

Chapter 1: Introduction





Chapter 1: Introduction

- What Operating Systems Do
- Computer-System Organization
- Computer-System Architecture
- Operating-System Structure
- Operating-System Operations
- Process Management
- Memory Management
- Storage Management
- Protection and Security
- Distributed Systems
- Special-Purpose Systems
- Computing Environments





Objectives

- To provide a grand tour of the major operating systems components
- To provide coverage of basic computer system organization





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.





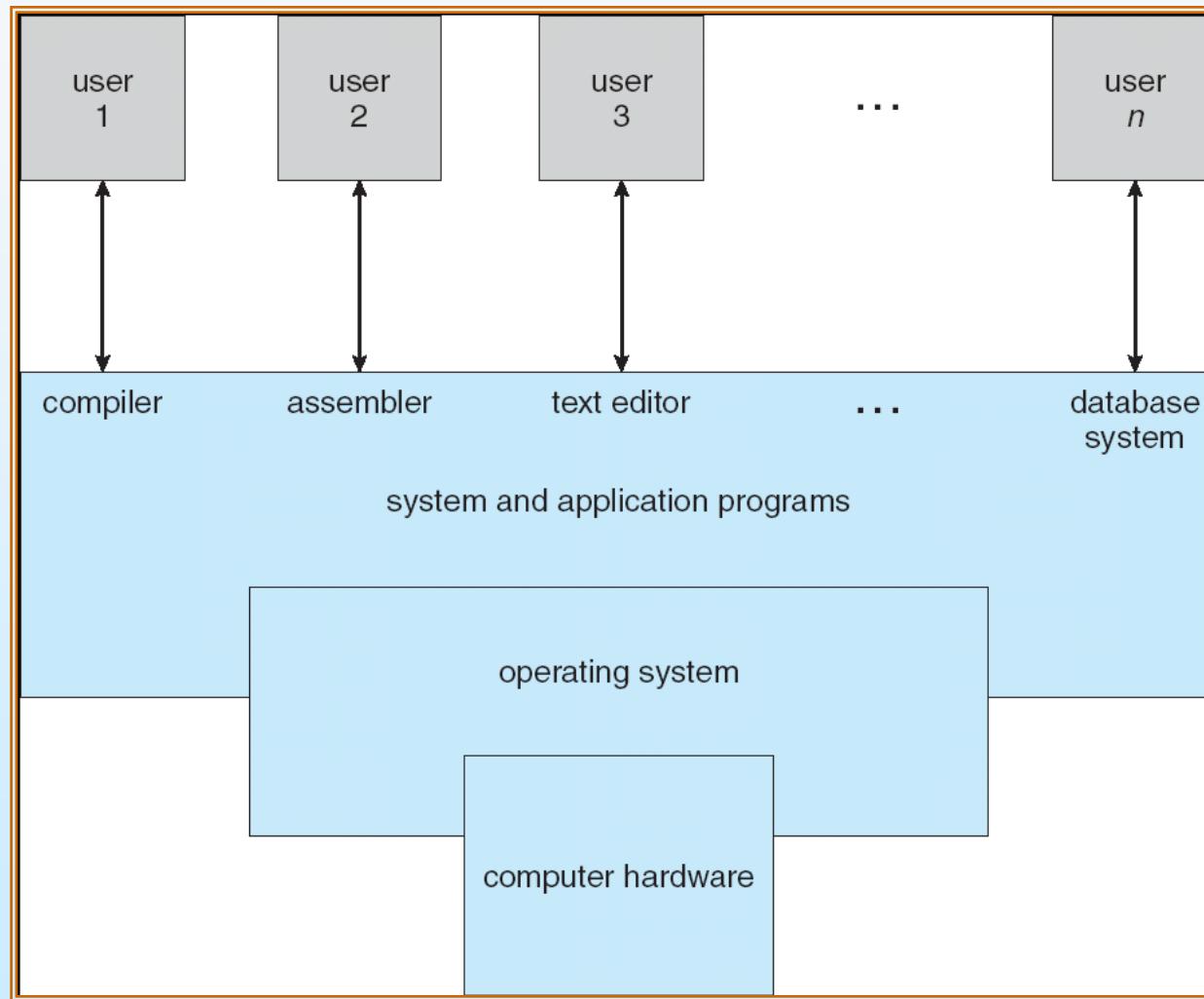
Computer System Structure

- 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





Four Components of a Computer System





Operating System Definition

- OS is a **resource allocator**
 - Manages all resources
 - Decides between conflicting requests for efficient and fair resource use
- OS is a **control program**
 - Controls execution of programs to prevent errors and improper use of the computer





Operating System Definition (Cont.)

- No universally accepted definition
- “Everything a vendor ships when you order an operating system” is good approximation
 - But varies wildly
- “The one program running at all times on the computer” is the **kernel**. Everything else is either a system program (ships with the operating system) or an application program





Computer Startup

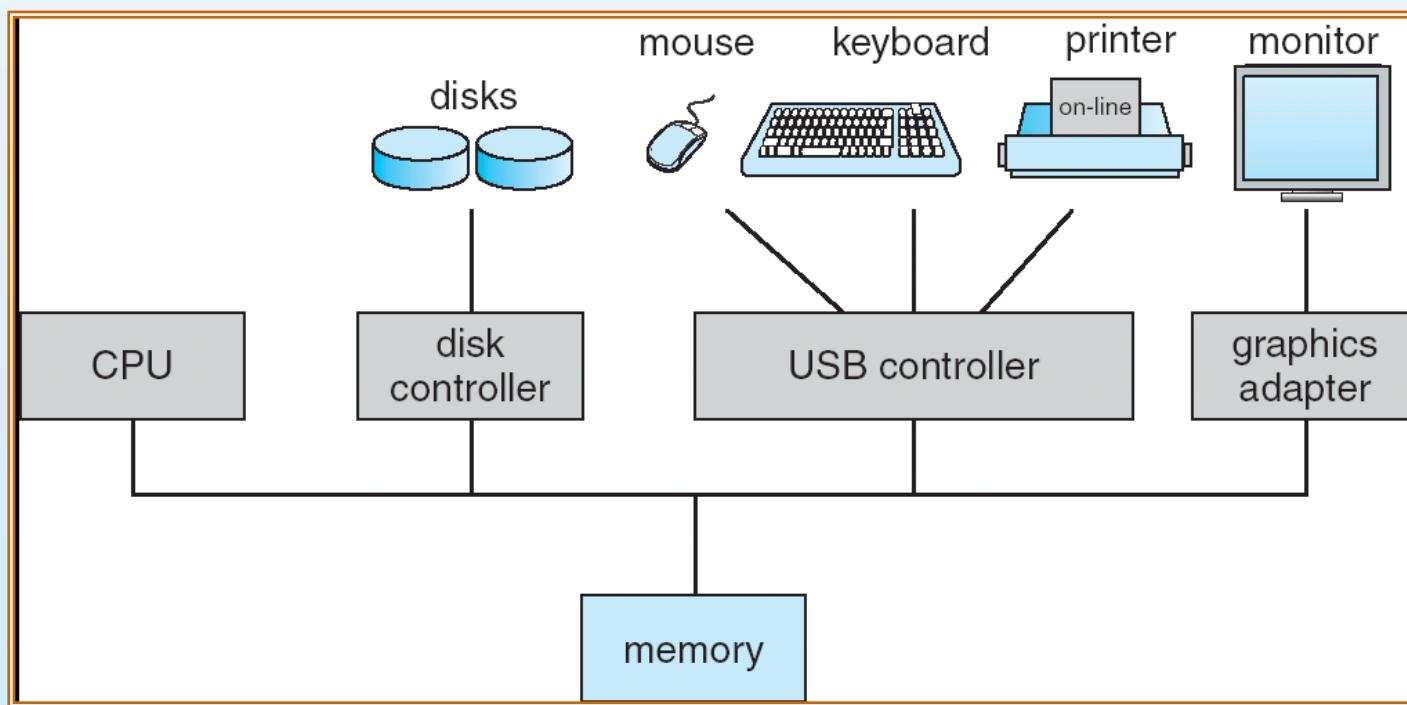
- **bootstrap program** is loaded at power-up or reboot
 - Typically stored in ROM or EEPROM, generally known as **firmware**
 - Initializes all aspects of system
 - Loads operating system kernel and starts execution





Computer System Organization

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





Computer-System Operation

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





Common Functions of Interrupts

- Interrupt transfers control to the interrupt service routine generally, through the *interrupt vector*, which contains the addresses of all the service routines.
- Interrupt architecture must save the address of the interrupted instruction.
- Incoming interrupts are *disabled* while another interrupt is being processed to prevent a *lost interrupt*.
- A *trap* is a software-generated interrupt caused either by an error or a user request.
- An operating system is *interrupt driven*.





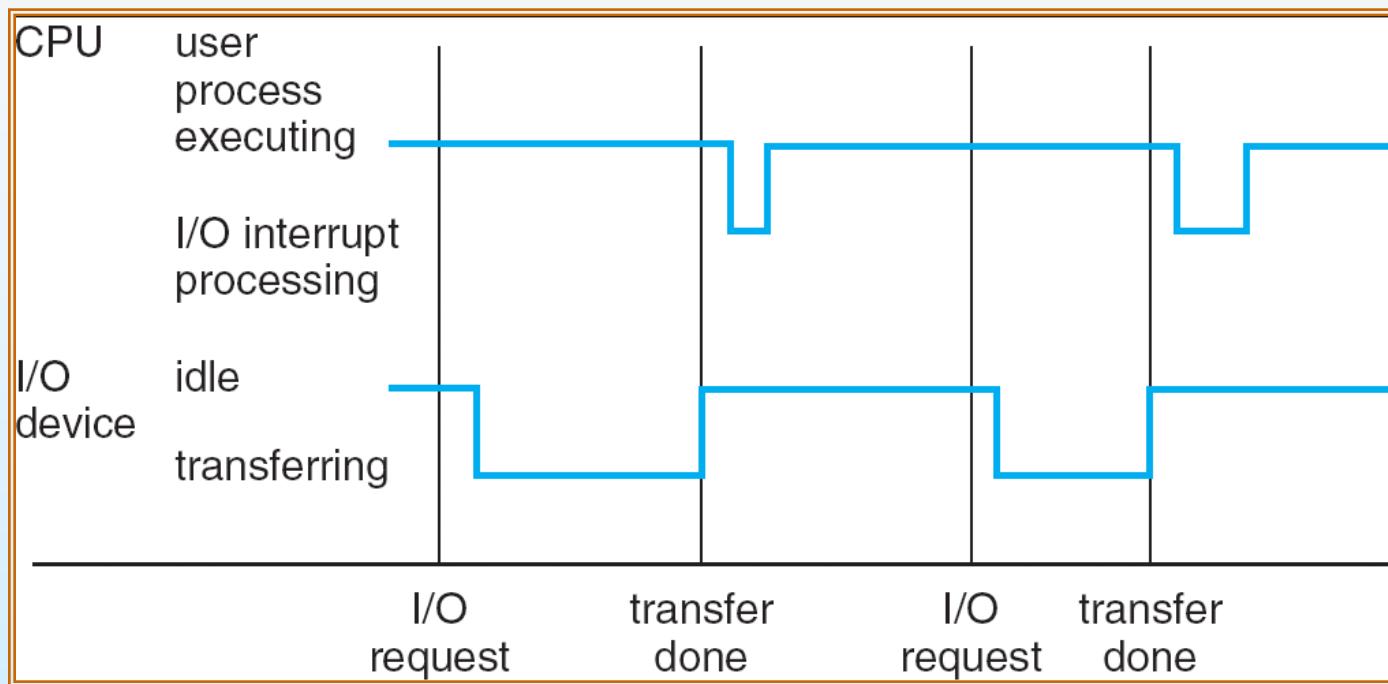
Interrupt Handling

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





Interrupt Timeline





I/O Structure

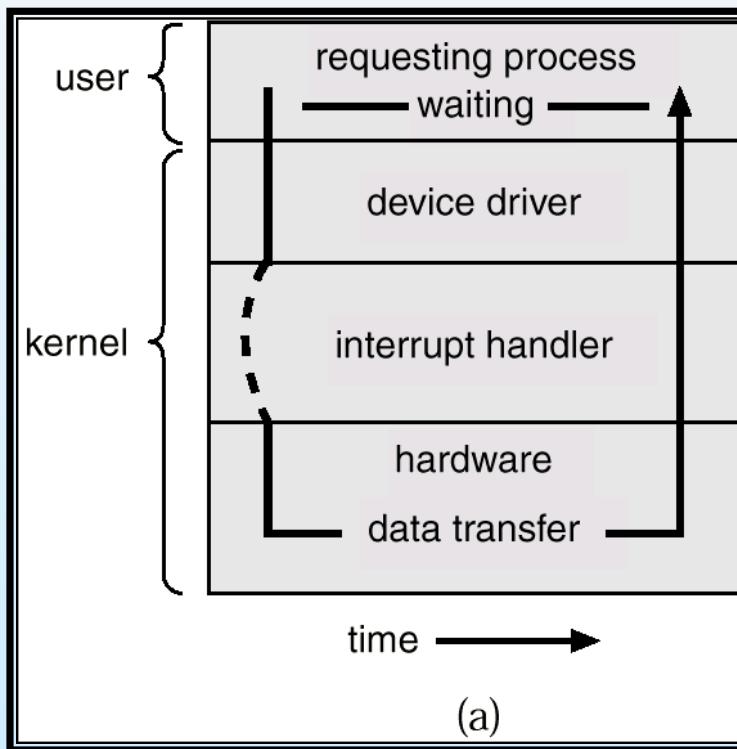
- After I/O starts, control returns to user program only upon I/O completion.
 - Wait instruction idles the CPU until the next interrupt
 - Wait loop (contention for memory access).
 - At most one I/O request is outstanding at a time, no simultaneous I/O processing.
- After I/O starts, control returns to user program without waiting for I/O completion.
 - *System call* – request to the operating system 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.
 - Operating system indexes into I/O device table to determine device status and to modify table entry to include interrupt.



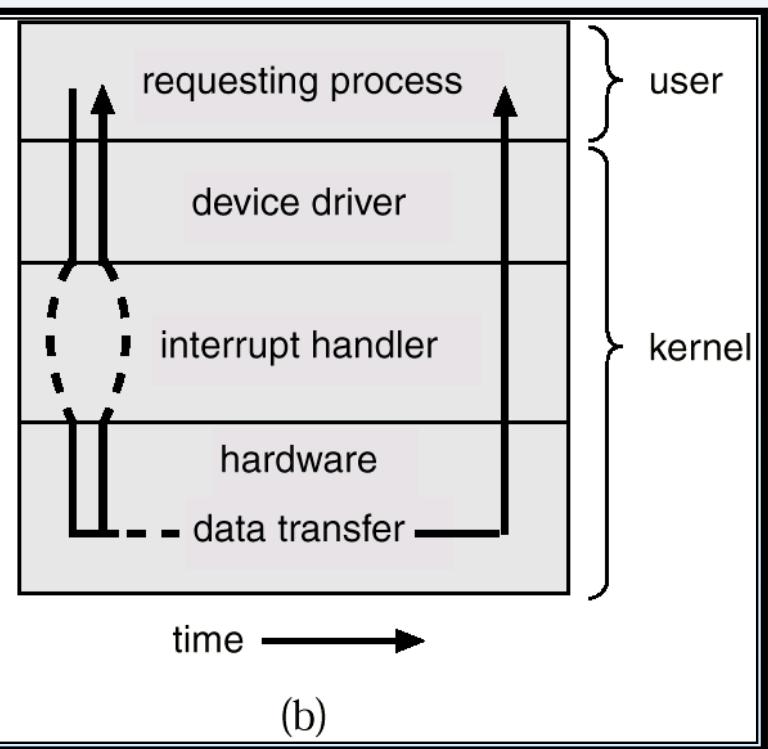


Two I/O Methods

Synchronous

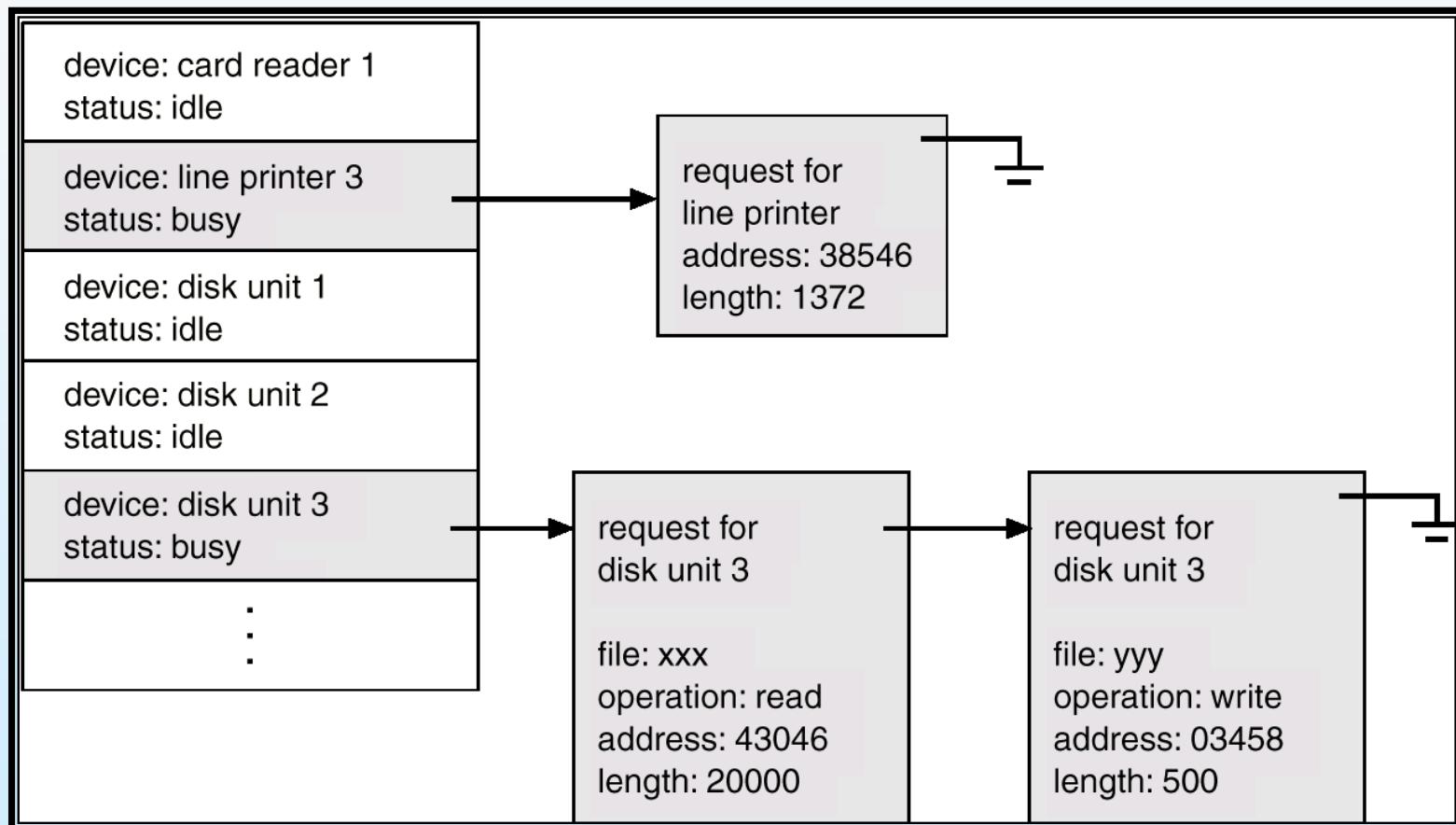


Asynchronous





Device-Status Table





Direct Memory Access Structure

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





Storage Structure

- Main memory – only large storage media that the CPU can access directly.
- Secondary storage – extension of main memory that provides large nonvolatile storage capacity.
- Magnetic disks – rigid metal or glass platters covered with magnetic recording material
 - Disk surface is logically divided into *tracks*, which are subdivided into *sectors*.
 - The *disk controller* determines the logical interaction between the device and the computer.





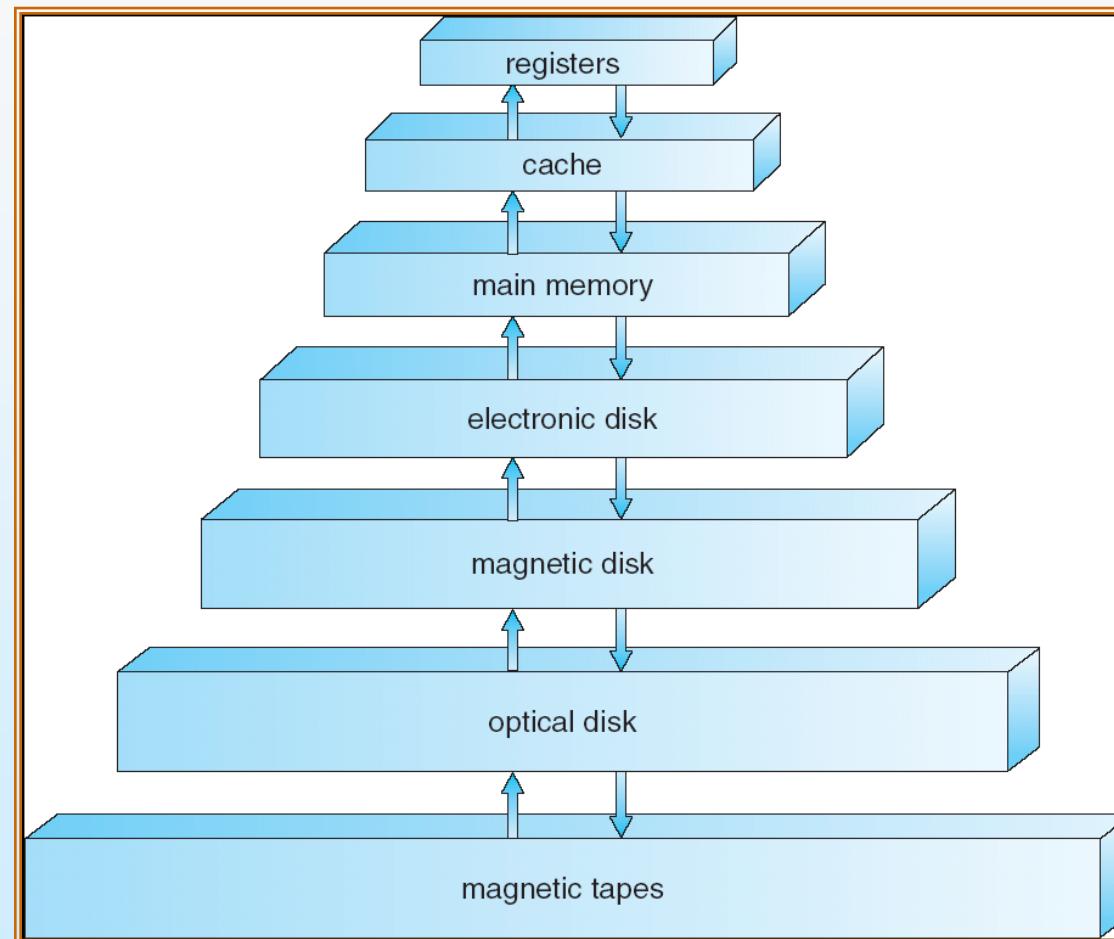
Storage Hierarchy

- Storage systems organized in hierarchy.
 - Speed
 - Cost
 - Volatility
- *Caching* – copying information into faster storage system; main memory can be viewed as a last *cache* for secondary storage.





Storage-Device Hierarchy





Caching

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





Performance of Various Levels of Storage

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

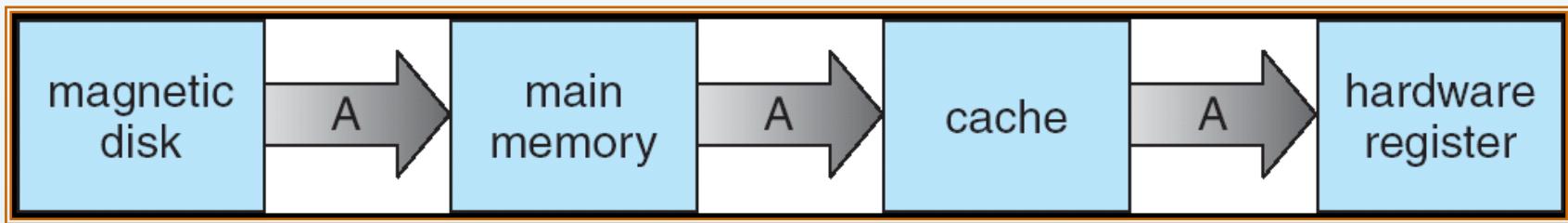
Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000,000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape





Migration of Integer A from Disk to Register

- Multitasking environments must be careful to use most recent value, not 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 17





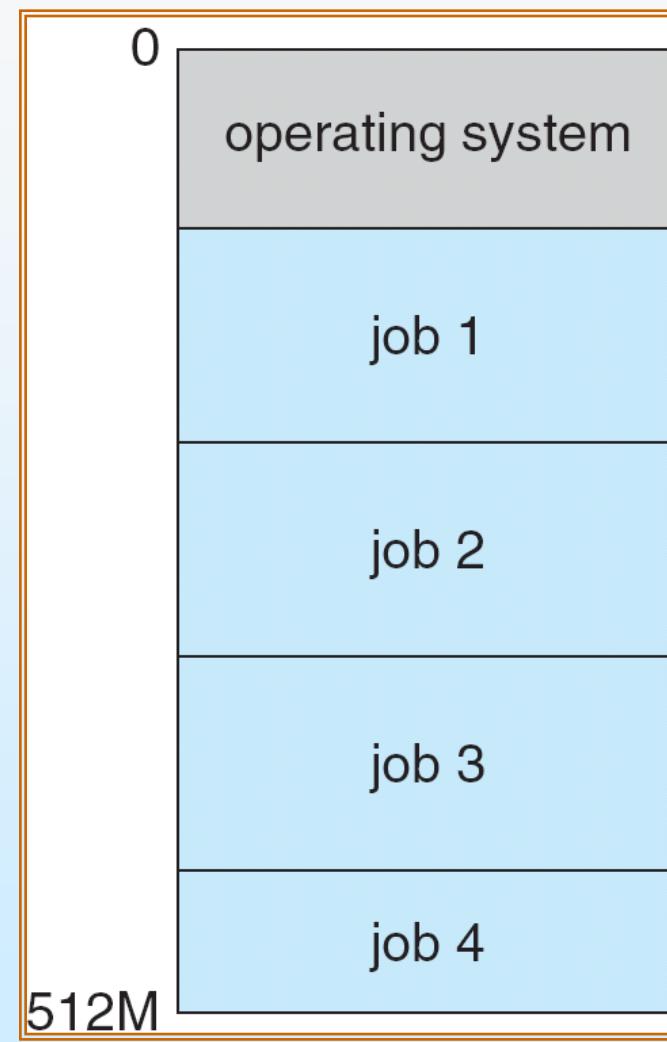
Operating System Structure

- **Multiprogramming** needed for efficiency
 - Single user cannot keep CPU and I/O devices busy at all times
 - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
 - A subset of total jobs in system is kept in memory
 - One job selected and run via **job scheduling**
 - When it has to wait (for I/O for example), OS switches to another job
- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
 - **Response time** should be < 1 second
 - Each user has at least one program executing in memory ⇒ **process**
 - If several jobs ready to run at the same time ⇒ **CPU scheduling**
 - If processes don't fit in memory, **swapping** moves them in and out to run
 - **Virtual memory** allows execution of processes not completely in memory





Memory Layout for Multiprogrammed System





Operating-System Operations

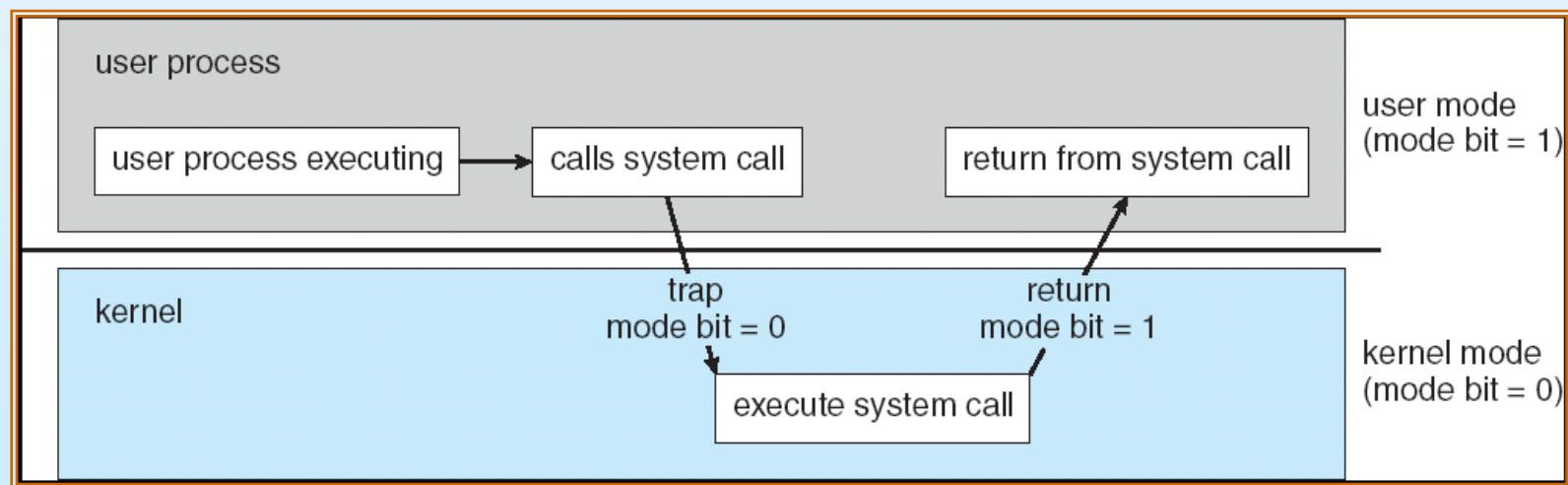
- Interrupt driven by hardware
- Software error or request creates **exception** or **trap**
 - Division by zero, request for operating system service
- Other process problems include infinite loop, processes modifying each other or the operating system
- **Dual-mode** operation allows OS to protect itself and other system components
 - **User mode** and **kernel mode**
 - **Mode bit** provided by hardware
 - ▶ Provides ability to distinguish when system is running user code or kernel code
 - ▶ Some instructions designated as **privileged**, only executable in kernel mode
 - ▶ System call changes mode to kernel, return from call resets it to user





Transition from User to Kernel Mode

- Timer to prevent infinite loop / process hogging resources
 - Set interrupt after specific period
 - Operating system decrements counter
 - 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

- All data in memory before and after processing
- All instructions in memory in order to execute
- Memory management determines what is in memory 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





Storage 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 dirs
 - ▶ 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
 - Free-space management
 - Storage allocation
 - Disk scheduling
- Some storage need not be fast
 - Tertiary storage includes optical storage, magnetic tape
 - Still must be managed
 - Varies between WORM (write-once, read-many-times) and RW (read-write)





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





Computing Environments

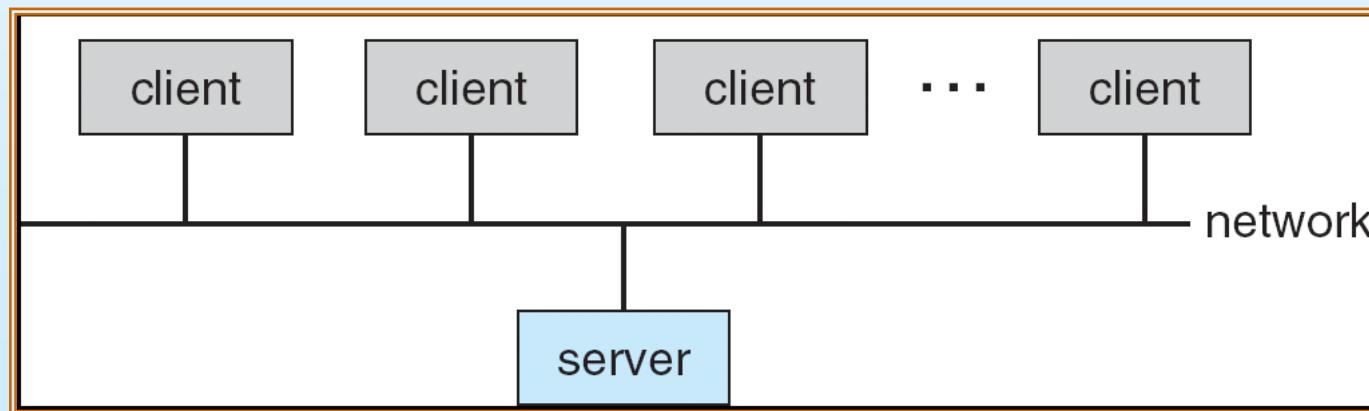
- Traditional computer
 - Blurring over time
 - Office environment
 - ▶ PCs connected to a network, terminals attached to mainframe or minicomputers providing batch and timesharing
 - ▶ Now portals allowing networked and remote systems access to same resources
 - Home networks
 - ▶ Used to be single system, then modems
 - ▶ Now firewalled, networked





Computing Environments (Cont.)

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





Peer-to-Peer Computing

- 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*



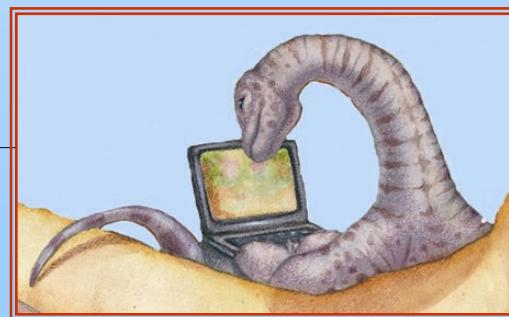


Web-Based Computing

- Web has become ubiquitous
- PCs most prevalent devices
- More devices becoming networked to allow web access
- New category of devices to manage web traffic among similar servers: **load balancers**
- Use of operating systems like Windows 95, client-side, have evolved into Linux and Windows XP, which can be clients and servers



End of Chapter 1



Chapter 2: Operating-System Structures





Chapter 2: Operating-System Structures

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





Objectives

- To describe the services an operating system provides to users, processes, and other systems
- To discuss the various ways of structuring an operating system
- To explain how operating systems are installed and customized and how they boot





Operating System Services

- 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), 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. Obviously, 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 - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code.
 - **Accounting** - To keep track of which users use how much and what kinds of computer resources
 - **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
 - ▶ **Protection** involves ensuring that all access to system resources is controlled
 - ▶ **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
 - ▶ If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.





User Operating System Interface - CLI

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





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 as “Aqua” GUI interface with UNIX kernel underneath and shells available
 - Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)





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 Program 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)
- Why use APIs rather than system calls?

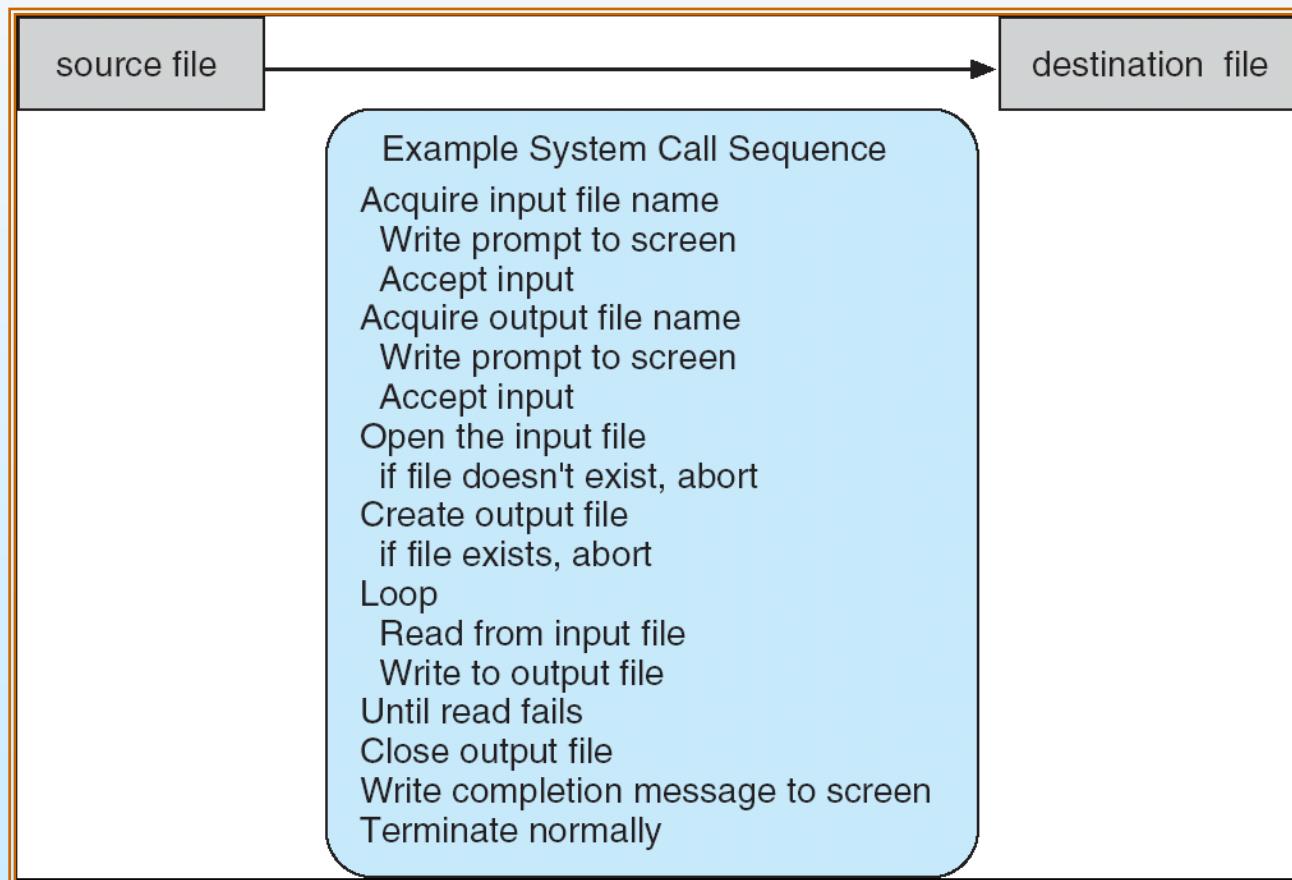
(Note that the system-call names used throughout this text are generic)





Example of System Calls

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





Example of Standard API

- Consider the ReadFile() function in the
- Win32 API—a function for reading from a file

```
return value
↓
BOOL ReadFile c (HANDLE file,
                  LPVOID buffer,
                  DWORD bytes To Read,
                  LPDWORD bytes Read,
                  LPOVERLAPPED ovl);
                  ] parameters
↑
function name
```

- A description of the parameters passed to ReadFile()
 - HANDLE file—the file to be read
 - LPVOID buffer—a buffer where the data will be read into and written from
 - DWORD bytesToRead—the number of bytes to be read into the buffer
 - LPDWORD bytesRead—the number of bytes read during the last read
 - LPOVERLAPPED ovl—indicates if overlapped I/O is being used





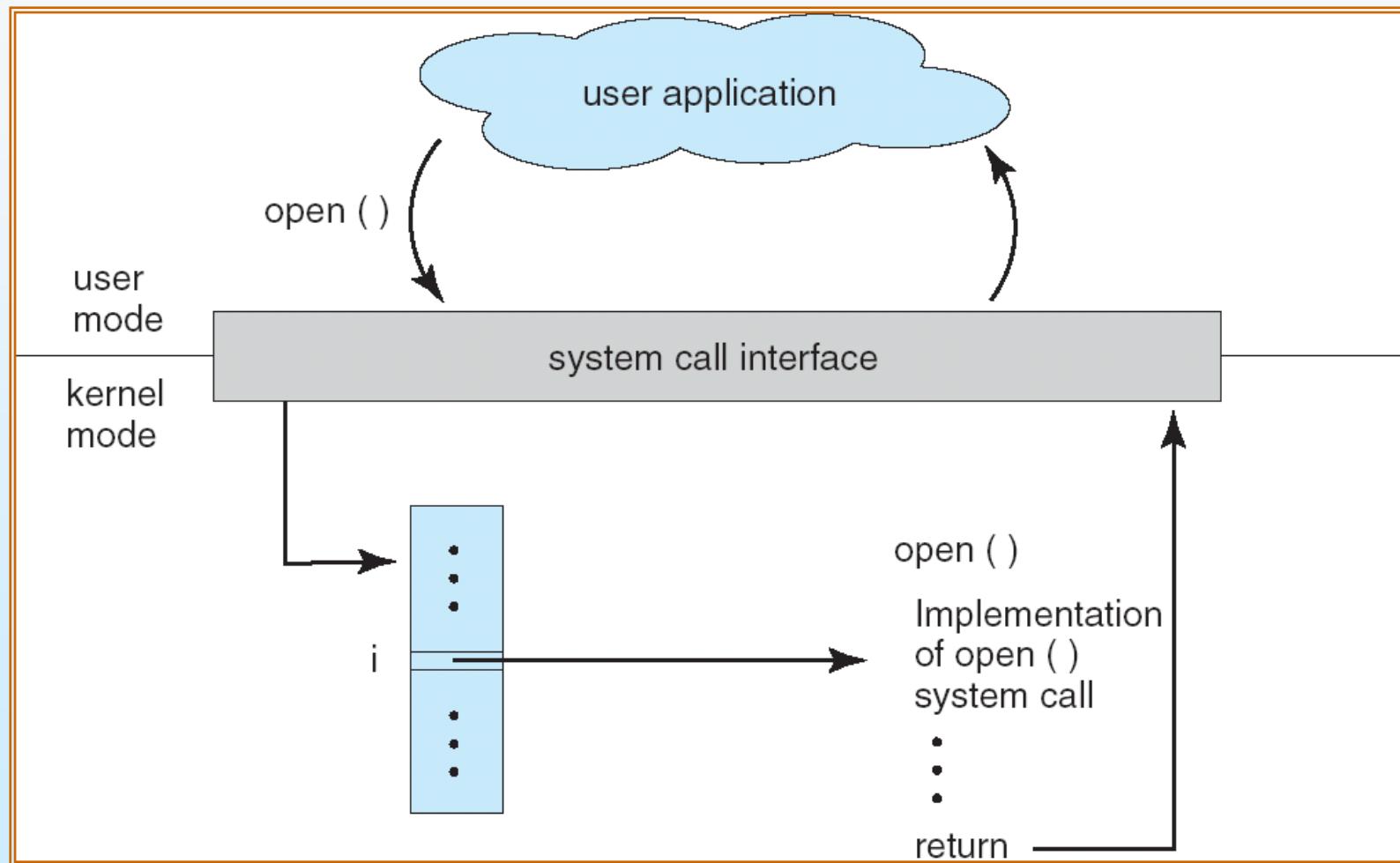
System Call Implementation

- Typically, a number associated with each system call
 - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes 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)





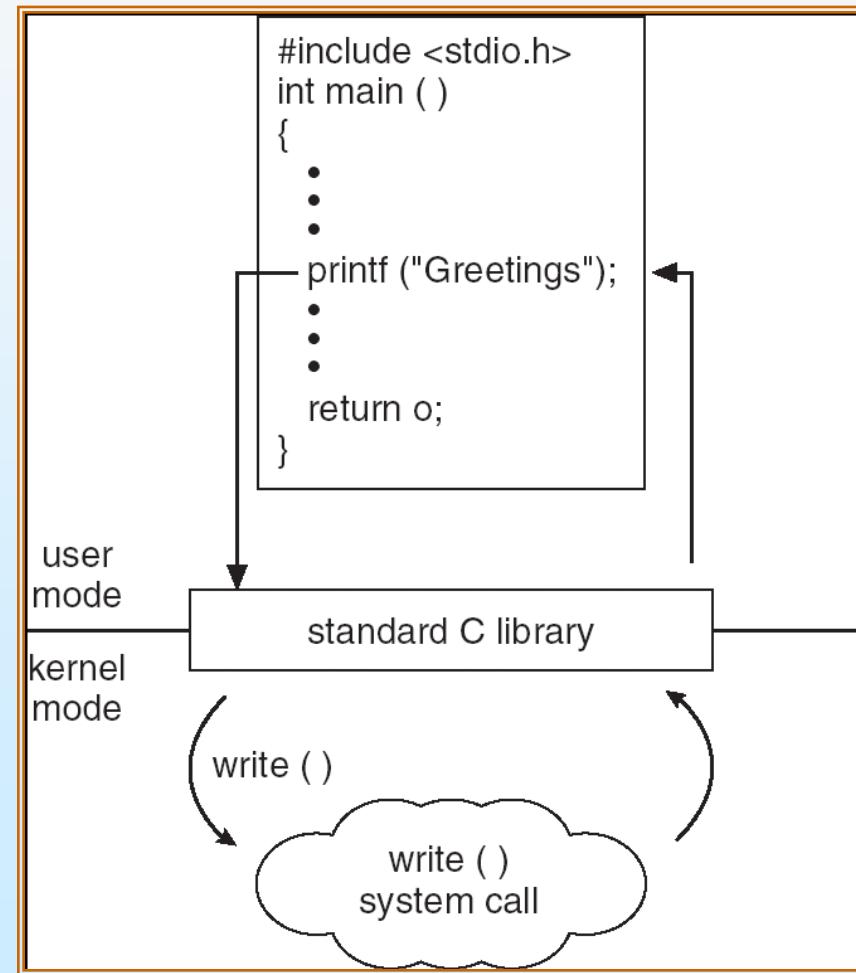
API – System Call – OS Relationship





Standard C Library Example

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





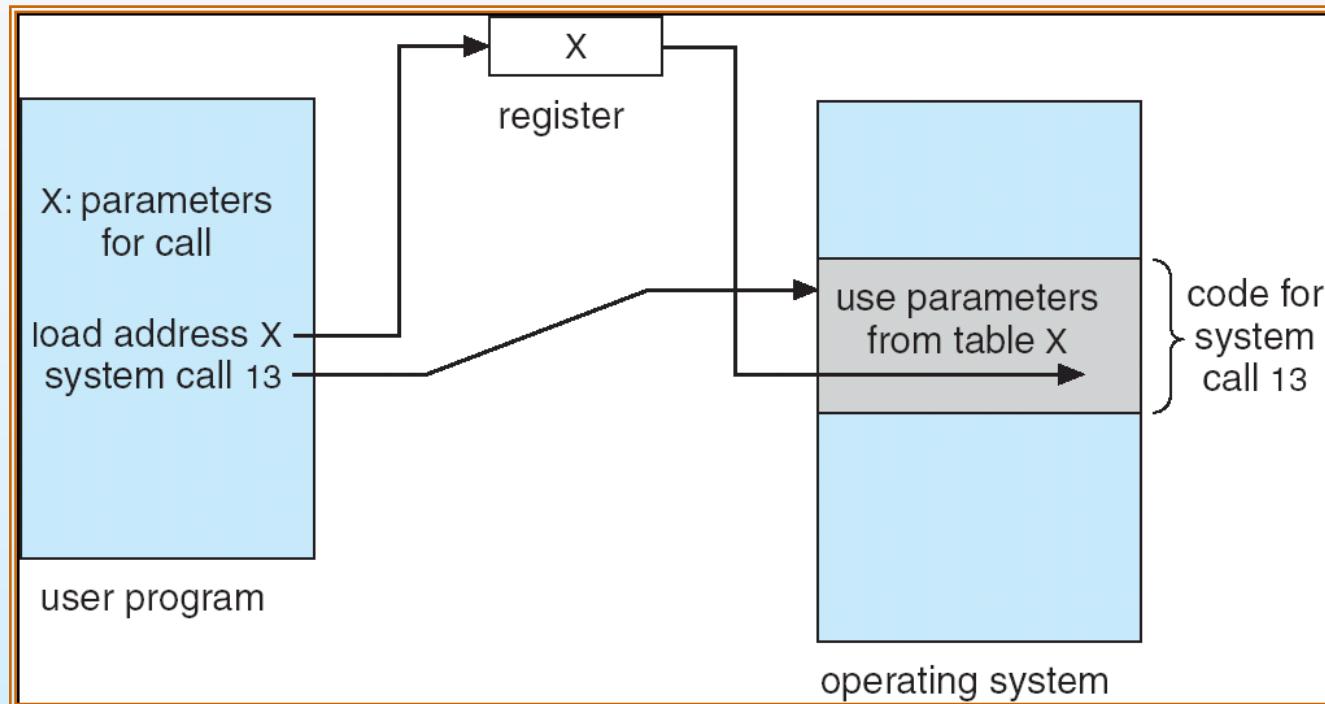
System Call Parameter Passing

- 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





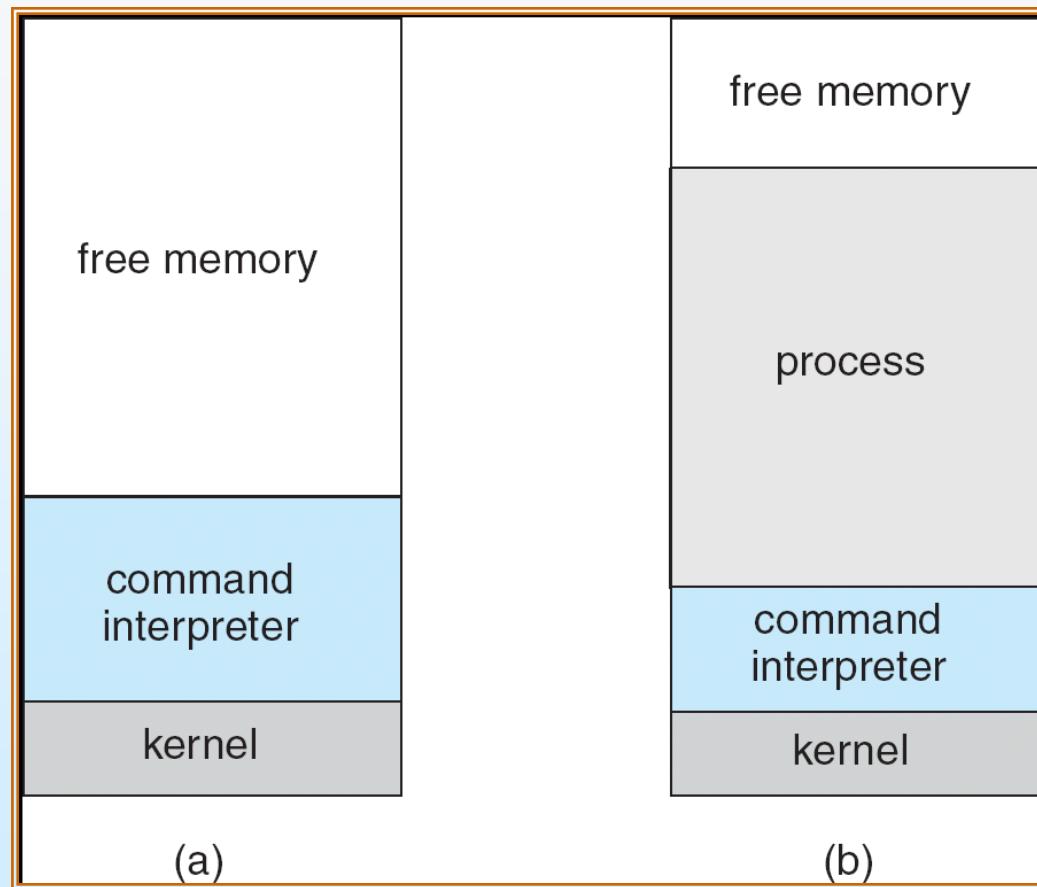
Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communications





MS-DOS execution

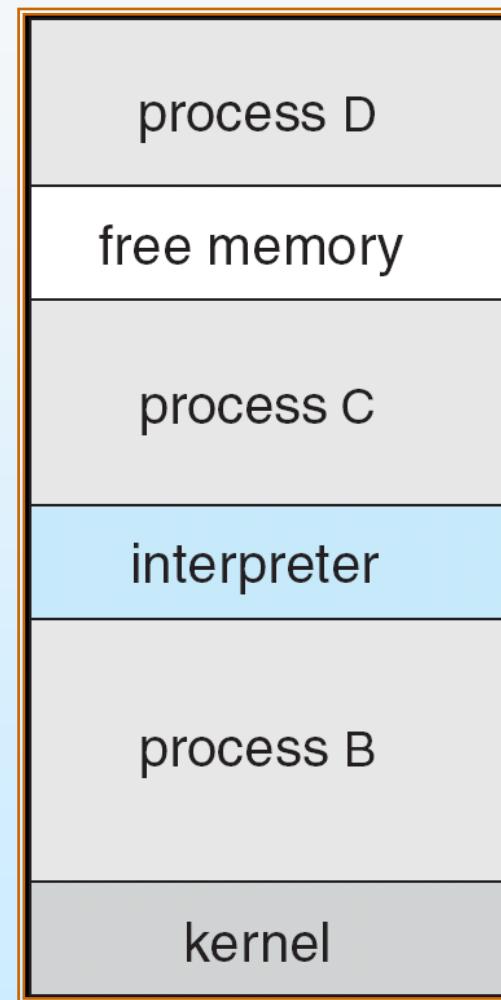


(a) At system startup (b) running a program





FreeBSD Running Multiple Programs





System Programs

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





Solaris 10 dtrace Following System Call

```
# ./all.d `pgrep xclock` XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued          U
  0 -> _XEventsQueued        U
  0 -> _X11TransBytesReadable U
  0 <- _X11TransBytesReadable U
  0 -> _X11TransSocketBytesReadable U
  0 <- _X11TransSocketBytesreadable U
  0 -> ioctl                  U
  0 -> ioctl                  K
  0 -> getf                  K
  0 -> set_active_fd          K
  0 <- set_active_fd          K
  0 <- getf                  K
  0 -> get_udatamodel         K
  0 <- get_udatamodel         K
...
  0 -> releaseef             K
  0 -> clear_active_fd        K
  0 <- clear_active_fd        K
  0 -> cv_broadcast           K
  0 <- cv_broadcast           K
  0 <- releaseef              K
  0 <- ioctl                  K
  0 <- ioctl                  U
  0 <- _XEventsQueued         U
  0 <- XEventsQueued          U
```





System Programs

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





System Programs (cont'd)

- 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





Operating System Design and Implementation

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start by defining goals and specifications
- Affected by choice of hardware, type of system
- *User* goals and *System* goals
 - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
 - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient





Operating System Design and Implementation (Cont.)

- Important principle to separate
 - Policy:** What will be done?
 - Mechanism:** How to do it?
- Mechanisms determine how to do something, policies decide what will be done
 - The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later





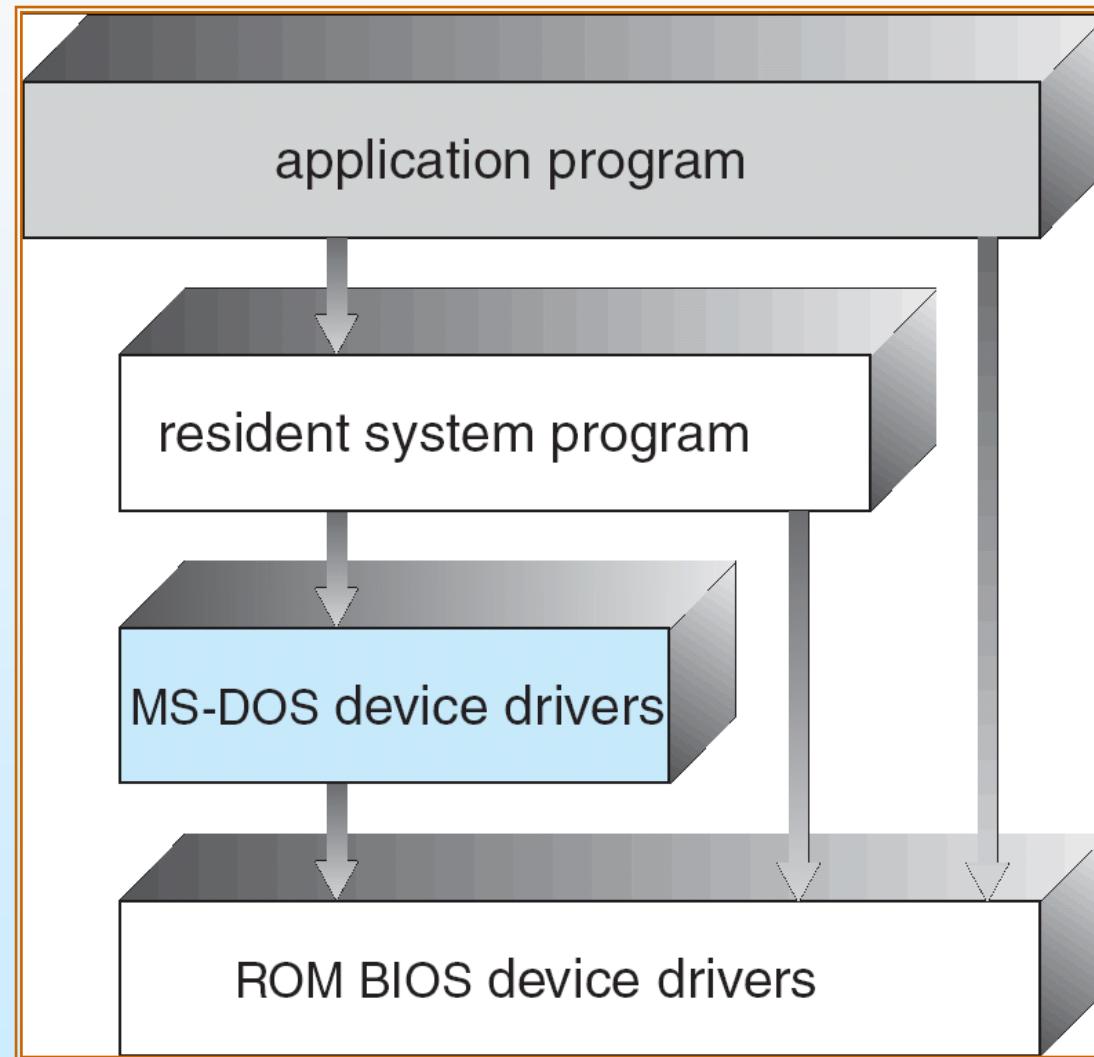
Simple Structure

- MS-DOS – written to provide the most functionality in the least space
 - Not divided into modules
 - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated





MS-DOS Layer Structure





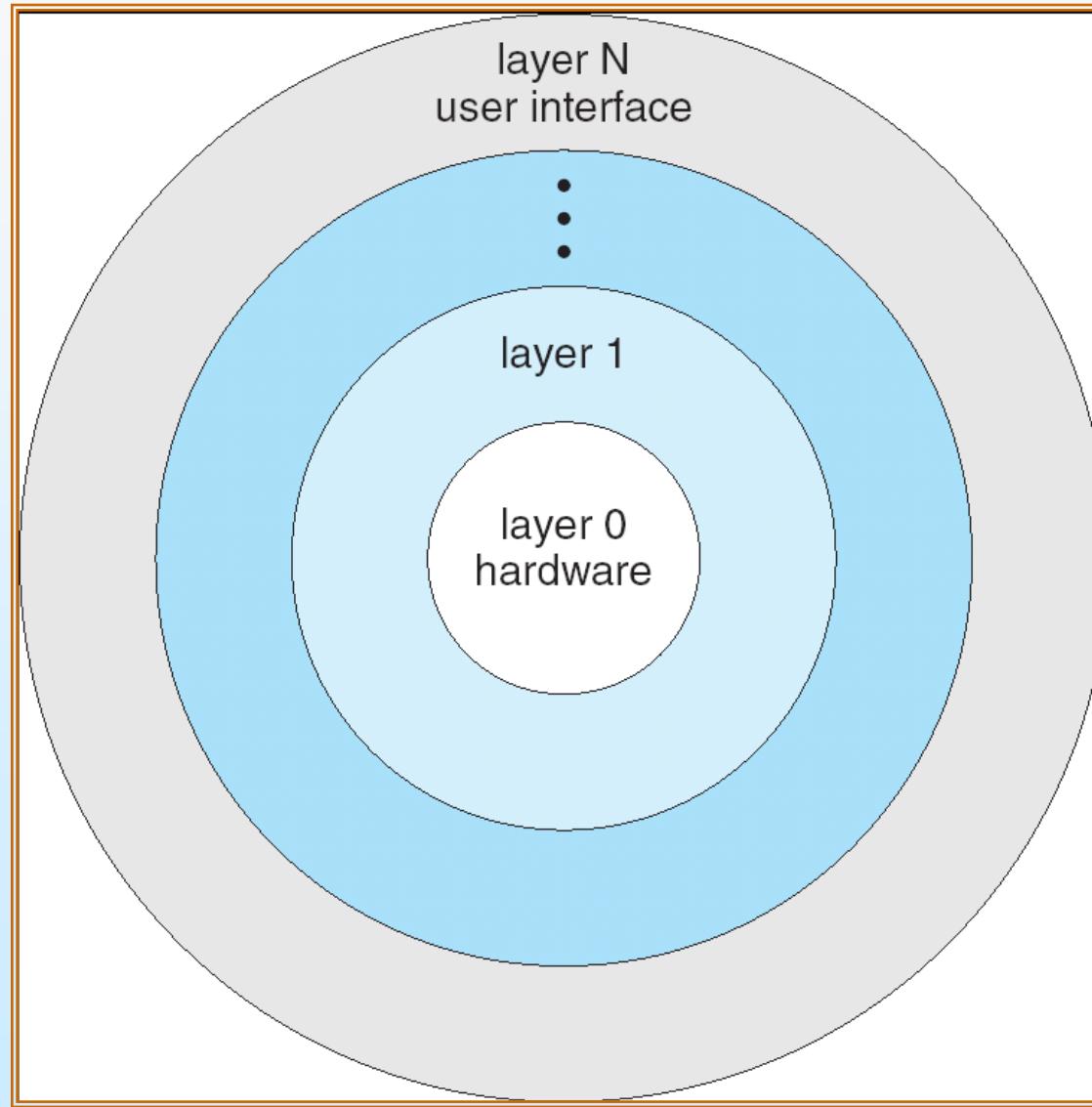
Layered Approach

- 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





Layered Operating System





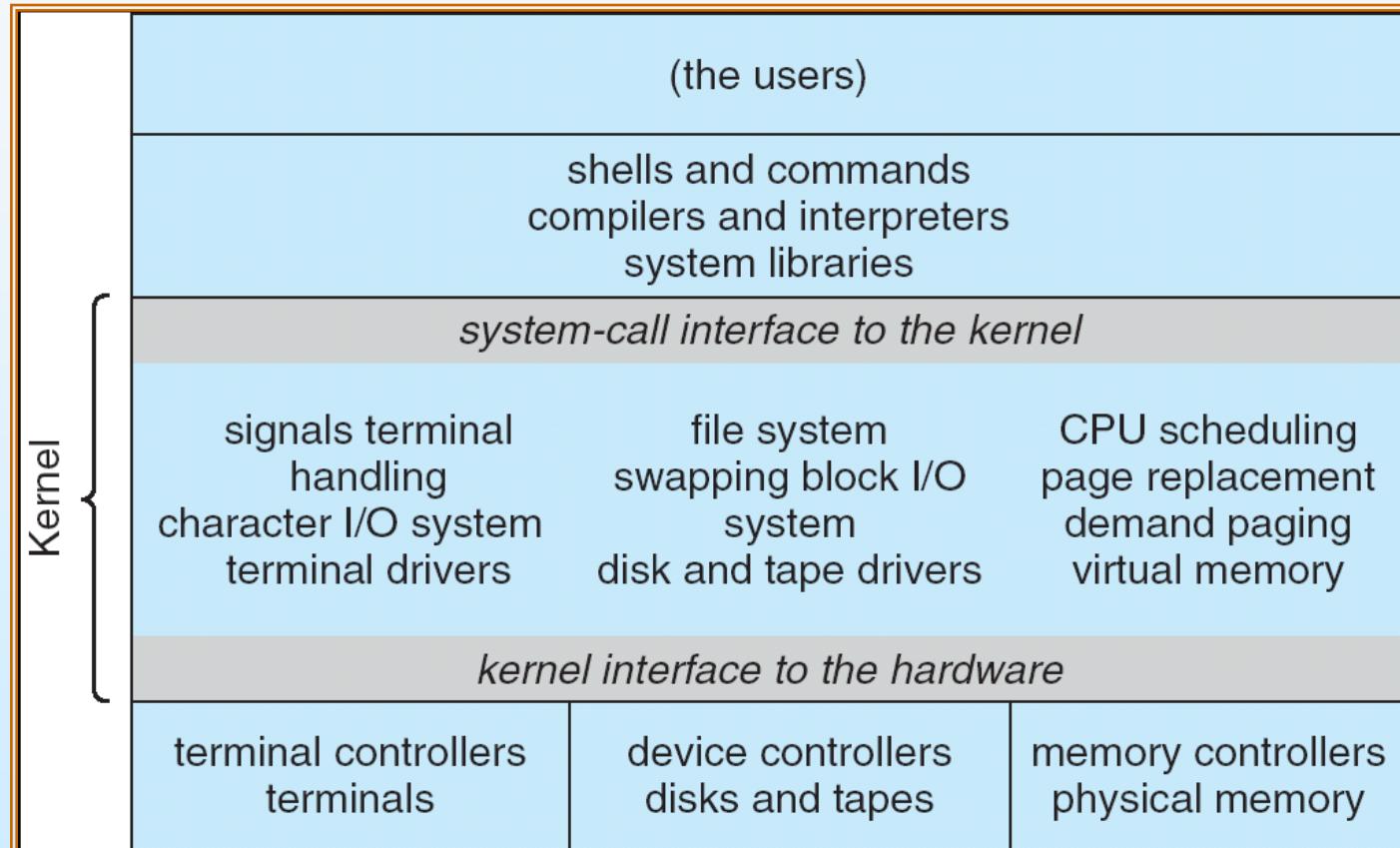
UNIX

- 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





UNIX System Structure





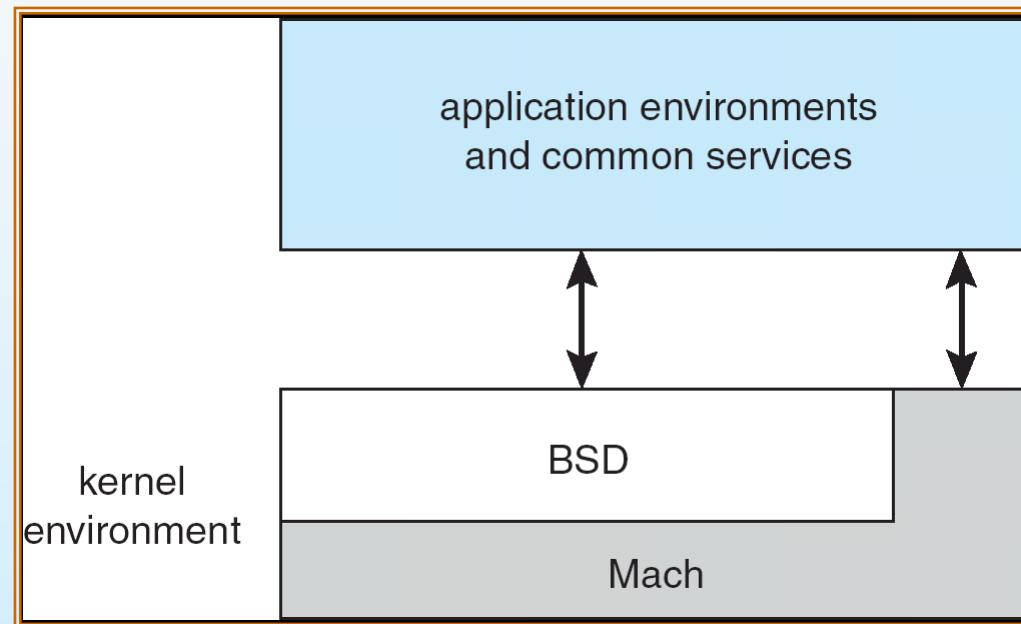
Microkernel System Structure

- Moves as much from the kernel into “*user*” space
- 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





Mac OS X Structure





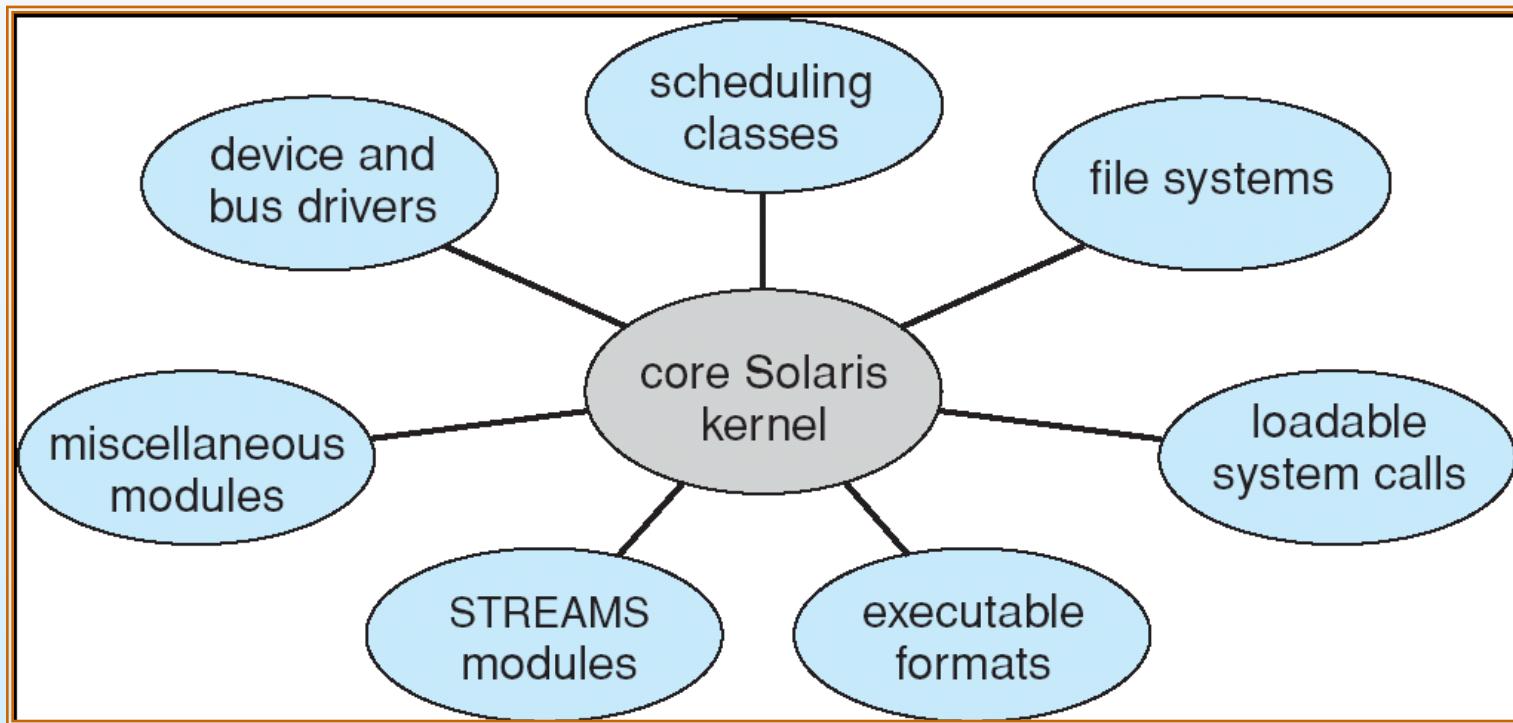
Modules

- Most modern operating systems implement kernel modules
 - 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





Solaris Modular Approach





Virtual Machines

- A *virtual machine* takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware
- A virtual machine provides an interface *identical* to the underlying bare hardware
- The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory





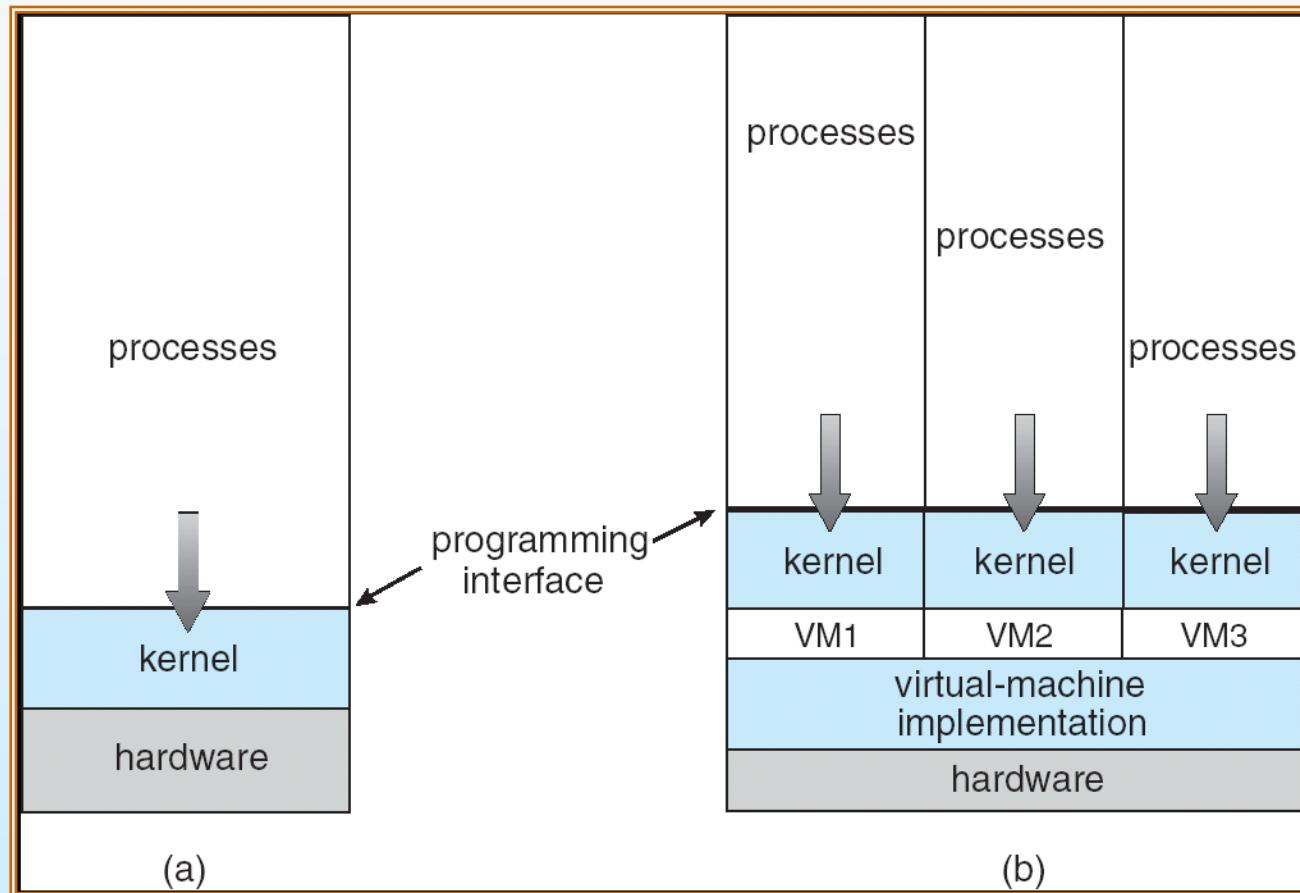
Virtual Machines (Cont.)

- The resources of the physical computer are shared to create the virtual machines
 - CPU scheduling can create the appearance that users have their own processor
 - Spooling and a file system can provide virtual card readers and virtual line printers
 - A normal user time-sharing terminal serves as the virtual machine operator's console





Virtual Machines (Cont.)



(a) Nonvirtual machine (b) virtual machine





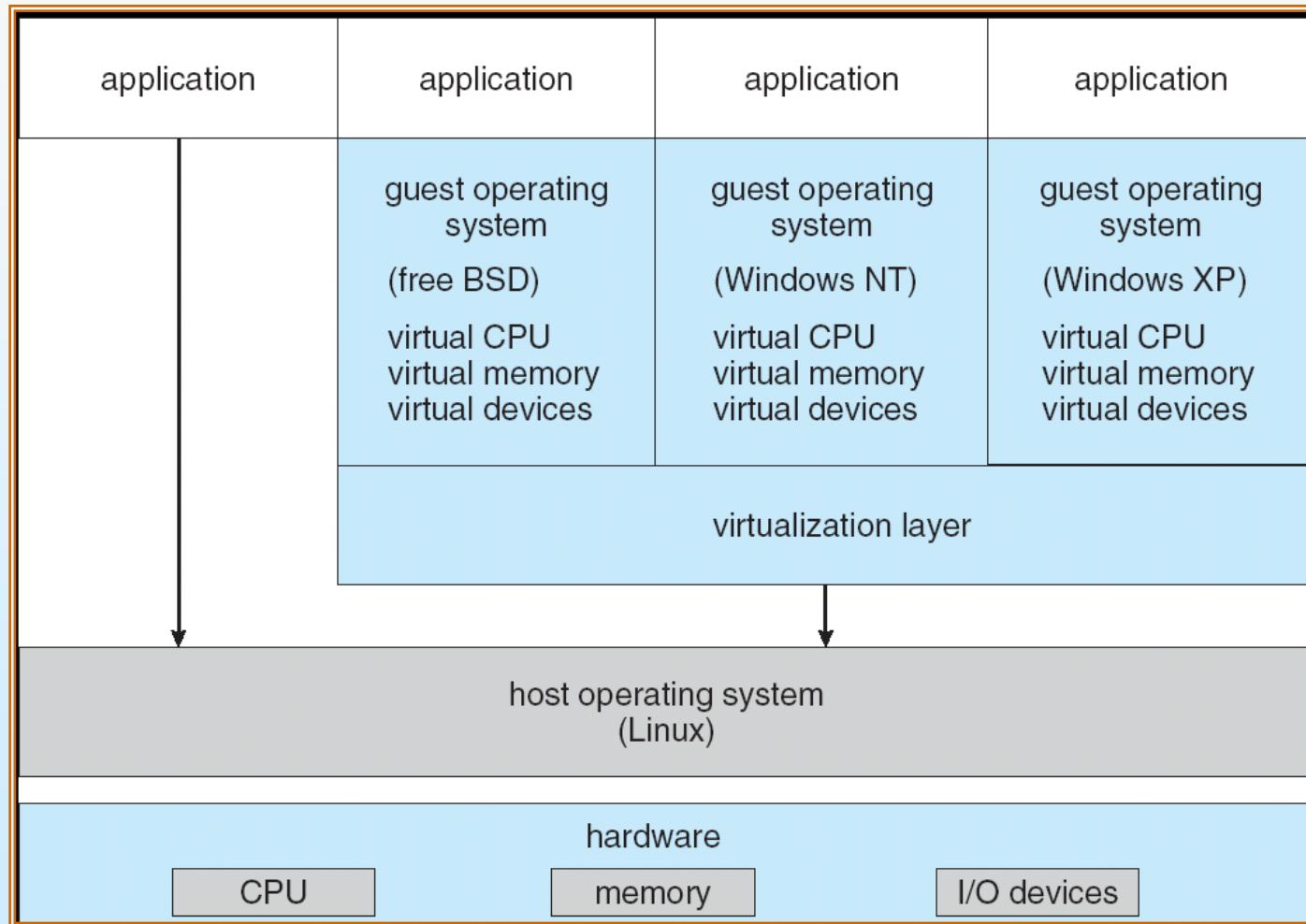
Virtual Machines (Cont.)

- The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources.
- A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
- The virtual machine concept is difficult to implement due to the effort required to provide an *exact* duplicate to the underlying machine



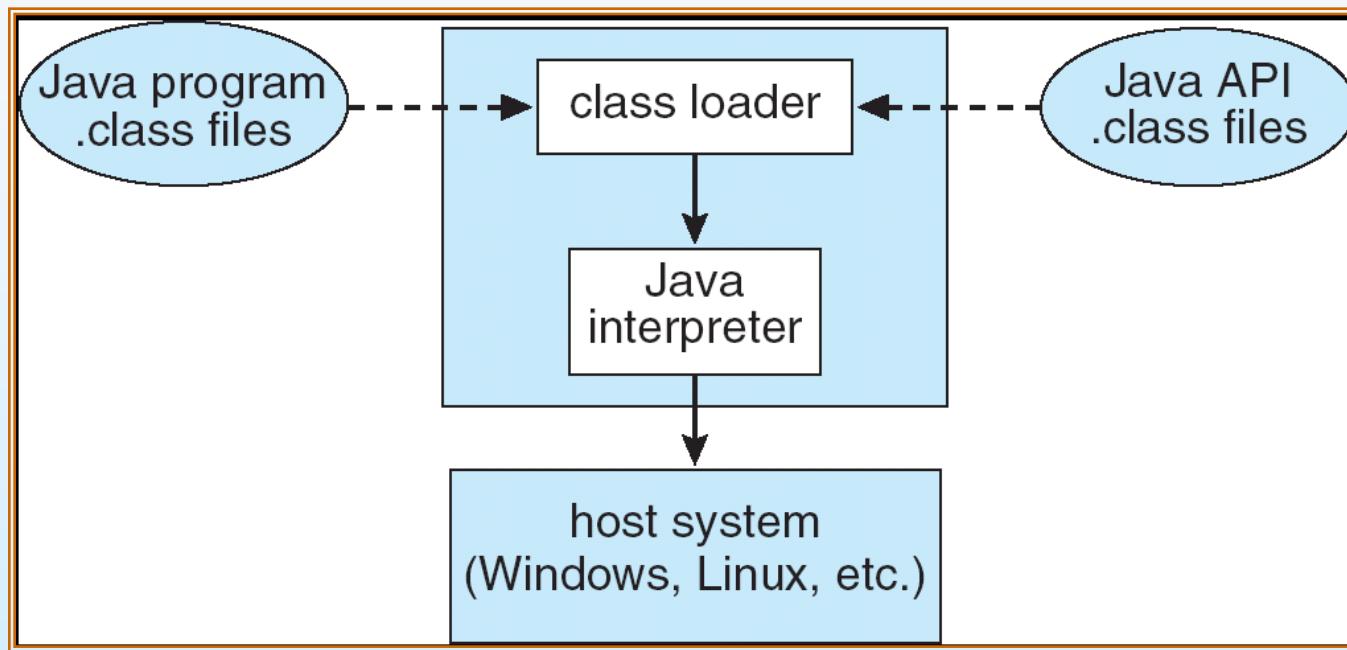


VMware Architecture





The Java Virtual Machine





Operating System Generation

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site
- SYSGEN program obtains information concerning the specific configuration of the hardware system
- *Booting* – starting a computer by loading the kernel
- *Bootstrap program* – code stored in ROM that is able to locate the kernel, load it into memory, and start its execution



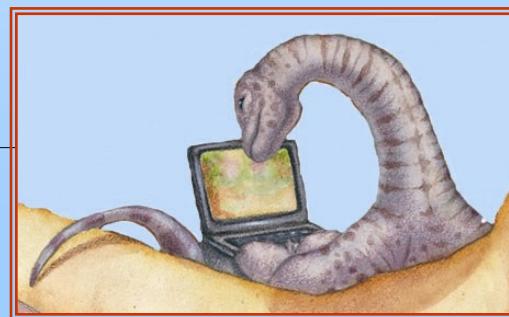


System Boot

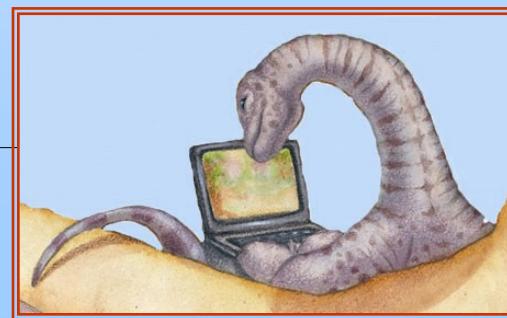
- Operating system must be made available to hardware so hardware can start it
 - Small piece of code – **bootstrap loader**, locates the kernel, loads it into memory, and starts it
 - Sometimes two-step process where **boot block** at fixed location loads bootstrap loader
 - When power initialized on system, execution starts at a fixed memory location
 - ▶ Firmware used to hold initial boot code



End of Chapter 2



Chapter 3: Processes





Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Cooperating Processes
- Interprocess Communication
- Communication in Client-Server Systems





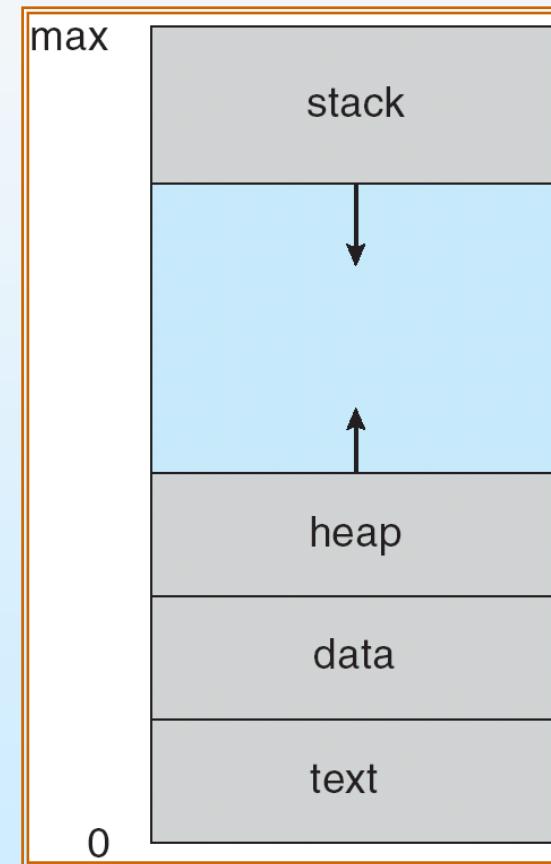
Process Concept

- An operating system executes a variety of programs:
 - Batch system – jobs
 - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
 - program counter
 - stack
 - data section





Process in Memory





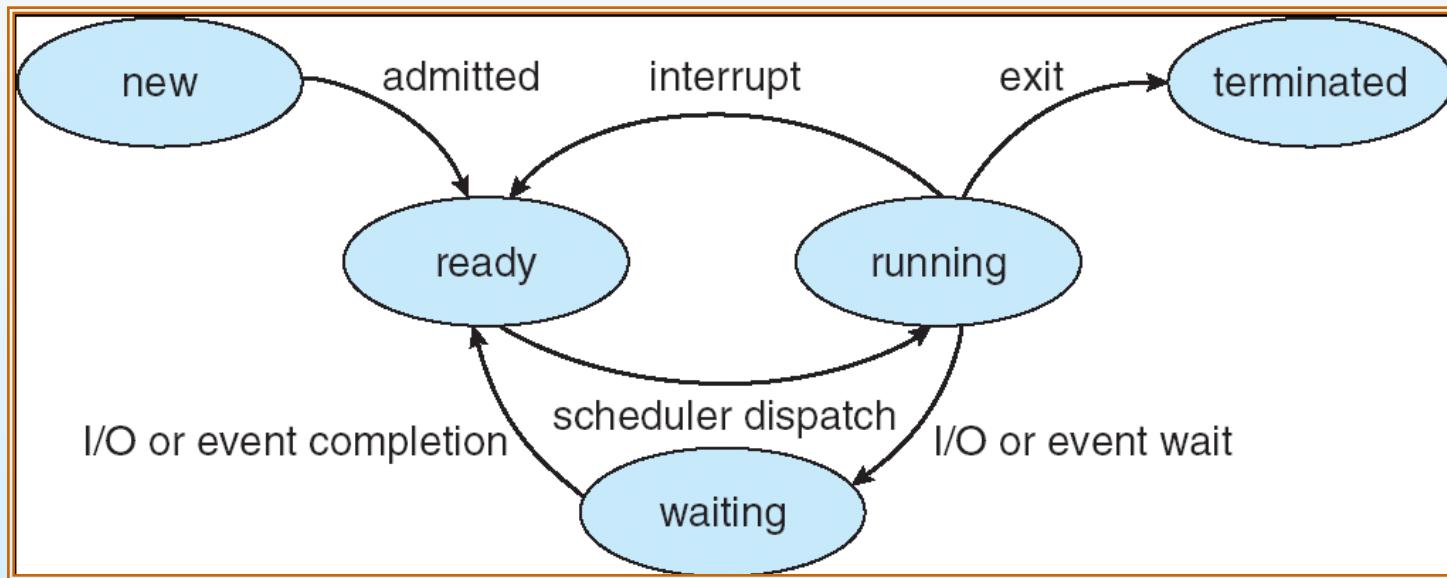
Process State

- 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 process
 - **terminated**: The process has finished execution





Diagram of Process State





Process Control Block (PCB)

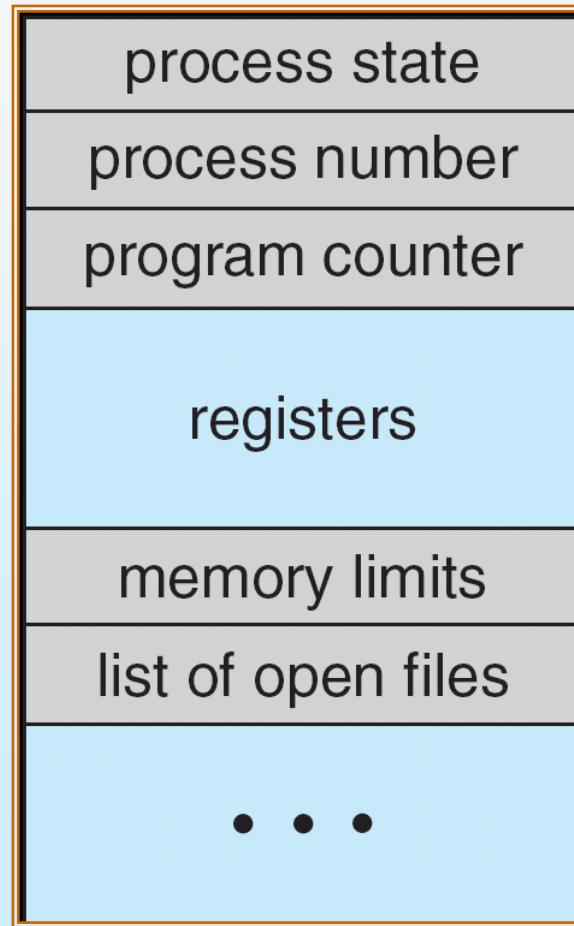
Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information



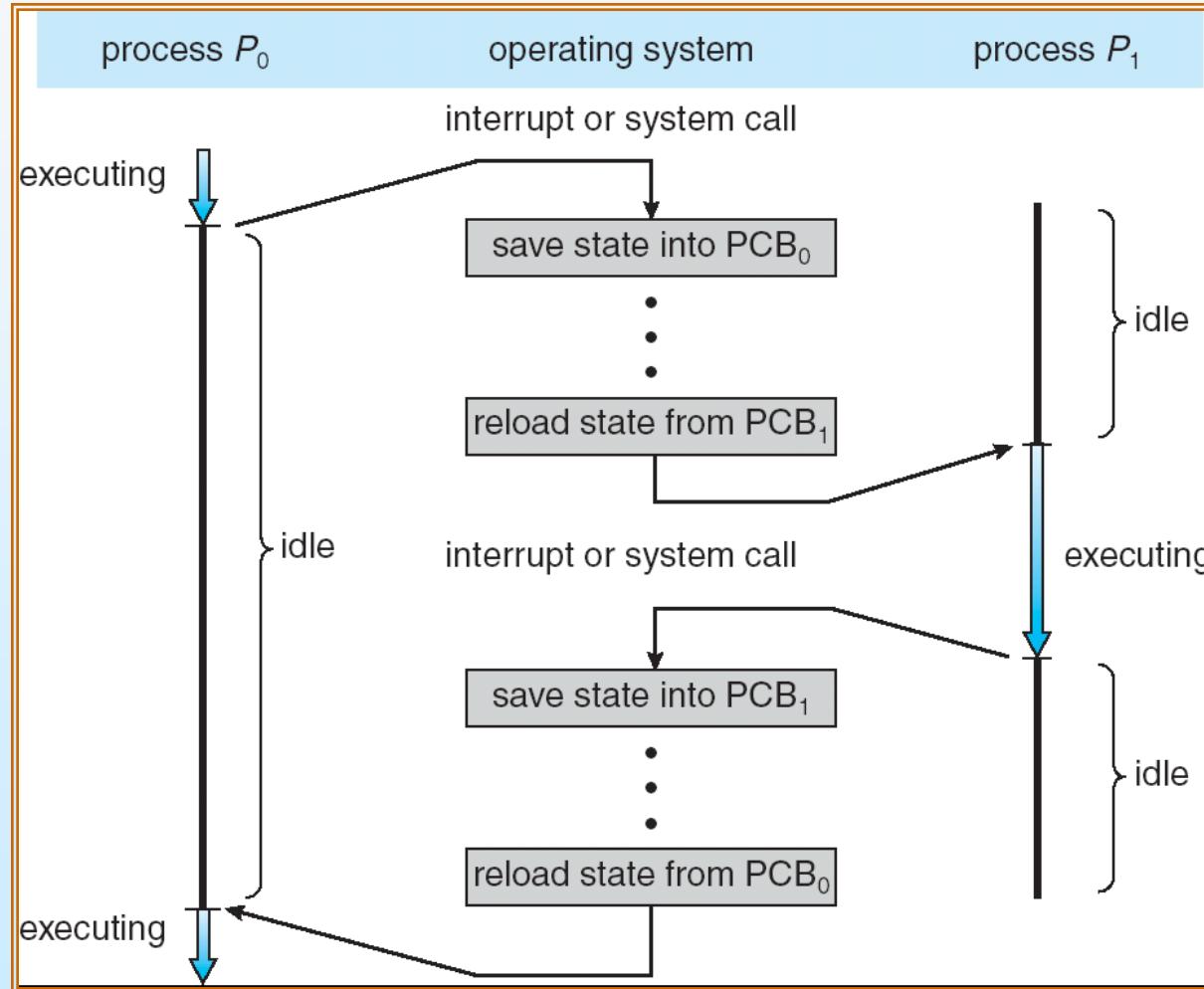


Process Control Block (PCB)





CPU Switch From Process to Process





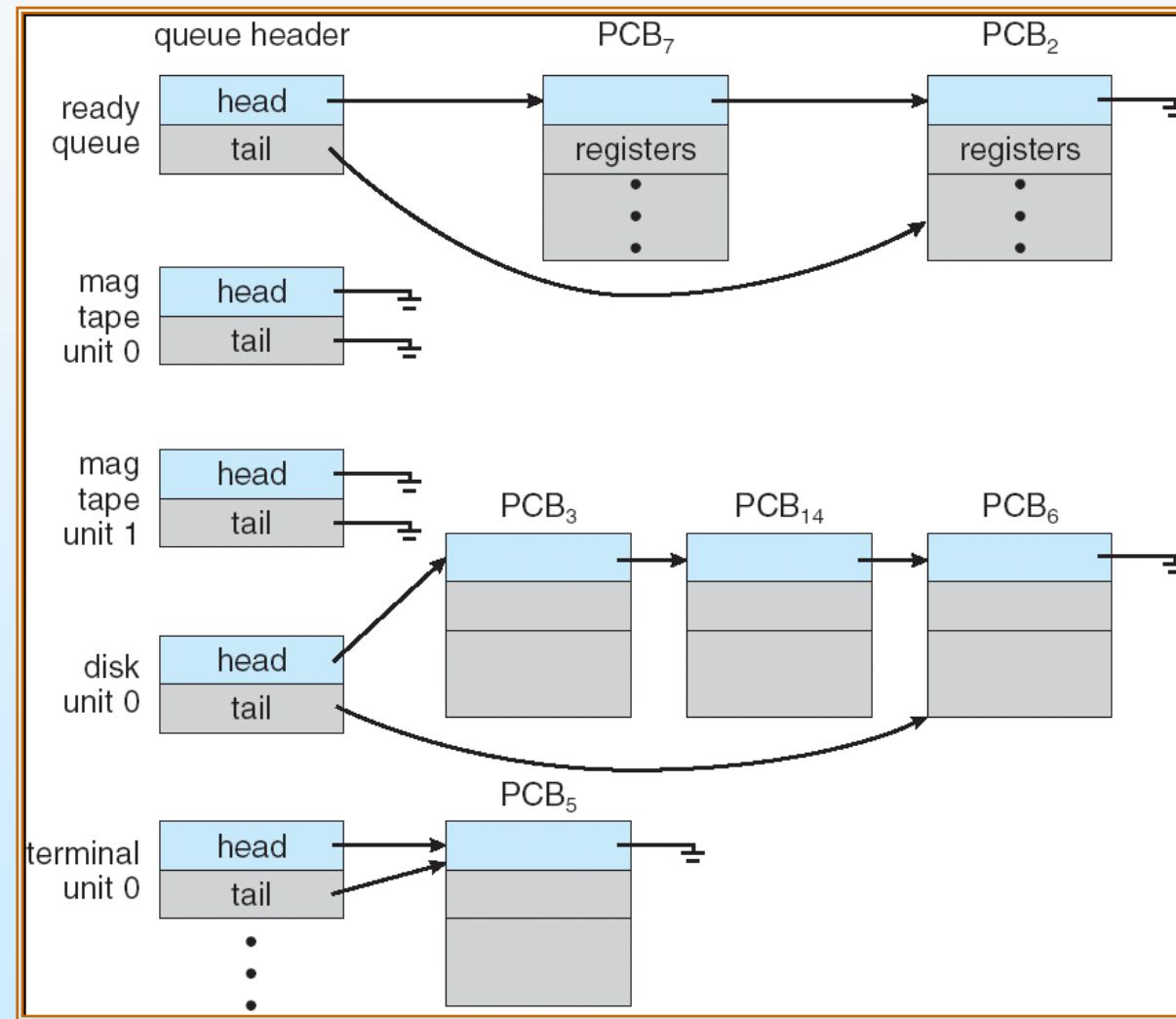
Process Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues



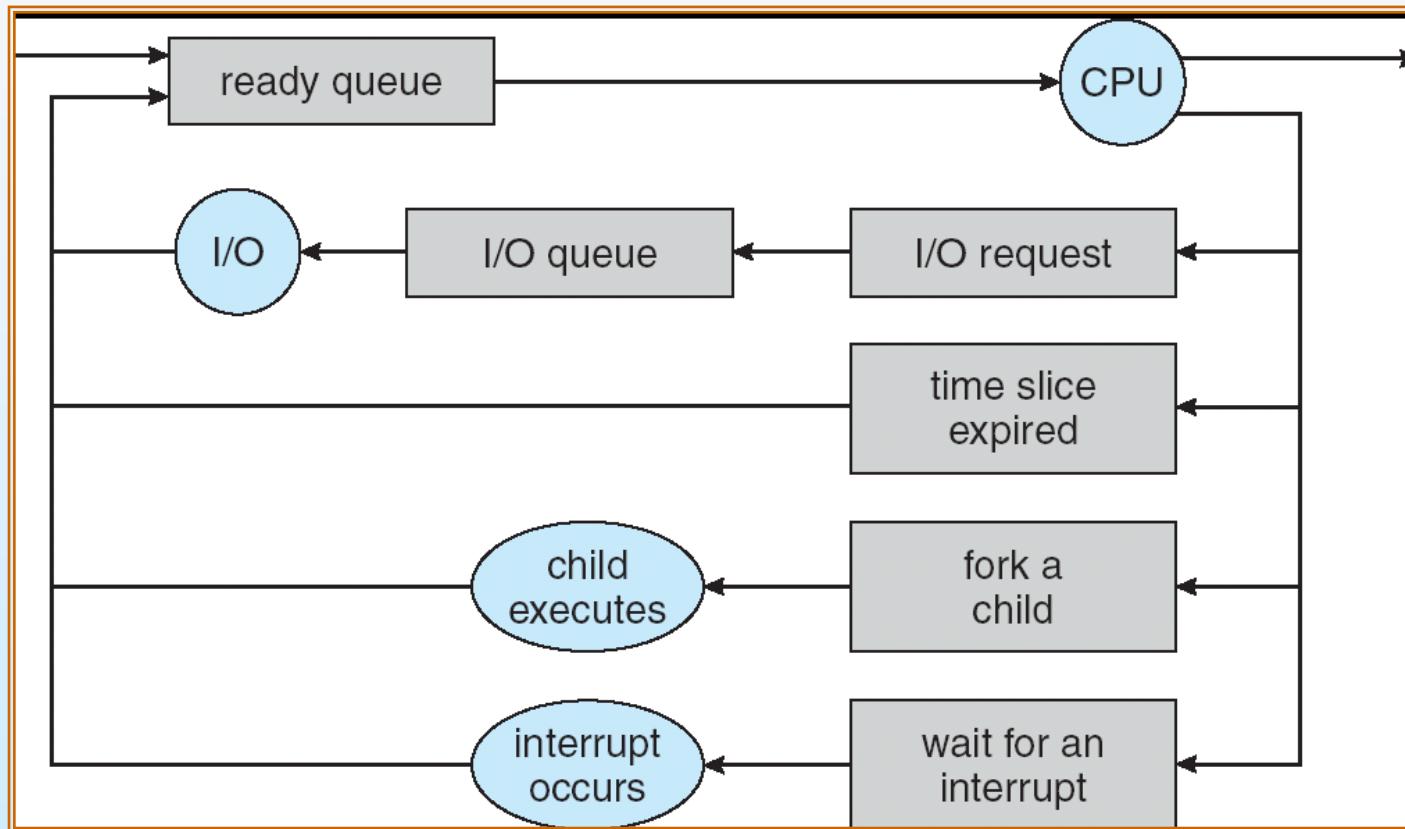


Ready Queue And Various I/O Device Queues





Representation of Process Scheduling





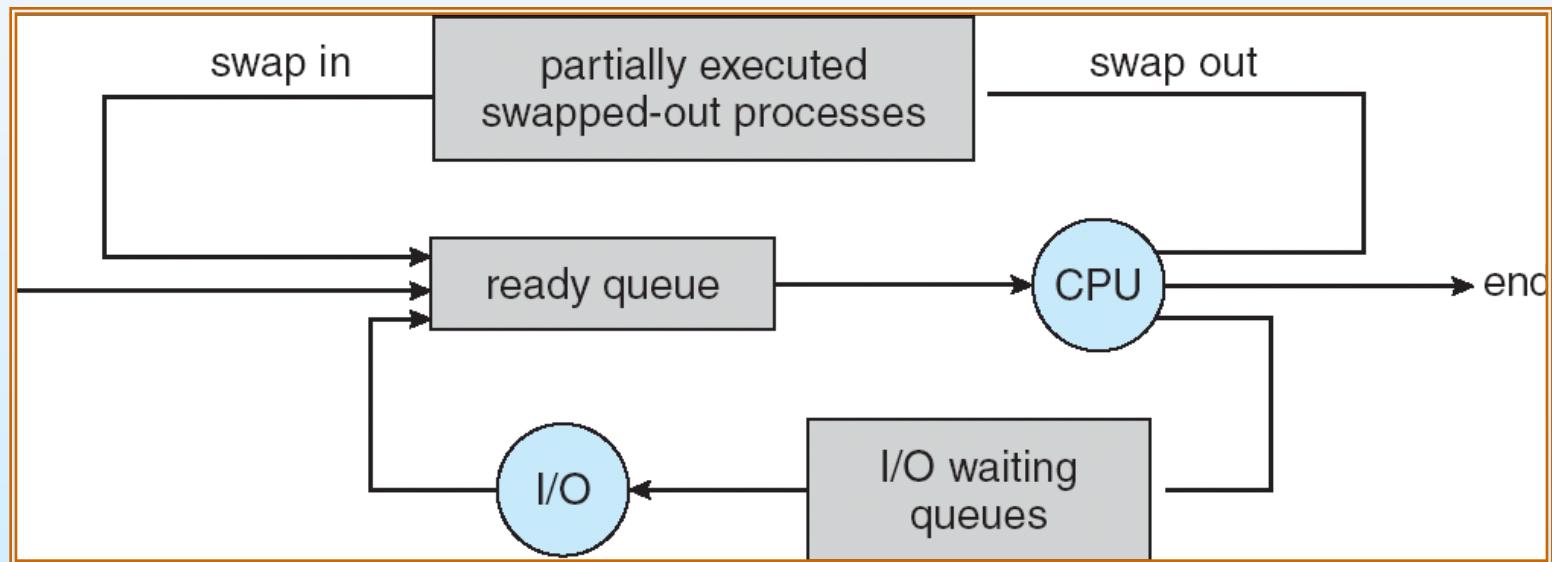
Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU





Addition of Medium Term Scheduling





Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
 - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
 - **CPU-bound process** – spends more time doing computations; few very long CPU bursts





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
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support





Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution
 - Parent and children execute concurrently
 - Parent waits until children terminate



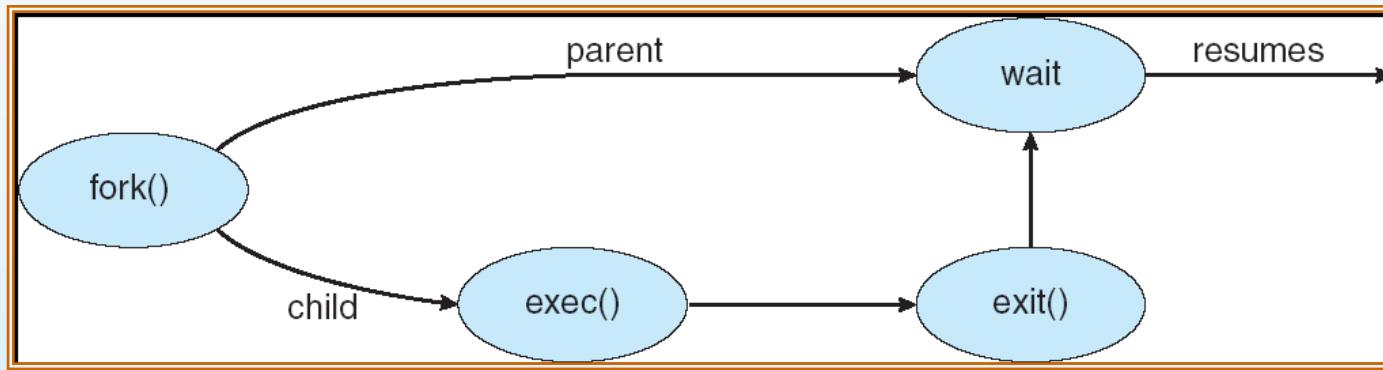


Process Creation (Cont.)

- 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



Process Creation





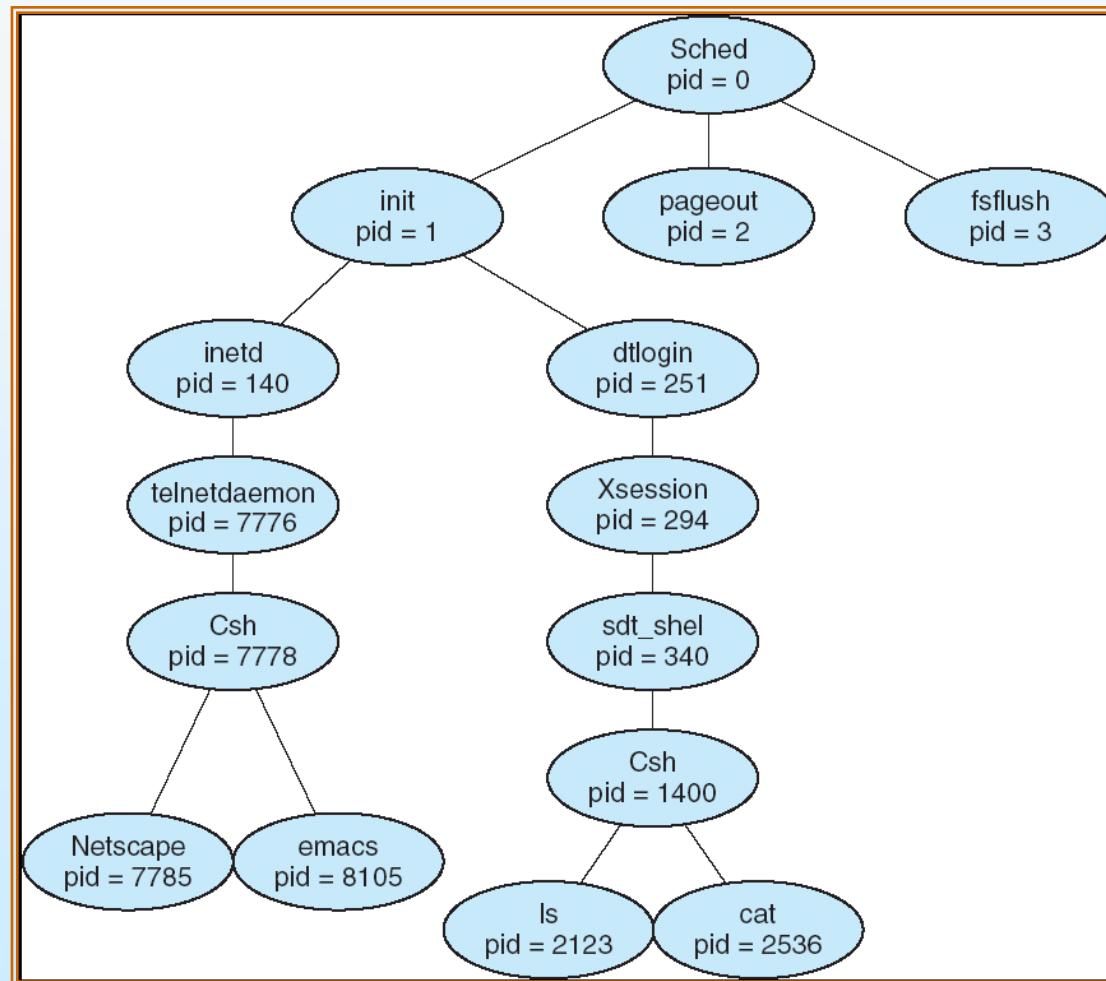
C Program Forking Separate Process

```
int main()
{
    Pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-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");
        exit(0);
    }
}
```





A tree of processes on a typical Solaris





Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
 - Output data from child to parent (via **wait**)
 - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - If parent is exiting
 - ▶ Some operating systems do not allow child to continue if its parent terminates
 - All children terminated - *cascading termination*





Cooperating Processes

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





Producer-Consumer Problem

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





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





Bounded-Buffer – Insert() Method

```
while (true) {  
    /* Produce an item */  
  
    while (((in = (in + 1) % BUFFER SIZE count) == out)  
        ; /* do nothing -- no free buffers */  
  
    buffer[in] = item;  
  
    in = (in + 1) % BUFFER SIZE;  
  
}
```





Bounded Buffer – Remove() Method

```
while (true) {  
    while (in == out)  
        ; // do nothing -- nothing to consume  
  
    // remove an item from the buffer  
    item = buffer[out];  
    out = (out + 1) % BUFFER SIZE;  
    return item;  
}
```





Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
 - **send(message)** – message size fixed or variable
 - **receive(message)**
- If P and Q wish to communicate, they need to:
 - establish a *communication link* between them
 - exchange messages via send/receive
- Implementation of communication link
 - physical (e.g., shared memory, hardware bus)
 - logical (e.g., logical properties)





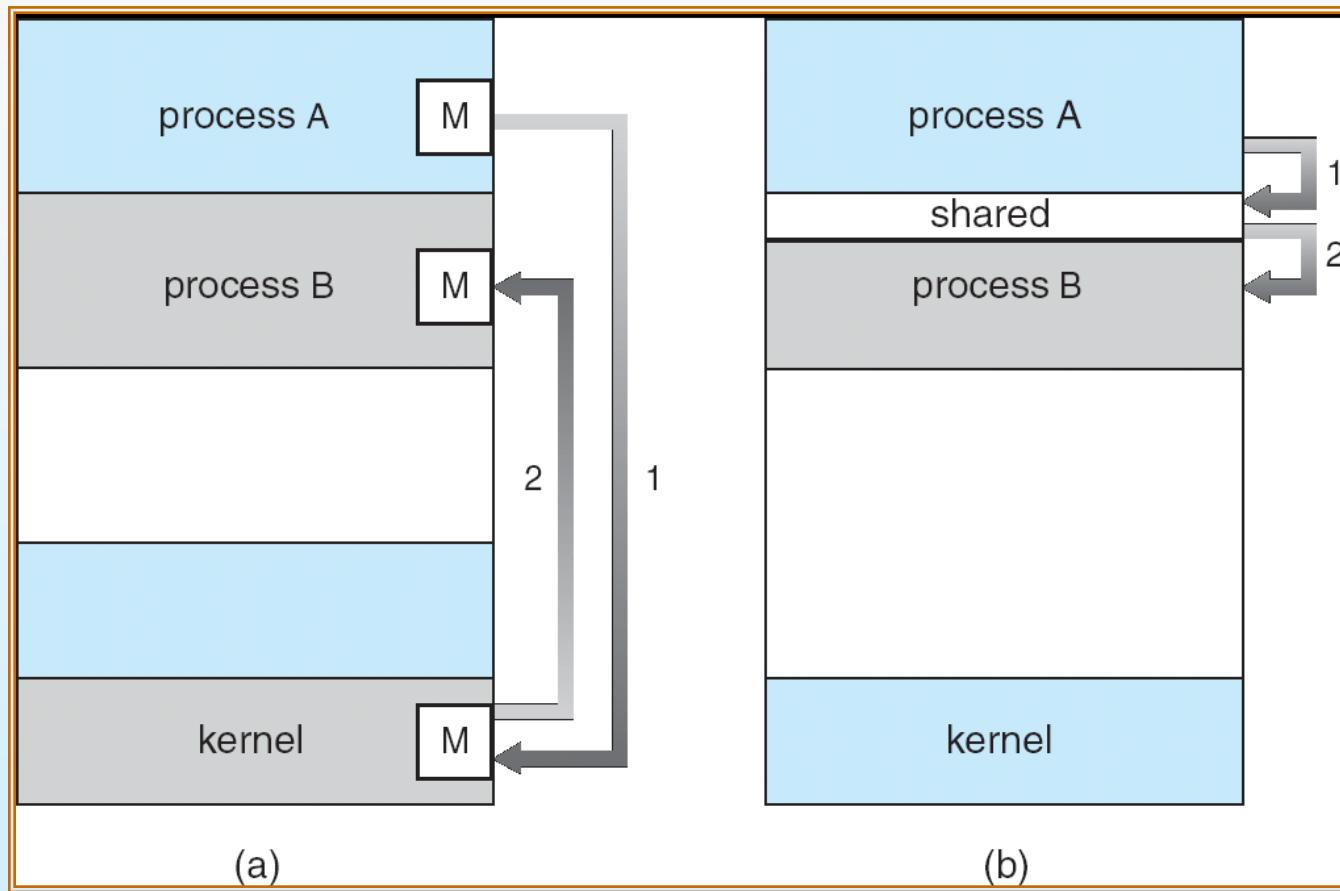
Implementation Questions

- 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?





Communications Models





Direct Communication

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





Indirect Communication

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





Indirect Communication

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





Indirect Communication

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





Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
 - **Blocking send** has the sender block until the message is received
 - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
 - **Non-blocking** send has the sender send the message and continue
 - **Non-blocking** receive has the receiver receive a valid message or null





Buffering

- Queue of messages attached to the link; implemented in one of three ways
 1. Zero capacity – 0 messages
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





Client-Server Communication

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





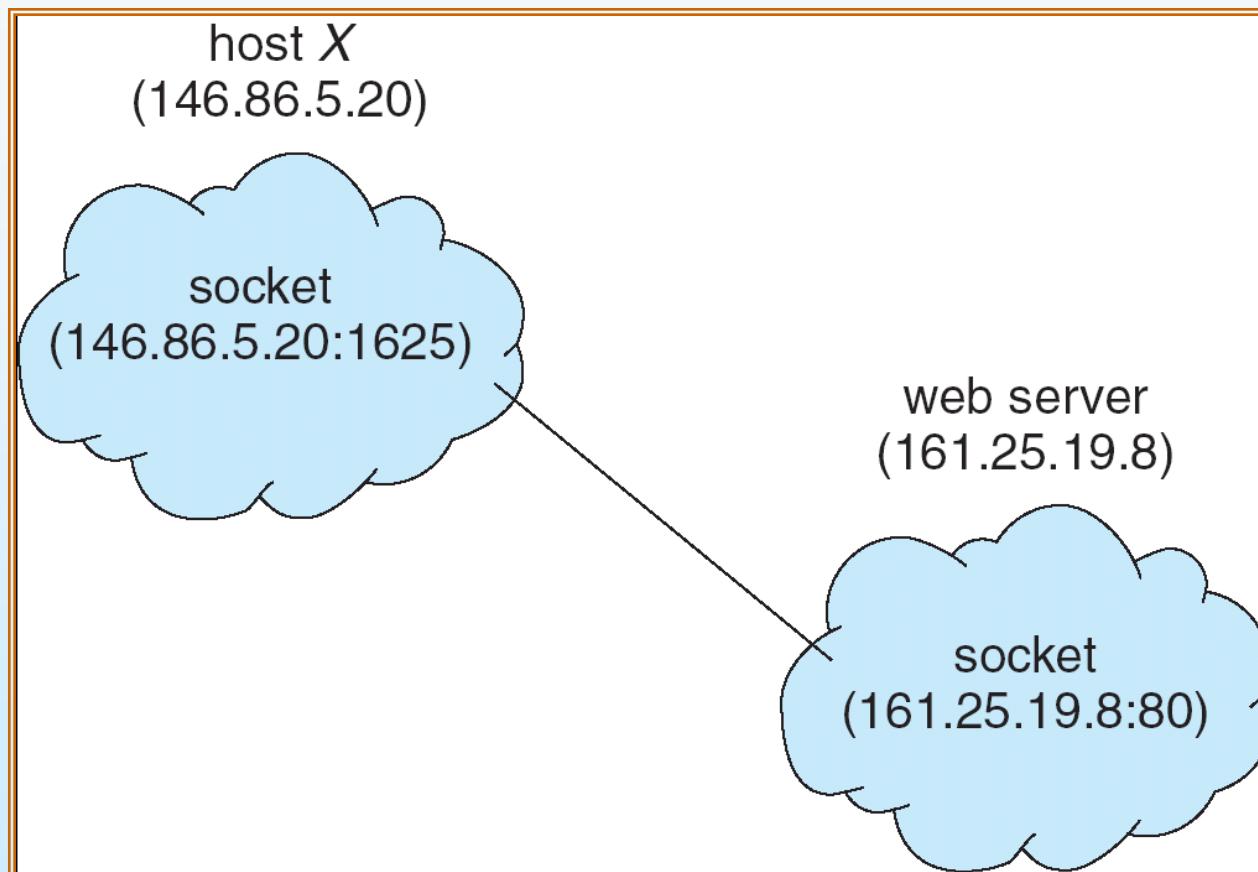
Sockets

- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets





Socket Communication





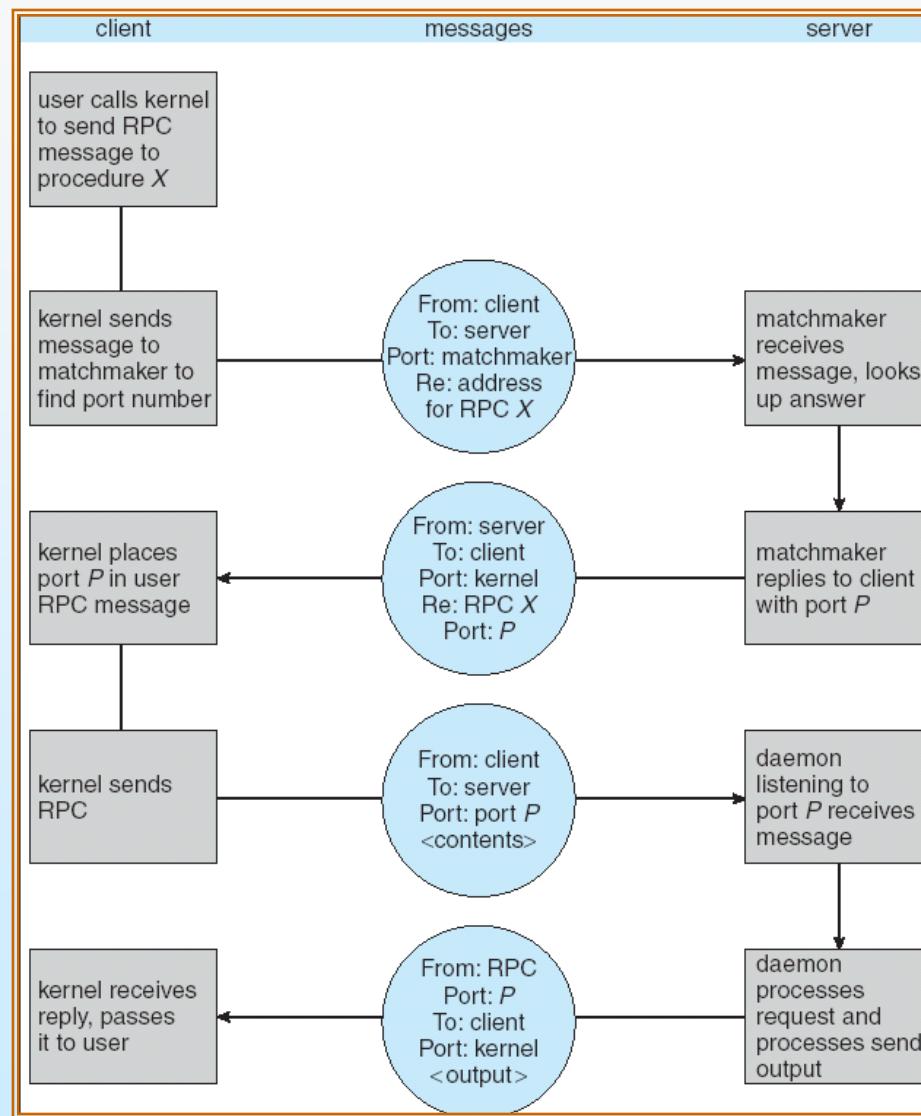
Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- **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.





