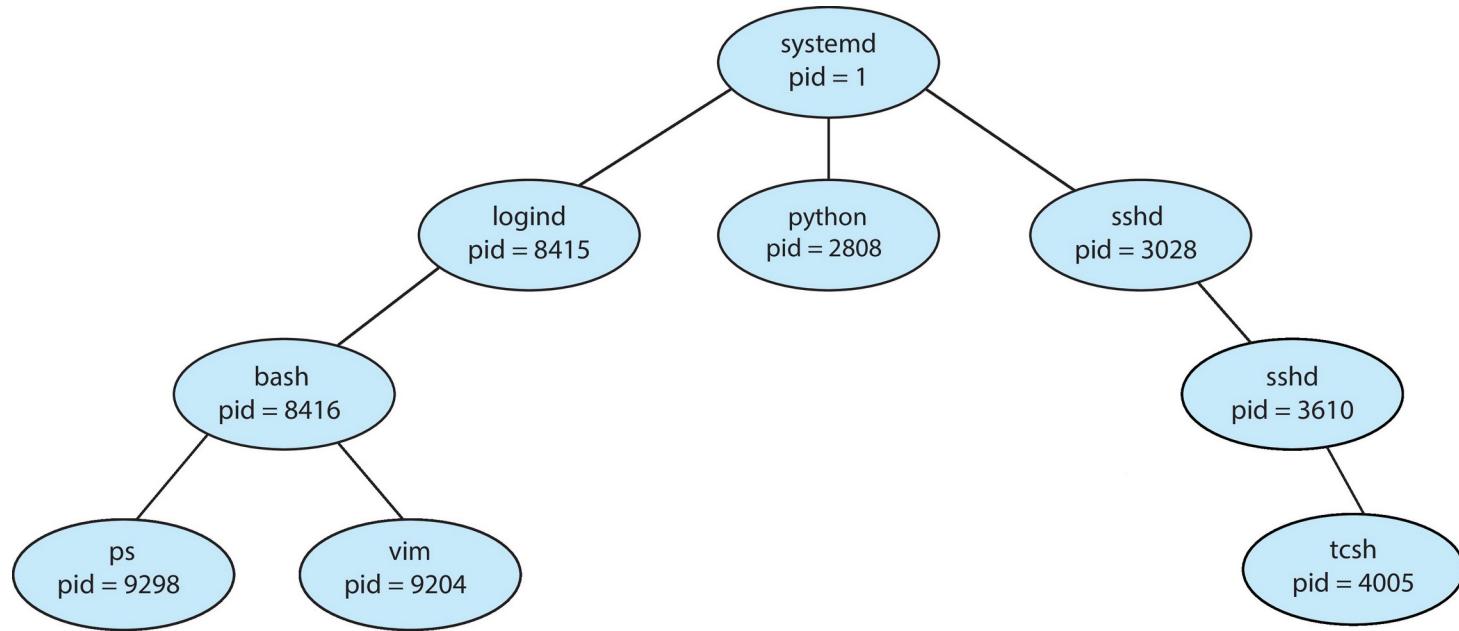
Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing 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





A Tree of Processes in Linux





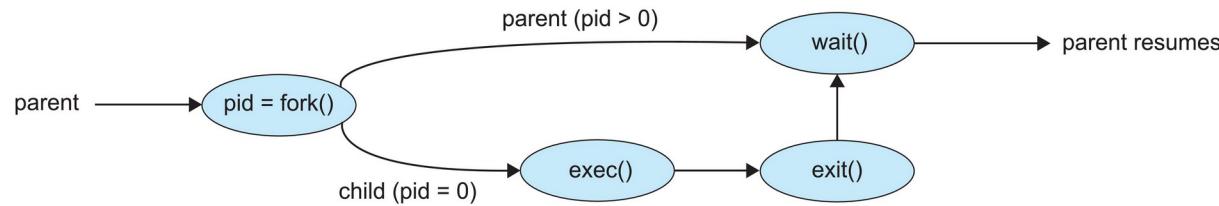
Process Creation (Cont.)

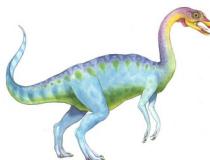
■ Address space

- Child duplicate of parent
- Child has a program loaded into it

■ UNIX examples

- **fork()** system call creates new process
- **exec()** system call used after a **fork()** to replace the process' memory space with a new program
- Parent process calls **wait()** for the child to terminate





C Program Forking Separate Process

```
#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);
}
```





Process Termination

- Process executes last statement and then asks the operating system to delete it using the **exit()** system call.
 - Returns status data from child to parent (via **wait()**)
 - Process' resources are deallocated by operating system
- 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





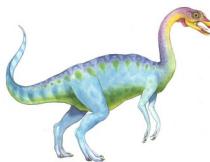
Process Termination

- Some operating systems do not allow child to exists 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 a **zombie**
- If parent terminated without invoking **wait** , process is an **orphan**

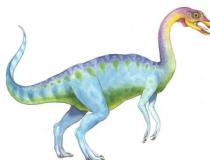




Android Process Importance Hierarchy

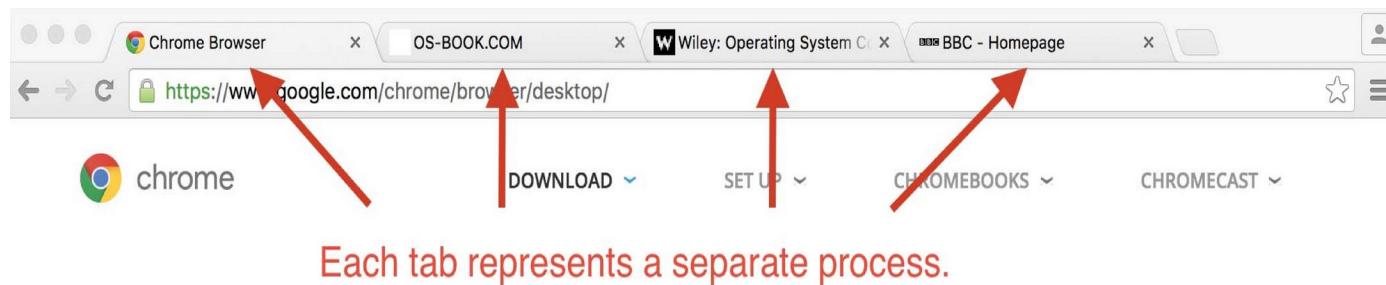
- Mobile operating systems often have to terminate processes to reclaim system resources such as memory. From **most** to **least** important:
 - Foreground process
 - Visible process
 - Service process
 - Background process
 - Empty process
- Android will begin terminating processes that are least important.

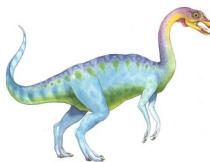




Multiprocess Architecture – Chrome Browser

- 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





Interprocess Communication

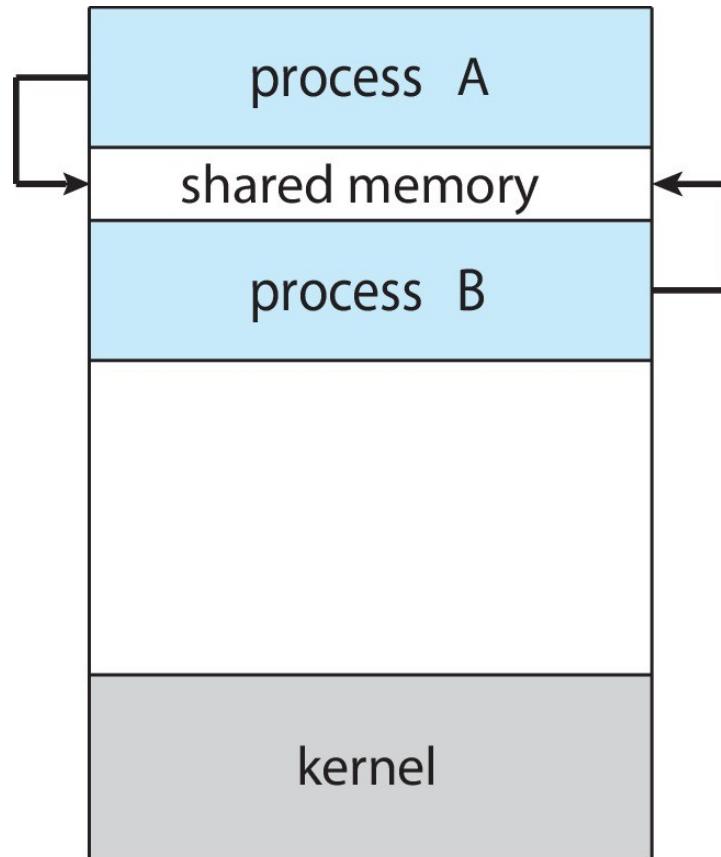
- Processes within a system may be *independent* or *cooperating*
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
 - Information sharing
 - Computation speedup
 - Modularity
 - Convenience
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
 - **Shared memory**
 - **Message passing**





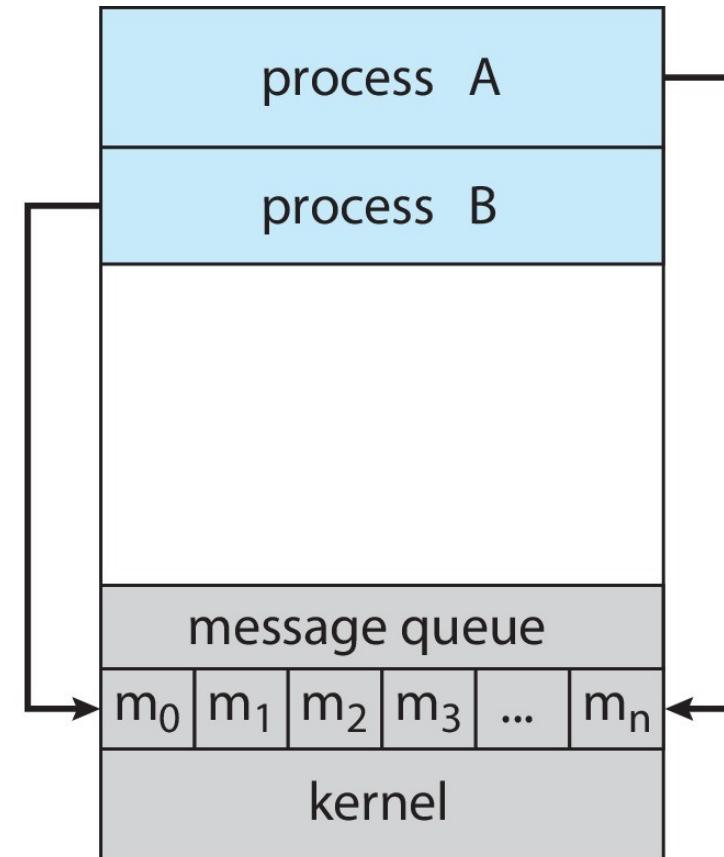
Communications Models

(a) Shared memory.



(a)

(b) Message passing.



(b)





Cooperating Processes

- ***Independent*** process cannot affect or be affected by the execution of another process
- ***Cooperating*** process can affect or be affected by the execution of another process
- Advantages of process cooperation
 - Information sharing
 - Computation speed-up
 - Modularity
 - Convenience





Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
 - **unbounded-buffer** places no practical limit on the size of the buffer
 - **bounded-buffer** assumes that there is a fixed buffer size





Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users 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 is discussed in great details in Chapters 6 & 7.





Bounded-Buffer – Shared-Memory Solution

- Shared data

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

- Solution is correct, but can only use **BUFFER_SIZE-1** elements





Producer Process – Shared Memory

```
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;
}
```





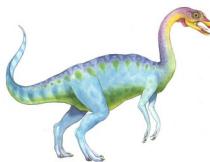
Consumer Process – Shared Memory

```
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 */
}
```

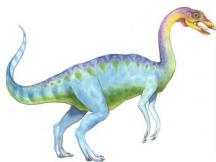




Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other 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?





Message Passing (Cont.)

■ 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

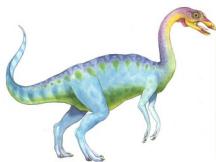




Indirect Communication

- 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
- 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

■ Operations

- create a new mailbox (port)
- send and receive messages through mailbox
- destroy 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

- 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.

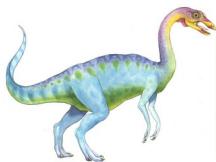




Synchronization

- 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**

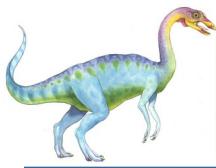




Producer – Shared Memory

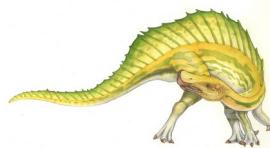
```
message next_produced;  
  
while (true) {  
    /* produce an item in next_produced */  
  
    send(next_produced);  
}
```





Consumer– Shared Memory

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





Buffering

- Queue of messages attached to the link.
- Implemented in one of 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





Examples of IPC Systems - POSIX

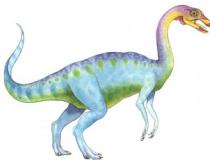
■ POSIX Shared Memory

- Process first creates shared memory segment
`shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`
- Also used to open an existing segment
- Set the size of the object

```
ftruncate(shm_fd, 4096);
```

- Use `mmap()` to memory-map a file pointer to the shared memory object
- Reading and writing to shared memory is done by using the pointer returned by `mmap()`.





IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* strings written to shared memory */
    const char *message_0 = "Hello";
    const char *message_1 = "World!";

    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory obect */
    void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

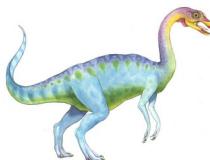
    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```





IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory obect */
    void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s", (char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

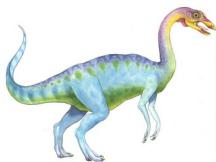




Examples of IPC Systems - Mach

- Mach communication is message based
 - Even system calls are messages
 - Each task gets two ports at creation- Kernel and Notify
 - Messages are sent and received using the `mach_msg()` function
 - Ports needed for communication, created via
`mach_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





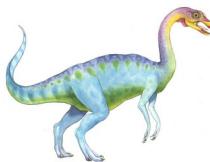
Mach Messages

```
#include<mach/mach.h>

struct message {
    mach_msg_header_t header;
    int data;
};

mach_port_t client;
mach_port_t server;
```





Mach Message Passing - Client

```
/* Client Code */

struct message message;

// construct the header
message.header.msgh_size = sizeof(message);
message.header.msgh_remote_port = server;
message.header.msgh_local_port = client;

// send the message
mach_msg(&message.header, // message header
         MACH_SEND_MSG, // sending a message
         sizeof(message), // size of message sent
         0, // maximum size of received message - unnecessary
         MACH_PORT_NULL, // name of receive port - unnecessary
         MACH_MSG_TIMEOUT_NONE, // no time outs
         MACH_PORT_NULL // no notify port
);
```





Mach Message Passing - Server

```
/* Server Code */

struct message message;

// receive the message
mach_msg(&message.header, // message header
         MACH_RCV_MSG, // sending a message
         0, // size of message sent
         sizeof(message), // maximum size of received message
         server, // name of receive port
         MACH_MSG_TIMEOUT_NONE, // no time outs
         MACH_PORT_NULL // no notify port
);
```

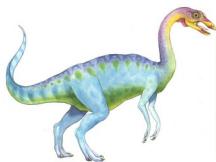




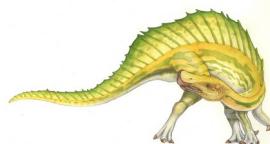
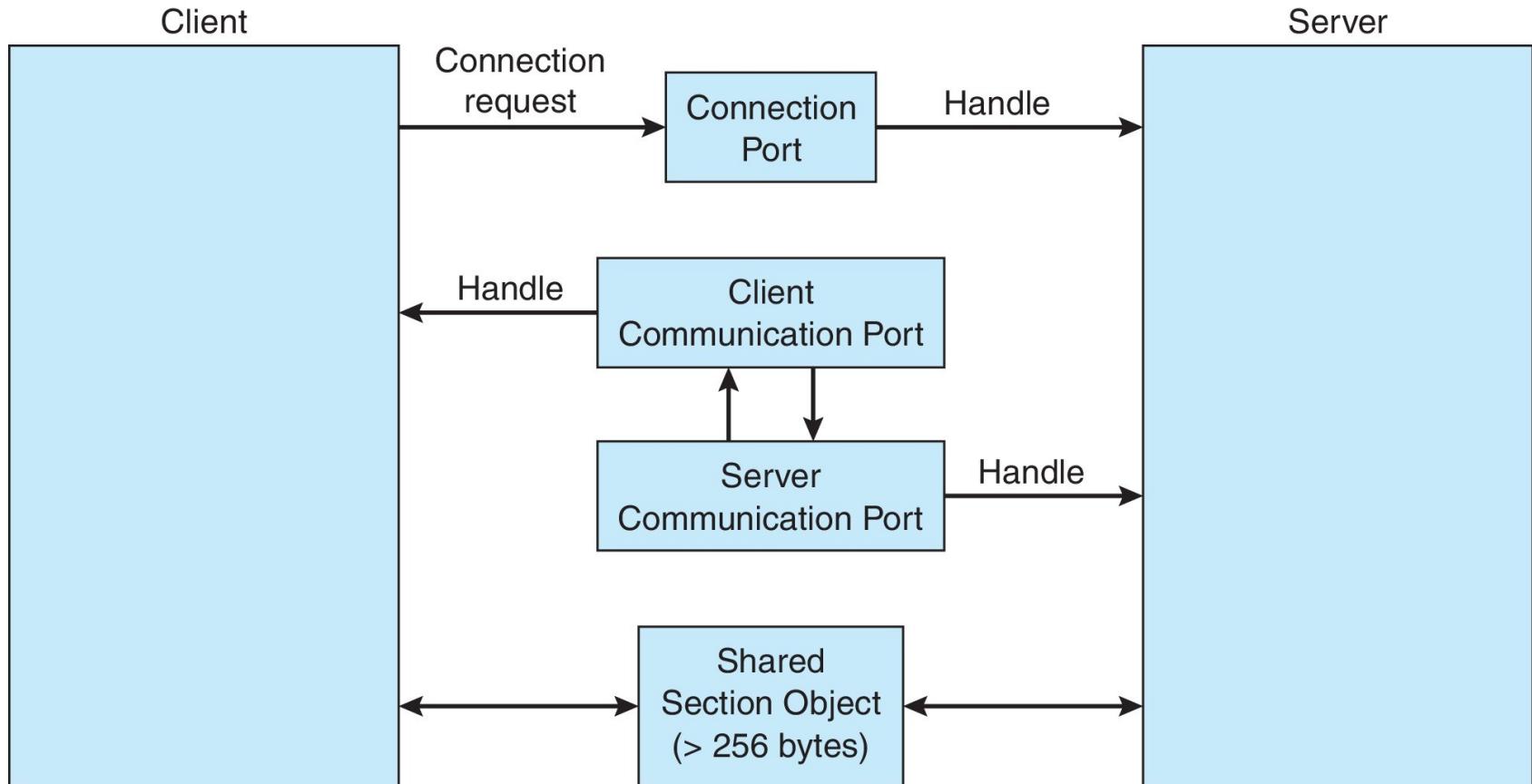
Examples of IPC Systems – Windows

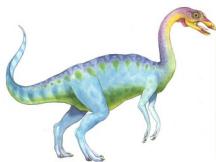
- Message-passing centric via **advanced local procedure call (LPC)** facility
 - Only works between processes on the same system
 - Uses ports (like mailboxes) to establish and maintain communication channels
 - Communication works as follows:
 - ▶ The client opens a handle to the subsystem's **connection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





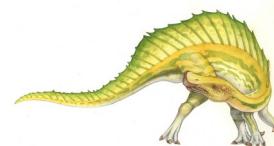
Local Procedure Calls in Windows





Pipes

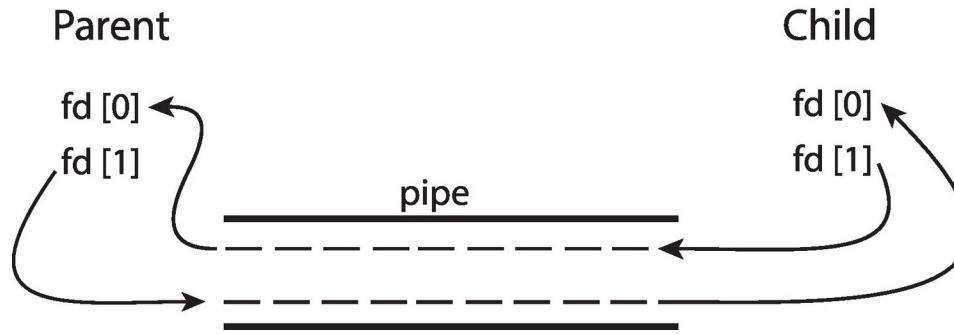
- Acts as a conduit allowing two processes to communicate
- Issues:
 - Is communication unidirectional or bidirectional?
 - In the case of two-way communication, is it half or full-duplex?
 - Must there exist a relationship (i.e., **parent-child**) between the communicating processes?
 - Can the pipes 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 communication 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 therefore unidirectional
- Require parent-child relationship between communicating processes



- Windows calls these **anonymous pipes**

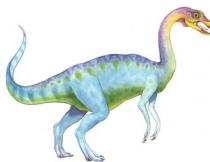




Named Pipes

- 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

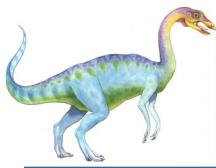




Communications in Client-Server Systems

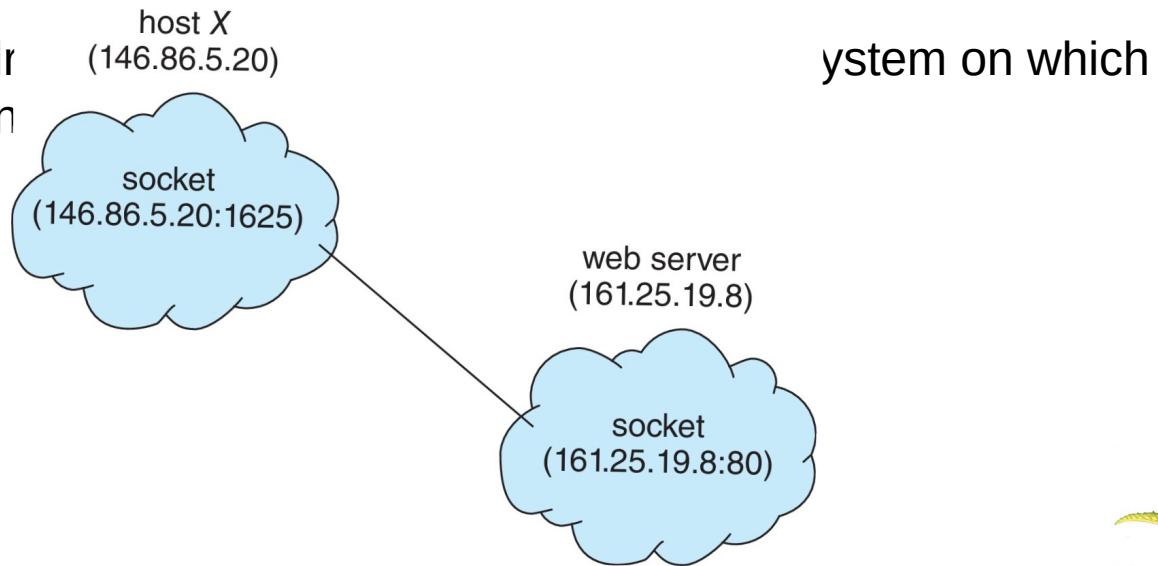
- Sockets
- Remote Procedure Calls





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 used when process is runn





Sockets in C

- Three types of sockets
 - **Connection-oriented (TCP)**
 - **Connectionless (UDP)**
 - **MulticastSocket** class– data can be sent to multiple recipients





