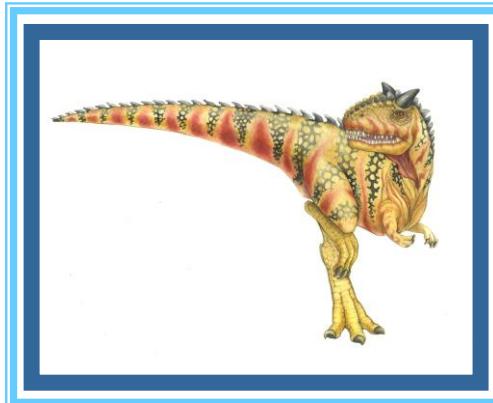


# Chapter 1: Introduction





# Chapter 1: Introduction

---

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



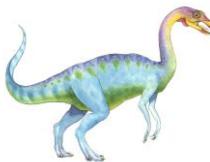


# Objectives

---

- 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





# What Does the Term Operating System Mean?

---

- An operating system is “(fill in the blanks)”
- What about:
  - Car
  - Airplane
  - Printer
  - Washing Machine
  - Toaster
  - Compiler
  - Etc.





# What is an Operating System?

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





# Computer System Structure

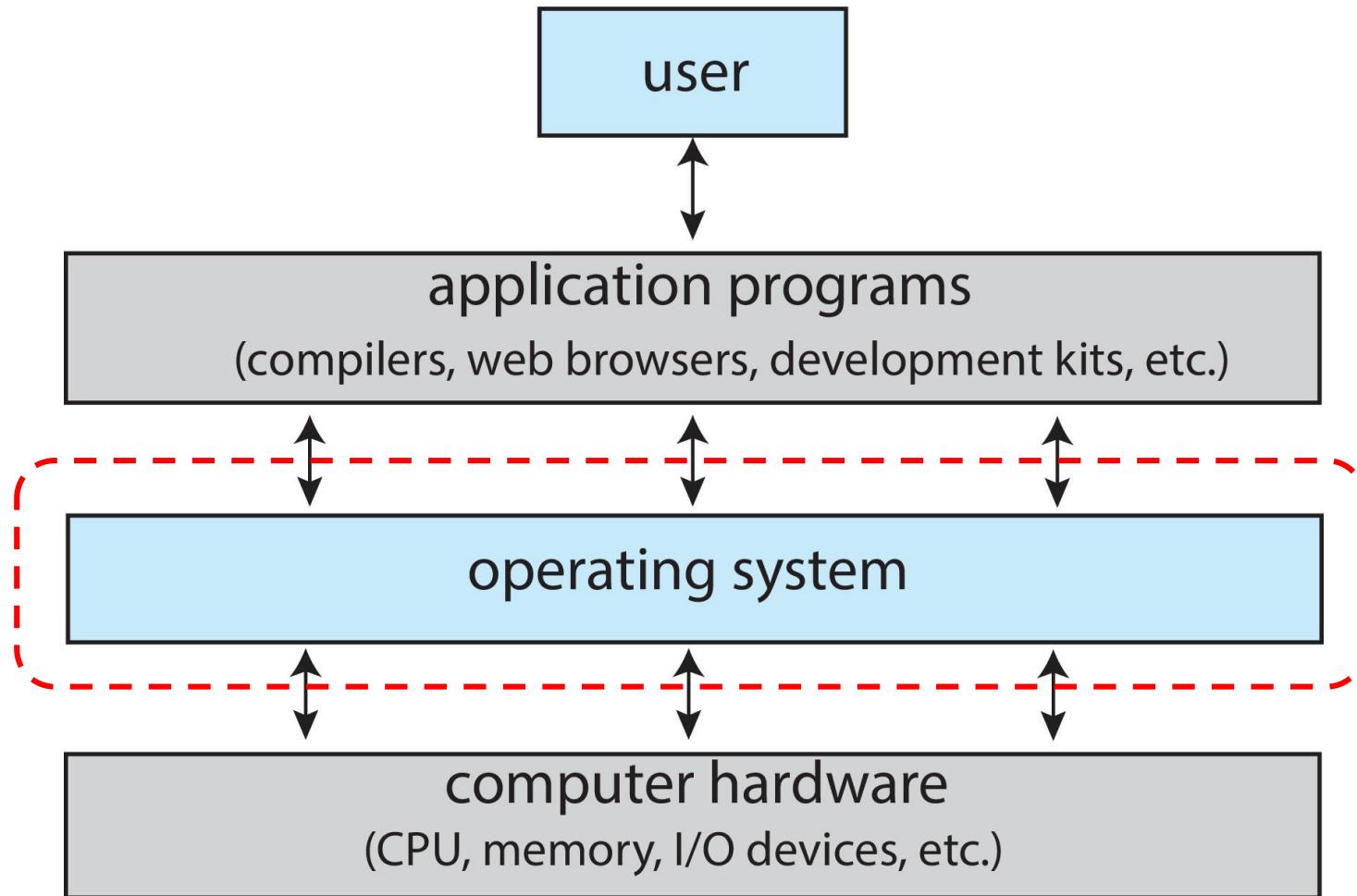
---

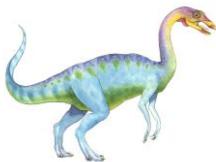
- Computer system can be divided into four components:
  - **Hardware** – provides basic computing resources
    - ▶ CPU, memory, I/O devices
  - **Operating system**
    - ▶ Controls and coordinates use of hardware among various applications and users
  - **Application programs** – define the ways in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





# Abstract View of Components of Computer

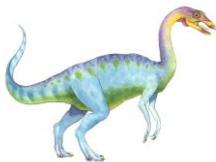




# What Operating Systems Do

- 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
  - OS is a **resource allocator** and **control program** making efficient use of HW and managing execution of user programs





# What Operating Systems Do

- 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

---

- The 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

- No universally accepted definition
- “Everything a vendor ships when you order an operating system” is a good approximation
  - But varies wildly
- A more common definition: “The one program running at all times on the computer” is the **kernel**, part of the OS

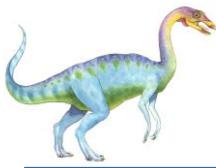




# Operating System Definition

- Everything else is either
  - A ***system program*** (ships with the OS, but not part of the kernel) , or
  - An ***application program***, all programs not associated with the OS
- Today's OSes for general purpose and mobile computing also include ***middleware*** – a set of software frameworks that provide additional services to application developers such as databases, multimedia, graphics





---

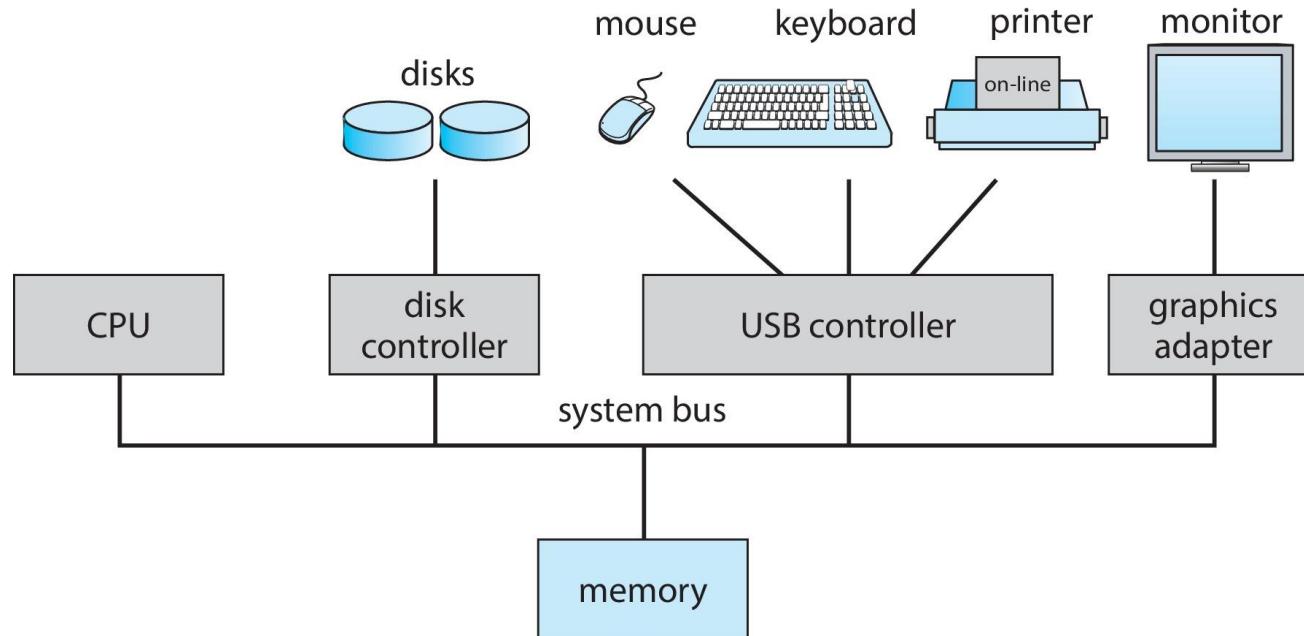
# Overview of Computer System Structure

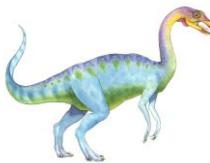




# Computer System Organization

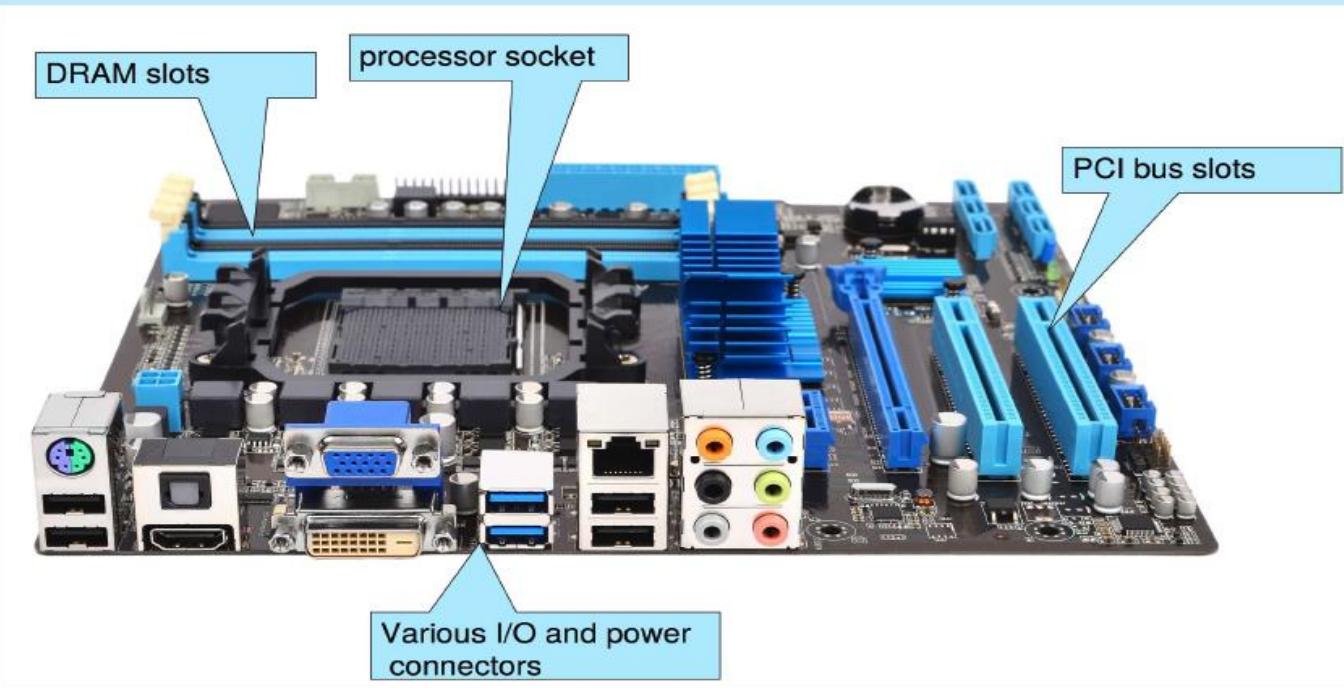
- Computer-system operation
  - One or more CPUs, device controllers connect through common **bus** providing access to shared memory
  - **Concurrent** execution of CPUs and devices competing for memory cycles





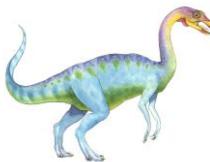
# PC Motherboard

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.





# 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 OS **device driver** to manage it
- 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**
- CPU moves data from/to main memory to/from local buffers

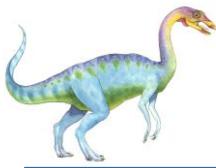




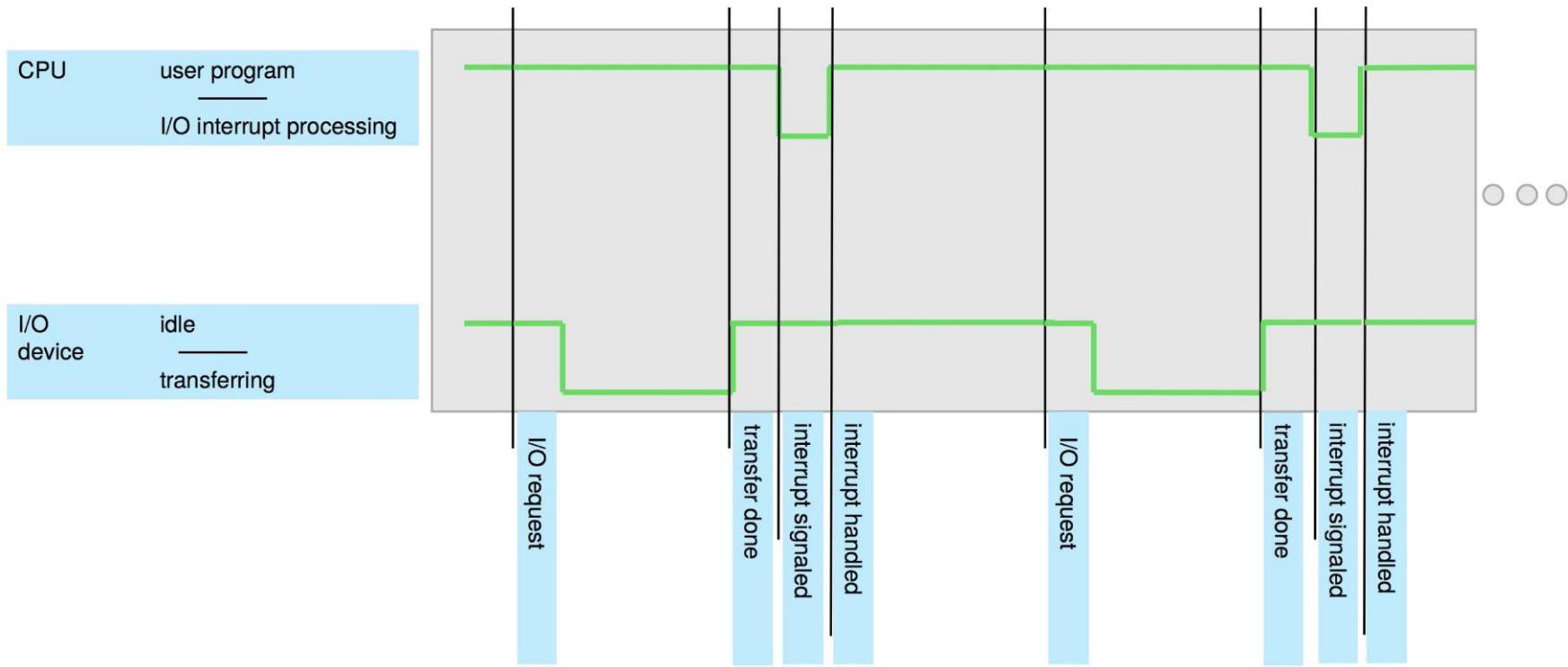
# Common Functions of Interrupts

- 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
- Modern OS is **interrupt-driven**





# Interrupt Timeline

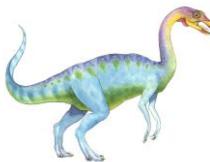




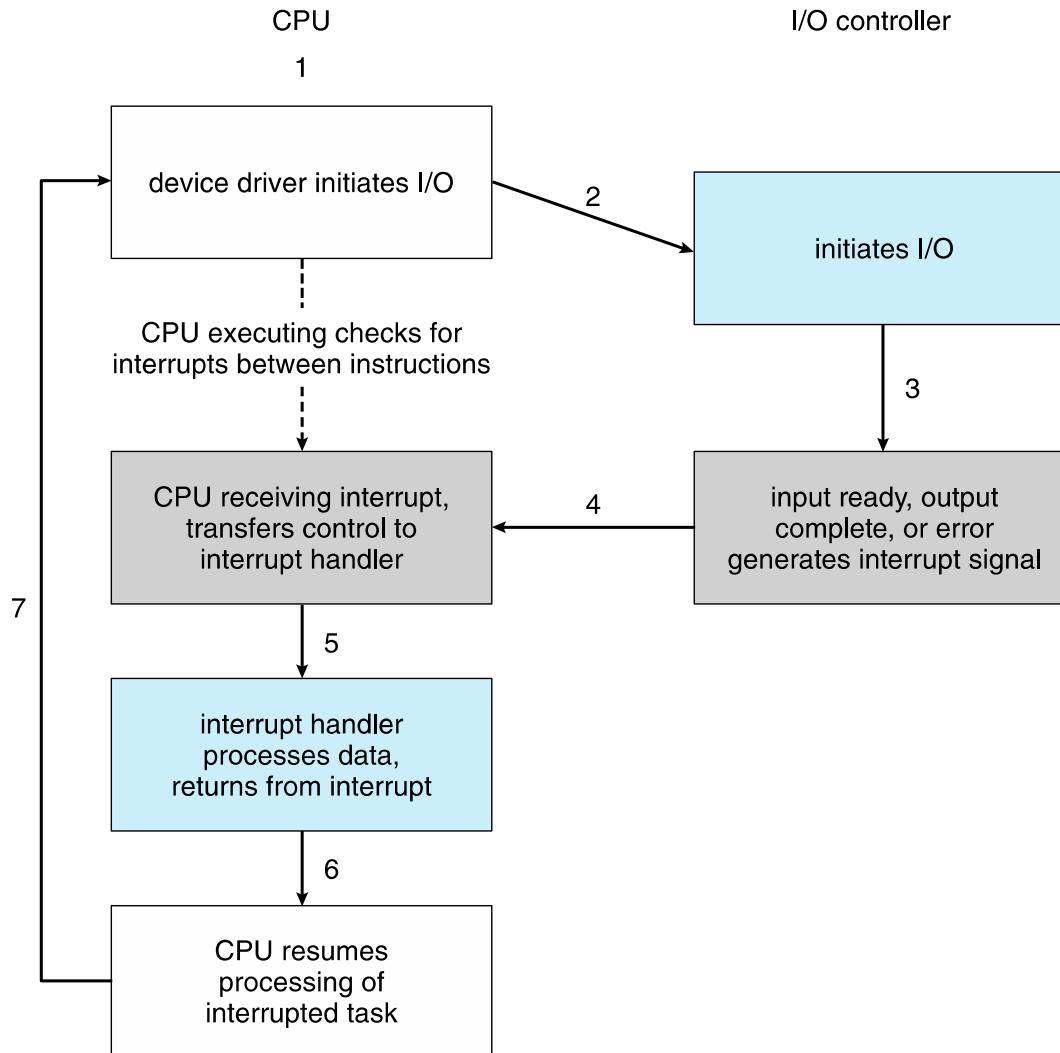
# Interrupt Handling

- The OS preserves the state of the CPU by storing the 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
  - interrupt service routine





# Interrupt-driven I/O Cycle





# I/O Structure

- Two methods for handling I/O
  - **Synchronous**: After I/O starts, control returns to user program **only upon I/O completion**
  - **Asynchronous**: After I/O starts, control returns to user program **without waiting** for I/O completion



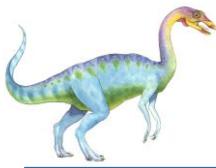


# I/O Structure (Cont.)

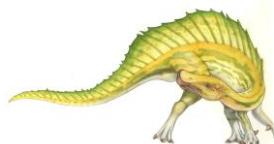
---

- Synchronous I/O
  - 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
- Asynchronous I/O
  - **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





# Storage Structure

---

- Main memory – the 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





# Storage Structure (Cont.)

---

- **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

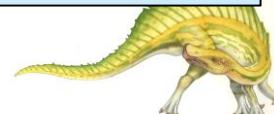


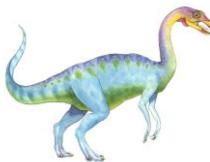


# Storage Definitions and Notation Review

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

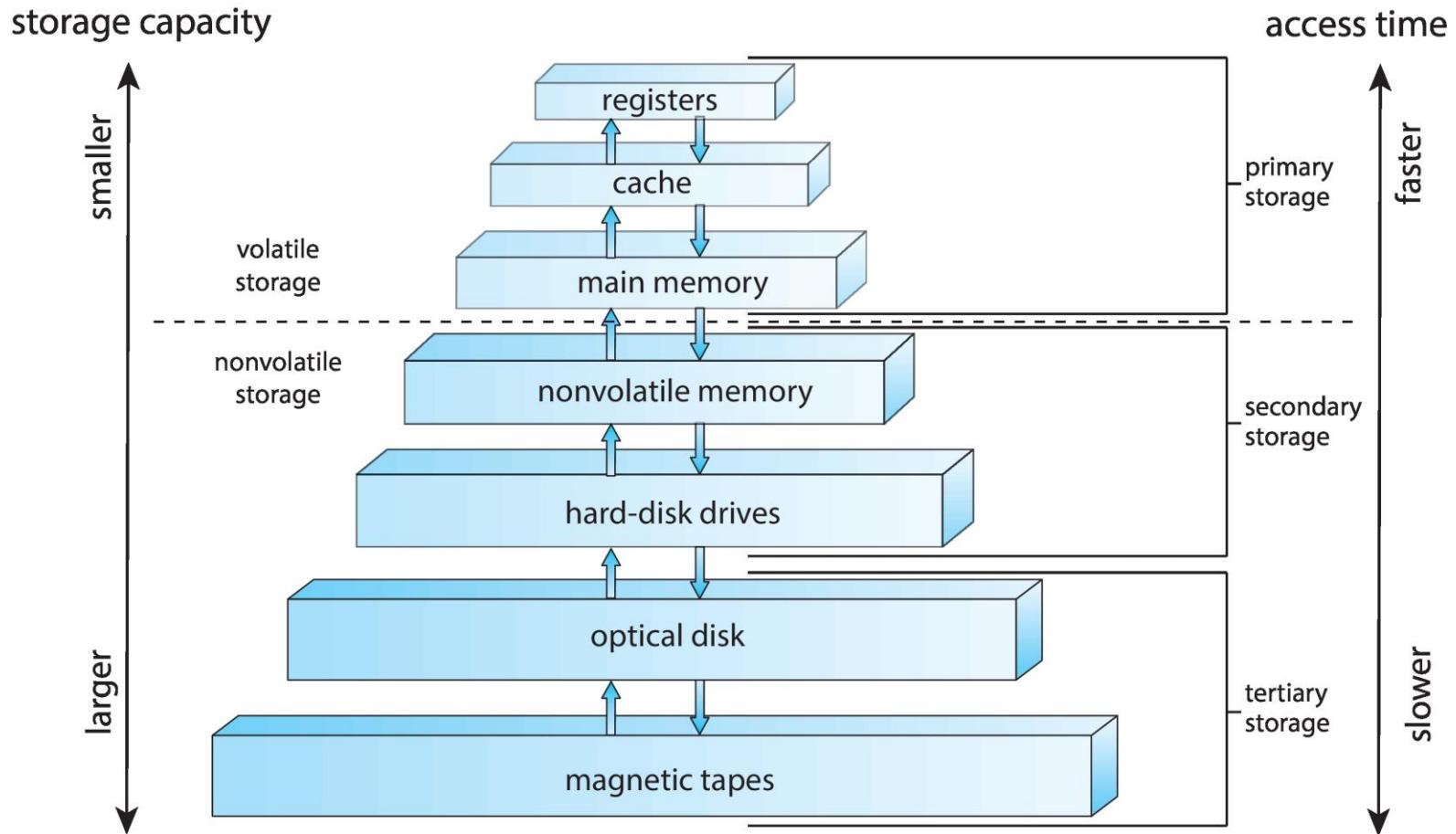
---

- 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



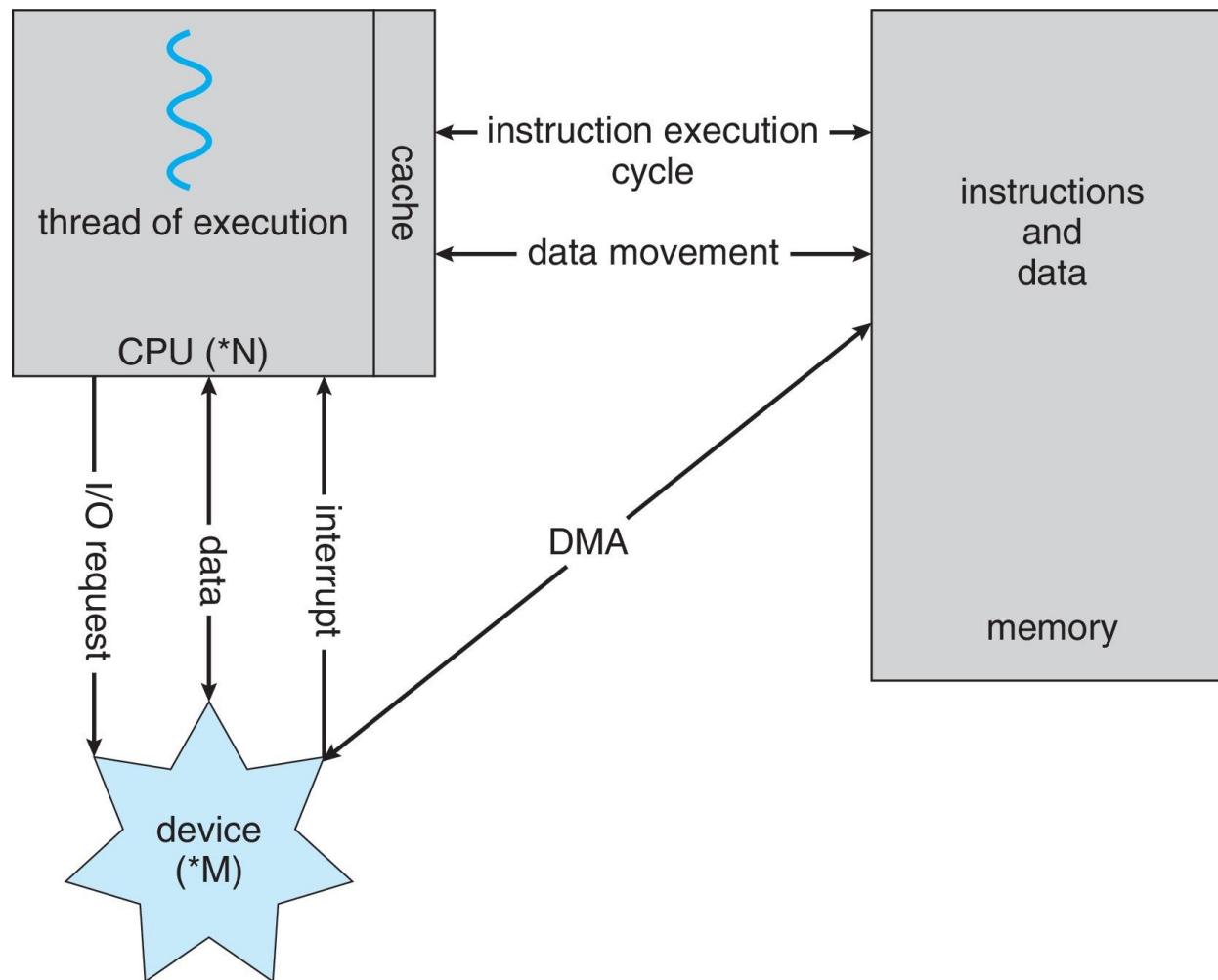


# Storage-Device Hierarchy





# How a Modern Computer Works



*A von Neumann architecture*



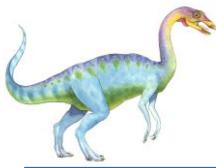


# Direct Memory Access Structure

---

- 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 one interrupt per byte





---

# Operating System Operations





# Operating-System Operations

---

- Bootstrap program – simple code to initialize the system, load the kernel
- Kernel loads
- Starts **system daemons** (services provided outside of the kernel)
- 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 OS service – **system call**
    - ▶ Other process problems include infinite loop, processes modifying each other or the OS

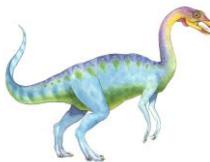




# Multiprogramming (Batch system)

- Single user cannot always keep CPU and I/O devices busy
- 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 job has to wait (for I/O for example), OS switches to another job





# Multitasking (Timesharing)

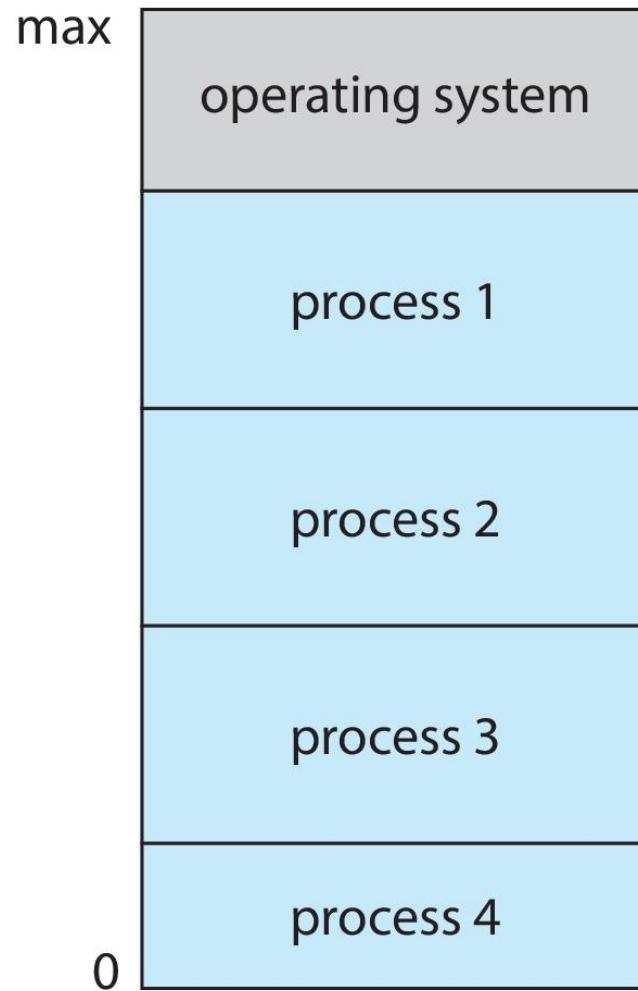
---

- A logical extension of Batch systems – the 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  $\Rightarrow$  **process**
  - If several jobs ready to run at the same time  $\Rightarrow$  **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 Operation

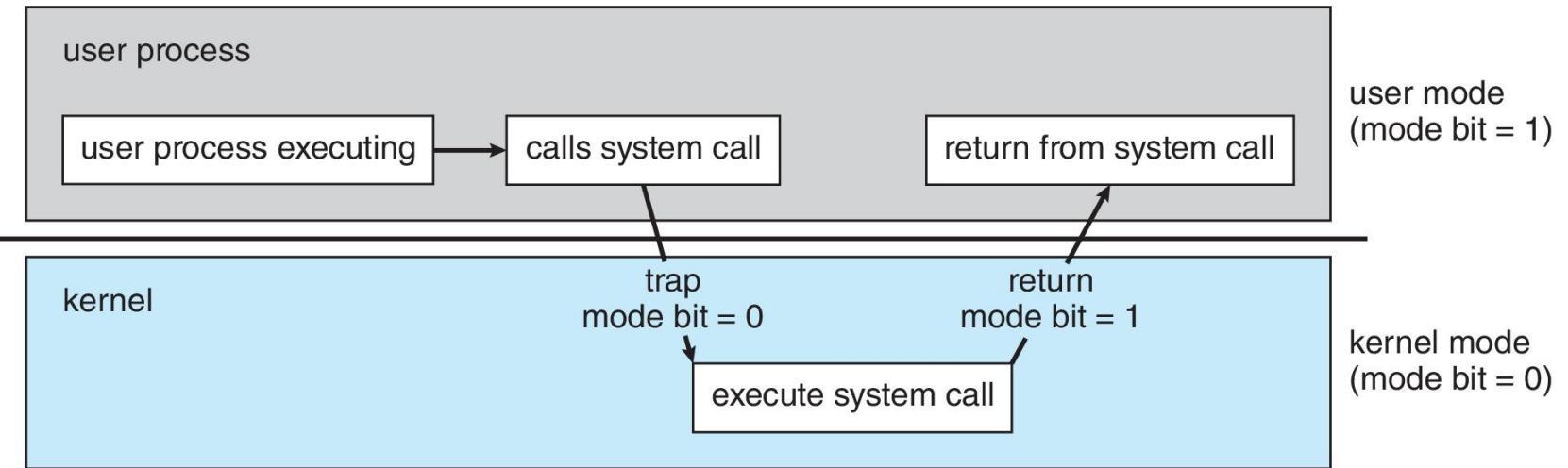
---

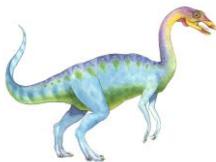
- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
- **Mode bit** provided by hardware
  - To distinguish when system is running user code or kernel code
  - When a user is running  $\Rightarrow$  mode bit is “user”
  - When kernel code is executing  $\Rightarrow$  mode bit is “kernel”
- How do we guarantee that user does not explicitly set the mode bit to “kernel”?
  - System call changes mode to kernel, return from call resets it to user
- Some instructions designated as **privileged**, only executable in kernel mode





# Transition from User to Kernel Mode





# Timer

- Timer to prevent infinite loop (or process hogging resources)
  - Timer is set to interrupt the computer after some time period
  - Keep a counter that is decremented by the physical clock
  - OS 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





# Process Management

- 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 OS 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

- Von Neumann architecture
  - 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

---

- 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



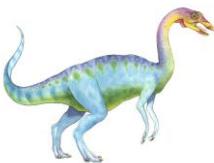


# Mass-Storage Management

---

- Storage management activities
  - Mounting and unmounting
  - Free-space management
  - Storage allocation
  - Disk scheduling
  - Partitioning
  - Protection





# 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



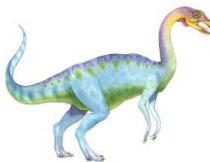


# I/O Subsystem

---

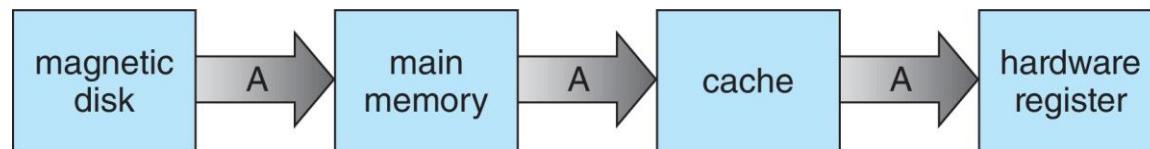
- One purpose of OS is to hide peculiarities of hardware devices from the user
- I/O subsystem responsible for
  - Memory management of I/O including
    - ▶ **buffering** (storing data temporarily while it is being transferred)
    - ▶ **caching** (storing part of data in faster storage for performance)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





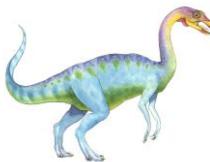
# Migration of data “A” from Disk to Register

- 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 Chap. 19



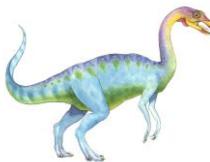


# Protection and Security

---

- **Protection** – any mechanism for controlling access of processes or users to resources defined by the OS
- **Security** – defense 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



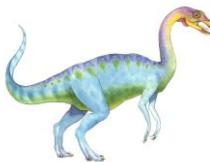


# Virtualization

---

- Allows OS 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 Win10 guests, each running applications, all on native Win10 **host** OS
  - **VMM** (virtual machine manager) provides virtualization services





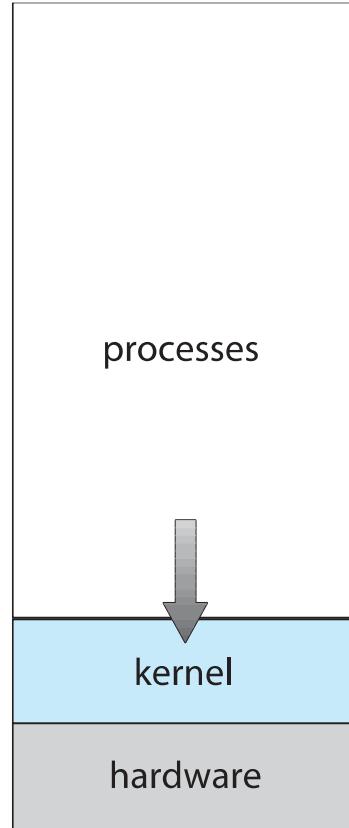
# Virtualization (cont.)

- 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
  - Quality assurance 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)

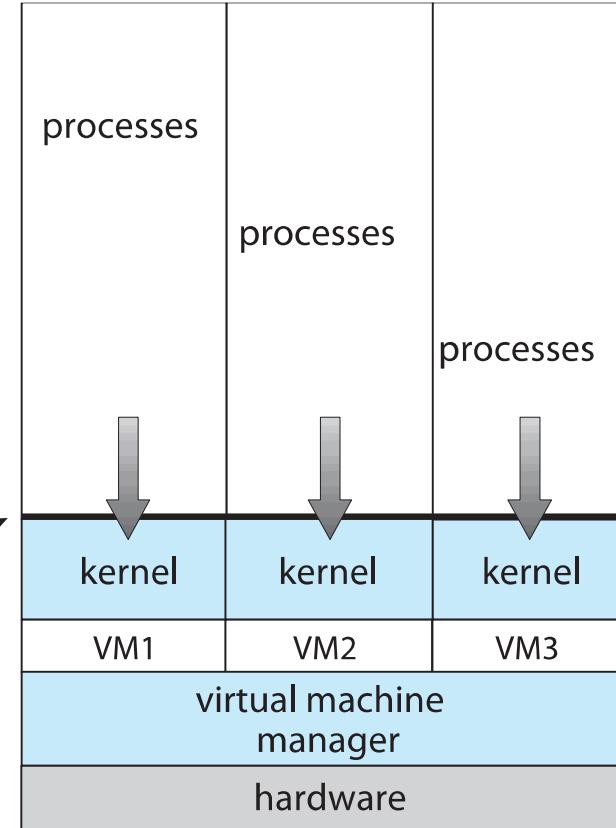




# Computing Environments - Virtualization

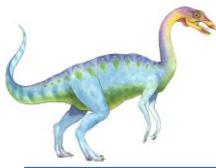


(a)



(b)





---

# Computer System Architecture





# Computer-System Architecture

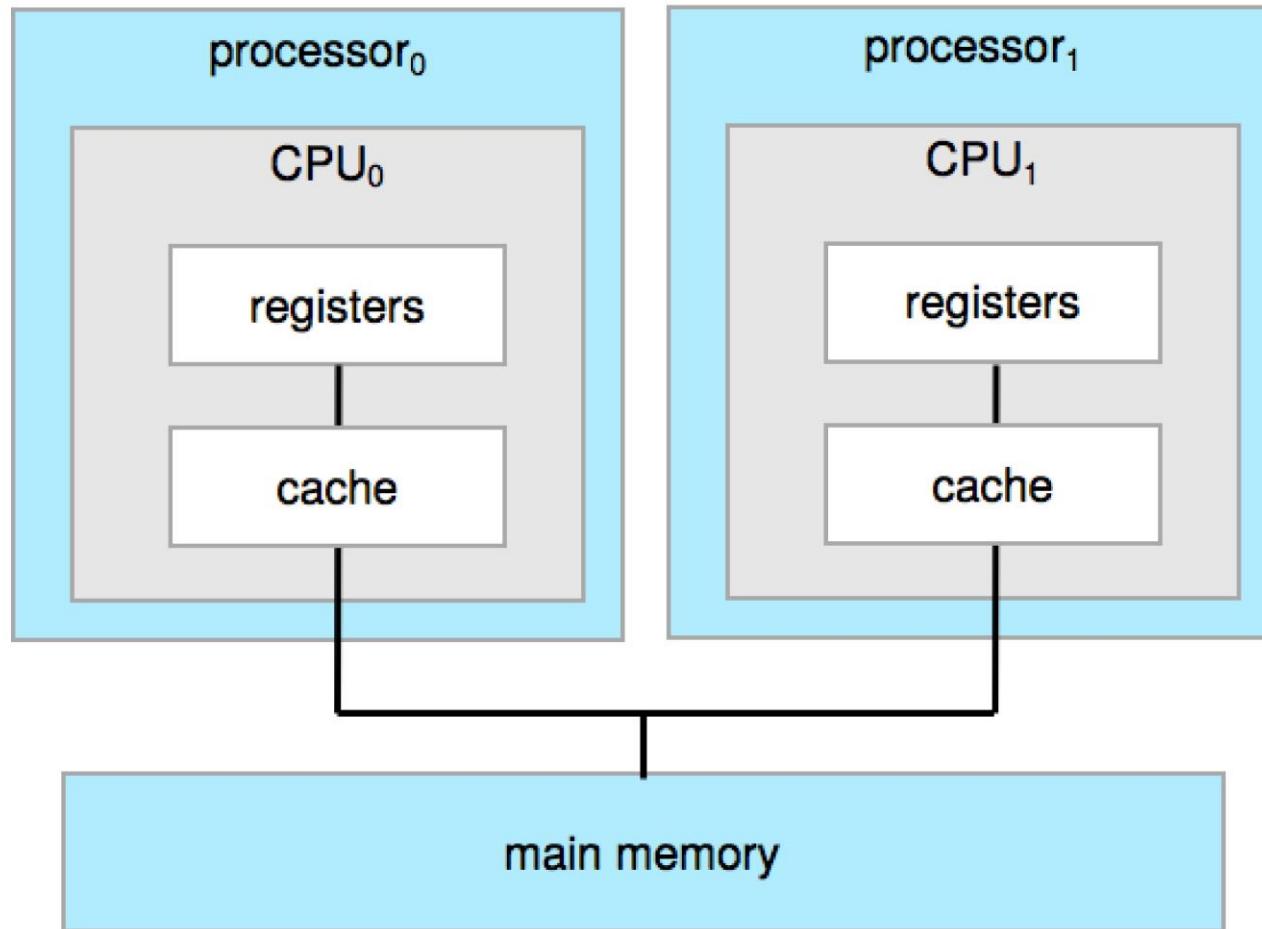
---

- 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





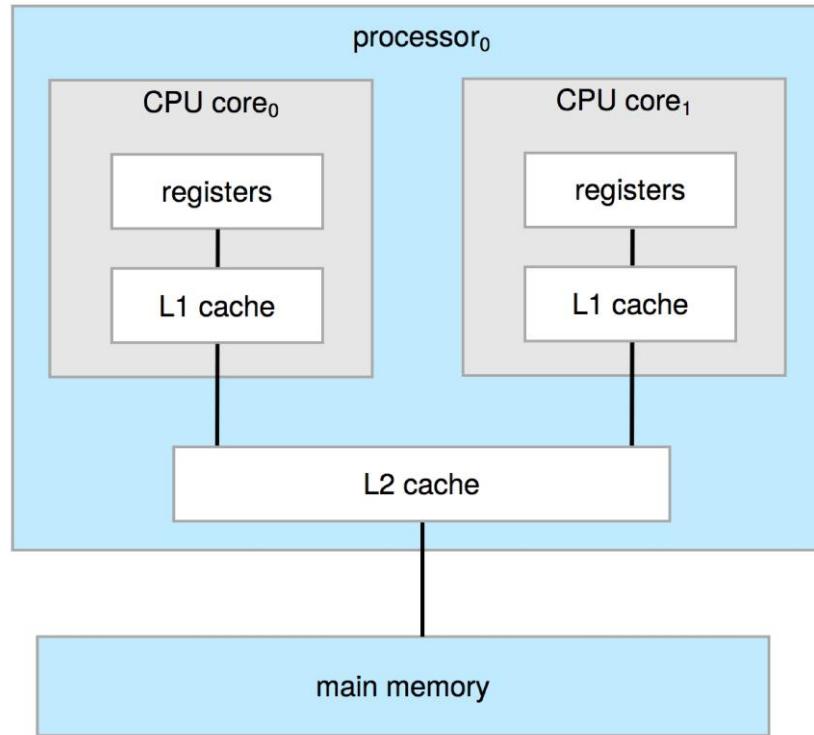
# Symmetric Multiprocessing Architecture





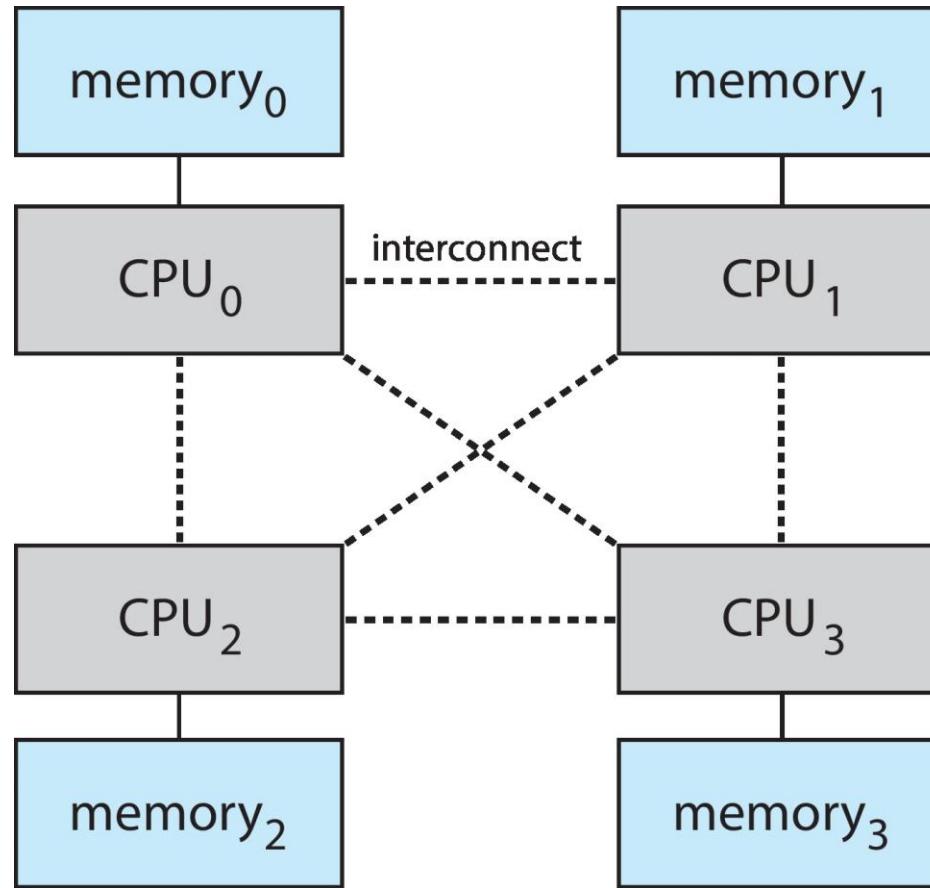
# Dual-Core Design

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





# Non-Uniform Memory Access System



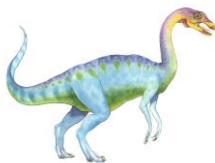


# Clustered Systems

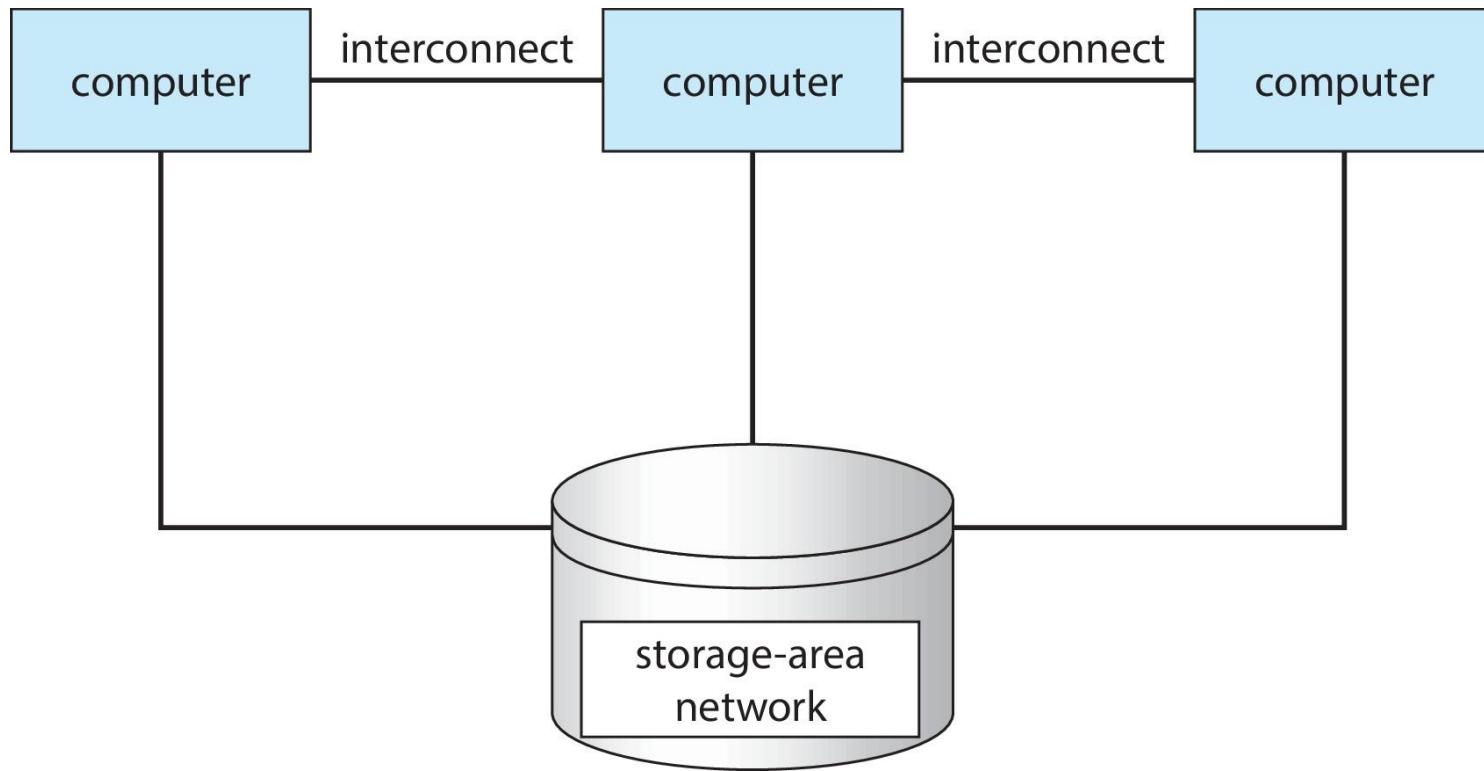
---

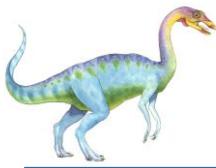
- Like multiprocessor systems, but multiple systems working together
  - Usually sharing storage via a **storage-area network (SAN)**
  - Provides a **high-availability** service which survives failures
    - ▶ **Asymmetric clustering** has one machine in hot-standby mode
    - ▶ **Symmetric clustering** 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





---

# Computer System Environments





# Computing Environments

---

- Traditional
- Mobile
- Client-Server
- Peer-to-Peer
- Cloud computing
- Real-time Embedded



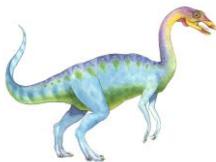


# Traditional

---

- 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





# Mobile

---

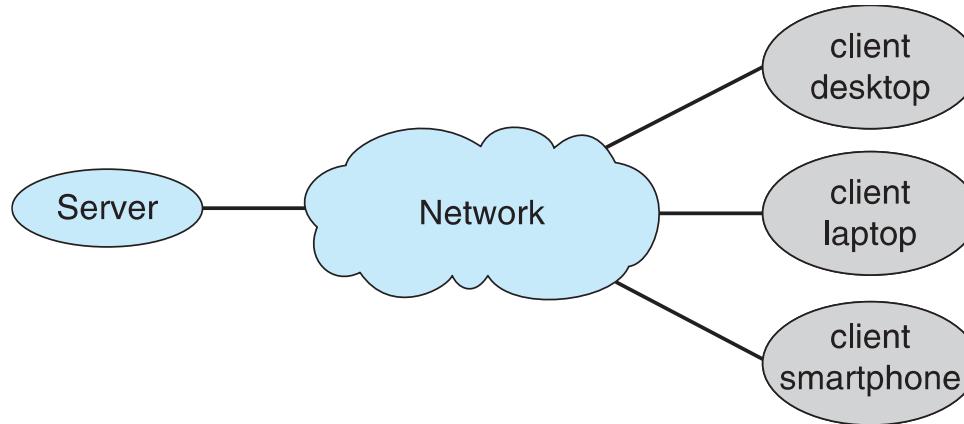
- Handheld smartphones, tablets, etc.
- What is the functional difference between them and a “traditional” laptop?
- Extra feature – more OS features (GPS, gyroscope)
- Allows new types of apps like ***augmented reality***
- Use IEEE 802.11 wireless, or cellular data networks for connectivity
- Leaders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





# Client Server

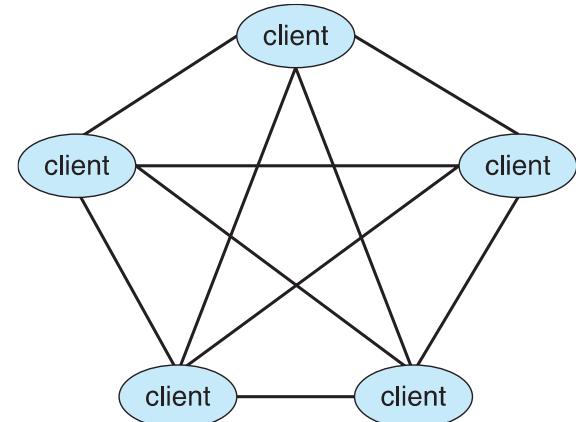
- Client-Server Computing
  - Dumb terminals supplanted by smart PCs
  - Many systems now **servers**, responding to requests generated by **clients**
    - ▶ **Compute-server system** provides an interface to client to request services (i.e., database)
    - ▶ **File-server system** provides interface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

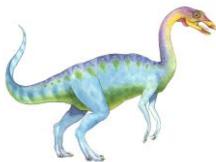




# Peer-to-Peer

- Another model of distributed system
- P2P does not distinguish clients and servers
  - All nodes are considered peers
  - Each acts 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





# 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





# Cloud Computing (Cont.)

---

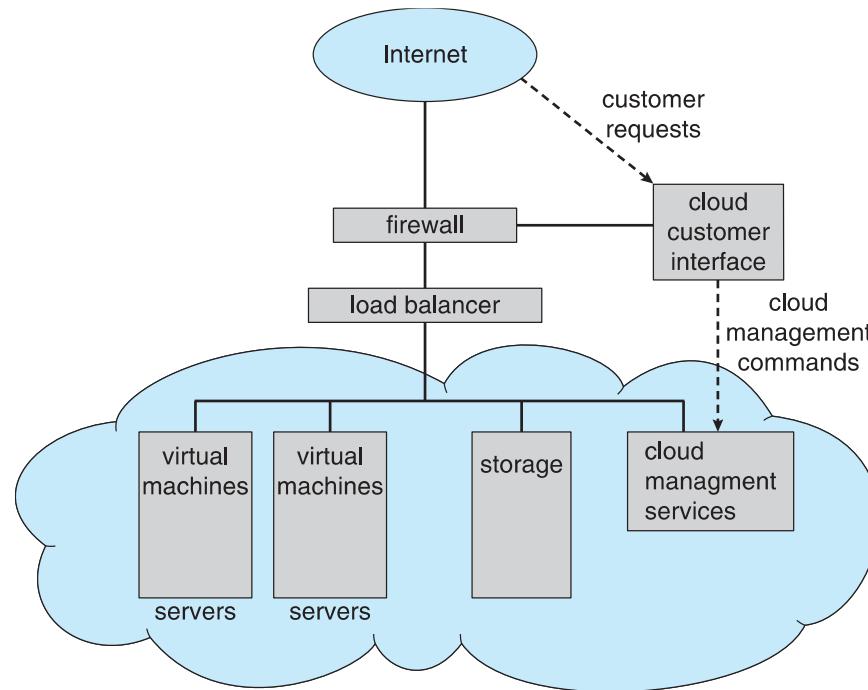
- Many types
  - **Public cloud** – available via Internet to anyone willing to pay
  - **Private cloud** – run by a company for the company's own use
  - **Hybrid cloud** – includes both public and private cloud components
  - Software as a Service (**SaaS**) – one or more applications available via 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)





# Cloud Computing (cont.)

- 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





# Real-Time Embedded Systems

---

- Real-time embedded systems most prevalent form of computers
  - Vary considerably, 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/>





# Free and Open-Source Operating Systems

---

- 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>)
- Used to run guest OS 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, BSD 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.

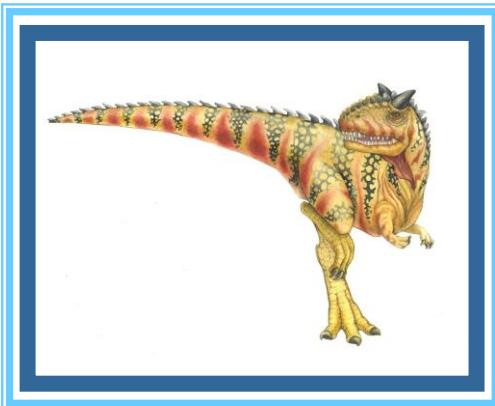
Operating systems that are no longer commercially viable have been open-sourced as well, enabling us to study how systems operated in a time of fewer CPU, memory, and storage resources. An extensive but incomplete list of open-source operating-system projects is available from [https://curlie.org/Computers/Software/Operating\\_Systems/Open\\_Source/](https://cur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

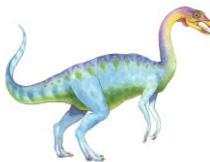
In addition, the rise of virtualization as a mainstream (and frequently free) computer function makes it possible to run many operating systems on top of one core system. For example, VMware (<http://www.vmware.com>) provides a free “player” for Windows on which hundreds of free “virtual appliances” 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.