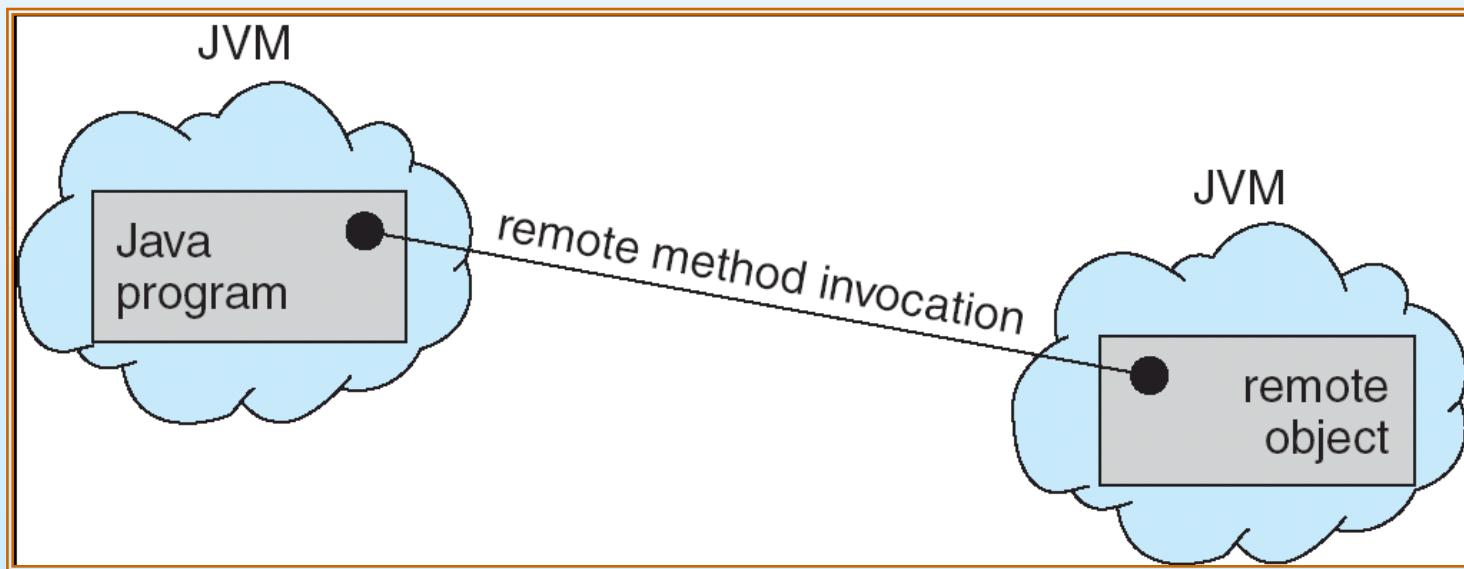
Execution of RPC





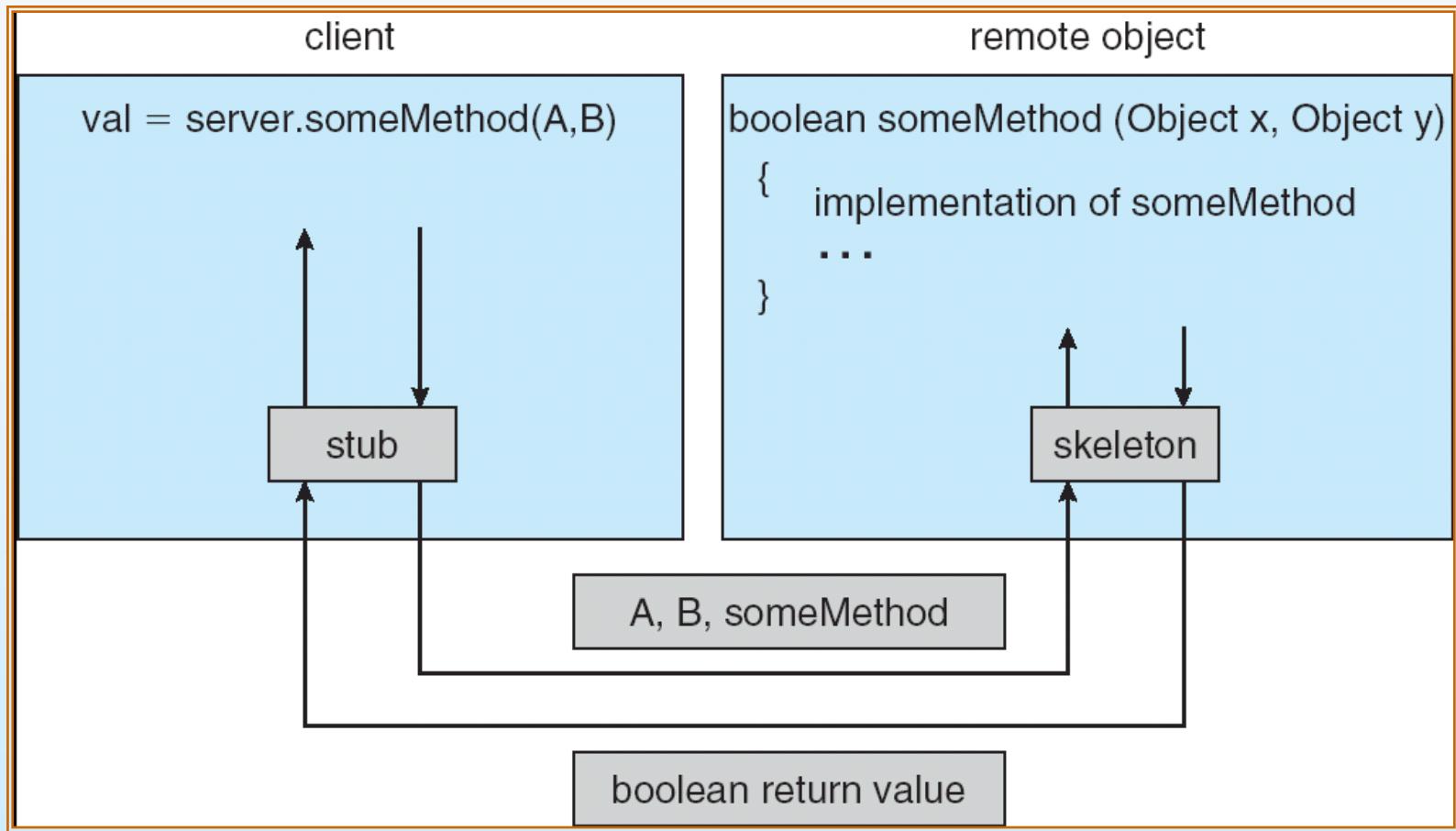
Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.
- RMI allows a Java program on one machine to invoke a method on a remote object.

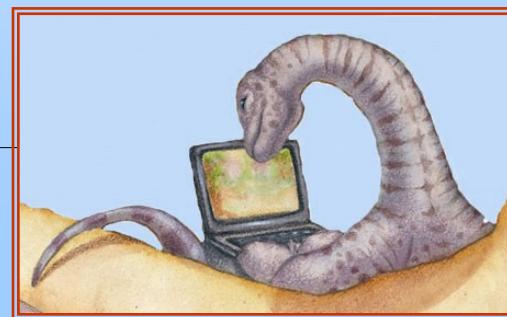




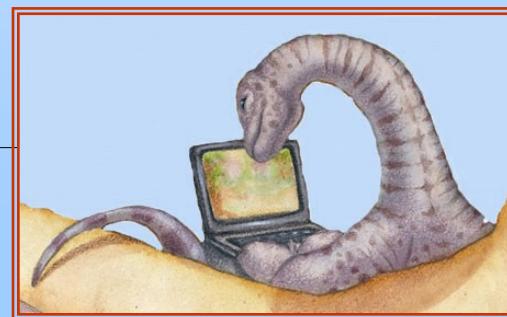
Marshalling Parameters



End of Chapter 3



Chapter 4: Threads





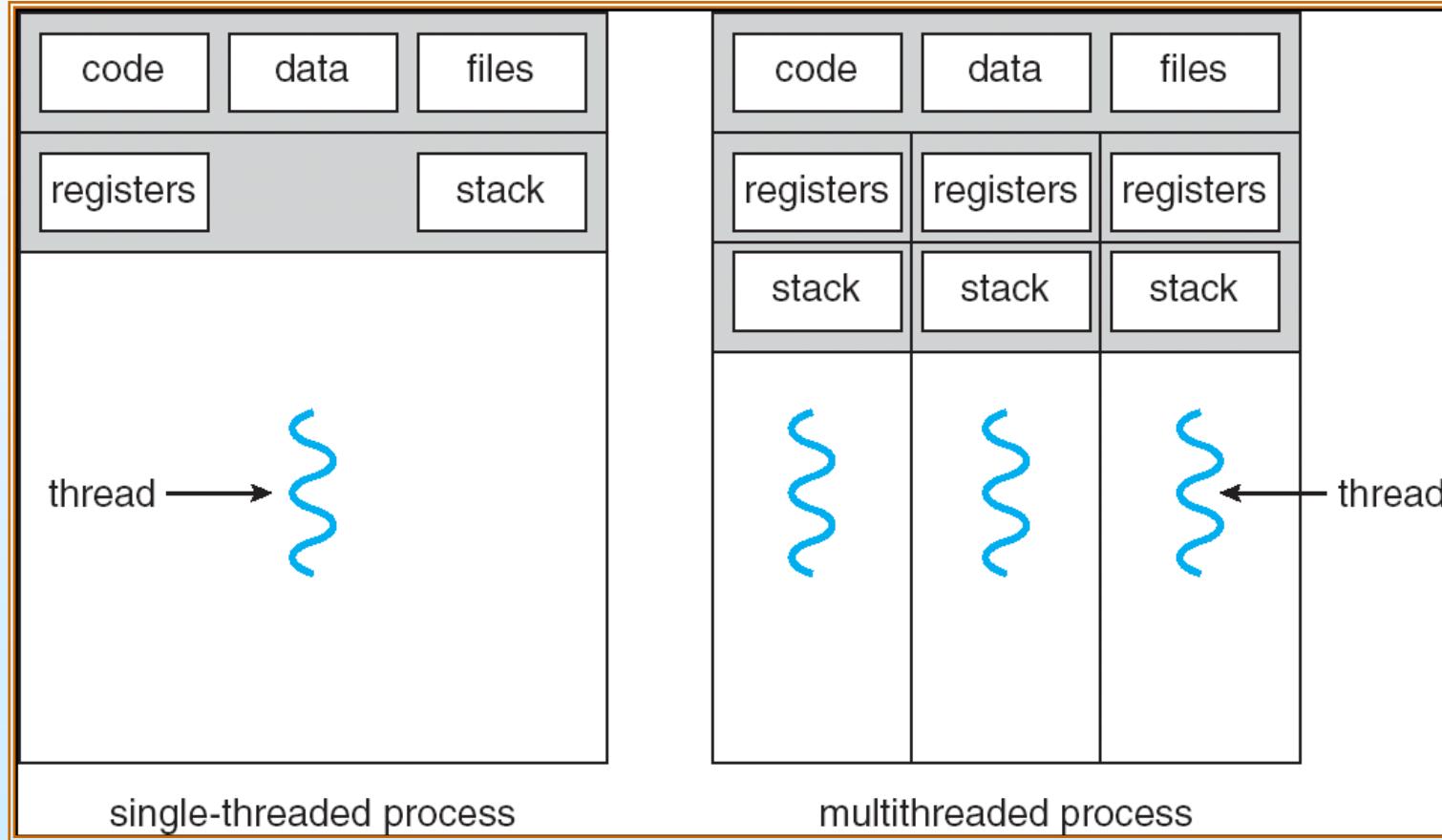
Chapter 4: Threads

- Overview
- Multithreading Models
- Threading Issues
- Pthreads
- Windows XP Threads
- Linux Threads
- Java Threads





Single and Multithreaded Processes





Benefits

- Responsiveness
- Resource Sharing
- Economy
- Utilization of MP Architectures





User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
 - POSIX Pthreads
 - Win32 threads
 - Java threads





Kernel Threads

- Supported by the Kernel
- Examples
 - Windows XP/2000
 - Solaris
 - Linux
 - Tru64 UNIX
 - Mac OS X





Multithreading Models

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





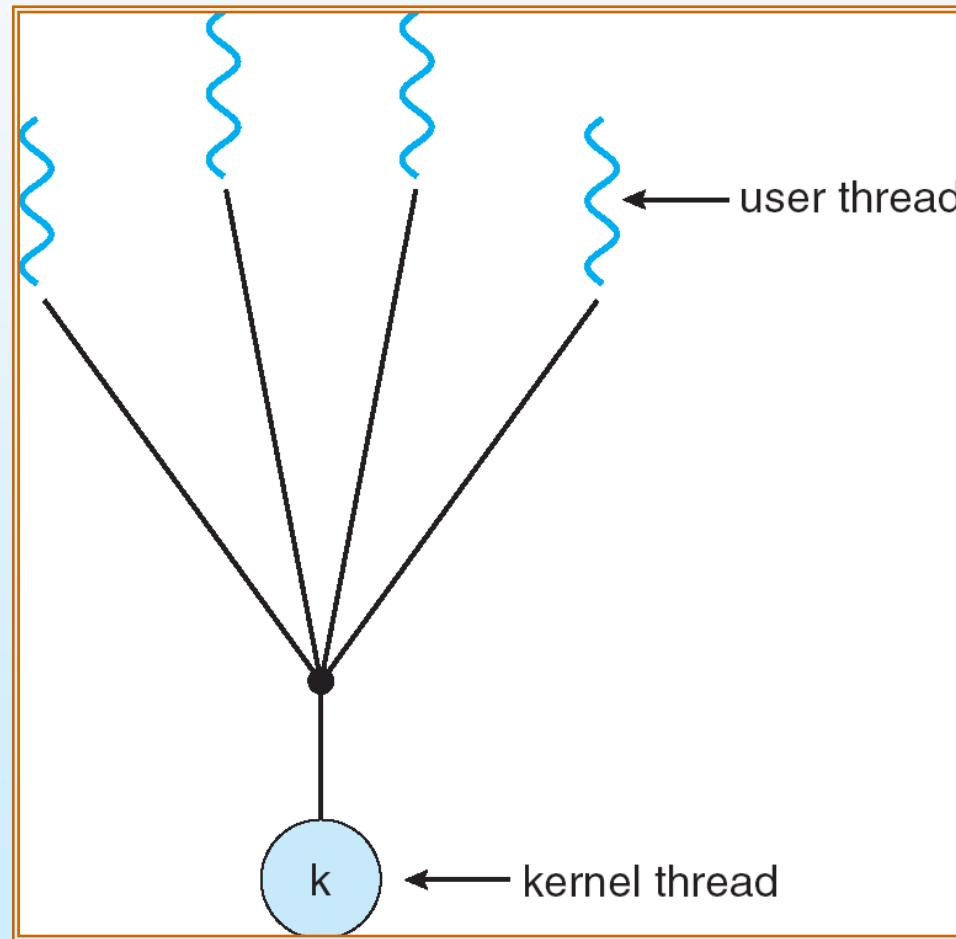
Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
 - Solaris Green Threads
 - GNU Portable Threads





Many-to-One Model





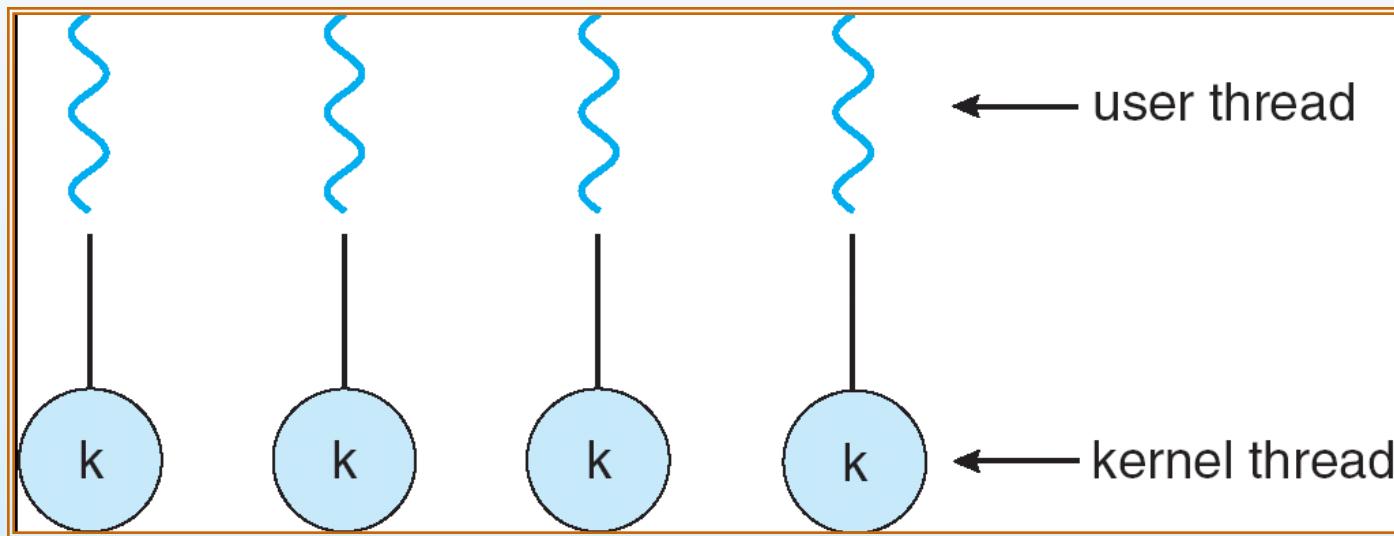
One-to-One

- Each user-level thread maps to kernel thread
- Examples
 - Windows NT/XP/2000
 - Linux
 - Solaris 9 and later





One-to-one Model





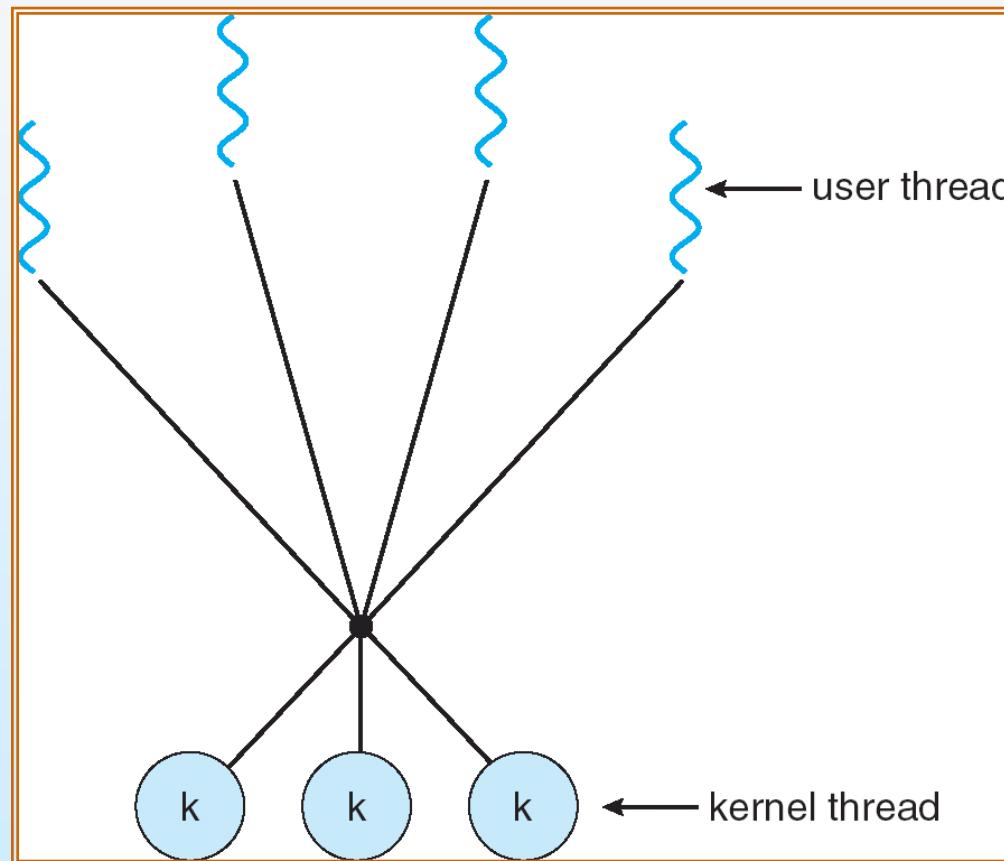
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package





Many-to-Many Model





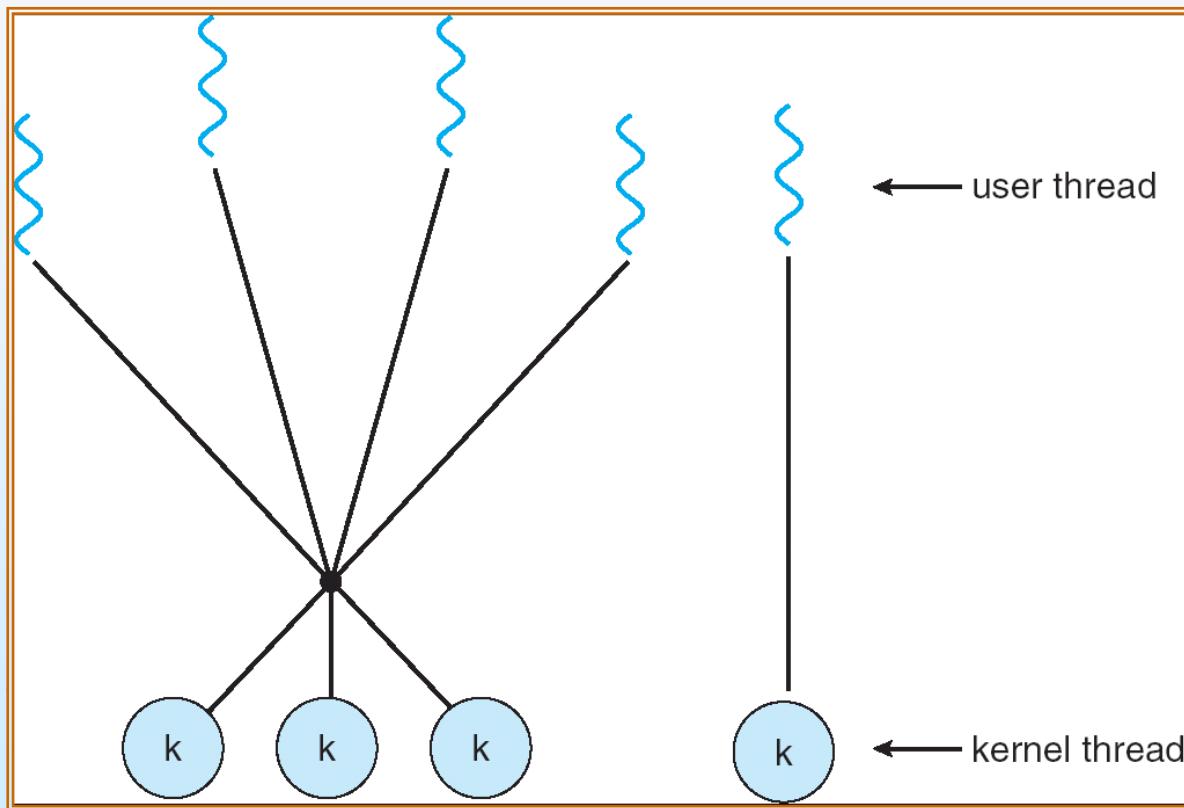
Two-level Model

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





Two-level Model





Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data
- Scheduler activations





Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?





Thread Cancellation

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





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
- Options:
 - 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 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





Thread Specific Data

- 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)





Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads





Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)





Windows XP Threads

- Implements the one-to-one mapping
- Each thread contains
 - A thread id
 - Register set
 - Separate user and kernel stacks
 - Private data storage area
- The register set, stacks, and private storage area are known as the **context** of the threads
- The primary data structures of a thread include:
 - ETHREAD (executive thread block)
 - KTHREAD (kernel thread block)
 - TEB (thread environment block)





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)





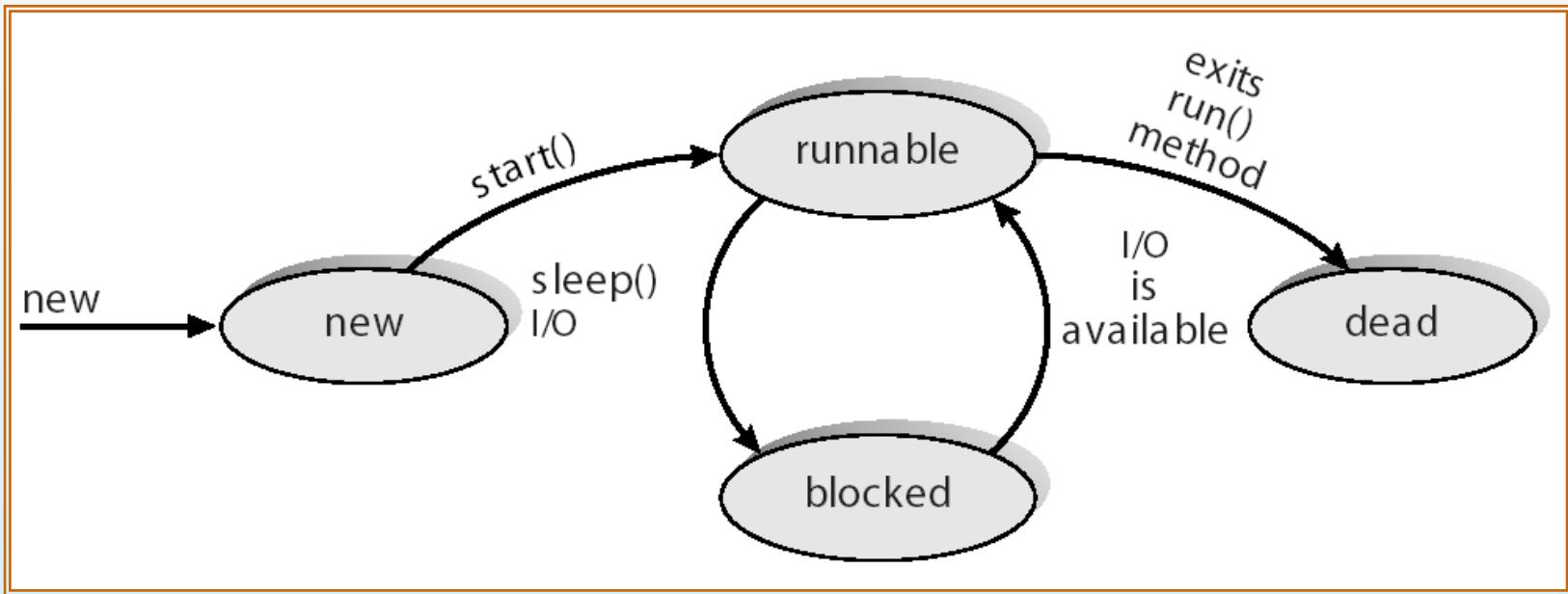
Java Threads

- Java threads are managed by the JVM
- Java threads may be created by:
 - Extending Thread class
 - Implementing the Runnable interface

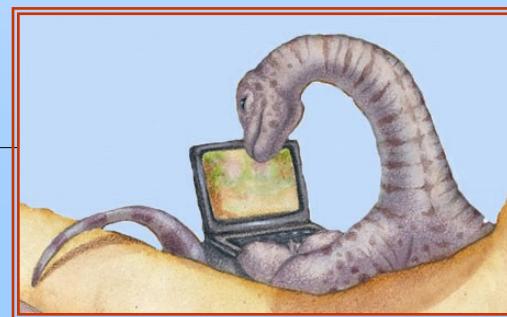




Java Thread States



End of Chapter 4



Chapter 5: CPU Scheduling





Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Real-Time Scheduling
- Thread Scheduling
- Operating Systems Examples
- Java Thread Scheduling
- Algorithm Evaluation





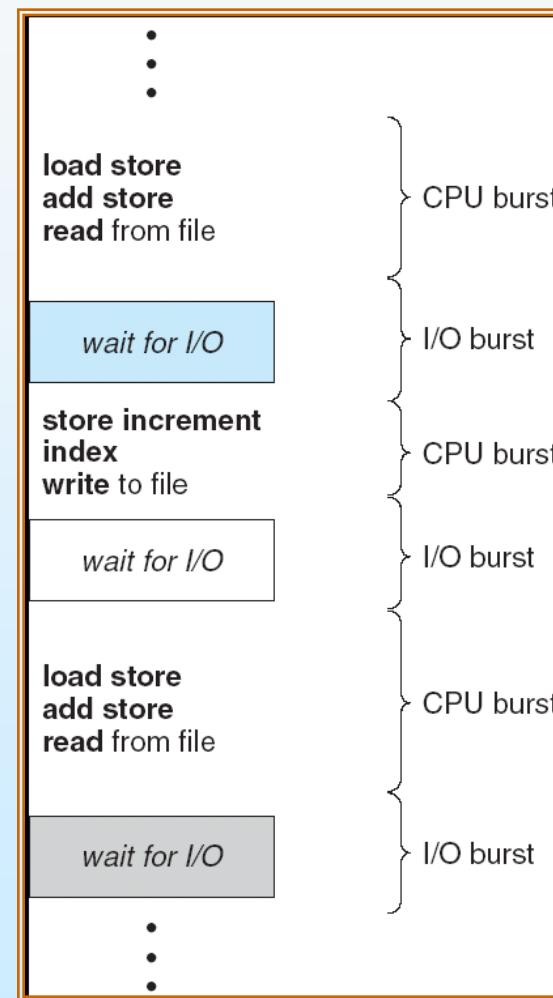
Basic Concepts

- 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 distribution



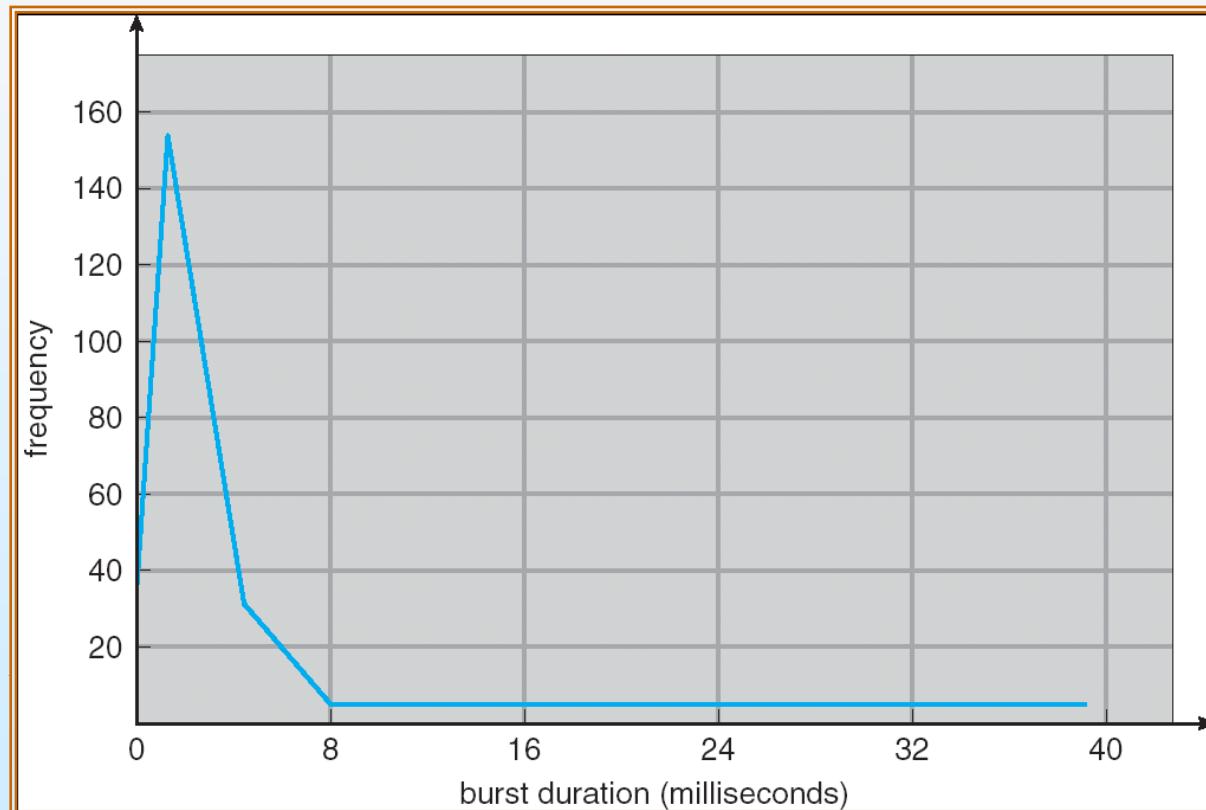


Alternating Sequence of CPU And I/O Bursts





Histogram of CPU-burst Times





CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
 1. Switches from running to waiting state
 2. Switches from running to ready state
 3. Switches from waiting to ready
 4. Terminates
- Scheduling under 1 and 4 is *nonpreemptive*
- All other scheduling is *preemptive*





Dispatcher

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





Scheduling Criteria

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





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





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
- Two schemes:
 - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
 - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF)
- SJF is optimal – gives minimum average waiting time for a given set of processes

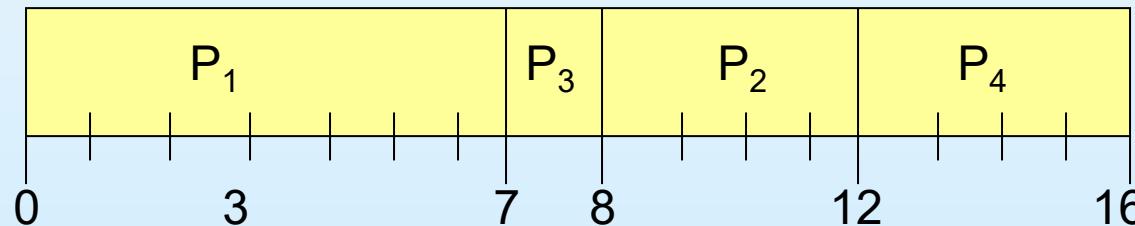




Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)



- Average waiting time = $(0 + 6 + 3 + 7)/4 = 4$

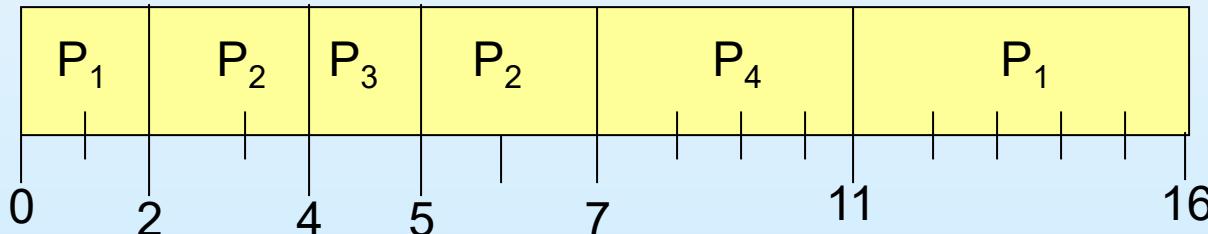




Example of Preemptive SJF

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Average waiting time = $(9 + 1 + 0 + 2)/4 = 3$





Determining Length of Next CPU Burst

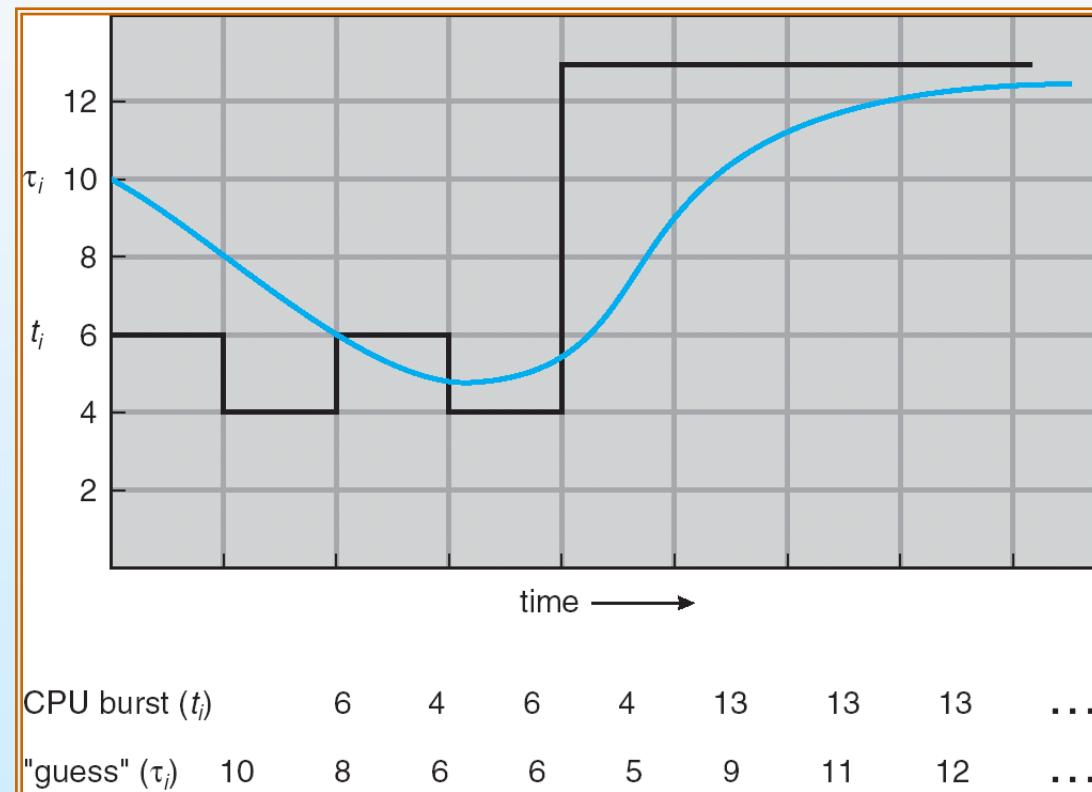
- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

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





Prediction of the Length of the Next CPU Burst





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





Priority Scheduling

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





Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), 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.
- Performance
 - q large \Rightarrow FIFO
 - q small \Rightarrow q must be large with respect to context switch, otherwise overhead is too high

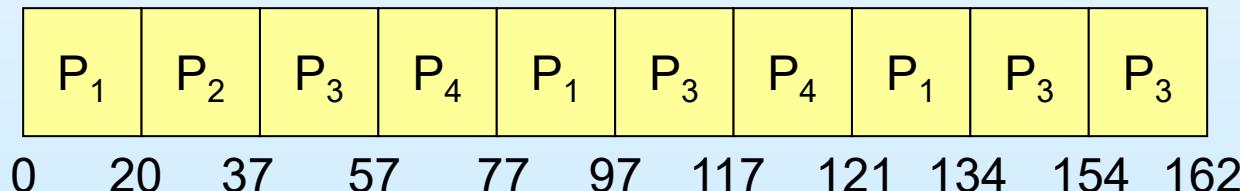




Example of RR with Time Quantum = 20

<u>Process</u>	<u>Burst Time</u>
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

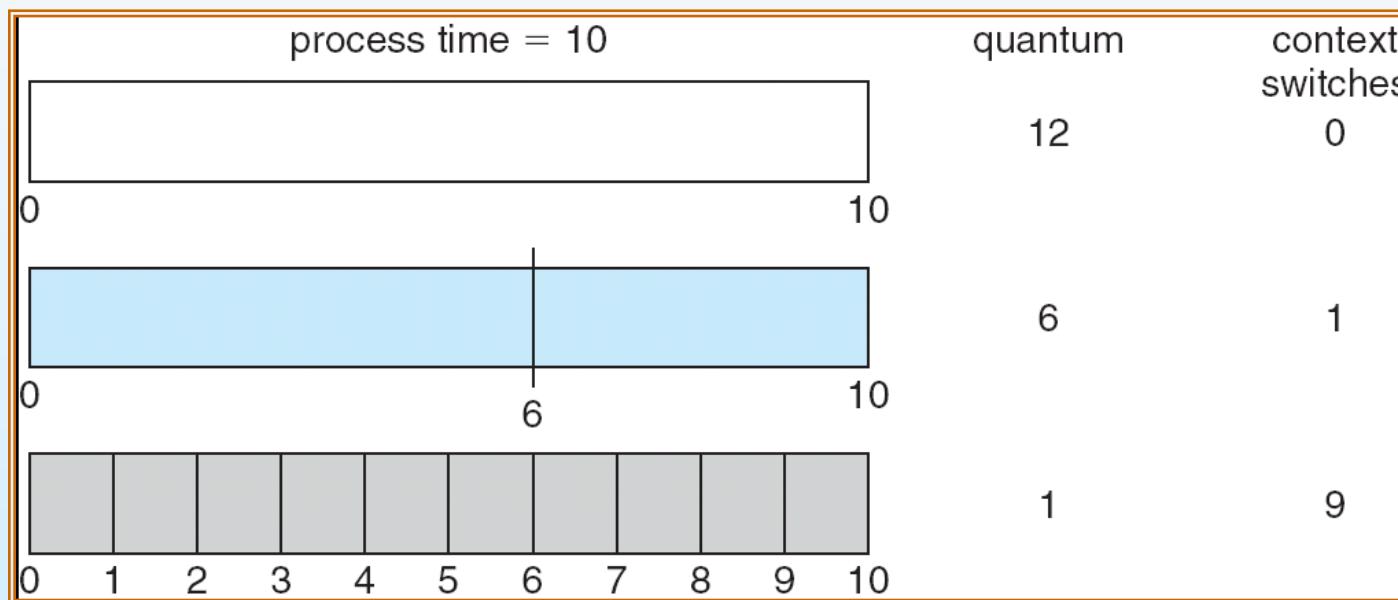


- Typically, higher average turnaround than SJF, but better *response*



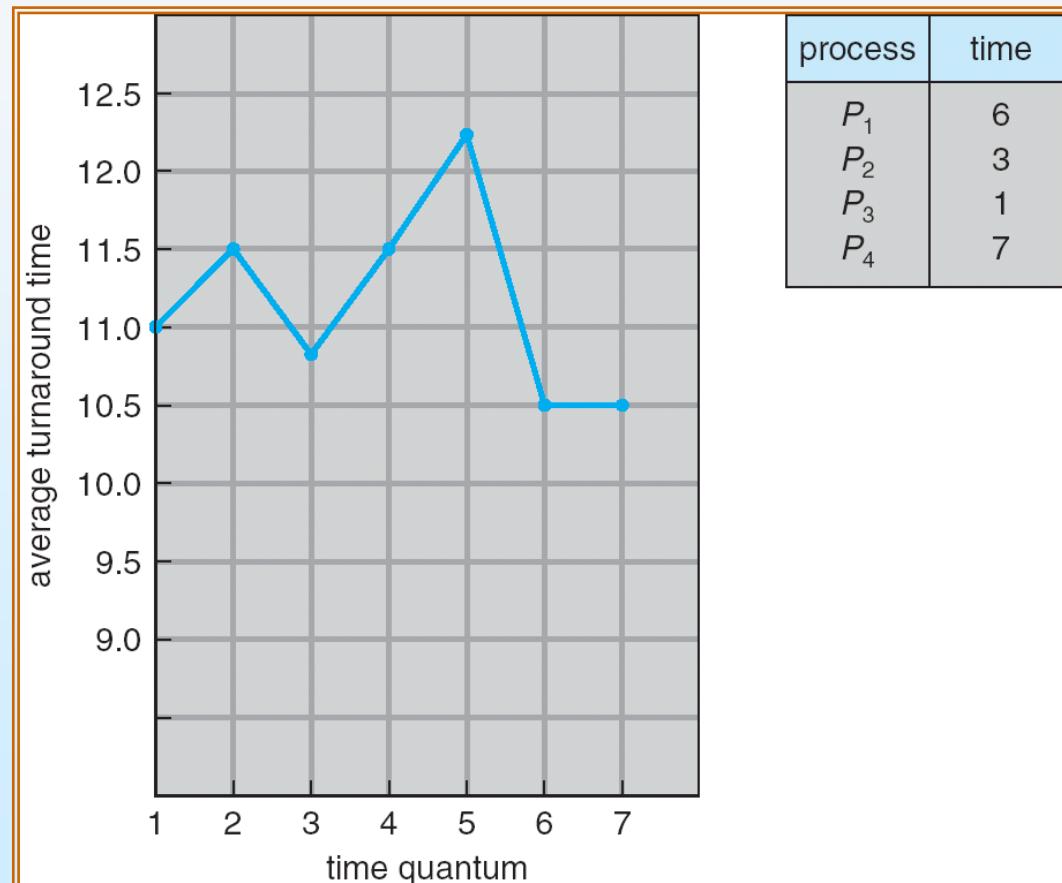


Time Quantum and Context Switch Time





Turnaround Time Varies With The Time Quantum





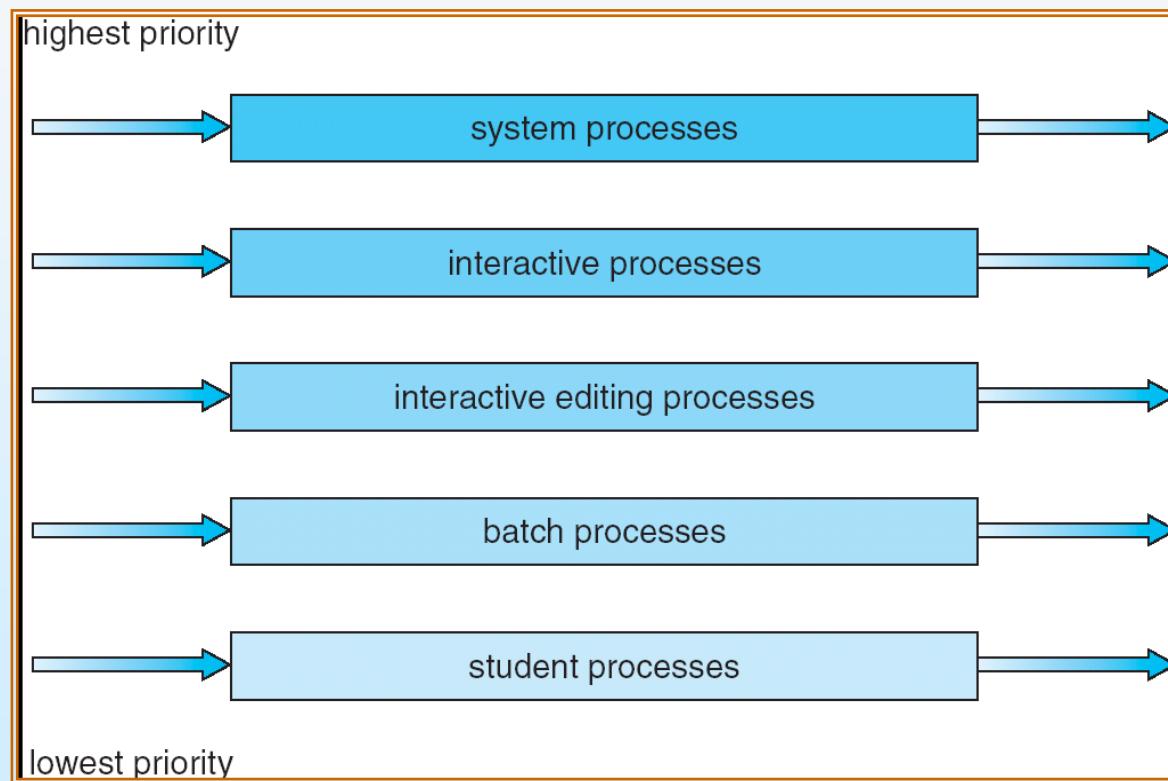
Multilevel Queue

- Ready queue is partitioned into separate queues:
foreground (interactive)
background (batch)
- Each queue has its own scheduling algorithm
 - foreground – RR
 - background – FCFS
- Scheduling must be done between the queues
 - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
 - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
 - 20% to background in FCFS





Multilevel Queue Scheduling





Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
 - number of queues
 - scheduling algorithms for each queue
 - method used to determine when to upgrade a process
 - method used to determine when to demote a process
 - method used to determine which queue a process will enter when that process needs service





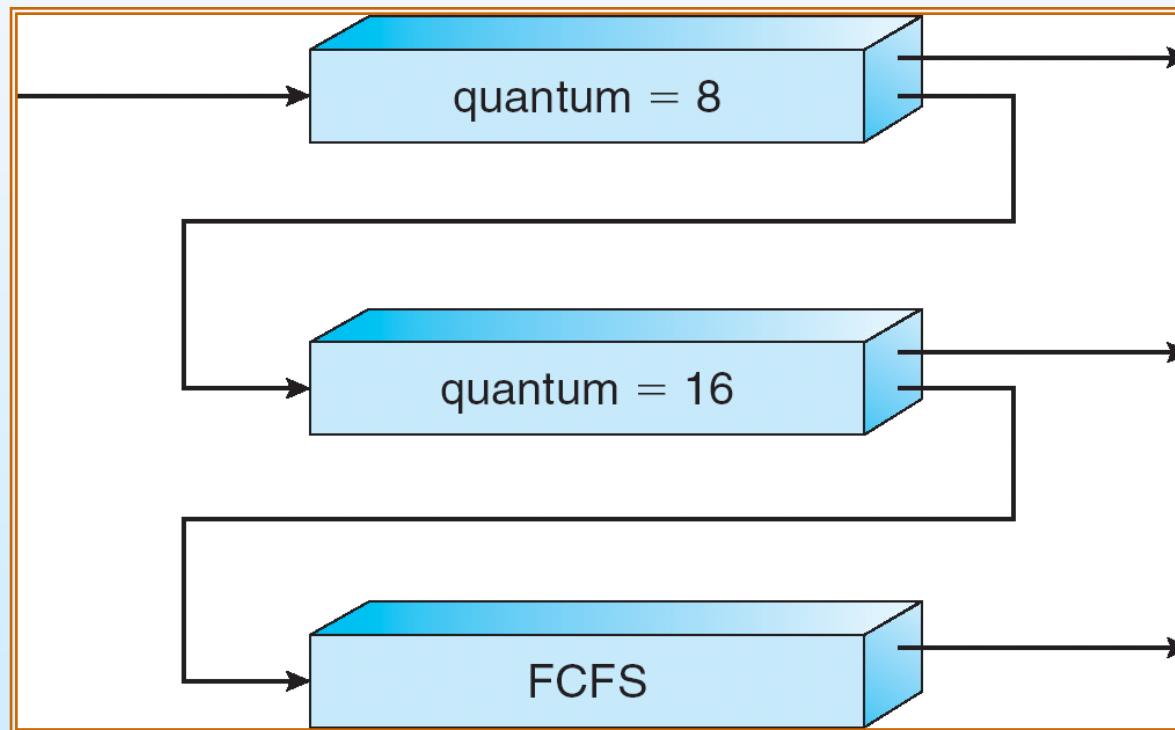
Example of Multilevel Feedback Queue

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





Multilevel Feedback Queues





Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- *Homogeneous processors* within a multiprocessor
- *Load sharing*
- *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing





Real-Time Scheduling

- *Hard real-time* systems – required to complete a critical task within a guaranteed amount of time
- *Soft real-time* computing – requires that critical processes receive priority over less fortunate ones





Thread Scheduling

- Local Scheduling – How the threads library decides which thread to put onto an available LWP
- Global Scheduling – How the kernel decides which kernel thread to run next





Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
    int i;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* set the scheduling policy - FIFO, RT, or OTHER */
    pthread_attr_setschedpolicy(&attr, SCHED_OTHER);
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);
```





Pthread Scheduling API

```
/* 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)
{
    printf("I am a thread\n");
    pthread exit(0);
}
```





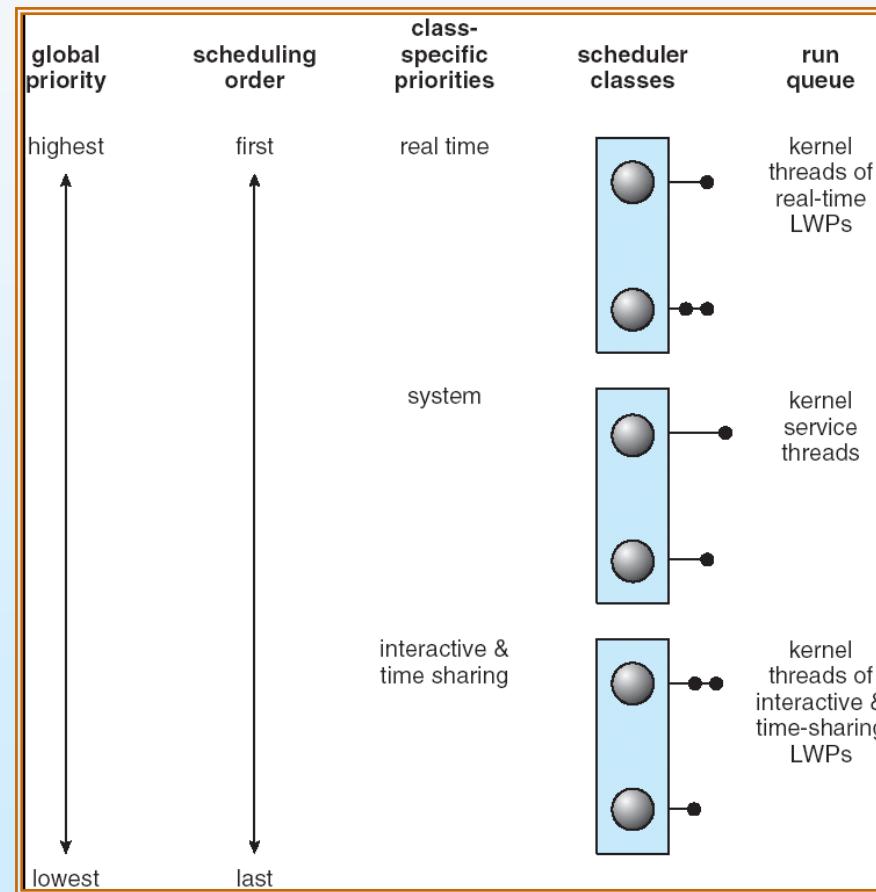
Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling





Solaris 2 Scheduling





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





Windows XP 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





Linux Scheduling

- Two algorithms: time-sharing and real-time
- Time-sharing
 - Prioritized credit-based – process with most credits is scheduled next
 - Credit subtracted when timer interrupt occurs
 - When credit = 0, another process chosen
 - When all processes have credit = 0, recrediting occurs
 - ▶ Based on factors including priority and history
- Real-time
 - Soft real-time
 - Posix.1b compliant – two classes
 - ▶ FCFS and RR
 - ▶ Highest priority process always runs first





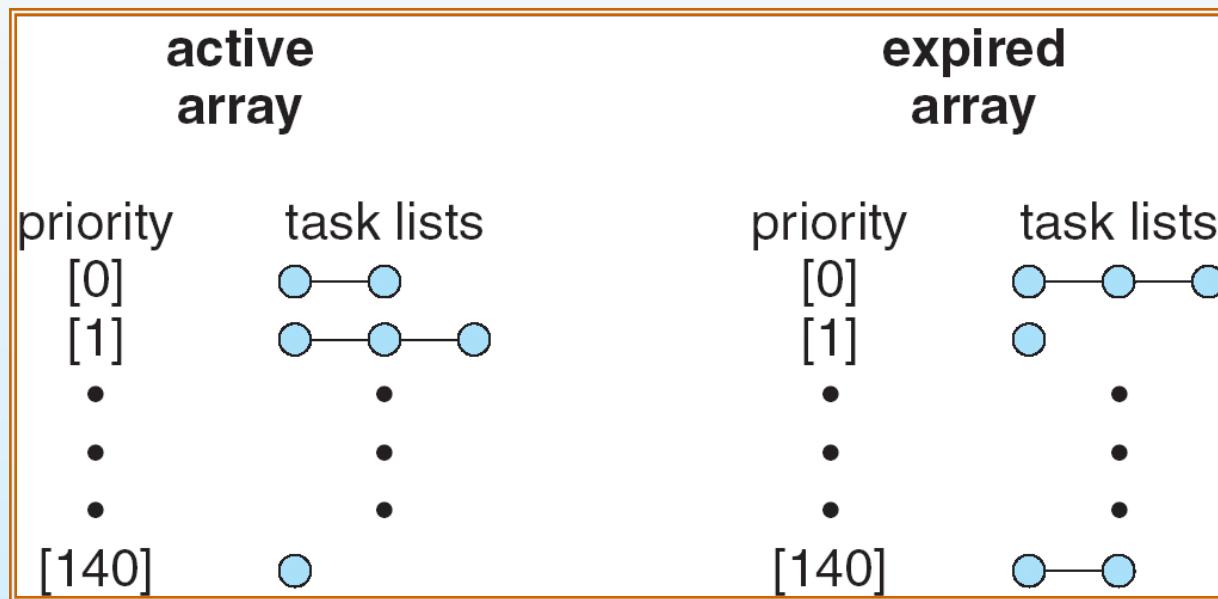
The Relationship Between Priorities and Time-slice length

numeric priority	relative priority	time quantum
0	highest	200 ms
•		
•		
•		
99		
100		
•		
•		
•		
140	lowest	10 ms





List of Tasks Indexed According to Priorities



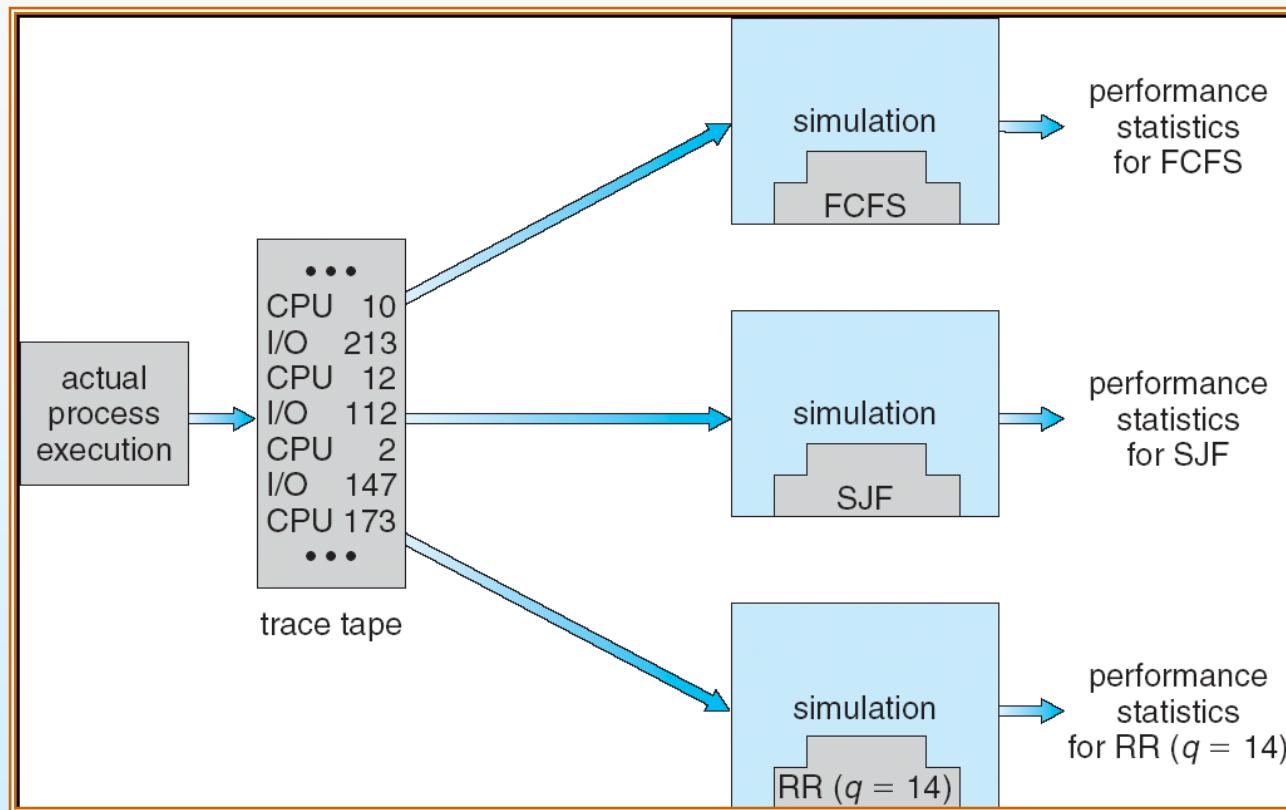


Algorithm Evaluation

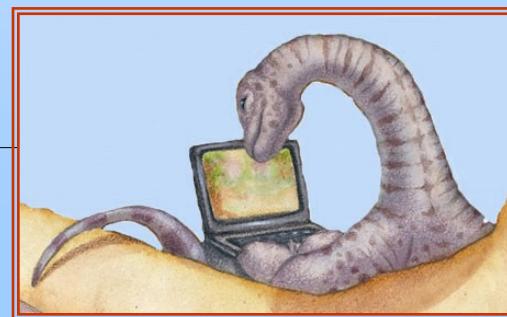
- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Queueing models
- Implementation



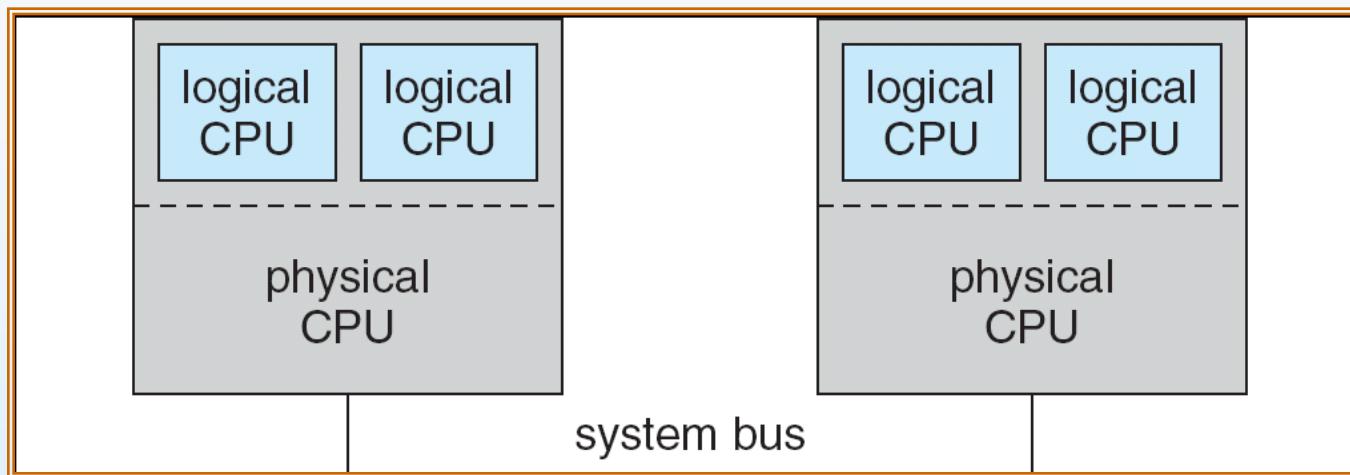
5.15



End of Chapter 5

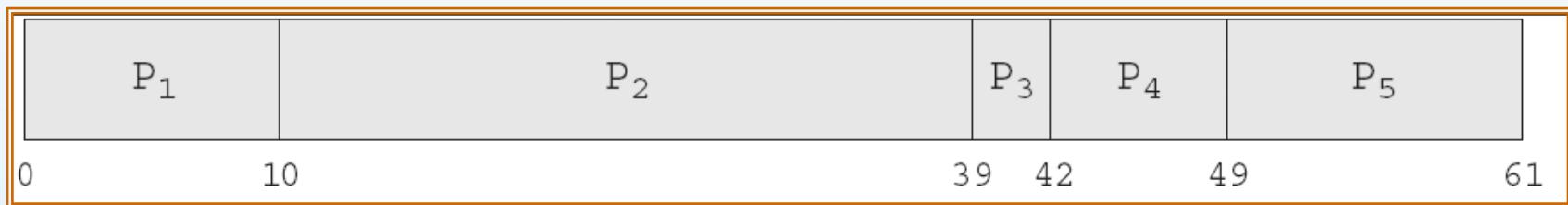


5.08



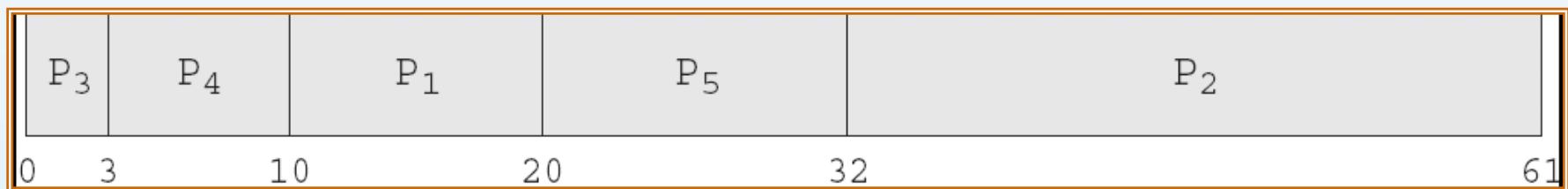


In-5.7



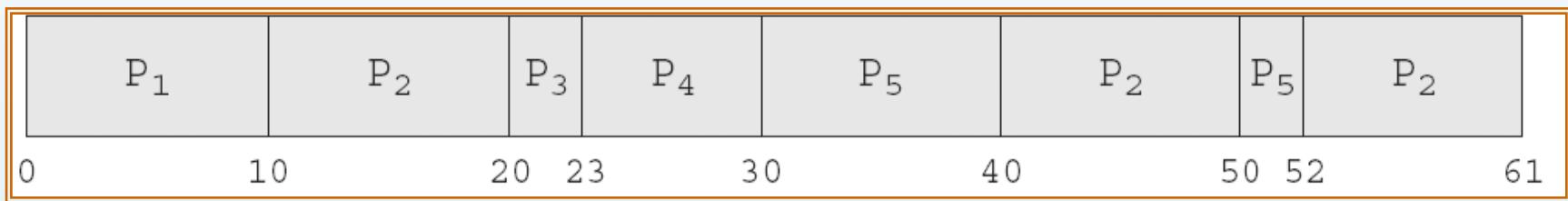


In-5.8

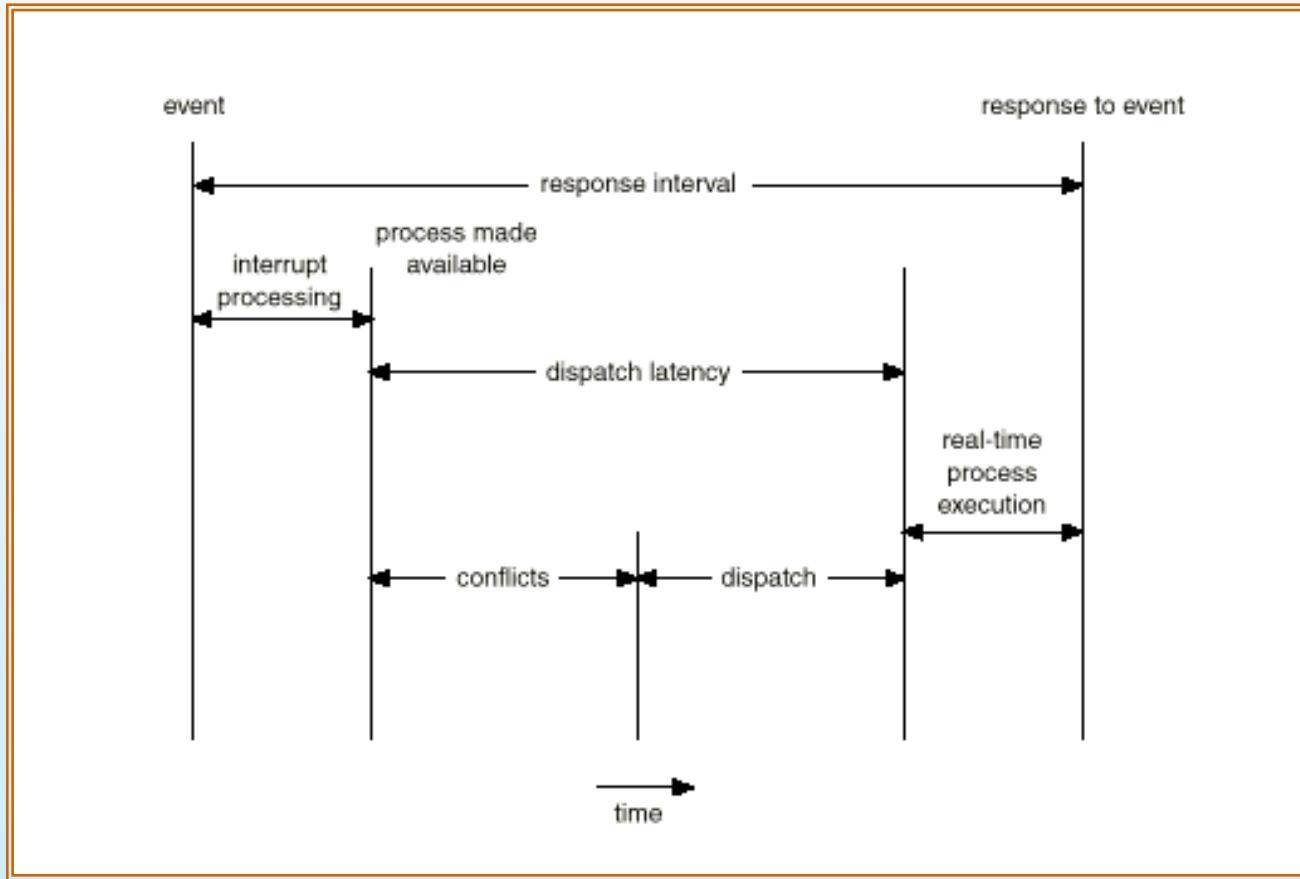




In-5.9



Dispatch Latency





Java Thread Scheduling

- JVM Uses a Preemptive, Priority-Based Scheduling Algorithm
- FIFO Queue is Used if There Are Multiple Threads With the Same Priority





Java Thread Scheduling (cont)

JVM Schedules a Thread to Run When:

1. The Currently Running Thread Exits the Runnable State
2. A Higher Priority Thread Enters the Runnable State

* Note – the JVM Does Not Specify Whether Threads are Time-Sliced or Not





Time-Slicing

Since the JVM Doesn't Ensure Time-Slicing, the yield() Method May Be Used:

```
while (true) {  
    // perform CPU-intensive task  
    . . .  
    Thread.yield();  
}
```

This Yields Control to Another Thread of Equal Priority





Thread Priorities

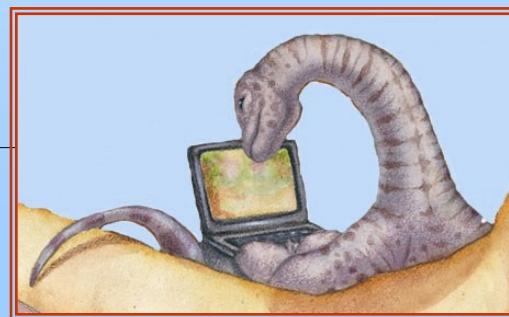
<u>Priority</u>	<u>Comment</u>
Thread.MIN_PRIORITY	Minimum Thread Priority
Thread.MAX_PRIORITY	Maximum Thread Priority
Thread.NORM_PRIORITY	Default Thread Priority

Priorities May Be Set Using `setPriority()` method:

```
setPriority(Thread.NORM_PRIORITY + 2);
```



Chapter 6: Process Synchronization





Module 6: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions





Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- 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 **count** that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.





Producer

```
while (true)
```

```
    /* produce an item and put in nextProduced
       while (count == BUFFER_SIZE)
           ; // do nothing
       buffer [in] = nextProduced;
       in = (in + 1) % BUFFER_SIZE;
       count++;
    }
```





Consumer

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





Race Condition

- `count++` could be implemented as

```
register1 = count  
register1 = register1 + 1  
count = register1
```

- `count--` could be implemented as

```
register2 = count  
register2 = register2 - 1  
count = register2
```

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

```
S0: producer execute register1 = count {register1 = 5}  
S1: producer execute register1 = register1 + 1 {register1 = 6}  
S2: consumer execute register2 = count {register2 = 5}  
S3: consumer execute register2 = register2 - 1 {register2 = 4}  
S4: producer execute count = register1 {count = 6 }  
S5: consumer execute count = register2 {count = 4}
```





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 processes 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





Peterson's Solution

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





Algorithm for Process P_i

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while ( flag[j] && turn == j);
```

CRITICAL SECTION

```
    flag[i] = FALSE;
```

REMAINDER SECTION

```
} while (TRUE);
```





Synchronization Hardware

- Many systems provide hardware support for 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
- Modern machines provide special atomic hardware instructions
 - ▶ **Atomic = non-interruptable**
 - Either test memory word and set value
 - Or swap contents of two memory words





TestAndndSet Instruction

- Definition:

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





Solution using TestAndSet

- Shared boolean variable lock., initialized to false.

- Solution:

```
do {  
    while ( TestAndSet (&lock ) )  
        ; /* do nothing  
  
        // critical section  
  
    lock = FALSE;  
  
    // remainder section  
  
} while ( TRUE);
```





Swap Instruction

□ Definition:

```
void Swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```





Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key.
- Solution:

```
do {  
    key = TRUE;  
    while ( key == TRUE)  
        Swap (&lock, &key );  
  
        // critical section  
  
    lock = FALSE;  
  
    // remainder section  
  
} while ( TRUE);
```





Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore S – integer variable
- Two standard operations modify S : `wait()` and `signal()`
 - Originally called `P()` and `V()`
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
 - `wait (S) {`
 `while S <= 0`
 `; // no-op`
 `S--;`
 `}`
 - `signal (S) {`
 `S++;`
 `}`





Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
 - Also known as mutex locks
- Can implement a counting semaphore **S** as a binary semaphore
- Provides mutual exclusion
 - Semaphore S; // initialized to 1
 - wait (S);
 Critical Section
 signal (S);





Semaphore Implementation

- Must guarantee that no two processes can execute `wait ()` and `signal ()` on the same semaphore at the same time
- Thus, 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.





Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
wait (S){  
    value--;  
    if (value < 0) {  
        add this process to waiting queue  
        block(); }  
}
```

- Implementation of signal:

```
Signal (S){  
    value++;  
    if (value <= 0) {  
        remove a process P from the waiting queue  
        wakeup(P); }  
}
```





Deadlock and Starvation

- **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

wait (S);
wait (Q);

.

signal (S);
signal (Q);

P_1

wait (Q);
wait (S);

.

signal (Q);
signal (S);

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.





Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem





Bounded-Buffer Problem

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





Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {  
  
    // produce an item  
  
    wait (empty);  
    wait (mutex);  
  
    // add the item to the buffer  
  
    signal (mutex);  
    signal (full);  
} while (true);
```





Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {  
    wait (full);  
    wait (mutex);  
  
    // remove an item from buffer  
  
    signal (mutex);  
    signal (empty);  
  
    // consume the removed item  
  
} while (true);
```





Readers-Writers Problem

- A data set is shared among a number of concurrent processes
 - Readers – only read the data set; they do **not** perform any updates
 - Writers – can both read and write.
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
- Shared Data
 - Data set
 - Semaphore **mutex** initialized to 1.
 - Semaphore **wrt** initialized to 1.
 - Integer **readcount** initialized to 0.





Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {  
    wait (wrt) ;  
  
    // writing is performed  
  
    signal (wrt) ;  
} while (true)
```





Readers-Writers Problem (Cont.)

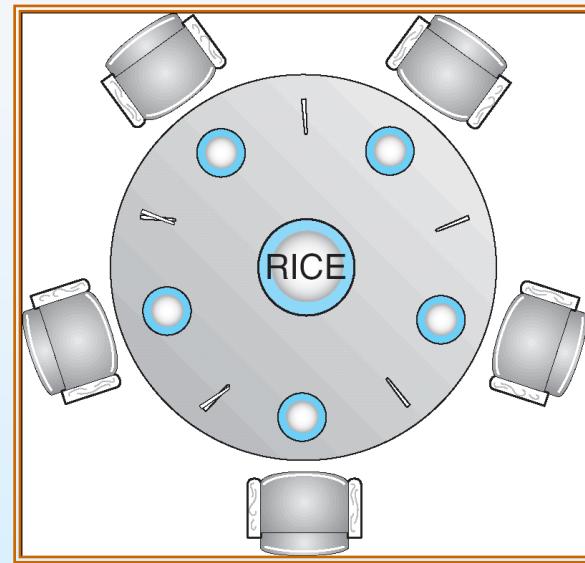
- The structure of a reader process

```
do {  
    wait (mutex) ;  
    readcount ++ ;  
    if (readcount == 1) wait (wrt) ;  
    signal (mutex)  
  
    // reading is performed  
  
    wait (mutex) ;  
    readcount - - ;  
    if redacount == 0) signal (wrt) ;  
    signal (mutex) ;  
} while (true)
```





Dining-Philosophers Problem



- Shared data
 - Bowl of rice (data set)
 - Semaphore **chopstick [5]** initialized to 1





Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

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





Problems with Semaphores

- Correct use of semaphore operations:
 - signal (mutex) wait (mutex)
 - wait (mutex) ... wait (mutex)
- Omitting of wait (mutex) or signal (mutex) (or both)





Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

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

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

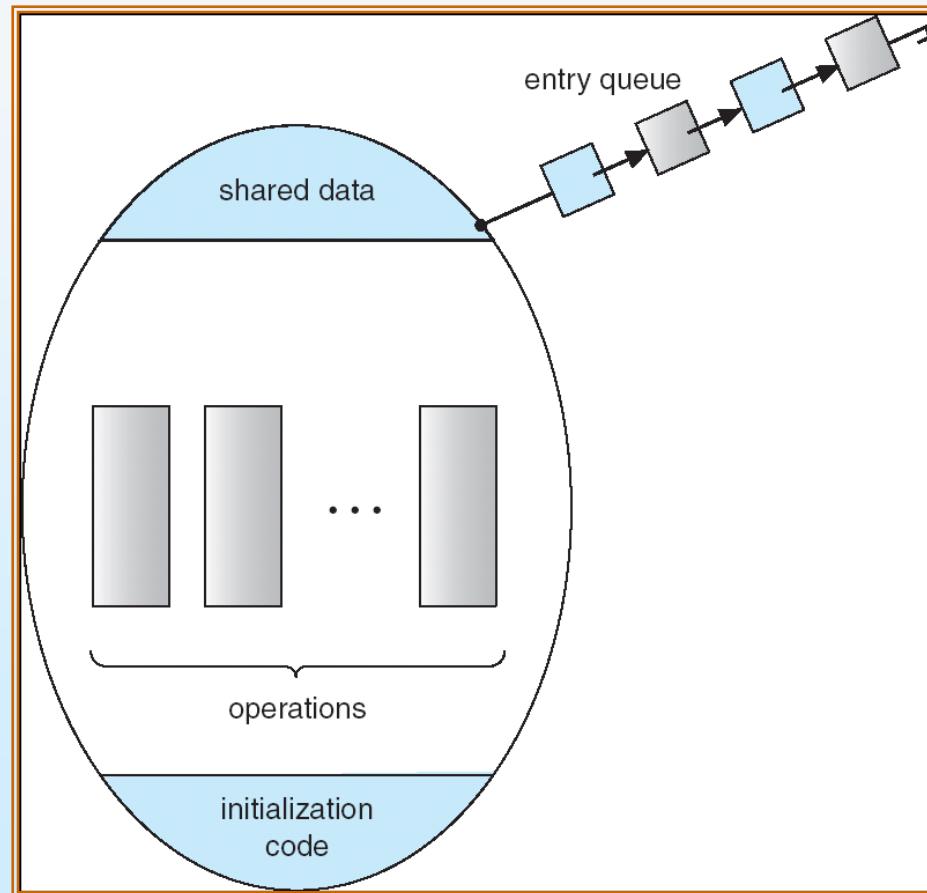
    Initialization code ( ....) { ... }

    ...
}
```





Schematic view of a Monitor





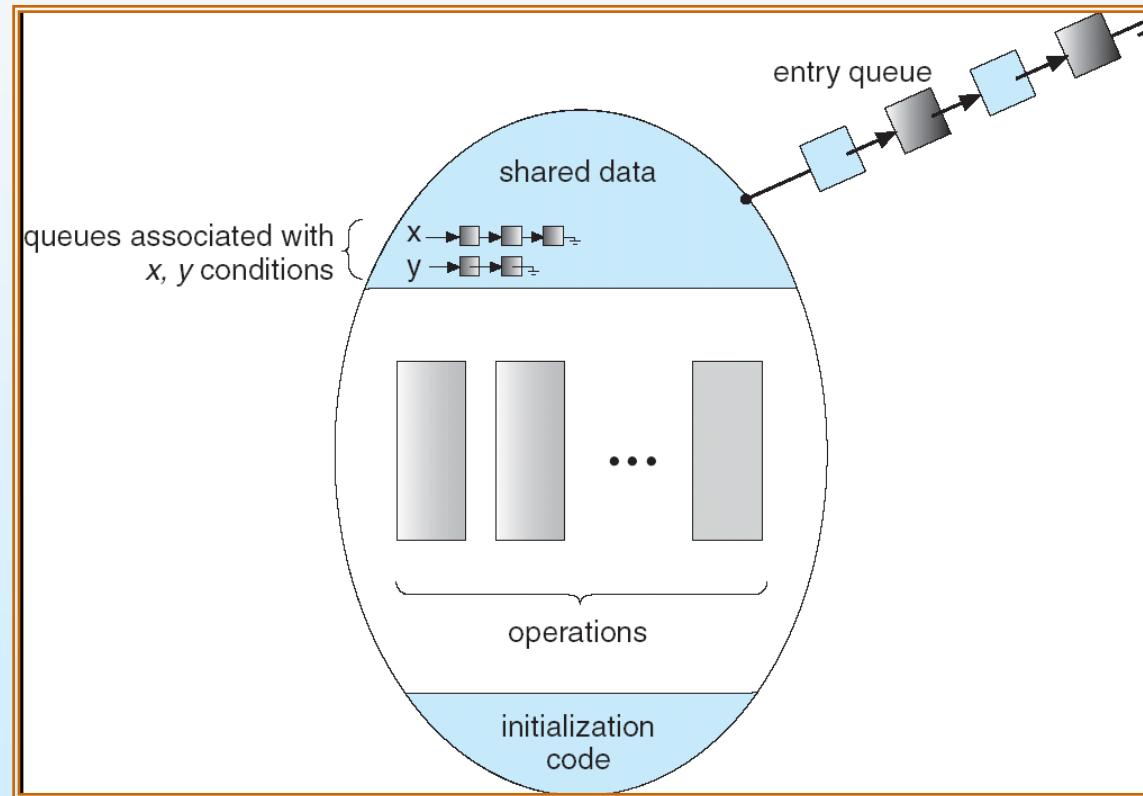
Condition Variables

- condition x, y;
- Two operations on a condition variable:
 - `x.wait ()` – a process that invokes the operation is suspended.
 - `x.signal ()` – resumes one of processes (if any) that invoked `x.wait ()`