“Date” server in C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <time.h>

int main() {
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in serv_addr = {0};
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(6013);

    bind(sockfd, (struct sockaddr*)&serv_addr,
        sizeof(serv_addr));

    listen(sockfd, 1);

    printf("Server running on port 6013...\n");
}
```

```
while (1) {
    int clientfd;
    clientfd = accept(sockfd, NULL, NULL);

    time_t now = time(NULL);
    char* date = ctime(&now);
    write(clientfd, date, strlen(date));

    close(clientfd);
}

close(sockfd);
return 0;
}
```





The equivalent Date client in C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

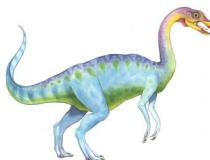
int main() {
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in serv_addr = {0};
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(6013);
    inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr);

    connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));

    char buffer[256];
    read(sockfd, buffer, sizeof(buffer));
    printf("%s", buffer);

    close(sockfd);
    return 0;
}
```





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
- 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)**





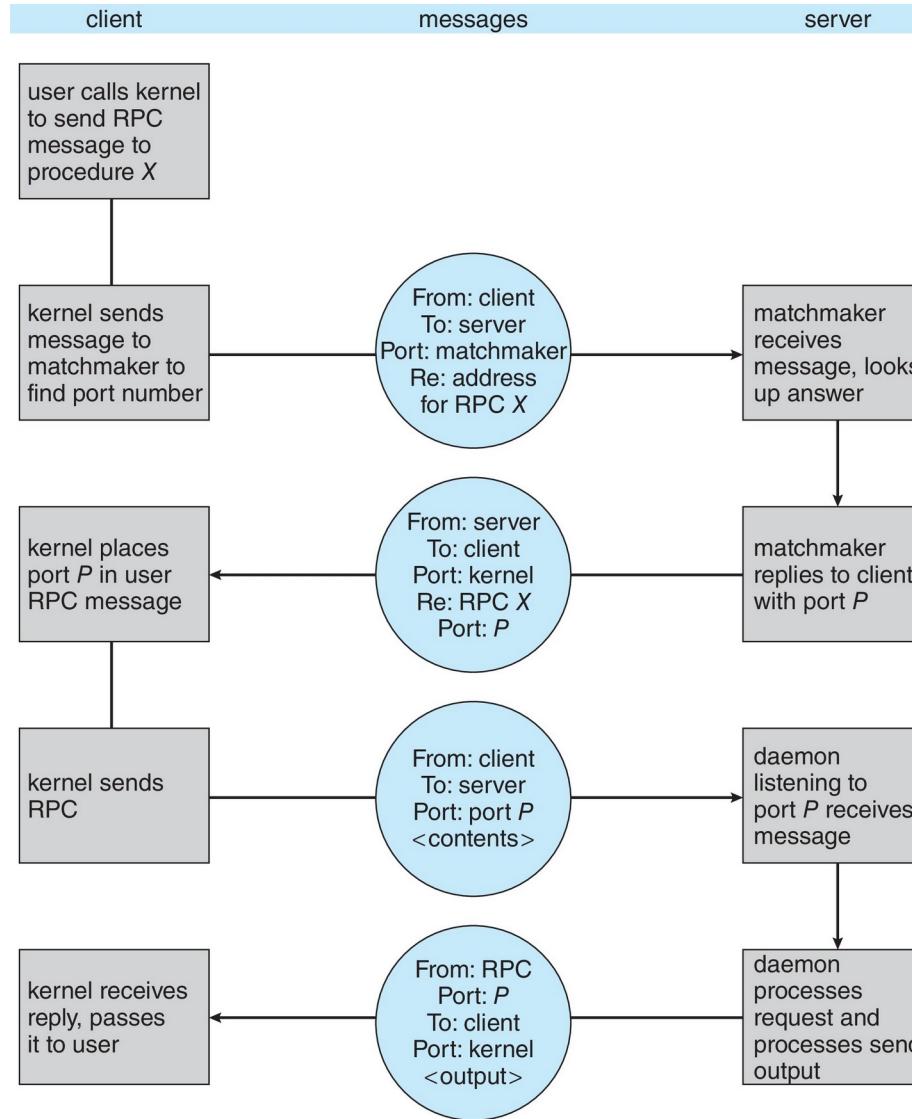
Remote Procedure Calls (Cont.)

- Data representation handled via **External Data Representation (XDL)** format to account for different architectures
 - **Big-endian** and **little-endian**
- 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

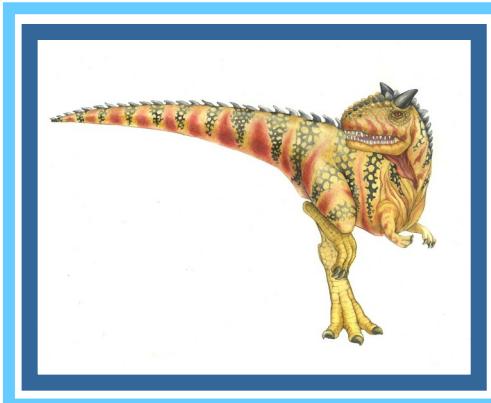


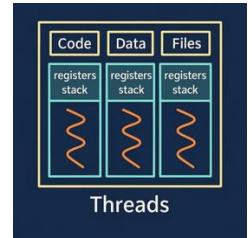


Execution of RPC



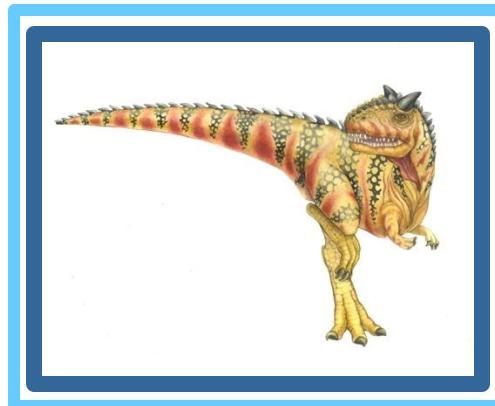
End of Chapter 3





Week 4

Threads & Concurrency





Chapter 4: Threads

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





Objectives

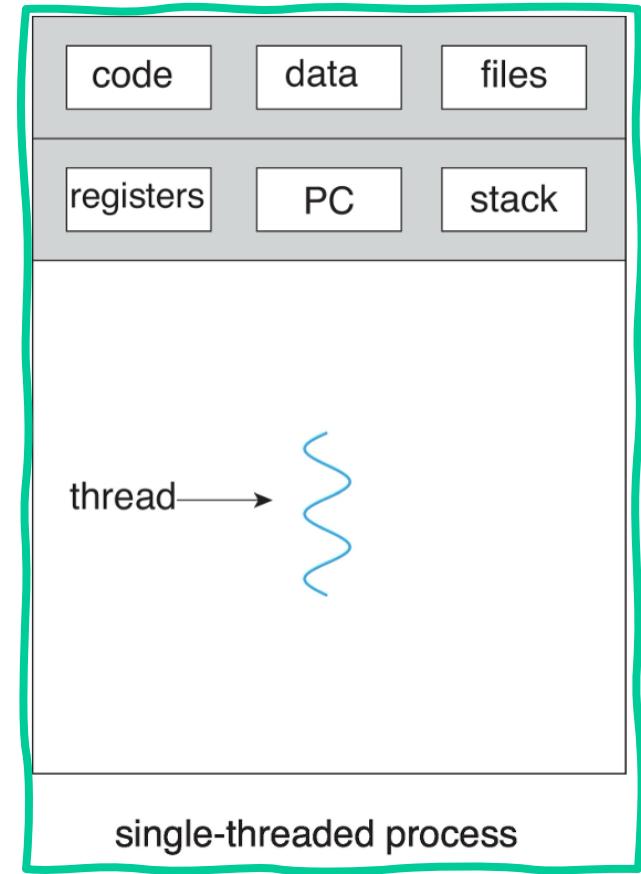
- ◆ Identify the basic components of a thread, and contrast threads and processes
- ◆ Describe the benefits and challenges of designing multithreaded applications
- ◆ Illustrate different approaches to implicit threading including thread pools, fork-join, and Grand Central Dispatch
- ◆ Describe how the Windows and Linux operating systems represent threads
- ◆ Design multithreaded applications using the Pthreads, Java, and Windows threading APIs





Single Threaded Process

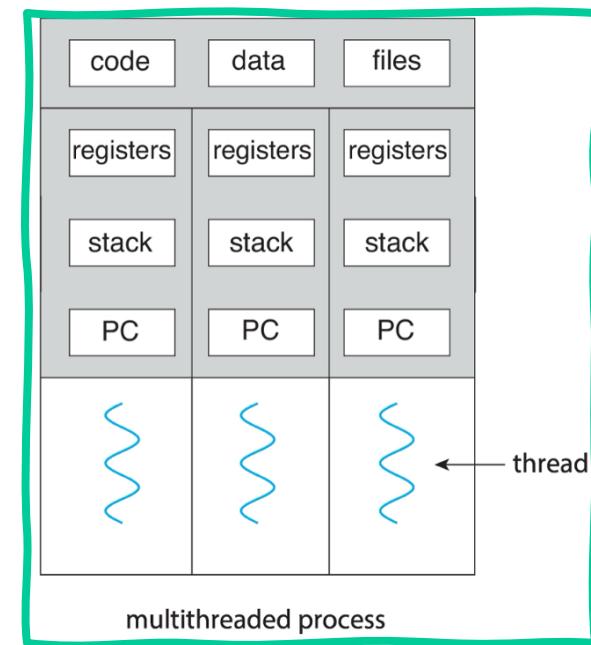
- ◆ A traditional process is by default single-threaded
- ◆ It has a single path of execution through the program code
- ◆ It executes only one instruction at a time
- ◆ It requires only one copy of the Register File and Stack
- ◆ A single-threaded process contains a single instance of each of the following:
 - ◆ Code Section
 - ◆ Data Section (Global variables)
 - ◆ System Resources (Open files)
 - ◆ Program Stack (Function call/return, Local variables)
 - ◆ CPU Registers (State)





Multithreaded Processes

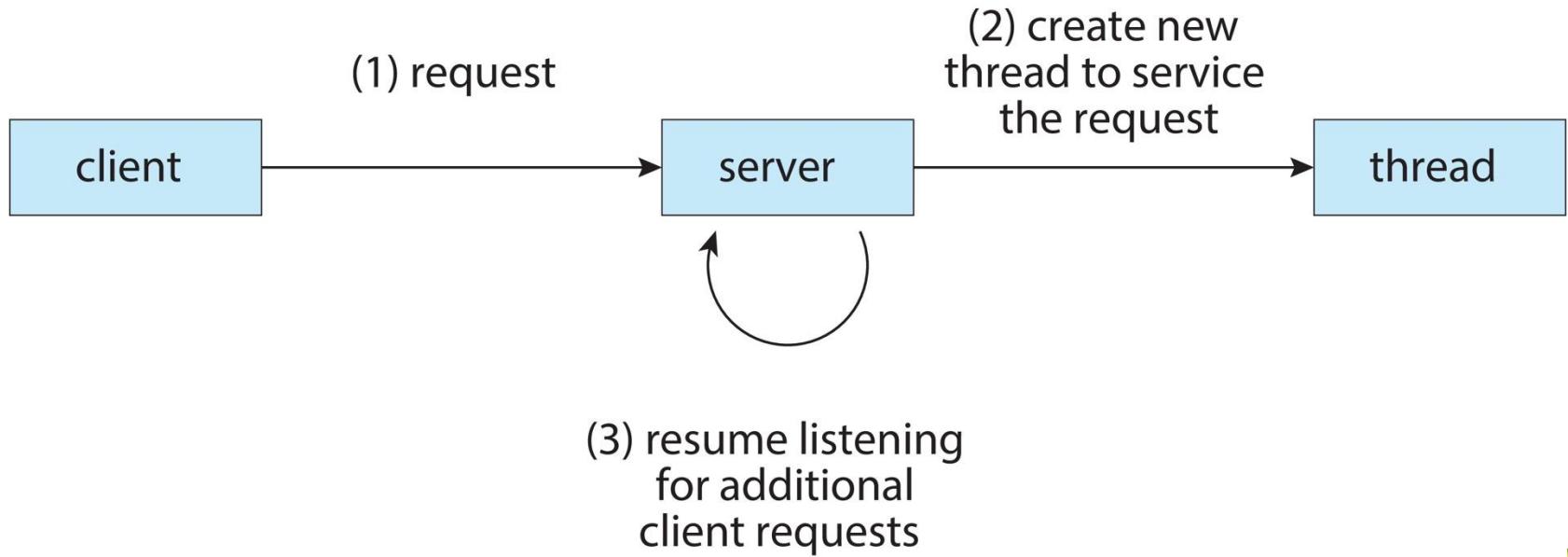
- ◆ The operating system supports multiple threads of execution within a single process
- ◆ Allows the process to handle multiple tasks concurrently, such as a web server serving multiple client requests simultaneously
- ◆ Threads run within a process, and a process will have one or more threads
- ◆ Threads that belong to the same process share several resources
- ◆ Sharing of the Code and Data sections means all threads use the same program logic and global variables
- ◆ Private copies of the Program Counter, Registers, and Stack enable each thread to:
 - ◆ Execute a different logical path within the original program code
 - ◆ Maintain its own record of local variables and function calls (its own stack)
 - ◆ Exist in its own execution state (e.g., Thread 1 running, Thread 2 waiting)





Multithreaded Server Architecture

- ◆ A **multithreaded server architecture** is a server design pattern where a single process handles multiple client connections or tasks simultaneously by creating and managing multiple execution threads.
- ◆ This design significantly improves a server's performance and responsiveness compared to a single-threaded model.





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 multicore architectures





Multicore Programming

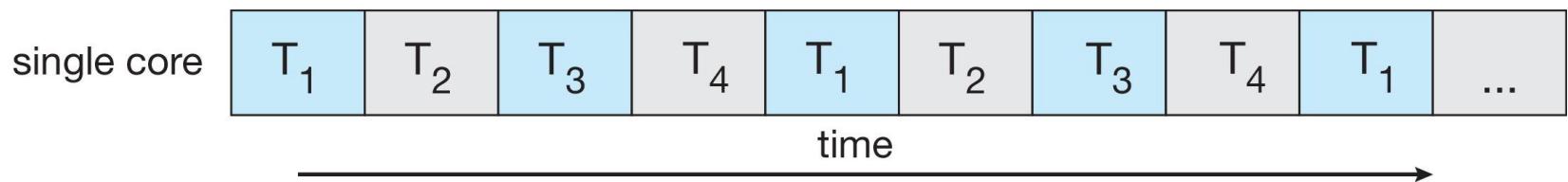
- ◆ **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
 - ◆ **Dividing activities**
 - ◆ **Balance**
 - ◆ **Data splitting**
 - ◆ **Data dependency**
 - ◆ **Testing and debugging**
- ◆ **Parallelism** implies a system can perform more than one task simultaneously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





Concurrency

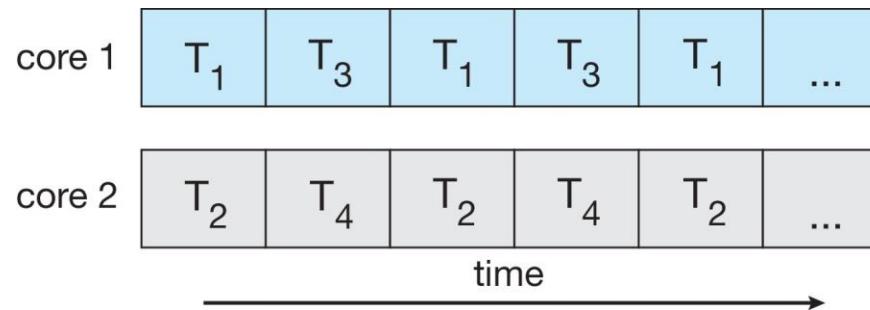
- ◆ Concurrency is about the structure of a program and its ability to manage multiple tasks at once, allowing them to make progress over overlapping time periods.
- ◆ **Goal:** Improve responsiveness and manage multiple independent operations (especially those that involve waiting, like I/O).
- ◆ **Execution:**
 - ◆ Tasks do not necessarily run at the same instant.
 - ◆ On a single CPU core, concurrency is achieved by the OS rapidly switching between tasks (Context Switching), giving the illusion of simultaneous execution.





Parallelism

- ◆ Parallelism is about the actual, simultaneous execution of multiple computations.
- ◆ **Goal:** Improve performance and achieve computational speedup by dividing a large workload.
- ◆ **Execution:** Tasks literally run at the exact same time. This requires hardware with multiple processing units (e.g., a multi-core CPU, multiple CPUs, or a GPU).



Nice video about Concurrency vs Parallelism

<https://www.youtube.com/watch?v=RIM9AfWf1WU&t=20s>





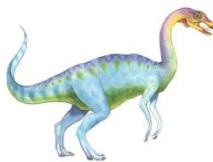
Concurrency vs Parallelism

Feature	Concurrency	Parallelism
Concept	Managing multiple tasks.	Executing multiple tasks simultaneously.
Goal	Responsiveness & resource utilization.	Speed & throughput (computational power).
Hardware	Can be achieved with one CPU core.	Requires multiple processing units (cores).
Execution	Tasks are interleaved (switched rapidly).	Tasks are literally simultaneous .
Relationship	Necessary for parallelism.	A way to achieve concurrency.

Nice video about Concurrency vs Parallelism

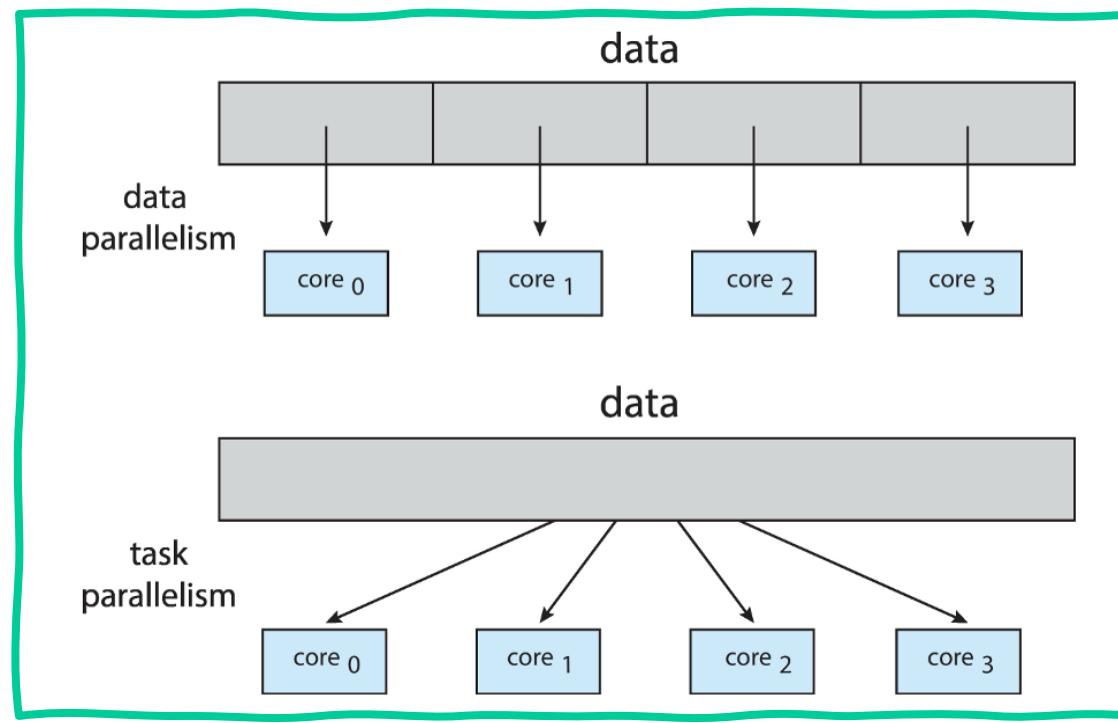
<https://www.youtube.com/watch?v=RIM9AfWf1WU&t=20s>





Multicore Programming

- ◆ 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





Amdahl's Law

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

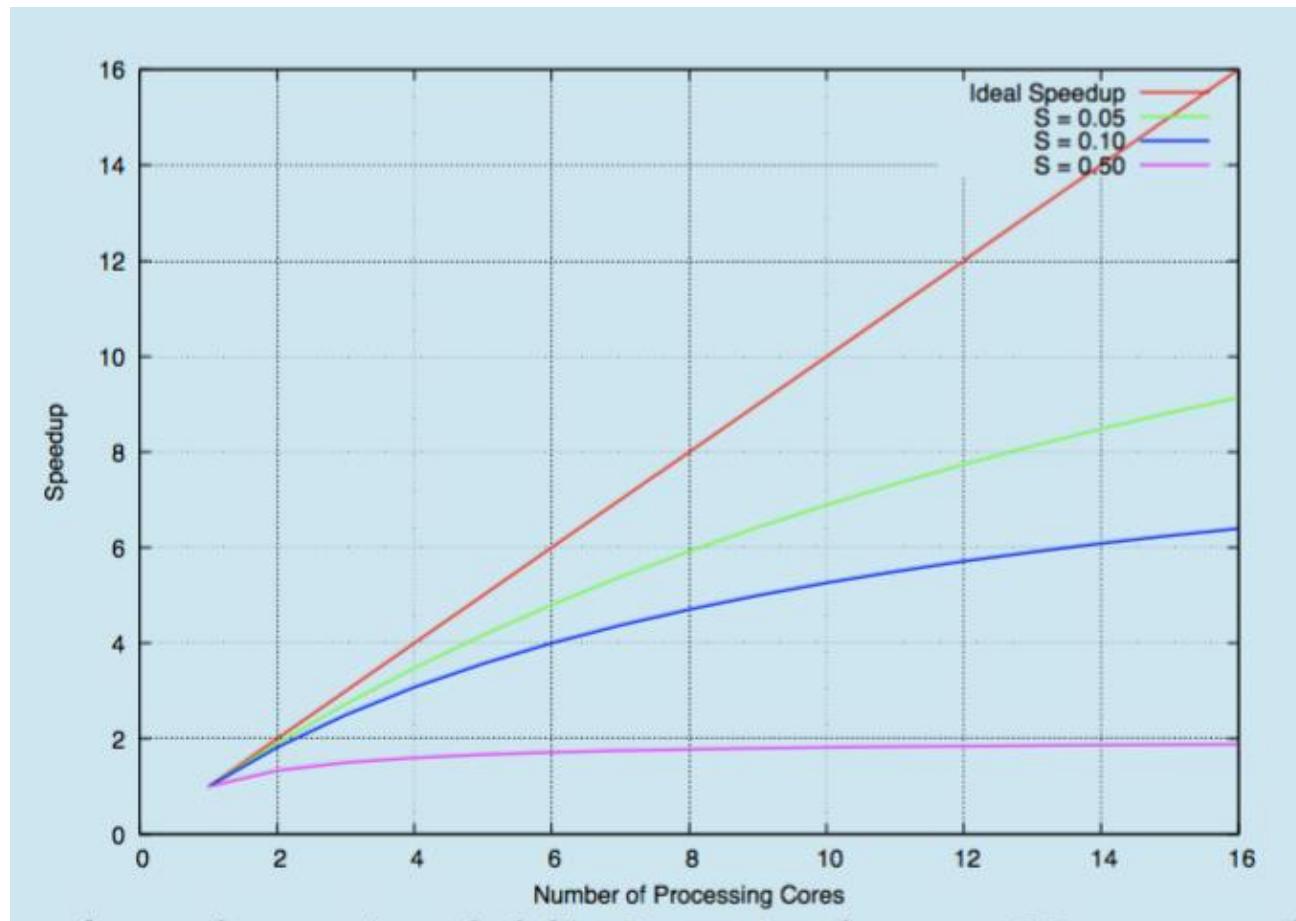
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- ◆ That is, if application is 75% parallel / 25% serial (S), moving from 1 to 2 cores (N) 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?





