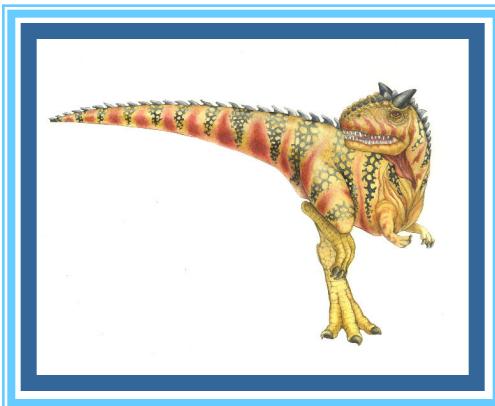
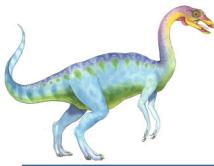


Chapter 1: Introduction





Chapter 1: Introduction

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





Objectives

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





What is an Operating System?

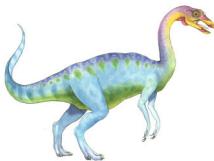
■ A program that acts as an intermediary between a user of a computer and the computer's hardware

- Manage the hardware resources
- Support applications; games, business, programs, ..., etc
- Designed to optimize hardware utilization

■ 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 (printer, keyboard, monitor, etc)
- 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, ..., etc
- Users
 - ▶ People, machines, other computers

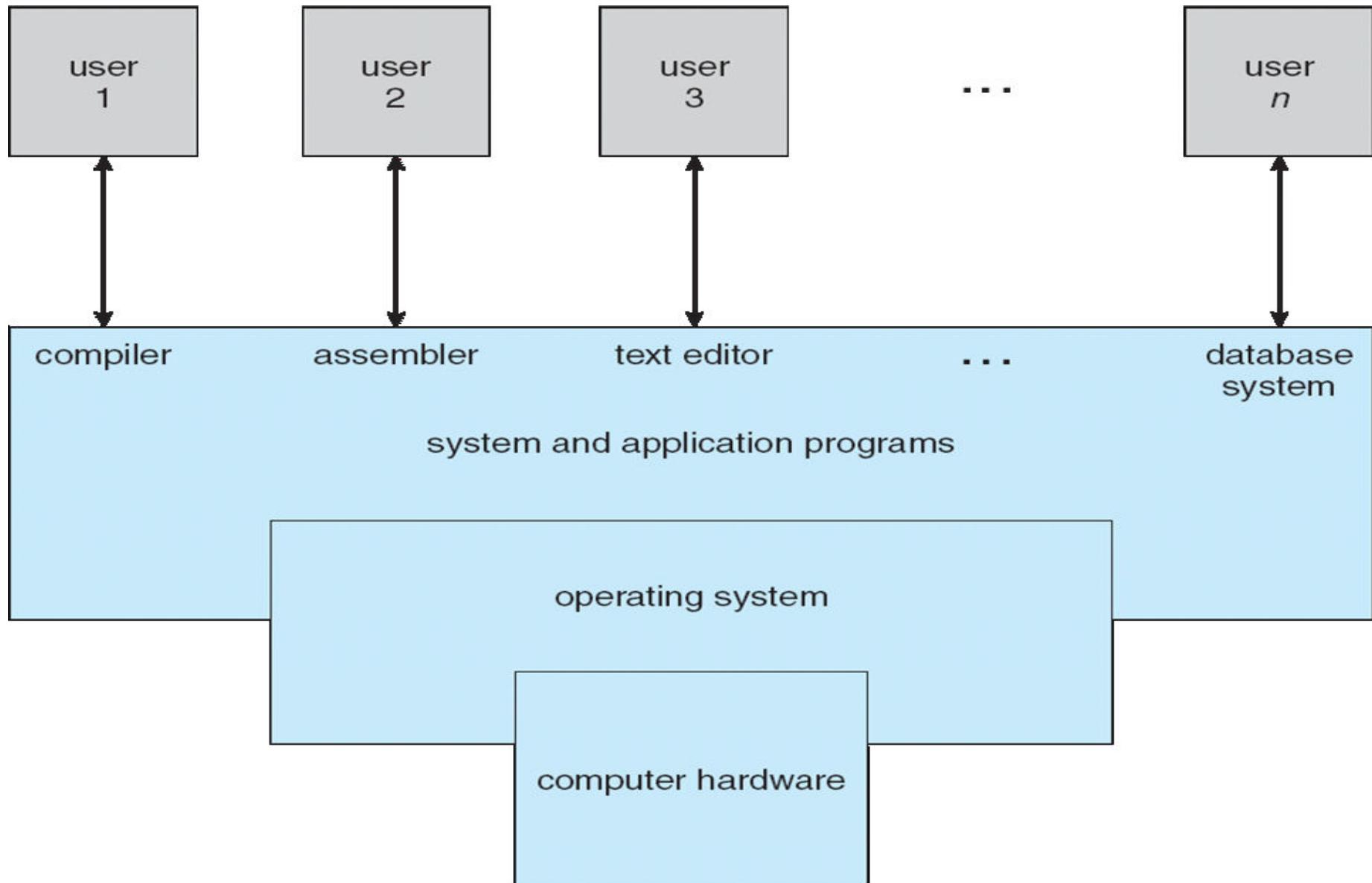
■ Computer system = hardware, software, and data.

- Operating system provides means for proper uses of these resources





Four Components of a Computer System





What Operating Systems Do (User View)

- PC users want convenience, **ease of use** and **good performance**
 - Don't care about **resource utilization**, but to maximize work
- But shared computer such as **mainframe** or **minicomputer** must keep all users happy
 - OS maximizes resource utilization, assures efficient use of resources
- Users of dedicate systems such as **workstations** have dedicated resources but frequently use shared resources from **servers**
 - OS compromises between individual usability and resource utilization
- Handheld computers are resource poor, optimized for usability and battery life
- Some computers have little or no user interface, such as embedded computers in devices and automobiles





What Operating Systems Do (System View)

■ OS is a resource allocator

- Manages all resources
 - ▶ CPU time, memory space, I/O devices, ..., etc
- Decides between numerous and/or conflicting requests for efficient and fair resource use
 - ▶ When many users access to the same resources

■ OS is a control program

- Controls execution of programs to prevent errors and improper use of the computer
 - ▶ For example: Fetch-and-Execute Cycle of the CPU





Operating System Definition

- No universally accepted definition
 - OSs exist to create a usable computing system.
- “Everything a vendor ships when you order an operating system” is a good approximation
 - But varies wildly
- OS commonly defined as the **kernel**.
 - That is: “**the one program running at all times on the computer**”.
- Everything else is either
 - a system program (ships with the operating system, ...) , or
 - an application program.





Computer-System Architecture

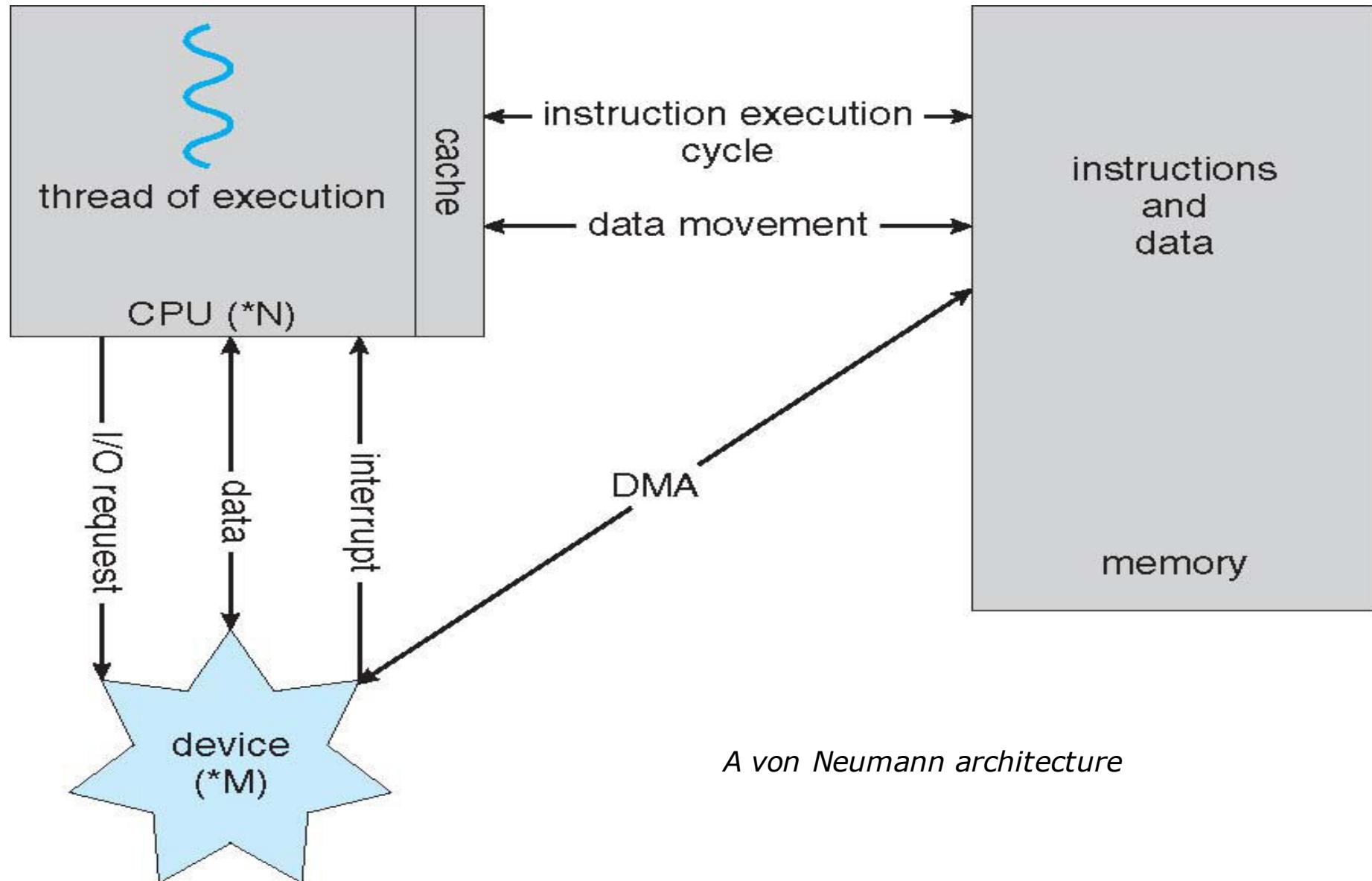
- Most systems use a single general-purpose processor, **one main CPU**
 - Most **other** systems have special-purpose processors as well
 - ▶ Device-specific processors, e.g. **graphic controllers, disk controllers, ... etc**
 - ▶ Relieves the main CPU of overheads of secondary tasks

- **Multiprocessors** systems growing in use and importance
 - Also known as **parallel systems, tightly-coupled systems**
 - Advantages include:
 1. **Increased throughput:** N CPUs with speedup $< N$
 2. **Economy of scale:** cost less than multiple single CPU systems
 3. **Increased reliability** – graceful degradation or fault tolerance
 - Two types:
 1. **Asymmetric Multiprocessing** – 1 master CPU and many slaves CPUs
 2. **Symmetric Multiprocessing** – each processor performs all tasks



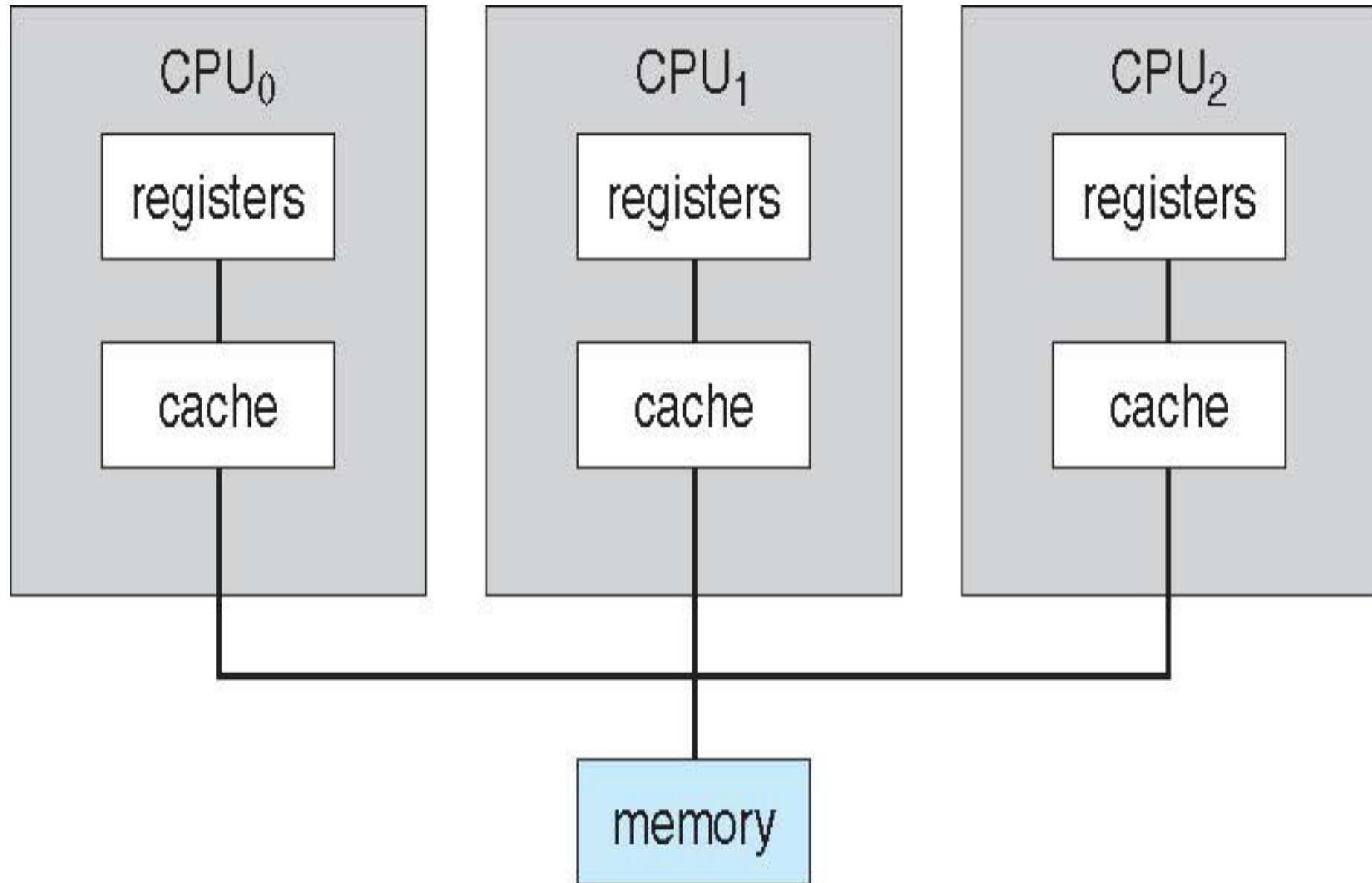


How a Modern Computer Works





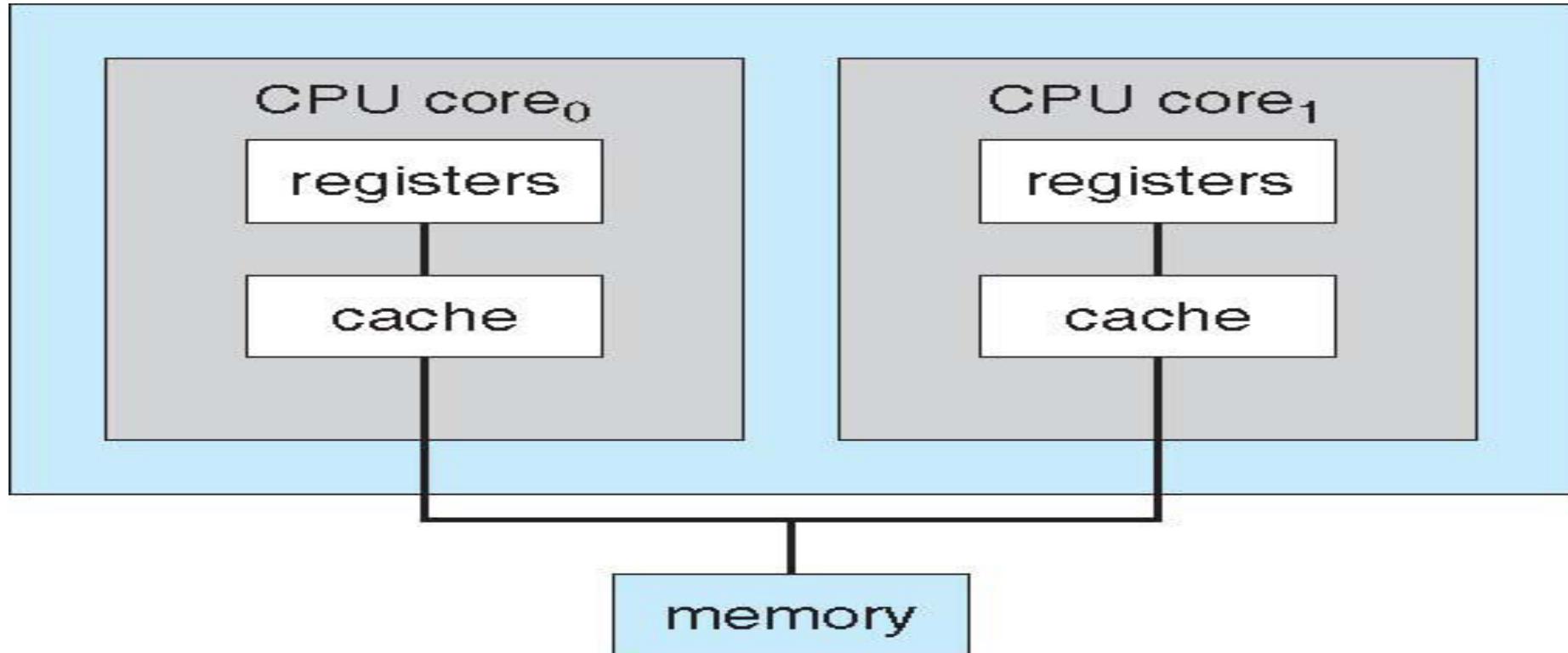
Symmetric Multiprocessing Architecture





A Dual-Core Design of SMP's

- Multi-chip; with single computing core or multiple computing cores
- **Multicore**; multiple computing cores on a single chip
- Systems containing all chips
 - Chassis containing multiple separate systems





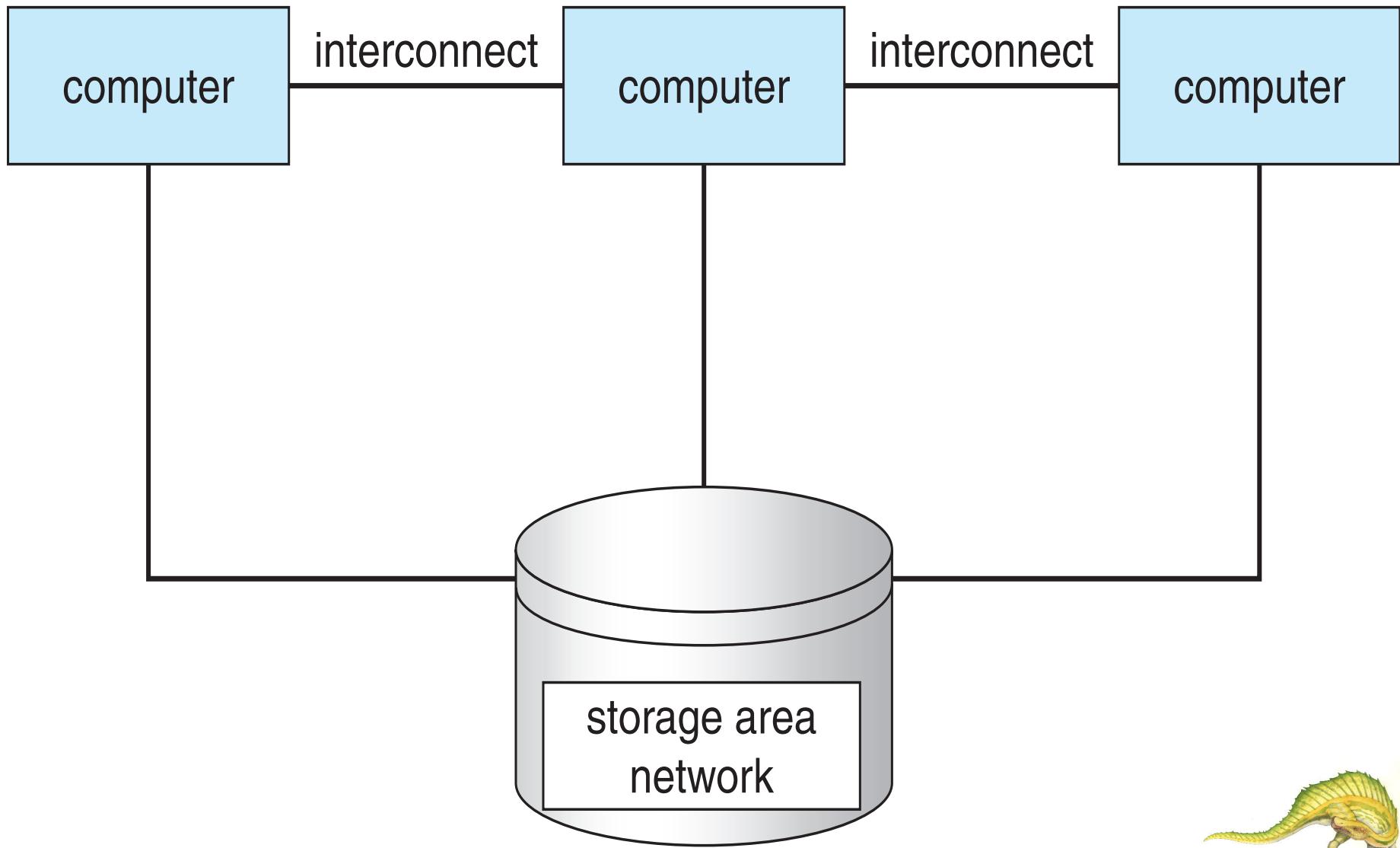
Clustered Systems

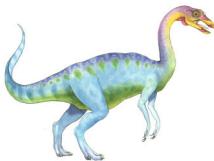
- Like multiprocessor systems, but multiple systems working together
 - Usually sharing storage via a **storage-area network (SAN)**
 - ▶ **Cluster nodes are closely linked via a fast interconnect (LAN, ...)**
 - Provides a **high-availability** service which survives failures
 - » Achieved by adding a level redundancy in the system
 - ▶ **Asymmetric clustering** has one machine in hot-standby mode
 - ... does nothing but monitor the active server which run applications
 - ▶ **Symmetric clustering** has multiple nodes running applications, monitoring each other
 - ... it is more efficient since all available nodes are used
 - 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
 - ▶ E.g., Oracle Parallel Server





Clustered Systems





Operating System Structure

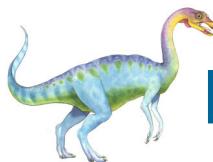
■ Multiprogramming (Batch system) needed for efficiency

- Single user cannot keep CPU and I/O devices busy at all times
- Multiprogramming organizes jobs (code and data) so CPU always has one to execute
- A subset of total jobs (the **job pool** on disk) in system is kept in memory
- One job selected and run via **job scheduling**
- When it has to wait (for I/O for example), OS switches to another job

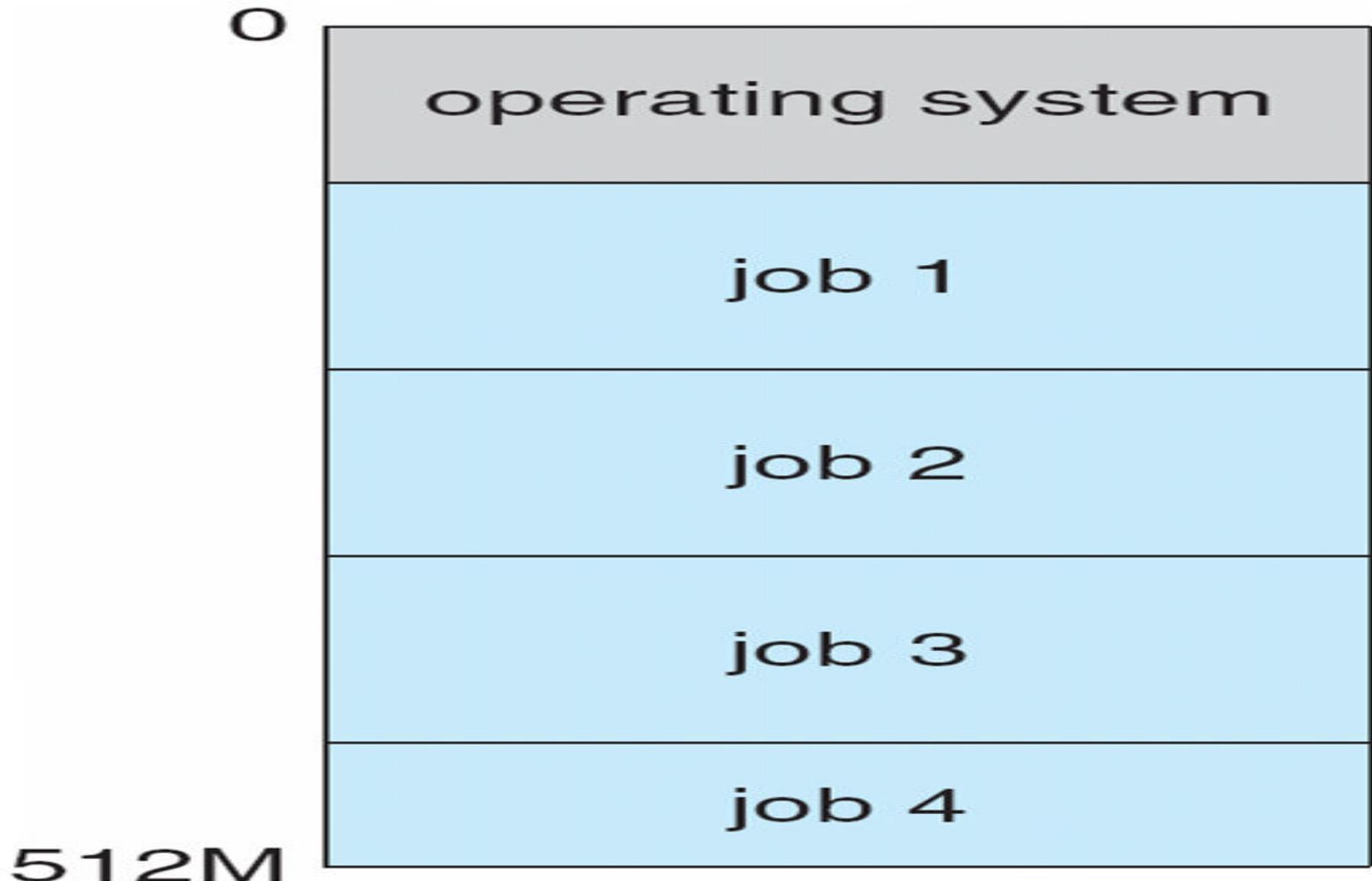
■ Timesharing (multitasking) is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing

- **Response time** should be < 1 second
- Each user has at least one program executing in memory \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





Operating-System Operations

■ Interrupt driven (hardware and software), i.e. event signals

- Hardware interrupt by one of the devices
- Software interrupt (**exception** or **trap**):
 - ▶ Software error (e.g., division by zero)
 - ▶ Request for operating system service
 - ▶ Other process problems include infinite loop, processes modifying each other or the operating system

■ OS determines action to take for an interrupt

- Interrupt Service Routine (ISR) in the interrupt vector





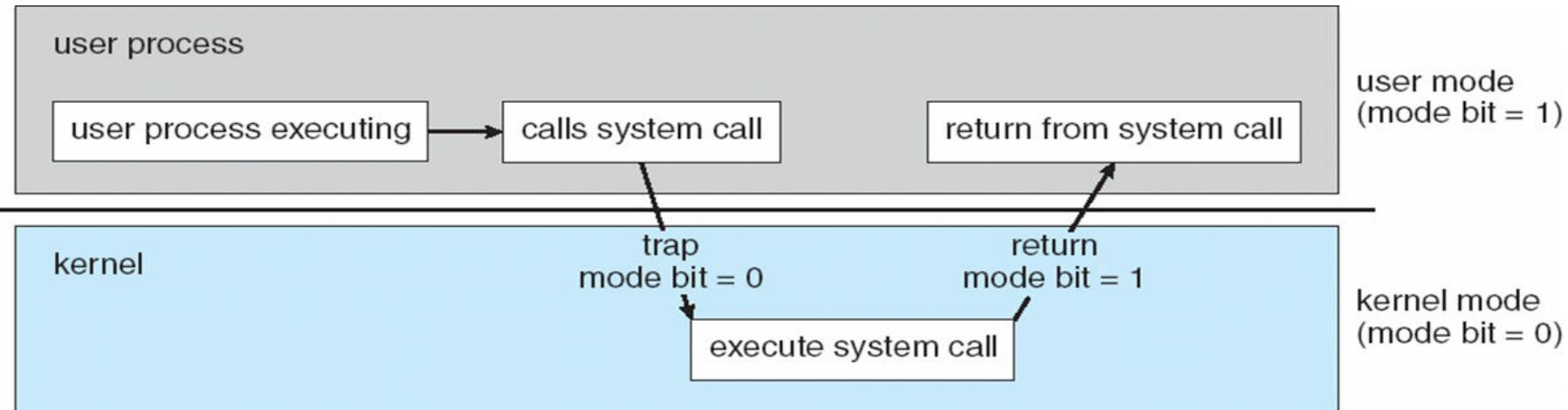
Operating-System Operations (cont.)

- Dual-mode operation allows OS to protect itself and other system components
 - ▶ To distinguish between the execution of OS code and user-defined code
 - User mode and kernel mode
 - Mode bit provided by hardware: kernel (0) or user (1)
 - ▶ Provides ability to distinguish when system is running user code or kernel code
 - ▶ Some instructions designated as privileged, only executable in kernel mode: e.g., instruction to switch to user mode, timer instructions, ... etc
 - ▶ System-call changes mode to kernel, return from the call resets it to user
 - ▶ At boot time:
 - Hardware starts in k-mode, OS is loaded then start u-apps in u-mode
- Increasingly CPUs support multi-mode operations
 - i.e. virtual machine manager (VMM) mode for guest VMs



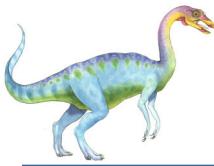


Transition from User to Kernel Mode



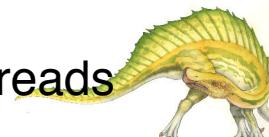
- Timer to prevent infinite loop / process hogging resources
 - ▶ Timer ensures that the OS maintains control over the CPU
 - Timer is set to interrupt the computer after some time period
 - Keep a counter that is decremented by the physical clock.
 - Operating system set the counter (privileged instruction)
 - When counter zero generate an interrupt
 - Set up before scheduling process to regain control or terminate program that exceeds allotted time





Process Management

- A process is a program in execution. It is a unit of work within the system.
Program is a ***passive entity***, process is an ***active entity***.
 - More than 1 process may be associated with the same program
- Process needs resources to accomplish its task
 - CPU, memory, I/O, files
 - Initialization data
- Process termination requires reclaim of any reusable resources
- Single-threaded process has one **program counter** specifying location of next instruction to execute
 - Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
 - Concurrency by multiplexing the CPUs among the processes / threads



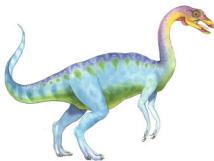


Process Management Activities

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

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

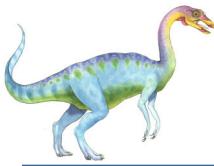




Memory Management

- To execute a program all (or part) of the instructions must be in memory
 - All (or part) of the data that is needed by a process must be in memory.
- Memory management determines what is in memory and when
 - Optimizing CPU utilization and computer response to users
 - ▶ Why? ... several programs are in memory
 - ▶ Hence ... need for efficient and effective memory management schemes
- 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

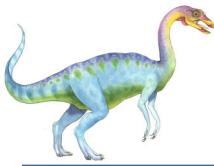




File-System Management

- OS provides uniform, logical view of information storage
 - Abstracts physical properties to logical storage unit - **file**
 - ▶ **File = collection of related information**
 - Each medium is controlled by device (i.e., disk drive, tape drive)
 - Types: **magnetic disk, optical disk, magnetic tape, ... etc**
 - ▶ Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- File-System management
 - Files usually organized into directories
 - Access control on most systems to determine who can access what
 - OS activities include
 - ▶ Creating and deleting files and directories
 - ▶ Primitives to manipulate files and directories: **read, write, append, ... etc**
 - ▶ Mapping files onto secondary storage
 - ▶ Backup files onto stable (non-volatile) storage media

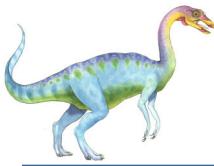




Mass-Storage Management

- Usually disks are 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
 - Secondary storage used frequently, and hence, must be used efficiently
- OS activities
 - Free-space management
 - Storage allocation
 - Disk scheduling
- Some storage need not be fast
 - Tertiary storage includes optical storage, magnetic tape
 - ▶ Slower and lower in cost than secondary storage, e.g., back-ups
 - Still must be managed – by OS or applications
 - Varies between WORM (write-once, read-many-times) and RW (read-write)





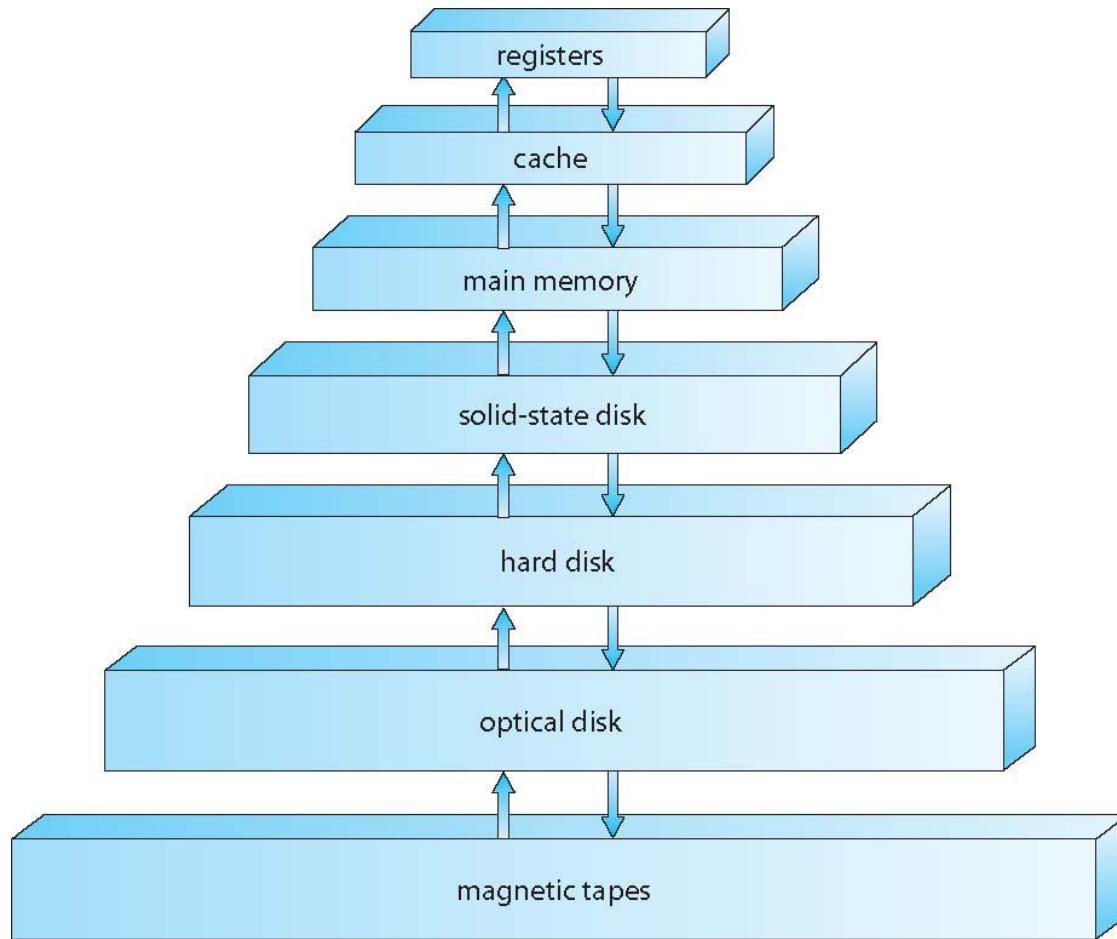
Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use is 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





Storage-Device Hierarchy





Performance of Various Levels of Storage

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

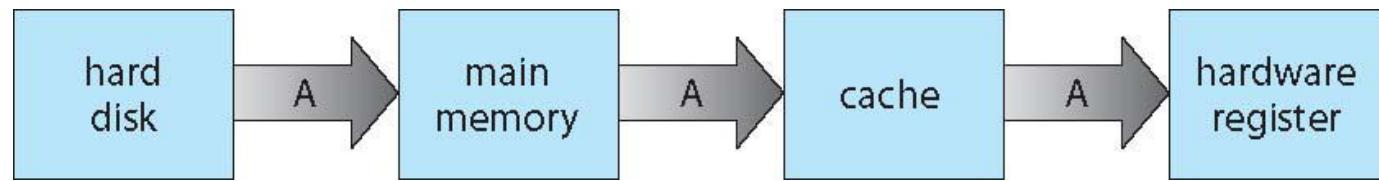
Movement between levels of storage hierarchy can be explicit or implicit





Migration of data “A” from Disk to Register

- Multitasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy
 - The same data “A” may appear in different levels of storage system
 - But... several processes may wish to access or update “A”



- Multiprocessor environment must provide **cache coherency** in hardware such that all CPUs have the most recent value in their cache
 - Each CPU has a local cache, which may contain “A” ... more complicated
- Distributed environment situation even more complex
 - The same “A” problems above are much more complex here
 - Several copies of a datum can exist on different computers in different levels
 - Various solutions covered in Chapter 17





I/O Subsystem

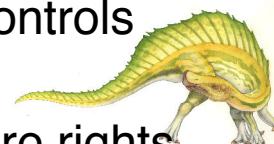
- One purpose of OS is to hide peculiarities of **specific** hardware devices from the user
- I/O subsystem responsible for
 - Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling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





Protection and Security

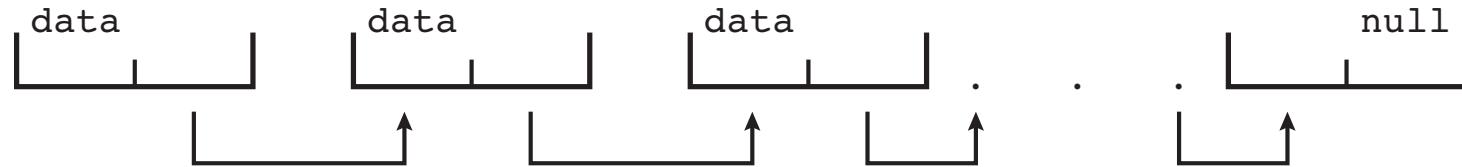
- **Regulation of data/rsrc access in a multiple-user multiple-process system**
 - Which process/user can access which data, ... how (long), where, and when?
- **Protection** – any mechanism for controlling access of processes or users to resources defined by the OS
 - Means for specifications of imposed controls and means for enforcement
- **Security** – defense of the system against internal and external attacks
 - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
 - User identities (**user IDs**, security IDs) include name and associated number, one per user
 - User ID then associated with all files, processes of that user to determine access control
 - Group identifier (**group ID**) allows set of users to be defined and controlled, managed, then also associated with each process, file
 - **Privilege escalation** allows user to change to effective ID with more rights



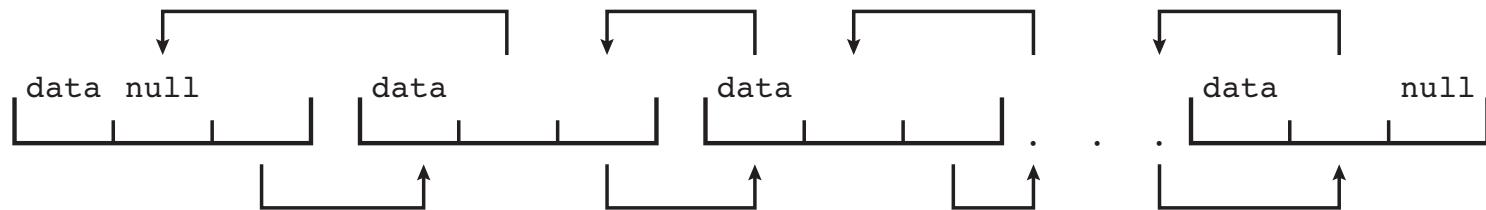


Kernel Data Structures

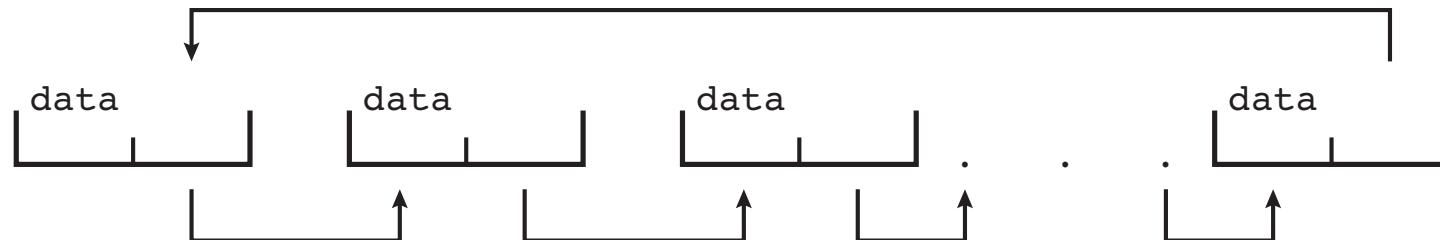
- Many similar to standard programming data structures
- ***Singly linked list***



- ***Doubly linked list***



- ***Circular linked list***



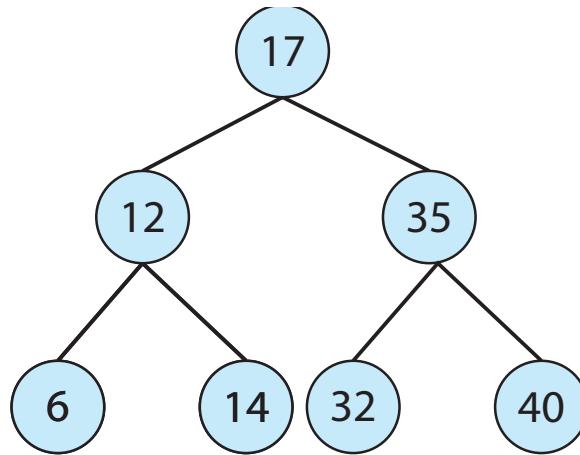


Kernel Data Structures

■ Binary search tree

left \leq right

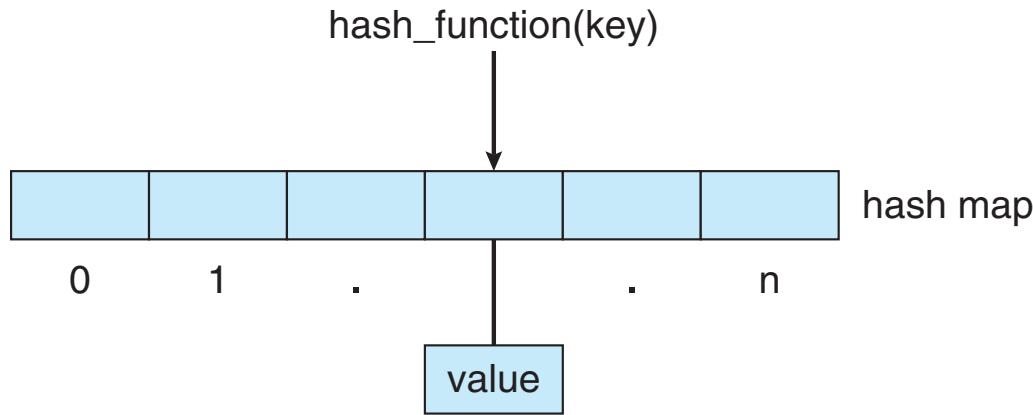
- Search performance is $O(n)$
- **Balanced binary search tree** is $O(\lg n)$





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>`





Computing Environments - Traditional

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





Computing Environments - Mobile

- Handheld smartphones, tablets, ... etc
 - Many more OS challenges to overcome due to their limited sizes
 - ▶ Small memory, small/slow processor, small screen, **small battery**, ... etc
 - Require efficient memory management and battery-life management
- What is the functional difference between them and a “traditional” laptop?
- Extra feature – more OS features (GPS, gyroscope)
- Allows new types of apps like ***augmented reality***
- Use IEEE 802.11 wireless, or cellular data networks for connectivity
- Leaders are **Apple iOS** and **Google Android**

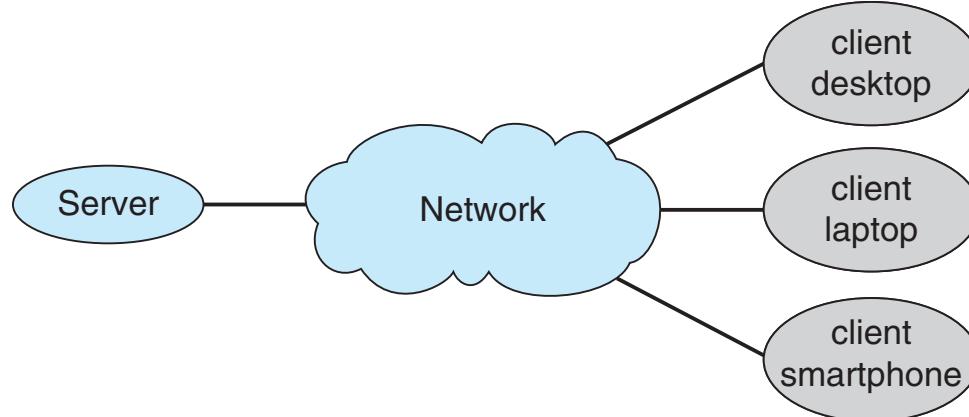




Computing Environments – Client-Server

Client-Server Computing

- Dumb terminals supplanted by smart PCs
 - ▶ No more centralized system architecture
- Many systems now act as **servers**, responding to requests generated by **clients**
 - ▶ **Compute-server system** provides an interface to client to request services (i.e., database)
 - ▶ **File-server system** provides interface for clients to store and retrieve files



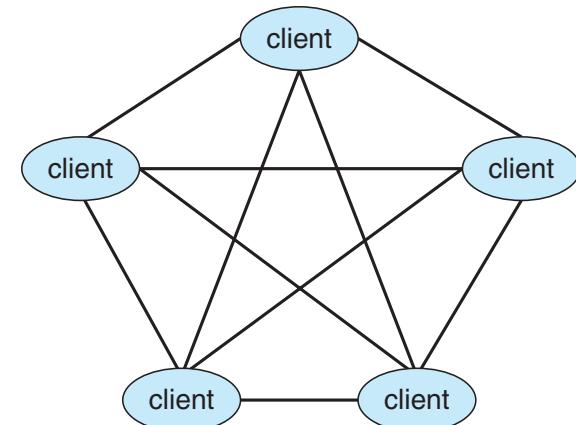


Computing Environments - Peer-to-Peer

- Another model of distributed system
 - Advantage: server is not a bottleneck

- P2P does not distinguish clients and servers

- Instead all nodes are considered peers
- May each act as client, server or both
- Node must join P2P network, and
 - ▶ Registers its service with central lookup service on network, or
 - ▶ Broadcast request for service and respond to requests for service via ***discovery protocol***
- Examples include Napster and Gnutella, **Voice over IP (VoIP)** such as Skype





Computing Environments - Virtualization

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





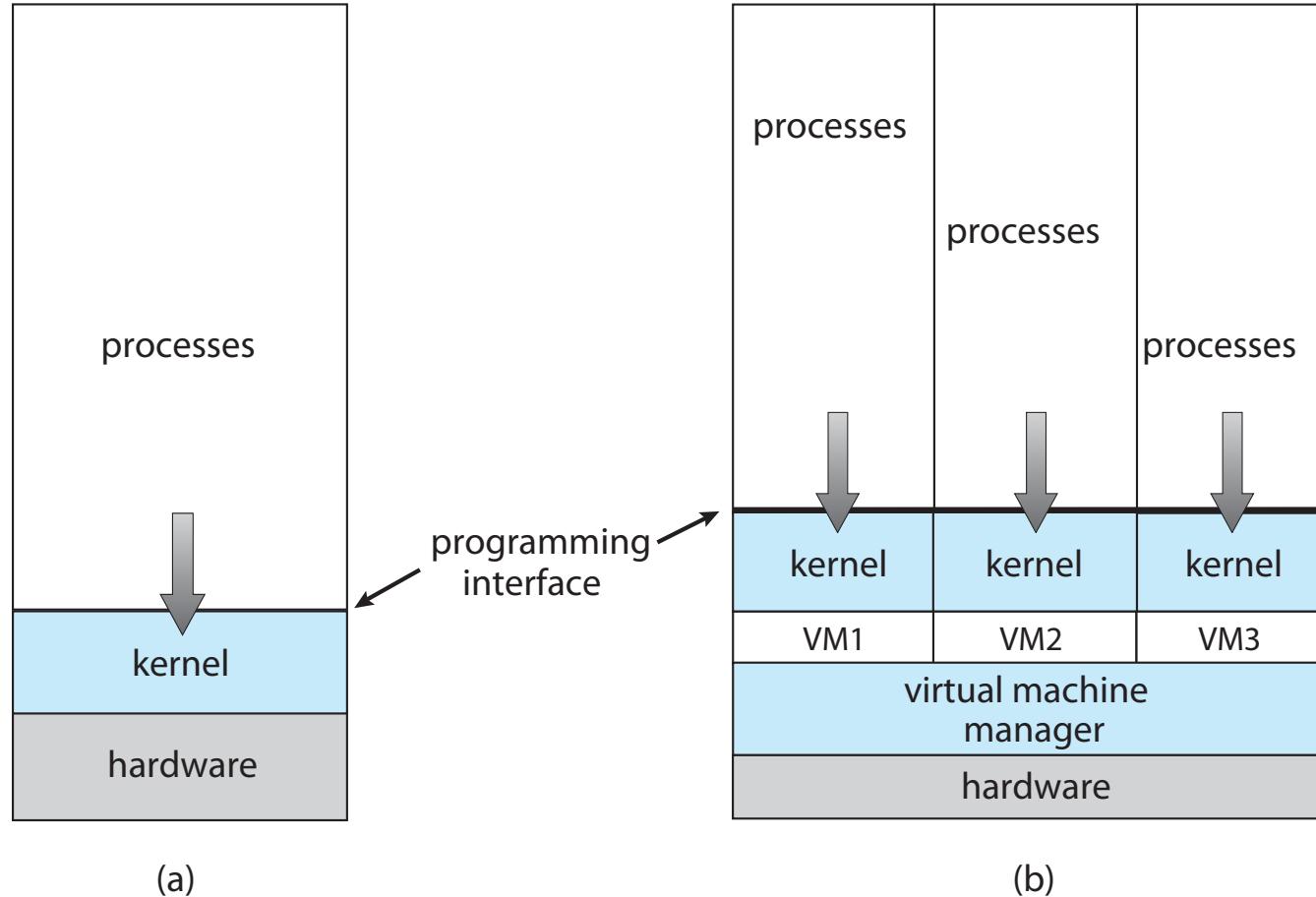
Computing Environments - Virtualization

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





Computing Environments - Virtualization





Computing Environments – Cloud Computing

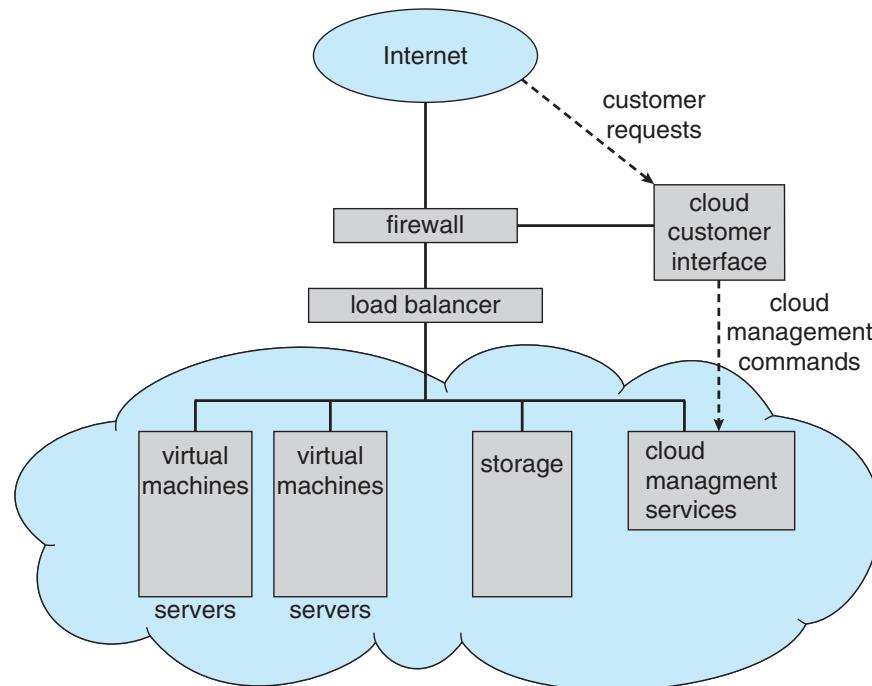
- Delivers computing, storage, even apps as a service across a network
- Logical extension of virtualization because it uses virtualization as the base for its functionality.
 - Amazon **EC2** has thousands of servers, millions of virtual machines, petabytes of storage available across the Internet,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





Computing Environments – Cloud Computing

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





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



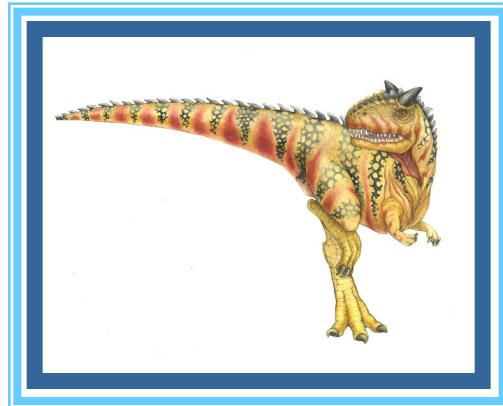


Open-Source Operating Systems

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



End of Chapter 1





Computer Startup

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

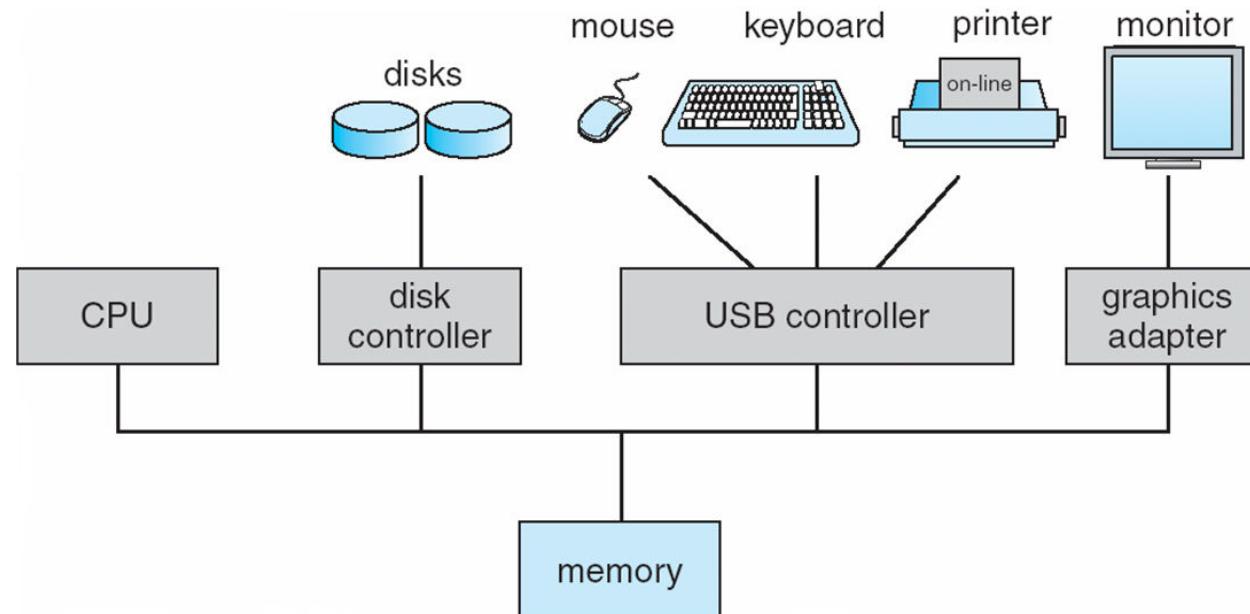




Computer System Organization

■ Computer-system operation

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





Computer-System Operation

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

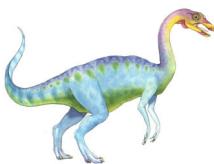




Common Functions of Interrupts

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





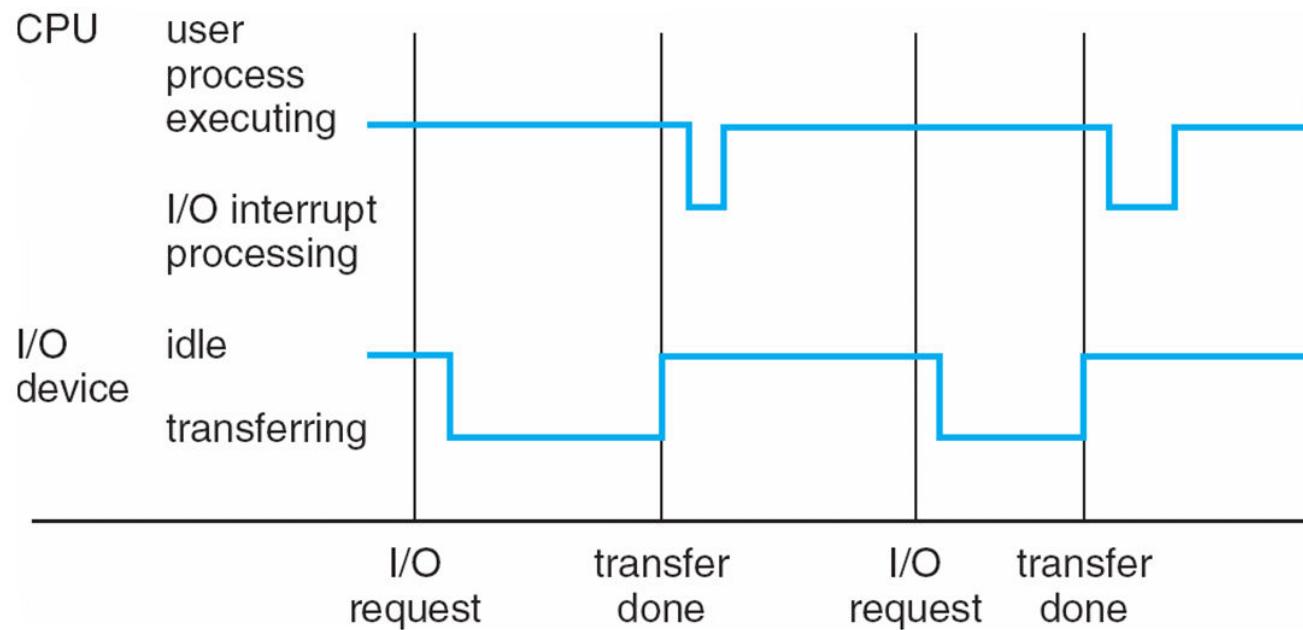
Interrupt Handling

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





Interrupt Timeline





I/O Structure

- After I/O starts, control returns to user program only upon I/O completion
 - Wait instruction idles the CPU until the next interrupt
 - Wait loop (contention for memory access)
 - At most one I/O request is outstanding at a time, no simultaneous I/O processing

- After I/O starts, control returns to user program without waiting for I/O completion
 - **System call** – request to the OS to allow user to wait for I/O completion
 - **Device-status table** contains entry for each I/O device indicating its type, address, and state
 - OS indexes into I/O device table to determine device status and to modify table entry to include interrupt