Monitor with Condition Variables





Solution to Dining Philosophers

```
monitor DP
{
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];

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

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





Solution to Dining Philosophers (cont)

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

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





Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads





Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments
- Uses **condition variables** and **readers-writers** locks when longer sections of code need access to data
- Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock





Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
- Also provides **dispatcher objects** which may act as either mutexes and semaphores
- Dispatcher objects may also provide **events**
 - An event acts much like a condition variable





Linux Synchronization

- Linux:
 - disables interrupts to implement short critical sections
- Linux provides:
 - semaphores
 - spin locks





Pthreads Synchronization

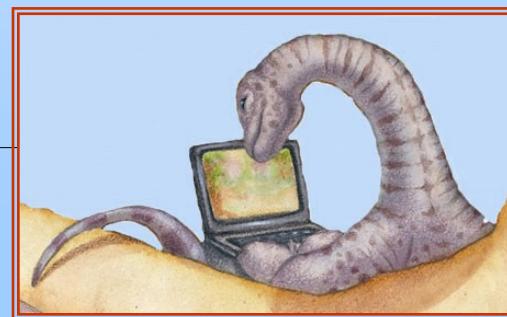
- Pthreads API is OS-independent
- It provides:
 - mutex locks
 - condition variables
- Non-portable extensions include:
 - read-write locks
 - spin locks



End of Chapter 6



Chapter 7: Deadlocks





Chapter 7: Deadlocks

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





Chapter Objectives

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system.





The Deadlock Problem

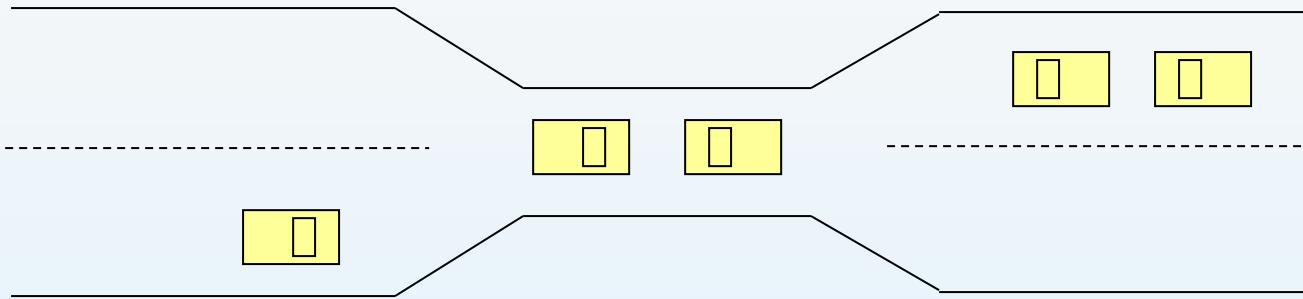
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
 - System has 2 tape drives.
 - P_1 and P_2 each hold one tape drive and each needs another one.
- Example
 - semaphores A and B , initialized to 1

P_0	P_1
<i>wait (A);</i>	<i>wait(B)</i>
<i>wait (B);</i>	<i>wait(A)</i>





Bridge Crossing Example



- Traffic only 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.





System Model

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





Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

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





Resource-Allocation Graph

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

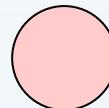
- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system.
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge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



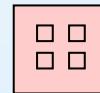


Resource-Allocation Graph (Cont.)

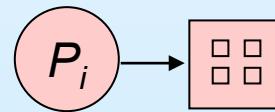
- Process



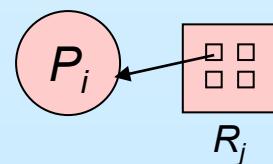
- Resource Type with 4 instances



- P_i requests instance of R_j

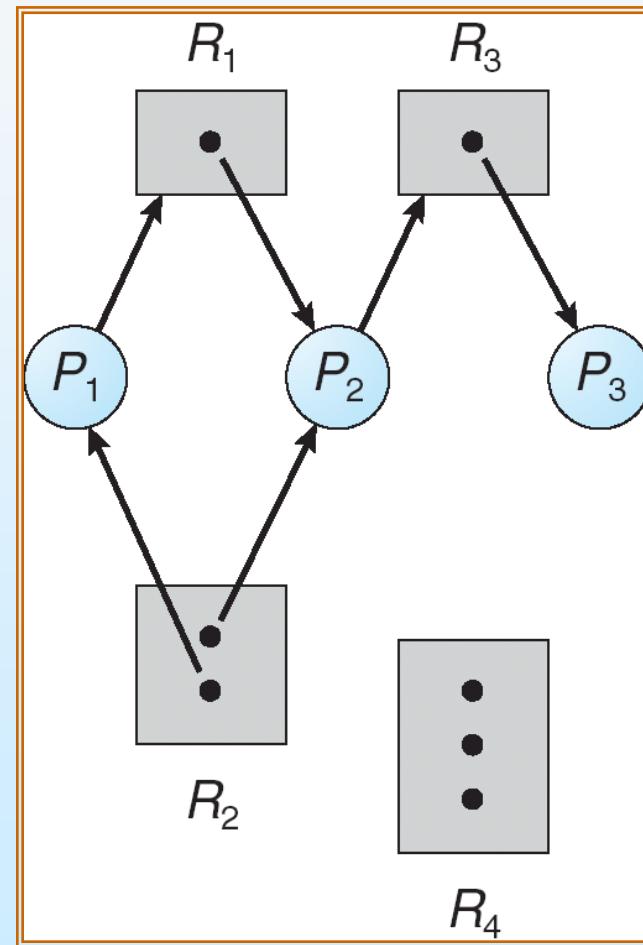


- P_i is holding an instance of R_j



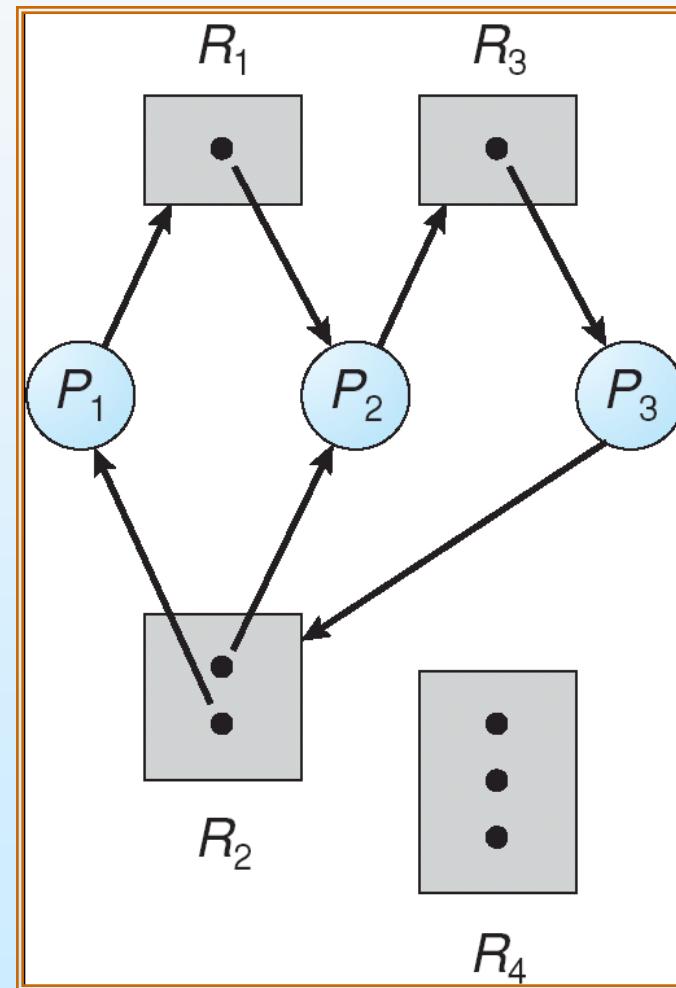


Example of a Resource Allocation Graph



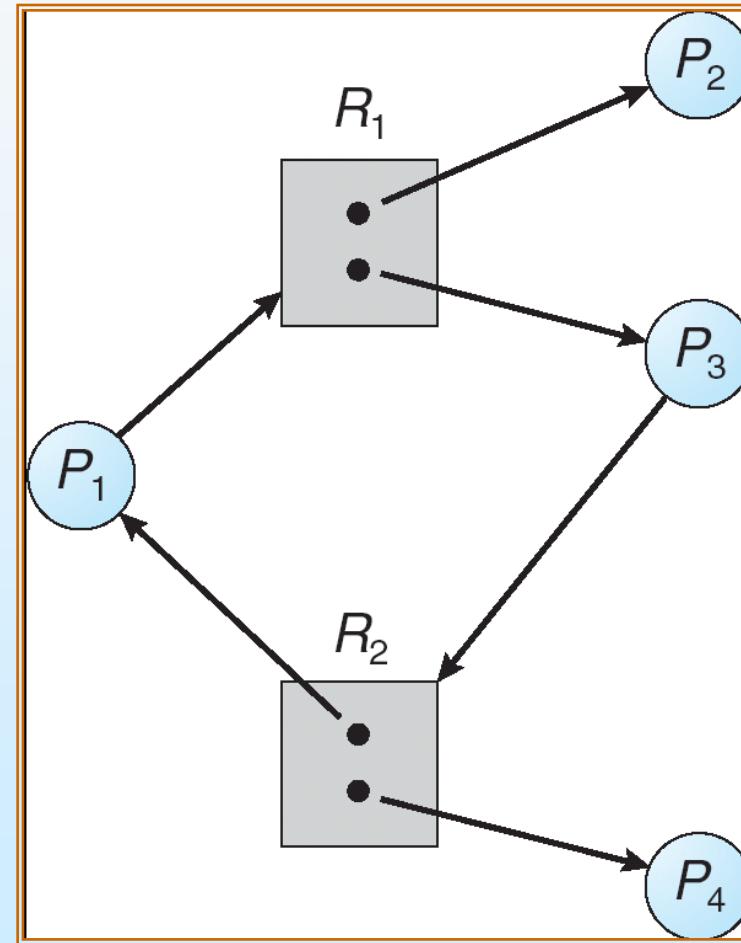


Resource Allocation Graph With A Deadlock





Resource Allocation Graph With A Cycle But No Deadlock





Basic Facts

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





Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.





Deadlock Prevention

Restrain the ways request can be made.

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable 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.
 - 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.





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 safe sequence of all processes.
- Sequence $\langle P_1, P_2, \dots, P_n \rangle$ is safe if 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 $j < i$.
 - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
 - When P_i 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.





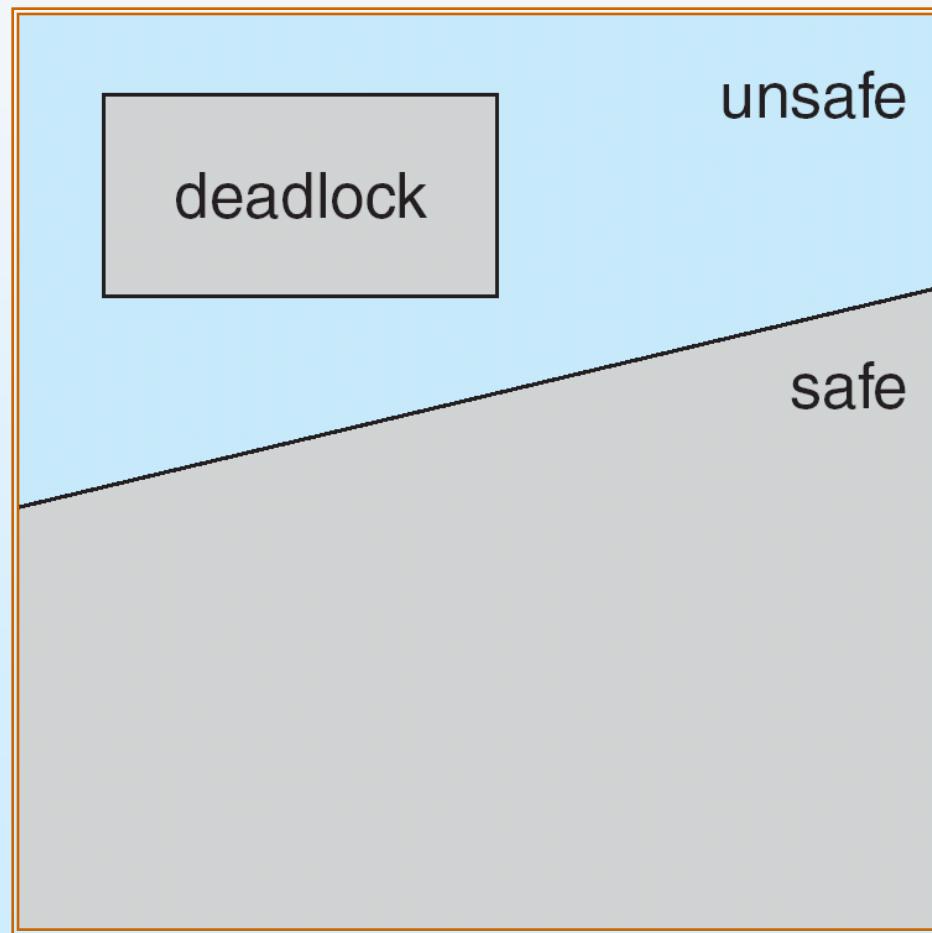
Basic Facts

- 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





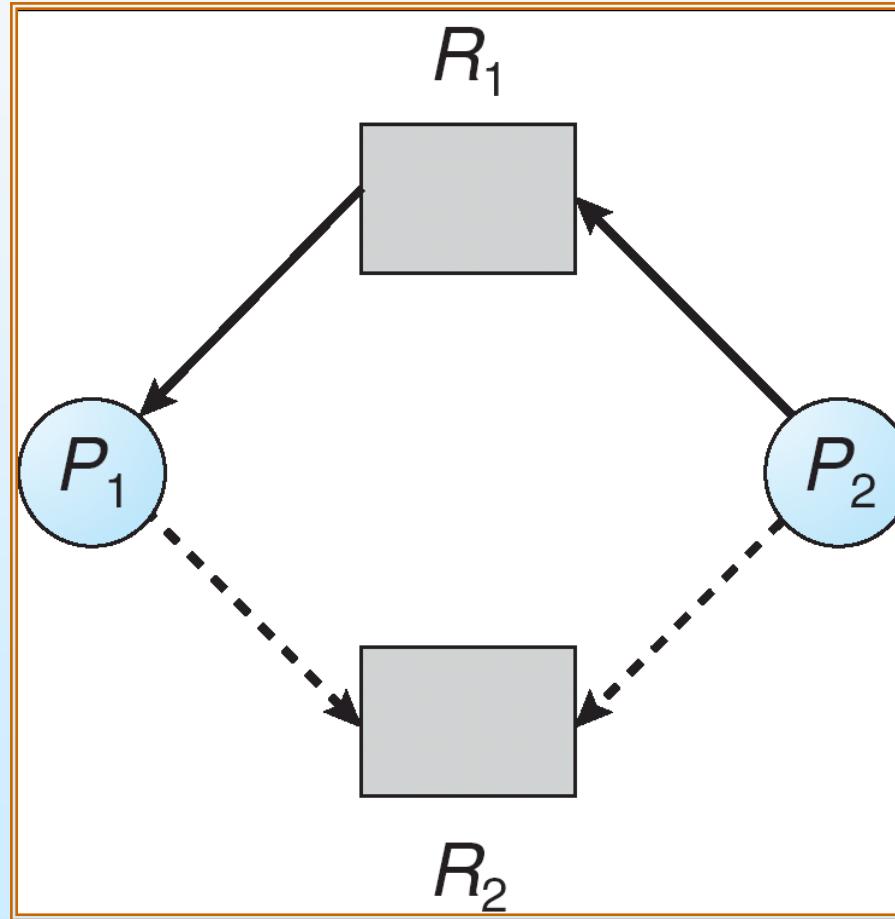
Resource-Allocation Graph Algorithm

- *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.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.



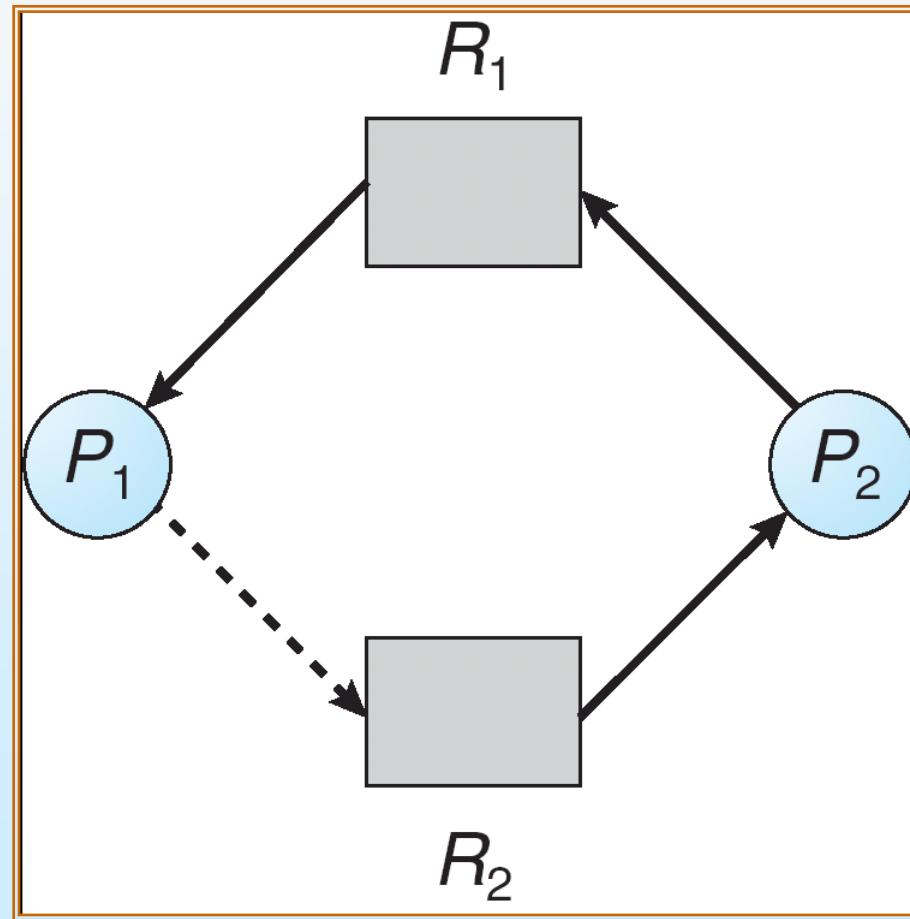


Resource-Allocation Graph For Deadlock Avoidance





Unsafe State In Resource-Allocation Graph





Banker's Algorithm

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





Data Structures for the Banker's Algorithm

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

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

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





Safety Algorithm

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

$Work = Available$

$Finish[i] = false$ for $i = 1, 2, \dots, n$.

2. Find and i such that both:

(a) $Finish[i] = false$

(b) $Need_i \leq Work$

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

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

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

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

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

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





Example of Banker's Algorithm

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

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





Example (Cont.)

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

Need

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) (Cont.)

- 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>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	1	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 $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ 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





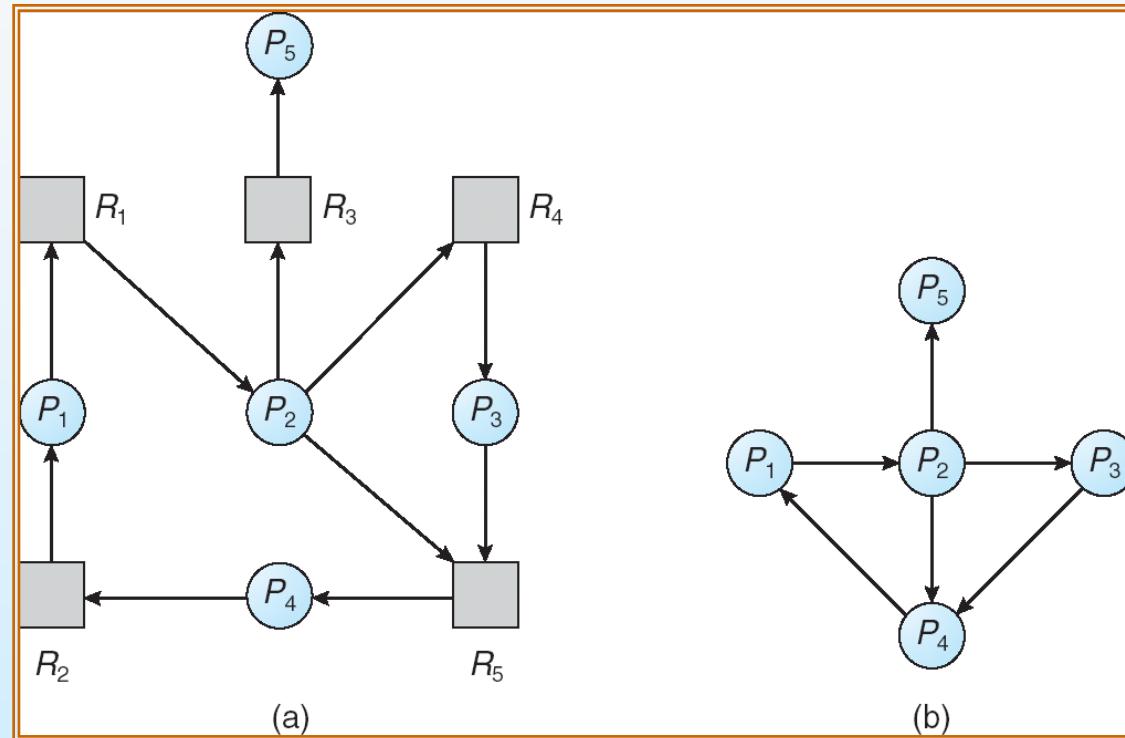
Single Instance of Each Resource Type

- 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.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.





Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph

Corresponding wait-for graph





Several Instances of a Resource Type

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





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 $Allocation_i \neq 0$, then
 $Finish[i] = \text{false}$; otherwise, $Finish[i] = \text{true}$.
2. Find an index i such that both:
 - (a) $Finish[i] == \text{false}$
 - (b) $Request_i \leq Work$

If no such i exists, go to step 4.





Detection Algorithm (Cont.)

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

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





Example of Detection Algorithm

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

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

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





Example (Cont.)

- P_2 requests an additional instance of type C.

Request

	A	B	C
P_0	0	0	0
P_1	2	0	1
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?
 - Priority of the process.
 - How long process has computed, and how much longer to completion.
 - Resources the process has used.
 - Resources process needs to complete.
 - How many processes will need to be terminated.
 - Is process interactive or batch?



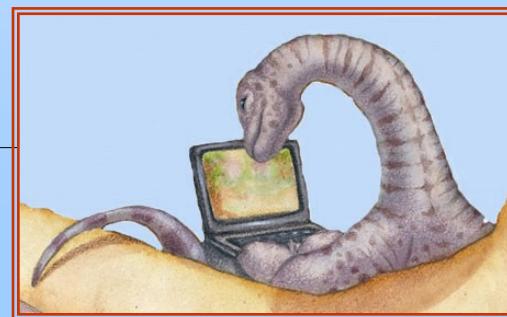


Recovery from Deadlock: Resource Preemption

- 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 7



Chapter 8: Memory Management





Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Paging
- Segmentation
- Segmentation with Paging





Background

- Program must be brought into memory and placed within a process for it to be run
- **Input queue** – collection of processes on the disk that are waiting to be brought into memory to run the program
- User programs go through several steps before being run





Binding of Instructions and Data to Memory

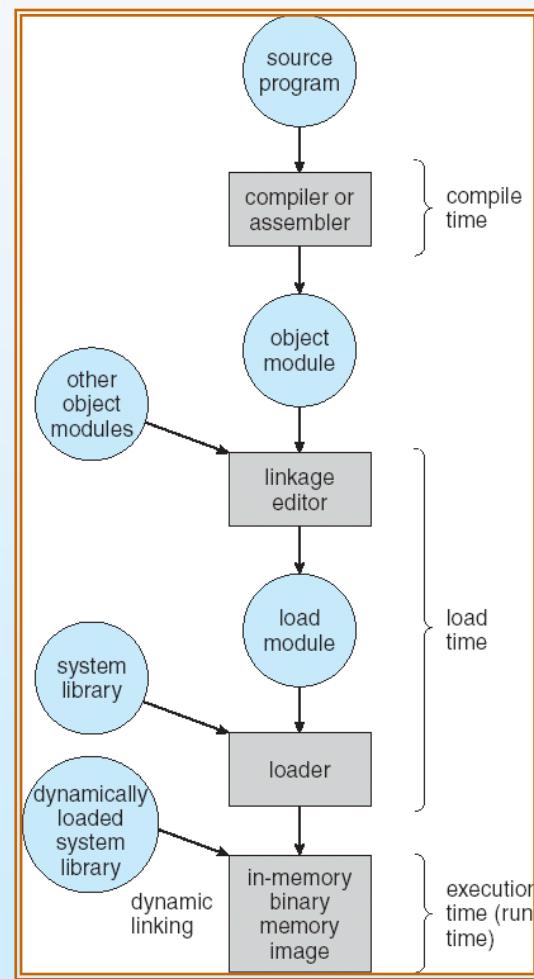
Address binding of instructions and data to memory addresses can happen at three different stages

- **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





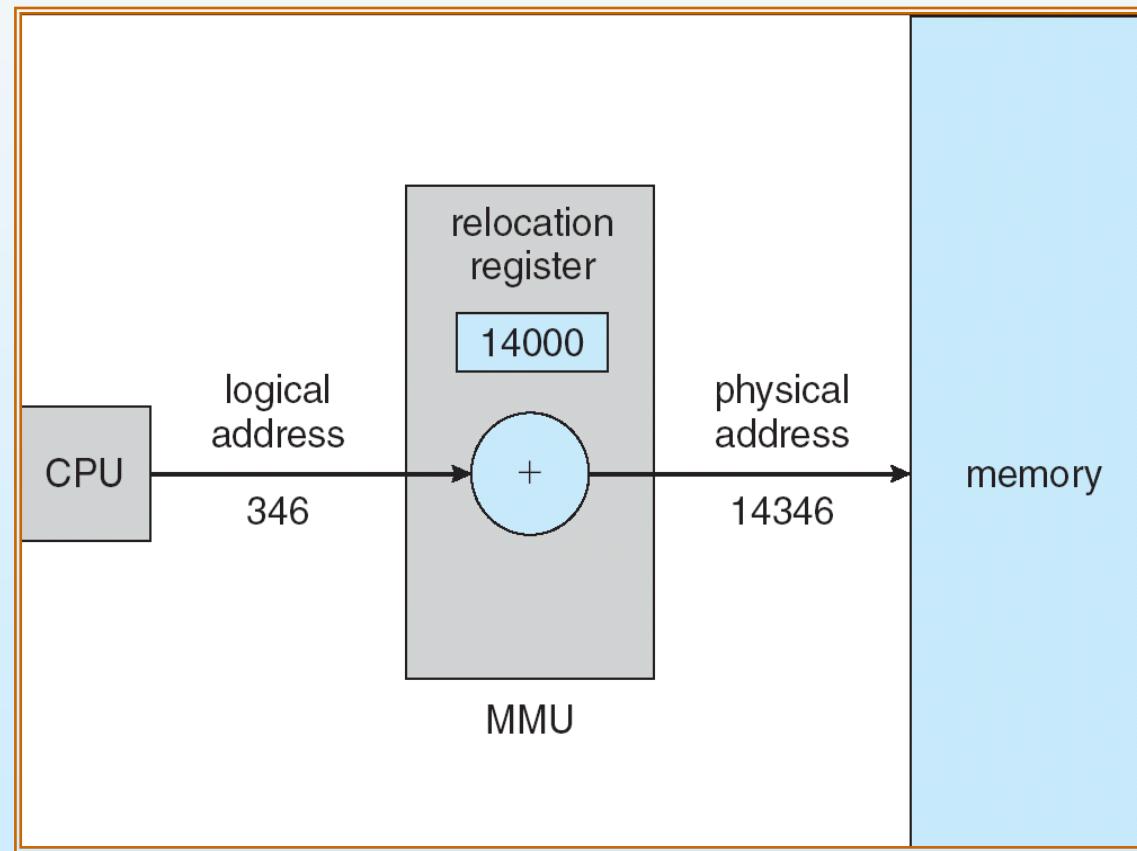
Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- In MMU scheme, 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





Dynamic relocation using a relocation register





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 operating system 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





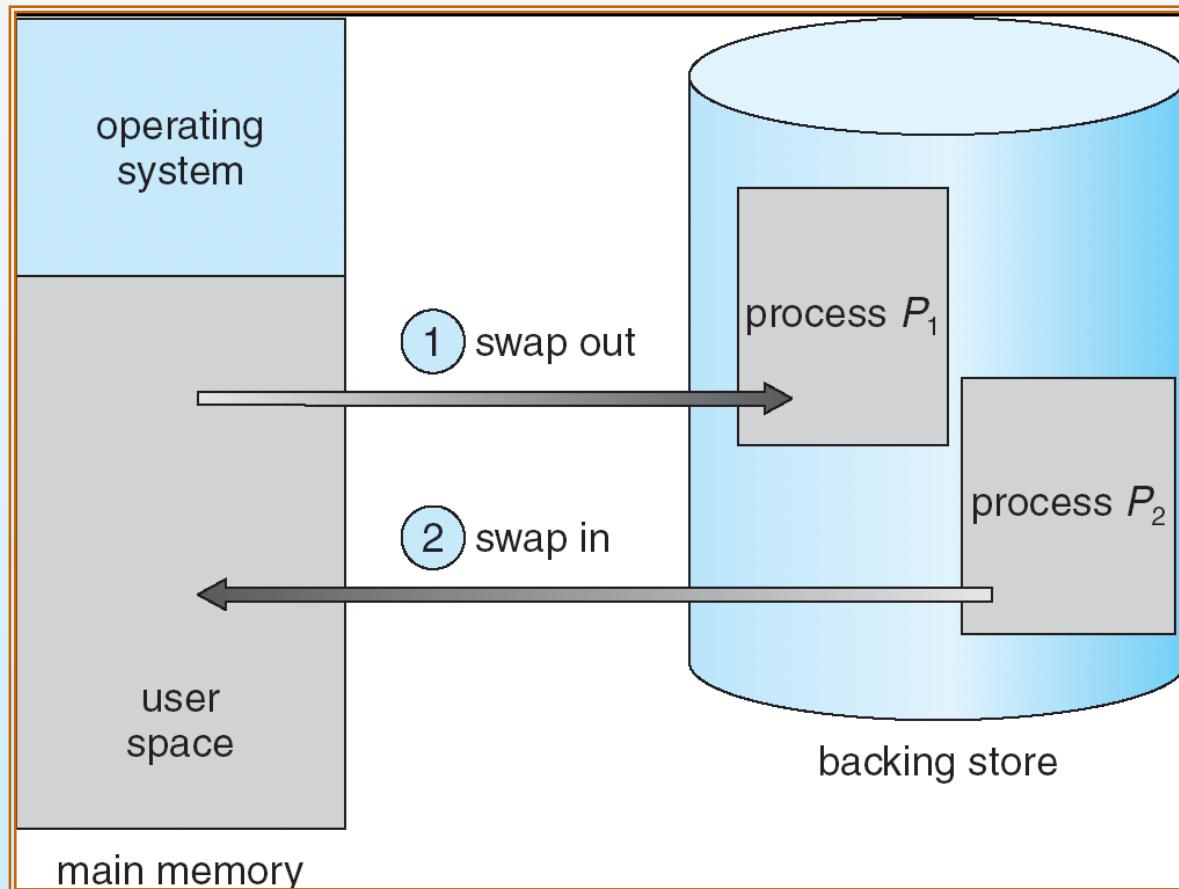
Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)





Schematic View of Swapping





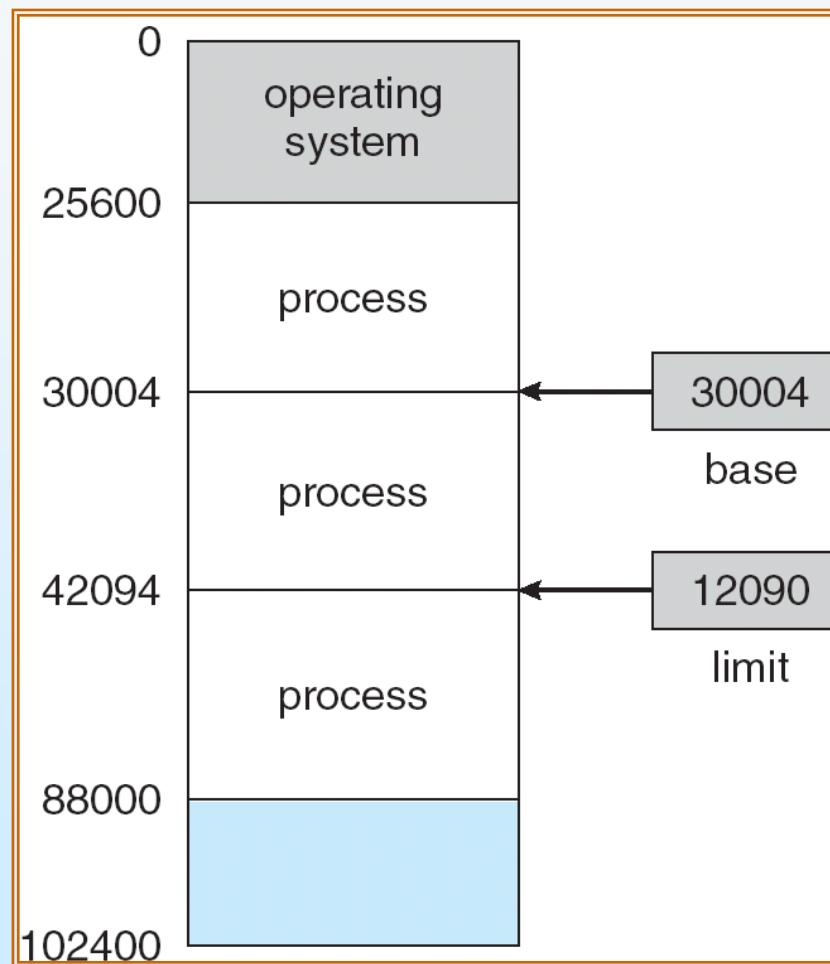
Contiguous Allocation

- Main memory usually into two partitions:
 - Resident operating system, usually held in low memory with interrupt vector
 - User processes then held in high memory
- Single-partition allocation
 - Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data
 - Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register



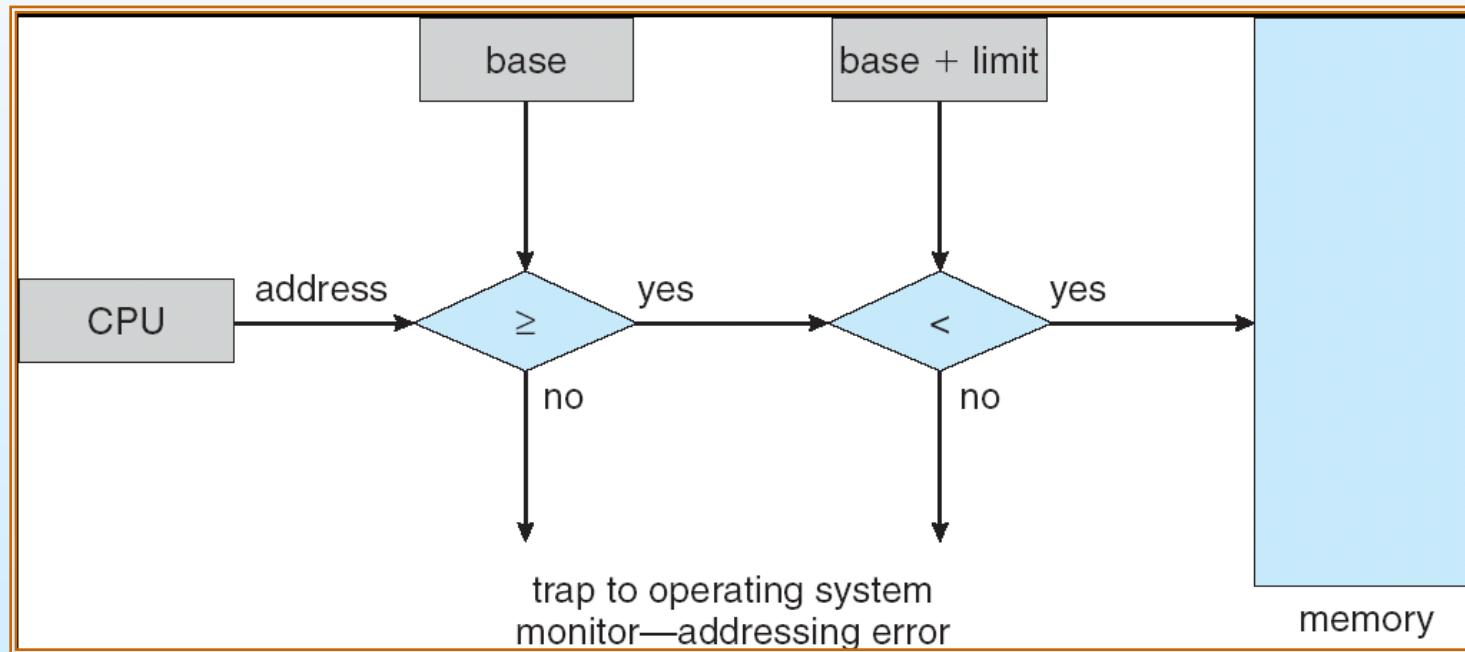


A base and a limit register define a logical address space





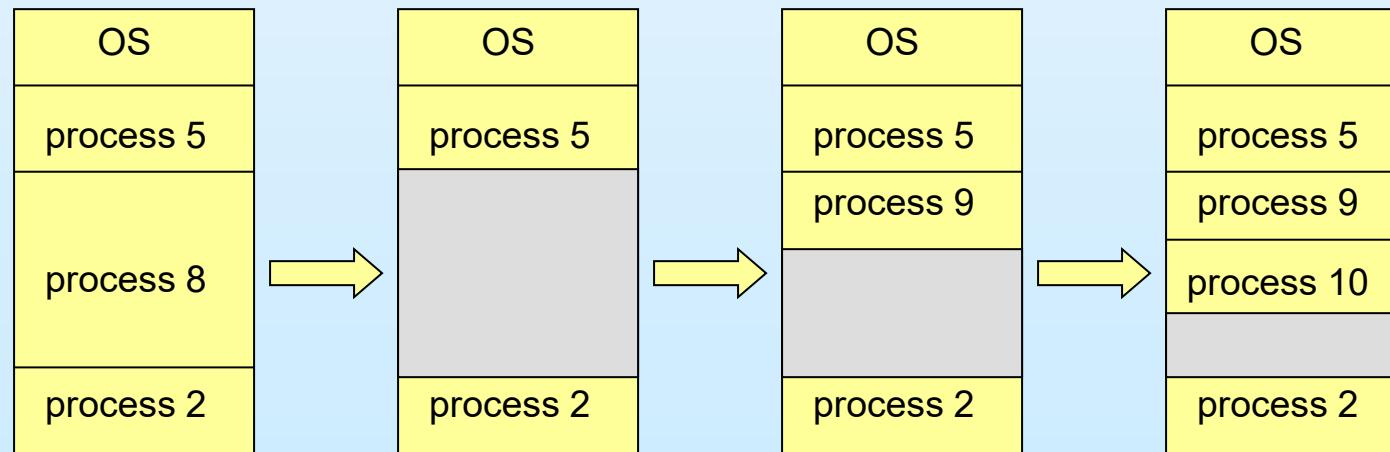
HW address protection with base and limit registers





Contiguous Allocation (Cont.)

- Multiple-partition allocation
 - *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
 - 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
- 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





Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes)
- 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
- Internal fragmentation





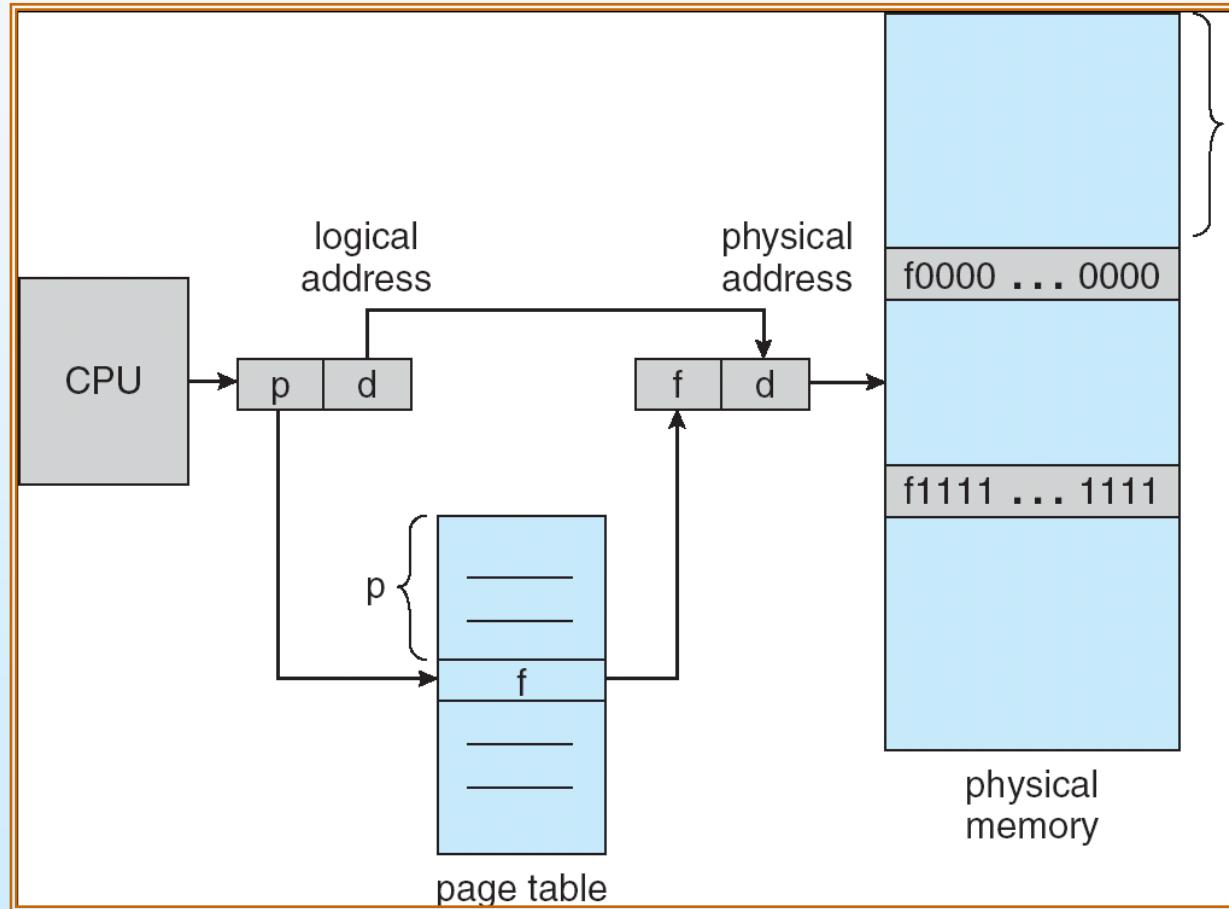
Address Translation Scheme

- 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



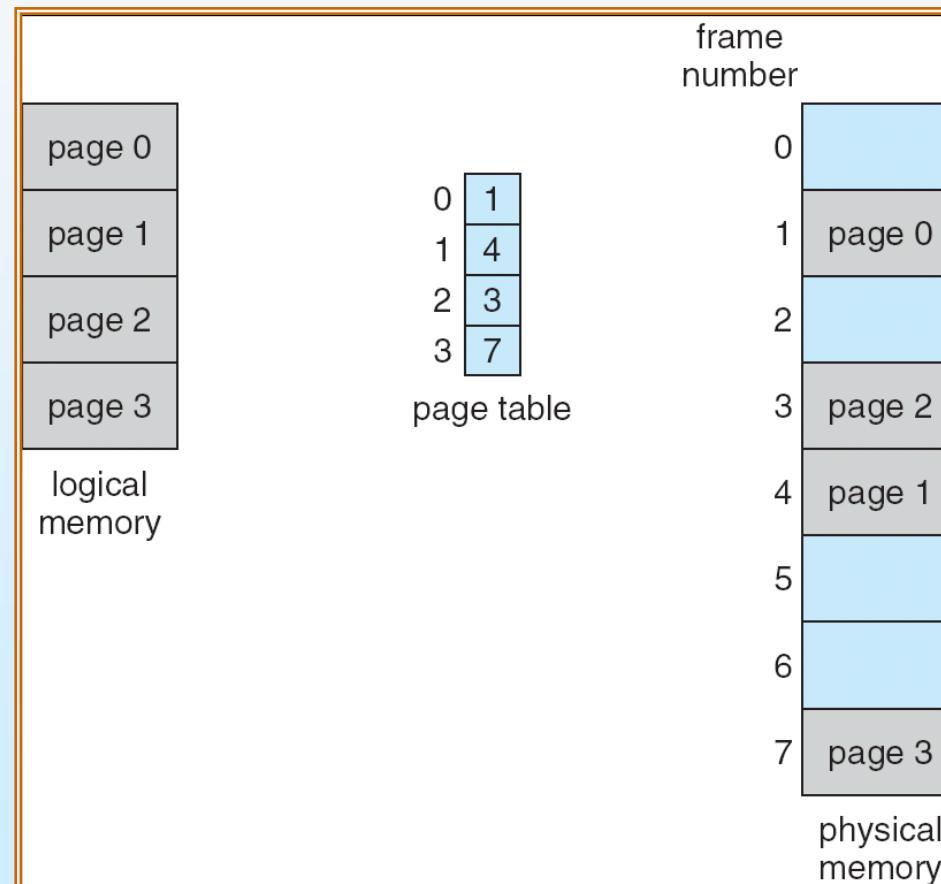


Address Translation Architecture



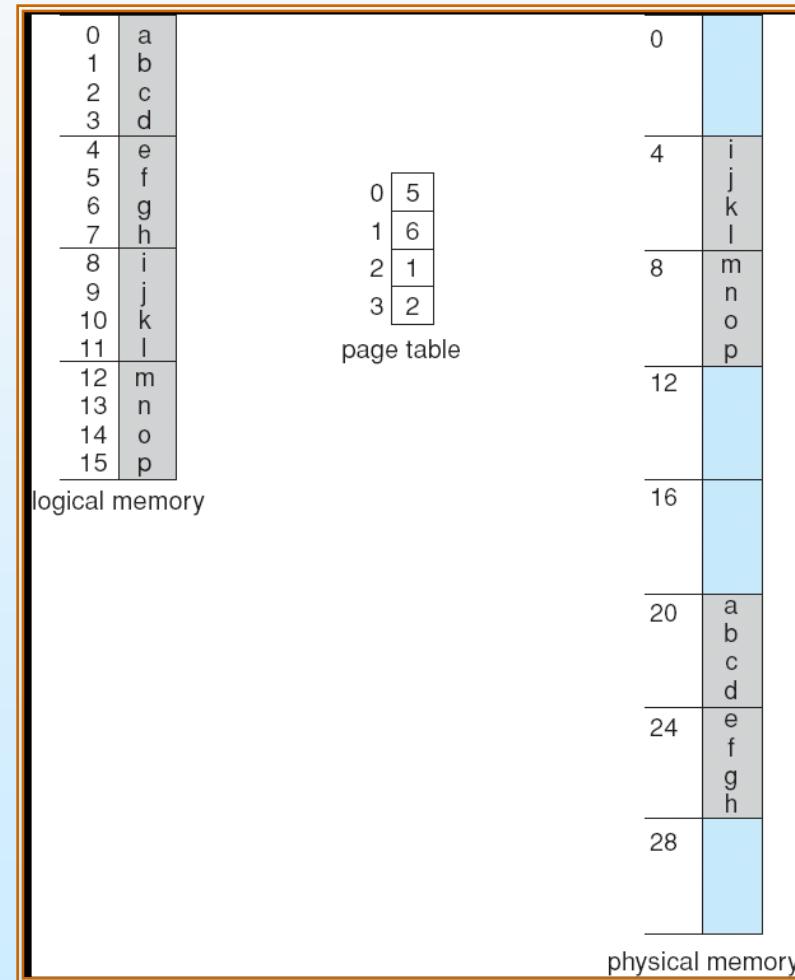


Paging Example

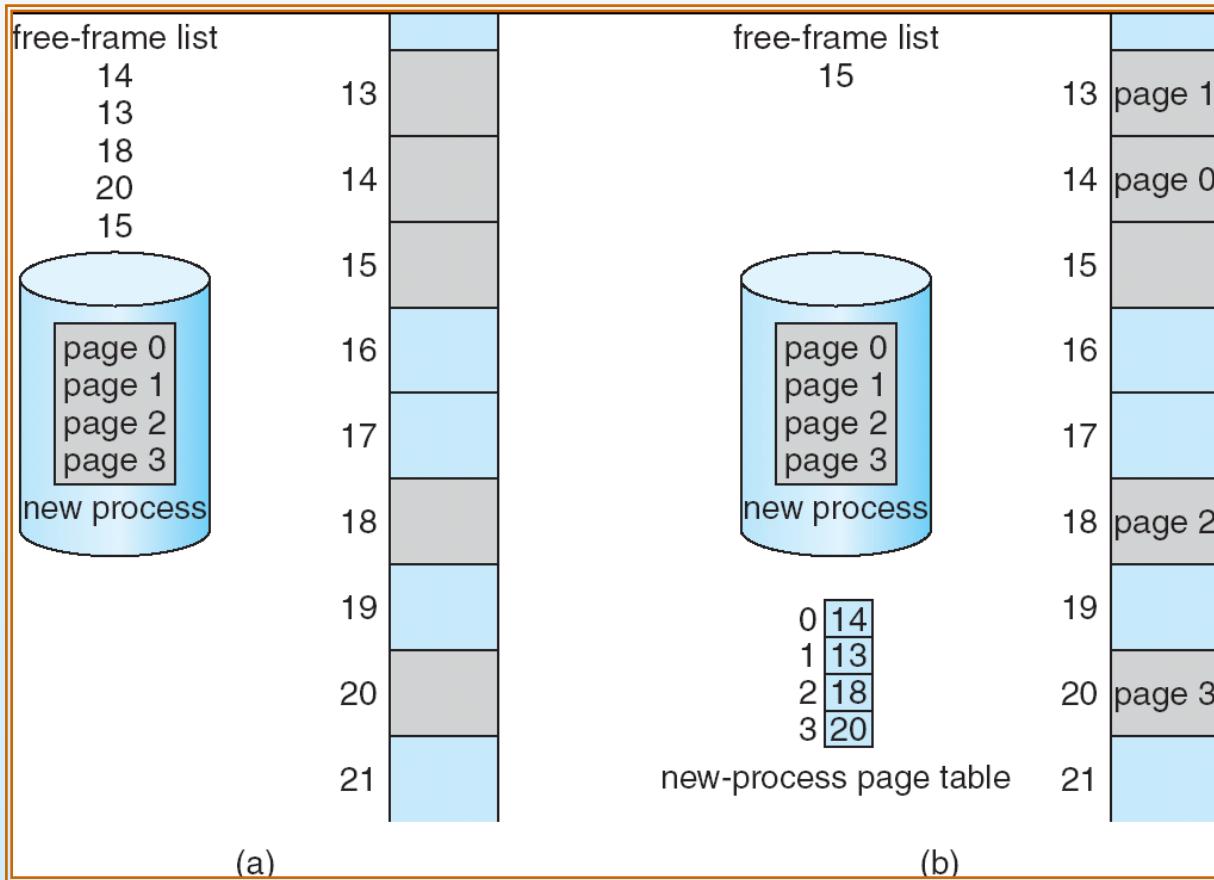




Paging Example



Free Frames





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* (PRLR) 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 **associative memory** or **translation look-aside buffers (TLBs)**





Associative Memory

- Associative memory – parallel search

Page #	Frame #

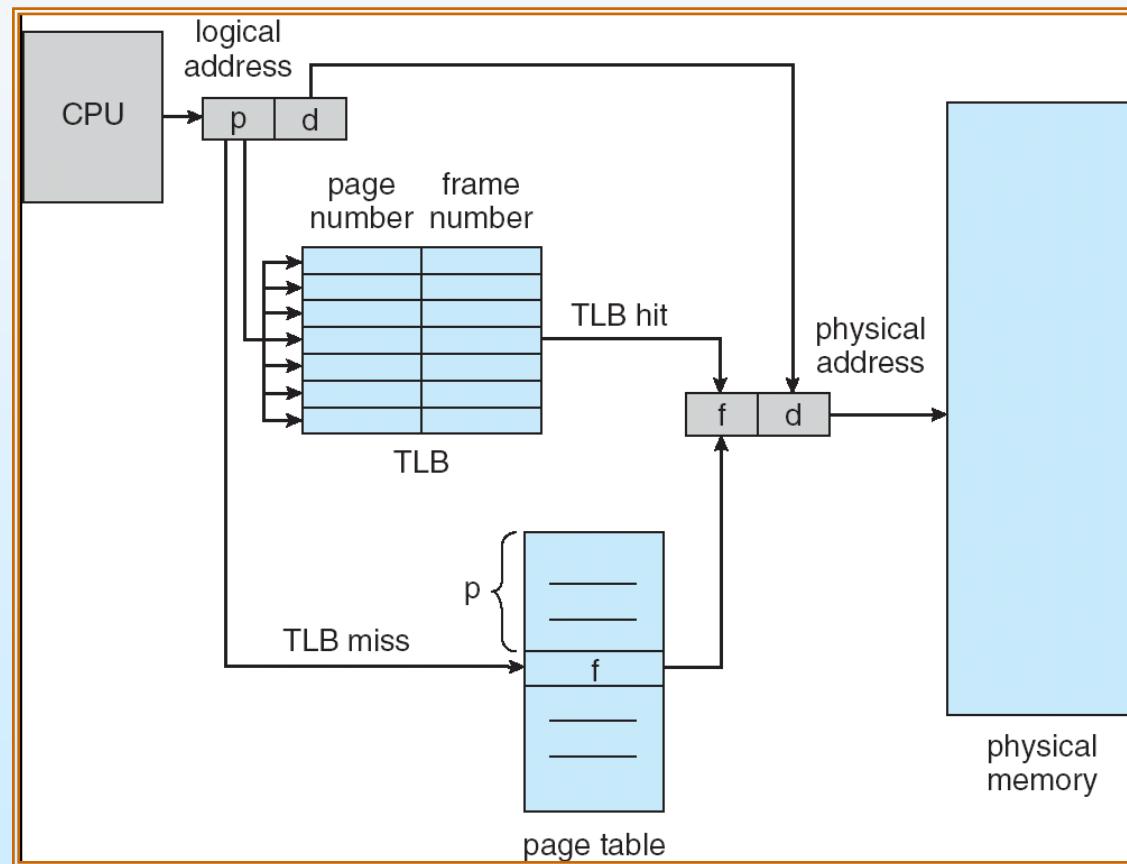
Address translation (A' , A'')

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





Paging Hardware With TLB





Effective Access Time

- Associative Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ration related to number of associative registers
- Hit ratio = α
- **Effective Access Time (EAT)**

$$\begin{aligned} \text{EAT} &= (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$





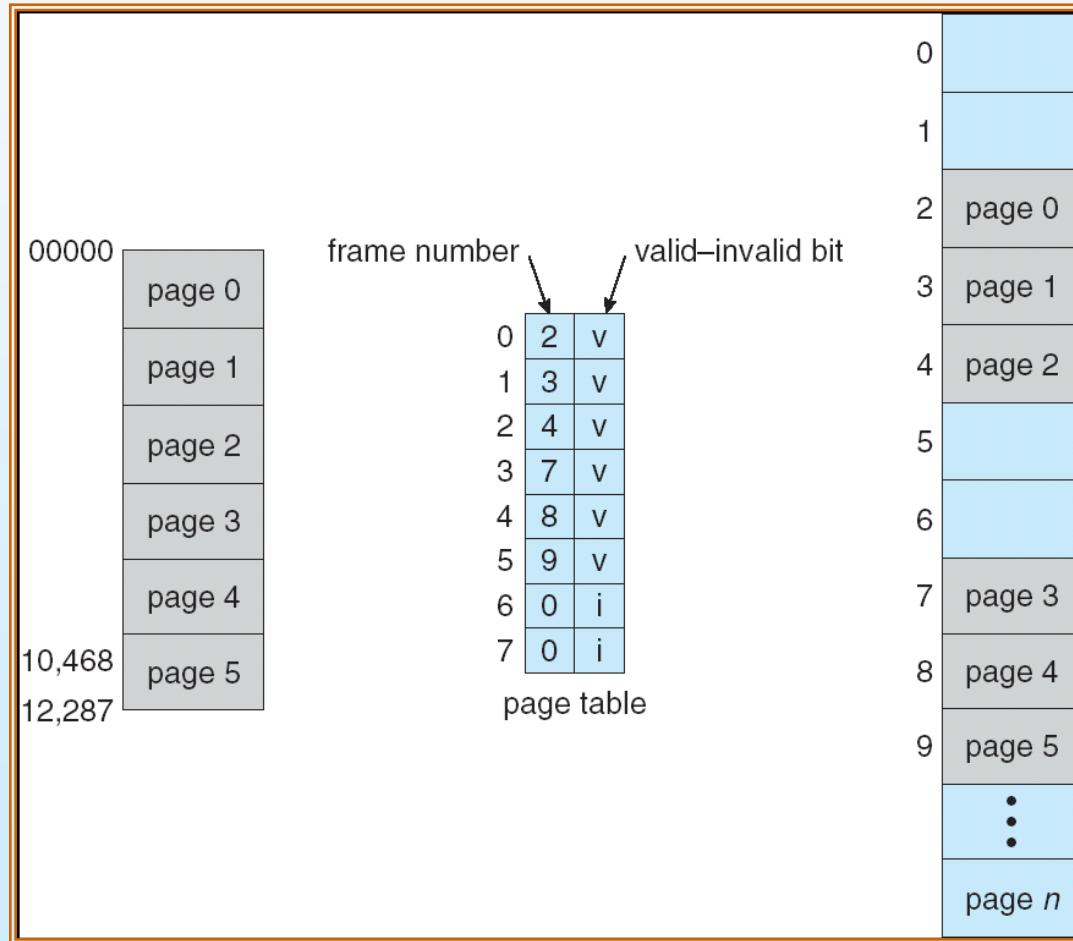
Memory Protection

- Memory protection implemented by associating protection bit with each frame
- **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





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





Page Table Structure

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables





Hierarchical Page Tables

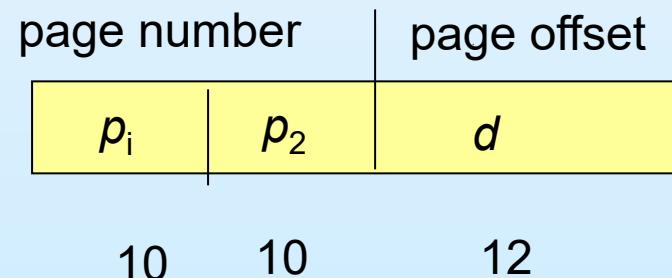
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table





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:

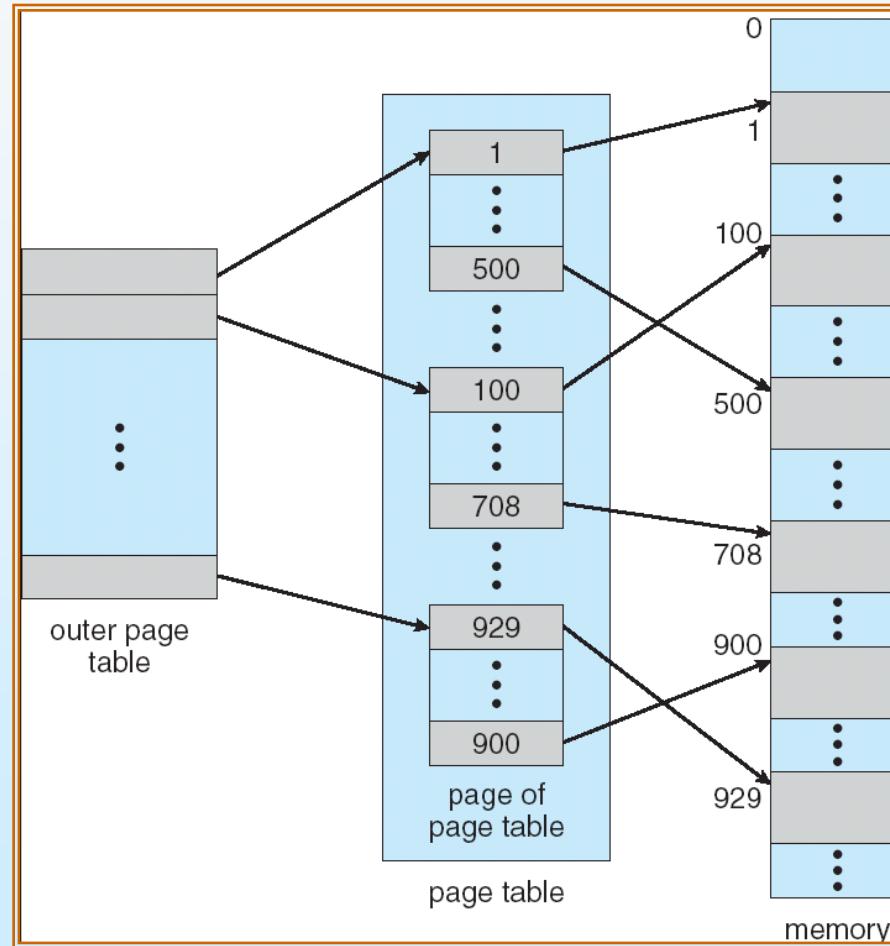


where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table





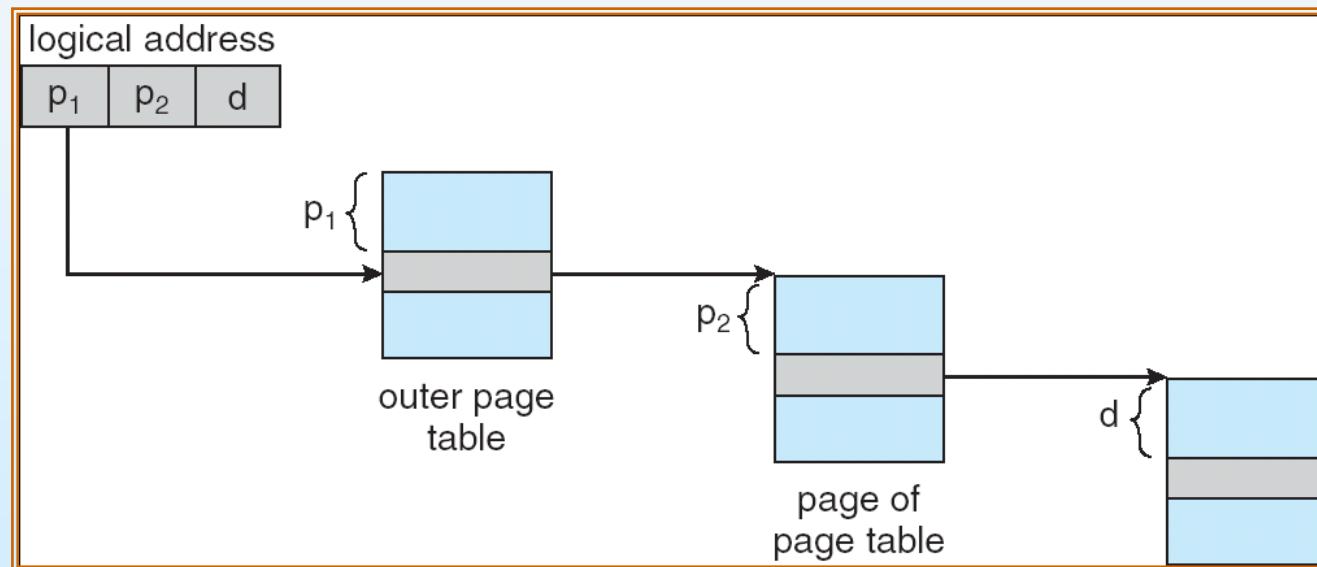
Two-Level Page-Table Scheme





Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture



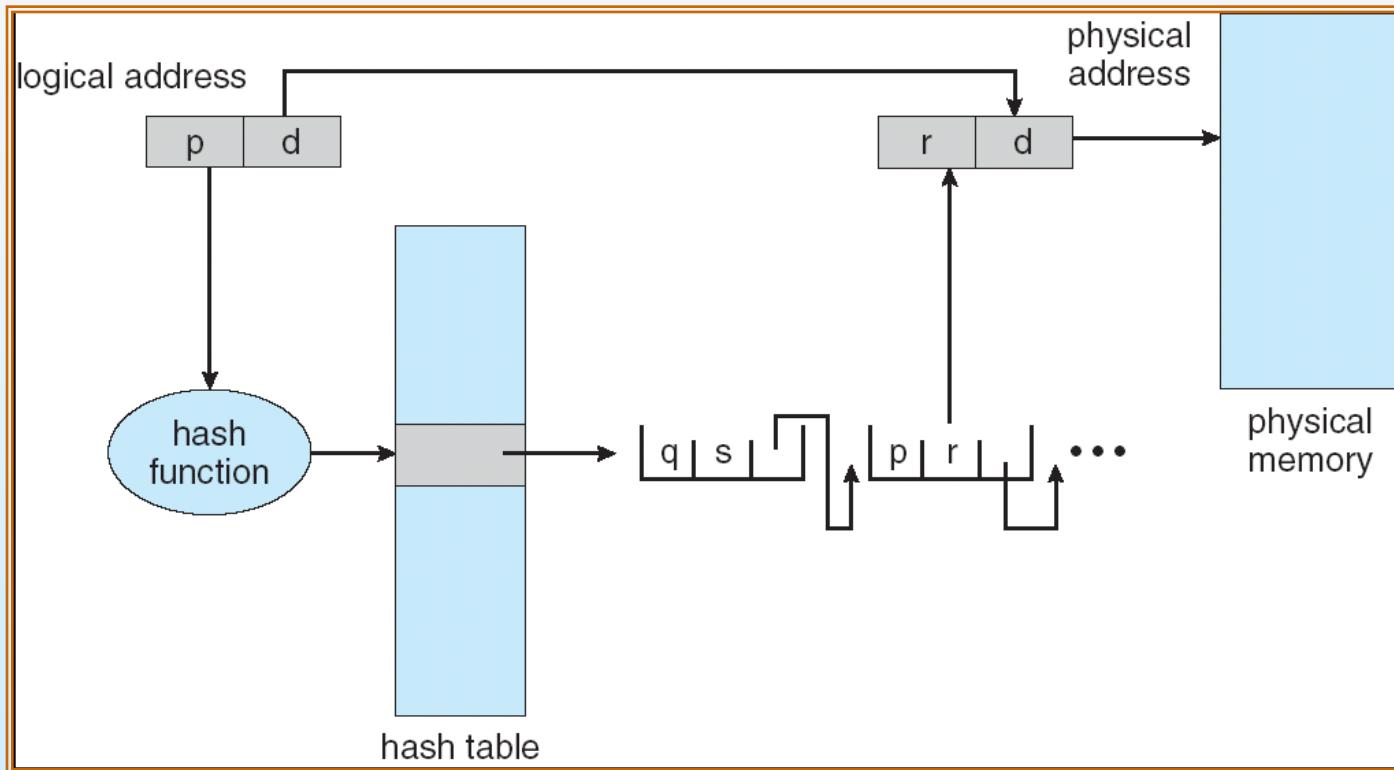


Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.



Hashed Page Table





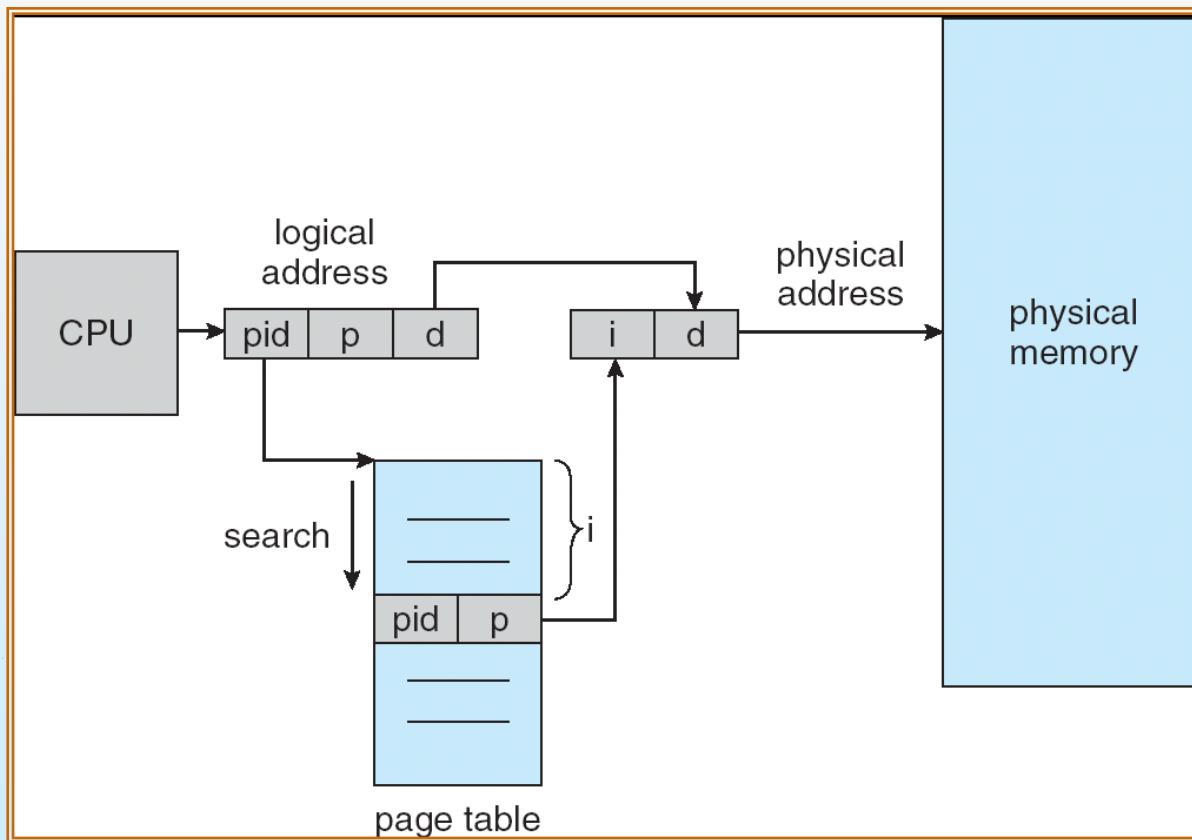
Inverted Page Table

- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries





Inverted Page Table Architecture





Shared Pages

□ Shared code

- One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
- Shared code must appear in same location in the logical address space of all processes

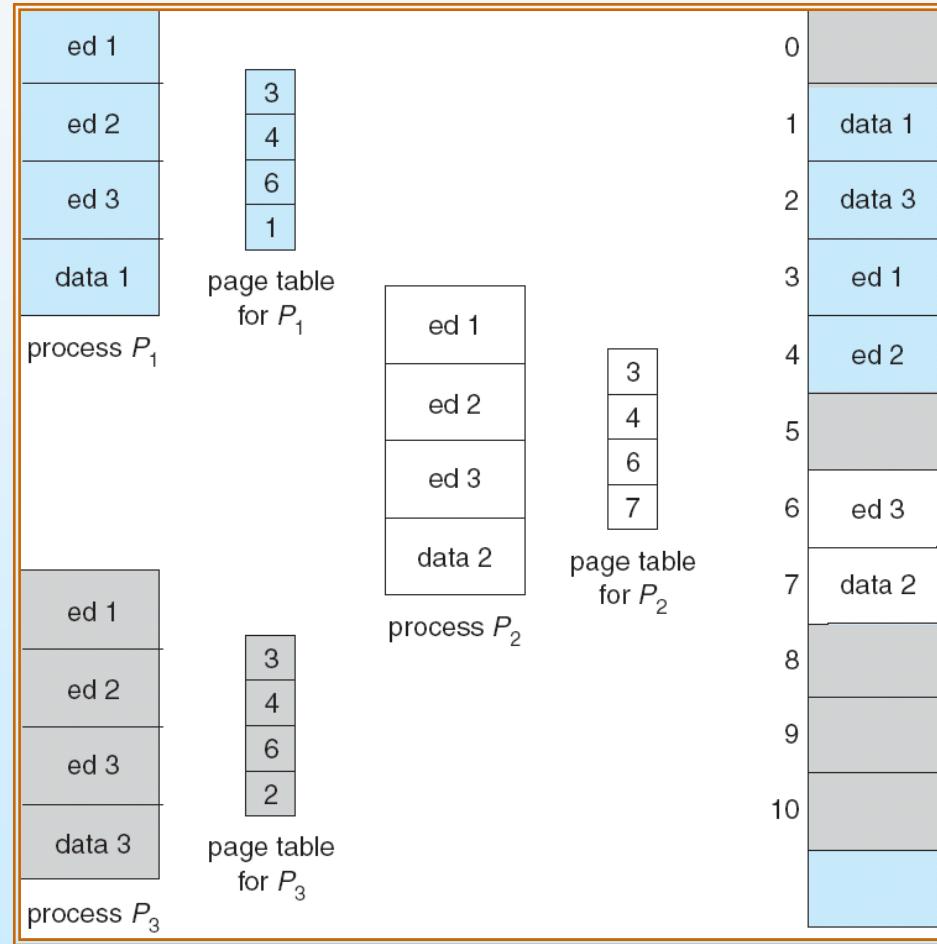
□ Private code and data

- 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





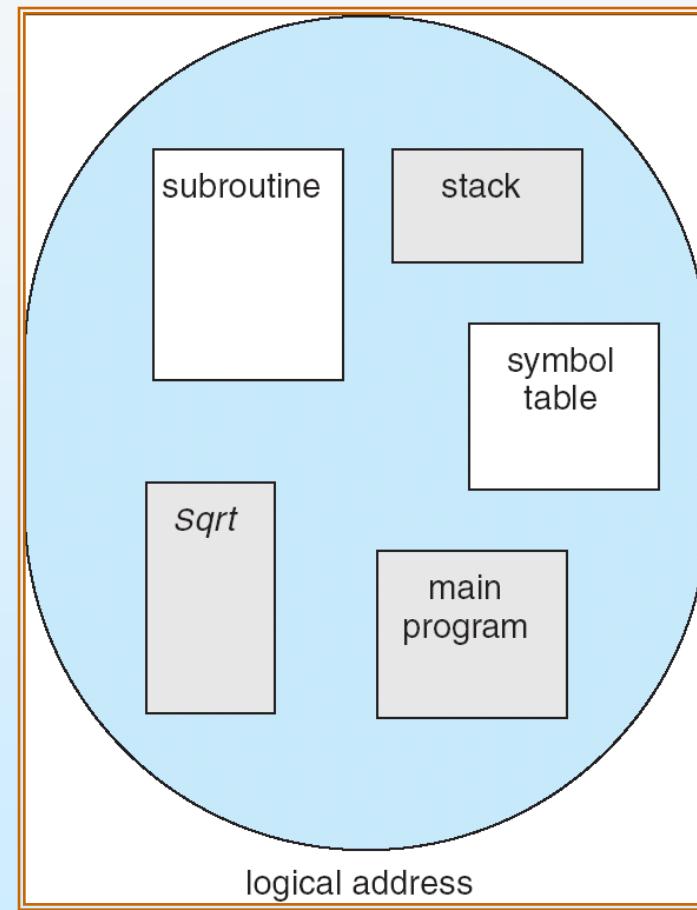
Segmentation

- 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



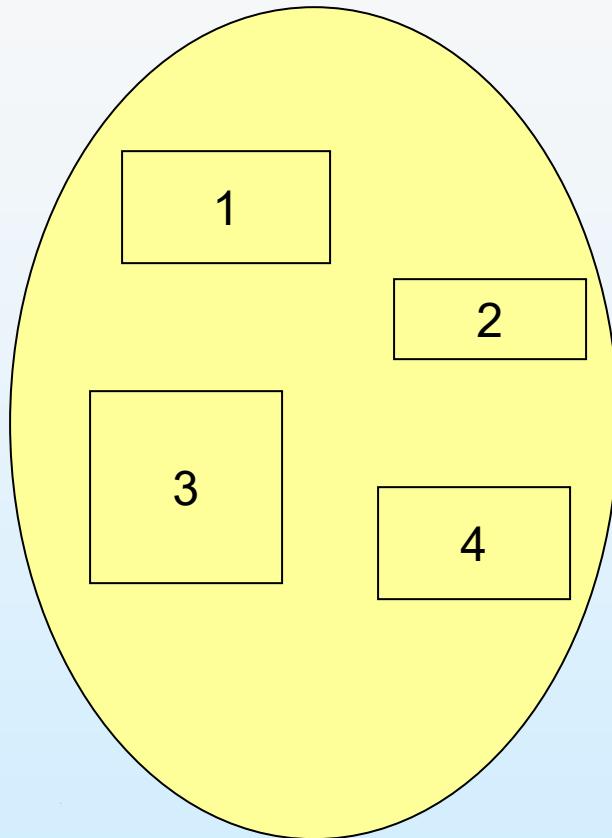


User's View of a Program

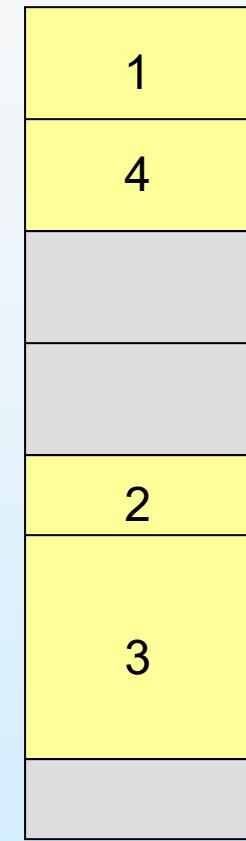




Logical View of Segmentation



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 $s < \text{STLR}$





Segmentation Architecture (Cont.)

- **Relocation.**

- dynamic
 - by segment table

- **Sharing.**

- shared segments
 - same segment number

- **Allocation.**

- first fit/best fit
 - external fragmentation





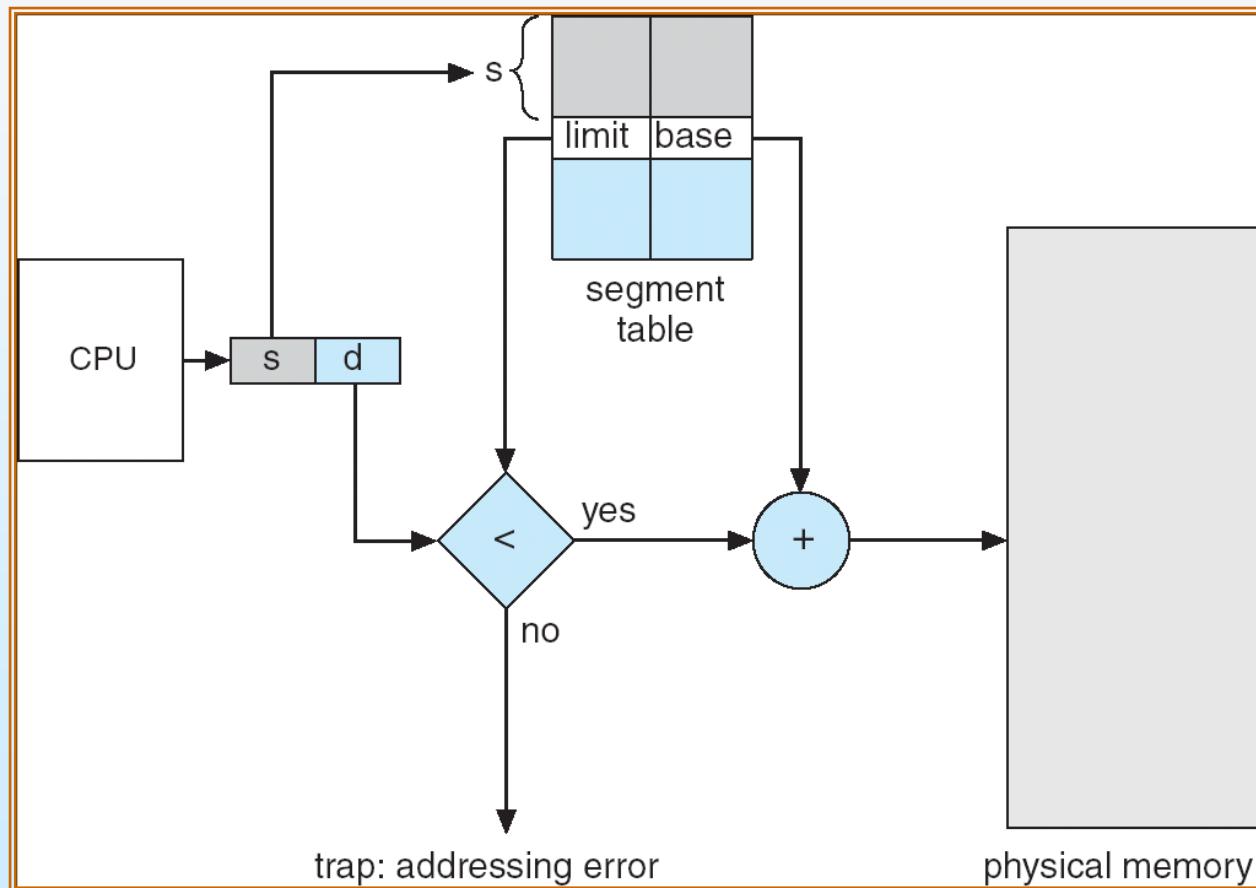
Segmentation Architecture (Cont.)

- 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



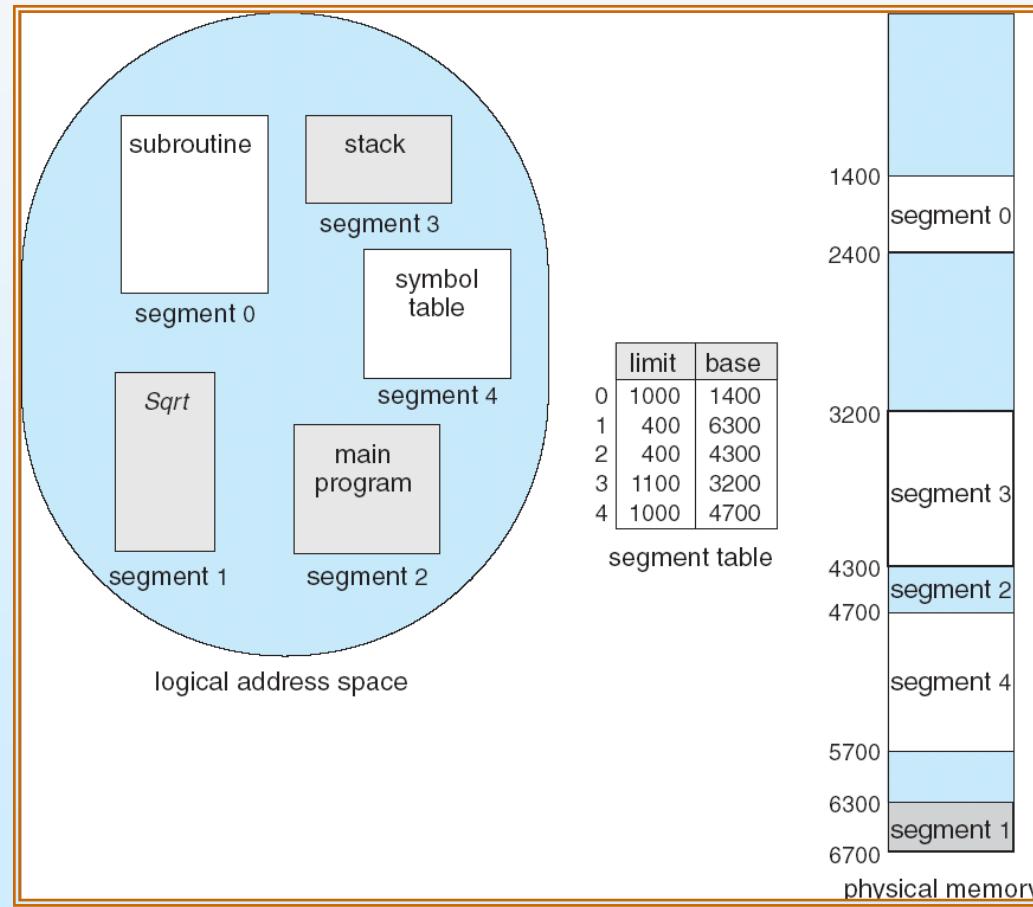


Address Translation Architecture



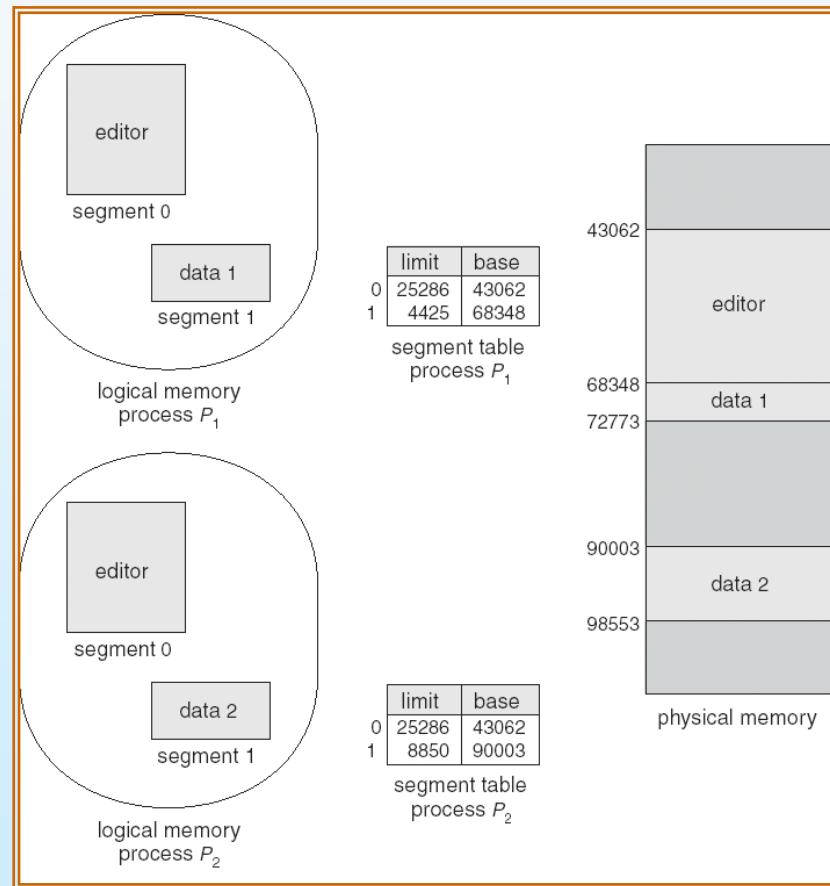


Example of Segmentation





Sharing of Segments





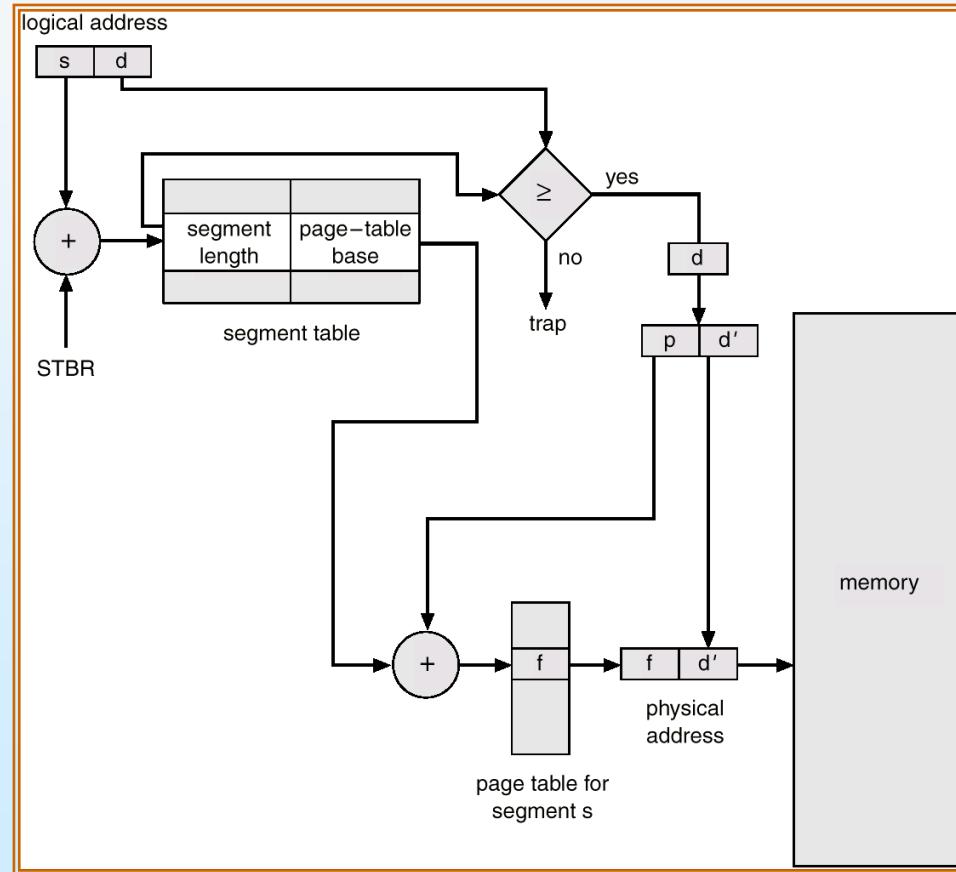
Segmentation with Paging – MULTICS

- The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments
- Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a *page table* for this segment





MULTICS Address Translation Scheme





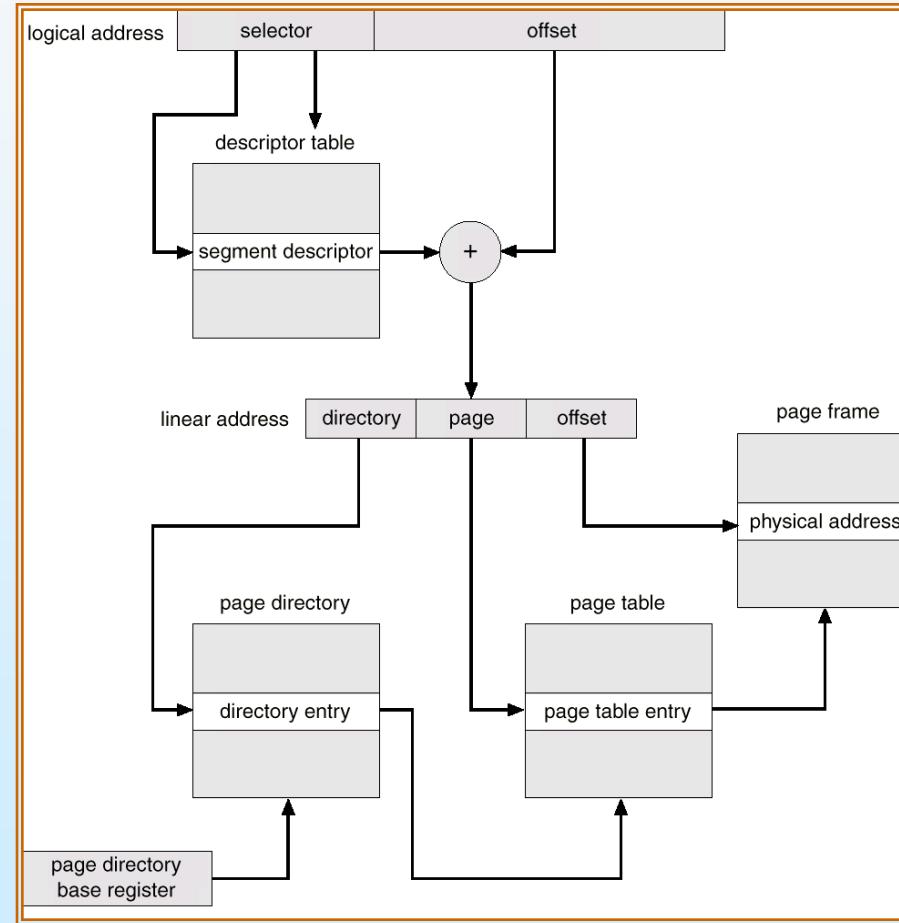
Segmentation with Paging – Intel 386

- As shown in the following diagram, the Intel 386 uses segmentation with paging for memory management with a two-level paging scheme





Intel 30386 Address Translation



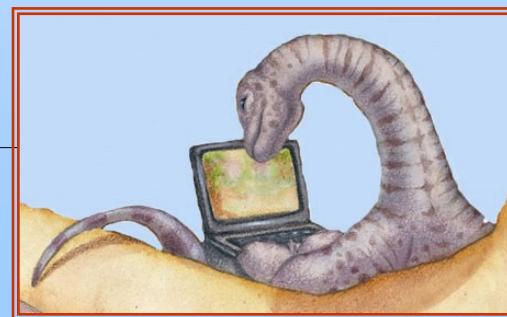


Linux on Intel 80x86

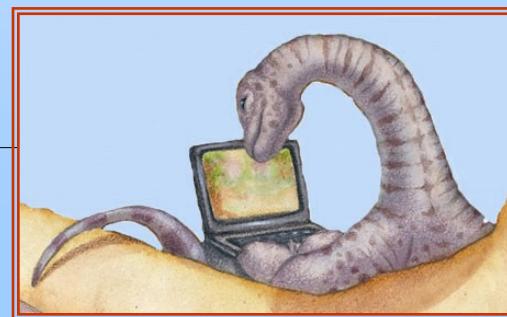
- Uses minimal segmentation to keep memory management implementation more portable
- Uses 6 segments:
 - Kernel code
 - Kernel data
 - User code (shared by all user processes, using logical addresses)
 - User data (likewise shared)
 - Task-state (per-process hardware context)
 - LDT
- Uses 2 protection levels:
 - Kernel mode
 - User mode



End of Chapter 8



Chapter 9: Virtual Memory





Chapter 9: Virtual Memory

- Background
- Demand Paging
- Process Creation
- Page Replacement
- Allocation of Frames
- Thrashing
- Demand Segmentation
- Operating System Examples





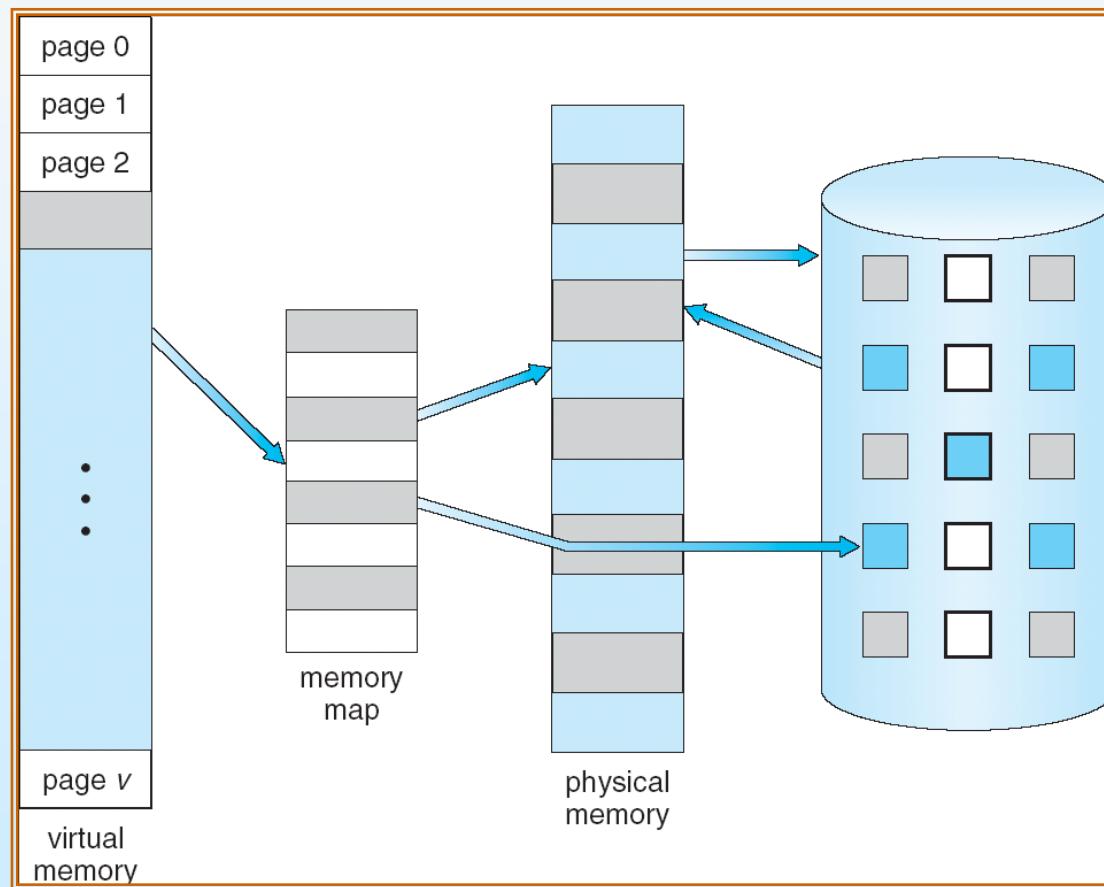
Background

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



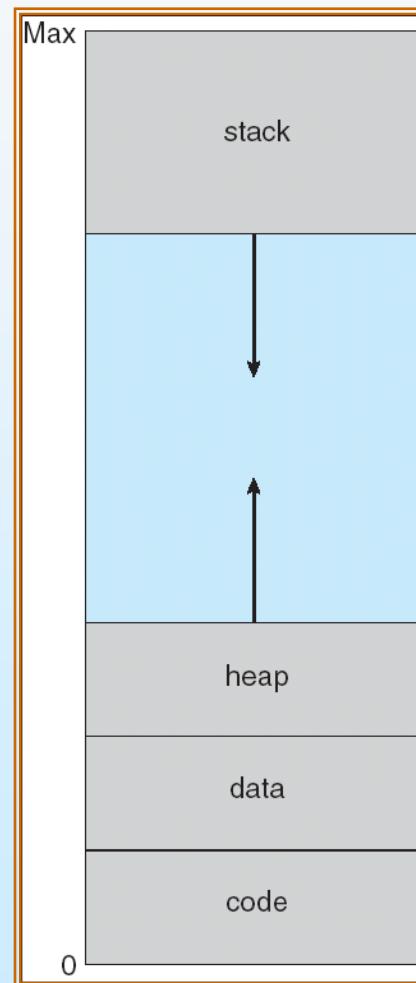


Virtual Memory That is Larger Than Physical Memory



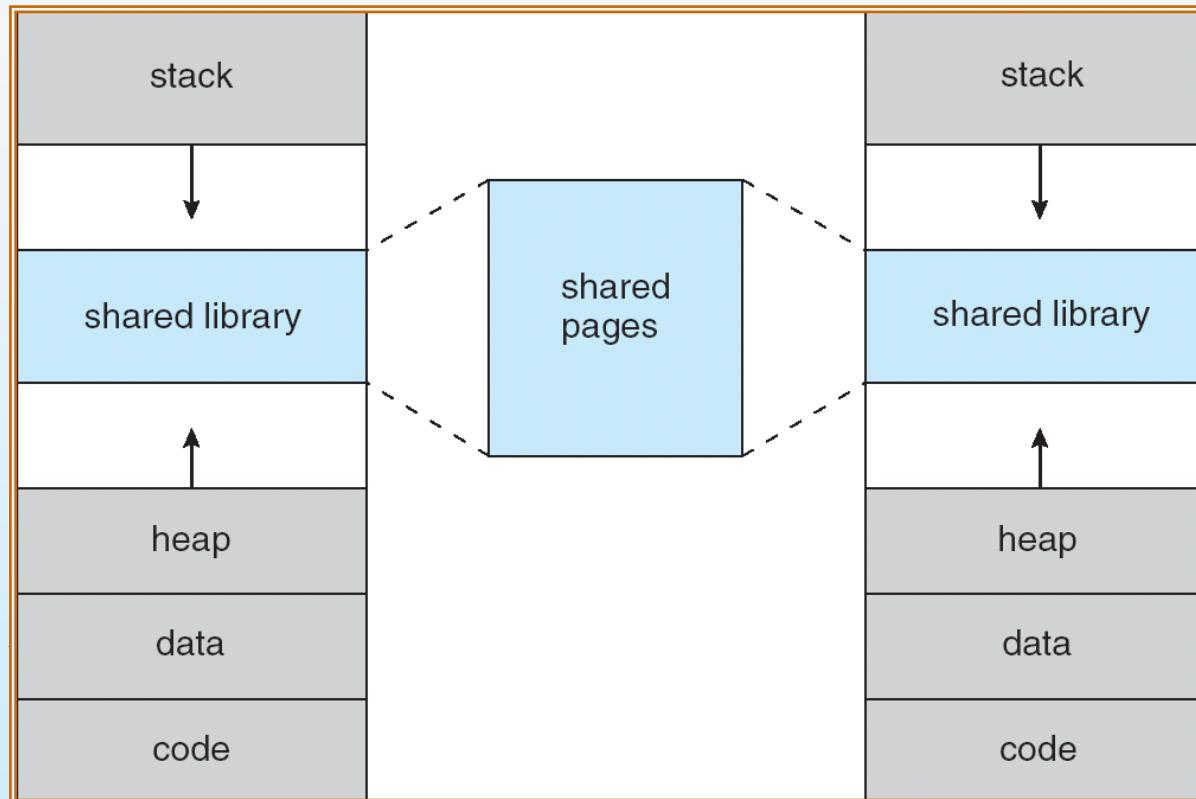


Virtual-address Space





Shared Library Using Virtual Memory





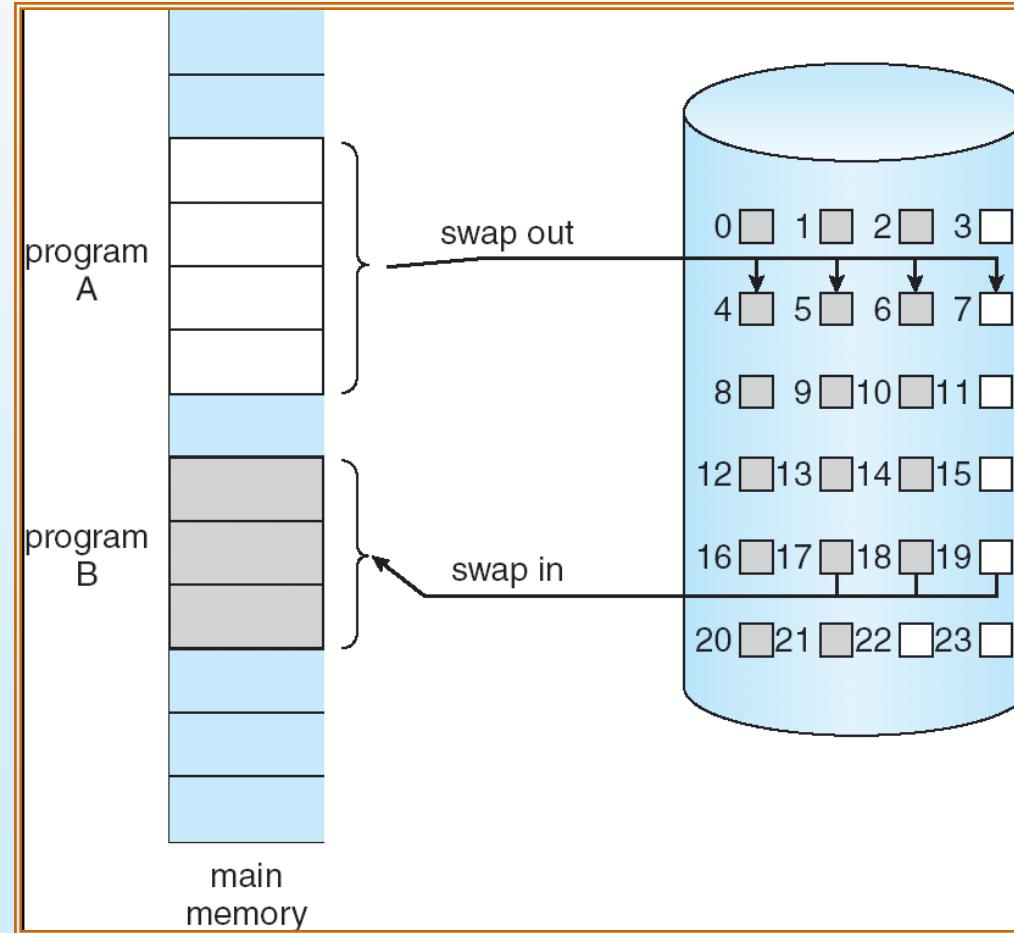
Demand Paging

- Bring a page into memory only when it is needed
 - Less I/O needed
 - Less memory needed
 - Faster response
 - More users
- Page is needed ⇒ reference to it
 - invalid reference ⇒ abort
 - not-in-memory ⇒ bring to memory





Transfer of a Paged Memory to Contiguous Disk Space





Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated ($1 \Rightarrow$ in-memory, $0 \Rightarrow$ not-in-memory)
- Initially valid–invalid but is set to 0 on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
	1
	1
	1
	1
	0
:	
	0
	0

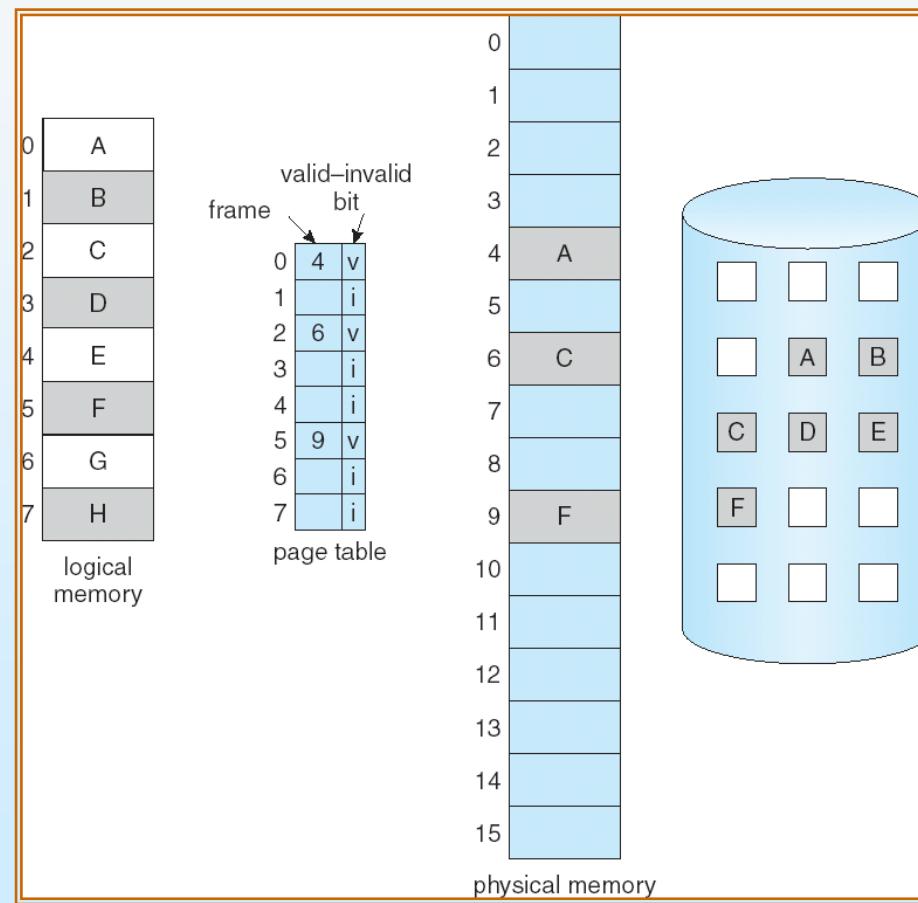
page table

- During address translation, if valid–invalid bit in page table entry is 0 \Rightarrow page fault





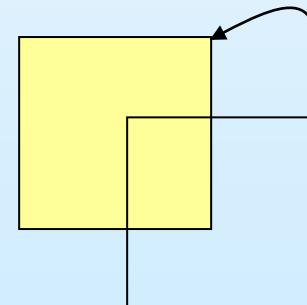
Page Table When Some Pages Are Not in Main Memory





Page Fault

- If there is ever a reference to a page, first reference will trap to OS \Rightarrow page fault
- OS looks at another table to decide:
 - Invalid reference \Rightarrow abort.
 - Just not in memory.
- Get empty frame.
- Swap page into frame.
- Reset tables, validation bit = 1.
- Restart instruction: Least Recently Used
 - block move

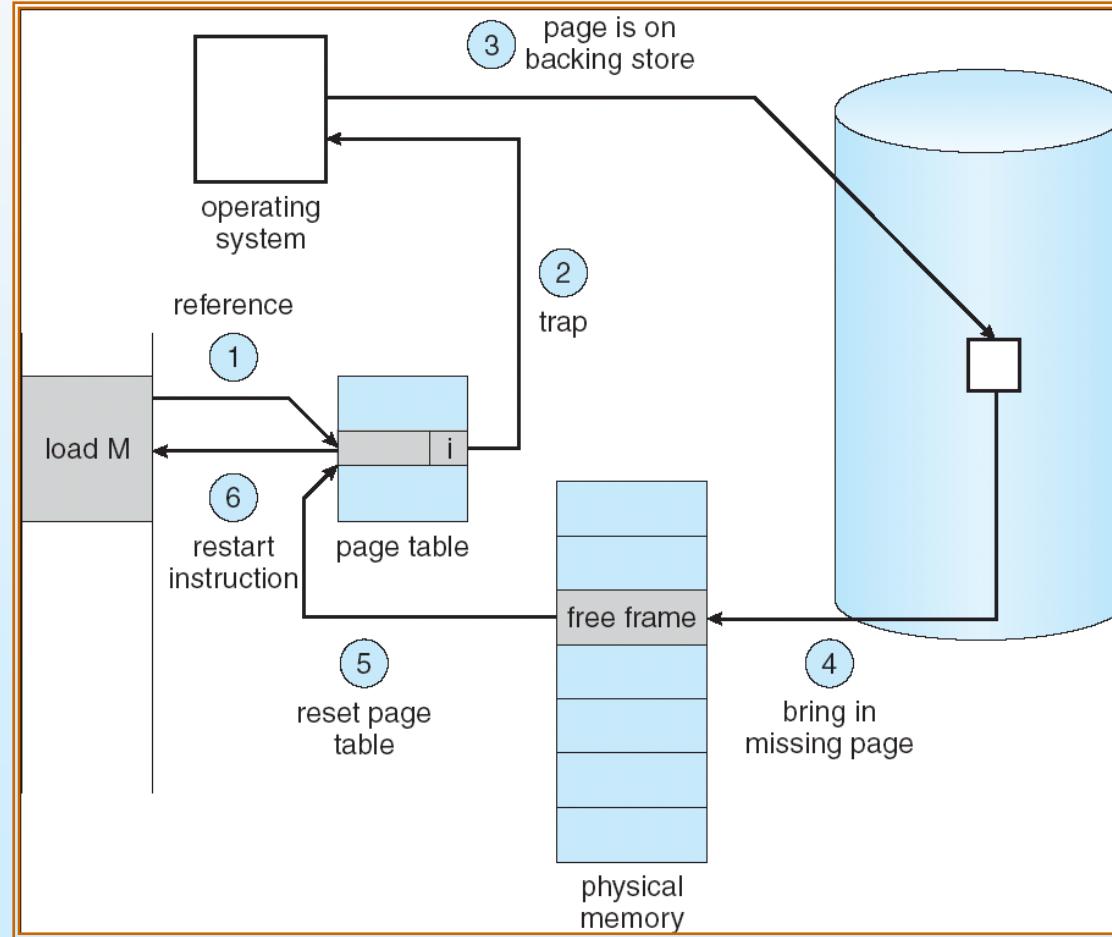


- auto increment/decrement location





Steps in Handling a Page Fault





What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out
 - algorithm
 - performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times





Performance of Demand Paging

- Page Fault Rate $0 \leq p \leq 1.0$
 - if $p = 0$ no page faults
 - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

$$\begin{aligned} \text{EAT} = & (1 - p) \times \text{memory access} \\ & + p \text{ (page fault overhead)} \\ & + [\text{swap page out}] \\ & + \text{swap page in} \\ & + \text{restart overhead}) \end{aligned}$$





Demand Paging Example

- Memory access time = 1 microsecond
- 50% of the time the page that is being replaced has been modified and therefore needs to be swapped out
- Swap Page Time = 10 msec = 10,000 msec
 - EAT = $(1 - p) \times 1 + p (15000)$
 - $1 + 15000P$ (in msec)





Process Creation

- Virtual memory allows other benefits during process creation:
 - Copy-on-Write
 - Memory-Mapped Files (later)





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
- Free pages are allocated from a **pool** of zeroed-out pages





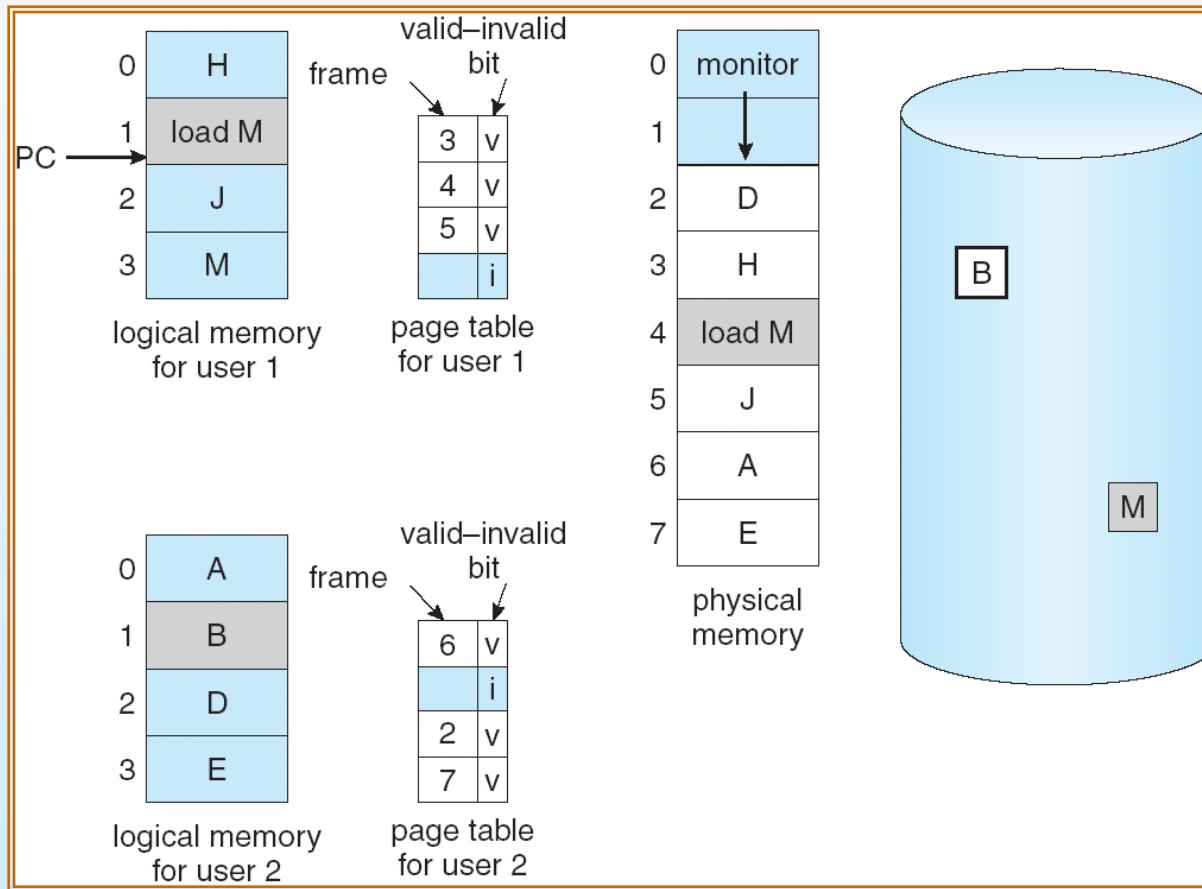
Page Replacement

- 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



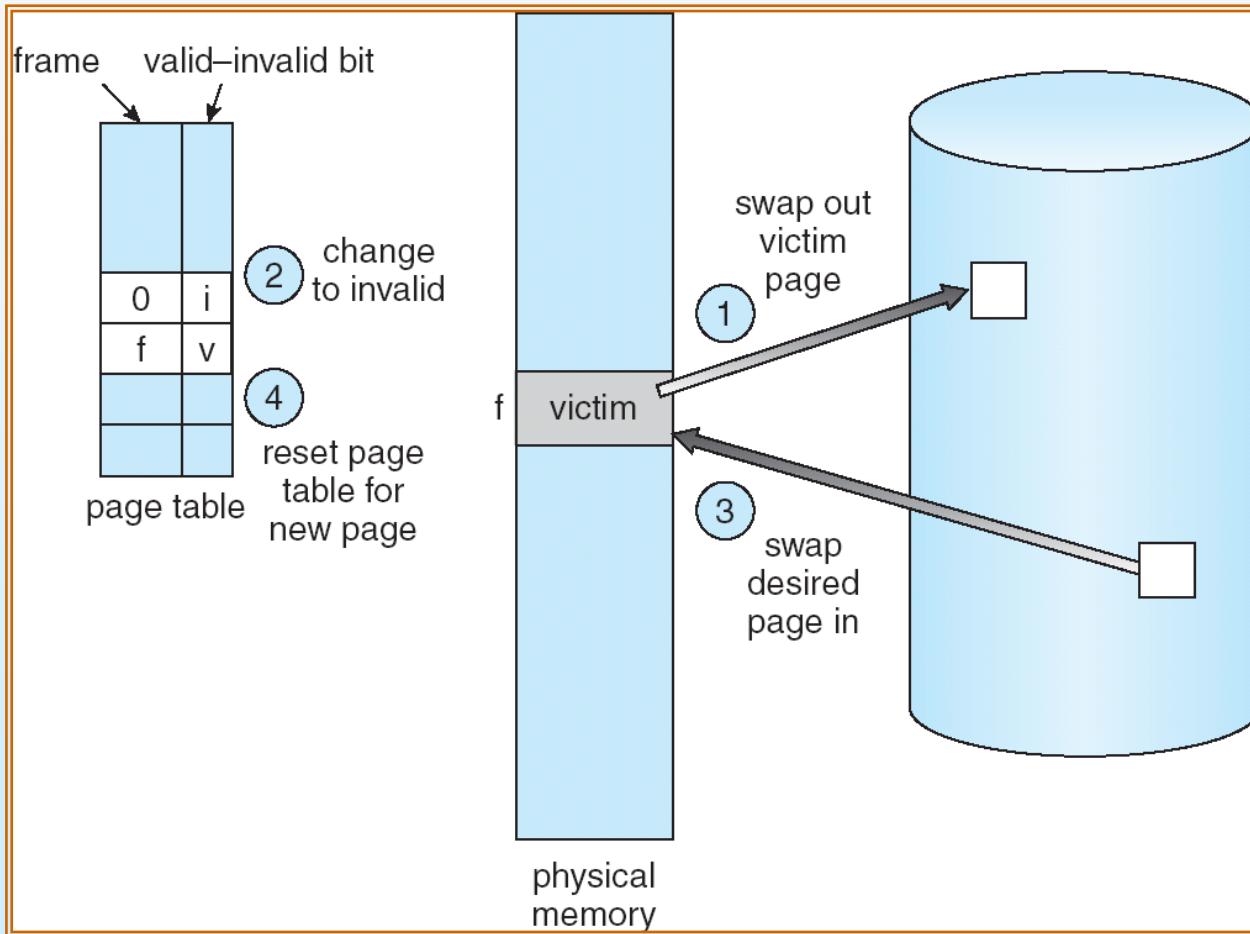


Basic Page Replacement

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
3. Read the desired page into the (newly) free frame. Update the page and frame tables.
4. Restart the process



Page Replacement





Page Replacement Algorithms

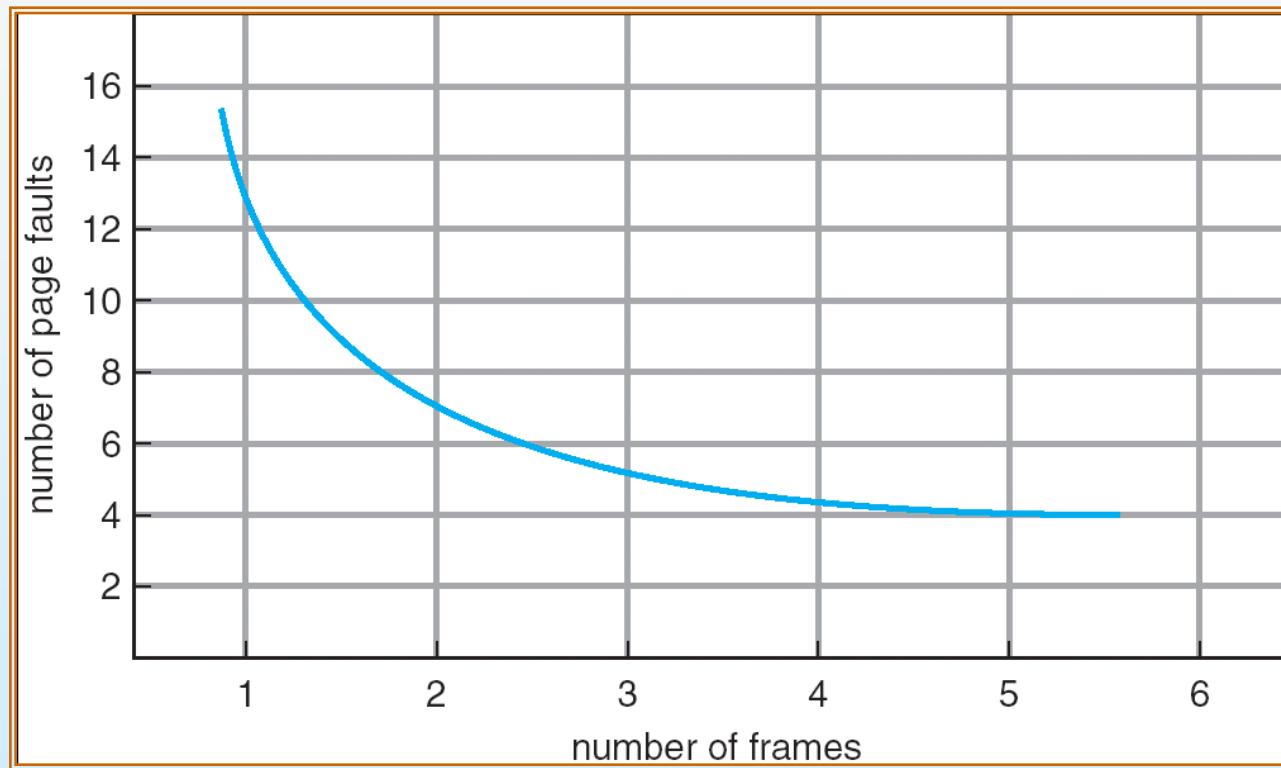
- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- In all our examples, the reference string is

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5





Graph of Page Faults Versus The Number of Frames





First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

- 4 frames

1	1	4	5
2	2	1	3 9 page faults
3	3	2	4

- FIFO Replacement – Belady's Anomaly
 - more frames \Rightarrow more page faults

1	1	5	4
2	2	1	5 10 page faults
3	3	2	
4	4	3	

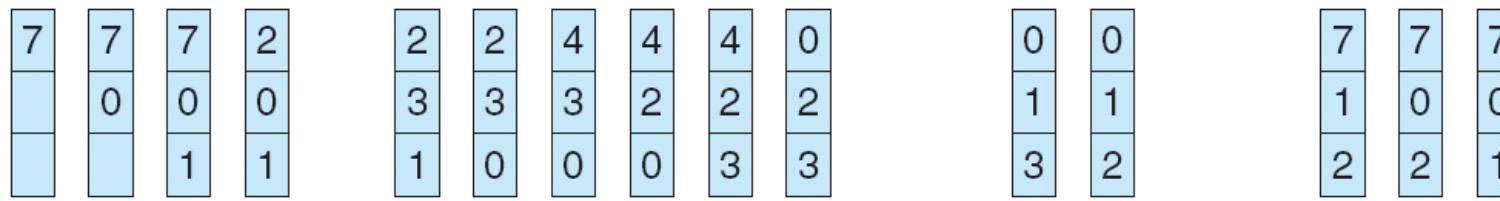




FIFO Page Replacement

reference string

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

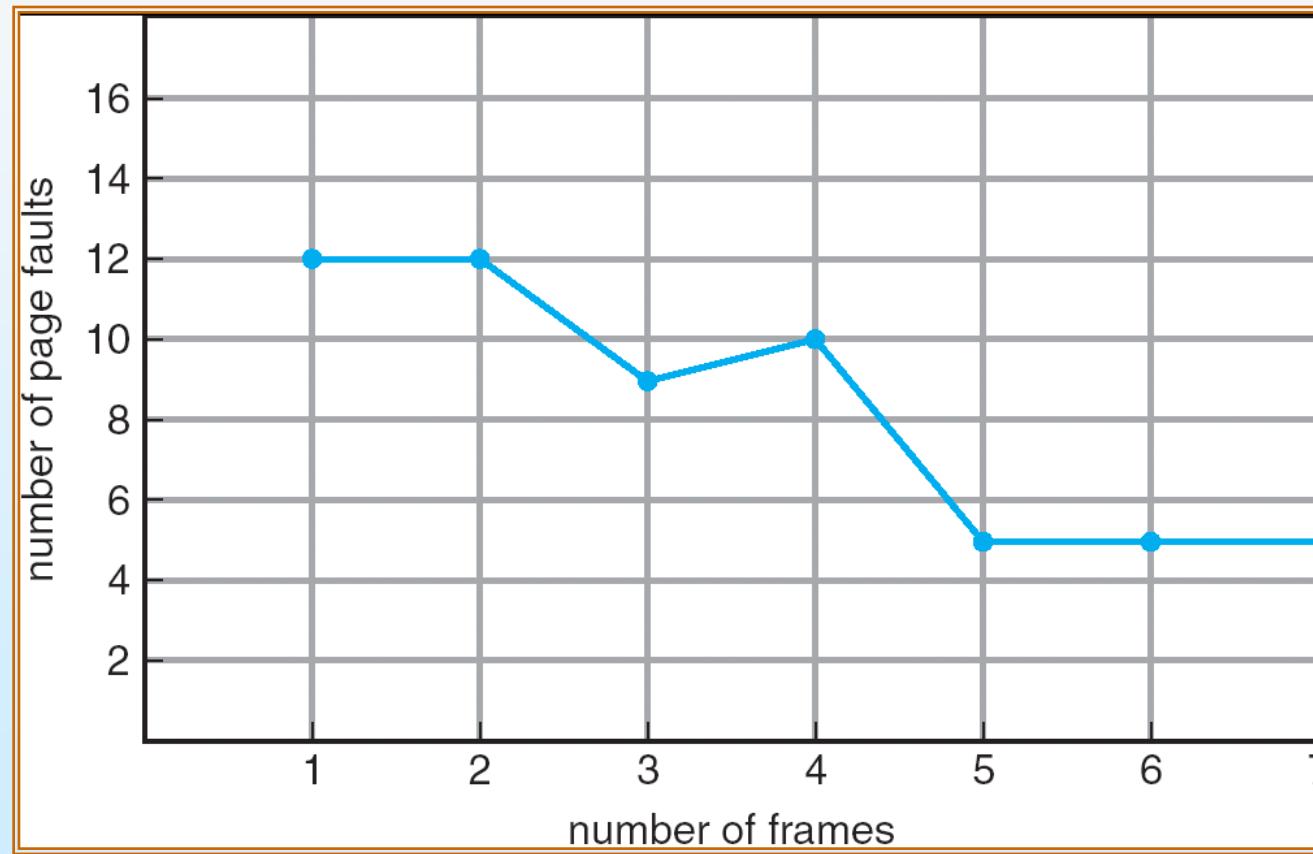


page frames





FIFO Illustrating Belady's Anomaly

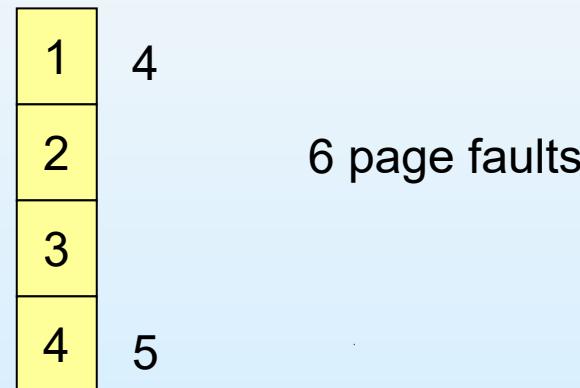




Optimal Algorithm

- Replace page that will not be used for longest period of time
- 4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

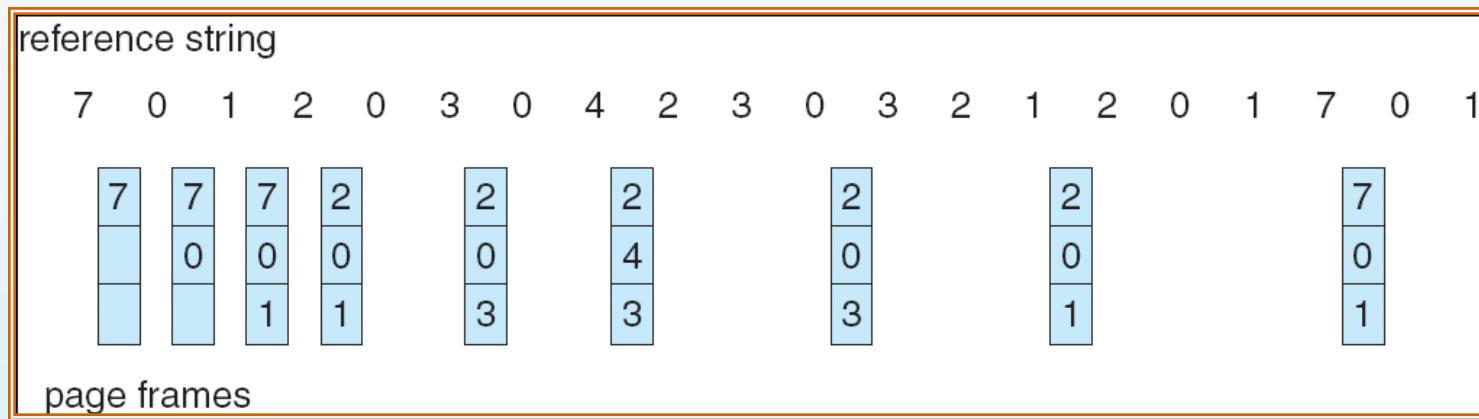


- How do you know this?
- Used for measuring how well your algorithm performs





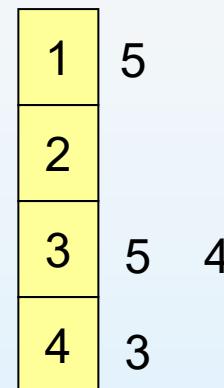
Optimal Page Replacement





Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



- 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 determine which are to change

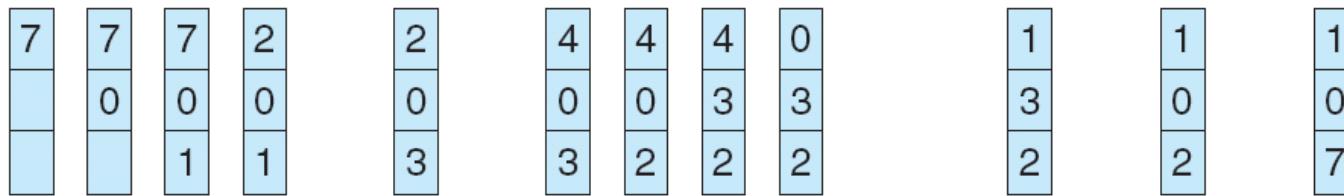




LRU Page Replacement

reference string

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



page frames





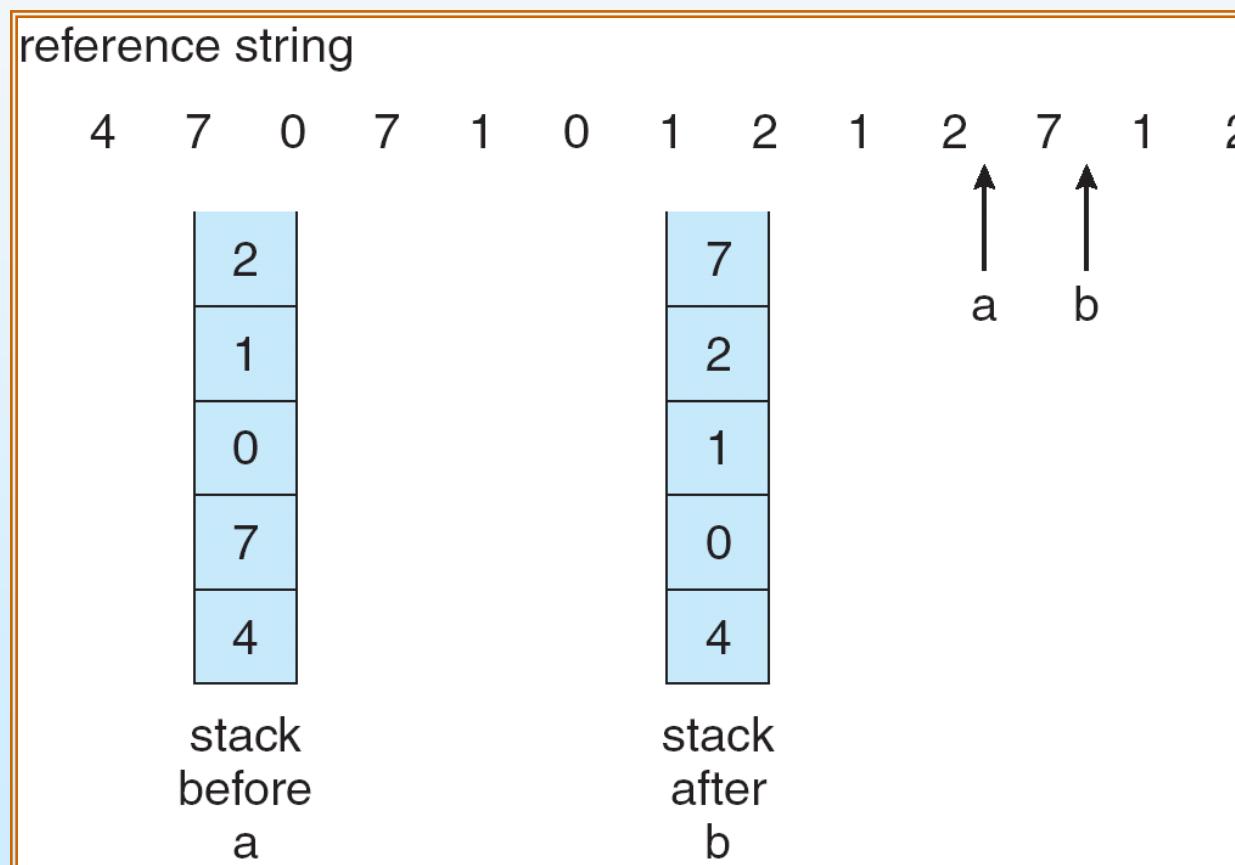
LRU Algorithm (Cont.)

- 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
 - No search for replacement





Use Of A Stack to Record The Most Recent Page References





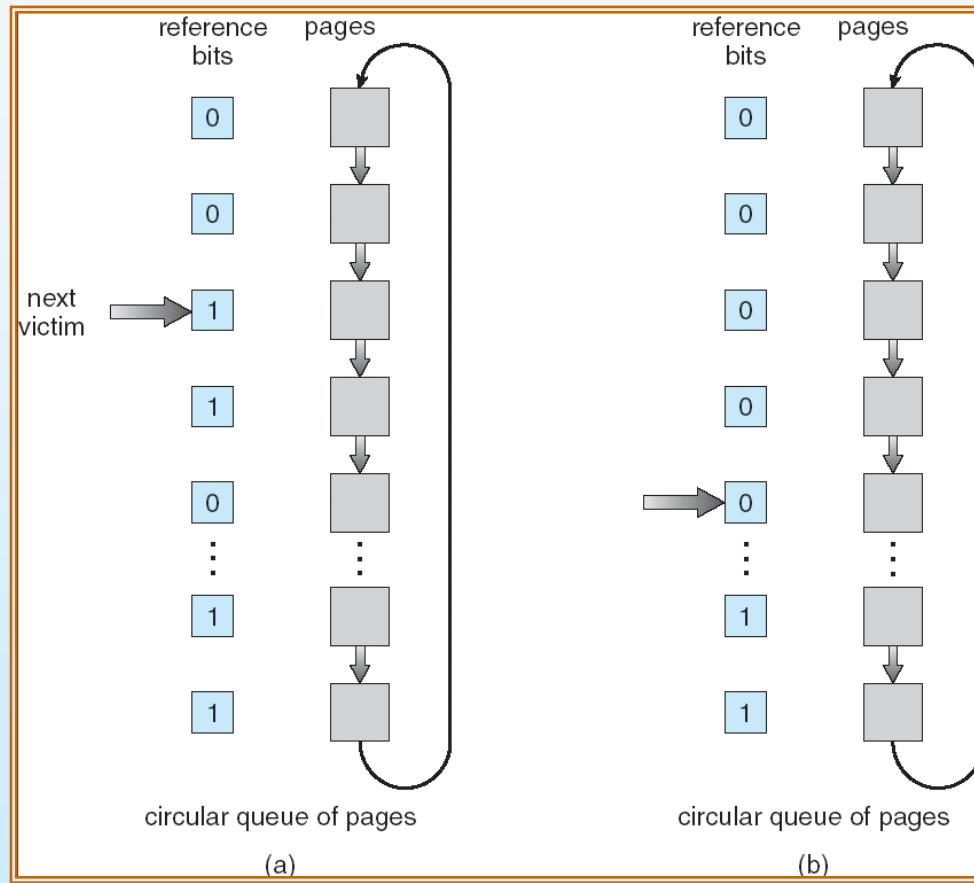
LRU Approximation Algorithms

- Reference bit
 - With each page associate a bit, initially = 0
 - When page is referenced bit set to 1
 - Replace the one which is 0 (if one exists). We do not know the order, however.
- Second chance
 - Need reference bit
 - Clock replacement
 - If page to be replaced (in clock order) has reference bit = 1 then:
 - ▶ set reference bit 0
 - ▶ leave page in memory
 - ▶ replace next page (in clock order), subject to same rules





Second-Chance (clock) Page-Replacement Algorithm





Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **LFU Algorithm:** replaces page with smallest count
- **MFU Algorithm:** based on the argument that the page with the smallest count was probably just brought in and has yet to be used





Allocation of Frames

- Each process needs *minimum* number of pages
- 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*
- Two major allocation schemes
 - fixed allocation
 - priority allocation





Fixed Allocation

- Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation – Allocate according to the size of process
 - s_i = size of process p_i
 - $S = \sum s_i$
 - m = total number of frames
 - a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$





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





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
- **Local replacement** – each process selects from only its own set of allocated frames





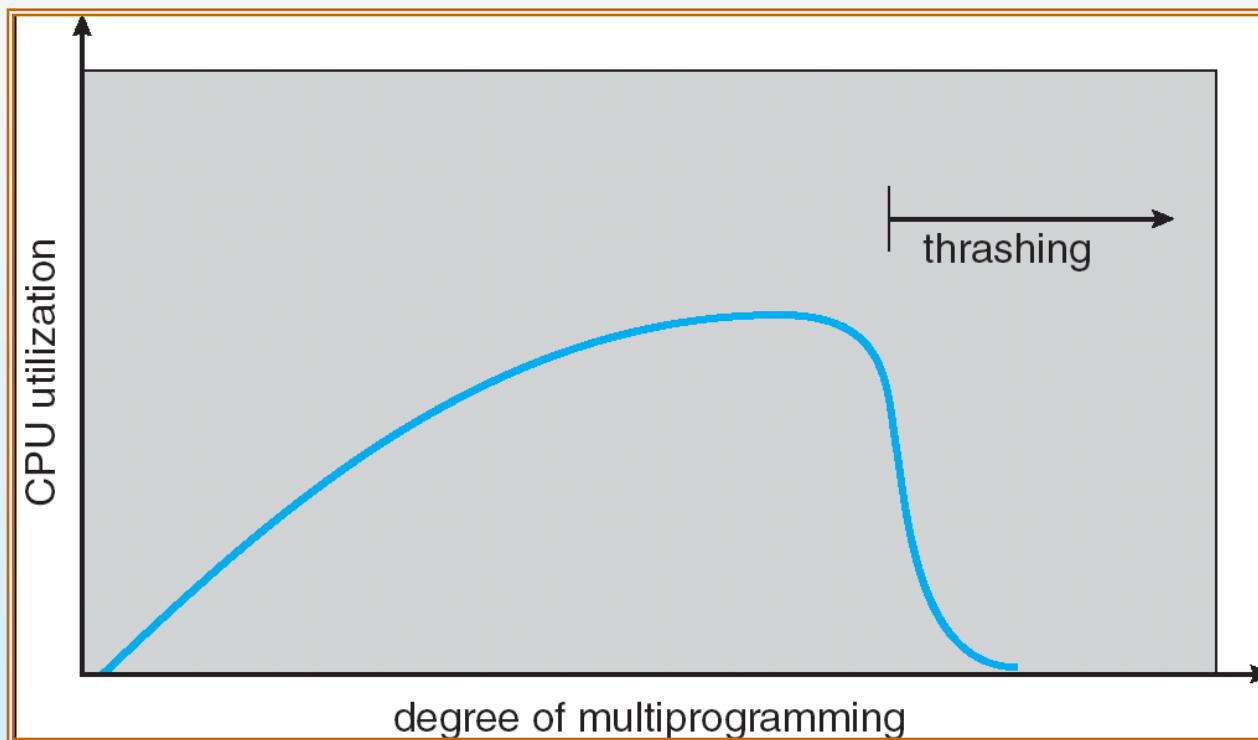
Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
 - low CPU utilization
 - operating system thinks that it needs to increase the degree of multiprogramming
 - another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out





Thrashing (Cont.)





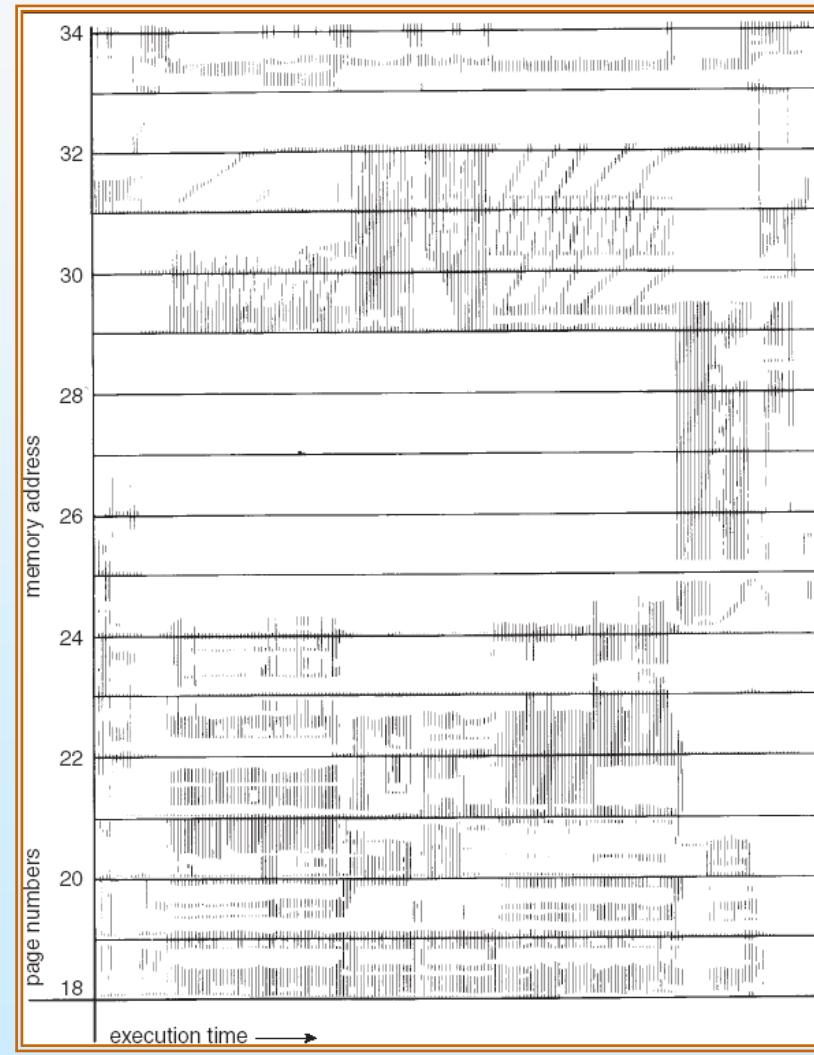
Demand Paging and Thrashing

- Why does demand paging work?
Locality model
 - Process migrates from one locality to another
 - Localities may overlap
- Why does thrashing occur?
 Σ size of locality > total memory size





Locality In A Memory-Reference Pattern





Working-Set Model

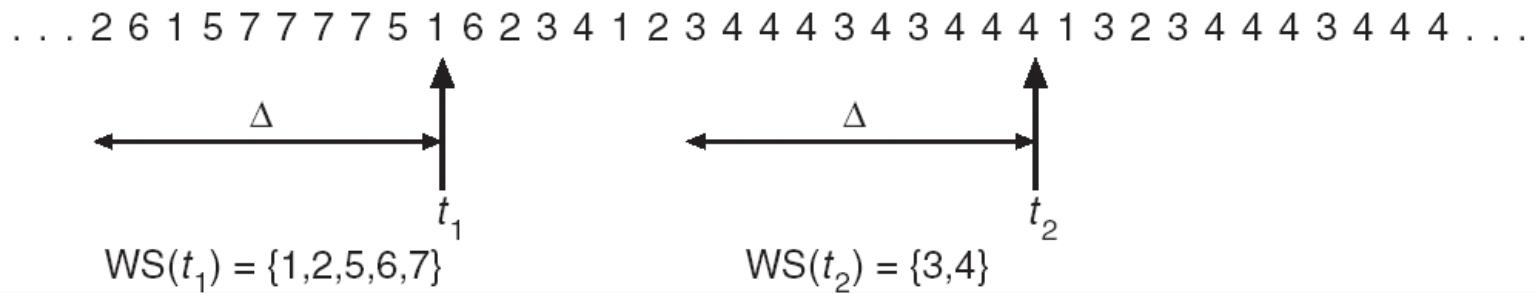
- $\Delta \equiv$ working-set window \equiv a fixed number of page references
Example: 10,000 instruction
- WSS_i (working set of Process P_i) =
total number of pages referenced in the most recent Δ (varies in time)
 - if Δ too small will not encompass entire locality
 - if Δ too large will encompass several localities
 - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \sum WSS_i \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend one of the processes





Working-set model

page reference table





Keeping Track of the Working Set

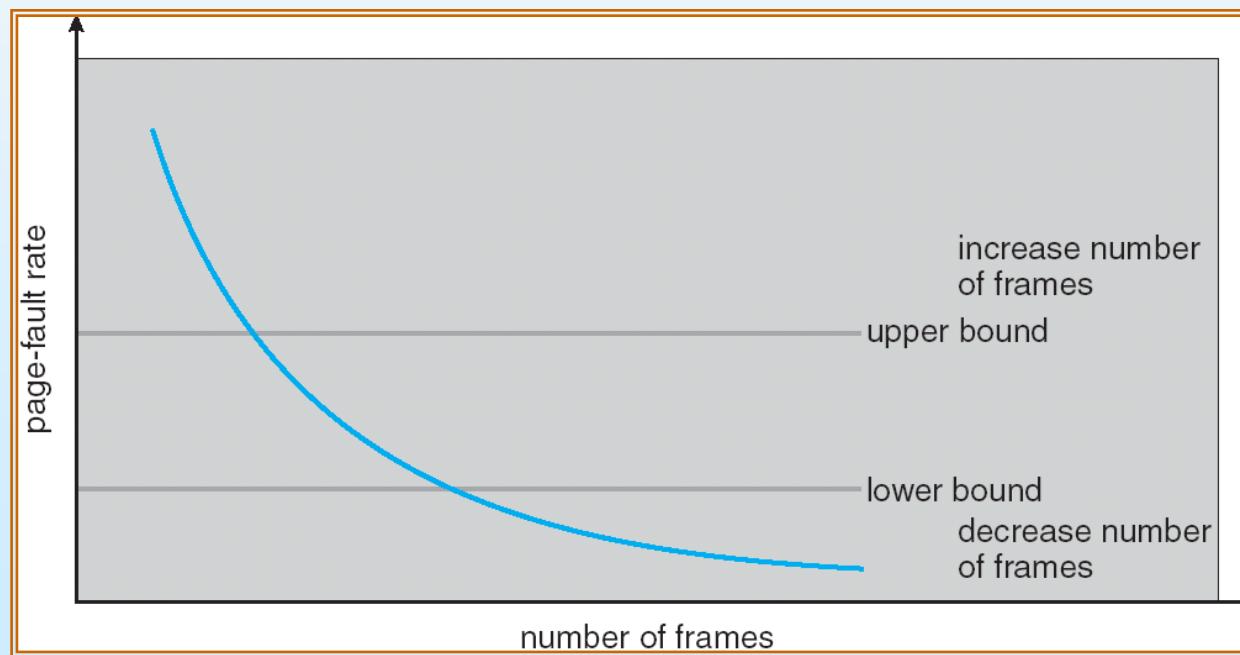
- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
 - Timer interrupts after every 5000 time units
 - Keep in memory 2 bits for each page
 - Whenever a timer interrupt occurs, copy and sets the values of all reference bits to 0
 - If one of the bits in memory = 1 \Rightarrow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





Page-Fault Frequency Scheme

- Establish “acceptable” page-fault rate
 - If actual rate too low, process loses frame
 - If actual rate too high, process gains frame





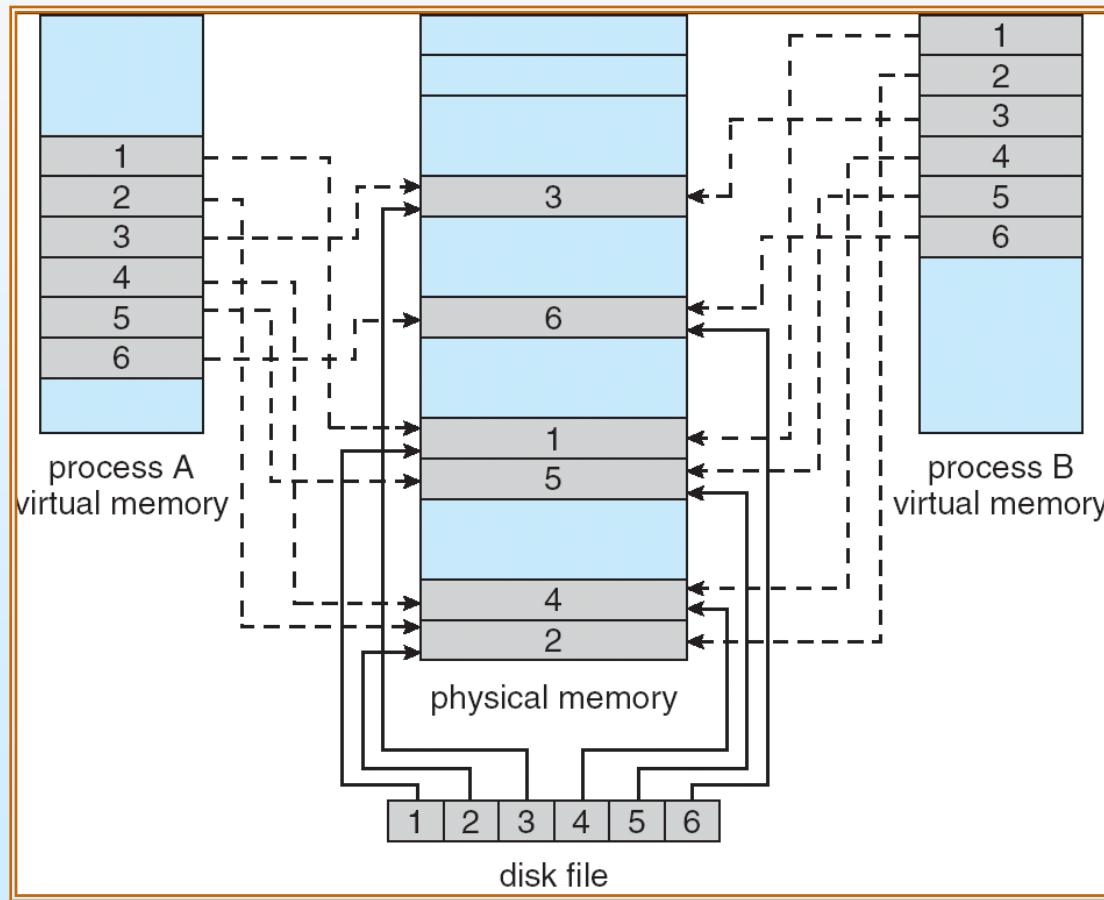
Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than **read() write()** system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared





Memory Mapped Files





Memory-Mapped Files in Java

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
public class MemoryMapReadOnly
{
    // Assume the page size is 4 KB
    public static final int PAGE_SIZE = 4096;
    public static void main(String args[]) throws IOException {
        RandomAccessFile inFile = new RandomAccessFile(args[0],"r");
        FileChannel in = inFile.getChannel();
        MappedByteBuffer mappedBuffer =
            in.map(FileChannel.MapMode.READ ONLY, 0, in.size());
        long numPages = in.size() / (long)PAGE_SIZE;
        if (in.size() % PAGE_SIZE > 0)
            ++numPages;
    }
}
```





Memory-Mapped Files in Java (cont)

```
// we will "touch" the first byte of every page
int position = 0;
for (long i = 0; i < numPages; i++) {
    byte item = mappedBuffer.get(position);
    position += PAGE SIZE;
}
in.close();
inFile.close();
}
```

- The API for the map() method is as follows:

```
map(mode, position, size)
```





Other Issues -- Prepaging

- Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume s pages are prepaged and 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





Other Issues – Page Size

- Page size selection must take into consideration:
 - fragmentation
 - table size
 - I/O overhead
 - locality





Other Issues – TLB Reach

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





Other Issues – Program Structure

- Program structure

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

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

128 x 128 = 16,384 page faults

- Program 2

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

128 page faults





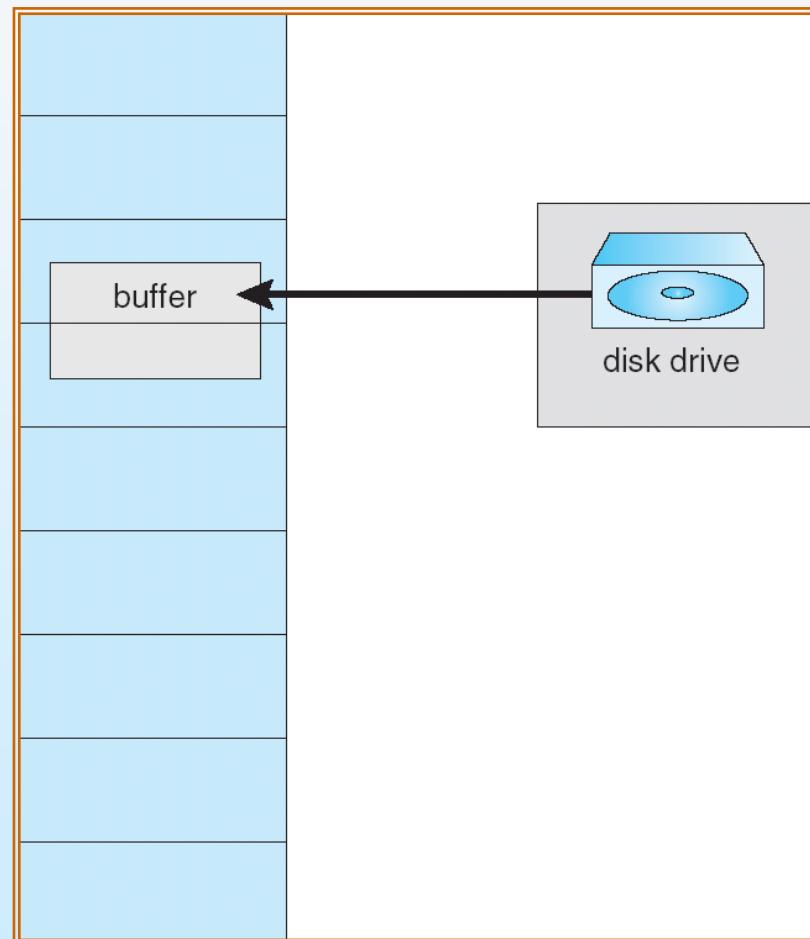
Other Issues – I/O interlock

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





Reason Why Frames Used For I/O Must Be In Memory





Operating System Examples

- Windows XP
- Solaris





Windows XP

- 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





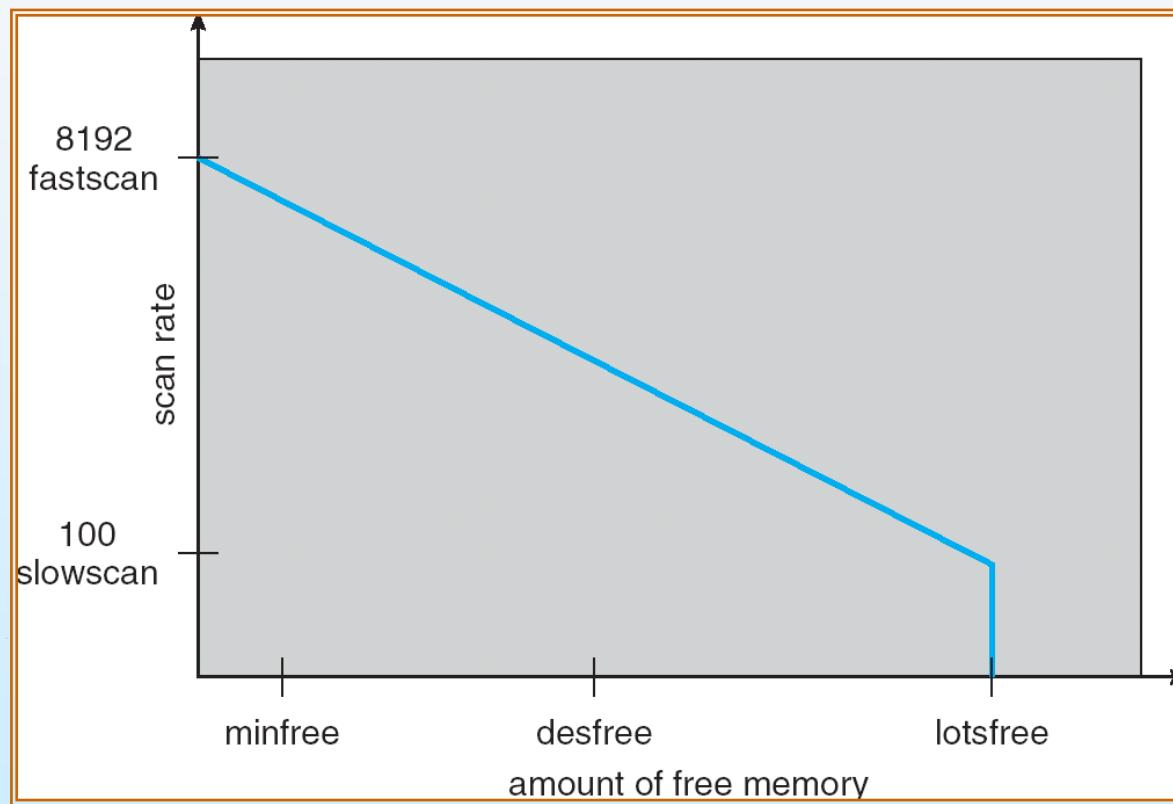
Solaris

- 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

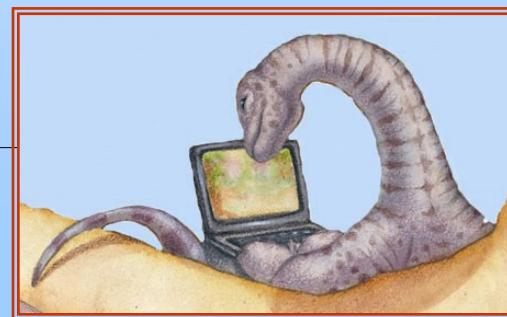




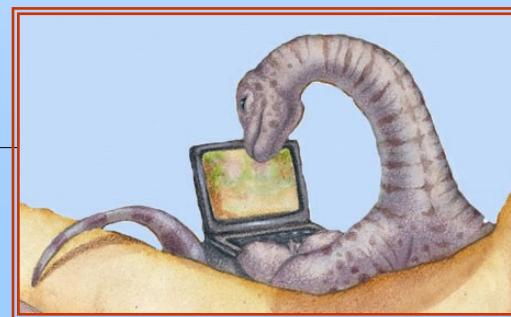
Solaris 2 Page Scanner



End of Chapter 9



Chapter 10: File-System Interface





Chapter 10: File-System Interface

- File Concept
- Access Methods
- Directory Structure
- File-System Mounting
- File Sharing
- Protection





Objectives

- To explain the function of file systems
- To describe the interfaces to file systems
- To discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures
- To explore file-system protection





File Concept

- Contiguous logical address space
- Types:
 - Data
 - ▶ numeric
 - ▶ character
 - ▶ binary
 - Program





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





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





File Operations

- File is an **abstract data type**
- **Create**
- **Write**
- **Read**
- **Reposition within file**
- **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:
 - File pointer: pointer to last read/write location, per process that has the file open
 - File-open count: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
 - Disk location of the file: cache of data access information
 - Access rights: per-process access mode information





Open File Locking

- Provided by some operating systems and file systems
- 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  
exclusiveLock.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





Access Methods

□ Sequential Access

read next
write next
reset
no read after last write
(rewrite)

□ Direct Access

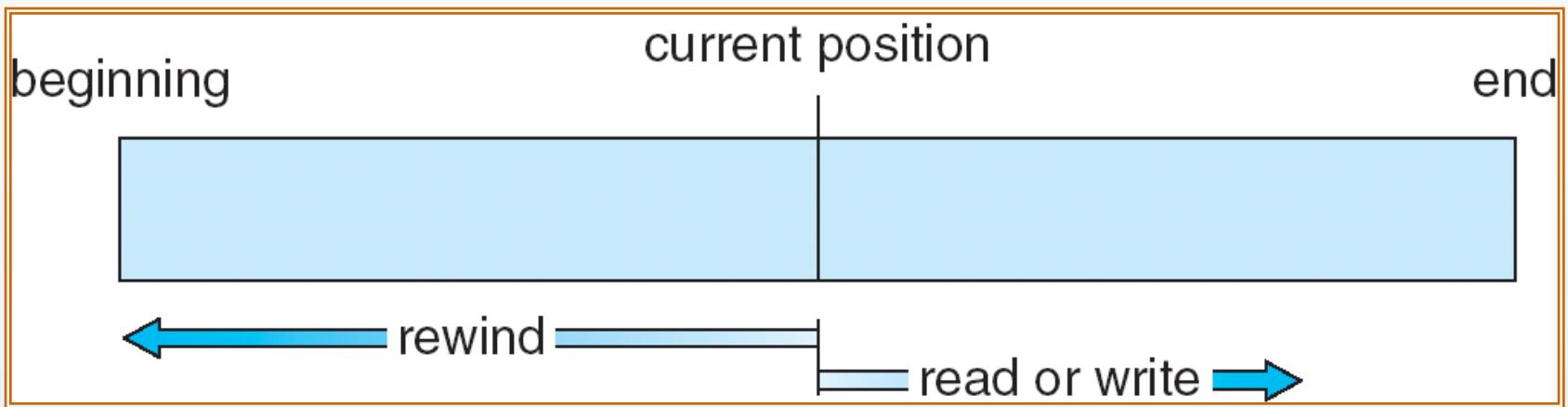
read n
write n
position to n
read next
write next
rewrite n

n = relative block number





Sequential-access File





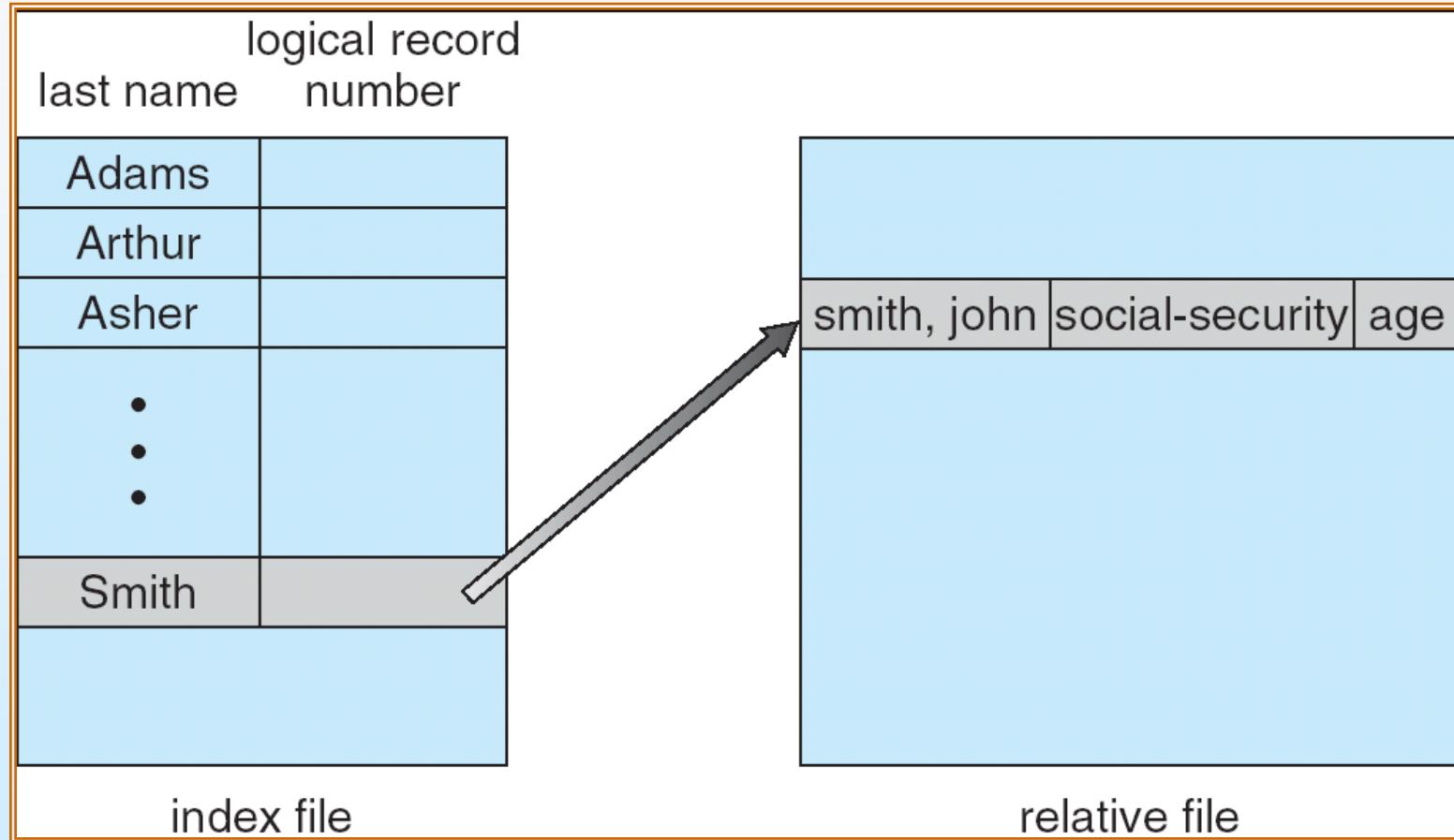
Simulation of Sequential Access on a Direct-access File

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> $cp = cp + 1;$





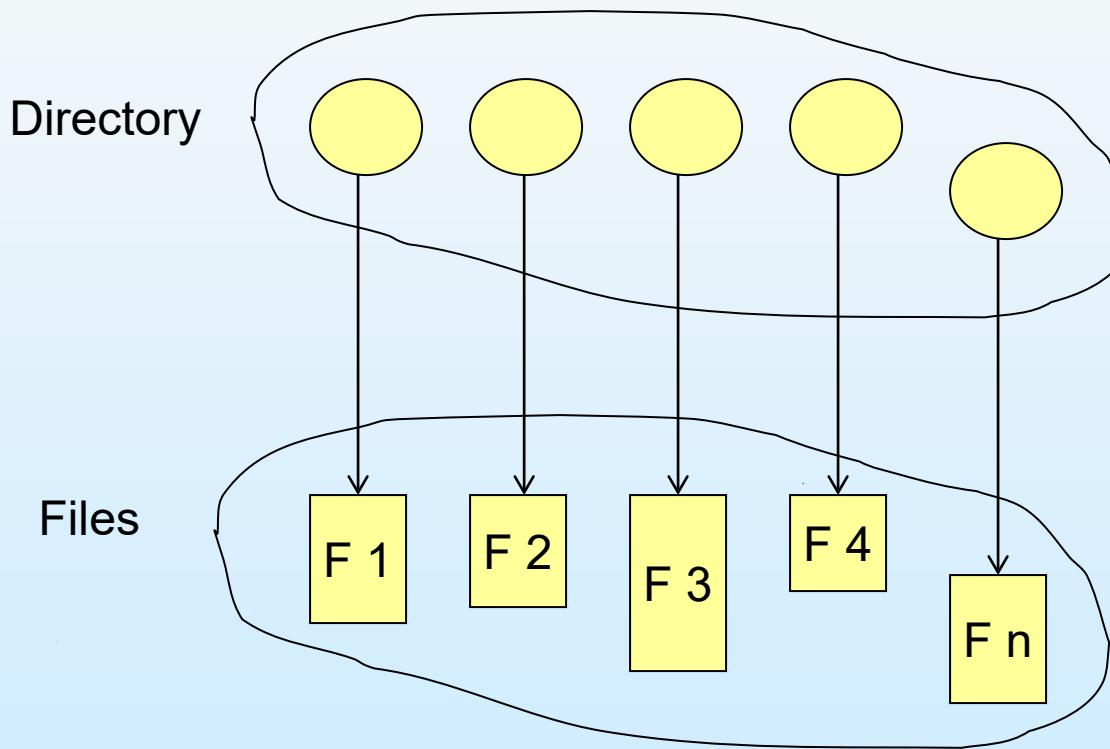
Example of Index and Relative Files





Directory Structure

- A collection of nodes containing information about all files

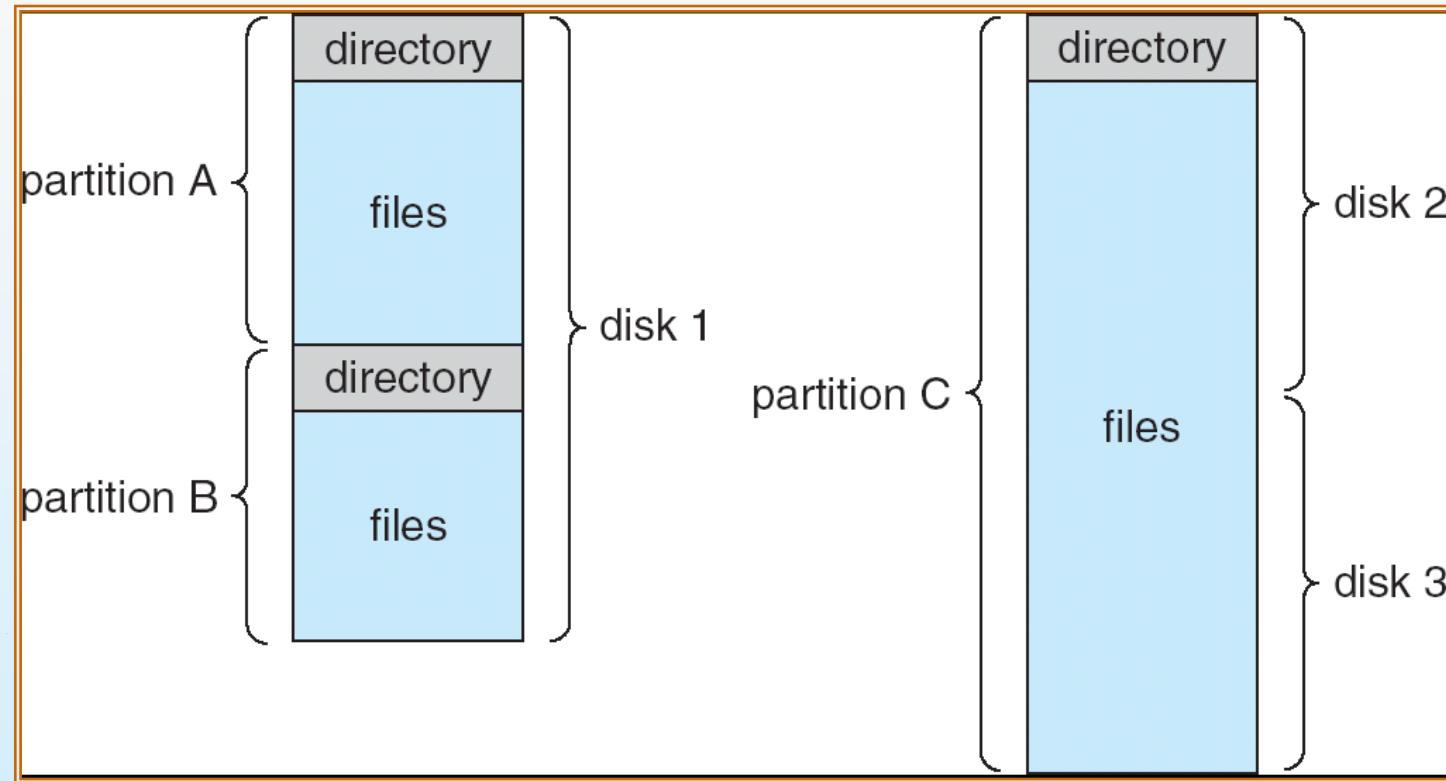


Both the directory structure and the files reside on disk
Backups of these two structures are kept on tapes





A Typical File-system Organization





Operations Performed on Directory

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system





Organize the Directory (Logically) to Obtain

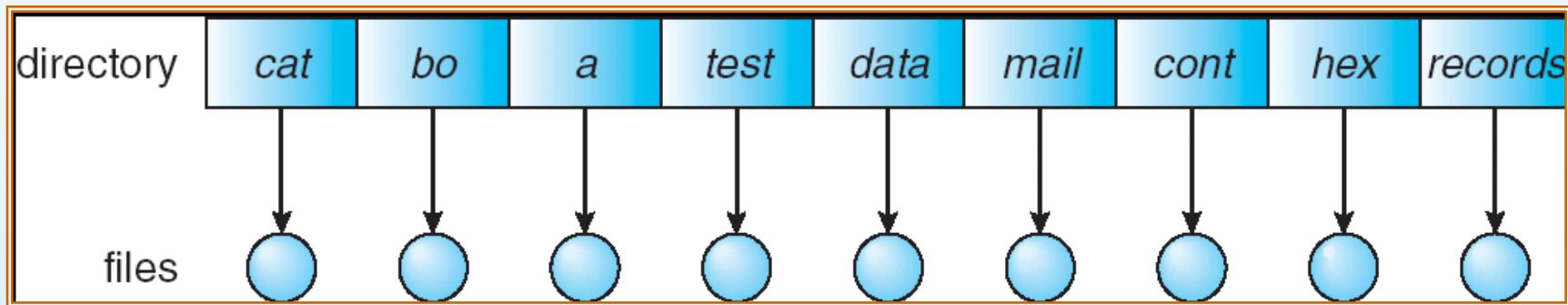
- Efficiency – locating a file quickly
- Naming – convenient to users
 - Two users can have same name for different files
 - The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, ...)





