

CS2042 - Operating Systems

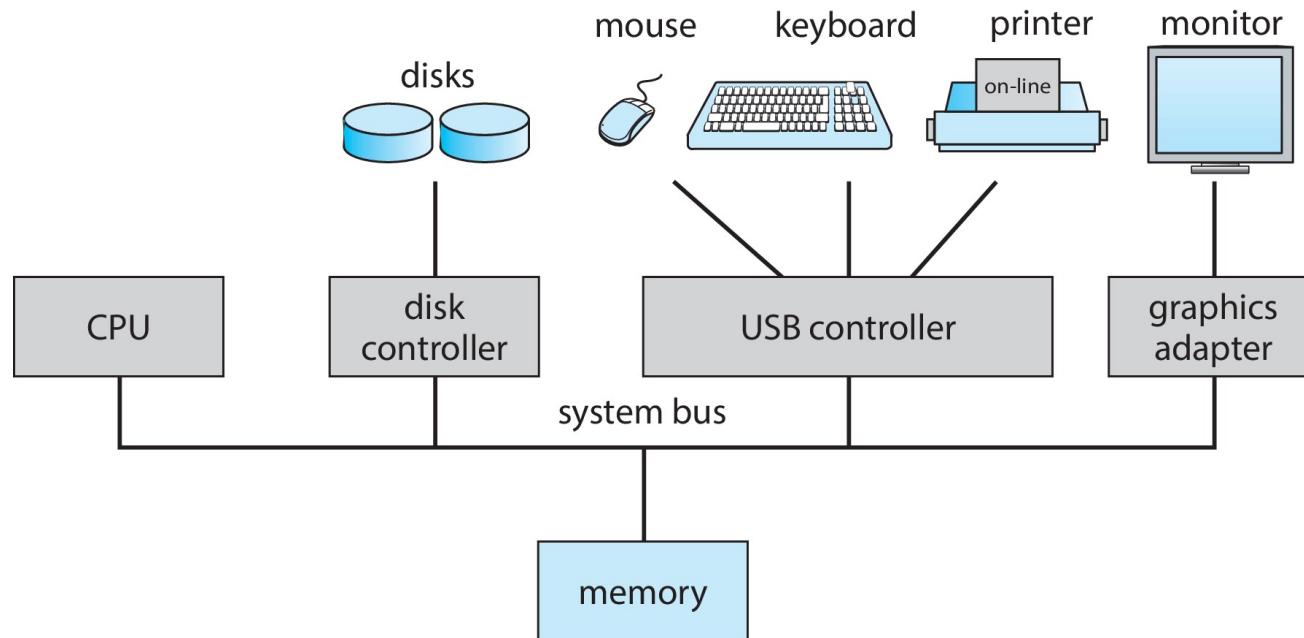
Vishaka Nanayakkara

Module Code	CS 2042	Module Title	Operating Systems			
Credits	2.5	Hours/Week	Lectures	2	Pre – requisites	CS1032
GPA/NGPA	GPA		Lab/Assignments	3/2		



Computer System Organization

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



Components of a Modern Computer (1 of 2)

- One or more processors
- Main memory
- Disks
- Printers
- Keyboard
- Mouse
- Display
- Network interfaces
- I/O devices

Components of a Modern Computer (2 of 2)

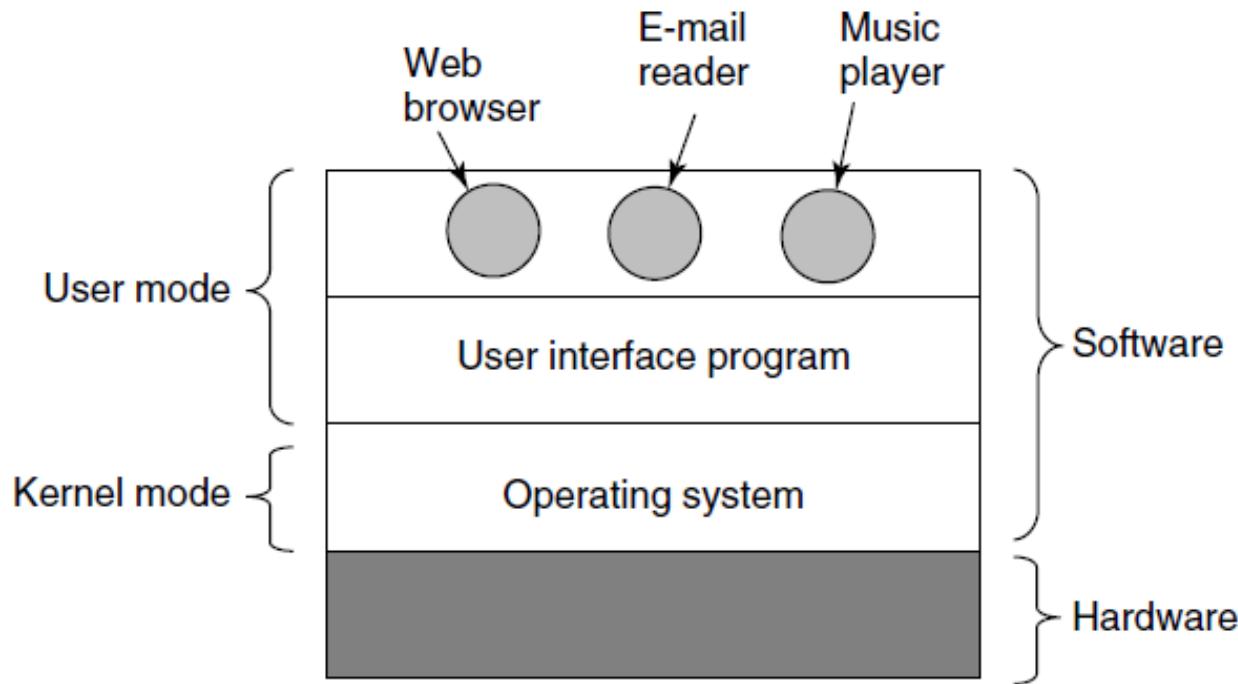


Figure 1-1. Where the operating system fits in.

The Operating System as an Extended Machine

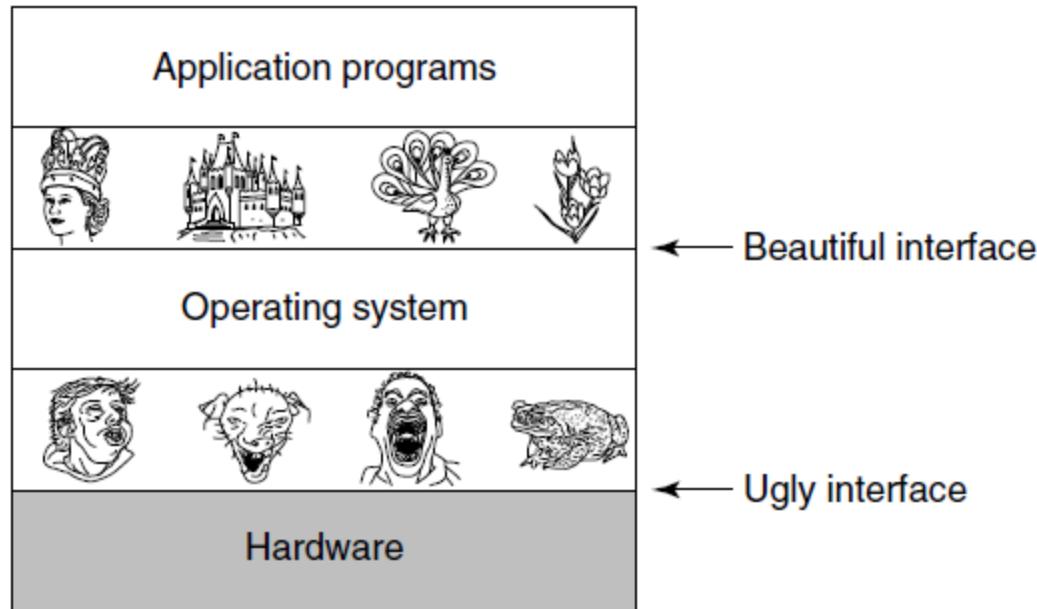


Figure 1-2. Operating systems turn ugly hardware into beautiful abstractions.

Learning Outcomes

- After completing this module, students should be able to
- illustrate concepts underlying design and implementation of contemporary operating systems, and chose and defend decisions on operating system implementation techniques
- develop system programs that simulate operating system behaviour
- apply knowledge about operating system behaviour to develop application programs
- evaluate decisions regarding selection of OSs for specific tasks

Layers of Computer System

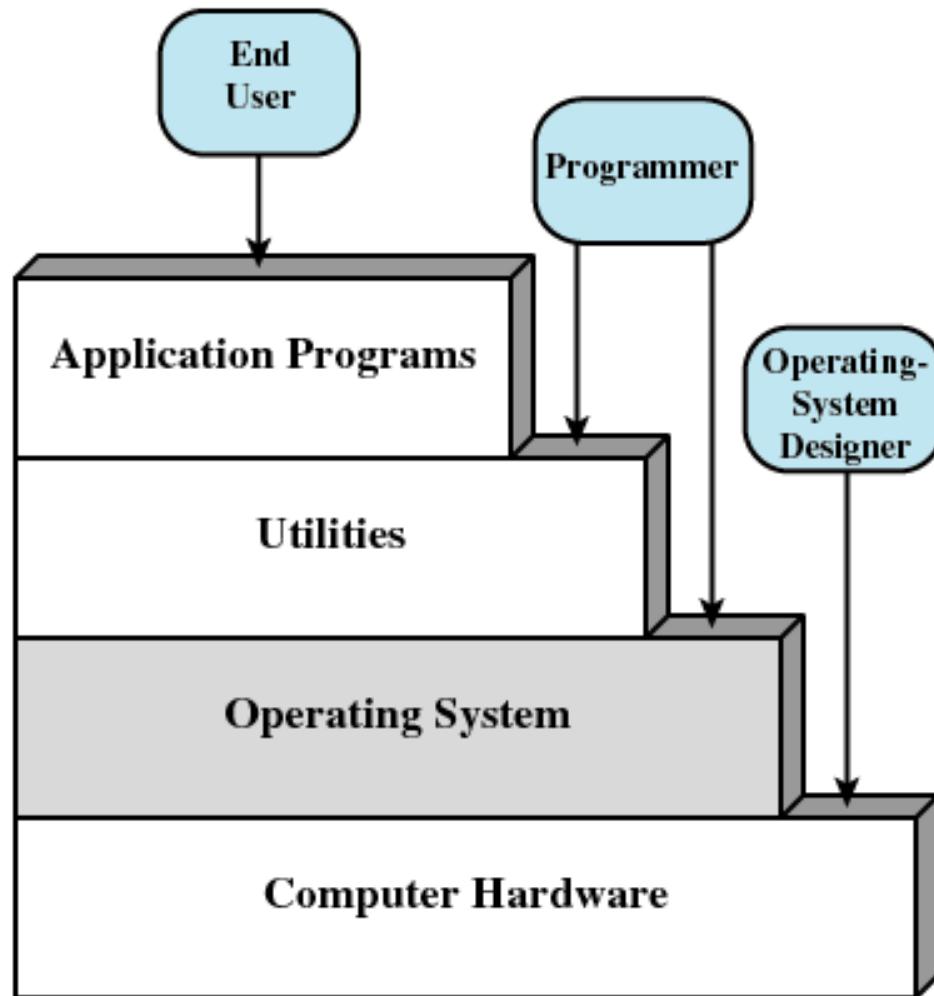
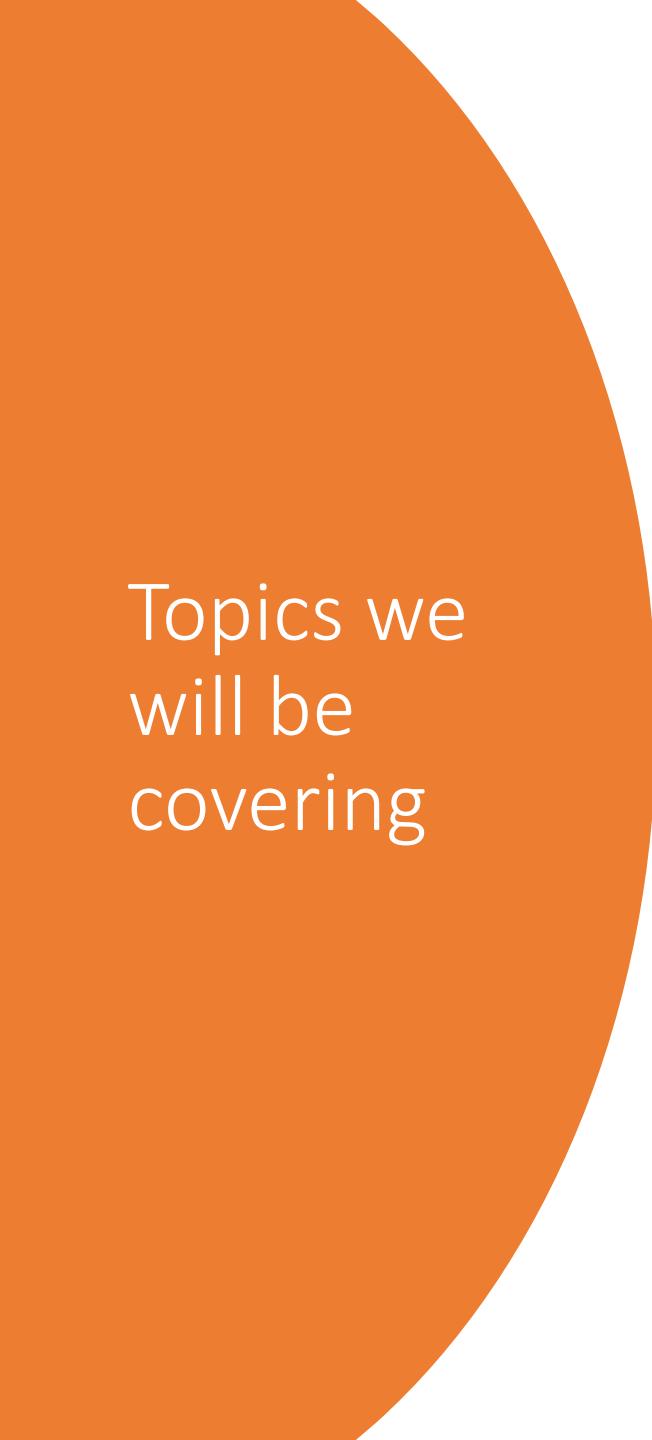
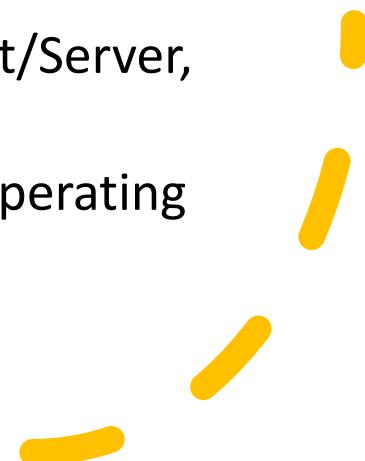


Figure 2.1 Layers and Views of a Computer System



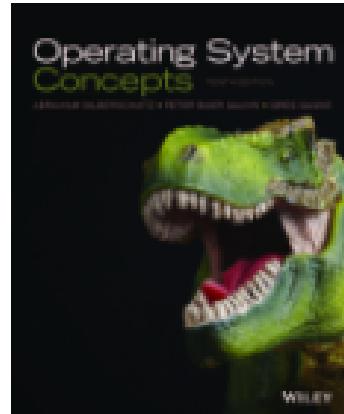
Topics we
will be
covering

1. Operating System Overview
 2. Process Description and Control
 3. Threads
 4. Concurrency: Mutual Exclusion
Synchronization and Deadlocks
 5. Memory Management & Virtual Memory
 6. Uniprocessor, Multiprocessor and Real-
Time Scheduling
 7. I/O Management and Disk Scheduling
 8. File Management
 9. Distributed Processing, Client/Server,
and Clusters
 10. Real –time and Embedded Operating
Systems
- 

Teaching/ Learning Requirements

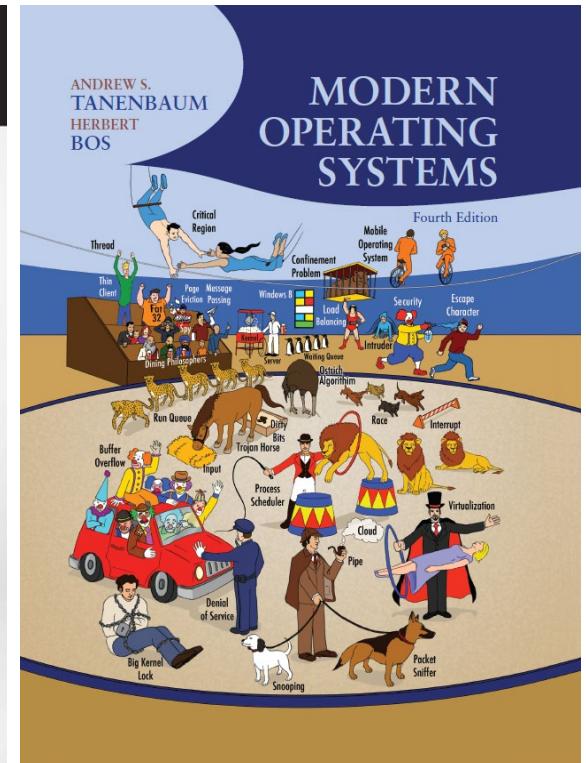
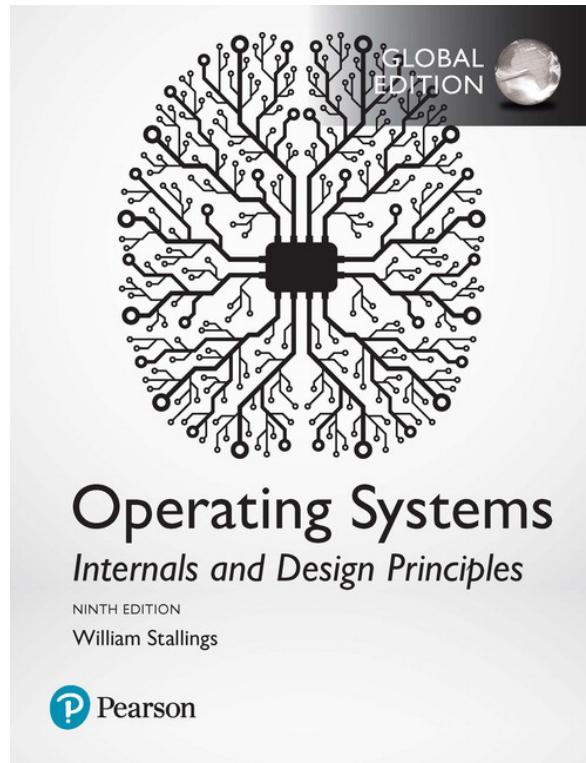
- Online Classes
 - Zoom, recorded lectures, slides and videos, etc
 - Time management
 - Regular work
- Labs and Assignments
 - Own work
 - Peer collaboration
- MPA
 - Multiple components
 - Will give a schedule on Week 3
- Text books

Textbooks



Operating System Concepts *Tenth Edition*

[Avi Silberschatz](#)
[Peter Baer Galvin](#)
[Greg Gagne](#)



Class Rules

Our classroom is a community. In our community, we have rules to help us get along with each other.

Our class rules are:

- Be respectful and responsible.
- Be organized and follow directions.
- Be on time.
- Bring a pen and a paper to take down notes
- Be prepared.
- CA must be your own work

Collaboration

- Students can learn from each other.
- We encourage you to help each other
 - In understanding the materials and, to learn from each other in doing the labs and tutorials.
- Yet, there is a clear distinction between cooperation and cheating.
 - Copying other's answers, code or designs is strictly prohibited.
 - We will punish transgressors severely.

What do you want to know about operating systems?



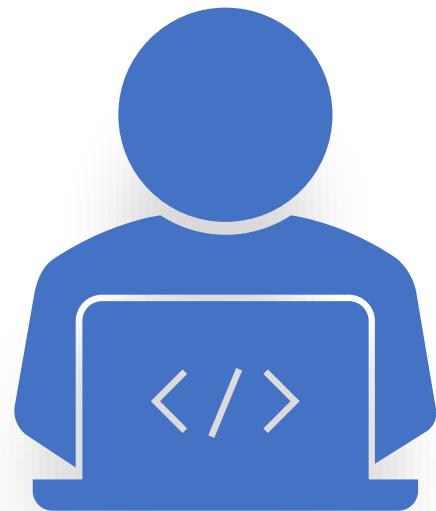
Take a minute to think



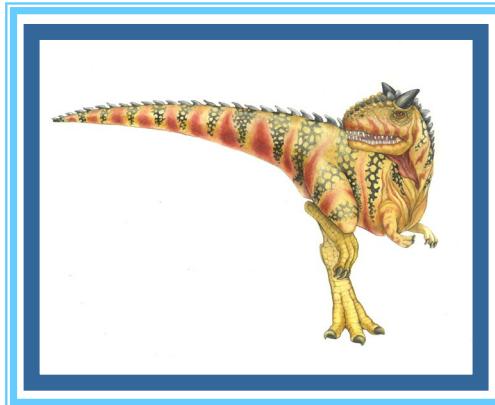
Write down on the chat

"If you want to travel around the world and be invited to speak at a lot of different places, just write a Unix operating system."

~ Linus Torvalds



Chapter 1: Introduction





Chapter 1: Introduction

- What Operating Systems Do
- Computer-System Organization
- Computer-System Architecture
- Operating-System Operations
- Resource Management
- Security and Protection
- Virtualization
- 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





What Does the Term Operating System Mean?

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





What is an Operating System?

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





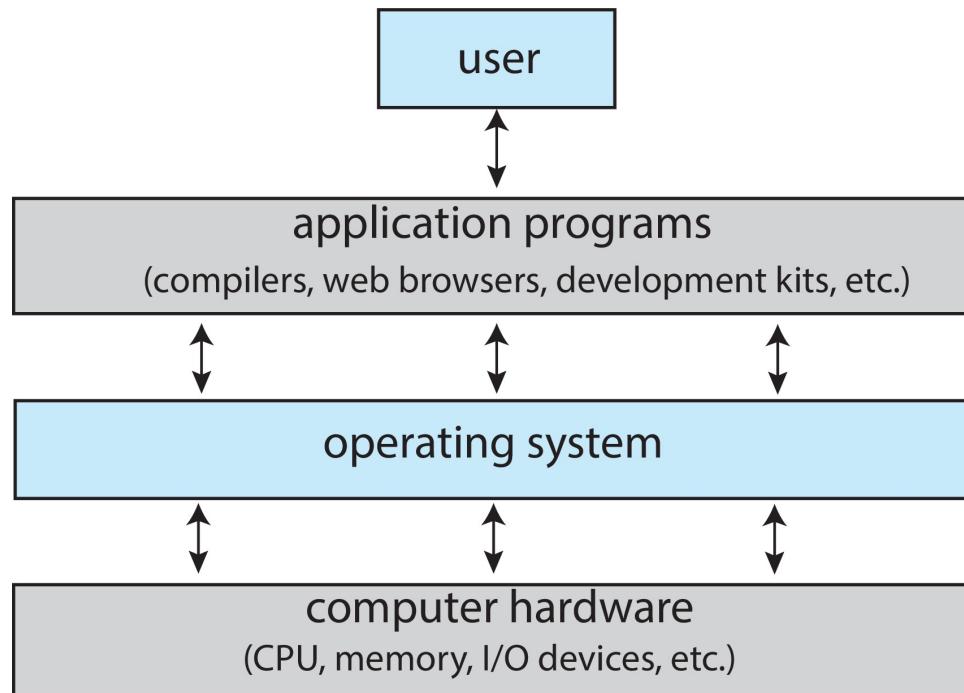
Computer System Structure

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





Abstract View of Components of Computer





What Operating Systems Do

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





Defining Operating Systems

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





Operating System Definition

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





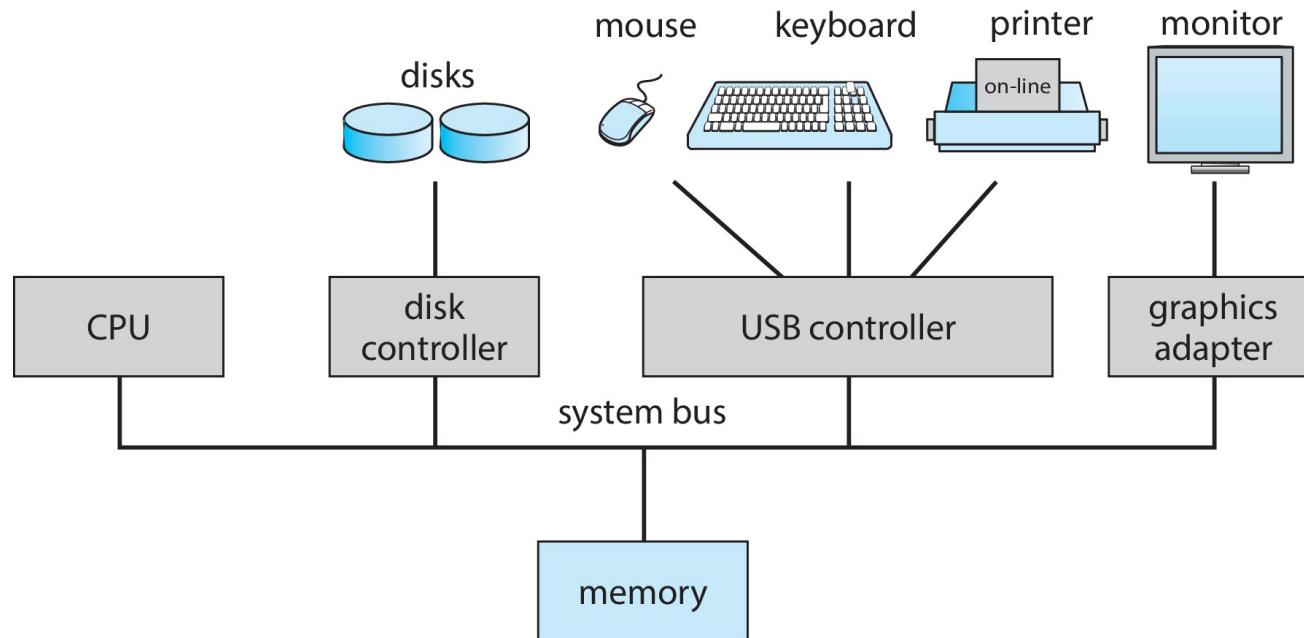
Overview of Computer System Structure





Computer System Organization

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





Computer-System Operation

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





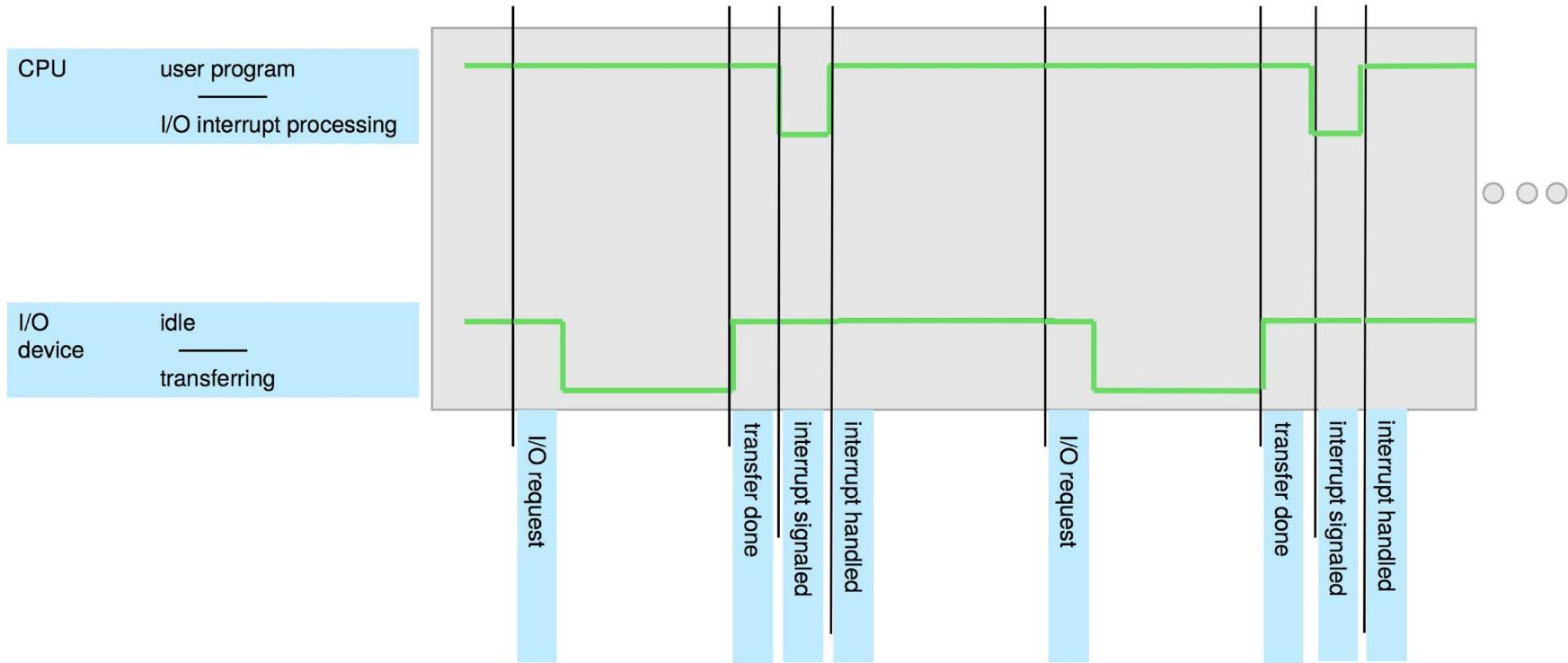
Common Functions of Interrupts

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





Interrupt Timeline





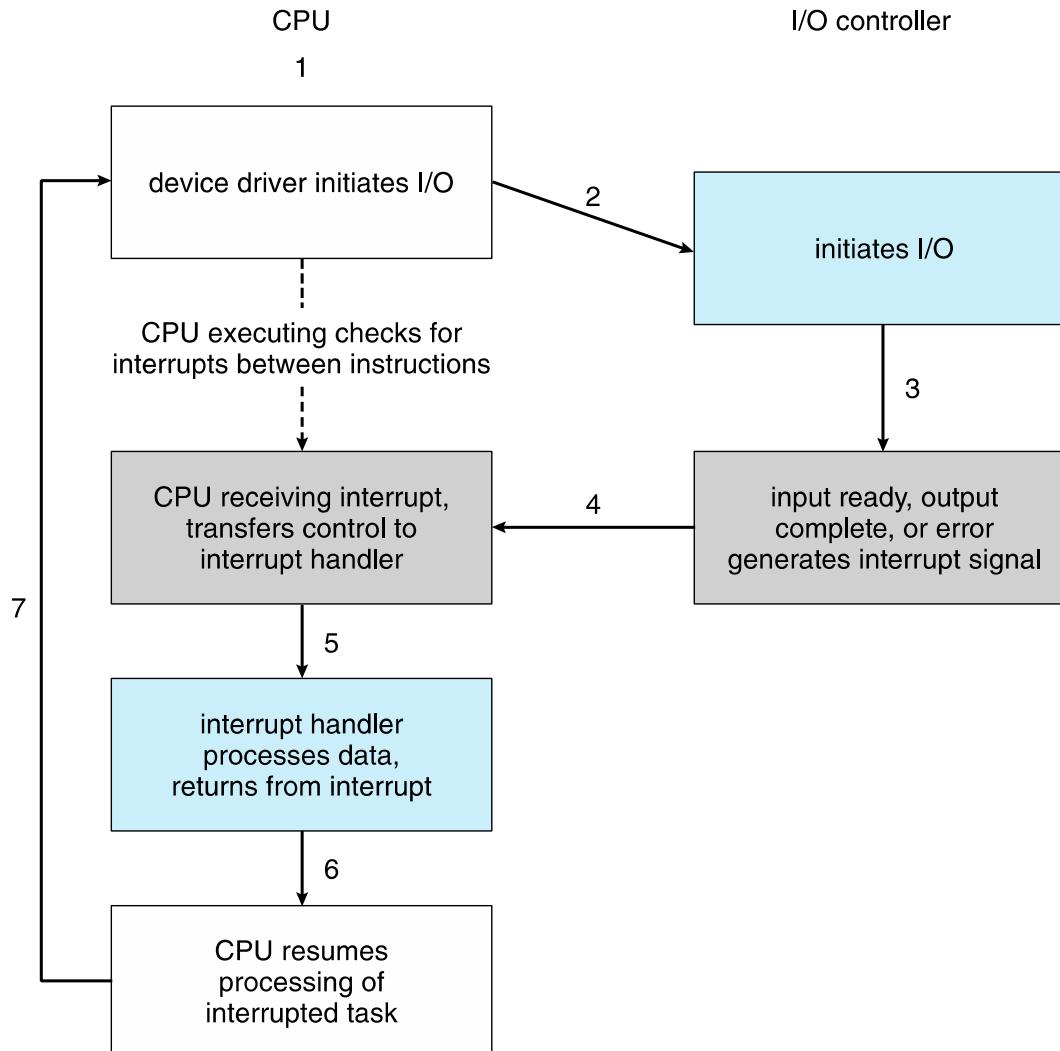
Interrupt Handling

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





Interrupt-drive I/O Cycle





I/O Structure

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





I/O Structure (Cont.)

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





Storage Structure





Storage Structure

- Main memory – only large storage media that the CPU can access directly
 - **Random access**
 - Typically **volatile**
 - Typically **random-access memory** in the form of **Dynamic Random-access Memory (DRAM)**
- Secondary storage – extension of main memory that provides large **nonvolatile** storage capacity

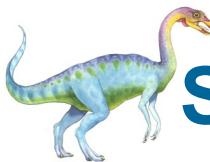




Storage Structure (Cont.)

- **Hard Disk Drives (HDD)** – rigid metal or glass platters covered with magnetic recording material
 - Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
 - The **disk controller** determines the logical interaction between the device and the computer
- **Non-volatile memory (NVM)** devices– faster than hard disks, nonvolatile
 - Various technologies
 - Becoming more popular as capacity and performance increases, price drops

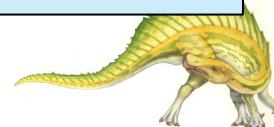




Storage Definitions and Notation Review

The basic unit of computer storage is the **bit**. A bit can contain one of two values, 0 and 1. All other storage in a computer is based on collections of bits. Given enough bits, it is amazing how many things a computer can represent: numbers, letters, images, movies, sounds, documents, and programs, to name a few. A **byte** is 8 bits, and on most computers it is the smallest convenient chunk of storage. For example, most computers don't have an instruction to move a bit but do have one to move a byte. A less common term is **word**, which is a given computer architecture's native unit of data. A word is made up of one or more bytes. For example, a computer that has 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words. A computer executes many operations in its native word size rather than a byte at a time.

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





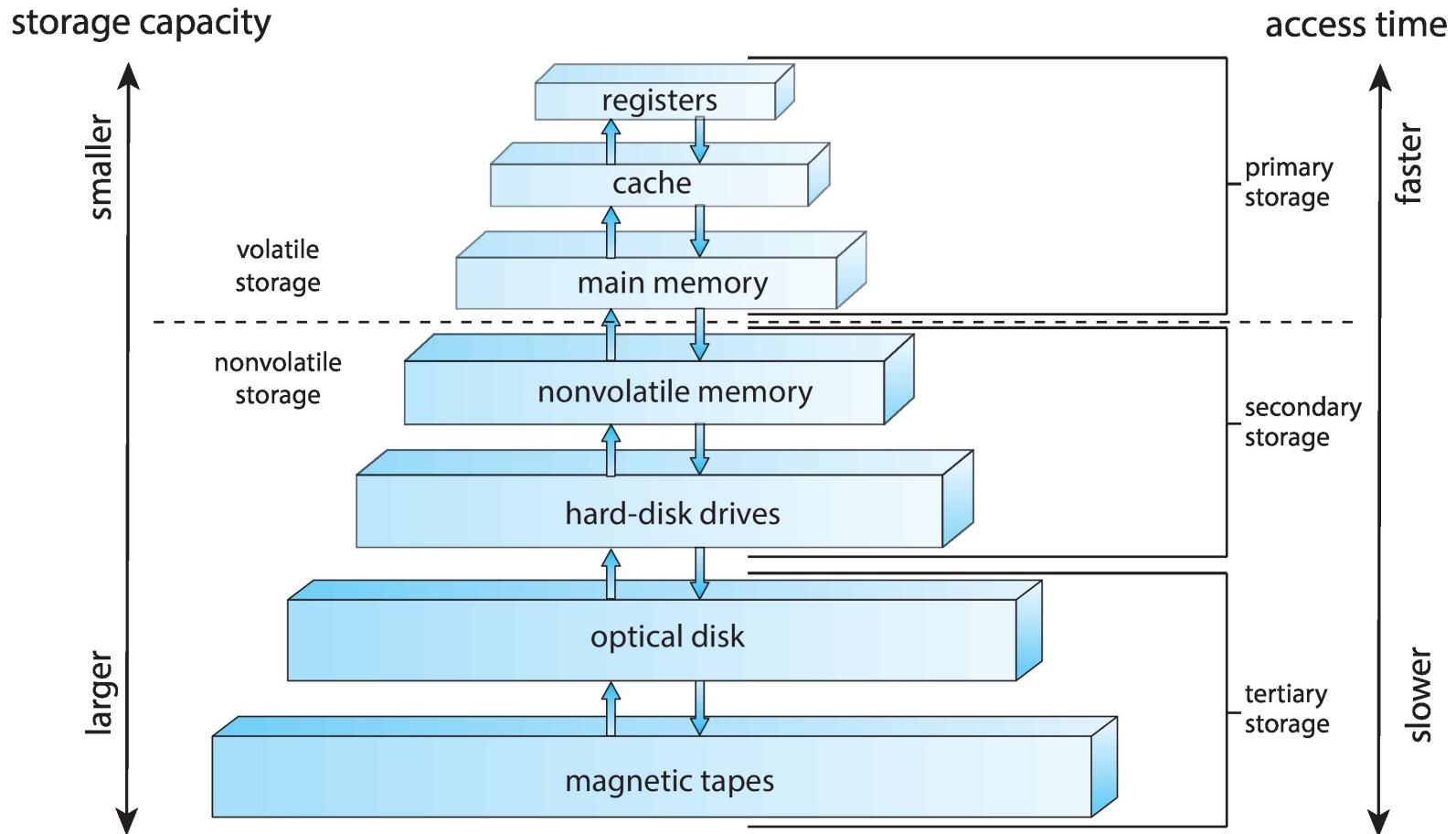
Storage Hierarchy

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



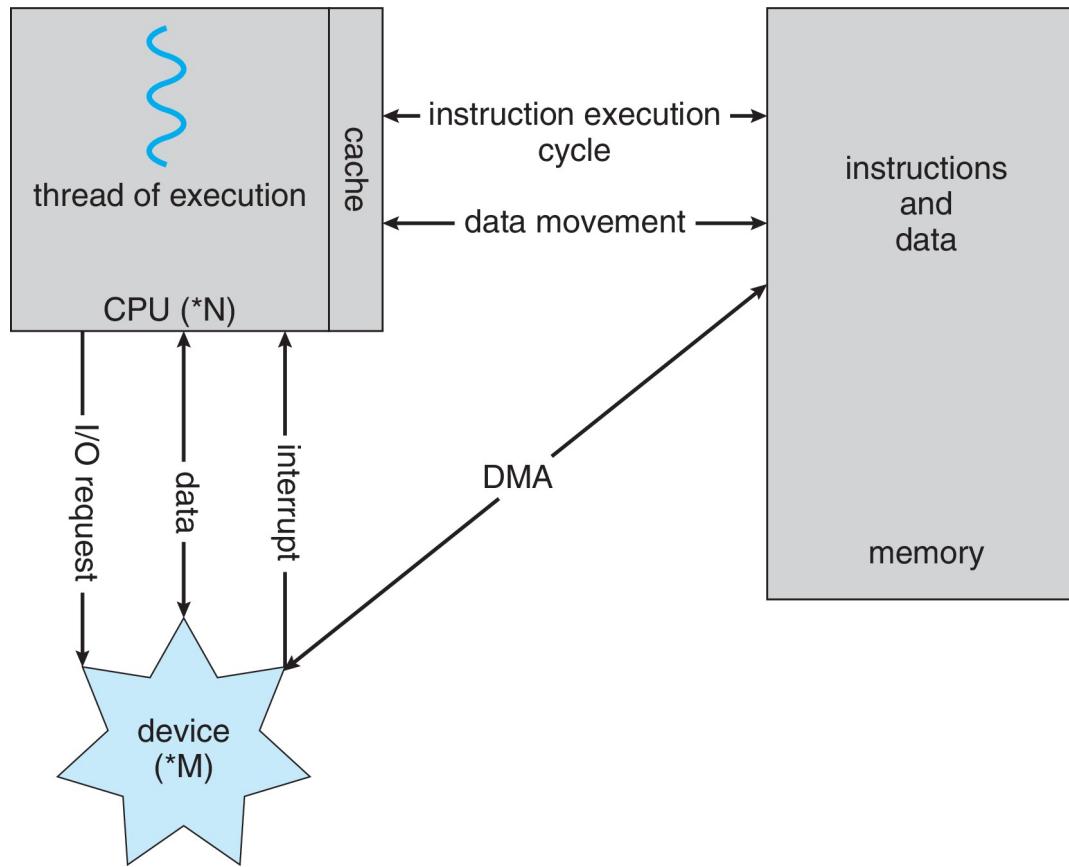


Storage-Device Hierarchy





How a Modern Computer Works



A von Neumann architecture





Direct Memory Access Structure

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





Operating-System Operations

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





Multiprogramming (Batch system)

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





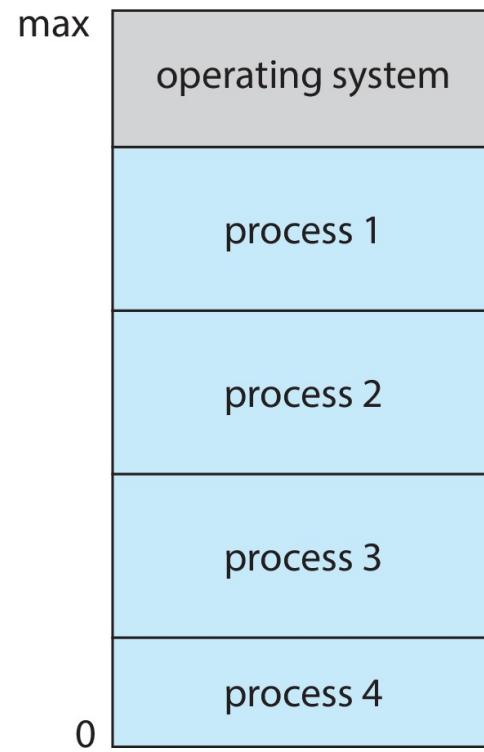
Multitasking (Timesharing)

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





Memory Layout for Multiprogrammed System





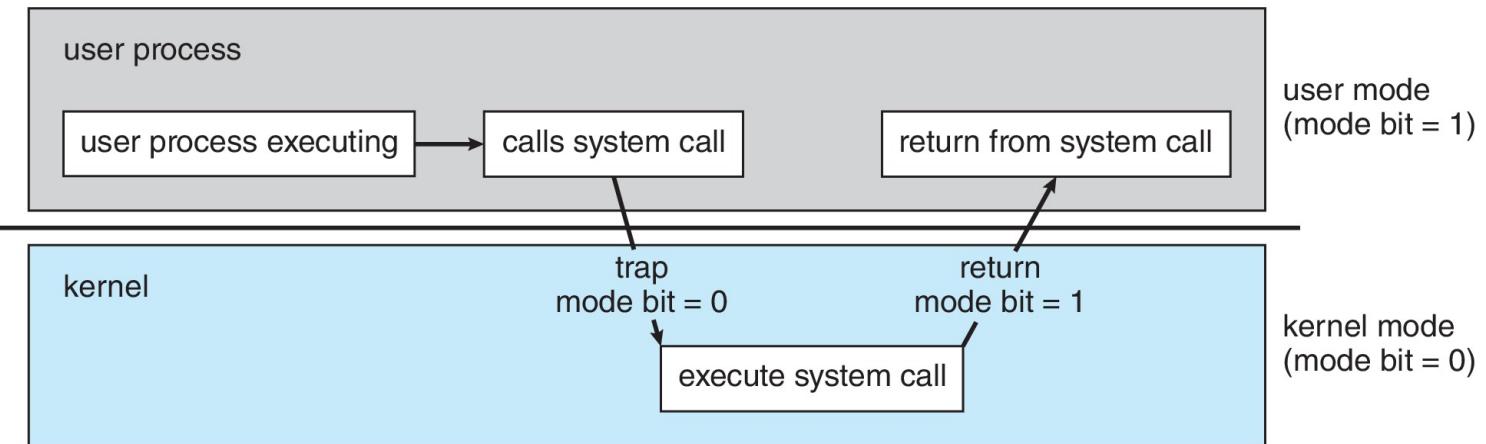
Dual-mode Operation

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





Transition from User to Kernel Mode





Timer

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





Process Management

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





Process Management Activities

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

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





Memory Management

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





File-system Management

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





Mass-Storage Management

- Usually disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time
- Proper management is of central importance
- Entire speed of computer operation hinges on disk subsystem and its algorithms
- OS activities
 - Mounting and unmounting
 - Free-space management
 - Storage allocation
 - Disk scheduling
 - Partitioning
 - Protection





Caching

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





Characteristics of Various Types of Storage

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

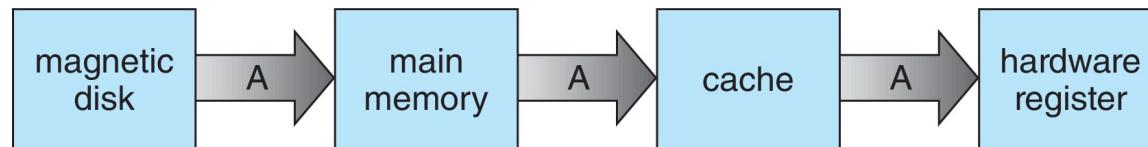
Movement between levels of storage hierarchy can be explicit or implicit





Migration of data “A” from Disk to Register

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



- Multiprocessor environment must provide **cache coherency** in hardware such that all CPUs have the most recent value in their cache
- Distributed environment situation even more complex
 - Several copies of a datum can exist
 - Various solutions covered in Chapter 19





I/O Subsystem

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





Protection and Security

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





Virtualization

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





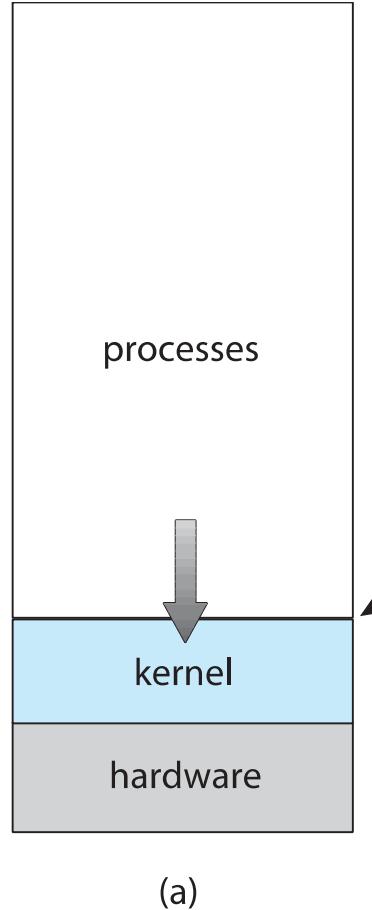
Virtualization (cont.)

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





Computing Environments - Virtualization



(a)



achine





Distributed Systems

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





Computer System Architecture





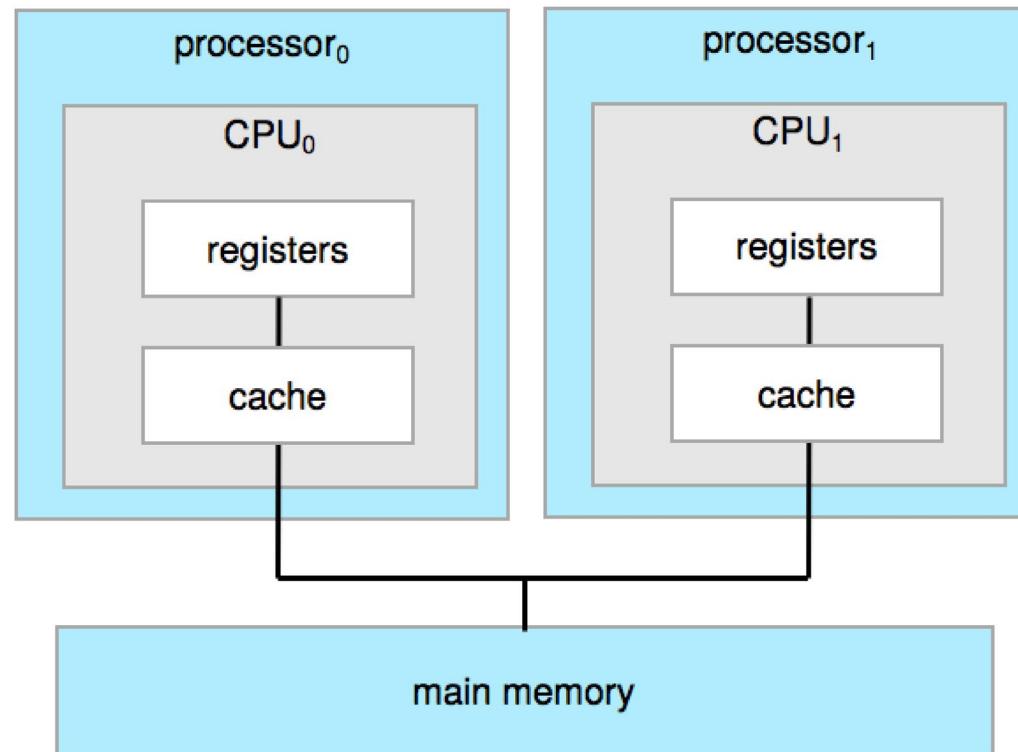
Computer-System Architecture

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





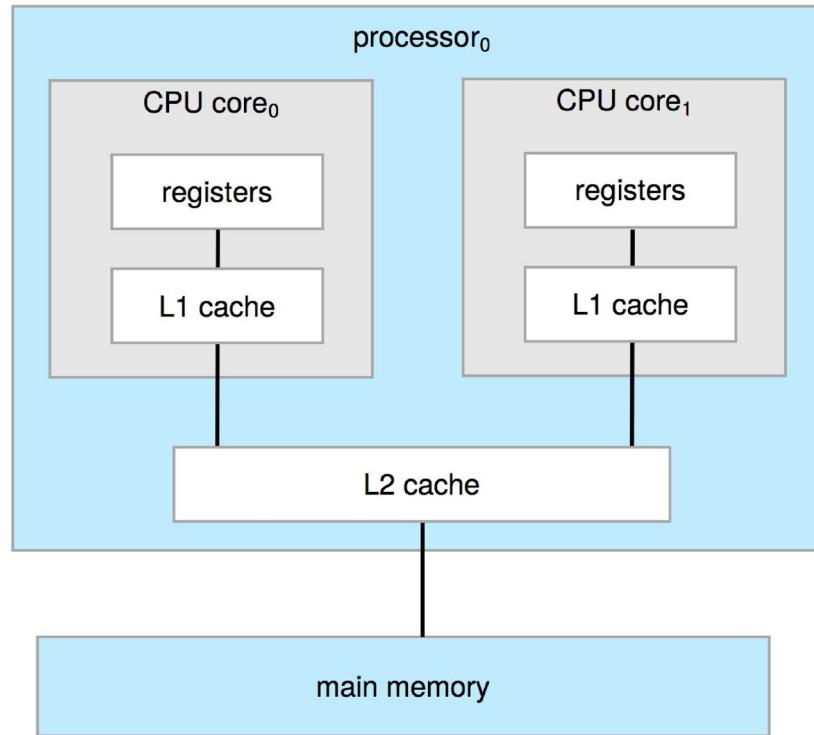
Symmetric Multiprocessing Architecture





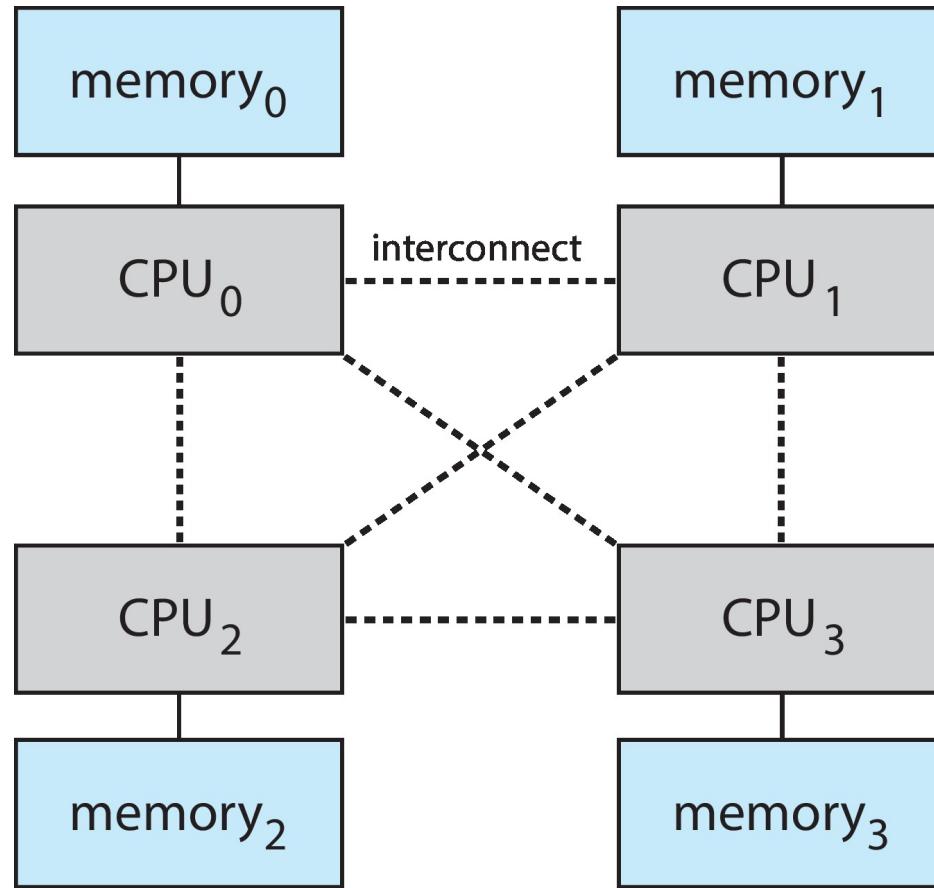
Dual-Core Design

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





Non-Uniform Memory Access System

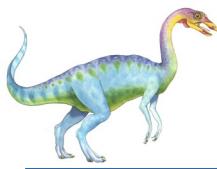




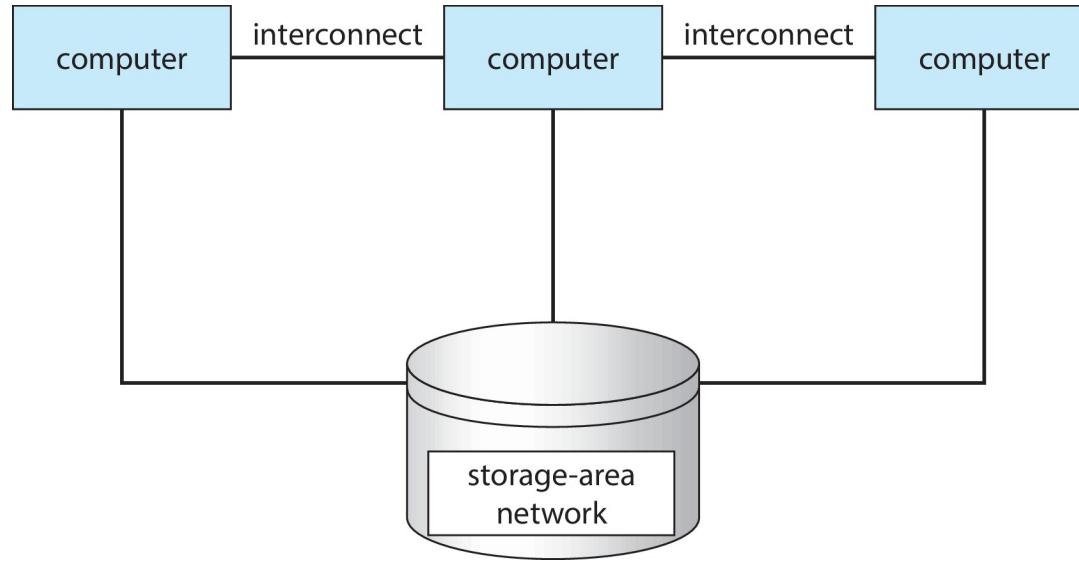
Clustered Systems

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





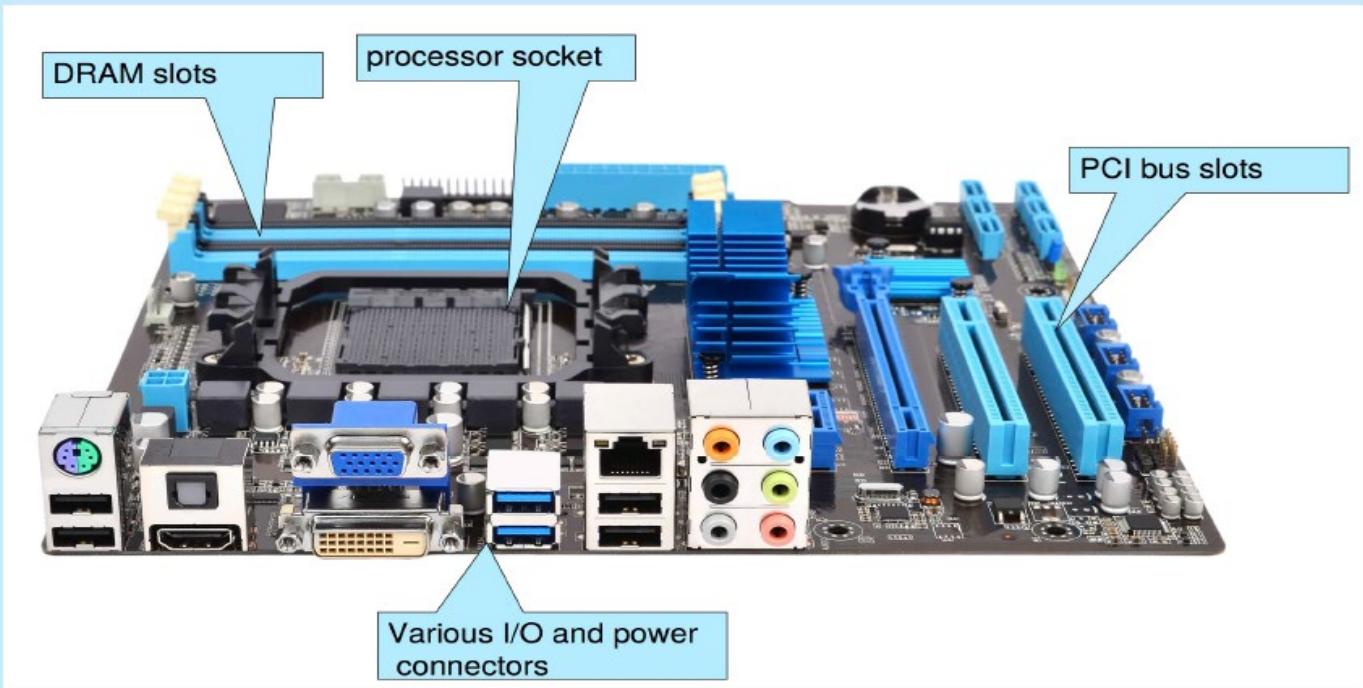
Clustered Systems





PC Motherboard

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



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





Computer System Environments





Computing Environments

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





Traditional

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





Mobile

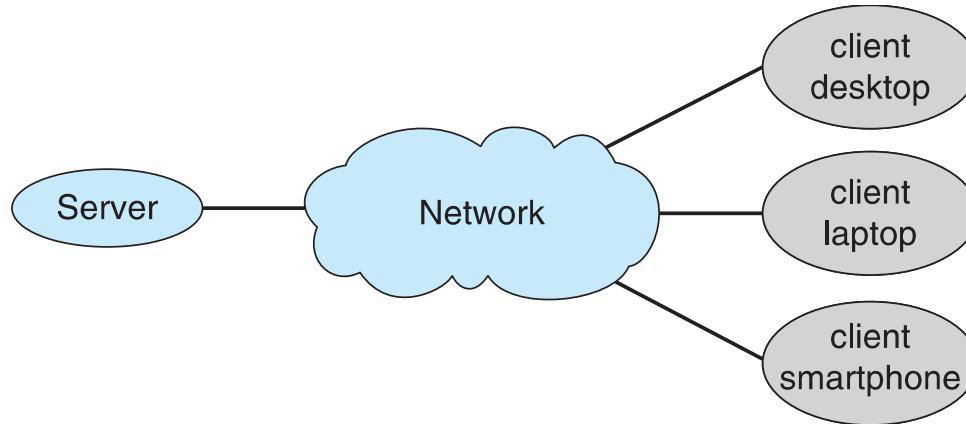
- Handheld smartphones, tablets, etc.
- What is the functional difference between them and a “traditional” laptop?
- Extra feature – more OS features (GPS, gyroscope)
- Allows new types of apps like ***augmented reality***
- Use IEEE 802.11 wireless, or cellular data networks for connectivity
- Leaders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





Client Server

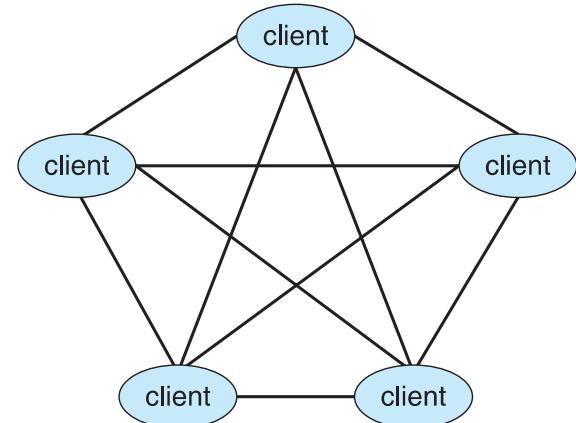
- Client-Server Computing
 - Dumb terminals supplanted by smart PCs
 - Many systems now **servers**, responding to requests generated by **clients**
 - ▶ **Compute-server system** provides an interface to client to request services (i.e., database)
 - ▶ **File-server system** provides interface for clients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





Peer-to-Peer

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





Cloud Computing

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





Cloud Computing (Cont.)

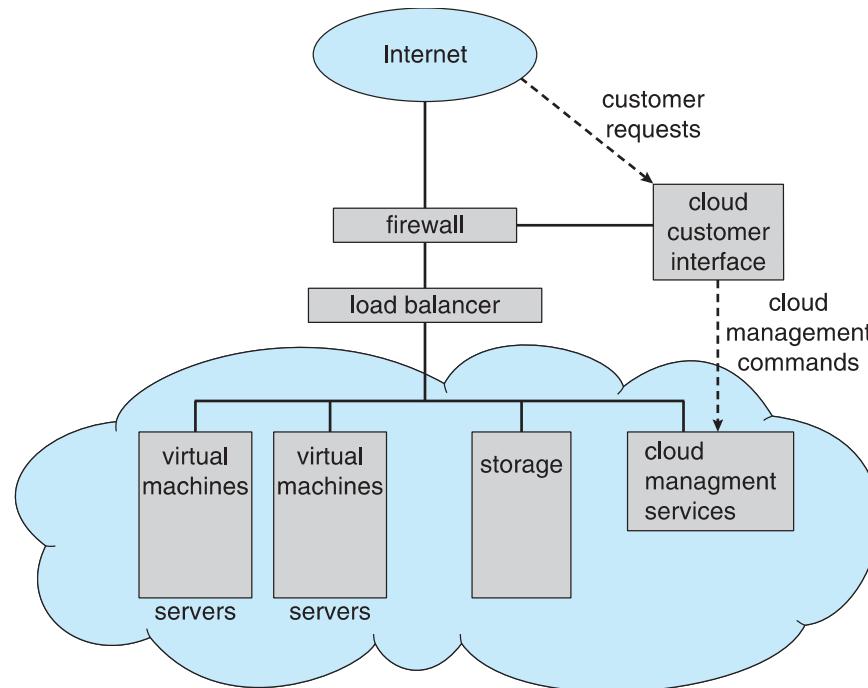
- Many types
 - **Public cloud** – available via Internet to anyone willing to pay
 - **Private cloud** – run by a company for the company's own use
 - **Hybrid cloud** – includes both public and private cloud components
 - Software as a Service (**SaaS**) – one or more applications available via the Internet (i.e., word processor)
 - Platform as a Service (**PaaS**) – software stack ready for application use via the Internet (i.e., a database server)
 - Infrastructure as a Service (**IaaS**) – servers or storage available over Internet (i.e., storage available for backup use)





Cloud Computing (cont.)

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





Real-Time Embedded Systems

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





Free and Open-Source Operating Systems

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





The Study of Operating Systems

There has never been a more interesting time to study operating systems, and it has never been easier. The open-source movement has overtaken operating systems, causing many of them to be made available in both source and binary (executable) format. The list of operating systems available in both formats includes Linux, 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.





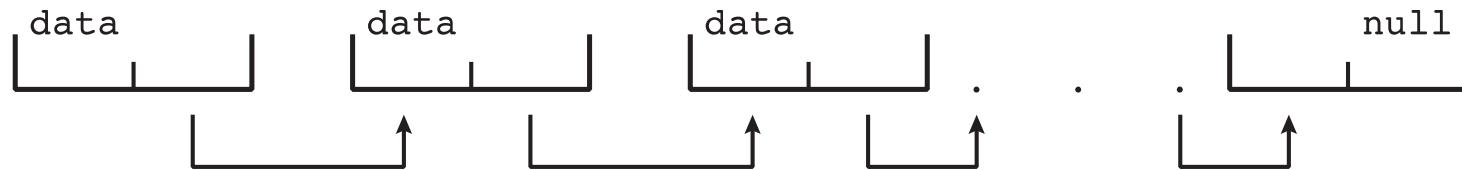
Kernel Data Structure



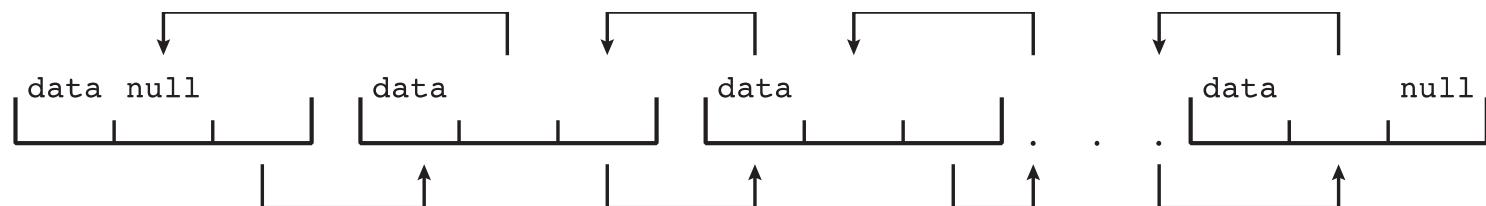


Kernel Data Structures

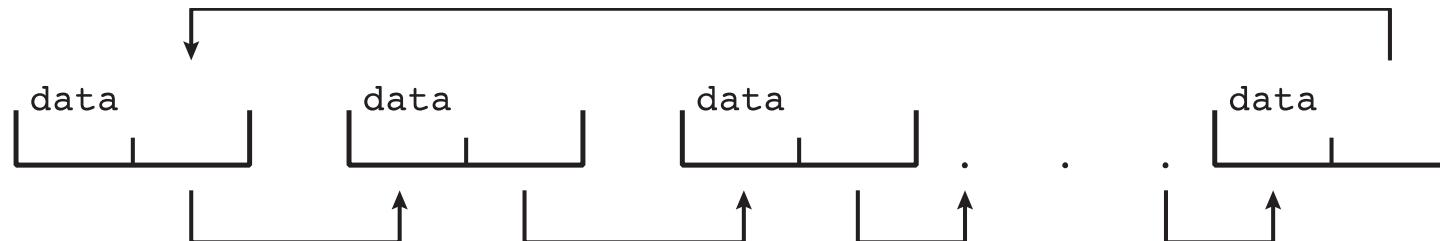
- Many similar to standard programming data structures
- ***Singly linked list***



- ***Doubly linked list***



- ***Circular linked list***



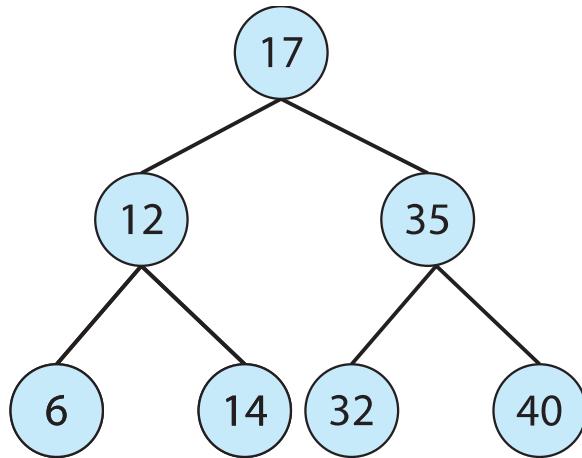


Kernel Data Structures

- **Binary search tree**

left \leq right

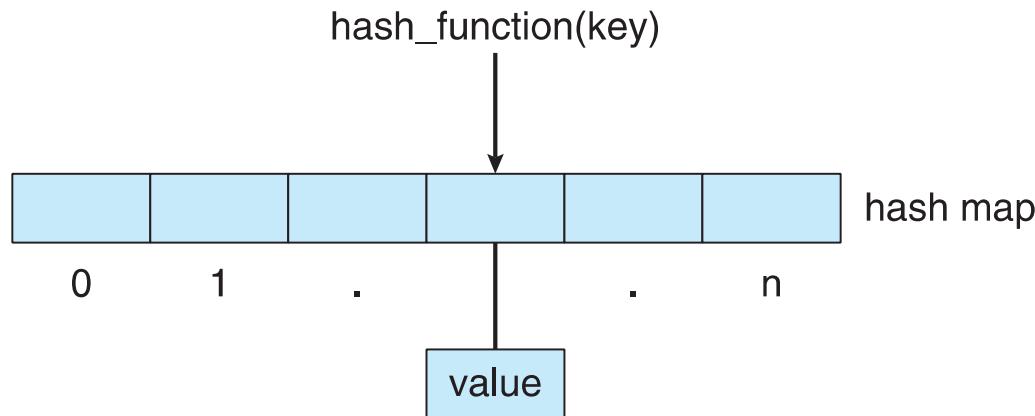
- Search performance is $O(n)$
- **Balanced binary search tree** is $O(\lg n)$





Kernel Data Structures

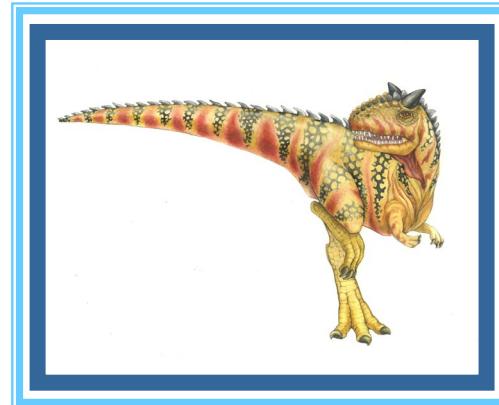
- Hash function can create a hash map



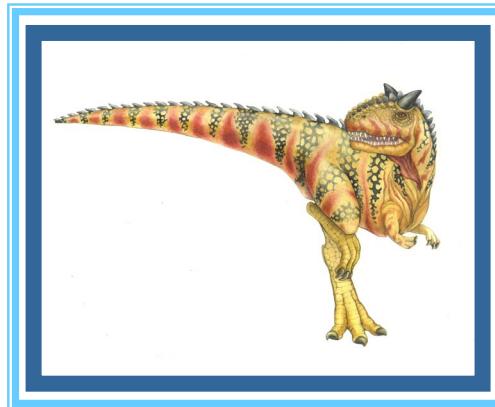
- Bitmap – string of n binary digits representing the status of n items
- Linux data structures defined in **include** files `<linux/list.h>`,
`<linux/kfifo.h>`, `<linux/rbtree.h>`



End of Chapter 1



Chapter 2: Operating-System Services





Outline

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





Objectives

- Identify services provided by an 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
- One set of operating-system services provides functions that are helpful to the user:
 - **User interface** - Almost all operating systems have a user interface (**UI**).
 - ▶ Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **touch-screen**, **Batch**
 - **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
 - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
 - **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file information, permission management.





Operating System Services (Cont.)

- One set of operating-system services provides functions that are helpful to the user (Cont.):
 - **Communications** – Processes may exchange information, on the same computer or between computers over a network
 - ▶ Communications may be via shared memory or through message passing (packets moved by the OS)
 - **Error detection** – OS needs to be constantly aware of possible errors
 - ▶ May occur in the CPU and memory hardware, in I/O devices, in user program
 - ▶ For each type of error, OS should take the appropriate action to ensure correct and consistent computing
 - ▶ Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system





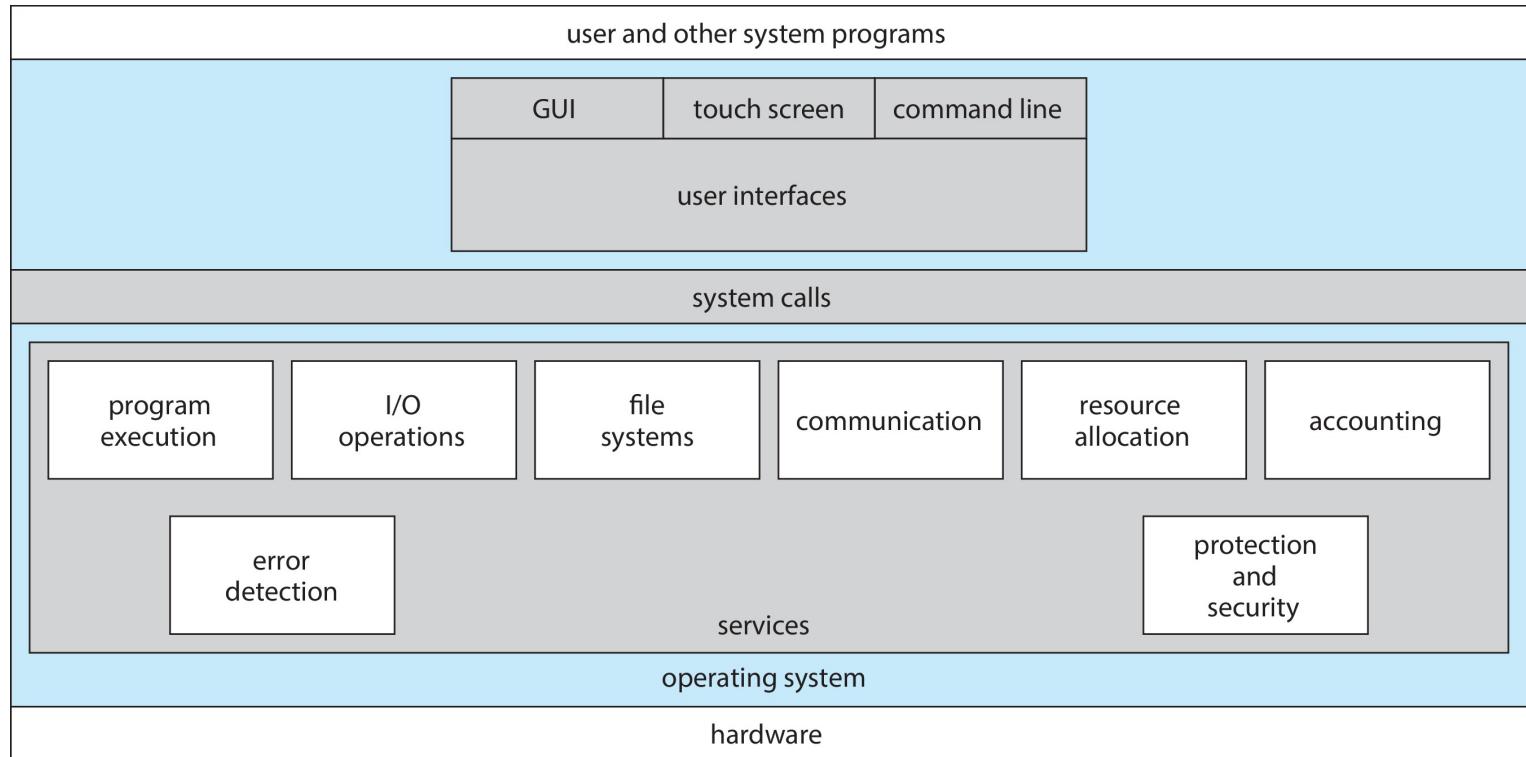
Operating System Services (Cont.)