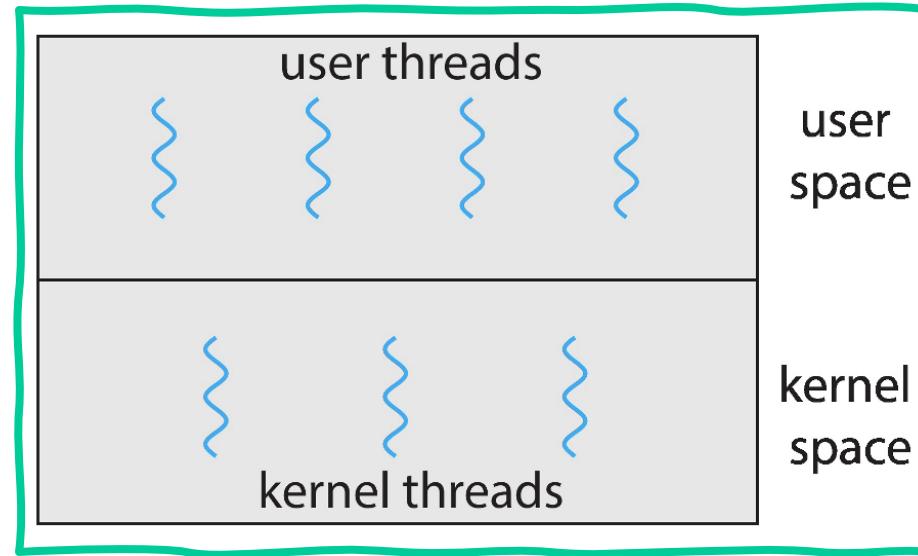
Amdahl's Law





User Threads and Kernel Threads

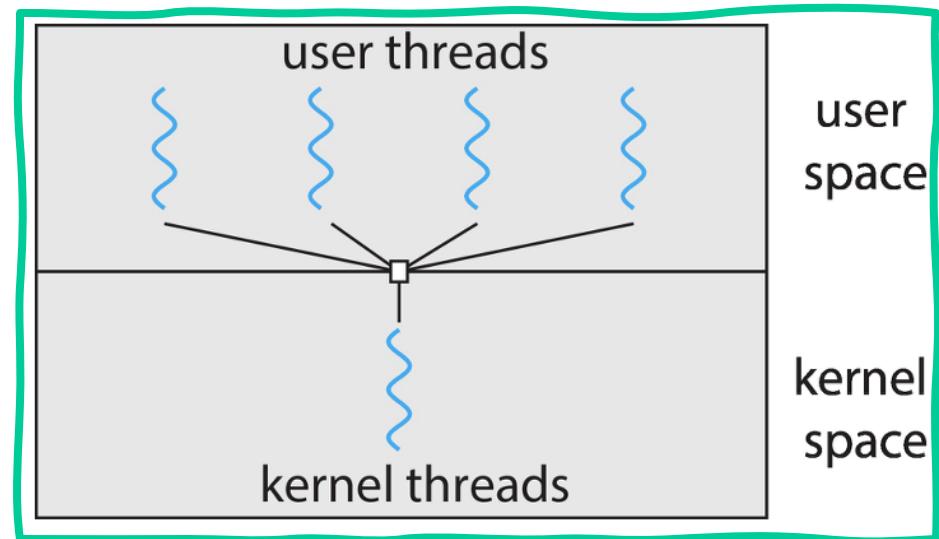
- ◆ **User threads** - management done by user-level threads library
- ◆ Three primary thread libraries:
 - ◆ POSIX **Pthreads**
 - ◆ Windows threads
 - ◆ Java threads
- ◆ **Kernel threads** - Supported by the Kernel
- ◆ Examples – virtually all general purpose operating systems, including:
 - ◆ Windows
 - ◆ Linux
 - ◆ Mac OS X
 - ◆ iOS
 - ◆ Android

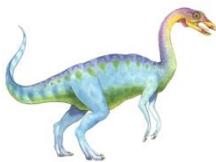




Multithreading Models: Many-to-One

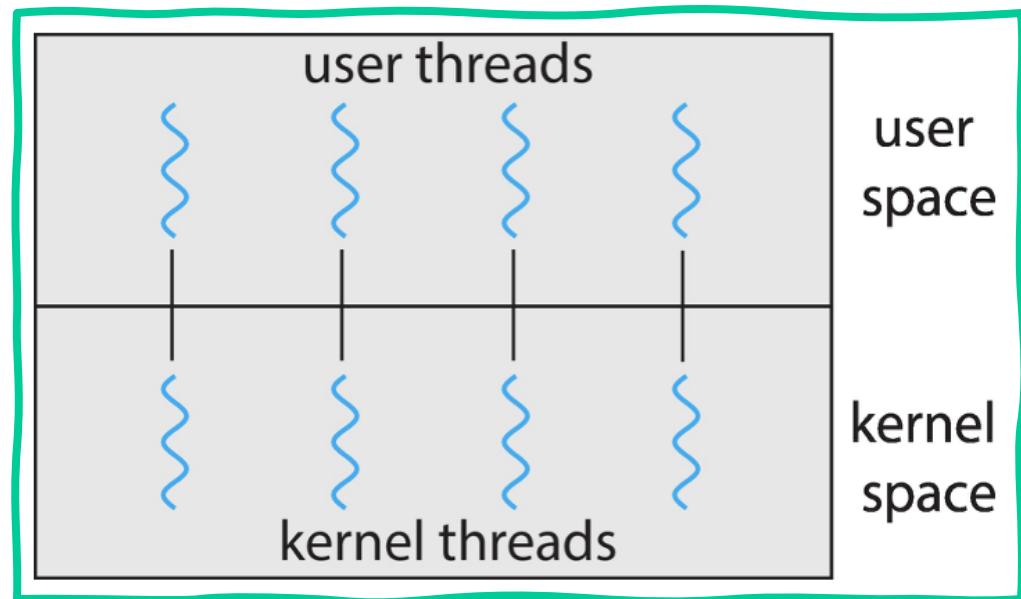
- ◆ Many user-level threads mapped to single kernel thread
- ◆ One thread blocking causes all to block
- ◆ Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- ◆ Few systems currently use this model
- ◆ Examples:
 - ◆ Solaris Green Threads
 - ◆ GNU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

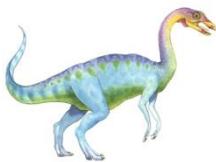




Multithreading Models: One-to-One

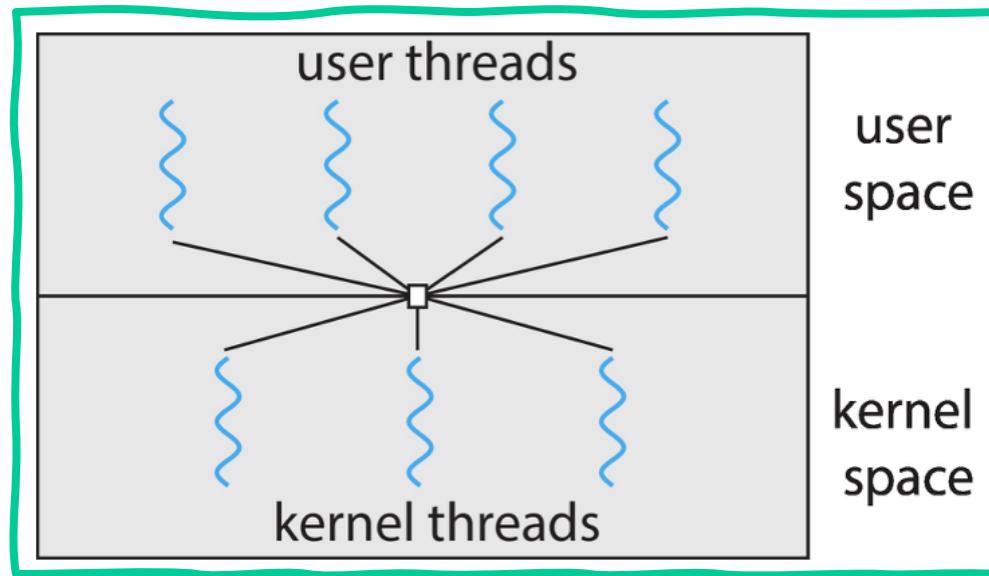
- ◆ Each user-level thread maps to kernel thread
- ◆ Creating a user-level thread creates a kernel thread
- ◆ More concurrency than many-to-one
- ◆ Number of threads per process sometimes restricted due to overhead
- ◆ Examples
 - ◆ Windows
 - ◆ Linux





Multithreading Models: Many-to-Many Model

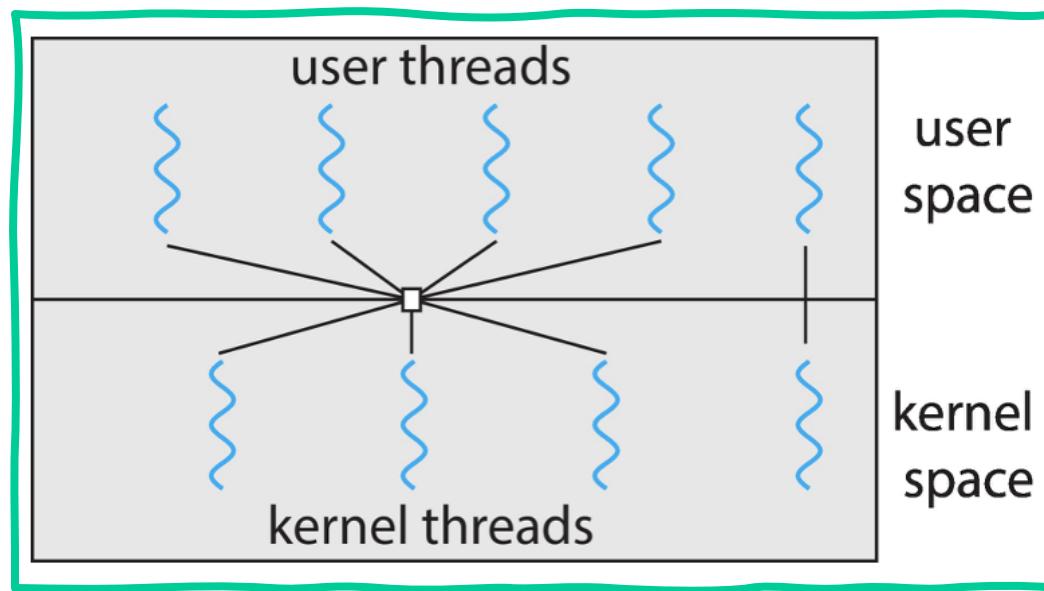
- ◆ Allows many user level threads to be mapped to many kernel threads
- ◆ Allows the operating system to create a sufficient number of kernel threads
- ◆ Windows with the *ThreadFiber* package
- ◆ Otherwise not very common





Two-level Model

- ◆ Similar to M:M, except that it allows a user thread to be **bound** to kernel thread





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





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 (Linux & Mac OS X)





Pthreads Example

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

#include <stdlib.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 */

    /* set the default attributes of the thread */
    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 execute 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);
}
```

```
#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);
```





C++ Threads

```
int sum = 0;

mutex sum_mutex;

void runner_shared(const string& param) {
    int upper = 0;
    try {
        upper = std::stoi(param);
    } catch (...) {
        return;
    }

    int local_calc = 0;
    for (int i = 1; i <= upper; ++i) {
        local_calc += i;
    }

    lock_guard<mutex> lock(sum_mutex);
    sum = local_calc;
}
```

```
int main(int argc, char* argv[]) {
    if (argc < 2) {
        cerr << "Usage: " << argv[0]
            << " <integer_upper_limit>" << endl;
        return 1;
    }

    thread worker(runner_shared, argv[1]);
    worker.join();

    cout << "sum = " << sum << endl;

    return 0;
}
```

```
#include <iostream>
#include <thread>
#include <string>
#include <mutex>
#include <stdexcept>

using namespace std;
```





Windows Multithreaded C Program

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

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 1; i <= Upper; i++)
        Sum += i;
    return 0;
}
```

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

    Param = atoi(argv[1]);
    /* 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 */

    /* 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
- ◆ Java threads may be created by:
 - ◆ Extending Thread class
 - ◆ Implementing the Runnable interface

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

- ◆ Standard practice is to implement Runnable interface





Java Threads

Implementing Runnable interface:

```
class Task implements Runnable
{
    public void run() {
        System.out.println("I am a thread.");
    }
}
```

Creating a thread:

```
Thread worker = new Thread(new Task());
worker.start();
```

Waiting on a thread:

```
try {
    worker.join();
}
catch (InterruptedException ie) { }
```





Java Executor Framework

- ◆ Rather than explicitly creating threads, Java also allows thread creation around the Executor interface:

```
public interface Executor
{
    void execute(Runnable command);
}
```

- ◆ The Executor is used as follows:

```
Executor service = new Executor;
service.execute(new Task());
```





Java Executor Framework

```
import java.util.concurrent.*;  
  
class Summation implements Callable<Integer>  
{  
    private int upper;  
    public Summation(int upper) {  
        this.upper = upper;  
    }  
  
    /* The thread will execute in this method */  
    public Integer call() {  
        int sum = 0;  
        for (int i = 1; i <= upper; i++)  
            sum += i;  
  
        return new Integer(sum);  
    }  
}
```

```
public class Driver  
{  
    public static void main(String[] args) {  
        int upper = Integer.parseInt(args[0]);  
  
        ExecutorService pool = Executors.newSingleThreadExecutor();  
        Future<Integer> result = pool.submit(new Summation(upper));  
  
        try {  
            System.out.println("sum = " + result.get());  
        } catch (InterruptedException | ExecutionException ie) { }  
    }  
}
```



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
- ◆ Five methods explored
 - ◆ Thread Pools
 - ◆ Fork-Join
 - ◆ OpenMP
 - ◆ Grand Central Dispatch
 - ◆ Intel Threading Building Blocks





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
 - ◆ i.e. Tasks could be scheduled to run periodically
- ◆ Windows API supports thread pools:

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





Java Thread Pools

- ◆ Three factory methods for creating thread pools in Executors class:

- static ExecutorService newSingleThreadExecutor()
- static ExecutorService newFixedThreadPool(int size)
- static ExecutorService newCachedThreadPool()

```
import java.util.concurrent.*;

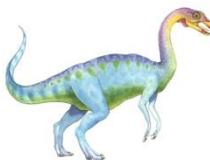
public class ThreadPoolExample
{
    public static void main(String[] args) {
        int numTasks = Integer.parseInt(args[0].trim());

        /* 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());

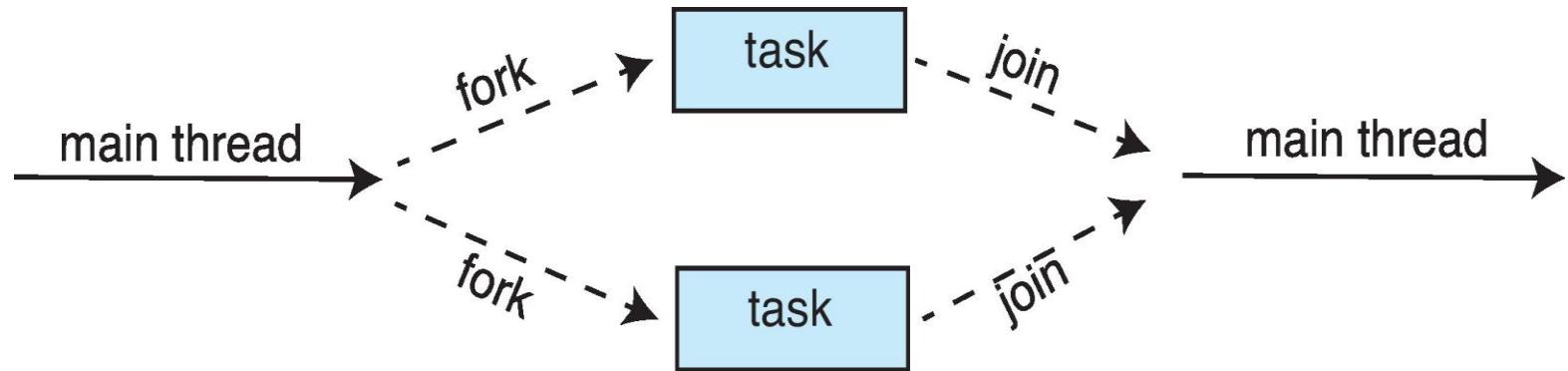
        /* Shut down the pool once all threads have completed */
        pool.shutdown();
    }
}
```





Fork-Join Parallelism

- Multiple threads (tasks) are **forked**, and then **joined**.

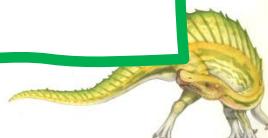


- General algorithm for fork-join strategy:

```
Task(problem)
    if problem is small enough
        solve the problem directly
    else
        subtask1 = fork(new Task(subset of problem))
        subtask2 = fork(new Task(subset of problem))

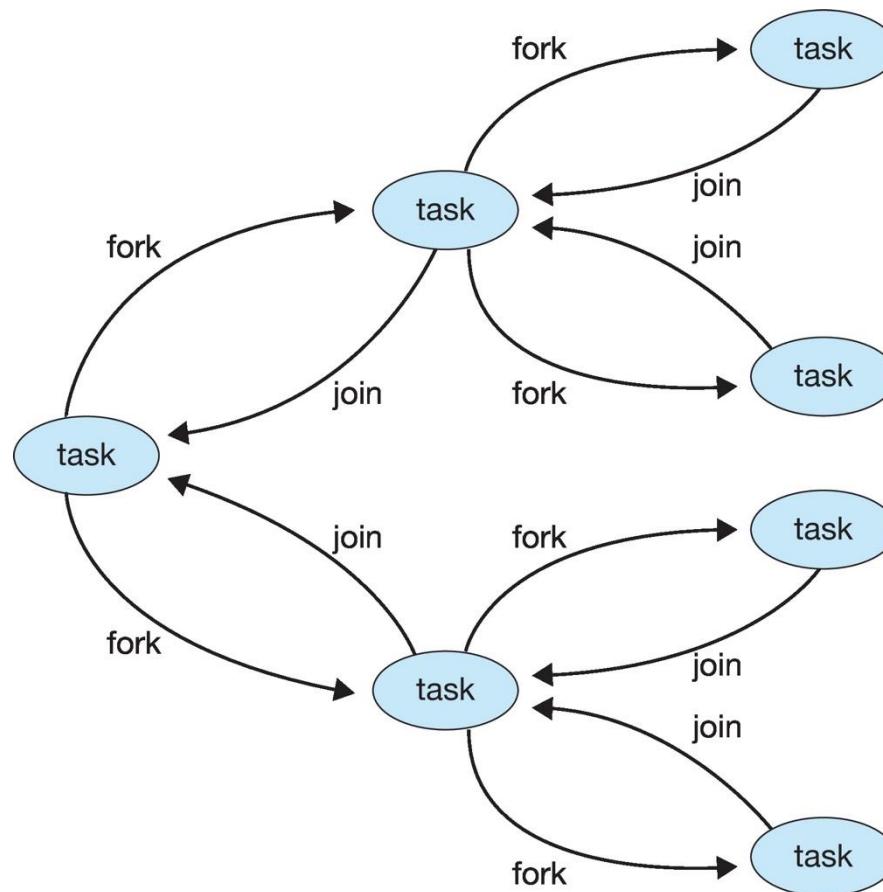
        result1 = join(subtask1)
        result2 = join(subtask2)

    return combined results
```





Fork-Join Parallelism





Fork-Join Parallelism in Java

```
ForkJoinPool pool = new ForkJoinPool();
// array contains the integers to be summed
int[] array = new int[SIZE];

SumTask task = new SumTask(0, SIZE - 1, array);
int sum = pool.invoke(task);
```

```
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;
    }

    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) / 2;

            SumTask leftTask = new SumTask(begin, mid, array);
            SumTask rightTask = new SumTask(mid + 1, end, array);

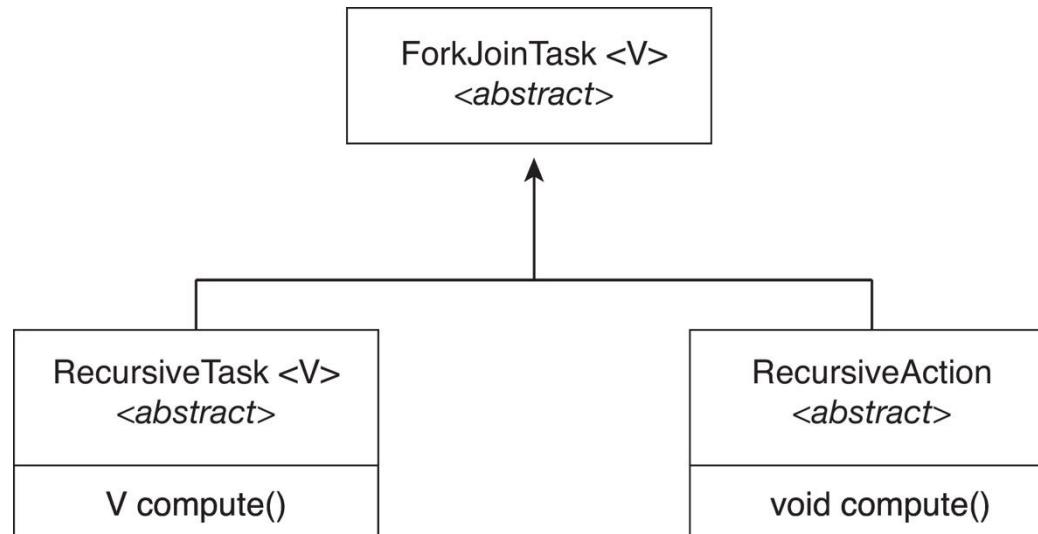
            leftTask.fork();
            rightTask.fork();

            return rightTask.join() + leftTask.join();
        }
    }
}
```



Fork-Join Parallelism in Java

- ◆ The **ForkJoinTask** is an abstract base class
- ◆ **RecursiveTask** and **RecursiveAction** classes extend **ForkJoinTask**
- ◆ **RecursiveTask** returns a result (via the return value from the **compute()** method)
- ◆ **RecursiveAction** does not return a result





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

- ◆ Run the for loop in parallel

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

```
#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;
}
```





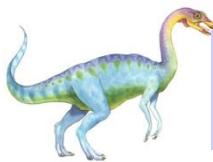
Grand Central Dispatch

- ◆ Apple technology for macOS and iOS operating systems
- ◆ Extensions to C, C++ and Objective-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

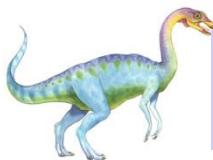




Grand Central Dispatch

- ◆ 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
- ◆ Four system wide queues divided by quality of service:
 - ◆ **`QOS_CLASS_USER_INTERACTIVE`**
 - ◆ **`QOS_CLASS_USER_INITIATED`**
 - ◆ **`QOS_CLASS_USER.Utility`**
 - ◆ **`QOS_CLASS_USER_BACKGROUND`**



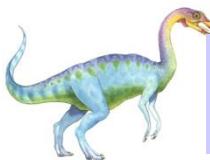


Grand Central Dispatch

- ◆ For the Swift language a task is defined as a closure – similar to a block, minus the caret
- ◆ Closures are submitted to the queue using the `dispatch_async()` function:

```
let queue = dispatch_get_global_queue  
    (QOS_CLASS_USER_INITIATED, 0)  
  
dispatch_async(queue, { print("I am a closure.") })
```





Intel Threading Building Blocks (TBB)

- ◆ Template library for designing parallel C++ programs
- ◆ A serial version of a simple for loop

```
for (int i = 0; i < n; i++) {  
    apply(v[i]);  
}
```

- ◆ The same for loop written using TBB with **parallel_for** statement:

```
parallel_for (size_t(0), n, [=](size_t i) {apply(v[i]);});
```





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 UNIXes have two versions of fork
- ◆ `exec()` usually works as normal – replace the running process including all threads

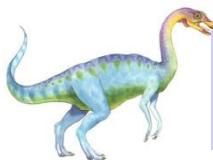




Signal Handling

- ◆ **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- ◆ A **signal handler** is used to process signals
 - ◆ Signal is generated by particular event
 - ◆ Signal is delivered to a process
 - ◆ Signal is handled by one of two signal handlers:
 - ◆ default
 - ◆ user-defined
- ◆ Every signal has **default handler** that 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





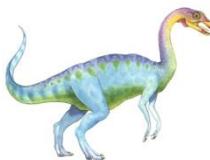
Thread Cancellation

- ◆ Terminating a thread before it has finished
- ◆ Thread to be canceled is **target thread**
- ◆ Two general approaches:
 - ◆ **Asynchronous cancellation** terminates the target thread immediately
 - ◆ **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

- ◆ **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);  
  
/* wait for the thread to terminate */  
pthread_join(tid,NULL);
```





Thread Cancellation (Cont.)

- ◆ Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

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

- ◆ If thread has cancellation disabled, cancellation remains pending until thread enables it
- ◆ Default type is deferred
 - ◆ Cancellation only occurs when thread reaches **cancellation point**
 - ◆ i.e. `pthread_testcancel()`
 - ◆ Then **cleanup handler** is invoked
 - ◆ On Linux systems, thread cancellation is handled through signals





Thread Cancellation in Java

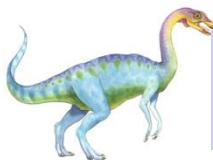
- Deferred cancellation uses the `interrupt()` method, which sets the interrupted status of a thread.