Single-Level Directory

- A single directory for all users



Naming problem

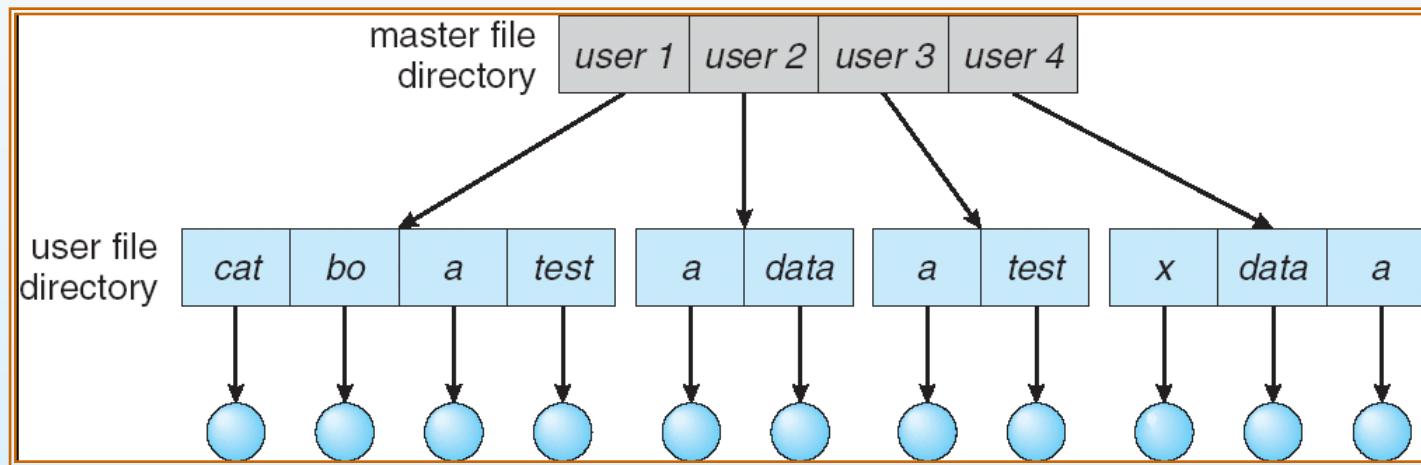
Grouping problem





Two-Level Directory

- Separate directory for each user

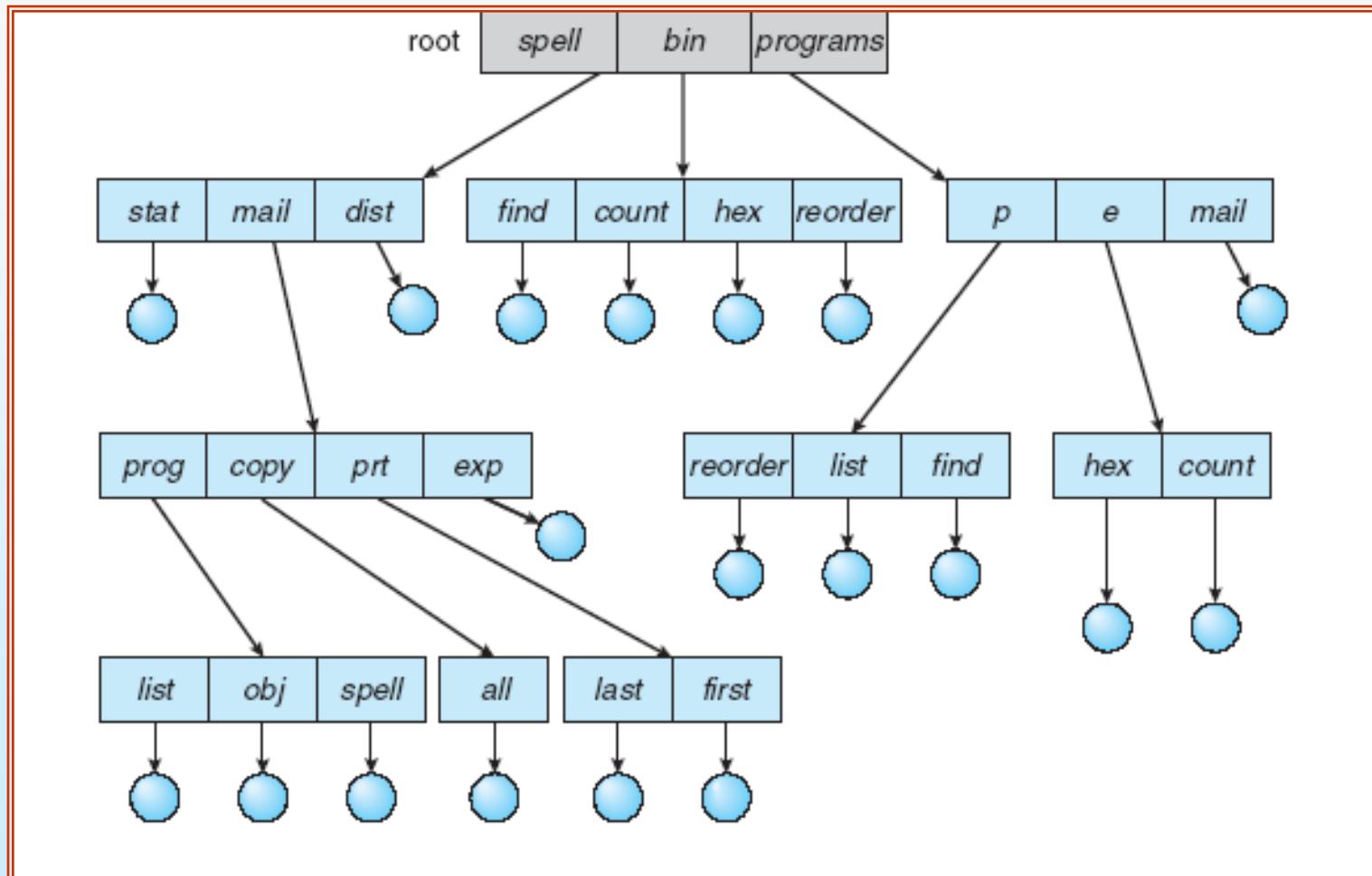


- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability





Tree-Structured Directories





Tree-Structured Directories (Cont)

- Efficient searching
- Grouping Capability
- Current directory (working directory)
 - `cd /spell/mail/prog`
 - `type list`





Tree-Structured Directories (Cont)

- **Absolute or relative** path name
- Creating a new file is done in current directory
- Delete a file

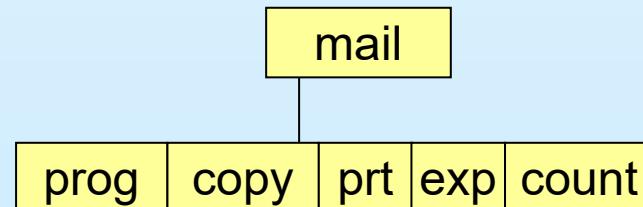
rm <file-name>

- Creating a new **subdirectory** is done in current directory

mkdir <dir-name>

Example: if in current directory /mail

mkdir count



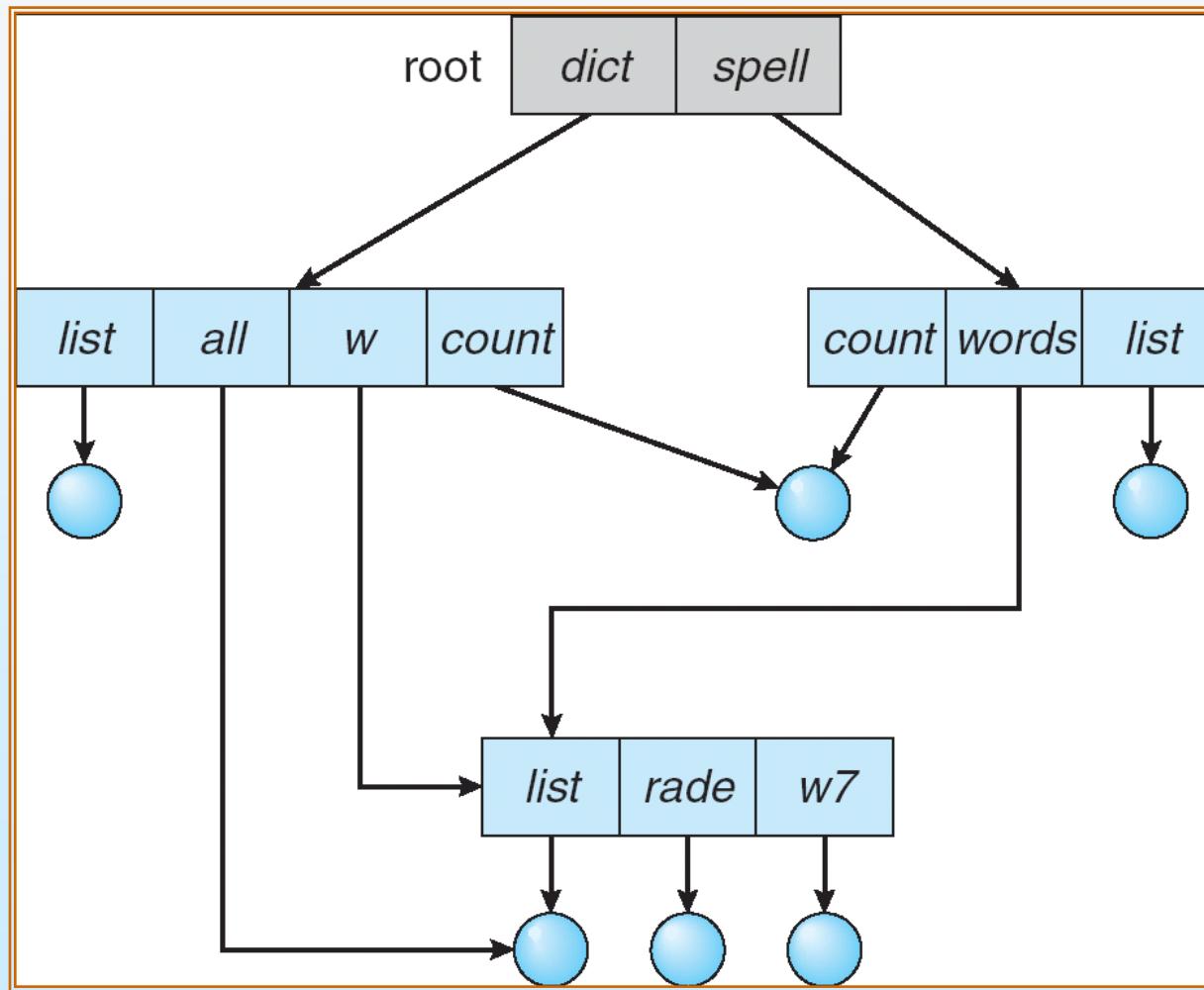
Deleting “mail” ⇒ deleting the entire subtree rooted by “mail”





Acyclic-Graph Directories

- Have shared subdirectories and files





Acyclic-Graph Directories (Cont.)

- Two different names (aliasing)
- If *dict* deletes *list* \Rightarrow dangling pointer

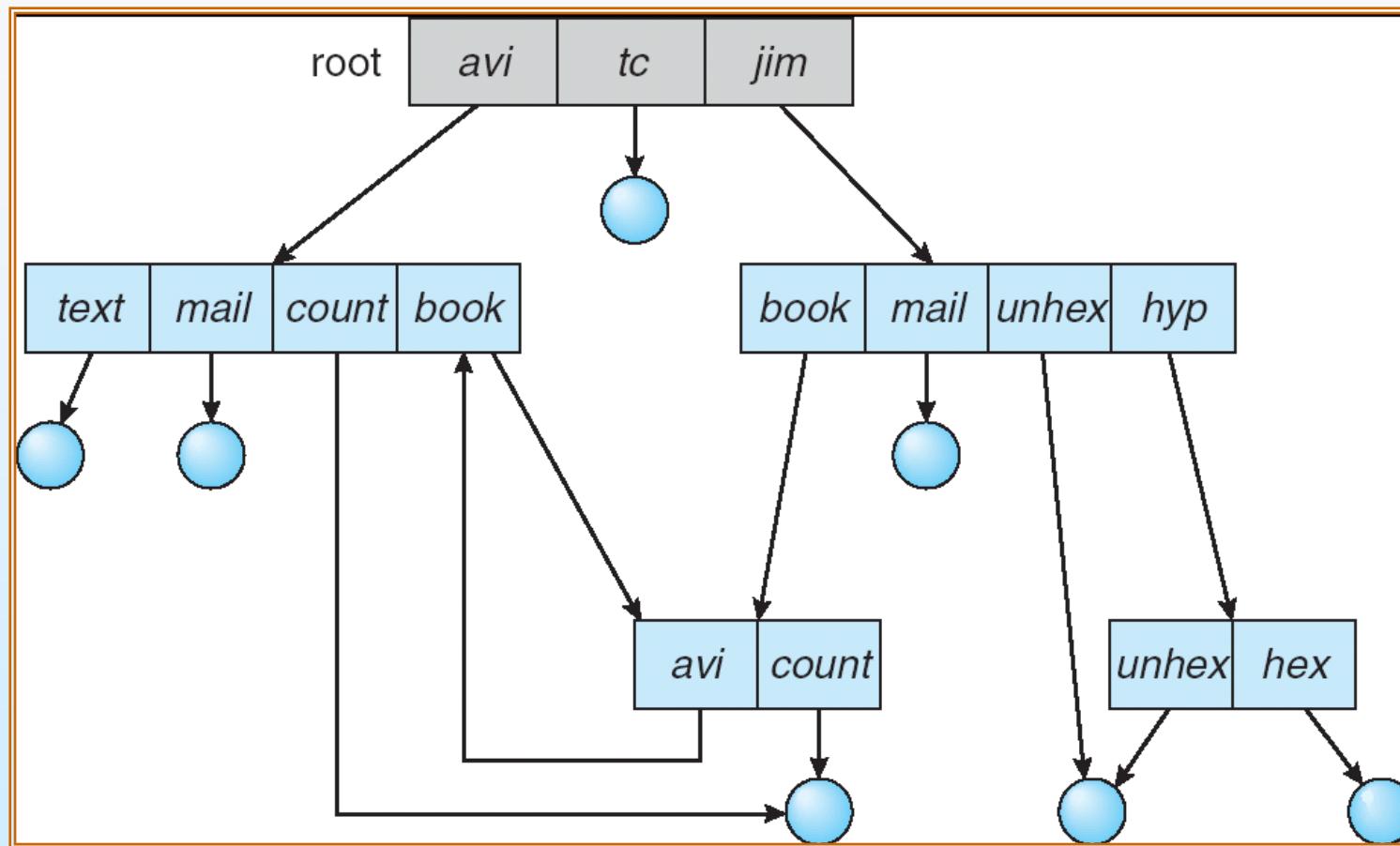
Solutions:

- 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 file not subdirectories
 - Garbage collection
 - Every time a new link is added use a cycle detection algorithm to determine whether it is OK





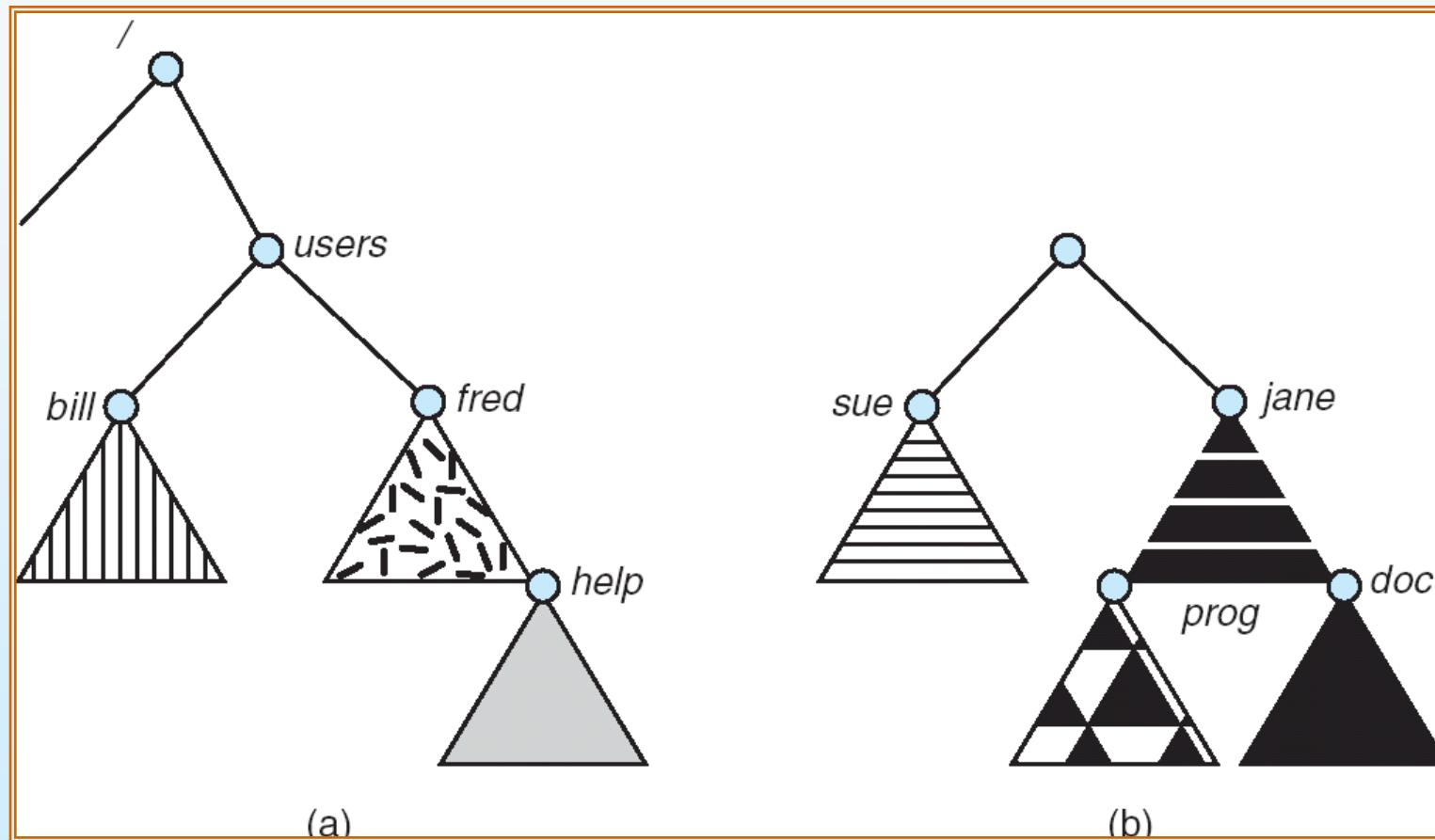
File System Mounting

- A file system must be **mounted** before it can be accessed
- A unmounted file system (i.e. Fig. 11-11(b)) is mounted at a **mount point**

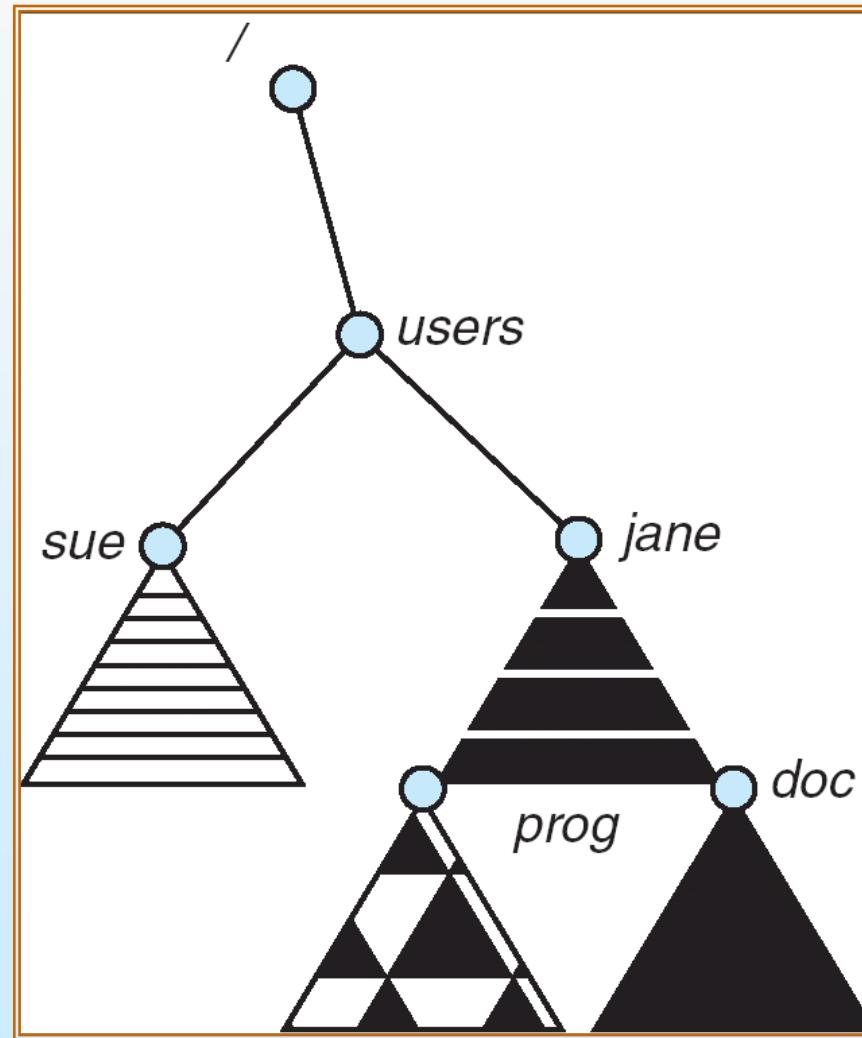




(a) Existing. (b) Unmounted Partition



Mount Point





File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a **protection** scheme
- On distributed systems, files may be shared across a network
- Network File System (NFS) is a common distributed file-sharing method





File Sharing – Multiple Users

- **User IDs** identify users, allowing permissions and protections to be per-user
- **Group IDs** allow users to be in groups, permitting group access rights





File Sharing – Remote File Systems

- Uses networking to allow file system access between systems
 - Manually via programs like **FTP**
 - Automatically, seamlessly using **distributed file systems**
 - Semi automatically via the **world wide web**
- **Client-server** model allows clients to mount remote file systems from servers
 - Server can serve multiple clients
 - Client and user-on-client identification is insecure or complicated
 - **NFS** is standard UNIX client-server file sharing protocol
 - **CIFS** is standard Windows protocol
 - Standard operating system file calls are translated into remote calls
- Distributed Information Systems (**distributed naming services**) such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing





File Sharing – Failure Modes

- Remote file systems add new failure modes, due to network failure, server failure
- Recovery from failure can involve state information about status of each remote request
- Stateless protocols such as NFS include all information in each request, allowing easy recovery but less security





File Sharing – Consistency Semantics

- **Consistency semantics** specify how multiple users are to access a shared file simultaneously
 - Similar to Ch 7 process synchronization algorithms
 - ▶ Tend to be less complex due to disk I/O and network latency (for remote file systems)
 - Andrew File System (AFS) implemented complex remote file sharing semantics
 - Unix file system (UFS) implements:
 - ▶ Writes to an open file visible immediately to other users of the same open file
 - ▶ Sharing file pointer to allow multiple users to read and write concurrently
 - AFS has session semantics
 - ▶ Writes only visible to sessions starting after the file is closed





Protection

- File owner/creator should be able to control:
 - what can be done
 - by whom
- Types of access
 - **Read**
 - **Write**
 - **Execute**
 - **Append**
 - **Delete**
 - **List**



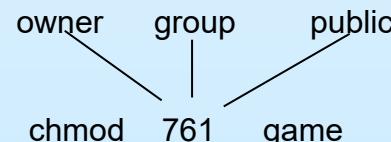


Access Lists and Groups

- Mode of access: read, write, execute
- Three classes of users

			RWX
a) owner access	7	⇒	1 1 1
b) group access	6	⇒	1 1 0
c) public access	1	⇒	0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.



Attach a group to a file

chgrp G game



Windows XP Access-control List Management

10.tex Properties

General Security Summary

Group or user names:

- Administrators (PBG-LAPTOP\Administrators)
- Guest (PBG-LAPTOP\Guest)**
- pbg (CT\pbg)
- SYSTEM
- Users (PBG-LAPTOP\Users)

Add... Remove

Permissions for Guest	Allow	Deny
Full Control	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Modify	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Read & Execute	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Read	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Write	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Special Permissions	<input type="checkbox"/>	<input type="checkbox"/>

For special permissions or for advanced settings, click Advanced.

Advanced

OK Cancel Apply



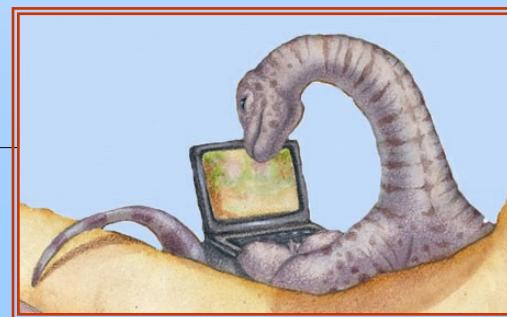


A Sample UNIX Directory Listing

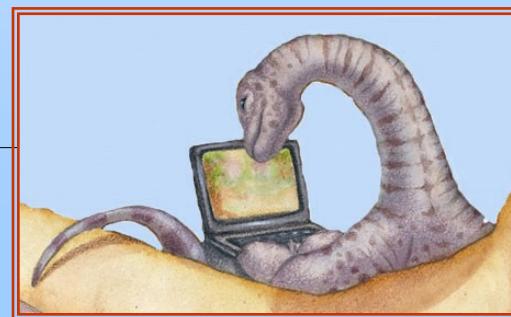
-rw-rw-r--	1	pbg	staff	31200	Sep 3 08:30	intro.ps
drwx-----	5	pbg	staff	512	Jul 8 09:33	private/
drwxrwxr-x	2	pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2	pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1	pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1	pbg	staff	20471	Feb 24 2003	program
drwx--x--x	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/



End of Chapter 10



Chapter 11: File System Implementation





Chapter 11: File System Implementation

- File-System Structure
- File-System Implementation
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery
- Log-Structured File Systems
- NFS
- Example: WAFL File System





Objectives

- To describe the details of implementing local file systems and directory structures
- To describe the implementation of remote file systems
- To discuss block allocation and free-block algorithms and trade-offs





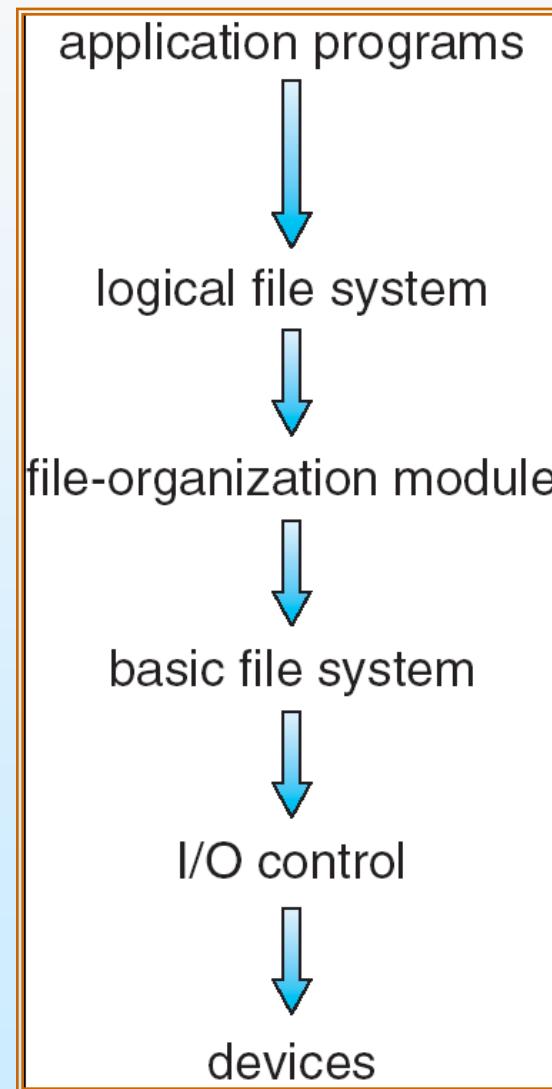
File-System Structure

- File structure
 - Logical storage unit
 - Collection of related information
- File system resides on secondary storage (disks)
- File system organized into layers
- **File control block** – storage structure consisting of information about a file





Layered File System





A Typical File Control Block

file permissions

file dates (create, access, write)

file owner, group, ACL

file size

file data blocks or pointers to file data blocks





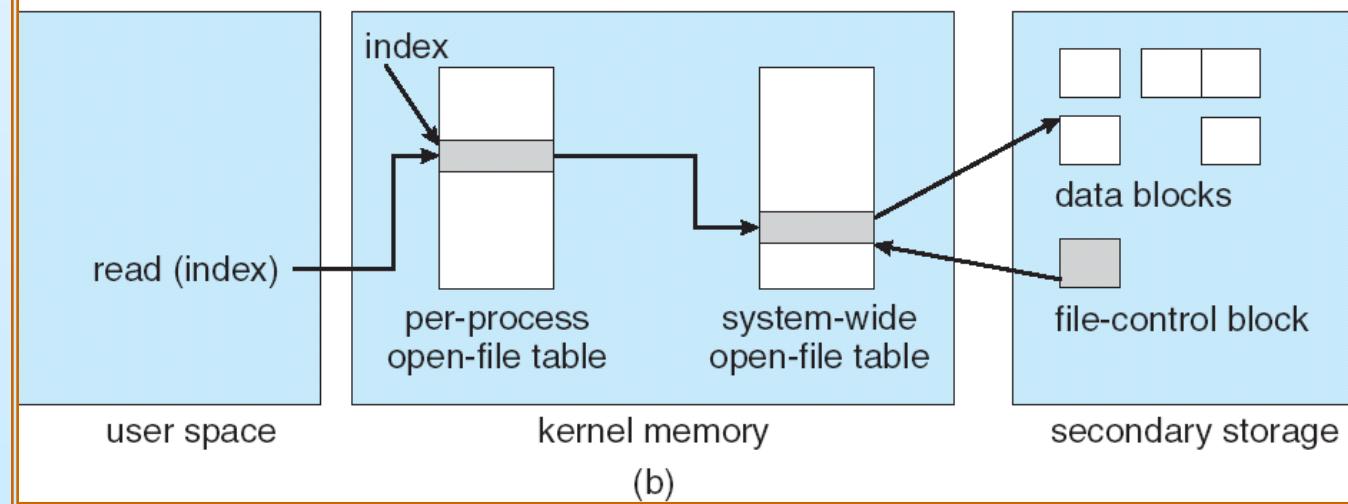
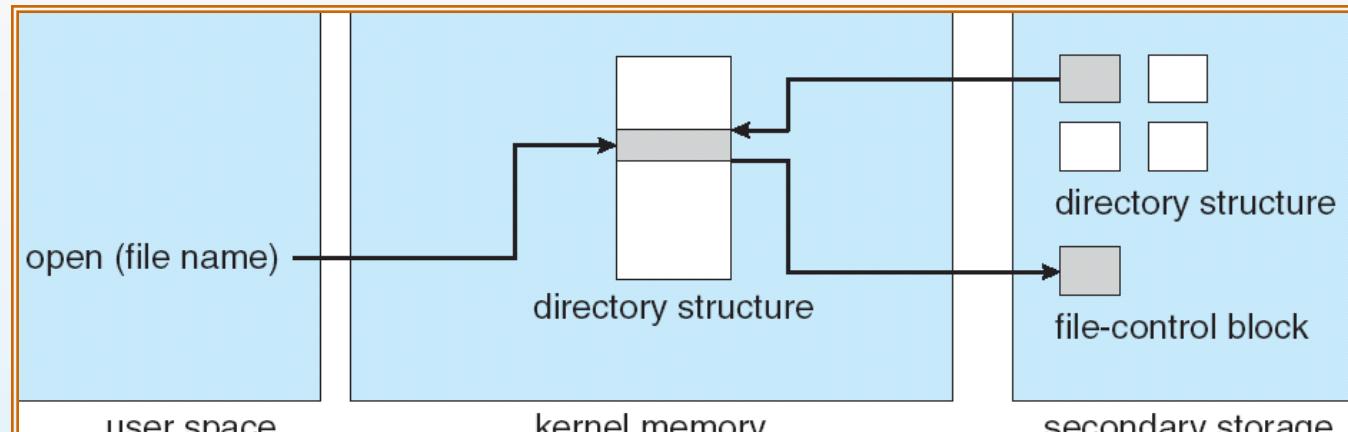
In-Memory File System Structures

- 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.





In-Memory File System Structures





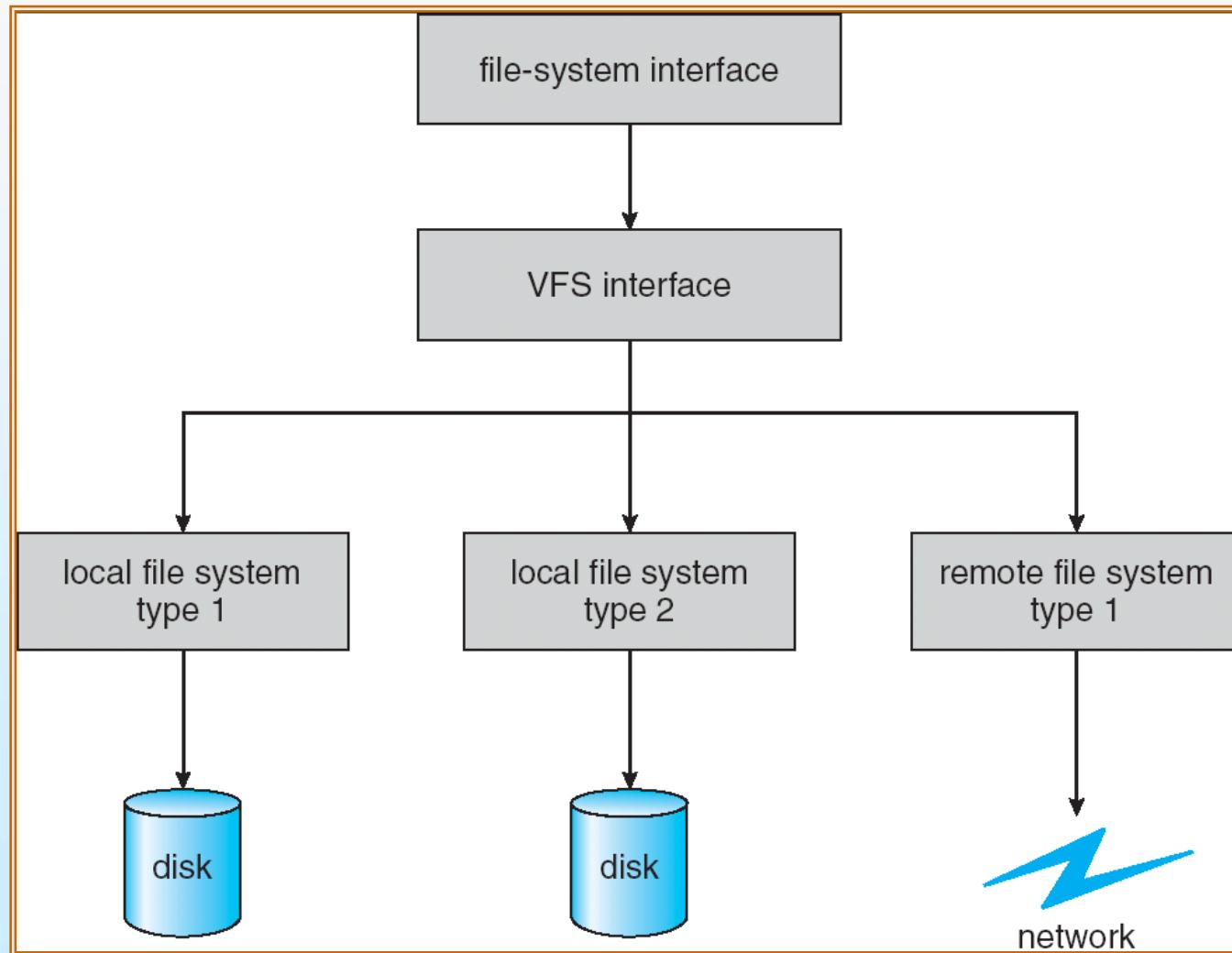
Virtual File Systems

- Virtual File Systems (VFS) 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.
- The API is to the VFS interface, rather than any specific type of file system.





Schematic View of Virtual File System





Directory Implementation

- **Linear list** of file names with pointer to the data blocks.
 - simple to program
 - time-consuming to execute
- **Hash Table** – linear list with hash data structure.
 - decreases directory search time
 - **collisions** – situations where two file names hash to the same location
 - fixed size





Allocation Methods

- An allocation method refers to how disk blocks are allocated for files:
- **Contiguous allocation**
- **Linked allocation**
- **Indexed allocation**





Contiguous Allocation

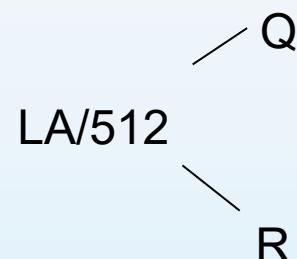
- Each file occupies a set of contiguous blocks on the disk
- Simple – only starting location (block #) and length (number of blocks) are required
- Random access
- Wasteful of space (dynamic storage-allocation problem)
- Files cannot grow





Contiguous Allocation

- Mapping from logical to physical

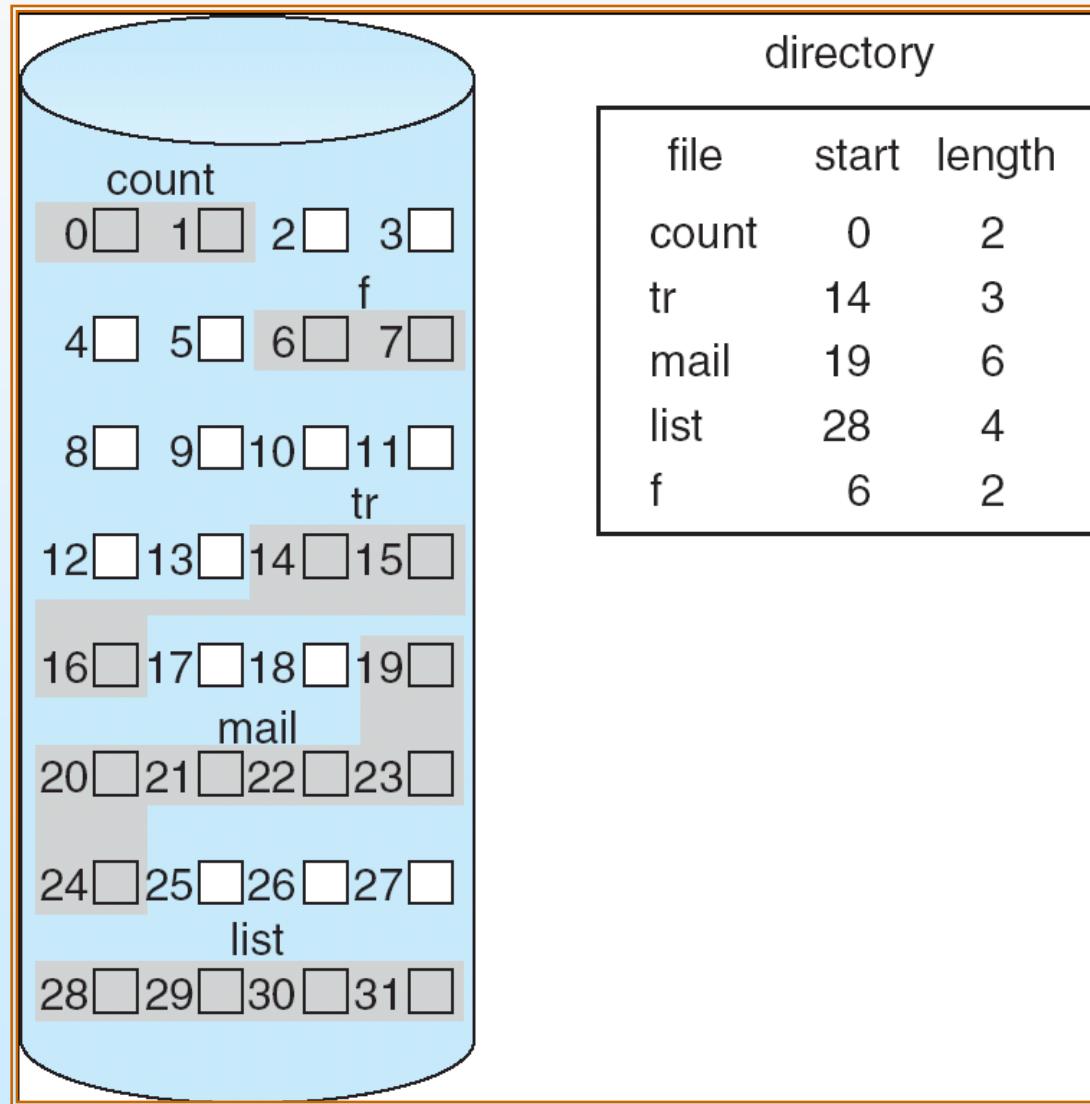


Block to be accessed = ! + starting address
Displacement into block = R





Contiguous Allocation of Disk Space





Extent-Based Systems

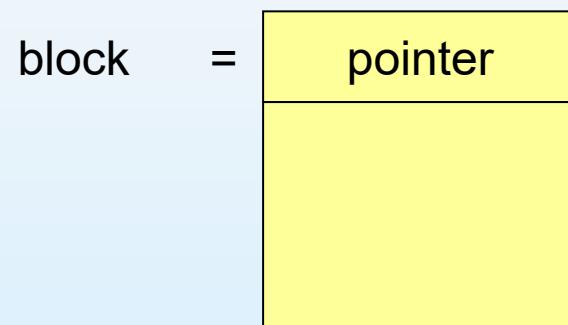
- Many newer file systems (I.e. Veritas File System) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in **extents**
- An **extent** is a contiguous block of disks
 - Extents are allocated for file allocation
 - A file consists of one or more extents.





Linked Allocation

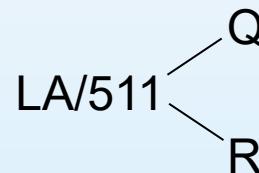
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.





Linked Allocation (Cont.)

- Simple – need only starting address
- Free-space management system – no waste of space
- No random access
- Mapping



Block to be accessed is the Q th block in the linked chain of blocks representing the file.

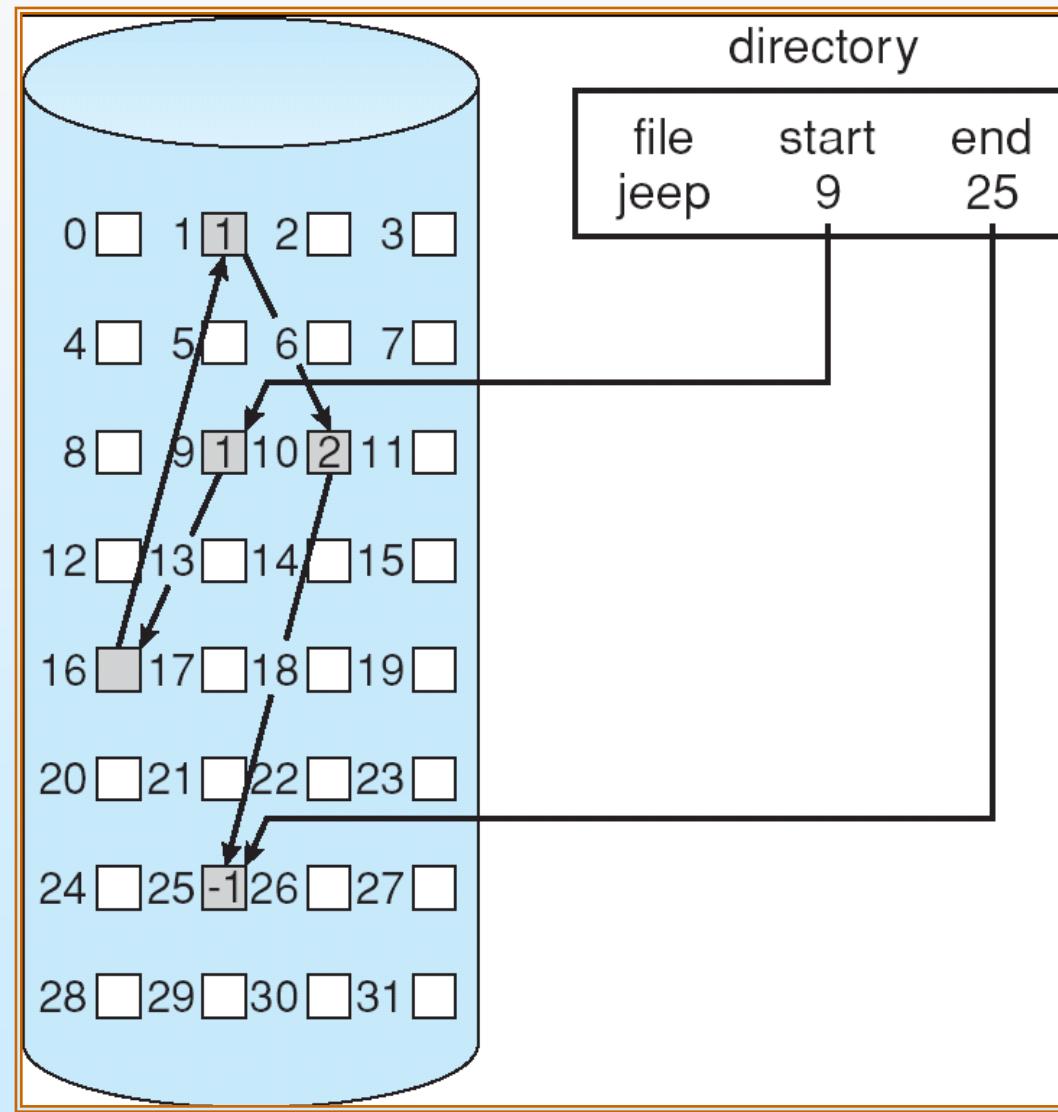
Displacement into block = $R + 1$

File-allocation table (FAT) – disk-space allocation used by MS-DOS and OS/2.



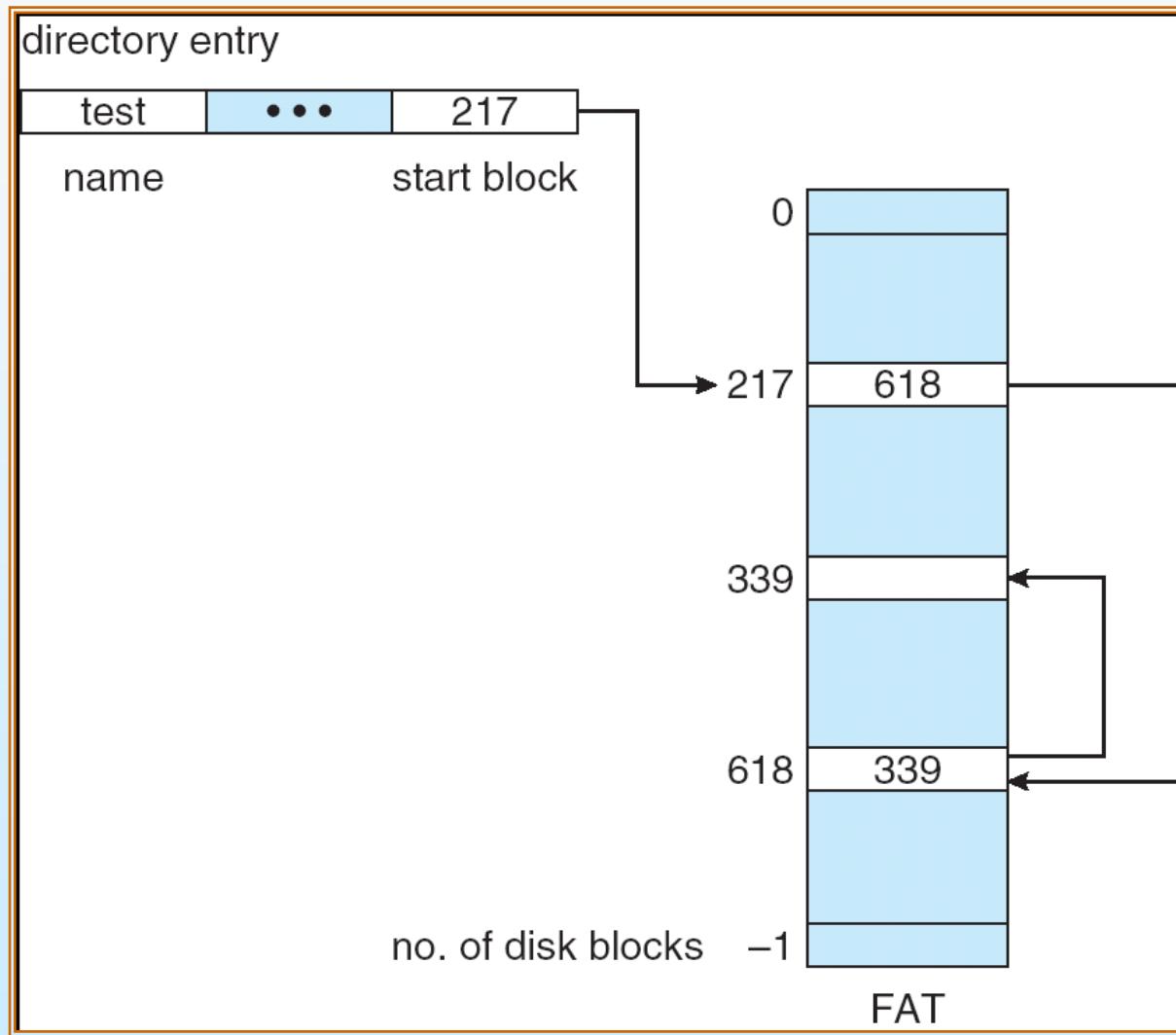


Linked Allocation





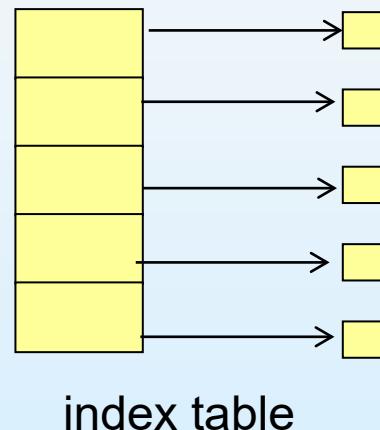
File-Allocation Table





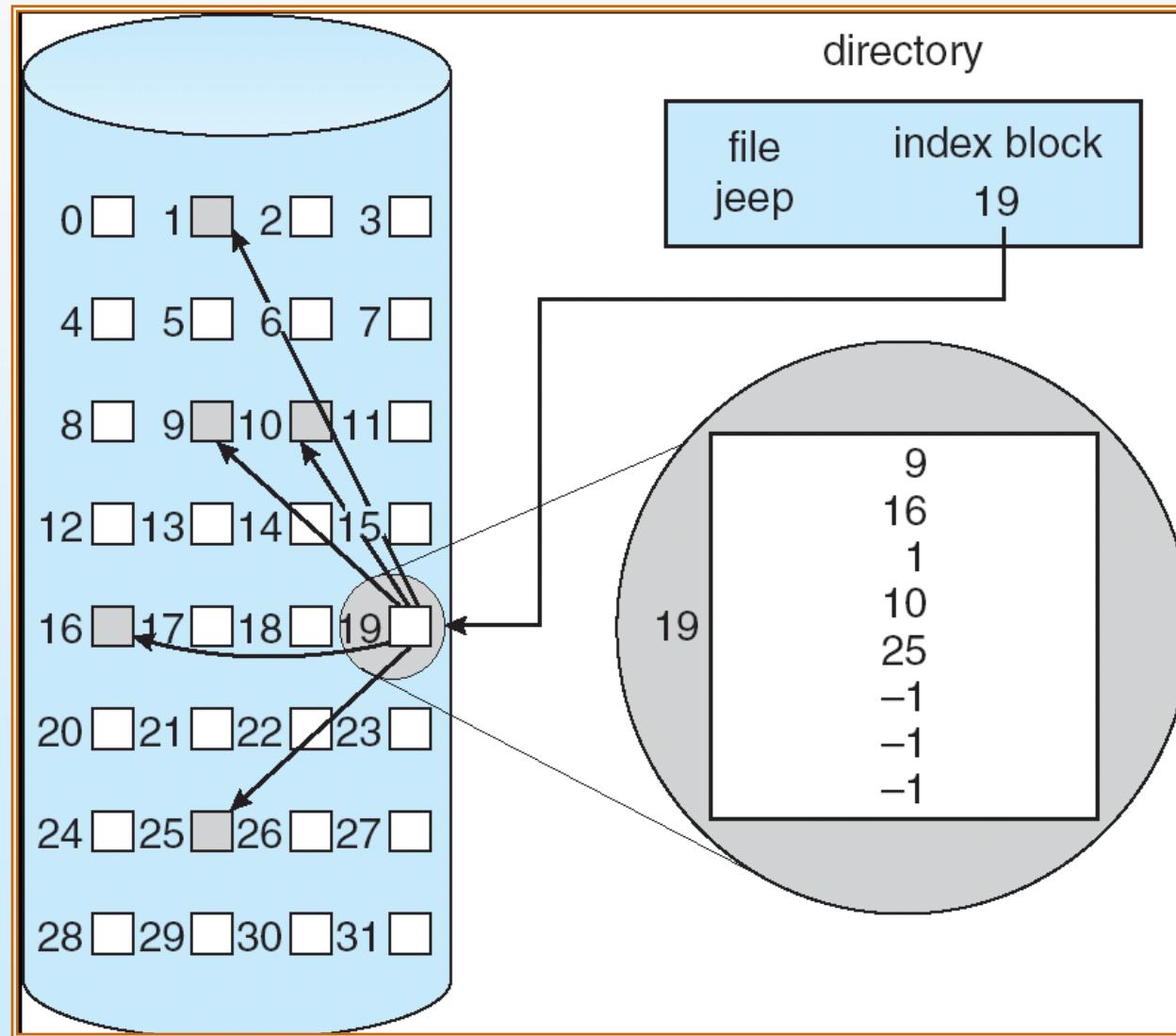
Indexed Allocation

- Brings all pointers together into the *index block*.
- Logical view.





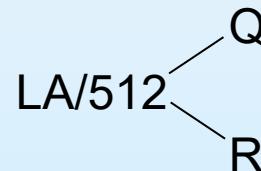
Example of Indexed Allocation





Indexed Allocation (Cont.)

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block.
- Mapping from logical to physical in a file of maximum size of 256K words and block size of 512 words. We need only 1 block for index table.



Q = displacement into index table

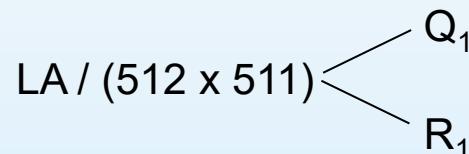
R = displacement into block





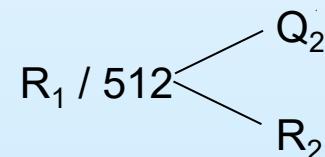
Indexed Allocation – Mapping (Cont.)

- Mapping from logical to physical in a file of unbounded length (block size of 512 words).
- Linked scheme – Link blocks of index table (no limit on size).



Q_1 = block of index table

R_1 is used as follows:



Q_2 = displacement into block of index table

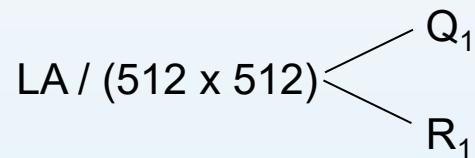
R_2 displacement into block of file:





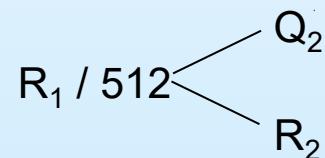
Indexed Allocation – Mapping (Cont.)

- Two-level index (maximum file size is 512^3)



Q_1 = displacement into outer-index

R_1 is used as follows:



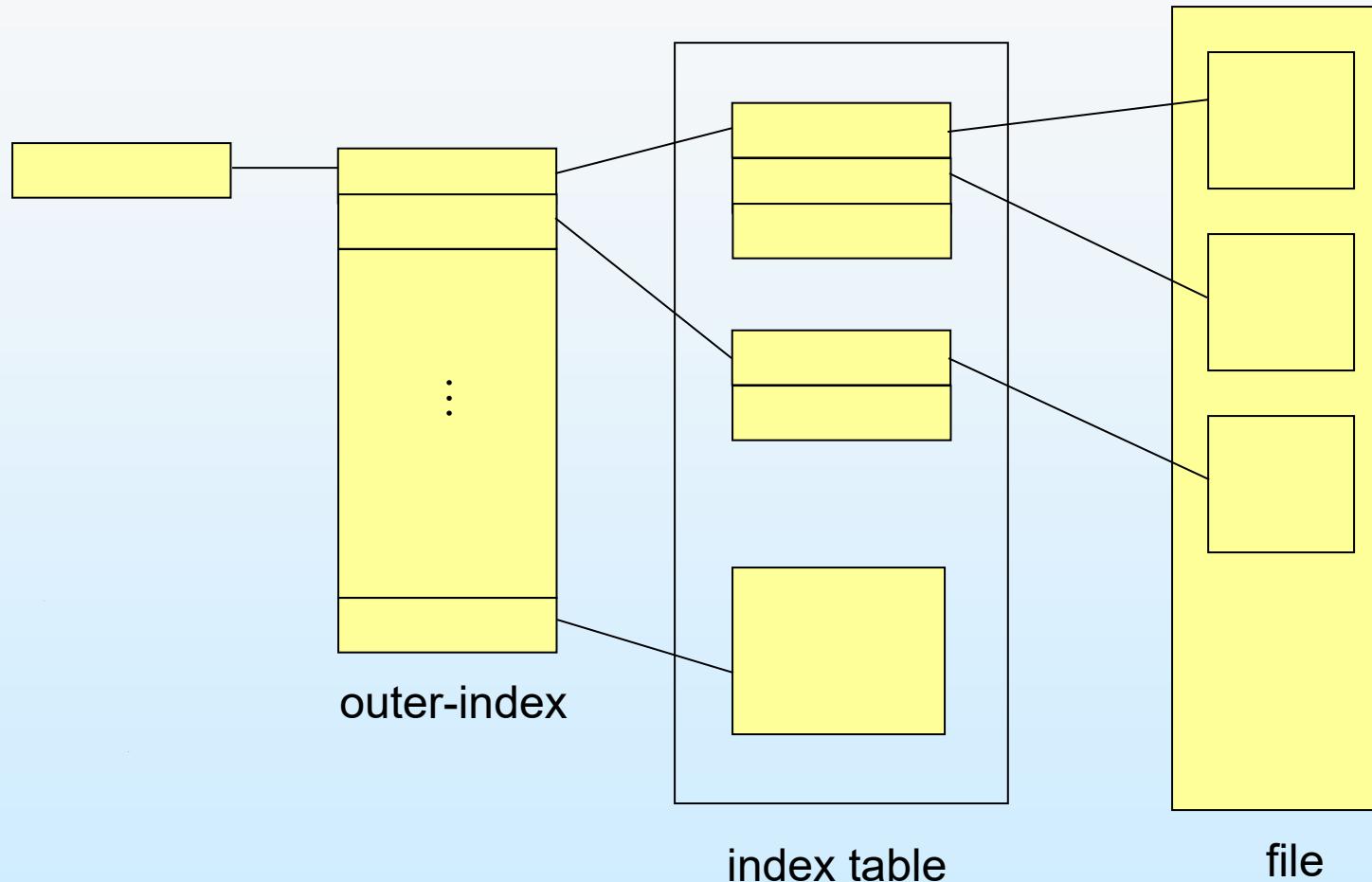
Q_2 = displacement into block of index table

R_2 displacement into block of file:



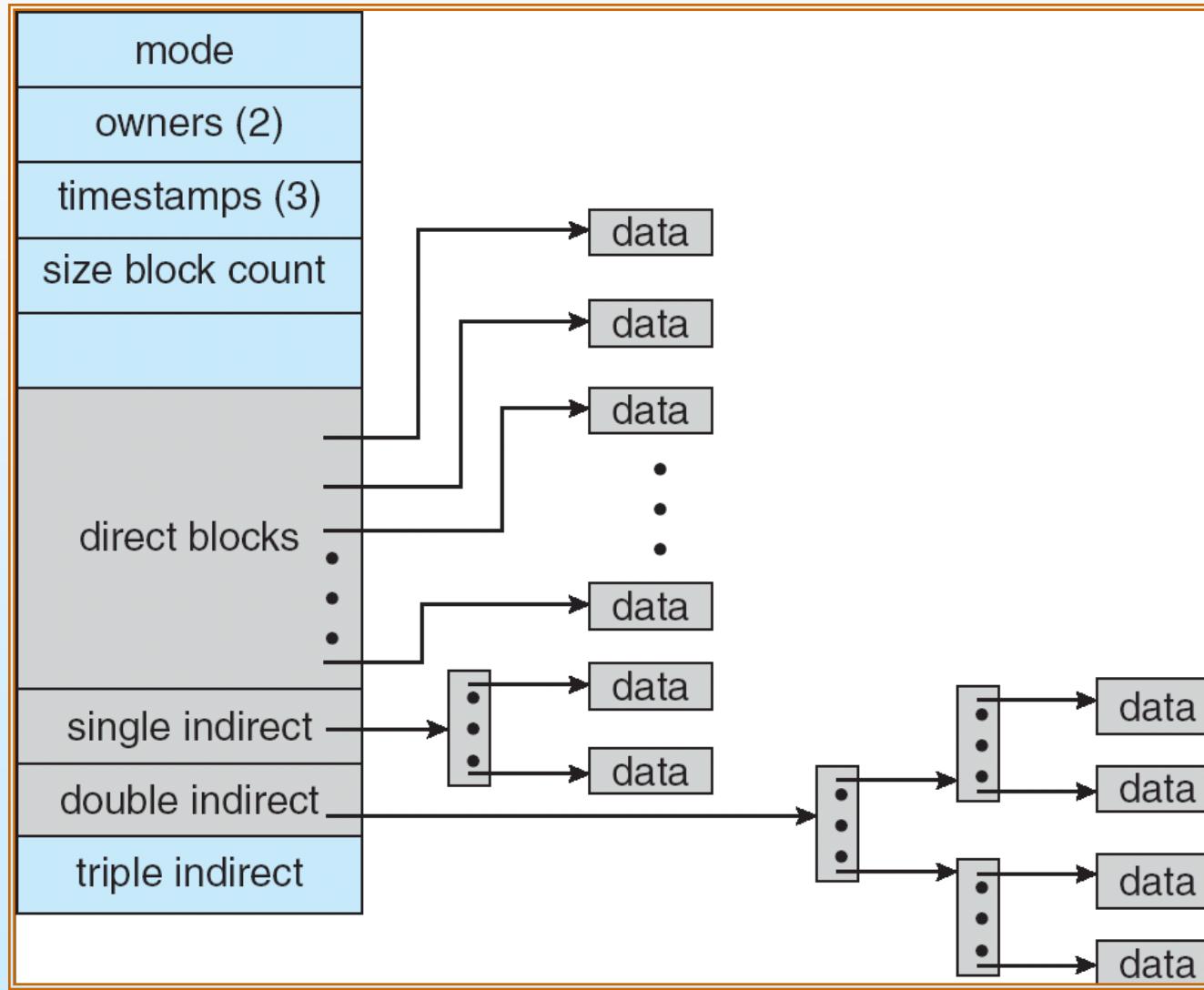


Indexed Allocation – Mapping (Cont.)





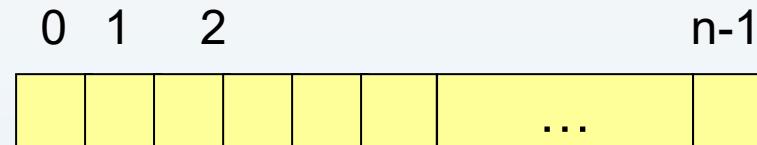
Combined Scheme: UNIX (4K bytes per block)





Free-Space Management

- Bit vector (n blocks)



$$\text{bit}[i] = \begin{cases} 0 & \Rightarrow \text{block}[i] \text{ free} \\ 1 & \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

Block number calculation

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





Free-Space Management (Cont.)

- Bit map requires extra space
 - Example:
 - block size = 2^{12} bytes
 - disk size = 2^{30} bytes (1 gigabyte)
 - $n = 2^{30}/2^{12} = 2^{18}$ bits (or 32K bytes)
- Easy to get contiguous files
- Linked list (free list)
 - Cannot get contiguous space easily
 - No waste of space
- Grouping
- Counting





Free-Space Management (Cont.)

- Need to protect:
 - Pointer to free list
 - Bit map
 - ▶ Must be kept on disk
 - ▶ Copy in memory and disk may differ
 - ▶ Cannot allow for $\text{block}[i]$ to have a situation where $\text{bit}[i] = 1$ in memory and $\text{bit}[i] = 0$ on disk
 - Solution:
 - ▶ Set $\text{bit}[i] = 1$ in disk
 - ▶ Allocate $\text{block}[i]$
 - ▶ Set $\text{bit}[i] = 1$ in memory





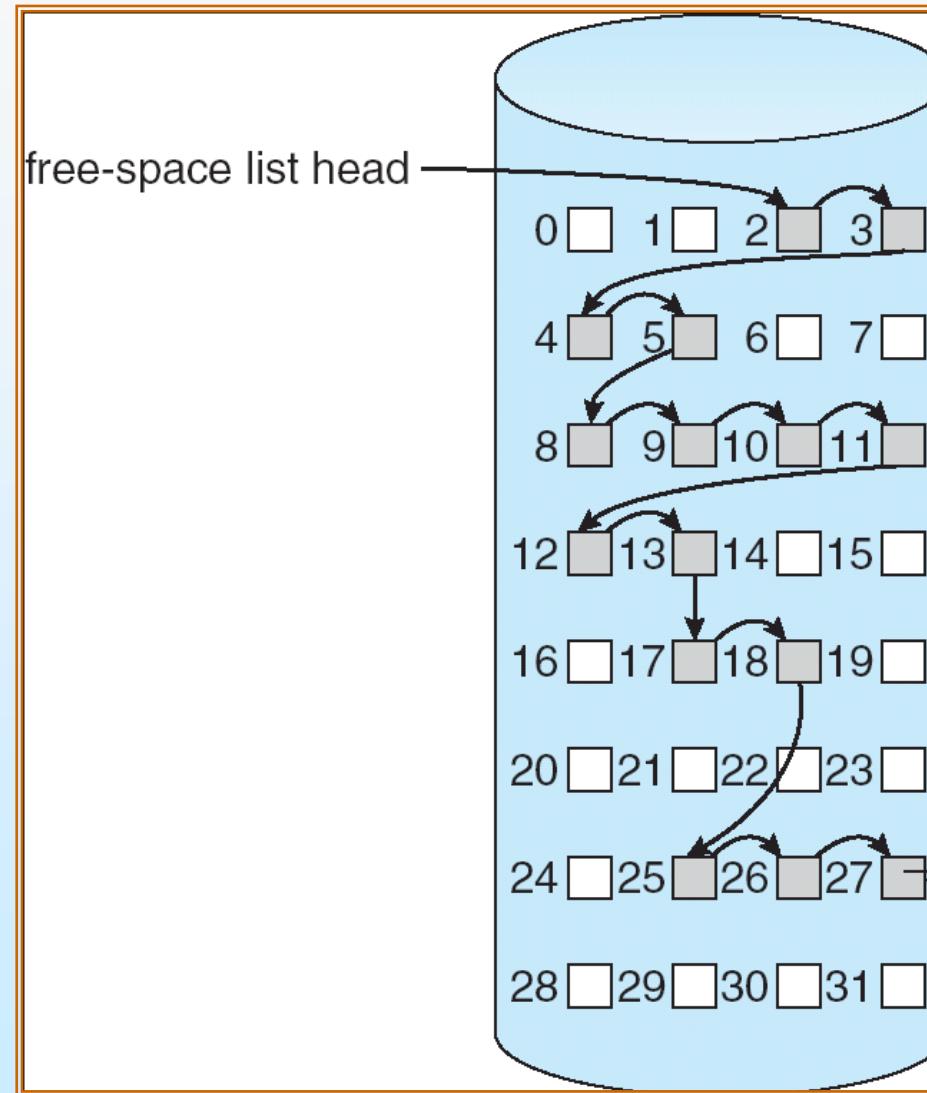
Directory Implementation

- Linear list of file names with pointer to the data blocks
 - simple to program
 - time-consuming to execute
- Hash Table – linear list with hash data structure
 - decreases directory search time
 - **collisions** – situations where two file names hash to the same location
 - fixed size





Linked Free Space List on Disk





Efficiency and Performance

- Efficiency dependent on:
 - disk allocation and directory algorithms
 - types of data kept in file's directory entry
- Performance
 - disk cache – separate section of main memory for frequently used blocks
 - free-behind and read-ahead – techniques to optimize sequential access
 - improve PC performance by dedicating section of memory as virtual disk, or RAM disk





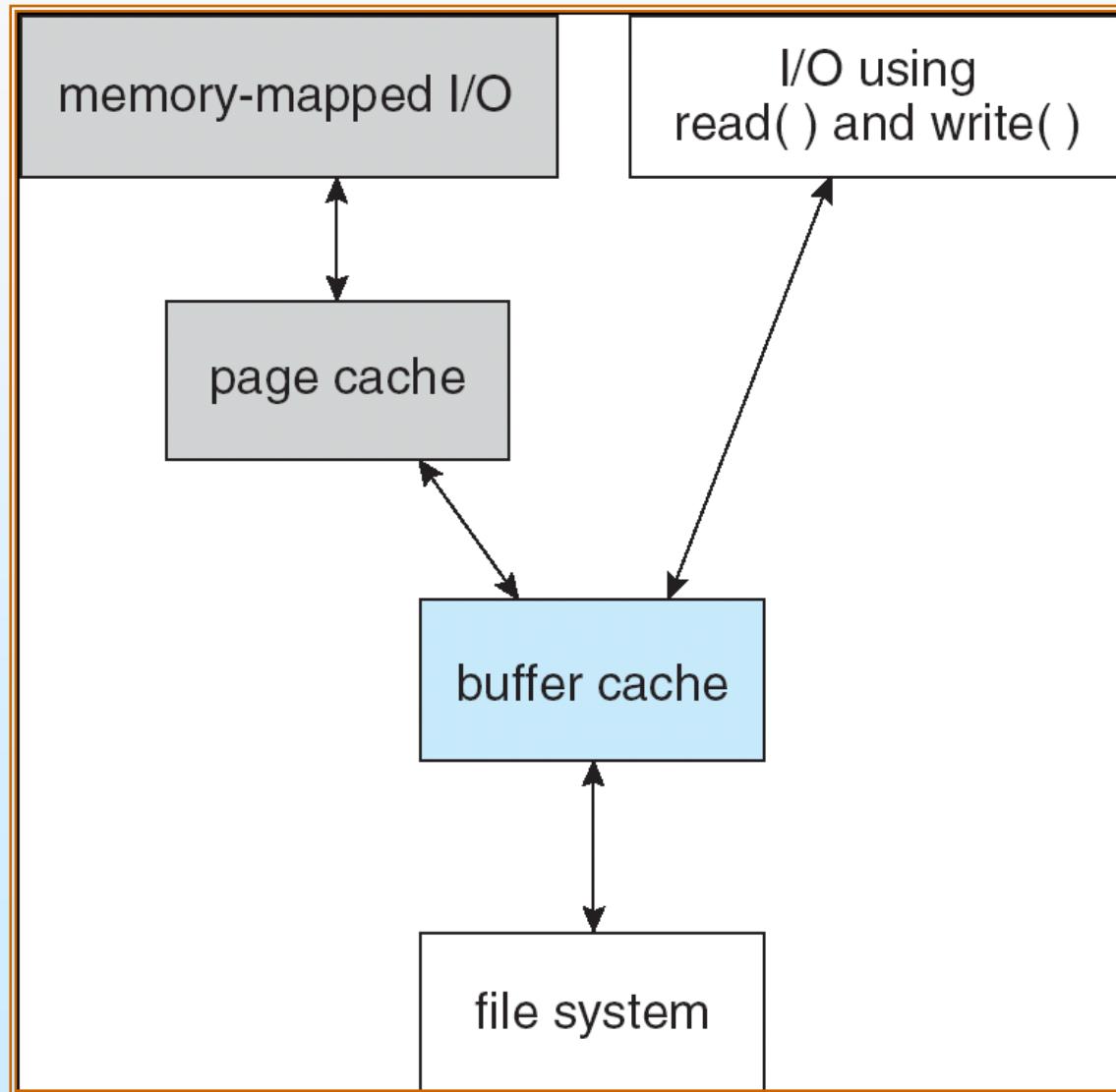
Page Cache

- A **page cache** caches pages rather than disk blocks using virtual memory techniques
- 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





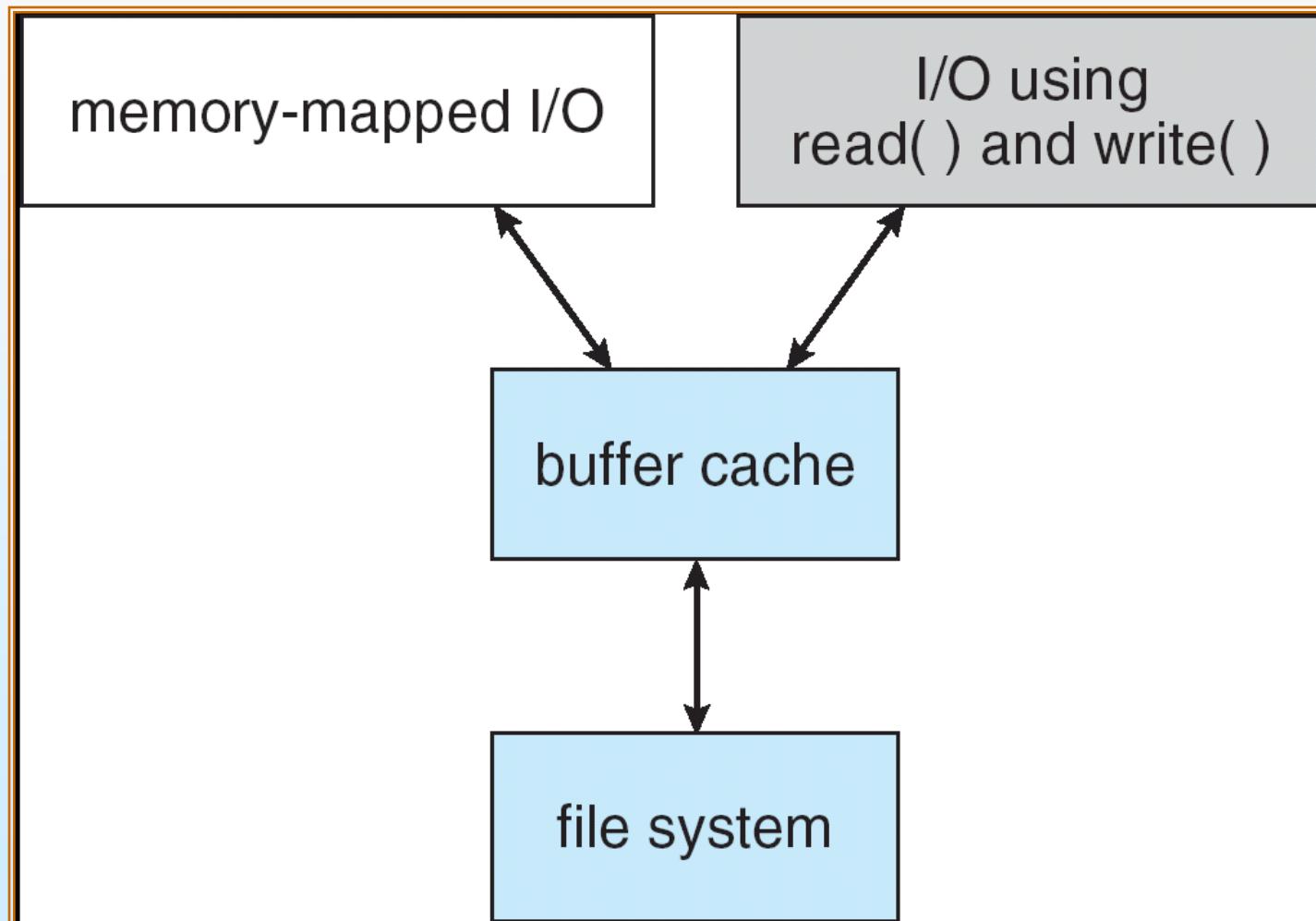
Unified Buffer Cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O





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
- Use system programs to **back up** data from disk to another storage device (floppy disk, 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 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
 - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system
 - When the file system is modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed





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 is part of the Solaris and SunOS operating systems running on Sun workstations using an unreliable datagram protocol (UDP/IP protocol and Ethernet)





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





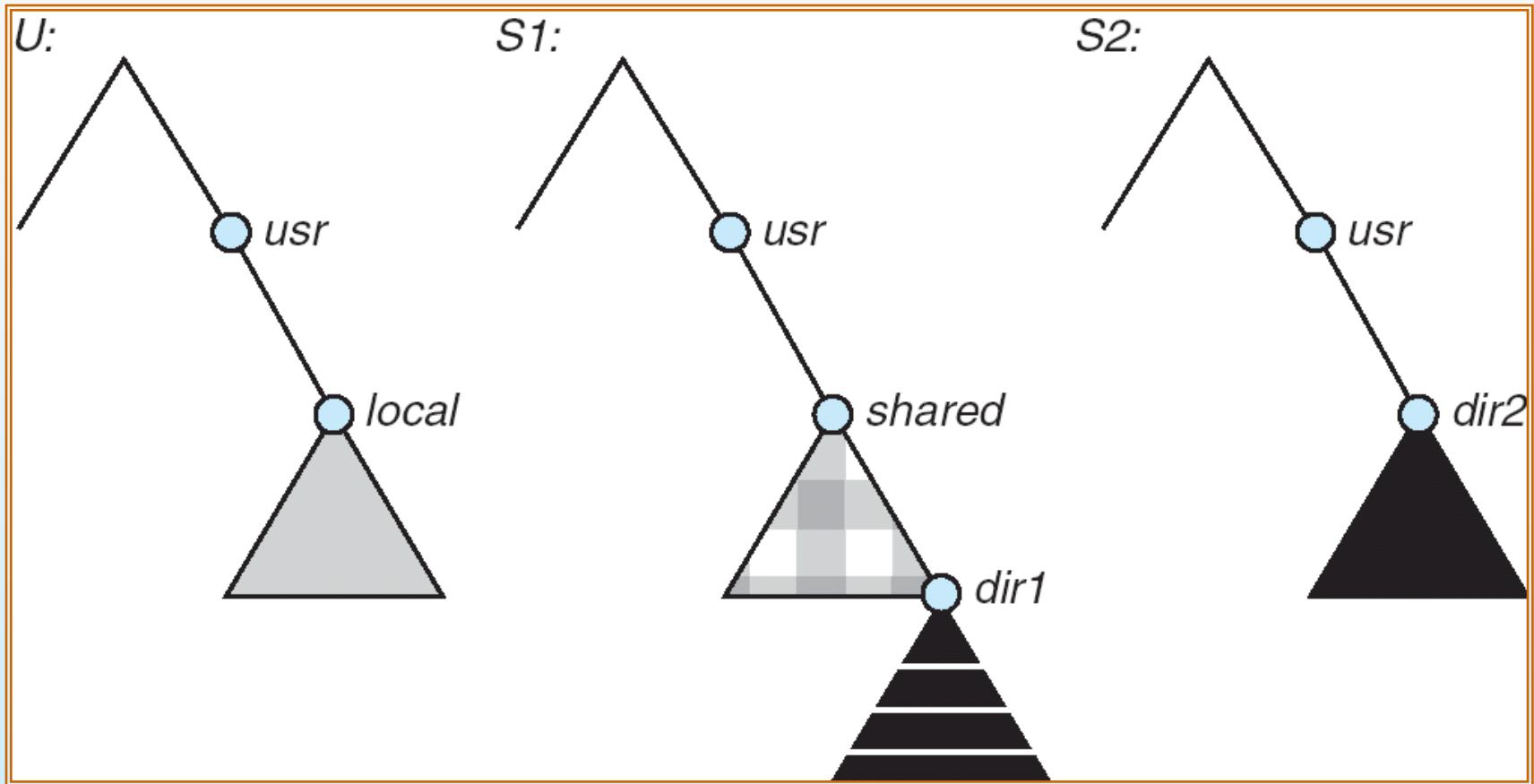
NFS (Cont.)

- 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





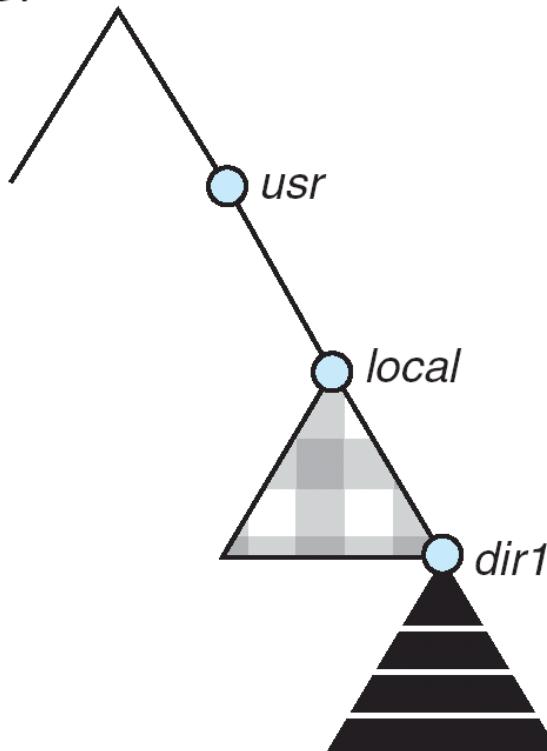
Three Independent File Systems





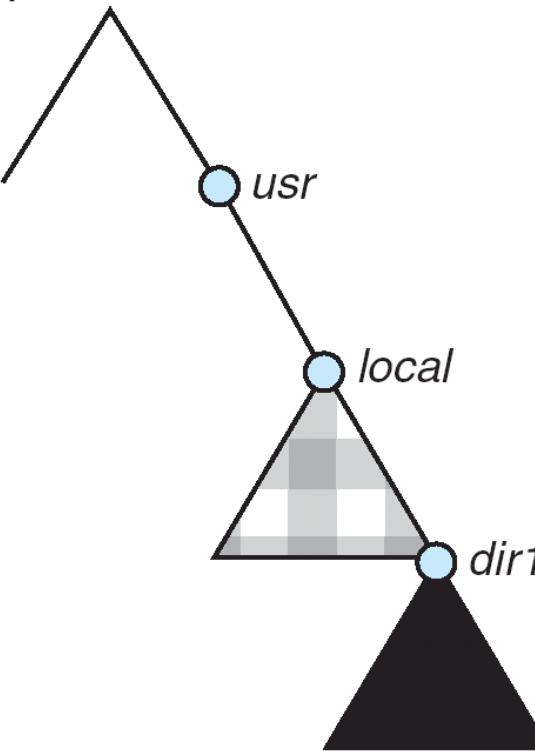
Mounting in NFS

U:



(a)

U:



(b)

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 just coming available – 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





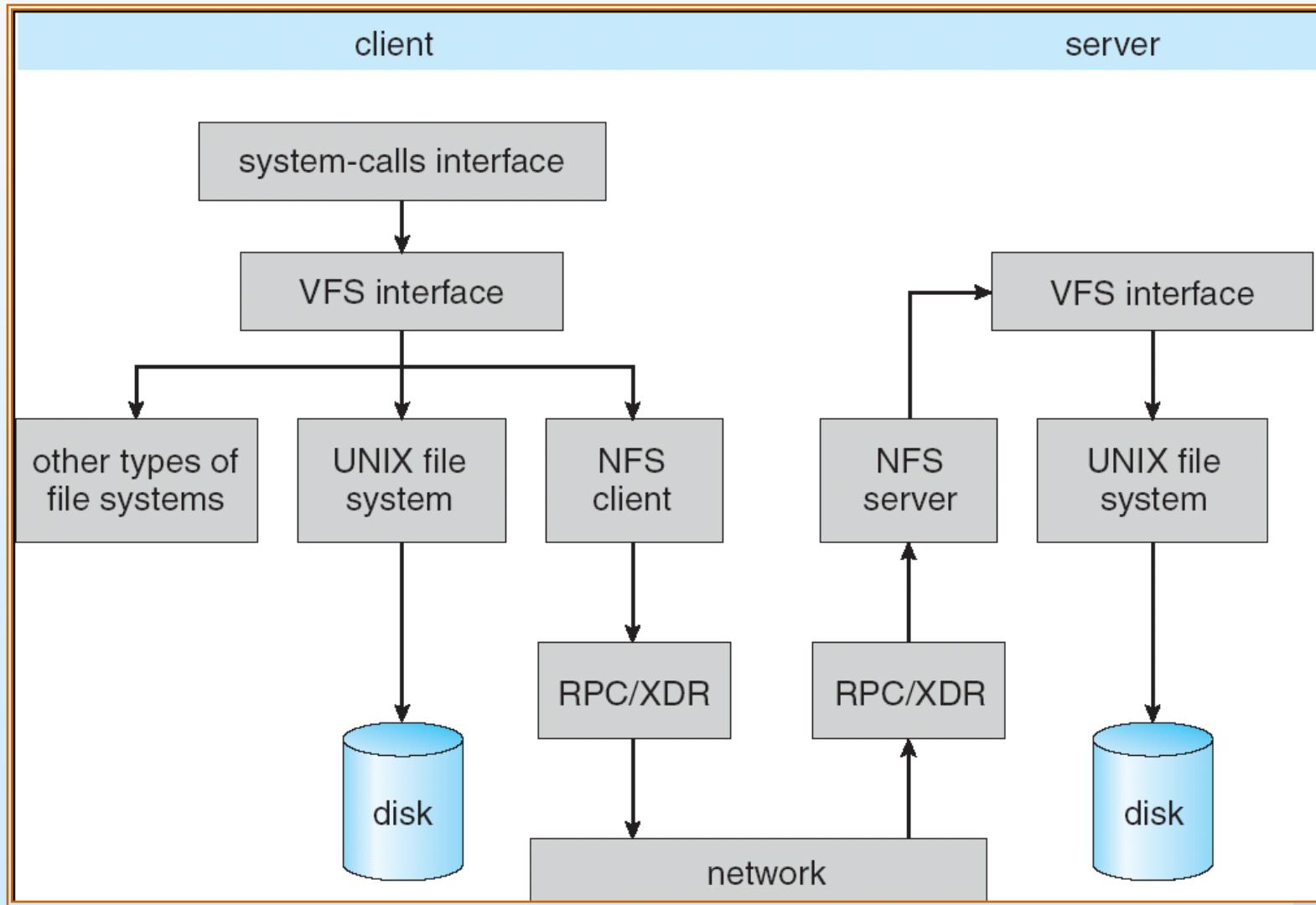
Three Major Layers of NFS Architecture

- 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





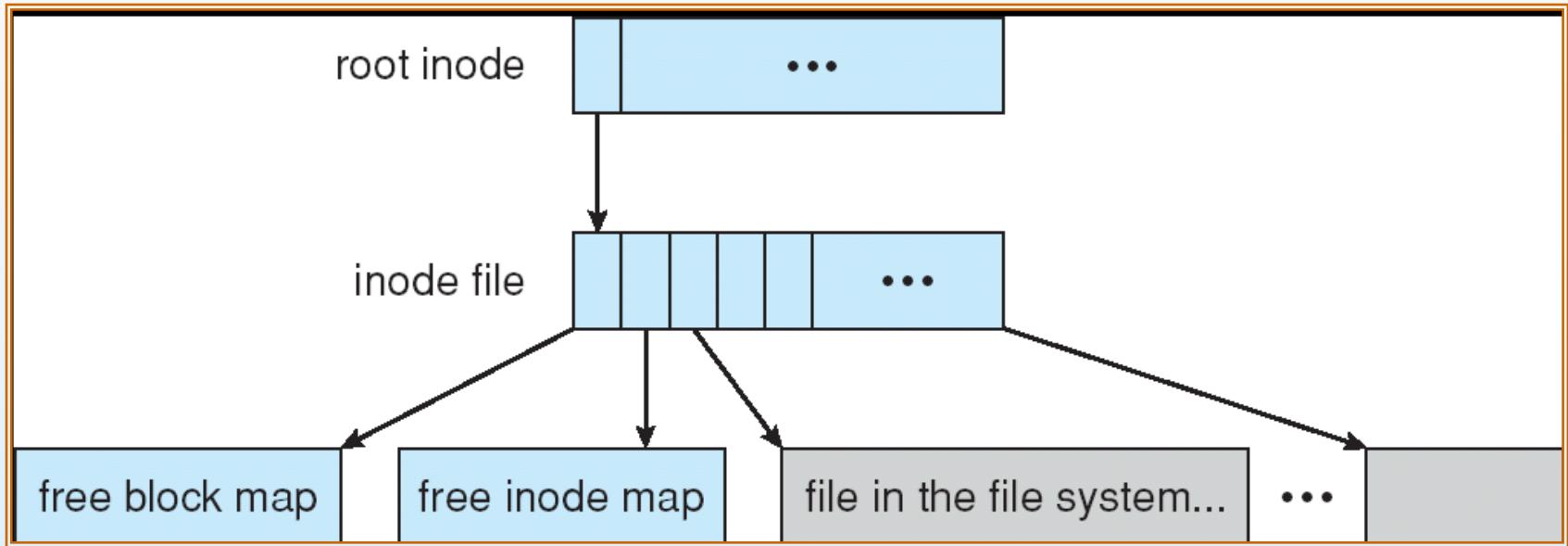
Example: WAFL File System

- 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



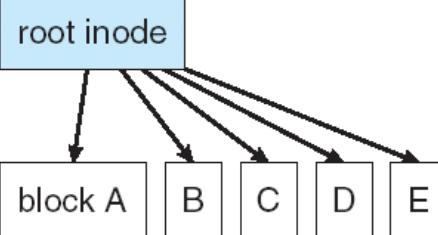


The WAFL File Layout

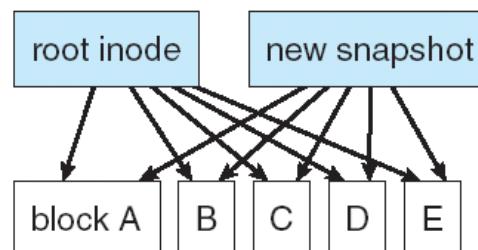




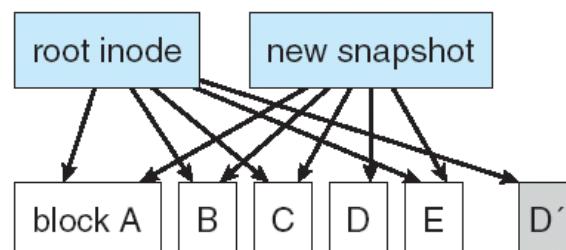
Snapshots in WAFL



(a) Before a snapshot.



(b) After a snapshot, before any blocks change



(c) After block D has changed to D'.





11.02

file permissions

file dates (create, access, write)

file owner, group, ACL

file size

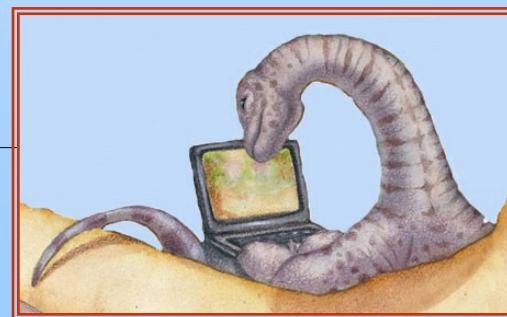
file data blocks or pointers to file data blocks



End of Chapter 11



Chapter 12: Mass-Storage Systems





Chapter 12: Mass-Storage Systems

- Overview of Mass Storage Structure
- Disk Structure
- Disk Attachment
- Disk Scheduling
- Disk Management
- Swap-Space Management
- RAID Structure
- Disk Attachment
- Stable-Storage Implementation
- Tertiary Storage Devices
- Operating System Issues
- Performance Issues





Objectives

- Describe the physical structure of secondary and tertiary storage devices and the resulting effects on the uses of the devices
- Explain the performance characteristics of mass-storage devices
- Discuss operating-system services provided for mass storage, including RAID and HSM





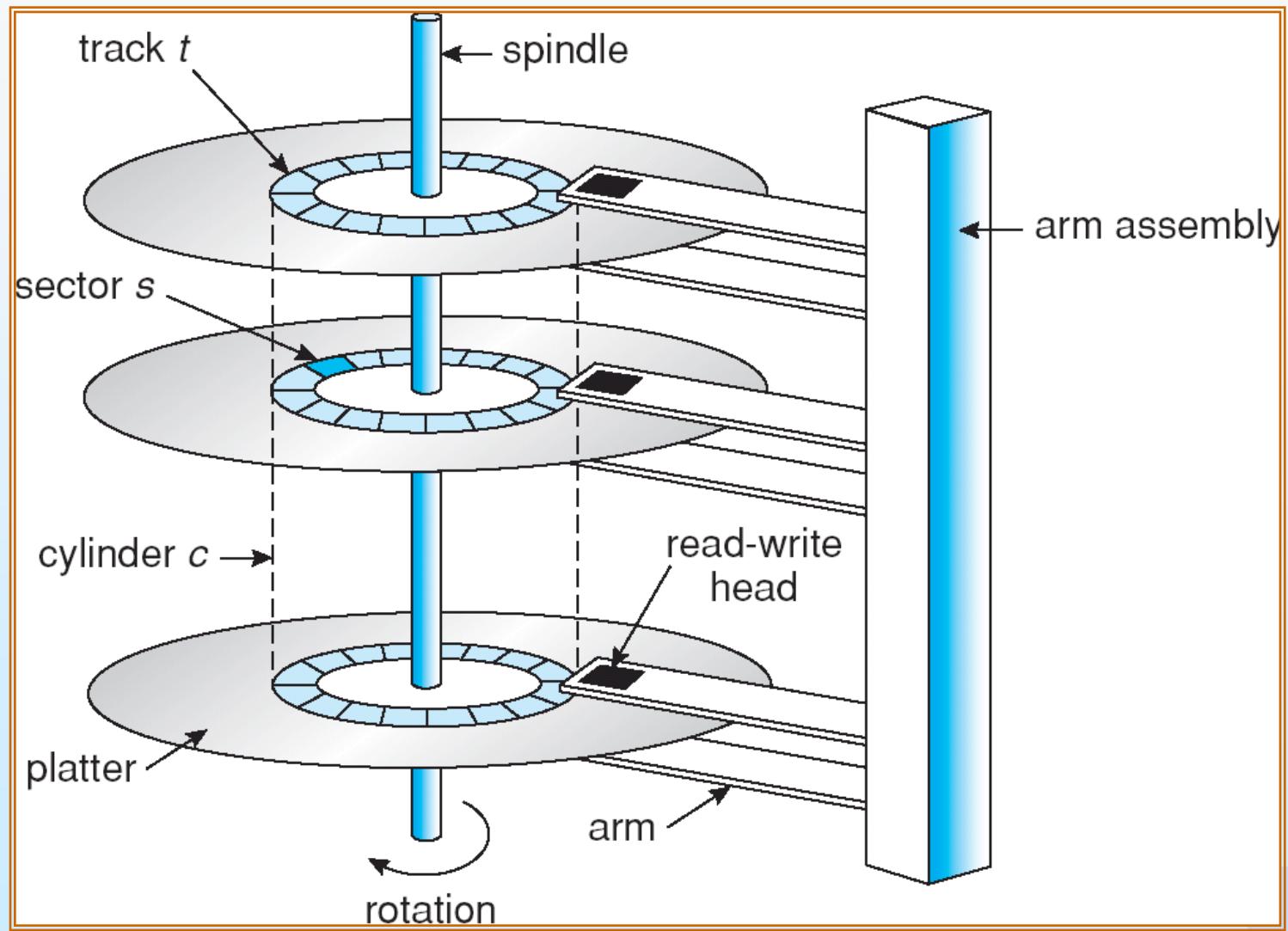
Overview of Mass Storage Structure

- Magnetic disks provide bulk of secondary storage of modern computers
 - Drives rotate at 60 to 200 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
- Drive attached to computer via **I/O bus**
 - Busses vary, including **EIDE, ATA, SATA, USB, Fibre Channel, SCSI**
 - **Host controller** in computer uses bus to talk to **disk controller** built into drive or storage array





Moving-head Disk Mechanism





Overview of Mass Storage Structure (Cont.)

- Magnetic tape

- Was early secondary-storage medium
- Relatively permanent and holds large quantities of data
- Access time slow
- Random access ~1000 times slower than disk
- Mainly used for backup, storage of infrequently-used data, transfer medium between systems
- Kept in spool and wound or rewound past read-write head
- Once data under head, transfer rates comparable to disk
- 20-200GB typical storage
- Common technologies are 4mm, 8mm, 19mm, LTO-2 and SDLT





Disk Structure

- Disk drives are addressed as large 1-dimensional arrays of *logical blocks*, where the logical block is the smallest unit of transfer.
- 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.





Disk Attachment

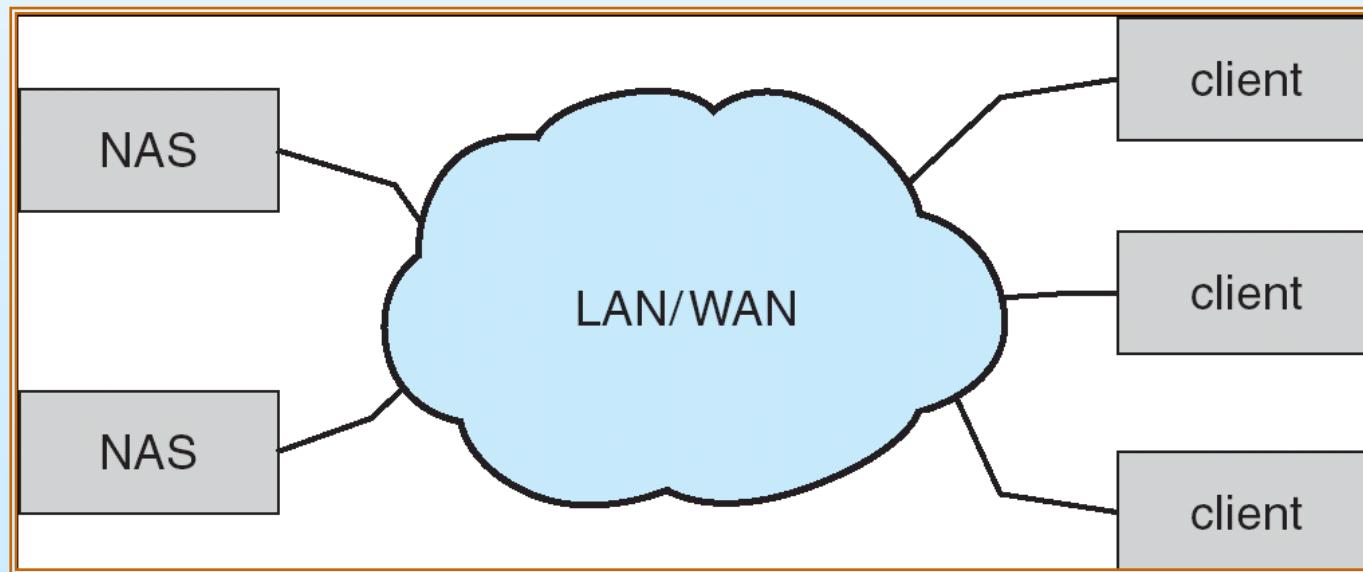
- Host-attached storage accessed through I/O ports talking to I/O busses
- SCSI itself is a bus, up to 16 devices on one cable, **SCSI initiator** requests operation and **SCSI targets** perform tasks
 - Each target can have up to 8 **logical units** (disks attached to device controller)
- FC is high-speed serial architecture
 - Can be switched fabric with 24-bit address space – the basis of **storage area networks (SANs)** in which many hosts attach to many storage units
 - Can be **arbitrated loop (FC-AL)** of 126 devices





Network-Attached Storage

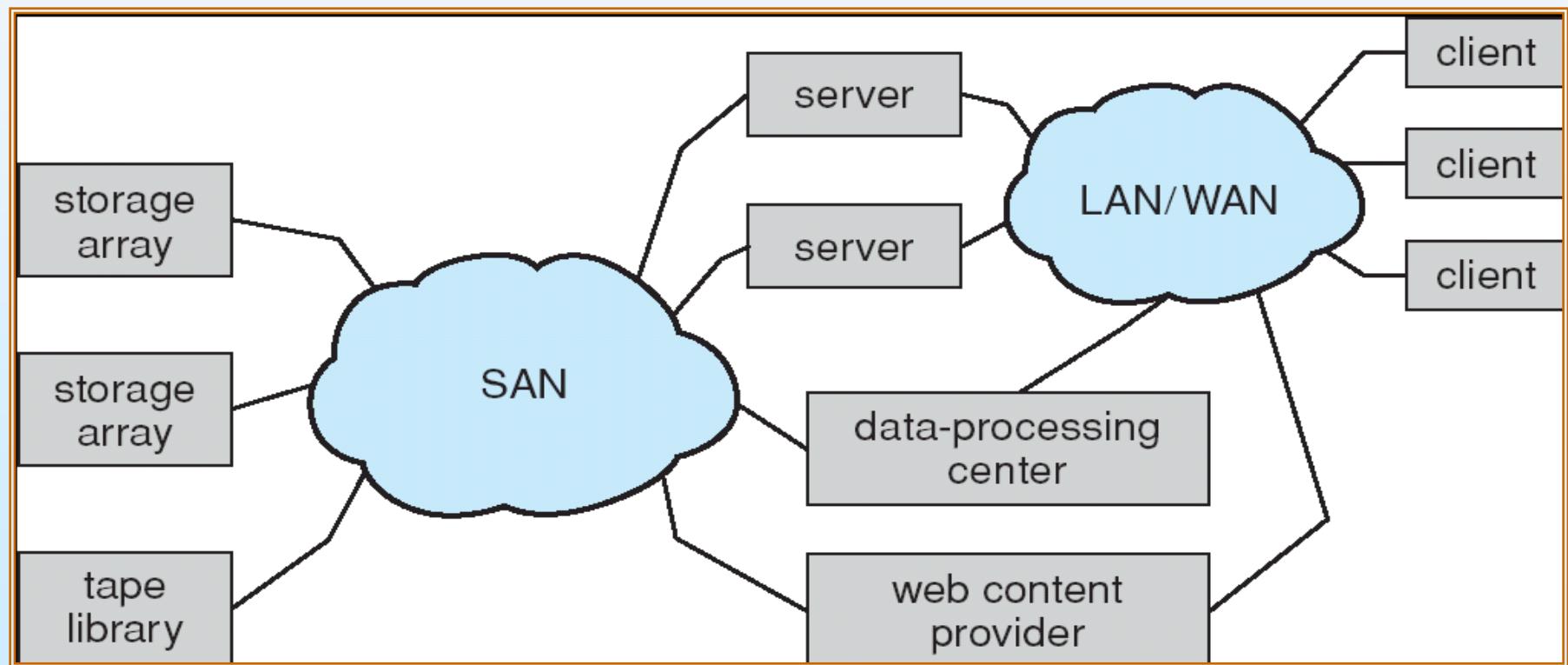
- Network-attached storage (**NAS**) is storage made available over a network rather than over a local connection (such as a bus)
- NFS and CIFS are common protocols
- Implemented via remote procedure calls (RPCs) between host and storage
- New iSCSI protocol uses IP network to carry the SCSI protocol





Storage Area Network

- Common in large storage environments (and becoming more common)
- Multiple hosts attached to multiple storage arrays - flexible





Disk Scheduling

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth.
- Access time has two major components
 - *Seek time* is the time for the disk are to move the heads to the cylinder containing the desired sector.
 - *Rotational latency* is the additional time waiting for the disk to rotate the desired sector to the disk head.
- 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.)

- Several algorithms exist to schedule the servicing of disk I/O requests.
- We illustrate them with a request queue (0-199).

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

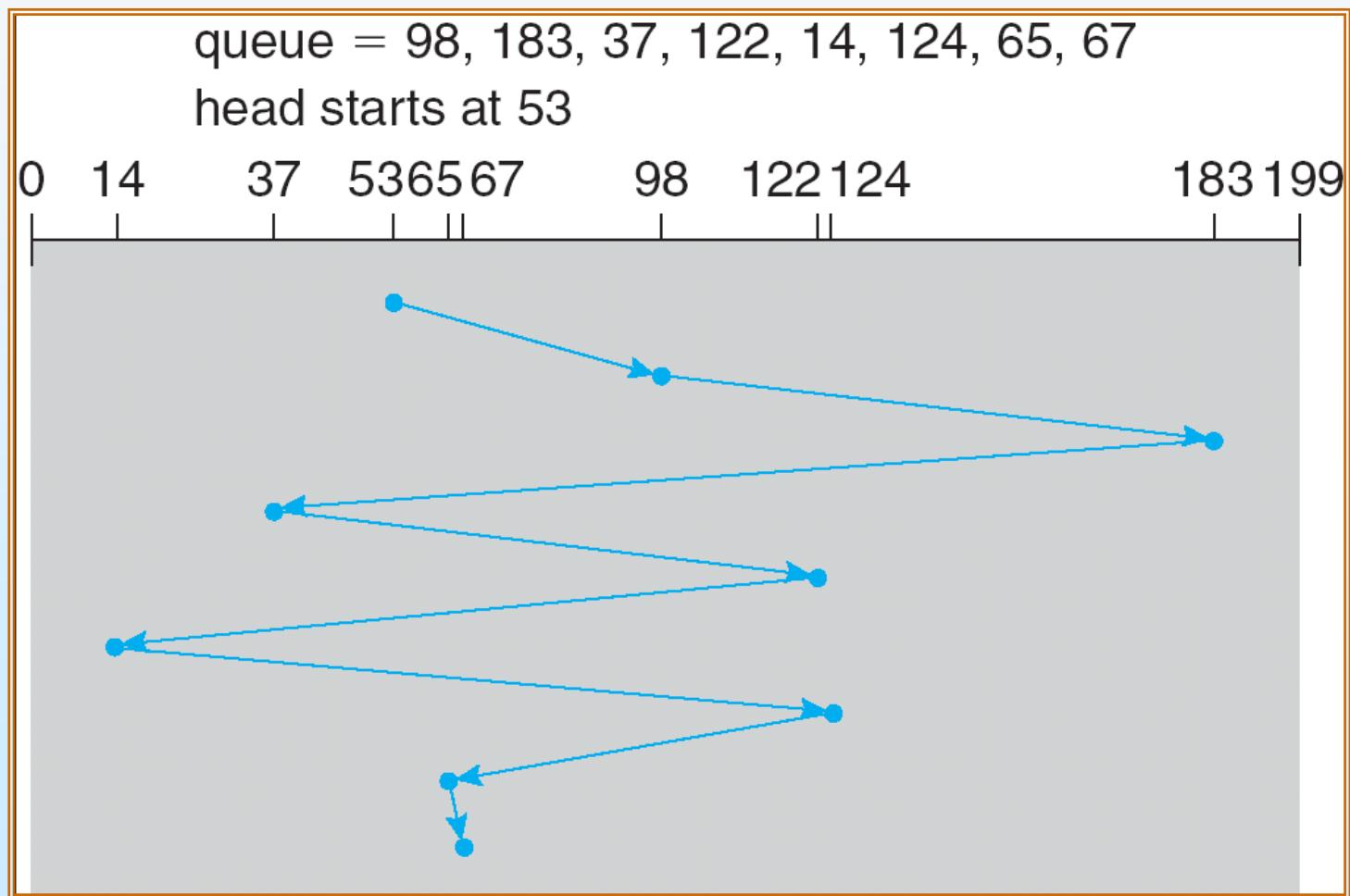
Head pointer 53





FCFS

Illustration shows total head movement of 640 cylinders.





SSTF

- Selects the request with the minimum seek time from the current head position.
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests.
- Illustration shows total head movement of 236 cylinders.