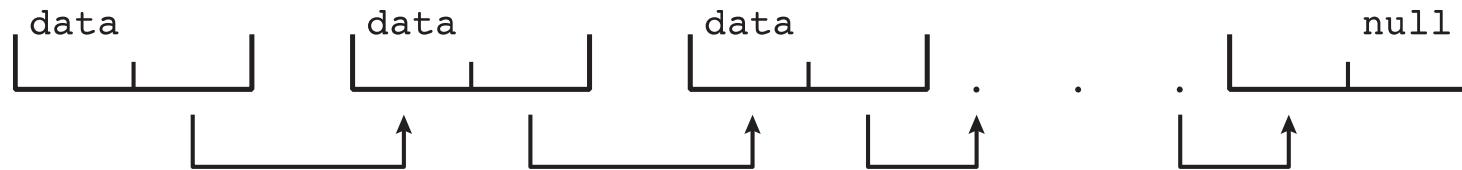
# End of Chapter 1



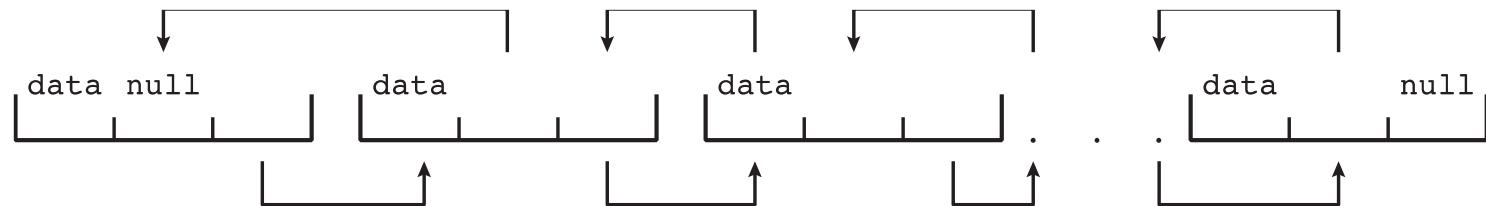


# Kernel Data Structures

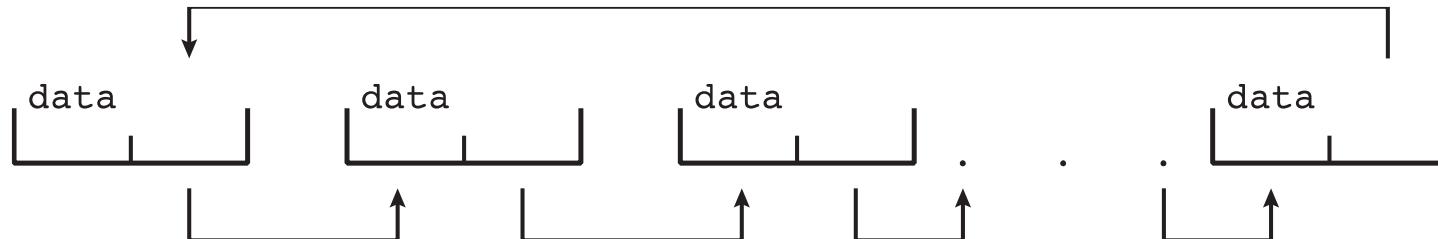
- Many similar to standard programming data structures
- ***Singly linked list***



- ***Doubly linked list***



- ***Circular linked list***



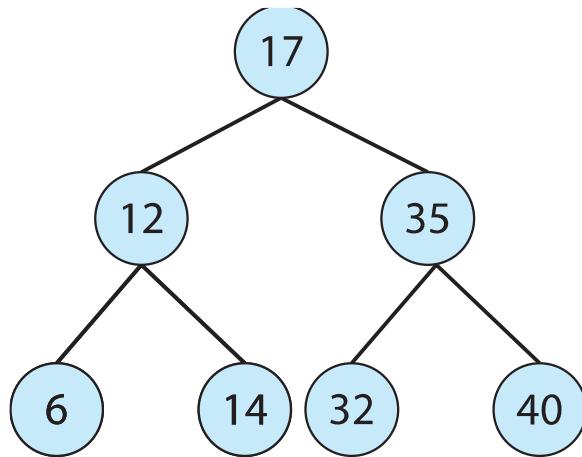


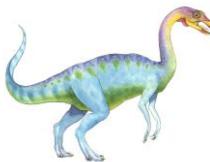
# Kernel Data Structures

- **Binary search tree**

left  $\leq$  right

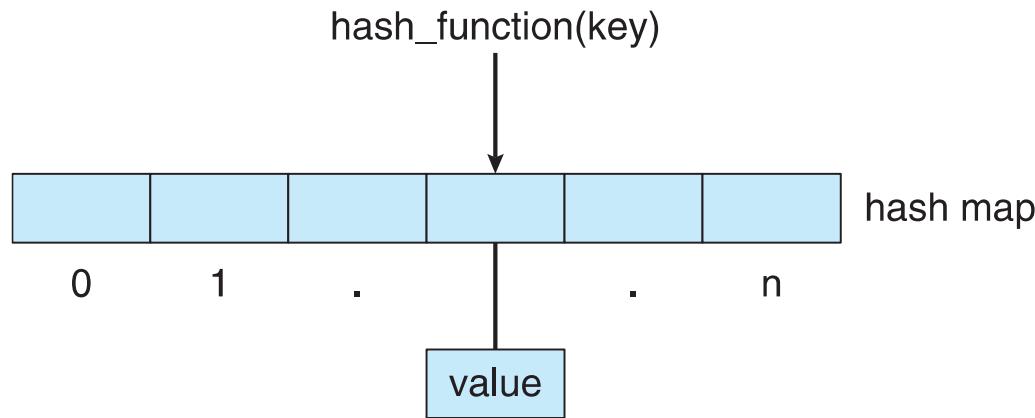
- Search performance is  $O(n)$
- **Balanced binary search tree** is  $O(\lg n)$





# 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>`





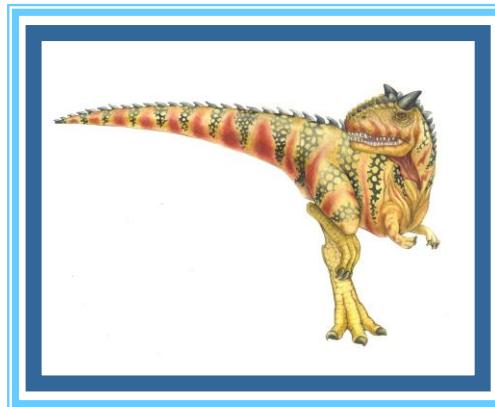
# Characteristics of Various Types of Storage

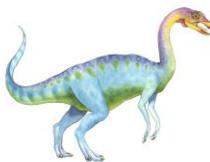
Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	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



# Chapter 2: Operating-System Services



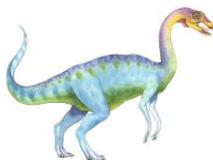


# Outline

---

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



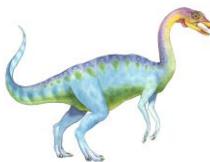


# Objectives

---

- Identify services provided by an OS
- Illustrate how **system calls** are used to provide OS services
- Compare and contrast monolithic, layered, microkernel, modular, and hybrid **strategies for designing OS**
- Illustrate the process for booting an OS
- Apply tools for monitoring OS performance



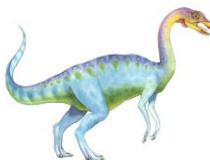


# Operating System Services

---

- OSes provide an environment for execution of programs and services to programs and users
- One set of OS services provides functions that are helpful to the user:
  - **User interface** - Almost all OSes 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)



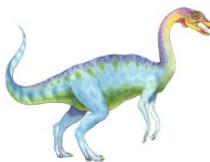


# Operating System Services

---

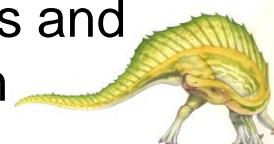
- One set of OS services provides functions that are helpful to the user (cont.):
  - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
  - **File-system manipulation** - The file system is of particular interest
    - ▶ Programs need to read and write files and directories, create and delete them, search them, list file information, permission management

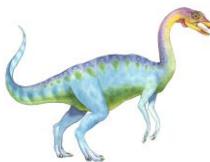




# Operating System Services (Cont.)

- One set of OS services provides functions that are helpful to the user (Cont.):
  - **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





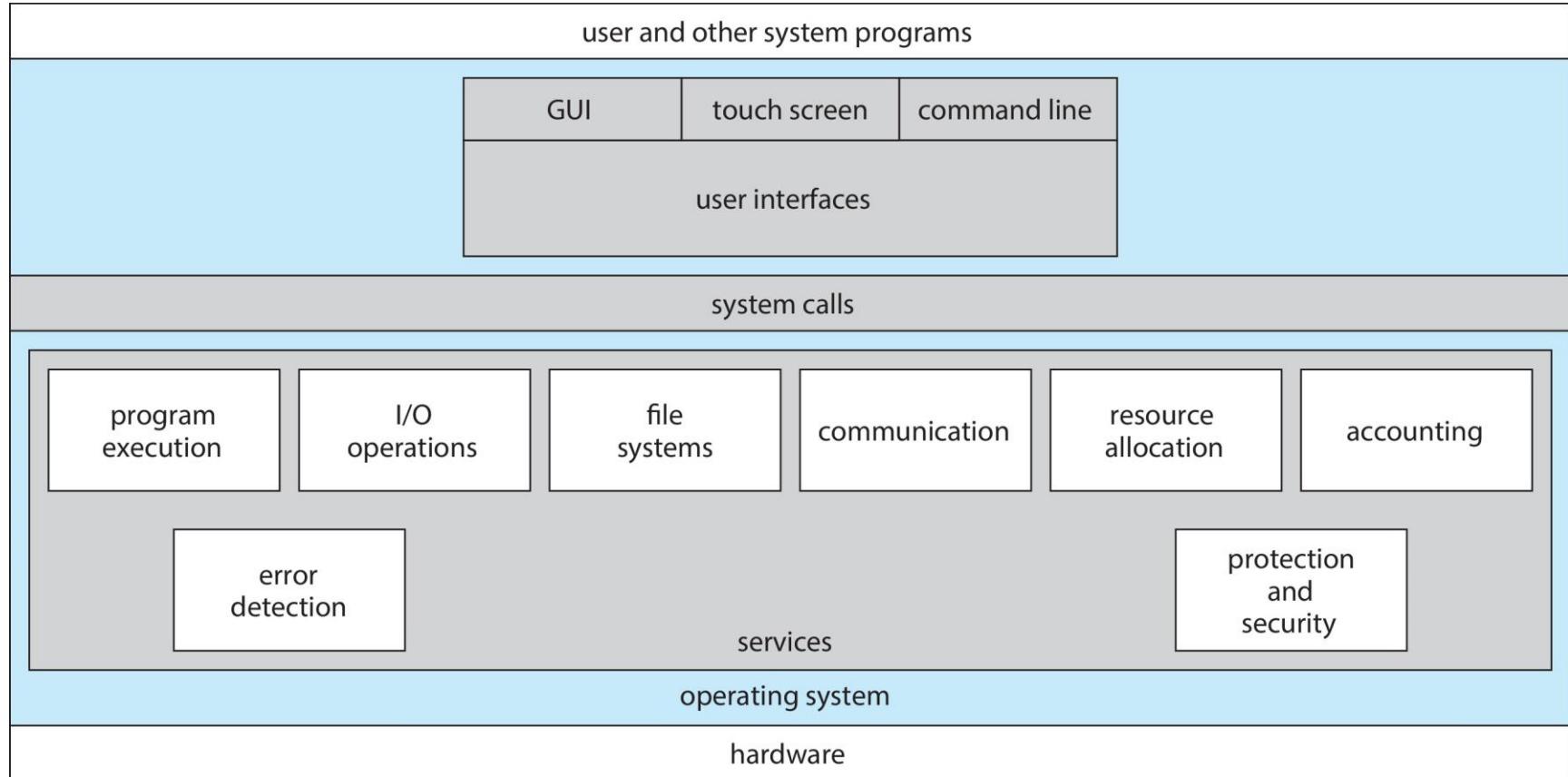
# Operating System Services (Cont.)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **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 multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - ▶ **Protection** involves ensuring that all access to system resources is controlled
    - ▶ **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts





# A View of Operating System Services

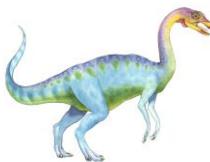




# Command Line interpreter

- CLI allows direct command entry
- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – **shells**
- 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

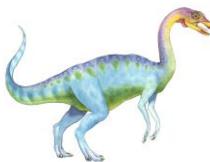




# Bourne Shell Command Interpreter

```
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:~ pbg$ 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

---

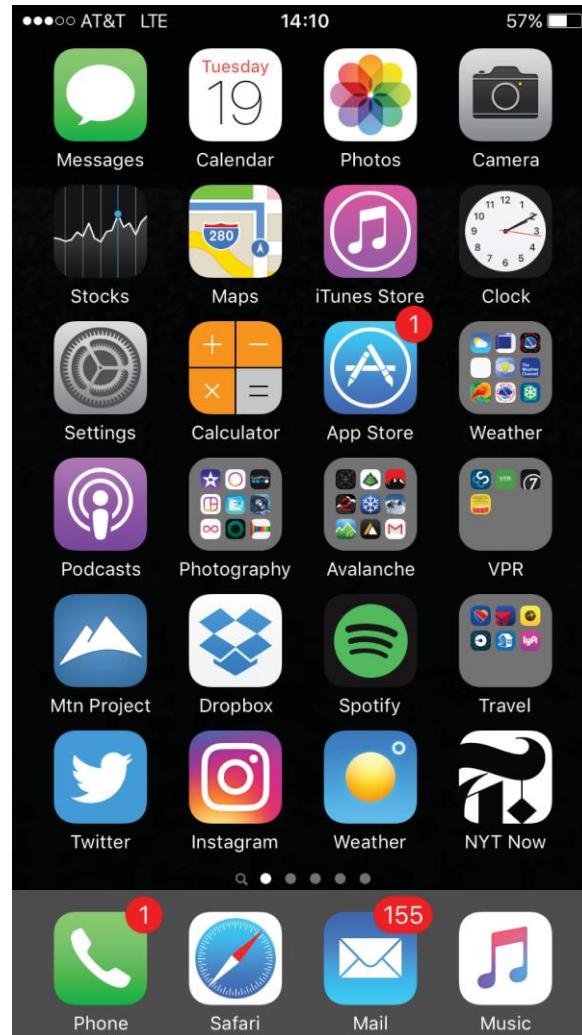
- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)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

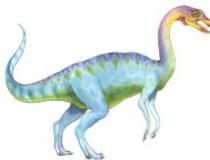




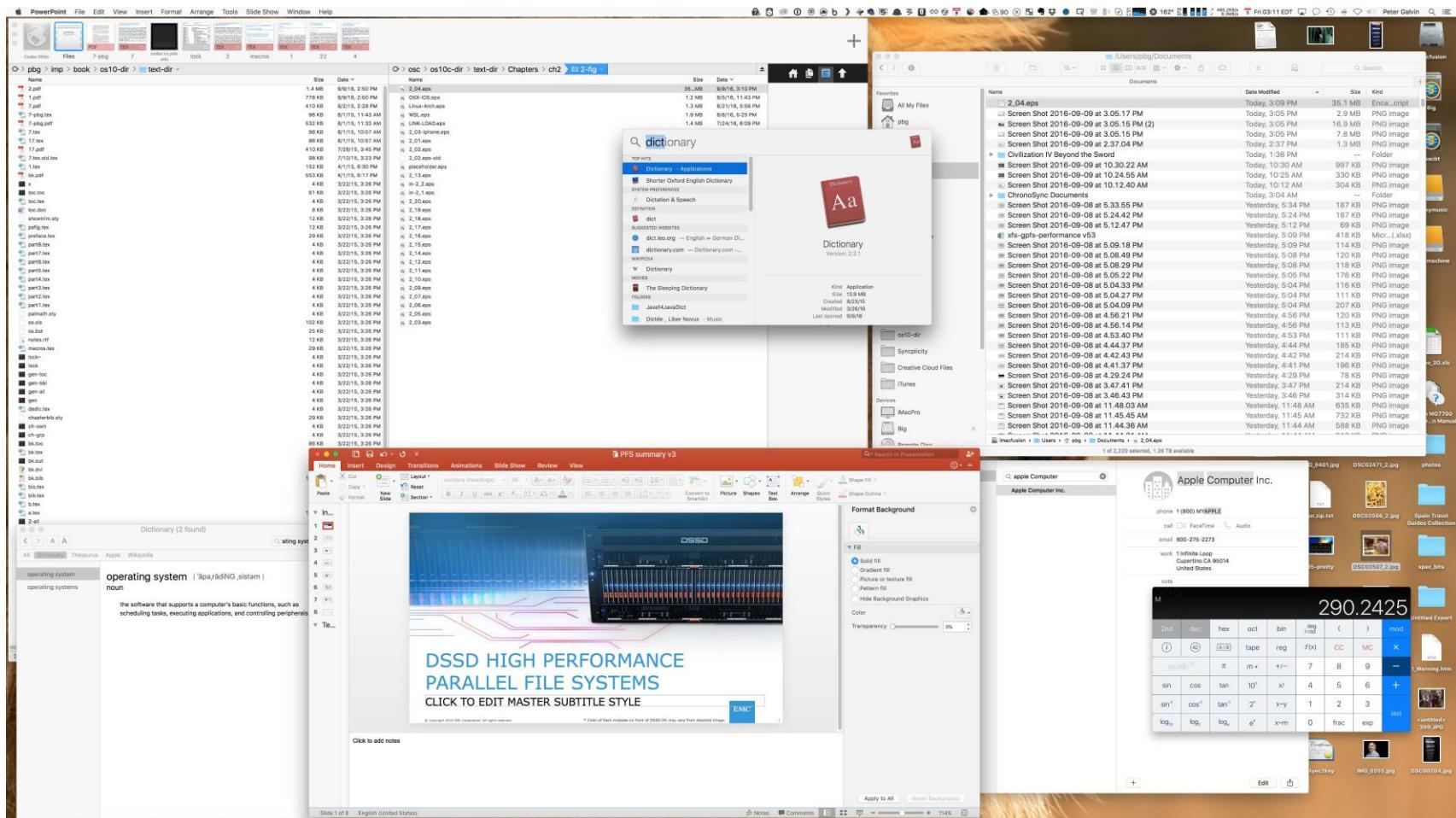
# Touchscreen Interfaces

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





# The Mac OS X GUI





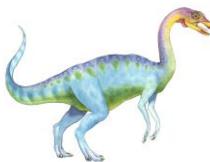
# System Calls

---

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call
- Three most common APIs:
  - 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

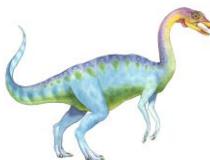
source file

destination file

## Example System Call Sequence

```
Acquire input file name
Write prompt to screen
Accept input
Acquire output file name
Write prompt to screen
Accept input
Open the input file
if file doesn't exist, abort
Create output file
if file exists, abort
Loop
Read from input file
Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```





# 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)
```

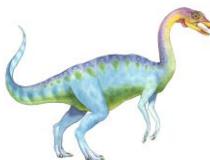
return function parameters  
value name

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.

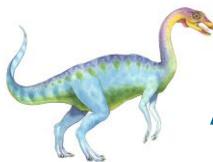




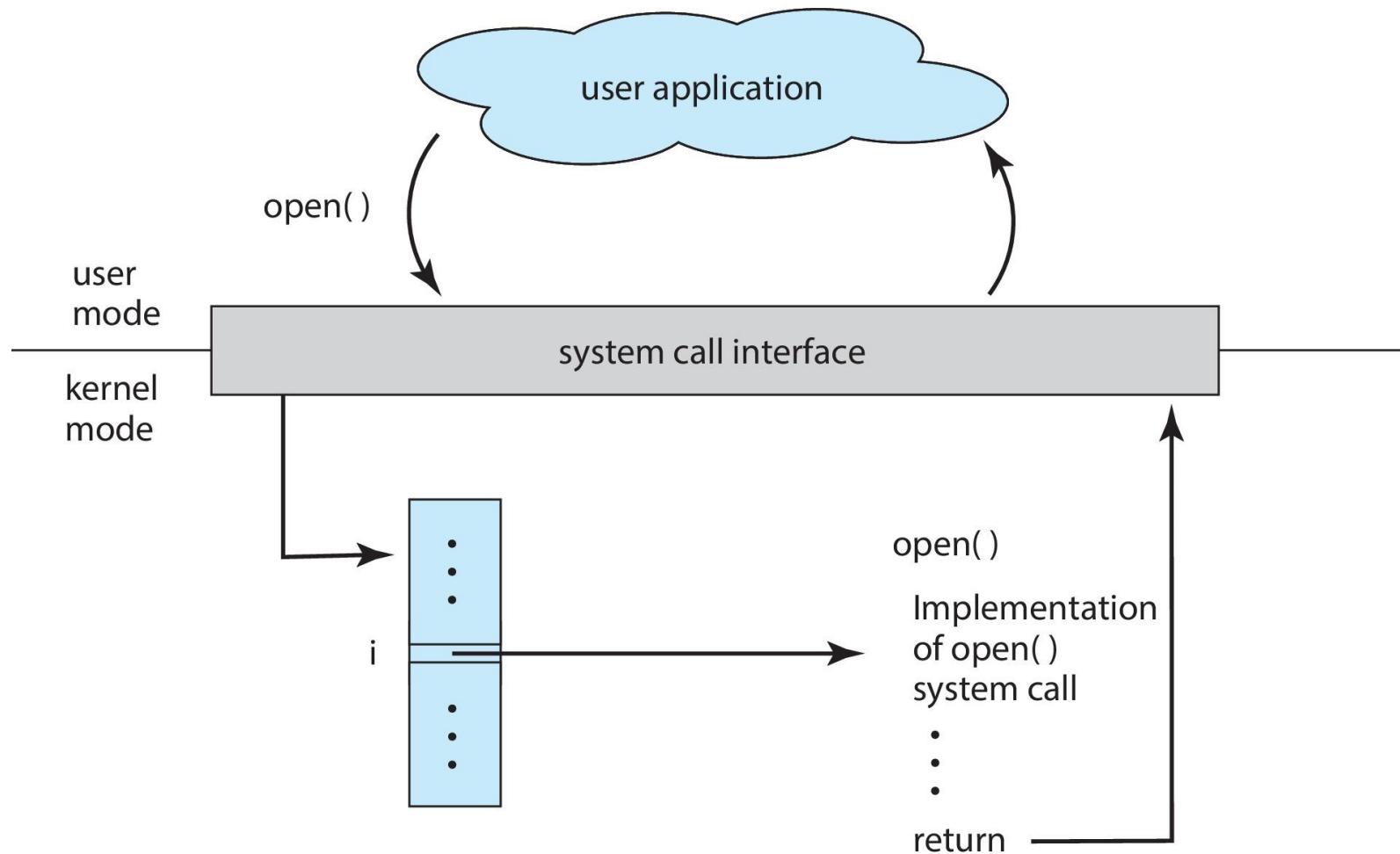
# System Call Implementation

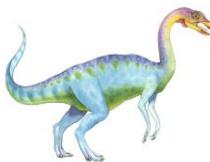
- Typically, a number is associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller needs to know 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





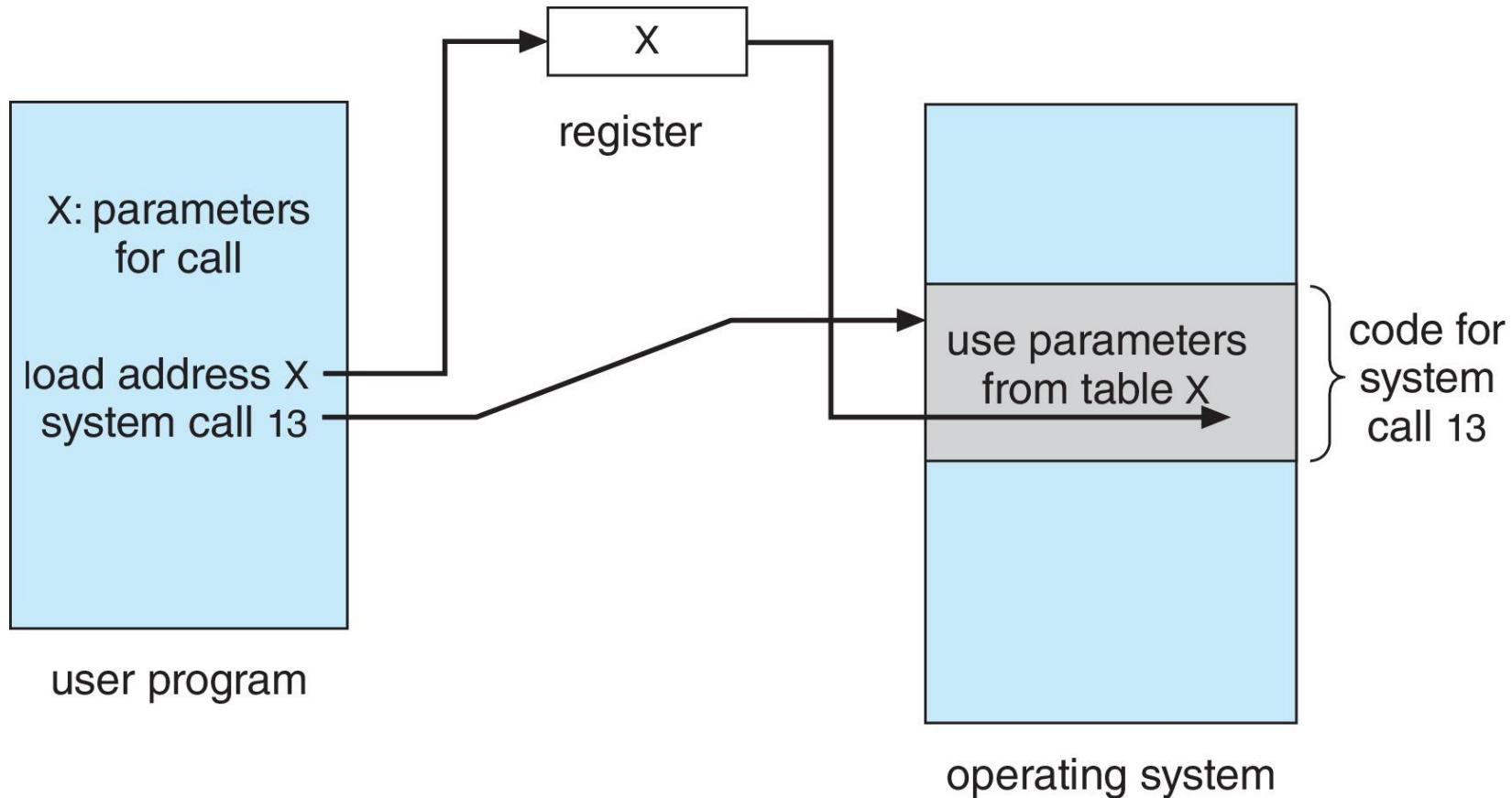
# System Call Parameter Passing

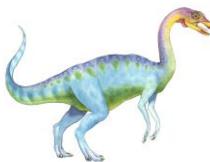
- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - ▶ In some cases, may be more parameters 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 OS
  - Block and stack methods do not limit the number or length of parameters being passed





# Parameter Passing via Table



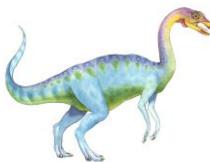


# Types of System Calls

---

- 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



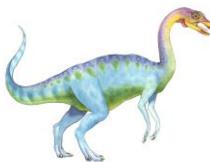


# Types of System Calls (Cont.)

---

- 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



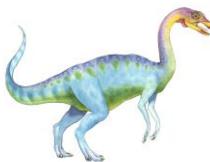


# Types of System Calls (Cont.)

---

- 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





# Types of System Calls (Cont.)

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





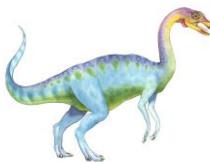
# Examples of Windows and Unix System Calls

## EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

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

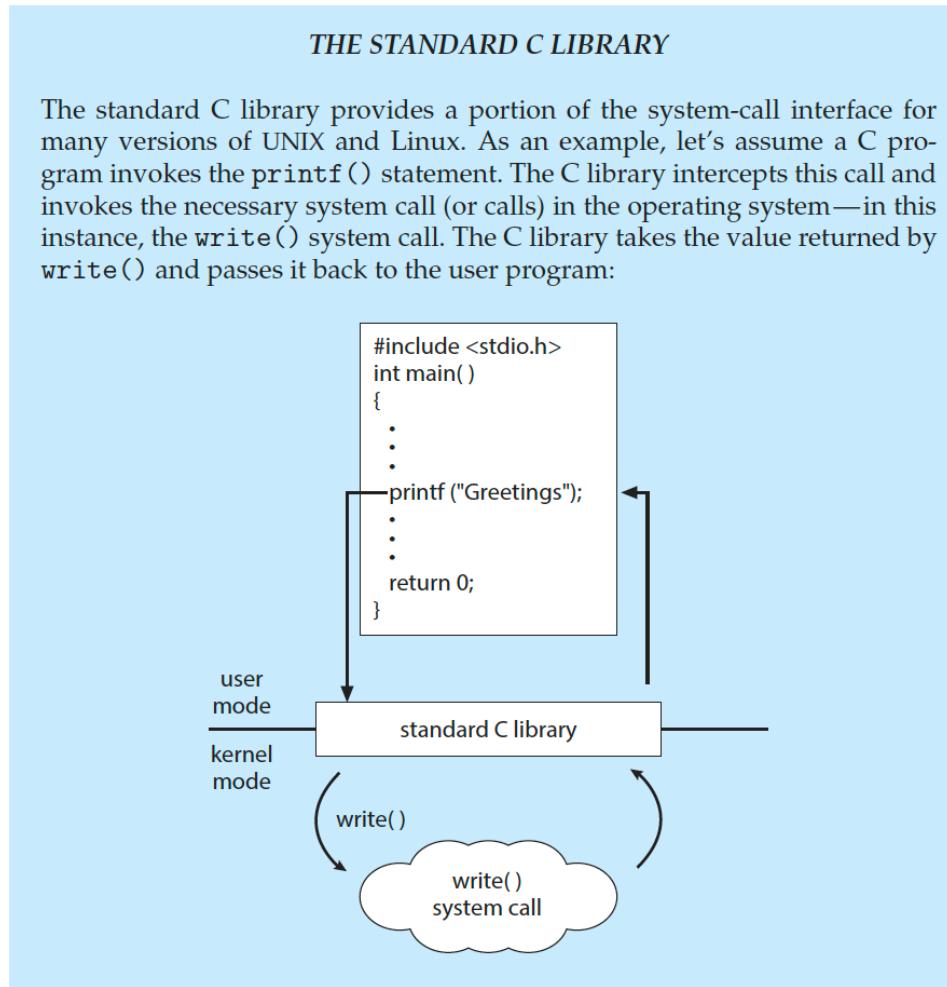
	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance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

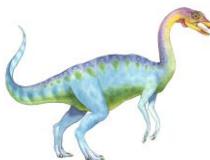




# Standard C Library Example

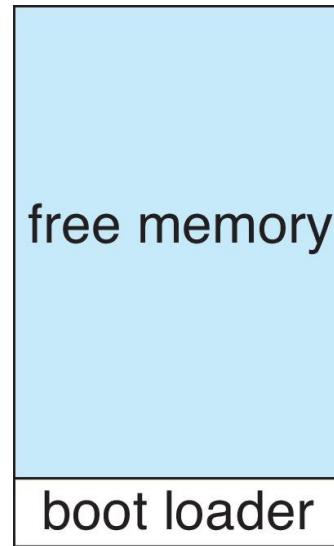
- C program invoking printf() library call, which calls write() system call





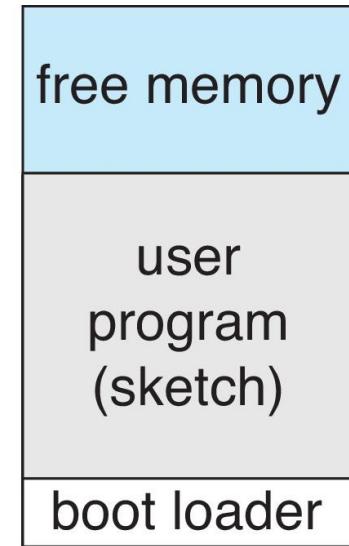
# Example: Arduino

- 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)

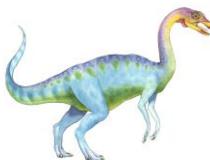
At system startup



(b)

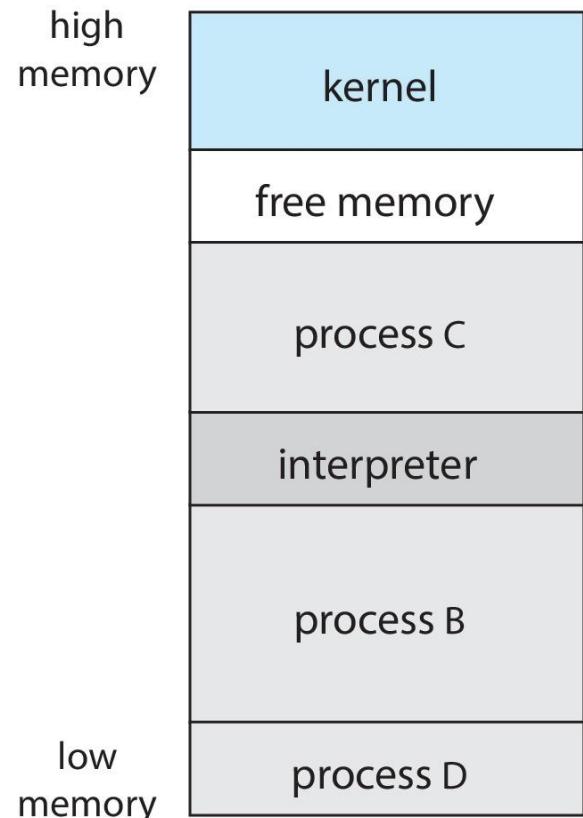
running a program

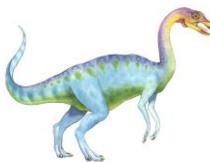




# Example: FreeBSD

- 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



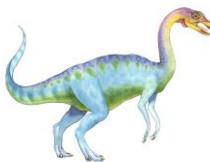


# System Services

---

- **System programs** provide a convenient environment for program development and execution
  - 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

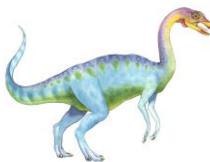




# System Services (Cont.)

- 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

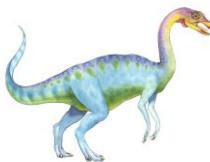




# System Services (Cont.)

- **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

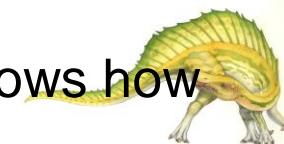




# Linkers and Loaders

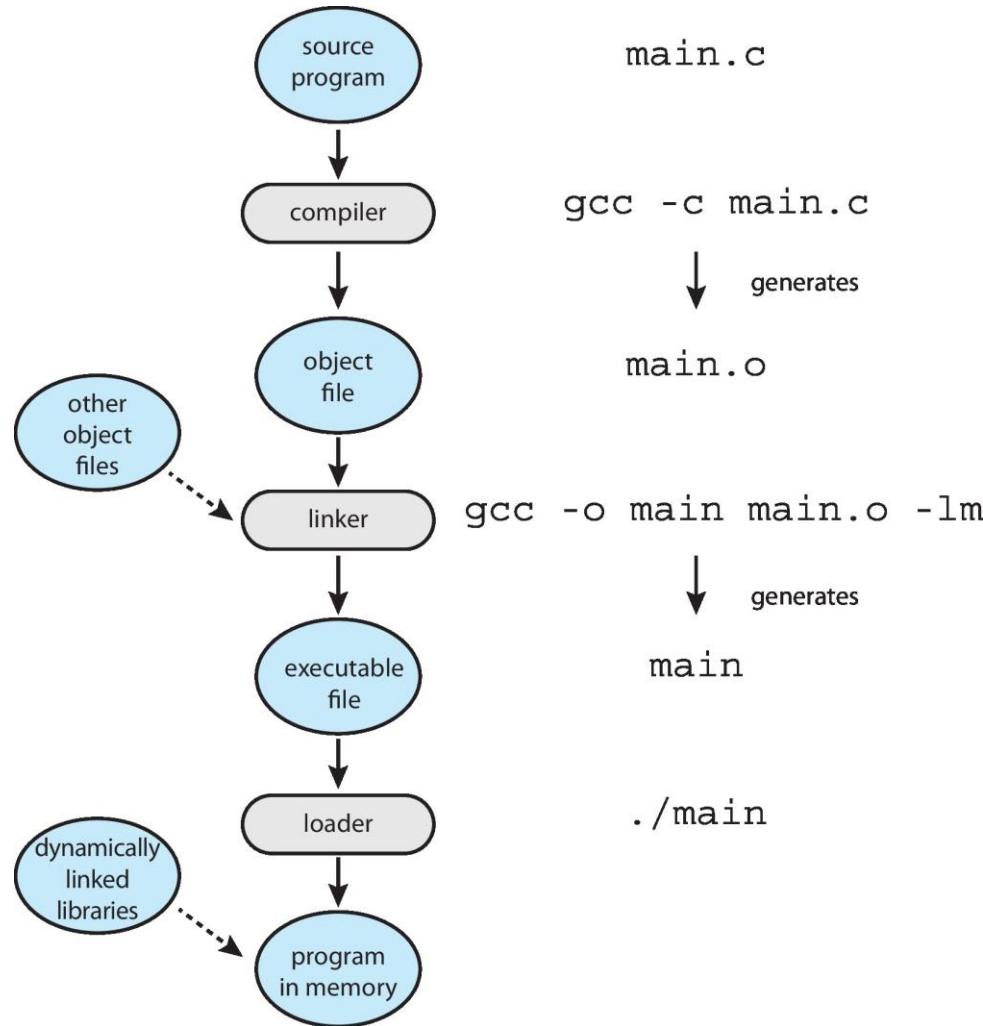
---

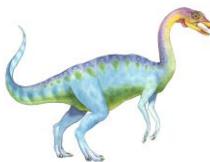
- Source code compiled into object files designed to be loaded into any physical memory location – **relocatable object file**
- **Linker** combines these into single binary **executable** file
  - Also brings in libraries
- Program resides on secondary storage as binary executable
- Must be brought 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 OS knows how to load and start them





# The Role of the Linker and Loader

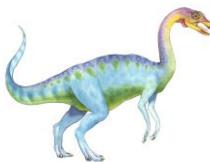




# Why Applications are Operating System Specific

- Apps compiled on one system usually not executable on other OS
- Each OS 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 OS
  - App written in language that includes a VM containing the running app (like Java)
  - Use standard language (like C), compile separately on each OS 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 OS on a given architecture, CPU, etc.



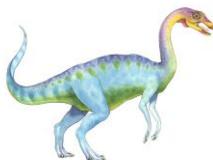


# Design and Implementation

---

- Design and Implementation of OS is not “solvable”, but some approaches have proven successful
- Internal structure of different OS 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 – OS should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – OS should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- Specifying and designing an OS is highly creative task of **software engineering**



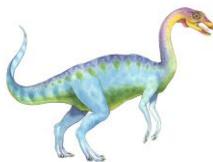


# Policy and Mechanism

---

- **Policy:** What needs to be done?
  - Example: Interrupt after every 100 seconds
- **Mechanism:** How to do something?
  - Example: timer
- Important principle: separate policy from mechanism
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later
  - Example: change 100 to 200

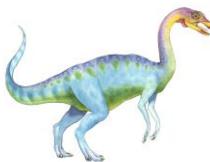




# Implementation

- 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 level 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

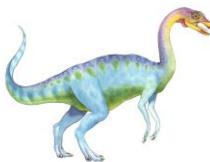




# Operating System Structure

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



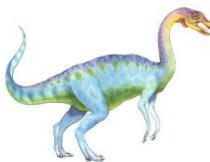


# Monolithic Structure – Original UNIX

---

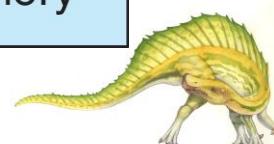
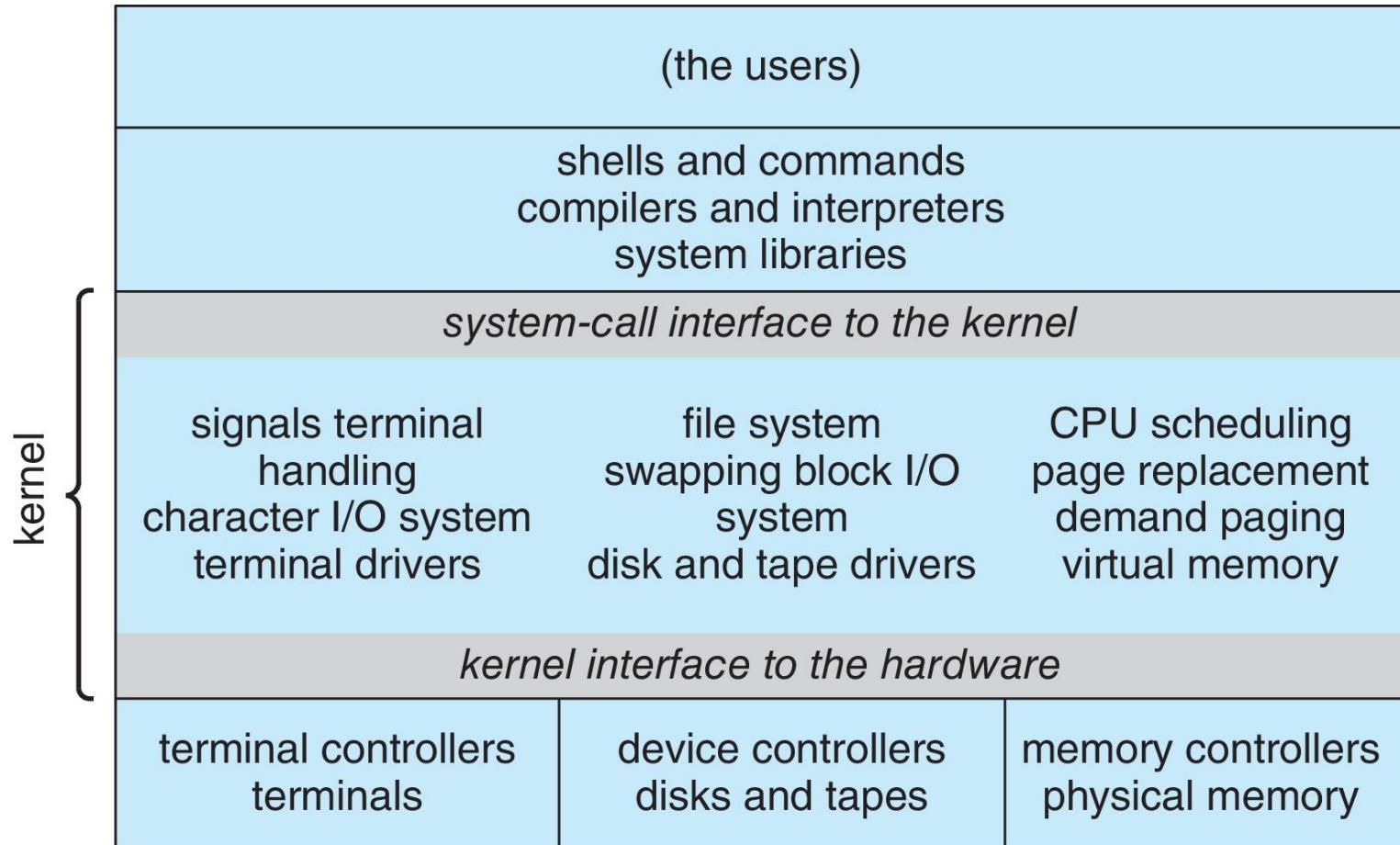
- UNIX – limited by hardware functionality, the original UNIX OS had limited structuring.
- The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - ▶ Consists of everything below the system-call interface and above the physical hardware
    - ▶ Provides the file system,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

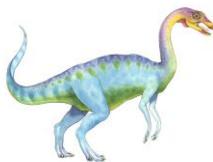




# Traditional UNIX System Structure

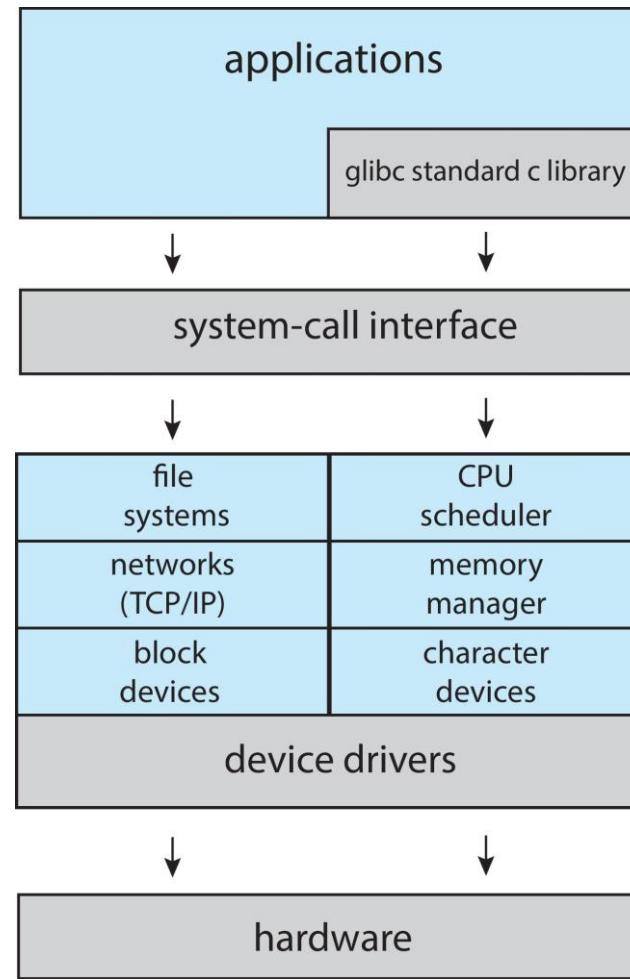
Beyond simple but not fully layered

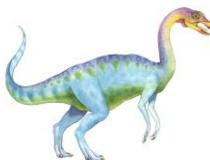




# Linux System Structure

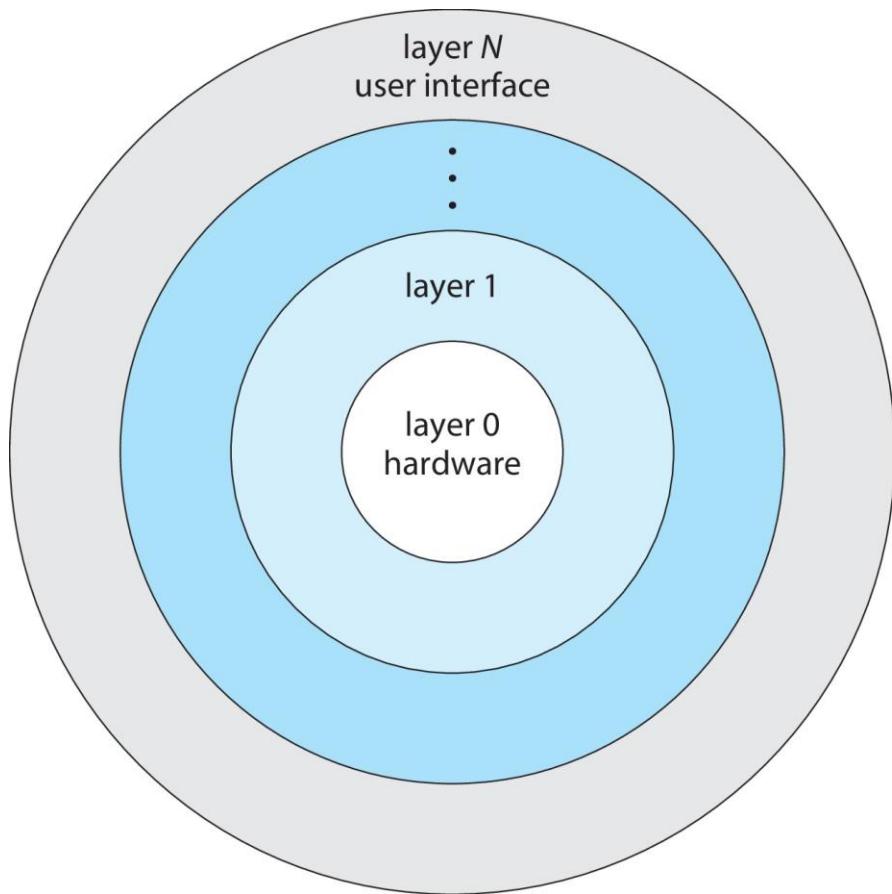
Monolithic plus modular design





# Layered Approach

- The OS 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





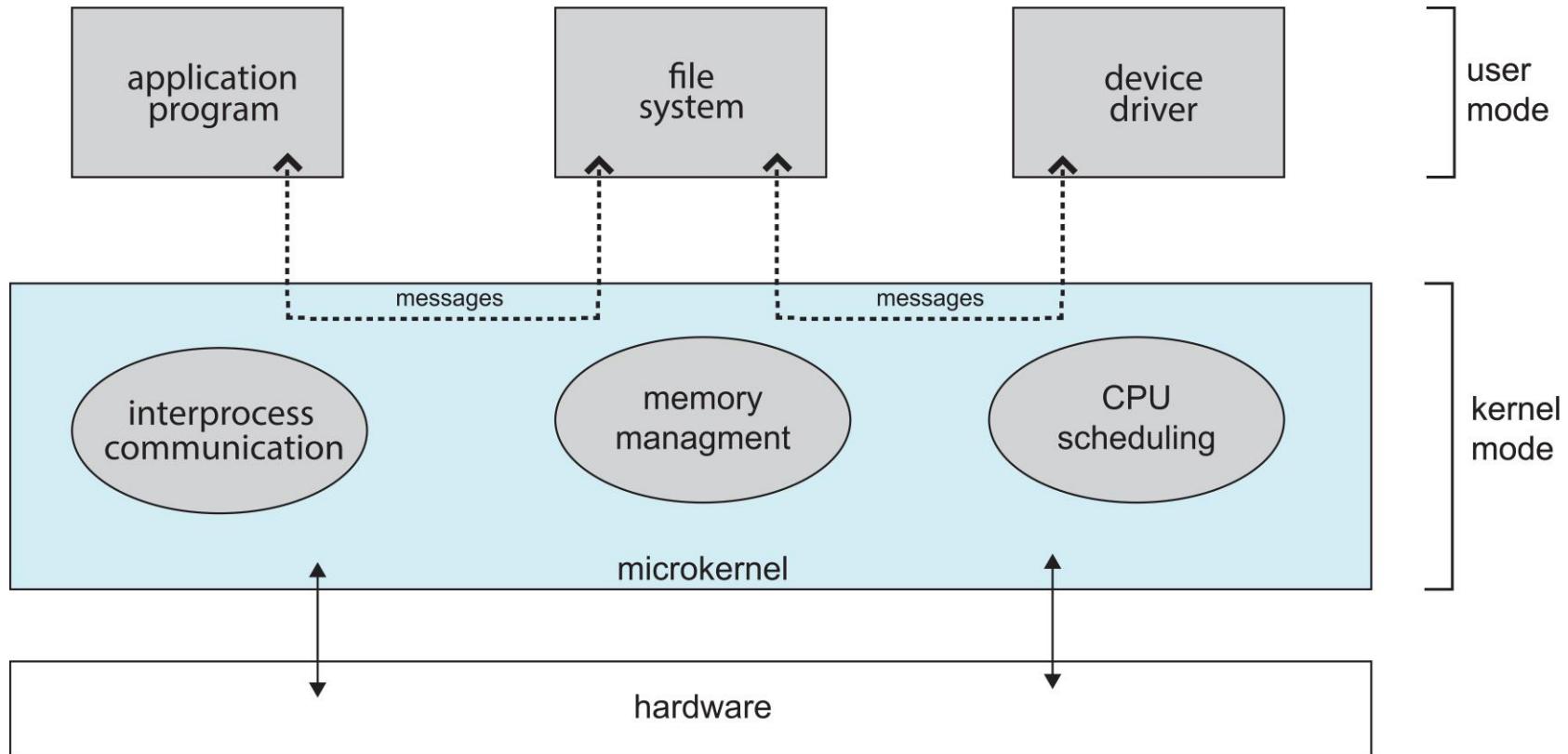
# Microkernels

- Moves as much from the kernel into user space
- **Mach** is an 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 OS to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication





# Microkernel System Structure

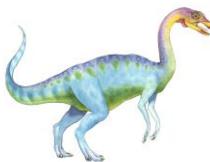




# Modules

- Many modern OS 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 more flexible
  - Linux, Solaris, etc.

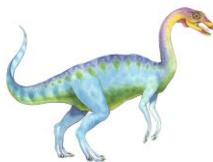




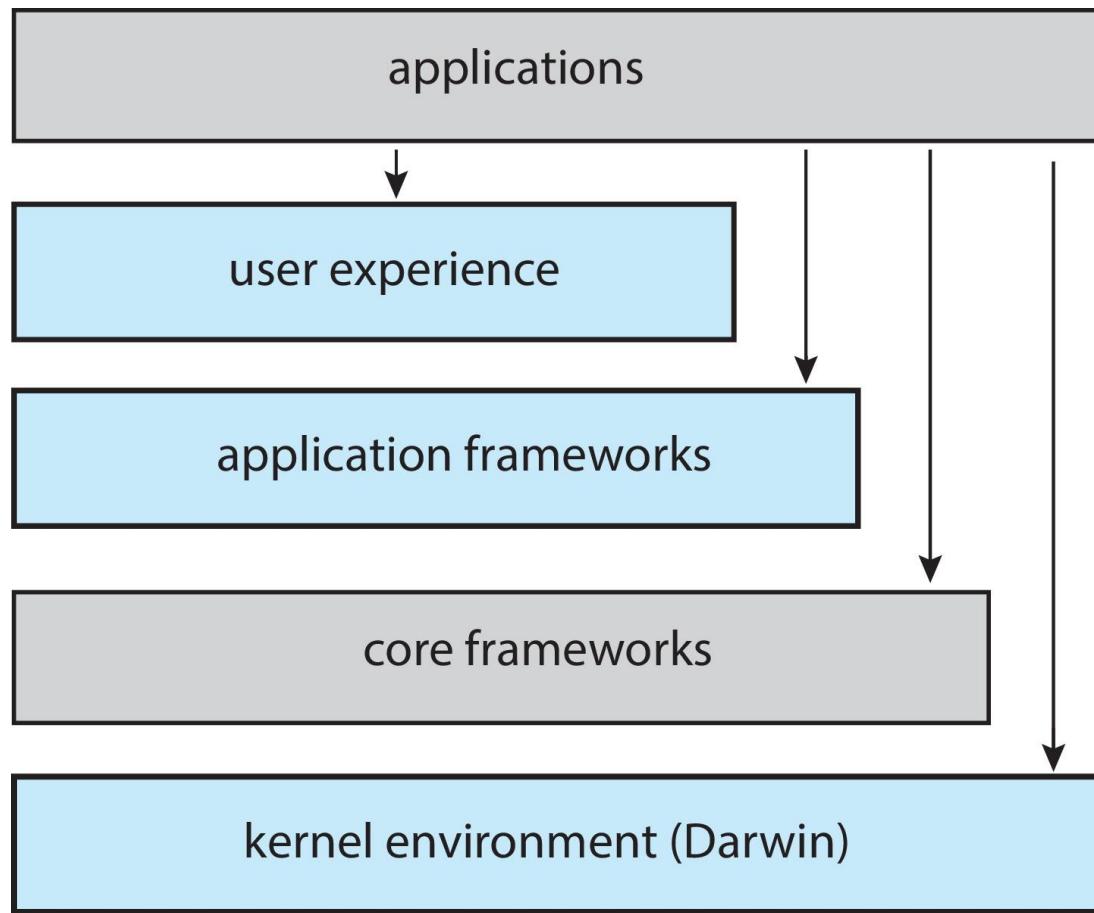
# Hybrid Systems

- Most modern OS are 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**)



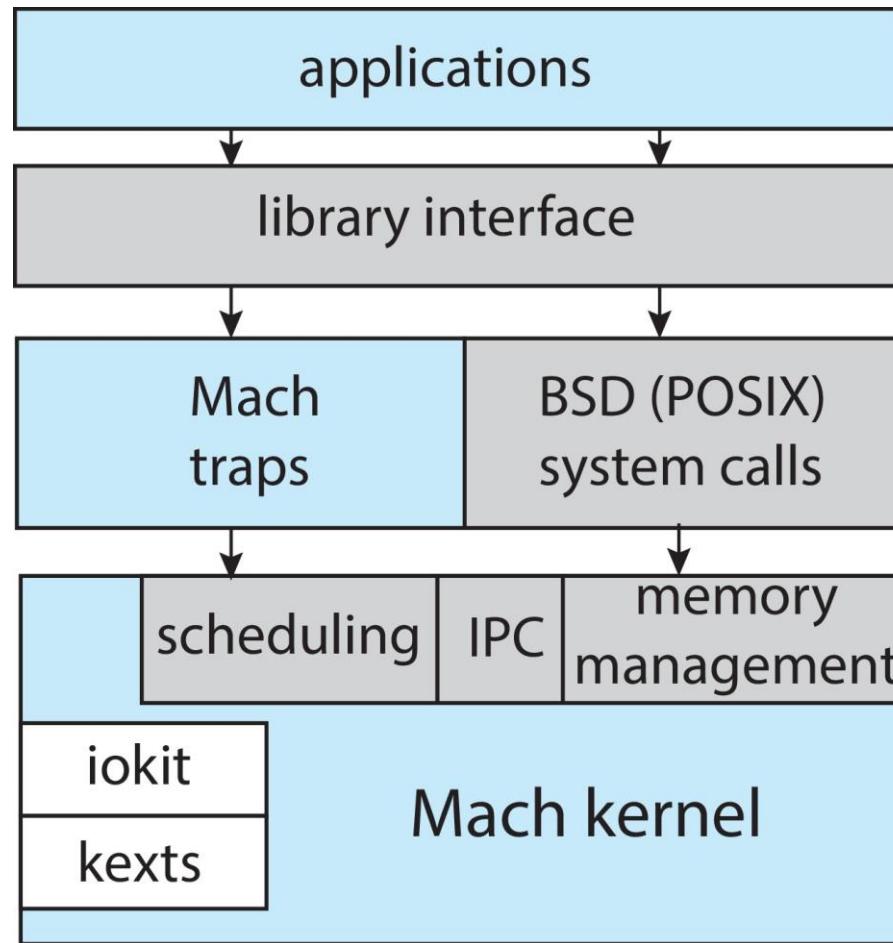


# macOS and iOS Structure





# Darwin



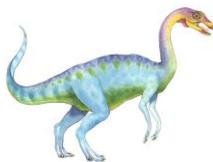


# Android

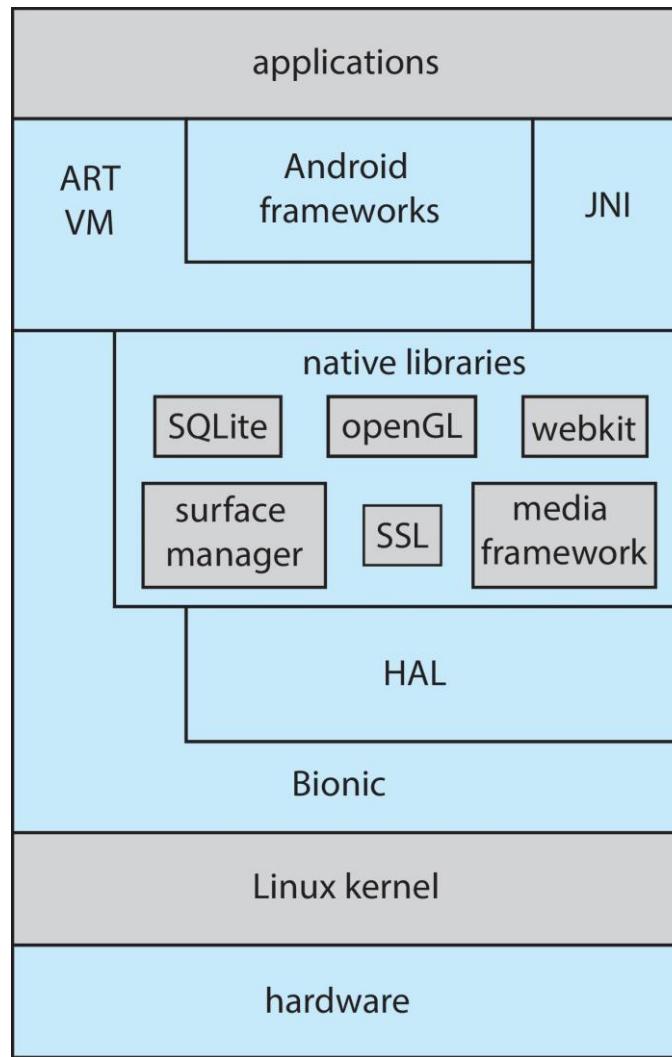
---

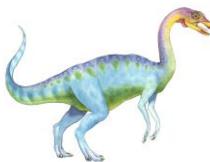
- Developed by Open Handset Alliance (mostly Google)
  - Open Source
- Similar stack to iOS
- Based on Linux kernel but modified
  - Provides process, memory, device-driver management
  - Adds power management
- Runtime environment includes core set of libraries and Dalvik virtual machine
  - Apps developed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



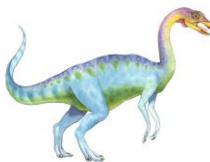


# Building and Booting an Operating System

---

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



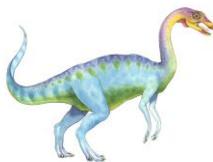


# Building and Booting Linux

---

- 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”



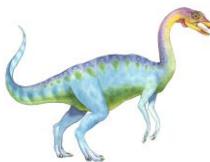


# System Boot

---

- When power initialized on system, execution starts at a fixed memory location
- OS 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



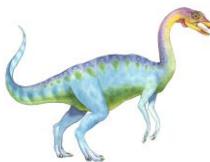


# Operating-System Debugging

- **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
- OS 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

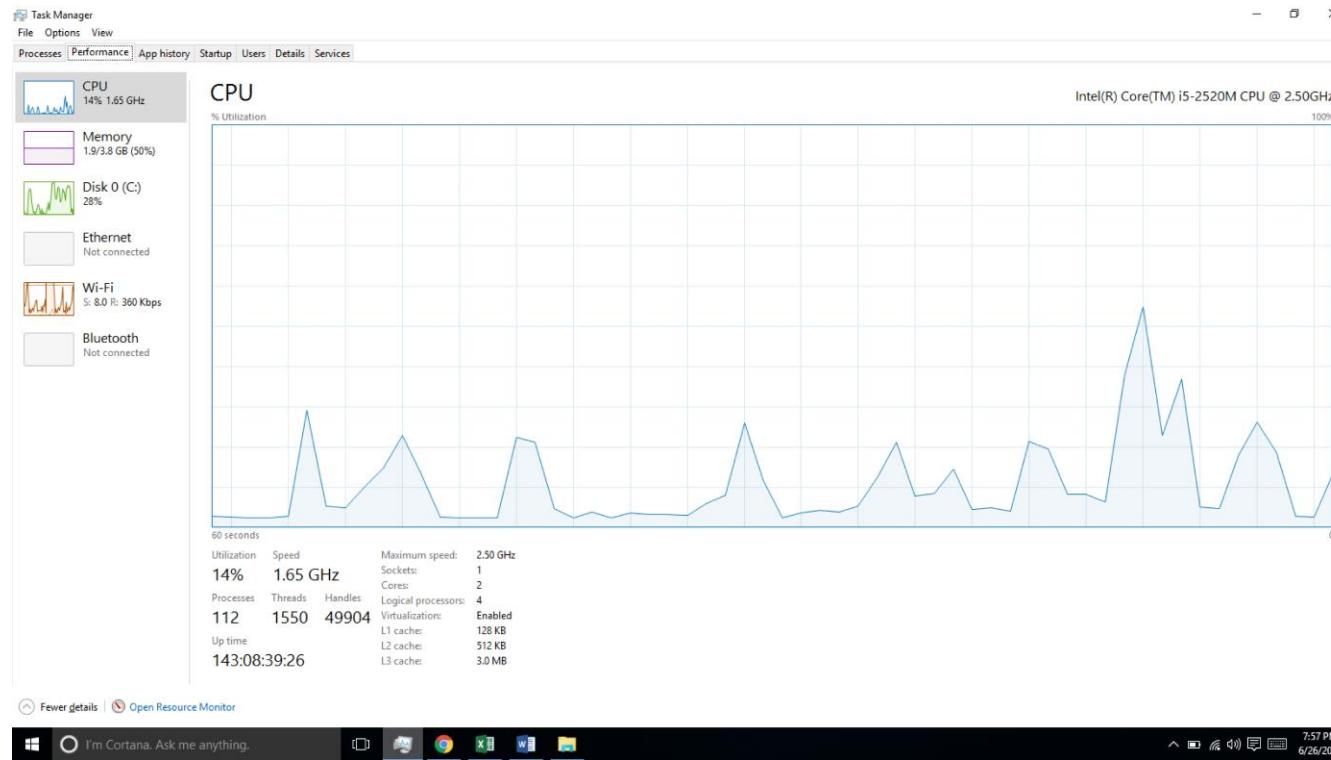
Kernighan's Law: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

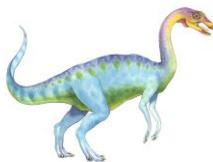




# Performance Tuning

- 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

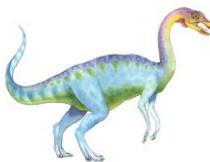




# Tracing

- 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





# BCC

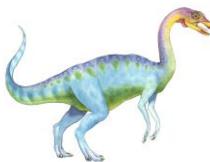
---

- Debugging interactions between user-level and kernel code nearly impossible without toolset that understands both and can instrument their actions
- BCC (BPF Compiler Collection) is a rich toolkit providing tracing features for Linux
  - See also the original DTrace
- For example, disksnoop.py 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

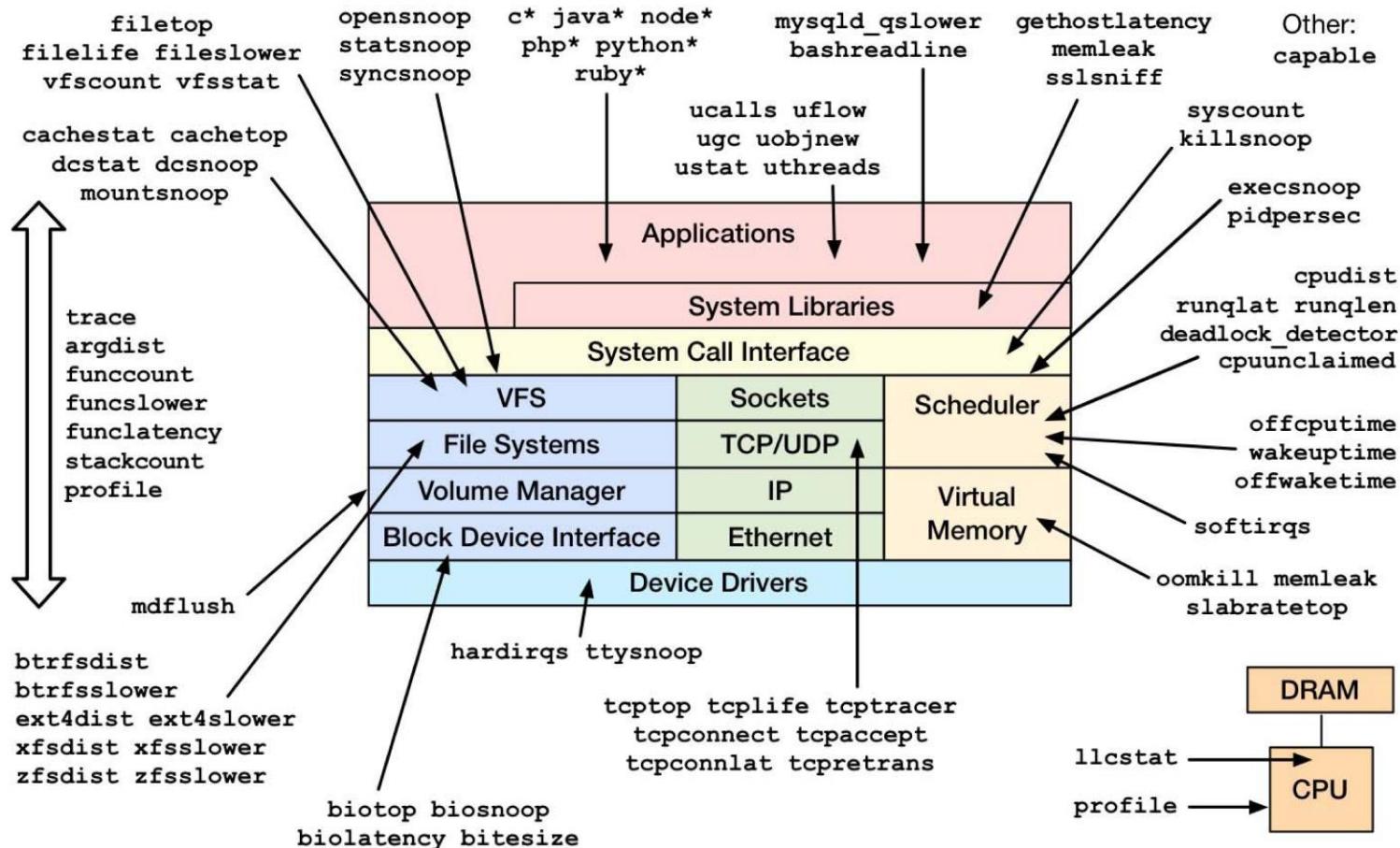
- Many other tools (next slide)





# Linux bcc/BPF Tracing Tools

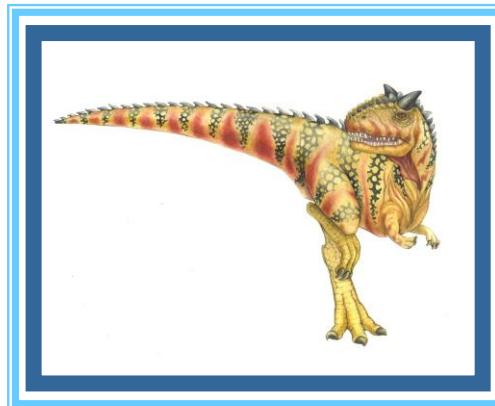
## Linux bcc/BPF Tracing Tools



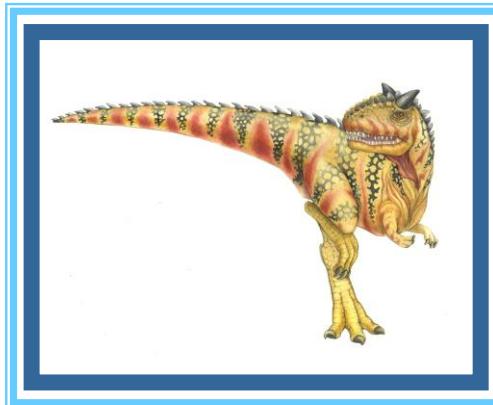
<https://github.com/iovisor/bcc#tools 2017>

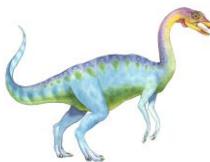


# End of Chapter 2



# Chapter 3: Processes



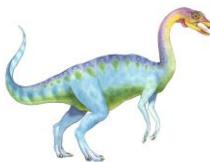


# Outline

---

- 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



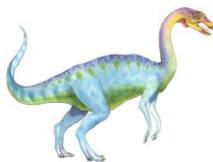


# Objectives

---

- Identify the separate components of a process and illustrate how they are **represented** and **scheduled** in an OS
- 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 use 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 OS





# Process Concept

---

- An OS executes a variety of programs that run as a process
- **Process** – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process
- 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

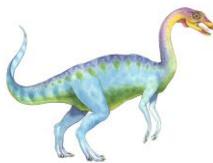




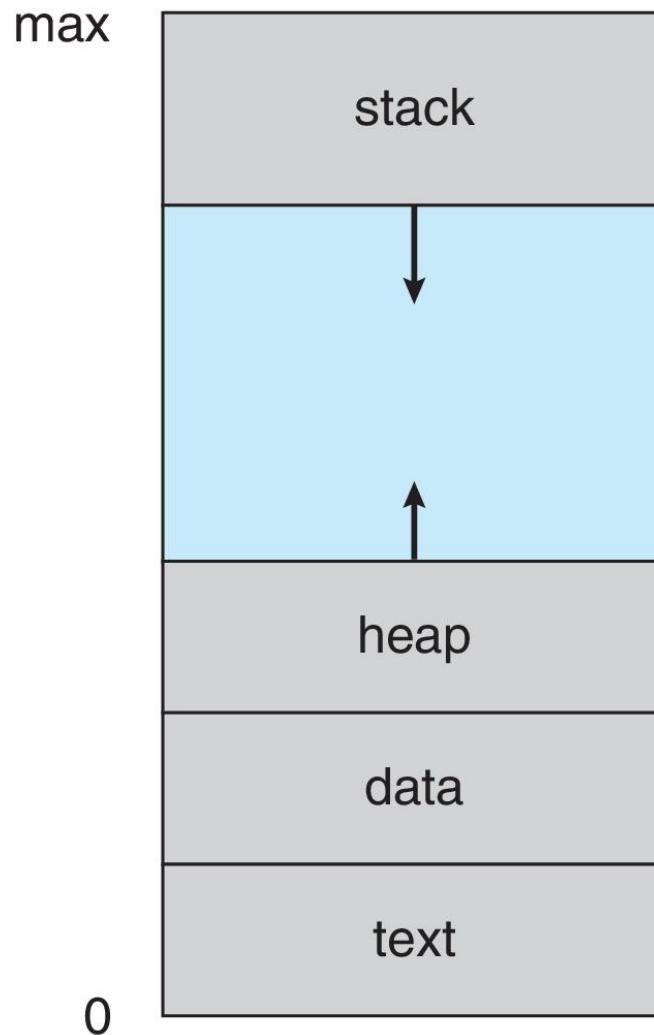
# Process Concept (Cont.)

- Program is **passive** entity stored on disk (**executable file**); process is **active**
  - Program becomes process when an executable file is 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



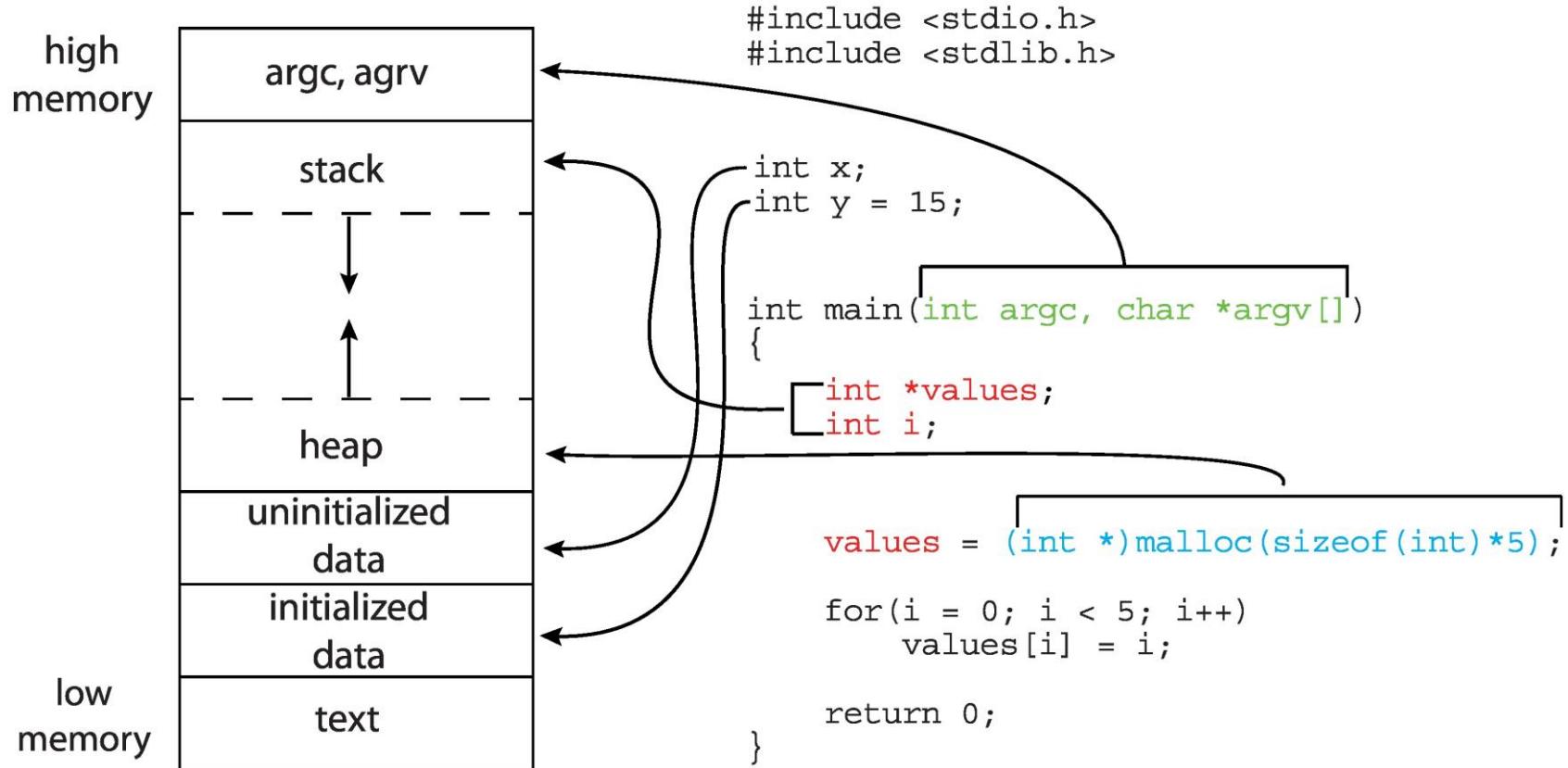


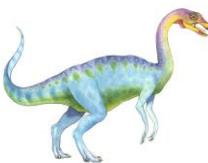
# Process in Memory





# Memory Layout of a C Program

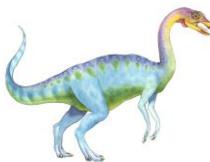




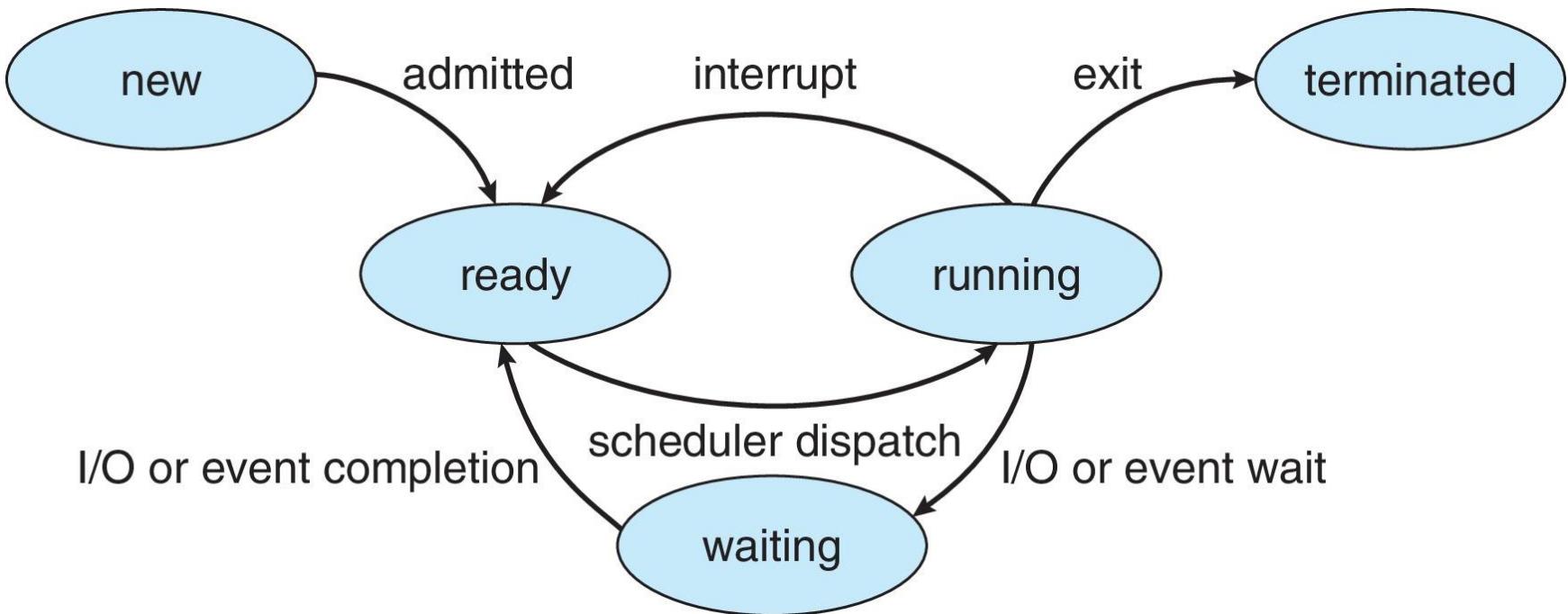
# Process State

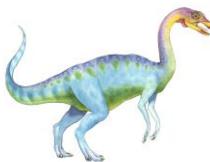
- As a process executes, it changes **state**
  - **New:** The process is being created
  - **Running:** Instructions are being executed
  - **Waiting:** The process is waiting for some event to occur
  - **Ready:** The process is waiting to be assigned to a processor
  - **Terminated:** The process has finished execution





# Diagram of Process State





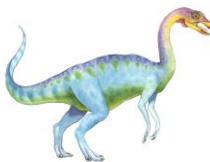
# Process Control Block (PCB)

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

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

process state
process number
program counter
registers
memory limits
list of open files
• • •

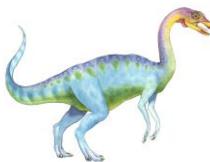




# Threads

- 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

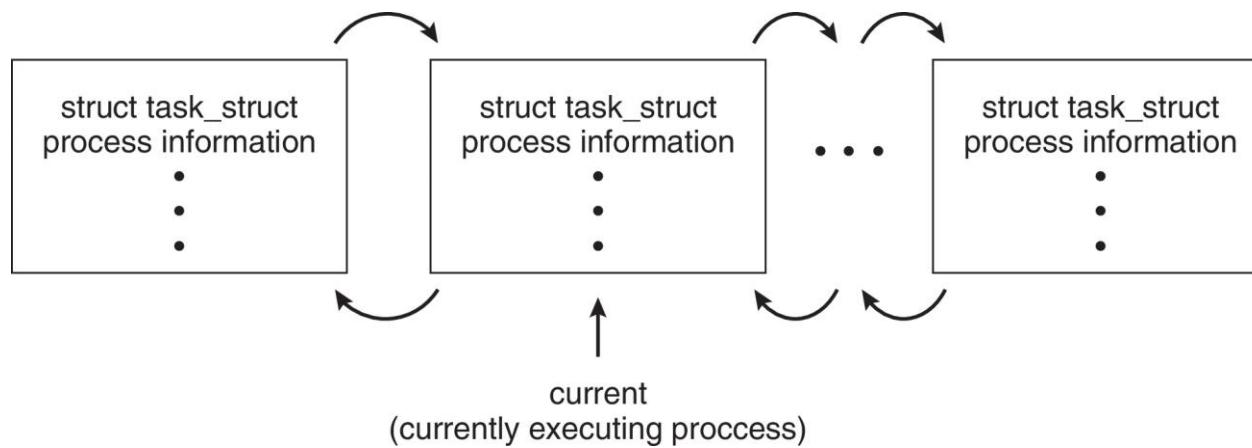


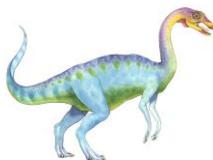


# Process Representation in Linux

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 */
```