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





A View of Operating System Services





Command Line interpreter

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





Bourne Shell Command Interpreter

```
1. root@r6181-d5-us01:~ (ssh)
X root@r6181-d5-u... ● ⌘1 X ssh ⚡ ⌘2 X root@r6181-d5-us01... ⚡3

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





User Operating System Interface - GUI

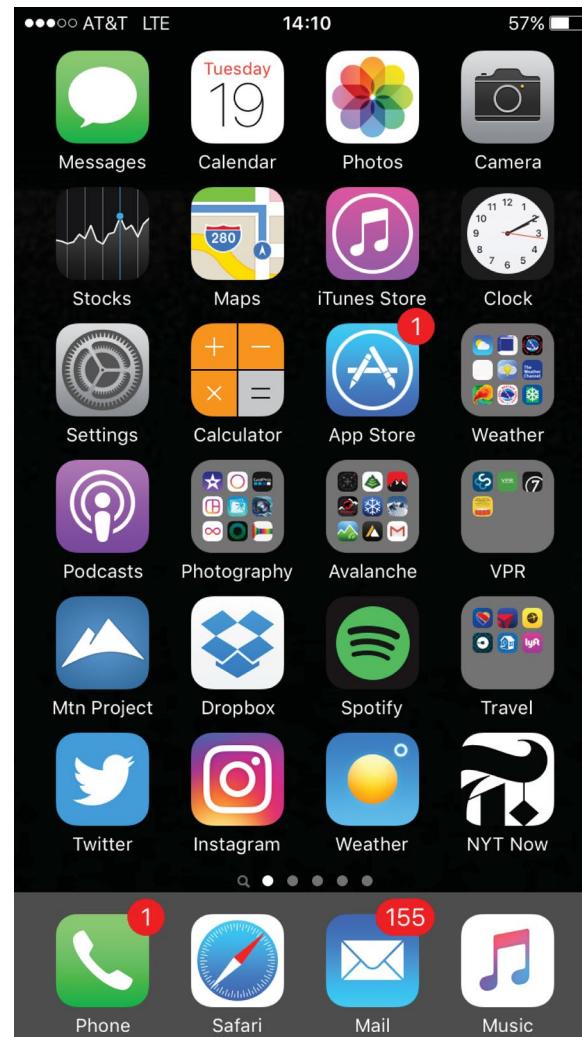
- User-friendly **desktop** metaphor interface
 - Usually mouse, keyboard, and monitor
 - **Icons** represent files, programs, actions, etc
 - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)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





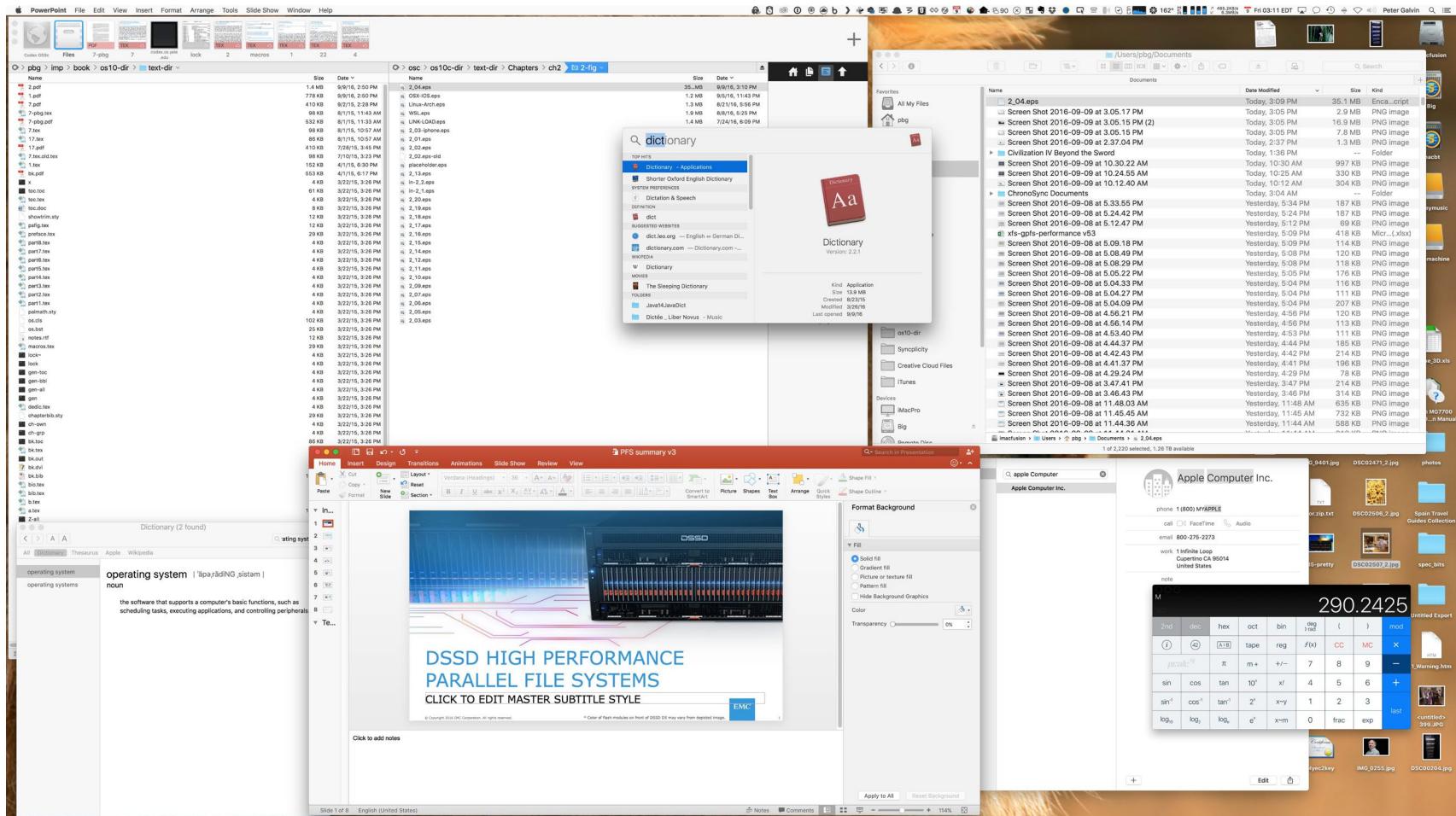
Touchscreen Interfaces

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





The Mac OS X GUI





System Calls

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

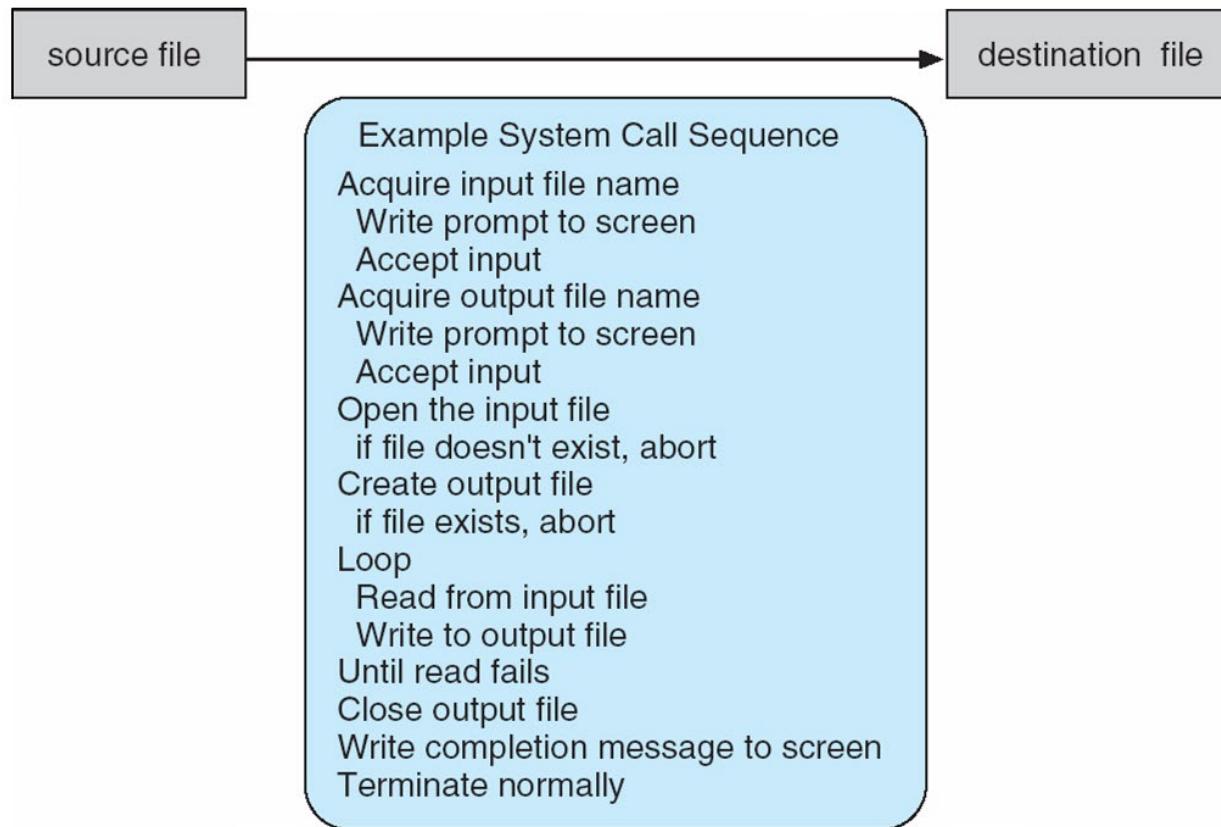
Note that the system-call names used throughout this text are generic





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.





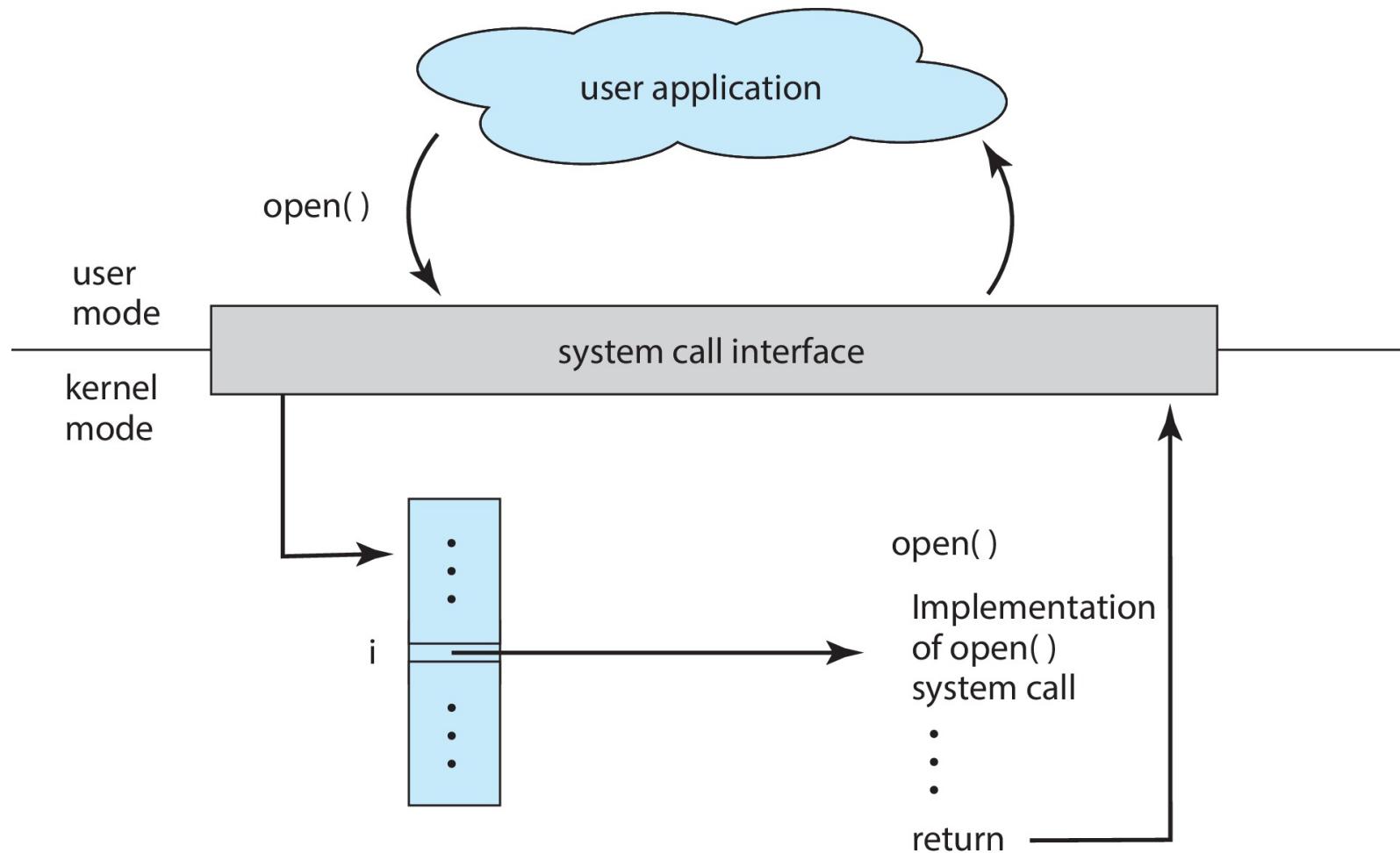
System Call Implementation

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





API – System Call – OS Relationship





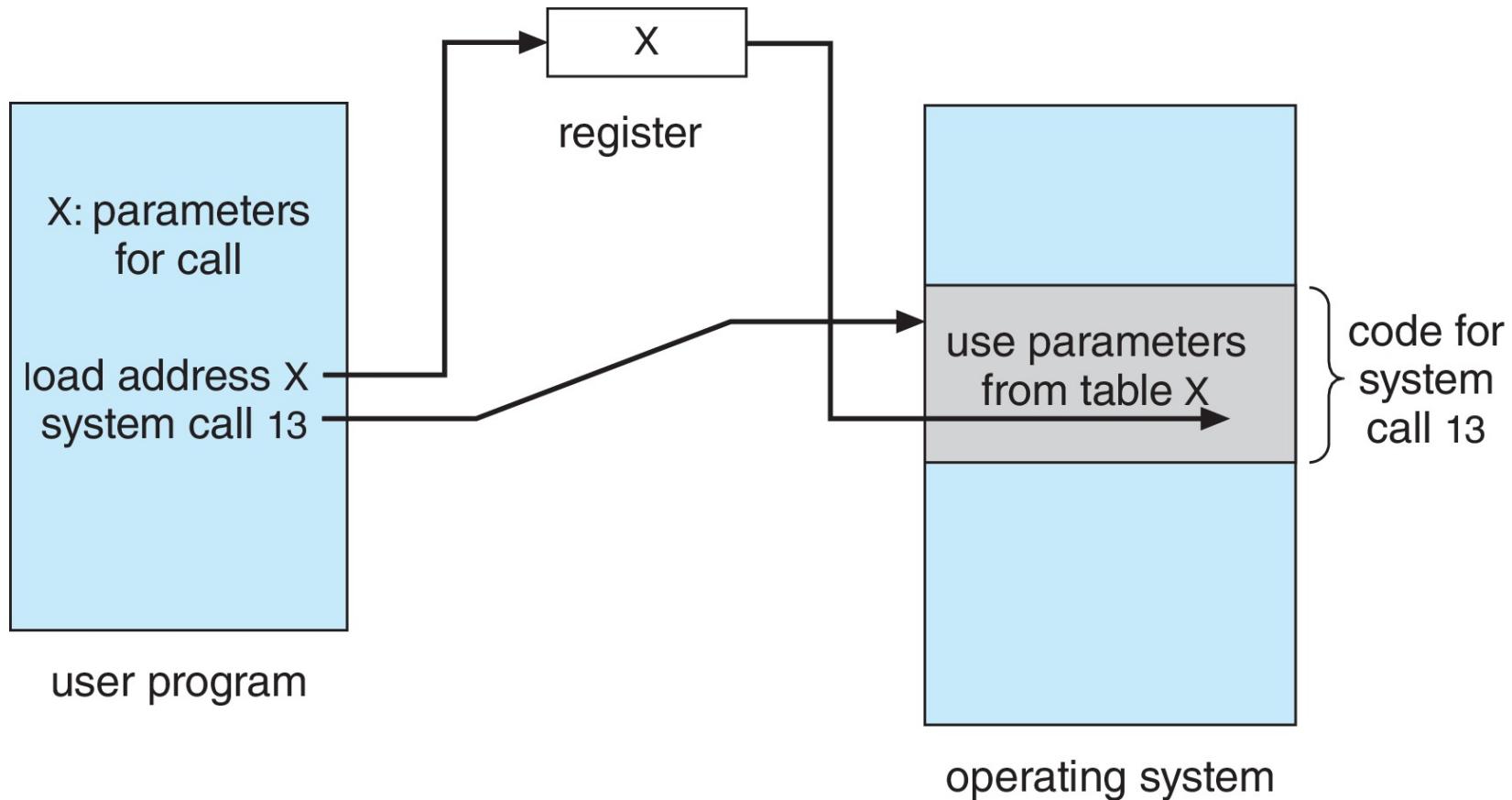
System Call Parameter Passing

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





Parameter Passing via Table





Types of System Calls

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





Types of System Calls (Cont.)

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





Types of System Calls (Cont.)

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





Types of System Calls (Cont.)

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





Examples of Windows and Unix System Calls

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

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

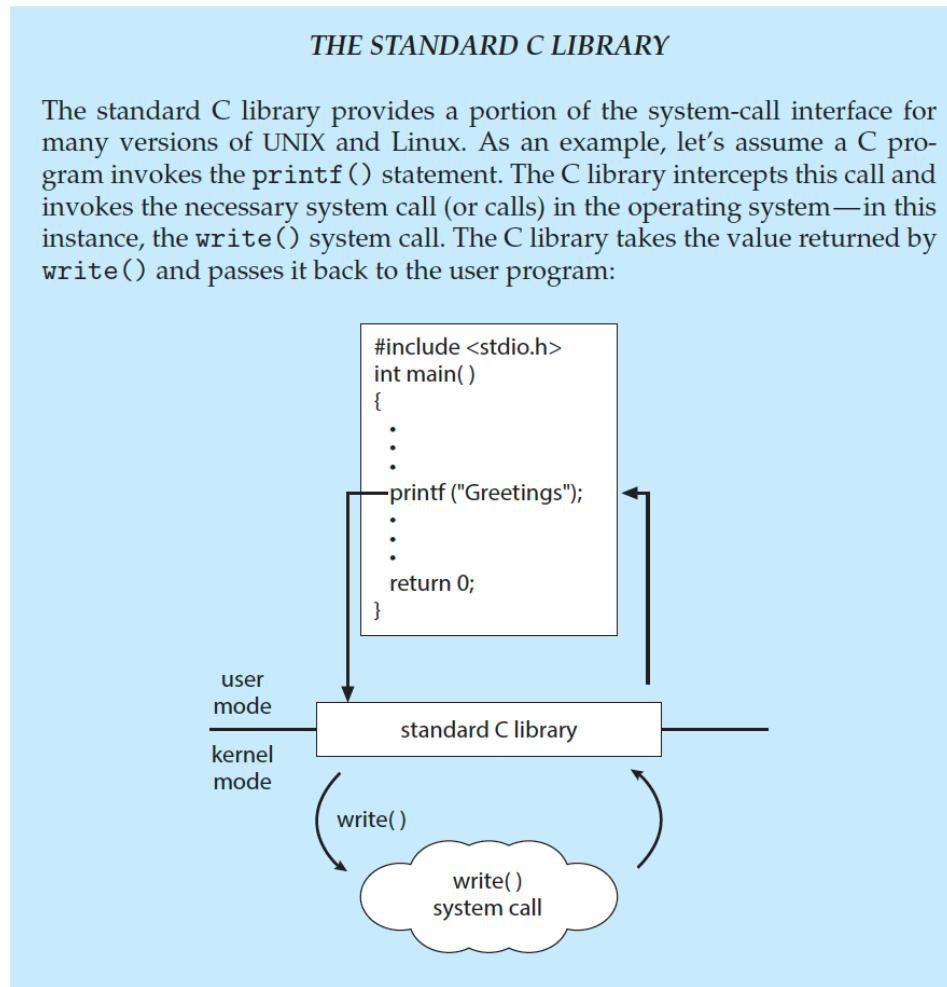
	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance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





Standard C Library Example

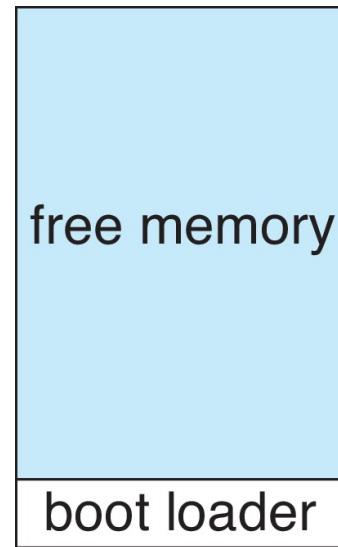
- C program invoking printf() library call, which calls write() system call





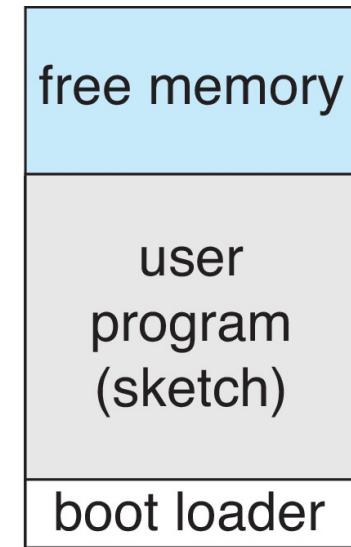
Example: Arduino

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



(a)

At system startup



(b)

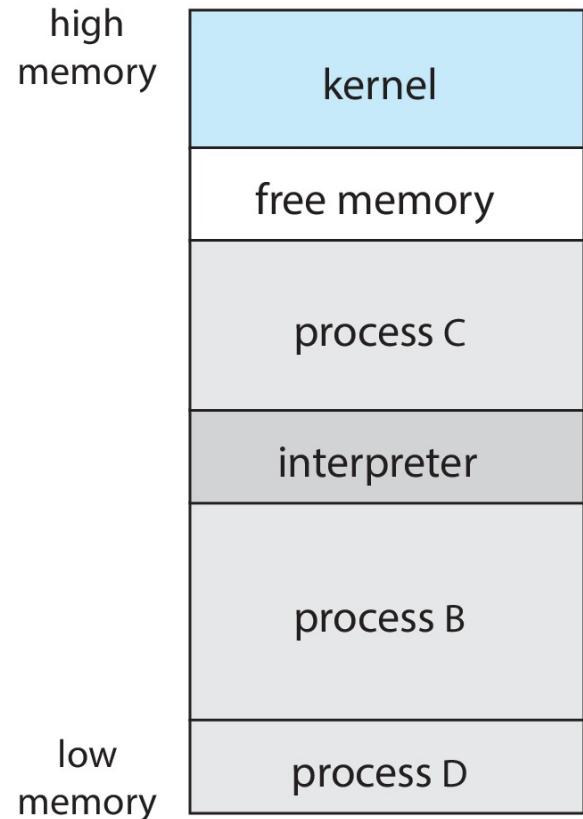
running a program





Example: FreeBSD

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





System Services

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





System Services (Cont.)

- Provide a convenient environment for program development and execution
 - Some of them are simply user interfaces to system calls; others are considerably more complex
- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
 - Some ask the system for info - date, time, amount of available memory, disk space, number of users
 - Others provide detailed performance, logging, and debugging information
 - Typically, these programs format and print the output to the terminal or other output devices
 - Some systems implement a **registry** - used to store and retrieve configuration information





System Services (Cont.)

- **File modification**
 - Text editors to create and modify files
 - Special commands to search contents of files or perform transformations of the text
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
 - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another





System Services (Cont.)

■ Background Services

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

■ Application programs

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





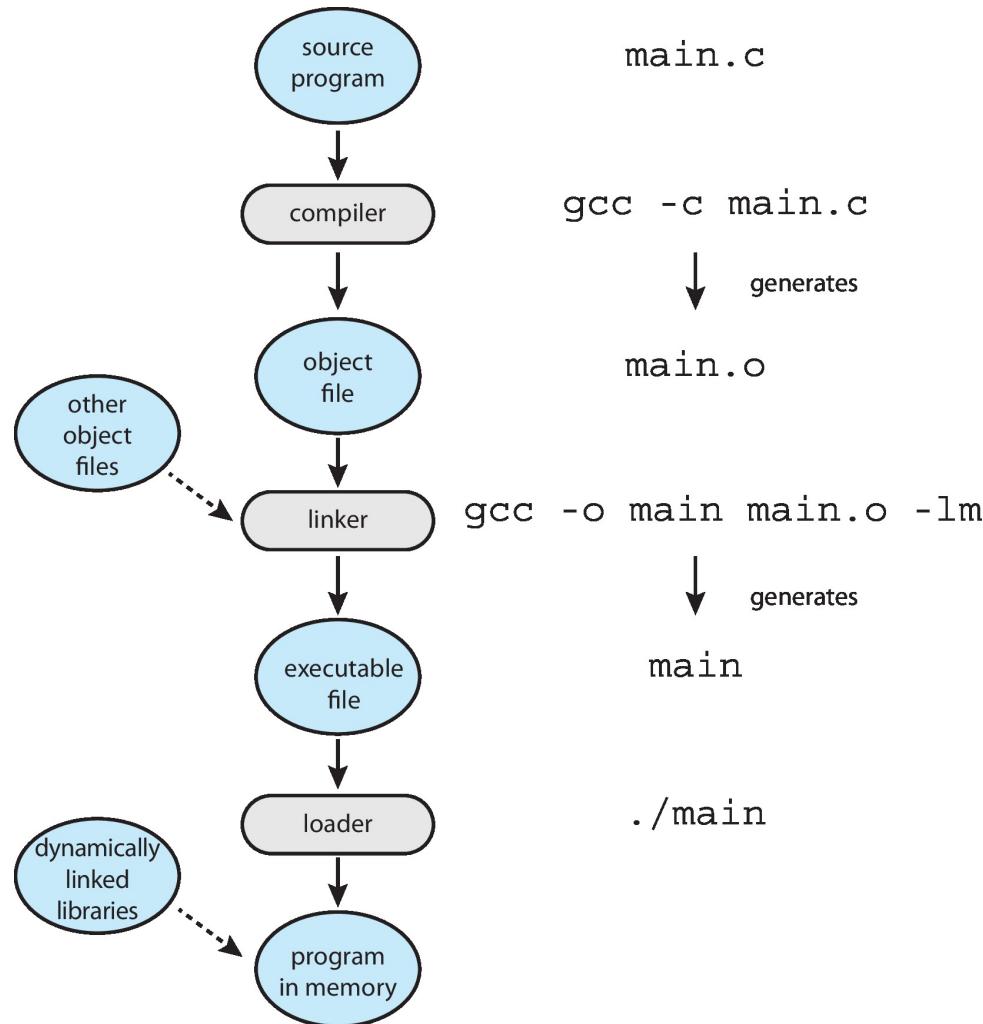
Linkers and Loaders

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





The Role of the Linker and Loader





Why Applications are Operating System Specific

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





Design and Implementation

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





Policy and Mechanism

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





Implementation

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





Operating System Structure

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





Monolithic Structure – Original UNIX

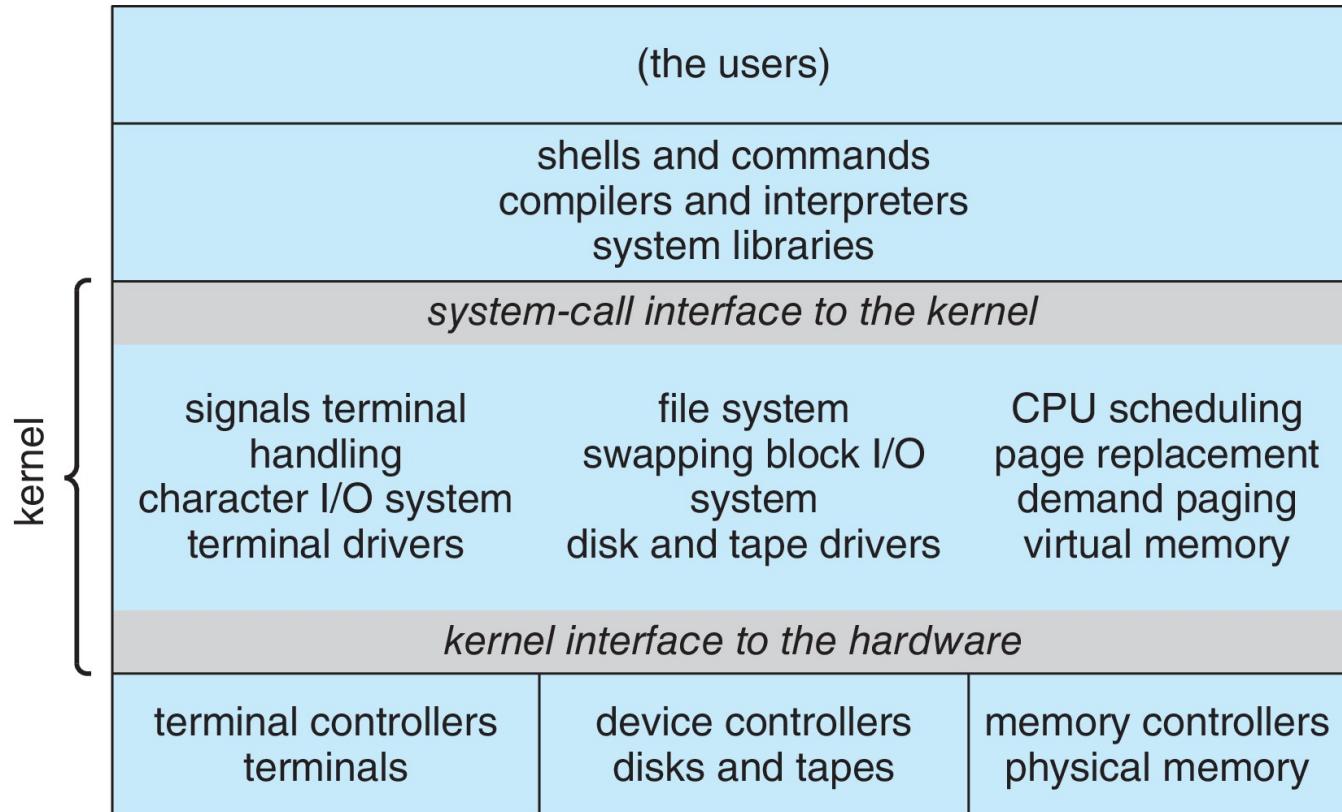
- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring.
- The UNIX OS consists of two separable parts
 - Systems programs
 - The kernel
 - ▶ Consists of everything below the system-call interface and above the physical hardware
 - ▶ Provides the file system,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





Traditional UNIX System Structure

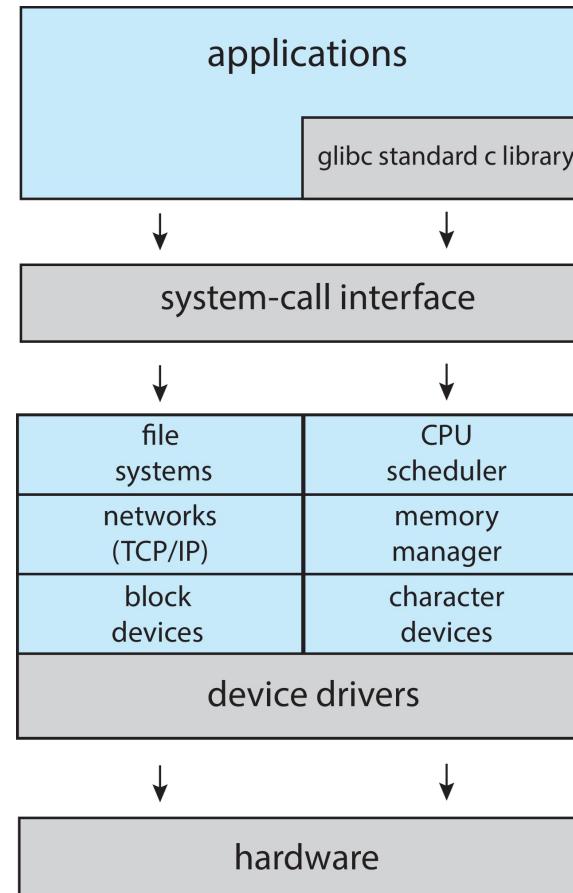
Beyond simple but not fully layered





Linux System Structure

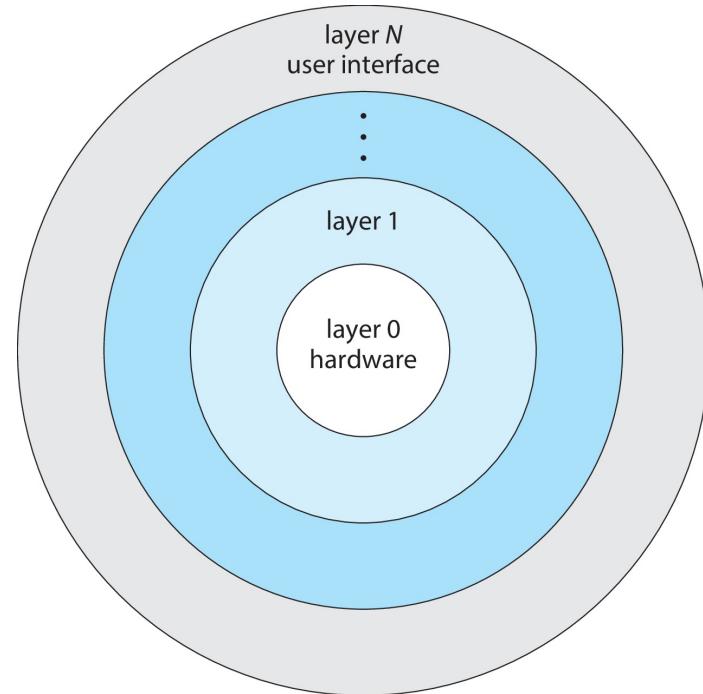
Monolithic plus modular design





Layered Approach

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





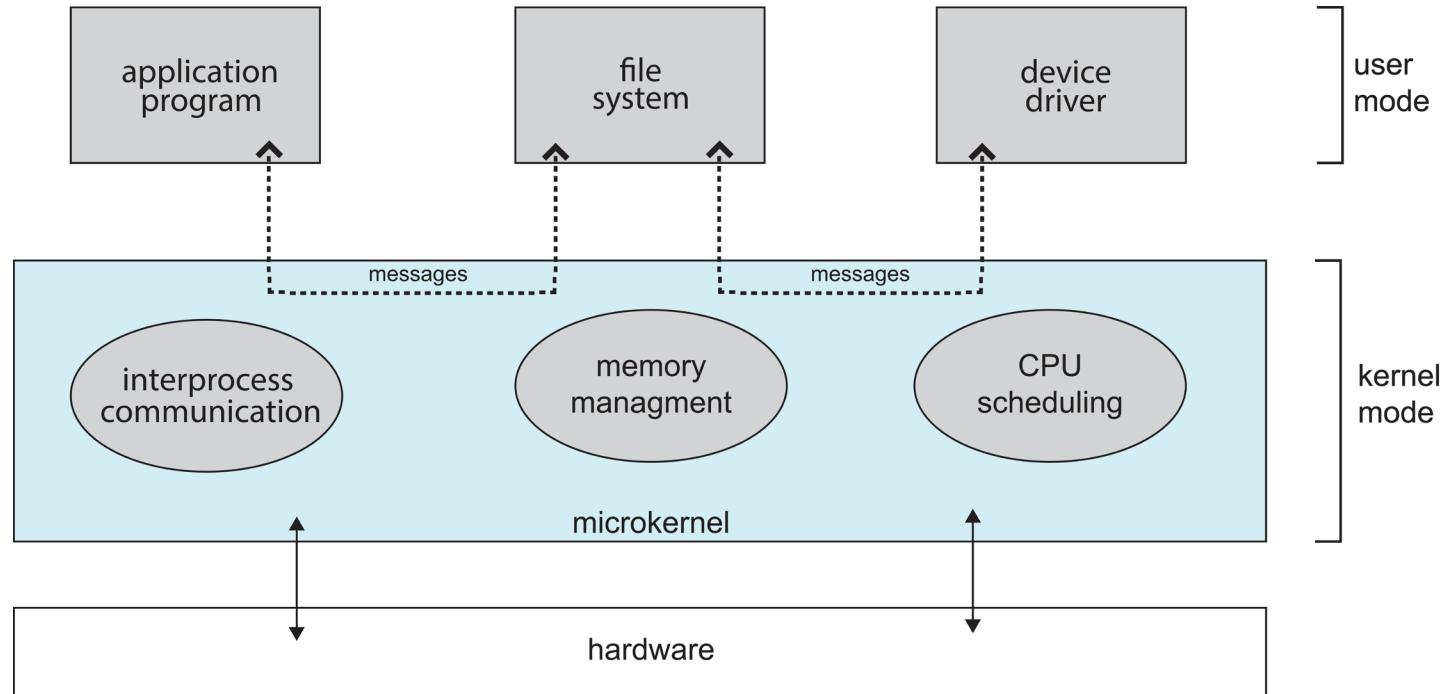
Microkernels

- Moves as much from the kernel into user space
- **Mach** is an example of **microkernel**
 - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between user modules using **message passing**
- Benefits:
 - Easier to extend a microkernel
 - Easier to port the 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.





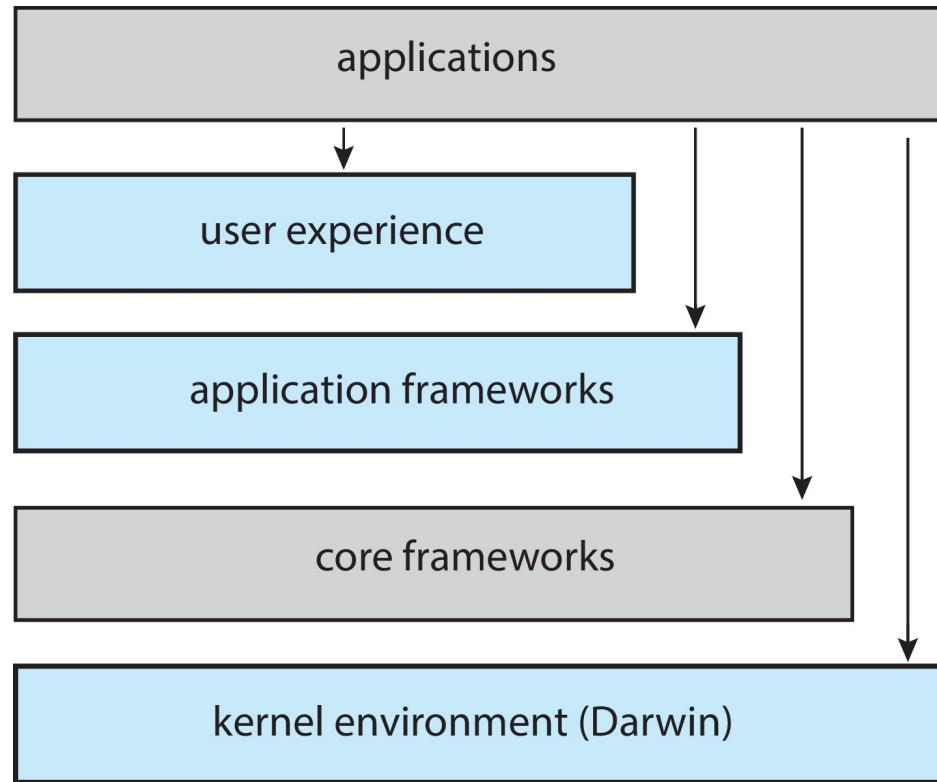
Hybrid Systems

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



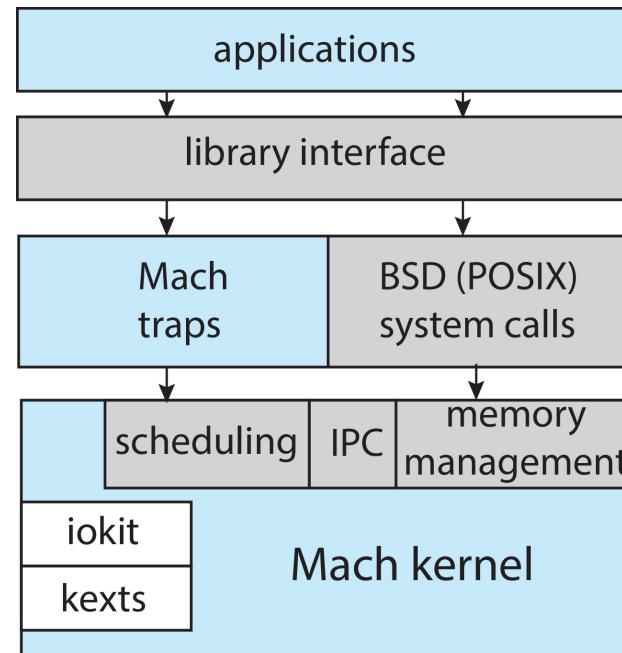


macOS and iOS Structure





Darwin





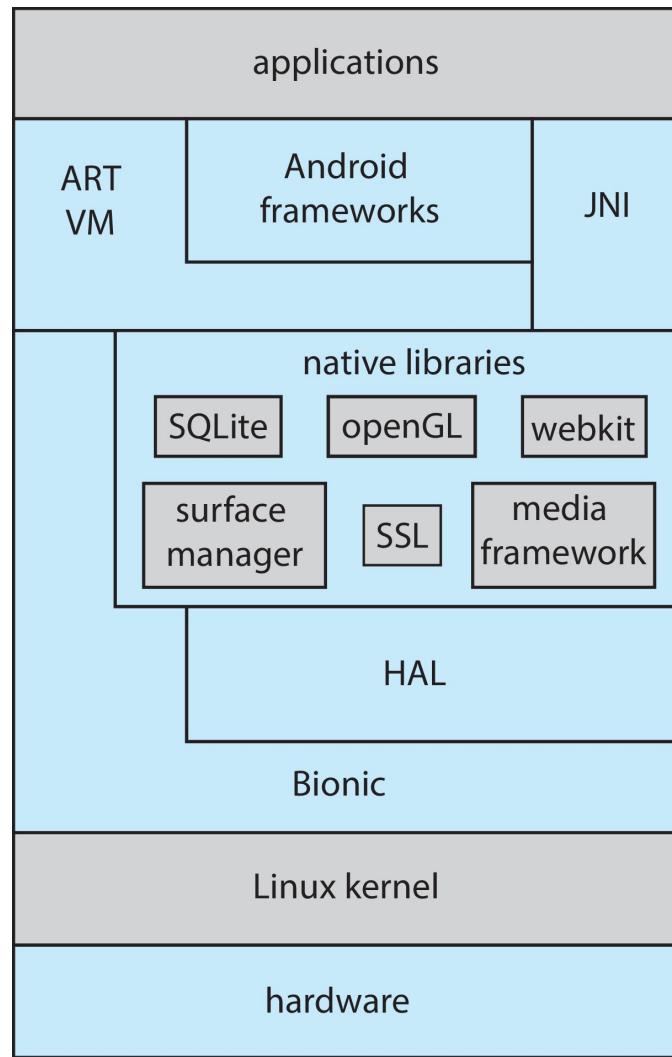
Android

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





Android Architecture





Building and Booting an Operating System

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





Building and Booting Linux

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





System Boot

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





Operating-System Debugging

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

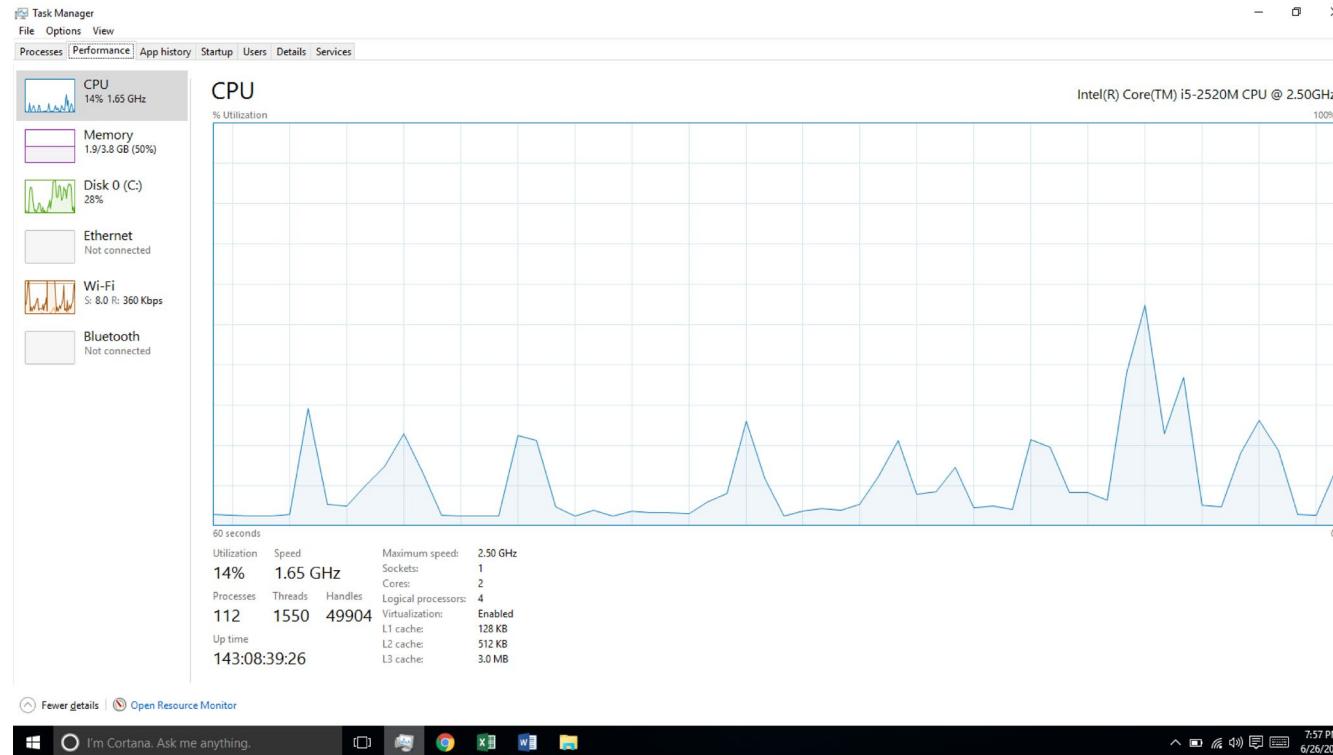
Kernighan's Law: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."





Performance Tuning

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





Tracing

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





BCC

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

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

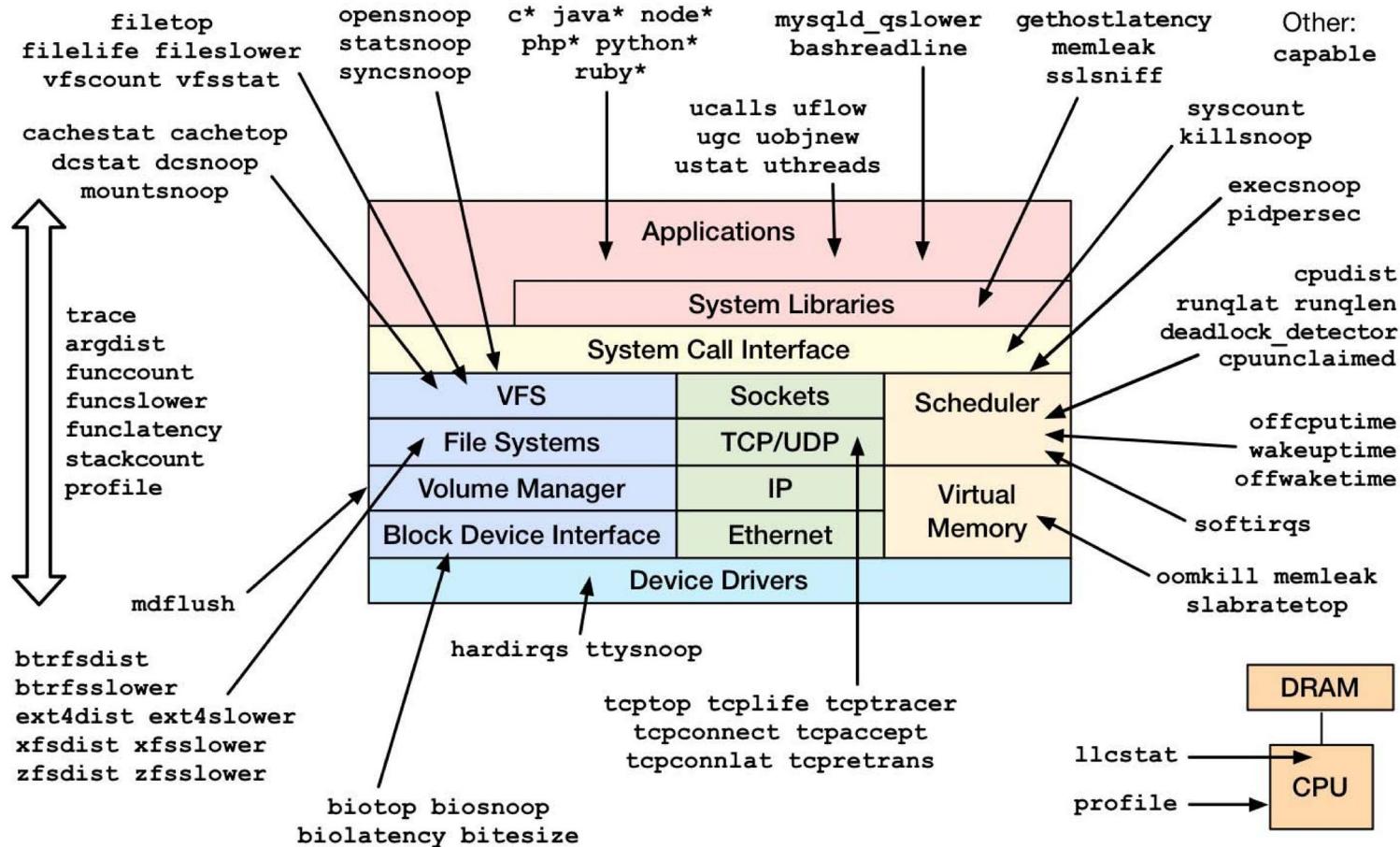
- Many other tools (next slide)





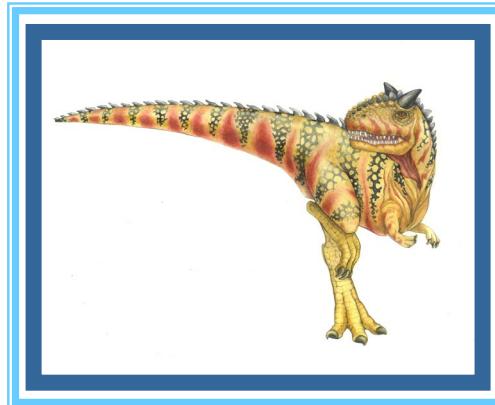
Linux bcc/BPF Tracing Tools

Linux bcc/BPF Tracing Tools

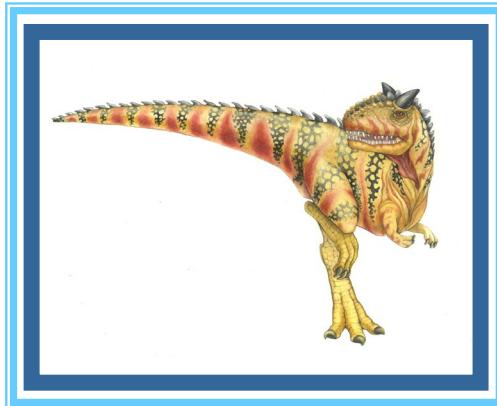


<https://github.com/iovisor/bcc#tools 2017>

End of Chapter 2



Chapter 3: Processes





Outline

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





Objectives

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





Process Concept

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process
- Multiple parts
 - The program code, also called **text section**
 - Current activity including **program counter**, processor registers
 - **Stack** containing temporary data
 - ▶ Function parameters, return addresses, local variables
 - **Data section** containing global variables
 - **Heap** containing memory dynamically allocated during run time





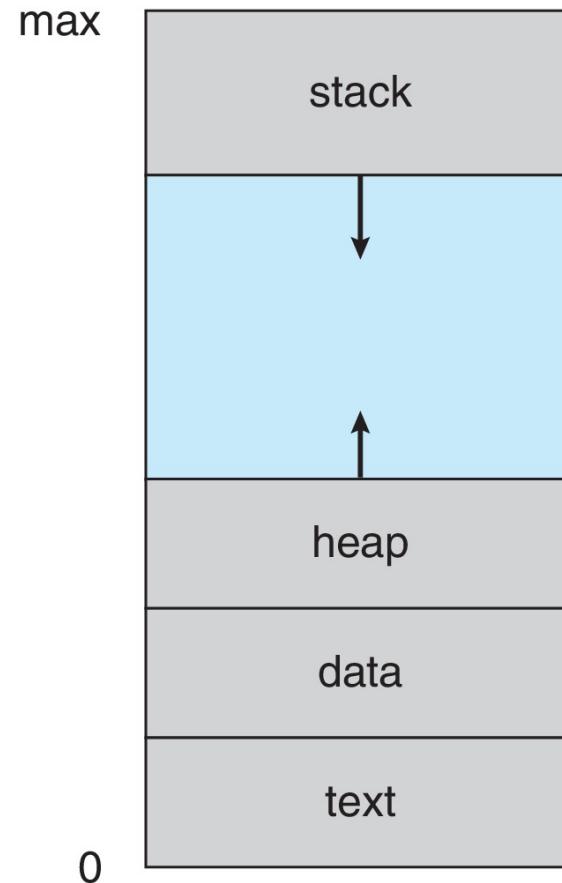
Process Concept (Cont.)

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



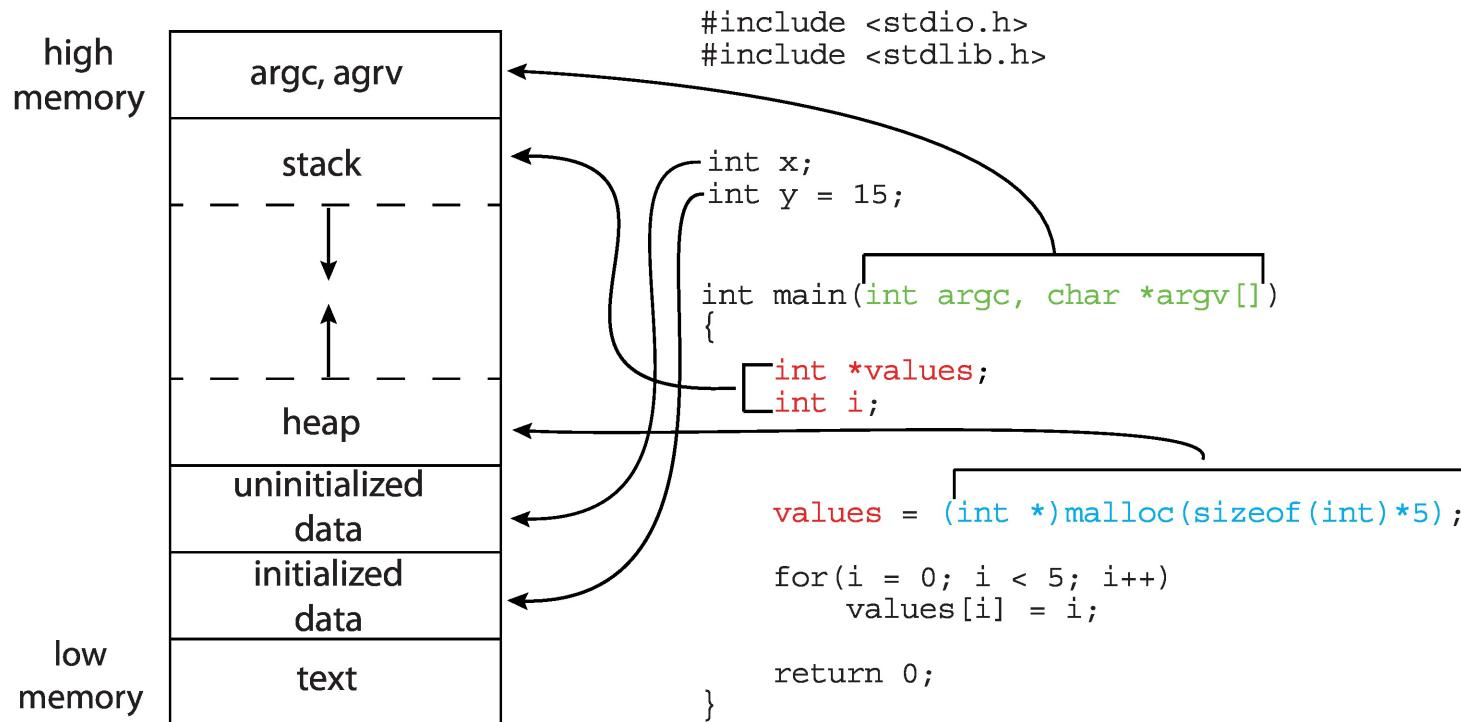


Process in Memory





Memory Layout of a C Program





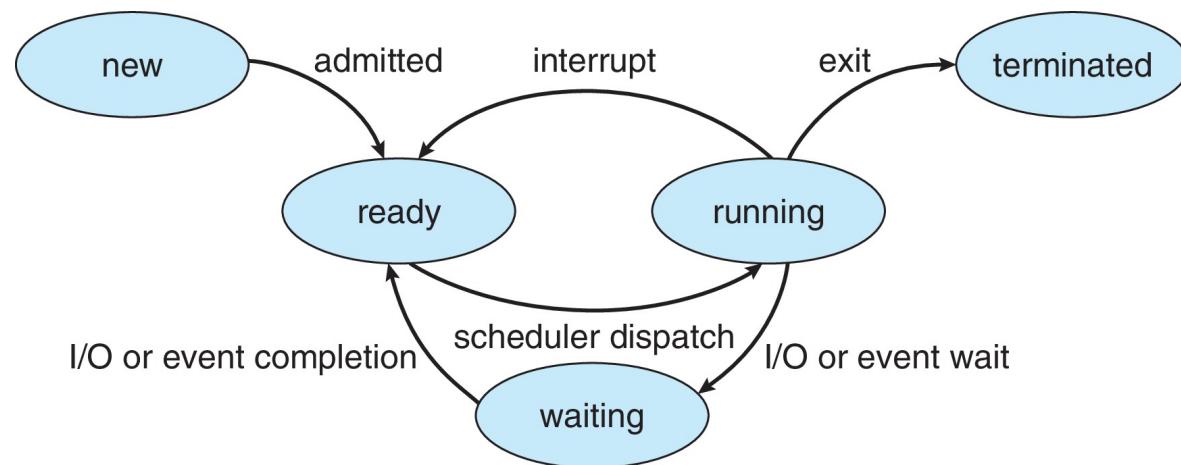
Process State

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





Diagram of Process State





Process Control Block (PCB)

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

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

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





Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
 - Multiple locations can execute at once
 - ▶ Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- Explore in detail in Chapter 4

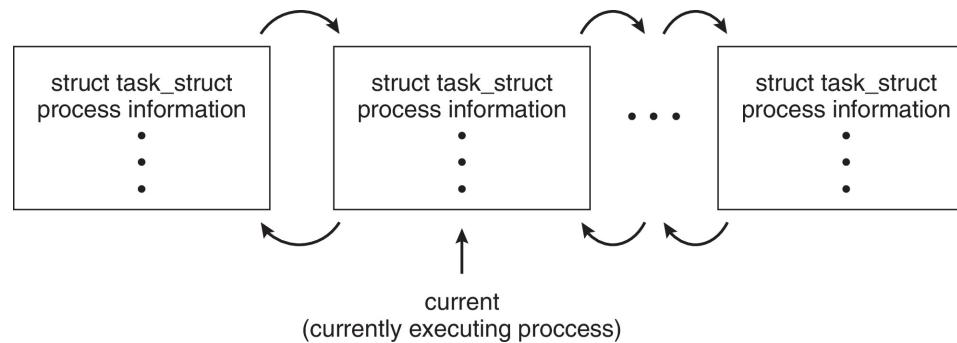




Process Representation in Linux

Represented by the C structure `task_struct`

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





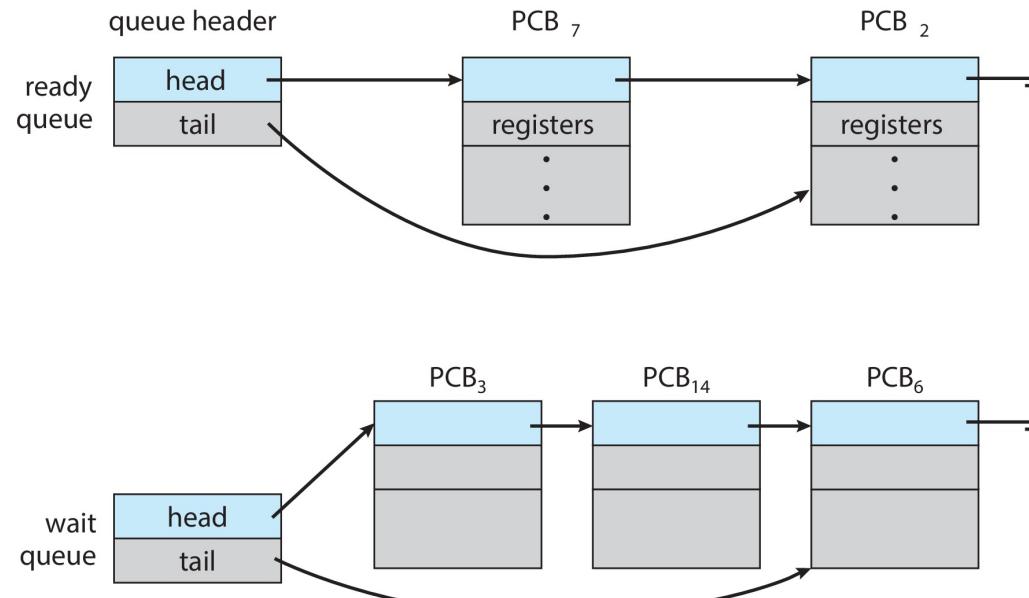
Process Scheduling

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



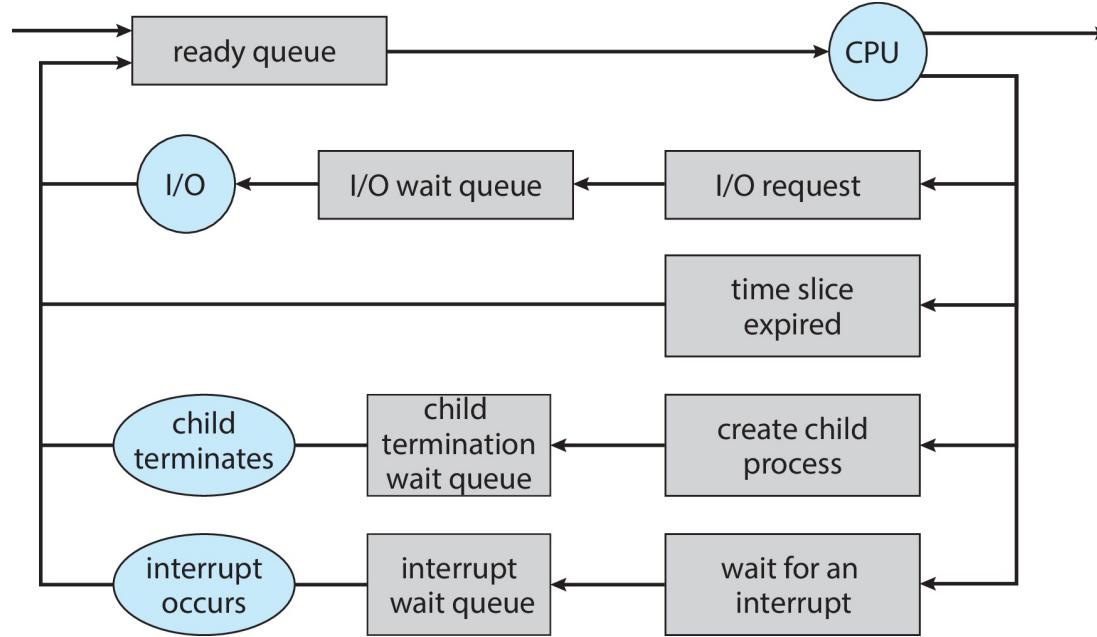


Ready and Wait Queues





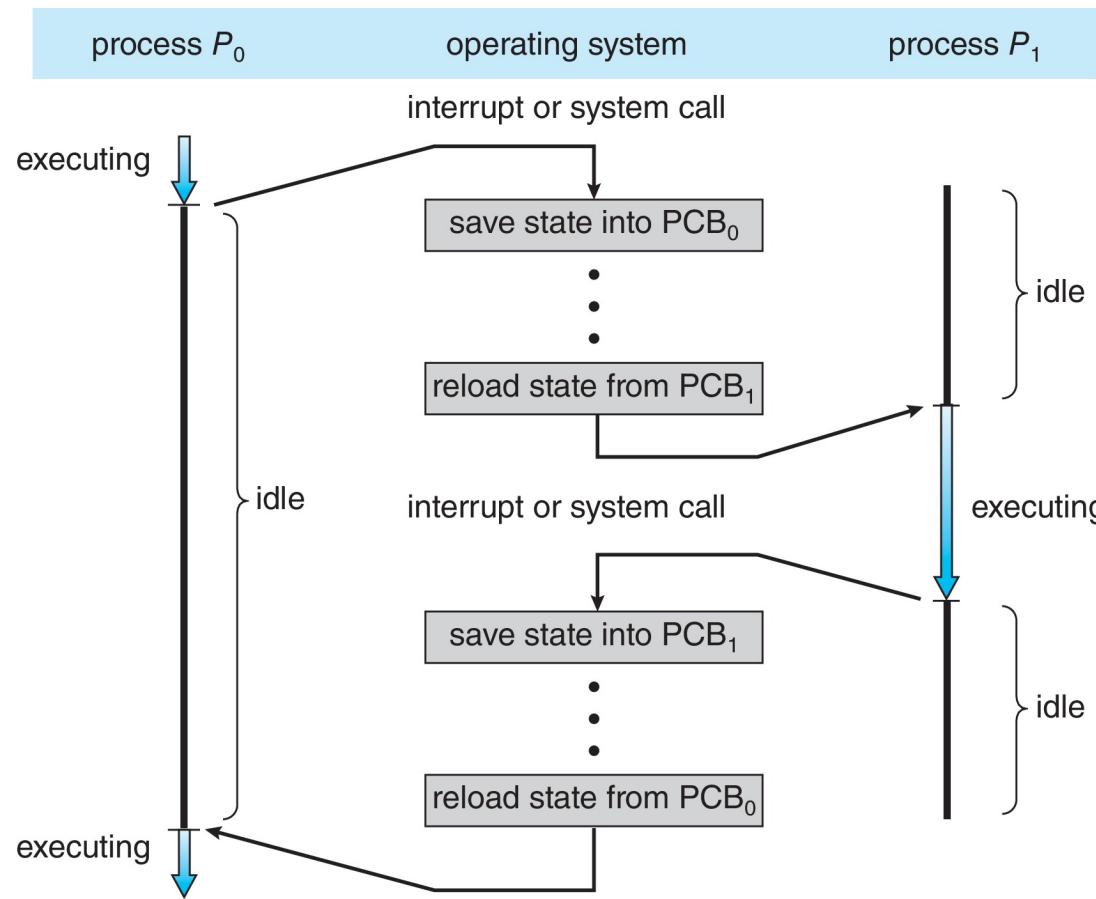
Representation of Process Scheduling





CPU Switch From Process to Process

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





Context Switch

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





Multitasking in Mobile Systems

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





Operations on Processes

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





Process Creation

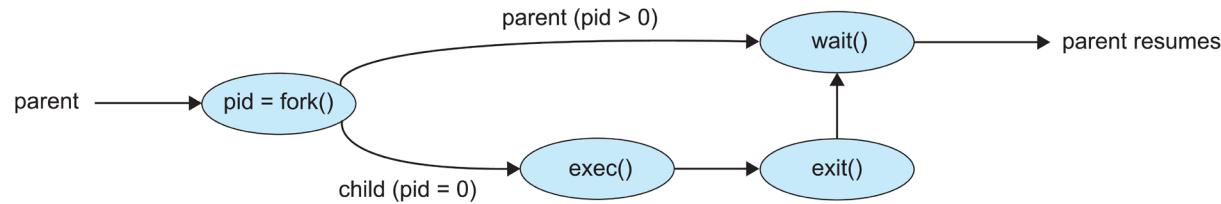
- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing options
 - Parent and children share all resources
 - Children share subset of parent's resources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





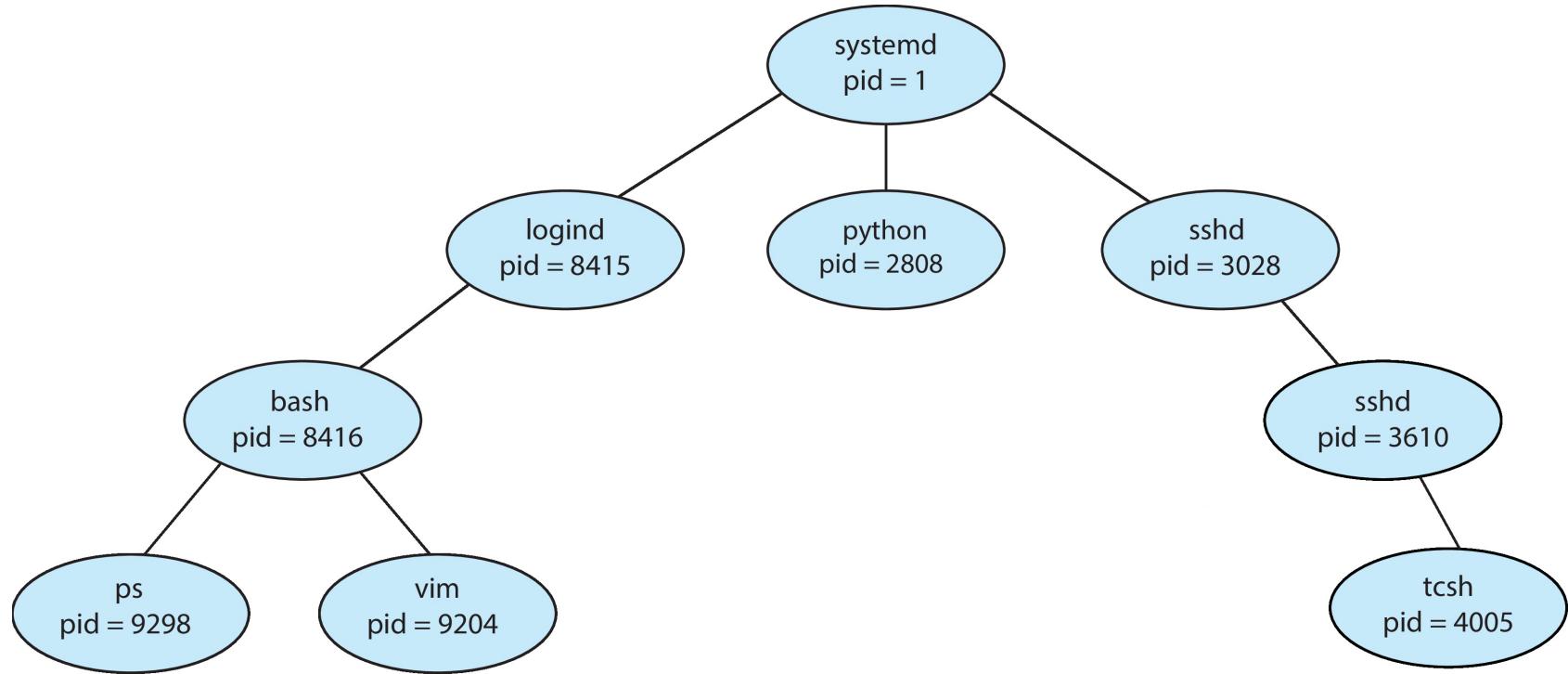
Process Creation (Cont.)

- Address space
 - Child duplicate of parent
 - Child has a program loaded into it
- UNIX examples
 - **fork()** system call creates new process
 - **exec()** system call used after a **fork()** to replace the process' memory space with a new program
 - Parent process calls **wait()** waiting for the child to terminate





A Tree of Processes in Linux





C Program Forking Separate Process

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

int main()
{
    pid_t pid;

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

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

return 0;
}
```





Creating a Separate Process via Windows API

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

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

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

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

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





Process Termination

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





Process Termination

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

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





Android Process Importance Hierarchy

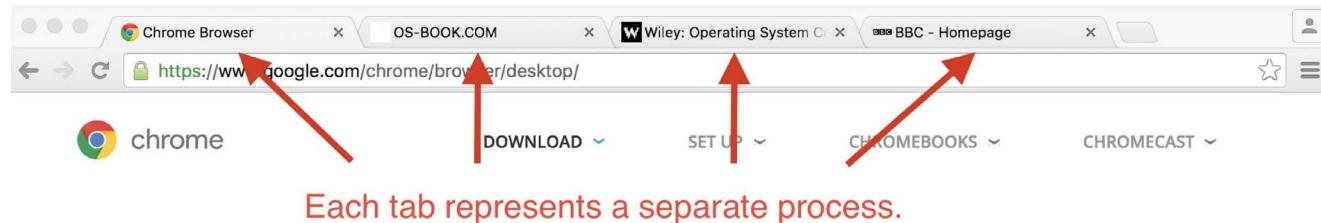
- Mobile operating systems often have to terminate processes to reclaim system resources such as memory. From **most** to **least** important:
 - Foreground process
 - Visible process
 - Service process
 - Background process
 - Empty process
- Android will begin terminating processes that are least important.





Multiprocess Architecture – Chrome Browser

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





Interprocess Communication

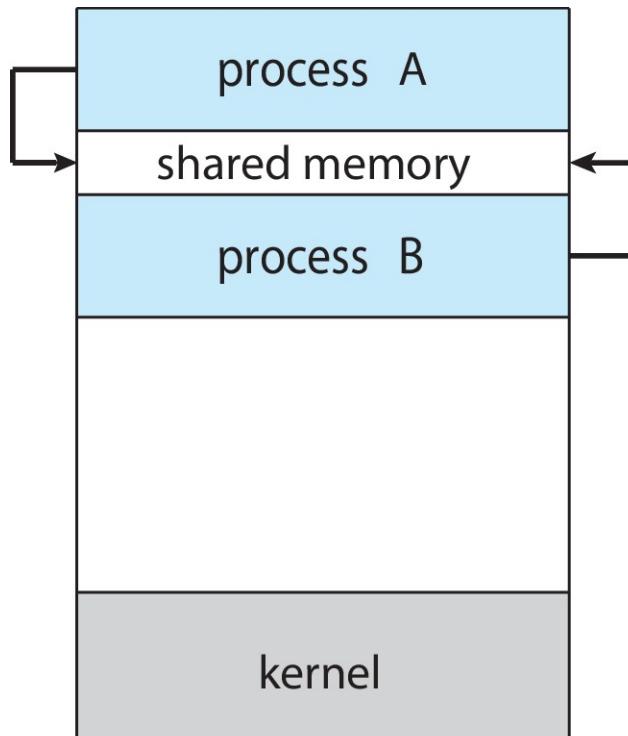
- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
 - Information sharing
 - Computation speedup
 - Modularity
 - Convenience
- Cooperating processes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





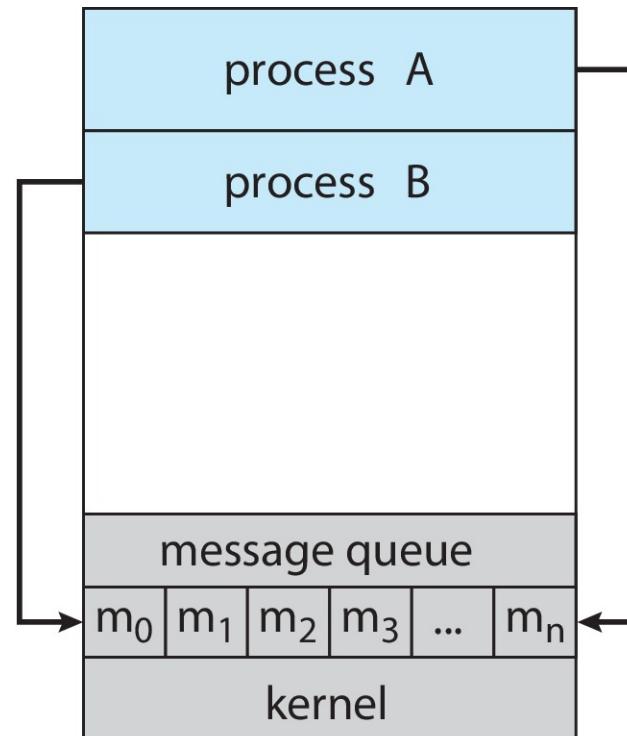
Communications Models

(a) Shared memory.



(a)

(b) Message passing.



(b)





Producer-Consumer Problem

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





IPC – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapters 6 & 7.





Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10

typedef struct {

    . . .

} item;

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

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





Producer Process – Shared Memory

```
item next_produced;

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





Consumer Process – Shared Memory

```
item next_consumed;

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

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





What about Filling all the Buffers?

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.
- We can do so by having an integer **counter** that keeps track of the number of full buffers.
- Initially, **counter** is set to 0.
- The integer **counter** is incremented by the producer after it produces a new buffer.
- The integer **counter** is and is decremented by the consumer after it consumes a buffer.





Producer

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





Consumer

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





Race Condition

- `counter++` could be implemented as

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

- `counter--` could be implemented as

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

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

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





Race Condition (Cont.)

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





IPC – Message Passing

- Processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
 - `send(message)`
 - `receive(message)`
- The *message size* is either fixed or variable





Message Passing (Cont.)

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





Implementation of Communication Link

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





Direct Communication

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





Indirect Communication

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





Indirect Communication (Cont.)

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





Indirect Communication (Cont.)

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

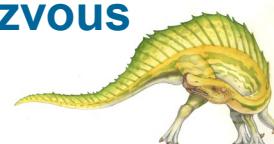




Synchronization

Message passing may be either blocking or non-blocking

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





Producer-Consumer: Message Passing

- Producer

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

    send(next_produced);
}
```

- Consumer

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

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





Buffering

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





Examples of IPC Systems - POSIX

- POSIX Shared Memory

- Process first creates shared memory segment
`shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`
 - Also used to open an existing segment
 - Set the size of the object
`ftruncate(shm_fd, 4096);`
 - Use `mmap()` to memory-map a file pointer to the shared memory object
 - Reading and writing to shared memory is done by using the pointer returned by `mmap()`.





IPC POSIX Producer

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

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

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

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

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

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

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

    return 0;
}
```





IPC POSIX Consumer

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

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

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

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

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

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

    return 0;
}
```





Examples of IPC Systems - Mach

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





Mach Messages

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

struct message {
    mach_msg_header_t header;
    int data;
};

mach_port_t client;
mach_port_t server;
```





Mach Message Passing - Client

```
/* Client Code */

struct message message;

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

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





Mach Message Passing - Server

```
/* Server Code */

struct message message;

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





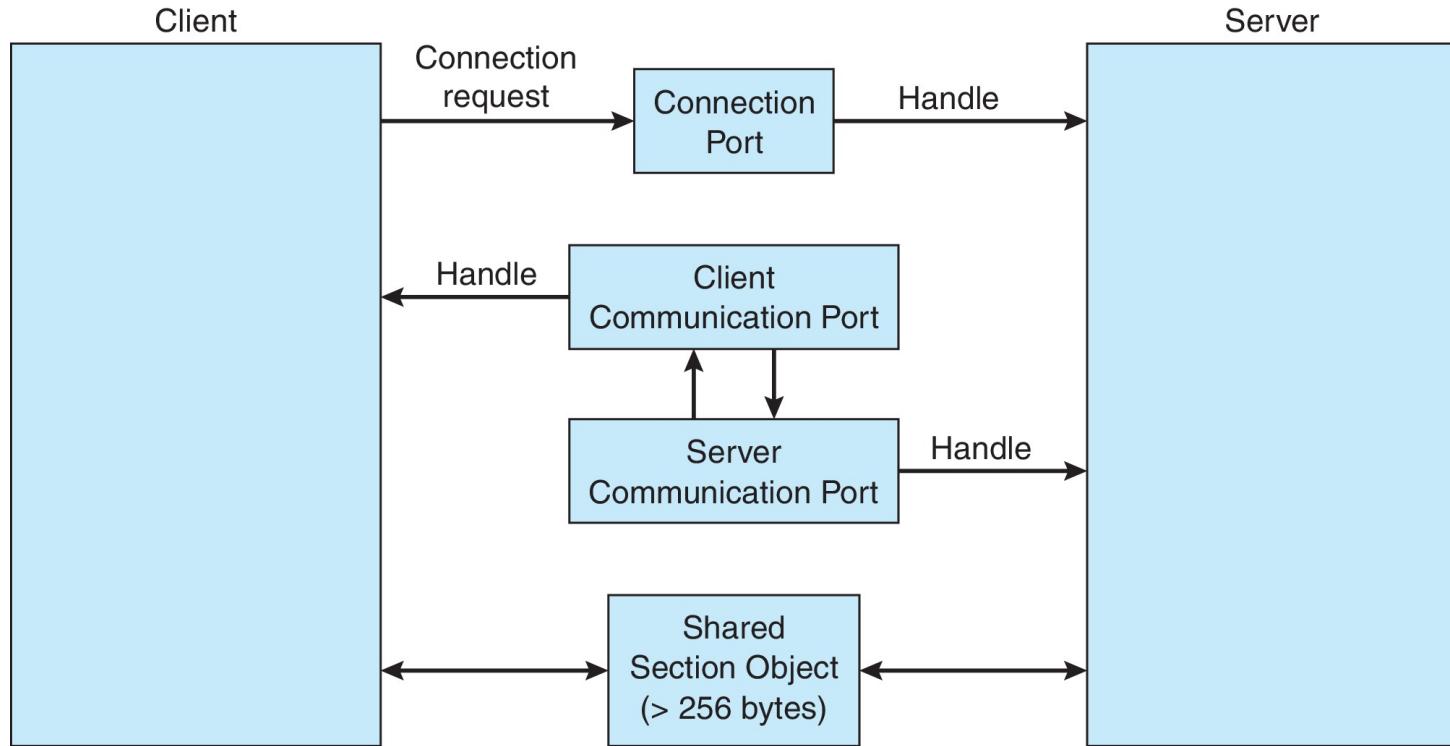
Examples of IPC Systems – Windows

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





Local Procedure Calls in Windows





Pipes

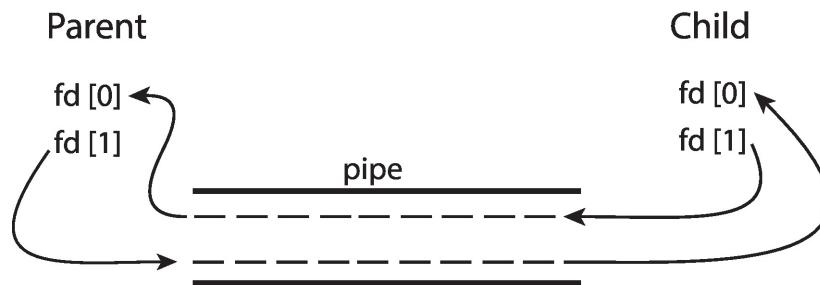
- Acts as a conduit allowing two processes to communicate
- Issues:
 - Is communication unidirectional or bidirectional?
 - In the case of two-way communication, is it half or full-duplex?
 - Must there exist a relationship (i.e., **parent-child**) between the communicating processes?
 - Can the pipes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





Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes



- Windows calls these **anonymous pipes**





Named Pipes

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





Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls





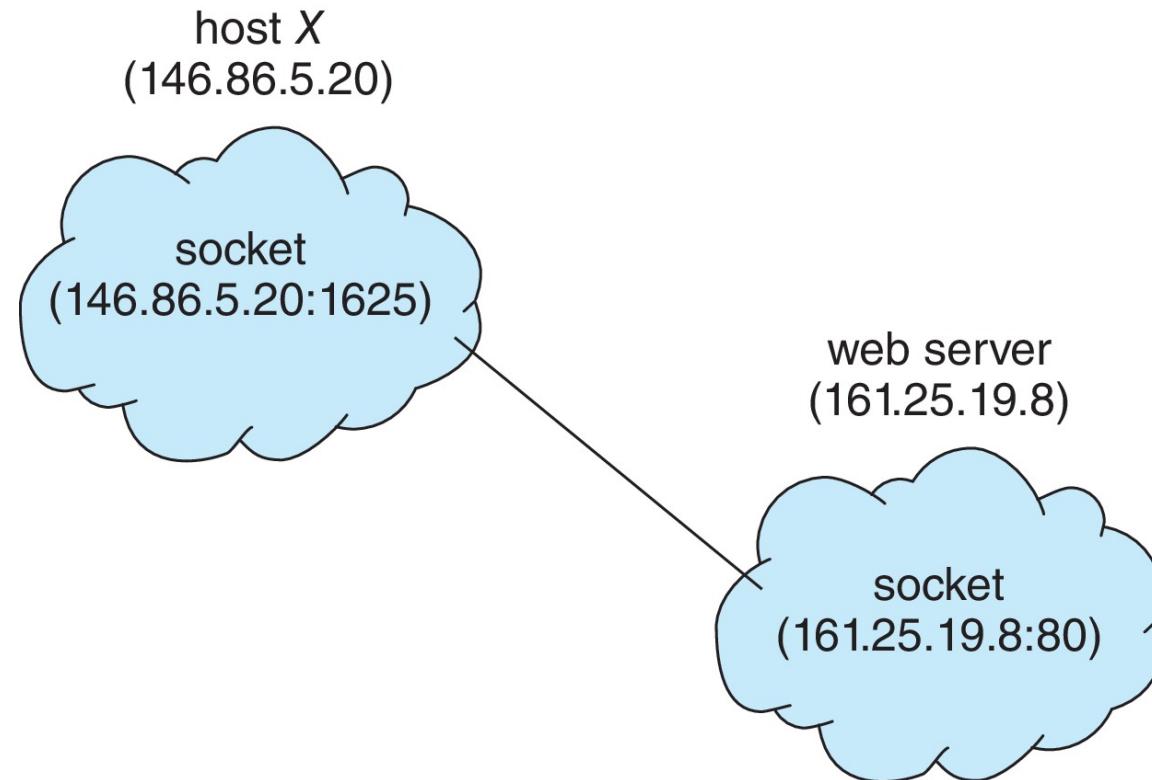
Sockets

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





Socket Communication





Sockets in Java

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

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

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

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

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

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

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





Sockets in Java

The equivalent Date client

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

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

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

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

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





Remote Procedure Calls

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





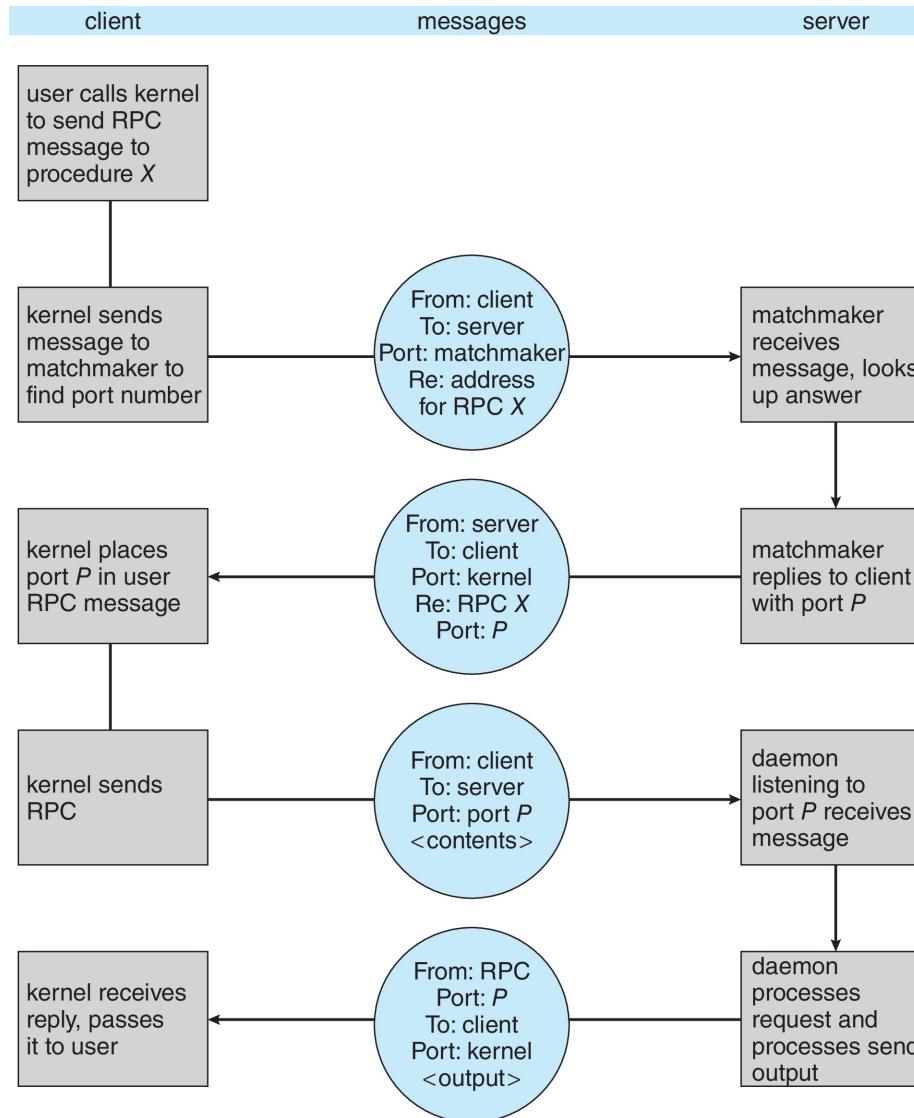
Remote Procedure Calls (Cont.)

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

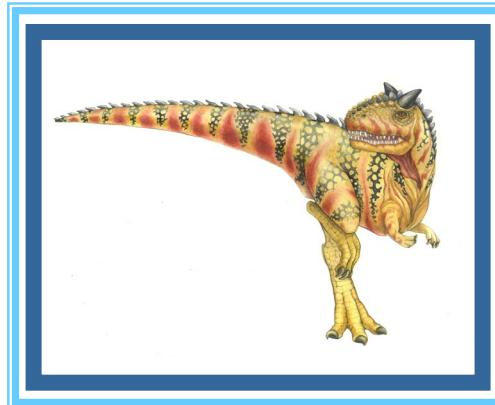




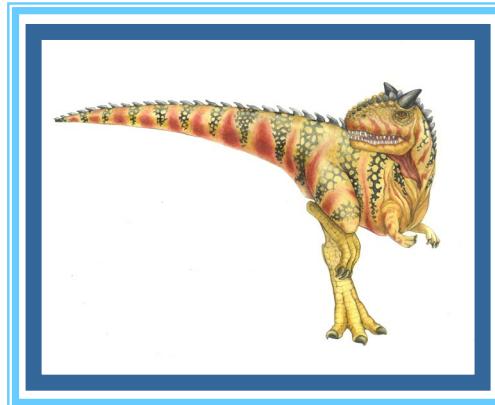
Execution of RPC



End of Chapter 3



Chapter 4: Threads & Concurrency





Outline

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





Objectives

- Identify the basic components of a thread, and contrast threads 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





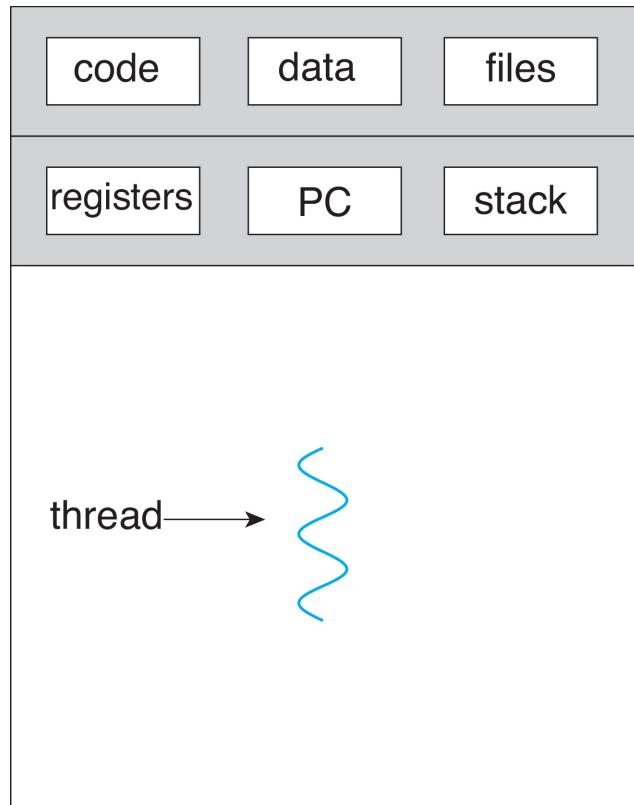
Motivation

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

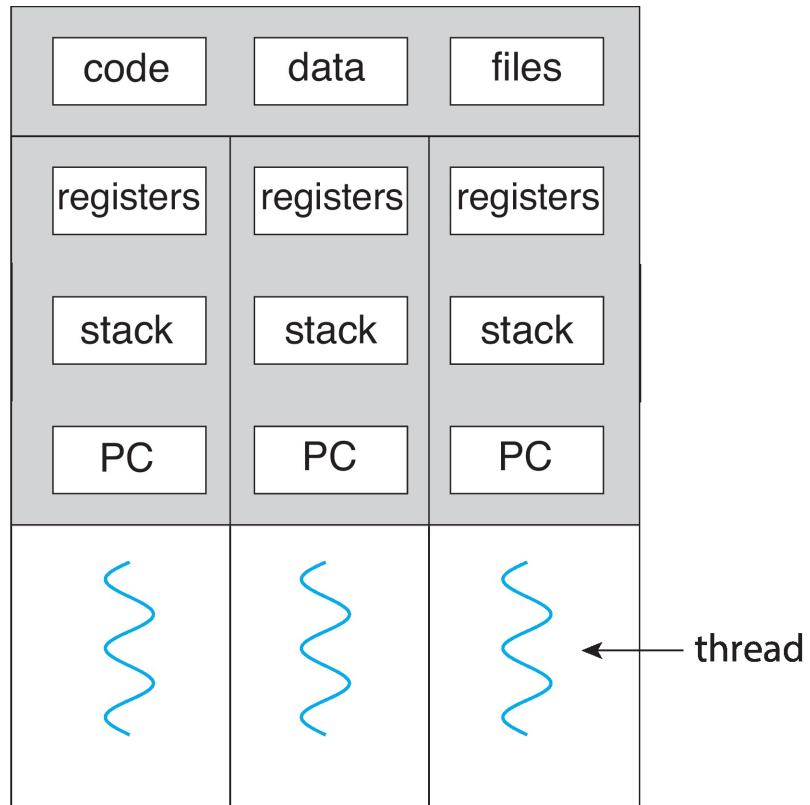




Single and Multithreaded Processes



single-threaded process

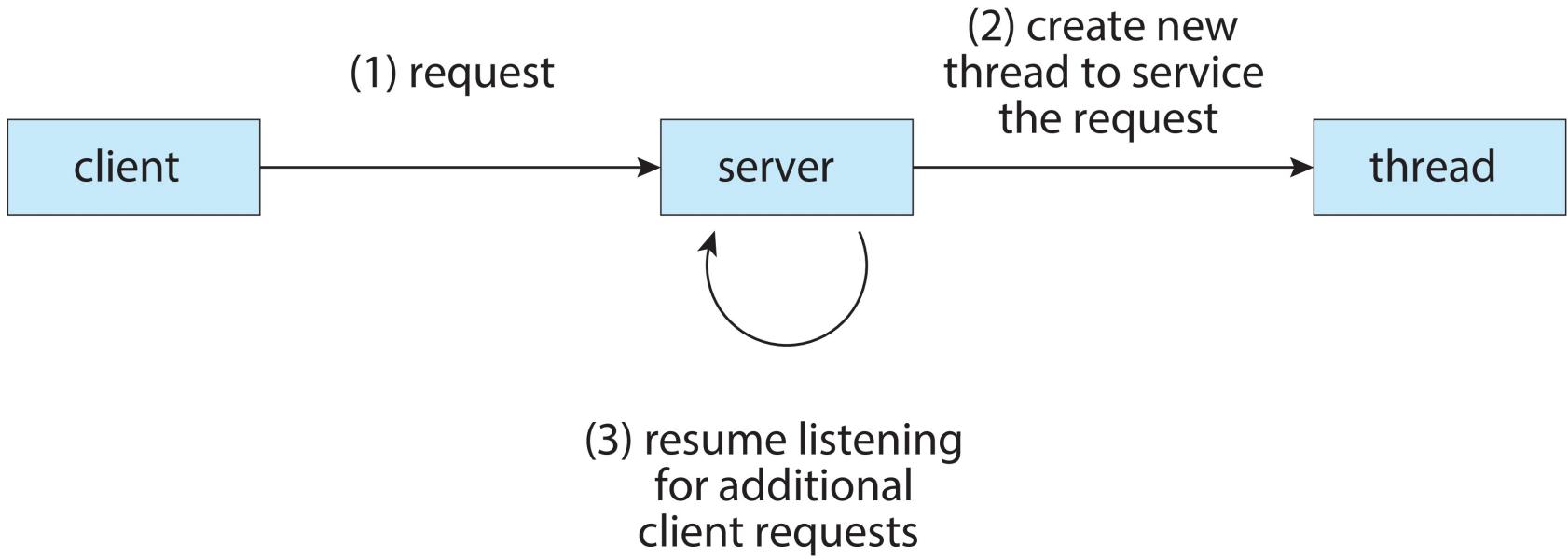


multithreaded process





Multithreaded Server Architecture





Benefits

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





Multicore Programming

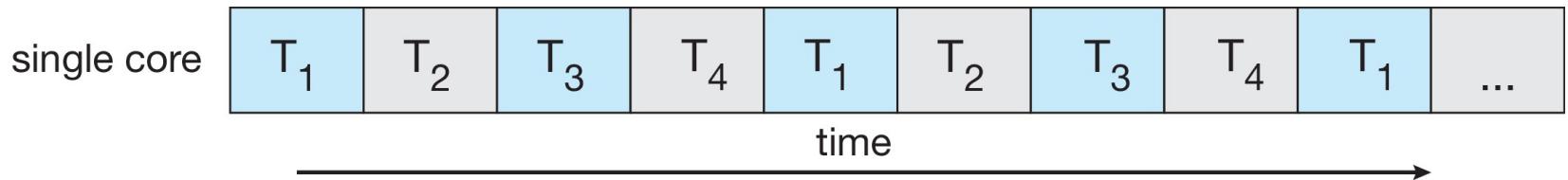
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
 - **Dividing activities**
 - **Balance**
 - **Data splitting**
 - **Data dependency**
 - **Testing and debugging**
- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency**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



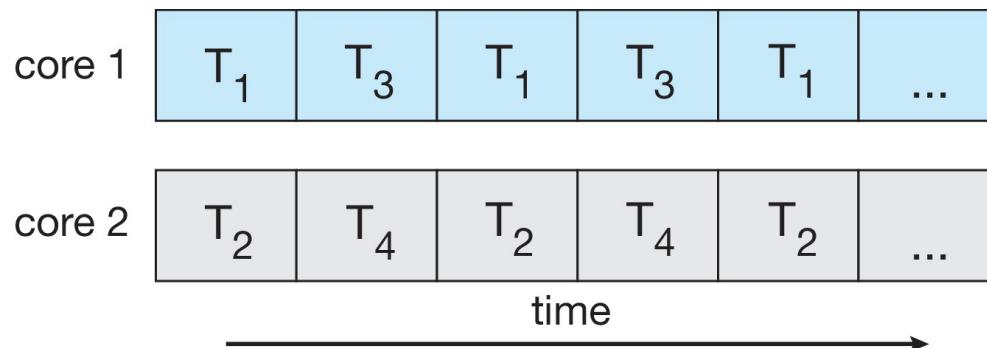


Concurrency vs. Parallelism

- Concurrent execution on single-core system:



- Parallelism on a multi-core system:





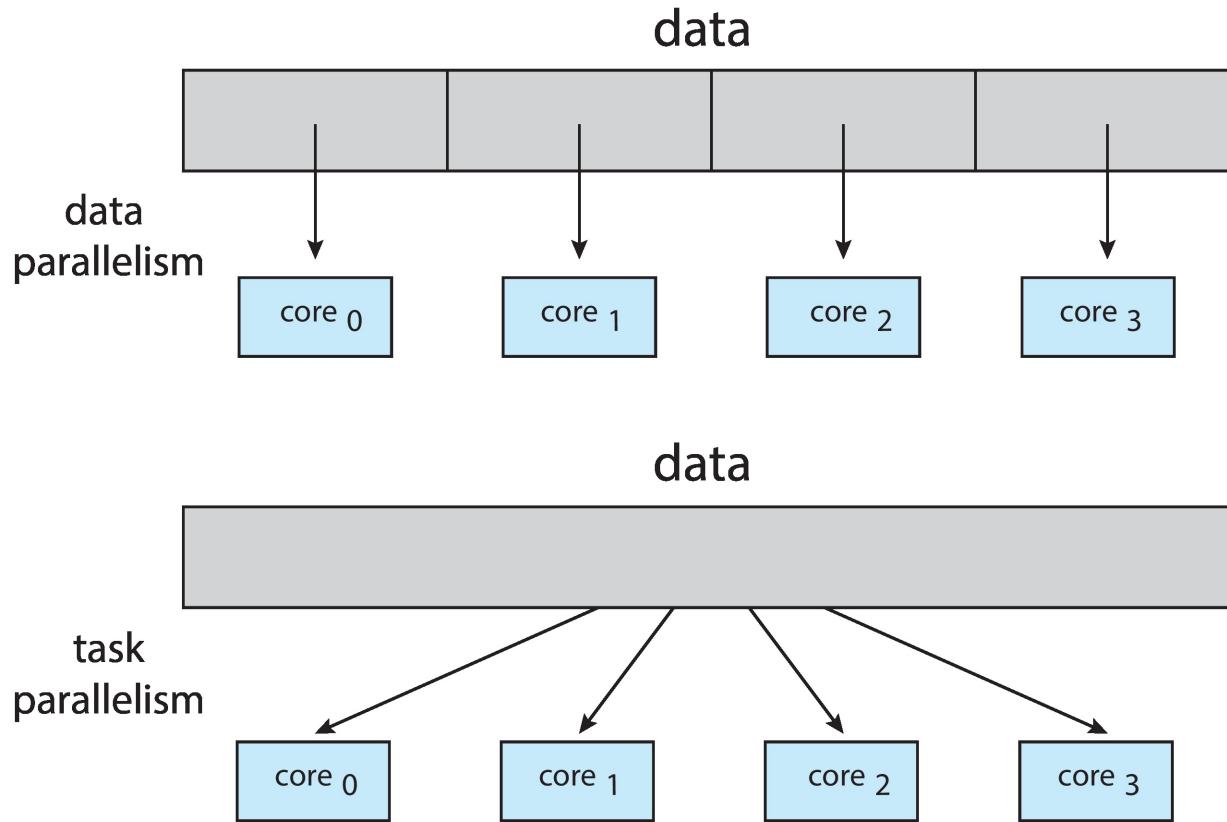
Multicore Programming

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





Data and Task Parallelism





Amdahl's Law

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

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

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

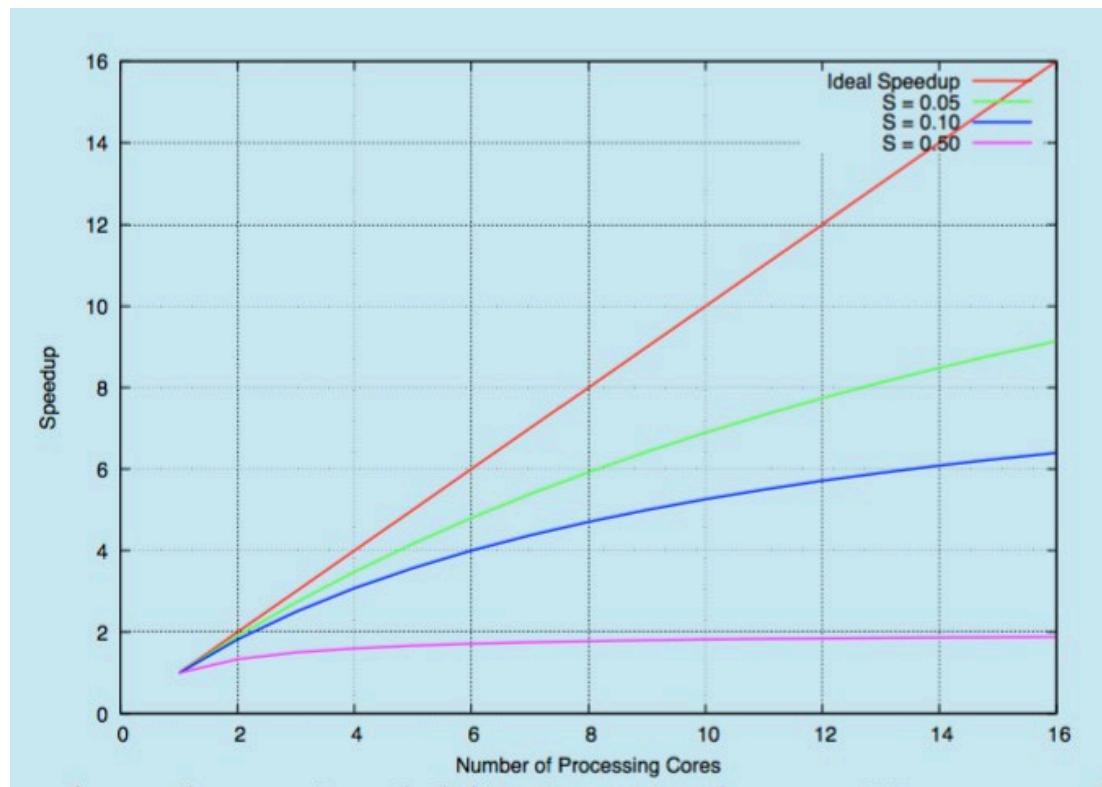
Serial portion of an application has disproportionate effect on performance gained by adding additional cores

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





Amdahl's Law





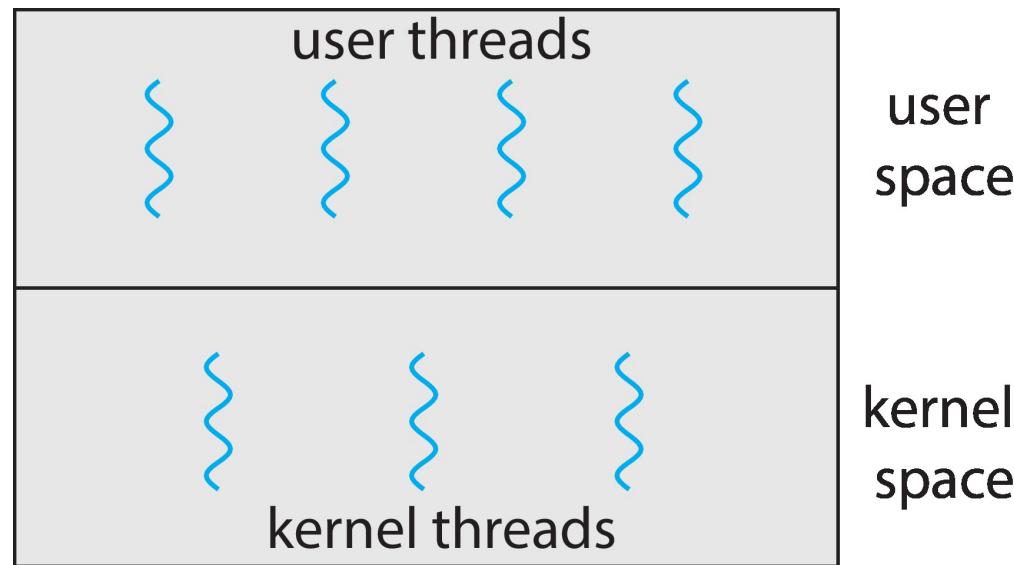
User Threads and Kernel Threads

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





User and Kernel Threads





Multithreading Models

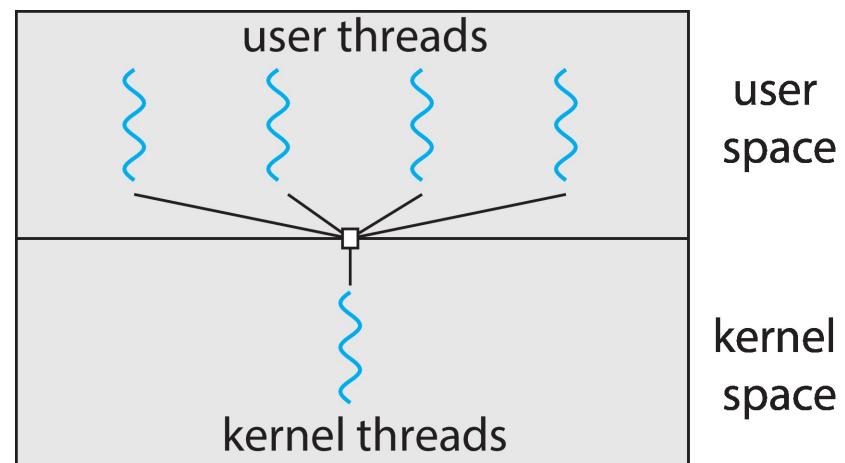
- Many-to-One
- One-to-One
- Many-to-Many





Many-to-One

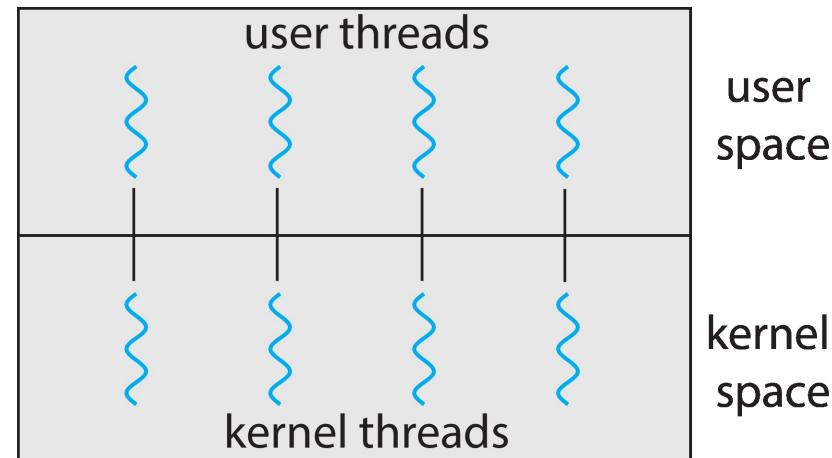
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
 - **Solaris Green Threads**
 - **GNU Portable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





One-to-One

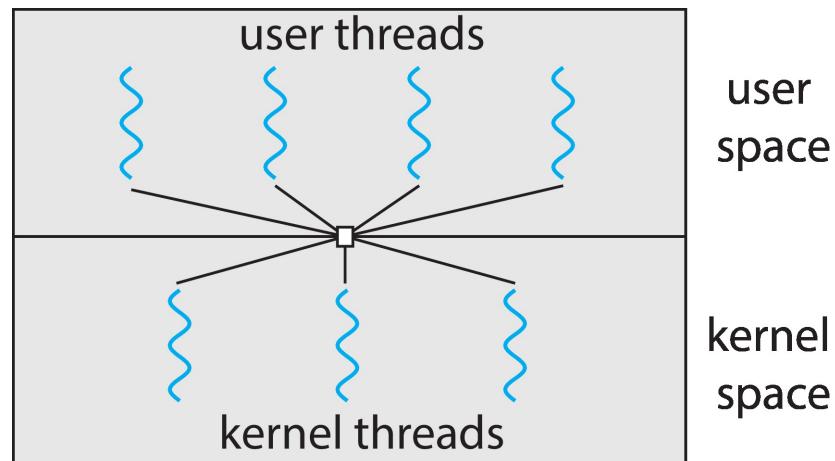
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
 - Windows
 - Linux





Many-to-Many Model

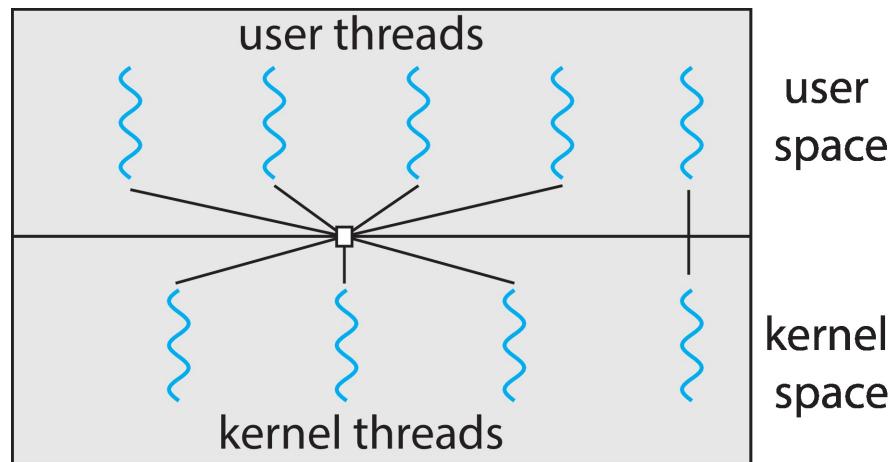
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the *ThreadFiber* package
- Otherwise not very common





Two-level Model

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





Thread Libraries

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





Pthreads

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





Pthreads Example

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

#include <stdlib.h>

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

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

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

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





Pthreads Example (Cont.)