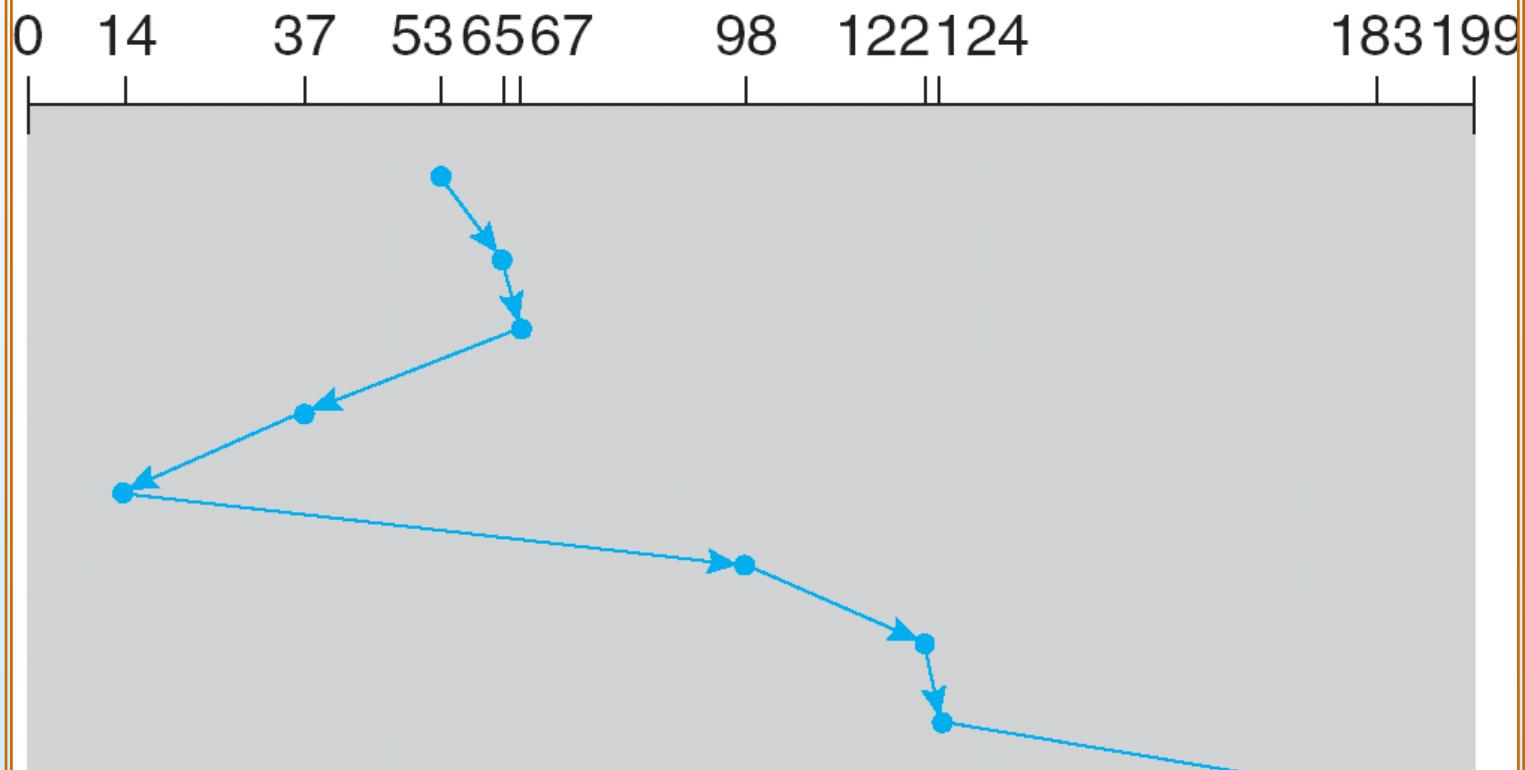




SSTF (Cont.)

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.
- Sometimes called the *elevator algorithm*.
- Illustration shows total head movement of 208 cylinders.

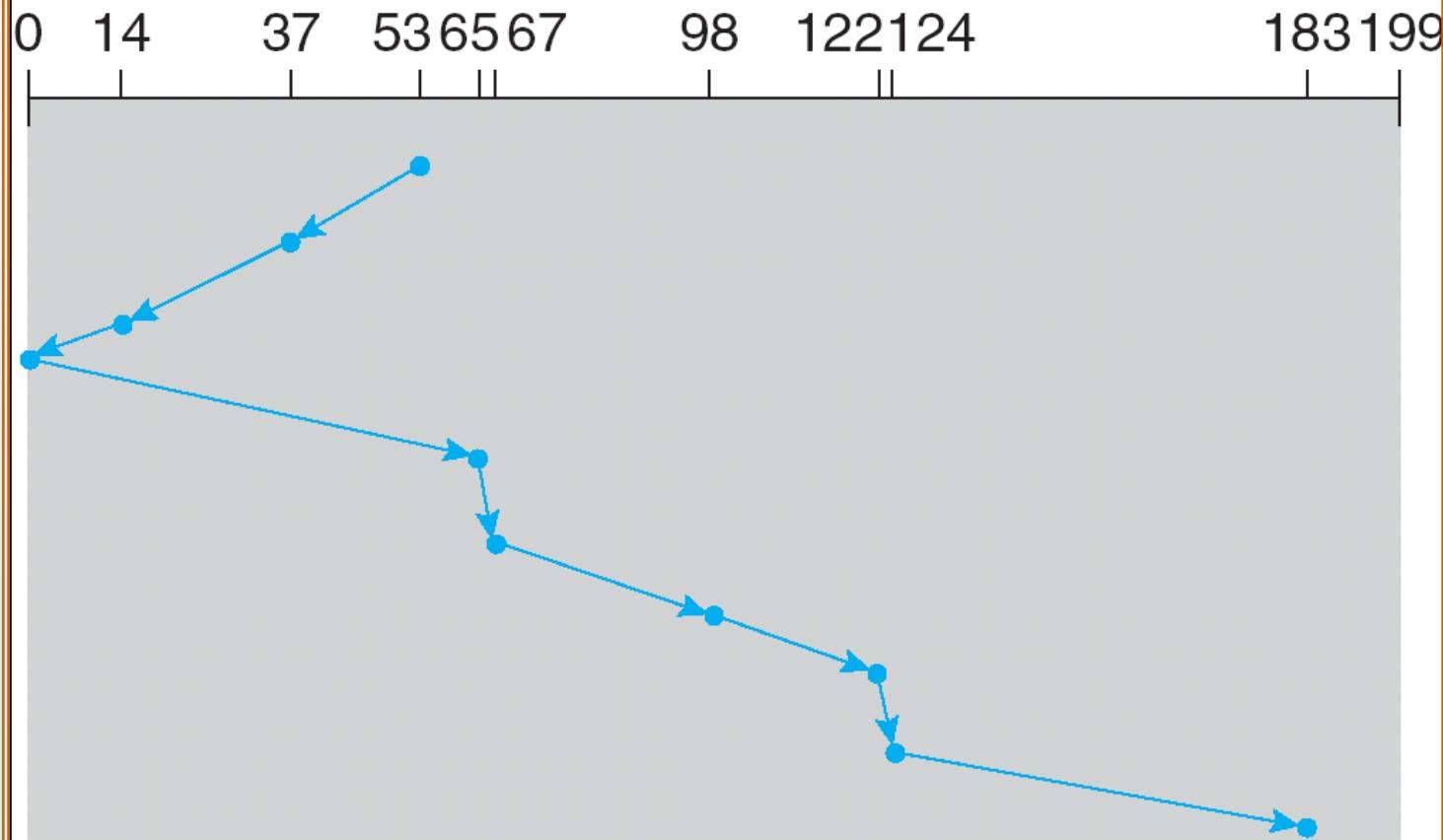




SCAN (Cont.)

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.

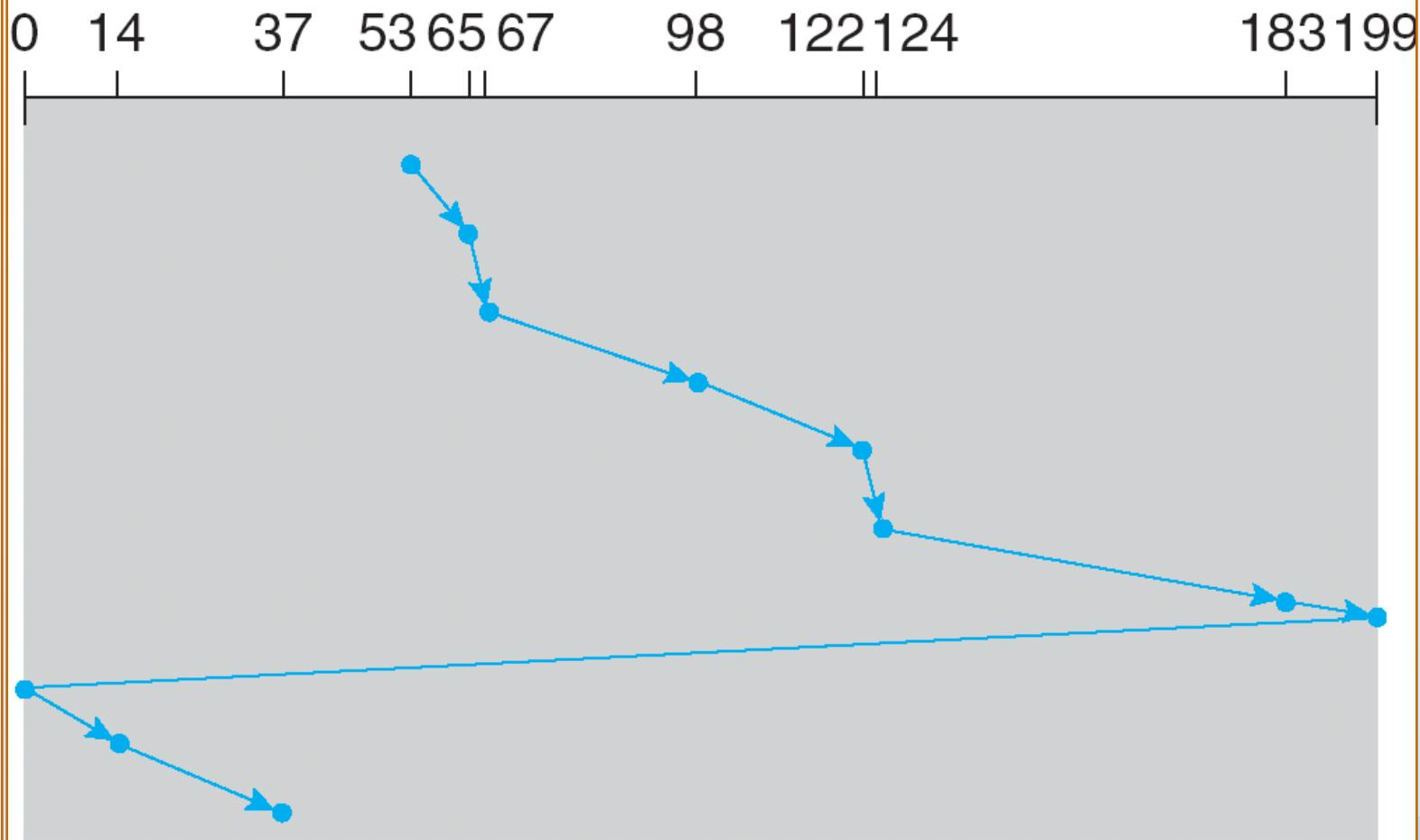




C-SCAN (Cont.)

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

head starts at 53





C-LOOK

- Version of C-SCAN
- Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.

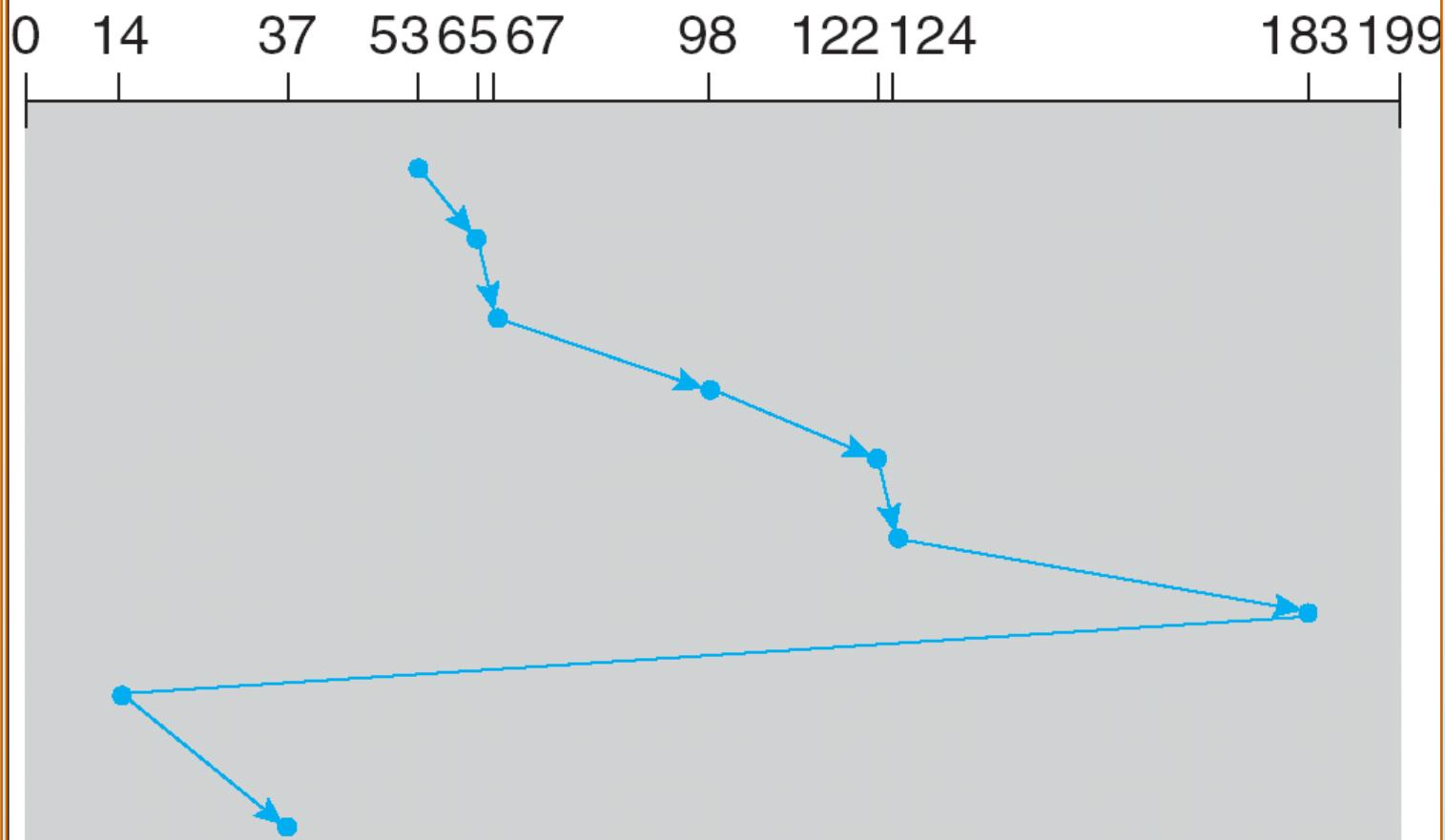




C-LOOK (Cont.)

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.
- Performance depends on the number and types of requests.
- Requests for disk service can be influenced by the file-allocation method.
- The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary.
- Either SSTF or LOOK is a reasonable choice for the default algorithm.





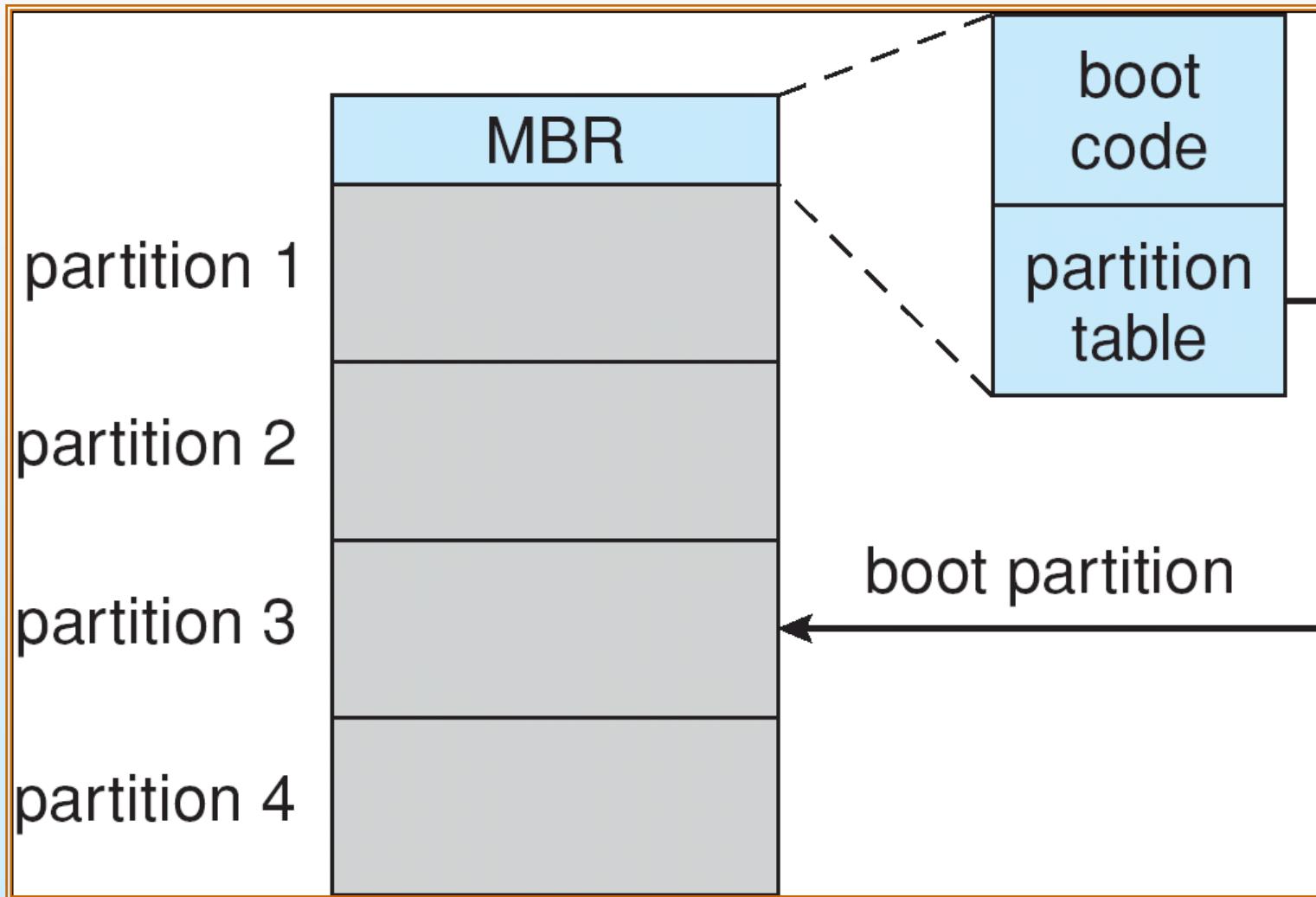
Disk Management

- *Low-level formatting*, or *physical formatting* — Dividing a disk into sectors that the disk controller can read and write.
- 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.
 - *Logical formatting* or “making a file system”.
- Boot block initializes system.
 - The bootstrap is stored in ROM.
 - *Bootstrap loader* program.
- Methods such as *sector sparing* used to handle bad blocks.





Booting from a Disk in Windows 2000



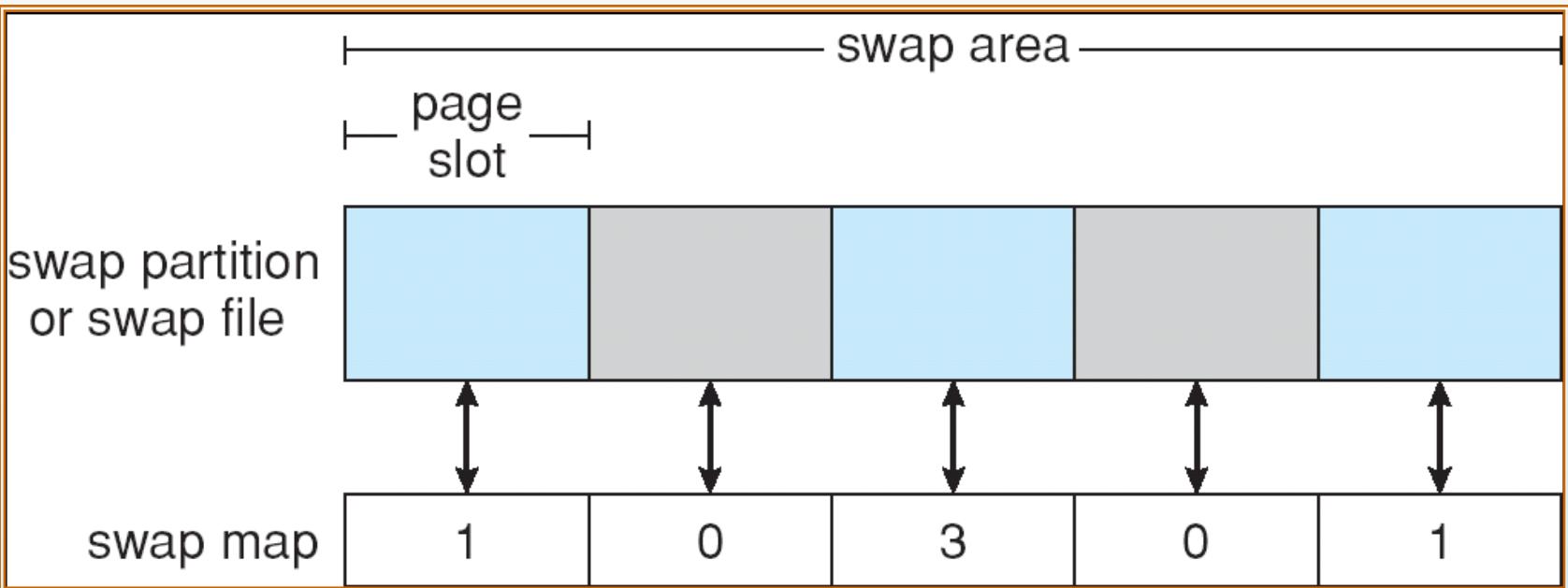


Swap-Space Management

- Swap-space — Virtual memory uses disk space as an extension of main memory.
- Swap-space can be carved out of the normal file system, or, more commonly, it can be in a separate disk partition.
- Swap-space management
 - 4.3BSD allocates swap space when process starts; holds *text segment* (the program) and *data segment*.
 - Kernel uses *swap maps* to track swap-space use.
 - Solaris 2 allocates swap space only when a page is forced out of physical memory, not when the virtual memory page is first created.



Data Structures for Swapping on Linux Systems





RAID Structure

- RAID – multiple disk drives provides **reliability** via **redundancy**.
- RAID is arranged into six different levels.





RAID (cont)

- Several improvements in disk-use techniques involve the use of multiple disks working cooperatively.
- Disk striping uses a group of disks as one storage unit.
- RAID schemes improve performance and improve the reliability of the storage system by storing redundant data.
 - *Mirroring or shadowing* keeps duplicate of each disk.
 - *Block interleaved parity* uses much less redundancy.





RAID Levels



(a) RAID 0: non-redundant striping.



(b) RAID 1: mirrored disks.



(c) RAID 2: memory-style error-correcting codes.



(d) RAID 3: bit-interleaved parity.



(e) RAID 4: block-interleaved parity.

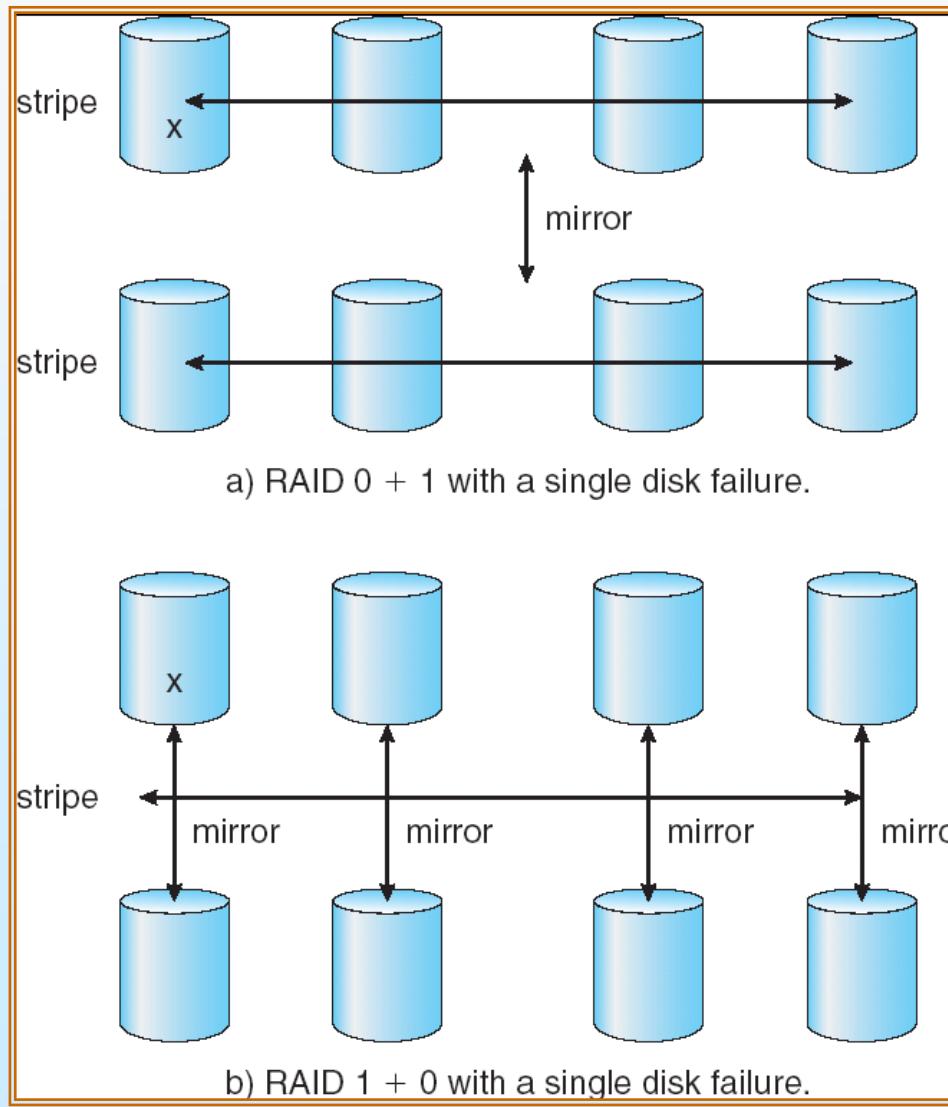


(f) RAID 5: block-interleaved distributed parity.





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





Stable-Storage Implementation

- Write-ahead log scheme requires stable storage.
- To implement stable storage:
 - Replicate information on more than one nonvolatile storage media with independent failure modes.
 - Update information in a controlled manner to ensure that we can recover the stable data after any failure during data transfer or recovery.





Tertiary Storage Devices

- Low cost is the defining characteristic of tertiary storage.
- Generally, tertiary storage is built using *removable media*
- Common examples of removable media are floppy disks and CD-ROMs; other types are available.





Removable Disks

- Floppy disk — thin flexible disk coated with magnetic material, enclosed in a protective plastic case.
- Most floppies hold about 1 MB; similar technology is used for removable disks that hold more than 1 GB.
- Removable magnetic disks can be nearly as fast as hard disks, but they are at a greater risk of damage from exposure.





Removable Disks (Cont.)

- A magneto-optic disk records data on a rigid platter coated with magnetic material.
 - Laser heat is used to amplify a large, weak magnetic field to record a bit.
 - Laser light is also used to read data (Kerr effect).
 - The magneto-optic head flies much farther from the disk surface than a magnetic disk head, and the magnetic material is covered with a protective layer of plastic or glass; resistant to head crashes.
- Optical disks do not use magnetism; they employ special materials that are altered by laser light.





WORM Disks

- The data on read-write disks can be modified over and over.
- WORM (“Write Once, Read Many Times”) disks can be written only once.
- Thin aluminum film sandwiched between two glass or plastic platters.
- To write a bit, the drive uses a laser light to burn a small hole through the aluminum; information can be destroyed by not altered.
- Very durable and reliable.
- *Read Only* disks, such ad CD-ROM and DVD, com from the factory with the data pre-recorded.





Tapes

- Compared to a disk, a tape is less expensive and holds more data, but random access is much slower.
- Tape is an economical medium for purposes that do not require fast random access, e.g., backup copies of disk data, holding huge volumes of data.
- Large tape installations typically use robotic tape changers that move tapes between tape drives and storage slots in a tape library.
 - stacker – library that holds a few tapes
 - silo – library that holds thousands of tapes
- A disk-resident file can be *archived* to tape for low cost storage; the computer can *stage* it back into disk storage for active use.





Operating System Issues

- Major OS jobs are to manage physical devices and to present a virtual machine abstraction to applications
- For hard disks, the OS provides two abstraction:
 - Raw device – an array of data blocks.
 - File system – the OS queues and schedules the interleaved requests from several applications.





Application Interface

- Most OSs handle removable disks almost exactly like fixed disks — a new cartridge is formatted and an empty file system is generated on the disk.
- Tapes are presented as a raw storage medium, i.e., and application does not open a file on the tape, it opens the whole tape drive as a raw device.
- Usually the tape drive is reserved for the exclusive use of that application.
- Since the OS does not provide file system services, the application must decide how to use the array of blocks.
- Since every application makes up its own rules for how to organize a tape, a tape full of data can generally only be used by the program that created it.





Tape Drives

- The basic operations for a tape drive differ from those of a disk drive.
- **locate** positions the tape to a specific logical block, not an entire track (corresponds to **seek**).
- The **read position** operation returns the logical block number where the tape head is.
- The **space** operation enables relative motion.
- Tape drives are “append-only” devices; updating a block in the middle of the tape also effectively erases everything beyond that block.
- An EOT mark is placed after a block that is written.





File Naming

- The issue of naming files on removable media is especially difficult when we want to write data on a removable cartridge on one computer, and then use the cartridge in another computer.
- Contemporary OSs generally leave the name space problem unsolved for removable media, and depend on applications and users to figure out how to access and interpret the data.
- Some kinds of removable media (e.g., CDs) are so well standardized that all computers use them the same way.





Hierarchical Storage Management (HSM)

- A hierarchical storage system extends the storage hierarchy beyond primary memory and secondary storage to incorporate tertiary storage — usually implemented as a jukebox of tapes or removable disks.
- Usually incorporate tertiary storage by extending the file system.
 - Small and frequently used files remain on disk.
 - Large, old, inactive files are archived to the jukebox.
- HSM is usually found in supercomputing centers and other large installations that have enormous volumes of data.





Speed

- Two aspects of speed in tertiary storage are bandwidth and latency.
- Bandwidth is measured in bytes per second.
 - Sustained bandwidth – average data rate during a large transfer; # of bytes/transfer time.
Data rate when the data stream is actually flowing.
 - Effective bandwidth – average over the entire I/O time, including **seek** or **locate**, and cartridge switching.
Drive's overall data rate.





Speed (Cont.)

- Access latency – amount of time needed to locate data.
 - Access time for a disk – move the arm to the selected cylinder and wait for the rotational latency; < 35 milliseconds.
 - Access on tape requires winding the tape reels until the selected block reaches the tape head; tens or hundreds of seconds.
 - Generally say that random access within a tape cartridge is about a thousand times slower than random access on disk.
- The low cost of tertiary storage is a result of having many cheap cartridges share a few expensive drives.
- A removable library is best devoted to the storage of infrequently used data, because the library can only satisfy a relatively small number of I/O requests per hour.





Reliability

- A fixed disk drive is likely to be more reliable than a removable disk or tape drive.
- An optical cartridge is likely to be more reliable than a magnetic disk or tape.
- A head crash in a fixed hard disk generally destroys the data, whereas the failure of a tape drive or optical disk drive often leaves the data cartridge unharmed.





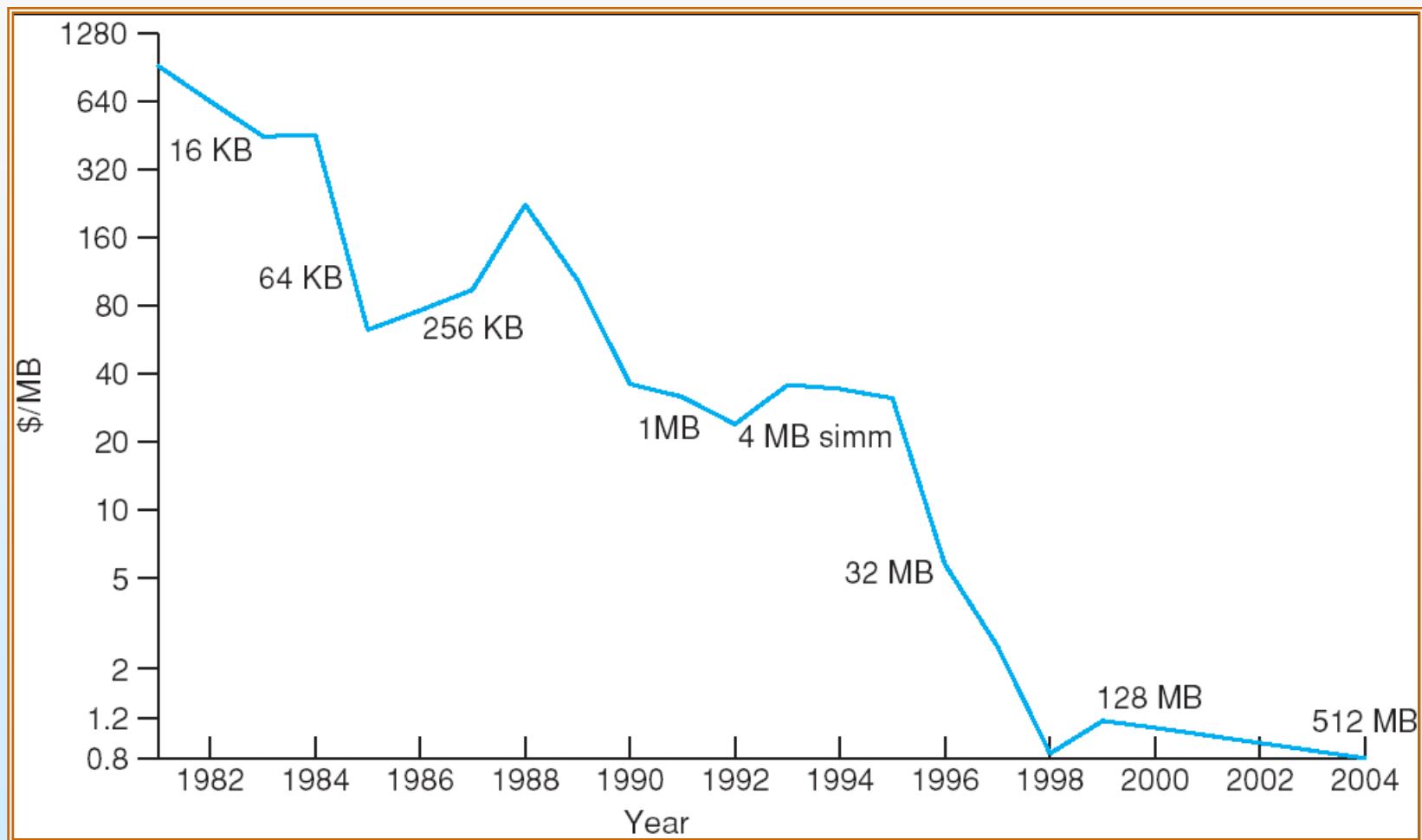
Cost

- Main memory is much more expensive than disk storage
- The cost per megabyte of hard disk storage is competitive with magnetic tape if only one tape is used per drive.
- The cheapest tape drives and the cheapest disk drives have had about the same storage capacity over the years.
- Tertiary storage gives a cost savings only when the number of cartridges is considerably larger than the number of drives.



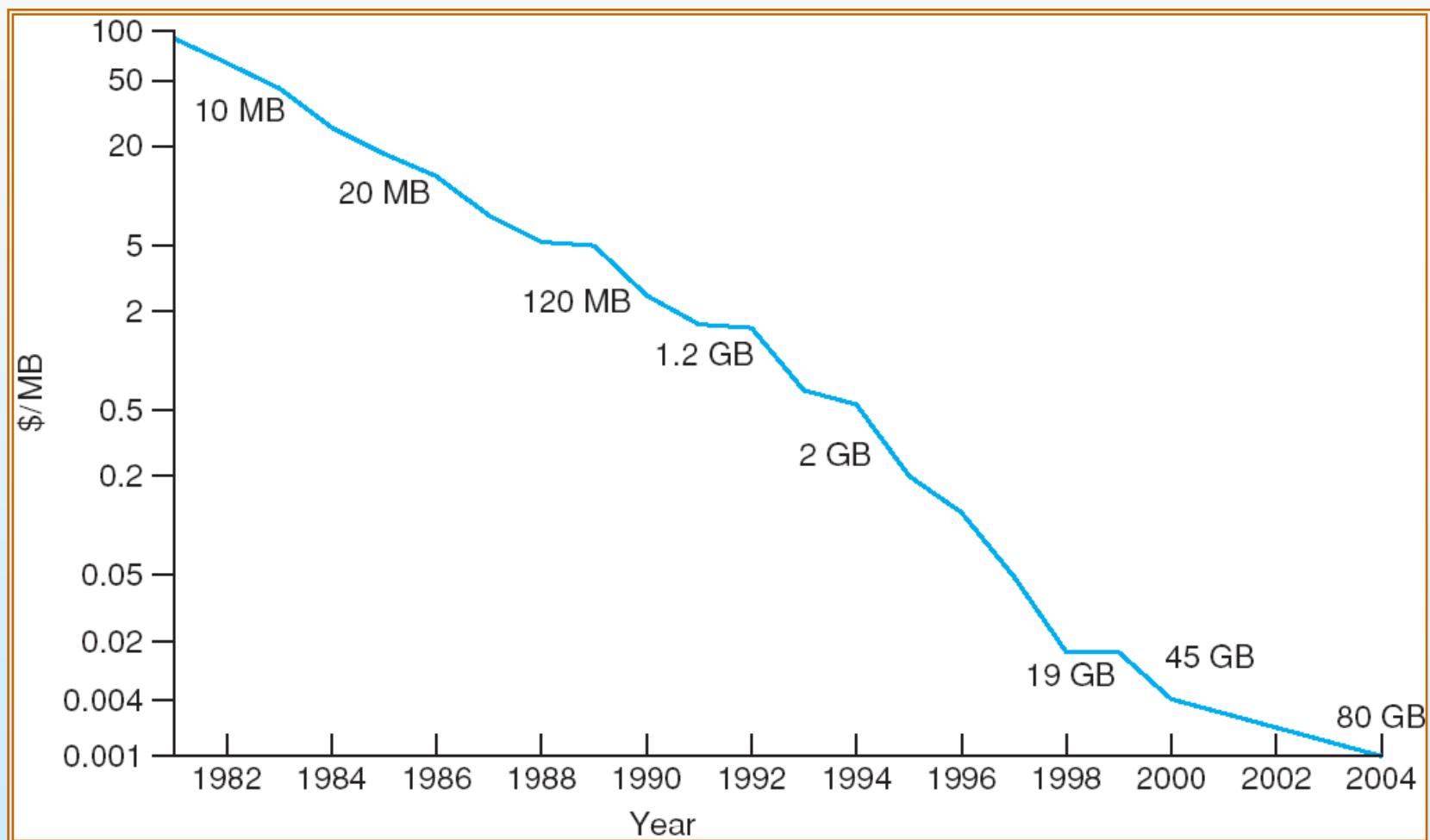


Price per Megabyte of DRAM, From 1981 to 2004



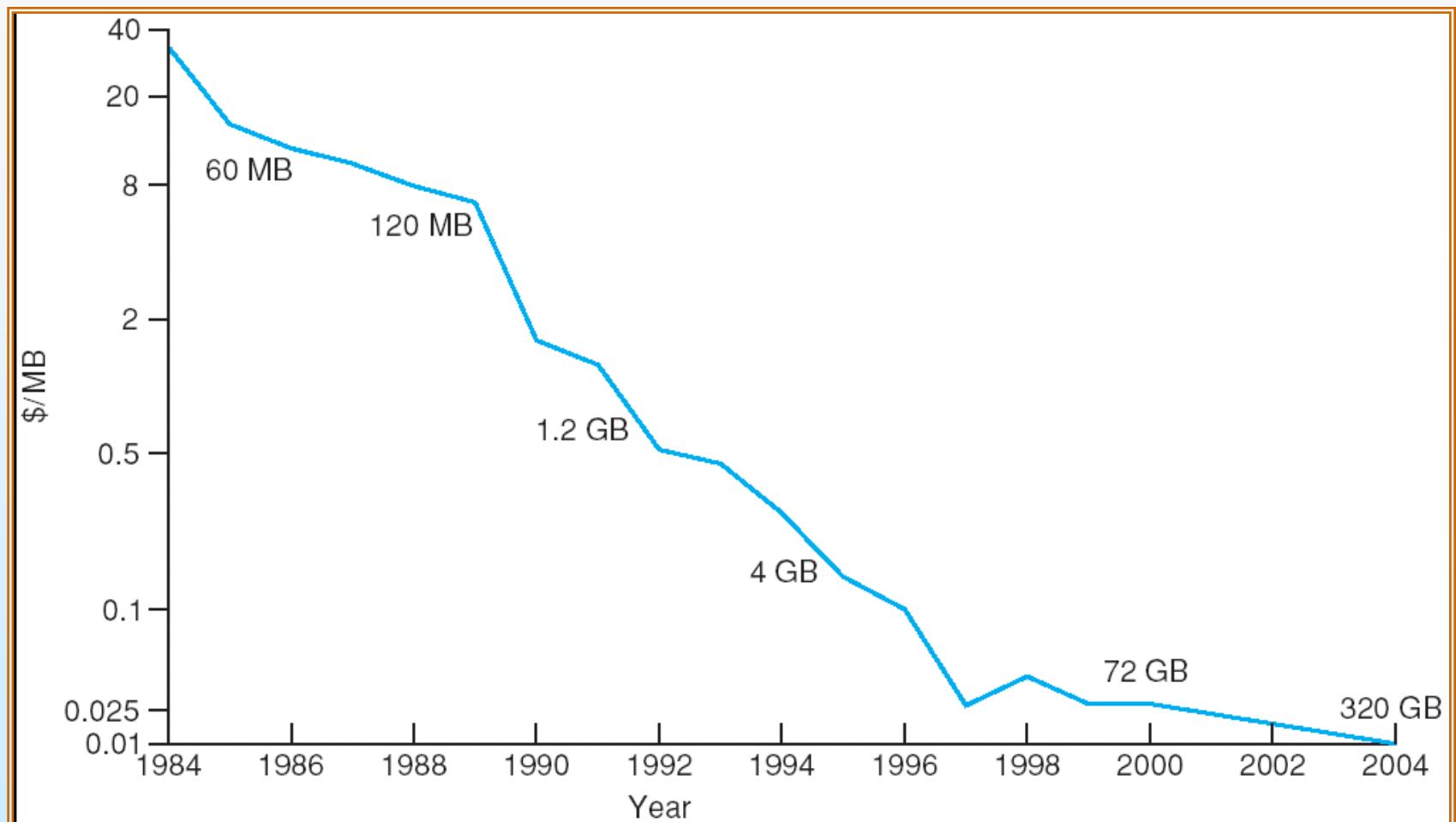


Price per Megabyte of Magnetic Hard Disk, From 1981 to 2004





Price per Megabyte of a Tape Drive, From 1984-2000



End of Chapter 12



Chapter 13: I/O Systems





Chapter 13: I/O Systems

- 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 of I/O hardware and its complexity
- Provide details of the performance aspects of I/O hardware and software





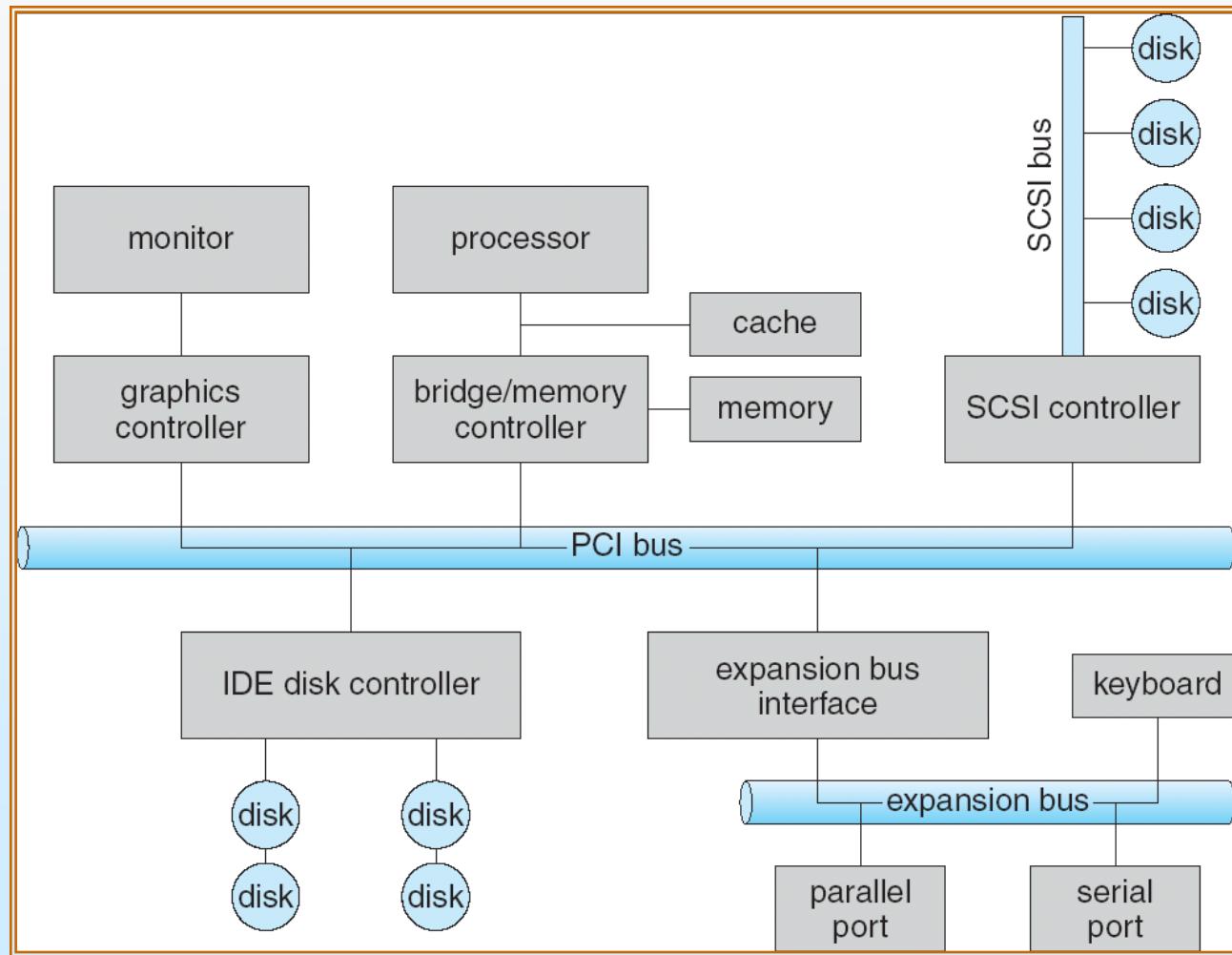
I/O Hardware

- Incredible variety of I/O devices
- Common concepts
 - **Port**
 - **Bus (daisy chain or shared direct access)**
 - **Controller (host adapter)**
- I/O instructions control devices
- Devices have addresses, used by
 - Direct I/O instructions
 - **Memory-mapped I/O**





A Typical PC Bus Structure





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

- Determines state of device
 - command-ready
 - busy
 - Error
- **Busy-wait** cycle to wait for I/O from device





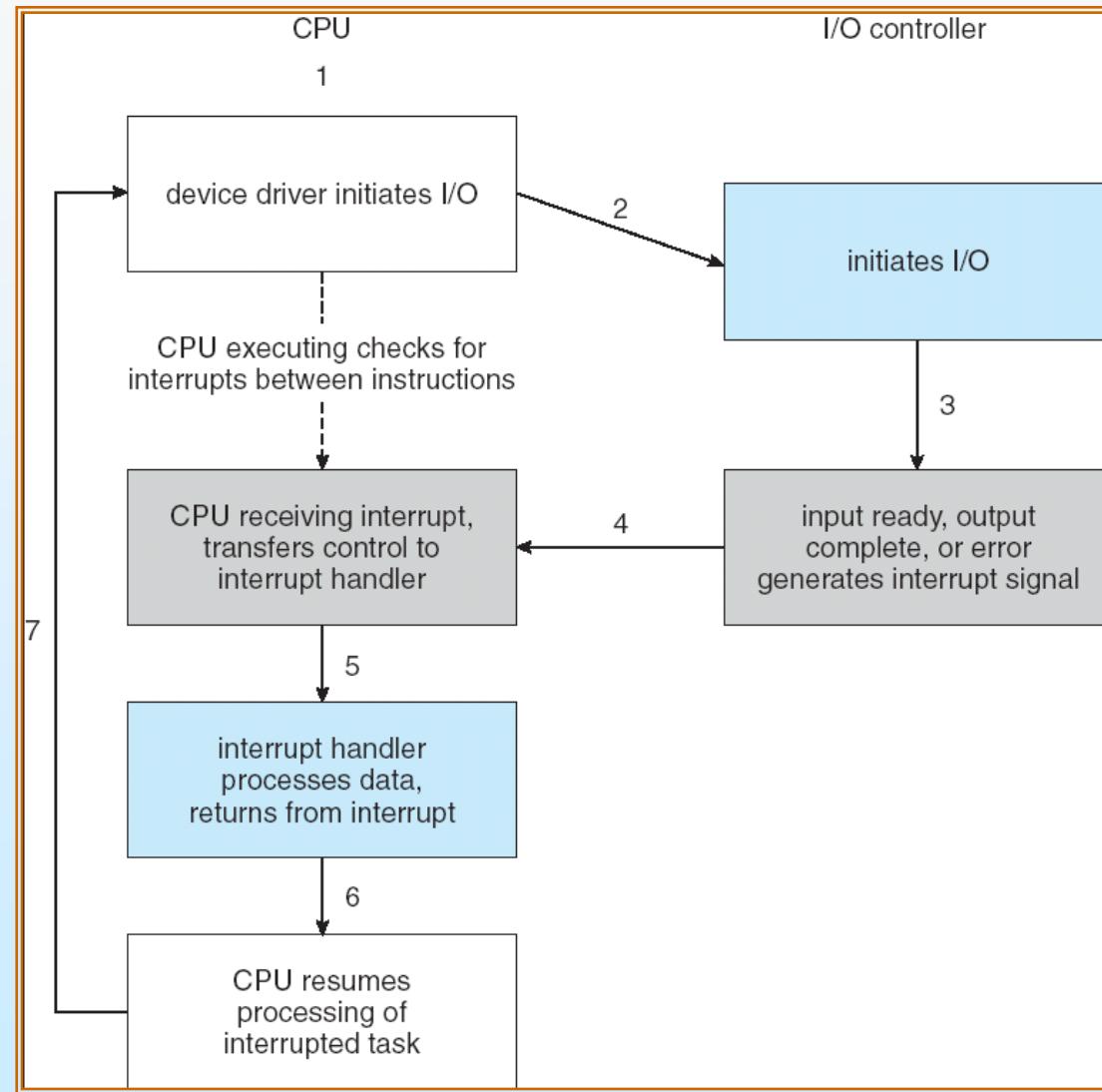
Interrupts

- CPU **Interrupt-request line** triggered by I/O device
- **Interrupt handler** receives interrupts
- **Maskable** to ignore or delay some interrupts
- Interrupt vector to dispatch interrupt to correct handler
 - Based on priority
 - Some **nonmaskable**
- Interrupt mechanism also used for exceptions





Interrupt-Driven I/O Cycle





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





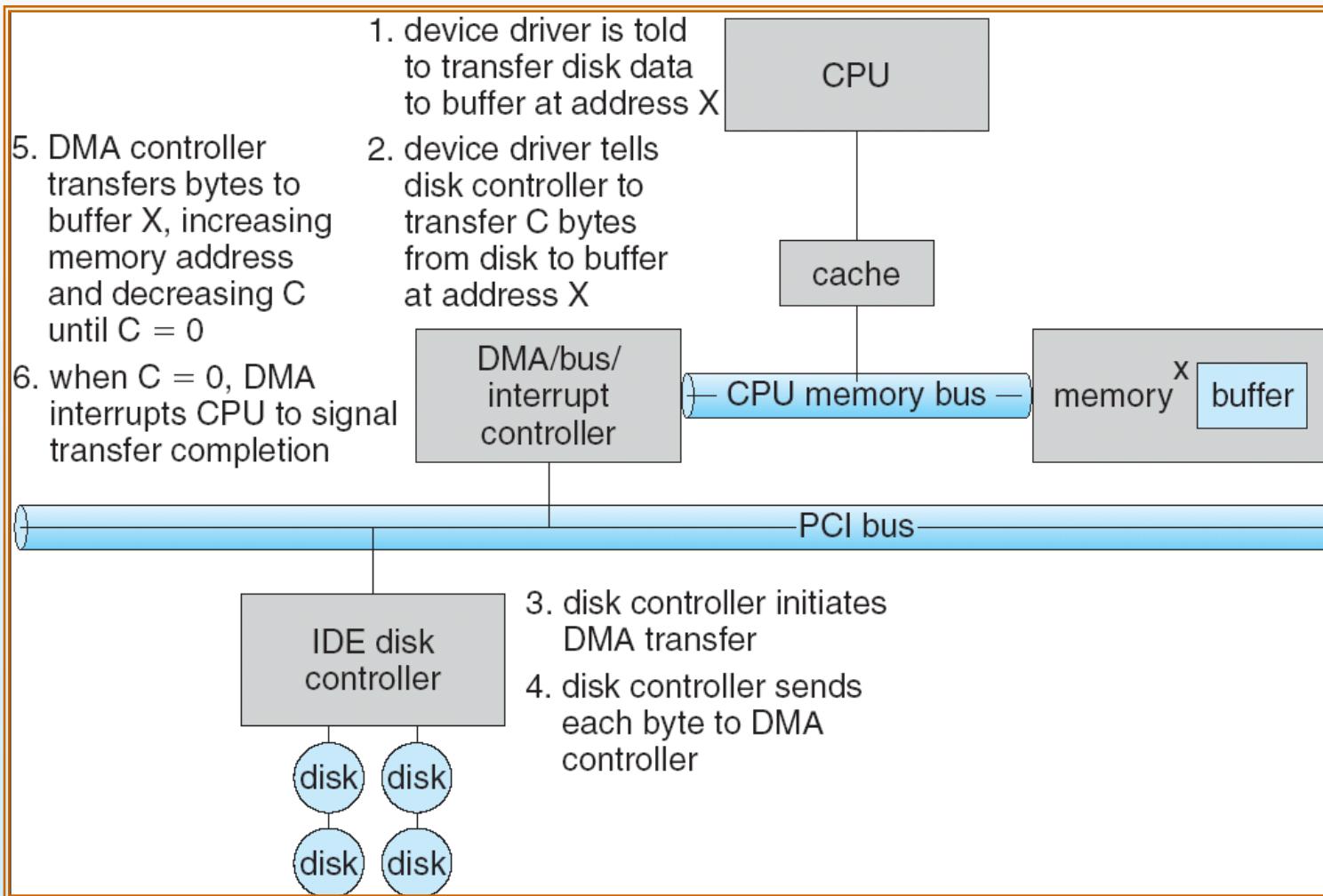
Direct Memory Access

- Used to avoid **programmed I/O** for large data movement
- Requires **DMA** controller
- Bypasses CPU to transfer data directly between I/O device and memory





Six Step Process to Perform DMA Transfer





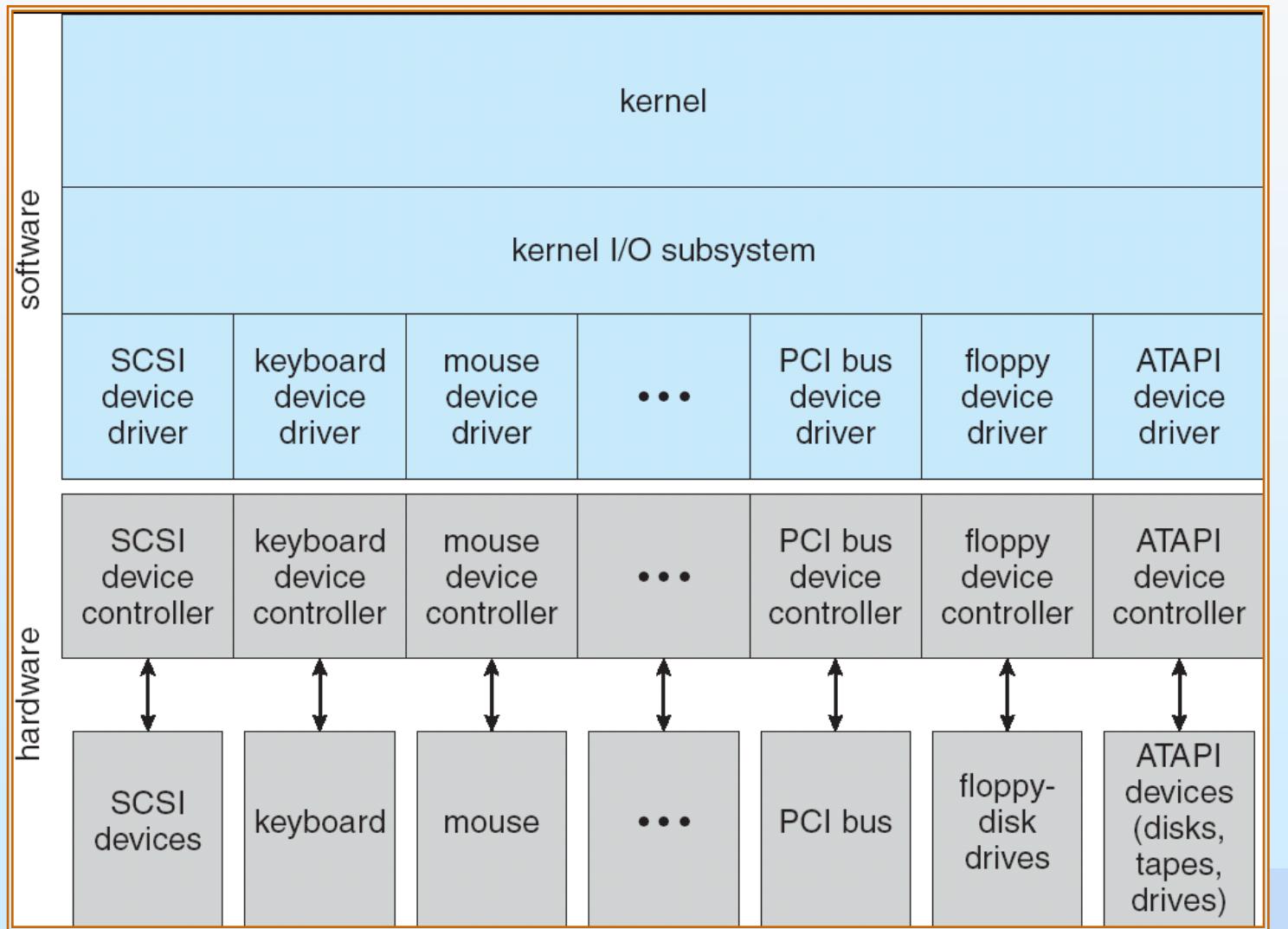
Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- Devices vary in many dimensions
 - **Character-stream or block**
 - **Sequential or random-access**
 - **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





Block and Character Devices

- Block devices include disk drives
 - Commands include read, write, seek
 - Raw I/O or file-system access
 - Memory-mapped file access possible

- 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
- Unix and Windows NT/9x/2000 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
- **Programmable interval timer** used for timings, periodic interrupts
- `ioctl` (on UNIX) covers odd aspects of I/O such as clocks and timers





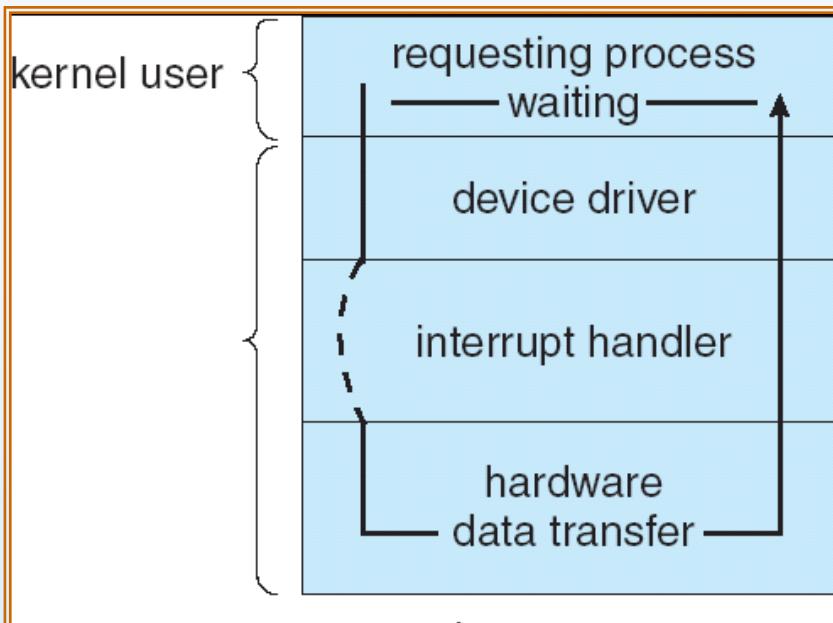
Blocking and Nonblocking I/O

- **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
- **Asynchronous** - process runs while I/O executes
 - Difficult to use
 - I/O subsystem signals process when I/O completed



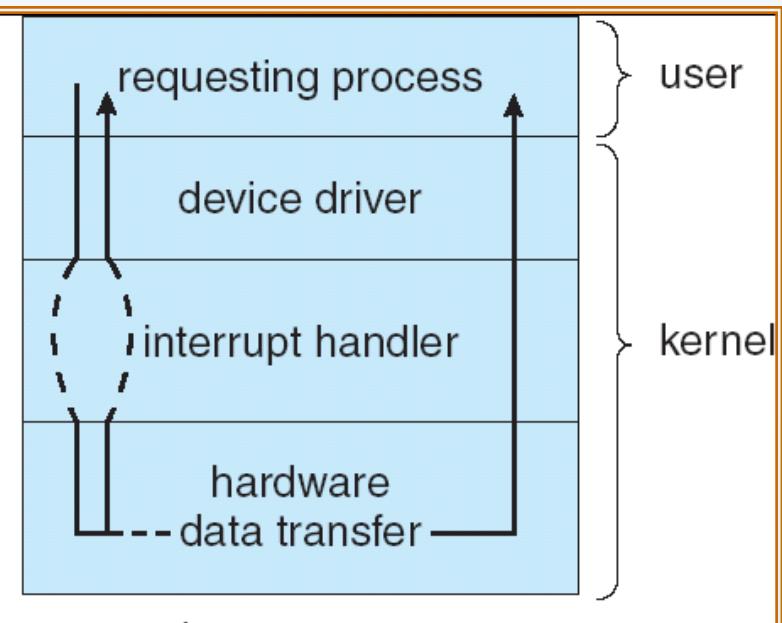


Two I/O Methods



(a)

Synchronous



(b)

Asynchronous





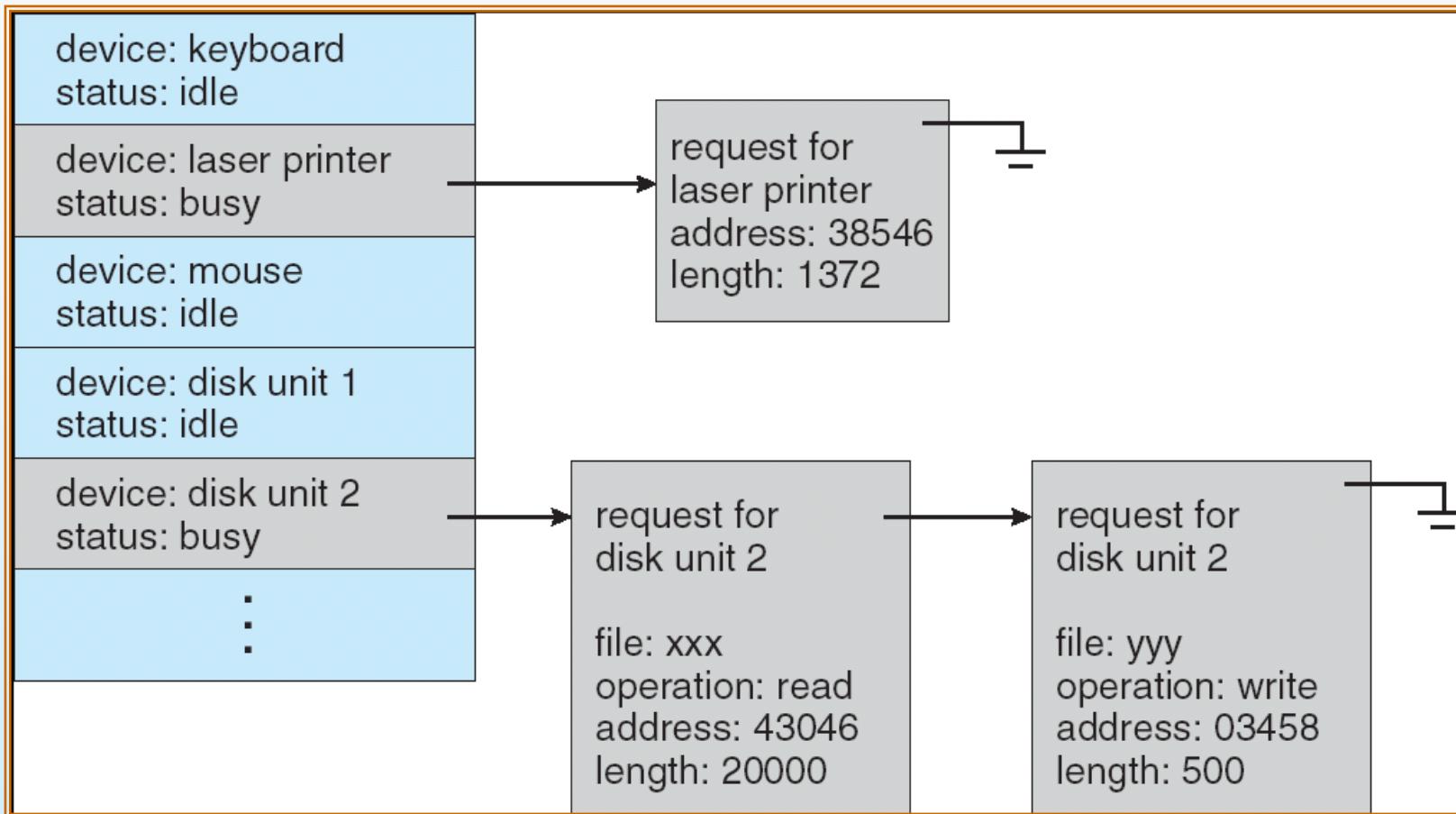
Kernel I/O Subsystem

- Scheduling
 - Some I/O request ordering via per-device queue
 - Some OSs try fairness
- 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”



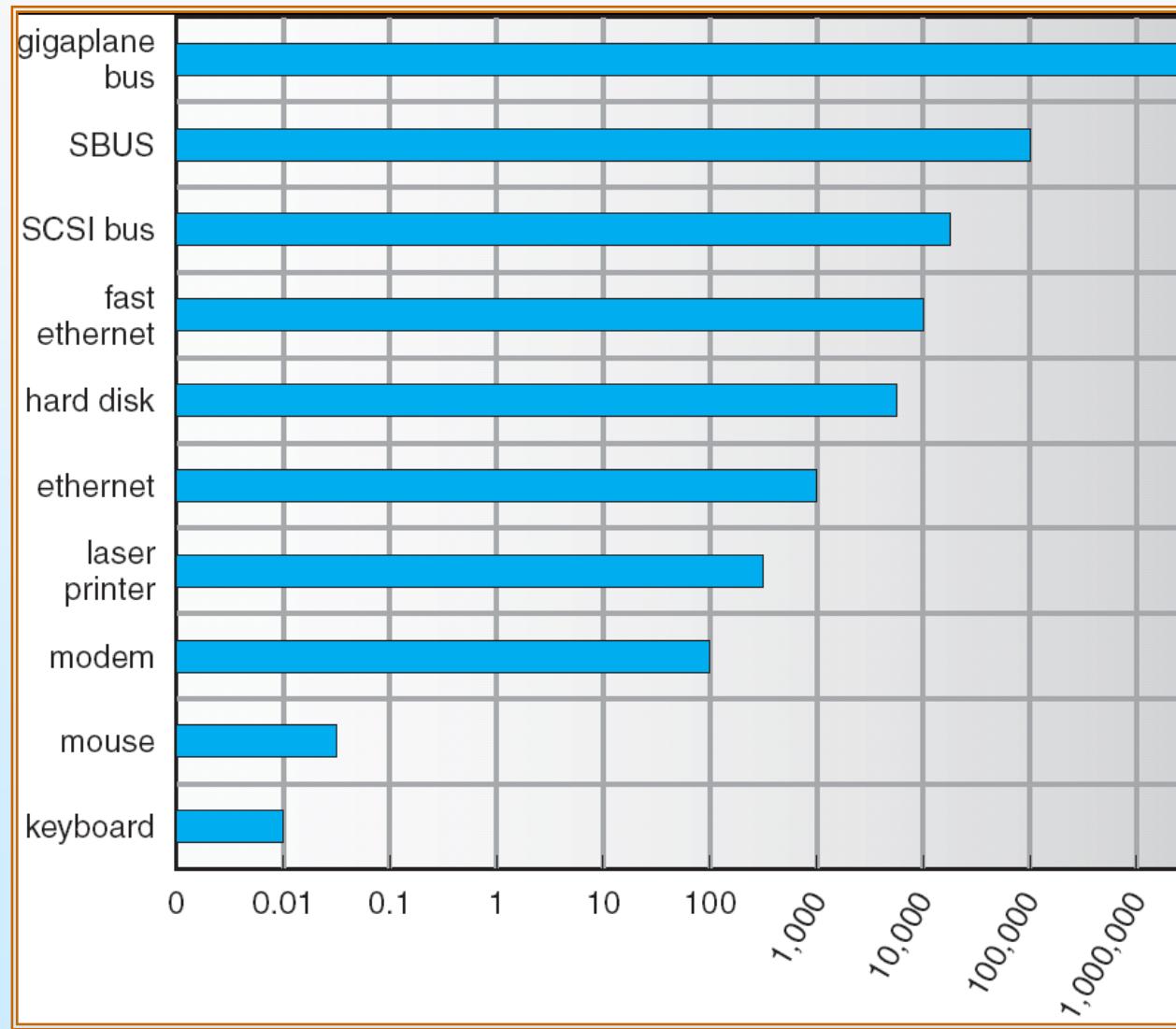


Device-status Table





Sun Enterprise 6000 Device-Transfer Rates





Kernel I/O Subsystem

- **Caching** - fast memory holding copy of data
 - Always just a copy
 - Key to performance
- **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 deallocation
 - Watch out for deadlock





Error Handling

- OS can recover from disk read, device unavailable, transient write failures
- Most return an error number or code when I/O request fails
- System error logs hold problem reports





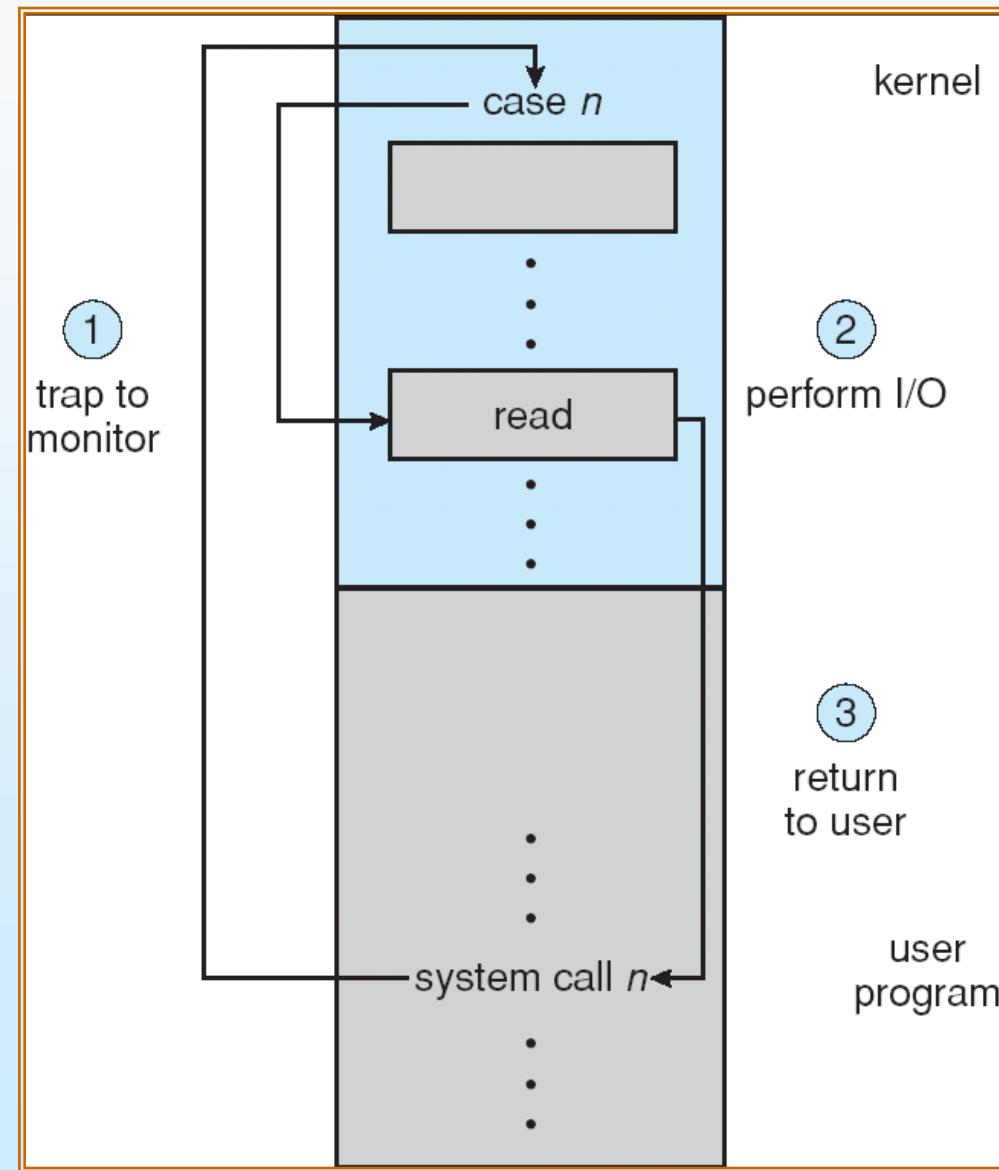
I/O Protection

- 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





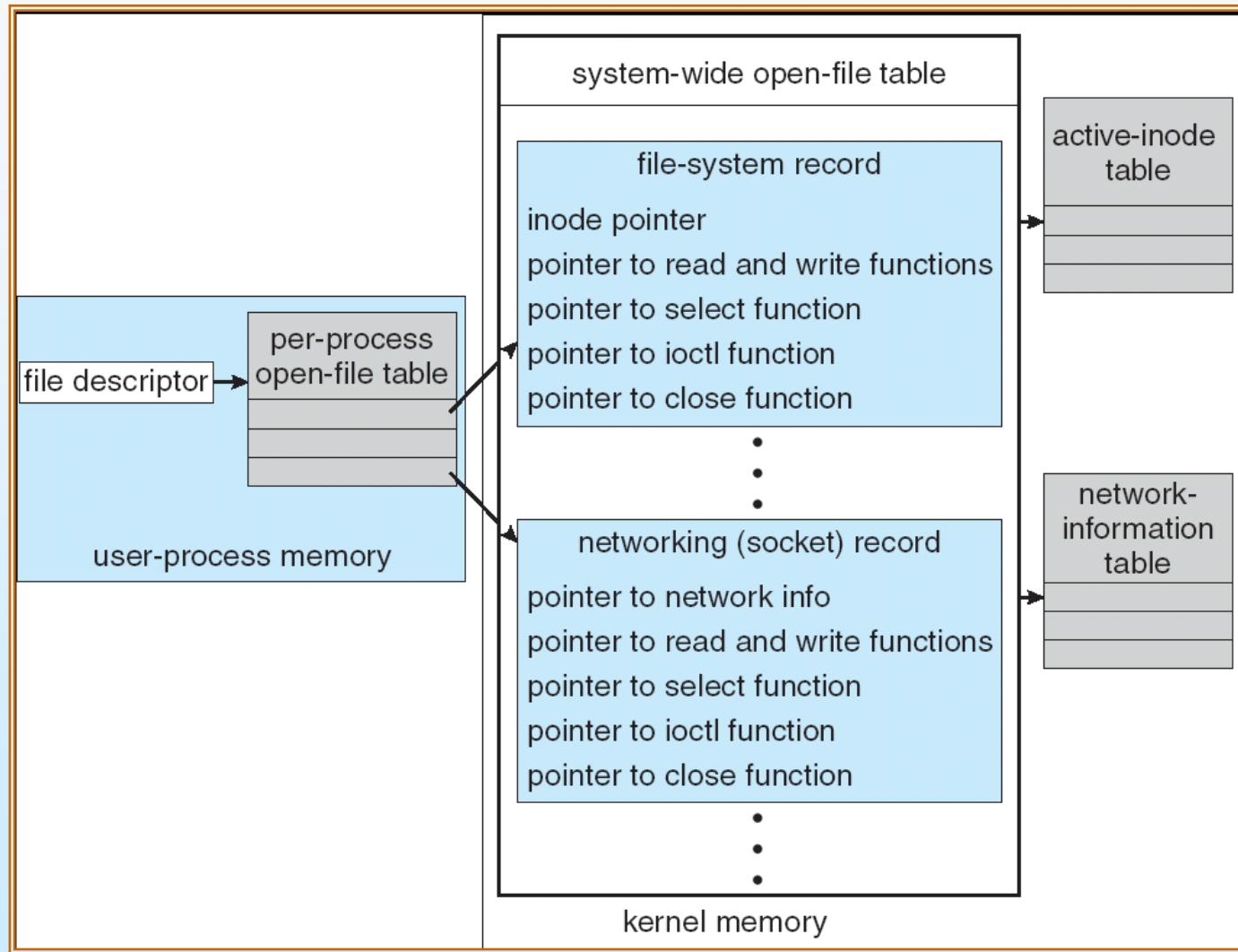
Kernel Data Structures

- 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





UNIX I/O Kernel Structure





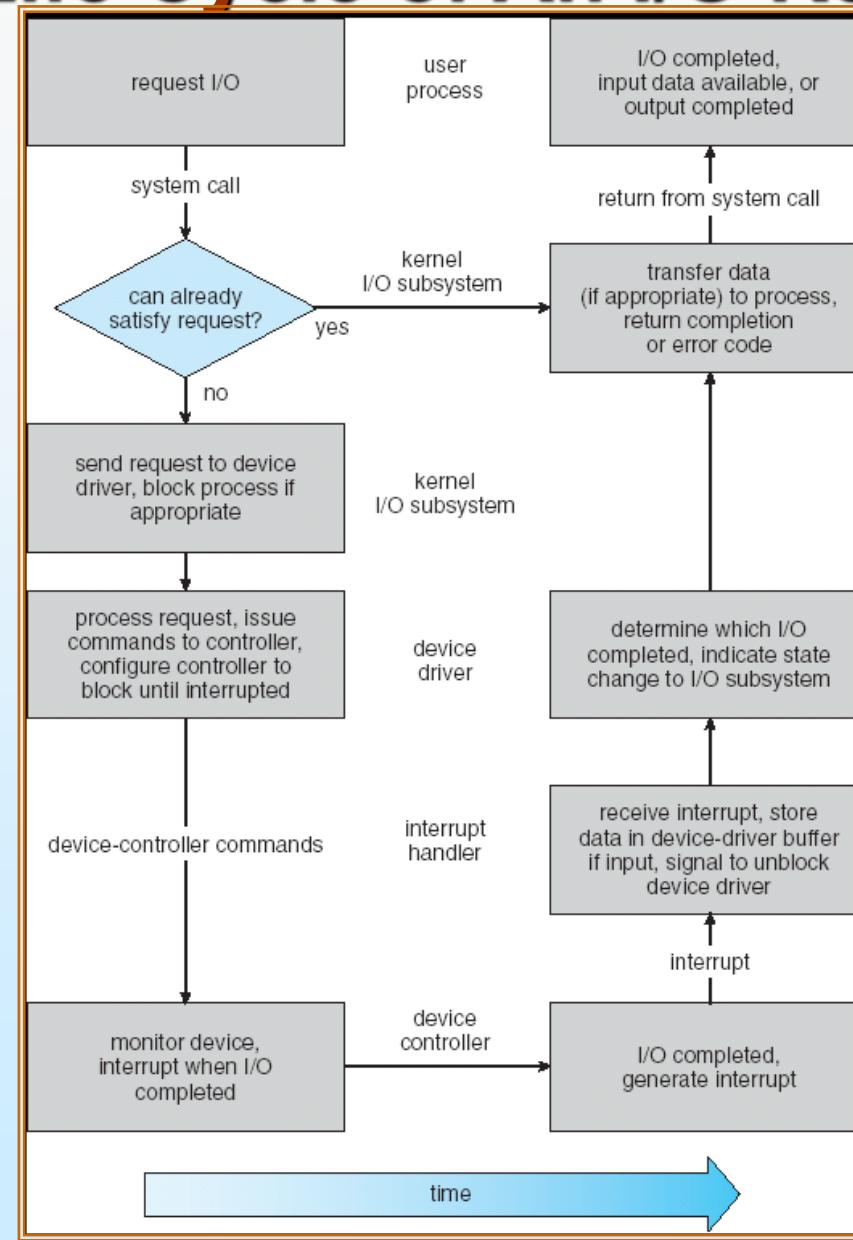
I/O Requests to Hardware Operations

- 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





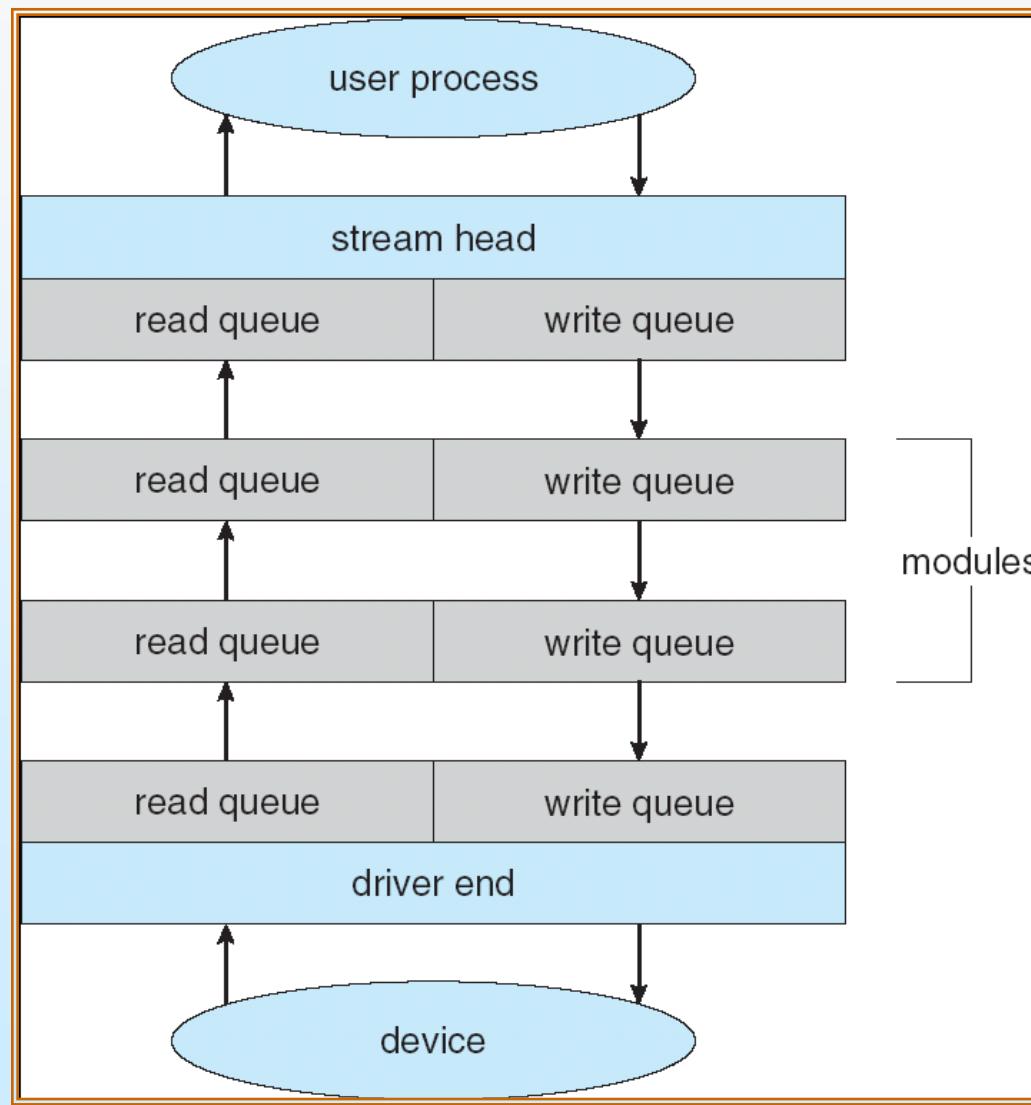
STREAMS

- **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





The STREAMS Structure





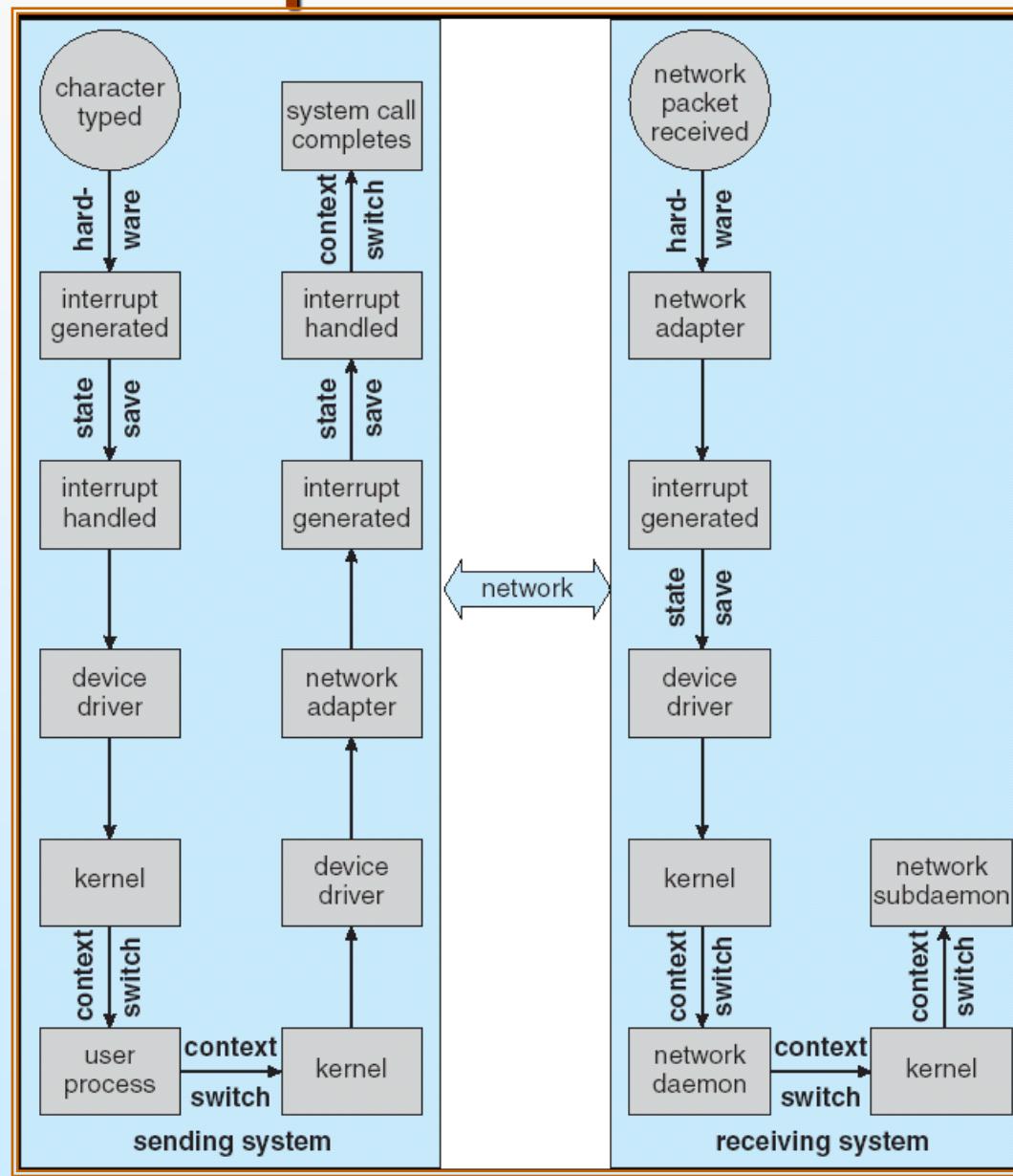
Performance

- 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





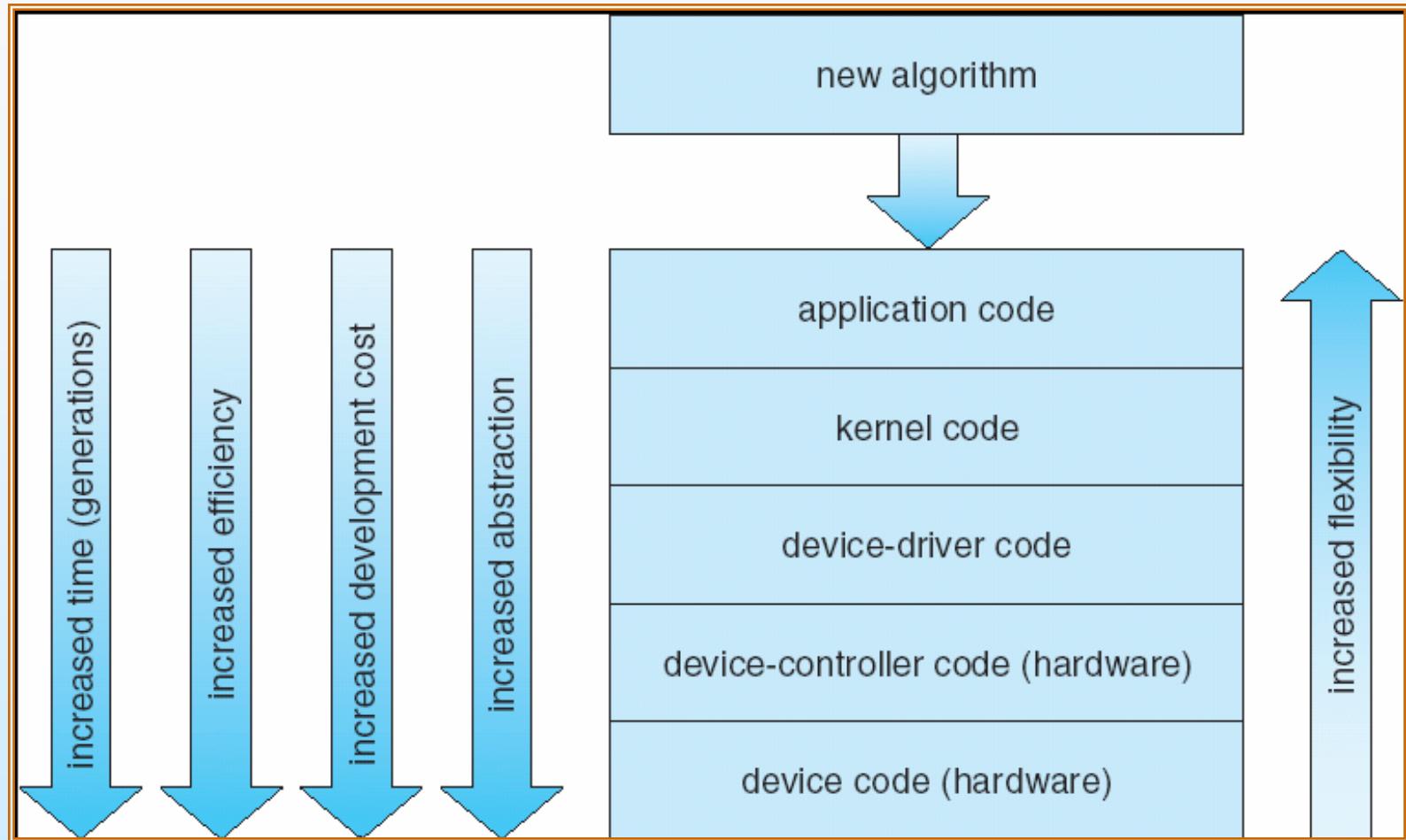
Improving Performance

- Reduce number of context switches
- Reduce data copying
- Reduce interrupts by using large transfers, smart controllers, polling
- Use DMA
- Balance CPU, memory, bus, and I/O performance for highest throughput

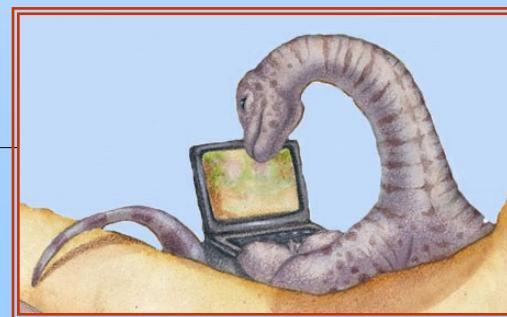




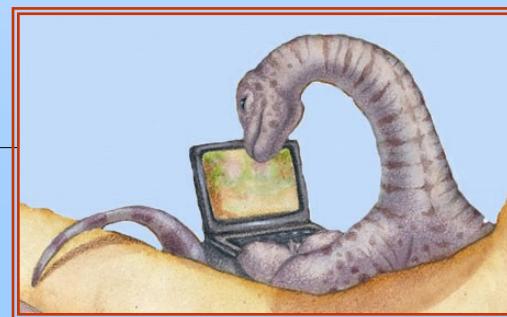
Device-Functionality Progression



End of Chapter 13



Chapter 14: Protection





Chapter 14: Protection

- Goals of Protection
- Principles of Protection
- Domain of Protection
- Access Matrix
- Implementation of Access Matrix
- Access Control
- Revocation of Access Rights
- Capability-Based Systems
- Language-Based Protection





Objectives

- Discuss the goals and principles of protection in a modern computer system
- Explain how protection domains combined with an access matrix are used to specify the resources a process may access
- Examine capability and language-based protection systems





Goals of Protection

- Operating system consists of a collection of objects, hardware or software
- Each object has a unique name and can be accessed through a well-defined set of operations.
- Protection problem - ensure that each object is accessed correctly and only by those processes that are allowed to do so.





Principles of Protection

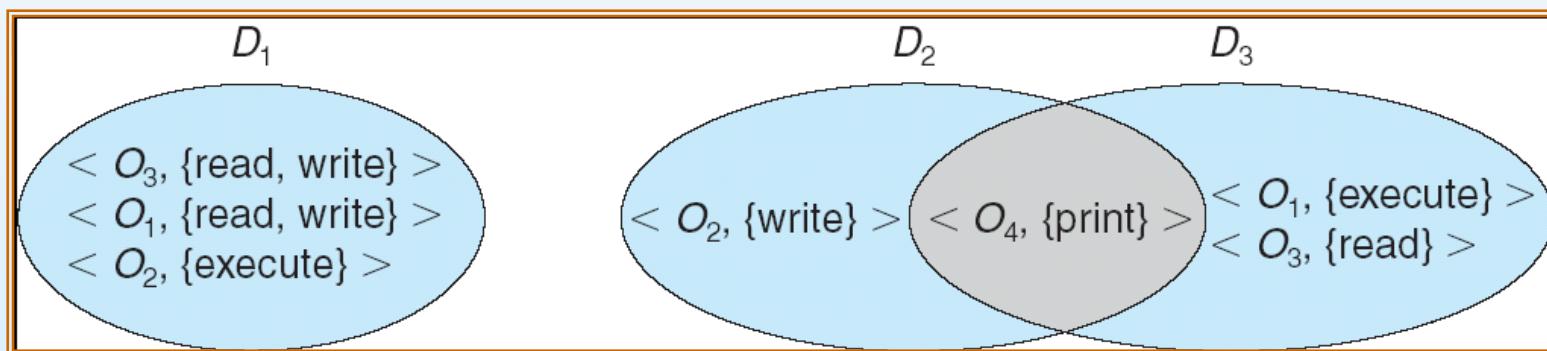
- Guiding principle – principle of least privilege
 - Programs, users and systems should be given just enough privileges to perform their tasks





Domain Structure

- Access-right = $\langle \text{object-name}, \text{rights-set} \rangle$
where *rights-set* is a subset of all valid operations that can be performed on the object.





Domain Implementation (UNIX)

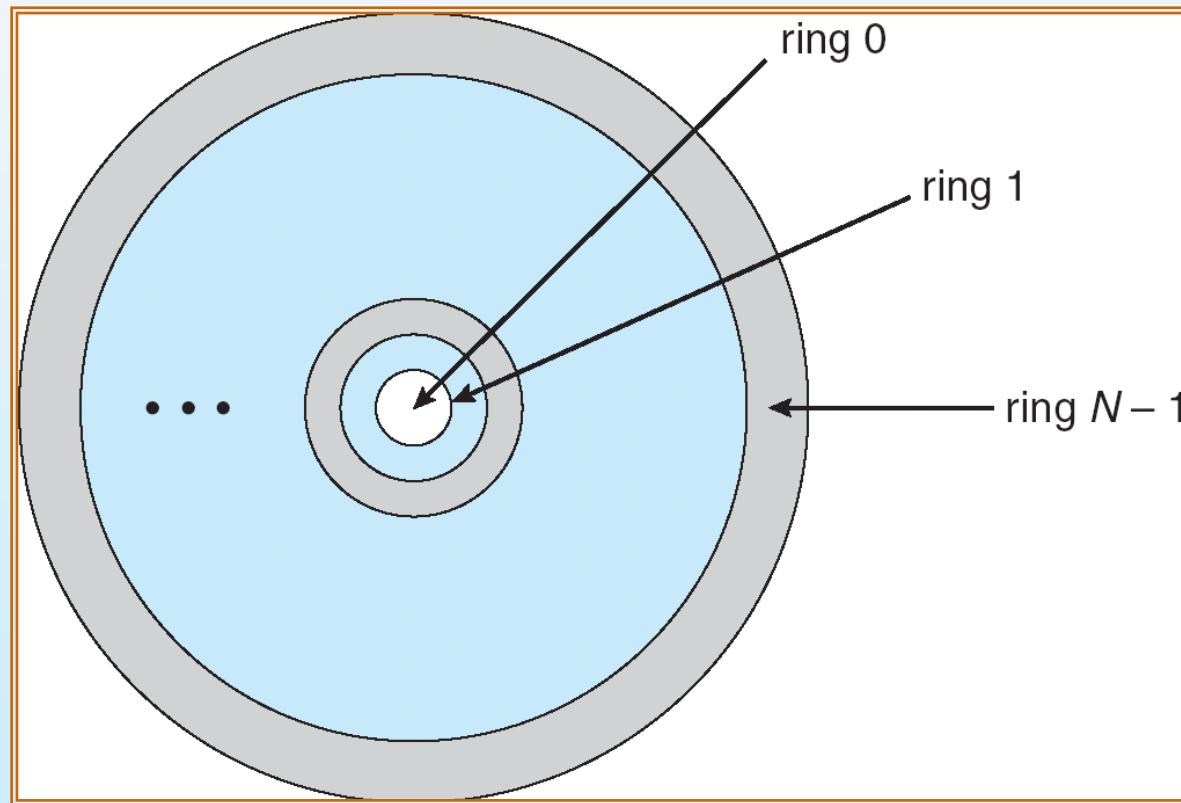
- System consists of 2 domains:
 - User
 - Supervisor
- UNIX
 - Domain = user-id
 - Domain switch accomplished via file system.
 - ▶ Each file has associated with it a domain bit (setuid bit).
 - ▶ When file is executed and setuid = on, then user-id is set to owner of the file being executed. When execution completes user-id is reset.





Domain Implementation (Multics)

- Let D_i and D_j be any two domain rings.
- If $j < i \Rightarrow D_i \subseteq D_j$



Multics Rings





Access Matrix

- View protection as a matrix (*access matrix*)
- Rows represent domains
- Columns represent objects
- $\text{Access}(i, j)$ is the set of operations that a process executing in Domain_i can invoke on Object_j





Access Matrix

object domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Figure A





Use of Access Matrix

- If a process in Domain D_i tries to do “op” on object O_j , then “op” must be in the access matrix.
- Can be expanded to dynamic protection.
 - Operations to add, delete access rights.
 - Special access rights:
 - ▶ *owner of O_i*
 - ▶ *copy op from O_i to O_j*
 - ▶ *control – D_i can modify D_j access rights*
 - ▶ *transfer – switch from domain D_i to D_j*





Use of Access Matrix (Cont.)

- Access matrix design separates mechanism from policy.
 - Mechanism
 - ▶ Operating system provides access-matrix + rules.
 - ▶ If ensures that the matrix is only manipulated by authorized agents and that rules are strictly enforced.
 - Policy
 - ▶ User dictates policy.
 - ▶ Who can access what object and in what mode.





Implementation of Access Matrix

- Each column = Access-control list for one object
Defines who can perform what operation.

Domain 1 = Read, Write

Domain 2 = Read

Domain 3 = Read

:

- Each Row = Capability List (like a key)
For each domain, what operations allowed on what objects.

Object 1 – Read

Object 4 – Read, Write, Execute

Object 5 – Read, Write, Delete, Copy





Access Matrix of Figure A With Domains as Objects

object domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Figure B





Access Matrix with Copy Rights

object domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

object domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)





Access Matrix With *Owner* Rights

object domain \ F _i	F ₁	F ₂	F ₃
D ₁	owner execute		write
D ₂		read* owner	read* owner write
D ₃	execute		

(a)

object domain \ F _i	F ₁	F ₂	F ₃
D ₁	owner execute		write
D ₂		owner read* write*	read* owner write
D ₃		write	write

(b)





Modified Access Matrix of Figure B

object domain \ object domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			





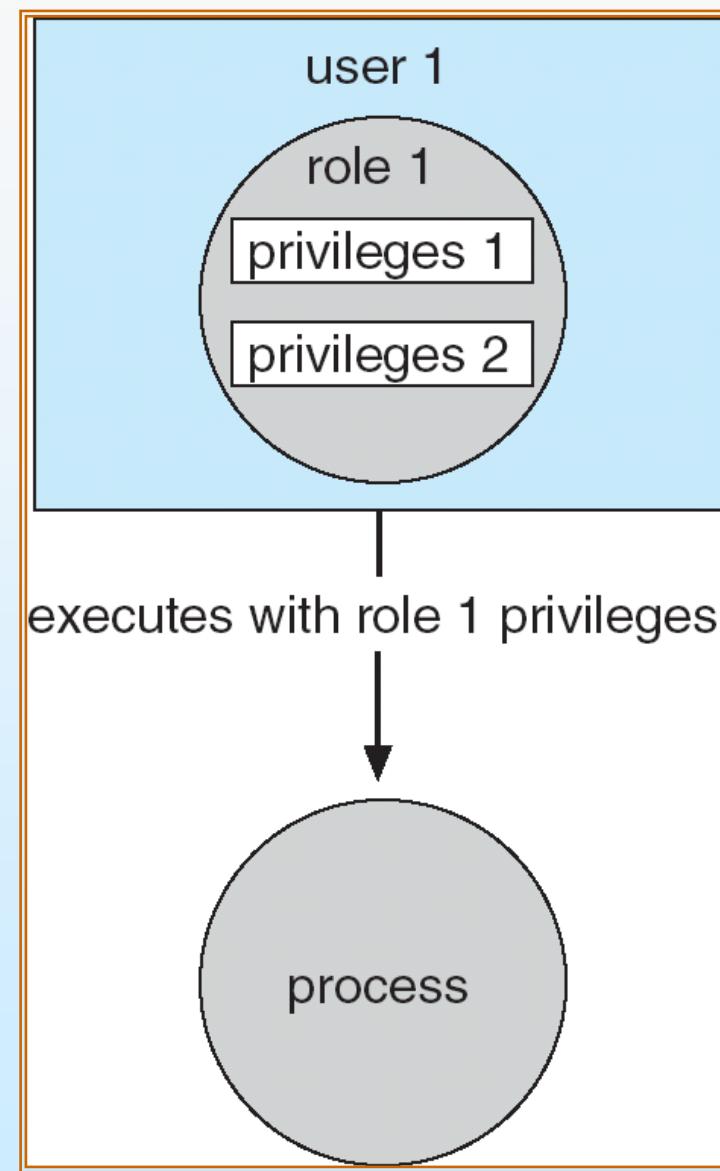
Access Control

- Protection can be applied to non-file resources
- Solaris 10 provides **role-based access control** to implement least privilege
 - Privilege is right to execute system call or use an option within a system call
 - Can be assigned to processes
 - Users assigned roles granting access to privileges and programs





Role-based Access Control in Solaris 10





Revocation of Access Rights

- *Access List* – Delete access rights from access list.
 - Simple
 - Immediate
- *Capability List* – Scheme required to locate capability in the system before capability can be revoked.
 - Reacquisition
 - Back-pointers
 - Indirection
 - Keys





Capability-Based Systems

- Hydra
 - Fixed set of access rights known to and interpreted by the system.
 - Interpretation of user-defined rights performed solely by user's program; system provides access protection for use of these rights.
- Cambridge CAP System
 - Data capability - provides standard read, write, execute of individual storage segments associated with object.
 - Software capability -interpretation left to the subsystem, through its protected procedures.





Language-Based Protection

- Specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources.
- Language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable.
- Interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system.





Protection in Java 2

- Protection is handled by the Java Virtual Machine (JVM)
- A class is assigned a protection domain when it is loaded by the JVM.
- The protection domain indicates what operations the class can (and cannot) perform.
- If a library method is invoked that performs a privileged operation, the stack is inspected to ensure the operation can be performed by the library.





Stack Inspection

protection domain:	untrusted applet	URL loader	networking
socket permission:	none	*.lucent.com:80, connect	any
class:	gui: ... get(url); open(addr); ...	get(URL u): ... doPrivileged { open('proxy.lucent.com:80'); } <request u from proxy> ...	open(Addr a): ... checkPermission (a, connect); connect (a); ...