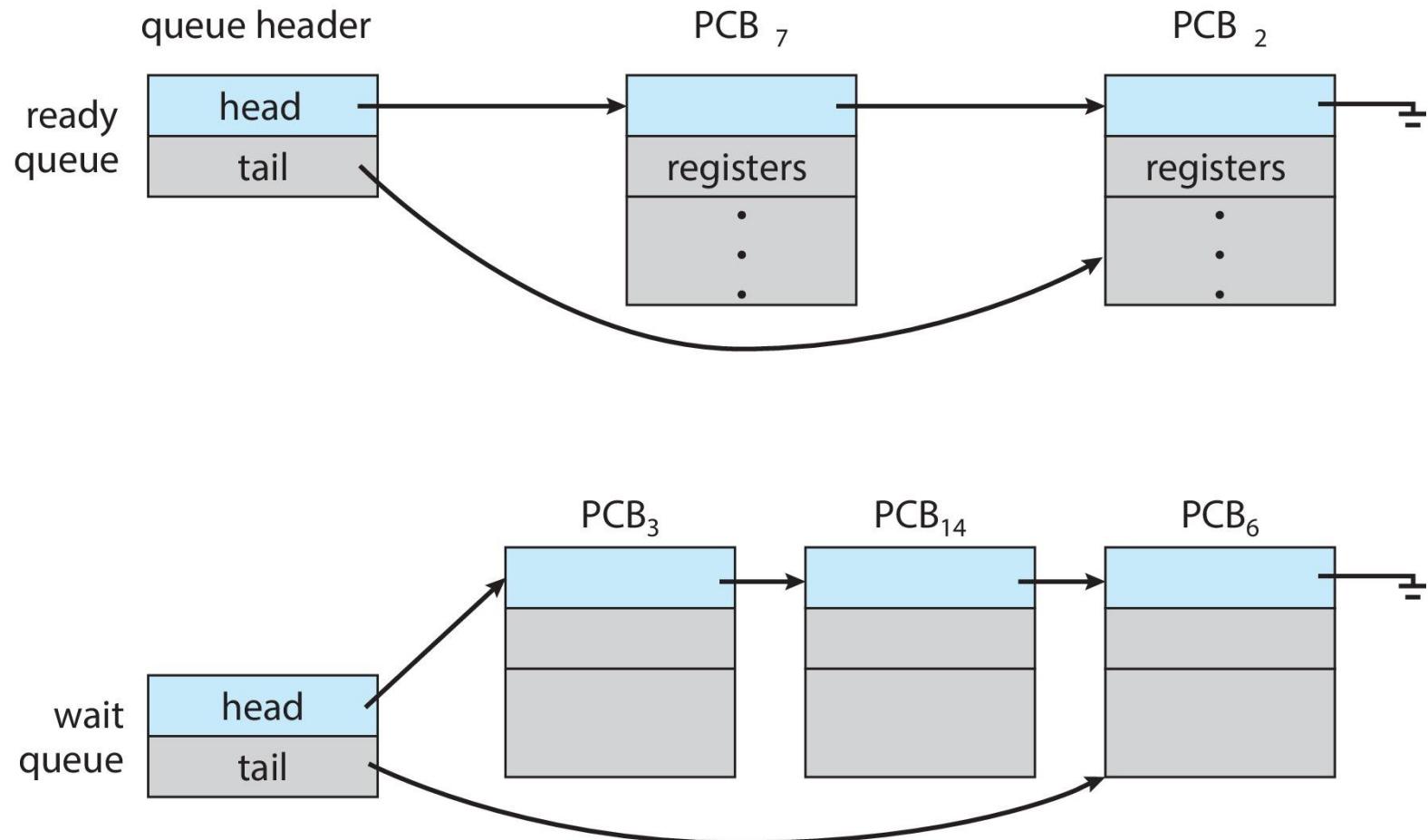
# Process Scheduling

- **Process scheduler** selects among available processes for next execution on CPU core
- Goal -- Maximize CPU use, quickly switch processes onto 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



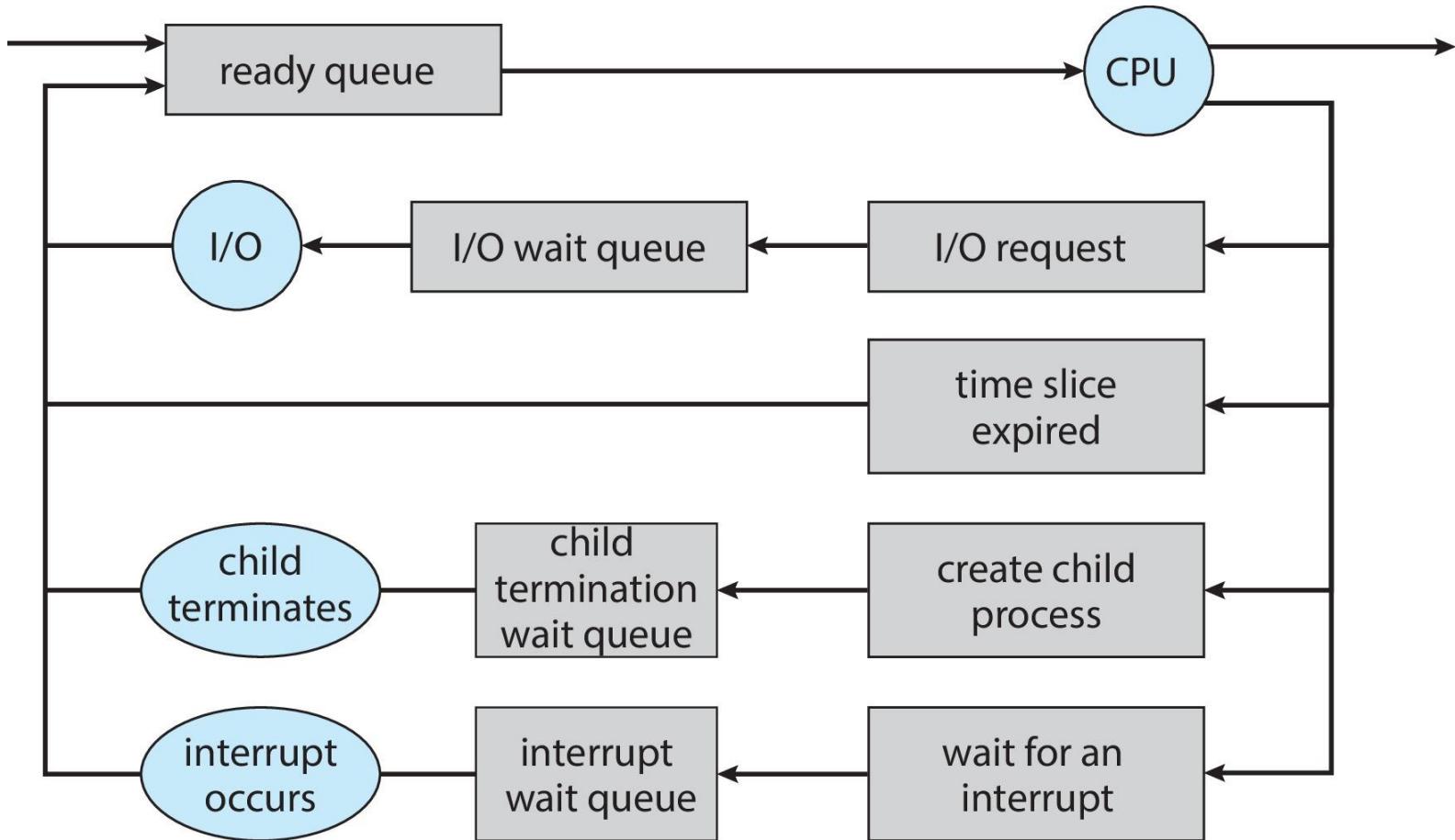


# Ready and Wait Queues





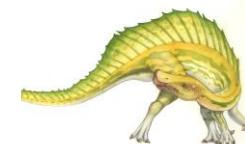
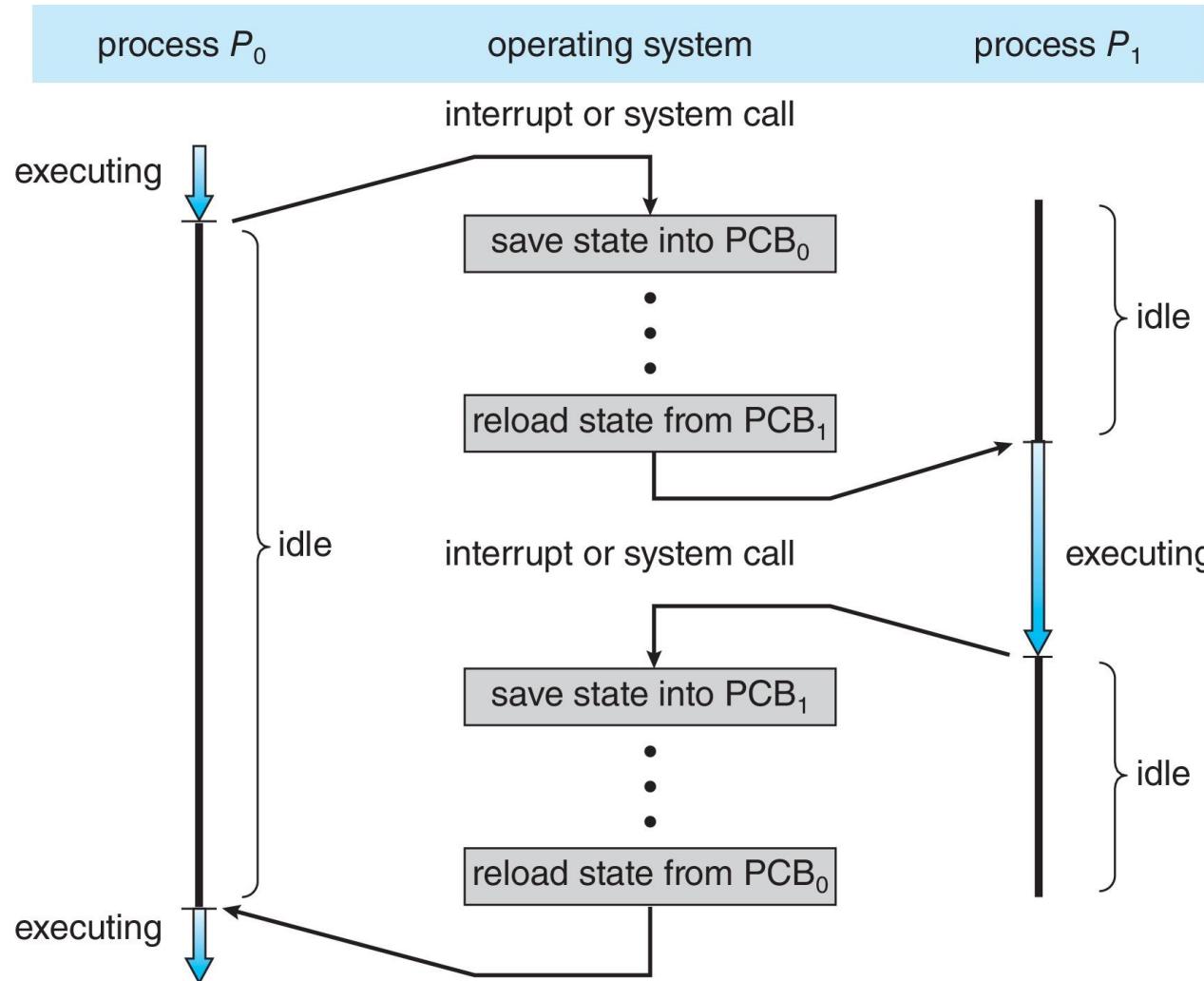
# Representation of Process Scheduling

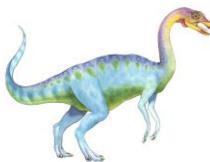




# CPU Switch From Process to Process

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

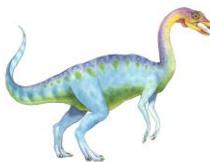




# Context Switch

- 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 pure overhead; the system does no useful work while switching
  - The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

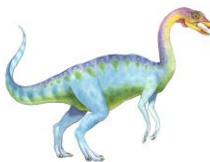




# Multitasking in Mobile Systems

- 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
  - 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



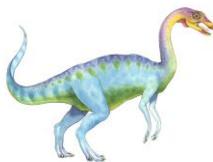


# Operations on Processes

---

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





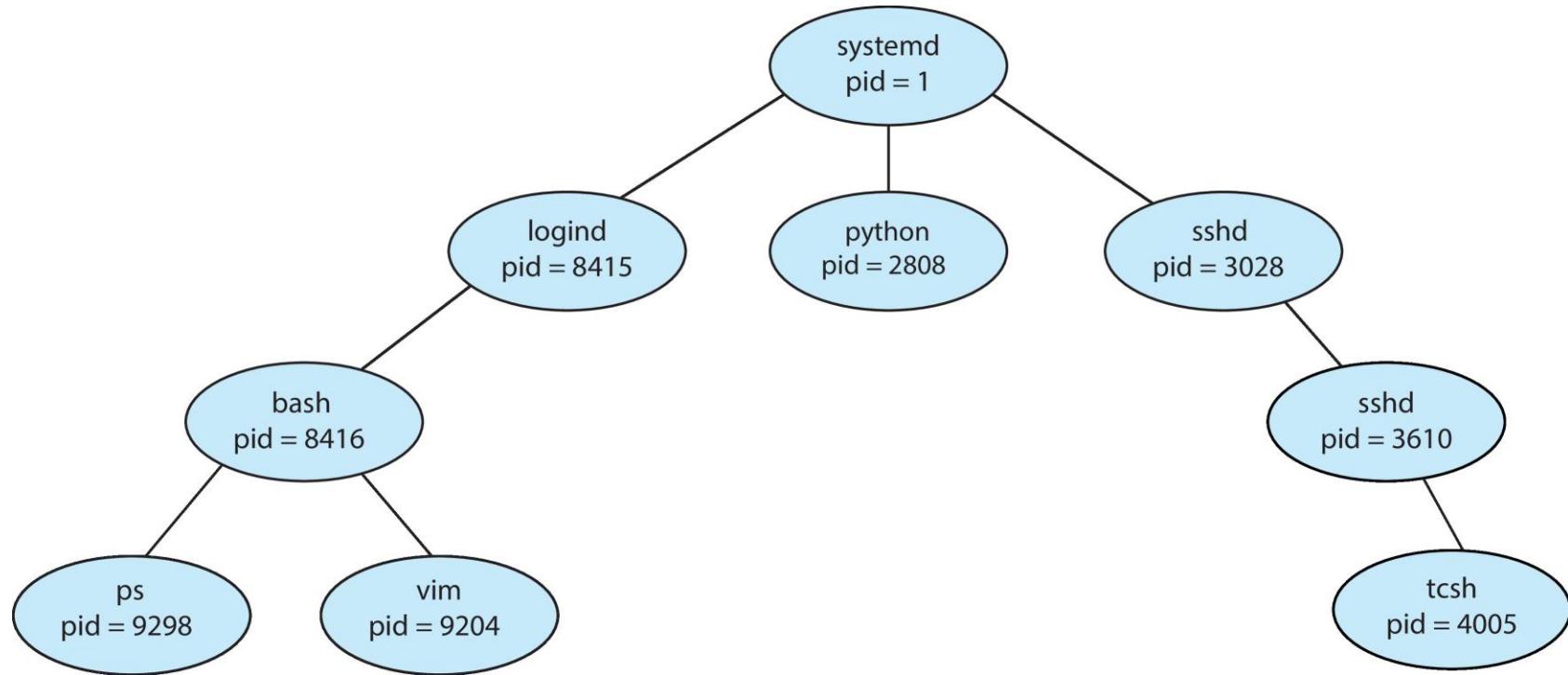
# Process Creation

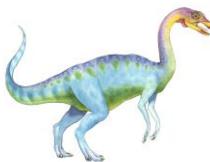
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier (pid)**
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate





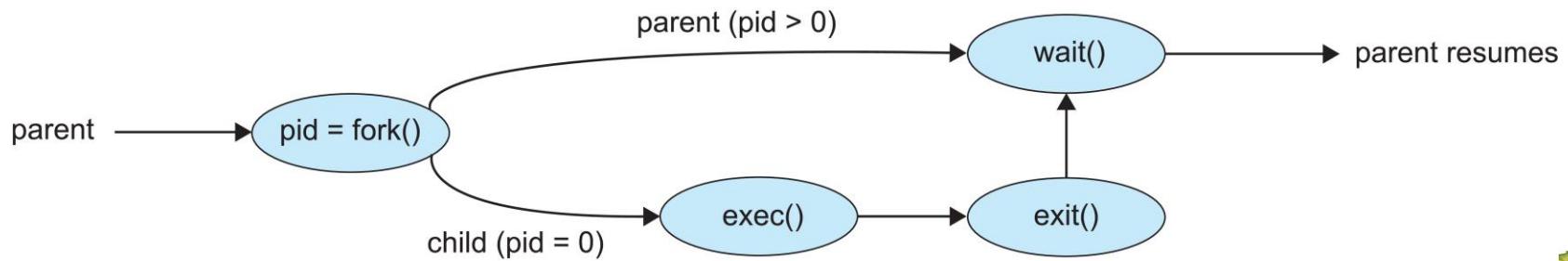
# A Tree of Processes in Linux

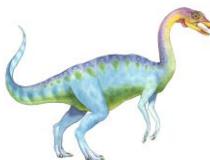




# Process Creation (Cont.)

- 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()** waiting for the child to terminate





# C Program Forking Separate Process

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

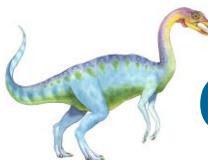
int main()
{
    pid_t pid;

    /* 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>

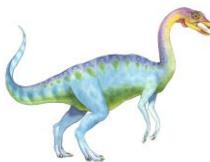
int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    /* 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);
}
```

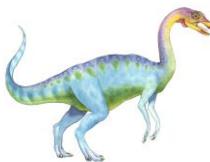




# Process Termination

- Process executes last statement and then asks the OS to delete it using the **exit()** system call
  - Returns status data from child to parent (via **wait()**)
  - Process' resources are released by OS
- 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 OS does not allow a child to continue if its parent terminates





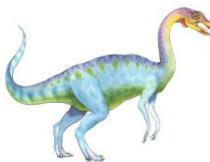
# Process Termination

- Some OS do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated
  - **cascading termination.** All children, grandchildren, etc., are terminated
  - The termination is initiated by the OS
- 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

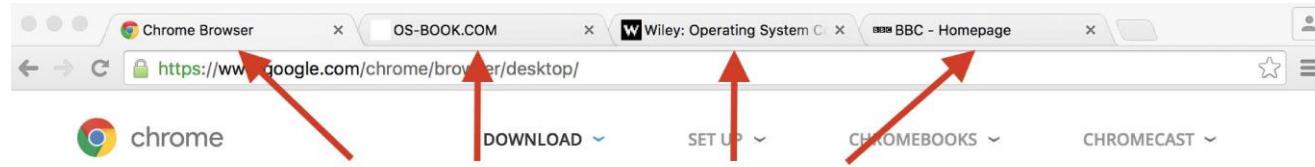
- Mobile OS often have to terminate processes to reclaim system resources such as memory. From **most to least** important:
  - Foreground process
  - Visible process
  - Service process
  - Background process
  - Empty process
- Android will begin terminating processes that are least important



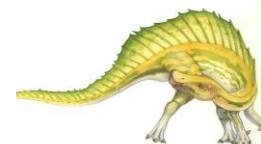


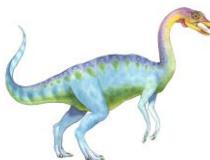
# Multiprocess Architecture – Chrome Browser

- 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



Each tab represents a separate process.

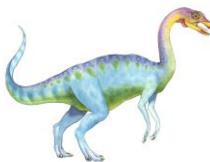




# Interprocess Communication

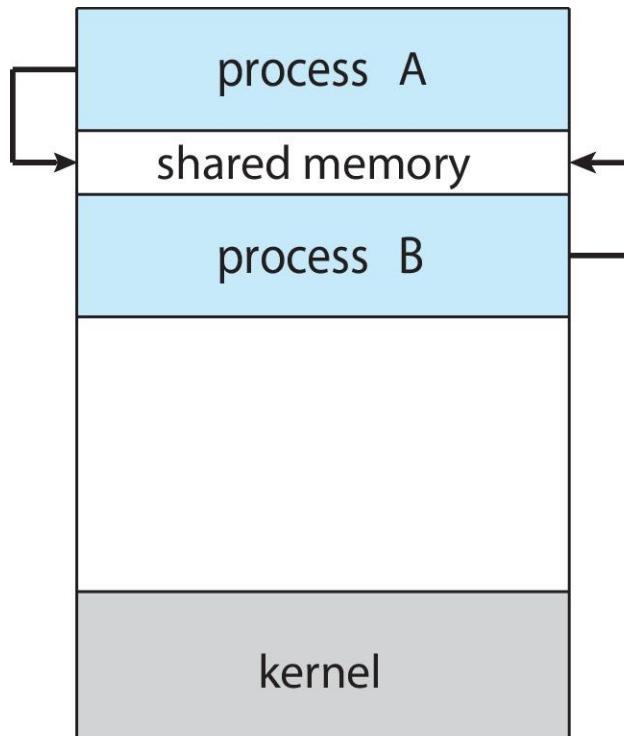
- Processes within a system may be *independent* or *cooperating*
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need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





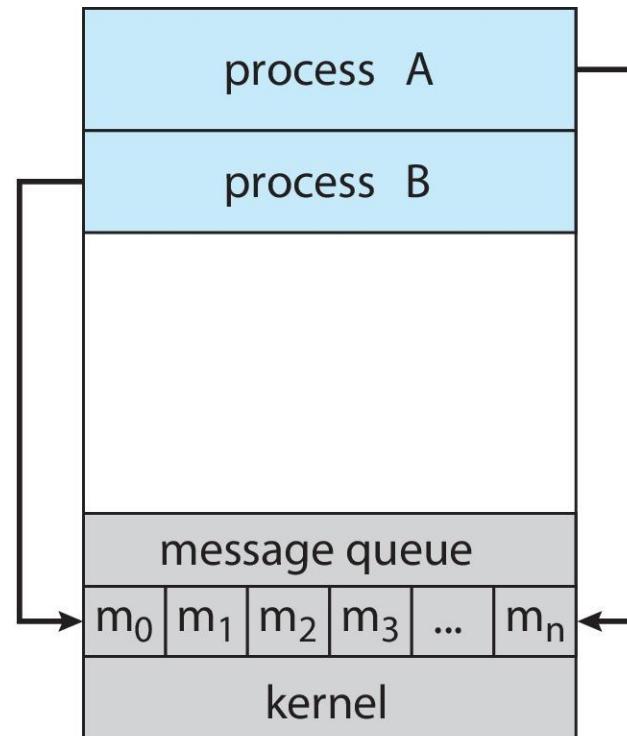
# Communications Models

(a) Shared memory



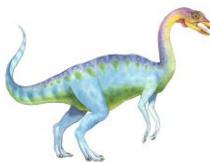
(a)

(b) Message passing



(b)

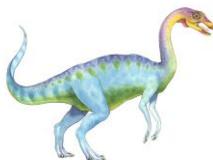




# Producer-Consumer Problem

- Paradigm for cooperating processes:
  - *producer* process produces information that is consumed by a *consumer* process
- Two variations:
  - **unbounded-buffer** places no practical limit on the size of the buffer:
    - ▶ Producer never waits
    - ▶ Consumer waits if there is no buffer to consume
  - **bounded-buffer** assumes that there is a fixed buffer size
    - ▶ Producer must wait if all buffers are full
    - ▶ Consumer waits if there is no buffer to consume



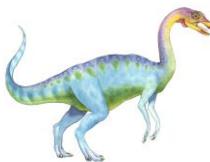


## IPC – Shared Memory

---

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the **user processes** not the OS
- Major issue 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





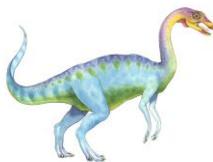
# Bounded-Buffer – Shared-Memory Solution

- 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



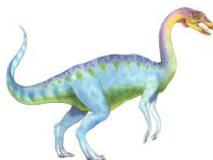


# Producer Process – Shared Memory

```
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;
}
```

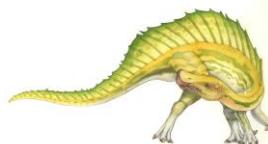


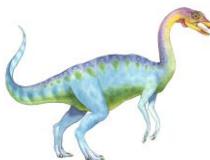


# Consumer Process – Shared Memory

```
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 */
}
```

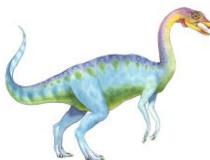




# What about Filling all the Buffers?

- 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
- The integer **counter** is incremented by the producer after it produces a new buffer
- The integer **counter** is decremented by the consumer after it consumes a buffer





# Producer

---

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

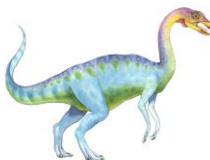




# Consumer

```
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
```

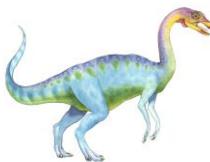
- `counter--` could be implemented as

```
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}



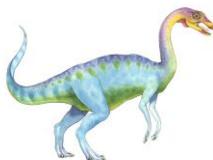


# Race Condition (Cont.)

---

- Question – why was there no race condition in the first solution where at most  $(N - 1)$  buffers can be filled?
- More in Chapter 6



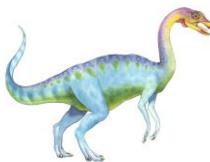


# IPC – Message Passing

---

- 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

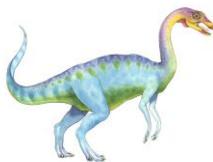




# Message Passing (Cont.)

- 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?

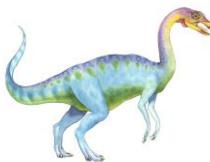




# Implementation of Communication Link

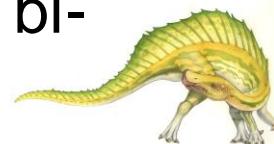
- Physical:
  - Shared memory
  - Hardware bus
  - Network
- Logical:
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering





# 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



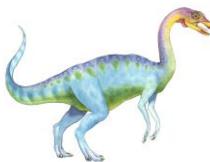


# Indirect Communication

---

- 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

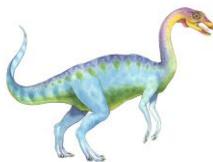




# Indirect Communication (Cont.)

- Operations
  - Create a new mailbox (port)
  - Send and receive messages through mailbox
  - Delete a mailbox
- Primitives are defined as:
  - **send(A, message)** – send a message to mailbox A
  - **receive(A, message)** – receive a message from mailbox A

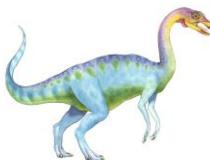




# Indirect Communication (Cont.)

- 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

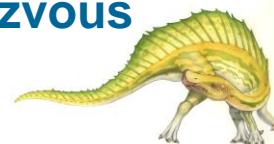




# Synchronization

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**



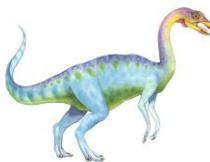


# Buffering

---

- 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





# Producer-Consumer: Message Passing

- Producer

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

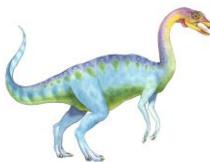
    send(next_produced);
}
```

- Consumer

```
message next_consumed;
while (true) {
    receive(next_consumed)

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





# 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

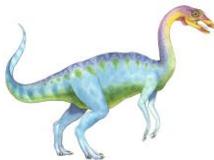
- Set the size of the object

```
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 object */
    void *ptr;

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

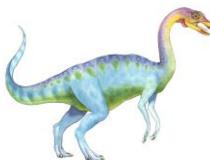
    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* 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);

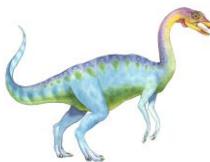
    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

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

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

    return 0;
}
```

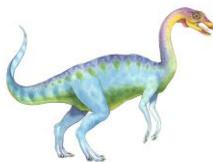




# Examples of IPC Systems - Mach

- 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





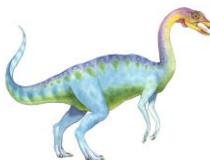
# Mach Messages

```
#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

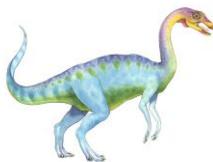
```
/* Client Code */

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
);
```





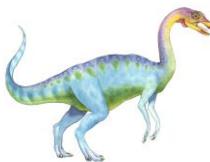
# Mach Message Passing - Server

```
/* 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
);
```

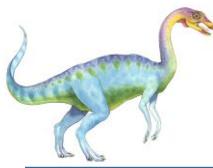




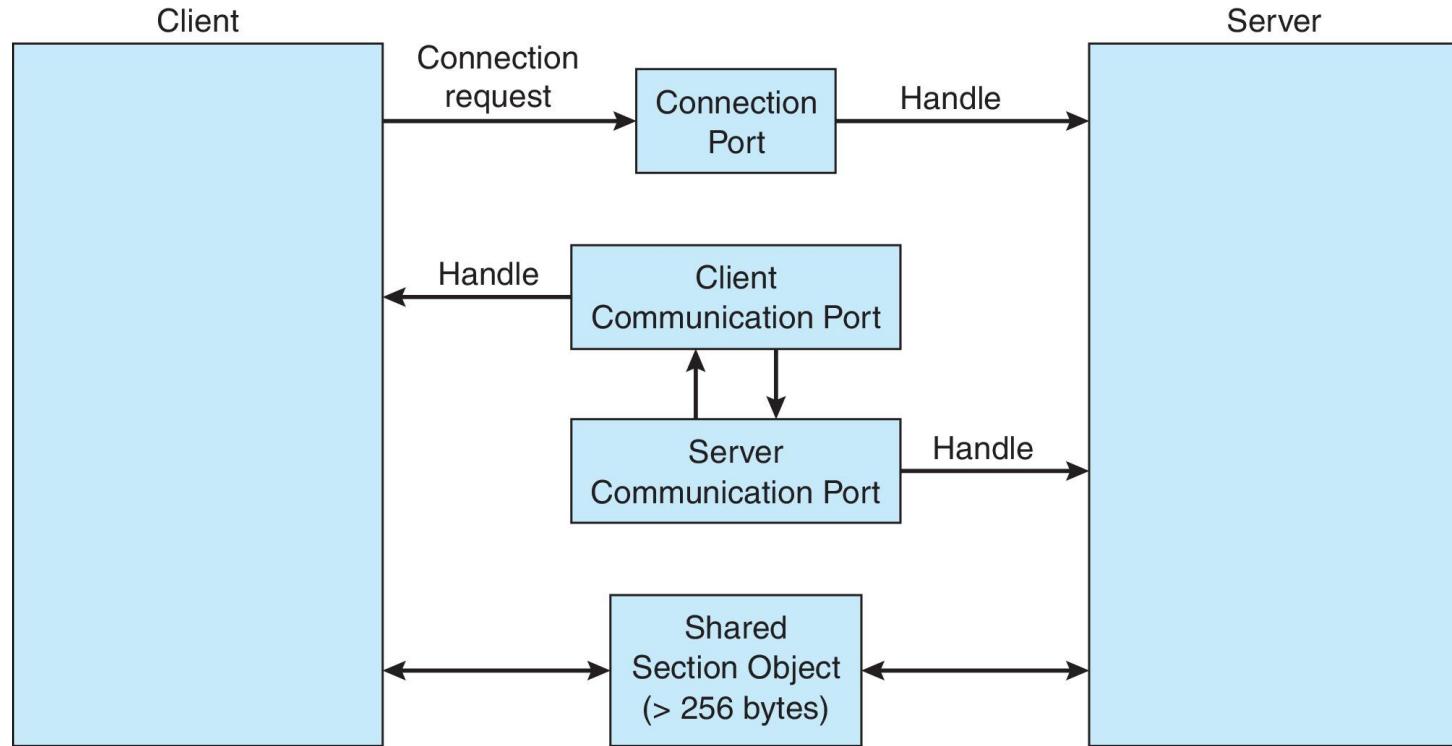
# Examples of IPC Systems – Windows

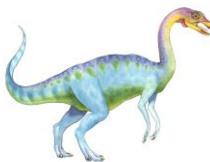
- Message-passing centric via **advanced local procedure call (LPC) facility**
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - The client opens a handle to the subsystem's **connection 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

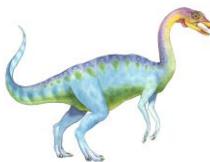




# Pipes

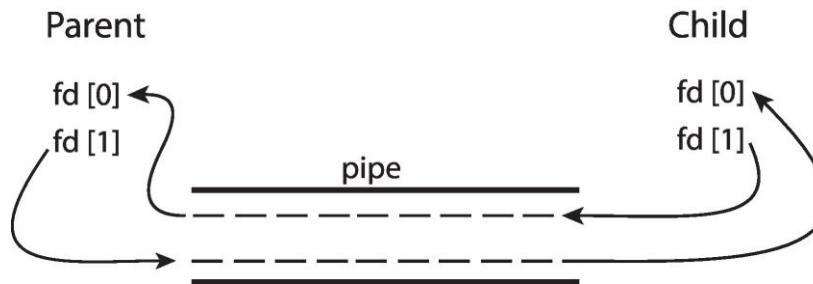
- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., **parent-child**) between the communicating processes?
  - Can the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





# Examples of Pipes in UNIX and Windows

---

- Ordinary Pipes in UNIX
  - fork(), read(), write(), wait(), close()
  - (See Fig. 3.25 & 3.26)
- Windows anonymous pipe – parent & child process
  - CreateProcess(), WriteFile(), ReadFile(), WaitForSingleObject(), CloseHandle()
  - (See Fig. 3.27, 3.28, & 3.29)





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

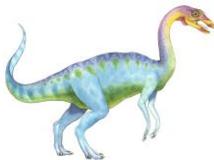
#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
    char write_msg[BUFFER_SIZE] = "Greetings";
    char read_msg[BUFFER_SIZE];
    int fd[2];
    pid_t pid;

    /* Program continues in Figure 3.26 */
}
```

**Figure 3.25** Ordinary pipe in UNIX.





```
/* create the pipe */
if (pipe(fd) == -1) {
    fprintf(stderr,"Pipe failed");
    return 1;
}

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

if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
}

if (pid > 0) { /* parent process */
    /* close the unused end of the pipe */
    close(fd[READ_END]);

    /* write to the pipe */
    write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

    /* close the write end of the pipe */
    close(fd[WRITE_END]);
}
else { /* child process */
    /* close the unused end of the pipe */
    close(fd[WRITE_END]);

    /* read from the pipe */
    read(fd[READ_END], read_msg, BUFFER_SIZE);
    printf("read %s",read_msg);

    /* close the write end of the pipe */
    close(fd[READ_END]);
}

return 0;
}
```

Figure 3.26 Figure 3.25, continued.





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

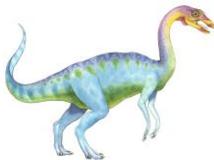
#define BUFFER_SIZE 25

int main(VOID)
{
    HANDLE ReadHandle, WriteHandle;
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    char message[BUFFER_SIZE] = "Greetings";
    DWORD written;

    /* Program continues in Figure 3.28 */
}
```

**Figure 3.27** Windows anonymous pipe – parent process.