Storage 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

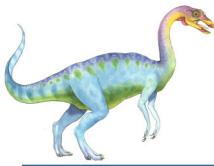
a **gigabyte**, or **GB**, is $1,024^3$ bytes

a **terabyte**, or **TB**, is $1,024^4$ bytes

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 Structure

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





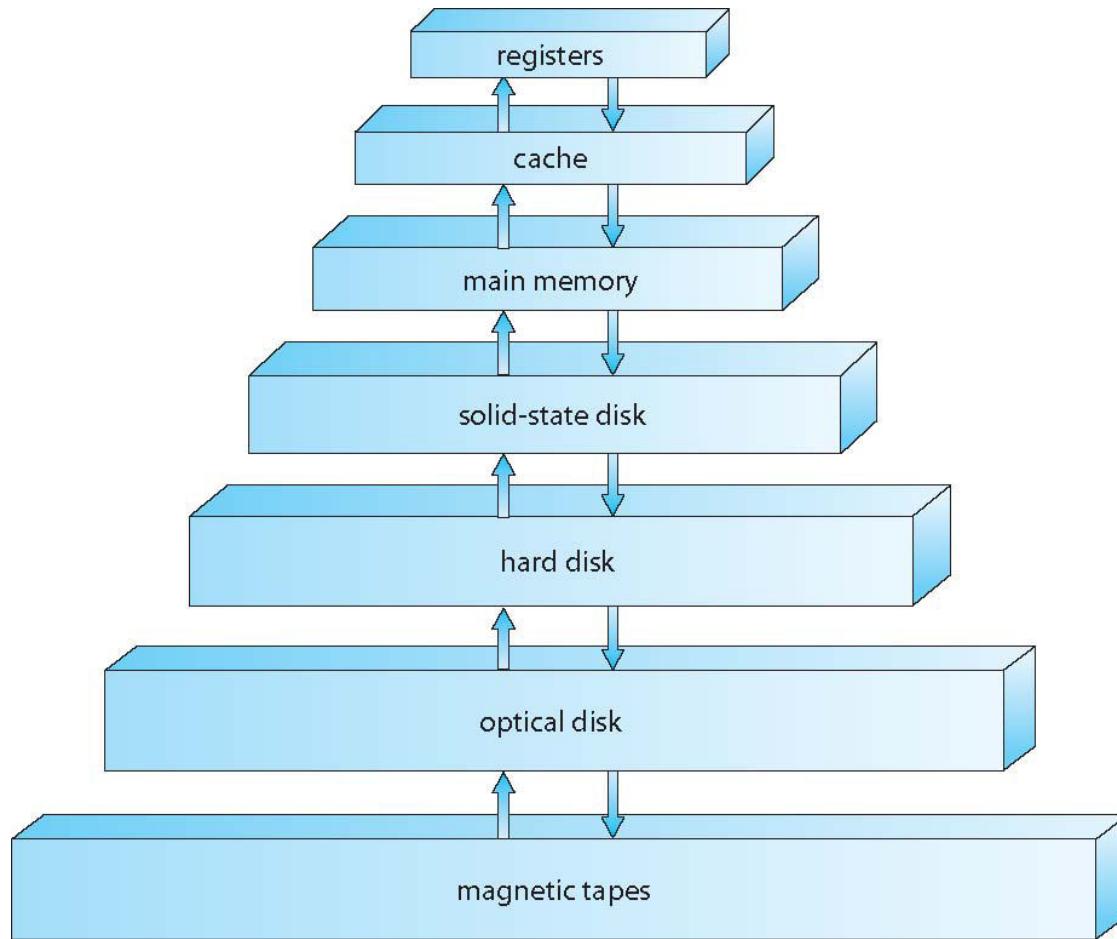
Storage Hierarchy

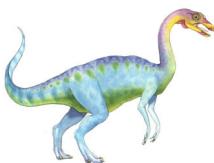
- Storage systems organized in hierarchy
 - Speed
 - Cost
 - Volatility
- **Caching** – copying information into faster storage system; main memory can be viewed as a cache for secondary storage
- **Device Driver** for each device controller to manage I/O
 - Provides uniform interface between controller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





Storage-Device Hierarchy





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





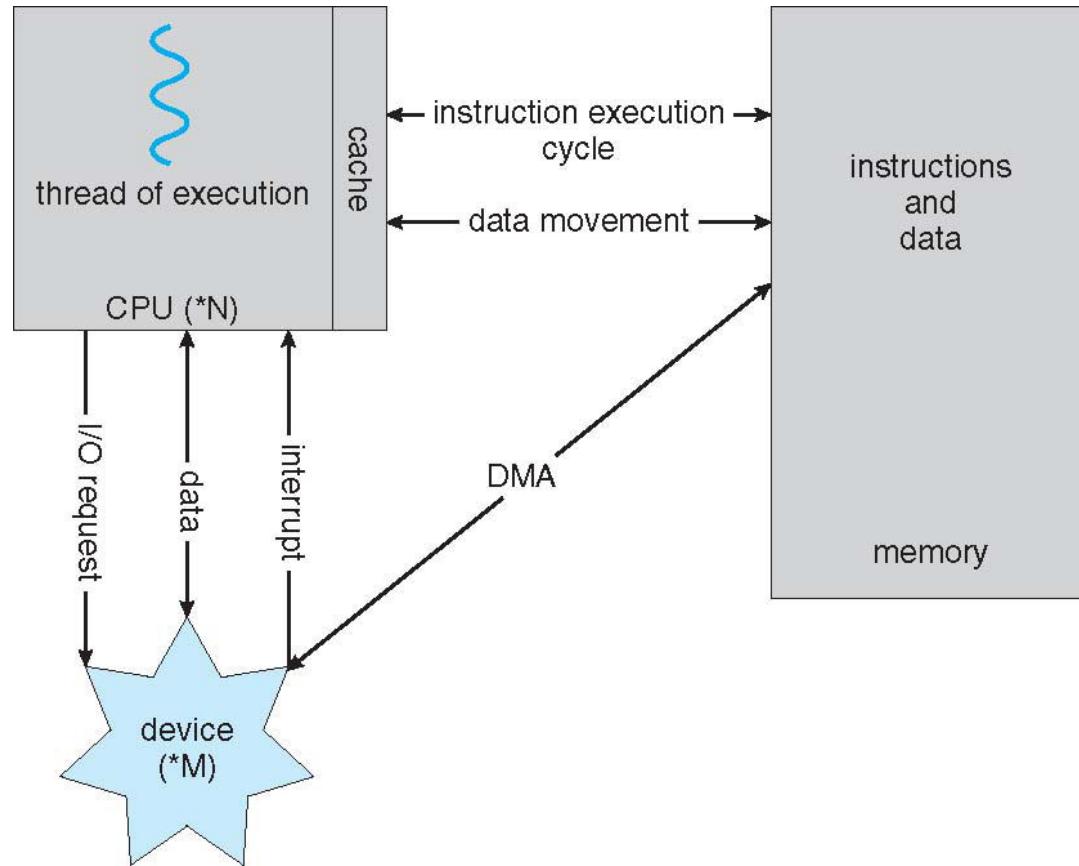
Direct Memory Access Structure

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





How a Modern Computer Works



A von Neumann architecture





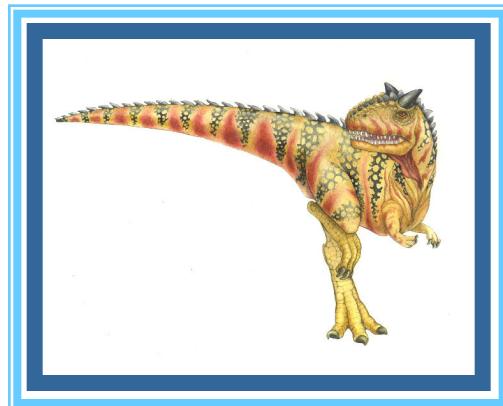
Computing Environments – Distributed

■ Distributed computing

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



Chapter 2: Operating-System Structures

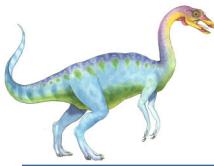




Chapter 2: Operating-System Structures

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Operating System Debugging
- Operating System Generation
- System Boot





Objectives

- Three views of an OS... each, respectively, focuses on
 - The services it provides [Ser]
 - ▶ User's view
 - The programming interface it makes available to users and programmers [Int]
 - ▶ Programmer's view
 - Its components and interconnections [Com]
 - ▶ OS designer's view
- To describe the services an operating system provides to users, processes, and other systems
- To discuss the various ways of structuring an operating system
- To explain how operating systems are installed and customized and how they boot





[Ser] Operating System Services (services helpful to the user - 6 categories)

- Operating systems provide an environment for execution of programs and services to programs and users
- One set of operating-system services provides functions that are **helpful to the user**:
 - **User interface** - Almost all operating systems have a user interface (**UI**).
 - ▶ Varies between
 - **Command-Line Interface (CLI)**: text commands and method for entering them
 - **Graphics User Interface (GUI)**: window system
 - **Batch Interface (BI)**: commands and directives are in makefiles
 - **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
 - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device





[Ser] Operating System Services (services helpful to the user - 6 categories)

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





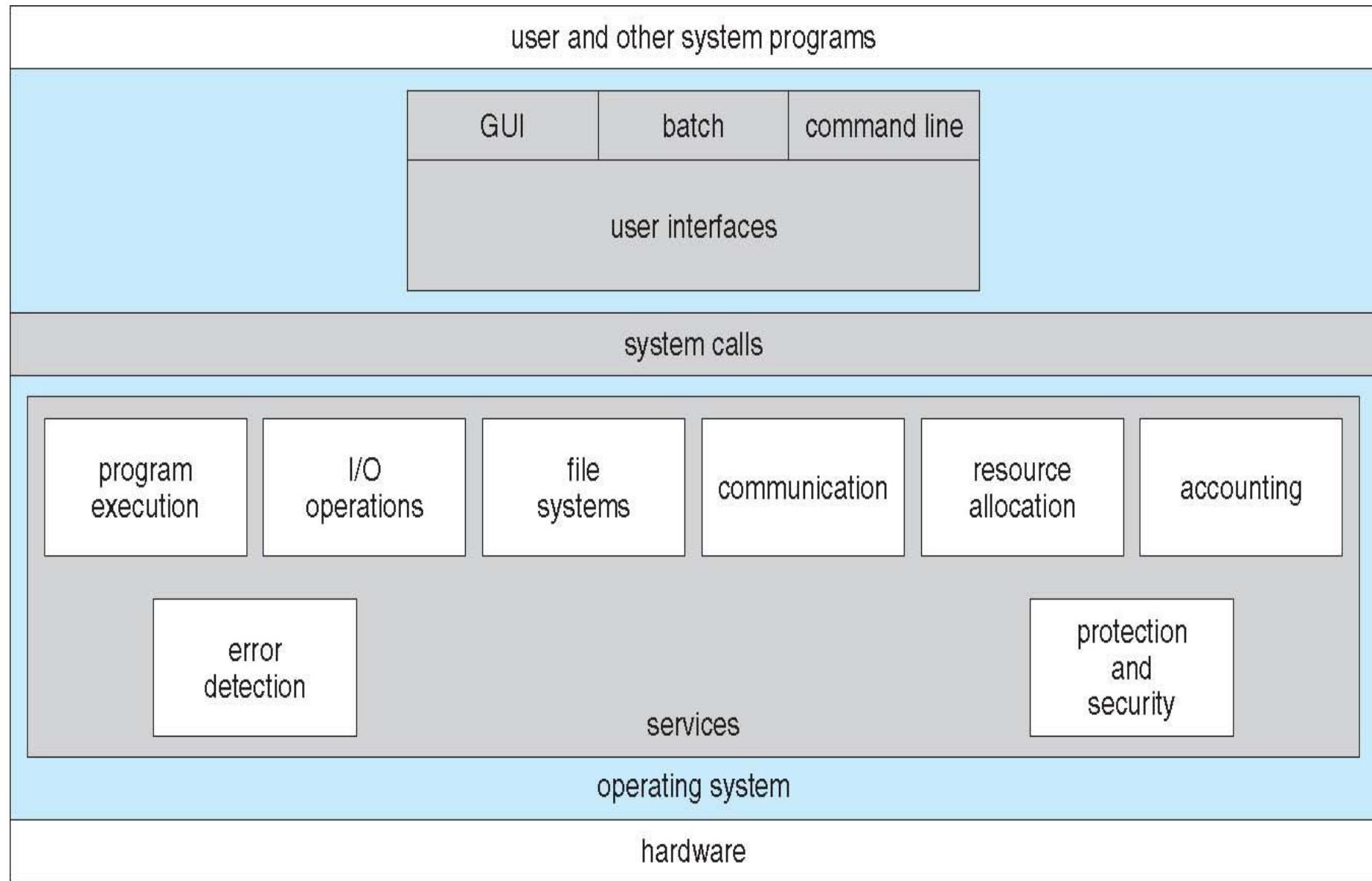
[Ser] Operating System Services (services for efficient operation of the OS - 3 categories)

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





[Ser] A View of Operating System Services





[Ser] Command-Line Interface - CLI

CLI or **command interpreter** allows direct command entry

- Sometimes implemented in kernel, sometimes by system programs (e.g., **Windows XP, UNIX**)
- Multiple CLIs in one system = **shells**
 - ▶ Bourne, C, Bash, or Korn shells,... etc in UNIX/Linux OS systems
- Function of CLI: fetches a command from user and executes it
 - ▶ Sometimes commands built into the CLI **software code**
 - Larger shells
 - ▶ Sometimes just names of system programs. Ex: “***rm file.txt***” in UNIX
 - **Smaller shells**. Adding new features doesn’t require shell modification





[Ser] Bourne Shell Command Interpreter

Default

New Info Close Execute Bookmarks

Default Default

```
PBG-Mac-Pro:~ pbg$ w
15:24  up 56 mins, 2 users, load averages: 1.51 1.53 1.65
USER      TTY      FROM          LOGIN@    IDLE WHAT
pbg      console   -
pbg      s000      -
PBG-Mac-Pro:~ pbg$ iostat 5
              disk0           disk1           disk10          cpu      load average
              KB/t tps  MB/s    KB/t tps  MB/s    KB/t tps  MB/s  us sy id  1m   5m   15m
 33.75 343 11.30   64.31 14  0.88   39.67  0  0.02  11 5 84  1.51 1.53 1.65
  5.27 320 1.65   0.00  0  0.00   0.00  0  0.00   4 2 94  1.39 1.51 1.65
  4.28 329 1.37   0.00  0  0.00   0.00  0  0.00   5 3 92  1.44 1.51 1.65
^C
PBG-Mac-Pro:~ pbg$ ls
Applications               Music
Applications (Parallels)   Pando Packages
Desktop                   Pictures
Documents                  Public
Downloads                 Sites
Dropbox                    Thumbs.db
Library                   Virtual Machines
Movies                     Volumes
WebEx
config.log
getsmartdata.txt
imp
log
panda-dist
prob.txt
scripts

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



[Ser] Graphical User Interface - GUI

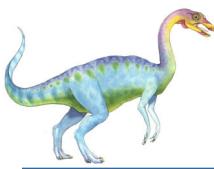
■ User-friendly **desktop** metaphor interface

- 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)





[Ser] Touchscreen Interfaces

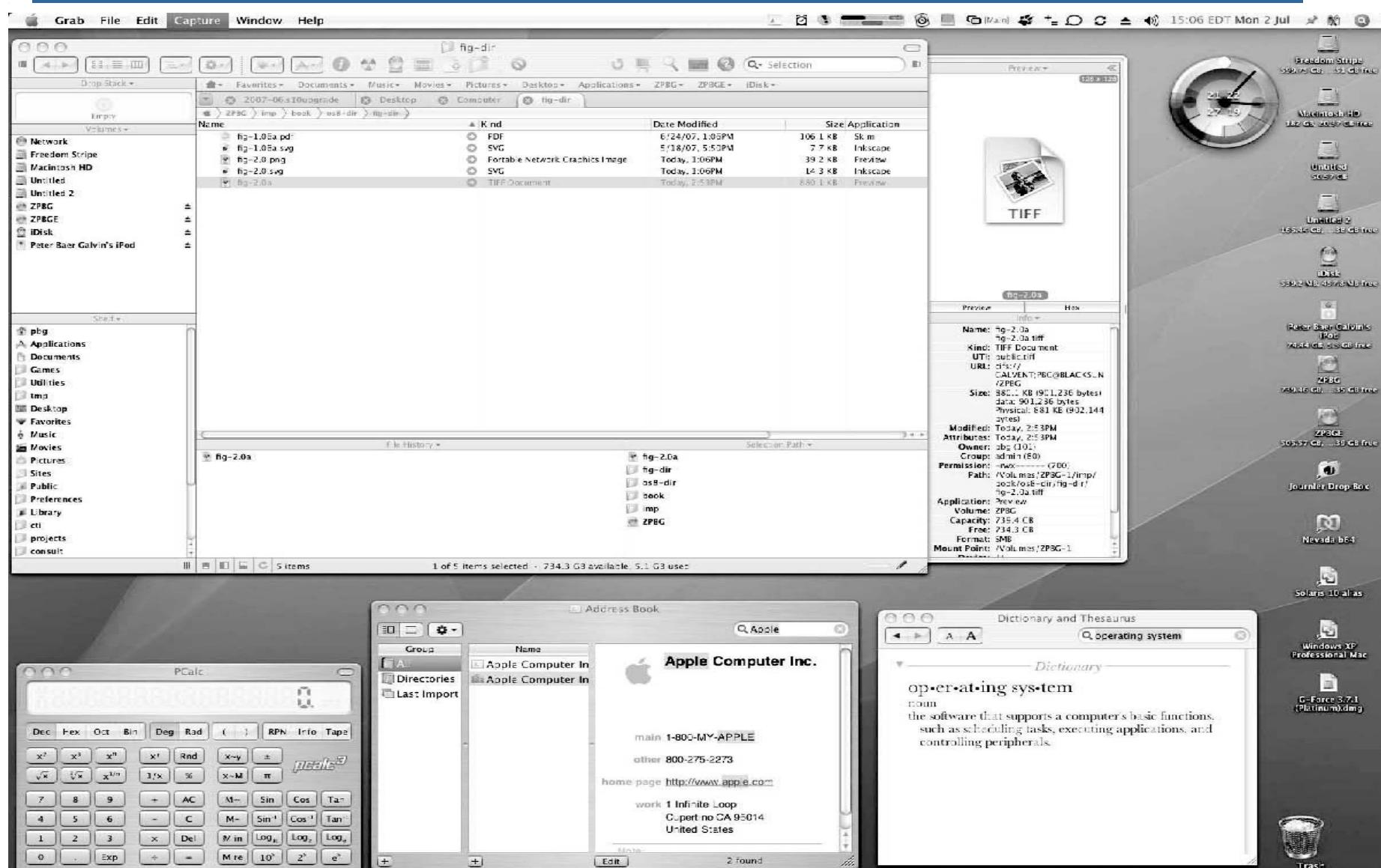
■ Touchscreen devices require new interfaces

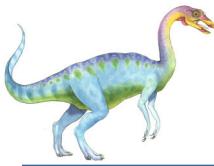
- Mouse not possible or not desired
- Actions and selection based on gestures
- Virtual keyboard for text entry
- Voice commands.





[Ser] The Mac OS X GUI

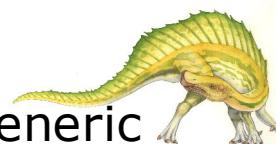




[Int] System-Calls

- Provide programming interface to the services made available by the OS
- Typically written in a high-level language (C or C++)
 - With hardware-level tasks written in *assembly language*
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system-call use
 - API specifies set of functions available to the programmer
 - ▶ Example: functions *ReadFile()* or *CreateProcess()* in WIN32 API
 - ▶ Functions invoke the actual system-calls on behalf of the programmer
 - Function *CreateProcess()* invokes system-call *NTCreateProcess()*
 - Why not invoke actual system-call (instead of using API) ? Because:
 - » Program portability: compile/run on systems supporting same API
 - Three most common APIs are
 - Win32 API for Windows
 - POSIX API for UNIX, Linux, and Mac OS X
 - Java API for the Java virtual machine (JVM)

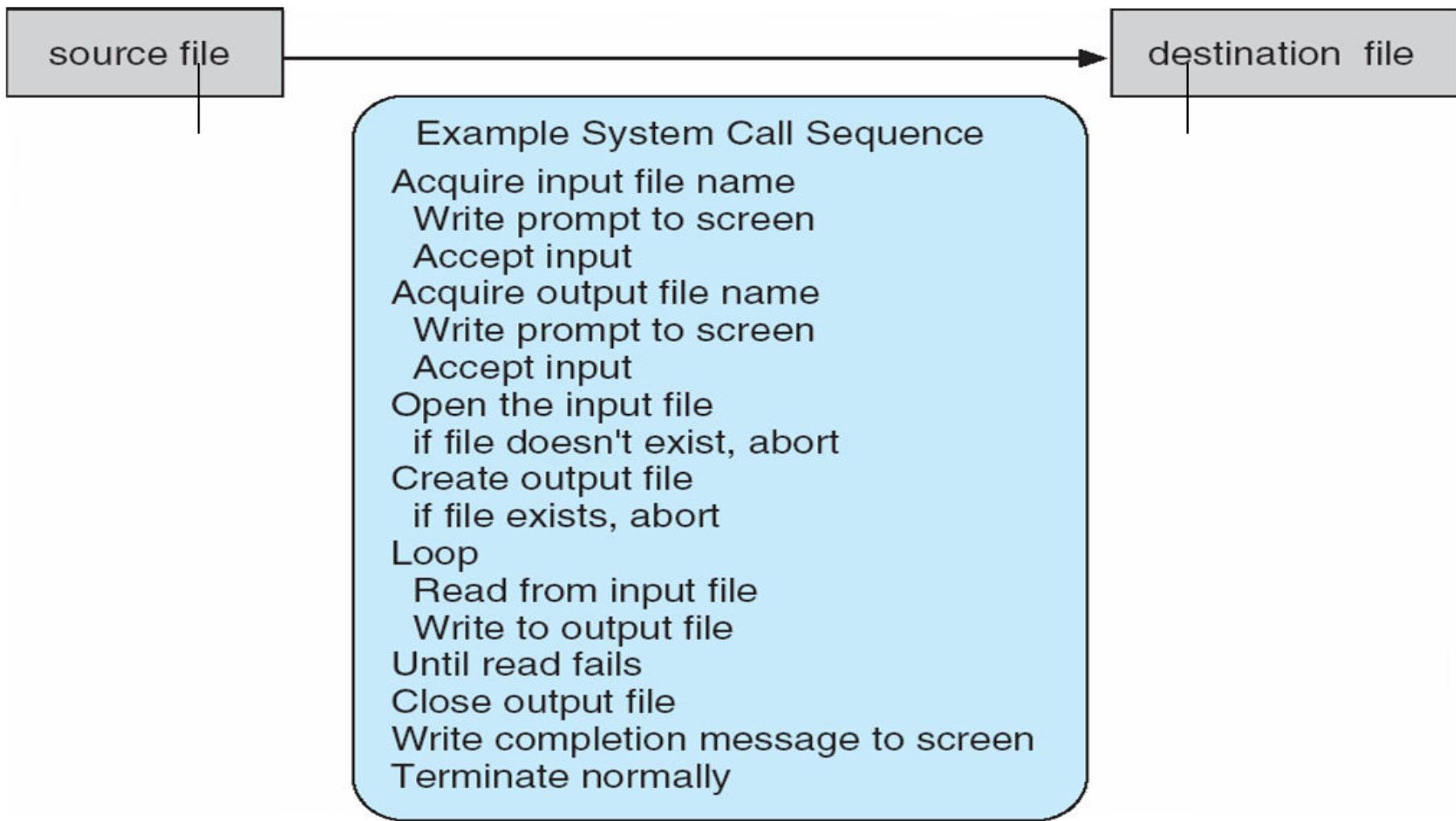
Note that the system-call names used throughout this text are generic





[Int] Example of System-Calls

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





[Int] Example of Standard API

EXAMPLE OF STANDARD API

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

```
man read
```

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

```
#include <unistd.h>

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

return function parameters
value name

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

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

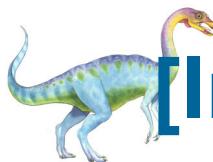
On a successful `read`, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns -1.



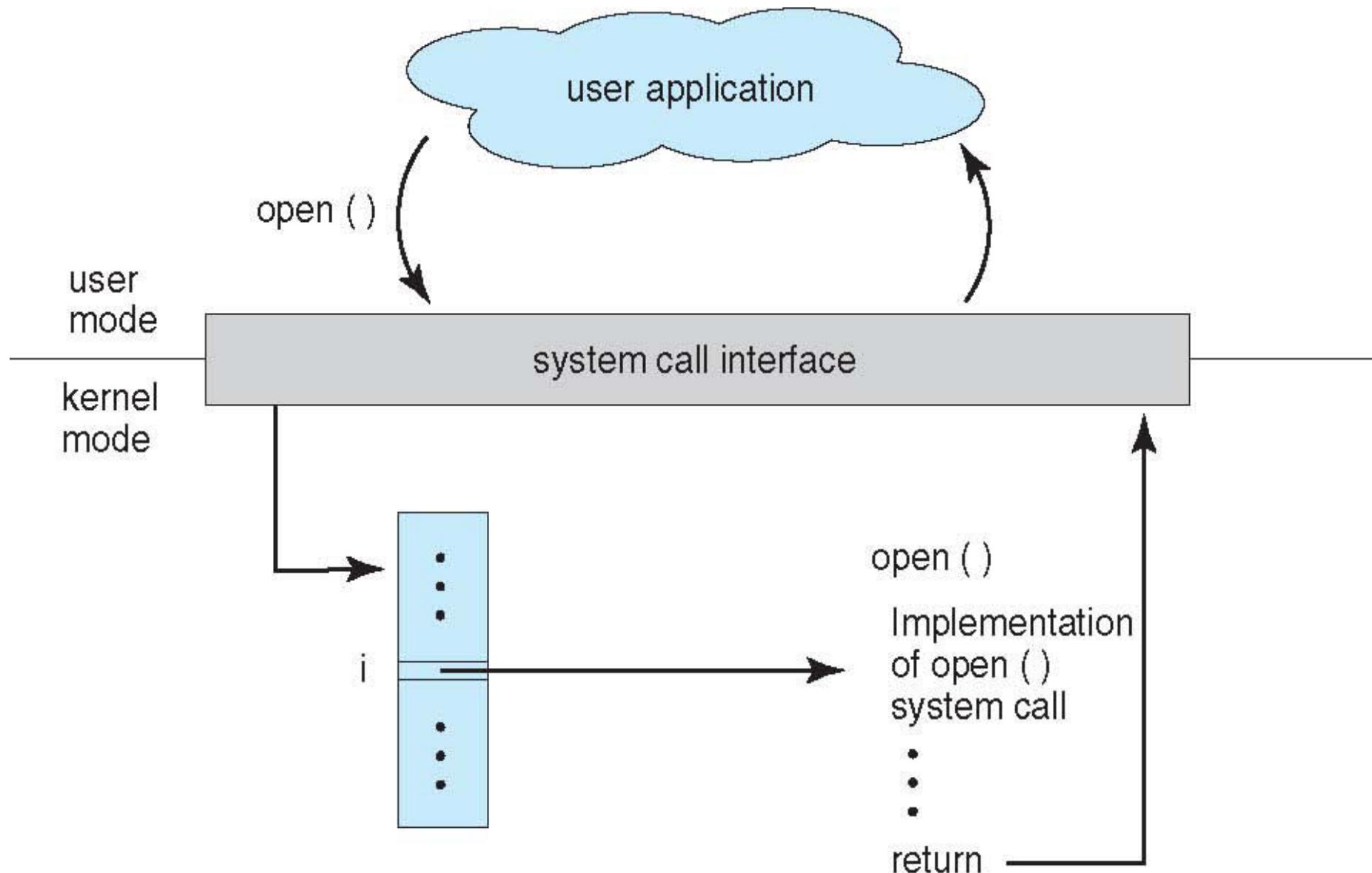
[Int] System-Call Implementation

- 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 the API and understand what OS will do as a result call
 - Most details of OS interface hidden from programmer by API
 - ▶ **System-call interface** is managed by run-time support library (set of functions built into libraries included with compiler)





[Int] API – System-Call – OS Relationship





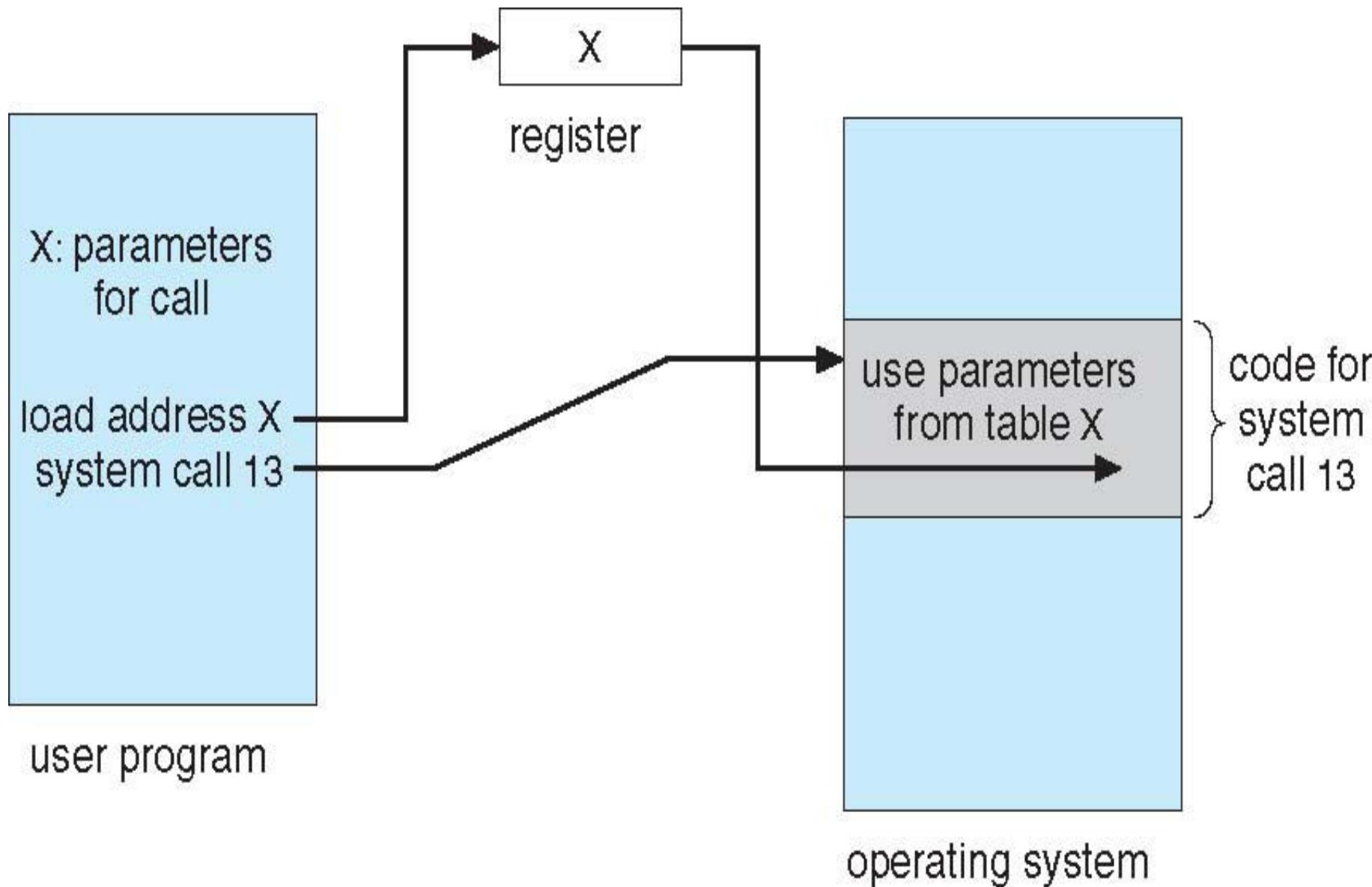
[Int] System-Call Parameter Passing

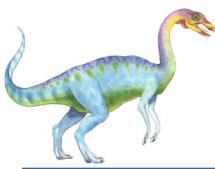
- Often, more information is required than simply the identity of desired system-call
 - Exact type and amount of information vary according to OS and system-call
- Three general methods used to pass parameters to the OS
(discussed in course 03-60-266)
 - Simplest: pass the parameters in registers; ie, EAX, EDX, ..., etc
 - ▶ 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 **runtime stack** by the program, and **popped** off the stack by the operating system when returning from syst-call
 - Block and stack methods **are preferred**... do not limit the number or length of parameters being passed





[Int] Parameter Passing via Table





[Int] Types of System-Calls (Process Control)

■ Process control

▶ Program should be able to start, control, and end or abort its execution

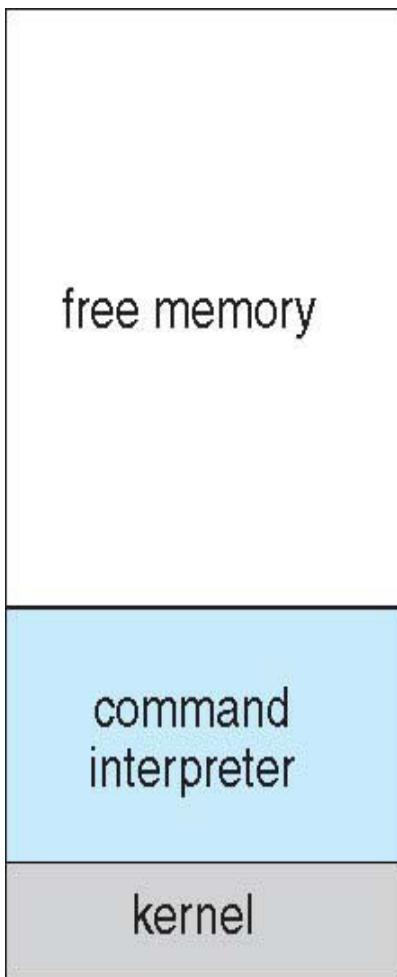
- create process, terminate process
- end, abort
- load, execute
- get process attributes, set process attributes
- wait for time
- wait event, signal event
- allocate and free memory
- Dump memory if error
- **Debugger** for determining **bugs, single step** execution
- **Locks** for managing access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





[Int] Process Control Example: MS-DOS

- Single-tasking
- Shell invoked when system booted
- Simple method to run program
 - No process created (?)
- Single memory space
- Loads program into memory, overwriting all but the kernel
- Program exit -> shell reloaded



(a)

At system startup



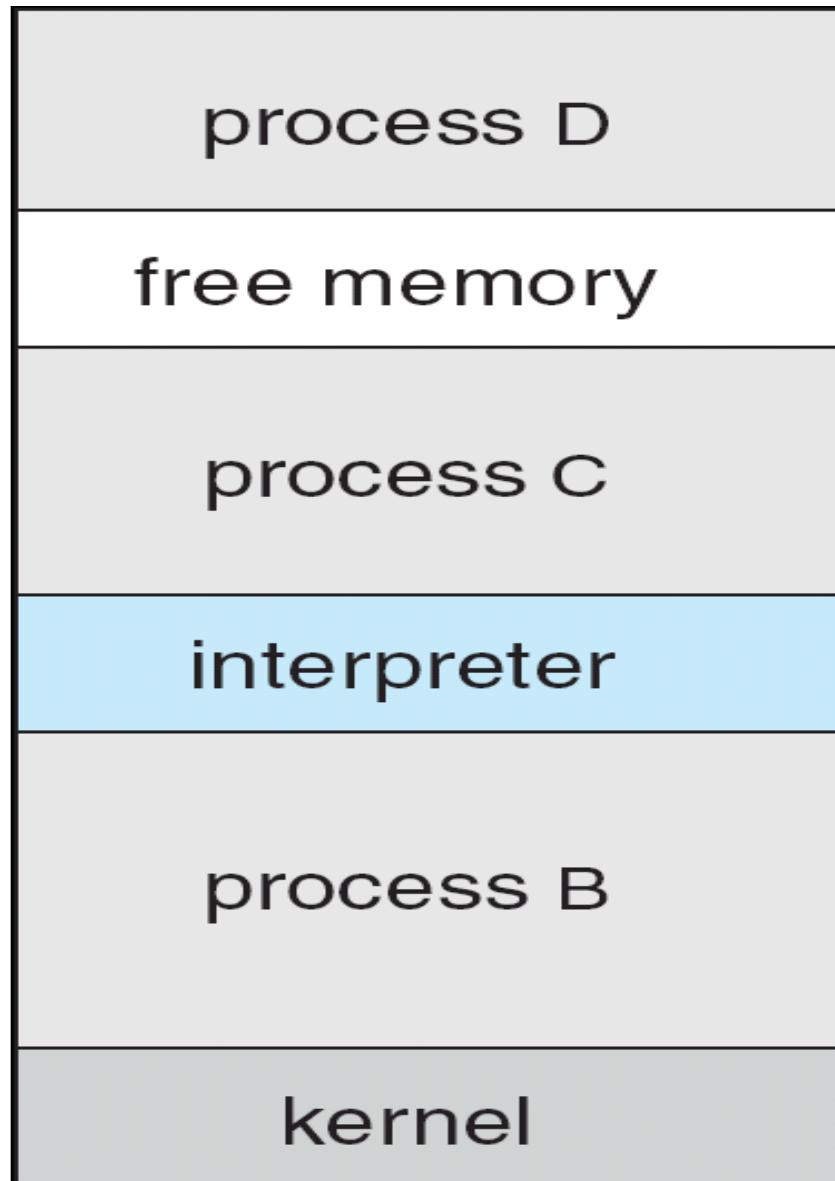
(b)

Running a program



[Int] Process Control 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





[Int] Types of System Calls

(File Management and Device Management)

■ File (and **directory**) management

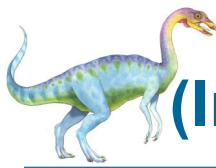
- create file, delete file
- open, close file
- read, write, reposition
- get and set file attributes

■ Device management

- ▶ Process needs several resources to execute
- ▶ Resources = Devices = memory, disk drives, files, ... etc, controlled by OS

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





[Int] Types of System Calls

(Information Maintenance and Communications)

■ Information maintenance

- ▶ All kinds of statistics and data that can be requested
 - Nb of users, size of free memory, OS version number, disk space, etc
- 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





[Int] Types of System Calls (Protection)

■ Protection

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





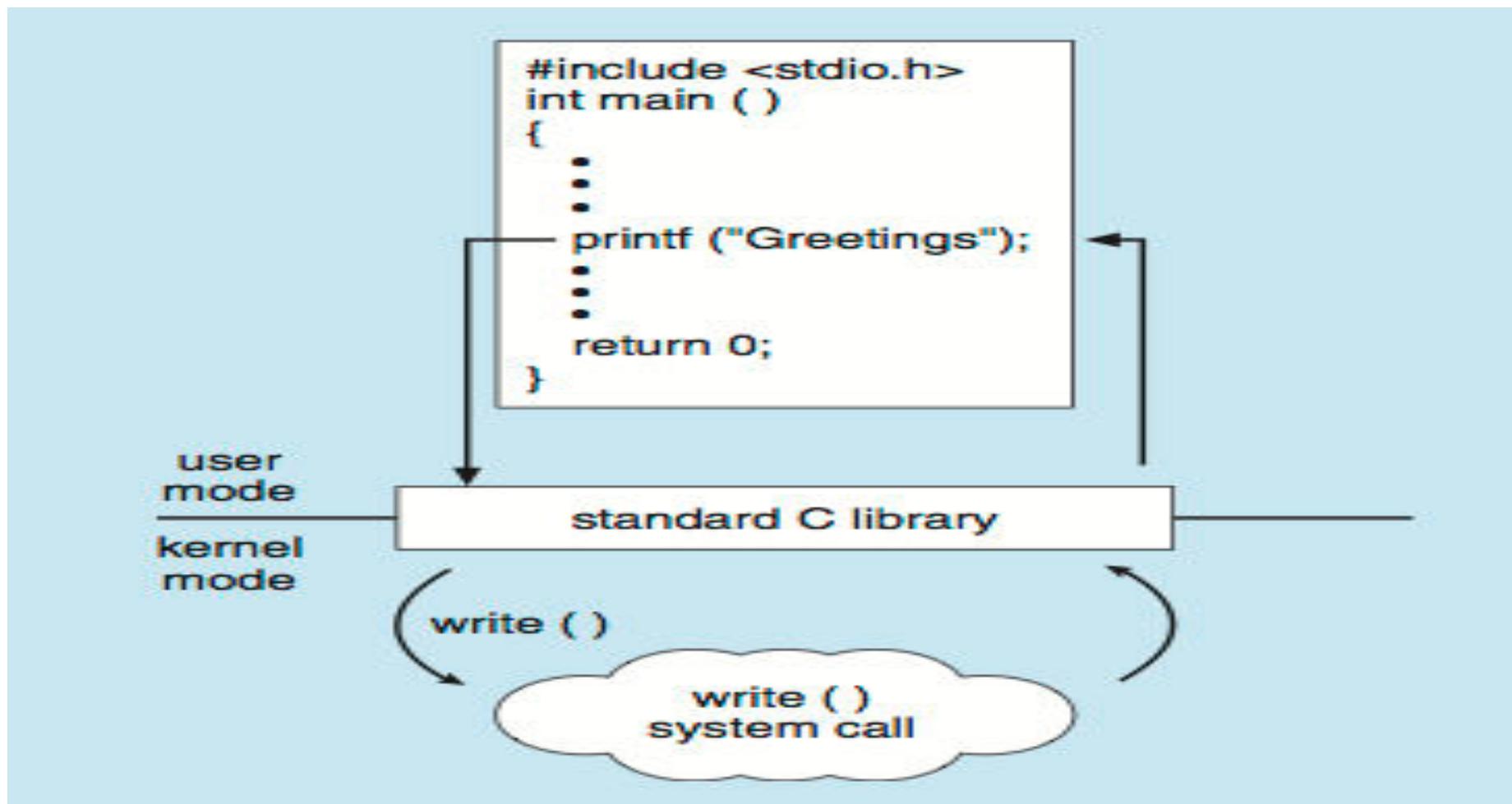
[Int] Examples of Windows and Unix System Calls

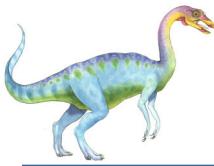
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information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



[Int] Standard C Library Example

- C program invoking printf() library call, which calls write() system call
 - The standard C library provides portion of system-call interface for many versions of UNIX and Linux





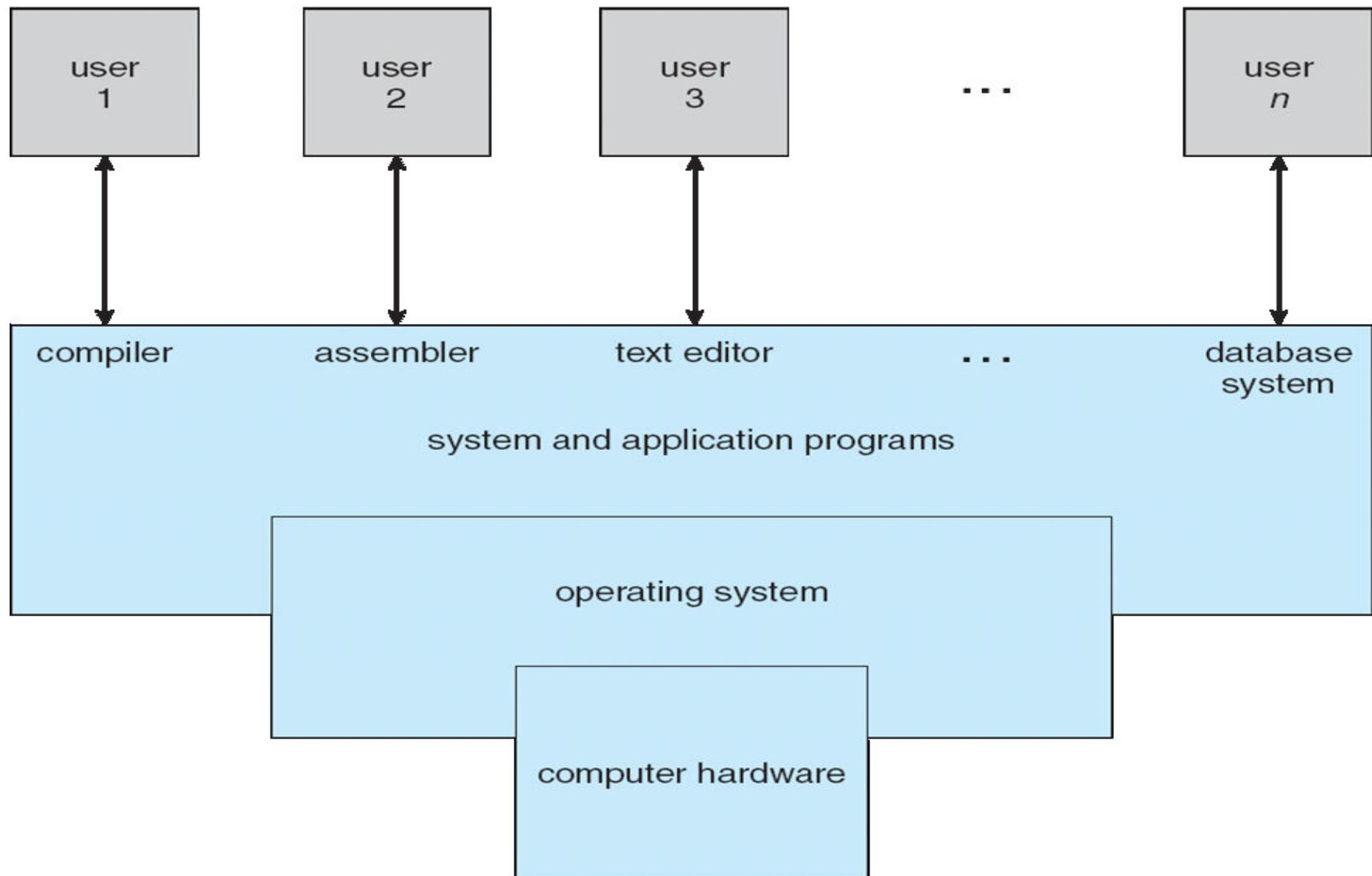
[Int] System Programs

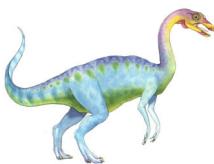
- System programs (or, **system utilities**) provide a convenient environment for program development and execution.
 - Some are user interfaces to system calls, others are very complex
 - ▶ E.g., browsers, formatters, assemblers, debuggers, defragmenters... etc
- They can be divided into:
 - File manipulation
 - Status information sometimes stored in a File modification
 - Programming language support
 - Program loading and execution
 - Communications
 - Background services
 - Application programs
- Most users' view of the operation system is defined by system programs, not the actual system calls





[Int] System Programs

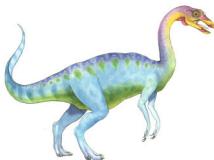




[Int] System Programs

- Provide a convenient environment for program development and execution
 - Some of them are simply user interfaces to system calls; others are considerably more complex
- **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





[Int] System Programs

■ 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





[Int] System Programs

■ 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; **but user think they are**
- Launched by command line, mouse click, finger poke

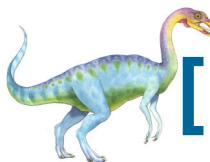




[Com] Operating System Structure

- General-purpose OS is very large program
 - ▶ Hence, OS must be engineered intelligently for easy use and modification
 - OS design: partition into modules and define interconnections
- Various ways to structure ones
 - Simple structure – MS-DOS: Monolithic, small kernel, not well separated modules, no protection, limited by Intel 8088 hardware
 - More complex -- original UNIX: Monolithic, large kernel, two-layered UNIX (separates kernel and system programs), initially limited by hardware
 - Layered – an abstraction: Modular OS, freedom to change/add modules
 - Microkernel – Mach: Modularized the expanded but large UNIX, keeps only essential component as system-level or user-level programs, smaller kernel, and easy to extend

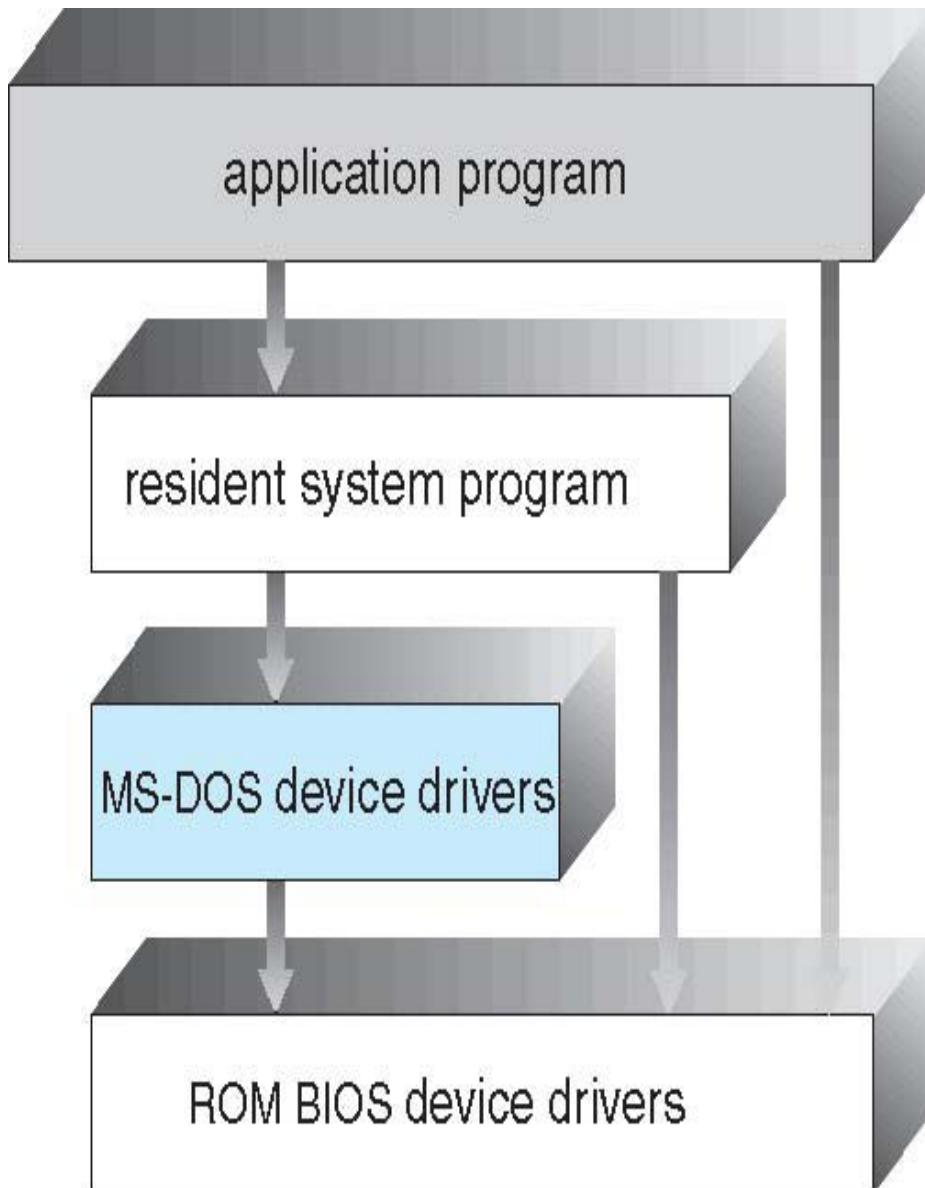


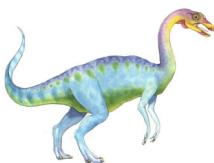


[Com] Simple Structure -- MS-DOS

- MS-DOS – written to provide the most functionality in the least space

- Not divided into modules
- Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated
 - ▶ E.g., app's can write directly to the display and disk drives
 - ▶ Intel 8088 processor had no **dual mode**
 - Vulnerable
 - No protection



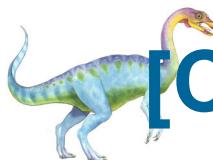


[Com] Non Simple Structure -- UNIX

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

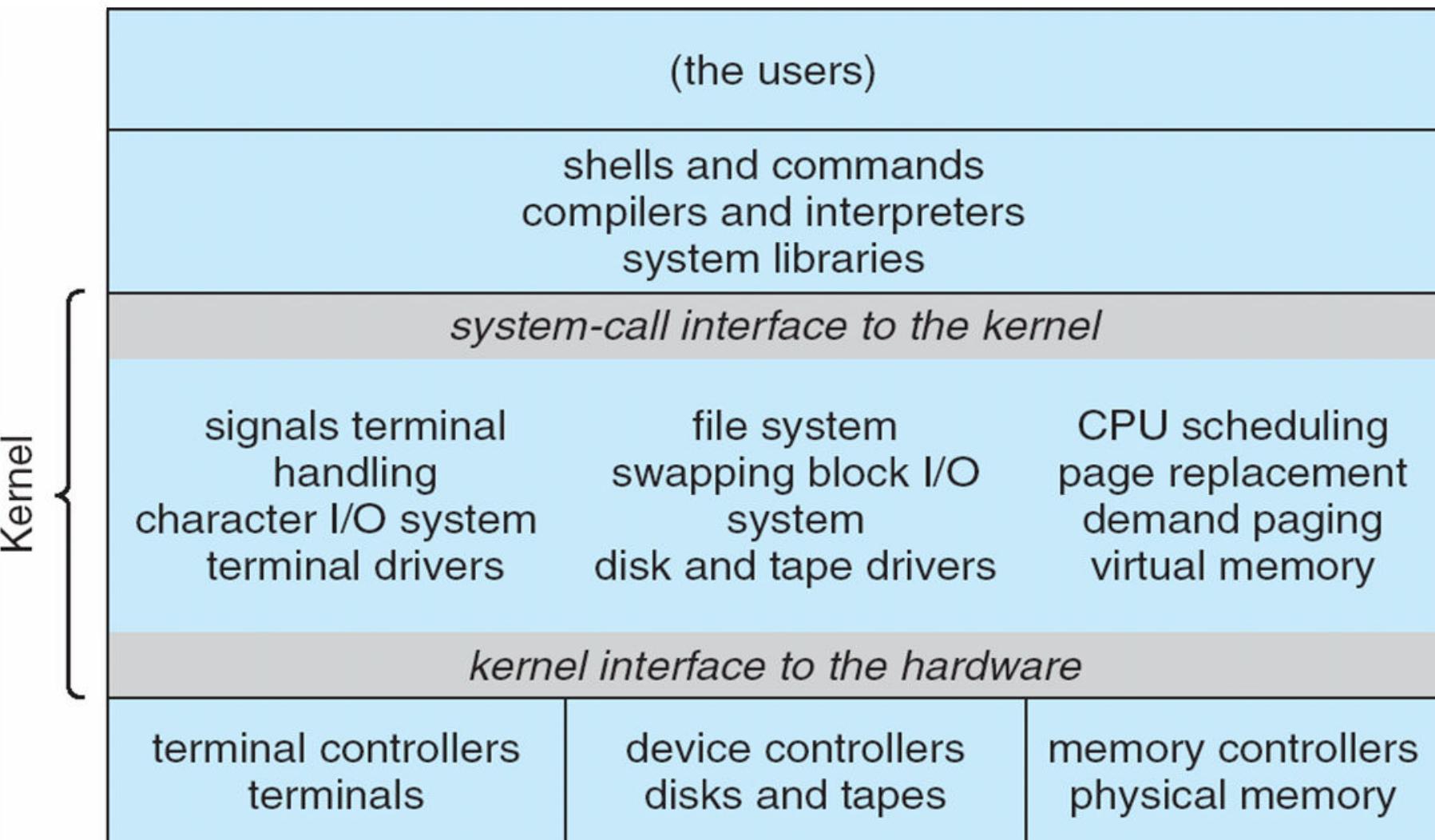
- Systems programs
- The kernel
 - ▶ Consists of everything below the system-call interface and above the physical hardware
 - ▶ Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level
 - **Very large kernel**

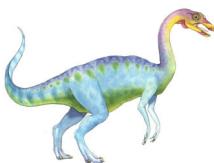




[Com] Traditional UNIX System Structure

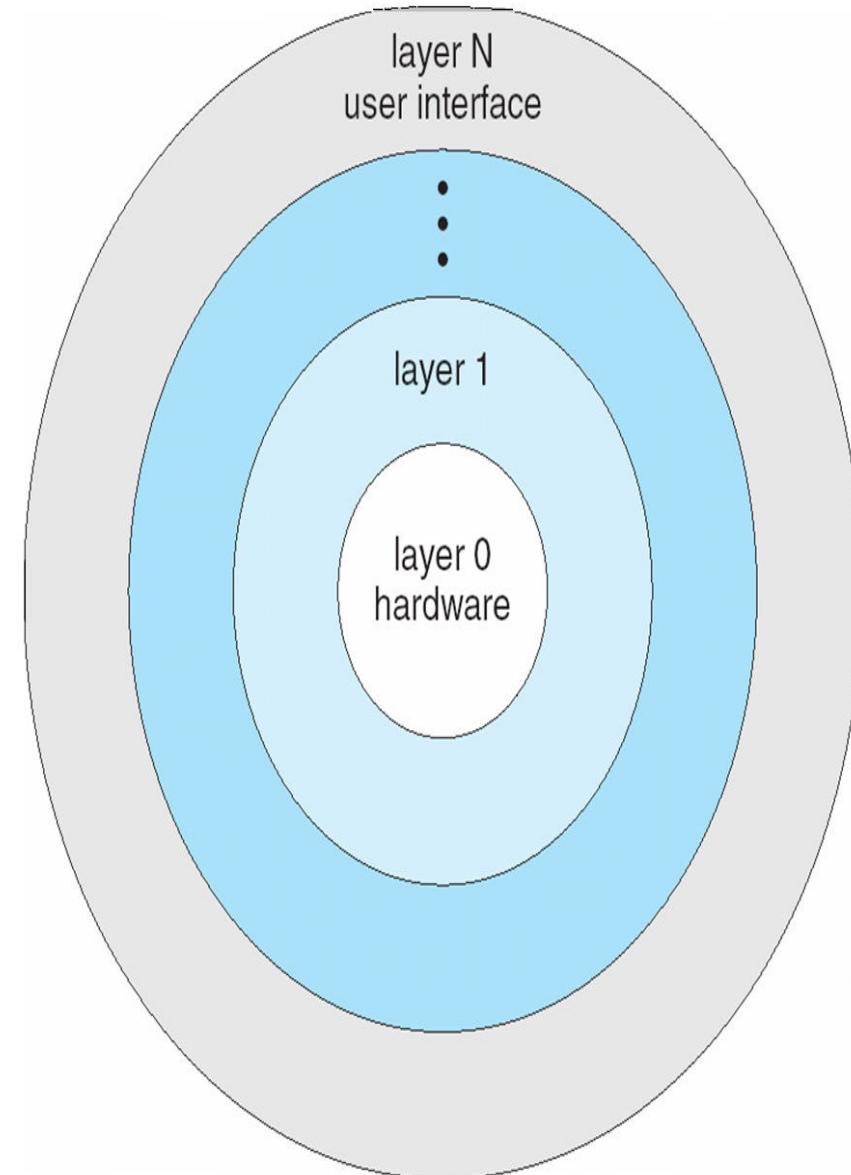
Beyond simple but not fully layered





[Com] Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers
- Advantage: each layer is
 - Abstraction: *data + operations on data*
 - Simple to construct
 - Easy to debug and verify
- Problems:
 - defining the various layers,
 - less efficient than non-layered OS





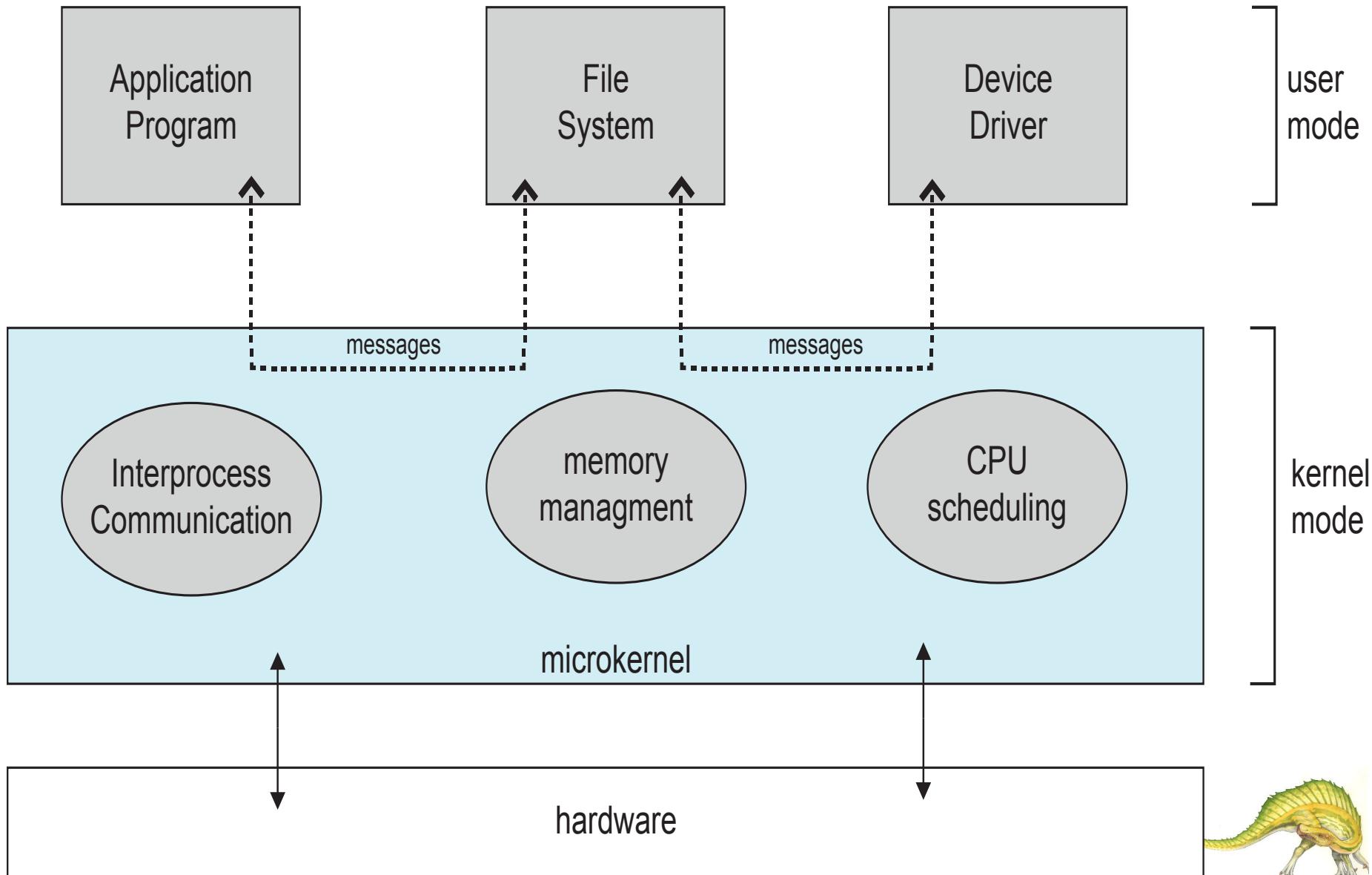
[Com] Microkernel System Structure

- Moves as much from the kernel into user space, hence, a small kernel
 - Kernel provides: process and memory management, and inter-process comm
 - All non-essential components are either user or system programs
- **Mach** example of **microkernel**
 - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between user modules using **message passing**
 - Function of microkernel: communication between client program and services
- Benefits:
 - Easier to extend a microkernel
 - Easier to port the operating system to new architectures
 - More reliable (less code is running in kernel mode)
 - More secure
- Detriments:
 - Performance overhead of user space to kernel space communication





[Com] Microkernel System Structure





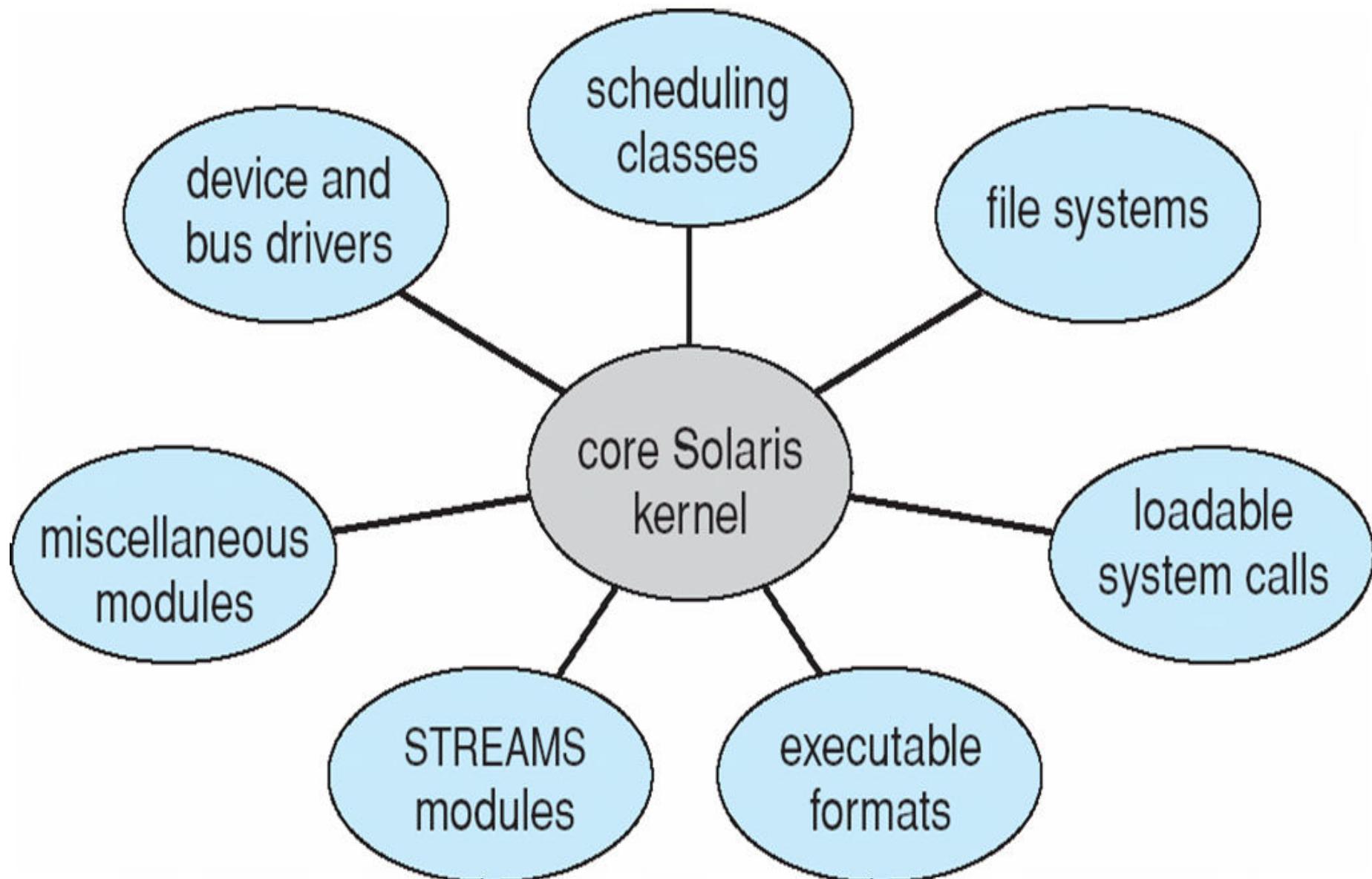
[Com] Modules

- Many modern operating systems implement **loadable kernel modules**
 - ▶ Kernel provides core services
 - ▶ Similar to microkernel
 - Uses object-oriented approach
 - Each core component is separate
 - Each talks to the others over known interfaces
 - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexibility
 - **Each kernel module is abstract but can call any other module**
 - Linux, Solaris, ... etc





[Com] Solaris Modular Approach





[Com] Hybrid Systems

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





[Com] Mac OS X Structure

graphical user interface

Aqua

application environments and services

Java

Cocoa

Quicktime

BSD

kernel environment

Mach

BSD

I/O kit

kernel extensions



- Apple mobile OS for ***iPhone, iPad***

- Structured on Mac OS X, added functionality
- Does not run OS X applications natively
 - ▶ Also runs on different CPU architecture (ARM vs. Intel)
- **Cocoa Touch** Objective-C API for developing apps
- **Media services** layer for graphics, audio, video
- **Core services** provides cloud computing, databases
- Core operating system, based on Mac OS X kernel

Cocoa Touch

Media Services

Core Services

Core OS



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

Application Framework

Libraries

SQLite

openGL

surface
manager

media
framework

webkit

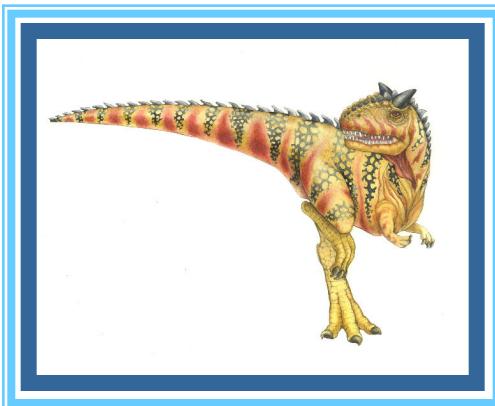
libc

Android runtime

Core Libraries

Dalvik
virtual machine

End of Chapter 2

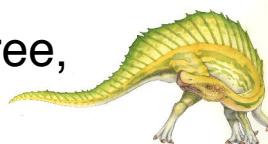




Operating System Design Goals

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system

- **User** goals and **System** goals
 - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
 - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient





Operating System Mechanisms and Policies

- Important principle to separate

Policy: *What* will be done?