End of Chapter 14



Chapter 15: Security





Chapter 15: Security

- The Security Problem
- Program Threats
- System and Network Threats
- Cryptography as a Security Tool
- User Authentication
- Implementing Security Defenses
- Firewalling to Protect Systems and Networks
- Computer-Security Classifications
- An Example: Windows XP





Objectives

- To discuss security threats and attacks
- To explain the fundamentals of encryption, authentication, and hashing
- To examine the uses of cryptography in computing
- To describe the various countermeasures to security attacks





The Security Problem

- Security must consider external environment of the system, and protect the system resources
- Intruders (crackers) attempt to breach security
- **Threat** is potential security violation
- **Attack** is attempt to breach security
- Attack can be accidental or malicious
- Easier to protect against accidental than malicious misuse





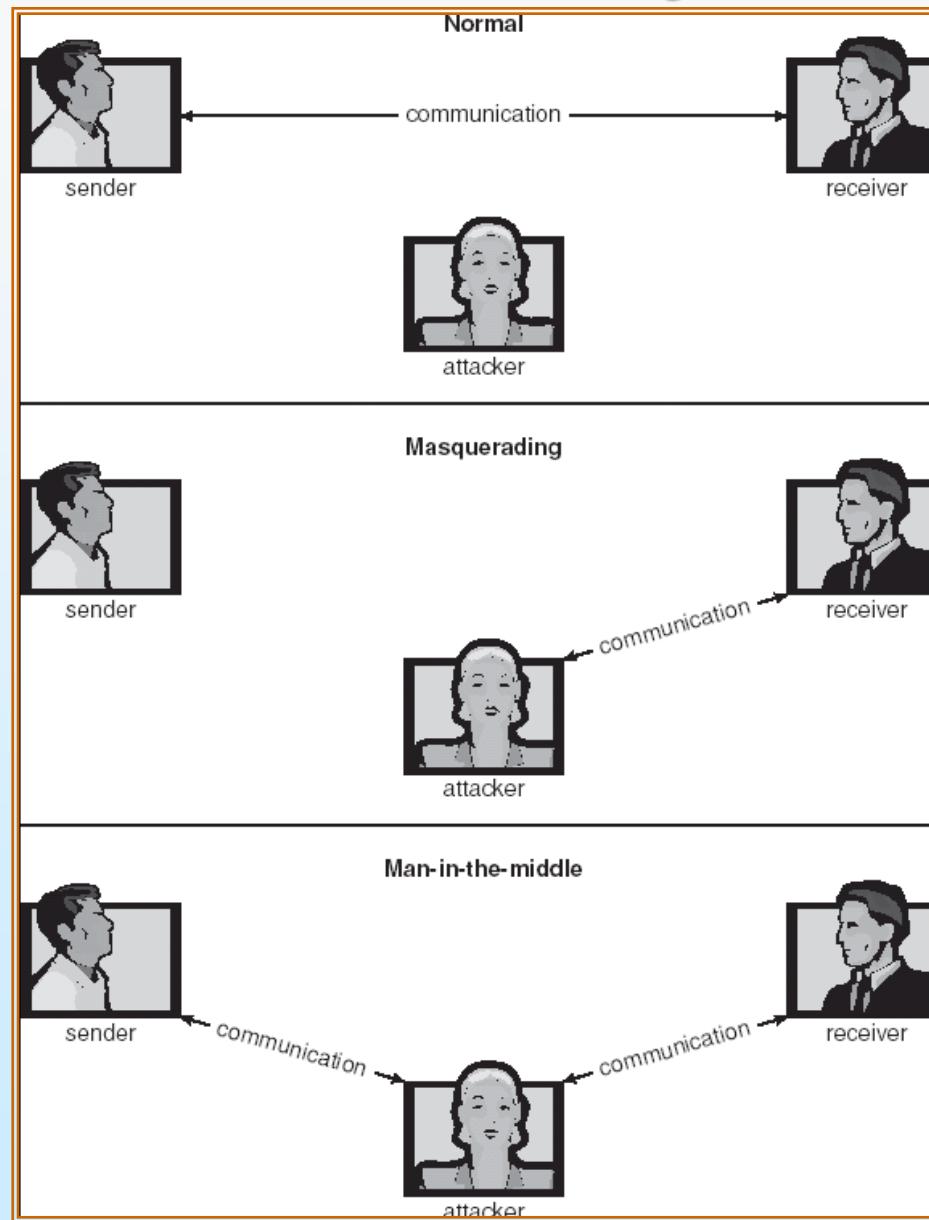
Security Violations

- Categories
 - **Breach of confidentiality**
 - **Breach of integrity**
 - **Breach of availability**
 - **Theft of service**
 - **Denial of service**
- Methods
 - **Masquerading (breach authentication)**
 - **Replay attack**
 - ▶ **Message modification**
 - **Man-in-the-middle attack**
 - **Session hijacking**





Standard Security Attacks





Security Measure Levels

- Security must occur at four levels to be effective:
 - Physical
 - Human
 - ▶ Avoid **social engineering, phishing, dumpster diving**
 - Operating System
 - Network
- Security is as weak as the weakest chain





Program Threats

- Trojan Horse
 - Code segment that misuses its environment
 - Exploits mechanisms for allowing programs written by users to be executed by other users
 - **Spyware, pop-up browser windows, covert channels**
- Trap Door
 - Specific user identifier or password that circumvents normal security procedures
 - Could be included in a compiler
- Logic Bomb
 - Program that initiates a security incident under certain circumstances
- Stack and Buffer Overflow
 - Exploits a bug in a program (overflow either the stack or memory buffers)





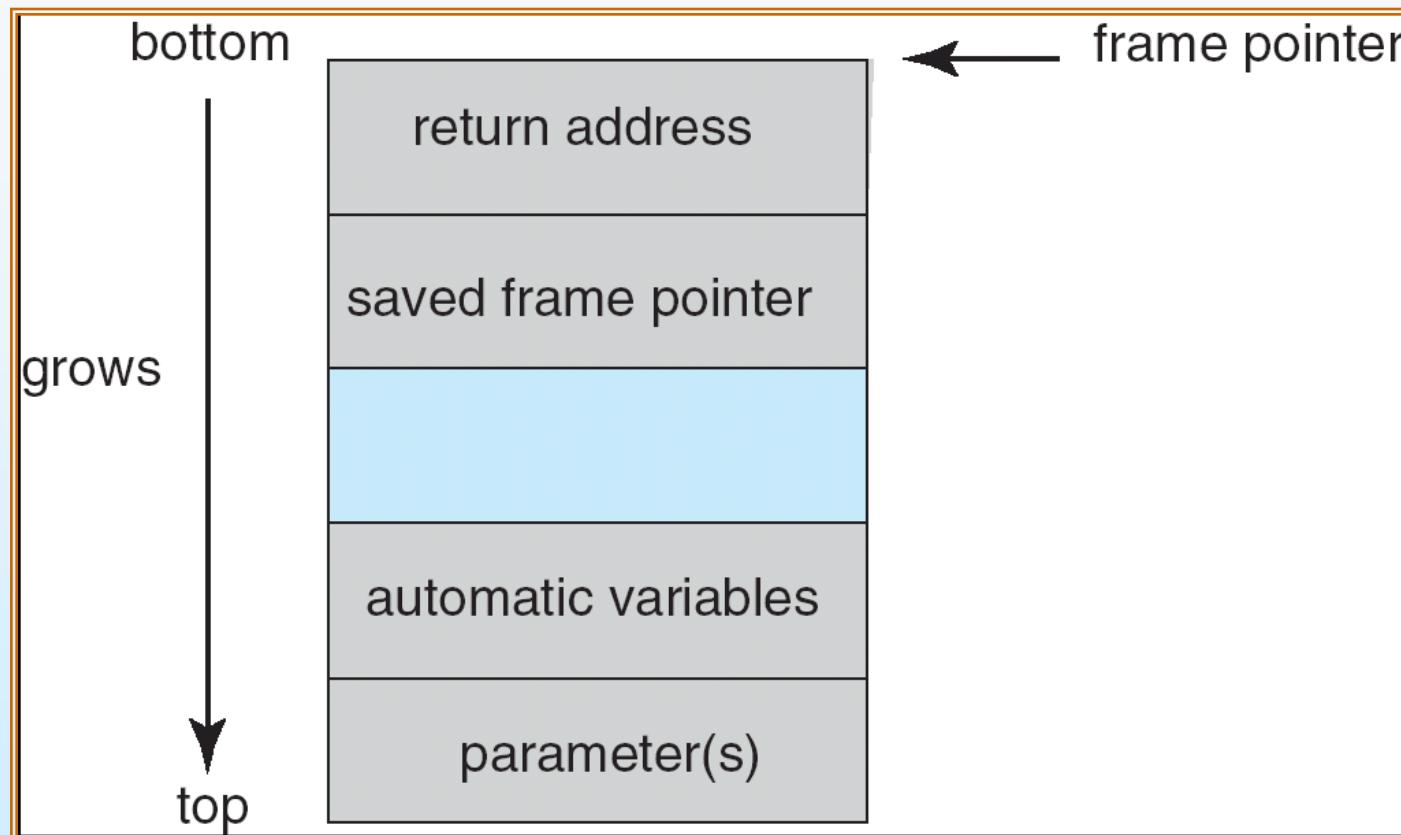
C Program with Buffer-overflow Condition

```
#include <stdio.h>
#define BUFFER SIZE 256
int main(int argc, char *argv[])
{
    char buffer[BUFFER SIZE];
    if (argc < 2)
        return -1;
    else {
        strcpy(buffer, argv[1]);
        return 0;
    }
}
```





Layout of Typical Stack Frame





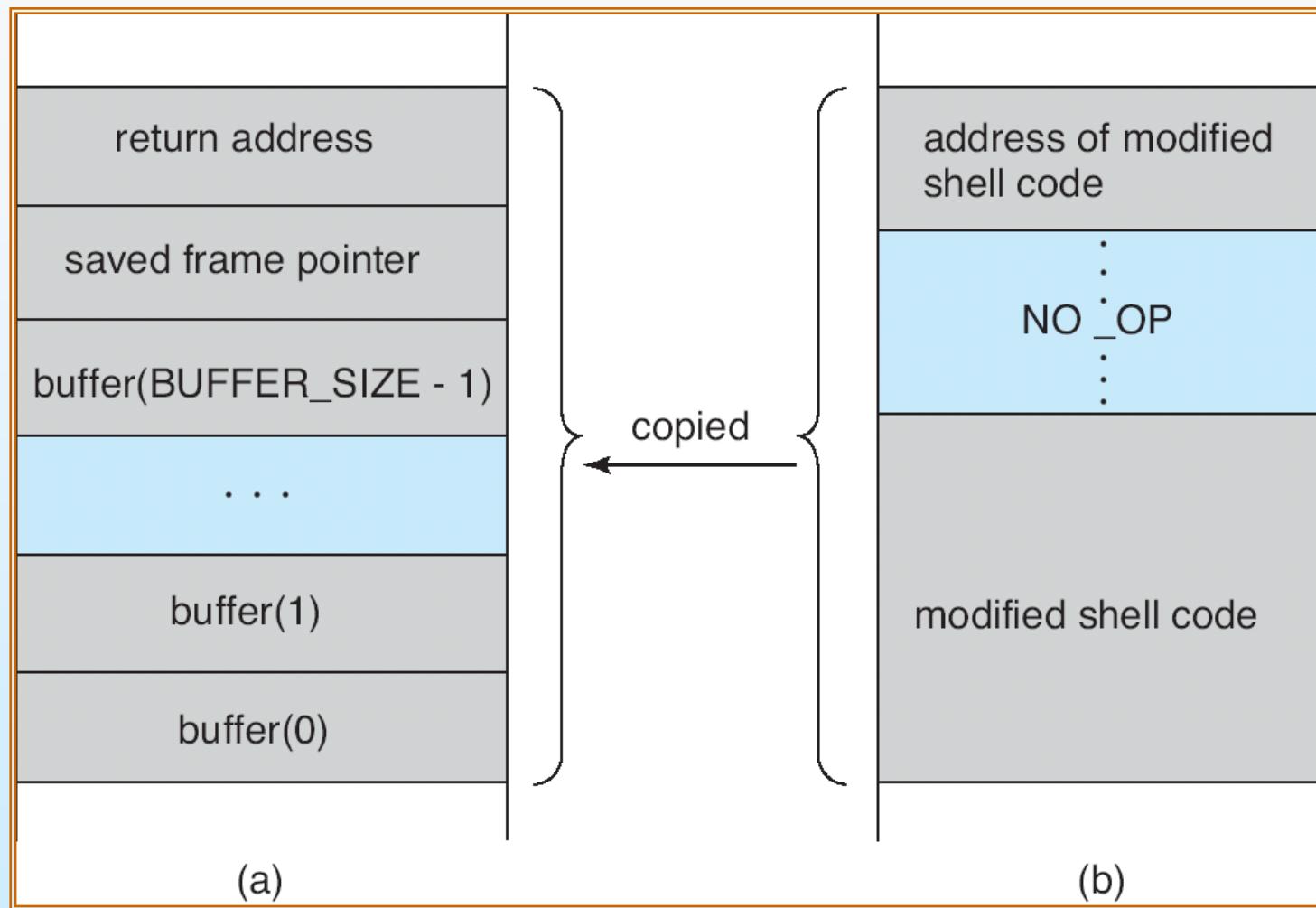
Modified Shell Code

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    execvp(``\bin\sh'', ``\bin \sh'', NULL);
    return 0;
}
```





Hypothetical Stack Frame



Before attack

After attack





Program Threats (Cont.)

□ Viruses

- Code fragment embedded in legitimate program
- Very specific to CPU architecture, operating system, applications
- Usually borne via email or as a macro
 - ▶ Visual Basic Macro to reformat hard drive

```
Sub AutoOpen()  
  
    Dim oFS  
  
    Set oFS =  
        CreateObject("Scripting.FileSystemObject")  
  
    vs = Shell("c:command.com /k format  
        c:", vbHide)  
  
