

Course: Operating Systems



What Is an
OPERATING SYSTEM (OS)
and How Does It Work

CLEVERISM.COM

Lai NGUYEN

□ Research & Interests

- **Distributed systems:** Analysis, optimization, and control of systems with limited communication.
- **Communication networks and protocols:** Network architecture, routing algorithms, protocols, applications, and services. Network design, measurement, analysis, optimization, and management.

□ Areas of specialty

- Networked dynamic systems, distributed cooperative control, network routing, constrained communication protocols, water systems.



- Faculty of Computer Science and Engineering
 - *Block A3, Ho Chi Minh City University of Technology*
- lai@hcmut.edu.vn



Course Syllabus

Credits	3			Code	CO2017
Credits Hours	Total: 60	Lecture: 30			Lab: 22 Assignment: 8
Evaluation	Exercise: 10%	Lab: 10%	Midterm:	Assignment: 30%	Final exam: 50%
Assessment method	Final exam: <i>Multiple choice questions</i> , ~ 90 minutes				
Prerequisites					
Co-requisites					
Undergraduate Programs	<i>Computer Science and Computer Engineering</i>				
Website	http://e-learning.hcmut.edu.vn/				



Course Outcomes

- Understanding of the fundamental concepts in modern Operating Systems (OSs): (1) *Processes and Threads: CPU scheduling, synchronization*; (2) *Memory management, main and virtual memory*; (3) *I/O management*; (4) *Persistent data storage and File systems, journaling*; (5) *Security and protection*.
- Understanding of *multi-programming* and *synchronization* in such programming models.
- Reasoning about *system performance*, applying the lessons of *algorithms* and *data structures* to the complex operations of an operating system.
- Practicing and performing *simulation experiments* (using programming language C, Java or Python)
- Presenting materials on *group projects*



Course Outline

- Introduction to Operating systems
- Processes/Threads management
 - CPU scheduling
 - Synchronization
- Memory management
 - Main memory
 - Virtual memory
- I/O management
- Storage management
 - File systems
- Security and protection
- Advanced topics
- Summary



Textbook and References

- [1] “Operating System Concepts”, [Abraham Silberschatz](#), [Greg Gagne](#), [Peter B. Galvin](#), 10th Edition, John Wiley & Sons, 2018.
ISBN1119439256, 9781119439257, 976 pages.
- [2] “Modern Operating Systems”, [Andrew S Tanenbaum](#), [Herbert Bos](#), 4th Edition, Pearson Education Limited, 2015.
ISBN1292061952, 9781292061955, 1136 pages.
- [3] “Operating Systems: Internals and Design Principles”, [William Stallings](#), 8th Edition, Pearson Education Limited, 2014.
ISBN1292061944, 9781292061948, 800 pages.
- [4] “Operating Systems: Three Easy Pieces”, [Remzi H. Arpac-Dusseau](#), [Andrea C. Arpac-Dusseau](#), CreateSpace Independent Publishing Platform, 2018.
ISBN198508659X, 9781985086593, 714 pages.



Chapter 1: Introduction



What Is an
OPERATING SYSTEM (OS)
and How Does It Work

CLEVERISM.COM

Chapter 1: Outline

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



Objectives

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



COMPUTER-SYSTEM ORGANIZATION

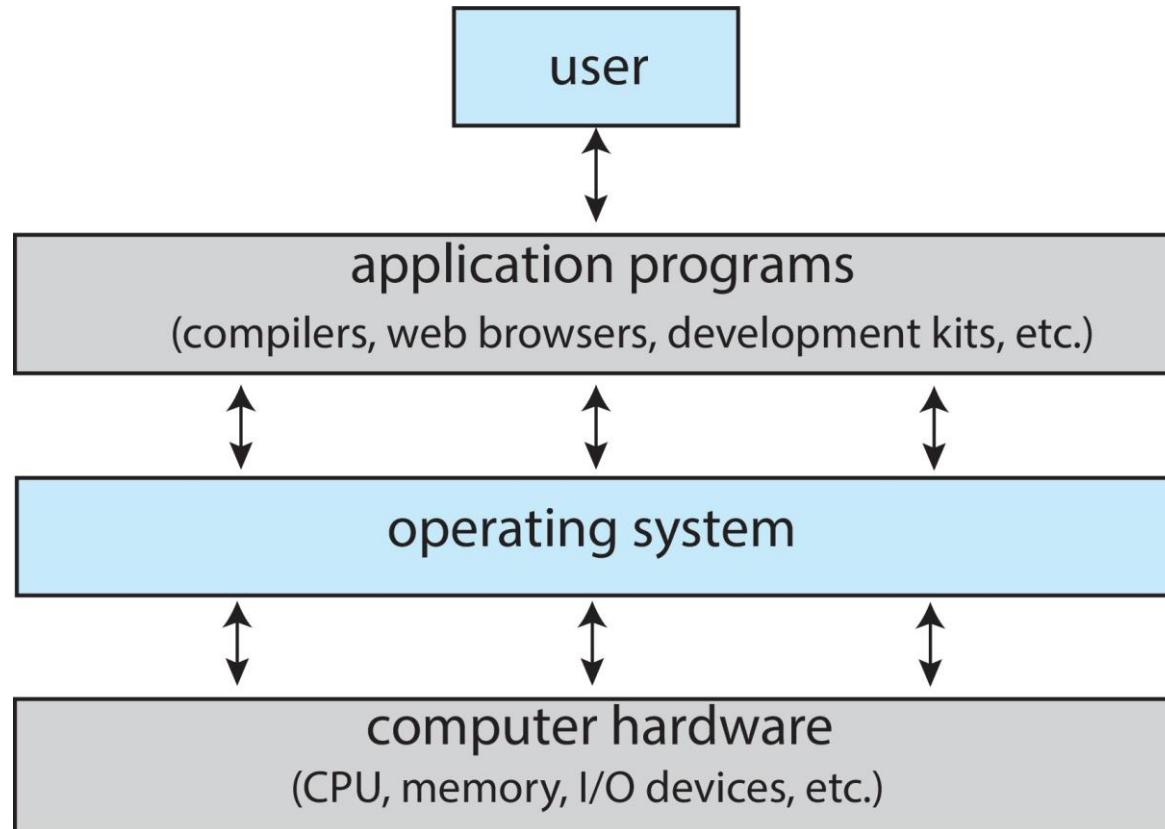


Computer-System Structure

- Computer system can be divided into **4** components:
 - **Hardware (HW)** – provides basic computing resources
 - ▶ E.g., Central Processing Unit (**CPU**), memory, Input/output (**I/O**) devices
 - **Operating system (OS)** – controls and coordinates use of hardware among various applications and users
 - ▶ E.g., Microsoft Windows, Unix, Linux, Apple MacOS
 - **Application programs** – define the ways in which the system resources are used to solve the computing problems of the users
 - ▶ E.g., Compilers, web browsers, development kits, word processors, database systems, video games, multimedia players
 - **Users**
 - ▶ E.g., People, machines, other computers



Abstract View of Computer Components



What Operating Systems Do

- **Users** want convenience, ease of use and good performance
 - ▶ But don't care about *resource utilization*
- **Shared computers** (e.g., *mainframe* or *minicomputer*) keep all users happy
 - ▶ Operating system is a *resource allocator* and *control program* making efficient use of hardware and managing execution of user programs
- **Dedicated systems** (e.g., *workstations*) have dedicated resources but users frequently utilize *shared resources from servers*
- **Mobile devices** (e.g., *smartphones* and *tablets*) are *resource poor*, have to be optimized for *battery life* and usability using *user interfaces* such as touch screens.
- Some computers have little or no user interface, such as **embedded computers** in devices and automobiles
 - ▶ Run primarily *without user intervention*



Defining Operating Systems

- Term OS covers many roles, because of *myriad designs* and *uses* of OSes
 - OSes *present* in toasters through ships, spacecraft, game machines, TVs and industrial control systems
 - OSes were *born* when fixed use of computers for military became more general purpose and needed resource management and program control
- No universally *accepted definition*



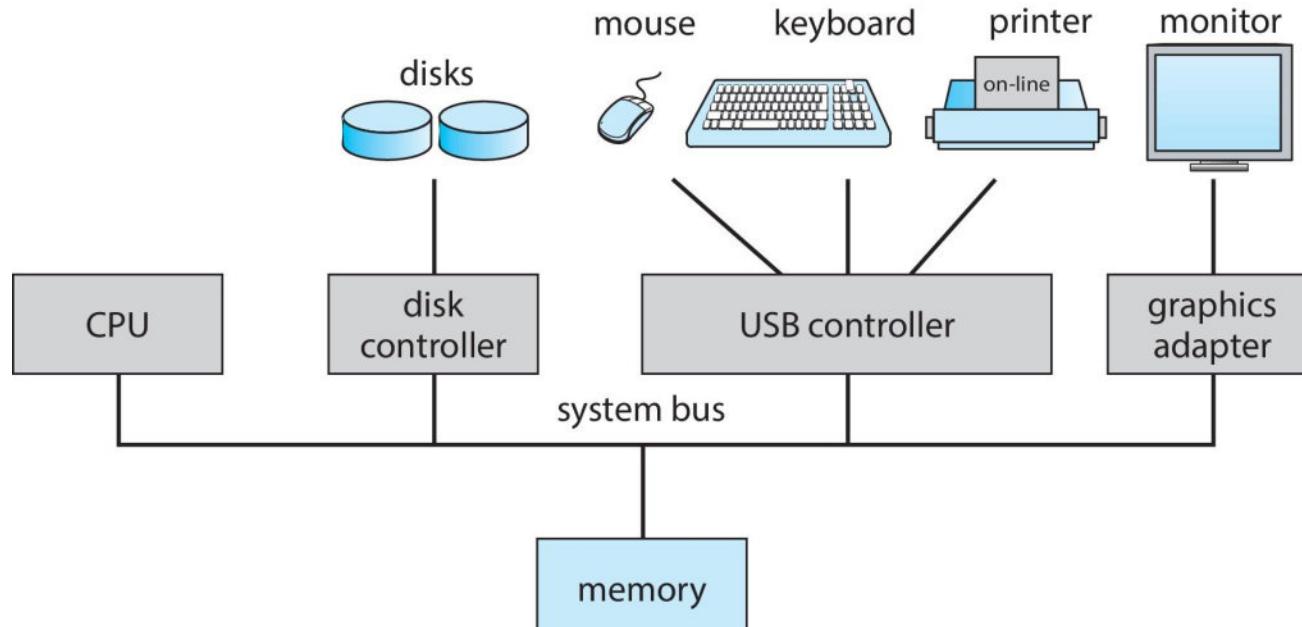
Operating System Definition (Cont.)

- “*Everything a vendor ships when you order an operating system*” is a good approximation (but varies wildly)
- “*The one program running at all times on the computer*” is the **kernel**, part of the operating system
- Everything else is either
 - a **system program** (ships with the operating system, but not part of the kernel), or
 - an **application program**, all programs not associated with the operating system
- Today’s OSes for general purpose and mobile computing also include **middleware** – *a set of software frameworks* that provide additional services to application developers such as databases, multimedia, graphics



Computer-System Organization

- Computer-system organization
 - 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 CPUs can execute *concurrently*
- The **device controller** (*on device*) determines the logical interaction between the device and the computer
 - 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 device does from the *device* to *local buffer* of controller
- Each device controller type has a **device driver** (*installed inside an operating system*) to manage I/O operation
 - Provides *uniform interface* between controller and kernel



Common Functions of Interrupts

□ **Interrupt/Polling**

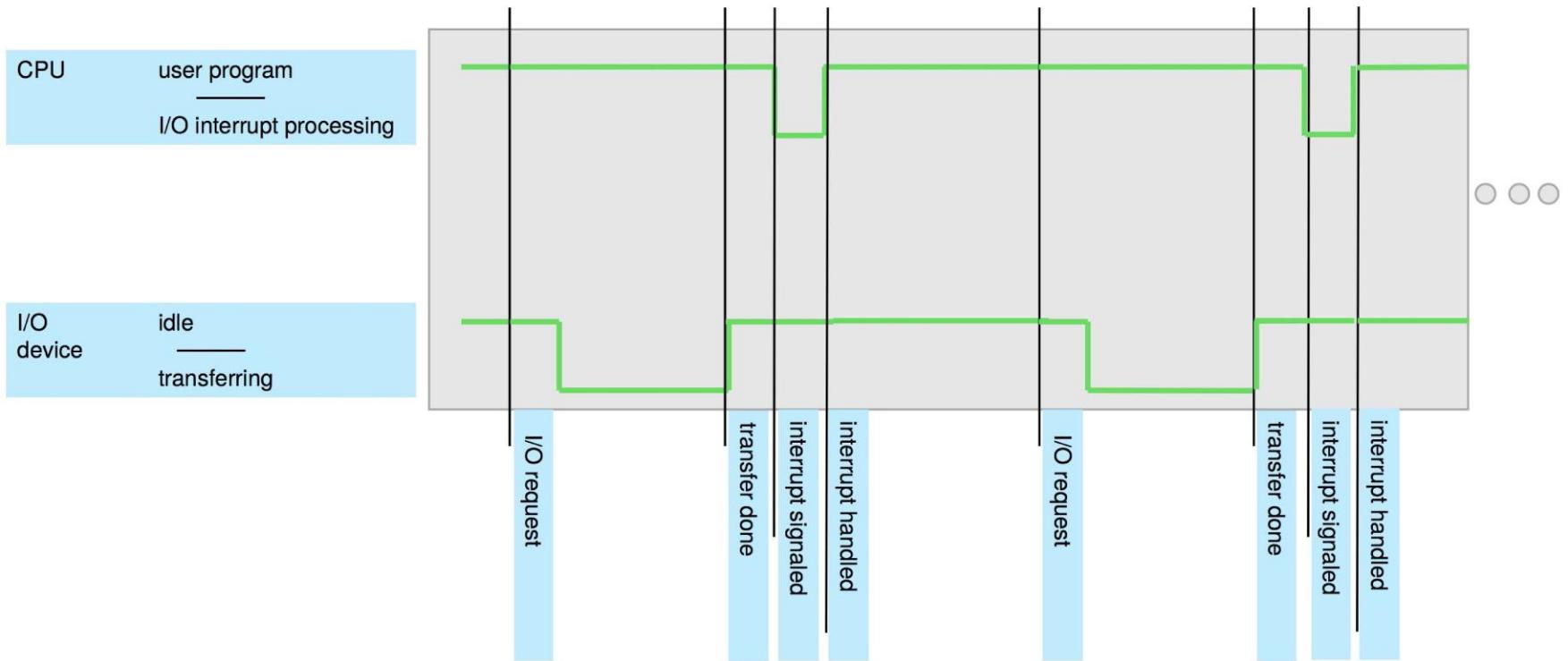
- Device controller informs CPU that it has finished its operation by raising an **interrupt** (*device interrupt* or *hardware interrupt*) or
- CPU has to do a **polling** for an I/O completion (possibly waste a large number of CPU cycles)

□ **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 and status of the *interrupted instruction*
- A **trap** (or **exception**) is a *software-generated interrupt* caused either by an *error* or a *user request*
- An operating system is *interrupt-driven*



Interrupt Timeline

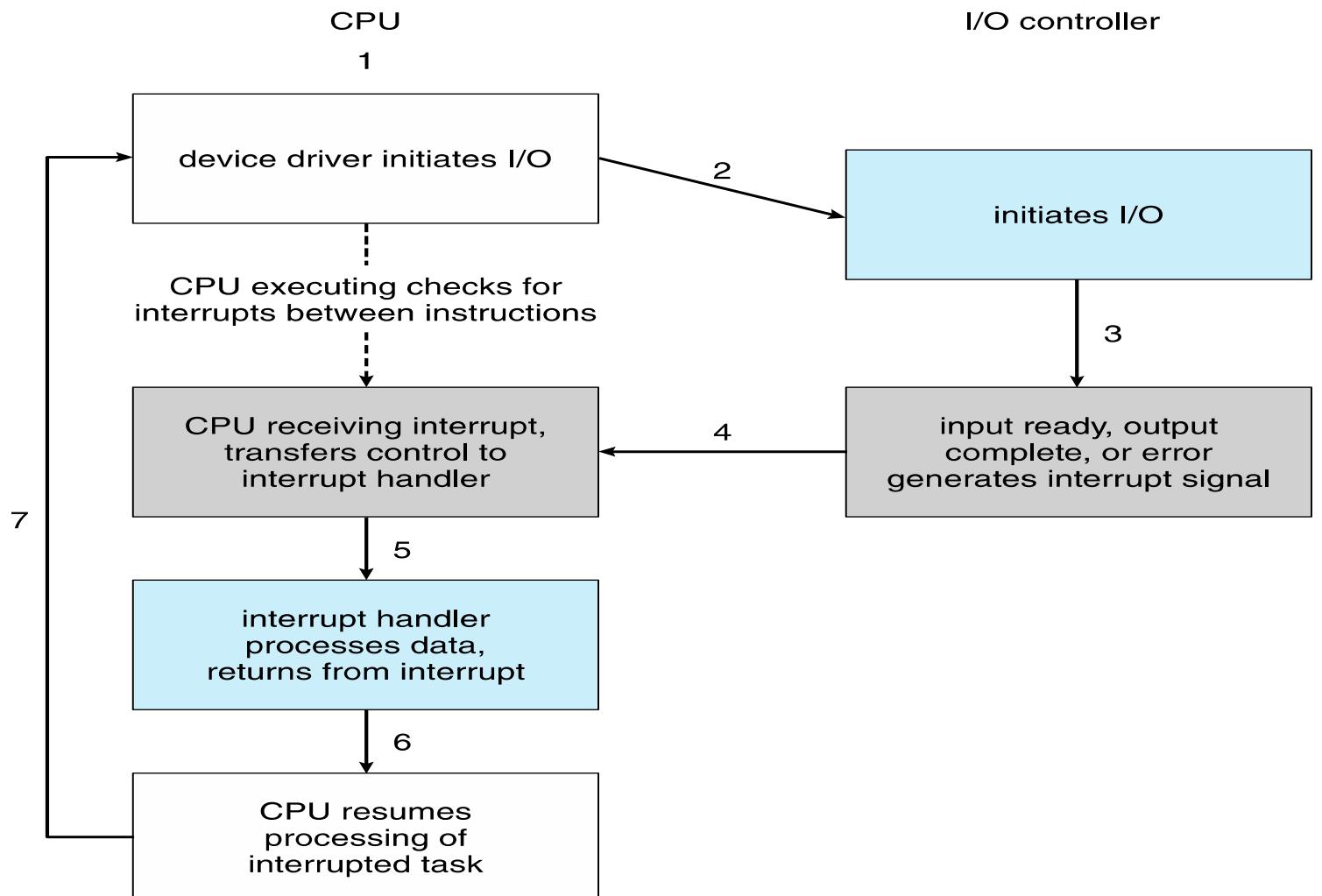


Interrupt Handling

- The operating system *preserves the state of the CPU* by storing **registers** and the **program counter (PC)**
- Determines which type of interrupt has occurred:
 - *Vectored interrupt system* used to handle asynchronous events and to trap to supervisor-mode routines in the kernel
 - *Separate segments of code* determine what action should be taken for each type of interrupt
- Some device drivers use *interrupts* when the *I/O rate is low* and switch to *polling* when the rate increases to the point where *polling is faster and more efficient*.



Interrupt-driven I/O Cycle



I/O Structure

- **First case**: 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 (e.g., contention for memory access)
 - At most one I/O request is outstanding at a time, *no simultaneous I/O processing*
- **Second case**: after I/O starts, *control returns to user program without waiting for I/O completion*
 - **System call** – request to the OS to allow user to wait for I/O completion
 - **Device-status table** contains entry for each I/O device indicating its *type*, *address*, and *state*
 - ▶ OS indexes into **I/O device table** to determine *device status* and to modify table entry to include *interrupt*



Storage Structure

- **Main memory** – only storage media that the *CPU can access directly*
 - *Random access*, typically in the form of **Dynamic Random-Access Memory (DRAM)**
 - Typically, *volatile*
- **Secondary storage** – extension of main memory that provides large *nonvolatile* storage capacity
 - **Hard Disk Drives (HDD)** – rigid metal or glass platters covered with magnetic recording material
 - ▶ Disk surface is logically divided into *tracks*, which are subdivided into *sectors*
 - **Non-volatile memory (NVM)** devices – *faster* than hard disks
 - ▶ Becoming more popular as capacity and performance increases, price drops
 - ▶ Various technologies



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 **1,024²** bytes; a **gigabyte**, or **GB**, is **1,024³** bytes; a **terabyte**, or **TB**, is **1,024⁴** bytes; and a **petabyte**, or **PB**, is **1,024⁵** bytes. Computer manufacturers often **round off** these numbers and say that a megabyte is **1 million** bytes, and a gigabyte is **1 billion** bytes. Networking measurements are an exception to this general rule; they are given in bits (because *networks move data a bit at a time*).

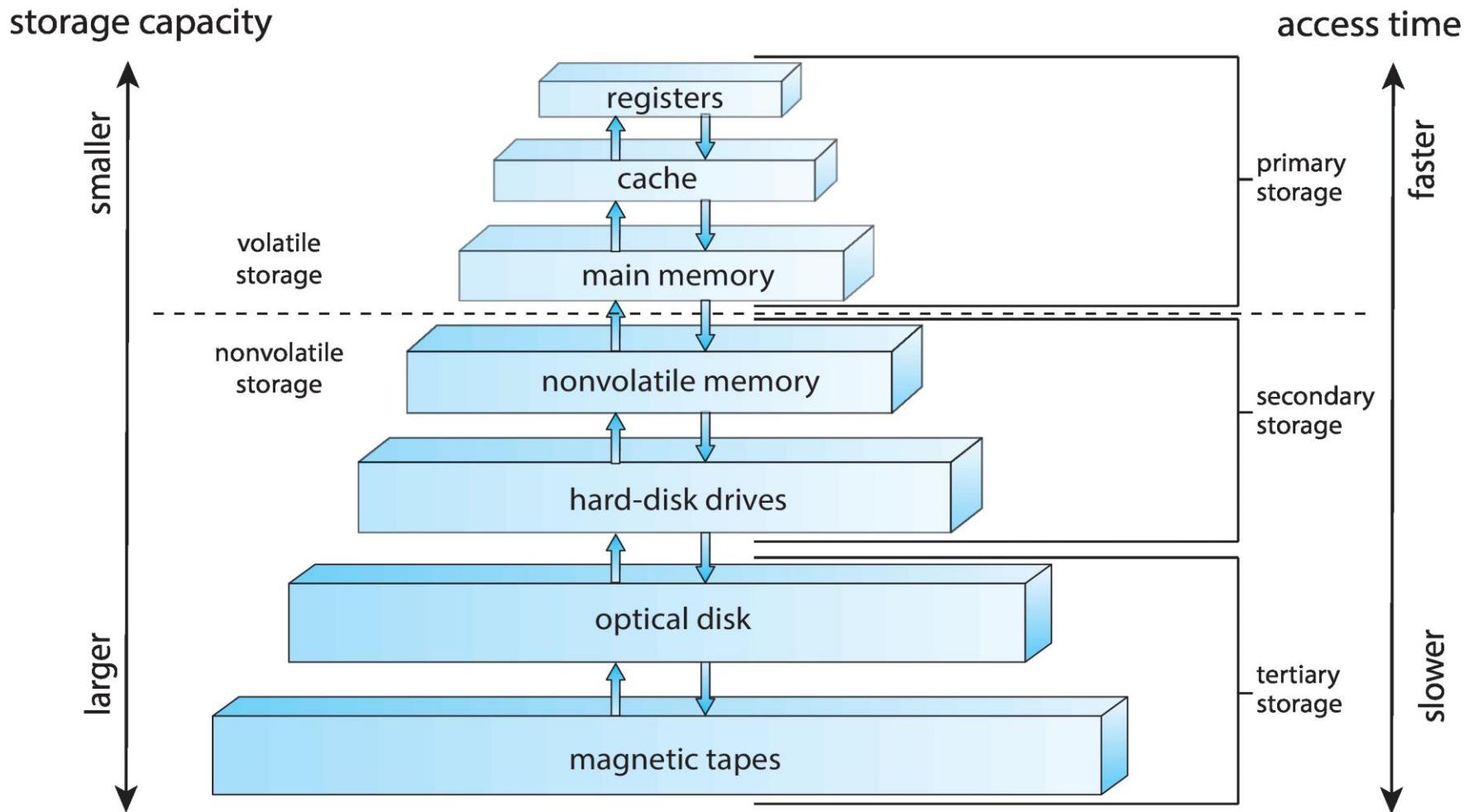


Storage Hierarchy

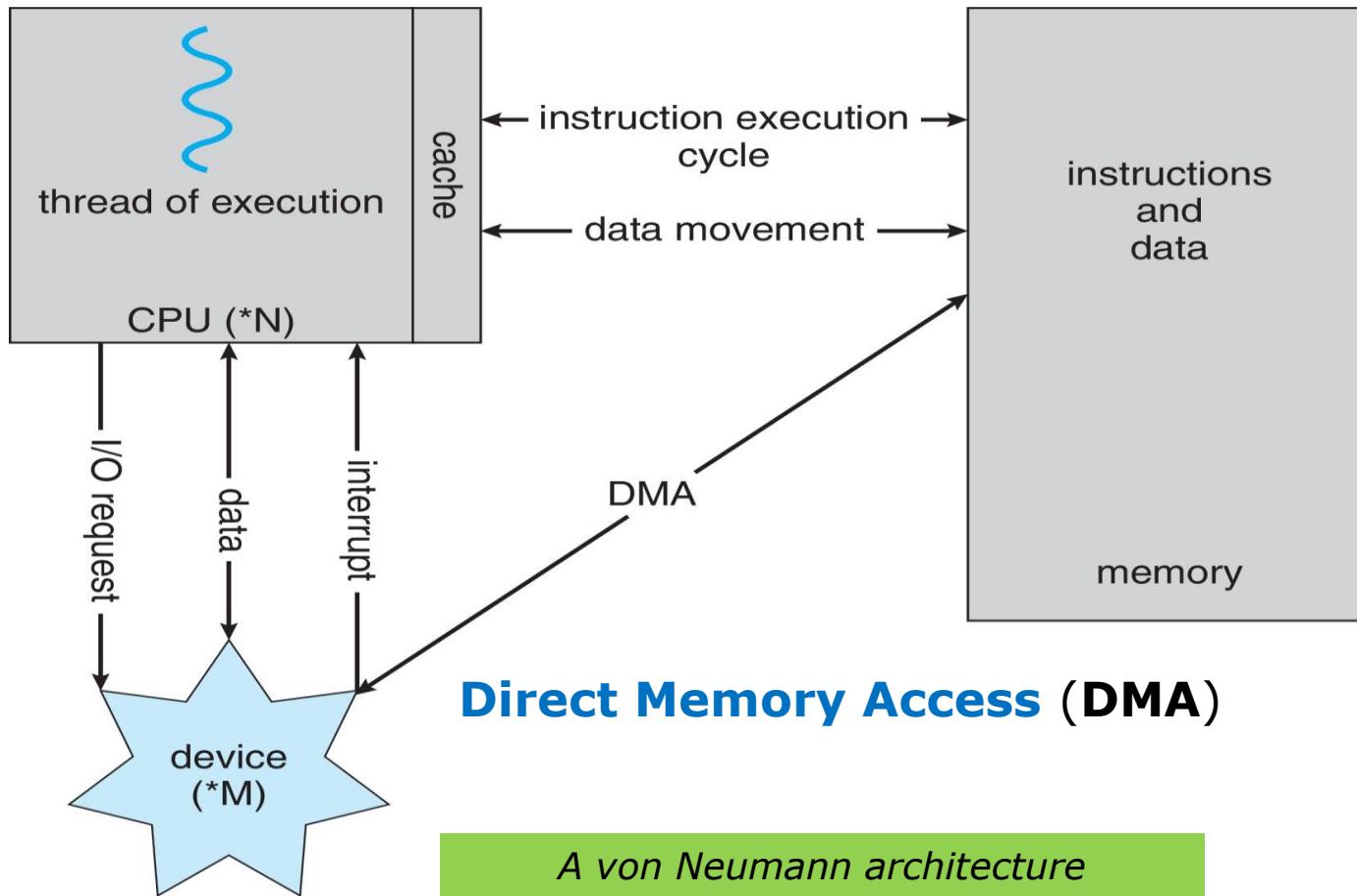
- Storage systems are organized in hierarchy according to
 - *Speed* (or *access time*)
 - *Capacity*
 - *Volatility*
 - *Cost*
- **Caching** – mechanism *copying data* into faster storage system
 - *Main memory* can be viewed as a *cache* for *secondary storage*



Storage-Device Hierarchy



How a Modern Computer Works



Direct Memory Access (DMA) Structure

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

COMPUTER-SYSTEM ARCHITECTURE

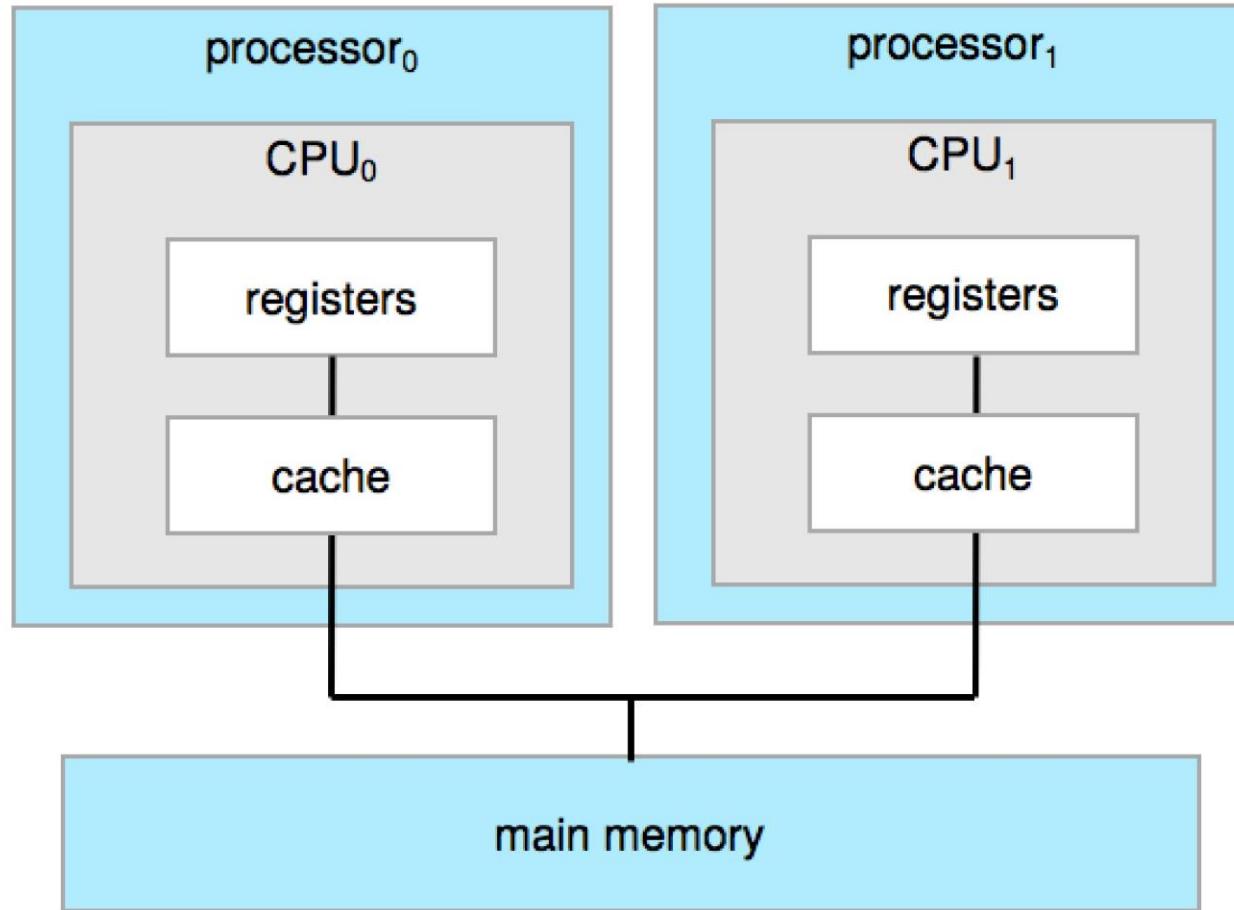


Computer-System Architecture

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

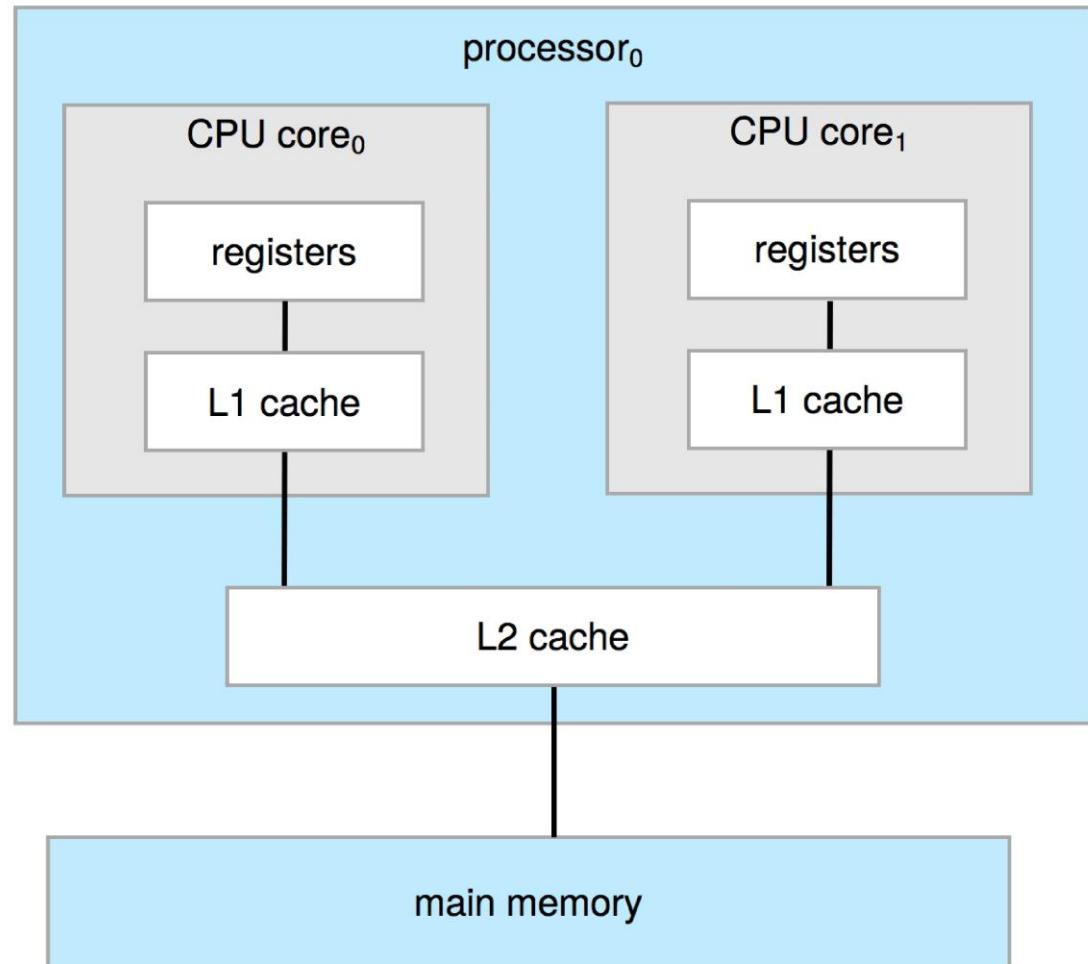


Symmetric Multiprocessing Architecture

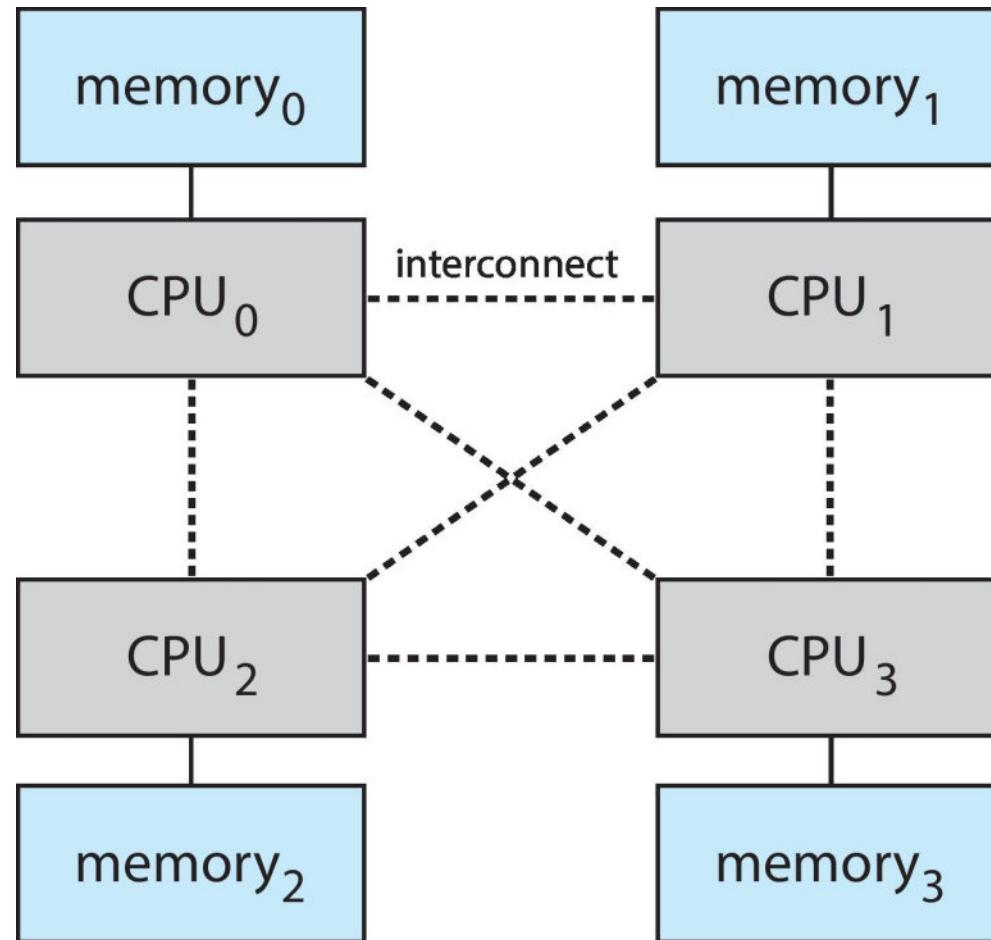


A Dual-Core Design

- **Chassis** containing multiple separate systems
- **Systems** containing all chips
- **Multiprocessor**
- **Multicore**
- Chassis -> Multi-systems -> Multi-core CPU



Non-Uniform Memory Access System

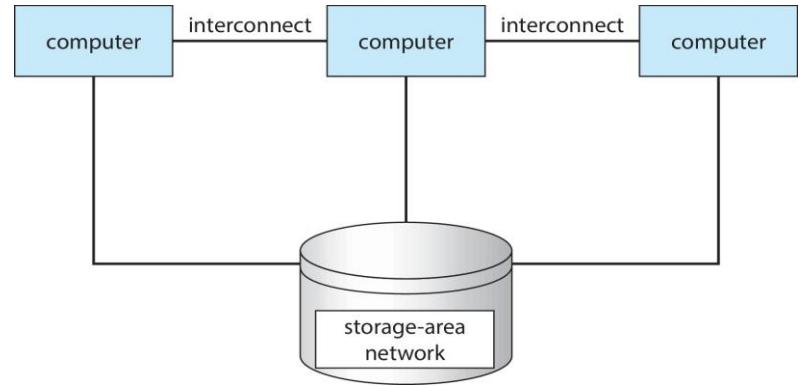


Non-Uniform Memory Access (**NUMA**)



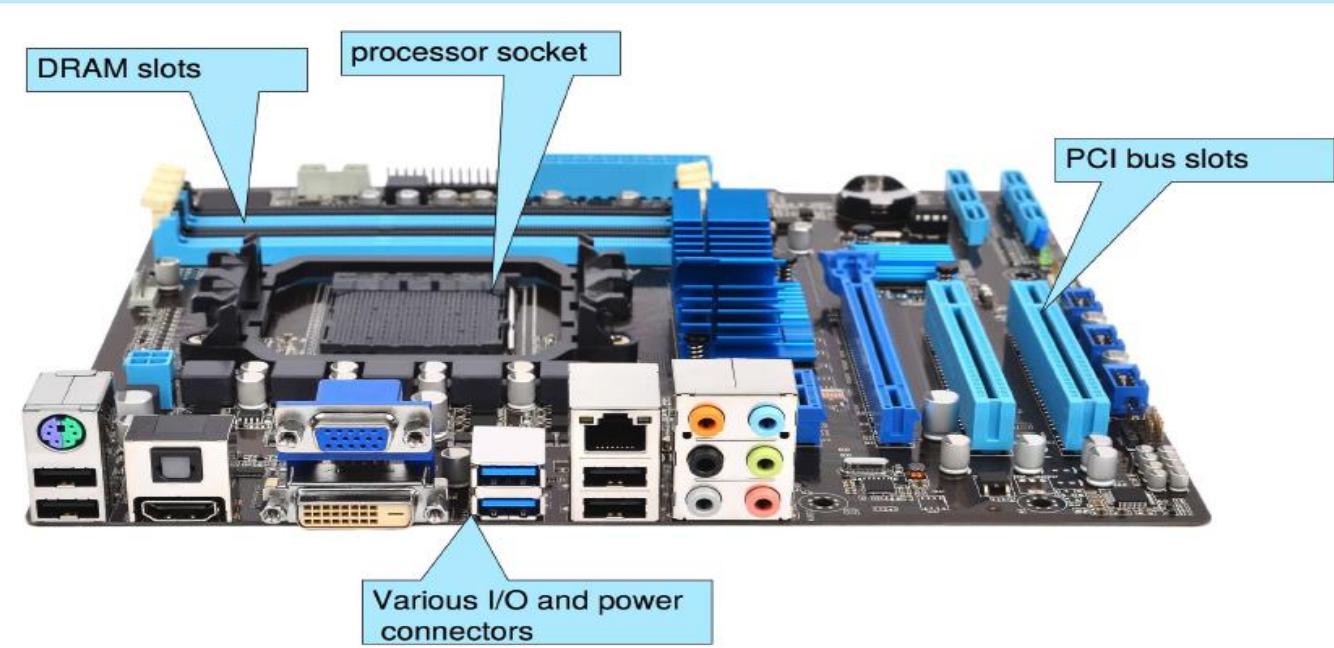
Clustered Systems

- Like multiprocessor systems, but *multiple systems working together*
 - Usually *sharing storage* via a **Storage-Area Network (SAN)**
 - Provides a *high-availability (HA) service* which survives failures
 - *Asymmetric clustering* has one machine in hot-standby mode
 - *Symmetric clustering* has multiple nodes running applications
- Some clusters are used for **High-Performance Computing (HPC)**
 - Applications must be written to use *parallelization*
- Some clusters have **Distributed Lock Manager (DLM)** to avoid conflicting operations



PC Motherboard

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



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



OPERATING-SYSTEM OPERATION



Operating-System Operation

- **Bootstrap program** 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
- Loads **kernel** (part of **OS**)
 - Kernel is *interrupt-driven* (hardware and software)
 - *Hardware interrupt* by one of the devices
 - *Software interrupt (exception or trap)*: Software error (e.g., division by zero), request for operating system service (i.e., **system call**), other process problems include infinite loop, processes modifying each other or modifying the operating system
- Starts **system daemons** (services provided *outside of the kernel*)



Multiprogramming

- **Multiprogramming** needed for efficiency
 - Single user cannot keep CPU and I/O devices busy at all times
 - *Multiprogramming organizes jobs (i.e., code and data) so that CPU always has one to execute*
 - A subset of total jobs in system is *kept in memory*
 - One job is selected and runs via **job scheduling**
 - When it has to wait (e.g., for I/O), *OS switches to another job*

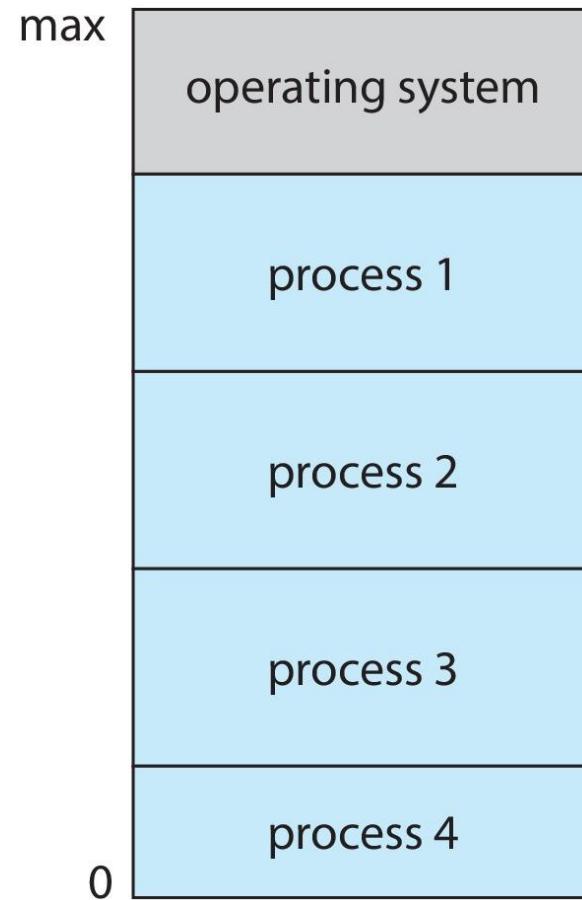


Multitasking

- **Time-sharing (multitasking)** is a logical extension in which CPU *switches jobs so frequently* that users can interact with each job while it is running, creating *interactive computing*
 - Response time should be < *1 second*
 - Each user has at least one program executing in memory ⇒ *process*
 - If several jobs ready to run at the same time ⇒ *CPU scheduling*
 - If processes don't fit in memory, *swapping* moves them in and out from/to a **Backing store**
 - *Virtual memory* allows execution of processes *not completely in memory*



Memory Layout for Multiprogrammed System

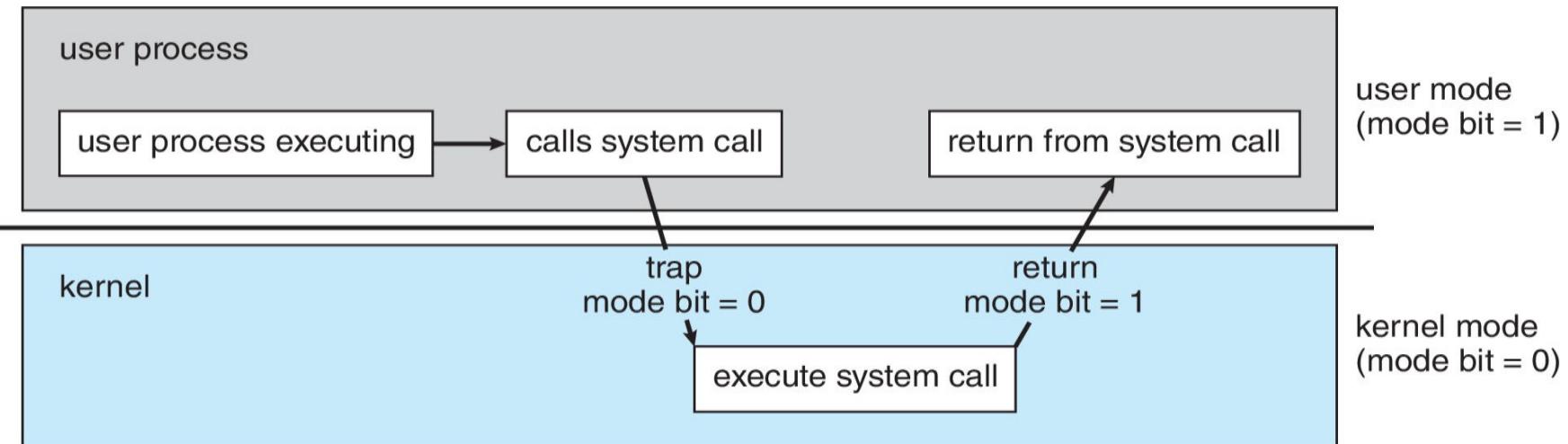


Dual-mode and Multimode Operation

- Dual-mode operation allows OS to *protect* itself and other system components
 - Two modes: **User mode** and **kernel mode**
 - Change mode using **Mode bit** provided by hardware
 - ▶ Provides ability to distinguish *when system is running user code or kernel code*
 - ▶ Some instructions designated as *privileged*, only executable in **kernel mode**
 - ▶ *System call* changes mode to kernel, return from call resets it to user mode
- Increasingly, CPUs support **multimode** operations
 - e.g., **Virtual Machine Manager (VMM)** mode for guest **Virtual Machine (VMs)**



Transition from User to Kernel Mode



- **Timer** to prevent *infinite loop / process hogging resources*
 - Timer is set to interrupt the computer after some time period. Operating system sets a *counter* (privileged instruction), keeps the counter that is *decremented by the physical clock*, when counter zero generate an interrupt.
 - Set up before scheduling process to regain control or terminate program that exceeds *allotted time*.



RESOURCE MANAGEMENT



Process Management

- A **process** is a program in execution. It is a unit of work within the system. Program is a *passive entity*; process is an *active entity*.
 - Process needs **resources** to accomplish its task
 - CPU, memory, I/O, files, initialization data
 - Process termination requires reclaim of any reusable resources
 - **Single-threaded process** has *one program counter* specifying location of next instruction to execute
 - Process executes instructions *sequentially*, one at a time, until completion
 - **Multi-threaded process** has *one program counter per thread*
- Typically, system has many processes (some user processes & some OS processes) *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** needed by the program must be *in memory*
- Memory management determines *what* is in memory and *when*
 - Optimizing *CPU utilization* and *computer response* to users
- OS 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



Filesystem Management

- OS provides *uniform, logical view* of *data storage*
 - Abstracts physical properties to logical storage unit - **file**
 - Each *medium* is controlled by device (i.e., disk drive, tape drive)
 - Varying properties include *access speed, capacity, data-transfer rate, access method (sequential or random)*
- Filesystem management
 - **Files** usually organized into **directories**
 - **Access control** on most systems to determine *who* can access *what*
- OS activities
 - *Creating* and *deleting* files / directories, primitives to *manipulate* files / directories, to *backup* files onto stable (non-volatile) storage media
 - *Mapping* files onto *secondary storage*



Mass-Storage Management

- Usually, **disks** used to store **programs** and **data** that do *not fit* in main memory or 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*
- Some storage need not to be fast
 - **Tertiary storage** includes optical storage, magnetic tape
 - Still must be managed – by OS or applications
- OS activities
 - *Mounting* and *unmounting*
 - *Free-space* management
 - Storage *allocation*
 - Disk *scheduling*
 - *Partitioning*
 - *Protection*



Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- *Information in use copied from slower to faster storage temporarily*
- Faster storage (**cache**) checked 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 is smaller than storage being cached
 - *Cache management* is an important design problem
 - *Cache size and replacement policy*



Various Types of Storage

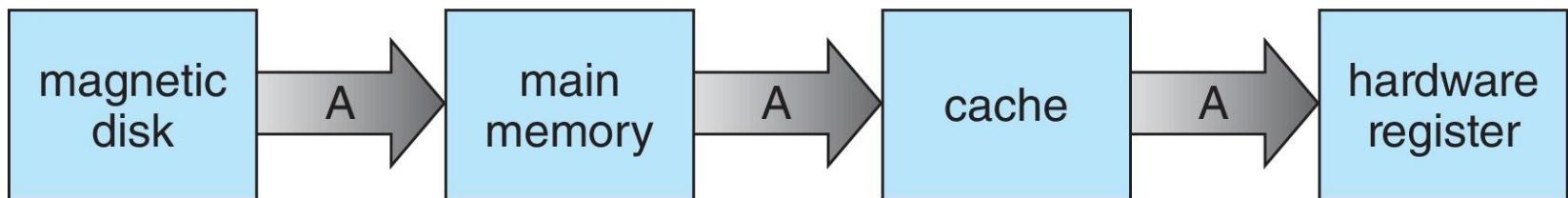
- Movement between levels of storage hierarchy can be *explicit* or *implicit*

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



Migration of Data from Disk to Register

- In **multitasking environment**, it must be careful to use most recent value, no matter where it is stored in the storage hierarchy
- **Multiprocessor environment** must provide *cache coherency* in hardware such that all CPUs have the most recent value in their cache
- In **distributed environment**, the situation is even more complex
 - Several copies of a datum can exist
 - Various solutions



I/O Subsystem

- One purpose of OS is to *hide peculiarities of hardware devices* from the user
- **I/O subsystem** is responsible for
 - Memory management of I/O includes
 - *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
- I/O subsystem includes
 - *General device-driver interface*
 - *Drivers* for specific hardware devices



OTHER ASPECTS



Protection and Security

- **Protection** – any mechanism for *controlling access* of processes or users to resources defined by the OS
- **Security** – defense *against internal and external attacks*
 - Huge range, including *denial-of-service, worms, viruses, identity theft, theft of service*
- Systems generally first distinguish among users, to determine *who can do what*
 - **User identity (UID**, or security ID) includes name and an associated number. User ID is then associated with all files, processes of that user to determine access control
 - **Group identifier (GID)** allows set of users to be defined for access control, then also associated with each process or file
 - **Privilege escalation** allows user to change to *effective ID* with more rights (e.g., using *setuid* in Linux systems)



Virtualization

- *Allows operating systems to run applications within other OSes*
 - Vast and growing industry
- **Emulation** used when source CPU type different from target CPU type (e.g., 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
 - E.g., Consider VMware running WinXP **guests**, each running applications, all on native WinXP **host** OS
 - **Virtual Machine Manager (VMM)** provides virtualization services

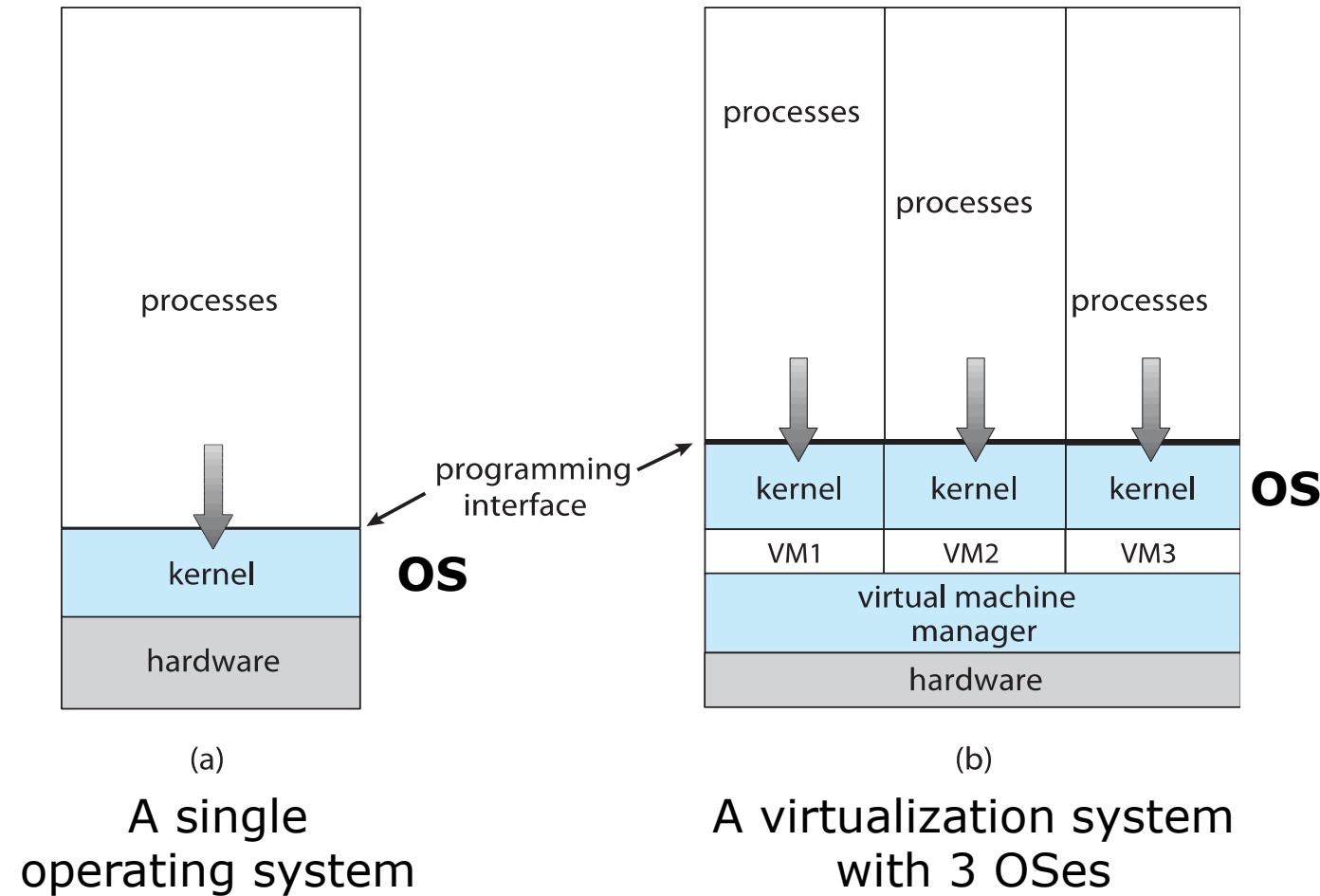


Virtualization (cont.)

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



Computing Environments - Virtualization



Distributed Systems

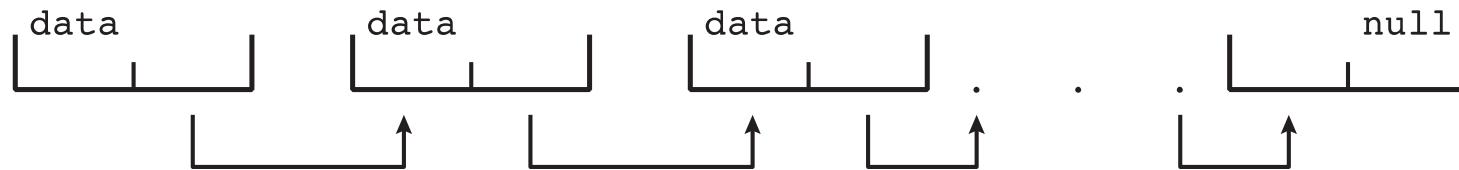
- **Distributed computing**
 - Collection of *separate* (possibly *heterogeneous*) *systems* networked together
 - **Network** is communications paths (**TCP/IP** is most common protocol stack)
 - **Local Area Network (LAN)**
 - **Wide Area Network (WAN)**
 - **Metropolitan Area Network (MAN)**
 - **Personal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



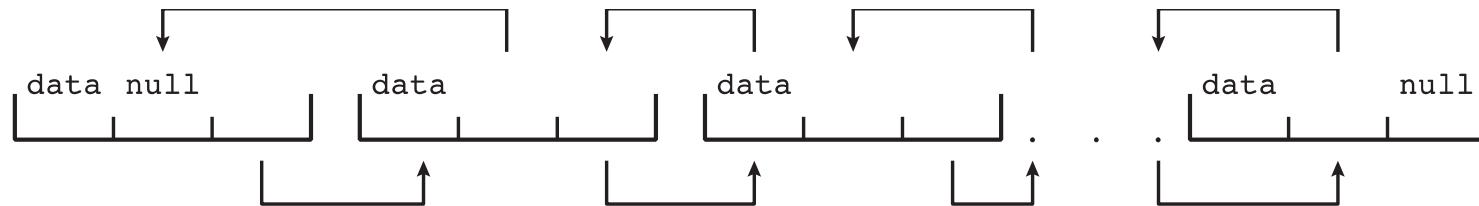
Kernel Data Structures

Many similar to *standard programming data structures*

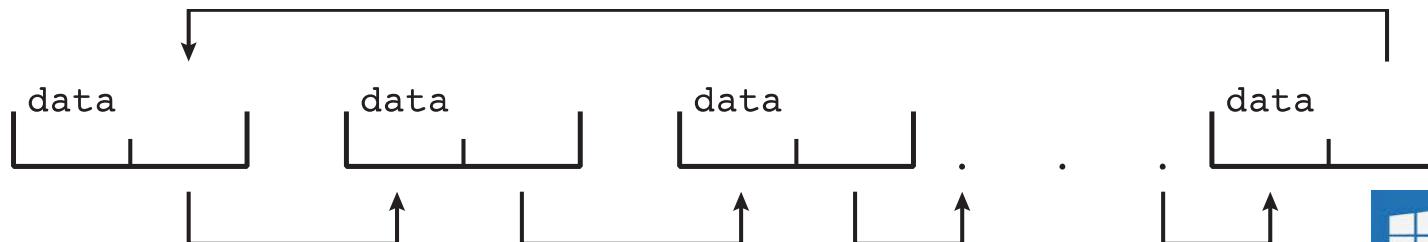
❑ **Singly linked list**



❑ **Doubly linked list**

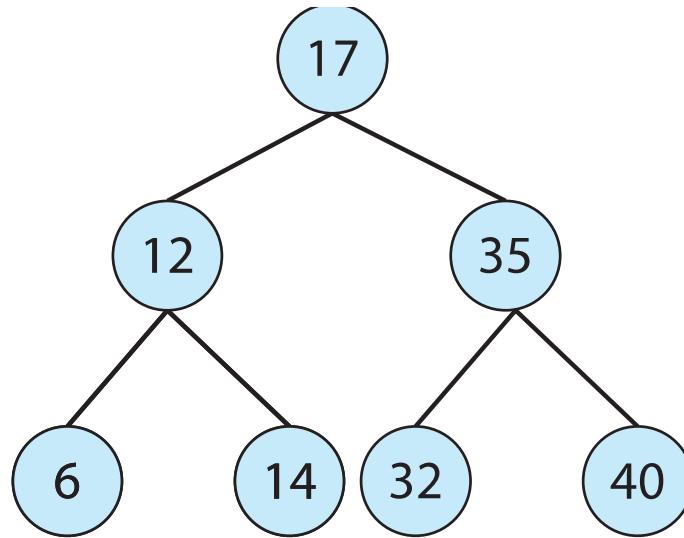


❑ **Circular linked list**



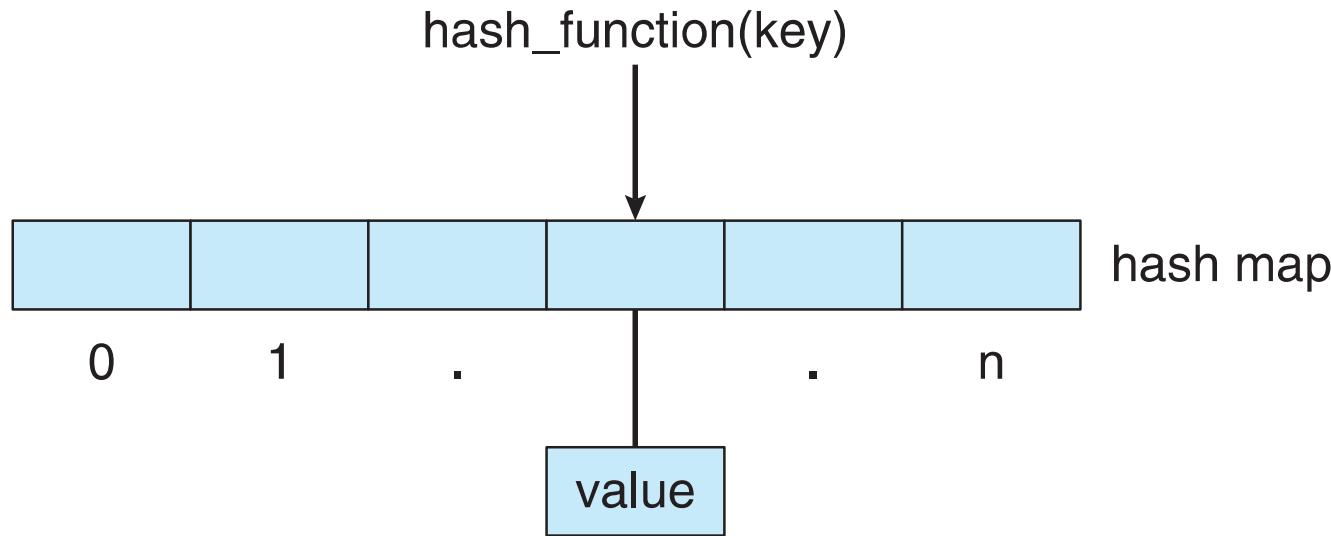
Kernel Data Structures

- **Binary search tree** ($\text{left} \leq \text{right}$)
 - Search performance is $O(n)$
 - **Balanced binary search tree** is $O(\log n)$



Kernel Data Structures

- **Hash function** can create a **hash map**



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



Evolution

- *Mainframe* system
- *Desktop* system
- *Multiprocessor* system
- *Distributed* system
- *Real-time* system
- *Handheld* system/*mobile* system



Computing Environments - Traditional

- *Stand-alone general-purpose machines*
 - But blurred as most systems interconnect with others (i.e., the Internet)
- **Portals** provide web access to internal systems
- **Network computers** (or *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

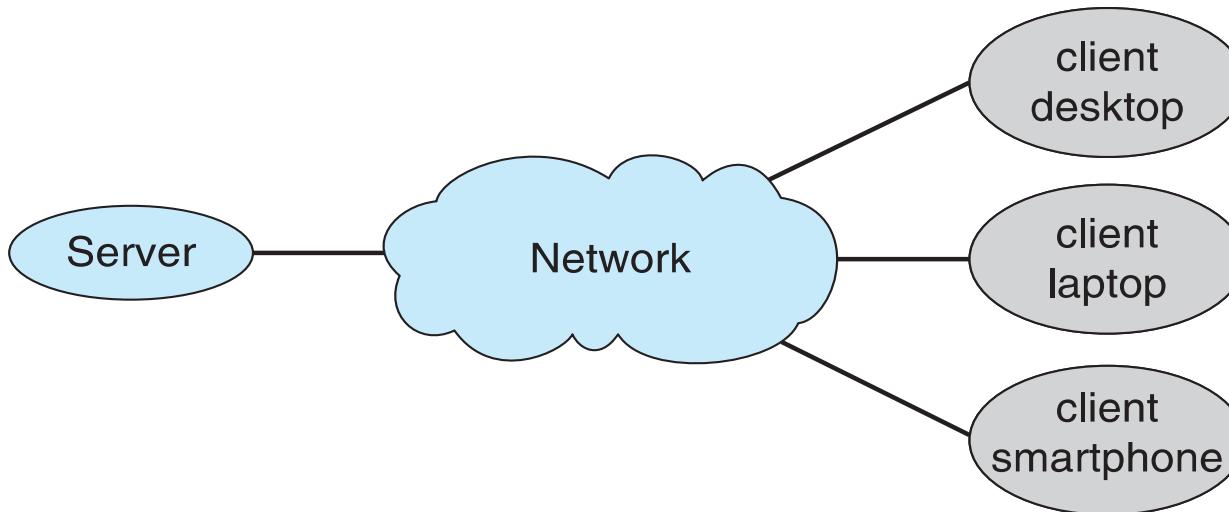
- Such as *handheld smartphones, tablets, etc.*
- What is the functional difference between them and a “traditional” laptop?
 - Extra feature – *more OS features* (e.g., GPS, gyroscope)
 - Allows *new types of apps* like **Augmented Reality (AR)**
 - Use IEEE 802.11 wireless, or cellular data networks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



Computing Environments – Client-Server

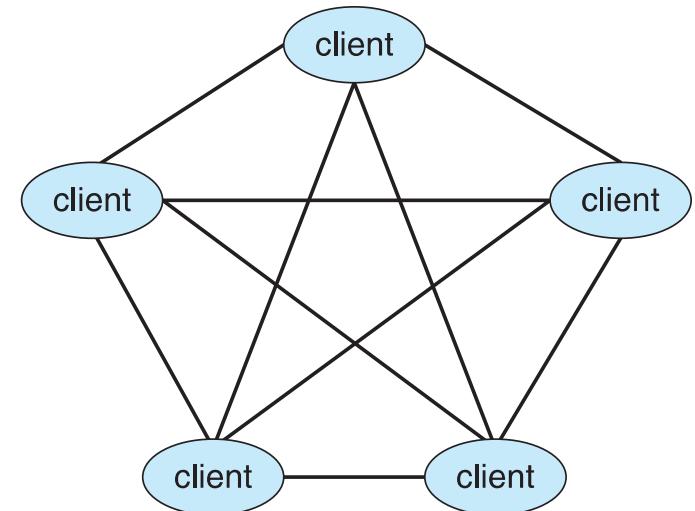
□ Client-Server Computing

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



Computing Environments - Peer-to-Peer

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



Computing Environments – Cloud Computing

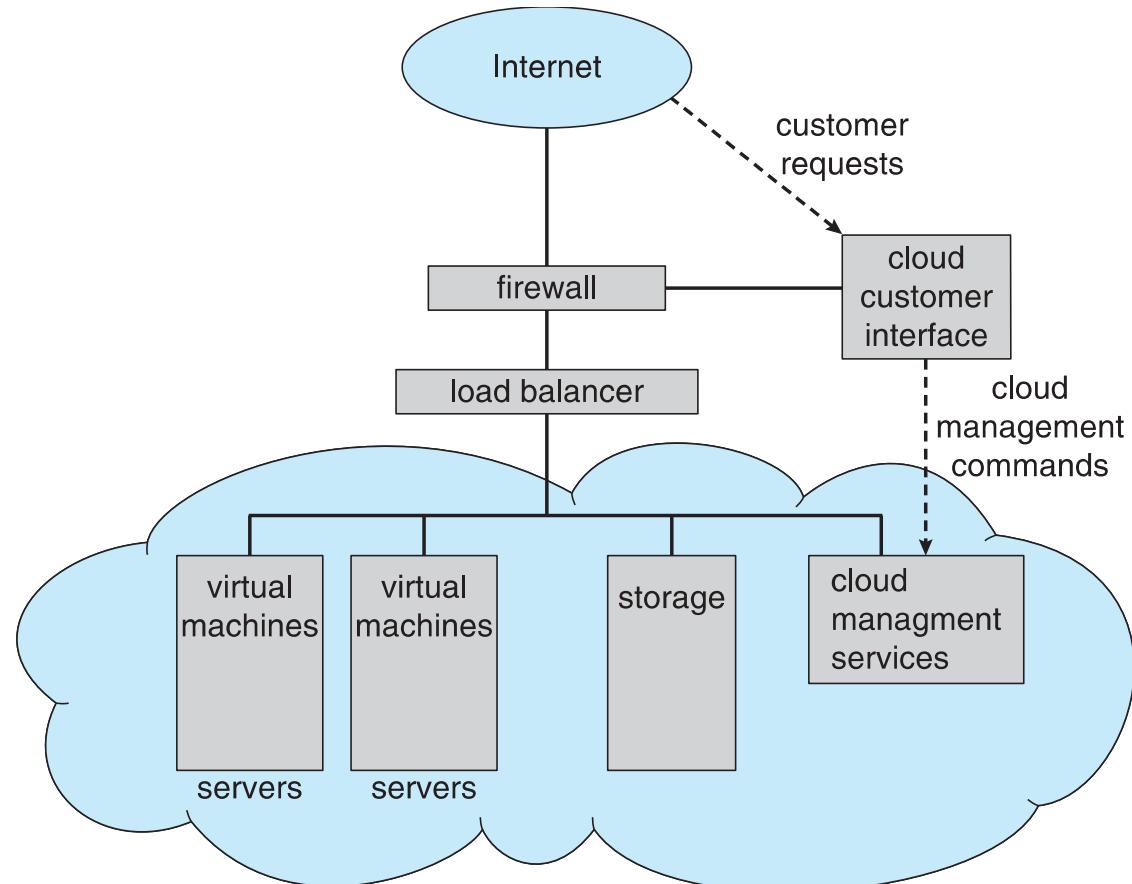
- Delivers *computing, storage, apps* as a service across a network
- Logical extension of *virtualization* because it uses virtualization as the base for its functionality.
 - E.g., **Amazon EC2** has thousands of servers, millions of virtual machines, petabytes of storage *available across the Internet*
- Many types of **services**
 - **Software as a Service (SaaS)** – one or more applications available via the Internet
 - **Platform as a Service (PaaS)** – software stack ready for application use via the Internet
 - **Infrastructure as a Service (IaaS)** – servers or storage available over Internet (i.e., storage available for backup use)
- Many types of **structure**
 - **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



Computing Environments – Cloud Computing

- Cloud computing environments composed of **traditional OSes**, plus **VMMs**, plus **cloud management tools**

- *Internet connectivity* requires security like **firewalls**
- *Load balancers* spread traffic across multiple applications



Computing Environments – Real-Time Embedded Systems

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



Free and Open-Source Operating Systems

- Operating systems made available in *source-code format* rather than just binary *closed-source* and *proprietary*
- The purpose is to counter to the *copy protection* and **Digital Rights Management (DRM)** movement
- Started by **Free Software Foundation (FSF)**, which has “copyleft” **GNU Public License (GPL)** or **Lesser GPL (LGPL)**
 - Free software and open-source software are *two different ideas* championed by different groups of people
 - ▶ <http://gnu.org/philosophy/open-source-misses-the-point.html/>
 - E.g., **GNU/Linux** and **BSD UNIX** (including **Darwin**, core of **Mac OS X**)
- Use VMM like **VMware Player** (Free on Windows), **VirtualBox**
 - Use to run *guest operating systems* for exploration



The Study of Operating Systems

- There has never been a more interesting time to study operating systems, and it has never been easier. The open-source movement has overtaken operating systems, causing many of them to be made available in both **source** and **binary** (executable) format. The list of operating systems available in both formats includes **Linux**, **BSD UNIX**, **Solaris**, and part of **macOS**. The availability of source code allows us to study operating systems from the inside out. Questions that we could once answer only by looking at documentation or the behavior of an operating system we can now answer by examining the code itself.
- Operating systems that are no longer commercially viable have been open-sourced as well, enabling us to study how systems operated in a time of fewer CPU, memory, and storage resources. An extensive but incomplete list of **open-source operating-system** projects is available from https://curlie.org/Computers/Software/Operating_Systems/Open_Source/
- In addition, the rise of **virtualization** as a mainstream (and frequently free) computer function makes it possible to run many operating systems on top of one core system. For example, **VMware** (<http://www.vmware.com>) provides a free “player” for Windows on which hundreds of free “virtual appliances” can run. **VirtualBox** (<http://www.virtualbox.com>) provides a free, open-source virtual machine manager on many operating systems. Using such tools, students can try out hundreds of operating systems without dedicated hardware.
- *The advent of open-source operating systems has also made it easier to make the move from student to operating-system developer.* With some knowledge, some effort, and an Internet connection, a student can even create a new operating-system distribution. Just a few years ago, it was difficult or impossible to get access to source code. Now, such access is limited only by how much interest, time, and disk space a student has.



Summary

- An **operating system** is software that *manages* the computer hardware, as well as *providing an environment* for application programs to run.
- **Interrupts** are a keyway in which hardware interacts with the operating system. A hardware device triggers an interrupt by sending a signal to the CPU to alert the CPU that some event requires attention. The interrupt is managed by the *interrupt handler*.
- For a computer to do its job of **executing programs**, the programs must be *in main memory*, which is the *only large storage area* that the processor can access directly.
- The **main memory** is usually a *volatile storage device* that loses its contents when power is turned off or lost.



Summary (Cont.)

- **Nonvolatile storage** is an extension of main memory and is capable of *holding large quantities of data permanently*.
- The most common nonvolatile storage device is a **hard disk**, which can provide storage of both programs and data.
- The wide variety of **storage systems** in a computer system can be organized in a hierarchy according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases.
- Modern computer architectures are **multiprocessor systems** in which each CPU contains *several computing cores*.



Summary (Cont.)

- To best utilize the CPU, modern operating systems employ **multiprogramming**, which allows *several jobs to be in memory* at the same time, thus ensuring that the CPU always has a job to execute.
- **Multitasking** is an extension of multiprogramming wherein CPU scheduling algorithms *rapidly switch between processes*, providing users with a fast response time.
- To prevent user programs from interfering with the proper operation of the system, the system hardware has two modes: **user mode** and **kernel mode**.
- **Various instructions are privileged** and can be executed only in kernel mode. Examples include the instruction to *switch to kernel mode*, *I/O control*, *timer management*, and *interrupt management*.



Summary (Cont.)

- A **process** is the fundamental *unit of work* in an operating system. Process management includes creating and deleting processes and providing mechanisms for processes to *communicate* and *synchronize* with each other.
- An operating system manages **memory** by keeping track of what parts of memory are being used and by whom. It is also responsible for dynamically allocating and freeing memory space.
- **Storage space** is managed by the operating system; this includes providing file systems for representing files and directories and managing space on mass-storage devices.
- Operating systems provide **mechanisms for protecting and securing** the operating system and users. Protection measures control the access of processes or users to the resources made available by the computer system.



Summary (Cont.)

- **Virtualization** involves abstracting a computer's hardware into several different execution environments.
- **Data structures** that are used in an operating system include *lists*, *stacks*, *queues*, *trees*, and *maps*.
- **Computing** takes place in a variety of **environments**, including *traditional computing*, *mobile computing*, *client-server systems*, *peer-to-peer systems*, *cloud computing*, and *real-time embedded systems*.
- **Free and open-source operating systems** are available in source-code format. *Free software is licensed to allow no-cost use, redistribution, and modification*. GNU/Linux, FreeBSD, and Solaris are examples of popular open-source systems.



End of Chapter 1



What Is an
OPERATING SYSTEM (OS)
and How Does It Work

CLEVERISM.COM

Chapter 2: Operating-System Structures





Chapter 2:Outline

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





Objectives

- Identify **services** provided by an operating system
- Illustrate how **system calls** are used to provide operating system services
- Compare and contrast **monolithic**, **layered**, **microkernel**, **modular**, and **hybrid** strategies for designing operating systems
- Illustrate the process for **booting** an operating system
- Apply tools for **monitoring** operating system performance
- Design and implement **kernel modules** for interacting with a Linux kernel





Operating System Services

- Operating systems provide an *environment for execution* of programs and *services* to programs and users
- A 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)**, **Graphical User Interface (GUI)**, Touch-screen
 - **Program execution** - The system must be able to *load* a program into memory, to *run* that program, and *end* execution, either normally or abnormally (indicating error)
 - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device





Operating System Services (Cont.)

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



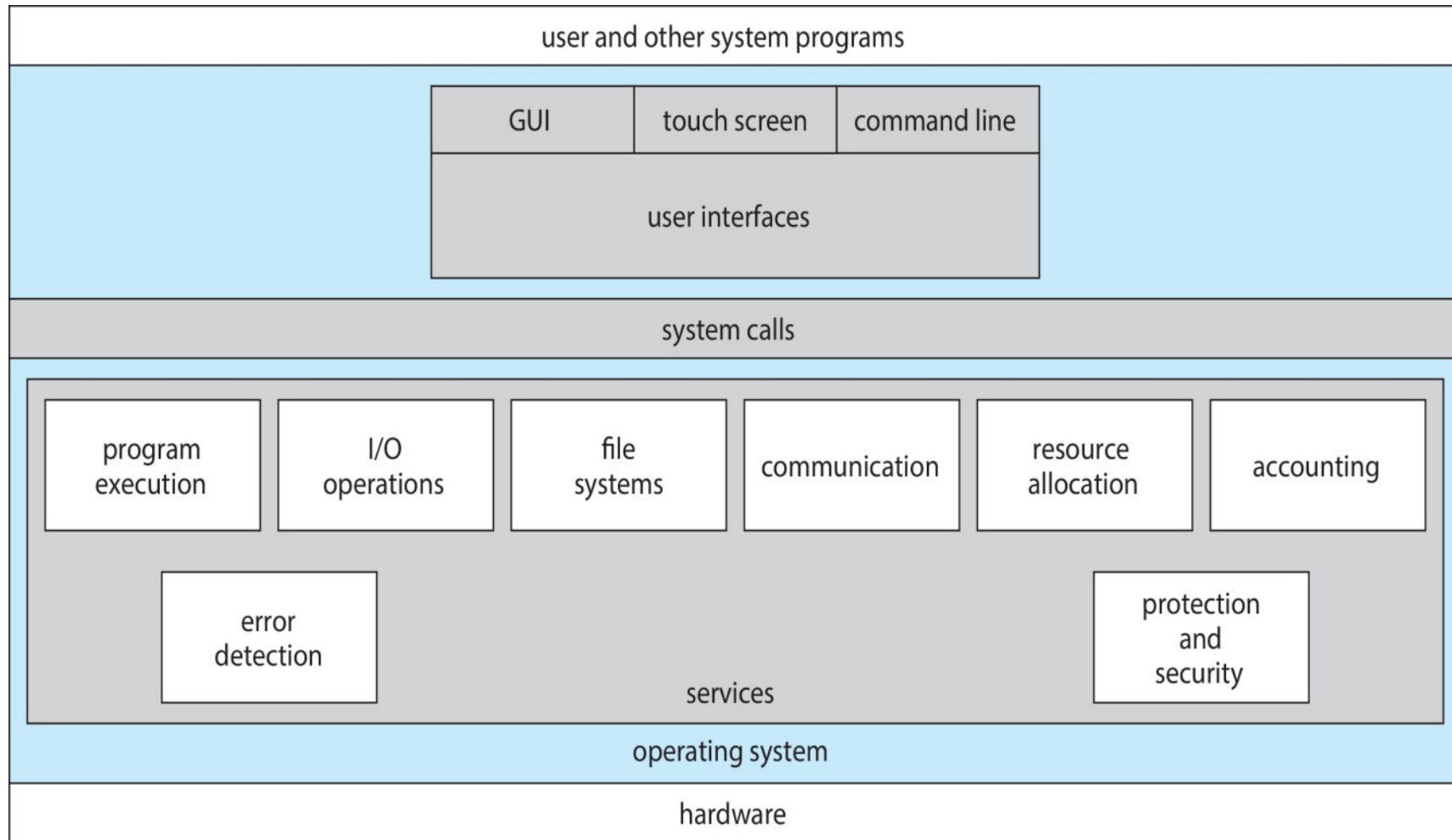


Operating System Services (Cont.)

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



A View of Operating System Services





User Operating System Interface - CLI

- CLI or **command interpreter** allows direct command entry
 - Sometimes implemented in kernel, sometimes by system programs
 - Sometimes multiple flavors implemented – *shells*
 - Primarily fetches a command from user and executes it
 - Sometimes commands built-in, sometimes just names of programs
 - ▶ If the latter, adding new features doesn't require shell modification





Bourne Shell Command Interpreter

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





User Operating System Interface - GUI

■ User-friendly desktop metaphor interface

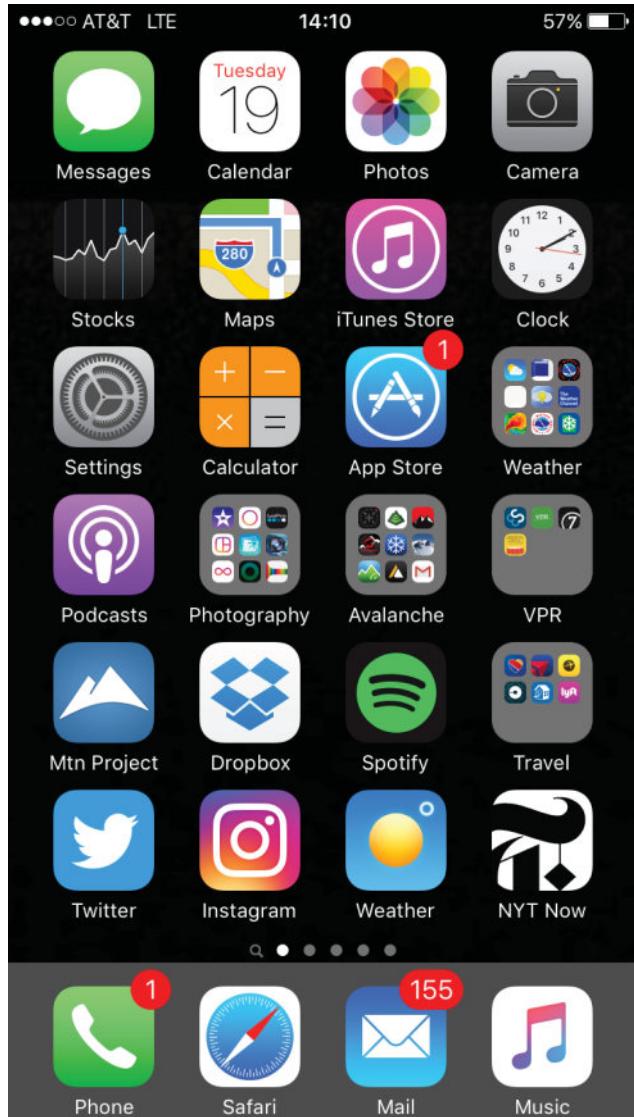
- Usually mouse, keyboard, and monitor
- Icons represent files, programs, actions, etc.
- Various mouse buttons over objects in the interface cause various actions providing information, options, execute function, open directory
- Invented at Xerox PARC

■ 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 (e.g., CDE, KDE, GNOME)



Touchscreen Interfaces



■ Touchscreen devices require new interfaces

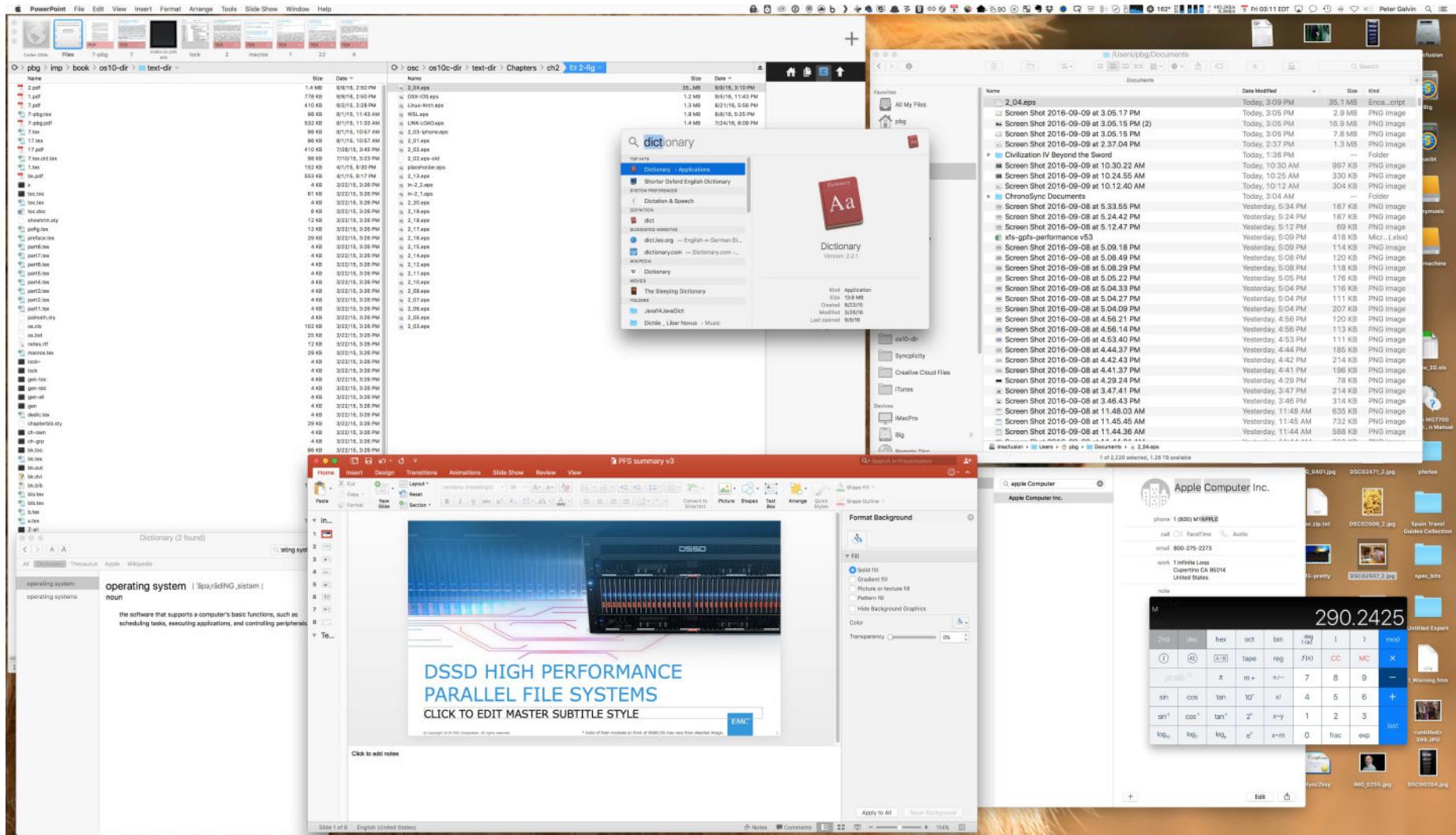
- Mouse not possible or not desired
- Actions and selection based on gestures
- Virtual keyboard for text entry

■ Voice commands





The Mac OS X GUI





System Calls

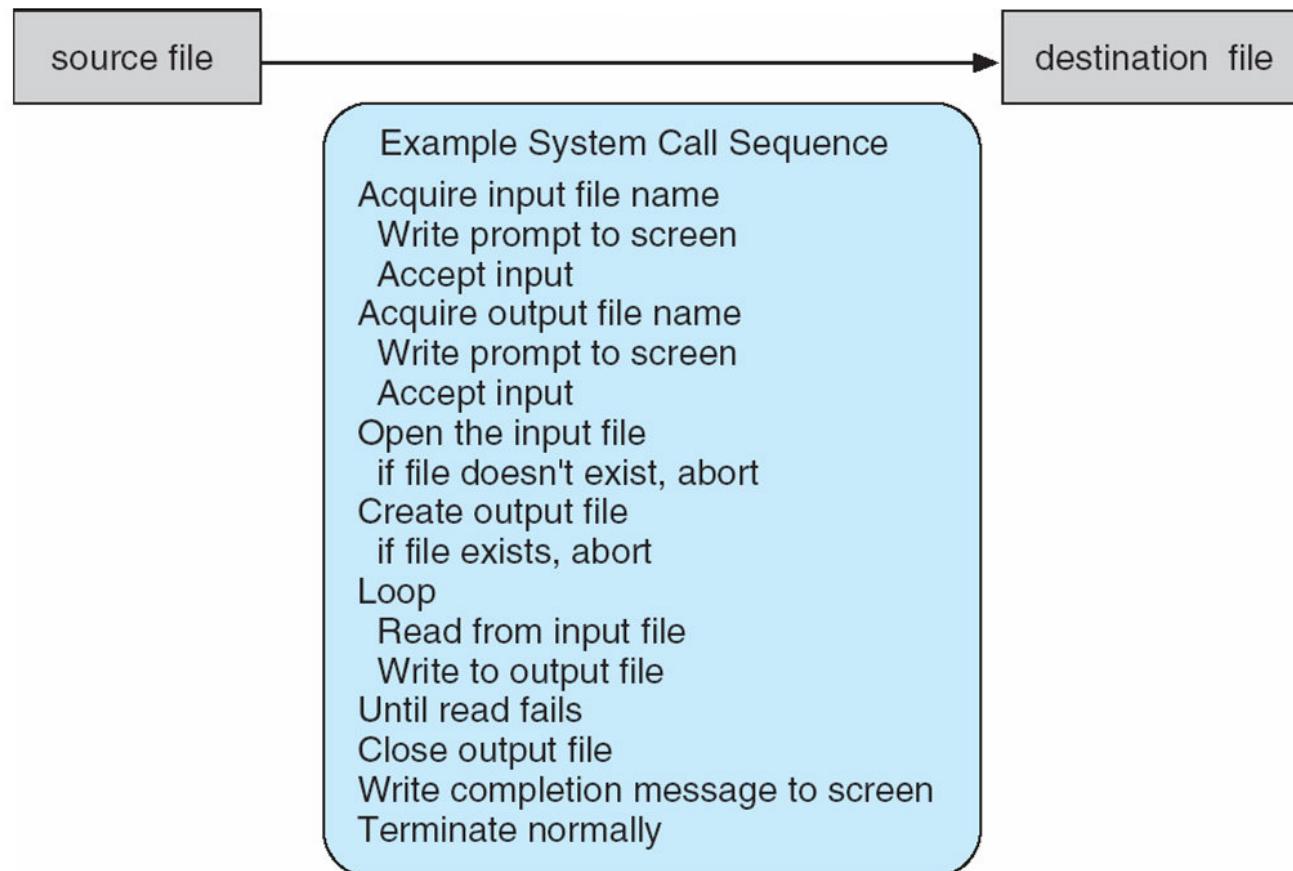
- *Programming interface* to the services provided by the OS
- Typically written in a high-level language (e.g., C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use
- Three most common APIs are **Win32 API** for Windows,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