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

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

    pthread_exit(0);
}
```





Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

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

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



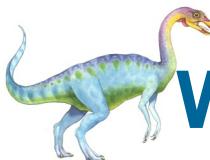


Windows Multithreaded C Program

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

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





Windows Multithreaded C Program (Cont.)

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

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

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

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

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





Java Threads

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

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

- Standard practice is to implement Runnable interface





Java Threads

Implementing Runnable interface:

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

Creating a thread:

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

Waiting on a thread:

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





Java Executor Framework

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

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

- The Executor is used as follows:

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





Java Executor Framework

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





Java Executor Framework (Cont.)

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

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

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





Implicit Threading

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





Thread Pools

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

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





Java Thread Pools

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





Java Thread Pools (Cont.)

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

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

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

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

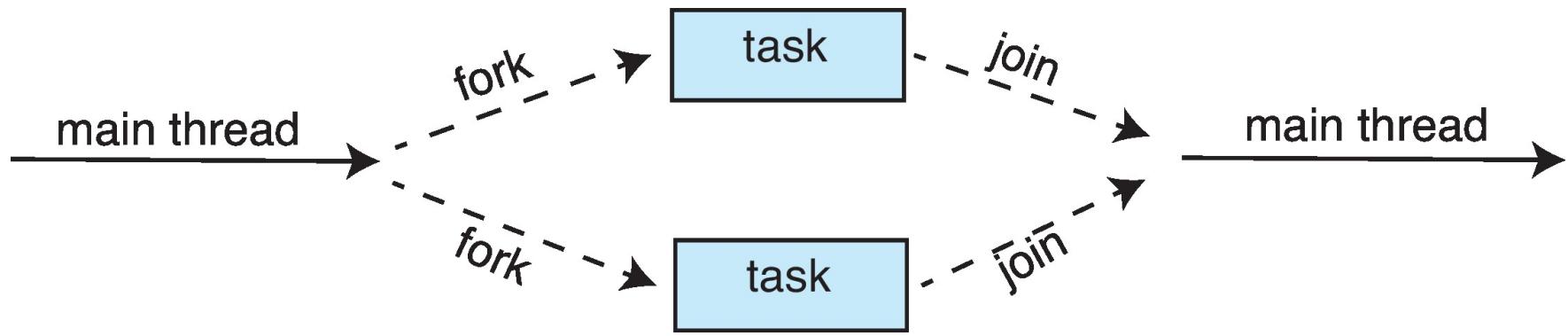
        /* Shut down the pool once all threads have completed */
        pool.shutdown();
    }
}
```





Fork-Join Parallelism

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





Fork-Join Parallelism

- General algorithm for fork-join strategy:

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

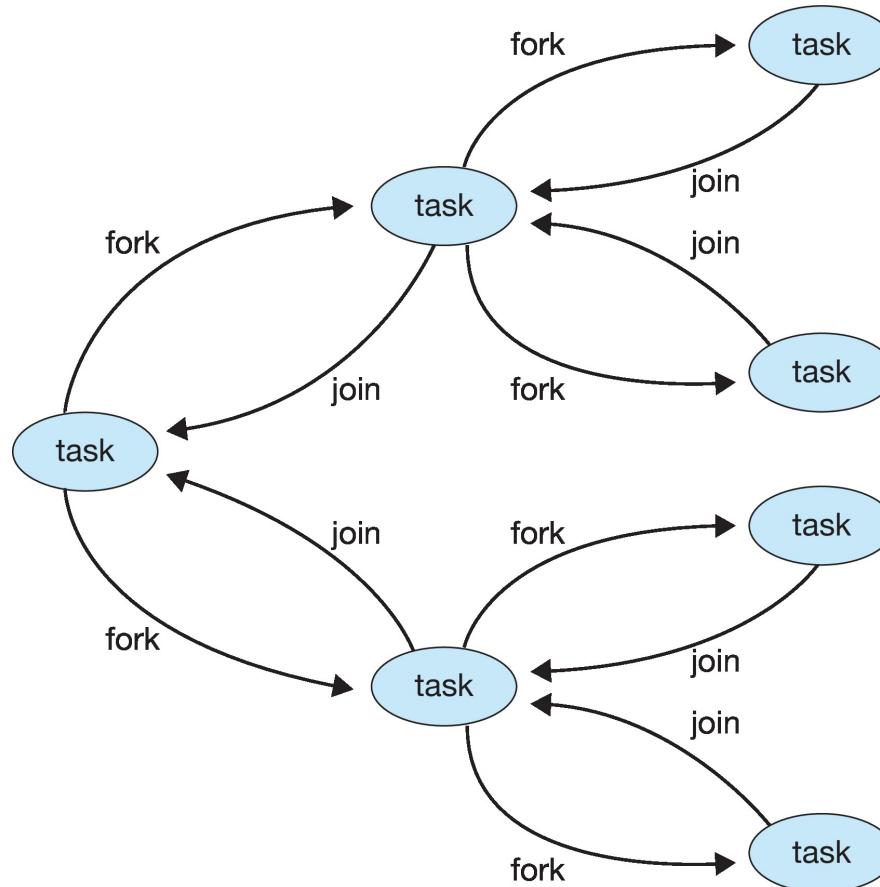
        result1 = join(subtask1)
        result2 = join(subtask2)

    return combined results
```





Fork-Join Parallelism





Fork-Join Parallelism in Java

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

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





Fork-Join Parallelism in Java

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

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

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

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

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

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

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

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

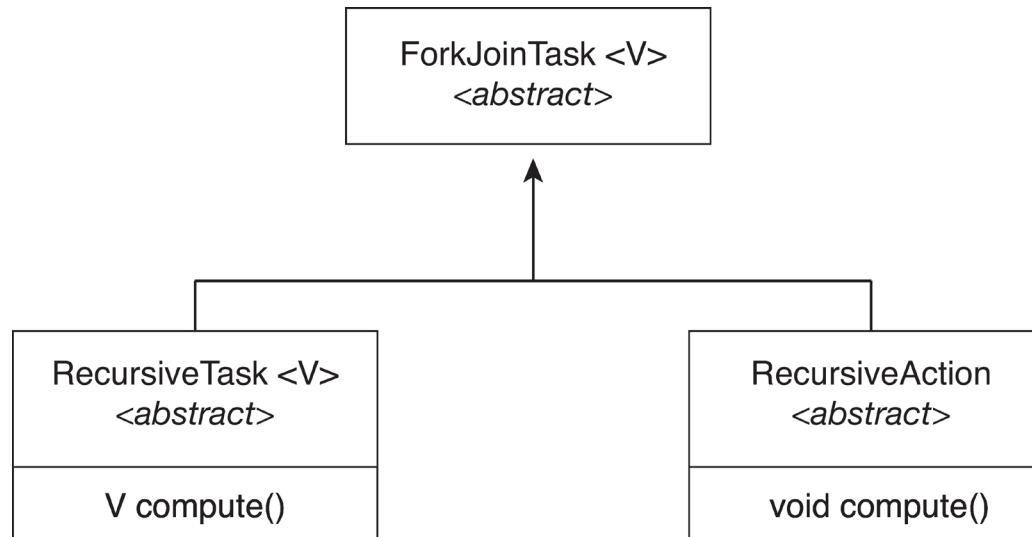
            return rightTask.join() + leftTask.join();
        }
    }
}
```





Fork-Join Parallelism in Java

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





OpenMP

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

#pragma omp parallel

Create as many threads as there are cores

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

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

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

    /* sequential code */

    return 0;
}
```





- Run the for loop in parallel

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





Grand Central Dispatch

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

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

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





Grand Central Dispatch

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





Grand Central Dispatch

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

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

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





Intel Threading Building Blocks (TBB)

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

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

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

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





Threading Issues

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





Semantics of fork() and exec()

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





Signal Handling

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





Signal Handling (Cont.)

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





Thread Cancellation

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

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);

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





Thread Cancellation (Cont.)

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

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

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





Thread Cancellation in Java

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

```
Thread worker;
```

```
    . . .
```

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

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

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





Thread-Local Storage

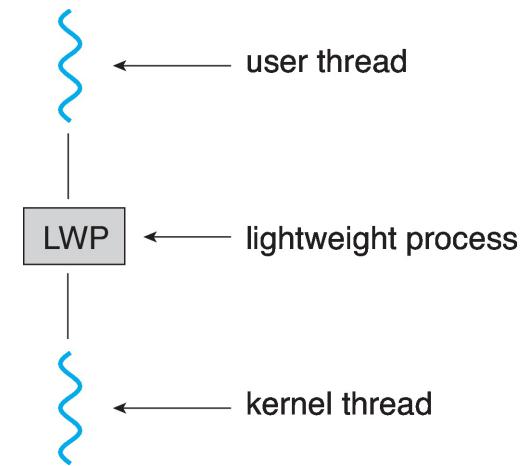
- **Thread-local storage (TLS)** allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
 - Local variables visible only during single function invocation
 - TLS visible across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





Scheduler Activations

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





Operating System Examples

- Windows Threads
- Linux Threads





Windows Threads

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





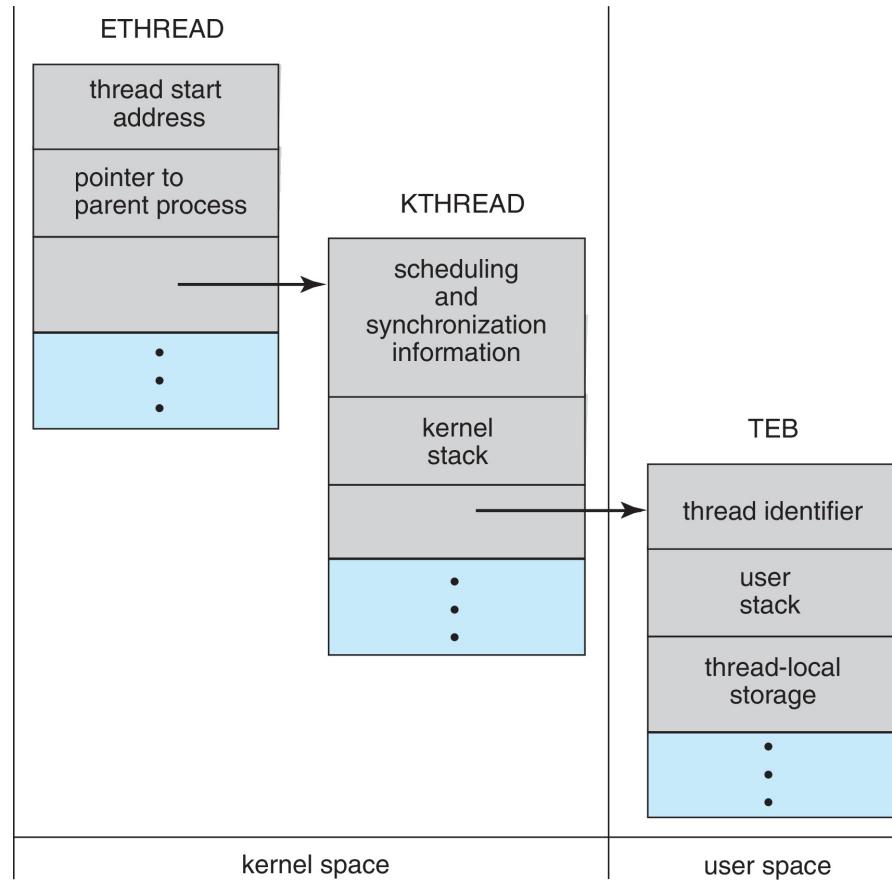
Windows Threads (Cont.)

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





Windows Threads Data Structures





Linux Threads

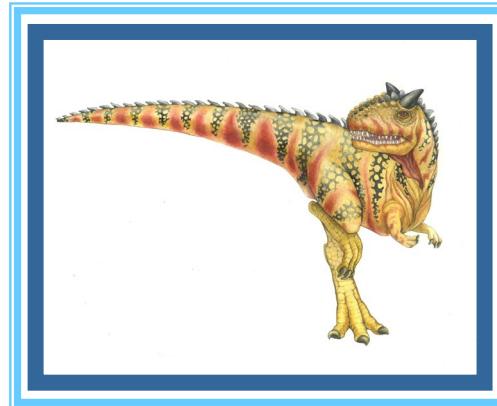
- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through `clone()` system call
- `clone()` allows a child task to share the address space of the parent task (process)
 - Flags control behavior

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

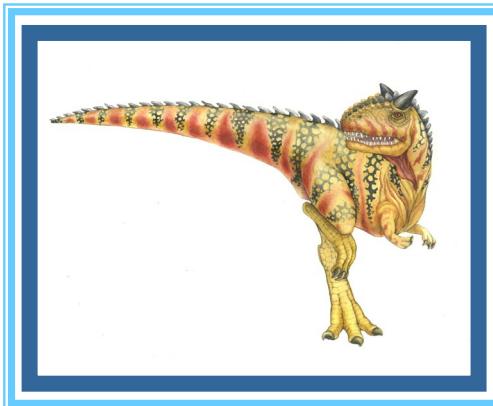
- `struct task_struct` points to process data structures (shared or unique)



End of Chapter 4



Chapter 5: CPU Scheduling





Outline

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





Objectives

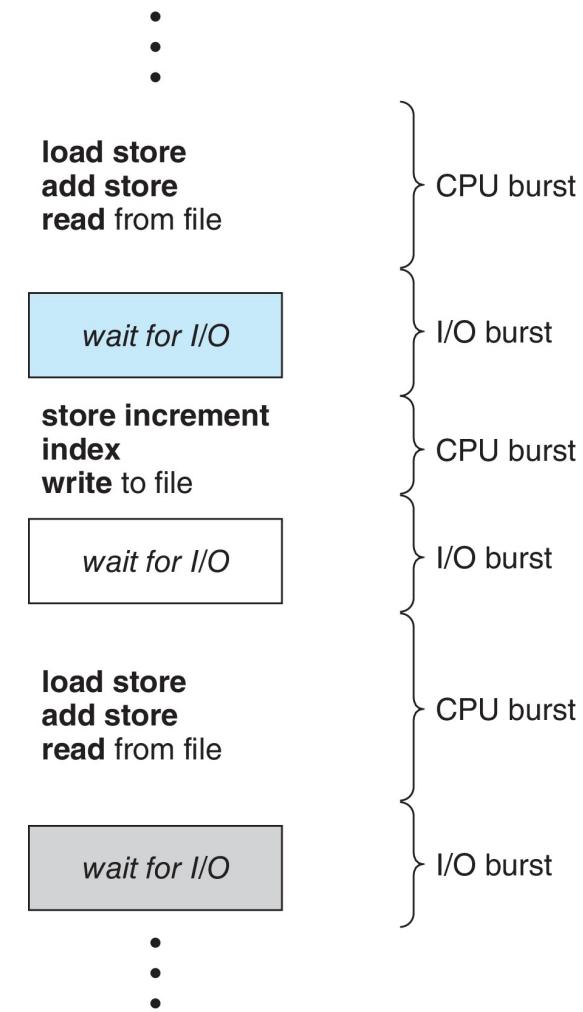
- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms
- Describe the scheduling algorithms used in the Windows, Linux, and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





Basic Concepts

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

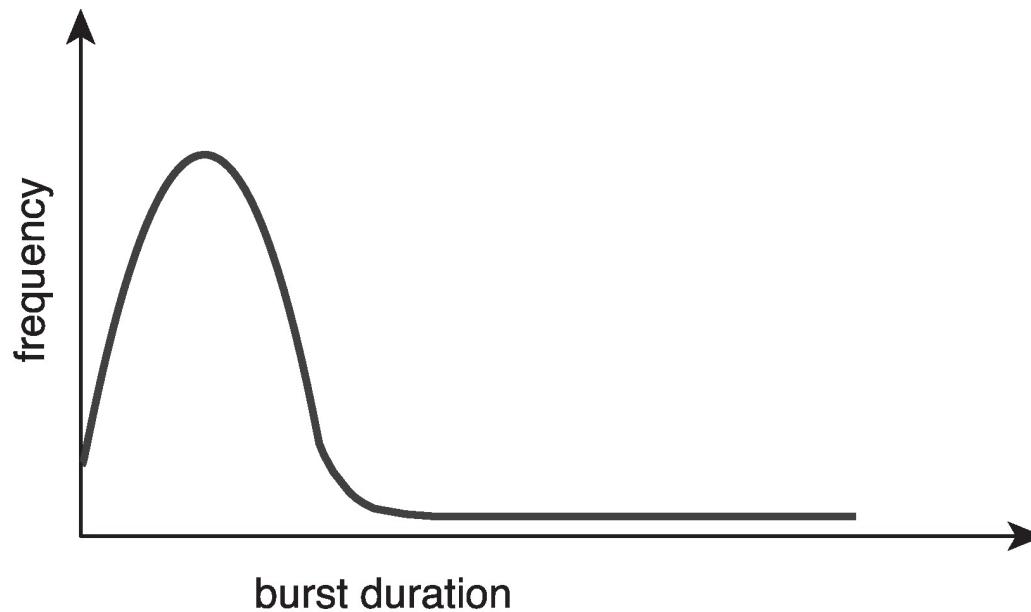




Histogram of CPU-burst Times

Large number of short bursts

Small number of longer bursts





CPU Scheduler

- The **CPU scheduler** selects from among the processes in ready queue, and allocates a CPU core to one of them
 - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
 1. Switches from running to waiting state
 2. Switches from running to ready state
 3. Switches from waiting to ready
 4. Terminates
- For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution.
- For situations 2 and 3, however, there is a choice.





Preemptive and Nonpreemptive Scheduling

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





Preemptive Scheduling and Race Conditions

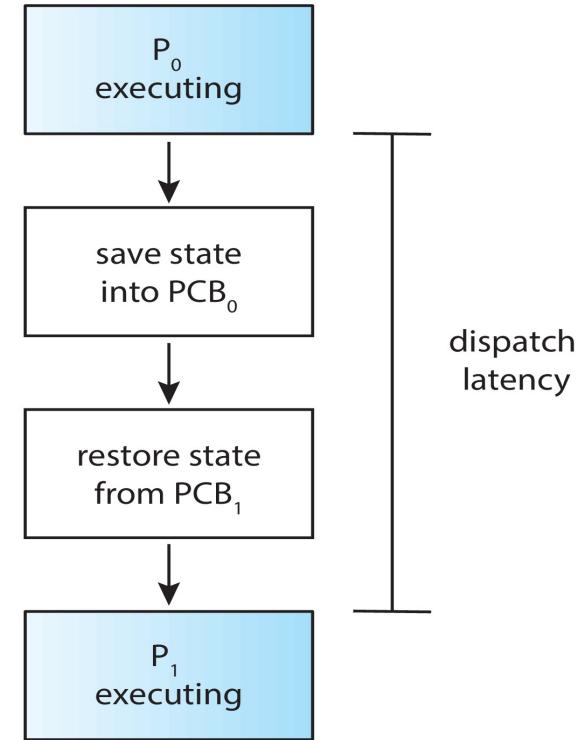
- Preemptive scheduling can result in race conditions when data are shared among several processes.
- Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are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





Dispatcher

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





Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced.





Scheduling Algorithm Optimization Criteria

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





First- Come, First-Served (FCFS) Scheduling

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

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



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





FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is:



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





Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
 - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
 - The difficulty is knowing the length of the next CPU request
 - Could ask the user





Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
 - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
- Preemptive version called **shortest-remaining-time-first**
- How do we determine the length of the next CPU burst?
 - Could ask the user
 - Estimate

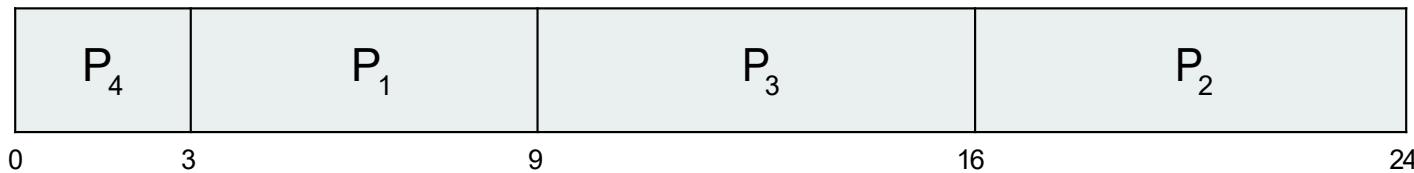




Example of SJF

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

- SJF scheduling chart



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





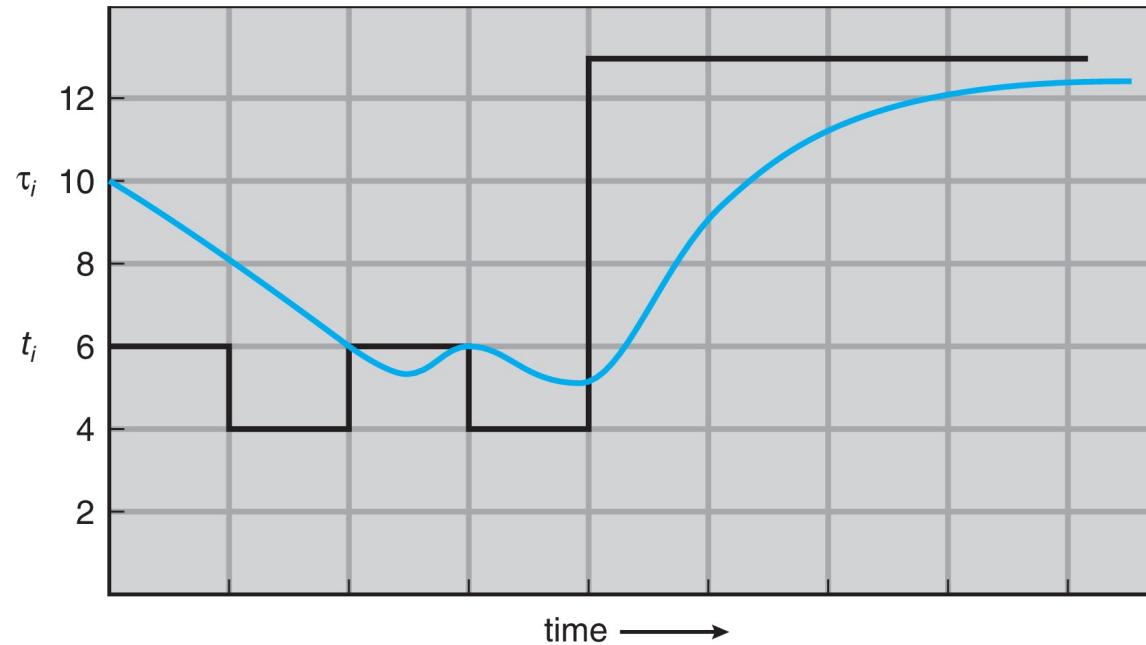
Determining Length of Next CPU Burst

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





Prediction of the Length of the Next CPU Burst



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





Examples of Exponential Averaging

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

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

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



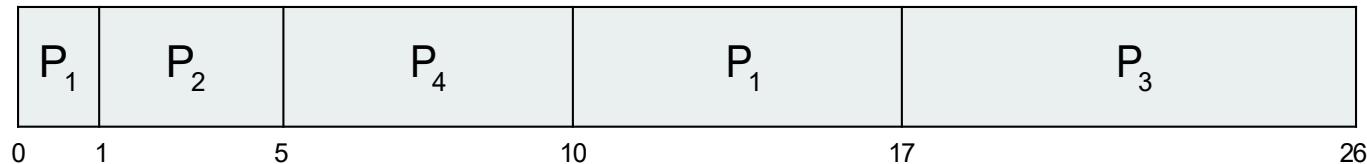


Example of Shortest-remaining-time-first

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

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

- Preemptive SJF Gantt Chart*



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





Round Robin (RR)

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

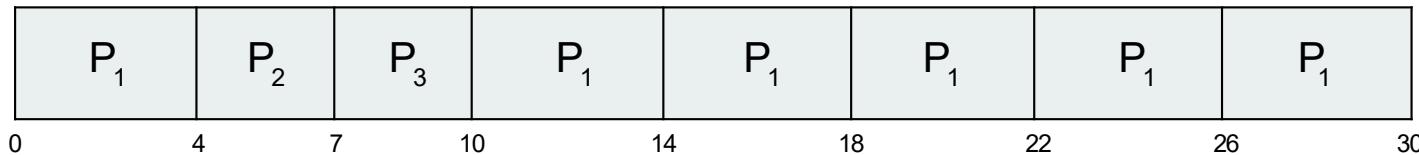




Example of RR with Time Quantum = 4

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

- The Gantt chart is:

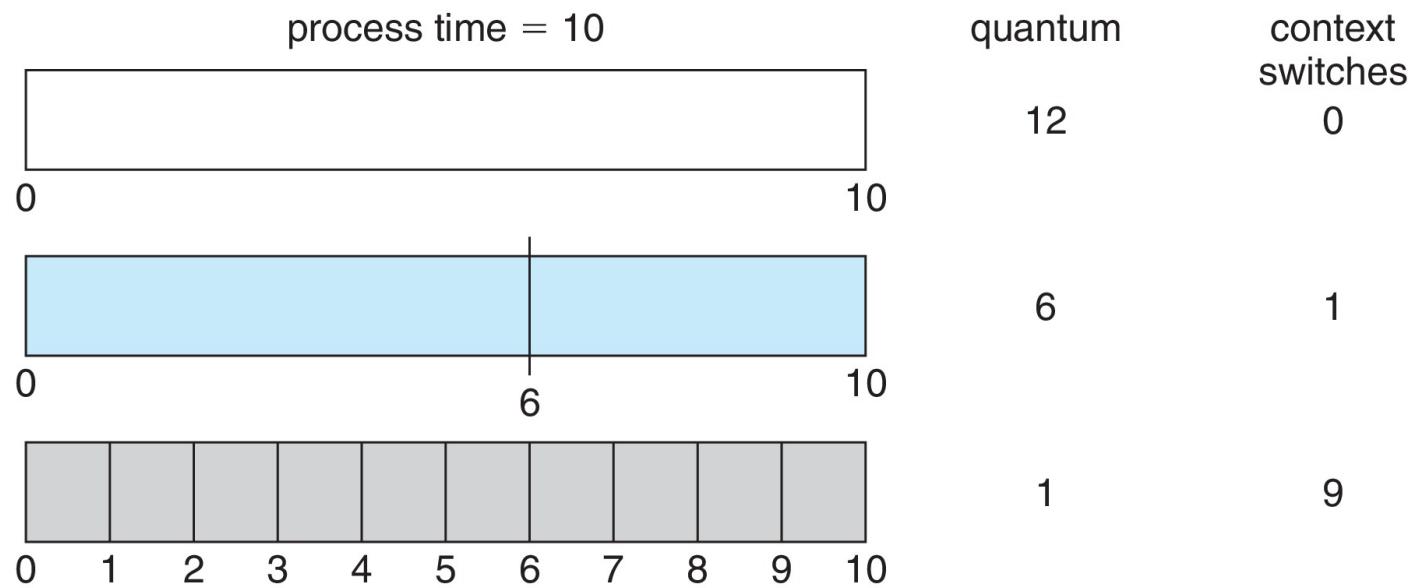


- Typically, higher average turnaround than SJF, but better **response**
- q should be large compared to context switch time
 - q usually 10 milliseconds to 100 milliseconds,
 - Context switch < 10 microseconds



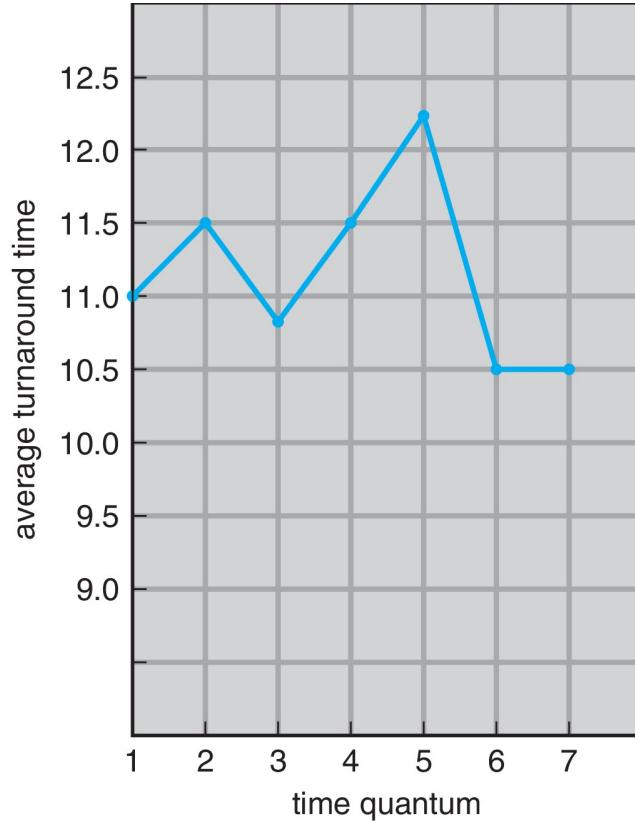


Time Quantum and Context Switch Time





Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts
should be shorter than q





Priority Scheduling

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

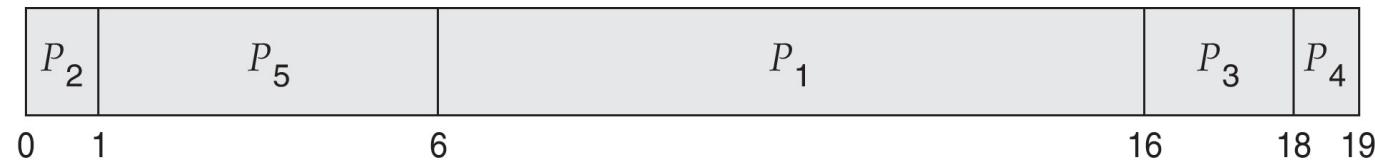




Example of Priority Scheduling

<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

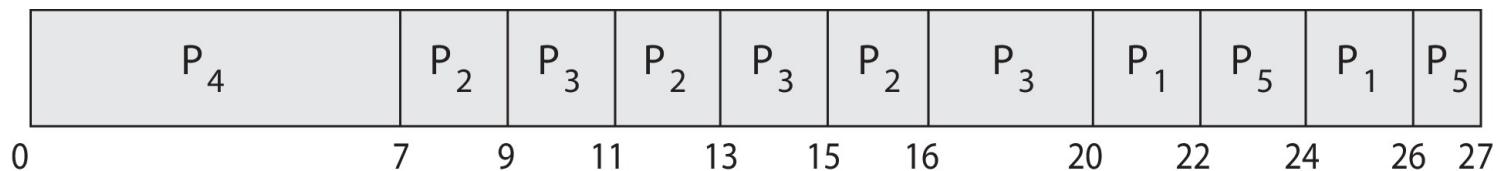




Priority Scheduling w/ Round-Robin

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

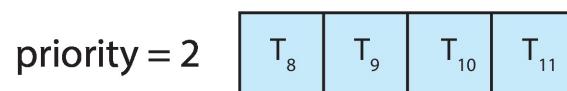
- Run the process with the highest priority. Processes with the same priority run round-robin
- Gantt Chart with time quantum = 2





Multilevel Queue

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



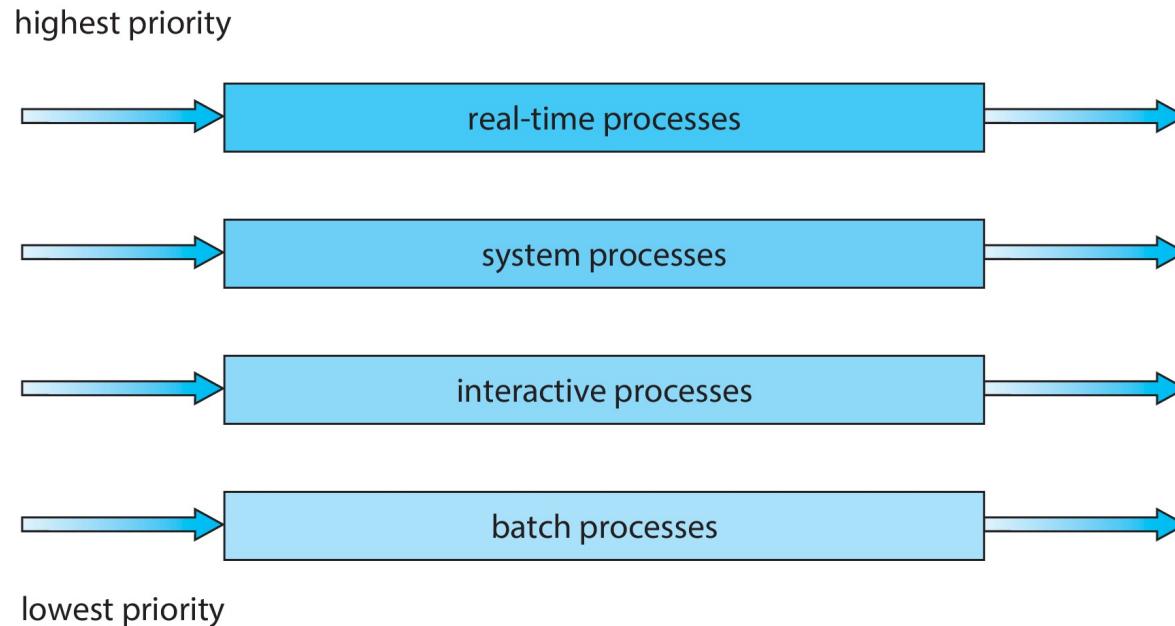
●
●
●





Multilevel Queue

- Prioritization based upon process type





Multilevel Feedback Queue

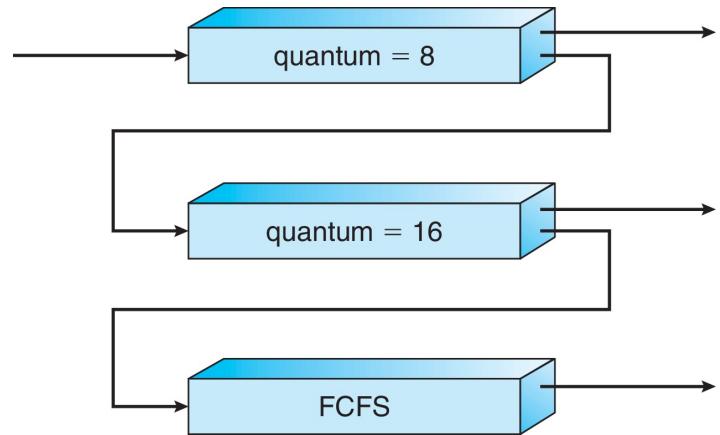
- A process can move between the various queues.
- Multilevel-feedback-queue scheduler defined by the following parameters:
 - Number of queues
 - Scheduling algorithms for each queue
 - Method used to determine when to upgrade a process
 - Method used to determine when to demote a process
 - Method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





Example of Multilevel Feedback Queue

- Three queues:
 - Q_0 – RR with time quantum 8 milliseconds
 - Q_1 – RR time quantum 16 milliseconds
 - Q_2 – FCFS
- Scheduling
 - A new process enters queue Q_0 which is served in RR
 - When it gains CPU, the process receives 8 milliseconds
 - If it does not finish in 8 milliseconds, the process is moved to queue Q_1
 - At Q_1 job is again served in RR and receives 16 additional milliseconds
 - If it still does not complete, it is preempted and moved to queue Q_2





Thread Scheduling

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





Pthread Scheduling

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





Pthread Scheduling API

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





Pthread Scheduling API

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





Multiple-Processor Scheduling

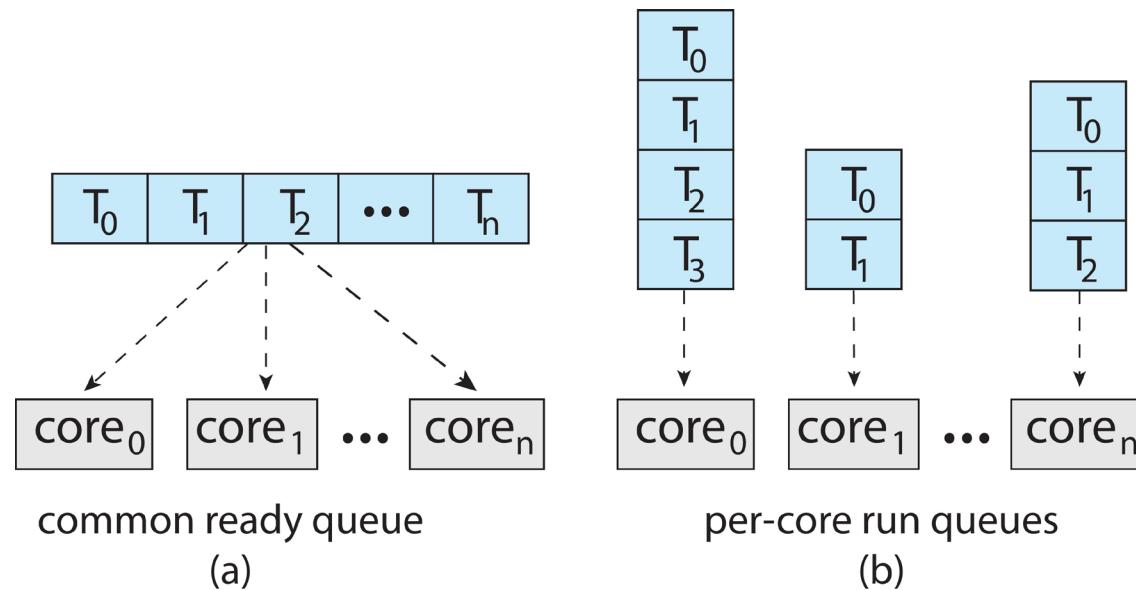
- CPU scheduling more complex when multiple CPUs are available
- Multiprocess may be any one of the following architectures:
 - Multicore CPUs
 - Multithreaded cores
 - NUMA systems
 - Heterogeneous multiprocessing





Multiple-Processor Scheduling

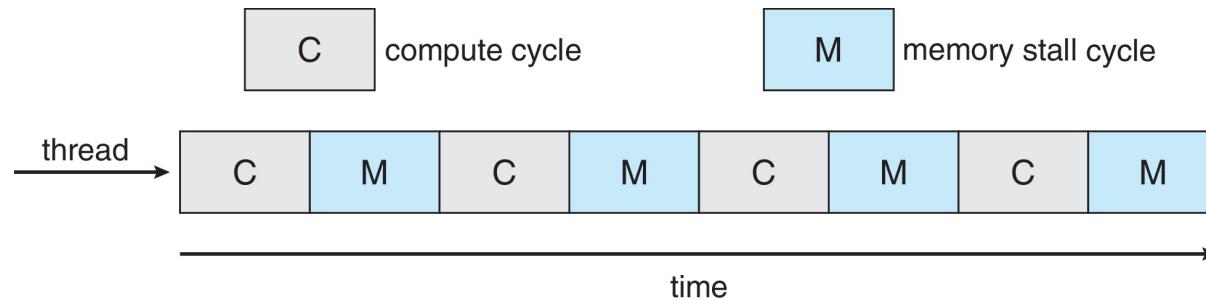
- Symmetric multiprocessing (SMP) is where each processor is self scheduling.
- All threads may be in a common ready queue (a)
- Each processor may have its own private queue of threads (b)





Multicore Processors

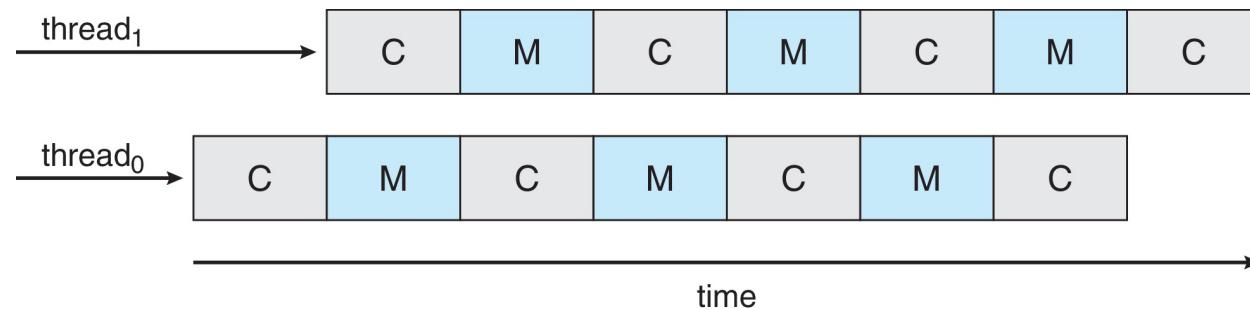
- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
 - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
- Figure





Multithreaded Multicore System

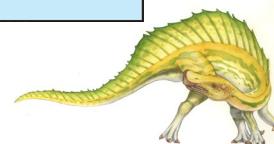
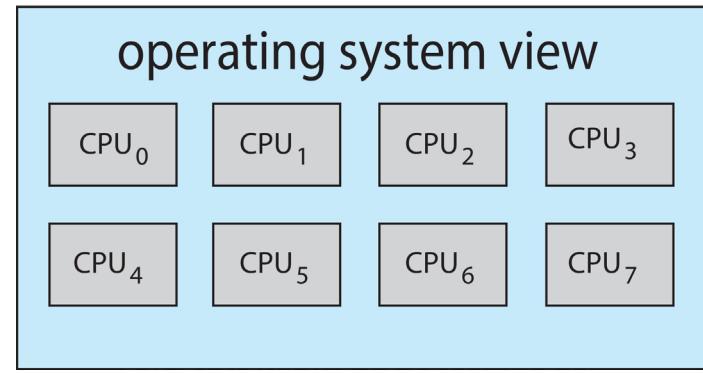
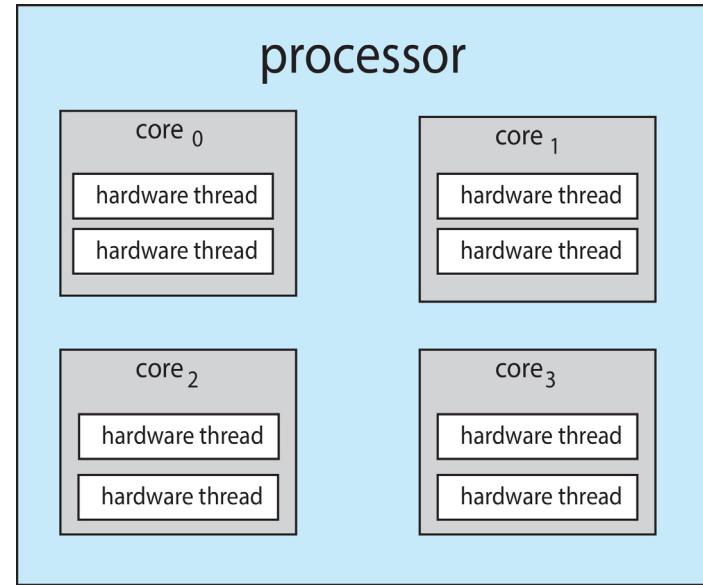
- Each core has > 1 hardware threads.
- If one thread has a memory stall, switch to another thread!
- Figure





Multithreaded Multicore System

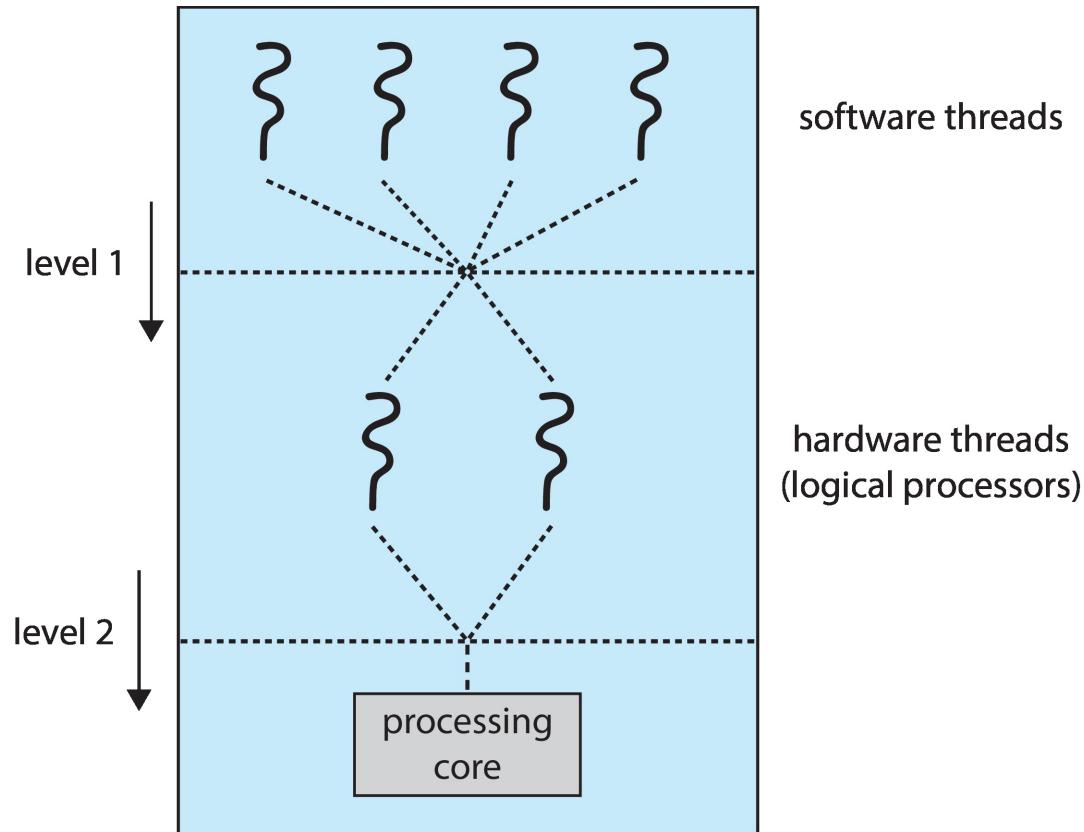
- **Chip-multithreading (CMT)** assigns each core multiple hardware threads. (Intel refers to this as **hyperthreading**.)
- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.





Multithreaded Multicore System

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





Multiple-Processor Scheduling – Load Balancing

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





Multiple-Processor Scheduling – Processor Affinity

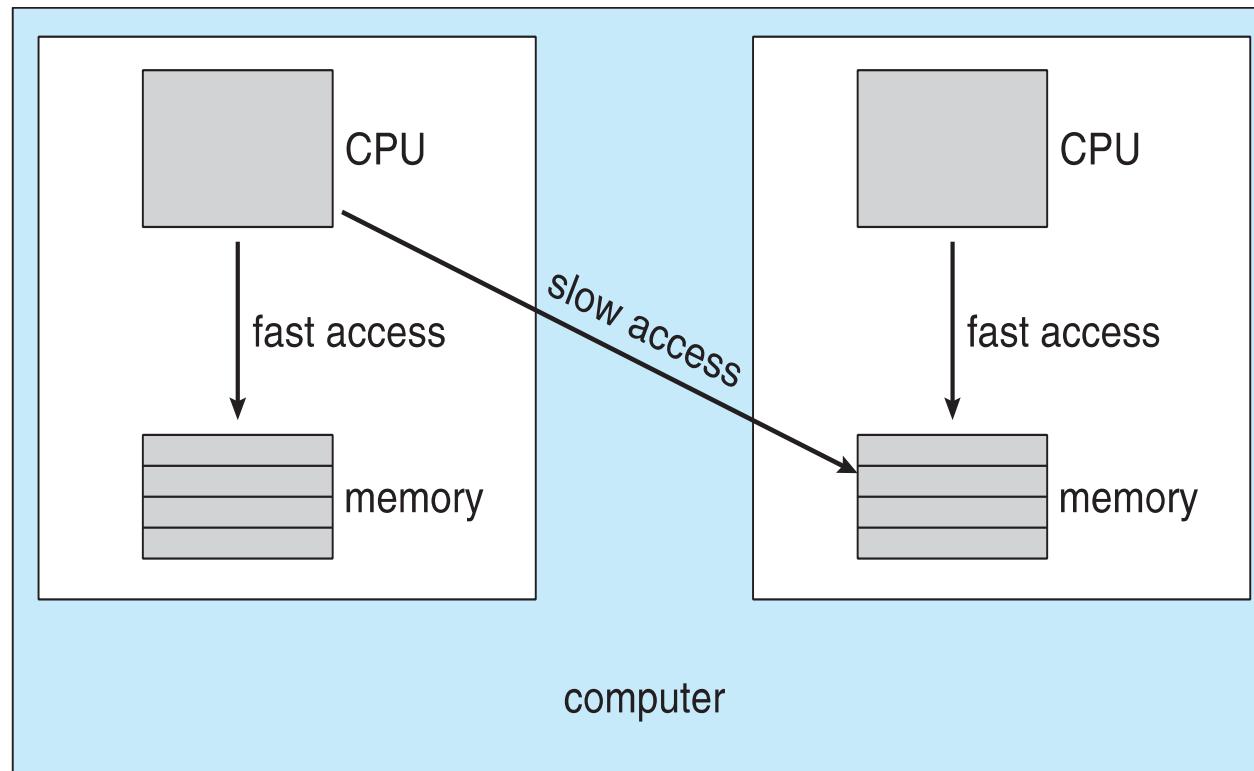
- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having affinity for a processor (i.e., “processor affinity”)
- Load balancing may affect processor affinity as a thread may be moved from one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





NUMA and CPU Scheduling

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





Real-Time CPU Scheduling

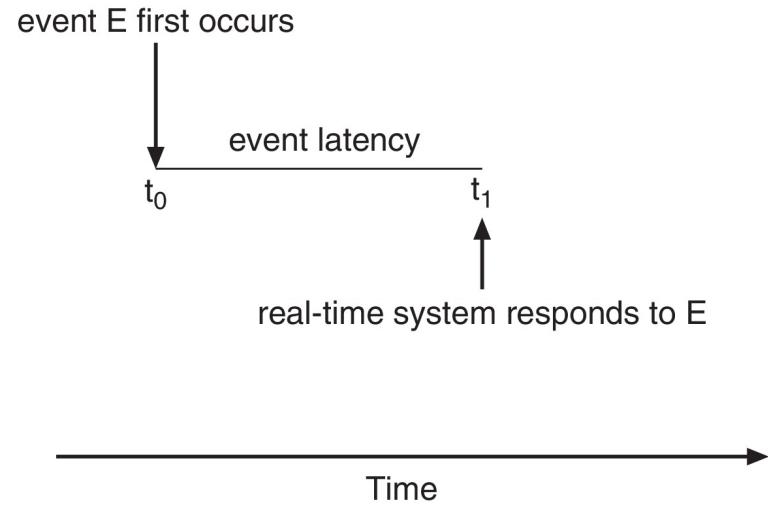
- Can present obvious challenges
- **Soft real-time systems** – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
- **Hard real-time systems – task must be serviced by its deadline**





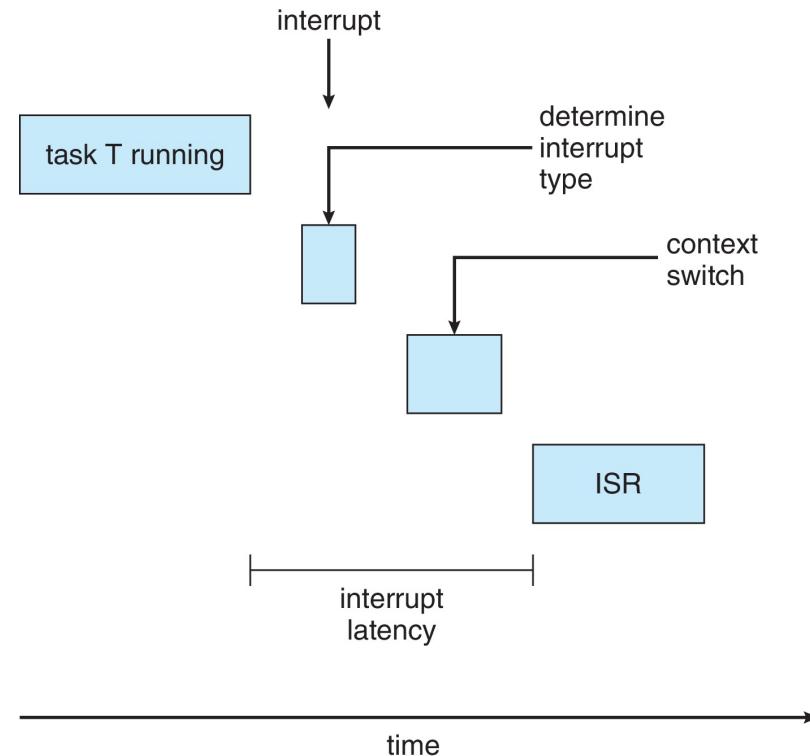
Real-Time CPU Scheduling

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





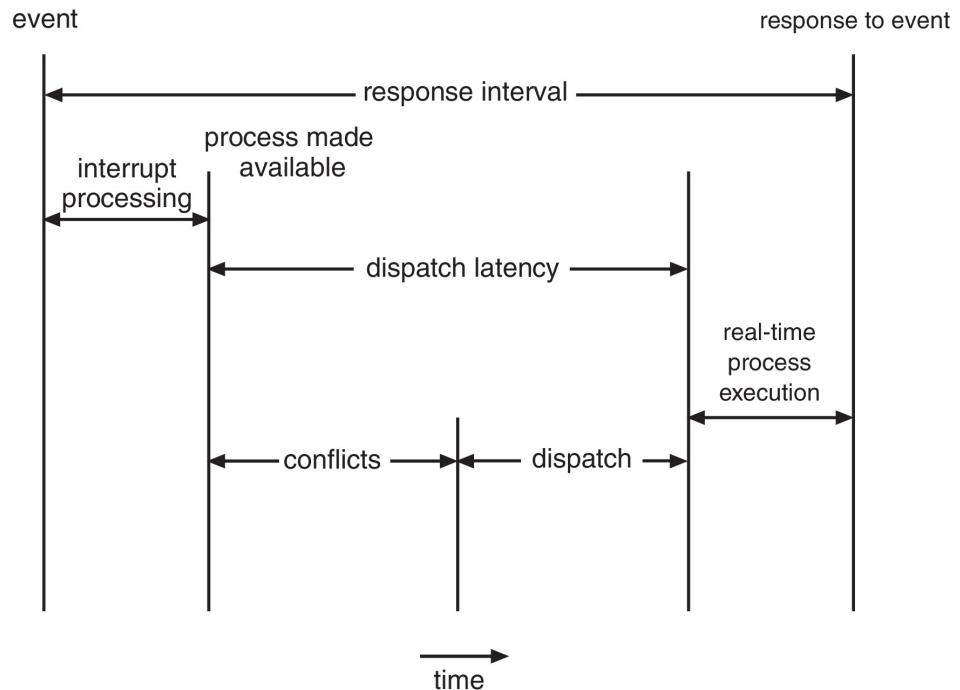
Interrupt Latency





Dispatch Latency

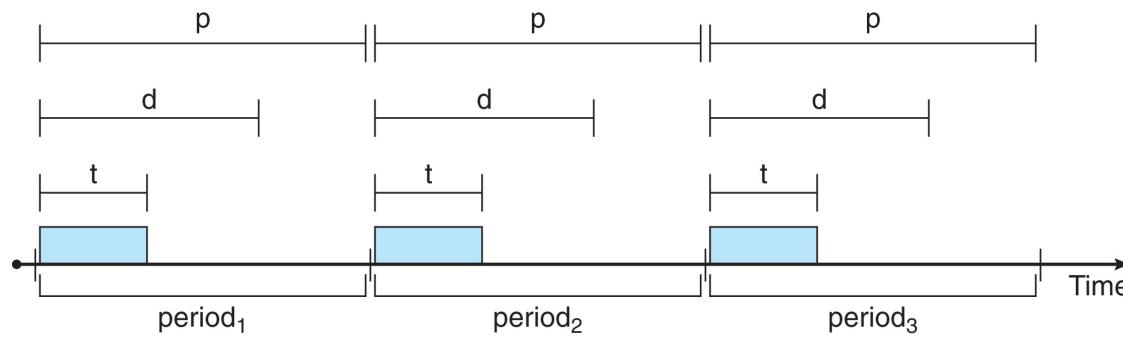
- Conflict phase of dispatch latency:
 1. Preemption of any process running in kernel mode
 2. Release by low-priority process of resources needed by high-priority processes





Priority-based Scheduling

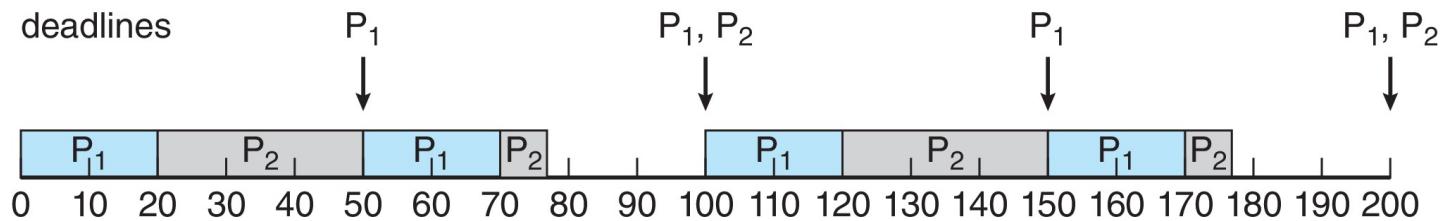
- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
 - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
 - Has processing time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





Rate Monotonic Scheduling

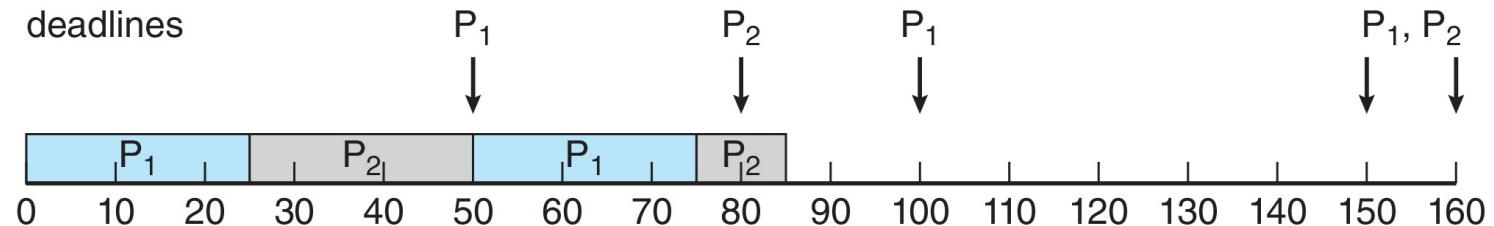
- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P_1 is assigned a higher priority than P_2 .





Missed Deadlines with Rate Monotonic Scheduling

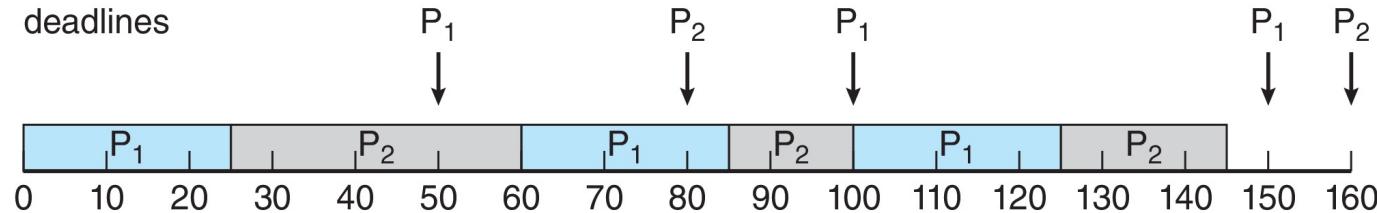
- Process P_2 misses finishing its deadline at time 80
- Figure





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
- Figure





Proportional Share Scheduling

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





POSIX Real-Time Scheduling

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





POSIX Real-Time Scheduling API

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





POSIX Real-Time Scheduling API (Cont.)

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

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

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

}

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





Operating System Examples

- Linux scheduling
- Windows scheduling
- Solaris scheduling





Linux Scheduling Through Version 2.5

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





Linux Scheduling in Version 2.6.23 +

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





Linux Scheduling in Version 2.6.23 + (Cont.)

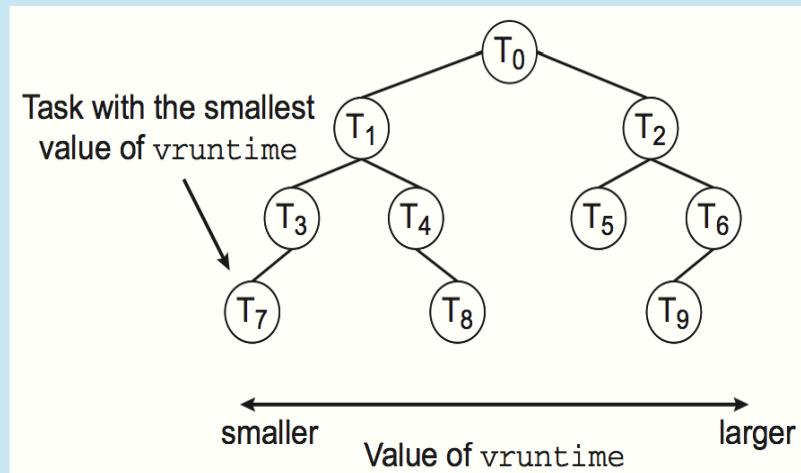
- Quantum calculated based on **nice value** from -20 to +19
 - Lower value is higher priority
 - Calculates **target latency** – interval of time during which task should run at least once
 - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual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





CFS Performance

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



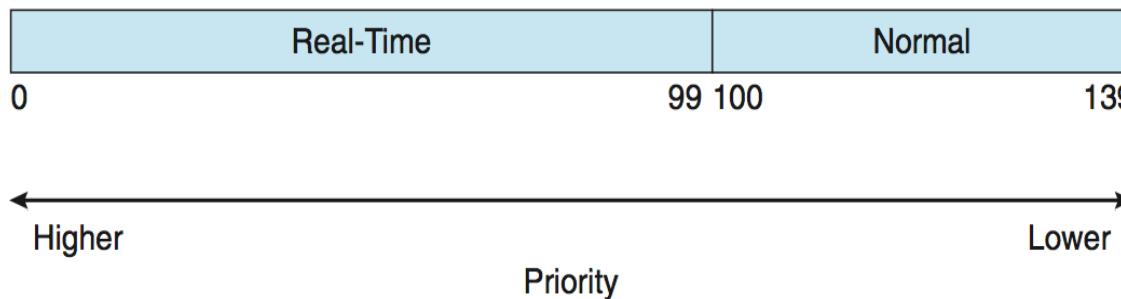
When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and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





Linux Scheduling (Cont.)

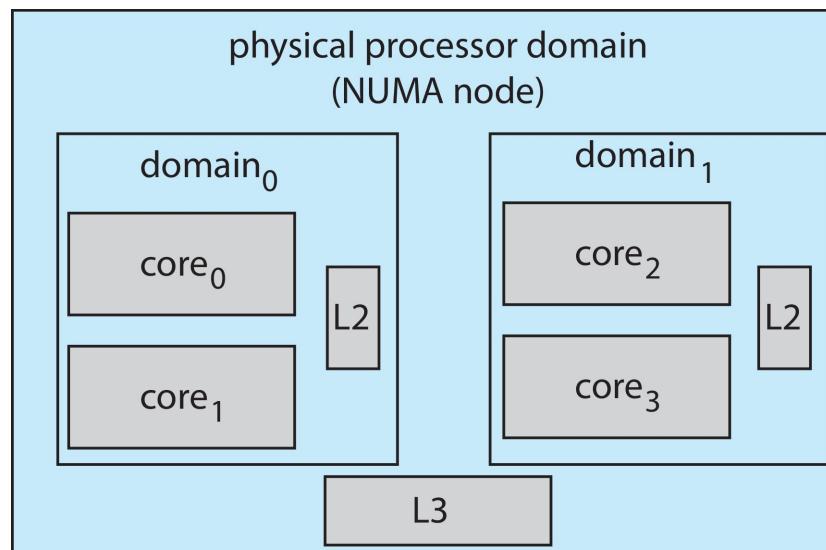
- Real-time scheduling according to POSIX.1b
 - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139





Linux Scheduling (Cont.)

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





Windows Scheduling

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





Windows Priority Classes

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

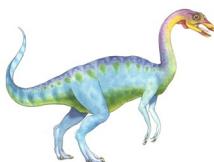




Windows Priority Classes (Cont.)

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





Windows Priorities

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





Solaris

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





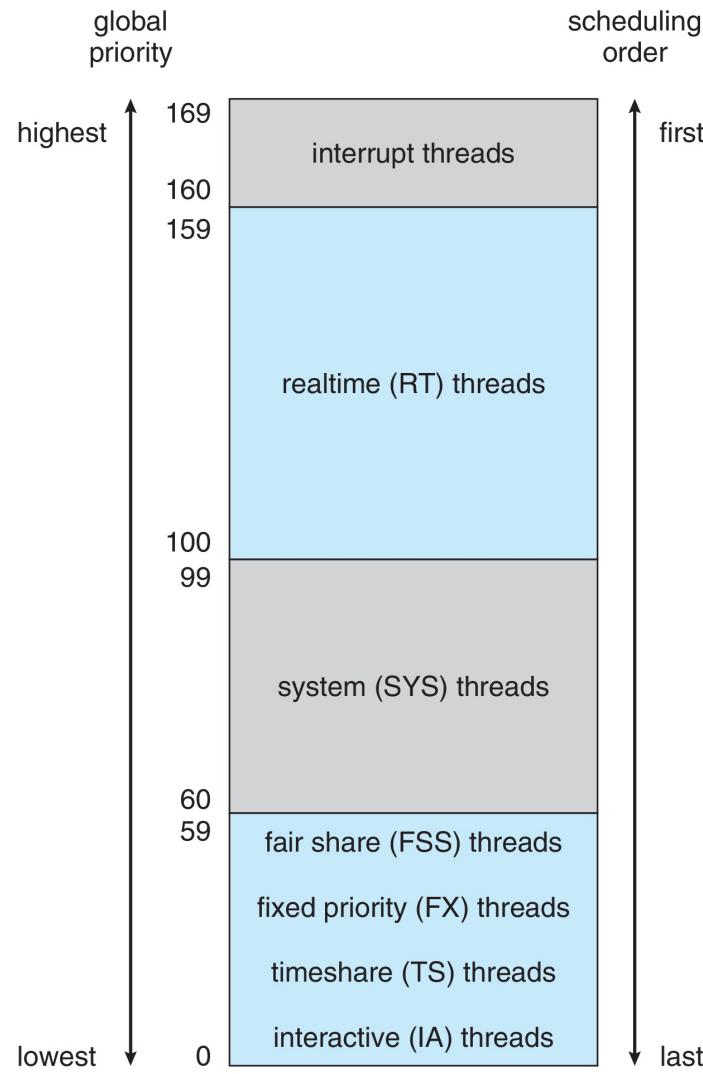
Solaris Dispatch Table

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





Solaris Scheduling





Solaris Scheduling (Cont.)

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





Algorithm Evaluation

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

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



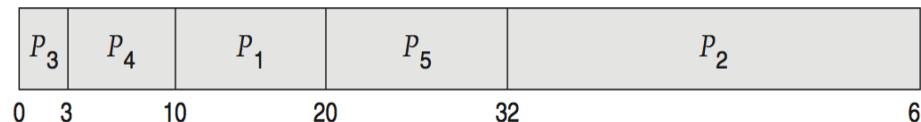


Deterministic Evaluation

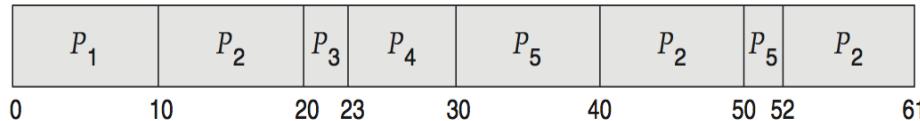
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
 - FCS is 28ms:



- 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





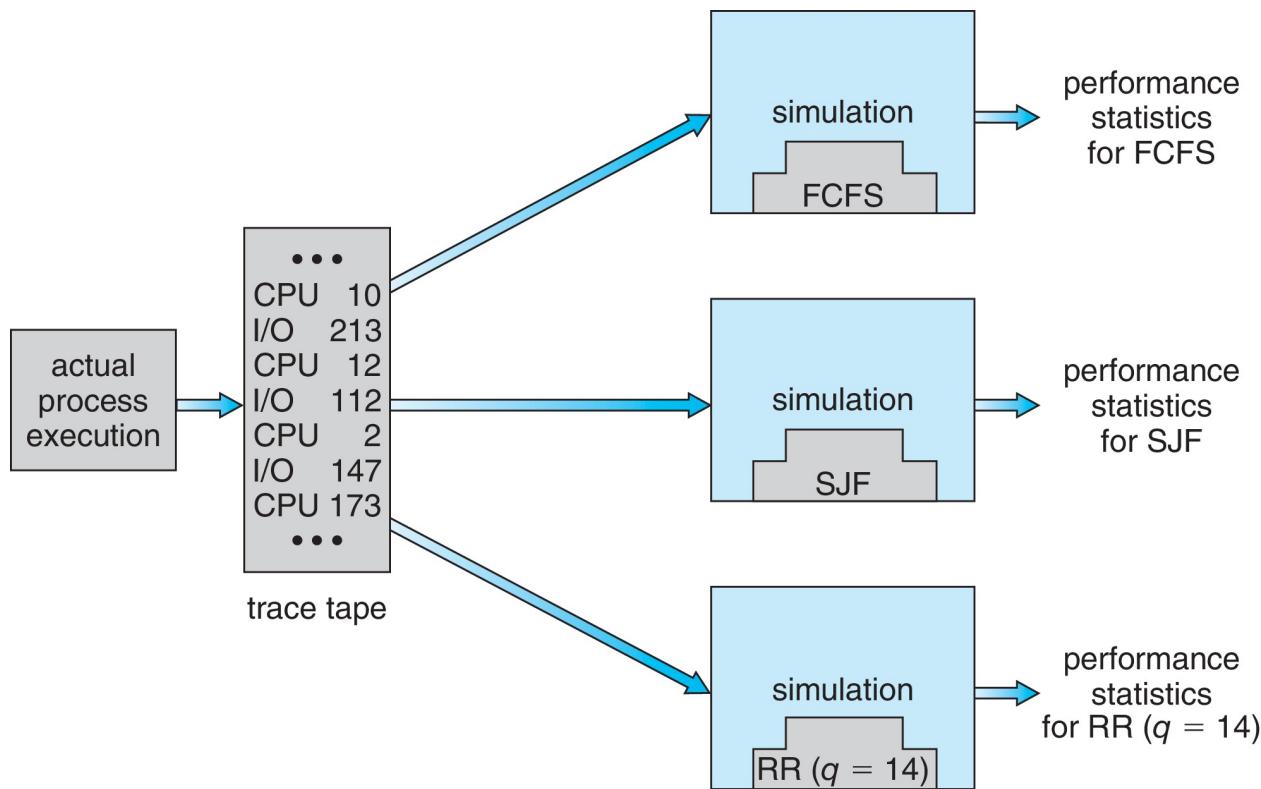
Simulations

- 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



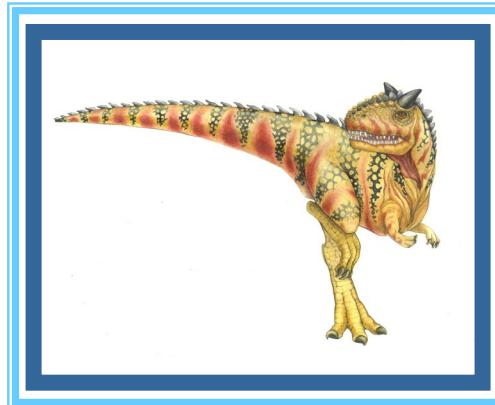


Implementation

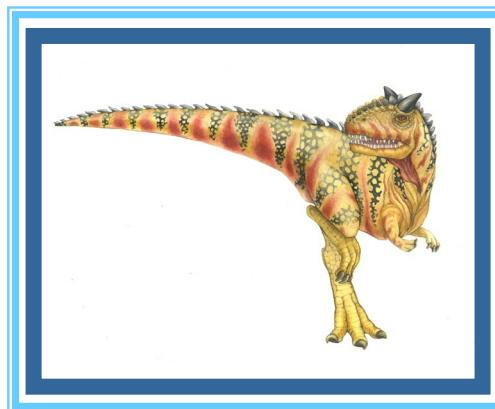
- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
 - High cost, high risk
 - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary



End of Chapter 5



Chapter 6: Synchronization Tools





Too much milk?

	Alice	Bob
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!





x is a global variable initialized to 0

Thread 1

```
void foo()
{
    x++;
}
```

Thread 2

```
void bar()
{
    x--;
}
```

- After thread 1 and thread 2 finishes, what is the value of x?
 - could be 0, 1, -1
 - Why?





Synchronization Motivation

- **Threads cooperate in multithreaded programs**
 - To share resources, access shared data structures
 - To coordinate their execution
- **For correctness, we need to control this cooperation**
 - Thread schedule is **non-deterministic**
 - ▶ Scheduling is not under program control
 - ▶ Threads **interleave executions arbitrarily** and at different rates
 - ▶ Behavior changes when re-run program
 - Multi-word operations are not atomic
 - Compiler/hardware instruction reordering





Shared Resources

We initially focus on coordinating access to shared resources

- **Basic problem**
 - If two concurrent threads (processes) are accessing a shared variable, and that variable is read/modified/written by those threads, then access to the variable must be controlled to avoid erroneous behavior
- **Over the next couple of lectures, we will look at**
 - Mechanisms to control access to shared resources
 - ▶ Locks, mutexes, sémaphores, monitors, condition variables, etc.
 - Patterns for coordinating accesses to shared resources
 - ▶ Bounded buffer, producer-consumer, etc.





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

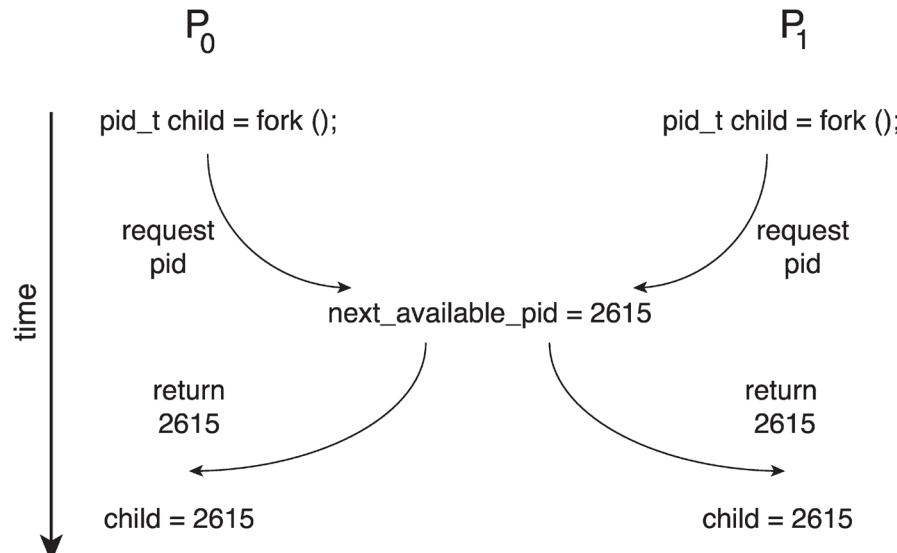
- Processes can execute concurrently
 - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- We illustrated in chapter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





Race Condition

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



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





Critical Section Problem

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





Critical Section

- General structure of process P_i

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





Critical-Section Problem (Cont.)

Requirements for solution to critical-section problem

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





HOW CAN WE SOLVE THIS?





Interrupt-based Solution

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





Software Solution 1

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





Algorithm for Process P_i

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





Correctness of the Software Solution

- Mutual exclusion is preserved

P_i enters critical section only if:

turn = i

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

- What about the Progress requirement?
- What about the Bounded-waiting requirement?





Peterson's Solution

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





Algorithm for Process P_i

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





Correctness of Peterson's Solution

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

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





Peterson's Solution and Modern Architecture

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





Modern Architecture Example

- Two threads share the data:

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

- Thread 1 performs

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

- Thread 2 performs

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

- What is the expected output?

100





Modern Architecture Example (Cont.)

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

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

for Thread 2 may be reordered

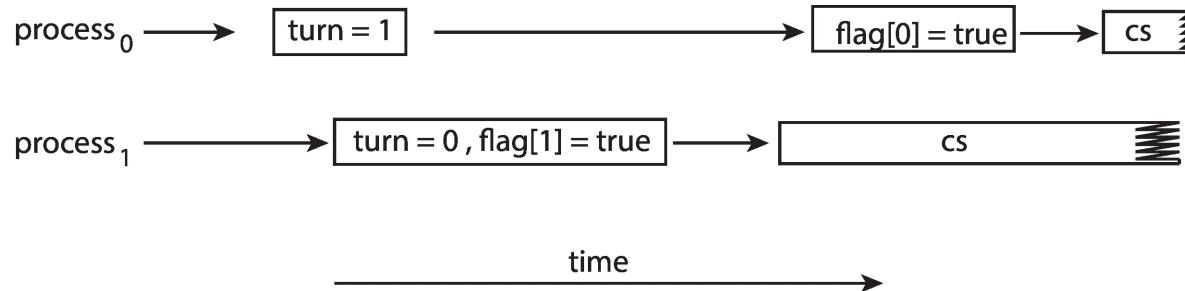
- If this occurs, the output may be 0!





Peterson's Solution Revisited

- The effects of instruction reordering in Peterson's Solution



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





Memory Barrier

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





Memory Barrier Instructions

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





Memory Barrier Example

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

```
while (!flag)
    memory_barrier();
print x
```
- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```
- For Thread 1 we are guaranteed that that the value of flag is loaded before the value of x.
- For Thread 2 we ensure that the assignment to x occurs before the assignment flag.





Synchronization Hardware

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





Hardware Instructions

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





The test_and_set Instruction

- Definition

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

- Properties

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





Solution Using test_and_set()

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

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

- Does it solve the critical-section problem?





The compare_and_swap Instruction

- Definition

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

- Properties

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





Solution using compare_and_swap

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

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

    /* critical section */

    lock = 0;

    /* remainder section */
}
```

- Does it solve the critical-section problem?





Bounded-waiting with compare-and-swap

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





Atomic Variables

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





Atomic Variables

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

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





Mutex Locks

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





Solution to CS Problem Using Mutex Locks

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





Semaphore

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

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

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

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





Semaphore (Cont.)

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





Semaphore Usage Example

- Solution to the CS Problem

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

```
    wait(mutex);
```

CS

```
    signal(mutex);
```

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

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

$P1:$

```
    S1;
```

```
    signal(synch);
```

$P2:$

```
    wait(synch);
```

```
    S2;
```





Semaphore Implementation

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





Semaphore Implementation with no Busy waiting

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





Implementation with no Busy waiting (Cont.)

- Waiting queue

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





Implementation with no Busy waiting (Cont.)

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





Problems with Semaphores

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





Monitors

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

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

    procedure P2 (...) { .... }

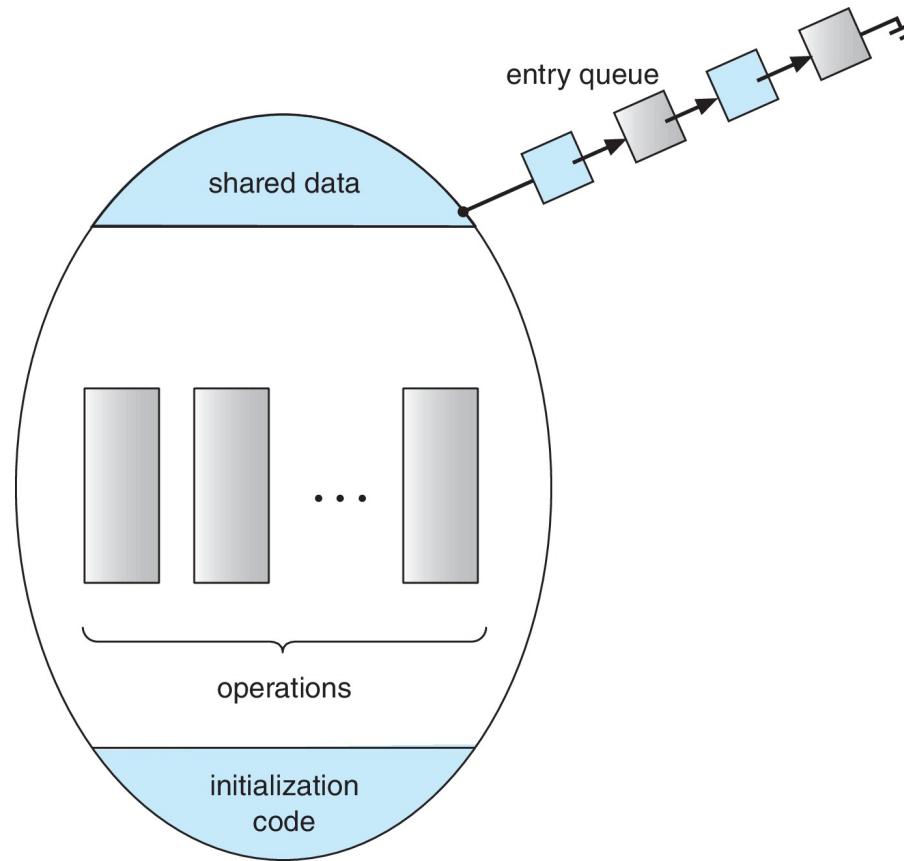
    procedure Pn (...) { ..... }

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





Schematic view of a Monitor





Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex  
mutex = 1
```

- Each procedure P is replaced by

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

- Mutual exclusion within a monitor is ensured





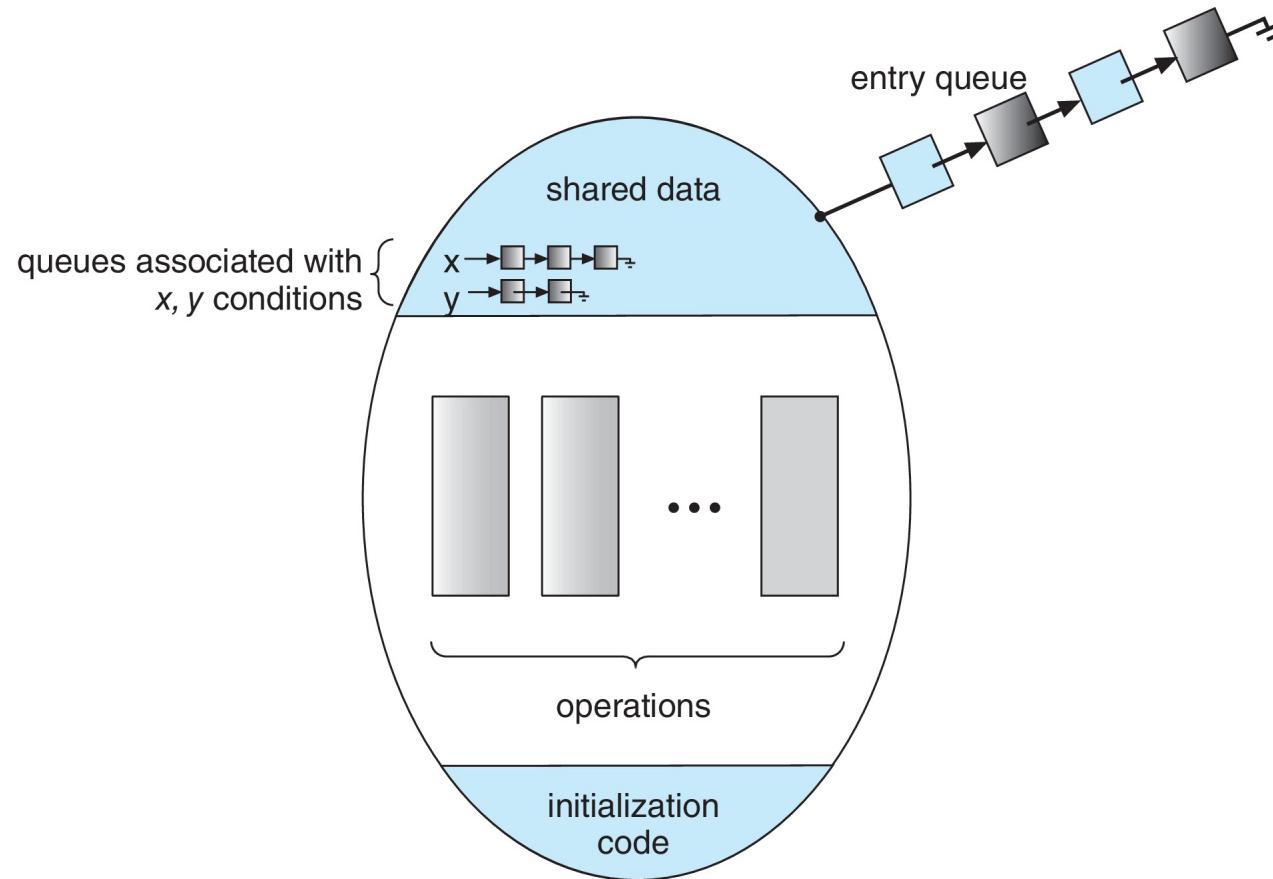
Condition Variables

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





Monitor with Condition Variables





Usage of Condition Variable Example

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

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

```
s1;  
  
done = true;  
  
x.signal();
```

- F2:**


```
if done = false  
  
x.wait()  
  
s2;
```





Monitor Implementation Using Semaphores

- Variables

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

- Each function **P** will be replaced by

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

- Mutual exclusion within a monitor is ensured





Implementation – Condition Variables

- For each condition variable x , we have:

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

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

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





Implementation (Cont.)

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

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





Resuming Processes within a Monitor

- If several processes queued on condition variable **x**, and **x.signal()** is executed, which process should be resumed?
- FCFS frequently not adequate
- Use the **conditional-wait** construct of the form

x.wait(c)

where:

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





Single Resource allocation

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

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

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

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





Single Resource allocation

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

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

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





A Monitor to Allocate Single Resource

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





Single Resource Monitor (Cont.)

- Usage:
acquire

...
release

- Incorrect use of monitor operations

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





Liveness

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





Liveness

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

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

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



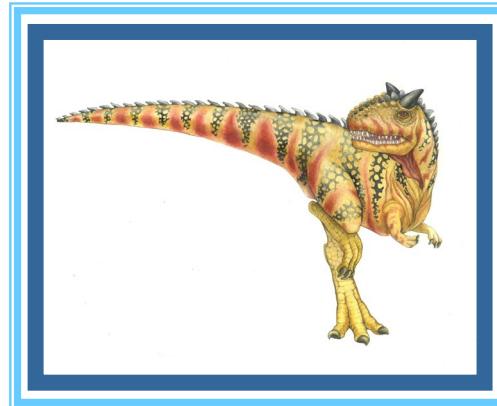


Liveness

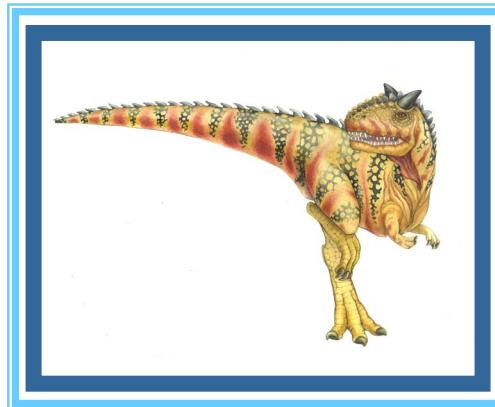
- Other forms of deadlock:
- **Starvation** – indefinite blocking
 - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
 - Solved via **priority-inheritance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



End of Chapter 6



Chapter 8: Deadlocks



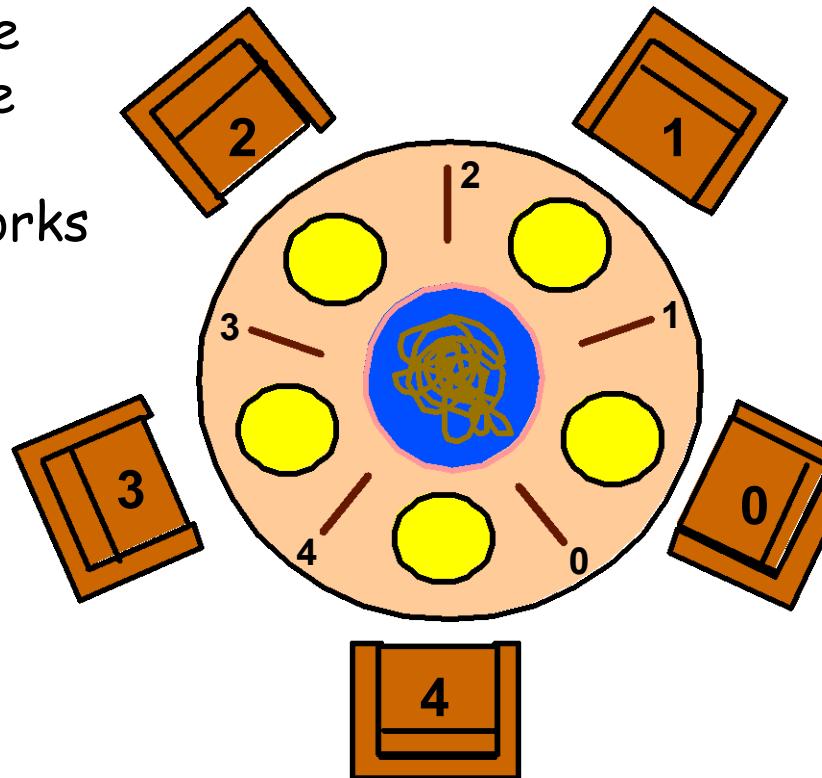
Deadlock

- ◆ **Synchronization is a live gun**
 - We can easily shoot ourselves in the foot
 - Incorrect use of synchronization can block all processes
 - You have likely been intuitively avoiding this situation already
- ◆ **If one process tries to access a resource that a second process holds, and vice-versa, they can never make progress**
- ◆ **We call this situation deadlock, and we'll look at:**
 - Definition and conditions necessary for deadlock
 - Representation of deadlock conditions
 - Approaches to dealing with deadlock

The Dining Philosophers

Five philosophers sit around a circular table. Each philosopher spends his life alternately **thinking** and **eating**. In the centre of the table is a large bowl of spaghetti. A philosopher needs two forks to eat a helping of spaghetti.

One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.



The Dining Philosophers

```
Semaphore[] fork = {1,1,1,1,1};
```

```
process philosopher[i=0..3]
while (true) {
    down(fork[i]);
    down(fork[i+1]);
    eat;
    up(fork[i]);
    up(fork[i+1]);
    think;
}
end philosopher;
```