Mechanism: *How* to do it?

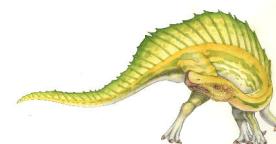
- Mechanisms determine how to do something, policies decide what will be done
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)
- Specifying and designing an OS is highly creative task of **software engineering**





Operating System Implementation

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





Operating-System Debugging

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

Kernighan's Law: "Debugging is twice as hard as writing the code in the first place.

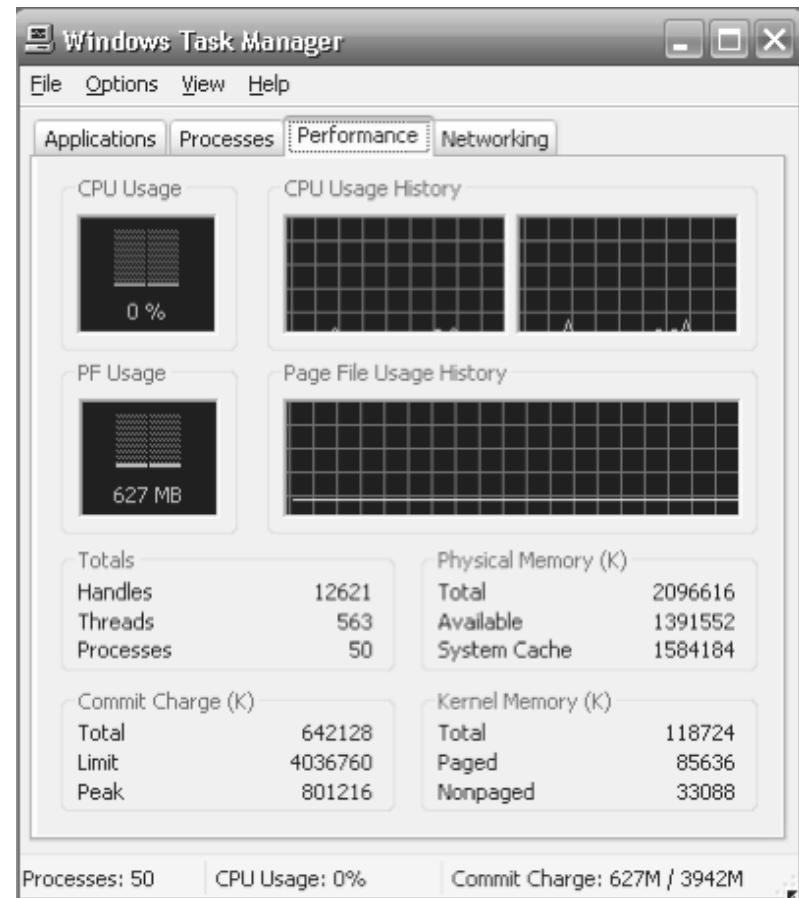
Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."





Performance Tuning

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





DTrace

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

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





Dtrace (Cont.)

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

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

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

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

Figure 2.21 Output of the D code.





Operating System Generation

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



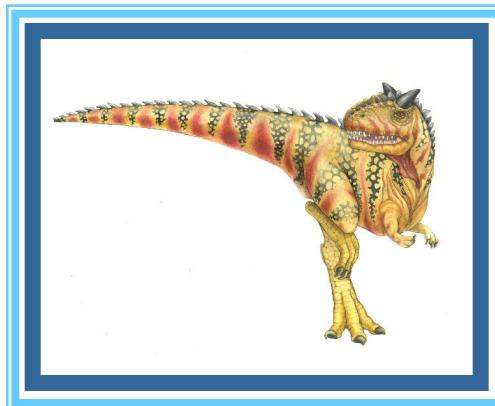


System Boot

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



Chapter 3: Processes





Chapter 3: Processes

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





Objectives

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





Process Concept

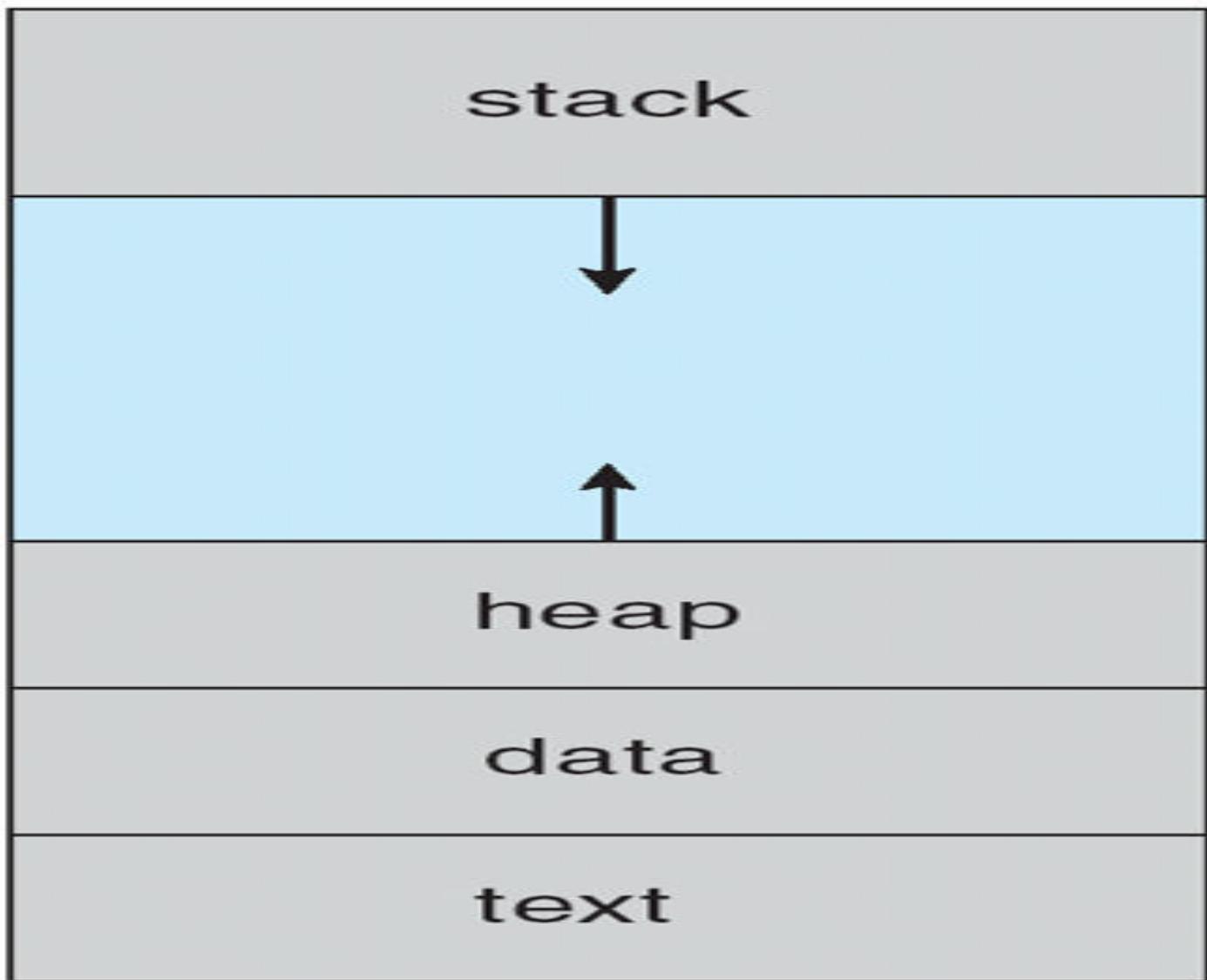
- An operating system executes a variety of programs:
 - Batch system – executes **jobs**
 - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms **job** and **process** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
 - ▶ System = collection of processes: OS processes and user processes
- Multiple parts (see 03-60-266)
 - The program code, also called **text section**; ie, **code segment**
 - Current activity including **program counter (EIP reg)**, processor registers
 - **Stack** containing temporary data; ie, **stack segment**
 - ▶ Function parameters, return addresses, local variables
 - **Data section** containing global variables; ie, **data segment**
 - **Heap** containing memory dynamically allocated during run time





Process in Memory

max



o



Process Concept

- Program is ***passive*** entity stored on disk (**executable file**), whereas a process is ***active***
 - Program becomes process when 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
 - They are separate processes with equivalent code segment (i.e. ***same text section***)





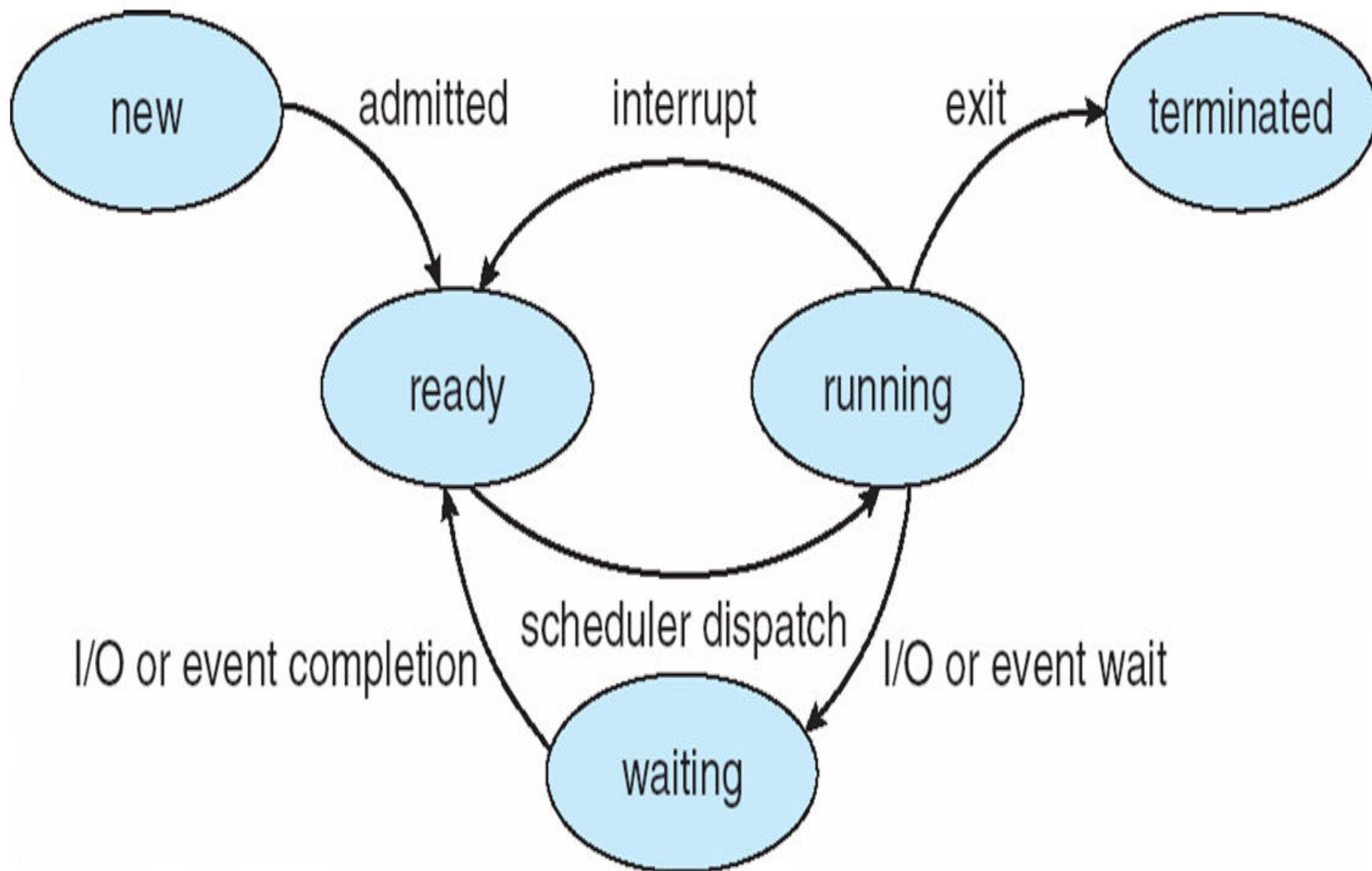
Process State

- As a process executes, it changes **state**
 - ▶ Arbitrary state names, and vary across OS's
 - ▶ Number of states varies across OS's
- **new**: The process is being created
- **ready**: The process is waiting to be assigned to a processor
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **terminated**: The process has finished execution





Diagram of Process State

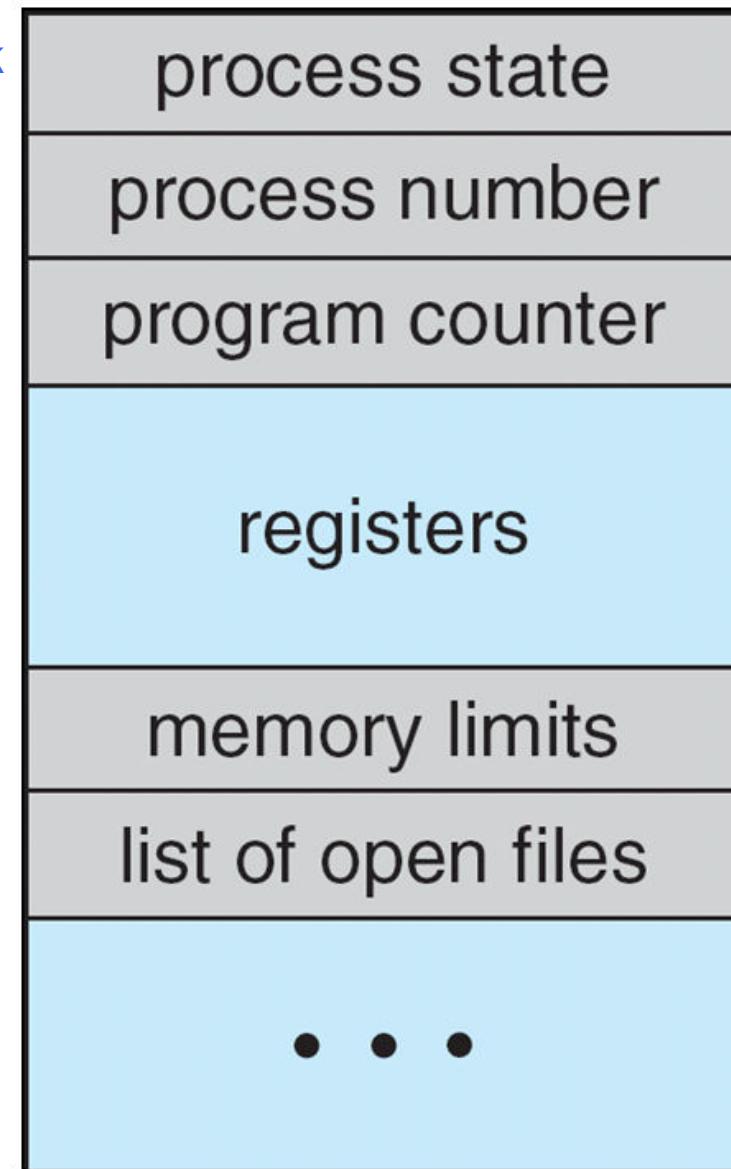




Process Control Block (PCB)

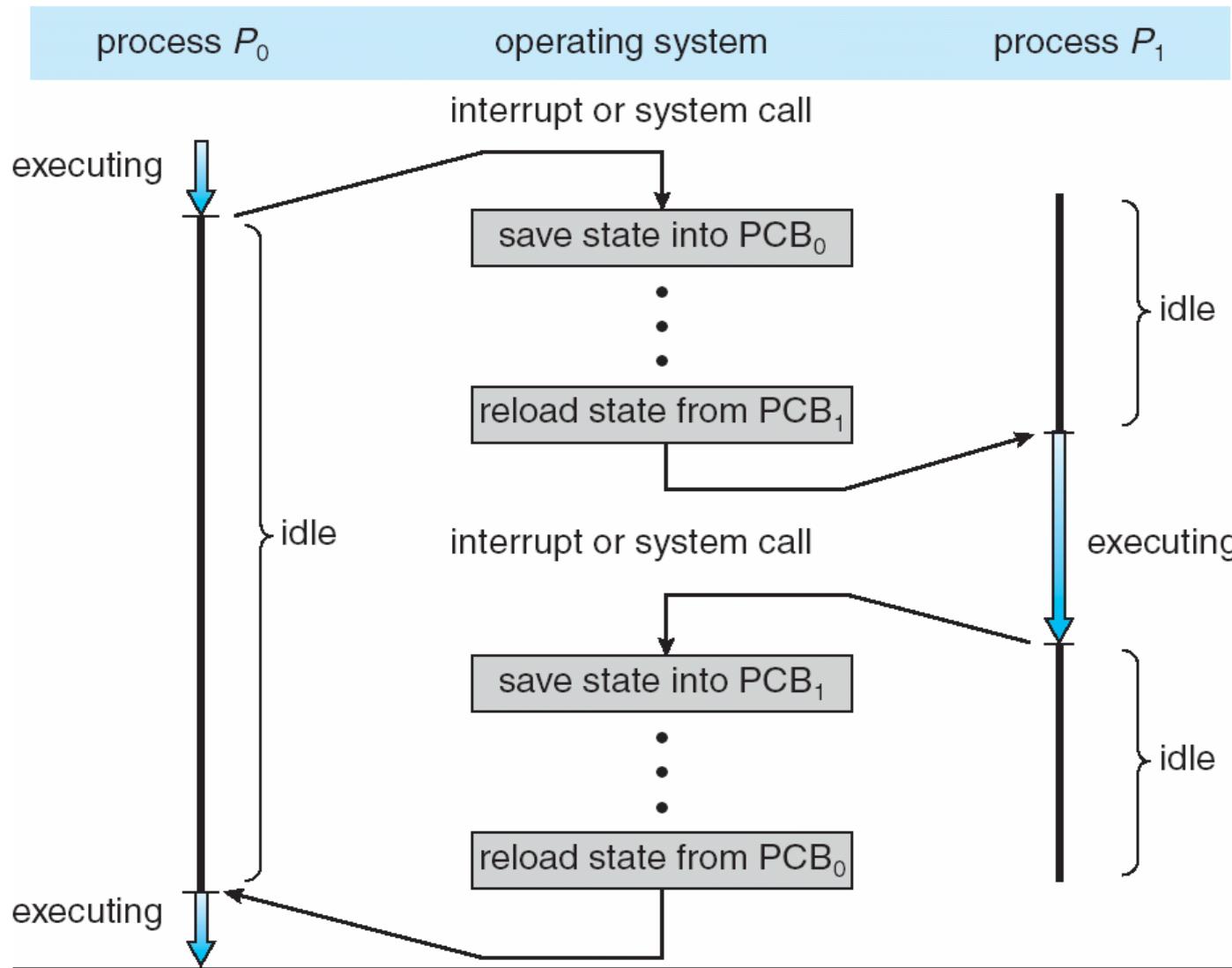
Process represented in OS by a **task control block**
(i.e., a PCB = information associated with task)

- Process state – running, waiting, etc
- Program counter – address of next instruction to be executed for this process
- CPU registers – contents of all process-centric registers: **EAX, ESI, ESP, EFLAGS, ... etc**
- CPU scheduling information – process priority, scheduling queue pointers, ... etc
- Memory-management information – memory allocated to the process, **EBP, segment registers, page and segment tables... etc**
- Accounting information – CPU used, clock time elapsed since start, time limits, ... etc
- I/O status information – I/O devices allocated to process, list of open files, ... etc





CPU Switch From Process to Process





Threads

- So far, we have implied that a process has a single thread of execution
 - Performs only 1 task at a time
- 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
- See next chapter



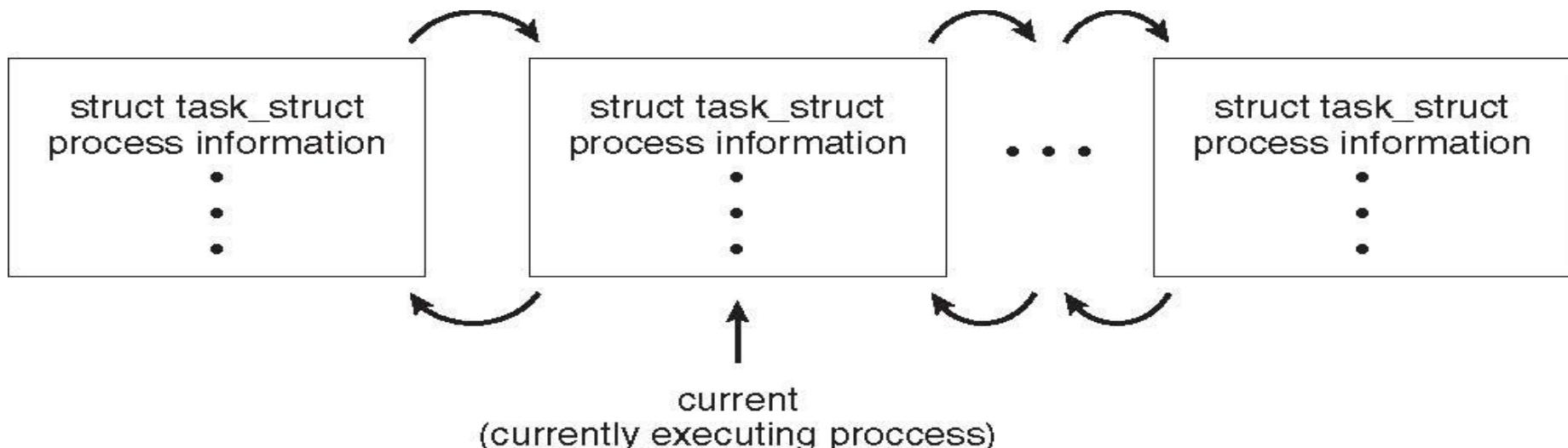


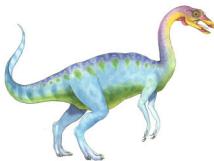
Process Representation in Linux

Represented by the C structure `task_struct`

This PCB contains all necessary info for a process

```
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 */
```





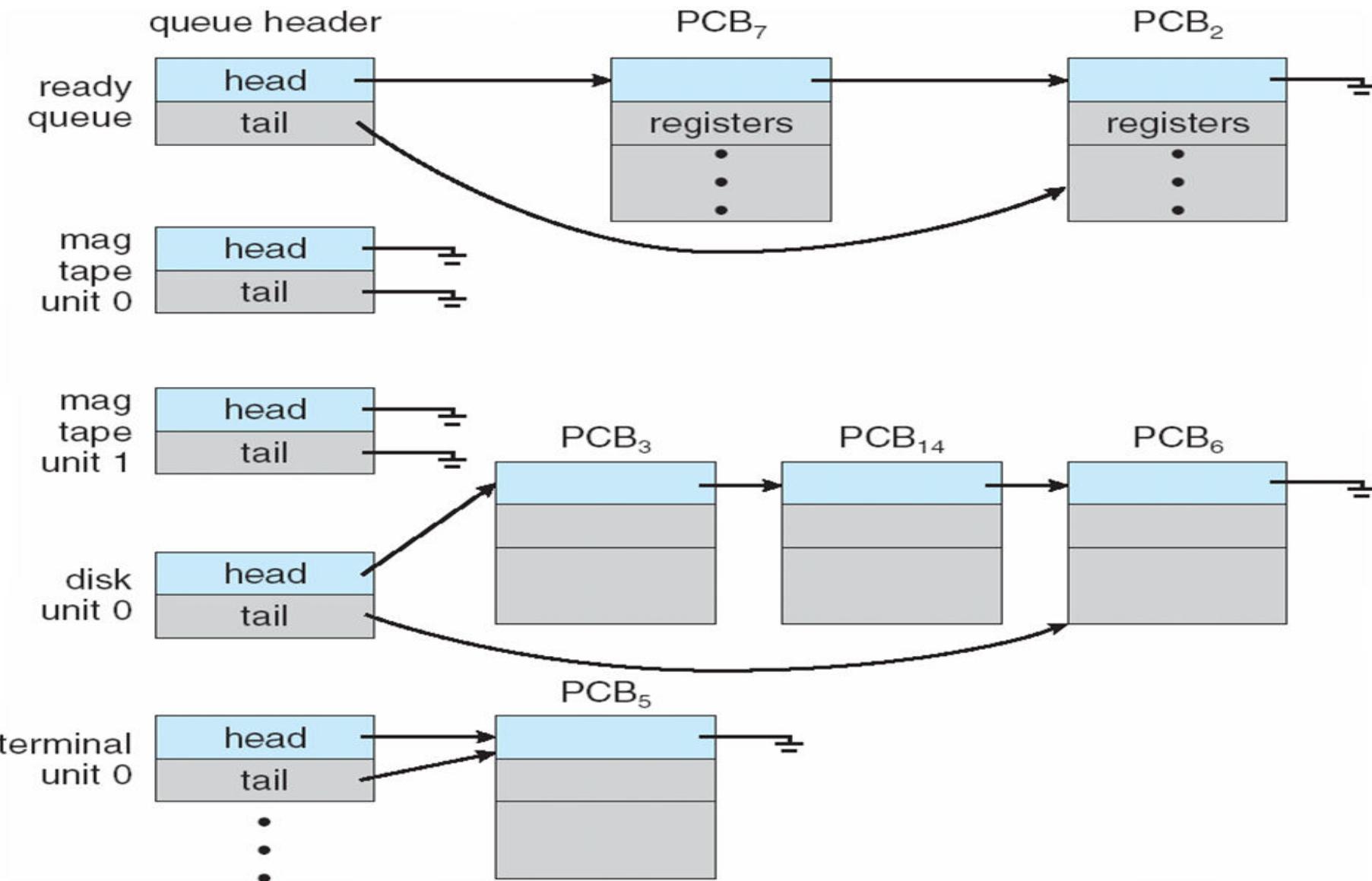
Process Scheduling

- **OS Objectives:** to maximize CPU utilization, and, to frequently switch among processes onto CPU for time sharing; **so that users can interact with programs**
- **Process scheduler** selects among available processes to be executed on CPU
 - Single-CPU system, multi-CPU system;
 - Process scheduler = ***CPU scheduler + Job scheduler + other schedulers***
- Maintains **scheduling queues** of processes
 - **Job queue** – set of all processes in the system
 - **Ready queue** – set of all processes ***residing in main memory***, ready and waiting to execute.
 - ▶ = Linked list of PCBs
 - **Device queues** – set of processes waiting for an I/O device
 - ▶ Each shared device has its associated device queue
 - Processes migrate among the various queues





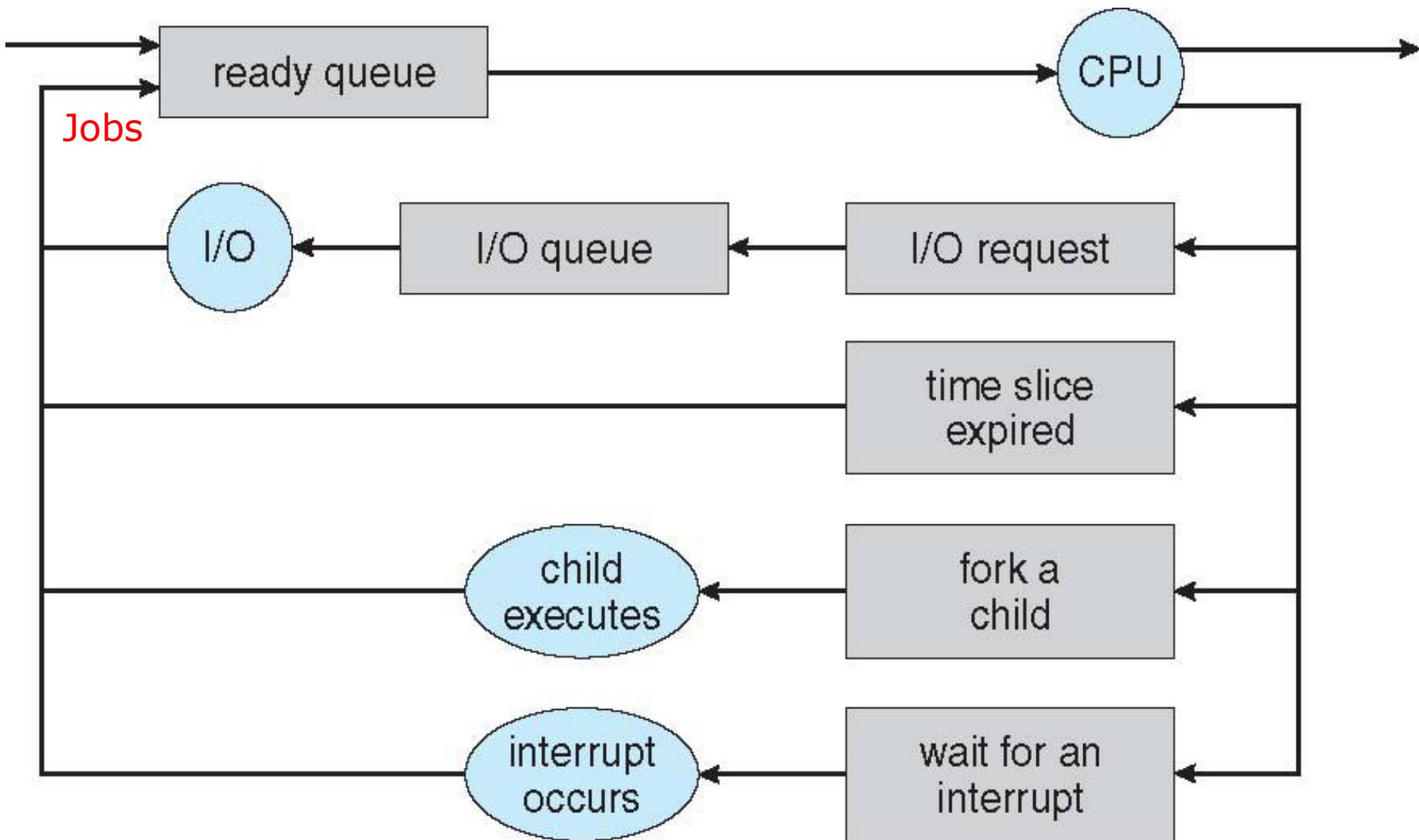
Ready Queue And Various I/O Device Queues

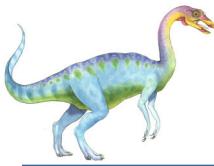




Representation of Process Scheduling

- Queueing diagram represents queues, resources, flows





Schedulers

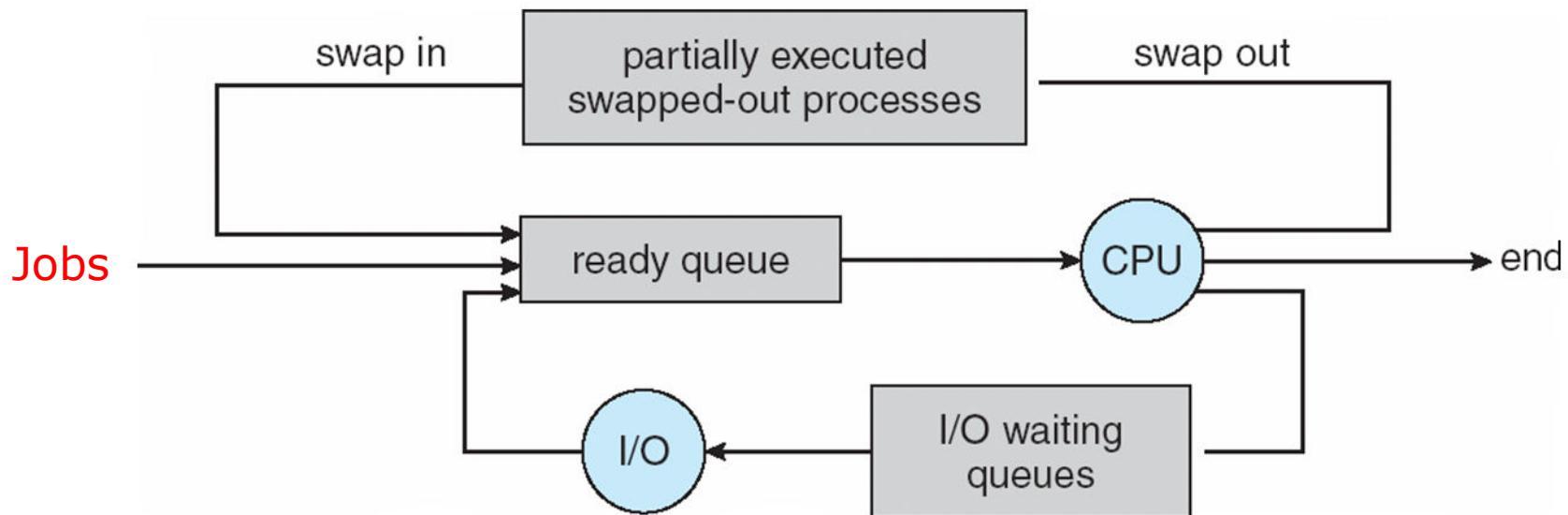
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates the CPU to that process
 - Sometimes the only scheduler in a system. **Time-sharing systems (UNIX, MS Windows)**
 - Short-term scheduler is invoked frequently (milliseconds)  (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
 - Long-term scheduler is invoked infrequently (seconds, minutes)  (may be slow)
 - The long-term scheduler controls the **degree of multiprogramming**:
 - ▶ **= Number of processes in memory** (i.e., in the ready queue)
 - ▶ Stable degree: aver nb of proc's creations = aver nb of proc's departure
 - ▶ Thus, invoked only when a process leaves the system
- Processes can be described as either:
 - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts. **The ready queue is almost always empty if all processes are I/O-bound**
 - **CPU-bound process** – spends more time doing computations; few very long CPU bursts. **The I/O queue is almost always empty if all processes are CPU-bound**
- Long-term scheduler strives for good **process mix** of I/O-bound and CPU-bound proc's
 - So that both queues are never almost always empty





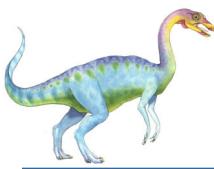
Addition of Medium Term Scheduling

- **Medium-term scheduler** added in some OS in order to reduce the degree of multi-programming (e.g. in some time-sharing systems)
 - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



- Swapping helps improve process mix
- Also necessary when memory needs to be freed up





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
 - = process state, all register values, memory information, ... etc
 - **Save/restore** contexts to/from PCBs when switching among processes
 - ▶ Known as **context switch**
- Context-switch time is **overhead**; the system does no useful work while switching
 - The more complex the OS and the PCB
 - ▶ → the longer the context switch. **Typical speed is a few milliseconds**
 - Depends on machine: memory speed, nb of registers, load/save instructions
- Time dependent on hardware support
 - Some hardware provides multiple sets of registers per CPU
 - ▶ → multiple contexts loaded at once

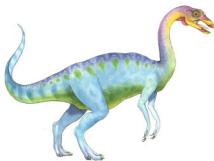




Operations on Processes

- Processes execute concurrently, are dynamically created/deleted
- Operating systems must provide mechanisms for:
 - process creation,
 - process termination,
 - and so on as detailed next





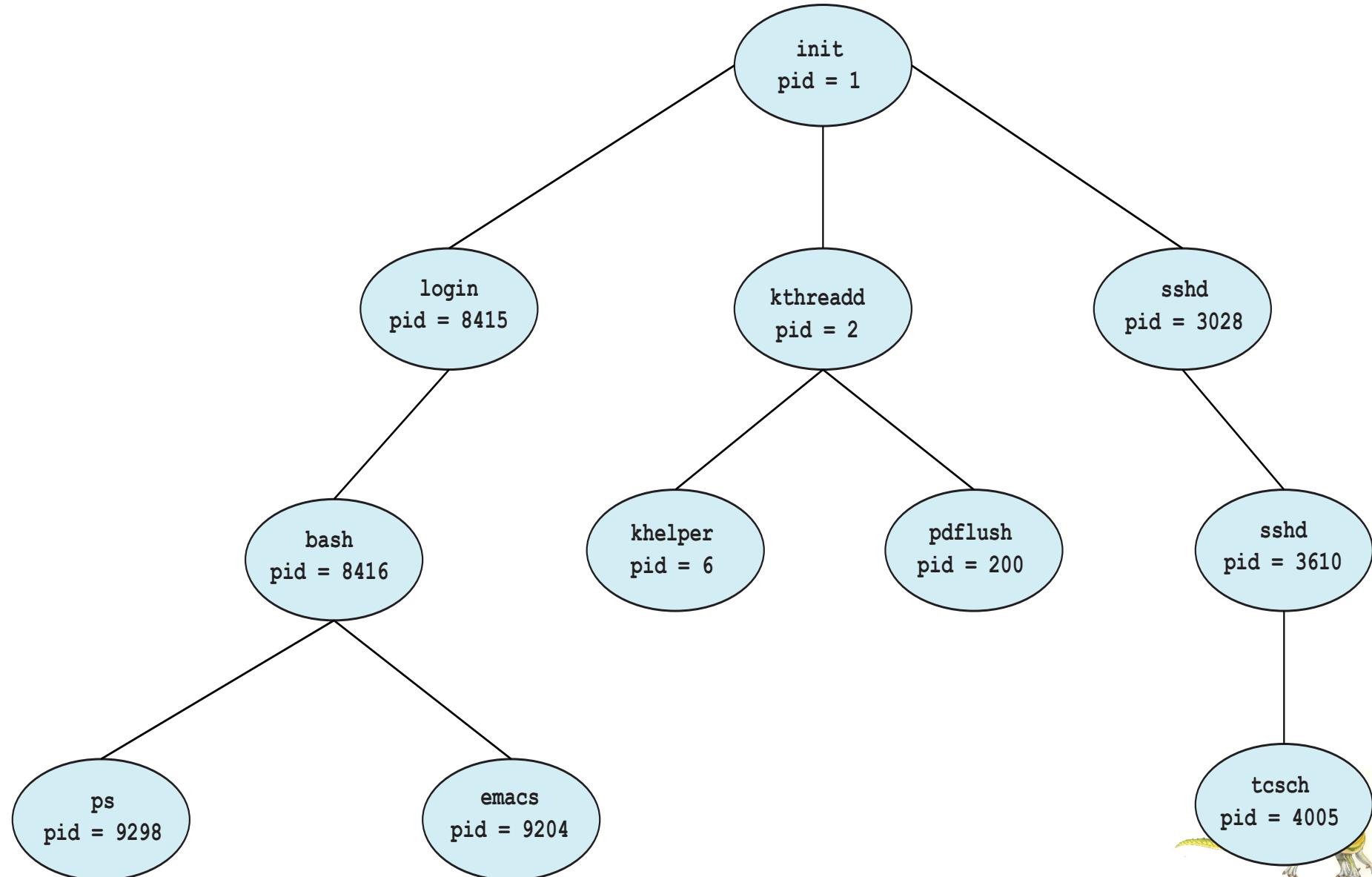
Process Creation

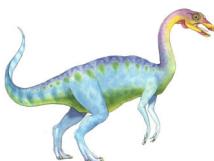
- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, a process is identified and managed via a process identifier (pid)
 - Unique handle to access various attributes of a process
- Resource sharing options
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution options when a process creates a new process
 - Parent and children execute concurrently
 - Parent waits until children terminate





A Tree of Processes in Linux

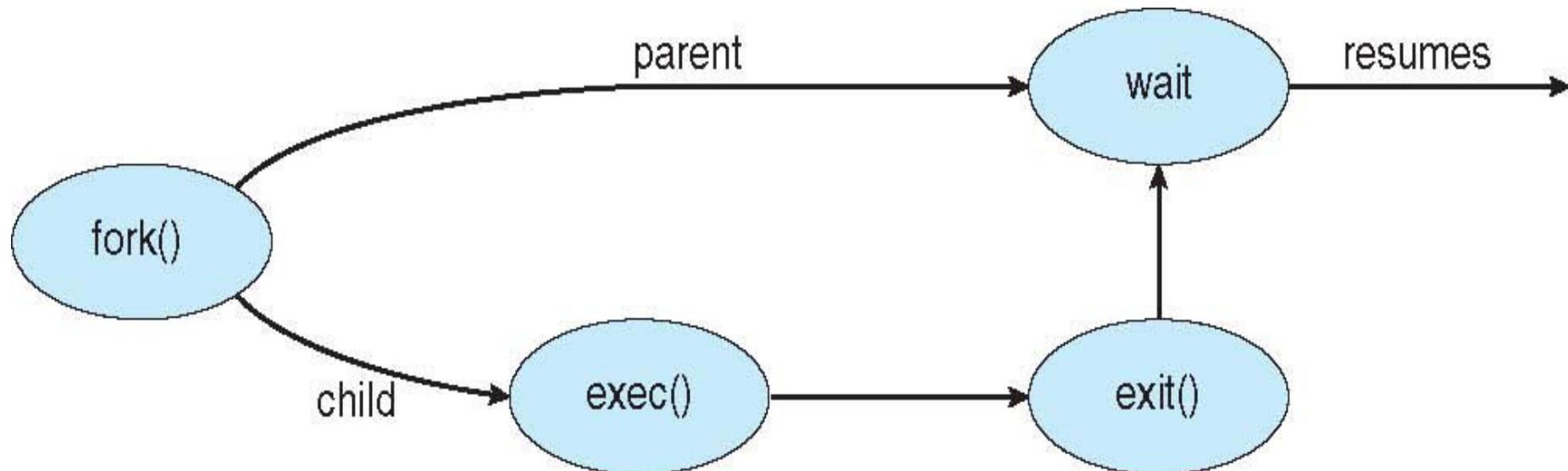


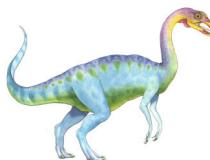


Process Creation (Cont.)

- Address-space options **when a process creates a new process**
 - Child is a duplicate of parent
 - Child has a **new** 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





C Program Forking Separate Process

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

int main()
{
    pid_t pid;

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

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

    return 0;
}
```



Creating a Separate Process via Windows API

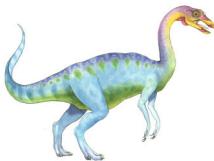
```
#include <stdio.h>
#include <windows.h>

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

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

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

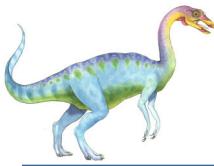
/* close handles */
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
}
```



Process Termination

- Process executes last statement and then asks the operating system to delete it using the `exit()` system-call.
 - Returns status data from child to parent (via `wait()`)
 - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the `abort()` [or `TerminateProcess()`] system-call. Some reasons for doing so:
 - [Parent needs to know the identities of its children]
 - Child has exceeded allocated resources
 - ▶ Parent must have a mechanism to inspect its children
 - Task assigned to child is no longer required
 - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates





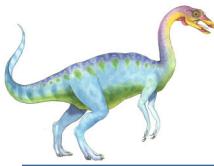
Process Termination

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

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

 - ▶ We can directly terminate a child using `exit(1)` with status parameter
 - ▶ `wait()` is passed a parameter allowing prt to obtain exit status of child
 - ▶ Parent knows which children has terminated
- **Zombie:** If parent has not yet invoked `wait()` but child process has terminated
- **Orphan:** If parent has terminated without invoking `wait` child process is alive





Interprocess Communication

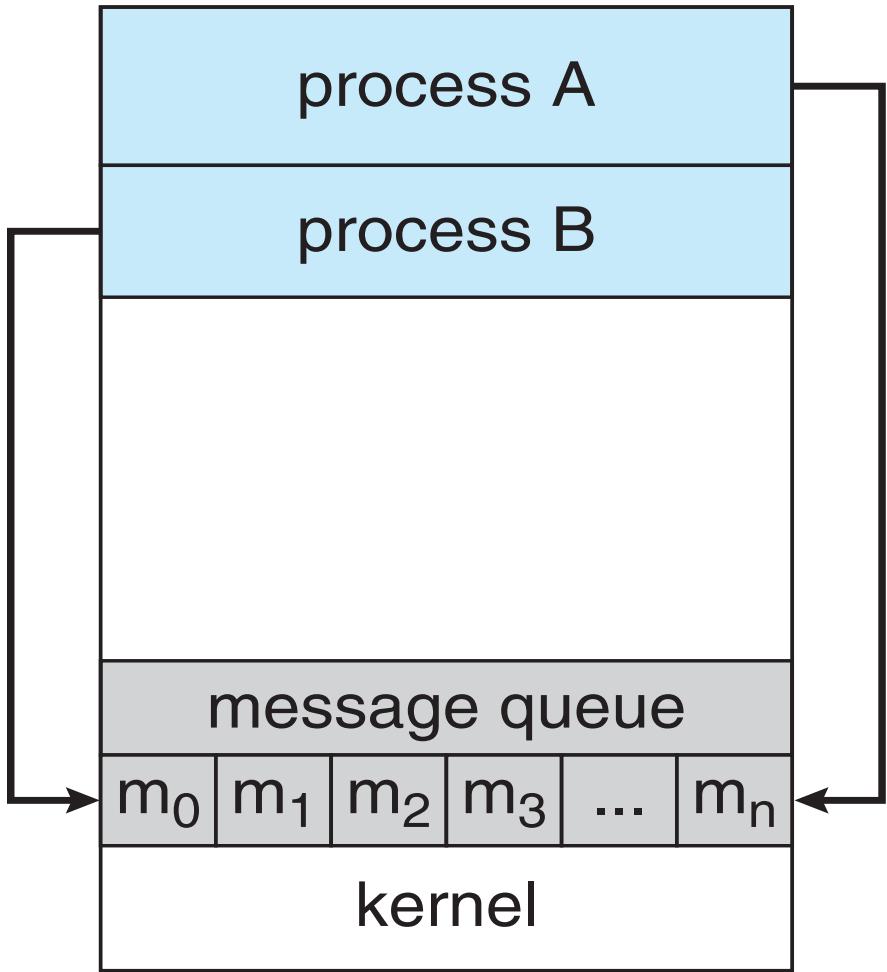
- Processes within a system may be ***independent*** or ***cooperating***
- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
 - Information sharing; **many users sharing the same file**
 - Computation speedup; **in multi-core systems**
 - Modularity; **recall Chap 2**
 - Convenience; **same user working on many tasks at the same time**
- Cooperating processes need **interprocess communication (IPC)** mechanism to exchange data and information
- Two models of IPC
 - ▶ Many OS's implement both IPC models
 - **Shared memory**; easier to implement and faster
 - **Message passing**; useful for exchanging small amounts of data





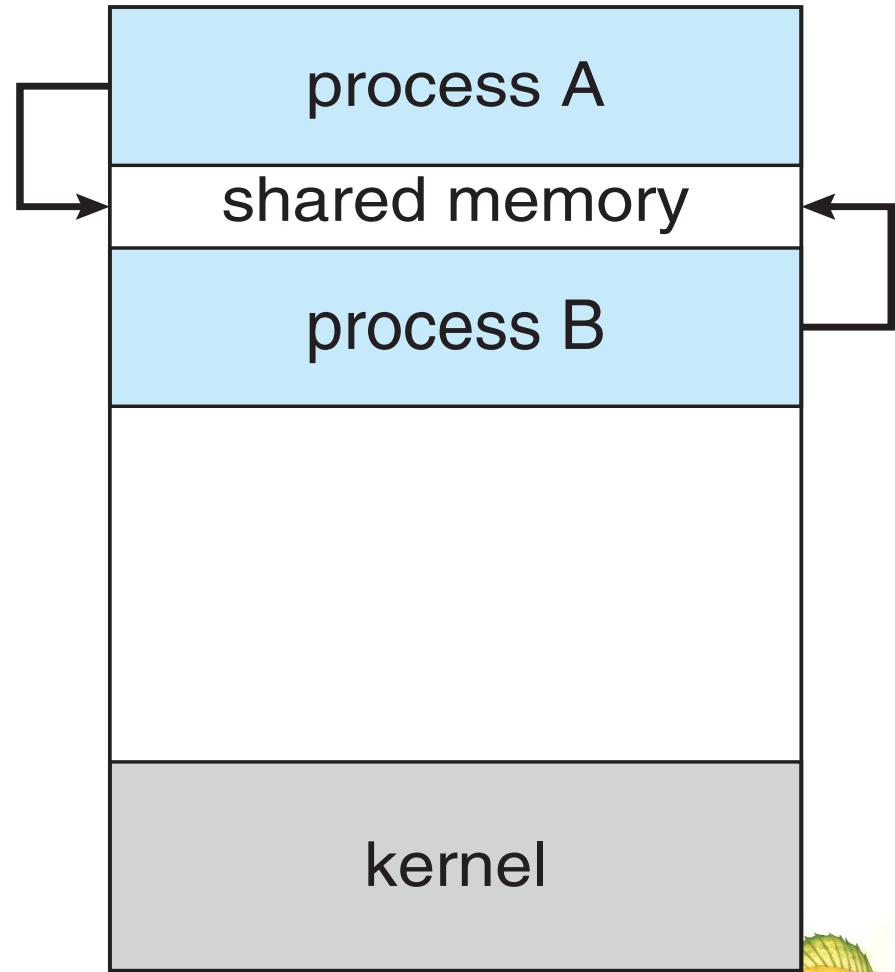
Communications Models

(a) Message passing.



(a)

(b) shared memory.



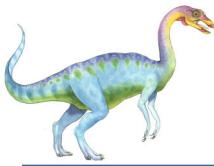
(b)



Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
 - Normally, OS prevent a process from accessing another process's memory.
 - ▶ Processes can agree to remove this restriction in shared-memory systems
- The communication is under the control of the users processes not the operating system.
 - Application programmer ***explicitly*** writes the code for sharing memory
 - Processes ensure that they not write to the same location simultaneously
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
 - Solution to the ***producer-consumer problem***
 - ▶ Discussed in following slides
- Synchronization is discussed in great details in Chapter 5.





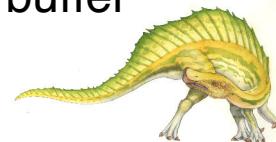
Shared-Memory Systems

■ Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
- Example:
 - ▶ Compiler outputs (**produces**) an assembly code
 - ▶ Assembler assembles (**consumes**) the assembly code
 - ▶ Provides a metaphor for the client-server paradigm
 - Server = producer. Ex: web server **provides** HTML files/images
 - Client = consumer. Ex: client web browser **reads** HTML files/images

■ Solution: producer and consumer processes share a buffer (**shared-memory**)

- Synchronization: consumer should not consume data not yet produced
- Two types of buffers:
 - ▶ **unbounded-buffer** places no practical limit on the size of the buffer
 - ▶ **bounded-buffer** assumes that there is a fixed buffer size





SM: Bounded-Buffer – Shared-Memory Solution

- Shared data – buffer implemented as a circular array

```
#define BUFFER_SIZE 10

typedef struct {

    . . .