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



Example of System Calls

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





Example of Standard API

EXAMPLE OF STANDARD API

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

```
man read
```

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

```
#include <unistd.h>

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

return function parameters
value name

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

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

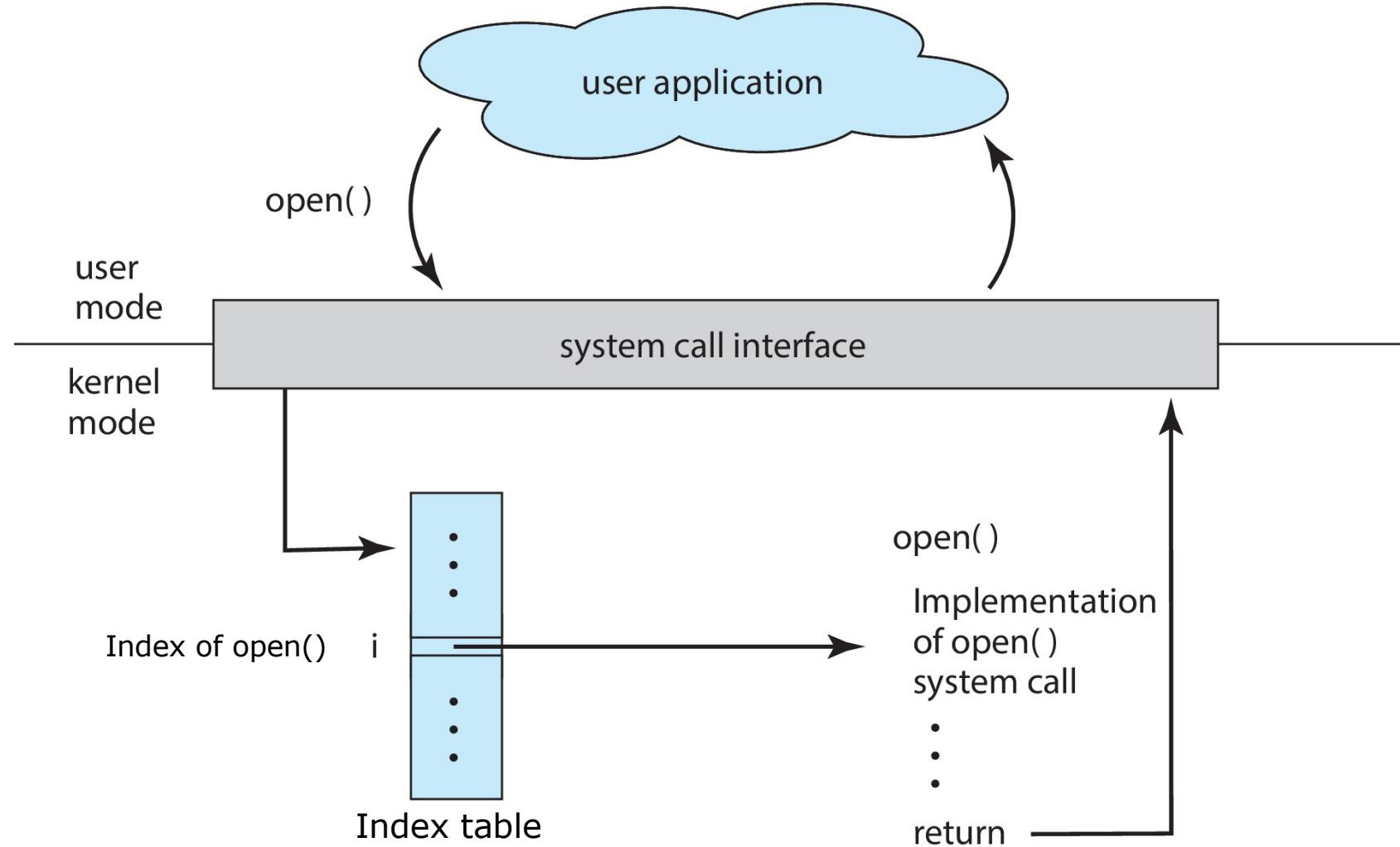
On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns -1.



- Typically, *a number associated with each system call*
 - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
 - Just needs to obey API and understand what OS will do as a result call
 - Most details of OS interface are hidden from programmer by API
 - ▶ Managed by **run-time support library** (set of functions built into libraries included with compiler)



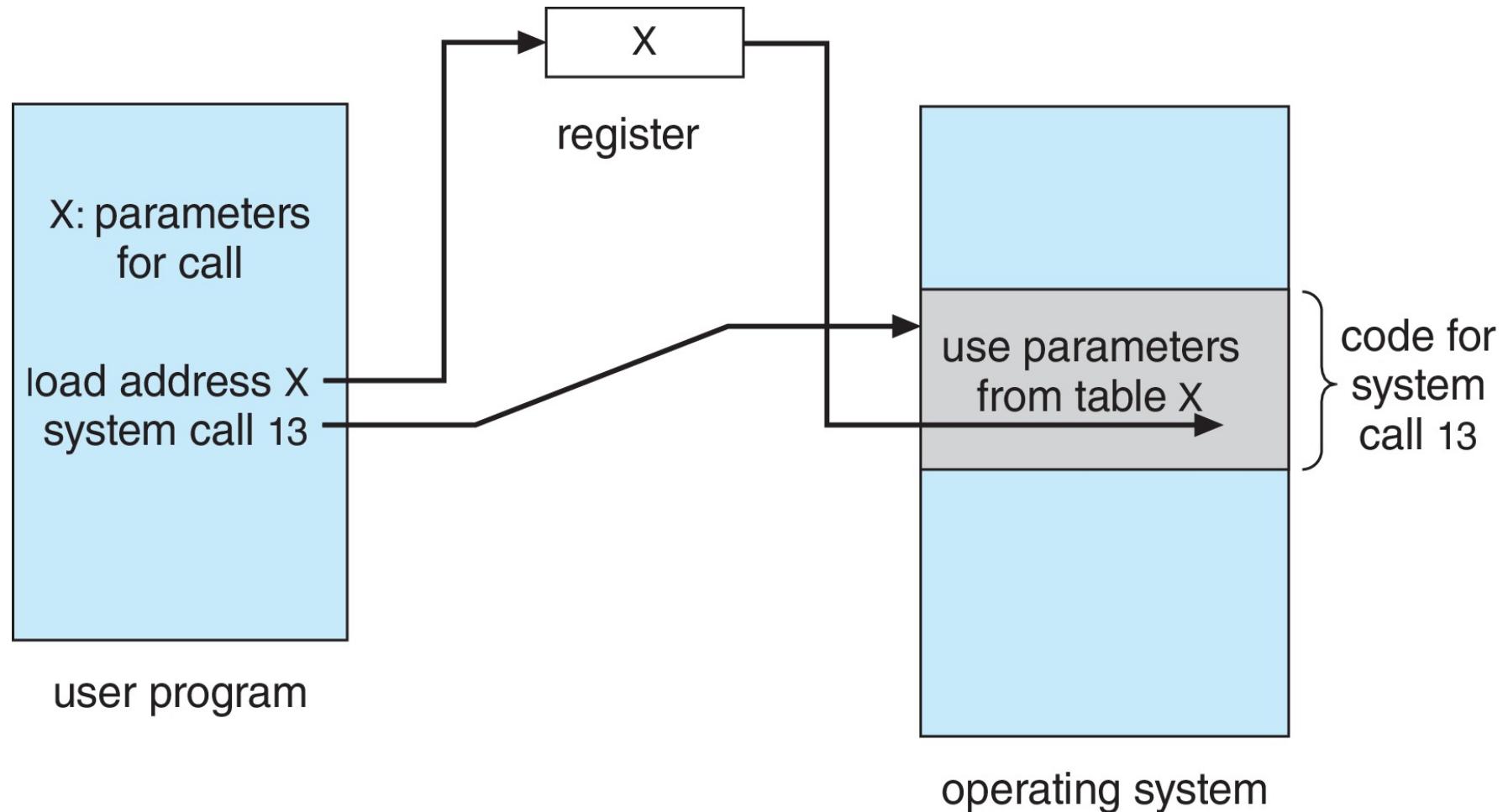
API – System Call – OS Relationship



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



Parameter Passing via Table



■ Process control

- create process, terminate process
- end, abort process execution
- 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



■ File management

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

■ Device management

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



■ Information maintenance

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

■ Communications

- create, delete communication connection
- send, receive messages if using *message passing model* to *host name* or *process name*
- *Shared-memory model* create and gain access to memory regions
- transfer status information
- attach and detach remote devices





Types of System Calls (Cont.)

■ Protection

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



Examples of Windows and Unix System Calls

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

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

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

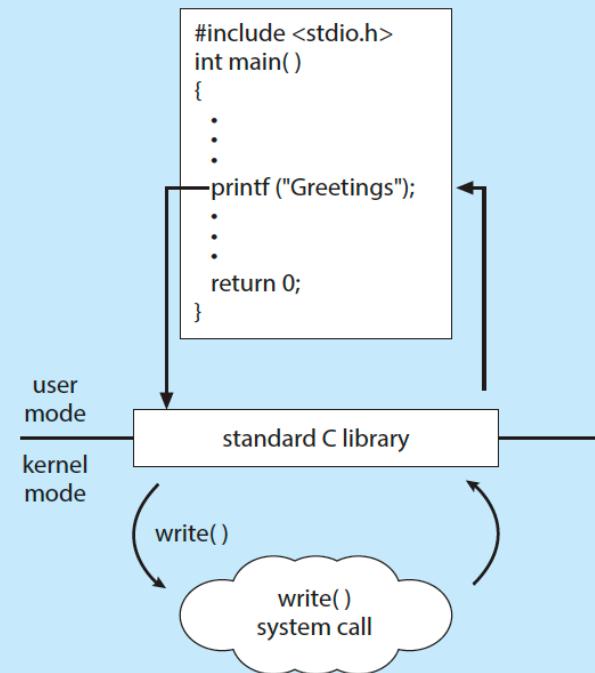


Standard C Library Example

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

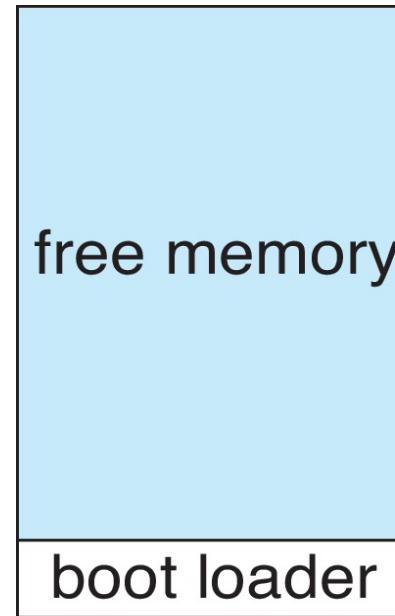
THE STANDARD C LIBRARY

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



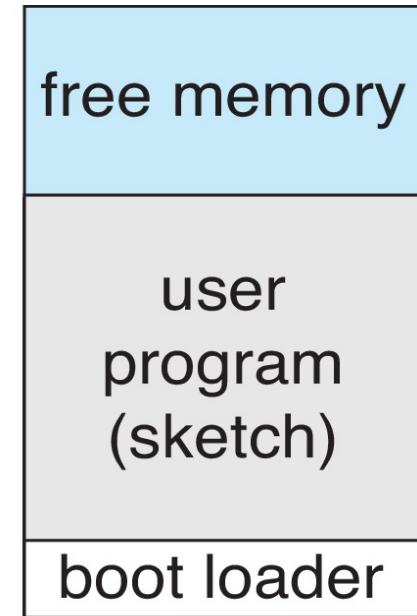
Example: Arduino

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



(a)

At system startup



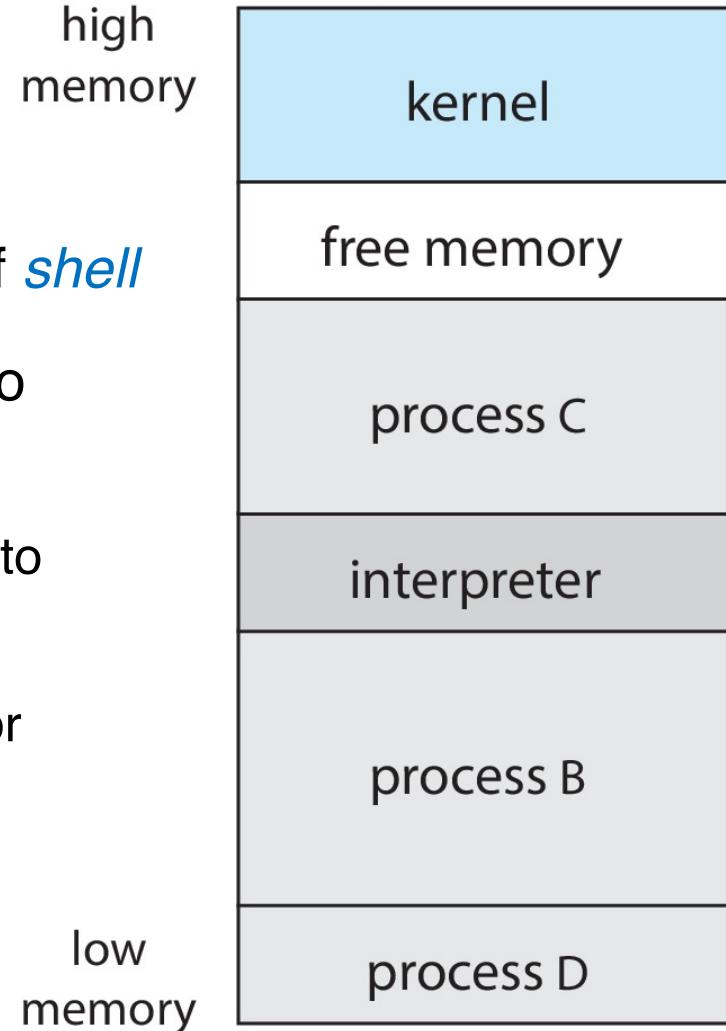
(b)

running a program



Example: FreeBSD

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



System Services

- System programs provide a convenient environment for *program development* and *execution*. They can be divided into:
 - File manipulation
 - Status information sometimes stored in a file
 - Programming language support
 - Program loading and execution
 - Communications
 - Background services
 - Application programs
- Most users' view of the operation system is defined by *system programs*, not the actual *system calls*





System Services (Cont.)

- Some of *system services* 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 *information* (e.g., 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



■ 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* are sometimes provided

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

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

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





System Services (Cont.)

■ Background Services

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

■ Application programs

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

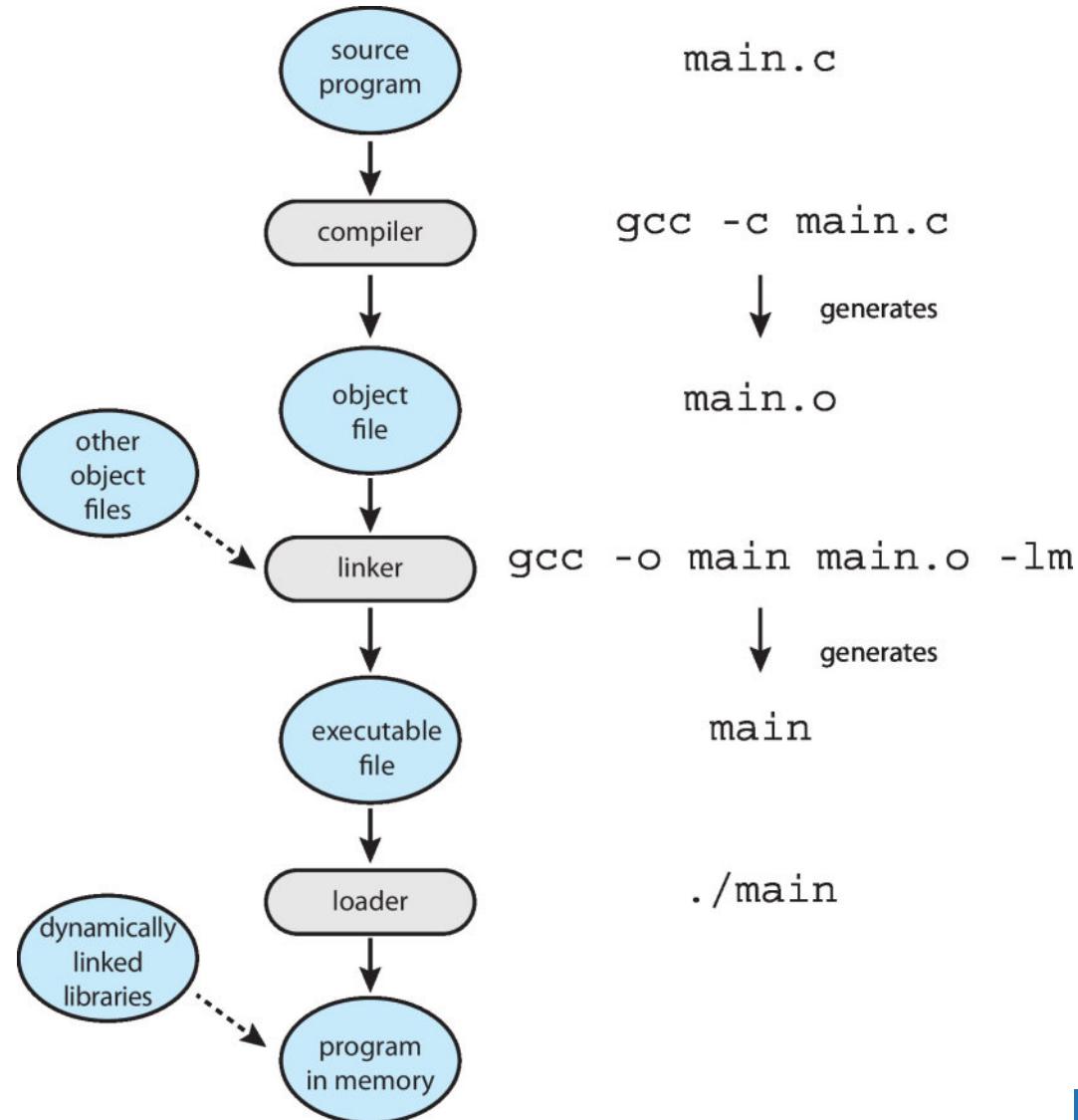


Linkers and Loaders

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



The Role of the Linker and Loader



Why Applications are Operating System Specific

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



- 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



- 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 (e.g., timer)
- Specifying and designing an OS is highly creative task of *software engineering*



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



Operating System Structure

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



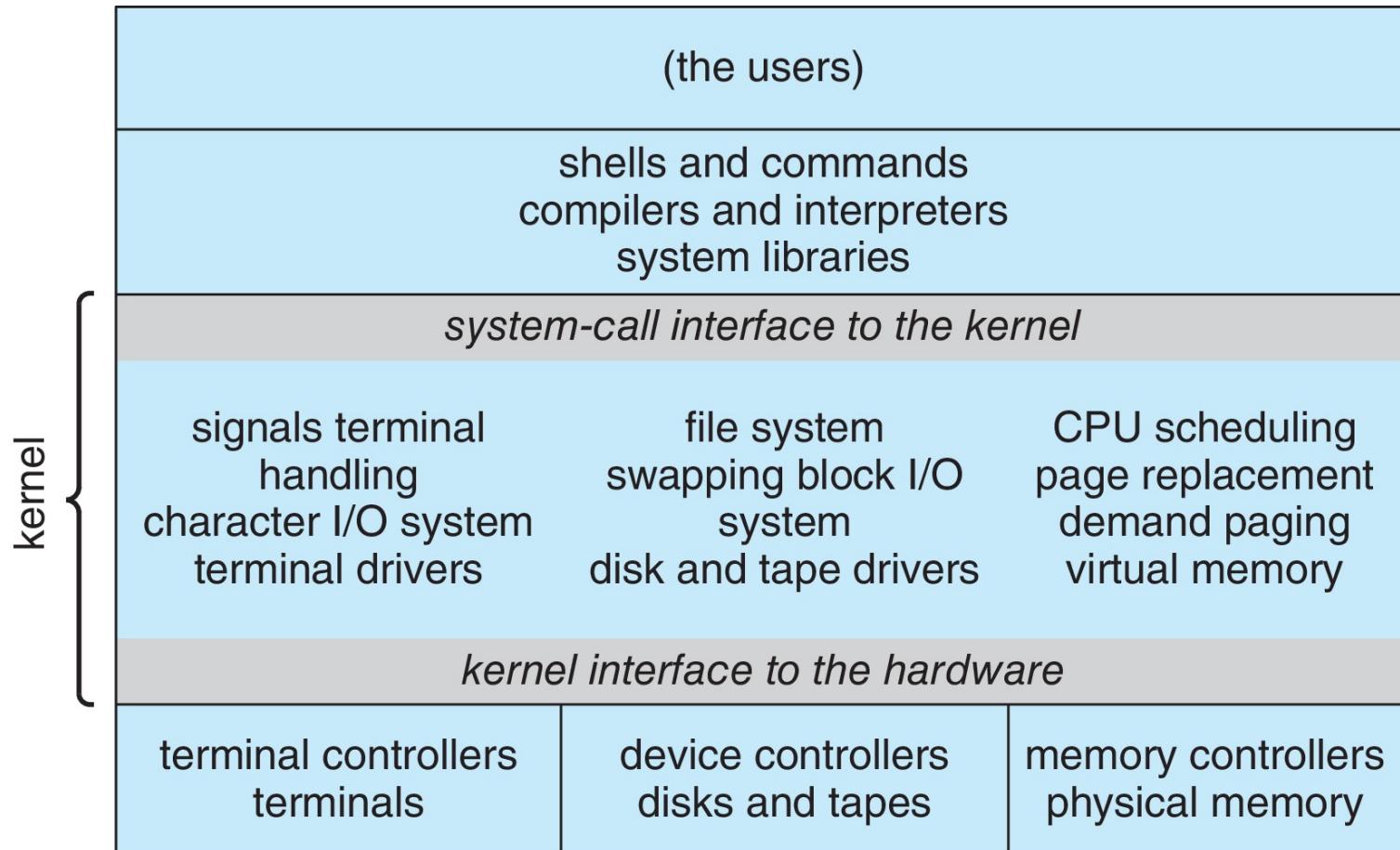
Monolithic Structure – Original UNIX

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

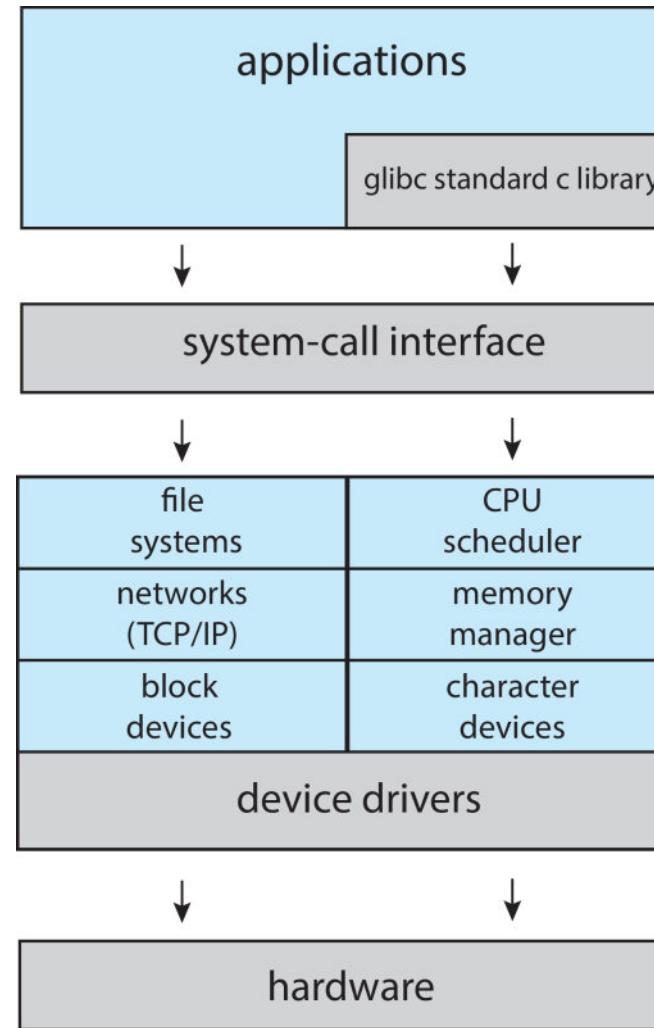


Traditional UNIX System Structure

- Beyond simple but not fully layered

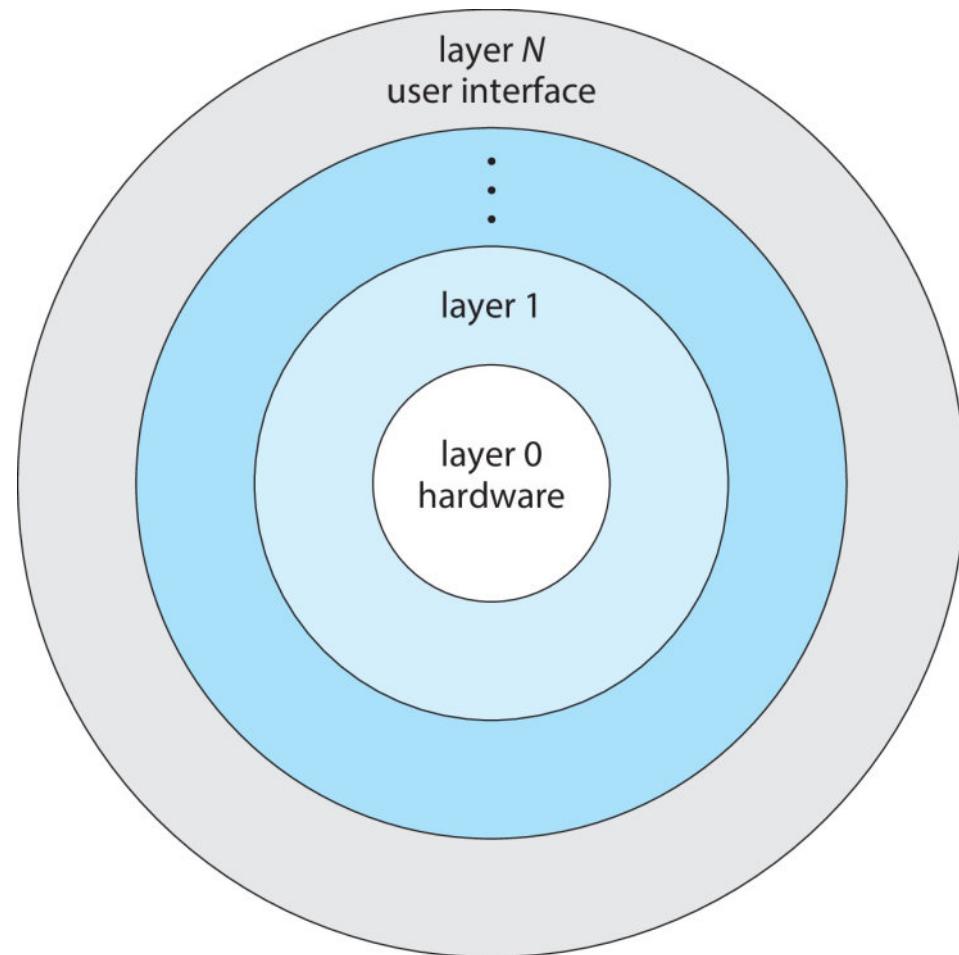


■ Monolithic plus modular design



Layered Approach

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

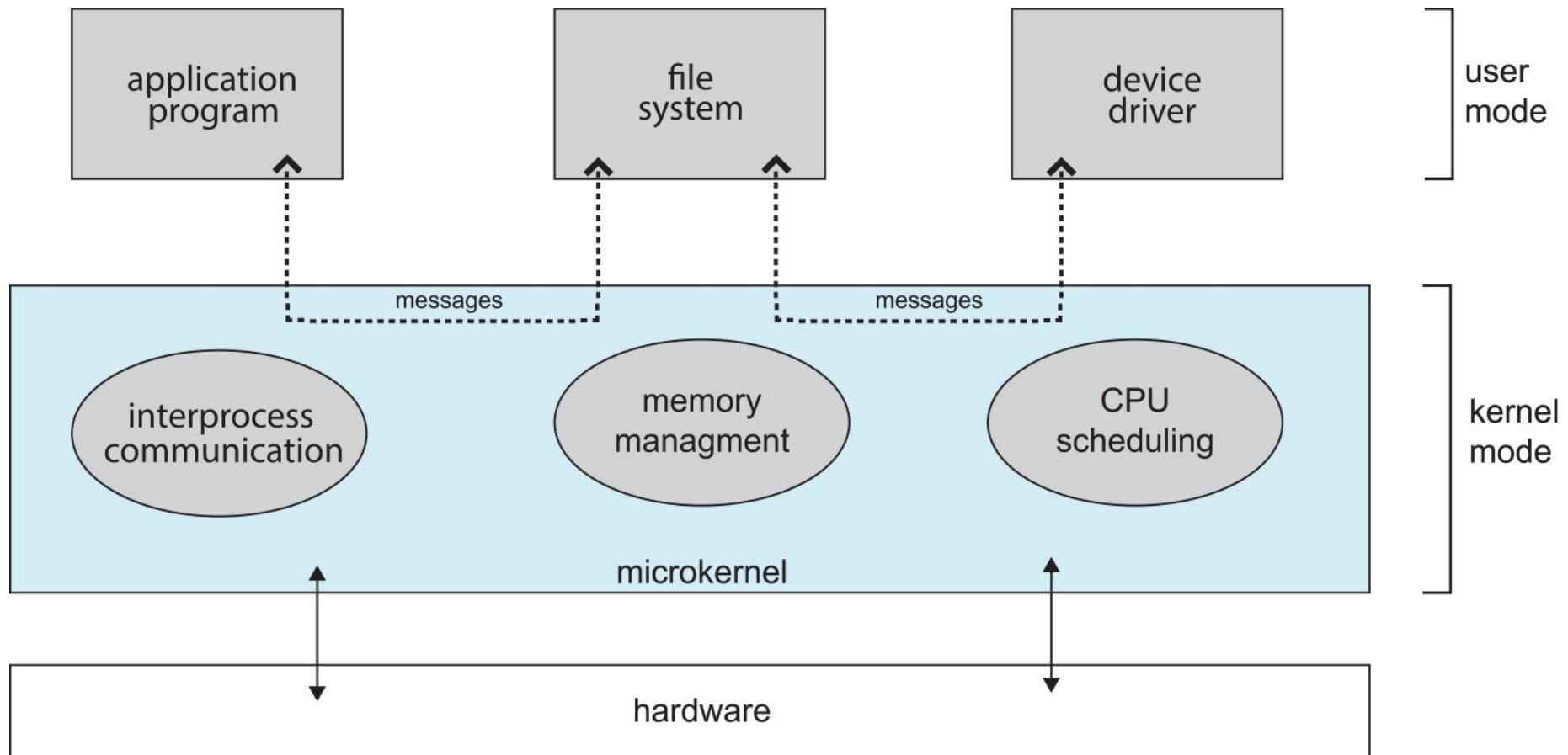


Microkernels

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



Microkernel System Structure





Modules

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

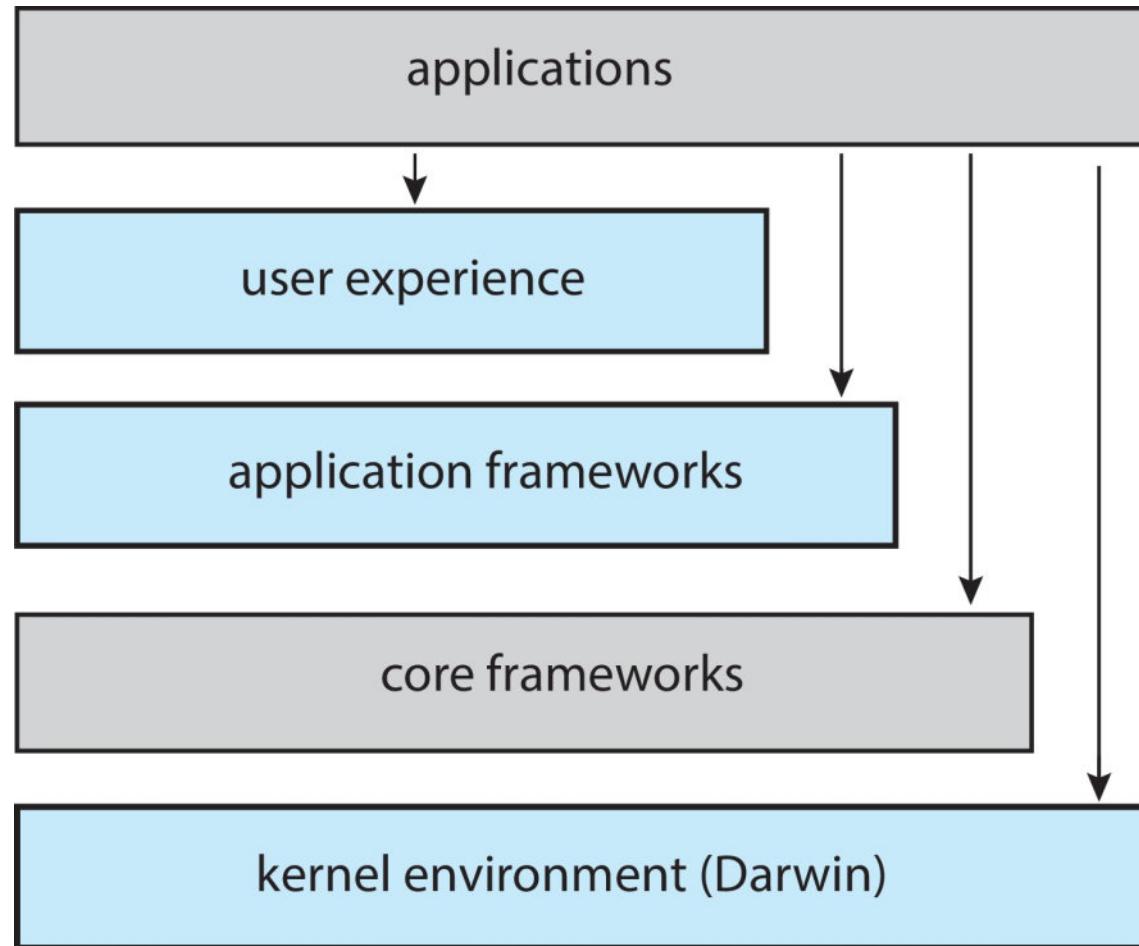


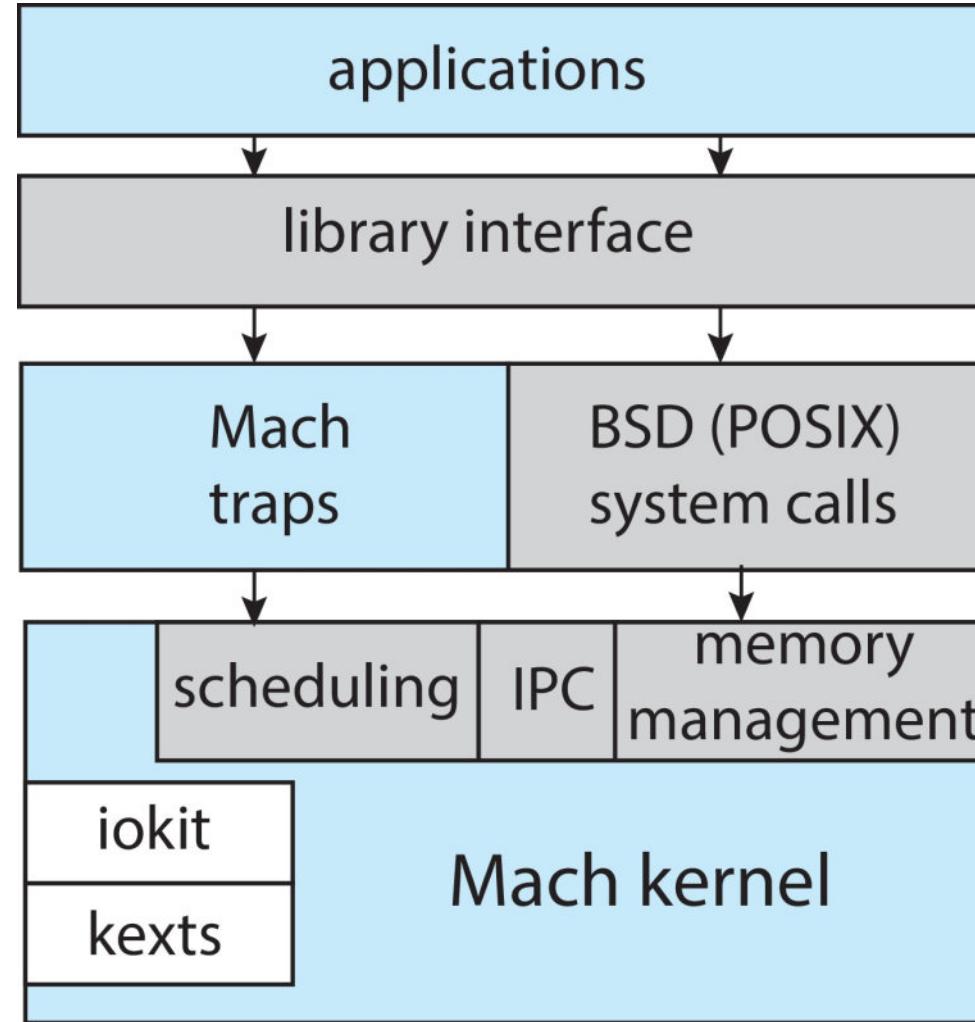
Hybrid Systems

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



macOS and iOS Structure





■ 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

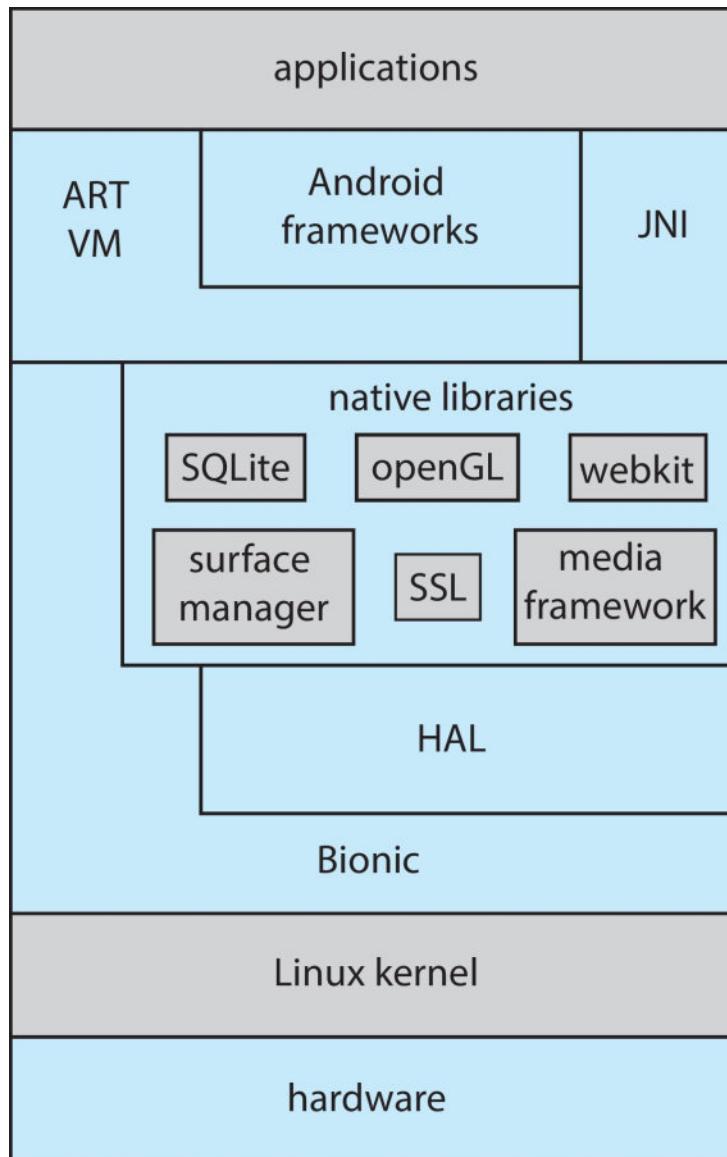
Core OS



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



Android Architecture





Building and Booting an Operating System

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





Building and Booting Linux

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



- When power on system, execution starts at a fixed memory location
- Operating system must be made available to hardware so hardware can start it
 - ▶ Small piece of code – **bootstrap loader**, **BIOS**, stored in **ROM** or **EEPROM** locates the kernel, loads it into memory, and starts it
 - ▶ Sometimes two-step process where **boot block** at fixed location loaded by ROM code, which loads bootstrap loader from disk
 - ▶ Modern systems replace BIOS with **Unified Extensible Firmware Interface (UEFI)**
- Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then *running*
- Boot loaders frequently allow various boot states





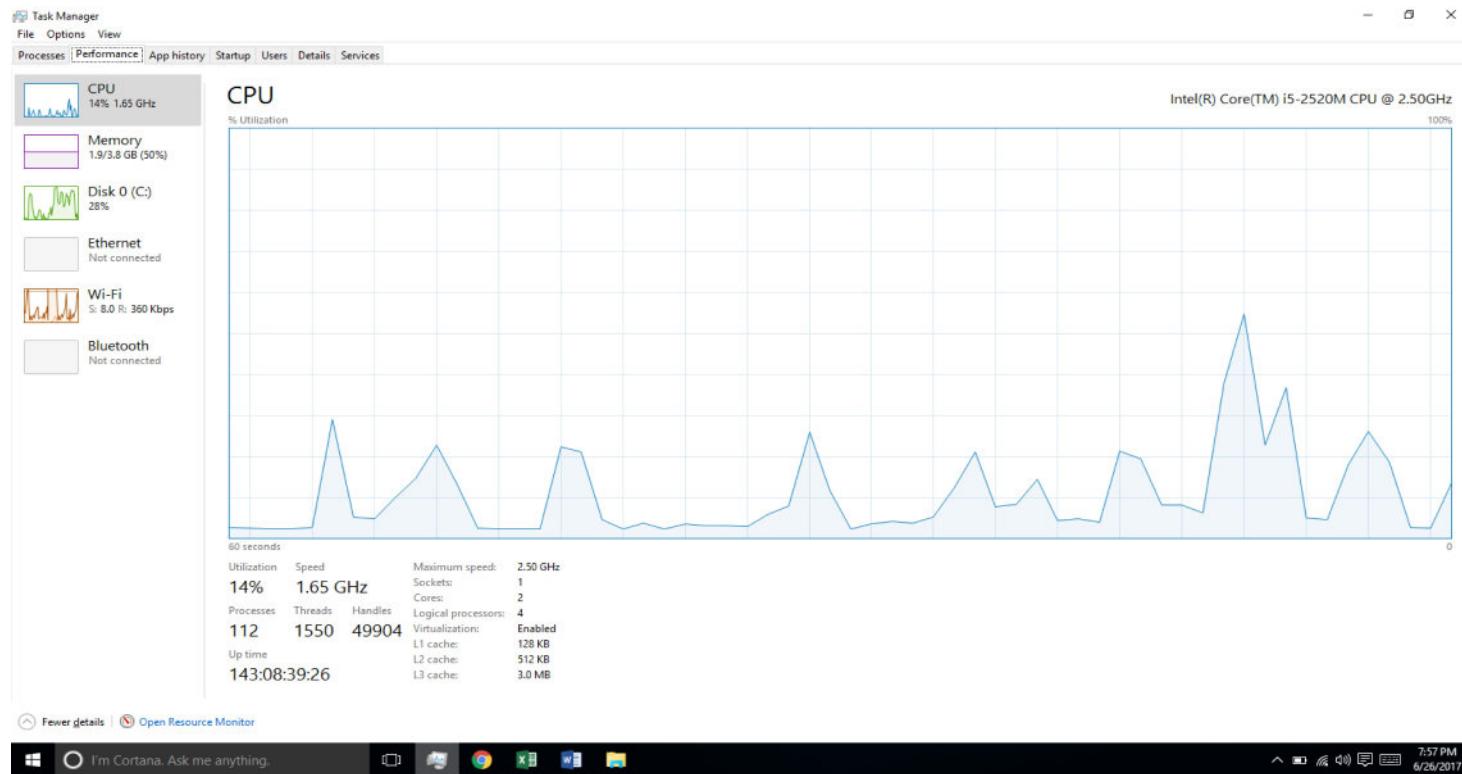
Operating-System Debugging

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

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



Performance Tuning



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





Tracing

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

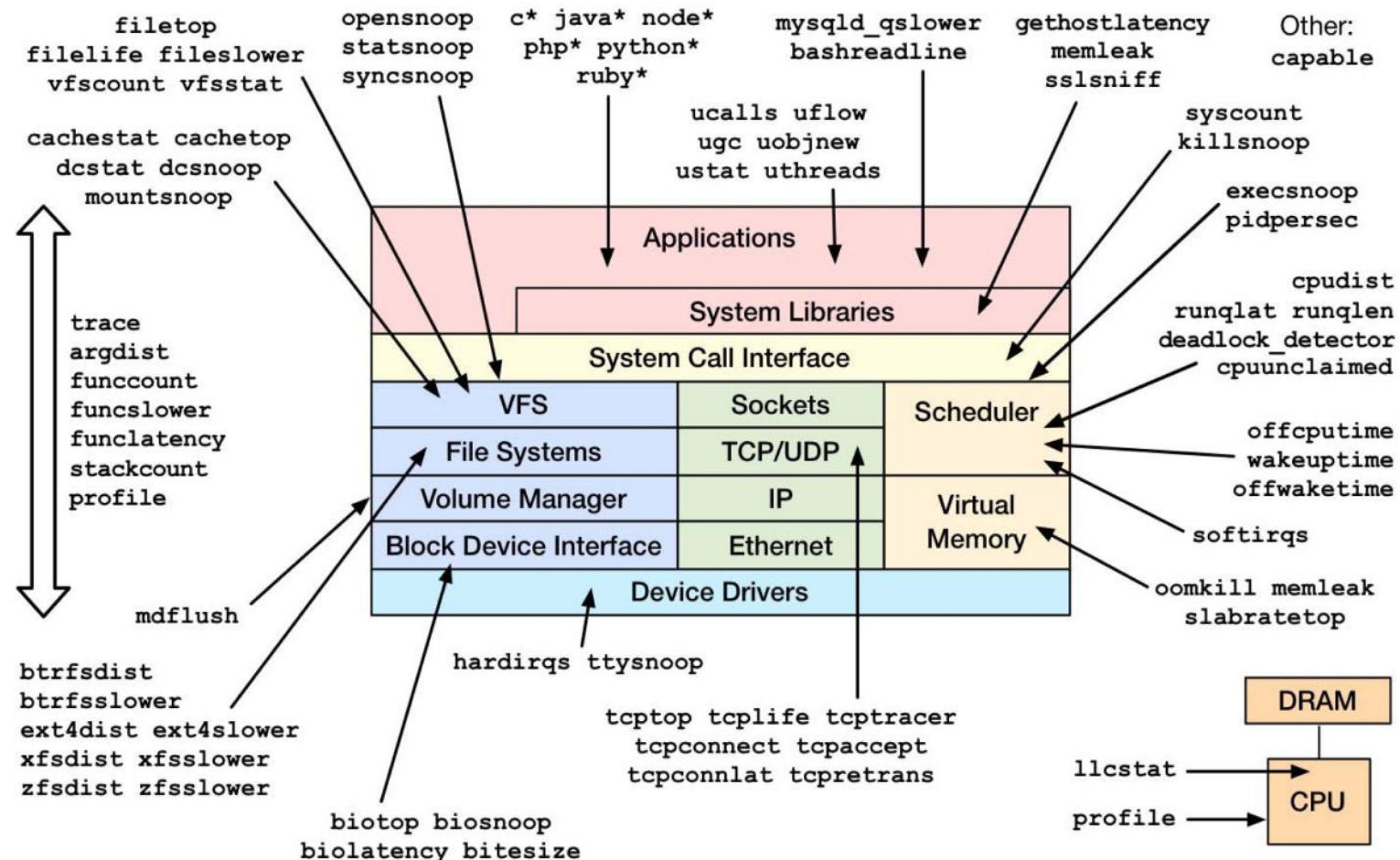


- Debugging interactions between user-level and kernel code nearly impossible without toolset that understands both and an instrument their actions
- **BPF Compiler Collection (BCC)** is a rich toolkit providing tracing features for Linux
 - See also the original **DTrace**
- For example, `disksnoop.py`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

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

Linux BCC/BPF Tracing Tools

Linux bcc/BPF Tracing Tools



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



Summary

- An ***operating system*** provides an environment for the execution of programs by providing services to users and programs.
- The ***three primary approaches for interacting*** with an operating system are (1) command interpreters, (2) graphical user interfaces, and (3) touch-screen interfaces.
- ***System calls*** provide an interface to the services made available by an operating system. Programmers use a system call's application programming interface (API) for accessing system-call services.
- System calls can be divided into ***six major categories***: (1) process control, (2) file management, (3) device management, (4) information maintenance, (5) communications, and (6) protection.
- The ***standard C library*** provides the system-call interface for UNIX and Linux systems.



Summary (Cont.)

- Operating systems also include a ***collection of system programs*** that provide utilities to users.
- A ***linker*** combines several relocatable object modules into a single binary executable file. A ***loader*** loads the executable file into memory, where it becomes eligible to run on an available CPU.
- There are several reasons why ***applications are operating-system specific***. These include different binary formats for program executables, different instruction sets for different CPUs, and system calls that vary from one operating system to another.
- An operating system is designed with specific goals in mind. These goals ultimately determine the ***operating system's policies***. An operating system implements these policies through specific mechanisms.



Summary (Cont.)

- A ***monolithic operating system*** has no structure; all functionality is provided in a single, static binary file that runs in a single address space. Although such systems are difficult to modify, their primary benefit is efficiency.
- A ***layered operating system*** is divided into a number of discrete layers, where the bottom layer is the hardware interface and the highest layer is the user interface.
- Although layered software systems have had some success, this approach is generally not ideal for designing operating systems due to performance problems.
- The ***microkernel*** approach for designing operating systems uses a minimal kernel; most services run as user-level applications. Communication takes place via message passing.



Summary (Cont.)

- A ***modular approach*** for designing operating systems provides operating-system services through modules that can be loaded and removed during run time. Many contemporary operating systems are constructed as hybrid systems using a combination of a monolithic kernel and modules.
- A ***boot loader*** loads an operating system into memory, performs initialization, and begins system execution.
- The ***performance*** of an operating system can be monitored using either counters or tracing. Counters are a collection of system-wide or per-process statistics, while tracing follows the execution of a program through the operating system.



End of Chapter 2



Chapter 3: Processes



What Is an
OPERATING SYSTEM (OS)
and How Does It Work

CLEVERISM.COM



Chapter 3: Processes

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





Objectives

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



Process Concept

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



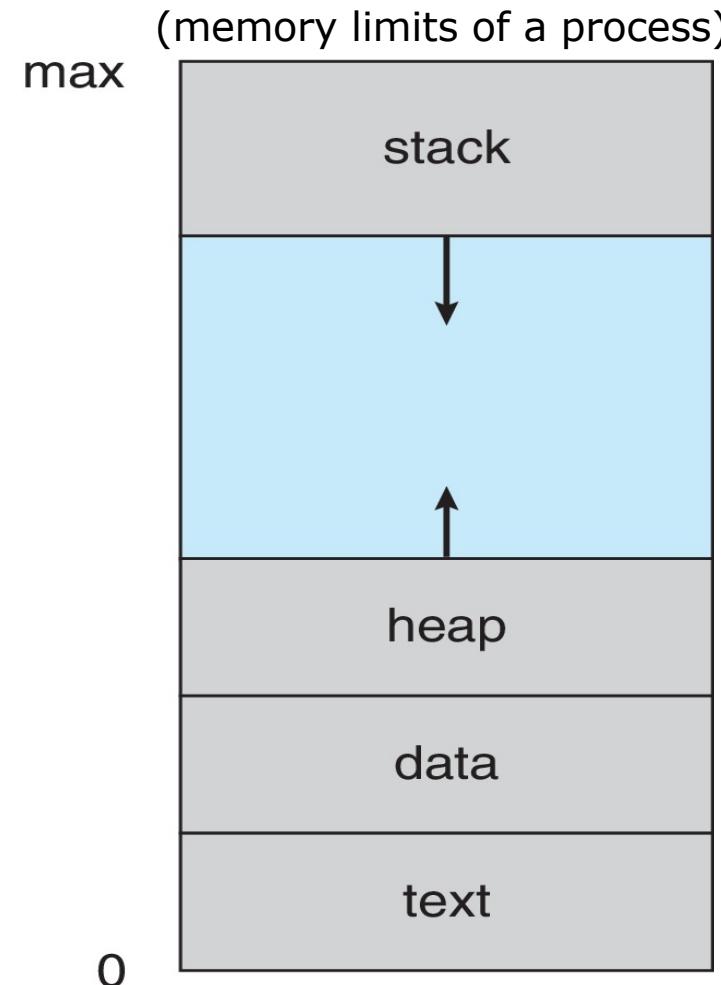


Process Concept (Cont.)

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



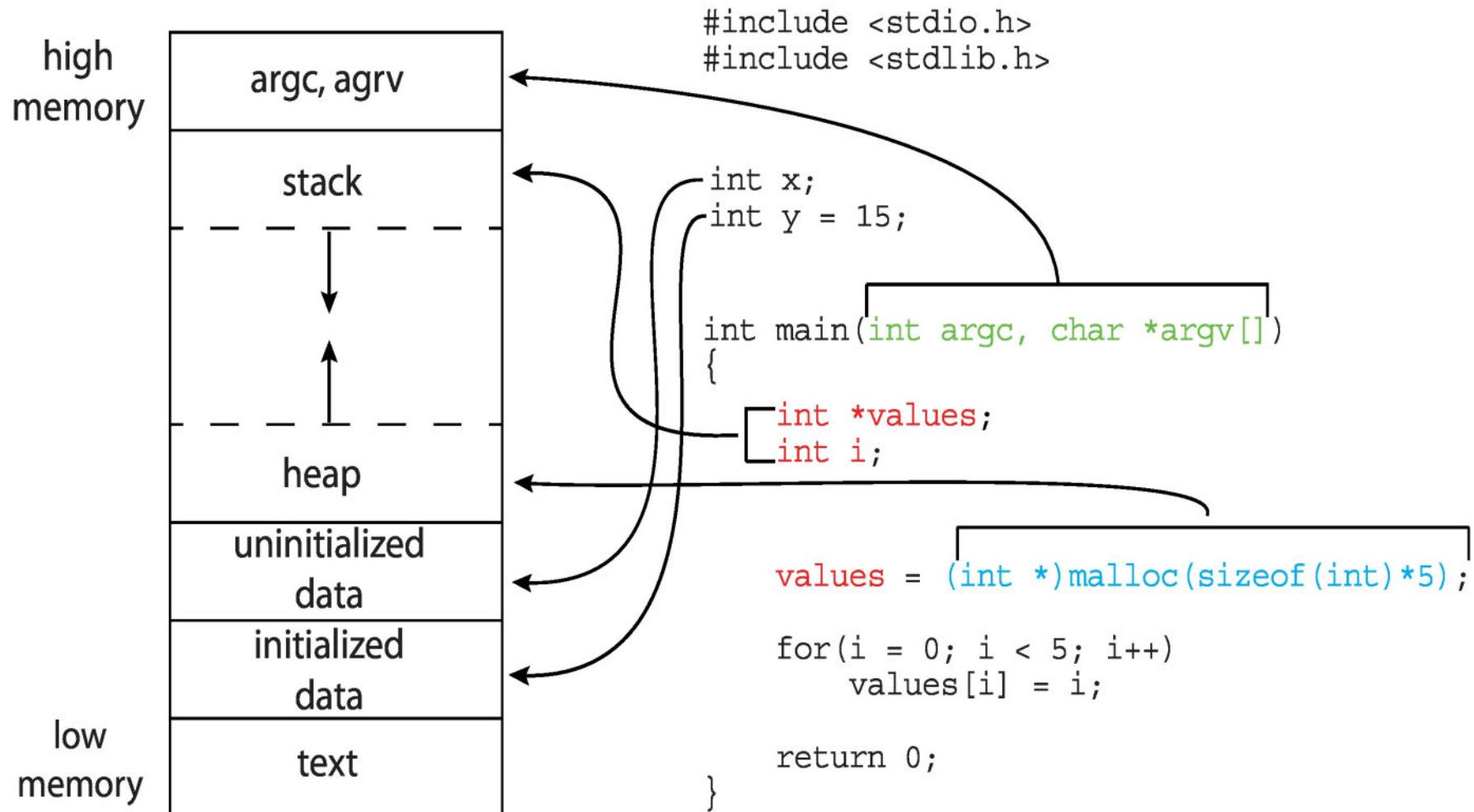
Process in Memory



#size <pid>



Memory Layout of a C Program





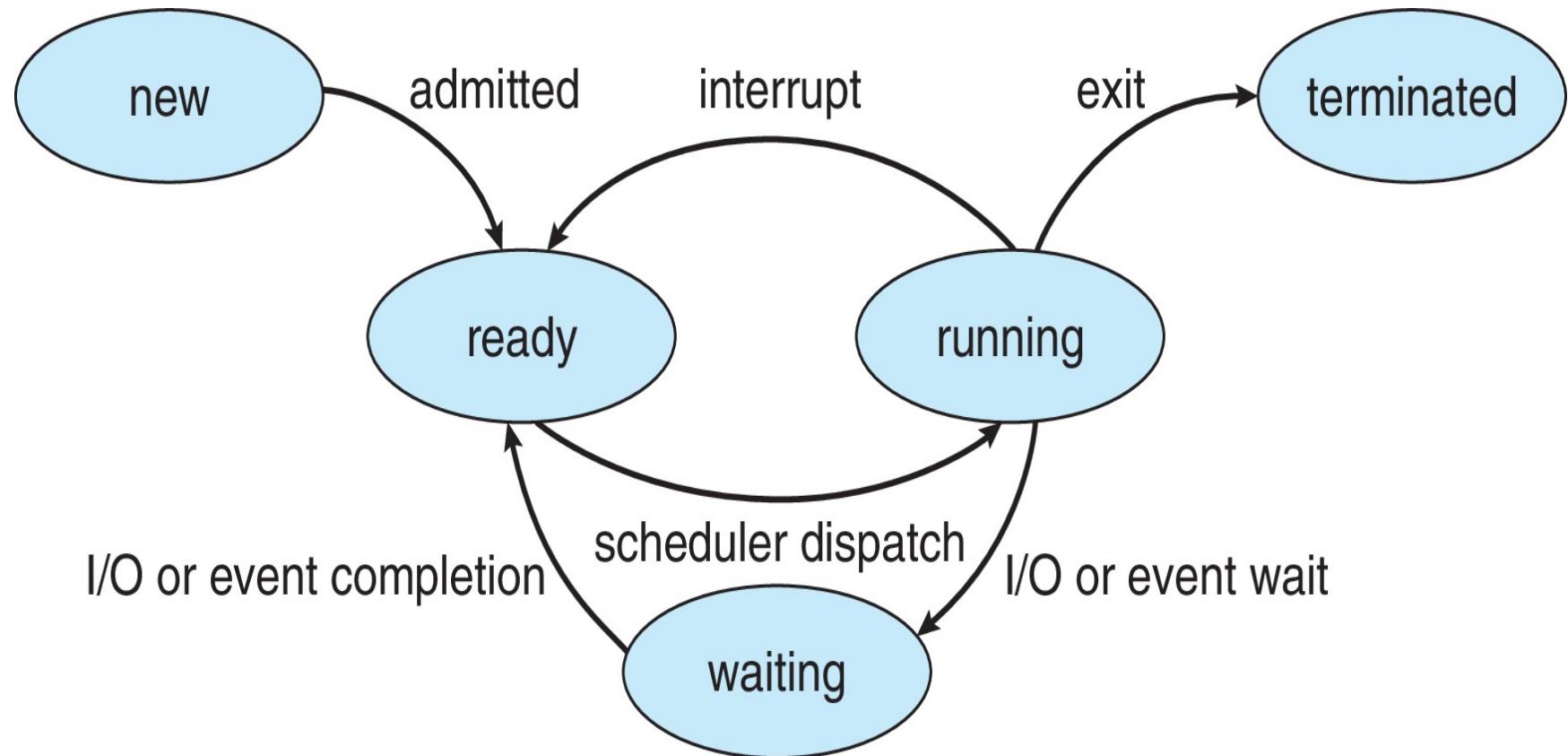
Process State

■ As a process executes, it changes *state*

- *New* – The process is being created
- *Running* – Instructions are being executed
- *Waiting* – The process is waiting for some event to occur
- *Ready* – The process is waiting to be assigned to a processor
- *Terminated* – The process has finished execution



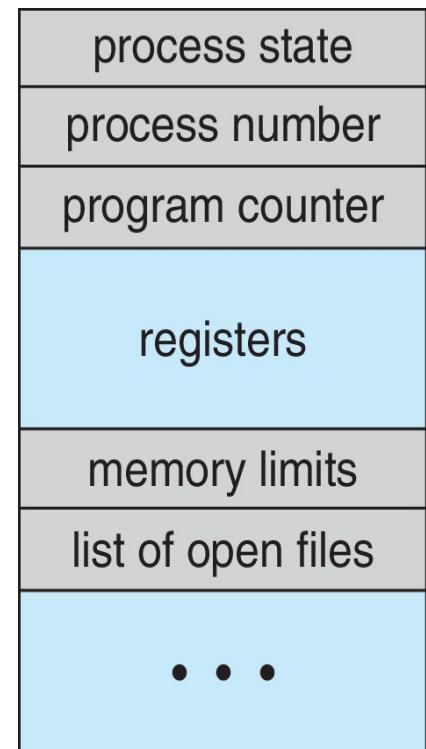
Diagram of Process State



Process Control Block (PCB)

- **Process Control Block (PCB)** – Information associated with each process, also called **Task Control Block (TCB)**, includes:

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





Threads

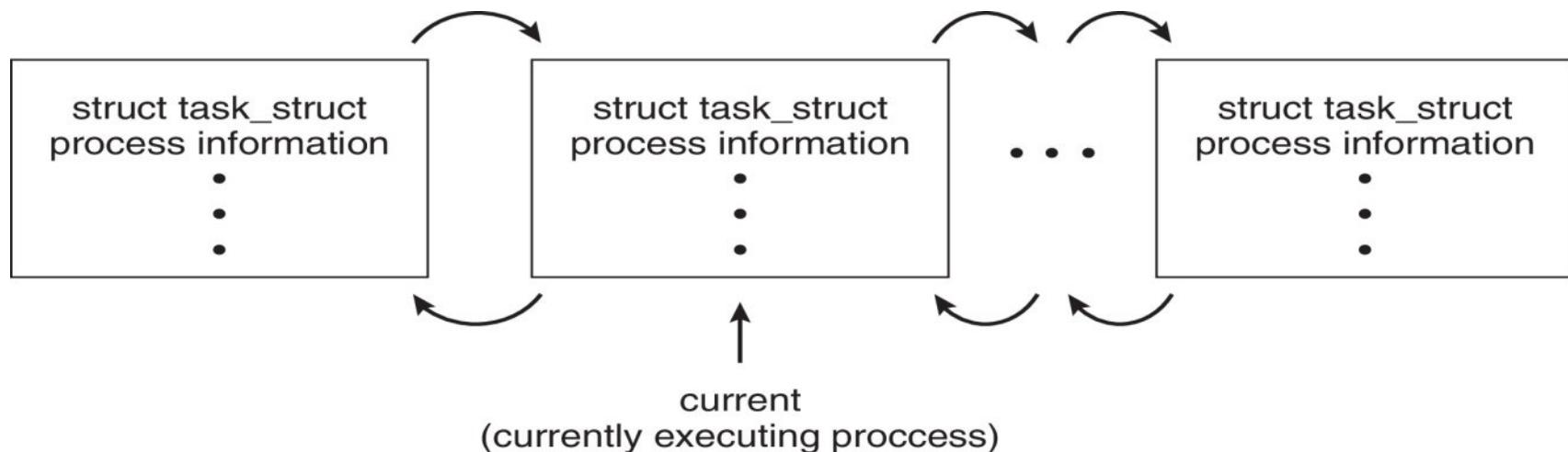
- So far, process has a *single thread* of execution
- Consider having *multiple program counters per process*
 - Multiple locations can execute at once
 - ▶ Multiple threads of control → *threads*
- Must then have *storage for thread* details
- Multiple program counters in PCB

(Explore in detail in Chapter 4)



■ Represented by the C structure `task_struct`

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



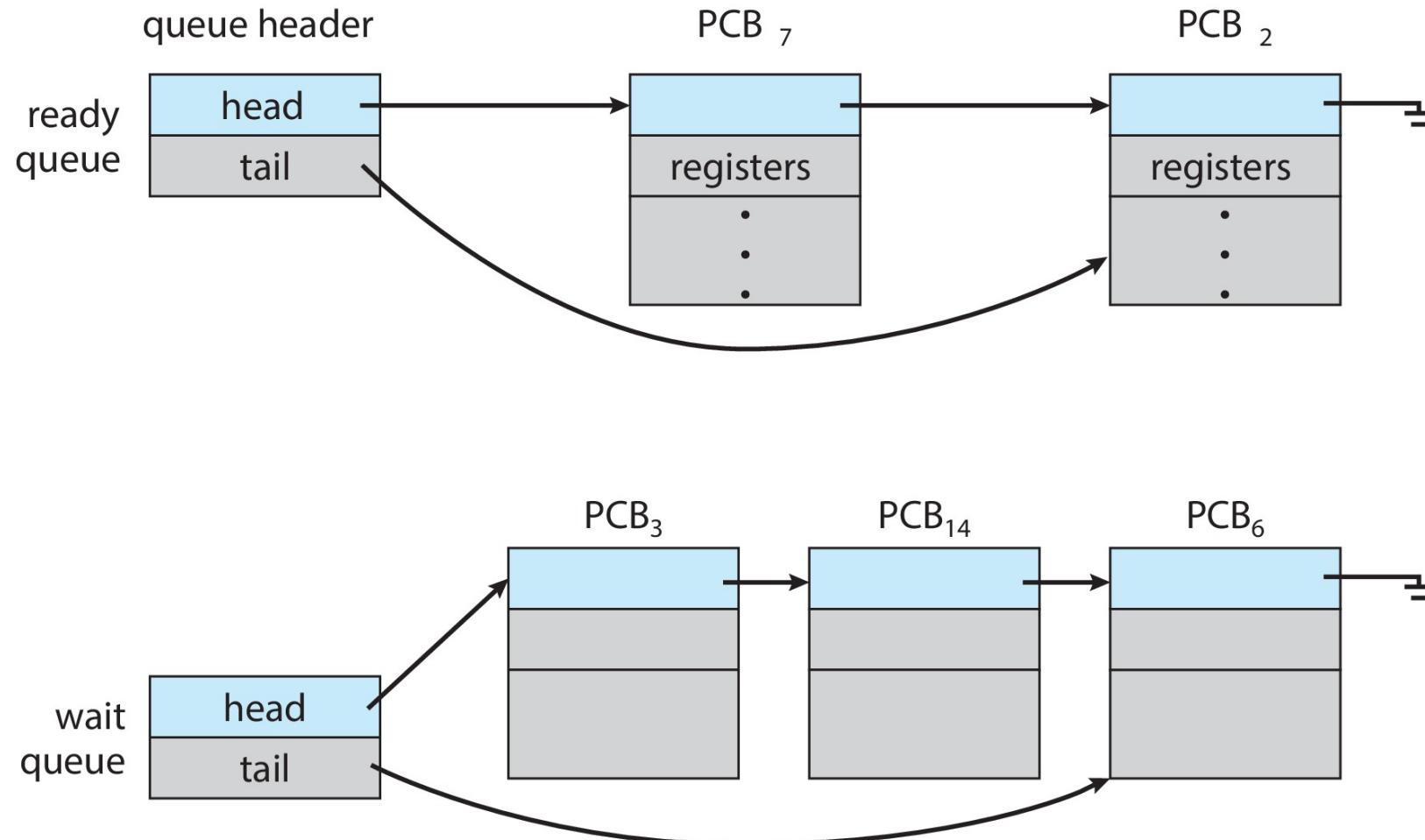


Process Scheduling

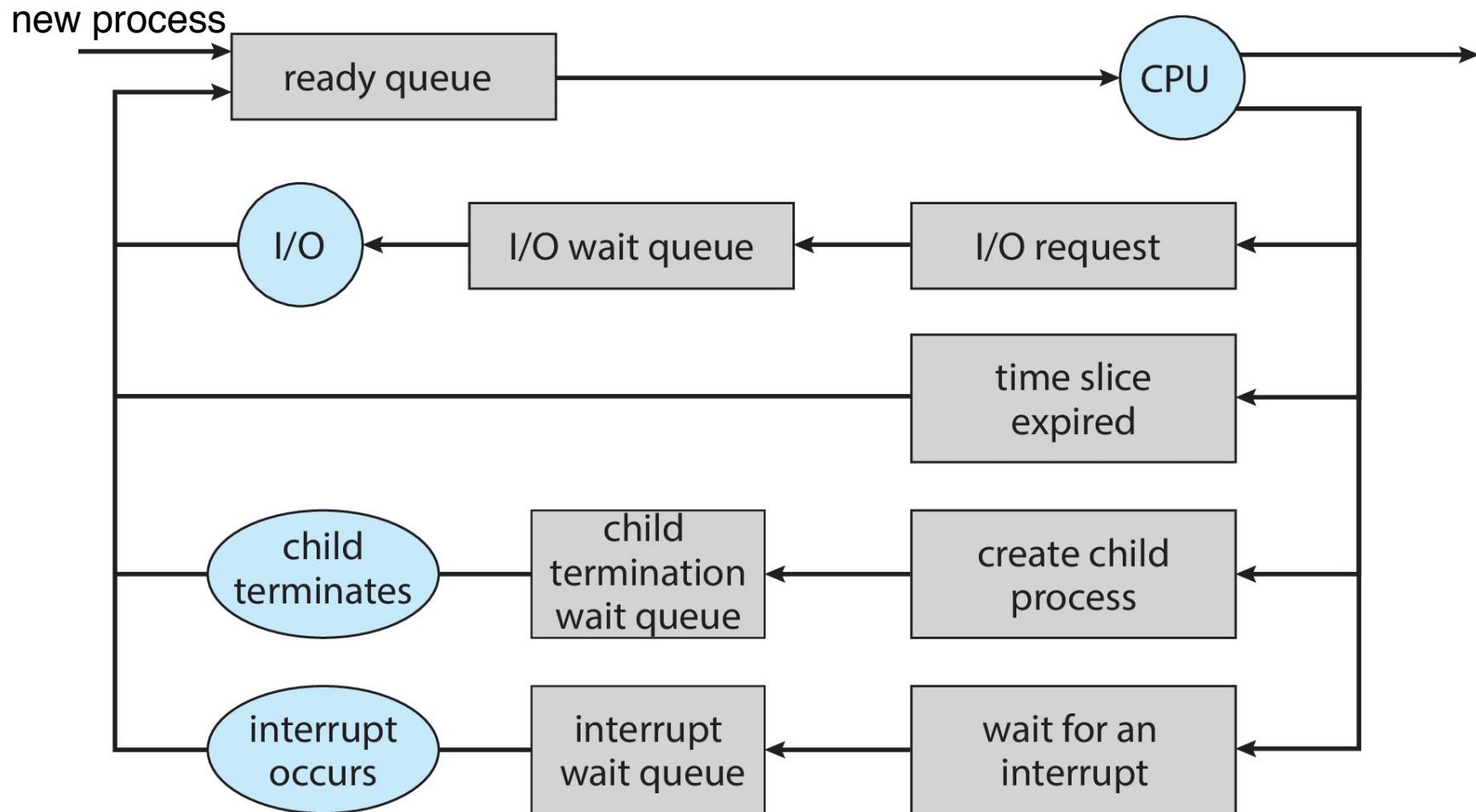
- Maximize CPU use → quickly switch processes onto CPU core
- *Process scheduler* selects among available (ready) processes for next execution on CPU core
- Maintains *scheduling queues* of processes
 - *Ready queue* – set of all processes residing in main memory, ready and waiting to execute
 - *Wait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



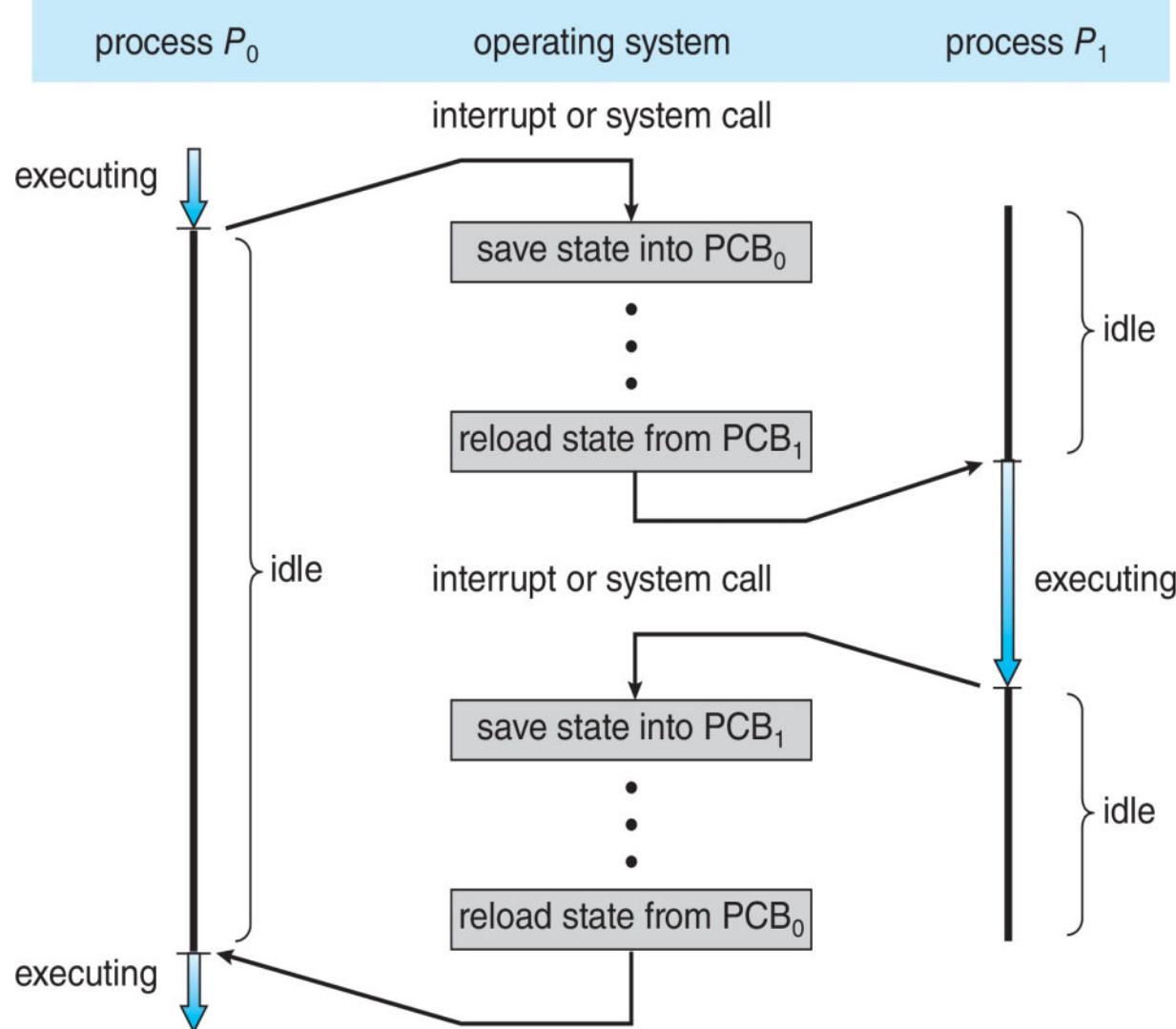
Ready and Wait Queues



Representation of Process Scheduling



CPU Switch from Process to Process



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



Context Switch

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



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



Operations on Processes

■ System must provide mechanisms for:

- process creation
- process termination



Process Creation

- *Parent process* create *children processes*, which, in turn create other processes, forming a *tree of processes*
- Process identified and managed via a **Process Identifier (pid)**
- **Resource sharing options**
 - Parent and children share *all* resources
 - Children share *subset* of parent's resources
 - Parent and child share *no* resources





Process Creation (Cont.)

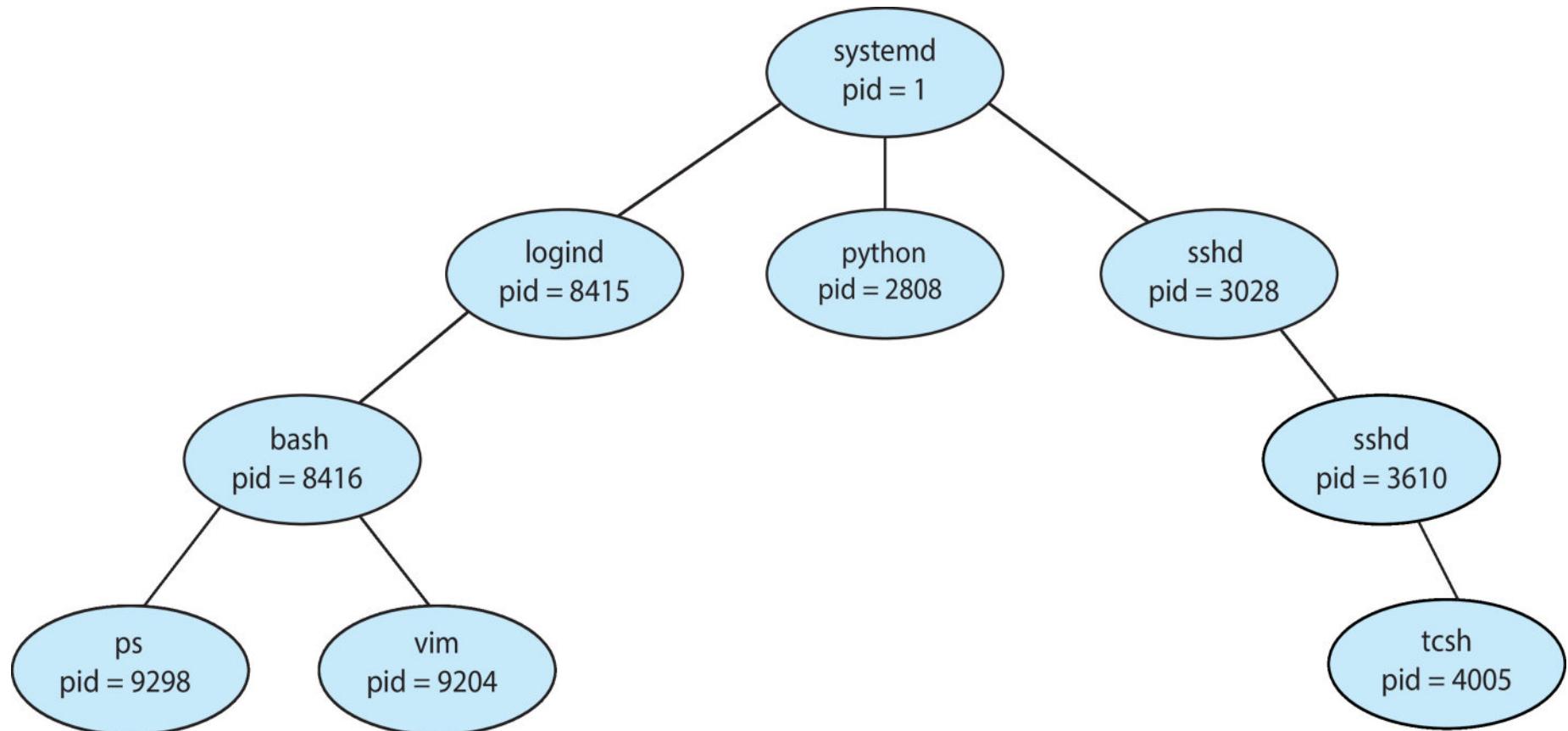
■ Execution options

- Parent and children execute concurrently
- Parent waits until children terminate

■ Address space

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

A Tree of Processes in Linux

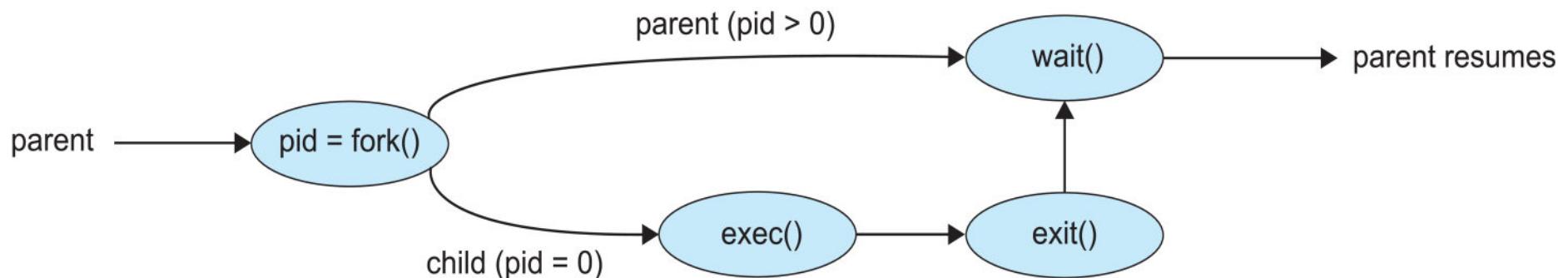


#pstree



■ UNIX examples

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





C Program Forking A Separate Process

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

int main()
{
    pid_t pid;

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

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

    return 0;
}
```





Creating a Separate Process via Windows API

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

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

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

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

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



Process Termination

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



Process Termination (Cont.)

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

```
pid = wait(&status);
```
- If no parent waiting (did not invoke `wait()`), process is a *zombie*
- If parent terminated without invoking `wait()` , process is an *orphan*



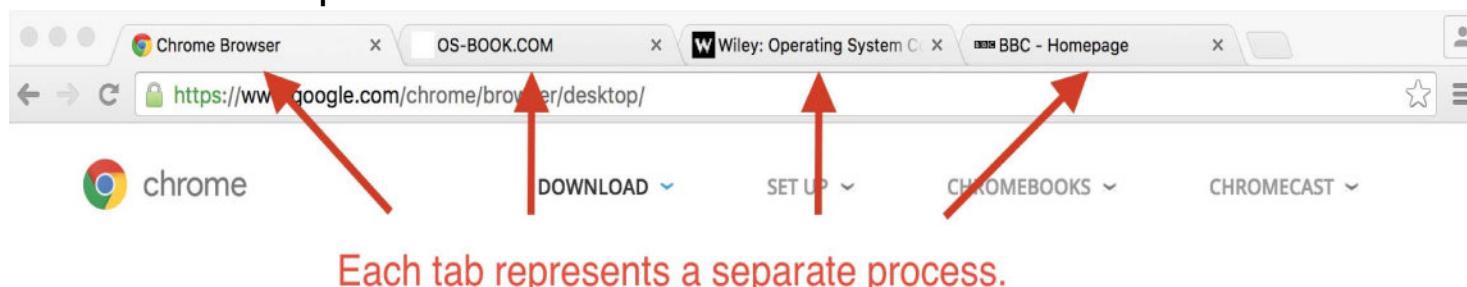


Importance Hierarchy of Android Process

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



- Many web browsers ran as a 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

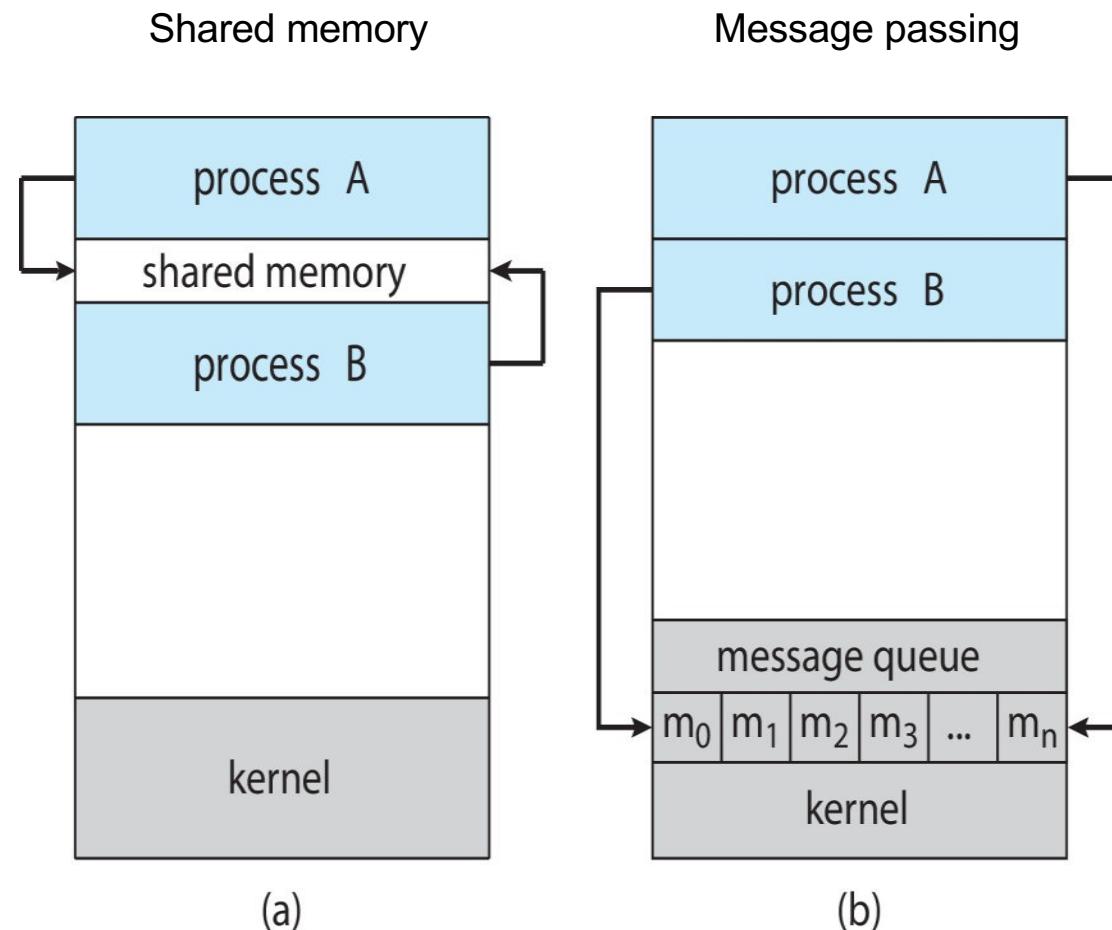


- Processes within a system may be *independent* or *cooperating*
 - *Independent process* does not share data with any other processes executing in the system
 - *Cooperating process* can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
 - Information sharing
 - Computation speed-up
 - Modularity
 - Convenience
- Cooperating processes need **Inter-Process communication (IPC)**



■ Two models of IPC

- *Shared memory*
- *Message passing*





Inter-Process Communication – Shared Memory

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

(Synchronization is discussed in great details in Chapters 6 & 7)



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



Bounded-Buffer – Shared-Memory Solution

■ Shared data

```
#define BUFFER_SIZE 10

typedef struct {

    . . .

} item;

item buffer[BUFFER_SIZE];

int in = 0;

int out = 0;
```

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



```
item next_produced;

while (true) {

    /* produce an item in next produced */

    while (((in + 1) % BUFFER_SIZE) == out)

        ; /* do nothing */

    buffer[in] = next_produced;

    in = (in + 1) % BUFFER_SIZE;

}
```



```
item next_consumed;

while (true) {
    while (in == out)

        ; /* do nothing */
    next_consumed = buffer[out];

    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}
```





Inter-Process Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- *Message system* – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
 - `send(message)`
 - `receive(message)`
- The message size is either fixed or variable



Message Passing (Cont.)

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



■ Implementation of communication link

- **Physical**

- ▶ Shared memory
- ▶ Hardware bus
- ▶ Network equipment

- **Logical**

- ▶ Direct or indirect
- ▶ Synchronous or asynchronous
- ▶ Automatic or explicit buffering
- ▶ Network protocols





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



- Messages are directed and received from *mailboxes* (also referred to as *ports*)
 - Each mailbox has a *unique ID*
 - Processes can communicate *only if they share a mailbox*
- Properties of communication link
 - Link established only if processes share a common mailbox
 - A link may be associated with many processes
 - Each pair of processes may share several communication links
 - Link may be unidirectional or bi-directional





Indirect Communication (Cont.)

■ Operations

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

■ Primitives are defined as:

- **send(A, message)** – send a message to mailbox A
- **receive(A, message)** – receive a message from mailbox A



■ Mailbox sharing

- Example

- ▶ P_1 , P_2 , and P_3 share mailbox A,
- ▶ P_1 sends; P_2 and P_3 receive.
- ▶ Who gets the message?

- Solutions

- ▶ Allow a link to be associated with at most two processes
- ▶ Allow only one process at a time to execute a receive operation
- ▶ Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was



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





Producer – Message Passing

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



Consumer – Message Passing

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



Buffering

- Queue of messages attached to the link.
- Implemented in one of three ways
 - *Zero capacity* – no messages are queued on a link
 - ▶ Sender must wait for receiver (rendezvous)
 - *Bounded capacity* – finite length of n messages
 - ▶ Sender must wait if link full
 - *Unbounded capacity* – infinite length
 - ▶ Sender never waits





Examples of IPC Systems - POSIX

■ POSIX Shared Memory

- Process first creates shared memory segment

```
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

- Also used to open an existing segment
- Set the size of the object

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

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



IPC POSIX Producer – Consumer

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

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

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

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

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

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

    return 0;
}
```





Examples of IPC Systems - Mach

■ Mach communication is *message based*

- Even *system calls are messages*
- Each task gets two ports at creation – *Task Self* port and *Notify* port
- Messages are sent and received using the `mach_msg()` function
- Ports needed for communication, created via `mach_port_allocate()`
- Send and receive are flexible, for example four options if mailbox full:
 - ▶ Wait indefinitely
 - ▶ Wait at most n milliseconds
 - ▶ Return immediately
 - ▶ Temporarily cache a message





Mach Messages

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

struct message {
    mach_msg_header_t header;
    int data;
};

mach_port_t client;
mach_port_t server;
```





Mach Message Passing - Client

```
/* Client Code */

struct message message;

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

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





Mach Message Passing - Server

```
/* Server Code */

struct message message;

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



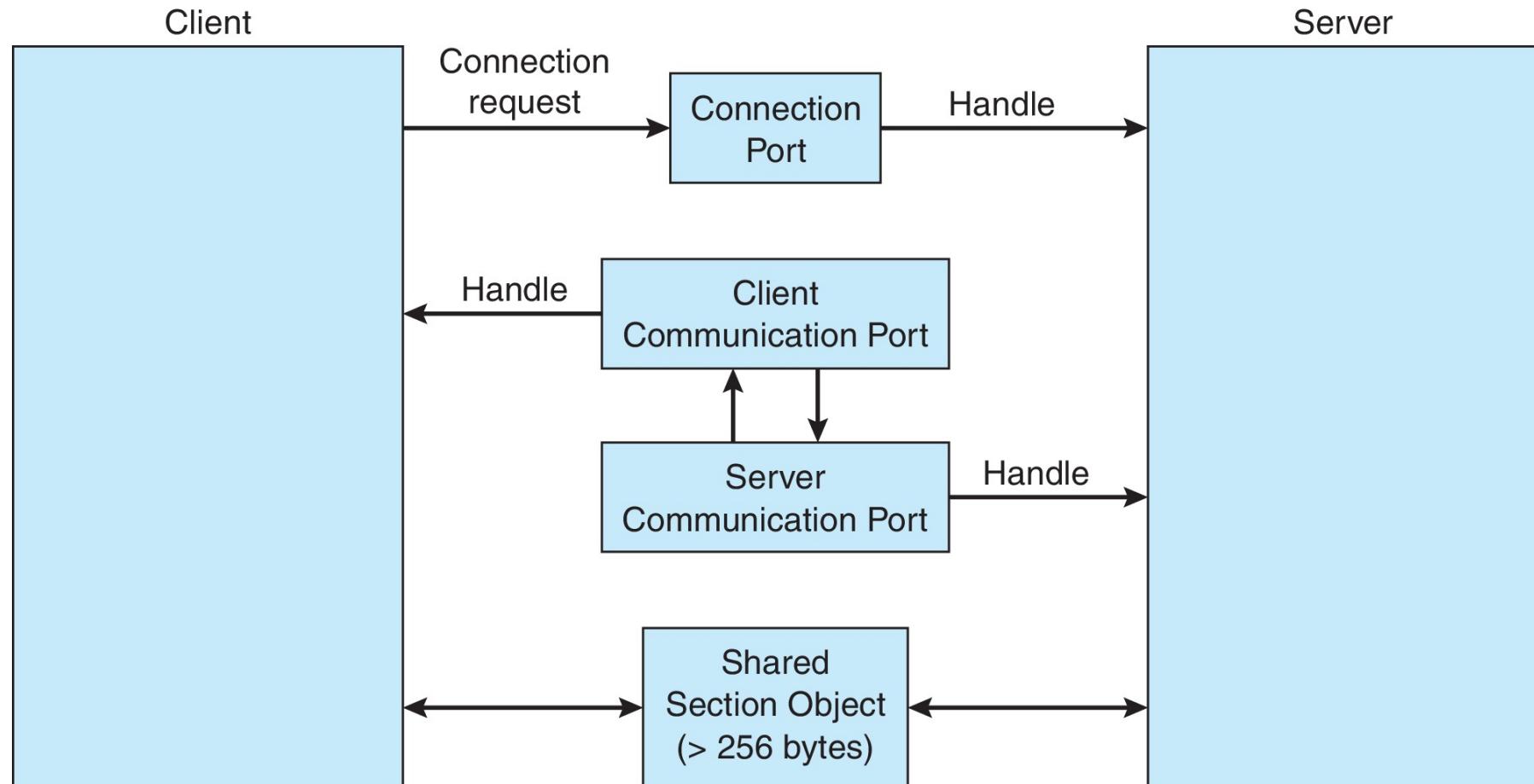


Examples of IPC Systems – Windows

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



Local Procedure Calls in Windows

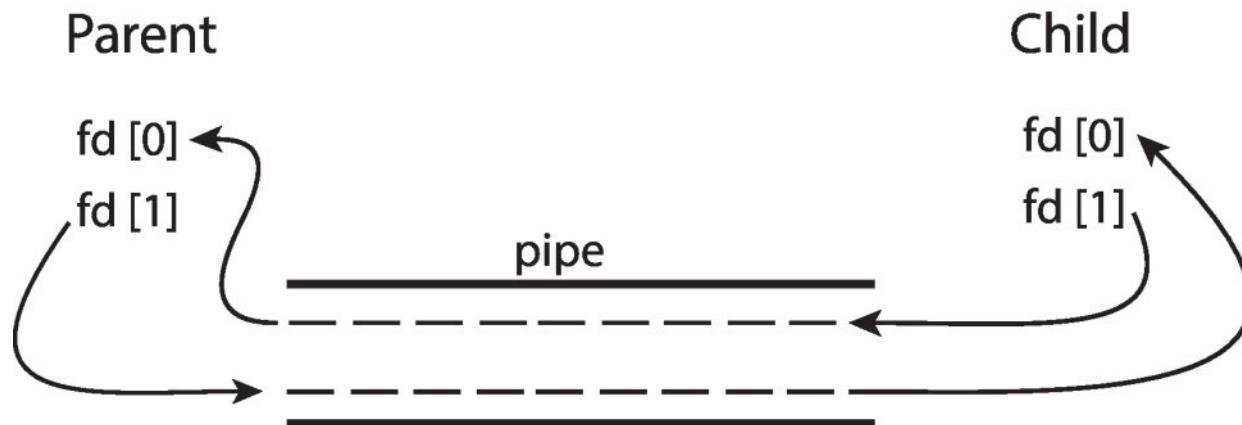


- Acts as a conduit allowing two processes to communicate
- Issues:
 - Is communication unidirectional or bidirectional?
 - In the case of two-way communication, is it half or full-duplex?
 - Must there exist a relationship (e.g., *parent-child*) between the communicating processes?
 - Can the pipes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





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



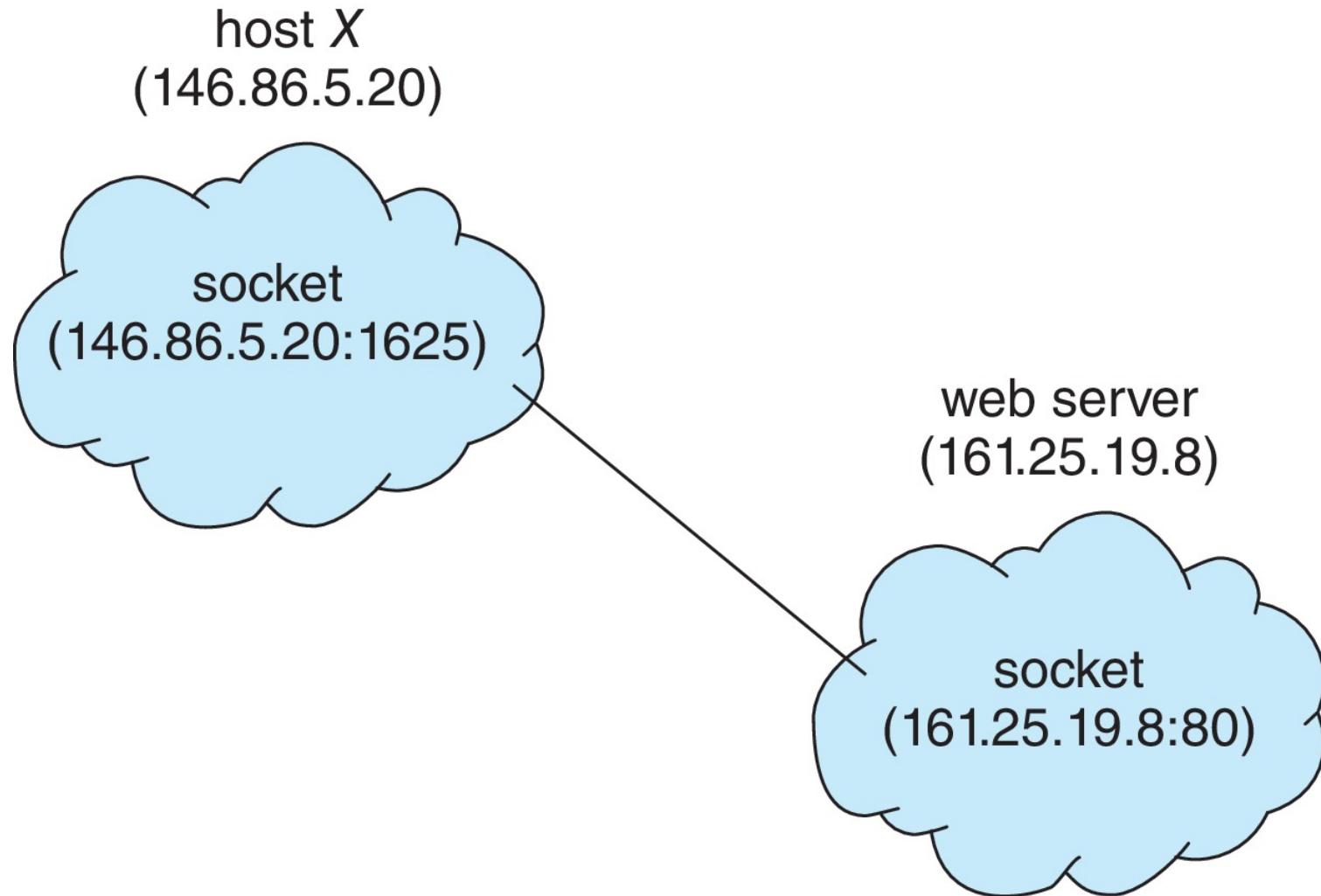
■ Sockets

- A *socket* is defined as an endpoint for communication
- It is a concatenation of *IP address* and *port* – a number included at start of message packet to differentiate network services on a host
 - ▶ E.g., 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

■ Remote Procedure Calls (RPC)



Socket Communication



Sockets in Java – 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);
        }
    }
}
```

■ Three types of sockets

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

■ Consider this “Date” *server* in Java:



Sockets in Java – Client

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

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

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

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

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

■ The equivalent
“Date” *client*





Remote Procedure Calls

- **Remote Procedure Call (RPC)** abstracts procedure calls between processes on *networked systems*
 - Again uses *ports* for service differentiation
- **Stubs** – proxies for the actual procedure on the server and client sides
 - The *client-side stub* locates the server and *marshals* 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)**



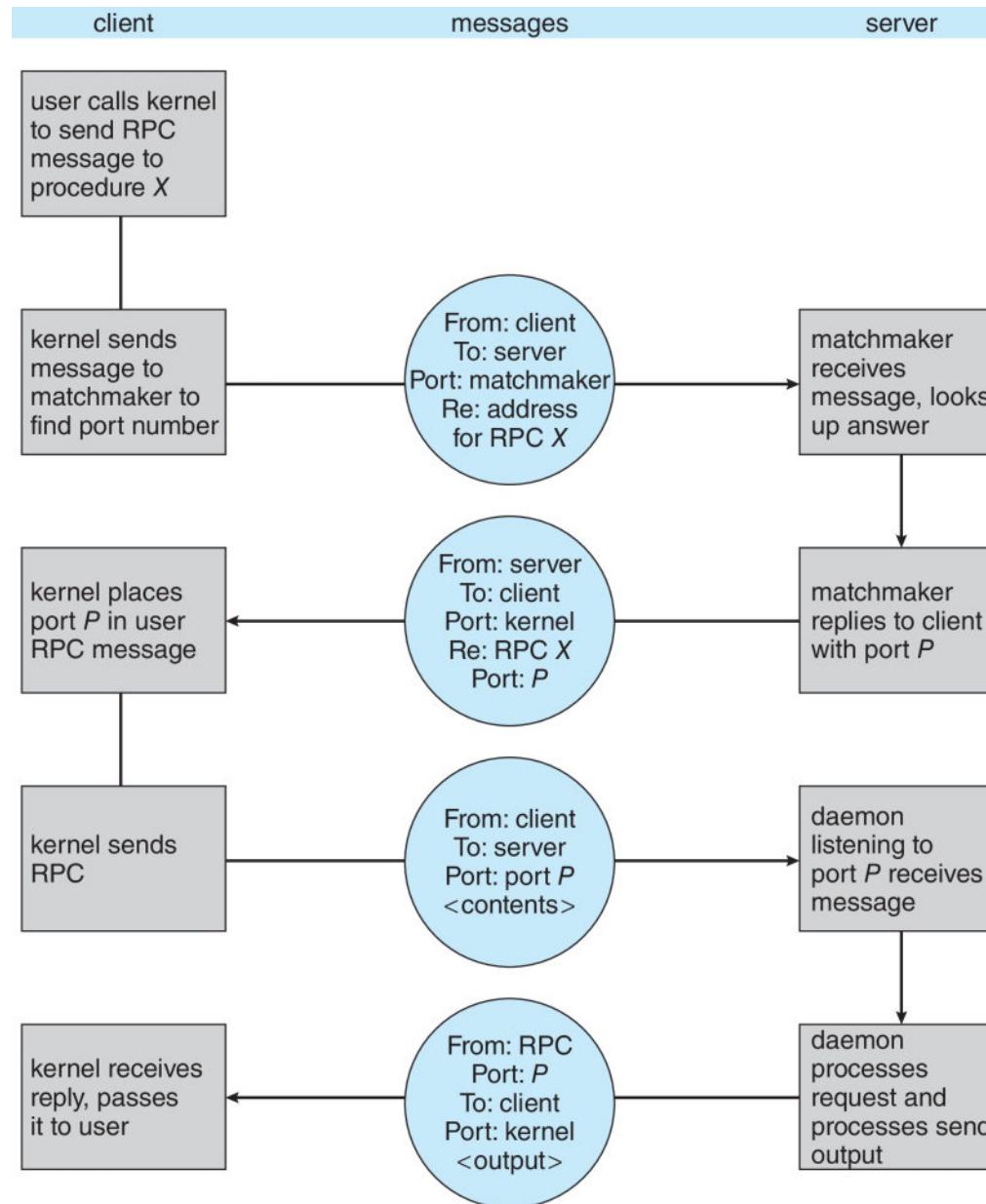


Remote Procedure Calls (Cont.)

- Data representation handled via **External Data Representation (XDR)** format to account for different architectures
 - E.g., *Big-endian* (Motorola) and *little-endian* (Intel x86)
- 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



Summary

- A **process** is a program in execution, and the status of the current activity of a process is represented by the program counter, as well as other registers.
- The **layout of a process in memory** is represented by four different sections: (1) text, (2) data, (3) heap, and (4) stack.
- As a process executes, it changes state. There are **four general states of a process**: (1) ready, (2) running, (3) waiting, and (4) terminated.
- A **process control block (PCB)** is the kernel data structure that represents a process in an operating system.
- The role of the **process scheduler** is to select an available process to run on a CPU.





Summary (Cont.)

- An operating system performs a ***context switch*** when it switches from running one process to running another.
- The ***fork()*** and ***CreateProcess()*** system calls are used to create processes on UNIX and Windows systems, respectively.
- When ***shared memory*** is used for communication between processes, two (or more) processes share the same region of memory. POSIX provides an API for shared memory.
- Two processes may communicate by exchanging messages with one another using ***message passing***. The Mach operating system uses message passing as its primary form of inter-process communication. Windows provides a form of message passing as well.



Summary (Cont.)