```
process philosopher[4]
while (true) {
    down(fork[0]);
    down(fork[4]);
    eat;
    up(fork[0]);
    up(fork[4]);
    think;
}
end philosopher;
```

Why does philosopher[4] have to be different from the rest ?

Deadlock Definition

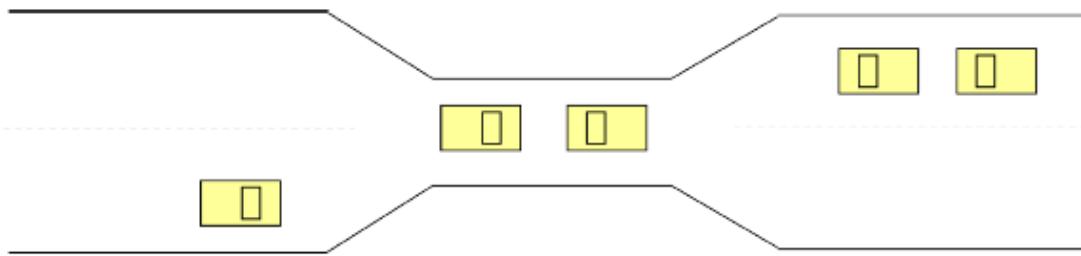
◆ Deadlock is a problem that can arise:

- When processes compete for access to limited resources
- When processes are incorrectly synchronized

◆ Definition:

- Deadlock exists among a set of processes if every process is waiting for an event that can be caused only by another process in the set.

Deadlocks w/o Computers



- **Real issue is *resources* & how required**
- **E.g., bridge only allows traffic in one direction**
 - Each section of a bridge can be viewed as a resource.
 - If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
 - Several cars may have to be backed up if a deadlock occurs.
 - Starvation is possible.

Conditions for Deadlock

1. **Mutual exclusion** – At least one resource must be held in a non-sharable mode
2. **Hold and wait** – There must be one process holding one resource and waiting for another resource
3. **No preemption** – Resources cannot be preempted (critical sections cannot be aborted externally)
4. **Circular wait** – There must exist a set of processes [P1, P2, P3,...,Pn] such that P1 is waiting for P2, P2 for P3, etc.

- ◆ All of 1–4 necessary for deadlock to occur
- ◆ Two approaches to dealing with deadlock:
 - Pro-active: prevention
 - Reactive: detection + corrective action

Prevent by Eliminating One Condition

1. Mutual exclusion

- Buy more resources, split into pieces, or virtualize to make "infinite" copies
- Threads: threads have copy of registers = no lock

2. Hold and wait

- Wait on all resources at once (must know in advance)

3. No preemption

- Physical memory: virtualized with VM, can take physical page away and give to another process!

4. Circular wait

- Single lock for entire system: (problems?)
- Partial ordering of resources (next)





Outline

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





Chapter Objectives

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





System Model

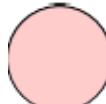
- System consists of resources
- Resource types R_1, R_2, \dots, R_m
 - *CPU cycles, memory space, I/O devices*
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
 - **request**
 - **use**
 - **release**

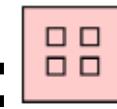


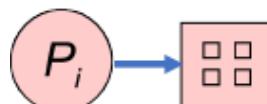
Resource Allocation Graph

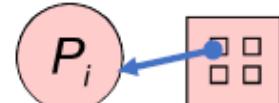
◆ View system as graph

- Processes and Resources are nodes
- Resource Requests and Assignments are edges

◆ Process: 

◆ Resource with 4 instances: 

◆ P_i requesting R_j : 

◆ P_i holding instance of R_j : 



Resource-Allocation Graph

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

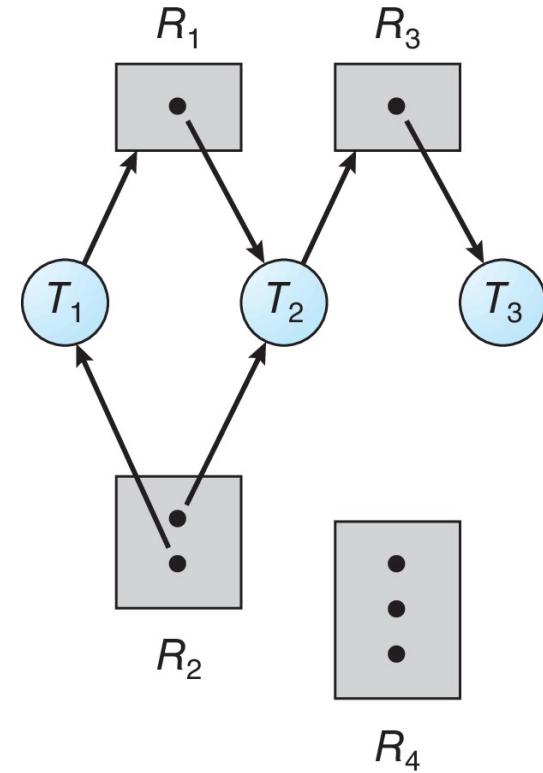
- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed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





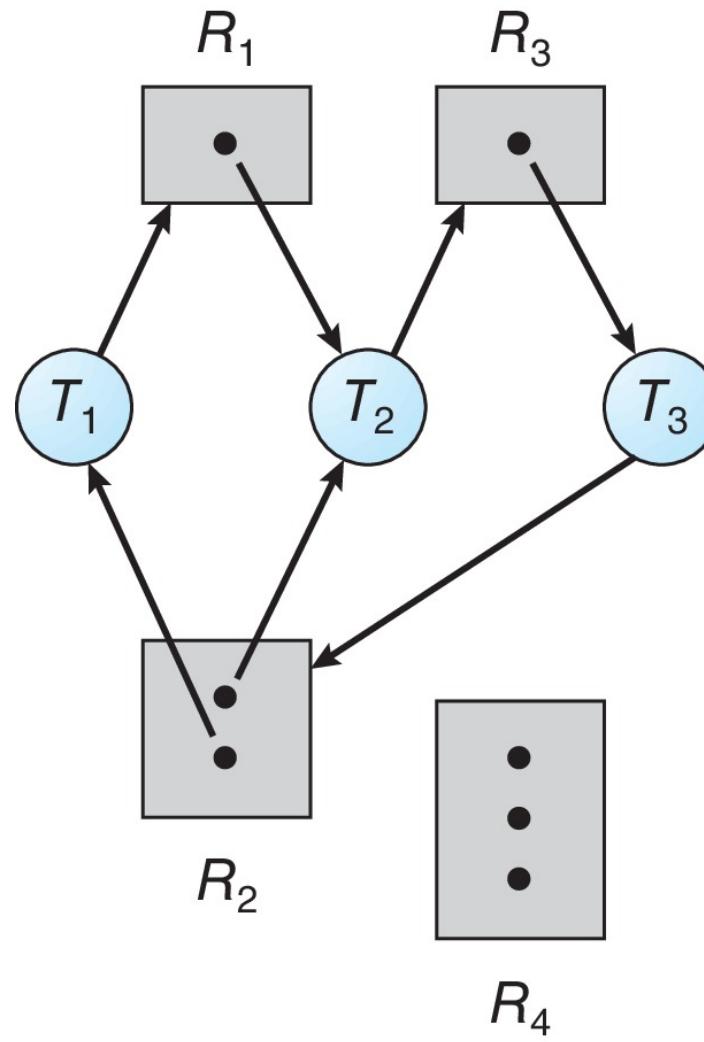
Resource Allocation Graph Example

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



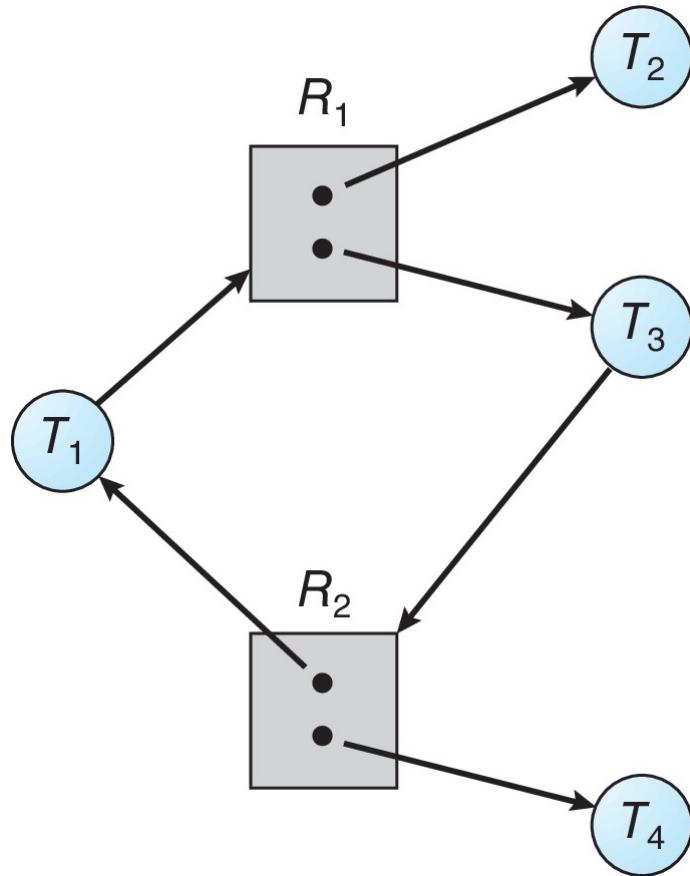


Resource Allocation Graph with a Deadlock





Graph with a Cycle But no Deadlock





Basic Facts

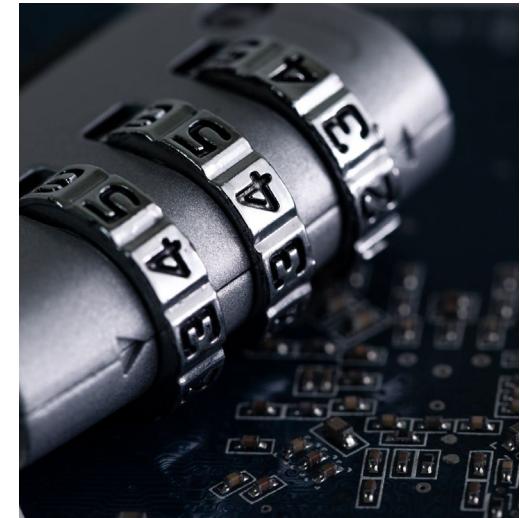
- **If graph has no cycles \Rightarrow no deadlock**
- **If graph contains a cycle**
 - Definitely deadlock if only one instance per resource ([waits-for graph \(WFG\)](#))
 - Otherwise, [maybe](#) deadlock, maybe not
- **Prevent deadlock with partial order on resources**
 - e.g., always acquire mutex m_1 before m_2
 - Usually design locking discipline for application this way



Dealing With Deadlock

There are four approaches for dealing with deadlock:

- ◆ Ignore it – how lucky do you feel?
- ◆ Prevention – make it impossible for deadlock to happen
- ◆ Avoidance – control allocation of resources
- ◆ Detection and Recovery – look for a cycle in dependencies



Ostrich Algorithm



- ◆ Pretend that deadlock never occurs
 - Occurs rarely
 - Difficult to detect
 - Cost of prevention is high



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 Avoidance

◆ **Avoidance**

- Provide information in advance about what resources will be needed by processes to guarantee that deadlock will not happen
- System only grants resource requests if it knows that the process can obtain all resources it needs in future requests
- Avoids circularities (wait dependencies)

◆ **Tough**

- Hard to determine all resources needed in advance
- Good theoretical problem, not as practical to use

Banker's Algorithm

◆ The Banker's Algorithm is the classic approach to deadlock avoidance for resources with multiple units

1. Assign a **credit limit** to each customer (process)

- Maximum credit claim must be stated in advance

2. Reject any request that leads to a **dangerous state**

- A dangerous state is one where a sudden request by any customer for the full credit limit could lead to deadlock
- A recursive reduction procedure recognizes dangerous states

3. In practice, the system must keep resource usage well below capacity to maintain a **resource surplus**

- Rarely used in practice due to low resource utilization

Detection and Recovery

◆ **Detection and recovery**

- If we don't have deadlock prevention or avoidance, then deadlock may occur
- In this case, we need to detect deadlock and recover from it

◆ **To do this, we need two algorithms**

- One to determine whether a deadlock has occurred
- Another to recover from the deadlock

◆ **Possible, but expensive (time consuming)**

Deadlock Detection

◆ **Detection**

- Traverse the resource graph looking for cycles
- If a cycle is found, preempt resource (force a process to release)

◆ **Expensive**

- Many processes and resources to traverse

◆ **Only invoke detection algorithm depending on**

- How often or likely deadlock is
- How many processes are likely to be affected when it occurs

Deadlock Recovery

◆ Once a deadlock is detected, we have two options

1. Abort processes

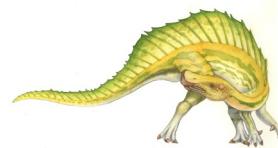
- Abort all deadlocked processes
 - ◆ Processes need to start over again
- Abort one process at a time until cycle is eliminated
 - ◆ System needs to rerun detection after each abort

2. Preempt resources (force their release)

- Need to select process and resource to preempt
- Need to rollback process to previous state
- Need to prevent starvation

Deadlock Summary

- ◆ **Deadlock occurs when processes are waiting on each other and cannot make progress**
 - Cycles in Resource Allocation Graph (RAG)
- ◆ **Deadlock requires four conditions**
 - Mutual exclusion, hold and wait, no resource preemption, circular wait
- ◆ **Four approaches to dealing with deadlock:**
 - Ignore it – Living life on the edge
 - Prevention – Make one of the four conditions impossible
 - Avoidance – Banker's Algorithm (control allocation)
 - Detection and Recovery – Look for a cycle, preempt or abort





Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

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





Deadlock Prevention (Cont.)

■ No Preemption:

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

■ Circular Wait:

- Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration



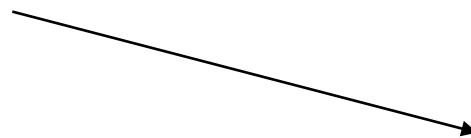


Circular Wait

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

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

code for **thread_two** could not be
written as follows:



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





Deadlock Avoidance

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

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





Safe State

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





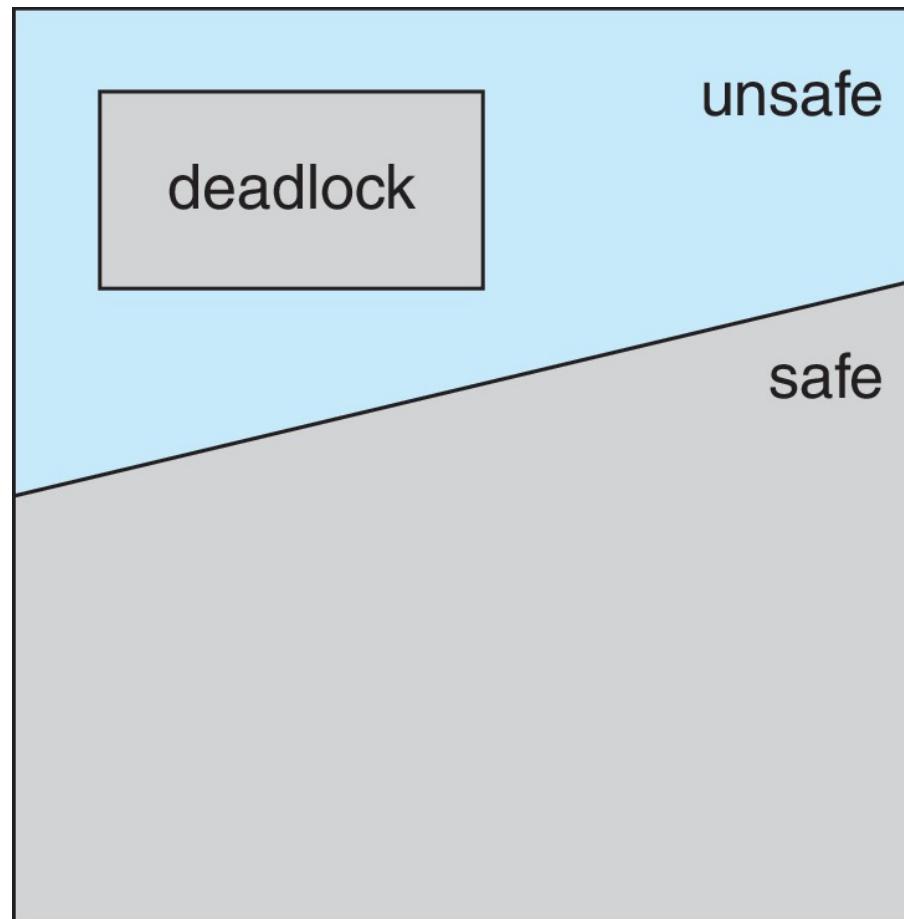
Basic Facts

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

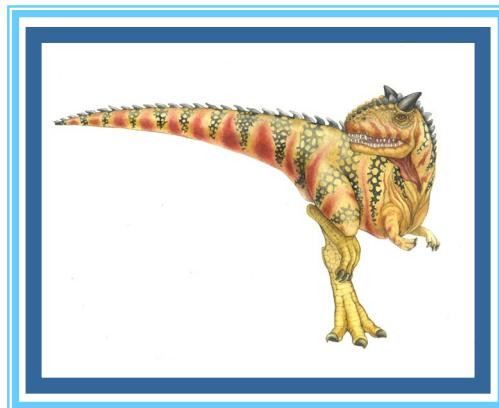




Safe, Unsafe, Deadlock State



ALGORITHMS





Avoidance Algorithms

- Single instance of a resource type
 - Use a resource-allocation graph

- Multiple instances of a resource type
 - Use the Banker's Algorithm





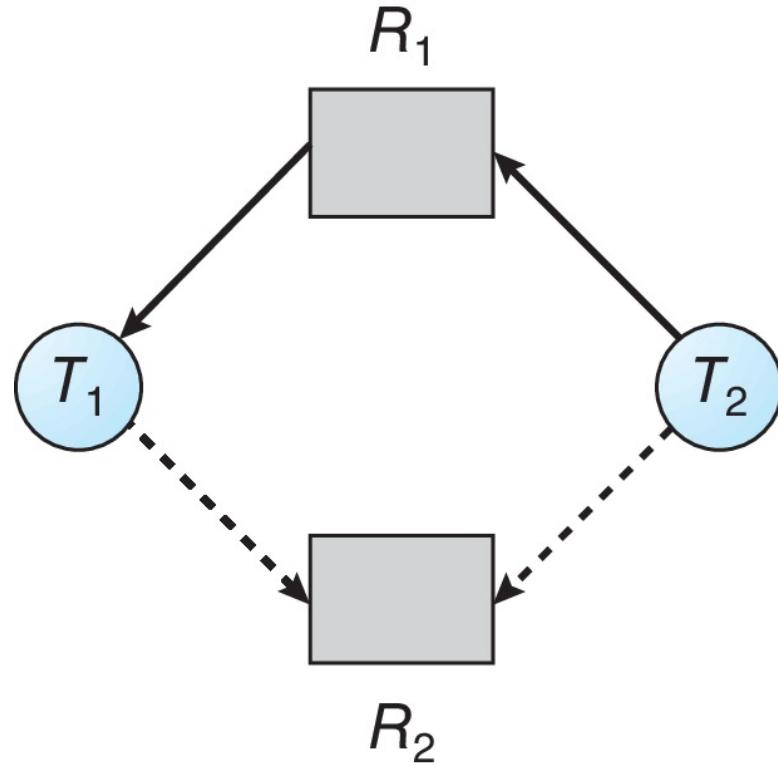
Resource-Allocation Graph Scheme

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



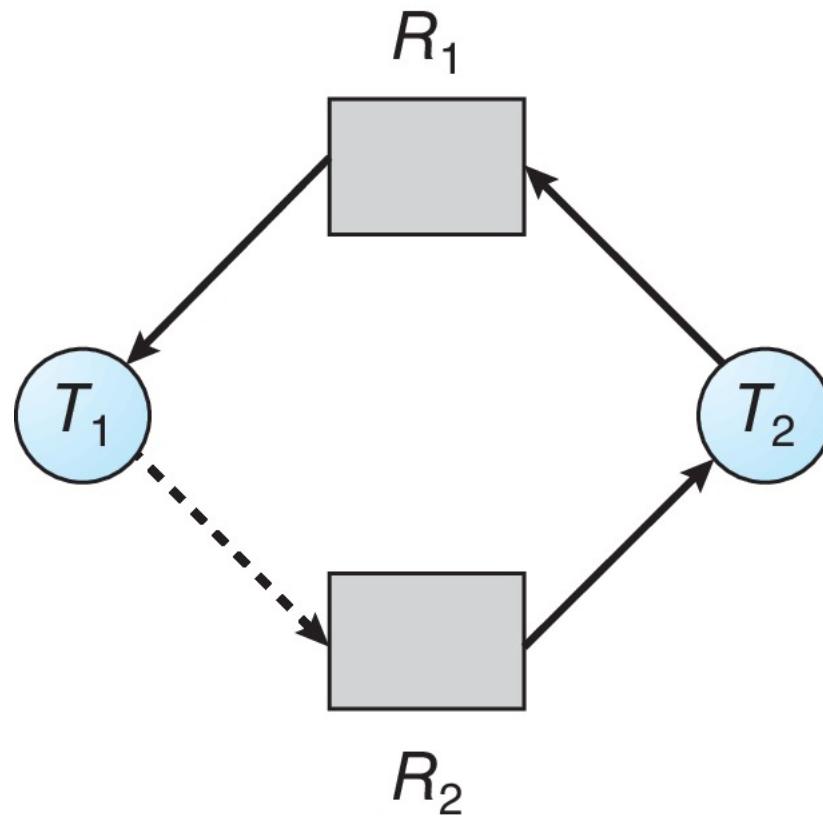


Resource-Allocation Graph





Unsafe State In Resource-Allocation Graph





Resource-Allocation Graph Algorithm

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

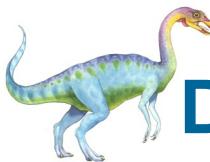




Banker's Algorithm

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





Data Structures for the Banker' s Algorithm

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

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

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$





Safety Algorithm

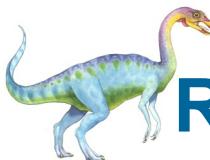
1. Let **Work** and **Finish** be vectors of length m and n , respectively.
Initialize:

Work = Available

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

2. Find an i such that both:
 - (a) **Finish [i] = false**
 - (b) **Need_i ≤ Work**If no such i exists, go to step 4
3. **Work = Work + Allocation_i**,
Finish[i] = true
go to step 2
4. If **Finish [i] == true** for all i , then the system is in a safe state





Resource-Request Algorithm for Process P_i

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 (5instances), 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>
P_0	0	1	0	7	5	3
P_1	2	0	0	3	2	2
P_2	3	0	2	9	0	2
P_3	2	1	1	2	2	2
P_4	0	0	2	4	3	3





Example (Cont.)

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

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

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





Example: P_1 Request (1,0,2)

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

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

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





Deadlock Detection

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





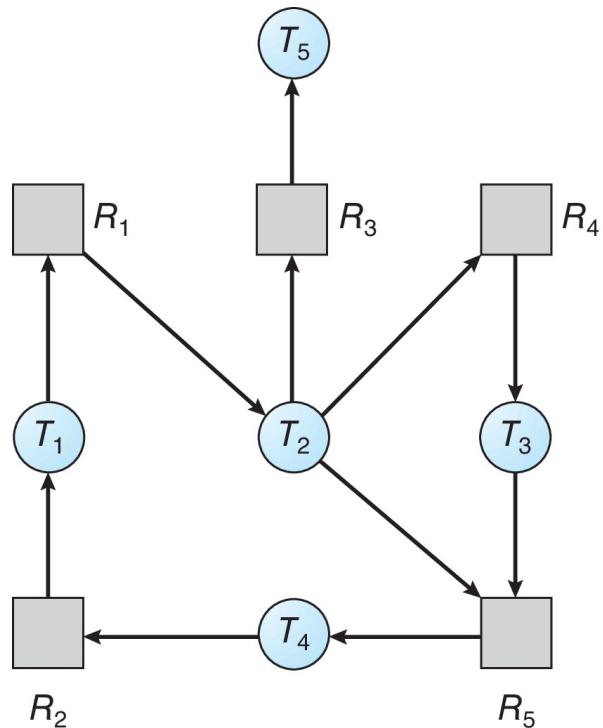
Single Instance of Each Resource Type

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

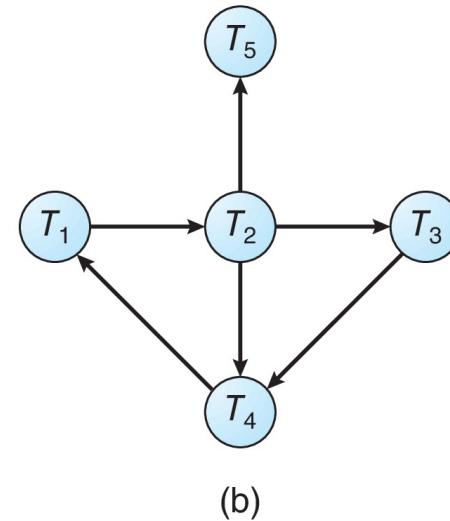




Resource-Allocation Graph and Wait-for Graph



(a)



(b)

Resource-Allocation Graph

Corresponding wait-for graph





Several Instances of a Resource Type

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





Detection Algorithm

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

If no such i exists, go to step 4





Detection Algorithm (Cont.)

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

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





Example of Detection Algorithm

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

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

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





Example (Cont.)

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

Request

	A	B	C
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

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





Detection-Algorithm Usage

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

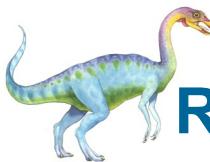




Recovery from Deadlock: Process Termination

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



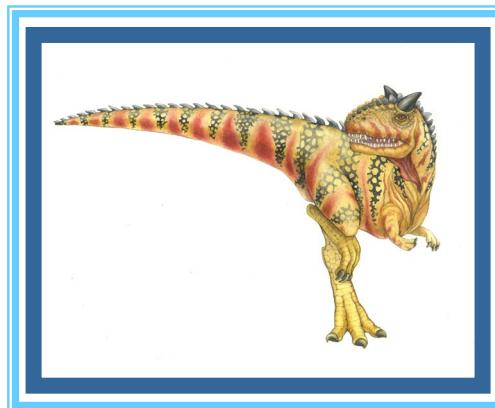


Recovery from Deadlock: Resource Preemption

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

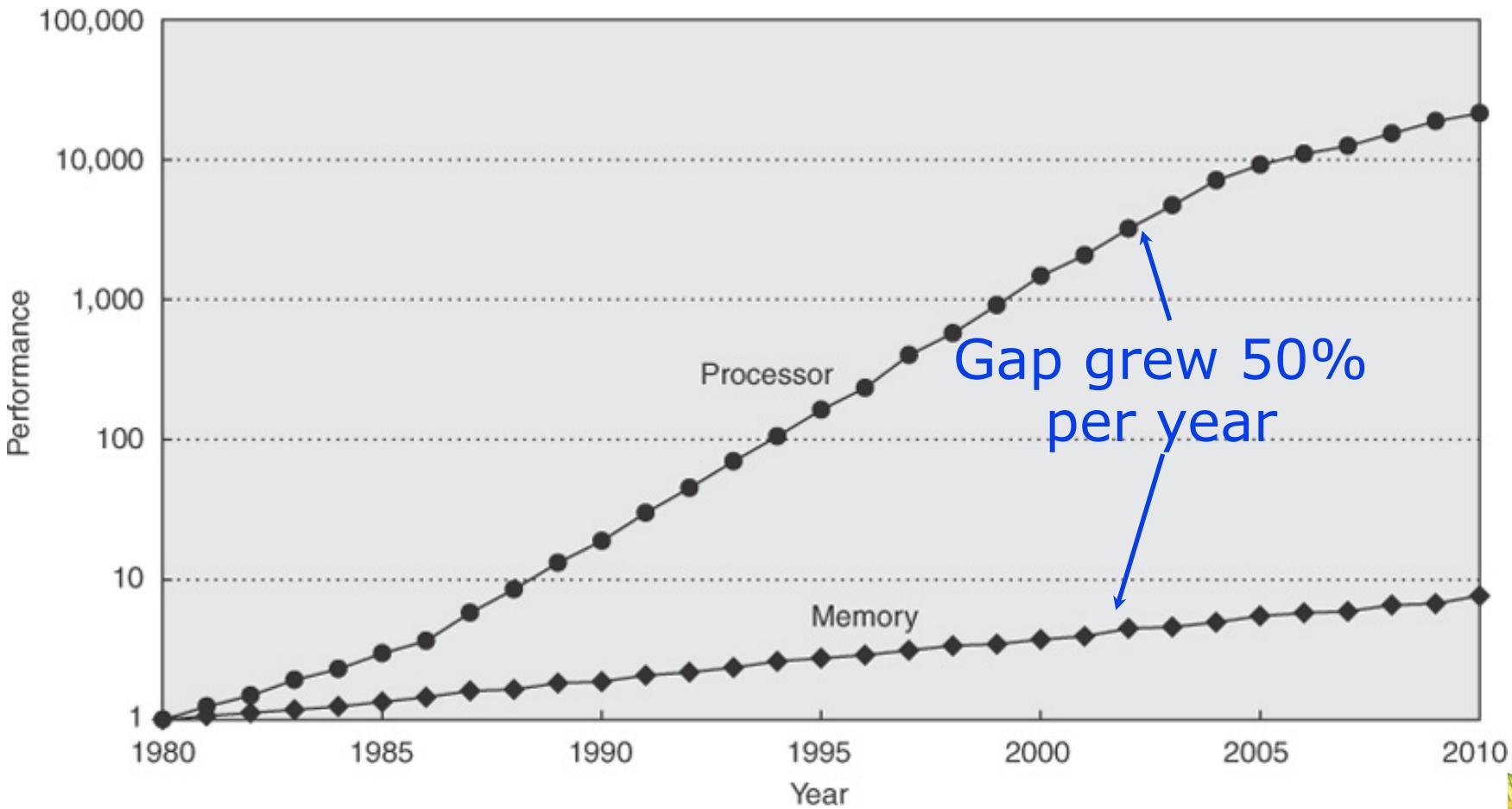


End of Chapter 8





Processor Memory Gap

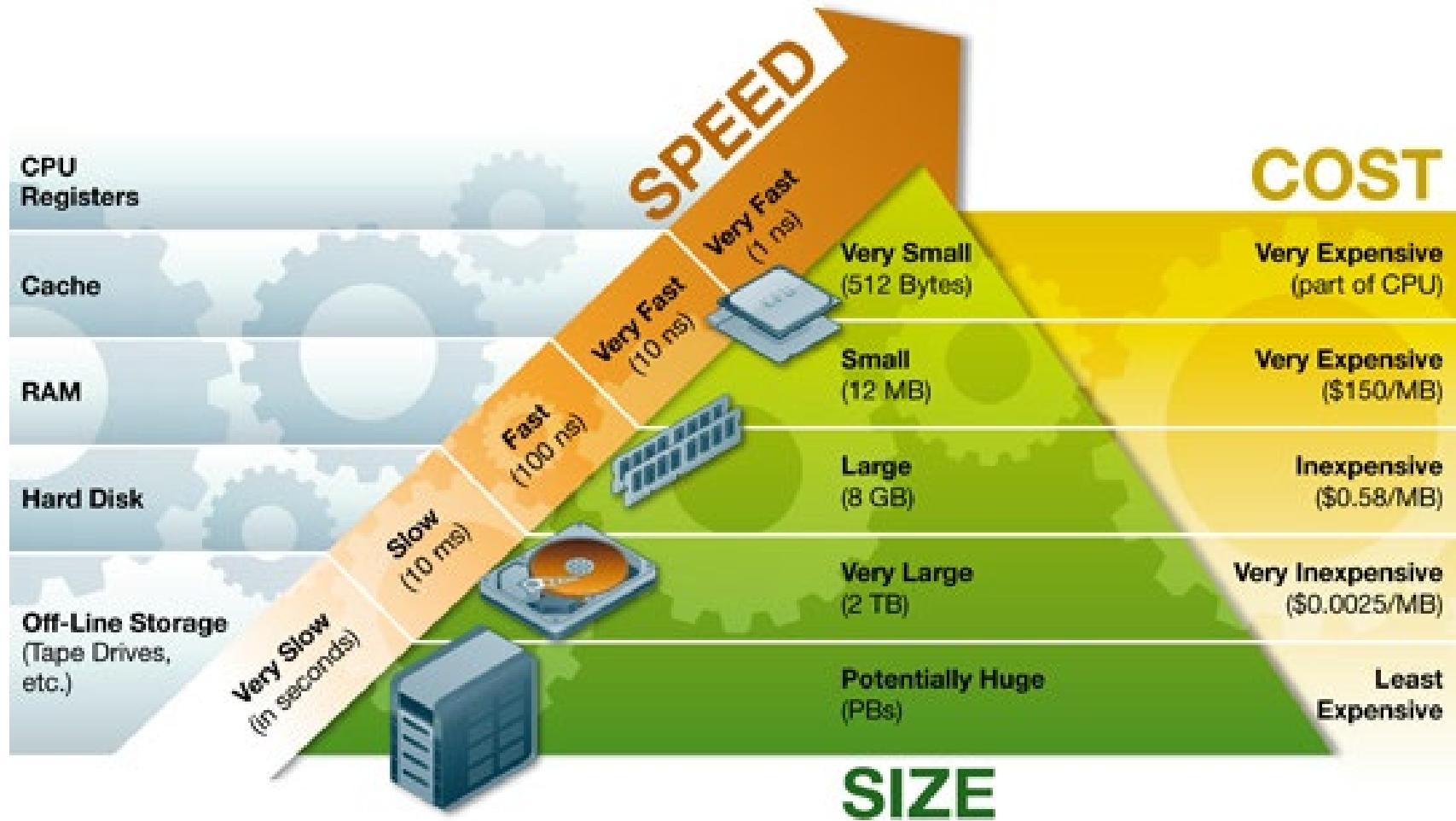


© 2007 Elsevier, Inc. All rights reserved.





Extended Memory Hierarchy



Source:

http://www.ts.avnet.com/uk/products_and_solutions/storage/hierarchy.html





Modern Memory Hierarchy

PROCESSOR

CPU

PROCESSOR
REGISTER
CPU CACHE

LEVEL 1 (L1) CACHE
LEVEL 2 (L2) CACHE
LEVEL 3 (L3) CACHE

SUPER FAST
SUPER EXPENSIVE
TINY CAPACITY

SD-RAM,
DDR-SDRAM, ...

PHYSICAL MEMORY

RANDOM ACCESS
MEMORY (RAM)

FASTER
EXPENSIVE
SMALL CAPACITY

SOLID STATE
DRIVES

SOLID STATE MEMORY

MECHANICAL
HARD DRIVES

NON-VOLATILE FLASH-BASED MEMORY

FAST
PRICED REASONABLY
AVERAGE CAPACITY

VIRTUAL MEMORY

FILE-BASED MEMORY

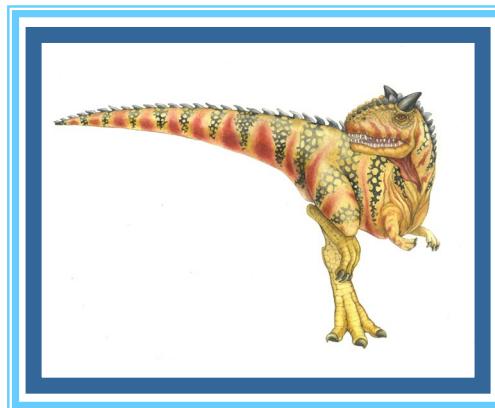
AVERAGE SPEED
PRICED REASONABLY
AVERAGE CAPACITY

SLOW
CHEAP
LARGE CAPACITY

Source:

<http://blog.teachbook.com.au/index.php/2012/02/memory-hierarchy/>

Chapter 9: Main Memory





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
- Memory unit only sees a stream of:
 - addresses + read requests, or
 - address + data and write requests
- Register access is done in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation





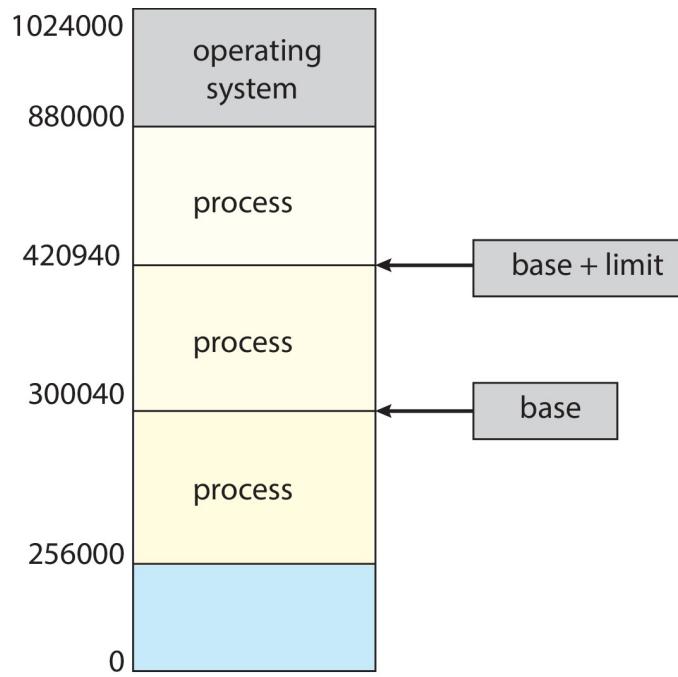
-
- Just as processes share the CPU, they also share physical memory. This chapter is about mechanisms for doing that sharing.





Protection

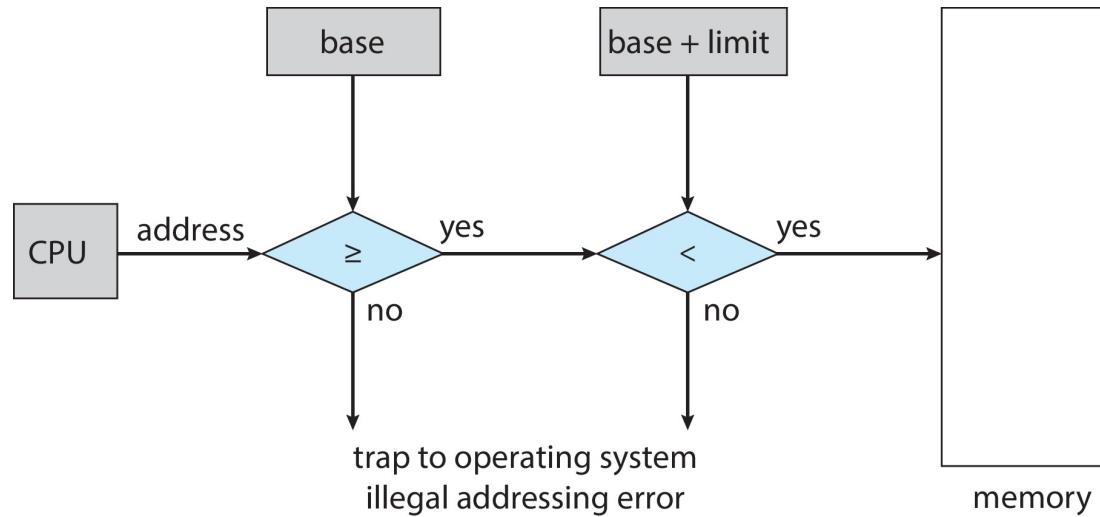
- Need to ensure that a process can access only those addresses in its address space.
- We can provide this protection by using a pair of **base** and **limit registers** to define the logical address space of a process





Hardware Address Protection

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



- the instructions to loading the base and limit registers are privileged





Address Binding

- Programs on disk, ready to be brought into memory to execute from an **input queue**
 - Without support, must be loaded into address 0000
- Inconvenient to have first user 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
 - ▶ i.e. “14 bytes from beginning of this module”
 - Linker or loader will bind relocatable addresses to absolute addresses
 - ▶ i.e. 74014
 - Each binding maps one address space to another

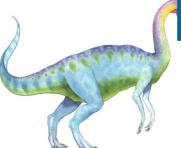




Binding of Instructions and Data to Memory

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





Multistep Processing of a User Program

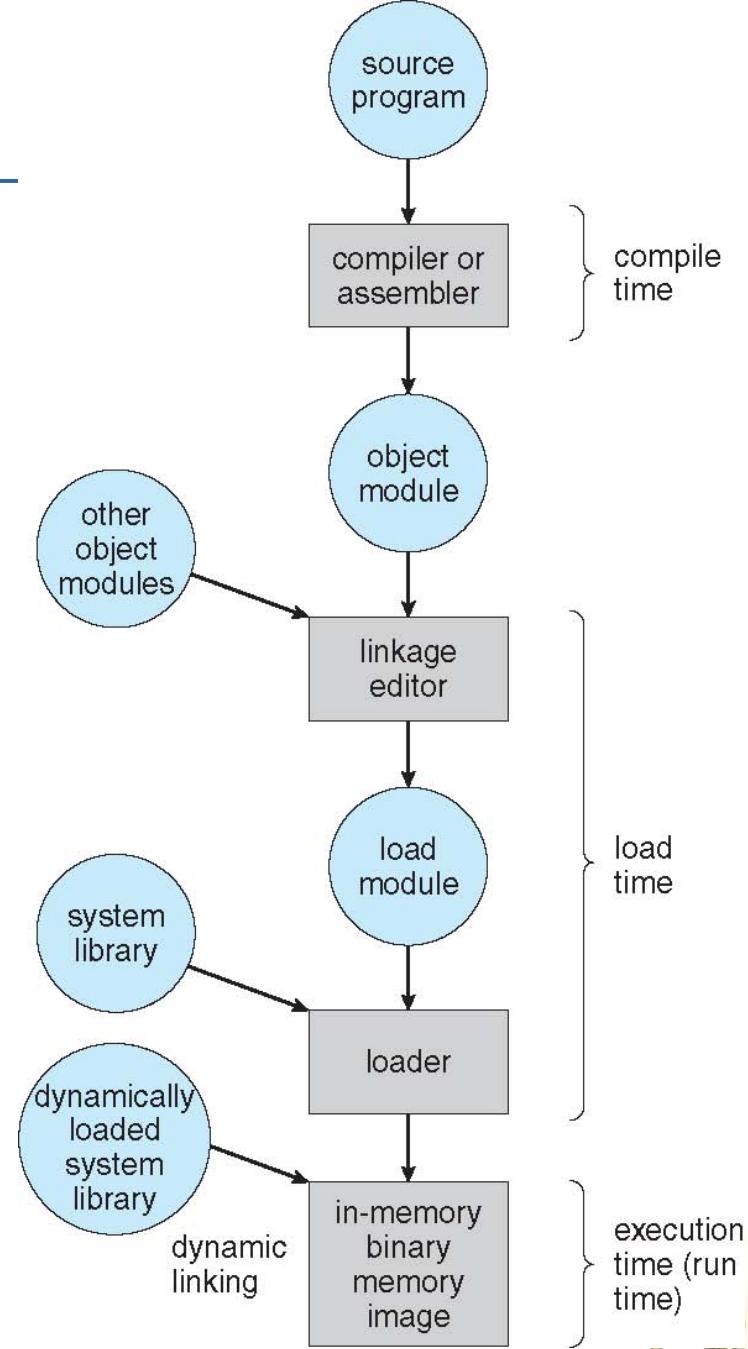
Relocatable Means that the program image can reside anywhere in physical memory.

Binding Programs need real memory in which to reside. When is the location of that real memory determined?

Compiler: If it's known where the program will reside, then absolute code is generated. Otherwise, compiler produces relocatable code.

Load: Binds relocatable to physical. Can find best physical location.

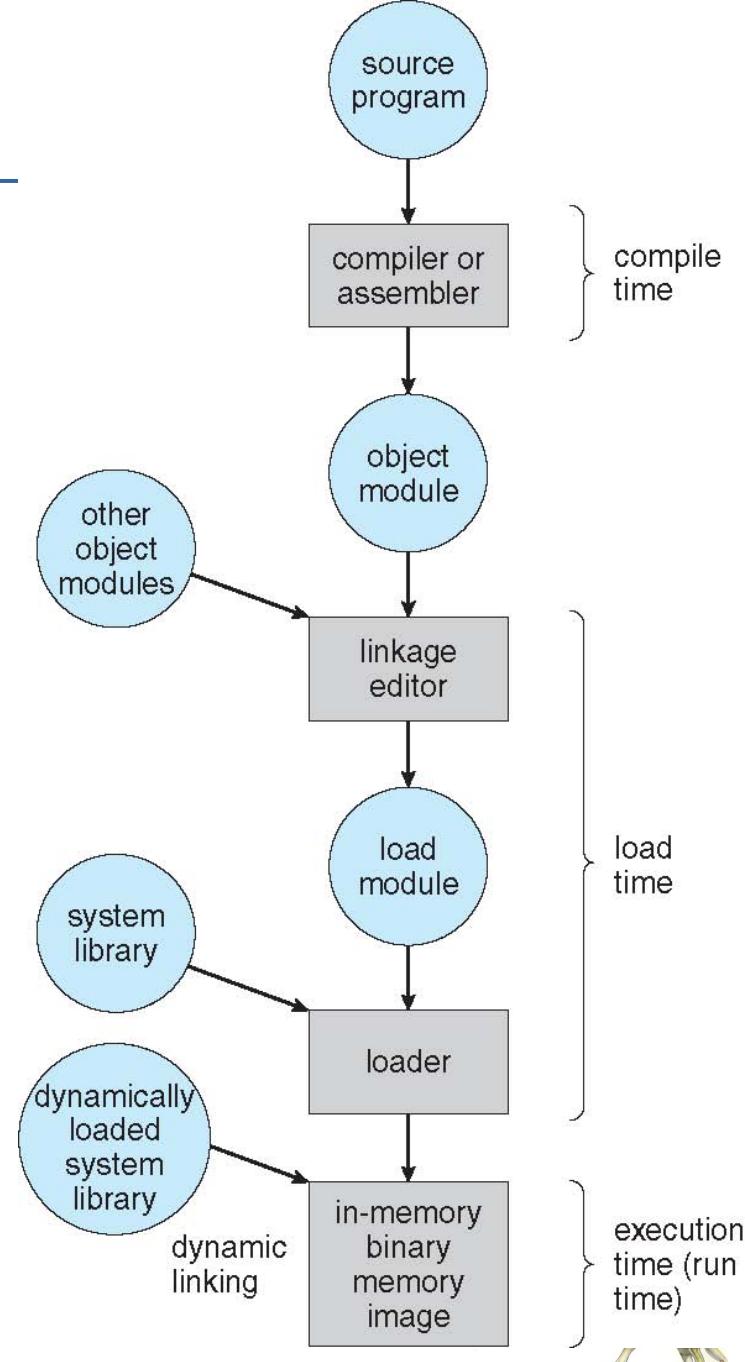
Execution: The code can be moved around during execution. Means flexible virtual mapping.

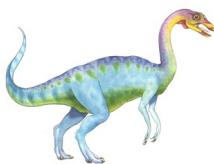




Dynamic loading

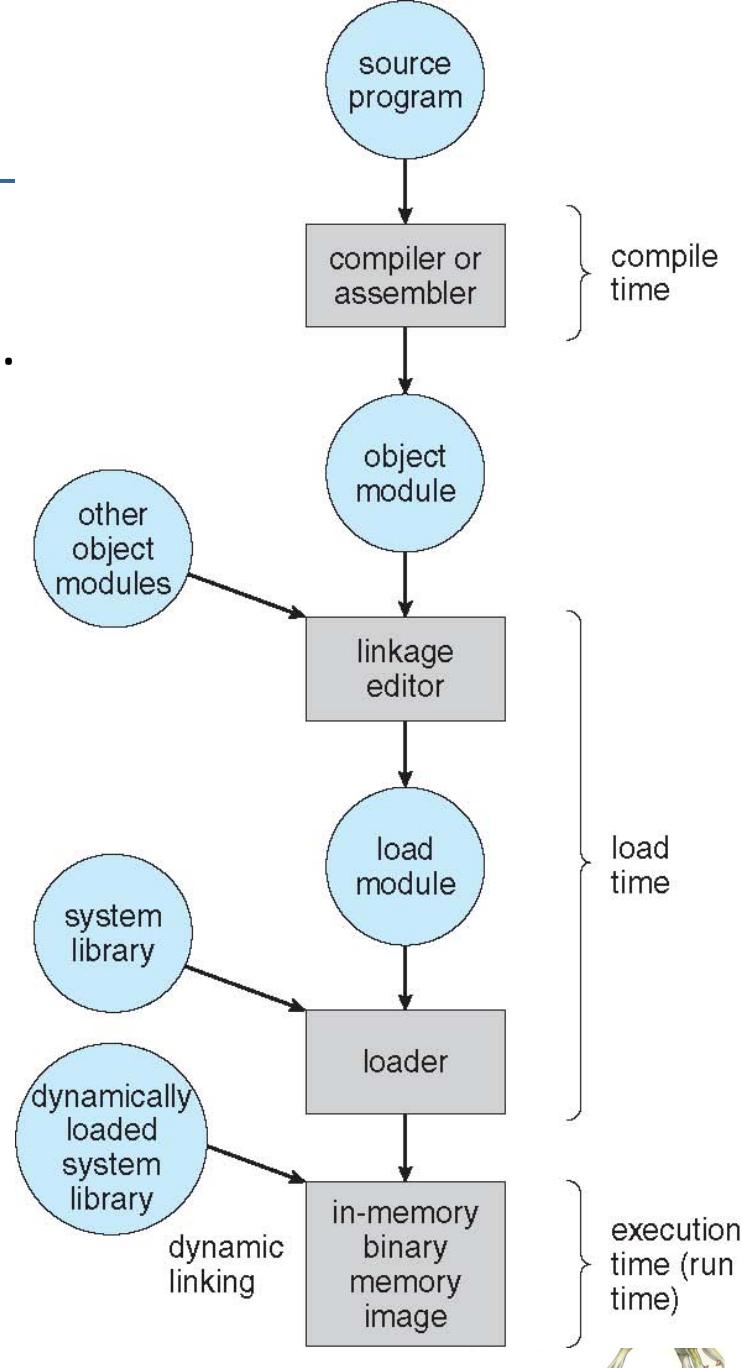
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded.
- Useful when large amounts of code are needed to handle infrequently occurring cases.
- No special support from the OS is required - implemented through program design.





Dynamic Linking

- Linking 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 needed to check if routine is in processes' memory address.
- Dynamic linking is particularly useful for libraries.





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





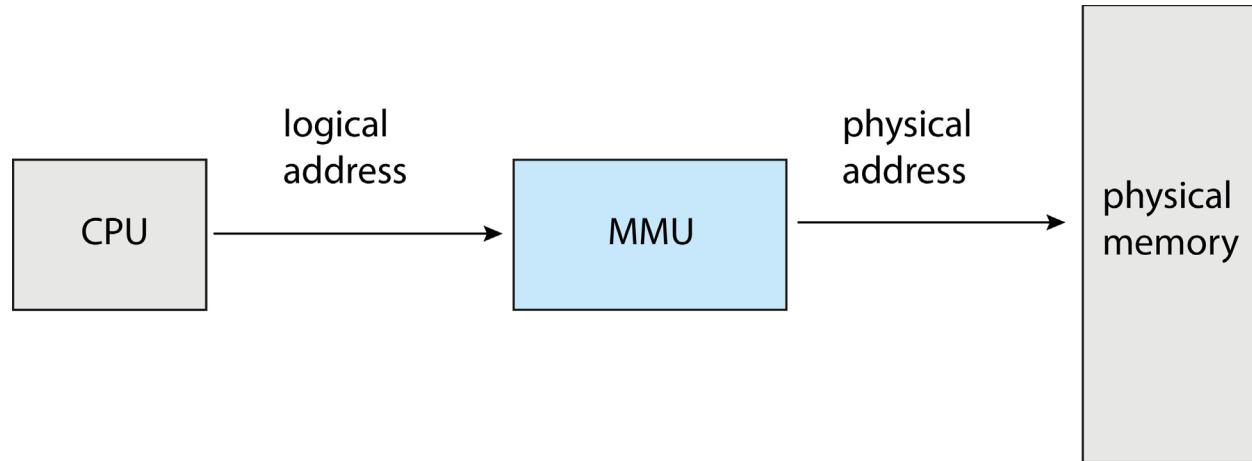
-
- Name two differences between logical and physical addresses.
 - A logical address does not refer to an actual existing address; rather, it refers to an abstract address in an abstract address space. Contrast this with a physical address that refers to an actual physical address in memory. A logical address is generated by the CPU and is translated into a physical address by the memory management unit(MMU). Therefore, physical addresses are generated by the MMU.





Memory-Management Unit (MMU)

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



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





Memory-Management Unit (Cont.)

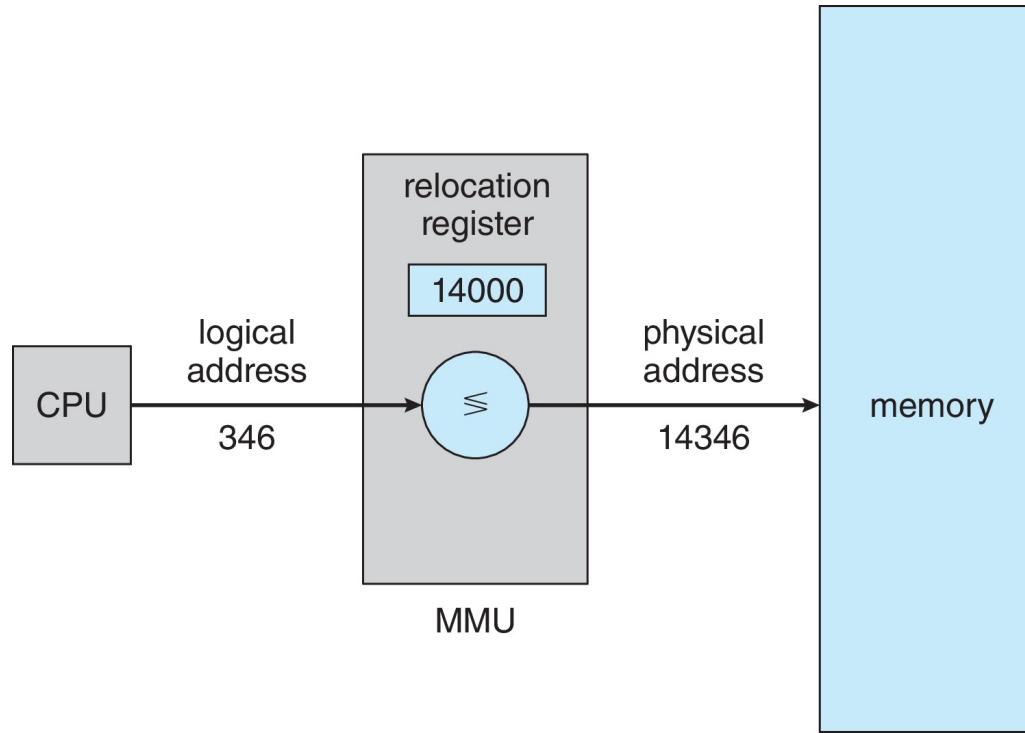
- Consider simple scheme, which is a generalization of the base-register scheme.
- The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical*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





Memory-Management Unit (Cont.)

- 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





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 into two **partitions**:
 - Resident operating system, usually held in low memory with interrupt vector
 - User processes then held in high memory
 - Each process contained in single contiguous section of memory





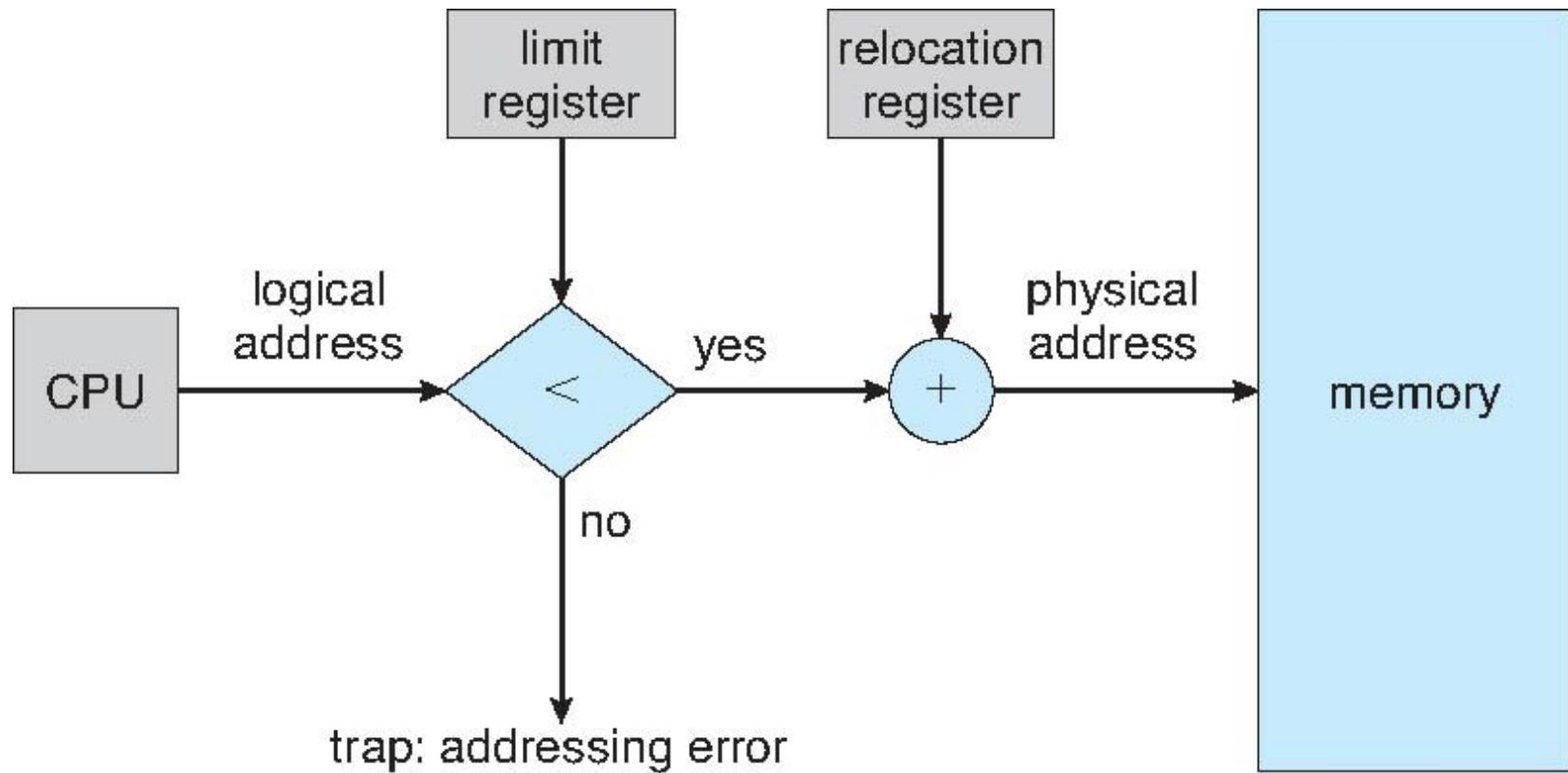
Contiguous Allocation (Cont.)

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
 - Base register contains value of smallest physical address
 - Limit register contains range of logical addresses – each logical address must be less than the limit register
 - MMU maps logical address *dynamically*
 - Can then allow actions such as kernel code being **transient** and kernel changing size





Hardware Support for Relocation and Limit Registers

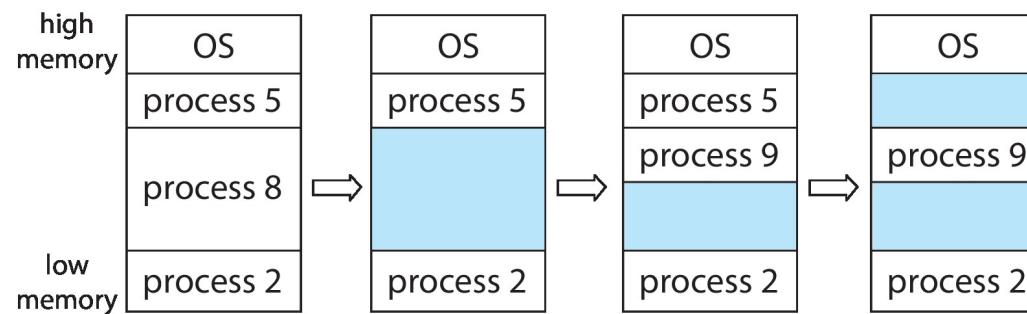




Variable Partition

□ 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 combined
- Operating system maintains information about:
 - allocated partitions
 - free partitions (hole)





Dynamic Storage-Allocation Problem

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

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

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





Fragmentation

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





Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
 - Shuffle memory contents to place all free memory together in one large block
 - Compaction is possible *only* if relocation is dynamic, and is done at execution time
 - 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





Can we avoid external fragmentation? How?

Answer in the chat





Paging

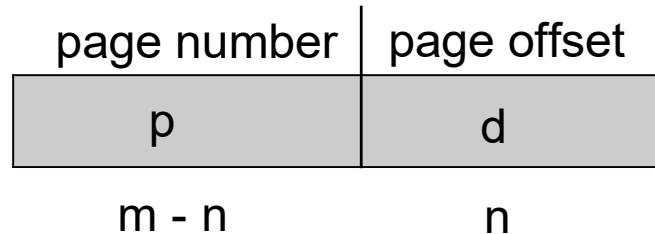
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
 - Avoids external fragmentation
 - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
 - Size is power of 2, 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 program of size **N** pages, need to find **N** free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation





Address Translation Scheme

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



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





- Why are page sizes always powers of 2?
- Paging is implemented by breaking up an address into a page and offset number. It is most efficient to break the address into X page bits and Y offset bits, rather than perform arithmetic on the address to calculate the page number and offset. Because each bit position represents a power of 2, splitting an address between bits results in a page size that is a power of 2.



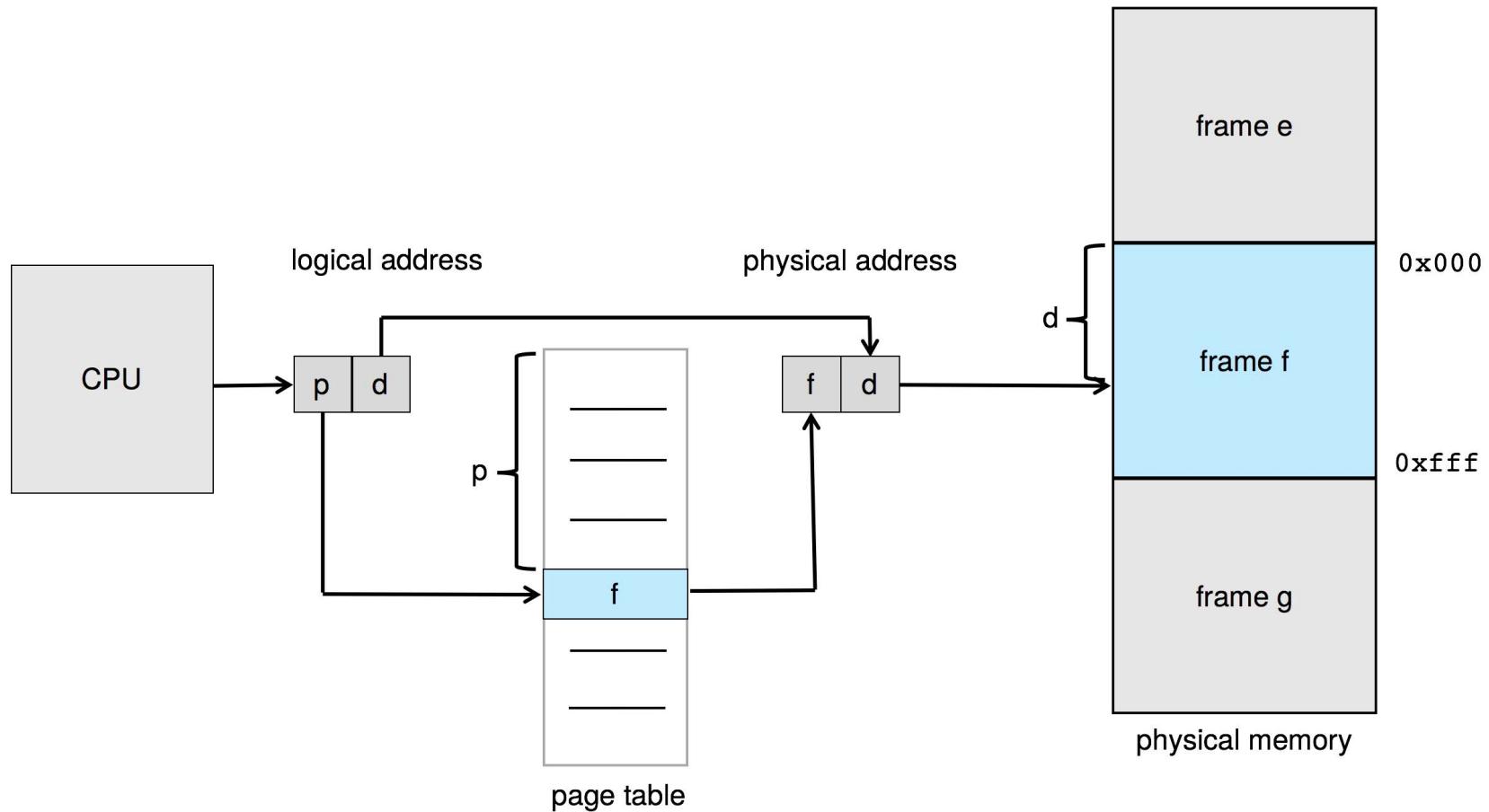


- Consider a logical address space of eight pages of 1024 words each, mapped onto a physical memory of 32 frames.
 - How many bits are there in the logical address?
 - How many bits are there in the physical address?
 - Logical address: 13 bits
 - Physical address: 15 bits



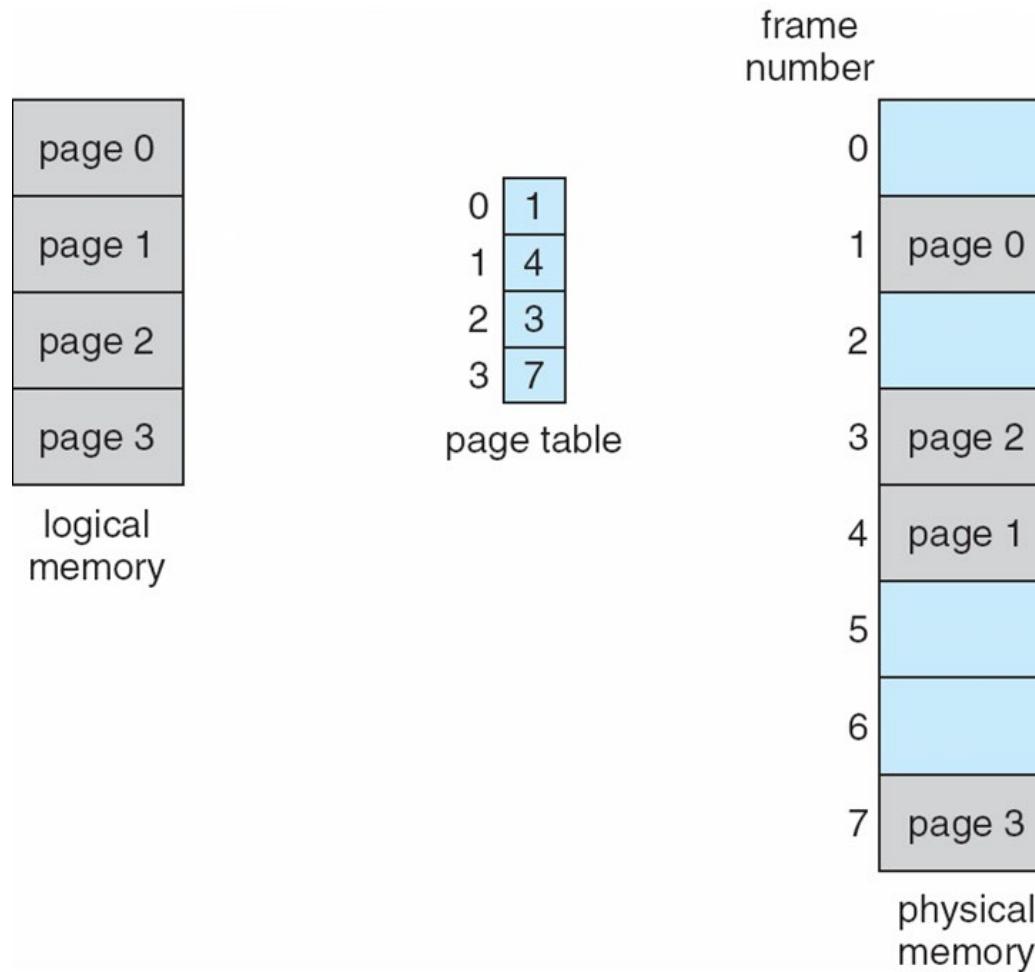


Paging Hardware





Paging Model of Logical and Physical Memory





Paging Example

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

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

logical memory

0	5
1	6
2	1
3	2

page table

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

physical memory

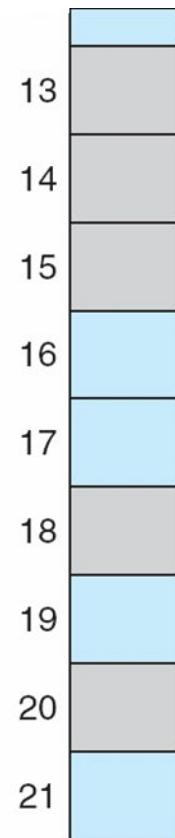
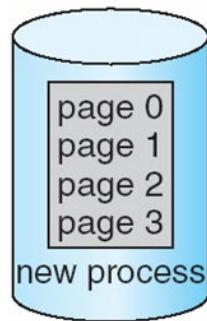




Free Frames

free-frame list

14
13
18
20
15

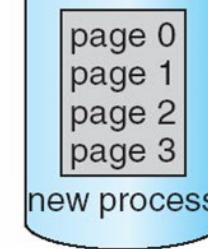


(a)

Before allocation

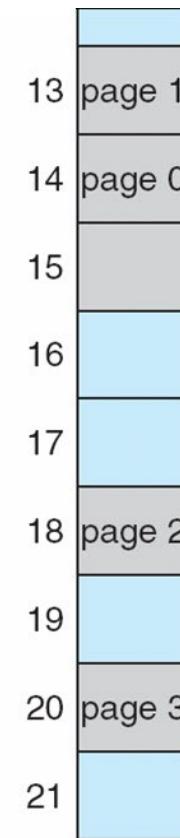
free-frame list

15



0	14
1	13
2	18
3	20

new-process page table



(b)

After allocation





Implementation of Page Table

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





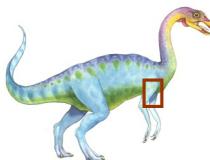
- Consider a simple paging system with the following parameters: 2^{32} bytes of physical memory; page size of 2^{10} bytes; 2^{16} pages of logical address space.
 - a. How many bits are in a logical address?
 - b. How many bytes in a frame?
 - c. How many bits in the physical address specify the frame?
 - d. How many entries in the page table?
 - e. How many bits in each page table entry? Assume each page table entry contains a valid/invalid bit.





-
- a. The number of bytes in the logical address space is $(2^{16} \text{ pages}) \times (2^{10} \text{ bytes/page}) = 2^{26} \text{ bytes}$. Therefore, 26 bits are required for the logical address.
 - b. A frame is the same size as a page, 2^{10} bytes.
 - c. The number of frames in main memory is $(2^{32} \text{ bytes of main memory}) / (2^{10} \text{ bytes/frame}) = 2^{22}$ frames. So 22 bits are needed to specify the frame.
 - d. There is one entry for each page in the logical address space. Therefore there are 2^{16} entries.
 - e. In addition to the valid/invalid bit, 22 bits are needed to specify the frame location in main memory, for a total of 23 bits.





- A system implements a paged virtual address space for each process using a one-level page table. The maximum size of an address space is 16 megabytes. The page table for the running process includes the following entries:

page	frame number
0:	4
1:	8
2:	16
3:	17
4:	9

The page size is 1024 bytes and the maximum physical memory size of the machine is 2 megabytes.

1. How many bits are required for each page table entry?
2. What is the maximum number of entries in a page table?
3. How many bits are there in a virtual address?
4. To which physical address will the virtual address 1524 translate to?
5. Which virtual address will translate to physical address 10020?





-
1. There can be at most $2^{21}/2^{10}$ frames in main memory, so a page table entry will require 11 bits for a frame number, plus protection and reference bits, etc.
 2. A page table may have up to $2^{24}/2^{10} = 2^{14}$ entries.
 3. A virtual address has 24 bits.
 4. Virtual address 1524 lies on page 1 which resides in frame 8 with an offset of 1524 modulo 1024, or 500. So, 1524 maps to physical address $8*1024+500 = 8192+500 = 8692$.
 5. Physical address 10020 is in frame 9, with an offset of 804. Virtual page 4 maps to frame 9, so virtual address $4*1024+804 = 4900$ will map to 10020.





Translation Look-Aside Buffer

- 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





Hardware

- Associative memory – parallel search

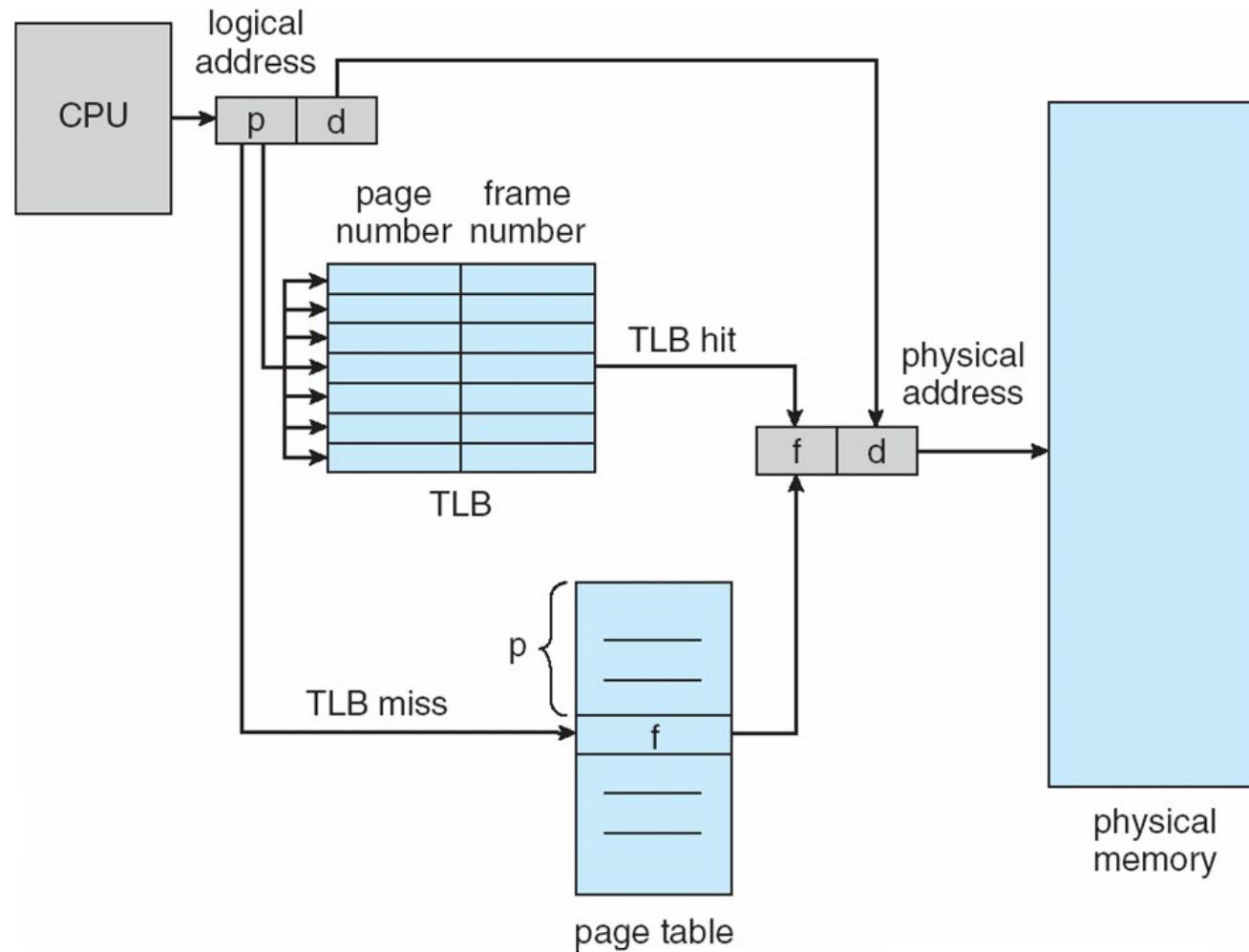
Page #	Frame #

- 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





- What is a *translation look-aside buffer (TLB)*? Do you need a TLB for correct execution of programs?
- TLB is a cache that stores **[PageNo] [FrameNo]** pairs.
You don't need it for correct execution of programs.





Memory Protection

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





Effective Access Time

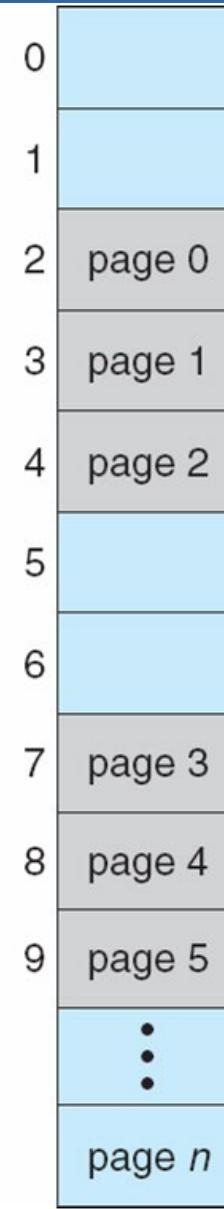
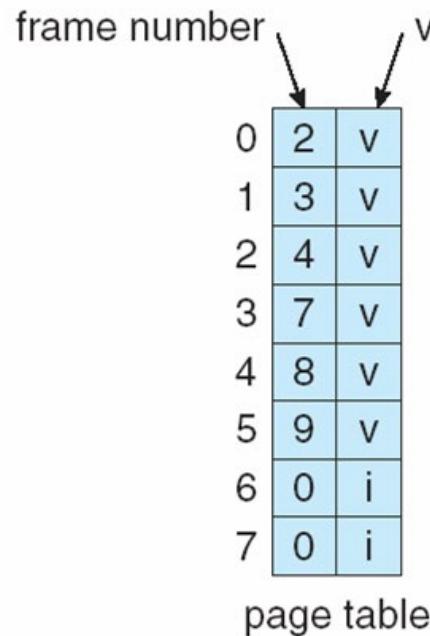
- Hit ratio – percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
 - If we find the desired page in TLB then a mapped-memory access take 10 ns
 - Otherwise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





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	





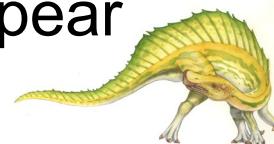
Shared Pages

□ 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 interprocess communication if sharing of read-write pages is allowed

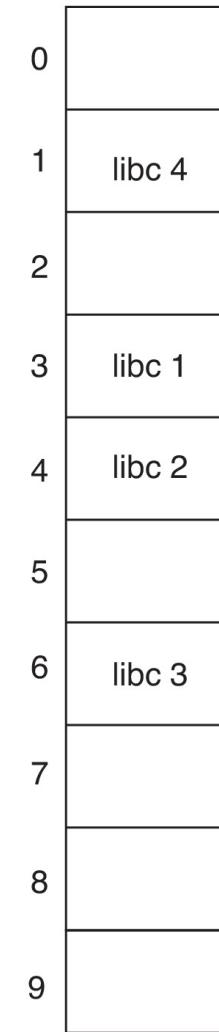
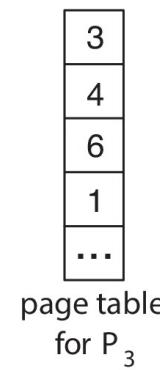
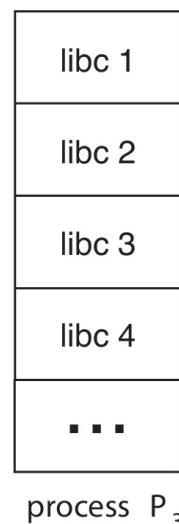
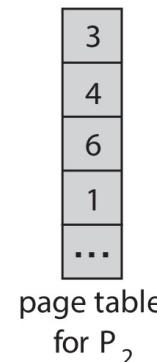
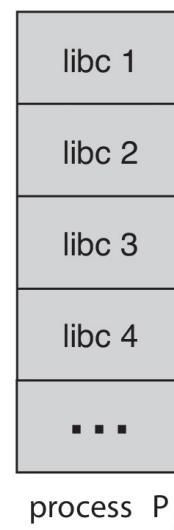
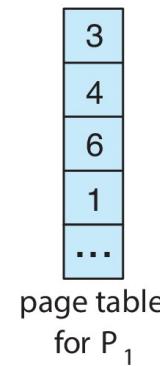
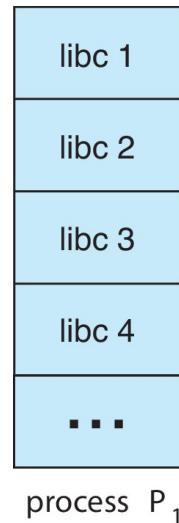
□ Private code and data

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





Shared Pages Example



physical memory





Structure of the Page Table

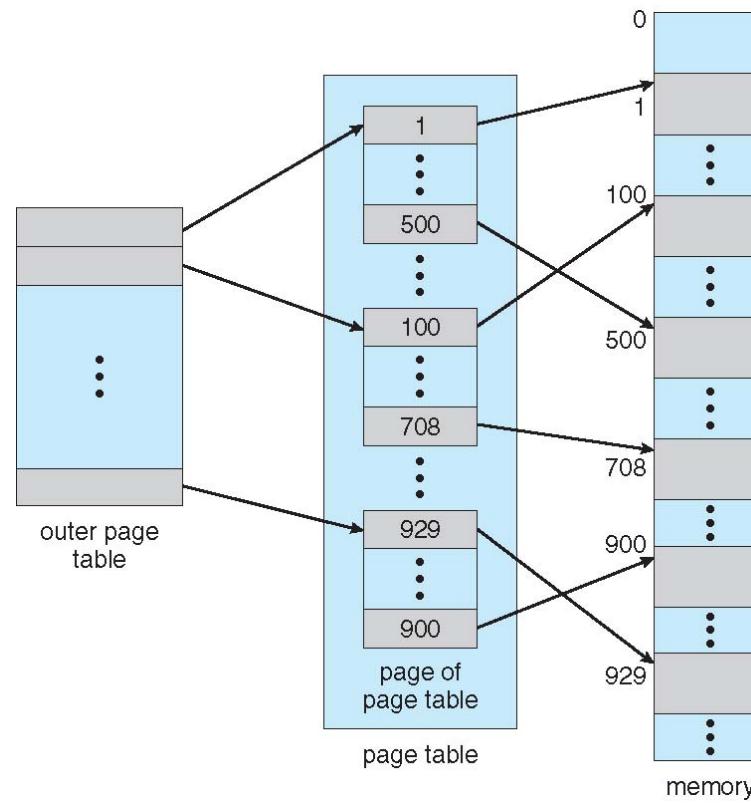
- Memory structures for paging can get huge using straight-forward methods
 - Consider a 32-bit logical address space
 - Page size of 4 KB (2^{12})
 - Page table would have 1 million entries ($2^{32} / 2^{12}$)
 - If each entry is 4 bytes → each process 4 MB of physical address space for the page table 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

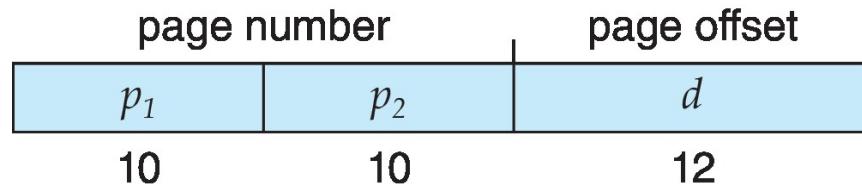
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table





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:

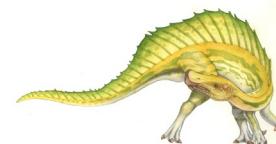
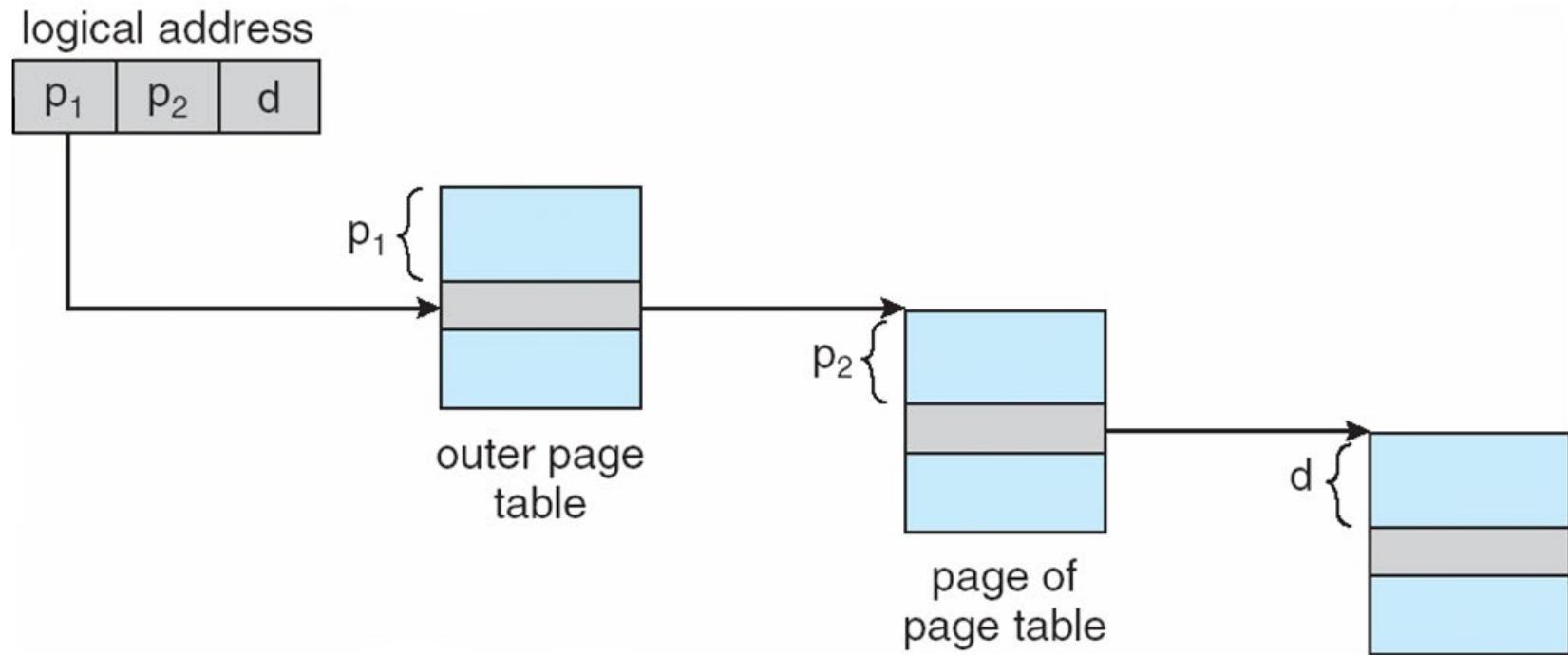


- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**





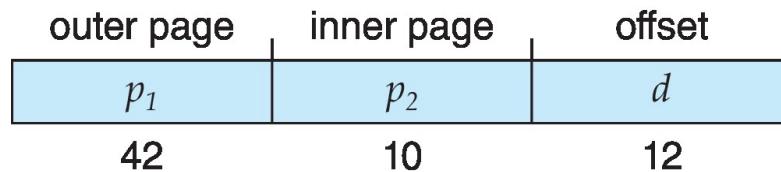
Address-Translation Scheme





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

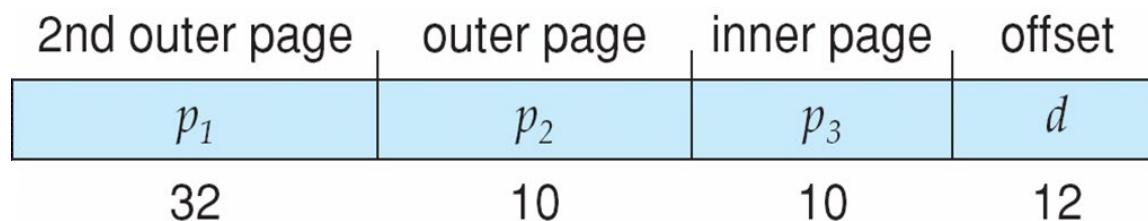
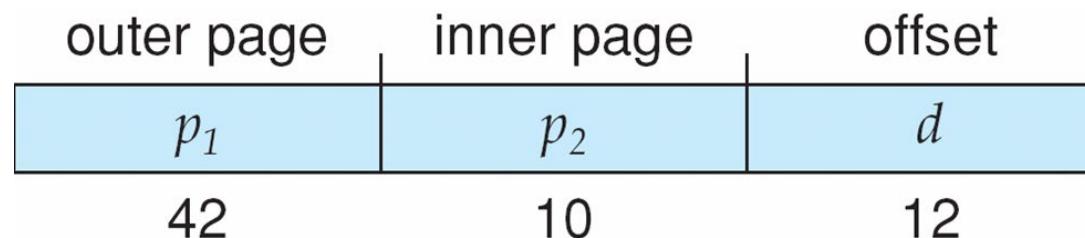


- Outer page table has 2^{42} entries or 2^{44} bytes
- One solution is to add a 2nd outer page table
 - But in the following example the 2nd outer page table is still 2^{34} bytes in size
 - ▶ And possibly 4 memory access to get to one physical memory location





Three-level Paging Scheme



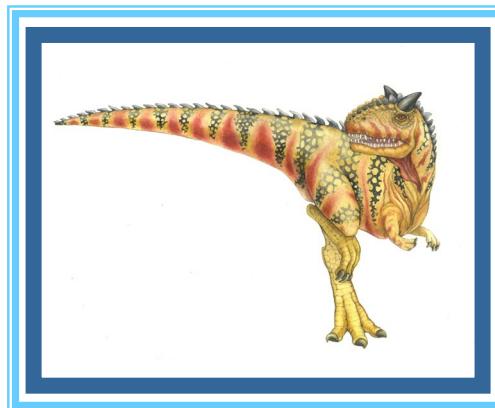


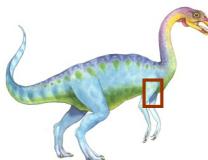
- A system implements a **paged** virtual memory for each process using **three levels of page tables**. The **MMU** includes a **TLB**. The time to access main memory is T . The TLB can be accessed very quickly (in essentially zero time). The running process needs to read the contents of virtual address v . What must the hit ratio of the TLB be if we wish this operation to take time $2T$, on average?
- Reading virtual address v will require four memory accesses in case of a TLB miss, one for the segment table, and one for each level of page table, and one for the actual physical memory location corresponding to v . In case of a TLB hit, only one memory access (for the physical location itself) will be required. So, if h is the hit ratio of the TLB, we need $4T(1 - h) + T(h) = 2T$, i.e., $h = 0.67$.



Chapter 9: Main Memory

Part 2 -10th Nov 2021





- A system implements a paged virtual address space for each process using a one-level page table. The maximum size of an address space is 16 megabytes. The page table for the running process includes the following entries:

page	frame number
0:	4
1:	8
2:	16
3:	17
4:	9

The page size is 1024 bytes and the maximum physical memory size of the machine is 2 megabytes.

1. How many bits are required for each page table entry?
2. What is the maximum number of entries in a page table?
3. How many bits are there in a virtual address?
4. To which physical address will the virtual address 1524 translate to?
5. Which virtual address will translate to physical address 10020?



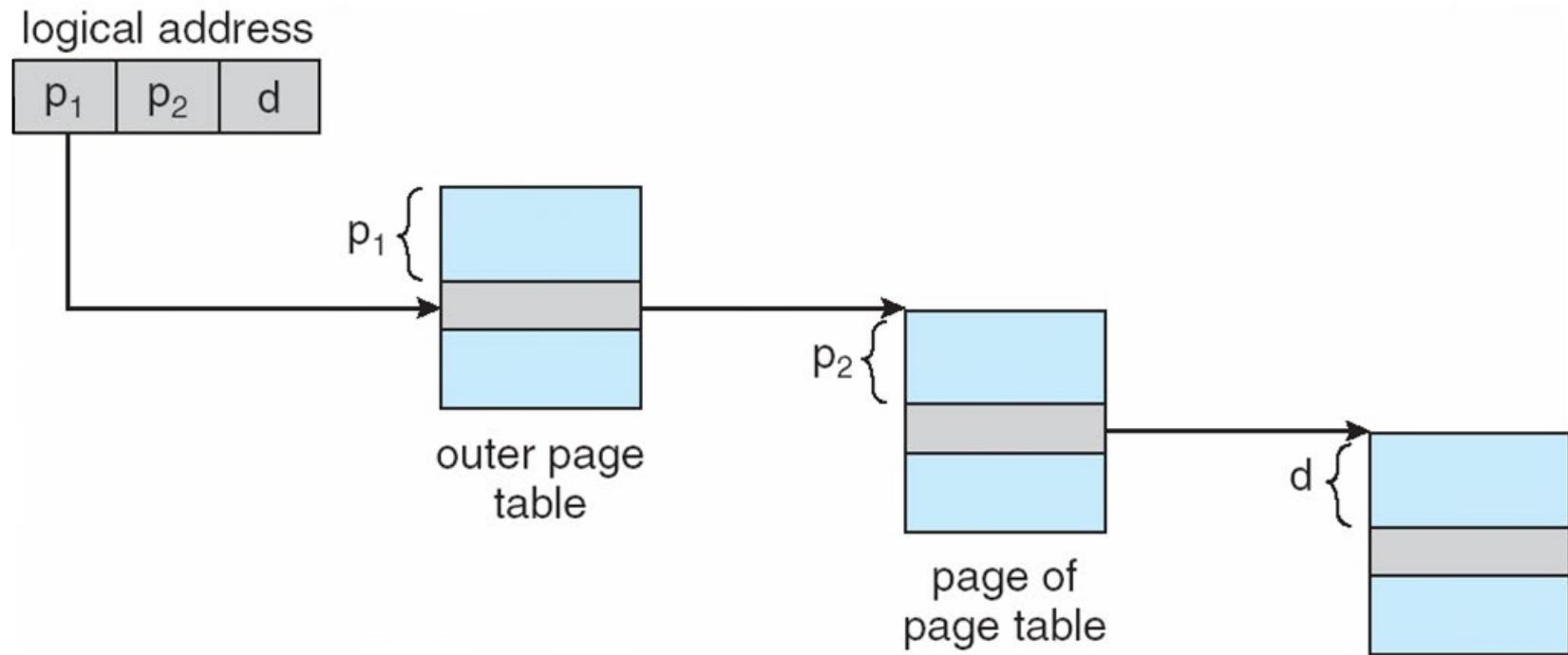


-
1. There can be at most $2^{21}/2^{10}$ frames in main memory, so a page table entry will require 11 bits for a frame number, plus protection and reference bits, etc.
 2. A page table may have up to $2^{24}/2^{10} = 2^{14}$ entries.
 3. A virtual address has 24 bits.
 4. Virtual address 1524 lies on page 1 which resides in frame 8 with an offset of 1524 modulo 1024, or 500. So, 1524 maps to physical address $8*1024+500 = 8192+500 = 8692$.
 5. Physical address 10020 is in frame 9, with an offset of 804. Virtual page 4 maps to frame 9, so virtual address $4*1024+804 = 4900$ will map to 10020.





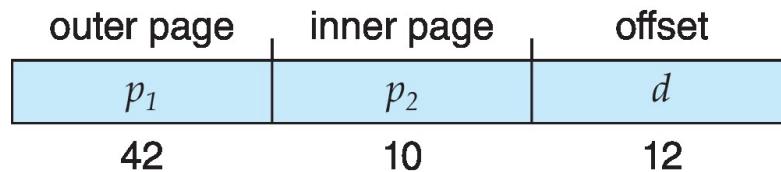
Address-Translation Scheme





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 2nd outer page table
 - But in the following example the 2nd outer page table is still 2^{34} bytes in size
 - ▶ And possibly 4 memory access to get to one physical memory location





- A system implements a **paged** virtual memory for each process using **three levels of page tables**. The **MMU** includes a **TLB**. The time to access main memory is T . The TLB can be accessed very quickly (in essentially zero time). The running process needs to read the contents of virtual address v . What must the hit ratio of the TLB be if we wish this operation to take time $2T$, on average?
- Reading virtual address v will require four memory accesses in case of a TLB miss, one for the segment table, and one for each level of page table, and one for the actual physical memory location corresponding to v . In case of a TLB hit, only one memory access (for the physical location itself) will be required. So, if h is the hit ratio of the TLB, we need $4T(1 - h) + T(h) = 2T$, i.e., $h = 0.67$.





-
- How many bits are required for each page table entry?
 - There can be at most $2^{21}/2^{10}$ frames in main memory, so a page table entry will require 11 bits for a frame number, plus protection and reference bits, etc.
 - What is the maximum number of entries in a page table?
 - A page table may have up to $2^{24}/2^{10} = 2^{14}$ entries.
 - How many bits are there in a virtual address?
 - A virtual address has 24 bits.





- To which physical address will the virtual address 1524 translate to?
- Virtual address 1524 lies on page 1 which resides in frame 8 with an offset of 1524 modulo 1024, or 500. So, 1524 maps to physical address $8 * 1024 + 500 = 8192 + 500 = 8692$.
- Which virtual address will translate to physical address 10020?
- Physical address 10020 is in frame 9, with an offset of 804. Virtual page 4 maps to frame 9, so virtual address $4 * 1024 + 804 = 4900$ will map to 10020.





- We introduced two-level page tables to reduce the overhead of storing page tables
 - Each page table costs $2^{32} / 2^{12} \times 4 \text{ B} = 4\text{MB}$ to store
- But even if we add another level, isn't the overhead the same?
 - 1024 secondary page tables
 - Each secondary page table has 2^{10} PTEs, thus has a size of 4KB
 - Total size of these page tables is $1024 \times 4\text{KB} = 4\text{MB}$...
 - In fact, we also have one master page table, which has a size of 4KB...





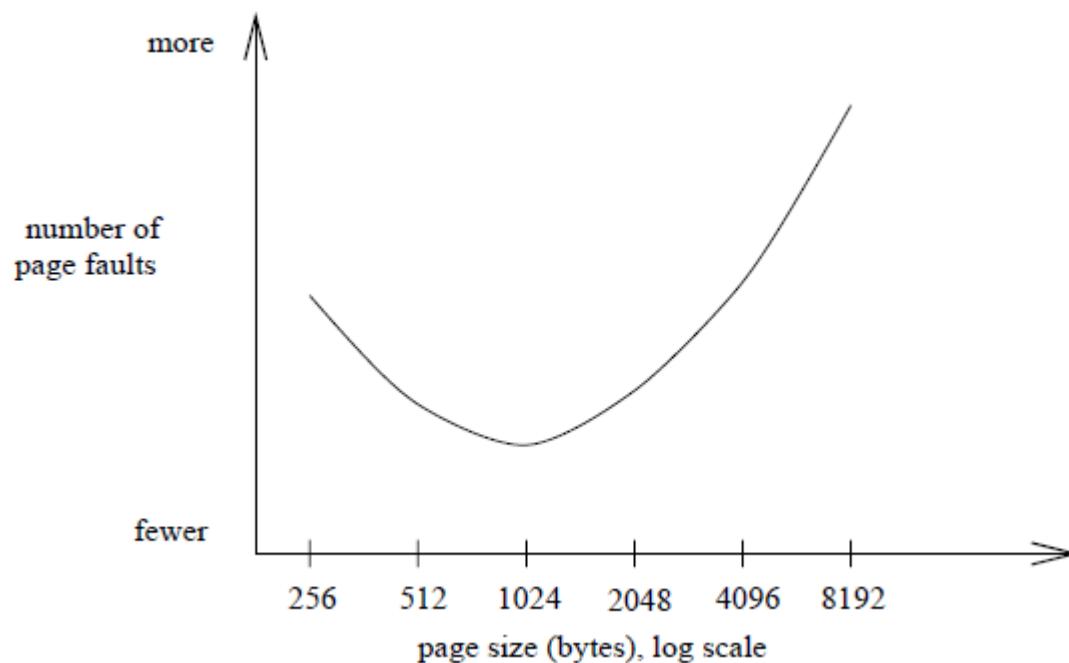
What is the page size?

- Why 4KB Page?
- An empirical choice
- Can choose larger size e.g., 8KB or even 4MB
 - Linux “Huge pages”
- But typically not smaller
- Cons of using smaller page size
 - More page tables needed
 - Likely more page faults
- Cons of using larger page size
 - Internal fragmentation





Suppose that a series of experiments is run to measure the number of page faults incurred by a test program as a function of the page size used by a demand paging virtual memory system. For each page size, the test program is run alone in the system and the total number of page faults that occur is measured. The results of the experiments are shown in the graph below.



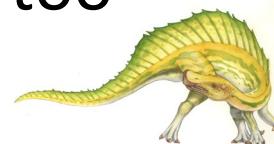


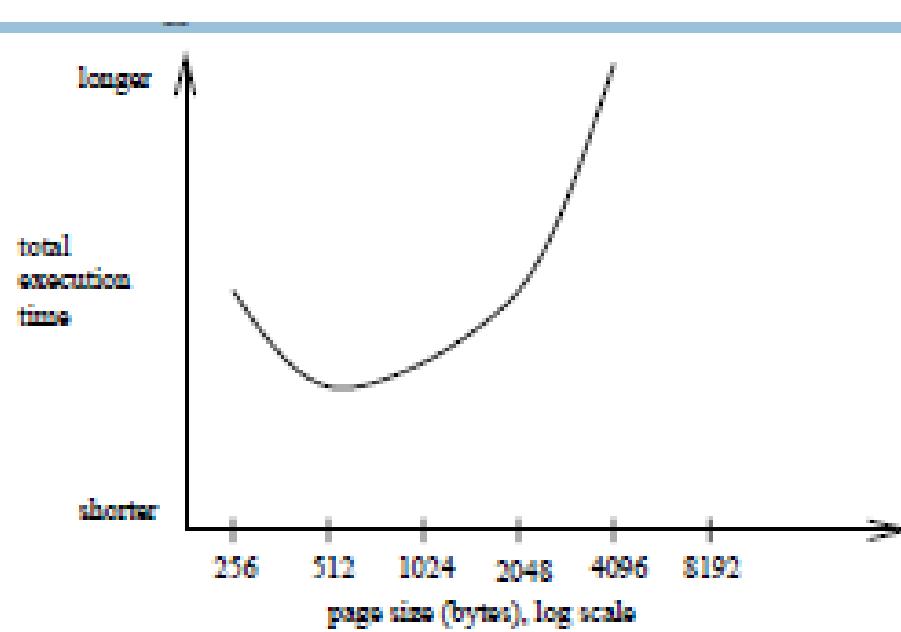
-
- a. Explain, in two or three sentences, the shape of the curve in the graph. Why is the number of page faults higher when the page size is very large or very small? Your explanation should involve the working set of the test program.
 - b. Suppose that the experiment described above had measured the total execution time of the test program rather than the number of page faults. Sketch the total execution time as a function of the page size, assuming the number of page faults varies as shown in part (a). State explicitly any assumptions that you make.





- a. Suppose that part or all of a process' working set is not in memory and must be paged in. If pages are very small, more page faults will be required to bring in the missing parts of the working set. This is what drives the curve up when the pages get too small. Large pages reduce the number of page faults required to bring in the working set. However, as pages become larger it becomes more likely that they will contain non-working-set data as well as working-set data. This unnecessary data will occupy memory space that could have been used for the working set. This drives the curve up when the page size gets too large.





This curve assumes that faults for larger pages take longer to handle than faults for smaller pages. This will tend to drive the minimum of the curve towards the left (where each page fault is cheaper). The curve sketched assumes this effect is strong enough to make a 512 byte page size better than an 1024 byte page size, though the latter results in fewer faults. If it is assumed that all faults take the same amount of time to handle, the minimum of the curve should remain at 1024.





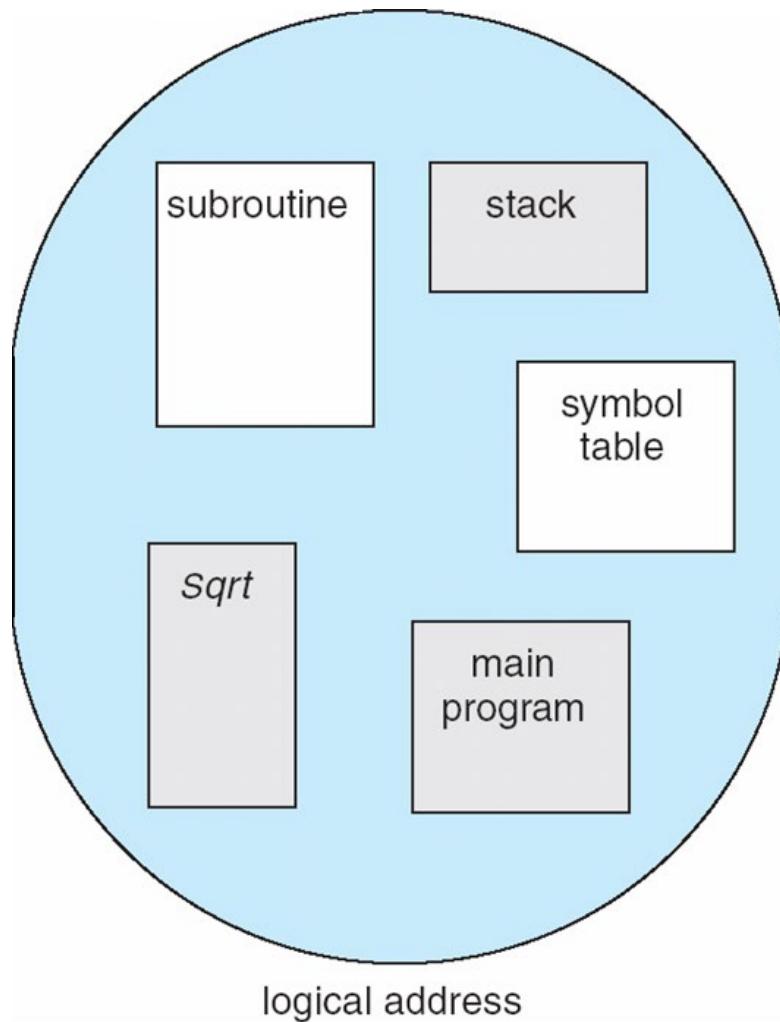
Segmentation

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



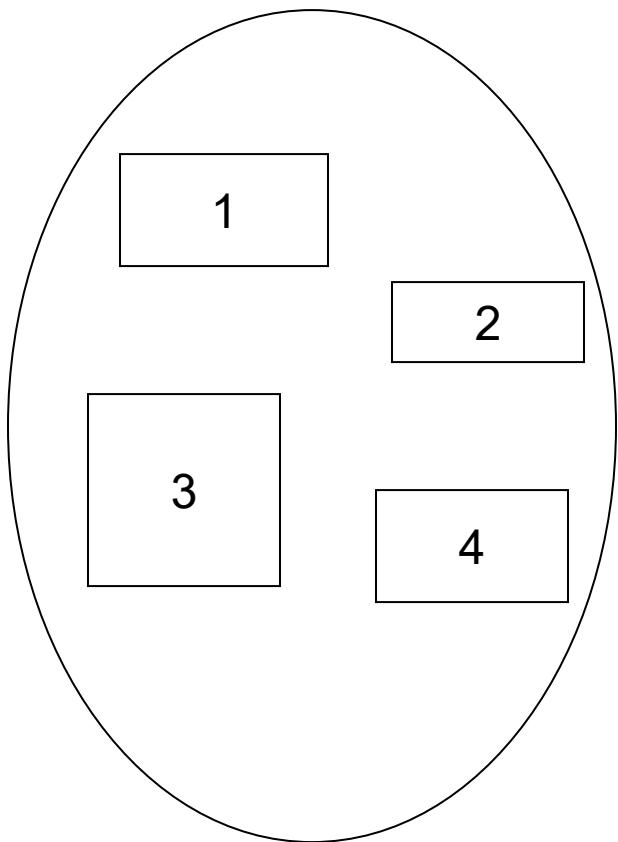


User's View of a Program

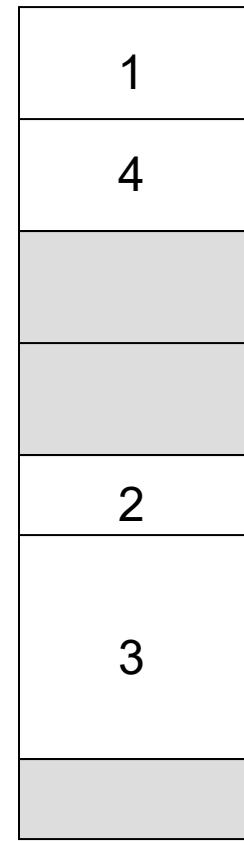




Logical View of Segmentation



user space



physical memory space





Segmentation Architecture

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





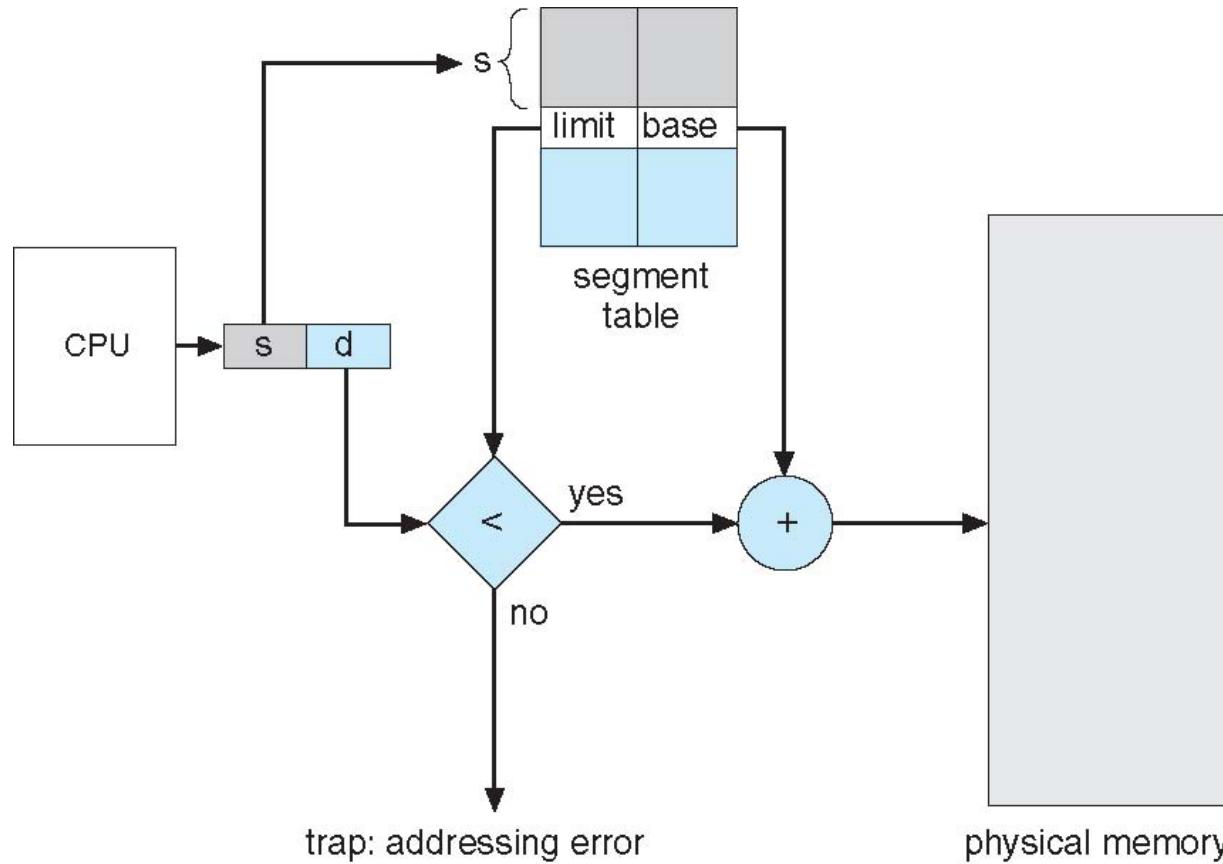
Segmentation Architecture (Cont.)

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





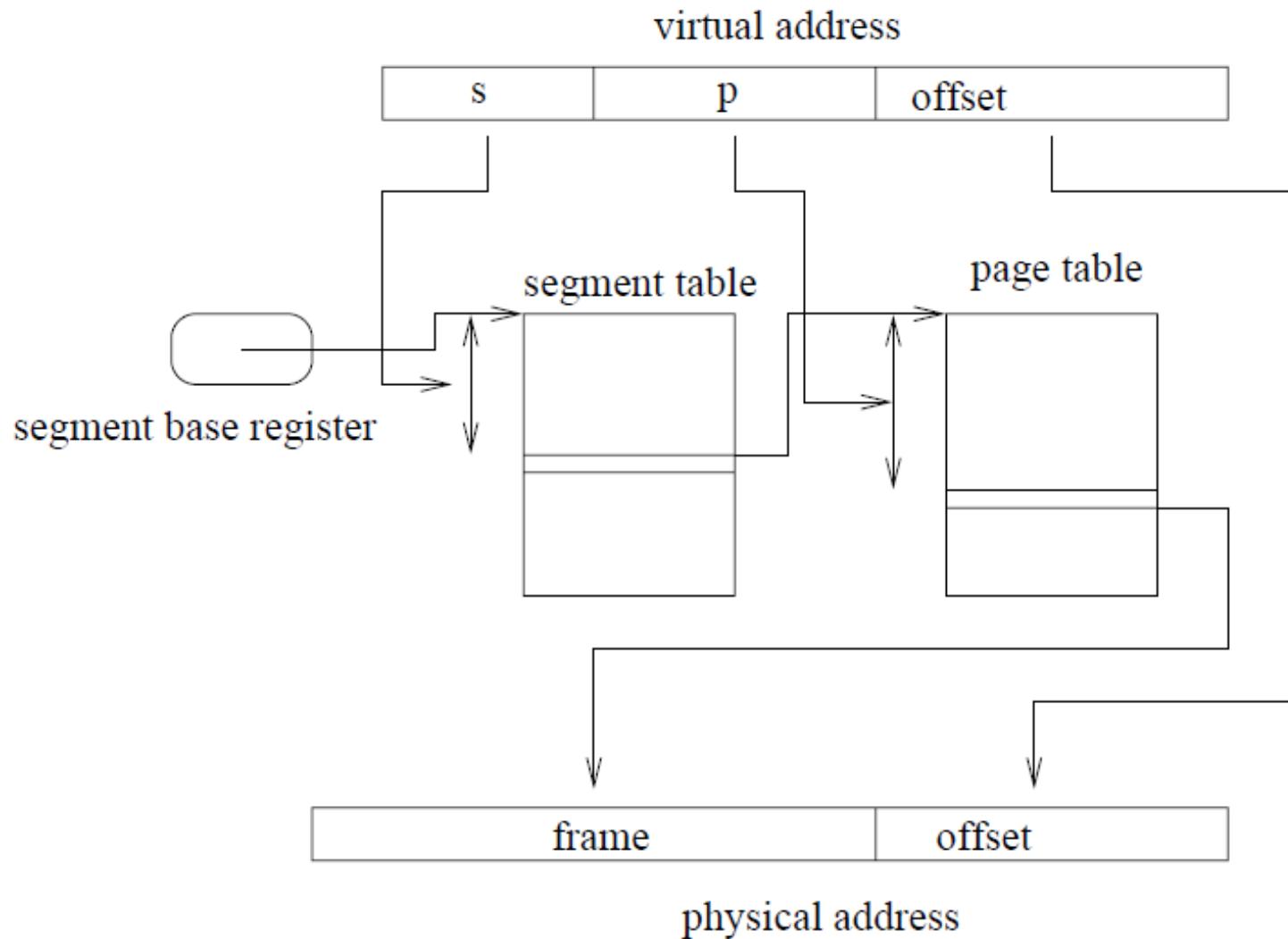
Segmentation Hardware

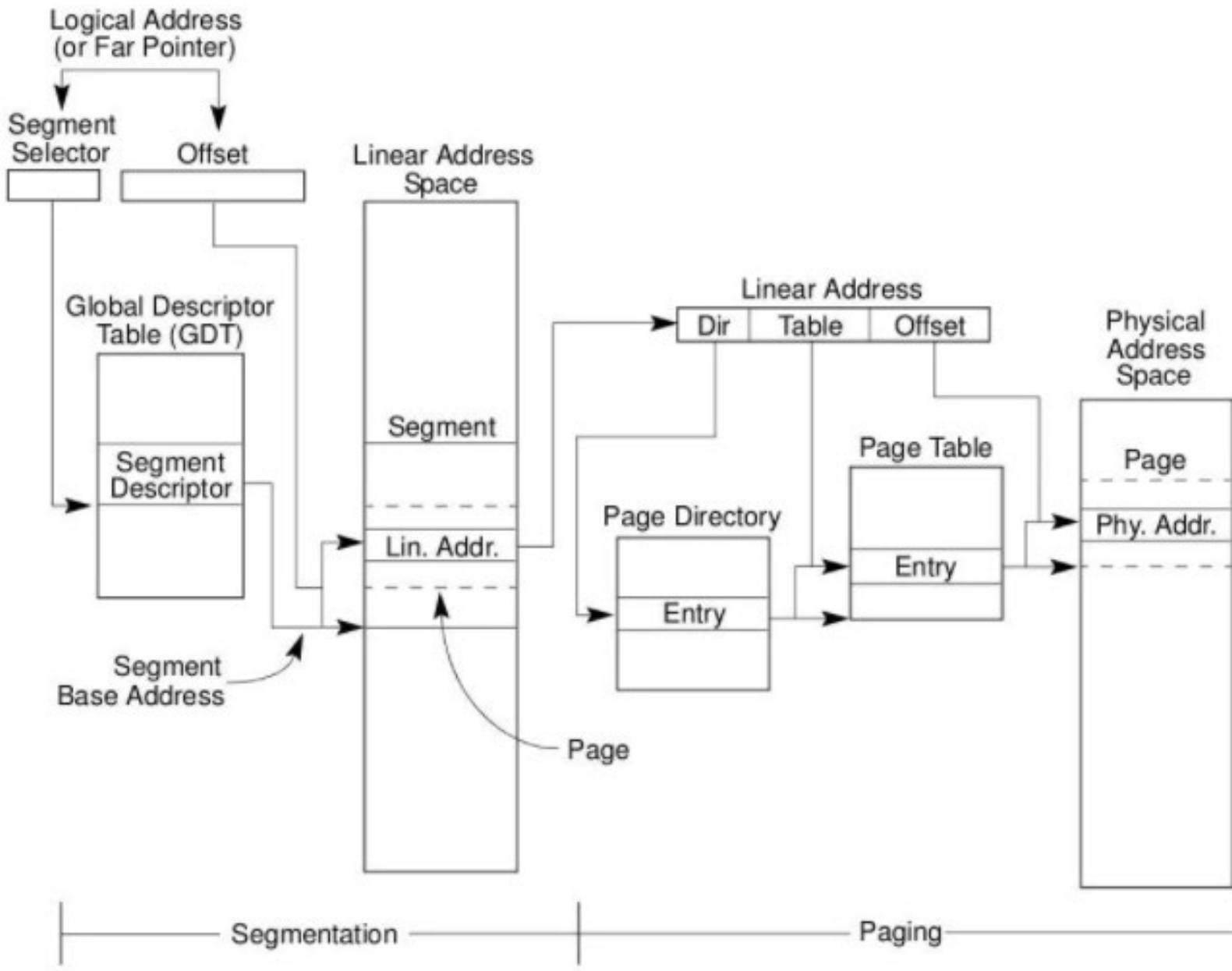




- Briefly sketch the address translation process of a system that utilizes a combination of segmentation and paging. Your diagram should include all relevant components (such a segment table, page table, virtual address, resulting physical address, etc.).









“In 64-bit mode, segmentation is generally (but not completely) disabled, creating a flat 64-bit linear-address space. **The processor treats the segment base of CS, DS, ES, SS as zero, creating a linear address that is equal to the effective address.** The FS and GS segments are exceptions. These segment registers (which hold the segment base) can be used as an additional base registers in linear address calculations. They facilitate addressing local data and certain operating system data structures.”

[Intel® 64 and IA-32 Architectures Software Developer's Manual](#)

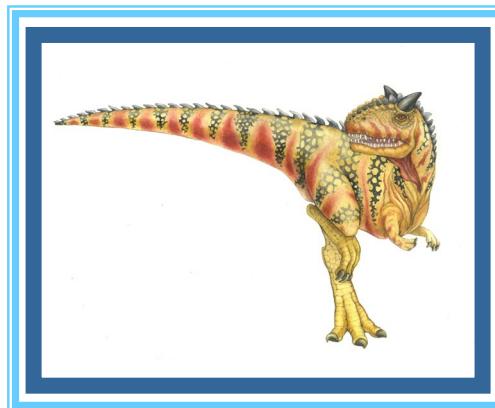




- A virtual memory system uses both paging and segmentation. Process virtual addresses consist of a total of 32 bits. The page size is 1024 (2^{10}) bytes. A page table may occupy at most one frame, and the size of a page table entry is 4 bytes. A single process may have up to 64 (2^6) segments.
 - How many levels of page tables are required for virtual address translation in this system?
 - What is the maximum size of a virtual memory segment?
 - Draw an address translation diagram that shows how a virtual address is translated in this system. Your diagram must show each component of the virtual and physical addresses, and the size of each component. It must show any tables used in translation, and their lengths. It must show how the components of the virtual address are used to index the tables, and how the components of the physical address are determined.



Chapter 9: Main Memory





Chapter 9: Memory Management

- 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

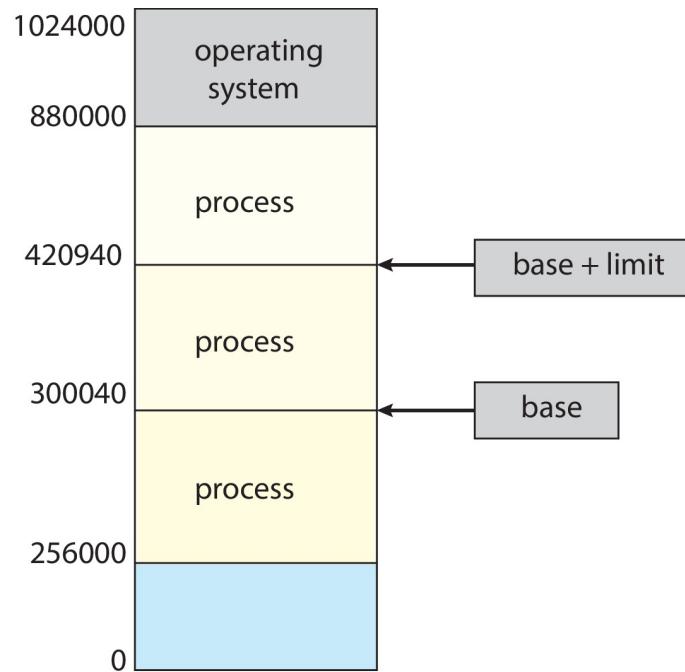
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of:
 - addresses + read requests, or
 - address + data and write requests
- Register access is done in one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





Protection

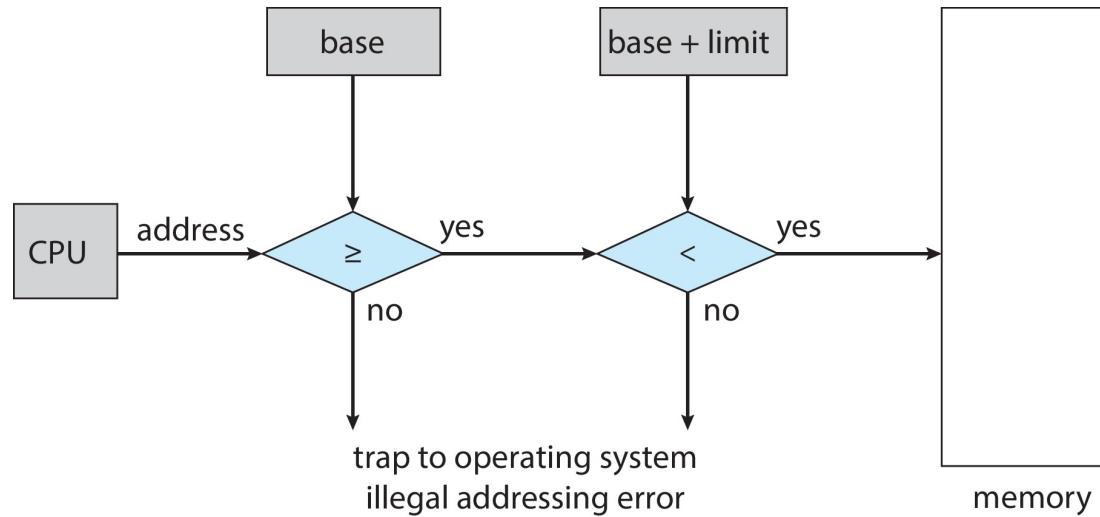
- Need to ensure that a process can access only those addresses in its address space.
- We can provide this protection by using a pair of **base** and **limit registers** define the logical address space of a process





Hardware Address Protection

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



- the instructions to loading the base and limit registers are privileged





Address Binding

- Programs on disk, ready to be brought into memory to execute from an **input queue**
 - Without support, must be loaded into address 0000
- Inconvenient to have first user 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
 - ▶ i.e., “14 bytes from beginning of this module”
 - Linker or loader will bind relocatable addresses to absolute addresses
 - ▶ i.e., 74014
 - Each binding maps one address space to another





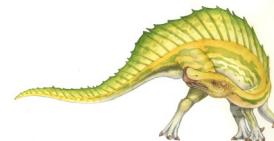
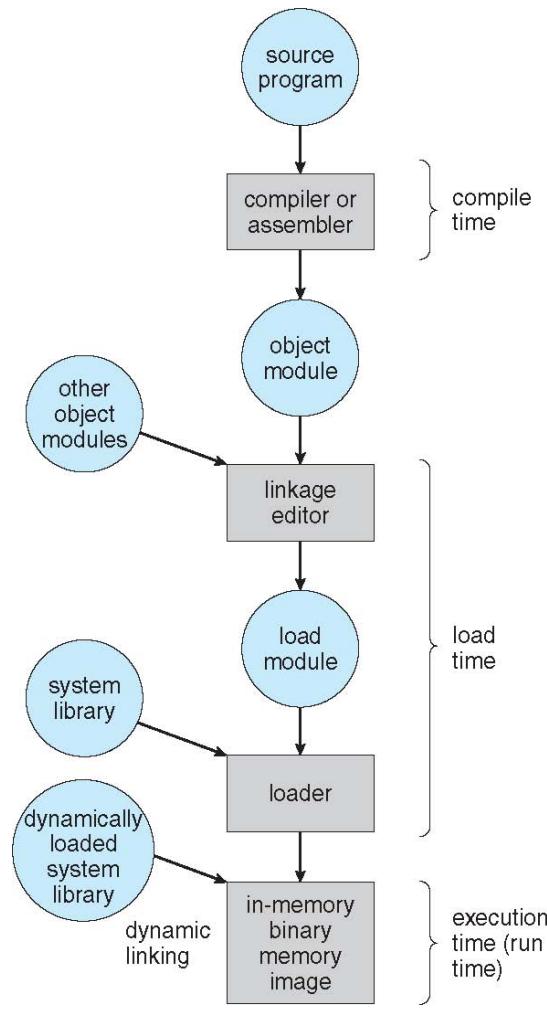
Binding of Instructions and Data to Memory

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





Multistep Processing of a User Program





Logical vs. Physical Address Space

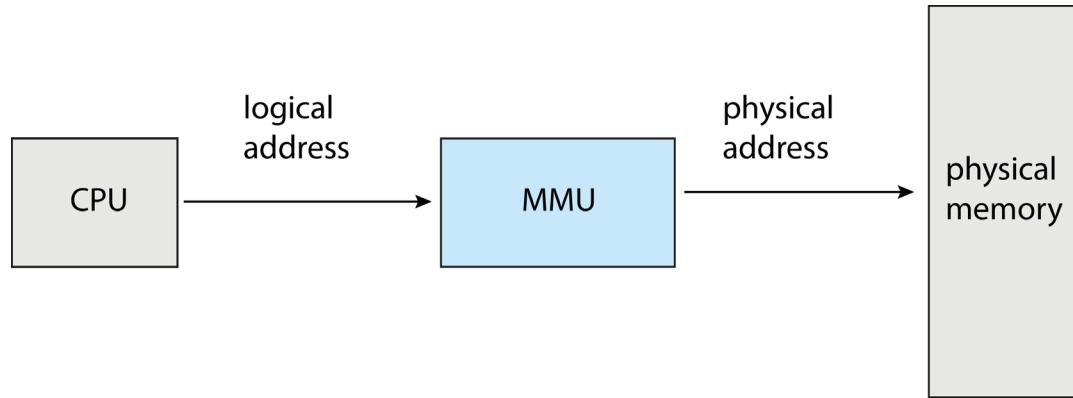
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
 - **Logical address** – generated by the CPU; also referred to as **virtual address**
 - **Physical address** – address seen by the memory unit
- Logical and physical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



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





Memory-Management Unit (Cont.)

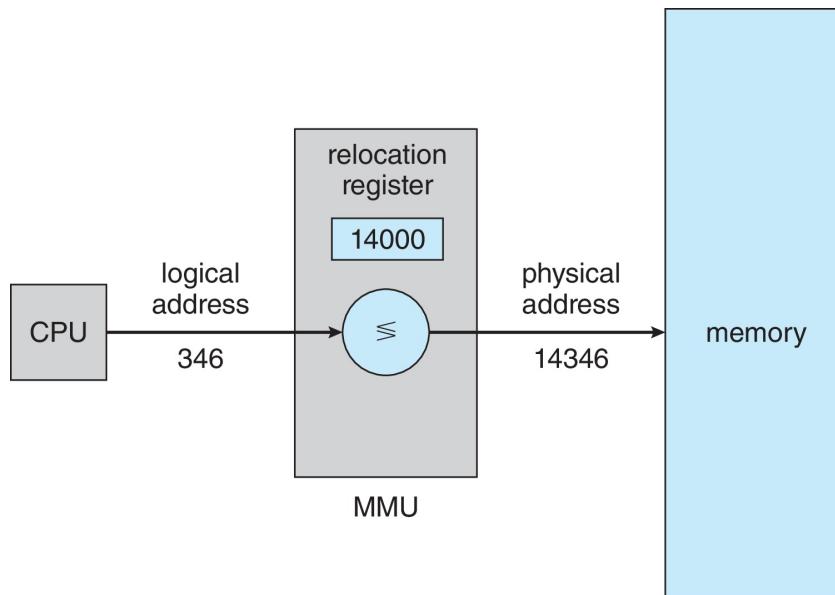
- Consider simple scheme, which is a generalization of the base-register scheme.
- The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical*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





Memory-Management Unit (Cont.)

- 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





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 into two **partitions**:
 - Resident operating system, usually held in low memory with interrupt vector
 - User processes then held in high memory
 - Each process contained in single contiguous section of memory

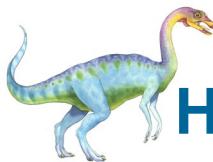




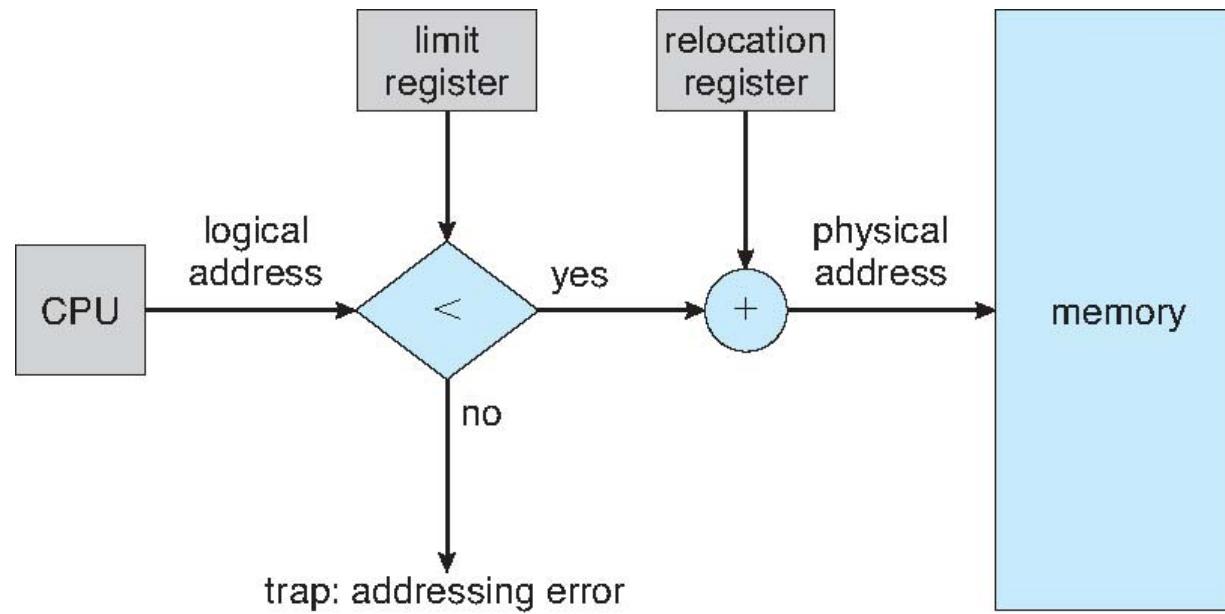
Contiguous Allocation (Cont.)

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
 - Base register contains value of smallest physical address
 - Limit register contains range of logical addresses – each logical address must be less than the limit register
 - MMU maps logical address *dynamically*
 - Can then allow actions such as kernel code being **transient** and kernel changing size





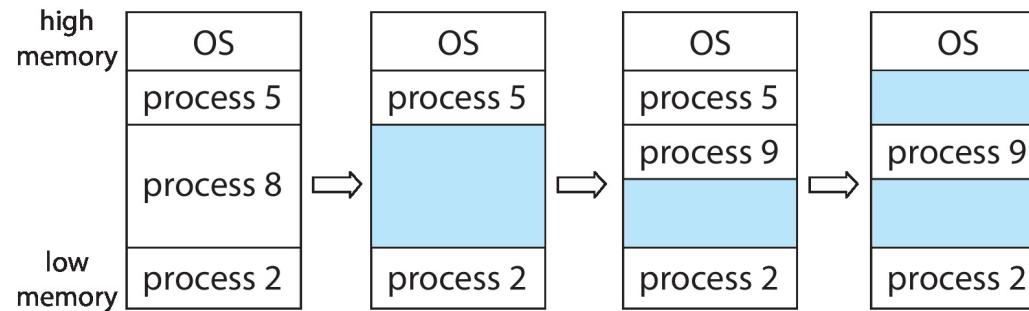
Hardware Support for Relocation and Limit Registers





Variable Partition

- 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 combined
 - Operating system maintains information about:
 - a) allocated partitions
 - b) free partitions (hole)





Dynamic Storage-Allocation Problem

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

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

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





Fragmentation

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





Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
 - Shuffle memory contents to place all free memory together in one large block
 - Compaction is possible *only* if relocation is dynamic, and is done at execution time
 - 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





Paging

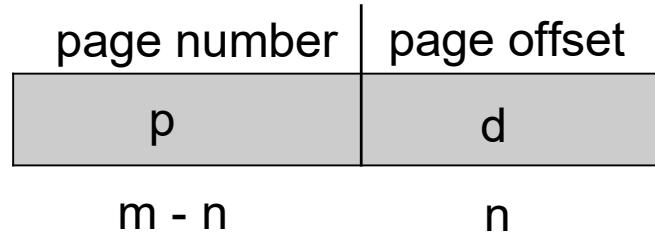
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
 - Avoids external fragmentation
 - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
 - Size is power of 2, 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 program of size N pages, need to find N free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation





Address Translation Scheme

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

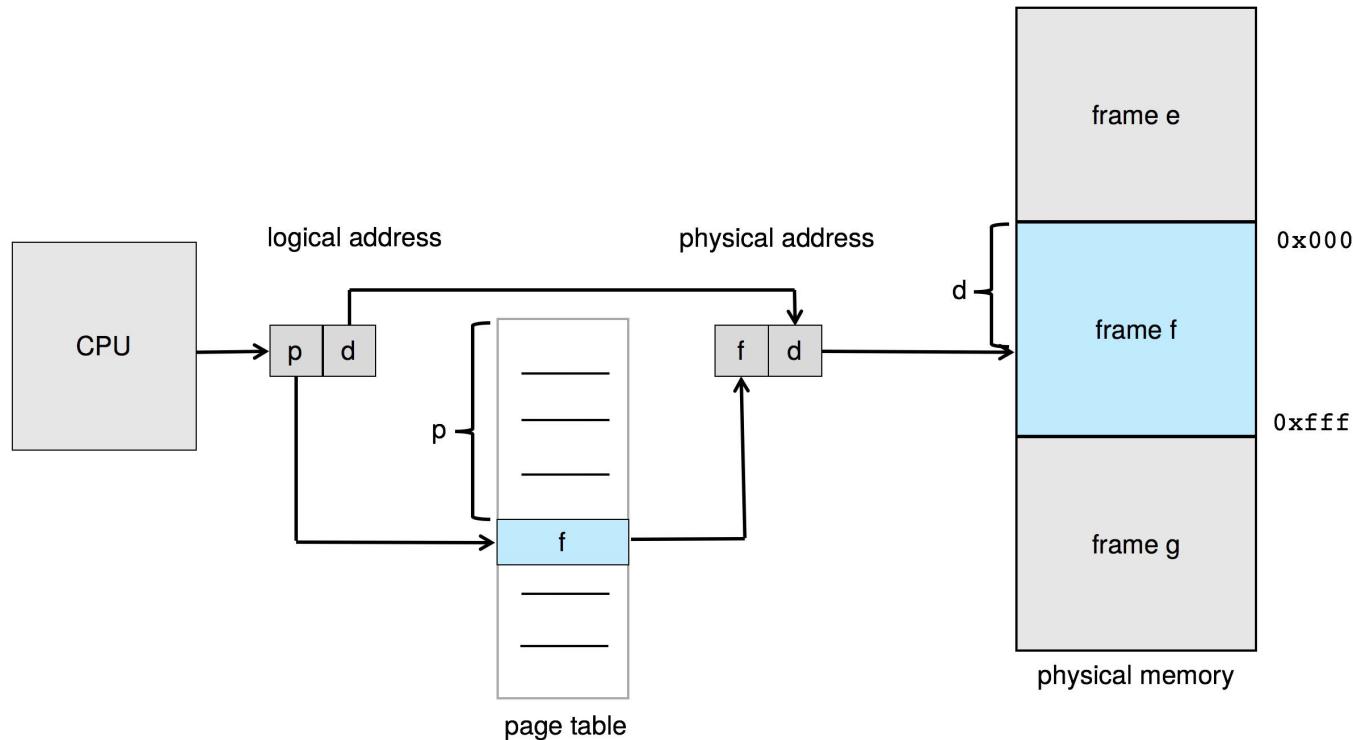


- For given logical address space 2^m and page size 2^n



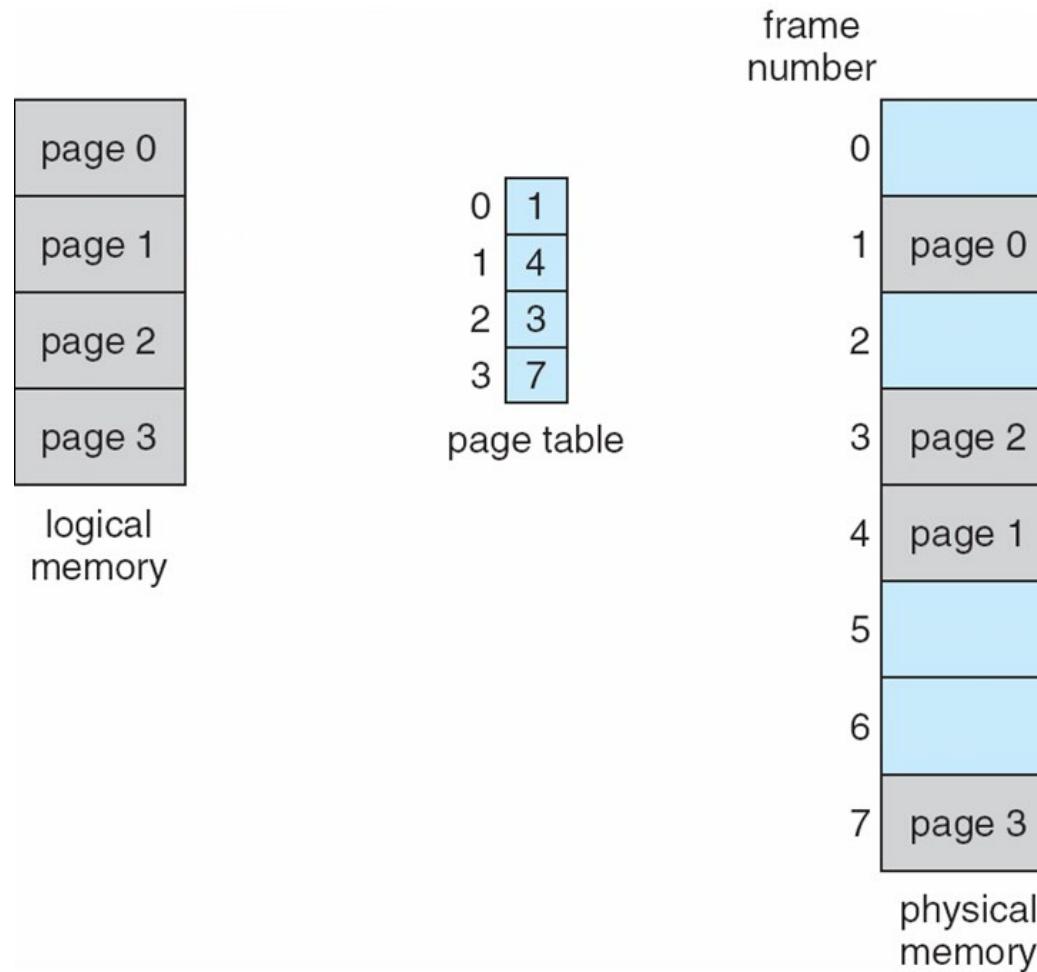


Paging Hardware





Paging Model of Logical and Physical Memory





Paging Example

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

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

logical memory

0	5
1	6
2	1
3	2

page table

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

physical memory

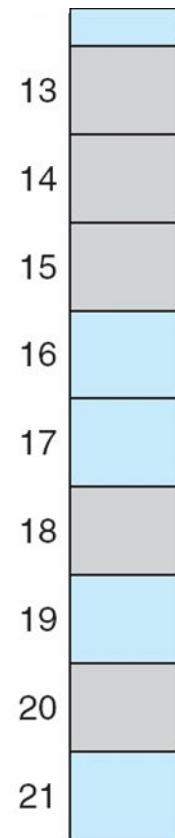
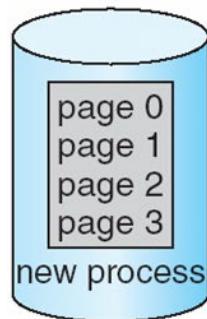




Free Frames

free-frame list

14
13
18
20
15

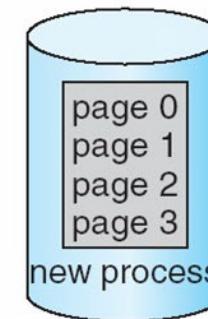


(a)

Before allocation

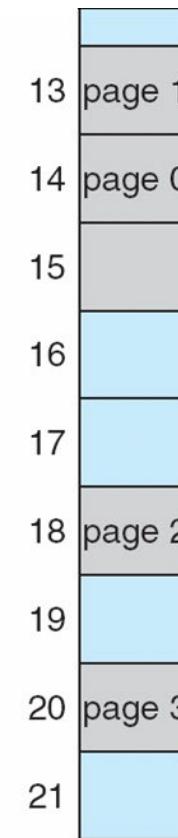
free-frame list

15



0	14
1	13
2	18
3	20

new-process page table



(b)

After allocation





Implementation of Page Table

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





Translation Look-Aside Buffer

- 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





Hardware

- Associative memory – parallel search

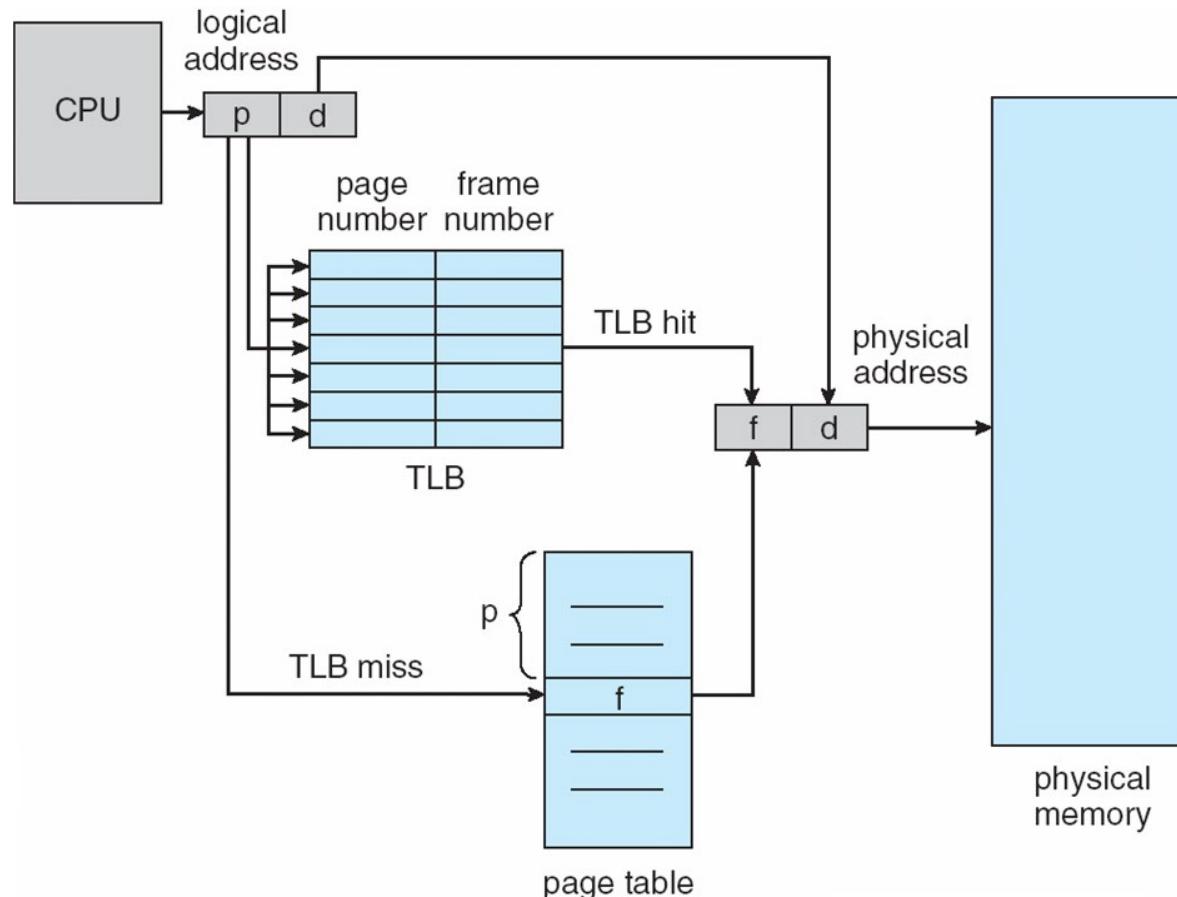
Page #	Frame #

- 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





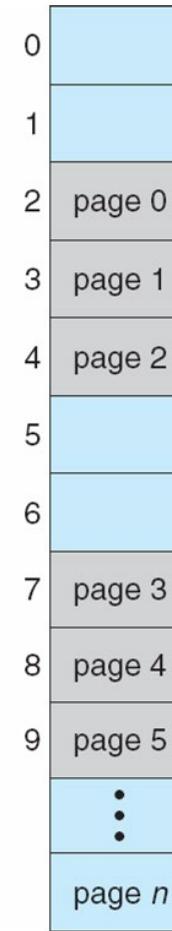
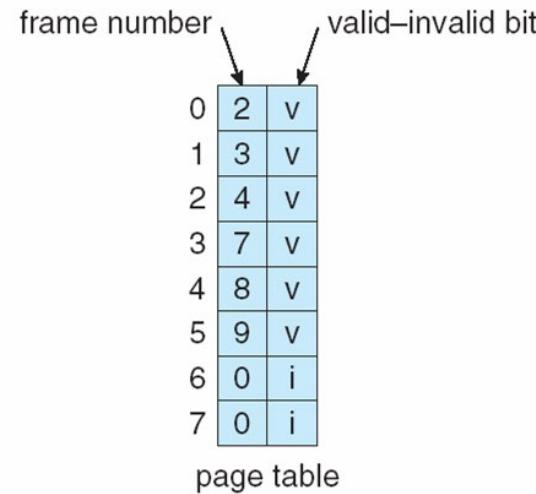
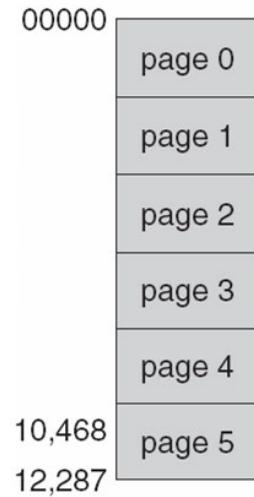
Memory Protection

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





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





Shared Pages

■ Shared code

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

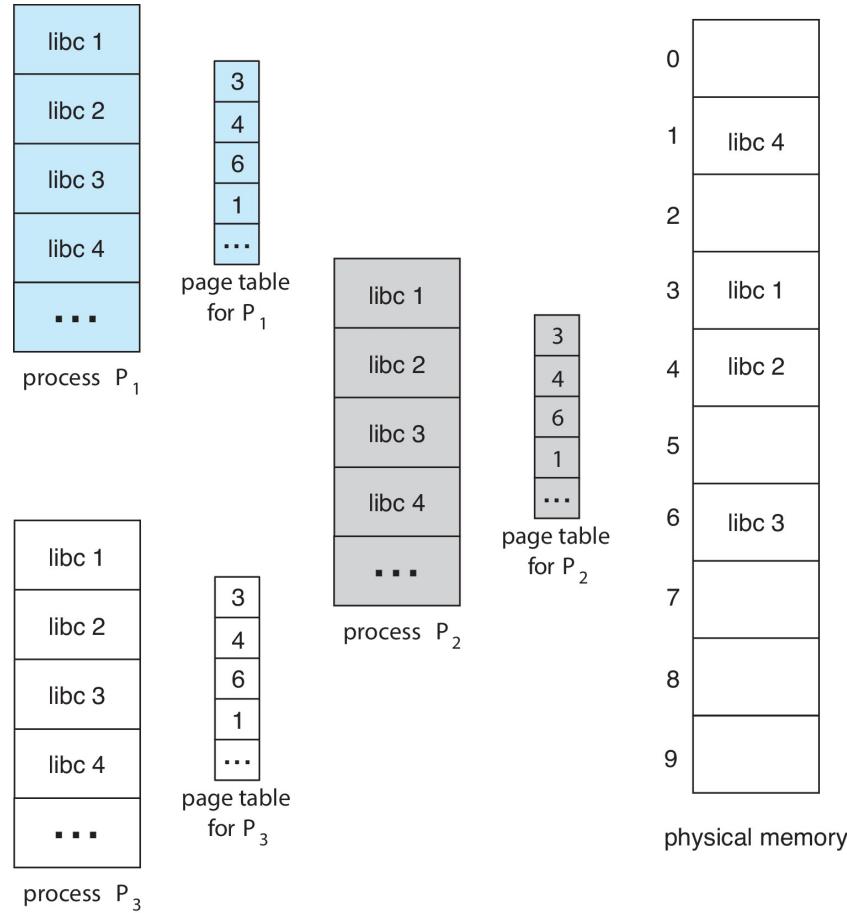
■ Private code and data

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





Shared Pages Example





Structure of the Page Table

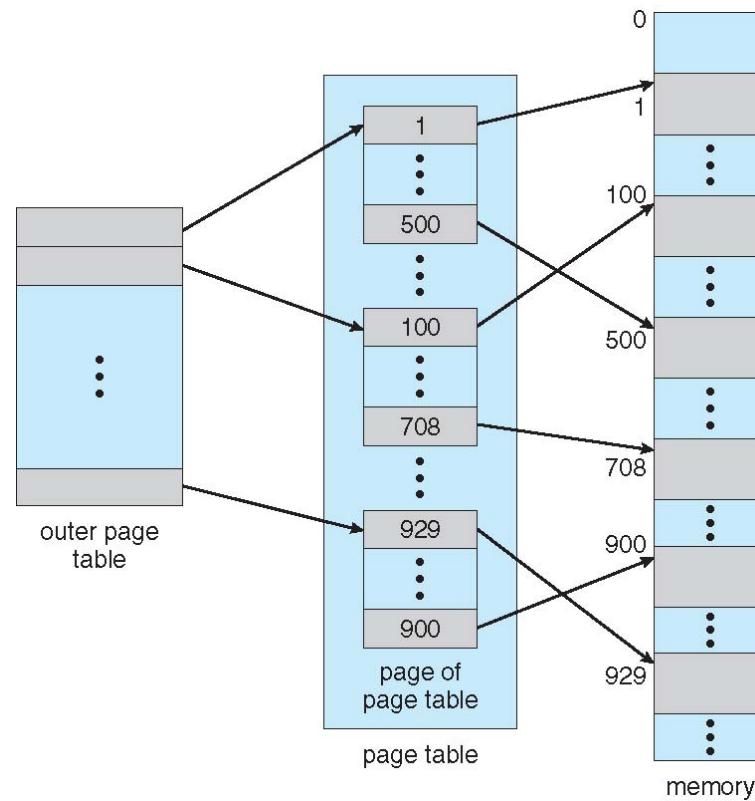
- Memory structures for paging can get huge using straight-forward methods
 - Consider a 32-bit logical address space as on modern computers
 - Page size of 4 KB (2^{12})
 - Page table would have 1 million entries ($2^{32} / 2^{12}$)
 - If each entry is 4 bytes → each process 4 MB of physical address 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

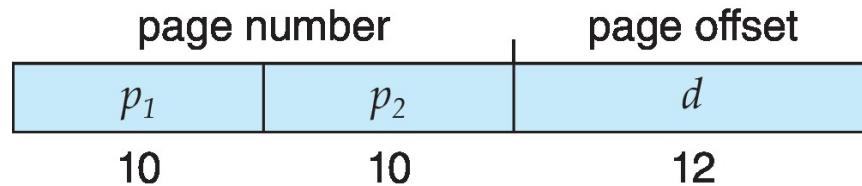
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table





Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
 - a page number consisting of 20 bits
 - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
 - a 10-bit page number
 - a 10-bit page offset
- Thus, a logical address is as follows:

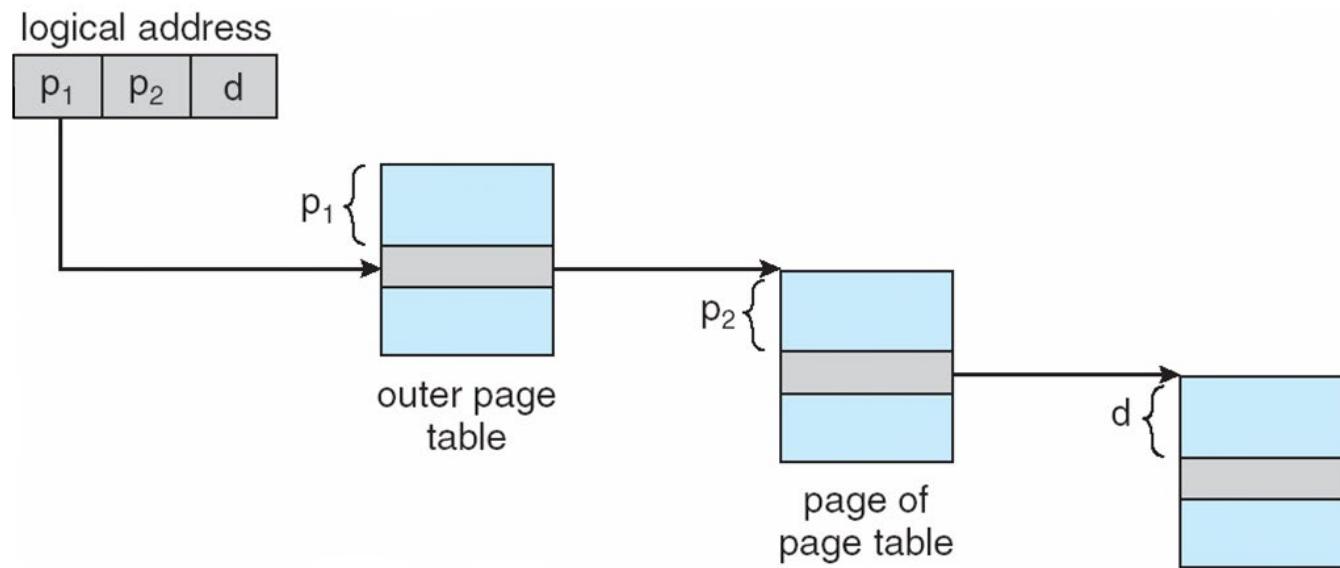


- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**





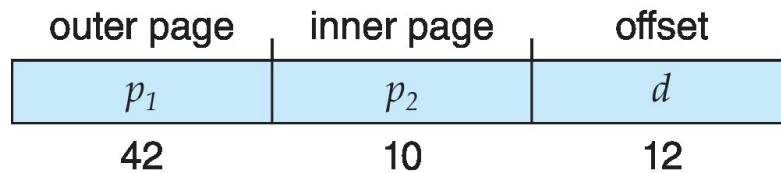
Address-Translation Scheme





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

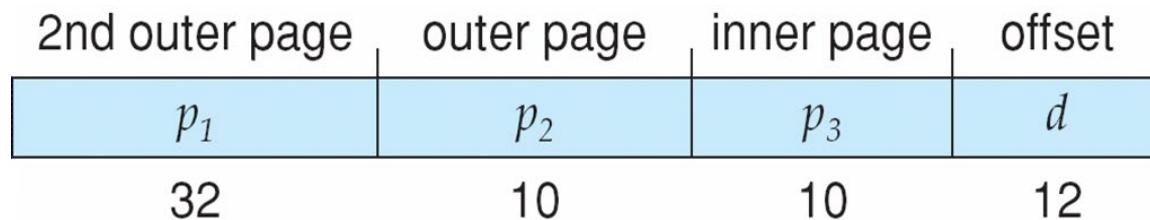
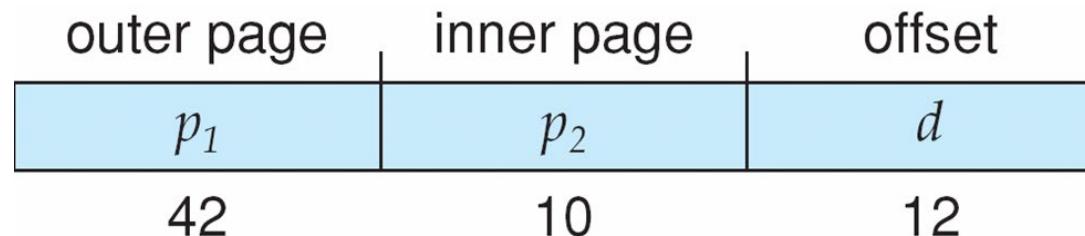


- Outer page table has 2^{42} entries or 2^{44} bytes
- One solution is to add a 2nd outer page table
- But in the following example the 2nd outer page table is still 2^{34} bytes in size
 - ▶ And possibly 4 memory access to get to one physical memory location

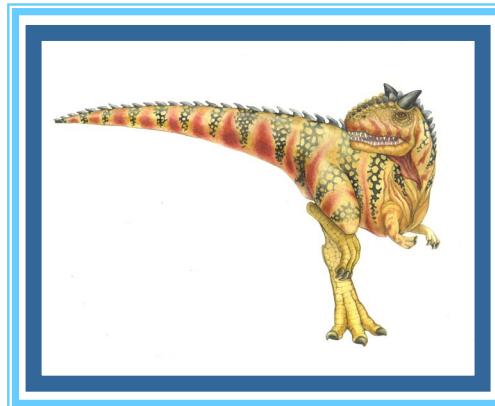




Three-level Paging Scheme



Chapter 10: Virtual Memory





Chapter 10: Virtual Memory

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





Objectives

- 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

- Code needs to be in memory to execute, but entire program rarely used
 - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
 - Program no longer constrained by limits of physical memory
 - Each program takes less memory while running -> more programs run at the same time
 - ▶ Increased CPU utilization and throughput with 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

- **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 therefore 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





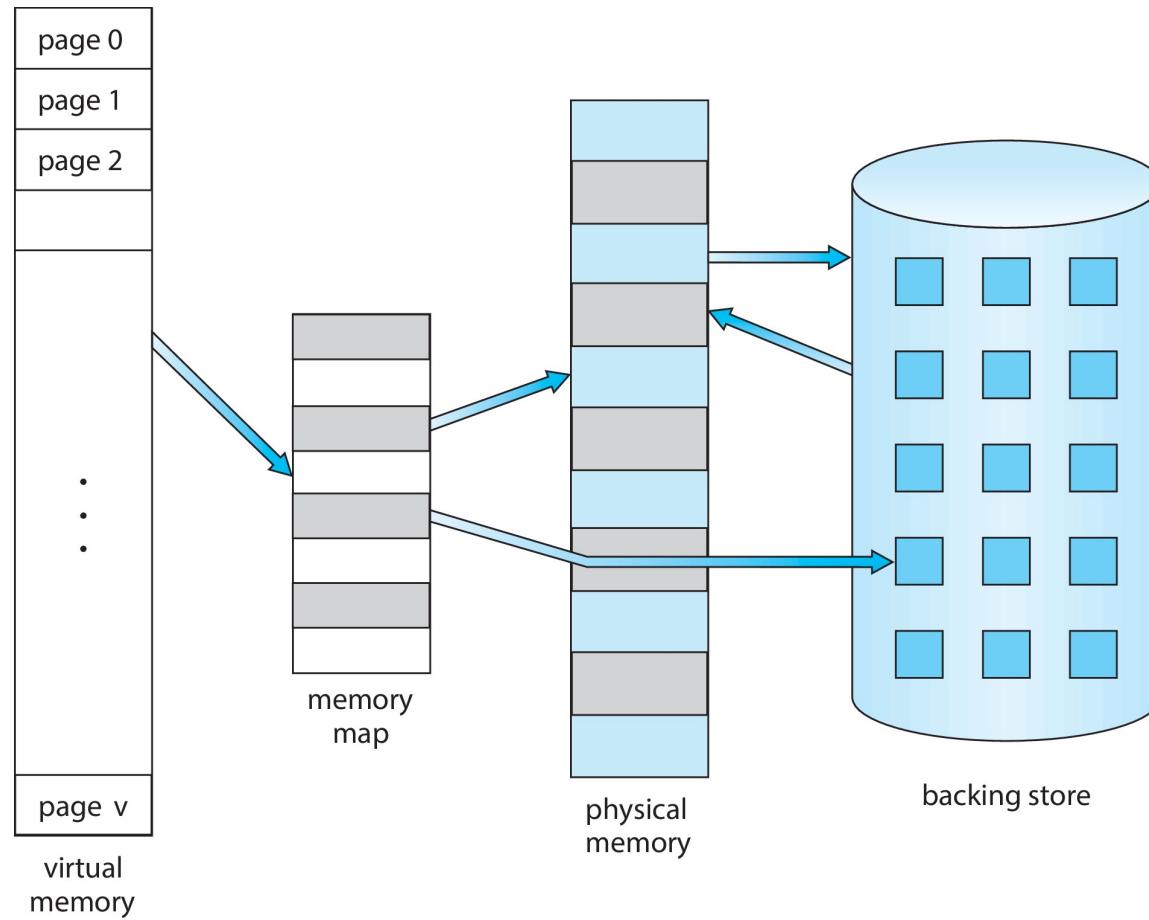
Virtual memory (Cont.)

- **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
- Virtual memory can be implemented via:
 - Demand paging
 - Demand segmentation





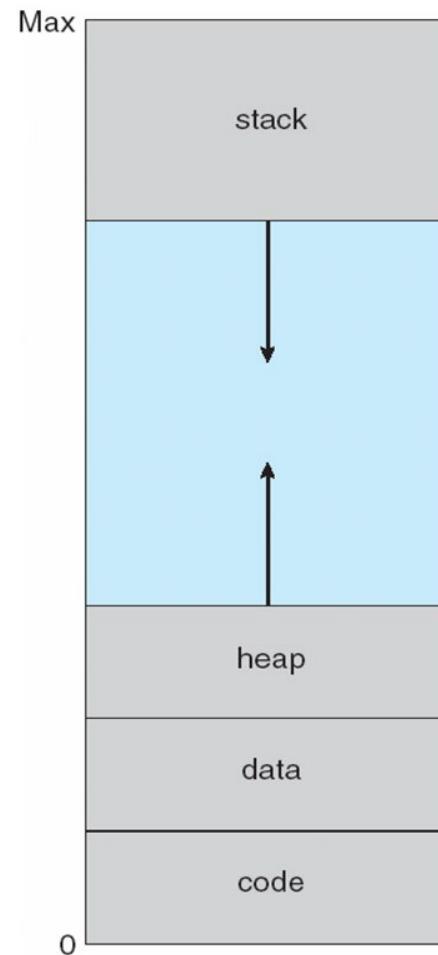
Virtual Memory That is Larger Than Physical Memory





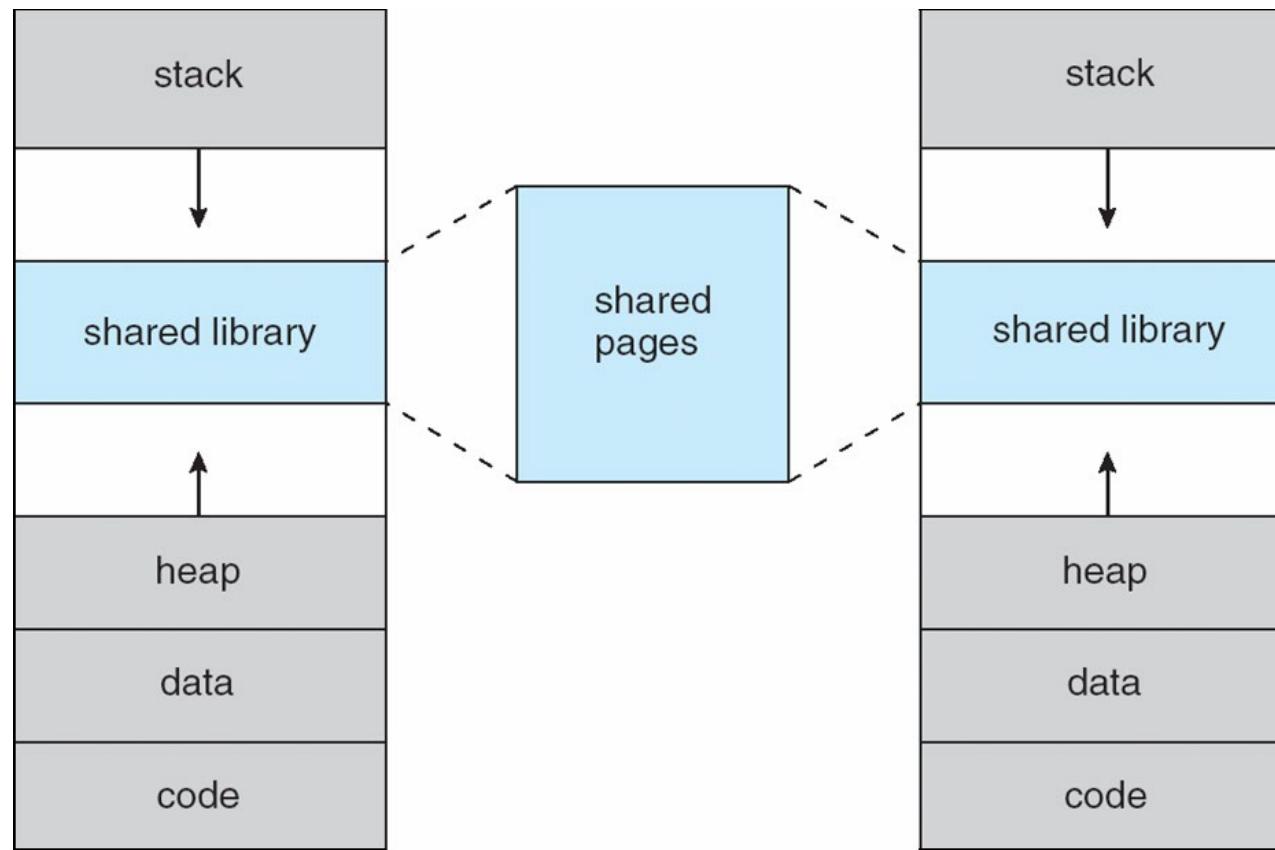
Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “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

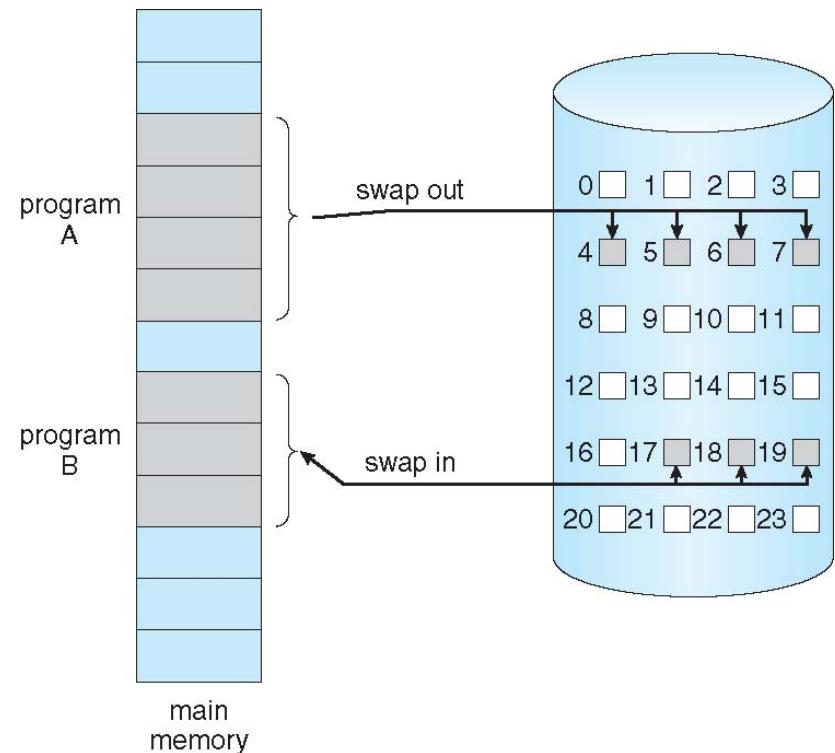
- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
 - Less I/O needed, no unnecessary I/O
 - Less memory needed
 - Faster response
 - More users
- Similar to paging system with swapping (diagram on right)
- Page is needed \Rightarrow reference to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





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 (diagram on right)





Basic Concepts

- 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 and 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

- 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
- Example of a page table snapshot:

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

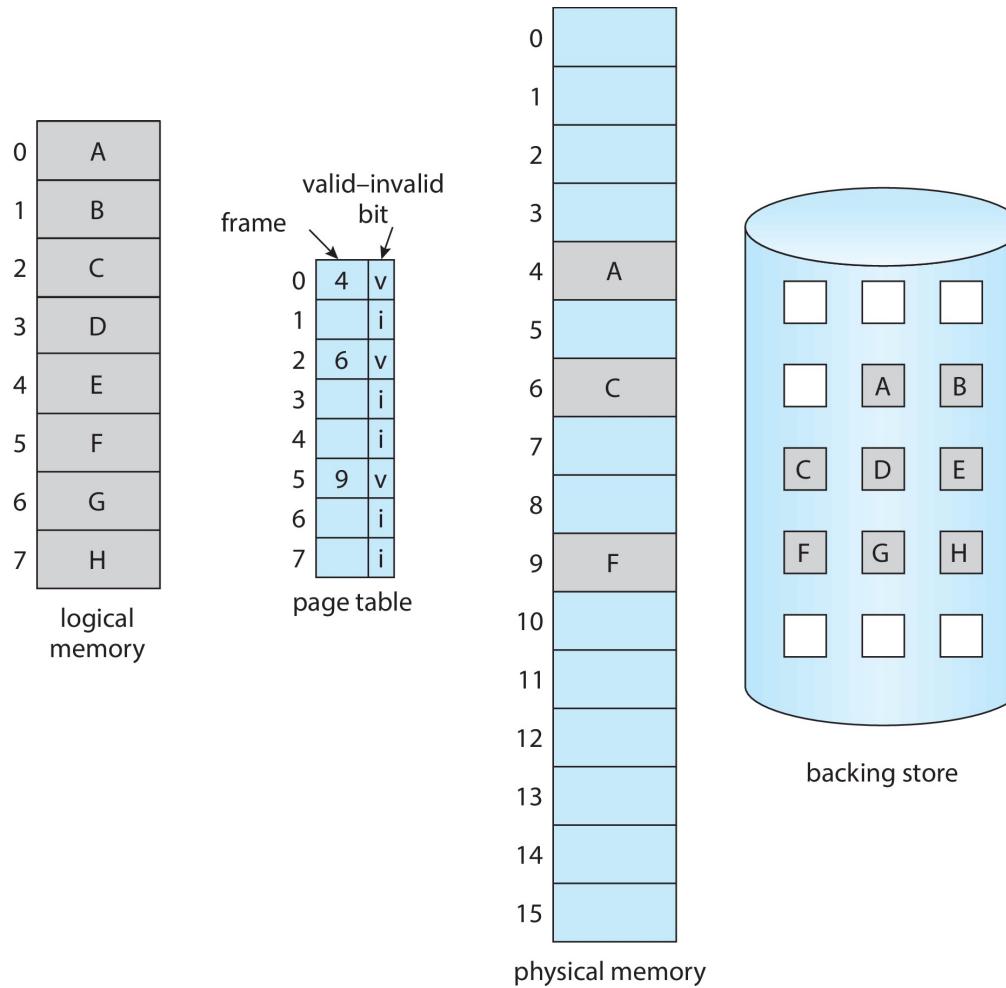
page table

- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault





Page Table When Some Pages Are Not in Main Memory





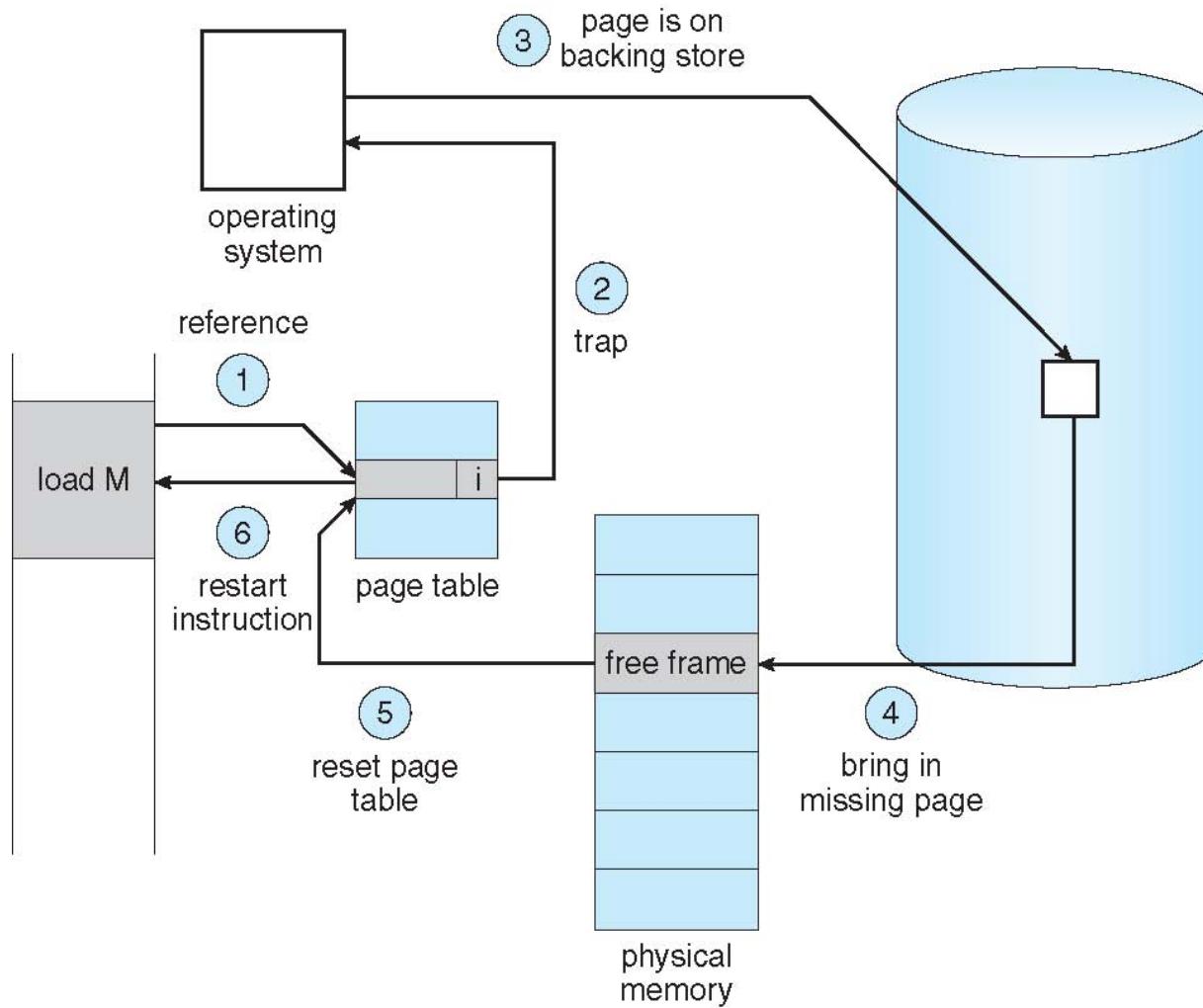
Steps in Handling Page Fault

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





Steps in Handling a Page Fault (Cont.)





Aspects of Demand Paging

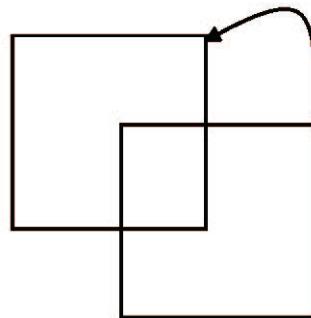
- Extreme case – start process with *no* pages in memory
 - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
 - And for every other process pages on first access
 - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
 - 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
 - Block move



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





Free-Frame List

- 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 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:
 - a) Wait in a queue for this device until the read request is serviced
 - b) Wait for the device seek and/or latency time
 - c) 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
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





Performance of Demand Paging

- 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 $0 \leq p \leq 1$
 - if $p = 0$ no page faults
 - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)
$$\begin{aligned} EAT = & (1 - p) \times \text{memory access} \\ & + p (\text{page fault overhead} \\ & + \text{swap page out} \\ & + \text{swap page in }) \end{aligned}$$





Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $$\begin{aligned} \text{EAT} &= (1 - p) \times 200 + p (8 \text{ milliseconds}) \\ &= (1 - p) \times 200 + p \times 8,000,000 \\ &= 200 + p \times 7,999,800 \end{aligned}$$
- 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 want performance degradation < 10 percent
 - $$\begin{aligned} 220 &> 200 + 7,999,800 \times p \\ 20 &> 7,999,800 \times p \end{aligned}$$
 - $p < .0000025$
 - < one page fault in every 400,000 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)





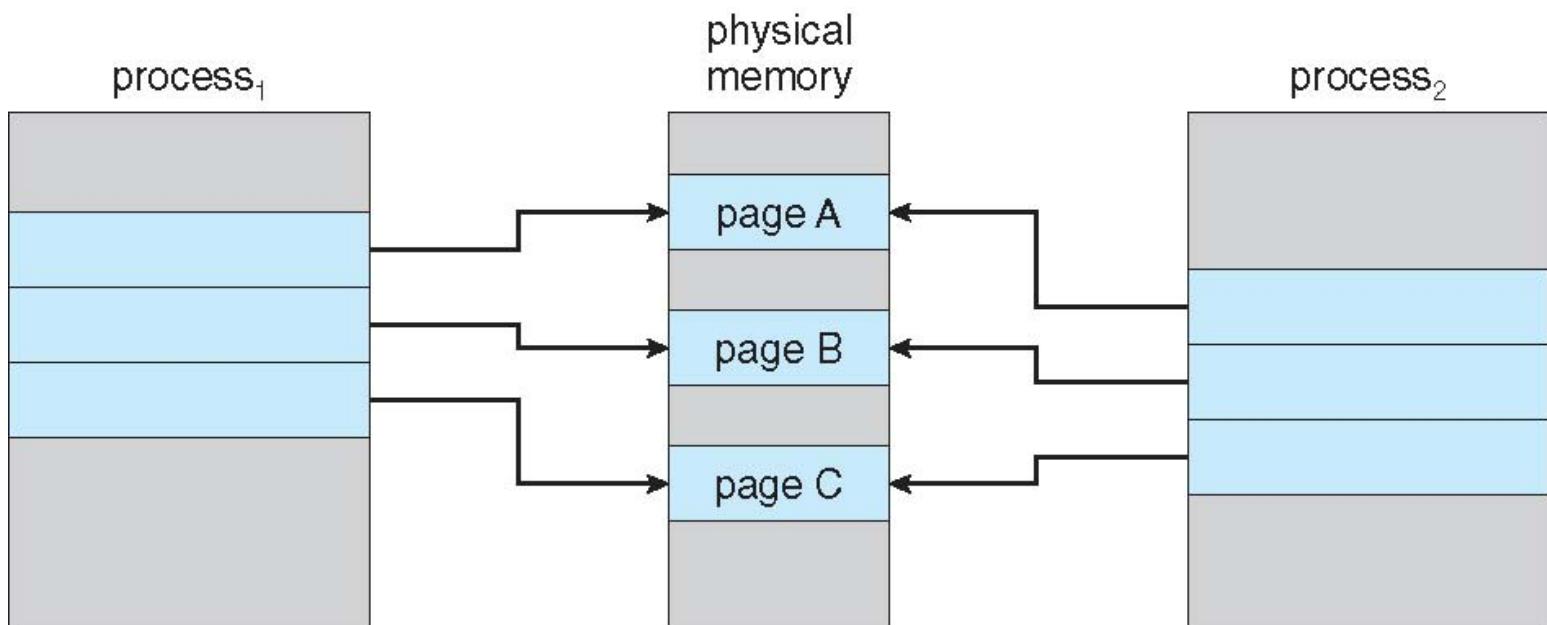
Copy-on-Write

- **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 suspend and child using copy-on-write address space of parent
 - Designed to have child call `exec()`
 - Very efficient



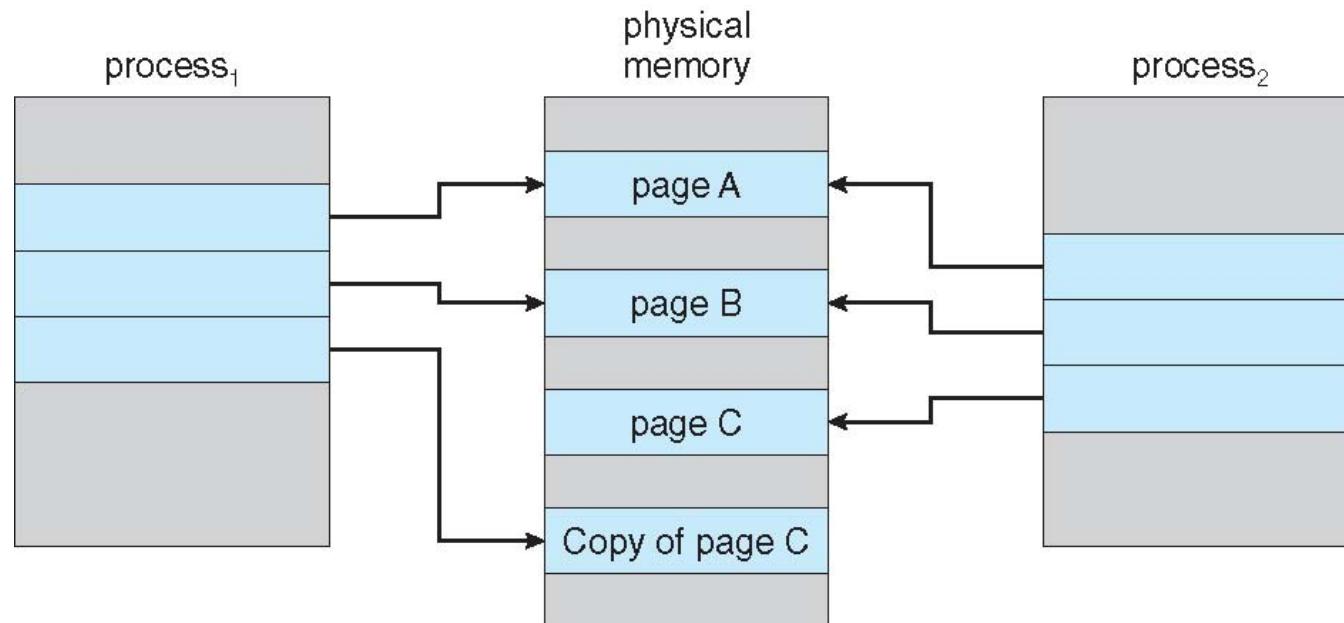


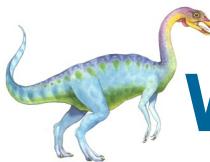
Before Process 1 Modifies Page C





After Process 1 Modifies Page C





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





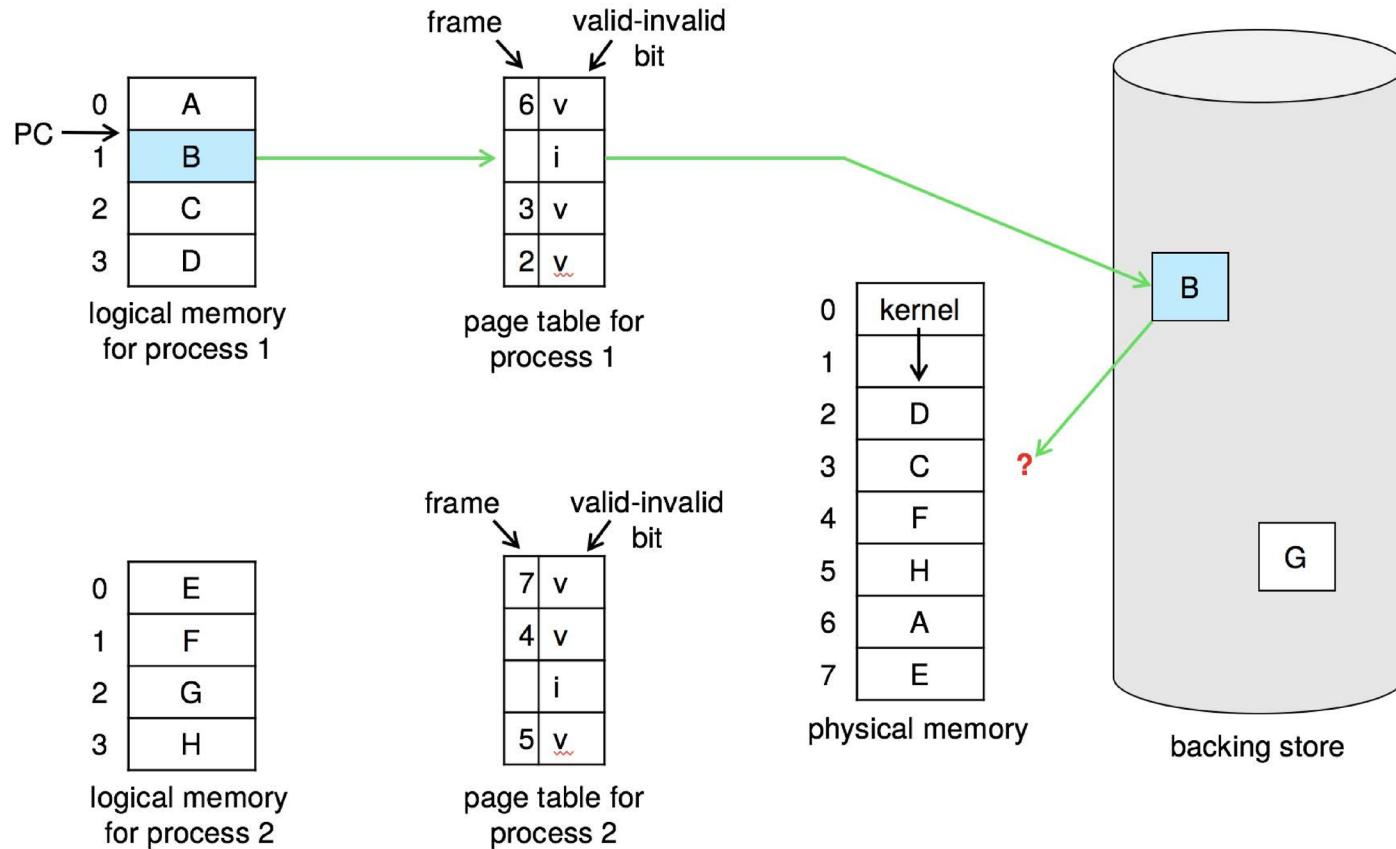
Page Replacement

- 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

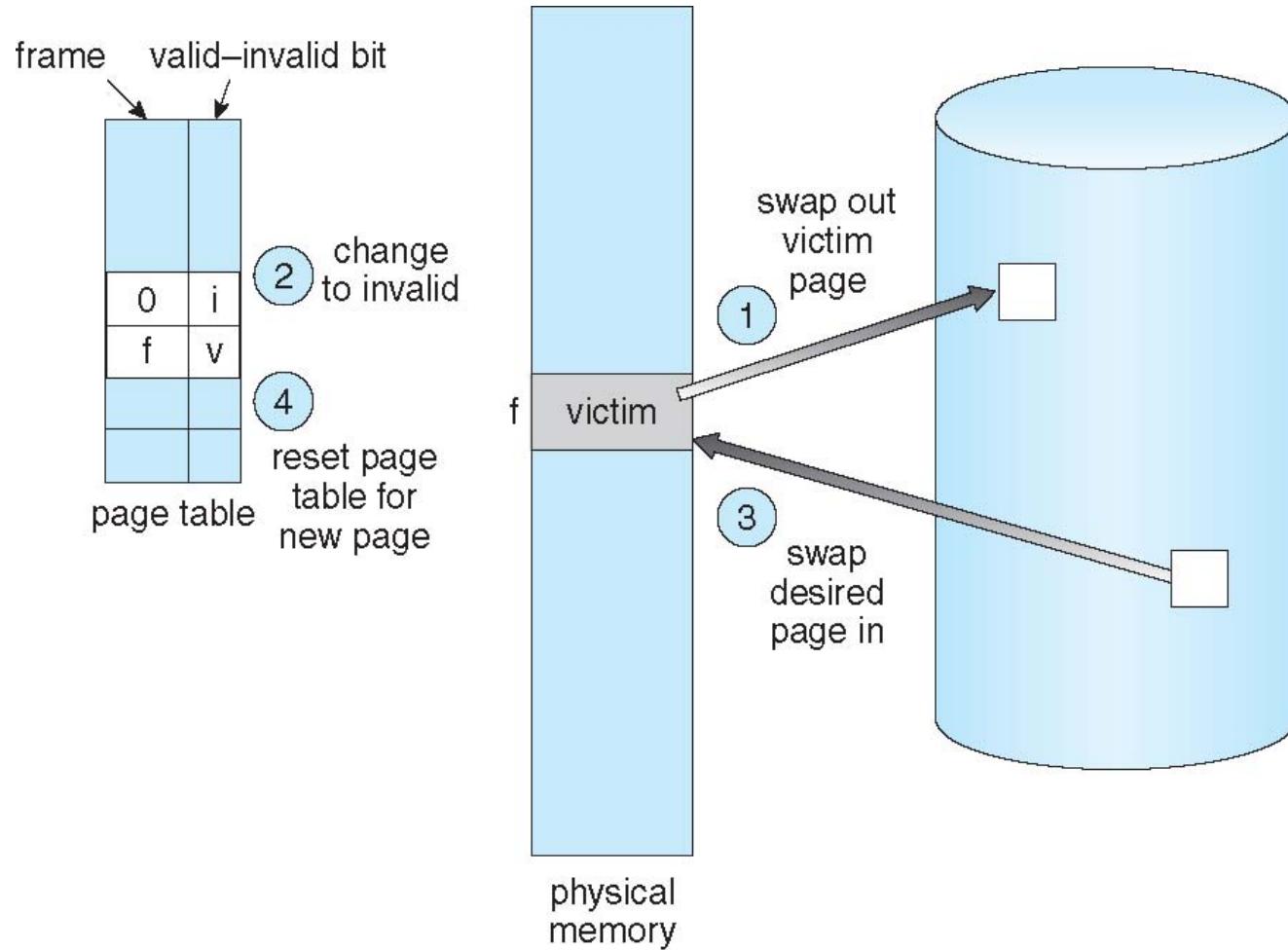
1. Find the location of the desired page on disk
2. Find a free frame:
 - If there is a free frame, use it
 - If there is no free frame, use a page replacement algorithm to select a **victim frame**
 - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the 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 – increasing EAT





Page Replacement





Page and Frame Replacement Algorithms

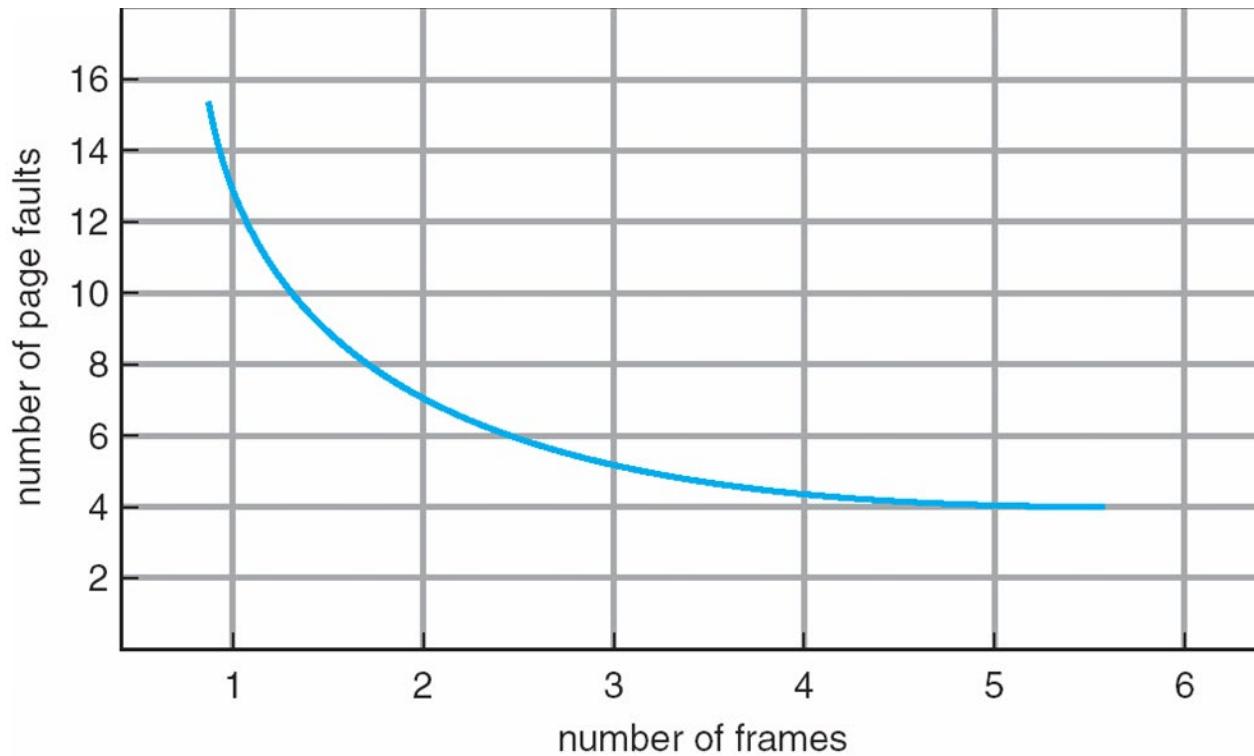
- **Frame-allocation algorithm** determines
 - How many frames to give each process
 - Which frames to replace
- **Page-replacement algorithm**
 - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string)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





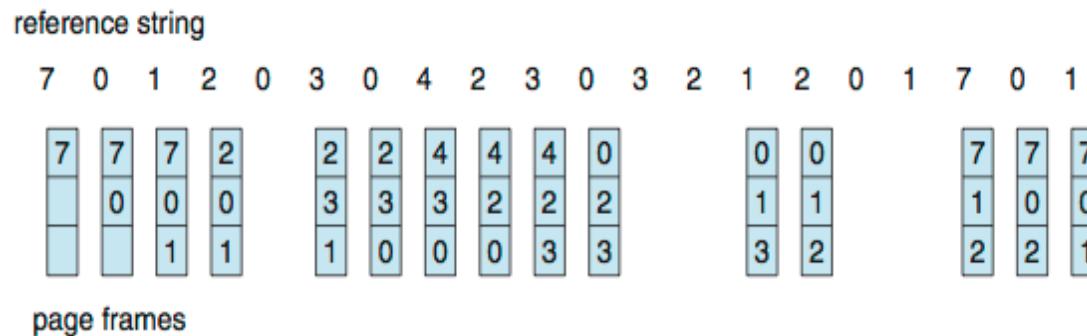
Graph of Page Faults Versus the Number of Frames





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)



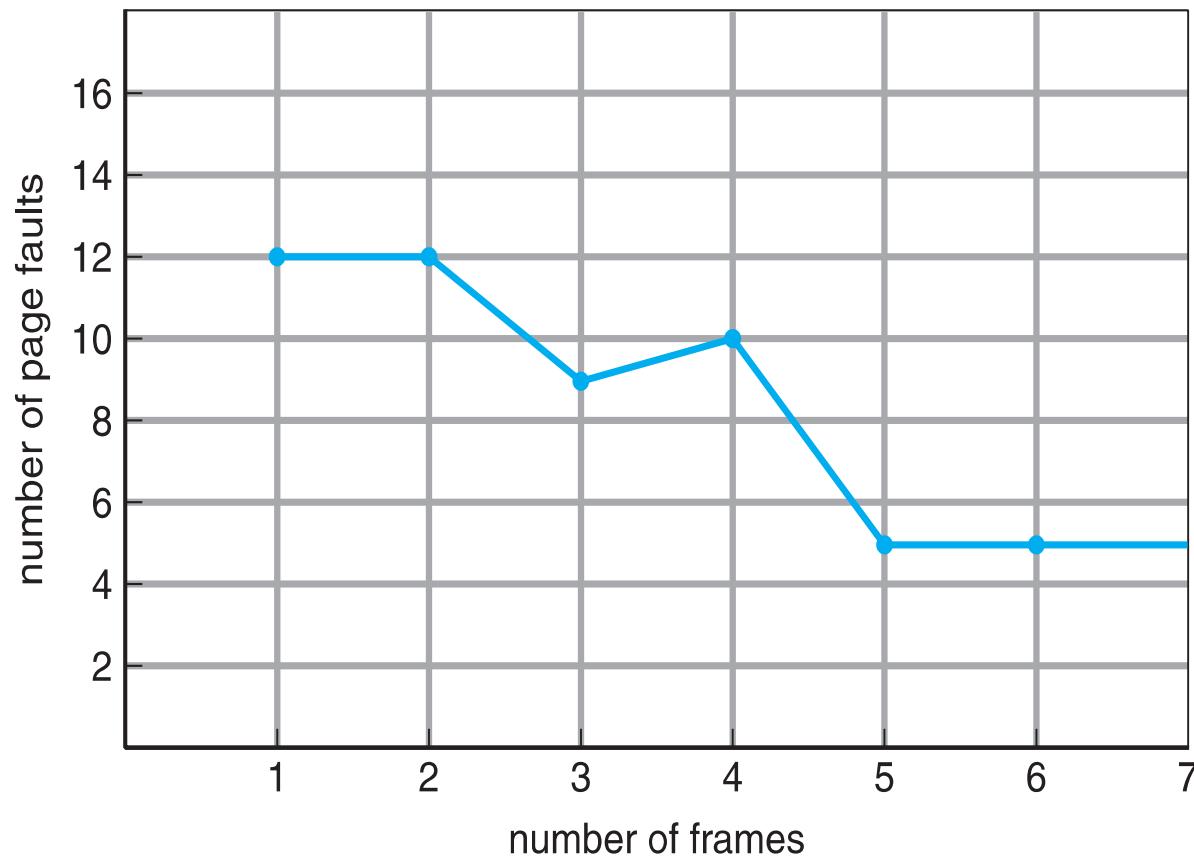
15 page faults

- 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



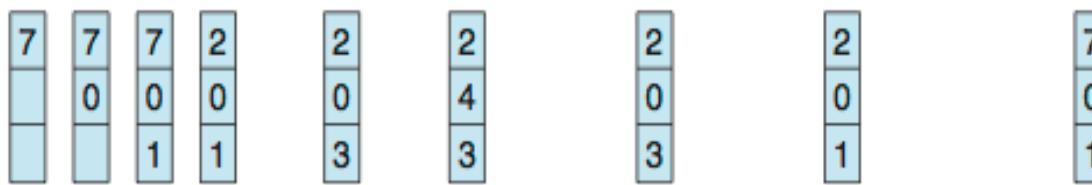


Optimal Algorithm

- 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



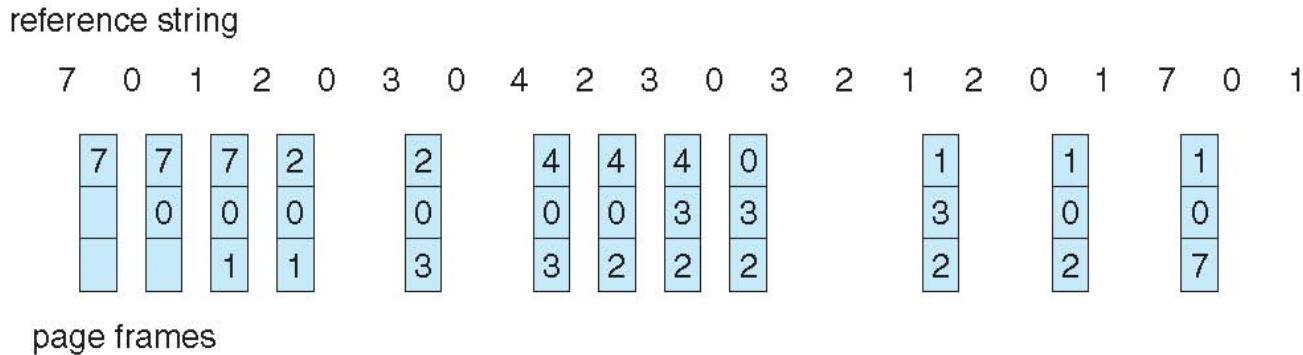
page frames





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



- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?





LRU Algorithm (Cont.)

- 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, look at the 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
 - But each update more expensive
 - No search for replacement

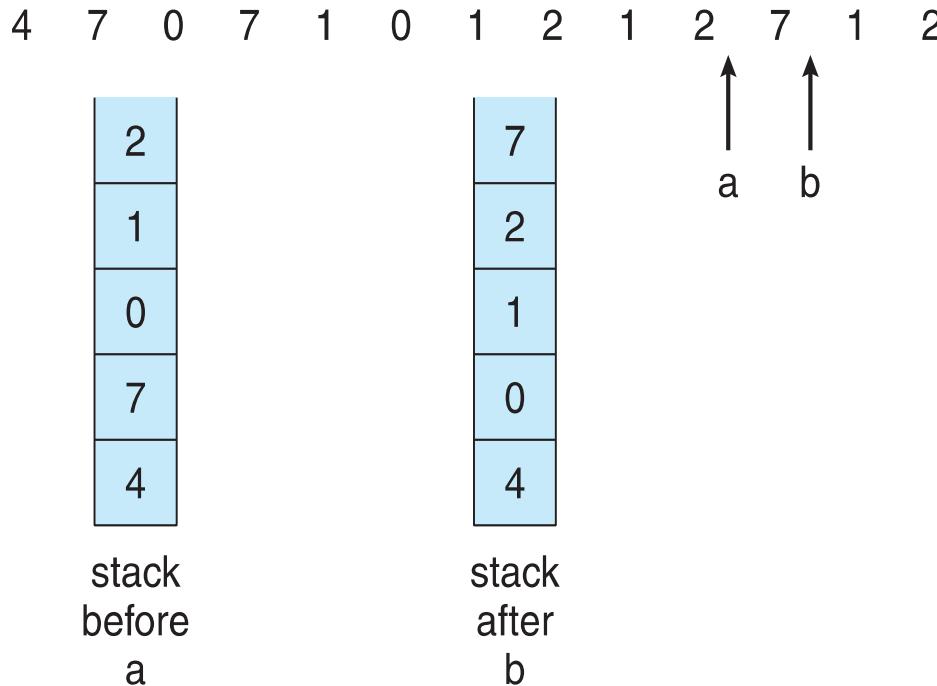




LRU Algorithm (Cont.)

- 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





LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**
 - With each page associate a bit, initially = 0
 - When page is referenced bit set to 1
 - Replace any with reference bit = 0 (if one exists)
 - ▶ We do not know the order, however





LRU Approximation Algorithms (cont.)

■ 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 Algorithm



next
victim



circular queue of pages

(a)





Enhanced Second-Chance 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
- Many 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





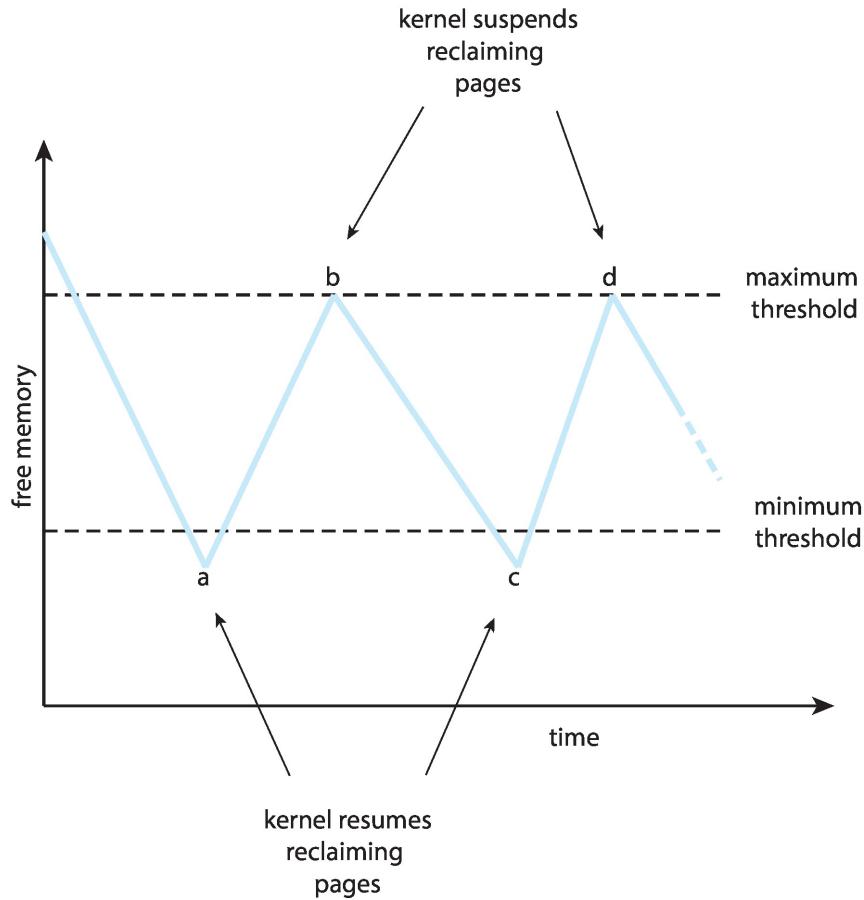
Reclaiming Pages

- A strategy to implement global page-replacement policy
- 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.





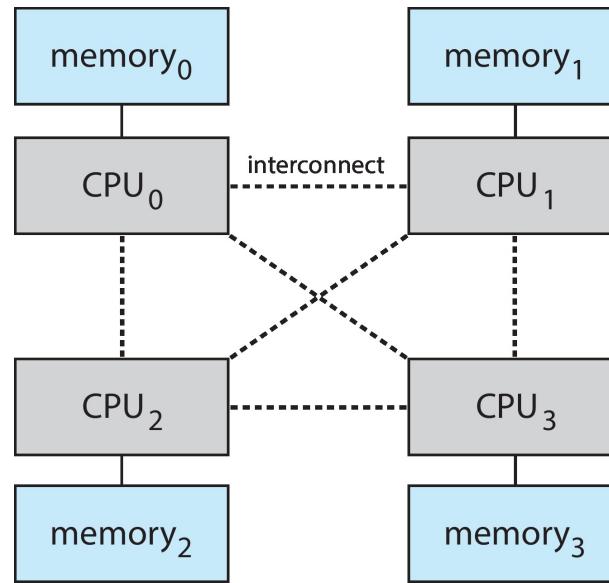
Reclaiming Pages Example





Non-Uniform Memory Access

- So far, we assumed that all memory accessed equally
- Many systems are **NUMA** – speed of access to memory varies
 - 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
 - Solved by Solaris by creating **Igroups**
 - ▶ Structure to track CPU / Memory low latency groups
 - ▶ Used my schedule and pager
 - ▶ When possible schedule all threads of a process and allocate all memory for that process within the Igroup





Thrashing

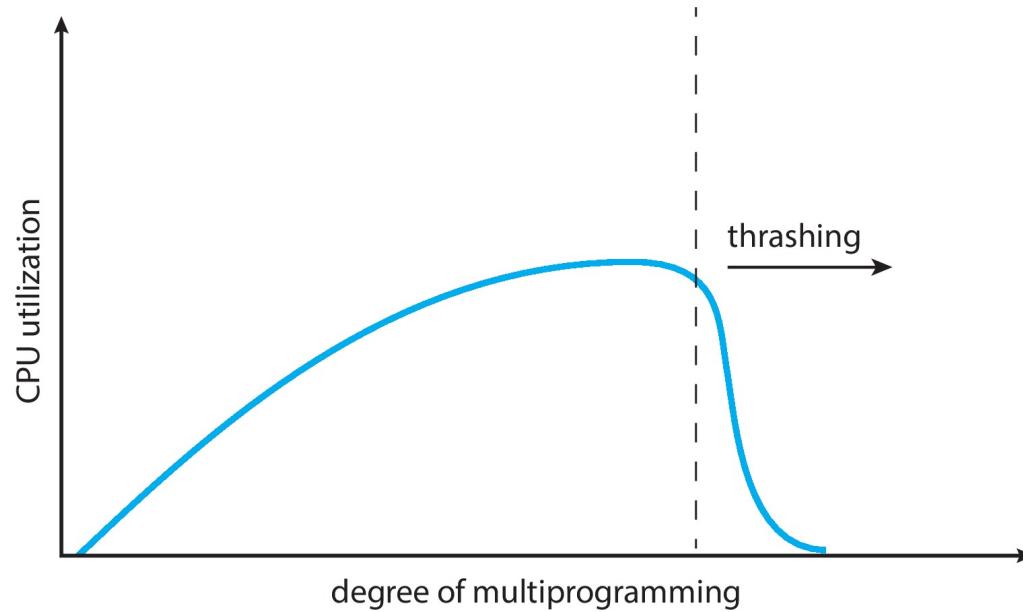
- If a process does not have “enough” pages, the page-fault rate is very high
 - Page fault to get page
 - Replace existing frame
 - But quickly need replaced frame back
 - This leads to:
 - ▶ Low CPU utilization
 - ▶ Operating system thinking that it needs to increase the degree of multiprogramming
 - ▶ 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?

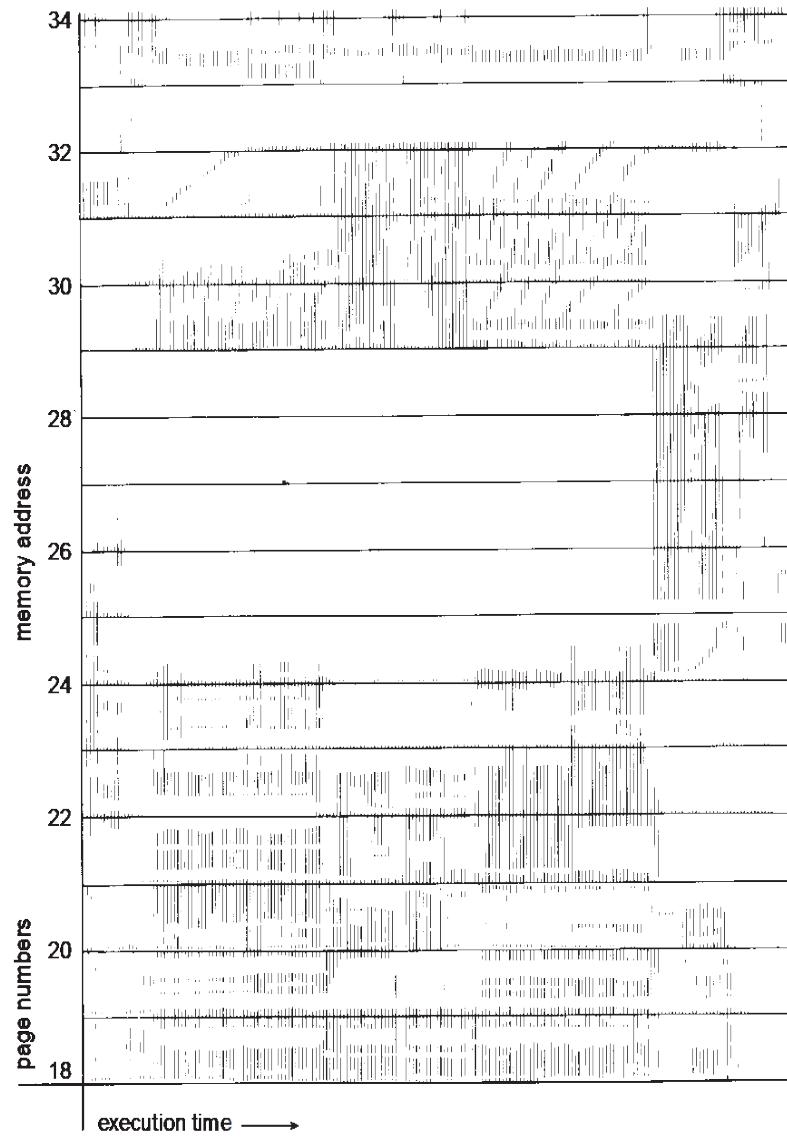
$$\Sigma \text{ size of locality} > \text{total memory size}$$

- 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
Example: 10,000 instructions
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
 - if Δ too small will not encompass entire locality
 - if Δ too large will encompass several localities
 - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \sum WSS_i \equiv$ total demand frames
 - Approximation of locality



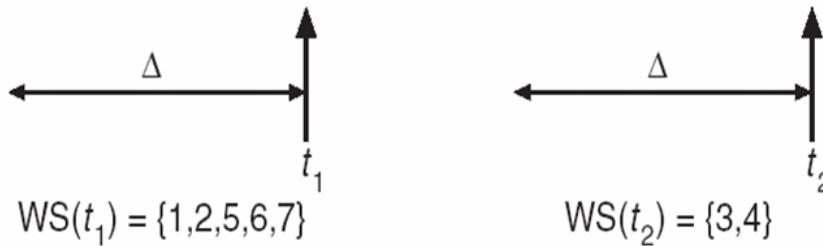


Working-Set Model (Cont.)

- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes

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 ...





Keeping Track of the Working Set

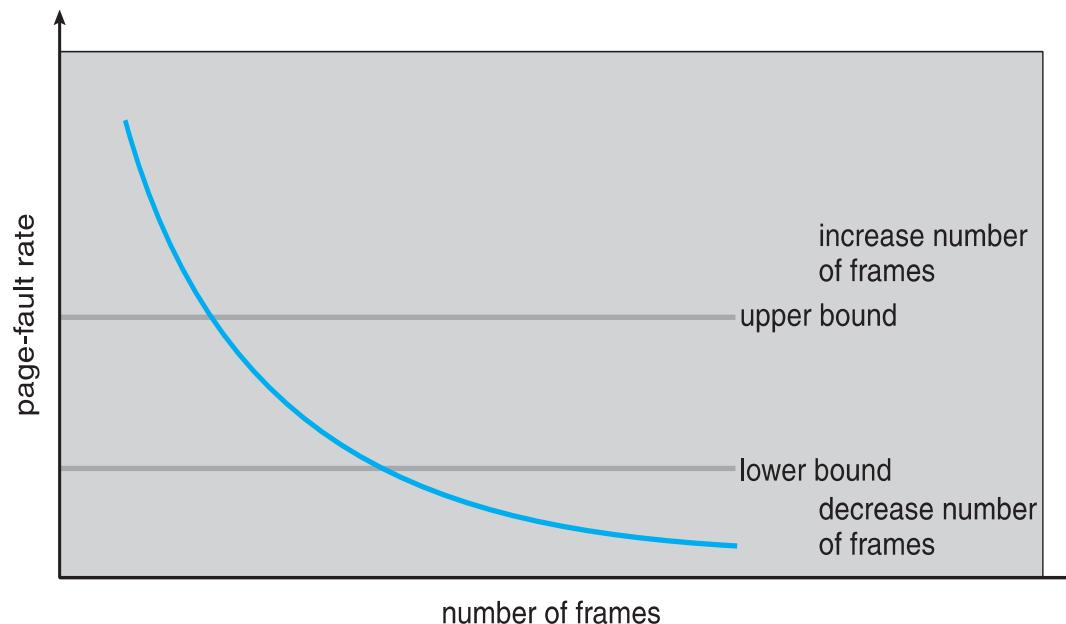
- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
 - Timer interrupts after every 5000 time units
 - Keep in memory 2 bits for each page
 - Whenever a timer interrupts copy and sets the values of all reference bits to 0
 - If one of the bits in memory = 1 \Rightarrow page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units





Page-Fault Frequency

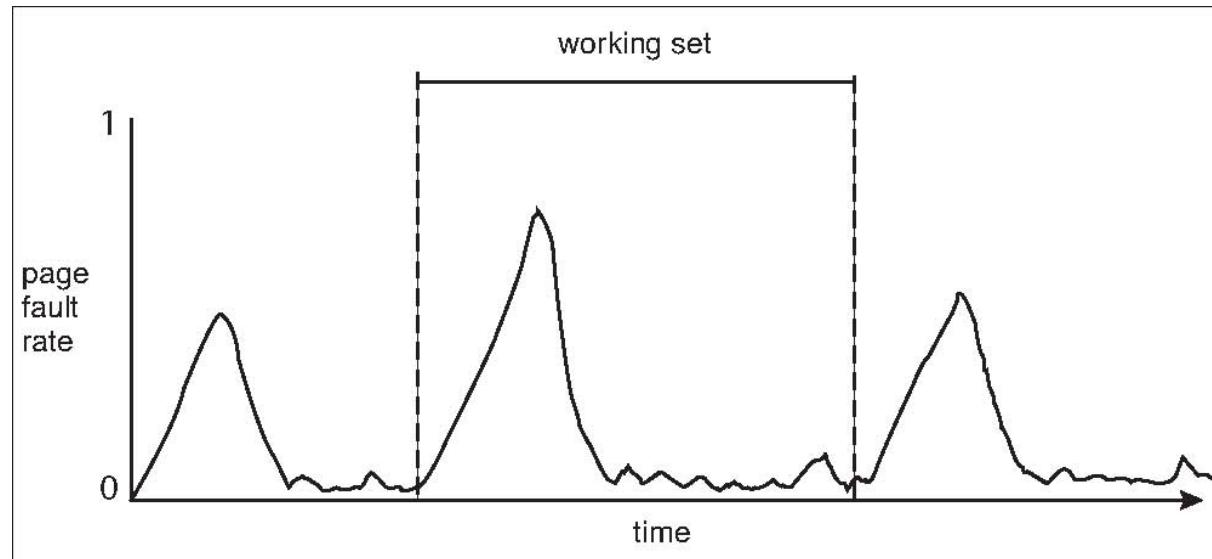
- More direct approach than WSS
- Establish “acceptable” **page-fault frequency (PFF)** rate and use local replacement policy
 - If actual rate too low, process loses frame
 - If actual rate too high, process gains frame





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
 - ▶ i.e., for device I/O





Buddy System

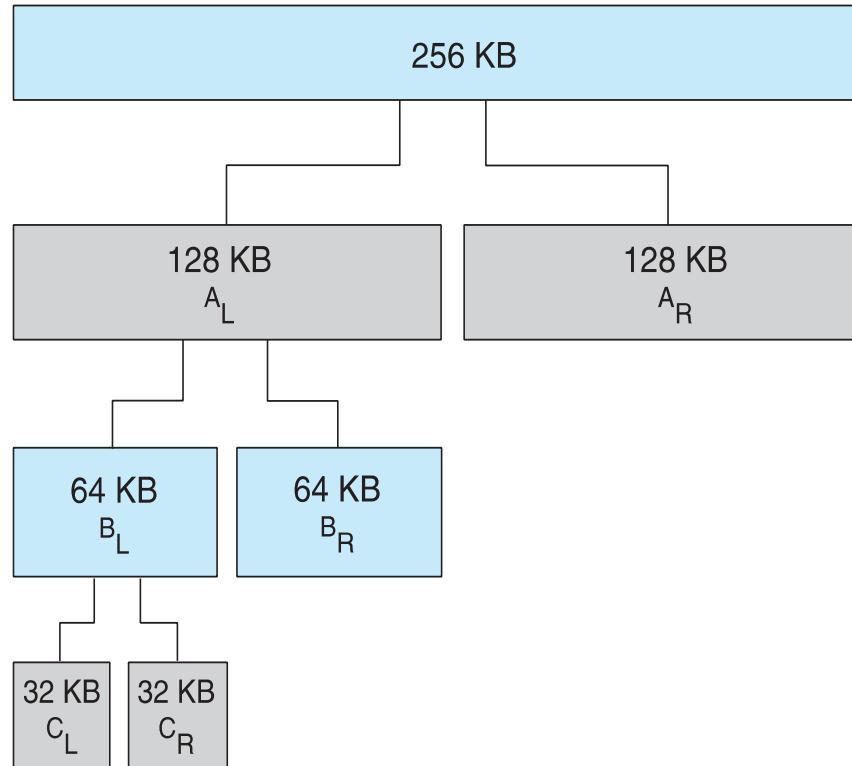
- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
 - Satisfies requests in units sized as power of 2
 - Request rounded up to next highest power of 2
 - When smaller allocation needed than is available, current chunk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





Buddy System Allocator

physically contiguous pages





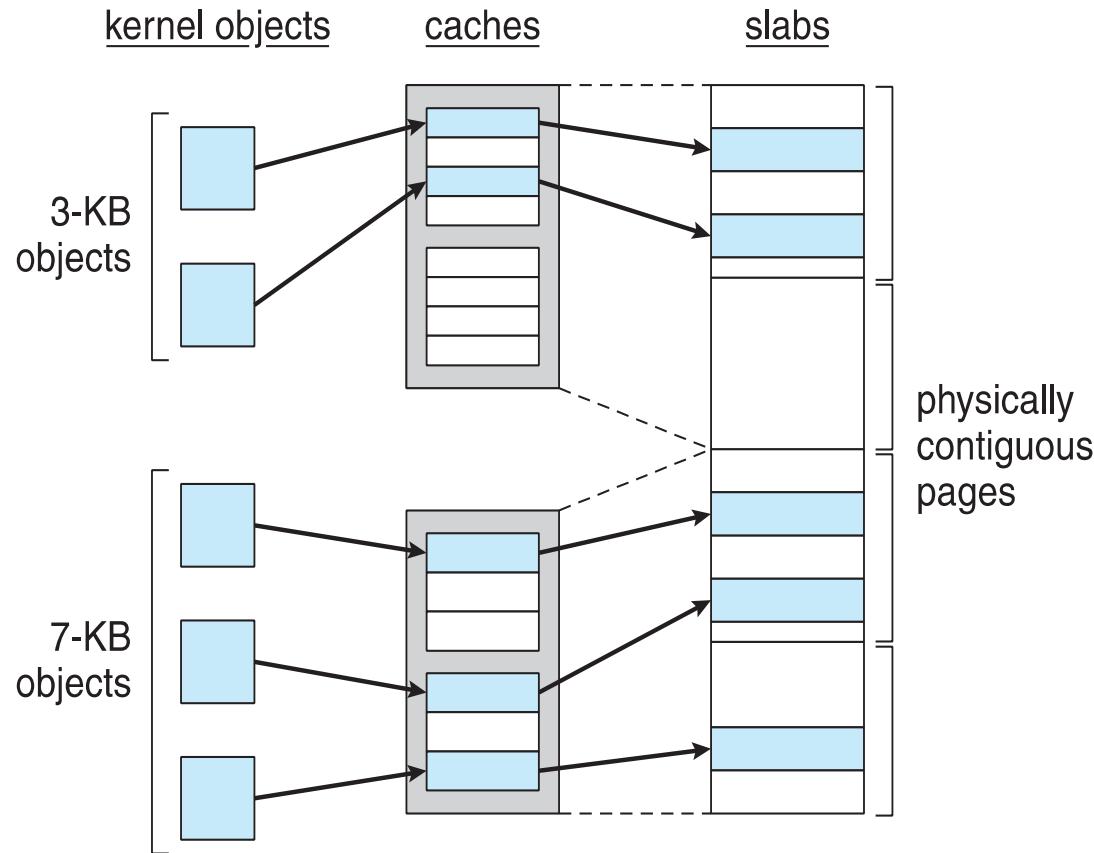
Slab Allocator

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





Slab Allocation





Slab Allocator in Linux

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





Slab Allocator in Linux (Cont.)

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





Other Considerations

- Prepaging
- Page size
- TLB reach
- Inverted page table
- Program structure
- I/O interlock and page locking





Prepaging

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





Page Size

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





TLB Reach

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





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;
```