```
Thread worker;  
  
    . . .  
  
/* set the interruption status of the thread */  
worker.interrupt()
```

- A thread can then check to see if it has been interrupted:

```
while (!Thread.currentThread().isInterrupted()) {  
    . . .  
}
```

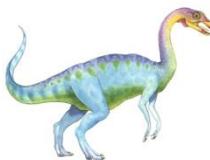




Thread-Local Storage

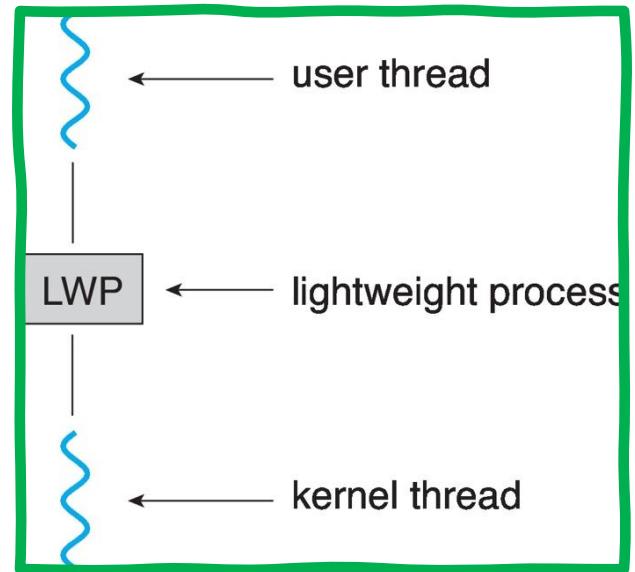
- ◆ **Thread-local storage (TLS)** allows each thread to have its own copy of data
- ◆ Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- ◆ Different from local variables
 - ◆ Local variables visible only during single function invocation
 - ◆ TLS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





Scheduler Activations

- ◆ Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- ◆ Typically use an intermediate data structure between user and kernel threads – **lightweight process (LWP)**
 - ◆ Appears to be a virtual processor on which process can schedule user thread to run
 - ◆ Each LWP attached to kernel thread
 - ◆ How many LWPs to create?
- ◆ 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





Windows Threads: Example

- ◆ Windows API – primary API for Windows applications
- ◆ 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

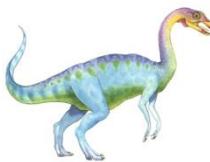




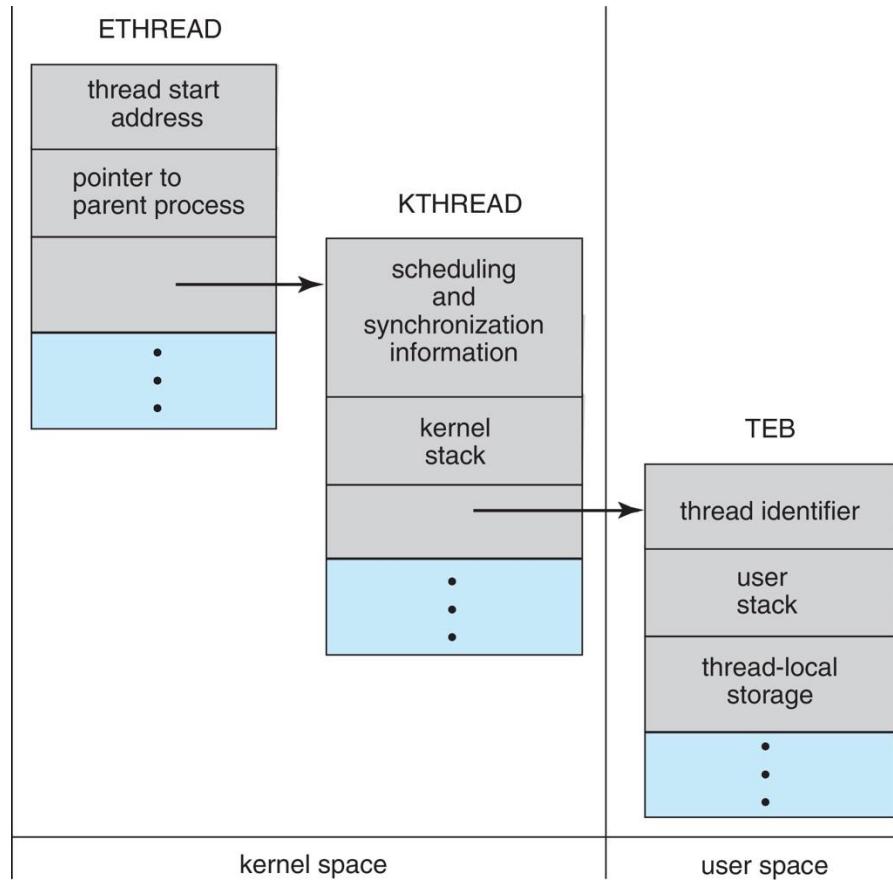
Windows Threads (Cont.)

- ◆ 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

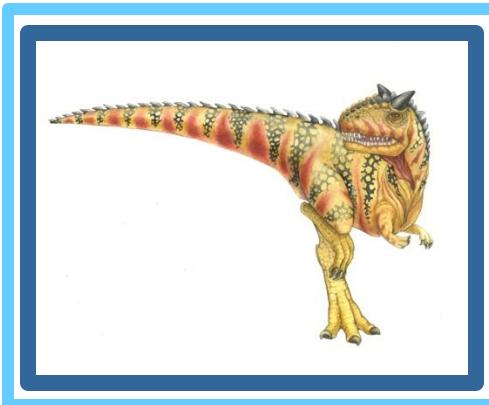
- ◆ Linux refers to them as **tasks** rather than **threads**
- ◆ Thread creation is done through **clone ()** system call
- ◆ **clone ()** allows a child task to share the address space of the parent task (process)
 - ◆ Flags control behavior

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.

- ◆ **struct task_struct** points to process data structures (shared or unique)

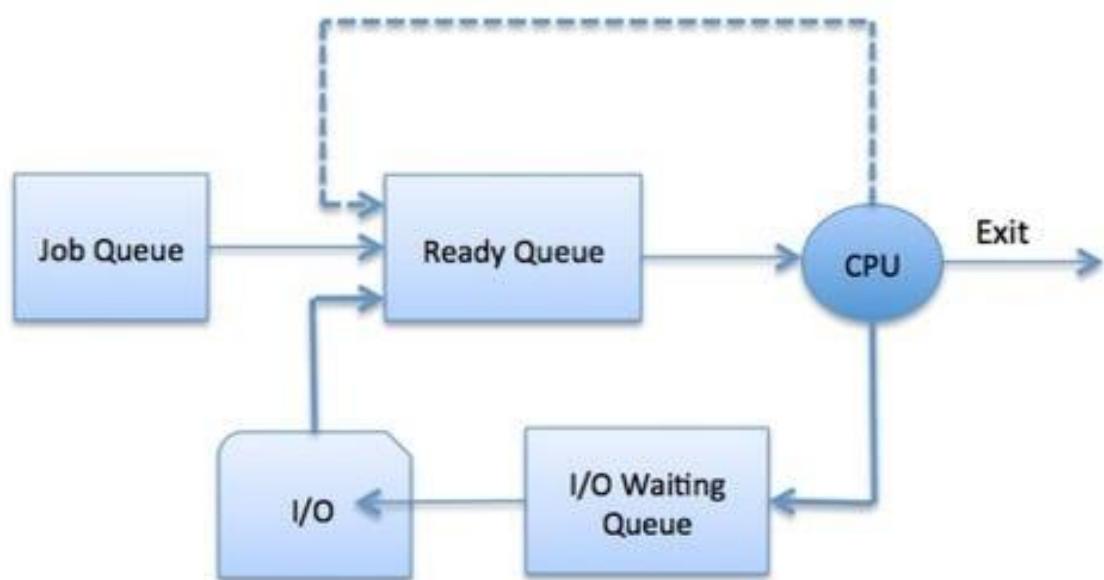


End of Chapter 4



Week 5

CPU Scheduling





Week 5: CPU Scheduling

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





Objectives

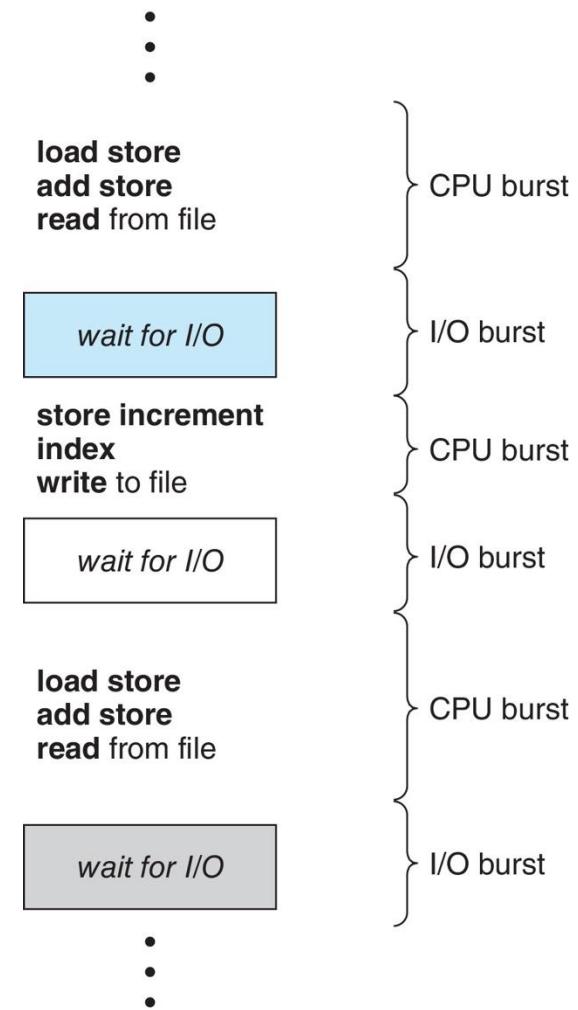
- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms
- Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems
- Apply modeling and simulations to evaluate CPU scheduling algorithms





Basic Concepts

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



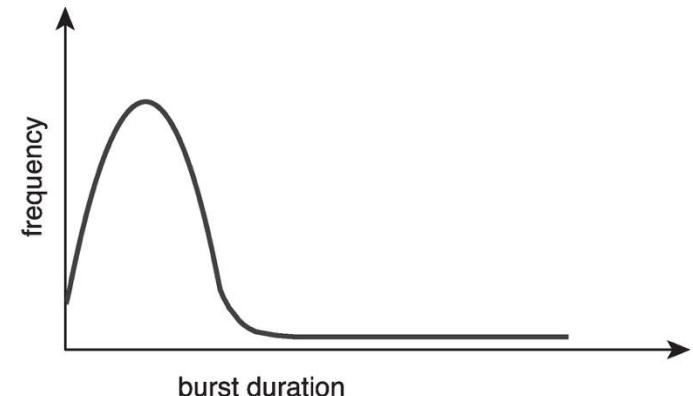


Histogram of CPU-burst Times

The graph illustrates the observed pattern of how long processes (or threads) typically use the CPU before requiring an I/O operation or yielding the processor. This period of CPU usage is called a **CPU burst**.

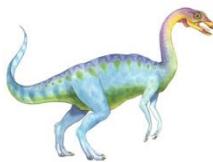
- **Large number of short bursts:**

The curve peaks sharply at the lower end of the "burst duration" axis, meaning that **most processes only require a small amount of CPU time** before waiting for input/output (I/O). The high frequency of short bursts is dominant.



- **Small number of longer bursts:** The curve quickly drops off and has a **long tail** extending far along the "burst duration" axis, indicating that only a **few processes require a very long, sustained period of CPU time**.





CPU Scheduler

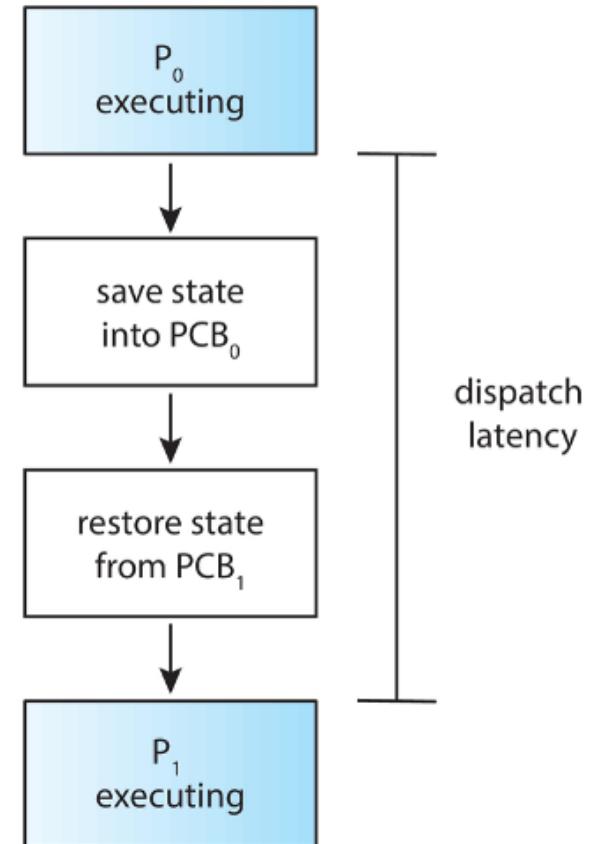
- The **CPU scheduler** selects from among the processes in ready queue, and allocates a CPU core to one of them
 - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
 1. Switches from running to waiting state
 2. Switches from running to ready state
 3. Switches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





Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term 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** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – 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)





Scheduling Algorithm Optimization Criteria

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





First- Come, First-Served (FCFS) Scheduling

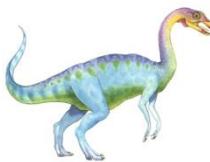
<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- 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: $(0 + 24 + 27)/3 = 17$





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





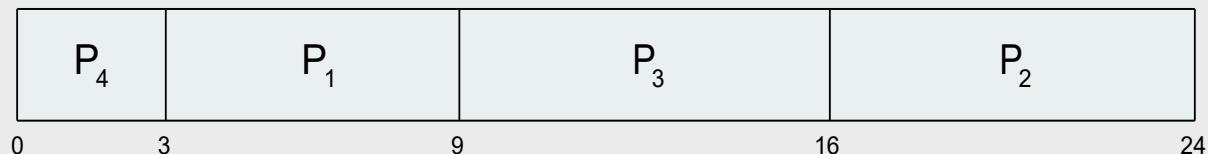
Shortest-Job-First (SJF) Scheduling

- 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
 - The difficulty is knowing the length of the next CPU request
 - Could ask the user

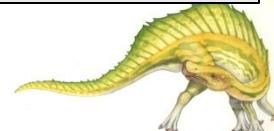
Example

Process	Burst Time
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 the length – should be similar to the previous one
 - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging

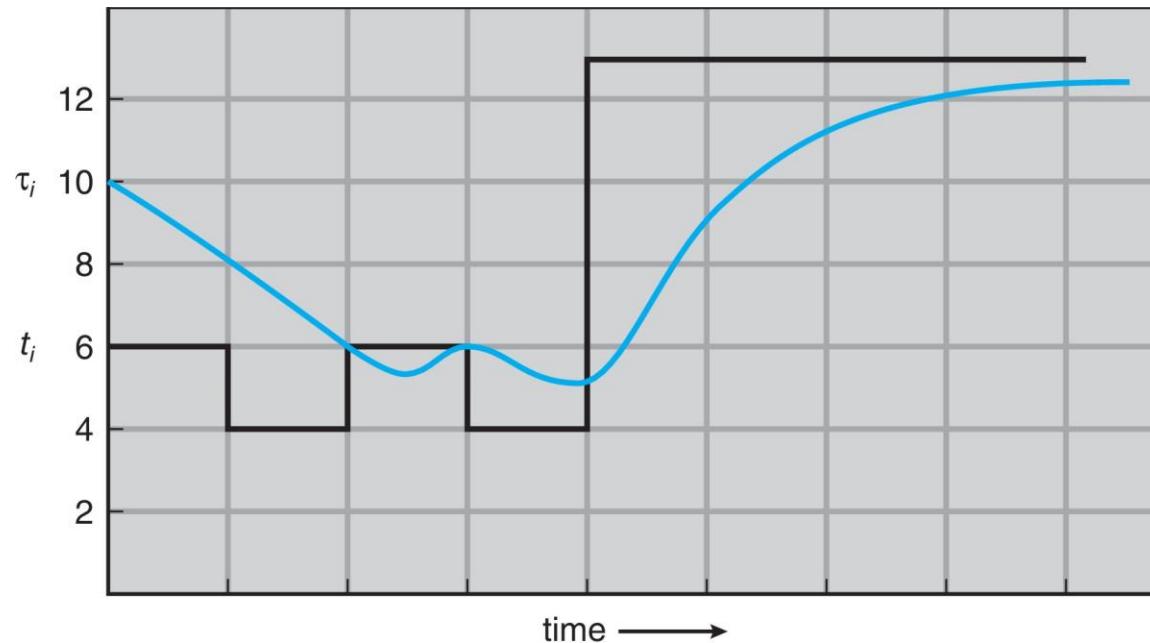
1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define : $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n.$

- Commonly, α set to $\frac{1}{2}$
- Preemptive version called **shortest-remaining-time-first**





Prediction of the Length of the Next CPU Burst



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





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



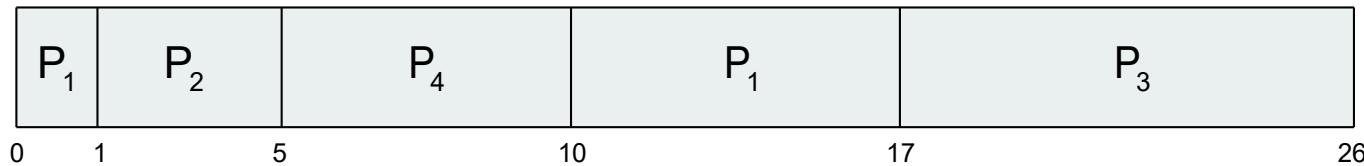


Example of Shortest-remaining-time-first

- 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





Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds. 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

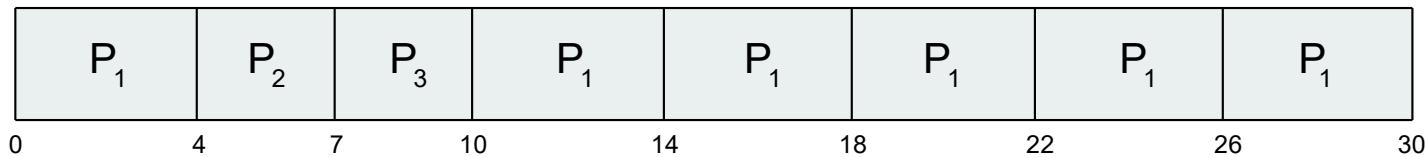




Example of RR with Time Quantum = 4

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

- The Gantt chart is:

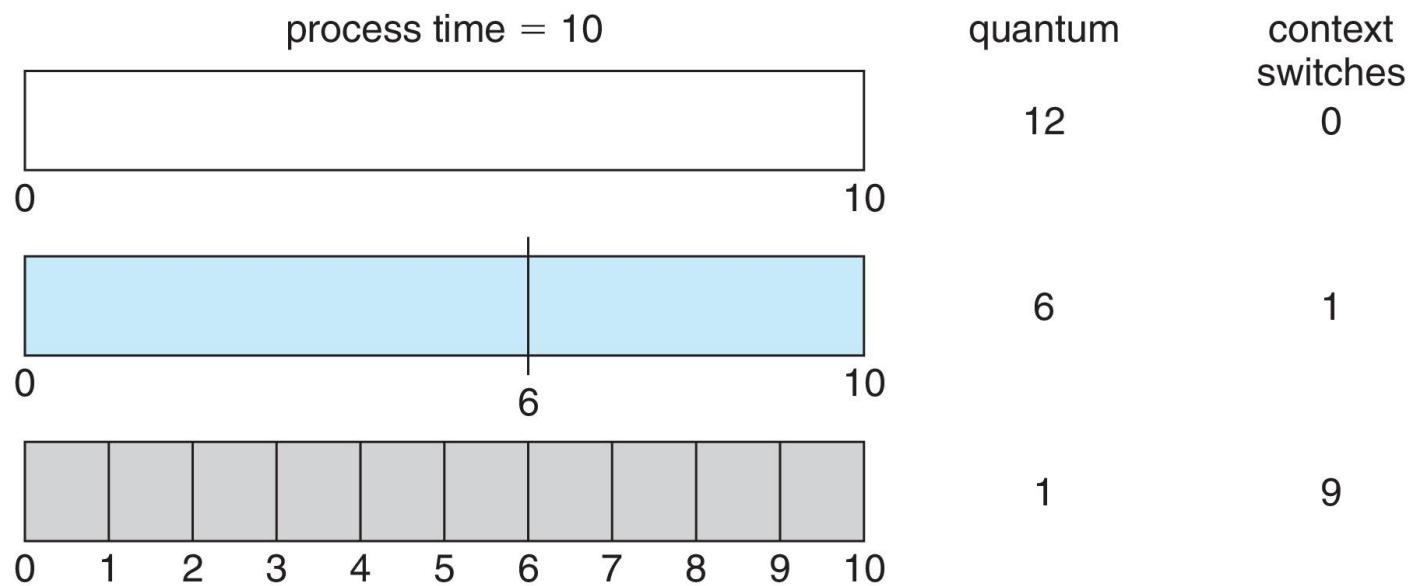


- Typically, higher average turnaround than SJF, but better **response**
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec



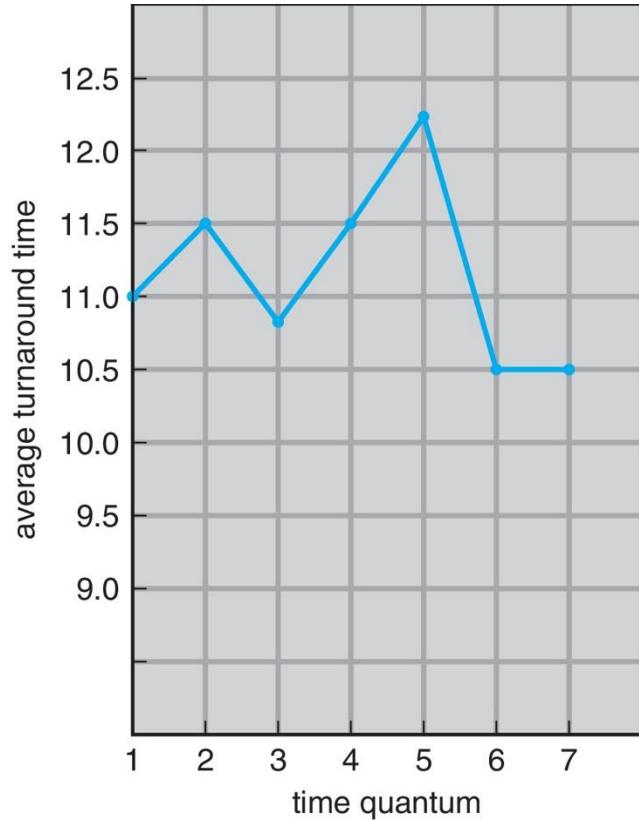


Time Quantum and Context Switch Time





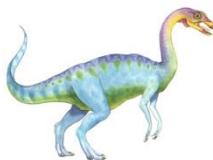
Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts
should be shorter than q





Priority Scheduling

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

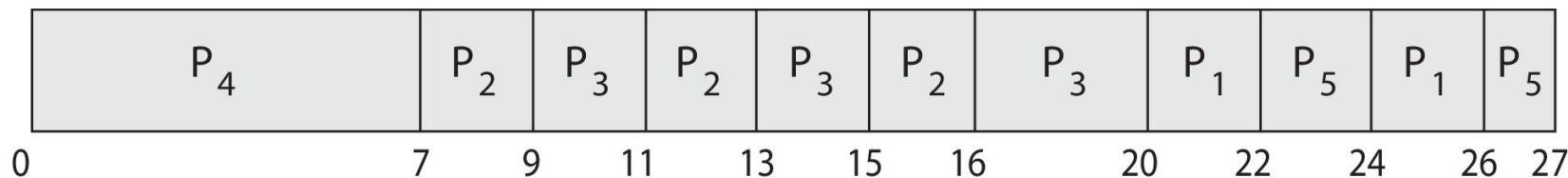




Priority Scheduling w/ Round-Robin

<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

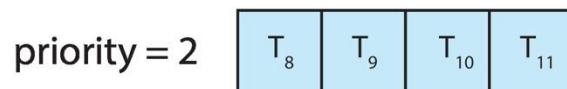
- Run the process with the highest priority. Processes with the same priority run round-robin
- Gantt Chart wit 2 ms time quantum





Multilevel Queue

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!



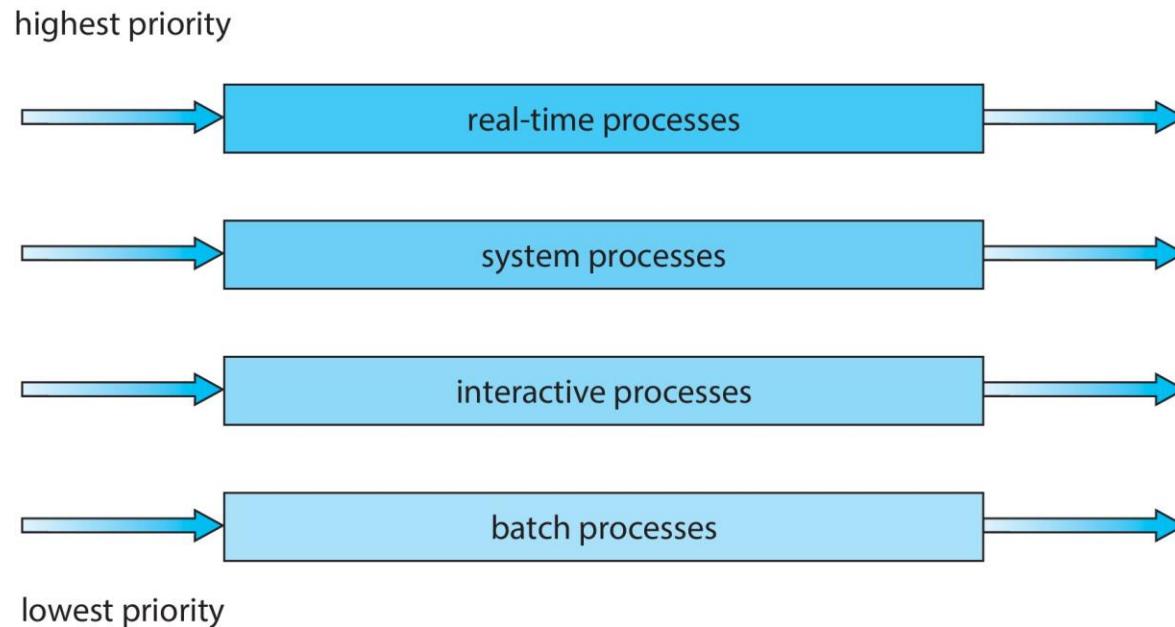
●
●
●





Multilevel Queue

- Prioritization based upon process type





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





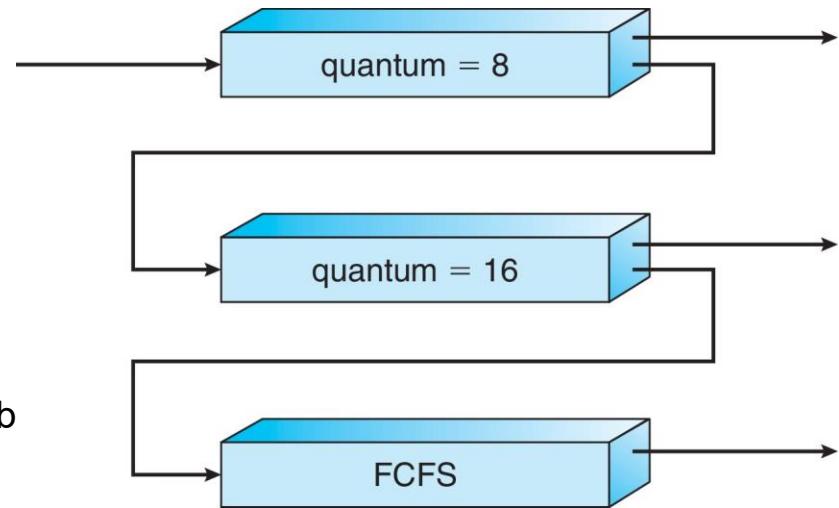
Example of Multilevel Feedback Queue

- 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





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

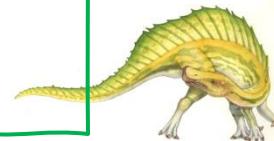
- API allows specifying either PCS or SCS during thread creation
 - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
 - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and macOS only allow PTHREAD_SCOPE_SYSTEM





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);
}
```