```
/* set up security attributes allowing pipes to be inherited */
SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};
/* allocate memory */
ZeroMemory(&pi, sizeof(pi));

/* create the pipe */
if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
    fprintf(stderr, "Create Pipe Failed");
    return 1;
}

/* establish the START_INFO structure for the child process */
GetStartupInfo(&si);
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

/* redirect standard input to the read end of the pipe */
si.hStdInput = ReadHandle;
si.dwFlags = STARTF_USESTDHANDLES;

/* don't allow the child to inherit the write end of pipe */
SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

/* create the child process */
CreateProcess(NULL, "child.exe", NULL, NULL,
    TRUE, /* inherit handles */
    0, NULL, NULL, &si, &pi);

/* close the unused end of the pipe */
CloseHandle(ReadHandle);

/* the parent writes to the pipe */
if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL))
    fprintf(stderr, "Error writing to pipe.");

/* close the write end of the pipe */
CloseHandle(WriteHandle);

/* wait for the child to exit */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
return 0;
}
```

Figure 3.28 Figure 3.27, continued.





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

#define BUFFER_SIZE 25

int main(VOID)
{
    HANDLE Readhandle;
    CHAR buffer[BUFFER_SIZE];
    DWORD read;

    /* get the read handle of the pipe */
    ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

    /* the child reads from the pipe */
    if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
        printf("child read %s",buffer);
    else
        fprintf(stderr, "Error reading from pipe");

    return 0;
}
```

**Figure 3.29** Windows anonymous pipes – child process.



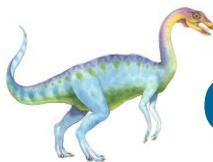


# Named Pipes

---

- 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

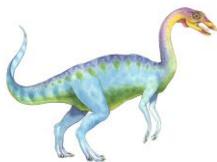




# Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls

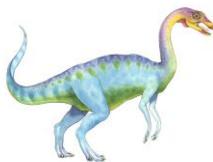




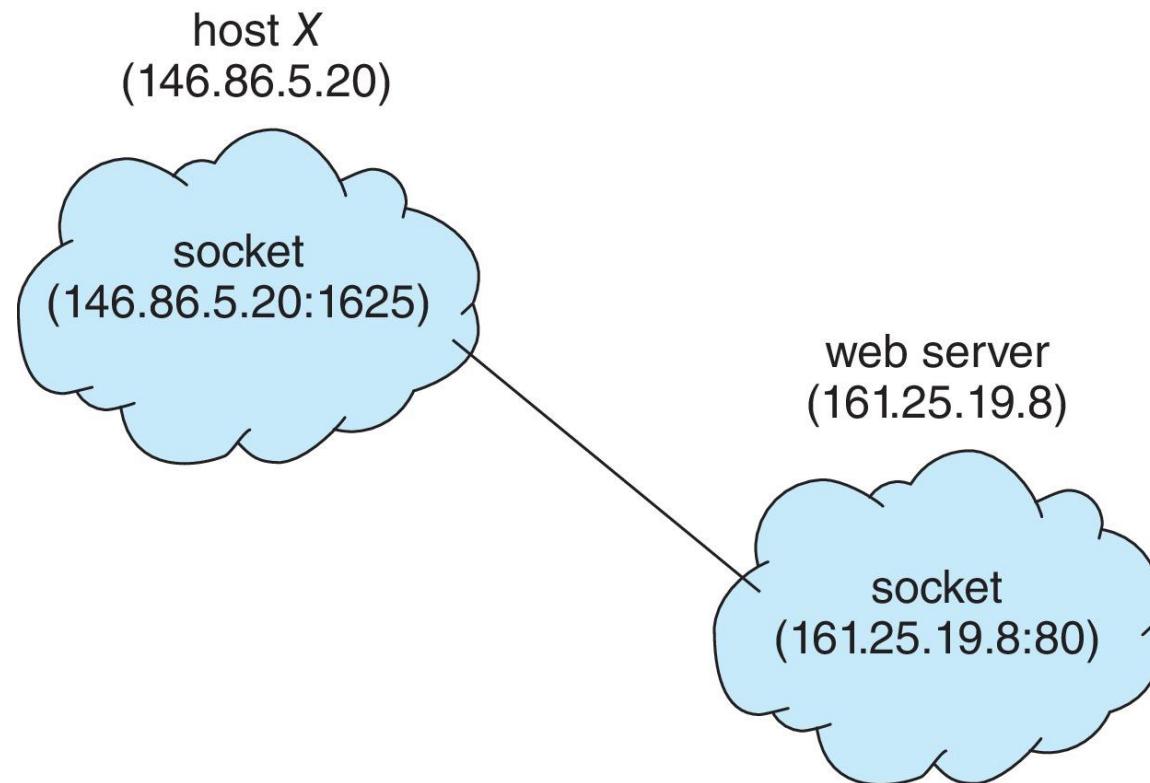
# Sockets

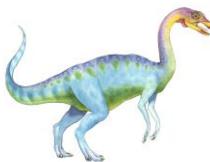
- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between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





# Socket Communication

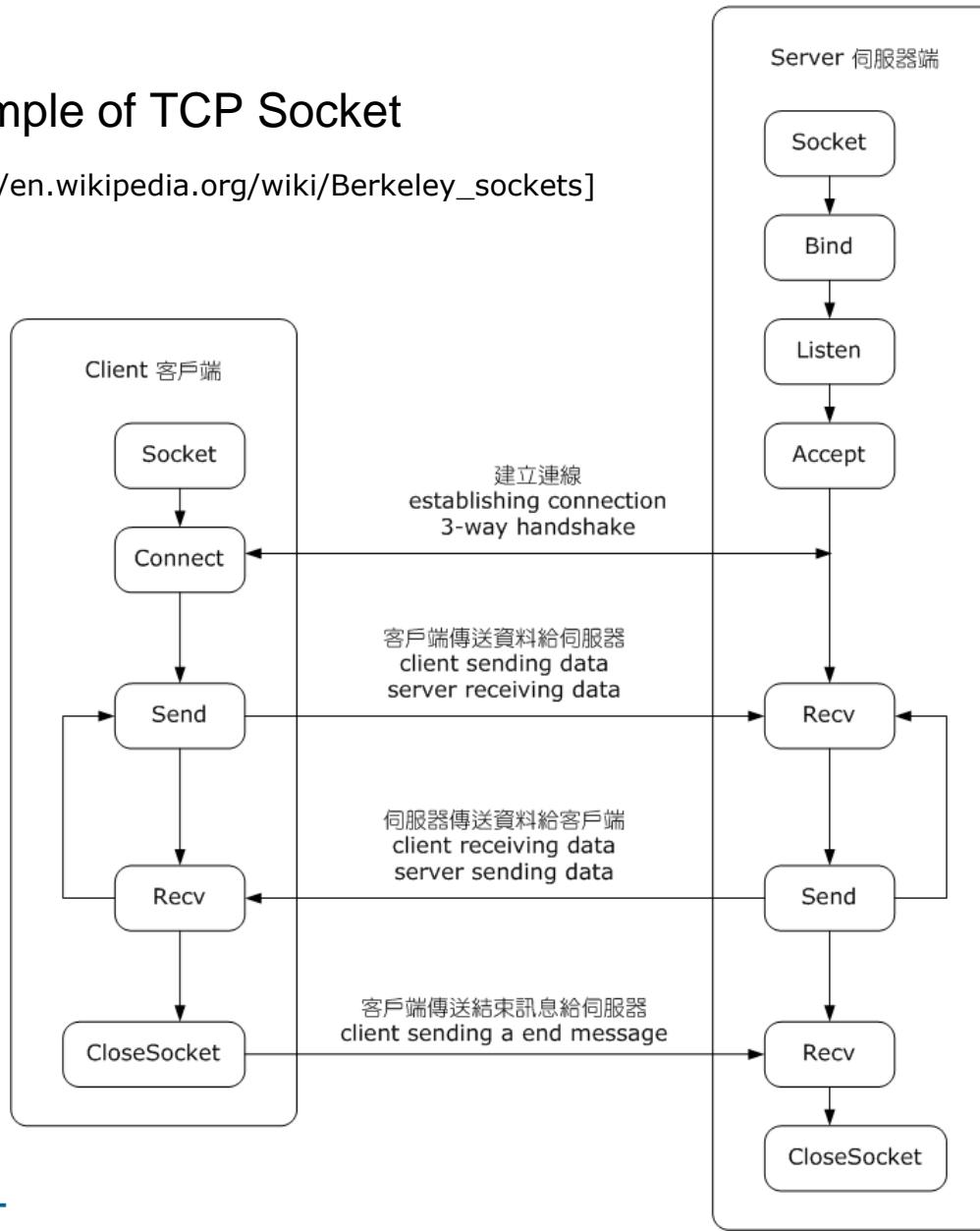


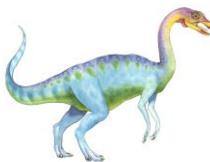


# Berkeley Socket API (in C)

## ■ An Example of TCP Socket

[Source: [https://en.wikipedia.org/wiki/Berkeley\\_sockets](https://en.wikipedia.org/wiki/Berkeley_sockets)]





# Sockets in Java

- Three types of sockets
  - Connection-oriented (TCP)
  - Connectionless (UDP)
  - MulticastSocket class – data can be sent to multiple recipients
- Consider this “Date” server in Java:

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

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

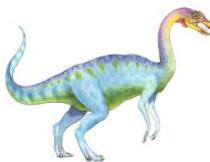
            /* now listen for connections */
            while (true) {
                Socket client = sock.accept();

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

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

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





# Sockets in Java

## The equivalent Date client

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

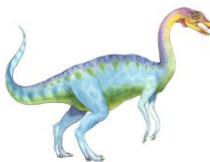
public class DateClient
{
    public static void main(String[] args) {
        try {
            /* make connection to server socket */
            Socket sock = new Socket("127.0.0.1",6013);

            InputStream in = sock.getInputStream();
            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));

            /* read the date from the socket */
            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);

            /* close the socket connection*/
            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

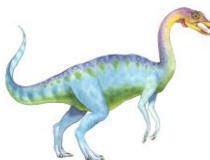




# Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters
- 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)**





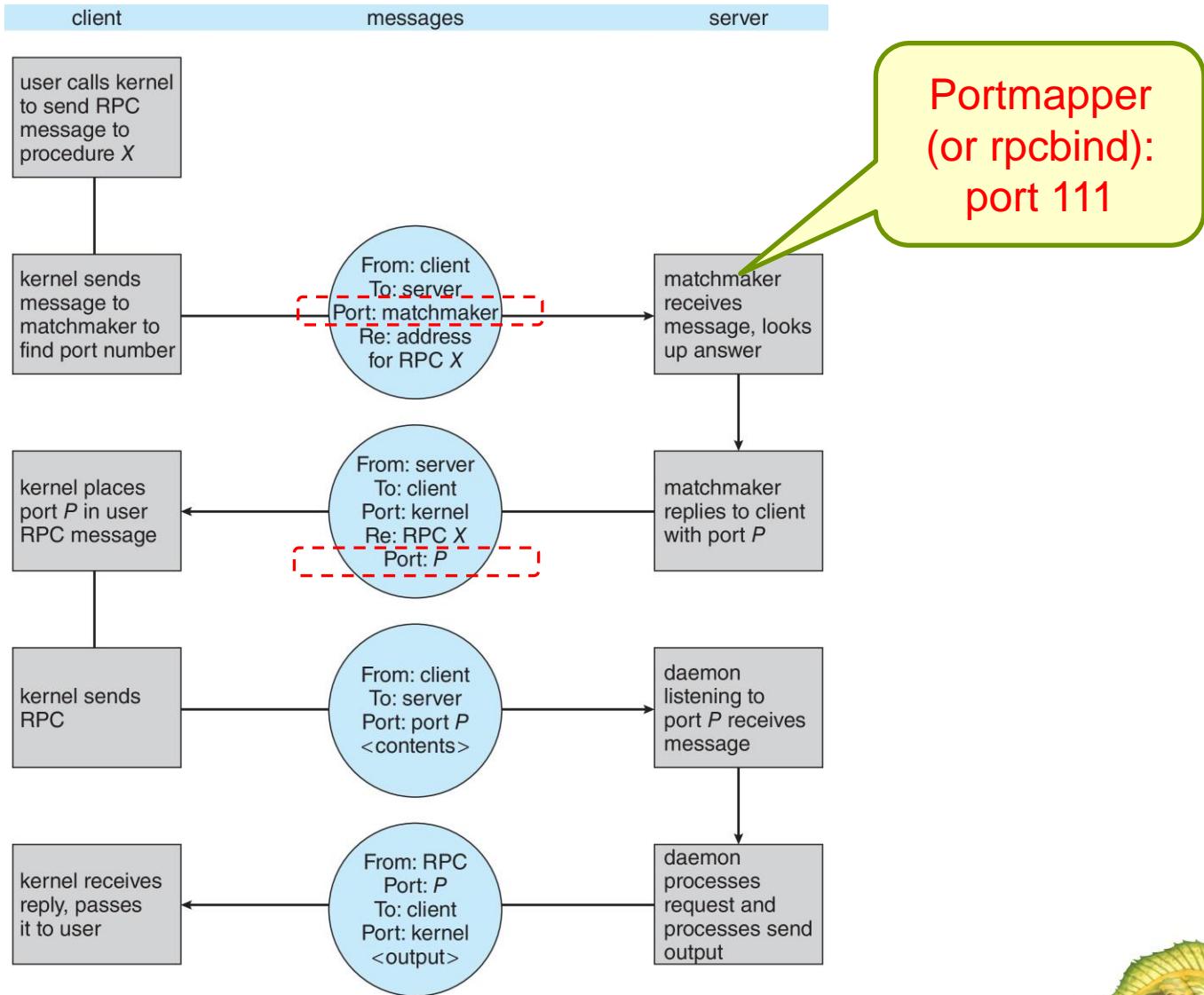
# Remote Procedure Calls (Cont.)

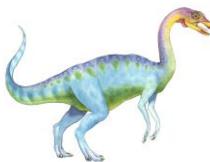
- Data representation handled via **External Data Representation (XDL)** format to account for different architectures
  - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
  - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server





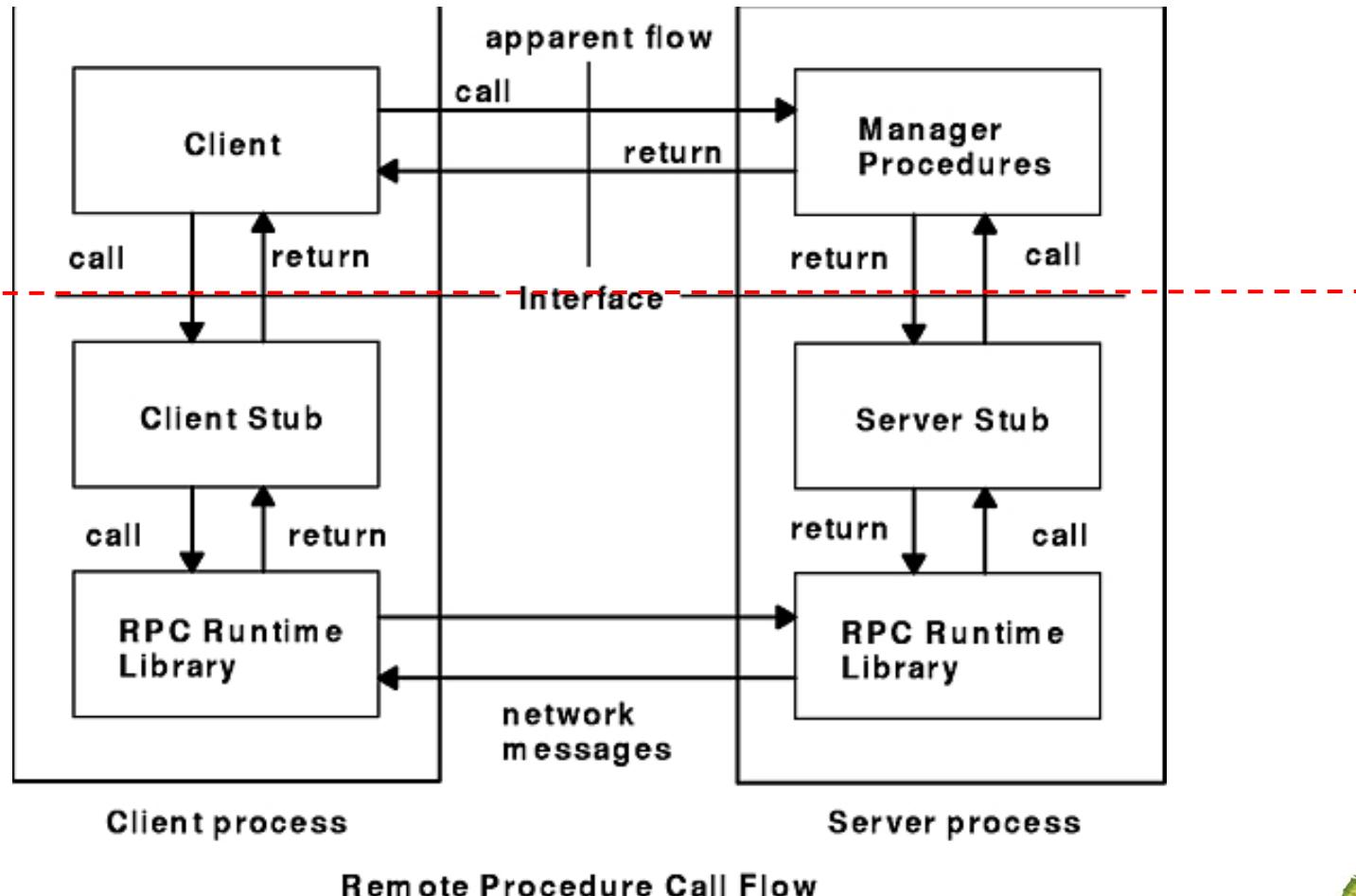
# Execution of RPC





# RPC Flow

- Check RFC 5531 for more details

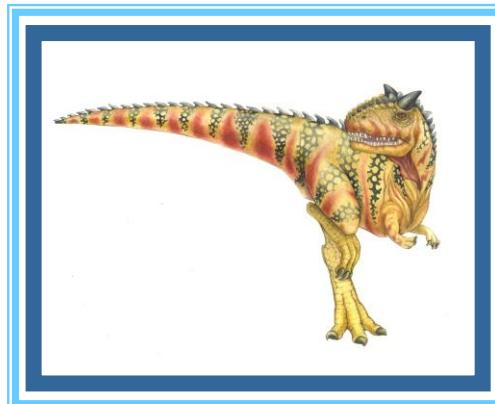


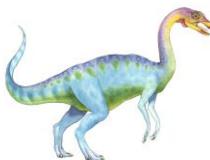
[Source:

[https://www.ibm.com/support/knowledgecenter/en/ssw\\_aix\\_71/commprogramming/rpc\\_mod.html](https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/commprogramming/rpc_mod.html)]



# End of Chapter 3

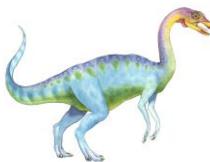




# Producer-Consumer Problem

- Paradigm for cooperating processes:
  - *producer* process produces information that is consumed by a *consumer* process
- Two variations:
  - **unbounded-buffer** places no practical limit on the size of the buffer:
    - ▶ Producer never waits
    - ▶ Consumer waits if there is no buffer to consume
  - **bounded-buffer** assumes that there is a fixed buffer size
    - ▶ Producer must wait if all buffers are full
    - ▶ Consumer waits if there is no buffer to consume



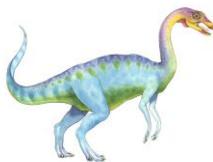


# Cooperating Processes

---

- ***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





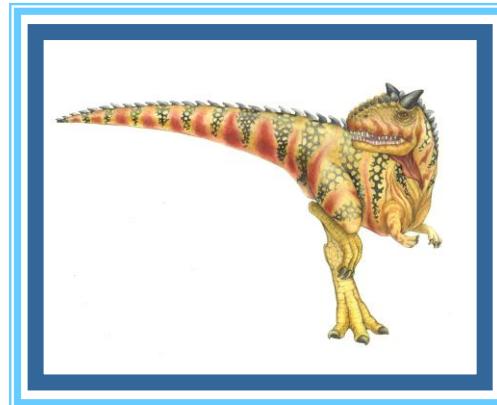
# Synchronization

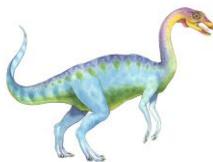
---

- 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**



# Chapter 4: Threads & Concurrency



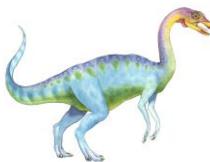


# Outline

---

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



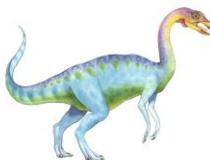


# Objectives

---

- Identify the basic components of a **thread**, and contrast threads with 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 Windows and Linux represent threads
- Design multithreaded applications using the Pthreads, Java, and Windows **threading APIs**



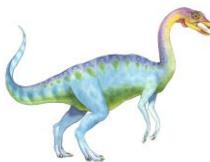


# Motivation

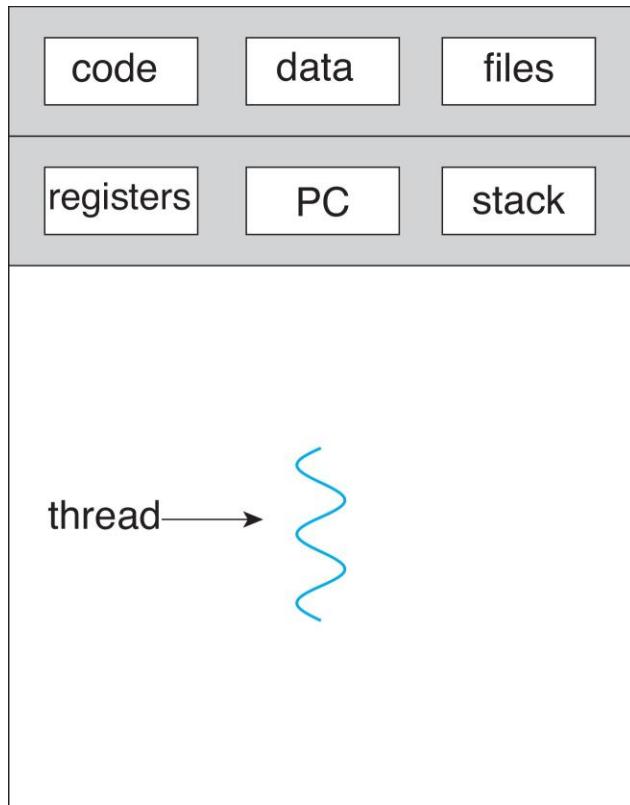
---

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

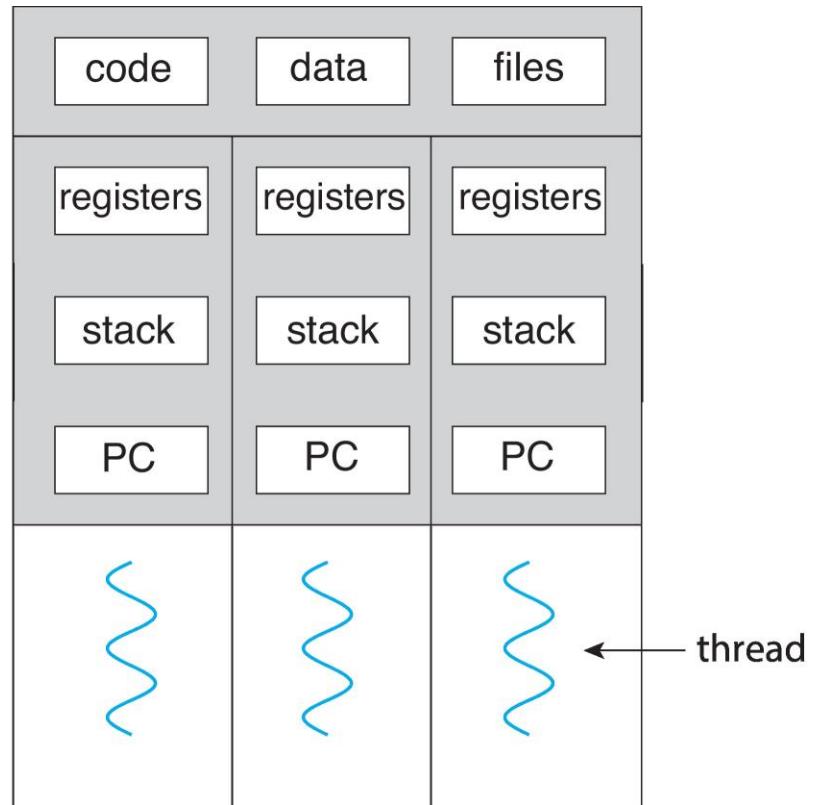




# Single and Multithreaded Processes

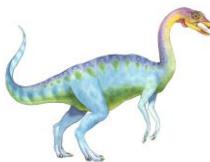


single-threaded process

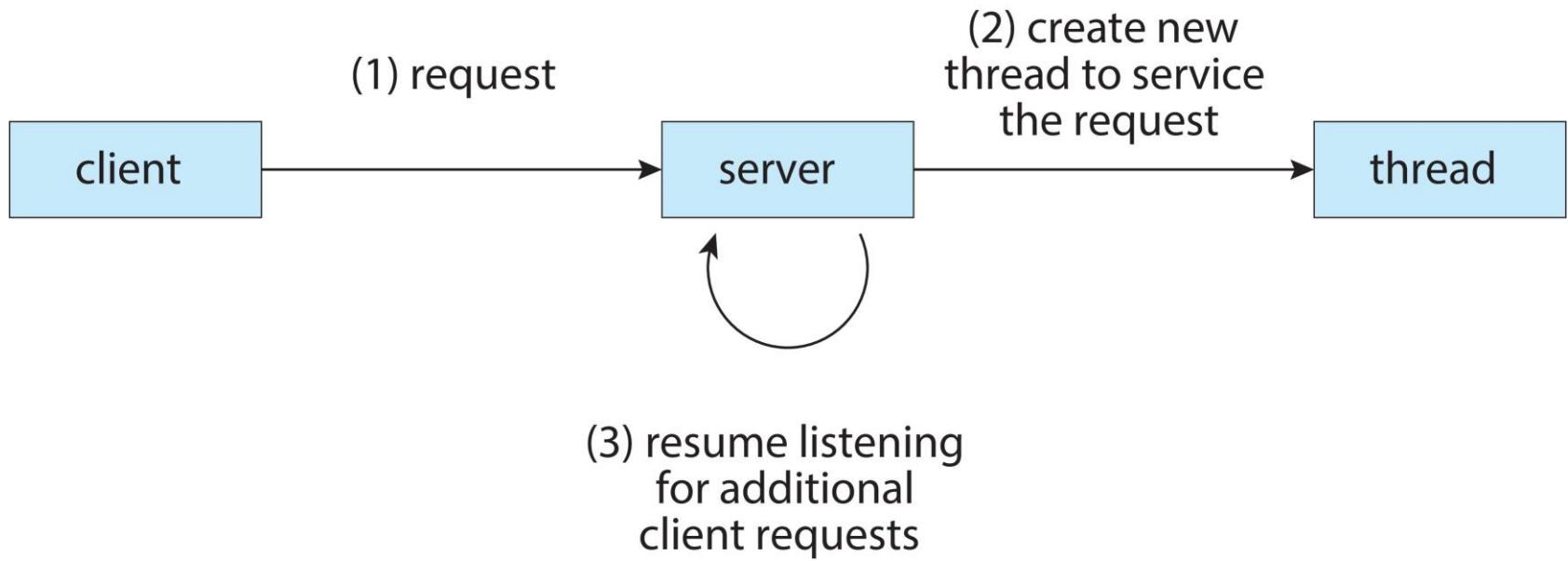


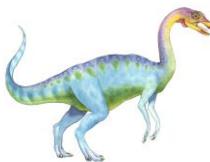
multithreaded process





# Multithreaded Server Architecture



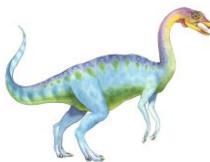


# Benefits

---

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

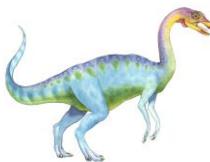




# Multicore Programming

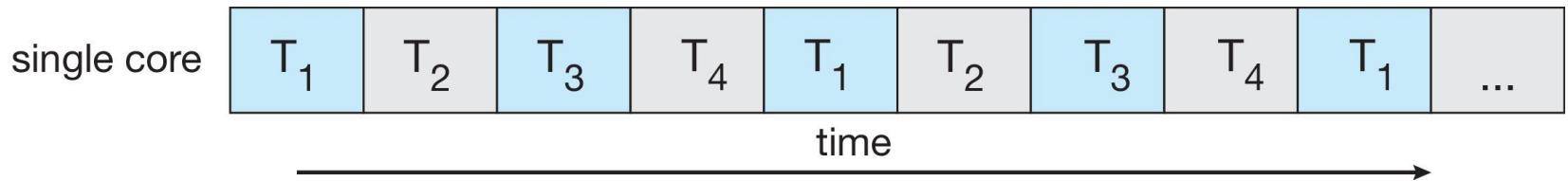
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Load balancing**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- **Parallelism** implies a system can perform more than one task simultaneously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



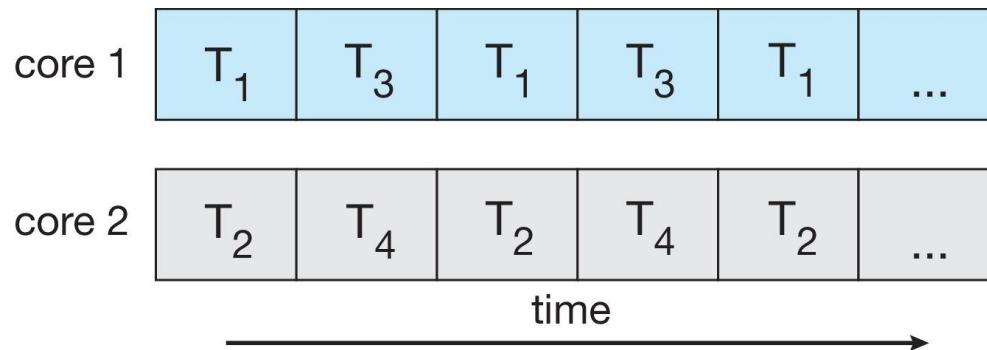


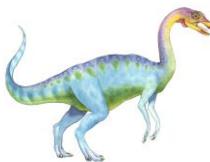
# Concurrency vs. Parallelism

- Concurrent execution on single-core system:



- Parallelism on a multi-core system:

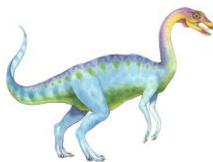




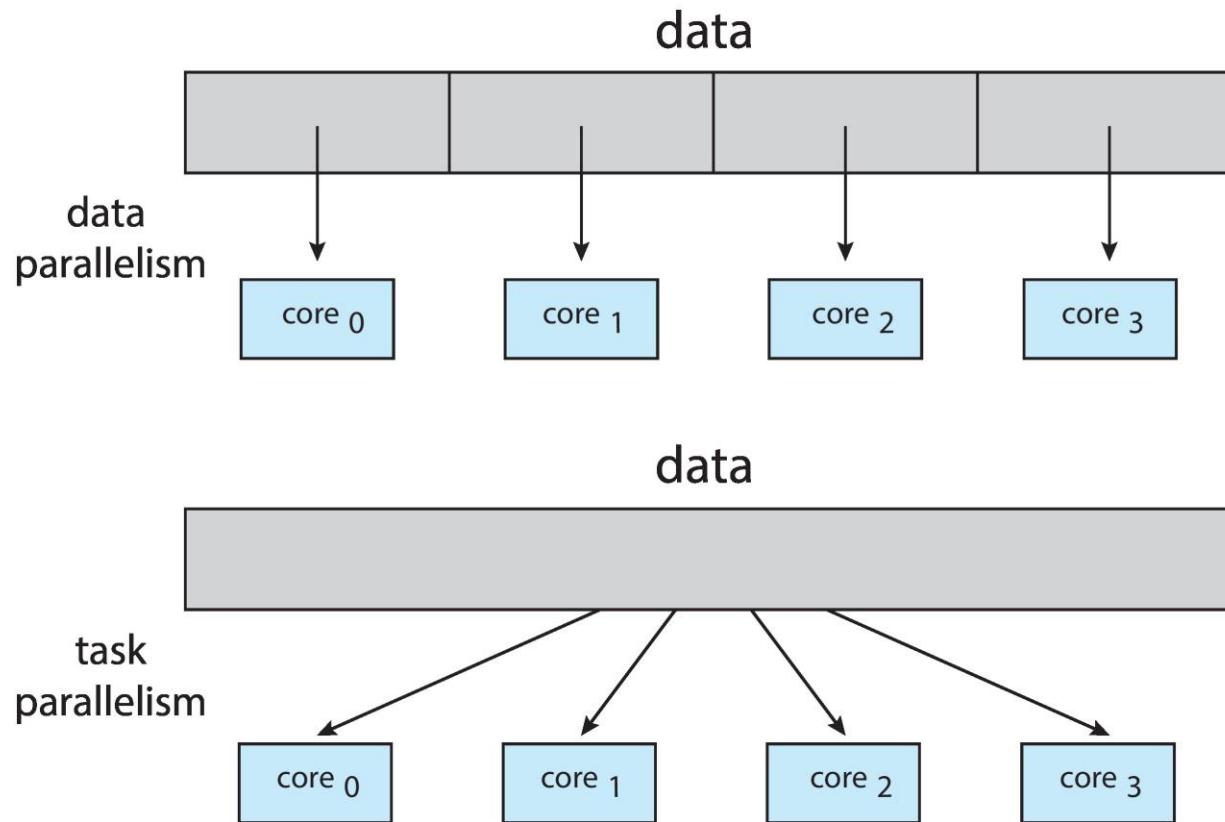
# Multicore Programming

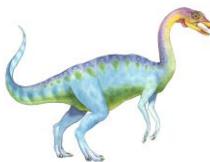
- Types of parallelism
  - **Data parallelism** – distributes different subsets of the whole data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation





# Data and Task Parallelism





# Amdahl's Law

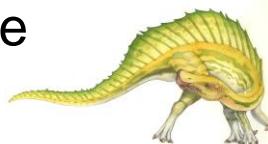
- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- S is serial portion,  $N$  processing cores

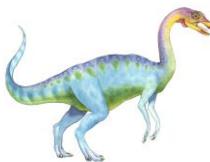
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As  $N$  approaches infinity, speedup approaches  $1 / S$

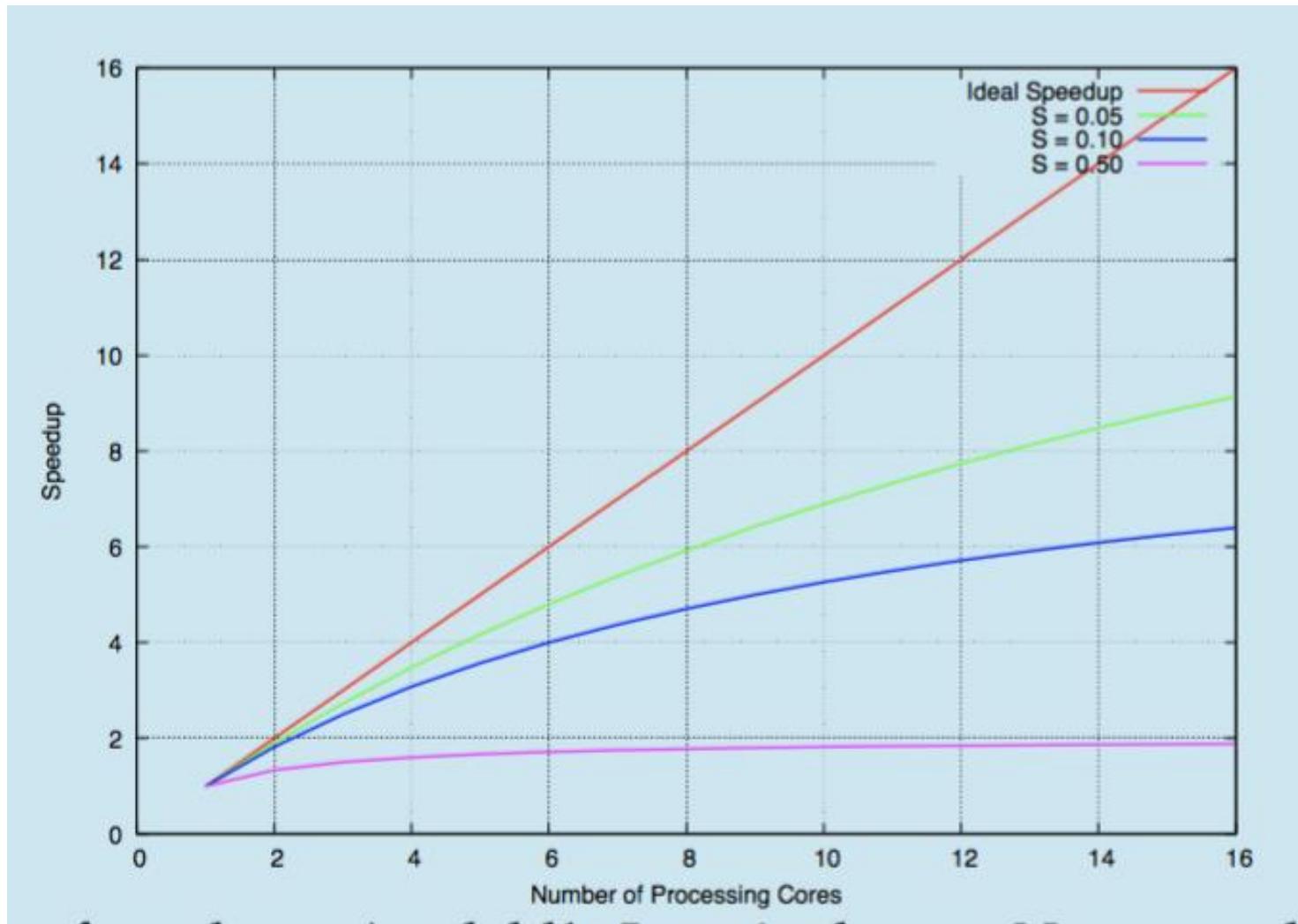
**Serial portion of an application has disproportionate effect on performance gained by adding additional cores**

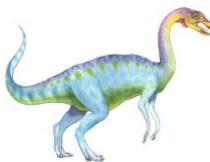
- But does the law take into account contemporary multicore systems?





# Amdahl's Law



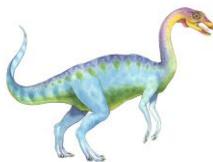


# User Threads and Kernel Threads

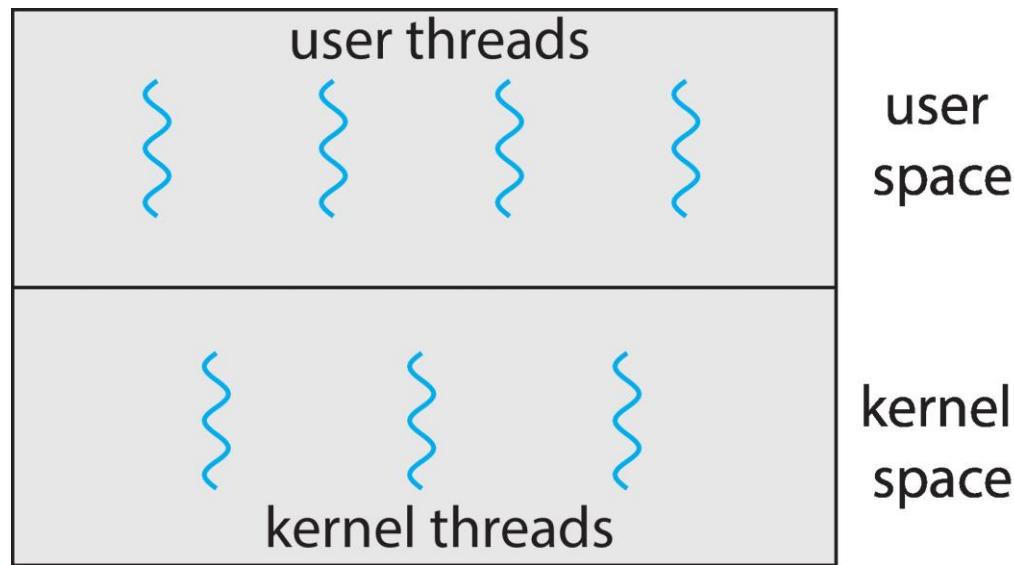
---

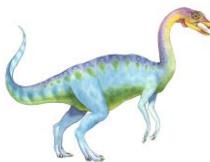
- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general-purpose OS, including:
  - Windows
  - Linux
  - Mac OS X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





# User and Kernel Threads



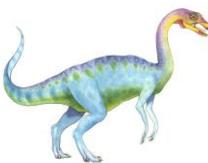


# Multithreading Models

---

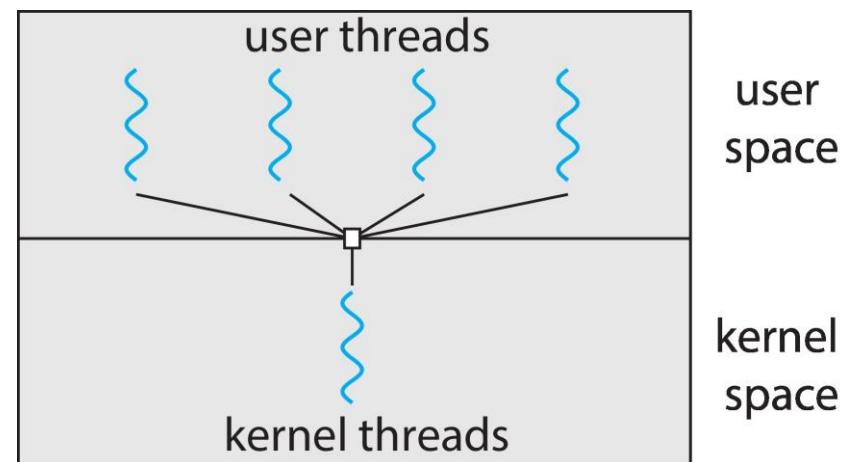
- Many-to-One
- One-to-One
- Many-to-Many

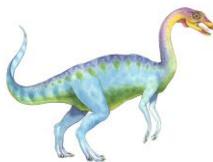




# Many-to-One

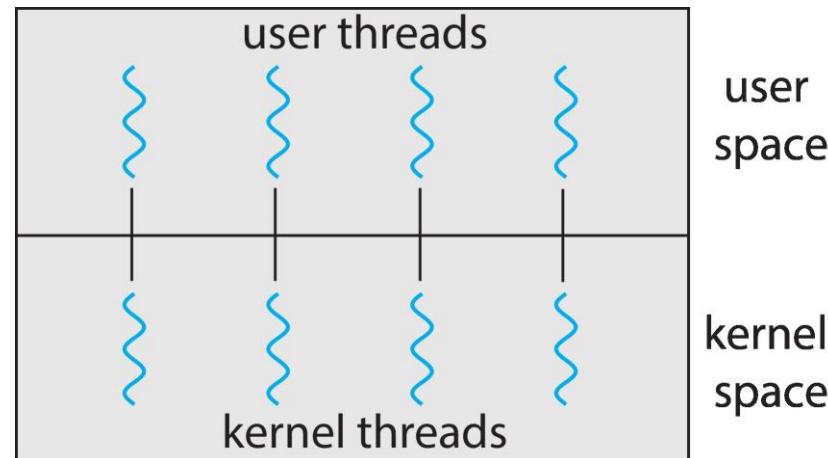
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- **Few** systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU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

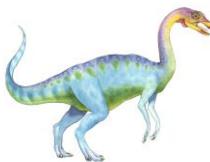




# One-to-One

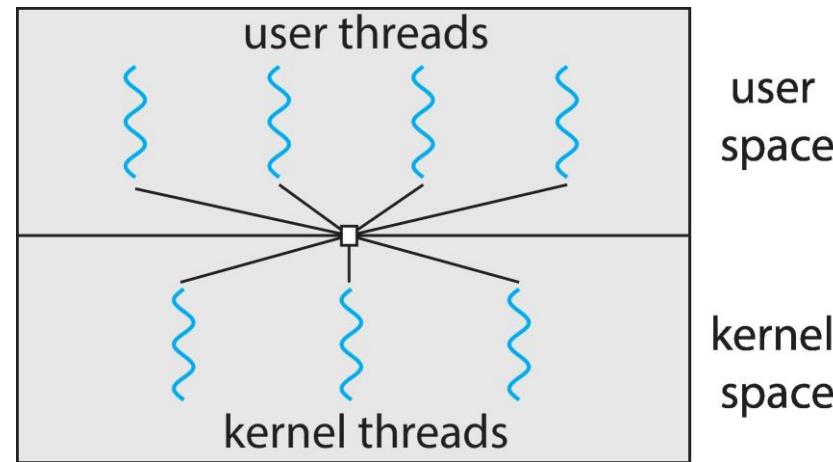
- Each user-level thread maps to a kernel thread
- Creating a user-level thread creates a kernel thread
- More **concurrency** than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux

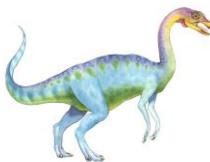




# Many-to-Many Model

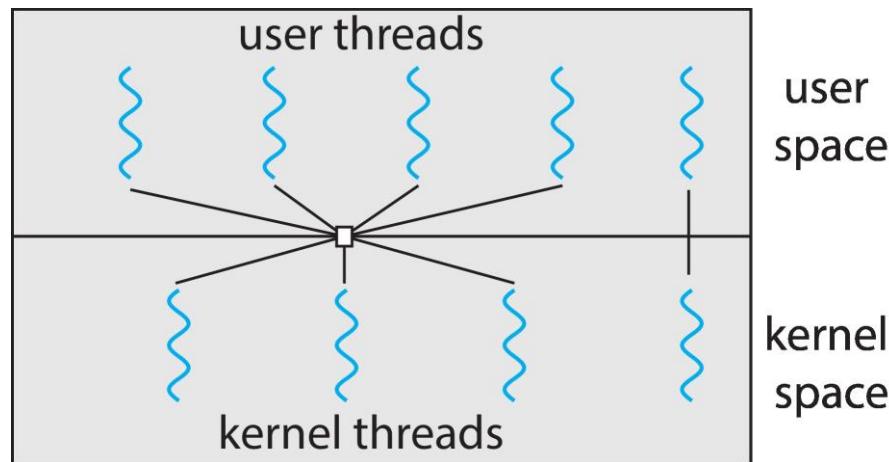
- Allows many user level threads to be mapped to many kernel threads
- Allows the OS to create a sufficient number of kernel threads
- Windows with the *ThreadFiber* package
- Otherwise not very common

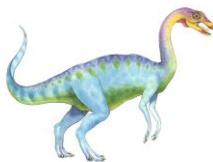




# Two-level Model

- 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



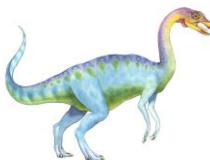


# Pthreads

---

- 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>

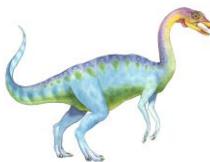
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

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);
}
```





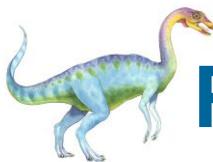
# Pthreads Example (Cont.)

```
/* 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);
}
```





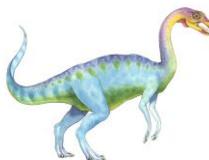
# Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

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

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





# Windows Multithreaded C Program

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

/* The thread 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;
}
```





# Windows Multithreaded C Program (Cont.)

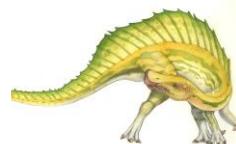
```
int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

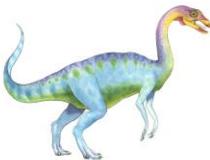
    Param = atoi(argv[1]);
    /* create the thread */
    ThreadHandle = CreateThread(
        NULL, /* default security attributes */
        0, /* default stack size */
        Summation, /* thread function */
        &Param, /* parameter to thread function */
        0, /* default creation flags */
        &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);
}
```





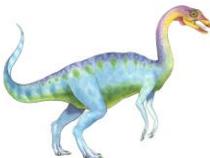
# Java Threads

- 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.");  
    }  
}
```

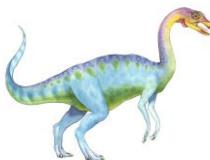
## Creating a thread:

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

## Waiting on a thread:

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





# Java Executor Framework

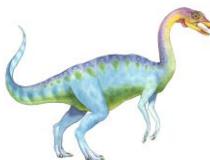
- Rather than explicitly creating threads, Java also allows thread creation around the Executor interface:

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

- The Executor is used as follows:

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



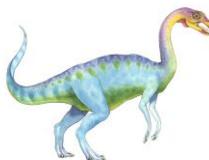


# Java Executor Framework

---

```
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);  
    }  
}
```





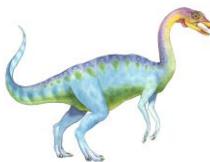
# Java Executor Framework (Cont.)

```
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



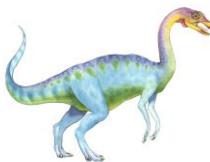


# Thread Pools

- 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.  
     */  
}
```

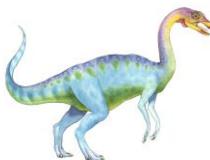




# Java Thread Pools

- Three factory methods for creating thread pools in Executors class:
  - static ExecutorService newSingleThreadExecutor()
  - static ExecutorService newFixedThreadPool(int size)
  - static ExecutorService newCachedThreadPool()





# Java Thread Pools (Cont.)

```
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());

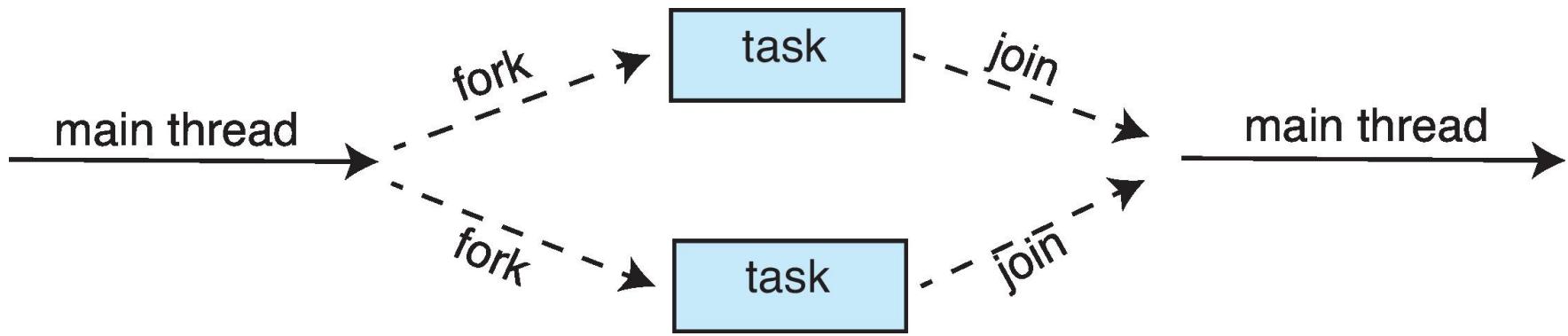
        /* Shut down the pool once all threads have completed */
        pool.shutdown();
    }
}
```

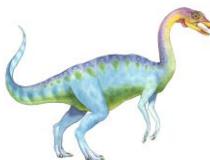




# Fork-Join Parallelism

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





# Fork-Join Parallelism

---

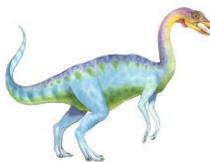
- General algorithm for fork-join strategy:

```
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)

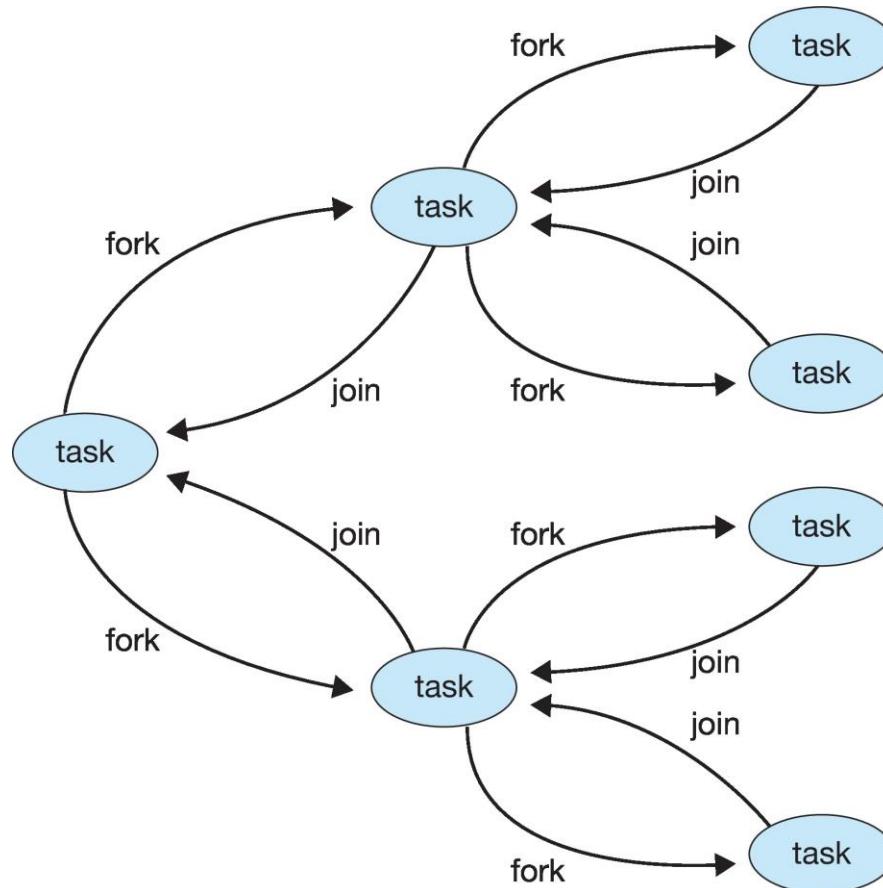
        result1 = join(subtask1)
        result2 = join(subtask2)

    return combined results
```





# Fork-Join Parallelism



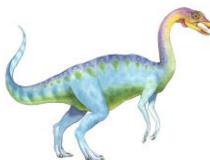


# Fork-Join Parallelism in Java

```
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);
```





# Fork-Join Parallelism in Java

```
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];

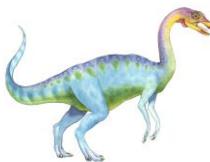
            return sum;
        }
        else {
            int mid = (begin + end) / 2;

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

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

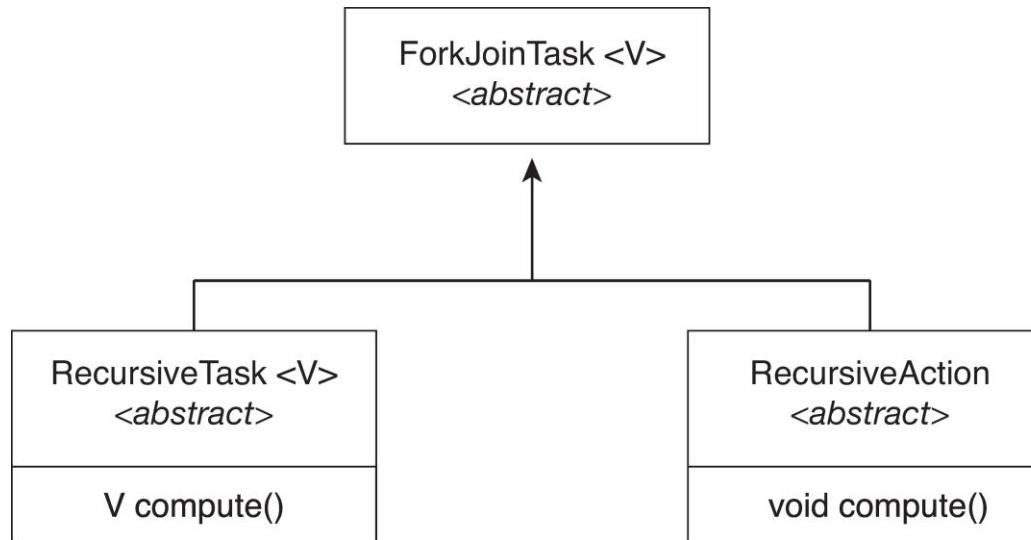
            return rightTask.join() + leftTask.join();
        }
    }
}
```

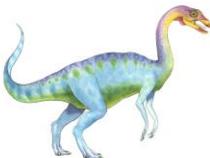




# Fork-Join Parallelism in Java

- 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

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

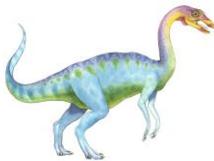
int main(int argc, char *argv[])
{
    /* sequential code */

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

    /* sequential code */

    return 0;
}
```

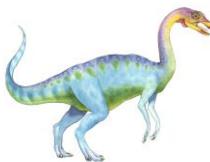




- Run the for loop in parallel

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

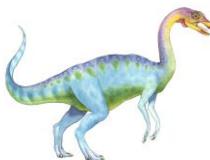




# Grand Central Dispatch

- 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

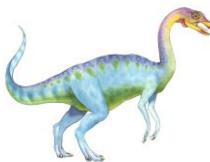




# Grand Central Dispatch

- 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





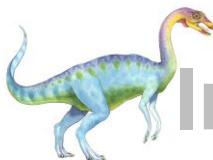
# Grand Central Dispatch

- 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)

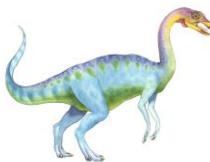
- Template library for designing parallel C++ programs
- A serial version of a simple for loop

```
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]);});
```

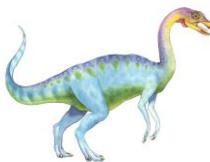




# Threading Issues

- 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

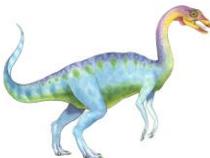




# Semantics of fork() and exec()

- 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

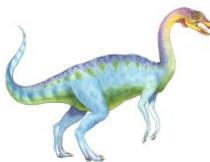




# Signal Handling

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

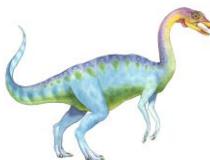




# Signal Handling (Cont.)

- 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



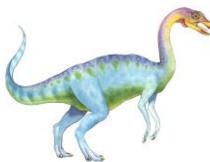


# Thread Cancellation

- 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);
```





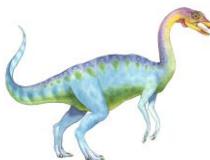
# Thread Cancellation (Cont.)

- 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

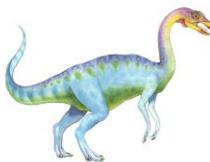
- Deferred cancellation uses the `interrupt()` method, which sets the interrupted status of a thread.