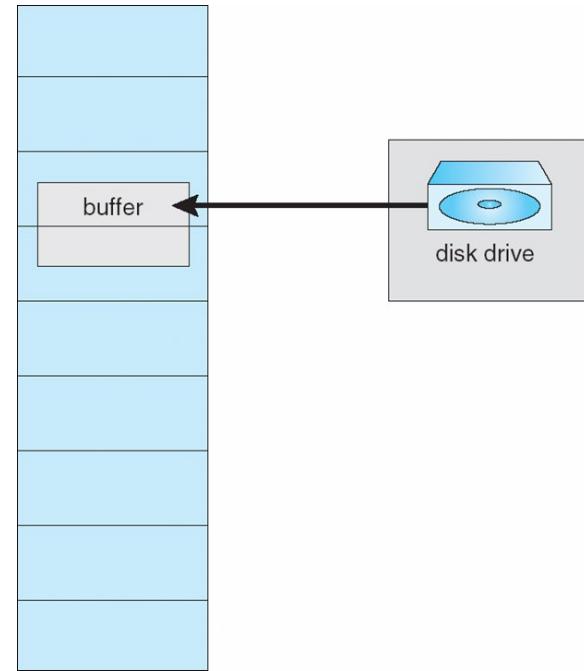
128 page faults





I/O interlock

- **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





Windows

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





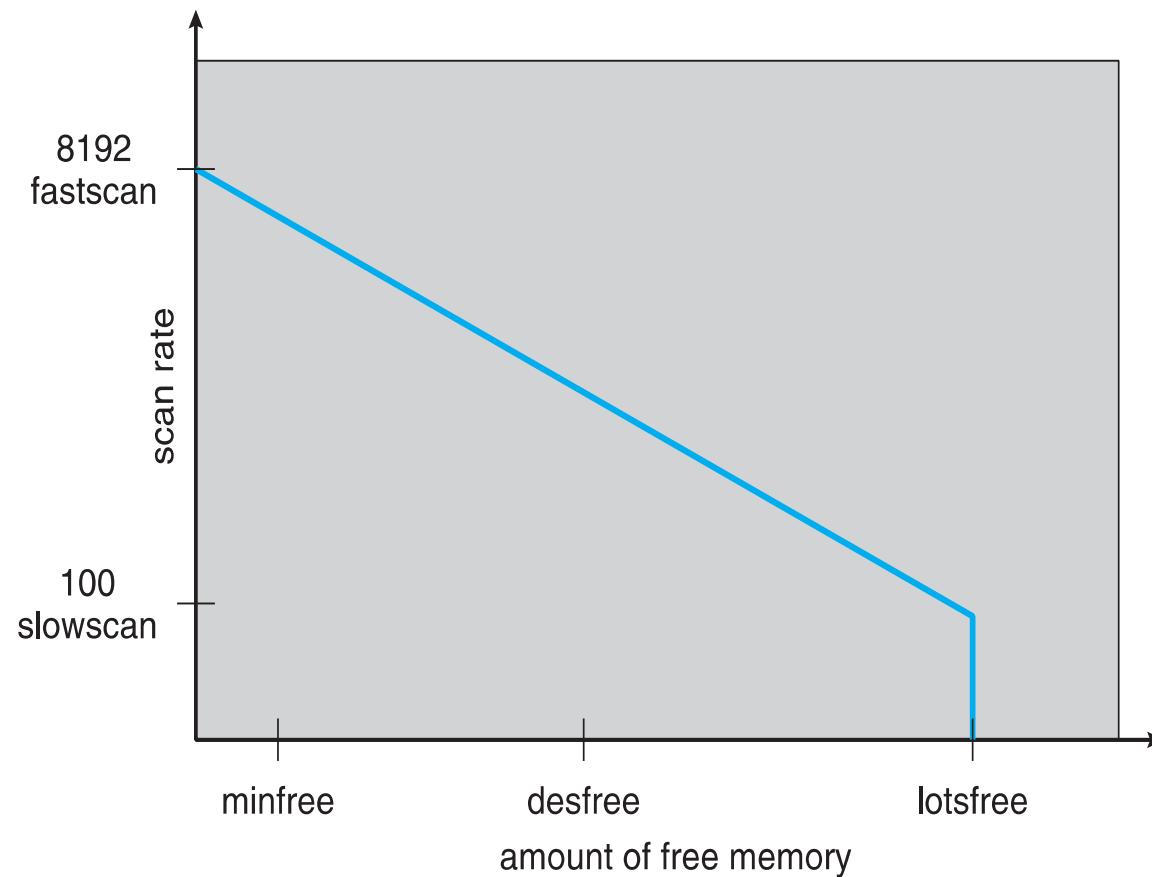
Solaris

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

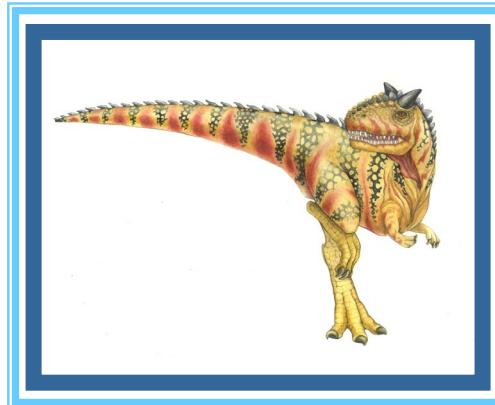




Solaris 2 Page Scanner



End of Chapter 10





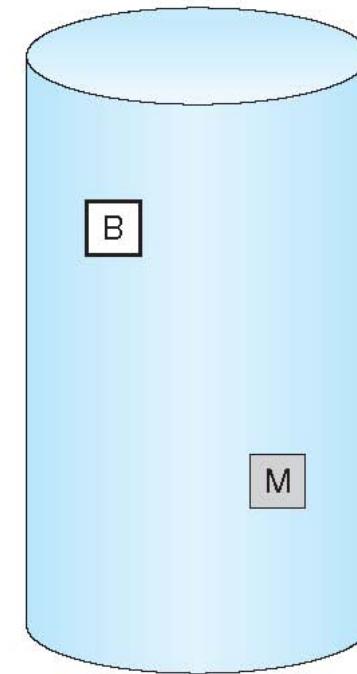
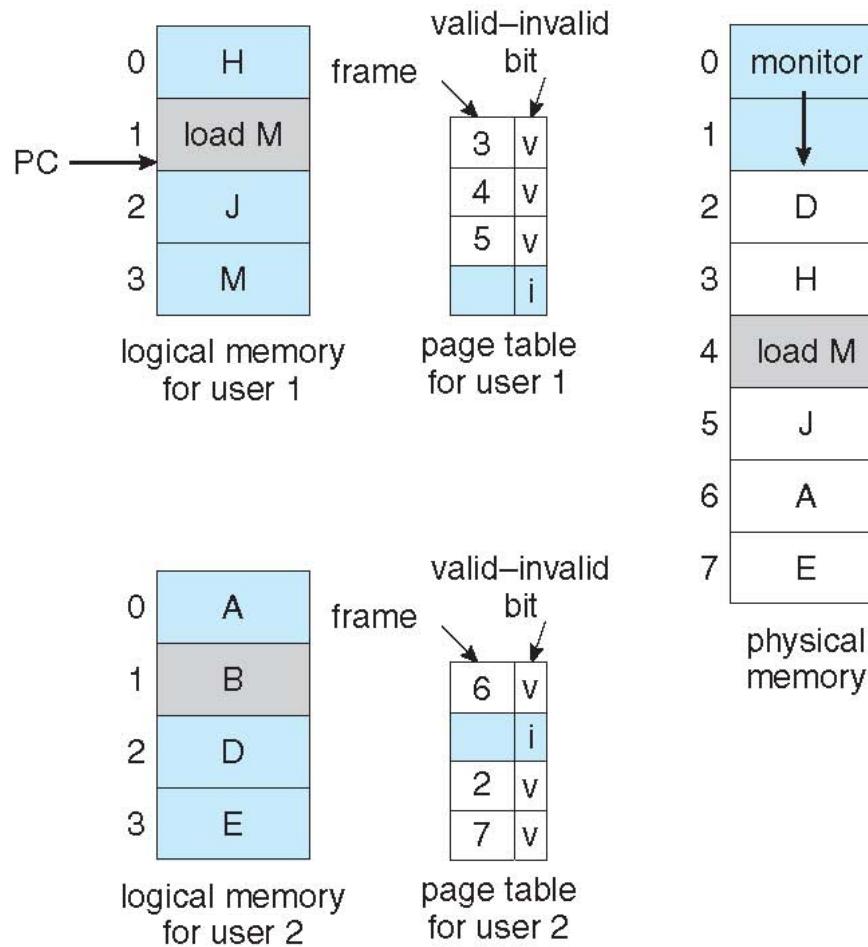
Performance of Demand Paging

- 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

- Use 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





Memory Compression

- **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.
- 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



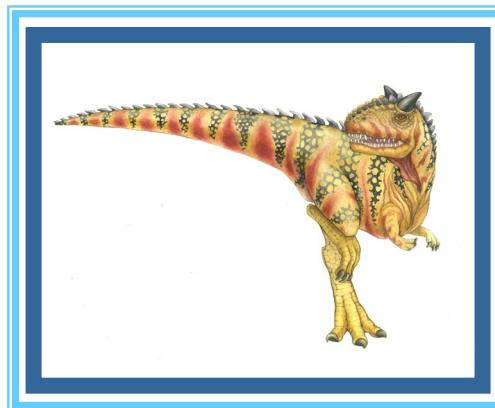
modified frame list



compressed frame list



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





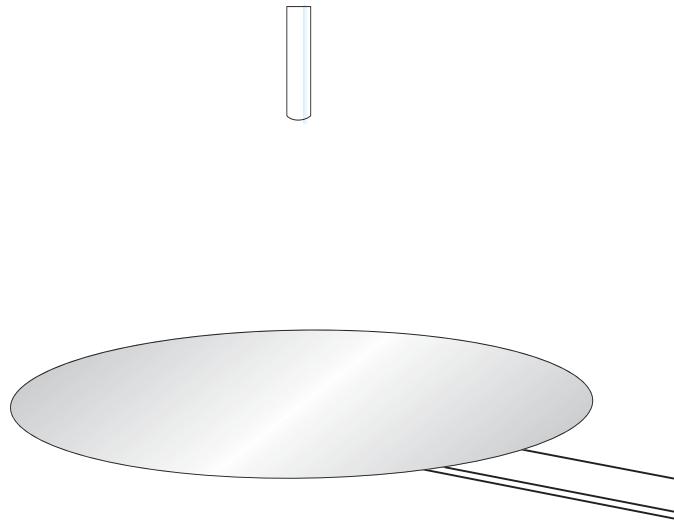
Overview of Mass Storage Structure

- 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
 - Drives rotate 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**)
 - **Head crash** results from disk head making contact with the disk surface -- That's bad
- Disks can be removable





Moving-head Disk Mechanism





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 – 9ms common for desktop drives
 - Average seek time measured or calculated based on 1/3 of tracks
 - Latency based on spindle speed
 - ▶ $1 / (\text{RPM} / 60) = 60 / \text{RPM}$
 - Average latency = $\frac{1}{2}$ latency





Hard Disk Performance

- **Access Latency = Average access time = average seek time + average latency**
 - For fastest disk $3\text{ms} + 2\text{ms} = 5\text{ms}$
 - For slow disk $9\text{ms} + 5.56\text{ms} = 14.56\text{ms}$
- 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 .1ms controller overhead =
 - $5\text{ms} + 4.17\text{ms} + 0.1\text{ms} + \text{transfer time} =$
 - Transfer time = $4\text{KB} / 1\text{Gb/s} * 8\text{Gb} / \text{GB} * 1\text{GB} / 1024^2\text{KB} = 32 / (1024^2) = 0.031 \text{ ms}$
 - Average I/O time for 4KB block = $9.27\text{ms} + .031\text{ms} = 9.301\text{ms}$





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
- Can be more reliable than HDDs
- More expensive per MB
- Maybe have shorter life span – need careful management
- Less capacity
- But much faster
- Busses can be too slow -> connect directly to PCI for example
- No moving parts, so no seek time or rotational latency





Nonvolatile Memory Devices

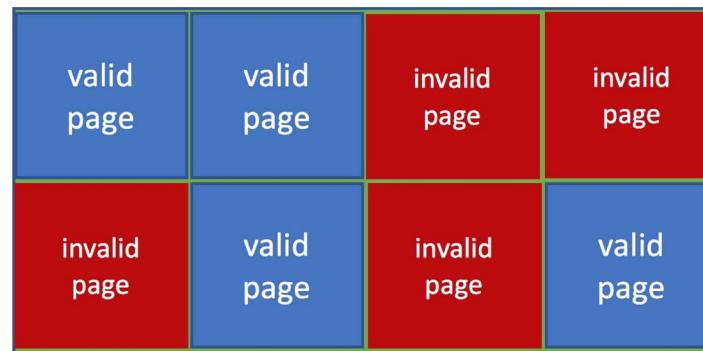
- Have characteristics that present challenges
- Read and written in “page” increments (think sector) but can’t overwrite in place
 - Must first be erased, and erases happen in larger “block” increments
 - Can only be erased a limited number of times before worn out –
~ 100,000
 - Life span measured in **drive writes per day (DWPD)**
 - ▶ 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** 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
- Computers have buffering, caching via RAM, so why RAM drives?
 - Caches / buffers allocated / managed by programmer, operating system, hardware
 - RAM drives under user control
 - Found in all major operating systems
 - ▶ Linux /dev/ram, macOS diskutil to create them, Linux /tmp of file system type tmpfs
- Used as high speed temporary storage
 - Programs could share bulk data, 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)**.
- Most common is SATA
- 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
- Minimize seek time
- 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





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 LBAs, handle sorting of requests
 - ▶ Some of the algorithms they use described next





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
- We illustrate scheduling algorithms with a request queue (0-199)

98, 183, 37, 122, 14, 124, 65, 67

Head pointer 53



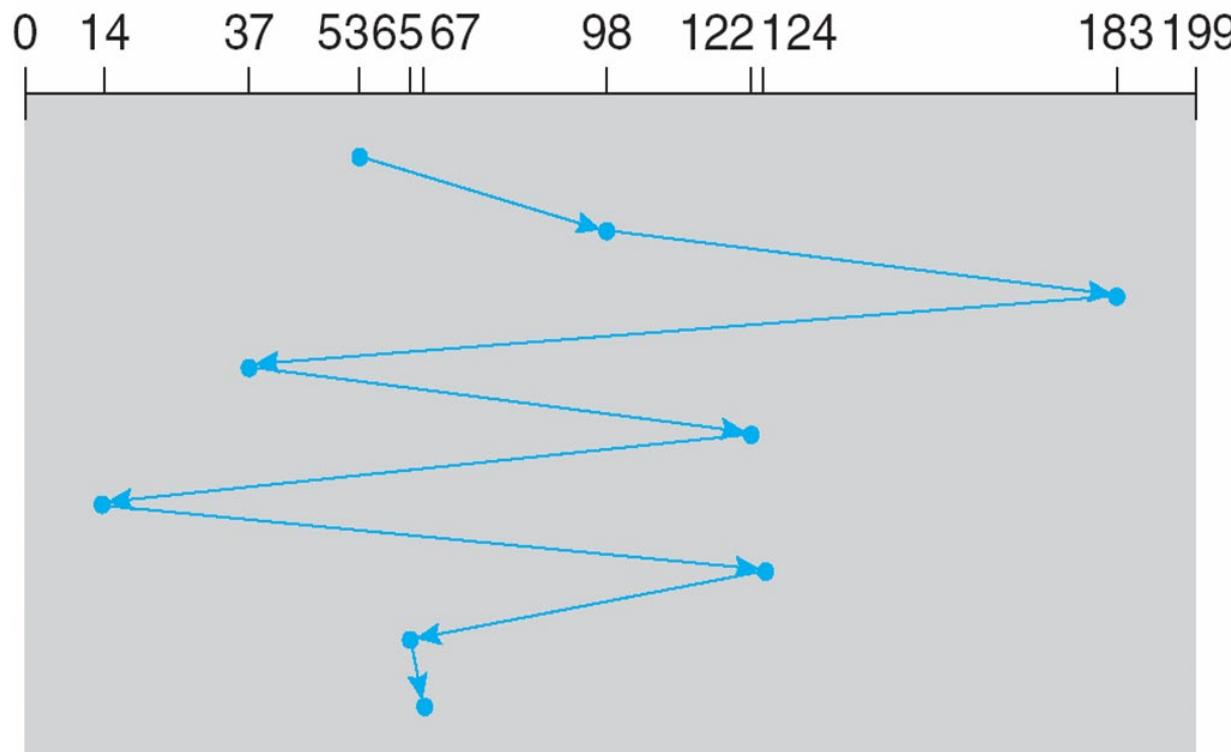


FCFS

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**
- Illustration shows total head movement of 208 cylinders
- But note that if requests are uniformly dense, largest density at other end of disk and those wait the longest

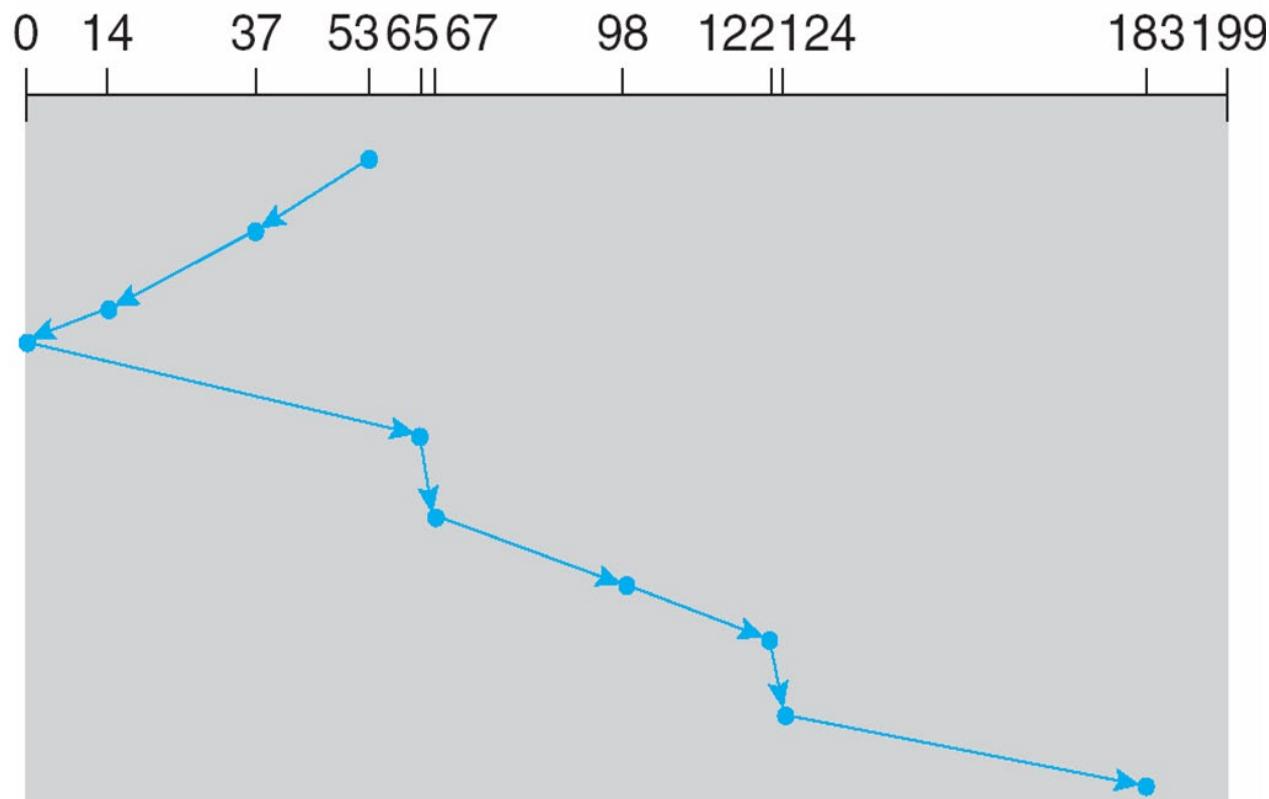




SCAN (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

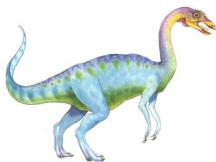




C-SCAN

- 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
- Total number of cylinders?

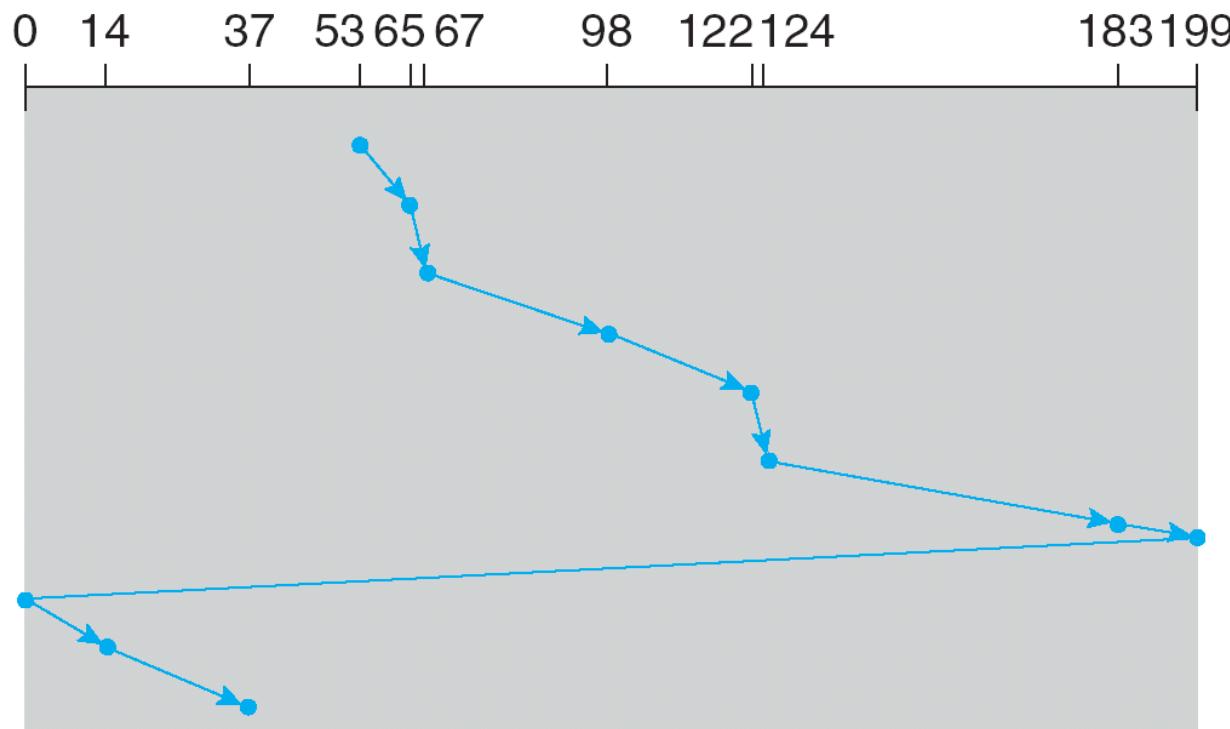




C-SCAN (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53





Selecting a Disk-Scheduling Algorithm

- SSTF 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, essentially 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 also **NOOP** and **completely fair queueing** scheduler (**CFQ**) 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.)

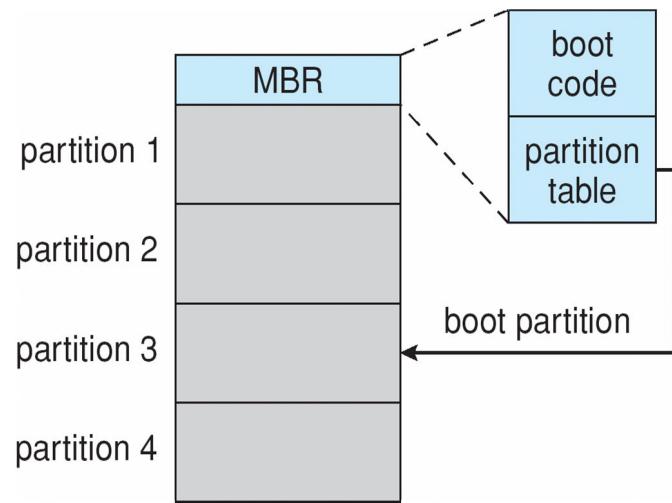
- **Root partition** contains the OS, other partitions can hold other Oses, other file systems, or be raw
 - **Mounted** at boot time
 - Other partitions can mount automatically or manually
- At mount time, file system consistency checked
 - Is all metadata correct?
 - ▶ If not, fix it, try again
 - ▶ 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



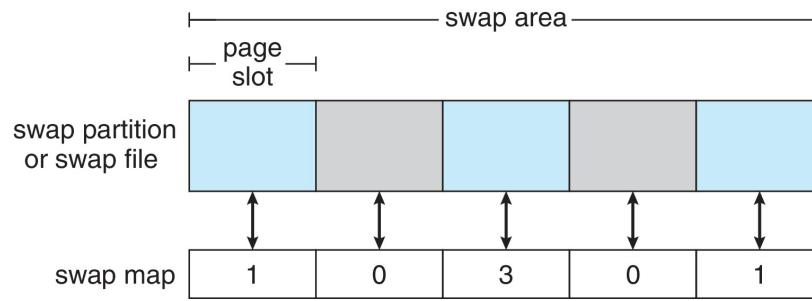
Booting from secondary storage in Windows





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

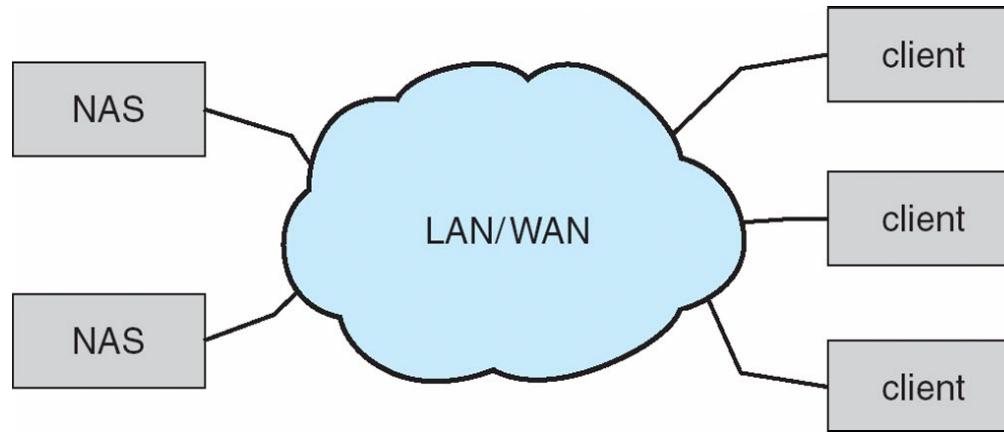
- Computers access storage in three ways
 - host-attached
 - network-attached
 - cloud
- Host attached access through local I/O ports, using one of several technologies
 - To attach many devices, use storage busses such as USB, firewire, thunderbolt
 - High-end systems use **fibre channel (FC)**
 - ▶ 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

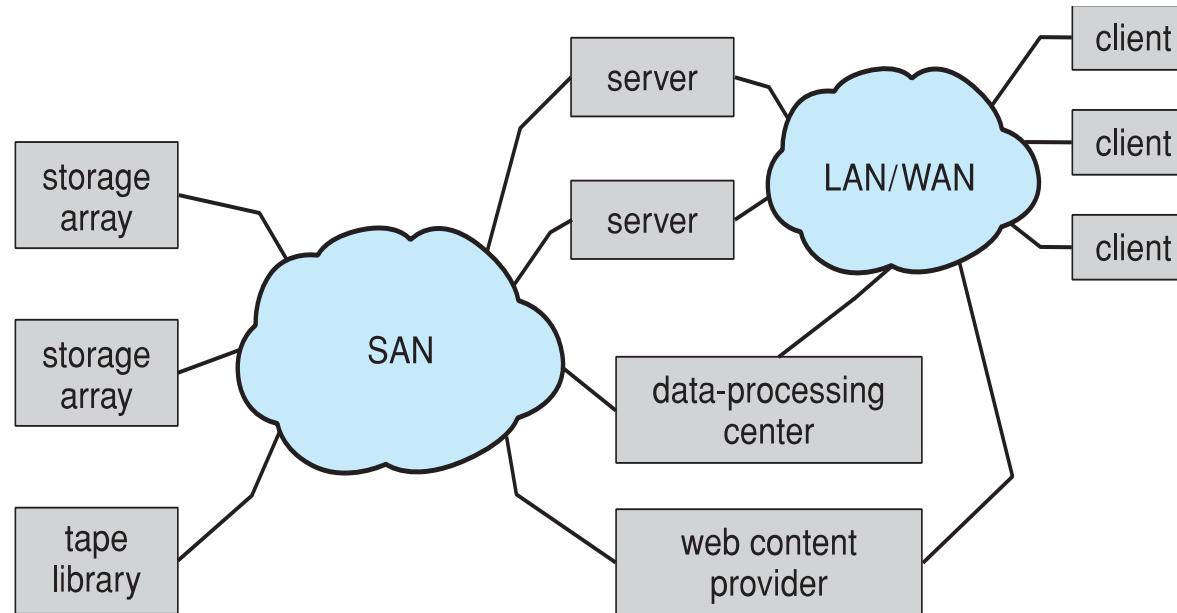
- Can just attach disks, or arrays of disks
- Avoids the NAS drawback of using network bandwidth
- Storage Array has controller(s), provides features to attached host(s)
 - Ports to connect hosts to array
 - Memory, controlling software (sometimes NVRAM, etc)
 - A few to thousands of disks
 - RAID, hot 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



A Storage Array





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 $100,000^2 / (2 * 10) = 500 * 10^6$ 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





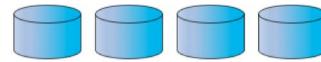
RAID (Cont.)

- 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





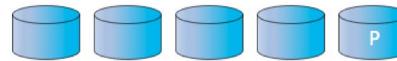
RAID Levels



(a) RAID 0: non-redundant striping.



(b) RAID 1: mirrored disks.



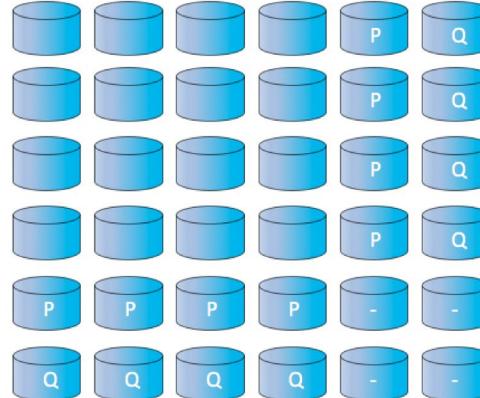
(c) RAID 4: block-interleaved parity.



(d) RAID 5: block-interleaved distributed parity.



(e) RAID 6: P + Q redundancy.

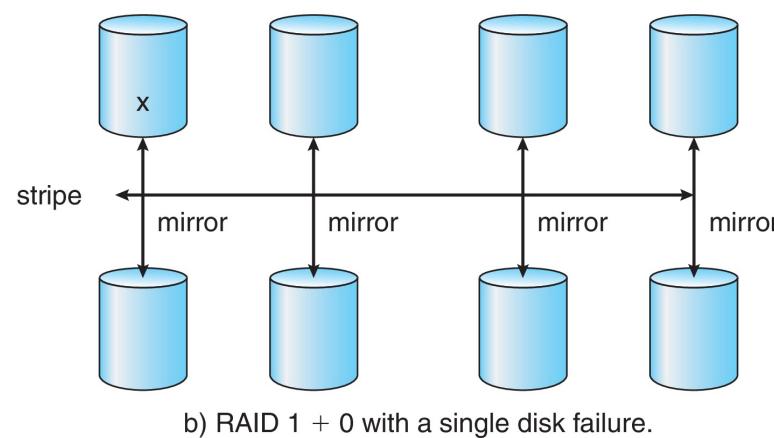
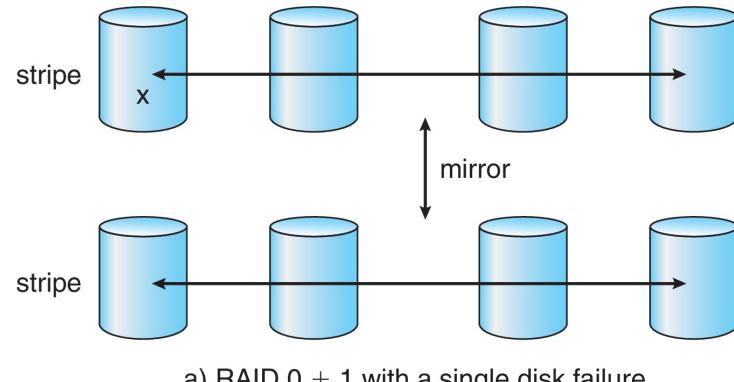


(f) Multidimensional RAID 6.





RAID (0 + 1) and (1 + 0)





Other Features

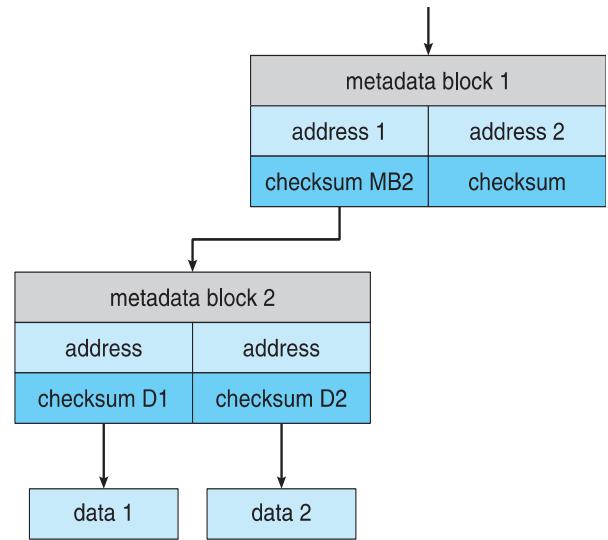
- Regardless of where RAID implemented, other useful features can be added
- **Snapshot** is a view of file system before a set of changes take place (i.e. at a point in time)
 - More in Ch 12
- Replication is automatic duplication of writes between separate sites
 - For redundancy and disaster recovery
 - 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





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



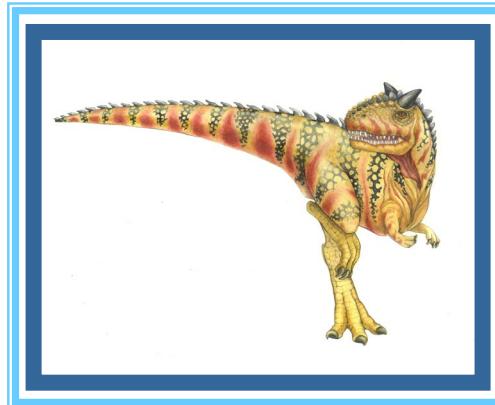


Object Storage (Cont.)

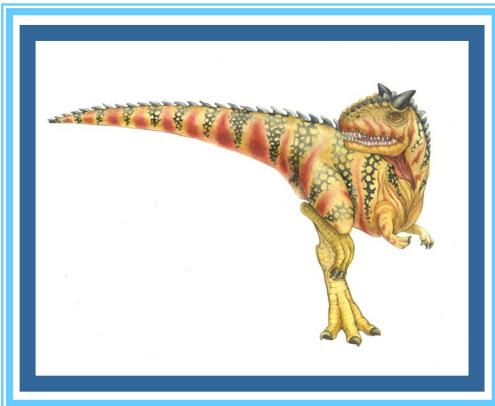
- Object storage management software like **Hadoop file system (HDFS)** and **Ceph** determine where to store objects, manages protection
 - Typically by storing N copies, across N systems, in the object storage cluster
 - **Horizontally scalable**
 - **Content addressable, unstructured**



End of Chapter 11



Chapter 12: I/O Systems





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





I/O Hardware

- 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, PCI Express (**PCle**)
 - ▶ **expansion bus** connects relatively slow devices
 - ▶ **Serial-attached SCSI (SAS)** common disk interface





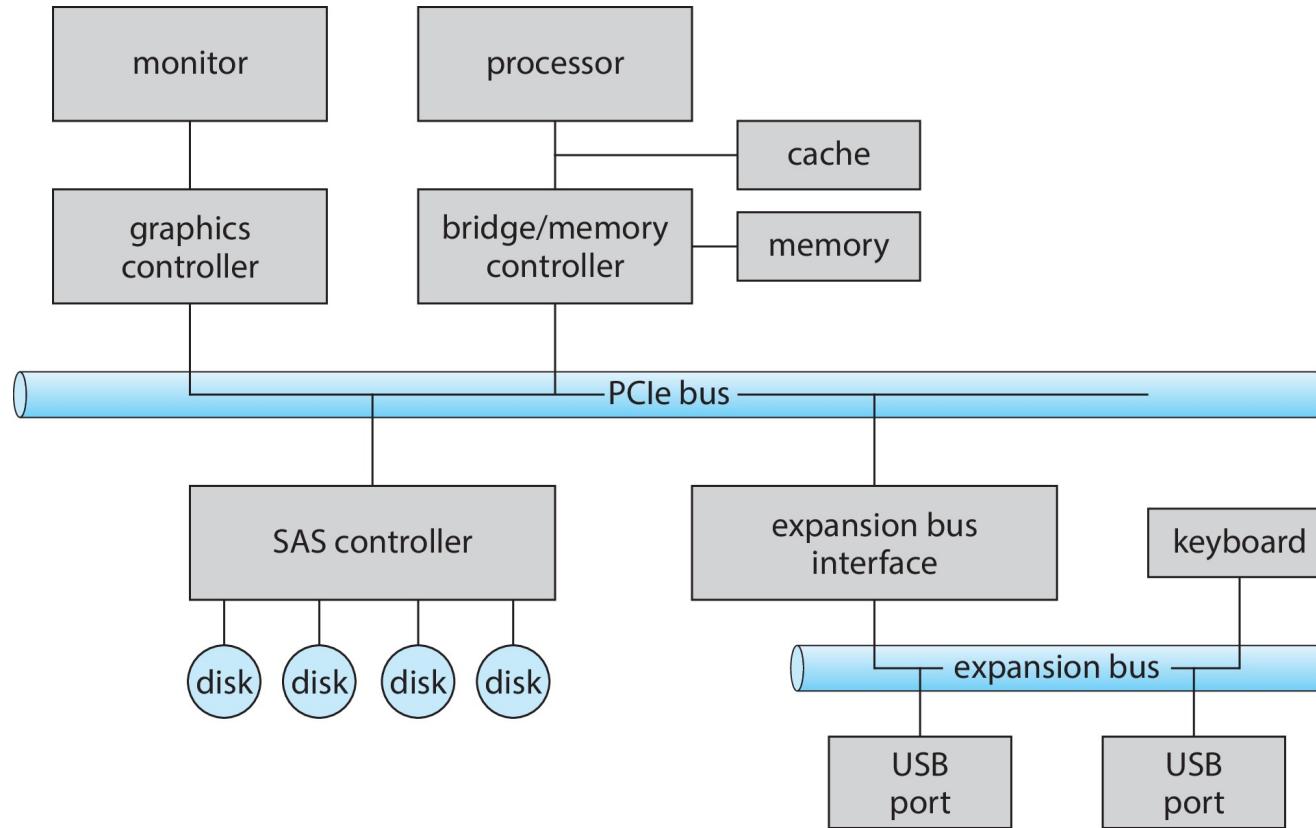
I/O Hardware (Cont.)

- **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, memory, etc.





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 1-4 bytes, or FIFO buffer





I/O Hardware (Cont.)

- 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 address spaces (graphics)





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
 1. Read busy bit from status register until 0
 2. Host sets read or write bit and if write copies data into data-out register
 3. Host sets command-ready bit
 4. Controller sets busy bit, executes transfer
 5. Controller clears busy bit, error bit, command-ready bit when transfer 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 overwritten / lost





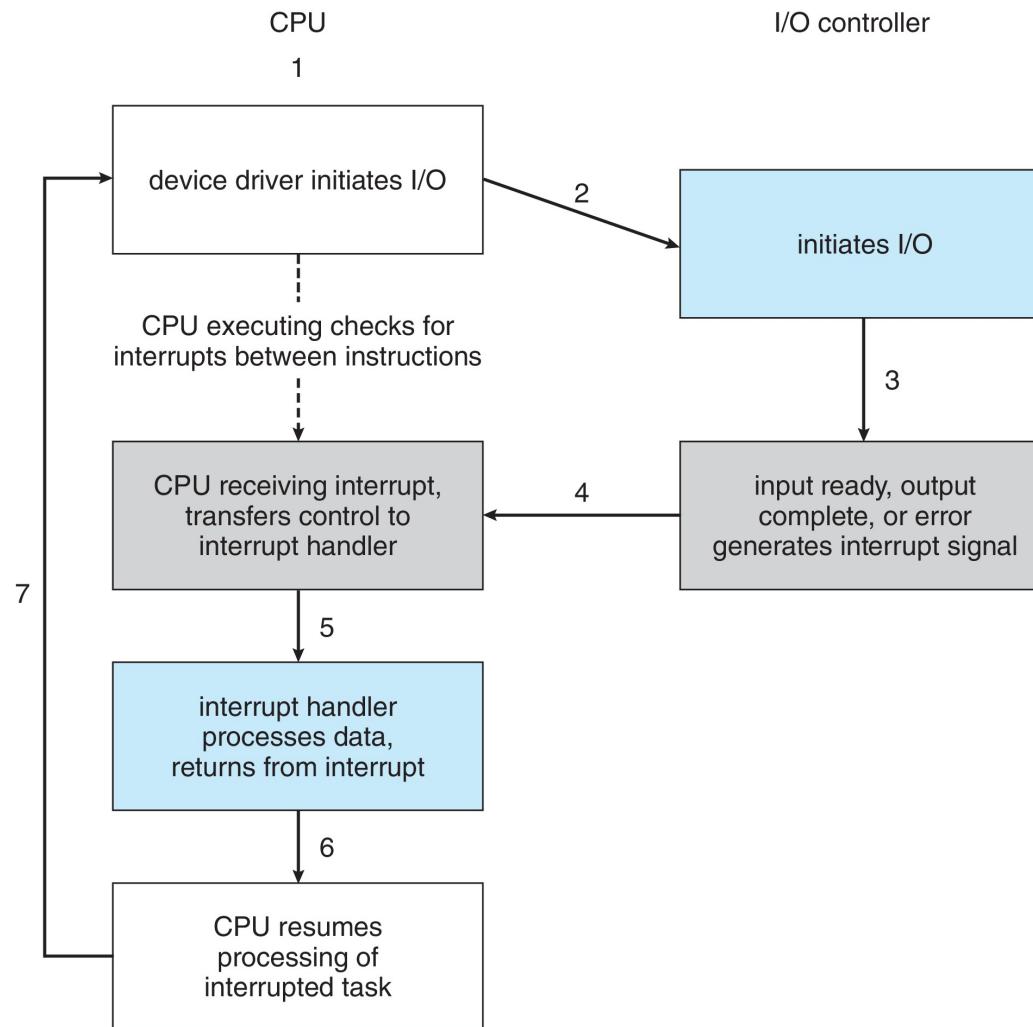
Interrupts

- 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 processor 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

		0:00:10
		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





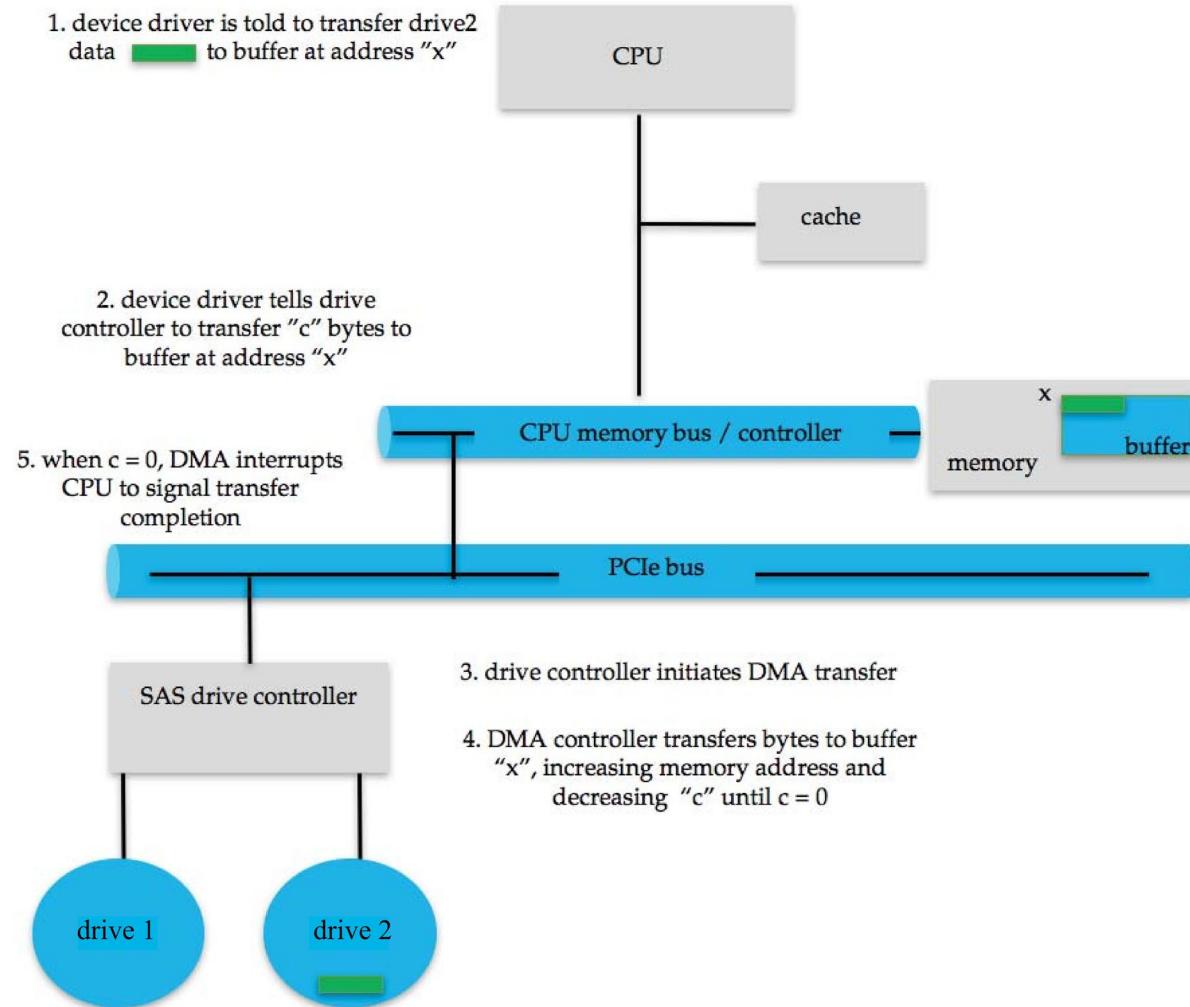
Direct Memory Access

- Used to avoid **programmed I/O** (one 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
 - 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 - **DVMA**





Six Step Process to Perform DMA Transfer





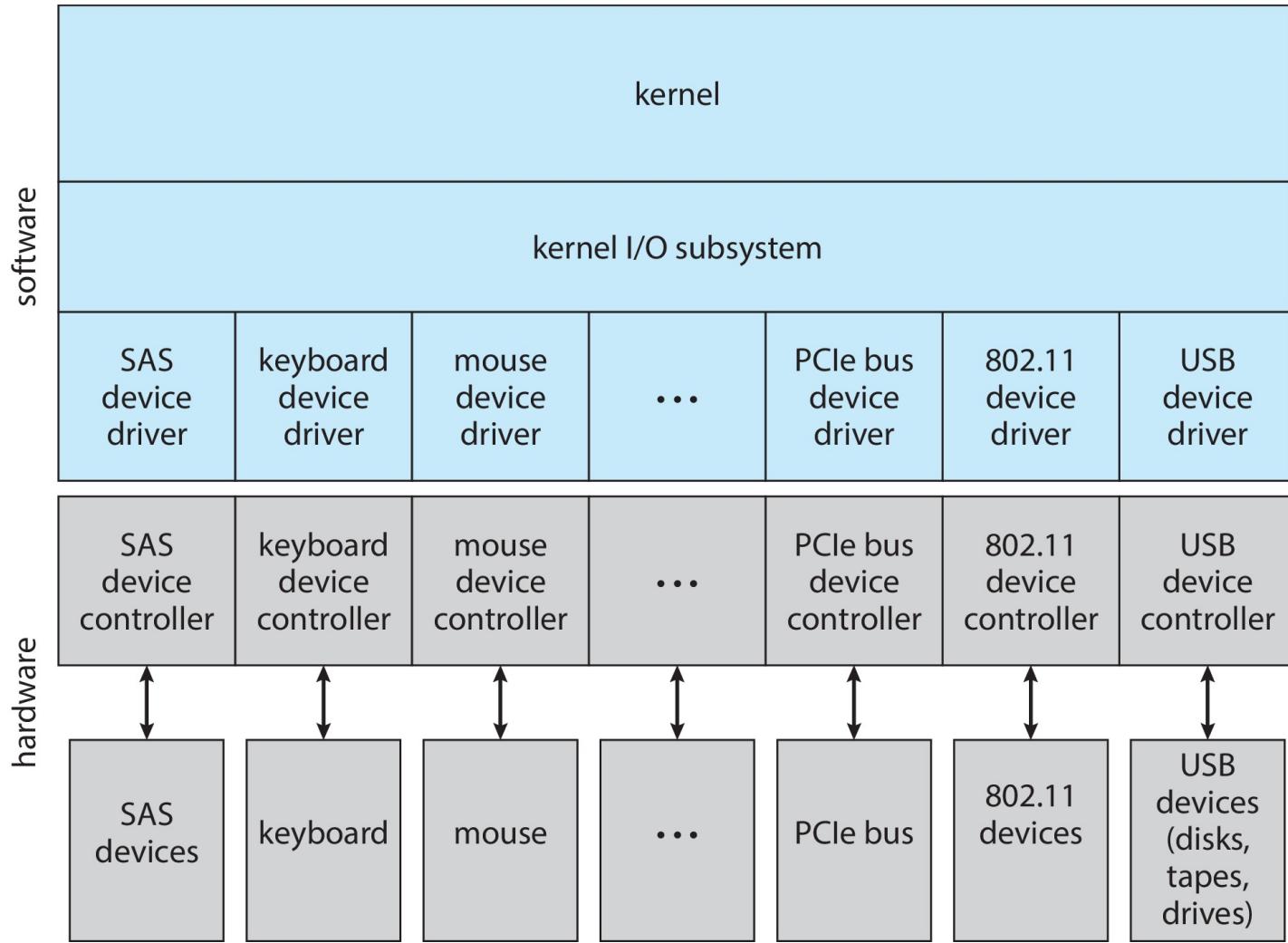
Application I/O Interface

- 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 block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read-write	CD-ROM graphics controller 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





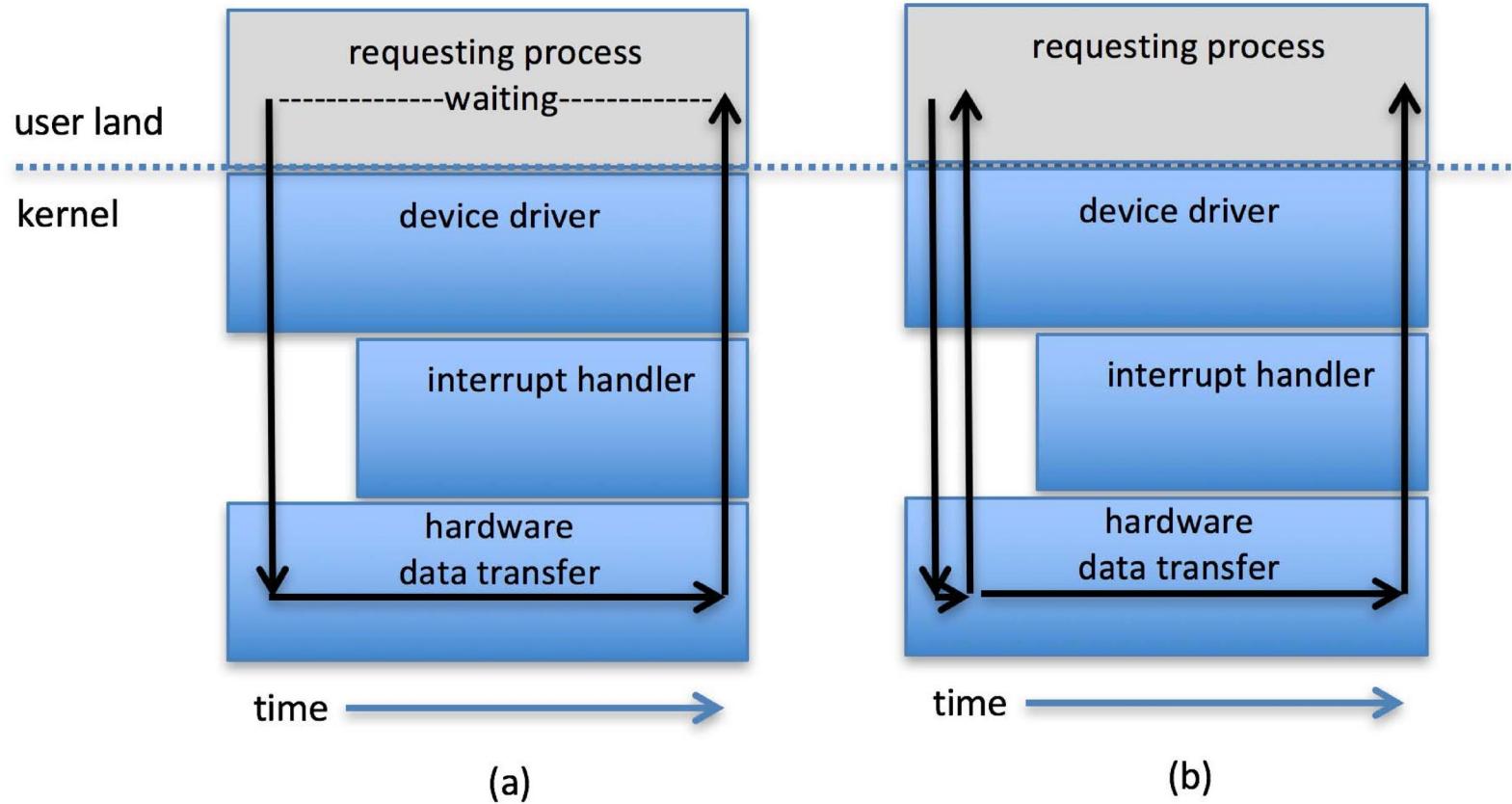
Nonblocking and Asynchronous I/O

- **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





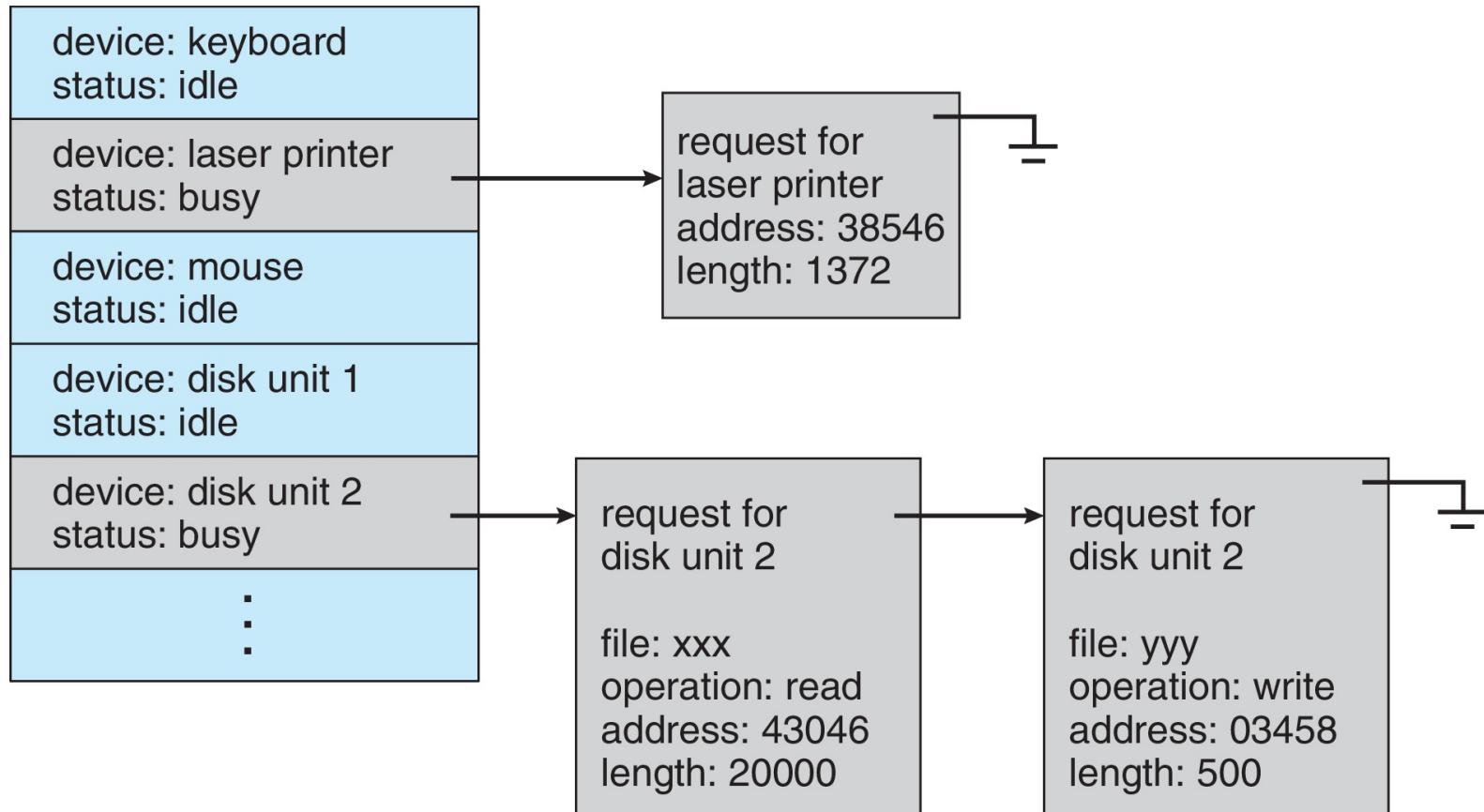
Kernel I/O Subsystem

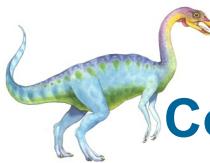
- Scheduling
 - Some I/O request ordering via per-device queue
 - Some OSs try fairness
 - Some implement Quality Of Service (i.e. IPQOS)
- **Buffering** - store data in memory while transferring between devices
 - To cope with device speed mismatch
 - To cope with device transfer size mismatch
 - To 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



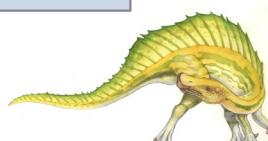
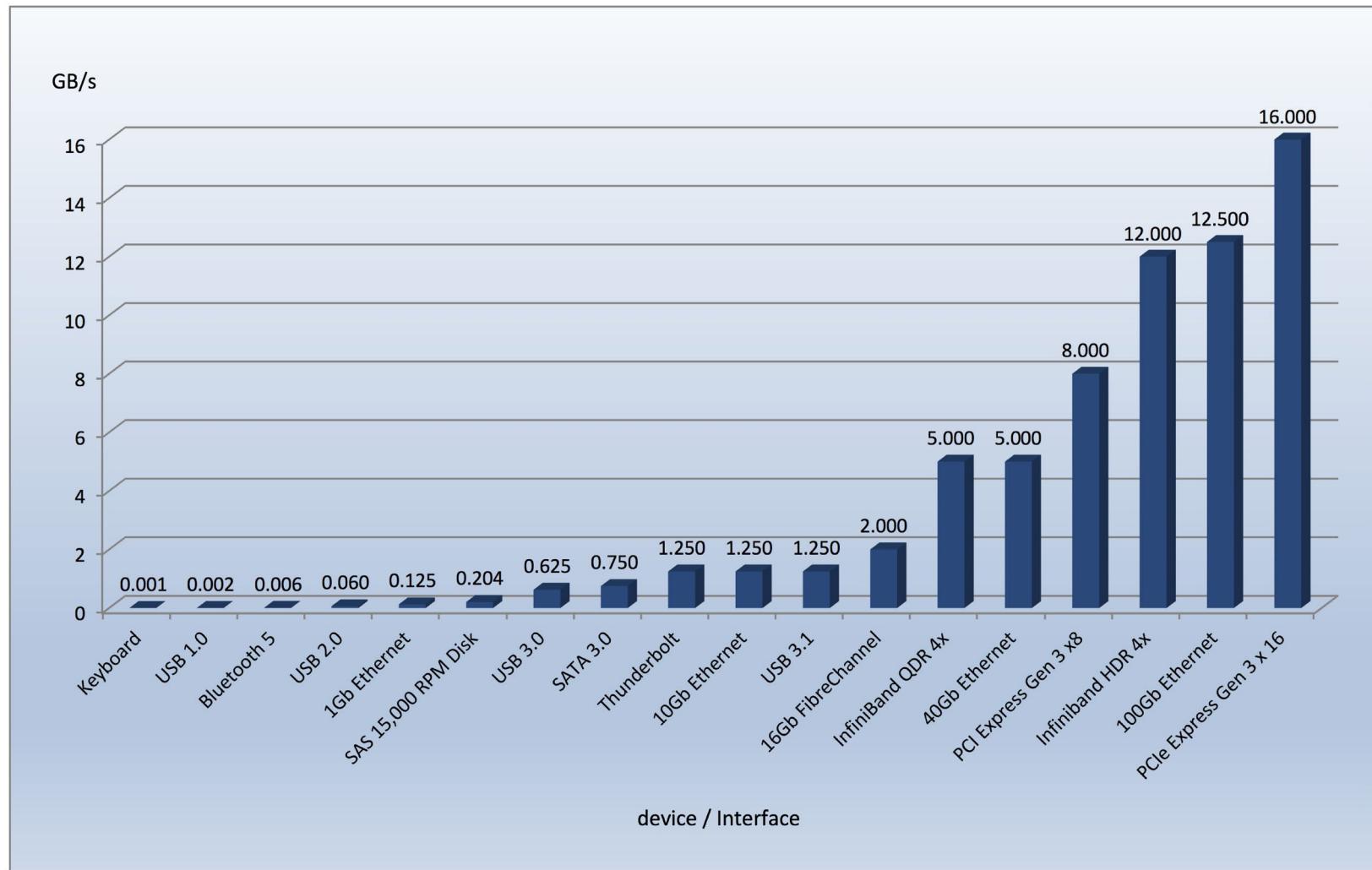


Device-status Table





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 retry-able errors
- Most return an error number or code when I/O request fails
- System error logs hold problem reports





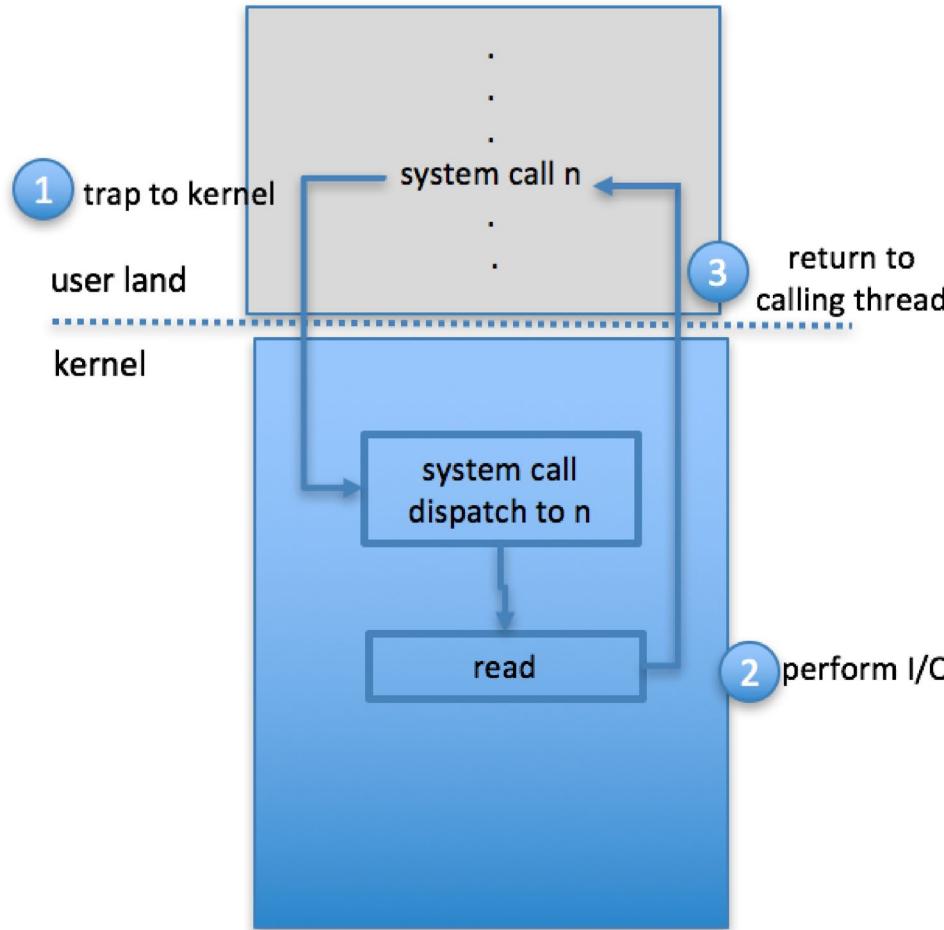
I/O Protection

- 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





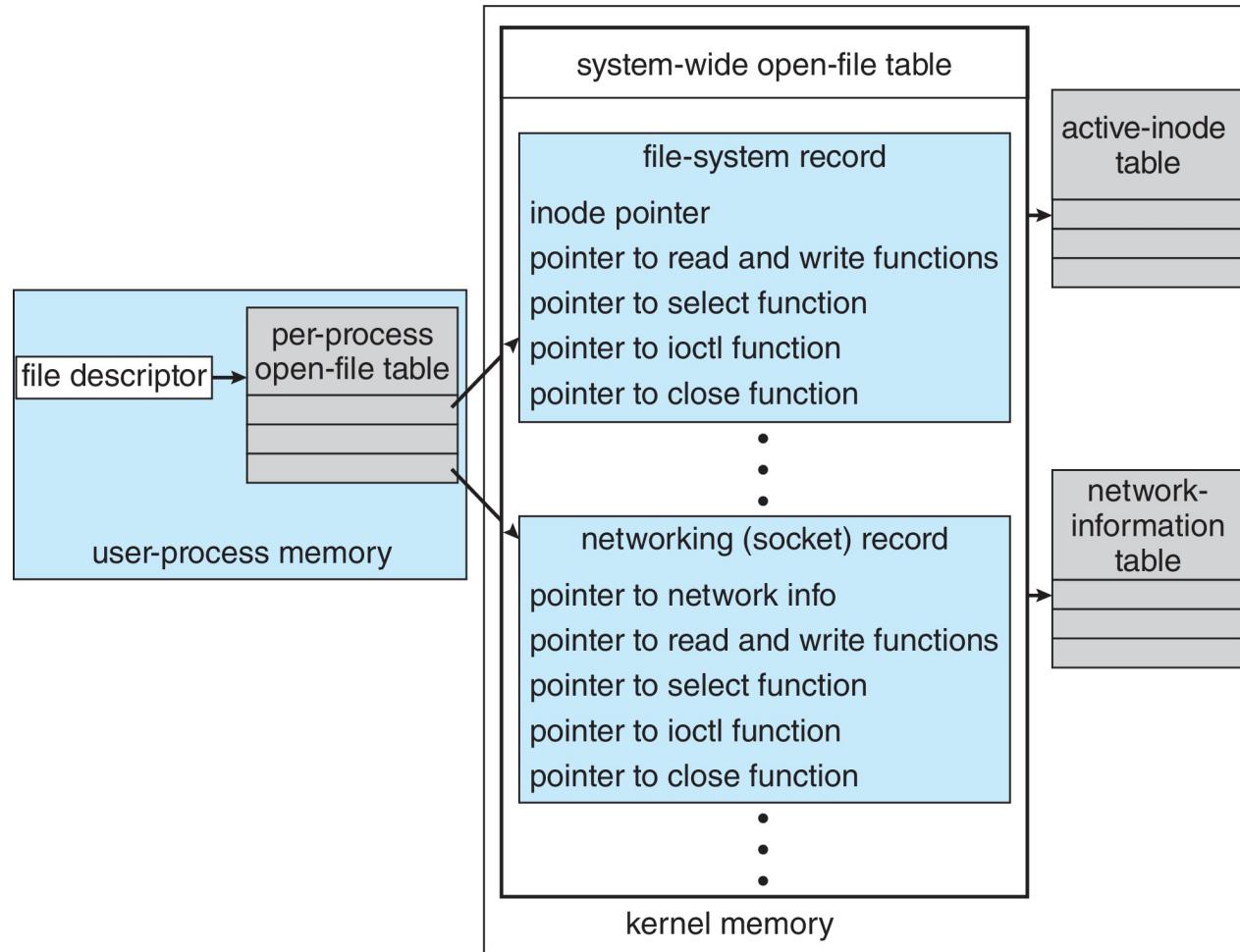
Kernel Data Structures

- 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.)

- For example, 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





Power Management (Cont.)