} item; item to be produced/consumed

item buffer[BUFFER_SIZE]; shared buffer

int in = 0; producer produces an item into in
int out = 0; consumer consumes an item from out
```

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

- Empty if **in = out** and Full if $((in + 1) \% BUFFER_SIZE) = out$





SM: Bounded-Buffer – Producer

```
item next_produced;           stores new item to be produced  
...  
while (true)  
{  
    /* produce an item in next_produced */  
  
    while (((in + 1) % BUFFER_SIZE) == out)  
        ; /* do nothing      when the buffer is full*/  
  
    buffer[in] = next_produced;      item is produced  
  
    in = (in + 1) % BUFFER_SIZE;    update in pointer  
}
```





SM: Bounded Buffer – Consumer

```
item next_consumed;           stores item to be consumed  
...  
while (true)  
{  
    while (in == out)  
        ; /* do nothing when the buffer is empty */  
  
    next_consumed = buffer[out];   item is consumed  
  
    out = (out + 1) % BUFFER_SIZE; update out pointer  
  
    /* consume the item in next_consumed */  
}
```

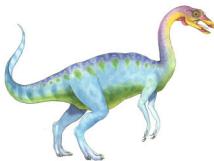




Message-Passing Systems

- Mechanism for processes to communicate and to synchronize their actions
 - Useful when communicating processes are in different computers
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
 - A **communication link** must exist between communicating processes, then
 - ▶ Communication operations:
 - **send(message)**
 - **receive(message)**
- The *message* size is either fixed-sized or variable-sized



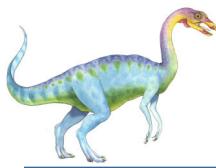


Message-Passing Systems

- If processes P and Q wish to communicate, they need to:
 - Establish a ***communication link*** between them
 - Exchange messages via send/receive

- Implementation issues: (we are concerned only with its logical implementation)
 - How are links established?
 - Can a link be associated with more than two processes?
 - How many links can there be between every pair of communicating processes?
 - What is the capacity of a link?
 - Is the size of a message that the link can accommodate fixed or variable?
 - Is a link unidirectional or bi-directional?





Message-Passing Systems

- Implementation of communication link
 - ▶ (we are concerned only with its logical implementation)
- Physical:
 - ▶ Shared memory; not the same as the logical SM discussed in Slide 29
 - ▶ Hardware bus
 - ▶ Network
- *Logical:*
 - ▶ Direct or indirect communication
 - ▶ Synchronous or asynchronous communication
 - ▶ Automatic or explicit buffering





MP: 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
- Direct communication schemes
 - Symmetric: both sender and receiver must name the other to communicate
 - Asymmetric: only the sender names the recipient
 - ▶ **send**(*P, message*) – send a message to process *P*
 - ▶ **receive**(*id, message*) – receive a message from **any** process *id*
- Problem with direct communication
 - Must explicitly state all process identifiers -- **hard-coding**





MP: Indirect Communication

- Messages are sent to 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





MP: Indirect Communication

- OS provides operations allowing a process to
 - create a new mailbox M (also called a port)
 - ▶ The **owner** is the process that creates the mailbox M
 - It can only receive messages through this mailbox M
 - ▶ A **user** is the process which can only send messages to this mailbox M
 - send and receive messages through mailbox
 - destroy a mailbox
- Primitives are defined as:

send(A , message) – send a message to mailbox A

receive(A , message) – receive a message from mailbox A





MP: Indirect Communication

■ Mailbox sharing

- Suppose processes P_1 , P_2 , and P_3 share mailbox A
- P_1 , sends a message to A by executing `send(A, message)`
- P_2 and P_3 execute `receive(A, message)`
 - ▶ Who gets the message? ... P_2 and P_3 ?

■ 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.
 - ▶ Round robin algorithm where processes take turn in receiving messages
 - ▶ Sender is notified who the receiver was.





MP: Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
 - **Blocking send** -- the sender is blocked until the message is delivered
 - **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 continues
 - **Non-blocking receive** -- the receiver retrieves:
 - A valid message, or
 - Null message
- Different combinations possible
 - If both send() and receive() are blocking, we have a **rendezvous**





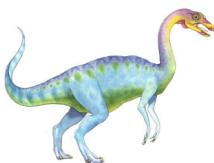
MP: Synchronization

- In a rendezvous, producer-consumer becomes trivial

```
message next_produced;  
while (true)  
{    producer invokes blocking send and waits until mess. delivered  
    /* produce an item in next produced */  
    send(M, next_produced);  
}
```

```
message next_consumed;  
while (true)  
{    consumer invokes blocking receive and waits until mess. available  
    receive(M, next_consumed);  
    /* consume the item in next consumed */  
}
```





MP: Buffering

- Queue of messages is attached to the communication link.
- Implemented in one of three ways
 1. Zero capacity – no messages are queued on a link.
Sender must wait for receiver (rendezvous) **to receive the message**
 1. **It means: there is no buffering: no message is waiting on the link**
 2. Bounded capacity – queue has a finite length of n messages
Sender must wait if link is full. **If not, then**
 2. **Messages are placed on the buffer without waiting for receiver to receive**
 3. Unbounded capacity – infinite length
Sender never waits





Examples of IPC Systems – POSIX SM

■ POSIX Shared-Memory (message-passing is also available in POSIX)

- Process first creates shared memory segment (*return int file desc for the sm*)
`shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`
 - Also used to open an existing segment to share it
- Set the size of the object: `ftruncate(shm_fd, 4096);`
- Map the shared memory to a file (*return pointer to the memory-mapped file*)
`shm_ptr = (0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);`
 - ▶ We use `shm_ptr` to access the shared-memory object `shm_fd`
- Now the process could write to the shared memory
`sprintf(shm_ptr, "Writing to shared memory");`
- Remove the shared memory object: `shm_unlink(name);`





IPC POSIX SM Producer

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

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

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

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

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

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

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

    return 0;
}
```



IPC POSIX SM 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 MP

- Mach communication is message based on message-passing
 - ▶ Especially designed for distributed systems or systems with few cores
 - Even system calls are messages
 - Each task gets two mailboxes at creation – Kernel-port and Notify-port
 - Only three system calls needed for message transfer
 - `msg_send()` , `msg_receive()` , `msg_rpc()` [remote procedure call]
 - `msg_rpc` sends message and wait for 1 return message from sender
 - Mailboxes needed for communication: created via `port_allocate()`
 - ▶ Empty queue of length 8 messages is also created for the link
 - 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





Examples of IPC Systems – Windows MP

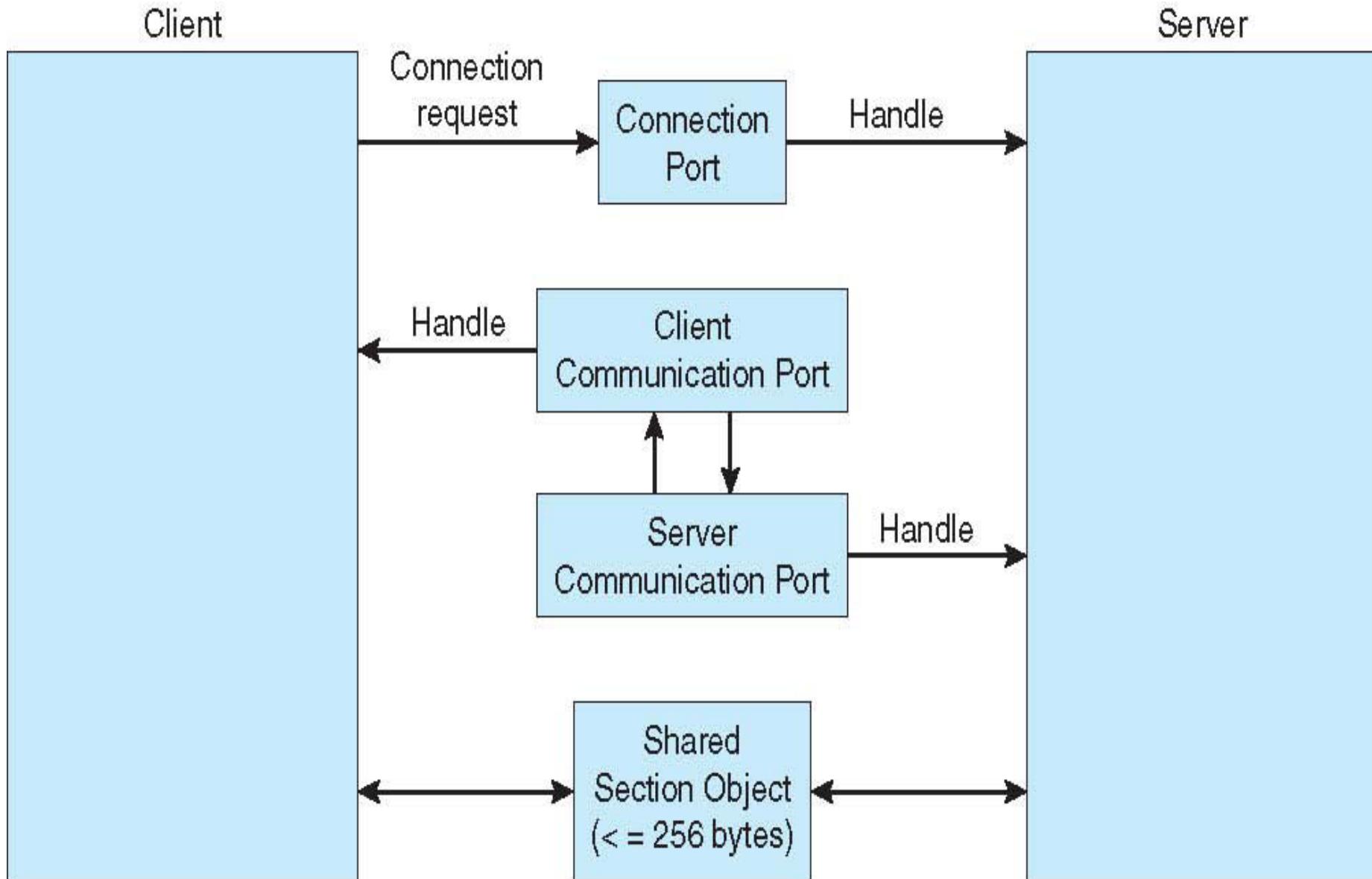
- Message-passing centric via **advanced local procedure call (LPC)** facility

- Only works between processes on the same system
- Uses ports (like mailboxes) to establish and maintain communication channels. **Two types of ports: connection port and communication port**
- 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.

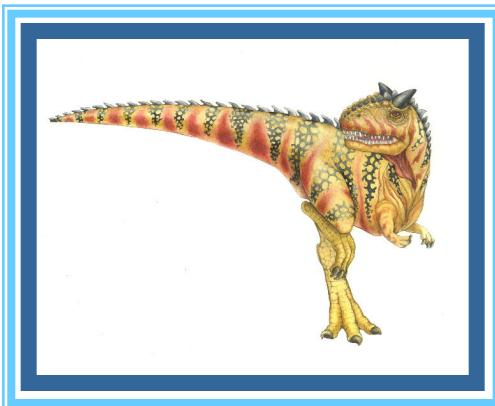




MP: Local Procedure Calls in Windows



End of Chapter 3





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 and user interface limits, iOS provides for a
 - Single **foreground** process- controlled via user interface
 - Multiple **background** processes– in memory, running, but not on the display, and with limits
 - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
 - Background process uses a **service** to perform tasks
 - Service can keep running even if background process is suspended
 - Service has no user interface, small memory use





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





Communications in Client-Server Systems

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





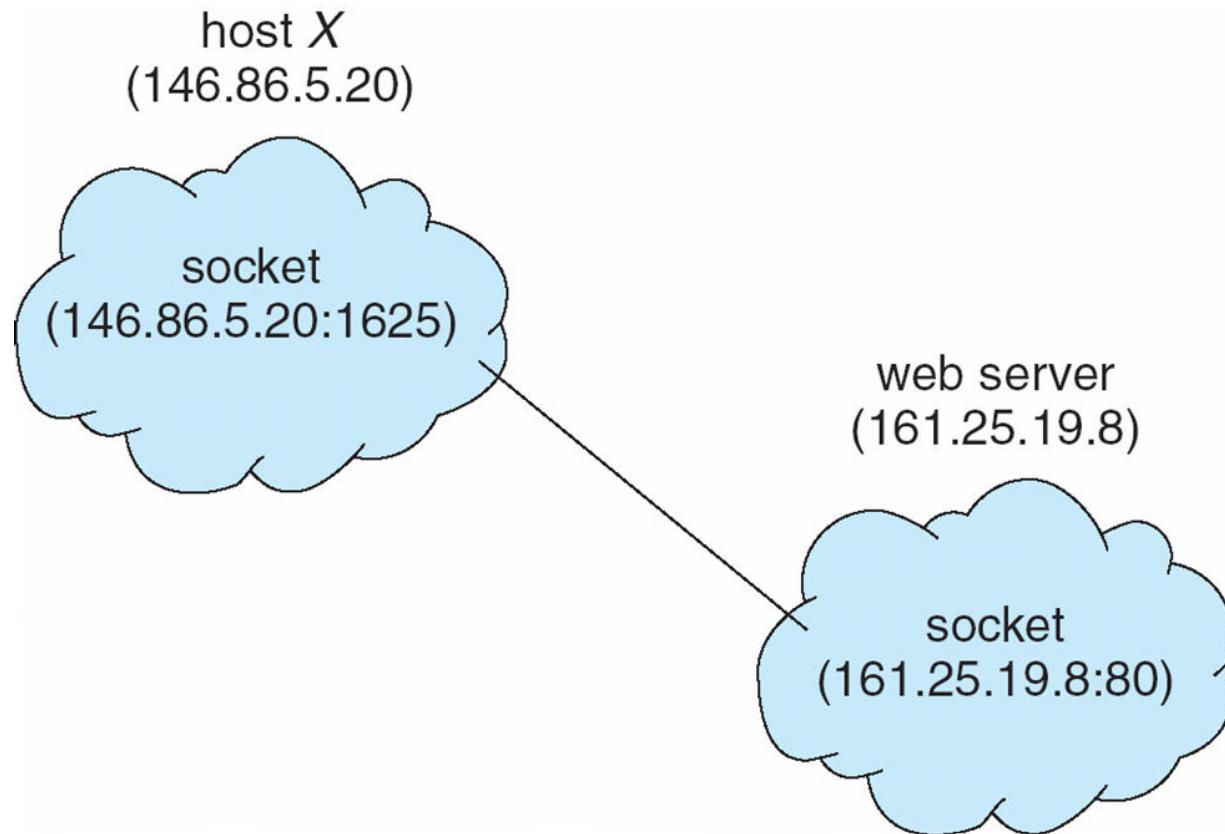
Sockets

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





Socket Communication





Sockets in Java

■ Three types of sockets

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

■ Consider this “Date” server:

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

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

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

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

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

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





Remote Procedure Calls

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





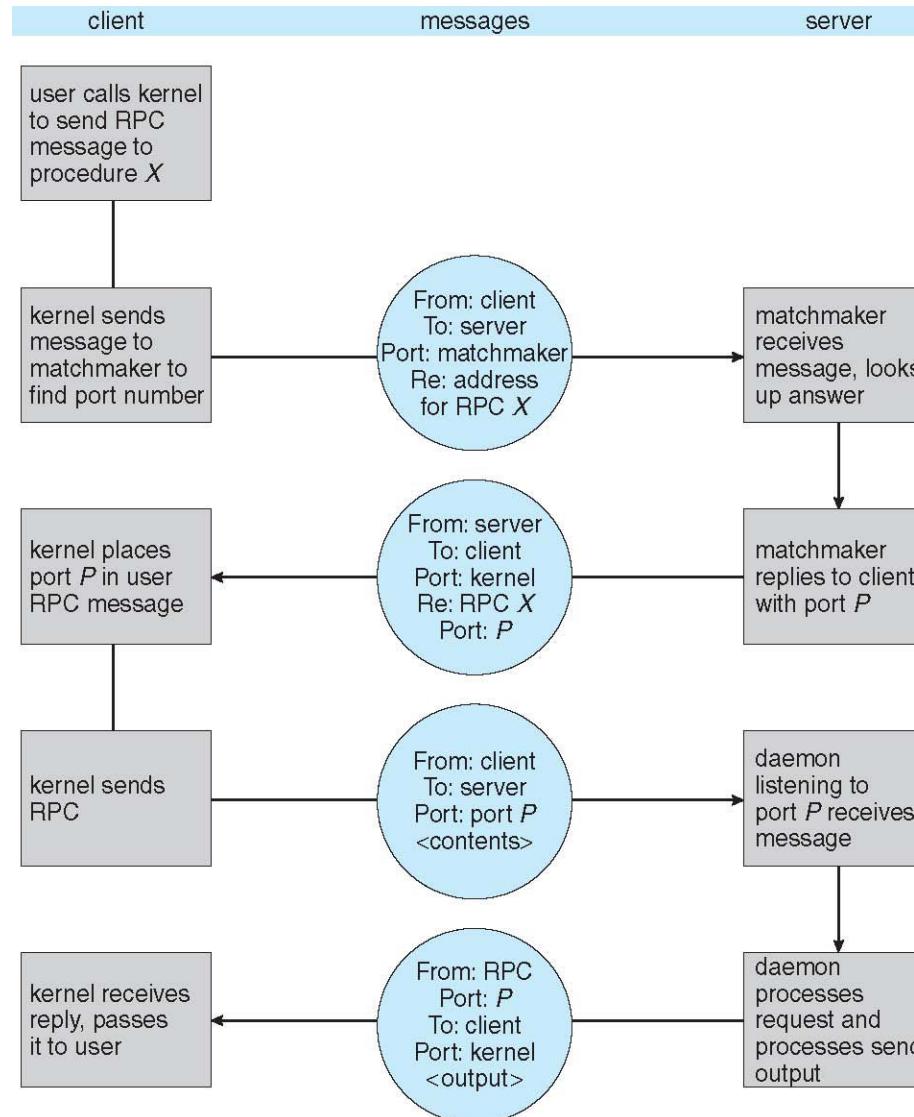
Remote Procedure Calls (Cont.)

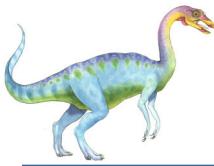
- Data representation handled via **External Data Representation (XDL)** format to account for different architectures
 - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
 - Messages can be delivered ***exactly once*** rather than ***at most once***
-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





Execution of RPC





Pipes

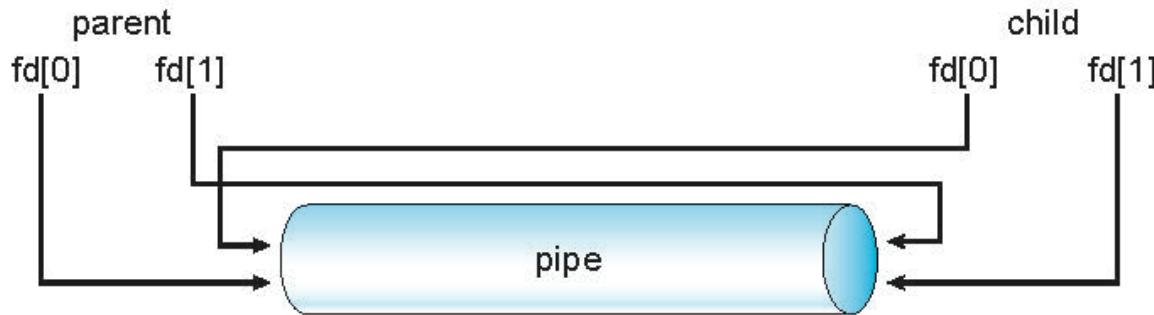
- Acts as a conduit allowing two processes to communicate
- Issues:
 - Is communication unidirectional or bidirectional?
 - In the case of two-way communication, is it half or full-duplex?
 - Must there exist a relationship (i.e., **parent-child**) between the communicating processes?
 - Can the pipes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





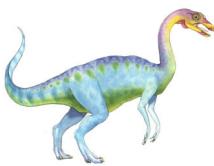
Ordinary Pipes

- 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**
- See Unix and Windows code samples in textbook



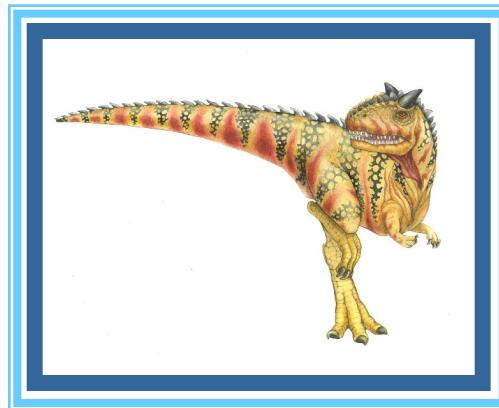


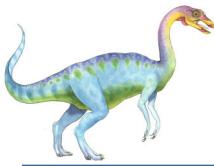
Named Pipes

- 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



Chapter 4: Threads

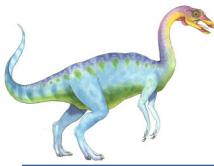




Chapter 4: Threads

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





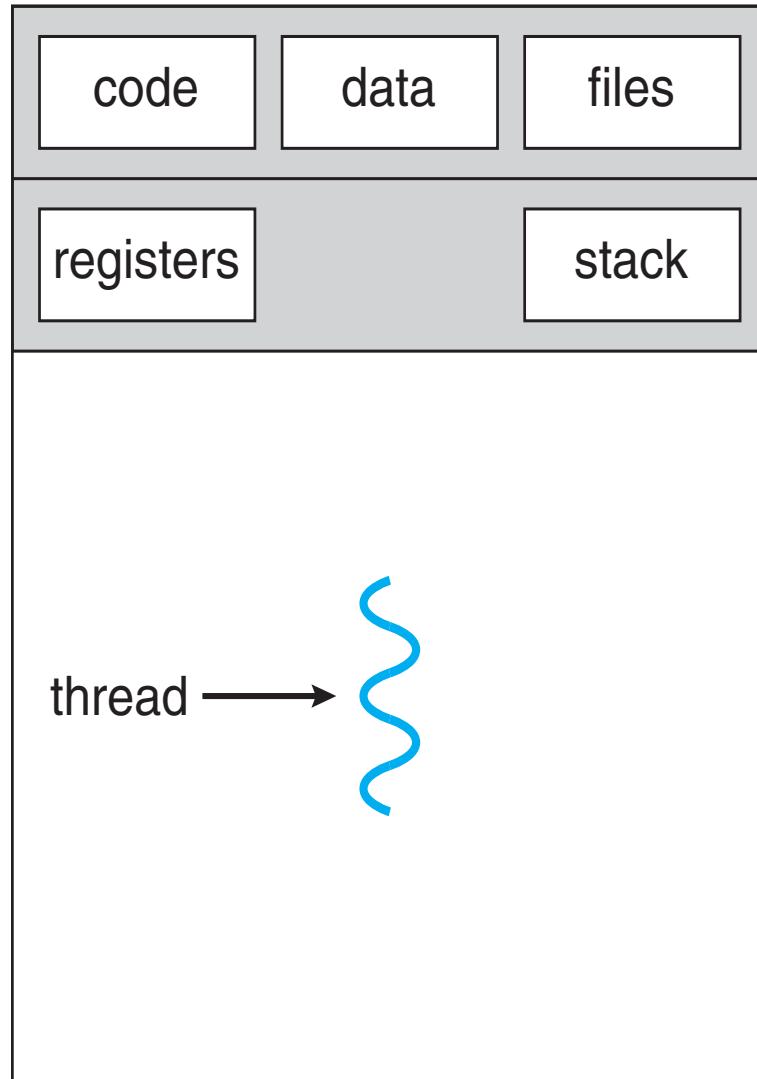
Objectives

- ***Thread*** of a process: basic unit of CPU utilization and is composed of a:
 - Thread ID
 - Program counter: register EIP
 - Register set
 - Stack
 - And ***it shares with other threads of the same process***, the
 - ▶ Code segment
 - ▶ Data segment
 - ▶ OS resources: open files, signals, ... etc
- To discuss the APIs for the Pthreads, Windows, and Java thread libraries
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux

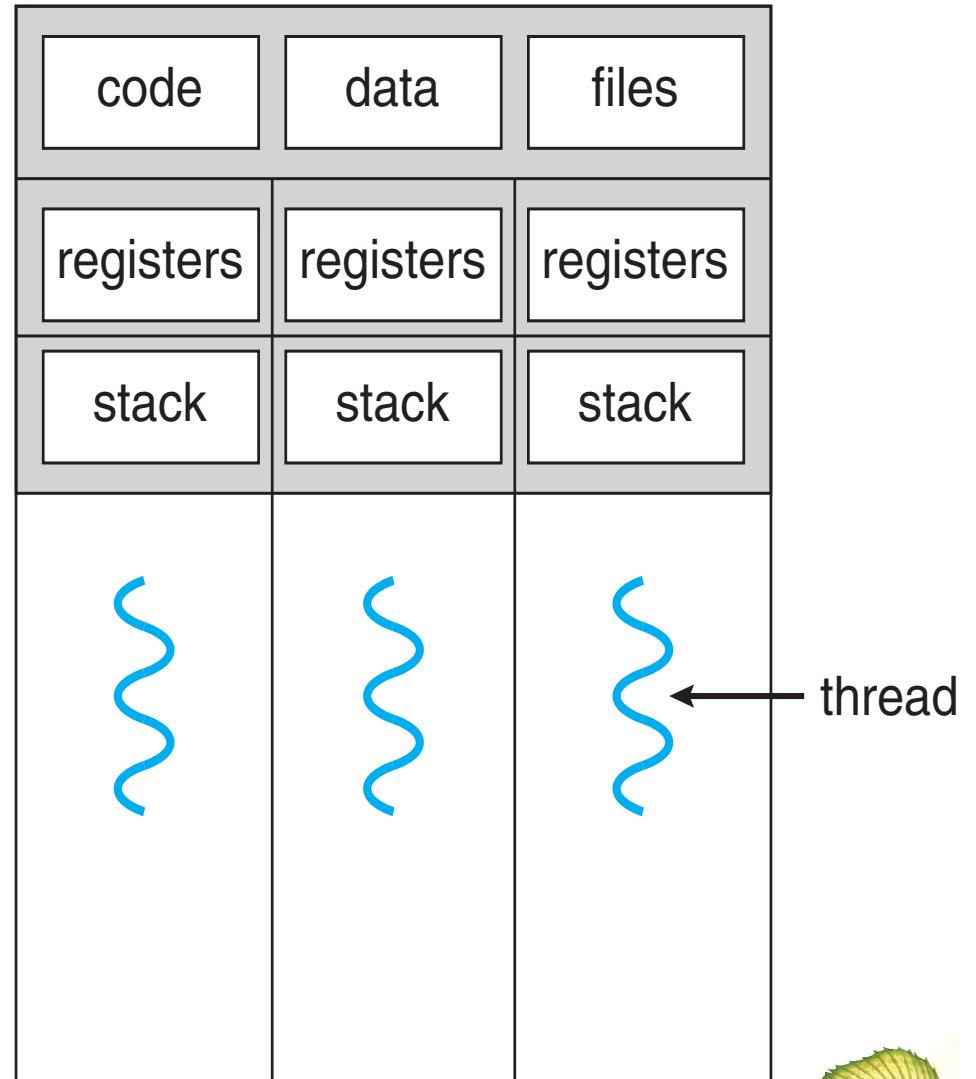




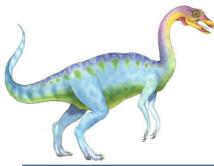
Single and Multithreaded Processes



single-threaded process



multithreaded process



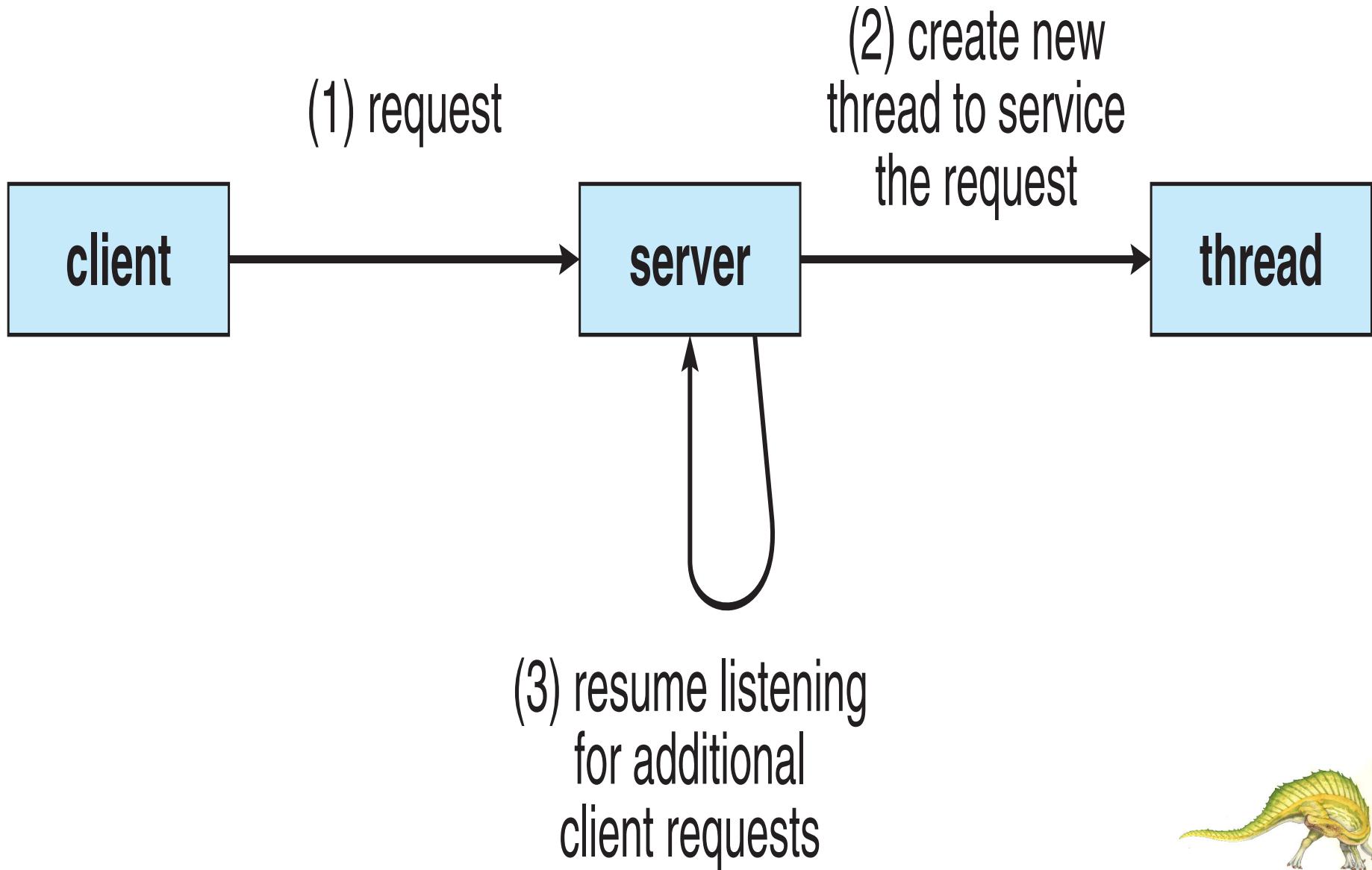
Motivation

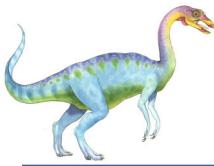
- Most modern applications **and computers** are multithreaded
- Threads run within application
- Multiple tasks within the application can be implemented by separate threads
 - Update display
 - Fetch data
 - Spell checking
 - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
 - For applications performing ***multiple similar*** tasks
 - ▶ It is costly to create a process for each task
 - Ex: web server serving multiple clients requesting the same service
 - ▶ Create a separate thread for each service
- Multithreaded programs: can simplify code, increase efficiency.
- Kernels are generally multithreaded. **Interrupt handling, device management, etc**





Multithreaded Server Architecture





Benefits

■ Responsiveness

- may allow continued execution if part (i.e. **thread**) of a process is time-consuming, especially important for user interfaces. **User needs not wait**

■ Resource Sharing

- threads share resources of process, easier than shared-memory or message-passing. **Since programmer need not explicitly code for communication between threads (see: codes for shared-memory or message-passing, in Chap-3)**

■ Economy

- cheaper than process creation, thread switching has lower overhead than context switching. **See figure on Slide-4: thread share most of process's PCB; thus, faster switching**

■ Scalability

- process can take advantage of multiprocessor architectures. **Threads can run in parallel on different processing cores. See figure on Slide-4 again**

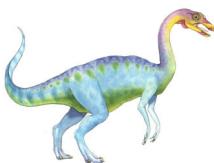




Multicore Programming

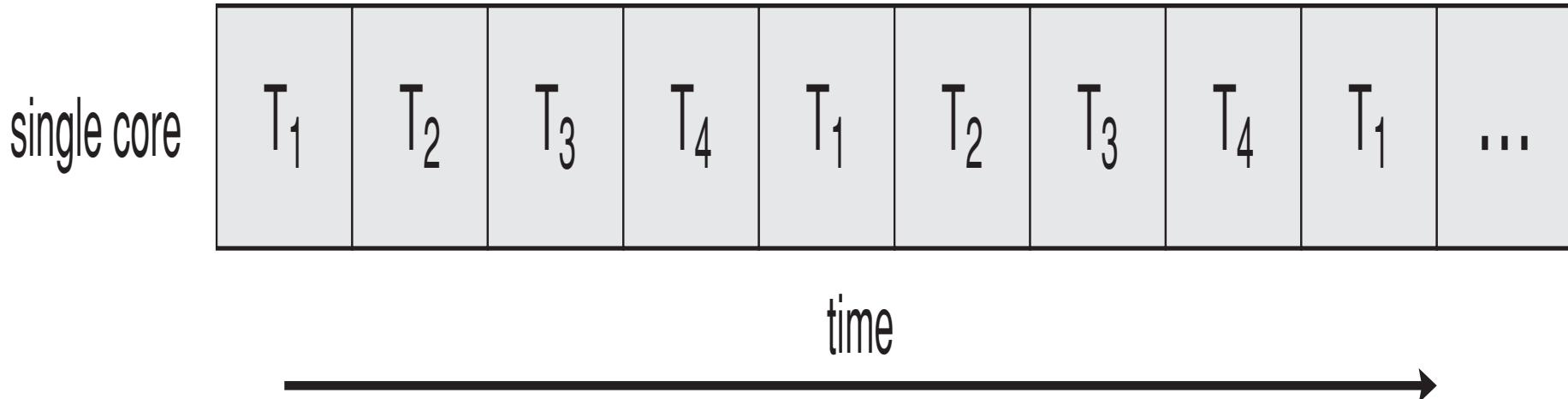
- Multicore or multiprocessor systems put pressure on programmers. OS designers must write CPU scheduling algorithms that use multiple cores. App programmers must design multithreaded programs. Challenges include:
 - **Identifying tasks:** What are the separate independent threads ?
 - **Balance:** How to divide the work (set of tasks) fairly among CPU cores ?
 - **Data splitting:** How to divide the data fairly among CPU cores ?
 - **Data dependency:** What do to when there is dependency between threads ?
 - **Testing and debugging:** What is a correct parallel/concurrent/multithreaded program?
- Multithreaded programming provides a mechanism for ***more efficient use of multiple cores and improved concurrency***. New software design paradigm ?
- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress on same core/CPU
 - Single processor / core. CPU scheduler providing concurrency



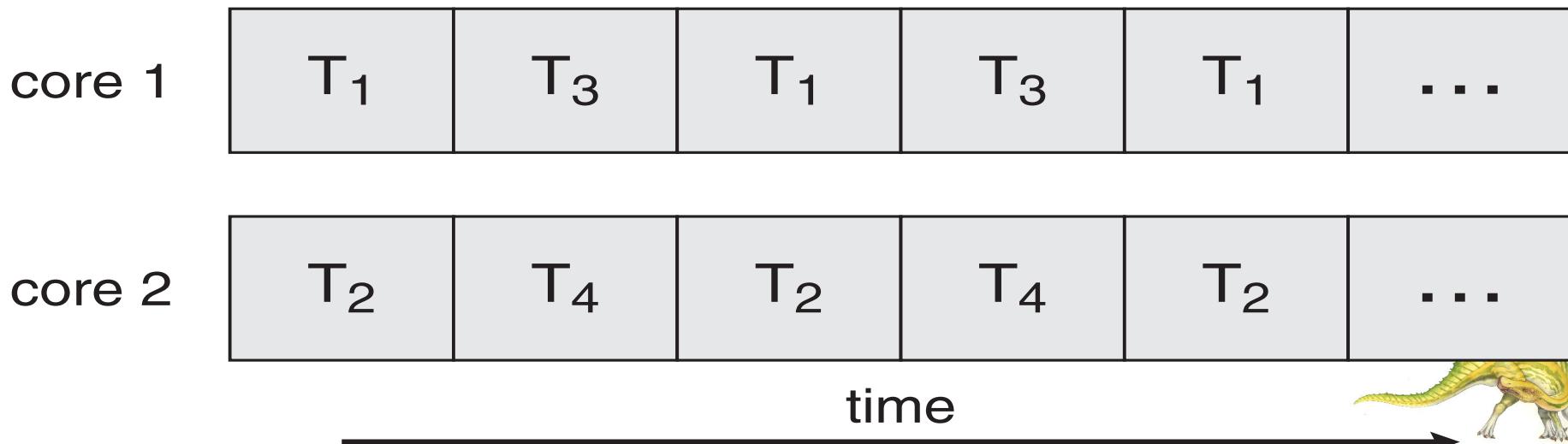


Concurrency vs. Parallelism

- Concurrent execution on single-core system:



- Parallelism on a multi-core system:





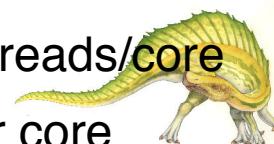
Multicore Programming

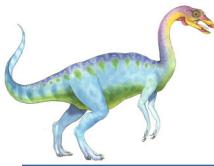
■ Types of parallelism (and concurrencies)

- **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
 - ▶ Ex: summing elements of a length- N array: 1-core vs 2-core system
- **Task parallelism** – distributing threads across cores, each thread performing unique operation. **Different threads may use same data or distinct data**
 - ▶ Ex: sorting and summing a length- N array: 1-core vs 2-core system
- **Hybrid Data-and-Task parallelism** in practice
 - ▶ Ex: sorting and summing a length- N array: 1-core, 2-core or 4-core system

■ As # of threads grows, so does architectural support for threading

- CPUs have cores as well as ***hardware threads***
- Modern Intel CPUs have two hardware threads, i.e. supports 2 threads/core
- Oracle SPARC T4 CPU has 8 cores, with 8 hardware threads per core





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

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

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

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

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





User Threads and Kernel Threads

- Support for threads either at user level or kernel level
- **User threads** - management done by user-level threads library
 - ▶ Supports thread **programming**: creating and managing program threads
 - Three primary thread libraries: (threads are managed without kernel support)
 - ▶ POSIX **Pthreads**
 - ▶ Windows threads
 - ▶ Java threads
- **Kernel threads** - Supported by the Kernel.
 - ▶ Managed directly by the OS
 - Examples – virtually all general purpose operating systems, including:
 - ▶ Windows
 - ▶ Solaris
 - ▶ Linux
 - ▶ Tru64 UNIX
 - ▶ Mac OS X

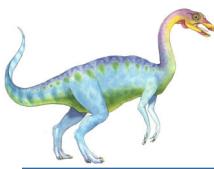




Multithreading Models

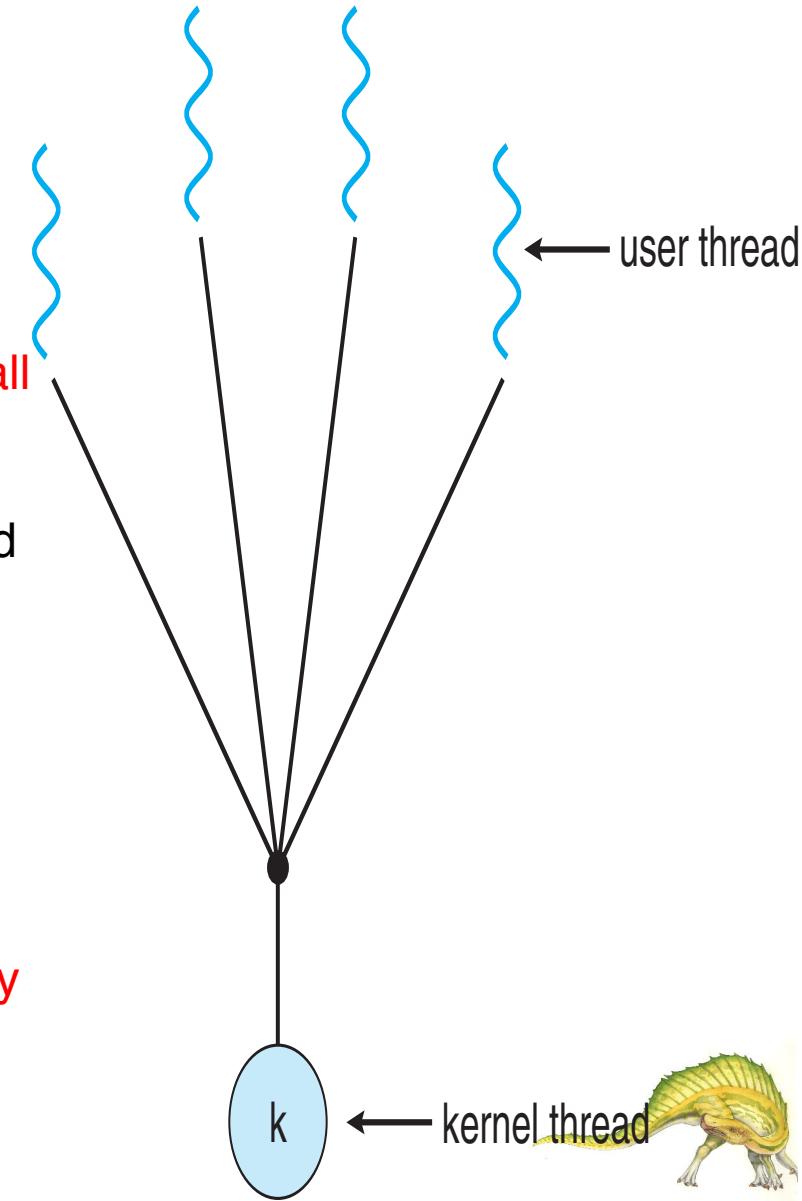
- A relationship must exist between user threads and kernel threads
 - Many-to-One
 - ▶ Many user-level threads mapped to a single kernel thread
 - One-to-One
 - ▶ Each user-level thread maps to one kernel thread
 - Many-to-Many
 - ▶ Allows many user-level threads to be mapped to many kernel threads

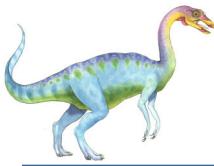




Many-to-One

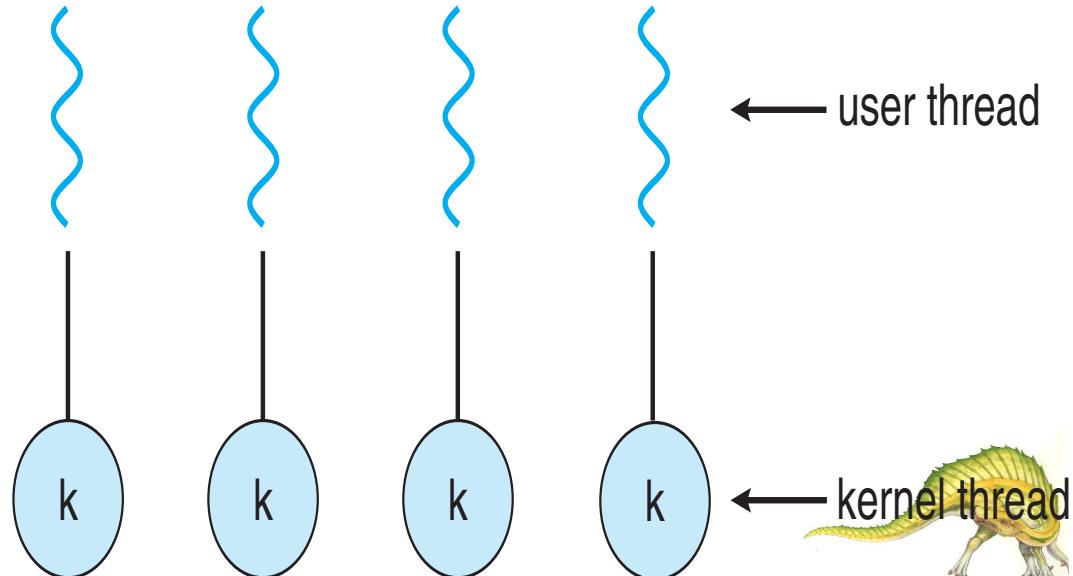
- Many user-level threads mapped to single kernel thread
 - Efficiently managed by the thread library
- One thread blocking causes all to block
 - If the thread makes a blocking system-call
- Multiple threads are unable to run in parallel on multicore system because only one thread can be in kernel at a time
 - Does not benefit from multiple cores
- Few systems currently use this model
- Examples of many-to-one models:
 - Solaris Green Threads (adopted in early versions of Java thread library)
 - GNU Portable Threads

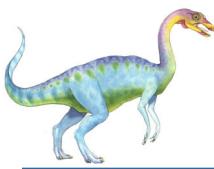




One-to-One

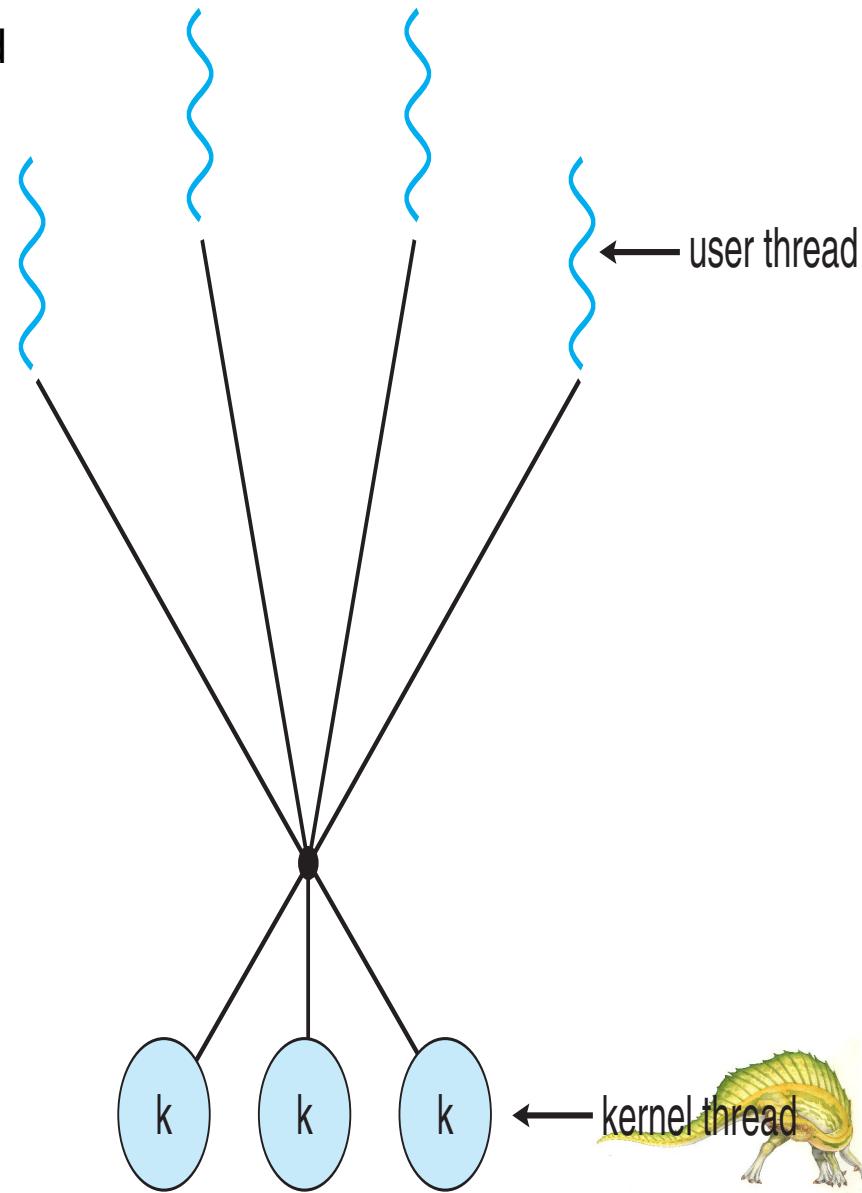
- Each user-level thread maps to one kernel thread
- Problem: creating a user thread requires creating the corresponding kernel thread
 - Thread ***creations*** burden the performance of an application; ***an overhead***
- Provides more concurrency than many-to-one model ***in case a thread has blocked***, and allows multiple threads to run in parallel on multiple CPU/core systems
- Number of threads per process sometimes restricted due to overhead
- Examples of one-to-one models
 - Windows
 - Linux
 - Solaris 9 and later

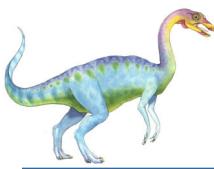




Many-to-Many Model

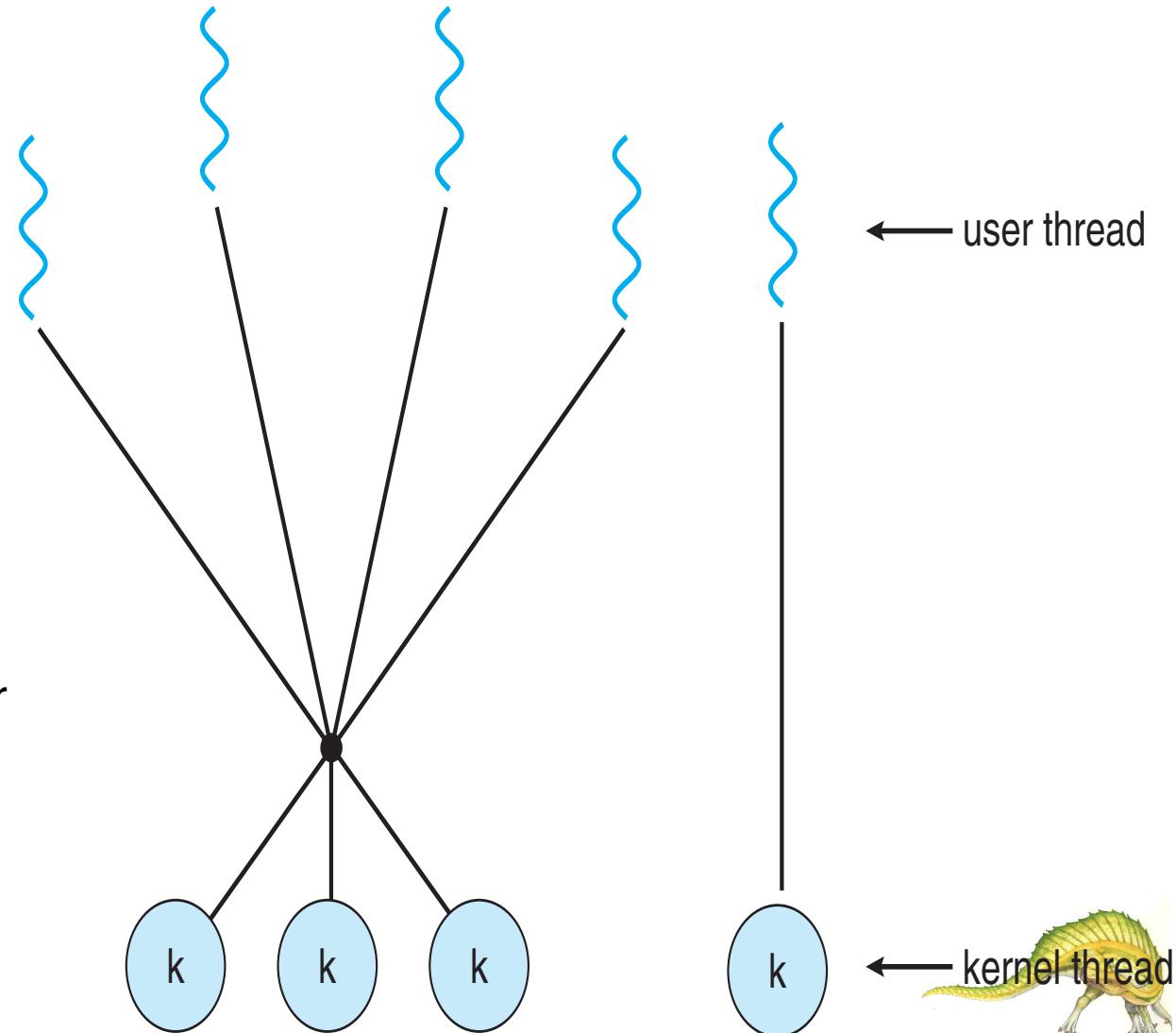
- Allows m user-level threads to be mapped to n kernel threads; $n \leq m$
- User can create as many user threads as wished
- Allows the operating system to create a sufficient number of kernel threads to be allocated to applications
- Does not have the problems of other models
- Solaris prior to version 9
- Windows with the *ThreadFiber* package





Two-Level Model

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

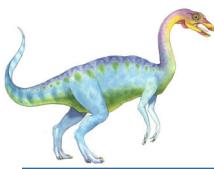




Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
 - Thread library is entirely in user space with no kernel support
 - ▶ Codes and data structures for thread library are available to the user
 - ▶ Thread library functions are *not* system-calls
 - Kernel-level thread library is supported directly by the OS
 - ▶ Codes and data structures for thread library are *not* available to the user
 - ▶ Thread library functions are system-calls to the kernel





POSIX Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- Threads extensions of POSIX may be provided either as user-level or kernel-level
- ***Specification, not implementation***
 - API specifies behavior of the thread library, implementation is up to development of the library. OS designers implement Pthreads specification in any way they wish
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)





Pthreads Example

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

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

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

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



Pthreads Example

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

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

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

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

    pthread_exit(0);
}
```



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

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

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



Windows Multithreaded C Program (Cont.)

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

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

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

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



Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:

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

- Extending Thread class
- Implementing the Runnable interface





Java Multithreaded Program

```
class Sum
{
    private int sum;

    public int getSum() {
        return sum;
    }

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

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

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

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



Java Multithreaded Program (Cont.)

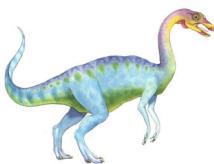
```
public class Driver
{
    public static void main(String[] args) {
        if (args.length > 0) {
            if (Integer.parseInt(args[0]) < 0)
                System.err.println(args[0] + " must be >= 0.");
            else {
                Sum sumObject = new Sum();
                int upper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



Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
 - Debugging an application containing 1000s of threads ?
- **Solution = *Implicit Threading*:** Let compilers and runtime libraries create and manage threads rather than programmers and applications developers
- Three methods explored
 - Thread Pools
 - OpenMP
 - Grand Central Dispatch
- Other methods include Microsoft Threading Building Blocks (TBB),
`java.util.concurrent` package





Thread Pools

- Create a number of threads **at process startup** in a pool where they await work
- Advantages:
 - Usually slightly faster to service a request with an existing thread than create a new thread. **A thread returns to pool once it completes servicing a request**
 - Allows the number of threads in the application(s) to be bound to the size of the pool. **Limits the number of threads that exist at any one point**
 - 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 **or after a time delay**
- Windows thread pool API supports thread pools:

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



OpenMP

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

#pragma omp parallel

Create as many threads as there are cores

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

Run for_loop in parallel

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

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

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

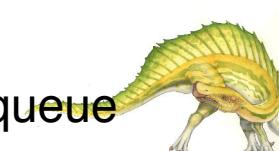
/* sequential code */

return 0;
}
```



Grand Central Dispatch

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





Grand Central Dispatch

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

```
dispatch_queue_t queue = dispatch_get_global_queue  
    (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);  
  
dispatch_async(queue, ^{ printf("I am a block."); });
```

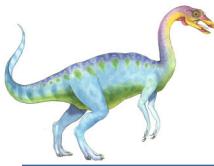


Operating System Examples

Windows Threads

- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
- Implements the one-to-one mapping, kernel-level
 - App run as separate processes, each process may contain many threads
- Each thread contains
 - A thread id
 - Register set representing state of processor
 - Separate user and kernel stacks; for threads running in user 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

- The primary data structures of a thread include:

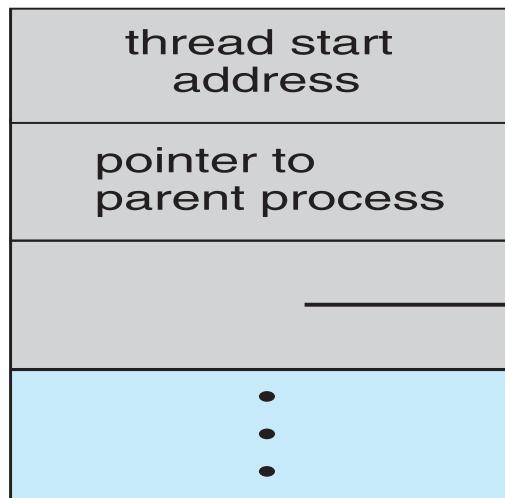
- ETHREAD (executive thread block) – includes: pointer to process to which thread belongs, **and address of routine in which the thread starts control**, and pointer to KTHREAD; **exists in kernel space only**
- KTHREAD (kernel thread block) – includes: scheduling and synchronization info, kernel-mode stack, pointer to TEB; **exists in kernel space only**
- TEB (thread environment block) – includes: thread id, user-mode stack, thread-local storage; **exists in user space only**



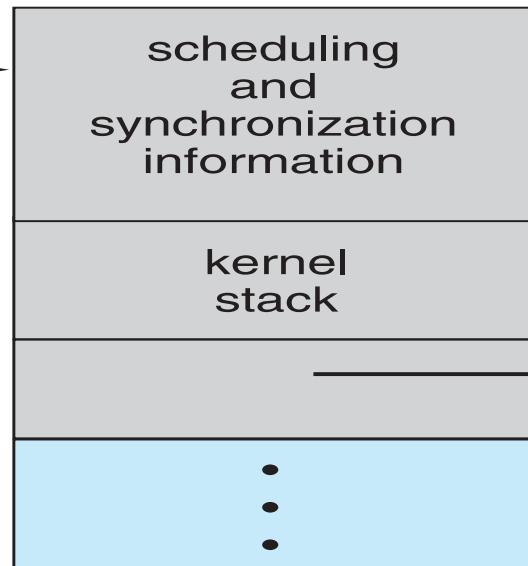


Windows Threads Data Structures

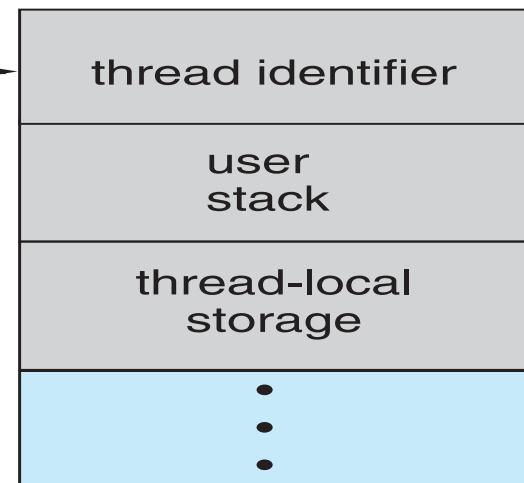
ETHREAD



KTHREAD

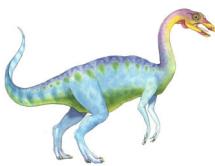


TEB



kernel space

user space



Operating System Examples

Linux Threads

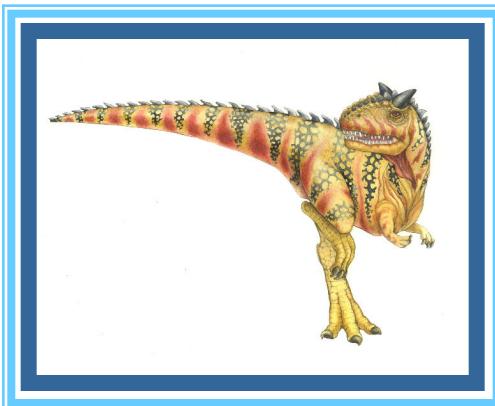
- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through `clone()` system call
- `clone()` allows a child task to share the address space of the parent task (process)
 - Flags control behavior: determine what and how much to share

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





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

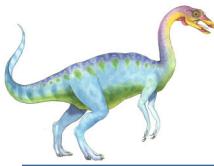




Thread Cancellation

- 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);
```



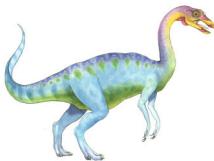
Thread Cancellation (Cont.)

- 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

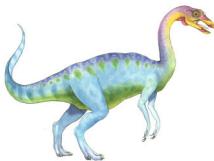




Thread-Local Storage

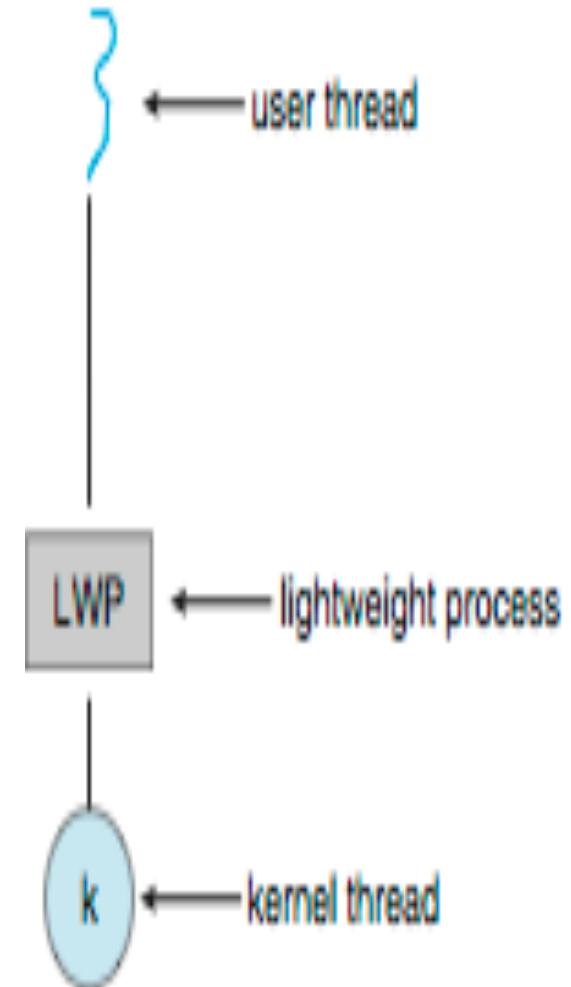
- Thread-local storage (TLS) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
 - Local variables visible only during single function invocation
 - TLS visible across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



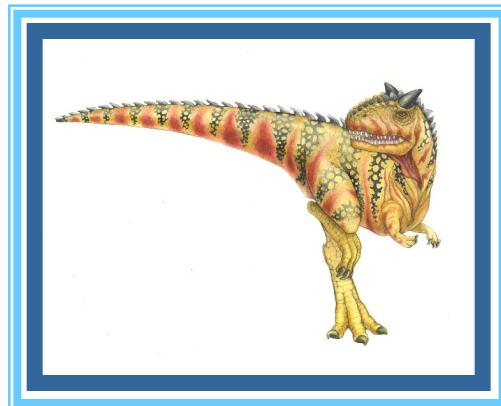


Scheduler Activations

- 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



Chapter 5: Process Synchronization





Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches



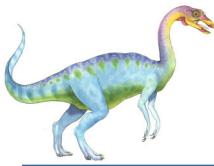


Objectives

- Cooperating processes affect [or, is affected by] other processes
 - ▶ Sharing data or code
 - ▶ How to maintain and ensure data consistency...?

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





Background

- Processes can execute concurrently
 - ▶ In Chap-3: Process scheduler switches among processes; **concurrency**
 - ▶ In Chap-4: Multiprogramming distributes tasks among cores; **parallelism**
 - May be interrupted at any time, partially completing execution
 - ▶ How to preserve the **integrity** of data shared by several processes...?
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- **Illustration of the problem:** Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffer; **the original solution in Chap-3 allowed at most `BUFFER_SIZE - 1` items in the buffer at the same time.** We can do so by having an integer **counter** that keeps track of the number of items in the buffers. Initially, **counter** is set to 0. It is incremented by the producer after it produces a new item and is decremented by the consumer after it consumes an item.





Chap-3: Producer [without a counter]

```
item next_produced;           stores new item to be produced  
...  
while (true)  
{  
    /* produce an item in next_produced */  
  
    while (((in + 1) % BUFFER_SIZE) == out)  
        ; /* do nothing      when the buffer is full*/  
  
    buffer[in] = next_produced;      item is produced  
  
    in = (in + 1) % BUFFER_SIZE;    update in pointer  
}
```





Chap-3: Consumer [without a counter]

```
item next_consumed;           stores item to be consumed  
...  
while (true)  
{  
    while (in == out)  
        ; /* do nothing when the buffer is empty */  
  
    next_consumed = buffer[out];   item is consumed  
  
    out = (out + 1) % BUFFER_SIZE; update out pointer  
  
    /* consume the item in next_consumed */  
}
```





Producer [with a counter]

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

    while (counter == BUFFER_SIZE) ;
          ; /* do nothing when buffer full */

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

counter++;
}
```





Consumer [with a counter]

```
while (true)
{
    while (counter == 0) ;
        ; /* do nothing when buffer empty */

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

    counter--;
    /* consume the item in next_consumed */
}
```





Race Condition

- Producer and consumer may function incorrectly when executed concurrently
 - » `counter` is a shared variable but accessed concurrently
- `counter++` could be implemented as

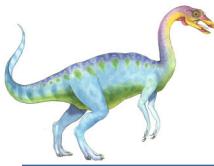
```
register1 = counter
register1 = register1 + 1
counter = register1
```
- `counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```
- Consider this execution interleaving with “`counter = 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** = concurrent access to variable + result depends on order
 - **Solution:** synchronization; only one process at a time accesses data





Critical Section Problem

- Consider system of n processes $\{P_0, P_1, \dots P_{n-1}\}$
- Each process P_i has a segment of code called a **critical section**, in which
 - It may be changing common variables, updating table, writing file, ... etc
 - When P_i is in its critical section, no process P_j should be in its critical section
 - ▶ **No two processes are executing in their critical sections at the same time**
- **Critical section problem** is to design protocol to solve this
 - That is: protocol that processes can use to cooperate
- Each process must ask permission to enter its critical section via the **entry section**, leave the critical section through the **exit section**, then resumes its operations in the **remainder section**





Critical Section

- General structure of a process P_i

do {

entry section

critical section

exit section

remainder section

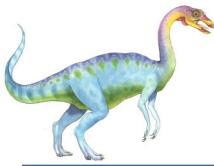
} while (true);



Solution to Critical-Section Problem

A solution to the critical section problem must satisfy the following 3 requirements:

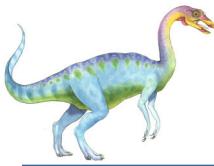
1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely
3. **Bounded Waiting** - A limit 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
 - We assume that each process executes at a nonzero speed
 - We make no assumption concerning **relative speed** of the n processes
 - Imagine case where many active kernel processes accessing a list of all open files in the system. This data structure is prone to possible **race conditions**.



