# UNIT – V

**File System Implementation**

# File Concept

* Contiguous logical address space and 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

* *Open(Fi)* – search the directory structure on disk for entry *Fi*, and move the content of entry to memory
* *Close (Fi)* – move the content of entry *Fi* 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

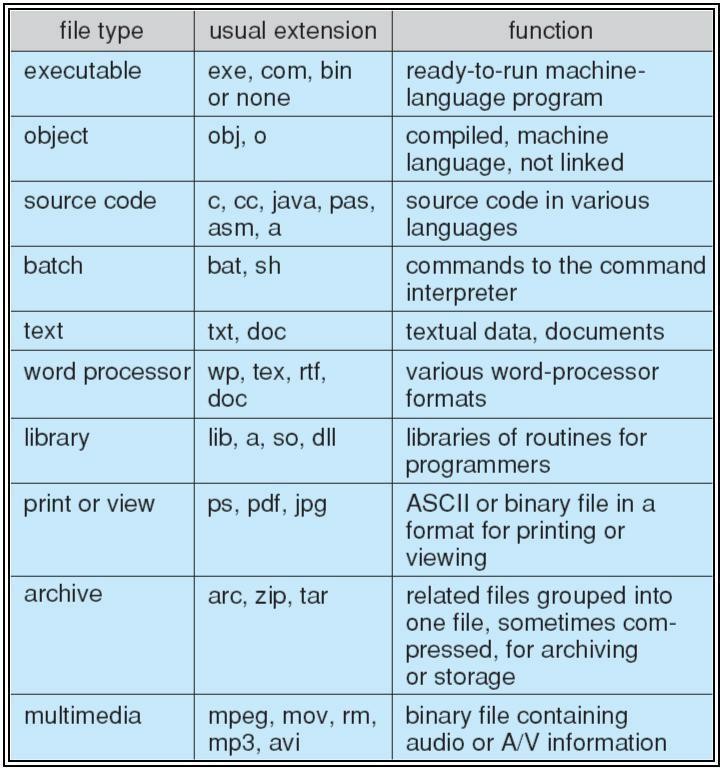