- A **pipe** provides a conduit for two processes to communicate. There are two forms of pipes, ordinary and named. Ordinary pipes are designed for communication between processes that have a parent-child relationship. Named pipes are more general and allow several processes to communicate.
- **UNIX** systems provide ordinary pipes through the ***pipe()*** system call. Ordinary pipes have a read end and a write end. A parent process can, for example, send data to the pipe using its write end, and the child process can read it from its read end. Named pipes in UNIX are termed FIFOs.

Summary (Cont.)

- Windows systems also provide ***two forms of pipes***—anonymous and named pipes. Anonymous pipes are similar to UNIX ordinary pipes. They are unidirectional and employ parent-child relationships between the communicating processes. Named pipes offer a richer form of inter-process communication than the UNIX counterpart, FIFOs.
- Two common forms of ***client-server communication*** are sockets and remote procedure calls (RPCs). Sockets allow two processes on different machines to communicate over a network. RPCs abstract the concept of function (procedure) calls in such a way that a function can be invoked on another process that may reside on a separate computer.
- The **Android** operating system uses RPCs as a form of inter-process communication using its binder framework.



End of Chapter 3



What Is an
OPERATING SYSTEM (OS)
and How Does It Work

CLEVERISM.COM

Chapter 4: Threads & Concurrency



What Is an
OPERATING SYSTEM (OS)
and How Does It Work

CLEVERISM.COM



Chapter 4: Outline

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





Objectives

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

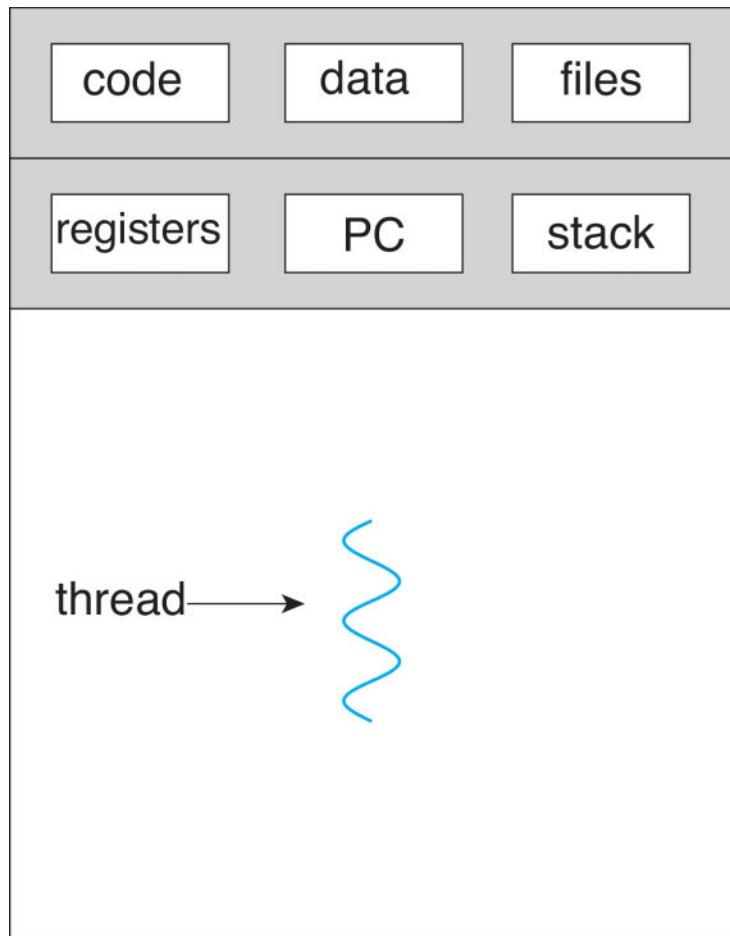


Motivation

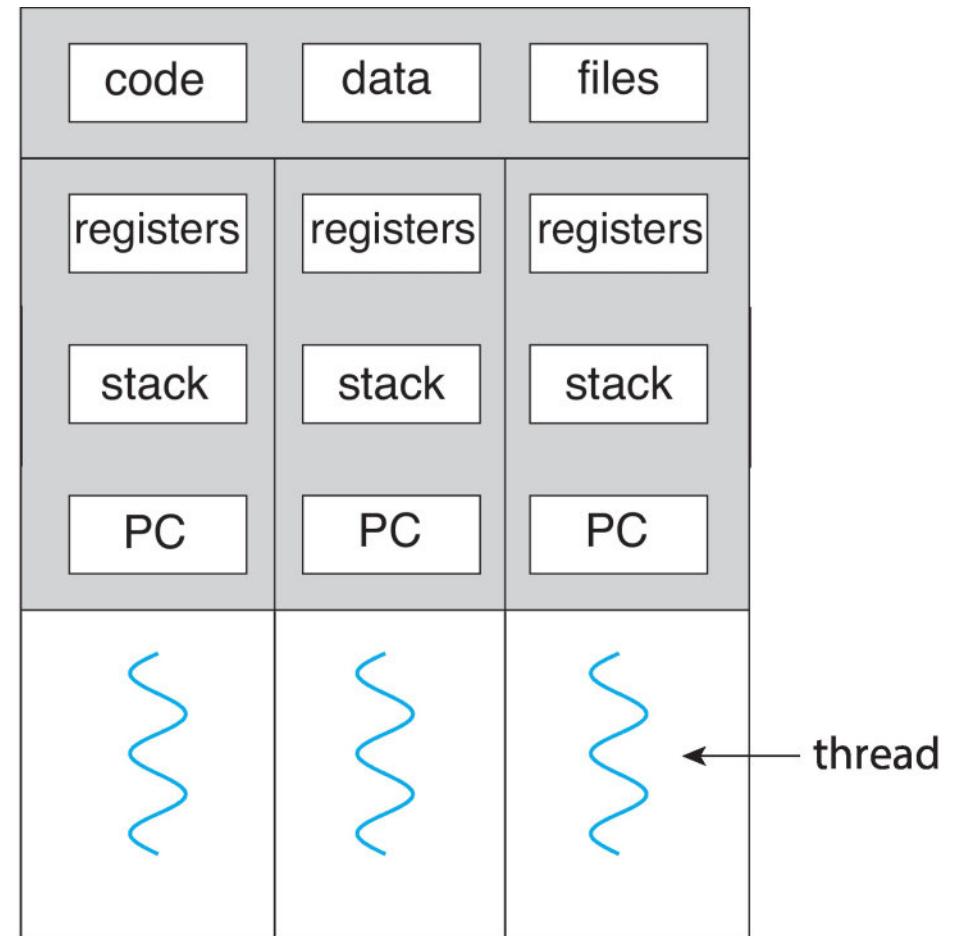
- Most modern applications are *multithreaded*
- Threads run *within* application
- *Multiple tasks* with the application can be implemented by separate threads, for example:
 - Update display
 - Fetch data
 - Spell checking
 - Answer a network request
- *Process creation* is heavy-weight while *thread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



Single and Multithreaded Processes



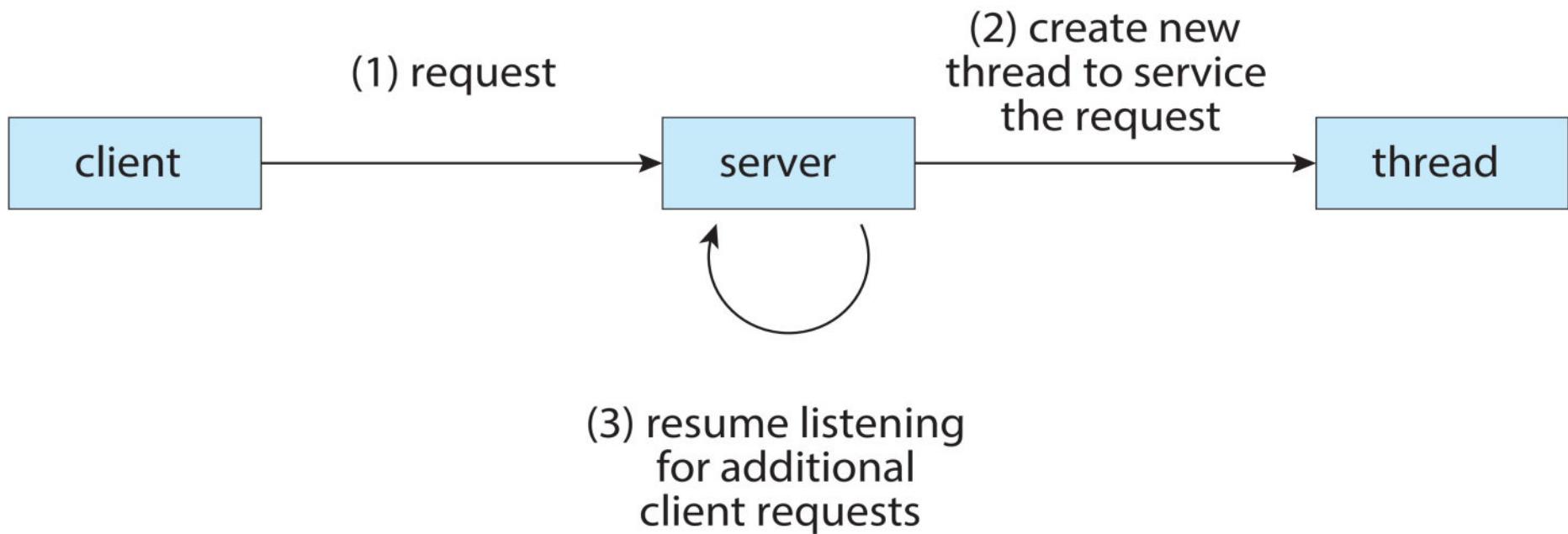
single-threaded process



multithreaded process



Multithreaded Server Architecture





Benefits

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

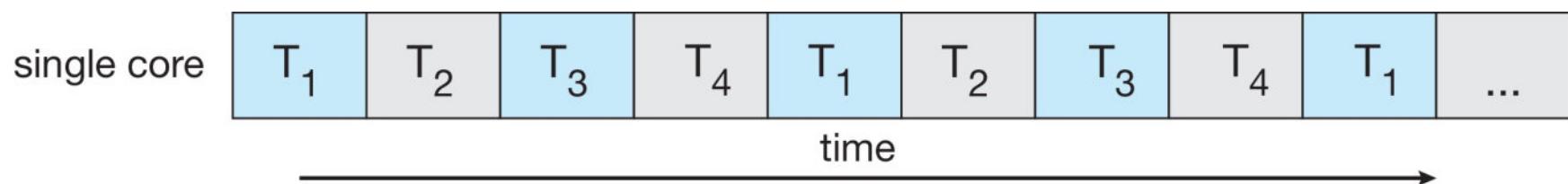


- *Multicore* or *multiprocessor* systems putting pressure on programmers, challenges include:
 - Dividing activities
 - Balance
 - Data splitting
 - Data dependency
 - Testing and debugging
- *Parallelism* implies a system can perform more than one task simultaneously
- *Concurrency* supports more than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

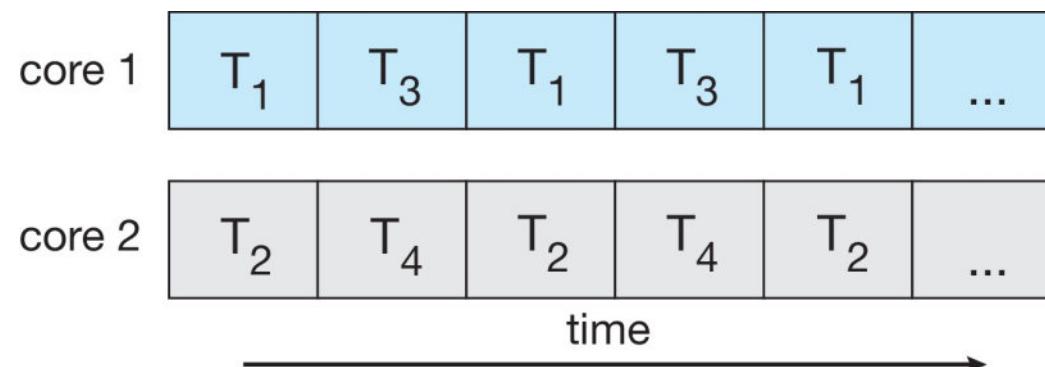


Concurrency vs. Parallelism

■ Concurrent execution on single-core system:

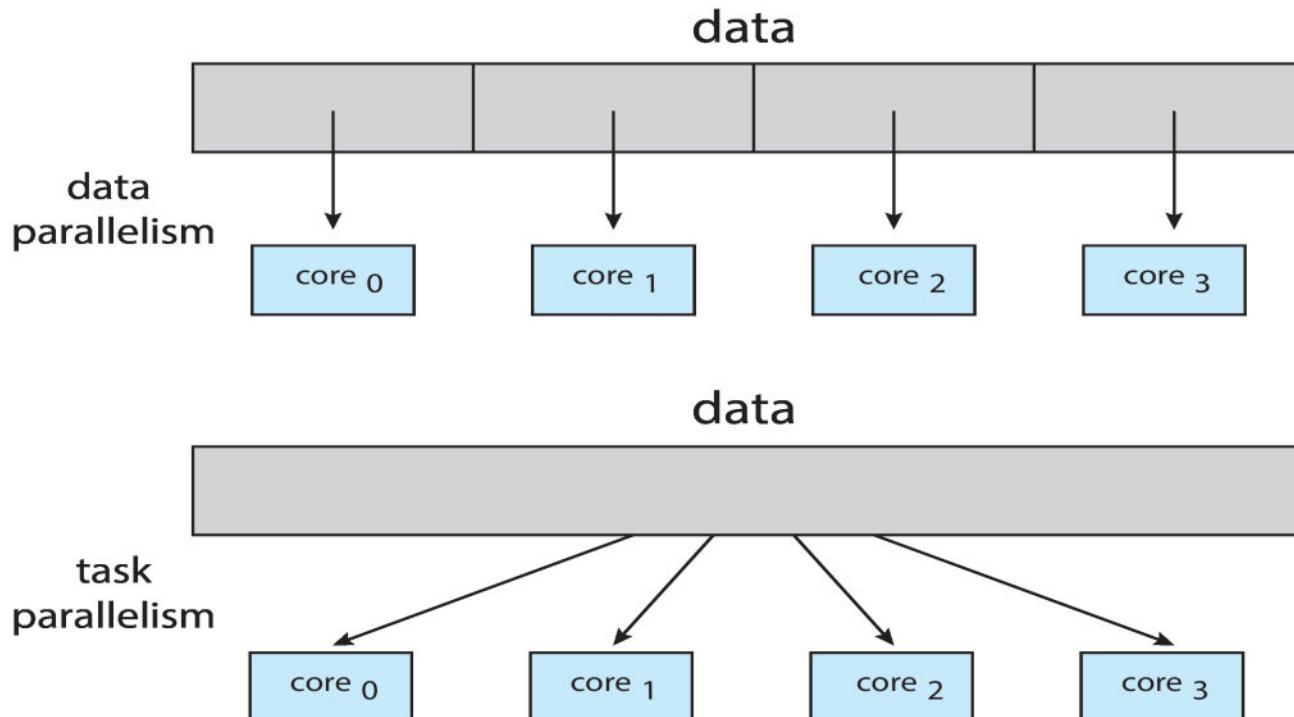


■ Parallelism on a multicore system:



■ Types of parallelism

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



- Identifies *performance gains* from adding additional cores to an application that has both serial and parallel components

- S:** serial portion
- N:** processing cores

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

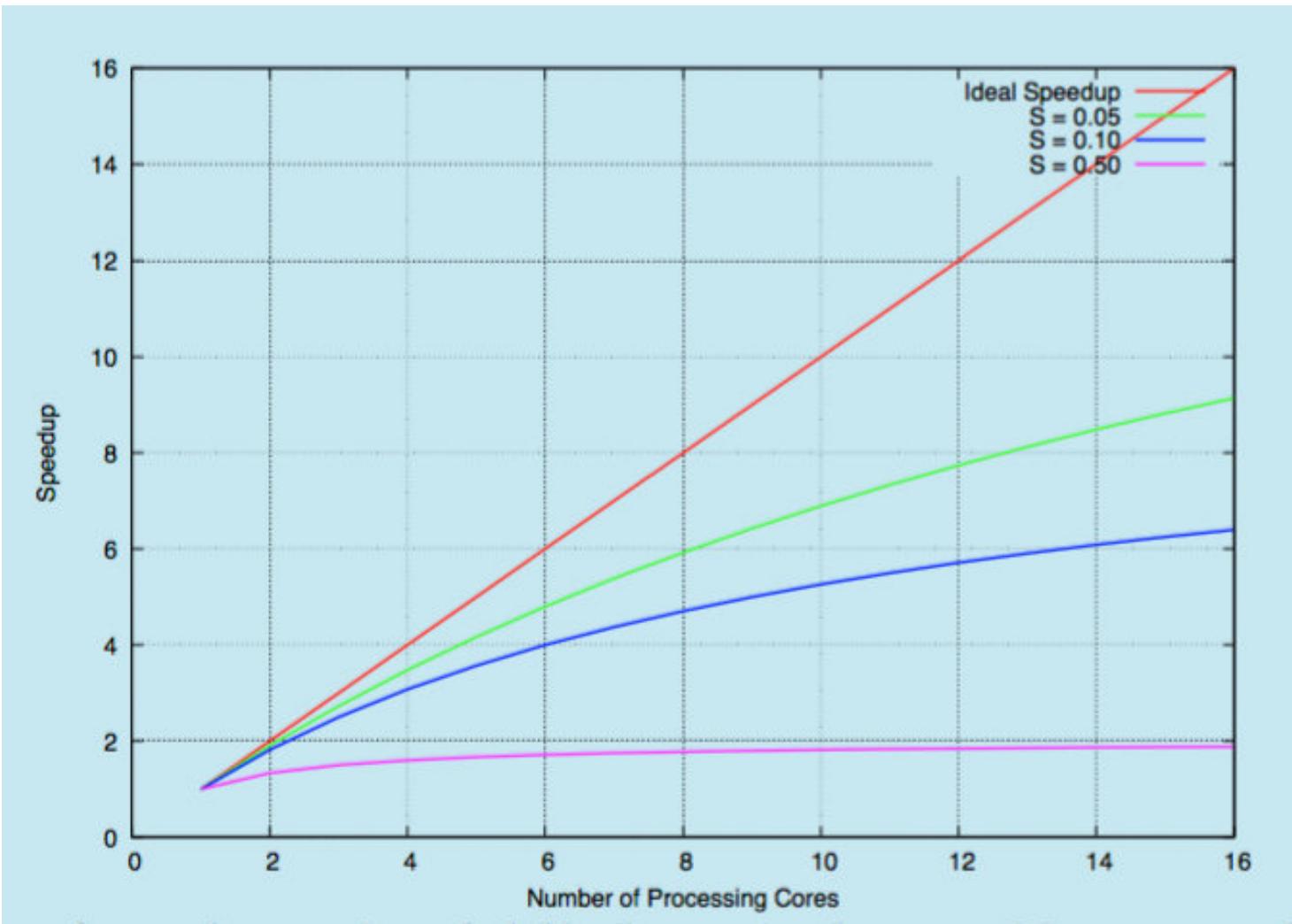
- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As **N** approaches infinity, speedup approaches **1/S**

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

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

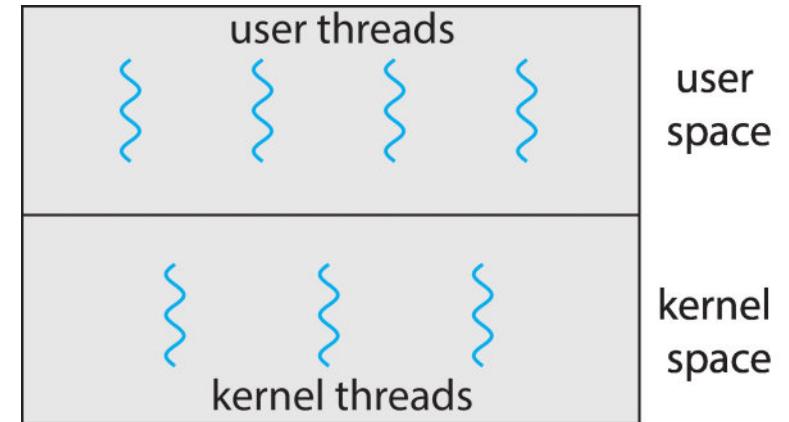


Amdahl's Law



■ *User threads* - management done by user-level threads library

- Three primary thread libraries:
 - ▶ **POSIX Pthreads**
 - ▶ **Windows threads**
 - ▶ **Java threads**



■ *Kernel threads* - supported by the Kernel

- Examples – virtually all general purpose operating systems, including:
 - ▶ Windows
 - ▶ Linux
 - ▶ Mac OS X
 - ▶ iOS, Android



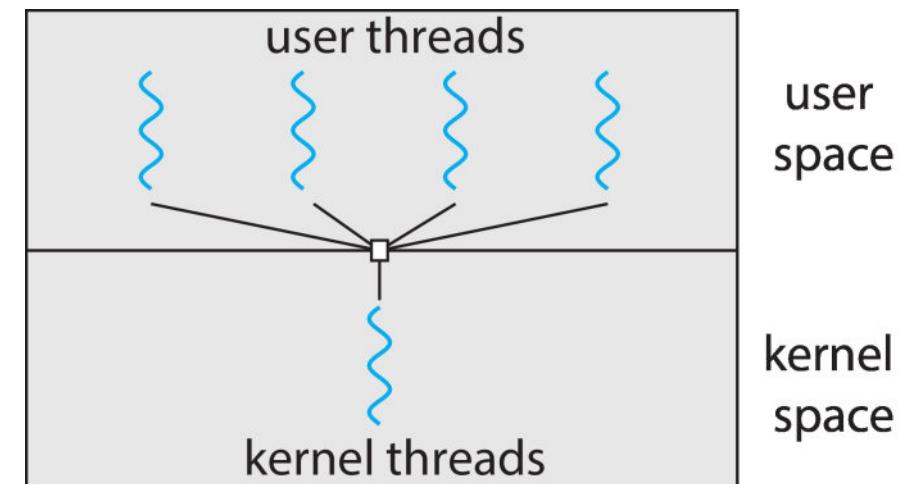


Multithreading Models

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

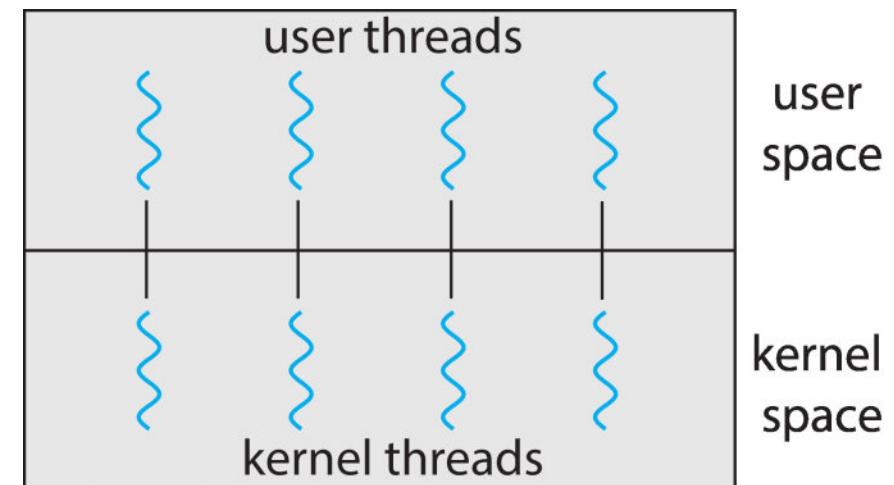
Many-to-One Model

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



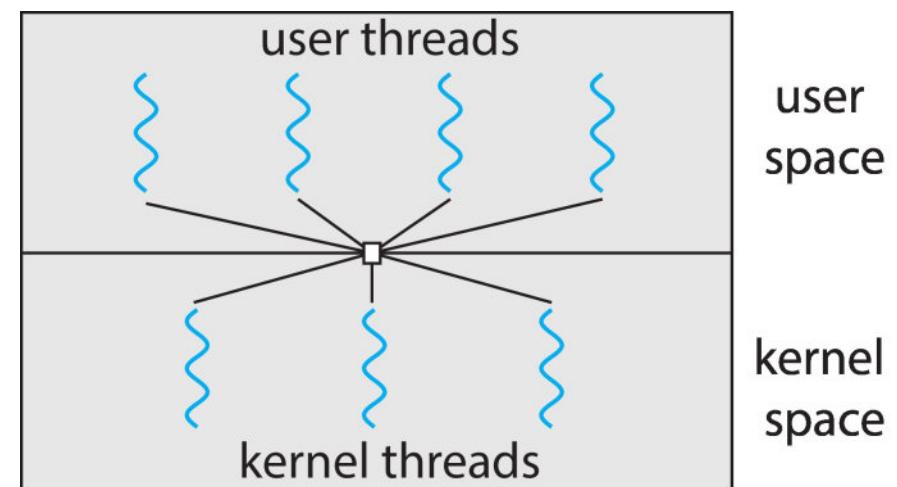
One-to-One Model

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



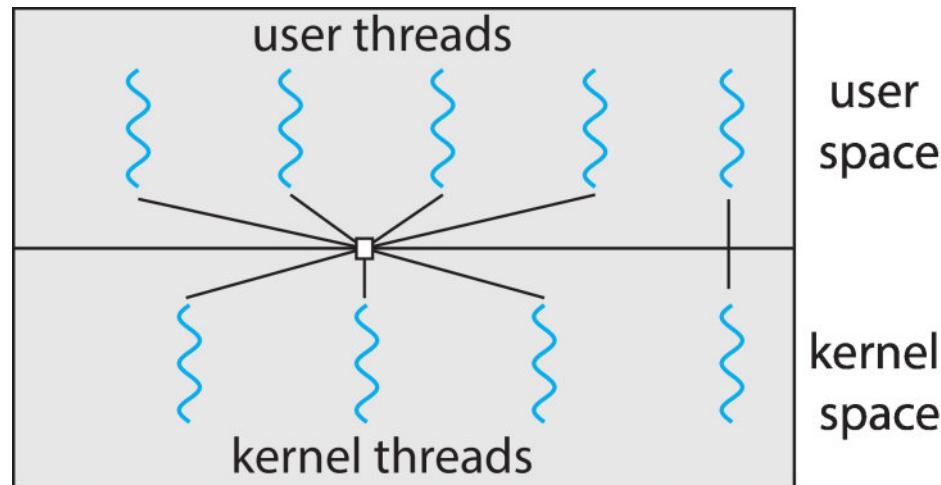
Many-to-Many Model

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



Two-level Model

- Similar to Many-to-Many model, except that it allows a user-level thread to be *bounded* to kernel thread





Thread Libraries

- **Thread library** provides programmers with API for creating and managing threads
- Two primary ways of implementing
 - Library entirely *in user space*
 - *Kernel-level library* supported by the OS





Pthreads

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





PThreads Example

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

#include <stdlib.h>

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

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

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

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





Pthreads Example (Cont.)

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

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

    pthread_exit(0);
}
```





Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

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

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





Windows Multithreaded C Program

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

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





Windows Multithreaded C Program (Cont.)

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

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

    /* now wait for the thread to finish */
    WaitForSingleObject(ThreadHandle, INFINITE);

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

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



- *Java threads* are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
 - Extending **Thread class**
 - Implementing the **Runnable interface**

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

- Standard practice is to implement Runnable interface





Java Threads (Cont.)

■ Implementing Runnable interface:

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

■ Creating a thread:

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

■ Waiting on a thread:

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





Java Executor Framework

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

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

- The **Executor** is used as follows:

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





Java Executor Framework

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





Java Executor Framework (Cont.)

```
public class Driver
{
    public static void main(String[] args) {
        int upper = Integer.parseInt(args[0]);

        ExecutorService pool = Executors.newSingleThreadExecutor();
        Future<Integer> result = pool.submit(new Summation(upper));

        try {
            System.out.println("sum = " + result.get());
        } catch (InterruptedException | ExecutionException ie) { }
    }
}
```





Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness is more difficult with explicit threads
- Creation and management of threads done by *compilers* and *run-time libraries* rather than programmers
- Five methods explored
 - *Thread Pools*
 - *Fork-Join*
 - *OpenMP*
 - *Grand Central Dispatch*
 - *Intel Threading Building Blocks*





Thread Pools

■ *Create a number of threads in a pool where they await work*

■ **Advantages:**

- Usually slightly faster to service a request with an existing thread than create a new thread
- Allows the number of threads in the application(s) to be bounded to the size of the pool
- Separating task to be performed from mechanics of creating task allows different strategies for running task
 - ▶ i.e., Tasks could be scheduled to run periodically

■ **Windows API** supports thread pools:

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





Java Thread Pools

■ Three factory methods for creating thread pools in **Executors** class:

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

■ Example

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

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

        /* Create the thread pool */
        ExecutorService pool = Executors.newCachedThreadPool();

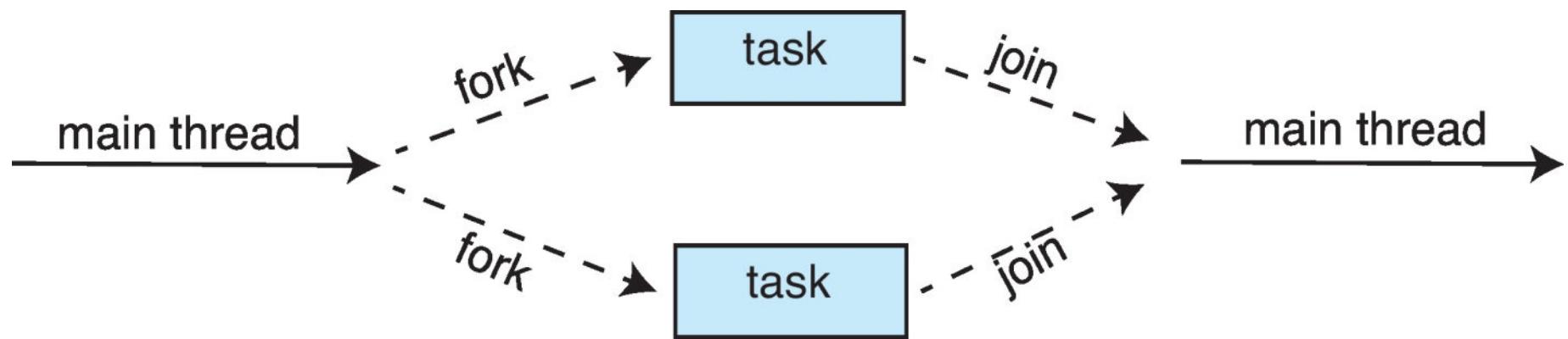
        /* Run each task using a thread in the pool */
        for (int i = 0; i < numTasks; i++)
            pool.execute(new Task());

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



Fork-Join Parallelism

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





Fork-Join Parallelism (Cont.)

- General algorithm for *fork-join strategy*:

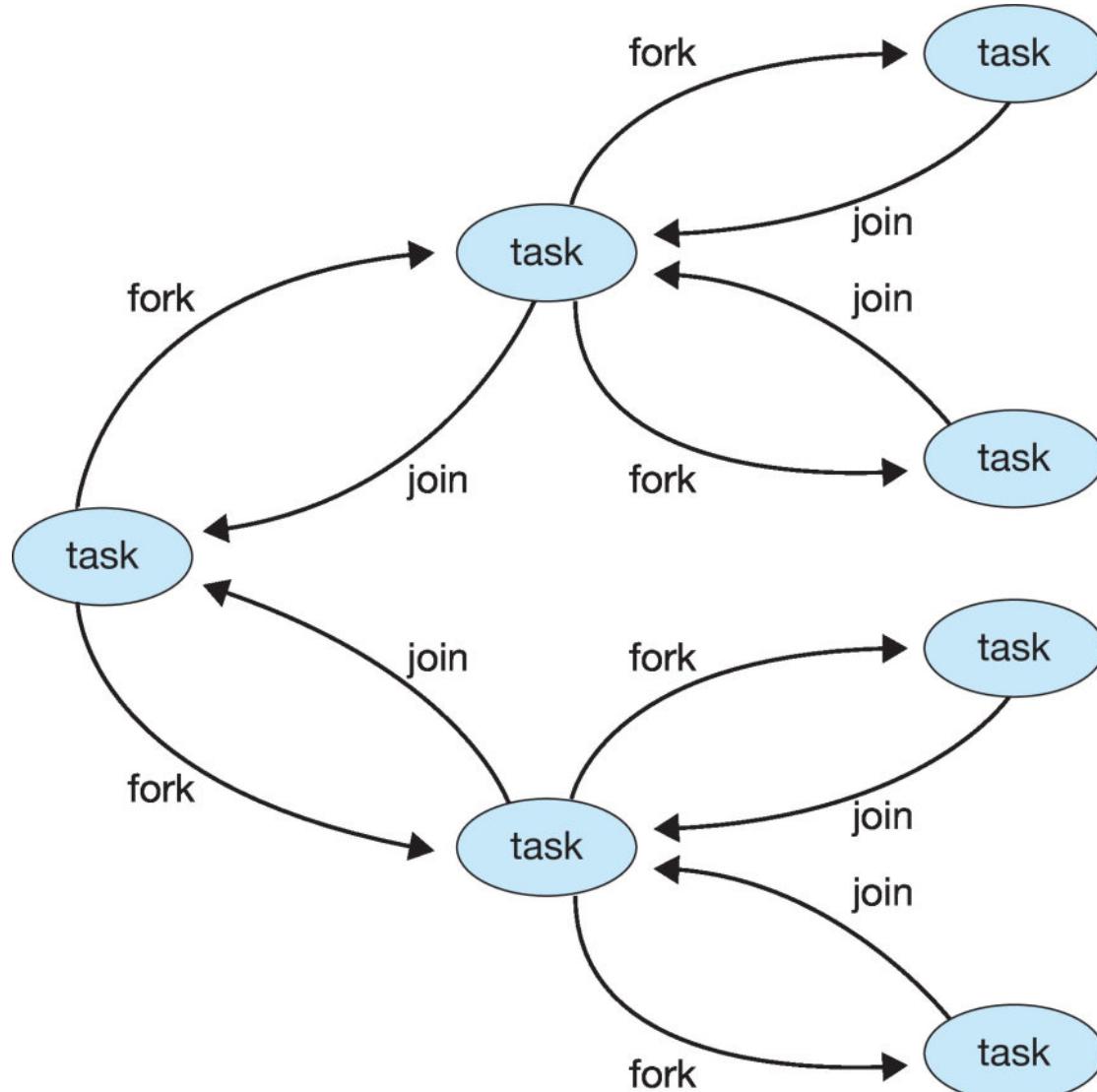
```
Task(problem)
    if problem is small enough
        solve the problem directly
    else
        subtask1 = fork(new Task(subset of problem)
        subtask2 = fork(new Task(subset of problem)

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

    return combined results
```



Fork-Join Parallelism (Cont.)





Fork-Join Parallelism in Java

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

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





Fork-Join Parallelism in Java (Cont.)

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

public class SumTask extends RecursiveTask<Integer>
{
    static final int THRESHOLD = 1000;

    private int begin;
    private int end;
    private int[] array;

    public SumTask(int begin, int end, int[] array) {
        this.begin = begin;
        this.end = end;
        this.array = array;
    }

    protected Integer compute() {
        if (end - begin < THRESHOLD) {
            int sum = 0;
            for (int i = begin; i <= end; i++)
                sum += array[i];

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

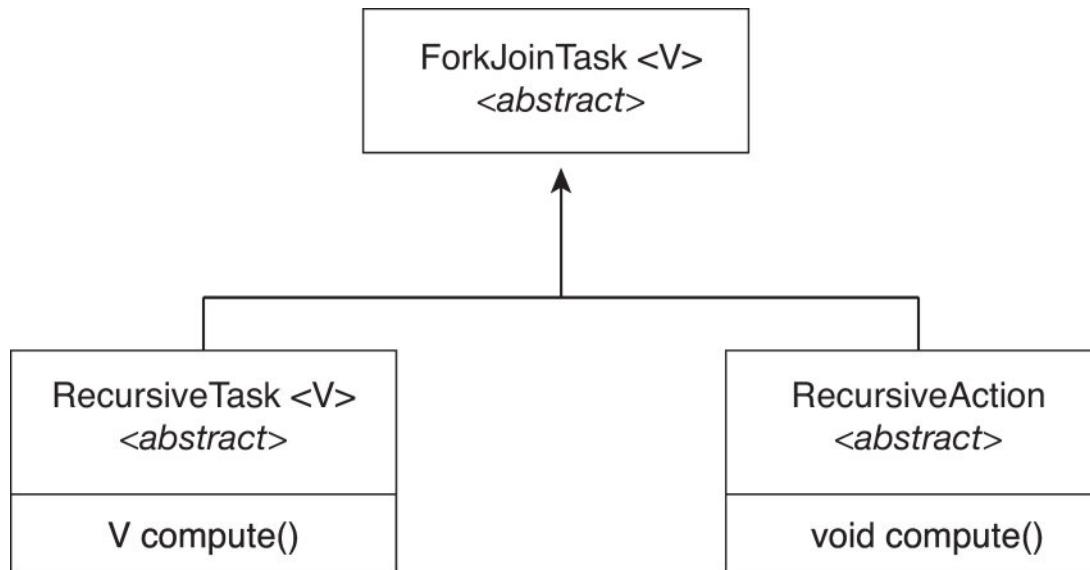
            SumTask leftTask = new SumTask(begin, mid, array);
            SumTask rightTask = new SumTask(mid + 1, end, array);

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

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



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





OpenMP

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

```
#pragma omp parallel
```

- *Create as many threads as there are cores*
- Run the **for** loop in parallel

```
#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;
}

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





Grand Central Dispatch

- *Apple technology* for **macOS** and **iOS** operating systems
- Extensions to C, C++ and Objective-C languages, API, and run-time library
- Allows identification of *parallel sections*
- Manages most of the details of threading
- Block is in “^{ }” :

```
^{ printf("I am a block") ; }
```

- Blocks placed in dispatch queue
 - Assigned to available thread in thread pool when removed from queue





Grand Central Dispatch (Cont.)

■ Two types of dispatch queues:

- **serial** – blocks removed in FIFO order, queue is per process, called *main queue*
 - ▶ Programmers can create additional serial queues within program
- **concurrent** – removed in FIFO order but several may be removed at a time
 - ▶ Four system-wide queues divided by *quality of service*:
 - `QOS_CLASS_USER_INTERACTIVE`
 - `QOS_CLASS_USER_INITIATED`
 - `QOS_CLASS_USER.Utility`
 - `QOS_CLASS_USER_BACKGROUND`





Grand Central Dispatch (Cont.)

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

```
let queue = dispatch_get_global_queue  
    (QOS_CLASS_USER_INITIATED, 0)
```

```
dispatch_async(queue, { print("I am a closure.") })
```



Intel Threading Building Blocks (TBB)

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

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

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

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





Threading Issues

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





Semantics of fork() and exec()

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



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





Signal Handling (Cont.)

- Where should a signal be delivered for *multithreaded*?
 - 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



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

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



Thread Cancellation (Cont.)

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

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

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





Thread Cancellation in Java

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

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

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

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





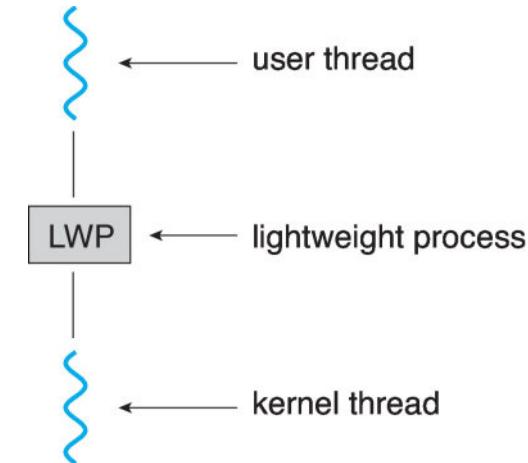
Thread-Local Storage

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



Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **LightWeight Process (LWP)**
 - Appears to be a virtual processor on which process can schedule user thread to run
 - Each LWP attached to kernel thread
 - How many LWPs to create?
- Scheduler activations provide *upcalls* - a communication mechanism from the kernel to the *upcall handler* in the thread library
- This communication allows an application to maintain the correct number kernel threads





Operating System Examples

- Windows Threads
- Linux Threads

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





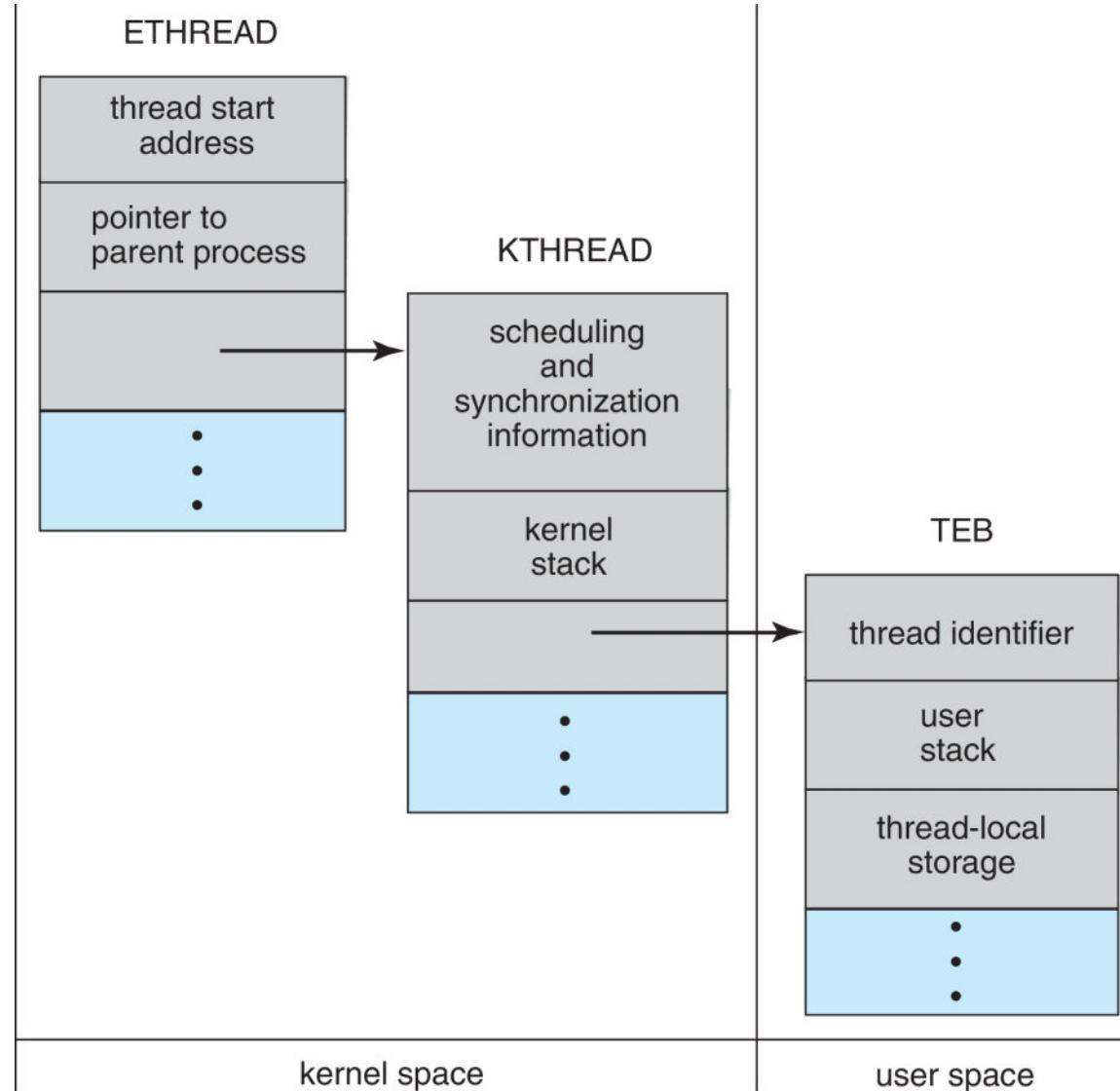
Windows Threads (Cont.)

■ The *primary data structures* of a thread include:

- **ETHREAD** (*executive thread block*) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
- **KTHREAD** (*kernel thread block*) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
- **TEB** (*thread environment block*) – thread ID, user-mode stack, thread-local storage, in user space



Windows Threads Data Structures



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

- Flags control behavior

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

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



Summary

- A ***thread*** represents a basic unit of CPU utilization, and threads belonging to the same process share many of the process resources, including code and data.
- There are ***four primary benefits*** to multithreaded applications:
(1) responsiveness, (2) resource sharing, (3) economy, and (4) scalability.
- ***Concurrency*** exists when multiple threads are making progress, whereas ***parallelism*** exists when multiple threads are making progress simultaneously. On a system with a single CPU, only concurrency is possible; parallelism requires a multicore system that provides multiple CPUs.



Summary (Cont.)

- There are several **challenges** in designing multithreaded applications. They include dividing and balancing the work, dividing the data between the different threads, and identifying any data dependencies. Finally, multithreaded programs are especially challenging to test and debug.
- **Data parallelism** distributes subsets of the same data across different computing cores and performs the same operation on each core. **Task parallelism** distributes not data but tasks across multiple cores. Each task is running a unique operation.
- User applications create **user-level threads**, which must ultimately be mapped to **kernel threads** to execute on a CPU. The **many-to-one model** maps many user-level threads to one kernel thread. Other approaches include the **one-to-one** and **many-to-many** models.



Summary (Cont.)

- A ***thread library*** provides an API for creating and managing threads. Three common thread libraries include Windows, Pthreads, and Java threading. Windows is for the Windows system only, while Pthreads is available for POSIX-compatible systems such as UNIX, Linux, and macOS. Java threads will run on any system that supports a Java virtual machine.
- ***Implicit threading*** involves identifying tasks—not threads—and allowing languages or API frameworks to create and manage threads. There are several approaches to implicit threading, including ***thread pools***, ***fork-join frameworks***, and ***Grand Central Dispatch***. Implicit threading is becoming an increasingly common technique for programmers to use in developing concurrent and parallel applications.





Summary (Cont.)

- Threads may be terminated using either ***asynchronous or deferred cancellation***. Asynchronous cancellation stops a thread immediately, even if it is in the middle of performing an update. Deferred cancellation informs a thread that it should terminate but allows the thread to terminate in an orderly fashion. In most circumstances, deferred cancellation is preferred to asynchronous termination.
- Unlike many other operating systems, ***Linux does not distinguish between processes and threads***; instead, it refers to each as a ***task***. The Linux `clone()` system call can be used to create tasks that behave either more like processes or more like threads.



End of Chapter 4



What Is an
OPERATING SYSTEM (OS)
and How Does It Work

CLEVERISM.COM

Chapter 5: CPU Scheduling





Chapter 5: Outline

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





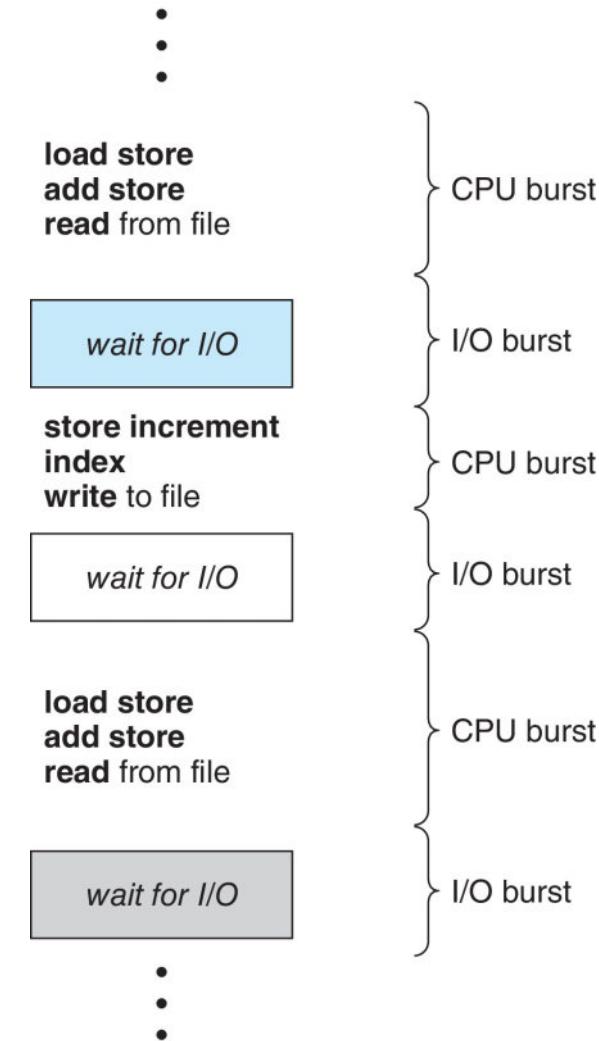
Objectives

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



Basic Concepts

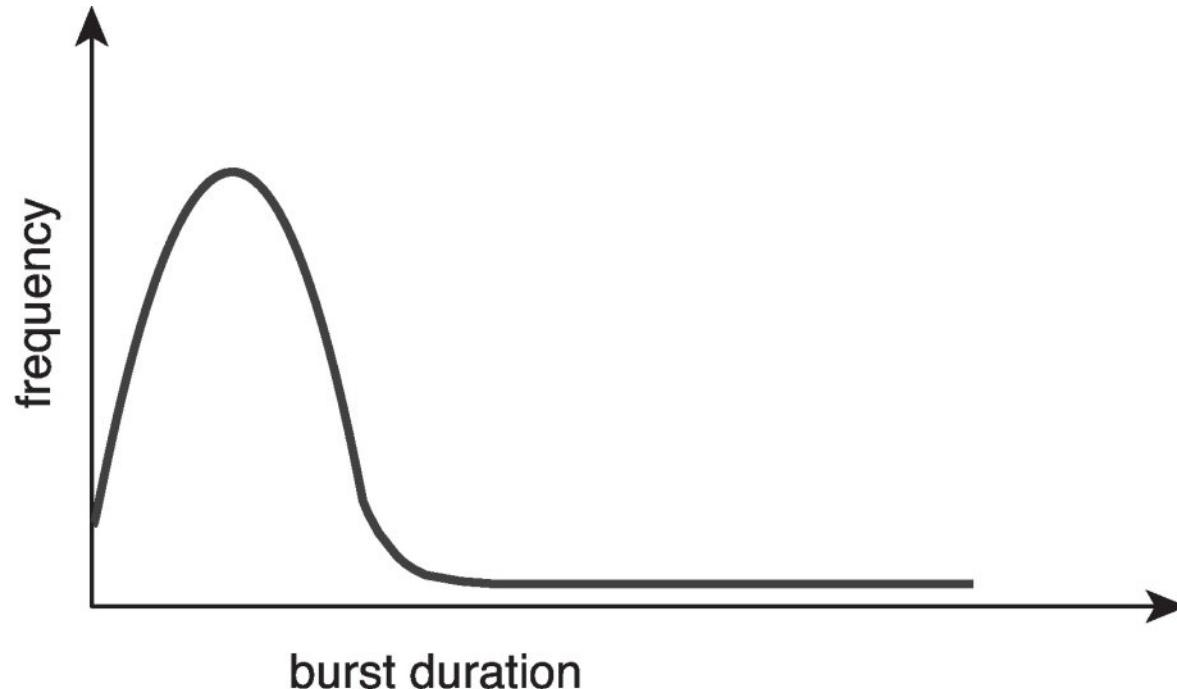
- Almost all computer resources are *scheduled* before use
- Maximum *CPU utilization* obtained with multiprogramming
- *CPU–I/O Burst Cycle* – Process execution consists of a cycle of CPU execution and I/O wait
 - *CPU burst* followed by *I/O burst*
 - *CPU burst* distribution is of main concern



Histogram of CPU-burst Times

■ Generally, frequency curve shows

- Large number of *short bursts*
- Small number of *longer bursts*

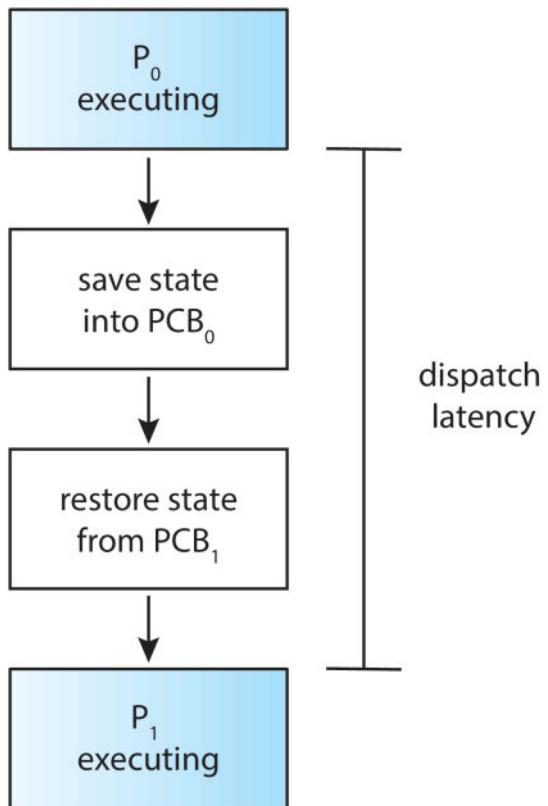


CPU Scheduler

- The *CPU scheduler* selects one process from among the processes in *ready queue*, and allocates the CPU core to it
 - ▶ Queue may be ordered in various ways: FIFO, priority, tree, linked list
- *CPU scheduling decisions* may take place when a process:
 1. switches from *running* to *waiting* state
 2. switches from *running* to *ready* state
 3. switches from *waiting* to *ready*
 4. terminates
- Scheduling under 1 and 4 is *nonpreemptive*
 - ▶ No choice in terms of scheduling
- All other scheduling is *preemptive*, and can result in *race conditions*
 - Consider access to shared data
 - Consider preemption while in kernel mode
 - Consider interrupts occurring during crucial OS activities



Dispatcher



- **Dispatcher module** gives control of the CPU to the process selected by the short-term scheduler; this involves:
 - switching context
 - ▶ The number of context switches can be obtained by using the `#vmstat` command or the `/proc` file system for a given process
 - switching to user mode
 - jumping to the proper location in the user program to resume that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

`#vmstat`





Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – number of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process spends waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not outputting the response (for time-sharing environment or in an interactive system)

#top





Scheduling Algorithm Optimization Criteria

- Max *CPU utilization*
 - Max *Throughput*
 - Min *Turnaround time*
 - Min *Waiting time*
 - Min *Response time*
- In most cases, it is necessary to *optimize the average measure*
- For interactive systems (such as a PC desktop or laptop system), it is more important to *minimize the variance* in the response time

Note: For next examples of the comparison of various CPU-scheduling algorithms

- ▶ Consider only *one CPU burst* (in milliseconds) per process
- ▶ The measure of comparison: **average waiting time**



- **Motivation:** for simplicity, consider FIFO-like policy

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

- Suppose that the processes arrive at time 0 in the order: P_1, P_2, P_3
- The *Gantt Chart* for the schedule is:



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



FCFS Scheduling (Cont.)

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



- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- **Average waiting time** = $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- **Convoys effect** – short processes behind a long process, all the other processes wait for the one big process to get off the CPU
 - Consider one CPU-bound and many I/O-bound processes
 - Result in *lower* CPU and device utilization



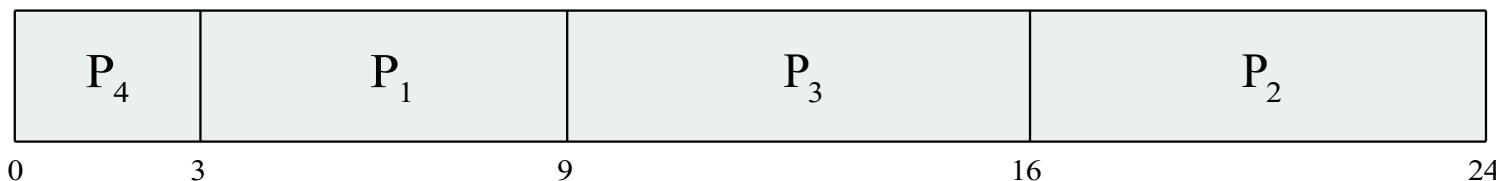
- **Motivation:** Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process
 - The *shortest-next-CPU-burst* algorithm
- Associate with each process *the length of its next CPU burst*
 - When the CPU is available, it is assigned to the process that has the *smallest next CPU burst*
 - *FCFS scheduling* is used if the next CPU bursts of two processes are the same
- SJF is provably *optimal* – gives minimum average waiting time for a given set of processes
 - The difficulty is how to know the length of the next CPU request
 - Could ask the user



Example of SJF scheduling

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

■ SJF scheduling *Gantt chart*



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



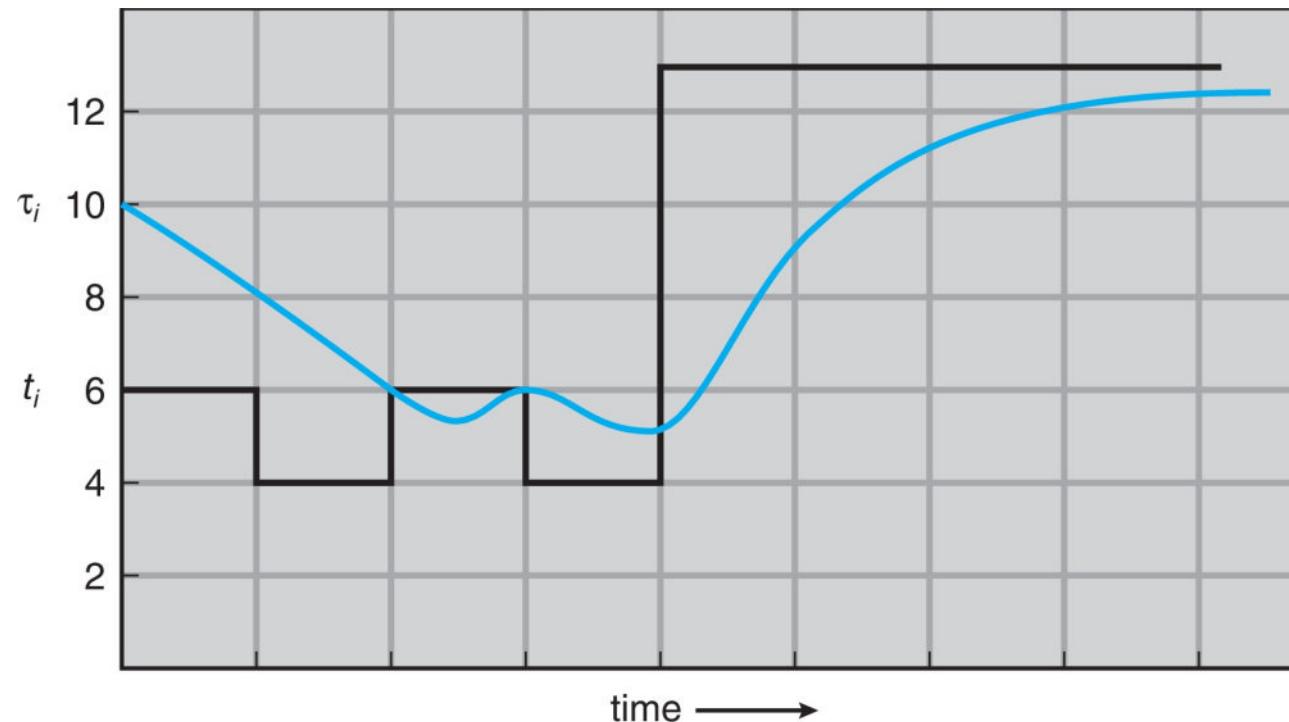
Determining Length of Next CPU Burst

- Can only *estimate* the length – should be similar to the previous one
 - Then pick process with shortest predicted next CPU burst
- Can be done by using *exponential averaging* of the measured lengths of previous CPU bursts as follows
 1. t_n = actual length of n^{th} CPU burst
 2. τ_{n+1} = predicted value for the next CPU burst
 3. $\alpha, 0 \leq \alpha \leq 1$
 4. Define : $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$.
- Commonly, α controls the relative weight of recent and past history in the prediction and *sets to ½*
- *Preemptive* version called *Shortest-Remaining-Time-First* (SRTF)



Prediction of the Length of the Next CPU Burst

- An exponential average with $\alpha = 1/2$ and $\tau_0 = 10$



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



Examples of Exponential Averaging

■ $\alpha = 0$

- $\tau_{n+1} = \tau_n$
- Recent history does not count

■ $\alpha = 1$

- $\tau_{n+1} = \alpha t_n$
- Only the actual last CPU burst counts

■ If we expand the formula, we get:

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

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

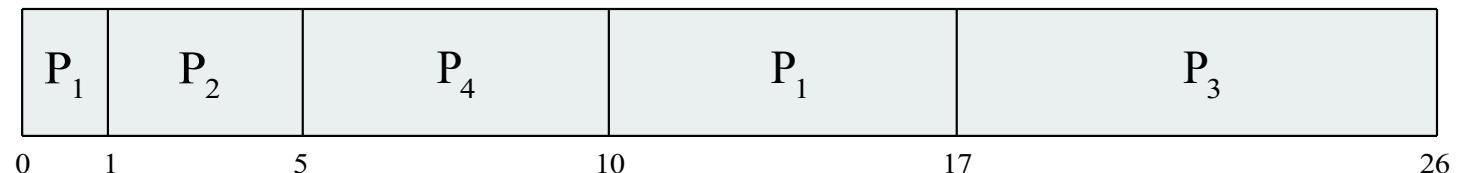


Shortest-Remaining-Time-First (SRTF)

- **Motivation:** now, we add the concepts of *varying arrival times* and *preemption* to the analysis

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

- Preemptive SJF *Gantt Chart*



- **Average waiting time** = $[(10-1)+(1-1)+(17-2)+(5-3)]/4 = 6.5 \text{ ms}$

- Which is the value for *nonpreemptive SJF scheduling*?



Round Robin (RR) Scheduling

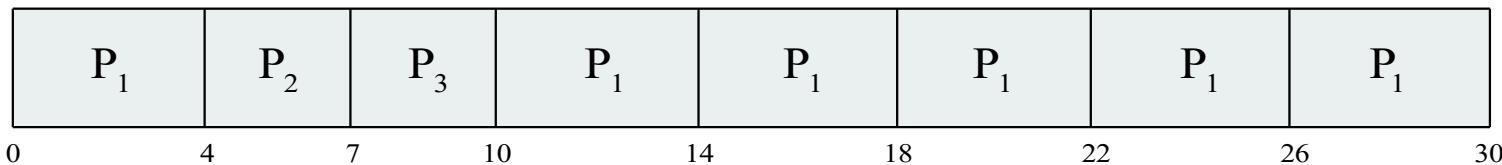
- **Motivation:** try scheduling algorithm similar to *FCFS scheduling*, but *preemption* is added to enable the system to switch between processes
- Each process gets a small unit of CPU time (*time quantum q*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue
- If there are n processes in the ready queue and the time quantum is q , then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units
- *Timer interrupts every quantum to schedule next process*
- Performance
 - q large \Rightarrow FIFO
 - q small \Rightarrow q must be large with respect to context switch, otherwise overhead is too high



Example of RR with Time Quantum = 4

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

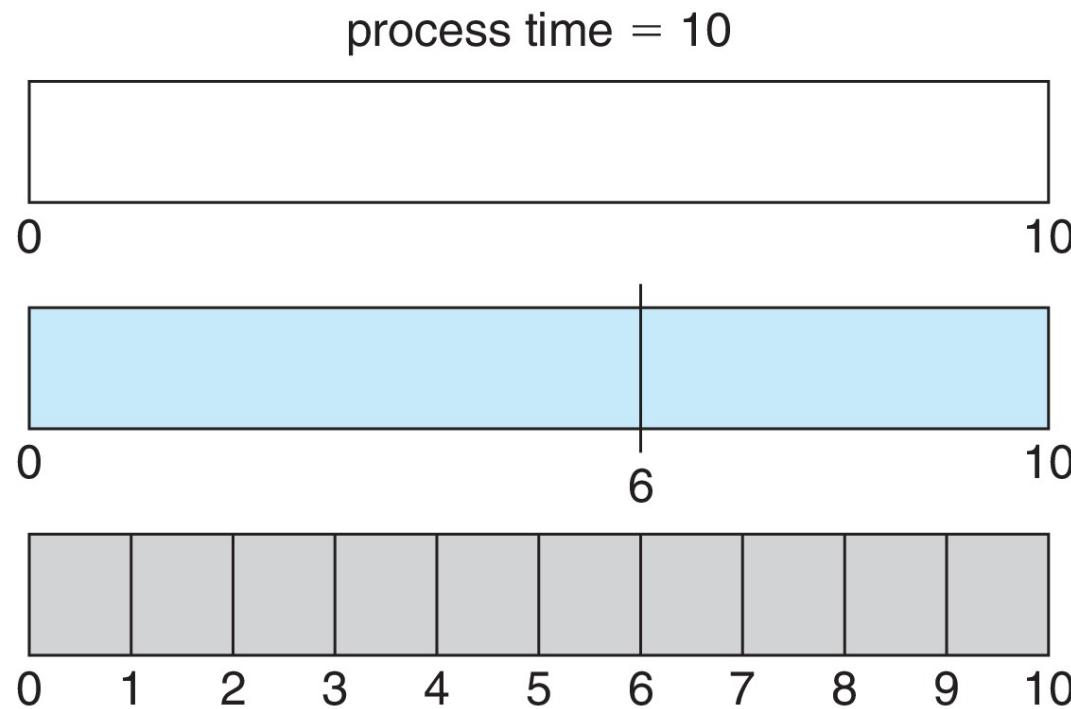
- The *Gantt chart* is:



- **Average waiting time = ?**
- Typically, *higher average turnaround* than SJF, but *better response*
- q should be large compared to context switch time
- q usually *10ms to 100ms*, context switch $< 10\mu\text{sec}$



Time Quantum and Context Switch Time



quantum

12

context switches

0

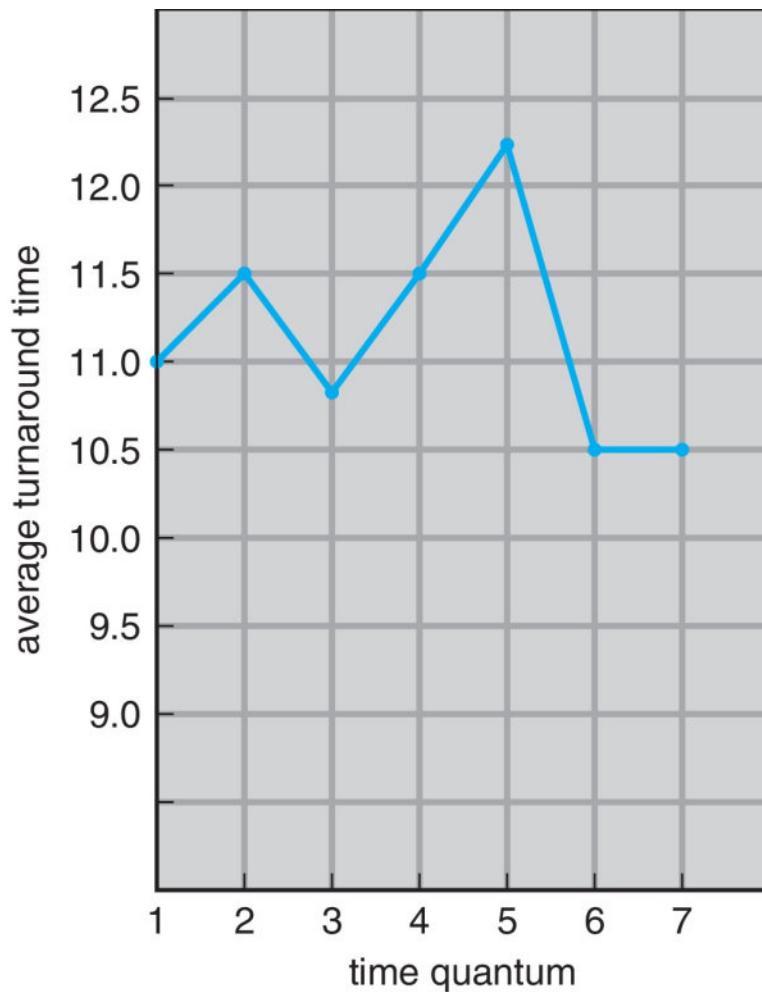
6

1

1

9

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

- 80% of CPU bursts should be shorter than q

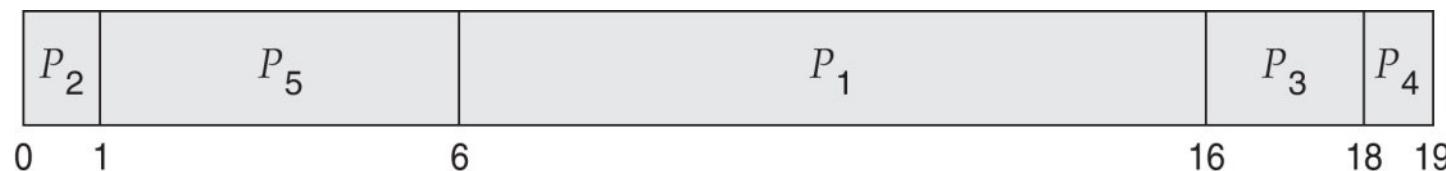
- **Motivation:** A *priority number* (*integer*) is associated with each process
- The CPU is allocated to the process with the *highest priority* (*smallest integer = highest priority*). Equal-priority processes are scheduled in FCFS or RR
 - Preemptive
 - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem \equiv *Starvation* – low priority processes may never execute
 - Solution \equiv *Aging* – as time progresses, increase the priority of the process



Example of Priority Scheduling

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

■ Priority scheduling *Gantt Chart*



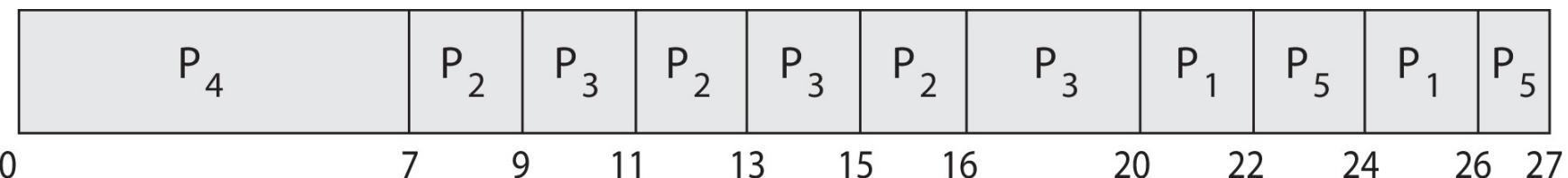
- **Average waiting time = 8.2 ms**



Priority Scheduling w/ Round-Robin

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

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

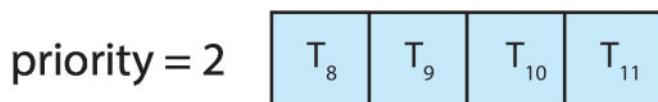


● Average waiting time = ?



Multilevel Queue

- **Motivation:** with priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!



●

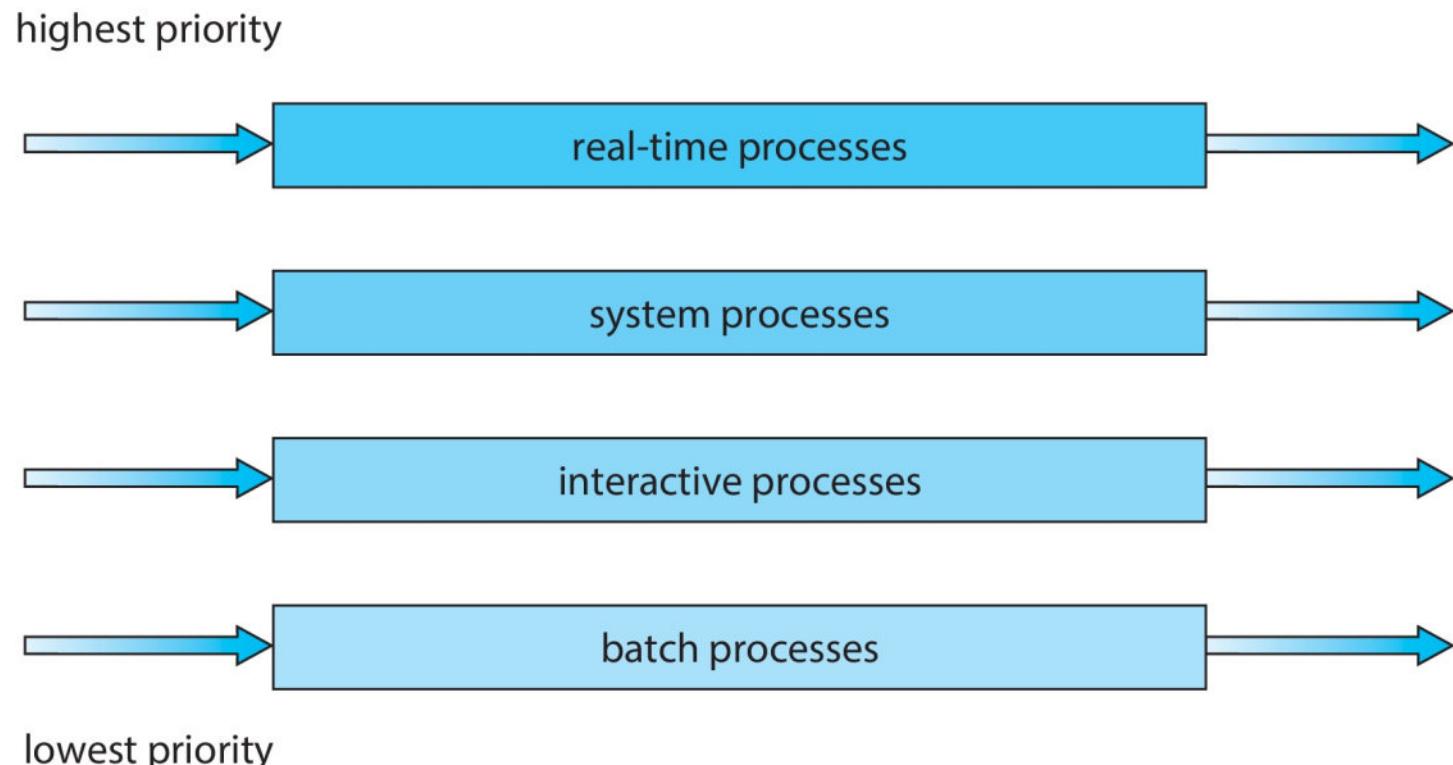
●

●



Example of Multilevel Queue

- Prioritization based upon process type



Multilevel Feedback Queue

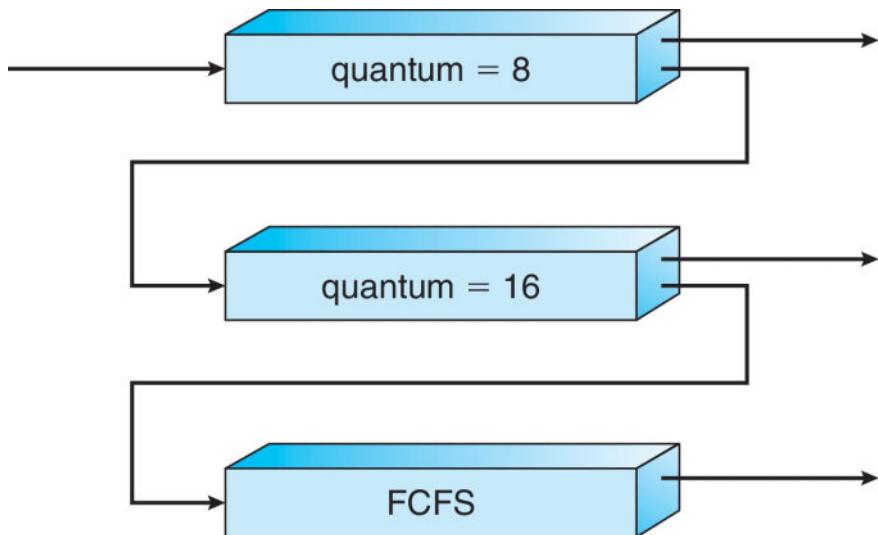
- **Motivation:** A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
 - number of queues
 - scheduling algorithms for each queue
 - method used to determine when to upgrade a process
 - method used to determine when to demote a process
 - method used to determine which queue a process will enter when that process needs service
- This scheme leaves I/O-bound and interactive processes—which are typically characterized by short CPU bursts — in the higher-priority queues and a process that waits too long in a lower-priority queue may be moved to a higher-priority queue



Example of Multilevel Feedback Queue

Three queues:

- ▶ Q0 – RR with time quantum 8 milliseconds
- ▶ Q1 – RR with time quantum 16 milliseconds
- ▶ Q2 – FCFS



Scheduling

- A new job enters queue Q0 which is served FCFS
 - ▶ When it gains CPU, job receives 8 milliseconds
 - ▶ If it does not finish in 8 milliseconds, job is moved to queue Q1
- At Q1 job is again served FCFS and receives 16 additional milliseconds
 - ▶ If it still does not complete, it is preempted and moved to queue Q2





Thread Scheduling

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





POSIX Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
 - **PTHREAD_SCOPE_PROCESS** schedules threads using PCS scheduling
 - **PTHREAD_SCOPE_SYSTEM** schedules threads using SCS scheduling
- Can be limited by OS – Linux and macOS only allow **PTHREAD_SCOPE_SYSTEM**
- Pthread IPC (Inter-process Communication) provides two functions for setting
 - `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
 - `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`





Pthread Scheduling API

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





Pthread Scheduling API (Cont.)

```
/* set the scheduling algorithm to PCS or SCS */

pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

/* create the threads */
for (i = 0; i < NUM_THREADS; i++)

    pthread_create(&tid[i], &attr, runner, NULL);

/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)

    pthread_join(tid[i], NULL);

}

/* Each thread will begin control in this function */

void *runner(void *param)
{

    /* do some work ... */

    pthread_exit(0);

}
```

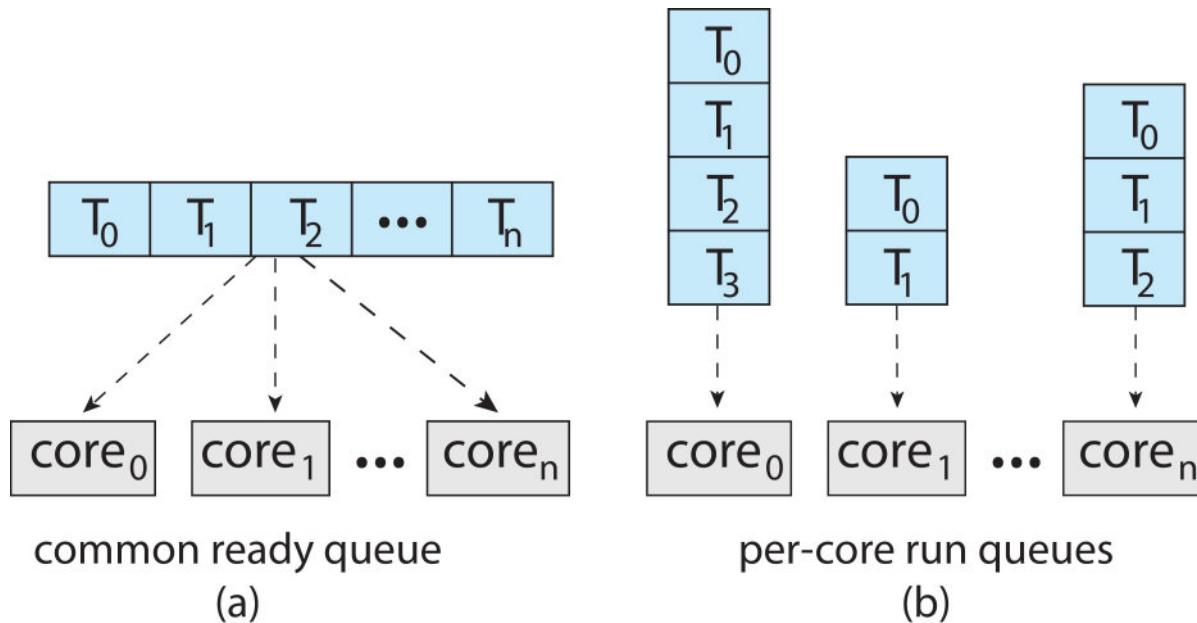


- CPU scheduling more complex when multiple CPUs are available
- *Multiprocessor* may be any one of the following architectures:
 - Multicore CPUs
 - Multithreaded cores
 - NUMA systems
 - Heterogeneous multiprocessing
- *Multiprocessor scheduling*
 - There is no one best solution



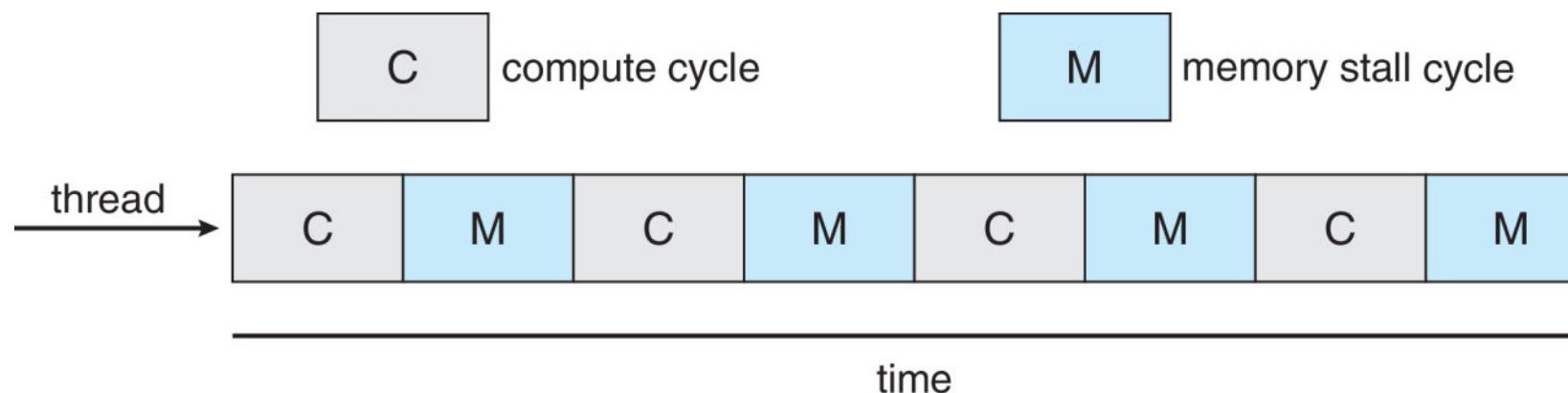
Multiple-Processor Scheduling (Cont.)

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

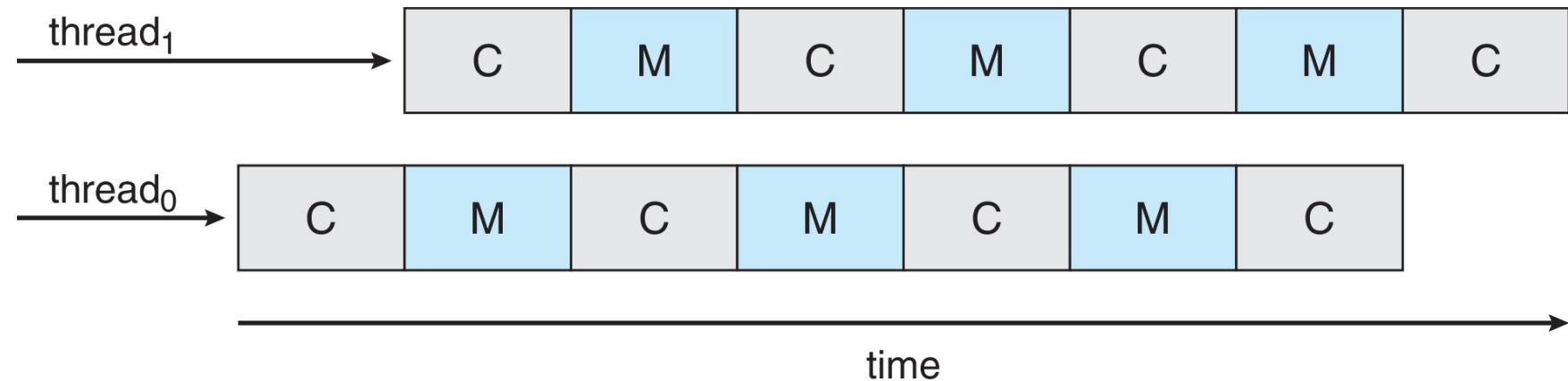


Multicore Processors

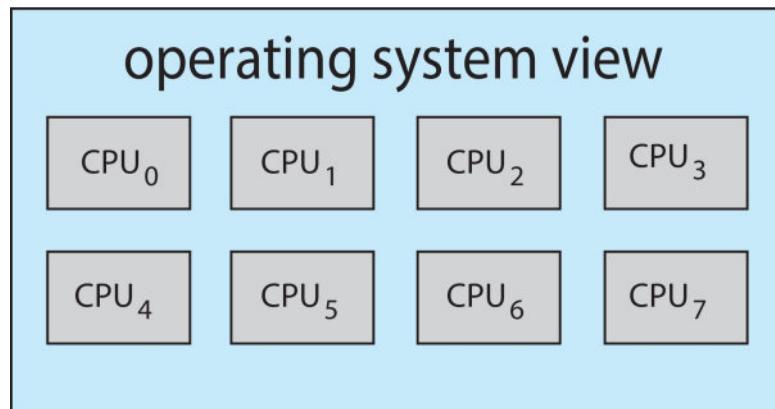
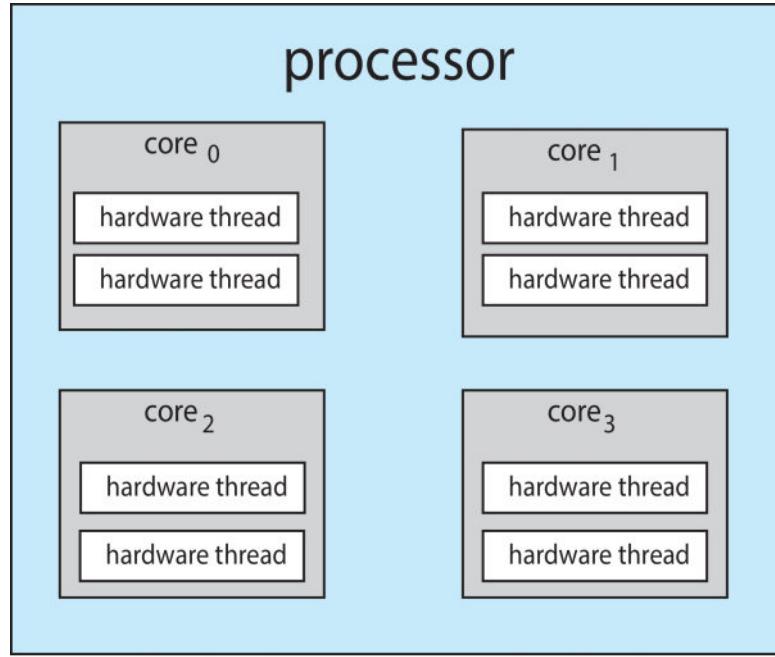
- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
 - Takes advantage of memory stall to make progress on another thread while memory retrieve happens



- Each core has > 1 hardware threads.
- If one thread has a memory stall, switch to another thread!