Critical-Section Handling in OS

- Two approaches depending on if kernel is preemptive or non-preemptive
 - **Preemptive** – allows preemption of process when running in kernel mode
 - ▶ **Preemption** = The ability of the OS to interrupt a currently scheduled task in favor of a higher priority task. It is normally carried out by a **privileged task** or part of the system known as a **preemptive process scheduler**, which has the power to **preempt**, or interrupt, and later resume, other tasks in the system. This only applies to processes running in kernel mode.
 - More responsive to users, but
 - » Shared kernel data **may not** be free from race conditions
 - Preemptive kernel must be carefully designed
 - **Non-preemptive** – process runs until it exits kernel mode, blocks, or *voluntarily* yields CPU
 - ▶ Essentially free of race conditions in kernel mode **as only one process is active in the kernel at a time**





Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
 - Software-based solutions not guaranteed to be correct on modern comp arch
 - ▶ Example: **Peterson's solution** to the CS problem (see Page-207)
- Solutions **in the following slides** are based on the idea of **locking**
 - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
 - ▶ Solve CS problem by **preventing interrupts** while modifying shared data
 - Currently running code would execute without preemption
 - Generally too inefficient on multiprocessor systems
 - ▶ Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
 - ▶ **Atomic** = non-interruptible
 - Either test memory word and set value
 - Or swap contents of two memory words





Solution to Critical-Section Problem Using Locks on Single-CPU Systems

```
do
```

```
{
```

```
    acquire lock
```

```
    critical section
```

```
    release lock
```

```
    remainder section
```

```
}
```

```
while (TRUE) ;
```





Test_and_Set Instruction

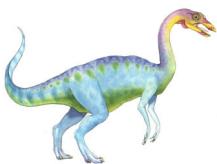
■ Definition:

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

1. Executed atomically; **a *non-interruptible unit* of execution**
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

- Thus, we can implement mutual exclusion on multi-CPU systems
 - See next slide





Solution using Test_and_Set() Instruction on Multi-CPU Systems

- Shared Boolean variable **lock**, declared and initialized to FALSE
- Solution, for a process P_i :

```
do
{
    while (test_and_set(&lock))      /*acquire lock*/
        ; /* do nothing */

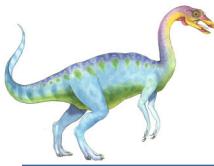
        /* critical section */

    lock = false;                      /*release lock*/

    /* remainder section */

}
while (true);
```





Compare_and_Swap Instruction

■ Definition:

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

    if (*value == expected)
        *value = new_value;

    return temp;
}
```

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

- Thus, we can implement mutual exclusion on multi-CPU systems
 - See next slide





Solution using Compare_and_Swap() Instruction on Multi-CPU Systems

- Shared integer variable **lock**, declared and initialized to 0;
- Solution; for a process P_i :

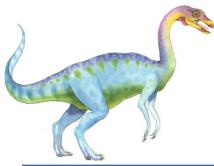
```
do
{
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

        /* critical section */

    lock = 0;
    /* remainder section */

}
while (true);
```





Bounded-Waiting Mutual Exclusion with Test_and_Set

```
Do /* Shared Boolean waiting[n] and lock initialized to false */

{
    waiting[i] = true;
    key = true;
    while (waiting[i] && key) { key = test_and_set(&lock); }
    waiting[i] = false;

    /* critical section; enters only if waiting[i] or key == false */

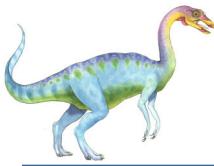
    j = (i + 1) % n;
    while ((j != i) && !waiting[j]) { j = (j + 1) % n; }

    if (j == i) { lock = false; }
    else { waiting[j] = false; }

    /* remainder section */
}

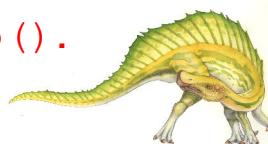
while (true);
```





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; short for mutual exclusion
- Protect a critical section by first **acquire()** a lock then **release()** the lock
 - Boolean variable indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
 - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
 - Process that tries to enter its CS must loop in the call to **acquire()**
 - Same thing is true with `test_and_set()` and `compare_and_swap()`.
 - This lock therefore is called a **spinlock**





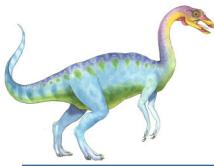
Acquire() and Release()

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

release() {
 available = true;
}

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





# Semaphores

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

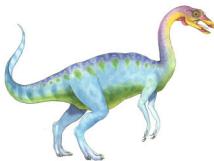
## ■ Definition of the **wait()** operation

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

## ■ Definition of the **signal()** operation

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





# Semaphore Usage

- **Counting semaphore** – integer value  $S$  can range over an unrestricted domain
  - Can be used to control access to resources ( $S$  initialized to number of resources)
    - ▶ Process that wishes to use a resource will execute `wait (S)`
    - ▶ Process that releases a resource will execute `signal (S)`
    - ▶ If  $S = 0$  then all resources are being used, and, Processes block until  $S > 0$ ;
- **Binary semaphore** – integer value  $S$  can range only between 0 and 1
  - Behaves similarly to **mutex lock**
- Can solve various synchronization problems
- Consider concurrent  $P_1$ , and  $P_2$  that require statement  $S_1$  to happen before  $S_2$   
Create a common semaphore “`synch`” shared by  $P_1$  and  $P_2$  and initialized to 0
  - $P_1 :$ 

```
s1;
signal(synch);
```
  - $P_2 :$ 

```
wait(synch);
s2; since synch is initialized to 0, P2 executes S2 only after P1 invokes signal(synch)
```
- Can implement a counting semaphore  $S$  as a binary semaphore





# Semaphore Implementation with no Busy Waiting

- With each semaphore  $S$  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 it on the appropriate  $S$  waiting queue
  - **wakeup** – transfer one of process from the  $S$  waiting queue to ready queue
- ```
typedef struct{  
    int value;  
    struct process *list;  list of processes waiting on the semaphore S  
} semaphore;
```





Semaphore Implementation with no Busy Waiting

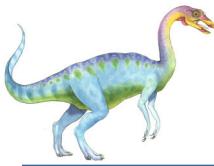
```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this process to S->list;  
        block();  
    }  
}
```

} **wait()** operation adds a process to the *S* queue and suspends it

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

} **signal()** operation removes one process from *S* queue and awakens it

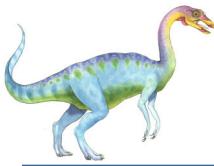




Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` operations on the same semaphore S at the same time
 - Hence, if P executes `wait()` or `signal()` then no other processes should
 - ▶ This is a CS-problem
 - Also, recall that `wait()` and `signal()` operations must be atomic
- Thus, in semaphore implementation the `wait()` and `signal()` codes are placed in **the critical section of a process P**
 - We have moved `wait()` from entry section and `signal()` from exit section of P
 - Could now have **busy waiting** in critical section implementation
 - ▶ But implementation code is short
 - ▶ Little busy waiting *if* critical sections of `wait()` and `signal()` rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution;





Deadlock and Starvation

- **Deadlock** – a process is waiting indefinitely for an event that can be caused only by another waiting process
- Let S and Q be two semaphores initialized to 1

P_0

```
wait(S) ;  
wait(Q) ;  
...  
signal(S) ;  
signal(Q) ;
```

P_1

```
wait(Q) ;  
wait(S) ;  
...      P and Q are deadlocked here  
signal(Q) ;  
signal(S) ;
```

■ 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





Classic Problems of Synchronization

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





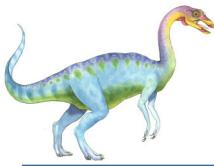
Bounded-Buffer Problem

- Producer and consumer processes share the following data

```
int n;  
semaphore mutex = 1;  
semaphore empty = n;  
semaphore full = 0;
```

- **n** buffers, each can hold one item
- Semaphore **mutex** initialized to the value 1; for permission access to buffer pool
- Semaphore **full** initialized to the value 0; number of full buffers
- Semaphore **empty** initialized to the value n; number of empty buffers



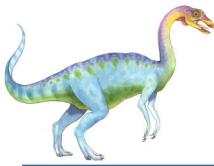


Bounded Buffer Problem

- The structure of the producer process; solution using semaphores

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





Bounded Buffer Problem

- The structure of the consumer process; **solution using semaphores**

Do

{

wait(full);

wait(mutex);

 ...

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

 ...

signal(mutex);

signal(empty);

 ...

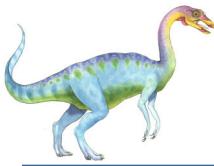
 /* consume the item in **next_consumed** */

 ...

}

while (true);





Readers-Writers Problem

- A data set is shared among a number of concurrent processes
 - Readers – only read the data set; they do **not** perform any updates
 - ▶ No problem if some *readers* access data simultaneously; no waiting needed
 - Writers – can both read and write;
 - ▶ Problem if some *writer* accesses data; thus, **exclusive** access only for writing
- Problem – allow *multiple writers* to access 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
- One solution with semaphores: the reader processes share the following data
 - semaphore **rw_mutex** = 1; common to both readers and writers
 - semaphore **mutex** = 1; mutual exclusion if **read_count** updated
 - int **read_count** = 0; # of processes currently reading





Readers-Writers Problem

- The structure of a writer process

```
do
{
    wait(rw_mutex);    mutual exclusion for writers
    ...
/* writing is performed */

    ...
signal(rw_mutex);

}
while (true);
```





Readers-Writers Problem

The structure of a reader process

```
do
{
    wait(mutex);
    read_count++; critical section when read_count being modified
    if (read_count == 1)          /* if first reader */
        wait(rw_mutex); in case a writer is writing
    signal(mutex);

    ...
/* reading is performed */

    ...
wait(mutex);
read count--; critical section when read_count being modified
if (read_count == 0)          /* if last reader */
    signal(rw_mutex); release lock
    signal(mutex); release lock
}
while (true);
```





Readers-Writers Problem Variations

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





Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors **when in thinking state**, occasionally try to pick up 2 chopsticks (but can pick only one at a time) to eat from bowl
 - Need **both** to eat, then release **both** when done **and then start thinking again**
- In the case of 5 philosophers (**solution with semaphores**)
 - Shared data
 - ▶ Bowl of rice (data set)
 - ▶ Semaphore **chopstick[5]** initialized to 1's; **each chopstick is a semaphore**





Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*: (no two neighbors are eating simultaneously)

```
do
{
    wait(chopstick[i]);
    wait(chopstick[(i + 1) % 5]);

        // eat

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

        // think
}
```

```
while (TRUE);
```

- What is the problem with this algorithm?

- Deadlock if all 5 philosophers are hungry at the same time and pick a stick





Dining-Philosophers Problem Algorithm (Cont.)

■ Deadlock handling

- Allow at most 4 philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up both chopsticks only if both are available (picking must be done in a critical section).
- Use an asymmetric solution -- An odd-numbered philosopher picks up first her left chopstick and then her right chopstick. Even-numbered philosopher picks up first her right chopstick and then her left chopstick.





Problems with Semaphores

- Incorrect use of semaphore operations:
 - signal (mutex) ... wait (mutex)
 - wait (mutex) ... wait (mutex)
 - Omitting of wait(mutex) or signal(mutex) or both
- Deadlock and starvation are possible.





Synchronization Examples

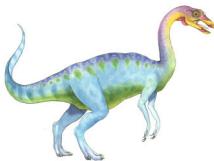
- Solaris
- Windows
- Linux
- Pthreads





Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency to protect only data accessed by short code segments
 - Starts as a standard semaphore implemented as a spin-lock; a **normal mutex**
 - If lock held by a thread currently running on another CPU, **then the thread spins**
 - If lock held by a non-running-state thread, then block and sleep waiting for signal of lock being released
- Uses **condition variables** (**skipped**; see **Monitors** in Section-5.8)
- Uses **readers-writers** locks when longer sections of code need access to data
 - To protect **frequently accessed read-only** data
- Uses **turnstile**s to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock; **queues containing threads blocked on a lock**
 - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile



Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
 - Spinlocking-thread will never be preempted
- Provides **dispatcher objects** in user-mode which may act as mutexes, semaphores, events, and timers
 - **Events**
 - ▶ An event acts much like a **condition variable** (see **Monitors** in Section-5.8)
 - ▶ **Notify a waiting thread when a desired condition occurs**
 - Timers notify one or more thread when time expired
 - ▶ **Notify threads that a specified amount of time has expired**
 - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)





Linux Synchronization

■ Linux:

- Prior to kernel Version 2.6, disables interrupts to implement short critical sections
- Version 2.6 and later, fully preemptive

■ Linux provides:

- Semaphores
- atomic integer operations
- spinlocks
- reader-writer versions of both

■ On single-CPU system, spinlocks replaced by enabling and disabling kernel preemption





Pthreads Synchronization

- Pthreads API is OS-independent

- It provides:

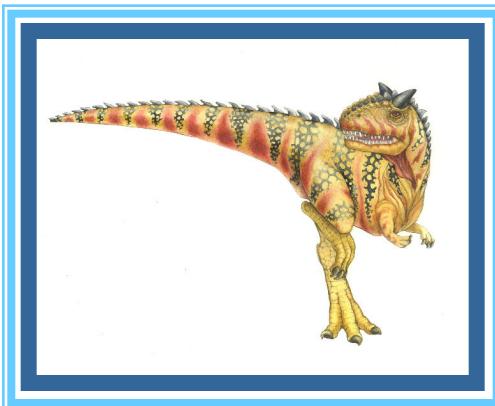
- mutex locks
- condition variable

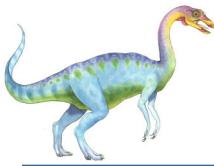
- Non-portable extensions include:

- read-write locks
- spinlocks



End of Chapter 5





Algorithm for Process P_i

do

{

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

critical section

```
turn = j;
```

remainder section

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





Peterson's Solution

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





Algorithm for Process P_i

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





Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:

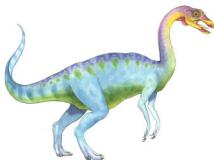
1. Mutual exclusion is preserved

P_i enters CS only if:

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

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





Monitors

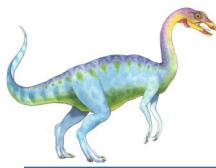
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

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

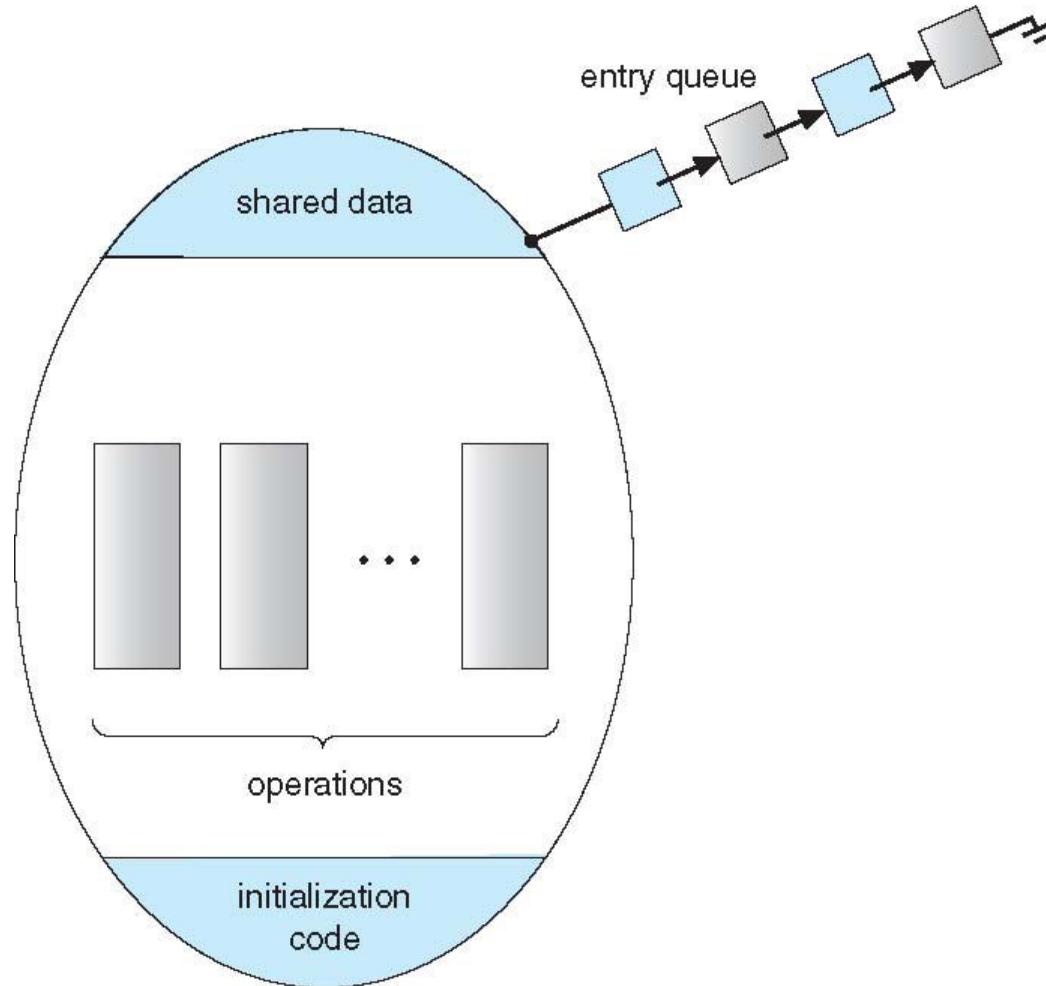
    procedure Pn (...) { ..... }

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





Schematic view of a Monitor





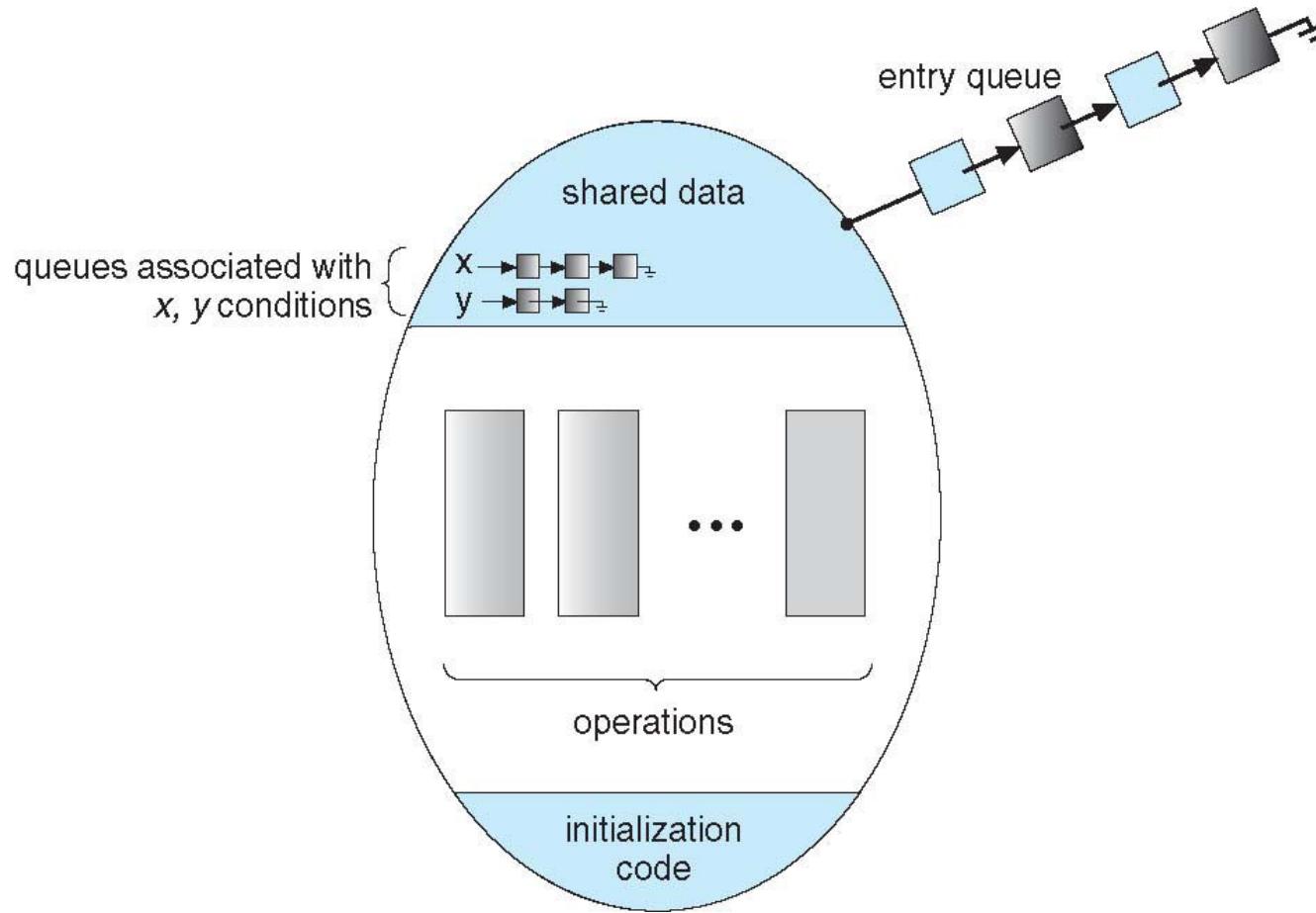
Condition Variables

- **condition x, y;**
- Two operations are allowed on a condition variable:
 - **x.wait()** – a process that invokes the operation is suspended until **x.signal()**
 - **x.signal()** – resumes one of processes (if any) that invoked **x.wait()**
 - ▶ If no **x.wait()** on the variable, then it has no effect on the variable





Monitor with Condition Variables





Condition Variables Choices

- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
 - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
 - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
 - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
 - Both have pros and cons – language implementer can decide
 - Monitors implemented in Concurrent Pascal compromise
 - ▶ P executing signal immediately leaves the monitor, Q is resumed
 - Implemented in other languages including Mesa, C#, Java





Monitor 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





Monitor Implementation Using Semaphores

- Variables

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

- Each procedure *F* will be replaced by

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

- Mutual exclusion within a monitor is ensured





Monitor Implementation – Condition Variables

- For each condition variable x , we have:

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

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

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





Monitor Implementation (Cont.)

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

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





Resuming Processes within a Monitor

- If several processes queued on condition x , and $x.signal()$ executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form $x.wait(c)$
 - Where c is **priority number**
 - Process with lowest number (highest priority) is scheduled next





Single Resource allocation

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

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

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





A Monitor to Allocate Single Resource

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





Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages





Transactional Memory

- A **memory transaction** is a sequence of read-write operations to memory that are performed atomically.