```
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()) {  
    . . .  
}
```

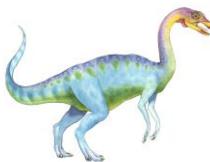




# Thread-Local Storage

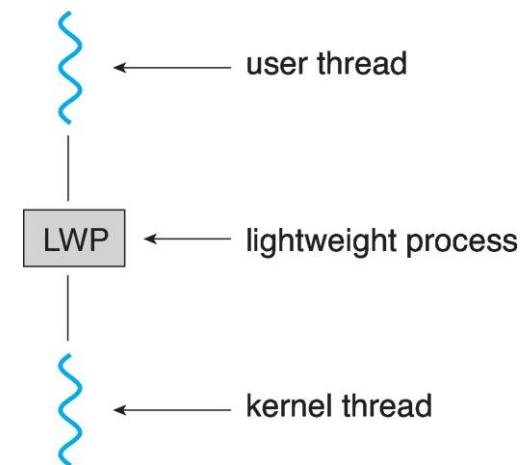
- **Thread-local storage (TLS)** allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible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

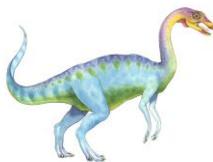




# Scheduler Activations

- 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

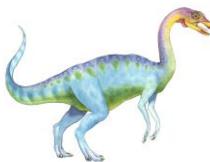




# Operating System Examples

- Windows Threads
- Linux Threads





# Windows Threads

- 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

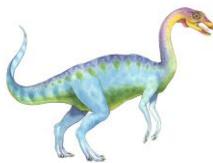




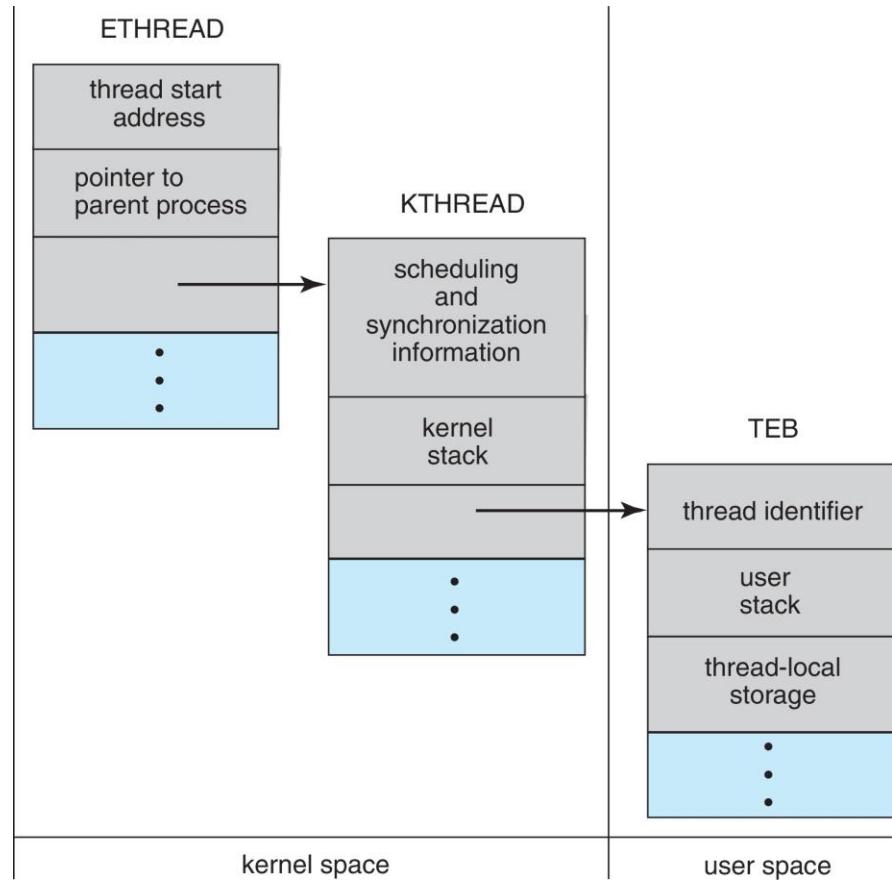
# Windows Threads (Cont.)

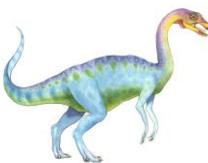
- The primary data structures of a thread include:
  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  -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





# Windows Threads Data Structures





# Linux Threads

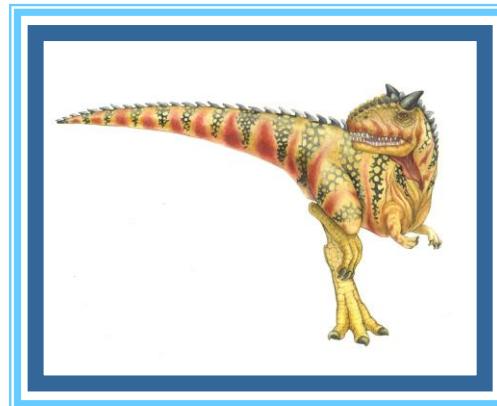
- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through **`clone()`** system call
- **`clone()`** allows a child task to share the address space of the parent task (process)
  - Flags control behavior

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

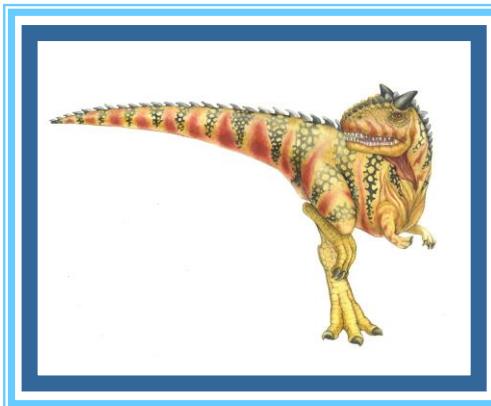
- **`struct task_struct`** points to process data structures (shared or unique)

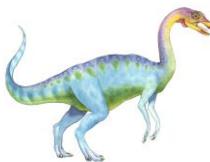


# End of Chapter 4



# Chapter 5: CPU Scheduling



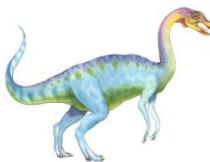


# Outline

---

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



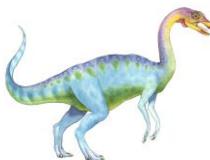


# Objectives

---

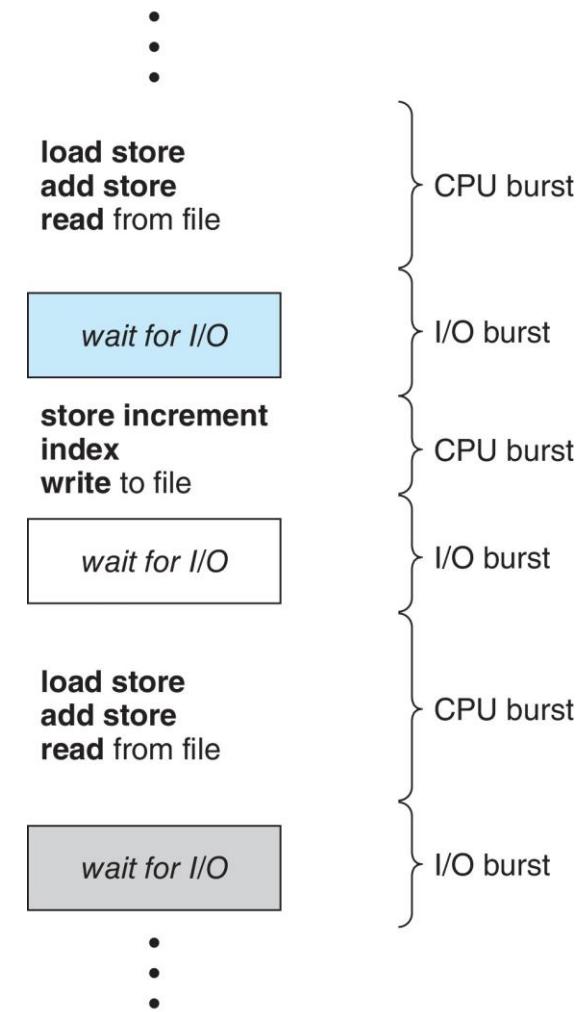
- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms
- Describe the scheduling algorithms used in the Windows, Linux, and 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



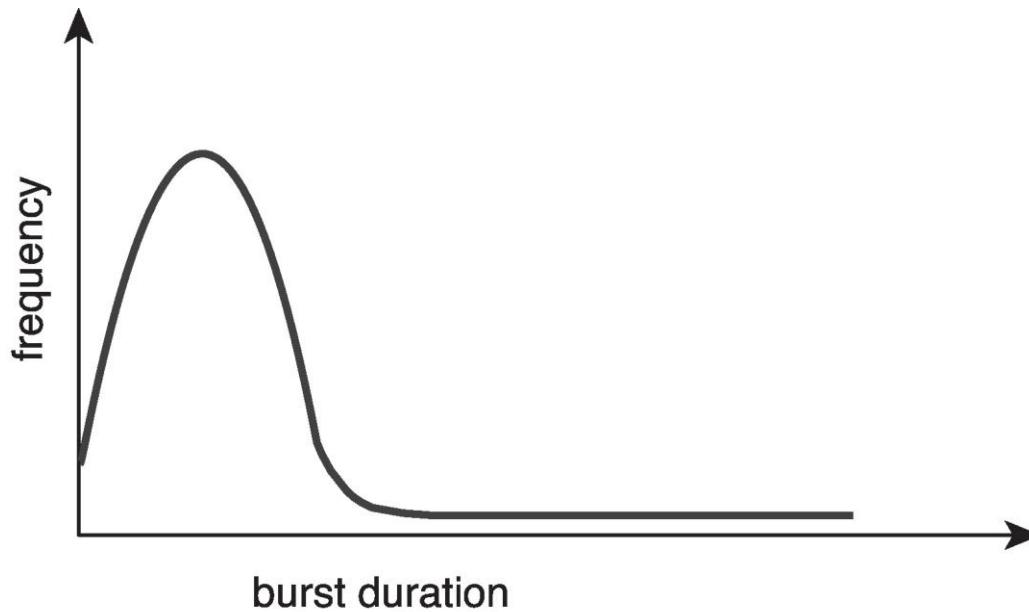


# Histogram of CPU-burst Times

---

Large number of short bursts

Small number of longer bursts

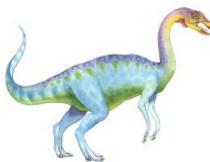




# CPU Scheduler

- The **CPU scheduler** selects from 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
- For situations 1 and 4, there is no choice in terms of scheduling
  - A new process (if one exists in the ready queue) must be selected for execution
- For situations 2 and 3, however, there is a choice

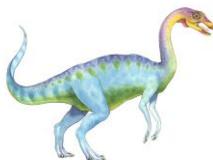




# Preemptive and Nonpreemptive Scheduling

- When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is **nonpreemptive**
  - Otherwise, it is **preemptive**
- Under Nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state
- Virtually all modern OS including Windows, MacOS, Linux, and UNIX use preemptive scheduling algorithms

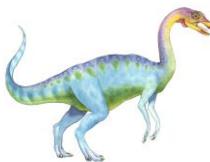




# Preemptive Scheduling and Race Conditions

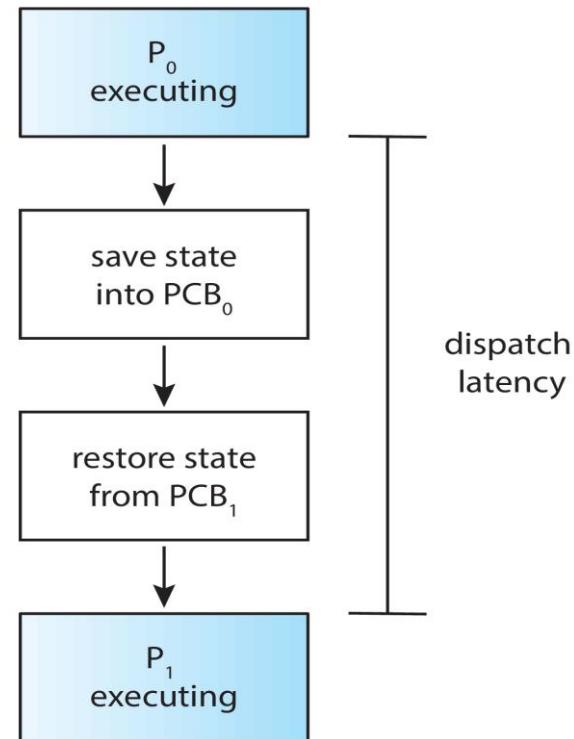
- Preemptive scheduling can result in race conditions when data are shared among several processes
- Consider the case of two processes that share data
  - While one process is updating the data, it is preempted so that the second process can run
  - The second process then tries to read the data, which are in an inconsistent state
- This issue will be explored in detail in Chapter 6

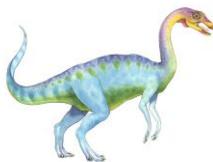




# Dispatcher

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



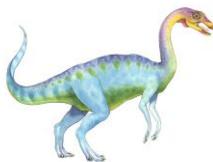


# Scheduling Criteria

---

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # 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



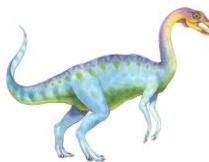


# Scheduling Algorithm Optimization Criteria

---

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





# First- Come, First-Served (FCFS) Scheduling

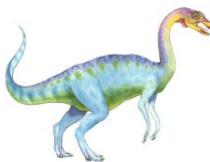
<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

- 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:

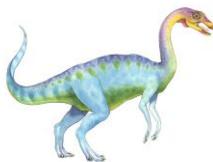
$$P_2, P_3, P_1$$

- 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

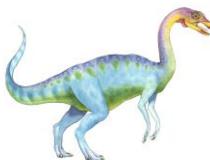




# Shortest-Job-First (SJF) Scheduling

- 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

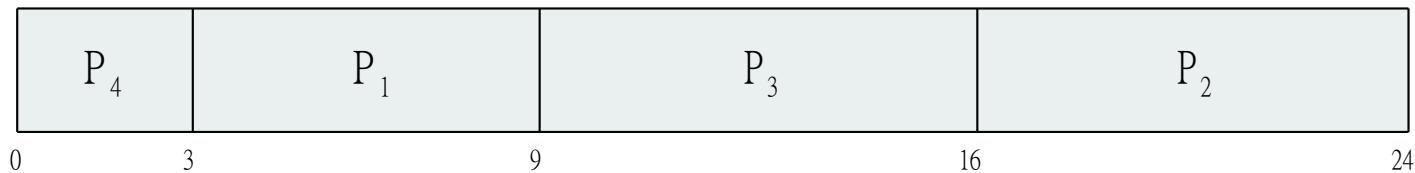




# Example of SJF

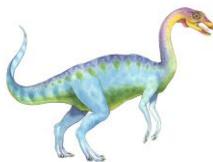
<u>Process</u>	<u>Burst Time</u>
$P_1$	6
$P_2$	8
$P_3$	7
$P_4$	3

- SJF scheduling chart



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





# Shortest-Job-First (SJF) Scheduling

---

- SJF is optimal – gives minimum average waiting time for a given set of processes
- Preemptive version called **shortest-remaining-time-first**
- How do we determine the length of the next CPU burst?
  - Could ask the user
  - Estimate



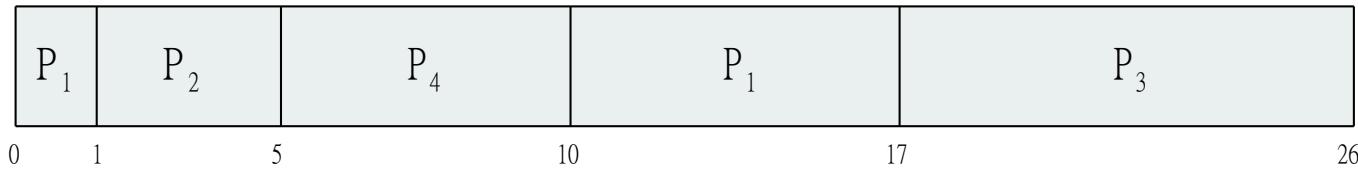


# Example of Shortest-remaining-time-first

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

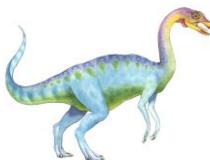
<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0	8
$P_2$	1	4
$P_3$	2	9
$P_4$	3	5

- Preemptive* SJF Gantt Chart



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

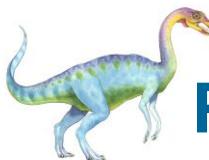




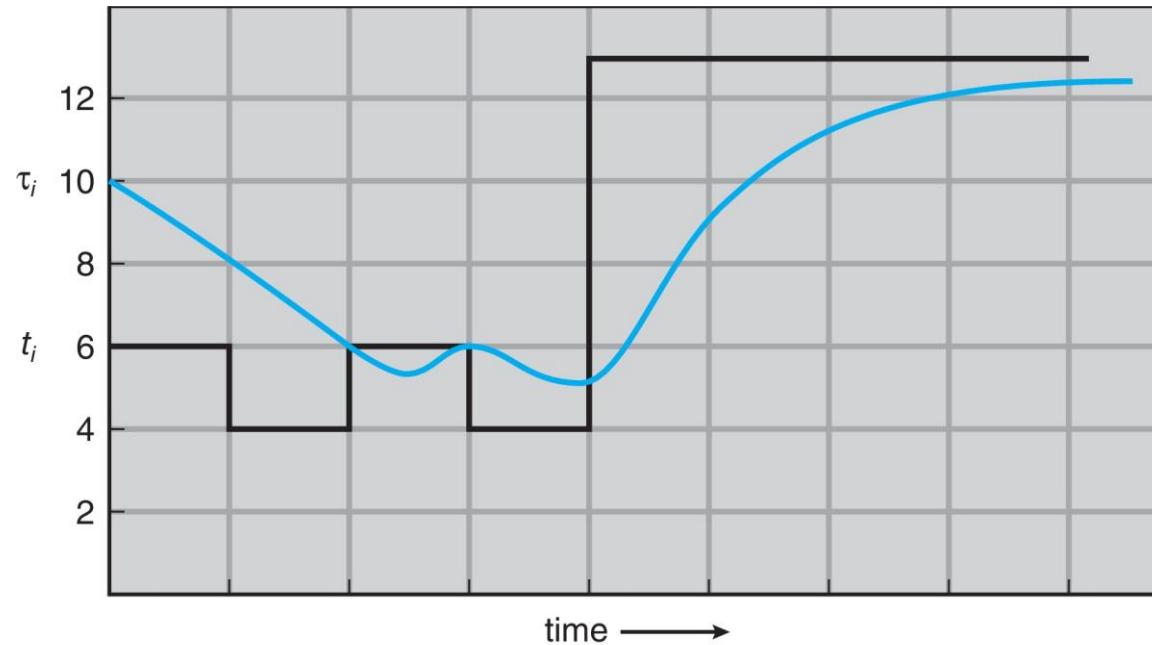
# Determining Length of Next CPU Burst

- 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.  $\tau_{n+1}$  = 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,  $\alpha$  set to  $\frac{1}{2}$





# Prediction of the Length of the Next CPU Burst



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





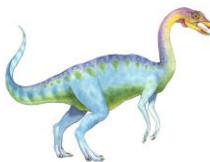
# Examples of Exponential Averaging

- $\alpha = 0$ 
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$ 
  - $\tau_{n+1} = t_n$
  - 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  $\alpha$  and  $(1 - \alpha)$  are less than or equal to 1, each successive term has less weight than its predecessor





# 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
  - 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

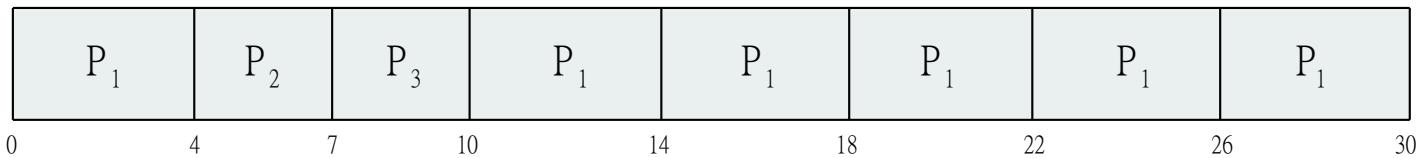




# Example of RR with Time Quantum = 4

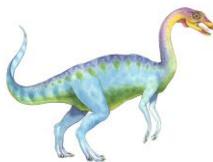
<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

- The Gantt chart is:

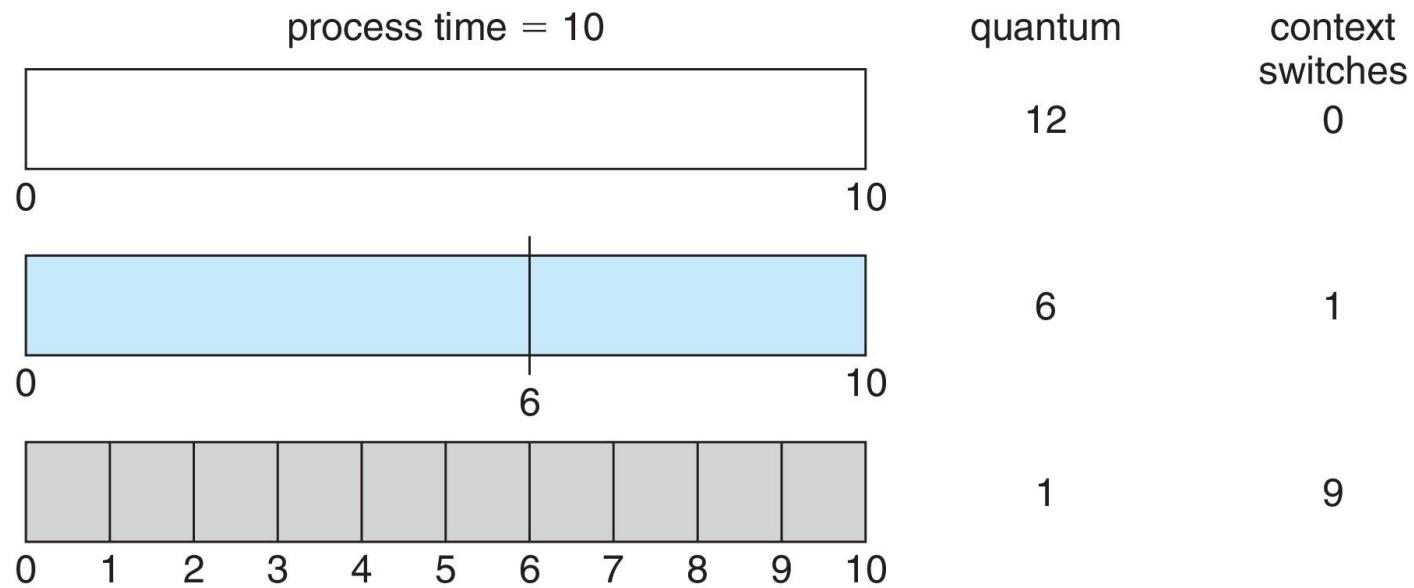


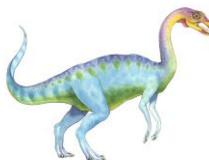
- Typically, higher average turnaround than SJF, but better **response**
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds



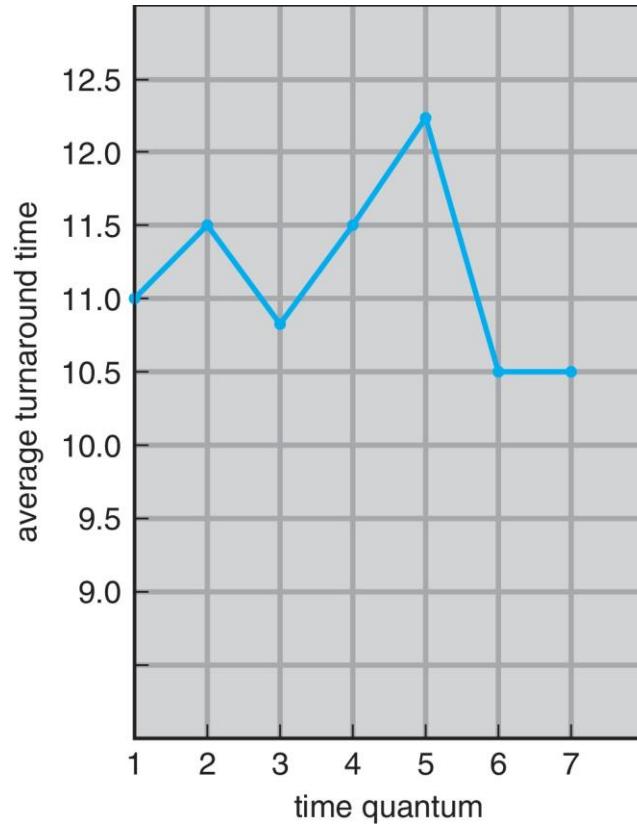


# Time Quantum and Context Switch Time





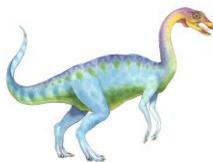
# Turnaround Time Varies With the Time Quantum



process	time
$P_1$	6
$P_2$	3
$P_3$	1
$P_4$	7

\* Rule of thumb:  
80% of CPU bursts  
should be shorter than  $q$

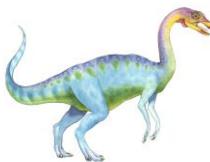




# Priority Scheduling

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

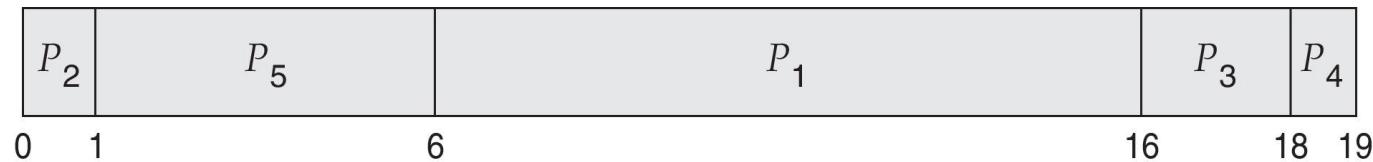




# Example of Priority Scheduling

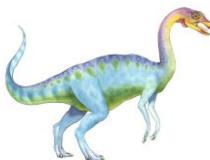
<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	10	3
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2

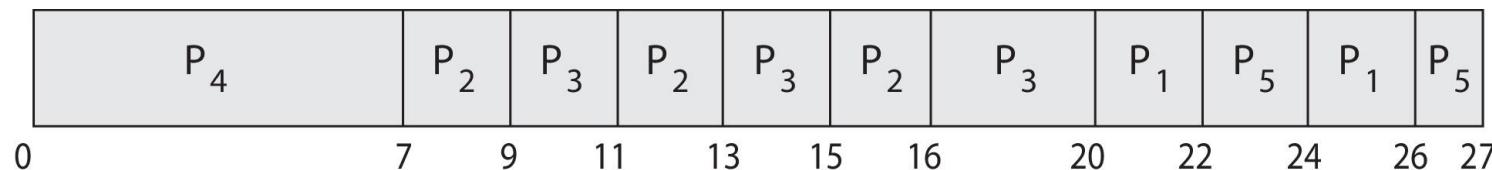




# Priority Scheduling w/ Round-Robin

<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

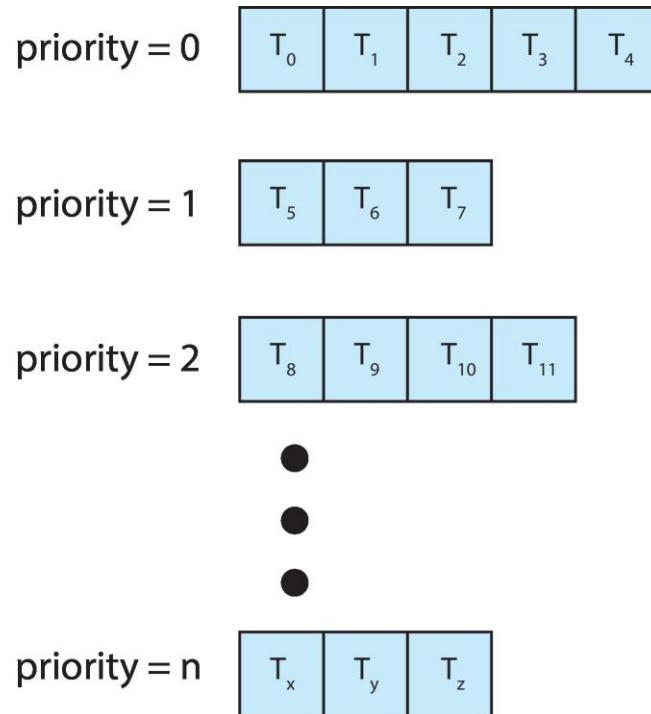
- Run the process with the highest priority. Processes with the same priority run round-robin
- Gantt Chart with time quantum = 2

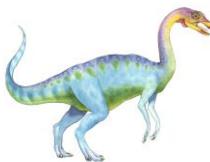




# Multilevel Queue

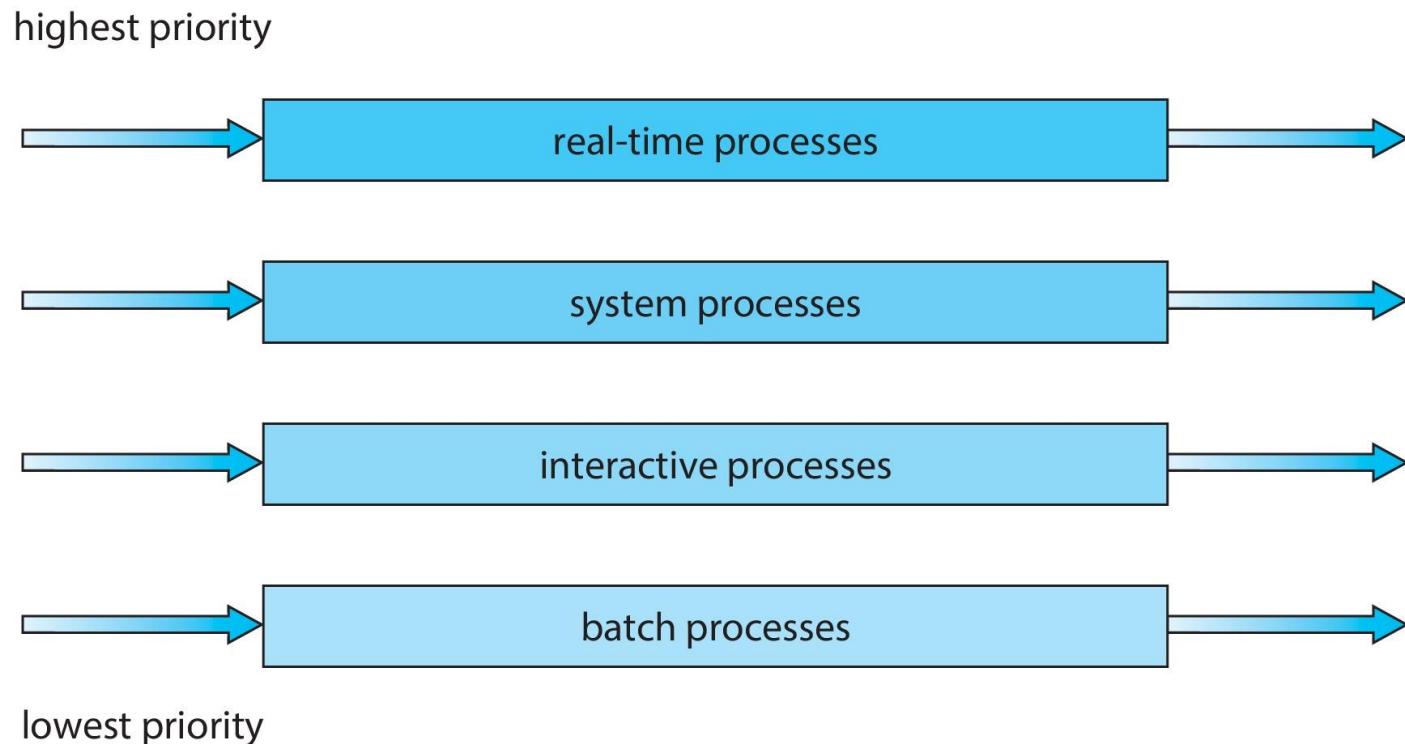
- With priority scheduling, have separate queues for each priority
- Schedule the process in the highest-priority queue!





# Multilevel Queue

- Prioritization based upon process type

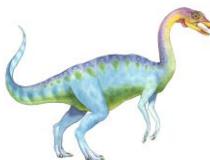




# Multilevel Feedback Queue

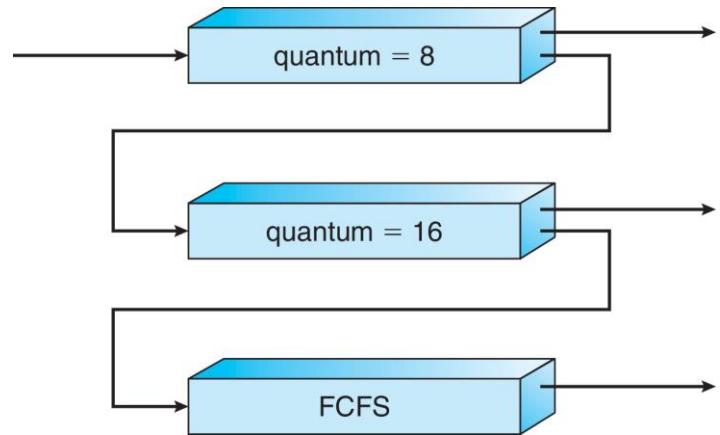
- A process can move between various queues
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to upgrade a process
  - Method used to determine when to demote a process
  - Method used to determine which queue a process will enter when that process needs service
- Aging can be implemented using multilevel feedback queue





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

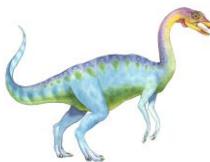




# Thread Scheduling

- 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

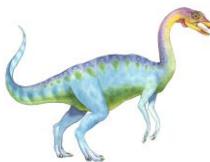




# Pthread Scheduling

- 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);
}
```



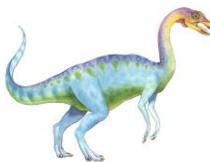


# Multiple-Processor Scheduling

---

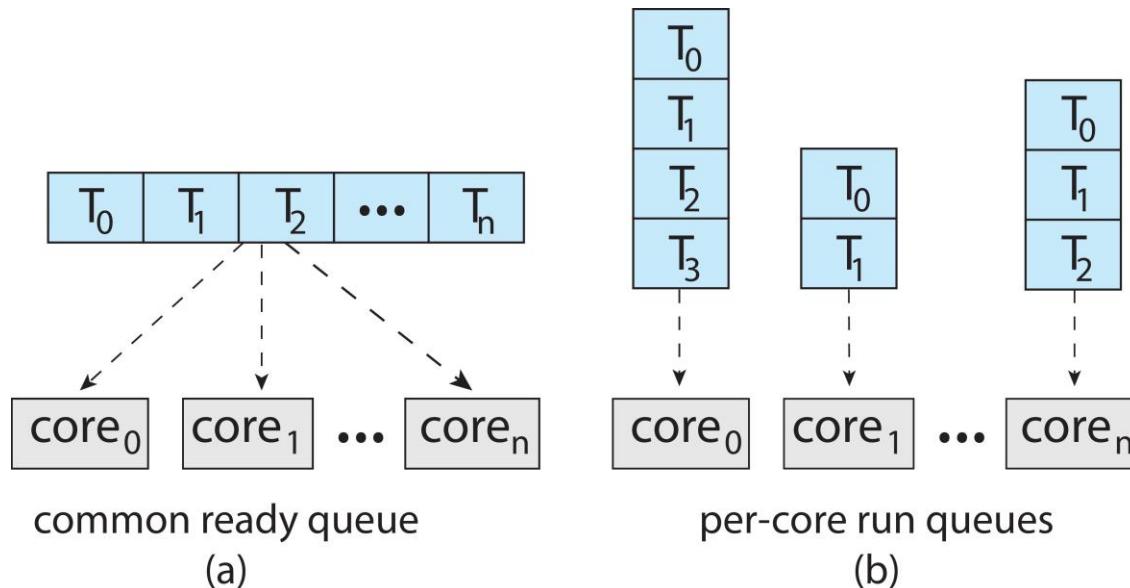
- CPU scheduling more complex when multiple CPUs are available
- Multiprocess may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems
  - Heterogeneous multiprocessing

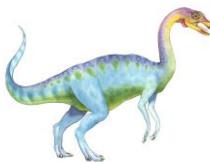




# Multiple-Processor Scheduling

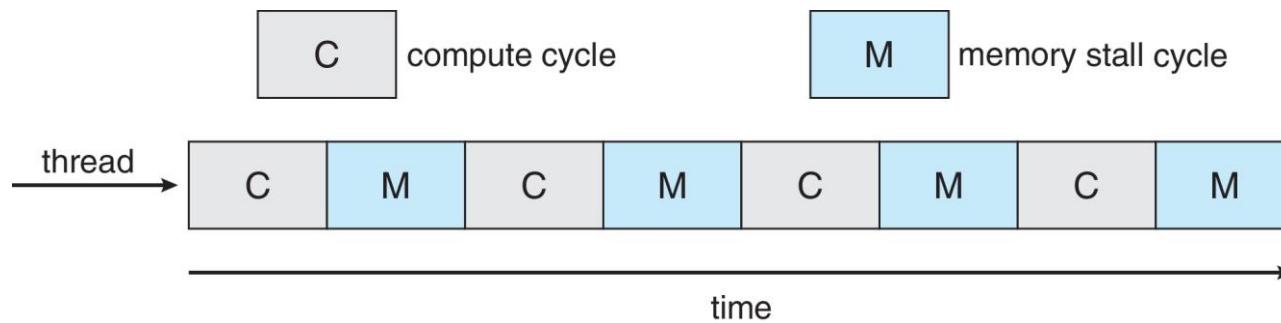
- Symmetric multiprocessing (SMP) is where each processor is self-scheduling
  - (a) All threads may be in a common ready queue
  - (b) Each processor may have its own private queue of threads

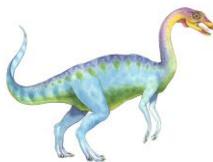




# Multicore Processors

- 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

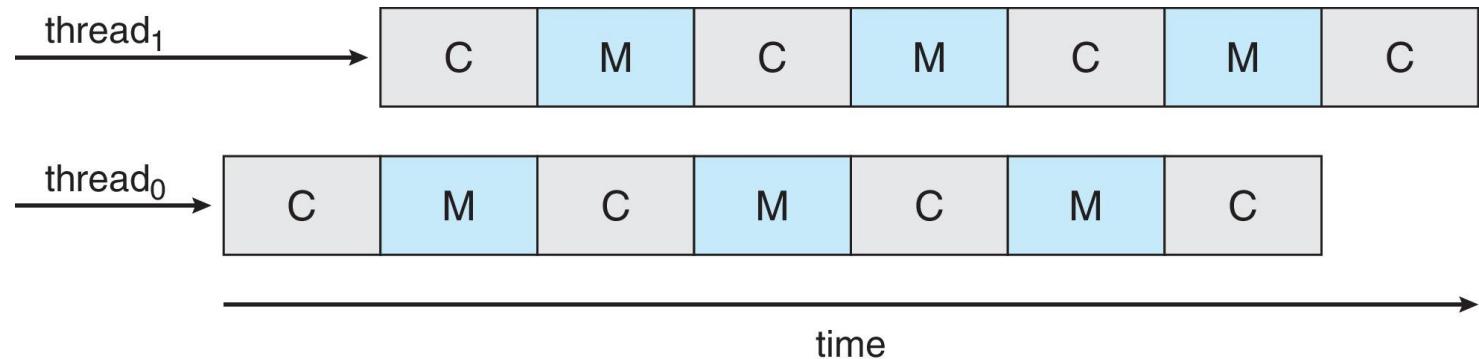


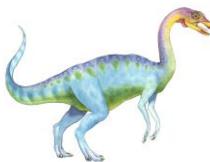


# Multithreaded Multicore System

Each core has > 1 hardware threads

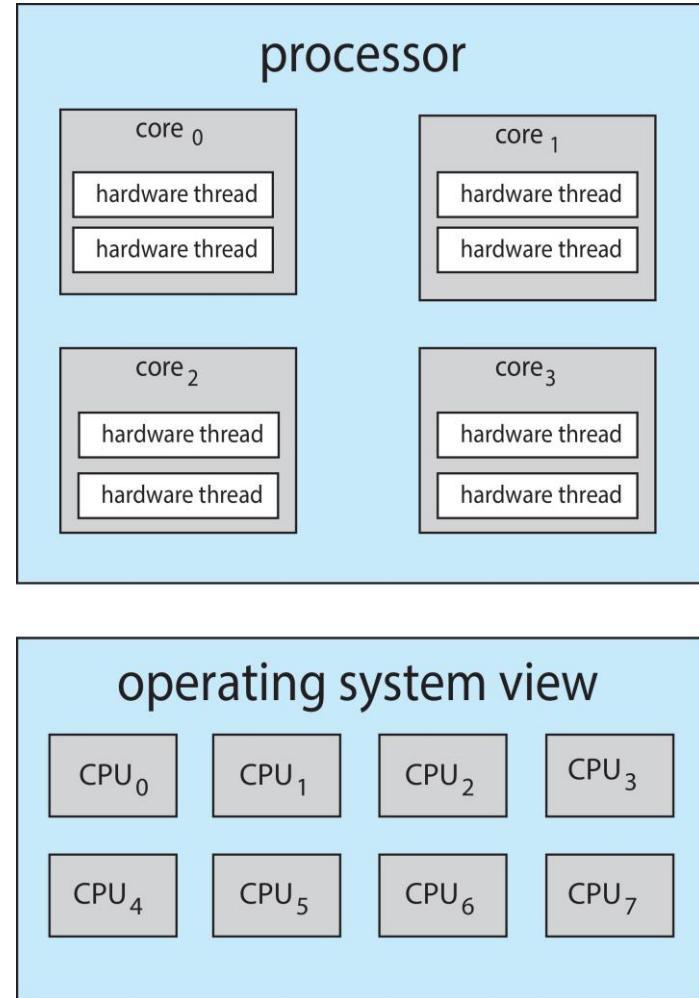
If one thread has a memory stall, switch to another thread!

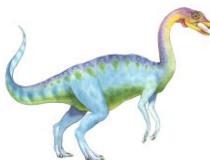




# Multithreaded Multicore System

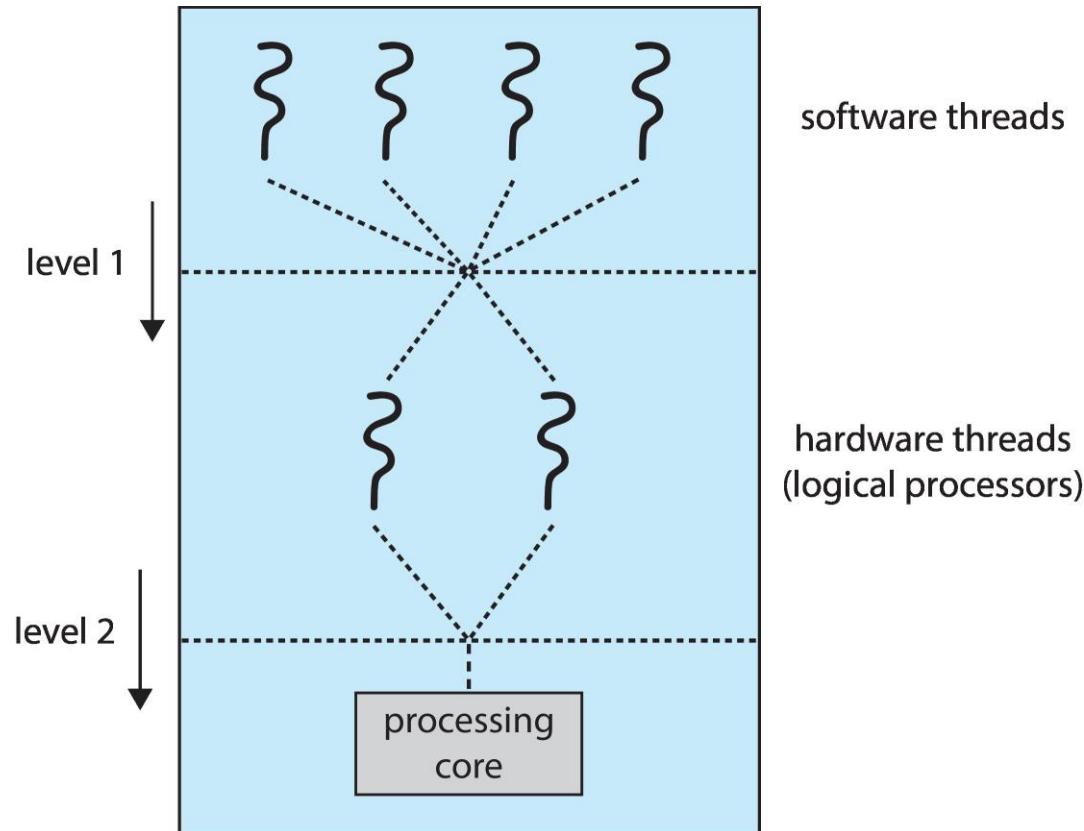
- **Chip-multithreading (CMT)** assigns each core multiple hardware threads (Intel refers to this as **hyperthreading**)
- On a quad-core system with 2 hardware threads per core, the OS sees 8 logical processors

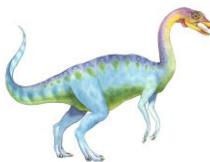




# Multithreaded Multicore System

- Two levels of scheduling:
  1. The OS deciding which software thread to run on a logical CPU
  2. How each core decides which hardware thread to run on the physical core

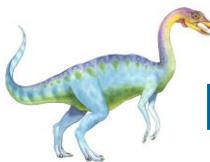




## Multiple-Processor Scheduling – Load Balancing

- 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

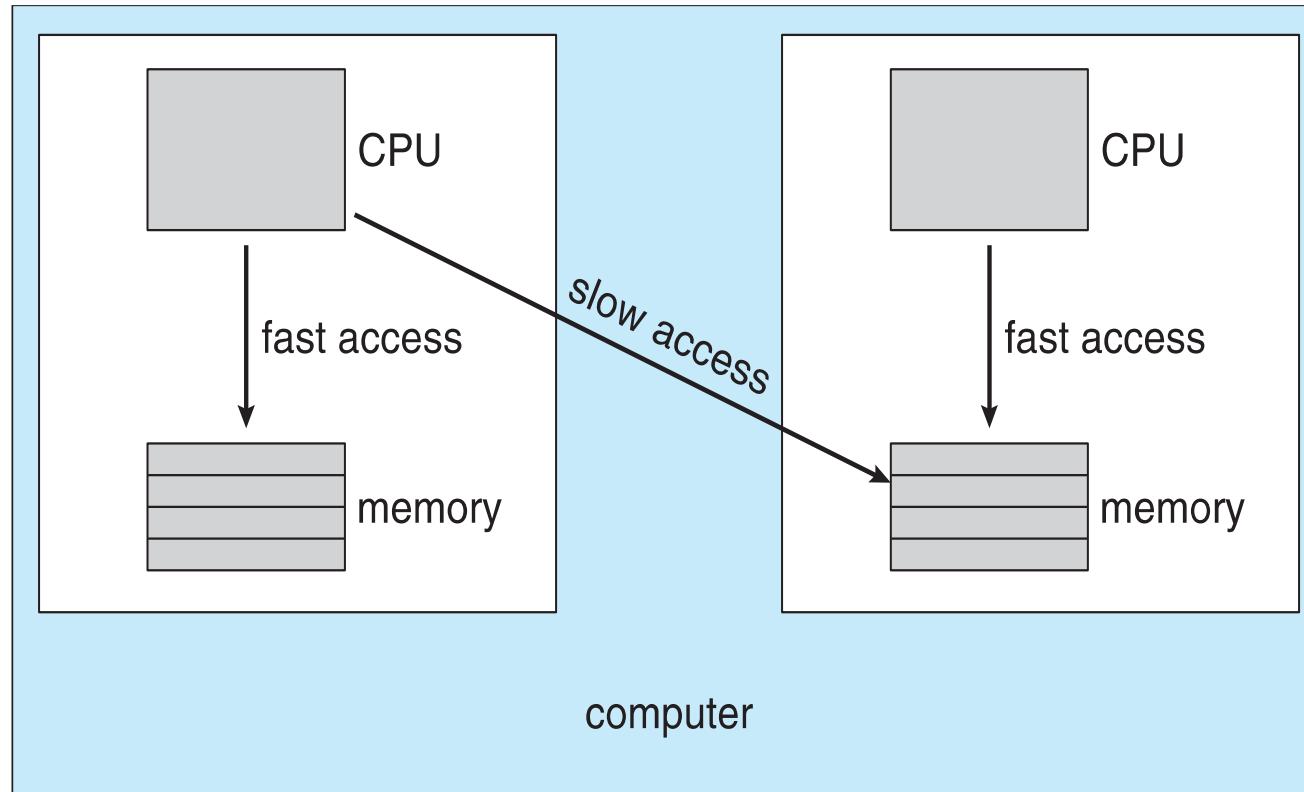
- When a thread has been running on one processor, the **cache** contents of that processor stores the memory accesses by that thread
  - Migrating the thread to another processor has high cost, and should be avoided
  - We refer to this as a thread having **affinity** for a processor (i.e., “processor affinity”)
- **Soft affinity** – the OS 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

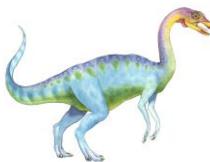




# NUMA and CPU Scheduling

If the OS is **NUMA-aware**, it will assign memory closest to the CPU the thread is running on



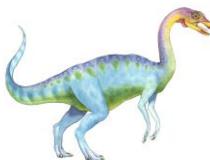


# Real-Time CPU Scheduling

---

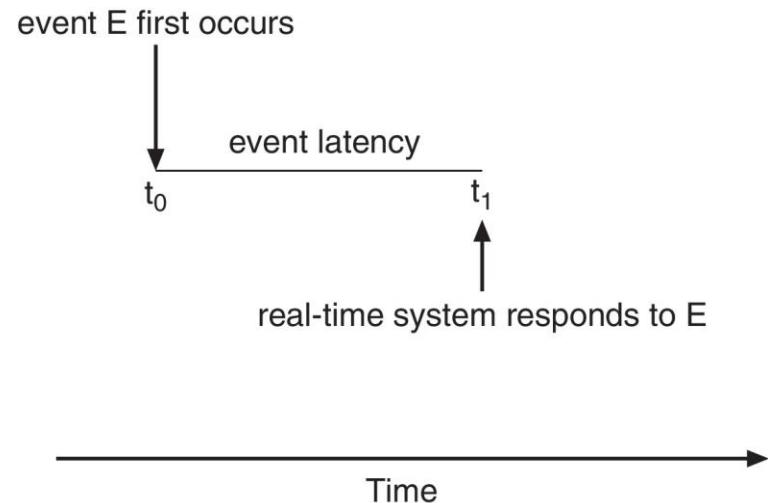
- Can present obvious challenges
- **Soft real-time systems** – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
- **Hard real-time systems – task must be serviced by its deadline**

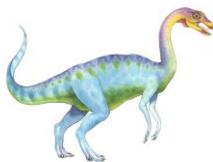




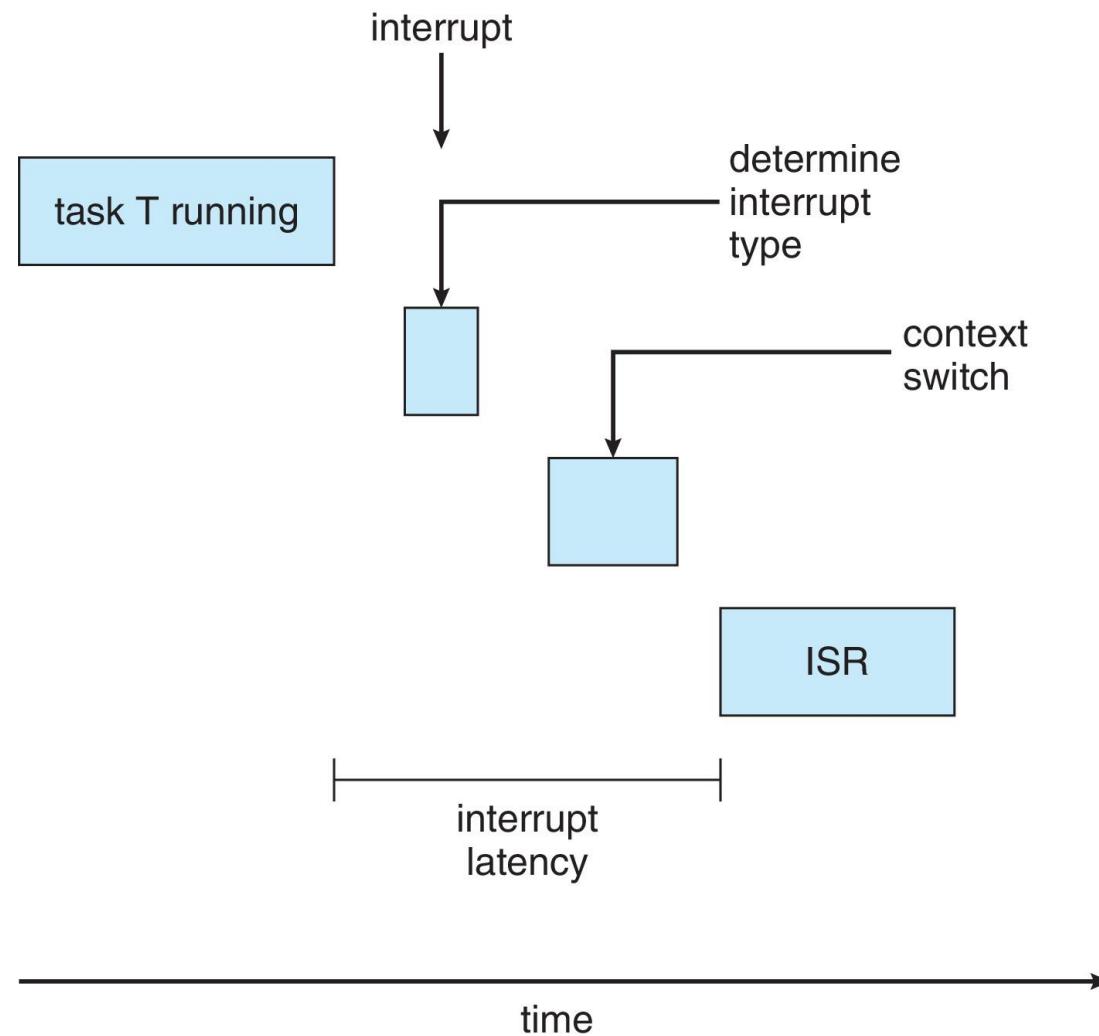
# Real-Time CPU Scheduling

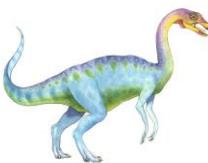
- Event latency – the amount of time that elapses from when an event occurs to when it is serviced
- Two types of latencies affect performance
  1. **Interrupt latency** – time from arrival of interrupt to start of routine that services interrupt
  2. **Dispatch latency** – time for scheduler to take current process off CPU and switch to another





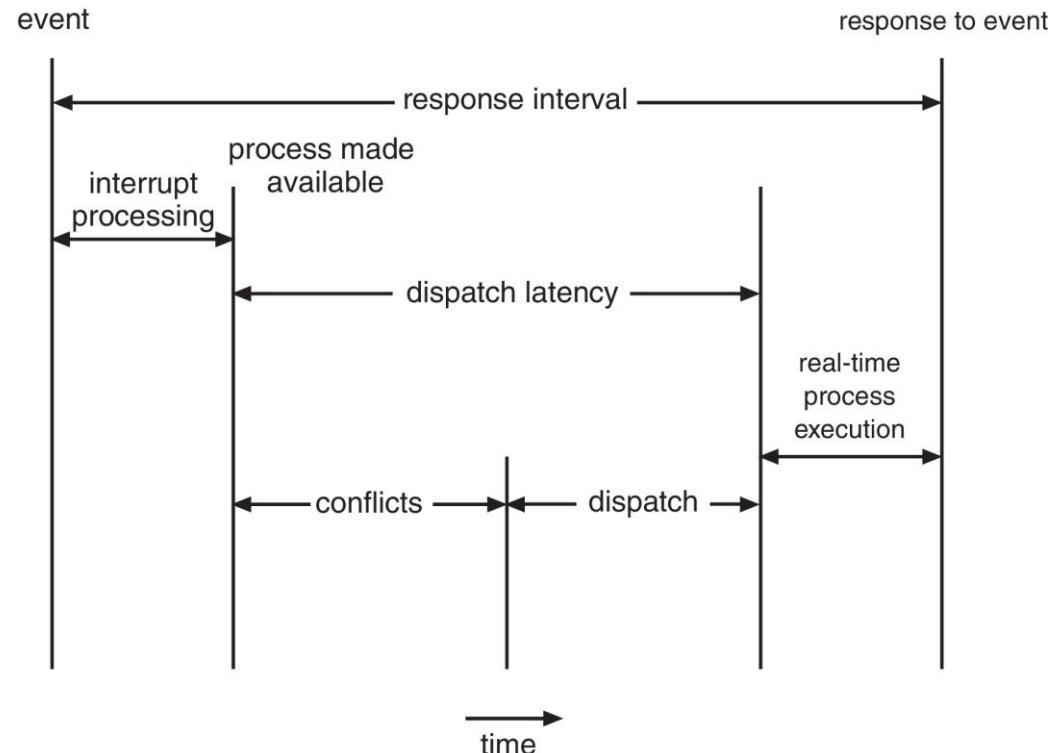
# Interrupt Latency

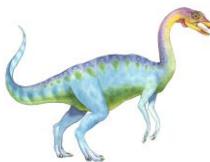




# Dispatch Latency

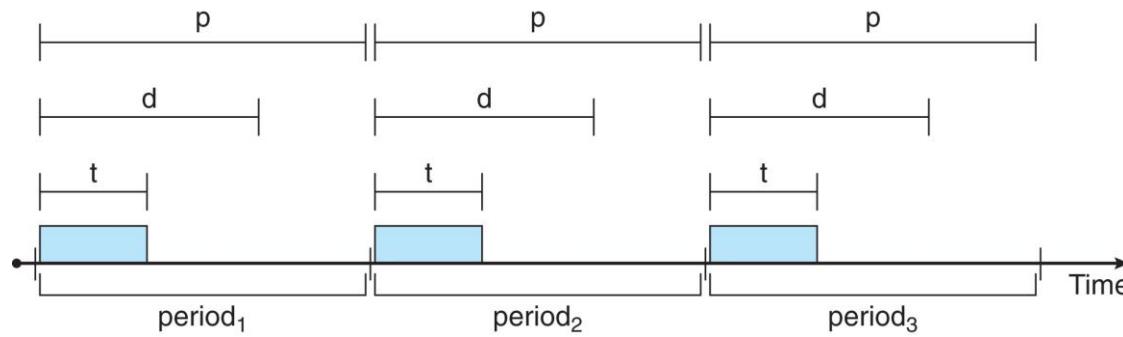
- Conflict phase of dispatch latency:
  - Preemption of any process running in kernel mode
  - Release by low-priority process of resources needed by high-priority processes

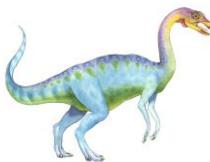




# Priority-based Scheduling

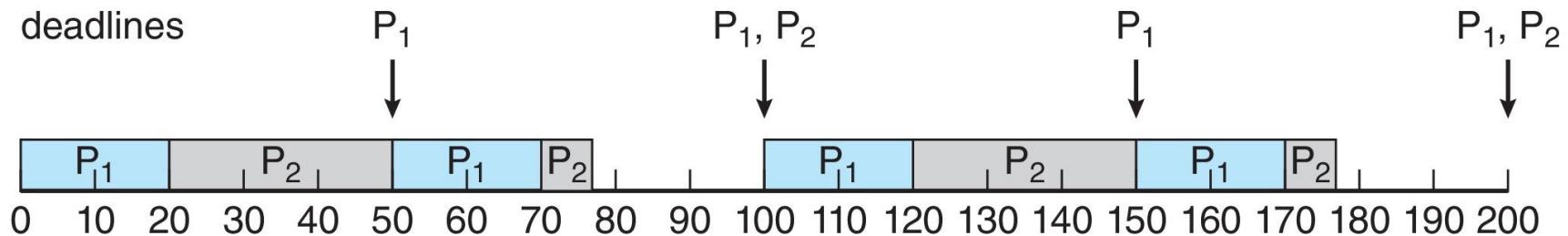
- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing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

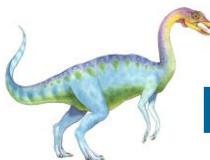




# Rate Monotonic Scheduling

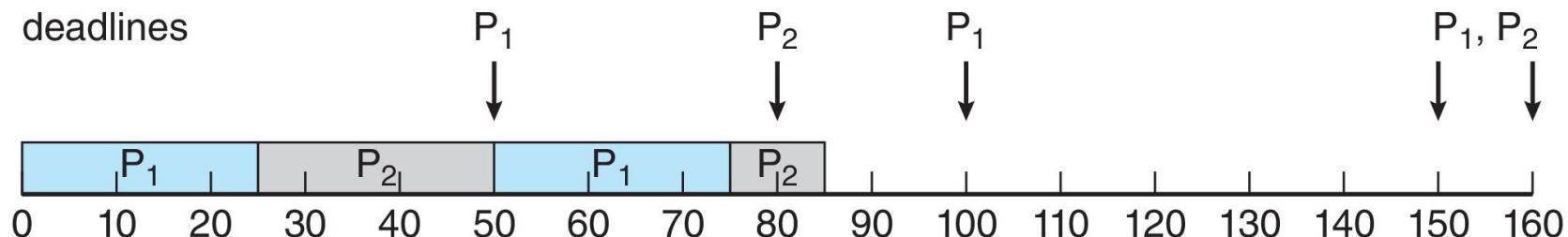
- A priority is assigned based on the inverse of its period
  - Shorter periods = higher priority
  - Longer periods = lower priority
- $P_1$  is assigned a higher priority than  $P_2$





# Missed Deadlines with Rate Monotonic Scheduling

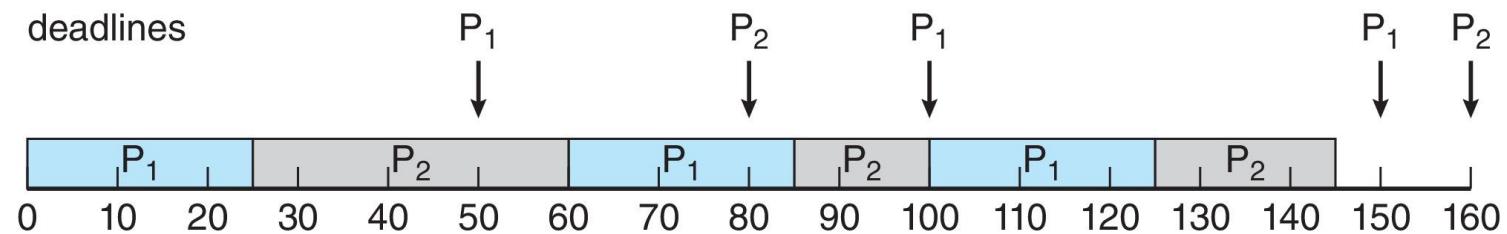
Process P2 misses finishing its deadline at time 80

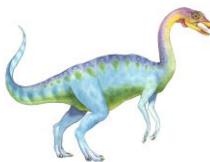




# Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - the earlier the deadline, the higher the priority
  - the later the deadline, the lower the priority

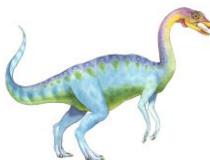




# Proportional Share Scheduling

- $T$  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



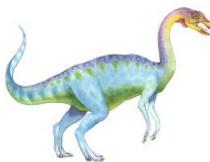


# POSIX Real-Time Scheduling

---

- 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.)