Multithreaded Multicore System

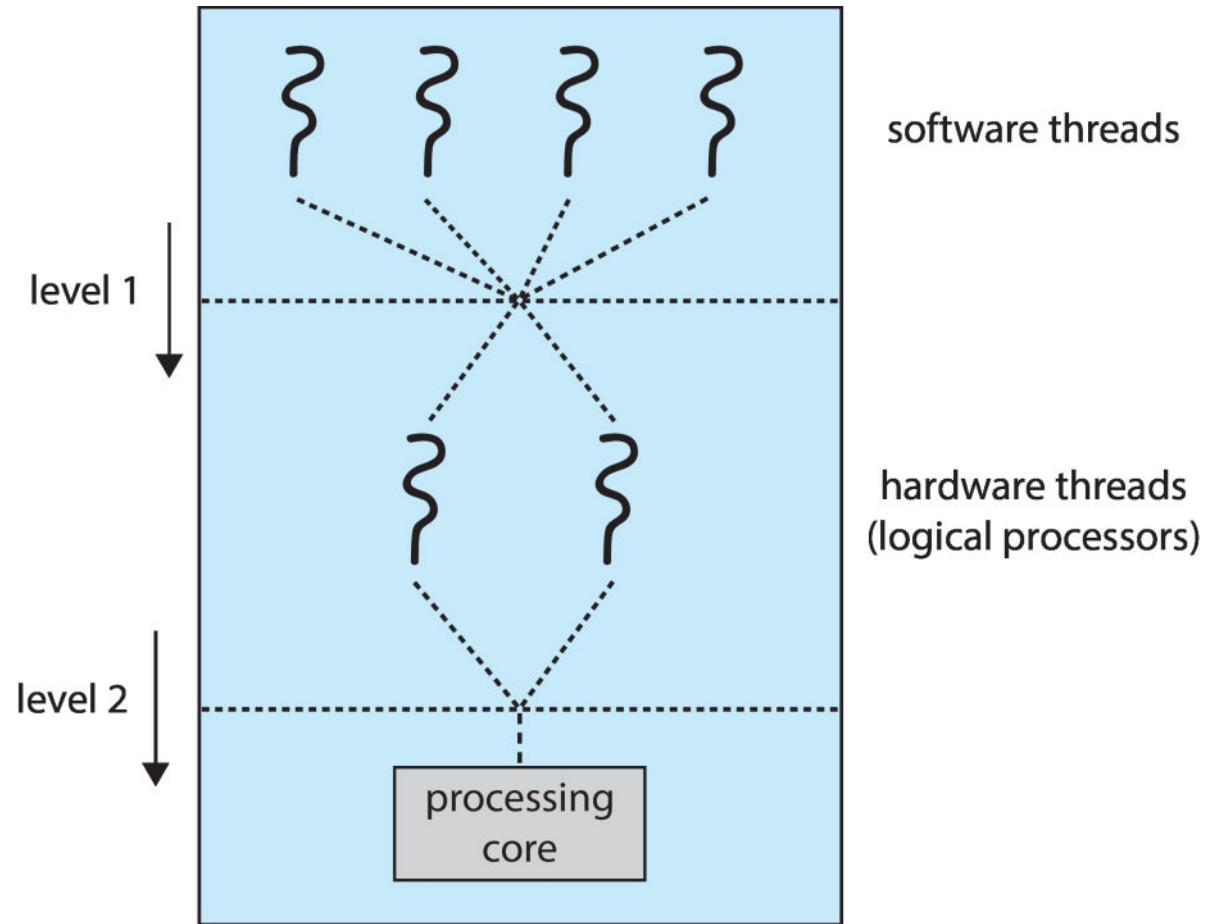


- *Chip-multithreading (CMT)* assigns each core multiple hardware threads (Intel refers to this as *hyperthreading*)
- Each hardware thread maintains its architectural state, such as *instruction pointer* and *register set*
- On a quad-core system with 2 hardware threads per core (e.g., Intel i7), the operating system sees 8 logical processors



- Two levels of scheduling:

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



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

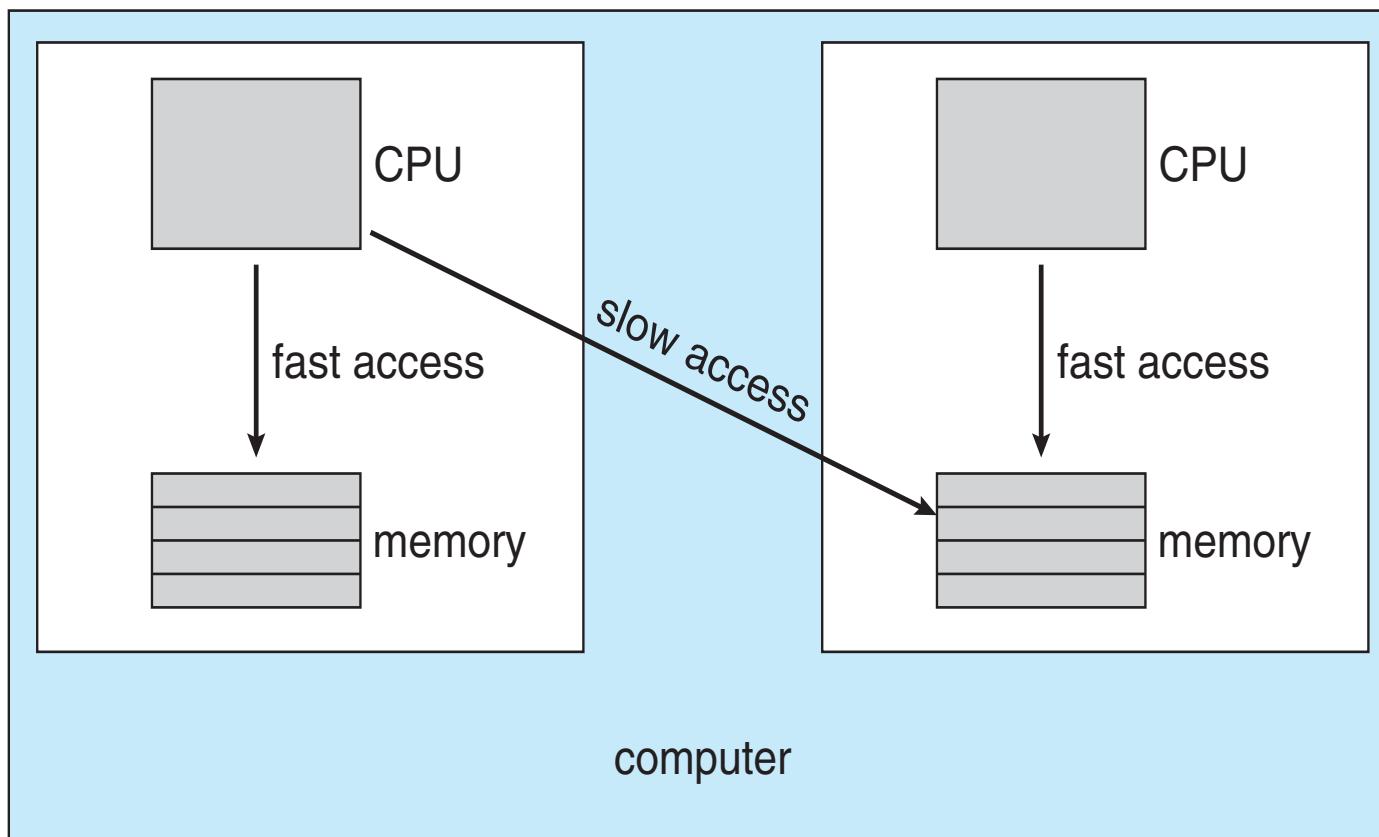


- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having affinity for a processor (i.e. “processor affinity”)
- Load balancing may affect processor affinity as a thread may be moved from one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



NUMA and CPU Scheduling

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





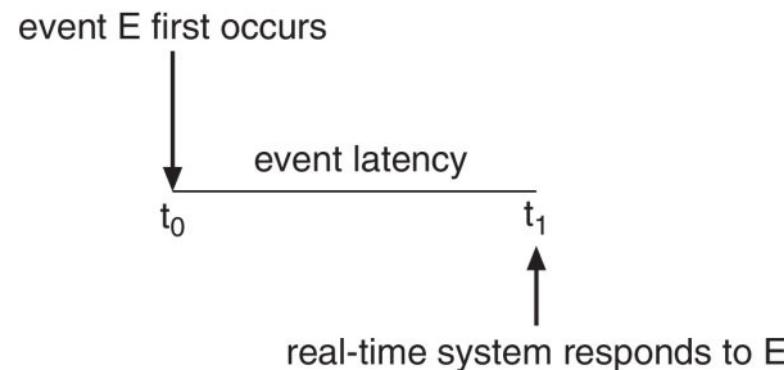
Real-Time CPU Scheduling

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

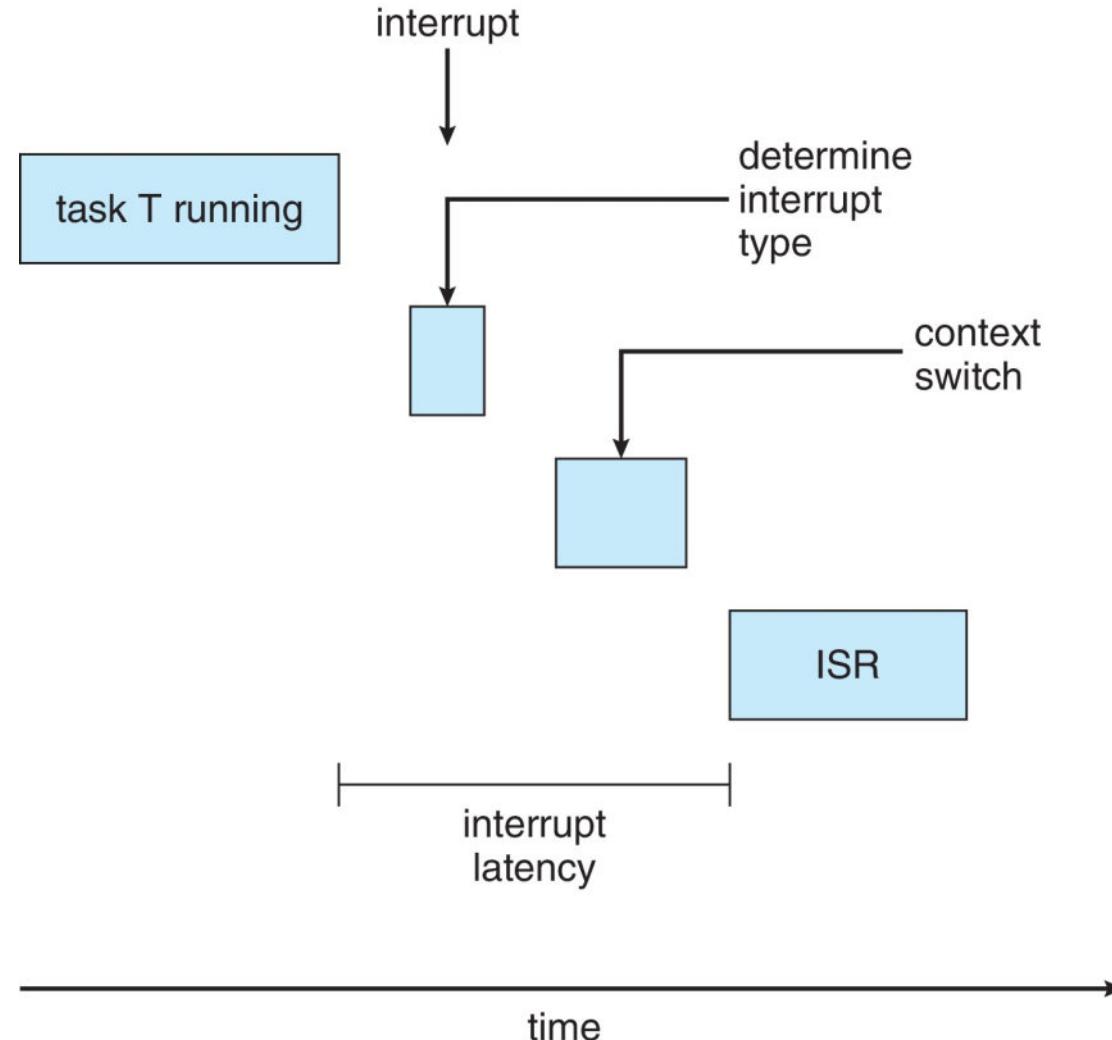


Types of latencies

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



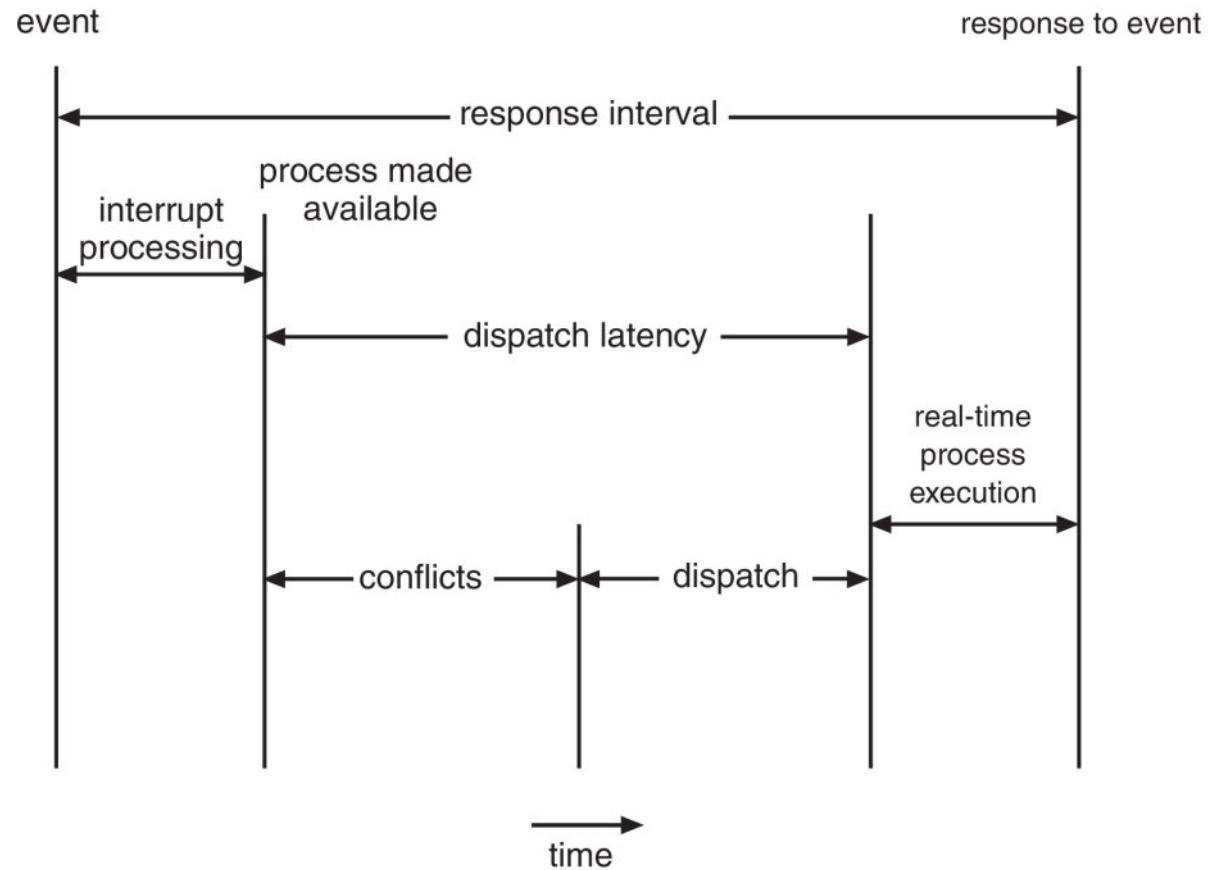
Interrupt Latency



Dispatch Latency

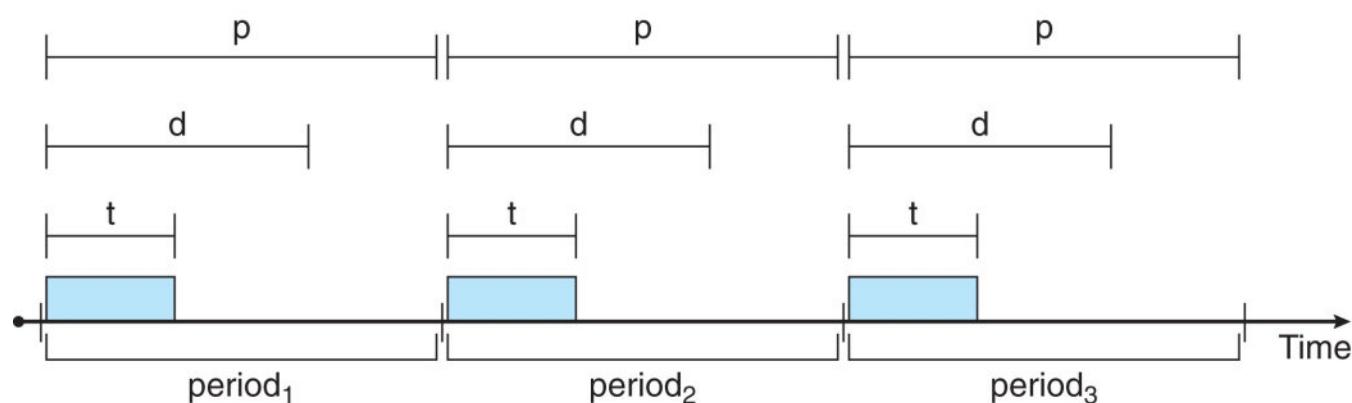
Conflict phase of dispatch latency:

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



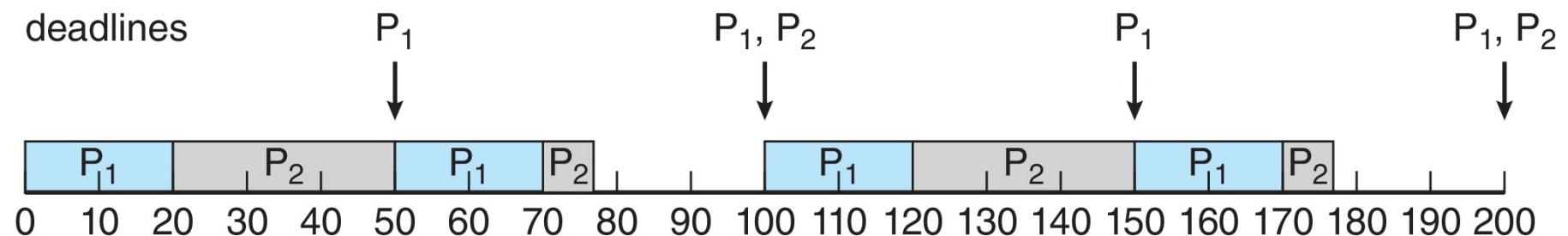
Priority-based Scheduling

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



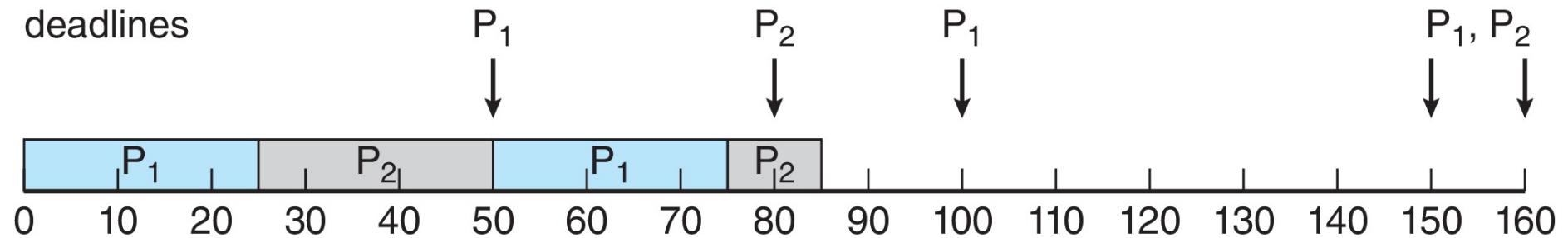
Rate Monotonic Scheduling

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



Missed Deadlines with Rate Monotonic Scheduling

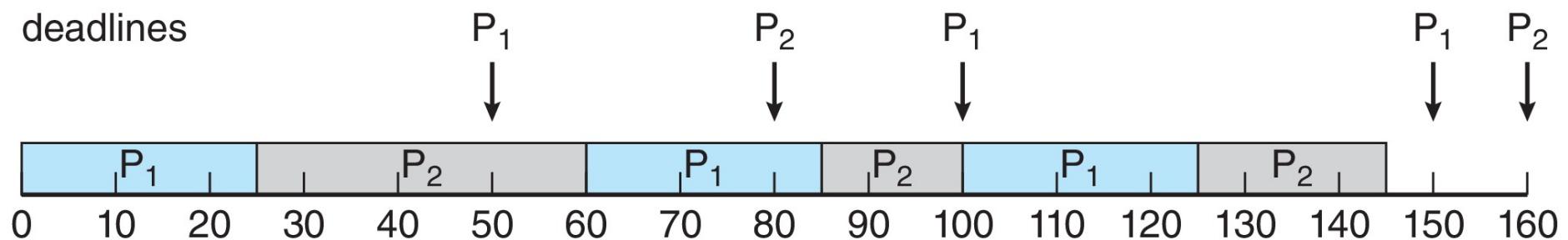
- Process P2 misses finishing its deadline at time 80



Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:

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





Proportional Share Scheduling

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



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



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



```
/* set the scheduling policy - FIFO, RR, or OTHER */
if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)

    fprintf(stderr, "Unable to set policy.\n");

/* create the threads */
for (i = 0; i < NUM_THREADS; i++)

    pthread_create(&tid[i], &attr, runner, NULL);

/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)

    pthread_join(tid[i], NULL);

}

/* Each thread will begin control in this function */

void *runner(void *param)
{

    /* do some work ... */

    pthread_exit(0);

}
```





Operating System Examples

- Linux scheduling
- Windows scheduling
- Solaris scheduling





Linux Scheduling Through Version 2.5

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





Linux Scheduling in Version 2.6.23 +

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

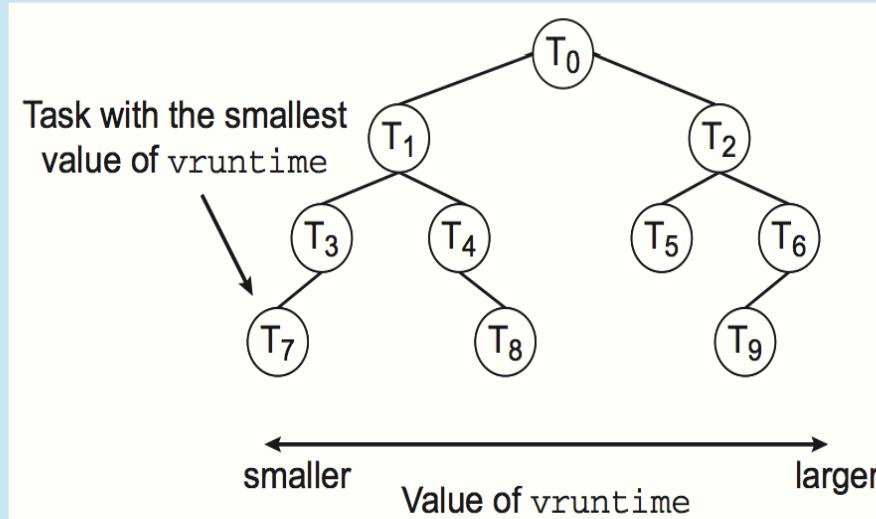


- Quantum calculated based on nice value from -20 to +19
 - Lower value is higher priority
 - Calculates target latency – interval of time during which task should run at least once
 - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task virtual run time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



CFS Performance

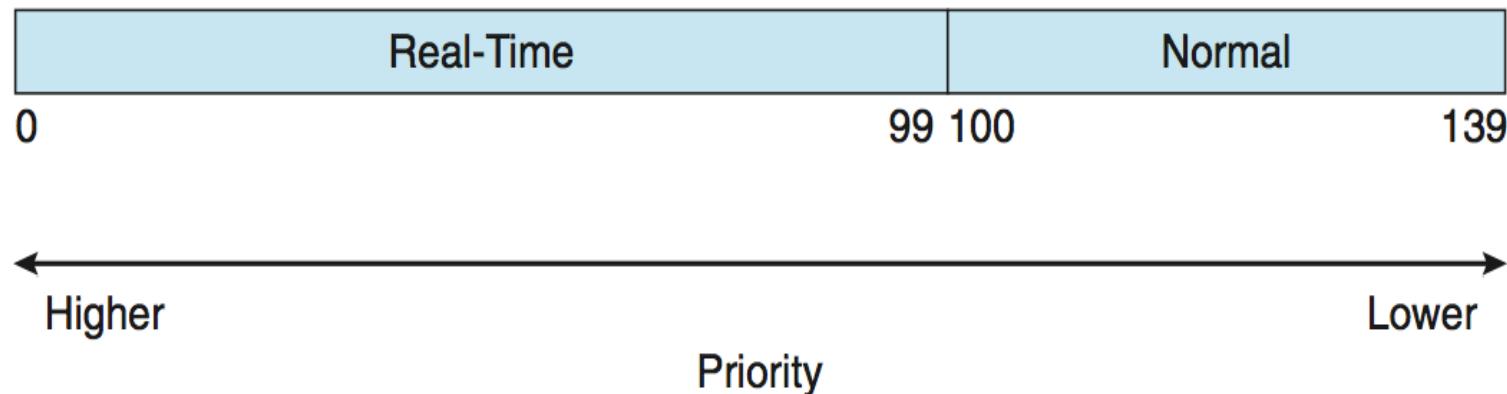
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



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

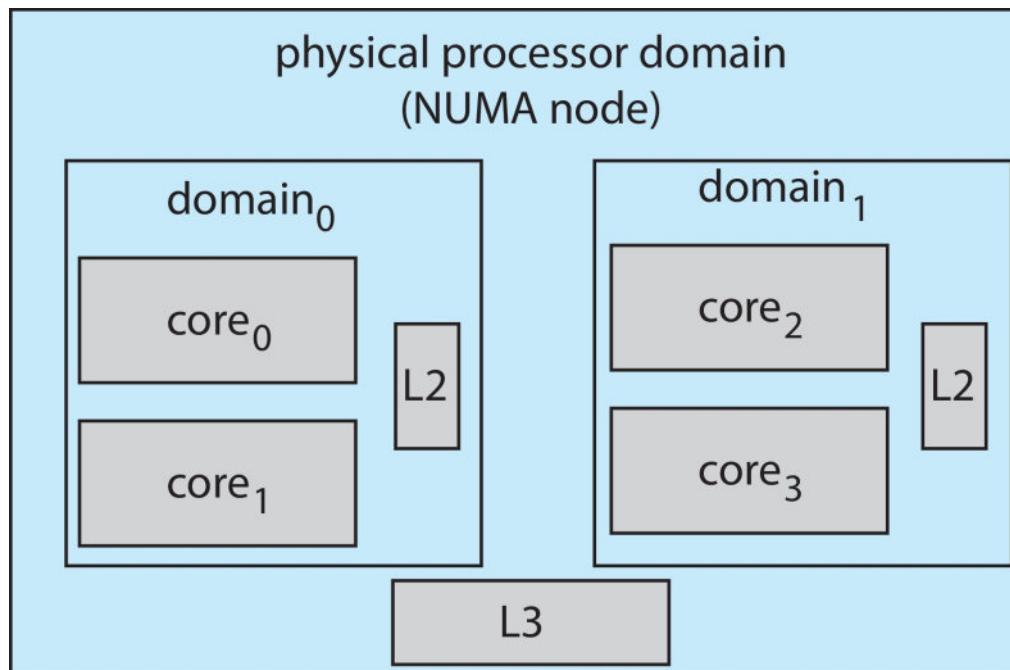


- Real-time scheduling according to *POSIX.1b*
 - Real-time tasks have static priorities
 - Real-time plus normal map into global priority scheme
 - Nice value of -20 maps to global priority 100
 - Nice value of +19 maps to priority 139



Linux Scheduling (Cont.)

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





Windows Scheduling

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





Windows Priority Classes

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





Windows Priority Classes (Cont.)

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



Windows Priorities

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



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

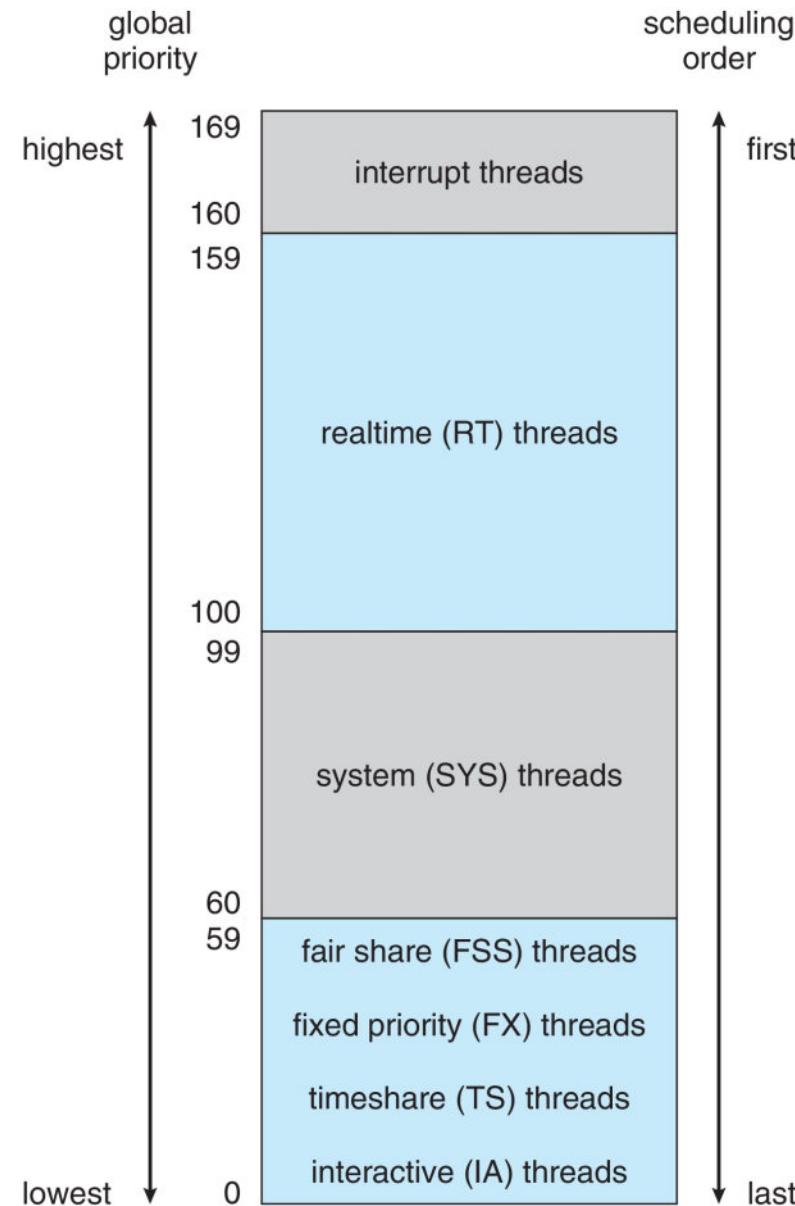


Solaris Dispatch Table

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



Solaris Scheduling





Solaris Scheduling (Cont.)

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



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

- Consider 5 processes arriving at time 0:

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12



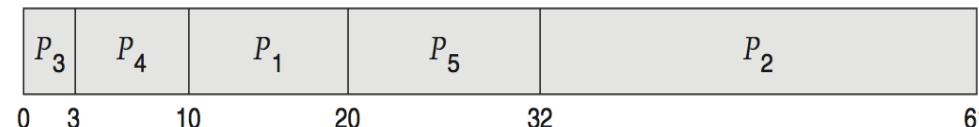
Deterministic Evaluation

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

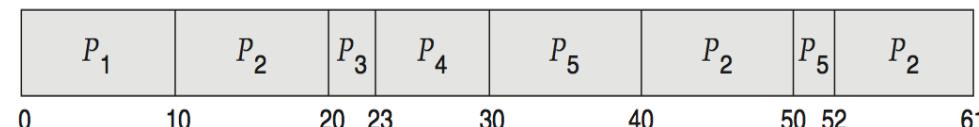
- FCS is 28ms:



- Non-preemptive SJF is 13ms:



- RR is 23ms:





Queueing Models

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



Little's Formula

- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:

$$n = \lambda \times W$$

- Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

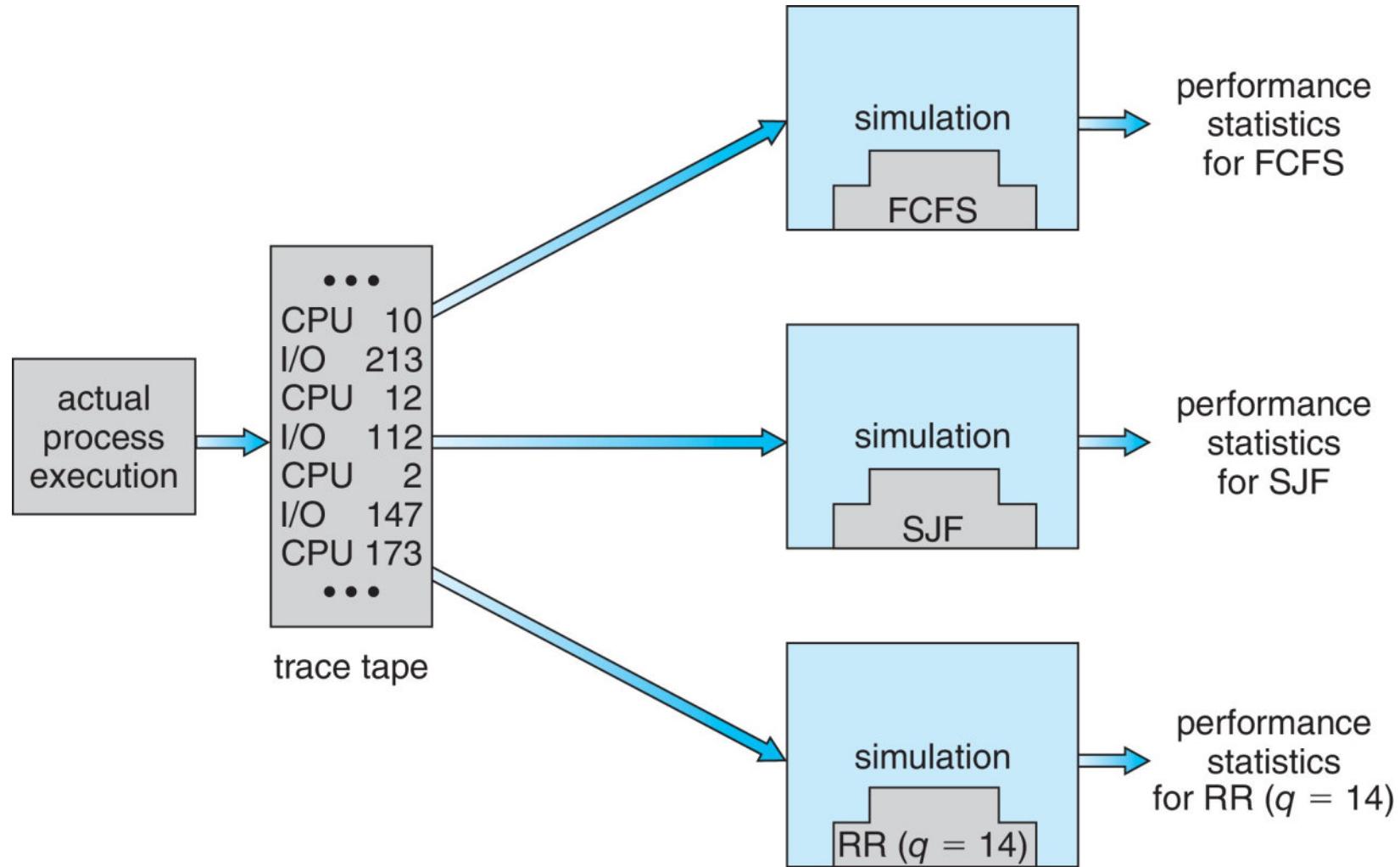


Simulations

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



Evaluation of CPU Schedulers by Simulation





Implementation

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



Summary

- *CPU scheduling* is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher.
- *Scheduling algorithms* may be either *preemptive* (where the CPU can be taken away from a process) or *nonpreemptive* (where a process must voluntarily relinquish control of the CPU). Almost all modern operating systems are preemptive.
- Scheduling algorithms can be evaluated according to the following five criteria: (1) *CPU utilization*, (2) *throughput*, (3) *turnaround time*, (4) *waiting time*, and (5) *response time*.
- *First-come, first-served (FCFS)* scheduling is the simplest scheduling algorithm, but it can cause short processes to wait for very long processes.



Summary (Cont.)

- ***Shortest-job-first (SJF)*** scheduling is provably optimal, providing the shortest average waiting time. Implementing SJF scheduling is difficult, however, because predicting the length of the next CPU burst is difficult.
- ***Round-robin (RR)*** scheduling allocates the CPU to each process for a time quantum. If the process does not relinquish the CPU before its time quantum expires, the process is preempted, and another process is scheduled to run for a time quantum.
- ***Priority scheduling*** assigns each process a priority, and the CPU is allocated to the process with the highest priority. Processes with the same priority can be scheduled in FCFS order or using RR scheduling.



Summary (Cont.)

- *Multilevel queue scheduling* partitions processes into several separate queues arranged by priority, and the scheduler executes the processes in the highest-priority queue. Different scheduling algorithms may be used in each queue.
- *Multilevel feedback queues* are similar to multilevel queues, except that a process may migrate between different queues.
- *Multicore processors* place one or more CPUs on the same physical chip, and each CPU may have more than one hardware thread. From the perspective of the operating system, each hardware thread appears to be a logical CPU.
- *Load balancing* on multicore systems equalizes loads between CPU cores, although migrating threads between cores to balance loads may invalidate cache contents and therefore may increase memory access times.



Summary (Cont.)

- *Soft real-time scheduling* gives priority to real-time tasks over non-real-time tasks. Hard real-time scheduling provides timing guarantees for real-time tasks,
- *Rate-monotonic real-time scheduling* schedules periodic tasks using a static priority policy with preemption.
- *Earliest-deadline-first (EDF)* scheduling assigns priorities according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority.
- *Proportional share scheduling* allocates T shares among all applications. If an application is allocated N shares of time, it is ensured of having $\frac{N}{T}$ of the total processor time.



Summary (Cont.)

- **Linux** uses the *completely fair scheduler (CFS)*, which assigns a proportion of CPU processing time to each task. The proportion is based on the virtual runtime (**vruntime**) value associated with each task.
- **Windows** scheduling uses a preemptive, 32-level priority scheme to determine the order of thread scheduling.
- **Solaris** identifies six unique scheduling classes that are mapped to a global priority. CPU-intensive threads are generally assigned lower priorities (and longer time quanta), and I/O-bound threads are usually assigned higher priorities (with shorter time quanta.)
- *Modeling and simulations* can be used to evaluate a CPU scheduling algorithm.



End of Chapter 5



What Is an
OPERATING SYSTEM (OS)
and How Does It Work

CLEVERISM.COM

Chapter 6: Synchronization Tools





Chapter 6: Outline

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





Objectives

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





Background

- Processes can execute *concurrently* (or *in parallel*)
 - May be interrupted at any time, partially completing execution
- Concurrent access to *shared data* may result in *data inconsistency*
- Maintaining data consistency requires mechanisms to ensure the *orderly execution of cooperating processes*
- Illustration of the problem:
 - Suppose that we wanted to provide a solution to the *consumer-producer problem* that fills *all* the buffers. We can do so by having an integer *counter* that keeps track of the number of full buffers. Initially, *counter* is set to 0. It is *incremented* by the producer after it adds a new item to the buffer and is *decremented* by the consumer after it consumes an item from the buffer

```
#define BUFFER_SIZE 8
/* 8 buffers */

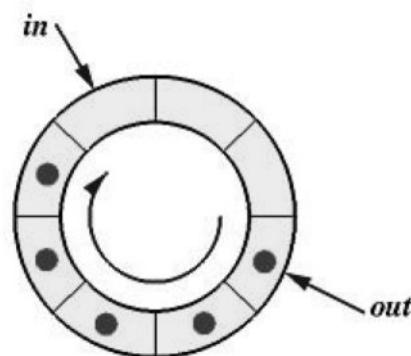
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];

int in = 0;
int out = 0;
int counter = 0;
```



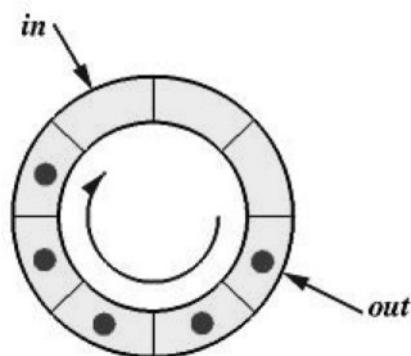
Producer

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



Consumer

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





Race Condition

- `counter++`; could be implemented as

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

- `counter--`; could be implemented as

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

- Consider this execution *interleaving* with “`counter = 5`” initially:

S0: producer execute `register1 = counter` {`register1 = 5`}

S1: producer execute `register1 = register1 + 1` {`register1 = 6`}

S2: consumer execute `register2 = counter` {`register2 = 5`}

S3: consumer execute `register2 = register2 - 1` {`register2 = 4`}

S4: producer execute `counter = register1` {`counter = 6` }

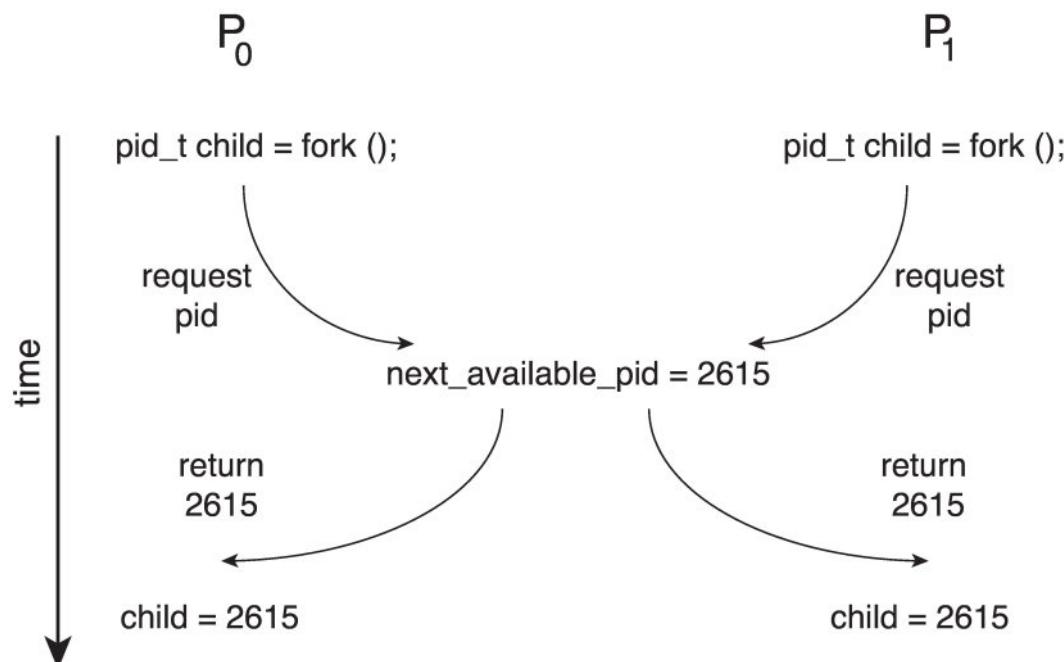
S5: consumer execute `counter = register2` {`counter = 4`}

=> Data inconsistency



Race Condition (Cont.)

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



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





Critical-Section Problem

- Consider system of n processes $\{P_0, P_1, \dots, P_{n-1}\}$
 - Each process has *critical section* (i.e., segment of code)
 - ▶ Process may be changing common variables, updating table, writing file, etc.
 - ▶ When one process in critical section, no other may be in its critical section
 - *Critical-section problem* needs to design a protocol to solve this
 - Each process must
 - ask permission to enter critical section in *entry section*,
 - may follow critical section with *exit section*,
 - then *remainder section*
- ```
do {
 entry section
 critical section
 exit section
 remainder section
} while(true);
```





# Critical Section (CS)

- General structure of the process  $P_i$

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





# Solution to Critical-Section Problem

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





# Critical-Section Handling in OS

- Two approaches depending on if kernel is preemptive or non-preemptive
  - *Preemptive* – allows preemption of process when running in kernel mode
  - *Non-preemptive* – runs until exits kernel mode, blocks, or voluntarily yields CPU
    - ▶ Essentially free of *race conditions* in kernel mode





# Peterson's Solution

- Not guaranteed to work on modern architectures!
  - (But good algorithmic description of solving the problem)
- *Two-processes* solution
- Assume that the **load** and **store** machine-language instructions are *atomic*; that is, it cannot be interrupted
- The two processes share *two variables*:
  - **int turn;**
  - **boolean flag[i]**
    - ▶ The variable **turn** indicates whose turn it is to enter the critical section
    - ▶ The **flag[]** array is used to indicate if a process is ready to enter the critical section
      - **flag[i] = true** implies that process  $P_i$  is ready!





# Algorithm for Process $P_i$

```
while (true) {
 flag[i] = true;
 turn = j;
 while (flag[j] && turn == j)
 ; /* do nothing */
 /* critical section */
 flag[i] = false;
 /* remainder section */
}
}
```





# Peterson's Solution (Cont.)

■ Provable that the three CS requirement are met:

1. *Mutual exclusion* is preserved
  - $P_i$  enters CS only if: either `flag[j] = false` or `turn = i`
2. *Progress* requirement is satisfied
3. *Bounded-waiting* requirement is met





# Remarks on Peterson's Solution

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





# Example of Peterson's Solution

- Two threads share the data:

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

- *Thread 1* performs

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

- *Thread 2* performs

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

- What is the expected output?

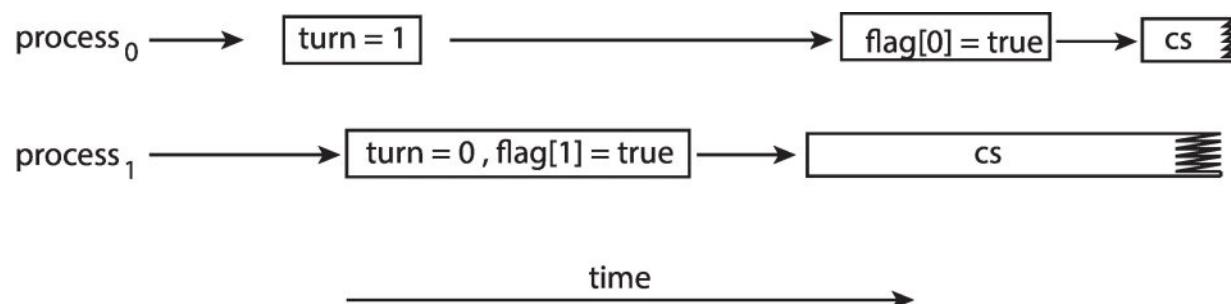


# Example of Peterson's Solution

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

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

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



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





# Synchronization Hardware

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





# Memory Barriers

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





# Example of Memory Barrier

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

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

- *Thread 2* now performs

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





# Hardware Instructions

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





# test\_and\_set Instruction

## ■ Definition:

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

1. Executed atomically
2. Returns the original value of passed parameter (i.e., **\*target**)
3. Set the new value of passed parameter to **true**  
(i.e., **\*target=true**)





# Solution using *test\_and\_set()*

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

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

 /* critical section */

 lock = false;

 /* remainder section */

} while (true);
```





# compare\_and\_swap Instruction

## ■ 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** the value of the passed parameter **new\_value** but only if **\*value == expected** is true. That is, the swap takes place only under this condition.





# Solution using *compare\_and\_swap*

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

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

 /* critical section */

 lock = 0;

 /* remainder section */

}
```





# Bounded-waiting Mutual Exclusion with compare-and-swap

```
while (true) {
 waiting[i] = true;
 key = 1;
 while (waiting[i] && key == 1)

 key = compare_and_swap(&lock,0,1);

 waiting[i] = false;

 /* critical section */

 j = (i + 1) % n;

 while ((j != i) && !waiting[j])

 j = (j + 1) % n;

 if (j == i)

 lock = 0;

 else

 waiting[j] = false;

 /* remainder section */

}
```





# Atomic Variables

- Typically, instructions such as *compare-and-swap* are used as building blocks for other synchronization tools.
- One tool is an *atomic variable* that provides *atomic* (uninterruptible) updates on basic data types such as Integers and Booleans.
- For example, the `increment()` operation on the atomic variable `sequence` ensures `sequence` is incremented without interruption:

```
increment(&sequence);
```





# Atomic Variables (Cont.)

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

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

 do {
 temp = *v;
 }
 while

(temp != compare_and_swap(v, temp, temp+1));

}
```





# Mutex Locks

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





# Solution to Critical-section Problem using Locks

```
while (true) {
 acquire lock;

 critical section;

 release lock;

 remainder section;
}
```





# Mutex Lock Definitions

- ▶ **acquire()** {  
    **while** (!available)  
        ;  
        /\* busy wait \*/  
  
    **available = false**;  
  
}
  
  - ▶ **release()** {  
  
    **available = true**;  
  
}
- 
- These two functions must be implemented *atomically*
  - Both *test-and-set* and *compare-and-swap* can be used to implement these functions





# Semaphore

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

```
wait(S) {
 while (S <= 0)
 ; // busy wait
 S--;
}
```
- Definition of the **signal()** operation

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



# Semaphore Usage

- *Counting semaphore* – integer value can range over an unrestricted domain
- *Binary semaphore* – integer value can range only between 0 and 1
  - Same as a mutex lock
  - Can solve various synchronization problems
- Can implement a counting semaphore  $S$  as a binary semaphore
- Consider  $P_1$  and  $P_2$  that require  $S_1$  to happen before  $S_2$ 
  - Create a semaphore “synch” initialized to 0

$P1:$

$s_1;$

`signal(synch);`

$P2:$

`wait(synch);`

$s_2;$





# Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Thus, the implementation becomes the critical-section problem where the `wait()` and `signal()` code are placed in the critical section
  - Could now have *busy waiting* in critical-section implementation
    - ▶ But implementation code is short
    - ▶ Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution





# Semaphore Implementation with no Busy waiting

- With each semaphore there is an *associated waiting queue*
- Each entry in a waiting queue has two data items:
  - **value** (of type integer)
  - **pointer** to next record in the list
- Two operations:
  - *block* – place the process invoking the operation on the appropriate waiting queue
  - *wakeup* – remove one of processes in the waiting queue and place it in the ready queue

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





## Implementation with no Busy waiting (Cont.)

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

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





# Problems with Semaphores

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





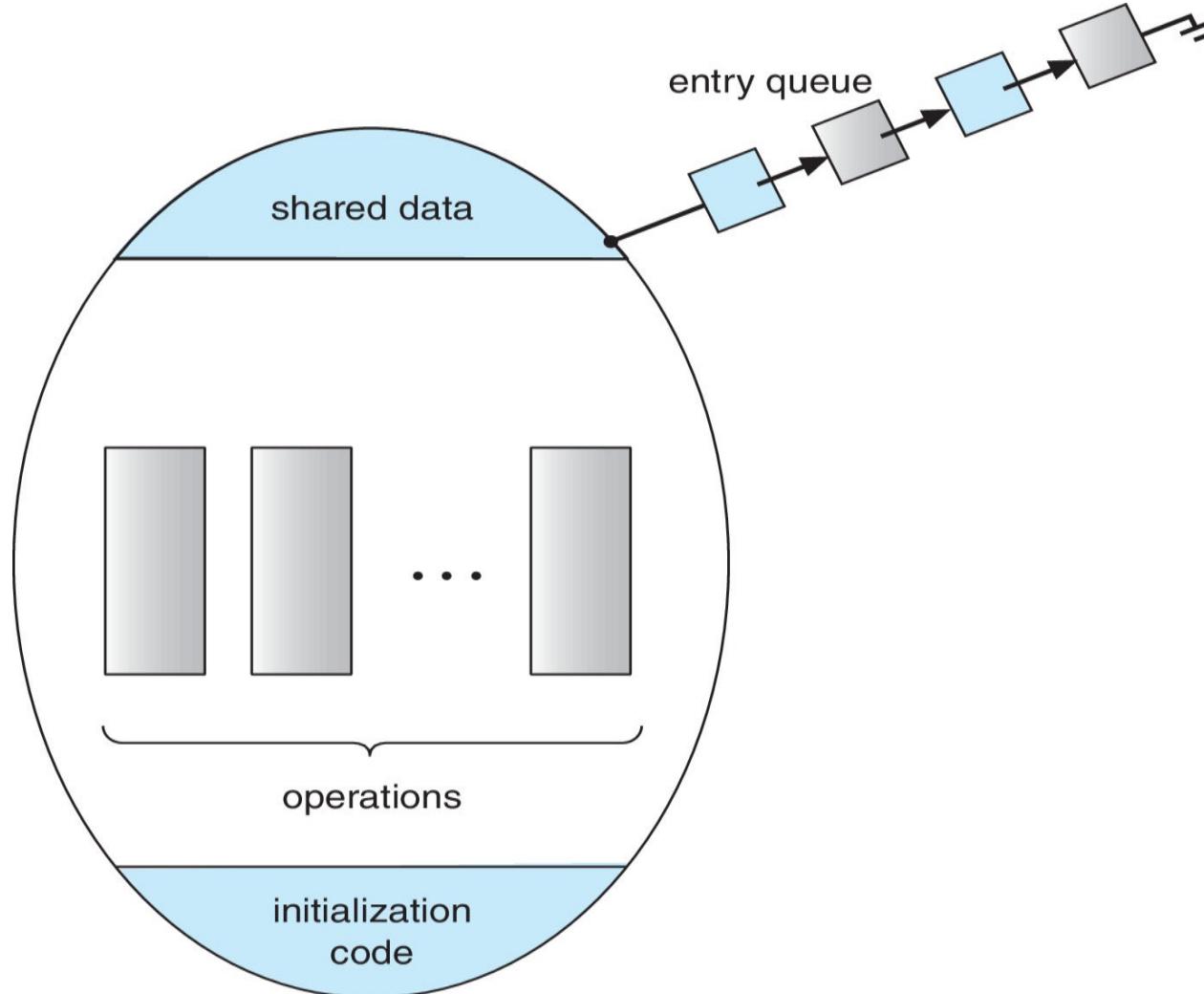
# Monitors

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

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



# Schematic View of a Monitor



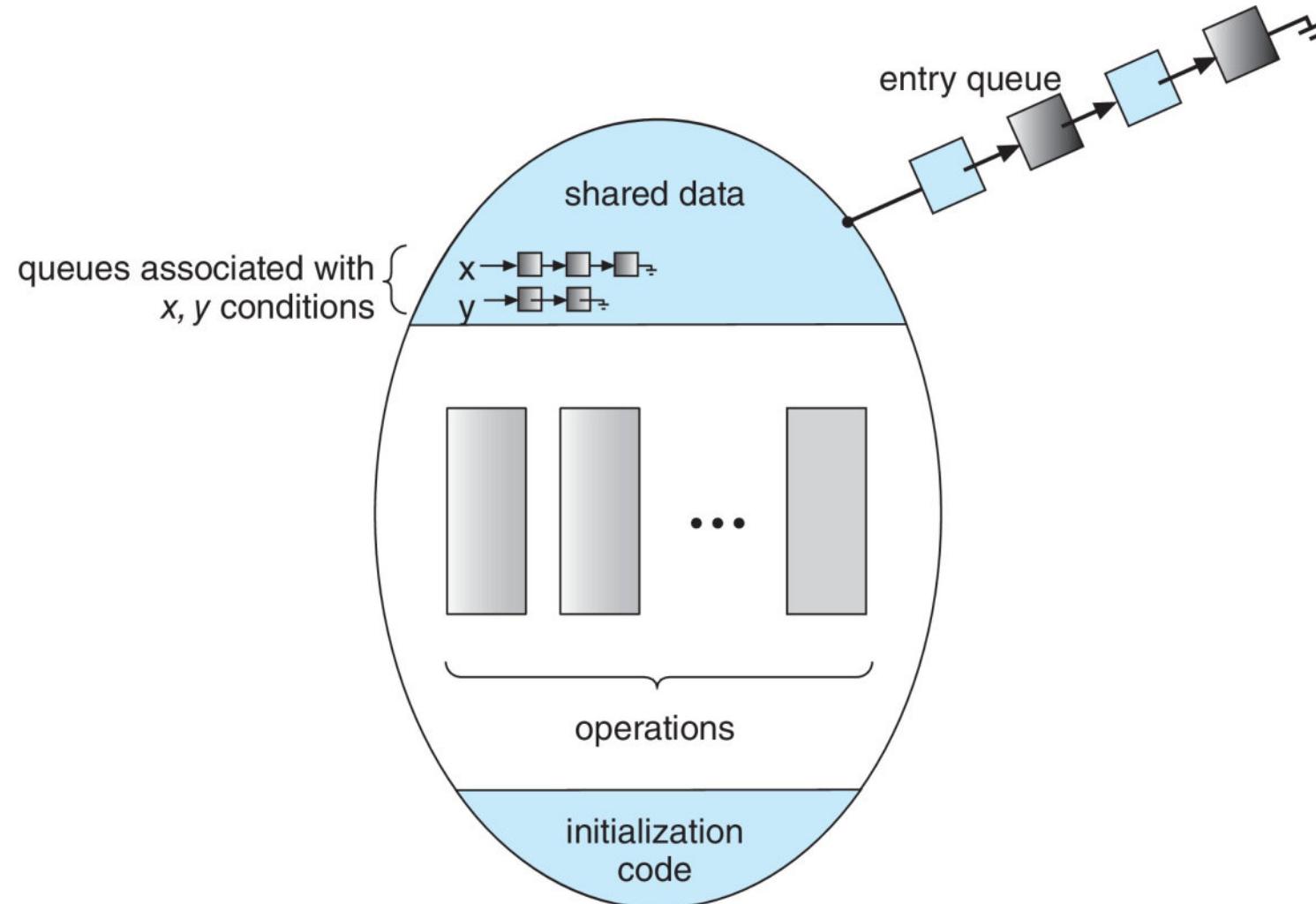


# Condition Variables

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



# Monitor with Condition Variables





# Condition Variables Choices

- If process **P** invokes `x.signal()`, and process **Q** is suspended in `x.wait()`, what should happen next?
  - ▶ Both **Q** and **P** can't 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: **Mesa**, **C#**, **Java**





# Monitor Implementation using Semaphores

## ■ Variables

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

## ■ Each function **F** will be replaced by

```
wait(mutex);

...
body of F;

...
if (next_count > 0)
 signal(next)
else
 signal(mutex);
```

## ■ *Mutual exclusion* within a monitor is ensured





## Monitor Implementation – Condition Variables

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

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

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

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





# Monitor Implementation (Cont.)

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

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





# Resuming Processes within a Monitor

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





# Single Resource allocation

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

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

R.release;
```

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





# A Monitor to Allocate Single Resource

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





# Liveness

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



# Liveness (Cont.)

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

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

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





# Liveness (Cont.)

Other forms of deadlock:

- **Starvation** – indefinite blocking

- A process may never be removed from the semaphore queue in which it is suspended

- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process

- Solved via **priority-inheritance protocol**



- Consider the scenario with three processes  $P_1$ ,  $P_2$ , and  $P_3$ .
  - $P_1$  has the highest priority,  $P_2$  the next highest, and  $P_3$  the lowest.
  - Assume a resource  $P_3$  is assigned a resource  $R$  that  $P_1$  wants. Thus,  $P_1$  must wait for  $P_3$  to finish using the resource.
  - However,  $P_2$  becomes runnable and preempts  $P_3$ .
  - What has happened is that  $P_2$  - a process with a lower priority than  $P_1$  - has indirectly prevented  $P_3$  from gaining access to the resource.
- To prevent this from occurring, a *priority inheritance protocol* is used.
  - This simply allows the priority of the highest thread waiting to access a shared resource to be assigned to the thread currently using the resource.
  - Thus, the current owner of the resource is assigned the priority of the highest priority thread wishing to acquire the resource.





# Summary

- A *race condition* occurs when processes have concurrent access to shared data and the final result depends on the particular order in which concurrent accesses occur. Race conditions can result in corrupted values of shared data.
- A *critical section* is a section of code where shared data may be manipulated and a possible race condition may occur. The critical-section problem is to design a protocol whereby processes can synchronize their activity to cooperatively share data.
- A *solution to the critical-section problem* must satisfy the following three requirements: (1) *mutual exclusion*, (2) *progress*, and (3) *bounded waiting*. Mutual exclusion ensures that only one process at a time is active in its critical section. Progress ensures that programs will cooperatively determine what process will next enter its critical section. Bounded waiting limits how much time a program will wait before it can enter its critical section.



# Summary (Cont.)

- Software solutions to the critical-section problem, such as *Peterson's solution*, do not work well on modern computer architectures.
- *Hardware support* for the critical-section problem includes memory barriers; hardware instructions, such as the compare-and-swap instruction; and atomic variables.
- A *mutex lock* provides mutual exclusion by requiring that a process acquire a lock before entering a critical section and release the lock on exiting the critical section.
- *Semaphores*, like mutex locks, can be used to provide mutual exclusion. However, whereas a mutex lock has a binary value that indicates if the lock is available or not, a semaphore has an integer value and can therefore be used to solve a variety of synchronization problems.





# Summary (Cont.)

- A *monitor* is an abstract data type that provides a high-level form of process synchronization. A monitor uses condition variables that allow processes to wait for certain conditions to become true and to signal one another when conditions have been set to true.
- Solutions to the critical-section problem may suffer from *liveness problems*, including *deadlock*.
- The *various tools* that can be used to solve the critical-section problem as well as to synchronize the activity of processes can be evaluated under varying levels of contention. Some tools work better under certain contention loads than others.



# End of Chapter 6



What Is an  
**OPERATING SYSTEM (OS)**  
and How Does It Work

CLEVERISM.COM

# Chapter 7: Synchronization Examples



What Is an  
**OPERATING SYSTEM (OS)**  
and How Does It Work

CLEVERISM.COM



# Chapter 7: Synchronization Examples

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



# Classical Problems of Synchronization

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



# Bounded-Buffer Problem

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

## ■ The structure of the producer process

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





# Bounded Buffer Problem (Cont.)

## ■ The structure of the consumer process

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

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

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

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

 ...
}
```



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





# Readers-Writers Problem (Cont.)

## ■ The structure of a writer process

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

 ...
 /* writing is performed */

 ...

 signal(rw_mutex);
}
```

## ■ The structure of a reader process

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

 . . . /* reading is performed */

 . . .

 wait(mutex);
 read_count--;
 if (read_count == 0)
 signal(rw_mutex);
 signal(mutex);

}
```



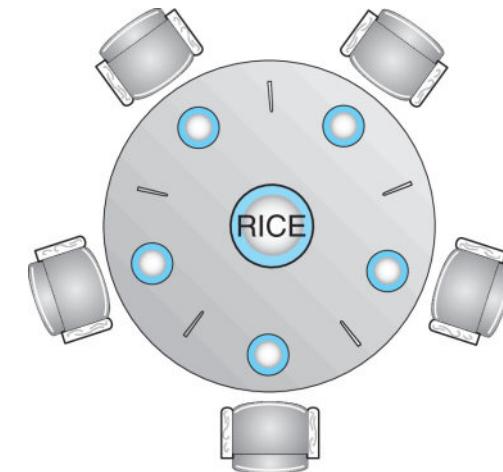


# Readers-Writers Problem Variations

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

# Dining-Philosophers Problem

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





# Dining-Philosophers Problem Algorithm

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

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

 /* eat for awhile */

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

 /* think for awhile */
}
```

- What is the problem with this algorithm?





# Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers {
 enum { THINKING, HUNGRY, EATING } state [5] ;
 condition self [5];
 void pickup (int i) {
 state[i] = HUNGRY;
 test(i);
 if (state[i] != EATING) self[i].wait;
 }
 void putdown (int i) {
 state[i] = THINKING;
 // test left and right neighbors
 test((i + 4) % 5);
 test((i + 1) % 5);
 }
}
```





# Solution to Dining Philosophers (Cont.)

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

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





## Solution to Dining Philosophers (Cont.)

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

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

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

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

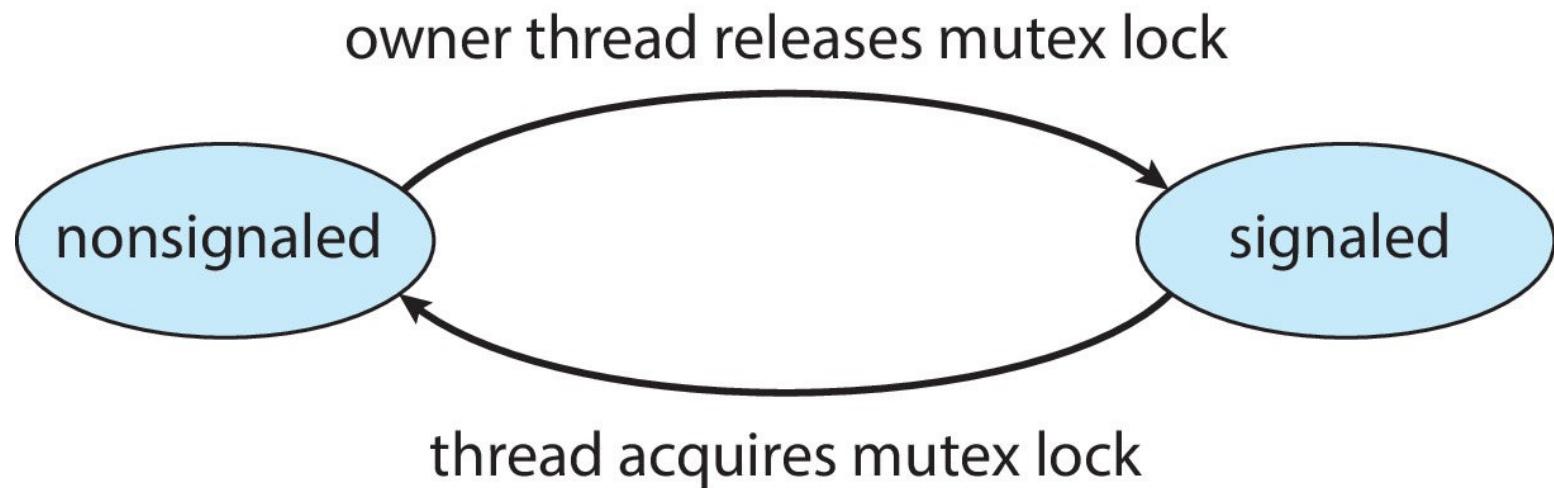
- No deadlock, but starvation is possible



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



- Mutex dispatcher object



## ■ Linux:

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

## ■ Linux provides:

- Semaphores
- atomic integers
- spinlocks
- reader-writer versions of both

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



## ■ Atomic variables

- `atomic_t` is the type for atomic integer

## ■ Consider the variables

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

### *Atomic Operation*

```
atomic_set(&counter,5);
atomic_add(10,&counter);
atomic_sub(4,&counter);
atomic_inc(&counter);
value = atomic_read(&counter);
```

### *Effect*

```
counter = 5
counter = counter + 10
counter = counter - 4
counter = counter + 1
value = 12
```



# POSIX Synchronization

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





# POSIX Mutex Locks

## ■ Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

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

## ■ Acquiring and releasing the lock

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

/* critical section */

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





# POSIX Semaphores

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



# POSIX Named Semaphores

- Creating an initializing the semaphore:

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

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

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

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

/* critical section */

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





# POSIX Unnamed Semaphores

## ■ Creating and initializing the semaphore:

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

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

## ■ Acquiring and releasing the semaphore:

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

/* critical section */

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





# POSIX Condition Variables

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

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

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



# POSIX Condition Variables

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

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

pthread_mutex_unlock(&mutex);
```

- Thread signaling another thread waiting on the condition variable:

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



# Java Synchronization

■ Java provides rich set of synchronization features:

- Java monitors
- Reentrant locks
- Semaphores
- Condition variables



# Java Monitors

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

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

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

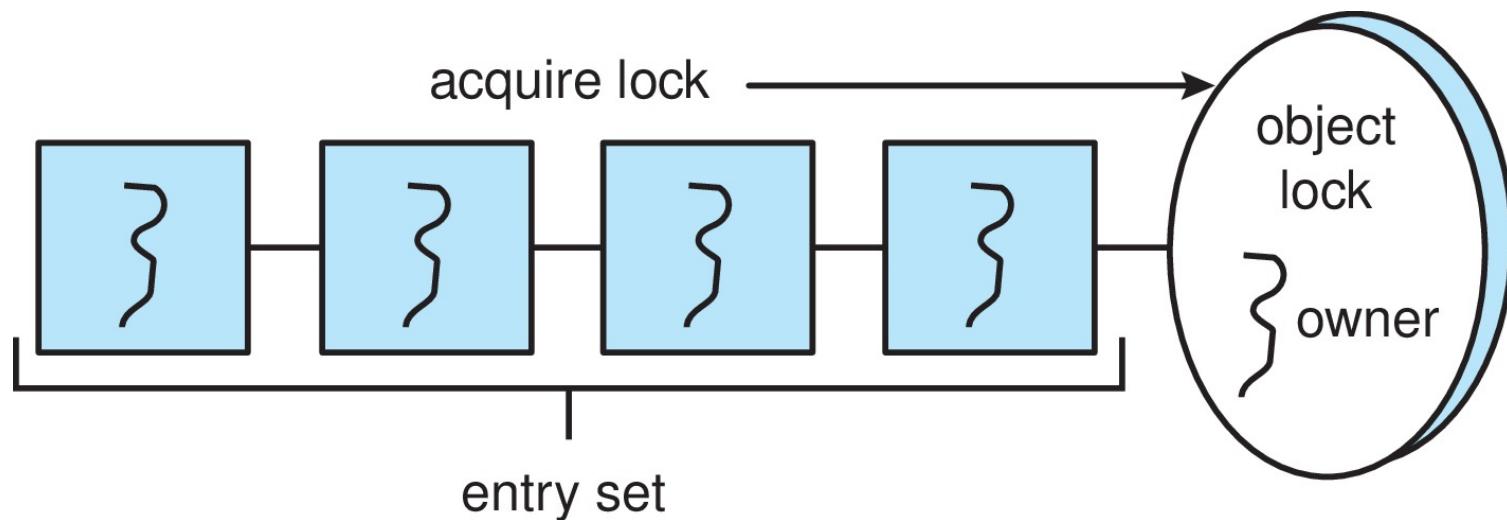
 public BoundedBuffer() {
 count = 0;
 in = 0;
 out = 0;
 buffer = (E[]) new Object[BUFFER_SIZE];
 }

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

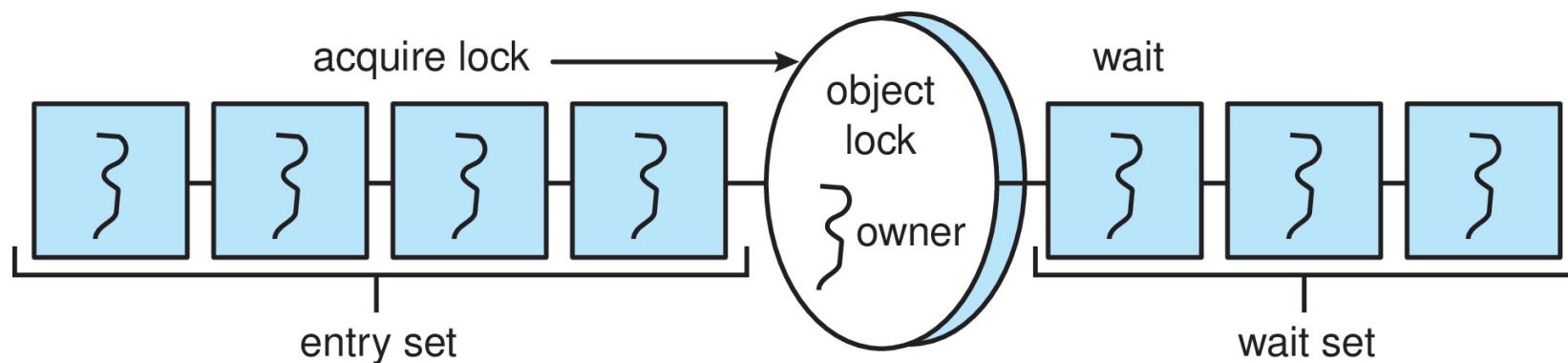
 /* Consumers call this method */
 public synchronized E remove() {
 /* See Figure 7.11 */
 }
}
```



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



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





# Java Synchronization

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

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

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

 notify();
}
```



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

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

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

 notify();

 return item;
}
```





# Java Reentrant Locks

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

```
Lock key = new ReentrantLock();

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





# Java Semaphores

- Constructor:

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

- Usage:

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

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





# Java Condition Variables

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

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

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

## ■ Example:

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

## ■ Necessary data structures:

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

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





# Java Condition Variables

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

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

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

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





# Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages

- Consider a function update() that must be called atomically. One option is to use mutex locks:
- A memory transaction is a sequence of read-write operations to memory that are performed atomically. A transaction can be completed by adding atomic{S} which ensure statements in S are executed atomically:

```
void update ()
{
 acquire();

 /* modify shared data */

 release();
}
```

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





# OpenMP

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

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

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





# Functional Programming Languages

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

# Summary

- Classic problems of process synchronization include the bounded-buffer, readers-writers, and dining-philosophers problems. Solutions to these problems can be developed using the tools presented in Chapter 6, including mutex locks, semaphores, monitors, and condition variables.
- Windows uses dispatcher objects as well as events to implement process synchronization tools.
- Linux uses a variety of approaches to protect against race conditions, including atomic variables, spinlocks, and mutex locks.
- The POSIX API provides mutex locks, semaphores, and condition variables. POSIX provides two forms of semaphores: named and unnamed. Several unrelated processes can easily access the same named semaphore by simply referring to its name. Unnamed semaphores cannot be shared as easily, and require placing the semaphore in a region of shared memory.





# Summary (Cont.)

- Java has a rich library and API for synchronization. Available tools include monitors (which are provided at the language level) as well as reentrant locks, semaphores, and condition variables (which are supported by the API).
- Alternative approaches to solving the critical-section problem include transactional memory, OpenMP, and functional languages. Functional languages are particularly intriguing, as they offer a different programming paradigm from procedural languages. Unlike procedural languages, functional languages do not maintain state and therefore are generally immune from race conditions and critical sections.

# End of Chapter 7



What Is an  
**OPERATING SYSTEM (OS)**  
and How Does It Work

CLEVERISM.COM

# Chapter 8: Deadlocks





# Chapter 8: Outline

---

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



# Chapter Objectives

---

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



# System Model

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



# Deadlock in Multithreaded Application

- Two mutex locks are created and initialized:

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;
```

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



# Deadlock in Multithreaded Application

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

 pthread_exit(0);
}

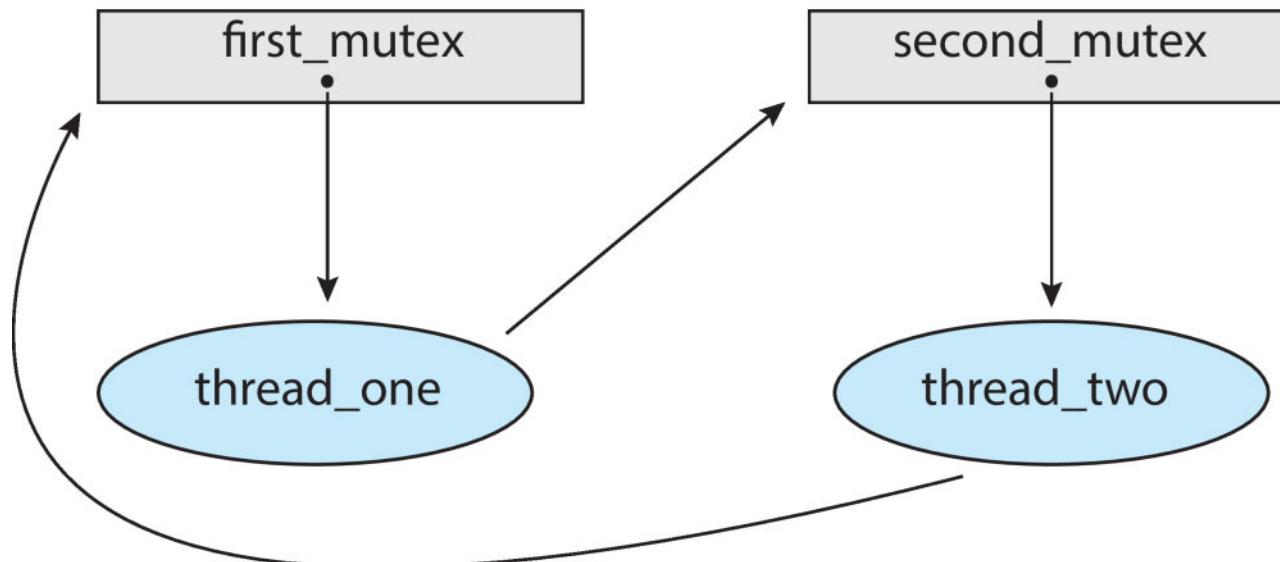
/* thread_two runs in this function */
void *do_work_two(void *param)
{
 pthread_mutex_lock(&second_mutex);
 pthread_mutex_lock(&first_mutex);
 /**
 * Do some work
 */
 pthread_mutex_unlock(&first_mutex);
 pthread_mutex_unlock(&second_mutex);

 pthread_exit(0);
}
```



# Deadlock in Multithreaded Application

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



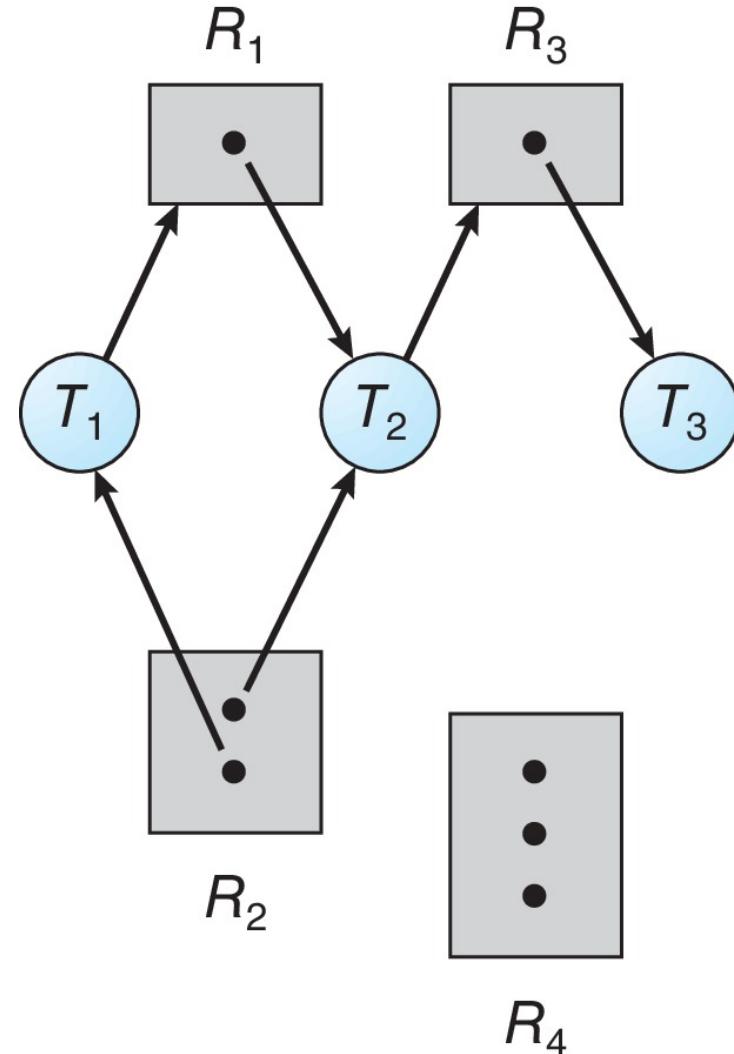
- Deadlock can arise if four conditions hold simultaneously.
  - **Mutual exclusion:** only one process at a time can use a resource
  - **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
  - **No preemption:** a resource can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

# Resource-Allocation Graph

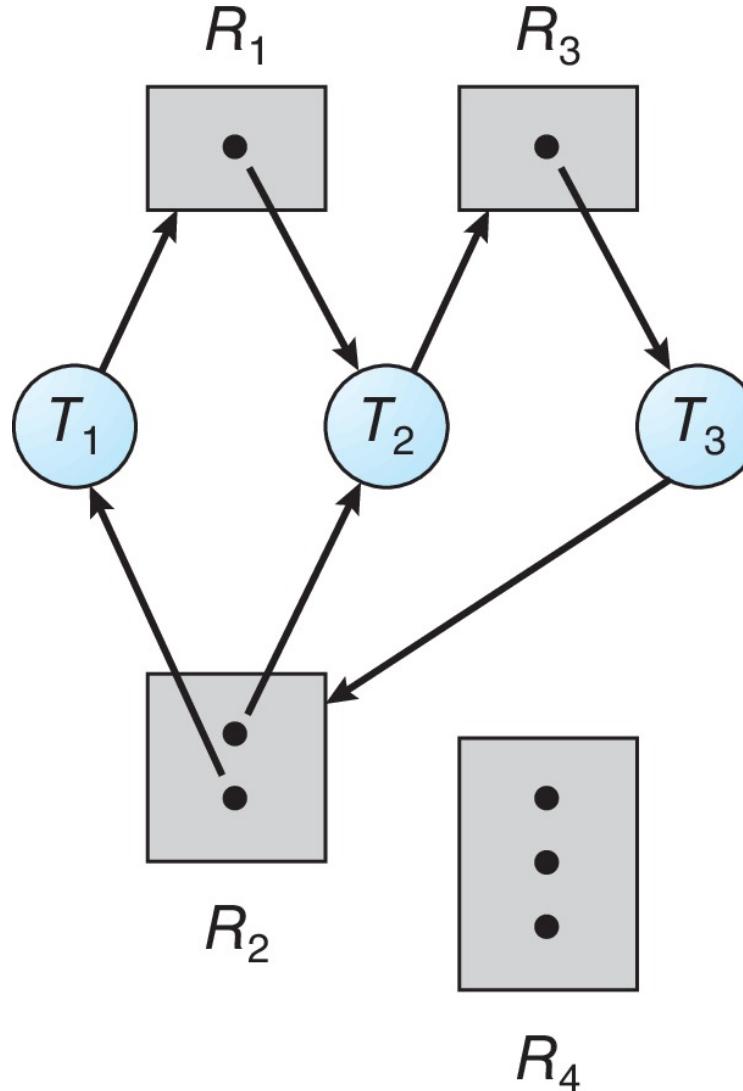
- A set of vertices  $V$  and a set of edges  $E$ .
- $V$  is partitioned into two types:
  - $P = \{P_1, P_2, \dots, P_n\}$ , the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system
- **request edge** – directed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

# Resource Allocation Graph Example

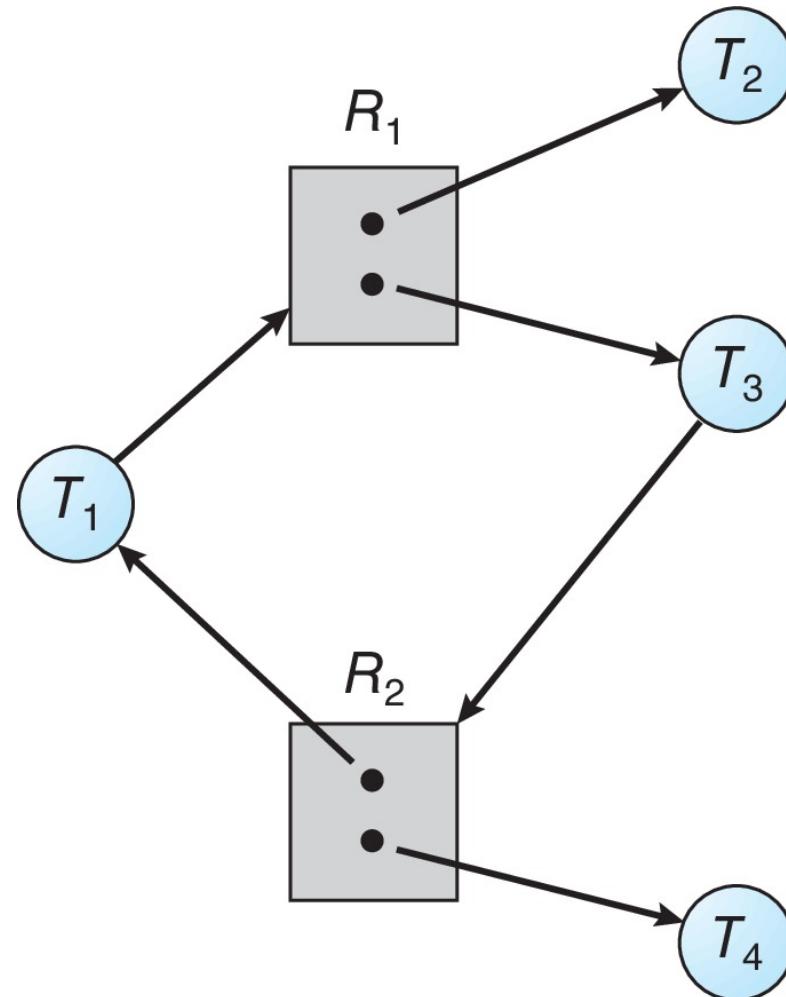
- One instance of R1
- Two instances of R2
- One instance of R3
- Three instances of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holds one instance of R3



# Resource Allocation Graph With A Deadlock



# Graph With A Cycle But No Deadlock





# Basic Facts

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



# Methods for Handling Deadlocks

- Ensure that the system will **never** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system.



# Deadlock Prevention

- Invalidate one of the four necessary conditions for deadlock:
  - **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
  - **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
    - ▶ Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
    - ▶ Low resource utilization; starvation possible



## ■ No Preemption

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
- Preempted resources are added to the list of resources for which the process is waiting
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