Multiple-Processor Scheduling

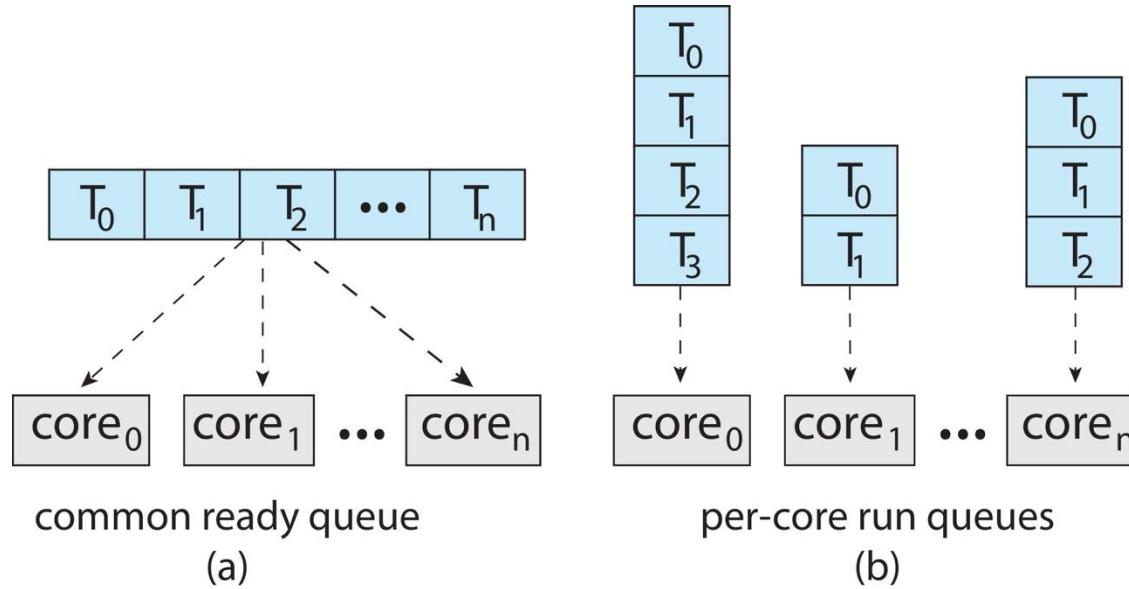
- CPU scheduling more complex when multiple CPUs are available
- Multi-process may be any one of the following architectures:
 - Multicore CPUs
 - Multithreaded cores
 - NUMA systems
 - Heterogeneous multiprocessing

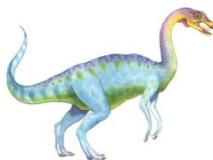




Multiple-Processor Scheduling

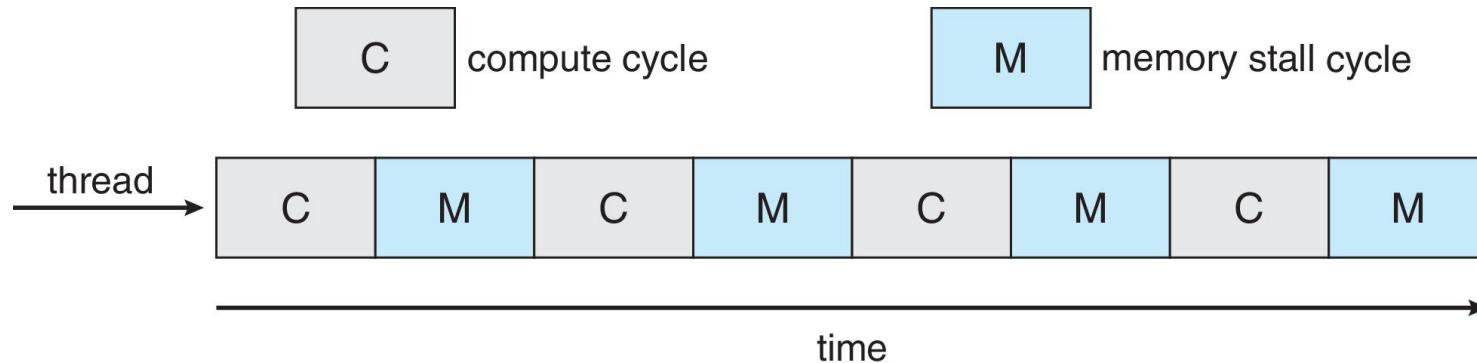
- Symmetric multiprocessing (SMP) is where each processor is self scheduling.
- All threads may be in a common ready queue (a)
- Each processor may have its own private queue of threads (b)





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

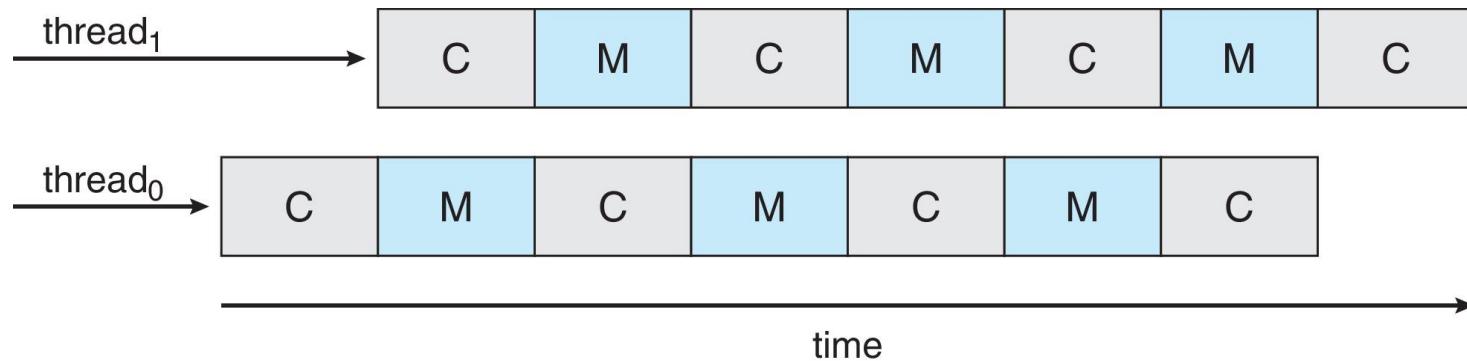




Multithreaded Multicore System

Each core has > 1 hardware threads.

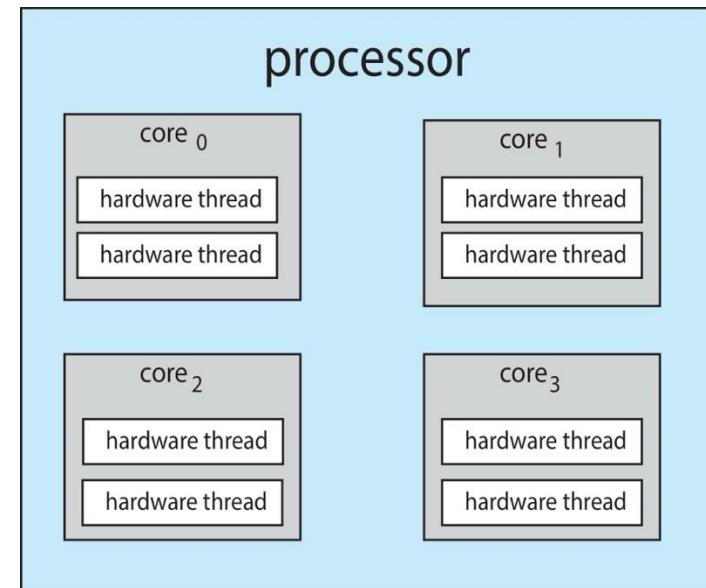
If one thread has a memory stall, switch to another thread!



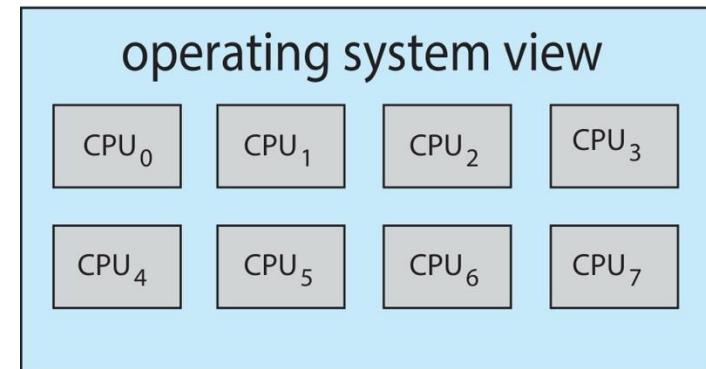


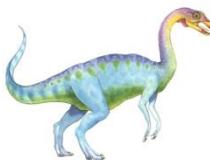
Multithreaded Multicore System

- Chip-multithreading (CMT) assigns each core multiple hardware threads. (Intel refers to this as **hyperthreading**)



- On a quad-core system with 2 hardware threads per core, the operating system sees **8 logical processors**.

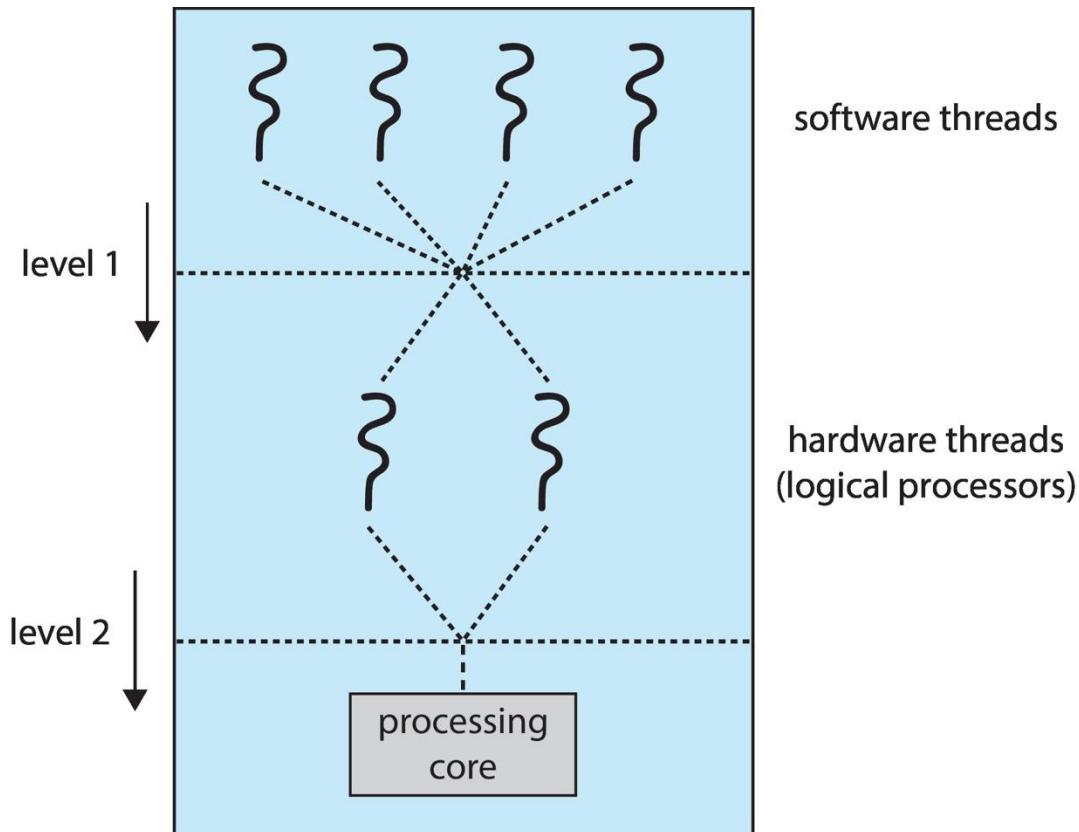




Multithreaded Multicore System

- Two levels of scheduling:

1. The operating system deciding which software thread to run on a logical CPU
2. How each core decides which hardware thread to run on the physical core.





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

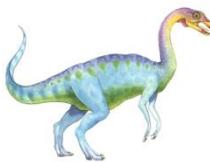




Multiple-Processor Scheduling – Processor Affinity

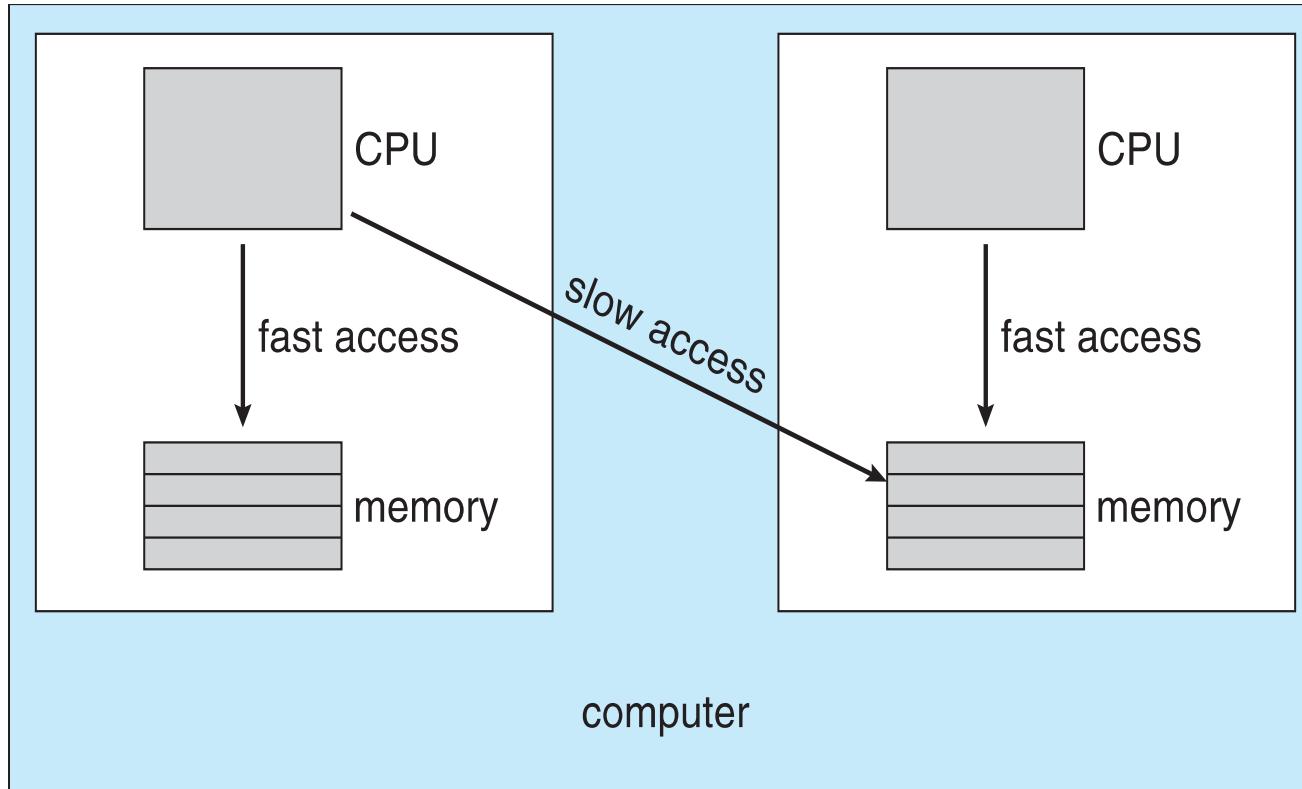
- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having affinity for a processor (i.e. “processor affinity”)
- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.
- Soft affinity – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- Hard affinity – allows a process to specify a set of processors it may run on.





NUMA and CPU Scheduling

If the operating system is **NUMA-aware**, it will assign memory closer to the CPU the thread is running on.