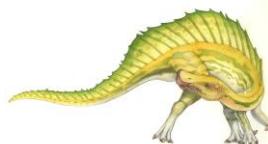
```
/* set the scheduling policy - FIFO, RR, or OTHER */
if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
    fprintf(stderr, "Unable to set policy.\n");

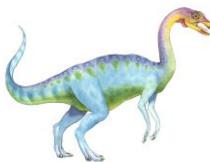
/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);

/* 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);
}
```



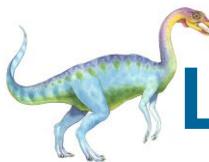


# Operating System Examples

---

- Linux scheduling
- Windows scheduling
- Solaris scheduling

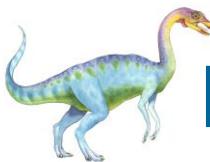




# Linux Scheduling Through Version 2.5

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





# Linux Scheduling in Version 2.6.23 +

- ***Completely Fair Scheduler* (CFS)**
- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - Two scheduling classes included, others can be added
    1. default
    2. real-time





# Linux Scheduling in Version 2.6.23 + (Cont.)

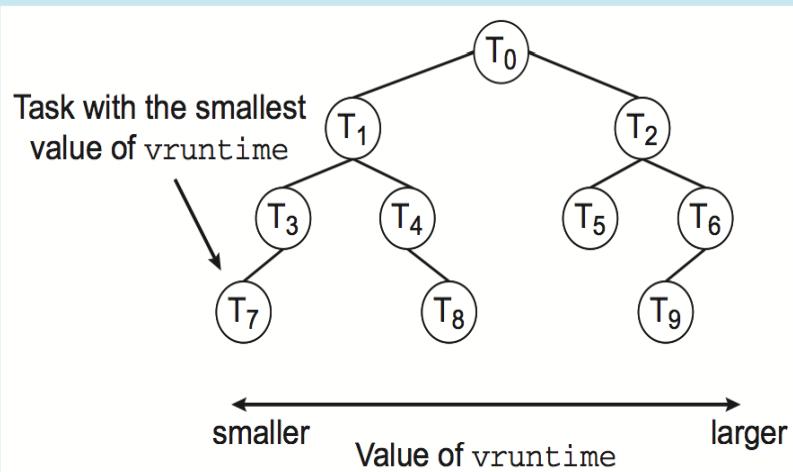
- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual run time** in variable **vruntime**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with **lowest virtual run time**





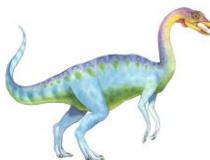
# CFS Performance

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



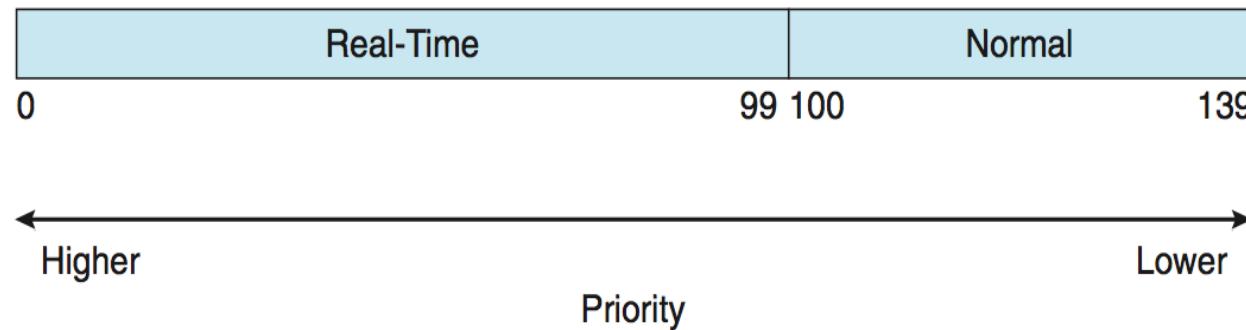
When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and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

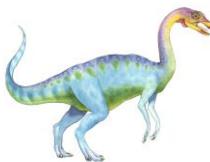




# Linux Scheduling (Cont.)

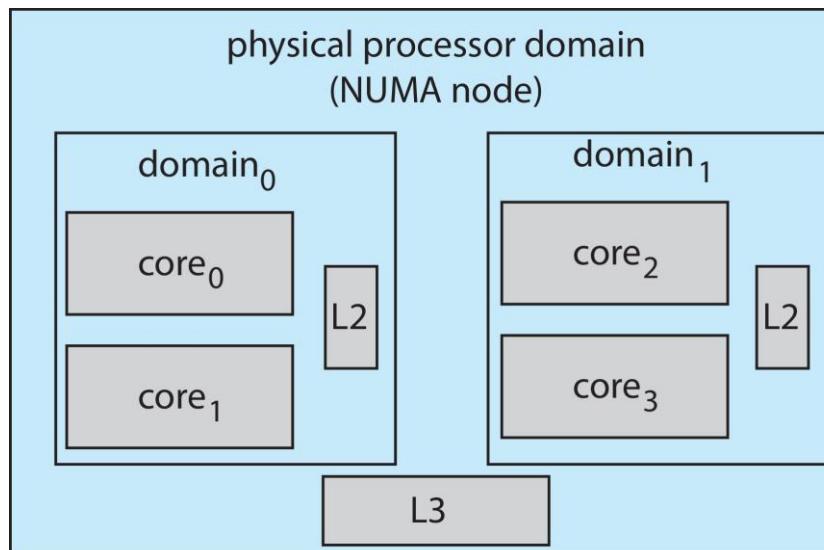
- Real-time scheduling according to POSIX.1b
    - Real-time tasks have static priorities
  - Real-time plus normal map into global priority scheme
    - Nice value of -20 maps to global priority 100
    - Nice value of +19 maps to priority 139

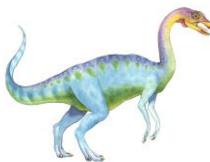




# Linux Scheduling (Cont.)

- Linux supports load balancing, but is also NUMA-aware
- **Scheduling domain** is a set of CPU cores that can be balanced against one another
- Domains are organized by what they share (i.e., cache memory)  
Goal is to keep threads from migrating between domains



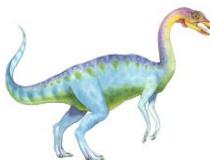


# Windows Scheduling

---

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



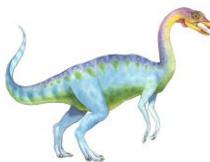


# Windows Priority Classes

---

- Win32 API identifies six **priority classes** to which a process can belong
  - REALTIME\_PRIORITY\_CLASS, HIGH\_PRIORITY\_CLASS, ABOVE\_NORMAL\_PRIORITY\_CLASS, NORMAL\_PRIORITY\_CLASS, BELOW\_NORMAL\_PRIORITY\_CLASS, IDLE\_PRIORITY\_CLASS
  - All are variable except REALTIME
- A thread within a given priority class has a **relative priority**
  - TIME\_CRITICAL, HIGHEST, ABOVE\_NORMAL, NORMAL, BELOW\_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
  - **Base** priority is NORMAL within the class
  - If quantum expires, priority lowered, but never below base





# Windows Priority Classes (Cont.)

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

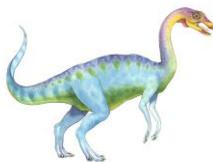




# Windows Priorities

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

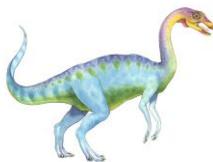




# Solaris

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



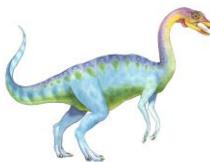


# Solaris Dispatch Table

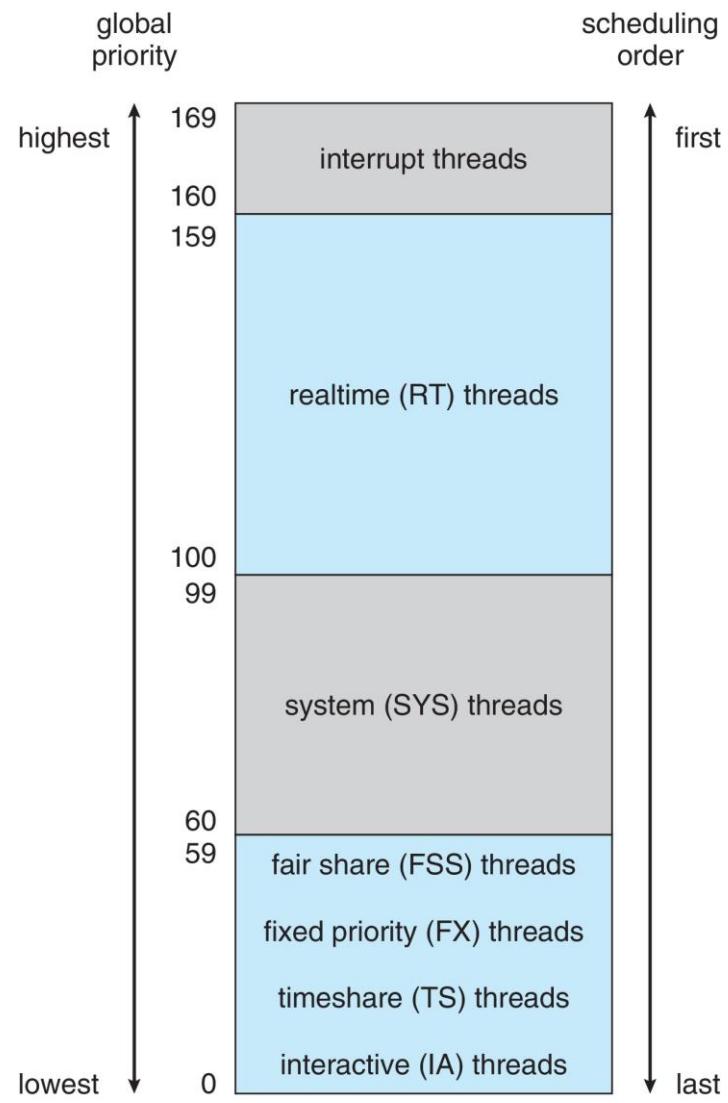
low

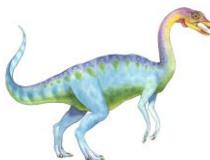
priority	time quantum	time quantum expired	return from sleep
low	0	200	0
	5	200	0
	10	160	0
	15	160	5
	20	120	10
	25	120	15
	30	80	20
	35	80	25
	40	40	30
	45	40	35
	50	40	40
	55	40	45
high	59	20	49





# Solaris Scheduling

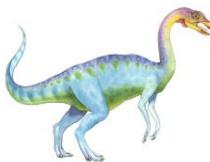




# Solaris Scheduling (Cont.)

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





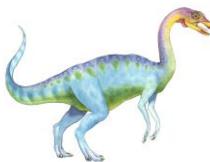
# Algorithm Evaluation

---

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

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



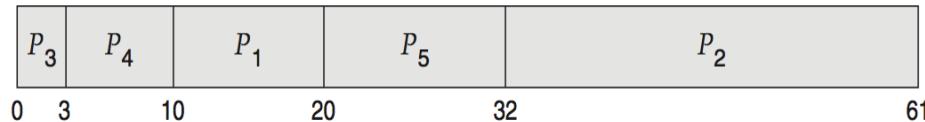


# Deterministic Evaluation

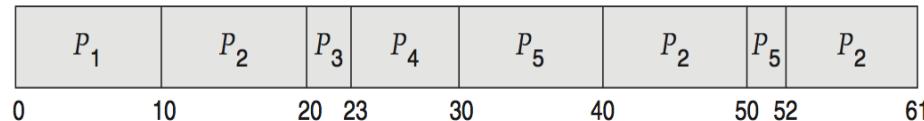
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:

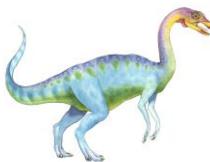


- Non-preemptive SJF is 13ms:



- RR is 23ms:

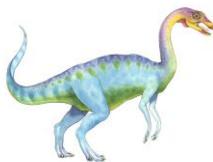




# Queueing Models

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



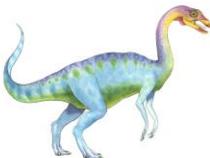


# Little's Formula

---

- $n$  = average queue length
- $W$  = average waiting time in queue
- $\lambda$  = 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





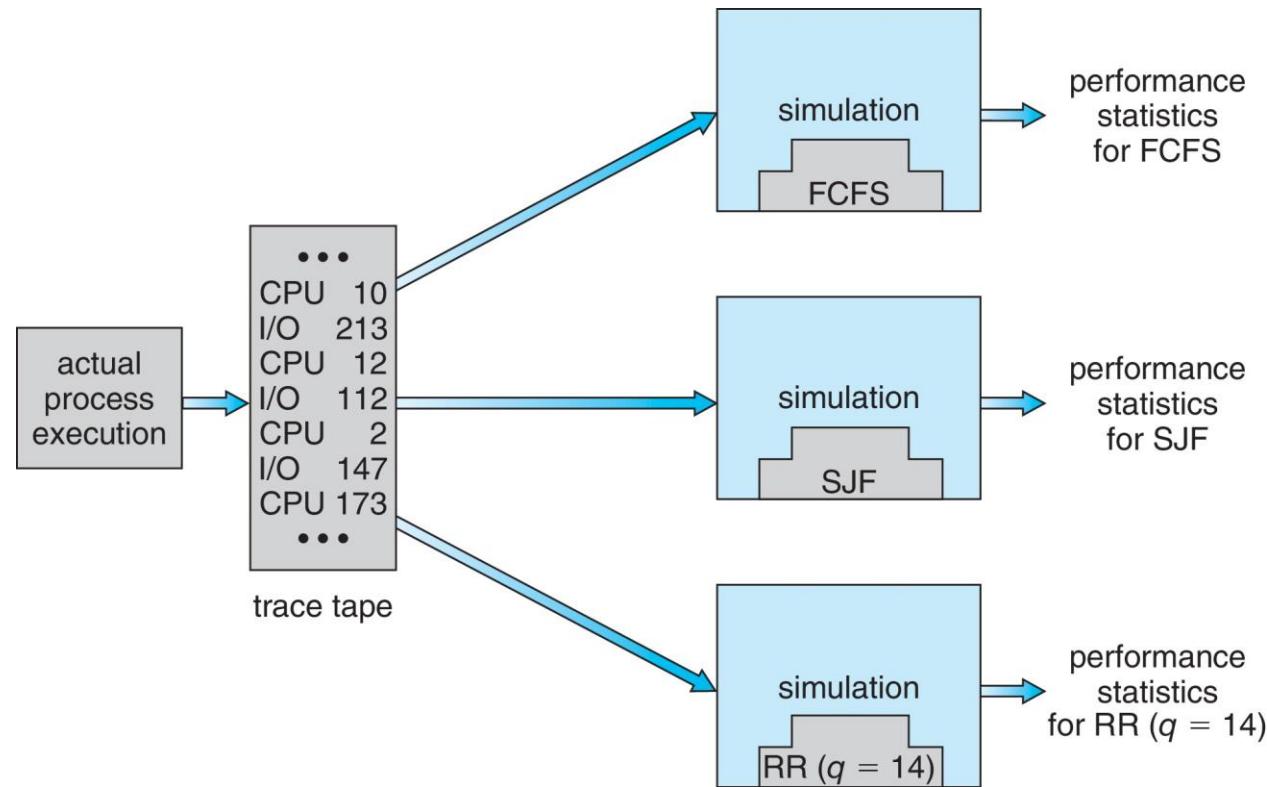
# Simulations

- 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





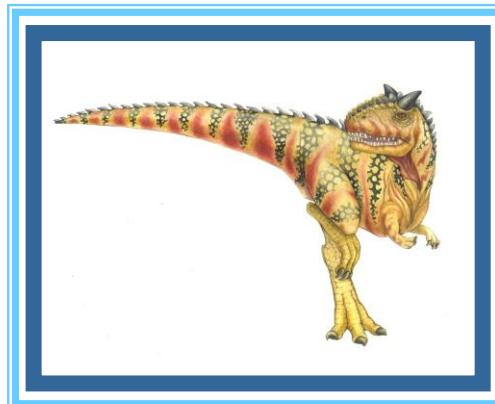
# Implementation

---

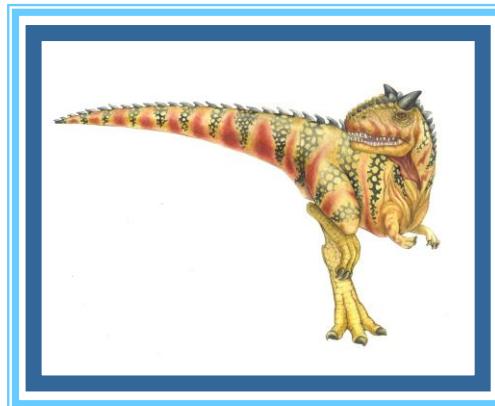
- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary

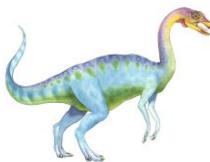


# End of Chapter 5



# Chapter 6: Synchronization Tools





# Outline

---

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- Liveness



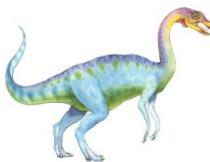


# Objectives

---

- 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





# Background

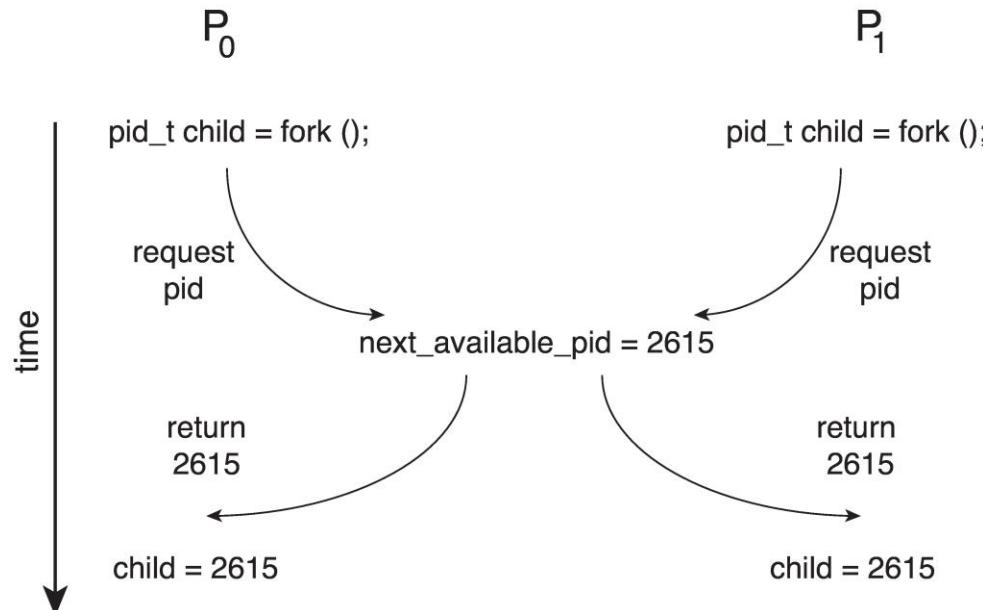
- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining **data consistency** requires mechanisms to ensure the orderly execution of cooperating processes
- We illustrated the problem in Chapter 4 when we considered the Bounded Buffer problem with use of a counter that is updated concurrently by the producer and consumer, which leads to race condition





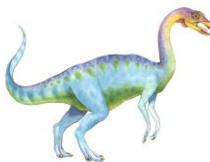
# Race Condition

- 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 a mechanism to prevent  $P_0$  and  $P_1$  from accessing the variable `next_available_pid` the same pid could be assigned to two different processes!

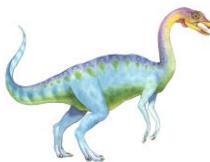




# Critical Section Problem

- Consider system of  $n$  processes  $\{p_0, p_1, \dots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
  - Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**





# Critical Section

---

- General structure of process  $P_i$

```
do {
```

```
    entry section
```

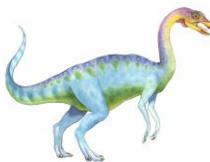
```
    critical section
```

```
    exit section
```

```
    remainder section
```

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





# Critical-Section Problem (Cont.)

Requirements for solution to critical-section problem

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





# Interrupt-based Solution

---

- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?
  - What if the critical section is code that runs for an hour?
  - Can some processes starve – never enter their critical section
  - What if there are two CPUs?



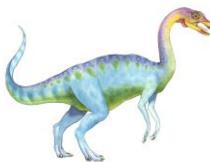


# Software Solution 1

---

- Two-process solution
- Assume that the **load** and **store** machine-language instructions are **atomic**; that is, cannot be interrupted
- The two processes share one variable:
  - **int turn;**
- The variable **turn** indicates whose turn it is to enter the critical section





# Algorithm for Process $P_i$

```
while (true) {
```

```
    turn = i;  
    while (turn == j)  
        ;
```

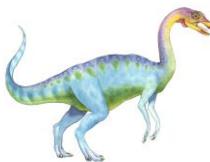
```
    /* critical section */
```

```
    turn = j;
```

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

```
}
```



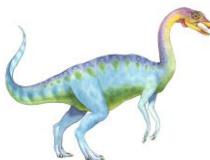


# Algorithm for Process $P_i$ and $P_j$

```
while (true)
{
    turn = i;
    while (turn == j)
        ;
    /* critical section
     */
    turn = j;
    /* remainder
       section */
}
```

```
while (true)
{
    turn = j;
    while (turn == i)
        ;
    /* critical section
     */
    turn = i;
    /* remainder
       section */
}
```





# Correctness of the Software Solution

- Mutual exclusion is preserved

$P_i$  enters critical section only if:

**turn = i**

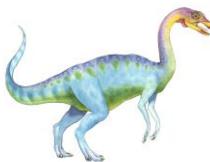
and **turn** cannot be both 0 and 1 at the same time

- What about the Progress requirement?

- E.g. consider the case when  $P_j$  is faster than  $P_i$ 
    - ▶ While  $P_i$  is waiting in the loop,  $P_j$  could be leaving (by setting  $turn=i$ ) and then entering again (by setting  $turn=j$ ) **before**  $P_i$  could proceed

- What about the Bounded-waiting requirement?



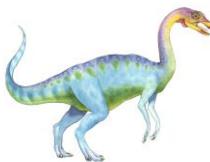


# Peterson's Solution

---

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





# Algorithm for Process $P_i$

```
while (true) {
```

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

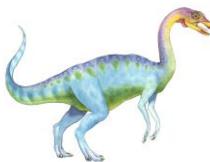
```
    /* critical section */
```

```
    flag[i] = false;
```

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

```
}
```



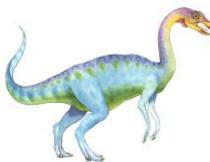


# Algorithm for Process $P_i$ and $P_j$

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

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





# Correctness of Peterson's Solution

---

- Provable that the three CS requirements are met:

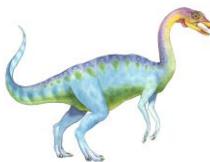
1. Mutual exclusion is preserved

$P_i$  enters CS only if:

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

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



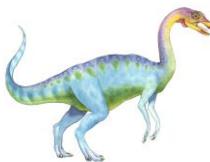


## Peterson's Solution and Modern Architecture

---

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





# Modern Architecture Example

- Two threads share the data:

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

- Thread 1 performs

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

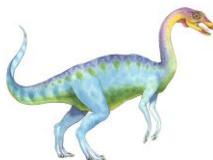
- Thread 2 performs

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

- What is the expected output?

100





# Modern Architecture Example (Cont.)

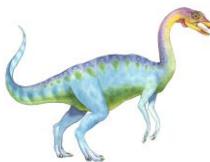
- However, since the variables `flag` and `x` are independent of each other, the instructions:

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

for Thread 2 may be reordered

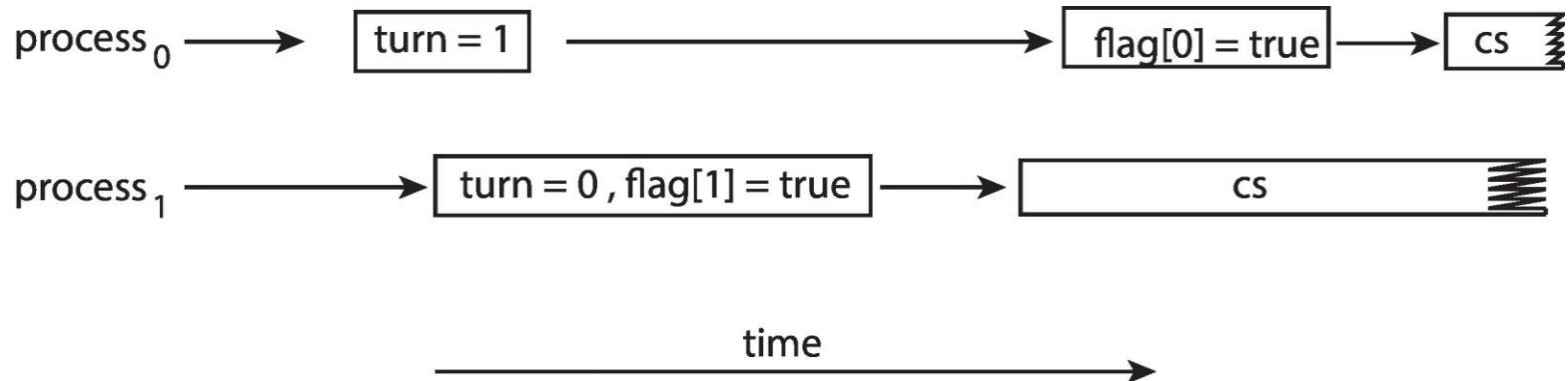
- If this occurs, the output may be 0!





# Peterson's Solution Revisited

- The effects of instruction reordering in Peterson's Solution



- This allows both processes to be in their critical section at the same time!
- To ensure that Peterson's solution will work correctly on modern computer architecture we must use **Memory Barrier**





# Effect of Instruction Reordering on Peterson's Solution

```
while (true)
{
    turn = 1;
    flag[0] = true;
    while (flag[1] && turn
           == 1)
        ;
    /* critical section */
    flag[0] = false;
    /* remainder section */
}
```

```
while (true)
{
    turn = 0;
    flag[1] = true;
    while (flag[0] && turn
           == 0)
        ;
    /* critical section */
    flag[1] = false;
    /* remainder section */
}
```



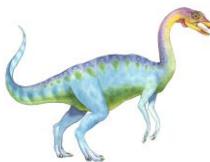


# Memory Barrier

---

- **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**

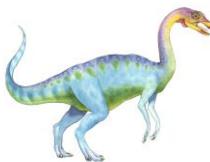




# Memory Barrier Instructions

- When a memory barrier instruction is performed, the system ensures that all loads and stores are **completed** before any subsequent load or store operations are performed
- Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed





# Memory Barrier Example

- Returning to the example of slides 6.17 - 6.18
- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

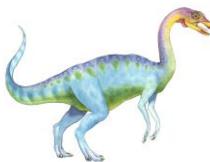
```
while (!flag)
    memory_barrier();
    print x
```

- Thread 2 now performs

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

- For Thread 1, we are guaranteed that the value of flag is loaded **before** the value of x
- For Thread 2, we ensure that the assignment to x occurs **before** the assignment flag



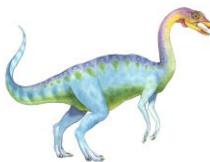


# Synchronization Hardware

---

- 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 two forms of hardware support:
  1. Hardware instructions
  2. Atomic variables





# Hardware Instructions

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





# The test\_and\_set Instruction

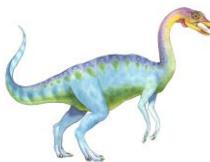
- Definition

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

- Properties

- Executed atomically
- Returns the original value of passed parameter
- Set the new value of passed parameter to **true**





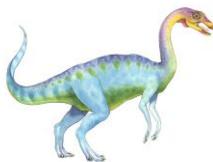
# Solution Using test\_and\_set()

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

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

- Does it solve the critical-section problem?





# The compare\_and\_swap Instruction

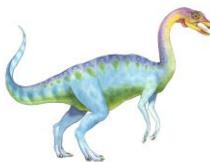
- Definition

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

- Properties

- Executed atomically
- Returns the original value of passed parameter **value**
- Set 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 */

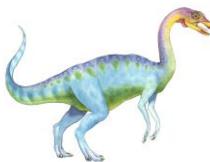
    /* critical section */

    lock = 0;

    /* remainder section */
}
```

- Does it solve the critical-section problem?

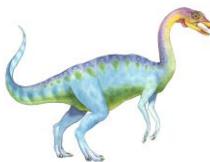




# Bounded-waiting with compare-and-swap

```
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 */  
}
```



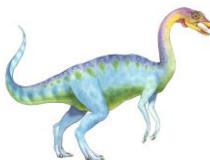


# Atomic Variables

---

- 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:
  - Let **sequence** be an atomic variable
  - Let **increment()** be operation on the atomic variable **sequence**
  - The Command:  
**increment (&sequence) ;**  
ensures **sequence** is incremented without interruption





# 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
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First **acquire()** a lock
  - Then **release()** the lock
- 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**

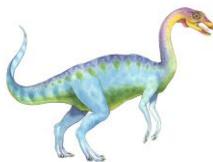




# Solution to CS Problem Using Mutex Locks

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





# Semaphore

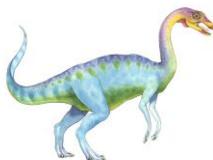
- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities
  - Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **wait()** and **signal()**
    - ▶ Originally called **P()** and **V()**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

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

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

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



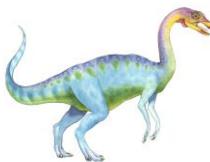


# Semaphore (Cont.)

---

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





# Semaphore Usage Example

---

- Solution to the CS Problem

- Create a semaphore “`mutex`” initialized to 1

```
    wait(mutex);
```

CS

```
    signal(mutex);
```

- Consider  $P_1$ , and  $P_2$  that with two statements  $S_1$ , and  $S_2$  and the requirement that  $S_1$  to happen before  $S_2$

- Create a semaphore “`synch`” initialized to 0

$P1:$

```
    S1;
```

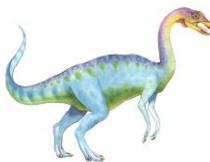
```
    signal(synch);
```

$P2:$

```
    wait(synch);
```

```
    S2;
```





# 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



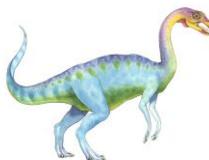


# Semaphore Implementation with no Busy waiting

---

- 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



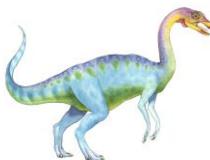


# Implementation with no Busy waiting (Cont.)

- Waiting queue

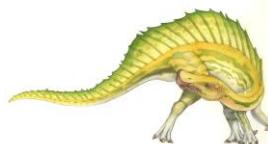
```
typedef struct {  
    int value;  
    struct process *list;  
} semaphore;
```





# 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);  
    }  
}
```

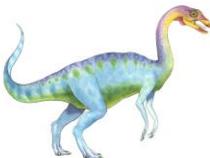




# Problems with Semaphores

- 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



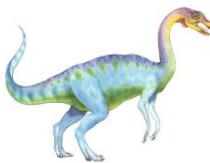


# Monitors

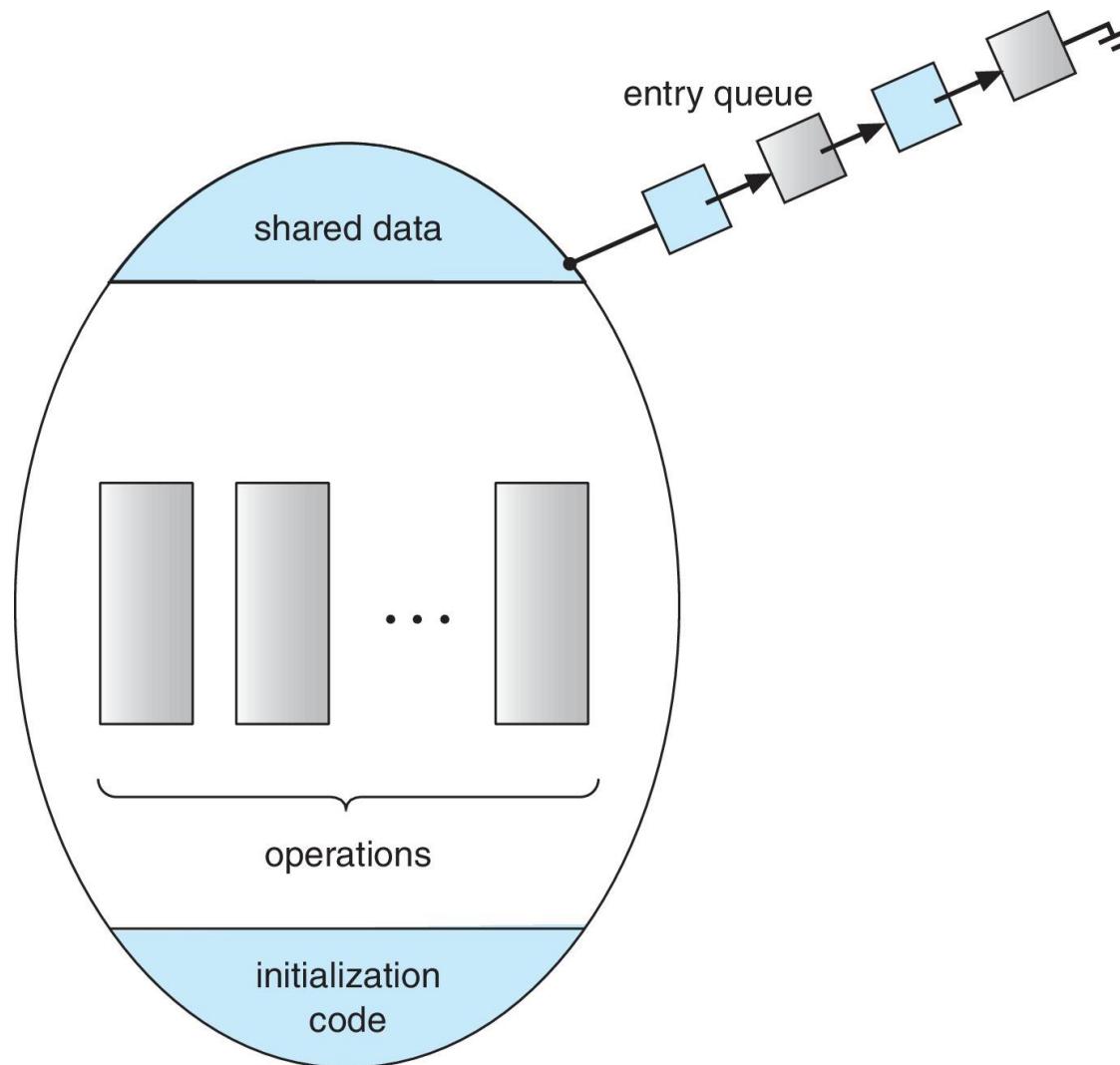
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
  - *Abstract data type*, internal variables only accessible by code within the procedure
  - Only one process may be active within the monitor at a time
- Pseudocode syntax of a monitor:

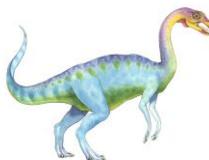
```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { ... }
    procedure P2 (...) { ... }
    procedure Pn (...) { ... }
    initialization code (...) { ... }
}
```





# Schematic view of a Monitor





# Monitor Implementation Using Semaphores

---

- Variables

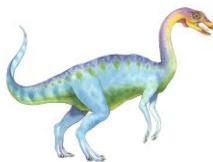
```
semaphore mutex;  
mutex = 1;
```

- Each procedure  $P$  is replaced by

```
wait(mutex);  
...  
// body of P;  
...  
signal(mutex);
```

- Mutual exclusion within a monitor is ensured

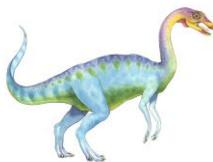




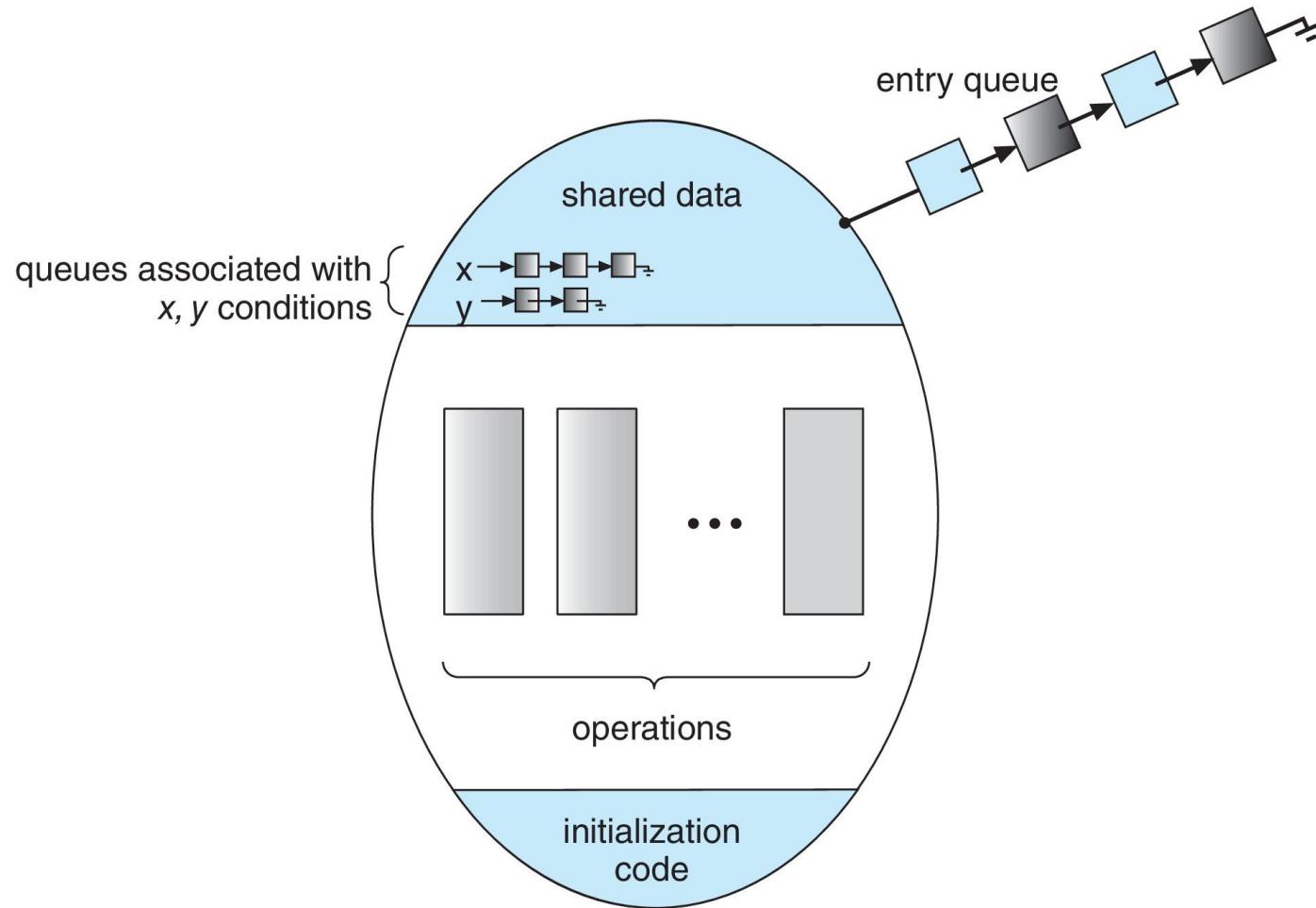
# Condition Variables

- **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





# Usage of Condition Variable Example

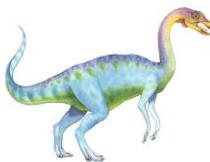
- Consider  $P_1$  and  $P_2$  that need to execute two statements  $S_1$  and  $S_2$  and the requirement that  $S_1$  to happen before  $S_2$ 
  - Create a monitor with two procedures  $F_1$  and  $F_2$  that are invoked by  $P_1$  and  $P_2$  respectively
  - One condition variable “x” initialized to 0
  - One Boolean variable “done”
  - F1:**

```
S1;  
  
done = true;  
  
x.signal();
```

- F2:**

```
if (done == false)  
  
x.wait();  
  
S2;
```





# Monitor Implementation Using Semaphores

- Variables

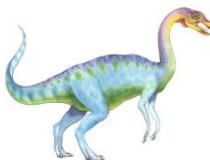
```
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next_count = 0; // number of processes
waiting inside the monitor
```

- Each function *P* will be replaced by

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

- Mutual exclusion within a monitor is ensured





# Implementation – Condition Variables

---

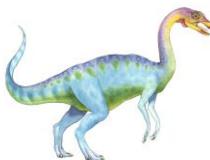
- For each condition variable **x**, we have:

```
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--;
```





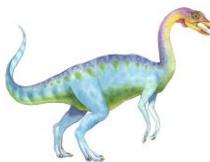
# Implementation (Cont.)

---

- 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
- Use the **conditional-wait** construct of the form

**x.wait(c)**

where:

- **c** is an integer (called the priority number)
- The process with lowest number (highest priority) is scheduled next





# Single Resource allocation

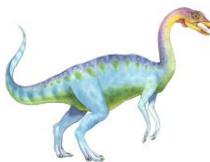
---

- Allocate a single resource among competing processes using priority numbers that specifies the maximum time a process plans to use the resource
  - The process with the shortest time is allocated the resource first
- Let R is an instance of type **ResourceAllocator** (next slide)
- Access to **ResourceAllocator** is done via:

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

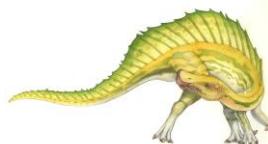
- Where **t** is the maximum time a process plans to use the resource

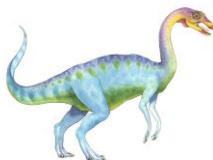




# A Monitor to Allocate Single Resource

```
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;
    }
}
```





# Single Resource Monitor (Cont.)

- Usage:

```
acquire()
```

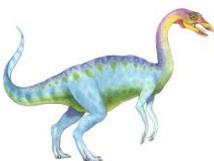
```
...
```

```
release()
```

- Incorrect use of monitor operations

- **release()** ... **acquire()**
- **acquire()** ... **acquire()**
- Omitting of **acquire()** and/or **release()**

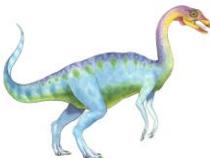




# Liveness

- 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





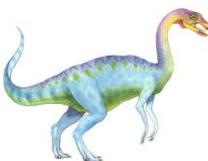
# Liveness

- **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**





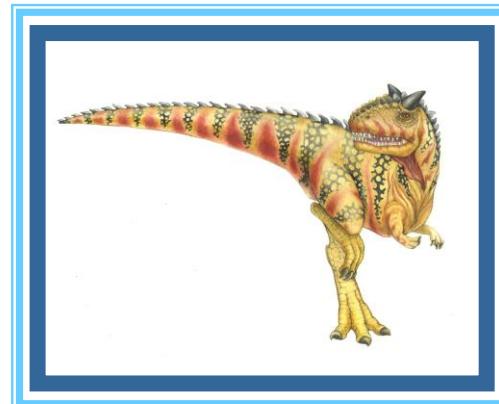
# Liveness

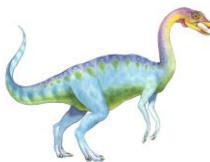
---

- 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**



# End of Chapter 6





# 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 that **P3** is holding semaphore **S** and that **P1** is waiting to **S** to be released
- Assume that **P2** is assigned the CPU and preempts **P3**
  - **P3** is still holding semaphore **S**
  - **P1** is waiting to **S** to be released
- 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.





# Usage of Condition Variable Example

- Consider  $P_1$  and  $P_2$  that need to execute two statements  $S_1$  and  $S_2$  and the requirement that  $S_1$  to happen before  $S_2$

- Create a monitor with two procedures  $F_1$  and  $F_2$  that are invoked by  $P_1$  and  $P_2$  respectively
- One condition variable “x” initialized to 0
- One Boolean variable “done”
- **F1 :**

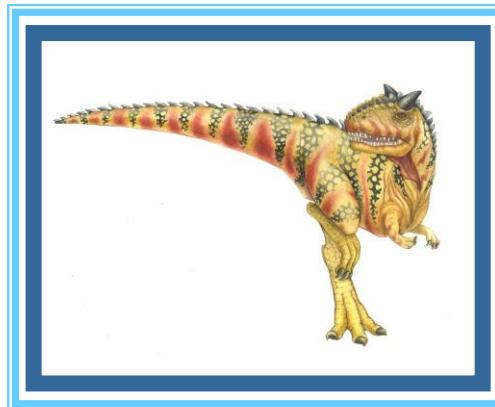
```
S1;  
done = true;  
x.signal();
```

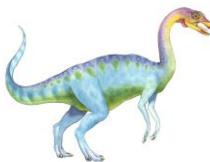
- **F2 :**

```
if (done == false)  
x.wait();  
S2;
```




# Chapter 7: Synchronization Examples



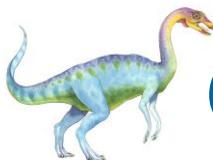


# Outline

---

- Explain the classical synchronization problems
  - the bounded-buffer problem
  - the readers-writers problem
  - the dining-philosophers problem
- 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



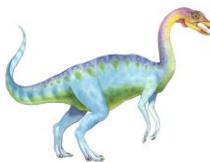


# Classical Problems of Synchronization

---

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



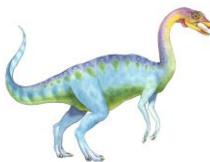


# Bounded-Buffer Problem

---

- $n$  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



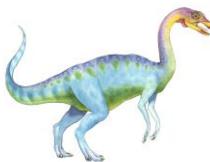


# Bounded Buffer Problem (Cont.)

- 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);  
}
```



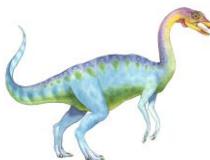


# Bounded Buffer Problem (Cont.)

- 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 */  
  
    ...  
}
```

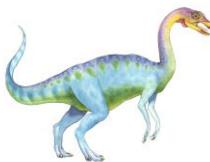




# Readers-Writers Problem

- 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

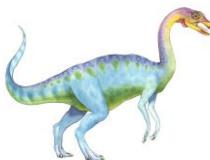




# Readers-Writers Problem (Cont.)

- Shared Data
  - Data set
  - Semaphore **rw\_mutex** initialized to 1
  - Semaphore **mutex** initialized to 1
  - Integer **read\_count** initialized to 0



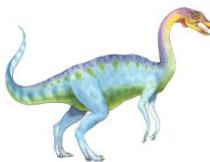


# Readers-Writers Problem (Cont.)

- 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:

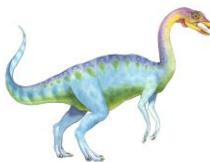
```
while (true) {
    wait(mutex);
    read_count++;
    if (read_count == 1) /* first reader */
        wait(rw_mutex);
    signal(mutex);

    ...
/* reading is performed */

    ...
wait(mutex);
read_count--;
if (read_count == 0) /* last reader */
    signal(rw_mutex);
signal(mutex);

}
```

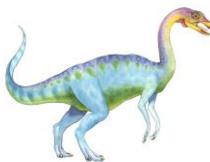




# Readers-Writers Problem Variations

- The solution in previous slide can result in a situation where a writer process never writes
  - “First reader-writer” problem
- The “Second reader-writer” problem is a variation the first reader-writer problem that state:
  - Once a writer is ready to write, no “newly arrived reader” is allowed to read
- Both the first and second may result in starvation, leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks





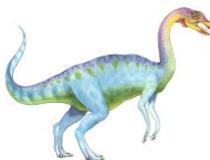
# Dining-Philosophers Problem

- N philosophers sit at a round table with a bowel of rice in the middle



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





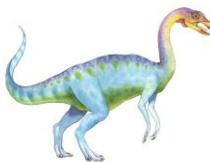
# Dining-Philosophers Problem Algorithm

- Semaphore solution
- The structure of Philosopher *i* :

```
while (true) {  
    wait (chopstick[i]);  
    wait (chopstick[(i + 1) % 5]);  
  
    /* eat for a while */  
  
    signal (chopstick[i]);  
    signal (chopstick[(i + 1) % 5]);  
  
    /* think for a while */  
}
```

- What is the problem with this algorithm?





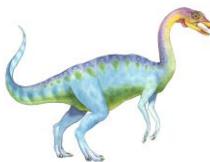
# Monitor Solution to Dining Philosophers

```
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);
    }
}
```

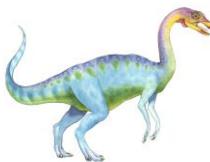




# Solution to Dining Philosophers (Cont.)

```
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:

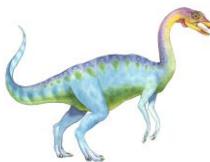
```
DiningPhilosophers.pickup(i) ;
```

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

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

- No deadlock, but starvation is possible



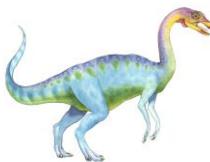


# Kernel Synchronization - Windows

---

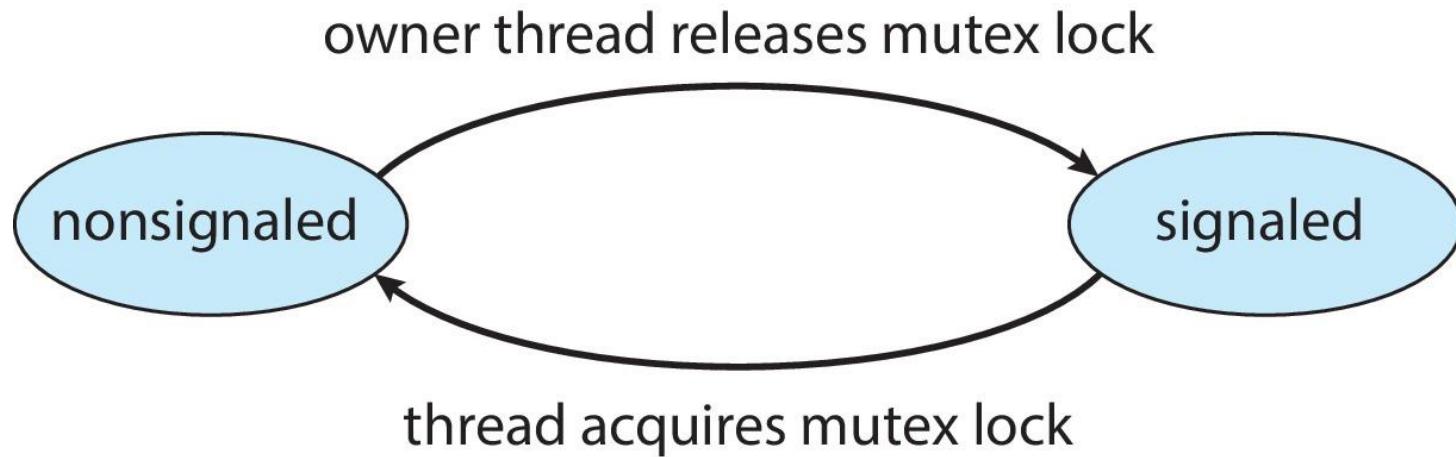
- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** including mutexes, semaphores, events, and timers
  - **Events**
    - ▶ An event acts much like a condition variable
    - Timers notify one or more threads when time expired
    - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

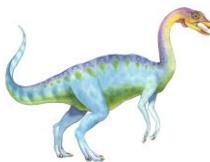




# Kernel Synchronization - Windows

- Mutex dispatcher object





# 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:
  - Atomic integers
  - Mutex locks
  - Spinlocks, Semaphores
  - Reader-writer versions of both
- On single-CPU system, spinlocks replaced by enabling and disabling kernel preemption





# Linux Synchronization

- Atomic variables

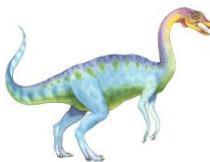
**atomic\_t** is the type for atomic integer

- Consider the variables

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

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

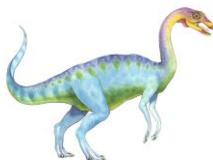




# POSIX Synchronization

- POSIX API provides
  - mutex locks
  - semaphores
  - condition variables
- 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);
```

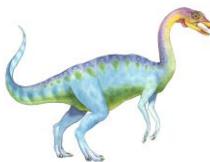
- Acquiring and releasing the lock

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

/* critical section */

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





# POSIX Semaphores

---

- POSIX provides two versions of semaphores – **named** and **unnamed**
- Named semaphores can be used by unrelated processes
- Unnamed semaphores can be used only by threads in the same process





# POSIX Named Semaphores

- Creating and initializing the named semaphore:

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

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

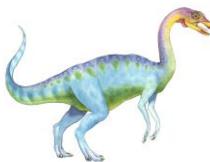
- Another process can access the semaphore by referring to its name **SEM**
- Acquiring and releasing the semaphore:

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

/* critical section */

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





# POSIX Unnamed Semaphores

---

- Creating and initializing the unnamed semaphore:

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

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

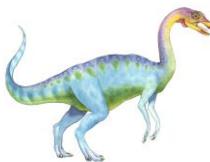
- Acquiring and releasing the semaphore:

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

/* critical section */

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



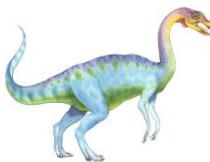


# POSIX Condition Variables

- 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:

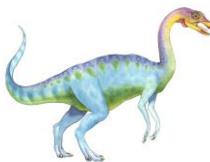
```
pthread_mutex_lock(&mutex);
while (a != b)
    pthread_cond_wait(&cond_var, &mutex);

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);
```



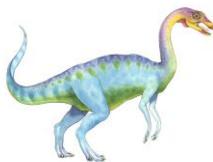


# Java Synchronization

---

- Java provides rich set of synchronization features:
  - Java monitors
  - Reentrant locks
  - Semaphores
  - Condition variables

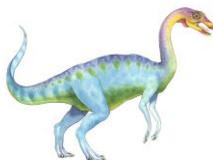




# Java Monitors

- Every Java object has associated with it a single lock.
- If a method is declared as **synchronized**, a calling thread must own the lock for the object.
- If the lock is owned by another thread, the calling thread must wait for the lock until it is released.
- Locks are released when the owning thread exits the **synchronized** method.





# Bounded Buffer – Java Synchronization

```
public class BoundedBuffer<E>
{
    private static final int BUFFER_SIZE = 5;

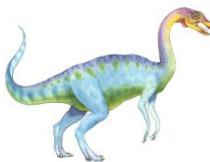
    private int count, in, out;
    private E[] buffer;

    public BoundedBuffer() {
        count = 0;
        in = 0;
        out = 0;
        buffer = (E[]) new Object[BUFFER_SIZE];
    }

    /* Producers call this method */
    public synchronized void insert(E item) {
        /* See Figure 7.11 */
    }

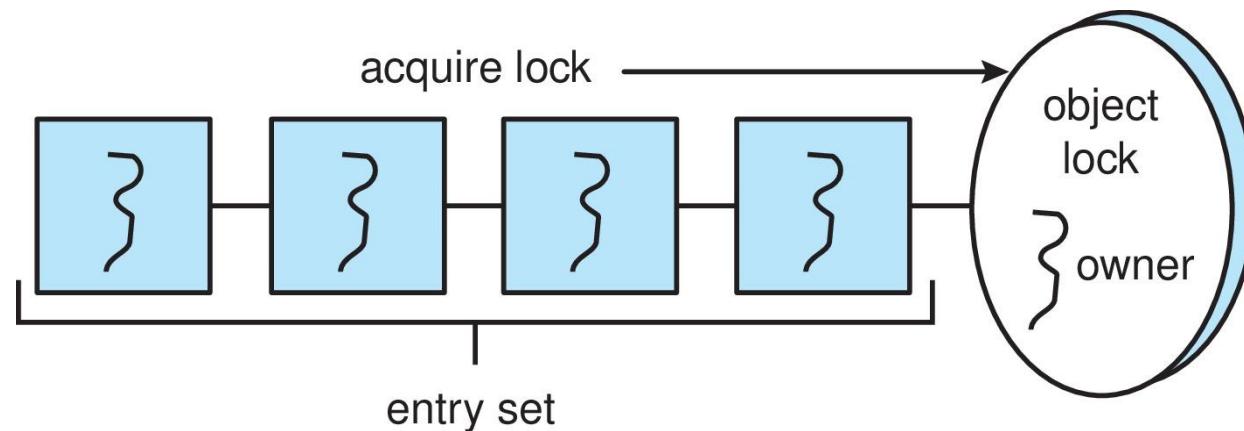
    /* Consumers call this method */
    public synchronized E remove() {
        /* See Figure 7.11 */
    }
}
```

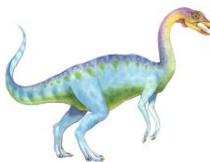




# Java Synchronization

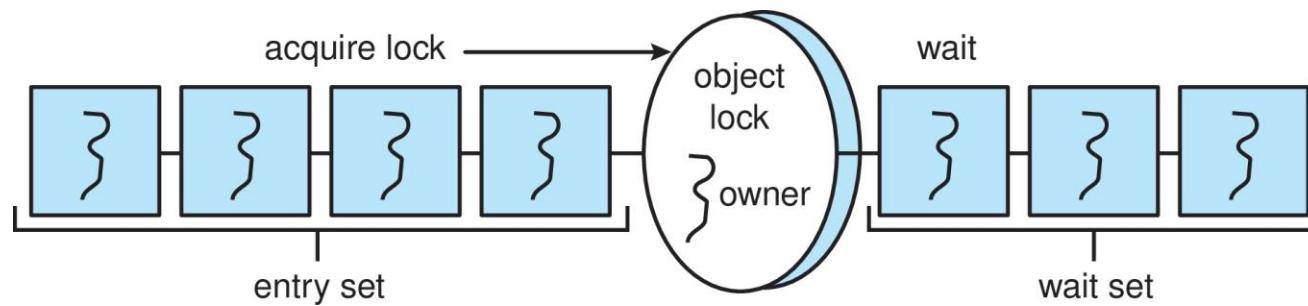
- A thread that tries to acquire an unavailable lock is placed in the object's **entry set**:





# Java Synchronization

- Similarly, each object also has a **wait set**.
- When a thread calls **wait()**:
  1. It releases the lock for the object
  2. The state of the thread is set to blocked
  3. The thread is placed in the wait set for the object





# Java Synchronization

---

- A thread typically calls `wait()` when it is waiting for a condition to become true.
- How does a thread get notified?
- When a thread calls `notify()`:
  1. An arbitrary thread T is selected from the wait set
  2. T is moved from the wait set to the entry set
  3. Set the state of T from blocked to runnable.
- T can now compete for the lock to check if the condition it was waiting for is now true.