End Sub
```





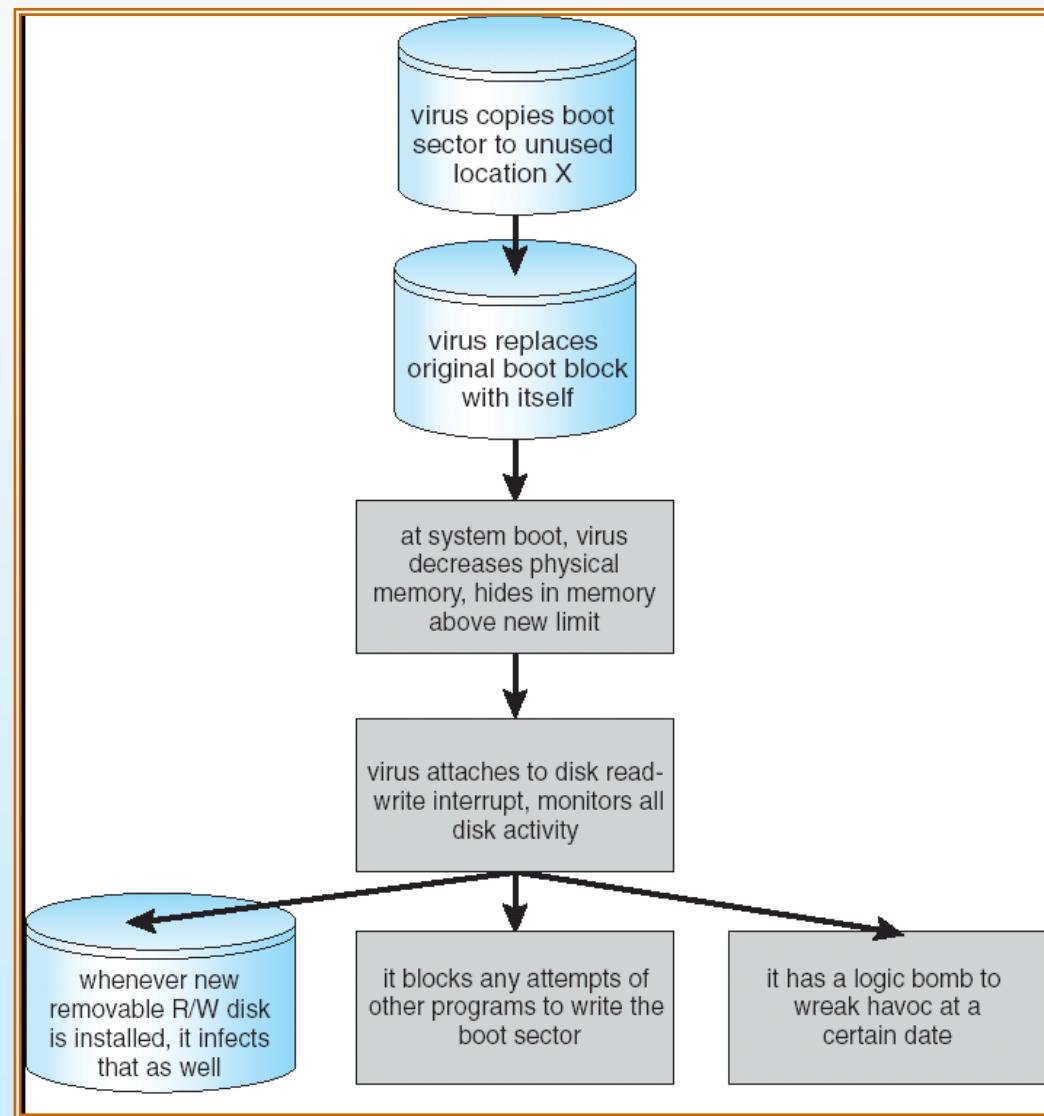
Program Threats (Cont.)

- **Virus dropper** inserts virus onto the system
- Many categories of viruses, literally many thousands of viruses
 - File
 - Boot
 - Macro
 - Source code
 - Polymorphic
 - Encrypted
 - Stealth
 - Tunneling
 - Multipartite
 - Armored





A Boot-sector Computer Virus





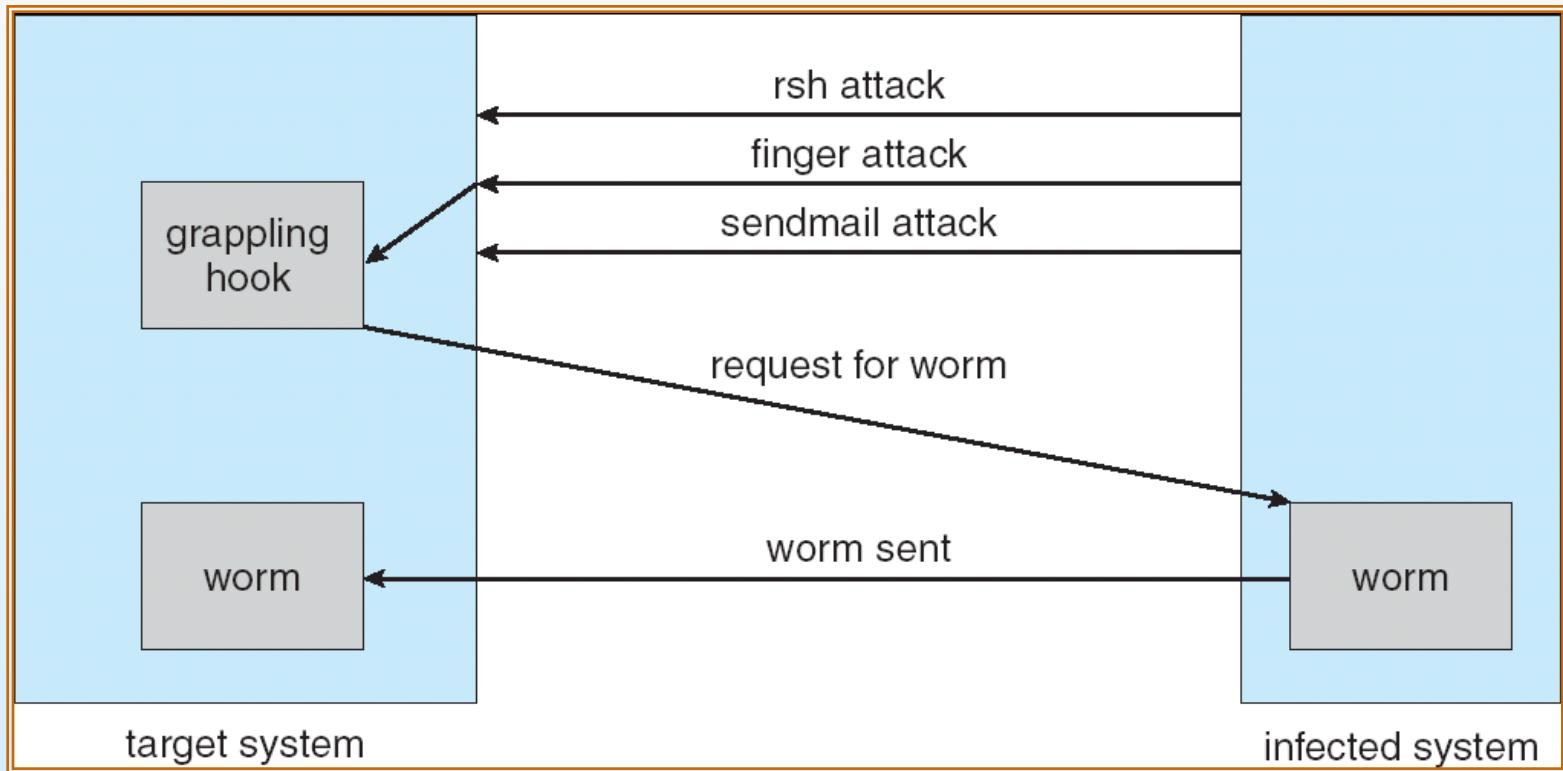
System and Network Threats

- Worms – use **spawn** mechanism; standalone program
- Internet worm
 - Exploited UNIX networking features (remote access) and bugs in *finger* and *sendmail* programs
 - **Grappling hook** program uploaded main worm program
- Port scanning
 - Automated attempt to connect to a range of ports on one or a range of IP addresses
- Denial of Service
 - Overload the targeted computer preventing it from doing any useful work
 - Distributed denial-of-service (**DDOS**) come from multiple sites at once





The Morris Internet Worm





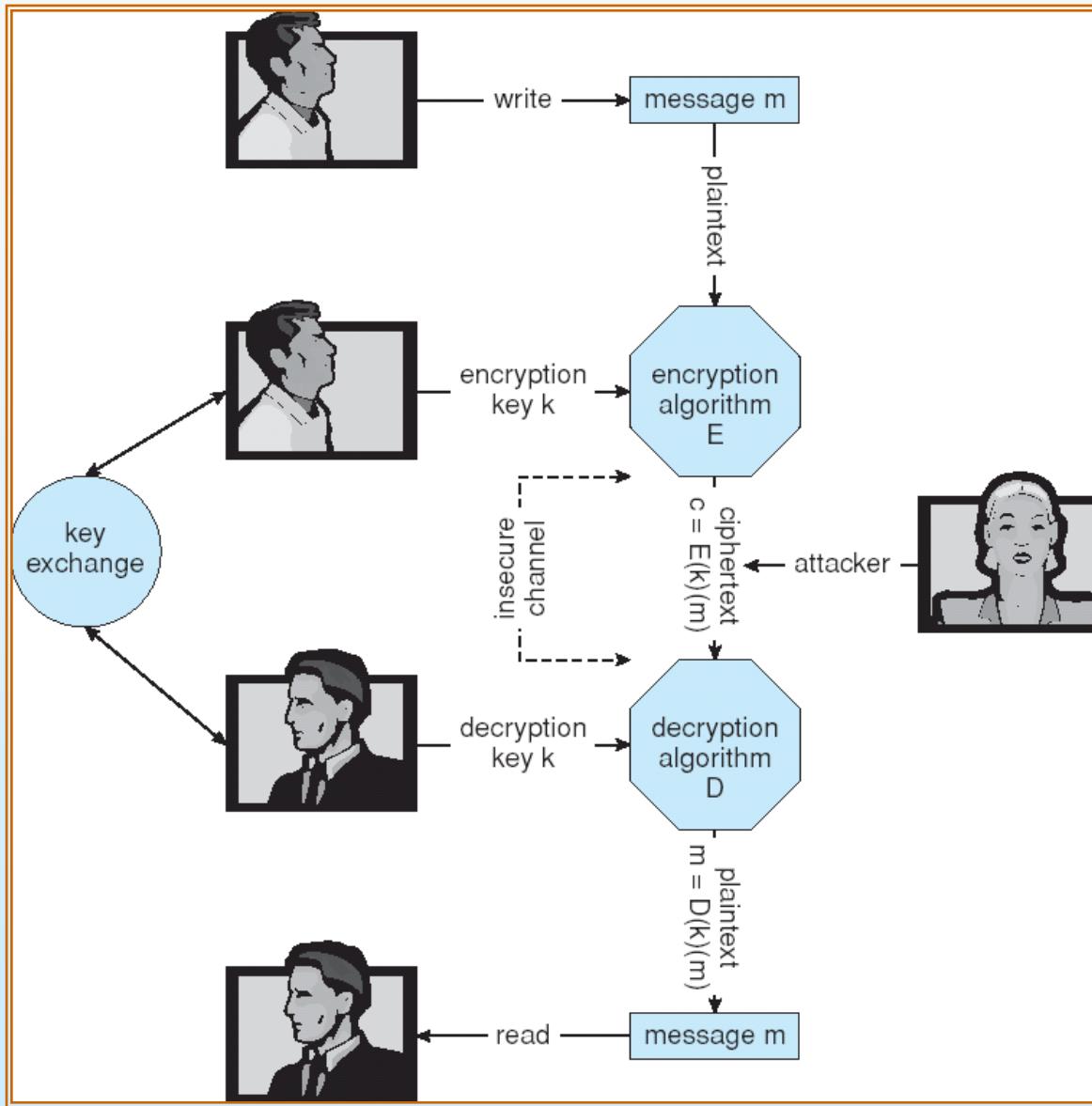
Cryptography as a Security Tool

- Broadest security tool available
 - Source and destination of messages cannot be trusted without cryptography
 - Means to constrain potential senders (*sources*) and / or receivers (*destinations*) of messages
- Based on secrets (**keys**)





Secure Communication over Insecure Medium





Encryption

- Encryption algorithm consists of
 - Set of K keys
 - Set of M Messages
 - Set of C ciphertexts (encrypted messages)
 - A function $E : K \rightarrow (M \rightarrow C)$. That is, for each $k \in K$, $E(k)$ is a function for generating ciphertexts from messages.
 - ▶ Both E and $E(k)$ for any k should be efficiently computable functions.
 - A function $D : K \rightarrow (C \rightarrow M)$. That is, for each $k \in K$, $D(k)$ is a function for generating messages from ciphertexts.
 - ▶ Both D and $D(k)$ for any k should be efficiently computable functions.
- An encryption algorithm must provide this essential property: Given a ciphertext $c \in C$, a computer can compute m such that $E(k)(m) = c$ only if it possesses $D(k)$.
 - Thus, a computer holding $D(k)$ can decrypt ciphertexts to the plaintexts used to produce them, but a computer not holding $D(k)$ cannot decrypt ciphertexts.
 - Since ciphertexts are generally exposed (for example, sent on the network), it is important that it be infeasible to derive $D(k)$ from the ciphertexts





Symmetric Encryption

- Same key used to encrypt and decrypt
 - $E(k)$ can be derived from $D(k)$, and vice versa
- DES is most commonly used symmetric block-encryption algorithm (created by US Govt)
 - Encrypts a block of data at a time
- Triple-DES considered more secure
- Advanced Encryption Standard (**AES**), **twofish** up and coming
- RC4 is most common symmetric stream cipher, but known to have vulnerabilities
 - Encrypts/decrypts a stream of bytes (i.e wireless transmission)
 - Key is a input to psuedo-random-bit generator
 - ▶ Generates an infinite **keystream**





Asymmetric Encryption

- Public-key encryption based on each user having two keys:
 - public key – published key used to encrypt data
 - private key – key known only to individual user used to decrypt data
- Must be an encryption scheme that can be made public without making it easy to figure out the decryption scheme
 - Most common is RSA block cipher
 - Efficient algorithm for testing whether or not a number is prime
 - No efficient algorithm is known for finding the prime factors of a number





Asymmetric Encryption (Cont.)

- Formally, it is computationally infeasible to derive $D(k_d, N)$ from $E(k_e, N)$, and so $E(k_e, N)$ need not be kept secret and can be widely disseminated
 - $E(k_e, N)$ (or just k_e) is the **public key**
 - $D(k_d, N)$ (or just k_d) is the **private key**
 - N is the product of two large, randomly chosen prime numbers p and q (for example, p and q are 512 bits each)
 - Encryption algorithm is $E(k_e, N)(m) = m^{k_e} \text{ mod } N$, where k_e satisfies $k_e k_d \text{ mod } (p-1)(q-1) = 1$
 - The decryption algorithm is then $D(k_d, N)(c) = c^{k_d} \text{ mod } N$





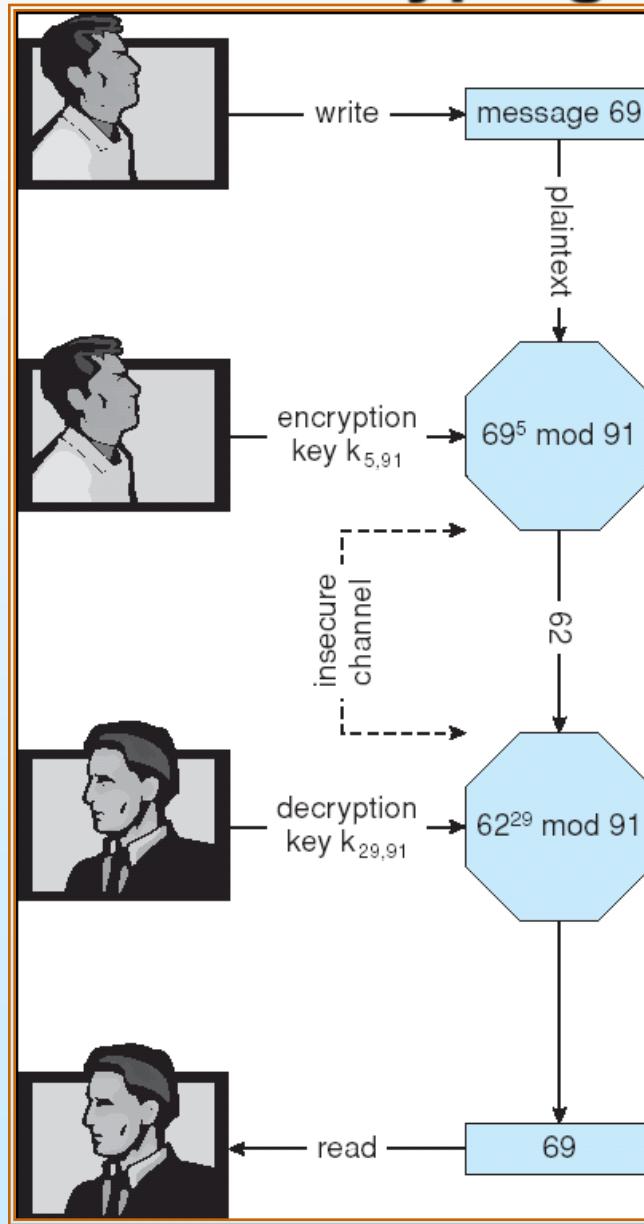
Asymmetric Encryption Example

- For example. make $p = 7$ and $q = 13$
- We then calculate $N = 7 * 13 = 91$ and $(p-1)(q-1) = 72$
- We next select k_e relatively prime to 72 and < 72, yielding 5
- Finally, we calculate k_d such that $k_e k_d \text{ mod } 72 = 1$, yielding 29
- We now have our keys
 - Public key, $k_e, N = 5, 91$
 - Private key, $k_d, N = 29, 91$
- Encrypting the message 69 with the public key results in the ciphertext 62
- Ciphertext can be decoded with the private key
 - Public key can be distributed in cleartext to anyone who wants to communicate with holder of public key





Encryption and Decryption using RSA Asymmetric Cryptography





Cryptography (Cont.)

- Note symmetric cryptography based on transformations, asymmetric based on mathematical functions
 - Asymmetric much more compute intensive
 - Typically not used for bulk data encryption





Authentication

- Constraining set of potential senders of a message
 - Complementary and sometimes redundant to encryption
 - Also can prove message unmodified
- Algorithm components
 - A set K of keys
 - A set M of messages
 - A set A of authenticators
 - A function $S : K \rightarrow (M \rightarrow A)$
 - ▶ That is, for each $k \in K$, $S(k)$ is a function for generating authenticators from messages
 - ▶ Both S and $S(k)$ for any k should be efficiently computable functions
 - A function $V : K \rightarrow (M \times A \rightarrow \{\text{true, false}\})$. That is, for each $k \in K$, $V(k)$ is a function for verifying authenticators on messages
 - ▶ Both V and $V(k)$ for any k should be efficiently computable functions





Authentication (Cont.)

- For a message m , a computer can generate an authenticator $a \in A$ such that $V(k)(m, a) = \text{true}$ only if it possesses $S(k)$
- Thus, computer holding $S(k)$ can generate authenticators on messages so that any other computer possessing $V(k)$ can verify them
- Computer not holding $S(k)$ cannot generate authenticators on messages that can be verified using $V(k)$
- Since authenticators are generally exposed (for example, they are sent on the network with the messages themselves), it must not be feasible to derive $S(k)$ from the authenticators





Authentication – Hash Functions

- Basis of authentication
- Creates small, fixed-size block of data (**message digest, hash value**) from m
- Hash Function H must be collision resistant on m
 - Must be infeasible to find an $m' \neq m$ such that $H(m) = H(m')$
- If $H(m) = H(m')$, then $m = m'$
 - The message has not been modified
- Common message-digest functions include **MD5**, which produces a 128-bit hash, and **SHA-1**, which outputs a 160-bit hash





Authentication - MAC

- Symmetric encryption used in **message-authentication code (MAC)** authentication algorithm
- Simple example:
 - MAC defines $S(k)(m) = f(k, H(m))$
 - ▶ Where f is a function that is one-way on its first argument
 - k cannot be derived from $f(k, H(m))$
 - ▶ Because of the collision resistance in the hash function, reasonably assured no other message could create the same MAC
 - ▶ A suitable verification algorithm is $V(k)(m, a) \equiv (f(k, m) = a)$
 - ▶ Note that k is needed to compute both $S(k)$ and $V(k)$, so anyone able to compute one can compute the other





Authentication – Digital Signature

- Based on asymmetric keys and digital signature algorithm
- Authenticators produced are **digital signatures**
- In a digital-signature algorithm, computationally infeasible to derive $S(k_s)$ from $V(k_v)$
 - V is a one-way function
 - Thus, k_v is the public key and k_s is the private key
- Consider the RSA digital-signature algorithm
 - Similar to the RSA encryption algorithm, but the key use is reversed
 - Digital signature of message $S(k_s)(m) = H(m)^{k_s} \text{ mod } N$
 - The key k_s again is a pair d, N , where N is the product of two large, randomly chosen prime numbers p and q
 - Verification algorithm is $V(k_v)(m, a) \equiv (a^{k_v} \text{ mod } N = H(m))$
 - ▶ Where k_v satisfies $k_v k_s \text{ mod } (p - 1)(q - 1) = 1$





Authentication (Cont.)

- Why authentication if a subset of encryption?
 - Fewer computations (except for RSA digital signatures)
 - Authenticator usually shorter than message
 - Sometimes want authentication but not confidentiality
 - ▶ Signed patches et al
 - Can be basis for **non-repudiation**



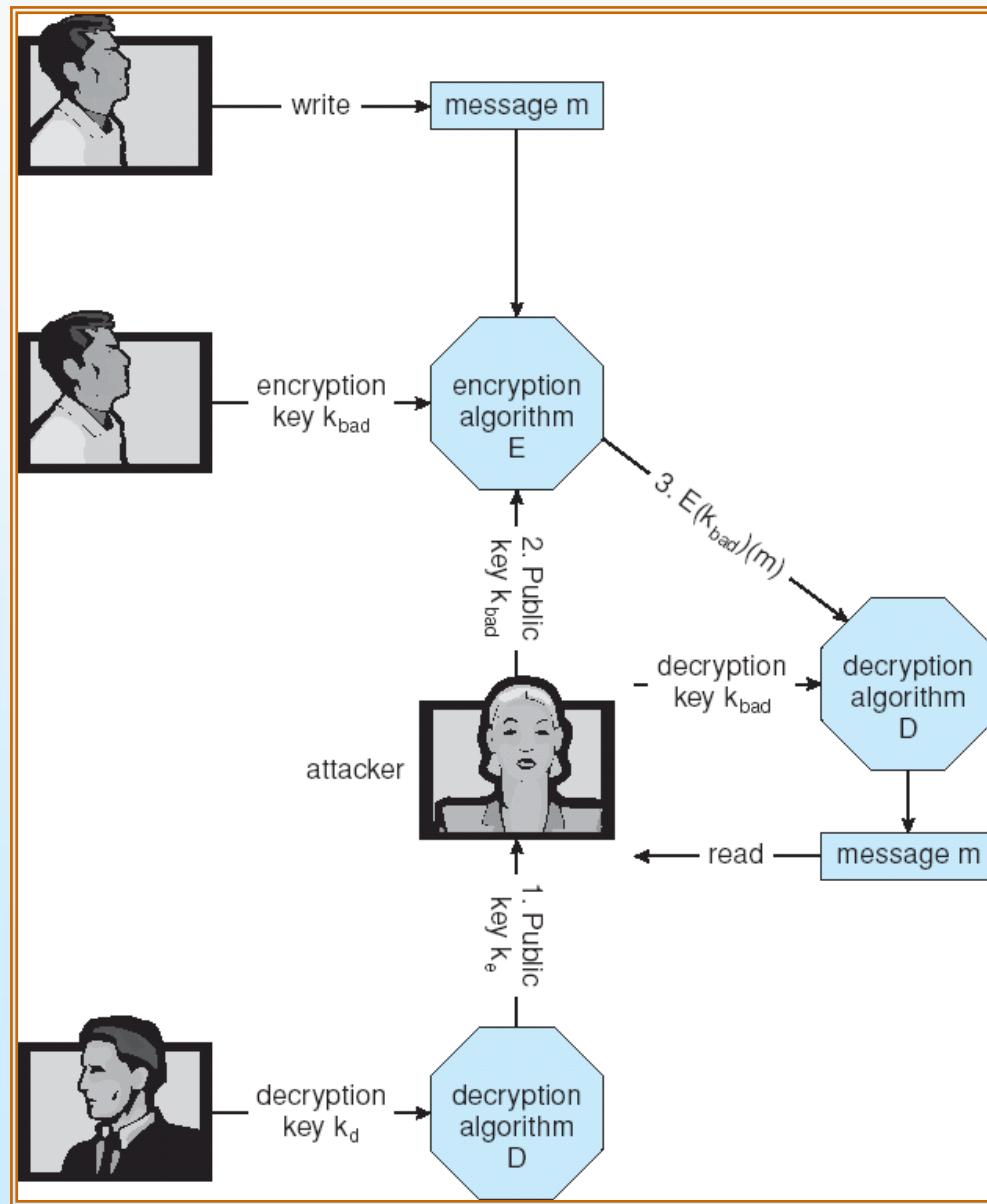


Key Distribution

- Delivery of symmetric key is huge challenge
 - Sometimes done **out-of-band**
- Asymmetric keys can proliferate – stored on **key ring**
 - Even asymmetric key distribution needs care – man-in-the-middle attack



Man-in-the-middle Attack on Asymmetric Cryptography





Digital Certificates

- Proof of who or what owns a public key
- Public key digitally signed a trusted party
- Trusted party receives proof of identification from entity and certifies that public key belongs to entity
- Certificate authority are trusted party – their public keys included with web browser distributions
 - They vouch for other authorities via digitally signing their keys, and so on





Encryption Example - SSL

- Insertion of cryptography at one layer of the ISO network model (the transport layer)
- SSL – Secure Socket Layer (also called TLS)
- Cryptographic protocol that limits two computers to only exchange messages with each other
 - Very complicated, with many variations
- Used between web servers and browsers for secure communication (credit card numbers)
- The server is verified with a **certificate** assuring client is talking to correct server
- Asymmetric cryptography used to establish a secure **session key** (symmetric encryption) for bulk of communication during session
- Communication between each computer theb uses symmetric key cryptography





User Authentication

- Crucial to identify user correctly, as protection systems depend on user ID
- User identity most often established through *passwords*, can be considered a special case of either keys or capabilities
 - Also can include something user has and /or a user attribute
- Passwords must be kept secret
 - Frequent change of passwords
 - Use of “non-guessable” passwords
 - Log all invalid access attempts
- Passwords may also either be encrypted or allowed to be used only once





Implementing Security Defenses

- **Defense in depth** is most common security theory – multiple layers of security
- Security policy describes what is being secured
- Vulnerability assessment compares real state of system / network compared to security policy
- Intrusion detection endeavors to detect attempted or successful intrusions
 - **Signature-based** detection spots known bad patterns
 - **Anomaly detection** spots differences from normal behavior
 - ▶ Can detect **zero-day** attacks
 - **False-positives** and **false-negatives** a problem
- Virus protection
- Auditing, accounting, and logging of all or specific system or network activities





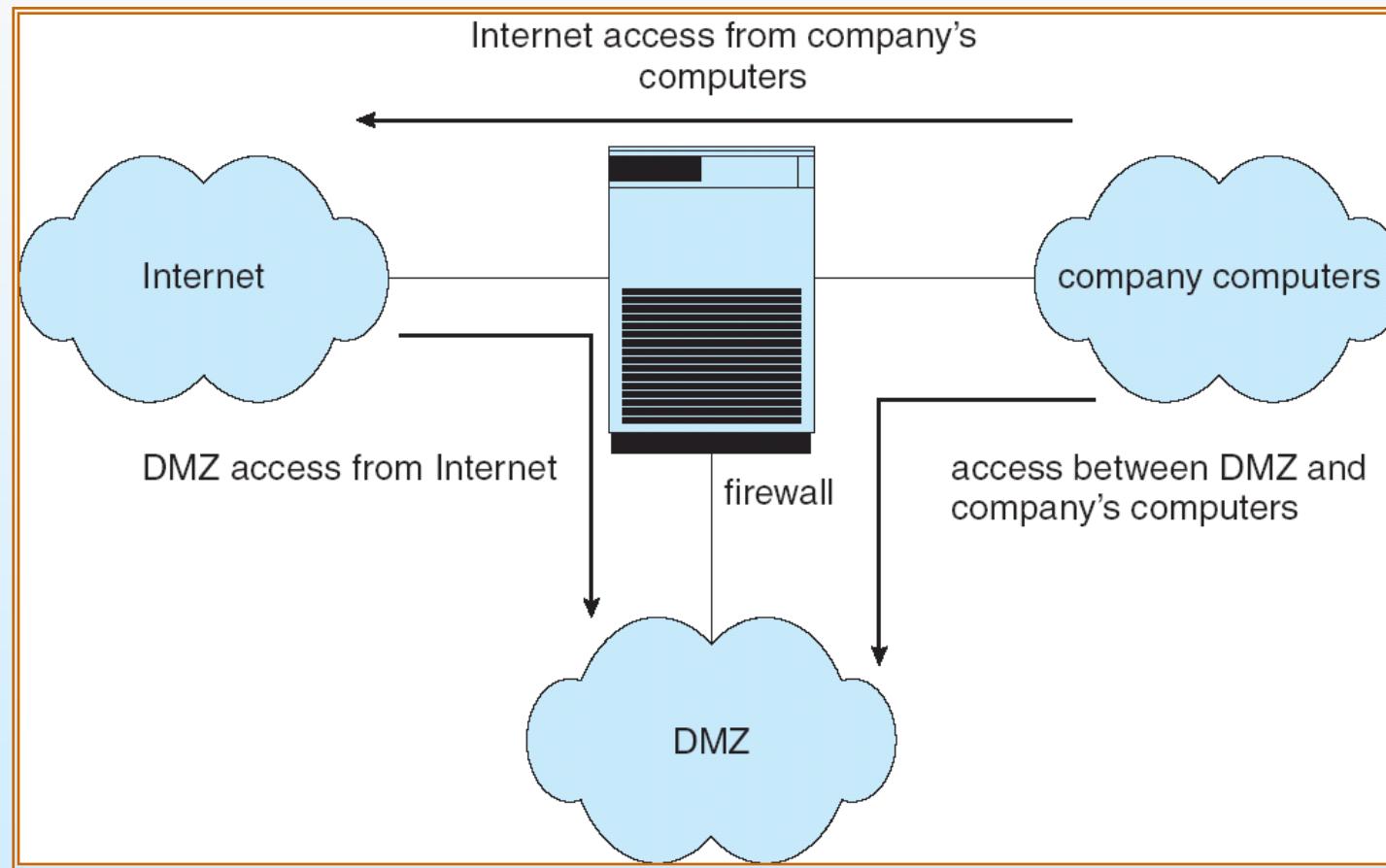
Firewalling to Protect Systems and Networks

- A network firewall is placed between trusted and untrusted hosts
 - The firewall limits network access between these two security domains
- Can be tunneled or spoofed
 - Tunneling allows disallowed protocol to travel within allowed protocol (i.e. telnet inside of HTTP)
 - Firewall rules typically based on host name or IP address which can be spoofed
- **Personal firewall** is software layer on given host
 - Can monitor / limit traffic to and from the host
- **Application proxy firewall** understands application protocol and can control them (i.e. SMTP)
- **System-call firewall** monitors all important system calls and apply rules to them (i.e. this program can execute that system call)





Network Security Through Domain Separation Via Firewall





Computer Security Classifications

- U.S. Department of Defense outlines four divisions of computer security: **A**, **B**, **C**, and **D**.
- **D** – Minimal security.
- **C** – Provides discretionary protection through auditing. Divided into **C1** and **C2**. **C1** identifies cooperating users with the same level of protection. **C2** allows user-level access control.
- **B** – All the properties of **C**, however each object may have unique sensitivity labels. Divided into **B1**, **B2**, and **B3**.
- **A** – Uses formal design and verification techniques to ensure security.





Example: Windows XP

- Security is based on user accounts
 - Each user has unique security ID
 - Login to ID creates **security access token**
 - ▶ Includes security ID for user, for user's groups, and special privileges
 - ▶ Every process gets copy of token
 - ▶ System checks token to determine if access allowed or denied
- Uses a subject model to ensure access security. A subject tracks and manages permissions for each program that a user runs
- Each object in Windows XP has a security attribute defined by a security descriptor
 - For example, a file has a security descriptor that indicates the access permissions for all users



End of Chapter 15



Module 16: Distributed System Structures





Module 16: Network Structures

- Motivation
- Types of Distributed Operating Systems
- Network Structure
- Network Topology
- Communication Structure
- Communication Protocols
- Robustness
- Design Issues
- An Example: Networking





Chapter Objectives

- To provide a high-level overview of distributed systems and the networks that interconnect them
- To discuss the general structure of distributed operating systems





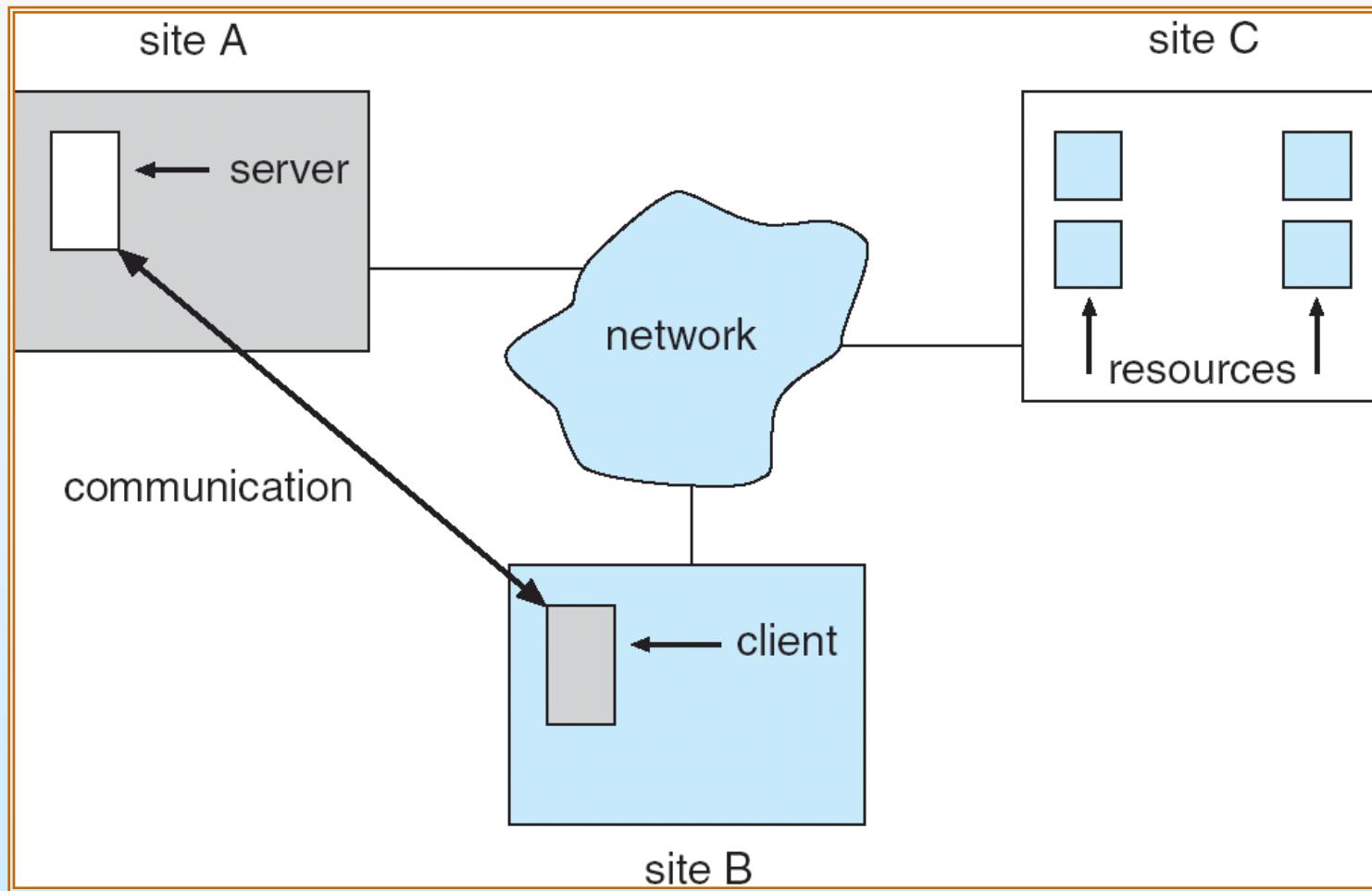
Motivation

- **Distributed system** is collection of loosely coupled processors interconnected by a communications network
- Processors variously called *nodes*, *computers*, *machines*, *hosts*
 - *Site* is location of the processor
- Reasons for distributed systems
 - Resource sharing
 - ▶ sharing and printing files at remote sites
 - ▶ processing information in a distributed database
 - ▶ using remote specialized hardware devices
 - Computation speedup – **load sharing**
 - Reliability – detect and recover from site failure, function transfer, reintegrate failed site
 - Communication – message passing





A Distributed System





Types of Distributed Operating Systems

- Network Operating Systems
- Distributed Operating Systems





Network-Operating Systems

- Users are aware of multiplicity of machines. Access to resources of various machines is done explicitly by:
 - Remote logging into the appropriate remote machine (telnet, ssh)
 - Transferring data from remote machines to local machines, via the File Transfer Protocol (FTP) mechanism





Distributed-Operating Systems

- Users not aware of multiplicity of machines
 - Access to remote resources similar to access to local resources
- Data Migration – transfer data by transferring entire file, or transferring only those portions of the file necessary for the immediate task
- Computation Migration – transfer the computation, rather than the data, across the system





Distributed-Operating Systems (Cont.)

- Process Migration – execute an entire process, or parts of it, at different sites
 - Load balancing – distribute processes across network to even the workload
 - Computation speedup – subprocesses can run concurrently on different sites
 - Hardware preference – process execution may require specialized processor
 - Software preference – required software may be available at only a particular site
 - Data access – run process remotely, rather than transfer all data locally





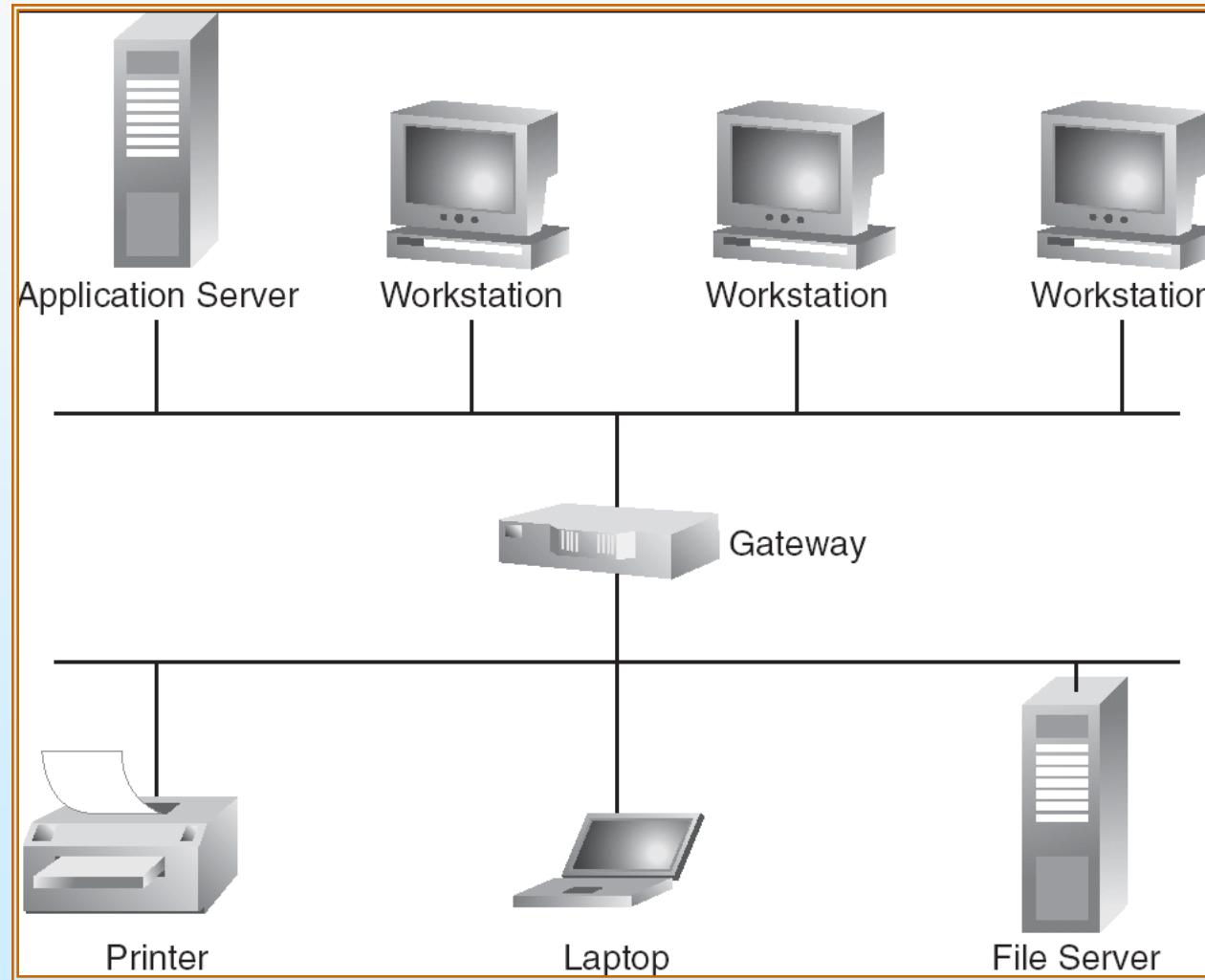
Network Structure

- Local-Area Network (LAN) – designed to cover small geographical area.
 - Multiaccess bus, ring, or star network
 - Speed \approx 10 megabits/second, or higher
 - Broadcast is fast and cheap
 - Nodes:
 - ▶ usually workstations and/or personal computers
 - ▶ a few (usually one or two) mainframes





Depiction of typical LAN





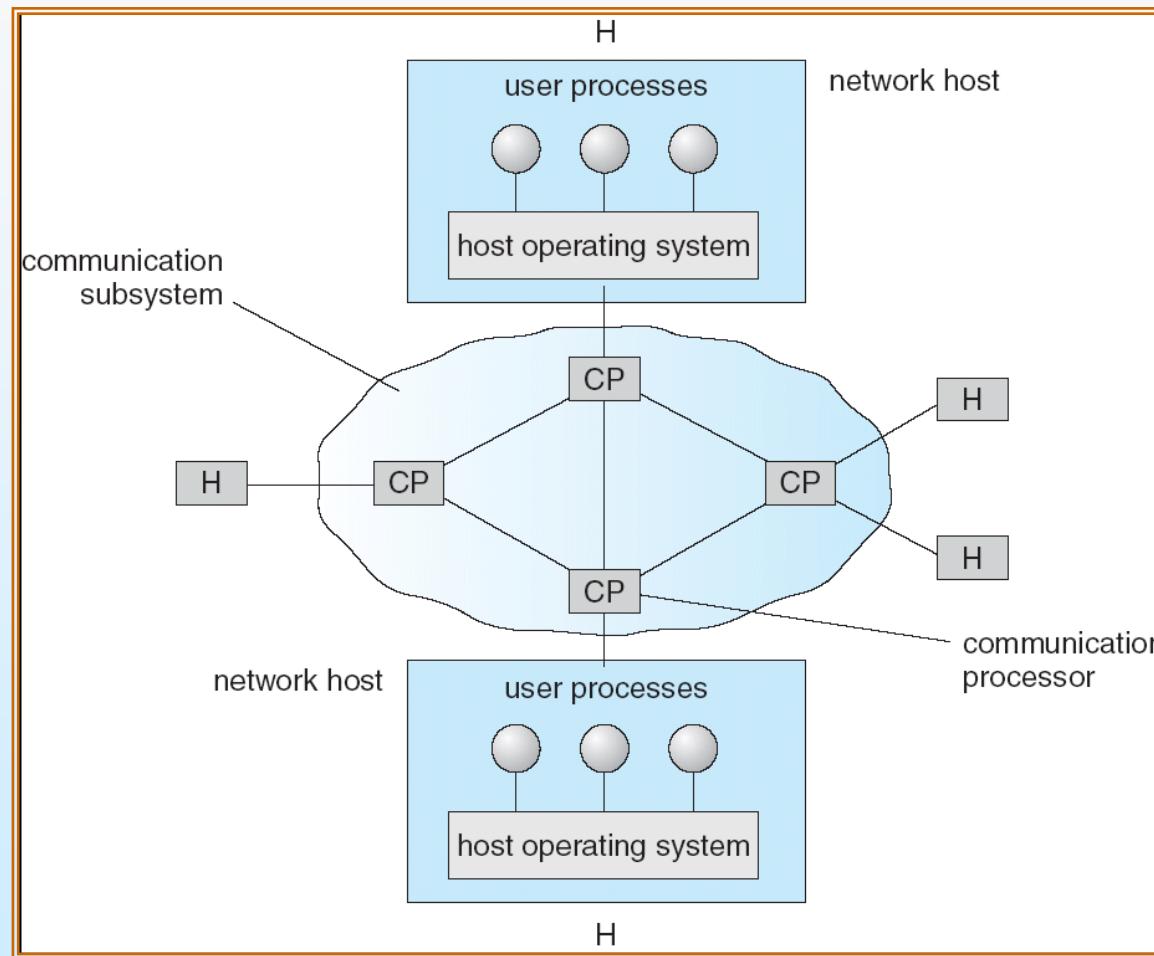
Network Types (Cont.)

- Wide-Area Network (WAN) – links geographically separated sites
 - Point-to-point connections over long-haul lines (often leased from a phone company)
 - Speed \approx 100 kilobits/second
 - Broadcast usually requires multiple messages
 - Nodes:
 - ▶ usually a high percentage of mainframes





Communication Processors in a Wide-Area Network





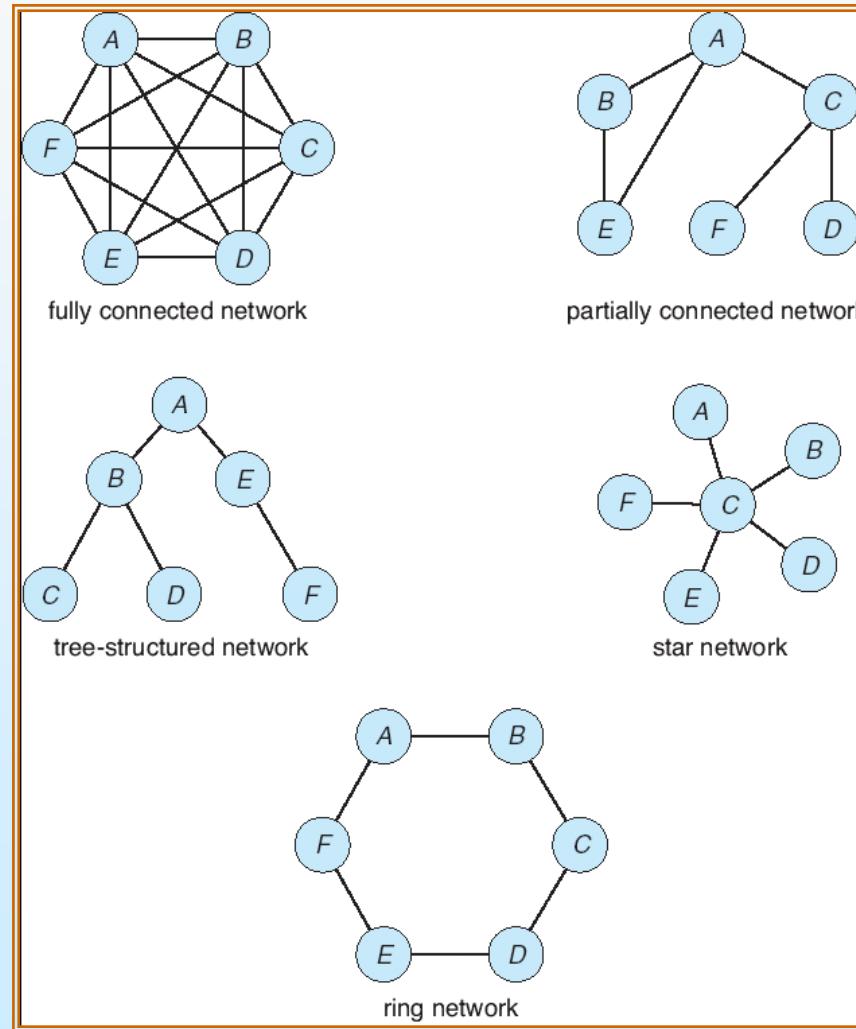
Network Topology

- Sites in the system can be physically connected in a variety of ways; they are compared with respect to the following criteria:
 - **Basic cost** - How expensive is it to link the various sites in the system?
 - **Communication cost** - How long does it take to send a message from site *A* to site *B*?
 - **Reliability** - If a link or a site in the system fails, can the remaining sites still communicate with each other?
- The various topologies are depicted as graphs whose nodes correspond to sites
 - An edge from node *A* to node *B* corresponds to a direct connection between the two sites
- The following six items depict various network topologies





Network Topology





Communication Structure

The design of a *communication* network must address four basic issues:

- **Naming and name resolution** - How do two processes locate each other to communicate?
- **Routing strategies** - How are messages sent through the network?
- **Connection strategies** - How do two processes send a sequence of messages?
- **Contention** - The network is a shared resource, so how do we resolve conflicting demands for its use?





Naming and Name Resolution

- Name systems in the network
- Address messages with the process-id
- Identify processes on remote systems by
 - <host-name, identifier> pair
- *Domain name service (DNS)* – specifies the naming structure of the hosts, as well as name to address resolution (Internet)





Routing Strategies

- **Fixed routing** - A path from *A* to *B* is specified in advance; path changes only if a hardware failure disables it
 - Since the shortest path is usually chosen, communication costs are minimized
 - Fixed routing cannot adapt to load changes
 - Ensures that messages will be delivered in the order in which they were sent
- **Virtual circuit** - A path from *A* to *B* is fixed for the duration of one session. Different sessions involving messages from *A* to *B* may have different paths
 - Partial remedy to adapting to load changes
 - Ensures that messages will be delivered in the order in which they were sent





Routing Strategies (Cont.)

- **Dynamic routing** - The path used to send a message from site *A* to site *B* is chosen only when a message is sent
 - Usually a site sends a message to another site on the link least used at that particular time
 - Adapts to load changes by avoiding routing messages on heavily used path
 - Messages may arrive out of order
 - ▶ This problem can be remedied by appending a sequence number to each message





Connection Strategies

- **Circuit switching** - A permanent physical link is established for the duration of the communication (i.e., telephone system)
- **Message switching** - A temporary link is established for the duration of one message transfer (i.e., post-office mailing system)
- **Packet switching** - Messages of variable length are divided into fixed-length packets which are sent to the destination
 - Each packet may take a different path through the network
 - The packets must be reassembled into messages as they arrive
- Circuit switching requires setup time, but incurs less overhead for shipping each message, and may waste network bandwidth
 - Message and packet switching require less setup time, but incur more overhead per message





Contention

Several sites may want to transmit information over a link simultaneously. Techniques to avoid repeated collisions include:

- **CSMA/CD** - Carrier sense with multiple access (CSMA); collision detection (CD)
 - A site determines whether another message is currently being transmitted over that link. If two or more sites begin transmitting at exactly the same time, then they will register a CD and will stop transmitting
 - When the system is very busy, many collisions may occur, and thus performance may be degraded
- CSMA/CD is used successfully in the Ethernet system, the most common network system





Contention (Cont.)

- **Token passing** - A unique message type, known as a token, continuously circulates in the system (usually a ring structure)
 - A site that wants to transmit information must wait until the token arrives
 - When the site completes its round of message passing, it retransmits the token
 - A token-passing scheme is used by some IBM and HP/Apollo systems
- **Message slots** - A number of fixed-length message slots continuously circulate in the system (usually a ring structure)
 - Since a slot can contain only fixed-sized messages, a single logical message may have to be broken down into a number of smaller packets, each of which is sent in a separate slot
 - This scheme has been adopted in the experimental Cambridge Digital Communication Ring





Communication Protocol

The communication network is partitioned into the following multiple layers:

- **Physical layer** – handles the mechanical and electrical details of the physical transmission of a bit stream
- **Data-link layer** – handles the *frames*, or fixed-length parts of packets, including any error detection and recovery that occurred in the physical layer
- **Network layer** – provides connections and routes packets in the communication network, including handling the address of outgoing packets, decoding the address of incoming packets, and maintaining routing information for proper response to changing load levels





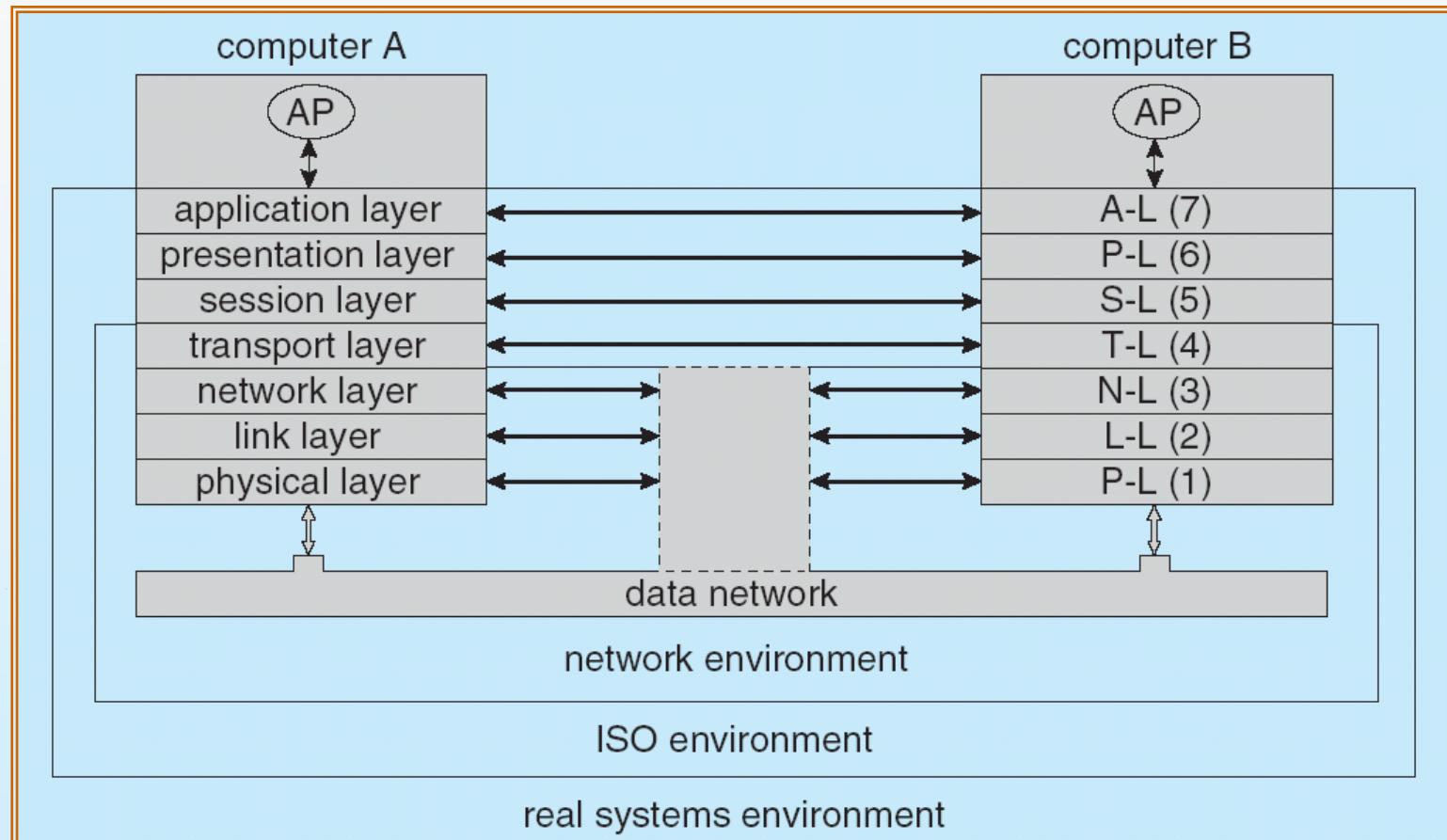
Communication Protocol (Cont.)

- **Transport layer** – responsible for low-level network access and for message transfer between clients, including partitioning messages into packets, maintaining packet order, controlling flow, and generating physical addresses
- **Session layer** – implements sessions, or process-to-process communications protocols
- **Presentation layer** – resolves the differences in formats among the various sites in the network, including character conversions, and half duplex/full duplex (echoing)
- **Application layer** – interacts directly with the users' deals with file transfer, remote-login protocols and electronic mail, as well as schemas for distributed databases



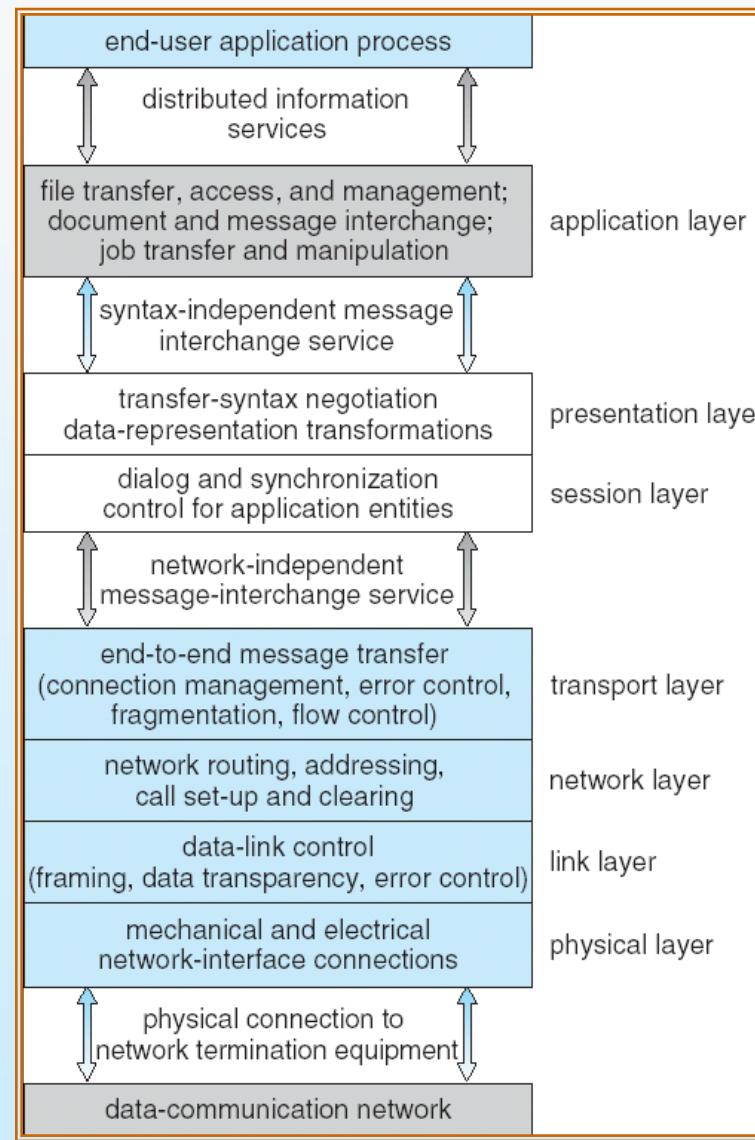


Communication Via ISO Network Model



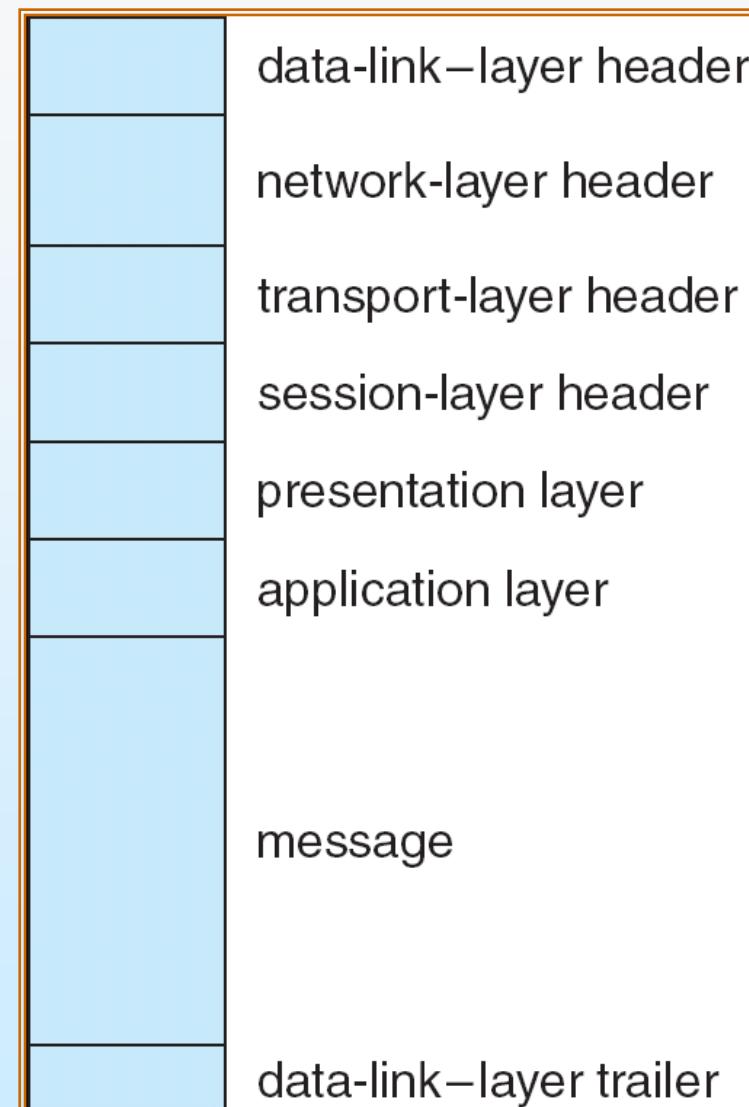


The ISO Protocol Layer



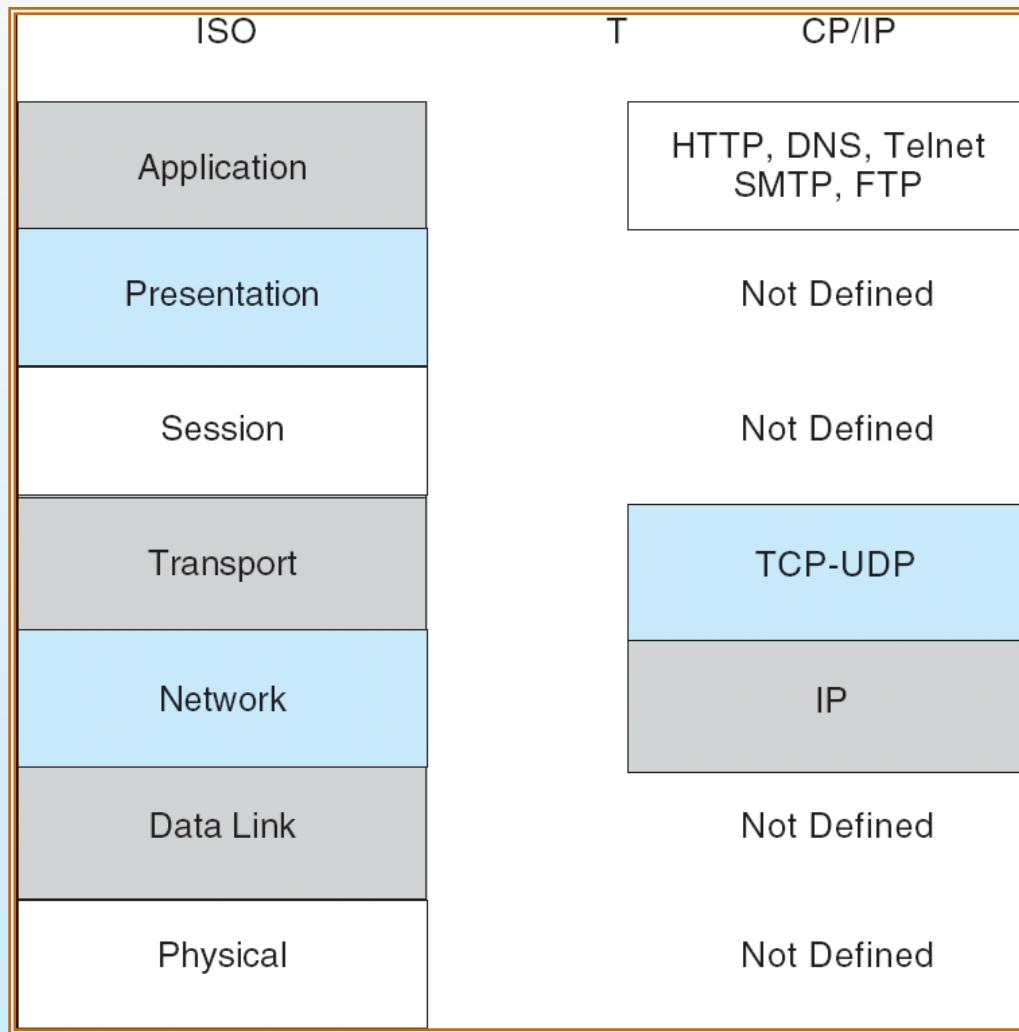


The ISO Network Message





The TCP/IP Protocol Layers





Robustness

- Failure detection
- Reconfiguration





Failure Detection

- Detecting hardware failure is difficult
- To detect a link failure, a handshaking protocol can be used
- Assume Site A and Site B have established a link
 - At fixed intervals, each site will exchange an *I-am-up* message indicating that they are up and running
- If Site A does not receive a message within the fixed interval, it assumes either (a) the other site is not up or (b) the message was lost
- Site A can now send an *Are-you-up?* message to Site B
- If Site A does not receive a reply, it can repeat the message or try an alternate route to Site B





Failure Detection (cont)

- If Site A does not ultimately receive a reply from Site B, it concludes some type of failure has occurred
- Types of failures:
 - Site B is down
 - The direct link between A and B is down
 - The alternate link from A to B is down
 - The message has been lost
- However, Site A cannot determine exactly **why** the failure has occurred





Reconfiguration

- When Site A determines a failure has occurred, it must reconfigure the system:
 1. If the link from A to B has failed, this must be broadcast to every site in the system
 2. If a site has failed, every other site must also be notified indicating that the services offered by the failed site are no longer available
- When the link or the site becomes available again, this information must again be broadcast to all other sites





Design Issues

- **Transparency** – the distributed system should appear as a conventional, centralized system to the user
- **Fault tolerance** – the distributed system should continue to function in the face of failure
- **Scalability** – as demands increase, the system should easily accept the addition of new resources to accommodate the increased demand
- **Clusters** – a collection of semi-autonomous machines that acts as a single system





Example: Networking

- The transmission of a network packet between hosts on an Ethernet network
- Every host has a unique IP address and a corresponding Ethernet (MAC) address
- Communication requires both addresses
- Domain Name Service (DNS) can be used to acquire IP addresses
- Address Resolution Protocol (ARP) is used to map MAC addresses to IP addresses
- If the hosts are on the same network, ARP can be used
 - If the hosts are on different networks, the sending host will send the packet to a *router* which routes the packet to the destination network





An Ethernet Packet

bytes		
7	preamble—start of packet	each byte pattern 10101010
1	start of frame delimiter	pattern 10101011
2 or 6	destination address	Ethernet address or broadcast
2 or 6	source address	Ethernet address
2	length of data section	length in bytes
0–1500	data	message data
0–46	pad (optional)	message must be > 63 bytes long
4	frame checksum	for error detection



End of Chapter 16



Chapter 17: Distributed-File Systems





Chapter 17 Distributed-File Systems

- Background
- Naming and Transparency
- Remote File Access
- Stateful versus Stateless Service
- File Replication
- An Example: AFS





Chapter Objectives

- To explain the naming mechanism that provides location transparency and independence
- To describe the various methods for accessing distributed files
- To contrast stateful and stateless distributed file servers
- To show how replication of files on different machines in a distributed file system is a useful redundancy for improving availability
- To introduce the Andrew file system (AFS) as an example of a distributed file system





Background

- Distributed file system (**DFS**) – a distributed implementation of the classical time-sharing model of a file system, where multiple users share files and storage resources
- A DFS manages set of dispersed storage devices
- Overall storage space managed by a DFS is composed of different, remotely located, smaller storage spaces
- There is usually a correspondence between constituent storage spaces and sets of files





DFS Structure

- **Service** – software entity running on one or more machines and providing a particular type of function to a priori unknown clients
- **Server** – service software running on a single machine
- **Client** – process that can invoke a service using a set of operations that forms its *client interface*
- A client interface for a file service is formed by a set of primitive *file operations* (create, delete, read, write)
- Client interface of a DFS should be transparent, i.e., not distinguish between local and remote files





Naming and Transparency

- **Naming** – mapping between logical and physical objects
- Multilevel mapping – abstraction of a file that hides the details of how and where on the disk the file is actually stored
- A **transparent** DFS hides the location where in the network the file is stored
- For a file being replicated in several sites, the mapping returns a set of the locations of this file's replicas; both the existence of multiple copies and their location are hidden





Naming Structures

- **Location transparency** – file name does not reveal the file's physical storage location
 - File name still denotes a specific, although hidden, set of physical disk blocks
 - Convenient way to share data
 - Can expose correspondence between component units and machines
- **Location independence** – file name does not need to be changed when the file's physical storage location changes
 - Better file abstraction
 - Promotes sharing the storage space itself
 - Separates the naming hierarchy from the storage-devices hierarchy





Naming Schemes — Three Main Approaches

- Files named by combination of their host name and local name; guarantees a unique systemwide name
- Attach remote directories to local directories, giving the appearance of a coherent directory tree; only previously mounted remote directories can be accessed transparently
- Total integration of the component file systems
 - A single global name structure spans all the files in the system
 - If a server is unavailable, some arbitrary set of directories on different machines also becomes unavailable





Remote File Access

- **Remove-service mechanism** is one transfer approach
- Reduce network traffic by retaining recently accessed disk blocks in a cache, so that repeated accesses to the same information can be handled locally
 - If needed data not already cached, a copy of data is brought from the server to the user
 - Accesses are performed on the cached copy
 - Files identified with one master copy residing at the server machine, but copies of (parts of) the file are scattered in different caches
 - **Cache-consistency problem** – keeping the cached copies consistent with the master file
 - ▶ Could be called **network virtual memory**





Cache Location – Disk vs. Main Memory

- Advantages of disk caches
 - More reliable
 - Cached data kept on disk are still there during recovery and don't need to be fetched again
- Advantages of main-memory caches:
 - Permit workstations to be diskless
 - Data can be accessed more quickly
 - Performance speedup in bigger memories
 - Server caches (used to speed up disk I/O) are in main memory regardless of where user caches are located; using main-memory caches on the user machine permits a single caching mechanism for servers and users





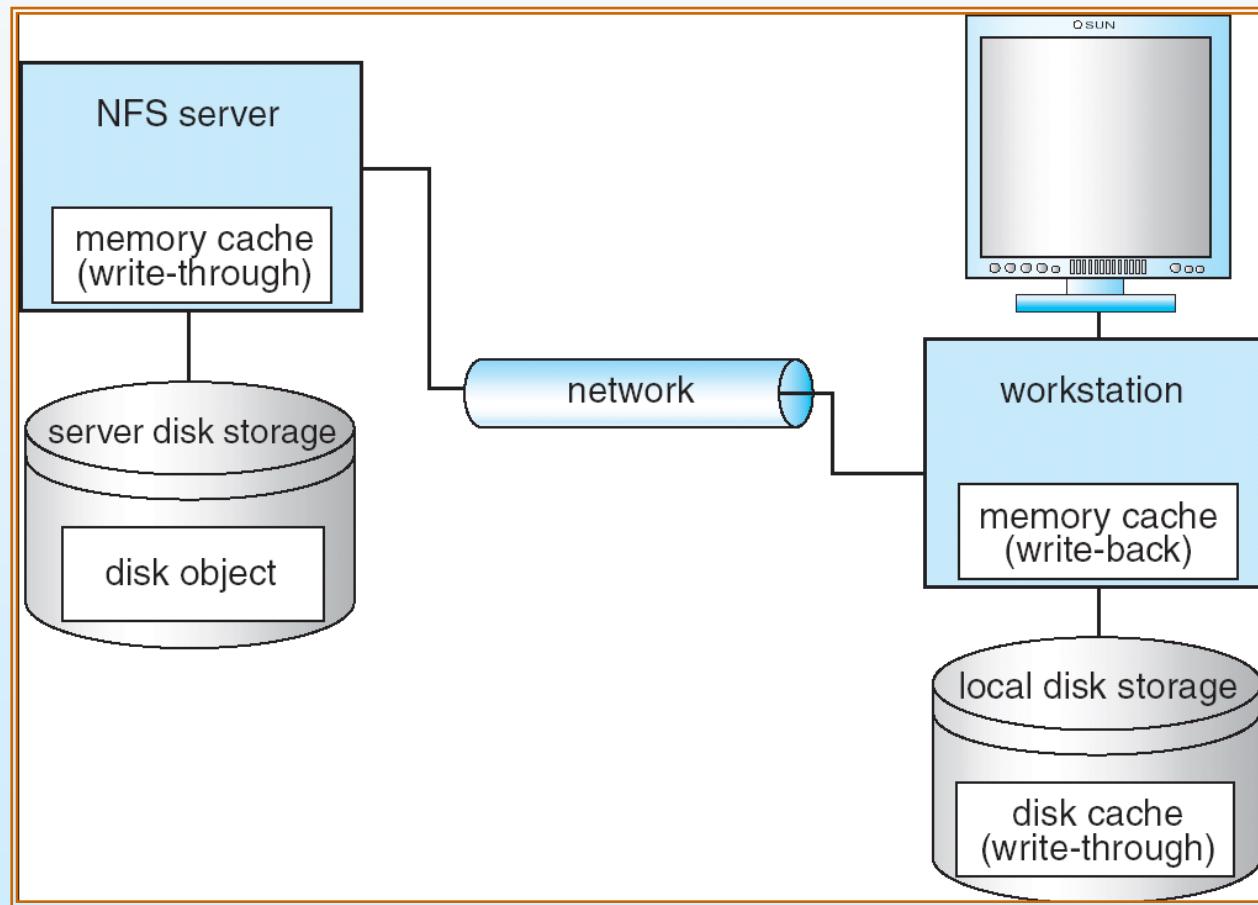
Cache Update Policy

- **Write-through** – write data through to disk as soon as they are placed on any cache
 - Reliable, but poor performance
- **Delayed-write** – modifications written to the cache and then written through to the server later
 - Write accesses complete quickly; some data may be overwritten before they are written back, and so need never be written at all
 - Poor reliability; unwritten data will be lost whenever a user machine crashes
 - Variation – scan cache at regular intervals and flush blocks that have been modified since the last scan
 - Variation – **write-on-close**, writes data back to the server when the file is closed
 - ▶ Best for files that are open for long periods and frequently modified





Cachefs and its Use of Caching





Consistency

- Is locally cached copy of the data consistent with the master copy?
- **Client-initiated approach**
 - Client initiates a validity check
 - Server checks whether the local data are consistent with the master copy
- **Server-initiated approach**
 - Server records, for each client, the (parts of) files it caches
 - When server detects a potential inconsistency, it must react





Comparing Caching and Remote Service

- In caching, many remote accesses handled efficiently by the local cache; most remote accesses will be served as fast as local ones
- Servers are contracted only occasionally in caching (rather than for each access)
 - Reduces server load and network traffic
 - Enhances potential for scalability
- Remote server method handles every remote access across the network; penalty in network traffic, server load, and performance
- Total network overhead in transmitting big chunks of data (caching) is lower than a series of responses to specific requests (remote-service)





Caching and Remote Service (Cont.)

- Caching is superior in access patterns with infrequent writes
 - With frequent writes, substantial overhead incurred to overcome cache-consistency problem
- Benefit from caching when execution carried out on machines with either local disks or large main memories
- Remote access on diskless, small-memory-capacity machines should be done through remote-service method
- In caching, the lower intermachine interface is different form the upper user interface
- In remote-service, the intermachine interface mirrors the local user-file-system interface





Stateful File Service

- Mechanism
 - Client opens a file
 - Server fetches information about the file from its disk, stores it in its memory, and gives the client a connection identifier unique to the client and the open file
 - Identifier is used for subsequent accesses until the session ends
 - Server must reclaim the main-memory space used by clients who are no longer active
- Increased performance
 - Fewer disk accesses
 - Stateful server knows if a file was opened for sequential access and can thus read ahead the next blocks





Stateless File Server

- Avoids state information by making each request self-contained
- Each request identifies the file and position in the file
- No need to establish and terminate a connection by open and close operations





Distinctions Between Stateful & Stateless Service

- Failure Recovery
 - A stateful server loses all its volatile state in a crash
 - ▶ Restore state by recovery protocol based on a dialog with clients, or abort operations that were underway when the crash occurred
 - ▶ Server needs to be aware of client failures in order to reclaim space allocated to record the state of crashed client processes (orphan detection and elimination)
 - With stateless server, the effects of server failure sand recovery are almost unnoticeable
 - ▶ A newly reincarnated server can respond to a self-contained request without any difficulty





Distinctions (Cont.)

- Penalties for using the robust stateless service:
 - longer request messages
 - slower request processing
 - additional constraints imposed on DFS design
- Some environments require stateful service
 - A server employing server-initiated cache validation cannot provide stateless service, since it maintains a record of which files are cached by which clients
 - UNIX use of file descriptors and implicit offsets is inherently stateful; servers must maintain tables to map the file descriptors to inodes, and store the current offset within a file





File Replication

- Replicas of the same file reside on failure-independent machines
- Improves availability and can shorten service time
- Naming scheme maps a replicated file name to a particular replica
 - Existence of replicas should be invisible to higher levels
 - Replicas must be distinguished from one another by different lower-level names
- Updates – replicas of a file denote the same logical entity, and thus an update to any replica must be reflected on all other replicas
- Demand replication – reading a nonlocal replica causes it to be cached locally, thereby generating a new nonprimary replica.





An Example: AFS

- A distributed computing environment (Andrew) under development since 1983 at Carnegie-Mellon University, purchased by IBM and released as **Transarc DFS**, now open sourced as OpenAFS
- AFS tries to solve complex issues such as uniform name space, location-independent file sharing, client-side caching (with cache consistency), secure authentication (via Kerberos)
 - Also includes server-side caching (via replicas), high availability
 - Can span 5,000 workstations





ANDREW (Cont.)

- Clients are presented with a partitioned space of file names: a **local name space** and a **shared name space**
- Dedicated servers, called *Vice*, present the shared name space to the clients as an homogeneous, identical, and location transparent file hierarchy
- The local name space is the root file system of a workstation, from which the shared name space descends
- Workstations run the *Virtue* protocol to communicate with Vice, and are required to have local disks where they store their local name space
- Servers collectively are responsible for the storage and management of the shared name space





ANDREW (Cont.)

- Clients and servers are structured in clusters interconnected by a backbone LAN
- A cluster consists of a collection of workstations and a cluster server and is connected to the backbone by a router
- A key mechanism selected for remote file operations is whole file caching
 - Opening a file causes it to be cached, in its entirety, on the local disk





ANDREW Shared Name Space

- Andrew's **volumes** are small component units associated with the files of a single client
- A **fid** identifies a Vice file or directory - A fid is 96 bits long and has three equal-length components:
 - volume number
 - **vnode number** – index into an array containing the inodes of files in a single volume
 - **uniquifier** – allows reuse of vnode numbers, thereby keeping certain data structures, compact
- Fids are location transparent; therefore, file movements from server to server do not invalidate cached directory contents
- Location information is kept on a volume basis, and the information is replicated on each server





ANDREW File Operations

- Andrew caches entire files from servers
 - A client workstation interacts with Vice servers only during opening and closing of files
- *Venus* – caches files from Vice when they are opened, and stores modified copies of files back when they are closed
- Reading and writing bytes of a file are done by the kernel without Venus intervention on the cached copy
- Venus caches contents of directories and symbolic links, for path-name translation
- Exceptions to the caching policy are modifications to directories that are made directly on the server responsibility for that directory





ANDREW Implementation

- Client processes are interfaced to a UNIX kernel with the usual set of system calls
- Venus carries out path-name translation component by component
- The UNIX file system is used as a low-level storage system for both servers and clients
 - The client cache is a local directory on the workstation's disk
- Both Venus and server processes access UNIX files directly by their inodes to avoid the expensive path name-to-inode translation routine





ANDREW Implementation (Cont.)

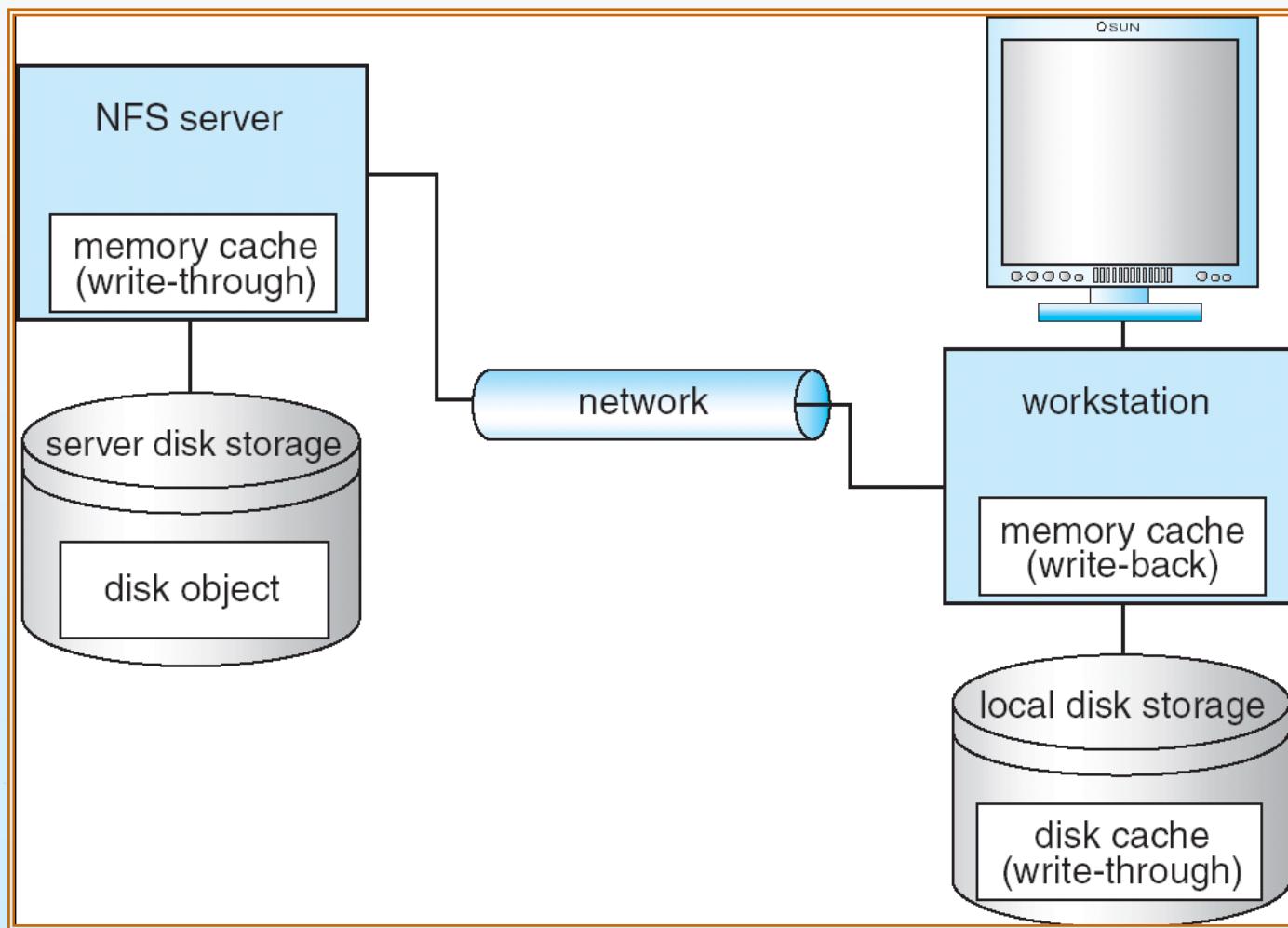
- Venus manages two separate caches:
 - one for status
 - one for data
- LRU algorithm used to keep each of them bounded in size
- The status cache is kept in virtual memory to allow rapid servicing of *stat* (file status returning) system calls
- The data cache is resident on the local disk, but the UNIX I/O buffering mechanism does some caching of the disk blocks in memory that are transparent to Venus



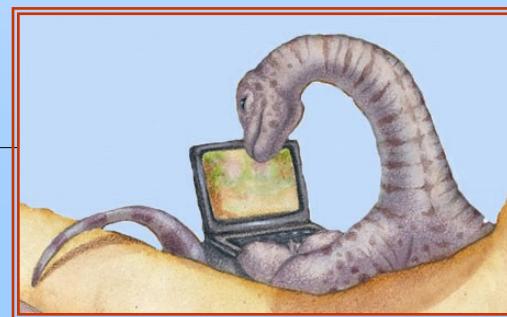
End of Chapter 17



Fig. 17.01



Chapter 18: Distributed Coordination





Chapter 18 Distributed Coordination

- Event Ordering
- Mutual Exclusion
- Atomicity
- Concurrency Control
- Deadlock Handling
- Election Algorithms
- Reaching Agreement





Chapter Objectives

- To describe various methods for achieving mutual exclusion in a distributed system
- To explain how atomic transactions can be implemented in a distributed system
- To show how some of the concurrency-control schemes discussed in Chapter 6 can be modified for use in a distributed environment
- To present schemes for handling deadlock prevention, deadlock avoidance, and deadlock detection in a distributed system





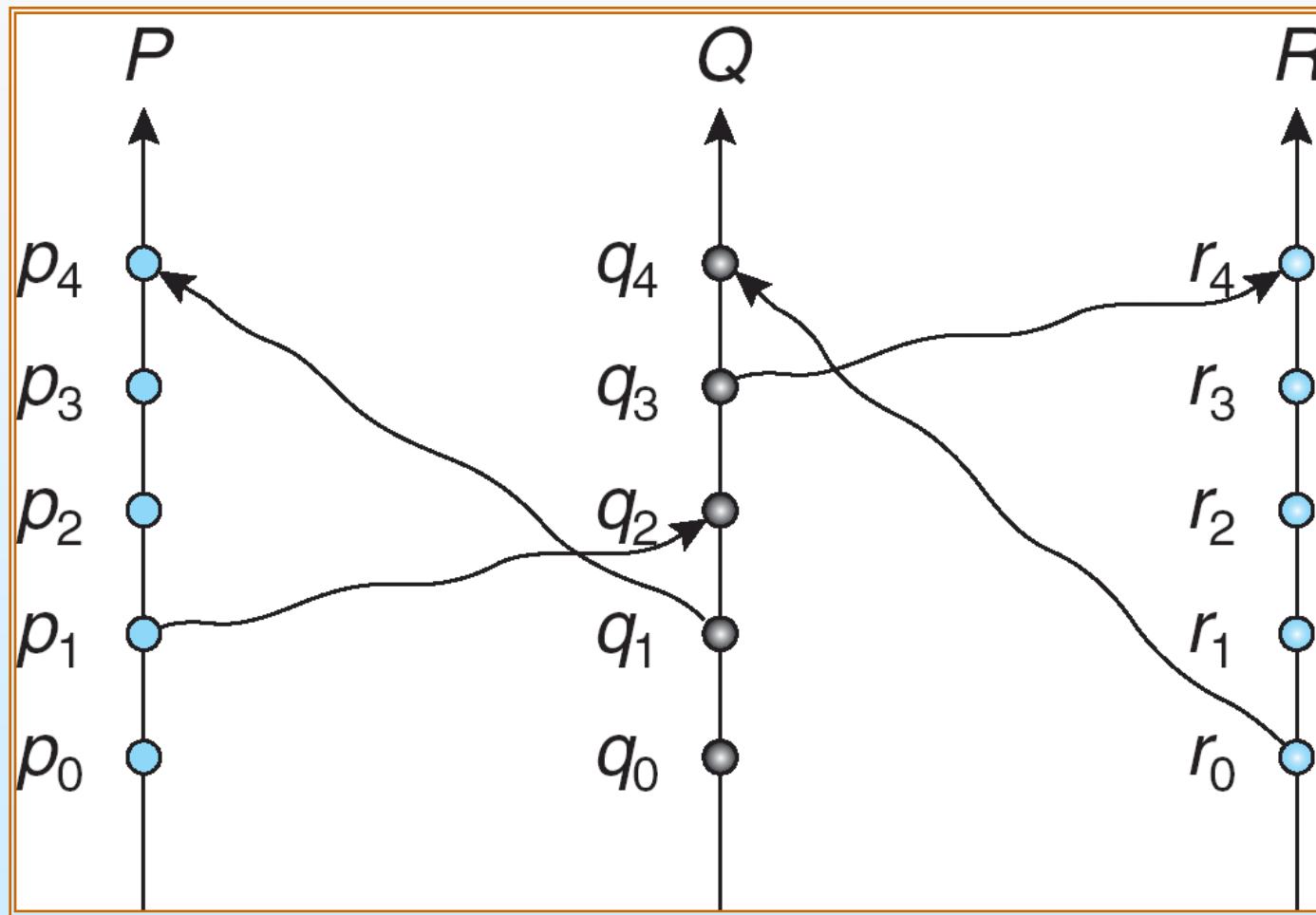
Event Ordering

- *Happened-before* relation (denoted by \rightarrow)
 - If A and B are events in the same process, and A was executed before B , then $A \rightarrow B$
 - If A is the event of sending a message by one process and B is the event of receiving that message by another process, then $A \rightarrow B$
 - If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$





Relative Time for Three Concurrent Processes





Implementation of →

- Associate a timestamp with each system event
 - Require that for every pair of events A and B, if $A \rightarrow B$, then the timestamp of A is less than the timestamp of B
- Within each process P_i a **logical clock**, LC_i is associated
 - The logical clock can be implemented as a simple counter that is incremented between any two successive events executed within a process
 - ▶ Logical clock is **monotonically increasing**
 - A process advances its logical clock when it receives a message whose timestamp is greater than the current value of its logical clock
 - If the timestamps of two events A and B are the same, then the events are concurrent
 - We may use the process identity numbers to break ties and to create a total ordering





Distributed Mutual Exclusion (DME)

- Assumptions
 - The system consists of n processes; each process P_i resides at a different processor
 - Each process has a critical section that requires mutual exclusion
- Requirement
 - If P_i is executing in its critical section, then no other process P_j is executing in its critical section
 - We present two algorithms to ensure the mutual exclusion execution of processes in their critical sections





DME: Centralized Approach

- One of the processes in the system is chosen to coordinate the entry to the critical section
- A process that wants to enter its critical section sends a request message to the coordinator
- The coordinator decides which process can enter the critical section next, and its sends that process a reply message
- When the process receives a reply message from the coordinator, it enters its critical section
- After exiting its critical section, the process sends a release message to the coordinator and proceeds with its execution
- This scheme requires three messages per critical-section entry:
 - request
 - reply
 - release





DME: Fully Distributed Approach

- When process P_i wants to enter its critical section, it generates a new timestamp, TS , and sends the message *request* (P_i , TS) to all other processes in the system
- When process P_j receives a *request* message, it may reply immediately or it may defer sending a reply back
- When process P_i receives a *reply* message from all other processes in the system, it can enter its critical section
- After exiting its critical section, the process sends *reply* messages to all its deferred requests





DME: Fully Distributed Approach (Cont.)

- The decision whether process P_j replies immediately to a $\text{request}(P_i, TS)$ message or defers its reply is based on three factors:
 - If P_j is in its critical section, then it defers its reply to P_i
 - If P_j does *not* want to enter its critical section, then it sends a *reply* immediately to P_i
 - If P_j wants to enter its critical section but has not yet entered it, then it compares its own request timestamp with the timestamp TS
 - ▶ If its own request timestamp is greater than TS , then it sends a *reply* immediately to P_i (P_i asked first)
 - ▶ Otherwise, the reply is deferred





Desirable Behavior of Fully Distributed Approach

- Freedom from Deadlock is ensured
- Freedom from starvation is ensured, since entry to the critical section is scheduled according to the timestamp ordering
 - The timestamp ordering ensures that processes are served in a first-come, first served order
- The number of messages per critical-section entry is

$$2 \times (n - 1)$$

This is the minimum number of required messages per critical-section entry when processes act independently and concurrently





Three Undesirable Consequences

- The processes need to know the identity of all other processes in the system, which makes the dynamic addition and removal of processes more complex
- If one of the processes fails, then the entire scheme collapses
 - This can be dealt with by continuously monitoring the state of all the processes in the system
- Processes that have not entered their critical section must pause frequently to assure other processes that they intend to enter the critical section
 - This protocol is therefore suited for small, stable sets of cooperating processes





Token-Passing Approach

- Circulate a token among processes in system
 - **Token** is special type of message
 - Possession of token entitles holder to enter critical section
- Processes *logically* organized in a **ring structure**
- Algorithm similar to Chapter 6 algorithm 1 but token substituted for shared variable
- Unidirectional ring guarantees freedom from starvation
- Two types of failures
 - Lost token – election must be called
 - Failed processes – new logical ring established





Atomicity

- Either all the operations associated with a program unit are executed to completion, or none are performed
- Ensuring **atomicity** in a distributed system requires a **transaction coordinator**, which is responsible for the following:
 - Starting the execution of the transaction
 - Breaking the transaction into a number of subtransactions, and distribution these subtransactions to the appropriate sites for execution
 - Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites





Two-Phase Commit Protocol (2PC)

- Assumes fail-stop model
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached
- When the protocol is initiated, the transaction may still be executing at some of the local sites
- The protocol involves all the local sites at which the transaction executed
- Example: Let T be a transaction initiated at site S_i and let the transaction coordinator at S_i be C_i





Phase 1: Obtaining a Decision

- C_i adds $\langle \text{prepare } T \rangle$ record to the log
- C_i sends $\langle \text{prepare } T \rangle$ message to all sites
- When a site receives a $\langle \text{prepare } T \rangle$ message, the transaction manager determines if it can commit the transaction
 - If no: add $\langle \text{no } T \rangle$ record to the log and respond to C_i with $\langle \text{abort } T \rangle$
 - If yes:
 - ▶ add $\langle \text{ready } T \rangle$ record to the log
 - ▶ force *all log records* for T onto stable storage
 - ▶ transaction manager sends $\langle \text{ready } T \rangle$ message to C_i





Phase 1 (Cont.)

- Coordinator collects responses
 - All respond “ready”, decision is *commit*
 - At least one response is “abort”, decision is *abort*
 - At least one participant fails to respond within time out period, decision is *abort*





Phase 2: Recording Decision in the Database

- Coordinator adds a decision record
 - <abort T > or <commit T >
 - to its log and forces record onto stable storage
- Once that record reaches stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally





Failure Handling in 2PC – Site Failure

- The log contains a <commit T > record
 - In this case, the site executes **redo(T)**
- The log contains an <abort T > record
 - In this case, the site executes **undo(T)**
- The log contains a <ready T > record; consult C_i
 - If C_i is down, site sends **query-status T** message to the other sites
- The log contains no control records concerning T
 - In this case, the site executes **undo(T)**





Failure Handling in 2PC – Coordinator C_i , Failure

- If an active site contains a $\langle \text{commit } T \rangle$ record in its log, the T must be committed
- If an active site contains an $\langle \text{abort } T \rangle$ record in its log, then T must be aborted
- If some active site does *not* contain the record $\langle \text{ready } T \rangle$ in its log then the failed coordinator C_i cannot have decided to commit T
 - Rather than wait for C_i to recover, it is preferable to abort T
- All active sites have a $\langle \text{ready } T \rangle$ record in their logs, but no additional control records
 - In this case we must wait for the coordinator to recover
 - Blocking problem – T is blocked pending the recovery of site S_i





Concurrency Control

- Modify the centralized concurrency schemes to accommodate the distribution of transactions
- Transaction manager coordinates execution of transactions (or subtransactions) that access data at local sites
- Local transaction only executes at that site
- Global transaction executes at several sites





Locking Protocols

- Can use the two-phase locking protocol in a distributed environment by changing how the lock manager is implemented
- Nonreplicated scheme – each site maintains a local lock manager which administers lock and unlock requests for those data items that are stored in that site
 - Simple implementation involves two message transfers for handling lock requests, and one message transfer for handling unlock requests
 - Deadlock handling is more complex





Single-Coordinator Approach

- A single lock manager resides in a single chosen site, all lock and unlock requests are made at that site
- Simple implementation
- Simple deadlock handling
- Possibility of bottleneck
- Vulnerable to loss of concurrency controller if single site fails
- **Multiple-coordinator approach** distributes lock-manager function over several sites





Majority Protocol

- Avoids drawbacks of central control by dealing with replicated data in a decentralized manner
- More complicated to implement
- Deadlock-handling algorithms must be modified; possible for deadlock to occur in locking only one data item





Biased Protocol

- Similar to majority protocol, but requests for shared locks prioritized over requests for exclusive locks
- Less overhead on read operations than in majority protocol; but has additional overhead on writes
- Like majority protocol, deadlock handling is complex





Primary Copy

- One of the sites at which a replica resides is designated as the primary site
 - Request to lock a data item is made at the primary site of that data item
- Concurrency control for replicated data handled in a manner similar to that of unreplicated data
- Simple implementation, but if primary site fails, the data item is unavailable, even though other sites may have a replica





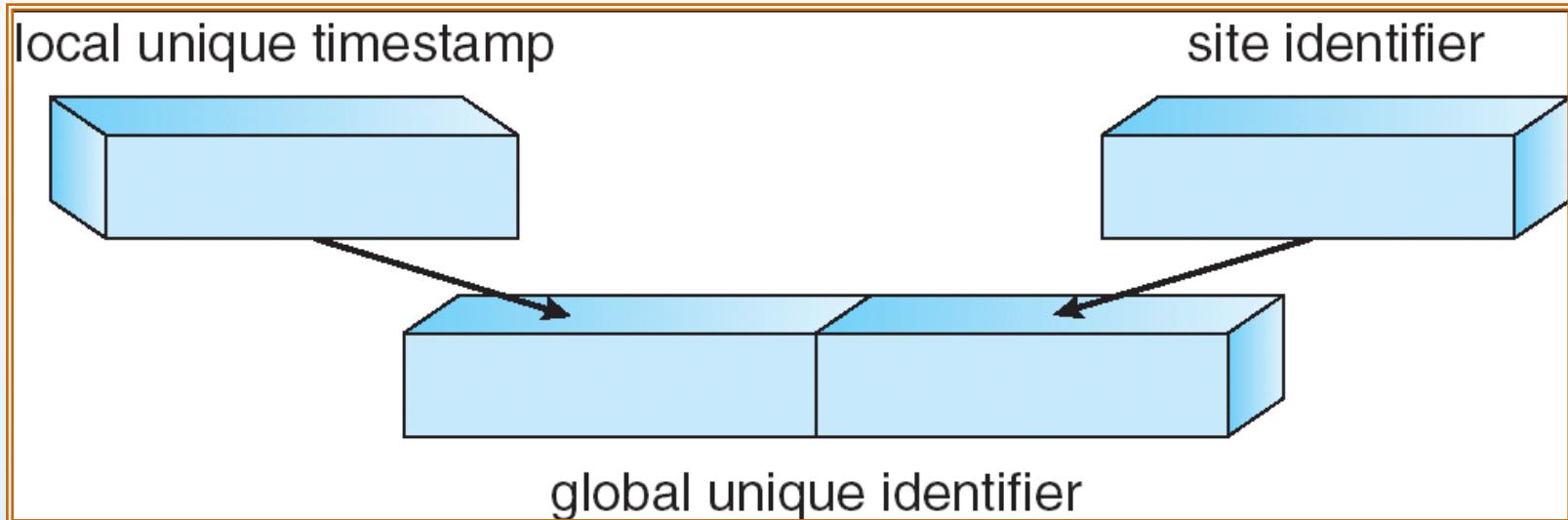
Timestamping

- Generate unique timestamps in distributed scheme:
 - Each site generates a unique local timestamp
 - The global unique timestamp is obtained by concatenation of the unique local timestamp with the unique site identifier
 - Use a *logical clock* defined within each site to ensure the fair generation of timestamps
- Timestamp-ordering scheme – combine the centralized concurrency control timestamp scheme with the 2PC protocol to obtain a protocol that ensures serializability with no cascading rollbacks





Generation of Unique Timestamps





Deadlock Prevention

- Resource-ordering deadlock-prevention – define a *global* ordering among the system resources
 - Assign a unique number to all system resources
 - A process may request a resource with unique number i only if it is not holding a resource with a unique number greater than i
 - Simple to implement; requires little overhead
- Banker's algorithm – designate one of the processes in the system as the process that maintains the information necessary to carry out the Banker's algorithm
 - Also implemented easily, but may require too much overhead





Timestamped Deadlock-Prevention Scheme

- Each process P_i is assigned a unique priority number
- Priority numbers are used to decide whether a process P_i should wait for a process P_j ; otherwise P_i is rolled back
- The scheme prevents deadlocks
 - For every edge $P_i \rightarrow P_j$ in the wait-for graph, P_i has a higher priority than P_j
 - Thus a cycle cannot exist





Wait-Die Scheme

- Based on a nonpreemptive technique
- If P_i requests a resource currently held by P_j , P_i is allowed to wait only if it has a smaller timestamp than does P_j (P_i is older than P_j)
 - Otherwise, P_i is rolled back (dies)
- Example: Suppose that processes P_1 , P_2 , and P_3 have timestamps t, 10, and 15 respectively
 - if P_1 request a resource held by P_2 , then P_1 will wait
 - If P_3 requests a resource held by P_2 , then P_3 will be rolled back





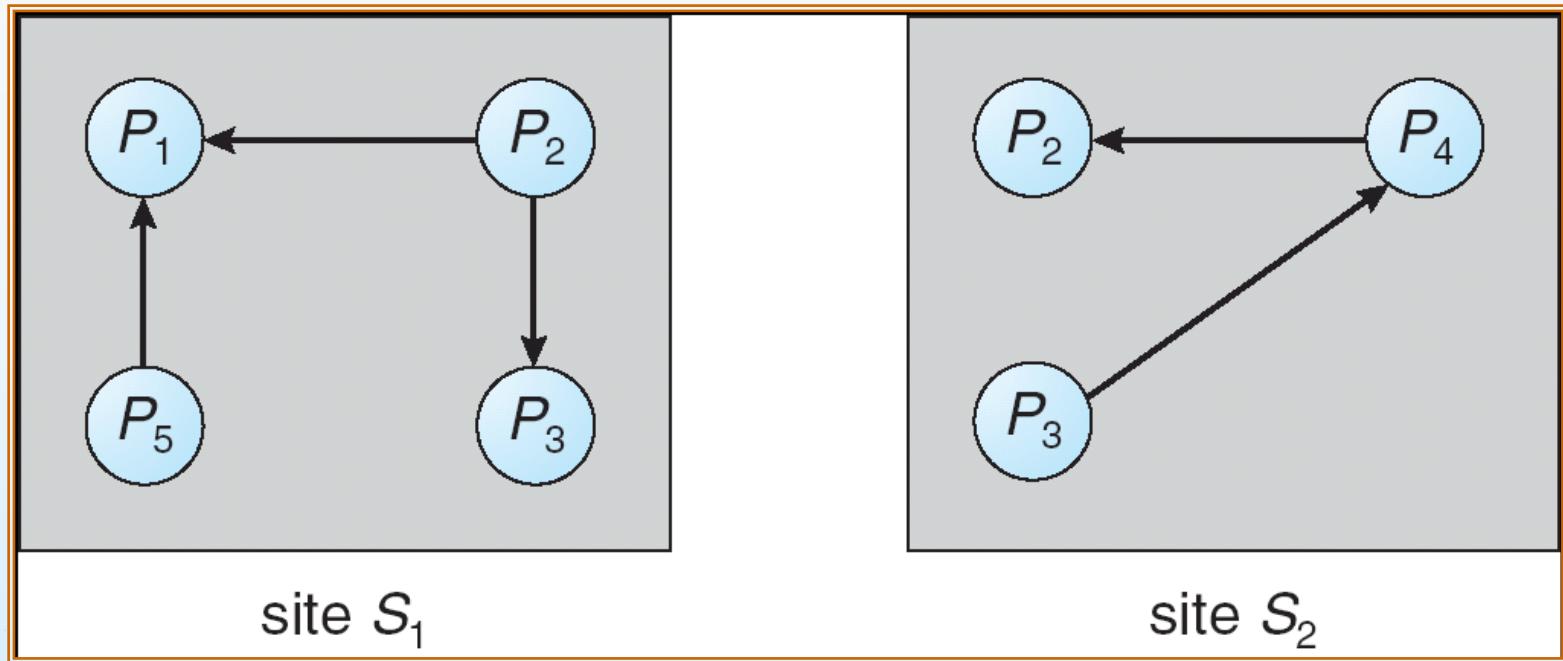
Would-Wait Scheme

- Based on a preemptive technique; counterpart to the wait-die system
- If P_i requests a resource currently held by P_j , P_i is allowed to wait only if it has a larger timestamp than does P_j (P_i is younger than P_j). Otherwise P_j is rolled back (P_j is wounded by P_i)
- Example: Suppose that processes P_1 , P_2 , and P_3 have timestamps 5, 10, and 15 respectively
 - If P_1 requests a resource held by P_2 , then the resource will be preempted from P_2 and P_2 will be rolled back
 - If P_3 requests a resource held by P_2 , then P_3 will wait



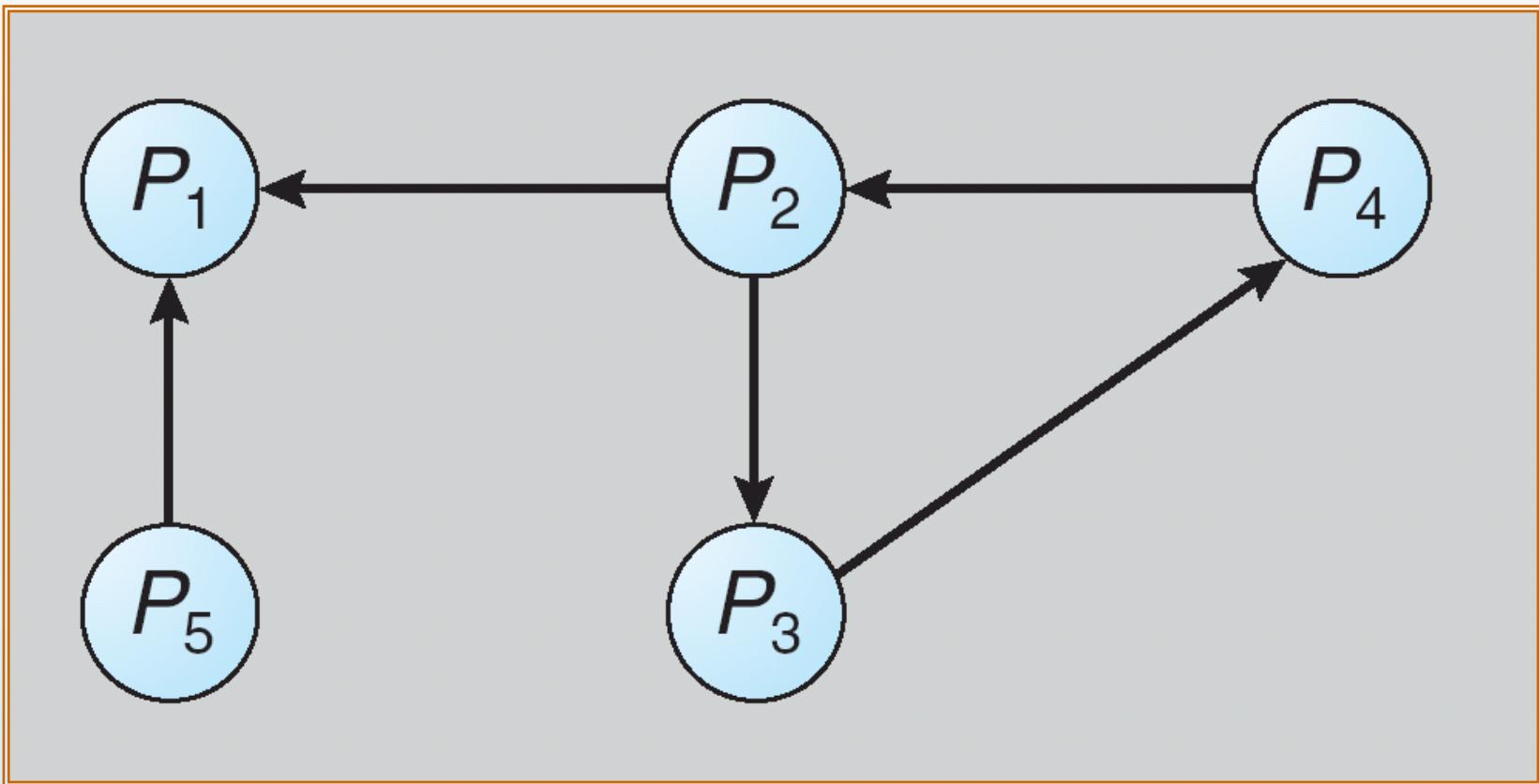


Two Local Wait-For Graphs





Global Wait-For Graph





Deadlock Detection – Centralized Approach

- Each site keeps a local wait-for graph
 - The nodes of the graph correspond to all the processes that are currently either holding or requesting any of the resources local to that site
- A global wait-for graph is maintained in a single coordination process; this graph is the union of all local wait-for graphs
- There are three different options (points in time) when the wait-for graph may be constructed:
 1. Whenever a new edge is inserted or removed in one of the local wait-for graphs
 2. Periodically, when a number of changes have occurred in a wait-for graph
 3. Whenever the coordinator needs to invoke the cycle-detection algorithm
- Unnecessary rollbacks may occur as a result of false cycles





Detection Algorithm Based on Option 3

- Append unique identifiers (timestamps) to requests from different sites
- When process P_i , at site A , requests a resource from process P_j , at site B , a request message with timestamp TS is sent
- The edge $P_i \rightarrow P_j$ with the label TS is inserted in the local wait-for of A . The edge is inserted in the local wait-for graph of B only if B has received the request message and cannot immediately grant the requested resource





The Algorithm

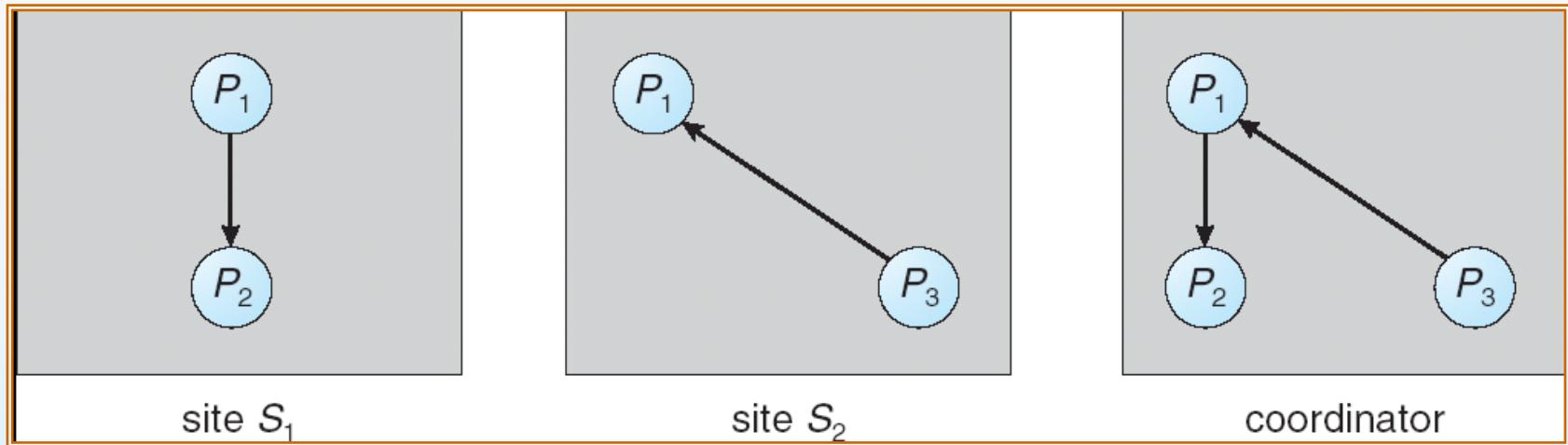
1. The controller sends an initiating message to each site in the system
2. On receiving this message, a site sends its local wait-for graph to the coordinator
3. When the controller has received a reply from each site, it constructs a graph as follows:
 - (a) The constructed graph contains a vertex for every process in the system
 - (b) The graph has an edge $P_i \rightarrow P_j$ if and only if
 - (1) there is an edge $P_i \rightarrow P_j$ in one of the wait-for graphs, or
 - (2) an edge $P_i \rightarrow P_j$ with some label TS appears in more than one wait-for graph

If the constructed graph contains a cycle \Rightarrow deadlock





Local and Global Wait-For Graphs





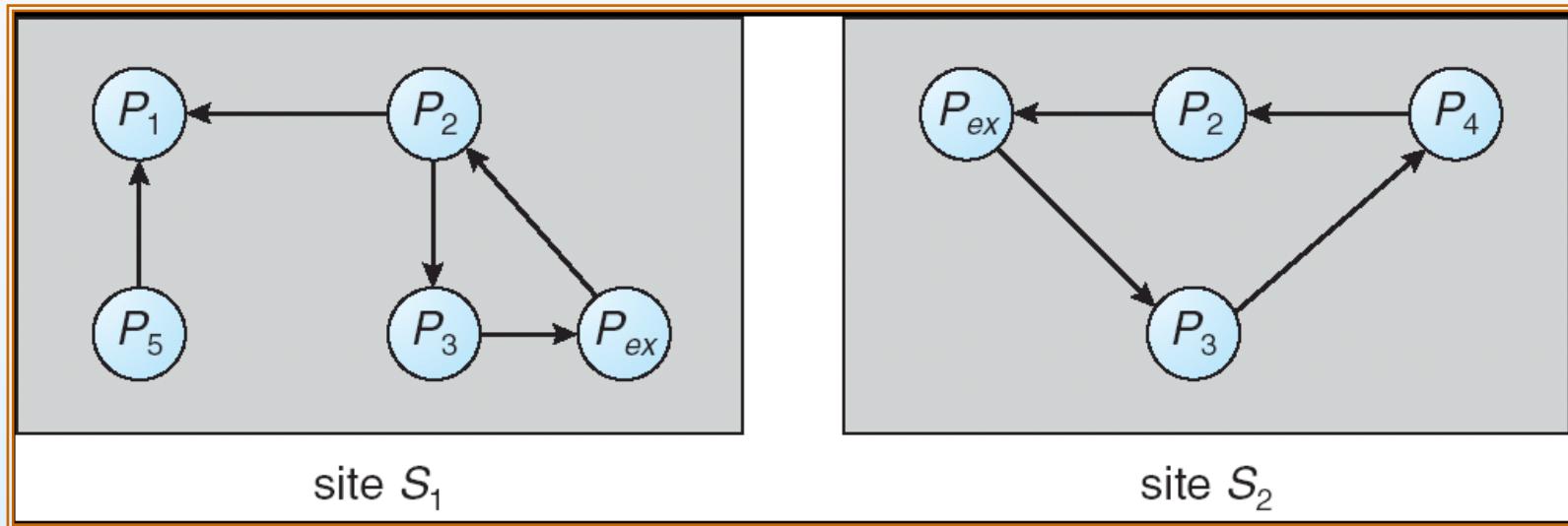
Fully Distributed Approach

- All controllers share equally the responsibility for detecting deadlock
- Every site constructs a wait-for graph that represents a part of the total graph
- We add one additional node P_{ex} to each local wait-for graph
- If a local wait-for graph contains a cycle that does not involve node P_{ex} , then the system is in a deadlock state
- A cycle involving P_{ex} implies the possibility of a deadlock
 - To ascertain whether a deadlock does exist, a distributed deadlock-detection algorithm must be invoked



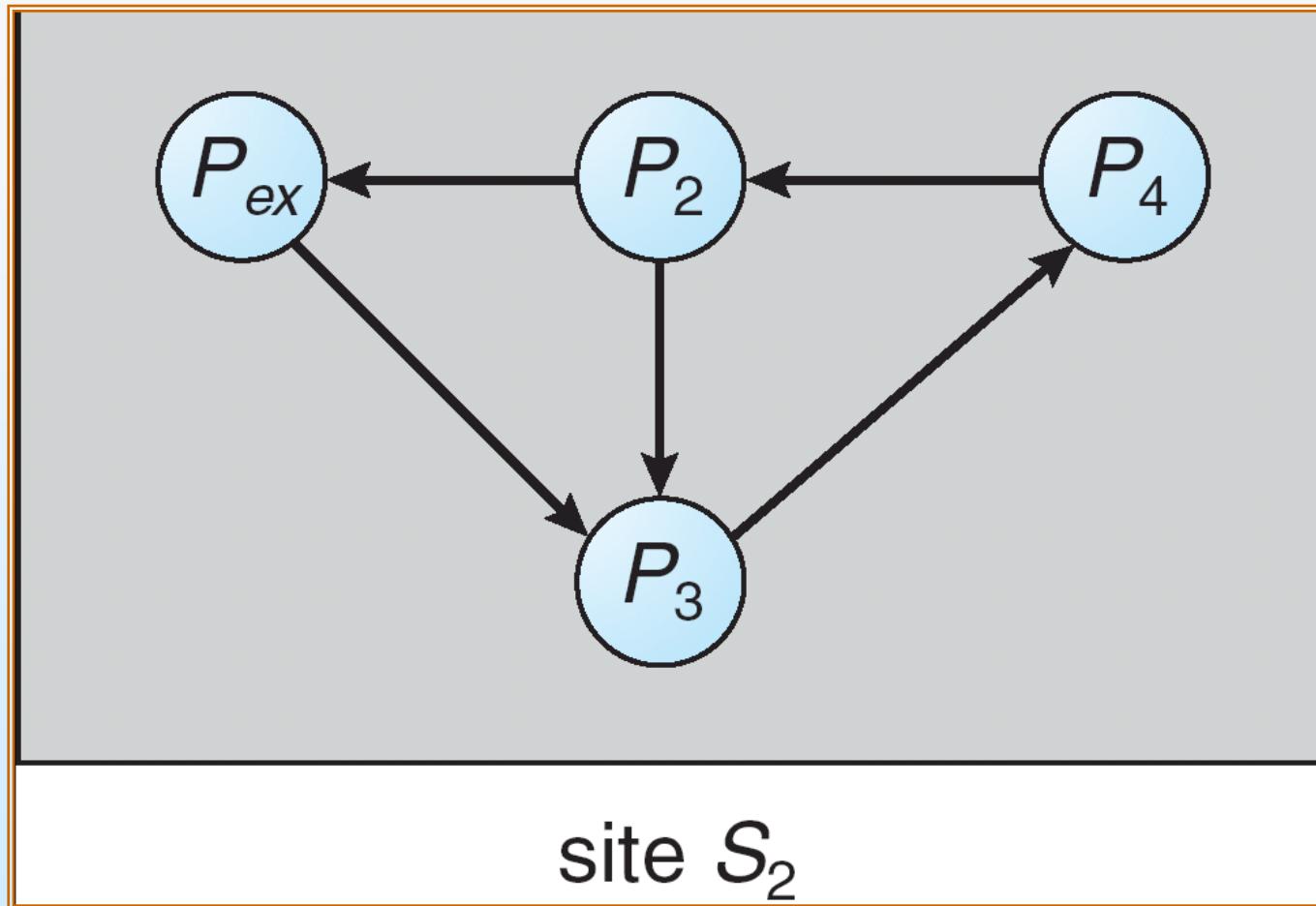


Augmented Local Wait-For Graphs





Augmented Local Wait-For Graph in Site S2





Election Algorithms

- Determine where a new copy of the coordinator should be restarted
- Assume that a unique priority number is associated with each active process in the system, and assume that the priority number of process P_i is i
- Assume a one-to-one correspondence between processes and sites
- The coordinator is always the process with the largest priority number. When a coordinator fails, the algorithm must elect that active process with the largest priority number
- Two algorithms, the bully algorithm and a ring algorithm, can be used to elect a new coordinator in case of failures





Bully Algorithm

- Applicable to systems where every process can send a message to every other process in the system
- If process P_i sends a request that is not answered by the coordinator within a time interval T , assume that the coordinator has failed; P_i tries to elect itself as the new coordinator
- P_i sends an election message to every process with a higher priority number, P_i then waits for any of these processes to answer within T





Bully Algorithm (Cont.)

- If no response within T , assume that all processes with numbers greater than i have failed; P_i elects itself the new coordinator
- If answer is received, P_i begins time interval T' , waiting to receive a message that a process with a higher priority number has been elected
- If no message is sent within T' , assume the process with a higher number has failed; P_i should restart the algorithm





Bully Algorithm (Cont.)

- If P_i is not the coordinator, then, at any time during execution, P_i may receive one of the following two messages from process P_j
 - P_j is the new coordinator ($j > i$). P_i , in turn, records this information
 - P_j started an election ($j > i$). P_i , sends a response to P_j and begins its own election algorithm, provided that P_i has not already initiated such an election
- After a failed process recovers, it immediately begins execution of the same algorithm
- If there are no active processes with higher numbers, the recovered process forces all processes with lower number to let it become the coordinator process, even if there is a currently active coordinator with a lower number





Ring Algorithm

- Applicable to systems organized as a ring (logically or physically)
- Assumes that the links are unidirectional, and that processes send their messages to their right neighbors
- Each process maintains an active list, consisting of all the priority numbers of all active processes in the system when the algorithm ends
- If process P_i detects a coordinator failure, it creates a new active list that is initially empty. It then sends a message $\text{elect}(i)$ to its right neighbor, and adds the number i to its active list





Ring Algorithm (Cont.)

- If P_i receives a message $\text{elect}(j)$ from the process on the left, it must respond in one of three ways:
 1. If this is the first elect message it has seen or sent, P_i creates a new active list with the numbers i and j
 - It then sends the message $\text{elect}(i)$, followed by the message $\text{elect}(j)$
 2. If $i \neq j$, then the active list for P_i now contains the numbers of all the active processes in the system
 - P_i can now determine the largest number in the active list to identify the new coordinator process
 3. If $i = j$, then P_i receives the message $\text{elect}(i)$
 - The active list for P_i contains all the active processes in the system
 - P_i can now determine the new coordinator process.





Reaching Agreement

- There are applications where a set of processes wish to agree on a common “value”
- Such agreement may not take place due to:
 - Faulty communication medium
 - Faulty processes
 - ▶ Processes may send garbled or incorrect messages to other processes
 - ▶ A subset of the processes may collaborate with each other in an attempt to defeat the scheme





Faulty Communications

- Process P_i at site A , has sent a message to process P_j at site B ; to proceed, P_i needs to know if P_j has received the message
- Detect failures using a time-out scheme
 - When P_i sends out a message, it also specifies a time interval during which it is willing to wait for an acknowledgment message from P_j
 - When P_j receives the message, it immediately sends an acknowledgment to P_i
 - If P_i receives the acknowledgment message within the specified time interval, it concludes that P_j has received its message
 - ▶ If a time-out occurs, P_j needs to retransmit its message and wait for an acknowledgment
 - Continue until P_i either receives an acknowledgment, or is notified by the system that B is down





Faulty Communications (Cont.)

- Suppose that P_j also needs to know that P_i has received its acknowledgment message, in order to decide on how to proceed
 - In the presence of failure, it is not possible to accomplish this task
 - It is not possible in a distributed environment for processes P_i and P_j to agree completely on their respective states





Faulty Processes (Byzantine Generals Problem)

- Communication medium is reliable, but processes can fail in unpredictable ways
- Consider a system of n processes, of which no more than m are faulty
 - Suppose that each process P_i has some private value of V_i
- Devise an algorithm that allows each nonfaulty P_i to construct a vector $X_i = (A_{i,1}, A_{i,2}, \dots, A_{i,n})$ such that:
 - If P_j is a nonfaulty process, then $A_{ij} = V_j$.
 - If P_i and P_j are both nonfaulty processes, then $X_i = X_j$.
- Solutions share the following properties
 - A correct algorithm can be devised only if $n \geq 3m + 1$
 - The worst-case delay for reaching agreement is proportionate to $m + 1$ message-passing delays



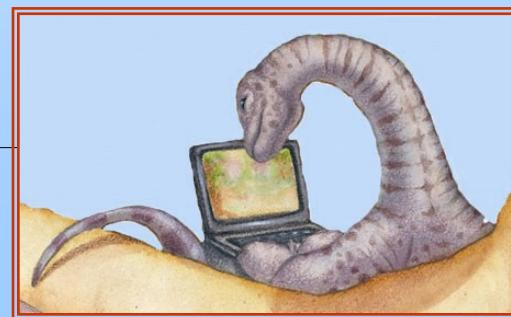


Faulty Processes (Cont.)

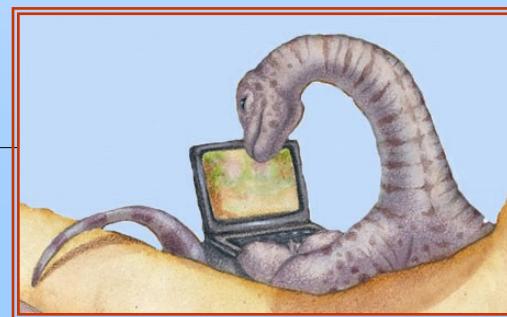
- An algorithm for the case where $m = 1$ and $n = 4$ requires two rounds of information exchange:
 - Each process sends its private value to the other 3 processes
 - Each process sends the information it has obtained in the first round to all other processes
- If a faulty process refuses to send messages, a nonfaulty process can choose an arbitrary value and pretend that that value was sent by that process
- After the two rounds are completed, a nonfaulty process P_i can construct its vector $X_i = (A_{i,1}, A_{i,2}, A_{i,3}, A_{i,4})$ as follows:
 - $A_{i,j} = V_i$
 - For $j \neq i$, if at least two of the three values reported for process P_j agree, then the majority value is used to set the value of A_{ij}
 - ▶ Otherwise, a default value (*nil*) is used



End of Chapter 18



Chapter 19: Real-Time Systems





Chapter 19: Real-Time Systems

- System Characteristics
- Features of Real-Time Systems
- Implementing Real-Time Operating Systems
- Real-Time CPU Scheduling
- VxWorks 5.x





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





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.





System Characteristics

- Single purpose
- Small size
- Inexpensively mass-produced
- Specific timing requirements





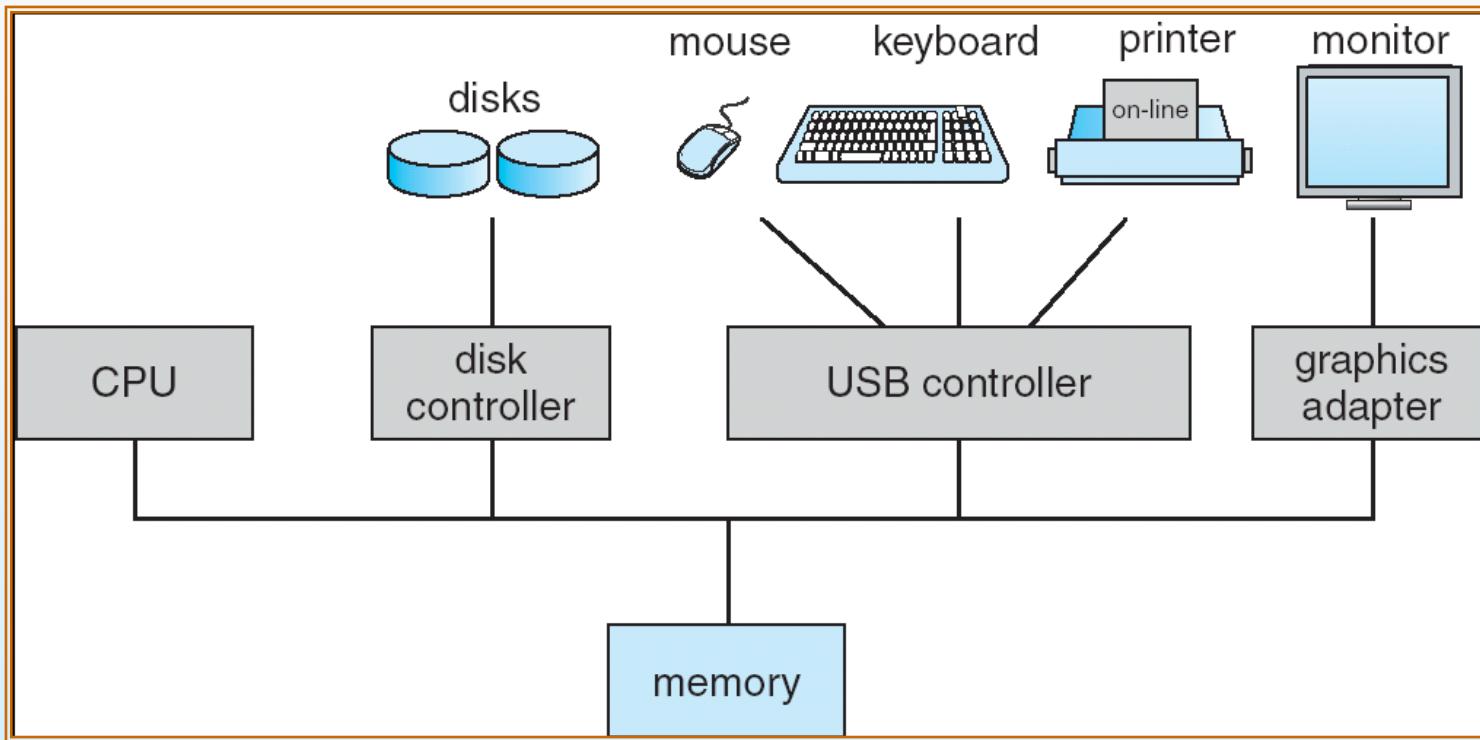
System-on-a-Chip

- 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





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.





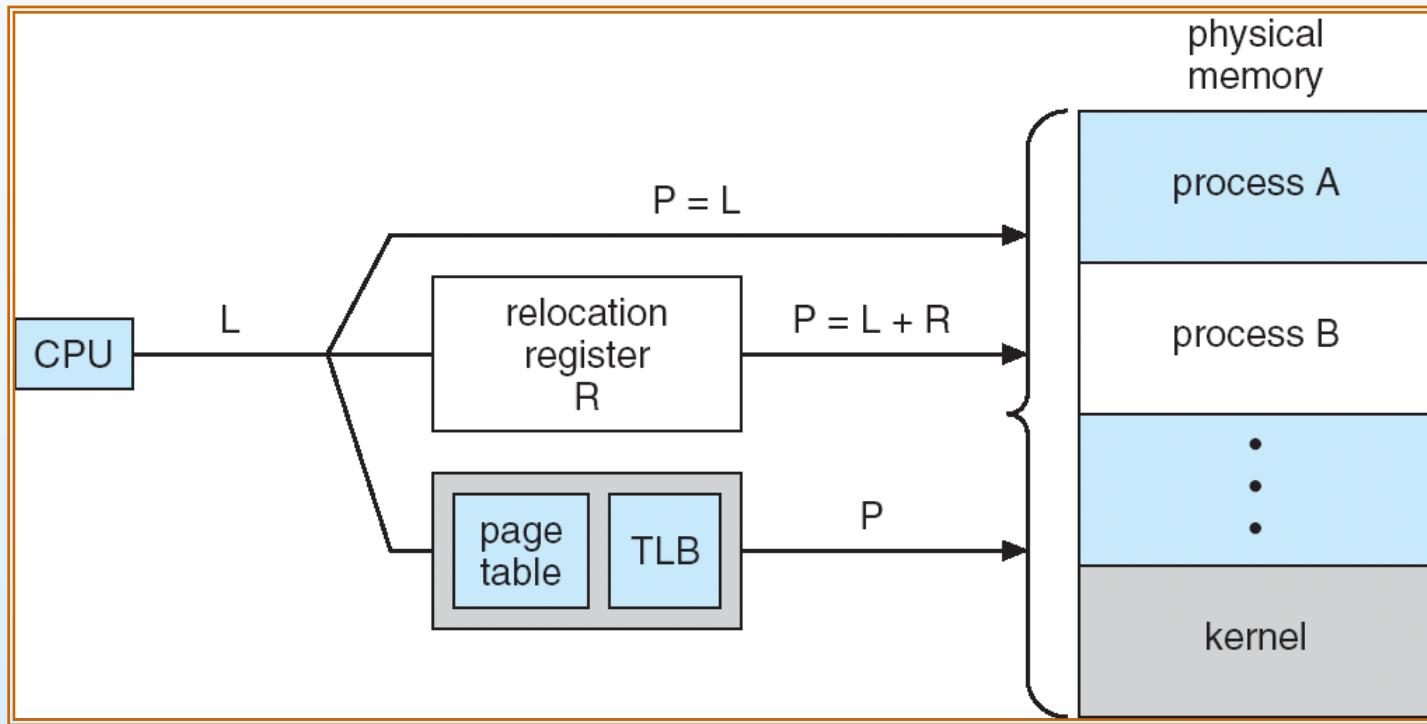
Virtual Memory in Real-Time Systems

- 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





Implementing Real-Time Operating Systems

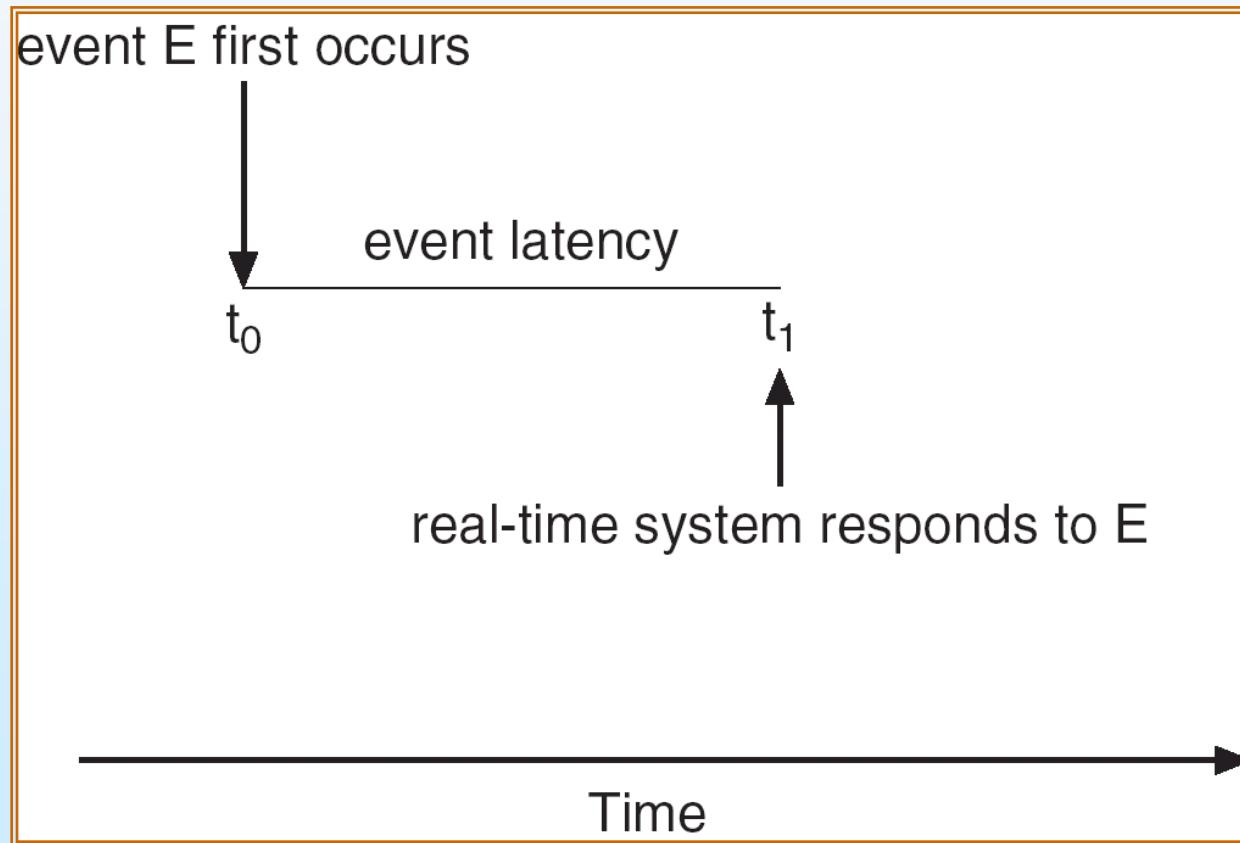
- In general, real-time operating systems must provide:
 - (1) Preemptive, priority-based scheduling
 - (2) Preemptive kernels
 - (3) Latency must be minimized





Minimizing Latency

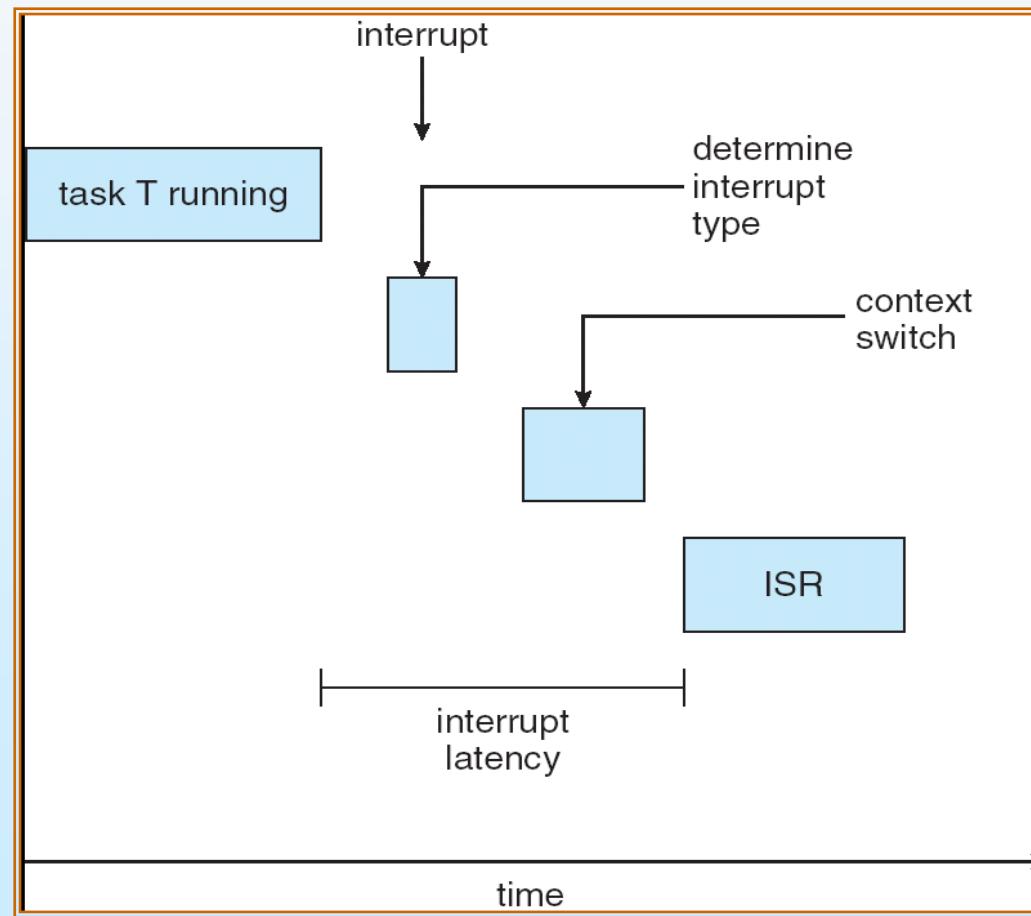
- **Event latency** is the amount of time from when an event occurs to when it is serviced.





Interrupt Latency

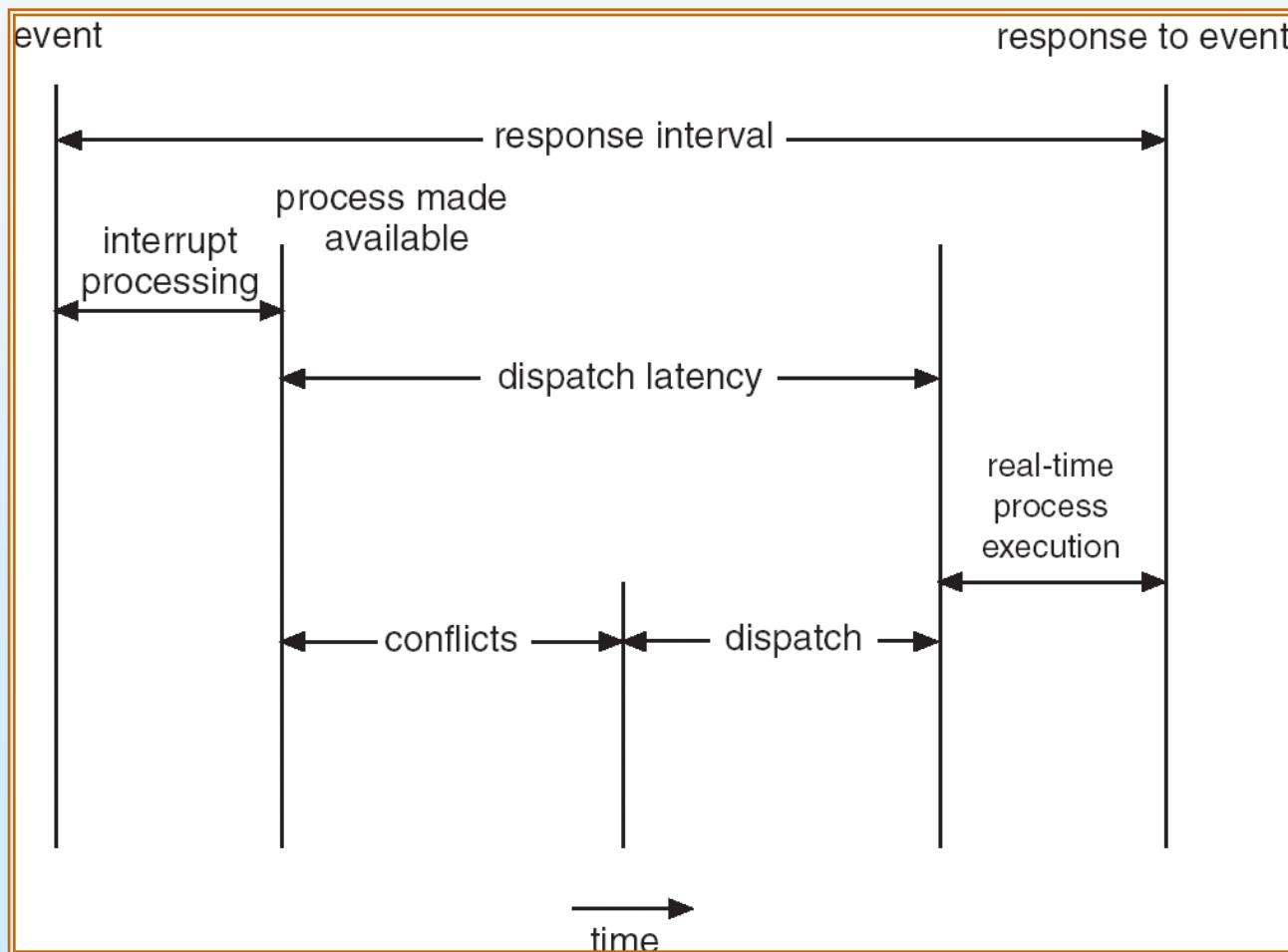
- Interrupt latency is the period of time from when an interrupt arrives at the CPU to when it is serviced.





Dispatch Latency

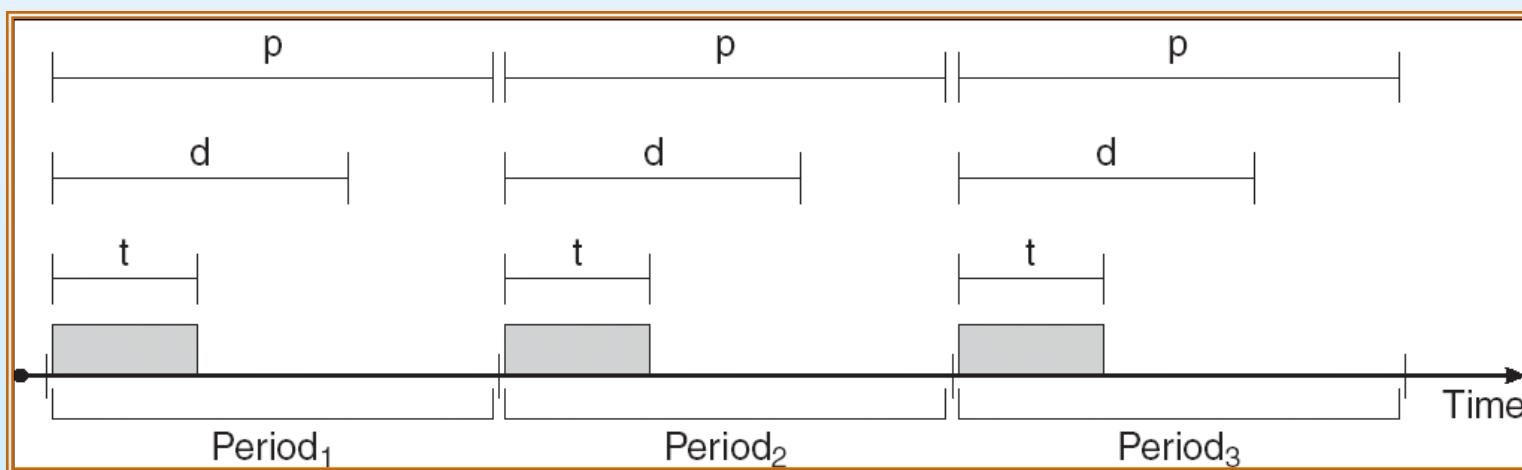
- **Dispatch latency** is the amount of time required for the scheduler to stop one process and start another.





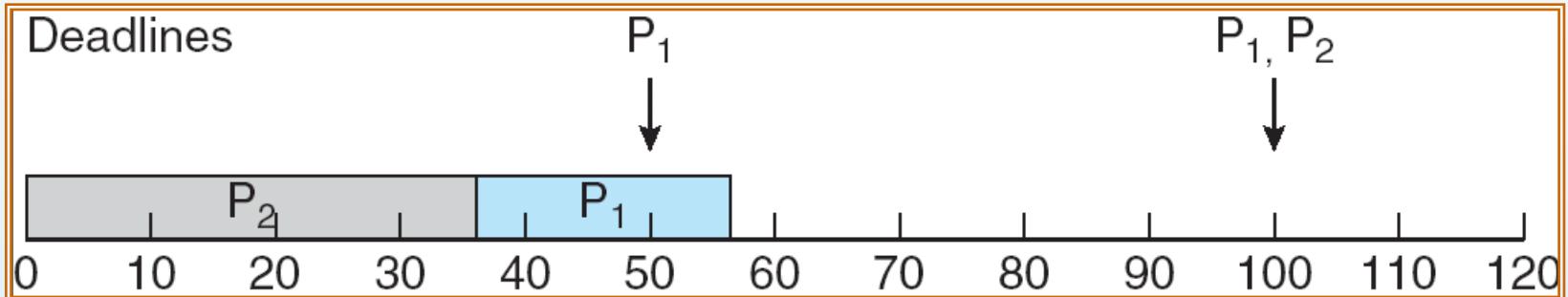
Real-Time CPU Scheduling

- 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





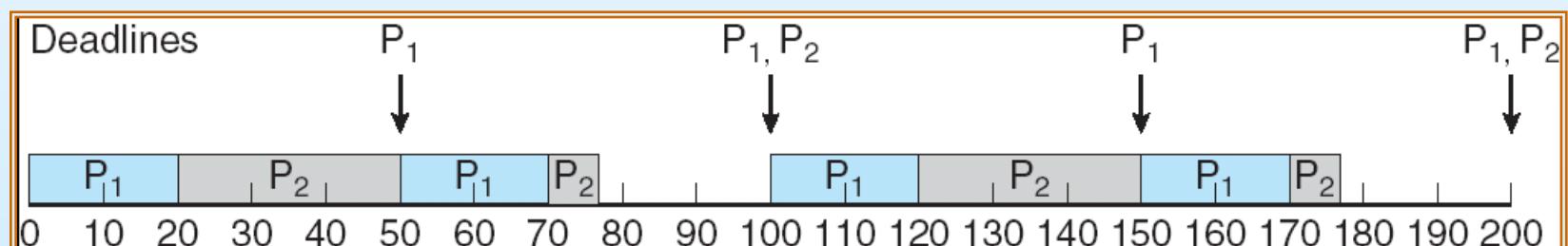
Scheduling of tasks when P_2 has a higher priority than P_1





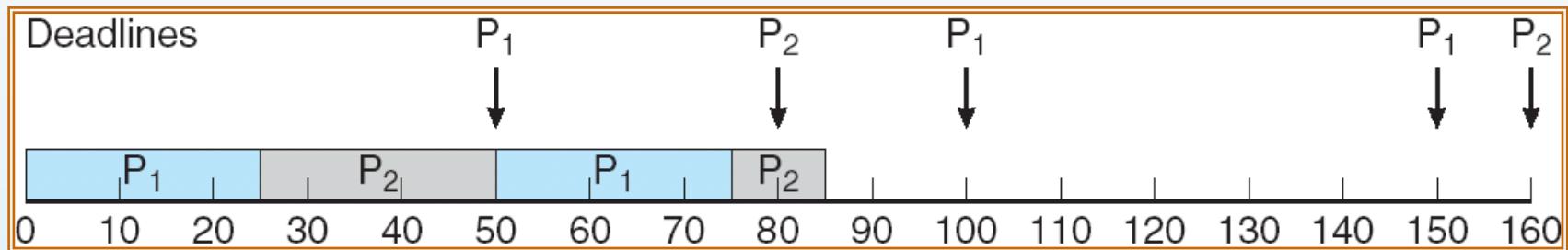
Rate Montonic Scheduling

- 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 .





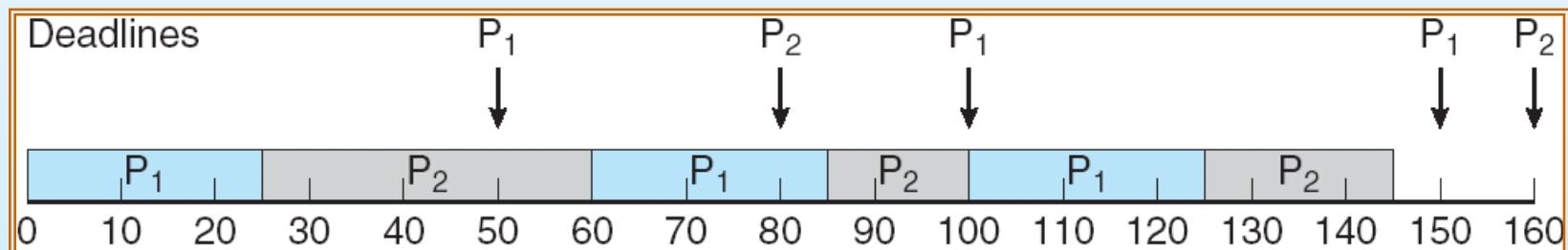
Missed Deadlines with Rate Monotonic Scheduling





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.





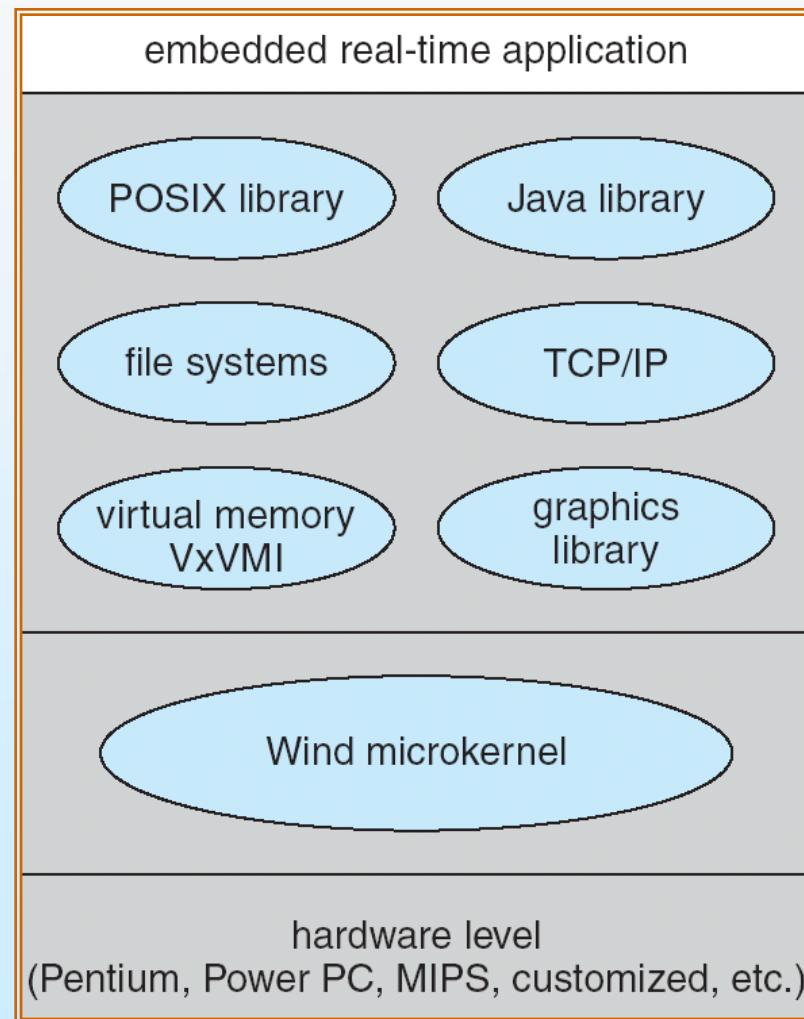
Pthread Scheduling

- The Pthread API provides functions for managing real-time threads.
- Pthreads 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.





VxWorks 5.0



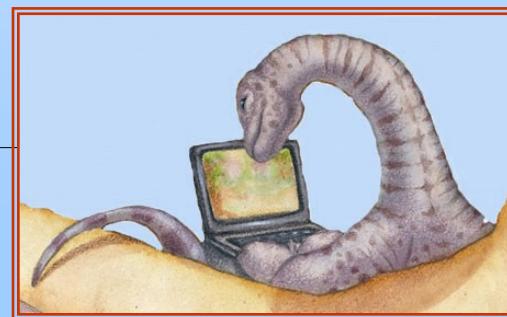


Wind Microkernel

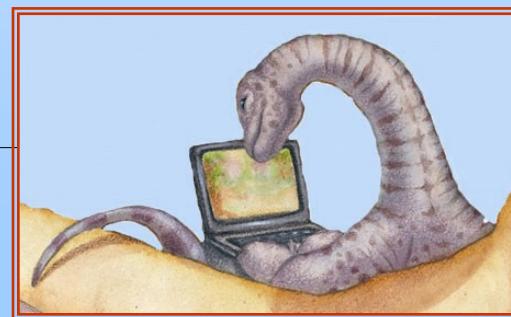
- The Wind microkernel provides support for the following:
 - (1) Processes and threads;
 - (2) preemptive and non-preemptive round-robin scheduling;
 - (3) manages interrupts (with bounded interrupt and dispatch latency times);
 - (4) shared memory and message passing interprocess communication facilities.



End of Chapter 19



Chapter 20: Multimedia Systems





Chapter 20: Multimedia Systems

- What is Multimedia
- Compression Techniques
- Requirements of Multimedia Kernels
- CPU Scheduling
- Disk Scheduling
- Network Management
- An Example: Cineblitz





Objectives

- To identify the characteristics of multimedia data
- To examine several algorithms used to compress multimedia data
- To explore the operating system requirements of multimedia data, including CPU and disk scheduling and network management





What is Multimedia?

- Multimedia data includes
 - audio and video clips (i.e. MP3 and MPEG files)
 - live webcasts

- Multimedia data may be delivered to
 - desktop PC's
 - handheld devices (PDAs, smart phones)





Media Delivery

- Multimedia data is stored in the file system like other ordinary data.
- However, multimedia data must be accessed with specific timing requirements.
- For example, video must be displayed at 24-30 **frames** per second. Multimedia video data must be delivered at a rate which guarantees 24-30 frames/second.
- **Continuous-media data** is data with specific rate requirements.





Streaming

- **Streaming** is delivering a multimedia file from a server to a client - typically the delivery occurs over a network connection.
- There are two different types of streaming:
 1. **Progressive download** - the client begins playback of the multimedia file as it is delivered. The file is ultimately stored on the client computer.
 2. **Real-time streaming** - the multimedia file is delivered to - but not stored on - the client's computer.





Real-time Streaming

- There are two types of real-time streaming:
 - (1) **Live streaming** - used to deliver a live event while it is occurring.
 - (2) **On-demand streaming** - used to deliver media streams such as movies, archived lectures, etc. The events are not delivered in real-time.





Multimedia Systems Characteristics

- Multimedia files can be quite large.
- Continuous media data may require very high data rates.
- Multimedia applications may be sensitive to timing delays during playback of the media.





Compression

- Because of the size and rate requirements of multimedia systems, multimedia files are often compressed into a smaller form.
- MPEG Compression:
 - (1) MPEG-1 - 352 X 240 @ 30 frames/second
 - (2) MPEG-2 - Used for compressing DVD and high-definition television (HDTV)
 - (3) MPEG-4 - Used to transmit audio, video, and graphics. Can be delivered over very slow connections (56 Kbps)





Operating Systems Issues

- The operating system must guarantee the specific data rate and timing requirements of continuous media.
- Such requirements are known as **Quality-of-Service (QoS)** guarantees.





QoS Guarantees

- Guaranteeing QoS has the following effects in a computer system:
 - (1) CPU processing
 - (2) Scheduling
 - (3) File systems
 - (4) Network protocols





Requirement of Multimedia Operating Systems

- There are three levels of QoS
 - (1) Best-effort service - the system makes a best effort with no QoS guarantees.
 - (2) Soft QoS - allows different traffic streams to be prioritized, however no QoS guarantees are made.
 - (3) Hard QoS - the QoS requirements are guaranteed.





Parameters Defining QoS

- Throughput - the total amount of work completed during a specific time interval.
- Delay - the elapsed time from when a request is first submitted to when the desired result is produced.
- Jitter - the delays that occur during playback of a stream.
- Reliability - how errors are handled during transmission and processing of continuous media.





Further QoS Issues

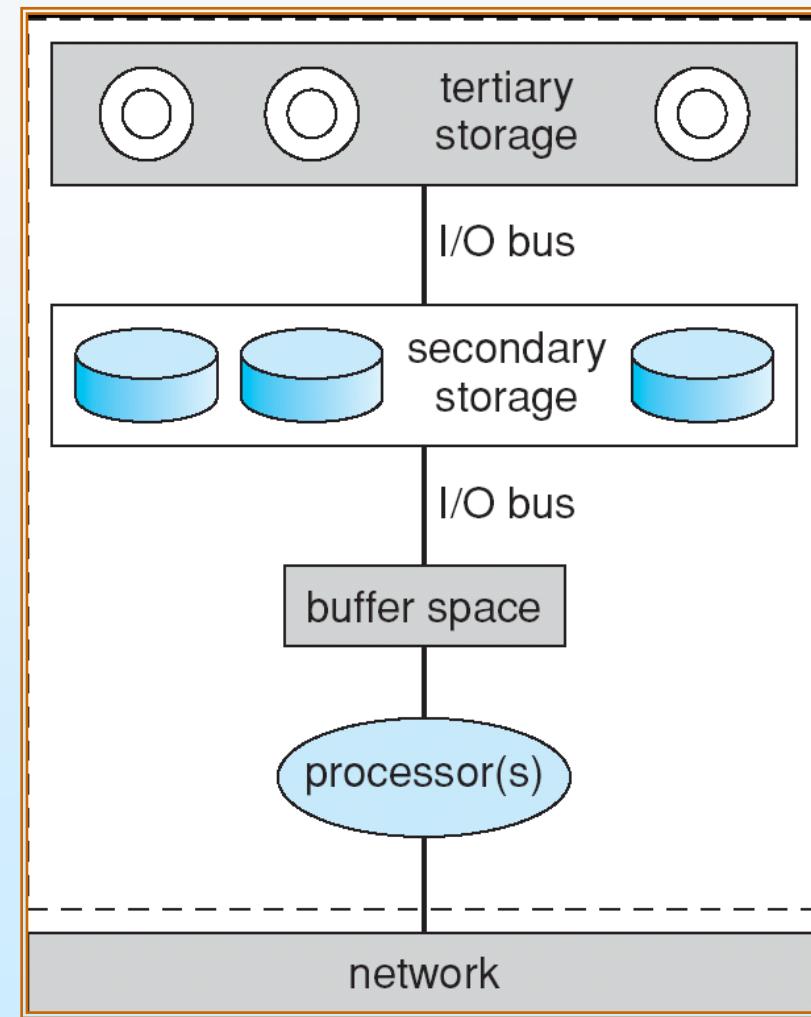
- QoS may be **negotiated** between the client and server.
- Operating systems often use an **admission control** algorithm that admits a request for a service only if the server has sufficient resources to satisfy the request.





Figure 20.1

Resources on a file server





CPU Scheduling

- Multimedia systems require **hard realtime** scheduling to ensure critical tasks will be serviced within timing deadlines.
- Most hard realtime CPU scheduling algorithms assign realtime processes static priorities that do not change over time.





Disk Scheduling

- Disk scheduling algorithms must be optimized to meet the timing deadlines and rate requirements of continuous media.
- Earliest-Deadline-First (EDF) Scheduling
- SCAN-EDF Scheduling





Disk Scheduling (cont)

- The EDF scheduler uses a queue to order requests according to the time it must be completed (its deadline.)

- SCAN-EDF scheduling is similar to EDF except that requests with the same deadline are ordered according to a SCAN policy.





Deadline and cylinder requests for SCAN-EDF scheduling

request	deadline	cylinder
A	150	25
B	201	112
C	399	95
D	94	31
E	295	185
F	78	85
G	165	150
H	125	101
I	300	85
J	210	90





Network Management

- Three general methods for delivering content from a server to a client across a network:
 - (1) **Unicasting** - the server delivers the content to a single client.
 - (2) **Broadcasting** - the server delivers the content to all clients, regardless whether they want the content or not.
 - (3) **Multicasting** - the server delivers the content to a group of receivers who indicate they wish to receive the content.





RealTime Streaming Protocol (RTSP)

- Standard HTTP is stateless whereby the server does not maintain the status of its connection with the client.





Figure 20.1

Streaming media from a conventional web server

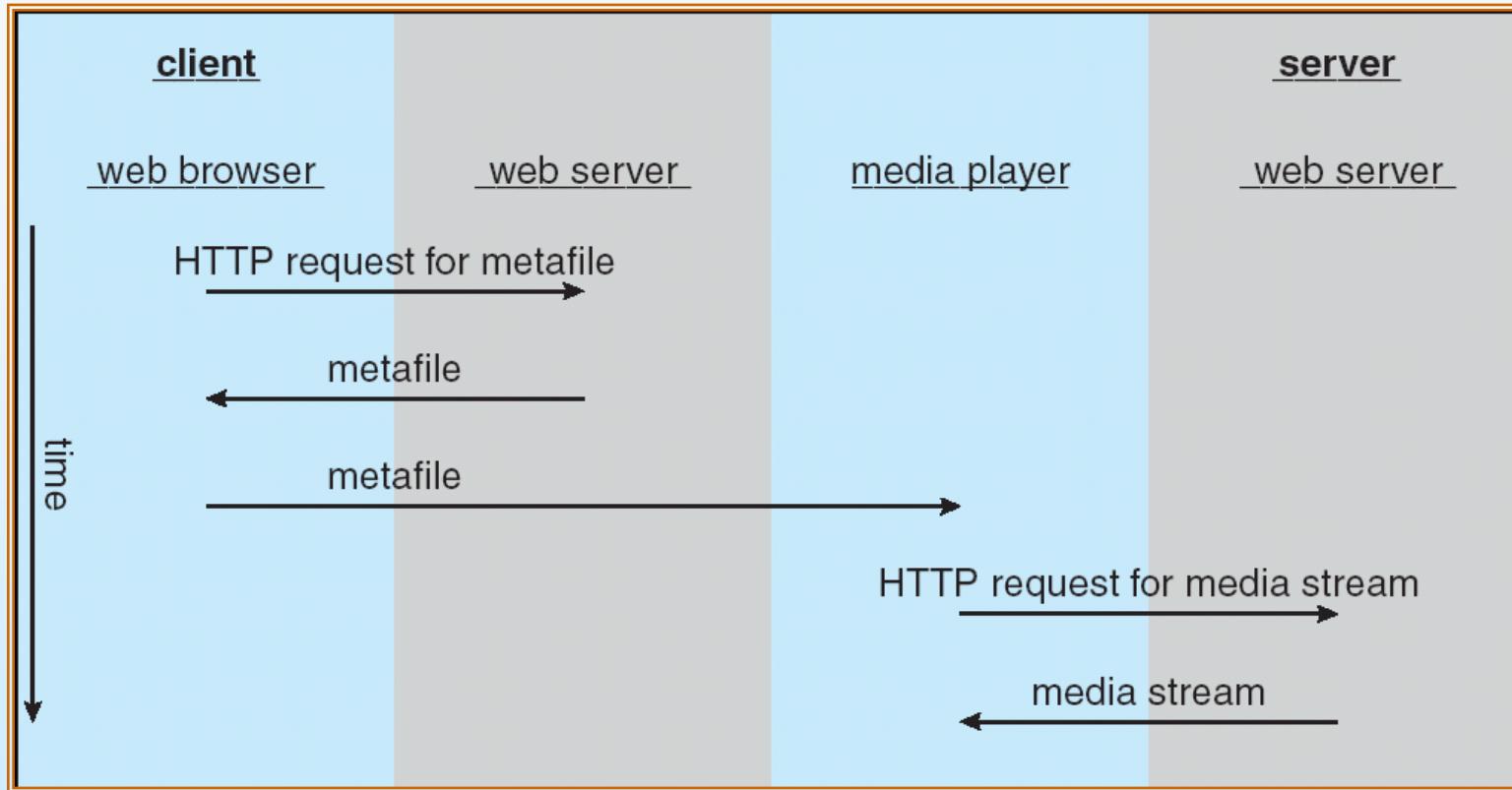
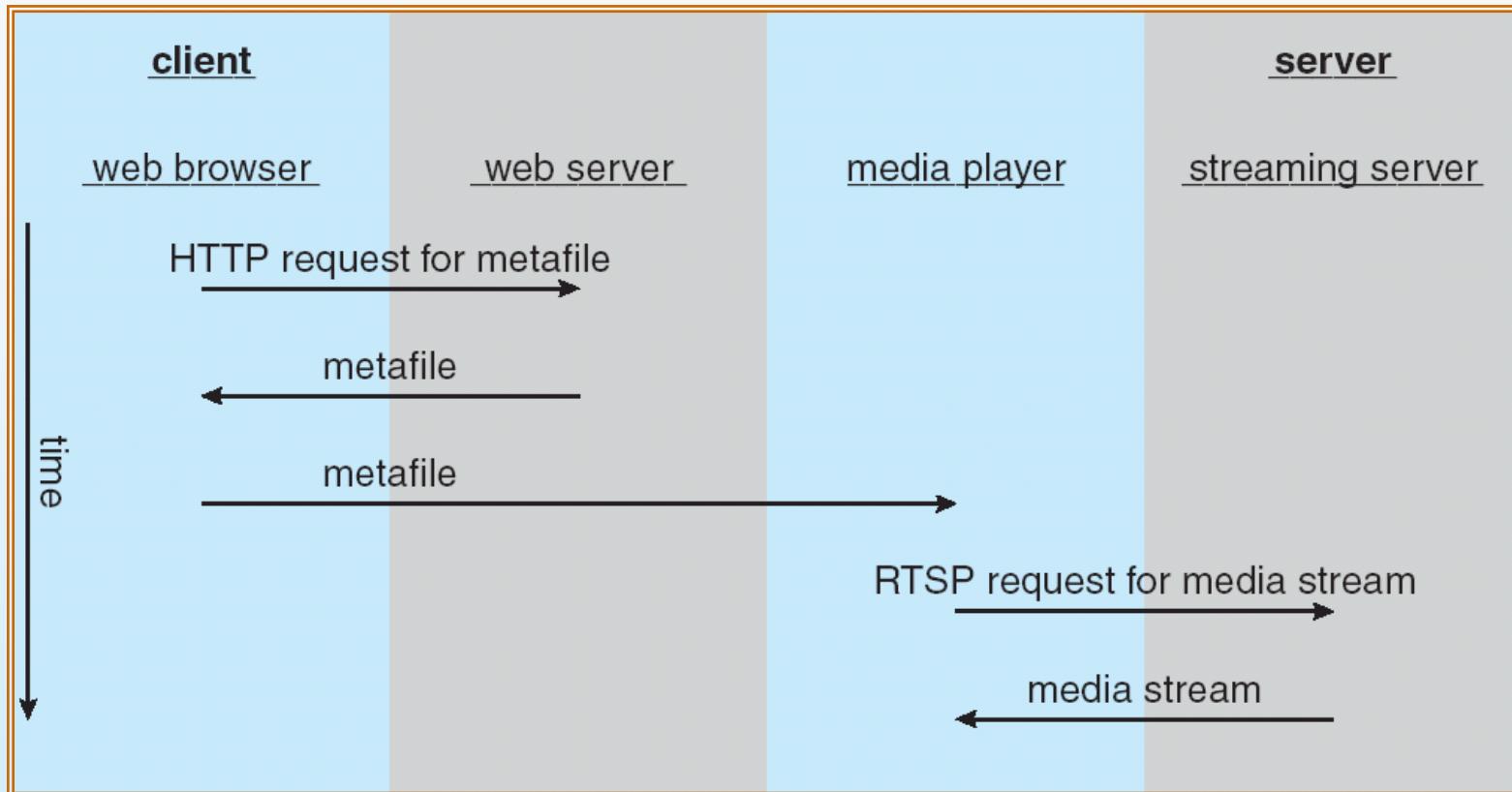




Figure 20.3

Realtime Streaming Protocol





RTSP States

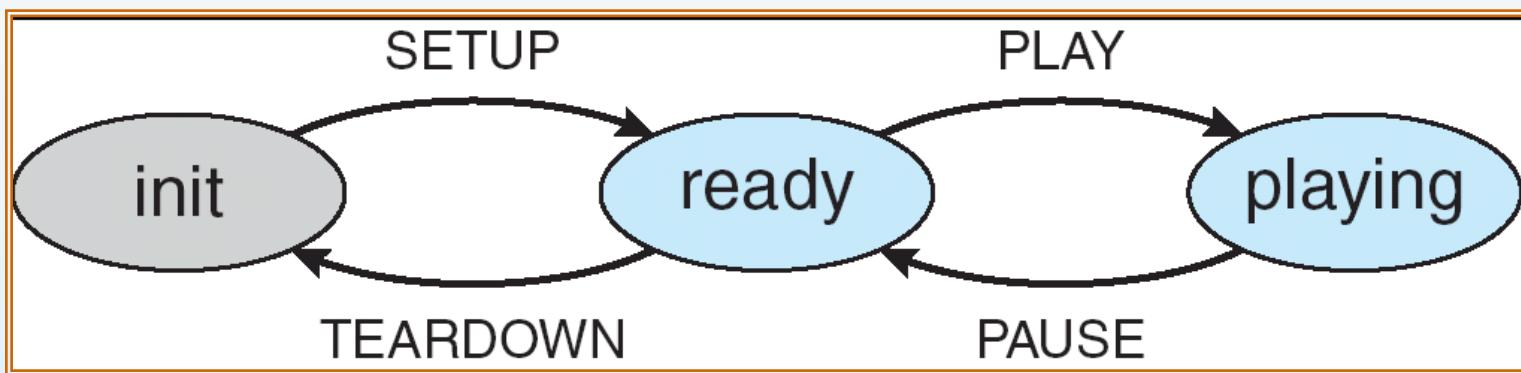
- SETUP - the server allocates resources for a client session.
- PLAY - the server delivers a stream to a client session.
- PAUSE - the server suspends delivery of a stream.
- TEARDOWN - the server breaks down the connection and releases the resources allocated for the session.





Figure 20.4

RTSP state machine





CineBlitz Multimedia Server

- CineBlitz supports both realtime and non-realtime clients.
- CineBlitz provides hard QoS guarantees to realtime clients using an admission control algorithm.
- The disk scheduler orders requests using C-SCAN order.





CineBlitz Admission Controller

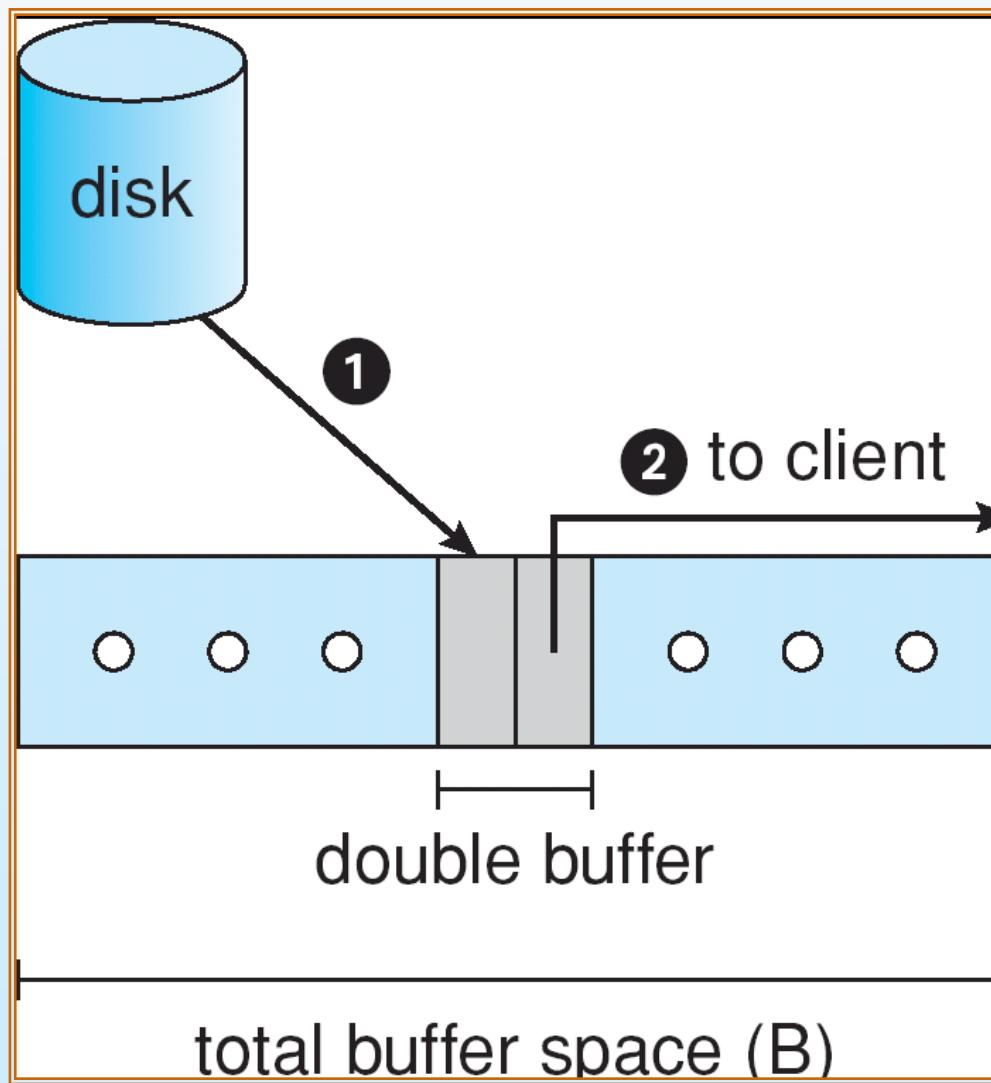
- Total buffer space required for N clients where client has rate requirement of r_i

$$\sum_{i=1}^N 2 \times T \times r_i \leq B.$$





Figure 20.05 Double buffering in CineBlitz





CineBlitz Admission Controller (cont)

- If tseek and trot are the worst-case seek and rotational delay times, the maximum latency for servicing N requests is

$$2 \times t_{seek} + \sum_{i=1}^N \left(\lceil \frac{T \times r_i}{b} \rceil + 1 \right) \times t_{rot}.$$





CineBlitz Admission Controller (cont)

- The CineBlitz admission controller only admits a new client if there is at least $2 \times T \times r_i$ bits of free buffer space and the following equation is satisfied

$$2 \times t_{seek} + \sum_{i=1}^N \left(\lceil \frac{T \times r_i}{b} \rceil + 1 \right) \times t_{rot} + \sum_{i=1}^N \frac{T \times r_i}{r_{disk}} \leq T.$$





In.20.1

request	deadline	cylinder
A	150	25
B	201	112
C	399	95
D	94	31
E	295	185
F	78	85
G	165	150
H	125	101
I	300	85
J	210	90





Exercise 20.10

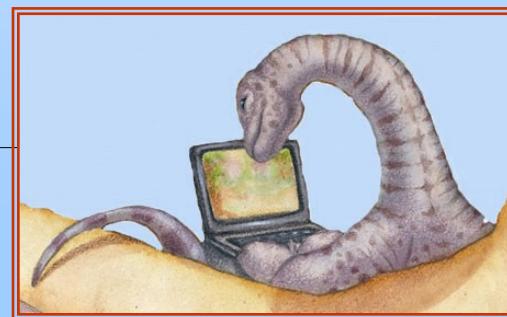
request	deadline	cylinder
R1	57	77
R2	300	95
R3	250	25
R4	88	28
R5	85	100
R6	110	90
R7	299	50
R8	300	77
R9	120	12
R10	212	2



End of Chapter 20



Chapter 21: The Linux System





Chapter 21: The Linux System

- Linux History
- Design Principles
- Kernel Modules
- Process Management
- Scheduling
- Memory Management
- File Systems
- Input and Output
- Interprocess Communication
- Network Structure
- Security





Objectives

- To explore the history of the UNIX operating system from which Linux is derived and the principles which Linux is designed upon
- To examine the Linux process model and illustrate how Linux schedules processes and provides interprocess communication
- To look at memory management in Linux
- To explore how Linux implements file systems and manages I/O devices





History

- Linux is a modern, free operating system based on UNIX standards
- First developed as a small but self-contained kernel in 1991 by Linus Torvalds, with the major design goal of UNIX compatibility
- Its history has been one of collaboration by many users from all around the world, corresponding almost exclusively over the Internet
- It has been designed to run efficiently and reliably on common PC hardware, but also runs on a variety of other platforms
- The core Linux operating system kernel is entirely original, but it can run much existing free UNIX software, resulting in an entire UNIX-compatible operating system free from proprietary code
- Many, varying **Linux Distributions** including the kernel, applications, and management tools





The Linux Kernel

- Version 0.01 (May 1991) had no networking, ran only on 80386-compatible Intel processors and on PC hardware, had extremely limited device-drive support, and supported only the Minix file system
- Linux 1.0 (March 1994) included these new features:
 - Support for UNIX's standard TCP/IP networking protocols
 - BSD-compatible socket interface for networking programming
 - Device-driver support for running IP over an Ethernet
 - Enhanced file system
 - Support for a range of SCSI controllers for high-performance disk access
 - Extra hardware support
- Version 1.2 (March 1995) was the final PC-only Linux kernel





Linux 2.0

- Released in June 1996, 2.0 added two major new capabilities:
 - Support for multiple architectures, including a fully 64-bit native Alpha port
 - Support for multiprocessor architectures
- Other new features included:
 - Improved memory-management code
 - Improved TCP/IP performance
 - Support for internal kernel threads, for handling dependencies between loadable modules, and for automatic loading of modules on demand
 - Standardized configuration interface
- Available for Motorola 68000-series processors, Sun Sparc systems, and for PC and PowerMac systems
- 2.4 and 2.6 increased SMP support, added journaling file system, preemptive kernel, 64-bit memory support





The Linux System

- Linux uses many tools developed as part of Berkeley's BSD operating system, MIT's X Window System, and the Free Software Foundation's GNU project
- The main system libraries were started by the GNU project, with improvements provided by the Linux community
- Linux networking-administration tools were derived from 4.3BSD code; recent BSD derivatives such as FreeBSD have borrowed code from Linux in return
- The Linux system is maintained by a loose network of developers collaborating over the Internet, with a small number of public ftp sites acting as de facto standard repositories





Linux Distributions

- Standard, precompiled sets of packages, or *distributions*, include the basic Linux system, system installation and management utilities, and ready-to-install packages of common UNIX tools
- The first distributions managed these packages by simply providing a means of unpacking all the files into the appropriate places; modern distributions include advanced package management
- Early distributions included SLS and Slackware
 - *Red Hat* and *Debian* are popular distributions from commercial and noncommercial sources, respectively
- The RPM Package file format permits compatibility among the various Linux distributions





Linux Licensing

- The Linux kernel is distributed under the GNU General Public License (GPL), the terms of which are set out by the Free Software Foundation
- Anyone using Linux, or creating their own derivative of Linux, may not make the derived product proprietary; software released under the GPL may not be redistributed as a binary-only product





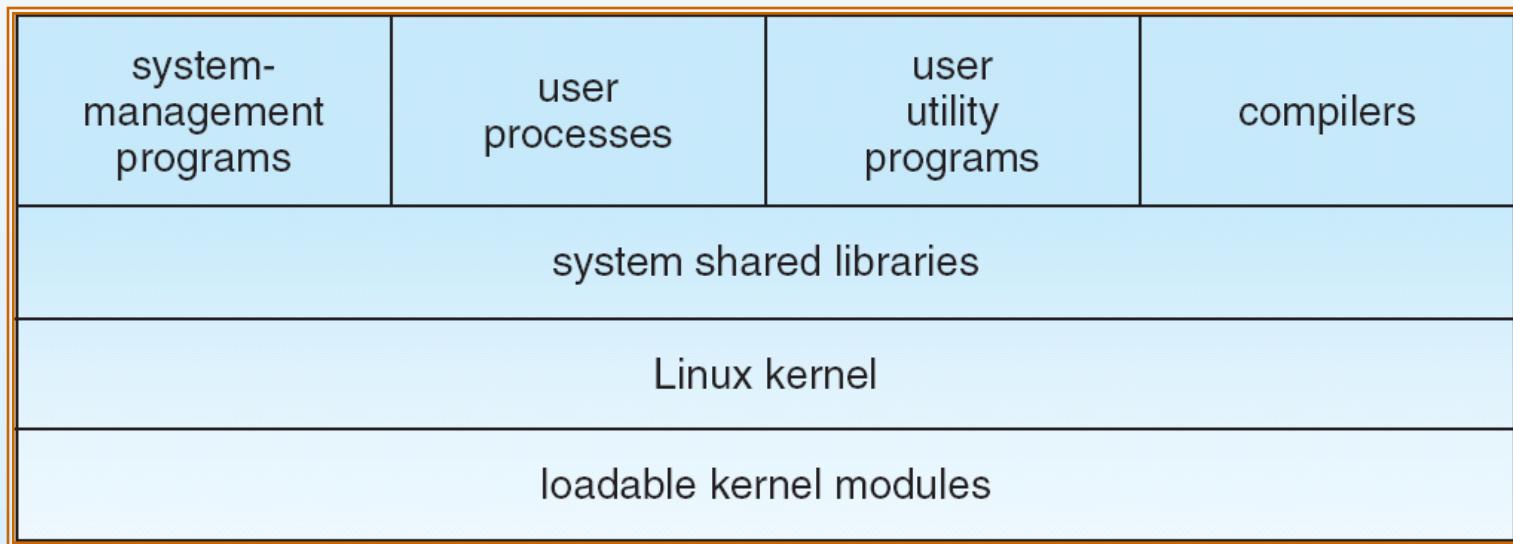
Design Principles

- Linux is a multiuser, multitasking system with a full set of UNIX-compatible tools
- Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model
- Main design goals are speed, efficiency, and standardization
- Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification
- The Linux programming interface adheres to the SVR4 UNIX semantics, rather than to BSD behavior





Components of a Linux System





Components of a Linux System (Cont.)

- Like most UNIX implementations, Linux is composed of three main bodies of code; the most important distinction between the kernel and all other components
- The **kernel** is responsible for maintaining the important abstractions of the operating system
 - Kernel code executes in *kernel mode* with full access to all the physical resources of the computer
 - All kernel code and data structures are kept in the same single address space





Components of a Linux System (Cont.)

- The **system libraries** define a standard set of functions through which applications interact with the kernel, and which implement much of the operating-system functionality that does not need the full privileges of kernel code

- The **system utilities** perform individual specialized management tasks





Kernel Modules

- Sections of kernel code that can be compiled, loaded, and unloaded independent of the rest of the kernel
- A kernel module may typically implement a device driver, a file system, or a networking protocol
- The module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL
- Kernel modules allow a Linux system to be set up with a standard, minimal kernel, without any extra device drivers built in
- Three components to Linux module support:
 - module management
 - driver registration
 - conflict resolution





Module Management

- Supports loading modules into memory and letting them talk to the rest of the kernel
- Module loading is split into two separate sections:
 - Managing sections of module code in kernel memory
 - Handling symbols that modules are allowed to reference
- The module requestor manages loading requested, but currently unloaded, modules; it also regularly queries the kernel to see whether a dynamically loaded module is still in use, and will unload it when it is no longer actively needed





Driver Registration

- Allows modules to tell the rest of the kernel that a new driver has become available
- The kernel maintains dynamic tables of all known drivers, and provides a set of routines to allow drivers to be added to or removed from these tables at any time
- Registration tables include the following items:
 - Device drivers
 - File systems
 - Network protocols
 - Binary format





Conflict Resolution

- A mechanism that allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver
- The conflict resolution module aims to:
 - Prevent modules from clashing over access to hardware resources
 - Prevent *autoprobes* from interfering with existing device drivers
 - Resolve conflicts with multiple drivers trying to access the same hardware





Process Management

- UNIX process management separates the creation of processes and the running of a new program into two distinct operations.
 - The **fork** system call creates a new process
 - A new program is run after a call to **execve**
- Under UNIX, a process encompasses all the information that the operating system must maintain to track the context of a single execution of a single program
- Under Linux, process properties fall into three groups: the process's identity, environment, and context





Process Identity

- Process ID (PID). The unique identifier for the process; used to specify processes to the operating system when an application makes a system call to signal, modify, or wait for another process
- Credentials. Each process must have an associated user ID and one or more group IDs that determine the process's rights to access system resources and files
- Personality. Not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can slightly modify the semantics of certain system calls
 - Used primarily by emulation libraries to request that system calls be compatible with certain specific flavors of UNIX





Process Environment

- The process's environment is inherited from its parent, and is composed of two null-terminated vectors:
 - The argument vector lists the command-line arguments used to invoke the running program; conventionally starts with the name of the program itself
 - The environment vector is a list of "NAME=VALUE" pairs that associates named environment variables with arbitrary textual values
- Passing environment variables among processes and inheriting variables by a process's children are flexible means of passing information to components of the user-mode system software
- The environment-variable mechanism provides a customization of the operating system that can be set on a per-process basis, rather than being configured for the system as a whole





Process Context

- The (constantly changing) state of a running program at any point in time
- The **scheduling context** is the most important part of the process context; it is the information that the scheduler needs to suspend and restart the process
- The kernel maintains **accounting** information about the resources currently being consumed by each process, and the total resources consumed by the process in its lifetime so far
- The **file table** is an array of pointers to kernel file structures
 - When making file I/O system calls, processes refer to files by their index into this table





Process Context (Cont.)

- Whereas the file table lists the existing open files, the **file-system context** applies to requests to open new files
 - The current root and default directories to be used for new file searches are stored here
- The **signal-handler table** defines the routine in the process's address space to be called when specific signals arrive
- The **virtual-memory context** of a process describes the full contents of its private address space





Processes and Threads

- Linux uses the same internal representation for processes and threads; a thread is simply a new process that happens to share the same address space as its parent
- A distinction is only made when a new thread is created by the **clone** system call
 - **fork** creates a new process with its own entirely new process context
 - **clone** creates a new process with its own identity, but that is allowed to share the data structures of its parent
- Using **clone** gives an application fine-grained control over exactly what is shared between two threads





Scheduling

- The job of allocating CPU time to different tasks within an operating system
- While scheduling is normally thought of as the running and interrupting of processes, in Linux, scheduling also includes the running of the various kernel tasks
- Running kernel tasks encompasses both tasks that are requested by a running process and tasks that execute internally on behalf of a device driver
- As of 2.5, new scheduling algorithm – preemptive, priority-based
 - Real-time range
 - nice value





Relationship Between Priorities and Time-slice Length

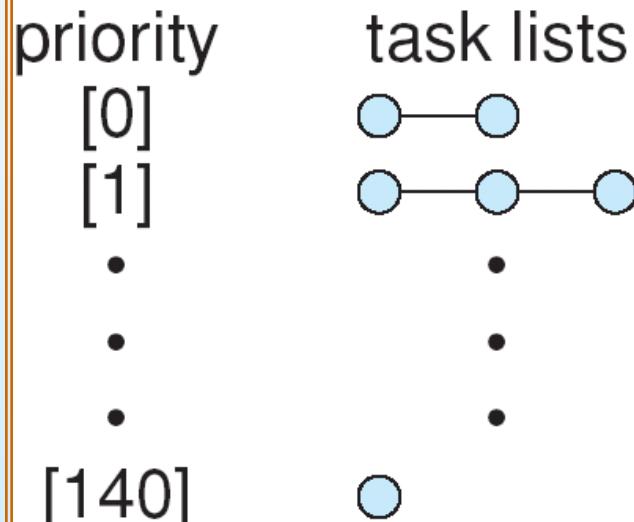
numeric priority	relative priority	time quantum
0	highest	200 ms
•		
•		
•		
99		
100		
•		
•		
•		
140	lowest	10 ms



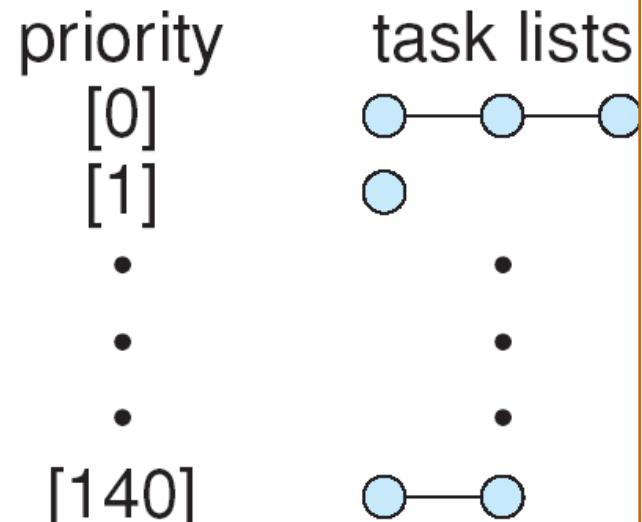


List of Tasks Indexed by Priority

**active
array**



**expired
array**





Kernel Synchronization

- A request for kernel-mode execution can occur in two ways:
 - A running program may request an operating system service, either explicitly via a system call, or implicitly, for example, when a page fault occurs
 - A device driver may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt
- Kernel synchronization requires a framework that will allow the kernel's critical sections to run without interruption by another critical section





Kernel Synchronization (Cont.)

- Linux uses two techniques to protect critical sections:
 1. Normal kernel code is nonpreemptible (until 2.4)
 - when a time interrupt is received while a process is executing a kernel system service routine, the kernel's **need_resched** flag is set so that the scheduler will run once the system call has completed and control is about to be returned to user mode
 2. The second technique applies to critical sections that occur in an interrupt service routines
 - By using the processor's interrupt control hardware to disable interrupts during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access of shared data structures





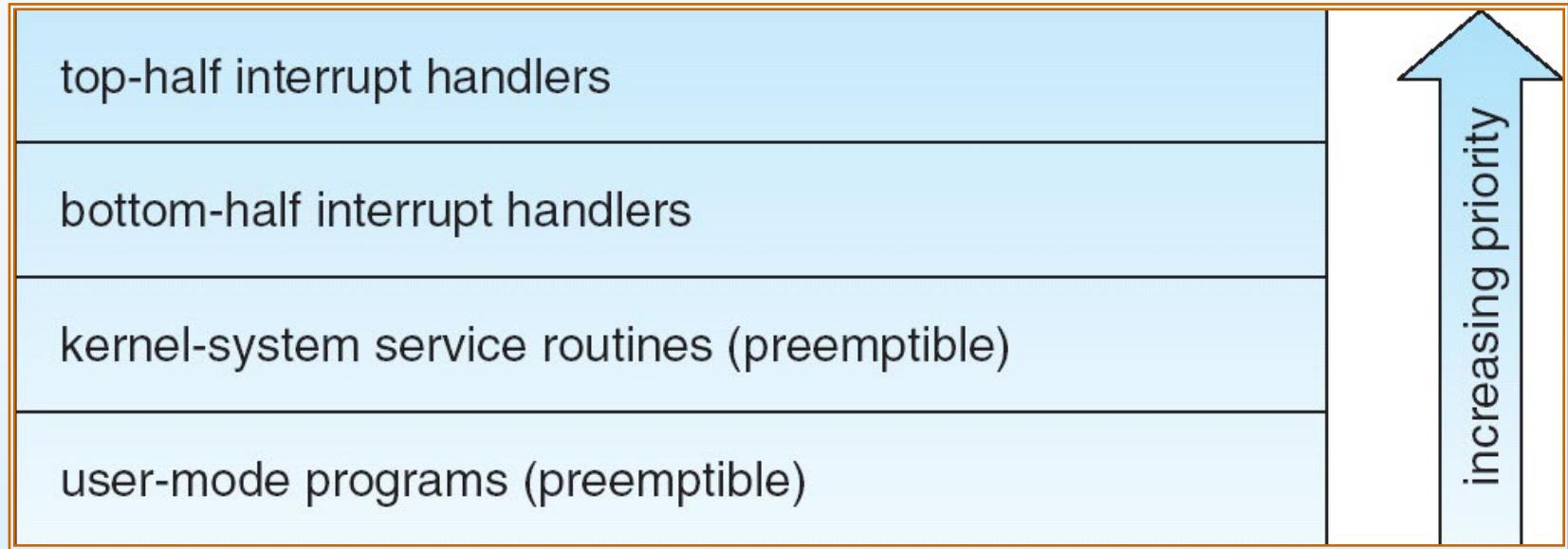
Kernel Synchronization (Cont.)

- To avoid performance penalties, Linux's kernel uses a synchronization architecture that allows long critical sections to run without having interrupts disabled for the critical section's entire duration
- Interrupt service routines are separated into a *top half* and a *bottom half*.
 - The top half is a normal interrupt service routine, and runs with recursive interrupts disabled
 - The bottom half is run, with all interrupts enabled, by a miniature scheduler that ensures that bottom halves never interrupt themselves
 - This architecture is completed by a mechanism for disabling selected bottom halves while executing normal, foreground kernel code





Interrupt Protection Levels



- Each level may be interrupted by code running at a higher level, but will never be interrupted by code running at the same or a lower level
- User processes can always be preempted by another process when a time-sharing scheduling interrupt occurs





Process Scheduling

- Linux uses two process-scheduling algorithms:
 - A time-sharing algorithm for fair preemptive scheduling between multiple processes
 - A real-time algorithm for tasks where absolute priorities are more important than fairness
- A process's scheduling class defines which algorithm to apply
- For time-sharing processes, Linux uses a prioritized, credit based algorithm
 - The crediting rule

$$\text{credits} := \frac{\text{credits}}{2} + \text{priority}$$

factors in both the process's history and its priority

- This crediting system automatically prioritizes interactive or I/O-bound processes





Process Scheduling (Cont.)

- Linux implements the FIFO and round-robin real-time scheduling classes; in both cases, each process has a priority in addition to its scheduling class
 - The scheduler runs the process with the highest priority; for equal-priority processes, it runs the process waiting the longest
 - FIFO processes continue to run until they either exit or block
 - A round-robin process will be preempted after a while and moved to the end of the scheduling queue, so that round-robbing processes of equal priority automatically time-share between themselves





Symmetric Multiprocessing

- Linux 2.0 was the first Linux kernel to support SMP hardware; separate processes or threads can execute in parallel on separate processors

- To preserve the kernel's nonpreemptible synchronization requirements, SMP imposes the restriction, via a single kernel spinlock, that only one processor at a time may execute kernel-mode code





Memory Management

- Linux's physical memory-management system deals with allocating and freeing pages, groups of pages, and small blocks of memory
- It has additional mechanisms for handling virtual memory, memory mapped into the address space of running processes
- Splits memory into 3 different **zones** due to hardware characteristics





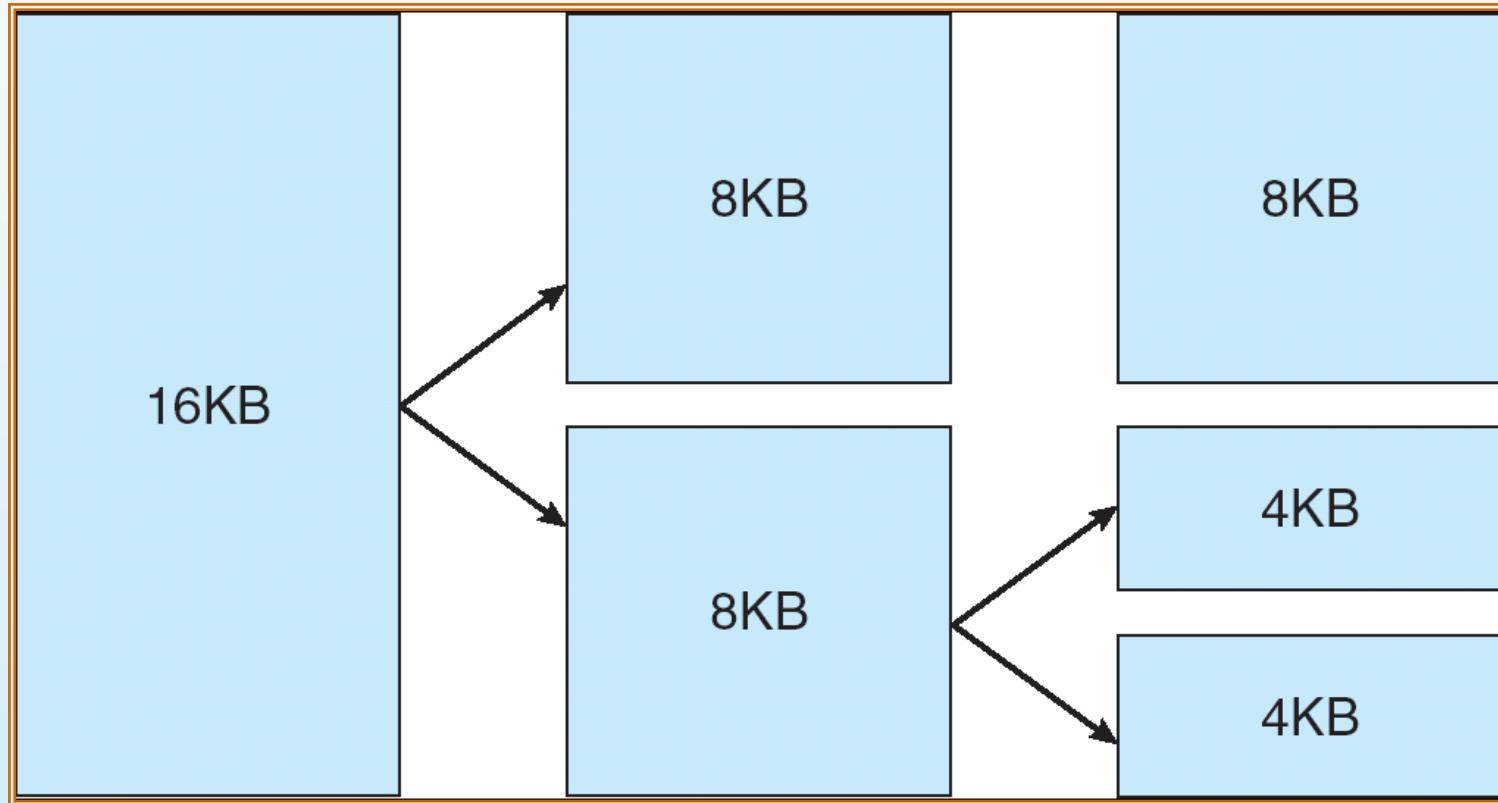
Relationship of Zones and Physical Addresses on 80x86

zone	physical memory
ZONE_DMA	< 16 MB
ZONE_NORMAL	16 .. 896 MB
ZONE_HIGHMEM	> 896 MB





Splitting of Memory in a Buddy Heap

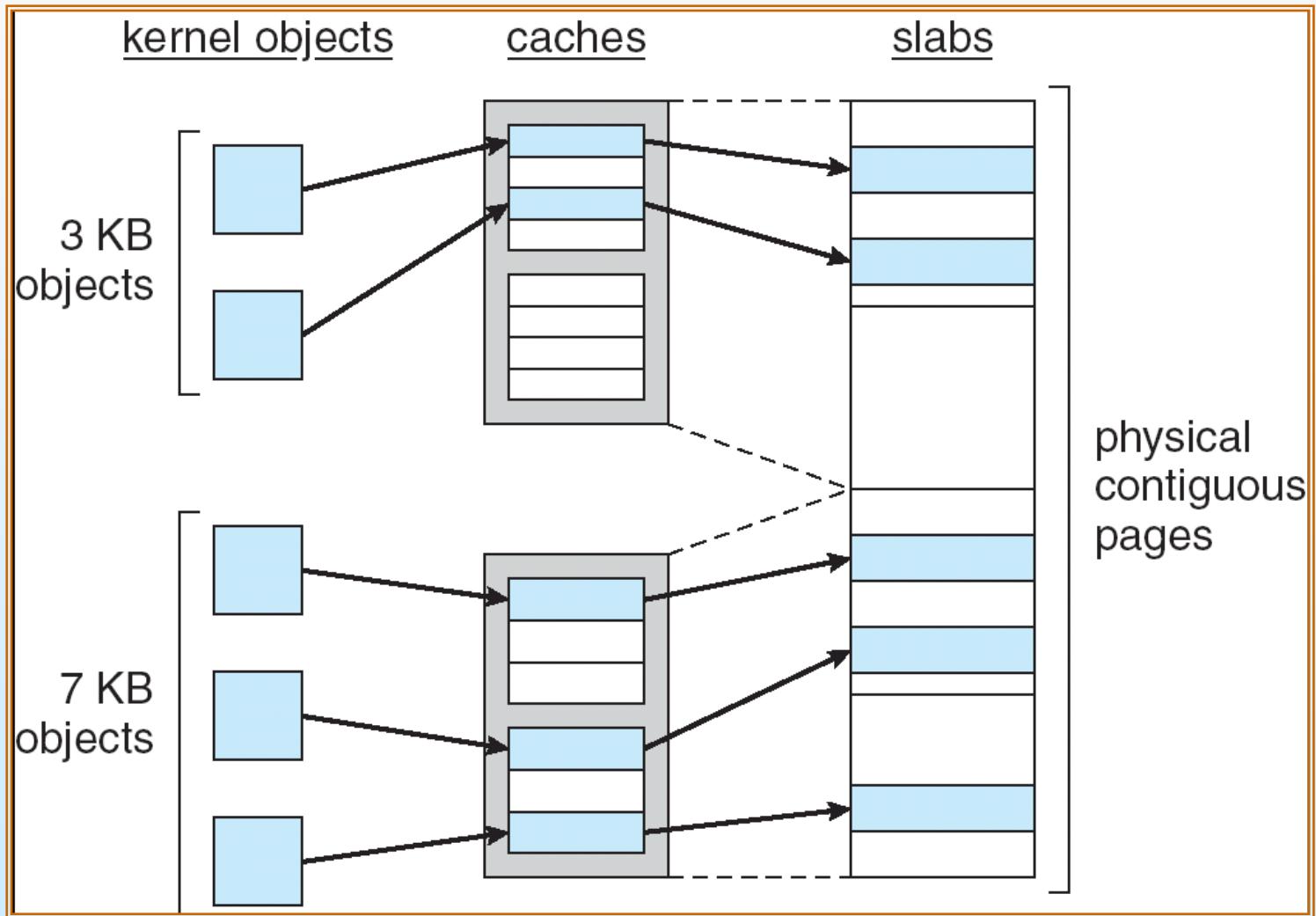




Managing Physical Memory

- The page allocator allocates and frees all physical pages; it can allocate ranges of physically-contiguous pages on request
- The allocator uses a buddy-heap algorithm to keep track of available physical pages
 - Each allocatable memory region is paired with an adjacent partner
 - Whenever two allocated partner regions are both freed up they are combined to form a larger region
 - If a small memory request cannot be satisfied by allocating an existing small free region, then a larger free region will be subdivided into two partners to satisfy the request
- Memory allocations in the Linux kernel occur either statically (drivers reserve a contiguous area of memory during system boot time) or dynamically (via the page allocator)
- Also uses **slab allocator** for kernel memory







Virtual Memory

- The VM system maintains the address space visible to each process: It creates pages of virtual memory on demand, and manages the loading of those pages from disk or their swapping back out to disk as required
- The VM manager maintains two separate views of a process's address space:
 - A logical view describing instructions concerning the layout of the address space
 - ▶ The address space consists of a set of nonoverlapping regions, each representing a continuous, page-aligned subset of the address space
 - A physical view of each address space which is stored in the hardware page tables for the process





Virtual Memory (Cont.)

- Virtual memory regions are characterized by:
 - The backing store, which describes from where the pages for a region come; regions are usually backed by a file or by nothing (*demand-zero* memory)
 - The region's reaction to writes (page sharing or copy-on-write)
- The kernel creates a new virtual address space
 1. When a process runs a new program with the **exec** system call
 2. Upon creation of a new process by the **fork** system call





Virtual Memory (Cont.)

- On executing a new program, the process is given a new, completely empty virtual-address space; the program-loading routines populate the address space with virtual-memory regions
- Creating a new process with **fork** involves creating a complete copy of the existing process's virtual address space
 - The kernel copies the parent process's VMA descriptors, then creates a new set of page tables for the child
 - The parent's page tables are copied directly into the child's, with the reference count of each page covered being incremented
 - After the fork, the parent and child share the same physical pages of memory in their address spaces





Virtual Memory (Cont.)

- The VM paging system relocates pages of memory from physical memory out to disk when the memory is needed for something else
- The VM paging system can be divided into two sections:
 - The pageout-policy algorithm decides which pages to write out to disk, and when
 - The paging mechanism actually carries out the transfer, and pages data back into physical memory as needed





Virtual Memory (Cont.)

- The Linux kernel reserves a constant, architecture-dependent region of the virtual address space of every process for its own internal use
- This kernel virtual-memory area contains two regions:
 - A static area that contains page table references to every available physical page of memory in the system, so that there is a simple translation from physical to virtual addresses when running kernel code
 - The remainder of the reserved section is not reserved for any specific purpose; its page-table entries can be modified to point to any other areas of memory





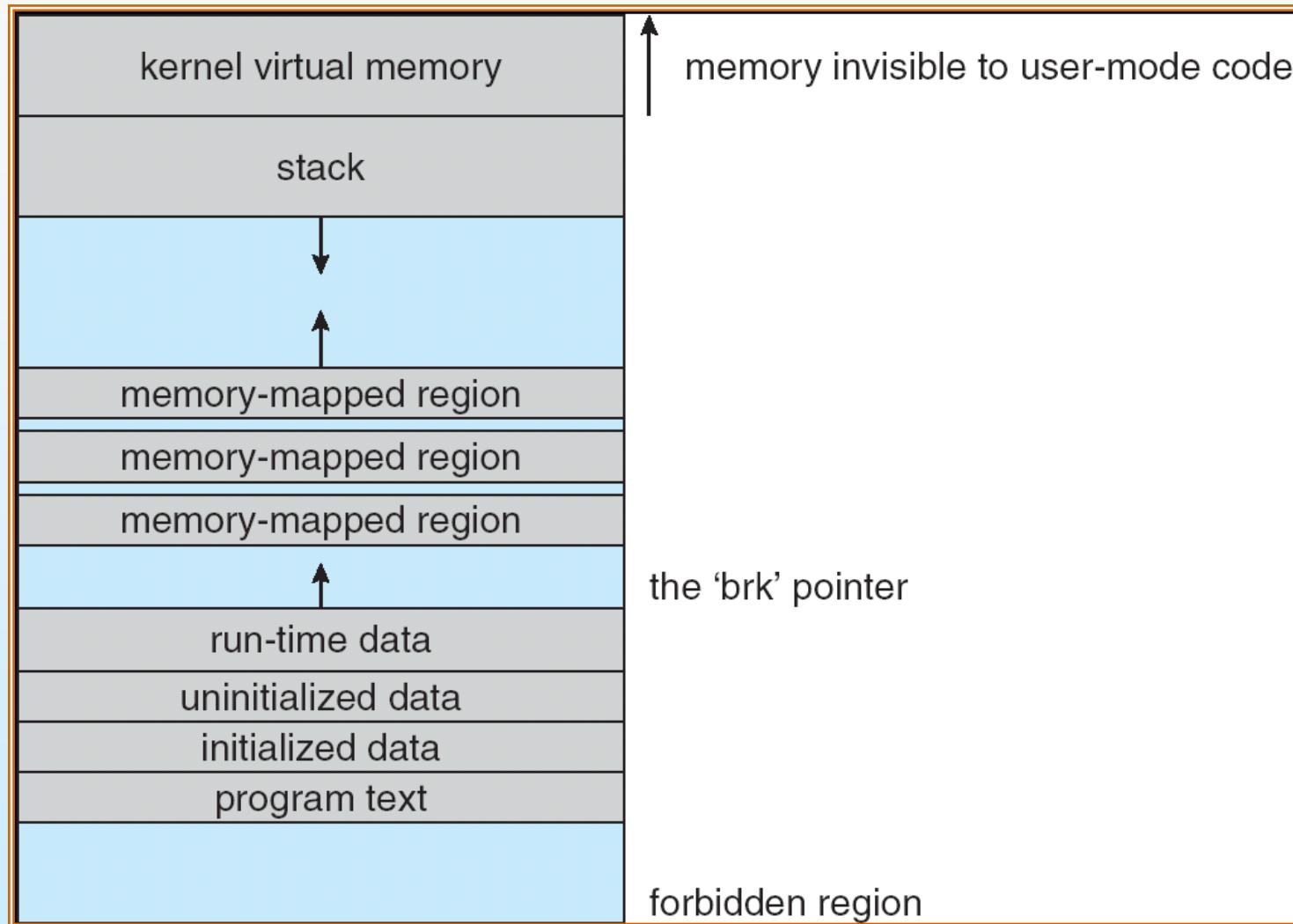
Executing and Loading User Programs

- Linux maintains a table of functions for loading programs; it gives each function the opportunity to try loading the given file when an exec system call is made
- The registration of multiple loader routines allows Linux to support both the ELF and **a.out** binary formats
- Initially, binary-file pages are mapped into virtual memory
 - Only when a program tries to access a given page will a page fault result in that page being loaded into physical memory
- An ELF-format binary file consists of a header followed by several page-aligned sections
 - The ELF loader works by reading the header and mapping the sections of the file into separate regions of virtual memory





Memory Layout for ELF Programs





Static and Dynamic Linking

- A program whose necessary library functions are embedded directly in the program's executable binary file is *statically* linked to its libraries
- The main disadvantage of static linkage is that every program generated must contain copies of exactly the same common system library functions
- *Dynamic* linking is more efficient in terms of both physical memory and disk-space usage because it loads the system libraries into memory only once





File Systems

- To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics
- Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the *virtual file system (VFS)*
- The Linux VFS is designed around object-oriented principles and is composed of two components:
 - A set of definitions that define what a file object is allowed to look like
 - ▶ The *inode-object* and the *file-object* structures represent individual files
 - ▶ the *file system object* represents an entire file system
 - A layer of software to manipulate those objects





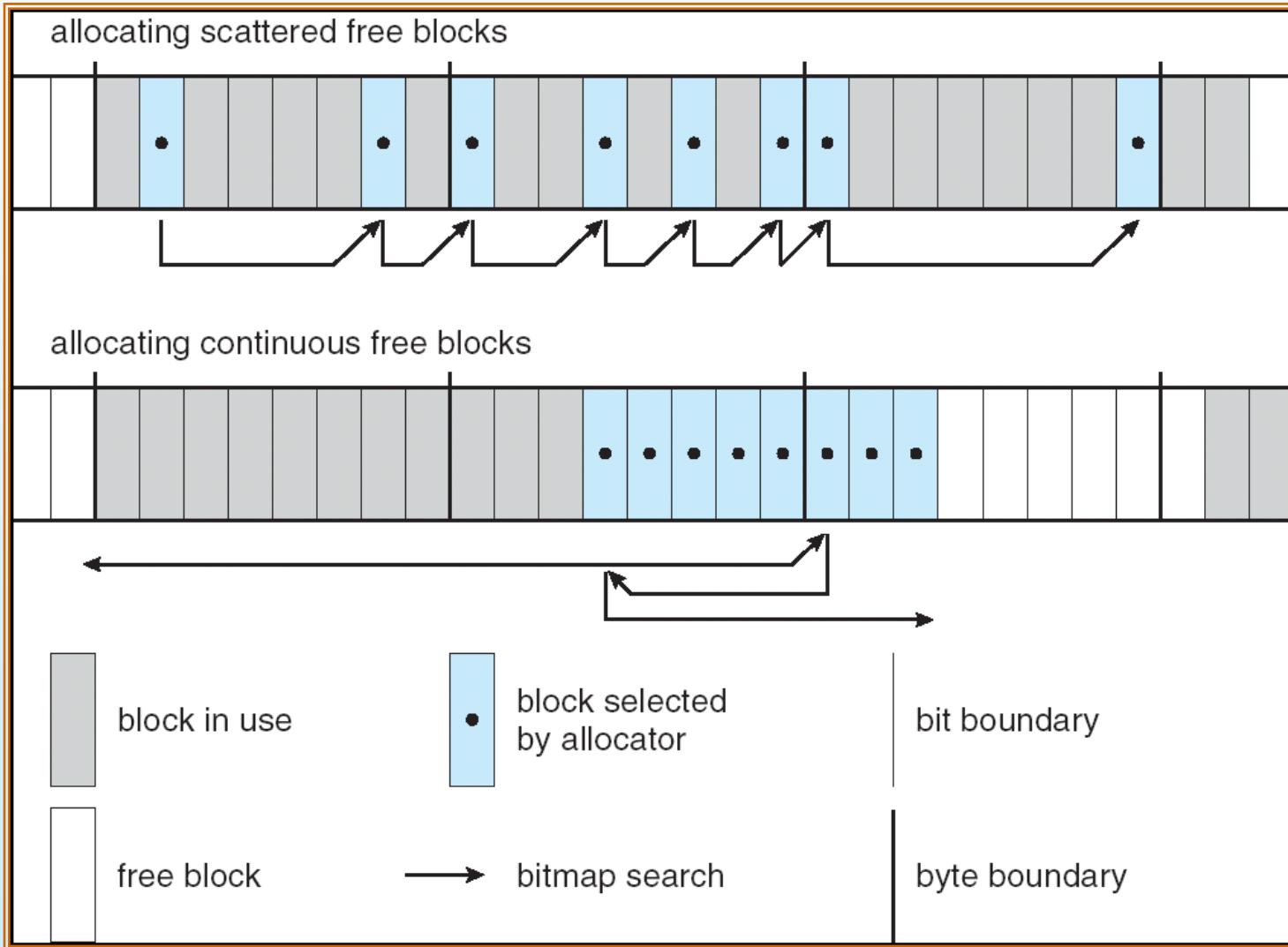
The Linux Ext2fs File System

- Ext2fs uses a mechanism similar to that of BSD Fast File System (ffs) for locating data blocks belonging to a specific file
- The main differences between ext2fs and ffs concern their disk allocation policies
 - In ffs, the disk is allocated to files in blocks of 8Kb, with blocks being subdivided into fragments of 1Kb to store small files or partially filled blocks at the end of a file
 - Ext2fs does not use fragments; it performs its allocations in smaller units
 - ▶ The default block size on ext2fs is 1Kb, although 2Kb and 4Kb blocks are also supported
 - Ext2fs uses allocation policies designed to place logically adjacent blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation





Ext2fs Block-Allocation Policies





The Linux Proc File System

- The **proc** file system does not store data, rather, its contents are computed on demand according to user file I/O requests
- **proc** must implement a directory structure, and the file contents within; it must then define a unique and persistent inode number for each directory and files it contains
 - It uses this inode number to identify just what operation is required when a user tries to read from a particular file inode or perform a lookup in a particular directory inode
 - When data is read from one of these files, **proc** collects the appropriate information, formats it into text form and places it into the requesting process's read buffer





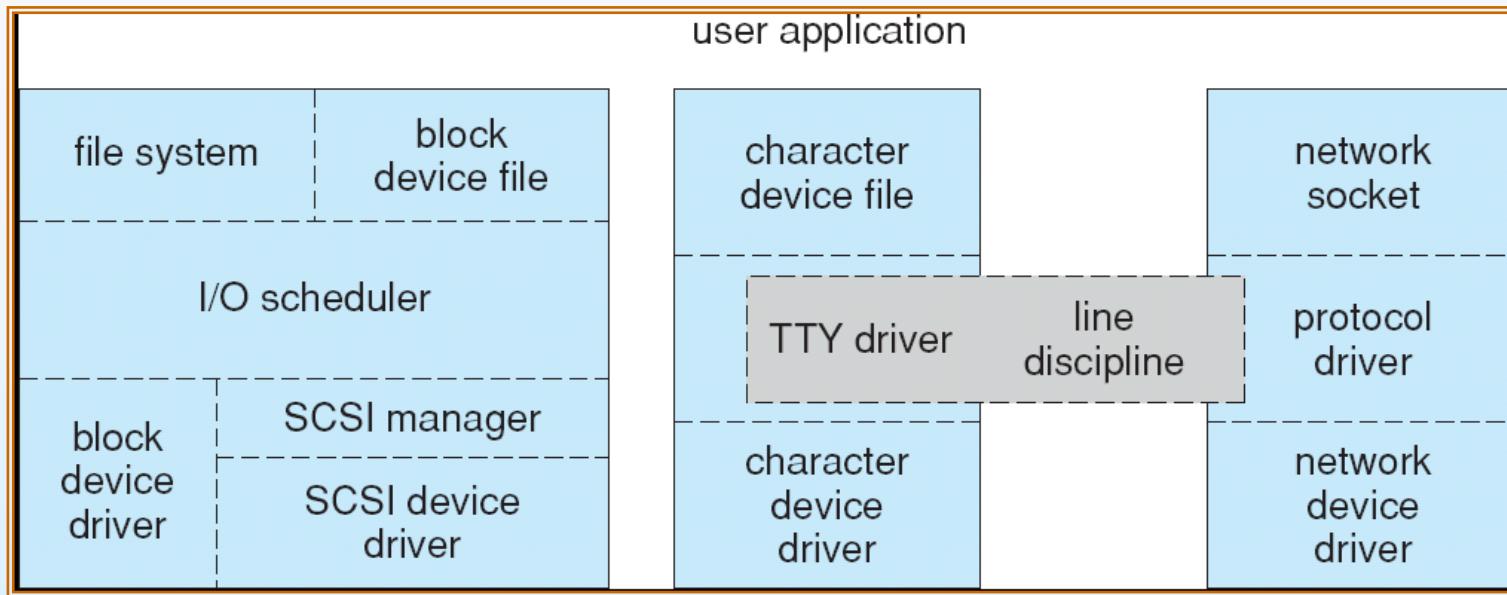
Input and Output

- The Linux device-oriented file system accesses disk storage through two caches:
 - Data is cached in the page cache, which is unified with the virtual memory system
 - Metadata is cached in the buffer cache, a separate cache indexed by the physical disk block
- Linux splits all devices into three classes:
 - *block devices* allow random access to completely independent, fixed size blocks of data
 - *character devices* include most other devices; they don't need to support the functionality of regular files
 - *network devices* are interfaced via the kernel's networking subsystem





Device-Driver Block Structure





Block Devices

- Provide the main interface to all disk devices in a system
- The *block buffer* cache serves two main purposes:
 - it acts as a pool of buffers for active I/O
 - it serves as a cache for completed I/O
- The *request manager* manages the reading and writing of buffer contents to and from a block device driver





Character Devices

- A device driver which does not offer random access to fixed blocks of data
- A character device driver must register a set of functions which implement the driver's various file I/O operations
- The kernel performs almost no preprocessing of a file read or write request to a character device, but simply passes on the request to the device
- The main exception to this rule is the special subset of character device drivers which implement terminal devices, for which the kernel maintains a standard interface





Interprocess Communication

- Like UNIX, Linux informs processes that an event has occurred via signals
- There is a limited number of signals, and they cannot carry information: Only the fact that a signal occurred is available to a process
- The Linux kernel does not use signals to communicate with processes that are running in kernel mode, rather, communication within the kernel is accomplished via scheduling states and **wait.queue** structures





Passing Data Between Processes

- The pipe mechanism allows a child process to inherit a communication channel to its parent, data written to one end of the pipe can be read at the other
- Shared memory offers an extremely fast way of communicating; any data written by one process to a shared memory region can be read immediately by any other process that has mapped that region into its address space
- To obtain synchronization, however, shared memory must be used in conjunction with another Interprocess-communication mechanism





Shared Memory Object

- The shared-memory object acts as a backing store for shared-memory regions in the same way as a file can act as backing store for a memory-mapped memory region
- Shared-memory mappings direct page faults to map in pages from a persistent shared-memory object
- Shared-memory objects remember their contents even if no processes are currently mapping them into virtual memory





Network Structure

- Networking is a key area of functionality for Linux.
 - It supports the standard Internet protocols for UNIX to UNIX communications
 - It also implements protocols native to nonUNIX operating systems, in particular, protocols used on PC networks, such as Appletalk and IPX
- Internally, networking in the Linux kernel is implemented by three layers of software:
 - The socket interface
 - Protocol drivers
 - Network device drivers





Network Structure (Cont.)

- The most important set of protocols in the Linux networking system is the internet protocol suite
 - It implements routing between different hosts anywhere on the network
 - On top of the routing protocol are built the UDP, TCP and ICMP protocols





Security

- The *pluggable authentication modules (PAM)* system is available under Linux
- PAM is based on a shared library that can be used by any system component that needs to authenticate users
- Access control under UNIX systems, including Linux, is performed through the use of unique numeric identifiers (**uid** and **gid**)
- Access control is performed by assigning objects a *protection mask*, which specifies which access modes—read, write, or execute—are to be granted to processes with owner, group, or world access



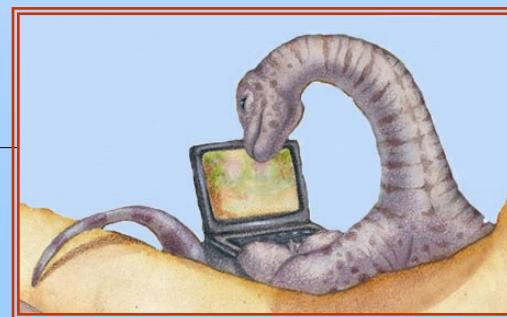


Security (Cont.)

- Linux augments the standard UNIX **setuid** mechanism in two ways:
 - It implements the POSIX specification's saved *user-id* mechanism, which allows a process to repeatedly drop and reacquire its effective uid
 - It has added a process characteristic that grants just a subset of the rights of the effective uid
- Linux provides another mechanism that allows a client to selectively pass access to a single file to some server process without granting it any other privileges



End of Chapter 21



Chapter 22: Windows XP





Module 22: Windows XP

- History
- Design Principles
- System Components
- Environmental Subsystems
- File system
- Networking
- Programmer Interface





Objectives

- To explore the principles upon which Windows XP is designed and the specific components involved in the system
- To understand how Windows XP can run programs designed for other operating systems
- To provide a detailed explanation of the Windows XP file system
- To illustrate the networking protocols supported in Windows XP
- To cover the interface available to system and application programmers





Windows XP

- 32-bit preemptive multitasking operating system for Intel microprocessors
- Key goals for the system:
 - portability
 - security
 - POSIX compliance
 - multiprocessor support
 - extensibility
 - international support
 - compatibility with MS-DOS and MS-Windows applications.
- Uses a micro-kernel architecture
- Available in four versions, Professional, Server, Advanced Server, National Server





History

- In 1988, Microsoft decided to develop a “new technology” (NT) portable operating system that supported both the OS/2 and POSIX APIs
- Originally, NT was supposed to use the OS/2 API as its native environment but during development NT was changed to use the Win32 API, reflecting the popularity of Windows 3.0





Design Principles

- Extensibility — layered architecture
 - Executive, which runs in protected mode, provides the basic system services
 - On top of the executive, several server subsystems operate in user mode
 - Modular structure allows additional environmental subsystems to be added without affecting the executive
- Portability —XP can be moved from one hardware architecture to another with relatively few changes
 - Written in C and C++
 - Processor-dependent code is isolated in a dynamic link library (DLL) called the “hardware abstraction layer” (HAL)





Design Principles (Cont.)

- Reliability —XP uses hardware protection for virtual memory, and software protection mechanisms for operating system resources
- Compatibility — applications that follow the IEEE 1003.1 (POSIX) standard can be complied to run on XP without changing the source code
- Performance —XP subsystems can communicate with one another via high-performance message passing
 - Preemption of low priority threads enables the system to respond quickly to external events
 - Designed for symmetrical multiprocessing
- International support — supports different locales via the national language support (NLS) API





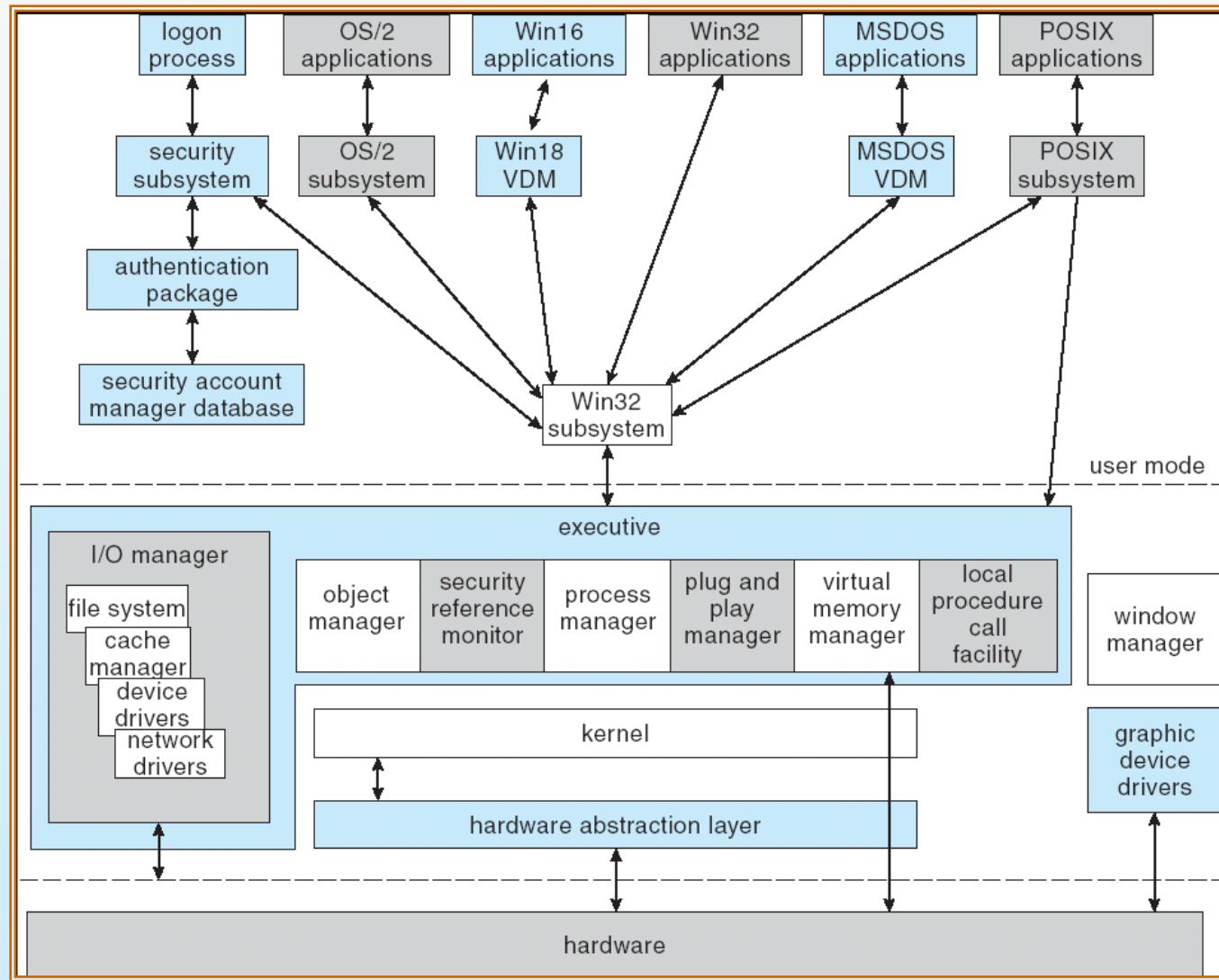
XP Architecture

- Layered system of modules
- Protected mode — HAL, kernel, executive
- User mode — collection of subsystems
 - Environmental subsystems emulate different operating systems
 - Protection subsystems provide security functions





Depiction of XP Architecture





System Components — Kernel

- Foundation for the executive and the subsystems
- Never paged out of memory; execution is never preempted
- Four main responsibilities:
 - thread scheduling
 - interrupt and exception handling
 - low-level processor synchronization
 - recovery after a power failure
- Kernel is object-oriented, uses two sets of objects
 - *dispatcher objects* control dispatching and synchronization (events, mutants, mutexes, semaphores, threads and timers)
 - *control objects* (asynchronous procedure calls, interrupts, power notify, power status, process and profile objects)





Kernel — Process and Threads

- The process has a virtual memory address space, information (such as a base priority), and an affinity for one or more processors
- Threads are the unit of execution scheduled by the kernel's dispatcher
- Each thread has its own state, including a priority, processor affinity, and accounting information
- A thread can be one of six states: *ready*, *standby*, *running*, *waiting*, *transition*, and *terminated*





Kernel — Scheduling

- The dispatcher uses a 32-level priority scheme to determine the order of thread execution
 - Priorities are divided into two classes
 - ▶ The real-time class contains threads with priorities ranging from 16 to 31
 - ▶ The variable class contains threads having priorities from 0 to 15
- Characteristics of XP's priority strategy
 - Trends to give very good response times to interactive threads that are using the mouse and windows
 - Enables I/O-bound threads to keep the I/O devices busy
 - Complete-bound threads soak up the spare CPU cycles in the background





Kernel — Scheduling (Cont.)

- Scheduling can occur when a thread enters the ready or wait state, when a thread terminates, or when an application changes a thread's priority or processor affinity
- Real-time threads are given preferential access to the CPU; but XP does not guarantee that a real-time thread will start to execute within any particular time limit
 - This is known as *soft realtime*





Windows XP Interrupt Request Levels

interrupt levels	types of interrupts
31	machine check or bus error
30	power fail
29	interprocessor notification (request another processor to act; e.g., dispatch a process or update the TLB)
28	clock (used to keep track of time)
27	profile
3–26	traditional PC IRQ hardware interrupts
2	dispatch and deferred procedure call (DPC) (kernel)
1	asynchronous procedure call (APC)
0	passive





Kernel — Trap Handling

- The kernel provides trap handling when exceptions and interrupts are generated by hardware or software
- Exceptions that cannot be handled by the trap handler are handled by the kernel's *exception dispatcher*
- The interrupt dispatcher in the kernel handles interrupts by calling either an interrupt service routine (such as in a device driver) or an internal kernel routine
- The kernel uses spin locks that reside in global memory to achieve multiprocessor mutual exclusion





Executive — Object Manager

- XP uses objects for all its services and entities; the object manager supervises the use of all the objects
 - Generates an object *handle*
 - Checks security
 - Keeps track of which processes are using each object
- Objects are manipulated by a standard set of methods, namely `create`, `open`, `close`, `delete`, `query name`, `parse` and `security`





Executive — Naming Objects

- The XP executive allows any object to be given a name, which may be either permanent or temporary
- Object names are structured like file path names in MS-DOS and UNIX
- XP implements a *symbolic link object*, which is similar to *symbolic links* in UNIX that allow multiple nicknames or aliases to refer to the same file
- A process gets an object handle by creating an object by opening an existing one, by receiving a duplicated handle from another process, or by inheriting a handle from a parent process
- Each object is protected by an access control list





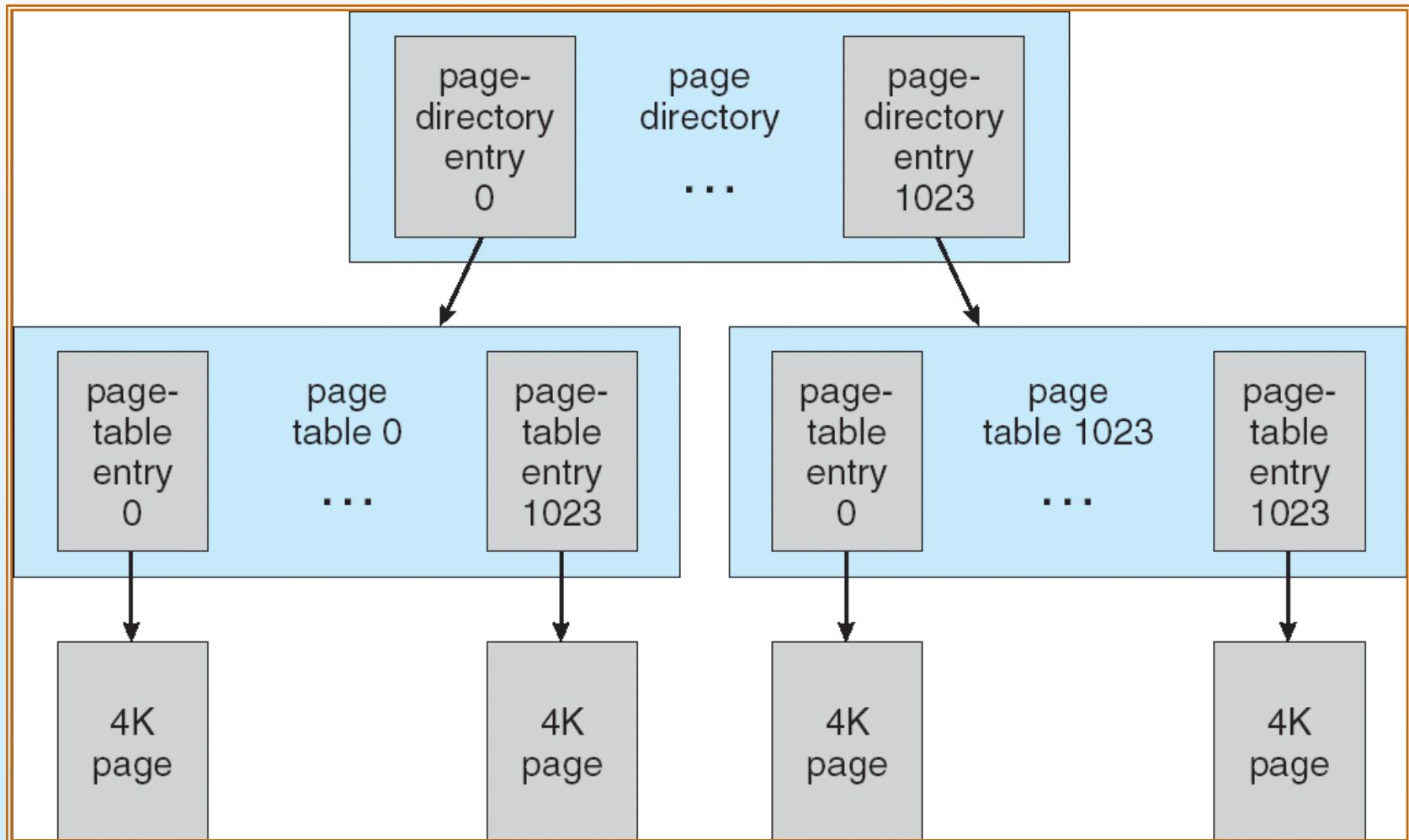
Executive — Virtual Memory Manager

- The design of the VM manager assumes that the underlying hardware supports virtual to physical mapping a paging mechanism, transparent cache coherence on multiprocessor systems, and virtual addressing aliasing
- The VM manager in XP uses a page-based management scheme with a page size of 4 KB
- The XP VM manager uses a two step process to allocate memory
 - The first step reserves a portion of the process's address space
 - The second step commits the allocation by assigning space in the 2000 paging file





Virtual-Memory Layout





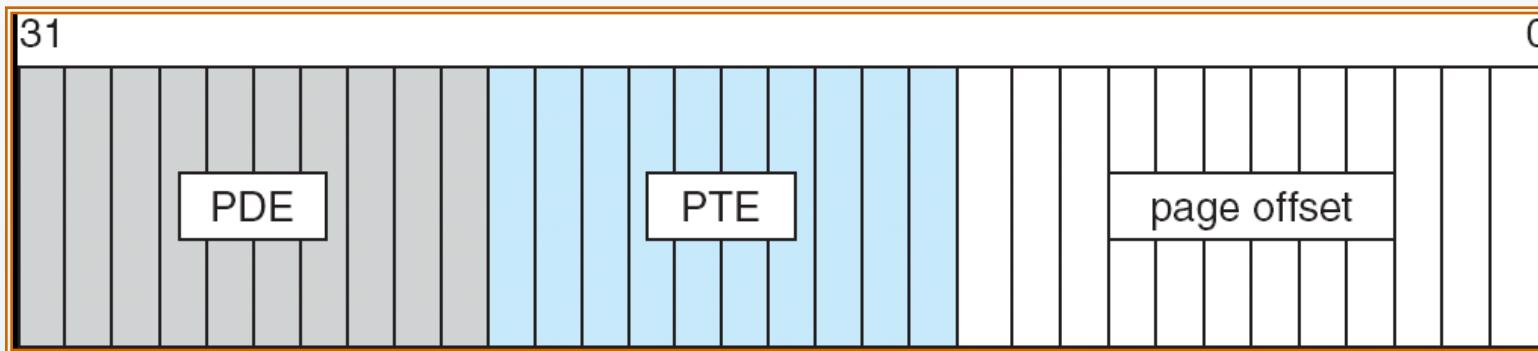
Virtual Memory Manager (Cont.)

- The virtual address translation in XP uses several data structures
 - Each process has a *page directory* that contains 1024 *page directory entries* of size 4 bytes
 - Each page directory entry points to a *page table* which contains 1024 *page table entries* (PTEs) of size 4 bytes
 - Each PTE points to a 4 KB *page frame* in physical memory
- A 10-bit integer can represent all the values from 0 to 1023, therefore, can select any entry in the page directory, or in a page table
- This property is used when translating a virtual address pointer to a byte address in physical memory
- A page can be in one of six states: valid, zeroed, free standby, modified and bad





Virtual-to-Physical Address Translation

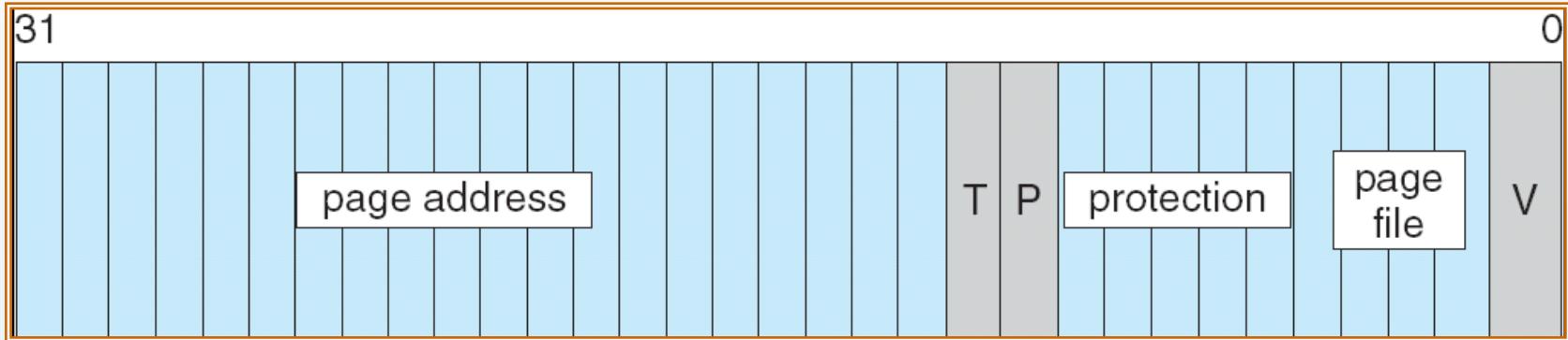


- 10 bits for page directory entry, 20 bits for page table entry, and 12 bits for byte offset in page





Page File Page-Table Entry



5 bits for page protection, 20 bits for page frame address, 4 bits to select a paging file, and 3 bits that describe the page state. $V = 0$





Executive — Process Manager

- Provides services for creating, deleting, and using threads and processes.
- Issues such as parent/child relationships or process hierarchies are left to the particular environmental subsystem that owns the process.





Executive — Local Procedure Call Facility

- The LPC passes requests and results between client and server processes within a single machine.
- In particular, it is used to request services from the various XP subsystems.
- When a LPC channel is created, one of three types of message passing techniques must be specified.
 - First type is suitable for small messages, up to 256 bytes; port's message queue is used as intermediate storage, and the messages are copied from one process to the other.
 - Second type avoids copying large messages by pointing to a shared memory section object created for the channel.
 - Third method, called *quick* LPC was used by graphical display portions of the Win32 subsystem.





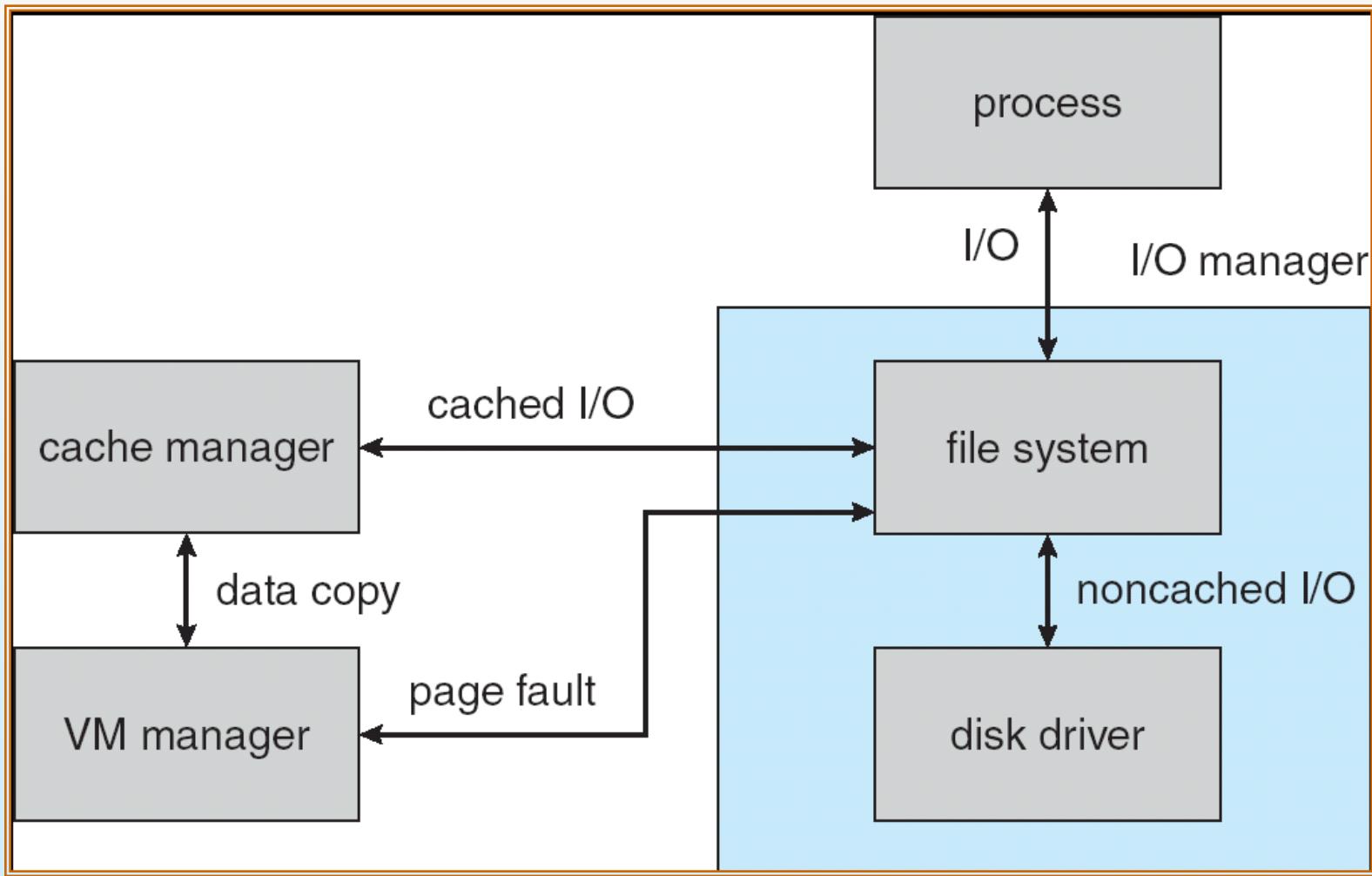
Executive — I/O Manager

- The I/O manager is responsible for
 - file systems
 - cache management
 - device drivers
 - network drivers
- Keeps track of which installable file systems are loaded, and manages buffers for I/O requests
- Works with VM Manager to provide memory-mapped file I/O
- Controls the XP cache manager, which handles caching for the entire I/O system
- Supports both synchronous and asynchronous operations, provides time outs for drivers, and has mechanisms for one driver to call another





File I/O





Executive — Security Reference Monitor

- The object-oriented nature of XP enables the use of a uniform mechanism to perform runtime access validation and audit checks for every entity in the system
- Whenever a process opens a handle to an object, the security reference monitor checks the process's security token and the object's access control list to see whether the process has the necessary rights





Executive – Plug-and-Play Manager

- Plug-and-Play (PnP) manager is used to recognize and adapt to changes in the hardware configuration
- When new devices are added (for example, PCI or USB), the PnP manager loads the appropriate driver
- The manager also keeps track of the resources used by each device





Environmental Subsystems

- User-mode processes layered over the native XP executive services to enable XP to run programs developed for other operating system
- XP uses the Win32 subsystem as the main operating environment; Win32 is used to start all processes
 - It also provides all the keyboard, mouse and graphical display capabilities
- MS-DOS environment is provided by a Win32 application called the *virtual dos machine* (VDM), a user-mode process that is paged and dispatched like any other XP thread





Environmental Subsystems (Cont.)

- 16-Bit Windows Environment:
 - Provided by a VDM that incorporates *Windows on Windows*
 - Provides the Windows 3.1 kernel routines and sub routines for window manager and GDI functions
- The POSIX subsystem is designed to run POSIX applications following the POSIX.1 standard which is based on the UNIX model





Environmental Subsystems (Cont.)

- OS/2 subsystems runs OS/2 applications
- Logon and Security Subsystems authenticates users logging on to Windows XP systems
 - Users are required to have account names and passwords
 - The authentication package authenticates users whenever they attempt to access an object in the system
 - Windows XP uses Kerberos as the default authentication package





File System

- The fundamental structure of the XP file system (NTFS) is a *volume*
 - Created by the XP disk administrator utility
 - Based on a logical disk partition
 - May occupy a portions of a disk, an entire disk, or span across several disks
- All *metadata*, such as information about the volume, is stored in a regular file
- NTFS uses *clusters* as the underlying unit of disk allocation
 - A cluster is a number of disk sectors that is a power of two
 - Because the cluster size is smaller than for the 16-bit FAT file system, the amount of internal fragmentation is reduced





File System — Internal Layout

- NTFS uses logical cluster numbers (LCNs) as disk addresses
- A file in NTFS is not a simple byte stream, as in MS-DOS or UNIX, rather, it is a structured object consisting of attributes
- Every file in NTFS is described by one or more records in an array stored in a special file called the Master File Table (MFT)
- Each file on an NTFS volume has a unique ID called a file reference.
 - 64-bit quantity that consists of a 48-bit file number and a 16-bit sequence number
 - Can be used to perform internal consistency checks
- The NTFS name space is organized by a hierarchy of directories; the index root contains the top level of the B+ tree





File System — Recovery

- All file system data structure updates are performed inside transactions that are logged
 - Before a data structure is altered, the transaction writes a log record that contains redo and undo information
 - After the data structure has been changed, a commit record is written to the log to signify that the transaction succeeded
 - After a crash, the file system data structures can be restored to a consistent state by processing the log records





File System — Recovery (Cont.)

- This scheme does not guarantee that all the user file data can be recovered after a crash, just that the file system data structures (the metadata files) are undamaged and reflect some consistent state prior to the crash
- The log is stored in the third metadata file at the beginning of the volume
- The logging functionality is provided by the XP *log file service*





File System — Security

- Security of an NTFS volume is derived from the XP object model
- Each file object has a security descriptor attribute stored in this MFT record
- This attribute contains the access token of the owner of the file, and an access control list that states the access privileges that are granted to each user that has access to the file





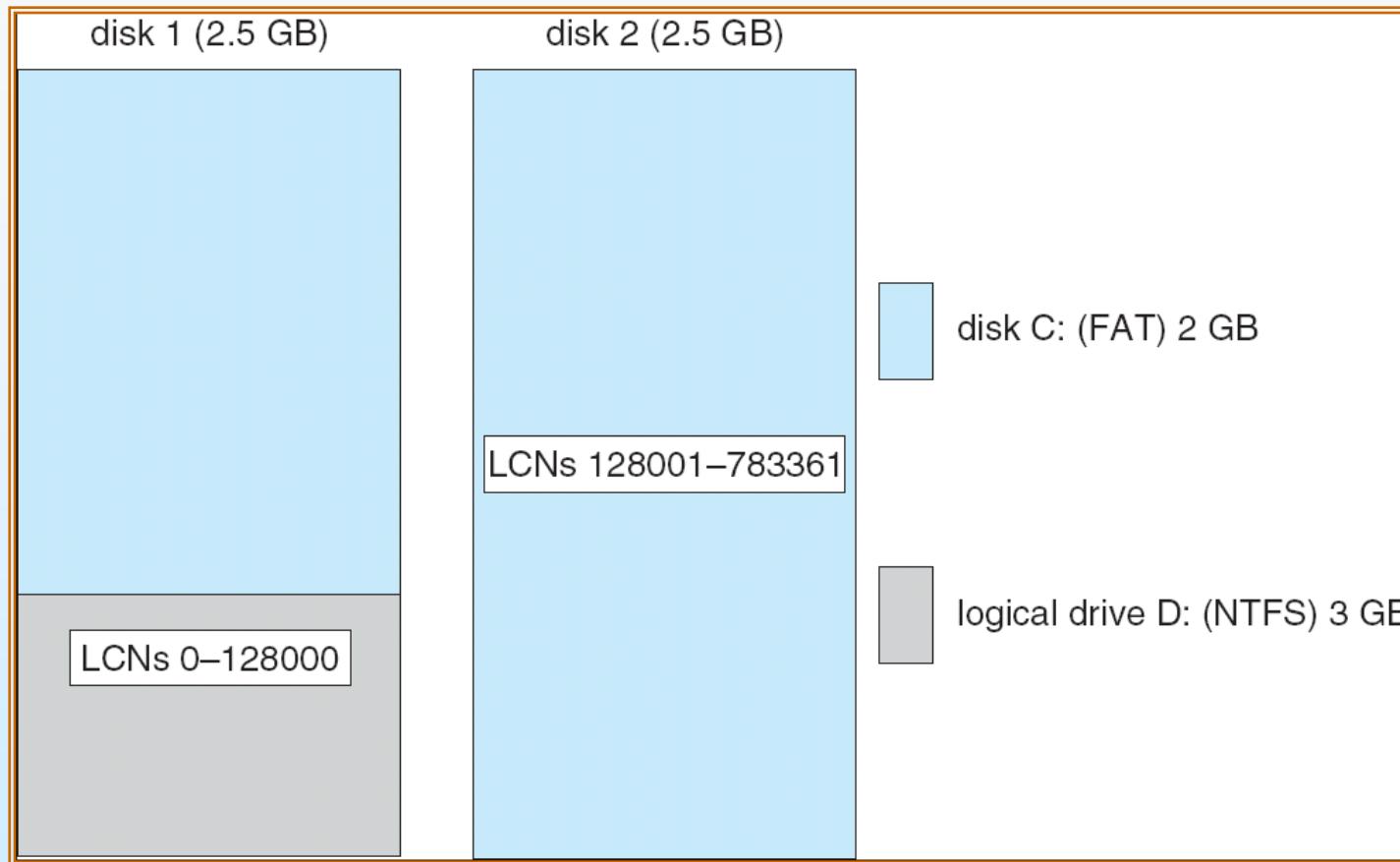
Volume Management and Fault Tolerance

- FtDisk, the fault tolerant disk driver for XP, provides several ways to combine multiple SCSI disk drives into one logical volume
- Logically concatenate multiple disks to form a large logical volume, a *volume set*
- Interleave multiple physical partitions in round-robin fashion to form a *stripe set* (also called RAID level 0, or “disk striping”)
 - Variation: *stripe set with parity*, or RAID level 5
- Disk mirroring, or RAID level 1, is a robust scheme that uses a *mirror set* — two equally sized partitions on tow disks with identical data contents
- To deal with disk sectors that go bad, FtDisk, uses a hardware technique called *sector sparing* and NTFS uses a software technique called *cluster remapping*



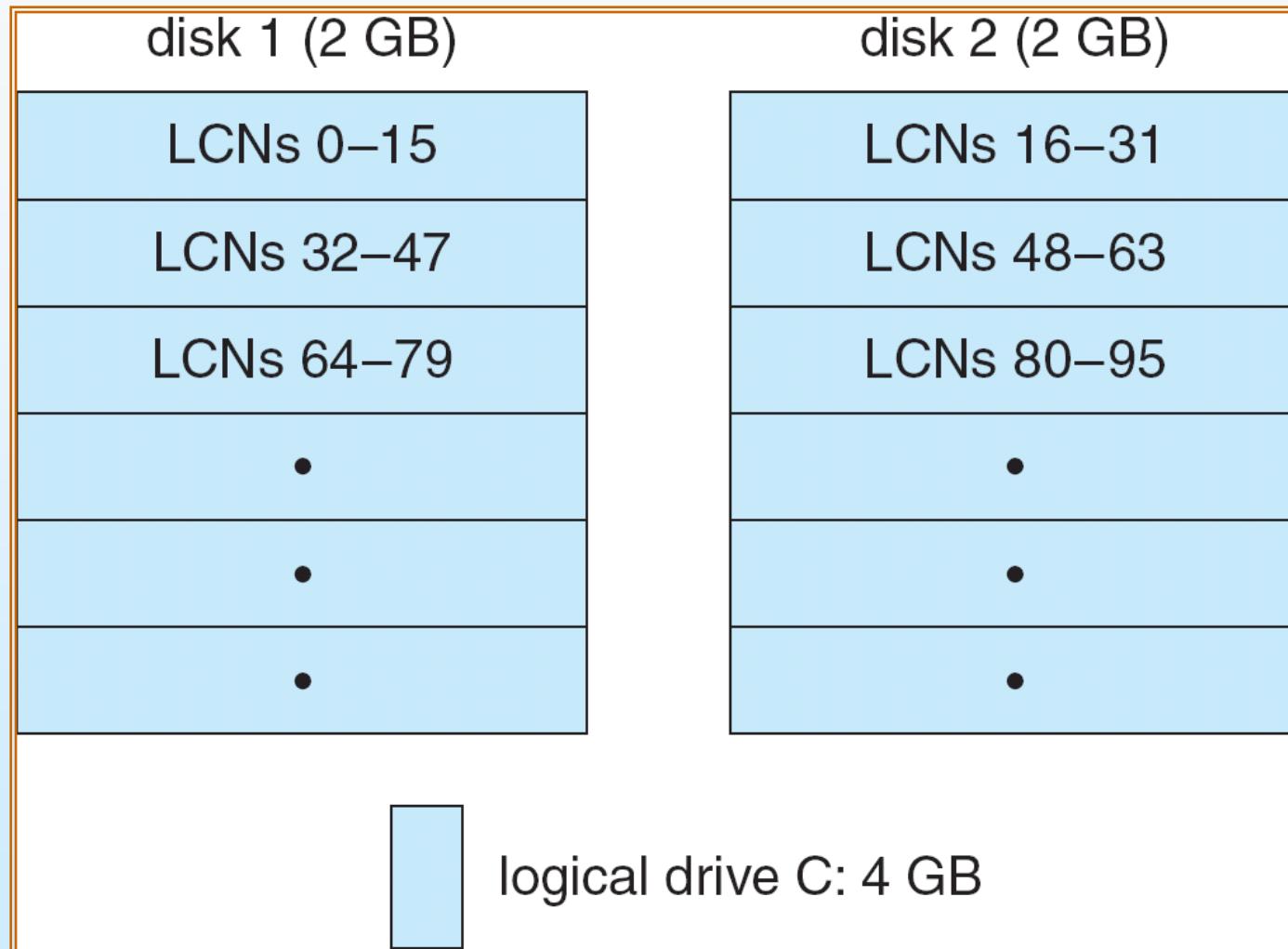


Volume Set On Two Drives



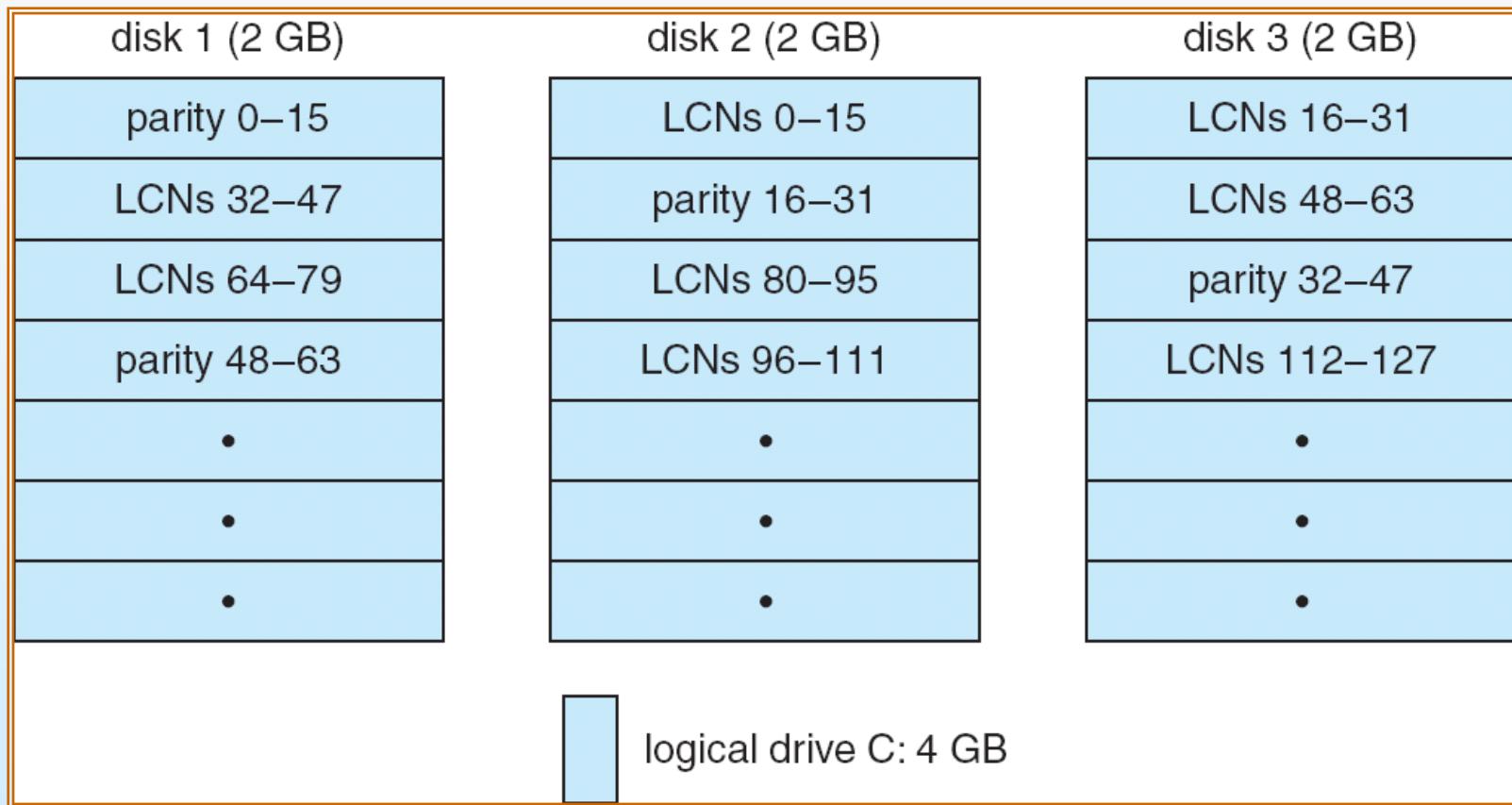


Stripe Set on Two Drives



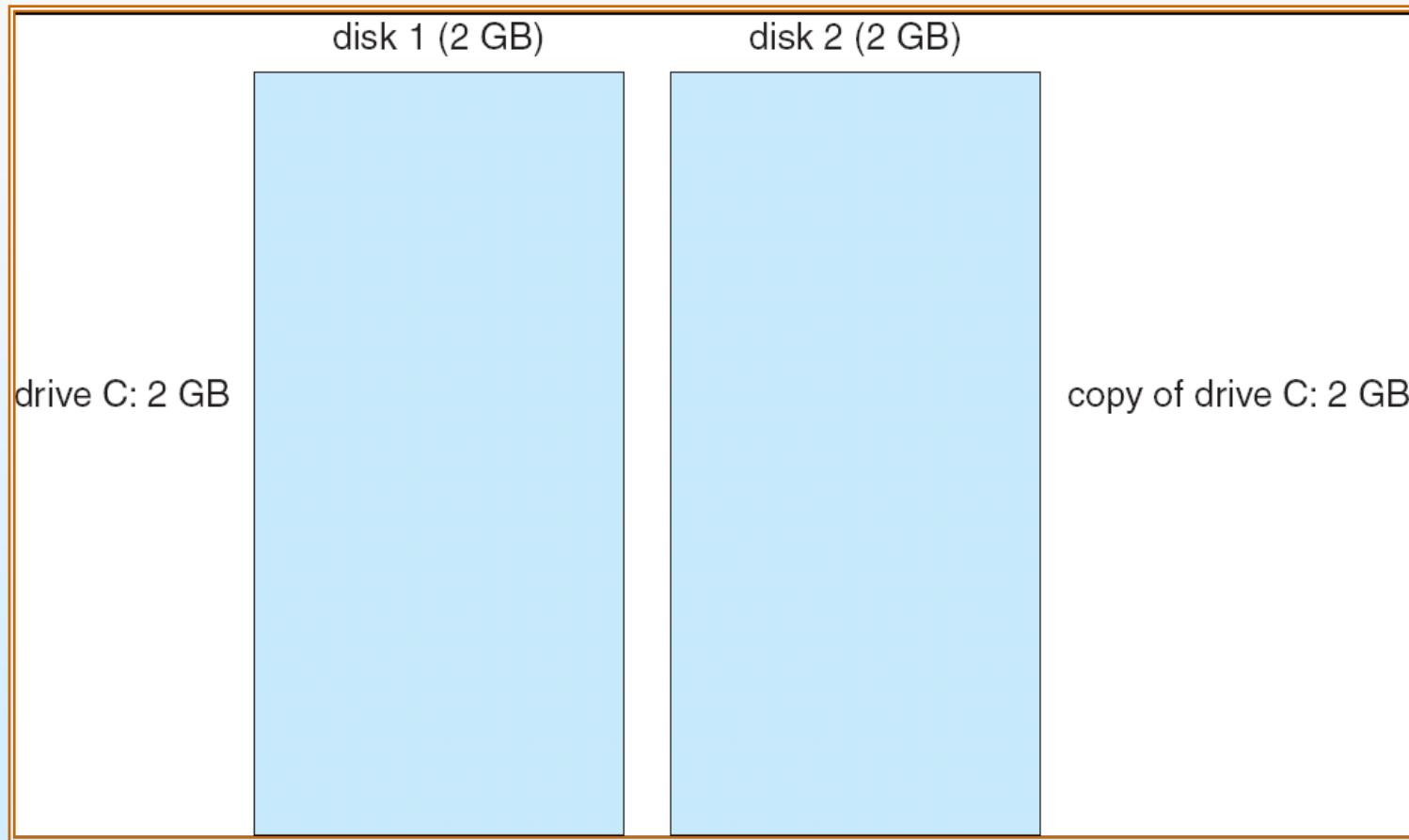


Stripe Set With Parity on Three Drives





Mirror Set on Two Drives





File System — Compression

- To compress a file, NTFS divides the file's data into *compression units*, which are blocks of 16 contiguous clusters
- For sparse files, NTFS uses another technique to save space
 - Clusters that contain all zeros are not actually allocated or stored on disk
 - Instead, gaps are left in the sequence of virtual cluster numbers stored in the MFT entry for the file
 - When reading a file, if a gap in the virtual cluster numbers is found, NTFS just zero-fills that portion of the caller's buffer





File System — Reparse Points

- A reparse point returns an error code when accessed. The reparse data tells the I/O manager what to do next
- Reparse points can be used to provide the functionality of UNIX *mounts*
- Reparse points can also be used to access files that have been moved to offline storage





Networking

- XP supports both peer-to-peer and client/server networking; it also has facilities for network management
- To describe networking in XP, we refer to two of the internal networking interfaces:
 - NDIS (Network Device Interface Specification) — Separates network adapters from the transport protocols so that either can be changed without affecting the other
 - TDI (Transport Driver Interface) — Enables any session layer component to use any available transport mechanism
- XP implements transport protocols as drivers that can be loaded and unloaded from the system dynamically





Networking — Protocols

- The server message block (SMB) protocol is used to send I/O requests over the network. It has four message types:
 - Session control
 - File
 - Printer
 - Message
- The network basic Input/Output system (NetBIOS) is a hardware abstraction interface for networks
 - Used to:
 - ▶ Establish logical names on the network
 - ▶ Establish logical connections of sessions between two logical names on the network
 - ▶ Support reliable data transfer for a session via NetBIOS requests or *SMBs*





Networking — Protocols (Cont.)

- NetBEUI (NetBIOS Extended User Interface): default protocol for Windows 95 peer networking and Windows for Workgroups; used when XP wants to share resources with these networks
- XP uses the TCP/IP Internet protocol to connect to a wide variety of operating systems and hardware platforms
- PPTP (Point-to-Point Tunneling Protocol) is used to communicate between Remote Access Server modules running on XP machines that are connected over the Internet
- The XP NWLink protocol connects the NetBIOS to Novell NetWare networks





Networking — Protocols (Cont.)

- The Data Link Control protocol (DLC) is used to access IBM mainframes and HP printers that are directly connected to the network
- XP systems can communicate with Macintosh computers via the Apple Talk protocol if an XP Server on the network is running the Windows XP Services for Macintosh package





Networking — Dist. Processing Mechanisms

- XP supports distributed applications via named NetBIOS, named pipes and mailslots, Windows Sockets, Remote Procedure Calls (RPC), and Network Dynamic Data Exchange (NetDDE)
- NetBIOS applications can communicate over the network using NetBEUI, NWLink, or TCP/IP
- Named pipes are connection-oriented messaging mechanism that are named via the uniform naming convention (UNC)
- Mailslots are a connectionless messaging mechanism that are used for broadcast applications, such as for finding components on the network
- Winsock, the windows sockets API, is a session-layer interface that provides a standardized interface to many transport protocols that may have different addressing schemes





Distributed Processing Mechanisms (Cont.)

- The XP RPC mechanism follows the widely-used Distributed Computing Environment standard for RPC messages, so programs written to use XP RPCs are very portable
 - RPC messages are sent using NetBIOS, or Winsock on TCP/IP networks, or named pipes on LAN Manager networks
 - XP provides the Microsoft *Interface Definition Language* to describe the remote procedure names, arguments, and results





Networking — Redirectors and Servers

- In XP , an application can use the XP I/O API to access files from a remote computer as if they were local, provided that the remote computer is running an MS-NET server
- A *redirector* is the client-side object that forwards I/O requests to remote files, where they are satisfied by a server
- For performance and security, the redirectors and servers run in kernel mode





Access to a Remote File

- The application calls the I/O manager to request that a file be opened (we assume that the file name is in the standard UNC format)
- The I/O manager builds an I/O request packet
- The I/O manager recognizes that the access is for a remote file, and calls a driver called a Multiple Universal Naming Convention Provider (MUP)
- The MUP sends the I/O request packet asynchronously to all registered redirectors
- A redirector that can satisfy the request responds to the MUP
 - To avoid asking all the redirectors the same question in the future, the MUP uses a cache to remember which redirector can handle this file





Access to a Remote File (Cont.)

- The redirector sends the network request to the remote system
- The remote system network drivers receive the request and pass it to the server driver
- The server driver hands the request to the proper local file system driver
- The proper device driver is called to access the data
- The results are returned to the server driver, which sends the data back to the requesting redirector





Networking — Domains

- NT uses the concept of a domain to manage global access rights within groups
- A domain is a group of machines running NT server that share a common security policy and user database
- XP provides three models of setting up trust relationships
 - *One way, A trusts B*
 - *Two way, transitive, A trusts B, B trusts C so A, B, C trust each other*
 - *Crosslink – allows authentication to bypass hierarchy to cut down on authentication traffic*





Name Resolution in TCP/IP Networks

- On an IP network, name resolution is the process of converting a computer name to an IP address

e.g., `www.bell-labs.com` resolves to `135.104.1.14`

- XP provides several methods of name resolution:
 - Windows Internet Name Service (WINS)
 - broadcast name resolution
 - domain name system (DNS)
 - a host file
 - an LMHOSTS file





Name Resolution (Cont.)

- WINS consists of two or more WINS servers that maintain a dynamic database of name to IP address bindings, and client software to query the servers
- WINS uses the Dynamic Host Configuration Protocol (DHCP), which automatically updates address configurations in the WINS database, without user or administrator intervention





Programmer Interface — Access to Kernel Obj.

- A process gains access to a kernel object named `xxx` by calling the `CreateXXX` function to open a *handle* to `xxx`; the handle is unique to that process
- A handle can be closed by calling the `CloseHandle` function; the system may delete the object if the count of processes using the object drops to 0
- XP provides three ways to share objects between processes
 - A child process inherits a handle to the object
 - One process gives the object a name when it is created and the second process opens that name
 - `DuplicateHandle` function:
 - ▶ Given a handle to process and the handle's value a second process can get a handle to the same object, and thus share it





Programmer Interface — Process Management

- Process is started via the `CreateProcess` routine which loads any dynamic link libraries that are used by the process, and creates a *primary thread*
- Additional threads can be created by the `CreateThread` function
- Every dynamic link library or executable file that is loaded into the address space of a process is identified by an *instance handle*





Process Management (Cont.)

- Scheduling in Win32 utilizes four priority classes:
 - IDLE_PRIORITY_CLASS (priority level 4)
 - NORMAL_PRIORITY_CLASS (level 8 — typical for most processes)
 - HIGH_PRIORITY_CLASS (level 13)
 - REALTIME_PRIORITY_CLASS (level 24)
- To provide performance levels needed for interactive programs, XP has a special scheduling rule for processes in the NORMAL_PRIORITY_CLASS
 - XP distinguishes between the *foreground process* that is currently selected on the screen, and the *background processes* that are not currently selected
 - When a process moves into the foreground, XP increases the scheduling quantum by some factor, typically 3





Process Management (Cont.)

- The kernel dynamically adjusts the priority of a thread depending on whether it is I/O-bound or CPU-bound
- To synchronize the concurrent access to shared objects by threads, the kernel provides synchronization objects, such as semaphores and mutexes
 - In addition, threads can synchronize by using the `WaitForSingleObject` or `WaitForMultipleObjects` functions
 - Another method of synchronization in the Win32 API is the critical section





Process Management (Cont.)

- A fiber is user-mode code that gets scheduled according to a user-defined scheduling algorithm
 - Only one fiber at a time is permitted to execute, even on multiprocessor hardware
 - XP includes fibers to facilitate the porting of legacy UNIX applications that are written for a fiber execution model





Programmer Interface — Interprocess Comm.

- Win32 applications can have interprocess communication by sharing kernel objects
- An alternate means of interprocess communications is message passing, which is particularly popular for Windows GUI applications
 - One thread sends a message to another thread or to a window
 - A thread can also send data with the message
- Every Win32 thread has its own input queue from which the thread receives messages
- This is more reliable than the shared input queue of 16-bit windows, because with separate queues, one stuck application cannot block input to the other applications





Programmer Interface — Memory Management

- Virtual memory:
 - `VirtualAlloc` reserves or commits virtual memory
 - `VirtualFree` decommits or releases the memory
 - These functions enable the application to determine the virtual address at which the memory is allocated
- An application can use memory by memory mapping a file into its address space
 - Multistage process
 - Two processes share memory by mapping the same file into their virtual memory





Memory Management (Cont.)

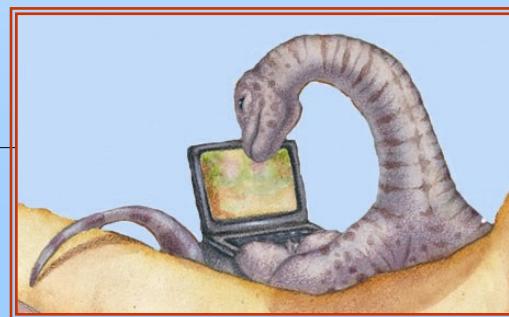
- A heap in the Win32 environment is a region of reserved address space
 - A Win 32 process is created with a 1 MB *default heap*
 - Access is synchronized to protect the heap's space allocation data structures from damage by concurrent updates by multiple threads
- Because functions that rely on global or static data typically fail to work properly in a multithreaded environment, the thread-local storage mechanism allocates global storage on a per-thread basis
 - The mechanism provides both dynamic and static methods of creating thread-local storage



End of Chapter 22



Module A: FreeBSD System





Module A: The FreeBSD System

- History
- Design Principles
- Programmer Interface
- User Interface
- Process Management
- Memory Management
- File System
- I/O System
- Interprocess Communication





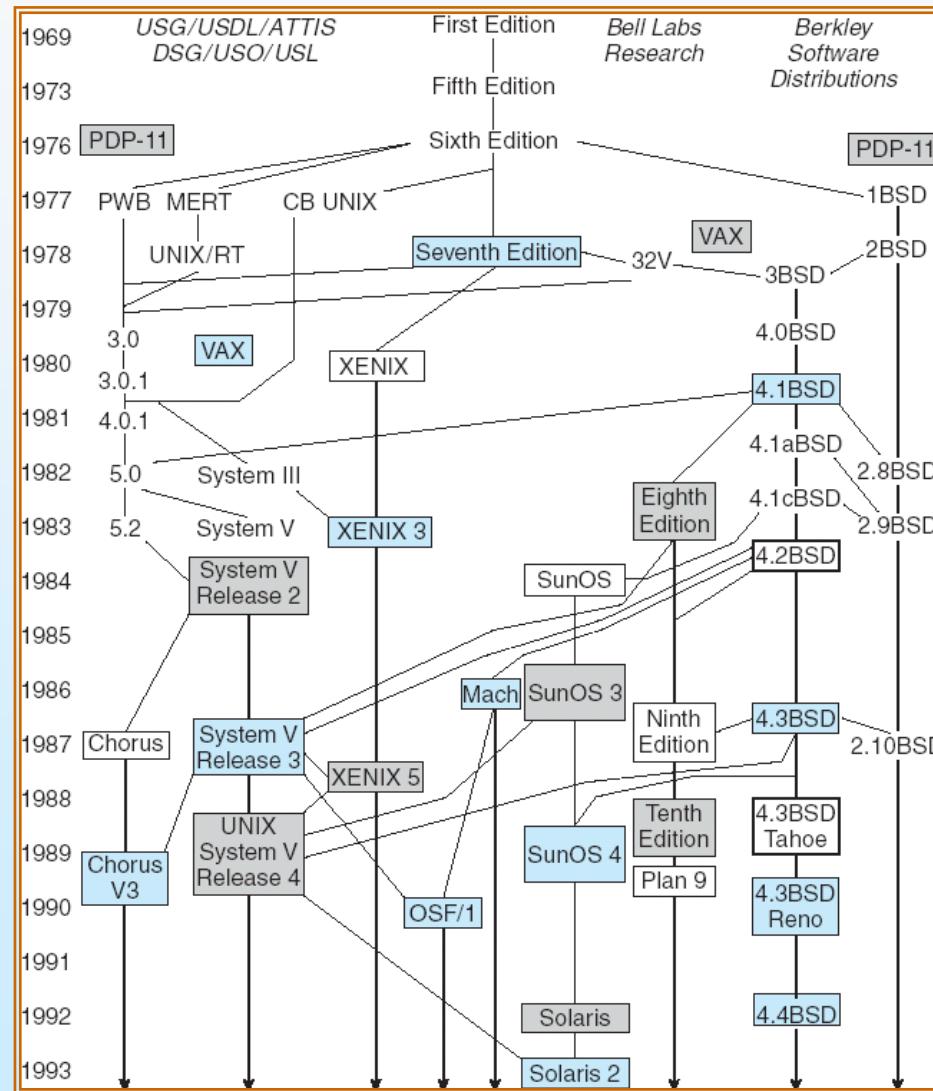
History

- First developed in 1969 by Ken Thompson and Dennis Ritchie of the Research Group at Bell Laboratories; incorporated features of other operating systems, especially MULTICS.
- The third version was written in C, which was developed at Bell Labs specifically to support UNIX.
- The most influential of the non-Bell Labs and non-AT&T UNIX development groups — University of California at Berkeley (Berkeley Software Distributions).
 - 4BSD UNIX resulted from DARPA funding to develop a standard UNIX system for government use.
 - Developed for the VAX, 4.3BSD is one of the most influential versions, and has been ported to many other platforms.
- Several standardization projects seek to consolidate the variant flavors of UNIX leading to one programming interface to UNIX.





History of UNIX Versions





Early Advantages of UNIX

- Written in a high-level language.
- Distributed in source form.
- Provided powerful operating-system primitives on an inexpensive platform.
- Small size, modular, clean design.





UNIX Design Principles

- Designed to be a time-sharing system.
- Has a simple standard user interface (shell) that can be replaced.
- File system with multilevel tree-structured directories.
- Files are supported by the kernel as unstructured sequences of bytes.
- Supports multiple processes; a process can easily create new processes.
- High priority given to making system interactive, and providing facilities for program development.





Programmer Interface

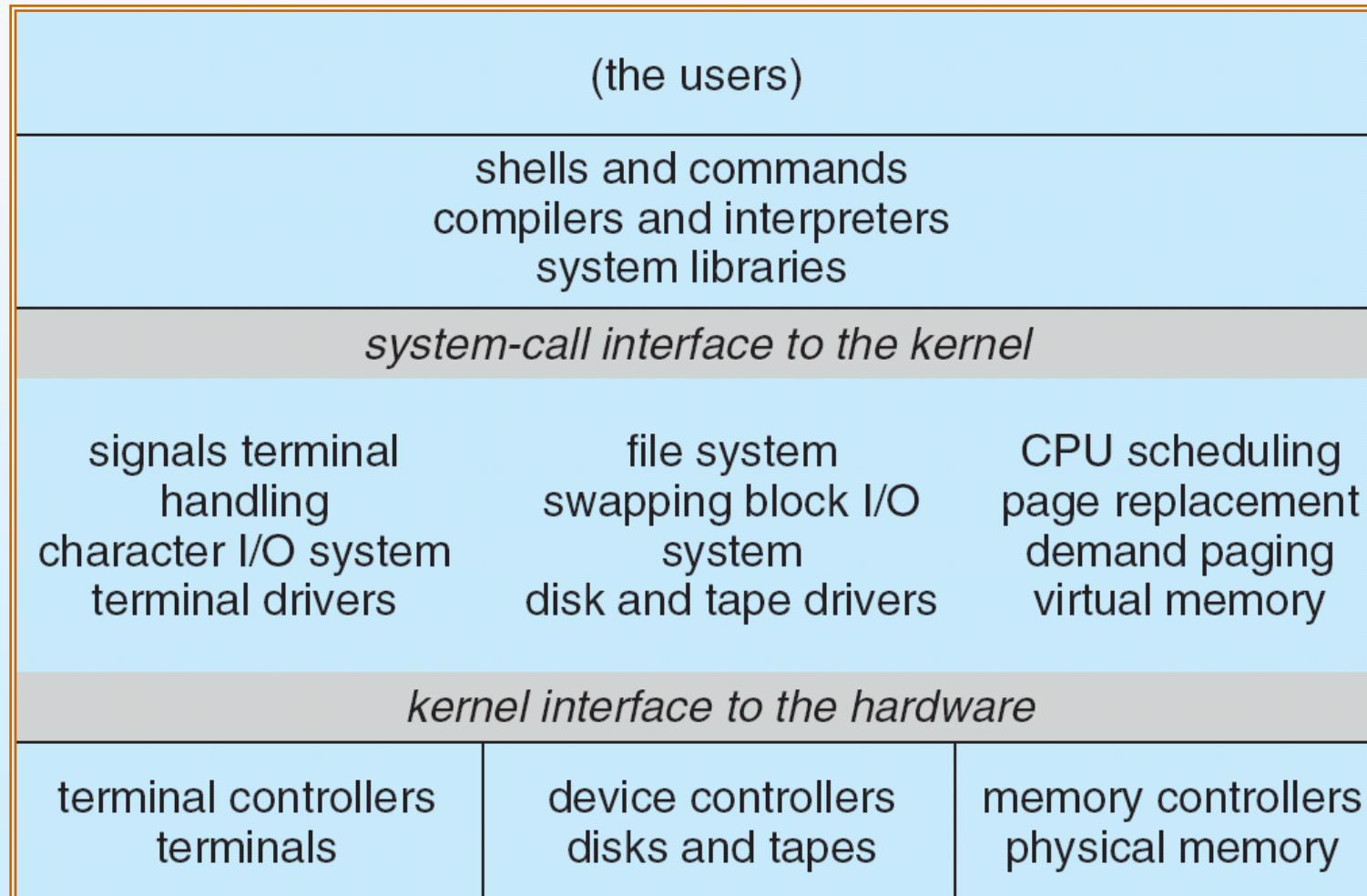
Like most computer systems, UNIX consists of two separable parts:

- Kernel: everything below the system-call interface and above the physical hardware.
 - Provides file system, CPU scheduling, memory management, and other OS functions through system calls.
- Systems programs: use the kernel-supported system calls to provide useful functions, such as compilation and file manipulation.





4.4BSD Layer Structure





System Calls

- System calls define the programmer interface to UNIX
- The set of systems programs commonly available defines the user interface.
- The programmer and user interface define the context that the kernel must support.
- Roughly three categories of system calls in UNIX.
 - File manipulation (same system calls also support device manipulation)
 - Process control
 - Information manipulation.





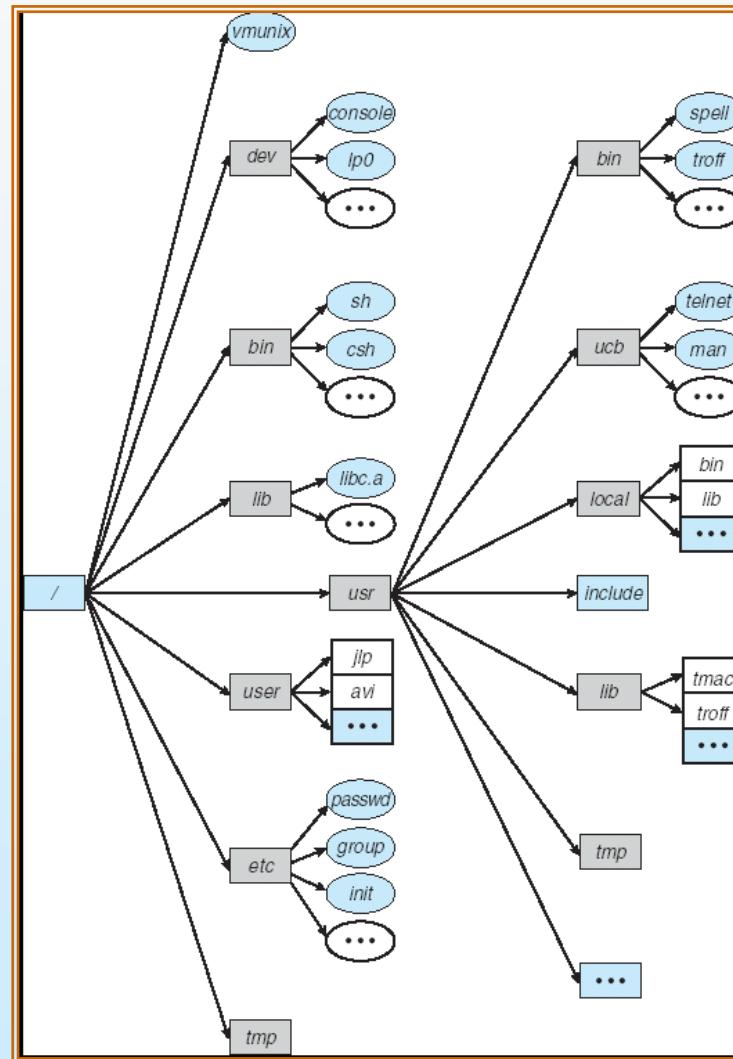
File Manipulation

- A *file* is a sequence of bytes; the kernel does not impose a structure on files.
- Files are organized in tree-structured *directories*.
- Directories are files that contain information on how to find other files.
- *Path name*: identifies a file by specifying a path through the directory structure to the file.
 - Absolute path names start at root of file system
 - Relative path names start at the current directory
- System calls for basic file manipulation: **create, open, read, write, close, unlink, trunc.**





Typical UNIX Directory Structure





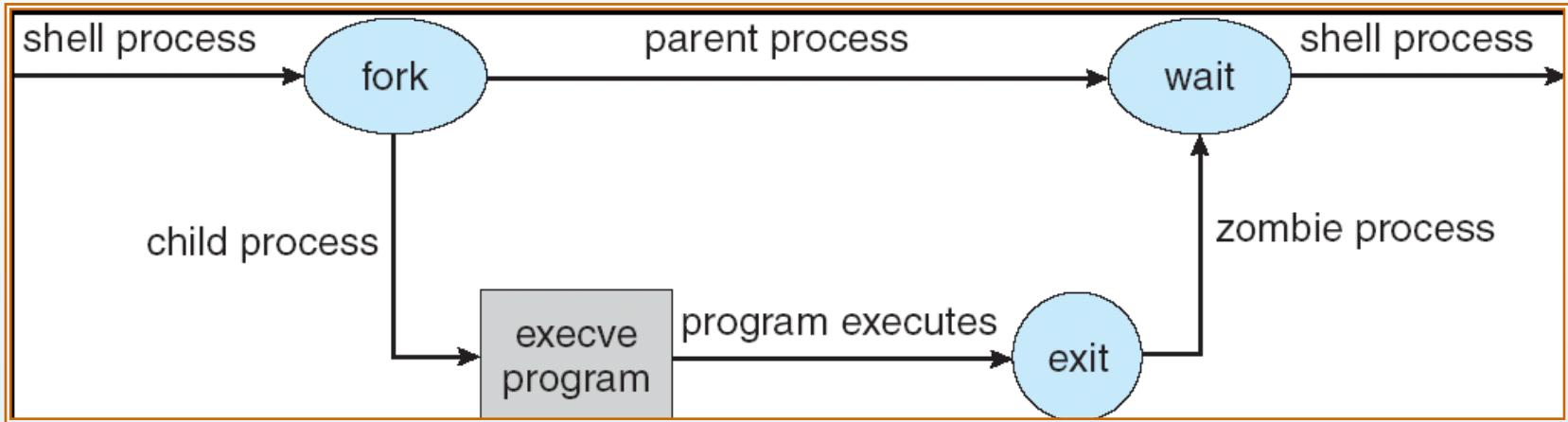
Process Control

- A process is a program in execution.
- Processes are identified by their process identifier, an integer.
- Process control system calls
 - **fork** creates a new process
 - **execve** is used after a fork to replace one of the two processes's virtual memory space with a new program
 - **exit** terminates a process
 - A parent may **wait** for a child process to terminate; **wait** provides the process id of a terminated child so that the parent can tell which child terminated.
 - **wait3** allows the parent to collect performance statistics about the child
- A *zombie* process results when the parent of a *defunct* child process exits before the terminated child.





Illustration of Process Control Calls





Process Control (Cont.)

- Processes communicate via pipes; queues of bytes between two processes that are accessed by a file descriptor.
- All user processes are descendants of one original process, *init*.
- *init* forks a *getty* process: initializes terminal line parameters and passes the user's *login name* to *login*.
 - *login* sets the numeric *user identifier* of the process to that of the user
 - executes a *shell* which forks subprocesses for user commands.





Process Control (Cont.)

- **setuid** bit sets the effective user identifier of the process to the user identifier of the owner of the file, and leaves the *real user identifier* as it was.
- **setuid** scheme allows certain processes to have more than ordinary privileges while still being executable by ordinary users.





Signals

- Facility for handling exceptional conditions similar to software interrupts.
- The *interrupt* signal, SIGINT, is used to stop a command before that command completes (usually produced by ^C).
- Signal use has expanded beyond dealing with exceptional events.
 - Start and stop subprocesses on demand
 - SIGWINCH informs a process that the window in which output is being displayed has changed size.
 - Deliver urgent data from network connections.





Process Groups

- Set of related processes that cooperate to accomplish a common task.
- Only one process group may use a terminal device for I/O at any time.
 - The foreground job has the attention of the user on the terminal.
 - Background jobs – nonattached jobs that perform their function without user interaction.
- Access to the terminal is controlled by process group signals.





Process Groups (Cont.)

- Each job inherits a controlling terminal from its parent.
 - If the process group of the controlling terminal matches the group of a process, that process is in the foreground.
 - SIGTTIN or SIGTTOU freezes a background process that attempts to perform I/O; if the user foregrounds that process, SIGCONT indicates that the process can now perform I/O.
 - SIGSTOP freezes a foreground process.





Information Manipulation

- System calls to set and return an interval timer:
getitimer/setitimer.
- Calls to set and return the current time:
gettimeofday/settimeofday.
- Processes can ask for
 - their process identifier: **getpid**
 - their group identifier: **getgid**
 - the name of the machine on which they are executing:
gethostname





Library Routines

- The system-call interface to UNIX is supported and augmented by a large collection of library routines
- Header files provide the definition of complex data structures used in system calls.
- Additional library support is provided for mathematical functions, network access, data conversion, etc.





User Interface

- Programmers and users mainly deal with already existing systems programs: the needed system calls are embedded within the program and do not need to be obvious to the user.
- The most common systems programs are file or directory oriented.
 - Directory: *mkdir*, *rmdir*, *cd*, *pwd*
 - File: *ls*, *cp*, *mv*, *rm*
- Other programs relate to editors (e.g., *emacs*, *vi*) text formatters (e.g., *troff*, *TEX*), and other activities.





Shells and Commands

- *Shell* – the user process which executes programs (also called command interpreter).
- Called a shell, because it surrounds the kernel.
- The shell indicates its readiness to accept another command by typing a prompt, and the user types a command on a single line.
- A typical command is an executable binary object file.
- The shell travels through the *search path* to find the command file, which is then loaded and executed.
- The directories /bin and /usr/bin are almost always in the search path.





Shells and Commands (Cont.)

- Typical search path on a BSD system:

(./home/prof/avi/bin /usr/local/bin /usr/ucb/bin/usr/bin)

- The shell usually suspends its own execution until the command completes.





Standard I/O

- Most processes expect three file descriptors to be open when they start:
 - *standard input* – program can read what the user types
 - *standard output* – program can send output to user's screen
 - *standard error* – error output
- Most programs can also accept a file (rather than a terminal) for standard input and standard output.
- The common shells have a simple syntax for changing what files are open for the standard I/O streams of a process — I/O *redirection*.





Standard I/O Redirection

command	meaning of command
% <i>ls</i> > <i>filea</i>	direct output of <i>ls</i> to file <i>filea</i>
% <i>pr</i> < <i>filea</i> > <i>fileb</i>	input from <i>filea</i> and output to <i>fileb</i>
% <i>lpr</i> < <i>fileb</i>	input from <i>fileb</i>
% % make program > & errs	save both standard output and standard error in a file





Pipelines, Filters, and Shell Scripts

- Can coalesce individual commands via a vertical bar that tells the shell to pass the previous command's output as input to the following command