## ■ Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration



# Circular Wait

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

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

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

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

 pthread_exit(0);
}

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

 pthread_exit(0);
}
```



# Deadlock Avoidance

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

# Safe State

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



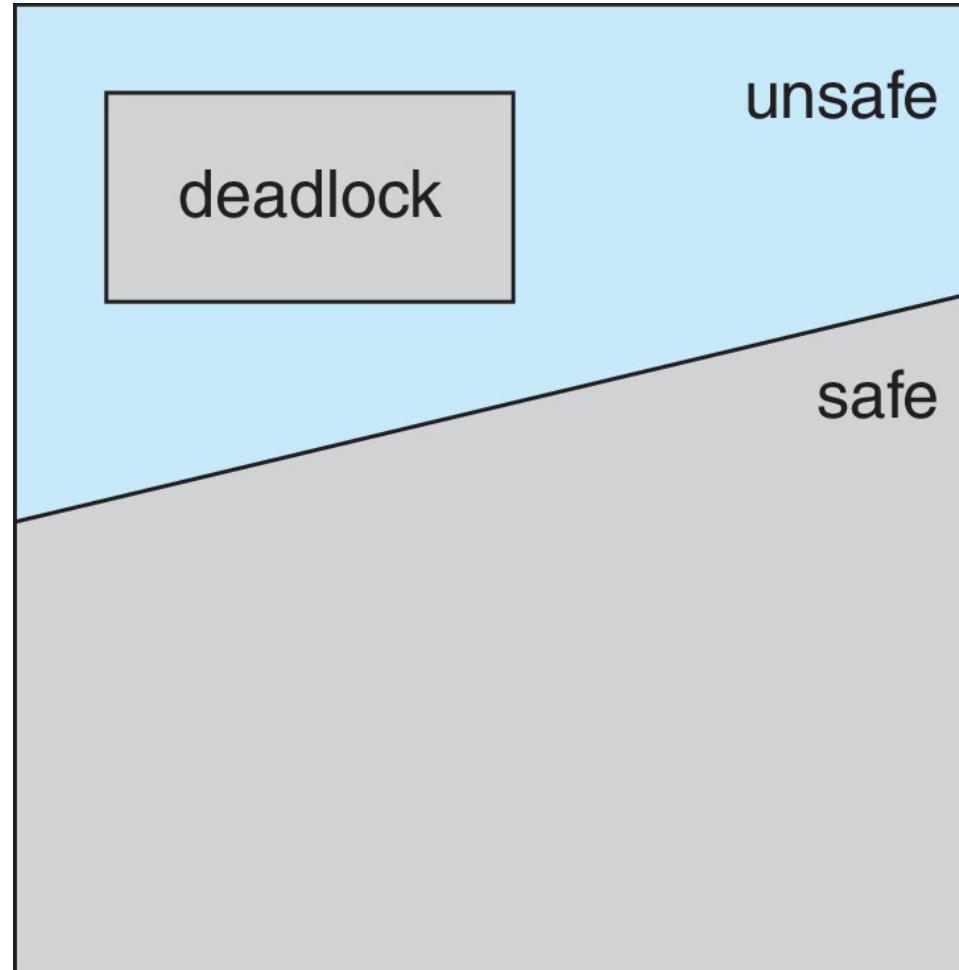


# Basic Facts

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



# Safe, Unsafe, Deadlock State





# Avoidance Algorithms

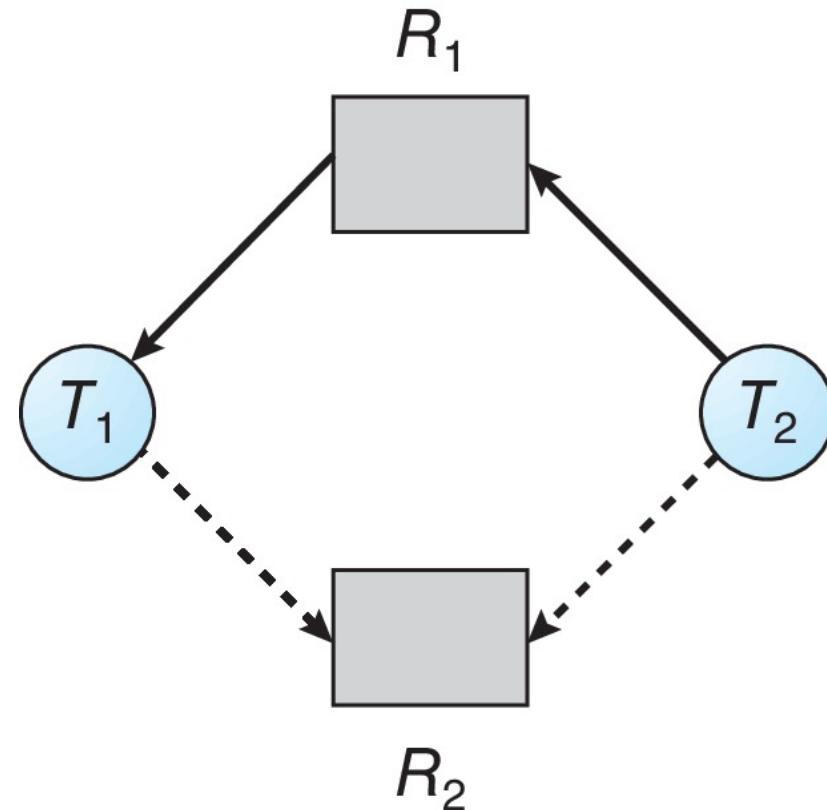
- Single instance of a resource type
  - Use a resource-allocation graph
  
- Multiple instances of a resource type
  - Use the Banker's Algorithm



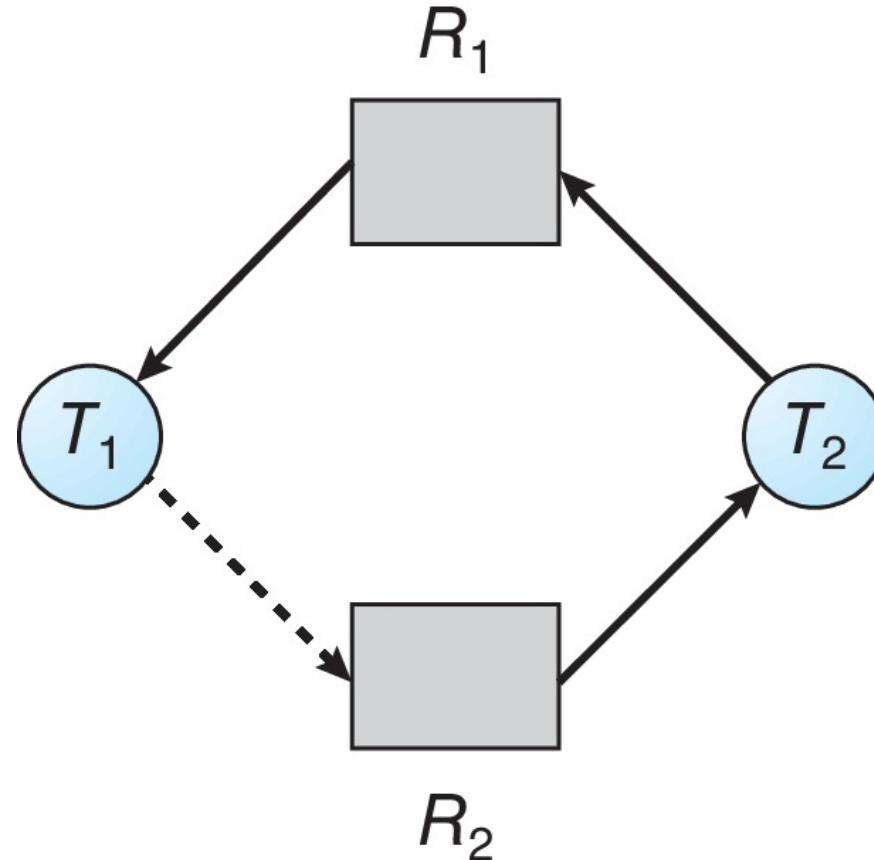
# Resource-Allocation Graph Scheme

- **Claim edge**  $P_i \rightarrow R_j$  indicated that process  $P_i$  may request resource  $R_j$ ; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

# Resource-Allocation Graph



# Unsafe State In Resource-Allocation Graph





# Resource-Allocation Graph Algorithm

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



# Banker's Algorithm

- Multiple instances of resources
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time



# Data Structures for the Banker's Algorithm

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

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$





# Safety Algorithm

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

**Work = Available**

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

2. Find an  $i$  such that both:

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

(b) **Need<sub>i</sub> ≤ Work**

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

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

**Finish[i] = true**

go to step 2

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





# Resource-Request Algorithm for Process $P_i$

$\text{Request}_i$  = request vector for process  $P_i$ . If  $\text{Request}_i[j] = k$  then process  $P_i$  wants  $k$  instances of resource type  $R_j$

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

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

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

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

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





# Example of Banker's Algorithm

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

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



# Example (Cont.)

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

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

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



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

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

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

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





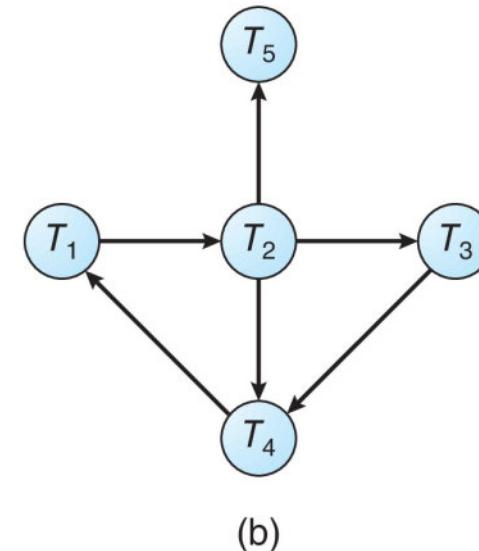
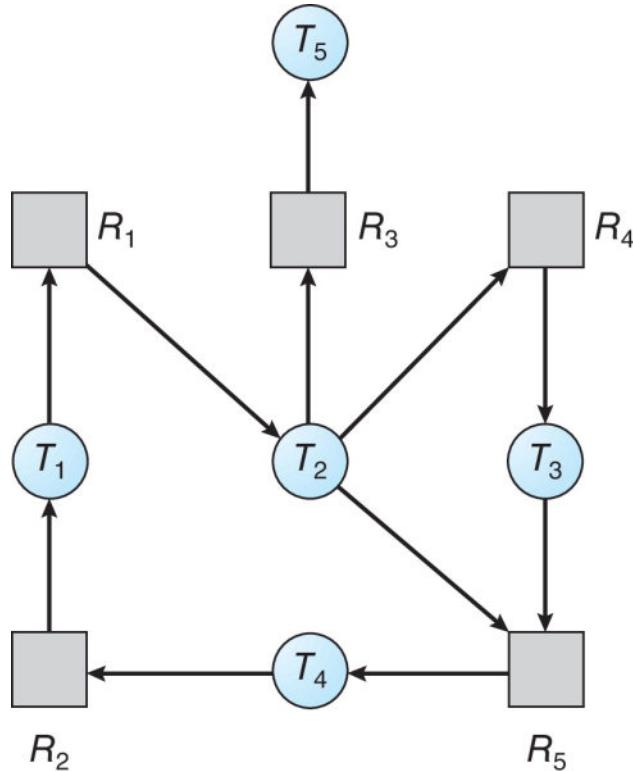
# Deadlock Detection

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

- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$  if  $P_i$  is waiting for  $P_j$
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of  $n^2$  operations, where  $n$  is the number of vertices in the graph



# Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph

Corresponding wait-for graph



# Several Instances of a Resource Type

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

1. Let **Work** and **Finish** be vectors of length  $m$  and  $n$ , respectively  
Initialize:
  - (a) **Work** = **Available**
  - (b) For  $i = 1, 2, \dots, n$ , if  $\text{Allocation}_i \neq 0$ , then  
**Finish**[ $i$ ] = **false**; otherwise, **Finish**[ $i$ ] = **true**
2. Find an index  $i$  such that both:
  - (a) **Finish**[ $i$ ] == **false**
  - (b) **Request** $_i \leq \text{Work}$

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



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

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

# Example of Detection Algorithm

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

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

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



# Example (Cont.)

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

Request

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

- State of system?

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



# Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - ▶ one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.





## Recovery from Deadlock: Process Termination

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





## Recovery from Deadlock: Resource Preemption

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

# Summary

- Deadlock occurs in a set of processes when every process in the set is waiting for an event that can only be caused by another process in the set.
- There are four necessary conditions for deadlock: (1) mutual exclusion, (2) hold and wait, (3) no preemption, and (4) circular wait. Deadlock is only possible when all four conditions are present.
- Deadlocks can be modeled with resource-allocation graphs, where a cycle indicates deadlock.
- Deadlocks can be prevented by ensuring that one of the four necessary conditions for deadlock cannot occur. Of the four necessary conditions, eliminating the circular wait is the only practical approach.





# Summary (Cont.)

- Deadlock can be avoided by using the banker's algorithm, which does not grant resources if doing so would lead the system into an unsafe state where deadlock would be possible.
- A deadlock-detection algorithm can evaluate processes and resources on a running system to determine if a set of processes is in a deadlocked state.
- If deadlock does occur, a system can attempt to recover from the deadlock by either aborting one of the processes in the circular wait or preempting resources that have been assigned to a deadlocked process.



# End of Chapter 8



What Is an  
**OPERATING SYSTEM (OS)**  
and How Does It Work

CLEVERISM.COM

# Chapter 9: Main Memory



What Is an  
**OPERATING SYSTEM (OS)**  
and How Does It Work

CLEVERISM.COM



# Chapter 9: Outline

---

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



# Objectives

---

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



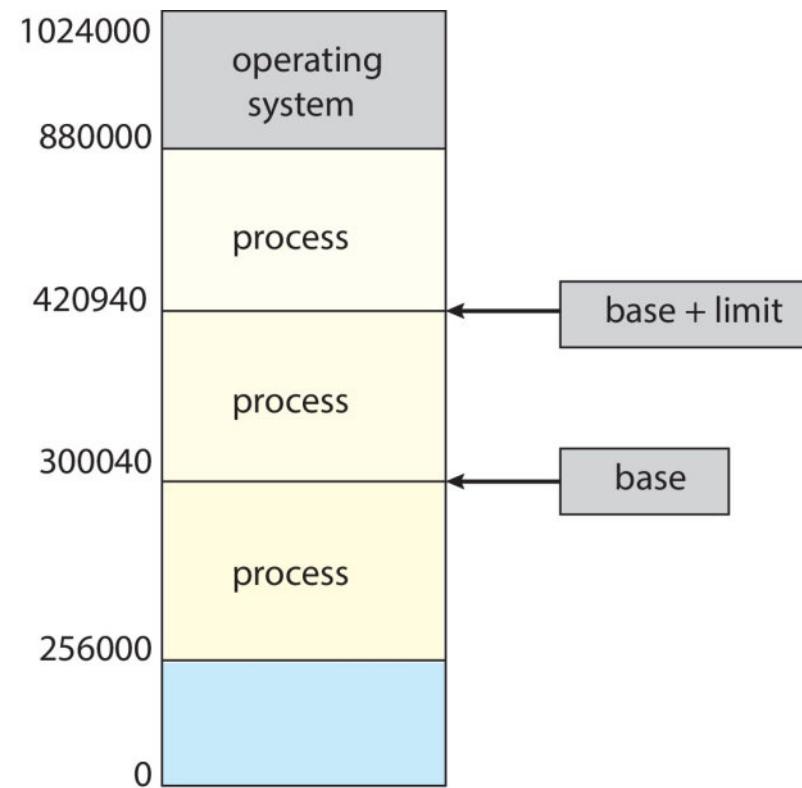
# Background

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



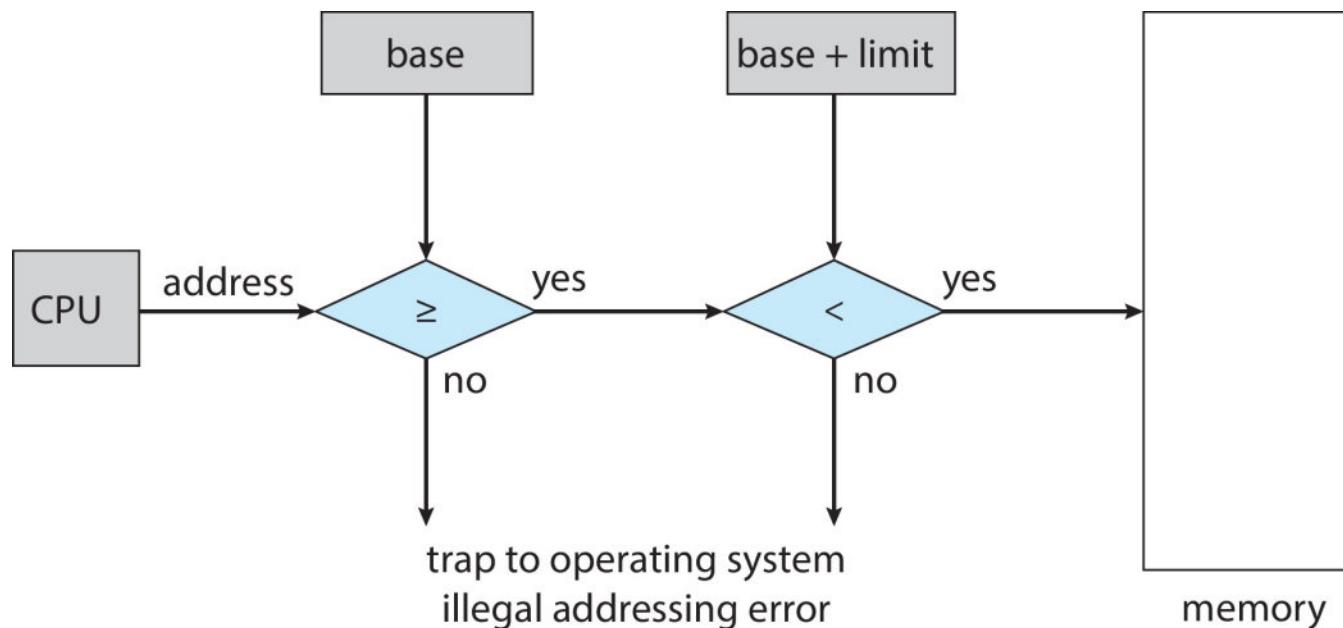
# Protection

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



# Hardware Address Protection

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



- Instructions to loading the base and limit registers are *privileged*

# Address Binding

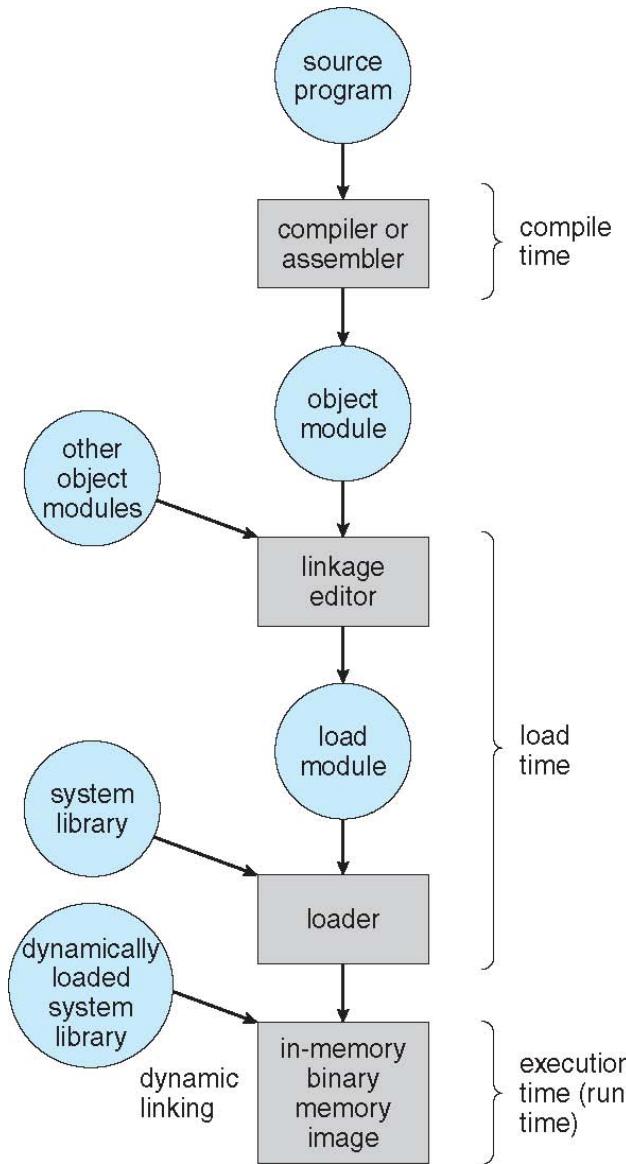
- Programs on disk, ready to be brought into memory to execute, form an *input queue*
  - Without support, it must be loaded into address *0000*
- Inconvenient to have first user process physical address always at *0000*
  - How can it not be?
- Addresses represented in different ways at different stages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



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



# Multistep Processing of a User Program



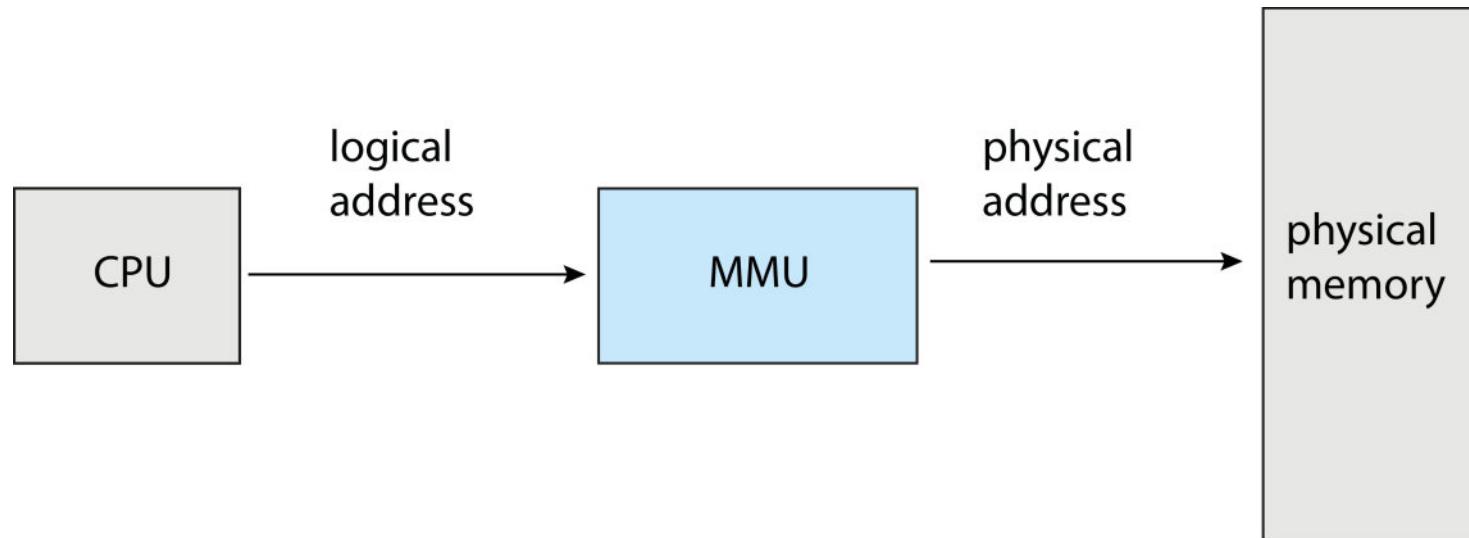
# Logical vs. Physical Address Space

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



# Memory-Management Unit (MMU)

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

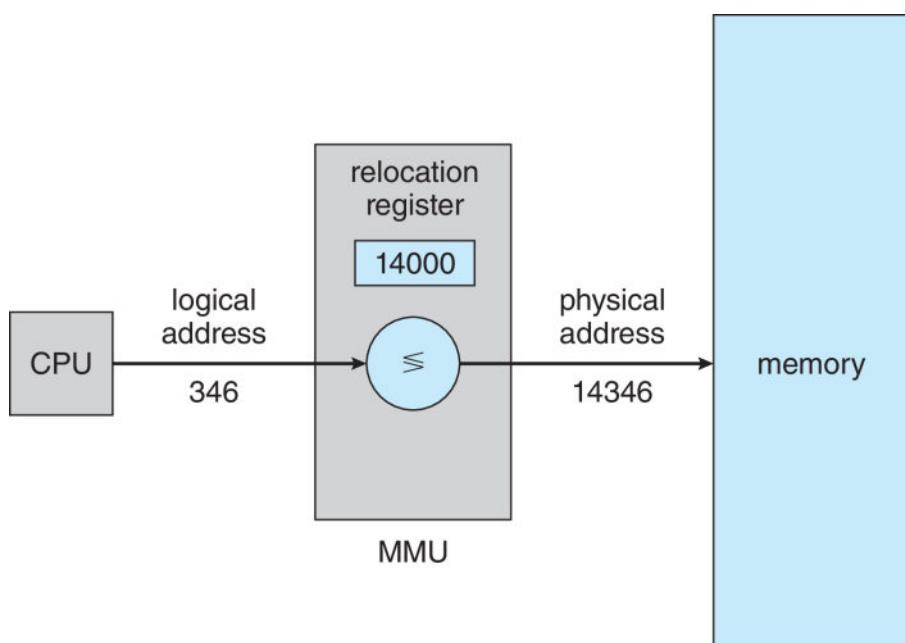


- Many methods possible, covered in the rest of this chapter



# Memory-Management Unit (Cont.)

- E.g., Consider simple scheme which is a generalization of the *base-register scheme*
  - The base register now called *relocation register*
  - The value in the relocation register is added to every address generated by a user process at the time it is sent to memory



- *The user program deals with logical addresses*; it never sees the real physical addresses
- *Execution-time binding* (i.e., logical addresses bound to physical addresses) occurs when reference is made to location in memory



# Dynamic Loading

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





# Dynamic Loading (Cont.)

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

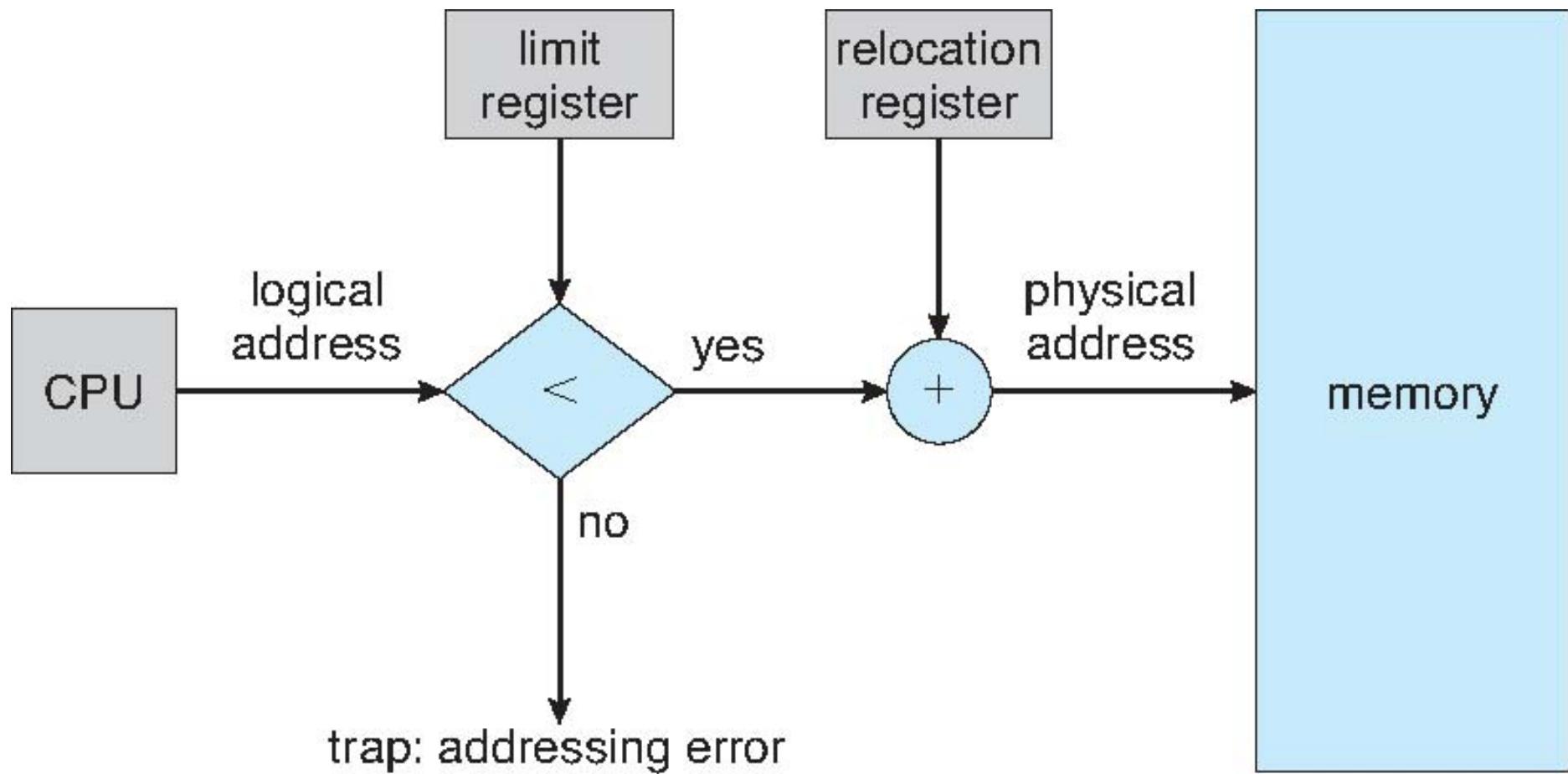


# Contiguous Allocation

- *Main memory* must support both OS and user processes
  - Limited resource, must allocate efficiently
- *Contiguous allocation* is one early method
  - Main memory usually divides into *two partitions*:
    - ▶ *Resident operating system*, usually held in *high memory* with interrupt vector
    - ▶ *User processes* then held in *low memory*
      - Each process contained in *single contiguous section of memory*
- *Relocation registers* used to protect user processes from each other, and from changing operating-system code and data
  - *Base register* contains value of smallest physical address
  - *Limit register* contains range of logical addresses
- **MMU** maps logical address *dynamically*
  - Can then allow actions such as kernel code being *transient* and kernel changing size

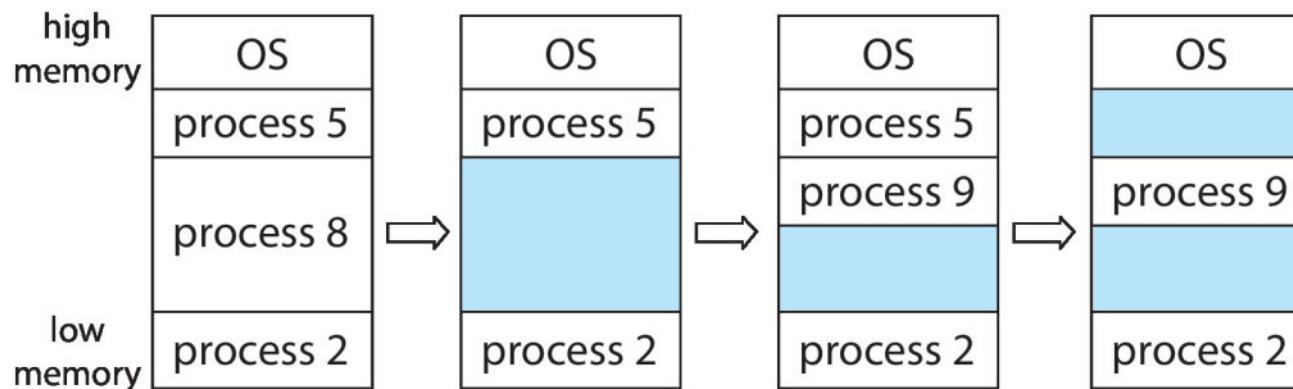


# Hardware Support for Relocation and Limit Registers



## ■ *Multiple-partition* allocation

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



- How to satisfy a memory 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
    - ▶ It must search entire list, unless ordered by size
    - ▶ Produces the smallest leftover hole
  - *Worst-fit*: Allocate the *largest* hole
    - ▶ It must also search entire list
    - ▶ Produces the largest leftover hole
- *First-fit* and *best-fit* strategies are better than *worst-fit* in terms of speed and memory utilization



# Fragmentation

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



# Fragmentation (Cont.)

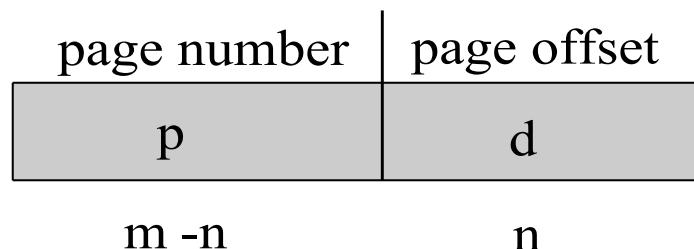
- Reduce external fragmentation by *compaction*
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - ▶ Latch job in memory while it is involved in I/O
    - ▶ Do I/O only into OS buffers
- Now consider that *backing store* has same fragmentation problems



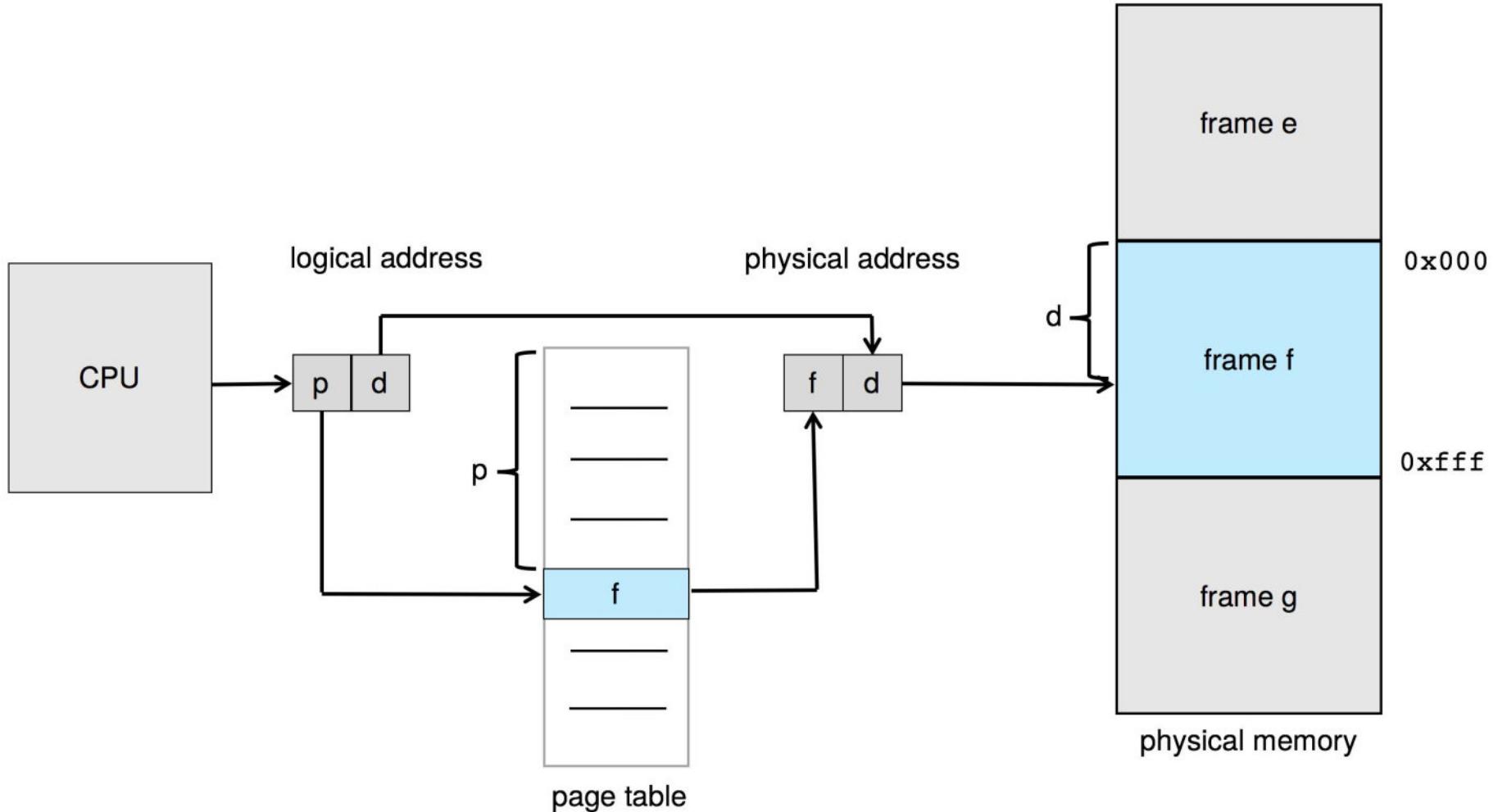
- *Physical address space of a process can be noncontiguous*; process is allocated physical memory whenever the latter is available
  - Avoids *external fragmentation*
  - Avoids problem of *varying sized memory chunks*
- Divide physical memory into fixed-sized blocks called *frames*
  - Size is *power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



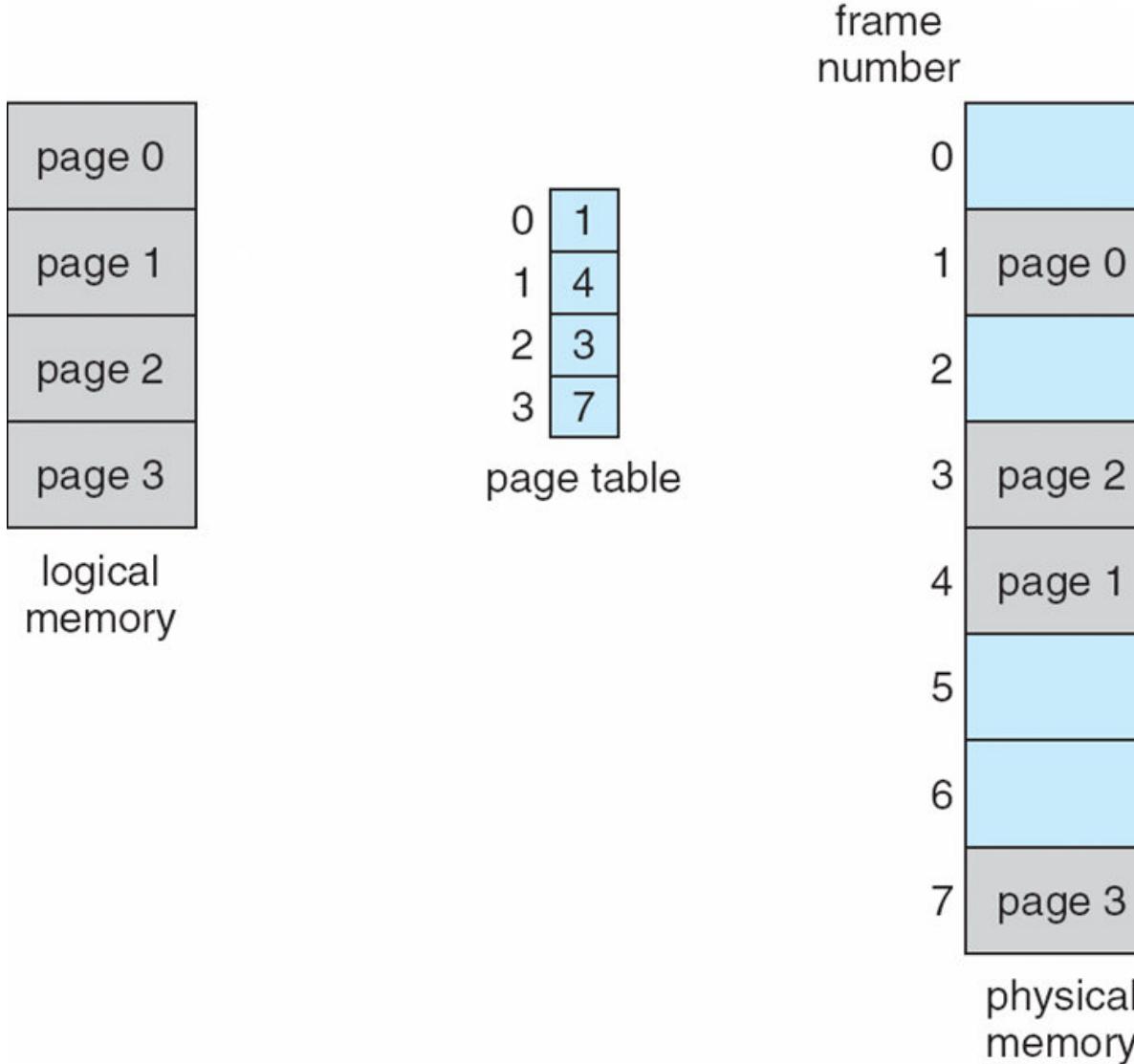
- Address generated by CPU is divided into:
  - *Page number (p)* – used as an index into a page table which contains base address of each page in physical memory
  - *Page offset (d)* – combined with base address to define the physical memory address that is sent to the memory unit
- For given logical address space  $2^m$  and page size  $2^n$



# Paging Hardware



# Paging Model of Logical and Physical Memory



# Example of Paging

|    |   |
|----|---|
| 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 | 5 |
| 1 | 6 |
| 2 | 1 |
| 3 | 2 |

page table

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

physical memory

- Using a page size of 4 bytes, a logical memory of 16 bytes (4 pages) and a physical memory of 32 bytes (8 frames)

- Page size:  $2^n$

- $n = 2$

- Logical address space:  $2^m$

- $m = 4$

- Physical address space:  $2^r$

- $r = 5$



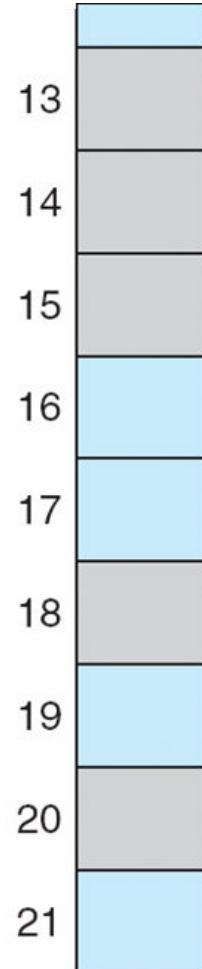
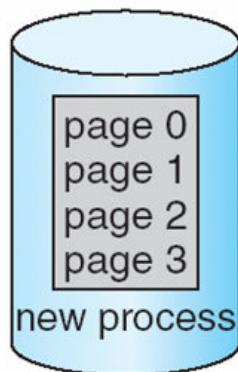
- Page size = *2,048* bytes
- Process size = *72,766* bytes
- *35* pages + *1,086* bytes
- Internal fragmentation of  $2,048 - 1,086 = 962$  bytes
- Worst case fragmentation = 1 frame – 1 byte
- On average fragmentation = 1 / 2 frame size
- So small frame sizes desirable?
  - But each page table entry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



# Free Frames

free-frame list

14  
13  
18  
20  
15

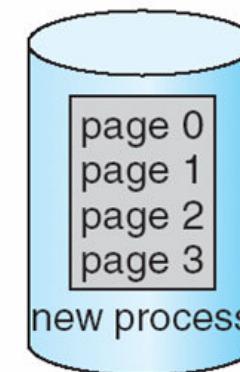


(a)

Before allocation

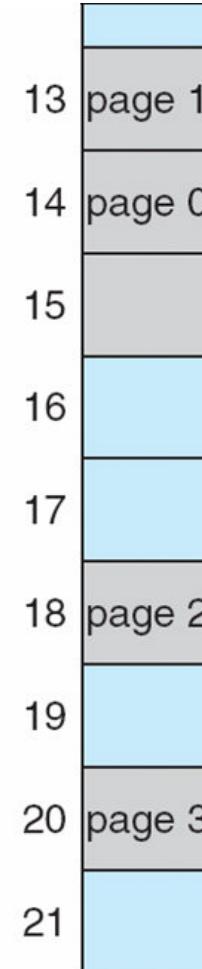
free-frame list

15



|   |    |
|---|----|
| 0 | 14 |
| 1 | 13 |
| 2 | 18 |
| 3 | 20 |

new-process page table



(b)

After allocation



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

# Translation Look-Aside Buffer (TLB)

- Some TLBs store *address-space identifiers* (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (**64** to **1,024** entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - *Replacement policies* must be considered
  - Some entries can be *wired down* for permanent fast access



## ■ *Associative memory – parallel search*

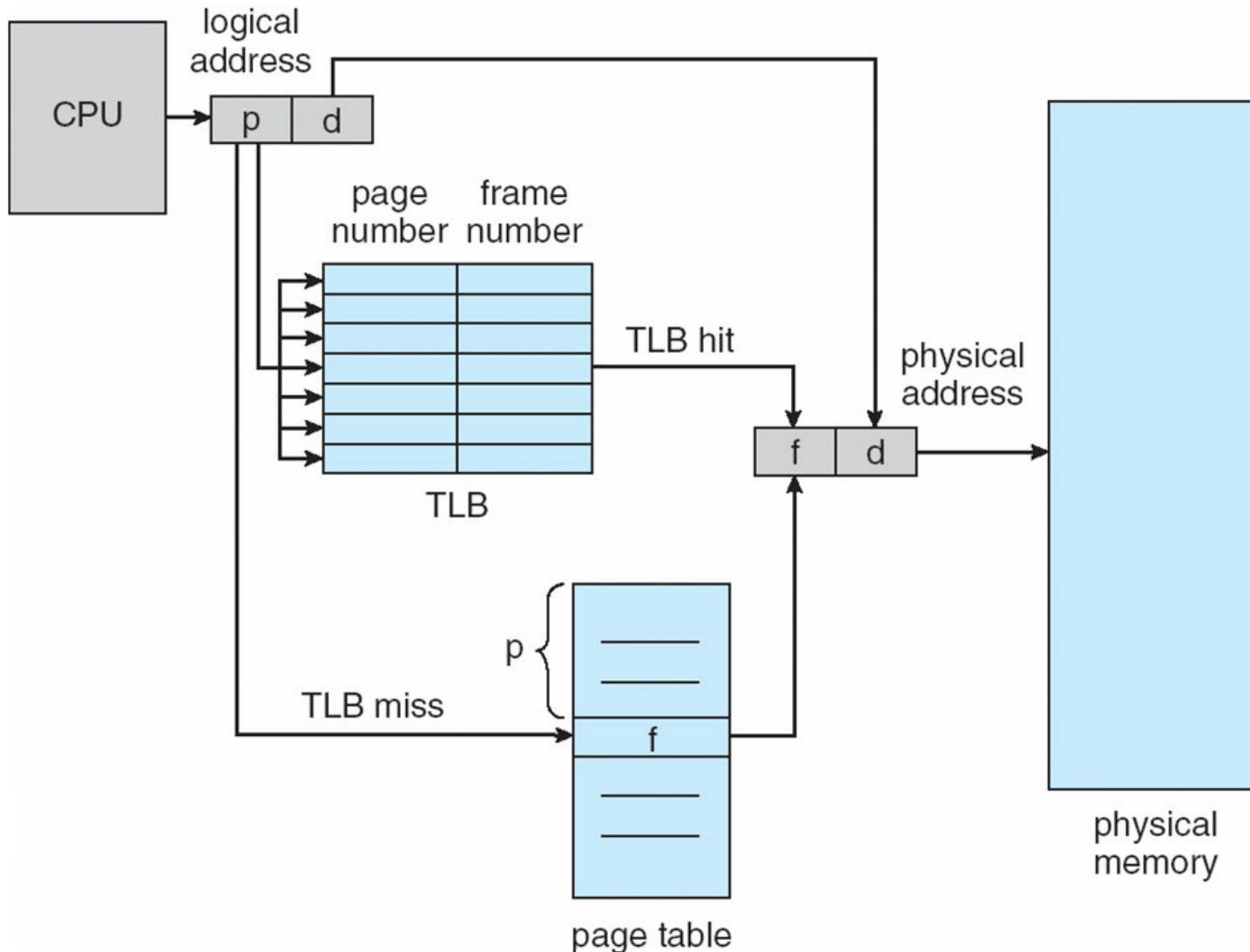
| Page # | Frame # |
|--------|---------|
|        |         |
|        |         |
|        |         |
|        |         |

## ■ *Address translation ( $p, d$ )*

- If  $p$  is in associative register, get frame # out
- Otherwise get frame # from page table in memory



# Paging Hardware With TLB



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

$$EAT = 0.80 \times 10 + 0.20 \times 20 = 12 \text{ nanoseconds}$$

implying 20% slowdown in access time

- Consider a more realistic hit ratio of 99%,

$$EAT = 0.99 \times 10 + 0.01 \times 20 = 10.1 \text{ ns}$$

implying only 1% slowdown in access time

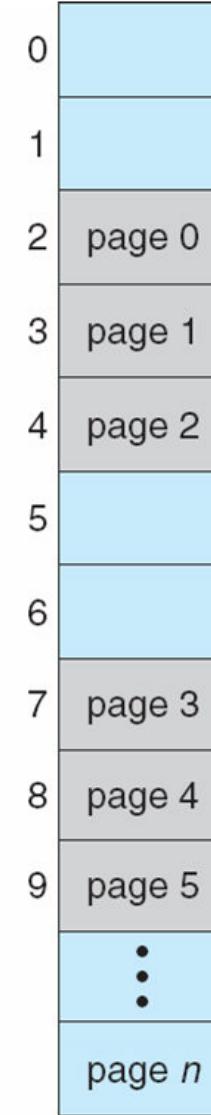
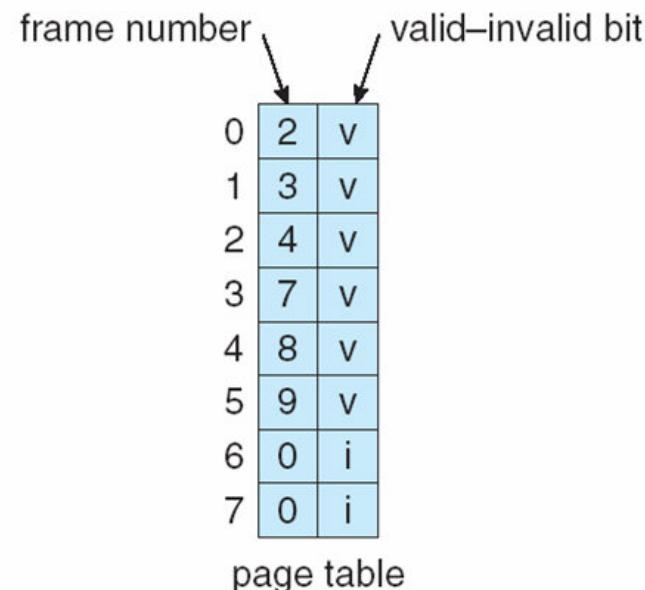


- Memory protection implemented by associating protection bit with each frame to indicate if *read-only* or *read-write* access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- *Valid-invalid* bit attached to each entry in the page table:
  - “*valid*” indicates that the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



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

|        |        |
|--------|--------|
| 00000  | page 0 |
|        | page 1 |
|        | page 2 |
|        | page 3 |
|        | page 4 |
| 10,468 | page 5 |
| 12,287 |        |





# Shared Pages

## ■ *Shared code*

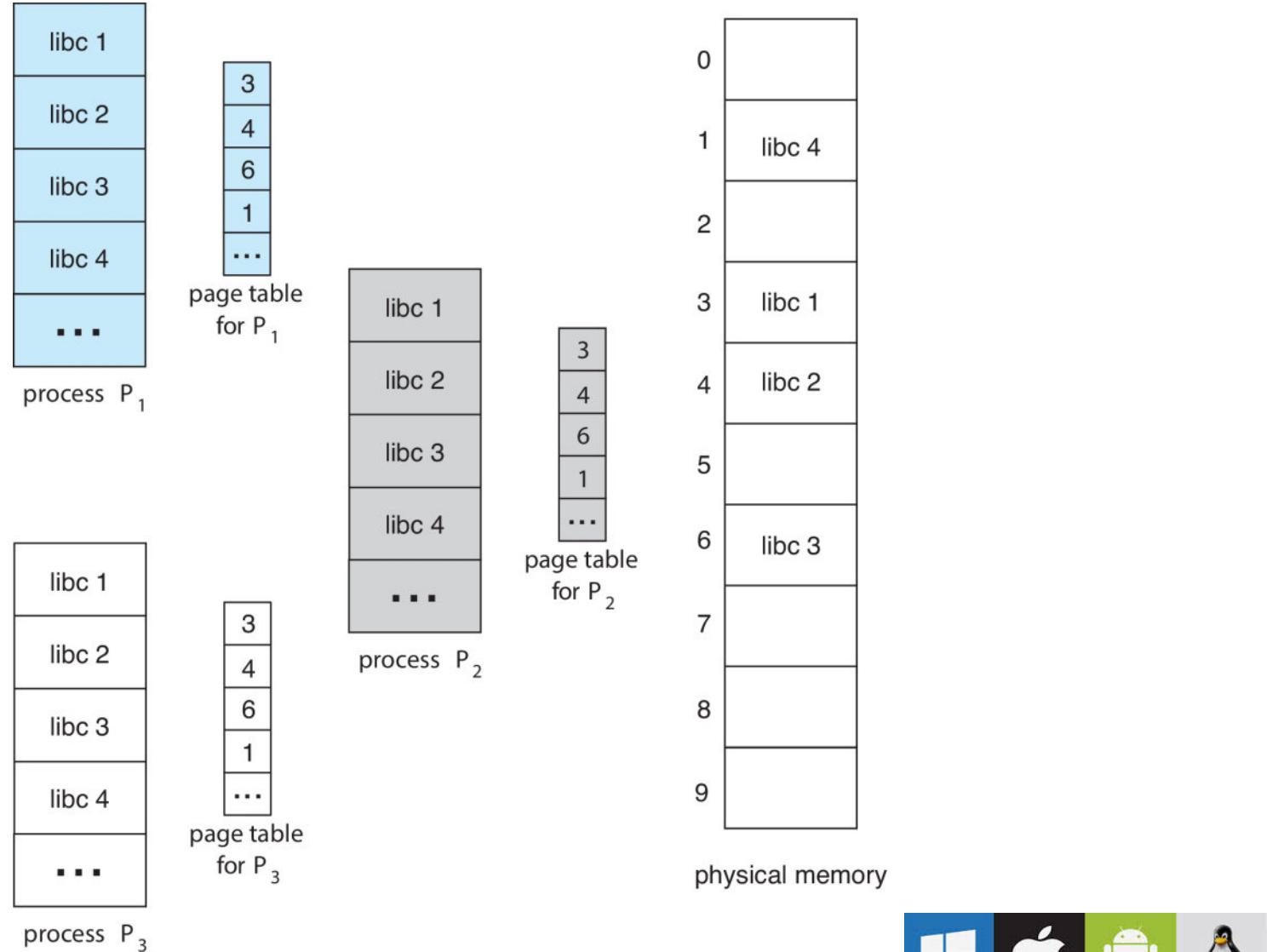
- One copy of read-only (*reentrant*) code shared among processes (i.e., text editors, compilers, window systems)
- Similar to *multiple threads* sharing the same process space
- Also useful for *inter-process communication* if sharing of read-write pages is allowed

## ■ *Private code and data*

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



# Shared Pages Example



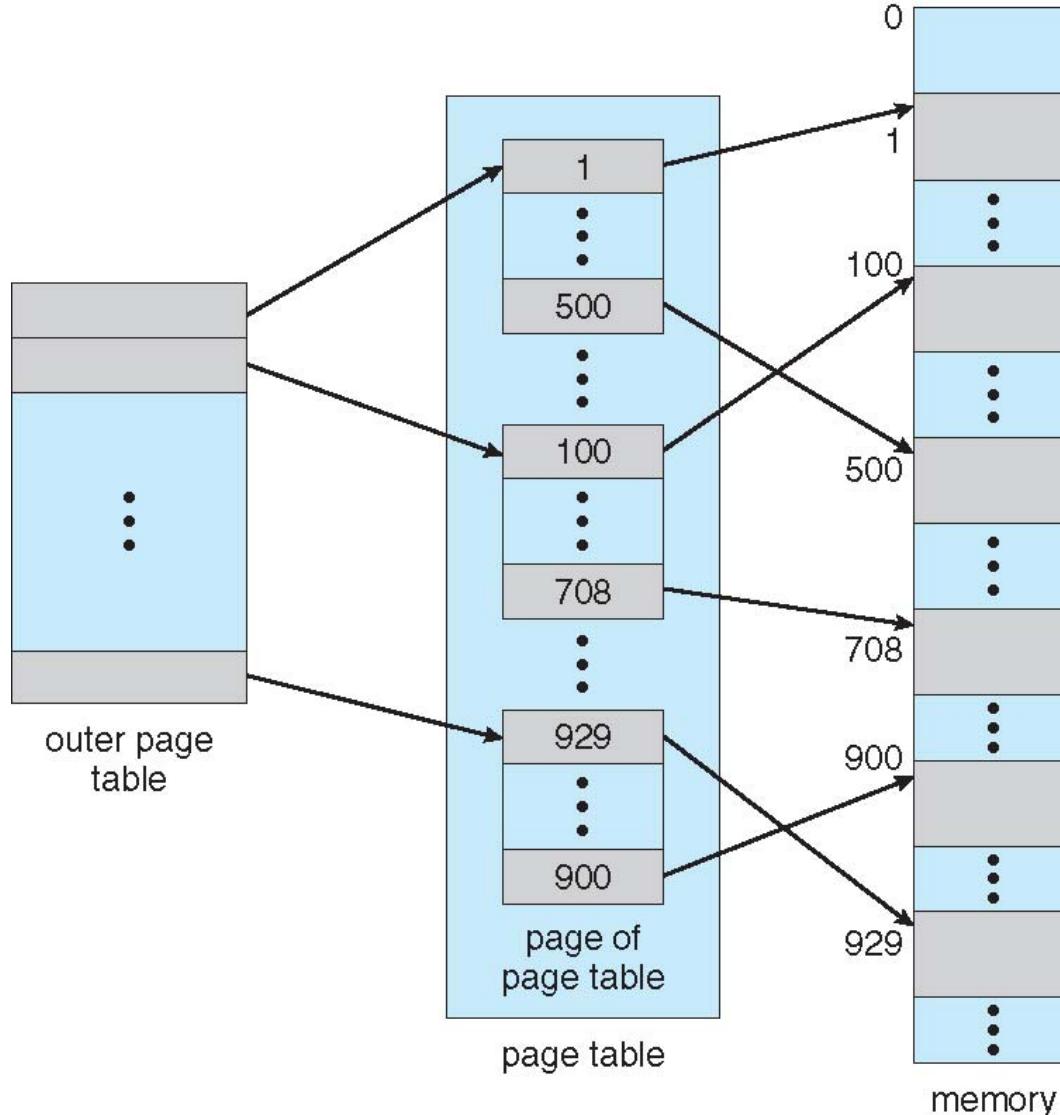
# Structure of the Page Table

- *Memory structures* for paging can get huge using straight-forward methods

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



# Hierarchical Page Tables



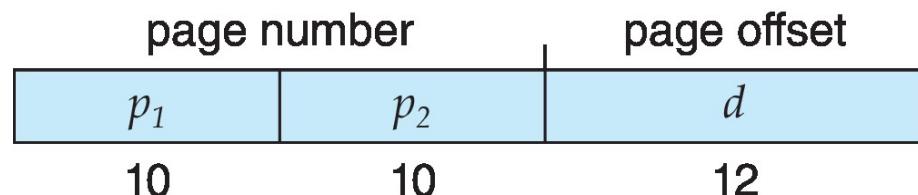
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

# Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - ▶ a page number consisting of 22 bits
  - ▶ a page offset consisting of 10 bits

- Since the page table is paged, the page number is further divided into:

- ▶ a 10-bit page number
- ▶ a 12-bit page offset



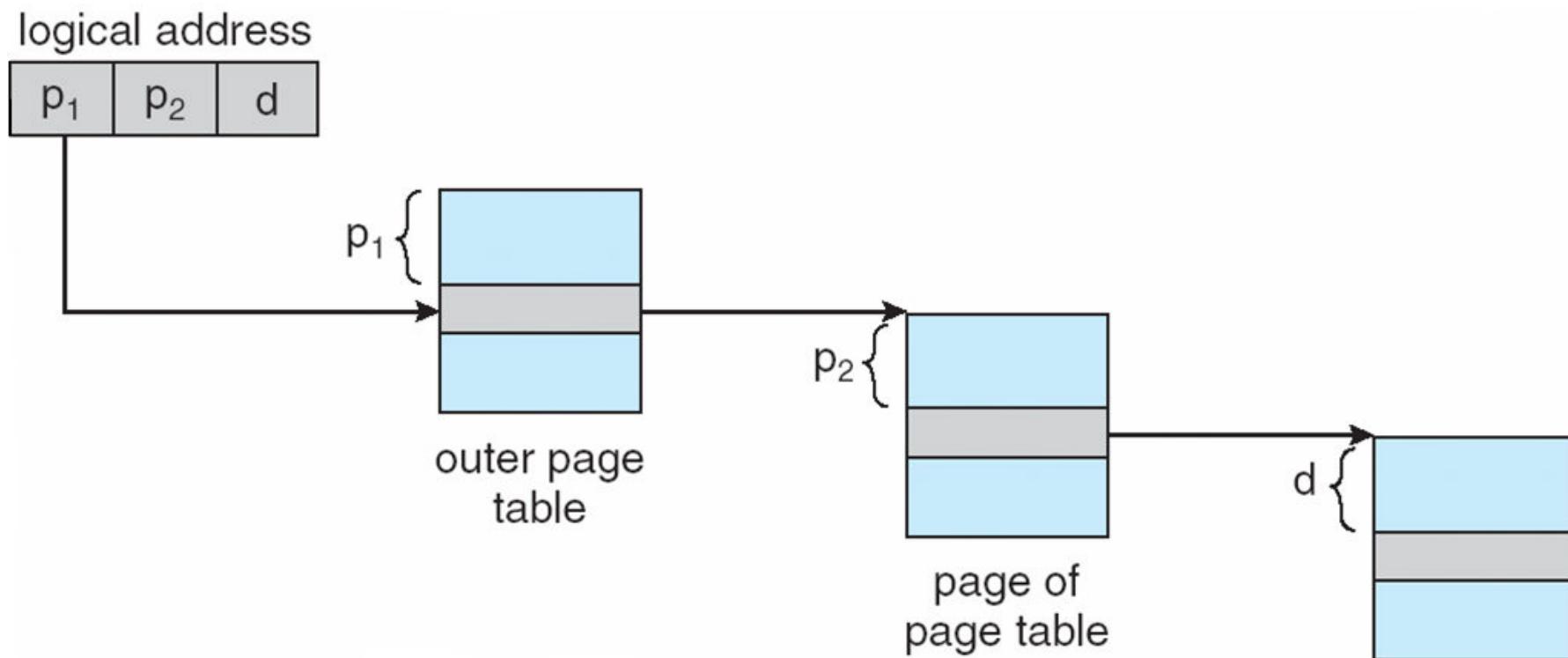
- Thus, a logical address is as follows:

where  $p_1$  is an index into the outer page table, and  $p_2$  is the displacement within the page of the inner page table

- Known as *forward-mapped page table*

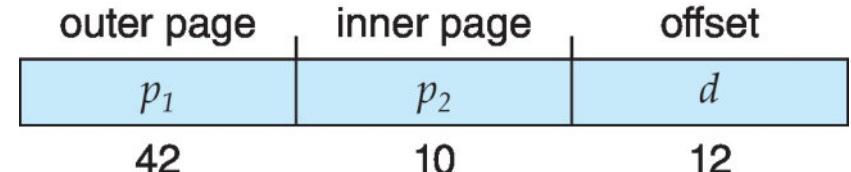


# Address-Translation Scheme

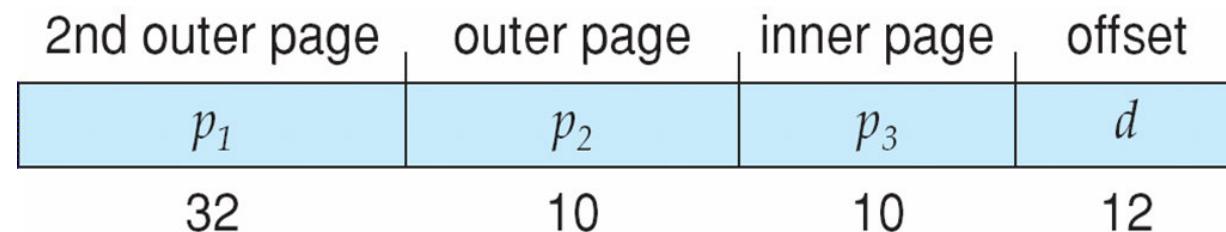
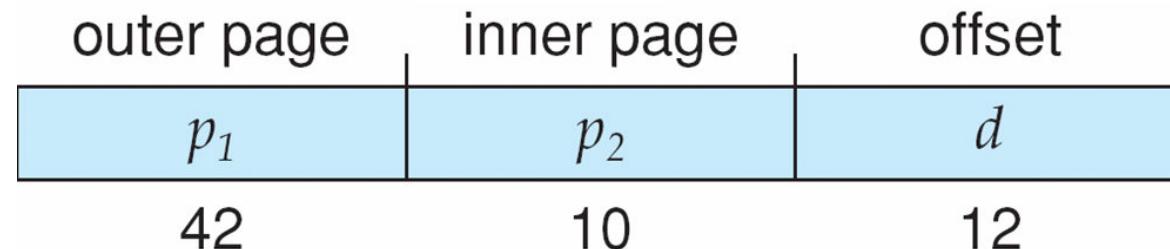


# 64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB ( $2^{12}$ )
  - Then page table has  $2^{52}$  entries
  - If two level scheme, inner page tables could be  $2^{10}$  4-byte entries
  - Address would look like
  - Outer page table has  $2^{42}$  entries or  $2^{44}$  bytes
  - One solution is to add a  $2^{nd}$  outer page table
  - But in the following example the  $2^{nd}$  outer page table is still  $2^{34}$  bytes in size
    - ▶ And possibly 4 memory access to get to one physical memory location



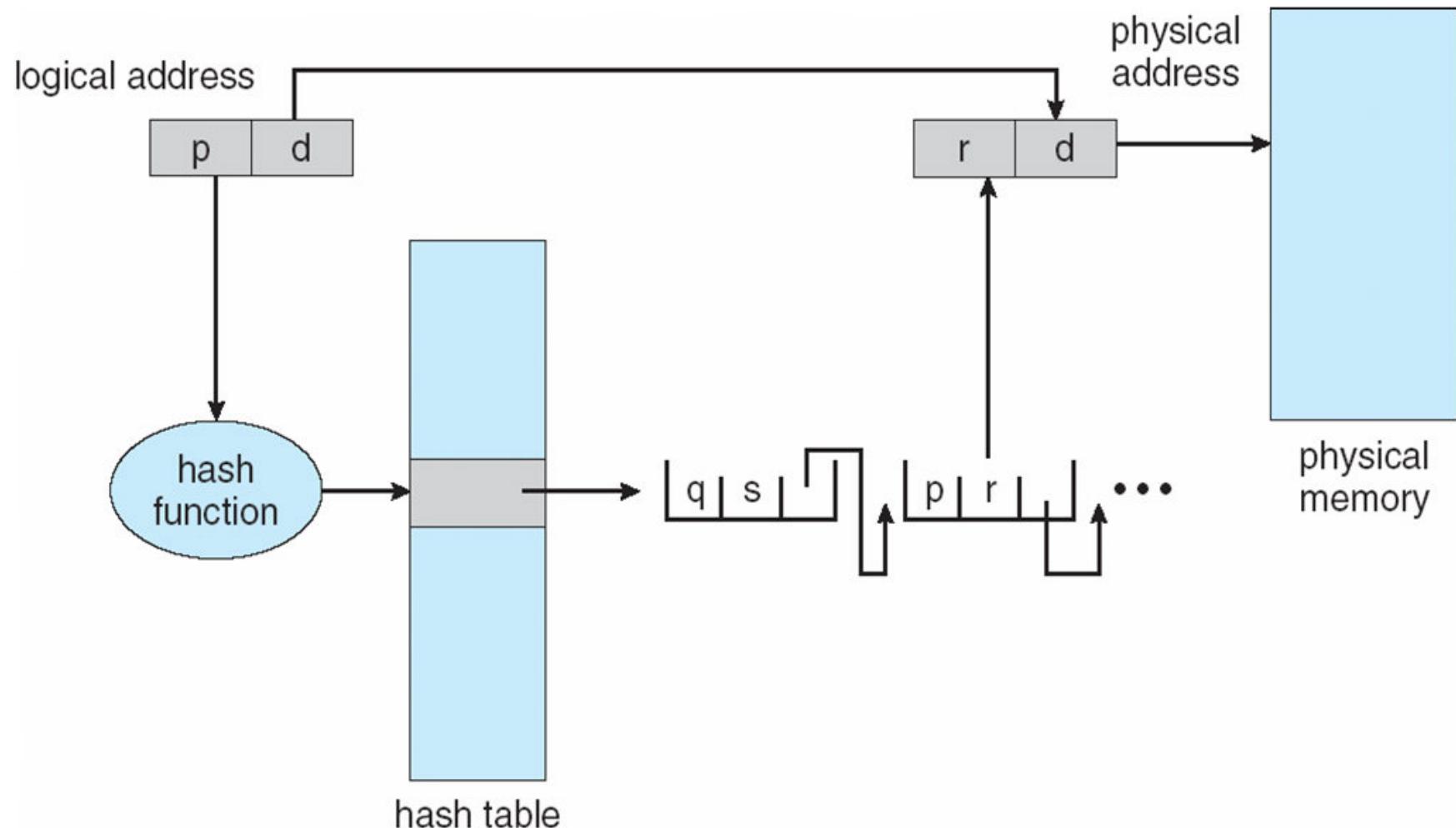
# Three-level Paging Scheme



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



# Hashed Page Table

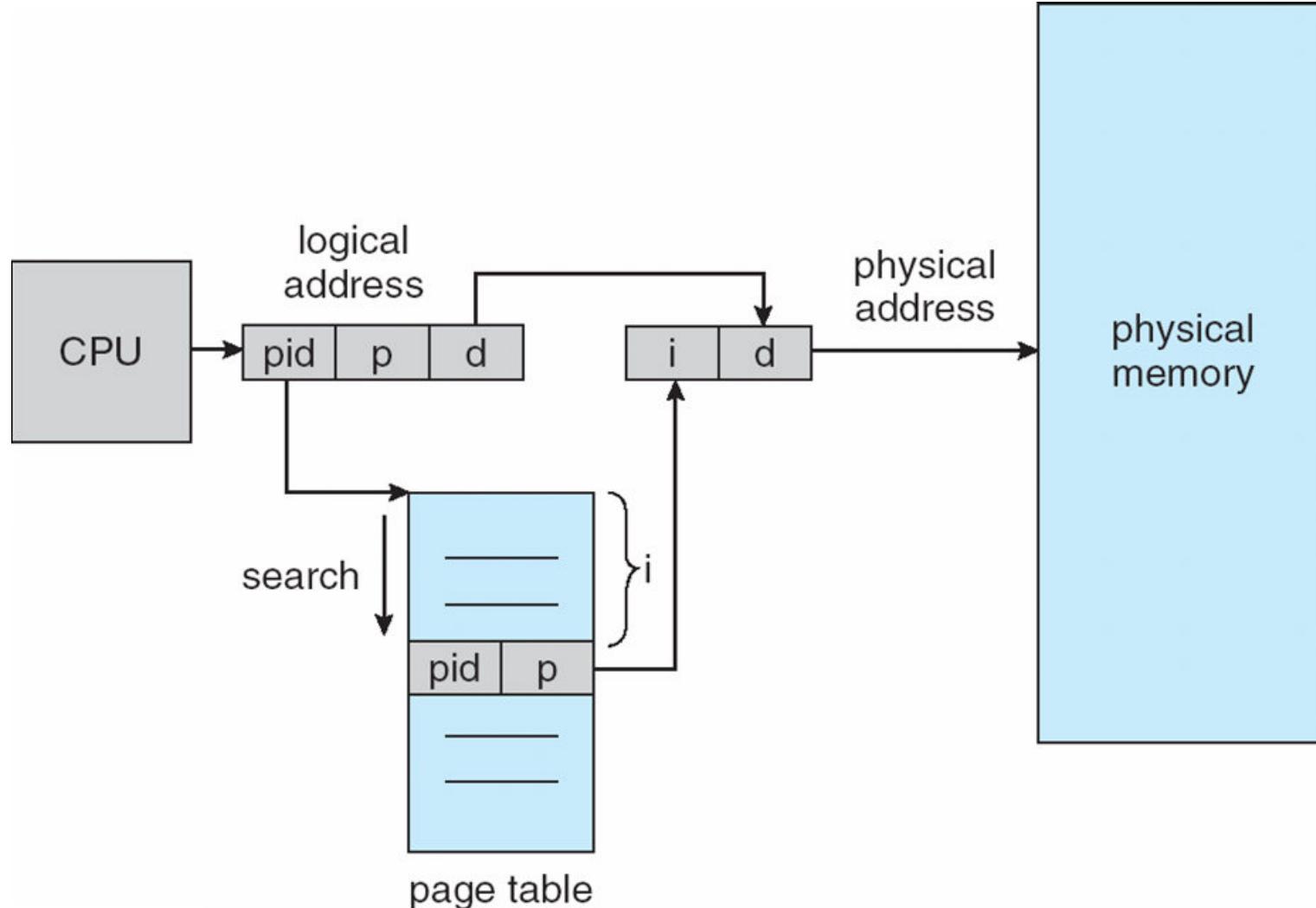


# Inverted Page Table

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



# Inverted Page Table Architecture



- 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 (Cont.)

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



# Swapping

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

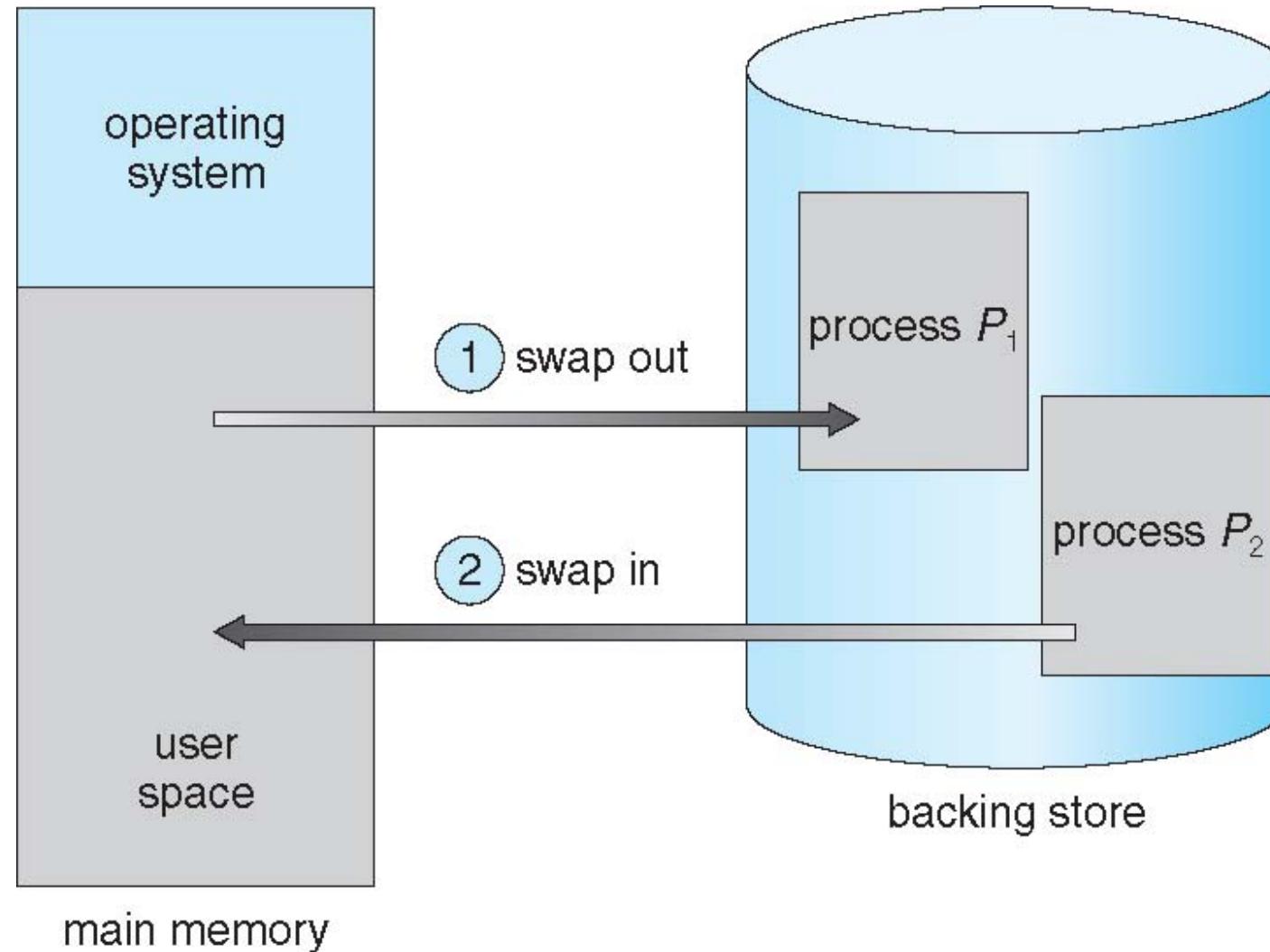


# Swapping (Cont.)

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



# Schematic View of Swapping



# Context Switch Time including Swapping

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



## ■ Other *constraints* as well on swapping

- Pending I/O – can't swap out as I/O would occur to wrong process
- Or always transfer I/O to kernel space, then to I/O device
  - ▶ Known as *double buffering*, adds overhead

## ■ Standard *swapping* not used in modern operating systems

- But modified version common
  - ▶ Swap only when free memory extremely low

# Swapping on Mobile Systems

## ■ Not typically supported

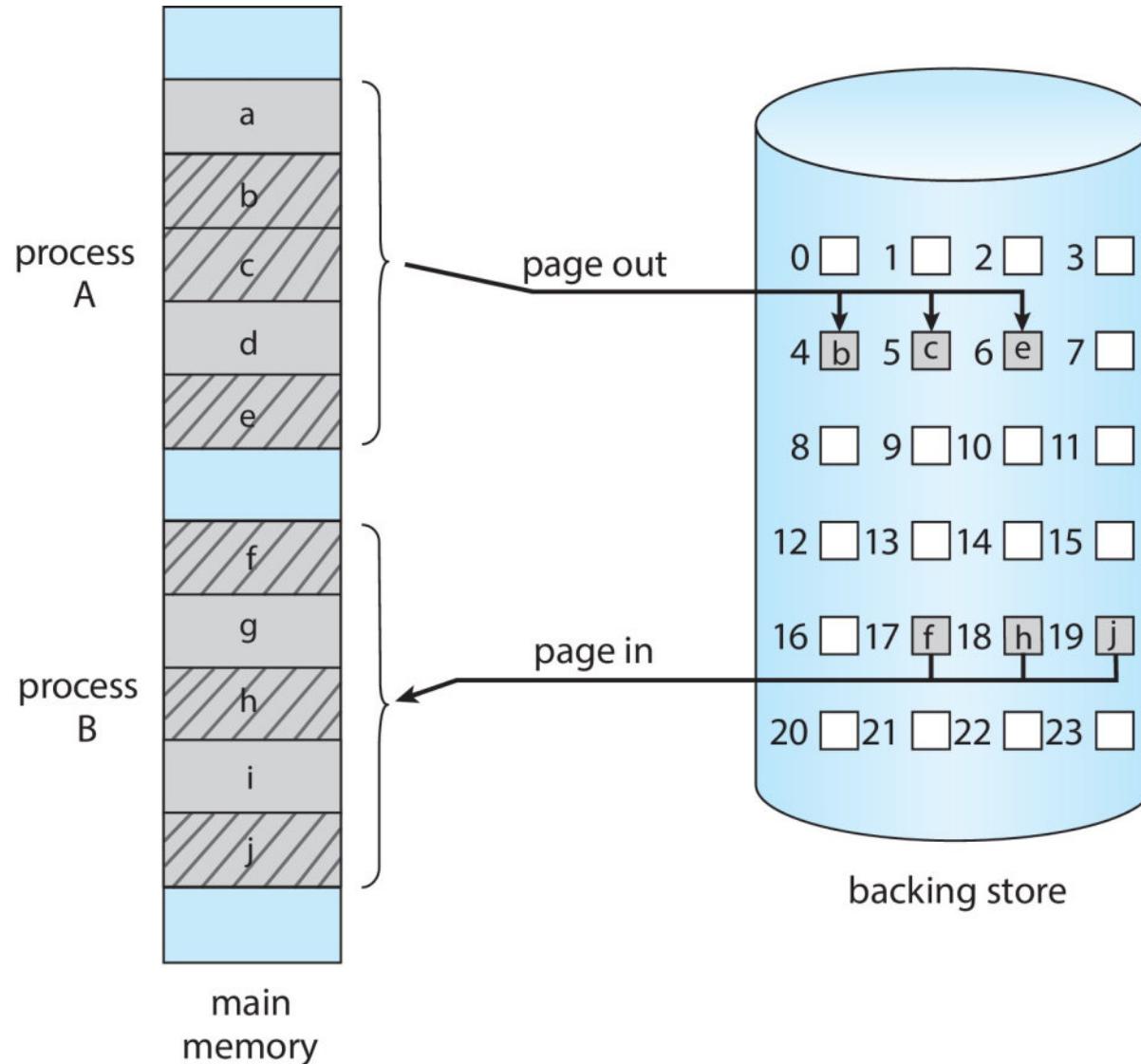
- Flash memory based
  - ▶ Small amount of space
  - ▶ Limited number of write cycles
  - ▶ Poor throughput between flash memory and CPU on mobile platform

## ■ Instead use other methods to free memory if low

- iOS asks apps to voluntarily relinquish allocated memory
  - ▶ Read-only data thrown out and reloaded from flash if needed
  - ▶ Failure to free can result in termination
- Android terminates apps if low free memory, but first writes application state to flash for fast restart
- Both OSes support paging as discussed below



# Swapping with Paging





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

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

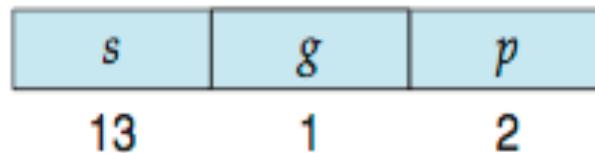


# Example: The Intel IA-32 Architecture

- Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB
  - Up to 16 K segments per process
  - Divided into two partitions
    - ▶ First partition of up to 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)*)

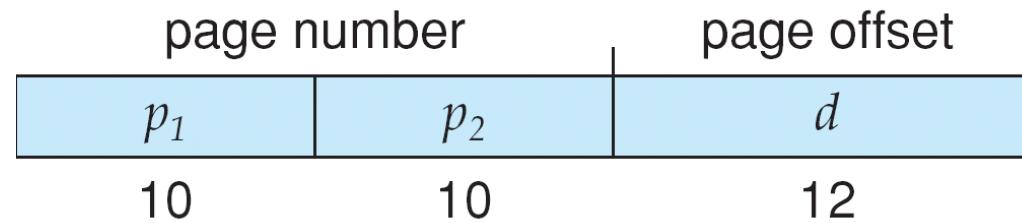
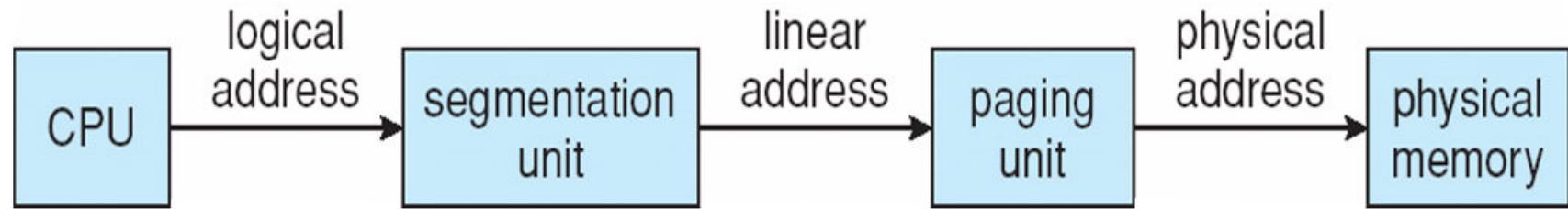
- CPU generates logical address

- Selector given to segmentation unit
    - ▶ Which produces linear addresses

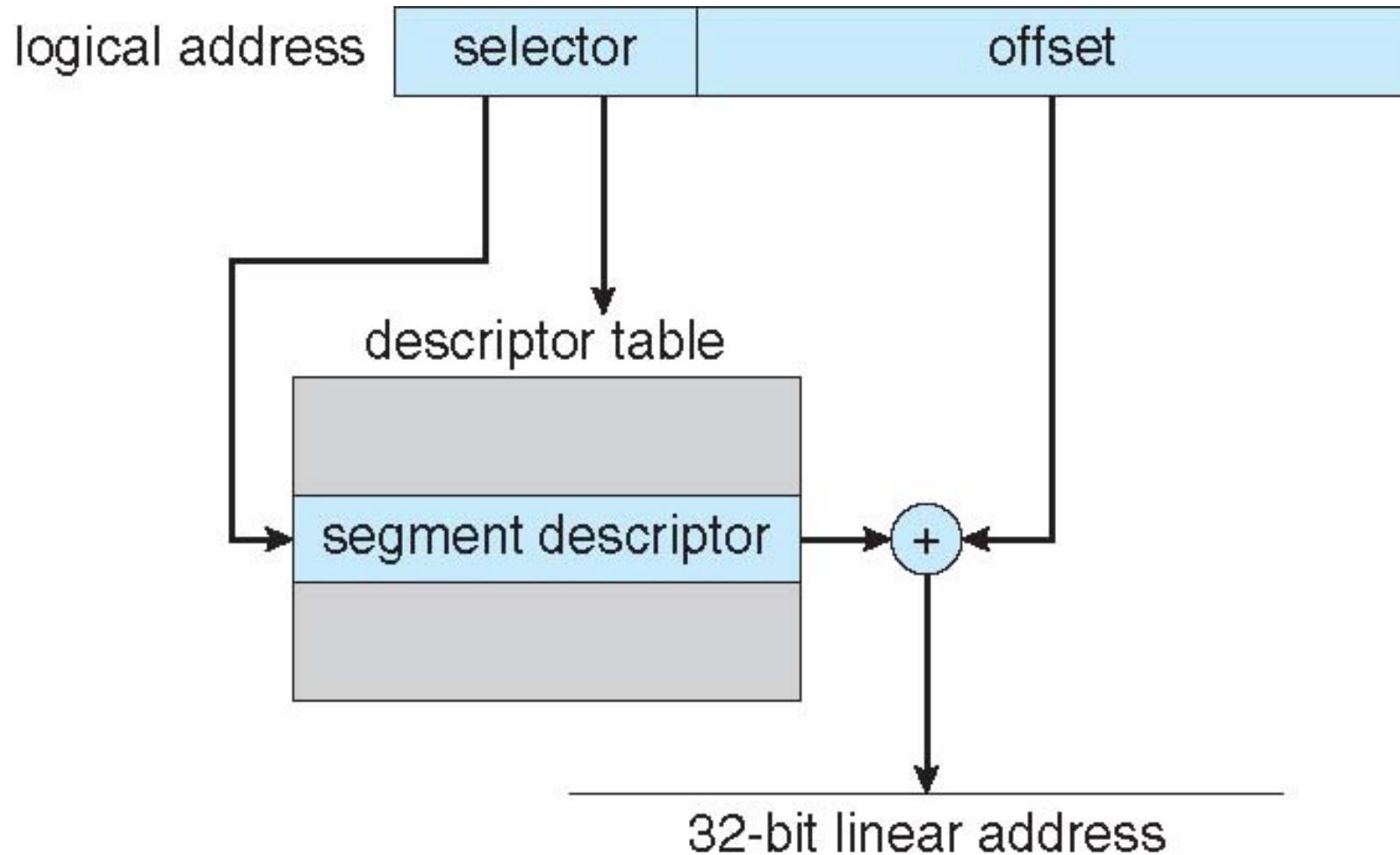


- Linear address given to paging unit
    - ▶ Which generates physical address in main memory
    - ▶ Paging units form equivalent of MMU
    - ▶ Pages sizes can be 4 KB or 4 MB

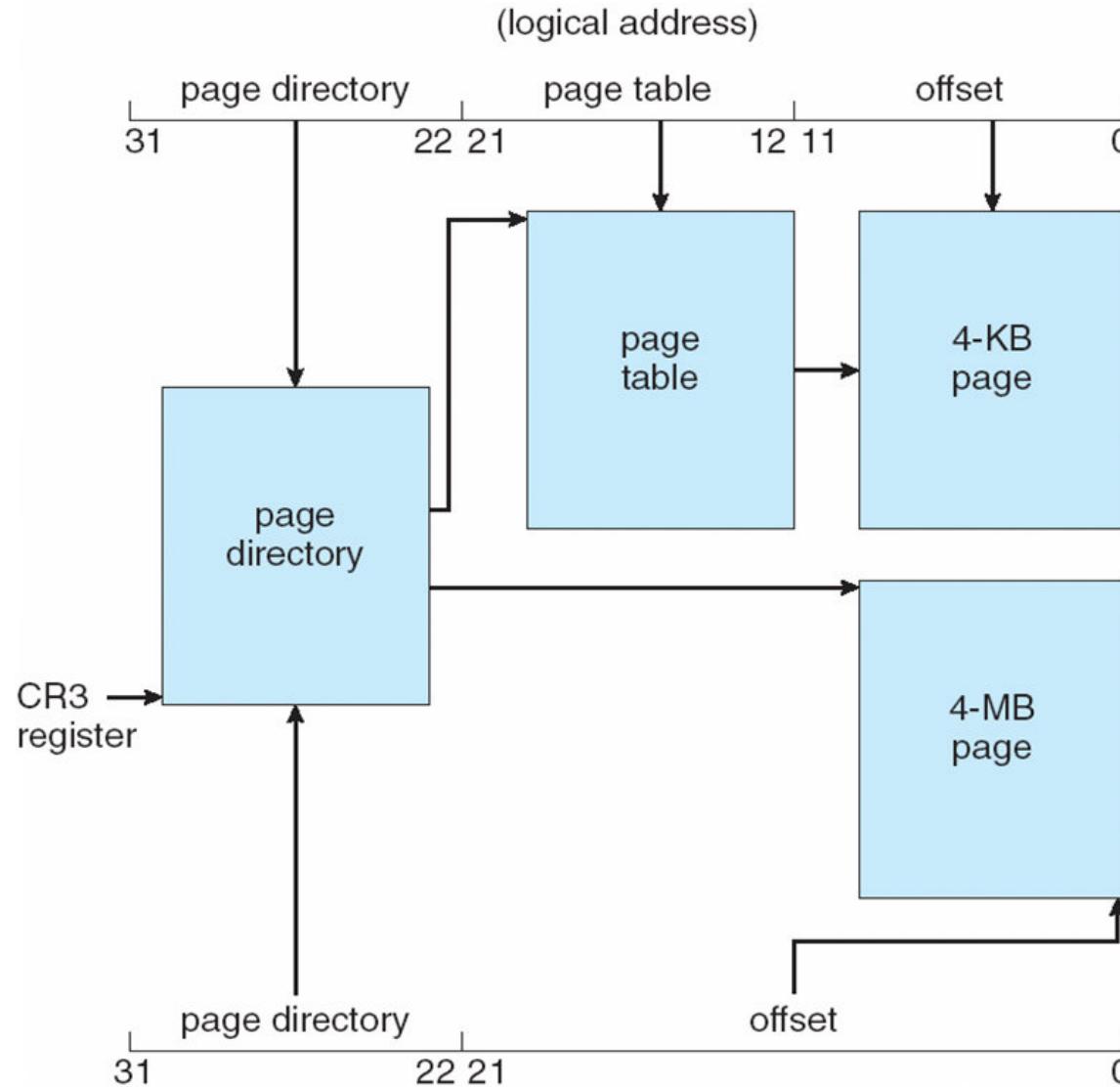




# Intel IA-32 Segmentation

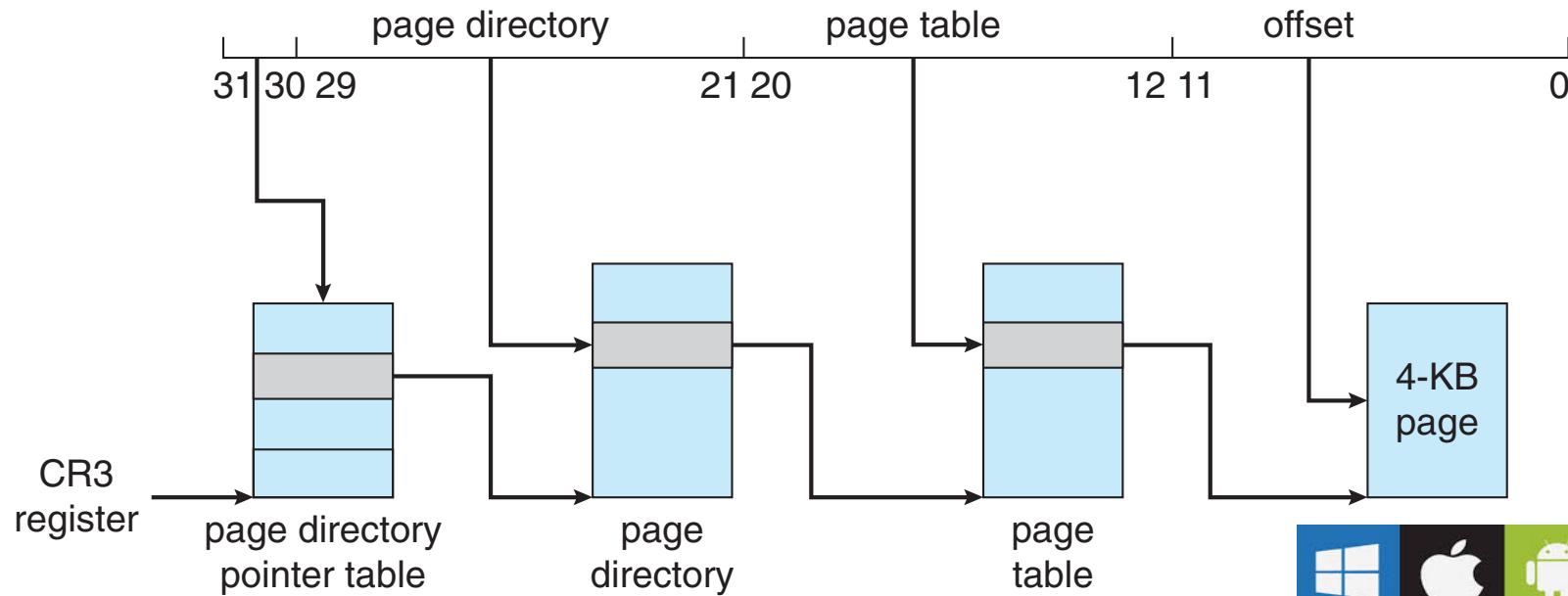


# Intel IA-32 Paging Architecture

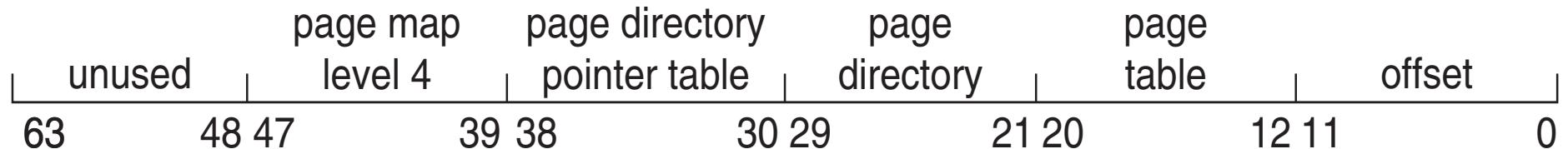


# Intel IA-32 Page Address Extensions

- 32-bit address limits led Intel to create **page address extension (PAE)**, allowing 32-bit apps access to more than 4GB 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

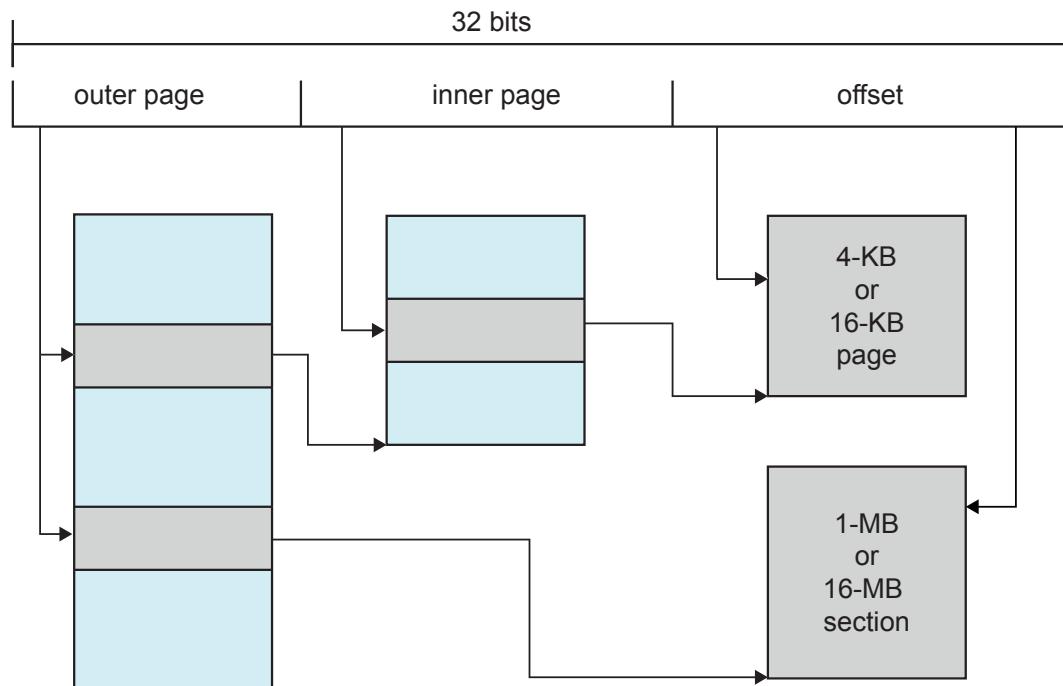


- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits



# 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 missouters are checked, and on misspage table walk performed by CPU





# Summary

- Memory is central to the operation of a modern computer system and
- consists of a large array of bytes, each with its own address.
- One way to allocate addressespace to each process is through the use of base and limit registers. The base register holds the smallest legal physical memory address, and the limit specifies the size of the range.
- Binding symbolic address references to actual physical addresses may occur during (1) compile, (2) load, or (3) execution time.
- An address generated by the CPU is known as a logical address, which the memory management unit (MMU) translates to a physical address in memory.





# Summary (Cont.)

- One approach to allocating memory is to allocate partitions of contiguous memory of varying sizes. These partitions may be allocated based on three possible strategies: (1) first fit, (2) best fit, and (3) worst fit.
- Modern operating systems use paging to manage memory. In this process, physical memory is divided into fixed-sized blocks called frames and logical memory into blocks of the same size called pages
- When paging is used, a logical address is divided into two parts: a page number and a page offset. The page number serves as an index into a per- process page table that contains the frame in physical memory that holds the page. The offset is the specific location in the frame being referenced.





# Summary (Cont.)

- A translation look-aside buffer (TLB) is a hardware cache of the page table. Each TLB entry contains a page number and its corresponding frame.
- Using a TLB in address translation for paging systems involves obtaining the page number from the logical address and checking if the frame for the page is in the TLB. If it is, the frame is obtained from the TLB. If the frame is not present in the TLB, it must be retrieved from the page table.
- Hierarchical paging involves dividing a logical address into multiple parts, each referring to different levels of page tables. As addresses expand beyond 32 bits, the number of hierarchical levels may become large. Two strategies that address this problem are hashed page tables and inverted page tables.





# Summary (Cont.)

- Swapping allows the system to move pages belonging to a process to disk to increase the degree of multiprogramming.
- The Intel 32-bit architecture has two levels of page tables and supports either 4-KB or 4-MB page sizes. This architecture also supports page- address extension, which allows 32-bit processors to access a physical address space larger than 4 GB. The x86-64 and ARMv9 architectures are 64-bit architectures that use hierarchical paging.



# End of Chapter 9



What Is an  
**OPERATING SYSTEM (OS)**  
and How Does It Work

CLEVERISM.COM

# Chapter 10: Virtual Memory



What Is an  
**OPERATING SYSTEM (OS)**  
and How Does It Work

CLEVERISM.COM



# Chapter 10: Outline

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





# Objectives

- Define *virtual memory* and describe its *benefits*.
- Illustrate how pages are loaded into memory using *demand paging*.
- Apply the *FIFO*, *optimal*, and *LRU page-replacement algorithms*.
- Describe the *working set of a process*, and explain how it is related to *program locality*.
- Describe how **Linux**, **Windows 10**, and **Solaris** manage virtual memory.
- Design a *virtual memory manager simulation* in the C programming language.





# Background

## ■ Observation

- *Code* needs to be in memory to be executed, but entire *program* rarely used. E.g., Error code, unusual routines, large data structures
- *Entire program code* are not needed at same time

## ■ Motivation: consider ability to execute *partially-loaded program*

- Program no longer *constrained by limits* of physical memory
- Each program takes *less memory* while running
  - more programs run at the same time
    - ▶ Increase *CPU utilization* and *throughput*
    - ▶ No increase in *response time* or *turnaround time*
- *Less I/O* needed to load or swap programs into memory
  - each user program runs faster



## ■ Virtual memory – separation of user logical memory from physical memory

- ▶ Only *part of the program* needs to be in memory for execution
- ▶ Logical address space can be much *larger than* physical address space
- ▶ Allows address spaces to be *shared* by several processes
- ▶ Allows for more efficient *process creation*
- ▶ *More programs* running concurrently
- ▶ *Less I/O* needed to load or swap processes

## ■ Virtual memory can be implemented via:

- *Demand paging*
- *Demand segmentation*

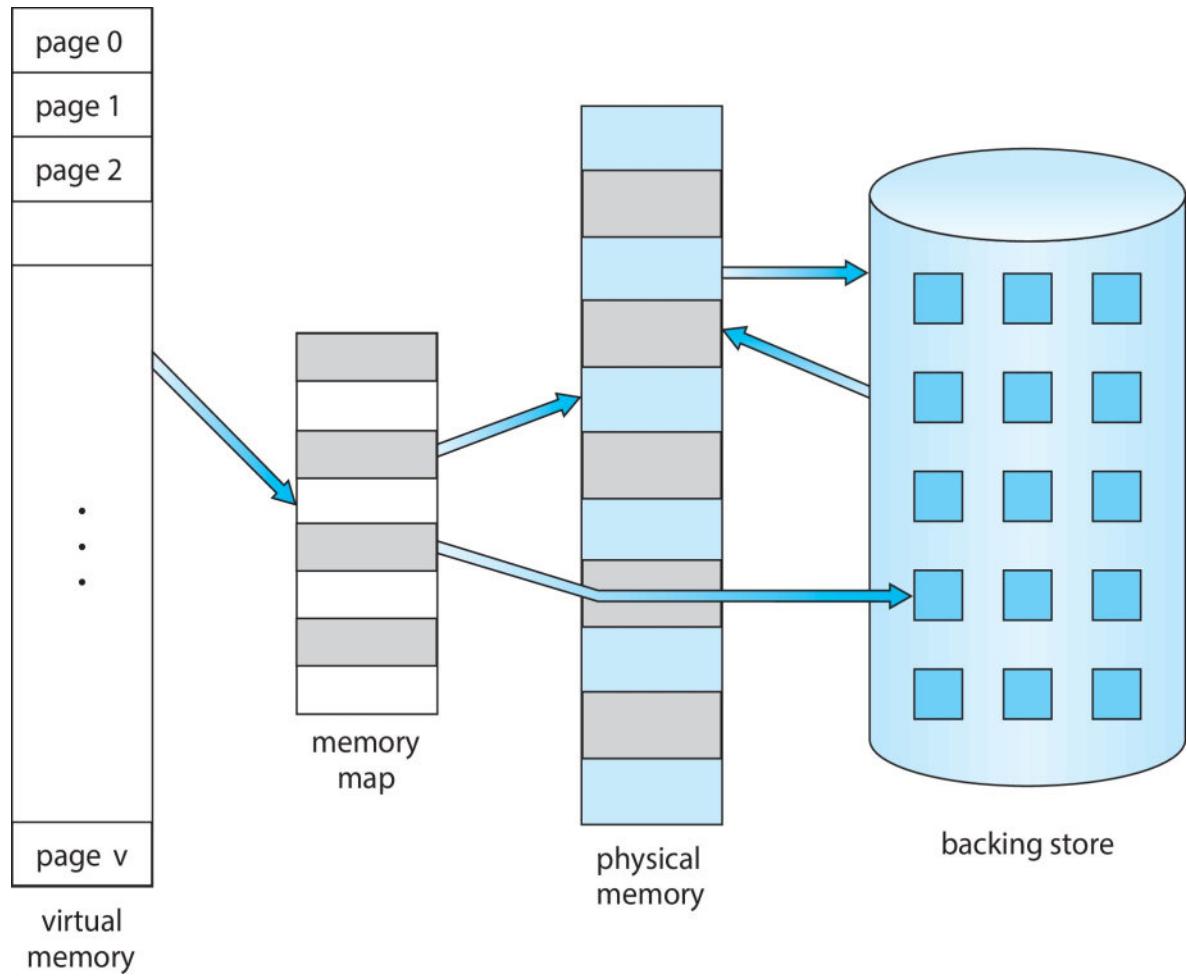


# Virtual Address Space

## ■ *Virtual address space*

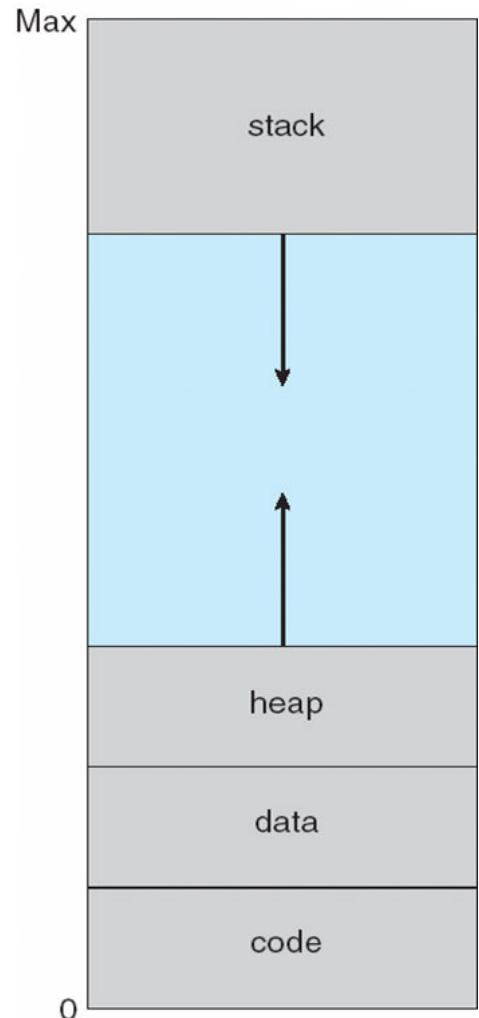
– logical view of how process is stored in memory

- Usually start at address **0**, contiguous addresses until *end of space*
- Meanwhile, physical memory organized in *page frames*
- **MMU** must *map* logical to physical addresses

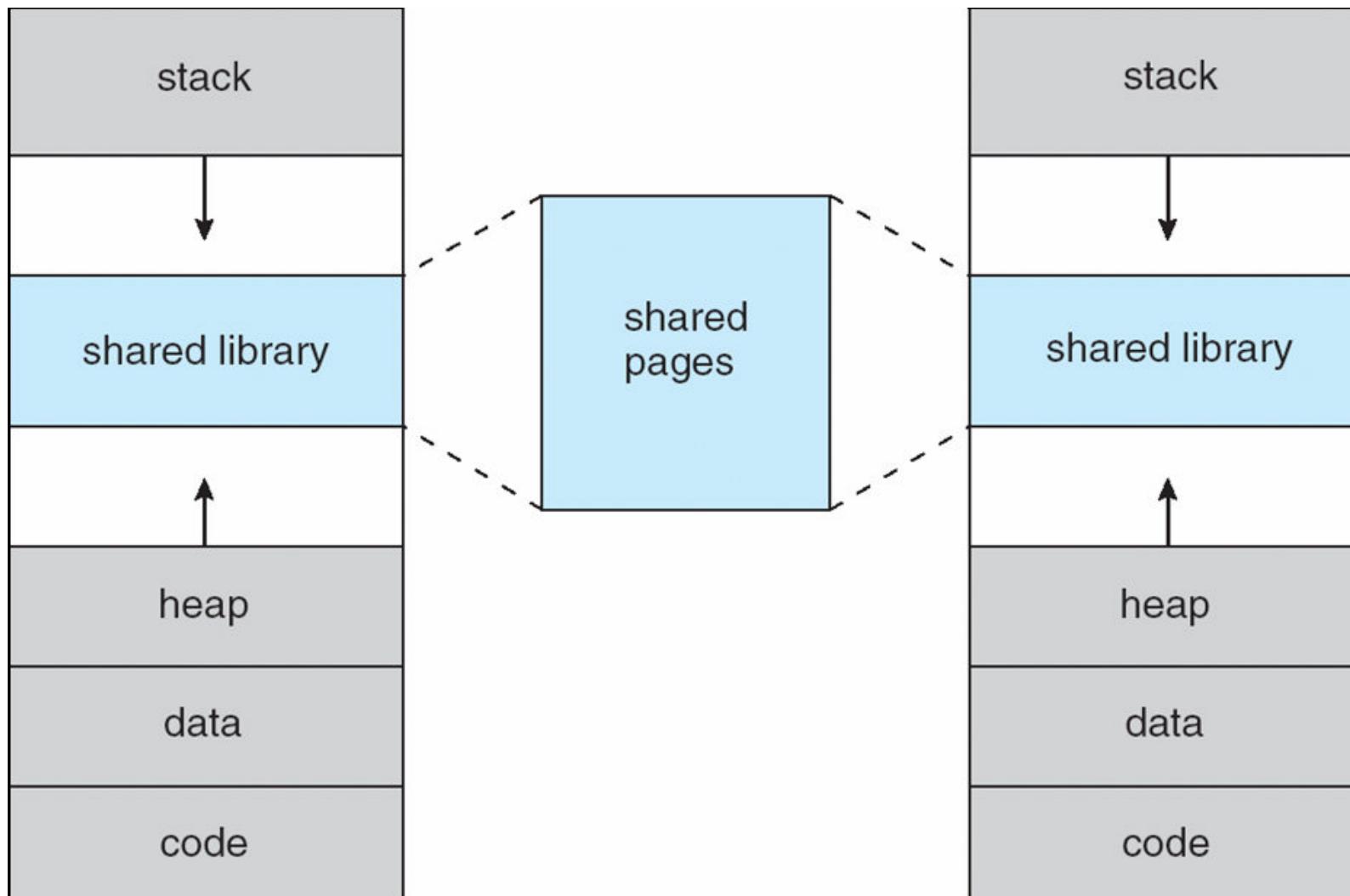


# Virtual Address Space (Cont.)

- Usually *design logical address space* for *stack* to start at Max logical address and grows “down” while *heap* grows “up”
  - Maximizes address space use
  - Unused address space between the two is *hole*
    - ▶ No physical memory needed until *heap* or *stack* grows to a given new page
- Enables *sparse address spaces* with holes left for growth, dynamically linked libraries, etc.
- *System libraries* shared via mapping into virtual address space
- *Shared memory* by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation

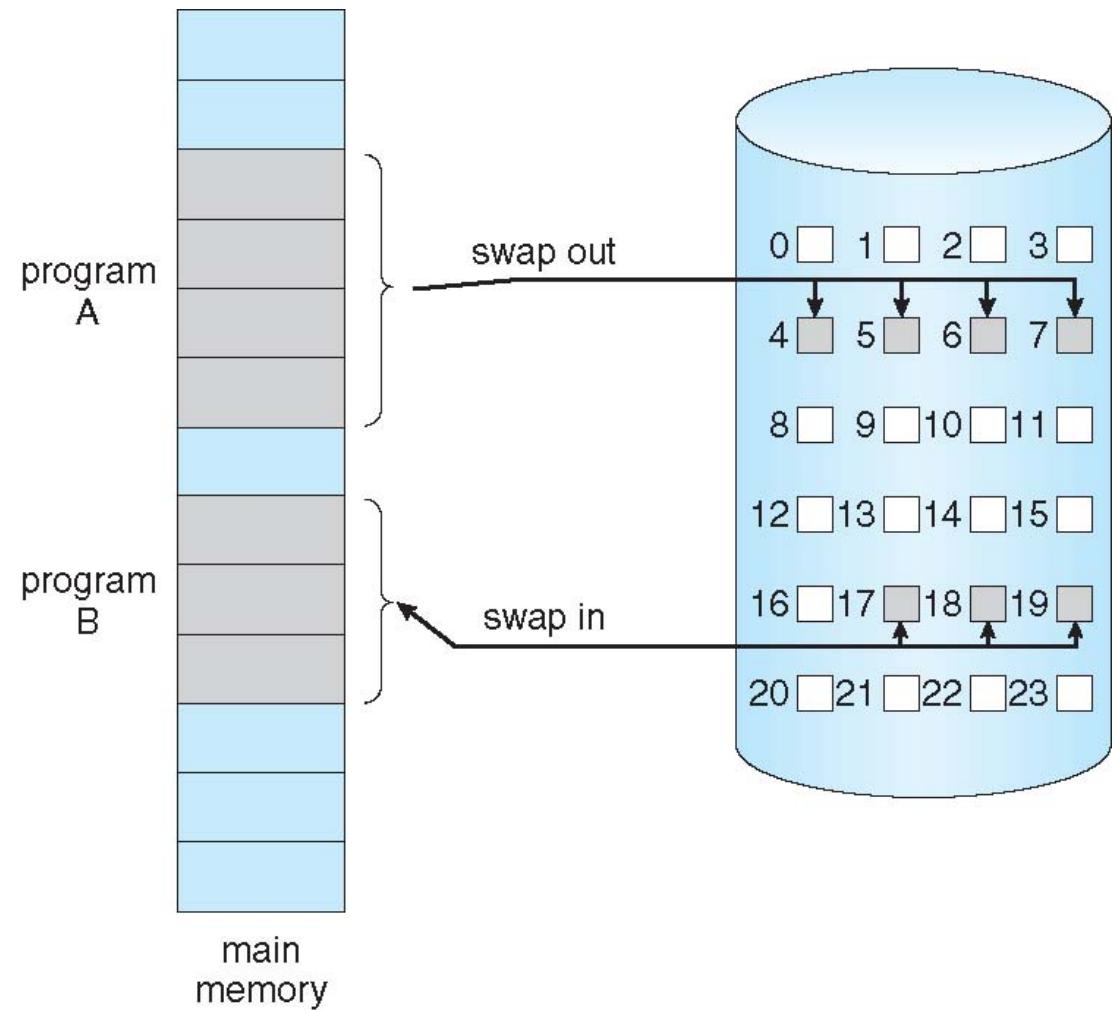


# Shared Library Using Virtual Memory



# Demand Paging

- Could bring entire process into memory at load time
- Or *bring a 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 *paging system with swapping*





# Basic Concepts

- *Lazy swapper* – never swaps a page into memory unless page will be needed (Swapper that deals with pages is a *pager*)
- With swapping, pager *guesses* which pages will be used before swapping out again
  - Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new **MMU** functionality to implement *demand paging*
- If pages needed are already *memory resident*
  - No difference from non demand-paging
- If page needed are not memory resident
  - *Need to detect and load the page into memory from storage*
    - ▶ Without changing program behavior
    - ▶ Without programmer needing to change code



# Valid-Invalid Bit

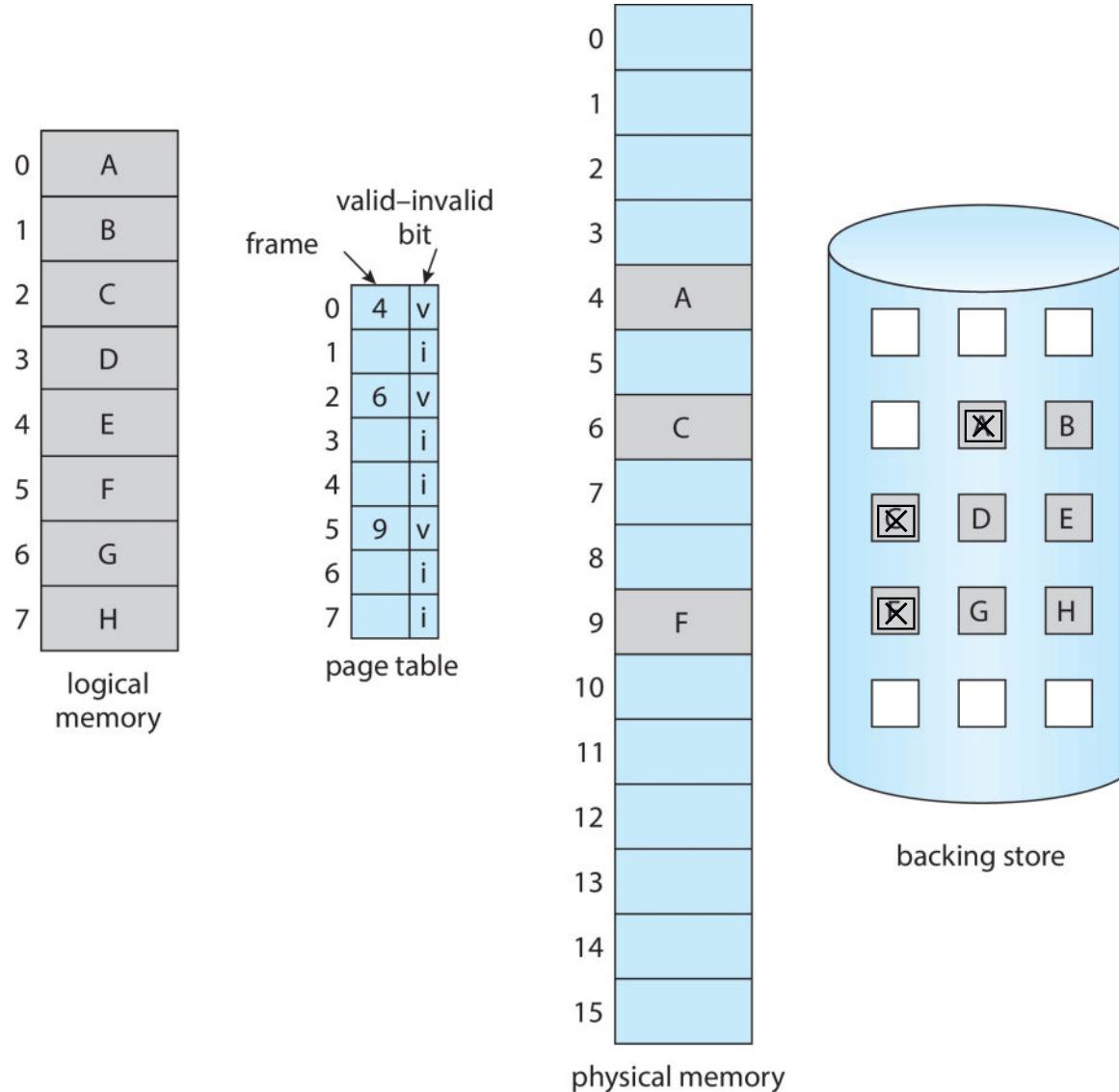
- Page is needed ⇒ reference to it
  - invalid reference? ⇒ abort
  - If valid, but not-in-memory? ⇒ bring to memory
- With each page-table entry, a *valid-invalid bit* is associated
  - **v** ⇒ in-memory (*memory resident*)
  - **i** ⇒ not-in-memory (Initially, valid-invalid bit is set to **i** on all entries)
- During **MMU** address translation, if valid-invalid bit in page-table entry is **i**  
⇒ *page fault*
- Example of a page-table snapshot is shown

| Frame # | valid-invalid bit |
|---------|-------------------|
|         |                   |
|         | v                 |
|         | v                 |
|         | v                 |
|         | i                 |
| ...     |                   |
|         | i                 |
|         | i                 |

page table



# Page Table When Some Pages Are Not in Main Memory



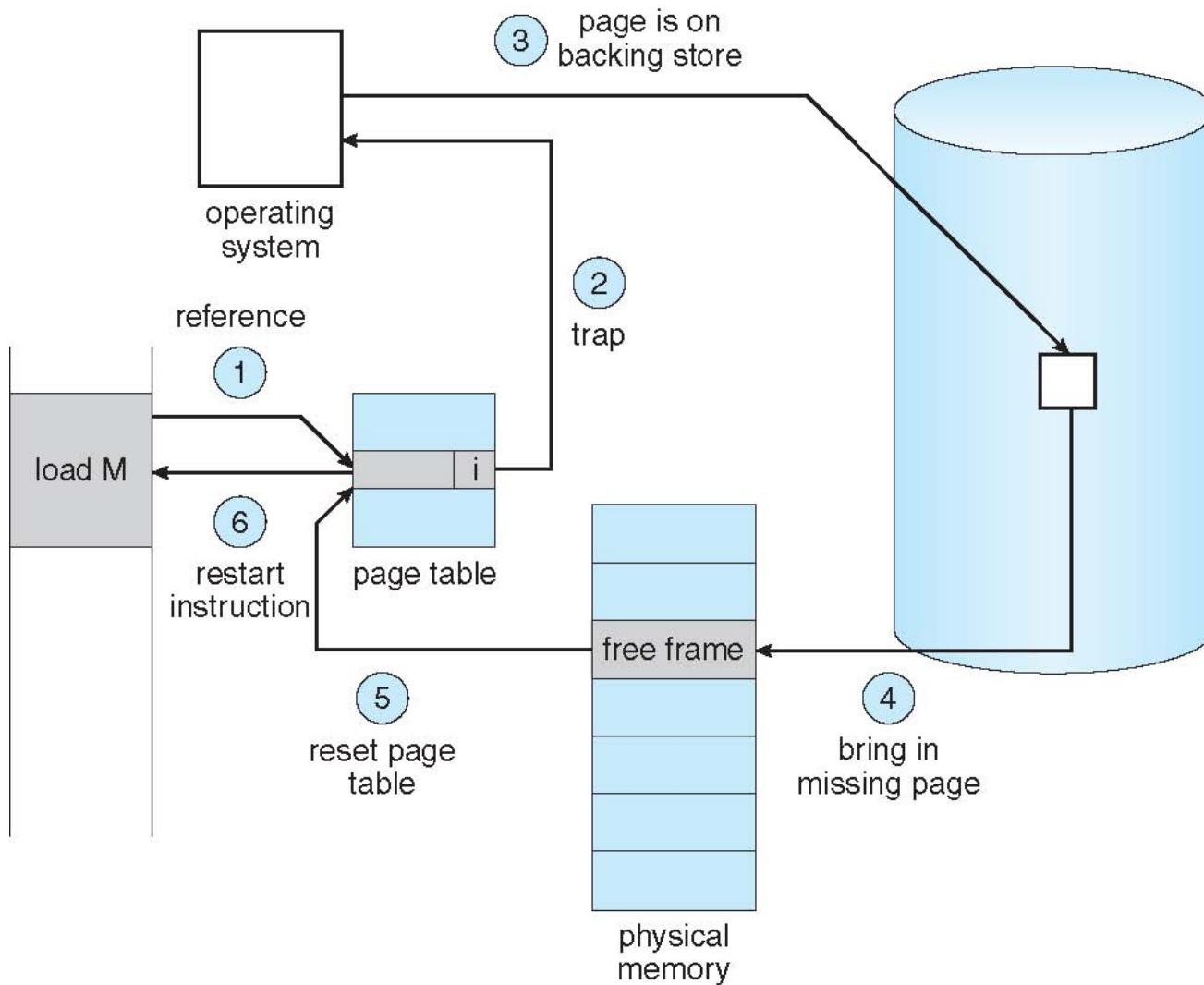


# Steps in Handling Page Fault

1. If there is a reference to a *missed page*, first reference to that page will trap to operating system  
    ⇒ *Page fault*
2. Operating system looks at another table to decide:
  - Invalid reference? ⇒ abort
  - Or just not-in-memory?
3. Find *free* frame
4. Swap page into frame via *scheduled disk operation*
5. Reset page tables to indicate that page now is in memory
  - Set valid-invalid bit = **v**
6. Restart the instruction that caused the page fault



# Steps in Handling a Page Fault (Cont.)



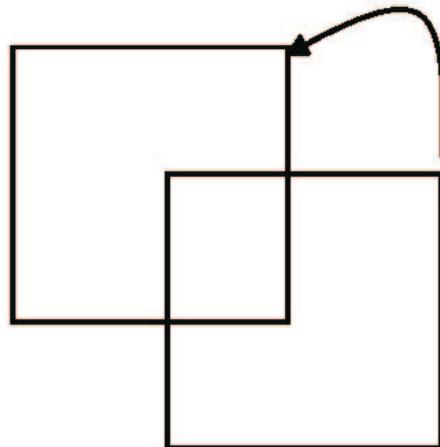
# Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident ⇒ *page fault*
  - And for every other process pages on first access ⇒ *pure demand paging*
- Actually, a given instruction could access multiple pages ⇒ *multiple page faults*
  - E.g., Consider *fetch* and *decode* of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of *locality of reference*
- Hardware support needed for *demand paging*
  - Page table with valid-invalid bit
  - Secondary memory (*swap device* with *swap space*)
  - Instruction restart

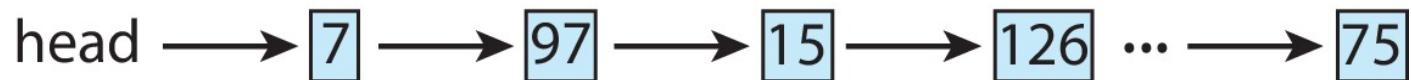


# Instruction Restart

- Consider an instruction that could access several different locations,  
e.g.,
  - Block move
  - Auto increment/decrement location
- Restart the whole operation?
  - ▶ What if source and destination overlap?



- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory
- Most operating systems maintain a *free-frame list* – a pool of free frames for satisfying such requests



- Operating system typically allocate free frames using a technique known as *zero-fill-on-demand* – the content of the frames zeroed-out before being allocated
- When a system starts up, all available memory is placed on the free-frame list





# Stages in Demand Paging – Worse Case

1. Trap to the operating system
2. Save the 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 (disk)
  3. Begin the transfer of the page to a free frame





## Stages in Demand Paging (Cont.)

6. While waiting, allocate the CPU to some other user processes
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user processes
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



## ■ Three major activities

- *Service the interrupt* – careful coding means just several hundred instructions needed
- Read the page – lots of time
- Restart the process – again just a small amount of time

## ■ Page fault rate $p$ , $0 \leq p \leq 1$

- if  $p = 0$ , no page faults
- if  $p = 1$ , every reference is a fault

## ■ *Effective Access Time (EAT)*

$$\text{EAT} = (1 - p) \times \text{memory access} + p \times (\text{page fault overhead} + \text{swap page out} + \text{swap page in})$$



# Example of Demand Paging

- Memory access time = *200 nanoseconds*
- Average page-fault service time = *8 milliseconds*
- $$\begin{aligned} \text{EAT} &= (1 - p) \times 200 + p \times (8 \text{ milliseconds}) \\ &= (1 - p) \times 200 + p \times 8,000,000 \\ &= 200 + p \times 7,999,800 \end{aligned}$$
- If one access out of 1,000 causes a page fault, then  
 $\text{EAT} = 8.2 \text{ microseconds.}$   
This is a slowdown by a factor of 40!!
- If we want performance degradation < 10 percent
  - $220 > 200 + 7,999,800 \times p \Rightarrow 20 > 7,999,800 \times p$
  - $\Rightarrow p < 0.0000025$
  - $\Rightarrow < \text{one page fault in every } 400,000 \text{ memory accesses}$





# Demand Paging Optimizations

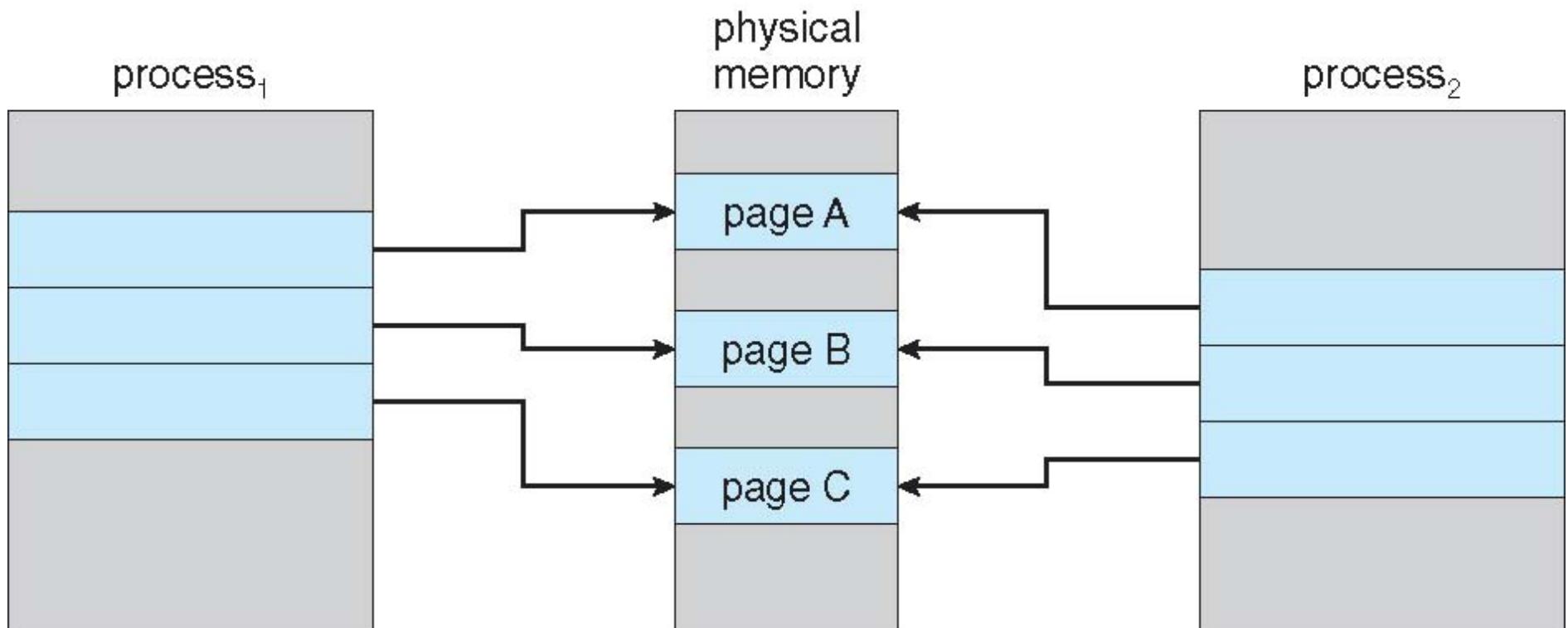
- *Swap space I/O faster than file system I/O* even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time, then page in and out of swap space
  - Used in older BSD Unix
- Demand page in from program 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)



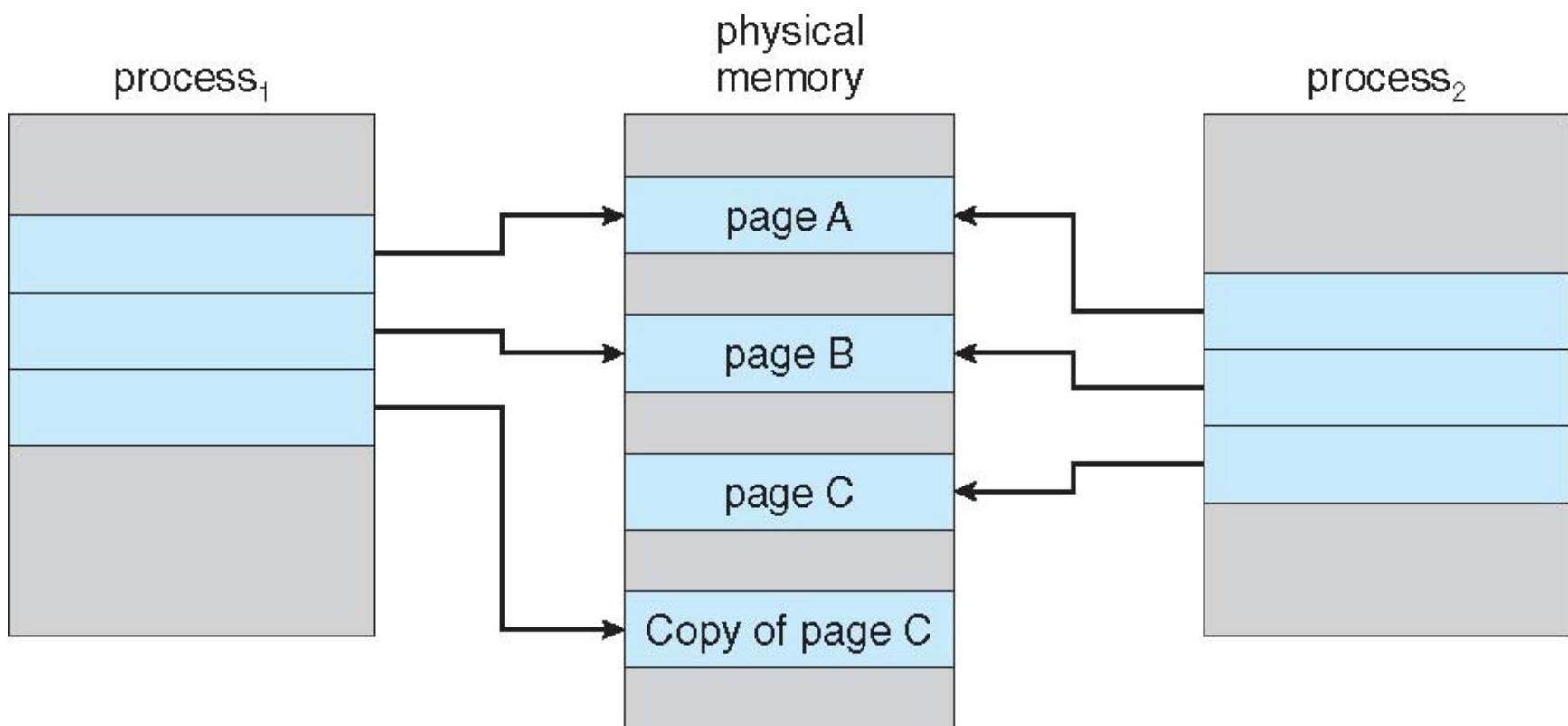
- *Copy-on-Write (COW)* allows both parent and child processes to initially *share the same pages in memory*
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as *only modified pages are copied*
- In general, free pages are allocated 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 and child suspend 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 Modified Page C





# What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc.
- How much to allocate to each?
- *Page replacement* – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

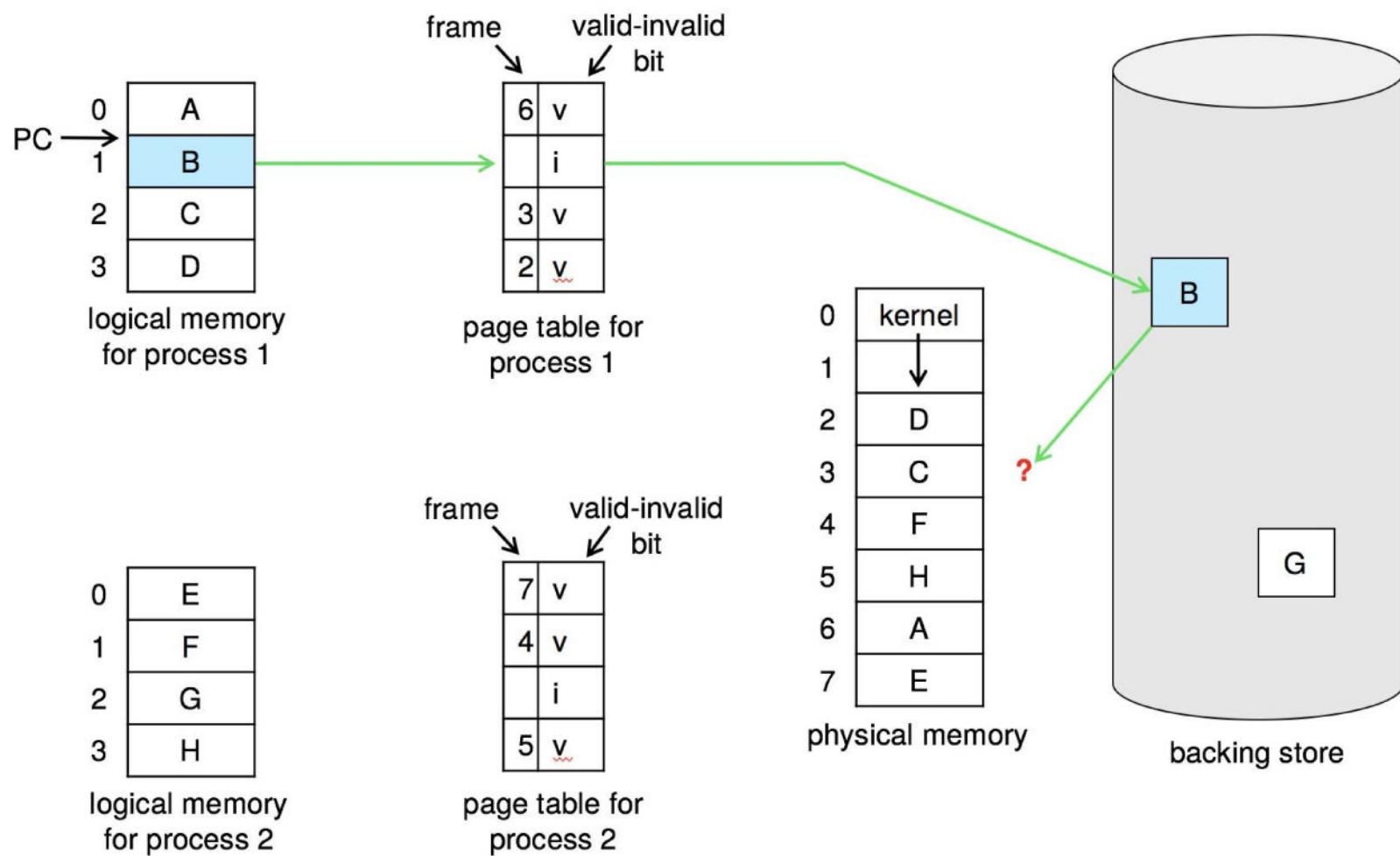


# Page Replacement

- 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*
- *Page replacement* completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory



# Need For Page Replacement



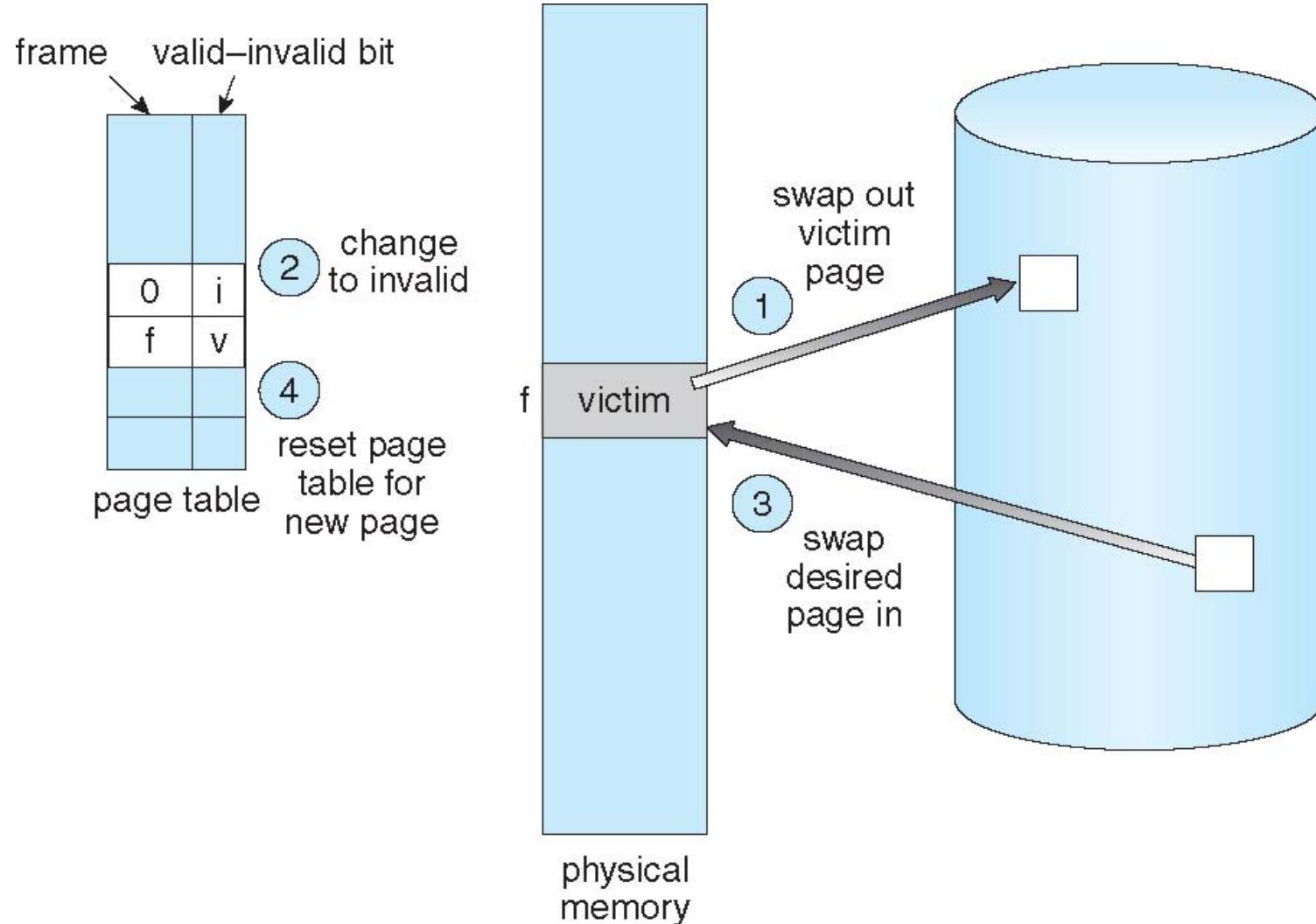


# Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
  - If there is a free frame, use it
  - If there is no free frame, use a *page replacement algorithm* to select a victim frame
    - ▶ Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap
  - **Note:** now potentially 2 page transfers for page fault  $\Rightarrow$  increasing EAT



# Page Replacement





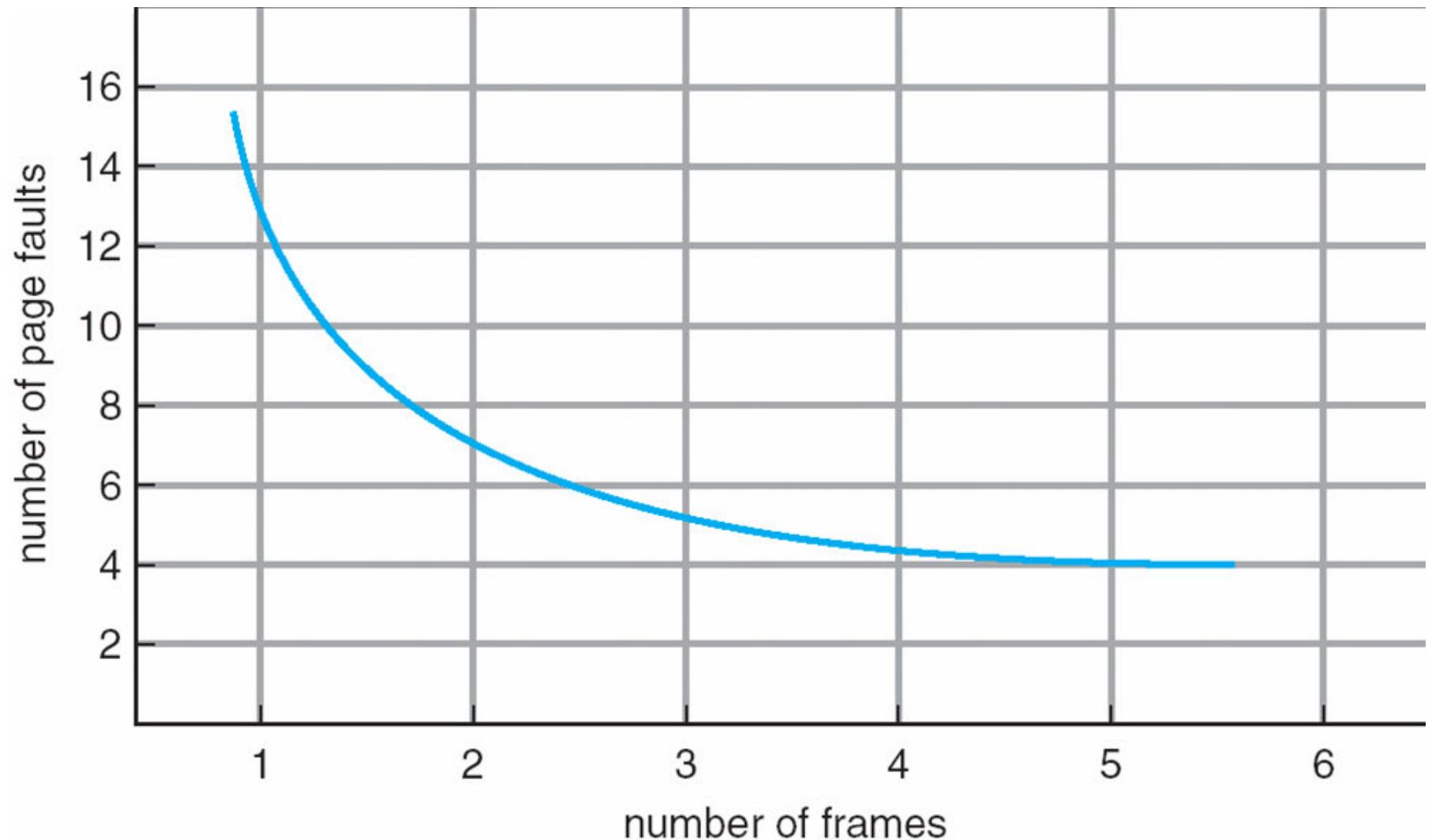
# Page and Frame Replacement Algorithms

- *Frame-allocation algorithm* determines
  - How many frames to give each process
  - Which frames to replace
- *Page-replacement algorithm*
  - Want *lowest page-fault rate* on both first access and re-access
- Evaluate algorithm by running it on a particular *string of memory references* (reference string) and computing the *number of page faults* on that string
  - String is just page numbers, 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**



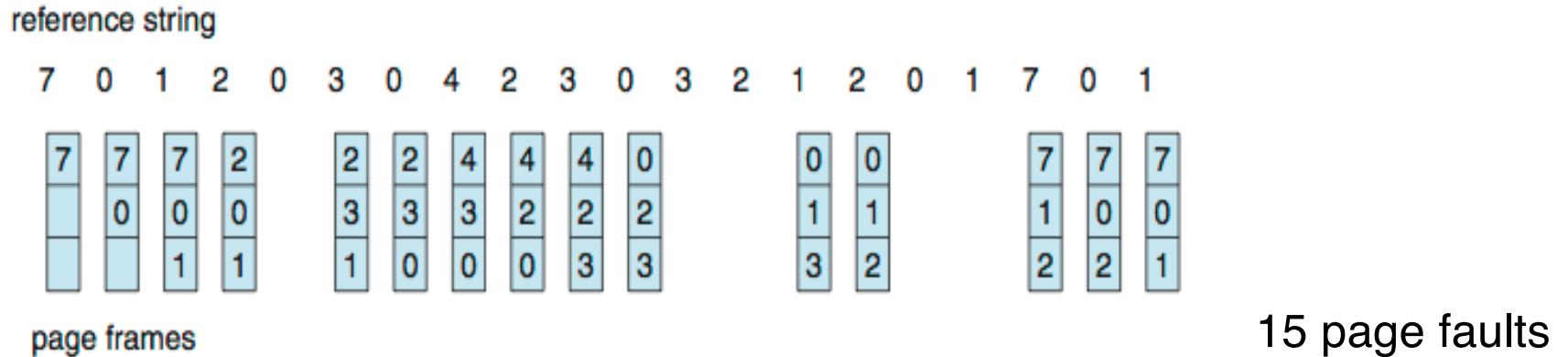


# Graph of Page Faults Versus The Number of Frames



# First-In-First-Out (FIFO) Algorithm

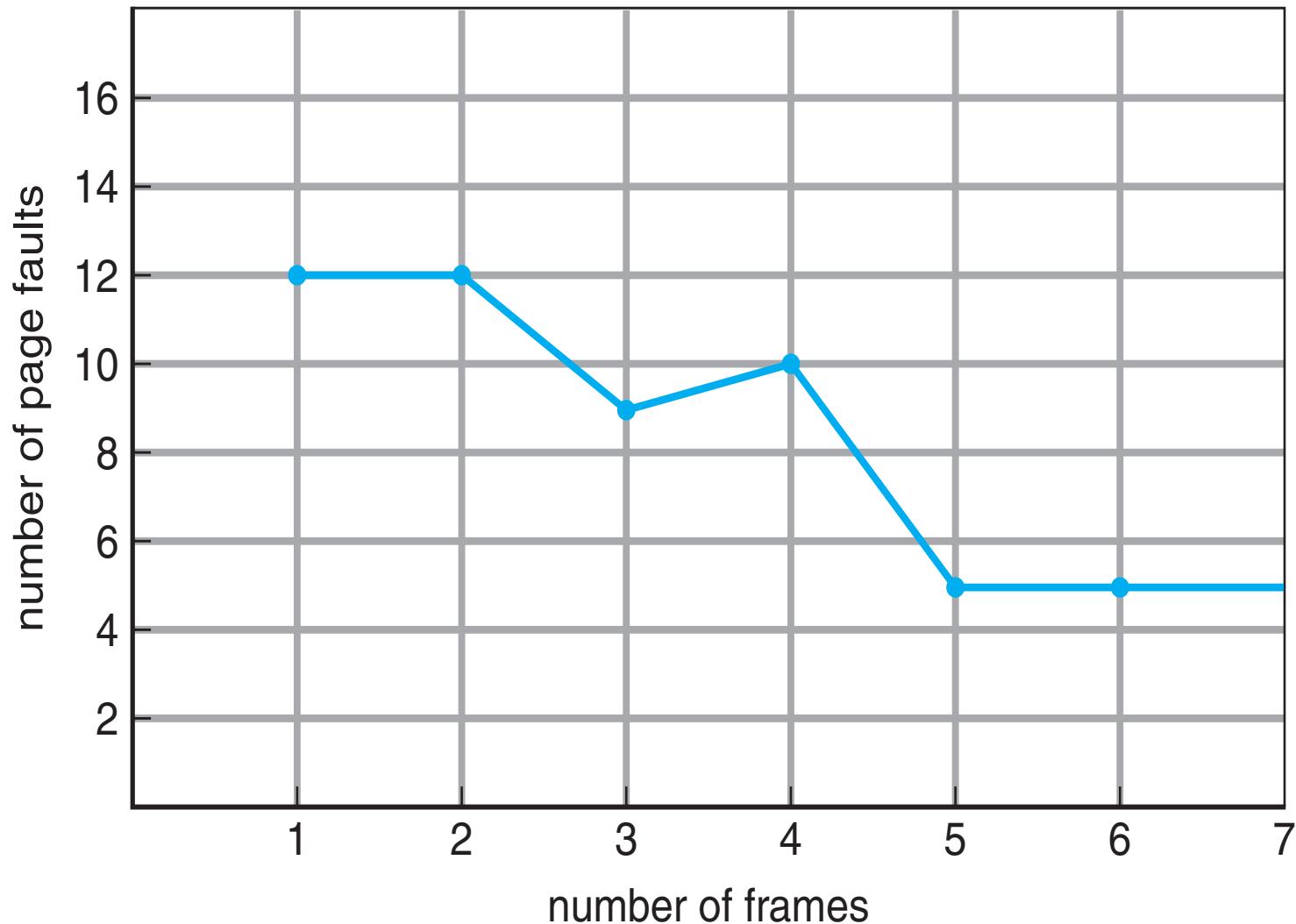
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



- 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**
- How to track ages of pages?
  - Just use a FIFO queue



# FIFO Illustrating Belady's Anomaly

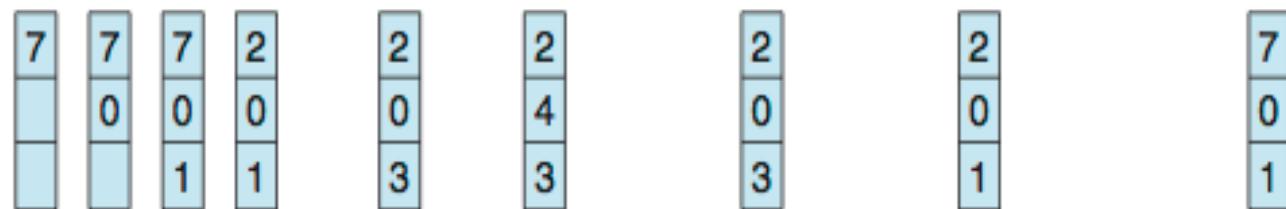


# Optimal (OPT) Algorithm

- *Replace page that will not be used for longest period of time*
  - 9 is optimal for the example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs

reference string

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

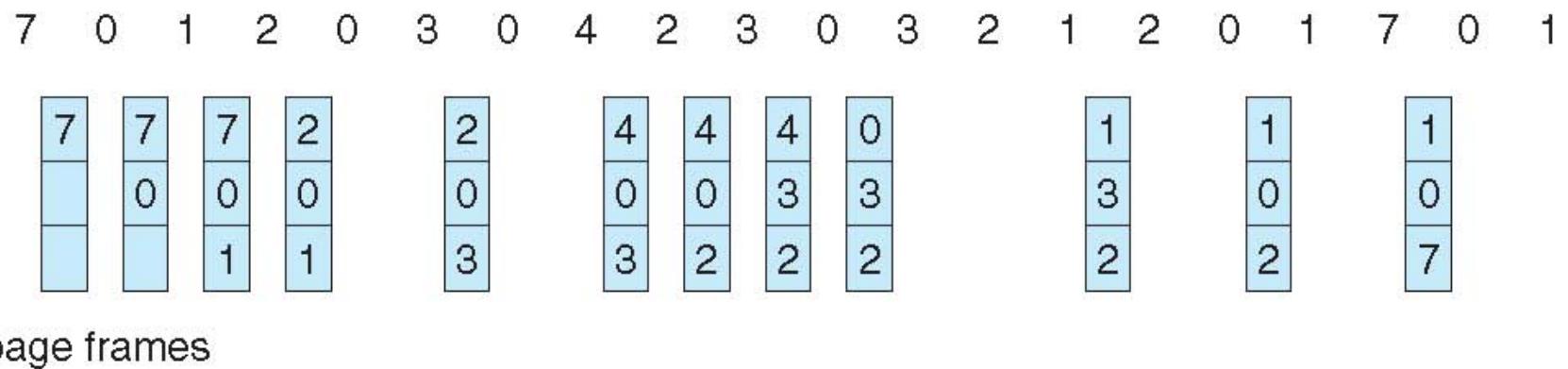


page frames



# Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- *Replace page that has not been used in the most amount of time*
- Associate time of last use with each page reference string



- 12 faults – better than FIFO but worse than OPT
- *Generally good algorithm and frequently used*
- But how to implement?



## ■ *Counter* implementation

- Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
- When a page needs to be changed, check counters to find smallest value
  - ▶ Search through table needed

## ■ *Stack* implementation

- Keep a stack of page numbers in a double link form:
- Page referenced:
  - ▶ move it to the top
  - ▶ requires 6 pointers to be changed (for the example of reference string)
- But each update more expensive
- No search for replacement

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

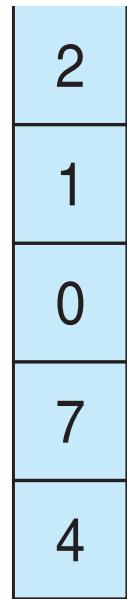




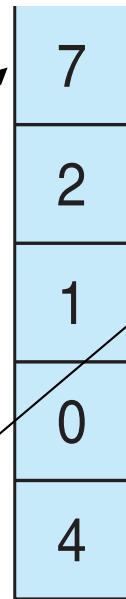
# Use Of A Stack to Record Most Recent Page References

reference string

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



stack  
before  
a



stack  
after  
b

a b

■ LRU needs special hardware and still slow

■ *Reference bit*

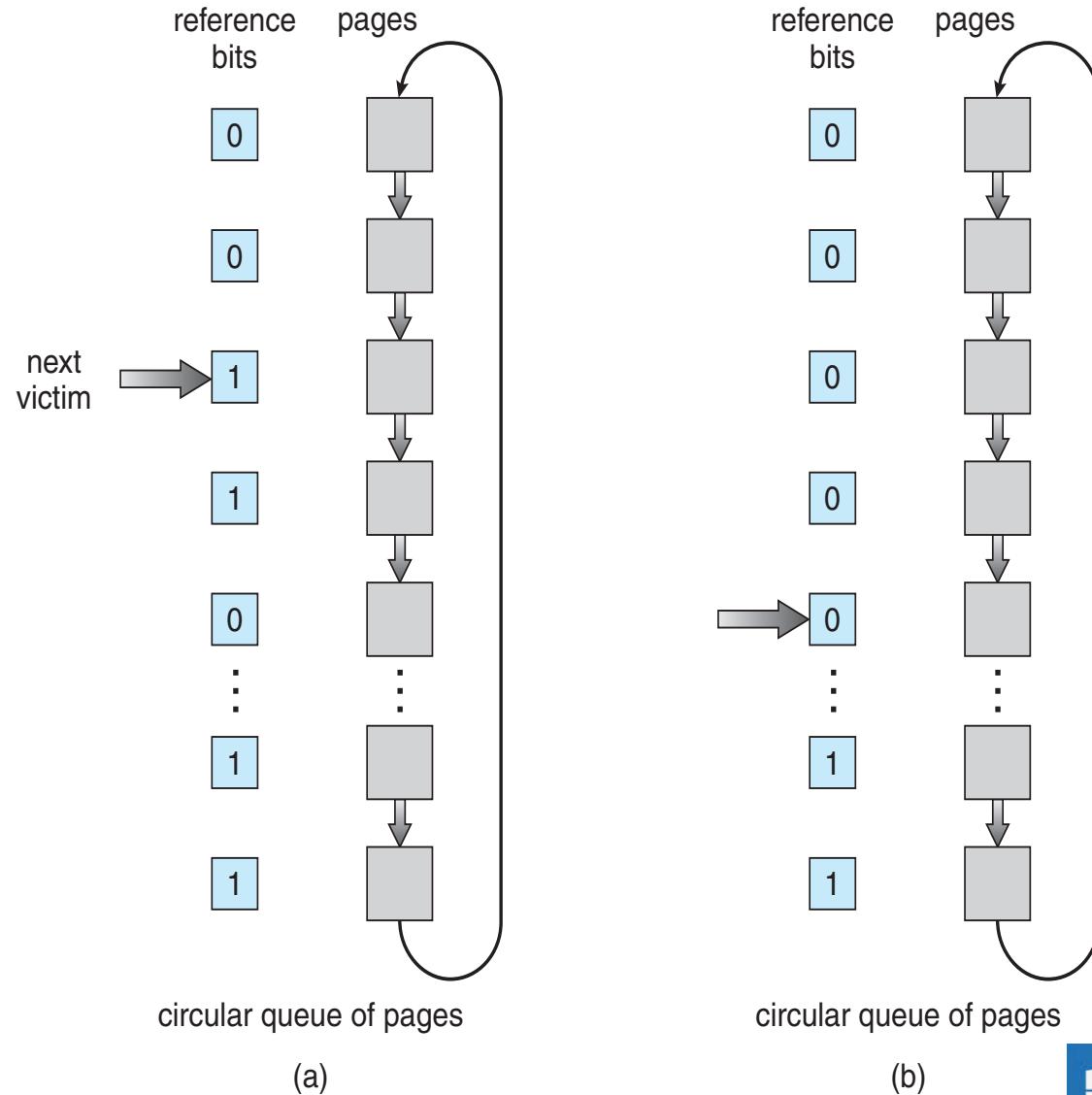
- With each page associate a bit, initially = 0
- When page is referenced, bit set to 1
- Replace any page with reference bit = 0 (if one exists)
  - ▶ We do not know the order, however

■ *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



- Improve algorithm by using *reference bit* and *modify bit* (if available) in concert
- Take ordered pair (*reference, modify*):
  - *(0, 0)* neither recently used nor modified => best page to replace
  - *(0, 1)* not recently used but modified => not quite as good, must write out before replacement
  - *(1, 0)* recently used but clean => probably will be used again soon
  - *(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





# Counting Algorithms

- Keep a *counter of the number of references* that have been made to each page
  - Not common
- *Least Frequently Used (LFU) Algorithm*: replaces page with smallest count
- *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



# 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.





# Allocation of Frames

- Each process needs *minimum* number of frames
- Example: IBM 370 – 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 total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- And any variations ...



# Fixed Allocation

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

$s_i$  = size of process  $p_i$

$S = \sum s_i$

$m$  = total number of frames

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

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

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

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





# Global vs. Local Allocation

- *Global replacement* – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput, so more common
- *Local replacement* – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

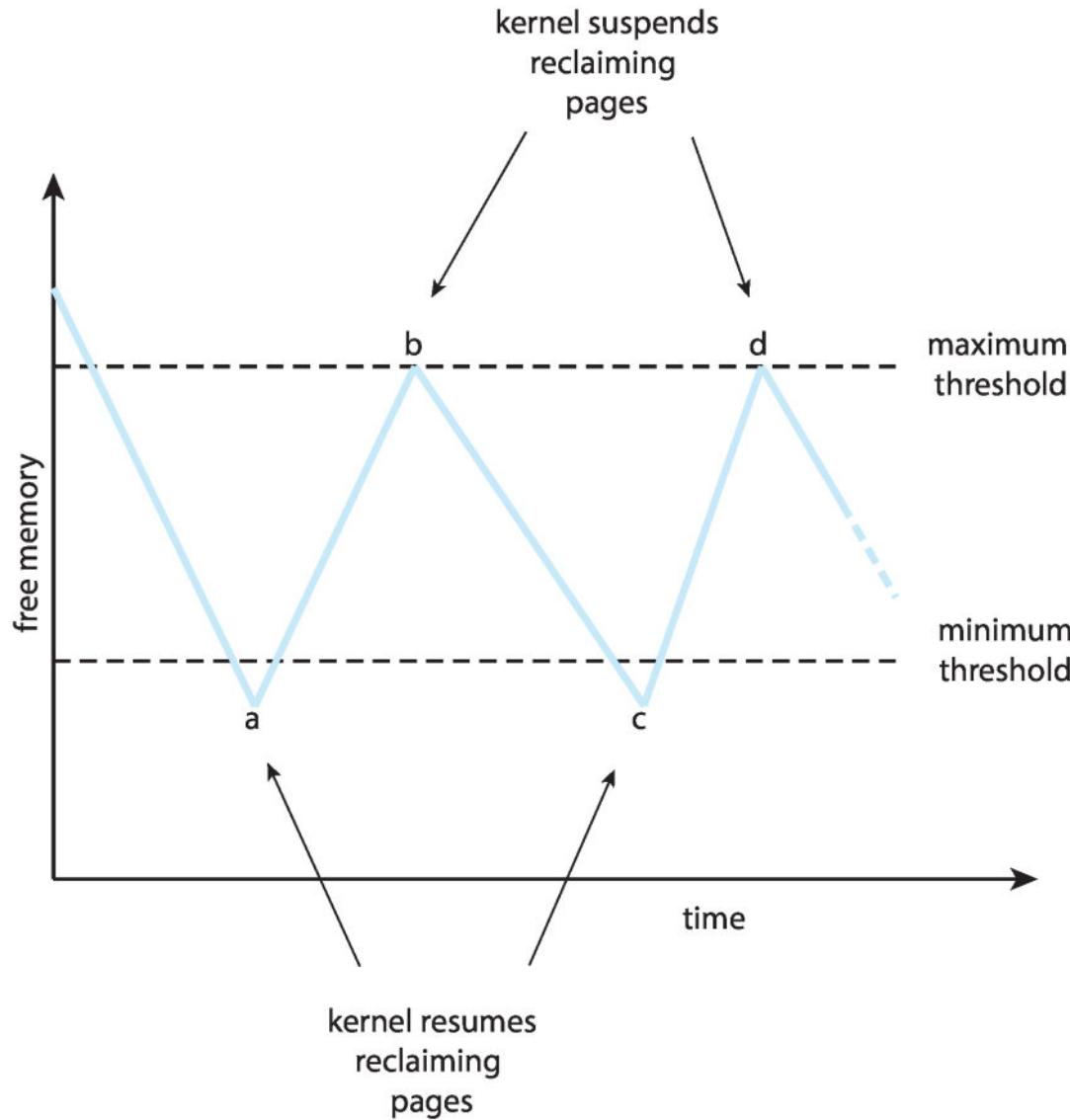




# Reclaiming Pages

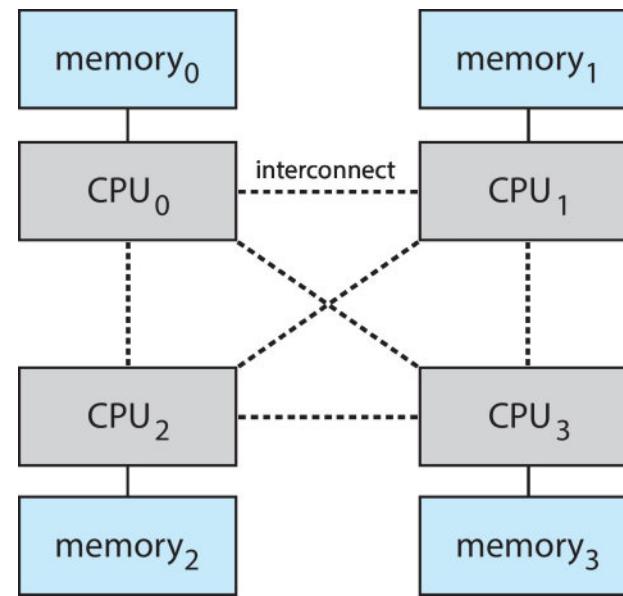
- A strategy to implement *global page-replacement policy*
- **Motivation:** All memory requests are satisfied from the free-frame list, rather than waiting for the list to drop to zero before we begin selecting pages for replacement
- *Page replacement is triggered when the list falls below a certain threshold*
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests

# Reclaiming Pages Example



# Non-Uniform Memory Access (NUMA)

- So far all memory accessed equally
- Many systems are **NUMA** – speed of access to memory varies
  - E.g., Consider system boards containing CPUs and memory, interconnected over a system bus
- NUMA multiprocessing architecture





# Non-Uniform Memory Access (Cont.)

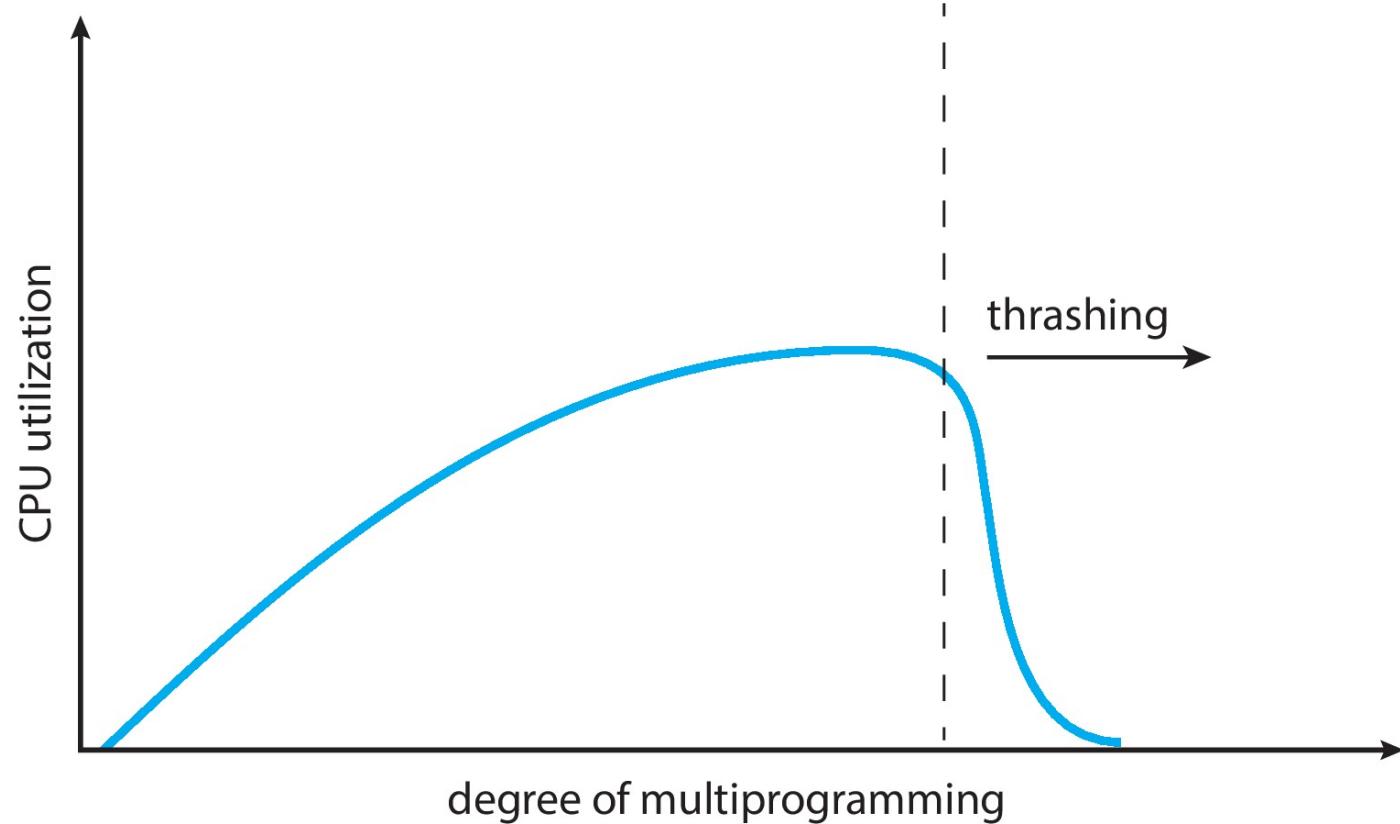
- Optimal performance comes from allocating memory “*close to*” the *CPU* on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - E.g., Solved by creating **Igroups** on Solaris
    - ▶ Structure to track CPU / Memory *low latency groups*
    - ▶ Used scheduler and pager
    - ▶ When possible, schedule all threads of a process and allocate all memory for that process within the *Igroup*

- If a process does not have “enough” frames, the page-fault rate is very high
  - Page fault to get frame
  - Replace existing frame
  - But, quickly need replaced frame back
  - More processes have page faults
  - This leads to:
    - ▶ Low CPU utilization
    - ▶ Operating system thinking that it needs to increase the degree of multiprogramming
    - ▶ Another process added to the system



## Thrashing (Cont.)

- *Thrashing.* A process is busy swapping pages in and out





# Demand Paging and Thrashing

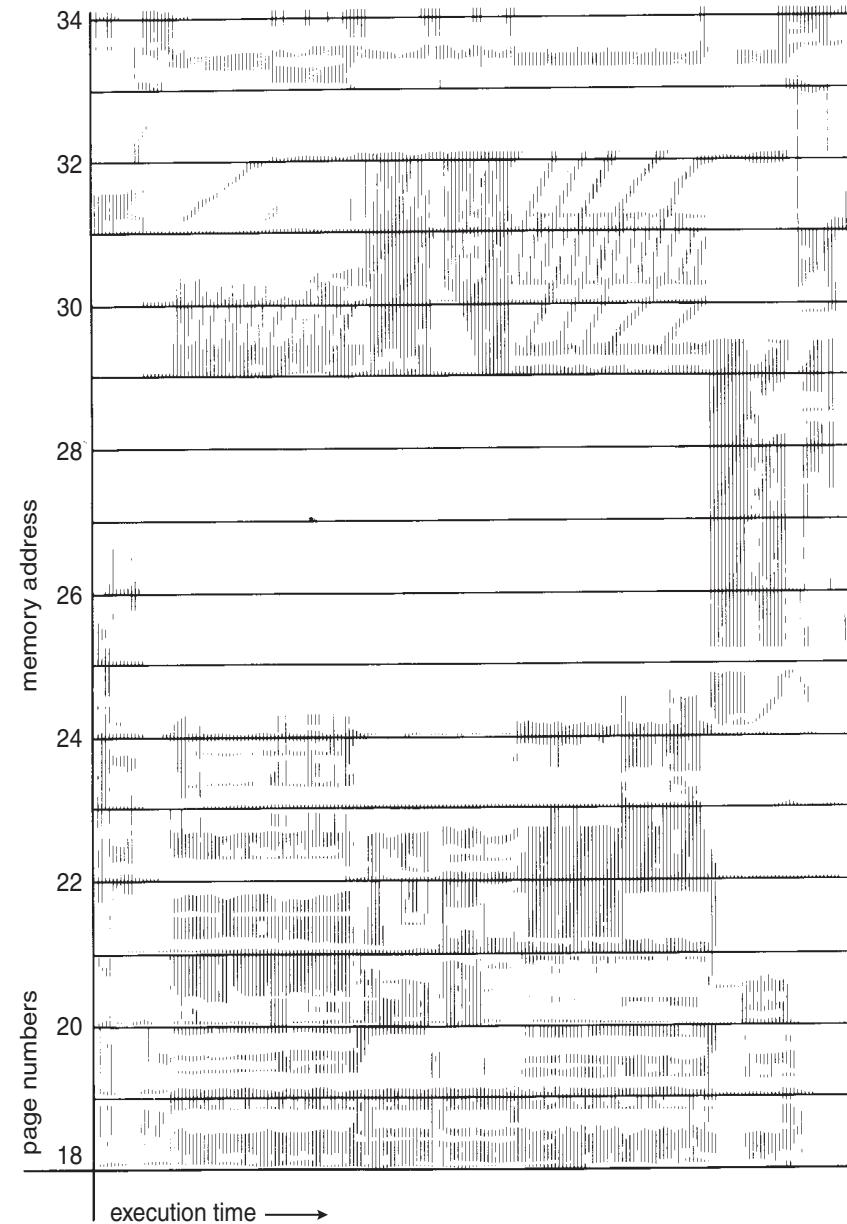
- Why does demand paging work?
- *Locality model*
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?

$\Sigma$  size of locality > total memory size

- Limit effects by using *local* or *priority page replacement*



# Locality In A Memory-Reference Pattern

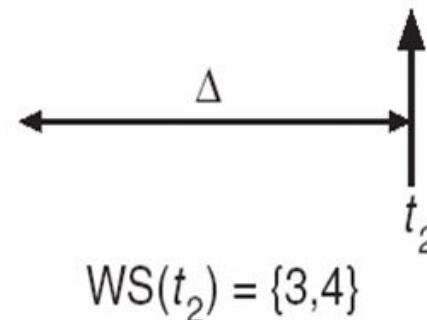
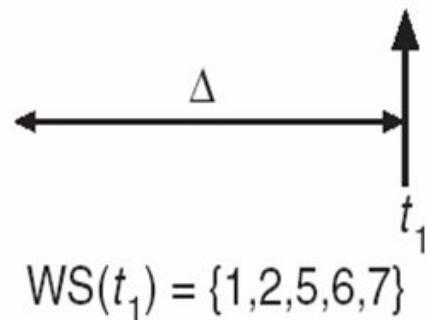


# Working-Set Model

- $\Delta \equiv$  working-set window  $\equiv$  a fixed number of page references
  - E.g., 10,000 instructions
- **WSS<sub>i</sub>** (working set of Process  $P_i$ ) = total number of pages referenced in the most recent  $\Delta$  (varies in time)

page reference table

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





# Working-Set Model (Cont.)

## ■ Working-set window ( $\Delta$ ) and locality

- if  $\Delta$  too small will not encompass entire locality
- if  $\Delta$  too large will encompass several localities
- if  $\Delta = \infty \Rightarrow$  will encompass entire program

## ■ $D = \Sigma WSS_i \equiv$ total demand frames

- Approximation of locality

## ■ $m \equiv$ total number of available frames

## ■ if $D > m \Rightarrow$ Thrashing

- One policy: *if  $D > m$ , then suspend or swap out one of the processes*





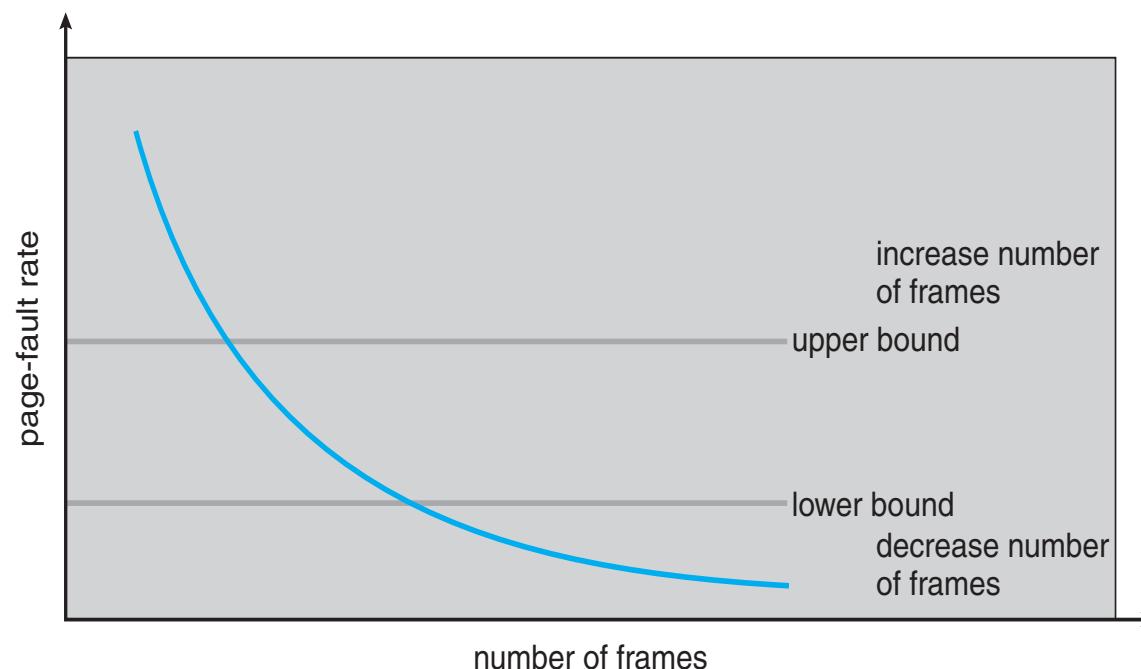
# Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example:  $\Delta = 10,000$ 
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts, copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1  $\Rightarrow$  page in working set
- Why this is not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units



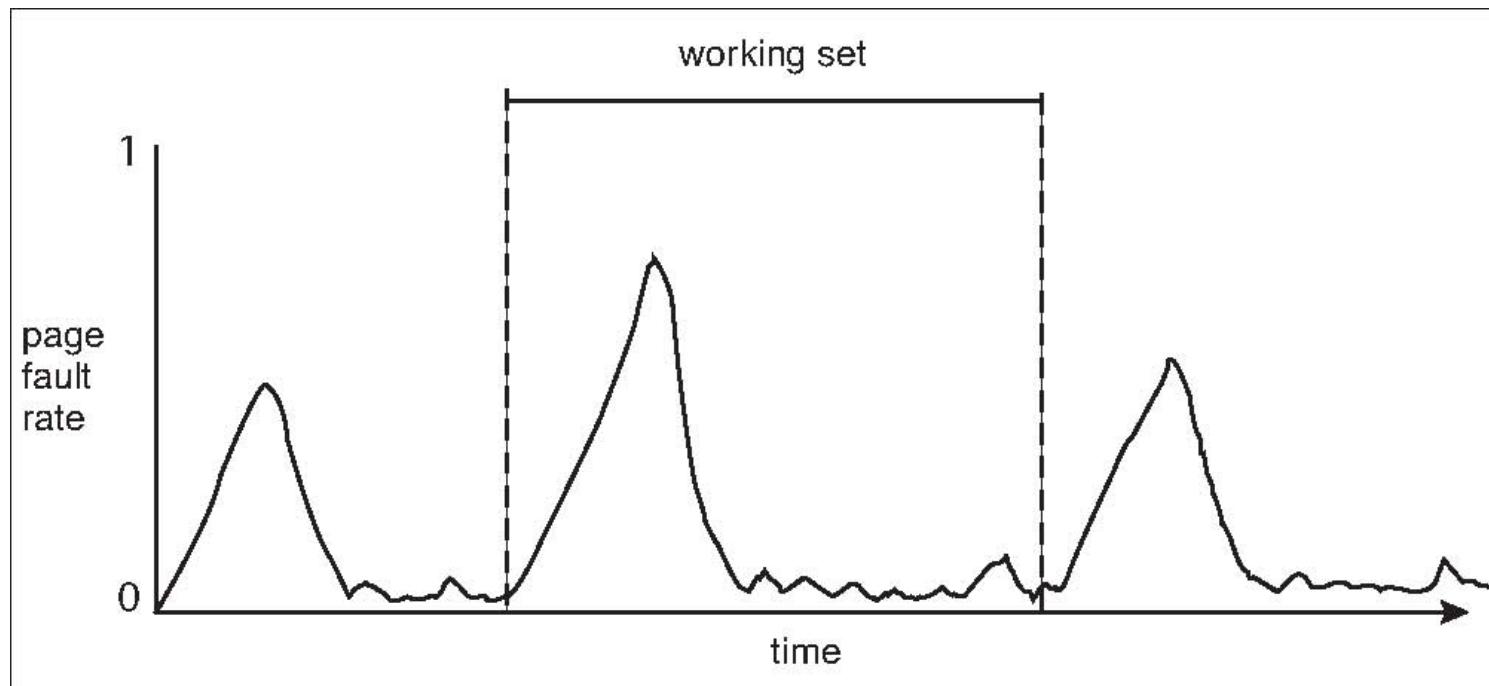
# Page-Fault Frequency

- More direct approach than WSS
- Establish “acceptable” *page-fault frequency (PFF) rate* and use *local replacement policy*