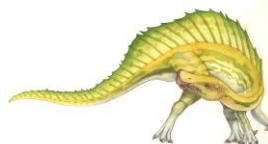


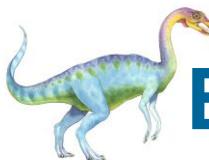
# Bounded Buffer – Java Synchronization

```
/* Producers call this method */
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE) {
        try {
            wait();
        }
        catch (InterruptedException ie) { }
    }

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    count++;

    notify();
}
```





# Bounded Buffer – Java Synchronization

```
/* Consumers call this method */
public synchronized E remove() {
    E item;

    while (count == 0) {
        try {
            wait();
        }
        catch (InterruptedException ie) { }

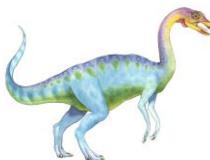
    }

    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;

    notify();

    return item;
}
```





# Java Reentrant Locks

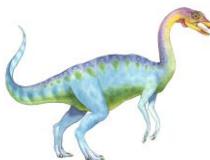
---

- Similar to mutex locks
- The **finally** clause ensures the lock will be released in case an exception occurs in the **try** block.

```
Lock key = new ReentrantLock();

key.lock();
try {
    /* critical section */
}
finally {
    key.unlock();
}
```





# Java Semaphores

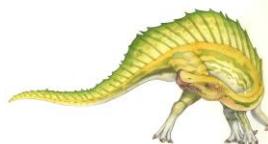
- Constructor:

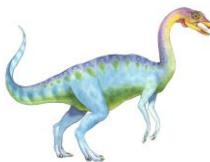
```
Semaphore(int value);
```

- Usage:

```
Semaphore sem = new Semaphore(1);

try {
    sem.acquire();
    /* critical section */
}
catch (InterruptedException ie) { }
finally {
    sem.release();
}
```





# Java Condition Variables

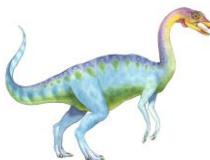
---

- Condition variables are associated with an **ReentrantLock**.
- Creating a condition variable using **newCondition()** method of **ReentrantLock**:

```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

- A thread waits by calling the **await()** method, and signals by calling the **signal()** method.





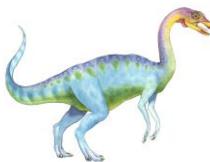
# Java Condition Variables

- Example:
- Five threads numbered 0 .. 4
- Shared variable **turn** indicating which thread's turn it is.
- Thread calls **doWork()** when it wishes to do some work. (But it may only do work if it is their turn.)
- If not their turn, wait
- If their turn, do some work for awhile .....
- When completed, notify the thread whose turn is next.
- Necessary data structures:

```
Lock lock = new ReentrantLock();
Condition[] condVars = new Condition[5];

for (int i = 0; i < 5; i++)
    condVars[i] = lock.newCondition();
```





# Java Condition Variables

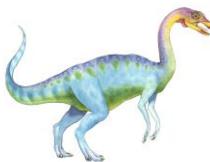
```
/* threadNumber is the thread that wishes to do some work */
public void doWork(int threadNumber)
{
    lock.lock();

    try {
        /**
         * If it's not my turn, then wait
         * until I'm signaled.
        */
        if (threadNumber != turn)
            condVars[threadNumber].await();

        /**
         * Do some work for awhile ...
        */

        /**
         * Now signal to the next thread.
        */
        turn = (turn + 1) % 5;
        condVars[turn].signal();
    }
    catch (InterruptedException ie) { }
    finally {
        lock.unlock();
    }
}
```



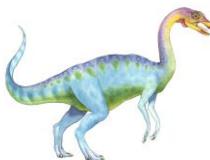


# Alternative Approaches

---

- Transactional Memory
- OpenMP
- Functional Programming Languages





# Transactional Memory

- Consider a function update() that must be called atomically. One option is to use mutex locks:

```
void update ()  
{  
    acquire();  
  
    /* modify shared data */  
  
    release();  
}
```

- A **memory transaction** is a sequence of read-write operations to memory that are performed atomically. A transaction can be completed by adding **atomic{S}** which ensure statements in S are executed atomically:

```
void update ()  
{  
    atomic {  
        /* modify shared data */  
    }  
}
```





# OpenMP

- OpenMP is a set of compiler directives and API that support parallel programming

```
void update(int value)
{
    #pragma omp critical
    {
        count += value
    }
}
```

- The code contained within the `#pragma omp critical` directive is treated as a critical section and performed atomically



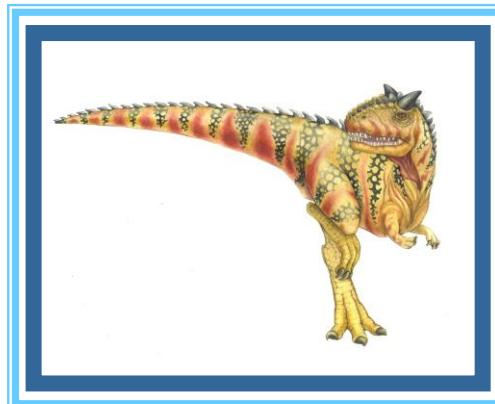


# Functional Programming Languages

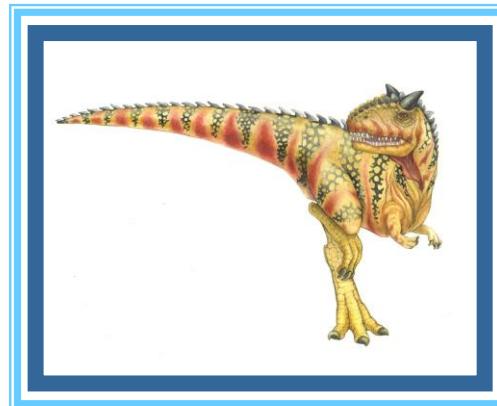
- Functional programming languages offer a different paradigm than procedural languages in that they do **not** maintain state
- Variables are treated as **immutable** and cannot change state once they have been assigned a value
- Most of the synchronization problems do not exist in functional languages
- There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races

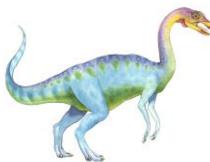


# End of Chapter 7



# Chapter 8: Deadlocks



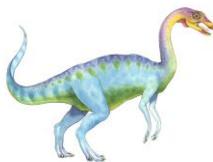


# Outline

---

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





# Chapter Objectives

---

- Illustrate how deadlock can occur when mutex locks are used
- Define the **four necessary conditions** that characterize deadlock
- Identify a deadlock situation in a resource allocation graph
- Evaluate the four different approaches for preventing deadlocks
- Apply the **banker's algorithm** for deadlock avoidance
- Apply the deadlock detection algorithm
- Evaluate approaches for recovering from deadlock

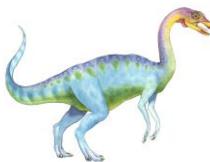




# System Model

- 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 with Semaphores

- Data:
  - A semaphore **s1** initialized to 1
  - A semaphore **s2** initialized to 1

- Two processes P1 and P2

- P1:

```
wait(s1)
```

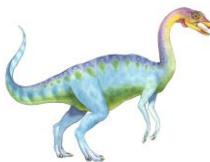
```
wait(s2)
```

- P2:

```
wait(s2)
```

```
wait(s1)
```



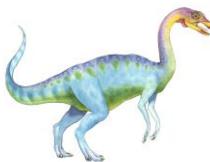


# Deadlock Characterization

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  $P_0$



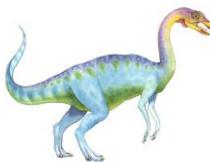


# Resource-Allocation Graph

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

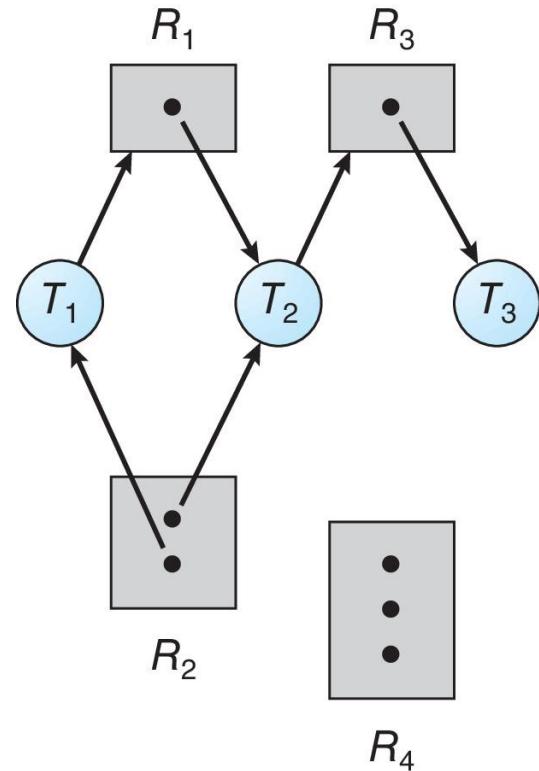
- $V$  is partitioned into two types:
  - $P = \{P_1, P_2, \dots, P_n\}$ , the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system
- **request edge** – directed edge  $P_i \rightarrow R_j$
- **assignment edge**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





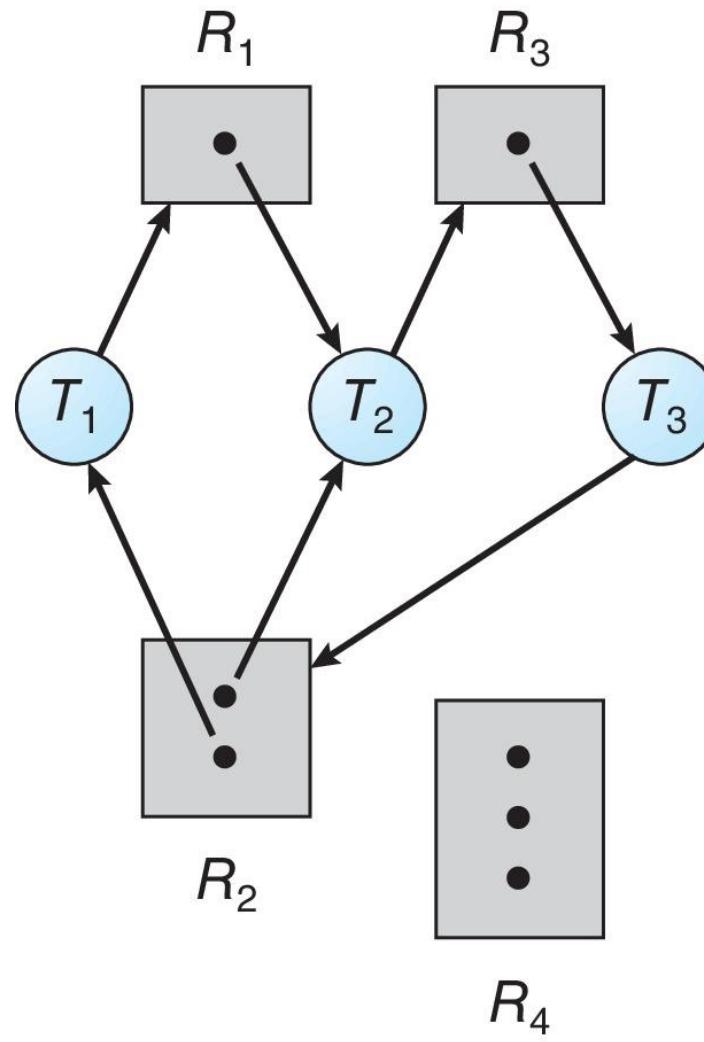
# Resource Allocation Graph Example

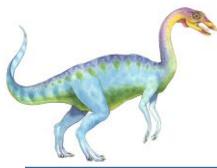
- Resources
  - One instance of R1
  - Two instances of R2
  - One instance of R3
  - Three instances of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 holds one instance of R3



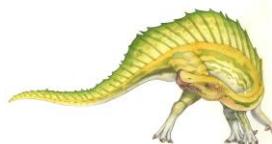
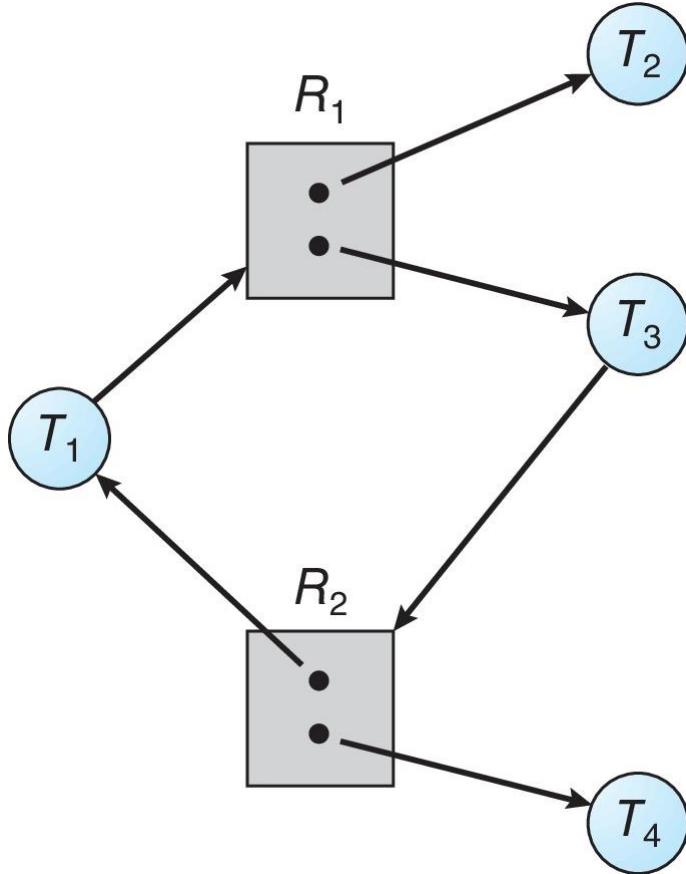


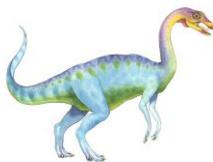
# Resource Allocation Graph with a Deadlock





# Graph with a Cycle But no Deadlock



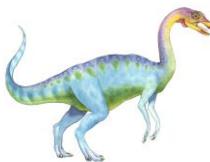


# Basic Facts

---

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

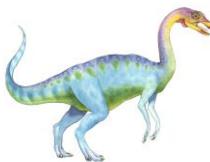




# Methods for Handling Deadlocks

- 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





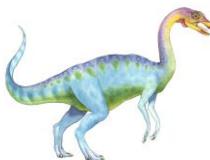
# Deadlock Prevention

---

To 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:

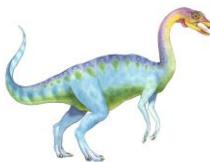
```
first_mutex = 1  
second_mutex = 5
```

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);  
}
```



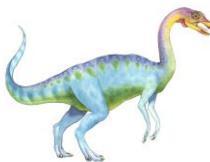


# Deadlock Avoidance

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

- Simplest and most useful model requires that each process declares 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 system 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  $P_j$  finishes,  $P_i$  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



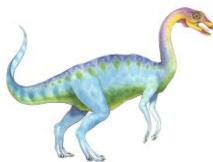


# Basic Facts

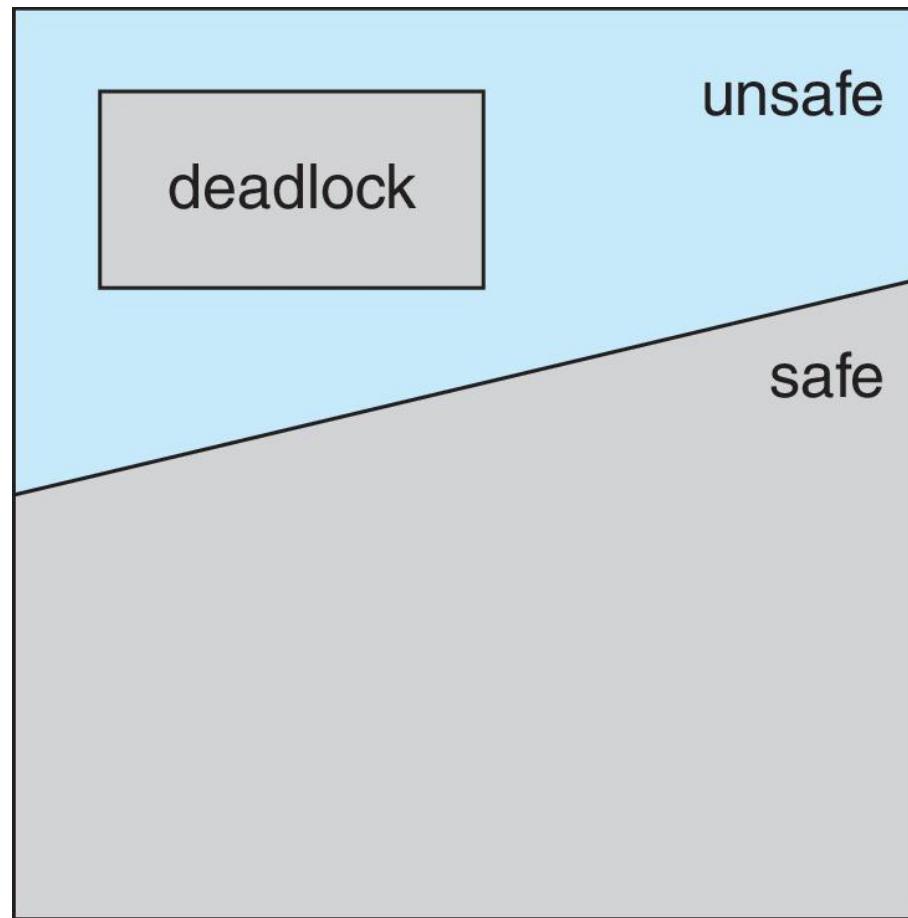
---

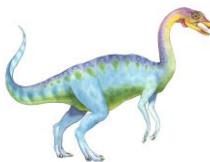
- If a system is in safe state  $\Rightarrow$  no deadlocks
- If a system is in unsafe state  $\Rightarrow$  possibility of deadlock
- Avoidance  $\Rightarrow$  ensure that a system will never enter an unsafe state





# Safe, Unsafe, Deadlock State

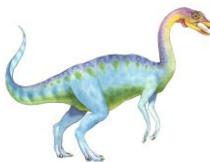




# Avoidance Algorithms

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



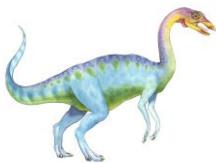


# Resource-Allocation Graph Scheme

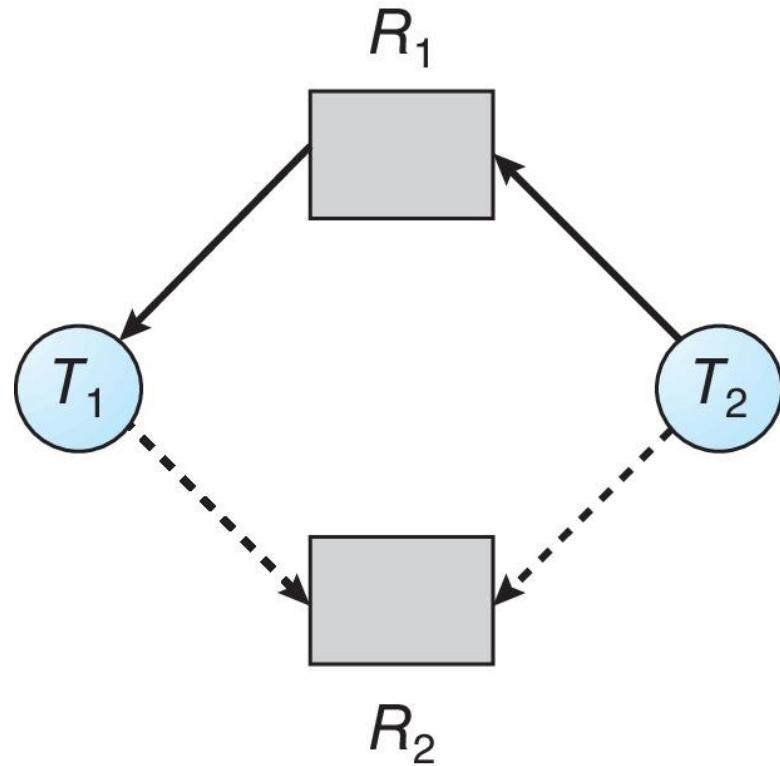
---

- **Claim edge**  $P_i \rightarrow R_j$  indicates that process  $P_i$  may request resource  $R_j$ ; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge is 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



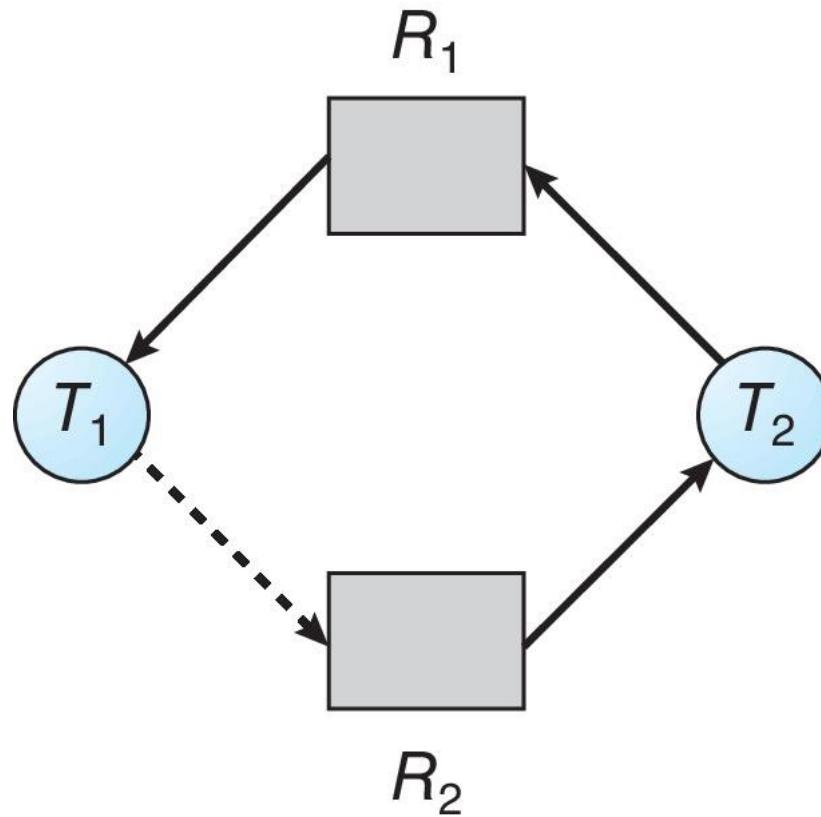


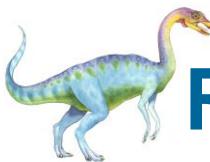
# Resource-Allocation Graph





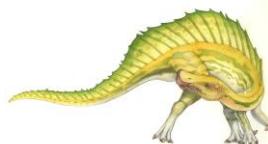
# Unsafe State in Resource-Allocation Graph

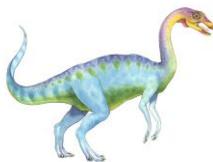




# Resource-Allocation Graph Algorithm

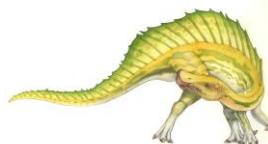
- Suppose that process  $P_i$  requests a resource  $R_j$
- The request can be granted **only if** converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

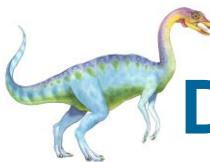




# Banker's Algorithm

- 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





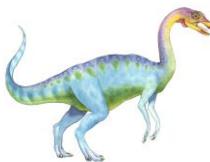
# Data Structures for the Banker's Algorithm

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**

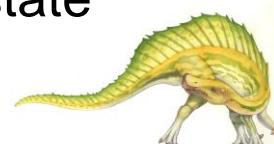
If no such  $i$  exists, go to step 4

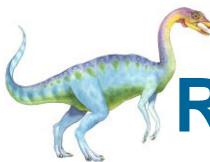
3. **Work = Work + Allocation<sub>i</sub>**,

**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:

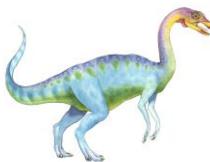
$$\text{Available} = \text{Available} - \text{Request}_i;$$

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

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

- 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>		
	$A$	$B$	$C$	$A$	$B$	$C$	$A$	$B$	$C$
$P_0$	0	1	0	7	5	3	3	3	2
$P_1$	2	0	0	3	2	2			
$P_2$	3	0	2	9	0	2			
$P_3$	2	1	1	2	2	2			
$P_4$	0	0	2	4	3	3			





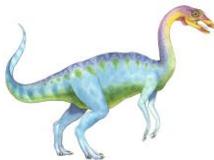
## Example (Cont.)

- 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





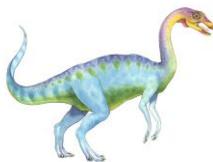
## Example: $P_1$ Request (1,0,2)

- 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





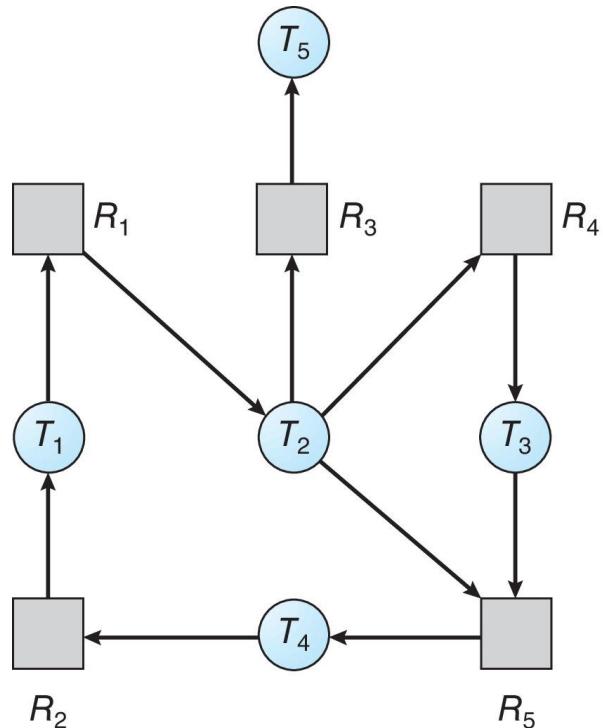
# Single Instance of Each Resource Type

- 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  $O(n^2)$  operations, where  $n$  is the number of vertices in the graph

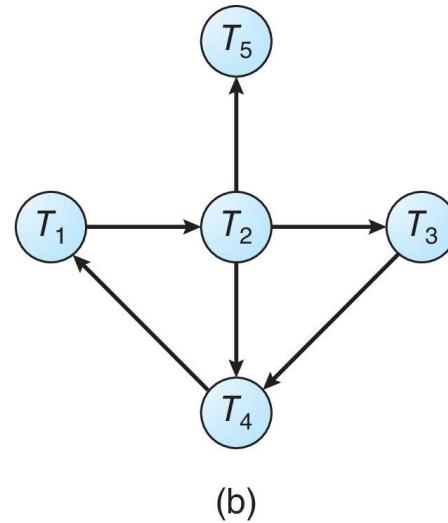




# Resource-Allocation Graph and Wait-for Graph



(a)

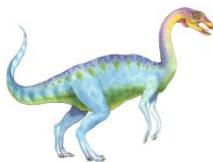


(b)

Resource-Allocation Graph

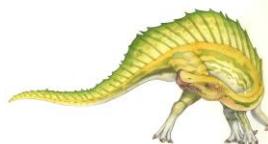
Corresponding wait-for graph





# 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  $R_j$





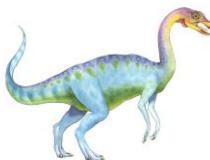
# Detection Algorithm

---

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



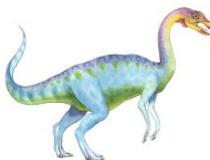


## Detection Algorithm (Cont.)

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





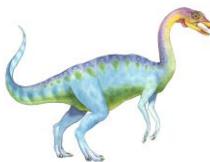
# Example of Detection Algorithm

- 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>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
$P_0$	0	1	0	0	0	0	0	0	0
$P_1$	2	0	0	2	0	2			
$P_2$	3	0	3	0	0	0			
$P_3$	2	1	1	1	0	0			
$P_4$	0	0	2	0	0	2			

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





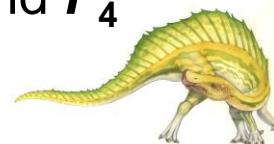
## Example (Cont.)

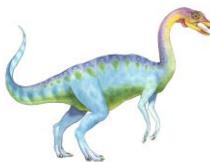
- $P_2$  requests an additional instance of type **C**

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



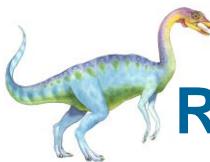


## Recovery from Deadlock: Process Termination

---

- 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?





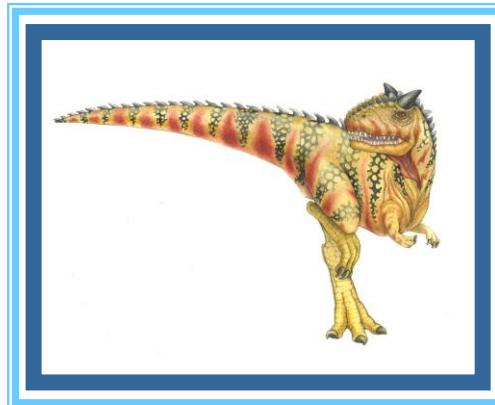
## Recovery from Deadlock: Resource Preemption

---

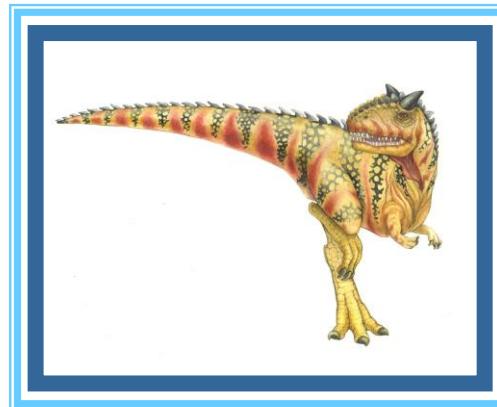
- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, so we should include number of rollbacks in cost factor

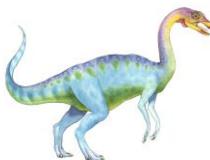


# End of Chapter 8



# Chapter 9: Main Memory





# Chapter 9: Memory Management

---

- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation
- Swapping
- Example: The Intel 32 and 64-bit Architectures
- Example: ARMv8 Architecture





# Objectives

---

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various **memory management** techniques
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging



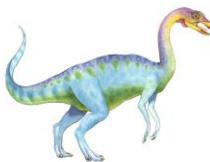


# Background

---

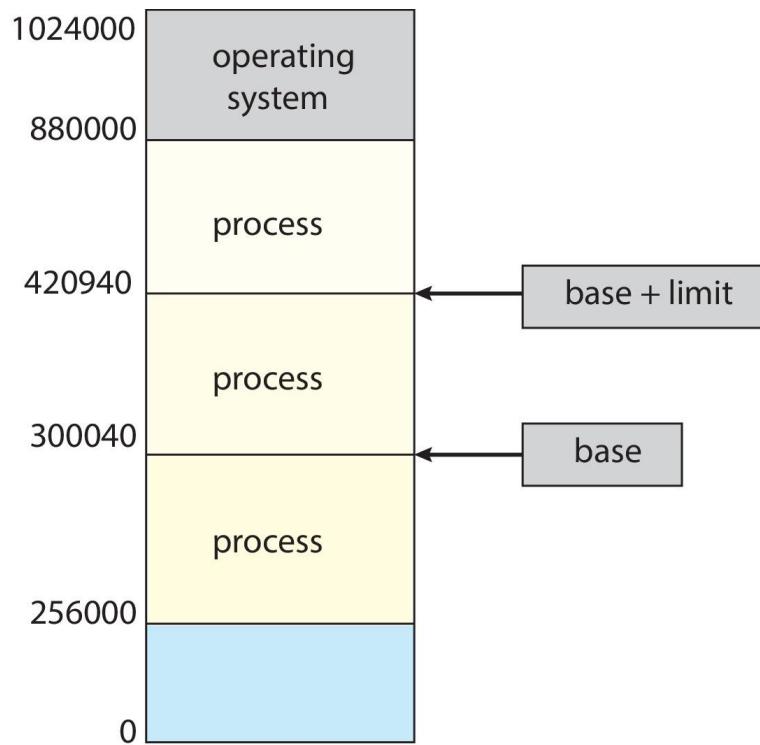
- Program must be brought (from disk) into memory and placed within a process for it to be run
  - Instruction, data
- Memory unit only sees a stream of:
  - addresses + read requests, or
  - addresses + data and write requests
- The only storage CPU can access directly
  - Register access is done in one CPU clock (or less)
  - Main memory can take many cycles, causing a **stall**
  - **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

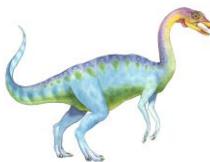




# Protection

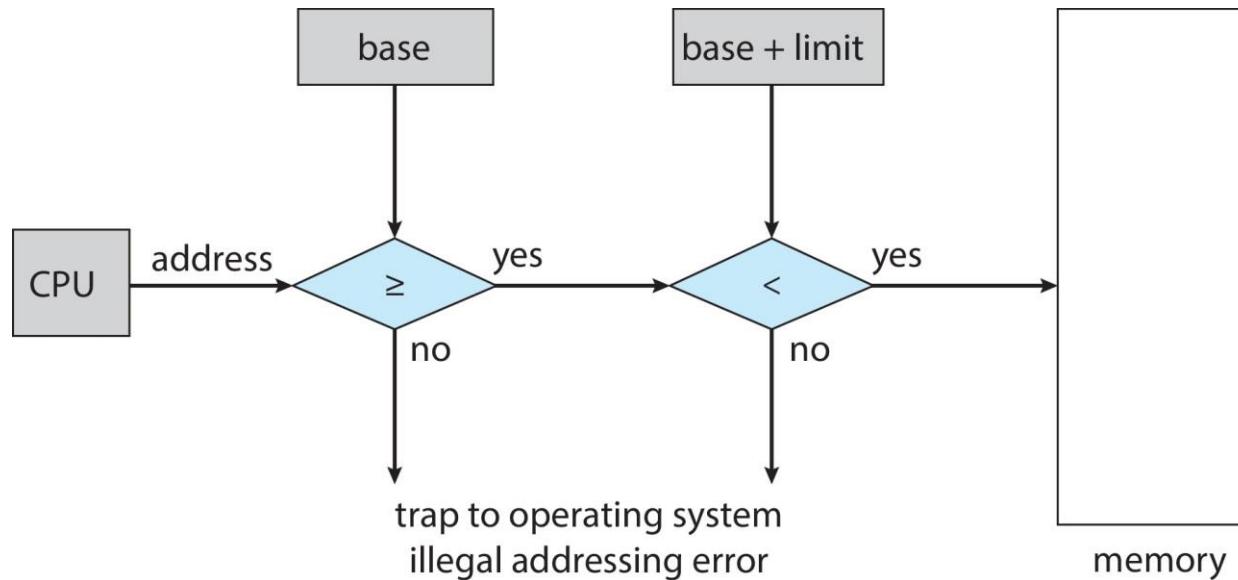
- Need to ensure that a process can only access those addresses in its address space
- We can provide this protection by using a pair of **base** and **limit registers** defining the logical address space of a process





# Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



- The instructions to load the base and limit registers are privileged



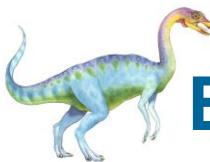


# Address Binding

---

- Programs on disk, ready to be brought into memory to execute from an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user physical address always at 0000
  - How can it not be?
- Addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - ▶ i.e., “14 bytes from beginning of this module”
  - Linker or loader will bind relocatable addresses to absolute addresses
    - ▶ i.e., 74014
  - Each binding maps one address space to another

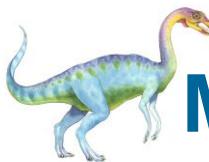




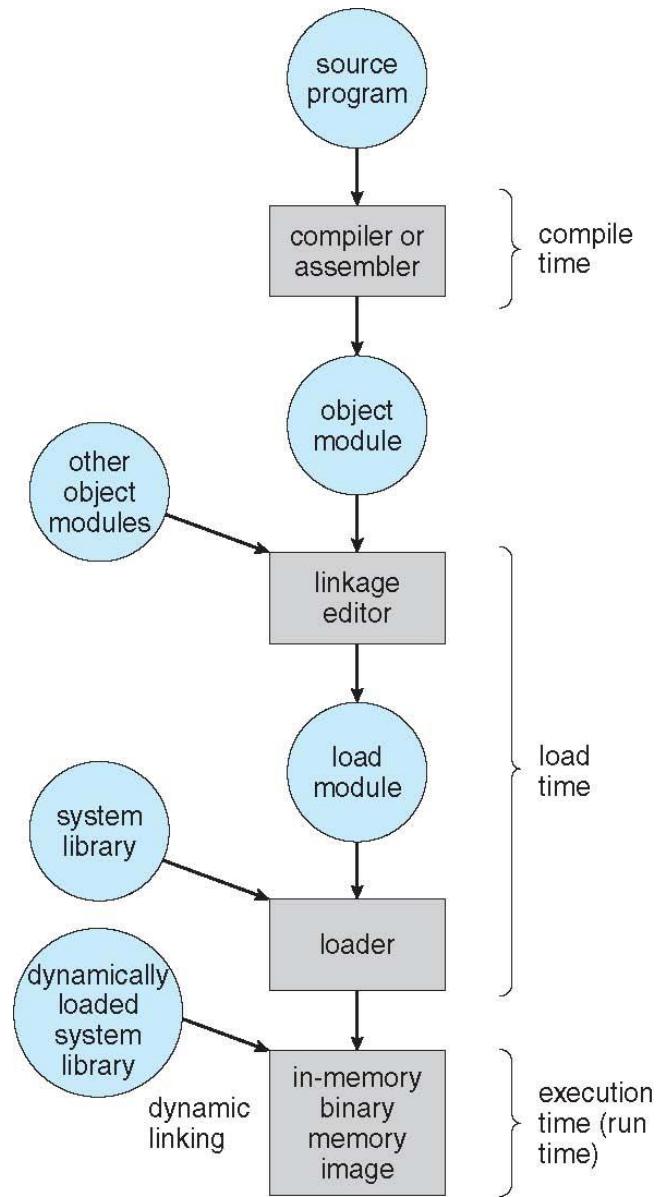
# Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time:** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time:** Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - ▶ Need hardware support for address maps (e.g., base and limit registers)





# Multistep Processing of a User Program



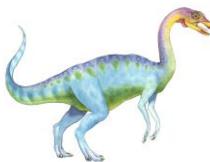


# Logical vs. Physical Address Space

---

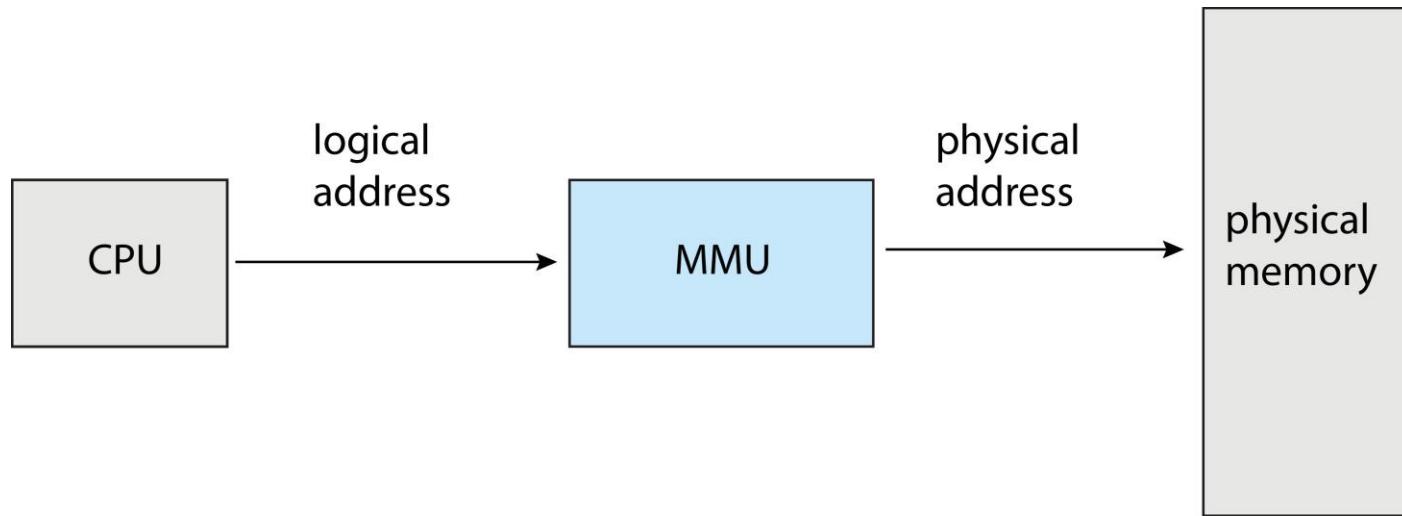
- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; they differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program





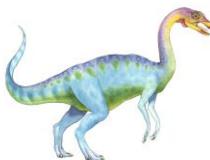
# Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address at run time



- Many methods possible, covered in the rest of this chapter

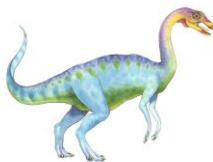




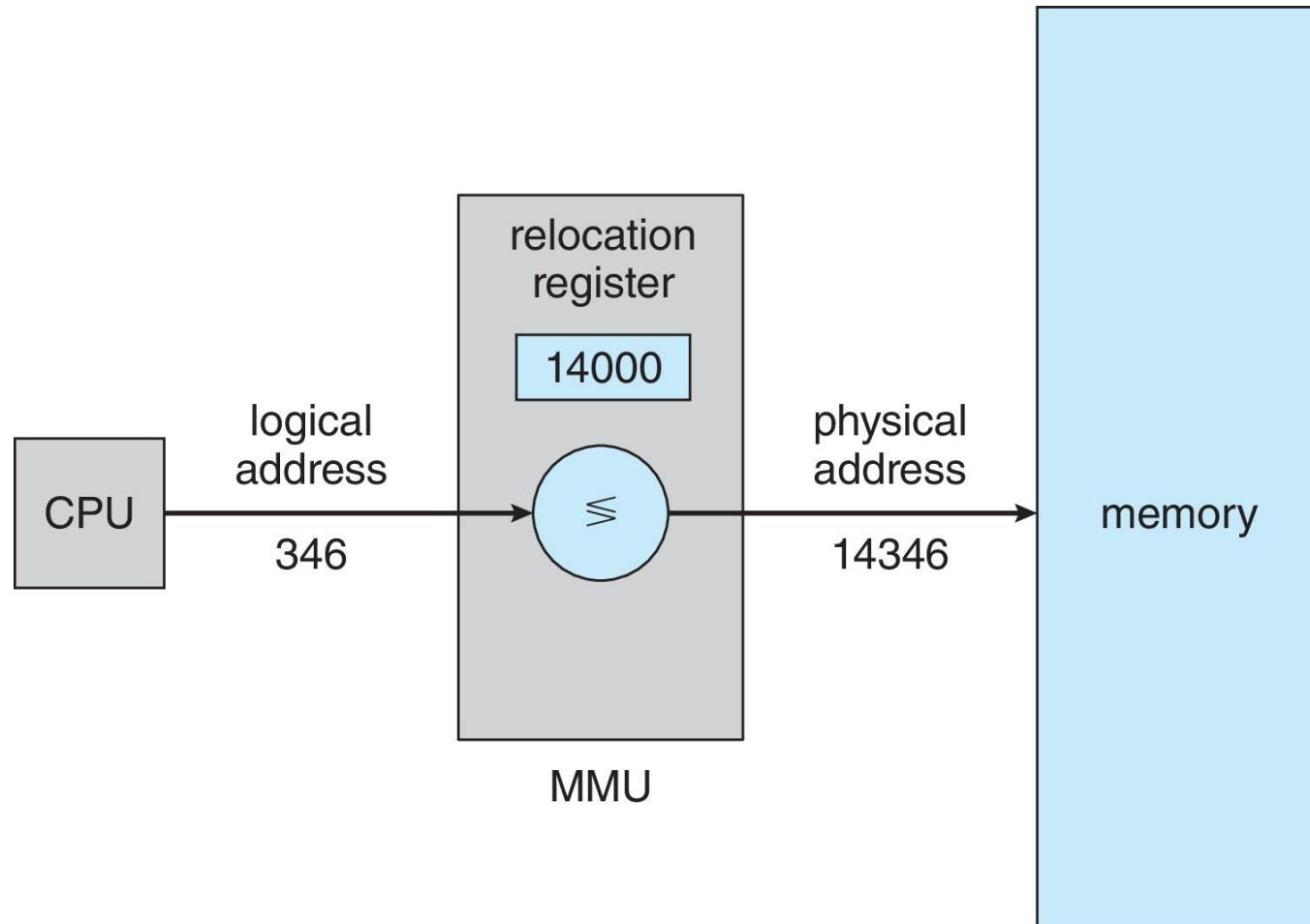
# Memory-Management Unit (Cont.)

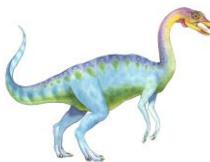
- Consider simple scheme, which is a generalization of the base-register scheme
  - The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses





# Memory-Management Unit (Cont.)





# Dynamic Loading

- Routine is not loaded until it is called
  - Better memory-space utilization; unused routine is never loaded
  - All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the OS is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading

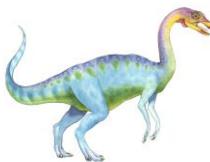




# Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking – linking postponed until execution time
  - Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
  - When executed, OS checks if routine is in processes' memory address
    - ▶ If not in address space, add to address space
  - Stub replaces itself with the address of the routine, and executes the routine
- Dynamic linking is particularly useful for libraries, also known as **shared libraries**
  - Consider applicability to patching system libraries
    - ▶ Versioning may be needed



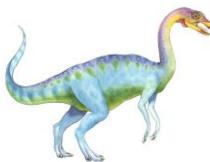


# Contiguous Allocation

---

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually divided into two **partitions**:
  - Resident OS, usually held in low memory with interrupt vector
  - User processes then held in high memory
    - ▶ Each process contained in single contiguous section of memory





# Contiguous Allocation (Cont.)

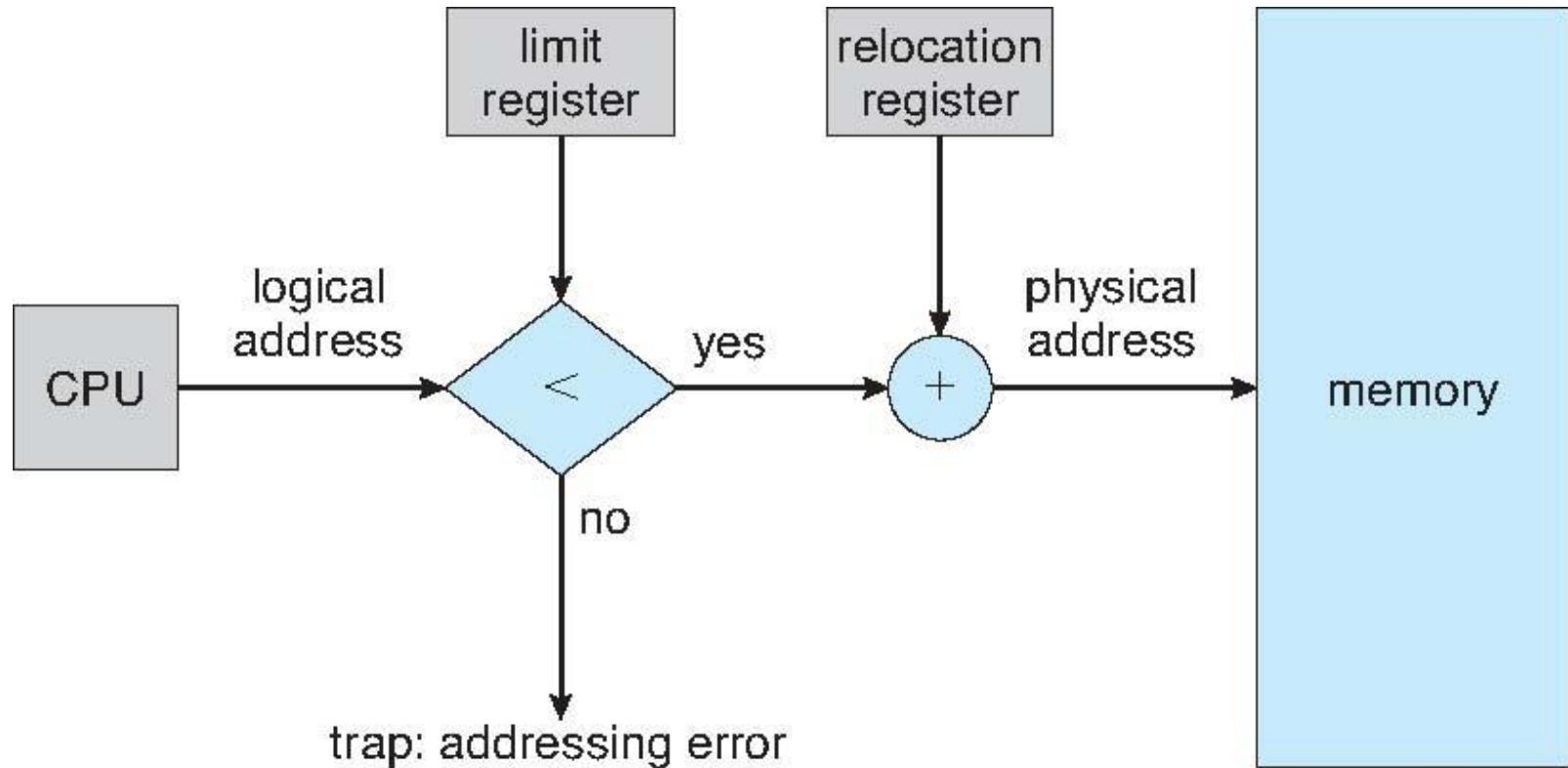
---

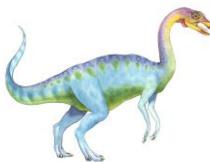
- Relocation registers used to protect user processes from each other, and from changing OS code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** and kernel changing size
    - ▶ E.g. for device driver not commonly used





# Hardware Support for Relocation and Limit Registers



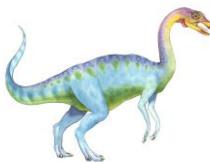


# Memory Allocation

---

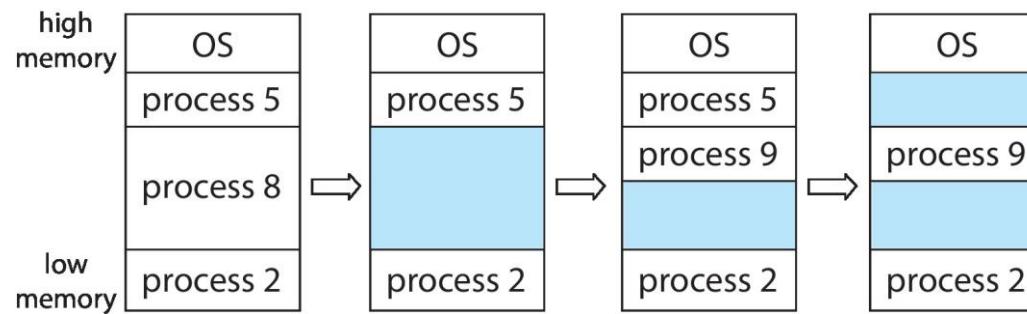
- Multiple-partition allocation
  - In the simplest method of **fixed-size** partition, degree of multiprogramming limited by number of partitions
    - ▶ It was used by IBM OS/360, but no longer in use
  - A generalization of fixed-partition scheme is used for batch environments, and also applicable to time-sharing systems for pure segmentation

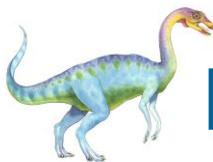




# Variable Partition

- **Variable-partition** sizes for efficiency (sized to process' needs)
  - **Hole** – block of available memory; holes of various sizes are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - OS maintains information about:
    - a) allocated partitions
    - b) free partitions (hole)





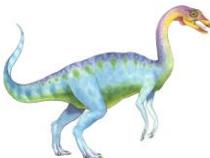
# Dynamic Storage-Allocation Problem

How to satisfy a request of size  $n$  from a list of free holes?