when last processes close it

* 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

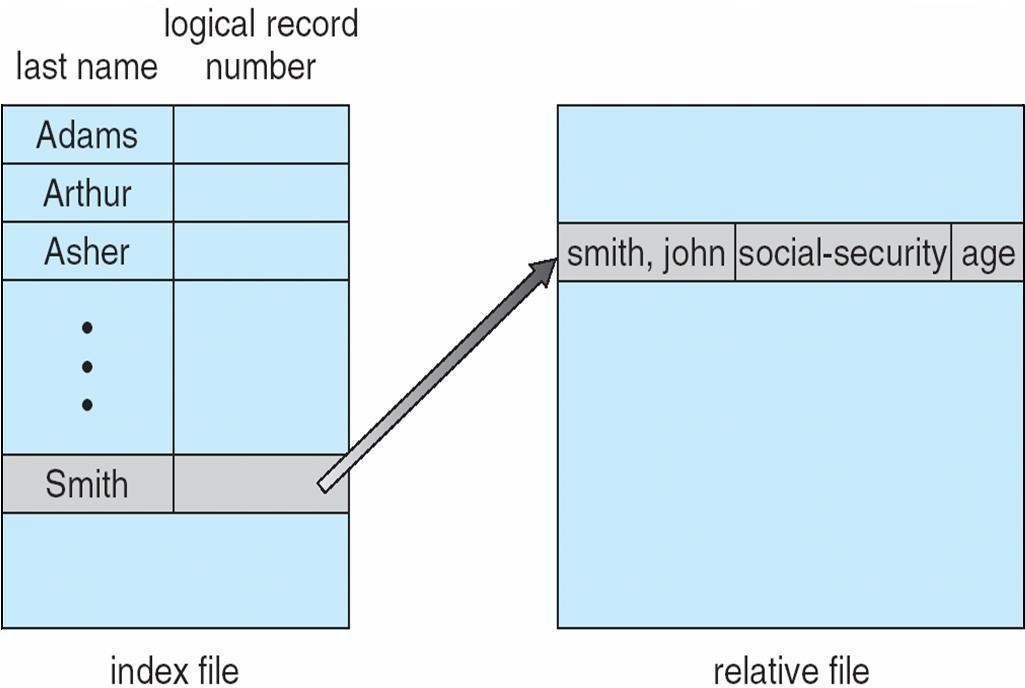
# File Types – Name, Extension



**Access Methods Sequential Access**

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

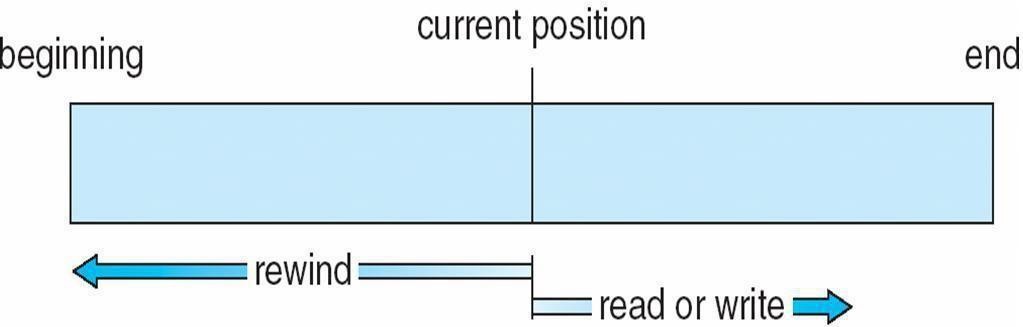
# Direct Access



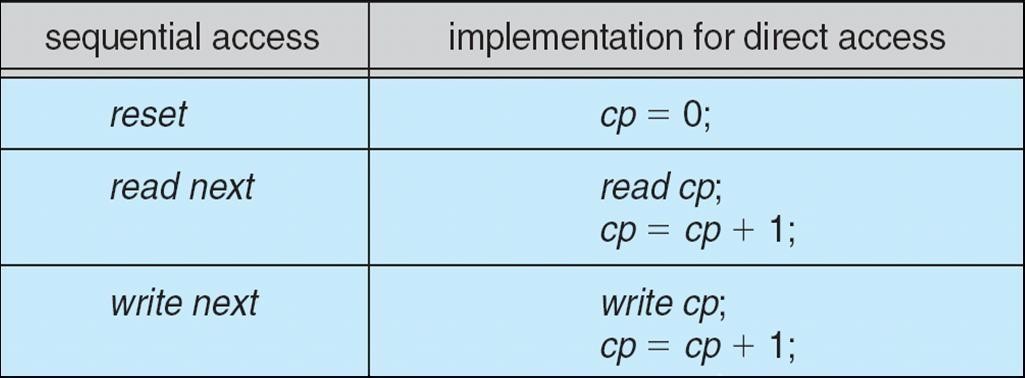
* read n
* write n
* position to n
* read next
* write next
* rewrite n