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



# 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





# 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*
    - ▶ E.g., for device I/O

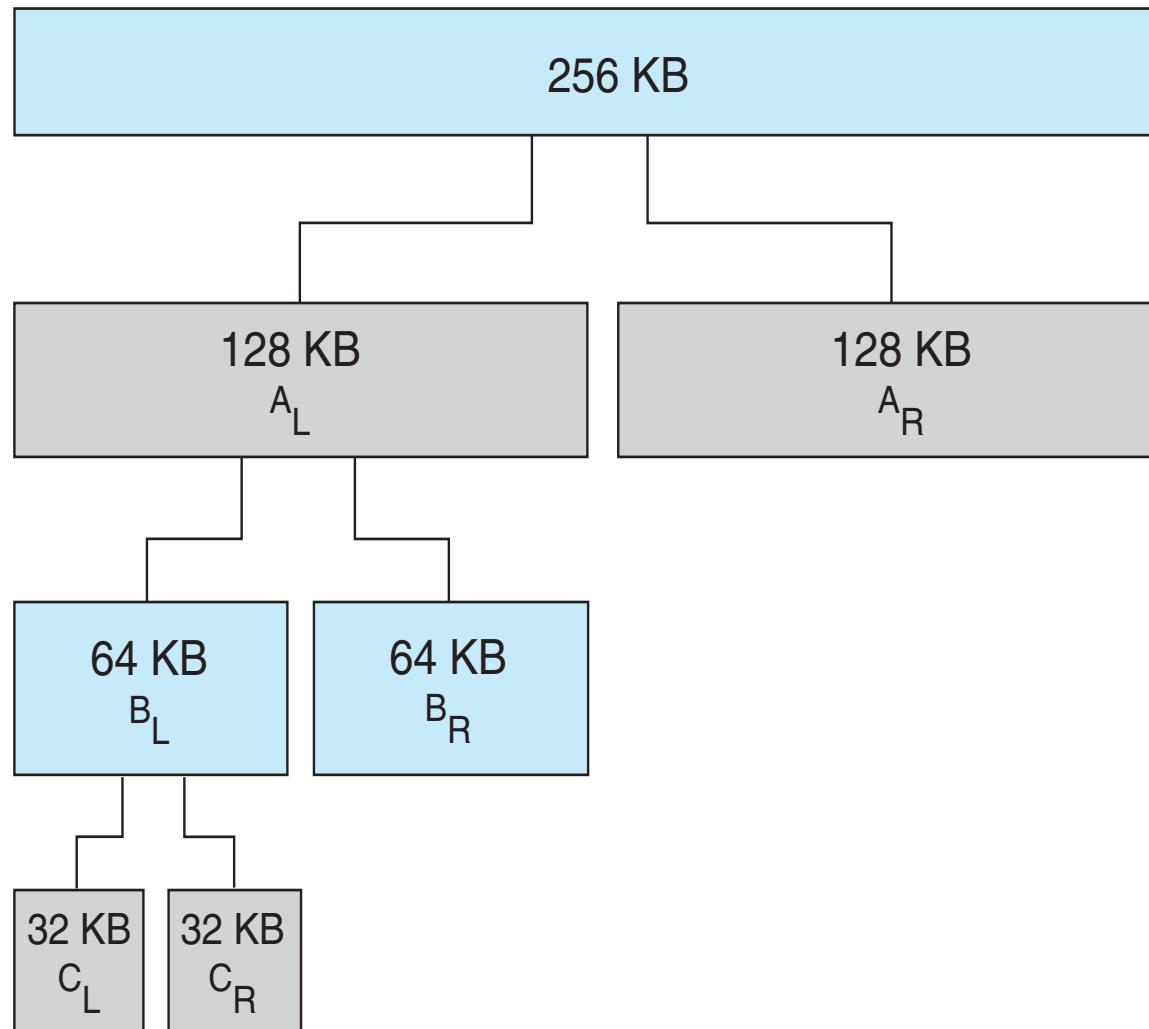
- Allocates memory from *fixed-size segment* consisting of physically-contiguous pages
- Memory allocated using *power-of-2 allocator*
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk 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



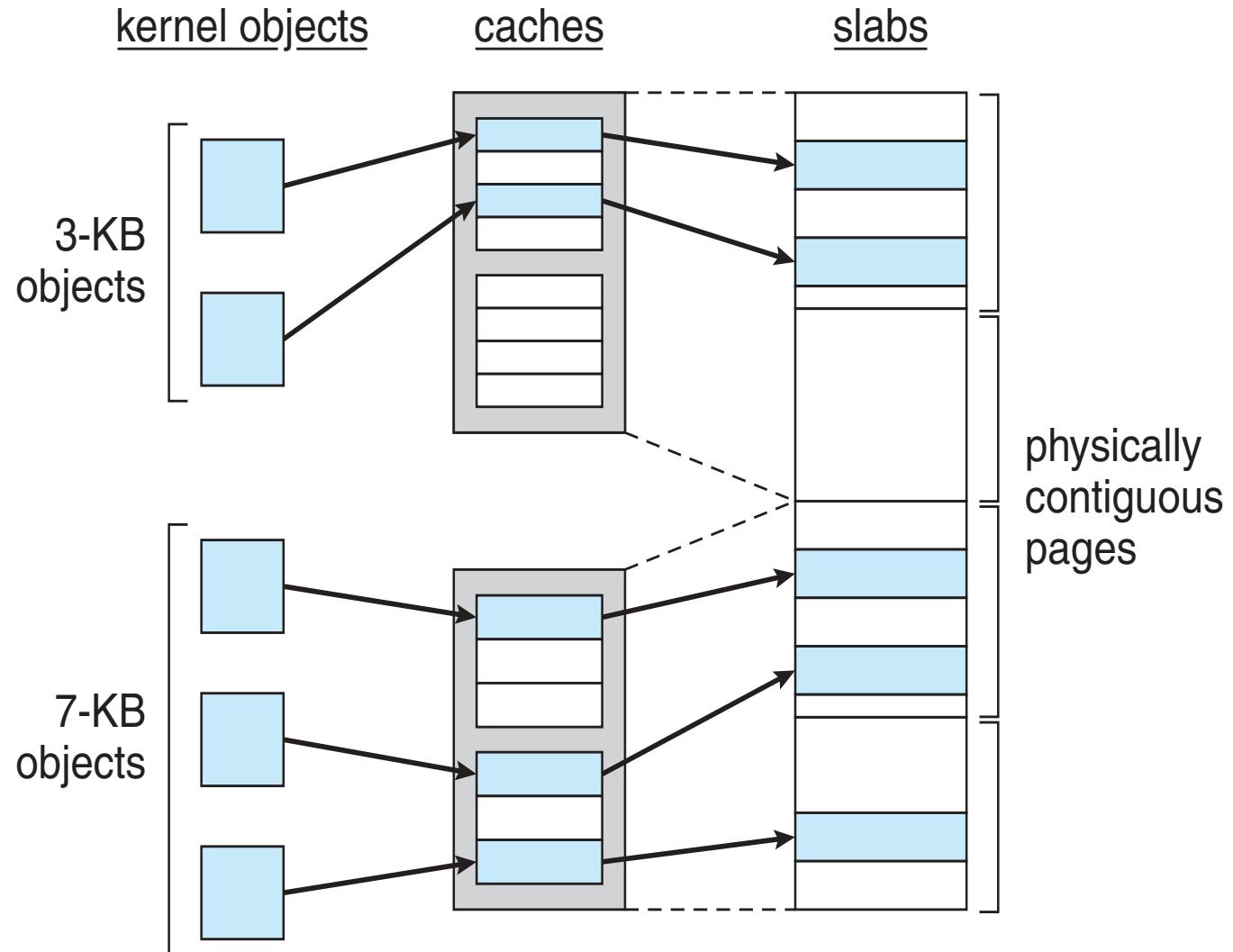


# Slab Allocator

- 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 (Simple List of Blocks) for systems with limited memory
    - ▶ 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

- Pre-paging
- Page size
- TLB reach
- Inverted page table
- Program structure
- I/O interlock and page locking





# Pre-paging

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

- Sometimes OS designers have a choice of page size
  - 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



- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (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





# Example of a 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

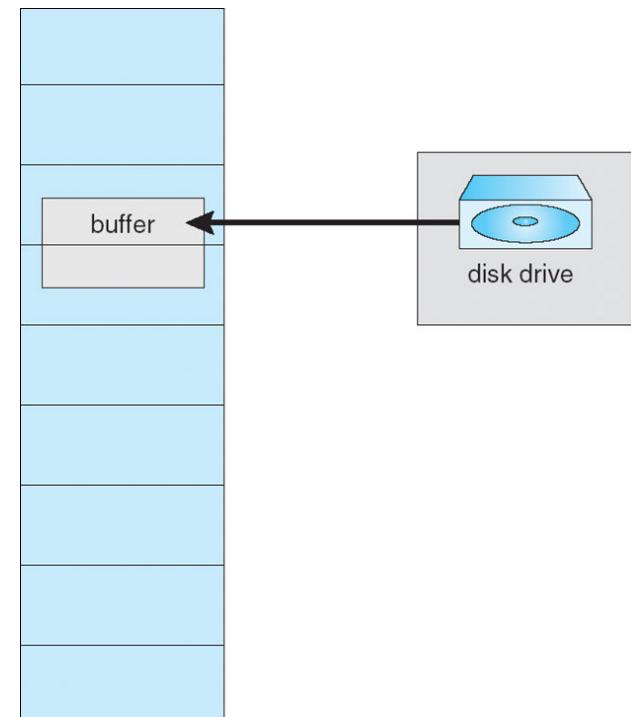
- [Program 2](#)

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

128 page faults



- *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





# Operating System Examples

- Windows
- Solaris



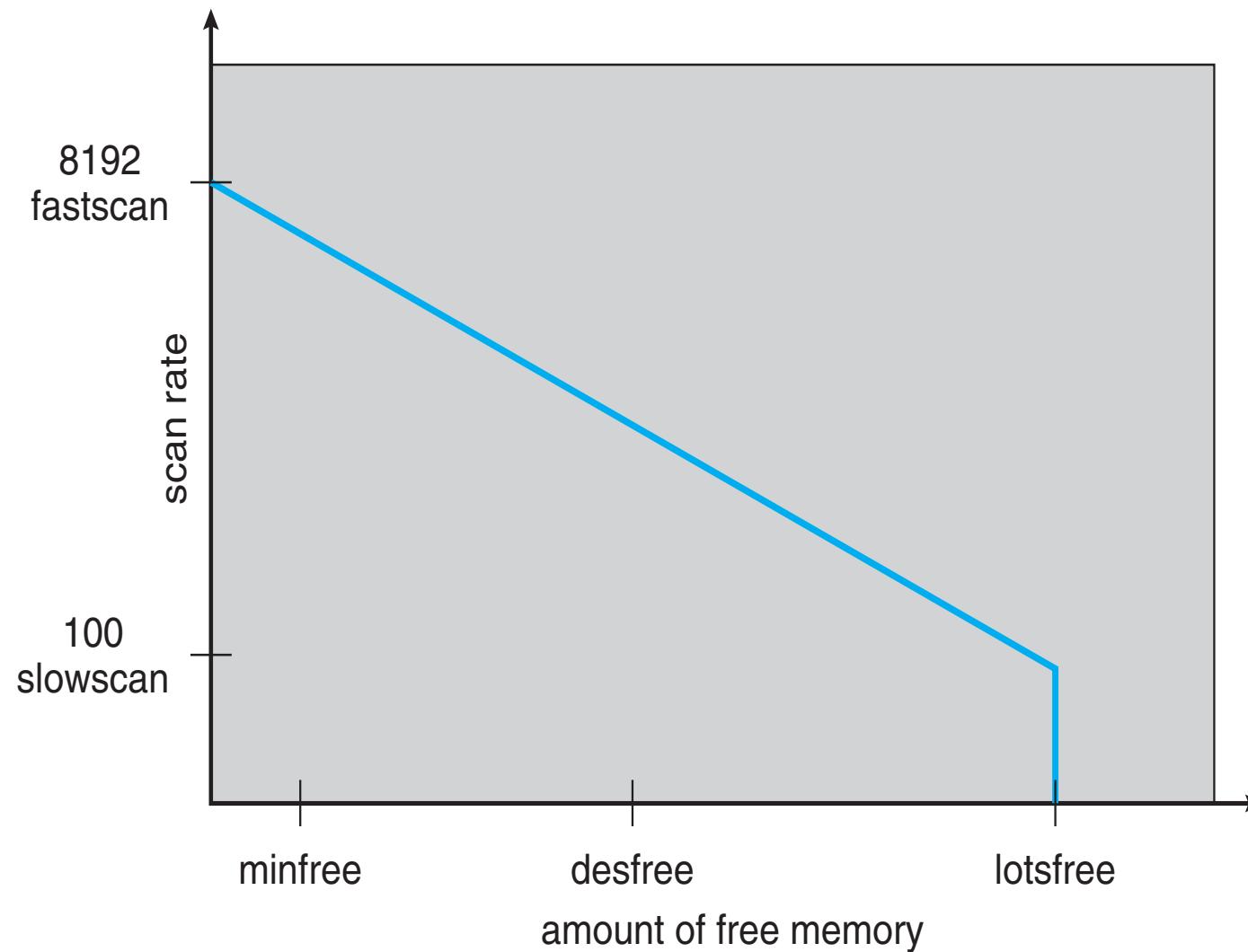
- Uses demand paging with *clustering*. Clustering brings in pages surrounding the faulting page
- Processes are assigned *working set minimum* and *working set maximum*
  - *Working set minimum* is the minimum number of pages the process is guaranteed to have in memory
  - A process may be assigned 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



- 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

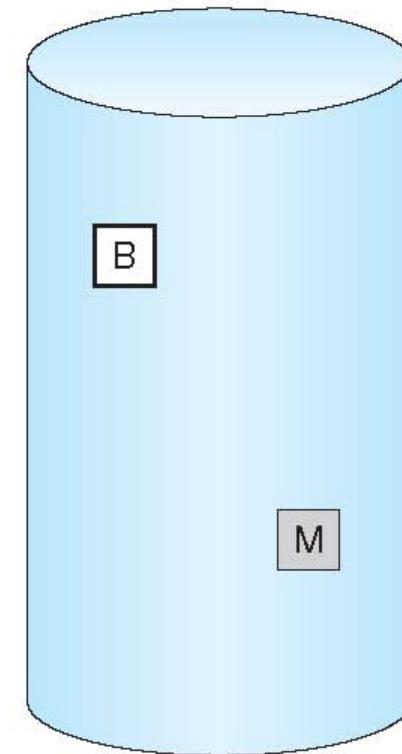
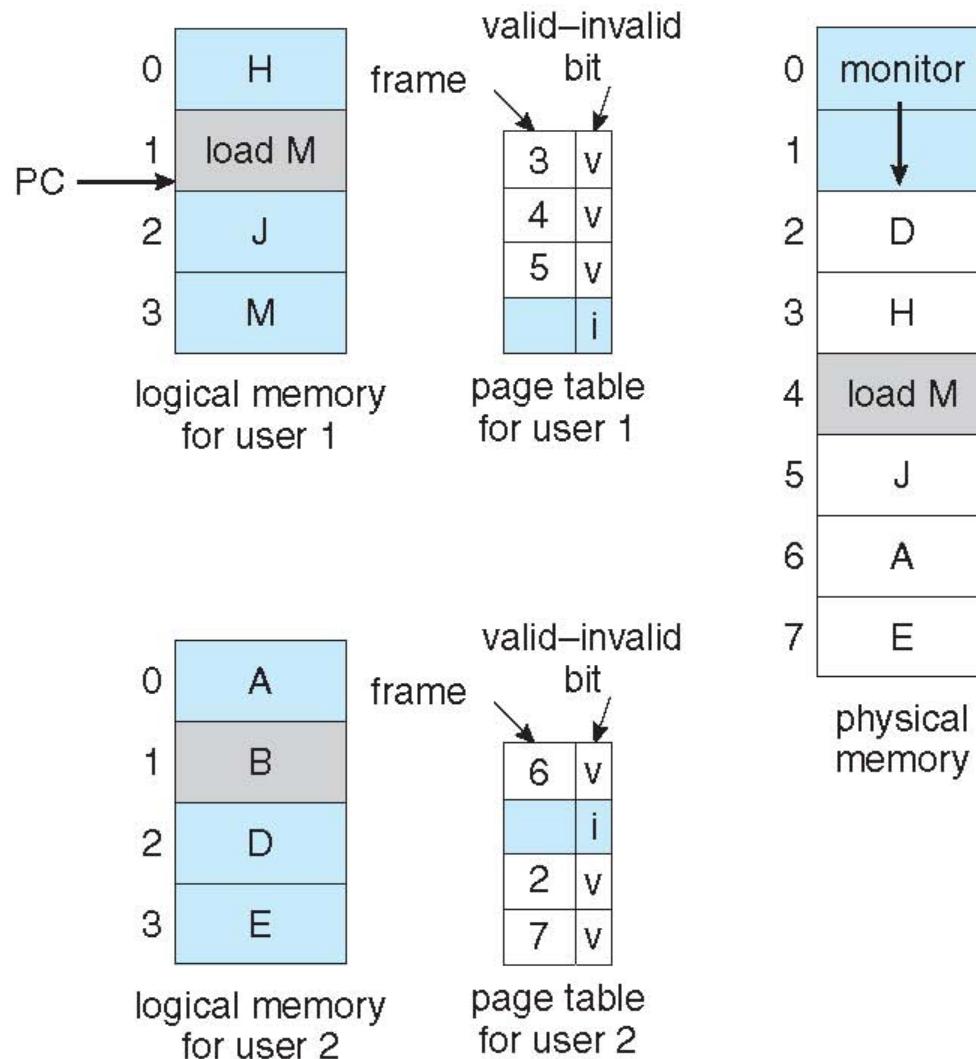


## ■ Stages in Demand Paging (worse case)

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
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
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



# Need For Page Replacement





# Priority Allocation

- Apply a *proportional allocation scheme* using priorities rather than size
- If 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



- Consider the following free-frame-list consisting of 6 frames

free-frame list



modified frame list



- Assume that this number of free frames falls below a certain threshold that triggers page replacement. The replacement algorithm (say, an LRU approximation algorithm) selects four frames - **15**, **3**, **35**, and **26** - to place on the free-frame list. It first places these frames on a modified-frame list. Typically, the modified-frame list would next be written to swap space, making the frames available to the free-frame list. An alternative strategy is to *compress a number of frames*, say three, and *store their compressed versions in a single page frame*.





# Memory Compression (Cont.)

- An alternative to paging is *memory compression*
- Rather than paging out modified frames to swap space, we *compress several frames into a single frame*, enabling the system to reduce memory usage without resorting to swapping pages

free-frame list



modified frame list



compressed frame list



# Summary

- *Virtual memory* abstracts physical memory into an extremely large uniform array of storage.
- The *benefits* of virtual memory include the following: (1) a program can be larger than physical memory, (2) a program does not need to be entirely in memory, (3) processes can share memory, and (4) processes can be created more efficiently.
- *Demand paging* is a technique whereby pages are loaded only when they are demanded during program execution. Pages that are never demanded are thus never loaded into memory.
- A *page fault* occurs when a page that is currently not in memory is accessed. The page must be brought from the backing store into an available page frame in memory.





## Summary (Cont.)

- *Copy-on-write* allows a child process to share the same address space as its parent. If either the child or the parent process writes (modifies) a page, a copy of the page is made.
- When available memory runs low, a *page-replacement algorithm* selects an existing page in memory to replace with a new page. Page-replacement algorithms include FIFO, optimal, and LRU. Pure LRU algorithms are impractical to implement, and most systems instead use LRU-approximation algorithms.
- *Global page-replacement algorithms* select a page from any process in the system for replacement, while *local page-replacement algorithms* select a page from the faulting process.
- *Thrashing* occurs when a system spends more time paging than executing.



## Summary (Cont.)

- A *locality* represents a set of pages that are actively used together. As a process executes, it moves from locality to locality. A *working set* is based on locality and is defined as the set of pages currently in use by a process.
- *Memory compression* is a memory-management technique that compresses a number of pages into a single page. Compressed memory is an alternative to paging and is used on mobile systems that do not support paging.
- *Kernel memory* is allocated differently than user-mode processes; it is allocated in contiguous chunks of varying sizes. Two common techniques for allocating kernel memory are (1) the buddy system and (2) slab allocation.





## Summary (Cont.)

- *TLB reach* refers to the amount of memory accessible from the TLB and is equal to the number of entries in the TLB multiplied by the page size. One technique for increasing TLB reach is to increase the size of pages.
- **Linux**, **Windows**, and **Solaris** manage virtual memory similarly, using demand paging and copy-on-write, among other features. Each system also uses a variation of LRU approximation known as the clock algorithm.



# End of Chapter 10



What Is an  
**OPERATING SYSTEM (OS)**  
and How Does It Work

CLEVERISM.COM

# Chapter 11: Mass-Storage Systems





# Chapter 11: Mass-Storage Systems

- Overview of Mass Storage Structure
- HDD Scheduling
- NVM Scheduling
- Error Detection and Correction
- Storage Device Management
- Swap-Space Management
- Storage Attachment
- RAID Structure





# Objectives

- Describe the *physical structure of secondary storage devices* and the effect of a device's structure on its uses
- Explain the *performance characteristics* of mass-storage devices
- Evaluate *I/O scheduling algorithms*
- Discuss *operating-system services* provided for mass storage, including **RAID**



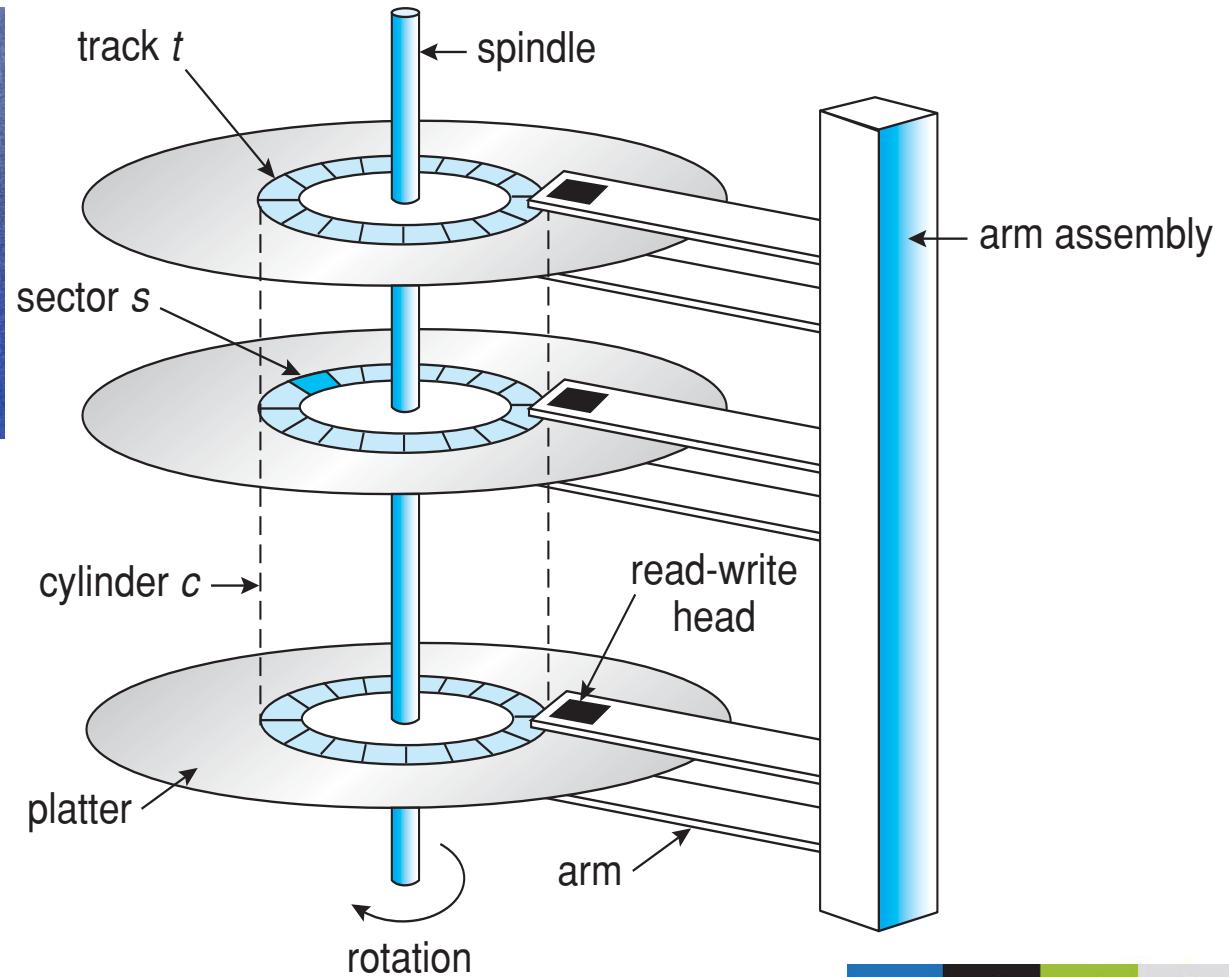


# Overview of Mass Storage Structure

- Bulk of secondary storage for modern computers is *hard disk drives* (**HDDs**) and *nonvolatile memory* (**NVM**) devices
- HDDs spin *platters* of magnetically-coated material under moving read-write *heads*
  - Disks can be removable
  - *Head crash* results from disk head making contact with the disk surface
    - ▶ That's bad
- Performance
  - *Drive rotation* is at 60 to 250 times per second
  - *Transfer rate* is rate at which data flow between drive and computer
  - *Positioning time* (random-access time) is time to move disk arm to desired cylinder (*seek time*) and time for desired sector to rotate under the disk head (*rotational latency*)



# Moving-head Disk Mechanism





# Example of Hard Disk Drives

- *Platters* range from .85 " to 14 "(historically)
  - Commonly 3.5 ", 2.5 ", and 1.8 "
- Range from *30GB* to *3TB* per drive
- Performance
  - *Transfer Rate* – theoretical – *6 Gb/sec*
    - ▶ *Effective Transfer Rate* – real – *1Gb/sec*
  - *Seek time* from *3ms* to *12ms* (e.g., *9ms* is common for desktop drives)
    - ▶ *Average seek time* is measured or calculated based on 1/3 of tracks
  - *Latency* based on spindle speed
    - ▶  $1 / (RPM / 60) = 60 / RPM$
    - ▶ *Average latency* =  $\frac{1}{2}$  latency





# Hard Disk Performance

- *Access Latency = Average access time = average seek time + average rotational latency*
  - For fastest disk:  $3ms + 2ms = 5ms$
  - For slow disk:  $9ms + 5.56ms = 14.56ms$
- *Average I/O time = average access time + (amount to transfer / transfer rate) + controller overhead*
- For example, to transfer a **4KB** block on a **7200 RPM** disk with a **5ms** average seek time, **1Gb/sec** transfer rate with a **0.1ms** controller overhead, the *average I/O time for 4KB block* is
  - $= 5ms + 4.17ms + 0.1ms + \text{transfer time}$ 
    - ▶ Transfer time =  $4KB / 1Gb/s * 8Gb / GB * 1GB / 10242KB = 32 / (10242) = 0.031 ms$
  - $= 9.27ms + .031ms = 9.301ms$



# The First Commercial Disk Drive



- 1956
- IBM RAMDAC computer included the IBM Model 350 disk storage system
- 5M (7 bit) characters
- 50 x 24" platters
- Access time < 1 second



# Nonvolatile Memory Devices

- If disk-drive like, then called *solid-state disks (SSDs)*
- Other forms include *USB drives* (thumb drive, flash drive), *DRAM disk* replacements, surface-mounted on motherboards, and main storage in devices like smartphones
  - Busses can be too slow → connect directly to PCIe for example
  - No moving parts, so no seek time or rotational latency
  - Can be more reliable than HDDs
  - More expensive per MB
  - Maybe have shorter life span – need careful management
  - Less capacity
  - But much faster





# Nonvolatile Memory Devices (Cont.)

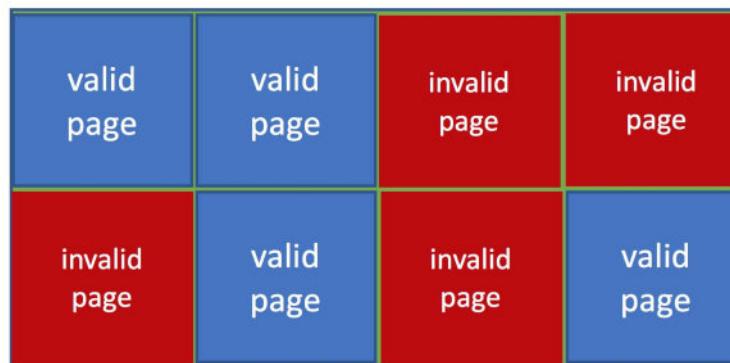
- Have characteristics that present challenges
- Read and written in “page” increments (think sector) but can’t overwrite in place
  - Must first be erased, and erases happen in larger ”block” increments
  - Can only be erased a limited number of times before worn out ~ 100,000
  - Life span measured in *drive writes per day (DWPD)*
    - ▶ E.g., A 1TB NAND drive with rating of 5DWPD is expected to have 5TB per day written within warranty period without failing





# NAND Flash Controller Algorithms

- With *no overwrite*, pages end up with *mix of valid and invalid data*
- To track which logical blocks are valid, controller maintains *flash translation layer (FTL)* table
- Also implements *garbage collection (GC)* to free invalid page space
- Allocates *overprovisioning* to provide working space for GC
- Each cell has lifespan, so *wear leveling* needed to write equally to all cells



NAND block with valid and invalid pages





# Volatile Memory

- **DRAM** frequently used as mass-storage device
  - Not technically secondary storage because volatile, but can have file systems, be used like very fast secondary storage
- **RAM drives** (with many names, including **RAM disks**) present as raw block devices, commonly file system formatted
  - Found in all major operating systems
    - ▶ Linux `/dev/ram`, macOS `diskutil` to create them, Solaris `/tmp` of file system type `tmpfs`
  - Computers have buffering, caching via RAM, so why RAM drives?
    - ▶ Caches / buffers allocated / managed by programmer, operating system, hardware
    - ▶ *RAM drives under user control*
- Used as high speed temporary storage
  - Programs could share bulk date, quickly, by reading/writing to RAM drive



# Magnetic Tape

**Magnetic tape** was used as an early secondary-storage medium. Although it is nonvolatile and can hold large quantities of data, its access time is slow compared with that of main memory and drives. In addition, random access to magnetic tape is about a thousand times slower than random access to HDDs and about a hundred thousand times slower than random access to SSDs so tapes are not very useful for secondary storage. Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another.

A tape is kept in a spool and is wound or rewound past a read-write head. Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can read and write data at speeds comparable to HDDs. Tape capacities vary greatly, depending on the particular kind of tape drive, with current capacities exceeding several terabytes. Some tapes have built-in compression that can more than double the effective storage. Tapes and their drivers are usually categorized by width, including 4, 8, and 19 millimeters and 1/4 and 1/2 inch. Some are named according to technology, such as LTO-6 (Figure 11.5) and SDLT.



**Figure 11.5** An LTO-6 Tape drive with tape cartridge inserted.



# Disk Attachment

- *Host-attached storage* accessed through I/O ports talking to I/O busses. Several busses available, including *advanced technology attachment (ATA)*, *serial ATA (SATA)*, *eSATA*, *serial attached SCSI (SAS)*, *universal serial bus (USB)*, and *fibre channel (FC)*
  - Because NVM much faster than HDD, new fast interface for NVM called *NVM express (NVMe)*, connecting directly to **PCI bus**
- Data transfers on a bus carried out by special electronic processors called *controllers* (or *host-bus adapters, HBAs*)
  - Host controller on the computer end of the bus, device controller on device end
  - Computer places command on host controller, using memory-mapped I/O ports
  - Host controller sends messages to device controller
  - Data transferred via DMA between device and computer DRAM





# Address Mapping

- *Disk drives* are addressed as *large 1-dimensional arrays of logical blocks*, where the logical block is the smallest unit of transfer
  - *Low-level formatting* creates logical blocks on physical media
- The 1-dimensional array of logical blocks is *mapped* into the sectors of the disk sequentially
  - Sector 0 is the first sector of the first track on the outermost cylinder
  - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost
  - Logical to physical address should be easy
    - ▶ Except for bad sectors
    - ▶ Non-constant # of sectors per track via constant angular velocity





# HDD Scheduling

- The operating system is responsible for using hardware efficiently – for the disk drives, this means having a *fast access time* and *disk bandwidth*
  - Seek time  $\approx$  seek distance
  - Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer
- *Minimize seek time*





# Disk Scheduling (Cont.)

- There are many sources of *disk I/O request*
  - OS
  - System processes
  - Users processes
- I/O request includes input or output mode, disk address, memory address, number of sectors to transfer
- OS maintains queue of requests, per disk or device
- Idle disk can immediately work on I/O request, busy disk means work must queue
  - *Optimization algorithms* only make sense when a queue exists
  - In the past, operating system responsible for queue management, disk drive head scheduling
    - ▶ Now, built into the storage devices, controllers
  - Just provide **LBA**s, handle sorting of requests





## Disk Scheduling (Cont.)

- Note that drive controllers have small buffers and can manage a queue of I/O requests (of varying “depth”)
- *Several algorithms exist to schedule the servicing of disk I/O requests*
- The analysis is true for one or many platters
- E.g., We illustrate scheduling algorithms with a request queue (0-199)
  - **98, 183, 37, 122, 14, 124, 65, 67**
  - Head pointer → 53

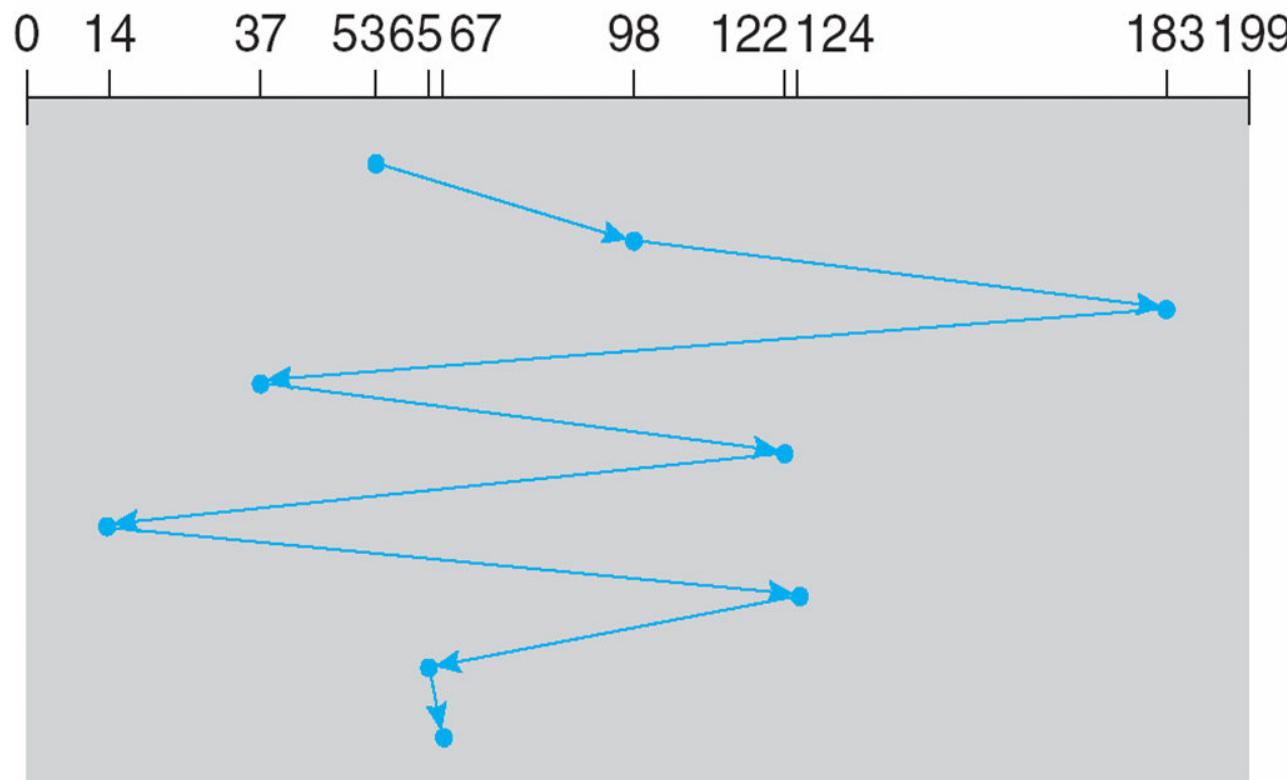


# First Come First Served (FCFS)

- Illustration shows total head movement of *640* cylinders

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53





# SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- SCAN algorithm sometimes called the *elevator algorithm*
- But note that if requests are uniformly dense, largest density at other end of disk and those wait the longest

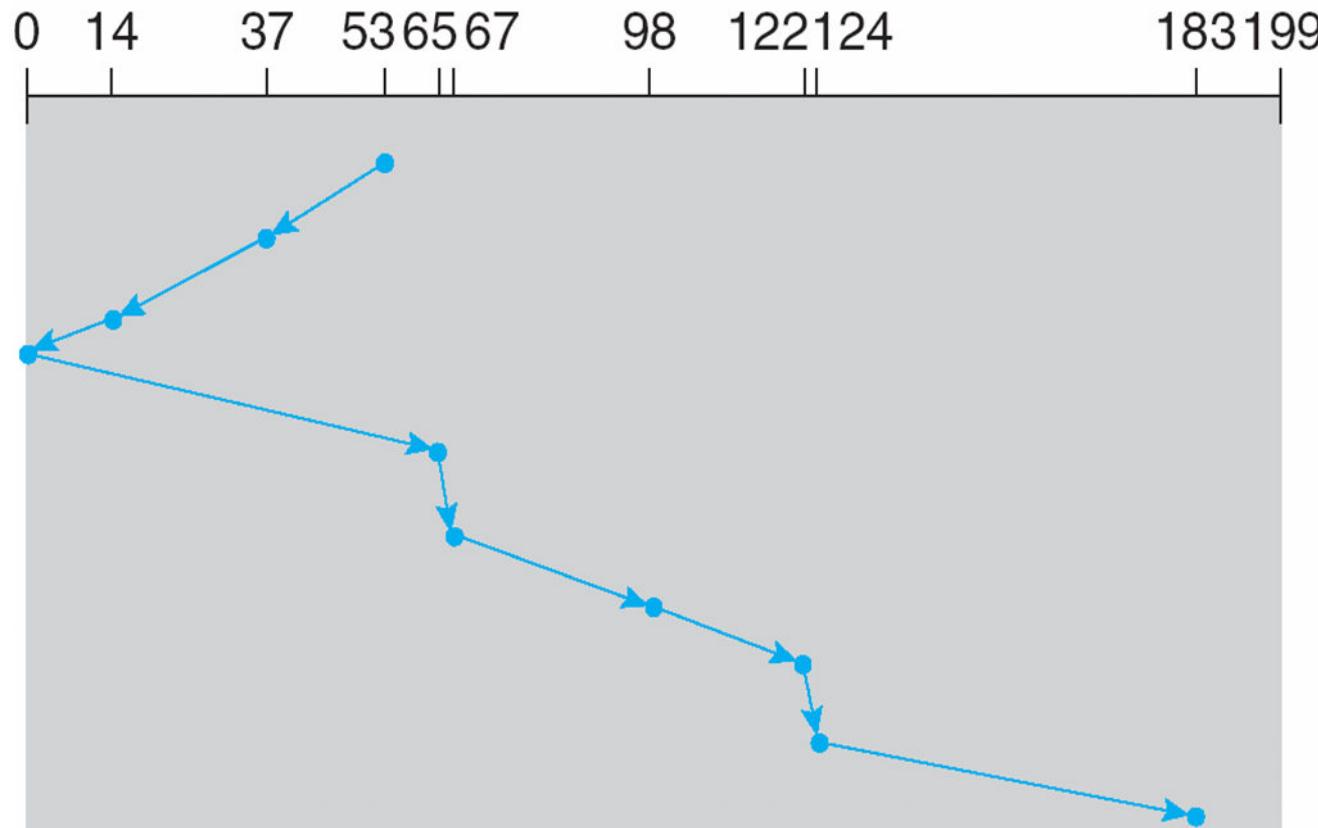


## SCAN (Cont.)

- Illustration shows total head movement of *208* cylinders

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53





## C-SCAN

- Provides a *more uniform wait time* than SCAN
- The head moves from one end of the disk to the other, servicing requests as it goes
  - When it reaches the other end, however, it immediately returns to the beginning of the disk, *without servicing any requests on the return trip*
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one

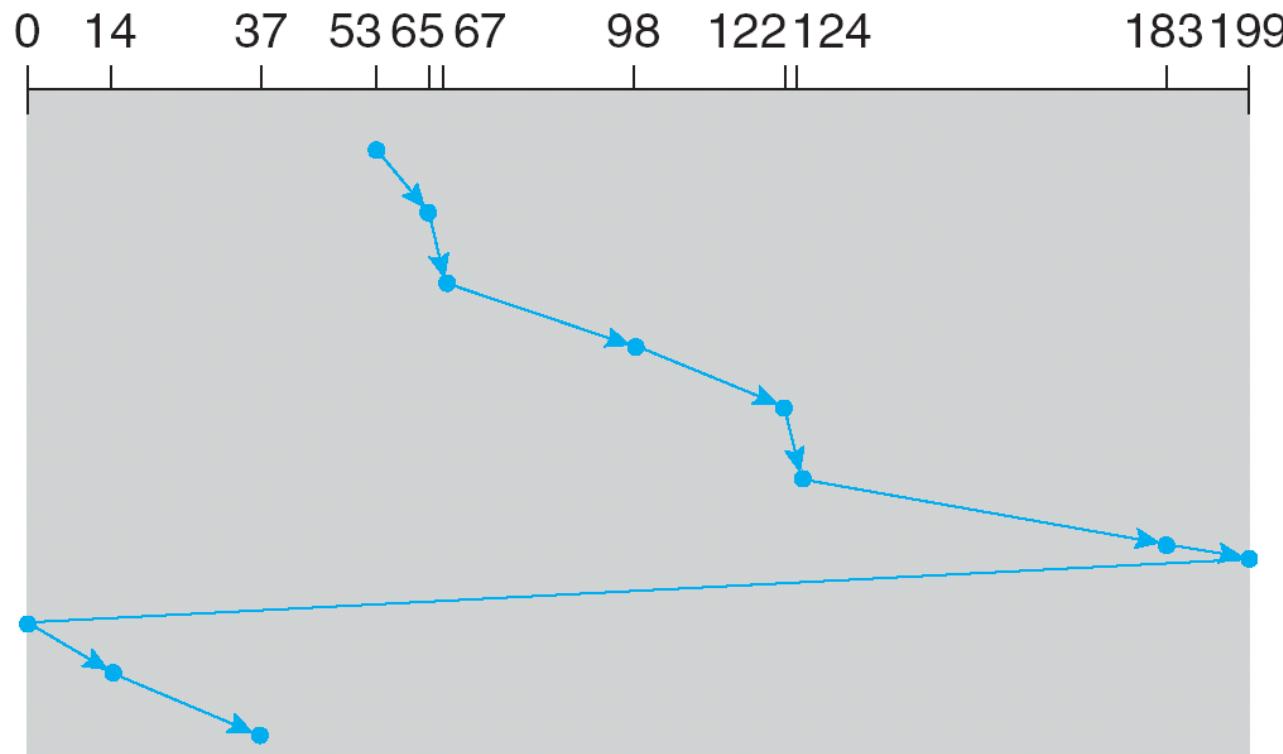


## C-SCAN (Cont.)

- Total number of cylinders?

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53





# Selecting a Disk-Scheduling Algorithm

- **FCFS** is common and has a natural appeal
- **SCAN** and **C-SCAN** perform better for systems that place a heavy load on the disk (less starvation, but still possible)
- To avoid starvation Linux implements *deadline scheduler*
  - Maintains separate read and write queues, gives read priority
    - ▶ Because processes more likely to block on read than write
  - Implements four queues: 2 x read and 2 x write
    - ▶ 1 read and 1 write queue sorted in LBA order, (implementing C-SCAN)
    - ▶ 1 read and 1 write queue sorted in FCFS order
    - ▶ All I/O requests sent in batch sorted in that queue's order
    - ▶ After each batch, checks if any requests in FCFS older than configured age (default 500ms)
      - If so, LBA queue containing that request is selected for next batch of I/O
  - In RHEL 7, **NOOP** and *completely fair queueing scheduler (CFQ)* are also available, defaults vary by storage device





# NVM Scheduling

- *No disk heads or rotational latency* but still room for optimization
- In RHEL 7, **NOOP** (no scheduling) is used but *adjacent LBA requests are combined*
  - NVM best at random I/O, HDD at sequential
  - Throughput can be similar
  - Input/Output operations per second (IOPS) much higher with NVM (hundreds of thousands vs hundreds)
  - But write amplification (one write, causing garbage collection and many read/writes) can decrease the performance advantage





# Error Detection and Correction

- Fundamental aspect of many parts of computing (memory, networking, storage)
- *Error detection* determines if there a problem has occurred (for example a bit flipping)
  - If detected, can halt the operation
  - Detection frequently done via parity bit
  - *Parity* – one form of checksum – uses modular arithmetic to compute, store, compare values of fixed-length words
  - Another error-detection method common in networking is *Cyclic Redundancy Check* (CRC) which uses hash function to detect multiple-bit errors
- *Error-correction code (ECC)* not only detects, but can correct some errors
  - *Soft errors* correctable, *hard errors* detected but not corrected





# Storage Device Management

- *Low-level formatting*, or *physical formatting* — Dividing a disk into sectors that the disk controller can read and write
  - Each sector can hold header information, plus data, plus error correction code (ECC)
  - Usually 512 bytes of data but can be selectable
- To use a disk to hold files, the operating system still needs to record its own data structures on the disk
  - Partition the disk into one or more groups of cylinders, each treated as a logical disk
  - *Logical formatting* or “making a file system”
  - To increase efficiency most file systems group blocks into clusters
    - ▶ Disk I/O done in blocks
    - ▶ File I/O done in clusters





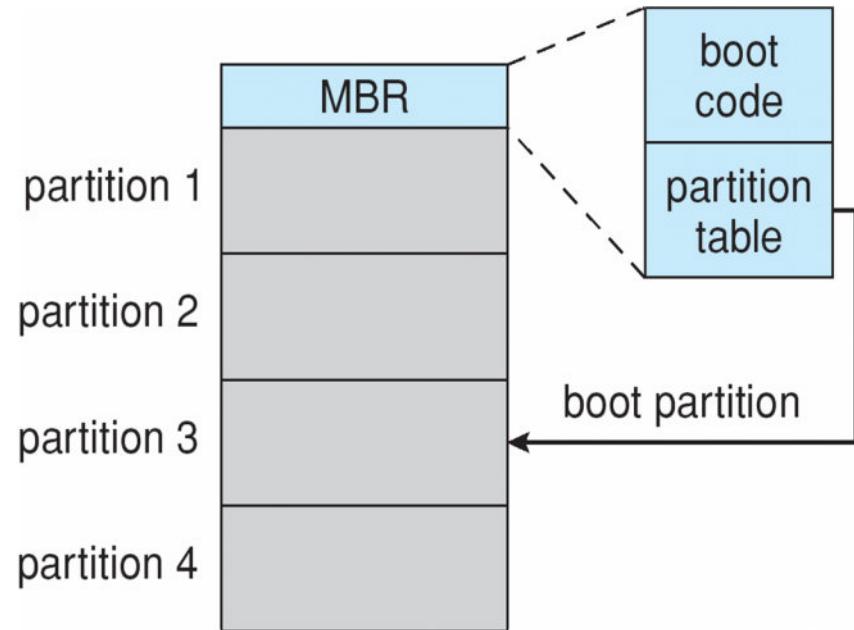
# Storage Device Management (cont.)

- *Root partition* contains the OS, other partitions can hold other Oses, other file systems, or be raw
  - Mounted at boot time
  - Other partitions can mount automatically or manually
- At mount time, file system consistency checked
  - Is all metadata correct?
    - ▶ If not, fix it, try again
    - ▶ If yes, add to mount table, allow access
- *Boot block* can point to *boot volume* or *boot loader set of blocks* that contain enough code to know how to load the kernel from the file system
  - Or a *boot management program* for multi-os booting



# Device Storage Management (Cont.)

- *Raw disk* access for apps that want to do their own block management, keep OS out of the way (databases for example)
- *Boot block* initializes system
  - The *bootstrap* is stored in ROM, firmware
  - *Bootstrap loader program* stored in boot blocks of boot partition
- Methods such as *sector sparing* used to handle bad blocks

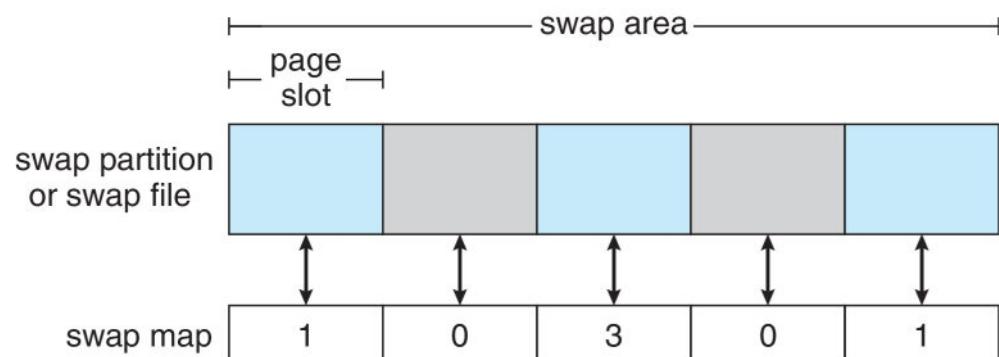


Booting from secondary storage in Windows



# Swap-Space Management

- Used for moving entire processes (swapping), or pages (paging), from DRAM to secondary storage when DRAM not large enough for all processes
- Operating system provides swap space management



- Secondary storage slower than DRAM, so important to optimize performance
- Usually multiple swap spaces possible – decreasing I/O load on any given device
- Best to have dedicated devices
- Can be in raw partition or a file within a file system (for convenience of adding)
- Data structures for swapping on Linux systems:





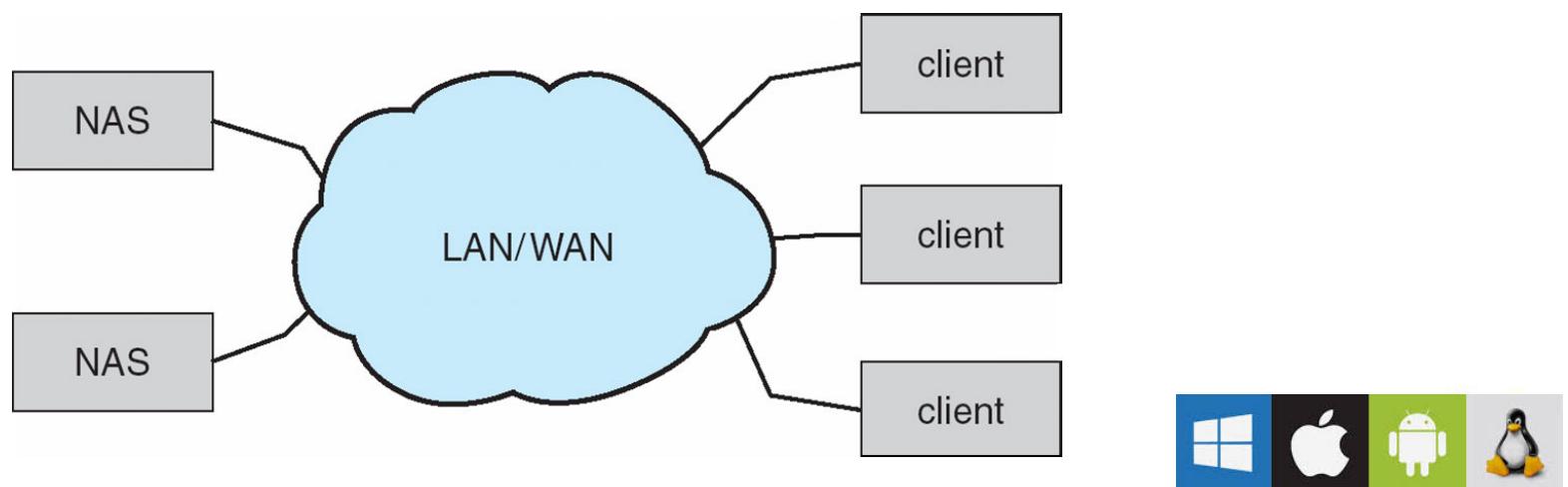
# Storage Attachment

- Computers access storage in three ways
  - host-attached
  - network-attached
  - cloud
- Host attached access through local I/O ports, using one of several technologies
  - To attach many devices, use storage busses such as USB, firewire, thunderbolt
  - High-end systems use *fibre channel* (**FC**)
    - ▶ High-speed serial architecture using fibre or copper cables
    - ▶ Multiple hosts and storage devices can connect to the FC fabric



# Network-Attached Storage

- *Network-attached storage (NAS)* is storage made available over a network rather than over a local connection (such as a bus)
  - Remotely attaching to file systems
- **NFS** and **CIFS** are common protocols
- Implemented via *remote procedure calls (RPCs)* between host and storage over typically TCP or UDP on IP network
- **iSCSI** protocol uses IP network to carry the SCSI protocol
  - Remotely attaching to devices (blocks)





# Cloud Storage

- Similar to NAS, provides access to storage across a network
  - Unlike NAS, *accessed over the Internet or a WAN to remote data center*
- NAS presented as just another file system, while cloud storage is API based, with programs using the APIs to provide access
  - Examples include Dropbox, Amazon S3, Microsoft OneDrive, Apple iCloud
  - Use APIs because of latency and failure scenarios (NAS protocols wouldn't work well)





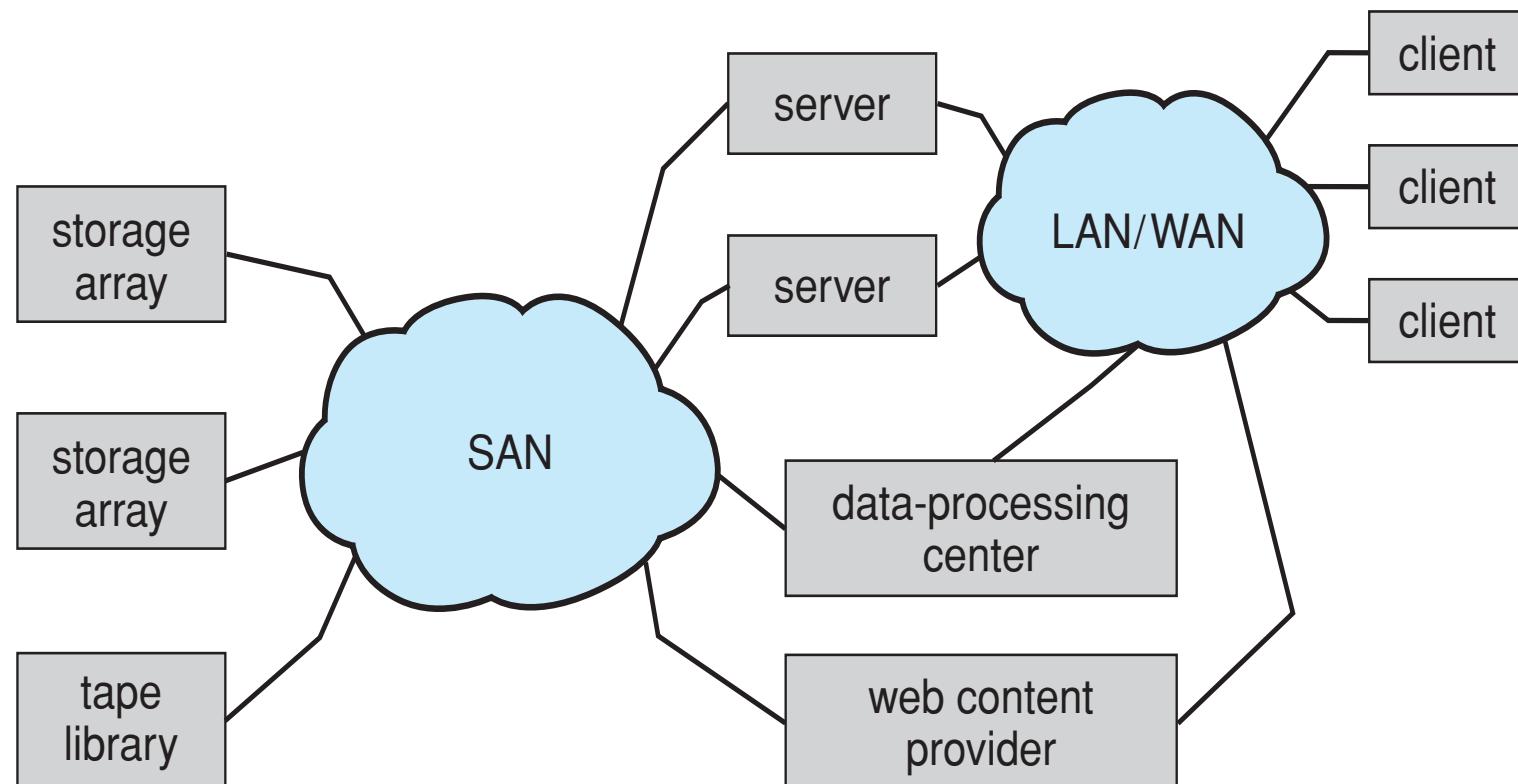
# Storage Array

- Can just attach disks, or arrays of disks
- Avoids the NAS drawback of using network bandwidth
- Storage Array has controller(s), provides features to attached host(s)
  - Ports to connect hosts to array
  - Memory, controlling software (sometimes NVRAM, etc)
  - A few to thousands of disks
  - RAID, hot spares, hot swap (discussed later)
  - Shared storage -> more efficiency
  - Features found in some file systems
    - ▶ Snapshots, clones, thin provisioning, replication, deduplication, etc



# Storage Area Network

- Common in large storage environments
- Multiple hosts attached to multiple storage arrays – flexible



# Storage Area Network (Cont.)

- SAN is one or more storage arrays
  - Connected to one or more Fibre Channel switches or InfiniBand (IB) network
- Hosts also attach to the switches
- Storage made available via LUN Masking from specific arrays to specific servers
- Easy to add or remove storage, add new host and allocate it storage
- Why have separate storage networks and communications networks?
  - Consider iSCSI, FCOE





# RAID Structure

- RAID – redundant array of inexpensive disks
  - multiple disk drives provides reliability via redundancy
- Increases the mean time to failure
- Mean time to repair – exposure time when another failure could cause data loss
- Mean time to data loss based on above factors
- If mirrored disks fail independently, consider disk with 1300,000 mean time to failure and 10 hour mean time to repair
  - Mean time to data loss is  $100,000^2 / (2 * 10) = 500 * 106$  hours, or **57,000 years!**
- Frequently combined with NVRAM to improve write performance
- Several improvements in disk-use techniques involve the use of multiple disks working cooperatively





## RAID (Cont.)

- Disk striping uses a group of disks as one storage unit
- RAID is arranged into six different levels
- RAID schemes improve performance and improve the reliability of the storage system by storing redundant data
  - Mirroring or shadowing (RAID 1) keeps duplicate of each disk
  - Striped mirrors (RAID 1+0) or mirrored stripes (RAID 0+1) provides high performance and high reliability
  - Block interleaved parity (RAID 4, 5, 6) uses much less redundancy
- RAID within a storage array can still fail if the array fails, so automatic replication of the data between arrays is common
- Frequently, a small number of hot-spare disks are left unallocated, automatically replacing a failed disk and having data rebuilt onto them



# RAID Levels



(a) RAID 0: non-redundant striping.



(b) RAID 1: mirrored disks.



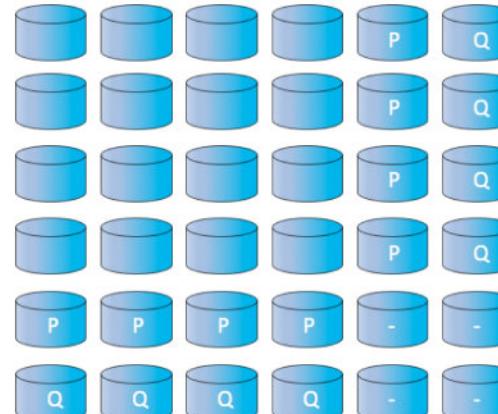
(c) RAID 4: block-interleaved parity.



(d) RAID 5: block-interleaved distributed parity.



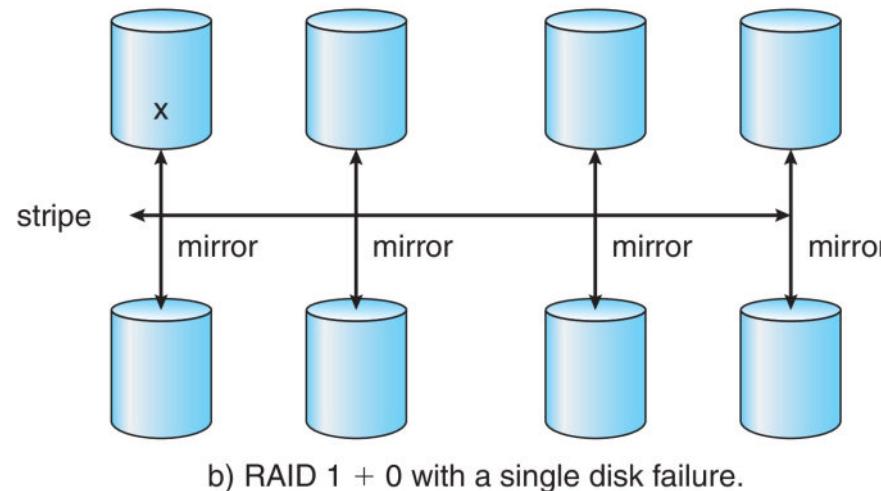
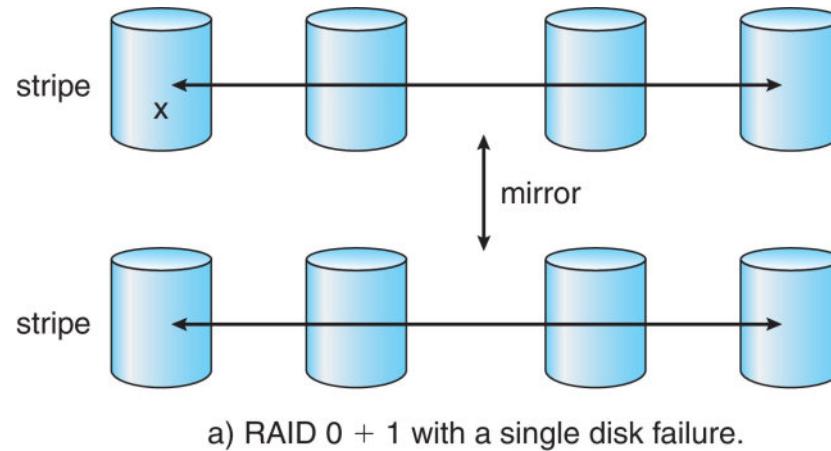
(e) RAID 6: P + Q redundancy.



(f) Multidimensional RAID 6.



# RAID (0 + 1) and (1 + 0)





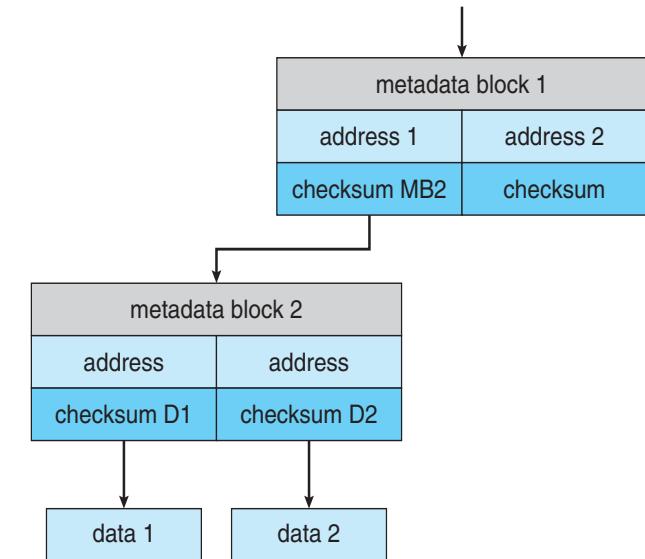
## Other Features

- Regardless of where RAID implemented, other useful features can be added
- Snapshot is a view of file system before a set of changes take place (i.e. at a point in time)
  - More in Ch 12
- Replication is automatic duplication of writes between separate sites
  - For redundancy and disaster recovery
  - Can be synchronous or asynchronous
- Hot spare disk is unused, automatically used by RAID production if a disk fails to replace the failed disk and rebuild the RAID set if possible
  - Decreases mean time to repair



# Extensions

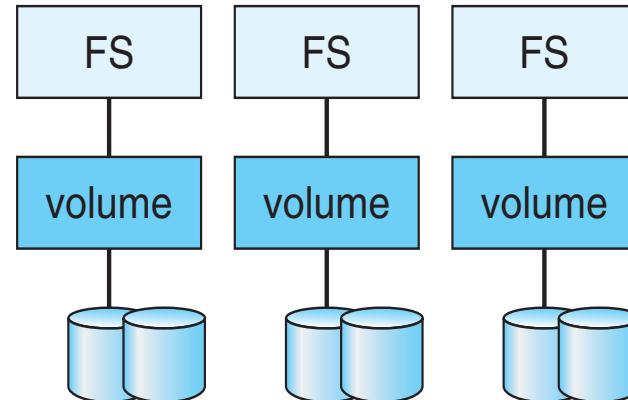
- RAID alone does not prevent or detect data corruption or other errors, just disk failures
- Solaris ZFS adds checksums of all data and metadata
- Checksums kept with pointer to object, to detect if object is the right one and whether it changed
- Can detect and correct data and metadata corruption
- ZFS also removes volumes, partitions
  - Disks allocated in pools
  - Filesystems with a pool share that pool, use and release space like malloc() and free() memory allocate / release calls



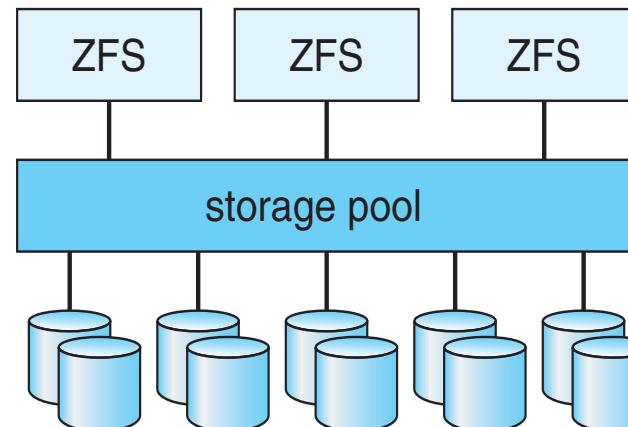
ZFS checksums all metadata and data



# Traditional and Pooled Storage



(a) Traditional volumes and file systems.



(b) ZFS and pooled storage.



# Object Storage

- General-purpose computing, file systems not sufficient for very large scale
  - Another approach – start with a storage pool and place objects in it
    - Object just a container of data
    - No way to navigate the pool to find objects (no directory structures, few services)
    - Computer-oriented, not user-oriented
  - Typical sequence
    - Create an object within the pool, receive an object ID
  - Access object via that ID
  - Delete object via that ID
- Object storage management software like *Hadoop file system (HDFS)* and **Ceph** determine where to store objects, manages protection
    - Typically by storing N copies, across N systems, in the object storage cluster
    - Horizontally scalable
    - Content addressable, unstructured





# Summary

- Hard disk drives and nonvolatile memory devices are the major secondary storage I/O units on most computers. Modern secondary storage is structured as large one-dimensional arrays of logical blocks.
- Drives of either type may be attached to a computer system in one of three ways: (1) through the local I/O ports on the host computer, (2) directly connected to motherboards, or (3) through a communications network or storage network connection.
- Requests for secondary storage I/O are generated by the file system and by the virtual memory system. Each request specifies the address on the device to be referenced in the form of a logical block number.
- Disk-scheduling algorithms can improve the effective bandwidth of HDDs, the average response time, and the variance in response time. Algorithms such as SCAN and C-SCAN are designed to make such improvements through strategies for disk-queue ordering. Performance of disks scheduling algorithms can vary greatly on hard disks. In contrast, because solid-state disks have no moving parts, performance varies little among scheduling algorithms, and quite often a simple FCFS strategy is used.





## Summary (Cont.)

- Data storage and transmission are complex and frequently result in errors. Error detection attempts to spot such problems to alert the system for corrective action and to avoid error propagation. Error correction can detect and repair problems, depending on the amount of correction data available and the amount of data that was corrupted.
- Storage devices are partitioned into one or more chunks of space. Each partition can hold a volume or be part of a multidevice volume. File systems are created in volumes.
- The operating system manages the storage device's blocks. New devices typically come pre-formatted. The device is partitioned, file systems are created, and boot blocks are allocated to store the system's bootstrap program if the device will contain an operating system. Finally, when a block or page is corrupted, the system must have a way to lock out that block or to replace it logically with a spare.





## Summary (Cont.)

- An efficient swap space is a key to good performance in some systems. Some systems dedicate a raw partition to swap space, and others use a file within the file system instead. Still other systems allow the user or system administrator to make the decision by providing both options.
- Because of the amount of storage required on large systems, and because storage devices fail in various ways, secondary storage devices are frequently made redundant via RAID algorithms. These algorithms allow more than one drive to be used for a given operation and allow continued operation and even automatic recovery in the face of a drive failure. RAID algorithms are organized into different levels; each level provides some combination of reliability and high transfer rates.
- Object storage is used for big data problems such as indexing the Internet and cloud photo storage. Objects are self-defining collections of data, addressed by object ID rather than file name. Typically it uses replication for data protection, computes based on the data on systems where a copy of the data exists, and is horizontally scalable for vast capacity and easy expansion.



# End of Chapter 11



What Is an  
**OPERATING SYSTEM (OS)**  
and How Does It Work

CLEVERISM.COM

# Chapter 12: I/O Systems



What Is an  
**OPERATING SYSTEM (OS)**  
and How Does It Work

CLEVERISM.COM



# Chapter 12: I/O Systems

- Overview
- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Transforming I/O Requests to Hardware Operations
- STREAMS
- Performance





# Objectives

- Explore the structure of an operating system's *I/O subsystem*
- Discuss the principles and complexities of *I/O hardware*
- Explain the *performance aspects* of *I/O hardware and software*





# Overview

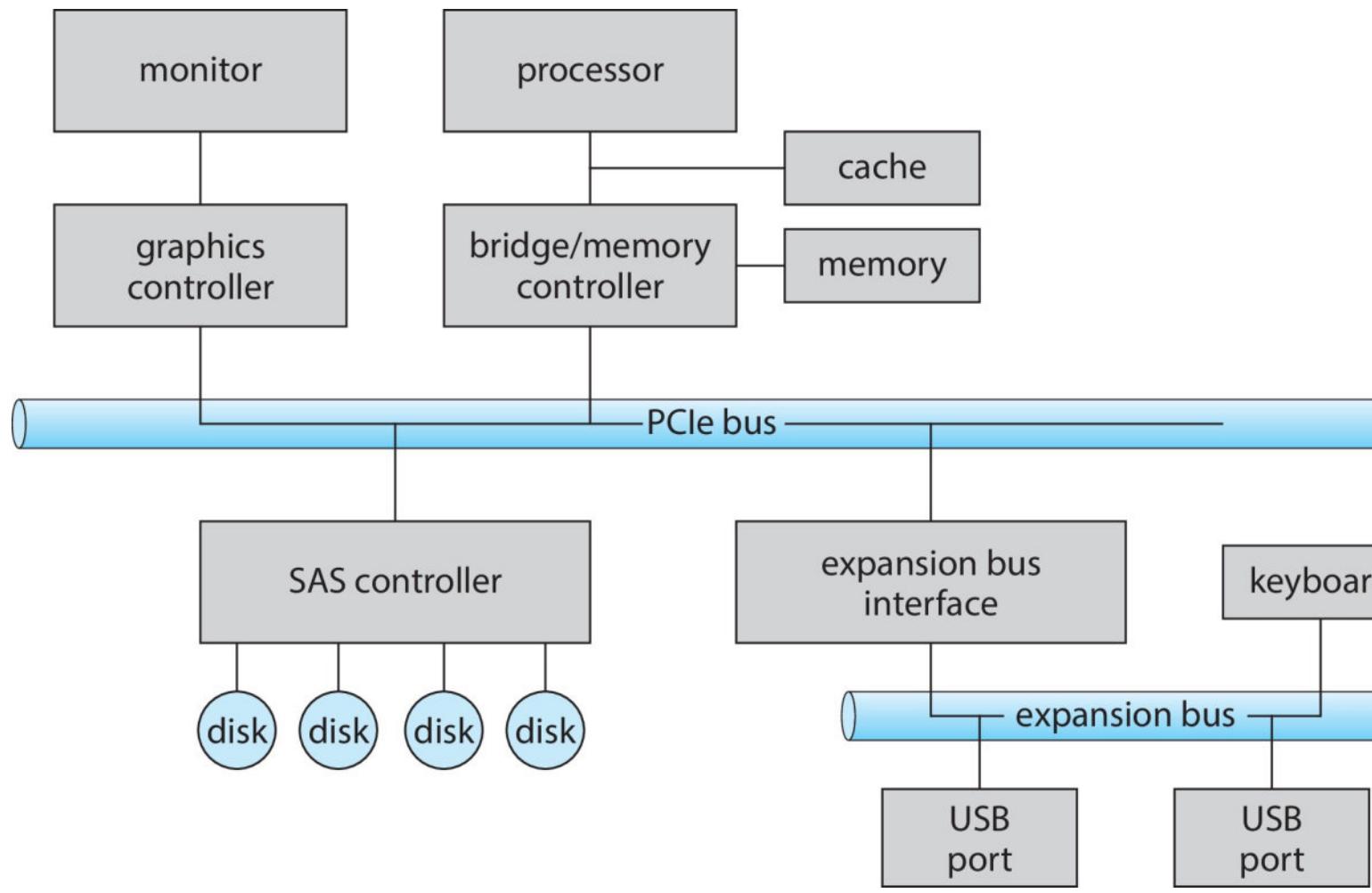
- *I/O management* is a major component of operating system design and operation
  - Important aspect of computer operation
  - I/O devices vary greatly
  - Various methods to control them
  - Performance management
  - New types of devices frequent
- *Ports, busses, device controllers* connect to various devices
- *Device drivers* encapsulate device details
  - Present uniform device-access interface to I/O subsystem



- Incredible variety of *I/O devices*
  - Storage
  - Transmission
  - Human-interface
- Common concepts – signals from I/O devices interface with computer
  - *Port* – connection point for device
  - *Bus* – daisy chain or shared direct access
    - ▶ *PCI bus* common in PCs and servers, e.g., *PCI Express* (PCIe)
    - ▶ *Expansion bus* connects relatively slow devices
    - ▶ *Serial-attached SCSI (SAS)* common disk interface
  - *Controller* (host adapter) – electronics that operate port, bus, device
    - ▶ Sometimes integrated, sometimes separate circuit board (host adapter)
    - ▶ Contains processor, microcode, private memory, bus controller, etc.
      - Some talk to per-device controller with bus controller, microcode,



# A Typical PC Bus Structure





## I/O Hardware (Cont.)

- *Fibre channel (FC)* is complex controller, usually separate circuit board (host-bus adapter, **HBA**) plugging into bus
- *I/O instructions* control devices
- Devices usually have *registers* where device driver places commands, addresses, and data to write, or read data from registers after command execution
  - *Data-in* register, *data-out* register, *status* register, *control* register
  - Typically, data registers are 1-4 bytes in size, and with FIFO buffer
- Devices have *addresses*, used by
  - *Direct I/O instructions*
  - *Memory-mapped I/O*
    - ▶ Device data and command registers mapped to processor address space
    - ▶ Especially for large memory-mapped region (e.g., graphics controller)





# Device I/O Port Locations on PCs (partial)

| I/O address range (hexadecimal) | device                    |
|---------------------------------|---------------------------|
| 000–00F                         | DMA controller            |
| 020–021                         | interrupt controller      |
| 040–043                         | timer                     |
| 200–20F                         | game controller           |
| 2F8–2FF                         | serial port (secondary)   |
| 320–32F                         | hard-disk controller      |
| 378–37F                         | parallel port             |
| 3D0–3DF                         | graphics controller       |
| 3F0–3F7                         | diskette-drive controller |
| 3F8–3FF                         | serial port (primary)     |





# Polling

- For each byte of I/O
  - Read busy bit from status register until 0
  - Host sets read or write bit and if write copies data into data-out register
  - Host sets command-ready bit
  - Controller sets busy bit, executes transfer
  - Controller clears busy bit, error bit, command-ready bit when transfer is done
- Step 1 is *busy-wait cycle* to wait for I/O from device
  - Reasonable if device is fast
  - But inefficient if device slow
  - CPU switches to other tasks?
    - ▶ But if miss a cycle, data will be overwritten / lost



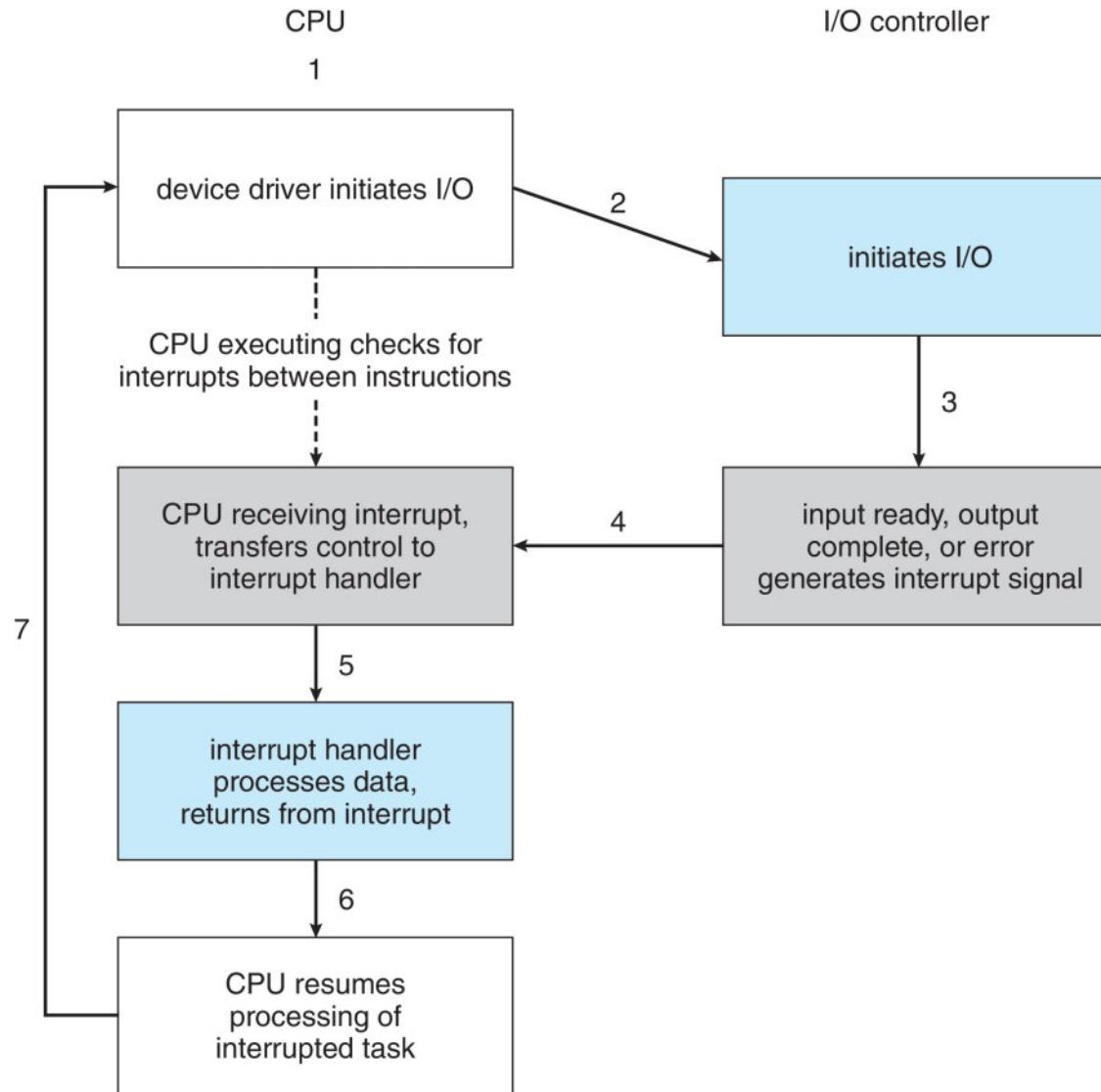


# Interrupts

- Polling can happen in *3 instruction cycles*
  - Read status, logical-and to extract status bit, branch if not zero
  - How to be more efficient if non-zero infrequently?
- *CPU Interrupt-request line* triggered by I/O device
  - Checked by CPU after each instruction
- *Interrupt handler* receives interrupts
  - Maskable to ignore or delay some interrupts
- *Interrupt vector* to dispatch interrupt to correct handler
  - Context switch at start and end
  - Based on priority
  - Some nonmaskable
  - Interrupt chaining if more than one device at same interrupt number



# Interrupt-Driven I/O Cycle





## Interrupts (Cont.)

- *Interrupt mechanism* also used for exceptions
  - Terminate process, crash system due to hardware error
- *Page fault* executes when memory access error
- *System call* executes via trap to trigger kernel to execute request
- *Multi-CPU systems* can process interrupts concurrently
  - If operating system designed to handle it
- Used for *time-sensitive processing, frequent*, must be *fast*



# Latency

- *Stressing* interrupt management because even single-user systems manage hundreds or interrupts per second and servers hundreds of thousands
- For example, a quiet **macOS** desktop generated 23,000 interrupts over 10 seconds

|                    | SCHEDULER | INTERRUPTS |
|--------------------|-----------|------------|
| total_samples      | 13        | 22998      |
| delays < 10 usecs  | 12        | 16243      |
| delays < 20 usecs  | 1         | 5312       |
| delays < 30 usecs  | 0         | 473        |
| delays < 40 usecs  | 0         | 590        |
| delays < 50 usecs  | 0         | 61         |
| delays < 60 usecs  | 0         | 317        |
| delays < 70 usecs  | 0         | 2          |
| delays < 80 usecs  | 0         | 0          |
| delays < 90 usecs  | 0         | 0          |
| delays < 100 usecs | 0         | 0          |
| total < 100 usecs  | 13        | 22998      |





# Intel Pentium Processor Event-Vector Table

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





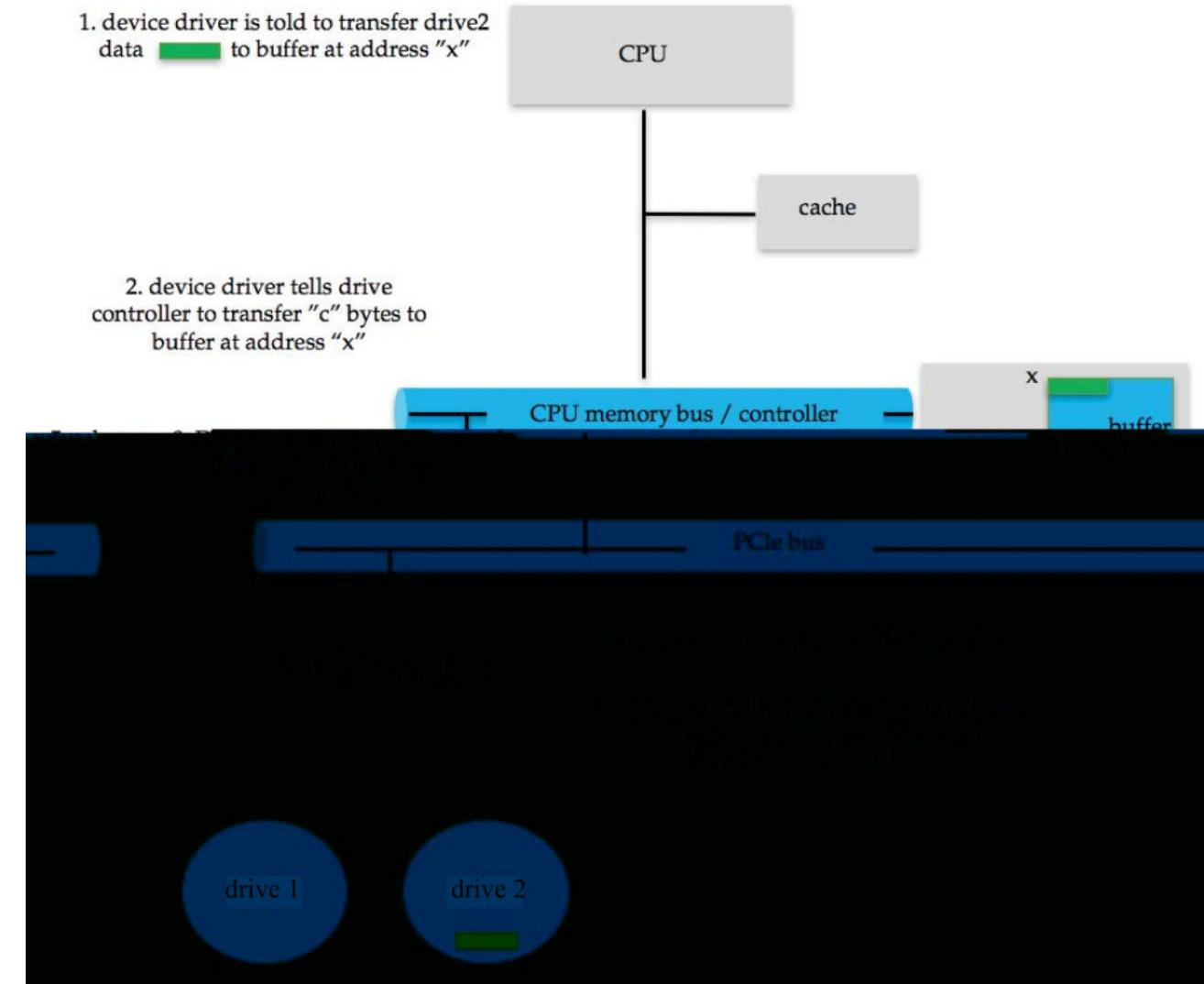
# Direct Memory Access (DMA)

- Used to avoid *programmed I/O* (1 byte at a time) for large data movement
- Requires *DMA controller*
  - Bypasses CPU to transfer data directly between I/O device and memory
- OS writes DMA command block into memory including
  - ▶ Source and destination addresses
  - ▶ Read or write mode
  - ▶ Count of bytes
  - Writes location of command block to DMA controller
    - ▶ Bus mastering of DMA controller – grabs bus from CPU
      - Cycle stealing from CPU but still much more efficient
    - ▶ When done, interrupts to signal completion
- Version that is aware of virtual addresses can be even more efficient
  - *Direct Virtual Memory Access (DVMA)*





# 6 Step Process to Perform DMA Transfer



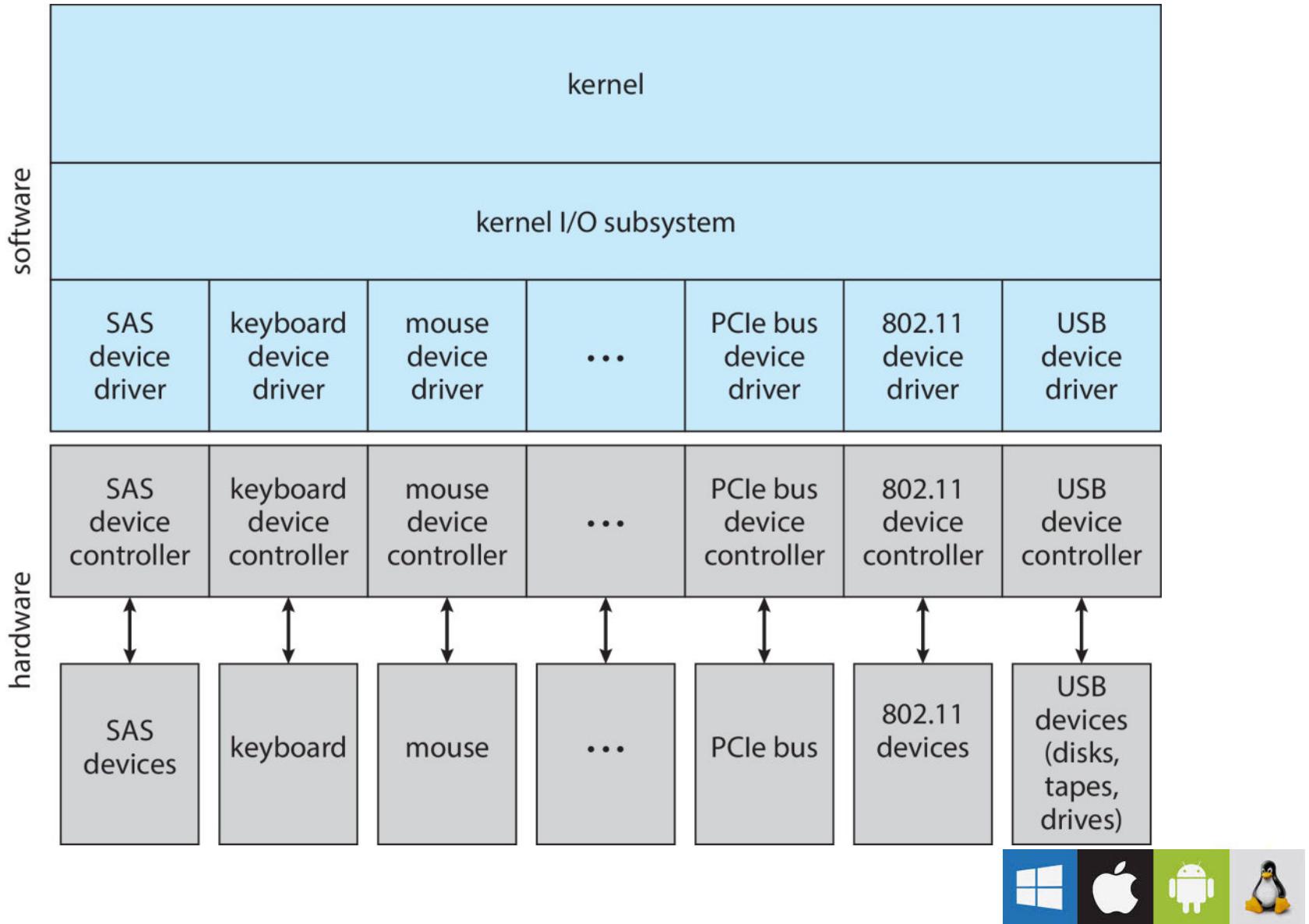


# Application I/O Interface

- *I/O system calls* encapsulate device behaviors in generic classes
- *Device-driver layer* hides differences among I/O controllers from kernel
- New devices talking *already-implemented protocols* need no extra work
- Each OS has its own *I/O subsystem structures* and *device driver frameworks*
- Devices vary in many dimensions
  - Character-stream or block
  - Sequential or random-access
  - Synchronous or asynchronous (or both)
  - Sharable or dedicated
  - Speed of operation
  - read-write, read only, or write only



# A Kernel I/O Structure



# Characteristics of I/O Devices

| aspect             | variation                                                         | example                               |
|--------------------|-------------------------------------------------------------------|---------------------------------------|
| data-transfer mode | character<br>block                                                | terminal<br>disk                      |
| access method      | sequential<br>random                                              | modem<br>CD-ROM                       |
| transfer schedule  | synchronous<br>asynchronous                                       | tape<br>keyboard                      |
| sharing            | dedicated<br>sharable                                             | tape<br>keyboard                      |
| device speed       | latency<br>seek time<br>transfer rate<br>delay between operations |                                       |
| I/O direction      | read only<br>write only<br>read-write                             | CD-ROM<br>graphics controller<br>disk |





## Characteristics of I/O Devices (Cont.)

- Subtleties of devices handled by device drivers
- Broadly I/O devices can be grouped by the OS into
  - Block I/O
  - Character I/O (Stream)
  - Memory-mapped file access
  - Network sockets
- For direct manipulation of I/O device specific characteristics, usually an escape / back door
  - Unix ioctl() call to send arbitrary bits to a device control register and data to device data register
- UNIX and Linux use tuple of “major” and “minor” device numbers to identify type and instance of devices (here major 8 and minors 0-4)  