- For example, Android implements (Cont.)
 - 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)
- 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





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





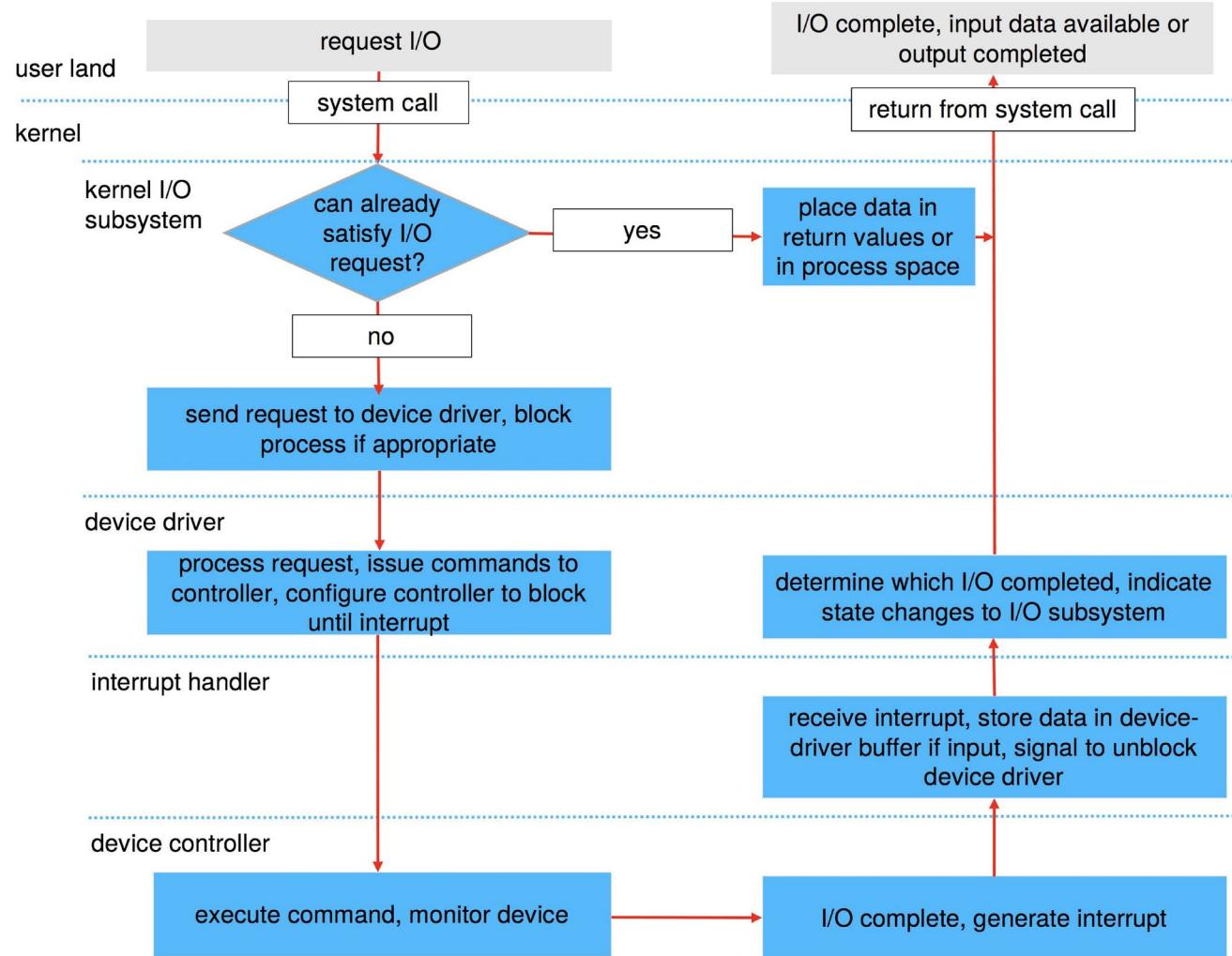
Transforming I/O Requests to Hardware Operations

- 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





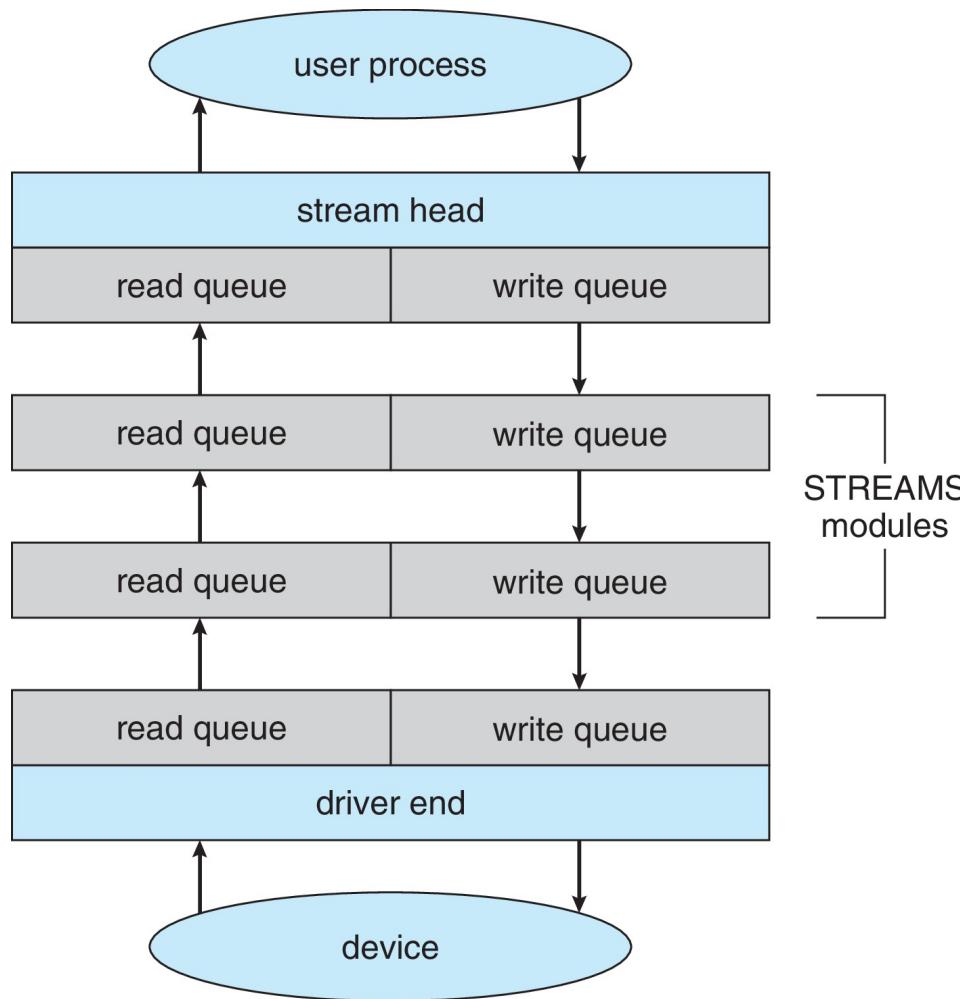
STREAMS

- **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





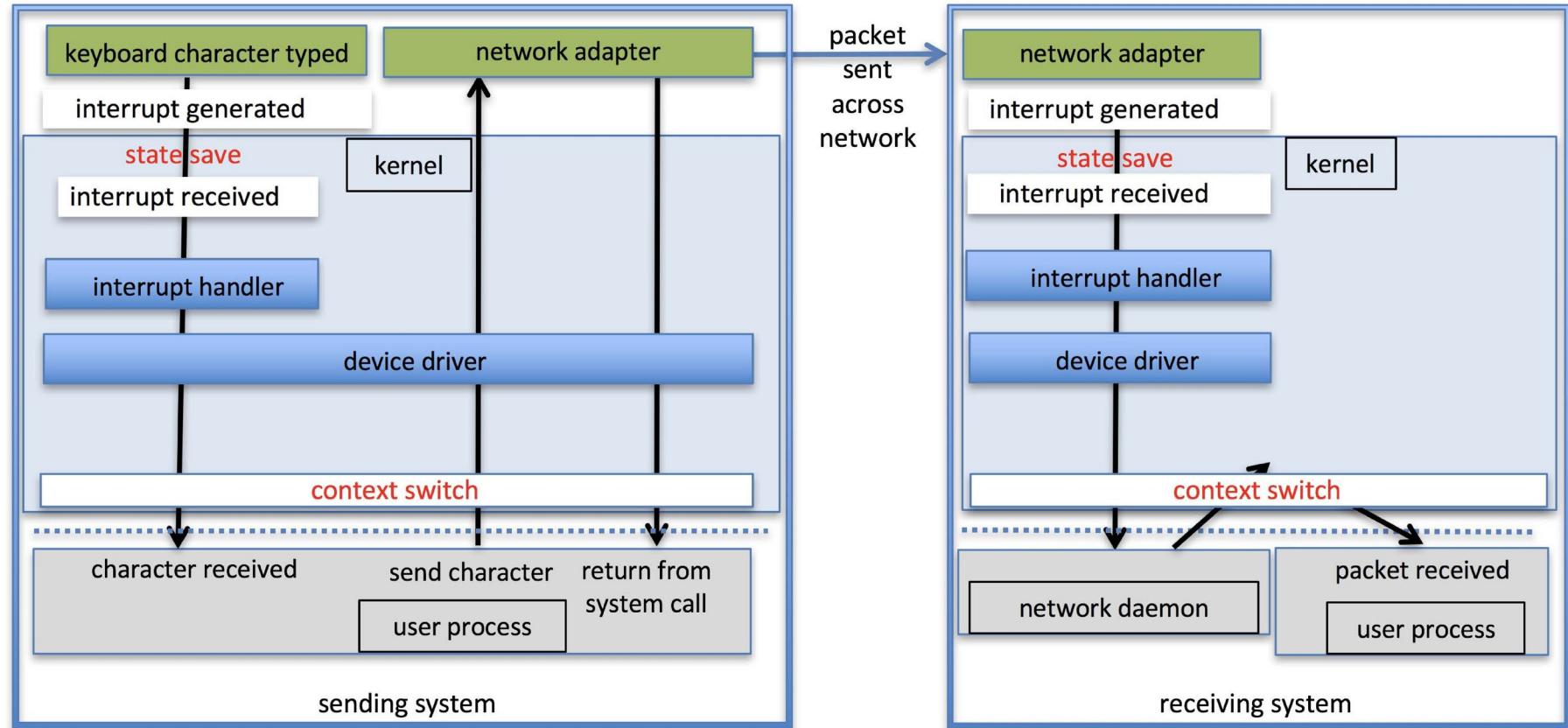
Performance

- 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





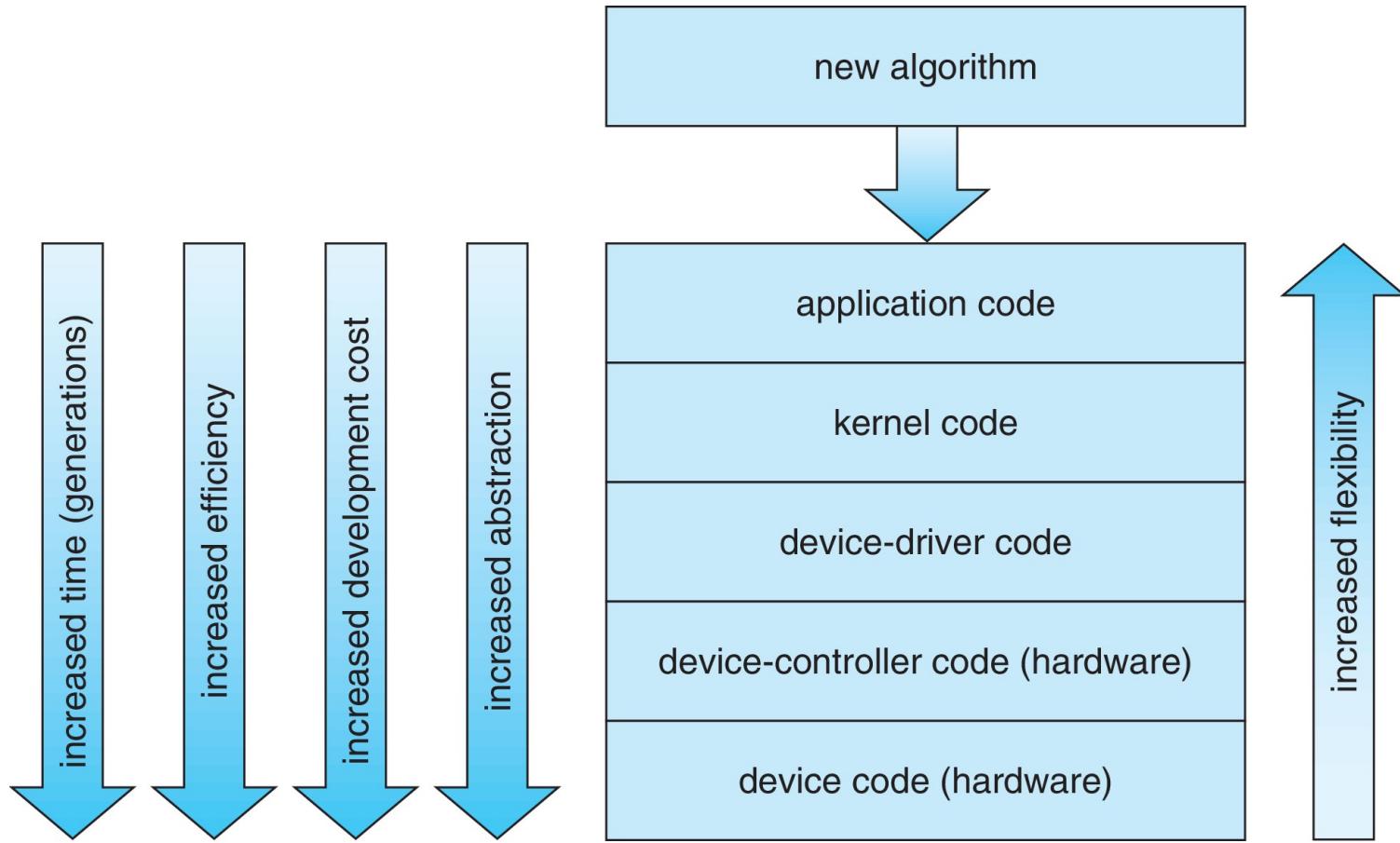
Improving Performance

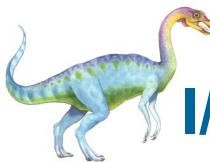
- Reduce number of context switches
- Reduce data copying
- Reduce interrupts by using large transfers, smart controllers, polling
- Use DMA
- Use smarter hardware devices
- Balance CPU, memory, bus, and I/O performance for highest throughput
- Move user-mode processes / daemons to kernel threads



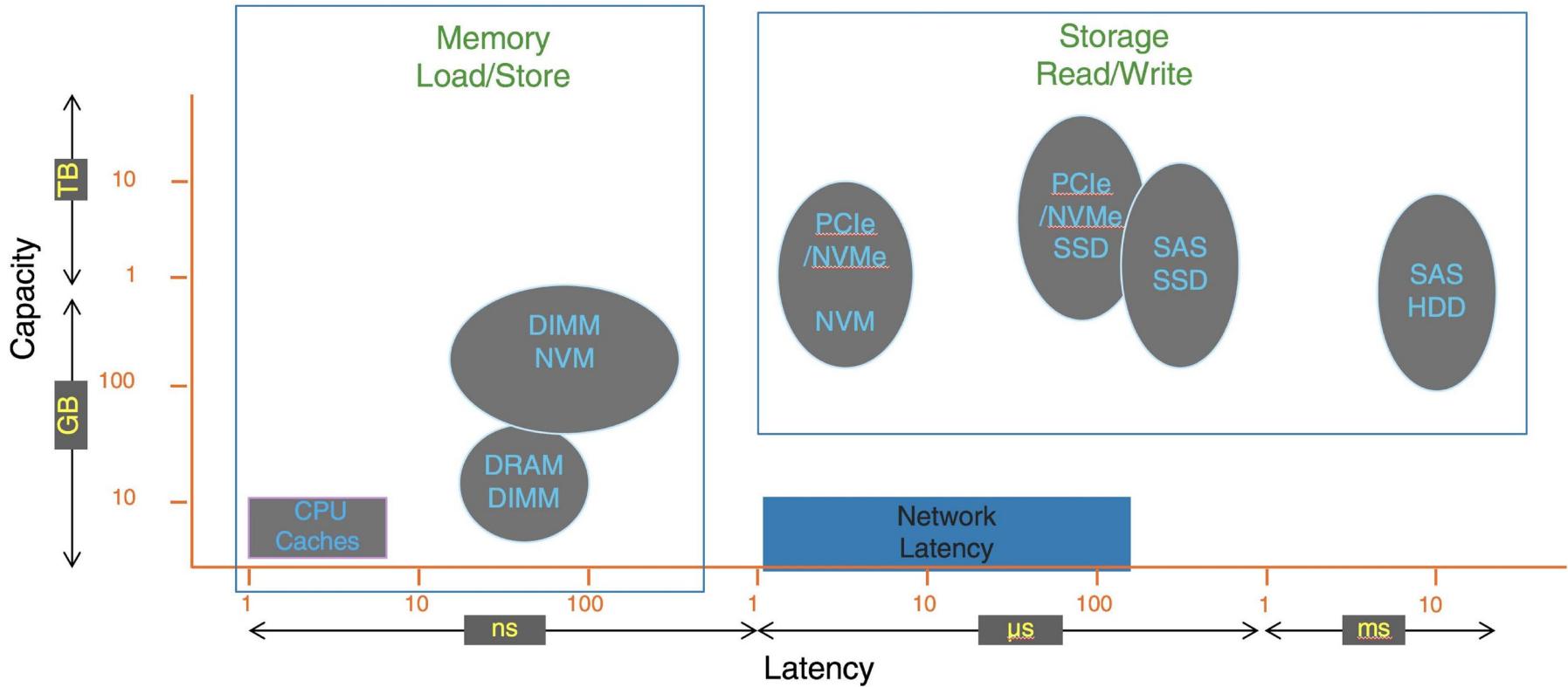


Device-Functionality Progression

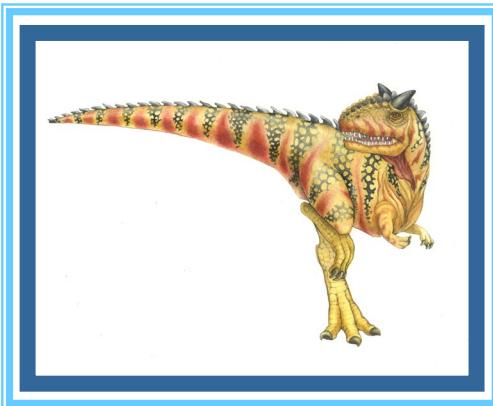




I/O Performance of Storage (and Network Latency)

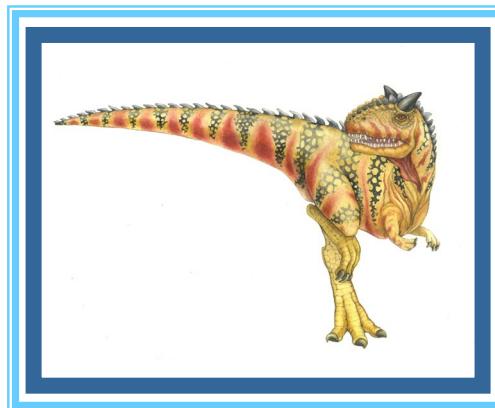


End of Chapter 12



Chapter 13:

File-System Interface





Outline

- File Concept
- Access Methods
- Disk and Directory Structure
- Protection
- Memory-Mapped Files





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
 - ▶ Numeric
 - ▶ Character
 - ▶ Binary
 - Program
- Contents defined by file's creator
 - Many types
 - ▶ **text file**,
 - ▶ **source file**,
 - ▶ **executable file**





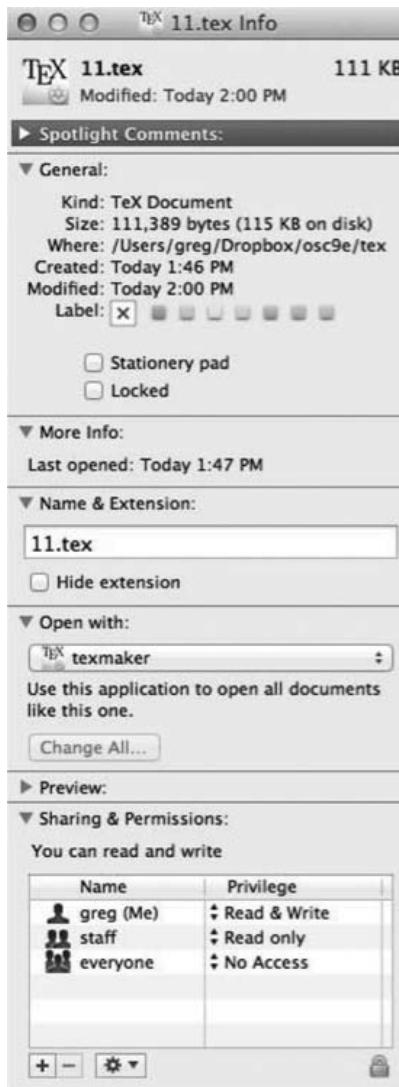
File Attributes

- **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
- Information kept in the directory structure





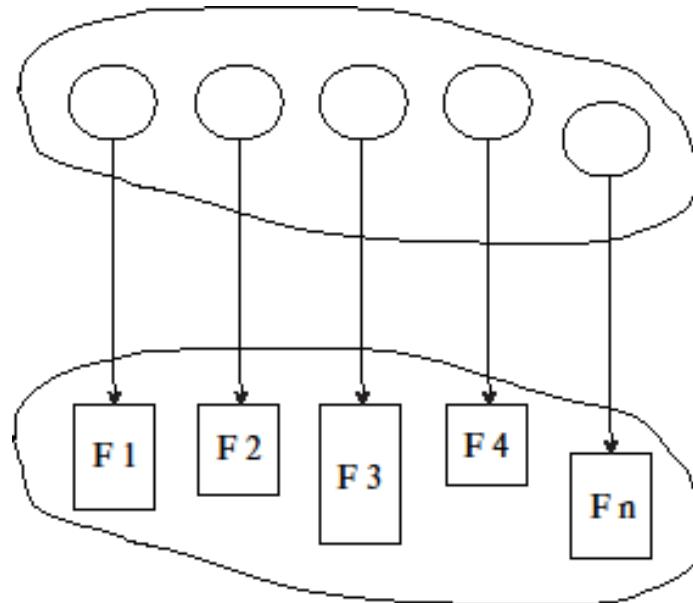
File info Window on Mac OS X





Directory Structure

- A collection of nodes containing information about all files



- Both the directory structure and the files reside on disk





File Operations

- **Create**
- **Write** – at **write pointer** location
- **Read** – at **read pointer** location
- **Reposition within file - 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





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, bytes
- Simple record structure
 - Lines
 - Fixed length
 - Variable length
- Complex Structures
 - Formatted document
 - Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters
- Who decides:
 - Operating system
 - Program





Access Methods

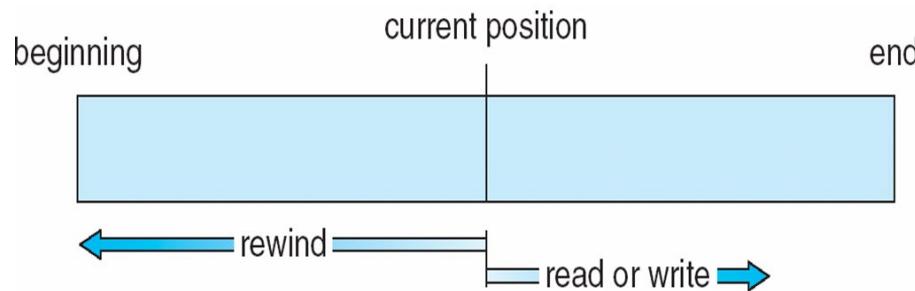
- A file is fixed length **logical records**
- **Sequential Access**
- **Direct Access**
- **Other Access Methods**





Sequential Access

- Operations
 - **read next**
 - **write next**
 - **Reset**
 - no read after last write (rewrite)
- Figure





Direct Access

- Operations
 - **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





Simulation of Sequential Access on Direct-access File

sequential access	implementation for direct access
<i>reset</i>	$cp = 0;$
<i>read next</i>	<i>read cp;</i> $cp = cp + 1;$
<i>write next</i>	<i>write cp;</i> $cp = cp + 1;$





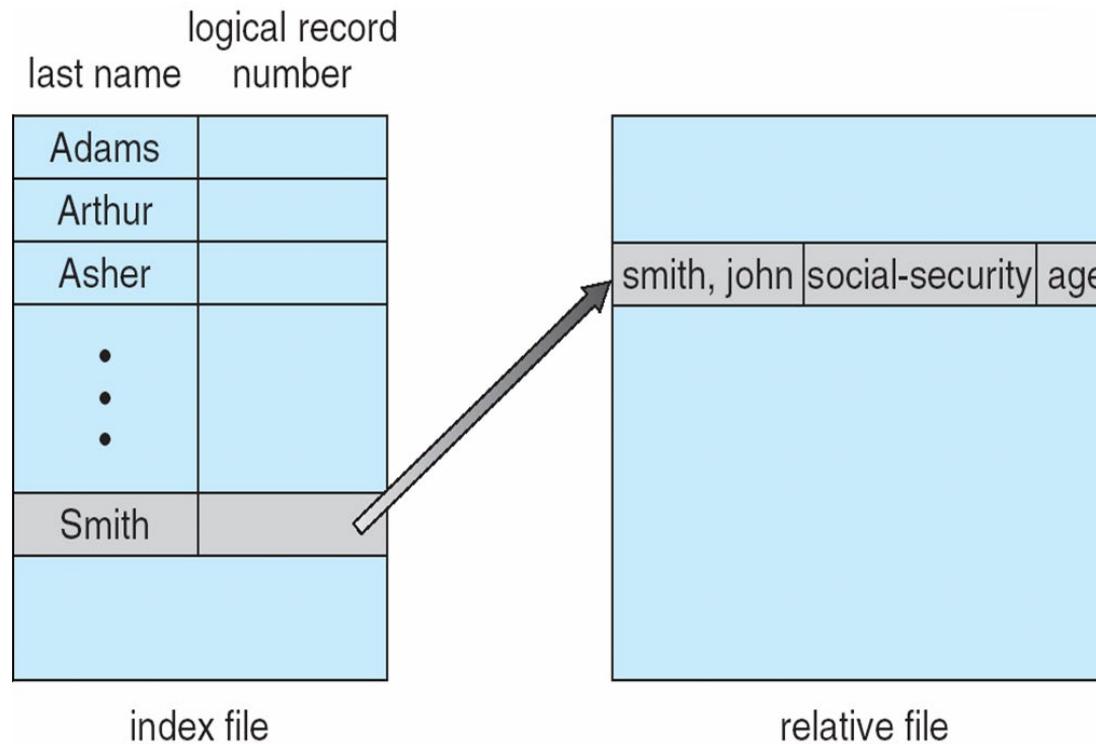
Other Access Methods

- Can be other access methods 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 Universal Produce Code (UPC code) plus record of data about that item)
- If the index is too large, create an in-memory index, which is an index of a disk index
- 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





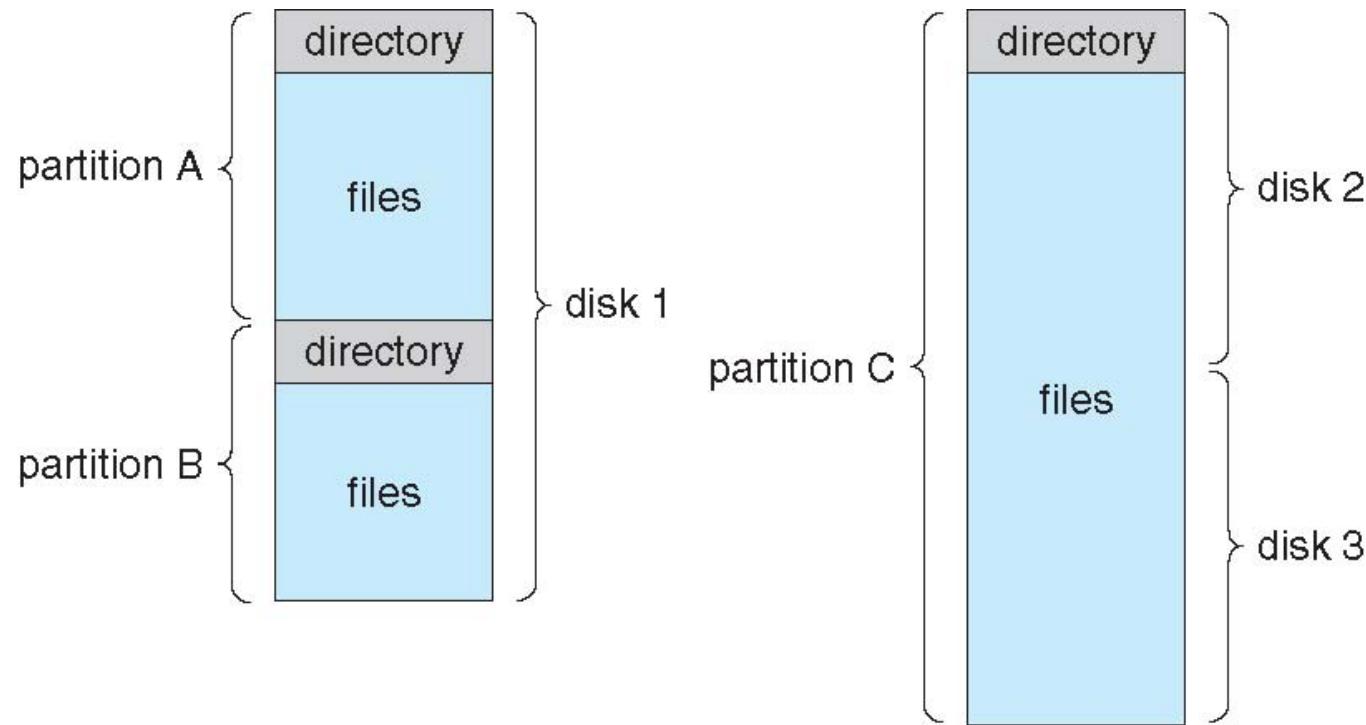
Disk Structure

- 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 is known as a **volume**
- Each volume containing a file system also tracks that file system's info in **device directory** or **volume table of contents**
- In addition to **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

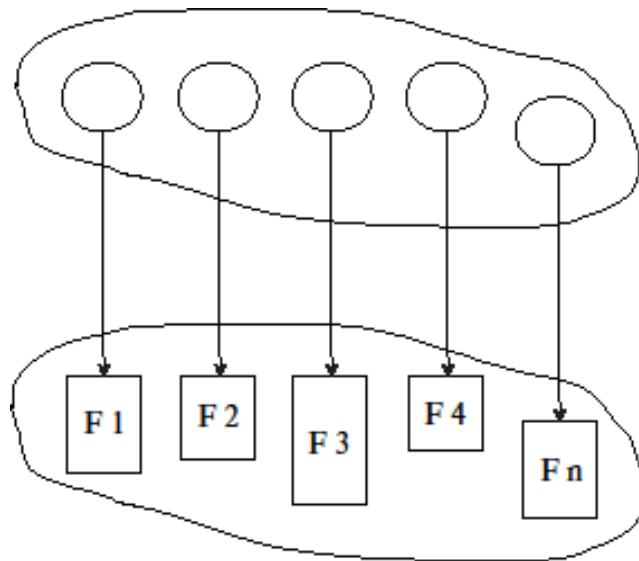
- We mostly talk of general-purpose file systems
- But systems frequently have many file systems, some general- and some special- purpose
- Consider Solaris has
 - tmpfs – memory-based volatile FS for fast, temporary I/O
 - objfs – interface into kernel memory to get kernel symbols for debugging
 - ctfs – 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





Directory Structure

- A collection of nodes containing information about all files



- Both the directory structure and the files reside on disk





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

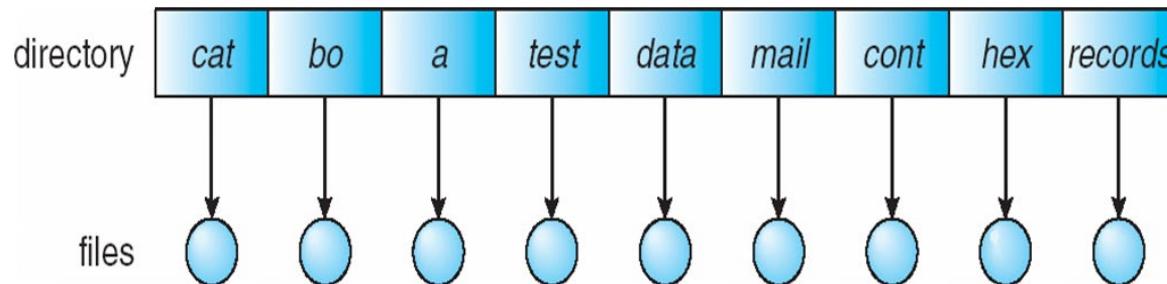
- Efficiency – locating a file quickly
- Naming – convenient to users
 - Two users can have same name for different files
 - The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, ...)





Single-Level Directory

- A single directory for all users



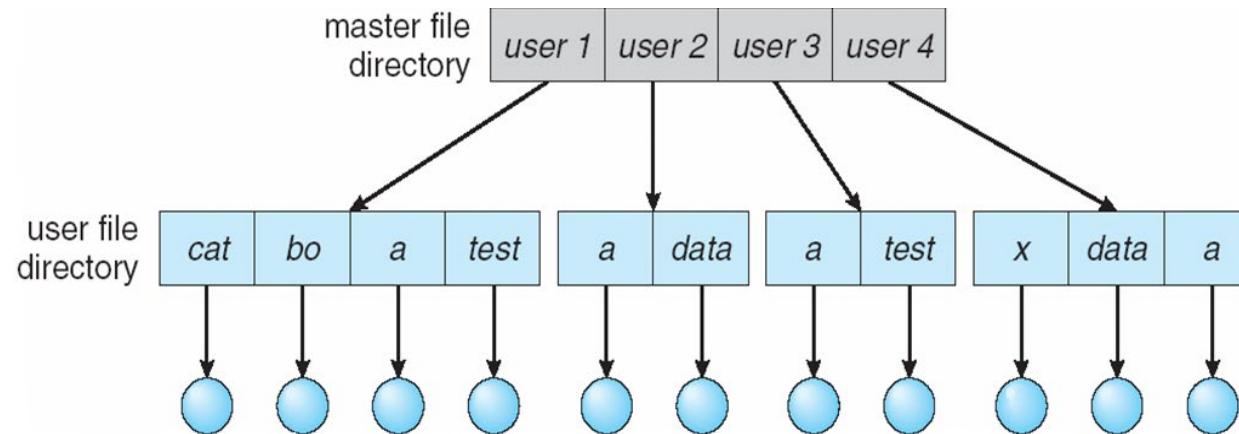
- Naming problem
- Grouping problem





Two-Level Directory

- Separate directory for each user

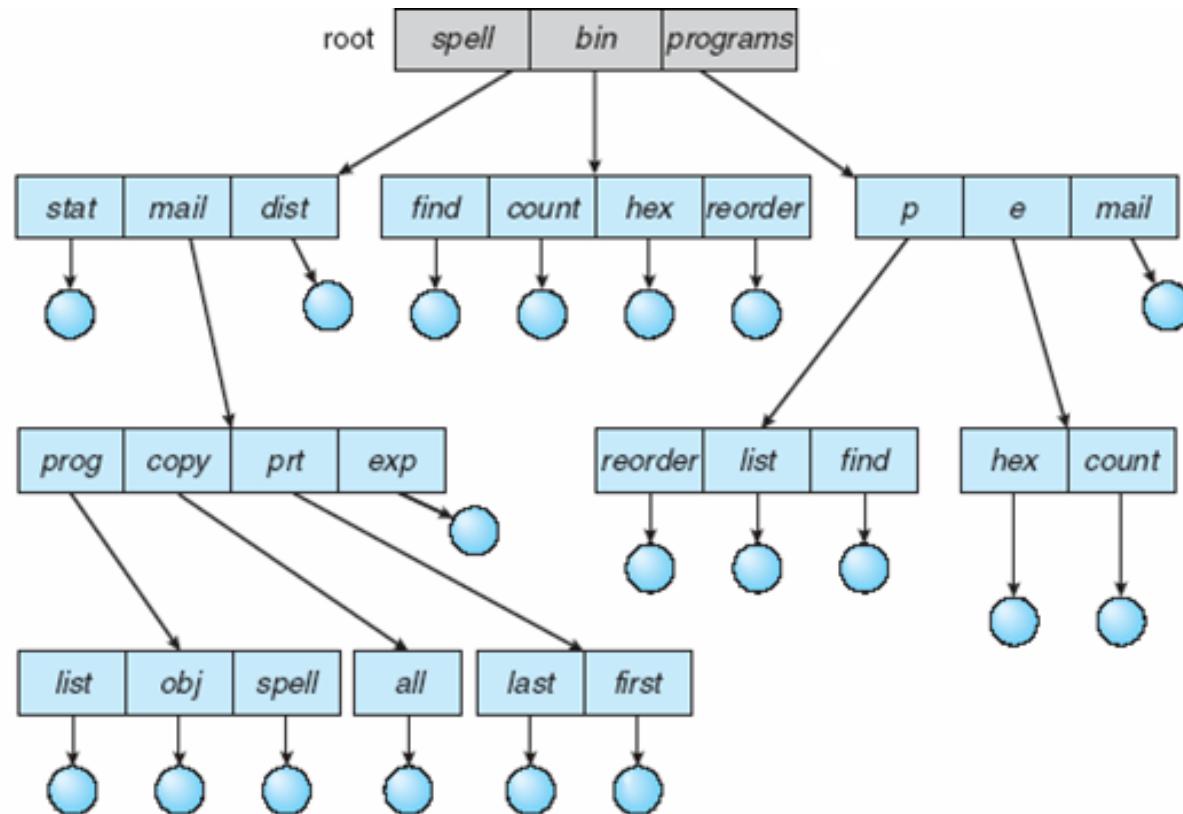


- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability





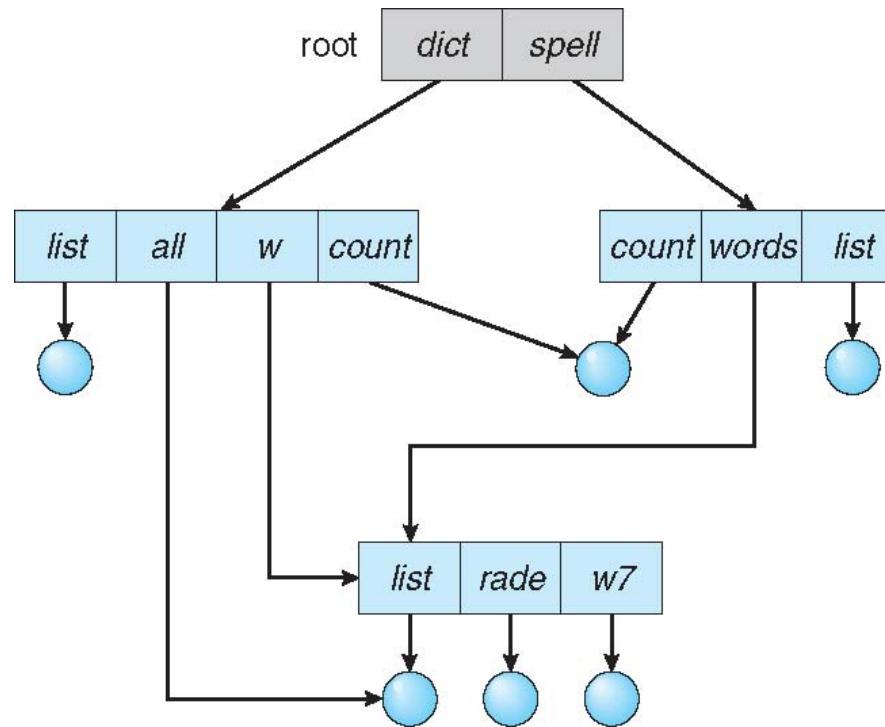
Tree-Structured Directories





Acyclic-Graph Directories

- Have shared subdirectories and files
- Example





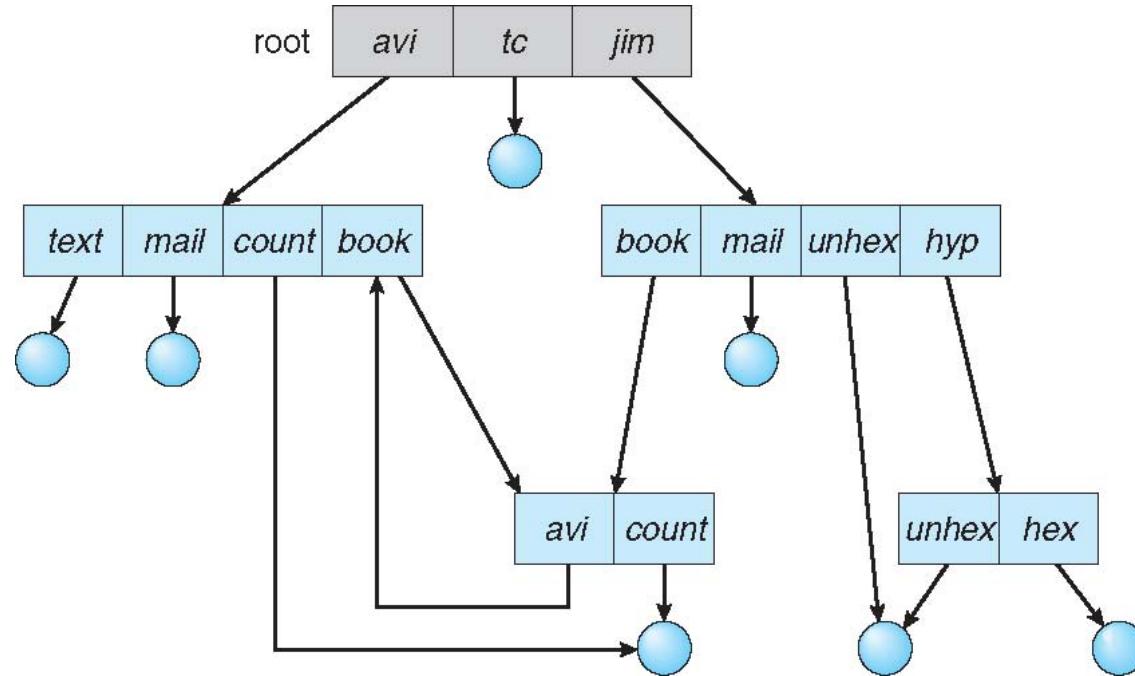
Acyclic-Graph Directories (Cont.)

- Two different names (aliasing)
- If **dict** deletes **w/list** \Rightarrow 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
- 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.)

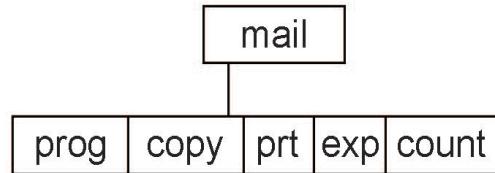
- How do we guarantee no cycles?
 - Allow only links to files not subdirectories
 - **Garbage collection**
 - Every time a new link is added use a cycle detection algorithm to determine whether it is OK





Current Directory

- Can designate one of the directories as the current (working) directory
 - `cd /spell/mail/prog`
 - `type list`
- Creating and deleting a file is done in current directory
- Example of creating a new file
 - If in current directory is `/mail`
 - The command
`mkdir <dir-name>`
 - Results in:



- Deleting “mail” ⇒ deleting the entire subtree rooted by “mail”





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 in Unix

- Mode of access: read, write, execute
- 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 file (say *game*) or subdirectory, define an appropriate access.



- Attach a group to a file

chgrp G game





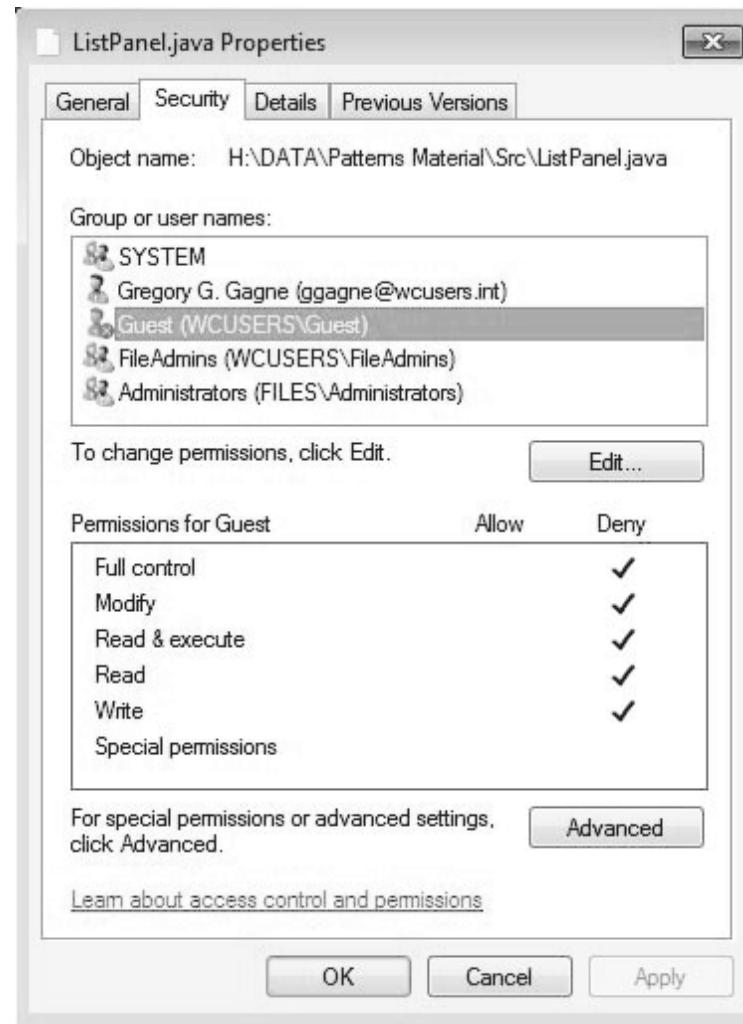
A Sample UNIX Directory Listing

-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/





Windows 7 Access-Control List Management

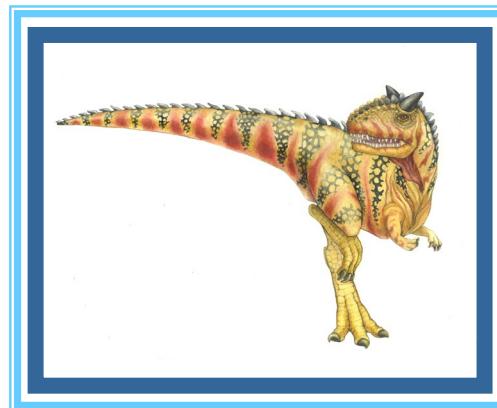




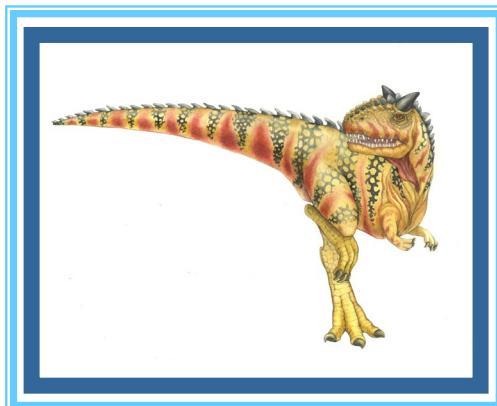
Memory-Mapped Files



End of Chapter 13



Chapter 14: File System Implementation





Outline

- 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





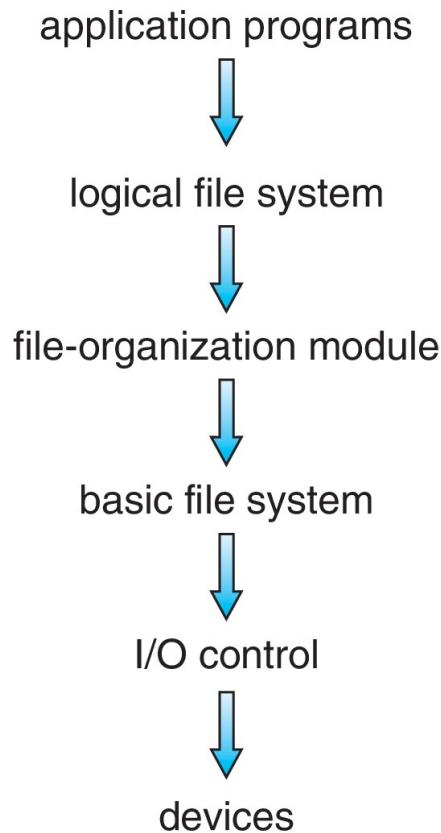
File-System Structure

- File structure
 - Logical storage unit
 - Collection of related information
- **File system** resides on secondary storage (disks)
 - Provided user interface to storage, mapping logical to physical
 - Provides efficient and convenient access to disk by allowing data to be stored, located 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 organized into layers





Layered File System





File System Layers

- **Device drivers** manage I/O devices at the I/O control layer

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 (**inodes** in UNIX)
 - Directory management
 - Protection
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
- Logical layers can be implemented by any coding method according to OS designer





File System Layers (Cont.)

- Many file systems, sometimes many 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, etc.)
 - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE





File-System Operations

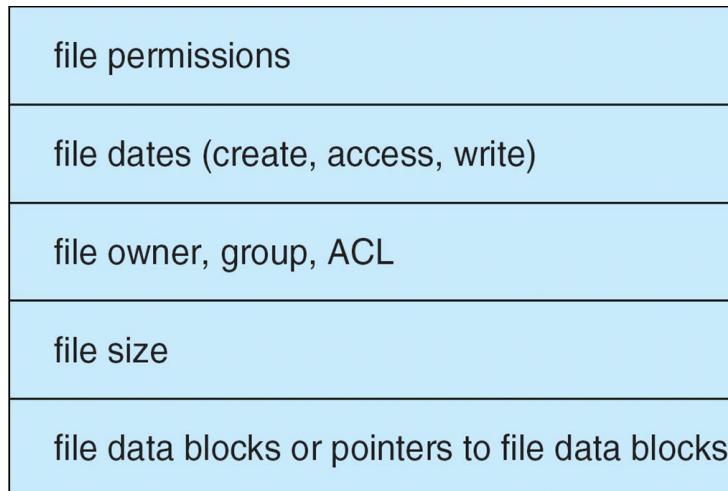
- We have system calls at the API level, but how do we implement their functions?
 - On-disk and in-memory structures
- **Boot control block** contains info needed by system to boot OS from that volume
 - Needed if volume contains OS, usually first block of volume
- **Volume control block (superblock, 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 Control Block (FCB)

- OS maintains **FCB** per file, which contains many details about the file
 - Typically, inode number, permissions, size, dates
 - Example





In-Memory File System Structures

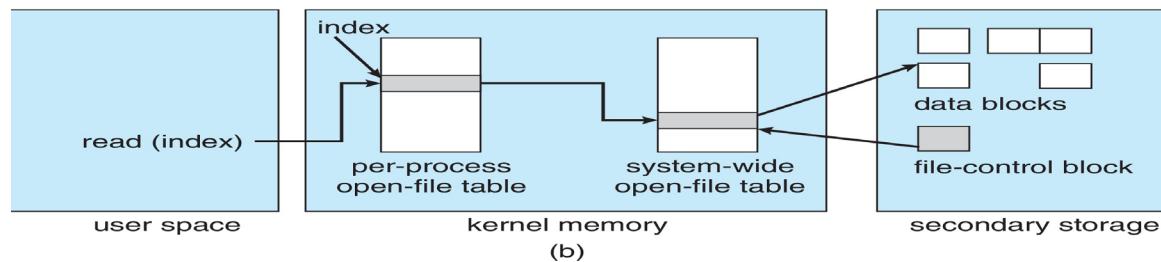
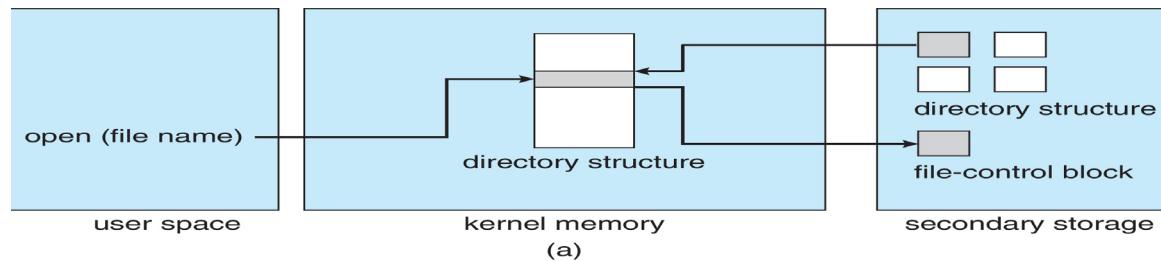
- **Mount table** storing file system mounts, mount points, file system types
- **System-wide open-file table** contains a copy of the FCB of each file and other info
- **Per-process open-file table** contains pointers to appropriate entries in system-wide open-file table as well as other info





In-Memory File System Structures (Cont.)

- Figure 12-3(a) refers to opening a file
- Figure 12-3(b) refers to reading a file





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 Method

- An allocation method refers to how disk blocks are allocated for files:
 - Contiguous
 - Linked
 - File Allocation Table (FAT)





Contiguous Allocation Method

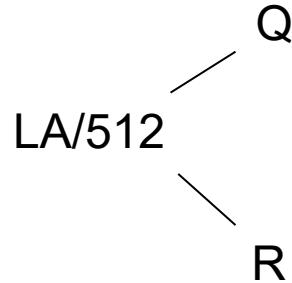
- An allocation method refers to how disk blocks are allocated for files:
- Each file occupies set of contiguous blocks
 - Best performance in most cases
 - Simple – only starting location (block #) and length (number of blocks) are required
 - Problems include:
 - ▶ Finding space on the disk for a file,
 - ▶ Knowing file size,
 - ▶ External fragmentation, need for **compaction off-line (downtime)** or **on-line**



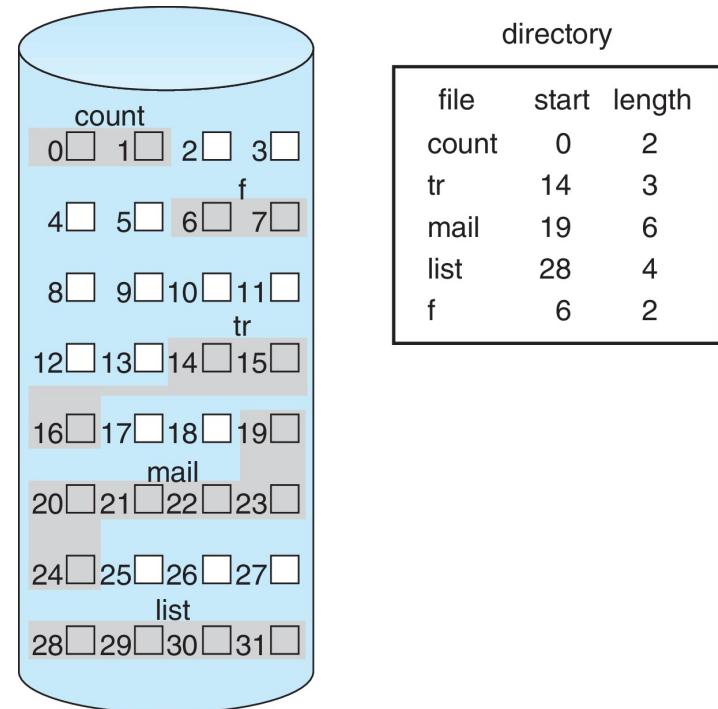


Contiguous Allocation (Cont.)

- Mapping from logical to physical
(block size = 512 bytes)



- Block to be accessed = starting address + Q
- Displacement into block = R





Extent-Based Systems

- Many newer file systems (i.e., 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





Linked Allocation

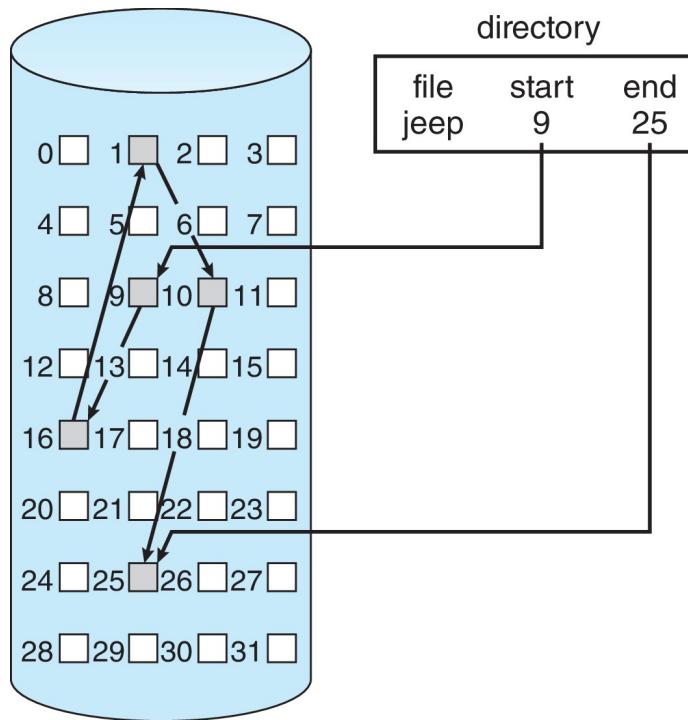
- Each file is a linked list of blocks
- File ends at nil pointer
- No external fragmentation
- Each block contains pointer to next block
- No compaction, external fragmentation
- Free space management system called when new block needed
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem
- Locating a block can take many I/Os and disk seeks





Linked Allocation Example

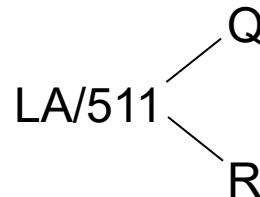
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk
- Scheme





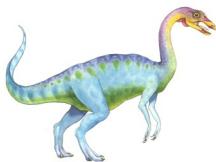
Linked Allocation (Cont.)

- Mapping



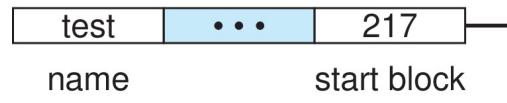
- Block to be accessed is the Q^{th} block in the linked chain of blocks representing the file.
- Displacement into block = $R + 1$





File-Allocation Table

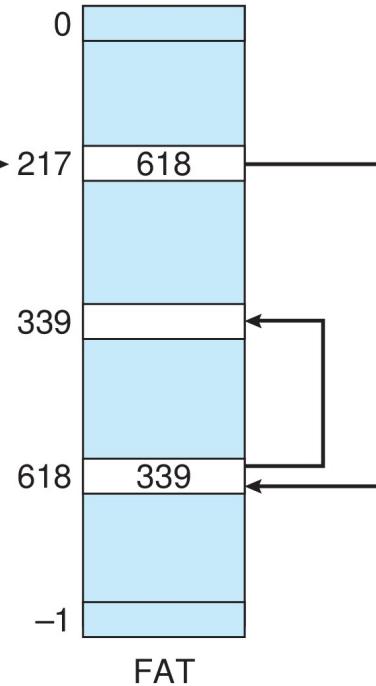
directory entry



start block

number of disk blocks

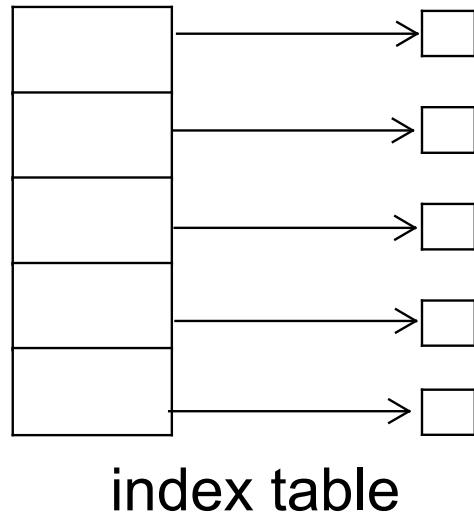
-1





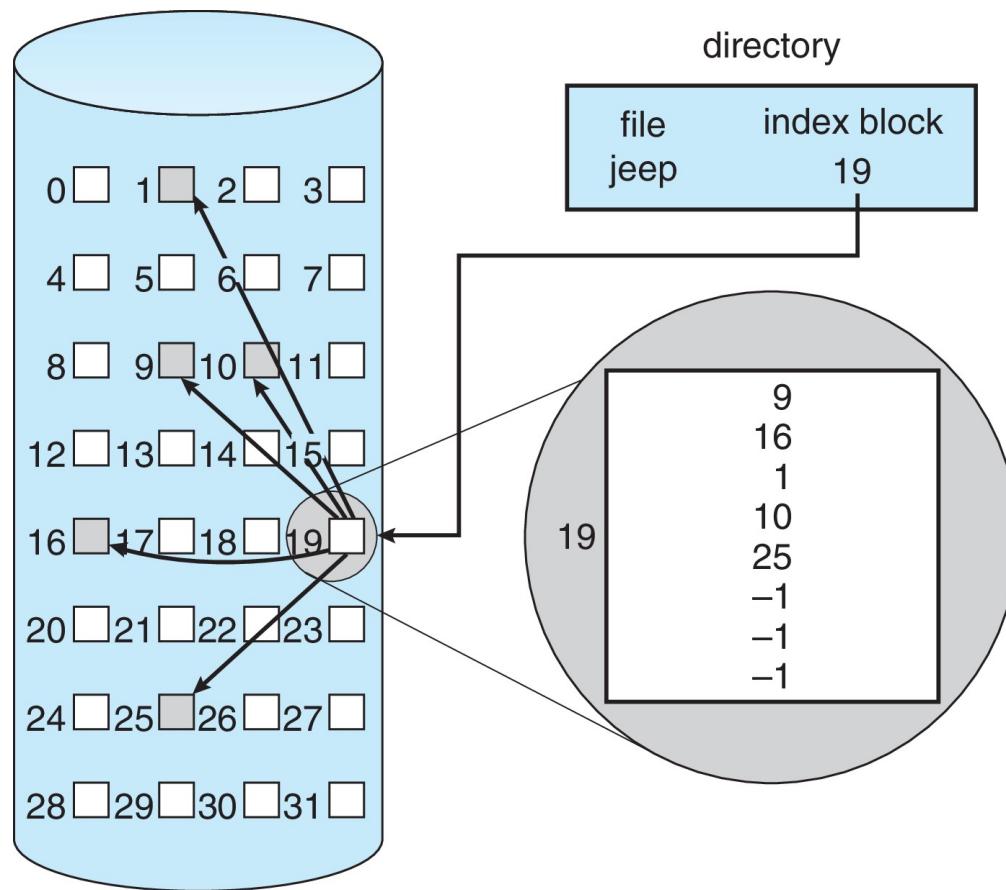
Indexed Allocation Method

- Each file has its own **index block**(s) of pointers to its data blocks
- Logical view





Example of Indexed Allocation





Performance

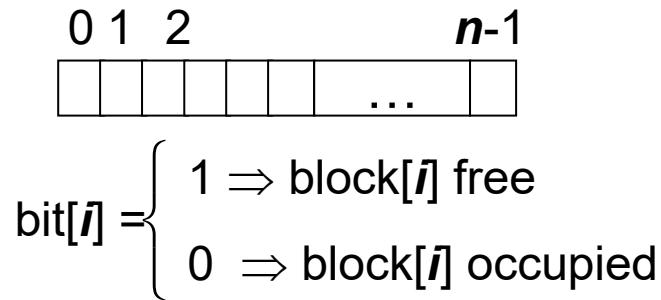
- Best method depends on file access type
 - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation
 - Select either contiguous or linked
- Indexed more complex
 - Single block access could require 2 index block reads then data block read
 - Clustering can help improve throughput, reduce CPU overhead
- For NVM, no disk head so different algorithms and optimizations needed
 - Using old algorithm uses many CPU cycles trying to avoid non-existent head movement
 - Goal is to reduce CPU cycles and overall path needed for I/O





Free-Space Management

- File system maintains **free-space list** to track available blocks/clusters
 - (Using term “block” for simplicity)
 - **Bit vector** or **bit map** (n blocks)



Block number calculation

(number of bits per word) *
(number of 0-value words) +
offset of first 1 bit

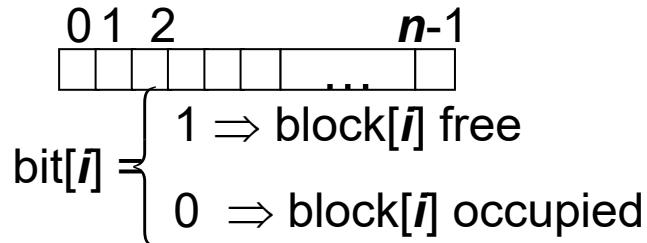
CPUs have instructions to return offset within word of first “1” bit





Free-Space Management

- File system maintains **free-space list** to track available blocks
- Bit vector** or **bit map** (n blocks)



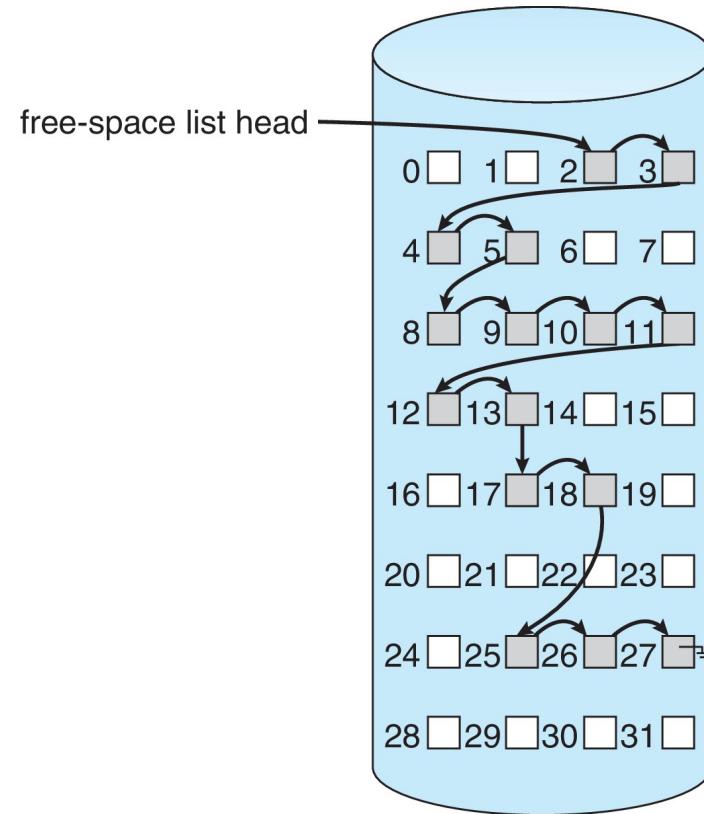
- Bit map requires extra space
 - Example:
 - block size = 4KB = 2^{12} bytes
 - disk size = 2^{40} bytes (1 terabyte)
 - $n = 2^{40}/2^{12} = 2^{28}$ bits (or 32MB)
 - if clusters of 4 blocks -> 8MB of memory
- Easy to get contiguous files





Linked Free Space List on Disk

- Linked list (free list)
 - Cannot get contiguous space easily
 - No waste. Linked Free Space List on Disk 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 cannot 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 (like 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





Efficiency and Performance (Cont.)

- 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





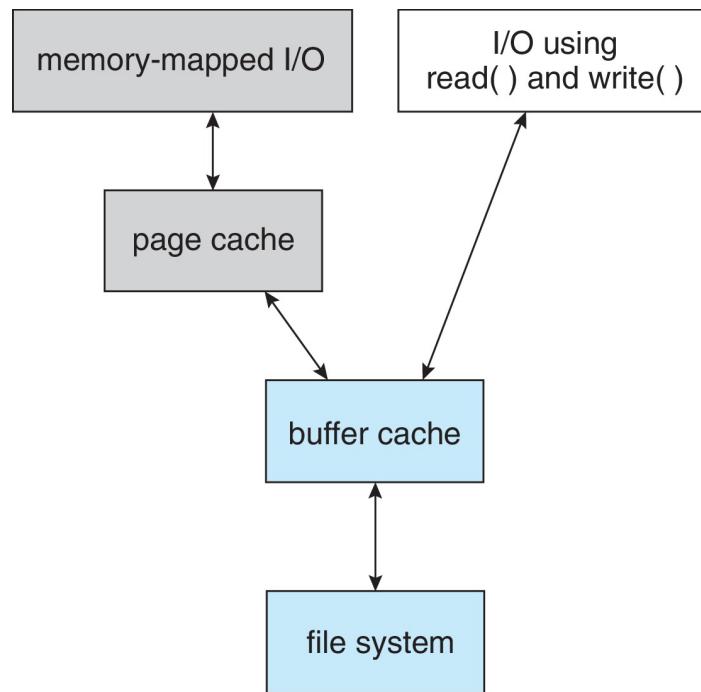
Page Cache

- 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

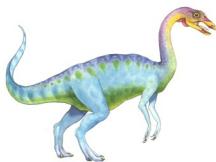




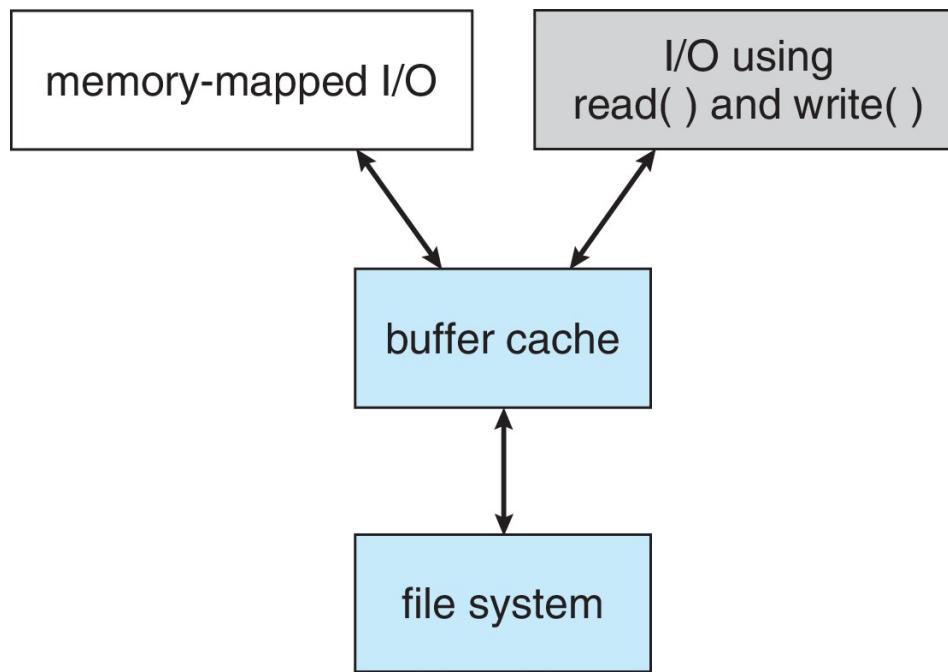
Unified Buffer Cache

- 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?





I/O Using a Unified Buffer Cache





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





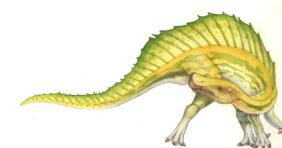
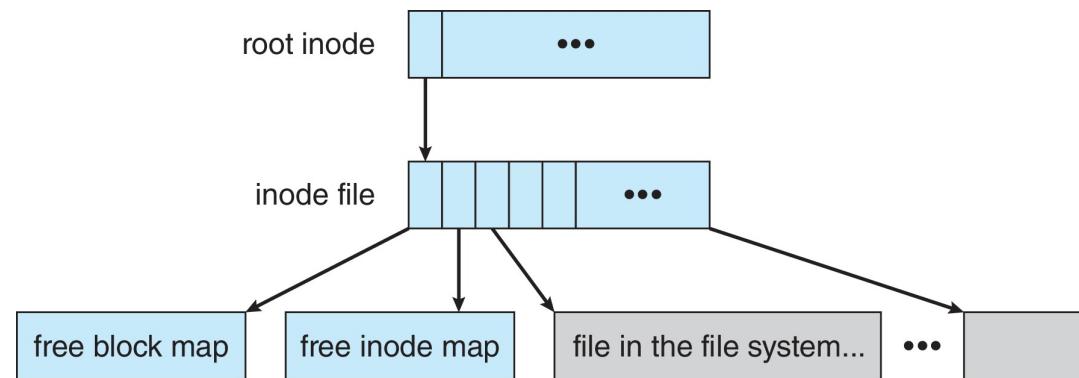
Example: WAFL File System

- Used on Network Appliance “Filers” – distributed file system appliances
- “Write-anywhere file layout”
- 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



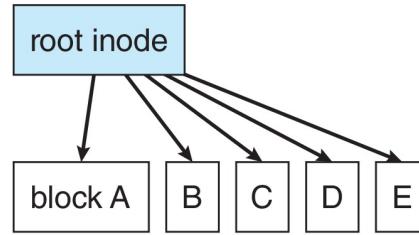


The WAFL File Layout

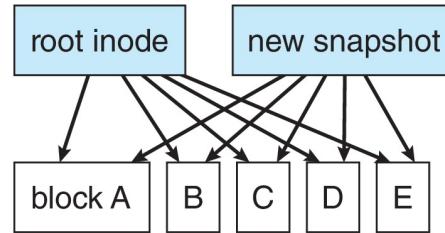




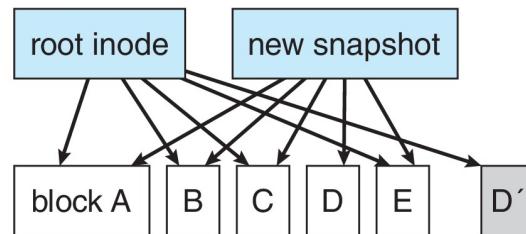
Snapshots in WAFL



(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.

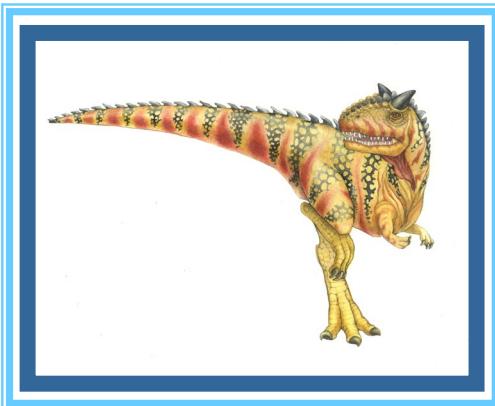
Apple File System (APFS) is a good example of such a design. Its goal is to run on all current Apple devices, from the Apple Watch through the iPhone to the Mac computers. Creating a file system that works in watchOS, I/Os, tvOS, and macOS is certainly a challenge. APFS is feature-rich, including 64-bit 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.

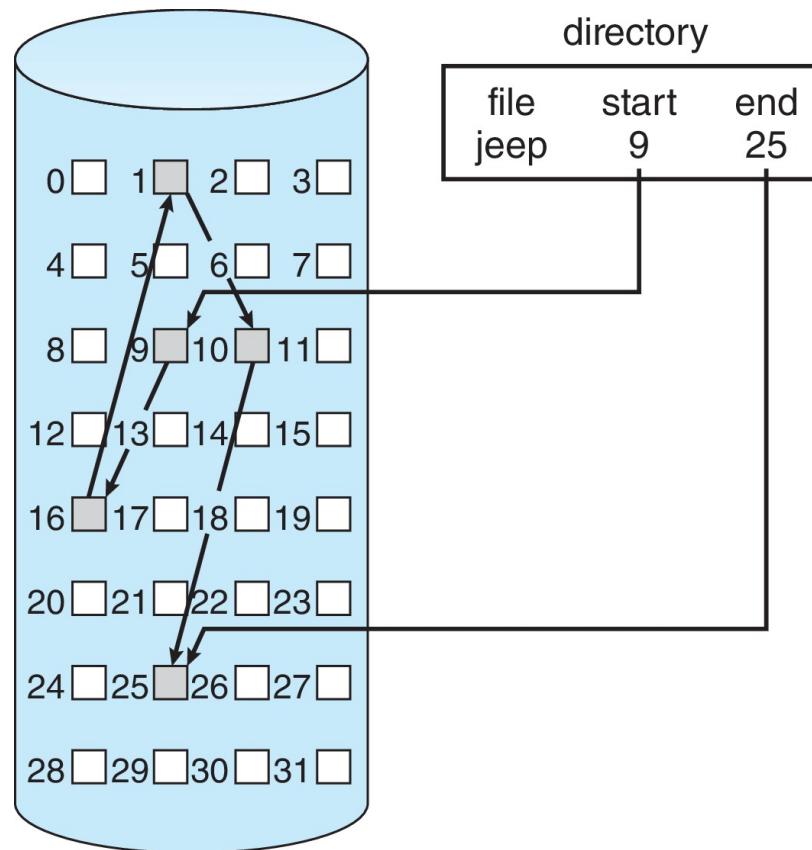


End of Chapter 14





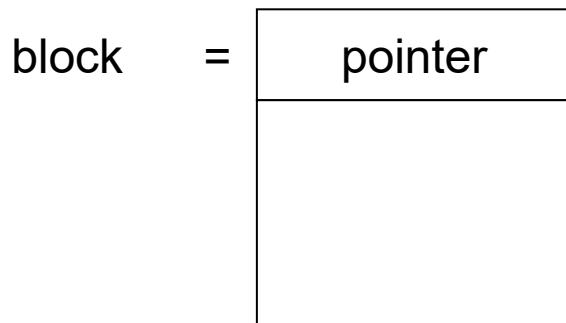
Linked Allocation



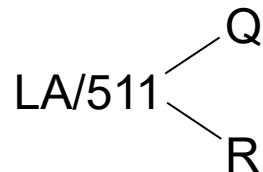


Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk



- Mapping

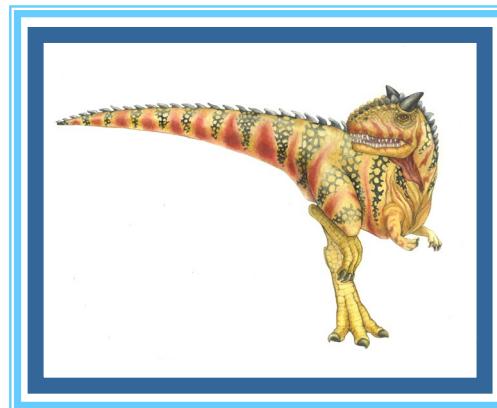


Block to be accessed is the Qth block in the linked chain of blocks representing the file.

Displacement into block = R + 1



End of Chapter 14



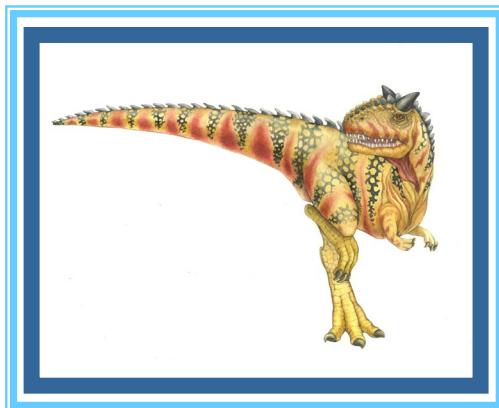


In-Memory File System Structures

- **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 12-3(a) refers to opening a file
- Figure 12-3(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 eventually copied to specified user process memory address



Chapter 15: File System Internals





Outline

- File Systems
- File-System Mounting
- Partitions and Mounting
- File Sharing
- Virtual File Systems
- Remote File Systems
- Consistency Semantics
- NFS





Objectives

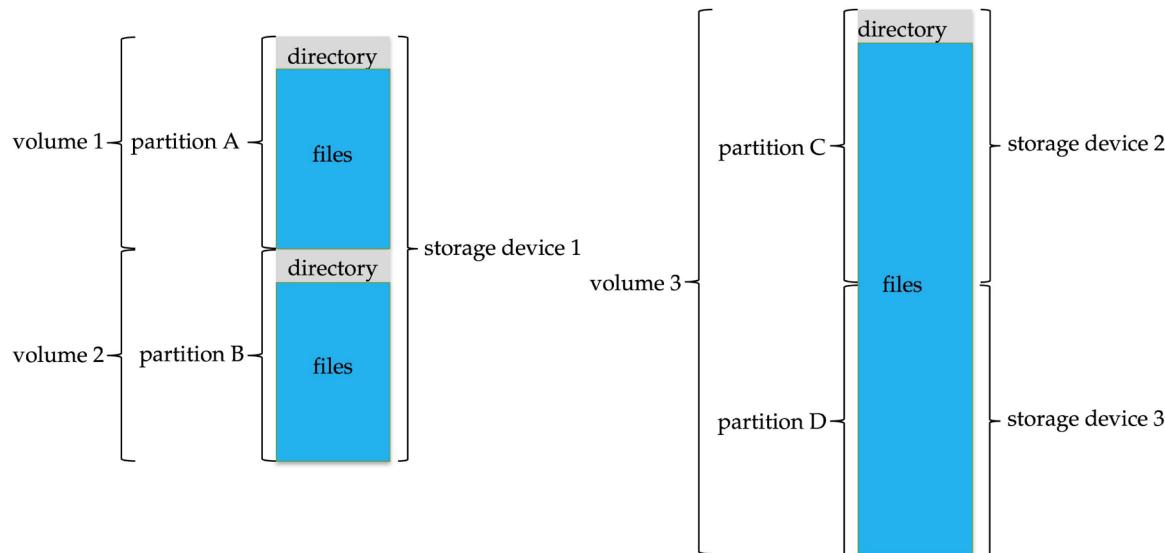
- Delve into the details of file systems and their implementation
- Explore booting and file sharing
- Describe remote file systems, using NFS as an example





File System

- General-purpose computers can have multiple storage devices
- Devices can be sliced into partitions, which hold volumes
- Volumes can span multiple partitions
- Each volume usually formatted into a file system
- # of file systems varies, typically dozens available to choose from
- Typical storage device organization:





Solaris File Systems

/	ufs
/devices	devfs
/dev	dev
/system/contract	ctfs
/proc	proc
/etc/mnttab	mntfs
/etc/svc/volatile	tmpfs
/system/object	objfs
/lib/libc.so.1	lofs
/dev/fd	fd
/var	ufs
/tmp	tmpfs
/var/run	tmpfs
/opt	ufs
/zpbge	zfs
/zpbge/backup	zfs
/export/home	zfs
/var/mail	zfs
/var/spool/mqueue	zfs
/zpbg	zfs
/zpbg/zones	zfs





Partitions and Mounting

- Partition can be a volume containing a file system (“cooked”) or **raw** – just a sequence of blocks with no file system
- 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
- **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 on **mount points** – location at which they can be accessed
- 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

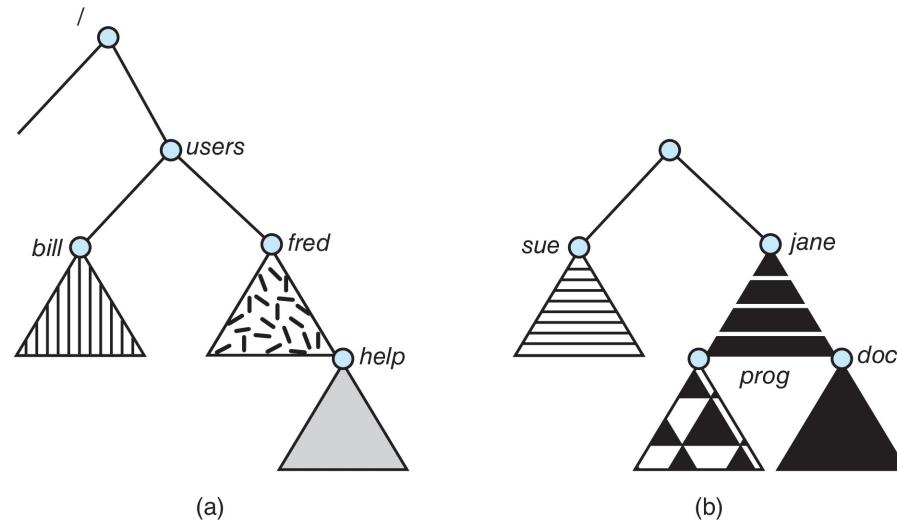




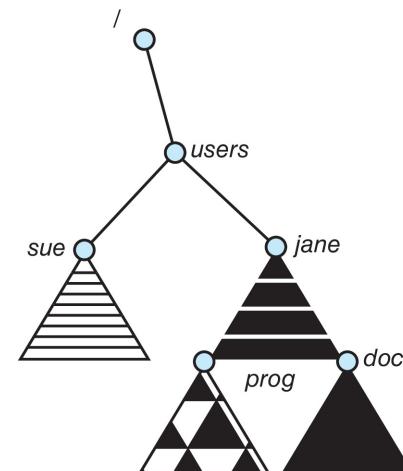
File Systems and Mounting

(a) Unix-like file system directory tree

(a) Unmounted file system



After mounting (b)
into the existing
directory tree





File Sharing

- Allows multiple users / systems access to the same files
- Permissions / protection must be implemented and accurate
 - Most systems provide concepts of owner, group member
 - Must have a way to apply these between systems





Virtual File Systems

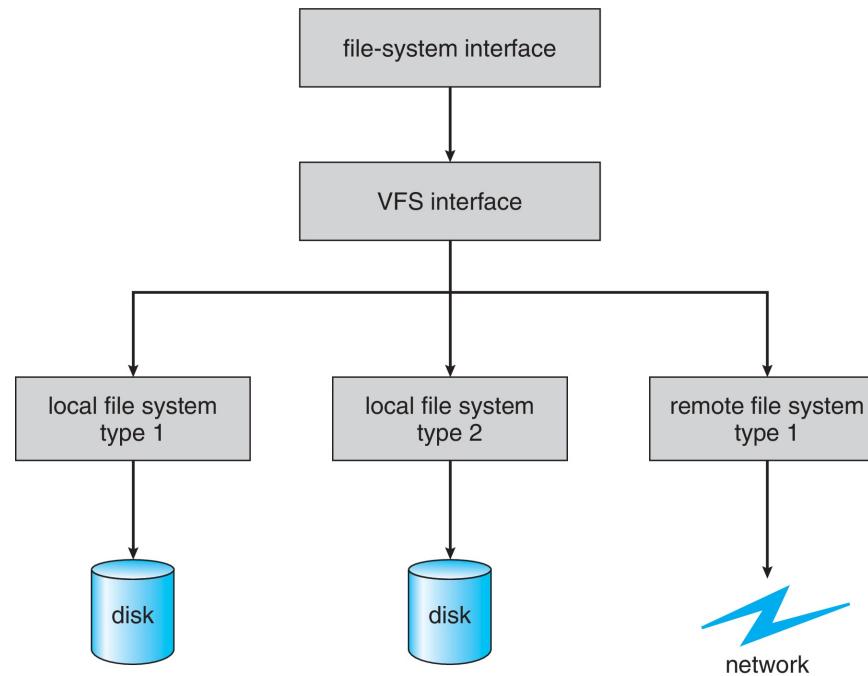
- **Virtual File Systems (VFS)** on Unix provide an object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
 - Separates file-system generic operations from implementation details
 - Implementation can be one of many file systems types, or network file system
 - ▶ Implements **vnodes** which hold inodes or network file details
 - Then dispatches operation to appropriate file system implementation routines





Virtual File Systems (Cont.)

- The API is to the VFS interface, rather than any specific type of file system
- Example





Virtual File System Implementation

- For example, Linux has four object types:
 - **inode, file, superblock, dentry**
- VFS defines set of operations on the objects that must be implemented
 - Every object has a pointer to a function table
 - ▶ Function table has addresses of routines to implement that function on that object
 - ▶ For example:
 - `int open(. . .)`—Open a file
 - `int close(. . .)`—Close an already-open file
 - `ssize_t read(. . .)`—Read from a file
 - `ssize_t write(. . .)`—Write to a file
 - `int mmap(. . .)`—Memory-map a file





Remote File Systems

- Sharing of files across a network
- First method involved manually sharing each file – programs like **ftp**
- Second method uses a **distributed file system (DFS)**
 - Remote directories visible from local machine
- Third method – **World Wide Web**
 - A bit of a revision to first method
 - Use browser to locate file/files and download /upload
 - **Anonymous** access doesn't require authentication





Client-Server Model

- Sharing between a server (providing access to a file system via a network protocol) and a client (using the protocol to access the remote file system)
- Identifying each other via network ID can be spoofed, encryption can be performance expensive
- NFS an example
 - User auth info on clients and servers must match (UserIDs for example)
 - Remote file system mounted, file operations sent on behalf of user across network to server
 - Server checks permissions, file handle returned
 - Handle used for reads and writes until file closed





Distributed Information Systems

- Aka **distributed naming services**, provide unified access to info needed for remote computing
- **Domain name system (DNS)** provides host-name-to-network-address translations for the Internet
- Others like **network information service (NIS)** provide user-name, password, userID, group information
- Microsoft's **common Internet file system (CIFS)** network info used with user auth to create network logins that server uses to allow to deny access
 - **Active directory** distributed naming service
 - **Kerberos-derived** network authentication protocol
- Industry moving toward **lightweight directory-access protocol (LDAP)** as secure distributed naming mechanism





Consistency Semantics

- Important criteria for evaluating file sharing-file systems
- Specify how multiple users are to access shared file simultaneously
 - When modifications of data will be observed by other users
 - Directly related to process synchronization algorithms, but atomicity across a network has high overhead (see Andrew File System)
- The series of accesses between file open and closed called **file session**
- UNIX semantics
 - Writes to open file immediately visible to others with file open
 - One mode of sharing allows users to share pointer to current I/O location in file
 - Single physical image, accessed exclusively, contention causes process delays
- Session semantics (Andrew file system (OpenAFS))
 - Writes to open file not visible during session, only at close
 - Can be several copies, each changed independently





The Sun Network File System (NFS)

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs)
- The implementation originally part of SunOS operating system, now industry standard / very common
- Can use unreliable datagram protocol (UDP/IP) or TCP/IP, over Ethernet or other networks

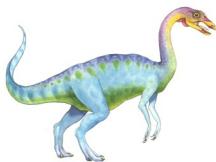




NFS (Cont.)

- Interconnected workstations viewed as a set of independent machines with independent file systems, which allows sharing among these file systems in a transparent manner
 - A remote directory is mounted over a local file system directory
 - ▶ The mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory
 - Specification of the remote directory for the mount operation is nontransparent; the host name of the remote directory has to be provided
 - ▶ Files in the remote directory can then be accessed in a transparent manner
 - Subject to access-rights accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory





NFS (Cont.)

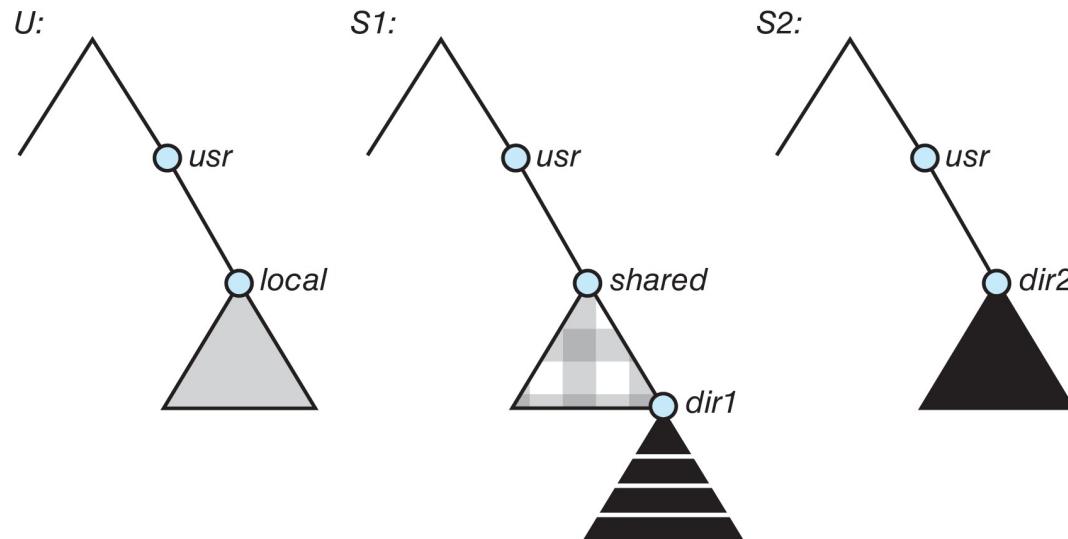
- NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures; the NFS specifications independent of these media
- This independence is achieved through the use of RPC primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces
- The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services





NFS Mounting Example

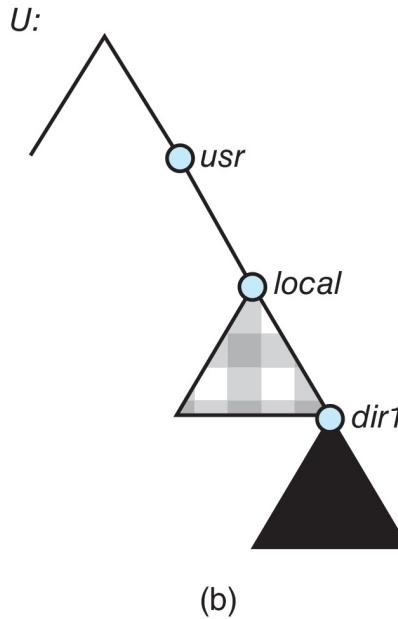
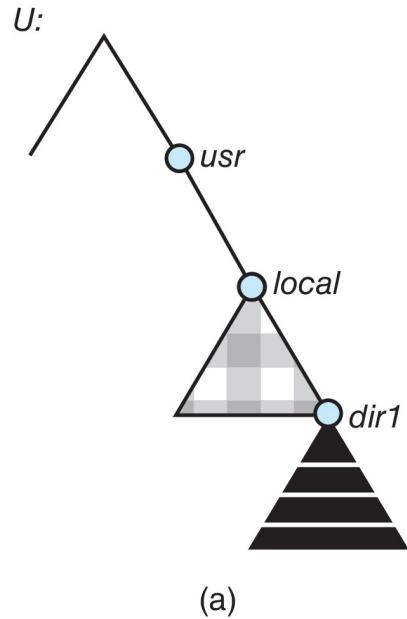
- Three independent file systems





NFS Mounting Example (Cont.)