*n* = relative block number

# Sequential-access File

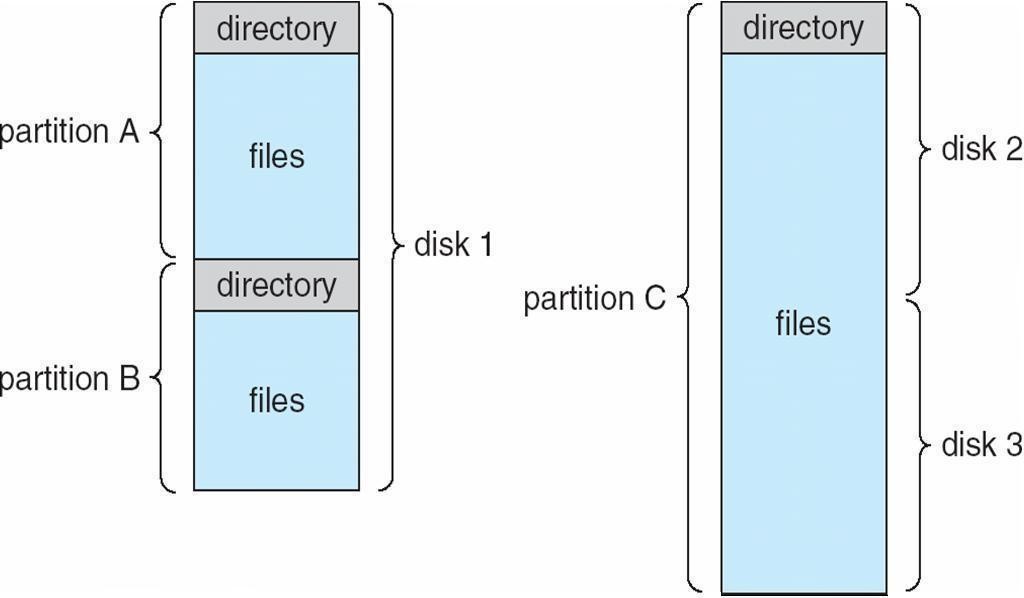


**Simulation of Sequential Access on Direct-access File**

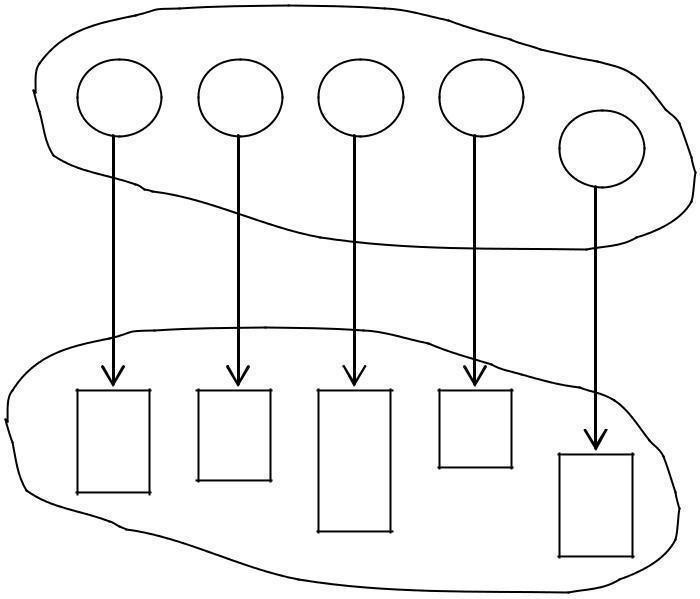


# Example of Index and Relative Files

**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



F

F

F

F

F

Files

# Disk Structure

* Disk can be subdivided into partitions
* Disks or partitions can be RAID protected against failure
* Disk or partition can be used raw – without a file system, or formatted with a file system
* Partitions also known as minidisks, slices
* Entity containing file system known as a volume
* Each volume containing file system also tracks that file system’s info in device directory or volume table of

contents

* As well as general-purpose file systems there are many special-purpose file systems, frequently all within the same operating system or computer

# A Typical File-system Organization

**Operations Performed on Directory**

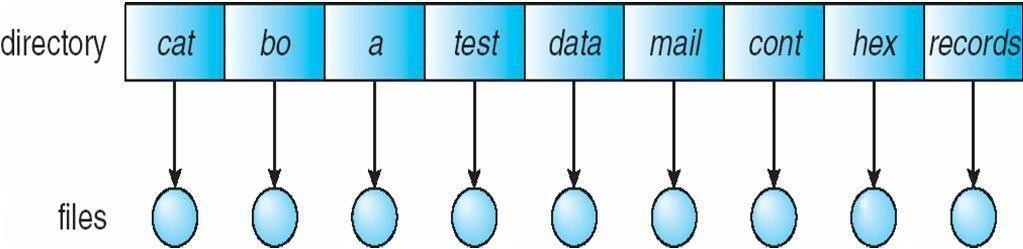
* Search for a file
* Create a file
* Delete a file
* List a directory
* Rename a file
* Traverse the file system