Real-Time CPU Scheduling

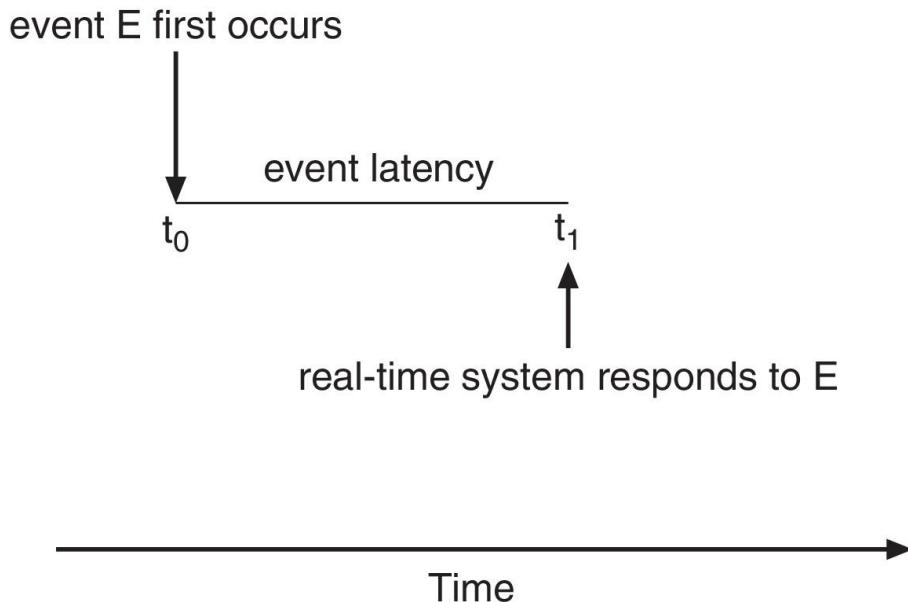
- Can present obvious challenges
- Soft real-time systems – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
- Hard real-time systems – task must be serviced by its deadline





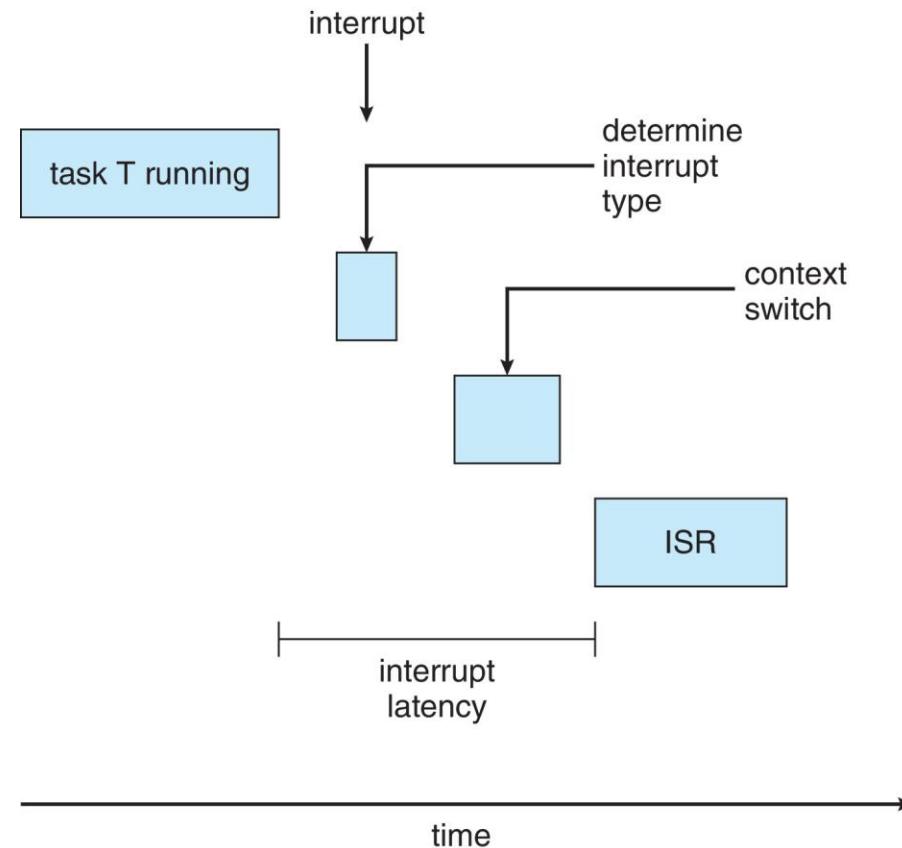
Real-Time CPU Scheduling

- Event latency – the amount of time that elapses from when an event occurs to when it is serviced.
- 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 schedule to take current process off CPU and switch to another





Interrupt Latency

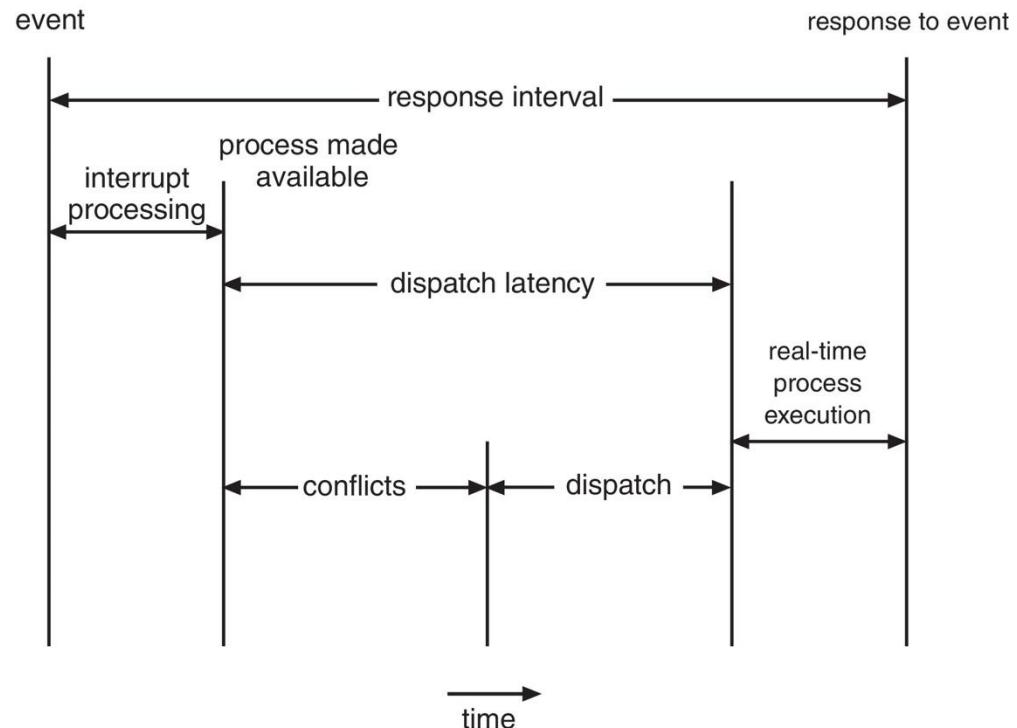




Dispatch Latency

- Conflict phase of dispatch latency:

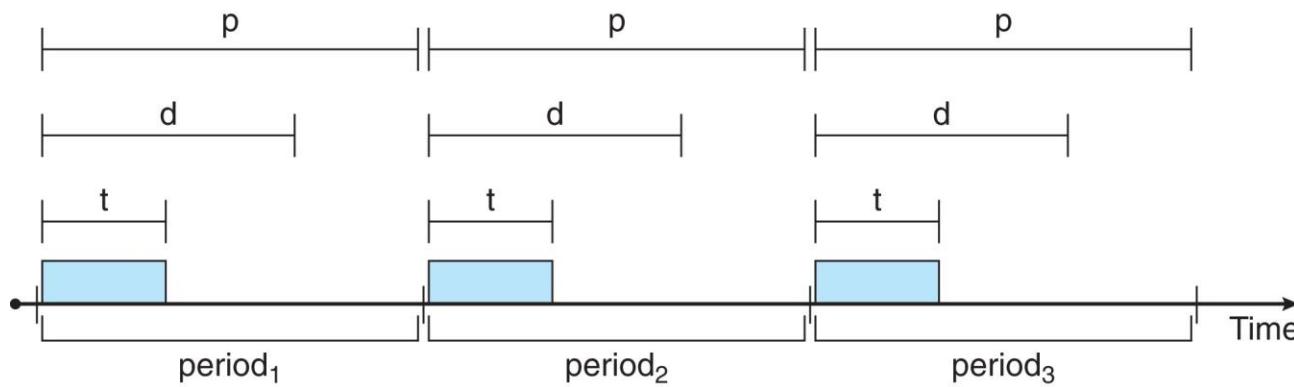
1. Preemption of any process running in kernel mode
2. Release by low-priority process of resources needed by high-priority processes





Priority-based Scheduling

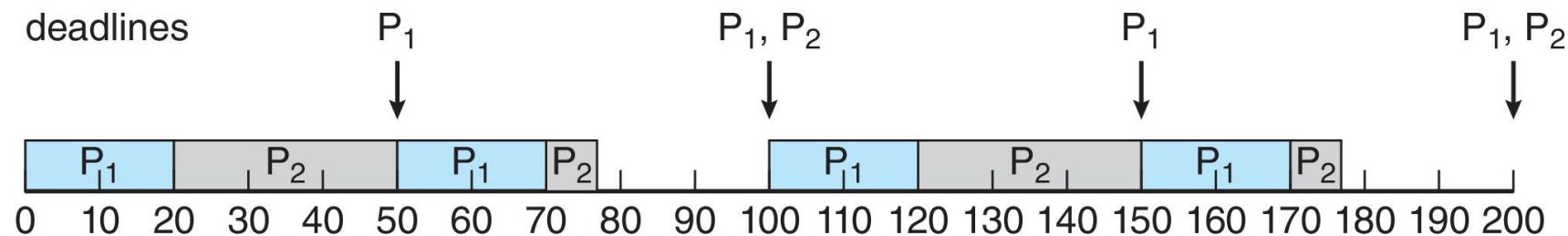
- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
 - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
 - Has processing time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





Rate Monotonic Scheduling

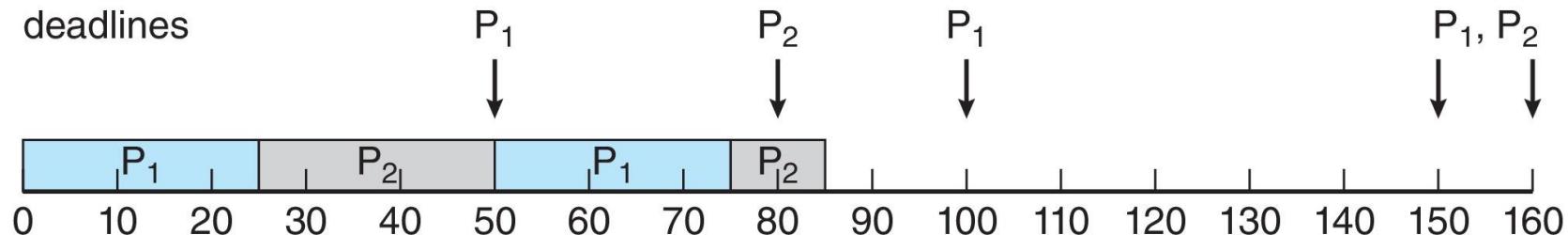
- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P_1 is assigned a higher priority than P_2 .





Missed Deadlines with Rate Monotonic Scheduling

Process P2 misses finishing its deadline at time 80



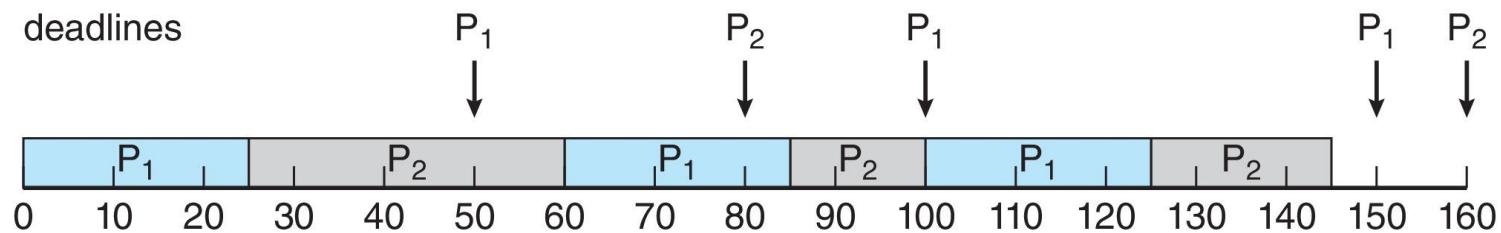


Earliest Deadline First Scheduling (EDF)

- 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);
}
```



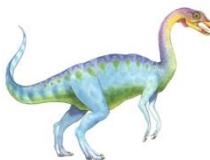


Algorithm Evaluation

- 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



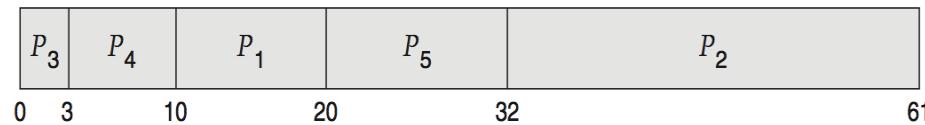


Deterministic Evaluation

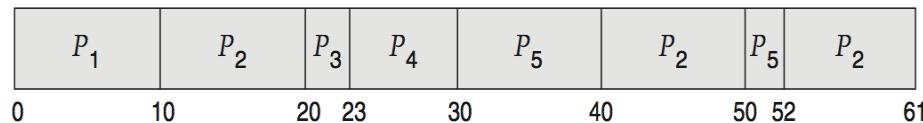
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
 - FCS is 28ms:



- 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





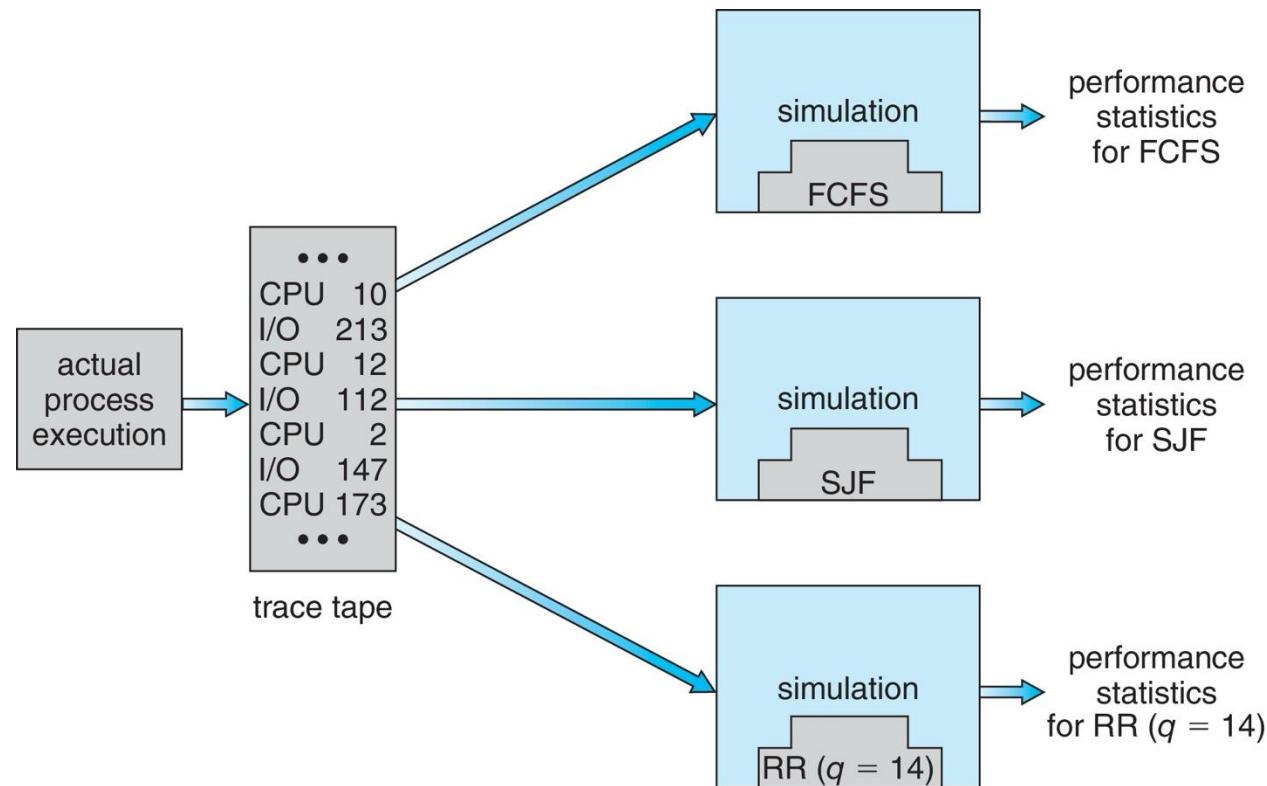
Simulations

- 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



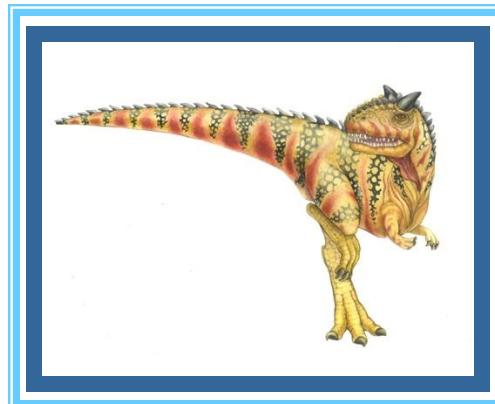


Implementation

- 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



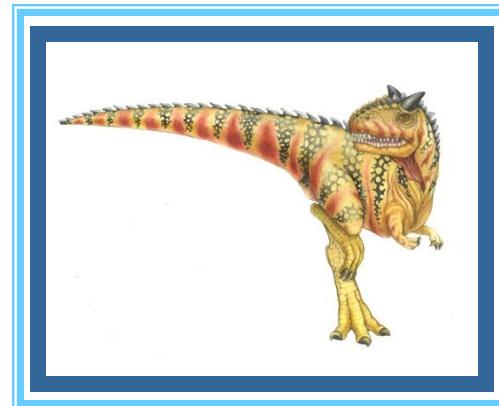
End of Chapter 5

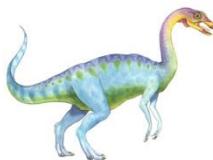


Week 6

Synchronization

Chapter 6

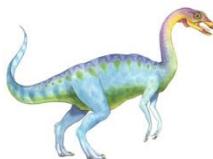




Objectives

- Describe the critical-section problem and illustrate a race condition
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical section problem
- Evaluate tools that solve the critical-section problem in low-, Moderate-, and high-contention scenarios





Background

- Processes can execute concurrently
 - May be interrupted at any time, partially completing execution
- 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:
Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **counter** that keeps track of the number of full buffers. Initially, **counter** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.





Producer / Consumer

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

    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Producer

```
Consumer
while (true) {
    while (counter == 0)
        ; /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;

    /* consume the item in next_consumed */
}
```





Race Condition

- `counter++` could be implemented as

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

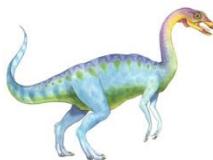
- `counter--` could be implemented as

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

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

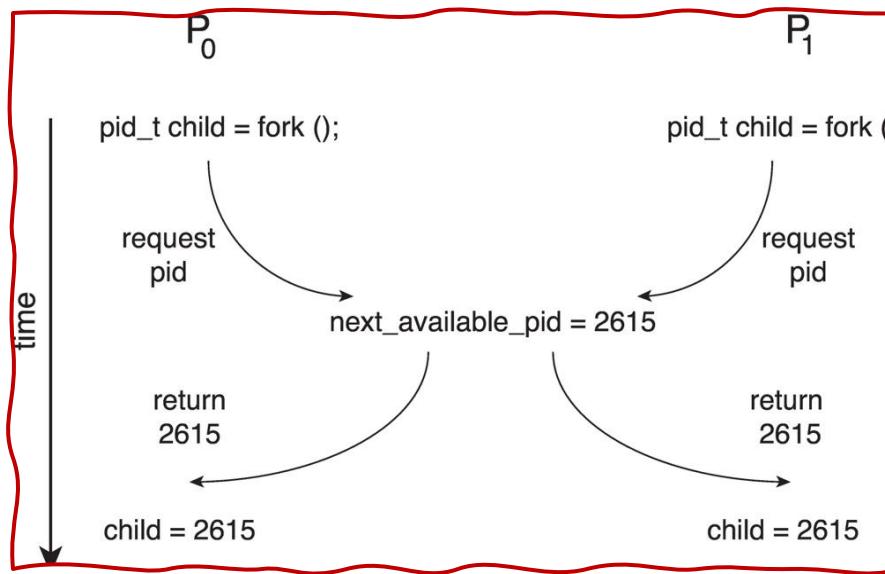
S0: producer execute <code>register1 = counter</code>	{register1 = 5}
S1: producer execute <code>register1 = register1 + 1</code>	{register1 = 6}
S2: consumer execute <code>register2 = counter</code>	{register2 = 5}
S3: consumer execute <code>register2 = register2 - 1</code>	{register2 = 4}
S4: producer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





Race Condition

- Processes P_0 and P_1 are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



- Unless there is mutual exclusion, the same pid could be assigned to two different processes!





Critical Section Problem

a **critical section** is a segment of code that accesses shared resources (like variables, files, or hardware devices) and must be executed by only one process or thread at any given time.

- 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**, may follow critical section with **exit section**, then **remainder section**





Structure of a Critical Section

A typical critical section implementation involves four conceptual parts:

- **Entry Section:** Code that requests permission to enter the critical section. It often involves acquiring a lock or checking a flag. If the critical section is currently in use, the process will block here.
- **Critical Section:** This is the actual code segment where the shared resource is accessed and modified. This is the part that must be executed **exclusively**.
- **Exit Section:** Code releases the permission to the critical section, signaling that it is now free for another process to enter. It often involves releasing a lock or modifying a flag.
- **Remainder Section:** This is the rest of the code that does not involve the shared resource and can be executed concurrently with other processes.

General structure
of process P_i

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



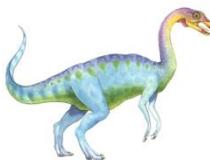


Solution to Critical-Section Problem

Any successful solution must meet these three conditions to ensure **correct and fair coordination** between competing processes:

1. **Mutual Exclusion:** Only one process can be inside its critical section at any given time. This is the most crucial requirement, preventing shared data from being corrupted by simultaneous access.
2. **Progress:** If the critical section is free and there are processes that want to enter, then the decision of which process enters next cannot be delayed forever. If a resource is available, processes shouldn't be forced to wait unnecessarily or indefinitely (Prevents deadlock).
3. **Bounded Waiting:** Once a process asks to enter the critical section, there's a limit on how many times other processes can enter before the waiting process gets its turn. This guarantees that a process won't wait an infinite amount of time to access the resource while others keep cycling through (Prevents starvation).





Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- ❑ **Preemptive** – allows preemption of process when running in kernel mode
- ❑ **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
 - ▶ Essentially free of race conditions in kernel mode





Peterson's Solution

- Not guaranteed to work on modern architectures! (But good algorithmic description of solving the problem)
- 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:
 - **flag[2]**: A boolean array where **flag[i] = true** indicates that process P_i is interested to enter the critical section.
 - **turn**: An integer variable that indicates whose turn it is to enter the critical section. If turn = i, process P_i is allowed to proceed if the other process is also interested.





Algorithm for Process P_i

```
while (true)
{
```

```
    flag[i] = true; ←
```

P_i declares its intent to enter.

```
    turn = j; ←
```

P_i gives the priority/turn to P_j .

```
    while (flag[j] && turn == j)
        ; ←
```

P_i waits (busy-waits) if: P_j is interested ($flag[j]=true$) AND P_j has the priority ($turn=j$).

```
    /* critical section */ ←
```

Only one process can reach this point at a time.

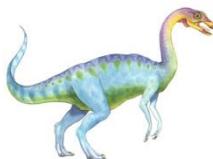
```
    flag[i] = false; ←
```

P_i is done and resets its intent flag, letting the other process proceed.