```
void update()  
{  
    /* read/write memory */  
}
```





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.



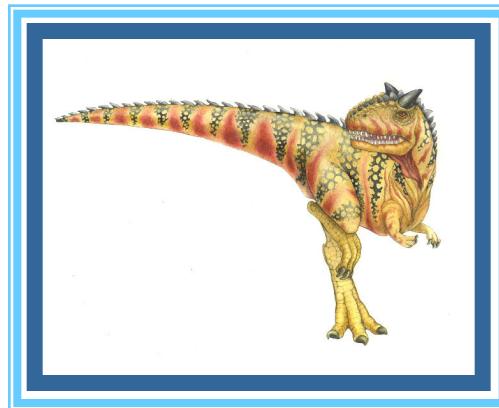


Functional Programming Languages

- 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.
- There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races.



Chapter 8: Main Memory





Chapter 8: Memory Management

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

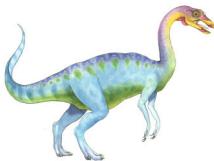




Objectives

- To increase CPU utilization and response speed:
 - Several processes must be kept in main memory
 - ▶ We must share memory
 - How to manage main memory resource...?
- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging





Background

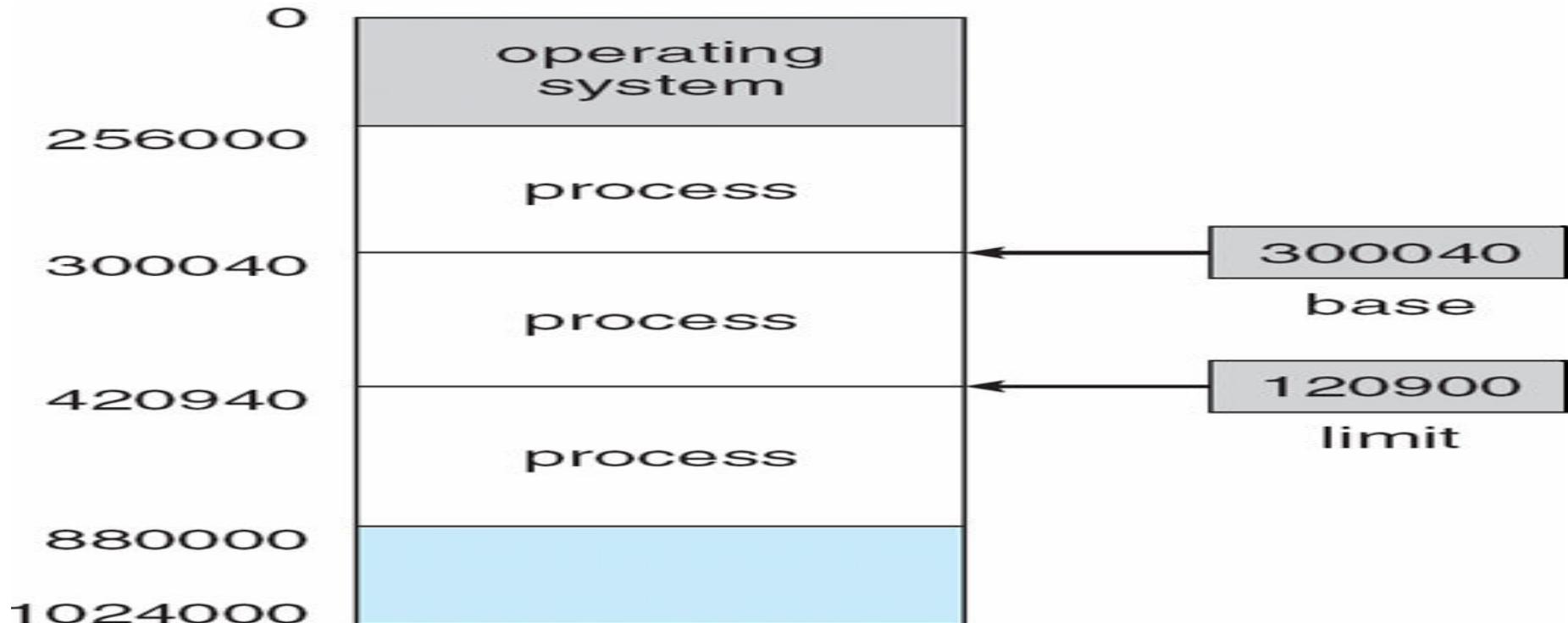
- Program must be brought (from disk) into memory and placed within a process for it to be run
 - Given current value of EIP register, then Fetch-and-Execute cycle
 - ▶ Fetch next instruction from memory
 - ▶ Decode the new instruction
 - If necessary: fetch operands (data) from memory
 - ▶ Execute the instruction
 - EIP = address of next instruction
 - ▶ If necessary: save results (new data) to memory
- Main memory and registers are the only storage that the CPU can access directly
 - There are ***no*** machine instructions taking ***disk addresses*** as arguments
- Memory unit only sees a stream of
 - **addresses and read requests**, or, **address + data and write requests**
- Registers are accessed within one tick of the CPU clock; **very fast memories**
- Main memory access can take many CPU clock ticks, causing a **stall**
- **Cache** sits between main memory and CPU registers; ***solution to stall issue***
- Protection of memory required to ensure correct operation; ***hardware-level***





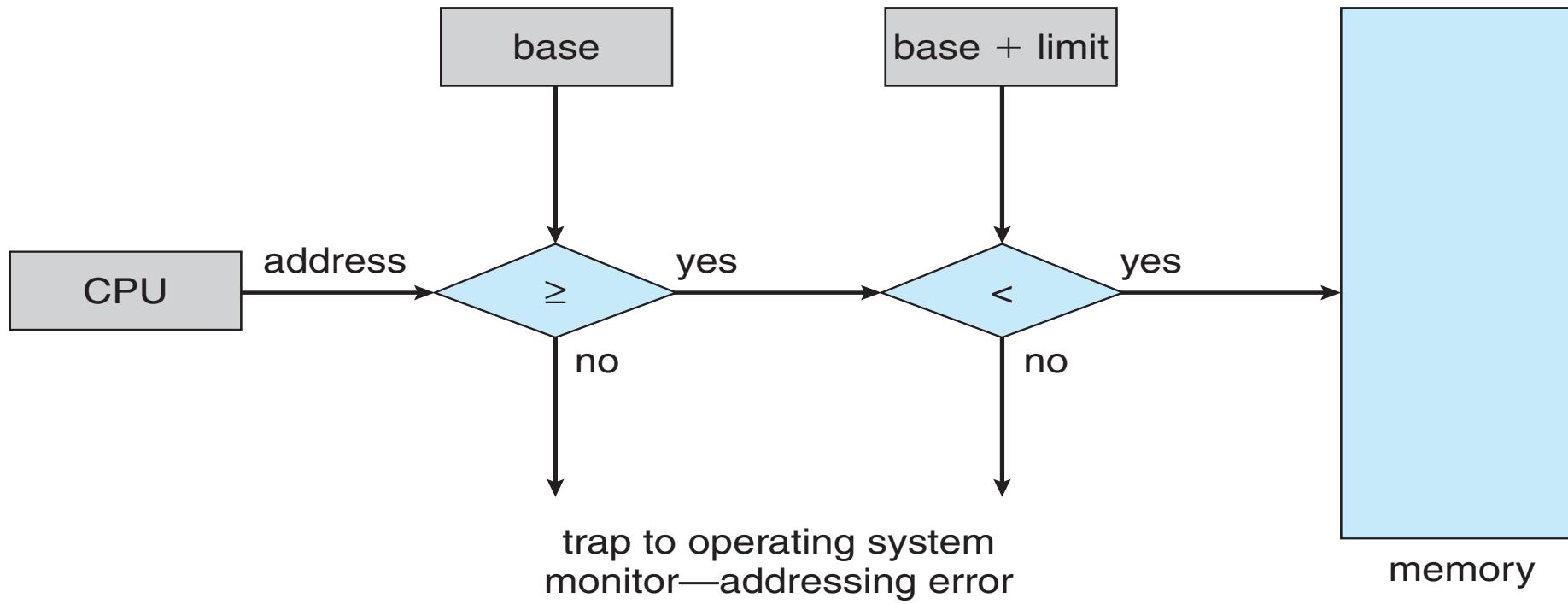
Base and Limit Registers

- For each process, the **base** and **limit registers** define its logical address space
 - ▶ To protect processes from each; each process has its own memory space
 - **Base register** holds the smallest legal physical memory address
 - **Limit register** specifies the size of the range of accessible addresses
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user; **to protect a process's memory space**



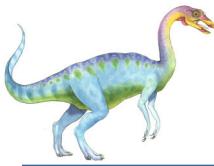


Hardware Address Protection



- Base and limit registers loaded only by the OS through a privileged instruction
 - To prevent users from changing these registers' contents
- OS has unrestricted access to OS memory and user memory
 - Load users' programs into users' memory, ... etc
 - Access and modify system-calls' parameter, ... etc





Address Binding

- Processes on disk waiting to be brought into memory for execution form an **input queue**
- Addresses represented in different ways at different stages of a program's life.
 - Addresses in source codes are symbolic. They are *names* of variable
 - Compilers **bind** symbolic addresses to relocatable addresses
 - ▶ i.e. “14 bytes from beginning of this module”
 - Linkers or loaders will bind relocatable addresses to absolute addresses
 - ▶ i.e. 74014
 - Each binding is a mapping from one address space to another



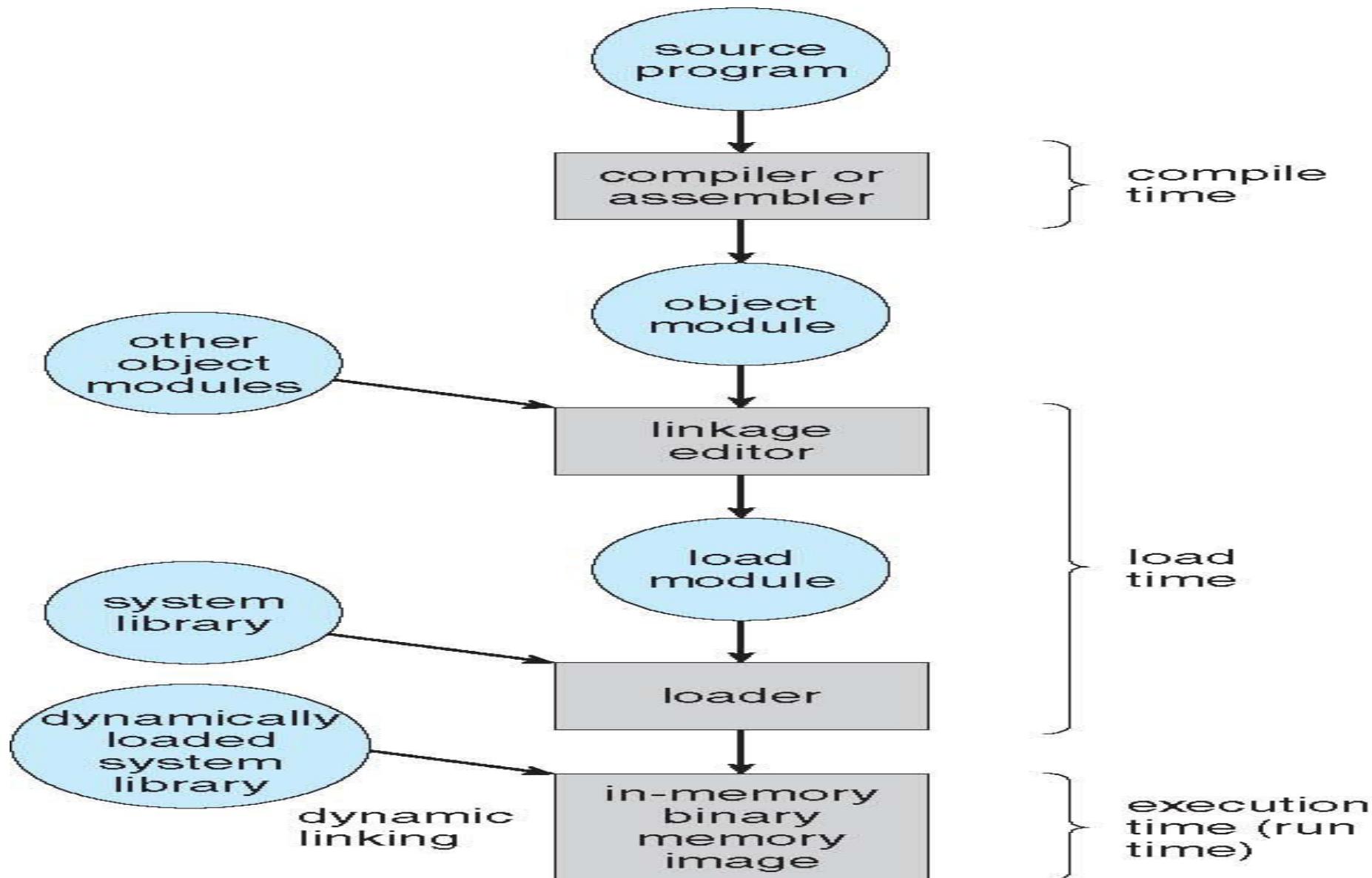


Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
 - **Compile time:** If memory location is known a priori at compile time, **absolute code** can be generated; must recompile code if starting location changes
 - ▶ If a process will reside at location L then its compiler code will start at L
 - ▶ Source code should be re-compiled if L changes
 - **Load time:** Compiler must generate **relocatable code** if memory location of the process is not known at compile time
 - ▶ That is, final binding is delayed until load 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-management unit** in its ***memory-address register***
- The compile-time and load-time address-binding schemes generate identical logical and physical addresses;
 - The execution-time address-binding scheme results in differing logical and physical addresses; hence, we refer to ***logical address*** as ***virtual address***
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses **corresponding to these logical addresses**





Memory-Management Unit (MMU)

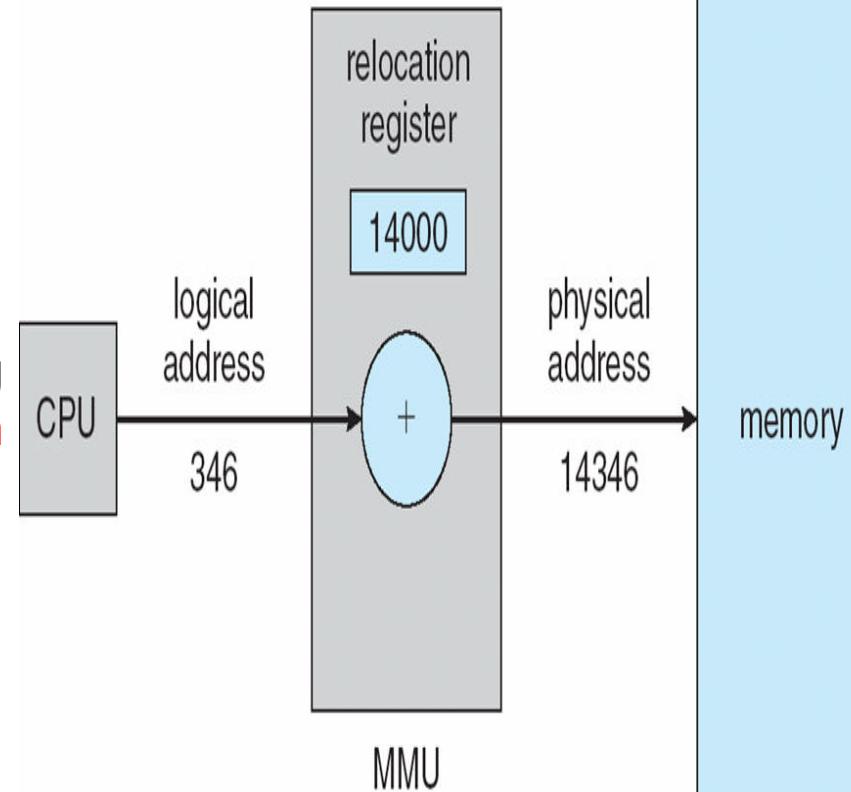
- **MMU** is the hardware that does *virtual-to-physical addr mappings* at run-time
- Many possible mapping methods; covered in the rest of this chapter
- **Example:** a simple MMU scheme which adds the value in the base register to every address generated by a user process at the time it is sent to memory
 - ▶ Generalizes the *base-register scheme* described in Slide-6
 - Base register now called **relocation register**
 - MS-DOS on Intel 80x86 used 4 relocation registers: **EBX, BX, BH, BL**
- 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
 - ▶ See Slide-8
 - Logical address **mapped** to physical addresses **before use**
 - ▶ Logical addresses: in range 0 to max
 - ▶ Physical addresses: in range $R + 0$ to $R + max$
 - For a base value R in the relocation register

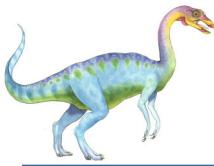




Dynamic relocation using a relocation register (Dynamic Loading)

- A **user** routine is not loaded until it is called. **Loading is delayed until run-time**
- 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. **Loaded program portion is much smaller than total program**
- No special support from the operating system is required
 - Implemented through program design
 - OS can help **users** by providing libraries to implement dynamic loading

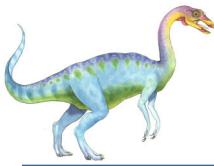




Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image (**the final *executable* file**)
- Dynamic linking – the linking is postponed until execution-time
 - *The executing program* (the running process)
- Small piece of code, the **stub**, indicates how to locate the appropriate memory-resident library routine, or, how to load the library if routine is not already present
- Operating system checks if the needed routine is in the process's memory space
 - If not in address space, **then load the library routine; when stub is executed**
- Stub replaces itself with the address of the routine and executes the routine
- Dynamic linking is particularly useful for [**large**] system libraries
 - Very useful also for **shared libraries** with different version numbers
- Consider applicability to patching system libraries
 - Versioning may be needed





Swapping

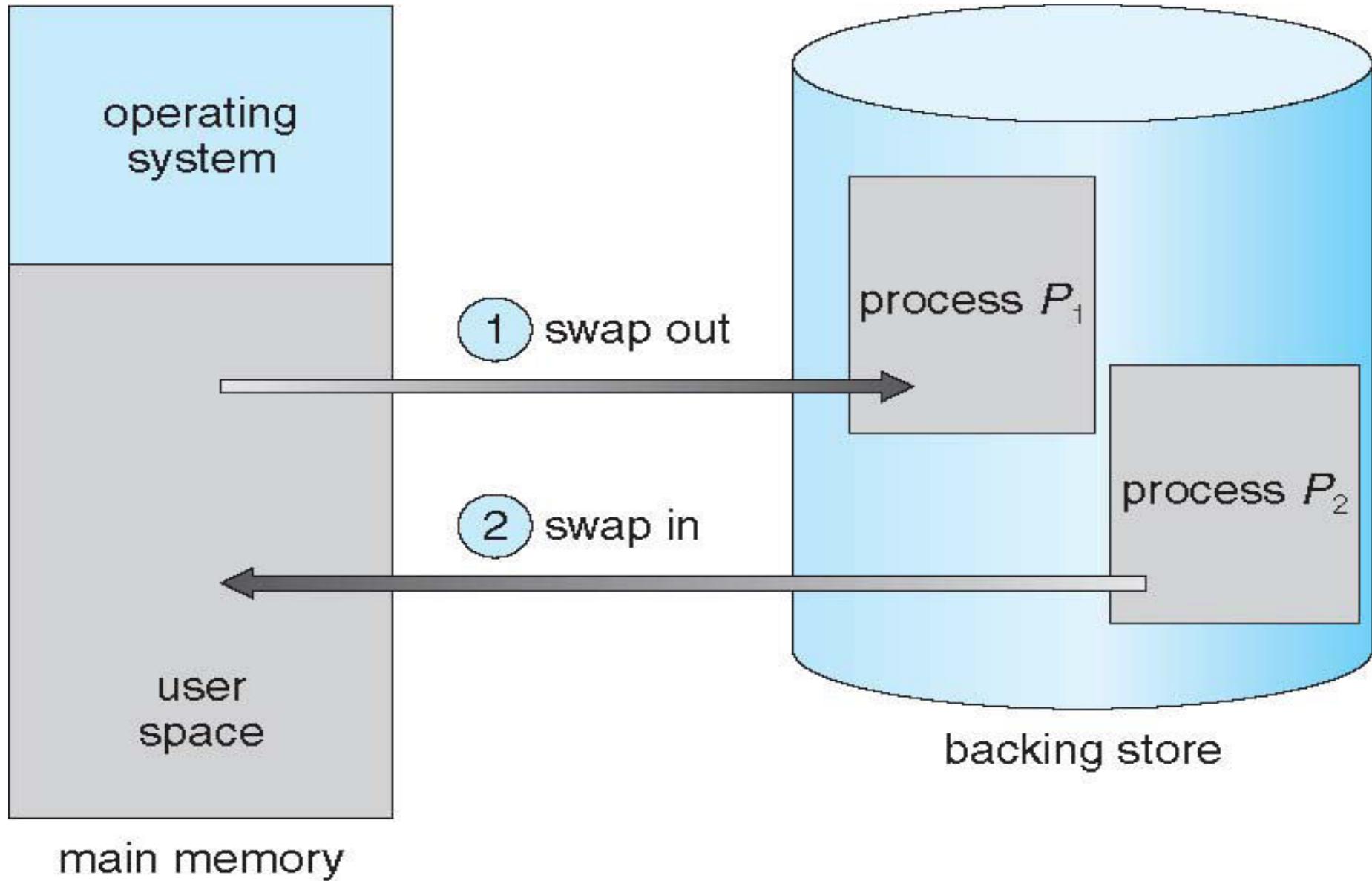
- A process can be **swapped** temporarily out of memory to a **backing store**, and then brought back into memory for continued execution
 - Total physical memory space of processes can exceed physical memory
 - ▶ Hence, swapping increases the degree of multiprogramming
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- System maintains a **ready queue** of ready-to-run processes which have memory images on the backing store or in memory. **Standard swapping method:**
 - Dispatcher called when the CPU scheduler selects a process P from queue
 - ▶ If P is not in the ready queue and not enough free space in memory for P
 - **Swap in P** from backing store and **swap out some Q** from memory
 - Reload registers and transfer control to P ; i.e. P is now in running state
- **The context-switch time in standard swapping is fairly high; see Slide-16**
- **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
 - Also, see **medium-term scheduler** in Chap-3





Schematic View of Swapping

(Swapping of two processes using a disk as a backing store)

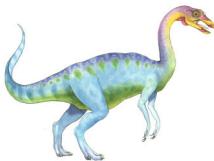




Context Switch Time including Swapping

- Context-switch time in swapping is very high
 - Assume swapping 100MB to backing store with a **transfer rate** of 50MB/sec
 - ▶ **Swap out** time of a process *Q* is 2000 ms ($= 100\text{MB} / 50\text{MB/sec} = 2 \text{ sec}$)
 - ▶ **Swap in** time of a same sized process *P* is also 2000 ms
 - ▶ **Total context-switch swapping component time** is **4000ms (= 4 sec)**
 - We have ignored other components of context-switch time and ignored other disk performance aspect. **But, the major part of swap time is transfer time**
- Total transfer time is directly proportional to the amount of memory swapped
 - We can reduce if we know exactly how much memory a process *is* using
 - ▶ Process with dynamic memory requirements will issue system-calls to inform OS of memory use via `request_memory()` and `release_memory()`





Context Switch Time and Swapping

■ Other constraints as well on swapping

- Does the swapped out process Q need to swap back in to same physical addresses?
 - ▶ Depends on address binding method
- Q must be **completely idle** before being swapped out; not just waiting
- Consider pending I/O to/from Q 's memory space
 - ▶ We can't swap out the process since I/O would occur to wrong process
 - ▶ **Solutions:**
 - Never swap out a process with pending I/O
 - Or, always transfer I/O operations to kernel space, then to I/O device
 - » Known as **double buffering**, adds overhead





Modified Standard Swapping

- Standard swapping not used in modern operating systems
 - But modified versions are common
 - ▶ Idea: swap only when free memory is extremely low

- Modified versions of swapping are found on many systems
 - UNIX, Linux, and Windows
 - ▶ Swapping normally disabled
 - ▶ Started only if amount of free memory falls below a threshold amount
 - ▶ Disabled again once amount of free memory increases beyond some threshold





Swapping on Mobile Systems

- Not typically supported **in any form**, because:
 - Mobile devices use **flash memory** as persistent storage
 - ▶ Small amount of space
 - ▶ Limited number of write cycles
 - ▶ Poor throughput between flash memory and CPU on mobile platforms
 - Number of I/O operations that can be completed per time units
- Instead, mobile devices use other methods to free up memory if low space
 - iOS **asks** apps to voluntarily relinquish allocated memory
 - ▶ Read-only data (e.g., **codes**) removed then reloaded from flash if needed
 - ▶ iOS may terminate any apps that fail to free up memory
 - Android adopts a similar strategy as iOS. But, for fast restart
 - ▶ It first saves **application state** to flash before terminating an app
 - Both OS's support paging as discussed in **Slide-27**

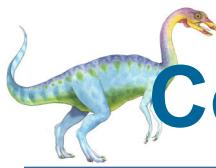




Contiguous Allocation

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





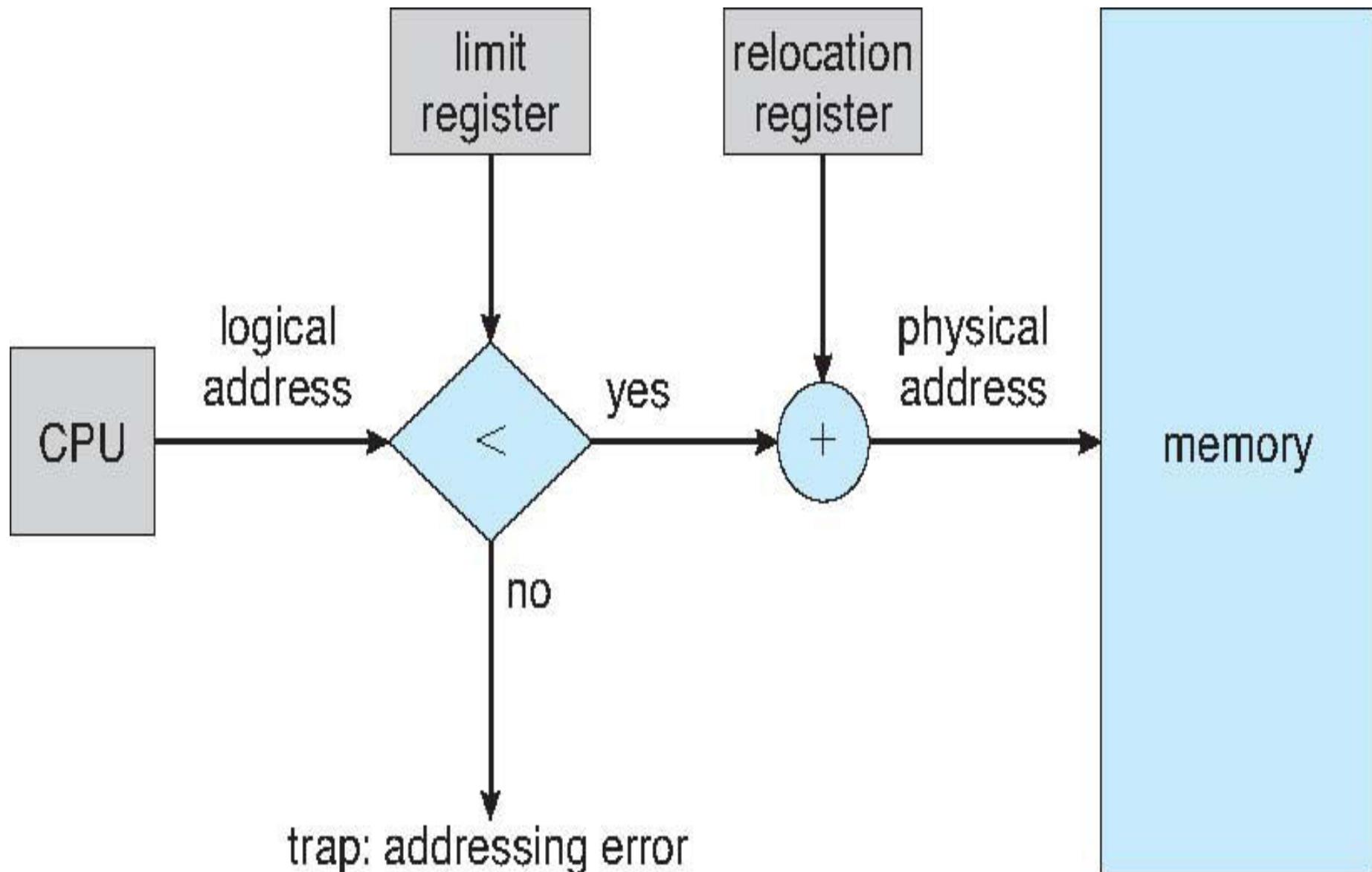
Contiguous Allocation - Memory Protection

- We can protect a process's memory by combining ideas in Slide-5 and Slide-11
- Relocation registers is used to protect user processes from each other, and from changing operating-system code and data.
 - For each process:
 - ▶ Relocation register contains value of smallest physical address; **Slide-11**
 - ▶ Limit register contains its range of logical addresses
 - Each logical address must be less than the limit register value; **Slide-5**
 - ▶ MMU maps logical addresses *dynamically*
 - By adding logical address to relocation value; **see Slide-22**
 - Dispatcher always loads these two registers with their correct values
 - ▶ All logical addresses are checked against these two registers
 - ▶ Thus, protecting both each OS and each user program and data
 - This relocation-register protection scheme allows actions such as kernel code being **transient** and kernel changing size ***dynamically***
 - ▶ Unused kernel code and data need not be in memory
 - Example: un-used device driver code need not be kept in memory





Hardware Support for Relocation and Limit Registers

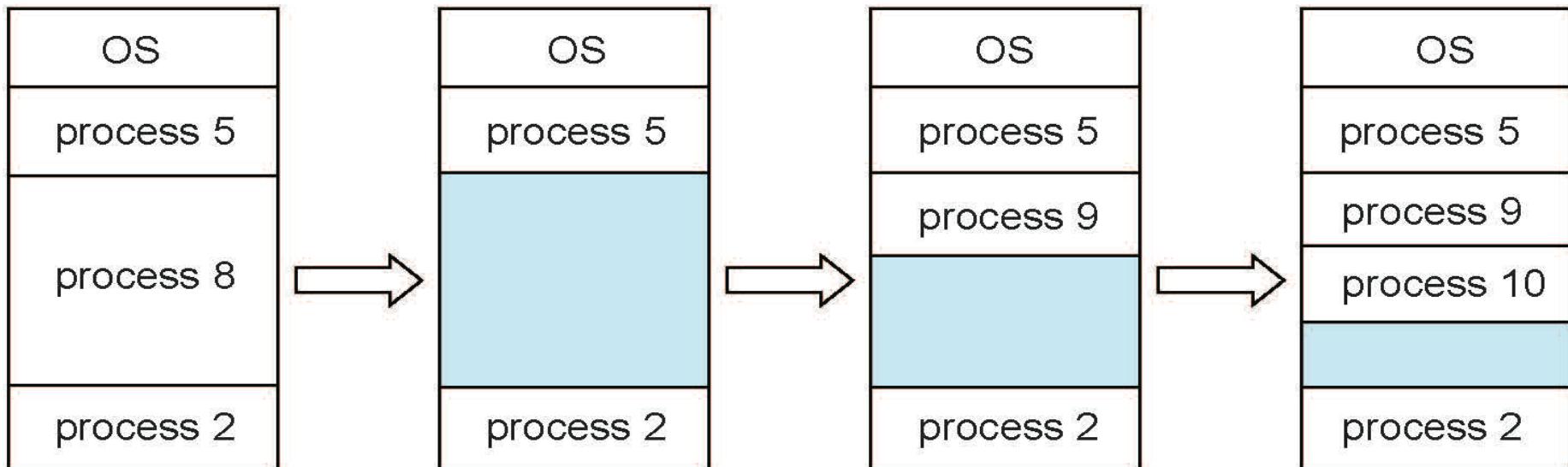




Multiple-Partition Allocation

■ Multiple-partition allocation: divide memory into many *fixed-sized partitions*

- One process in each partition. Degree of multiprogramming limited by # of partitions
 - ▶ Fixed-sized partitioning is simple but limited due to size and number of partitions
- Better: use **variable-partition** sizes for efficiency (sized to a given process' needs)
 - ▶ **Hole** – block of free memory; **variable-sized holes** scattered throughout memory
- Operating system maintains information about: a) allocated partitions, and b) holes
- **Job scheduling algorithm** considers holes and memory need of each process
 - ▶ Allocate memory to a new process from a hole large enough to accommodate it
 - ▶ Terminating processes free their partition; adjacent free partitions are combined





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 search entire list, unless ordered by size
 - ▶ Produces the largest leftover hole
 - Which may be more useful than the smaller leftover hole from a best-fit approach; see Slide-25
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization
- First-fit is generally the fastest

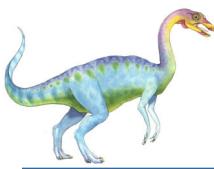




Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
 - Free memory space is broken into pieces as processes are loaded and removed from memory
 - Both, first-fit and best-fit strategies suffer from external fragmentation
 - Statistical analysis of first-fit reveals that given N allocated blocks, $0.5N$ blocks will be lost to fragmentation
 - ▶ That is, 1/3 of memory may be unusable -> **50-percent rule**
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
 - Example: a process requests 18,462 bytes and is allocated memory hole of 18,464 bytes; ***we are then left with a internal hole of 2 bytes***

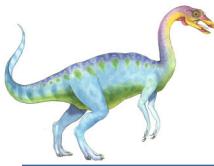




Fragmentation

- Reduce external fragmentation by **compaction**
 - Shuffle memory contents to place all free memory together in one large block
 - Compaction is possible *only* if address relocation is dynamic, and is done at execution time
 - I/O problem
 - ▶ Latch job in memory while it is involved in I/O
 - ▶ Do I/O only into OS buffers
 - Compaction can be very expensive in terms of time
 - ▶ All processes are moved toward one end of memory
- Other solutions to external fragmentation problems: **Segmentation and Paging**
 - Permit the logical address space of each process to be non-contiguous
 - ▶ Process can be allocated physical memory **wherever** it is available
- Now consider that backing store has same fragmentation problems
 - As processes are swapped in and out. Also true for any storage device





Paging

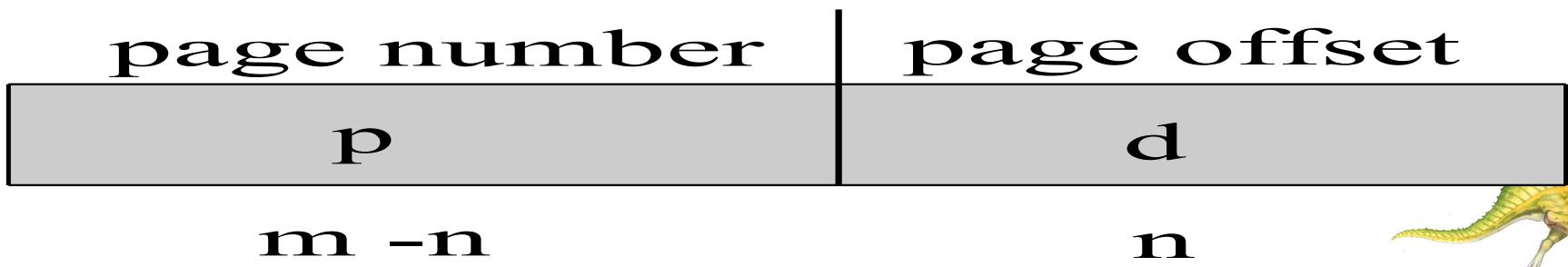
- Physical address space of a process can be non-contiguous; process is allocated physical memory whenever the latter is available
 - Avoids external fragmentation **and the need for compaction**
 - Avoids problem of varying sized memory chunks
- **Paging**
 - Divide physical memory into fixed-sized blocks called **frames**
 - ▶ Block size is a power of 2; between 512 bytes and 1 Gbytes
 - Divide logical memory into blocks of same size **as the frames** called **pages**
 - **The page size (like the frame size) is defined by the hardware; see Slide-29**
 - Divide backing store into [**clusters of**] blocks of same size as the frames
- 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
- Still have Internal fragmentation





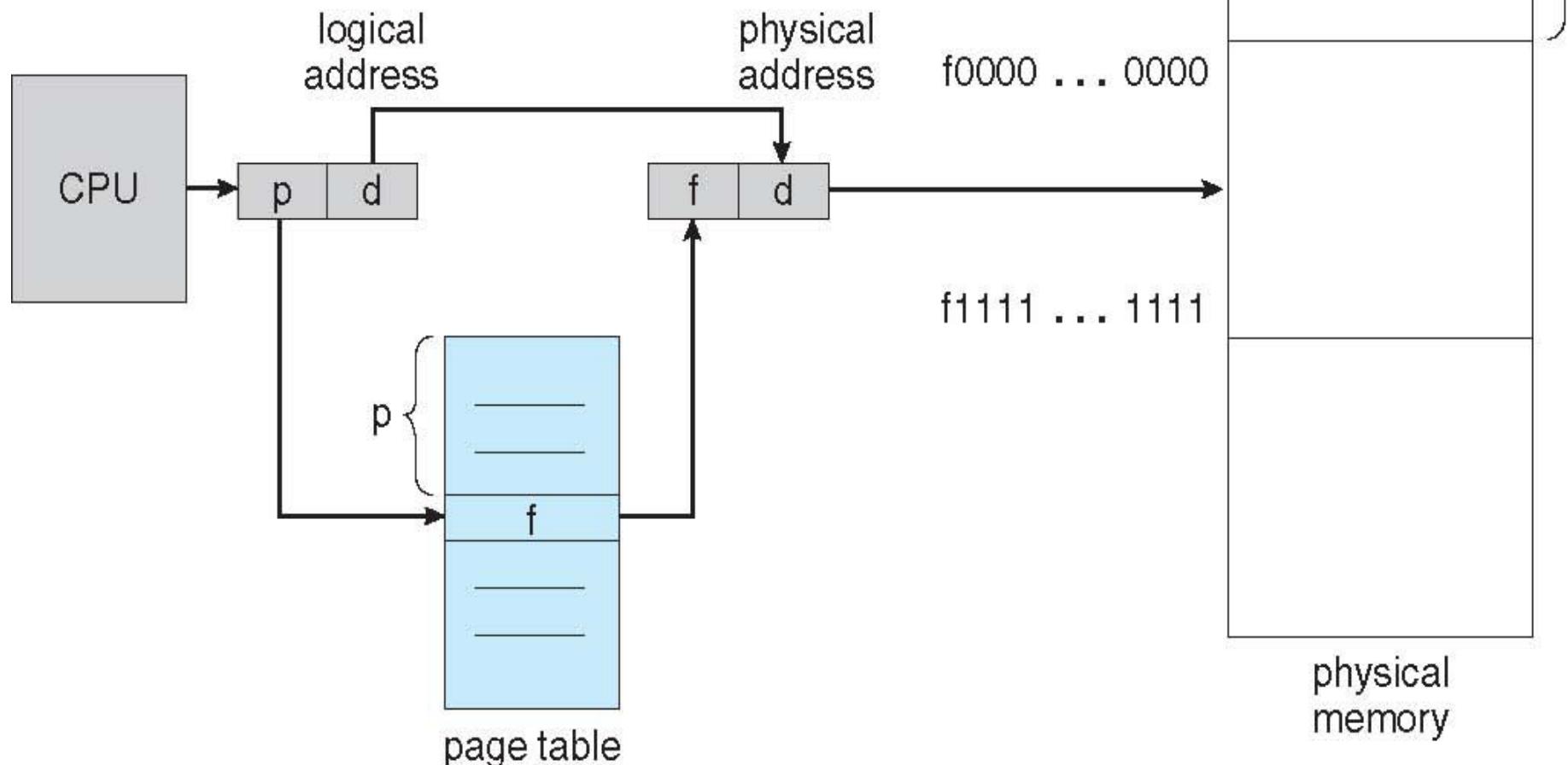
Address Translation Scheme

- Each address generated by CPU is divided into two parts:
 - **Page number (p)** – used as an index into a **page table**
 - ▶ Page table contains the base address of each page in physical memory
 - **Page offset (d)** – combined with the base address to define the physical memory address that is sent to the memory unit
 - If the logical address space is 2^m and the page size is 2^n
 - ▶ The binary representation of the logical address has m bits, such that
 - The $m - n$ leftmost bits designate the page number p
 - » **p is index into the page table**
 - The rightmost n bits designate the page offset d
 - » **d is displacement within the page (and its corresponding frame)**



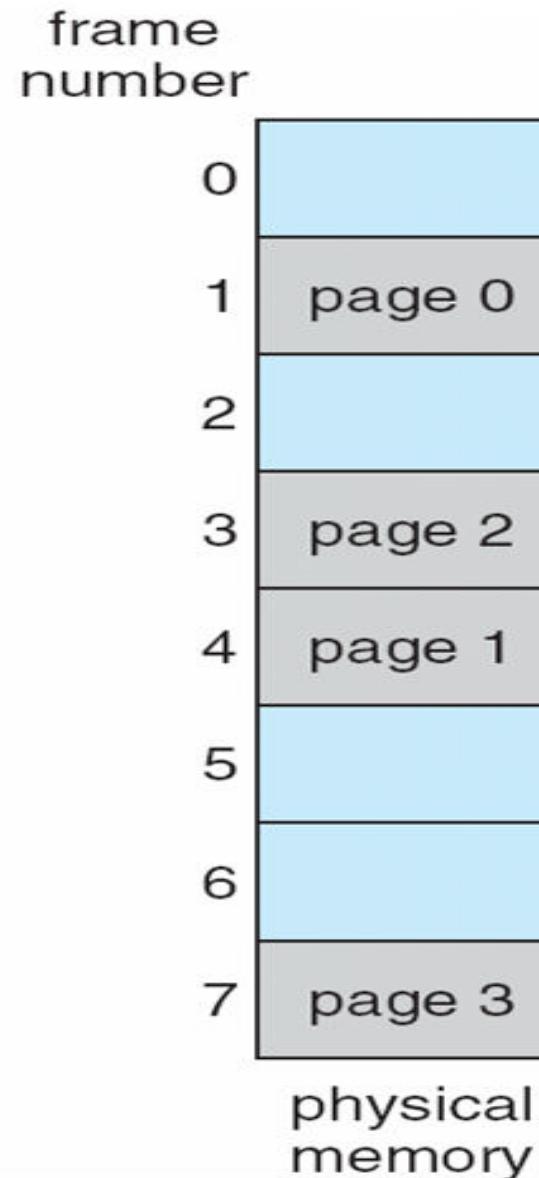
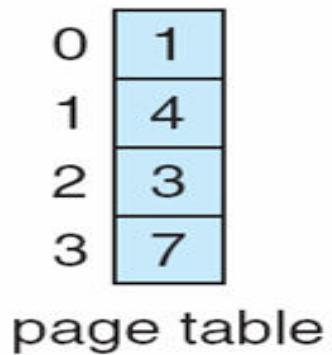
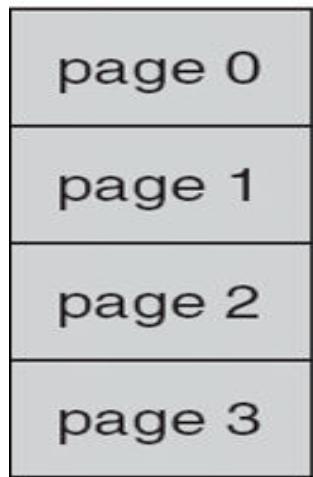


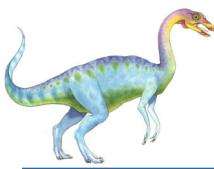
Paging Hardware





Paging Model of Logical and Physical Memory



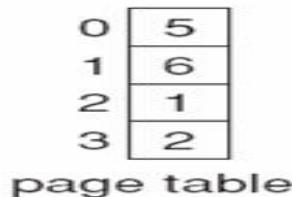


Paging Example

- Example: $n = 2$ and $m = 4$ and using a 32-byte phys memory (8 pages) with 4-byte pages
- Logical address 0 is [$p = 0, d = 0$]; Page 0 is in frame $f = 5$ at phys address $(f \times m) + d$
 - Thus logical address 0 maps to physical address 20 [= $(5 \times 4) + 0$];

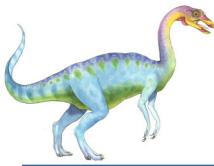
0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory



0	
4	i j k l
8	m n o p
12	
16	
20	a b c d e f g h
24	
28	

physical memory



Paging

- There is no external fragmentation when using paging scheme
- **Internal fragmentation is possible.** Calculating internal fragmentation
 - Page size = 2,048 bytes; thus, $n = 11$.
 - Process size = 72,766 bytes = $[(2048 * 35) + 1086]$ bytes
 - ▶ Process has 35 pages + 1,086 bytes; thus, 36 frames are required
 - ▶ Internal fragmentation of $2,048 - 1,086 = 962$ bytes of un-used memory
 - Worst case fragmentation is when 1 frame contains only 1 byte of used mem
 - Average fragmentation is about 1-half page size per process
 - So: are small page sizes more desirable?
 - ▶ Not necessarily; each page-table entry takes memory to track (overhead)
 - ▶ Large frame sizes better when transferring data to/from disk; efficient disk I/O
 - Page size: 4KB ~ 8KB but growing over time; now, research on variable page size
 - ▶ Solaris supports two page sizes – 8 KB and 4 MB; fixed multiple page sizes
- Process view and physical memory now very different
- By implementation process can only access its own memory

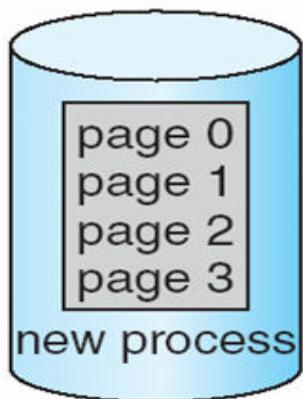




Free Frames

free-frame list

14
13
18
20
15

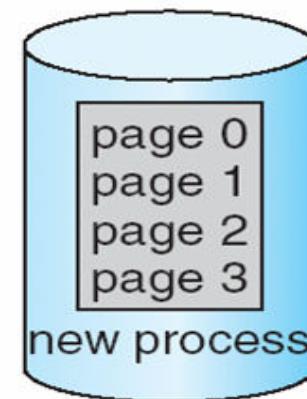


(a)

Before allocation

free-frame list

15



(b)

After allocation



Implementation of Page Table

- Process's 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
 - Both, PTBR and PTLR are also stored in the process's PCB
- In this scheme every access to data or instruction requires two memory accesses
 - **First:** access the page table using PTBR value to retrieve its frame number
 - **Second:** access the actual memory location given the frame number
 - ▶ This is a serious time overhead that needs to be reduced
- The two-memory-access problem can be solved by the use of a **special fast-lookup hardware cache** called **associative memory** or **translation look-aside buffers (TLBs)**





Associative Memory - TLB

- Associative memory – parallel search

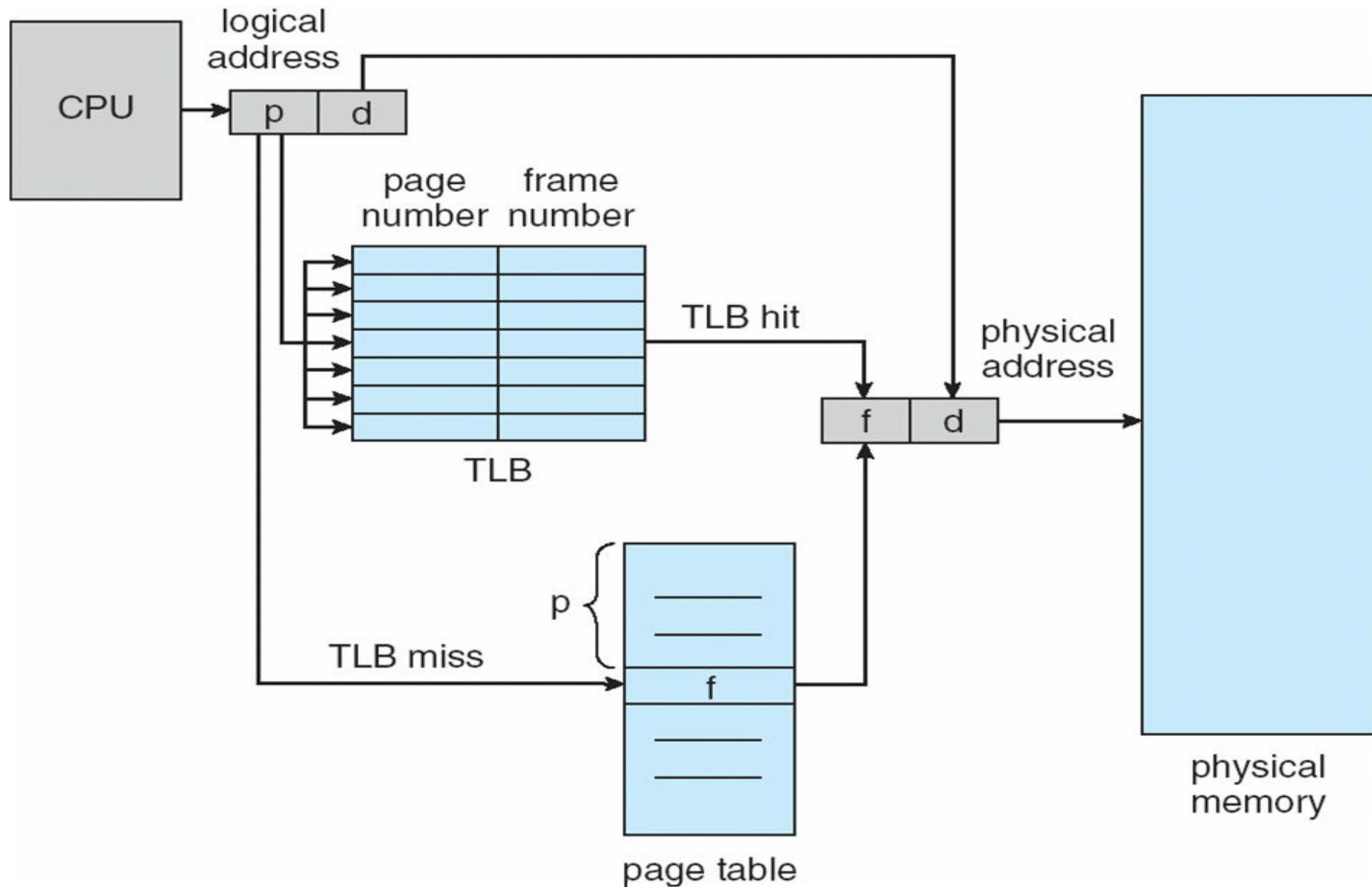
Page #	Frame #

- Address translation give (p, d)
 - If p is in associative memory, then retrieve the corresponding frame #
 - Otherwise retrieve the corresponding frame # from the page table in memory
 - ▶ This is a **TLB miss**; the **page # and frame #** are then added to the TLB for future references





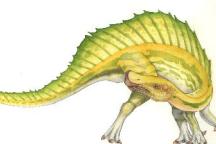
Paging Hardware With TLB

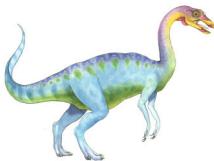




Implementation of Page Table

- TLBs are typically small (64 to 1,024 entries)
- On a TLB miss, new entry (*p, f*) is loaded into the TLB for faster access next time
 - Replacement policies must be considered; **if TLB is already full.**
 - ▶ Examples:
 - **LRU – Least Recently Used** replacement algorithm
 - **RR – Round Robin** replacement algorithm
 - **Random** replacement algorithm
 - Some entries can be **wired down** for permanent fast access
 - ▶ Cannot be removed by any replacement algorithm
 - Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry
 - Uniquely identifies each process and is used to provide address-space protection for that process
 - ▶ Otherwise need to flush the current TLB at every context switch





Effective Memory-Access Time

- Associative Lookup Time = ϵ time units
 - Can be < 10% of memory access time
- Hit ratio = α
 - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Consider hit ratio α , TLB search time ϵ , and memory access time m
 - **Effective Access Time (EAT)**
$$\text{EAT} = (m + \epsilon)\alpha + (2m + \epsilon)(1 - \alpha)$$
$$= 2m + \epsilon - m\alpha$$
hit time + miss time
- Consider $\alpha = 80\%$, $\epsilon = 20\text{ns}$ for TLB search, $m = 100\text{ns}$ for memory access
 - EAT = $0.80 \times (100 + 20) + 0.20 \times (200 + 20) = 140\text{ns}$ (**40% slowdown**)
- Consider more realistic hit ratio $\alpha = 99\%$, $\epsilon = 20\text{ns}$ for TLB search, $m = 100\text{ns}$ for memory access
 - EAT = $0.99 \times (100 + 20) + 0.01 \times (200 + 20) = 121\text{ns}$ (**21% slowdown**)





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 frame protection as
 - ▶ execute-only, read-write, or read-only,
 - Or, add separate protection bit for each kind of access
- **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
 - OS sets this bit for each page to allow/disallow access to its frame
 - “**invalid**” indicates that the page is not in the process’ logical address space
 - Or use **page-table length register (PTLR)**; see Slide-34
- Any violations result in a trap to the kernel





Valid (v) or Invalid (i) Bit in a Page Table

00000

page 0
page 1
page 2
page 3
page 4
page 5

10,468

12,287

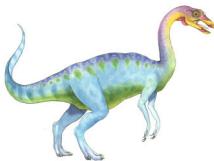
frame number

valid-invalid bit

0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