```
% ls -l /dev/sda*
```

|                                                    |
|----------------------------------------------------|
| brw-rw---- 1 root disk 8, 0 Mar 16 09:18 /dev/sda  |
| brw-rw---- 1 root disk 8, 1 Mar 16 09:18 /dev/sda1 |
| brw-rw---- 1 root disk 8, 2 Mar 16 09:18 /dev/sda2 |
| brw-rw---- 1 root disk 8, 3 Mar 16 09:18 /dev/sda3 |





# Block and Character Devices

- *Block devices* include disk drives
  - Commands include read, write, seek
  - Raw I/O, direct I/O, or file-system access
  - Memory-mapped file access possible
    - ▶ File mapped to virtual memory and clusters brought via demand paging
  - DMA
- *Character devices* include keyboards, mice, serial ports
  - Commands include **get()**, **put()**
  - Libraries layered on top allow line editing





# Network Devices

- Varying enough from block and character to have own interface
- **Linux, Unix, Windows** and many others include *socket* interface
  - Separates network protocol from network operation
  - Includes **select()** functionality
- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)





# Clocks and Timers

- Provide current time, elapsed time, timer
- Normal resolution about 1/60 second
- Some systems provide higher-resolution timers
- Programmable interval timer used for timings, periodic interrupts
- **ioctl()** (on UNIX) covers odd aspects of I/O such as clocks and timers



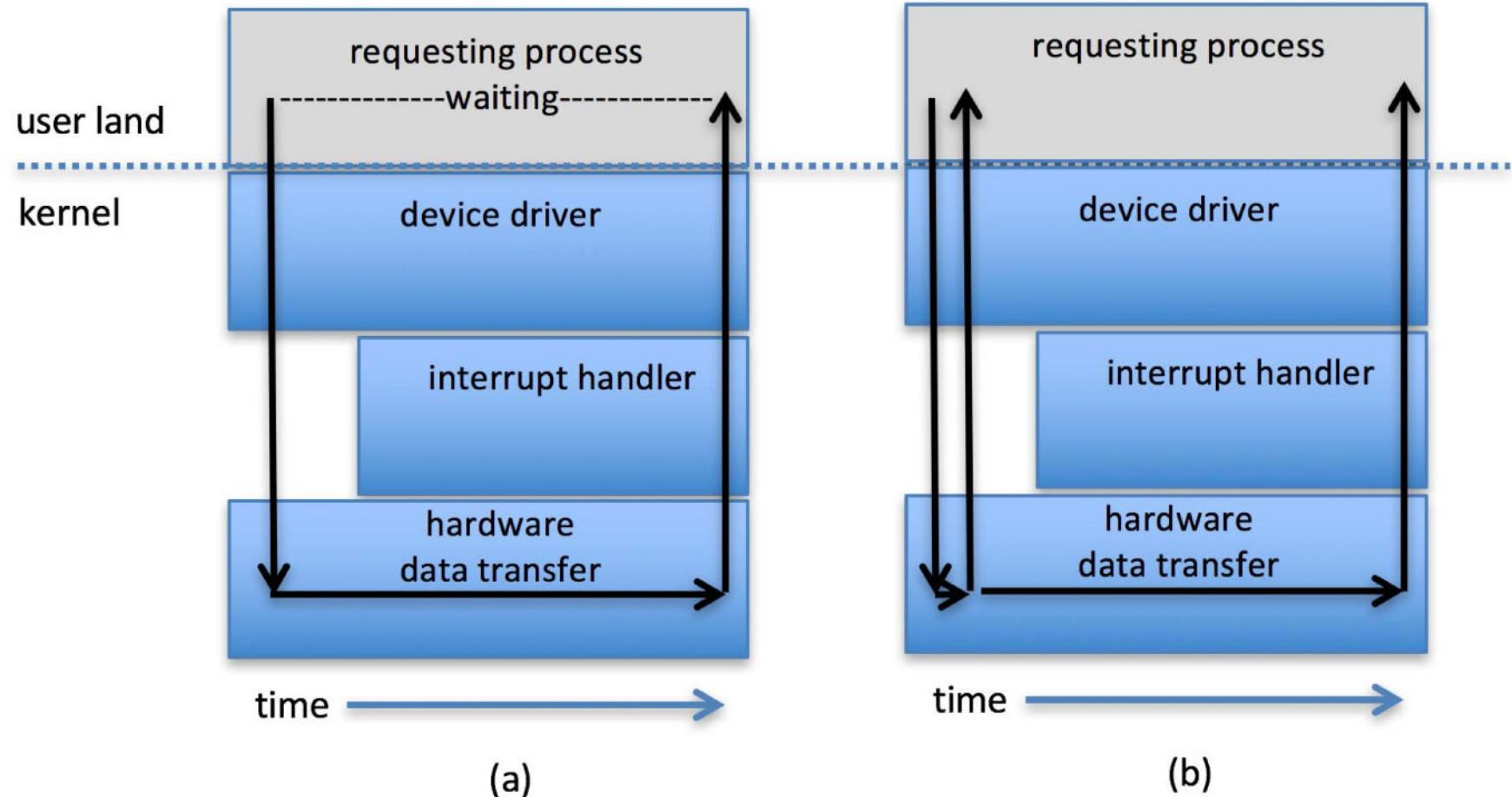


# Nonblocking and Asynchronous I/O

- *Blocking* – process suspended until I/O completed
  - Easy to use and understand
  - Insufficient for some needs
- *Nonblocking* – I/O call returns as much as available
  - User interface, data copy (buffered I/O)
  - Implemented via multi-threading
  - Returns quickly with count of bytes read or written
  - `select()` to find if data ready then `read()` or `write()` to transfer
- *Asynchronous* – process runs while I/O executes
  - Difficult to use
  - I/O subsystem signals process when I/O completed



# Two I/O Methods





## Vectored I/O

- *Vectored I/O* allows one system call to perform multiple I/O operations
- For example, Unix **readve()** accepts a vector of multiple buffers to read into or write from
- This *scatter-gather method* better than multiple individual I/O calls
  - Decreases context switching and system call overhead
  - Some versions provide atomicity
    - ▶ Avoid for example worry about multiple threads changing data as reads / writes occurring





# Kernel I/O Subsystem

## ■ Scheduling

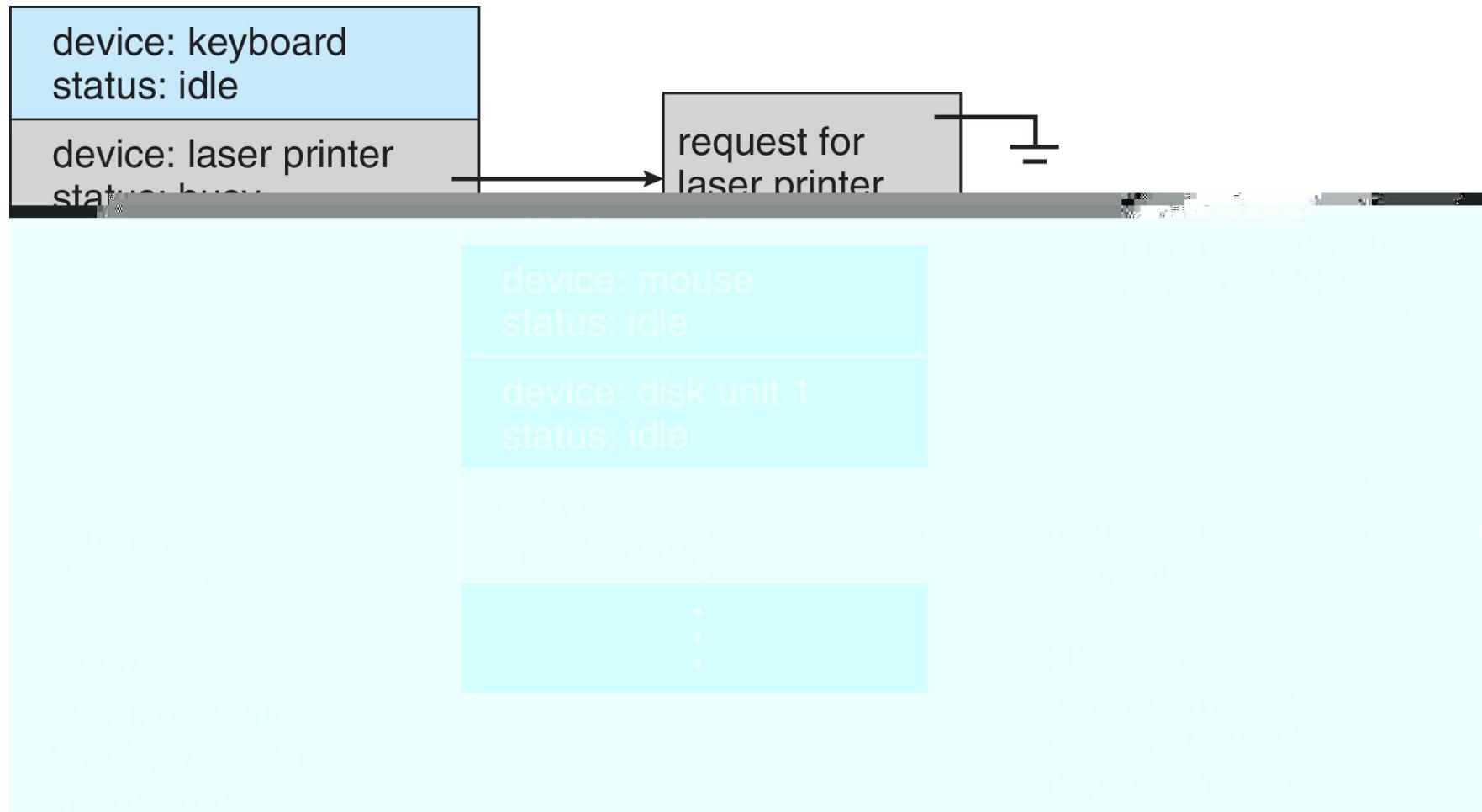
- Some I/O request ordering via per-device queue
- Some OSs try fairness
- Some implement Quality Of Service (i.e. IPQOS)

## ■ *Buffering* – store data in memory while transferring between devices

- To cope with device speed mismatch
- To cope with device transfer size mismatch
- To maintain “copy semantics”
- Double buffering – two copies of the data
  - ▶ Kernel and user
  - ▶ Varying sizes
  - ▶ Full / being processed and not-full / being used
  - ▶ Copy-on-write can be used for efficiency in some cases

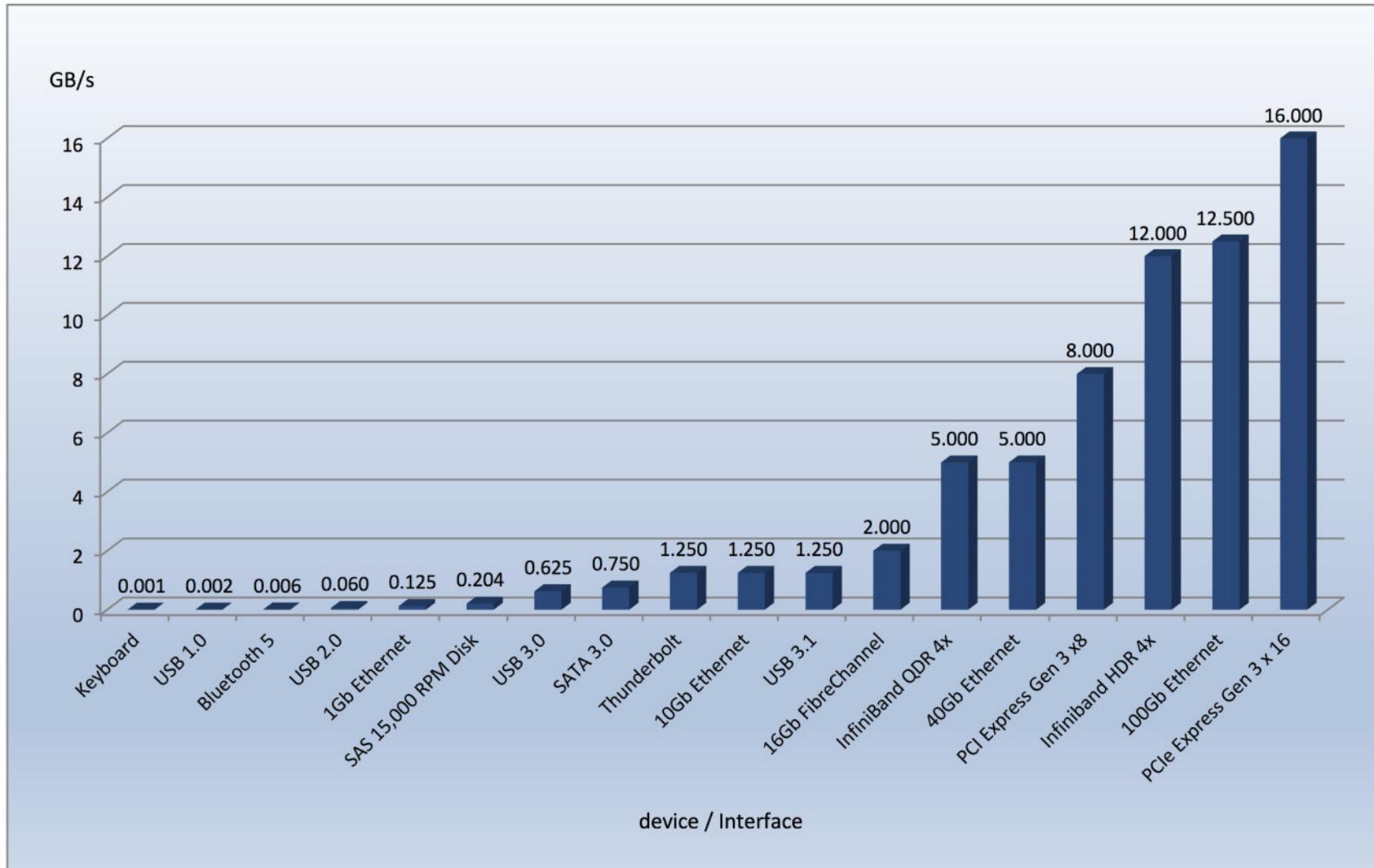


# Device-status Table





# Common PC and Data-center I/O devices and Interface Speeds





# Kernel I/O Subsystem

- *Caching* – faster device holding copy of data
  - Always just a copy
  - Key to performance
  - Sometimes combined with buffering
- *Spooling* – hold output for a device
  - If device can serve only one request at a time
  - i.e., Printing
- *Device reservation* – provides exclusive access to a device
  - System calls for allocation and de-allocation
  - Watch out for deadlock





# Error Handling

- OS can recover from disk read, device unavailable, transient write failures
  - Retry a read or write, for example
  - Some systems more advanced – Solaris FMA, AIX
    - ▶ Track error frequencies, stop using device with increasing frequency of retryable errors
- Most return an error number or code when I/O request fails
- System error logs hold problem reports



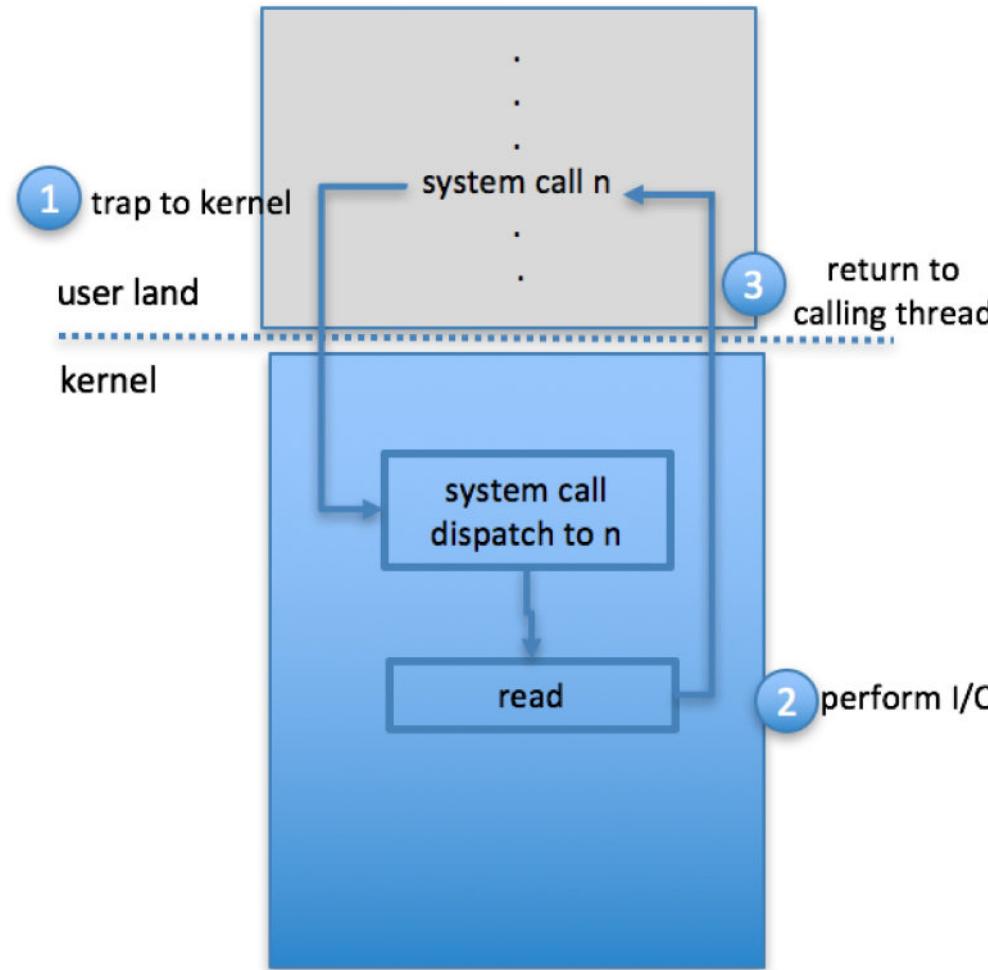


# I/O Protection

- User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions
  - All I/O instructions defined to be privileged
  - I/O must be performed via system calls
    - ▶ Memory-mapped and I/O port memory locations must be protected too



# Use of a System Call to Perform I/O



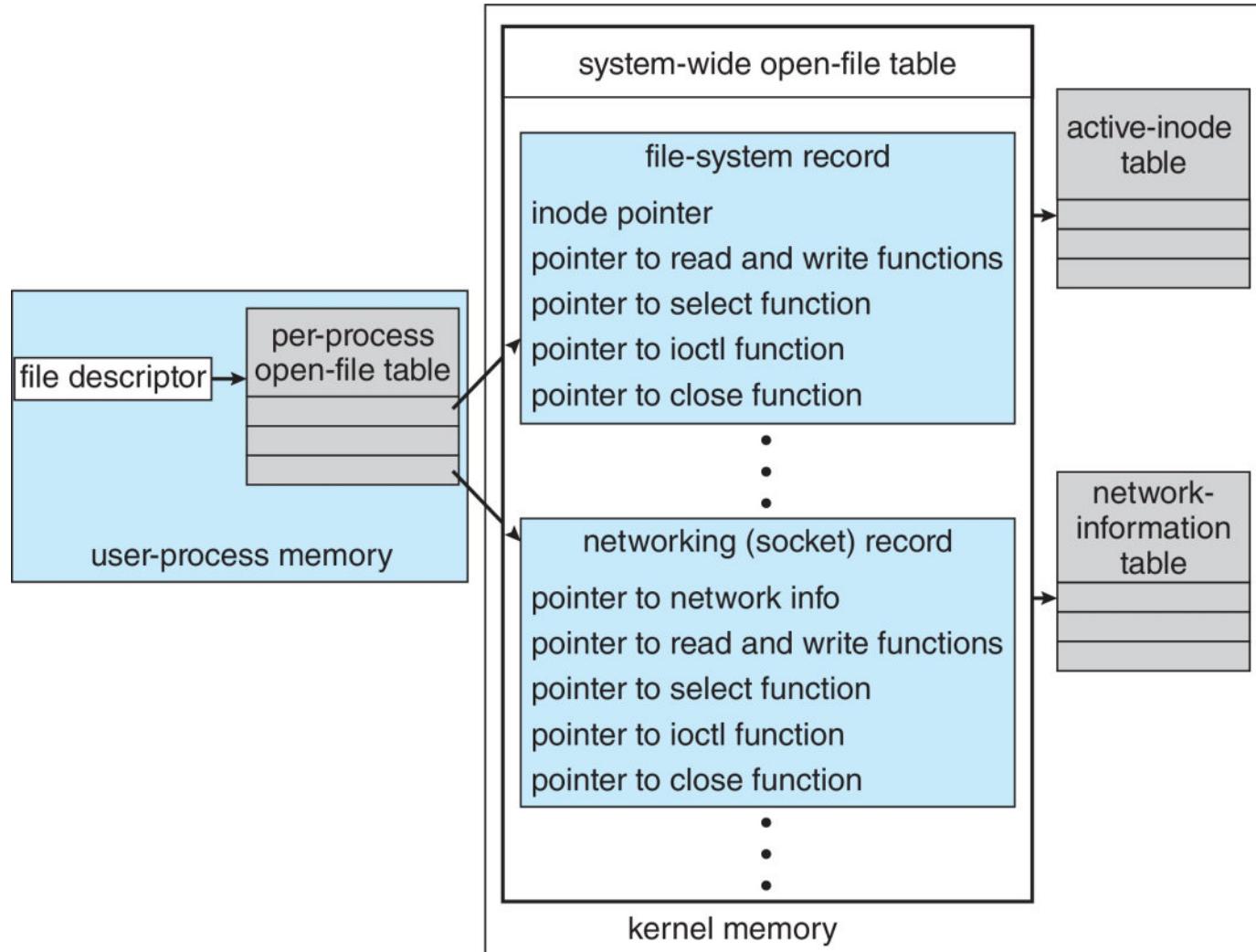


# Kernel Data Structures

- Kernel keeps state info for I/O components, including open file tables, network connections, character device state
- Many, many complex data structures to track buffers, memory allocation, “dirty” blocks
- Some use object-oriented methods and message passing to implement I/O
  - Windows uses message passing
    - ▶ Message with I/O information passed from user mode into kernel
    - ▶ Message modified as it flows through to device driver and back to process
    - ▶ Pros / cons?



# UNIX I/O Kernel Structure





# Power Management

- Not strictly domain of I/O, but much is I/O related
- Computers and devices use electricity, generate heat, frequently require cooling
- OSes can help manage and improve use
  - Cloud computing environments move virtual machines between servers
    - ▶ Can end up evacuating whole systems and shutting them down
- Mobile computing has power management as first class OS aspect



# Power Management (Cont.)

- Modern systems use advanced configuration and power interface (ACPI) firmware providing code that runs as routines called by kernel for device discovery, management, error and power management
- E.g., Android implements
  - Component-level power management
    - ▶ Understands relationship between components
    - ▶ Build device tree representing physical device topology
    - ▶ System bus -> I/O subsystem -> {flash, USB storage}
    - ▶ Device driver tracks state of device, whether in use
    - ▶ Unused component – turn it off
    - ▶ All devices in tree branch unused – turn off branch
  - Wake locks – like other locks but prevent sleep of device when lock is held
  - Power collapse – put a device into very deep sleep
    - ▶ Marginal power use
    - ▶ Only awake enough to respond to external stimuli (button press, incoming call)





# Kernel I/O Subsystem Summary

- In summary, the I/O subsystem coordinates an extensive collection of services that are available to applications and to other parts of the kernel
  - ▶ Management of the name space for files and devices
  - ▶ Access control to files and devices
  - ▶ Operation control (for example, a modem cannot seek())
  - ▶ File-system space allocation
  - ▶ Device allocation
  - ▶ Buffering, caching, and spooling
  - ▶ I/O scheduling
  - ▶ Device-status monitoring, error handling, and failure recovery
  - ▶ Device-driver configuration and initialization
  - ▶ Power management of I/O devices
- The upper levels of the I/O subsystem access devices via the uniform interface provided by the device drivers



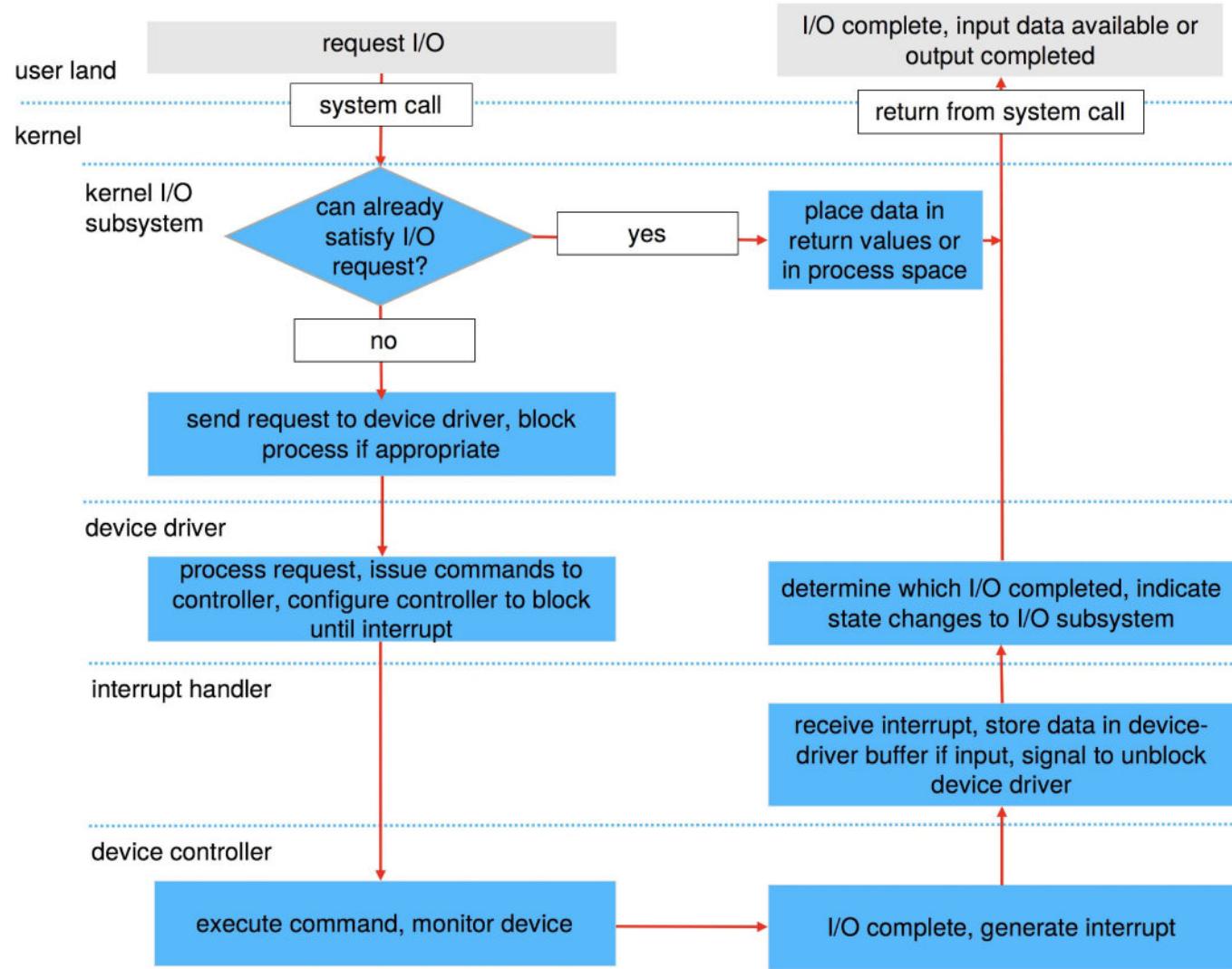


# Transforming I/O Requests to Hardware Operations

- Consider reading a file from disk for a process:
  - Determine device holding file
  - Translate name to device representation
  - Physically read data from disk into buffer
  - Make data available to requesting process
  - Return control to process



# Life Cycle of An I/O Request



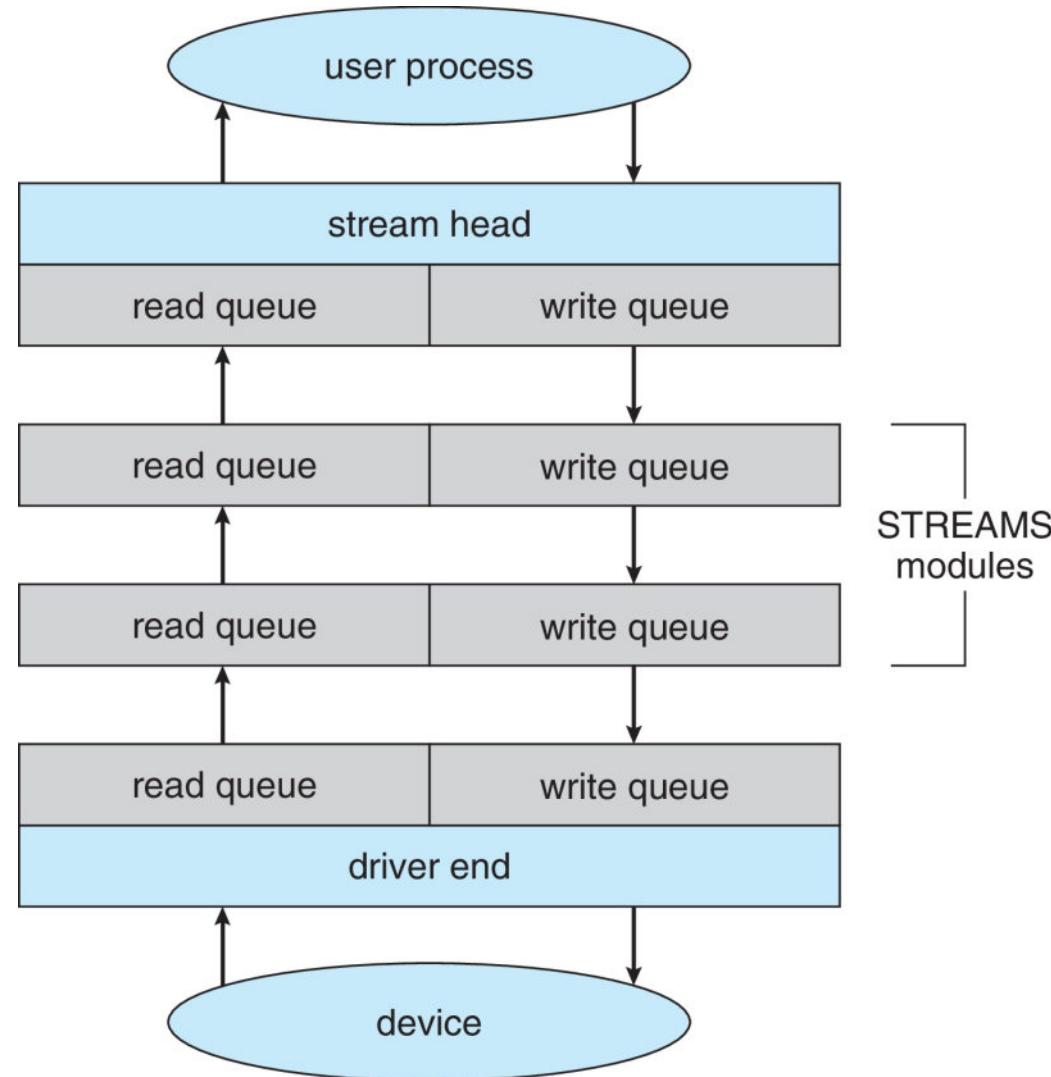


# STREAMS

- STREAM – a full-duplex communication channel between a user-level process and a device in Unix System V and beyond
- A STREAM consists of:
  - STREAM head interfaces with the user process
  - driver end interfaces with the device
  - zero or more STREAM modules between them
- Each module contains a read queue and a write queue
- Message passing is used to communicate between queues
  - Flow control option to indicate available or busy
- Asynchronous internally, synchronous where user process communicates with stream head



# The STREAMS Structure



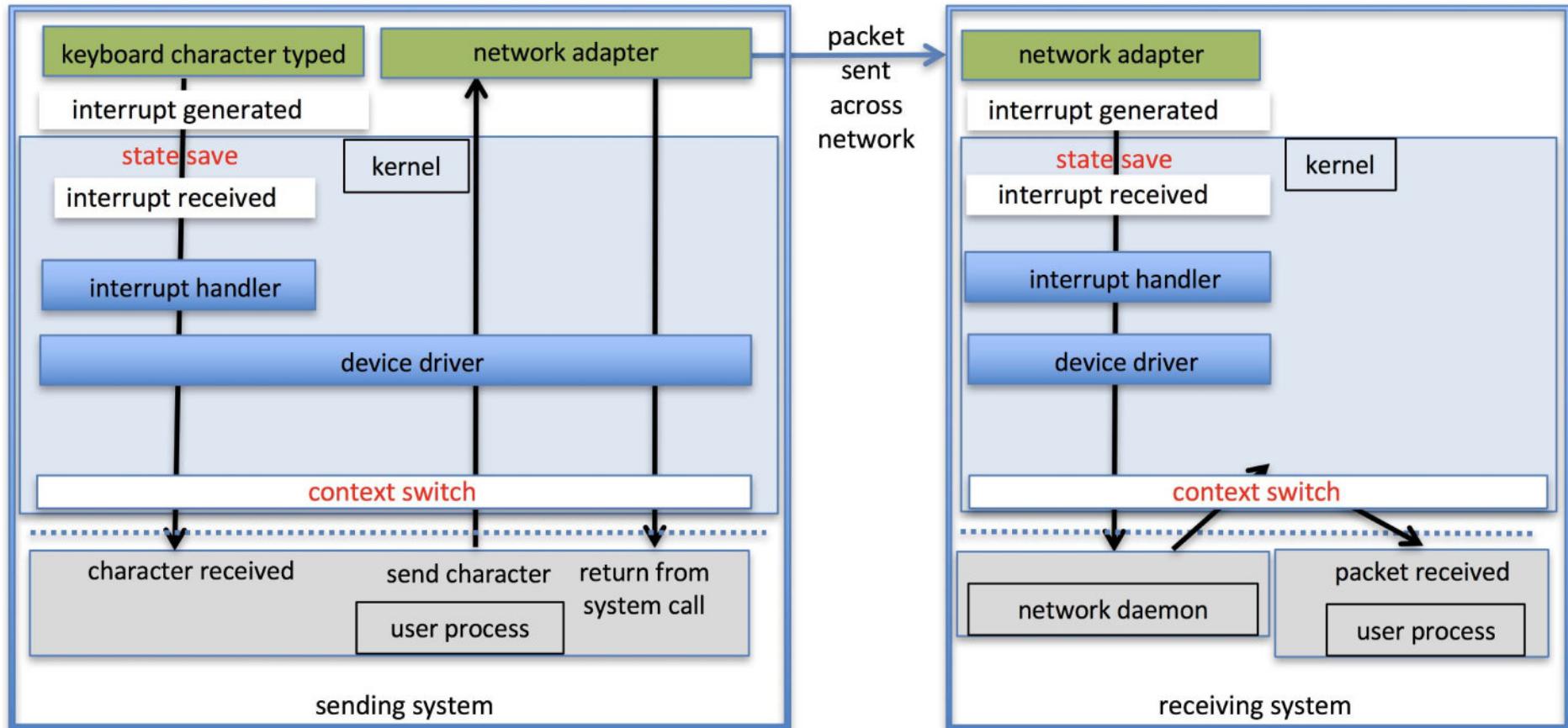


# Performance

- I/O a major factor in system performance:
  - Demands CPU to execute device driver, kernel I/O code
  - Context switches due to interrupts
  - Data copying
  - Network traffic especially stressful



# Intercomputer Communications



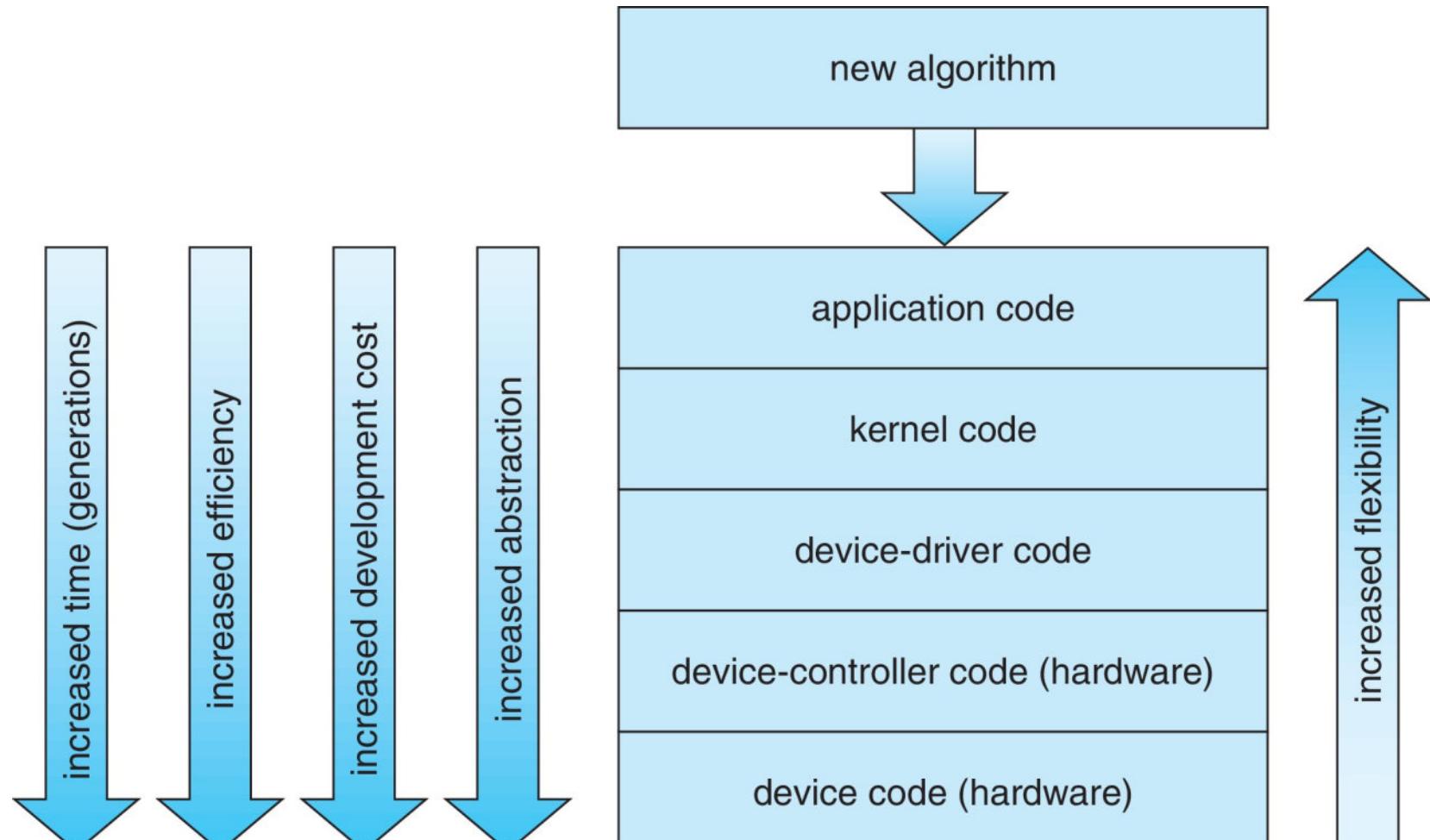


# Improving Performance

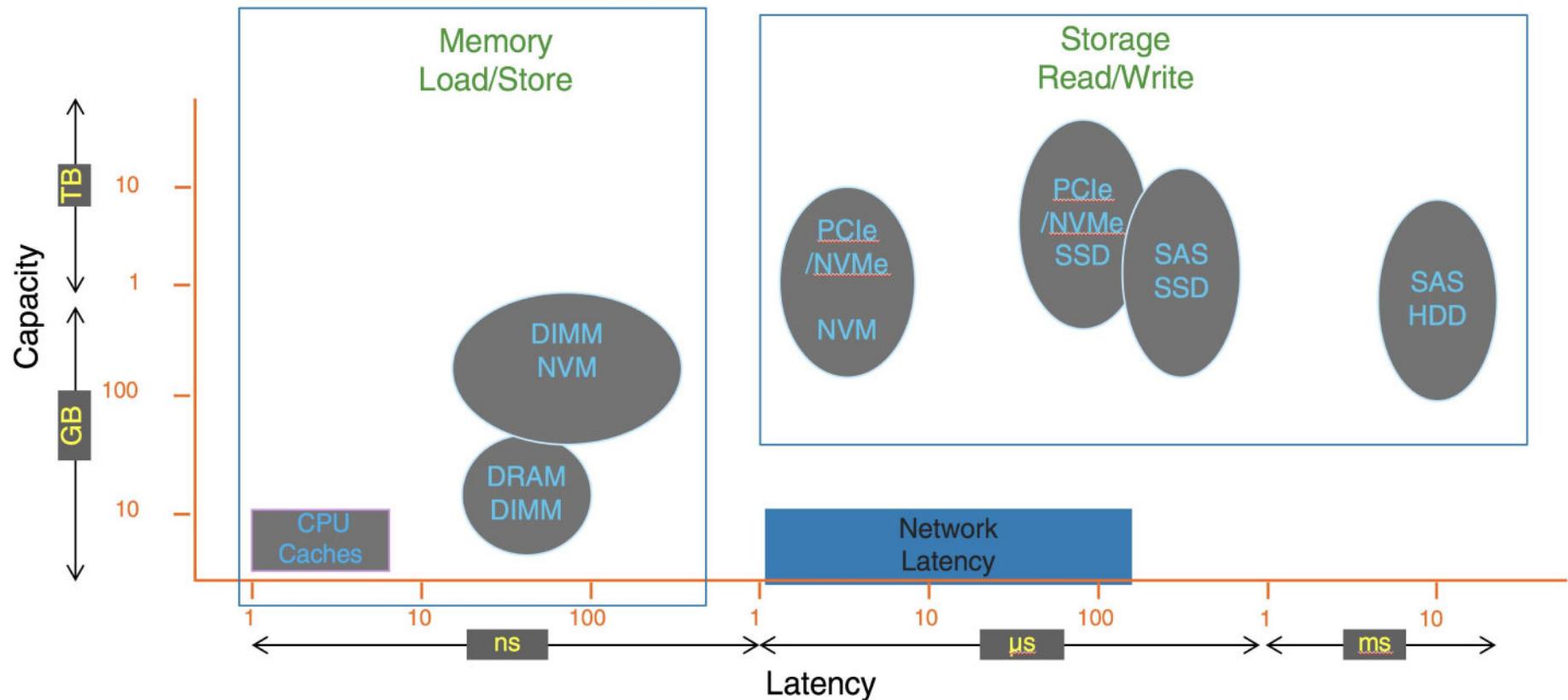
- Reduce number of context switches
- Reduce data copying
- Reduce interrupts by using large transfers, smart controllers, polling
- Use DMA
- Use smarter hardware devices
- Balance CPU, memory, bus, and I/O performance for highest throughput
- Move user-mode processes / daemons to kernel threads



# Device-Functionality Progression



# I/O Performance of Storage (and Network Latency)



# Summary

- The basic hardware elements involved in I/O are buses, device controllers, and the devices themselves.
- The work of moving data between devices and main memory is performed by the CPU as programmed I/O or is offloaded to a DMA controller.
- The kernel module that controls a device is a device driver. The system-call interface provided to applications is designed to handle several basic categories of hardware, including block devices, character-stream devices, memory-mapped files, network sockets, and programmed interval timers. The system calls usually block the processes that issue them, but nonblocking and asynchronous calls are used by the kernel itself and by applications that must not sleep while waiting for an I/O operation to complete.
- The kernel's I/O subsystem provides numerous services. Among these are I/O scheduling, buffering, caching, spooling, device reservation, error handling. Another service, name translation, makes the connections between hardware devices and the symbolic file names used by applications.





## Summary (Cont.)

It involves several levels of mapping that translate from character-string names, to specific device drivers and device addresses, and then to physical addresses of I/O ports or bus controllers. This mapping may occur within the file-system name space, as it does in UNIX, or in a separate device name space, as it does in MS-DOS.

- STREAMS is an implementation and methodology that provides a framework for a modular and incremental approach to writing device drivers and network protocols. Through STREAMS, drivers can be stacked, with data passing through them sequentially and bidirectionally for processing.
- I/O system calls are costly in terms of CPU consumption because of the many layers of software between a physical device and an application. These layers imply overhead from several sources: context switching to cross the kernel's protection boundary, signal and interrupt handling to service the I/O devices, and the load on the CPU and memory system to copy data between kernel buffers and application space.



# End of Chapter 12



What Is an  
**OPERATING SYSTEM (OS)**  
and How Does It Work

CLEVERISM.COM

# Chapter 13:

# File-System Interface





# Chapter 13: File-System Interface

- File Concept
- Access Methods
- Disk and Directory Structure
- File-System Mounting
- File Sharing
- Protection





# Objectives

- To explain the *function of file systems*
- To describe the *interfaces to file systems*
- To discuss *file-system design tradeoffs*, including access methods, file sharing, file locking, and directory structures
- To explore *file-system protection*





# File Concept

- Contiguous *logical address space*
- Types:
  - Data
    - ▶ complexe
    - ▶ numeric
    - ▶ character
    - ▶ binary
  - Program
- Contents defined by file's creator
  - Many types
    - ▶ Text file
    - ▶ Source file
    - ▶ Executable file





# File Attributes

- *Name* – only information kept in human-readable form
- *Identifier* – unique tag (number) identifies file within file system
- *Type* – needed for systems that support different types
- *Location* – pointer to file location on device
- *Size* – current file size
- *Protection* – controls who can do reading, writing, executing
- *Time, date, and user identification* – data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk
- Many variations, including extended file attributes such as file checksum





# A window of file info on Mac OS X



| Name      | Privilege    |
|-----------|--------------|
| greg (Me) | Read & Write |
| staff     | Read only    |
| everyone  | No Access    |





# File Operations

- File is an *abstract data type*
  - *Create*
  - *Write* – at write pointer location
  - *Read* – at read pointer location
  - *Reposition* within file (or seek)
  - *Delete*
  - *Truncate*
- *Open( $F_i$ )* – search the directory structure on disk for entry  $F_i$ , and move the content of entry to memory
- *Close( $F_i$ )* – move the content of entry  $F_i$  in memory to directory structure on disk





# Open Files

- Several pieces of data are needed to manage open files:
  - *Open-file table*: tracks open files
  - *File pointer*: pointer to last read/write location, per process that has the file open
  - *File-open count*: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
  - *Disk location of the file*: cache of data access information
  - *Access rights*: per-process access mode information





# Open File Locking

- Provided by some operating systems and file systems
  - Similar to *reader-writer locks*
  - *Shared lock* similar to reader lock – several processes can acquire concurrently
  - *Exclusive lock* similar to writer lock
- Mediates access to a file
- Mandatory or advisory:
  - *Mandatory* – access is denied depending on locks held and requested
  - *Advisory* – processes can find status of locks and decide what to do





# File Locking Example – Java API

```
import java.io.*;
import java.nio.channels.*;
public class LockingExample {
 public static final boolean EXCLUSIVE = false;
 public static final boolean SHARED = true;
 public static void main(String args[]) throws IOException {
 FileLock sharedLock = null;
 FileLock exclusiveLock = null;
 try {
 RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");
 // get the channel for the file
 FileChannel ch = raf.getChannel();
 // this locks the first half of the file - exclusive
 exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);
 /** Now modify the data . . . */
 // release the lock
 exclusiveLock.release();
 }
 }
}
```





# File Locking Example – Java API (Cont.)