```
    /* remainder section */
```

```
}
```





Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:

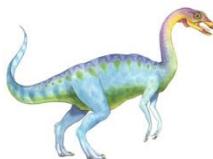
1. Mutual exclusion is preserved

P_i enters CS only if:

either `flag[j] = false` or `turn = i`

2. Progress requirement is satisfied
3. Bounded-waiting requirement is met





Peterson's Solution

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
- Understanding why it will not work is also useful for better understanding race conditions.
- To improve performance, processors and/or compilers may reorder operations that have no dependencies.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!





Peterson's Solution

- Two threads share the data:

```
boolean flag = false;  
int x = 0;
```

- Thread 1 performs

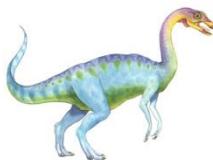
```
while (!flag)  
;  
print x
```

- Thread 2 performs

```
x = 100;  
flag = true
```

- What is the expected output?



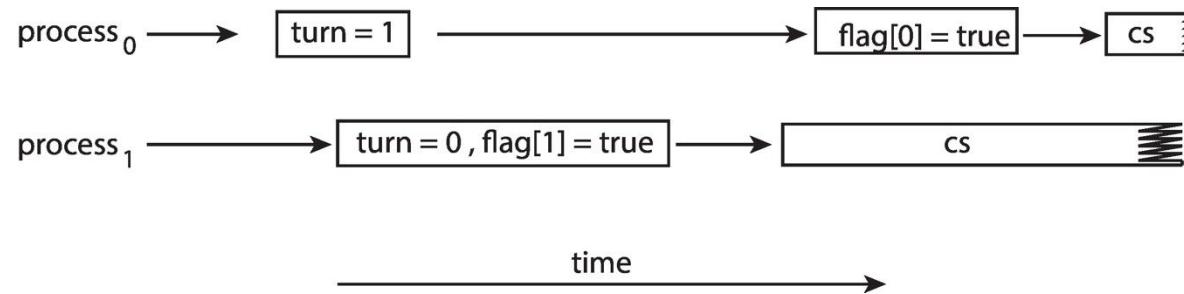


Peterson's Solution

- 100 is the expected output.
- However, the operations for Thread 2 may be reordered:

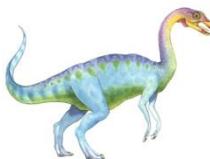
```
flag = true;  
x = 100;
```

- If this occurs, the output may be 0!
- The effects of instruction reordering in Peterson's Solution



- This allows both processes to be in their critical section at the same time!





Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
 - Currently running code would execute without preemption
 - Generally, too inefficient on multiprocessor systems
 - ▶ Operating systems using this not broadly scalable
- We will look at three forms of hardware support:
 1. Memory barriers
 2. Hardware instructions
 3. Atomic variables





Memory Barriers

- **Memory model** are the memory guarantees a computer architecture makes to application programs.
- Memory models may be either:
 - **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
 - **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.





Memory Barrier

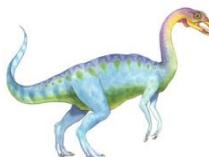
- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
    memory_barrier();
print x
```

- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```





Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or two *swap* the contents of two words atomically (uninterruptibly.)
- **Test-and-Set** instruction
- **Compare-and-Swap** instruction





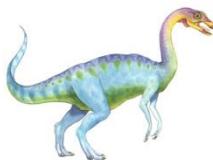
test_and_set Instruction

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = true;

    return rv;
}
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to **true**





Solution using test_and_set()

- Shared boolean variable **lock**, initialized to **false**
- Solution:

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





compare_and_swap Instruction

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

1. Executed atomically
2. Returns the original value of passed parameter **value**
3. Set the variable **value** the value of the passed parameter **new_value** but only if ***value == expected** is true. That is, the swap takes place only under this condition.





Solution using compare_and_swap

- Shared integer `lock` initialized to 0;
- Solution:

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





Bounded-waiting Mutual Exclusion with compare-and-swap

```
while (true)
{
    waiting[i] = true;
    key = 1;
    while (waiting[i] && key == 1)
        key = compare_and_swap(&lock,0,1);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = 0;
    else
        waiting[j] = false;
    /* remainder section */
}
```





Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans.
- For example, the **increment()** operation on the atomic variable **sequence** ensures **sequence** is incremented without interruption:
increment(&sequence);
- The **increment()** function can be implemented as follows:

```
void increment(atomic_int *v) {  
    int temp;  
  
    do {  
        temp = *v;  
    }while (temp != (compare_and_swap(v, temp, temp+1));  
}
```





Atomic Variables

- The **increment()** function can be implemented as follows:

```
void increment(atomic_int *v)
{
    int temp;

    do
    {
        temp = *v;
    }while (temp != (compare_and_swap(v, temp, temp+1)));
}
```

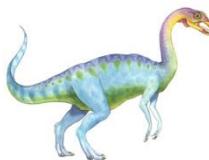




Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is **mutex lock**
- Protect a critical section by first **acquire()** a lock then **release()** the lock
 - Boolean variable indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
 - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**
 - This lock therefore called a spinlock

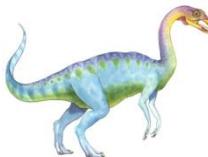




Solution to Critical-section Problem Using Locks

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





Mutex Lock Definitions

- `acquire() {
 while (!available)
 ; /* busy wait */
 available = false;;
}

□ release() {
 available = true;
}`

These two functions must be implemented **atomically**. Both test-and-set and compare-and-swap can be used to implement these functions.





Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process 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()**)

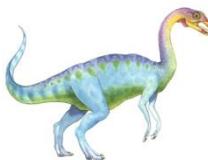
- Definition of the **wait() operation**

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

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

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





Semaphore Usage

- **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 solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2
Create a semaphore “**synch**” initialized to 0

P1:

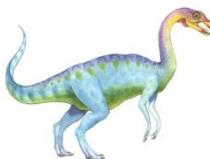
```
S1;  
    signal(synch);
```

P2:

```
wait(synch);  
S2;
```

- Can implement a counting semaphore **S** as a binary semaphore





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
 - Could now have **busy waiting** in critical section implementation
 - ▶ But implementation code is short
 - ▶ Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

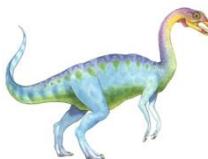




Semaphore Implementation with no Busy waiting

- 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
- Two operations:
 - **block** – place the process invoking the operation on the appropriate waiting queue
 - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
- ```
typedef struct {
 int value;
 struct process *list;
} semaphore;
```





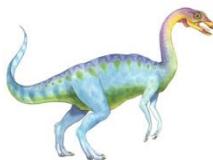
## Implementation with no Busy waiting (Cont.)

```
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);
 }
}
```





# Problems with Semaphores

- Incorrect use of semaphore operations:
  - `signal (mutex) ... wait (mutex)`
  - `wait (mutex) ... wait (mutex)`
  - Omitting of `wait (mutex)` and/or `signal (mutex)`
- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.





# 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
- Pseudocode syntax of a monitor:

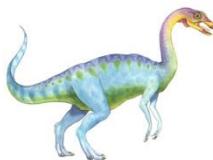
```
monitor monitor-name
{
 // shared variable declarations
 function P1 (...) { }

 function P2 (...) { }

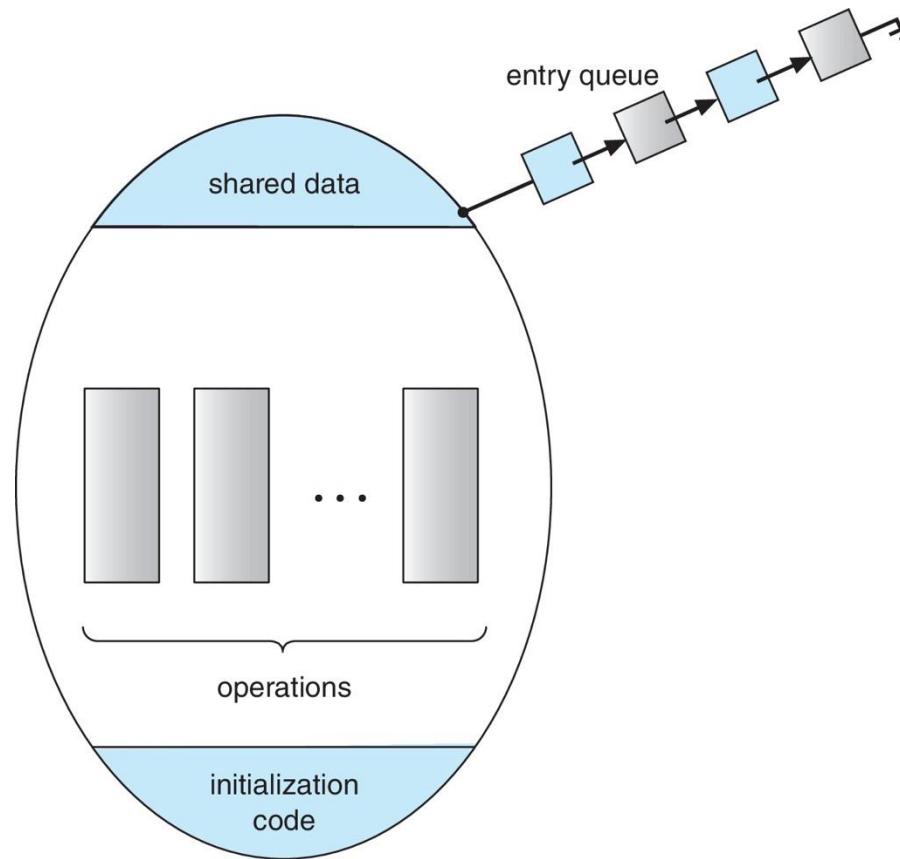
 function Pn (...) {.....}

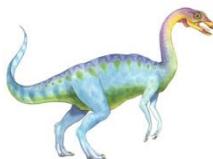
 initialization code (...) { ... }
}
```





# Schematic view of a Monitor

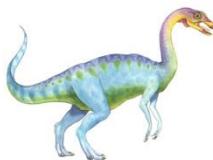




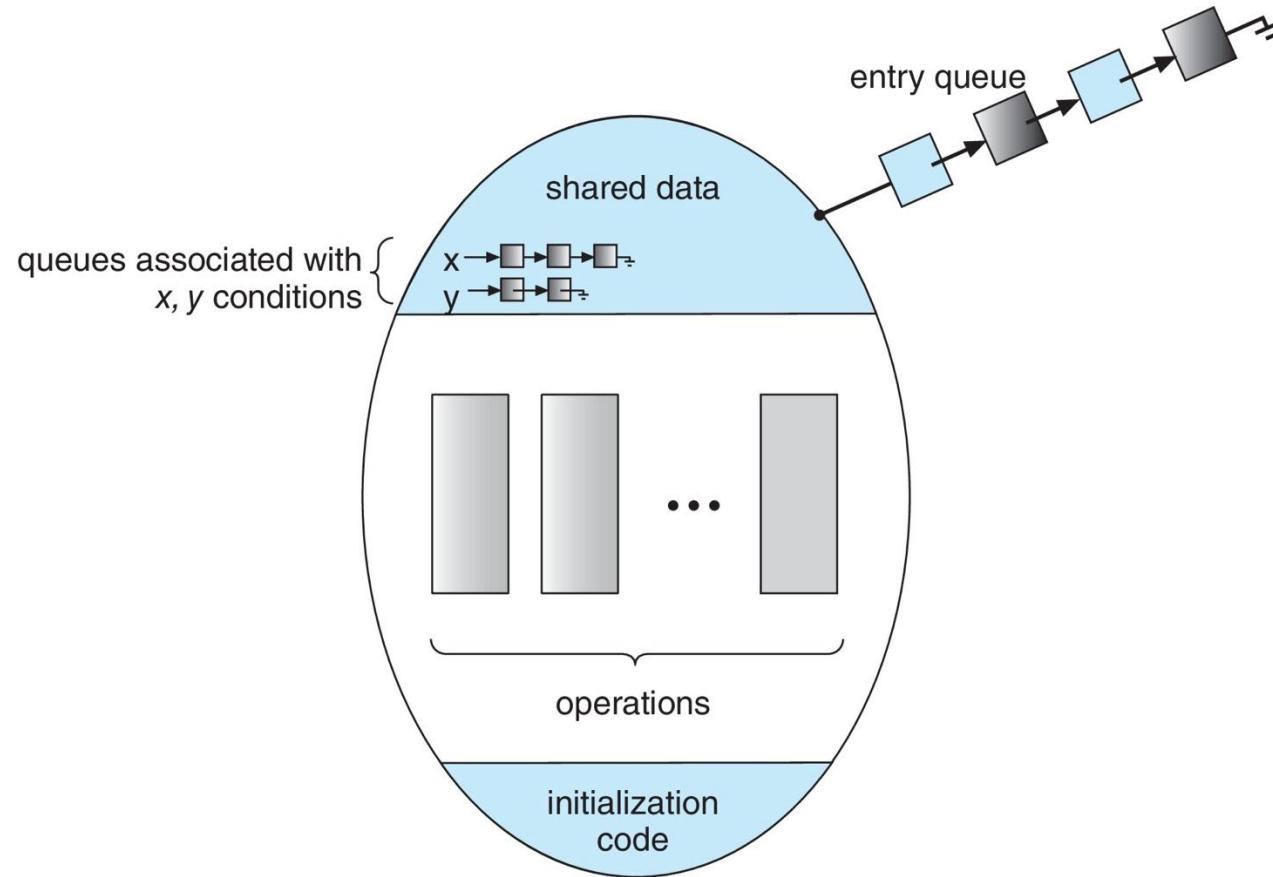
# Condition Variables

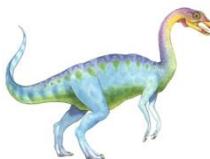
- **condition x, y;**
- Two operations are allowed on a condition variable:
  - **x.wait()** – a process that invokes the operation is suspended until **x.signal()**
  - **x.signal()** – resumes one of processes (if any) that invoked **x.wait()**
    - ▶ If no **x.wait()** on the variable, then it has 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 waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - ▶ P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java





# Monitor Implementation Using Semaphores

- Variables

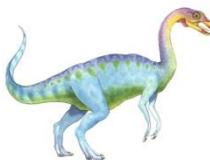
```
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next_count = 0;
```

- Each function *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 Implementation – Condition Variables

- 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--;
```





# Monitor 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 queued on condition variable `x`, and `x.signal()` is executed, which process 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 (highest priority) is scheduled next





## Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

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

```
...
R.release;
```

- Where R is an instance of type **ResourceAllocator**





# A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
 boolean busy;
 condition x;
 void acquire(int time) {
 if (busy)
 x.wait(time);
 busy = true;
 }
 void release() {
 busy = FALSE;
 x.signal();
 }
 initialization code() {
 busy = false;
 }
}
```





# Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
- **Liveness** refers to a set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a liveness failure.





# Liveness

- **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> |

- Consider if  $P_0$  executes `wait(S)` and  $P_1$  `wait(Q)`. When  $P_0$  executes `wait(Q)`, it must wait until  $P_1$  executes `signal(Q)`
- However,  $P_1$  is waiting until  $P_0$  execute `signal(S)`.
- Since these `signal()` operations will never be executed,  $P_0$  and  $P_1$  are **deadlocked**.





# Liveness

- Other forms of deadlock:
- **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**





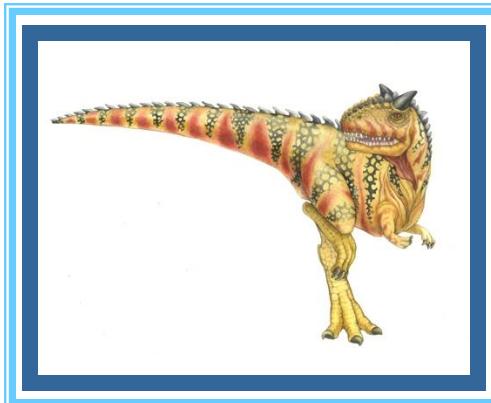
# Priority Inheritance Protocol

- Consider the scenario with three processes **P1**, **P2**, and **P3**. **P1** has the highest priority, **P2** the next highest, and **P3** the lowest. Assume a resource **P3** is assigned a resource **R** that **P1** wants. Thus, **P1** must wait for **P3** to finish using the resource. However, **P2** becomes runnable and preempts **P3**. What has happened is that **P2** - a process with a lower priority than **P1** - has indirectly prevented **P3** from gaining access to the resource.
  
- To prevent this from occurring, a **priority inheritance protocol** is used. This simply allows the priority of the highest thread waiting to access a shared resource to be assigned to the thread currently using the resource. Thus, the current owner of the resource is assigned the priority of the highest priority thread wishing to acquire the resource.



# Examples of Synchronization

## Chapter 7

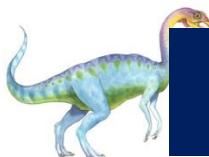




# Chapter 7: Synchronization Examples

- Explain the bounded-buffer, readers-writers, and dining philosophers synchronization problems.
- Describe the tools used by Linux and Windows to solve synchronization problems.
- Illustrate how POSIX and Java can be used to solve process synchronization problems.

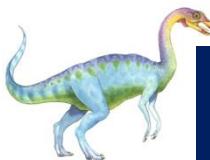




# Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

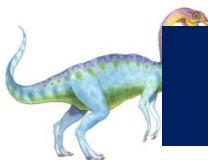




# Bounded-Buffer Problem

- $n$  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

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





# Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
 wait(full);
 wait(mutex);

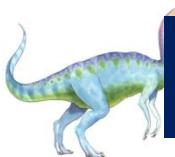
 ...
 /* remove an item from buffer to next_consumed */

 ...
 signal(mutex);
 signal(empty);

 ...
 /* consume the item in next_consumed */

 ...
}
```



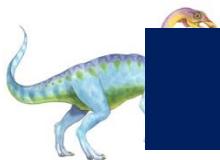


# Readers-Writers Problem

---

- 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
  - Data set
  - 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

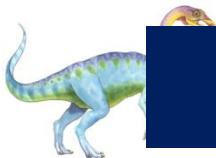
```
while (true) {
 wait(rw_mutex);

 ...
 /* writing is performed */

 ...

 signal(rw_mutex);
}
```





# Readers-Writers Problem (Cont.)

- The structure of a reader process

```
while (true) {
 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);
}
```



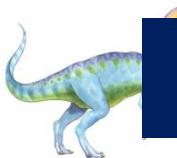


# Readers-Writers Problem Variations

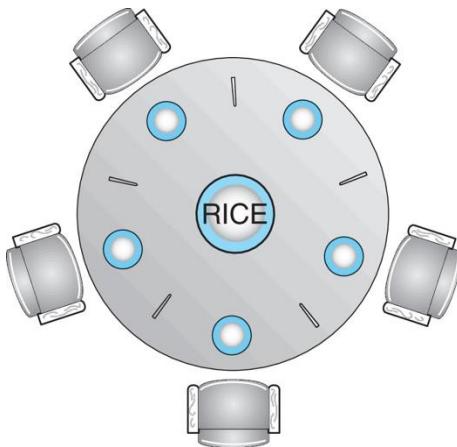
---

- **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
- Don't 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
  - Shared data
    - Bowl of rice (data set)
    - Semaphore **chopstick [5]** initialized to 1





# Dining-Philosophers Problem Algorithm

- Semaphore Solution
- The structure of Philosopher  $i$ :

```
while (true) {
 wait (chopstick[i]);
 wait (chopStick[(i + 1) % 5]);

 /* eat for awhile */

 signal (chopstick[i]);
 signal (chopstick[(i + 1) % 5]);

 /* think for awhile */
}
```

- What is the problem with this algorithm?





# Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
 enum { THINKING, HUNGRY, EATING} state [5] ;
 condition self [5];

 void pickup (int i) {
 state[i] = HUNGRY;
 test(i);
 if (state[i] != EATING) self[i].wait;
 }

 void putdown (int i) {
 state[i] = THINKING;
 // test left and right neighbors
 test((i + 4) % 5);
 test((i + 1) % 5);
 }
}
```

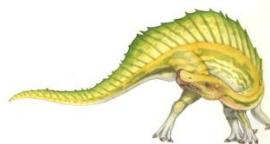


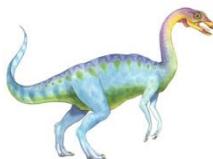


# Solution to Dining Philosophers (Cont.)

```
void test (int i) {
 if ((state[(i + 4) % 5] != EATING) &&
 (state[i] == HUNGRY) &&
 (state[(i + 1) % 5] != EATING)) {
 state[i] = EATING ;
 self[i].signal () ;
 }
}

initialization_code() {
 for (int i = 0; i < 5; i++)
 state[i] = THINKING;
}
}
```





# Solution to Dining Philosophers (Cont.)

---

- Each philosopher  $i$  invokes the operations **pickup ()** and **putdown ()** in the following sequence:

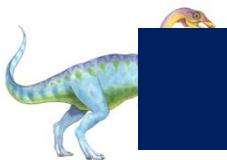
```
DiningPhilosophers.pickup(i) ;
```

```
 /** EAT **/
```

```
DiningPhilosophers.putdown(i) ;
```

- No deadlock, but starvation is possible

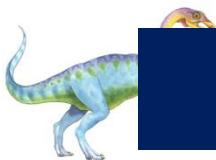




# Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - atomic integers
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption





# Linux Synchronization

- Atomic variables

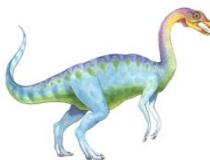
`atomic_t` is the type for atomic integer

- Consider the variables

```
atomic_t counter;
int value;
```

| <i>Atomic Operation</i>                         | <i>Effect</i>                       |
|-------------------------------------------------|-------------------------------------|
| <code>atomic_set(&amp;counter,5);</code>        | <code>counter = 5</code>            |
| <code>atomic_add(10,&amp;counter);</code>       | <code>counter = counter + 10</code> |
| <code>atomic_sub(4,&amp;counter);</code>        | <code>counter = counter - 4</code>  |
| <code>atomic_inc(&amp;counter);</code>          | <code>counter = counter + 1</code>  |
| <code>value = atomic_read(&amp;counter);</code> | <code>value = 12</code>             |



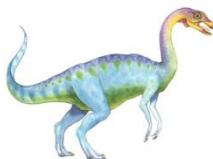


# POSIX Synchronization

---

- POSIX API provides
  - mutex locks
  - semaphores
  - condition variable
- Widely used on UNIX, Linux, and macOS





# POSIX Mutex Locks

## □ Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

## □ Acquiring and releasing the lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

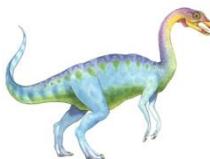




# POSIX Semaphores

- POSIX provides two versions – **named** and **unnamed**.
- Named semaphores can be used by unrelated processes, unnamed cannot.





# POSIX Named Semaphores

- Creating an initializing the semaphore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

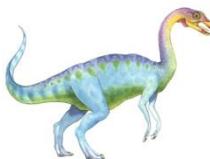
- Another process can access the semaphore by referring to its name **SEM**.
- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```





# POSIX Unnamed Semaphores

- Creating and initializing the semaphore:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```





# POSIX Condition Variables

- Since POSIX is typically used in C/C++ and these languages do not provide a monitor, POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion: Creating and initializing the condition variable:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```





# POSIX Condition Variables

- Thread waiting for the condition `a == b` to become true:

```
pthread_mutex_lock(&mutex);
while (a != b)
 pthread_cond_wait(&cond_var, &mutex);

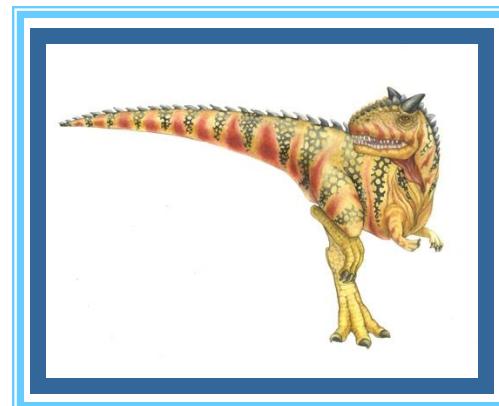
pthread_mutex_unlock(&mutex);
```

- Thread signaling another thread waiting on the condition variable:

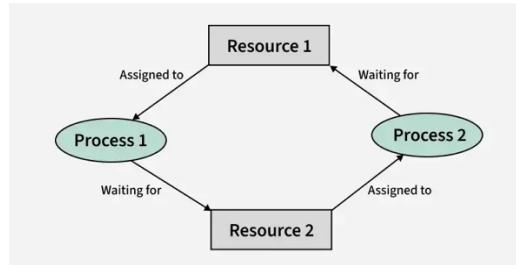
```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```



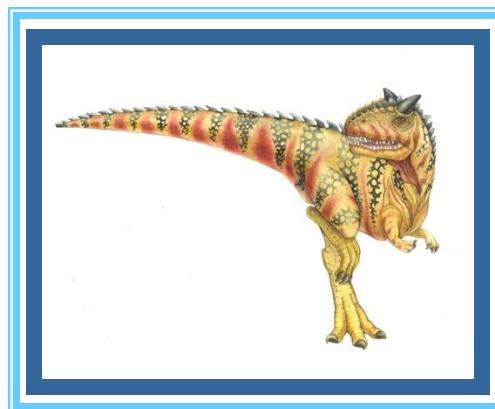
# End of Week 6



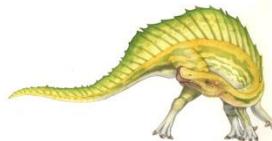
# Week 7



# Deadlocks



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





## 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 in Multithreaded Application

- Two mutex locks are created and initialized:

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);
```





# Deadlock in Multithreaded Application

```
/* 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);

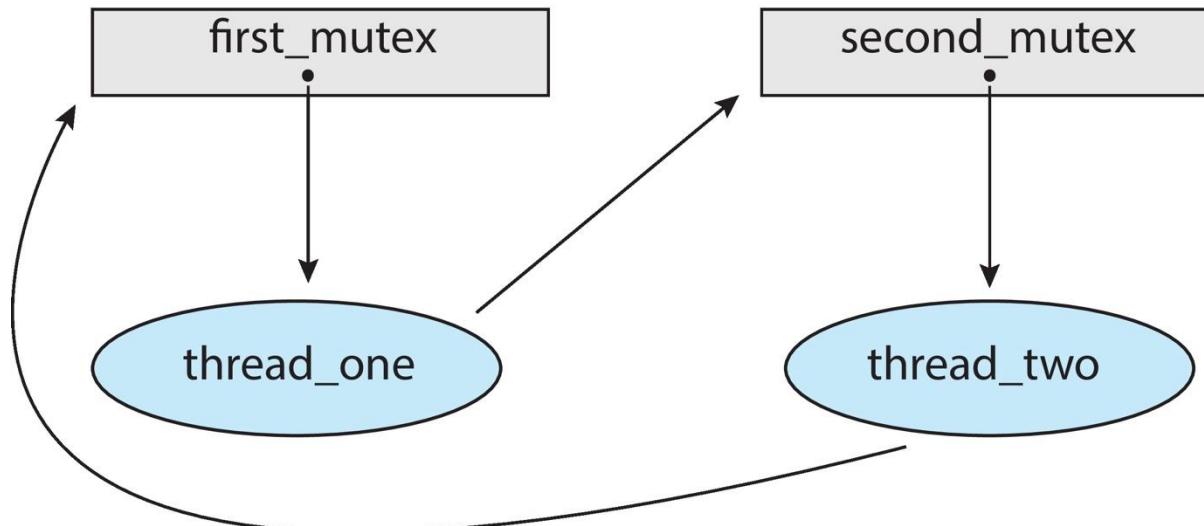
 pthread_exit(0);
}
```

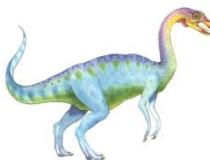




# Deadlock in Multithreaded Application

- Deadlock is possible if thread 1 acquires `first_mutex` and thread 2 acquires `second_mutex`. Thread 1 then waits for `second_mutex` and thread 2 waits for `first_mutex`.
- Can be illustrated with a **resource allocation graph**:



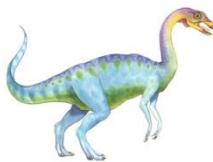


## Deadlock Characterization

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  $E$ .