- Mounts and cascading mounts



Mounts

Cascading mounts





NFS Mount Protocol

- Establishes initial logical connection between server and client
- Mount operation includes name of remote directory to be mounted and name of server machine storing it
 - Mount request is mapped to corresponding RPC and forwarded to mount server running on server machine
 - Export list – specifies local file systems that server exports for mounting, along with names of machines that are permitted to mount them
- Following a mount request that conforms to its export list, the server returns a file handle—a key for further accesses
- File handle – a file-system identifier, and an inode number to identify the mounted directory within the exported file system
- The mount operation changes only the user's view and does not affect the server side





NFS Protocol

- Provides a set of remote procedure calls for remote file operations. The procedures support the following operations:
 - searching for a file within a directory
 - reading a set of directory entries
 - manipulating links and directories
 - accessing file attributes
 - reading and writing files
- NFS servers are **stateless**; each request has to provide a full set of arguments (NFS V4 is newer, less used – very different, stateful)
- Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching)
- The NFS protocol does not provide concurrency-control mechanisms





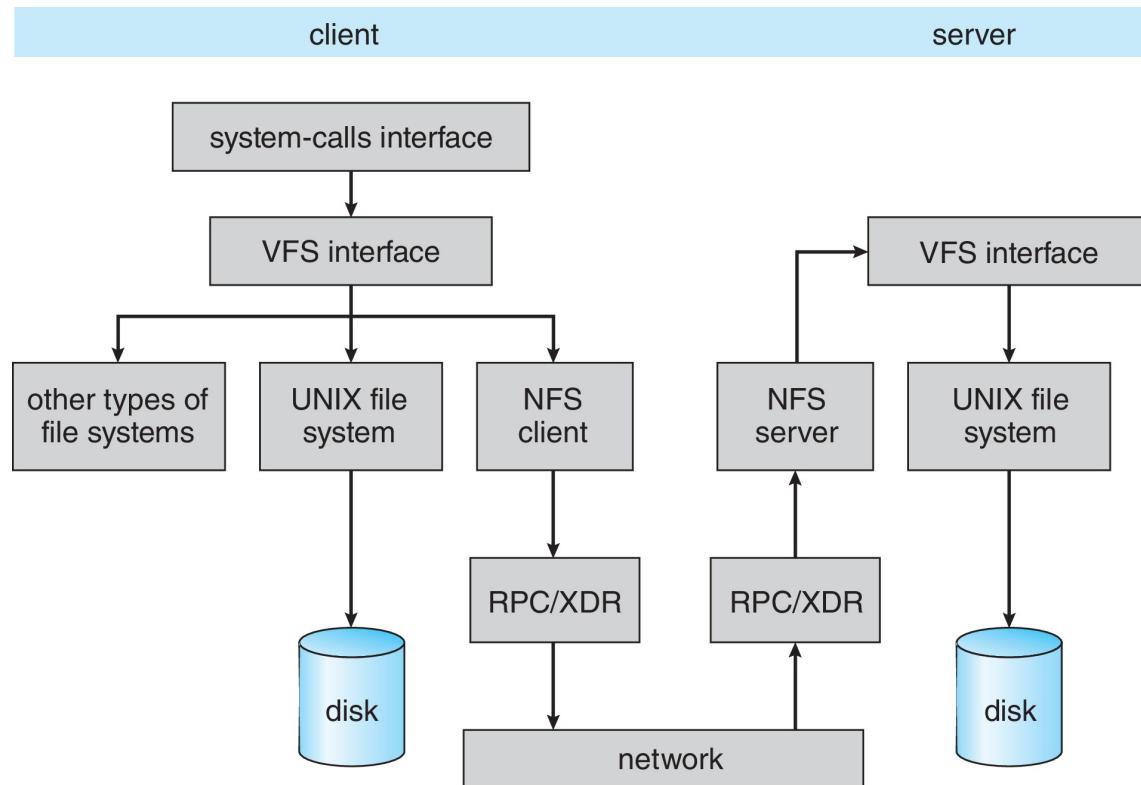
Three Major Layers of NFS Architecture

- UNIX file-system interface (based on the **open**, **read**, **write**, and **close** calls, and **file descriptors**)
- Virtual File System (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types
 - The VFS activates file-system-specific operations to handle local requests according to their file-system types
 - Calls the NFS protocol procedures for remote requests
- NFS service layer – bottom layer of the architecture
 - Implements the NFS protocol





Schematic View of NFS Architecture





NFS Path-Name Translation

- Performed by breaking the path into component names and performing a separate NFS lookup call for every pair of component name and directory vnode
- To make lookup faster, a directory name lookup cache on the client's side holds the vnodes for remote directory names



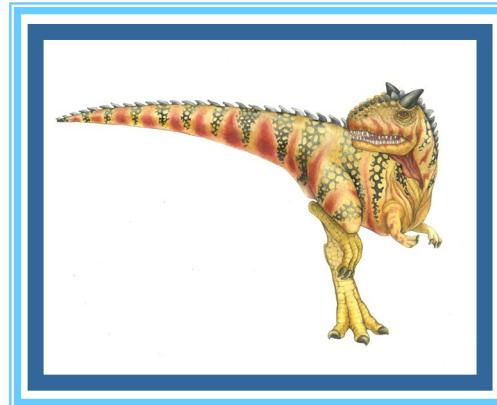


NFS Remote Operations

- Nearly one-to-one correspondence between regular UNIX system calls and the NFS protocol RPCs (except opening and closing files)
- NFS adheres to the remote-service paradigm, but employs buffering and caching techniques for the sake of performance
- File-blocks cache – when a file is opened, the kernel checks with the remote server whether to fetch or revalidate the cached attributes
 - Cached file blocks are used only if the corresponding cached attributes are up to date
- File-attribute cache – the attribute cache is updated whenever new attributes arrive from the server
- Clients do not free delayed-write blocks until the server confirms that the data have been written to disk



End of Chapter 15



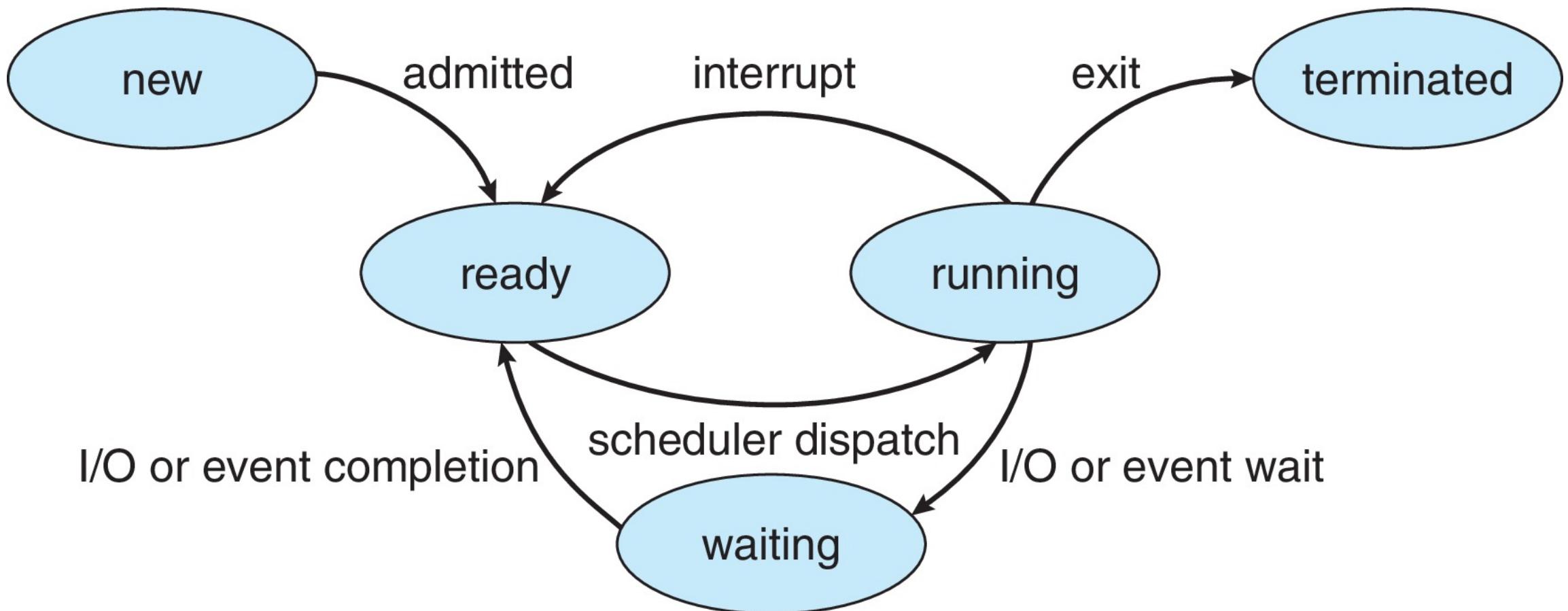
Process State Transition Game.

This is based on one of the games in:

J. M. D. Hill, C. K. Ray, J. R.S. Blair, and C. A. Carver Jr., “Puzzles and Games: Addressing Different Learning Styles in Teaching Operating Systems Concepts,” ACM SIGCSE ‘03, Feb. 2003

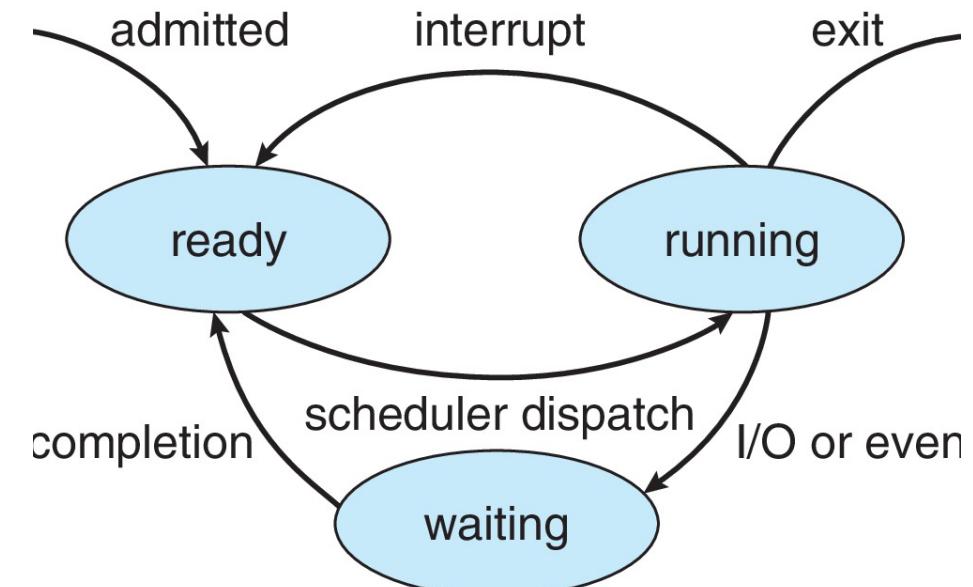


Diagram of Process State



Process States

- We did discuss a five state process transition.
- When memory is not enough a process is *swapped out* of memory.
- We say that the process is *suspended* when it is swapped out of memory.
- A process is suspended when the entire process is not in main memory (no virtual memory).
- **Characteristics of a Suspended Process**
 - The process is not immediately available for execution
 - The process may or may not be waiting on an event



What would be the new states?

- A blocked (or waiting) process may be swapped out of memory
 - > Blocked/ Suspend state is needed to accommodate such processes
Process is waiting for I/O and is not in the memory as well
- A ready process may not be in memory or may be swapped out of memory
 - > Ready/ Suspend State



Step 1



Construct the process state diagram using the 7 states given in next slide.



Clearly mark the state transitions.



Assume a process is suspended when the entire process is not in main memory (no virtual memory).



Block
ed/Su
spend

Exit

Runn
ing

Read
y

Block
ed

Read
y/Sus
pend

New



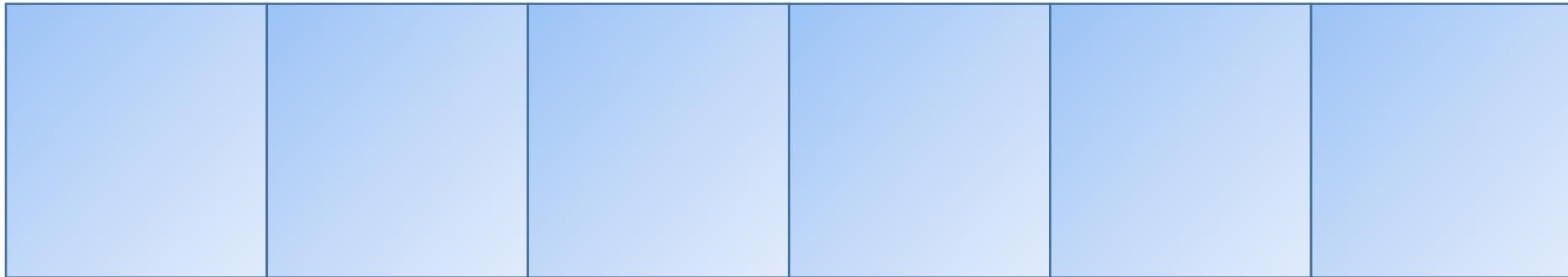
Step 2

- The process execution plan is as follows:

Process A	P3, B4, P2, B5, P3	Memory 3 slots
Process B	P2, B3, P3, B2, P1	Memory 2 slots
Process C	P3, B5, P1, B3, P2	Memory 2 slots

- where P – Process, B – Blocked for I/O, Value - time to spend in processing/blocked.
e.g., P3 – process for 3 time units
- Time quantum is 1 clock tick. You have **only 6 memory slots** as shown in the next slide.
- When the process is suspended, the memory markers are lifted, indicating that the process has been moved out of main memory (usually to make room for another process).
- Execute the processes, using the memory slots given in the next slide. Copy the state transition diagram you constructed in the last slide to the next slide. You can move the instances of the processes to memory slots.
- Construct a timing diagram to calculate the total execution time. Submit your saved slides in pdf format on Process state game Assignment. Also indicate the total execution time as text.





Memory Slots

A
A
A

B
B
B
B

C
C
C
C
C

Time – sequence diagram

You may use something similar to the following table to show the process execution.

You are free to use any other format.





Real time systems



Objectives

- To explain the timing requirements of real-time systems
- To distinguish between hard and soft real-time systems
- To discuss the defining characteristics of real-time systems
- To describe scheduling algorithms for hard real-time systems





What is a Real-Time System?

“A real-time system is a software system that maintains an ongoing and timely interaction with its environment”

Bran Selic

- Real-Time: Pertaining to the performance of a computation during the actual time that the related physical process transpires in order that the results of the computation can be used in guiding the physical process.
[IEEE Standard Dictionary]

What is a Real-Time System?

- “any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement”

The Oxford Dictionary of Computing

- Correctness depends not only on logical result but also time at which results are produced!
[Stankovic'88]
 - timeliness = correctness
- Current system output depends upon “timed behavior” of input.

Types of Real-Time Requirements

- Hard real time
 - timing requirement must always be met (e.g. some reactor trips)
 - Missing deadlines means system failure
- Soft real time
 - timeliness of events is important but not necessarily critical.
 - occasionally missing a deadline does not result in system failure (e.g. RT-video, user interface, data logging, even many control loops!)

Types of Real-Time Requirements

- Real-time systems are frequently classified as hard or soft real-time systems.
 - A hard real-time system has time-critical deadlines that must be met; otherwise a catastrophic system failure can occur.
 - In a soft real-time system, it is considered undesirable, but not catastrophic, if deadlines are occasionally missed.
- Safety Critical
 - Missing deadlines can cause serious loss



Overview of Real-Time Systems

- A **real-time system** requires that results be produced within a specified deadline period
- An **embedded system** is a computing device that is part of a larger system (I.e. automobile, airliner)
- A **safety-critical system** is a real-time system with catastrophic results in case of failure
- A **hard real-time system** guarantees that real-time tasks be completed within their required deadlines
- A **soft real-time system** provides priority of real-time tasks over non real-time tasks



Characteristics of Real-Time Systems

- Real-time systems tend to be:
 - Embedded Systems: i.e. a component of a larger hardware/software system.
 - Interact with an external “environment” (i.e. the “plant” or system under control)
 - Reactive Systems: event-driven and must respond to external stimuli (“events”). System response is typically state dependent.
 - Concurrent Processing: many events that need to be processed in parallel.
 - Safety Critical

Real Time Applications

■ Traditional Real Time Applications

- Avionics
- Data Acquisition
- Digital Controllers
- Defense Systems
- Industrial Process Control



Real Time Applications

- Emerging Embedded Real-Time Applications
 - Cell-phones, VoIP phone, PDA's
 - MP3 players
 - Set-top boxes, Game Consoles
 - Automotive Systems
 - Network Boxes
 - On Demand Servers
 - Embedded Devices



System Characteristics

- Single purpose
- Small size
- Inexpensively mass-produced
- Specific timing requirements





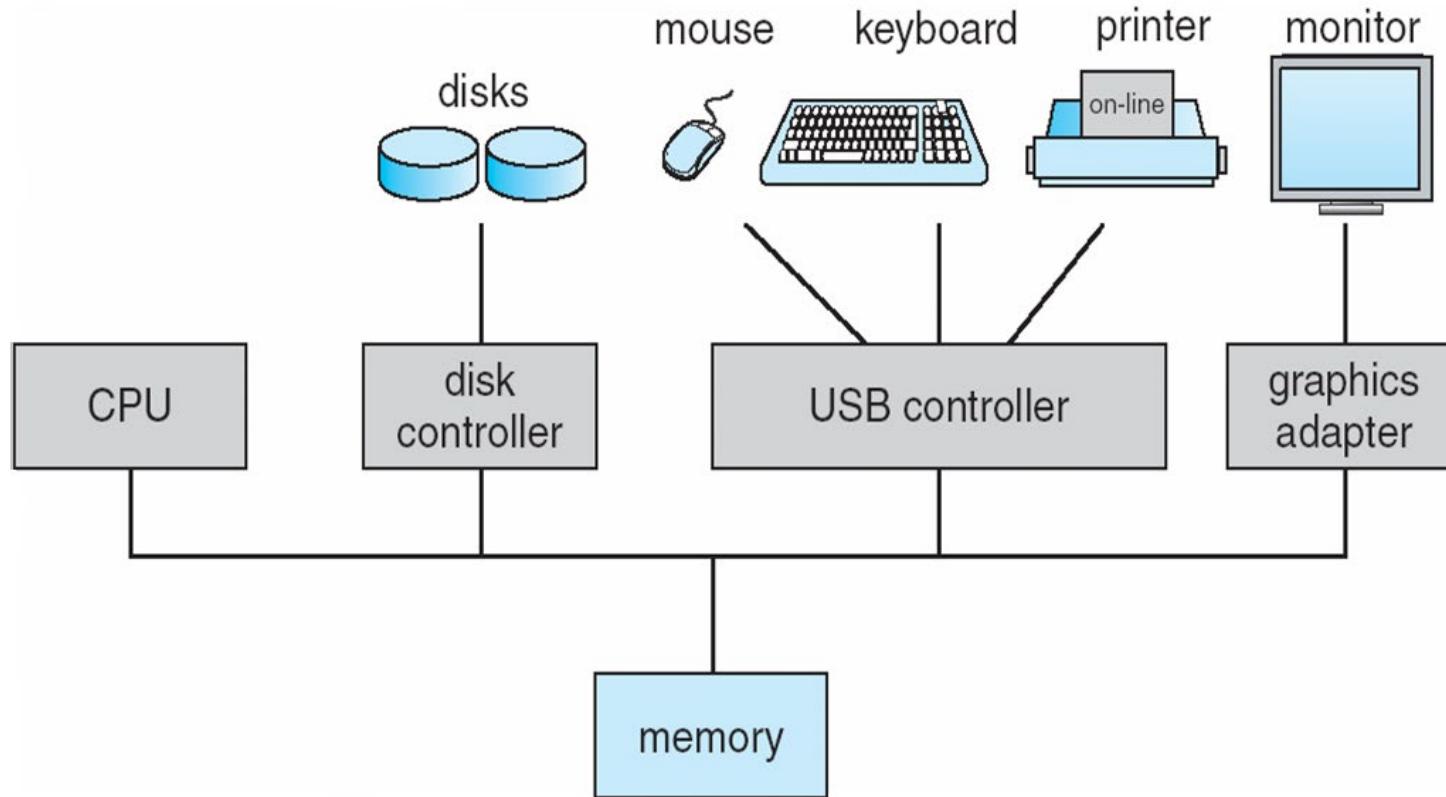
System-on-a-Chip

- Many real-time systems are designed using system-on-a-chip (SOC) strategy
- SOC allows the CPU, memory, memory-management unit, and attached peripheral ports (I.e. USB) to be contained in a single integrated circuit





Bus-Oriented System





Traditional Real-Time OS Requirements

- Multi-Tasking kernel
 - Fixed priority, preemptive scheduling
- Inter-task communication
 - Semaphores, Message Queues, etc.
 - Priority Inheritance
- Timers
- Device Drivers
- Interrupt Management

Real time O/S Functions

- Task management
 - scheduling, dispatcher
 - communication (pipe, queue)
 - synchronization (semaphore, event)
- Memory management
- Time management
- Device driver
- Interrupt service



Features of Real-Time Kernels

- Most real-time systems do not provide the features found in a standard desktop system
- Reasons include
 - Real-time systems are typically single-purpose
 - Real-time systems often do not require interfacing with a user
 - Features found in a desktop PC require more substantial hardware than what is typically available in a real-time system





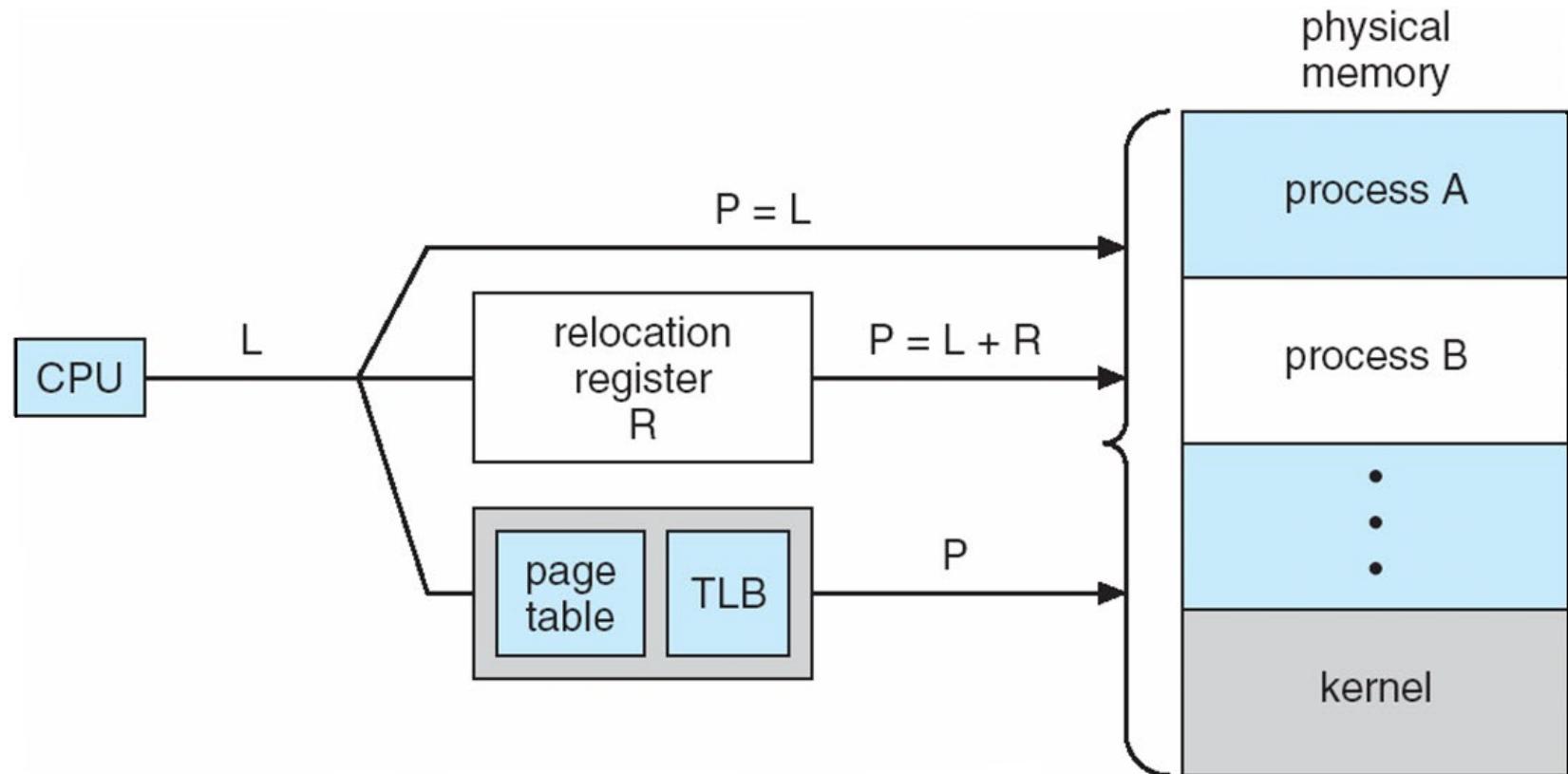
Virtual Memory in Real-Time Systems

- Address translation may occur via:
 - (1) Real-addressing mode where programs generate actual addresses
 - (2) Relocation register mode
 - (3) Implementing full virtual memory





Address Translation



Real-Time OS: Examples

- Traditional Real-Time OSes
 - *VxWorks, pSOS, QNX, LynxOS*
- Minimal Real-Time OSes
 - *ThreadX, Ercos, Nucleus*
- General Purpose OS Made Real-Time
 - *Solaris, NT, Linux*

Response Time Factors

- Execution Time
 - *Amount of time taken in the absence of resource contention*
- Higher Priority Non Schedulable Entities
 - *Interrupt Handlers*
- Scheduling
 - *Time taken by higher priority work*
- Shared Resources
 - *Time taken by lower priority work*
 - Often, due to contention on some shared resource



Implementing Real-Time Systems

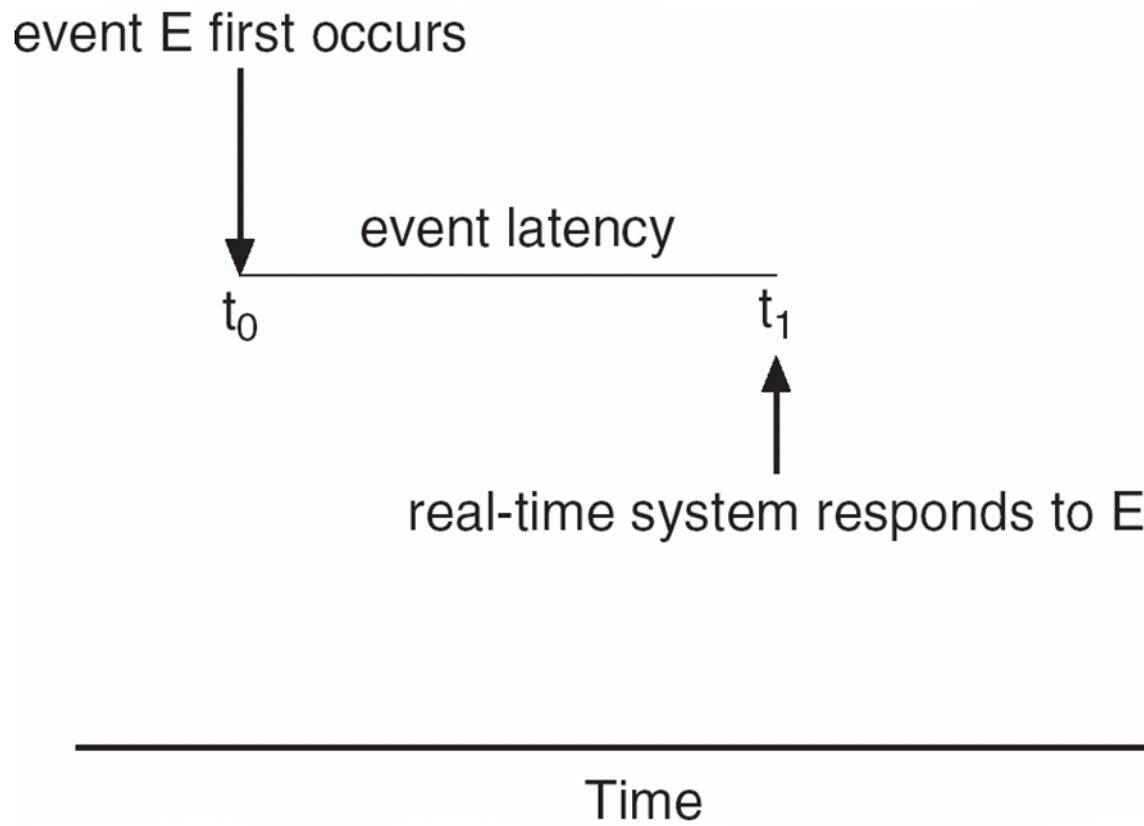
- In general, real-time operating systems must provide:
 - (1) Preemptive, priority-based scheduling
 - (2) Preemptive kernels
 - (3) Latency must be minimized





Minimizing Latency

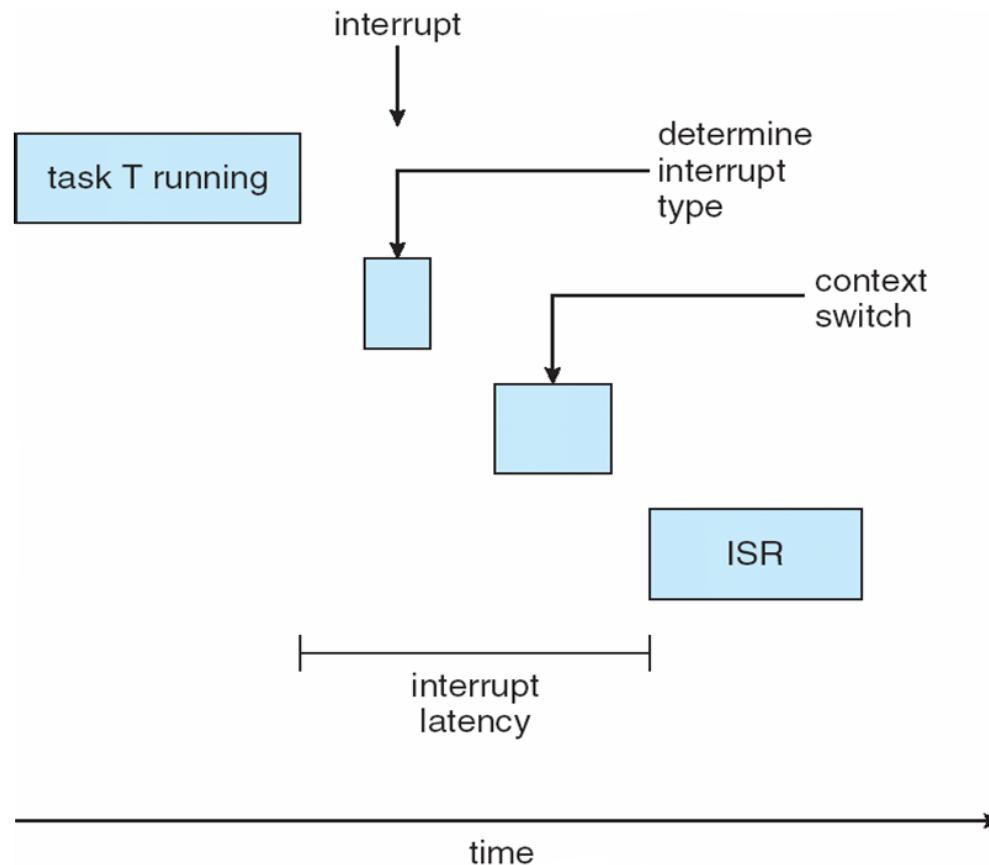
- **Event latency** is the amount of time from when an event occurs to when it is serviced.





Interrupt Latency

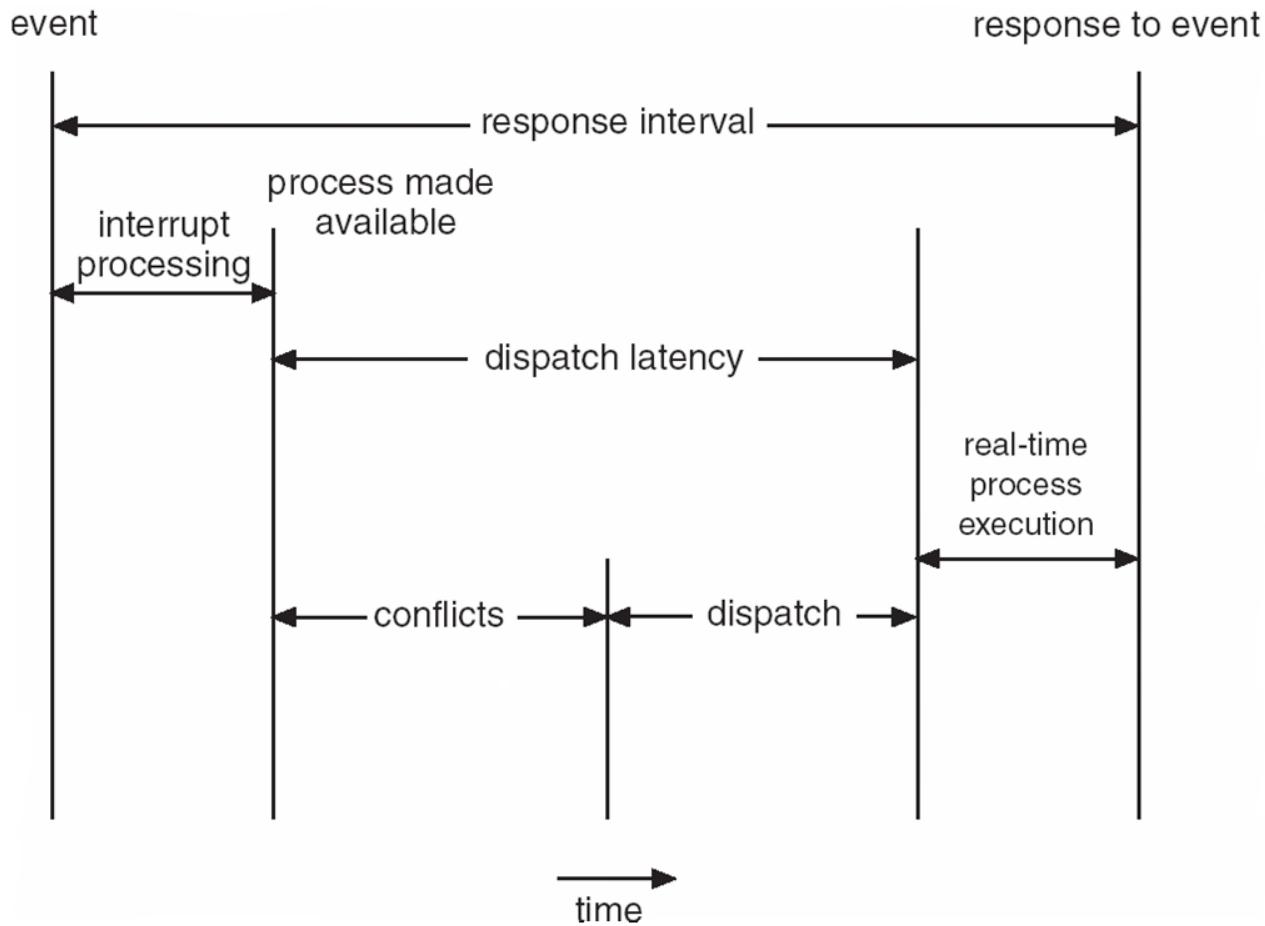
- Interrupt latency is the period of time from when an interrupt arrives at the CPU to when it is serviced





Dispatch Latency

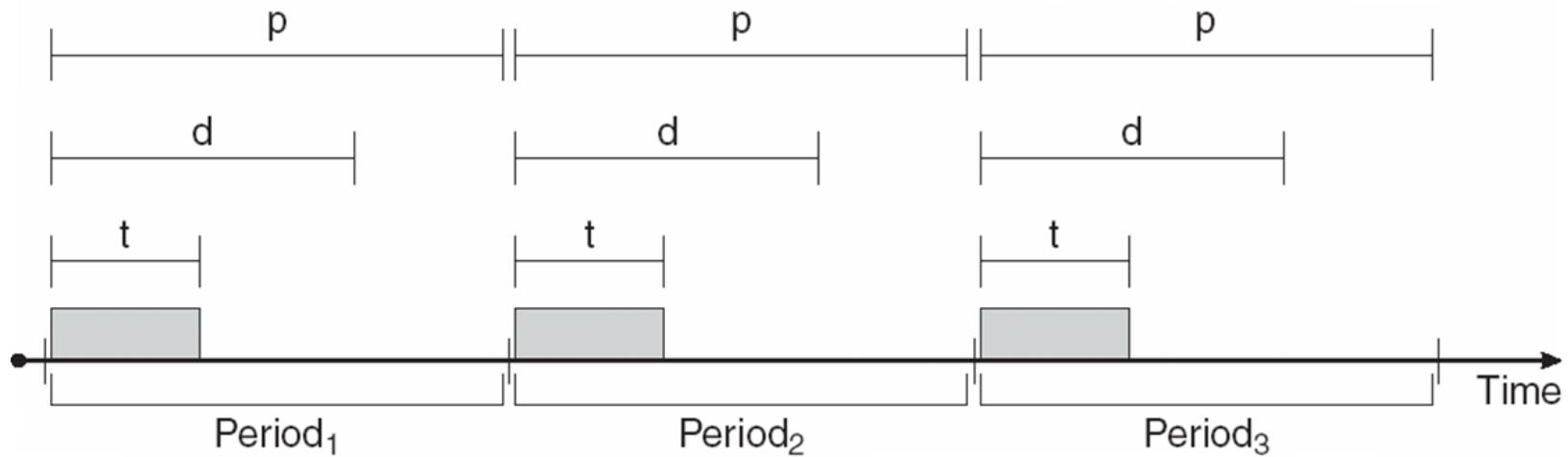
- Dispatch latency is the amount of time required for the scheduler to stop one process and start another





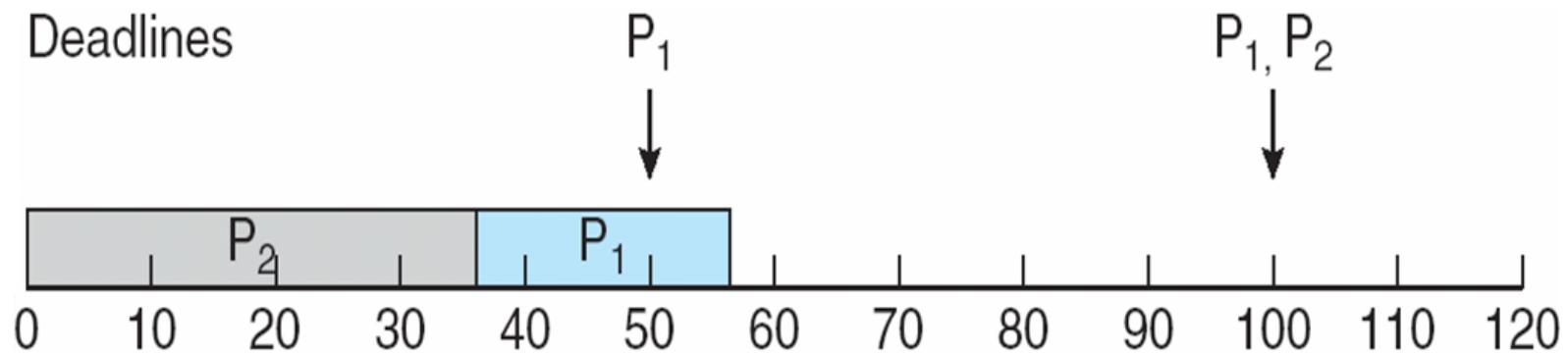
Real-Time CPU Scheduling

- Periodic processes require the CPU at specified intervals (periods)
- p is the duration of the period
- d is the deadline by when the process must be serviced
- t is the processing time





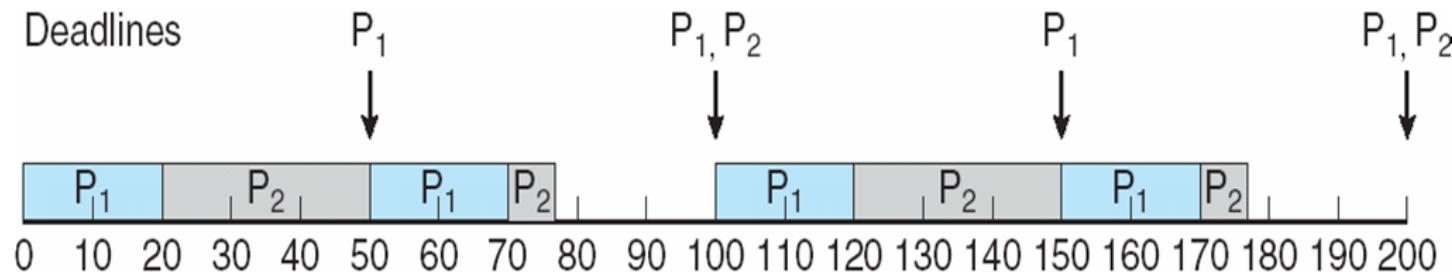
Scheduling of tasks when P_2 has a higher priority than P_1





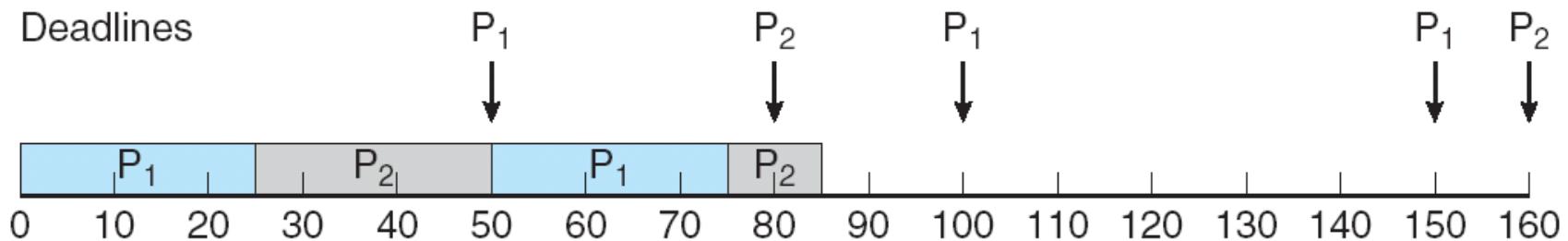
Rate Montonic Scheduling

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





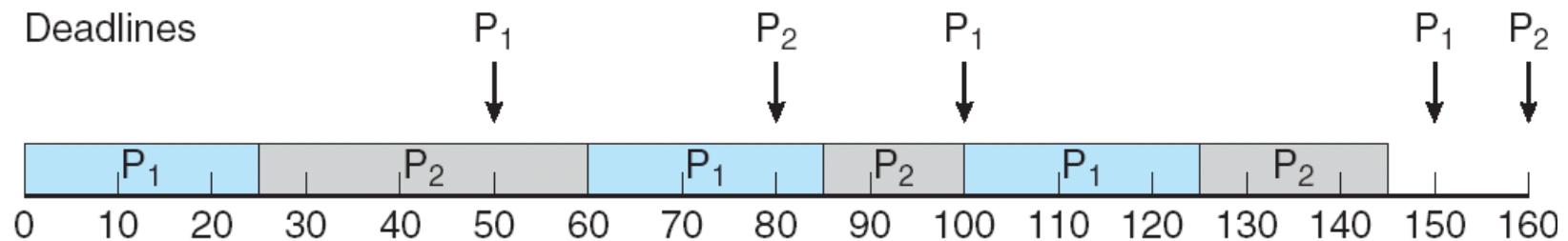
Missed Deadlines with Rate Monotonic Scheduling





Earliest Deadline First Scheduling

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





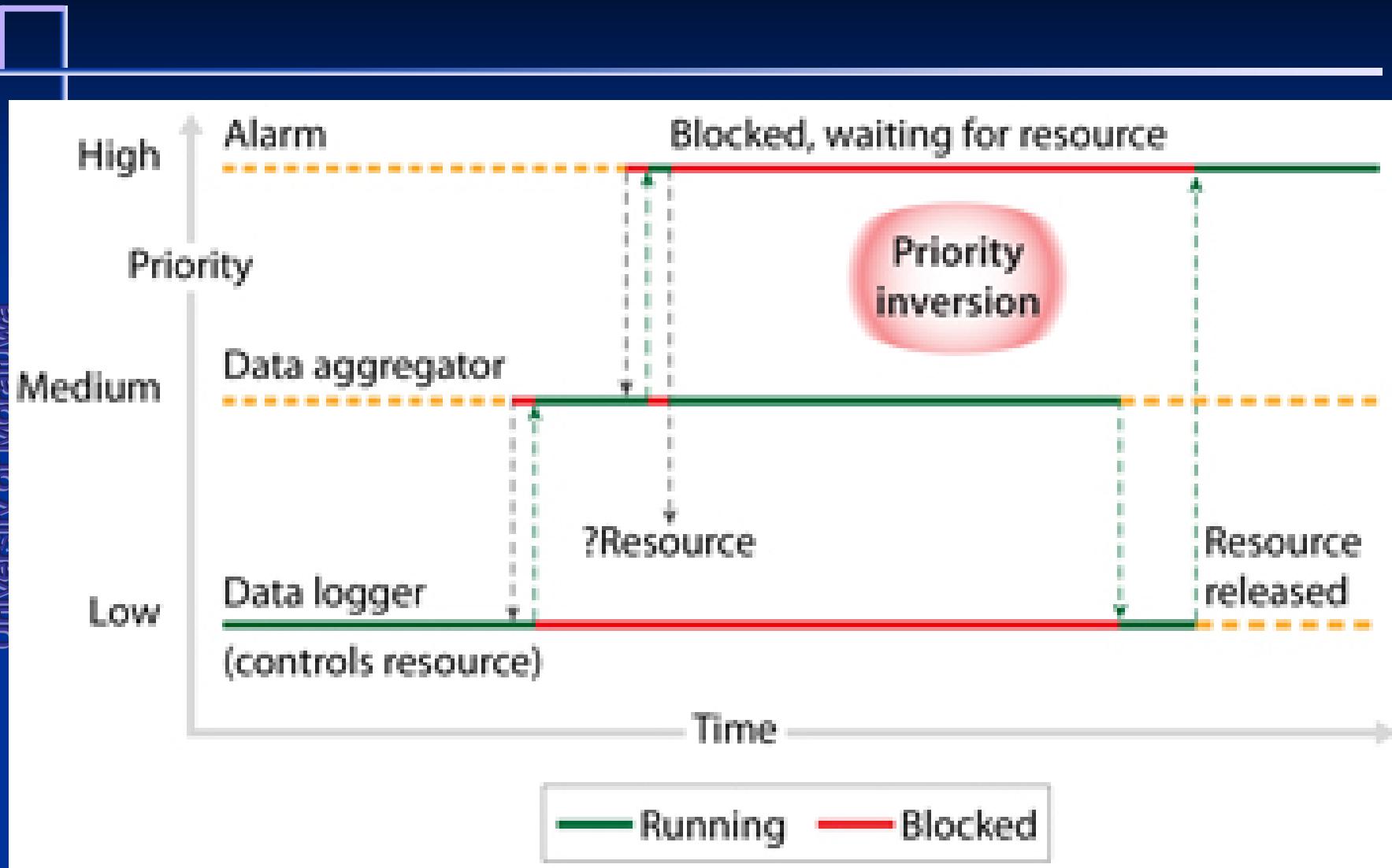
Proportional Share Scheduling

- 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



Pathfinder Mars Rover

- Landing: July 4, 1997; initial successes
- Intermittent software system resets
 - Delay of mission, serious loss of data
 - Happens when “too much” data are sent over a shared information bus
 - Low priority data collection task locks the bus, gets interrupted by medium priority tasks
 - High priority data distribution task fails to complete: cannot get shared bus
 - Scheduler detects pending high priority task and resets all the hardware and software



Priority Inversion Problem

Priority inversion is an undesirable situation in which a higher priority task gets blocked (waits for CPU) for more time than that it is supposed to, by lower priority tasks.

Example:

- Let T_1 , T_2 , and T_3 be the three periodic tasks with decreasing order of priorities.
- Let T_1 and T_3 share a resource S.

Priority Inversion - Example

- T3 obtains a lock on the semaphore S and enters its critical section to use a shared resource.
- T1 becomes ready to run and preempts T3. Then, T1 tries to enter its critical section by first trying to lock S. But, S is already locked by T3 and hence T1 is blocked.
- T2 becomes ready to run. Since only T2 and T3 are ready to run, T2 preempts T3 while T3 is in its critical section.

Ideally, one would prefer that the highest priority task (T1) be blocked no longer than the time for T3 to complete its critical section. However, the duration of blocking is, in fact, unpredictable because task T2 got executed in between.

Priority Inversion example

T1
Highest priority

T2
Medium priority

T3
Least priority

T1 and T3 share resource S

Makes a request for resource S and gets blocked

A higher priority task waits for a lower priority task

T2 completes

T3 completes

Preempted by higher priority task T1

Preempt higher priority task T2

$$\text{Total blocking time for task T1} = (K_2 + K_3) + (L_1)$$

Resource S is shared and T1 is scheduled before

Priority Inheritance Protocol

Priority inheritance protocol solves the problem of priority inversion.

Under this protocol, if a higher priority task T_H is blocked by a lower priority task T_L , because T_L is currently executing critical section needed by T_H , T_L temporarily inherits the priority of T_H .

When blocking ceases (i.e., T_L exits the critical section), T_L resumes its original priority.

Unfortunately, priority inheritance may lead to deadlock.

Priority Inheritance Protocol – Deadlock

Assume T2 has higher priority than T1

task	operation sequence on critical section
T_1	$Lock(CS_2)$ $Lock(CS_1)$ $Unlock(CS_1)$ $Unlock(CS_2)$
T_2	$Lock(CS_1)$ $Lock(CS_2)$ $Unlock(CS_2)$ $Unlock(CS_1)$

time	task	action
t_0	T_1	starts execution
t_1	T_1	locks CS_2
t_2	T_2	activated and preempts T_1 due to its higher priority
t_3	T_2	locks CS_1
t_4	T_2	attempts to lock CS_2 , but is blocked because T_1 has a lock on it
t_5	T_1	inherits the priority of T_2 and starts executing
t_6	T_1	attempts to lock CS_1 , but is blocked because T_2 has a lock on it
$\geq t_7$	-	both the tasks cannot proceed (deadlocked)

Figure 1: Example for priority inheritance protocol resulting in deadlock

Priority Ceiling Protocol

- Priority ceiling protocol solves the priority inversion problem without getting into deadlock.
- For each semaphore, a *priority ceiling* is defined, whose value is the highest priority of all the tasks that may lock it.
- When a task T_i attempts to execute one of its critical sections, it will be suspended unless its priority is higher than the priority ceiling of all semaphores currently locked by tasks other than T_i .
- If task T_i is unable to enter its critical section for this reason, the task that holds the lock on the semaphore with the highest priority ceiling is said to be blocking T_i and hence inherits the priority of T_i .

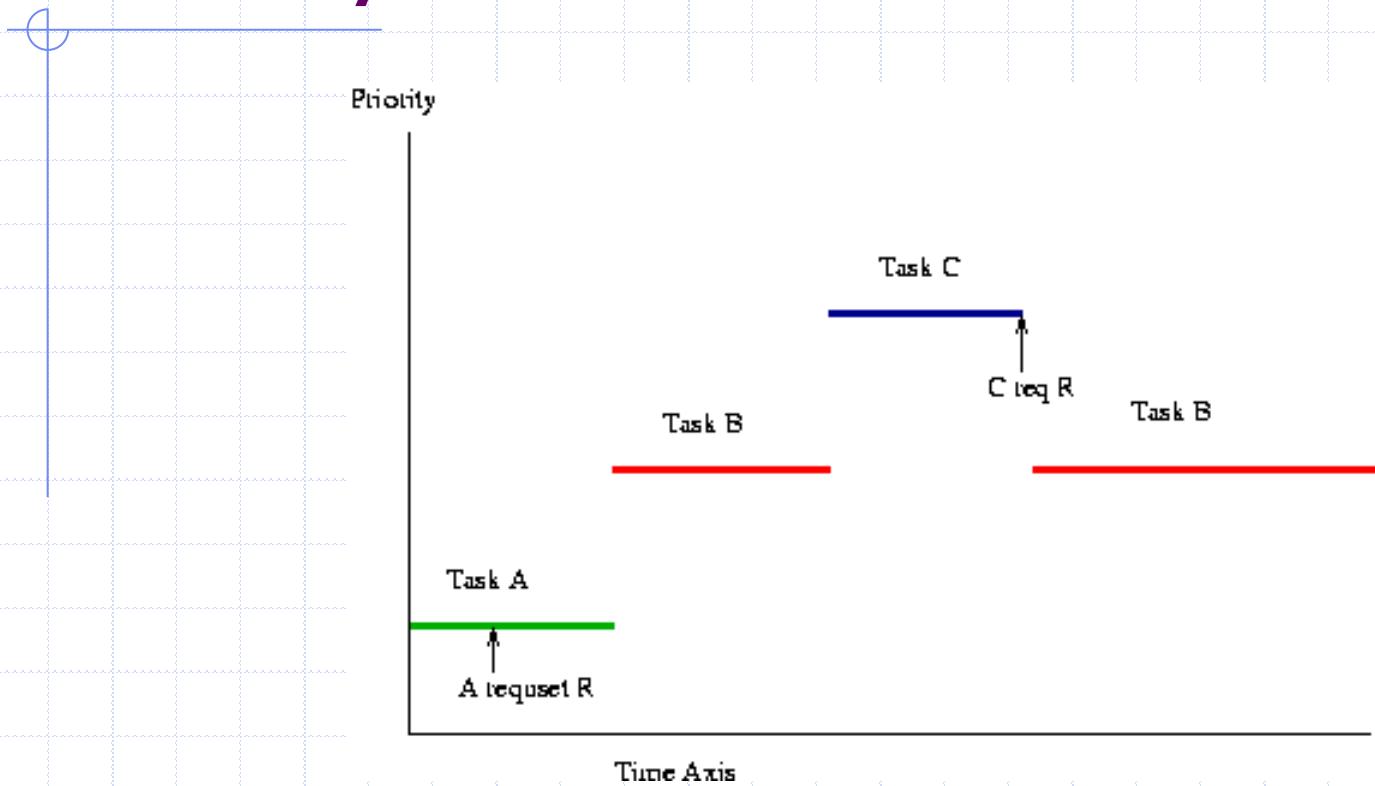
Priority Ceiling Protocol - properties

- This protocol is the same as the priority inheritance protocol, except that a task T_i can also be blocked from entering a critical section if any other task is currently holding a semaphore whose priority ceiling is greater than or equal to the priority of task T_i .

Priority Ceiling Protocol - Example

- For the previous example, the priority ceiling for both CS_1 and CS_2 is the priority of T_2 .
- From time t_0 to t_2 , the operations are the same as before.
- At time t_3 , T_2 attempts to lock CS_1 , but is blocked since CS_2 (which has been locked by T_1) has a priority ceiling equal to the priority of T_2 .
- Thus T_1 inherits the priority of T_2 and proceeds to completion, thereby preventing deadlock situation.

Priority Inversion



Solutions to Priority Inversion

◆ Non Blocking Critical Section

- Higher priority Thread may get blocked by unrelated low priority thread

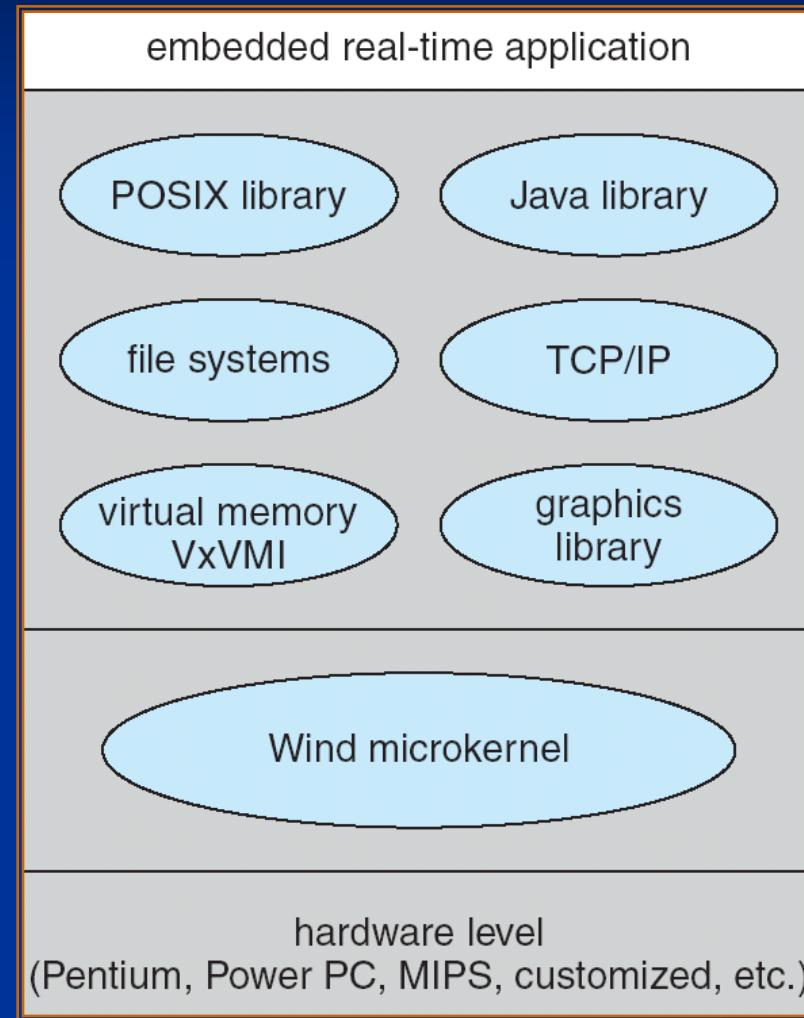
◆ Priority Ceiling

- Each resource has an assigned priority
- Priority of thread is the highest of all priorities of the resources it's holding

◆ Priority Inheritance

- The thread holding a resource inherits the priority of the thread blocked on that resource

VxWorks 5.0



Wind Microkernel

- The Wind microkernel provides support for the following:
 - (1) Processes and threads;
 - (2) preemptive and non-preemptive round-robin scheduling;
 - (3) manages interrupts (with bounded interrupt and dispatch latency times);
 - (4) shared memory and message passing interprocess communication facilities.



The Real-Time Linux Challenge

- How to leverage the advantages of Linux, while making it suitable for real-time systems?

Approaches to Real-Time Linux

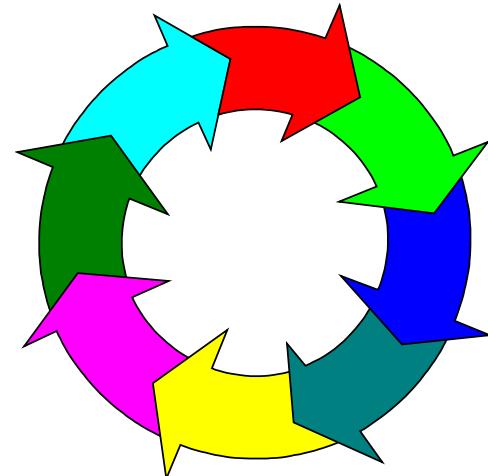
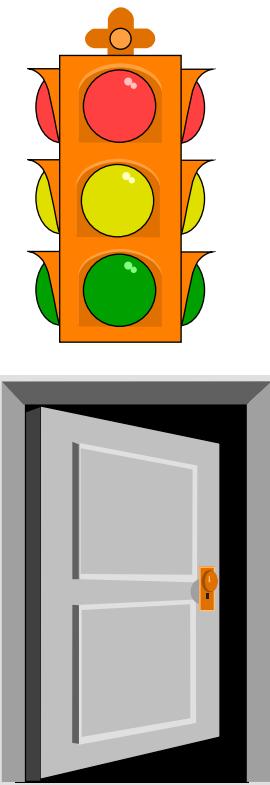
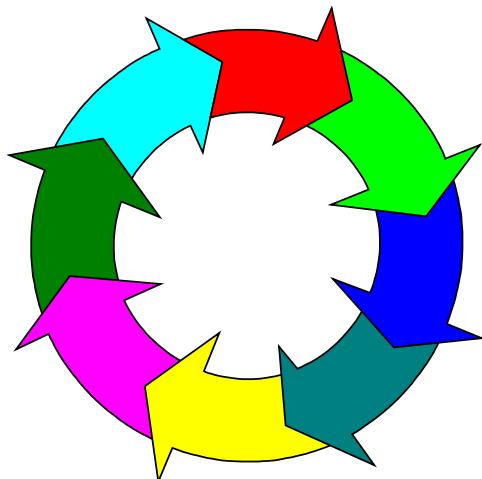
- Approaches limiting Real-time and Non Real-time Task Interactions
 - *Compliant Kernel Approach*
 - LynxOS/Blue Cat Linux
 - *Thin Kernel Approach*
 - RTLinux/RTAI

Approaches to Real-Time Linux

- Approaches that integrate Real-time and Non Realtime tasks
 - *Core Kernel Approach*
 - TimeSys Linux/RT, Hard Hat Linux
 - *Resource Kernel Approach*
 - TimeSys Linux/RT

Slides are based mostly
on Professor N S Scott's (Queens University, Belfast) slides

Semaphores



Semaphores

A railway semaphore is a signal flag that indicates whether or not the track ahead is clear or occupied by another train. As the train proceeds, semaphores are set and cleared. They are a mechanism which ensures mutual exclusive occupancy of critical section of track.

Previously we considered algorithms which run on a **bare machine**, i.e. they use only ordinary sequential programming constructs. Here we introduce the notion of a **semaphore** (first described by Dijkstra in 1968) as a programming primitive on a higher level than machine instructions.

In concurrent programs semaphores provide a basic signalling mechanism to implement mutual exclusion and condition synchronization.

Definition of a Semaphore

A **semaphore** is an integer-valued variable which can take only non-negative values. Exactly two operations are defined on a semaphore S :

down(S)

if $S > 0$ then

$S = S - 1$

else

suspend the action of the process making the call. The process is said to be suspended on S .

up(S)

if there are processes that have been suspended on S

wake one of them.

else

$S = S + 1$

Definition of a Semaphore

A semaphore has the following **properties**:

- ◆ $\text{down}(S)$ and $\text{up}(S)$ are atomic instructions. In particular, no instructions can be interleaved between the test that $S > 0$ and the decrement of S or the suspension of the calling process.
- ◆ A semaphore must be given a non-negative initial value.
- ◆ The $\text{up}(S)$ operation must awaken one of the suspended processes. The definition does not specify which process must be awakened.

A **general semaphore** is a semaphore which can take any non-negative value.

A **binary semaphore** is a semaphore which can take only the values 0 and 1. In this case the definition of $\text{up}(S)$ may be replaced by **if ... else $S = 1$** .

Definition of a Semaphore

Show that a semaphore S satisfies the following two invariants,

$$S \geq 0 \quad (1)$$

$$S = S_0 + \#Up(S) - \#Down(S) \quad (2)$$

where,

S_0 is the initial value of the semaphore,

$\#Up(S)$ is the number of Up operations executed on S ,

$\#Down(S)$ is the number of completed Down operations executed on S .

Mutual Exclusion using a Semaphore

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

Process P1

```
while (true) {  
    nonCriticalSection1;  
    down(S);  
    criticalSection1;  
    up(S);  
}  
end P1;
```

Process P2

```
while (true) {  
    nonCriticalSection2;  
    down(S);  
    criticalSection2;  
    up(S);  
}  
end P2;
```

This algorithm is similar to the second attempt (see earlier) except that the atomic implementation of the semaphore instruction prevents interleaving between the test of *S* and the assignment to *S*.

It differs from the test and set instruction in that a process suspended on a semaphore no longer executes instructions checking variables in a busy-wait loop.

Mutual Exclusion using a Semaphore

The above algorithm generalises to N processes, $N > 1$. In addition it satisfies all the desirable properties described earlier.

Mutual exclusion is satisfied

Proof: Let $\#CS$ be the number of processes in their critical sections.
We will prove that

$$\#CS + S = 1 \quad (3)$$

is invariant. Then from invariant (1) $\#CS \leq 1$, which proves the mutual exclusion property.

Proof that (3) is invariant.

1. $\#CS = \#Down(S) - \#Up(S)$ (show that this is an invariant)
2. $S = 1 + \#Up(S) - \#Down(S)$ (2)
3. $S = 1 - \#CS$

Mutual Exclusion using a Semaphore

The solution cannot deadlock

Proof: For the program to deadlock both processes must be suspended on Down(S). Then $S=0$ and $\#CS=0$. This contradicts the invariant (3) $\#CS + S = 1$, which is impossible.

There is no individual starvation (For $N=2$)

Proof: If P_1 is suspended, $S=0$. By the invariant (3) P_2 is in the critical section. When P_2 exits the critical section it will execute up(S) which will wake some process suspended on S . P_1 is the only process suspended on S .

There is no starvation in the absence of contention (For $N=2$)

Proof: In absence of contention, $S=1$ and no single process will be delayed.

Other Semaphore Definitions

Blocked-set semaphore :

down(S)

if $S > 0$ then
 $S = S - 1$

else

suspend the action of the process making the call.

up(S)

if there are processes that have been suspended on S
 wake one of them.

else

$S = S + 1$

A signaling process awakens one (unspecified) of the suspended processes.

Other Semaphore Definitions

Blocked-queue semaphore :

down(S)

if $S > 0$ then

$S = S - 1$

else

suspend the action of the process making the call and
append to a FIFO queue.

up(S)

if there are processes that have been suspended on S
wake the process at the head of the queue.

else

$S = S + 1$.

The suspended processes are kept in a FIFO queue and
awakened in the same order they were suspended.

Other Semaphore Definitions

Busy-wait semaphore :

```
down(S)
  loop
    <if (S>0) {
      S=S-1; break;
    }>
  end loop;
up(S)
  S = S + 1.
```

The value of S is tested in a busy-wait loop. The entire if-statement is executed as an atomic operation, but there may be interleaving between cycles of the loop.

Other Semaphore Definitions

Note that satisfaction of the desirable mutual exclusion properties depends on the semaphore definition chosen. In particular, we observe the following results.

For a busy-wait semaphore, starvation is possible.

Consider the following execution sequence:

1. P1 executes down(S) and enters its CS.
2. P2 finds $S=0$ and loops.
3. P1 completes an entire cycle consisting of post-protocol, non-CS, pre-protocol and re-enters its CS.
4. P2 finds $S=0$ and loops.

Other Semaphore Definitions

For a blocked-queue semaphore, starvation is impossible.

Suppose P1 is blocked on S. Then there are at most $N - 1$ processes ahead of P1 on the queue for S. Thus P1 will enter its CS after at most $N - 1$ more executions of up (S).

For a blocked-set semaphore, starvation is possible for $N > 2$.

For $N=2$ we showed that starvation is impossible. For $N > 2$ it is possible to construct an execution sequence where there will always be at least two processes blocked on S. Since up (S) is only required to wake one arbitrary process, it can conspire to ignore one process, causing starvation. The construction is left as an exercise.

Exercise

Bill and Ben work in a cafe that specialises in a soup and sandwich combination.

Bill makes two rounds of sandwiches while, at the same time, Ben makes two bowls of soup. When the sandwiches and soup are ready Bill and Ben each, at the same time, serve a soup and sandwich combination to a customer. These actions are repeated throughout the day.

[Note: Bill cannot serve until Ben has made the soup and Ben cannot serve until Bill has made the sandwiches.]

Exercise

Let Bill and Ben be represented by two processes with the following structure:

Variable declarations

```
process Bill {  
    while (true) {  
        // Bill makes sandwiches //  
        SYNCH 1;  
        // Bill delivers //  
    }  
}
```

```
process Ben {  
    while (true) {  
        // Ben makes soup //  
        SYNCH 2;  
        // Ben delivers //  
    }  
}
```

Provide variable declarations (including any initialization) and code for **SYNCH 1** and **SYNCH 2** using semaphores so that the above algorithm behaves correctly.

Exercise

Solution

```
Semaphore first = 0;  
Semaphore second = 0;
```

```
process Bill {  
    while (true) {  
        // Bill makes sandwiches //  
        down(first);  
        up(second);  
        // Bill delivers //  
    }  
}
```

```
process Ben {  
    while (true) {  
        // Ben makes soup //  
        up(first);  
        down(second);  
        // Ben delivers //  
    }  
}
```

Exercise

OR ...

```
Semaphore first = 0;  
Semaphore second = 0;
```

```
process Bill {  
    while (true) {  
        // Bill makes sandwiches //  
        up(second);  
        down(first);  
        // Bill delivers //  
    }  
}
```

```
process Ben {  
    while (true) {  
        // Ben makes soup //  
        down(second);  
        up(first);  
        // Ben delivers //  
    }  
}
```

Exercise

```
Semaphore first = 0;  
Semaphore second = 0;
```

```
process Bill {  
    while (true) {  
        // Bill makes sandwiches //  
        up(second);  
        down(first);  
        // Bill delivers //  
    }  
}
```

```
process Ben {  
    while (true) {  
        // Ben makes soup //  
        up(first);  
        down(second);  
        // Ben delivers //  
    }  
}
```

Producer – Consumer Problem

The mutual exclusion problem is an abstraction of **synchronization** problems common in computer systems. Here we consider an abstraction of **communication** problems called the producer-consumer problem.

Producer: By executing an internal operation, **produce**, this process creates a data element which must be sent to the consumers.

Consumer: Upon receipt of a data element, this process performs some computation in an internal operation **consume**.

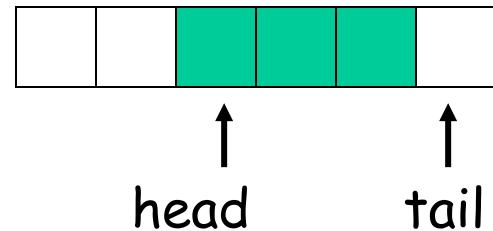
For example this models an application **producing** a listing that must be **consumed** by a printer.

Producer – Consumer Problem

Whenever a data item is sent from one process to another an elementary solution is **synchronous communication** : when one process is ready to send and the other ready to receive the data is sent.

If more flexibility is required a **buffer** is used to allow **asynchronous communication**. The sending process appends data elements to the tail of the queue, the receiving process removes an element from the head of the queue.

A buffer is a queue of data elements



Producer – Consumer Problem: infinite buffer

We begin by assuming an infinite buffer.

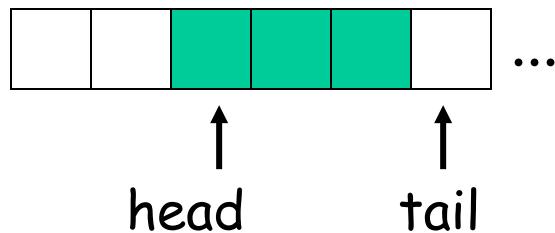
```
integer [] buffer = new integer [infinite];
integer head =0, tail = 0;
```

```
process producer
integer i;
while (true) {
    i=produce();
    buffer[tail]=i;
    tail=tail+1;
}
end producer;
```

```
process consumer
integer i;
while (true) {
    <wait until (tail > head);>
    i=buffer[head];
    head=head+1;
    consume(i);
}
end consumer;
```

Mutual exclusion is not an issue. Why ?

Producer – Consumer Problem: infinite buffer



The buffer has the following invariants:

$$\#E = 0 + \text{tail} - \text{head}$$

$$E \geq 0$$

This suggests that the problem can be solved using a **semaphore**, '**elements**', that counts the number of data elements in the buffer.

Producer – Consumer Problem: infinite buffer

```
integer [] buffer = new integer [infinite];
integer head =0, tail = 0;
Semaphore elements = new Semaphore(0);
```

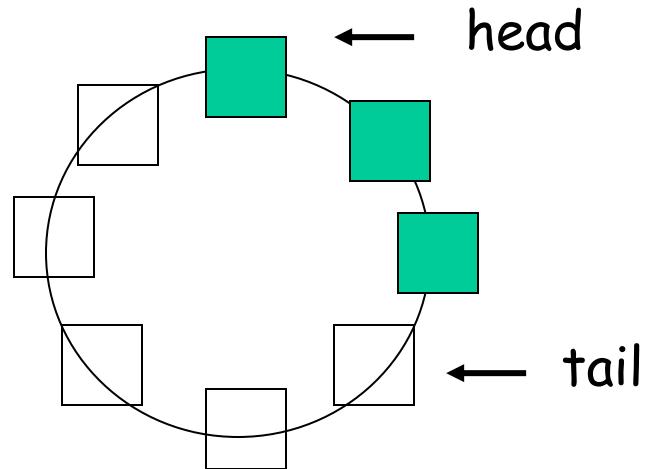
```
process producer
integer i;
while (true) {
    i=produce();
    buffer[tail]=i;
    tail=tail+1;
    up(elements);
}
end producer;
```

```
process consumer
integer i;
while (true) {
    down(elements);
    i=buffer[head];
    head=head+1;
    consume(i);
}
end consumer;
```

The consumer executes **down(elements)** which suspends if the number of elements in the buffer is 0. At the end of each loop iteration the producer executes **up(elements)**.

Producer – Consumer Problem: circular buffer

We now extend the previous solution to the more practical situation of a **bounded-buffer**. In particular, we consider a circular buffer.



In this example semaphores serve as resource counters. One counts the number of empty buffer slots and the other counts the number of occupied slots.

The synchronization problems are to:

- ◆ suspend the producer when the buffer is full;
- ◆ to suspend the consumer when the buffer is empty;
- ◆ to make sure that only one process at a time manipulates a buffer slot (this is left as an exercise).

Producer – Consumer Problem: circular buffer

```
integer [] buffer = new integer [N];
integer head =0, tail = 0;
Semaphore elements = new Semaphore(0);
Semaphore spaces = new Semaphore(N);
```

```
process producer
integer i;
while (true) {
    down(spaces);
    i=produce();
    buffer[tail]=i;
    tail=(tail+1)% N;
    up(elements);
}
end producer;
```

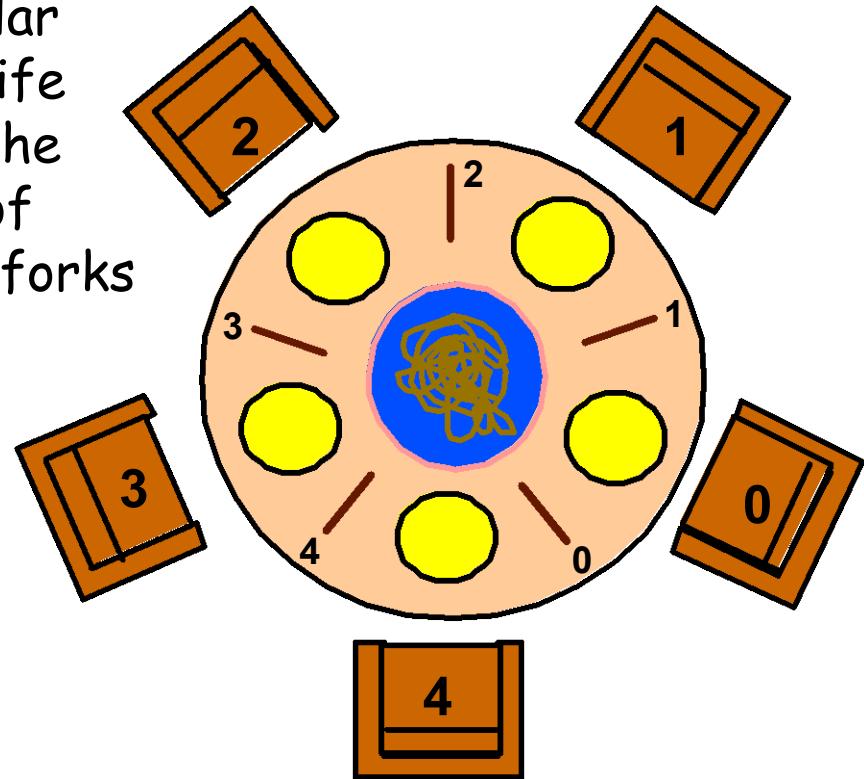
```
process consumer
integer i;
while (true) {
    down(elements);
    i=buffer[head];
    head=(head+1)% N;
    consume(i);
    up(spaces);
}
end consumer;
```

The producer now executes `down(spaces)` which suspends if the buffer is full. At the end of each loop iteration the consumer executes `up(spaces)`.

The Dining Philosophers

Five philosophers sit around a circular table. Each philosopher spends his life alternately **thinking** and **eating**. In the centre of the table is a large bowl of spaghetti. A philosopher needs two forks to eat a helping of spaghetti.

One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.



The Dining Philosophers

The synchronization problem is to coordinate the use of the forks so that:

- ◆ only one philosopher at a time uses a fork;
- ◆ there is no deadlock (each philosopher holding a fork and refusing to relinquish it);
- ◆ no philosopher starves (as a result of neighbours collaborating);

Because forks are a shared resource we will focus on acquiring and releasing them. Each fork is like critical section: it can be held by at most one philosopher at a time.

The Dining Philosophers

```
Semaphore[] fork = {1,1,1,1,1};
```

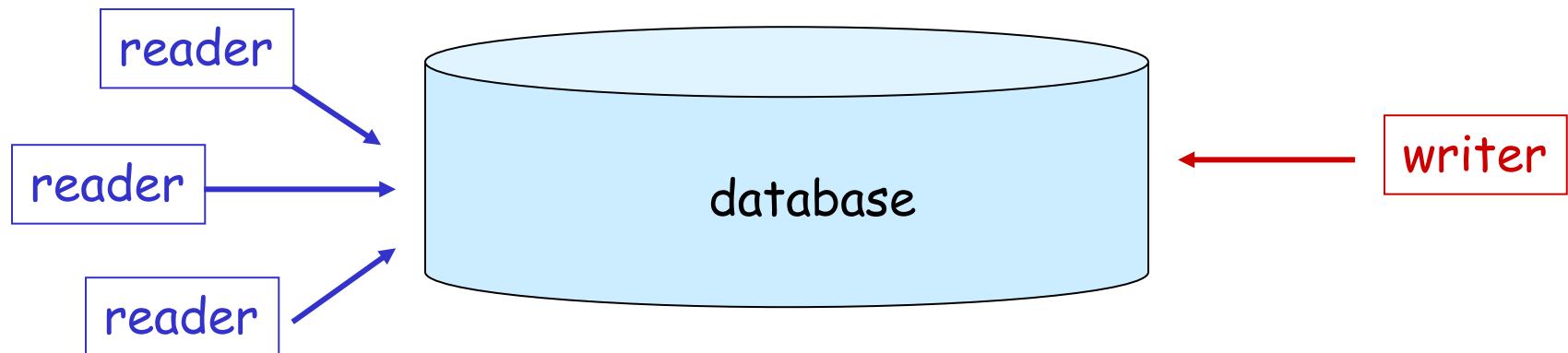
```
process philosopher[i=0..3]
while (true) {
    down(fork[i]);
    down(fork[i+1]);
    eat;
    up(fork[i]);
    up(fork[i+1]);
    think;
}
end philosopher;
```

```
process philosopher[4]
while (true) {
    down(fork[0]);
    down(fork[4]);
    eat;
    up(fork[0]);
    up(fork[4]);
    think;
}
end philosopher;
```

Why does philosopher[4] have to be different from the rest ?

The Readers and Writers Problem

In the readers and writers problem, readers may access a database simultaneously as long as no writer writes, but only one writer at a time may write, provided there are no active readers.



The Readers and Writers Problem

```
Semaphore rw = 1;
```

```
process Reader[i=1..M]
while (true) {
    down(rw);
    read_the_database;
    up(rw);
}
end Reader;
```

```
process Writer[i=1..N]
while (true) {
    down(rw);
    write_the_database;
    up(rw);
}
end Writer;
```

An over constrained solution.

The Readers and Writers Problem

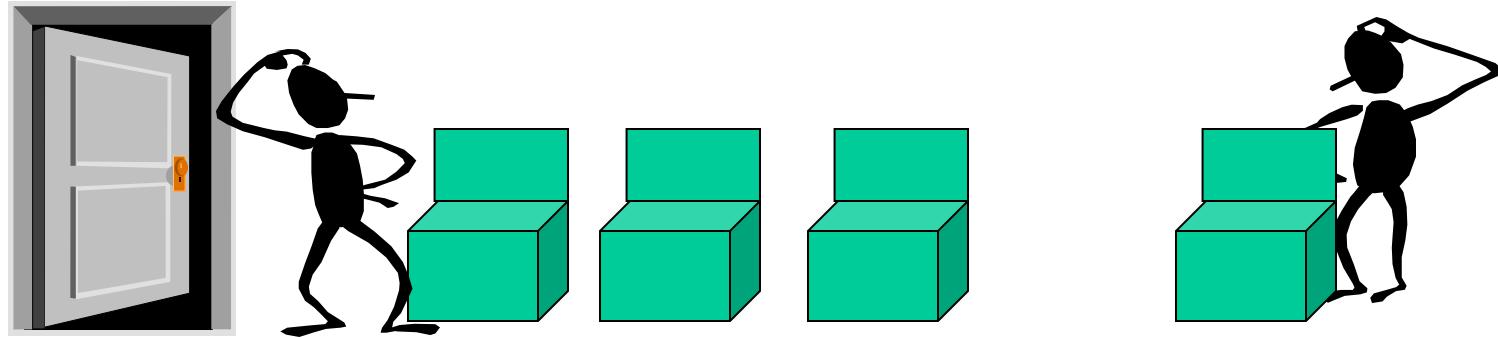
```
process Reader[i=1..M]
while (true) {
    down (mutexr);
    nr=nr+1;
    if (nr==1) down (rw);
    up (mutexr);
    read_the_database;
    down (mutexr)
    nr=nr-1;
    if (nr==0) up (rw);
    up (mutexr);
}
end Reader;
```

```
int nr =0;
Semaphore rw =1;
Semaphore mutexr = 1;
```

```
process Writer[i=1..N]
while (true) {
    down (rw);
    write_the_database;
    up (rw);
}
end Writer;
```

Under what circumstances
is this solution unfair ?

The Sleeping Barber



Customers who need a haircut enter the waiting room. If the room is full, the customer comes back later. If the barber is busy but there is a vacant chair the customer takes a seat. If the waiting room is empty and the barber is daydreaming the customer sits in the barber chair and wakes the barber.

When the barber finishes cutting a customer's hair, the barber fetches another customer from the waiting room. If the waiting room is empty the barber daydreams.

The Sleeping Barber

```
Semaphore mutex=1;  
  
Semaphore customers=0, cutting=0, barber=0;  
  
int waiting=0, numChairs= ...;
```

```
process Customer[i=1..M]  
while (true) {  
    down(mutex);  
    if (waiting<numChairs) {  
        waiting++;  
        up(customers);  
        up(mutex);  
        down(barber);  
        up(cutting);  
    }  
    else up(mutex);  
}  
end Customer;
```

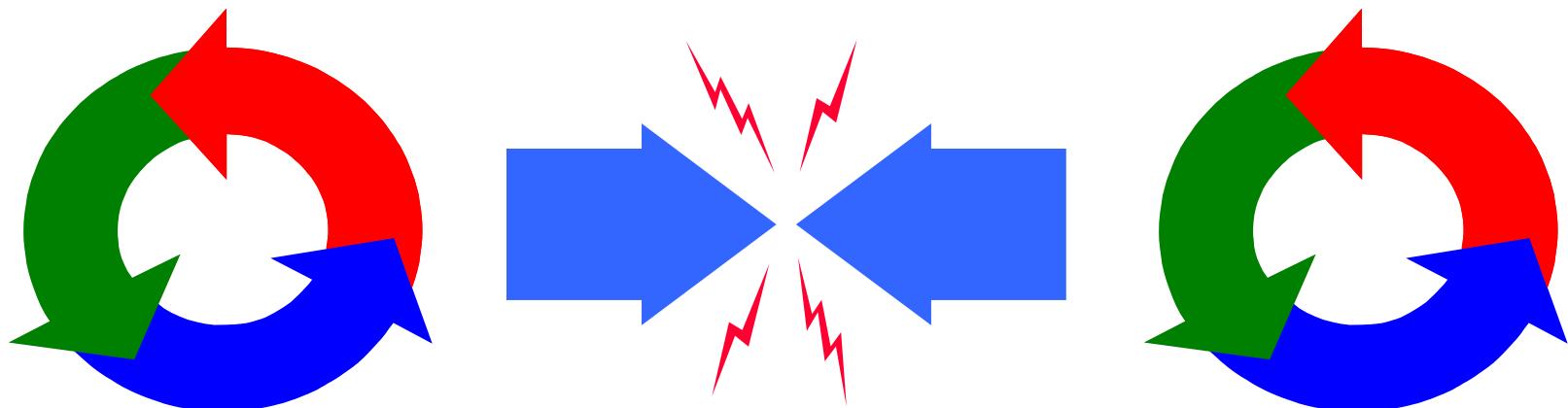
```
process SleepingBarber  
while (true) {  
    down(customers);  
    down(mutex);  
    waiting--;  
    up(barber);  
    up(mutex);  
    down(cutting);  
}  
end SleepingBarber;
```

The Sleeping Barber

This example illustrates the use of semaphores for both mutual exclusion and condition synchronization. The customers and barber use semaphores to control each other's flow through the barber shop. This interaction is an example of a client-server relationship. The customer and barber rendezvous to interact: each waits at a certain point for the other to arrive.

Slides are based mostly
on Professor N.S. Scott's (Queens University, Belfast) slides

The Mutual Exclusion Problem



The Mutual Exclusion Problem

Eliminating undesirable interleavings is called the **mutual exclusion problem**.

We need to identify **critical sections** that only one thread at a time can enter.

We need to devise a **pre-protocol** and a **post-protocol** to keep two or more threads from being in their critical sections at the same time.

```
while (true) {  
    nonCriticalSection;  
    preProtocol;  
    criticalSection;  
    postProtocol;  
}
```

The Mutual Exclusion Problem for N processes

- ◆ N processes are executing, in an infinite loop, a sequence of instructions, which can be divided into two subsequences: the **critical section** and the **non-critical section**. The program must satisfy the **mutual exclusion property**: instructions from the critical sections of two or more processes must not be interleaved.
- ◆ The solution is described by inserting into the loop additional instructions that are to be executed by a process wishing to enter and leave its critical section - the **pre-protocol** and **post-protocol**, respectively. These protocols may require additional variables.
- ◆ A process may halt in its non-critical section. It may not halt during execution of its protocols or critical section. If one process halts in its non-critical section, it must not interfere with the operation of other processes.

The Mutual Exclusion Problem for N processes

- ◆ The program must not **deadlock**. If some processes are trying to enter their critical sections then one of them must **eventually** succeed. The program is deadlocked if no process ever succeeds in making the transition from pre-protocol to critical section.
- ◆ There must be no **starvation** of any of the processes. If a process indicates its intention to enter its critical section by commencing execution of the pre-protocol, then eventually it must succeed.
- ◆ In the **absence of contention** for the critical section a single process wishing to enter its critical section will succeed. A good solution will have minimal overhead in this case.

The Mutual Exclusion Problem for 2 processes

We will solve the mutual exclusion problem for **two** processes using Load and Store to common memory as the only atomic instructions.

```
c1=1;  
if (c1==1)      Atomic  
c2=c1;          instructions
```

One solution to the mutual exclusion problem for two processes is called **Dekker's algorithm**. We will develop this algorithm in step-by-step sequence of incorrect algorithms: each will demonstrate some pathological behaviour that is typical of concurrent algorithms.

First Attempt

```
int turn=1;
```

```
process P1
while (true) {
    nonCriticalSection1;
    while (turn == 2) {}
    criticalSection1;
    turn = 2;
}
end P1;
```

```
process P2
while (true) {
    nonCriticalSection2;
    while (turn == 1) {}
    criticalSection2;
    turn = 1;
}
end P2;
```

A single shared variable **turn** indicates whose turn it is to enter the critical section.

First Attempt

Mutual exclusion No deadlock No starvation **No starvation in absence of contention**



Mutual exclusion is satisfied

Proof: Suppose that at some point both processes are in their critical sections. Without loss of generality, assume that P1 entered at time t_1 , and that P2 entered at time t_2 , where $t_1 < t_2$. P1 remained in its critical section during the interval from t_1 to t_2 .

At time t_1 , $\text{turn}==1$, and at time t_2 $\text{turn}==2$. But during the interval t_1 to t_2 P1 remained in its critical section and did not execute its post-protocol which is the only means of assigning 2 to turn . At t_2 turn must still be 1, contradicting the previous statement.

First Attempt

The solution cannot deadlock

Proof: For the program to deadlock each process must execute the test on `turn` infinitely often failing each time. Therefore, in P1 `turn==2` and in P2 `turn==1`, which is impossible.

There is no starvation

Proof: For starvation to exist one process must enter its critical section infinitely often, while the other executes its pre-protocol forever without progressing to its critical section.

But if P1 executes its even once, it will set `turn==2` in its post-protocol allowing P2 to enter its critical section.

First Attempt

There is starvation in the absence of contention

Proof: Suppose that P2 halts in its non-critical section: **turn** will never be changed from **2** to **1**. P1 may enter its critical section at most one more time. Once P1 sets **turn** to **2**, it will never again be able to progress from its pre-protocol to its critical section.

Even if both processes are guaranteed not to halt this solution must be rejected. **Why ?**

Second Attempt

```
int c1=1;  
int c2=1;
```

```
process P1  
while (true) {  
    nonCriticalSection1;  
    while (c2!=1) {}  
    c1=0;  
    criticalSection1;  
    c1=1;  
}  
  
end P1;
```

```
process P2  
while (true) {  
    nonCriticalSection2;  
    while (c1!=1) {}  
    c2=0;  
    criticalSection2;  
    c2=1;  
}  
  
end P2;
```

Each process P_i now has its own variable c_i . Shared variable $C_i=0$ signals that P_i is about to enter its critical section.

Second Attempt

Mutual exclusion

✗

Mutual exclusion is not satisfied

Proof: Consider the following interleaving beginning with the initial state.

1. P1 checks c_2 and finds $c_2==1$.
2. P2 checks c_1 and finds $c_1==1$.
3. P1 sets c_1 to 0.
4. P2 sets c_2 to 0.
5. P1 enters its critical section.
6. P2 enters its critical section.

Third Attempt

```
int c1=1;  
int c2=1;
```

```
process P1  
while (true) {  
    nonCriticalSection1;  
    c1=0;  
    while (c2!=1) {}  
    criticalSection1;  
    c1=1;  
}  
end P1;
```

```
process P2  
while (true) {  
    nonCriticalSection2;  
    c2=0;  
    while (c1!=1) {}  
    criticalSection2;  
    c2=1;  
}  
end P2;
```

In Attempt 2 the assignment **Ci=0** was effectively located in the critical section. Try moving it to the beginning of the pre-protocol, **Ci==0** now signals that Pi wishes to enter its critical section.

Third Attempt

Mutual exclusion **No deadlock**



Mutual exclusion is satisfied

Proof: Suppose that at some point both processes are in their critical sections. Without loss of generality, assume that P1 entered at time t_1 , and that P2 entered at time t_2 , where $t_1 < t_2$. P1 remained in its critical section during the interval from t_1 to t_2 .

At time t_1 , $c_1 == 0$ and $c_2 == 1$ and at time t_2 $c_2 == 0$ and $c_1 == 1$. But during the interval t_1 to t_2 P1 remained in its critical section and did not execute its post-protocol which is the only means of assigning **1** to c_1 . At t_2 c_1 must still be **0**, contradicting the previous statement.

Third Attempt

The program can deadlock

Proof: Consider the following interleaving beginning with the initial state.

1. P1 sets c_1 to 0.
2. P2 sets c_2 to 0.
3. P1 tests c_2 and remains in the loop.
4. P2 tests c_1 and remains in the loop.

Both processes are locked forever in their pre-protocols.

A Correct Solution: Dekker's Algorithm

- ◆ It avoids the strict alternation of a naïve turn-taking algorithm, and was one of the first mutual exclusion algorithms to be invented.
- ◆ If both processes attempt to access the critical section at the same time, the algorithm will choose the process whose turn it is. If the other process is already modifying the critical section, it will wait for the process to finish

Dekker's Algorithm – Prove that this algorithm provides Mutual Exclusion and is free from Starvation & deadlock and progresses in the absence of contention

```
int c1=1;  
int c2=1;  
int turn=1;
```

```
process P1  
while (true) {  
    nonCriticalSection1;  
    c1=0;  
    while (c2!=1)  
        if (turn==2) {  
            c1=1;  
            while (turn!=1) {}  
            c1=0;  
        }  
    criticalSection1;  
    c1=1; turn=2;  
}  
end P1;
```

```
process P2  
while (true) {  
    nonCriticalSection2;  
    c2=0;  
    while (c1!=1)  
        if (turn==1) {  
            c2=1;  
            while (turn!=2) {}  
            c2=0;  
        }  
    criticalSection2;  
    c2=1; turn=1;  
}  
end P2;
```

The threads now take turns at backing off.

Dekker's Algorithm

Mutual exclusion No deadlock No starvation No starvation in absence
of contention



Exercise

Consider the following proposed solution to the mutual exclusion problem.

```
int p=1, q=1;
```

```
process P {
    while (true) {
a1:      non-critical-1;
b1:      while (q!=1) {}
c1:      p=0;
d1:      critical-1;
e1:      p=1;
    }
}
```

```
process Q {
    while (true) {
a2:      non-critical-2;
b2:      while (p!=1) {}
c2:      q=0;
d2:      critical-2;
e2:      q=1;
    }
}
```

Exercise

For each of (i) and (ii) below justify your answer

- (i) Does the algorithm operate correctly in the absence of contention?
- (ii) Can the algorithm deadlock?

Exercise

Solution

- (i) The program does operate correctly in the absence of contention. Assume P has stopped in its non-critical section. Then the value of p is 1 either by initialisation or following execution of $e1$ (and noting that Q does not make any assignment to p). With p having the value 1, Q is free to enter its critical section as it wishes.
- (ii) The algorithm cannot deadlock. For, suppose it is deadlocked, then P is at $b1$ and Q is at $b2$. But, if P is at $b1$ then p has the value 1 (either by initialisation or following execution of $e1$ and noting that Q does not assign p). Since $p = 1$, Q cannot be stuck at $b2$.

Peterson's algorithm

- ◆ Both the turn variable and the status flags are used, as in Dekker's algorithm.
- ◆ After setting our flag we immediately give away the turn. We then wait for the turn if and only if the other flag is set.
- ◆ By waiting on the **and** of two conditions, we avoid the need to clear and reset the flags.
- ◆ Possible for n number of processes

Peterson's Algorithm

```
process P1
while (true) {
    nonCriticalSection1;
    c1=0;
    turn=2
        while (turn==2 &&
c2==0) {};

    criticalSection1;
    c1=1;
}
end P1;
```

```
int c1=1;
int c2=1;
int turn=1;
```

```
process P2
while (true) {
    nonCriticalSection2;
    c2=0;
    turn=1
        while (turn==1 &&
c1==0) {};

    criticalSection2;
    c2=1;
}
end P2;
```

```
do { flag [i] = TRUE;
      turn = j;
      while ( flag [j] && turn == j); //Note: empty
loop
          CRITICAL SECTION
      flag [i] = FALSE;
          REMAINDER SECTION
} while (TRUE);
```

Hardware Assisted Mutual Exclusion

The difficulty in implementing a solution to the **mutual exclusion property** is caused by possible interleaving of load and store instructions. This difficulty disappears if a load and store are allowed in a single atomic instruction. This is provided on some computers as a **test-and-set** machine language instruction.

`testAndSet(Li)` is equivalent to the atomic execution of

```
Li=c;  
c=1;
```

where `c` is a global variable with initial value 0; and `Li` is a variable local to process `Pi`.

Hardware Assisted Mutual Exclusion

```
int c=0;  
  
process Pi  
    private int Li = 0;  
    while (true) {  
        nonCriticalSection;  
        do {  
            testAndSet(Li);  
        } while (Li != 0);  
        criticalSection;  
        c = 0;  
    }  
end Pi;
```

Li=c;
c=1;

Test_and_Set solution to
the mutual exclusion
property for N processes

Hardware Assisted Mutual Exclusion

```
int c=0;
```

```
process Pi
    private int Li = 1;
    while (true) {
        nonCriticalSection();
        do {
            exchange(c,Li);
        } while (Li != 0);
        criticalSection();
        exchange(c,Li);
    }
end Pi;
```

```
temp=c;
c=Li;
Li=temp;
```

Exchange solution to the
mutual exclusion property
for N processes

Exercise

Consider the following program that uses the *Exchange* (hardware) instruction described in the notes.

```
int C=0;
```

```
process P {
    int p=1;
    while (true) {
a1:        non-critical-1;
        while (true) {
b1:            Exchange(C,p);
c1:            if (p==0) {break;}
        }
d1:        critical-1;
e1:        Exchange(C,p);
    }
}
```

```
process Q {
    int q=1;
    while (true) {
a2:        non-critical-2;
        while (true) {
b2:            Exchange(C,q);
c2:            if (q==0) {break;}
        }
d2:        critical-2;
e2:        Exchange(C,q);
    }
}
```

Exercise

- (i) Show that the above solution can lead to starvation.
- (ii) Show that the program satisfies the mutual exclusion property. [Hint: consider the number of variables having the value 1 at any given time and then consider the implication of both processes being in their critical section at the same time.]

Exercise

Solution

(i) After the statements:

a1; a2; b1; b2;

the value of p is 0 and the values of q and C are both 1.
We can then execute the sequence of statements:

c1; d1; e1; a1; b1; c2; b2;

infinitely often which shows that P can enter its critical section infinitely often while Q is being starved.

Exercise

Solution

(ii) There are three variables in the program that all have the value 0 or 1. Initially there are two variables (p and q) which have the value 1 and one which has the value 0 (C). The number of variables that have the value 1 is NOT changed by execution of the *Exchange* instruction (which is the ONLY mechanism by which a variable may have its value changed during program execution). Thus, there are ALWAYS two variables which have the value 1.

Suppose that P and Q are both in their critical sections. Then, from the program code, BOTH p and q must have the value 0, contradicting the observation above.

Thus the mutual exclusion property IS satisfied.