```
// this locks the second half of the file - shared
sharedLock = ch.lock(raf.length()/2+1, raf.length(), SHARED);
/** Now read the data . . . */
// release the lock
sharedLock.release();
} catch (java.io.IOException ioe) {
 System.err.println(ioe);
}finally {
 if (exclusiveLock != null)
 exclusiveLock.release();
 if (sharedLock != null)
 sharedLock.release();
}
}
```



# File Types – Name, Extension

| file type      | usual extension          | function                                                                            |
|----------------|--------------------------|-------------------------------------------------------------------------------------|
| executable     | exe, com, bin or none    | ready-to-run machine-language program                                               |
| object         | obj, o                   | compiled, machine language, not linked                                              |
| source code    | c, cc, java, pas, asm, a | source code in various languages                                                    |
| batch          | bat, sh                  | commands to the command interpreter                                                 |
| text           | txt, doc                 | textual data, documents                                                             |
| word processor | wp, tex, rtf, doc        | various word-processor formats                                                      |
| library        | lib, a, so, dll          | libraries of routines for programmers                                               |
| print or view  | ps, pdf, jpg             | ASCII or binary file in a format for printing or viewing                            |
| archive        | arc, zip, tar            | related files grouped into one file, sometimes compressed, for archiving or storage |
| multimedia     | mpeg, mov, rm, mp3, avi  | binary file containing audio or A/V information                                     |





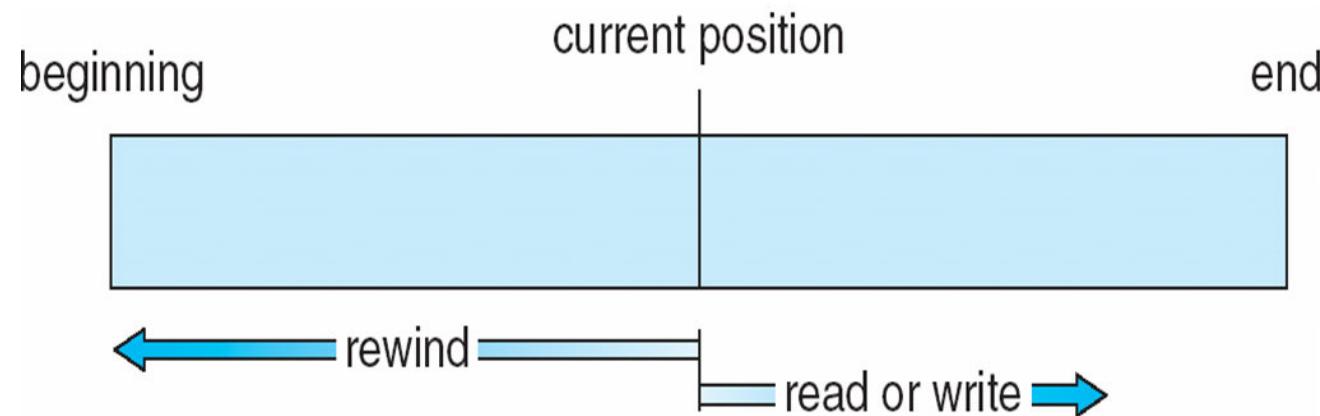
# File Structure

- *None* - sequence of words or bytes
- *Simple record structures*
  - Lines
  - Fixed length
  - Variable length
- *Complex structures*
  - Formatted document
  - Relocatable load file (i.e., executable file)
- Can simulate last two with first method by inserting appropriate control characters
- Who decides:
  - Operating system
  - Program





# Sequential-Access File





# Access Methods

## ■ Sequential Access

**read next**  
**write next**  
**reset**  
no read after last write  
(rewrite)

## ■ Direct Access – file is *fixed-length logical records*

**read  $n$**   
**write  $n$**   
**position to  $n$**   
**read next**  
**write next**  
**rewrite  $n$**

$n$  = *relative block number*

- Relative block numbers allow OS to decide where file should be placed
- See *allocation problem* in Chapter 12





# Simulation of Sequential-Access on Direct-Access File

| sequential access | implementation for direct access |
|-------------------|----------------------------------|
| <i>reset</i>      | $cp = 0;$                        |
| <i>read next</i>  | $read cp;$<br>$cp = cp + 1;$     |
| <i>write next</i> | $write cp;$<br>$cp = cp + 1;$    |



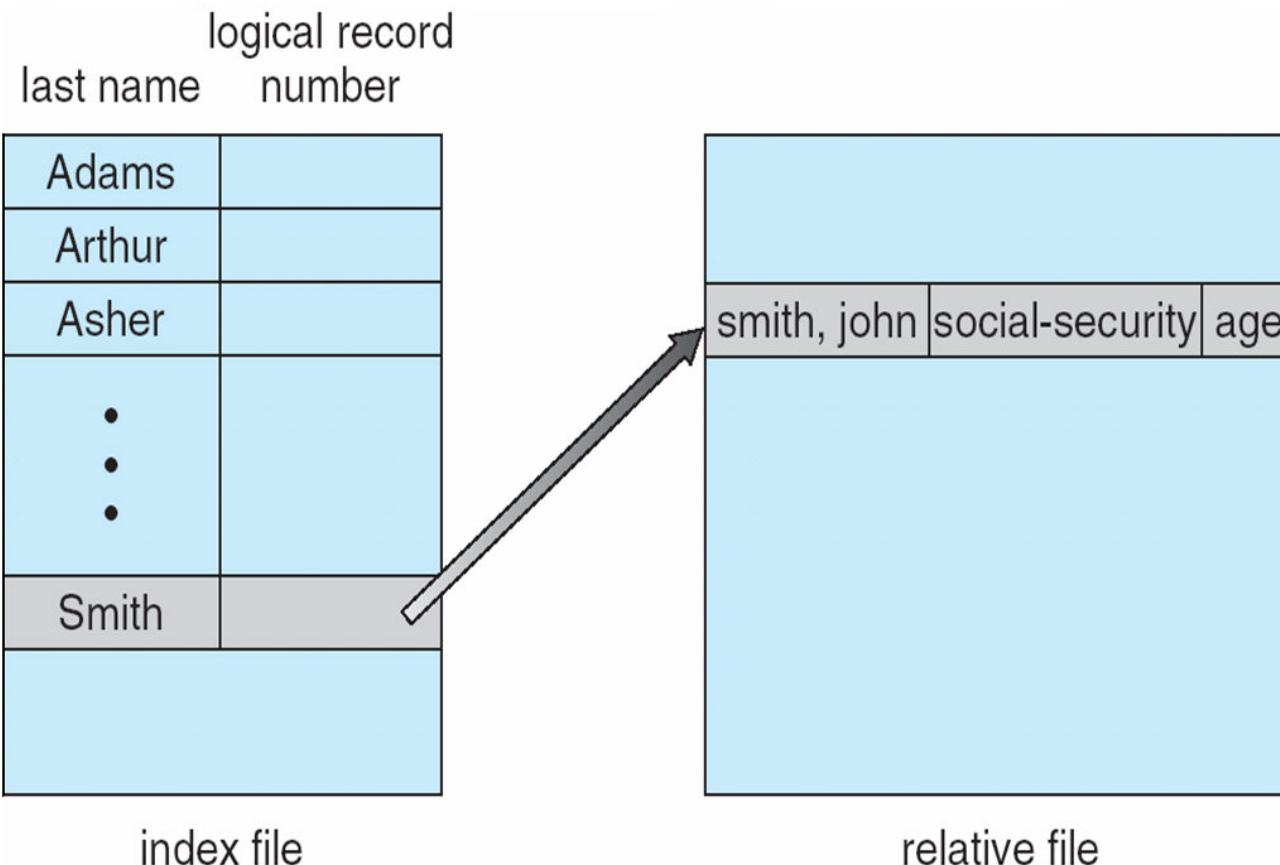


## Other Access Methods

- Can be built on top of base methods
- General involve creation of an *index* for the file
- Keep index in memory for fast determination of location of data to be operated on (consider UPC code plus record of data about that item)
- If too large, index (in memory) of the index (on disk)
- E.g., **IBM Indexed Sequential-Access Method (ISAM)**
  - Small master index, points to disk blocks of secondary index
  - File kept sorted on a defined key
  - All done by the OS
- **VMS** operating system provides index and relative files as another example (see next slide)

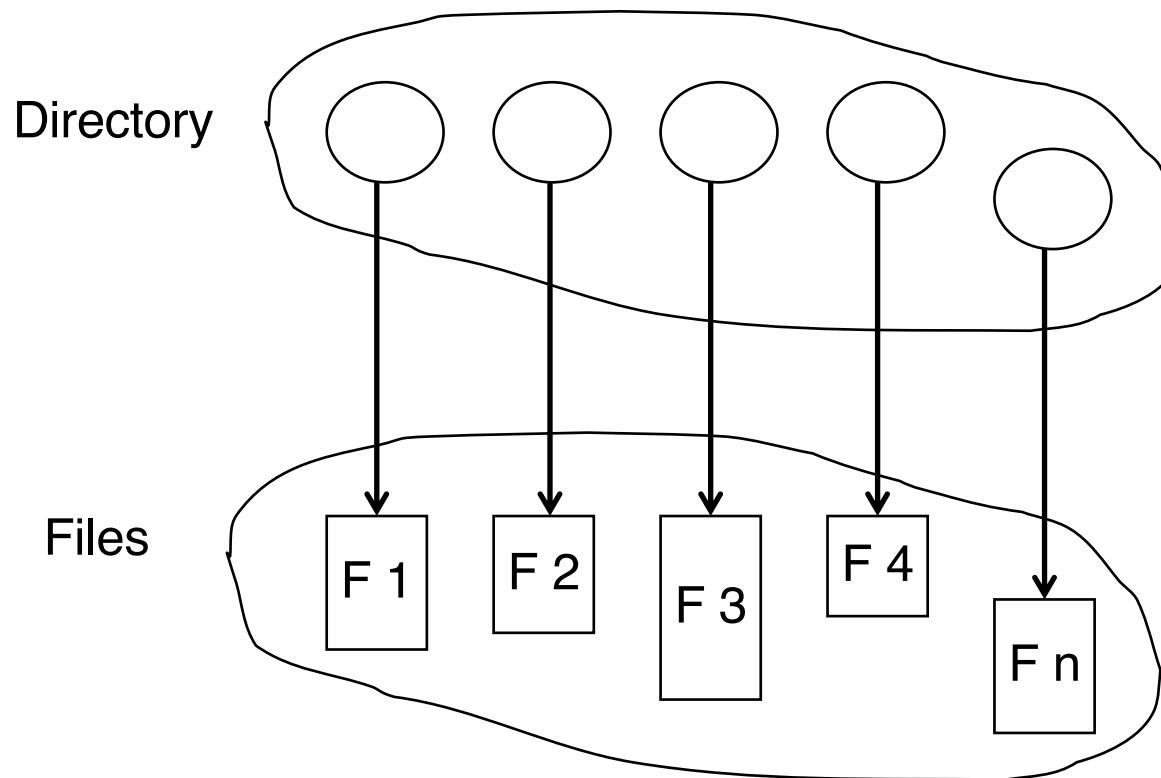


# Example of Index and Relative Files



# Directory Structure

- A *collection of nodes* containing information about all files



Both the directory structure and the files reside on disk



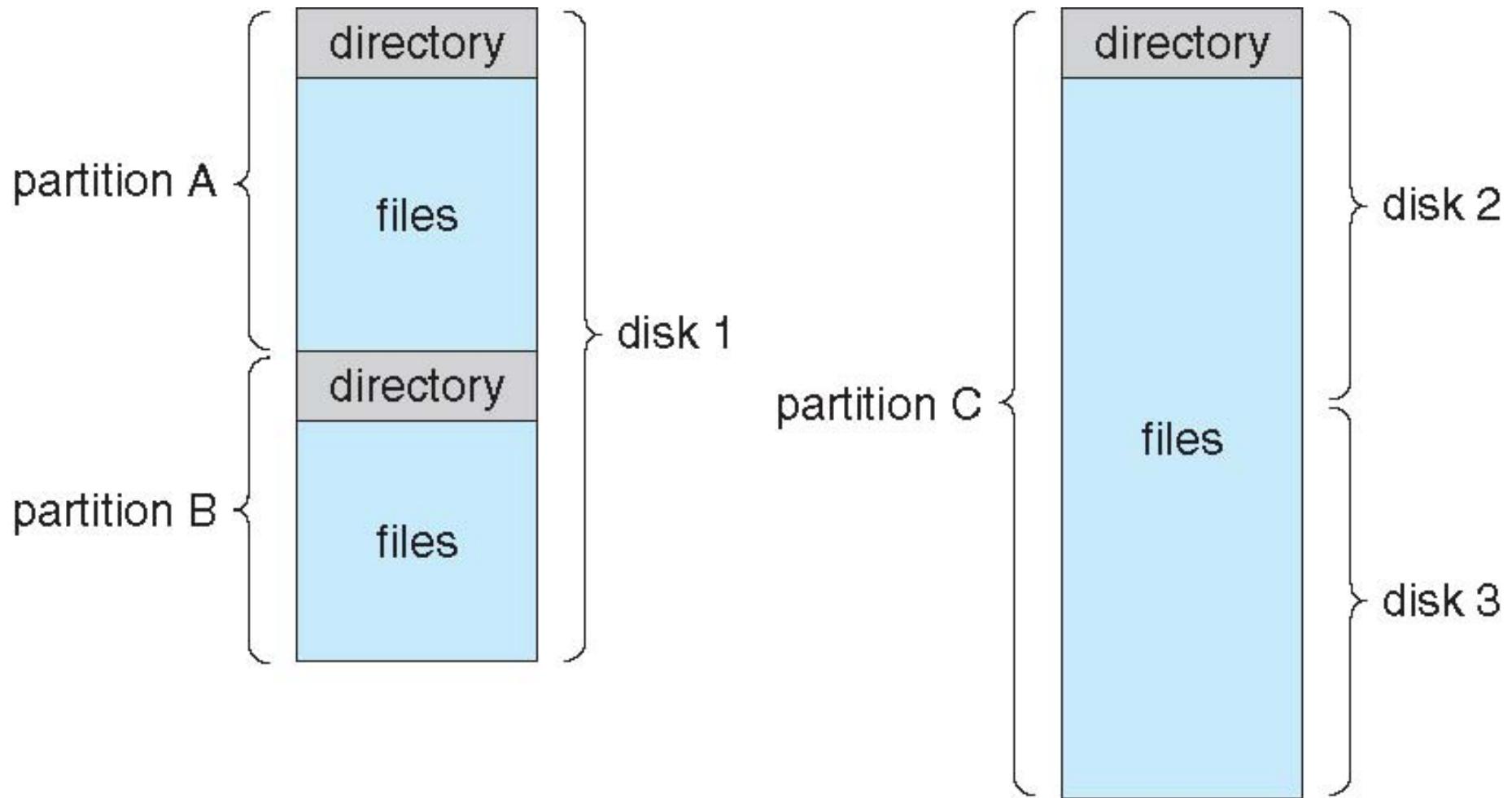


# Disk Structure

- *Disk* can be subdivided into *partitions*
  - Disks or partitions can be **RAID** protected against failure
  - Disk or partition can be used *raw* – without a file system, or *formatted* with a file system
- Partitions also known as minidisks, slices
- Entity containing file system known as a *volume*
  - Each volume containing file system also tracks that file system's info in device directory or volume table of contents
- As well as general-purpose file systems there are many special-purpose file systems, frequently all within the same operating system or computer



# A Typical File-System Organization





# Types of File Systems

- We mostly talk of *general-purpose file systems*
- But systems frequently have many file systems, some general- and some special- purpose
- E.g., **Solaris** has
  - *tmpfs* – memory-based volatile FS for fast, temporary I/O
  - *objfs* – interface into kernel memory to get kernel symbols for debugging
  - *ctfs* – contract file system for managing daemons
  - *lofs* – loopback file system allows one FS to be accessed in place of another
  - *procfs* – kernel interface to process structures
  - *ufs, zfs* – general purpose file systems





# Operations Performed on Directory

- *Search* for a file
- *Create* a file
- *Delete* a file
- *List* a directory
- *Rename* a file
- *Traverse* the file system





# Directory Organization

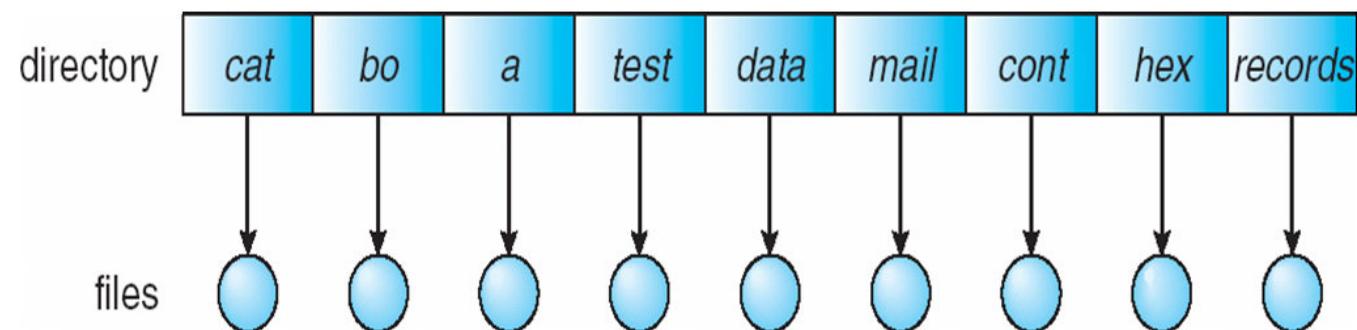
- The directory is organized logically to obtain
  - *Efficiency* – locating a file quickly
  - *Naming* – convenient to users
    - ▶ Two users can have same name for different files
    - ▶ The same file can have several different names
  - *Grouping* – logical grouping of files by properties, (e.g., all Java programs, all games, ...)





# Single-Level Directory

- *A single directory for all users*

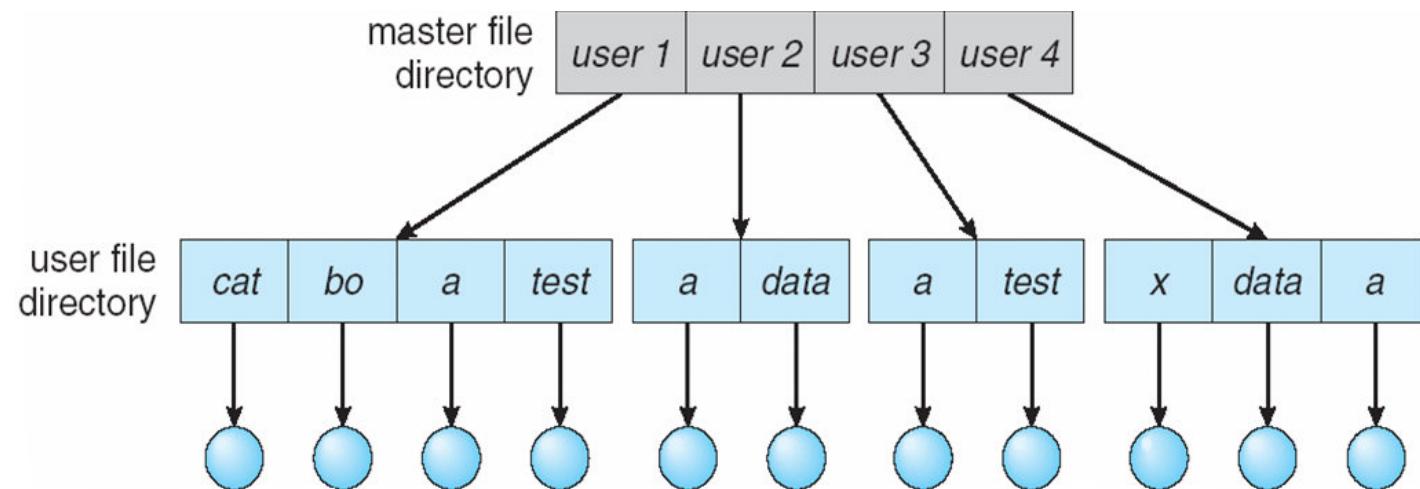


- Naming problem
- Grouping problem

# Two-Level Directory

- *Separate directory for each user*

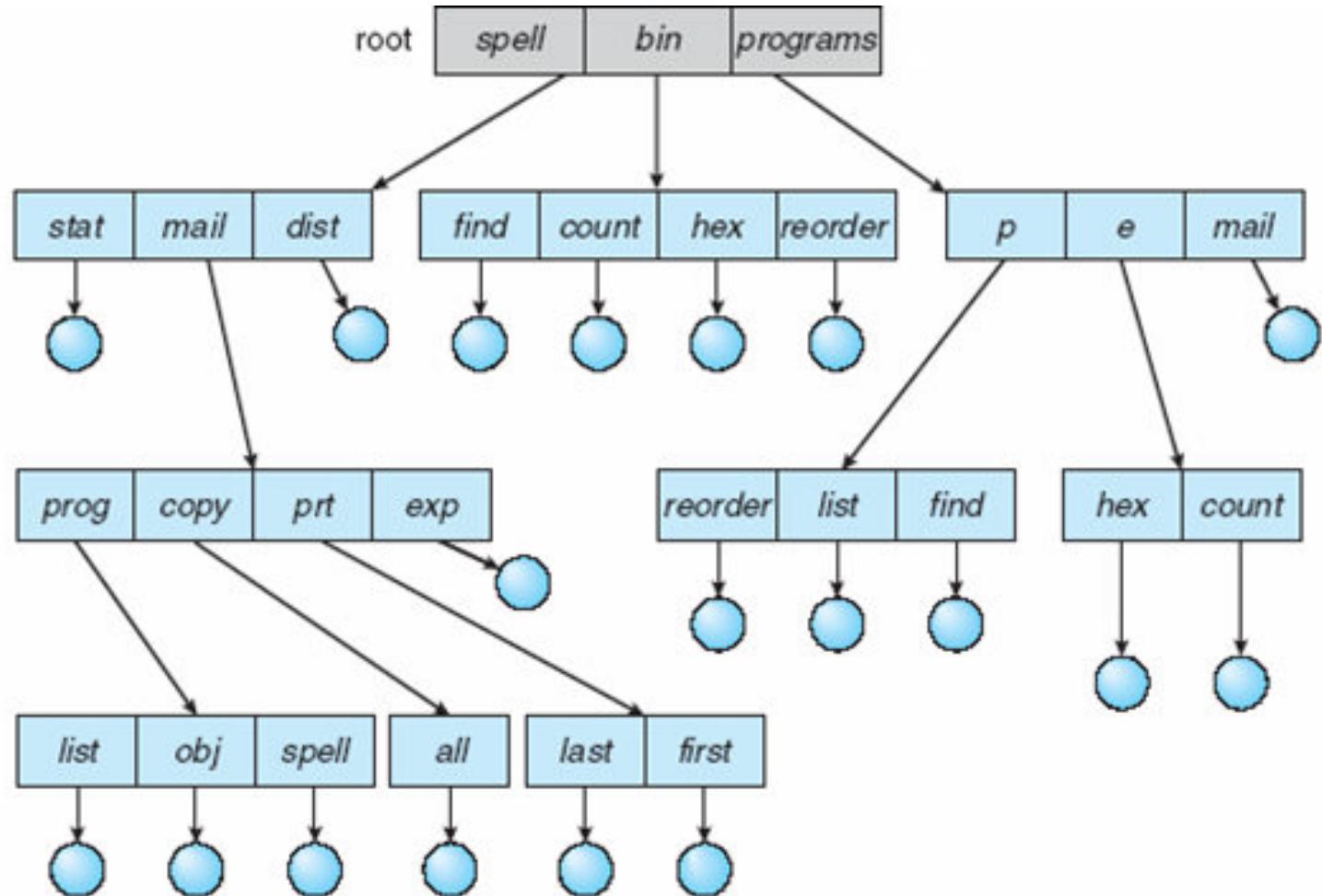
- Master file directory (MFD)
- User file directory (UFD)



- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability



# Tree-Structured Directories





# Tree-Structured Directories (Cont.)

- Efficient searching
- Grouping Capability
- Current directory (or working directory)
  - E.g., For Linux OS,  
`cd /spell/mail/prog`  
`type list`





## Tree-Structured Directories (Cont.)

- Using *absolute* or *relative* path name
- Creating a new file is done in current directory
- Delete a file

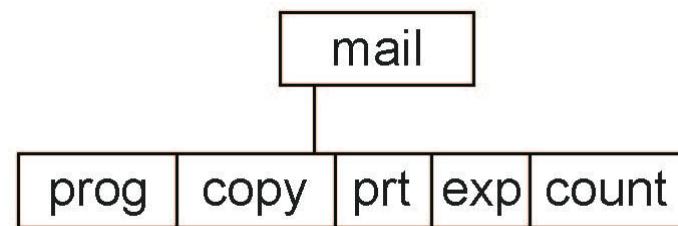
`rm <file-name>`

- Creating a new subdirectory is done in current directory

`mkdir <dir-name>`

- Example: if in current directory `/mail`

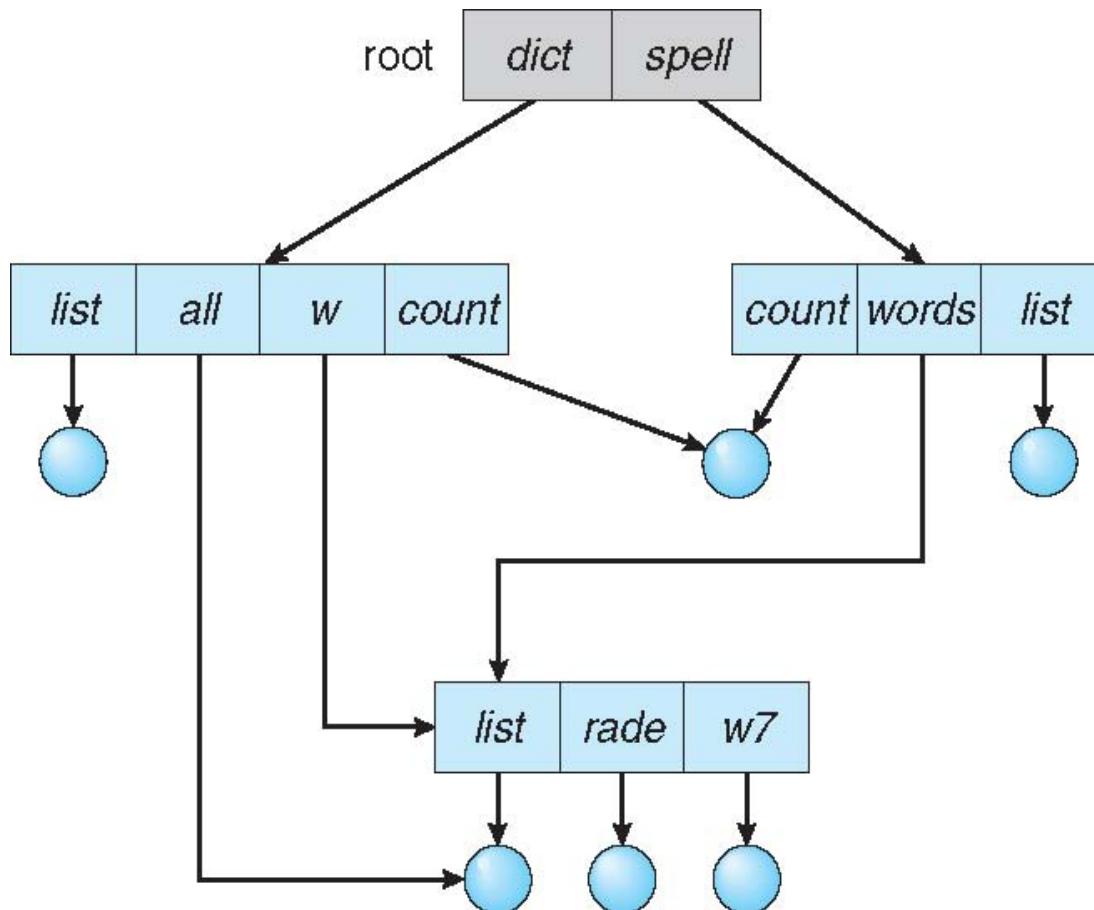
`mkdir count`



- ▶ Deleting “mail” ⇒ deleting the entire subtree rooted by “mail”



# Acyclic-Graph Directories



- Have shared subdirectories and files

- Two different names (*aliasing*)
  - ▶ If *dict* deletes *list* ⇒ dangling pointer.
- Solutions:
  - ▶ *Backpointers*, so we can delete all pointers
  - Variable size records a problem
    - Backpointers using a daisy chain organization
  - ▶ *Entry-hold-count* solution



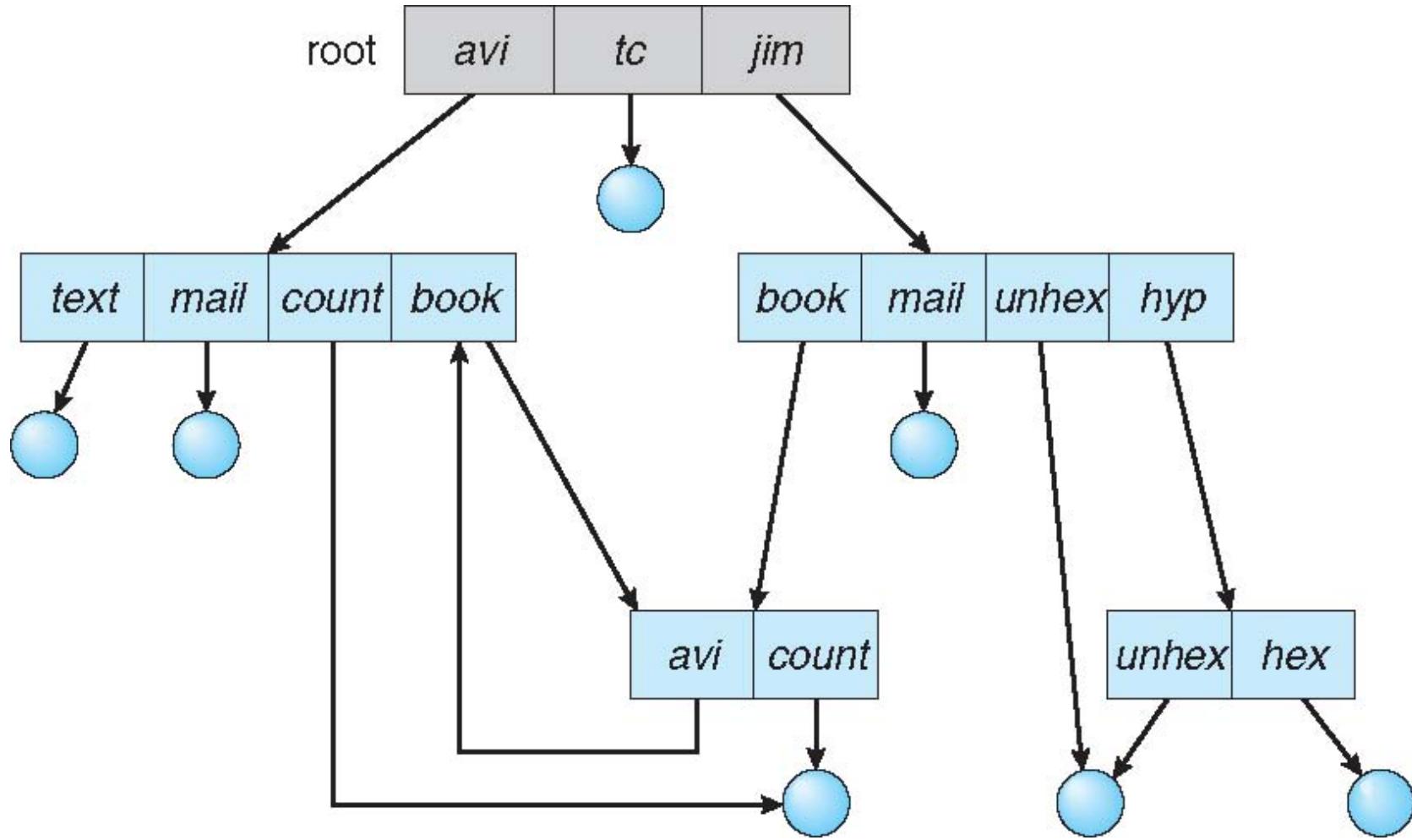


## Acyclic-Graph Directories (Cont.)

- New directory entry type
  - *Link* – another name (pointer) to an existing file
  - *Resolve the link* – follow pointer to locate the file



# General Graph Directory





## General Graph Directory (Cont.)

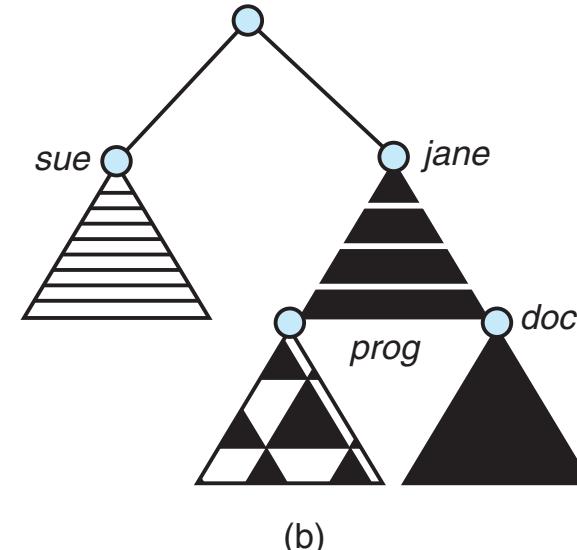
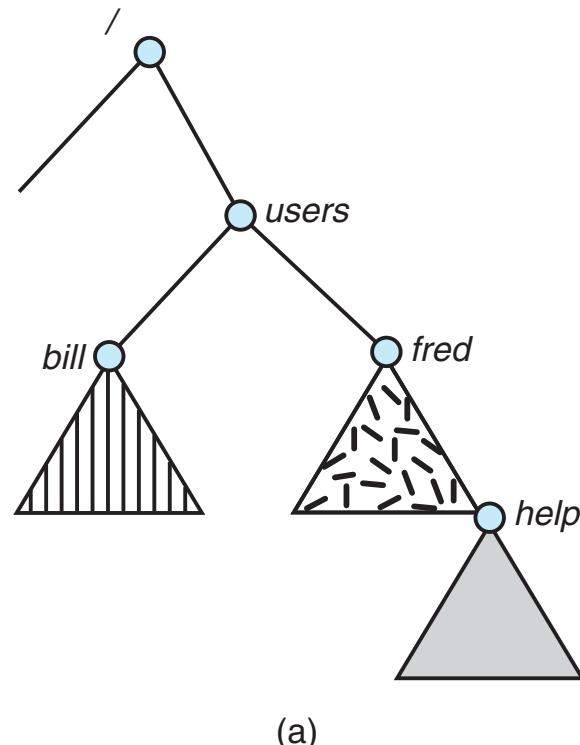
- How do we guarantee no cycles?

- Allow *only links to file* not subdirectories
- *Garbage collection*
- Every time a new link is added use a *cycle detection algorithm* to determine whether it is OK

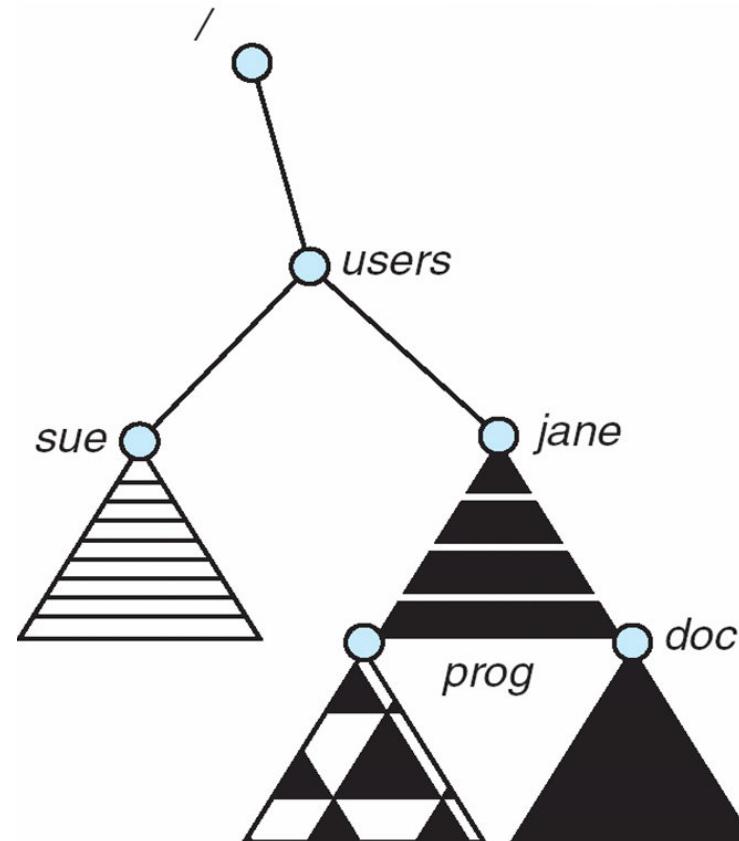


# File System Mounting

- A file system must be *mounted* before it can be *accessed*
- An unmounted file system (i.e., Fig. (b)) is mounted at a mount point



# Mount Point





# File Sharing

- Sharing of files on *multi-user systems* is desirable
- Sharing may be done through a protection scheme
- On distributed systems, files may be shared across a network
  - *Network File System (NFS)* is a common distributed file-sharing method
- If multi-user system
  - Owner of a file / directory
    - ▶ *User IDs* identify users, allowing permissions and protections to be per-user
  - Group of a file / directory
    - ▶ *Group IDs* allow users to be in groups, permitting group access rights





# File Sharing – Remote File Systems

- Uses *networking* to allow file system access between systems
  - Manually via programs like FTP
  - Automatically, seamlessly using *distributed file systems*
  - Semi automatically via the world wide web
- *Client-server model* allows clients to mount remote file systems from servers
  - Server can serve multiple clients
  - Client and user-on-client identification is insecure or complicated
  - **NFS** is standard UNIX client-server file sharing protocol
  - **CIFS** is standard Windows protocol
  - Standard operating system file calls are translated into remote calls
- *Distributed Information Systems* (distributed naming services) such as **LDAP**, **DNS**, **NIS**, **Active Directory** implement unified access to information needed for remote computing





# File Sharing – Failure Modes

- All file systems have *failure modes*
  - For example corruption of directory structures or other non-user data, called *metadata*
- Remote file systems add new failure modes, due to *network failure*, *server failure*
- Recovery from failure can involve state information about status of each remote request
- Stateless protocols such as **NFS v.3** include all information in each request, allowing easy recovery but less security





# File Sharing – Consistency Semantics

- Specify how multiple users are to access a shared file simultaneously
  - Similar to Ch. 5 *process synchronization algorithms*
    - ▶ Tend to be less complex due to disk I/O and network latency (for remote file systems)
  - *Andrew File System (AFS)* implemented complex remote file sharing semantics
    - ▶ AFS has session semantics
      - Writes only visible to sessions starting after the file is closed
  - *Unix file system (UFS)* implements:
    - ▶ Writes to an open file visible immediately to other users of the same open file
    - ▶ Sharing file pointer to allow multiple users to read and write concurrently





# Protection

- *File owner/creator* should be able to control:
  - what can be done
  - by whom
- *Types of access*
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List



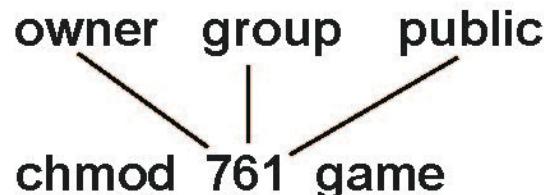


# Access Lists and Groups

- Mode of access: *read* (R), *write* (W), *execute* (X)
- Three classes of users on Unix / Linux

|                  |   |   |       |
|------------------|---|---|-------|
|                  |   |   | RWX   |
| a) owner access  | 7 | ⇒ | 1 1 1 |
|                  |   |   | RWX   |
| b) group access  | 6 | ⇒ | 1 1 0 |
|                  |   |   | RWX   |
| c) public access | 1 | ⇒ | 0 0 1 |

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.



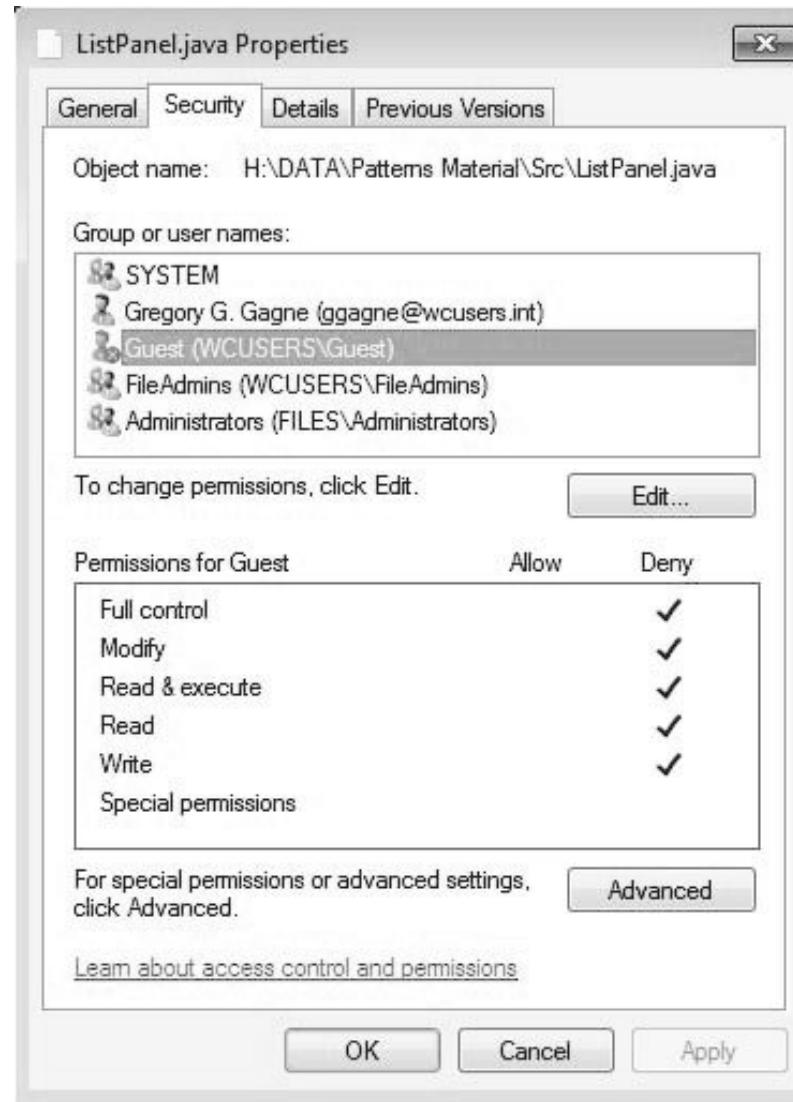
Attach a group to a file

chgrp      G      game





# Windows 7 Access-Control List Management





# A Sample UNIX Directory Listing

|            |   |     |         |       |              |               |
|------------|---|-----|---------|-------|--------------|---------------|
| -rw-rw-r-- | 1 | pbg | staff   | 31200 | Sep 3 08:30  | intro.ps      |
| drwx-----  | 5 | pbg | staff   | 512   | Jul 8 09:33  | private/      |
| drwxrwxr-x | 2 | pbg | staff   | 512   | Jul 8 09:35  | doc/          |
| drwxrwx--- | 2 | pbg | student | 512   | Aug 3 14:13  | student-proj/ |
| -rw-r--r-- | 1 | pbg | staff   | 9423  | Feb 24 2003  | program.c     |
| -rwxr-xr-x | 1 | pbg | staff   | 20471 | Feb 24 2003  | program       |
| drwx--x--x | 4 | pbg | faculty | 512   | Jul 31 10:31 | lib/          |
| drwx-----  | 3 | pbg | staff   | 1024  | Aug 29 06:52 | mail/         |
| drwxrwxrwx | 3 | pbg | staff   | 512   | Jul 8 09:35  | test/         |





# Summary

- A *file* is an abstract data type defined and implemented by the operating system. It is a *sequence of logical records*. A logical record may be a byte, a line (of fixed or variable length), or a more complex data item. The operating system may specifically support various record types or may leave that support to the application program.
- A major task for the operating system is to *map the logical file concept onto physical storage devices* such as hard disk or NVM device. Since the physical record size of the device may not be the same as the logical record size, it may be necessary to order logical records into physical records. Again, this task may be supported by the operating system or left for the application program.
- Within a *file system*, it is useful to create *directories* to allow files to be organized. A *single-level directory* in a multiuser system causes naming problems, since each file must have a unique name. A *two-level directory* solves this problem by creating a separate directory for each user's files. The directory lists the files by name and includes the file's location on the disk, length, type, owner, time of creation, time of last use, ...





## Summary (Cont.)

- The natural generalization of a two-level directory is a *tree-structured directory*. A tree-structured directory allows a user to create subdirectories to organize files. *Acyclic-graph directory structures* enable users to share subdirectories and files but complicate searching and deletion. A general graph structure allows complete flexibility in the sharing of files and directories but sometimes requires garbage collection to recover unused disk space.
- *Remote file systems* present challenges in reliability, performance, and security. Distributed information systems maintain user, host, and access information so that clients and servers can share state information to manage use and access.
- Since files are the main information-storage mechanism in most computer systems, *file protection* is needed on multiuser systems. Access to files can be controlled separately for each type of access — read, write, execute, append, delete, list directory, and so on. File protection can be provided by access lists, passwords, or other techniques.



# End of Chapter 13



What Is an  
**OPERATING SYSTEM (OS)**  
and How Does It Work

CLEVERISM.COM

# Chapter 14: File-System Implementation



What Is an  
**OPERATING SYSTEM (OS)**  
and How Does It Work

CLEVERISM.COM



# Chapter 14: File-System Implementation

- File-System Structure
- File-System Operations
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery
- Example: WAFL File System





# Objectives

- Describe the details of *implementing local file systems* and *directory structures*
- Discuss *block allocation* and *free-block algorithms* and trade-offs
- Explore *file system efficiency* and *performance issues*
- Look at recovery from *file system failures*
- Describe the *WAFL file system* as a concrete example





# File-System Structure

- *File structure*

- Logical storage unit
- Collection of related information

- *File system* resides on secondary storage (e.g., disks)

- Provided user interface to storage, mapping logical to physical
- Provides efficient and convenient access to disk by allowing data to be stored, located, and retrieved easily

- *Disk* provides in-place rewrite and random access

- I/O transfers performed in blocks of sectors (usually 512 bytes)

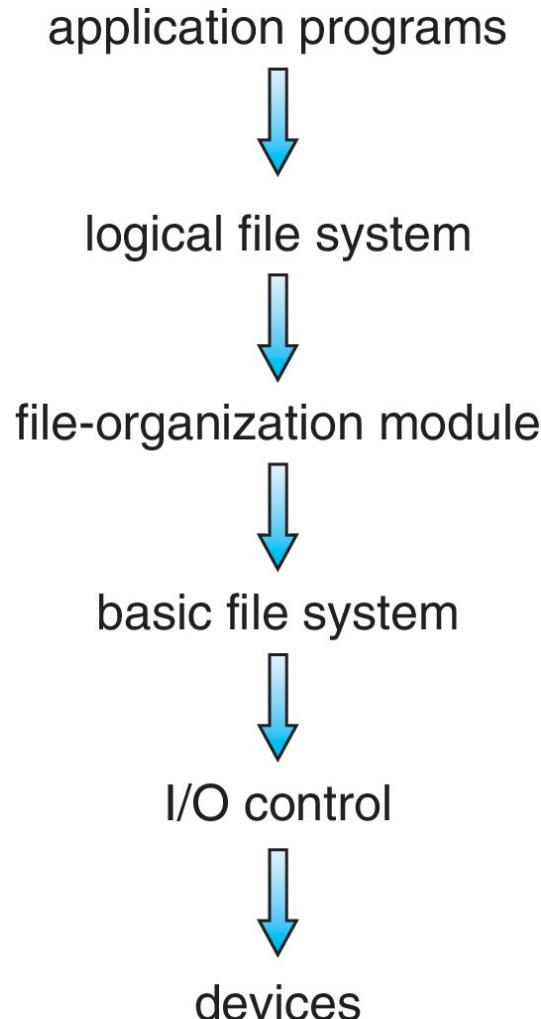
- *File control block (FCB)* – storage structure consisting of information about a file

- *Device driver* controls the physical device

- File system is organized into *layers*



# Layered File System



- *Layering* useful for reducing complexity and redundancy, but adds overhead and can decrease performance



# File System Layers

- *Device drivers* manage I/O devices at the I/O control layer
  - E.g., Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level hardware specific commands to hardware controller
- *Basic file system* given command like “retrieve block 123” translates to device driver
  - Also manages memory buffers and caches (allocation, freeing, replacement)
    - ▶ *Buffers* hold data in transit
    - ▶ *Caches* hold frequently used data
- *File organization module* understands files, logical address, and physical blocks
  - Translates logical block # to physical block #
  - Manages free space, disk allocation





# File System Layers (Cont.)

- *Logical file system* manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (i.e., *inodes* in UNIX)
  - Directory management
  - Protection
  - Logical layers can be implemented by any coding method according to OS designer
- Many file systems exist within an operating system
  - Each with its own format (CD-ROM is ISO 9660; Unix has UFS, FFS; Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray, Linux has more than 130 types, with extended file system ext3 and ext4 leading; plus distributed file systems NFS, etc.)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE, Lustre



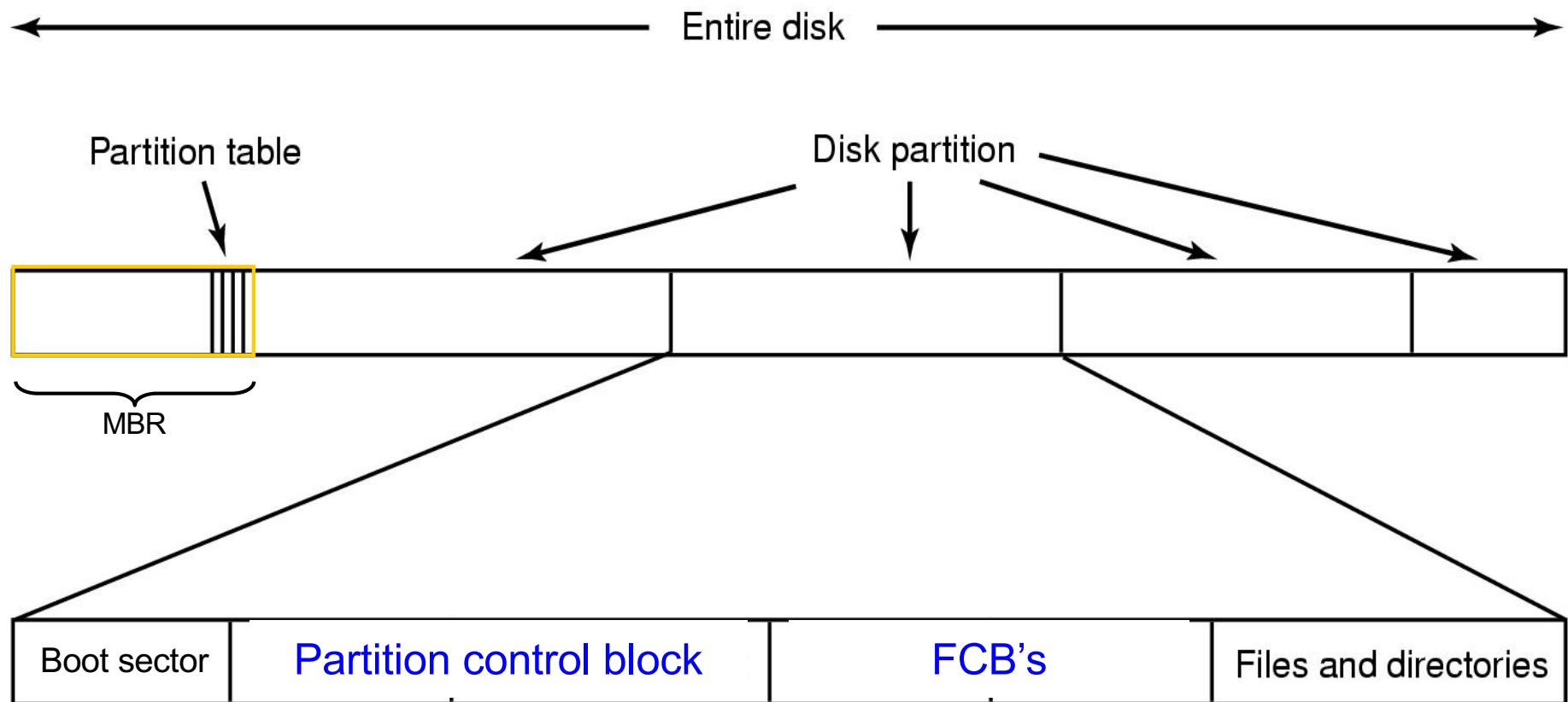


# File-System Implementation

- We have *system calls* at the API level, but how do we implement their functions?
  - *On-disk* and *in-memory* structures
- *Boot control block* contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- *Volume control block* (e.g., superblock, master file table) contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- *Directory structure* organizes the files
  - Names and inode numbers, master file table



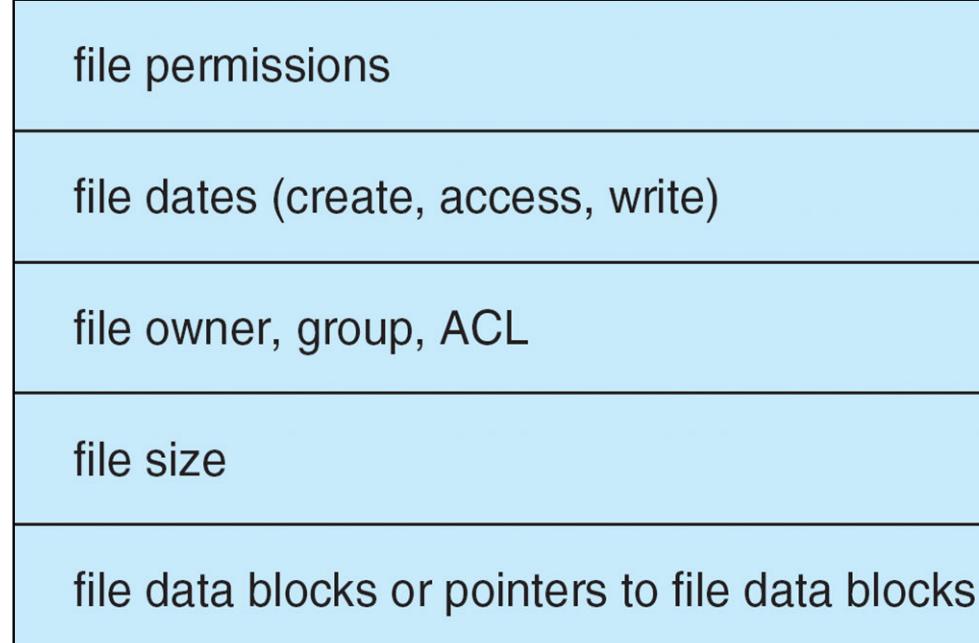
# File-System Implementation (Cont.)





## File-System Implementation (Cont.)

- Per-file *File Control Block (FCB)* contains many details about the file
  - typically inode number, permissions, size, dates
  - NTFS stores into in *master file table* using relational DB structures



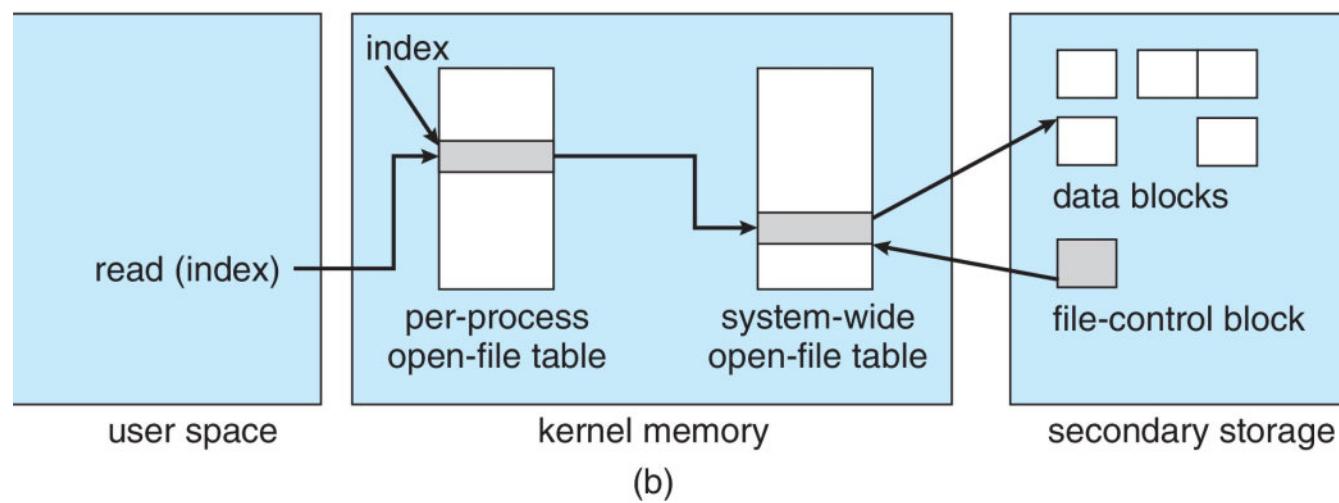
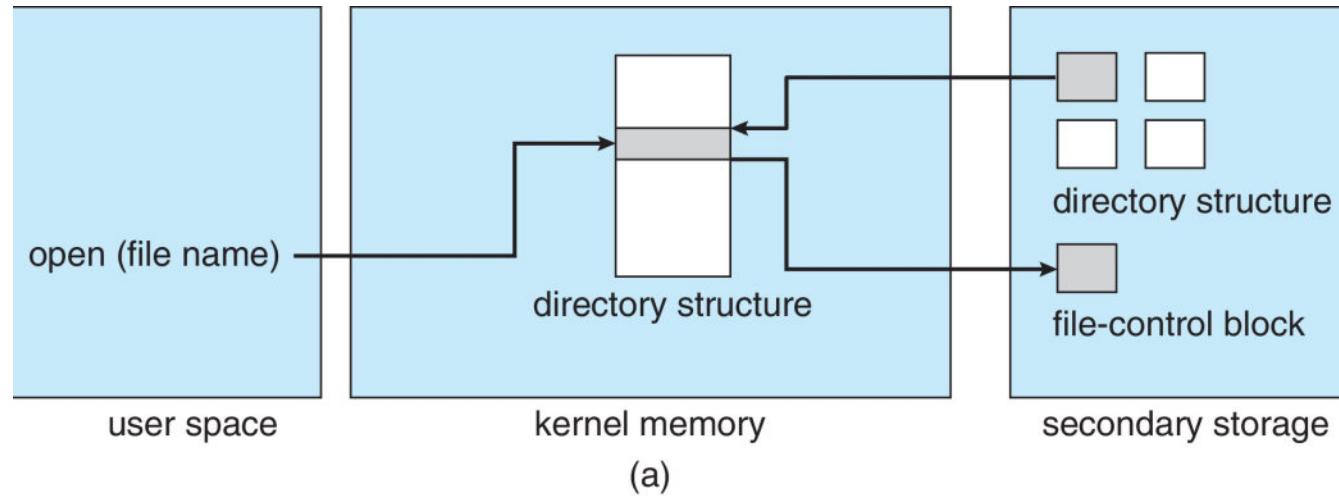


# In-Memory File System Structures

- *Mount table* storing file system mounts, mount points, file system types
- *System-wide open-file table* contains a copy of the FCB of each file and other info
- *Per-process open-file table* contains pointers to appropriate entries in system-wide open-file table as well as other info
- The following figure illustrates the necessary file system structures provided by the operating systems
  - Figure (a) refers to opening a file
  - Figure (b) refers to reading a file
- Plus *buffers* hold data blocks from secondary storage
- Open returns a *file handle* for subsequent use
- Data from read copied to specified user process memory address



# In-Memory File System Structures





# Directory Implementation

## ■ *Linear list of file names* with pointer to the data blocks

- Simple to program
- Time-consuming to execute
  - ▶ Linear search time
  - ▶ Could keep ordered alphabetically via linked list or use B+ tree

## ■ *Hash Table* – linear list with hash data structure

- Decreases directory search time
- Collisions – situations where two file names hash to the same location
- Only good if entries are fixed size, or use chained-overflow method





# Allocation Methods – Contiguous

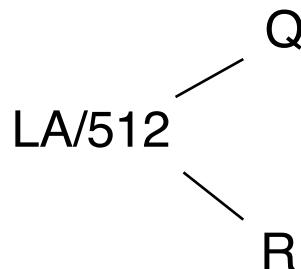
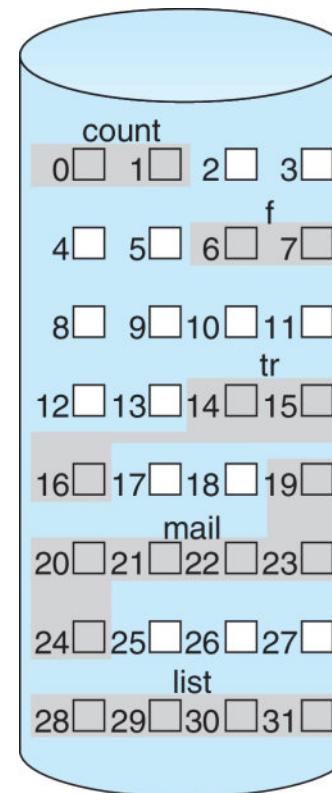
- An allocation method refers to *how disk blocks are allocated for files*
- *Contiguous allocation* – each file occupies set of contiguous blocks
  - Best performance in most cases
  - Simple – only starting location (block #) and length (number of blocks) are required
  - Problems include finding space for file, knowing file size, external fragmentation, need for compaction off-line (downtime) or on-line



# Contiguous Allocation

## ■ Mapping from logical to physical

LA/512

| directory |       |        |
|-----------|-------|--------|
| file      | start | length |
| count     | 0     | 2      |
| tr        | 14    | 3      |
| mail      | 19    | 6      |
| list      | 28    | 4      |
| f         | 6     | 2      |

- Block to be accessed = Q + starting address
- Displacement into block = R





# Extent-Based Systems

- Many newer file systems (e.g., *Veritas File System*) use a modified contiguous allocation scheme
- *Extent-based file systems* allocate disk blocks in extents
- An extent is a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents





# Allocation Methods - Linked

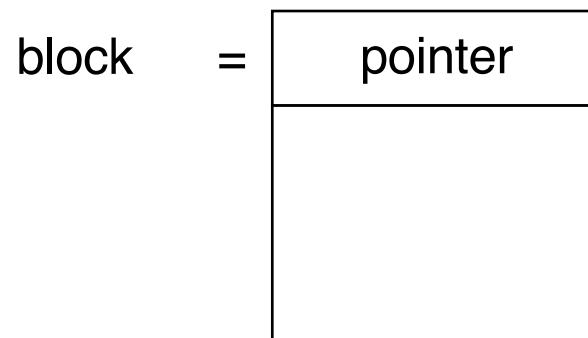
- *Linked allocation* – each file is a linked list of blocks
  - File ends at nil pointer
  - No external fragmentation
  - Each block contains pointer to next block
  - No compaction, external fragmentation
  - Free space management system called when new block needed
  - Improve efficiency by clustering blocks into groups but increases internal fragmentation
  - Reliability can be a problem
  - Locating a block can take many I/Os and disk seeks
- *File Allocation Table (FAT)* variation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple



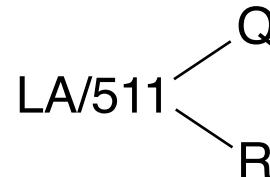


# Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk



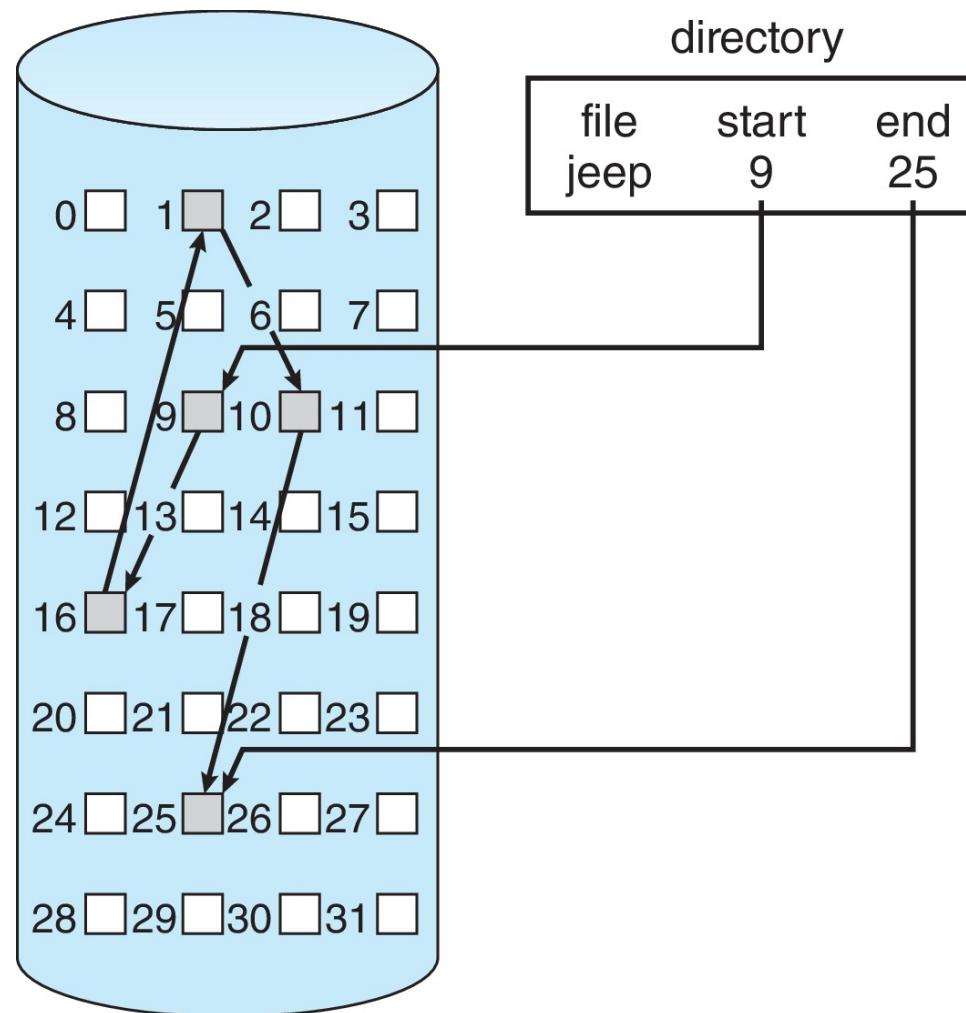
- Mapping



- Block to be accessed is the  $Q^{\text{th}}$  block in the linked chain of blocks representing the file
- Displacement into block =  $R + 1$

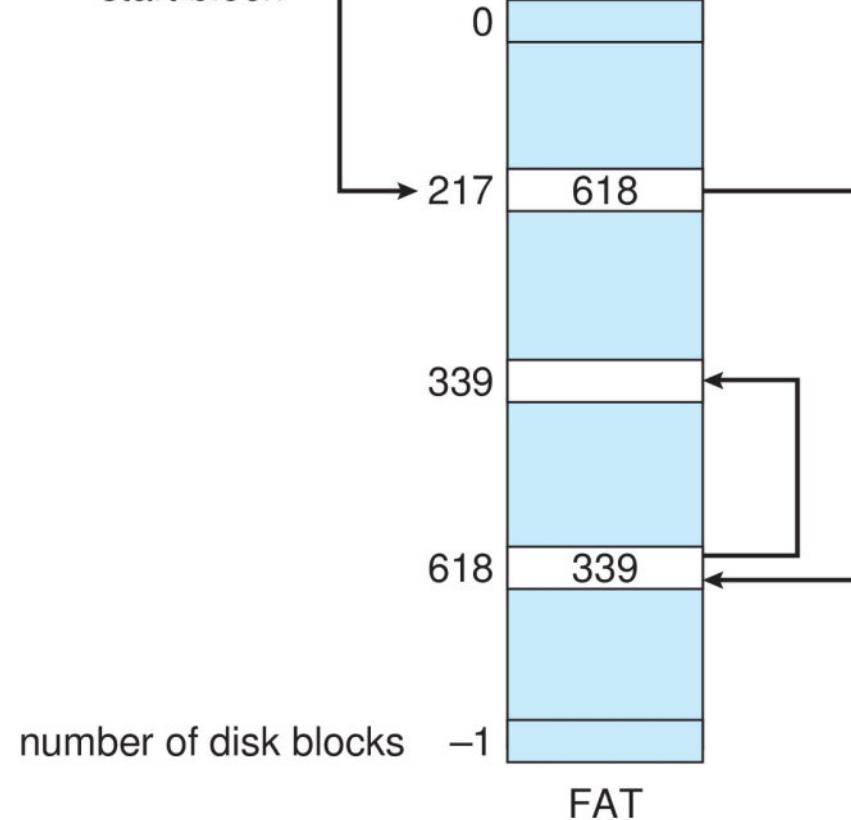
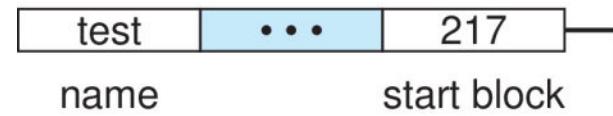


# Linked Allocation



# File-Allocation Table (FAT)

directory entry

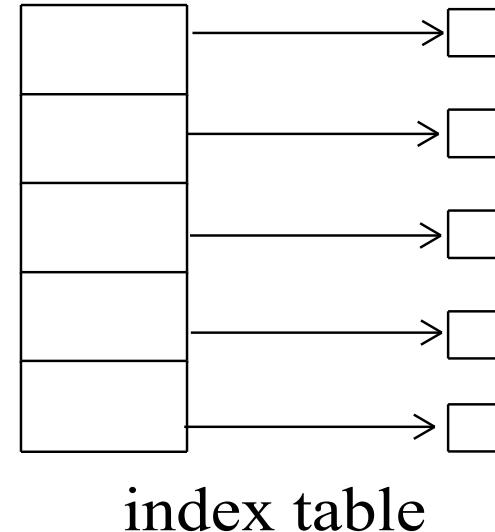


# Allocation Methods – Indexed

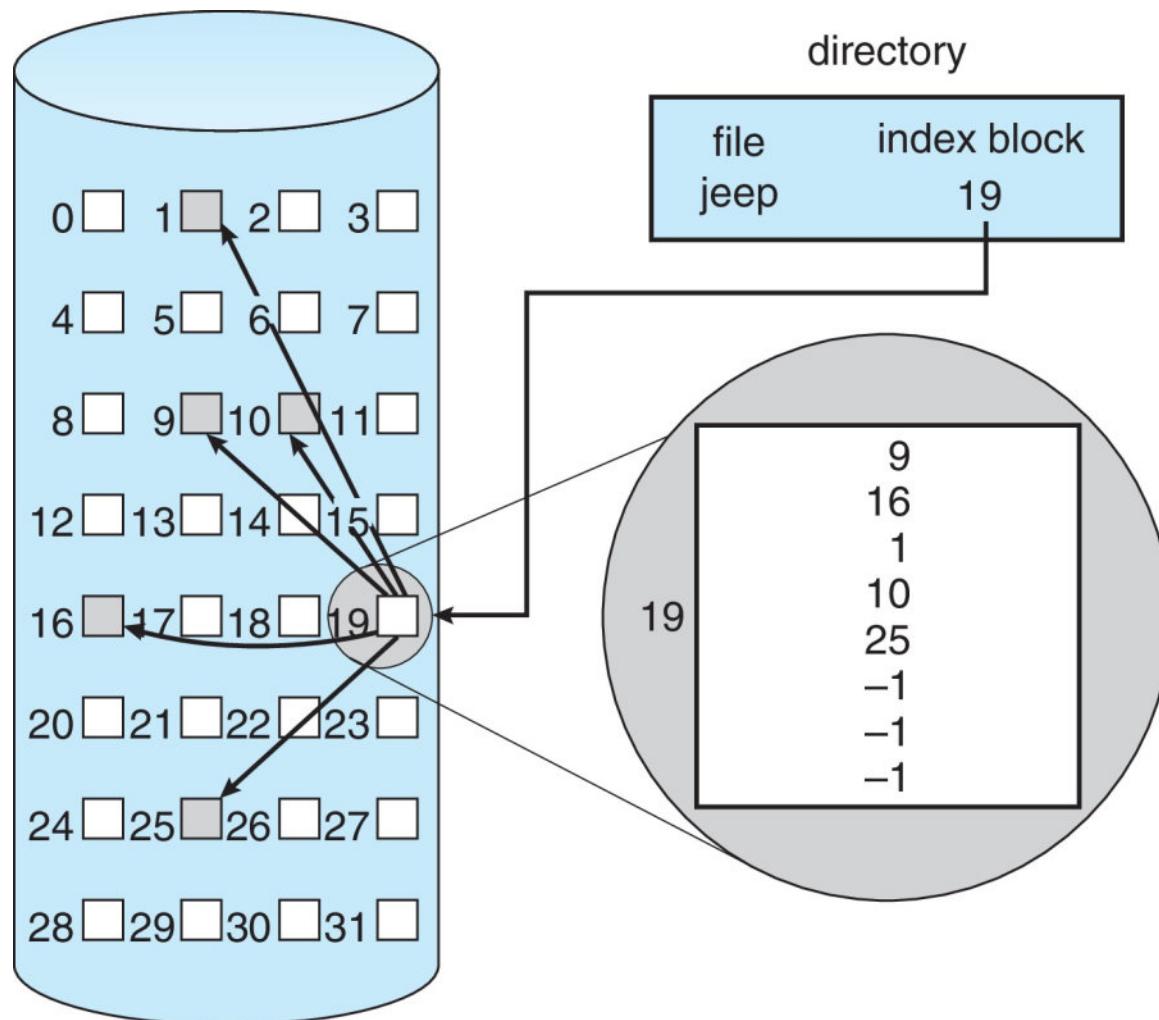
## ■ Indexed allocation

- Each file has its own index block(s) of pointers to its data blocks

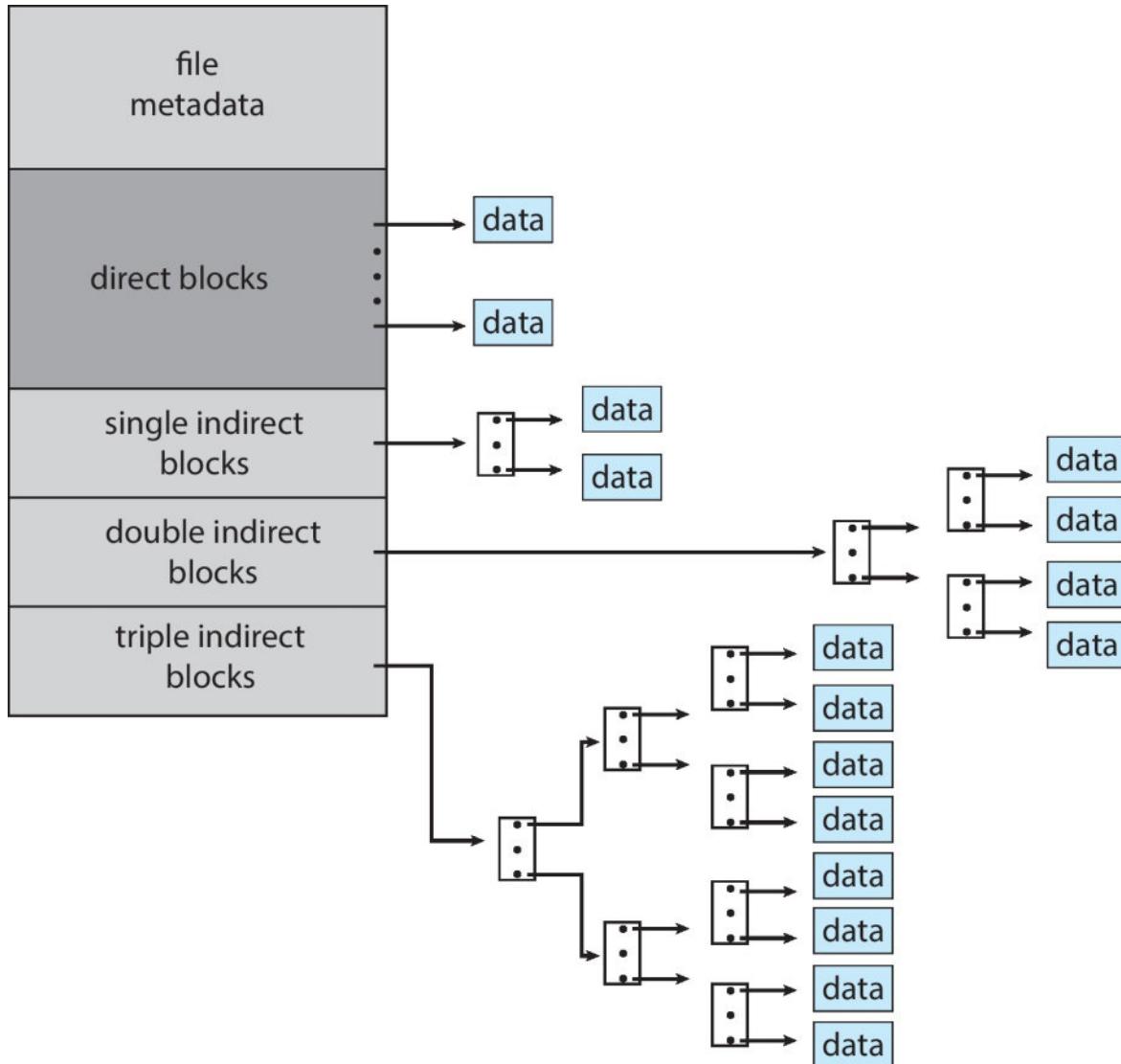
## ■ Logical view



# Example of Indexed Allocation



# Combined Scheme: UNIX UFS



- 4K bytes per block, 32-bit addresses
  - More index blocks than can be addressed with 32-bit file pointer



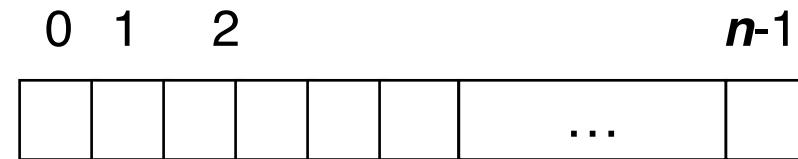


# Performance

- *Best method* depends on file access type
  - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation → select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads, then data block read
  - Clustering can help improve throughput, reduce CPU overhead
- For NVM, no disk head so different algorithms and optimizations needed
  - An old algorithm uses many CPU cycles for trying to avoid non-existent head movement
  - With NVM goal is to reduce CPU cycles and overall path needed for I/O



- File system maintains free-space list to track available blocks/clusters
  - (Using term “block” for simplicity)
- Bit vector or bit map (n blocks)


$$\text{bit}[i] = \begin{cases} 1 & \Rightarrow \text{block}[i] \text{ free} \\ 0 & \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

- Block number calculation
  - $(\text{number of bits per word}) * (\text{number of 0-value words}) + \text{offset of first 1 bit}$
- CPUs have instructions to return offset within word of first “1” bit





## Free-Space Management (Cont.)

- Bit map requires extra space

- Example:

block size = 4KB = 212 bytes

disk size = 240 bytes (1 terabyte)

$n = 240/212 = 228$  bits (or 32MB)

if clusters of 4 blocks -> 8MB of memory

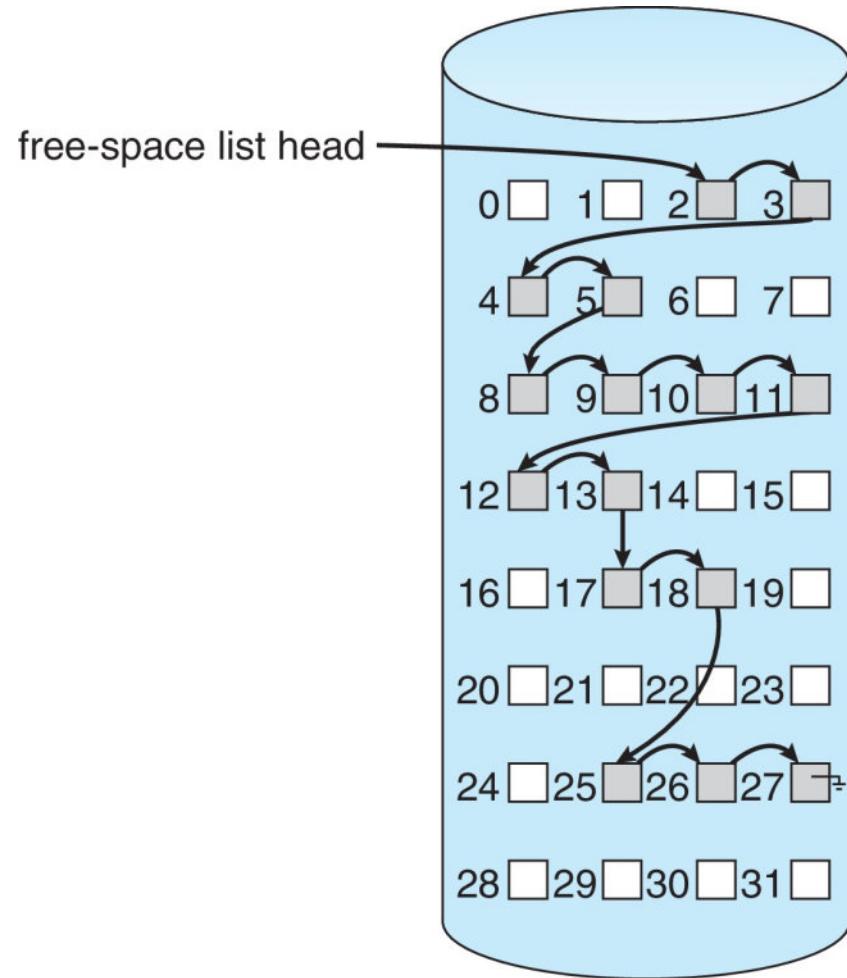
- Easy to get contiguous files



# Linked Free Space List on Disk

## ■ Linked list (free list)

- Cannot get contiguous space easily
- No waste of space
- No need to traverse the entire list (if # free blocks recorded)





# Free-Space Management (Cont.)

## ■ Grouping

- Modify linked list to store address of next ( $n-1$ ) free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this one)

## ■ Counting

- Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering
  - ▶ Keep address of first free block and count of following free blocks
  - ▶ Free space list then has entries containing addresses and counts





# Free-Space Management (Cont.)

## ■ Space Maps

- Used in ZFS
- Consider meta-data I/O on very large file systems
  - ▶ Full data structures like bit maps couldn't fit in memory → thousands of I/Os
- Divides device space into metaslab units and manages metaslabs
  - ▶ Given volume can contain hundreds of metaslabs
- Each metaslab has associated space map
  - ▶ Uses counting algorithm
- But records to log file rather than file system
  - ▶ Log of all block activity, in time order, in counting format
- Metaslab activity → load space map into memory in balanced-tree structure, indexed by offset
  - ▶ Replay log into that structure
  - ▶ Combine contiguous free blocks into single entry





# TRIMing Unused Blocks

- **HDDs** overwrite in place so need only free list
  - Blocks not treated specially when freed
  - Keeps its data but without any file pointers to it, until overwritten
- Storage devices not allowing overwrite (e.g., **NVM**) suffer badly with same algorithm
  - Must be erased before written, erases made in large chunks (blocks, composed of pages) and are slow
  - **TRIM** is a newer mechanism for the file system to inform the NVM storage device that a page is free
    - ▶ Can be garbage collected or if block is free, now block can be erased





# Efficiency and Performance

## ■ *Efficiency* dependent on:

- Disk allocation and directory algorithms
- Types of data kept in file's directory entry
- Pre-allocation or as-needed allocation of metadata structures
- Fixed-size or varying-size data structures

## ■ *Performance*

- Keeping data and metadata close together
- Buffer cache – separate section of main memory for frequently used blocks
- Synchronous writes sometimes requested by apps or needed by OS
  - ▶ No buffering / caching – writes must hit disk before acknowledgement
  - ▶ Asynchronous writes more common, buffer-able, faster
- Free-behind and read-ahead – techniques to optimize sequential access
- Reads frequently slower than writes



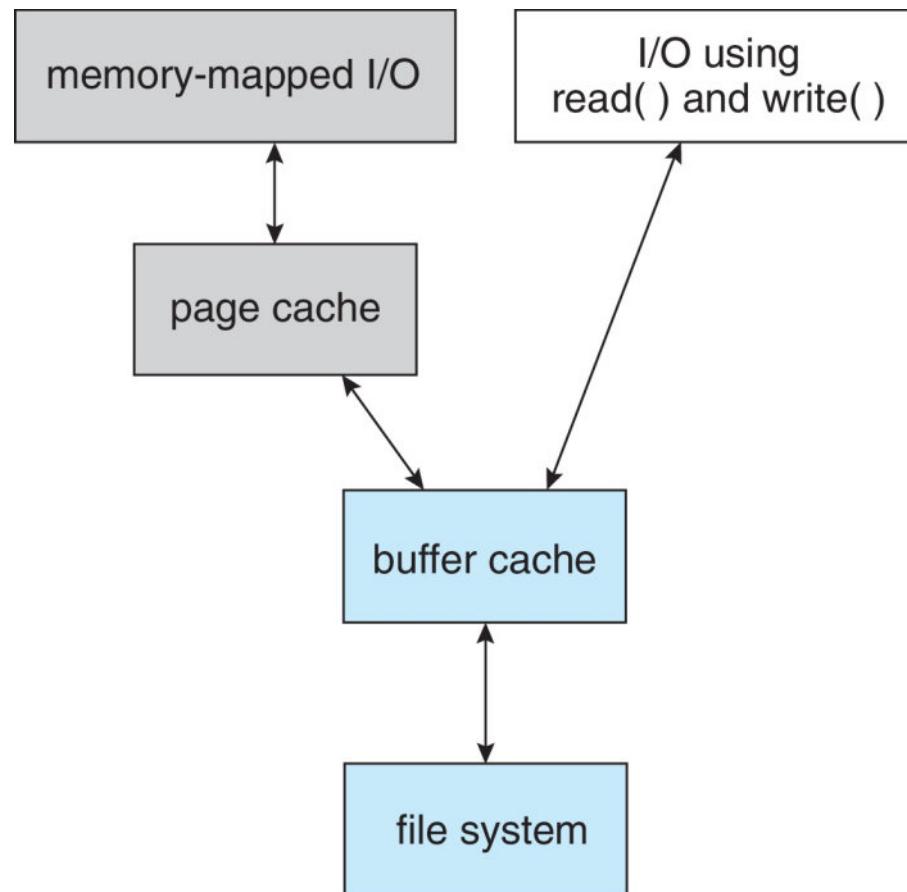


# Page Cache

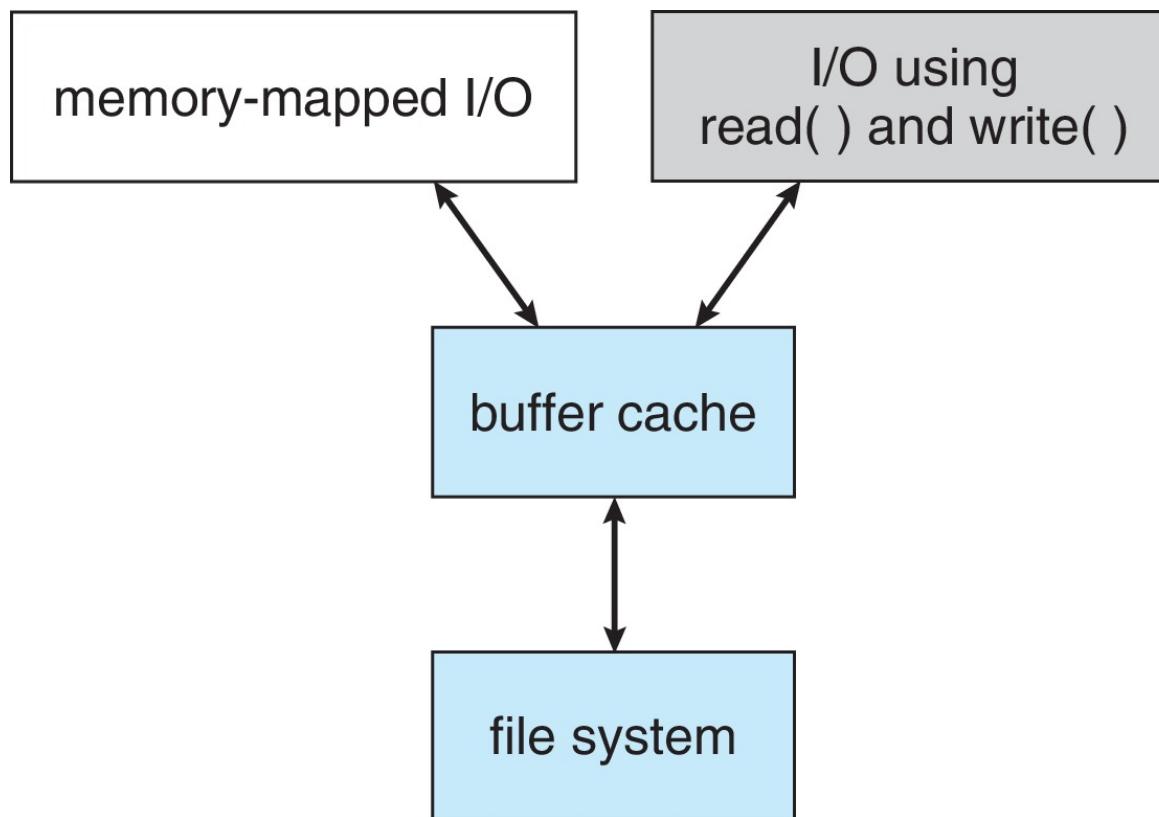
- A page cache caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure



# I/O Without a Unified Buffer Cache



# I/O Using a Unified Buffer Cache



- A *unified buffer cache* uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching
- But which caches get priority, and what replacement algorithms to use?





# Recovery

- *Consistency checking* – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- Use system programs to *back up data* from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by *restoring data from backup*





# Log Structured File Systems

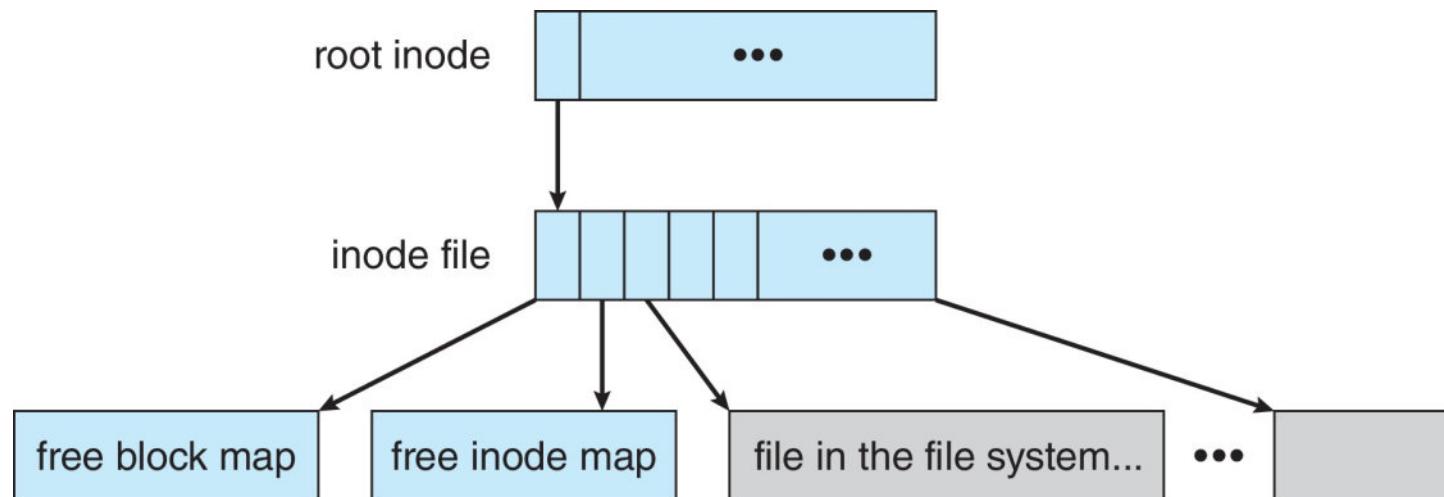
- *Log structured (or journaling) file systems* record each metadata update to the file system as a transaction
  - All transactions are written to a log
    - ▶ A transaction is considered committed once it is written to the log (sequentially)
    - ▶ Sometimes to a separate device or section of disk
    - ▶ However, the file system may not yet be updated
  - The transactions in the log are asynchronously written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata



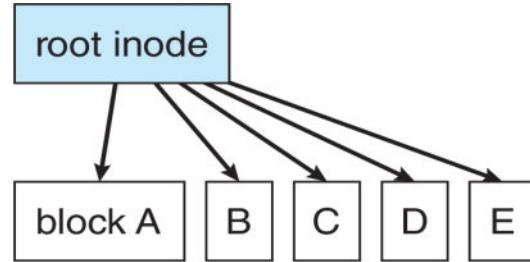
# Example: WAFL File System

## ■ *Write-Anywhere File Layout (WAFL)*

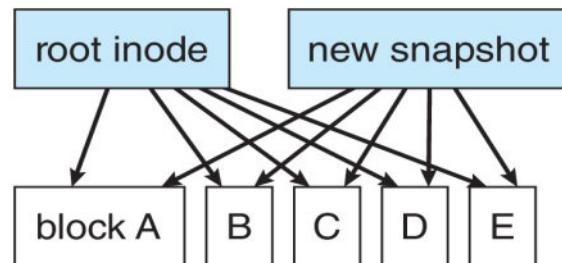
- Used on Network Appliance “Filers” – distributed file system appliances
- Serves up NFS, CIFS, http, ftp
- Random I/O optimized, write optimized
  - ▶ NVRAM for write caching
- Similar to Berkeley Fast File System, with extensive modifications



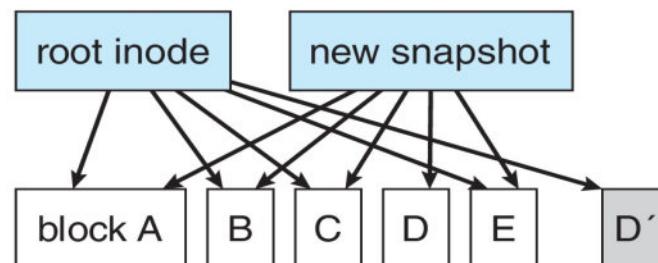
# Snapshots in WAFL



(a) Before a snapshot.



(b) After a snapshot, before any blocks change.



(c) After block D has changed to D'.





# The Apple File System

In 2017, Apple, Inc., released a new file system to replace its 30-year-old HFS+ file system. HFS+ had been stretched to add many new features, but as usual, this process added complexity, along with lines of code, and made adding more features more difficult. Starting from scratch on a blank page allows a design to start with current technologies and methodologies and provide the exact set of features needed.

[Apple File System \(APFS\)](#) is a good example of such a design. Its goal is to run on all current Apple devices, from the Apple Watch through the iPhone to the Mac computers. Creating a file system that works in watchOS, iOS, tvOS, and macOS is certainly a challenge. APFS is feature-rich, including 64-bit pointers, clones for files and directories, snapshots, space sharing, fast directory sizing, atomic safe-save primitives, copy-on-write design, encryption (single- and multi-key), and I/O coalescing. It understands NVM as well as HDD storage.

Most of these features we've discussed, but there are a few new concepts worth exploring. [Space sharing](#) is a ZFS-like feature in which storage is available as one or more large free spaces ([containers](#)) from which file systems can draw allocations (allowing APFS-formatted volumes to grow and shrink).

[Fast directory sizing](#) provides quick used-space calculation and updating.

[Atomic safe-save](#) is a primitive (available via API, not via file-system commands) that performs renames of files, bundles of files, and directories as single atomic operations. I/O coalescing is an optimization for NVM devices in which several small writes are gathered together into a large write to optimize write performance.

Apple chose not to implement RAID as part of the new APFS, instead depending on the existing Apple RAID volume mechanism for software RAID. APFS is also compatible with HFS+, allowing easy conversion for existing deployments.





# Summary

- Most file systems reside on *secondary storage*, which is designed to hold a large amount of data permanently. The most common secondary-storage medium is the *disk*, but the use of *NVM devices* is increasing.
- Storage devices are segmented into *partitions* to control media use and to allow *multiple, possibly varying, file systems on a single device*. These file systems are mounted onto a logical file system architecture to make them available for use.
- *File systems* are often implemented in a *layered or modular structure*. The lower levels deal with the physical properties of storage devices and communicating with them. Upper levels deal with symbolic file names and logical properties of files.
- The various files within a file system can be allocated space on the storage device in three ways: through *contiguous*, *linked*, or *indexed allocation*. Contiguous allocation can suffer from external fragmentation. Direct access is very inefficient with linked allocation. Indexed allocation may require substantial overhead for its index block.





## Summary (Cont.)

These algorithms can be optimized in many ways. Contiguous space can be enlarged through extents to increase flexibility and to decrease external fragmentation. Indexed allocation can be done in clusters of multiple blocks to increase throughput and to reduce the number of index entries needed. Indexing in large clusters is similar to contiguous allocation with extents.

- *Free-space allocation methods* also influence the efficiency of disk-space use, the performance of the file system, and the reliability of secondary storage. The methods used include bit vectors and linked lists. Optimizations include grouping, counting, and the FAT, which places the linked list in one contiguous area.
- *Directory-management routines* must consider efficiency, performance, and reliability. A hash table is a commonly used method, as it is fast and efficient. Unfortunately, damage to the table or a system crash can result in inconsistency between the directory information and the disk's contents.





## Summary (Cont.)

- A *consistency checker* can be used to repair damaged file-system structures. Operating-system backup tools allow data to be copied to magnetic tape or other storage devices, enabling the user to recover from data loss or even entire device loss due to hardware failure, operating system bug, or user error.
- Due to the fundamental role that file systems play in system operation, their *performance* and *reliability* are crucial. Techniques such as log structures and caching help improve performance, while log structures and RAID improve reliability. The WAFL file system is an example of optimization of performance to match a specific I/O load.



# End of Chapter 14



What Is an  
**OPERATING SYSTEM (OS)**  
and How Does It Work

CLEVERISM.COM