page table

0	
1	
2	page 0
3	page 1
4	page 2
5	
6	
7	page 3
8	page 4
9	page 5
	⋮
	page <i>n</i>



Shared Pages

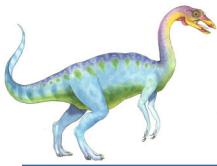
■ Shared code

- Also useful in time-sharing environments; e.g. many users using a text editor
- One copy of read-only (**reentrant**) code is shared among processes
 - ▶ i.e., text editors, compilers, window systems, ... etc
 - ▶ More than two processes can execute the same code at the same time
 - Only one copy of the code (e.g., text editor) need be kept in memory
- Similar to multiple threads sharing the same process space
- Also useful for inter-process communication if sharing of read-write pages is allowed.
 - ▶ Some OS implement shared-memory using shared pages.

■ 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

ed 1
ed 2
ed 3
data 1

process P_1

3
4
6
1

page table
for P_1

ed 1
ed 2
ed 3
data 3

process P_3

3
4
6
2

page table
for P_3

ed 1
ed 2
ed 3
data 2

process P_2

3
4
6
7

page table
for P_2

0
1
2
3
4
5
6
7
8
9
10
11



Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
 - Consider a 32-bit logical address space as on modern computers
 - Page size of 4 KB (2^{12})
 - ▶ Page table would have 1 million entries ($2^{32} / 2^{12}$); **number of pages**
 - If each entry is 4 bytes
 - ▶ Then each process may need up to 4 MB of physical address space to store its page table alone
 - Very costly to store **in main memory**
 - Do not want to allocate the page table contiguously in main memory

Solutions:

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

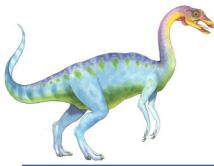




Hierarchical Page Tables

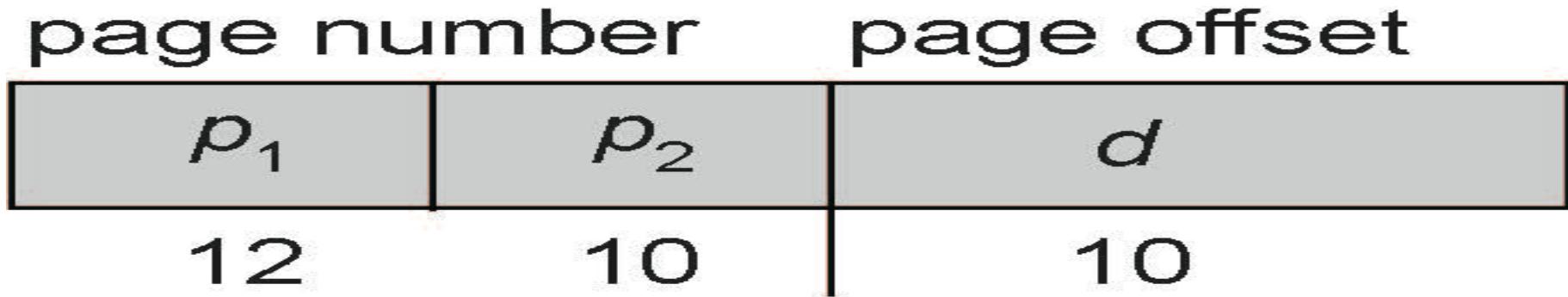
- Two-Level Paging Algorithm
 - The page table itself is also *paged*
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table





Two-Level Paging Example

- Logical address (on 32-bit machine with 1KB page size) is divided into two parts:
 - a page number p consisting of 22 bits; see $m - n$ in Slide-28, $m = 32$
 - a page offset d consisting of 10 bits; see n in Slide-28, $n = 10$
- Since the page table is paged, the page number p is further divided into 2 parts:
 - a 12-bit **outer** page number p_1
 - a 10-bit **inner** page offset p_2
- Thus, a logical address is as follows:

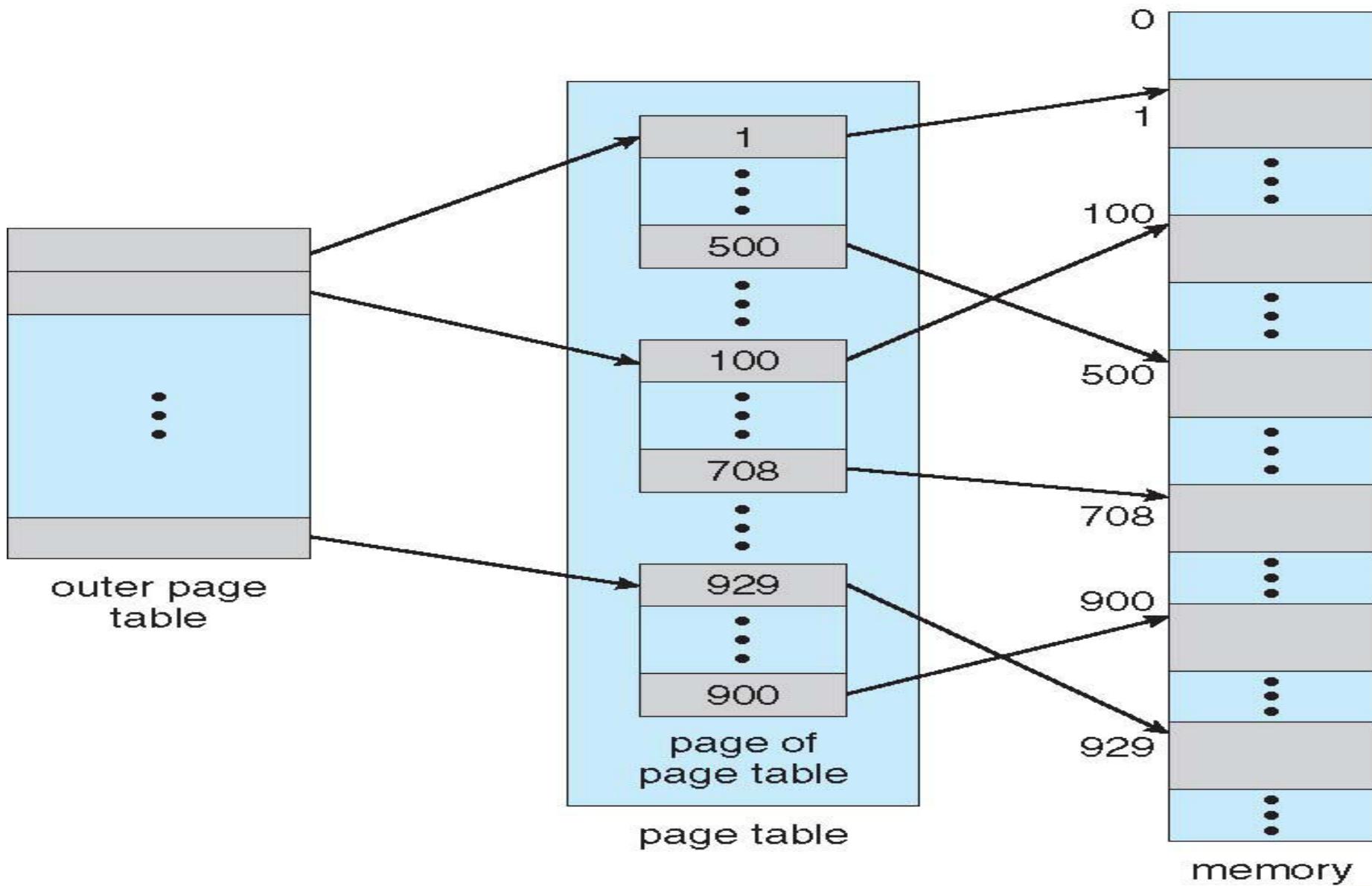


- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**





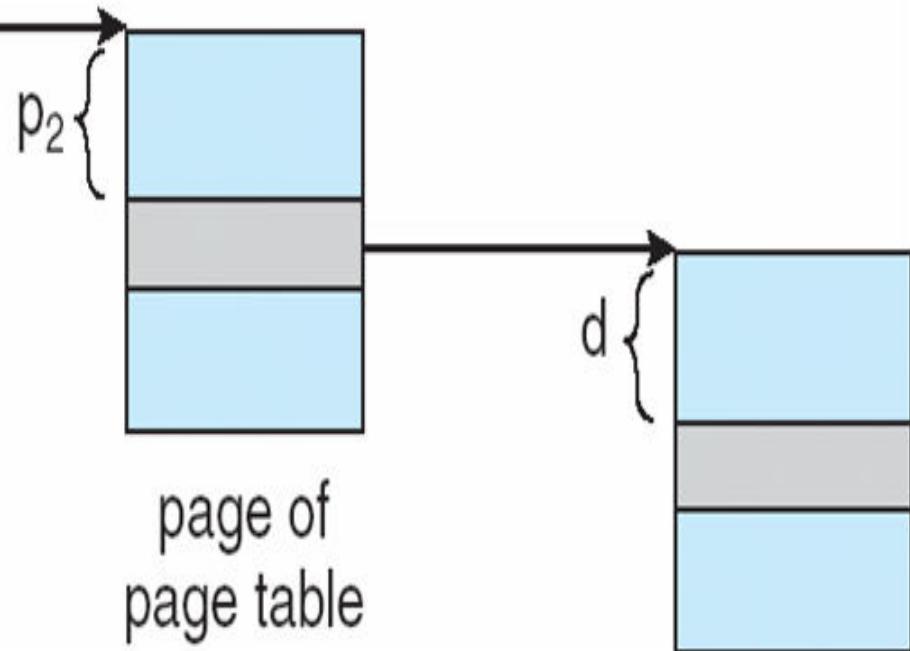
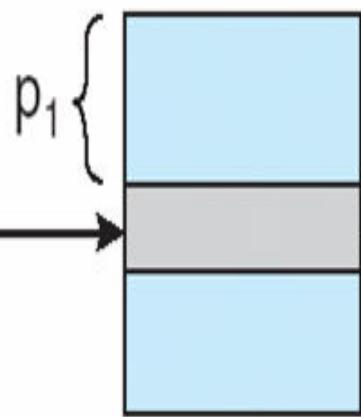
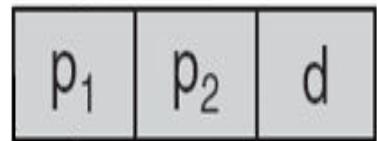
Two-Level Page-Table Scheme

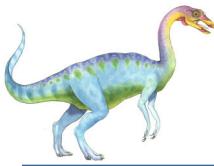




Address-Translation Scheme

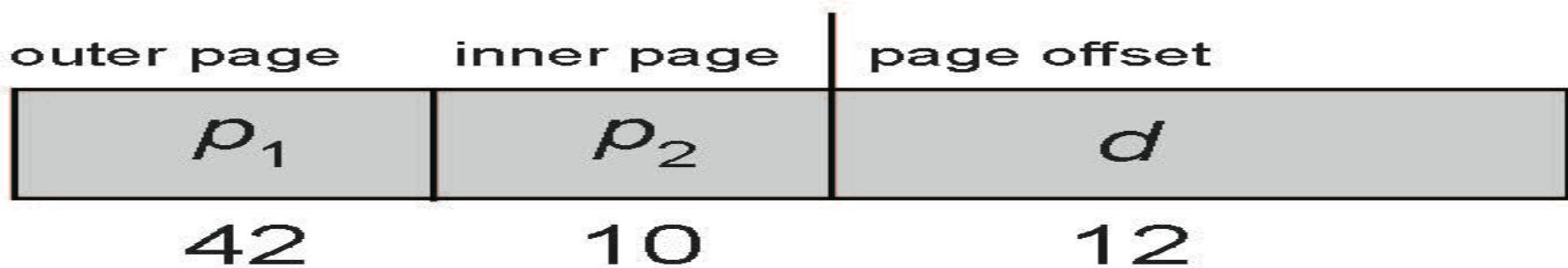
logical address





64-bit Logical Address Space

- Two-level paging scheme may not be sufficient
 - 64-bit addressing = 2^{64} of physical address space to store a page table ☺
- If page size is 4 KB (2^{12})
 - Then page table has 2^{52} entries ($2^{64} / 2^{12}$); number of pages
 - If we use two level scheme, then inner page table 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 2nd outer page table
- But in the following example the 2nd outer page table is still 2^{32} bytes in size
 - ▶ And possibly 4 memory access to get to one physical memory location





Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12
2nd outer page	outer page	inner page
p_1	p_2	p_3
32	10	10
		12



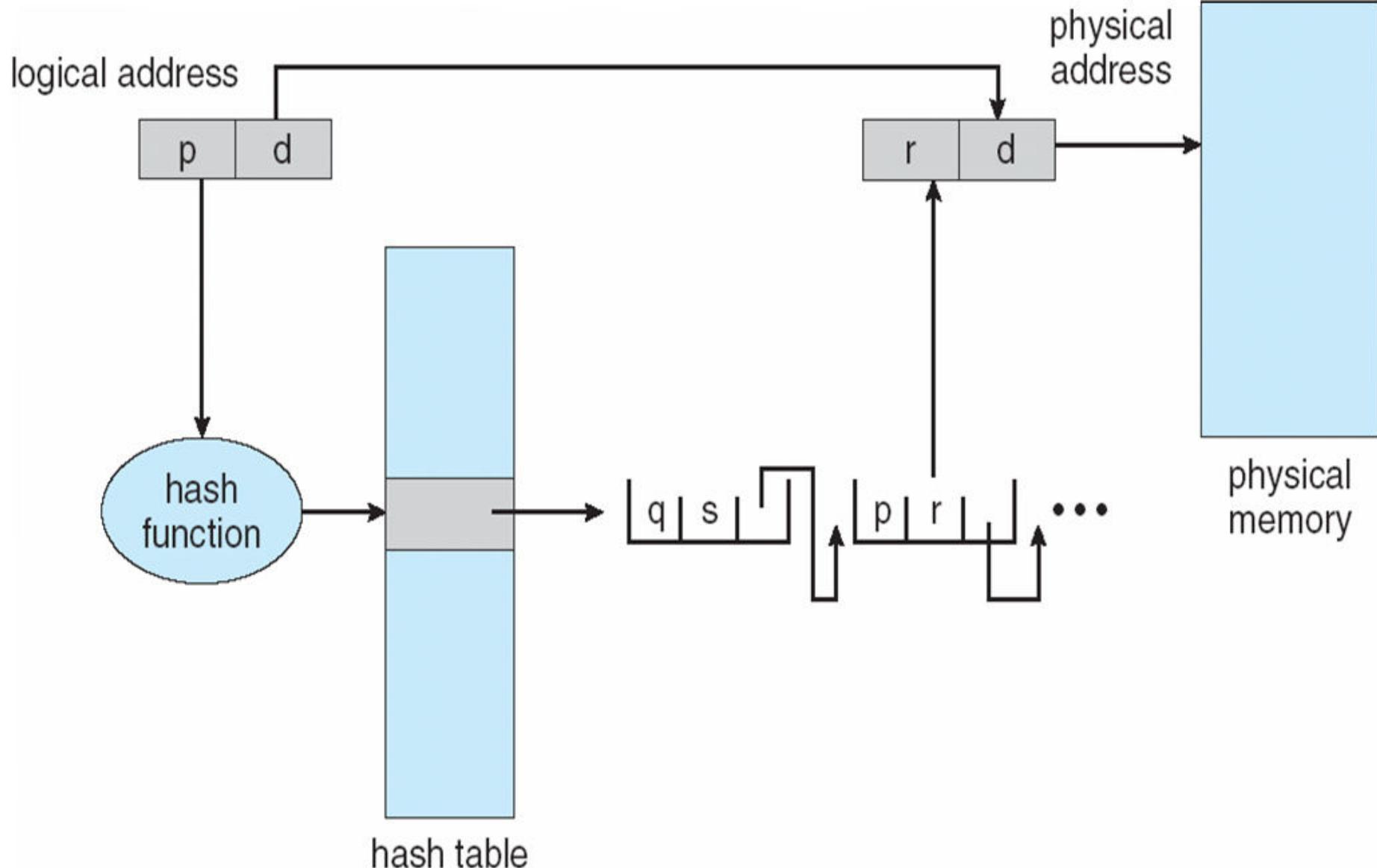
Hashed Page Tables

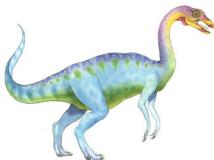
- Common in address space > 32 bits
- The virtual page number (**the hash value**) is hashed into a page table
 - Each entry in table is a **linked list of elements** hashing to the same location
 - ▶ For collision handling
- Each element contains: (1) the virtual page number *p*, (2) the value of the mapped page frame *f*, and (3) a pointer to the next element *e*
- **Algorithm:** Virtual page numbers are compared in this list 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





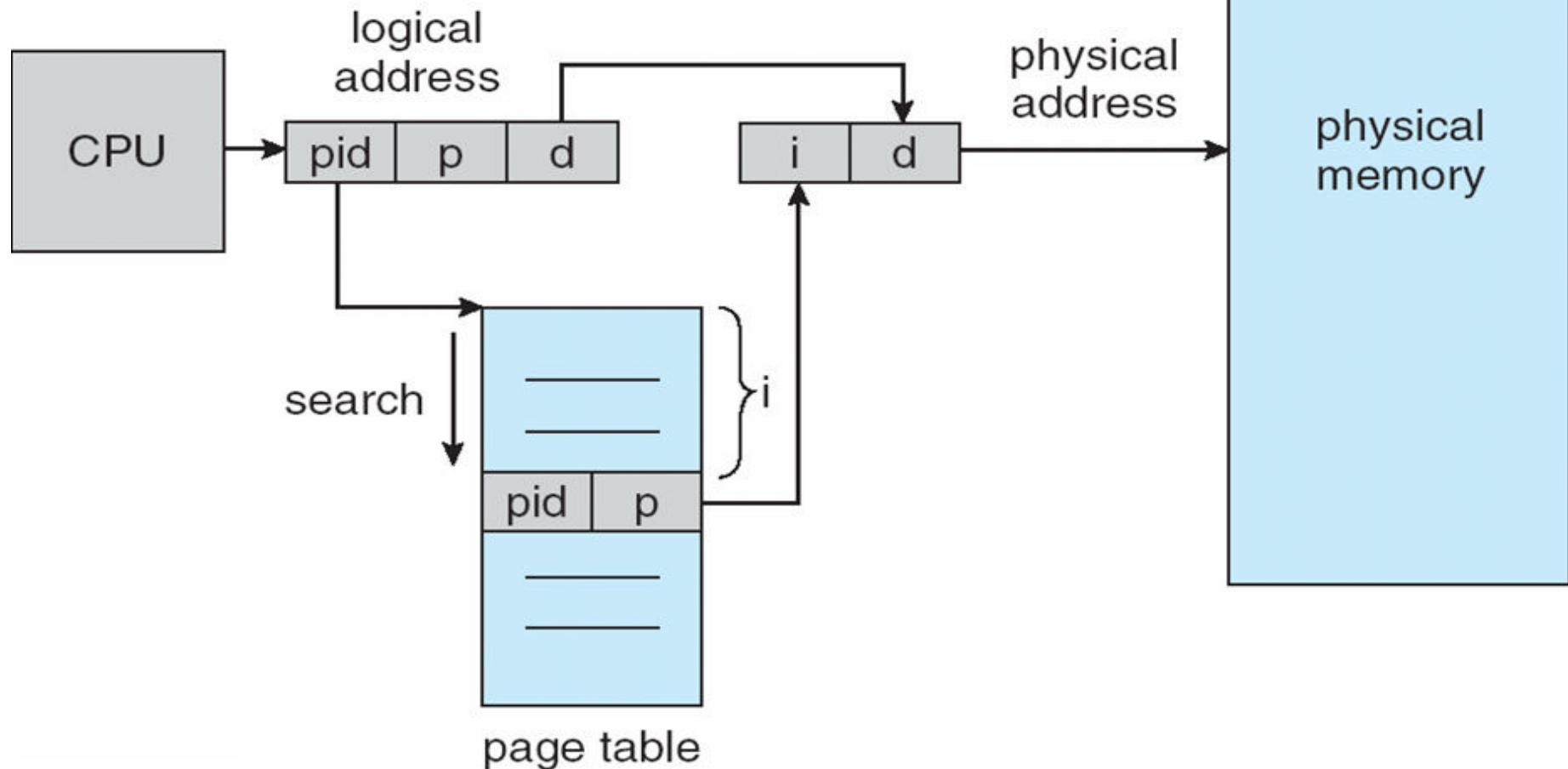
Inverted Page Table

- Normally, each process has a page table which keeps track of all its logical pages and its physical pages (the frames)
 - What if the table contains billions of entries...? Too costly
- **Solution:** use an inverted page table
 - It contains one entry for each **real page of memory; i.e., a frame**
 - Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
 - Thus: **only one page table** is in the system; having **only one entry** or each frame
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs; **whole table may be searched**
- 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 operating system example with tightly integrated HW
 - Goals are efficiency, low overhead
- Based on hashing, but more complex
- Two hash tables
 - One 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

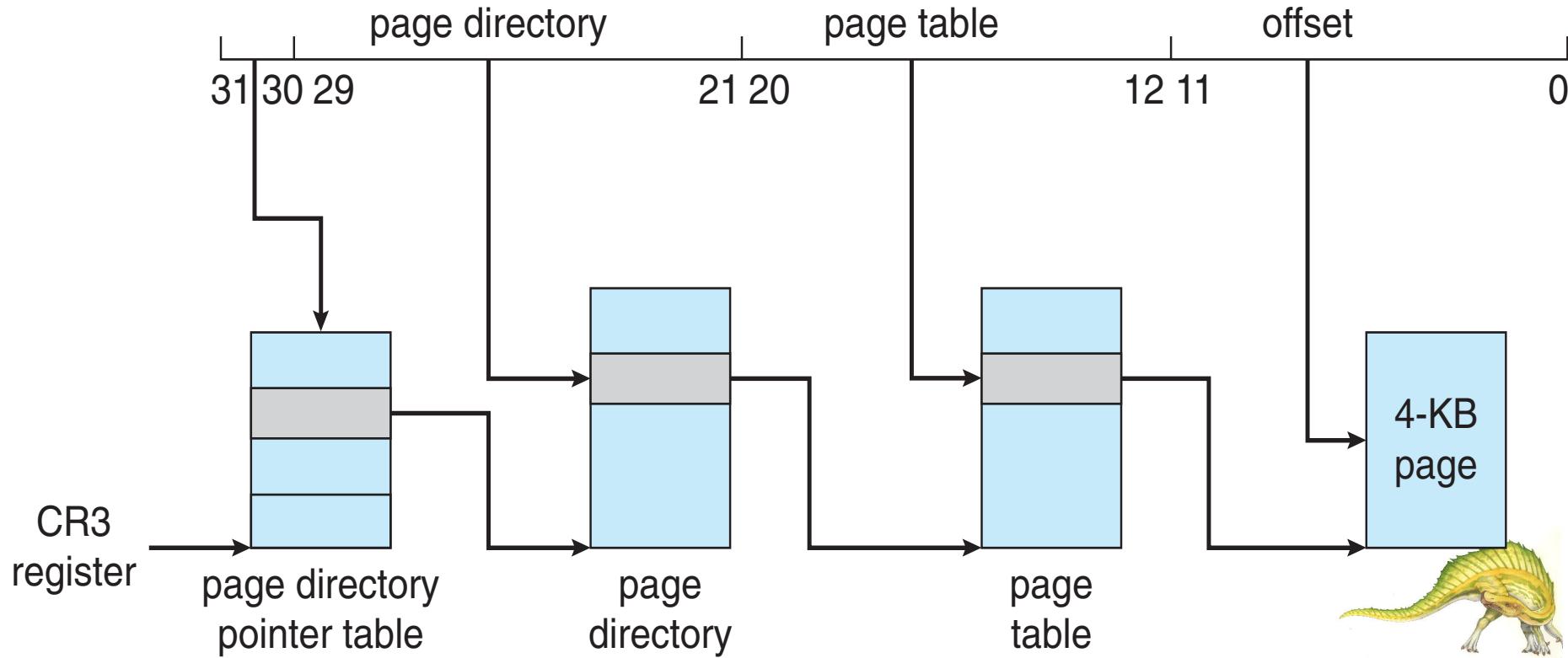
- TLB holds translation table entries (TTEs) for fast hardware lookups
 - A cache of TTEs reside in a translation storage buffer (TSB)
 - ▶ Includes an entry per recently accessed page
- Virtual address reference causes TLB search
 - If miss, hardware walks the in-memory TSB looking for the TTE corresponding 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.





Intel IA-32 Page Address Extensions

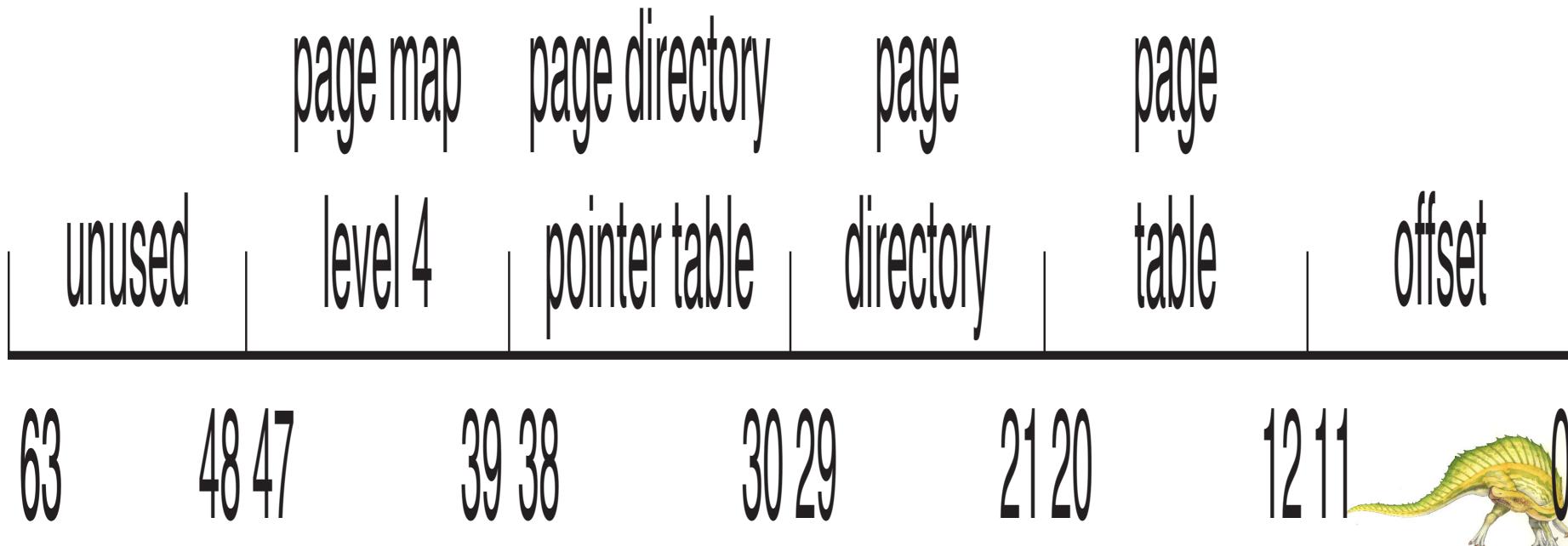
- 32-bit address limits led Intel to create **page address extension (PAE)**, allowing 32-bit apps access to more than 4GB of memory space
 - Paging went to a 3-level scheme
 - Top two bits refer to a **page directory pointer table**
 - Page-directory and page-table entries moved to 64-bits in size
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





Intel x86-64

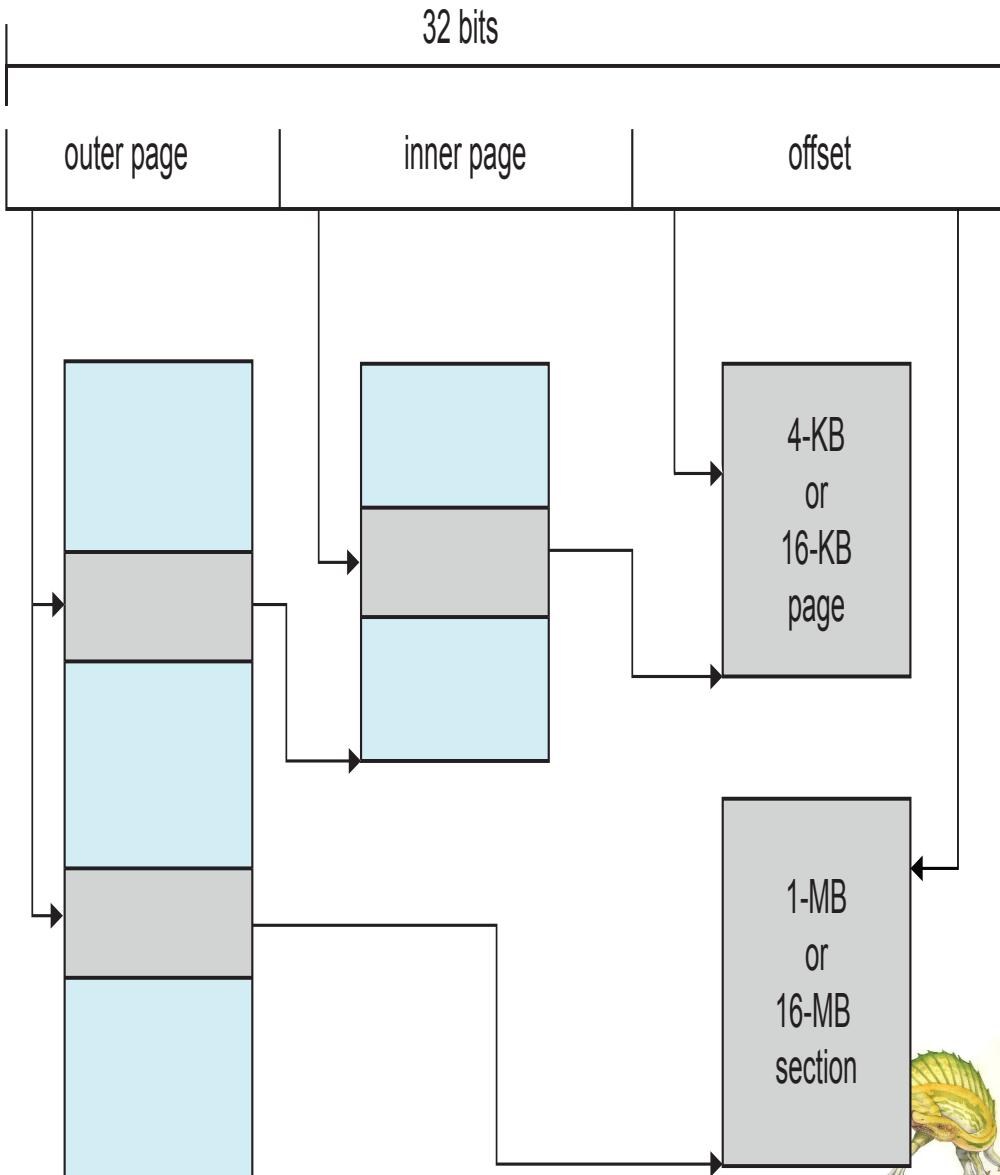
- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
 - Page sizes of 4 KB, 2 MB, 1 GB
 - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits



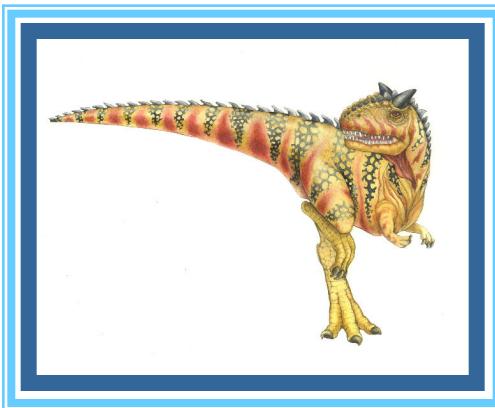


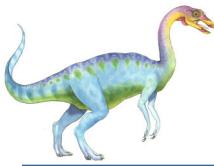
Example: ARM Architecture

- 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 8





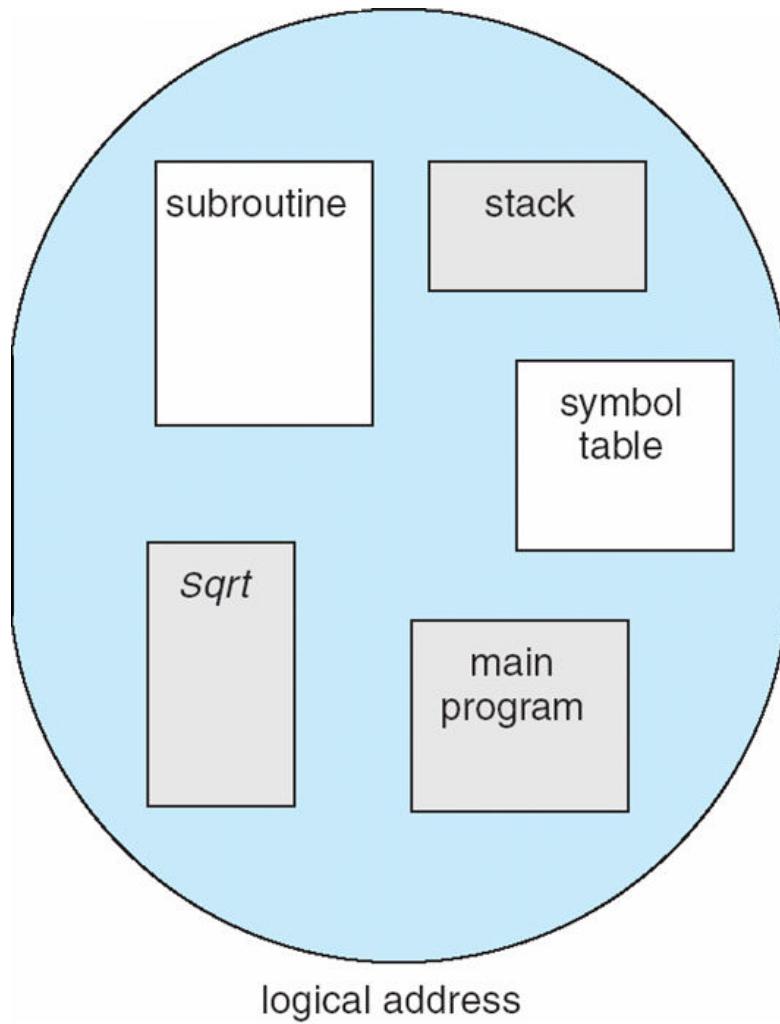
Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
 - A segment is a logical unit such as:
 - main program
 - procedure
 - function
 - method
 - object
 - local variables, global variables
 - common block
 - stack
 - symbol table
 - arrays



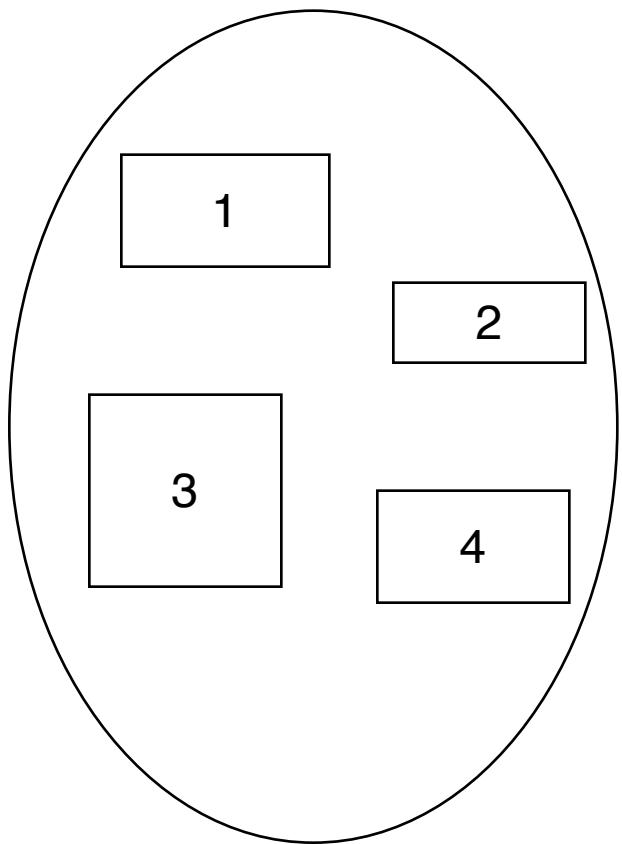


User's View of a Program

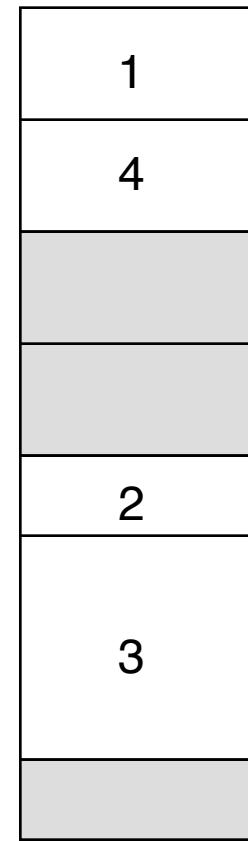




Logical View of Segmentation



user space



physical memory space





Segmentation Architecture

- Logical address consists of a two tuple:
 $\langle \text{segment-number}, \text{offset} \rangle,$
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
 - **base** – contains the starting physical address where the segments reside in memory
 - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;
segment number **s** is legal if **s < STLR**





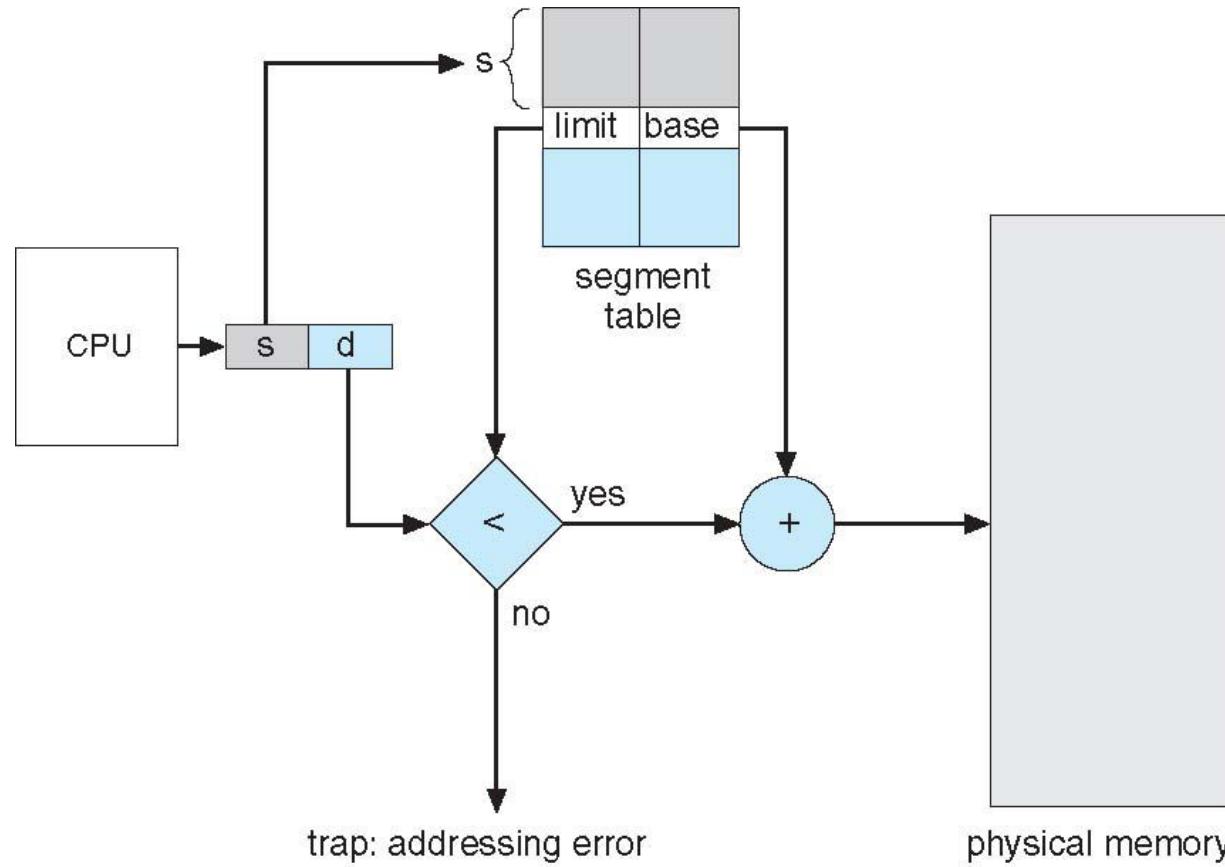
Segmentation Architecture (Cont.)

- Protection
 - With each entry in segment table associate:
 - ▶ validation bit = 0  illegal segment
 - ▶ read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram





Segmentation Hardware





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

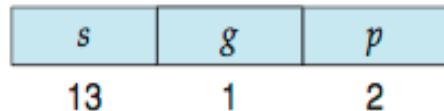
- Supports both segmentation and segmentation with paging
 - Each segment can be 4 GB
 - Up to 16 K segments per process
 - Divided into two partitions
 - ▶ First partition of up to 8 K segments are private to process (kept in **local descriptor table (LDT)**)
 - ▶ Second partition of up to 8K segments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

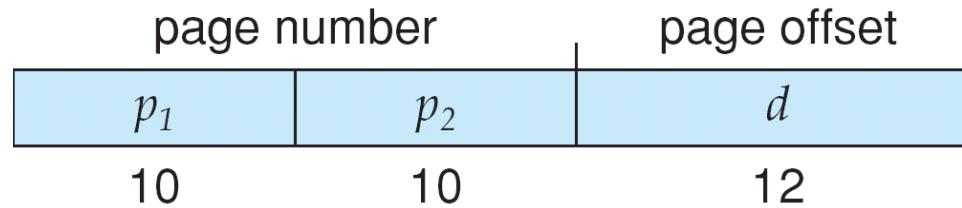
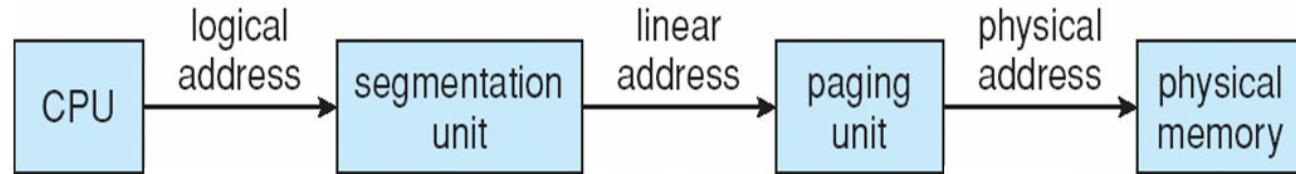


- Linear address given to paging unit
 - ▶ Which generates physical address in main memory
 - ▶ Paging units form equivalent of MMU
 - ▶ Pages sizes can be 4 KB or 4 MB



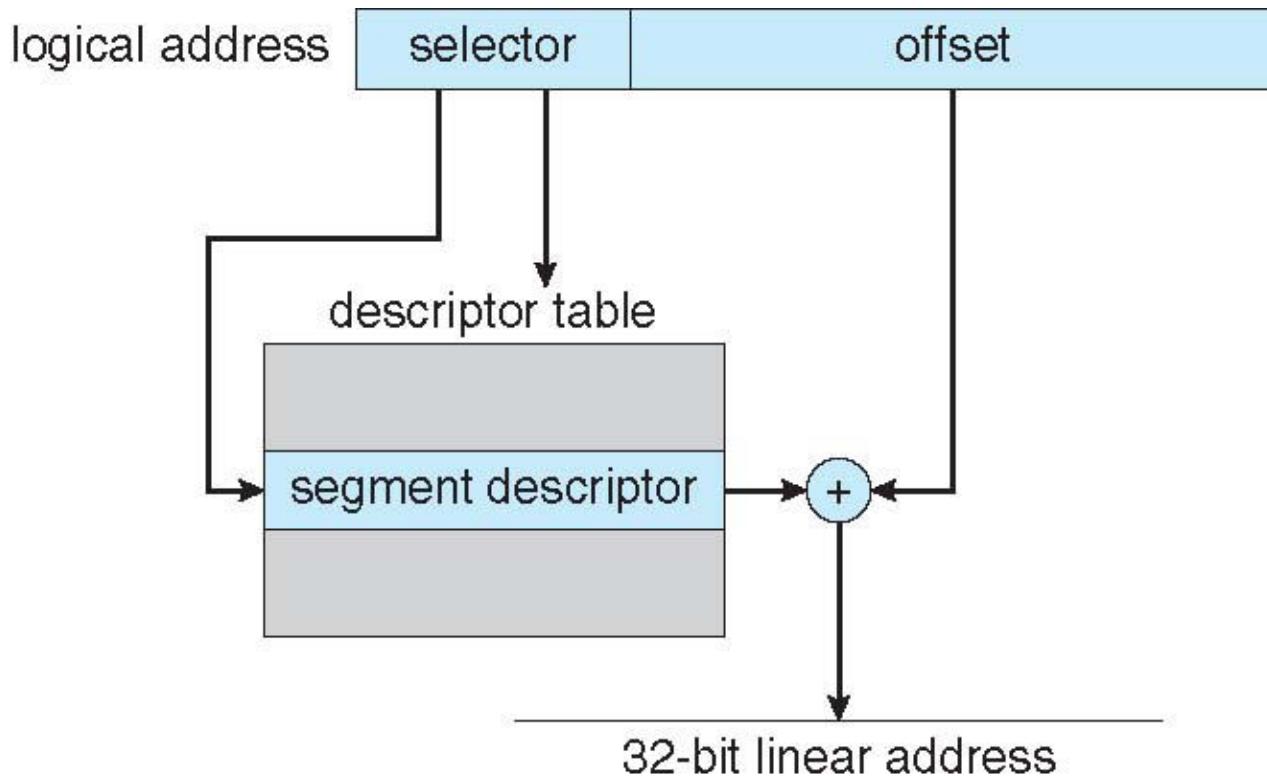


Logical to Physical Address Translation in IA-32



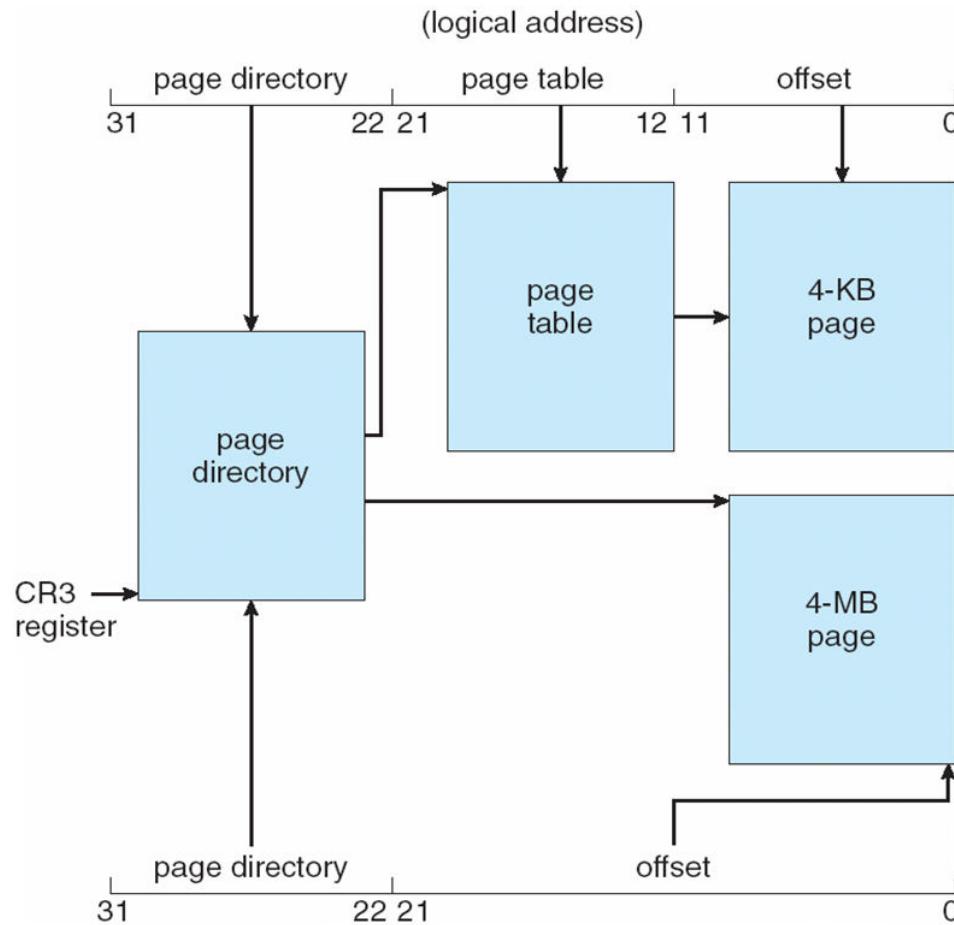


Intel IA-32 Segmentation

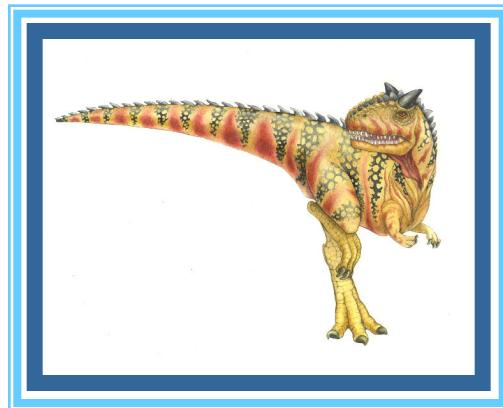




Intel IA-32 Paging Architecture



Chapter 9: Virtual Memory





Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

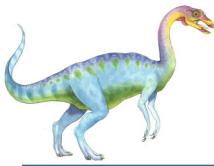




Objectives

- To describe the benefits of a virtual memory system
 - **Goal of memory-management strategies:** keep many processes in main memory to allow multi-programming; see Chap-8
 - ▶ **Problem:** Entire processes must be in memory before they can execute
 - **Virtual Memory** technique: running process need not be in memory entirely
 - ▶ Programs' memory need can be larger than physical memory
 - ▶ Abstraction of main memory; need not concern with storage limitations
 - ▶ Allows easy sharing of files and memory
 - ▶ Provide efficient mechanism for process creation
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model
- To examine the relationship between shared memory and memory-mapped files
- To explore how kernel memory is managed

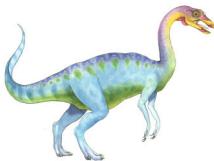




Background

- Code needs to be in memory to execute, but entire program rarely used
 - Error code, unusual routines, large data structures; **are all seldom used**
 - ▶ **Ex: declared array of size 100 cells but only 10 cells are used**
- Entire program code not needed (**in main memory**) at the same time
- Consider ability to execute partially-loaded program
 - Program no longer constrained by limits of physical memory
 - Each program takes less memory while running
 - ▶ Thus, more [**partially-loaded**] programs can run at the same time
 - ▶ Increased CPU utilization and throughput with no increase in response time or turnaround time; **more multi-programming and time-sharing**
 - Less I/O needed to load or swap programs into/from memory
 - ▶ Thus, each user program would run faster

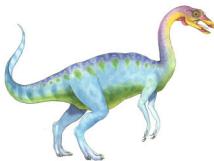




Background

- **Virtual memory** – separation of user logical memory from physical memory
 - ▶ As perceived by users; **that programs exist in contiguous memory**
 - ▶ Abstracts physical memory: need not worry about memory requirements
 - Only part of the program needs to be in memory for execution
 - Logical address space can therefore be much larger than physical address space
 - ▶ Programmers can work as if memory is an unlimited resource
 - Allows address spaces to be shared by several processes
 - Allows for more efficient process creation
 - More programs running concurrently; **increased multi-programming and/or time-sharing**
 - Less I/O needed to load or swap processes; **hence, faster program execution**





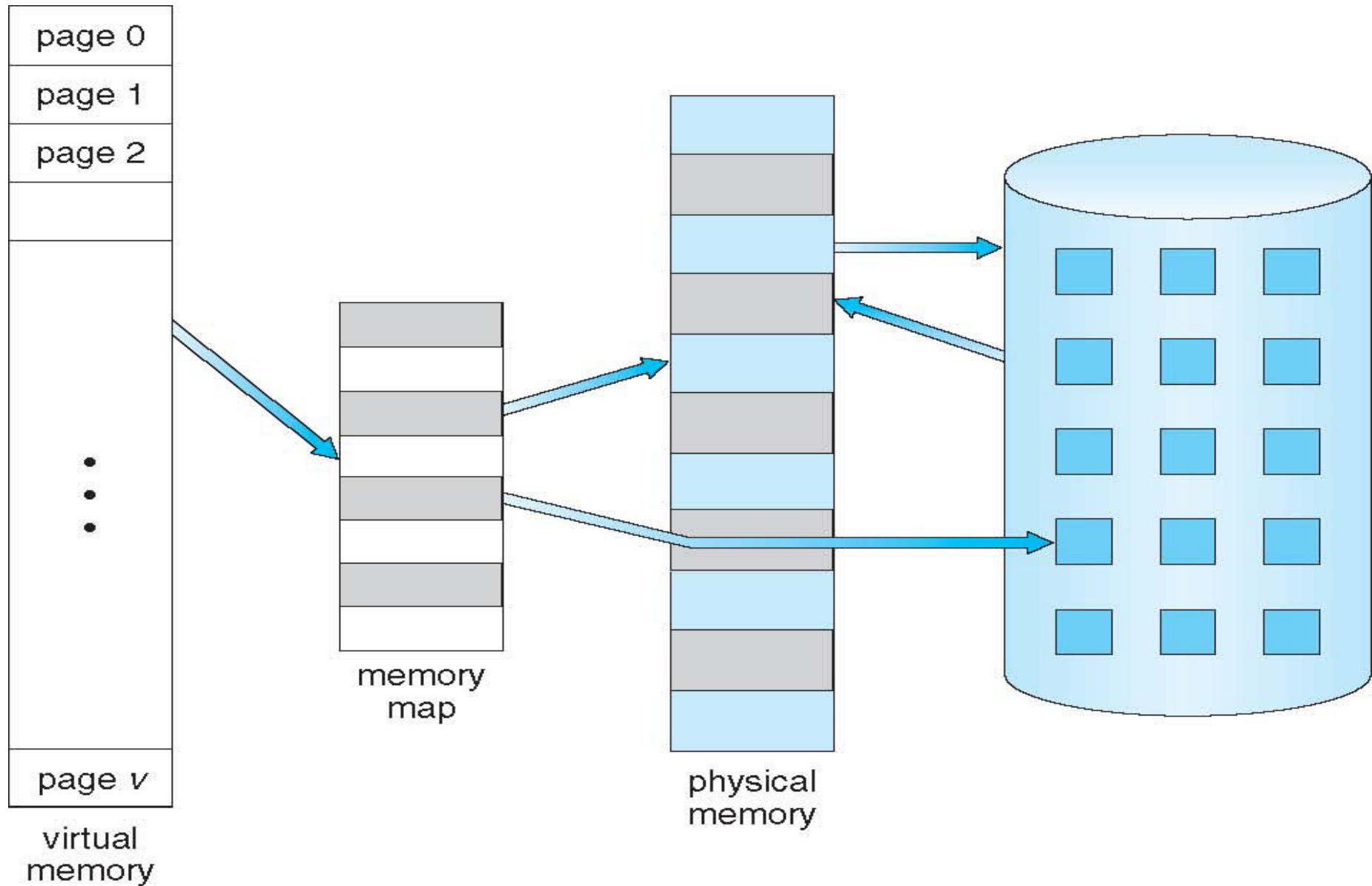
Background (Cont.)

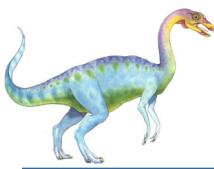
- **Virtual address space** – logical view of how process is stored in memory
 - Process starts at address 0 with contiguous addresses until end of its address space
 - As far as the user is concerned, his program is **contiguous** and *is* in memory
 - Meanwhile, physical memory organized in page frames; **not contiguous** (see Chap-8)
 - MMU maps logical pages (**user view**) to physical pages (i.e., **frames**) in memory
- Virtual memory can be implemented via:
 - Demand paging
 - Demand segmentation





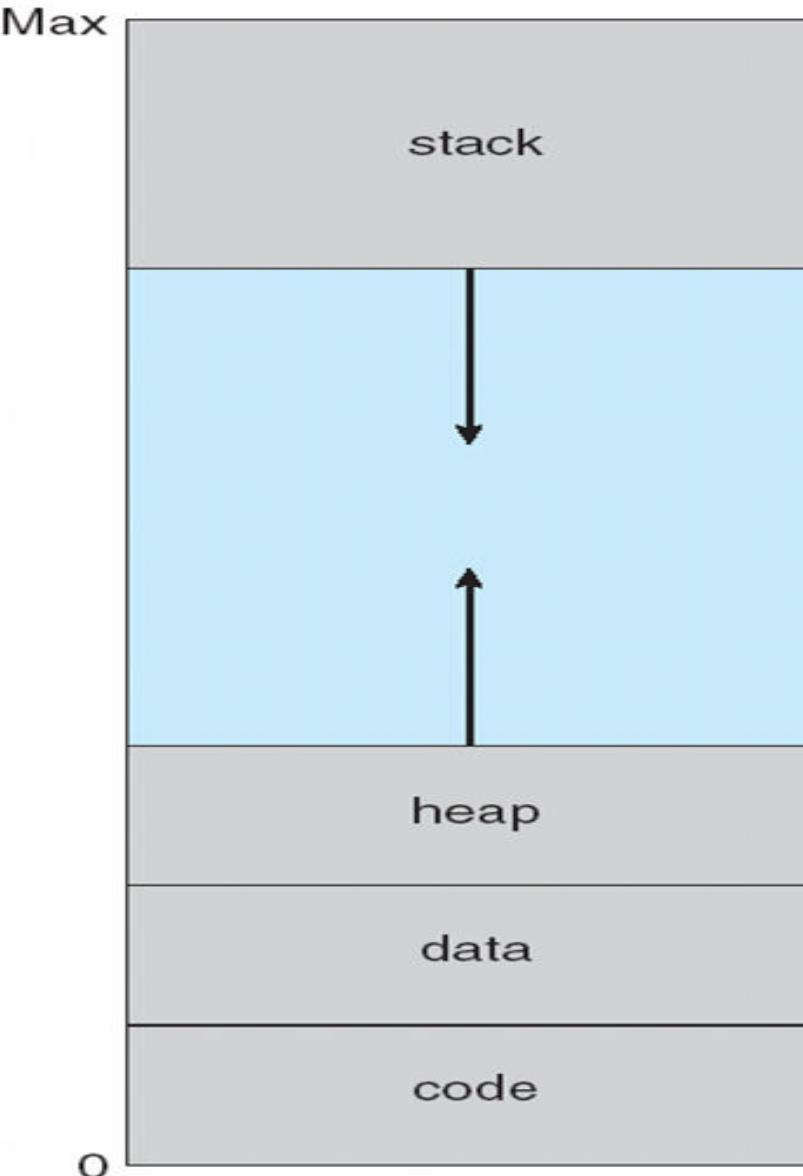
Virtual Memory That is Larger Than Physical Memory





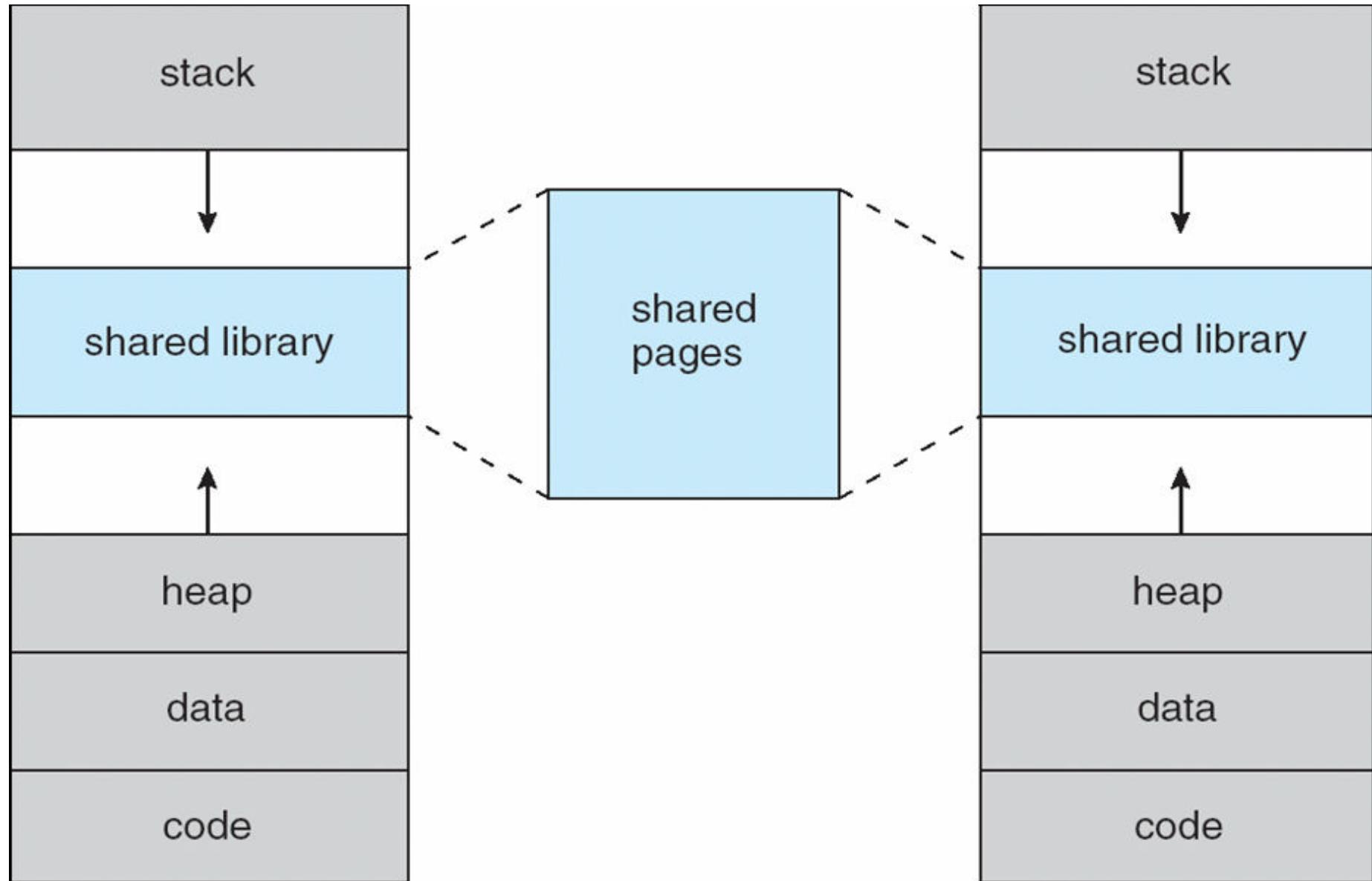
Virtual-Address Space of a Process

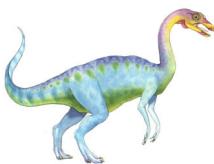
- For a process: heap grows upward while stack grows downward in memory; in process's space
 - Unused address space between the two is a **hole**; part of **virtual-address space**
 - ▶ Require actual physical pages only if the heap or the stack grow
 - ▶ Maximizes address space use
- Enables **sparse** address spaces with holes left for growth, or to dynamically link libraries, etc
- System libraries can be shared by many processes through mapping of the shared objects into virtual address space
- Processes can share memory by mapping read-write pages into virtual address space
- Pages can be shared during process creation with the `fork()`; speeding up process creation





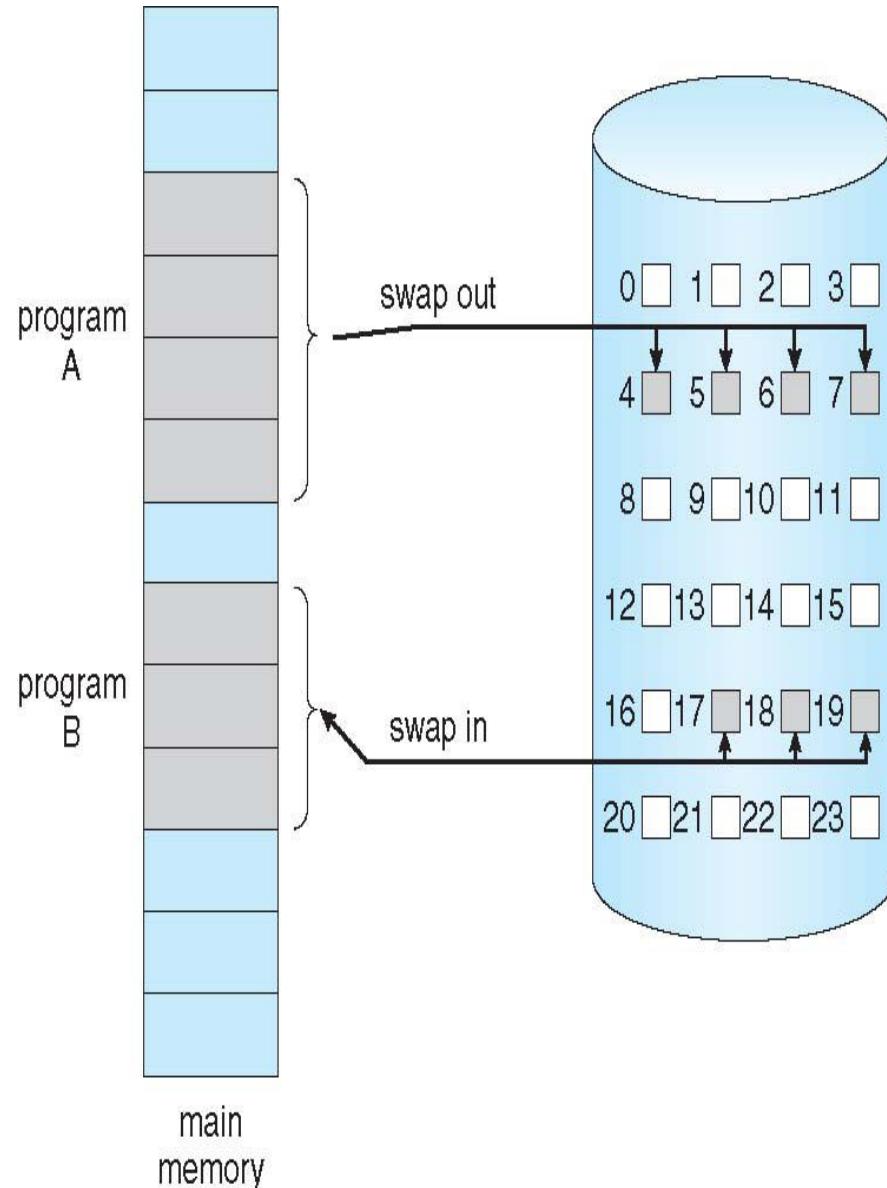
Shared Library Using Virtual Memory

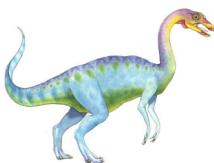




Demand Paging

- Could bring an entire process into memory at load time.
- Or bring a process's page into memory only when it is needed
 - Less I/O needed, no unnecessary I/O
 - Less memory needed
 - Faster response
 - More users
- Similar to a paging system with swapping (diagram on right)
- **Lazy swapper** – never swaps a page into memory unless page will be needed
 - Swapper that deals with pages is a [pager](#)
- Page is needed \Rightarrow reference it; [see Slide-14](#)
 - invalid reference \Rightarrow abort
 - not-in-memory \Rightarrow bring to memory

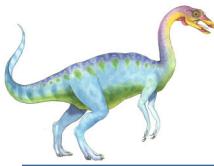




Basic Concepts

- When swapping in a process, the pager guesses which **pages** will be used before swapping it out again
 - The pager brings in **only** those **needed** pages into memory
 - ▶ Thus, decreases swap time and amount of needed physical memory
 - How to determine that set of pages?
- Need new MMU hardware support to implement demand paging; **see Slide-15**
 - To distinguish between **in-memory** pages and **on-disk** pages
 - ▶ Uses the **valid—invalid scheme** of Slide-40 Chap-8
- If pages needed are already **memory resident**
 - Execution proceeds normally, as in non demand-paging
- If page needed and is not memory resident; **see Slide-14**
 - Need to find the needed page from the disk and load it into memory
 - ▶ Without changing program behavior
 - ▶ Without programmer needing to change code





Valid-Invalid Bit

- A valid–invalid bit is associated with each page-table entry; see Chap-8, Slide-40 ($v \Rightarrow$ in-memory – **memory resident** ; $i \Rightarrow$ not-in-memory)
- Initially valid–invalid bit is set to i on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
-	

page table

- During MMU address translation:
 - if valid–invalid bit in page-table entry is $i \Rightarrow$ there is a page fault

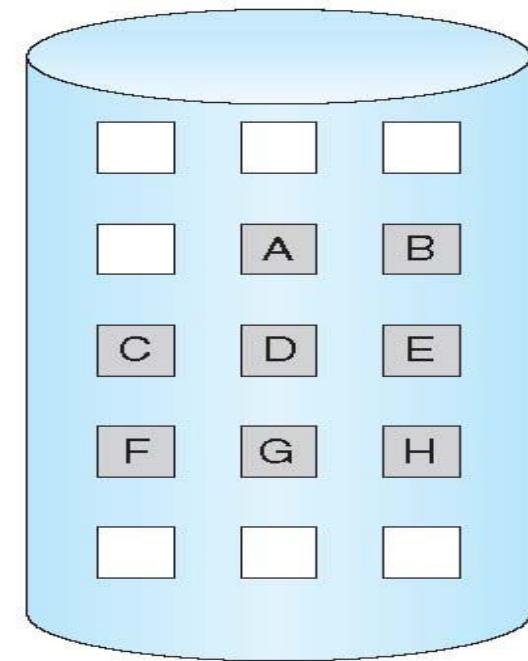
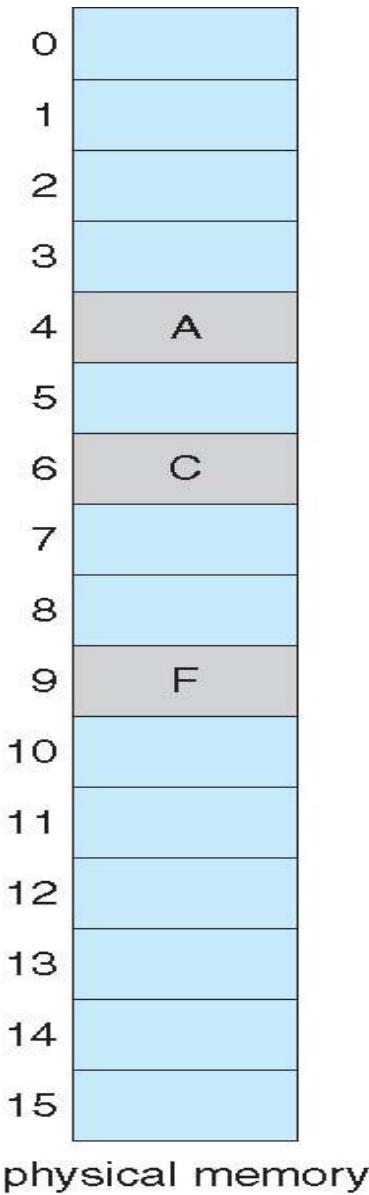
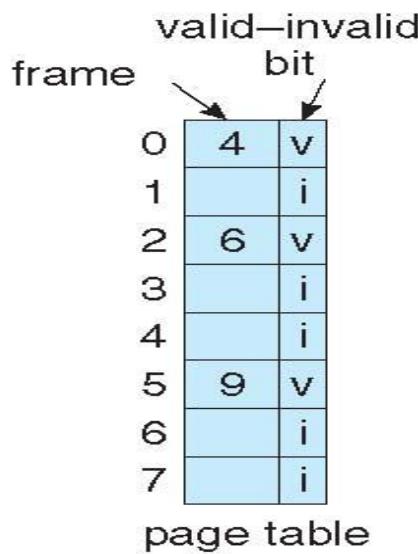


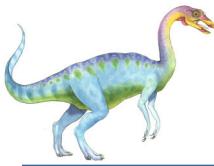


Page Table When Some Pages Are Not in Main Memory

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

logical memory





Page Fault

What if the process refers to (i.e., tries to access) a page not in-memory ?

- The [first] reference (i.e., address) to that **invalid** page will trap to operating system and causes a **page fault**

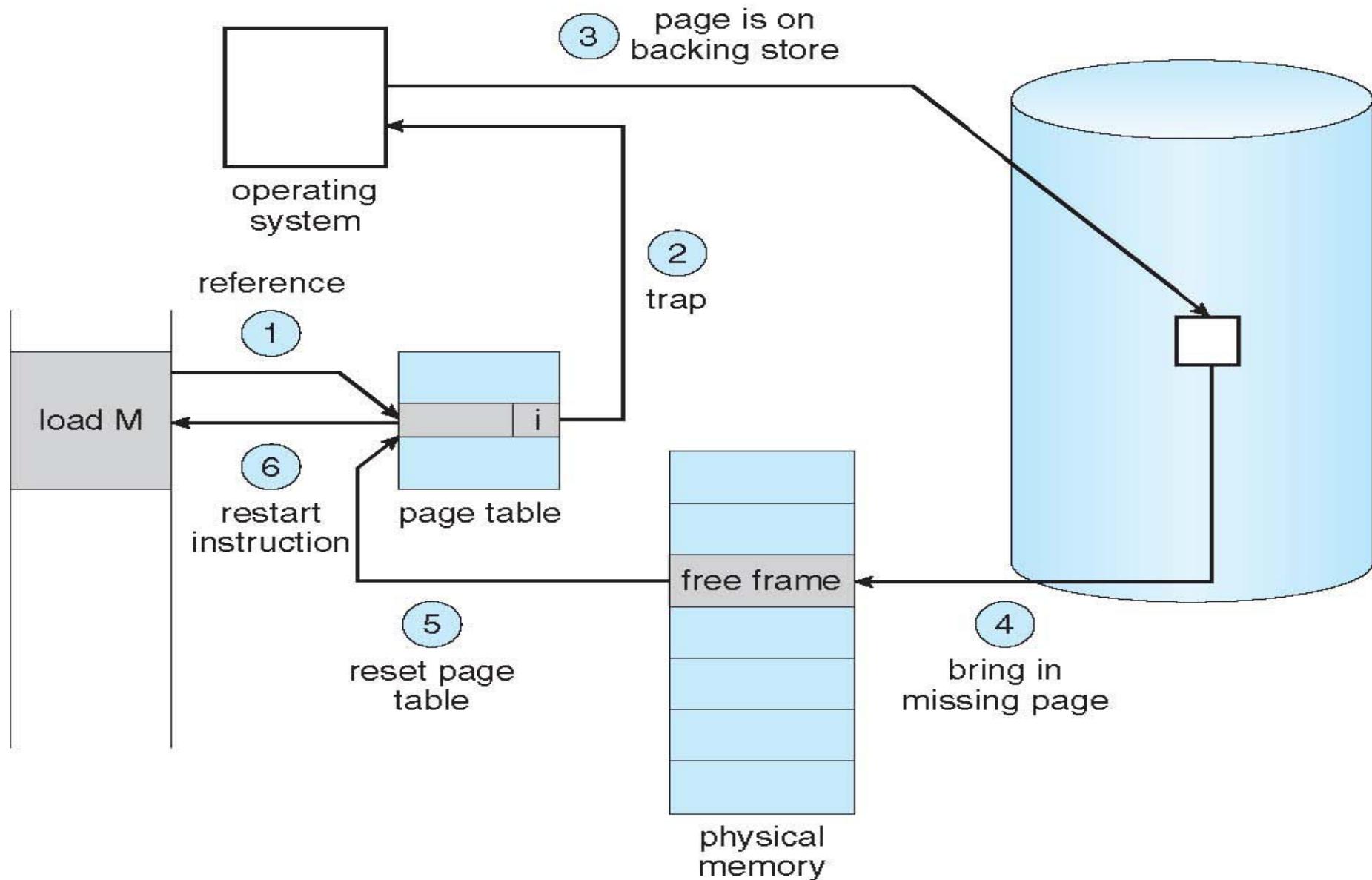
Procedure for handling a page fault

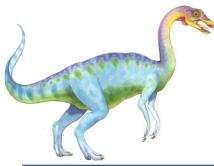
1. OS checks an internal table to see if reference is valid or invalid memory access
2. If
 - Invalid reference \Rightarrow abort the process
 - ▶ address is not in logical address space of process
 - Just not in memory \Rightarrow page in the referred page from the disk
 - ▶ logical address is valid but page is simply not in-memory
3. Find a free frame; **see Chap-8**
4. Read the referred page into this allocated frame via scheduled disk operation
5. Update both internal table and page-table by setting validation bit = **v**
6. Restart the instruction that caused the page fault and resume process execution





Steps in Handling a Page Fault





Aspects of Demand Paging

- Extreme case – start process with **no pages** in memory
 - OS sets instruction pointer to first instruction of process; **logical address = (p, d)**
 - ▶ Since page p is non-memory-resident then a page fault is issued
 - ▶ Page p is loaded and... same for all other process pages on first reference
 - This scheme is **pure demand paging**: load a page only when it is needed
- A given instruction may refer to multiple distinct pages; **thus, multiple page faults**
 - Consider fetching the instruction “**ADD A, B**” and fetching the values of data **A** and **B** from memory and then storing the result back to memory
 - ▶ Addresses of “**ADD A, B**”, “**A**”, and “**B**” may all be in three different pages
 - Multiple page fault per instruction results in unacceptable performance
 - ▶ Very unlikely, fortunately, due to **locality of reference**; see Slide-51
- Hardware support needed for demand paging; **same as hardware for paging and swapping**
 - **Page table** with valid / invalid bit, or special protection bits
 - **Secondary memory**: swap device with **swap space**; for not in-memory pages
 - **Instruction restart**; ability to restart any instruction after a page fault





Performance of Demand Paging

■ What is the Effective Access Time in demand paging? (worst case number of steps)

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
 1. Wait in a queue for this device until the read request is serviced
 2. Wait for the device seek and/or latency time
 3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user (**CPU scheduling, optional**)
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user (**if Step-6 is executed**)
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

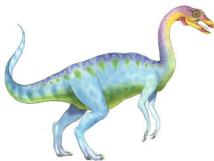




Performance of Demand Paging

- Not all steps (in Slide-18) are necessary in every case; e.g., Step-6
- Three major components of the page-fault service-time
 1. Service the interrupt; between 1 to 100 microseconds
 - ▶ Careful coding of the ISR means just several hundred instructions needed
 2. Read in the page – lots of time; at least 8 milliseconds + time in device-queue + ...
 3. Restart the process; between 1 to 100 microseconds – again, careful coding...
- Page Fault Rate $0 \leq p \leq 1$; p = probability of a page-fault and we want $p \approx 0$
 - if $p = 0$ then there is no page faults
 - if $p = 1$ then every memory reference causes a page-fault
- Effective Access Time (EAT)
 - $EAT = [(1 - p) \times \text{memory_access_time}] + [p \times \text{page_fault_time}]$
 - ▶ page_fault_time = page fault overhead + swap page out + swap page in





Demand Paging Example

- Memory access time = 200 nanoseconds; **between 10 to 200ns in most computers**
- Average page-fault service time = 8 milliseconds
- $$\begin{aligned} \text{EAT} &= [(1 - p) \times (200 \text{ nanoseconds})] + [p \times (8 \text{ milliseconds})] \\ &= [(1 - p) \times 200] + [p \times 8,000,000] \text{ nanoseconds} \\ &= 200 + 7,999,800p \text{ nanoseconds}; \text{ thus, EAT is directly proportional to } p \end{aligned}$$
- If one access out of 1,000 causes a page fault, then
 - $\text{EAT} = 8,199.8 \text{ nanoseconds} = 8.2 \text{ microseconds.}$
 - ▶ This is a slowdown by a factor of 40%!!; **Because of demand paging**
- If we want performance degradation < 10 percent
 - $200 + 7,999,800p < 220$
 $7,999,800p < 20$
 - Thus, we must have $p < .0000025$
 - **That is, to keep slowdown to <10% due to demand paging**
 - ▶ $p <$ one page fault in every 399,990 memory accesses





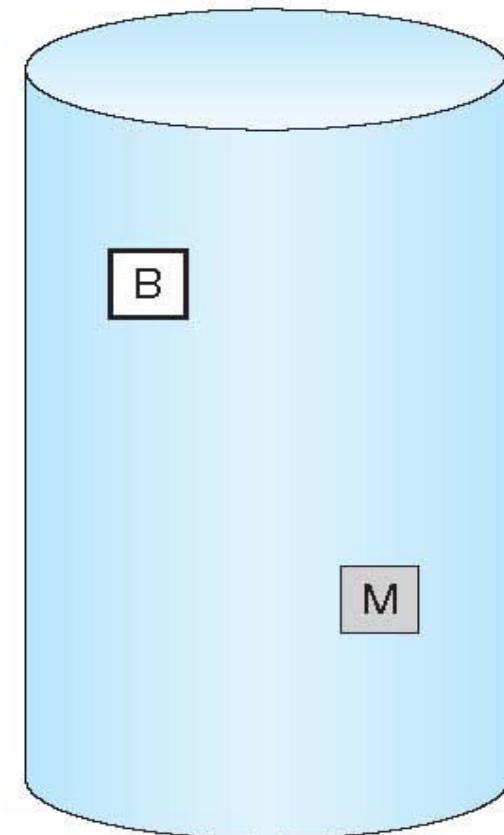
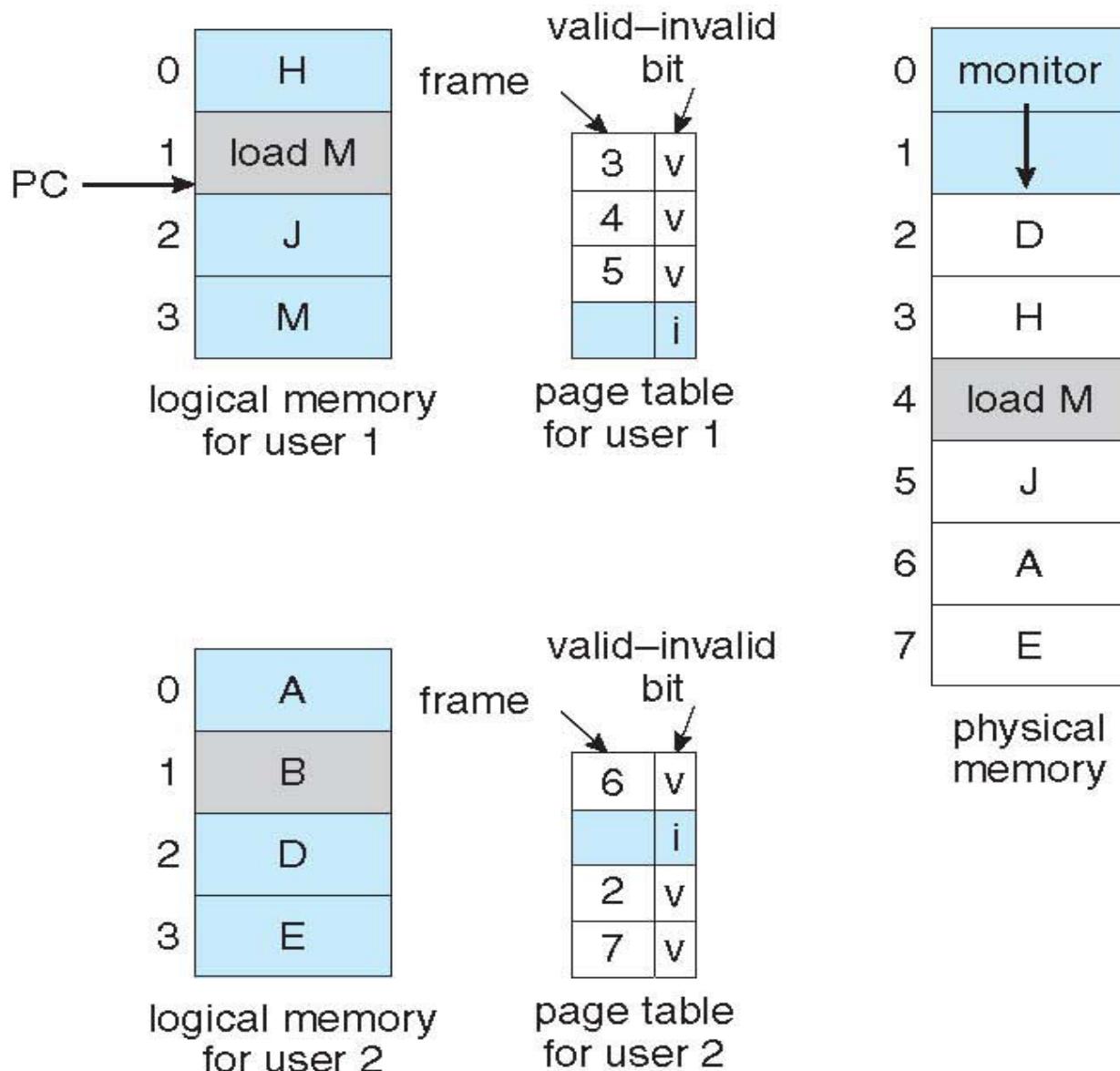
What Happens if There is no Free Frame?

- Many pages need to be loaded but not enough free frames available for them
 - Memory is being used up by process pages; **user processes**
 - Also memory is in demand from the kernel, I/O buffers, etc; **kernel processes**
- How much memory to allocate to both user and kernel processes or data?
- **Solution:** Page replacement; when paging in pages of a process but no free frames
 - Terminate the process? **Big fat no**
 - Swap out some process? **Yes**, but not always a good option
 - Find currently un-used frame to free it; **Page it out and page in process page**
 - ▶ **Replacing the un-used memory page with the new page**
 - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times





Need For Page Replacement

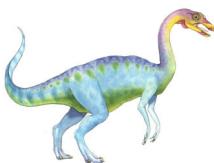




Basic Page Replacement Algorithm

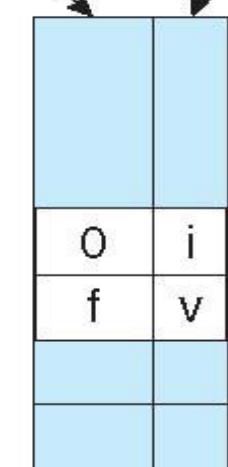
- The page-fault service routine is modified to include page replacement
 1. Find the location of the desired page on disk
 2. Find a free frame:
 1. If there is a free frame, use it
 2. If there is no free frame, use a page-replacement algorithm to select a **victim frame**
 3. Write the victim frame to the disk [**if dirty**]; change the page and the frame tables accordingly
 3. Read the desired page into the newly freed frame; change the page and frame tables
 4. Continue the user process from where the page fault occurred
- We have potentially **two** page transfers to do – increasing EAT
 - Only if no frames are free; **one page in required and one page out required**



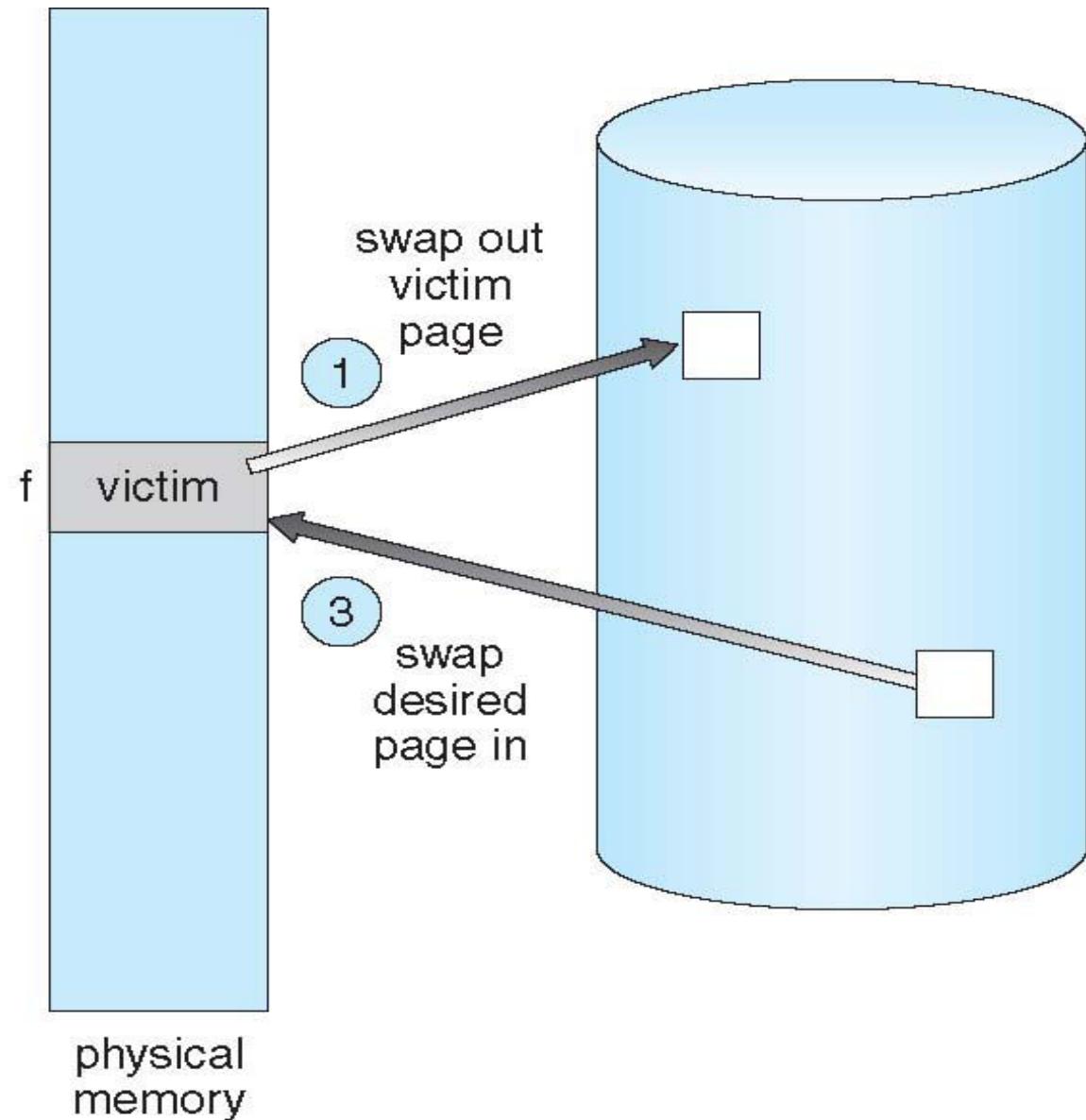


Page Replacement

frame valid-invalid bit



- 2 change to invalid
- 4 reset page table for new page

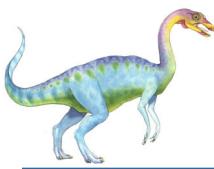




Page Replacement ...

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk ; **see Slide-28**
 - Each page or frame is associated with a **modify bit**
 - Set by the hardware whenever a page is modified
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory
 - **A user process of 20 pages can be executed in 10 frames** simply by using demand-paging and using a page-replacement algorithm to find a free frame whenever necessary





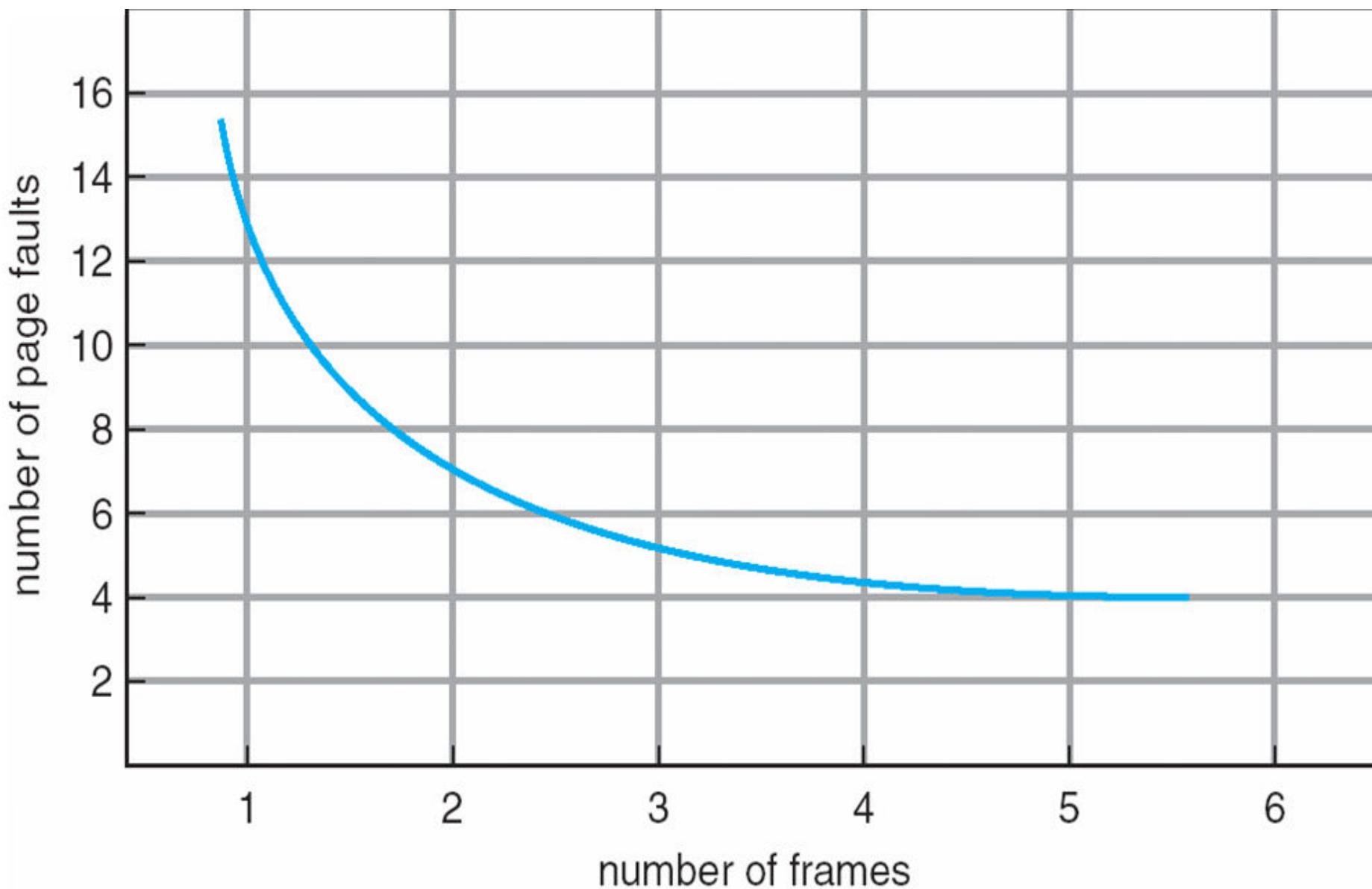
Page- Replacement and Frame-Allocation Algorithms

- Two major demand-paging problems : **frame allocation** and **page replacement**