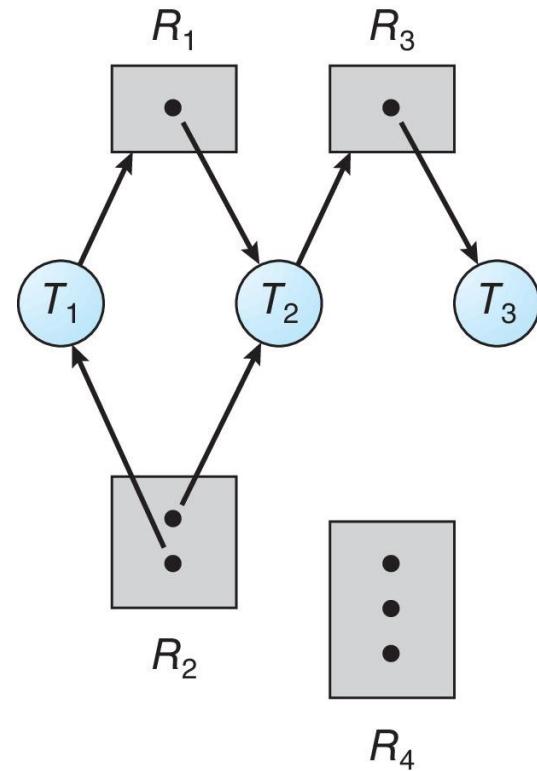
- $V$  is partitioned into two types:
  - $P = \{P_1, P_2, \dots, P_n\}$ , the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system
- **request edge** – directed edge  $P_i \rightarrow R_j$
- **assignment edge**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

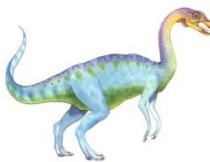




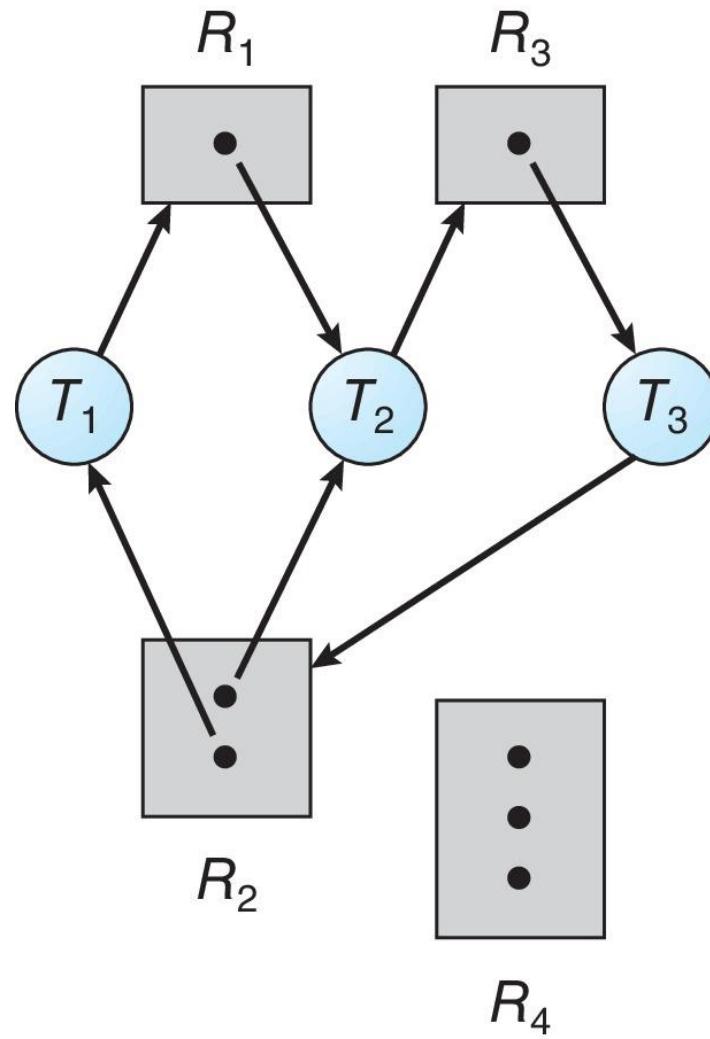
## Resource Allocation Graph Example

- One instance of R1
- Two instances of R2
- One instance of R3
- Three instances of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holding one instance of R3



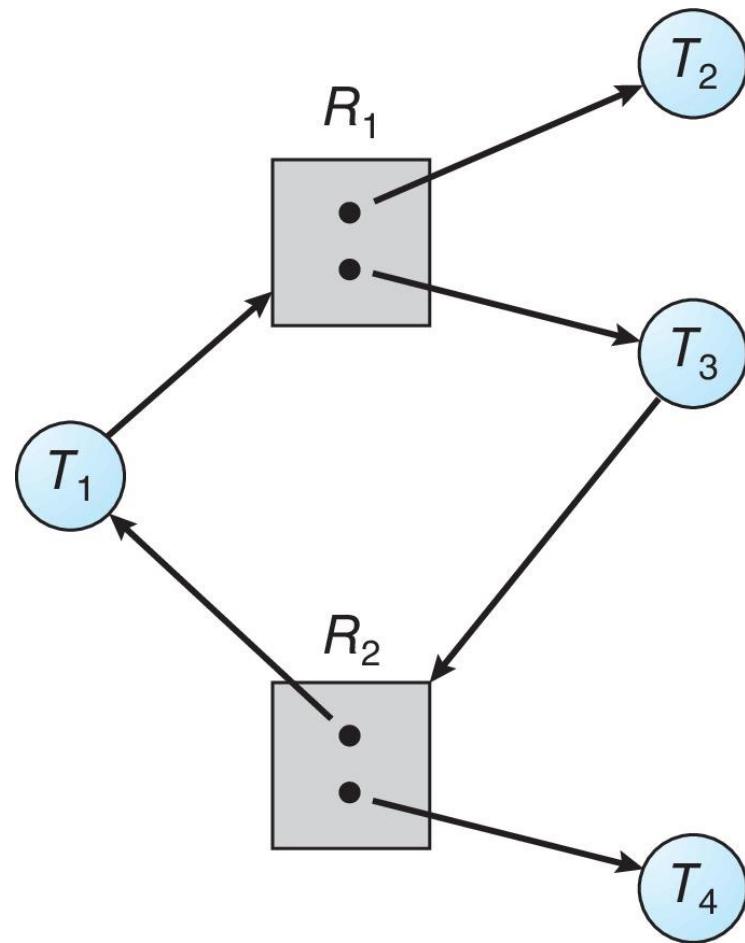


# Resource Allocation Graph With A Deadlock





## Graph With A Cycle But No Deadlock





## 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
  - if several instances per resource type, possibility of deadlock

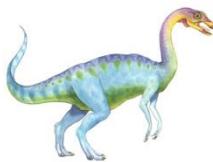




## Methods for Handling Deadlocks

- 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.



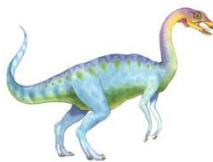


## Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **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 –**

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held 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



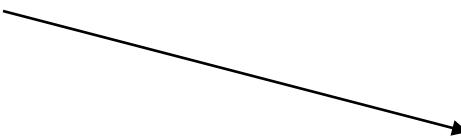


## Circular Wait

- Invalidating the circular wait condition is most common.
- Simply assign each resource (i.e. mutex locks) a unique number.
- Resources must be acquired in order.
- If:

```
first_mutex = 1
second_mutex = 5
```

code for **thread\_two** could not be written as follows:



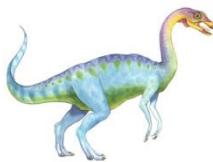
```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
 pthread_mutex_lock(&first_mutex);
 pthread_mutex_lock(&second_mutex);
 /**
 * Do some work
 */
 pthread_mutex_unlock(&second_mutex);
 pthread_mutex_unlock(&first_mutex);

 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 Avoidance

Requires that the system has some additional *a priori* information available

- 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

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a 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  $P_j$  have finished
  - When  $P_j$  is finished,  $P_i$  can obtain needed resources, execute, return allocated resources, and terminate
  - When  $P_i$  terminates,  $P_{i+1}$  can obtain its needed resources, and so on

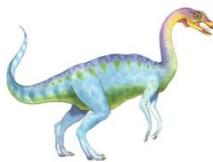




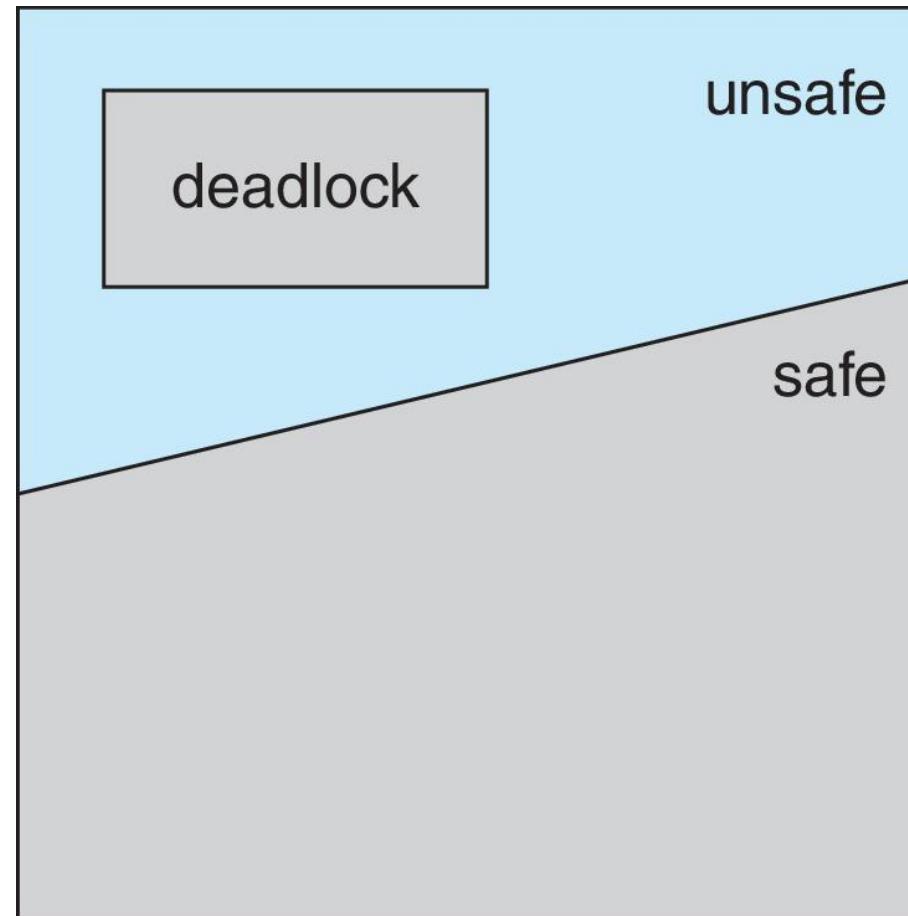
## Basic Facts

- If a system is in safe state  $\Rightarrow$  no deadlocks
- If a system is in unsafe state  $\Rightarrow$  possibility of deadlock
- Avoidance  $\Rightarrow$  ensure that a system will never enter an unsafe state.





# Safe, Unsafe, Deadlock State





## Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the Banker's Algorithm





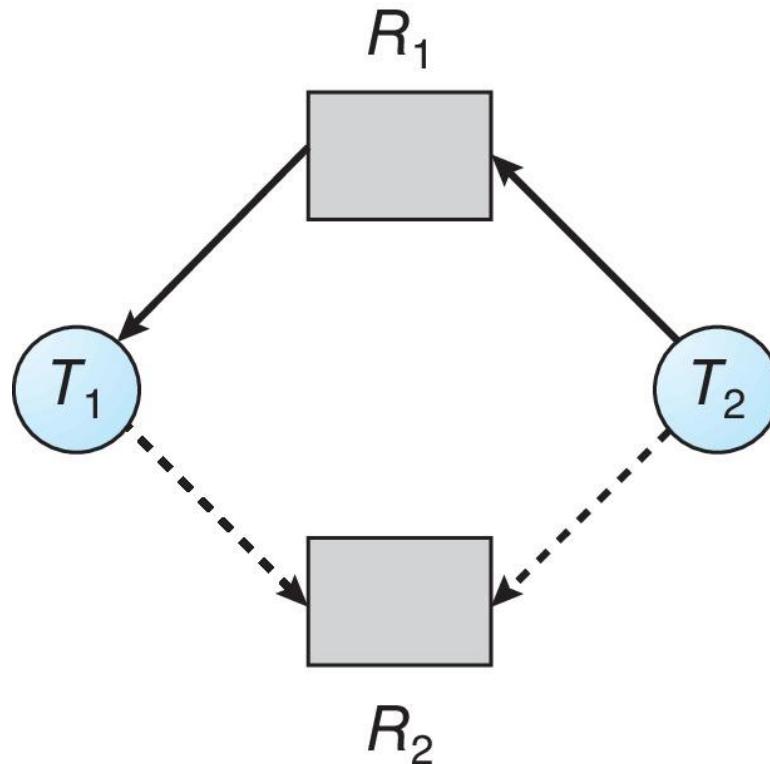
## Resource-Allocation Graph Scheme

- **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



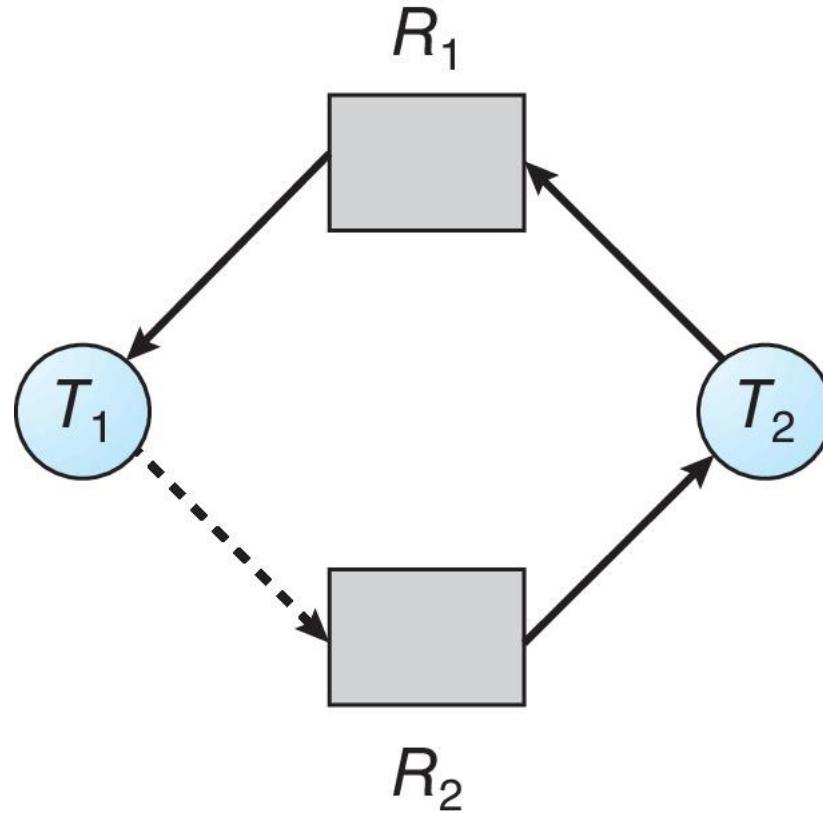


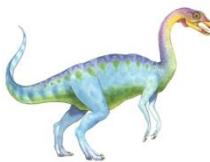
# Resource-Allocation Graph





## Unsafe State In Resource-Allocation Graph





# 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





## Banker's Algorithm

- Multiple instances of resources
- 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





## Data Structures for the Banker's Algorithm

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

- **Available:** Vector of length  $m$ . If  $\text{available}[j] = k$ , 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  $k$  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<sub>i</sub> ≤ Work**If no such  $i$  exists, go to step 4
3. **Work = Work + Allocation<sub>i</sub>**,  
**Finish[i] = true**  
go to step 2
4. If **Finish [i] == true** for all  $i$ , then the system is in a safe state





## Resource-Request Algorithm for Process $P_i$

$\text{Request}_i$  = request vector for process  $P_i$ . If  $\text{Request}_i[j] = k$  then 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> |   |
|-------|-------------------|---|---|------------|------------------|---|
|       | A                 | B | C | A          | B                | C |
| $P_0$ | 0                 | 1 | 0 | 7          | 5                | 3 |
| $P_1$ | 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 (Cont.)

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

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





## Example: $P_1$ Request (1,0,2)

- Check that Request  $\leq$  Available (that is,  $(1,0,2) \leq (3,3,2) \Rightarrow$  true

|       | <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       | 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
- Can request for (3,3,0) by  $P_4$  be granted?
- Can request for (0,2,0) by  $P_0$  be granted?

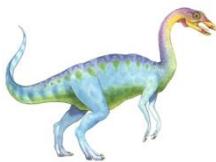




## Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme





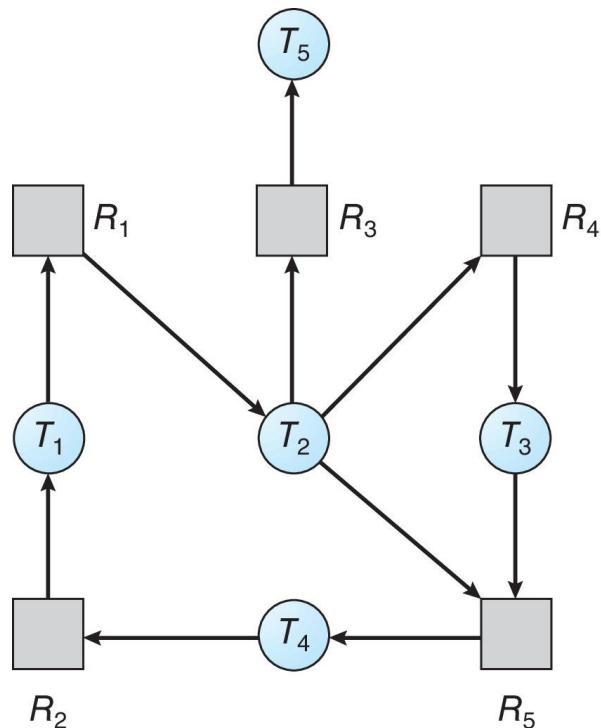
# Single Instance of Each Resource Type

- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$  if  $P_i$  is waiting for  $P_j$
- 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

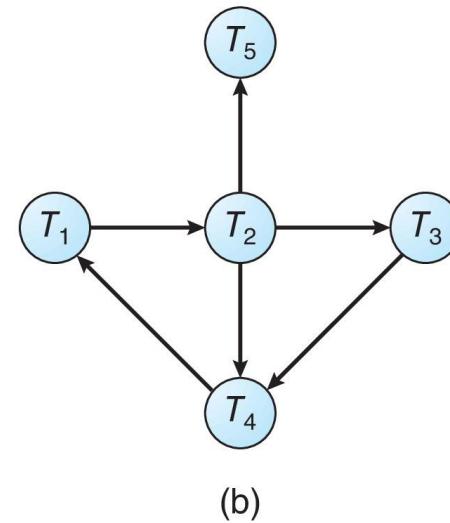




# Resource-Allocation Graph and Wait-for Graph



(a)



(b)

Resource-Allocation Graph

Corresponding wait-for graph





## Several Instances of a Resource Type

- **Available:** A vector of length  $m$  indicates the number of available resources of each type
- **Allocation:** An  $n \times m$  matrix defines the number of resources of each type currently allocated to each process
- **Request:** An  $n \times m$  matrix indicates the current request of each process. If  $\text{Request}[i][j] = k$ , then process  $P_i$  is requesting  $k$  more instances of resource type  $R_j$ .





## Detection Algorithm

1. Let **Work** and **Finish** be vectors of length  $m$  and  $n$ , respectively  
Initialize:
  - (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<sub>i</sub> ≤ Work**

If no such  $i$  exists, go to step 4





## Detection Algorithm (Cont.)

3.  $\text{Work} = \text{Work} + \text{Allocation}_i$ ,  
 $\text{Finish}[i] = \text{true}$   
go to step 2
4. If  $\text{Finish}[i] == \text{false}$ , for some  $i$ ,  $1 \leq i \leq n$ , then the system is in deadlock state. Moreover, if  $\text{Finish}[i] == \text{false}$ , then  $P_i$  is deadlocked

Algorithm requires an order of  $O(m \times n^2)$  operations to detect whether the system is in deadlocked state





## 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> |   |   |
|-------|-------------------|---|---|----------------|---|---|------------------|---|---|
|       | 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 | 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 (Cont.)

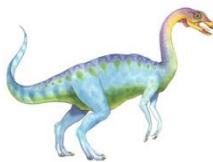
- $P_2$  requests an additional instance of type **C**

Request

|       | A | B | C |
|-------|---|---|---|
| $P_0$ | 0 | 0 | 0 |
| $P_1$ | 2 | 0 | 2 |
| $P_2$ | 0 | 0 | 1 |
| $P_3$ | 1 | 0 | 0 |
| $P_4$ | 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

- When, and how often, to invoke 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.

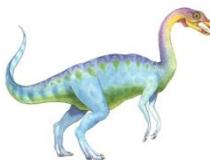




## Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. 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 Week 7