- **First-fit:** Allocate the *first* hole that is big enough
- **Best-fit:** Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit:** Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization





# Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given  $N$  blocks allocated,  $0.5 N$  blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

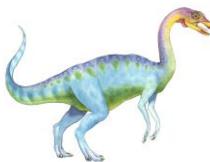




# Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - The cost is expensive by moving all processes toward one end of memory
- Another solution is to permit logical address space to be noncontiguous
  - Segmentation
  - Paging

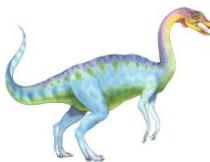




# Paging

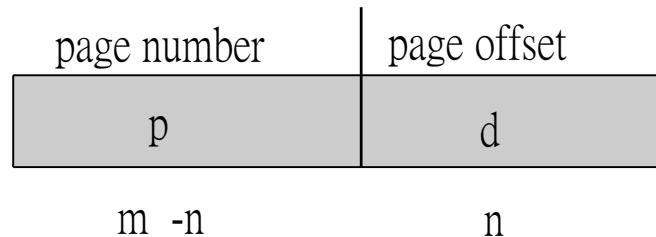
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever it is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
  - To run a program of size  **$N$**  pages, need to find  **$N$**  free frames and load program
  - Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have **Internal** fragmentation





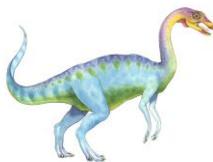
# Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number ( $p$ )** – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset ( $d$ )** – combined with base address to define the physical memory address that is sent to the memory unit

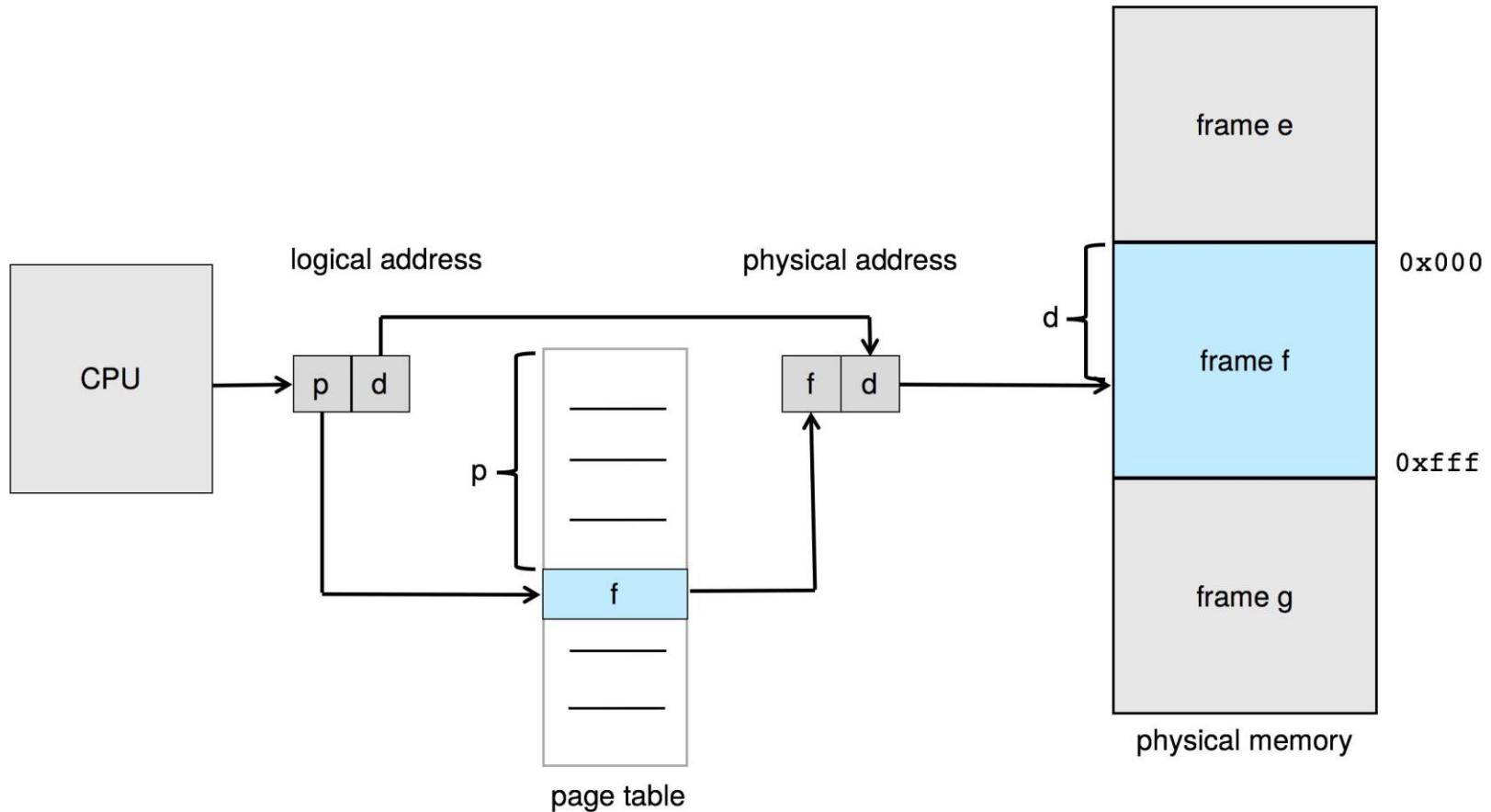


- For given logical address space  $2^m$  and page size  $2^n$



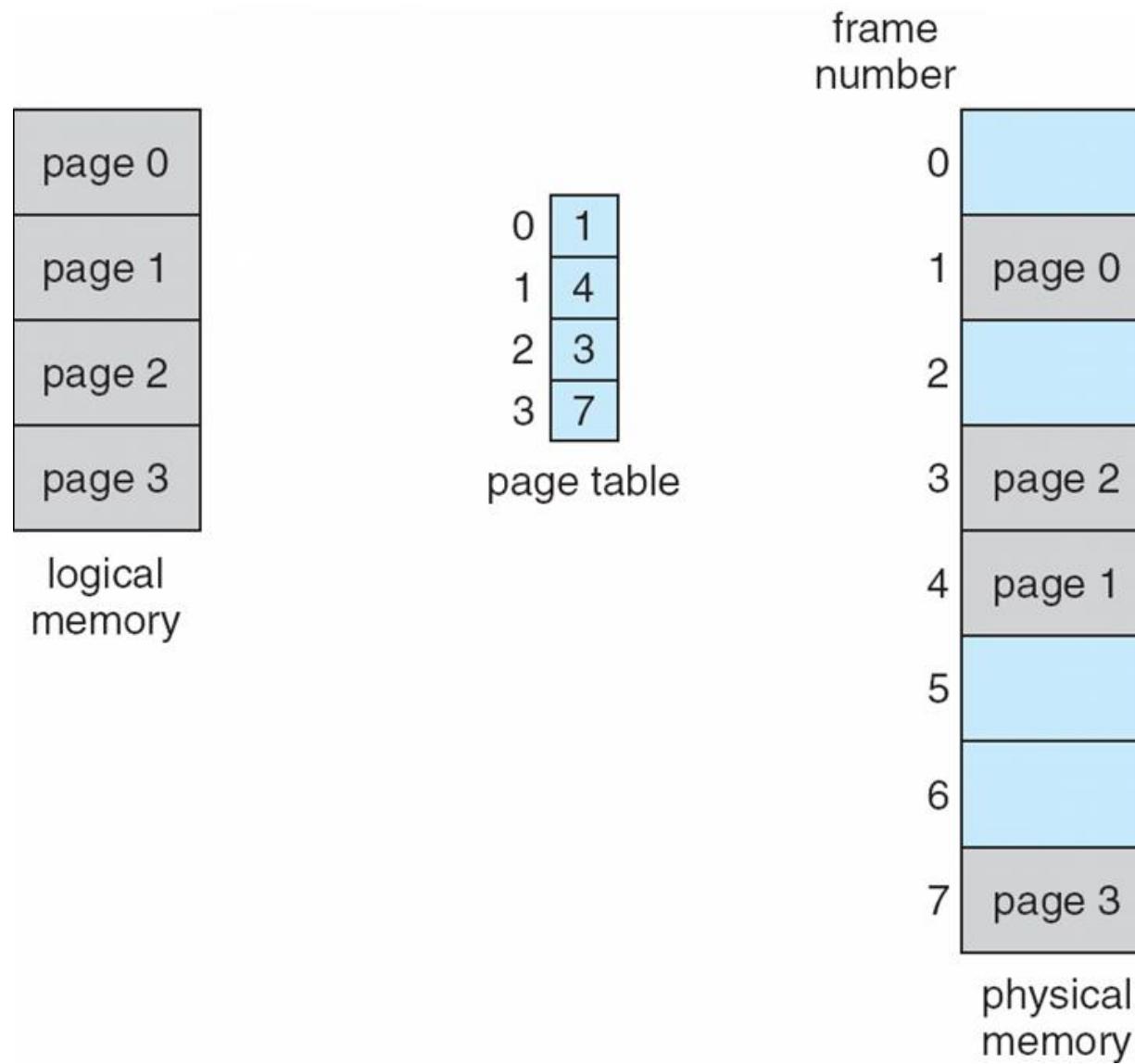


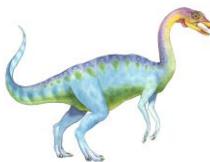
# Paging Hardware





# Paging Model of Logical and Physical Memory



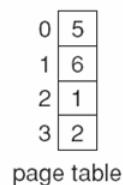


# Paging Example

- Logical address:  $n = 2$  and  $m = 4$ . Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

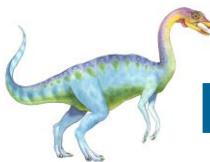
logical memory



0	
4	
8	
12	
16	
20	a
21	b
22	c
23	d
24	e
25	f
26	g
27	h
28	

physical memory





# Paging -- Calculating internal fragmentation

---

- Page size = 2,048 bytes
- Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of  $2,048 - 1,086 = 962$  bytes
- Worst case fragmentation = 1 frame – 1 byte
- On average fragmentation = 1 / 2 frame size
- So small frame sizes desirable?
  - But each page table entry takes memory to track
- Page sizes growing over time
  - Solaris supports two page sizes – 8 KB and 4 MB

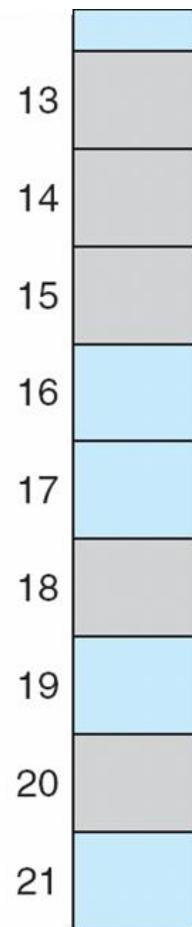
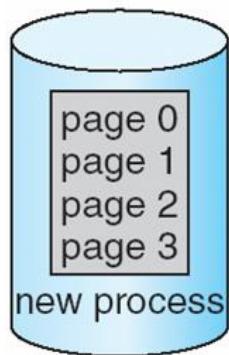




# Free Frames

free-frame list

14  
13  
18  
20  
15

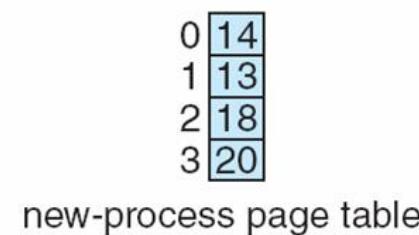
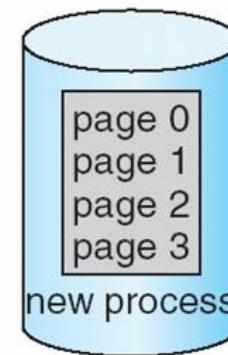


(a)

Before allocation

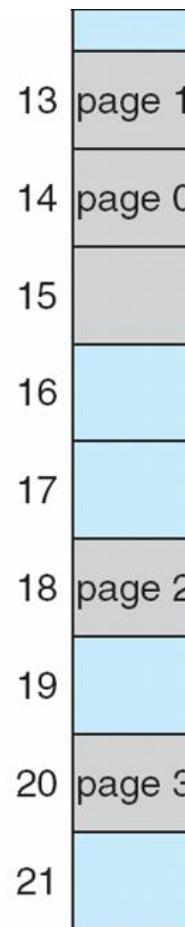
free-frame list

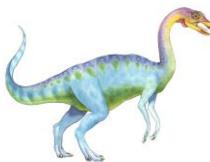
15



(b)

After allocation

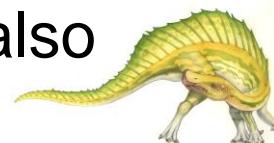


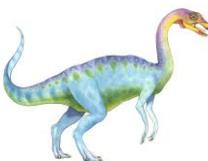


# Implementation of Page Table

---

- Page table is kept in main memory
  - **Page-table base register (PTBR)** points to the page table
  - **Page-table length register (PTLR)** indicates size of the page table
- In this scheme every data/instruction access requires **two** memory accesses
  - One for the page table and one for the data / instruction
- The two-memory access problem can be solved by the use of a special fast-lookup hardware cache called **translation look-aside buffers (TLBs)** (also called **associative memory**)





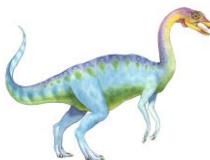
# TLB Hardware

- Associative memory – parallel search on key-value pairs

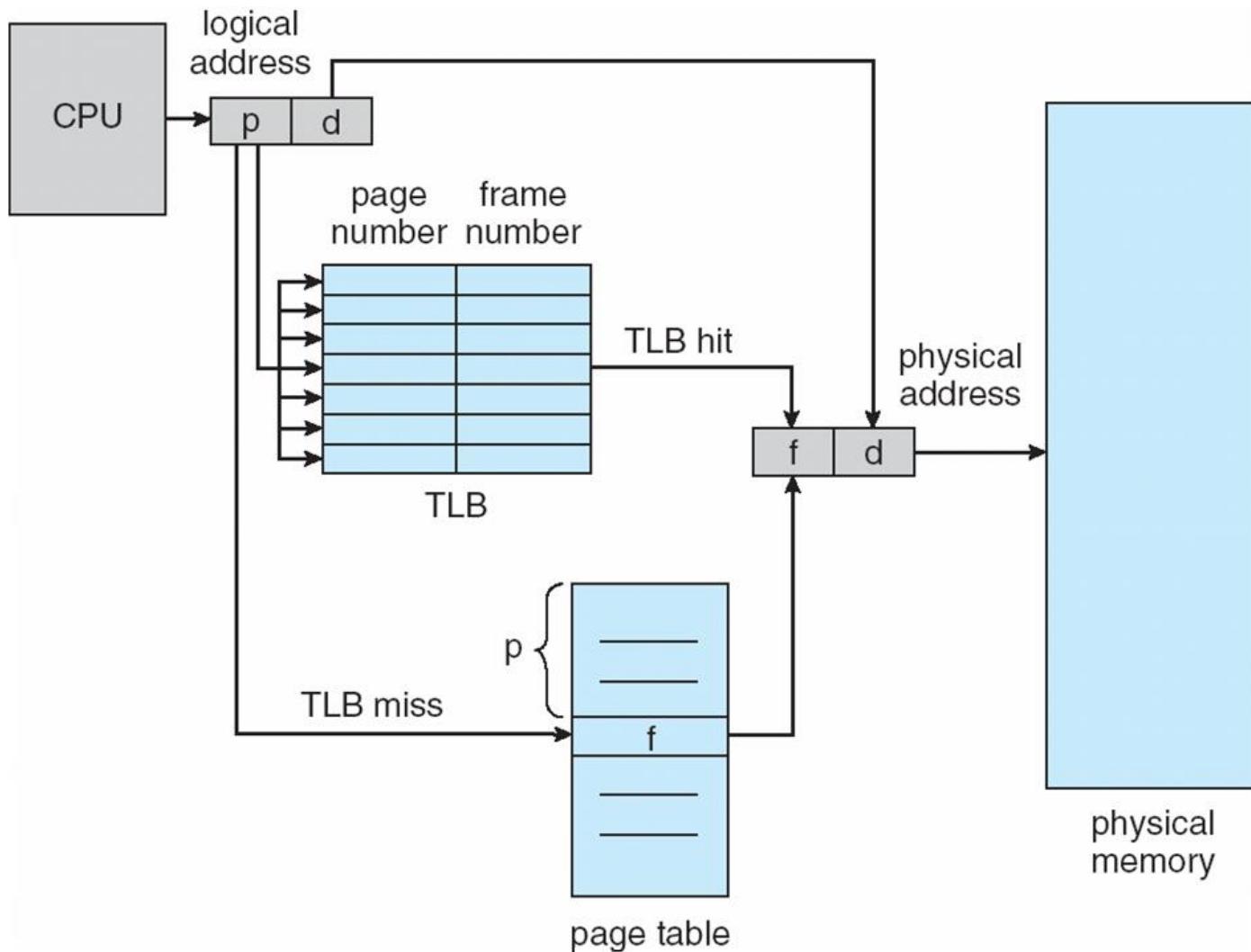
Page #	Frame #

- TLBs typically small (64 to 1,024 entries)
- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise, for a TLB miss, get frame # from page table in memory, and value is loaded into TLB for faster access next time





# Paging Hardware With TLB





# Effective Access Time

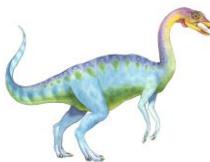
---

- Hit ratio – percentage of times that a page number is found in the TLB
  - An 80% hit ratio means that we find the desired page number in the TLB 80% of the time
- Suppose that 10 nanoseconds to access memory
  - If we find the desired page in TLB then a mapped-memory access take 10 ns
  - Otherwise we need two memory access so it is 20 ns
- Effective Access Time (EAT)
$$\text{EAT} = 0.80 \times 10 + 0.20 \times 20 = 12 \text{ nanoseconds}$$

implying 20% slowdown in access time
- Consider a more realistic hit ratio of 99%,
$$\text{EAT} = 0.99 \times 10 + 0.01 \times 20 = 10.1\text{ns}$$

implying only 1% slowdown in access time



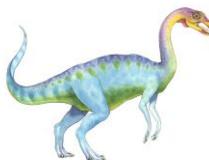


# Memory Protection

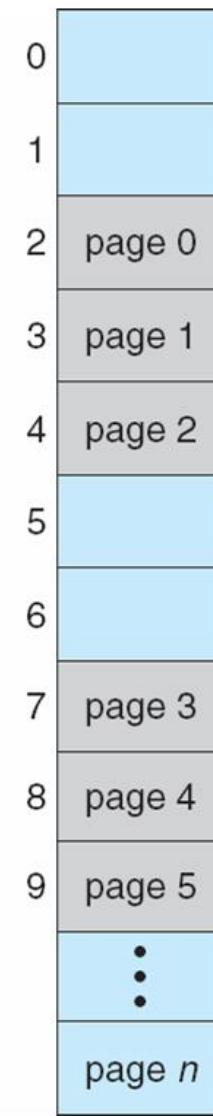
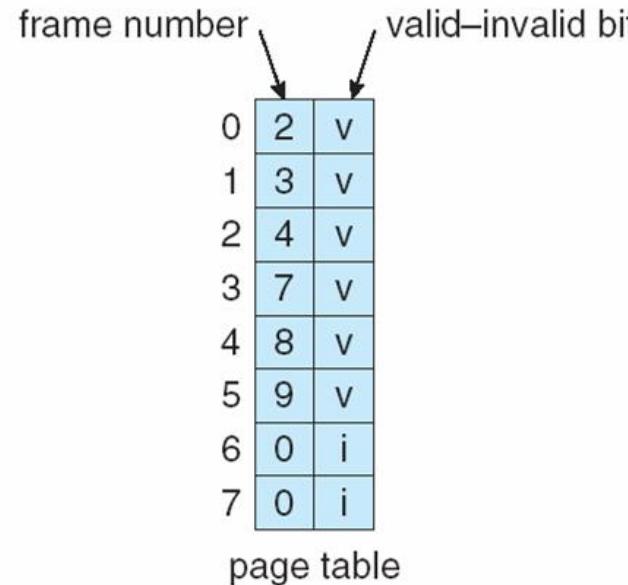
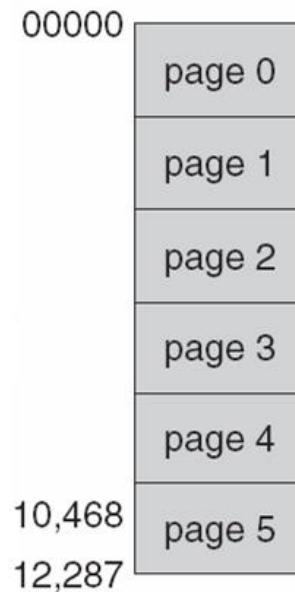
---

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
  - “invalid” indicates that the page is not in the process’ logical address space
  - Or use **page-table length register (PTLR)**
- Any violations result in a trap to the kernel





# Valid (v) or Invalid (i) Bit In A Page Table





# Shared Pages

---

## ■ Shared code

- One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
- Similar to multiple threads sharing the same process space
- Also useful for interprocess communication if sharing of read-write pages is allowed

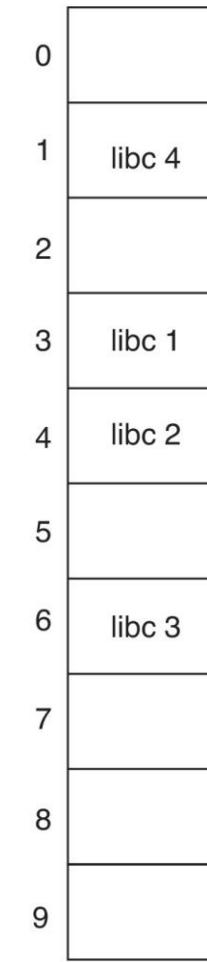
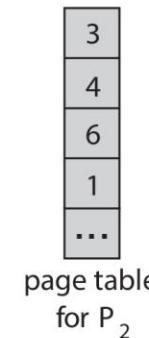
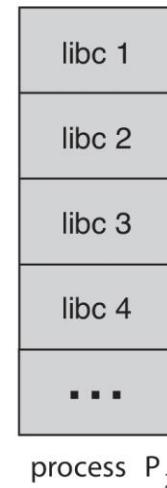
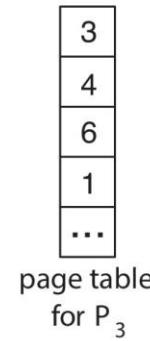
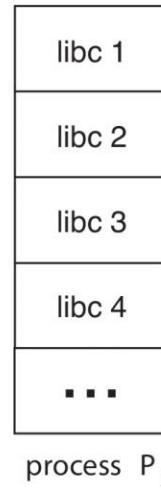
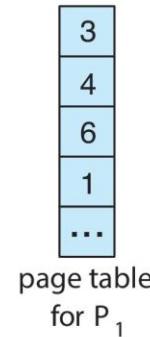
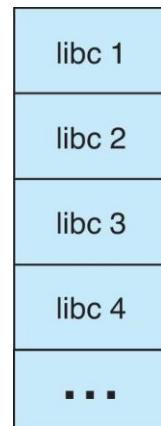
## ■ Private code and data

- Each process keeps a separate copy of the code and data
- The pages for the private code and data can appear anywhere in the logical address space





# Shared Pages Example

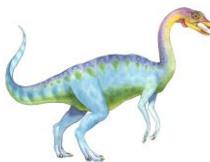




# Structure of the Page Table

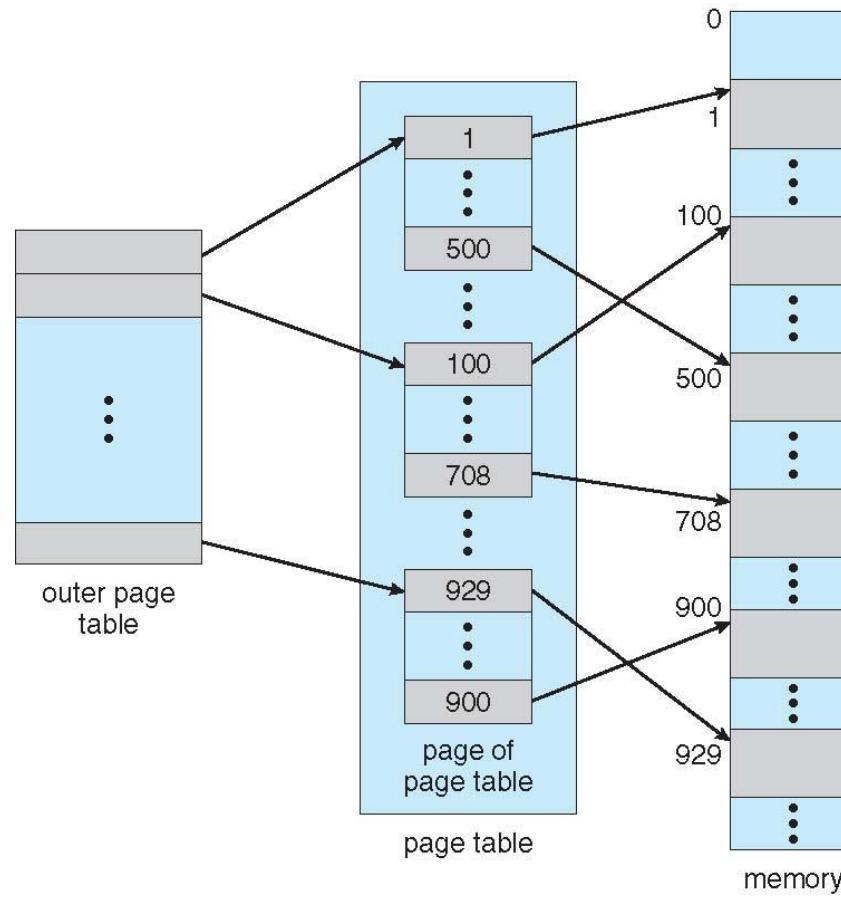
- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address
    - ▶ Page size of 4 KB ( $2^{12}$ )
    - ▶ Page table would have 1 million entries ( $2^{32} / 2^{12}$ )
    - ▶ If each entry is 4 bytes → each process 4 MB of physical address space for the page table alone
      - Don't want to allocate that contiguously in main memory
  - One simple solution is to divide the page table into smaller units
    - ▶ **Hierarchical Paging**
    - ▶ **Hashed Page Tables**
    - ▶ **Inverted Page Tables**

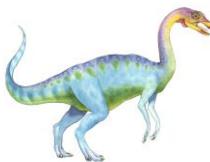




# Hierarchical Page Tables

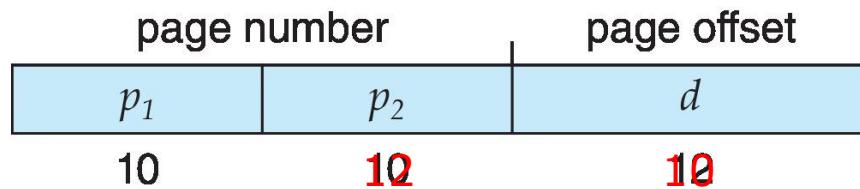
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table





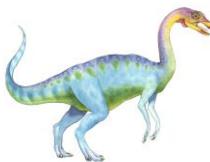
# Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 12-bit page offset
- Thus, a logical address is as follows:

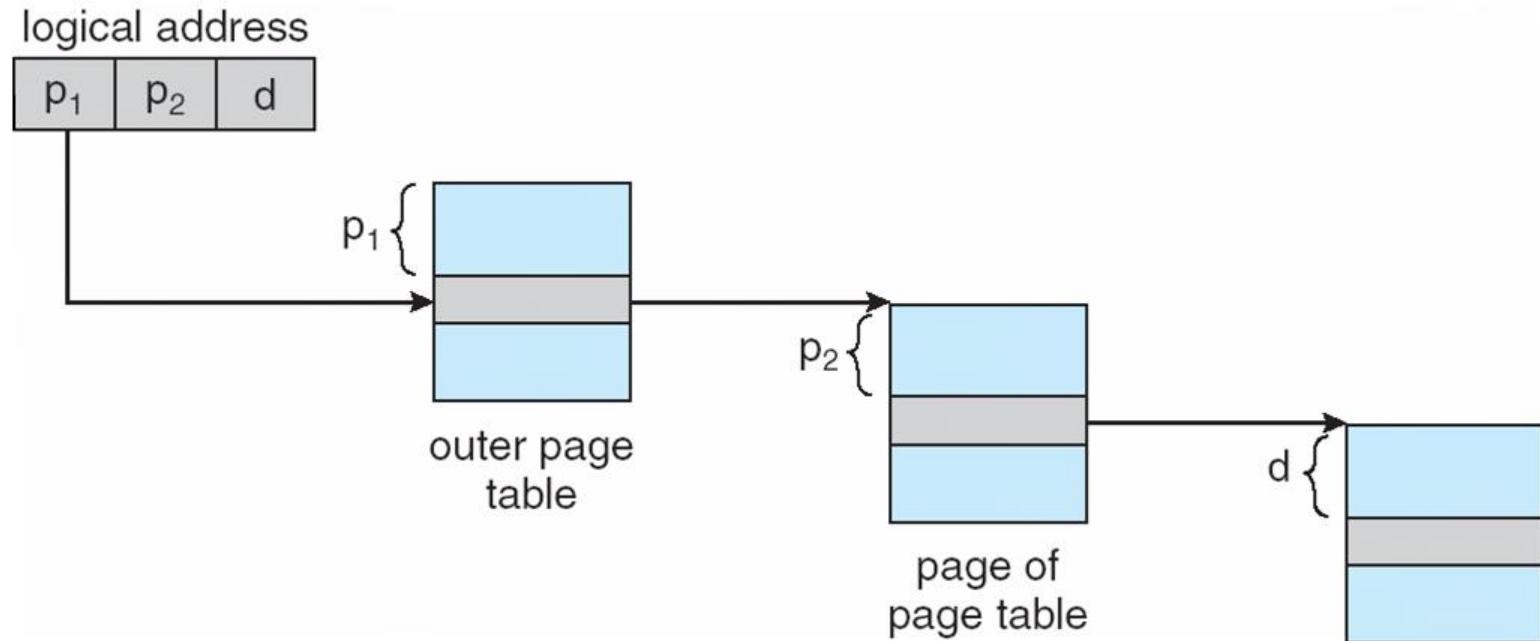


- where  $p_1$  is an index into the outer page table, and  $p_2$  is the displacement within the page of the inner page table
- Known as **forward-mapped page table**





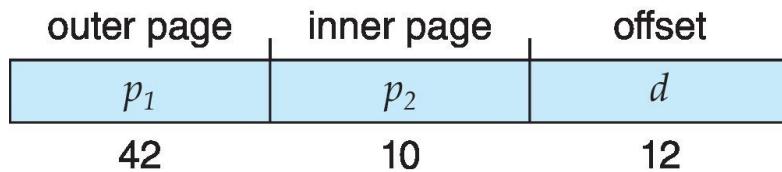
# Address-Translation Scheme





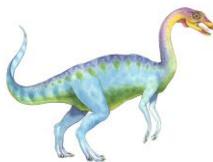
# 64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB ( $2^{12}$ )
  - Then page table has  $2^{52}$  entries
  - If two level scheme, inner page tables could be  $2^{10}$  4-byte entries
  - Address would look like

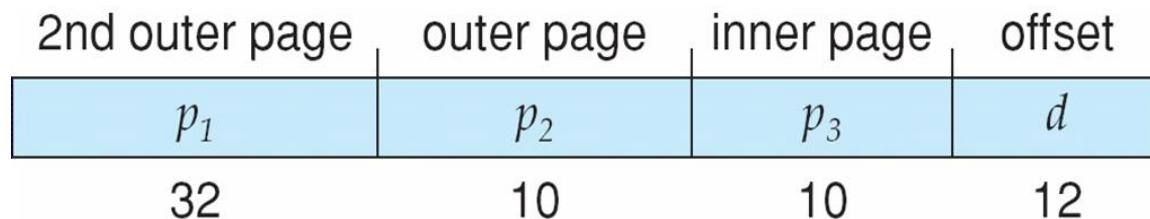
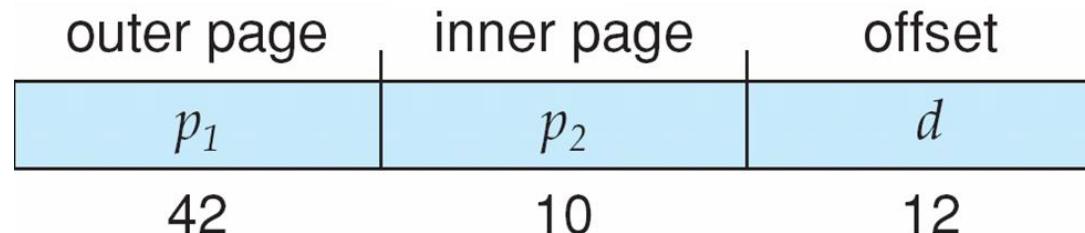


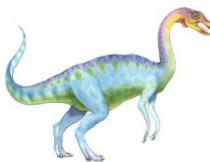
- Outer page table has  $2^{42}$  entries or  $2^{44}$  bytes
- One solution is to add a 2<sup>nd</sup> outer page table
- But in the following example the 2<sup>nd</sup> outer page table is still  $2^{34}$  bytes in size
  - ▶ And possibly 4 memory accesses to get to one physical memory location





# Three-level Paging Scheme



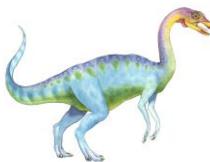


# Hashed Page Tables

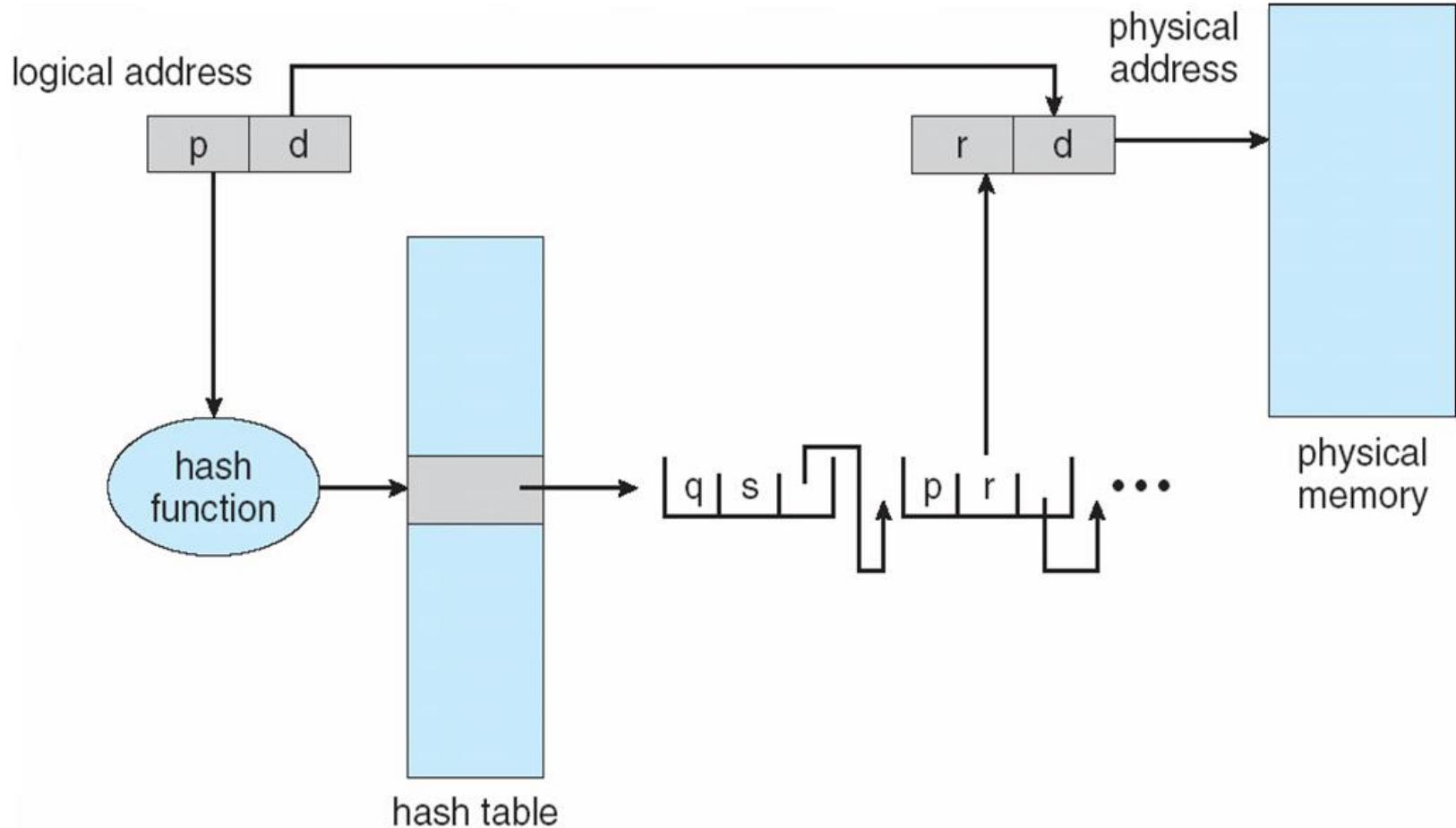
---

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)





# Hashed Page Table

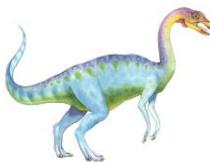




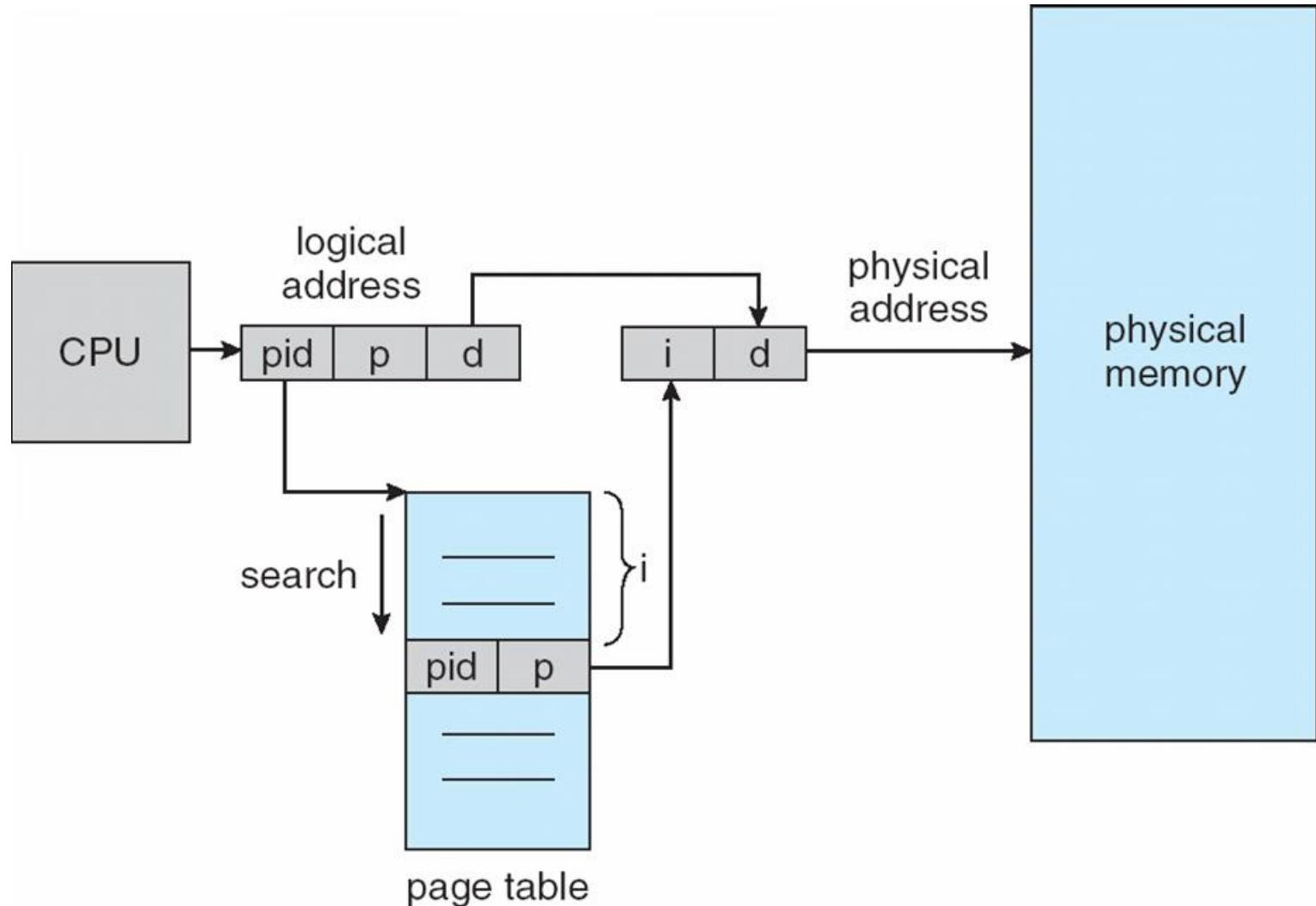
# Inverted Page Table

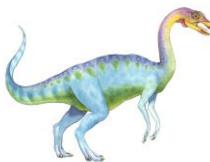
- Rather than each process having a page table and keeping track of all possible logical pages, we track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address





# Inverted Page Table Architecture



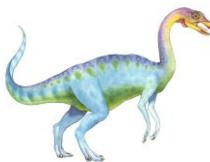


# Oracle SPARC Solaris

---

- Consider modern, 64-bit OS example with tightly integrated HW
  - Goals are efficiency, low overhead
- Based on hashing, but more complex
- Two hash tables
  - One for kernel and one for all user processes
  - Each maps memory addresses from virtual to physical memory
  - Each entry represents a contiguous area of mapped virtual memory
    - ▶ More efficient than having a separate hash-table entry for each page
  - Each entry has base address and span (indicating the number of pages the entry represents)





# Oracle SPARC Solaris (Cont.)

- TLB holds translation table entries (TTEs) for fast hardware lookups
  - A cache of TTEs reside in a translation storage buffer (TSB)
    - ▶ Includes an entry per recently accessed page
- Virtual address reference causes TLB search
  - If miss, hardware walks the in-memory TSB looking for the TTE corresponding to the address
    - ▶ If match found, the CPU copies the TSB entry into the TLB and translation completes
    - ▶ If no match found, kernel interrupted to search the hash table
      - The kernel then creates a TTE from the appropriate hash table and stores it in the TSB, Interrupt handler returns control to the MMU, which completes the address translation





# Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought **back** into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk



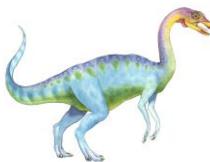


# Swapping (Cont.)

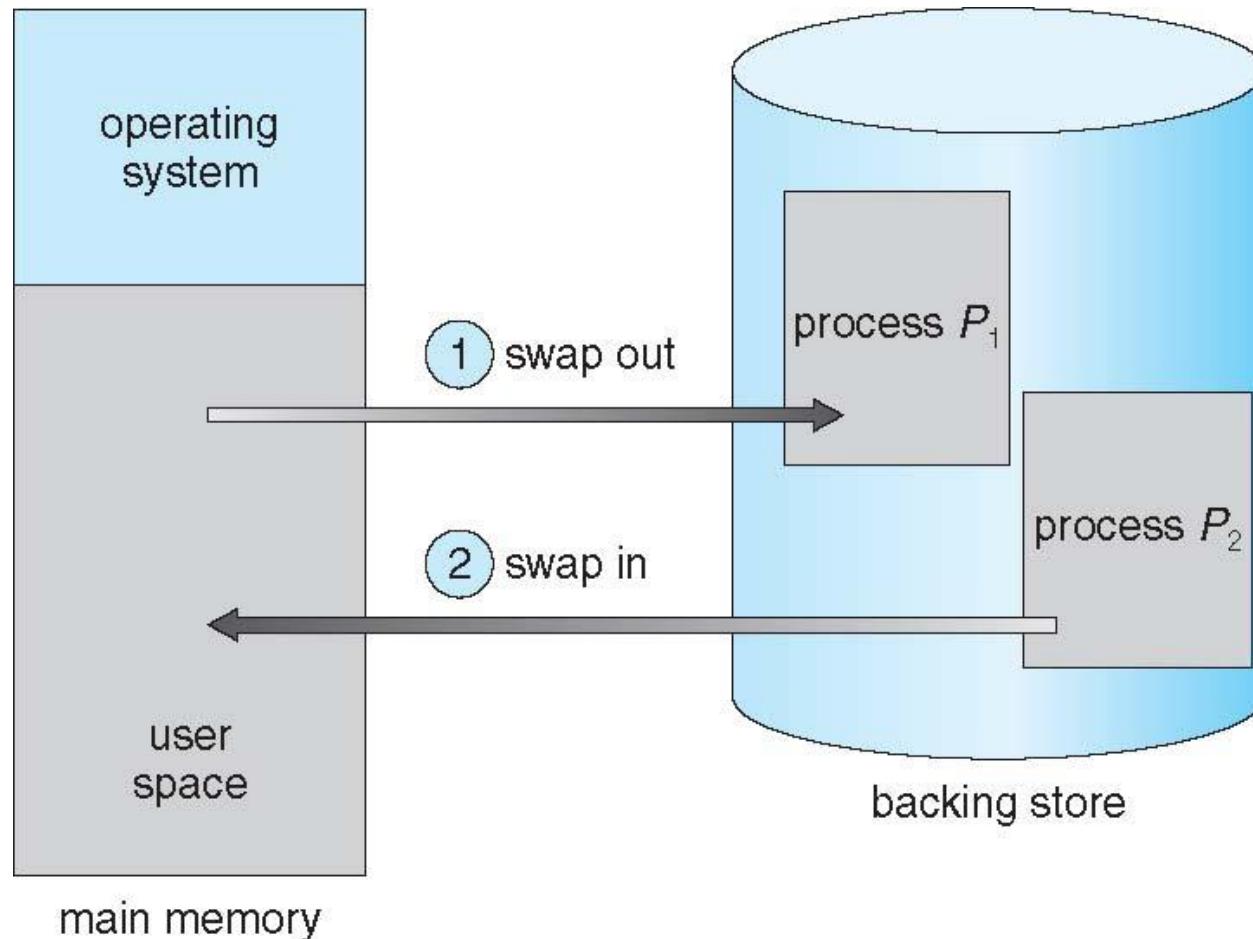
---

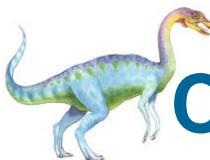
- Does the swapped out process need to swap back in to same physical addresses?
  - Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold





# Schematic View of Swapping

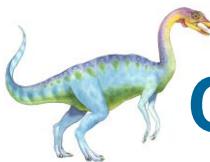




# Context Switch Time including Swapping

- If next process to be put on CPU is not in memory, need to swap out a process and swap in target process
  - Context switch time can then be very high
  - 100MB process swapping to hard disk with transfer rate of 50MB/sec
    - ▶ Swap out time of 2000 ms
    - ▶ Plus swap in of same sized process
    - ▶ Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`

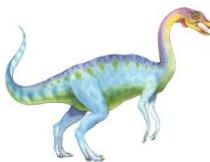




# Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - ▶ Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
  - But modified version common
    - ▶ Swap only when free memory extremely low

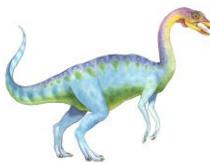




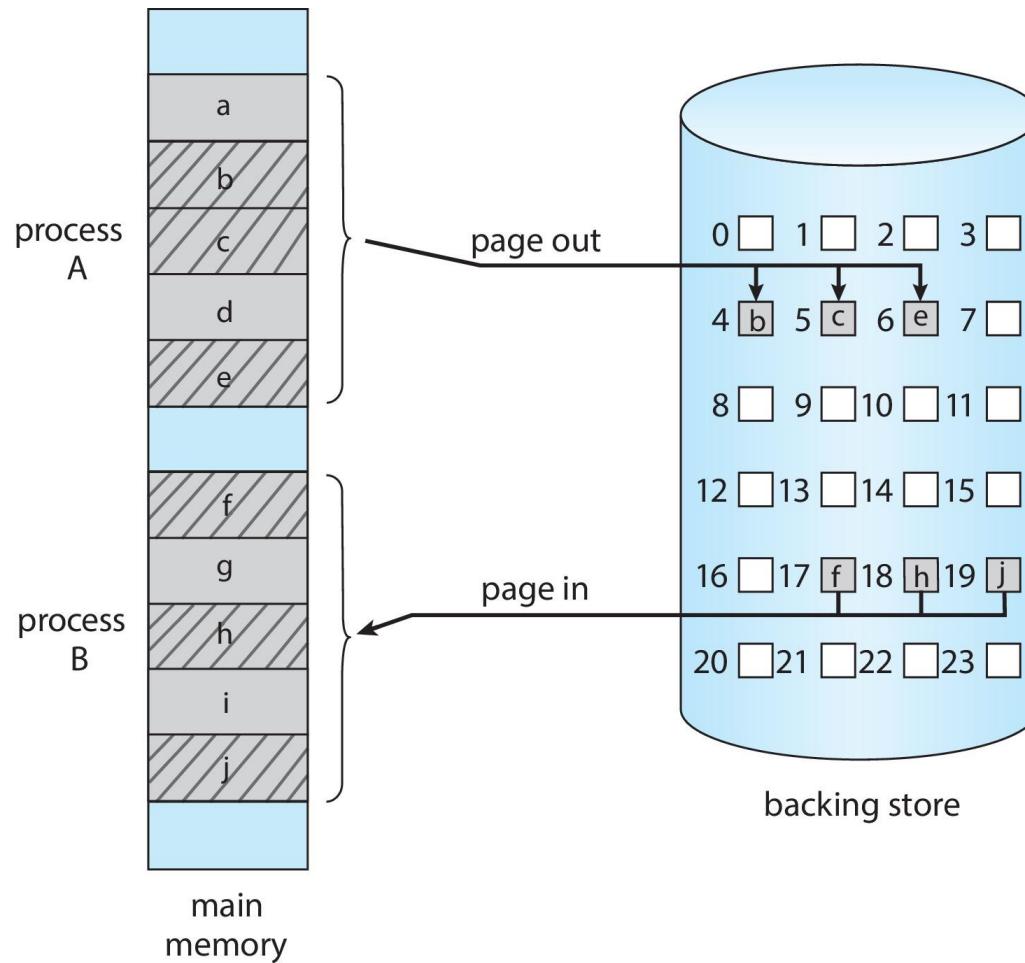
# Swapping on Mobile Systems

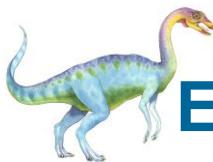
- Not typically supported
  - Flash memory based
    - ▶ Small amount of space
    - ▶ Limited number of write cycles
    - ▶ Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS **asks** apps to voluntarily relinquish allocated memory
    - ▶ Read-only data thrown out and reloaded from flash if needed
    - ▶ Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support paging as discussed below





# Swapping with Paging





## Example: The Intel 32 and 64-bit Architectures

---

- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here



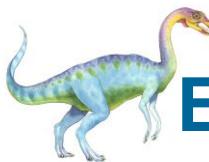


## Example: The Intel IA-32 Architecture

---

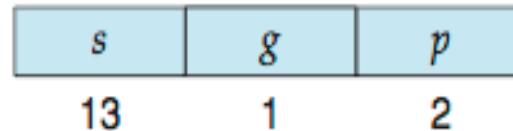
- Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB
  - Up to 16 K segments per process
  - Divided into two partitions
    - ▶ First partition of up to 8K segments are private to process (kept in **local descriptor table (LDT)**)
    - ▶ Second partition of up to 8K segments shared among all processes (kept in **global descriptor table (GDT)**)





## Example: The Intel IA-32 Architecture (Cont.)

- CPU generates logical address
  - Selector given to segmentation unit
    - ▶ Which produces linear addresses

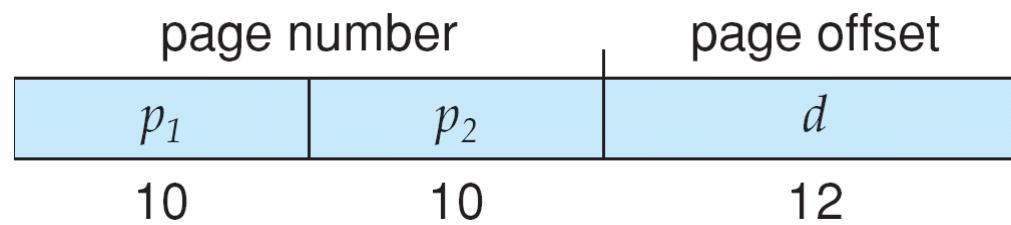
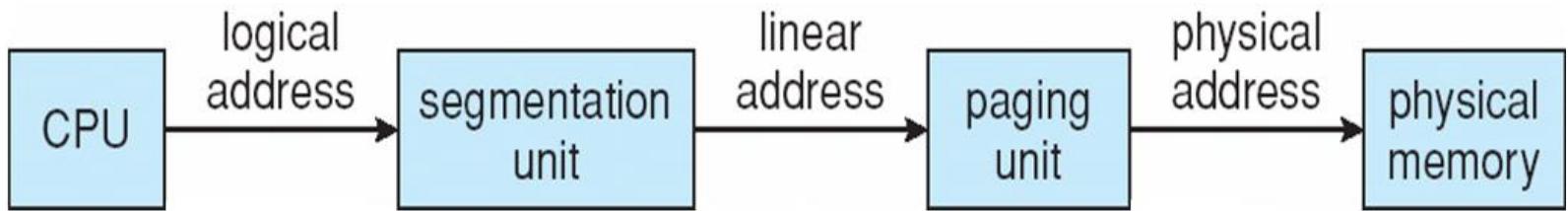


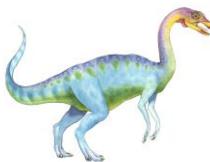
- Linear address given to paging unit
  - ▶ Which generates physical address in main memory
  - ▶ Paging units form equivalent of MMU
  - ▶ Pages sizes can be 4 KB or 4 MB



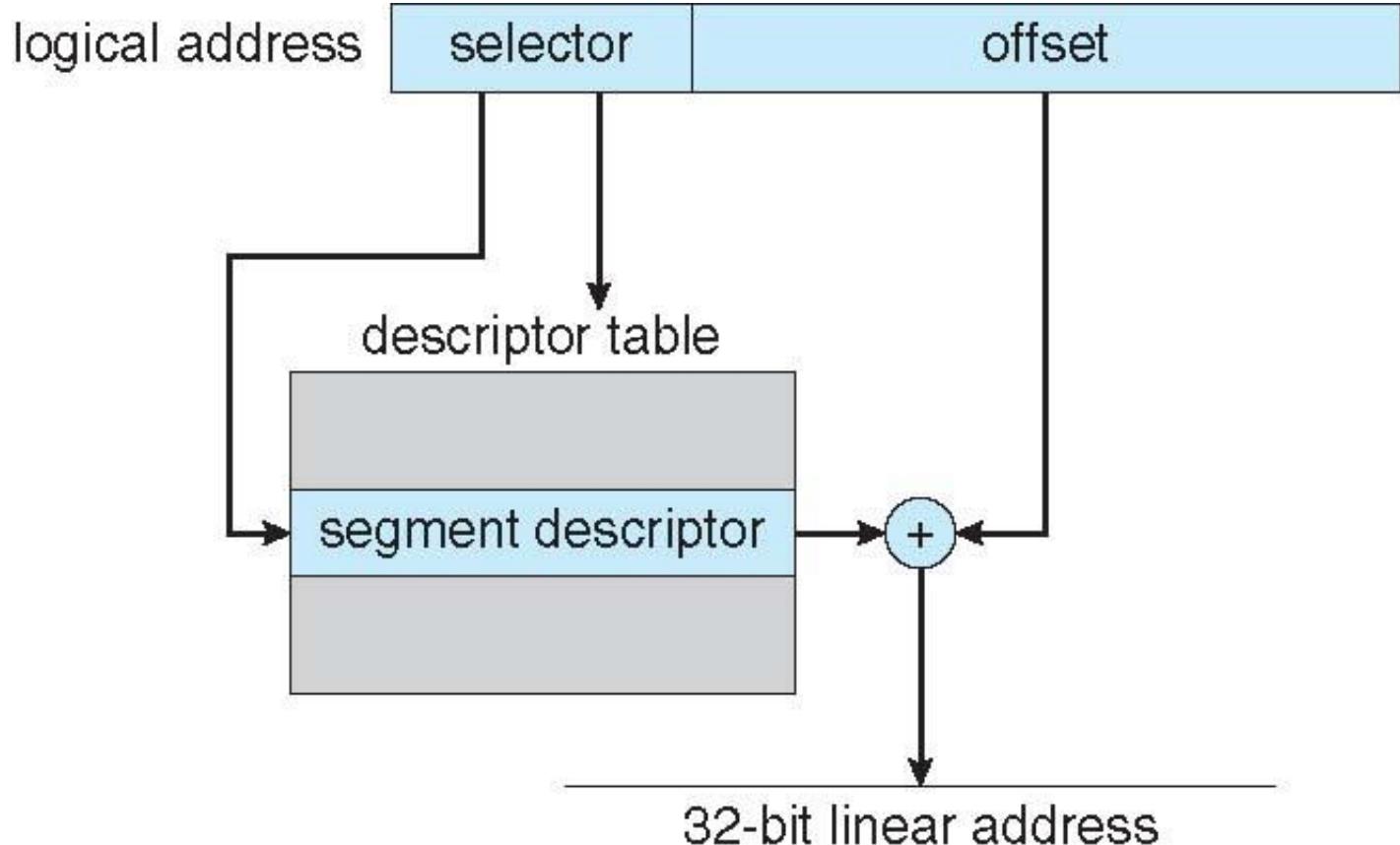


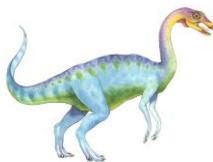
# Logical to Physical Address Translation in IA-32



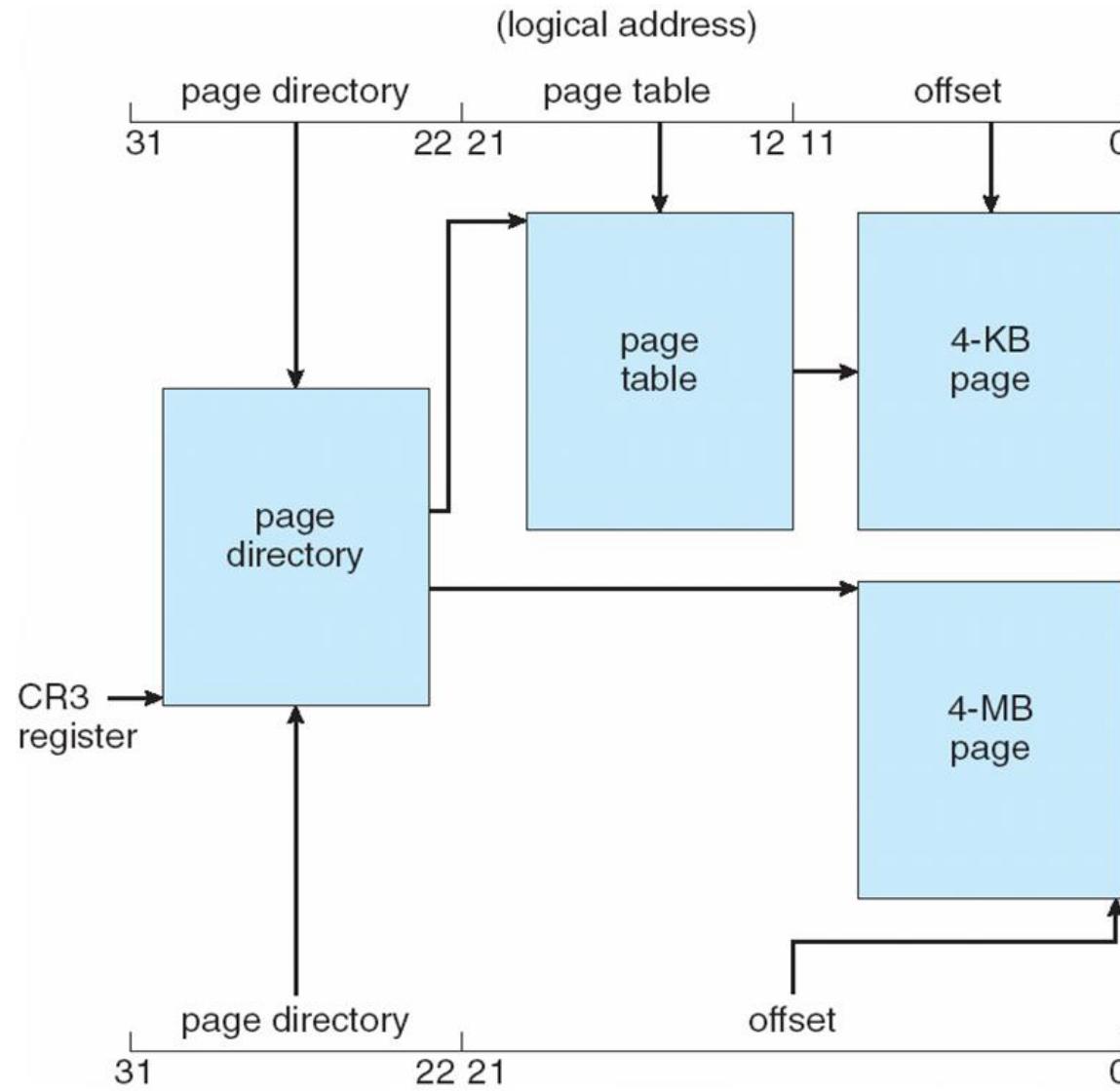


# Intel IA-32 Segmentation





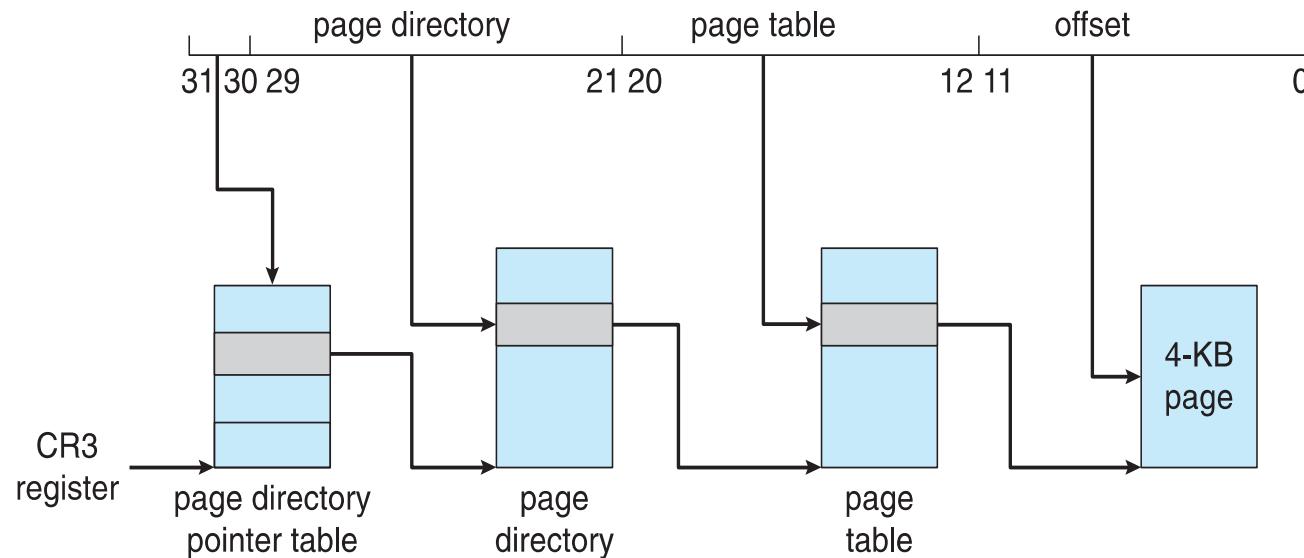
# Intel IA-32 Paging Architecture





# Intel IA-32 Page Address Extensions

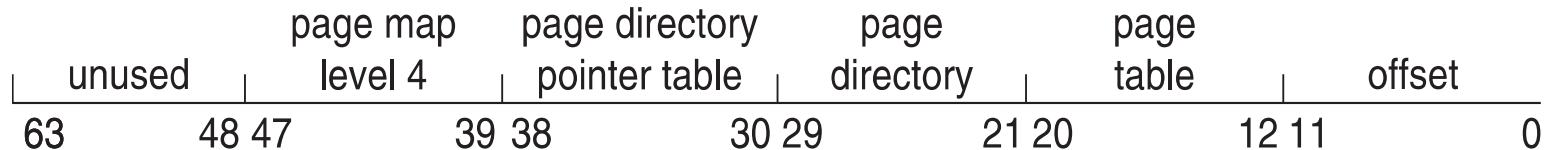
- 32-bit address limits led Intel to create **page address extension (PAE)**, allowing 32-bit apps access to more than 4GB memory space
  - Paging went to a 3-level scheme
  - Top two bits refer to a **page directory pointer table**
  - Page-directory and page-table entries moved to 64-bits
  - Net effect is increasing address space to 36 bits – 64GB of physical memory





# Intel x86-64

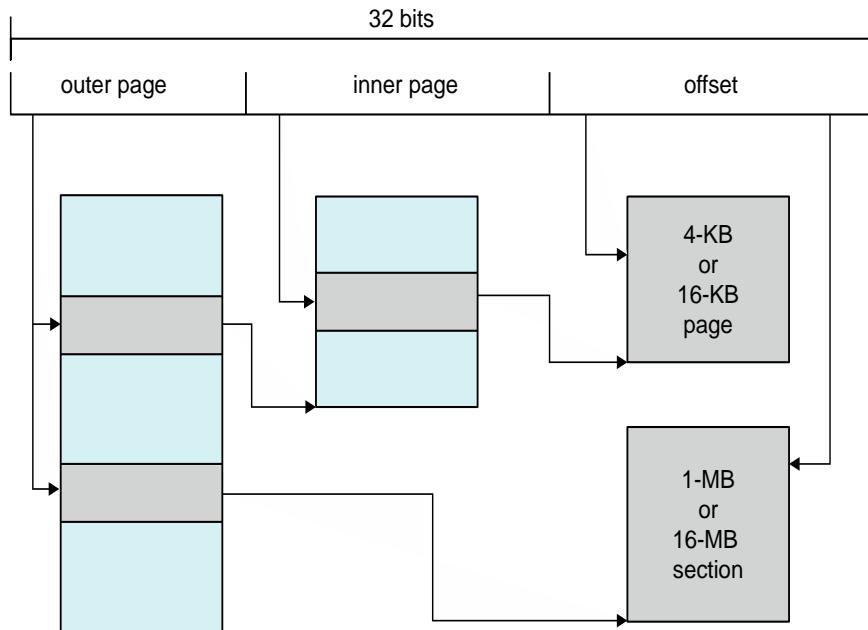
- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits





# Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
  - Modern, energy efficient, 32-bit CPU
  - 4 KB and 16 KB pages
  - 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



# End of Chapter 9