- **Frame-allocation algorithm** determines
 - How many frames to allocate to each process
 - Which frames to replace **when page replacement is required**
- **Page-replacement algorithm**
 - We want an algorithm which yields the lowest page-fault rate
- Evaluate an algorithm by running it on a particular string of memory references (the **reference string**) and computing the number of page faults on that string
 - String is just page numbers **p**, not full addresses
 - Repeated access to the same page does not cause a page fault
 - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1 for a memory with three frames





Graph of Page Faults Versus The Number of Frames



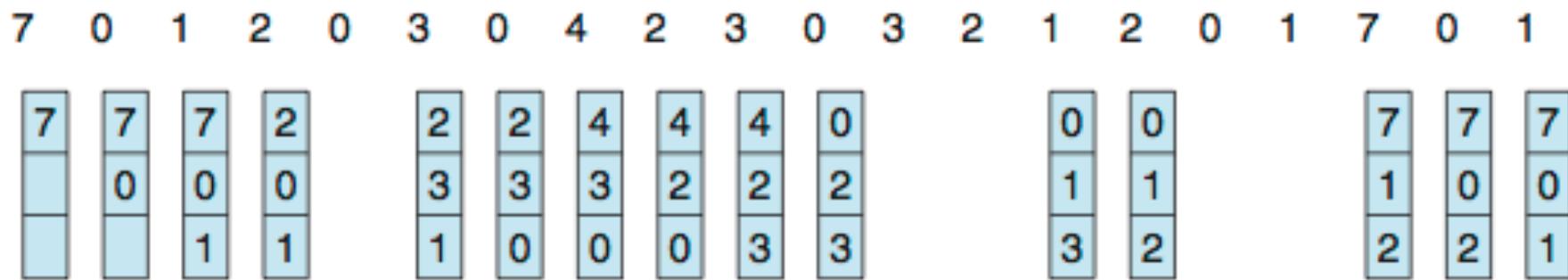


FIFO Page Replacement Algorithm

[Reference string = 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1 and Memory = 3 frames]

- Each page brought into memory is also inserted into a **first-in first-out queue**
 - Page to be replaced is the **oldest** page; the one at the head of the queue
- Our example yields 15 page faults

reference string



page frames

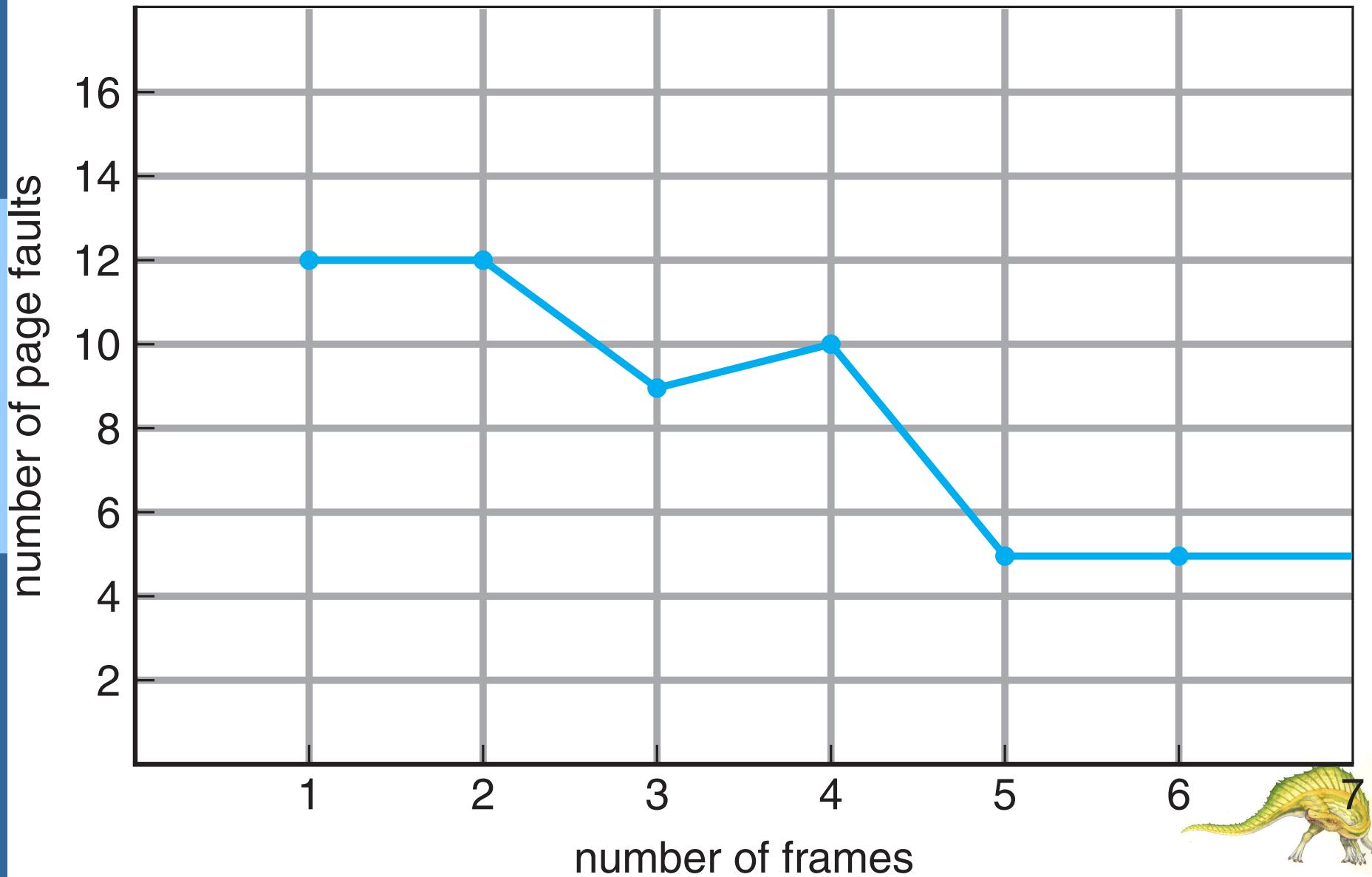
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
 - Adding more frames can cause more page faults!

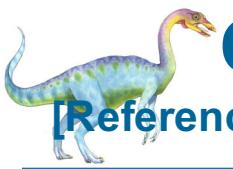
► **Belady's Anomaly**





FIFO Illustrating Belady's Anomaly





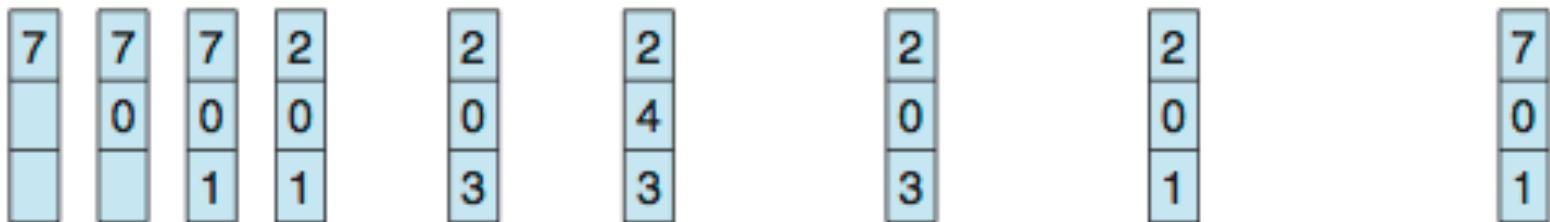
Optimal Page Replacement Algorithm

[Reference string = 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1 and Memory = 3 frames]

- Replace the page that will not be used for longest period of time
- Our example yields 9 page faults
- Unfortunately, OPR is **not feasible** to implement
 - Because: we can't know the future; i.e., what is the next page?
 - ▶ We have assumed that we know the reference string. No, we don't
- OPR is used only for comparing with new algorithms; **how close to the optimal?**

reference string

7 0 1 2 0 3 0 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1



page frames



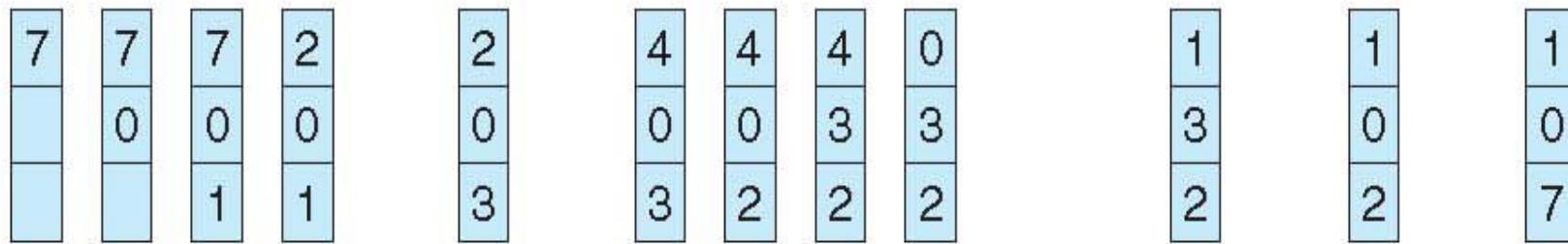
LRU Page Replacement Algorithm

[Reference string = 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1 and Memory = 3 frames]

- Use the recent past as an approximation of the near future
- Replace the page that *has not been used* for the longest period of time
 - That is, the **least recently used** page
- Associate time of last use with each page

reference string

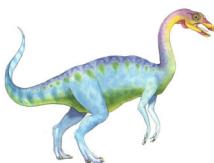
7 0 1 2 0 3 0 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1



page frames

- Our example yields 12 page faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- Algorithm is feasible but not easy to implement.
 - LRU algorithm may require substantial hardware support





LRU Algorithm

■ Counter implementation

- Each page-table entry has a counter; every time the page is referenced through this entry, copy the current clock value into the counter
- When a page needs to be changed, look at the counters to find smallest value
 - ▶ Search through the page-table needed; **to find the LRU page**

■ Stack implementation

- Keep a stack of page numbers in a double link form, **with head and tail pointers**:
- Whenever a page is referenced:
 - ▶ move it to the top; **most recently used page is always at the top of stack**
 - ▶ requires 6 pointers to be changed
- But each update more expensive
- No search for replacement; **as LRU page is always at the bottom of the stack**

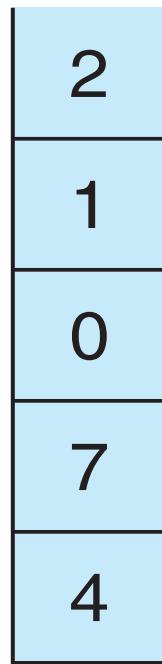
■ LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly



Use of A Stack to Record Most Recent Page References

reference string

4 7 0 7 1 0 1 2 1 2 7 1 2

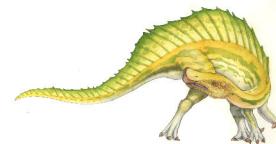


stack
before
a



stack
after
b

a b



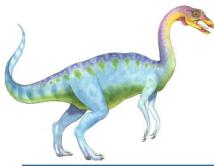


Counting-Based Page Replacement Algorithms

[Reference string = 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1 and Memory = 3 frames]

- Keep a counter of the number of references that have been made to each page
 - ▶ Not common
 - **Least Frequently Used (LFU) Algorithm:** replaces the page with the smallest count
 - ▶ An actively used page should have a large count value
 - But... Pages may be heavily used initially and never used again
 - **Most Frequently Used (MFU) Algorithm:** based on the argument that the page with the smallest count was probably just brought in and has yet to be used
- Counting-based algorithms are very expensive to implement, and they do not approximate OPT replacement well





Allocation of Frames

- Each process needs to be allocated a **minimum** number of frames
 - Number of page-faults increases as the # of allocated frames decreases
 - The minimum number of frames is defined by the computer architecture
 - ▶ Example: IBM 370 – references 6 pages to handle SS MOVE instruction:
 - instruction is 6 bytes, might span 2 pages
 - 2 pages to handle *from*
 - 2 pages to handle *to*
- **Maximum** of course is the total number of frames in the system
- Two major allocation schemes
 - fixed allocation
 - priority allocation
- Many variations





Fixed Allocation

- **Equal allocation** – For example: after allocating frames to the OS, and if there are 100 free frames and 5 processes, then allocate 20 frames to each process.
 - Keep the leftover free frames as free-frame buffer pool
- **Proportional allocation** – Allocate free frames according to the size of process
 - Dynamic, as the degree of multiprogramming and the process sizes change

s_i = size of process p_i

$S = s_i$

m = total number of frames

a_i = allocation for p_i = $\frac{s_i}{S} \cdot m$

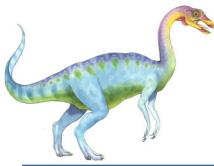
$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

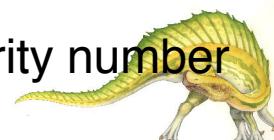
$$a_1 = \frac{10}{137} \times 62 \approx 4$$

$$a_2 = \frac{127}{137} \times 62 \approx 57$$



Priority Allocation

- Use a proportional allocation scheme using priorities rather than size
 - Ratio of frames depends on the priorities of processes, or, on a combination of both their priorities and their sizes
 - We may want to allocate more frames to a high-priority process, in order to speed up its execution, to the detriment of low-priority processes
- In this case, the replacement algorithm is modified to consider process's priorities
 - If high-priority process P_i generates a page fault,
 - ▶ select for replacement one of its frames
 - ▶ select for replacement a frame from a process with lower priority number





Global vs. Local Allocation

- **Global replacement** – allows a process to select a replacement frame from the set of **all** frames; that is, one process can take a frame from another
 - But then process execution time can vary greatly
 - But greater throughput, so more common

- **Local replacement** – requires that a process selects from only its own set of allocated frames
 - More consistent per-process performance
 - But possibly underutilized memory

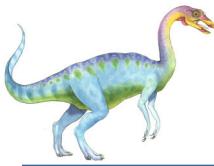




Non-Uniform Memory Access

- So far we have assumed that all main memory is accessed equally
- Many systems are **NUMA** – speed of access to memory varies
 - Consider system boards containing CPUs and memory, interconnected over a system bus or a high-speed network
- Optimal performance comes from allocating memory frames “*as close as possible to*” the CPU on which the process or thread is running or scheduled
 - And modifying the scheduler to schedule the thread on the same system board when possible; thus, **taking NUMA into account**
 - **Thread issues** solved by Solaris by creating **Igroups**; that is, **latency groups**
 - ▶ Each Igroup gathers together close CPUs and memories
 - ▶ Tries to schedule all threads of a process and allocate all memory of a process within an Igroup
 - If not possible, then pick a nearby Igroup





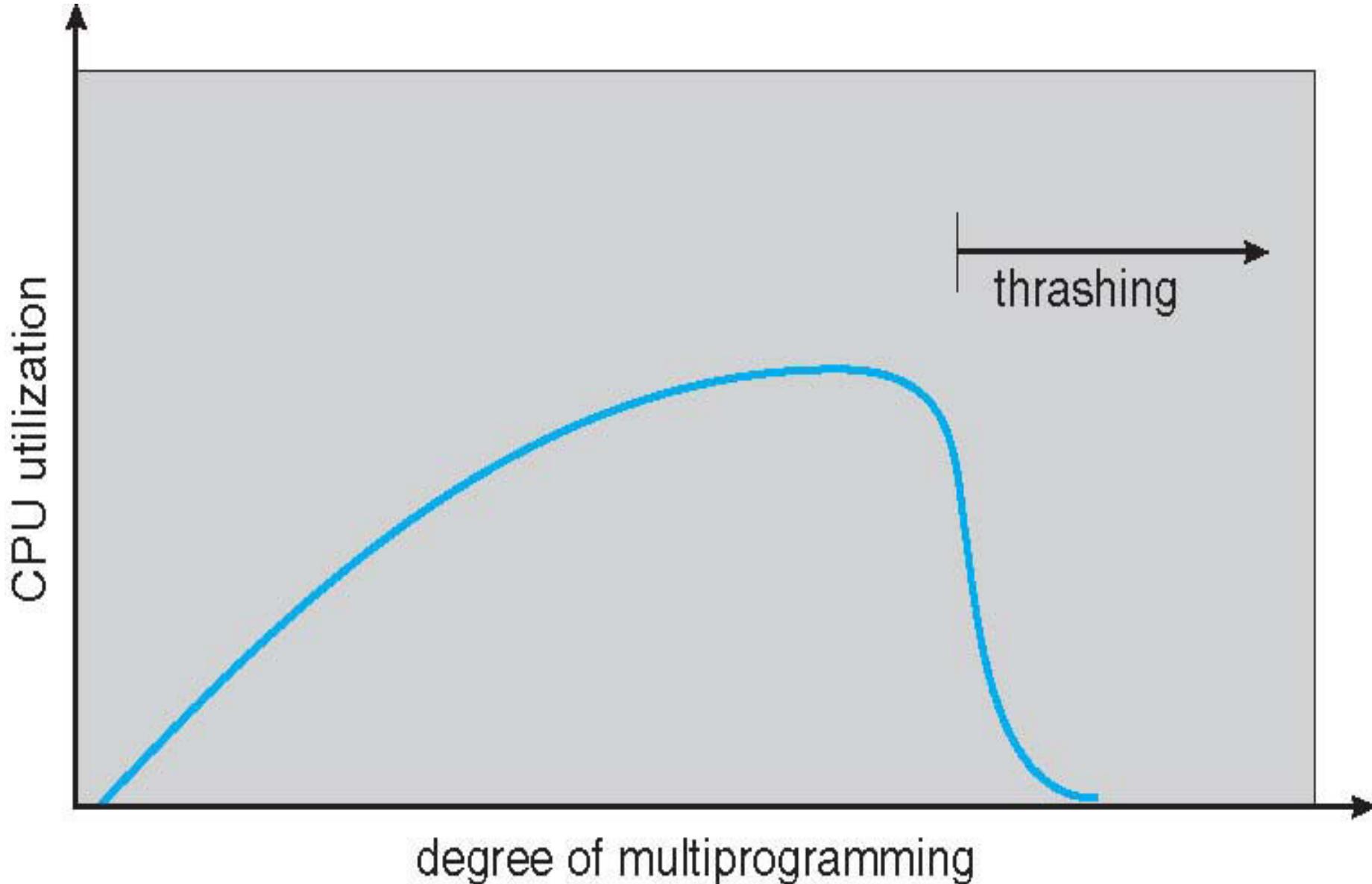
Thrashing

- If a process does not have “enough” frames, the page-fault rate is very high
 - Too many swapping, i.e. page-in’s and page-out’s
 - ▶ Page fault to get the page; **page-in**
 - ▶ Replace an existing page; **page-out**
 - ▶ But quickly need replaced pageback due to another page-fault
 - This leads to severe performance problems:
 - ▶ Low CPU utilization due to page-in’s and page-out’s
 - ▶ Then, OS thinks that it needs to increase the degree of multiprogramming
 - ▶ And then, another **low-frames** process added to the system
 - ▶ More low-frames processes, more page-faults, lower CPU utilization
- **Thrashing** ≡ process is busy swapping pages in and out
 - Instead of utilizing the CPU, processes spend their time in paging





Thrashing (Cont.)





Demand Paging and Thrashing

■ Why does demand paging work? **Locality model**

- Process migrates from one locality to another as it executes
 - ▶ Locality = set of pages that are actively used together
 - ▶ Localities are defined by a program structure and its data structure
- Localities may overlap and a program consists of many different localities

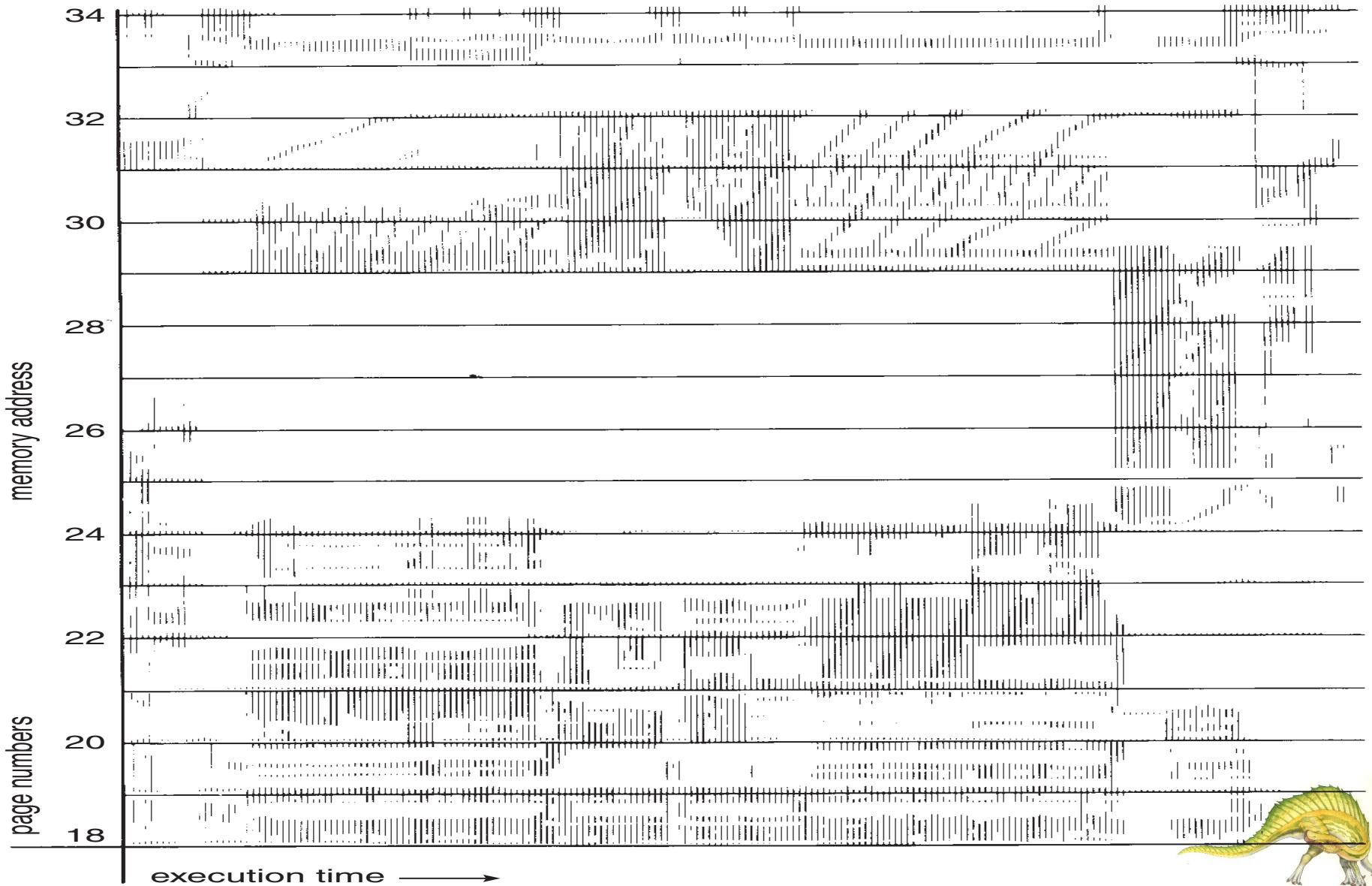
■ Why does thrashing occur? \sum size of locality > total **frame allocation** size

- Limit effects by using local or priority page replacement
 - ▶ Local replacement: if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well
 - Does not solve the problem entirely though
- To prevent thrashing, we must provide a process with as many frames as it needs
 - ▶ How to know its frame needs? (many techniques)
 - *Working-set strategy*: starts by looking at the number of frames a process is actually using





Locality In A Memory-Reference Pattern



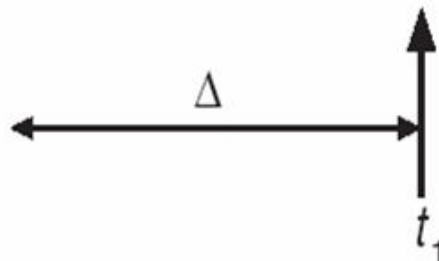


Working-Set Model

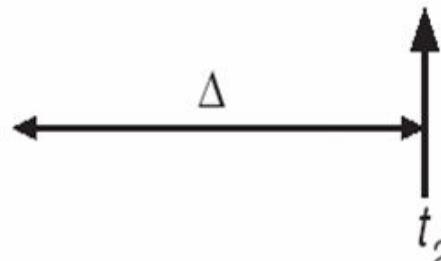
- $\Delta \equiv$ working-set window \equiv a fixed number of page references. Example: 10,000 instructions
- WSS_i (working-set of P_i) = total # of pages referenced in the most recent Δ (varies in time)
 - if Δ too small will not encompass entire locality
 - if Δ too large will encompass several localities
 - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \sum WSS_i \equiv$ total demand for frames; is an approximation of a program's locality
- if $D > m \Rightarrow$ Thrashing; where $m = \#$ of available frames
- Policy if $D > m$, then suspend or swap out one of the processes
 - OS monitors each process's working-set and accordingly allocate enough frames

page reference table

... 2 6 1 5 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



$$WS(t_1) = \{1, 2, 5, 6, 7\}$$



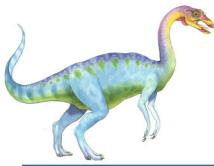
$$WS(t_2) = \{3, 4\}$$



Keeping Track of the Working Set

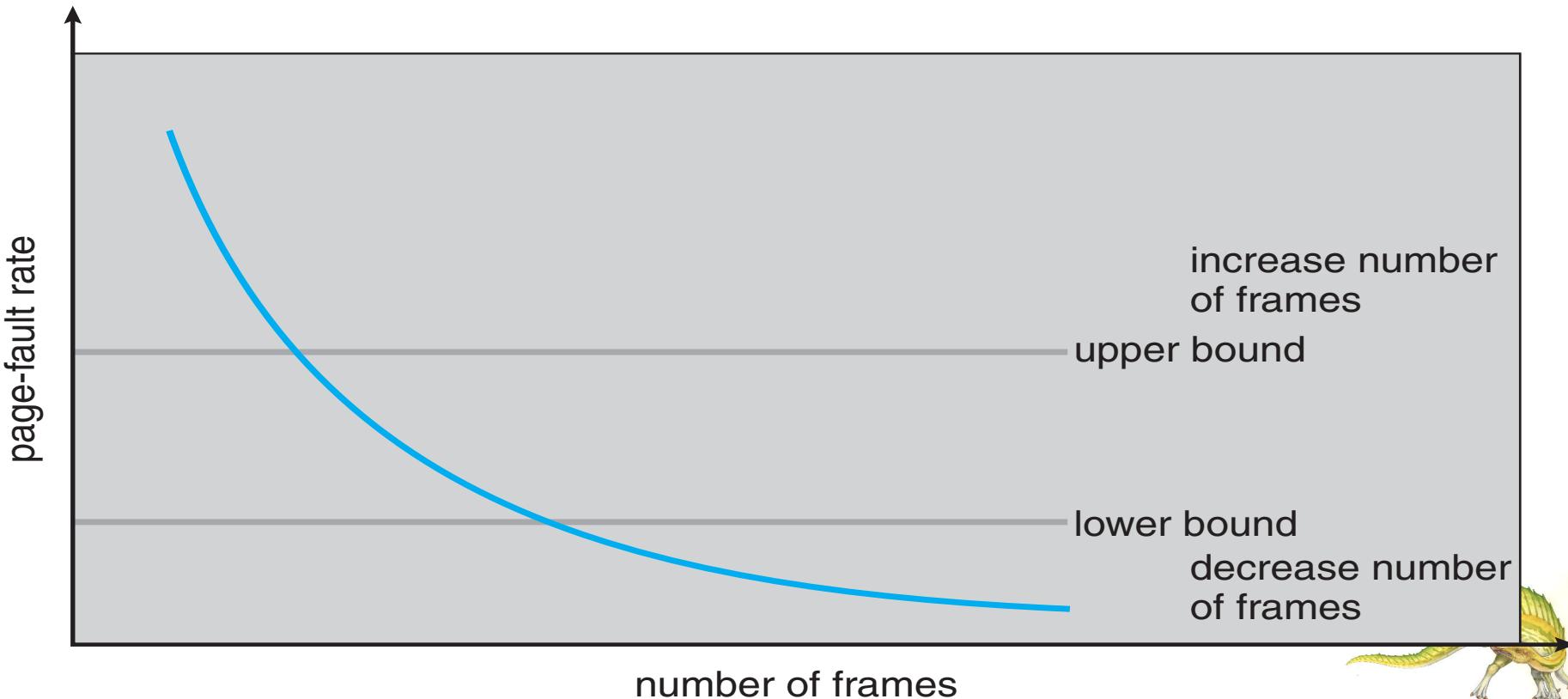
- OS can approximate with a fixed-interval timer interrupt + a reference bit
- Example: $\Delta = 10,000$ references
 - Timer interrupts after every 5000 time units
 - Keep in memory 2 bits for each page
 - Whenever a timer interrupts, copy and clear reference-bit values to 0
 - Thus, if page-fault occurs
 - ▶ Examine current reference-bit and two in-memory bits to check if a page was used within the last 10,000 to 15,000 references
 - ▶ If one of the bits in memory = 1 \Rightarrow page in working set
- Not completely accurate, we cannot where a reference occurred in interval 5,000
- Improvement = 10 bits and interrupt every 1000 time units
 - However, cost of ISR is higher due to frequent interrupts





Page-Fault Frequency

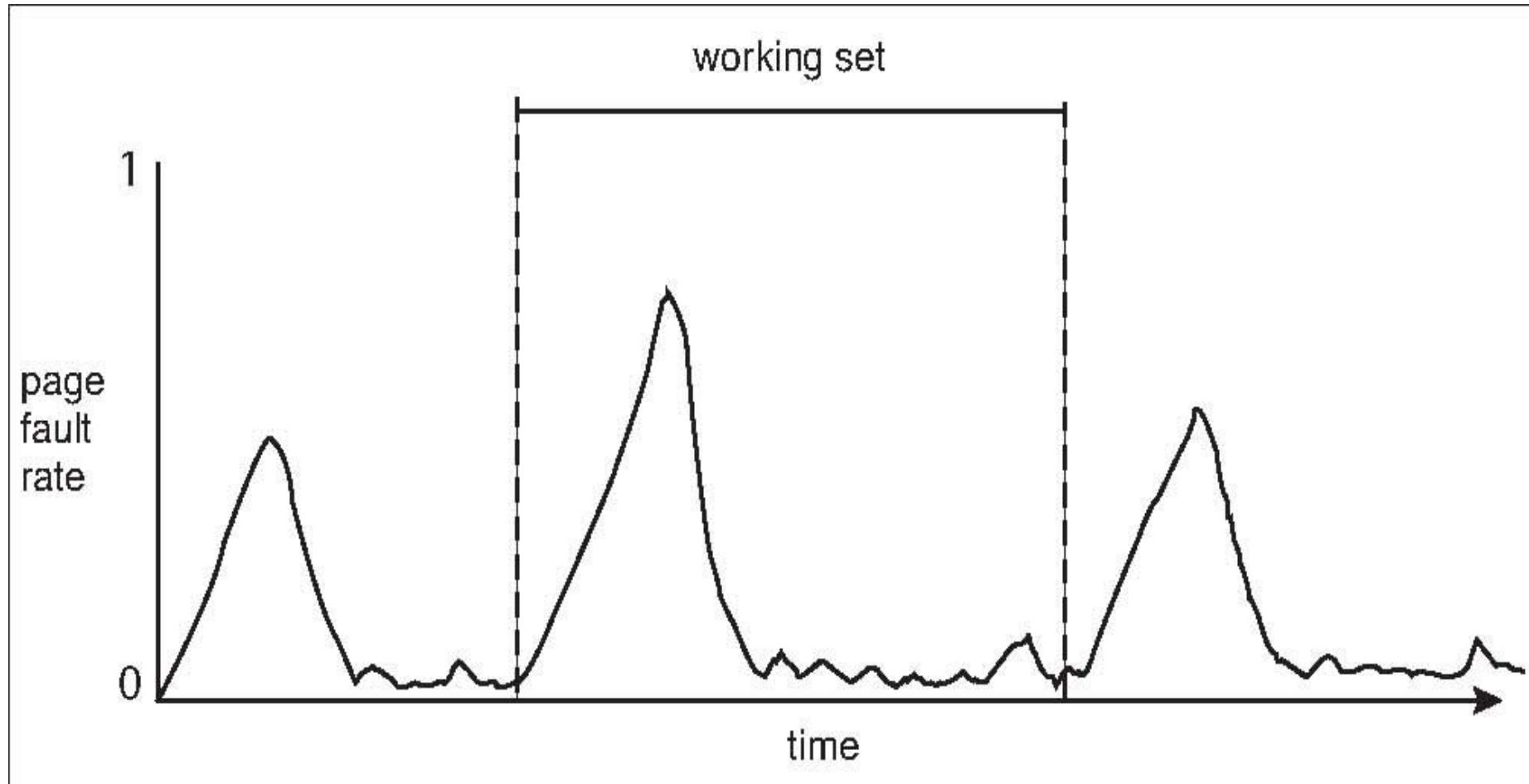
- More direct approach than WSS; we want to control the page-fault rate
- To prevent thrashing (which has a high page-fault rate), establish “acceptable” **page-fault frequency (PFF)** rate and use local replacement policy
 - If actual rate too low, process loses frame; it has too many frames
 - If actual rate too high, process gains frame; it needs more frames





Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time

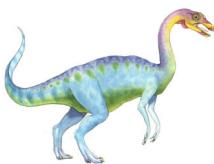




Operating System Examples

- Windows
- Solaris





Windows

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum





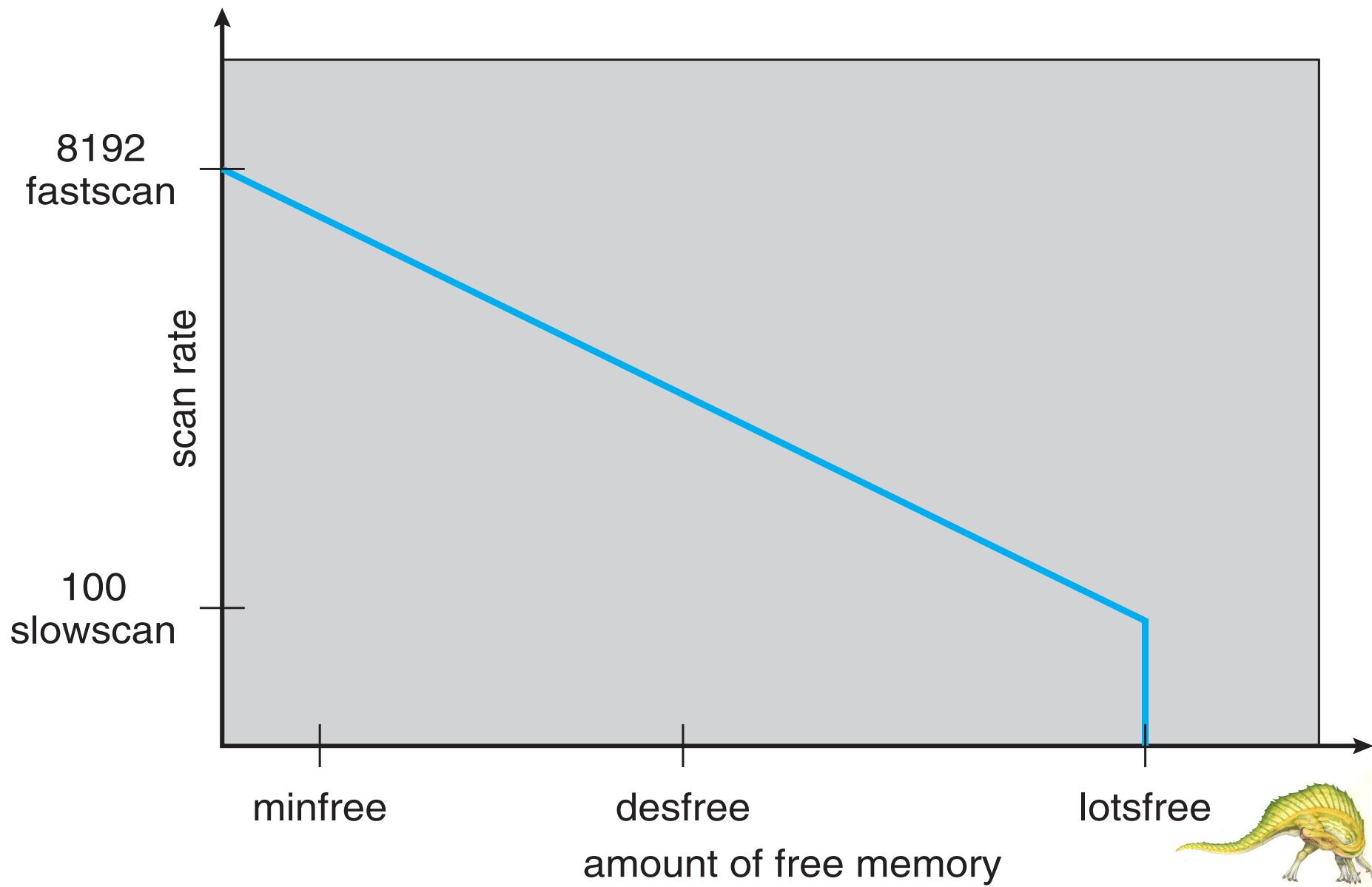
Solaris

- Maintains a list of free pages to assign faulting processes
- **Lotsfree** – threshold parameter (amount of free memory) to begin paging
- **Desfree** – threshold parameter to increasing paging
- **Minfree** – threshold parameter to begin swapping
- Paging is performed by **pageout** process
- **Pageout** scans pages using modified clock algorithm
- **Scanrate** is the rate at which pages are scanned. This ranges from **slowscan** to **fastscan**
- **Pageout** is called more frequently depending upon the amount of free memory available
- **Priority paging** gives priority to process code pages

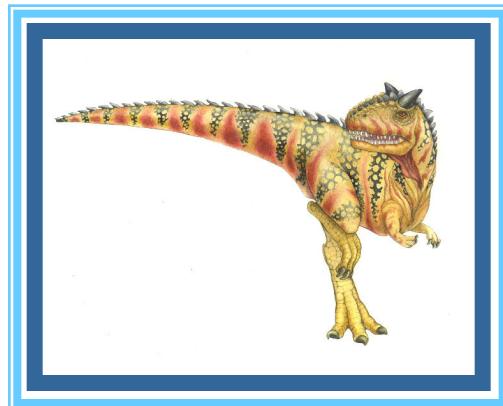




Solaris 2 Page Scanner



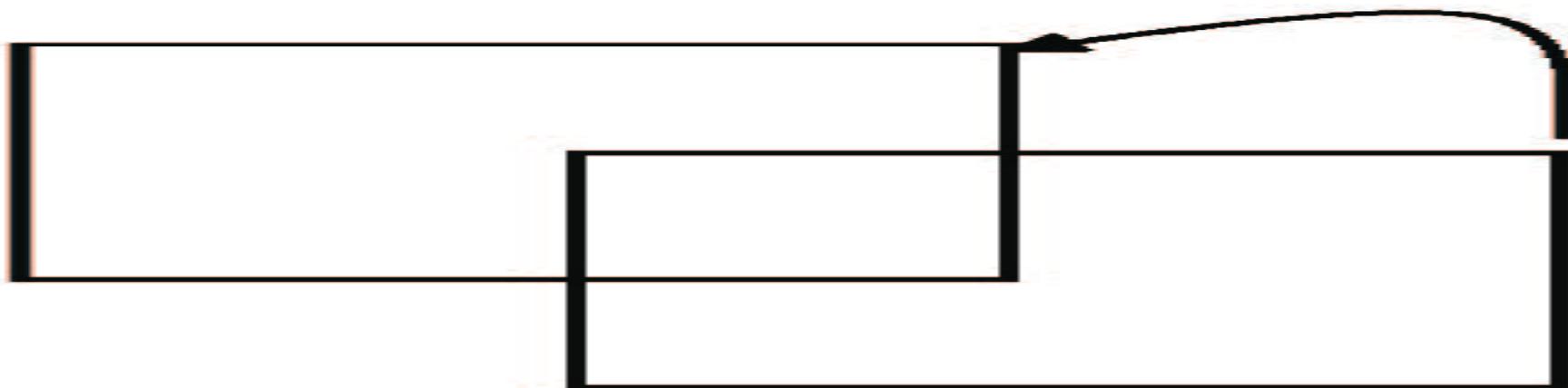
End of Chapter 9





Instruction Restart

- Consider an instruction that could access several different locations
 - block move



- auto increment/decrement location
- Restart the whole operation?
 - ▶ What if source and destination overlap?

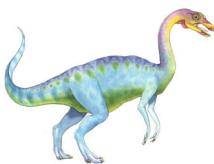




Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
 - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
 - Then page in and out of swap space
 - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
 - Used in Solaris and current BSD
 - Still need to write to swap space
 - ▶ Pages not associated with a file (like stack and heap) – **anonymous memory**
 - ▶ Pages modified in memory but not yet written back to the file system
- Mobile systems
 - Typically don't support swapping
 - Instead, demand page from file system and reclaim read-only pages (such as code)





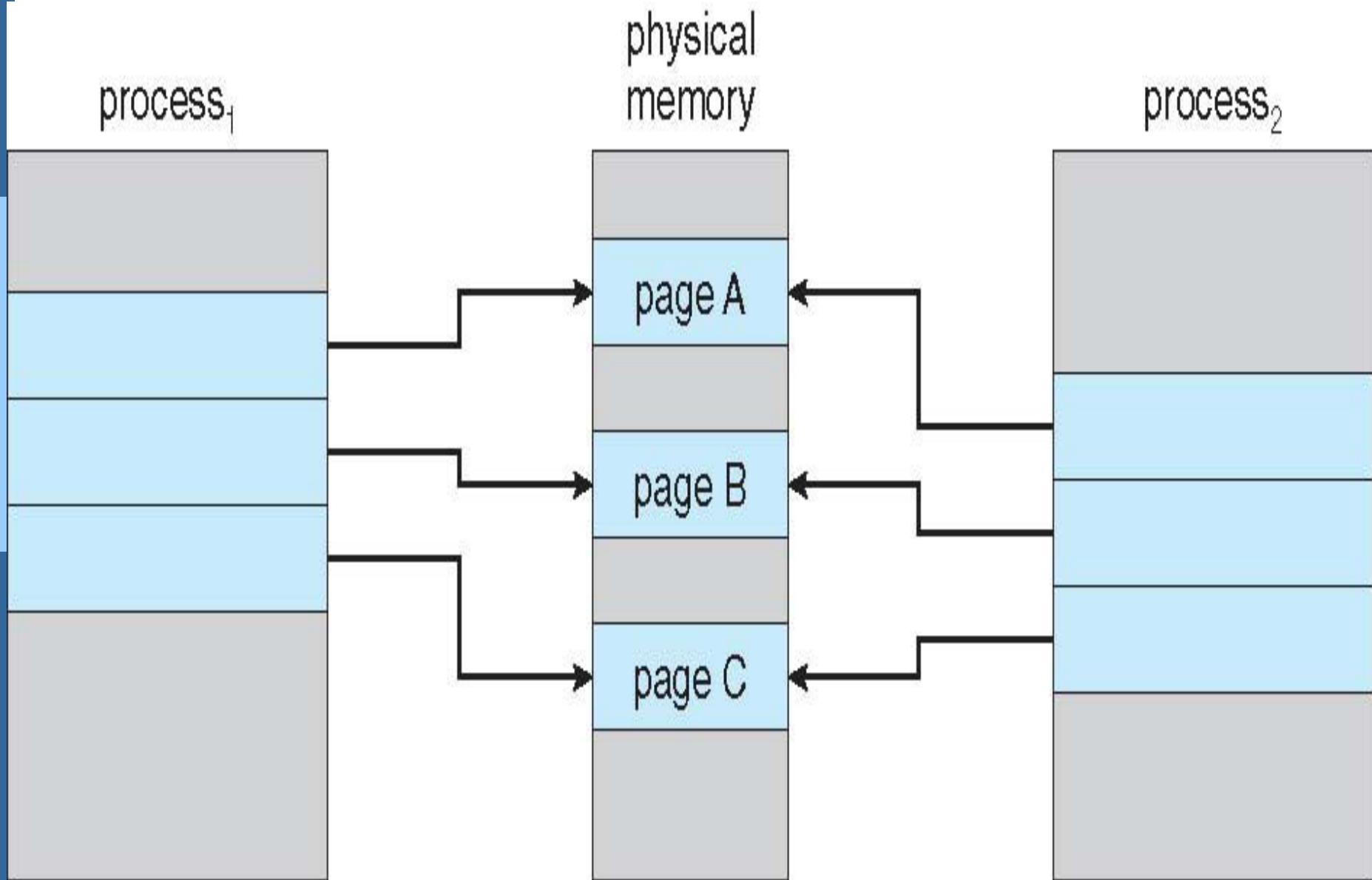
Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially **share** the same pages in memory
 - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
 - Pool should always have free frames for fast demand page execution
 - ▶ Don't want to have to free a frame as well as other processing on page fault
 - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system call has parent suspended and child using copy-on-write address space of parent
 - Designed to have child call `exec()`
 - Very efficient



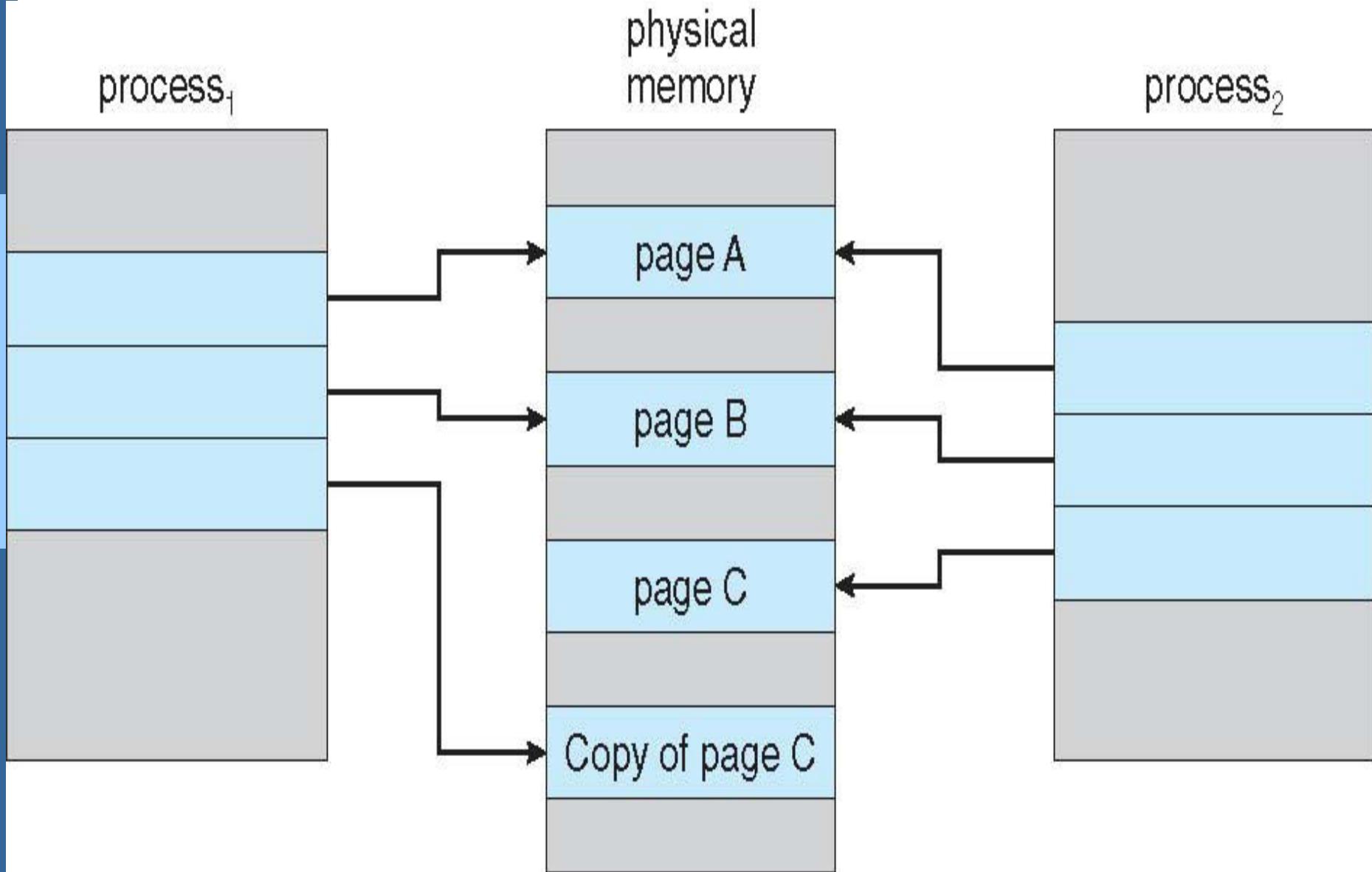


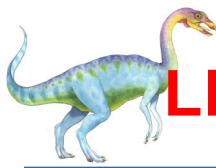
Before Process 1 Modifies Page C





After Process 1 Modifies Page C





LRU-Approximation Page Replacement Algorithms

■ LRU needs special hardware and still slow

■ Reference bit

- With each page associate a bit, initially = 0
- When page is referenced bit set to 1
- Replace any with reference bit = 0 (if one exists)
 - ▶ We do not know the order, however

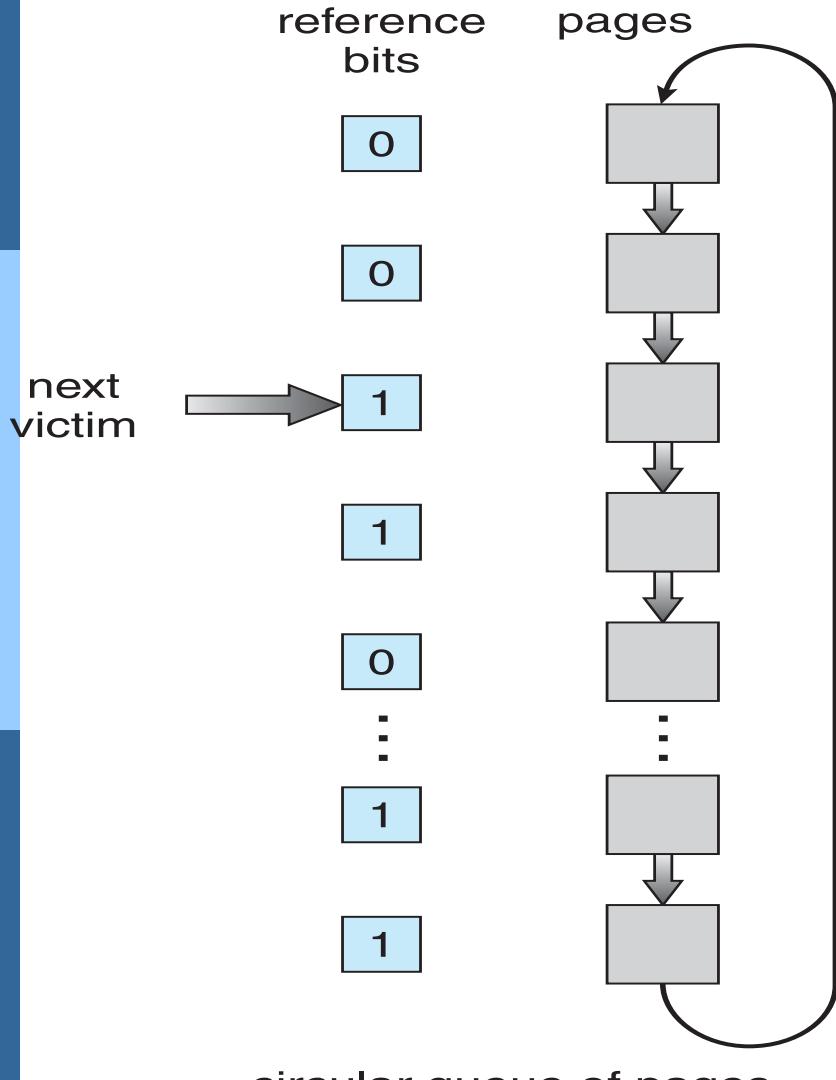
■ Second-chance algorithm

- Generally FIFO, plus hardware-provided reference bit
- **Clock** replacement
- If page to be replaced has
 - ▶ Reference bit = 0 -> replace it
 - ▶ reference bit = 1 then:
 - set reference bit 0, leave page in memory
 - replace next page, subject to same rules

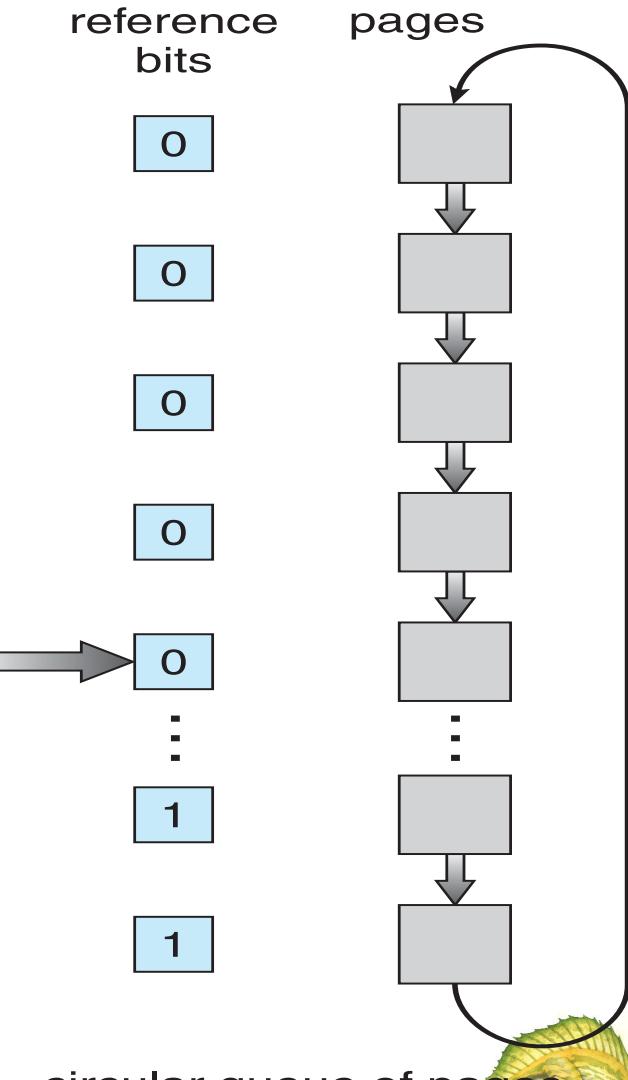




Second-Chance (clock) Page-Replacement Algorithm



(a)



(b)



Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify)
 1. (0, 0) neither recently used nor modified – best page to replace
 2. (0, 1) not recently used but modified – not quite as good, must write out before replacement
 3. (1, 0) recently used but clean – probably will be used again soon
 4. (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
 - Might need to search circular queue several times





Page-Buffering Algorithms

- Keep a pool of free frames, always
 - Then frame available when needed, not found at fault time
 - Read page into free frame and select victim to evict and add to free pool
 - When convenient, evict victim
- Possibly, keep list of modified pages
 - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
 - If referenced again before reused, no need to load contents again from disk
 - Generally useful to reduce penalty if wrong victim frame selected





Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
 - OS keeps copy of page in memory as I/O buffer
 - Application keeps page in memory for its own work
- Operating system can give direct access to the disk, getting out of the way of the applications
 - **Raw disk** mode
- Bypasses buffering, locking, etc





Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging
 - A page-sized portion of the file is read from the file system into a physical page
 - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than `read()` and `write()` system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
 - Periodically and / or at file `close()` time
 - For example, when the pager scans for dirty pages





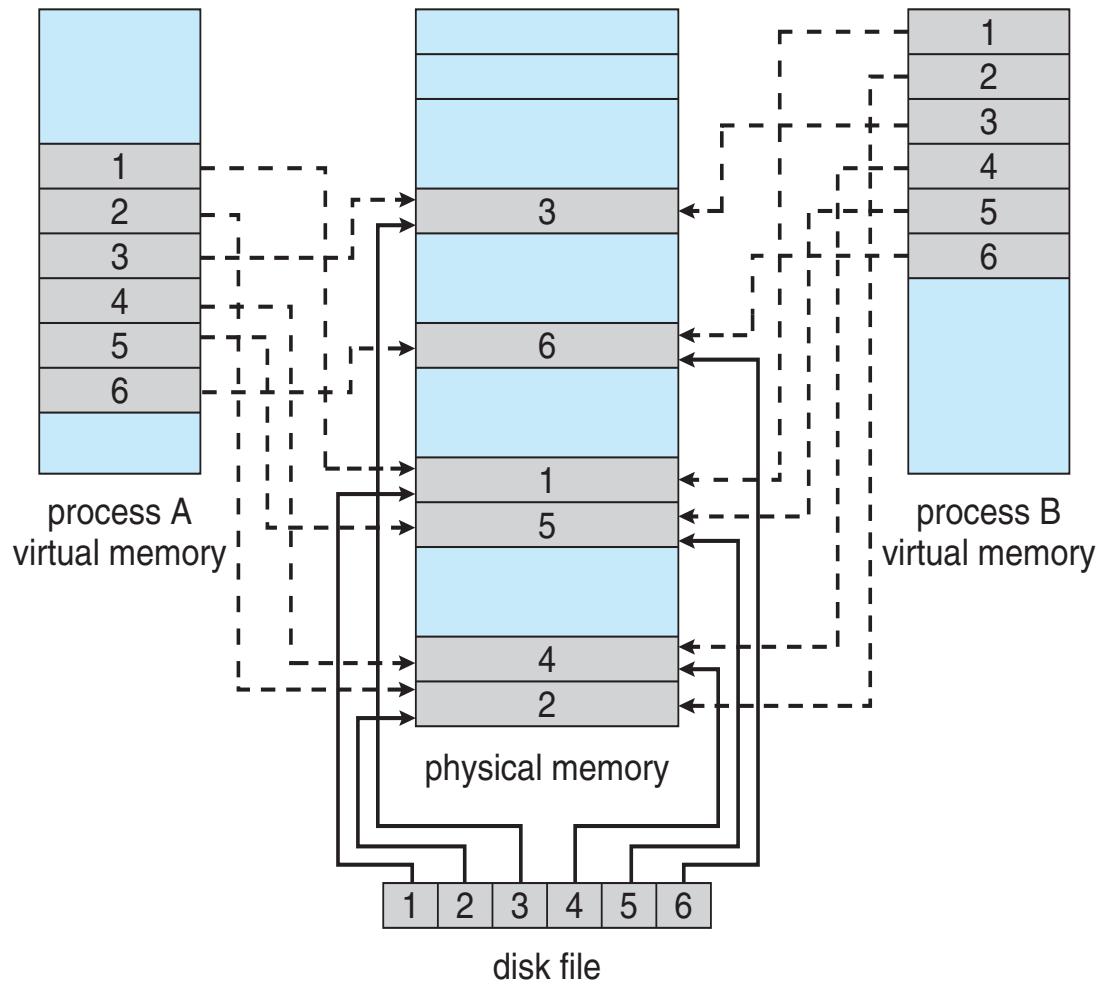
Memory-Mapped File Technique for all I/O

- Some OSes uses memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via `mmap()` system call
 - Now file mapped into process address space
- For standard I/O (`open()`, `read()`, `write()`, `close()`), `mmap` anyway
 - But map file into kernel address space
 - Process still does `read()` and `write()`
 - ▶ Copies data to and from kernel space and user space
 - Uses efficient memory management subsystem
 - ▶ Avoids needing separate subsystem
- COW can be used for read/write non-shared pages
- Memory mapped files can be used for shared memory (although again via separate system calls)



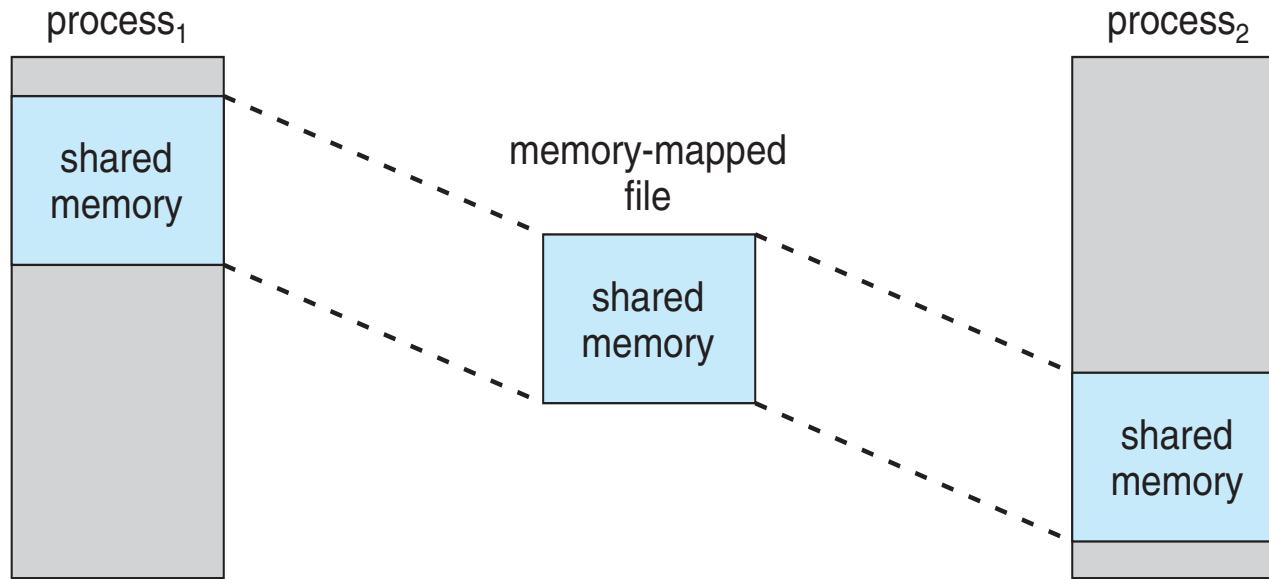


Memory Mapped Files





Shared Memory via Memory-Mapped I/O





Shared Memory in Windows API

- First create a **file mapping** for file to be mapped
 - Then establish a view of the mapped file in process's virtual address space
- Consider producer / consumer
 - Producer create shared-memory object using memory mapping features
 - Open file via `CreateFile()`, returning a `HANDLE`
 - Create mapping via `CreateFileMapping()` creating a **named shared-memory object**
 - Create view via `MapViewOfFile()`
- Sample code in Textbook

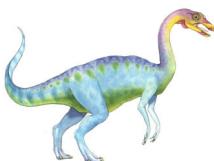




Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
 - Kernel requests memory for structures of varying sizes
 - Some kernel memory needs to be contiguous
 - ▶ I.e. for device I/O





Buddy System

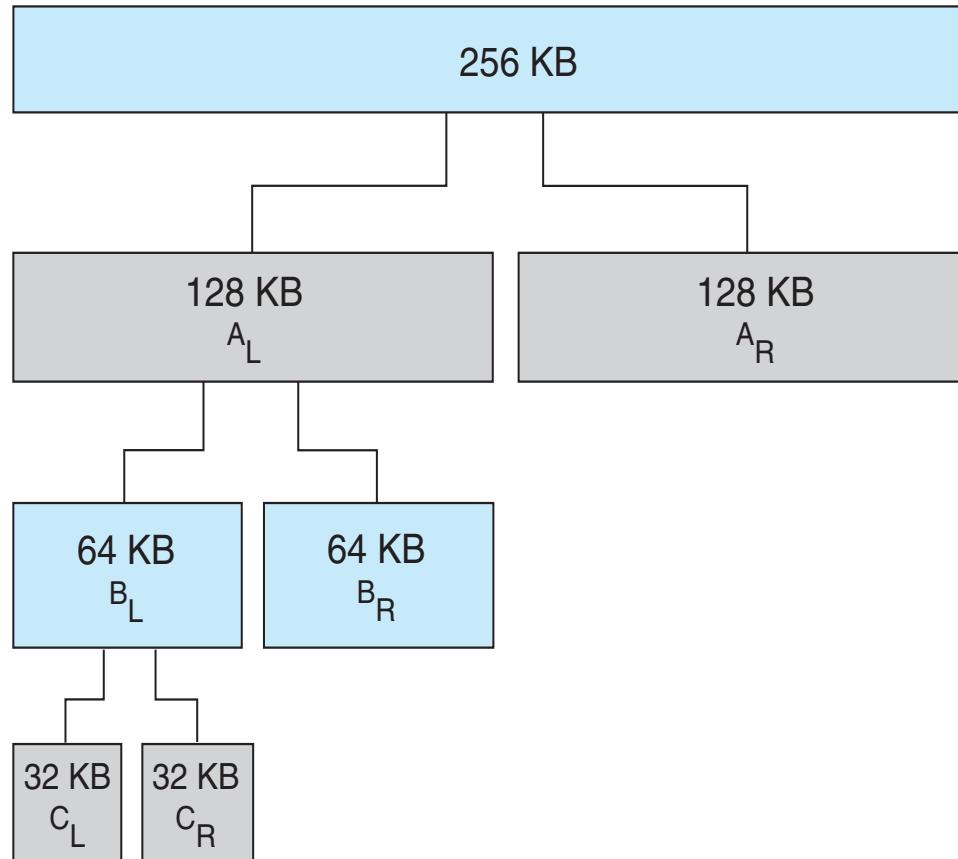
- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
 - Satisfies requests in units sized as power of 2
 - Request rounded up to next highest power of 2
 - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
 - ▶ Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB
 - Split into A_L and A_R of 128KB each
 - ▶ One further divided into B_L and B_R of 64KB
 - One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantage – quickly **coalesce** unused chunks into larger chunk
- Disadvantage - fragmentation





Buddy System Allocator

physically contiguous pages

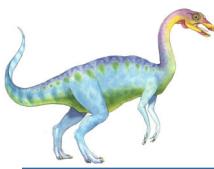




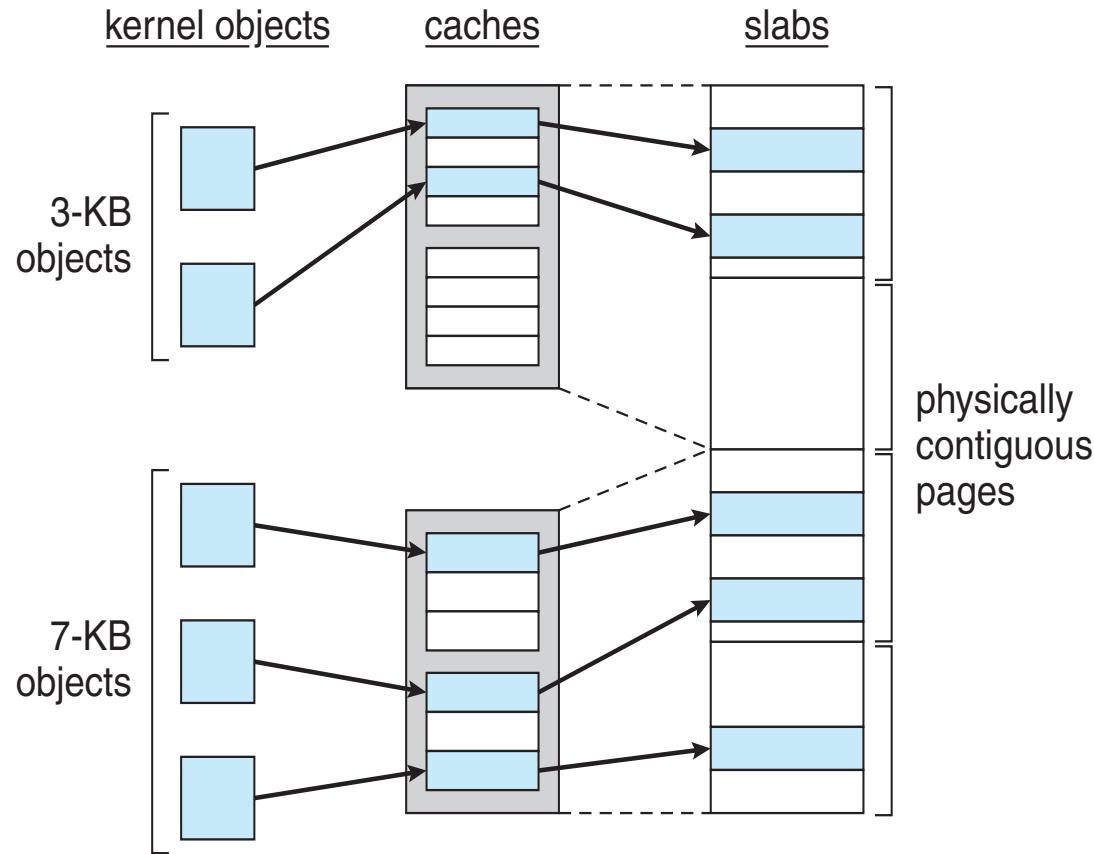
Slab Allocator

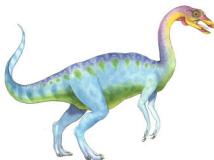
- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
 - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
 - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction





Slab Allocation





Slab Allocator in Linux

- For example process descriptor is of type `struct task_struct`
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
 - Will use existing free `struct task_struct`
- Slab can be in three possible states
 1. Full – all used
 2. Empty – all free
 3. Partial – mix of free and used
- Upon request, slab allocator
 1. Uses free struct in partial slab
 2. If none, takes one from empty slab
 3. If no empty slab, create new empty





Slab Allocator in Linux (Cont.)

- Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes
- Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
 - SLOB for systems with limited memory
 - ▶ Simple List of Blocks – maintains 3 list objects for small, medium, large objects
 - SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure





Other Considerations -- Prepaging

■ Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume s pages are prepaged and a of the pages is used
 - ▶ Is cost of $s * a$ save pages faults > or < than the cost of prepaging
 - ▶ $s * (1 - a)$ unnecessary pages?
 - ▶ a near zero \Rightarrow prepaging loses





Other Issues – Page Size

- Sometimes OS designers have a choice
 - Especially if running on custom-built CPU
- Page size selection must take into consideration:
 - Fragmentation
 - Page table size
 - **Resolution**
 - I/O overhead
 - Number of page faults
 - Locality
 - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time





Other Issues – TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- $\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$
- Ideally, the working set of each process is stored in the TLB
 - Otherwise there is a high degree of page faults
- Increase the Page Size
 - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
 - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation





Other Issues – Program Structure

■ Program structure

- int [128,128] data;
- Each row is stored in one page
- Program 1

```
for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i, j] = 0;
```

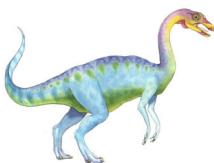
$128 \times 128 = 16,384$ page faults

- Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i, j] = 0;
```

128 page faults





Other Issues – I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- **Pinning** of pages to lock into memory