```
% ls | pr | lpr
```

- Filter – a command such as pr that passes its standard input to its standard output, performing some processing on it.
- Writing a new shell with a different syntax and semantics would change the user view, but not change the kernel or programmer interface.
- X Window System is a widely accepted iconic interface for UNIX.





Process Management

- Representation of processes is a major design problem for operating system.
- UNIX is distinct from other systems in that multiple processes can be created and manipulated with ease.
- These processes are represented in UNIX by various control blocks.
 - Control blocks associated with a process are stored in the kernel.
 - Information in these control blocks is used by the kernel for process control and CPU scheduling.





Process Control Blocks

- The most basic data structure associated with processes is the *process structure*.
 - unique process identifier
 - scheduling information (e.g., priority)
 - pointers to other control blocks
- The *virtual address space* of a user process is divided into text (program code), data, and stack segments.
- Every process with sharable text has a pointer from its process structure to a *text structure*.
 - always resident in main memory.
 - records how many processes are using the text segment
 - records where the page table for the text segment can be found on disk when it is swapped.





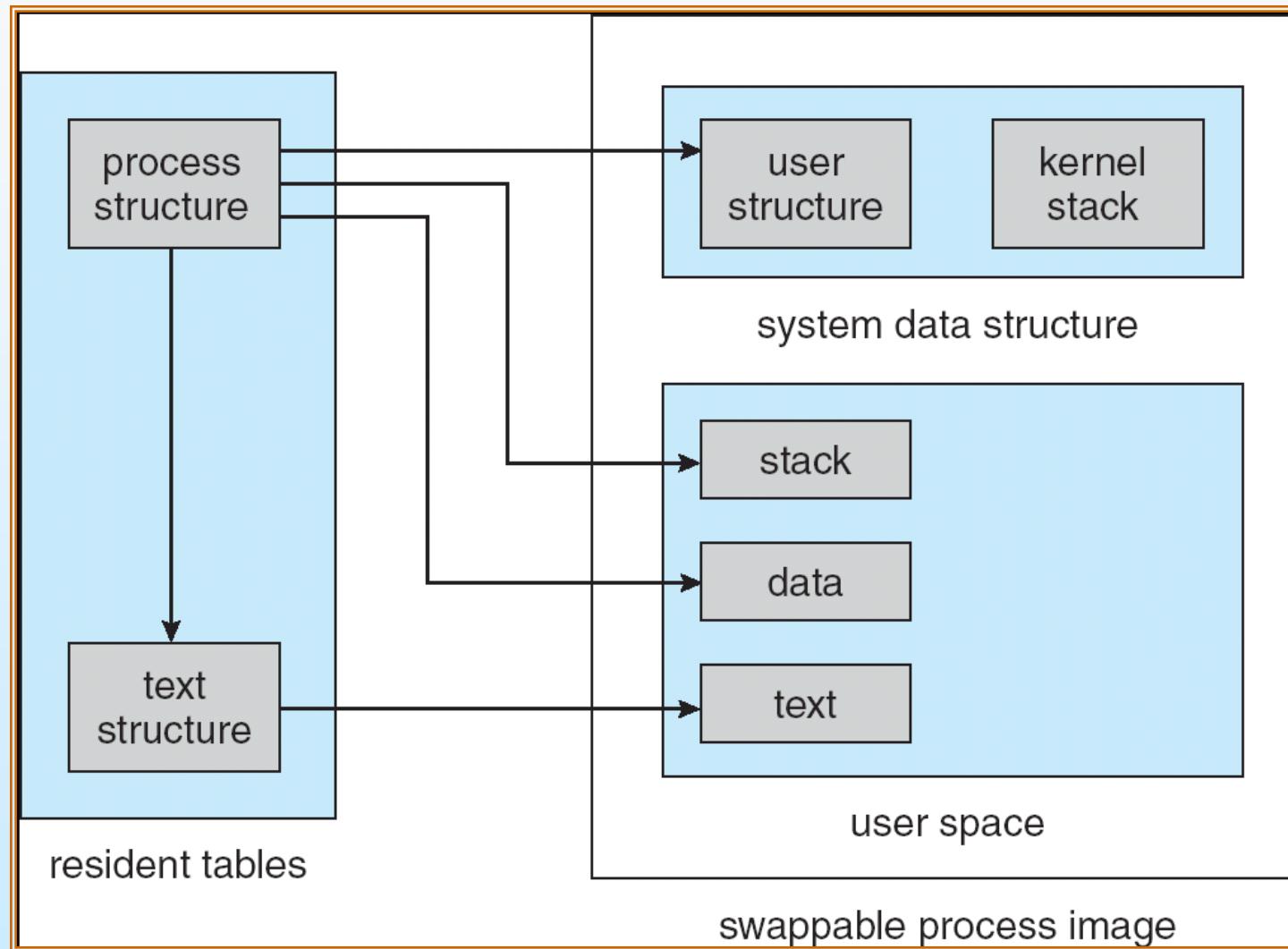
System Data Segment

- Most ordinary work is done in *user mode*; system calls are performed in *system mode*.
- The system and user phases of a process never execute simultaneously.
- a *kernel stack* (rather than the user stack) is used for a process executing in system mode.
- The kernel stack and the user structure together compose the *system data segment* for the process.





Finding parts of a process using process structure





Allocating a New Process Structure

- fork allocates a new process structure for the child process, and copies the user structure.
 - new page table is constructed
 - new main memory is allocated for the data and stack segments of the child process
 - copying the user structure preserves open file descriptors, user and group identifiers, signal handling, etc.





Allocating a New Process Structure (Cont.)

- **vfork** does *not* copy the data and stack to the new process; the new process simply shares the page table of the old one.
 - new user structure and a new process structure are still created
 - commonly used by a shell to execute a command and to wait for its completion
- A parent process uses **vfork** to produce a child process; the child uses **execve** to change its virtual address space, so there is no need for a copy of the parent.
- Using **vfork** with a large parent process saves CPU time, but can be dangerous since any memory change occurs in both processes until **execve** occurs.
- **execve** creates no new process or user structure; rather the text and data of the process are replaced.





CPU Scheduling

- Every process has a *scheduling priority* associated with it; larger numbers indicate lower priority.
- Negative feedback in CPU scheduling makes it difficult for a single process to take all the CPU time.
- Process aging is employed to prevent starvation.
- When a process chooses to relinquish the CPU, it goes to *sleep* on an *event*.
- When that event occurs, the system process that knows about it calls *wakeup* with the address corresponding to the event, and *all* processes that had done a *sleep* on the same address are put in the ready queue to be run.





Memory Management

- The initial memory management schemes were constrained in size by the relatively small memory resources of the PDP machines on which UNIX was developed.
- Pre 3BSD system use swapping exclusively to handle memory contention among processes: If there is too much contention, processes are swapped out until enough memory is available.
- Allocation of both main memory and swap space is done first-fit.





Memory Management (Cont.)

- Sharable text segments do not need to be swapped; results in less swap traffic and reduces the amount of main memory required for multiple processes using the same text segment.
- The *scheduler process* (or *swapper*) decides which processes to swap in or out, considering such factors as time idle, time in or out of main memory, size, etc.
- In f.3BSD, swap space is allocated in pieces that are multiples of power of 2 and minimum size, up to a maximum size determined by the size or the swap-space partition on the disk.





Paging

- Berkeley UNIX systems depend primarily on paging for memory-contention management, and depend only secondarily on swapping.
- *Demand paging* – When a process needs a page and the page is not there, a page fault to the kernel occurs, a frame of main memory is allocated, and the proper disk page is read into the frame.
- A *pagedaemon* process uses a modified second-chance page-replacement algorithm to keep enough free frames to support the executing processes.
- If the scheduler decides that the paging system is overloaded, processes will be swapped out whole until the overload is relieved.





File System

- The UNIX file system supports two main objects: files and directories.
- Directories are just files with a special format, so the representation of a file is the basic UNIX concept.





Blocks and Fragments

- Most of the file system is taken up by *data blocks*.
- 4.2BSD uses *two* block sizes for files which have no indirect blocks:
 - All the blocks of a file are of a large *block size* (such as 8K), except the last.
 - The last block is an appropriate multiple of a smaller *fragment size* (i.e., 1024) to fill out the file.
 - Thus, a file of size 18,000 bytes would have two 8K blocks and one 2K fragment (which would not be filled completely).





Blocks and Fragments (Cont.)

- The *block* and *fragment* sizes are set during file-system creation according to the intended use of the file system:
 - If many small files are expected, the fragment size should be small.
 - If repeated transfers of large files are expected, the basic block size should be large.
- The maximum block-to-fragment ratio is 8 : 1; the minimum block size is 4K (typical choices are 4096 : 512 and 8192 : 1024).





Inodes

- A file is represented by an *inode* — a record that stores information about a specific file on the disk.
- The inode also contains 15 pointer to the disk blocks containing the file's data contents.
 - First 12 point to *direct blocks*.
 - Next three point to *indirect blocks*
 - ▶ First indirect block pointer is the address of a *single indirect block* — an index block containing the addresses of blocks that do contain data.
 - ▶ Second is a *double-indirect-block pointer*, the address of a block that contains the addresses of blocks that contain pointer to the actual data blocks.
 - ▶ A *triple indirect* pointer is not needed; files with as many as 232 bytes will use only double indirection.





Directories

- The inode type field distinguishes between plain files and directories.
- Directory entries are of variable length; each entry contains first the length of the entry, then the file name and the inode number.
- The user refers to a file by a path name, whereas the file system uses the inode as its definition of a file.
 - The kernel has to map the supplied user path name to an inode
 - Directories are used for this mapping.





Directories (Cont.)

- First determine the starting directory:
 - If the first character is “/”, the starting directory is the root directory.
 - For any other starting character, the starting directory is the current directory.
- The search process continues until the end of the path name is reached and the desired inode is returned.
- Once the inode is found, a file structure is allocated to point to the inode.
- 4.3BSD improved file system performance by adding a directory name cache to hold recent directory-to-inode translations.





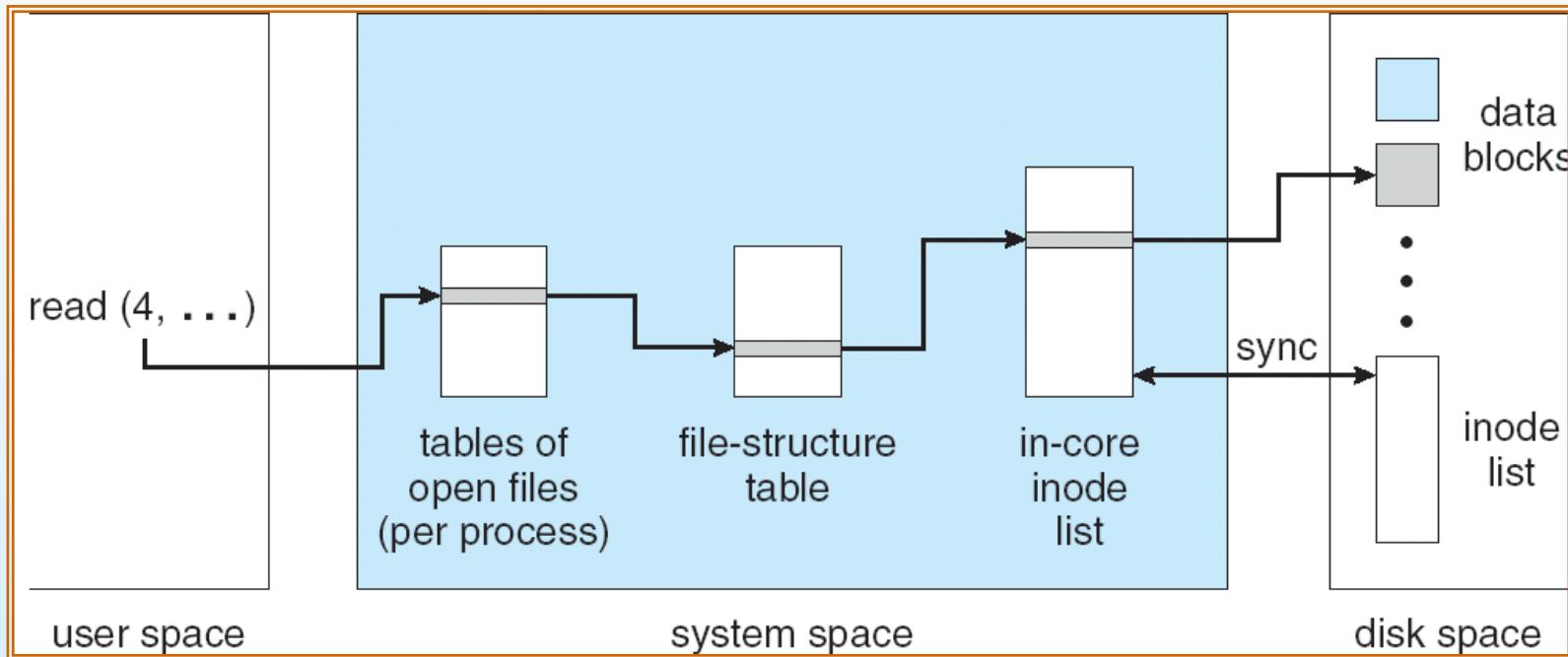
Mapping of a File Descriptor to an Inode

- System calls that refer to open files indicate the file is passing a file descriptor as an argument.
- The file descriptor is used by the kernel to index a table of open files for the current process.
- Each entry of the table contains a pointer to a file structure.
- This file structure in turn points to the inode.
- Since the open file table has a fixed length which is only setable at boot time, there is a fixed limit on the number of concurrently open files in a system.





File-System Control Blocks





Disk Structures

- The one file system that a user ordinarily sees may actually consist of several physical file systems, each on a different device.
- Partitioning a physical device into multiple file systems has several benefits.
 - Different file systems can support different uses.
 - Reliability is improved
 - Can improve efficiency by varying file-system parameters.
 - Prevents one program from using all available space for a large file.
 - Speeds up searches on backup tapes and restoring partitions from tape.





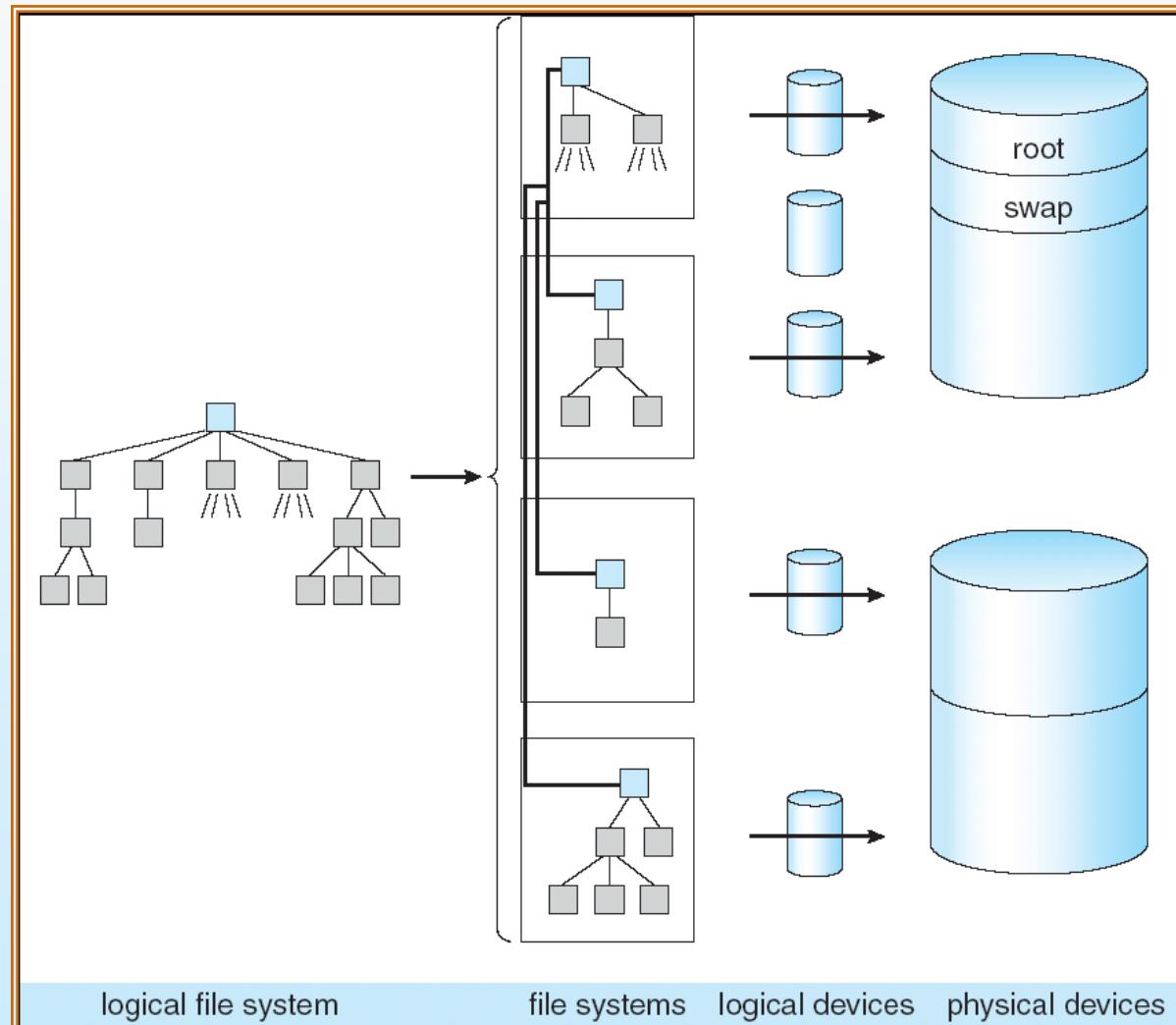
Disk Structures (Cont.)

- The *root file* system is always available on a drive.
- Other file systems may be *mounted* — i.e., integrated into the directory hierarchy of the root file system.
- The following figure illustrates how a directory structure is partitioned into file systems, which are mapped onto logical devices, which are partitions of physical devices.





Mapping File System to Physical Devices





Implementations

- The user interface to the file system is simple and well defined, allowing the implementation of the file system itself to be changed without significant effect on the user.
- For Version 7, the size of inodes doubled, the maximum file and file system sized increased, and the details of free-list handling and superblock information changed.
- In 4.0BSD, the size of blocks used in the file system was increased from 512 bytes to 1024 bytes — increased internal fragmentation, but doubled throughput.
- 4.2BSD added the Berkeley Fast File System, which increased speed, and included new features.
 - New directory system calls
 - **truncate** calls
 - Fast File System found in most implementations of UNIX.





Layout and Allocation Policy

- The kernel uses a *<logical device number, inode number>* pair to identify a file.
 - The logical device number defines the file system involved.
 - The inodes in the file system are numbered in sequence.
- 4.3BSD introduced the *cylinder group* — allows localization of the blocks in a file.
 - Each cylinder group occupies one or more consecutive cylinders of the disk, so that disk accesses within the cylinder group require minimal disk head movement.
 - Every cylinder group has a superblock, a cylinder block, an array of inodes, and some data blocks.





4.3BSD Cylinder Group

data blocks

superblock

cylinder block

inodes

data blocks





I/O System

- The I/O system hides the peculiarities of I/O devices from the bulk of the kernel.
- Consists of a buffer caching system, general device driver code, and drivers for specific hardware devices.
- Only the device driver knows the peculiarities of a specific device.





4.3 BSD Kernel I/O Structure

<i>system-call interface to the kernel</i>					
socket	plain file	cooked block interface	raw block interface	raw tty interface	cooked TTY
protocols	file system				line discipline
network interface	block-device driver			character-device driver	
<i>the hardware</i>					





Block Buffer Cache

- Consist of buffer headers, each of which can point to a piece of physical memory, as well as to a device number and a block number on the device.
- The buffer headers for blocks not currently in use are kept in several linked lists:
 - Buffers recently used, linked in LRU order (LRU list).
 - Buffers not recently used, or without valid contents (AGE list).
 - EMPTY buffers with no associated physical memory.
- When a block is wanted from a device, the cache is searched.
- If the block is found it is used, and no I/O transfer is necessary.
- If it is not found, a buffer is chosen from the AGE list, or the LRU list if AGE is empty.





Block Buffer Cache (Cont.)

- Buffer cache size effects system performance; if it is large enough, the percentage of cache hits can be high and the number of actual I/O transfers low.
- Data written to a disk file are buffered in the cache, and the disk driver sorts its output queue according to disk address — these actions allow the disk driver to minimize disk head seeks and to write data at times optimized for disk rotation.





Raw Device Interfaces

- Almost every block device has a character interface, or *raw device interface* — unlike the block interface, it bypasses the block buffer cache.
- Each disk driver maintains a queue of pending transfers.
- Each record in the queue specifies:
 - whether it is a read or a write
 - a main memory address for the transfer
 - a device address for the transfer
 - a transfer size
- It is simple to map the information from a block buffer to what is required for this queue.





C-Lists

- Terminal drivers use a character buffering system which involves keeping small blocks of characters in linked lists.
- A **write** system call to a terminal enqueues characters on a list for the device. An initial transfer is started, and interrupts cause dequeuing of characters and further transfers.
- Input is similarly interrupt driven.
- It is also possible to have the device driver bypass the canonical queue and return characters directly from the raw queue — *raw mode* (used by full-screen editors and other programs that need to react to every keystroke).





Interprocess Communication

- The *pipe* is the IPC mechanism most characteristic of UNIX.
 - Permits a reliable unidirectional byte stream between two processes.
 - A benefit of pipes small size is that pipe data are seldom written to disk; they usually are kept in memory by the normal block buffer cache.
- In 4.3BSD, pipes are implemented as a special case of the *socket* mechanism which provides a general interface not only to facilities such as pipes, which are local to one machine, but also to networking facilities.
- The socket mechanism can be used by unrelated processes.





Sockets

- A socket is an endpoint of communication.
- An in-use socket is usually bound with an address; the nature of the address depends on the *communication domain* of the socket.
- A characteristic property of a domain is that processes communication in the same domain use the same *address format*.
- A single socket can communicate in only one domain — the three domains currently implemented in 4.3BSD are:
 - the UNIX domain (AF_UNIX)
 - the Internet domain (AF_INET)
 - the XEROX Network Service (NS) domain (AF_NS)





Socket Types

- **Stream sockets** provide reliable, duplex, sequenced data streams. Supported in Internet domain by the TCP protocol. In UNIX domain, pipes are implemented as a pair of communicating stream sockets.
- **Sequenced packet sockets** provide similar data streams, except that record boundaries are provided. Used in XEROX AF_NS protocol.
- **Datagram sockets** transfer messages of variable size in either direction. Supported in Internet domain by UDP protocol
- **Reliably delivered message sockets** transfer messages that are guaranteed to arrive. Currently unsupported.
- **Raw sockets** allow direct access by processes to the protocols that support the other socket types; e.g., in the Internet domain, it is possible to reach TCP, IP beneath that, or a deeper Ethernet protocol. Useful for developing new protocols.





Socket System Calls

- The **socket** call creates a socket; takes as arguments specifications of the communication domain, socket type, and protocol to be used and returns a small integer called a *socket descriptor*.
- A name is bound to a socket by the **bind** system call.
- The **connect** system call is used to initiate a connection.
- A server process uses **socket** to create a socket and **bind** to bind the well-known address of its service to that socket.
 - Uses **listen** to tell the kernel that it is ready to accept connections from clients.
 - Uses **accept** to accept individual connections.
 - Uses **fork** to produce a new process after the **accept** to service the client while the original server process continues to listen for more connections.





Socket System Calls (Cont.)

- The simplest way to terminate a connection and to destroy the associated socket is to use the **close** system call on its socket descriptor.
- The **select** system call can be used to multiplex data transfers on several file descriptors and /or socket descriptors





Network Support

- Networking support is one of the most important features in 4.3BSD.
- The socket concept provides the programming mechanism to access other processes, even across a network.
- Sockets provide an interface to several sets of protocols.
- Almost all current UNIX systems support UUCP.
- 4.3BSD supports the DARPA Internet protocols UDP, TCP, IP, and ICMP on a wide range of Ethernet, token-ring, and ARPANET interfaces.
- The 4.3BSD networking implementation, and to a certain extent the socket facility, is more oriented toward the ARPANET Reference Model (ARM).





Network Reference models and Layering

ISO reference model	ARPANET reference model	4.2BSD layers	example layering
application	process applications	user programs and libraries	telnet
presentation		sockets	sock_stream
session transport		protocol	TCP
network data link	host–host	network interfaces	IP
hardware		network hardware	Ethernet driver
	network hardware	network hardware	interlan controller



End of Module A



Module C: Windows 2000





Module C: Windows 2000

- History
- Design Principles
- System Components
- Environmental Subsystems
- File system
- Networking
- Programmer Interface





Windows 2000

- 32-bit preemptive multitasking operating system for Intel microprocessors.
- Key goals for the system:
 - portability
 - security
 - POSIX compliance
 - multiprocessor support
 - extensibility
 - international support
 - compatibility with MS-DOS and MS-Windows applications.
- Uses a micro-kernel architecture.
- Available in four versions, Professional, Server, Advanced Server, National Server.
- In 1996, more NT server licenses were sold than UNIX licenses





History

- In 1988, Microsoft decided to develop a “new technology” (NT) portable operating system that supported both the OS/2 and POSIX APIs.
- Originally, NT was supposed to use the OS/2 API as its native environment but during development NT was changed to use the Win32 API, reflecting the popularity of Windows 3.0.





Design Principles

- Extensibility — layered architecture.
 - Executive, which runs in protected mode, provides the basic system services.
 - On top of the executive, several server subsystems operate in user mode.
 - Modular structure allows additional environmental subsystems to be added without affecting the executive.
- Portability — 2000 can be moved from one hardware architecture to another with relatively few changes.
 - Written in C and C++.
 - Processor-dependent code is isolated in a dynamic link library (DLL) called the “hardware abstraction layer” (HAL).





Design Principles (Cont.)

- Reliability — 2000 uses hardware protection for virtual memory, and software protection mechanisms for operating system resources.
- Compatibility — applications that follow the IEEE 1003.1 (POSIX) standard can be complied to run on 2000 without changing the source code.
- Performance — 2000 subsystems can communicate with one another via high-performance message passing.
 - Preemption of low priority threads enables the system to respond quickly to external events.
 - Designed for symmetrical multiprocessing
- International support — supports different locales via the national language support (NLS) API.





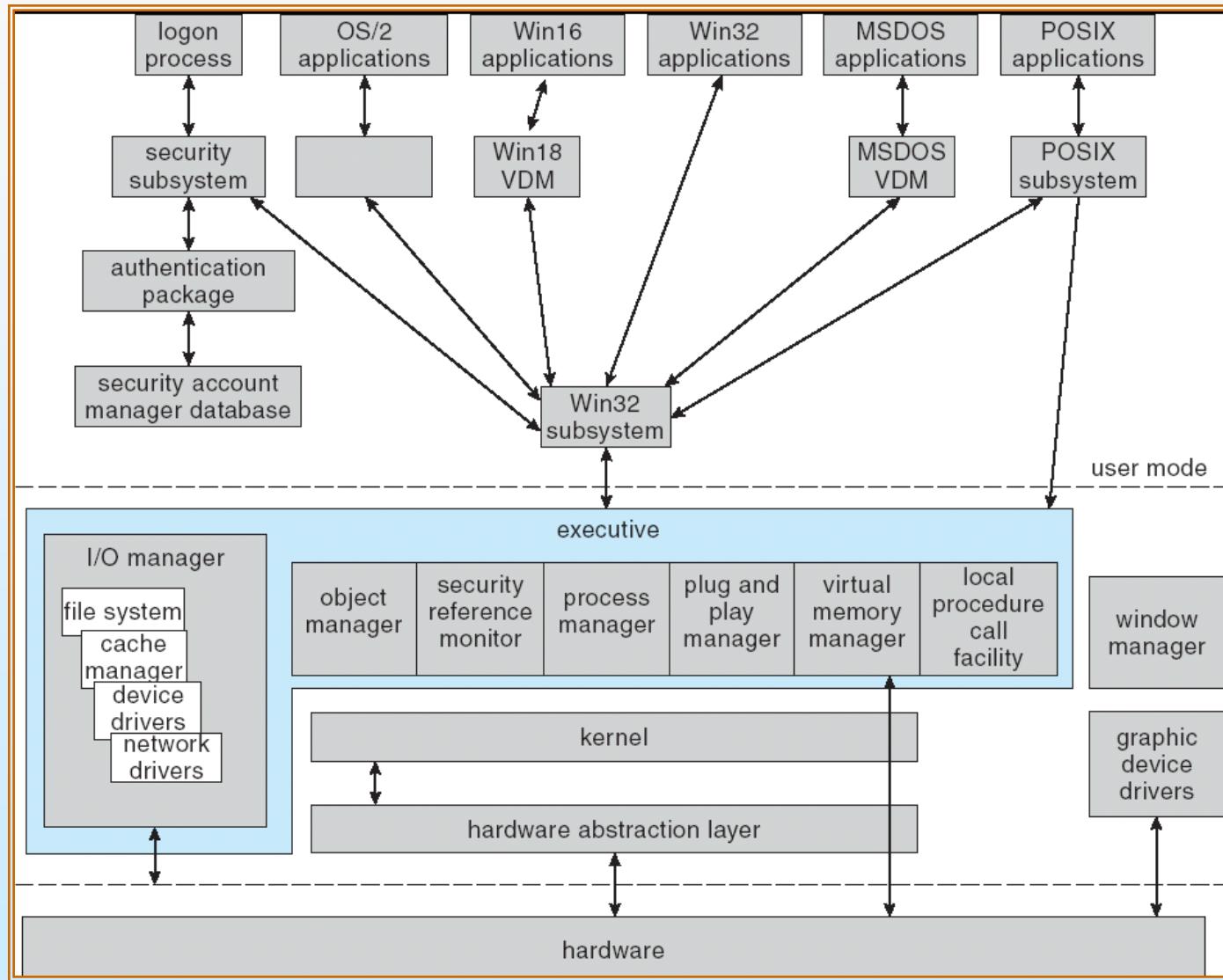
2000 Architecture

- Layered system of modules.
- Protected mode — HAL, kernel, executive.
- User mode — collection of subsystems
 - Environmental subsystems emulate different operating systems.
 - Protection subsystems provide security functions.





Depiction of 2000 Architecture





System Components — Kernel

- Foundation for the executive and the subsystems.
- Never paged out of memory; execution is never preempted.
- Four main responsibilities:
 - thread scheduling
 - interrupt and exception handling
 - low-level processor synchronization
 - recovery after a power failure
- Kernel is object-oriented, uses two sets of objects.
 - *dispatcher objects* control dispatching and synchronization (events, mutants, mutexes, semaphores, threads and timers).
 - *control objects* (asynchronous procedure calls, interrupts, power notify, power status, process and profile objects.)





Kernel — Process and Threads

- The process has a virtual memory address space, information (such as a base priority), and an affinity for one or more processors.
- Threads are the unit of execution scheduled by the kernel's dispatcher.
- Each thread has its own state, including a priority, processor affinity, and accounting information.
- A thread can be one of six states: *ready*, *standby*, *running*, *waiting*, *transition*, and *terminated*.





Kernel — Scheduling

- The dispatcher uses a 32-level priority scheme to determine the order of thread execution. Priorities are divided into two classes..
 - The real-time class contains threads with priorities ranging from 16 to 31.
 - The variable class contains threads having priorities from 0 to 15.
- Characteristics of 2000's priority strategy.
 - Trends to give very good response times to interactive threads that are using the mouse and windows.
 - Enables I/O-bound threads to keep the I/O devices busy.
 - Complete-bound threads soak up the spare CPU cycles in the background.





Kernel — Scheduling (Cont.)

- Scheduling can occur when a thread enters the ready or wait state, when a thread terminates, or when an application changes a thread's priority or processor affinity.
- Real-time threads are given preferential access to the CPU; but 2000 does not guarantee that a real-time thread will start to execute within any particular time limit. (This is known as *soft realtime*.)





Windows 2000 Interrupt Request Levels

interrupt levels	types of interrupts
31	machine check or bus error
30	power fail
29	interprocessor notification (request another processor to act; e.g., dispatch a process or update the TLB)
28	clock (used to keep track of time)
27	profile
3–26	traditional PC IRQ hardware interrupts
2	dispatch and deferred procedure call (DPC) (kernel)
1	asynchronous procedure call (APC)
0	passive





Kernel — Trap Handling

- The kernel provides trap handling when exceptions and interrupts are generated by hardware or software.
- Exceptions that cannot be handled by the trap handler are handled by the kernel's *exception dispatcher*.
- The interrupt dispatcher in the kernel handles interrupts by calling either an interrupt service routine (such as in a device driver) or an internal kernel routine.
- The kernel uses spin locks that reside in global memory to achieve multiprocessor mutual exclusion.





Executive — Object Manager

- 2000 uses objects for all its services and entities; the object manager supervises the use of all the objects.
 - Generates an object *handle*
 - Checks security.
 - Keeps track of which processes are using each object.
- Objects are manipulated by a standard set of methods, namely `create`, `open`, `close`, `delete`, `query name`, `parse` and `Security`.





Executive — Naming Objects

- The 2000 executive allows any object to be given a name, which may be either permanent or temporary.
- Object names are structured like file path names in MS-DOS and UNIX.
- 2000 implements a *symbolic link object*, which is similar to *symbolic links* in UNIX that allow multiple nicknames or aliases to refer to the same file.
- A process gets an object handle by creating an object by opening an existing one, by receiving a duplicated handle from another process, or by inheriting a handle from a parent process.
- Each object is protected by an access control list.





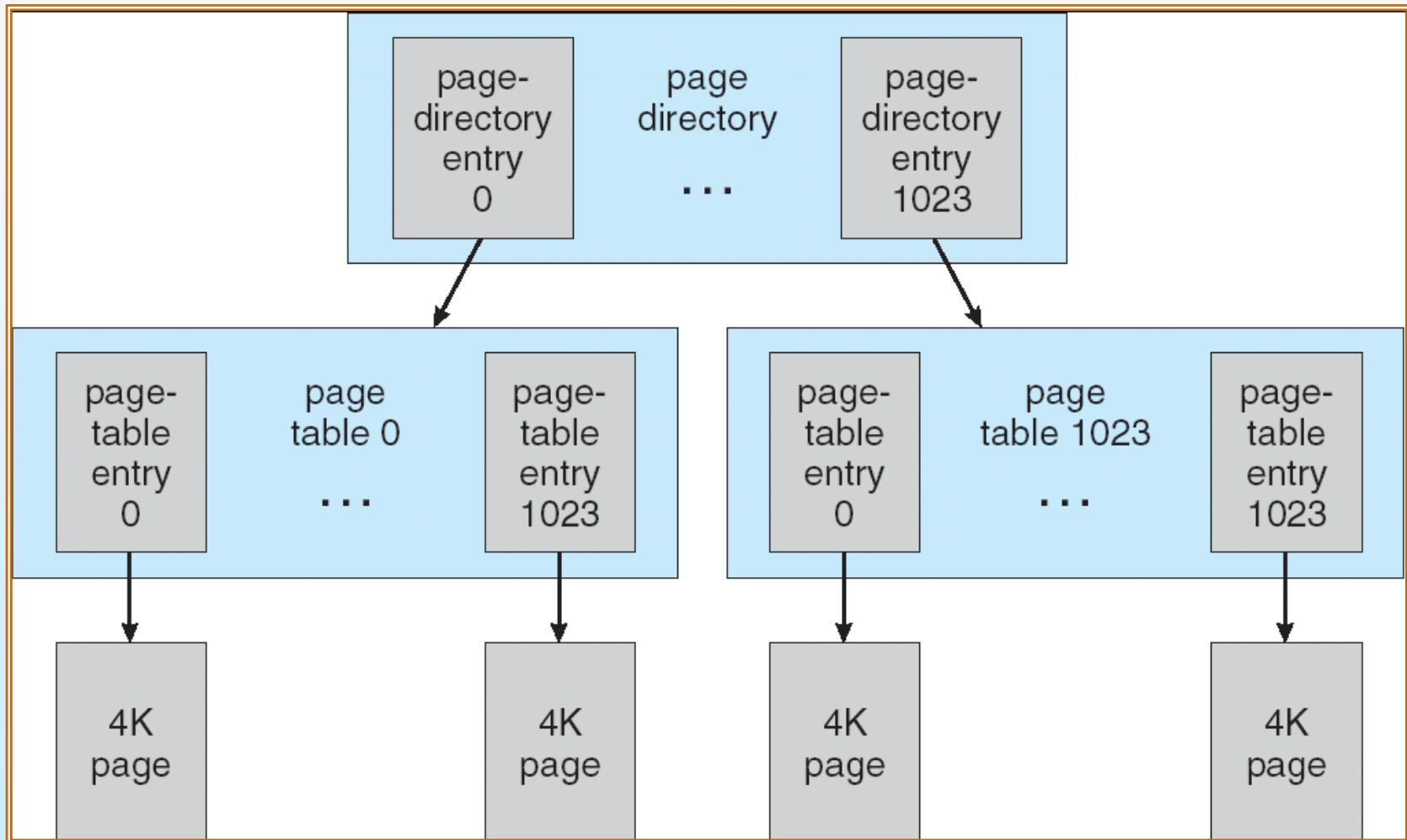
Executive — Virtual Memory Manager

- The design of the VM manager assumes that the underlying hardware supports virtual to physical mapping a paging mechanism, transparent cache coherence on multiprocessor systems, and virtual addressing aliasing.
- The VM manager in 2000 uses a page-based management scheme with a page size of 4 KB.
- The 2000 VM manager uses a two step process to allocate memory.
 - The first step reserves a portion of the process's address space.
 - The second step commits the allocation by assigning space in the 2000 paging file.





Virtual-Memory Layout





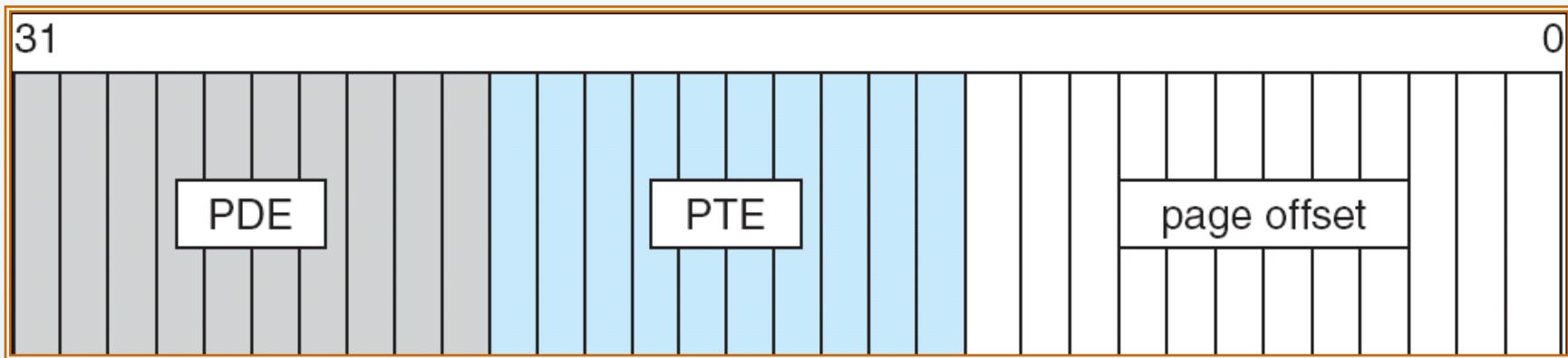
Virtual Memory Manager (Cont.)

- The virtual address translation in 2000 uses several data structures.
 - Each process has a *page directory* that contains 1024 *page directory entries* of size 4 bytes.
 - Each page directory entry points to a *page table* which contains 1024 *page table entries* (PTEs) of size 4 bytes.
 - Each PTE points to a 4 KB *page frame* in physical memory.
- A 10-bit integer can represent all the values from 0 to 1023, therefore, can select any entry in the page directory, or in a page table.
- This property is used when translating a virtual address pointer to a byte address in physical memory.
- A page can be in one of six states: valid, zeroed, free standby, modified and bad.





Virtual-to-Physical Address Translation

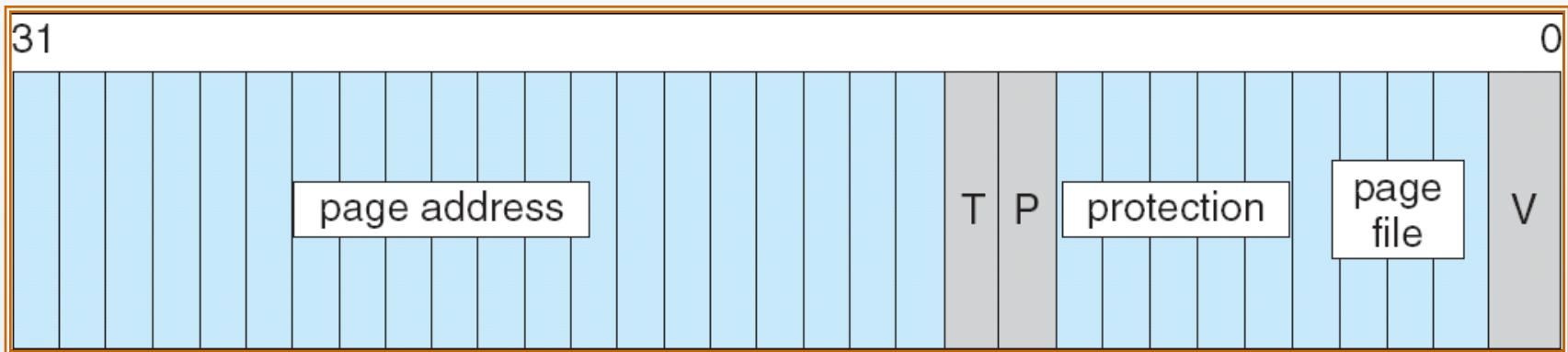


- 10 bits for page directory entry, 20 bits for page table entry, and 12 bits for byte offset in page.





Page File Page-Table Entry



5 bits for page protection, 20 bits for page frame address, 4 bits to select a paging file, and 3 bits that describe the page state. $V = 0$





Executive — Process Manager

- Provides services for creating, deleting, and using threads and processes.
- Issues such as parent/child relationships or process hierarchies are left to the particular environmental subsystem that owns the process.





Executive — Local Procedure Call Facility

- The LPC passes requests and results between client and server processes within a single machine.
- In particular, it is used to request services from the various 2000 subsystems.
- When a LPC channel is created, one of three types of message passing techniques must be specified.
 - First type is suitable for small messages, up to 256 bytes; port's message queue is used as intermediate storage, and the messages are copied from one process to the other.
 - Second type avoids copying large messages by pointing to a shared memory section object created for the channel.
 - Third method, called *quick* LPC was used by graphical display portions of the Win32 subsystem.





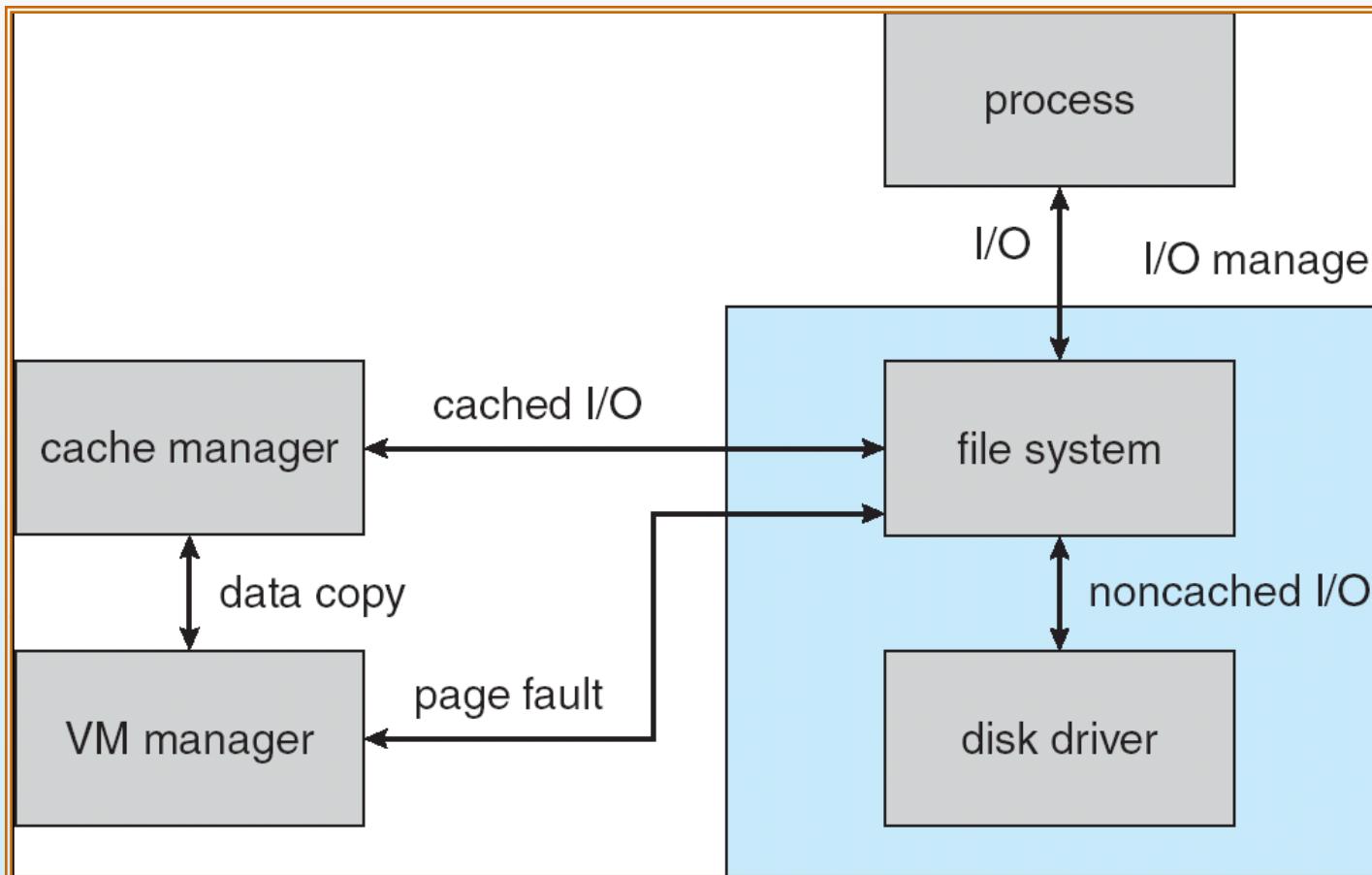
Executive — I/O Manager

- The I/O manager is responsible for
 - file systems
 - cache management
 - device drivers
 - network drivers
- Keeps track of which installable file systems are loaded, and manages buffers for I/O requests.
- Works with VM Manager to provide memory-mapped file I/O.
- Controls the 2000 cache manager, which handles caching for the entire I/O system.
- Supports both synchronous and asynchronous operations, provides time outs for drivers, and has mechanisms for one driver to call another.





File I/O





Executive — Security Reference Monitor

- The object-oriented nature of 2000 enables the use of a uniform mechanism to perform runtime access validation and audit checks for every entity in the system.

- Whenever a process opens a handle to an object, the security reference monitor checks the process's security token and the object's access control list to see whether the process has the necessary rights.





Executive – Plug-and-Play Manager

- Plug-and-Play (PnP) manager is used to recognize and adapt to changes in the hardware configuration.
- When new devices are added (for example, PCI or USB), the PnP manager loads the appropriate driver.
- The manager also keeps track of the resources used by each device.





Environmental Subsystems

- User-mode processes layered over the native 2000 executive services to enable 2000 to run programs developed for other operating system.
- 2000 uses the Win32 subsystem as the main operating environment; Win32 is used to start all processes. It also provides all the keyboard, mouse and graphical display capabilities.
- MS-DOS environment is provided by a Win32 application called the *virtual dos machine* (VDM), a user-mode process that is paged and dispatched like any other 2000 thread.





Environmental Subsystems (Cont.)

- 16-Bit Windows Environment:
 - Provided by a VDM that incorporates *Windows on Windows*.
 - Provides the Windows 3.1 kernel routines and sub routines for window manager and GDI functions.
- The POSIX subsystem is designed to run POSIX applications following the POSIX.1 standard which is based on the UNIX model.





Environmental Subsystems (Cont.)

- OS/2 subsystems runs OS/2 applications.
- Logon and Security Subsystems authenticates users logging to Windows 2000 systems. Users are required to have account names and passwords.
 - The authentication package authenticates users whenever they attempt to access an object in the system. Windows 2000 uses Kerberos as the default authentication package.





File System

- The fundamental structure of the 2000 file system (NTFS) is a *volume*.
 - Created by the 2000 disk administrator utility.
 - Based on a logical disk partition.
 - May occupy portions of a disk, an entire disk, or span across several disks.
- All *metadata*, such as information about the volume, is stored in a regular file.
- NTFS uses *clusters* as the underlying unit of disk allocation.
 - A cluster is a number of disk sectors that is a power of two.
 - Because the cluster size is smaller than for the 16-bit FAT file system, the amount of internal fragmentation is reduced.





File System — Internal Layout

- NTFS uses logical cluster numbers (LCNs) as disk addresses.
- A file in NTFS is not a simple byte stream, as in MS-DOS or UNIX, rather, it is a structured object consisting of attributes.
- Every file in NTFS is described by one or more records in an array stored in a special file called the Master File Table (MFT).
- Each file on an NTFS volume has a unique ID called a file reference.
 - 64-bit quantity that consists of a 48-bit file number and a 16-bit sequence number.
 - Can be used to perform internal consistency checks.
- The NTFS name space is organized by a hierarchy of directories; the index root contains the top level of the B+ tree.





File System — Recovery

- All file system data structure updates are performed inside transactions that are logged.
 - Before a data structure is altered, the transaction writes a log record that contains redo and undo information.
 - After the data structure has been changed, a commit record is written to the log to signify that the transaction succeeded.
 - After a crash, the file system data structures can be restored to a consistent state by processing the log records.





File System — Recovery (Cont.)

- This scheme does not guarantee that all the user file data can be recovered after a crash, just that the file system data structures (the metadata files) are undamaged and reflect some consistent state prior to the crash.
- The log is stored in the third metadata file at the beginning of the volume.
- The logging functionality is provided by the 2000 *log file service*.





File System — Security

- Security of an NTFS volume is derived from the 2000 object model.
- Each file object has a security descriptor attribute stored in this MFT record.
- This attribute contains the access token of the owner of the file, and an access control list that states the access privileges that are granted to each user that has access to the file.





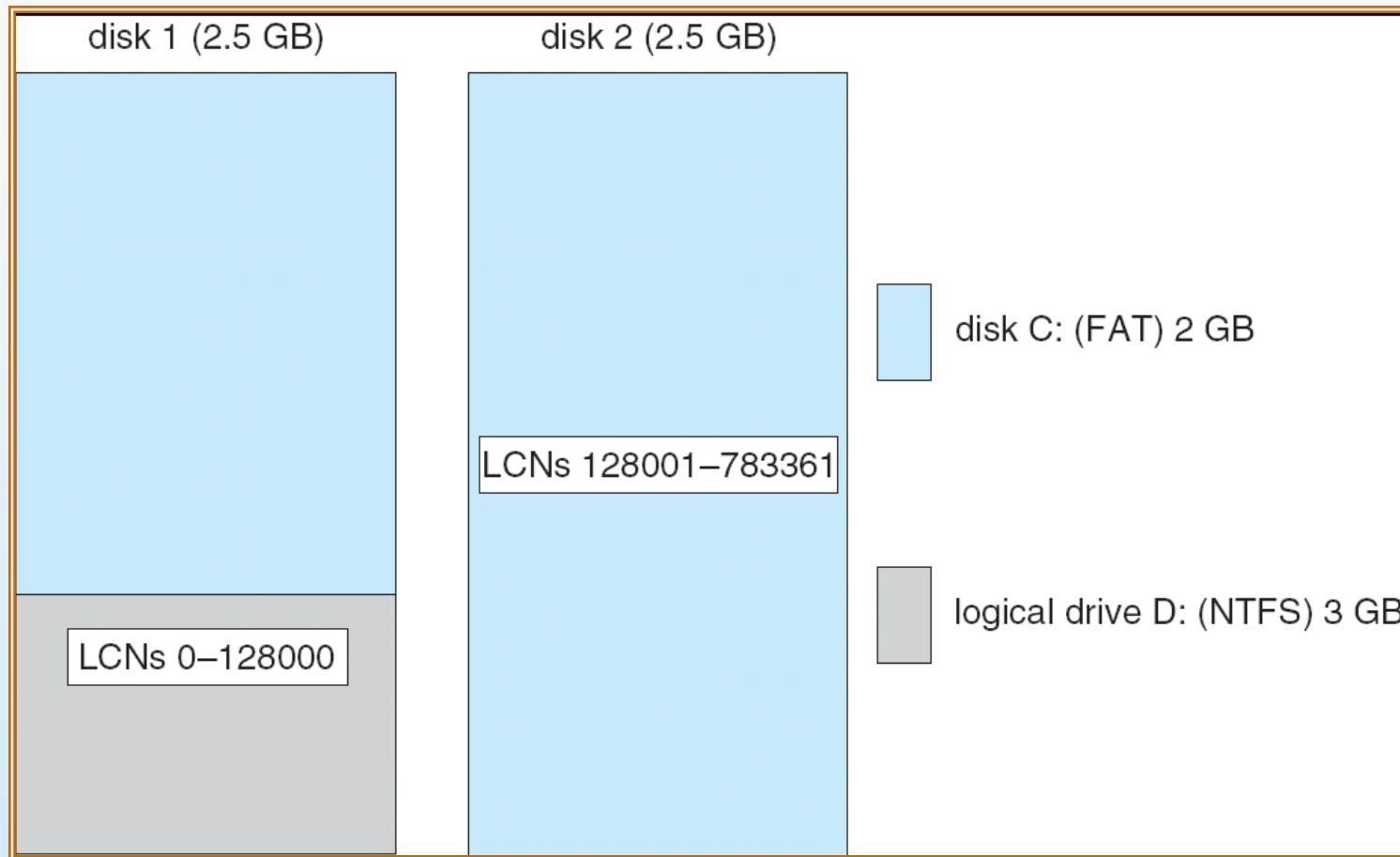
Volume Management and Fault Tolerance

- FtDisk, the fault tolerant disk driver for 2000, provides several ways to combine multiple SCSI disk drives into one logical volume.
- Logically concatenate multiple disks to form a large logical volume, a *volume set*.
- Interleave multiple physical partitions in round-robin fashion to form a *stripe set* (also called RAID level 0, or “disk striping”).
 - Variation: *stripe set with parity*, or RAID level 5.
- Disk mirroring, or RAID level 1, is a robust scheme that uses a *mirror set* — two equally sized partitions on tow disks with identical data contents.
- To deal with disk sectors that go bad, FtDisk, uses a hardware technique called *sector sparing* and NTFS uses a software technique called *cluster remapping*.



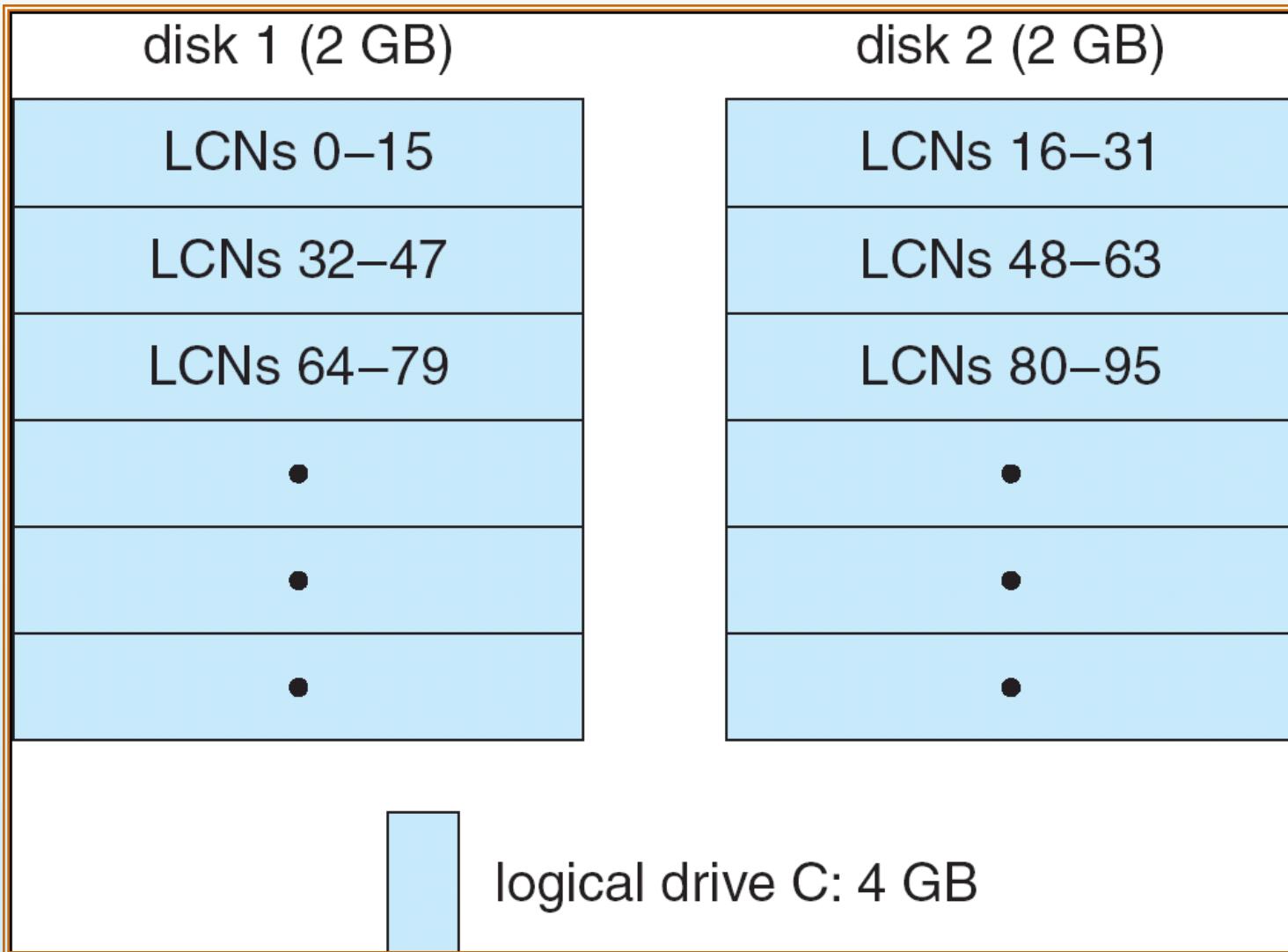


Volume Set On Two Drives



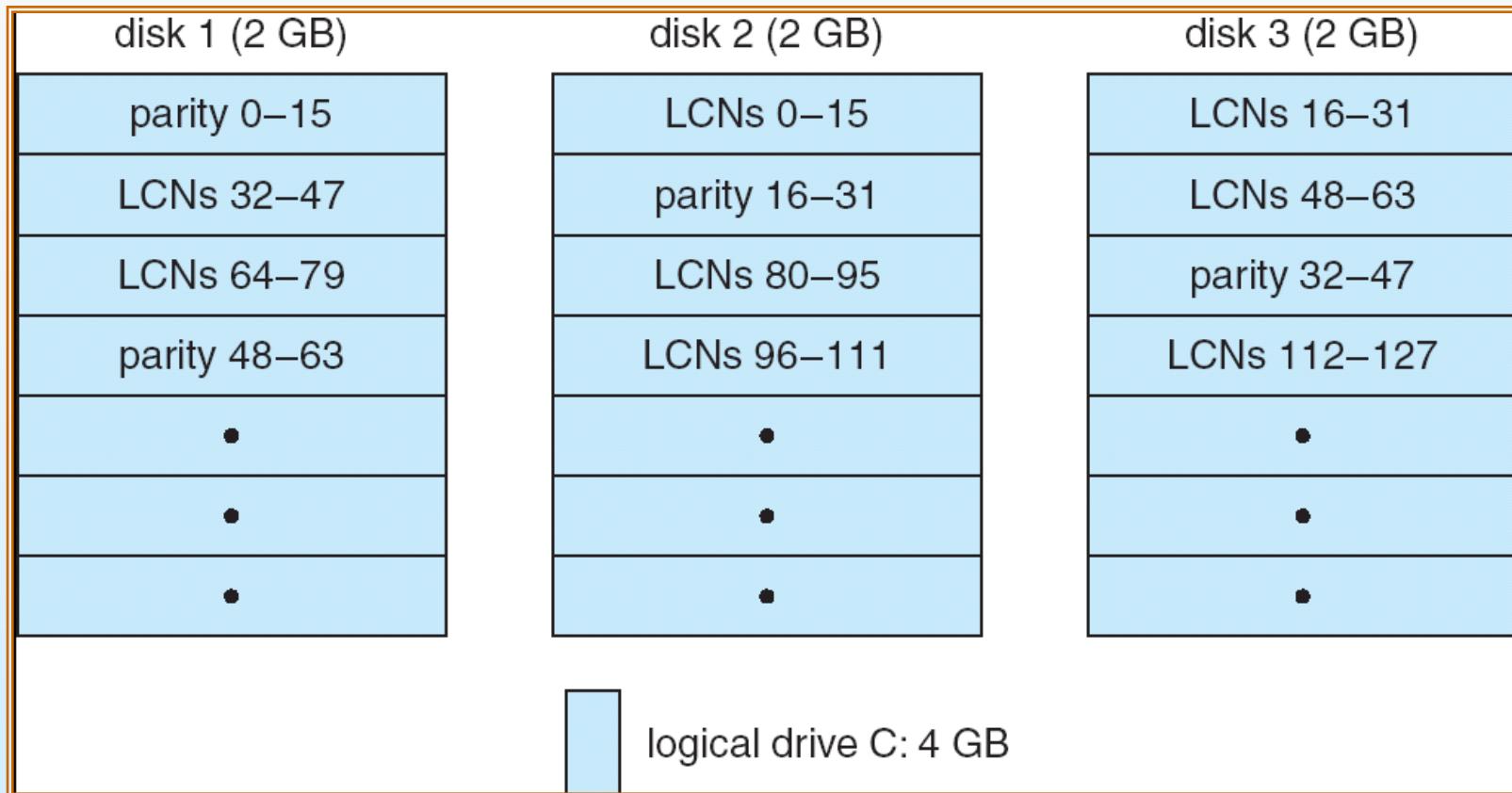


Stripe Set on Two Drives



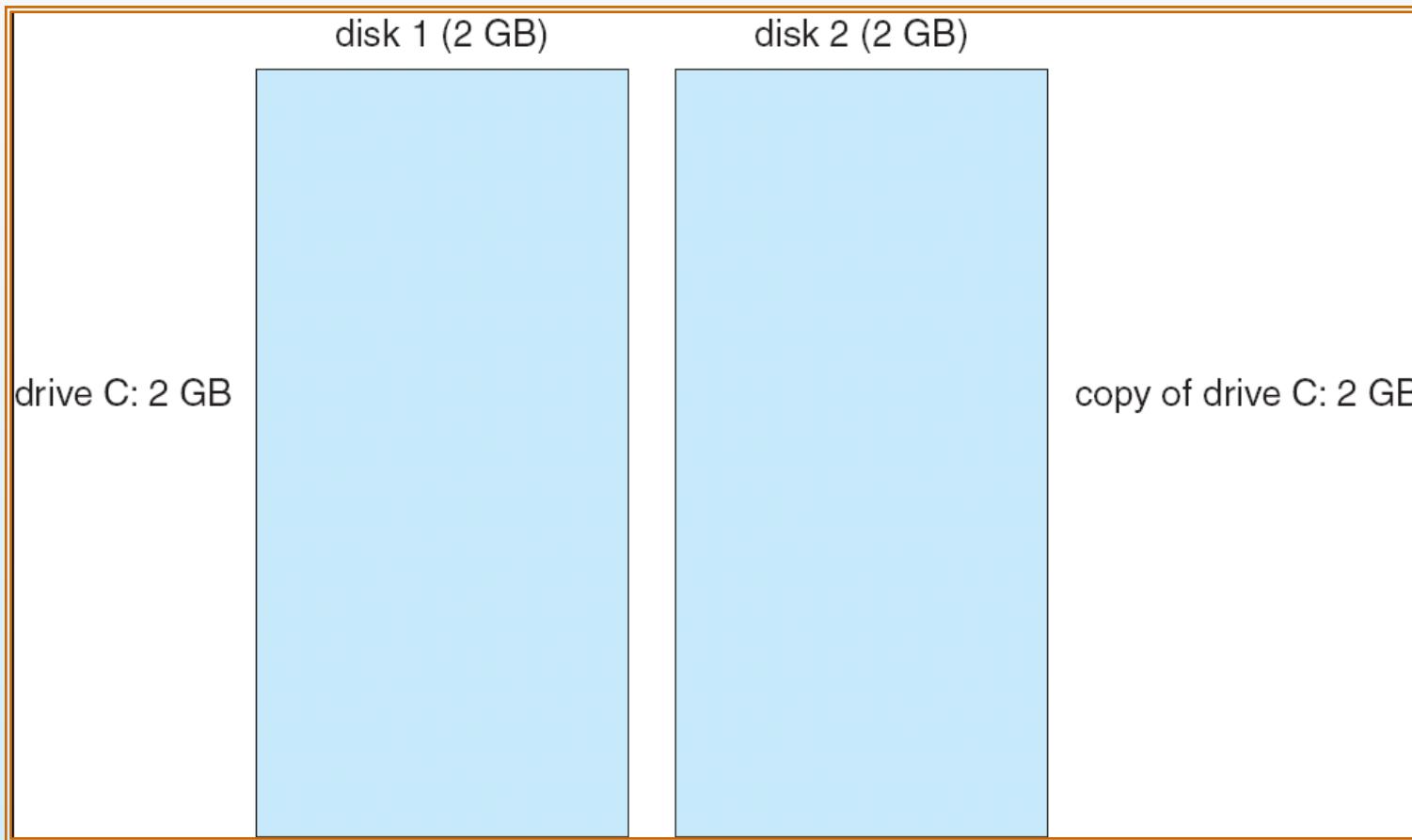


Stripe Set With Parity on Three Drives





Mirror Set on Two Drives





File System — Compression

- To compress a file, NTFS divides the file's data into *compression units*, which are blocks of 16 contiguous clusters.
- For sparse files, NTFS uses another technique to save space.
 - Clusters that contain all zeros are not actually allocated or stored on disk.
 - Instead, gaps are left in the sequence of virtual cluster numbers stored in the MFT entry for the file.
 - When reading a file, if a gap in the virtual cluster numbers is found, NTFS just zero-fills that portion of the caller's buffer.





File System — Reparse Points

- A reparse point returns an error code when accessed. The reparse data tells the I/O manager what to do next.
- Reparse points can be used to provide the functionality of UNIX *mounts*
- Reparse points can also be used to access files that have been moved to offline storage.





Networking

- 2000 supports both peer-to-peer and client/server networking; it also has facilities for network management.
- To describe networking in 2000, we refer to two of the internal networking interfaces:
 - NDIS (Network Device Interface Specification) — Separates network adapters from the transport protocols so that either can be changed without affecting the other.
 - TDI (Transport Driver Interface) — Enables any session layer component to use any available transport mechanism.
- 2000 implements transport protocols as drivers that can be loaded and unloaded from the system dynamically.





Networking — Protocols

- The server message block (SMB) protocol is used to send I/O requests over the network. It has four message types:
 - Session control
 - File
 - Printer
 - Message
- The network basic Input/Output system (NetBIOS) is a hardware abstraction interface for networks. Used to:
 - Establish logical names on the network.
 - Establish logical connections of sessions between two logical names on the network.
 - Support reliable data transfer for a session via NetBIOS requests or *SMBs*





Networking — Protocols (Cont.)

- NetBEUI (NetBIOS Extended User Interface): default protocol for Windows 95 peer networking and Windows for Workgroups; used when 2000 wants to share resources with these networks.
- 2000 uses the TCP/IP Internet protocol to connect to a wide variety of operating systems and hardware platforms.
- PPTP (Point-to-Point Tunneling Protocol) is used to communicate between Remote Access Server modules running on 2000 machines that are connected over the Internet.
- The 2000 NWLink protocol connects the NetBIOS to Novell NetWare networks.





Networking — Protocols (Cont.)

- The Data Link Control protocol (DLC) is used to access IBM mainframes and HP printers that are directly connected to the network.
- 2000 systems can communicate with Macintosh computers via the Apple Talk protocol if an 2000 Server on the network is running the Windows 2000 Services for Macintosh package.





Networking — Dist. Processing Mechanisms

- 2000 supports distributed applications via named NetBIOS, named pipes and mailslots, Windows Sockets, Remote Procedure Calls (RPC), and Network Dynamic Data Exchange (NetDDE).
- NetBIOS applications can communicate over the network using NetBEUI, NWLink, or TCP/IP.
- Named pipes are connection-oriented messaging mechanism that are named via the uniform naming convention (UNC).
- Mailslots are a connectionless messaging mechanism that are used for broadcast applications, such as for finding components on the network,
- Winsock, the windows sockets API, is a session-layer interface that provides a standardized interface to many transport protocols that may have different addressing schemes.





Distributed Processing Mechanisms (Cont.)

- The 2000 RPC mechanism follows the widely-used Distributed Computing Environment standard for RPC messages, so programs written to use 2000 RPCs are very portable.
 - RPC messages are sent using NetBIOS, or Winsock on TCP/IP networks, or named pipes on LAN Manager networks.
 - 2000 provides the Microsoft *Interface Definition Language* to describe the remote procedure names, arguments, and results.





Networking — Redirectors and Servers

- In 2000, an application can use the 2000 I/O API to access files from a remote computer as if they were local, provided that the remote computer is running an MS-NET server.
- A *redirector* is the client-side object that forwards I/O requests to remote files, where they are satisfied by a server.
- For performance and security, the redirectors and servers run in kernel mode.





Access to a Remote File

- The application calls the I/O manager to request that a file be opened (we assume that the file name is in the standard UNC format).
- The I/O manager builds an I/O request packet.
- The I/O manager recognizes that the access is for a remote file, and calls a driver called a Multiple Universal Naming Convention Provider (MUP).
- The MUP sends the I/O request packet asynchronously to all registered redirectors.
- A redirector that can satisfy the request responds to the MUP.
 - To avoid asking all the redirectors the same question in the future, the MUP uses a cache to remember which redirector can handle this file.





Access to a Remote File (Cont.)

- The redirector sends the network request to the remote system.
- The remote system network drivers receive the request and pass it to the server driver.
- The server driver hands the request to the proper local file system driver.
- The proper device driver is called to access the data.
- The results are returned to the server driver, which sends the data back to the requesting redirector.





Networking — Domains

- NT uses the concept of a domain to manage global access rights within groups.
- A domain is a group of machines running NT server that share a common security policy and user database.
- 2000 provides three models of setting up trust relationships.
 - *One way, A trusts B*
 - *Two way, transitive, A trusts B, B trusts C so A, B, C trust each other*
 - *Crosslink – allows authentication to bypass hierarchy to cut down on authentication traffic.*





Name Resolution in TCP/IP Networks

- On an IP network, name resolution is the process of converting a computer name to an IP address.

e.g., `www.bell-labs.com` resolves to `135.104.1.14`

- 2000 provides several methods of name resolution:
 - Windows Internet Name Service (WINS)
 - broadcast name resolution
 - domain name system (DNS)
 - a host file
 - an LMHOSTS file





Name Resolution (Cont.)

- WINS consists of two or more WINS servers that maintain a dynamic database of name to IP address bindings, and client software to query the servers.
- WINS uses the Dynamic Host Configuration Protocol (DHCP), which automatically updates address configurations in the WINS database, without user or administrator intervention.





Programmer Interface — Access to Kernel Obj.

- A process gains access to a kernel object named `xxx` by calling the `CreateXXX` function to open a *handle* to `xxx`; the handle is unique to that process.
- A handle can be closed by calling the `CloseHandle` function; the system may delete the object if the count of processes using the object drops to 0.
- 2000 provides three ways to share objects between processes.
 - A child process inherits a handle to the object.
 - One process gives the object a name when it is created and the second process opens that name.
 - `DuplicateHandle` function:
 - ▶ Given a handle to process and the handle's value a second process can get a handle to the same object, and thus share it.





Programmer Interface — Process Management

- Process is started via the `CreateProcess` routine which loads any dynamic link libraries that are used by the process, and creates a *primary thread*.
- Additional threads can be created by the `CreateThread` function.
- Every dynamic link library or executable file that is loaded into the address space of a process is identified by an *instance handle*.





Process Management (Cont.)

- Scheduling in Win32 utilizes four priority classes:
 - IDLE_PRIORITY_CLASS (priority level 4)
 - NORMAL_PRIORITY_CLASS (level 8 — typical for most processes)
 - HIGH_PRIORITY_CLASS (level 13)
 - REALTIME_PRIORITY_CLASS (level 24)
- To provide performance levels needed for interactive programs, 2000 has a special scheduling rule for processes in the NORMAL_PRIORITY_CLASS .
 - 2000 distinguishes between the *foreground process* that is currently selected on the screen, and the *background processes* that are not currently selected.
 - When a process moves into the foreground, 2000 increases the scheduling quantum by some factor, typically 3.





Process Management (Cont.)

- The kernel dynamically adjusts the priority of a thread depending on whether it is I/O-bound or CPU-bound.
- To synchronize the concurrent access to shared objects by threads, the kernel provides synchronization objects, such as semaphores and mutexes.
 - In addition, threads can synchronize by using the `WaitForSingleObject` or `WaitForMultipleObjects` functions.
 - Another method of synchronization in the Win32 API is the critical section.





Process Management (Cont.)

- A fiber is user-mode code that gets scheduled according to a user-defined scheduling algorithm.
 - Only one fiber at a time is permitted to execute, even on multiprocessor hardware.
 - 2000 includes fibers to facilitate the porting of legacy UNIX applications that are written for a fiber execution model.





Programmer Interface — Interprocess Comm.

- Win32 applications can have interprocess communication by sharing kernel objects.
- An alternate means of interprocess communications is message passing, which is particularly popular for Windows GUI applications.
 - One thread sends a message to another thread or to a window.
 - A thread can also send data with the message.
- Every Win32 thread has its own input queue from which the thread receives messages.
- This is more reliable than the shared input queue of 16-bit windows, because with separate queues, one stuck application cannot block input to the other applications.





Programmer Interface — Memory Management

- Virtual memory:
 - `VirtualAlloc` reserves or commits virtual memory.
 - `VirtualFree` decommits or releases the memory.
 - These functions enable the application to determine the virtual address at which the memory is allocated.
- An application can use memory by memory mapping a file into its address space.
 - Multistage process.
 - Two processes share memory by mapping the same file into their virtual memory.





Memory Management (Cont.)

- A heap in the Win32 environment is a region of reserved address space.
 - A Win 32 process is created with a 1 MB *default heap*.
 - Access is synchronized to protect the heap's space allocation data structures from damage by concurrent updates by multiple threads.
- Because functions that rely on global or static data typically fail to work properly in a multithreaded environment, the thread-local storage mechanism allocates global storage on a per-thread basis.
 - The mechanism provides both dynamic and static methods of creating thread-local storage.



End of Module C