# 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

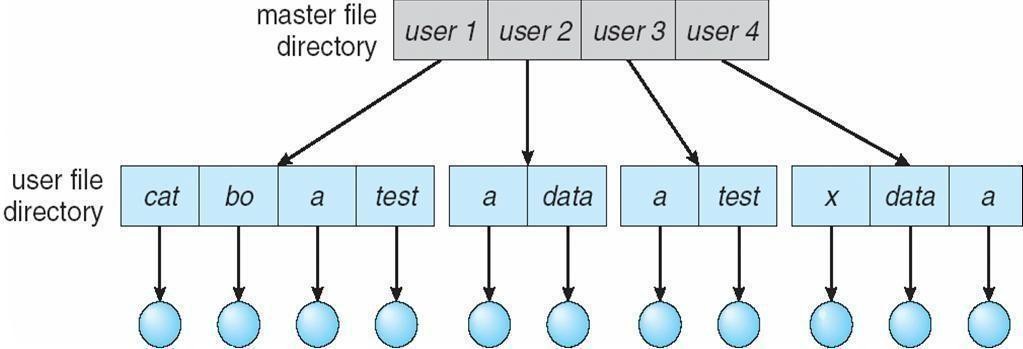
* A single directory for all users



* Naming problem
* Grouping problem

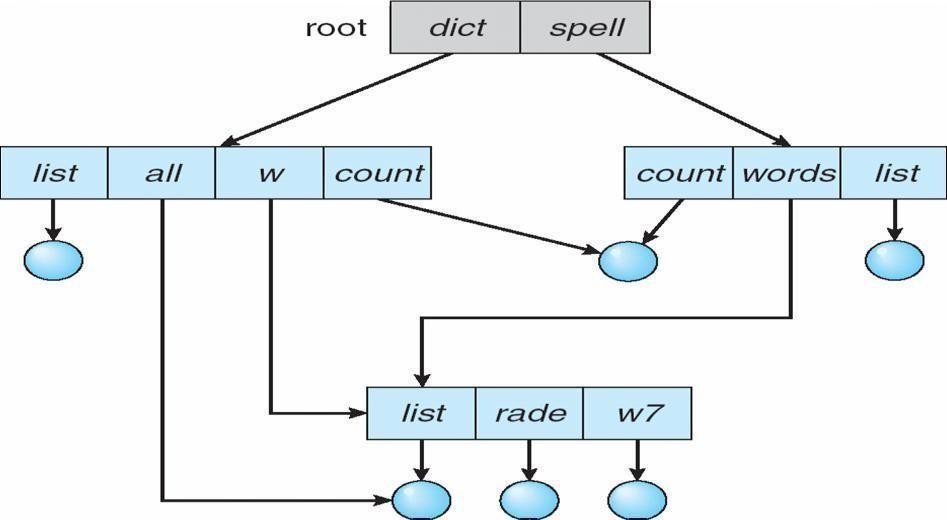
# Two-Level Directory

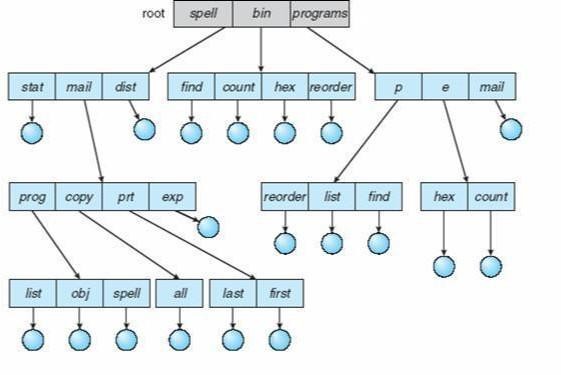
* Separate directory for each user



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

# Tree-Structured Directories





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

mail

* Deleting ―mail‖ Þ deleting the entire subtree rooted by ―mail‖

prog

copy

pr

ex

coun

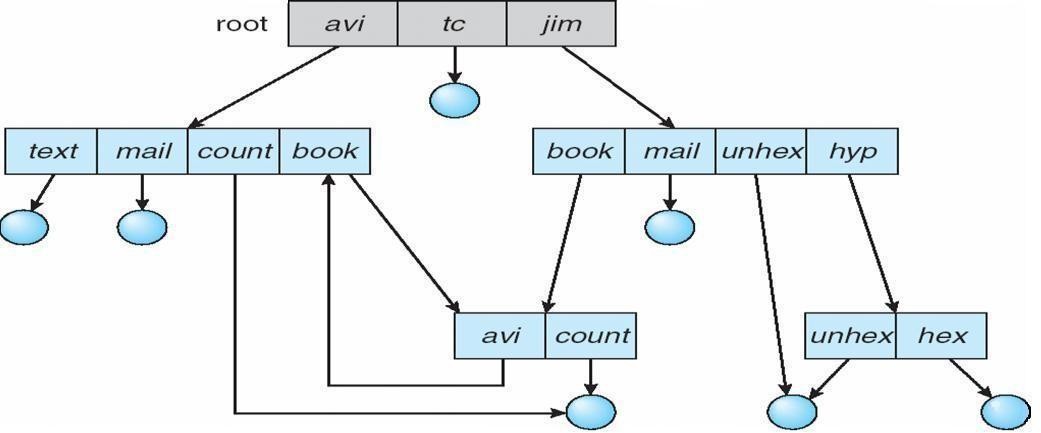
# Acyclic-Graph Directories

* Have shared subdirectories and files

Two different names (aliasing)If *dict* deletes *list* Þ 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

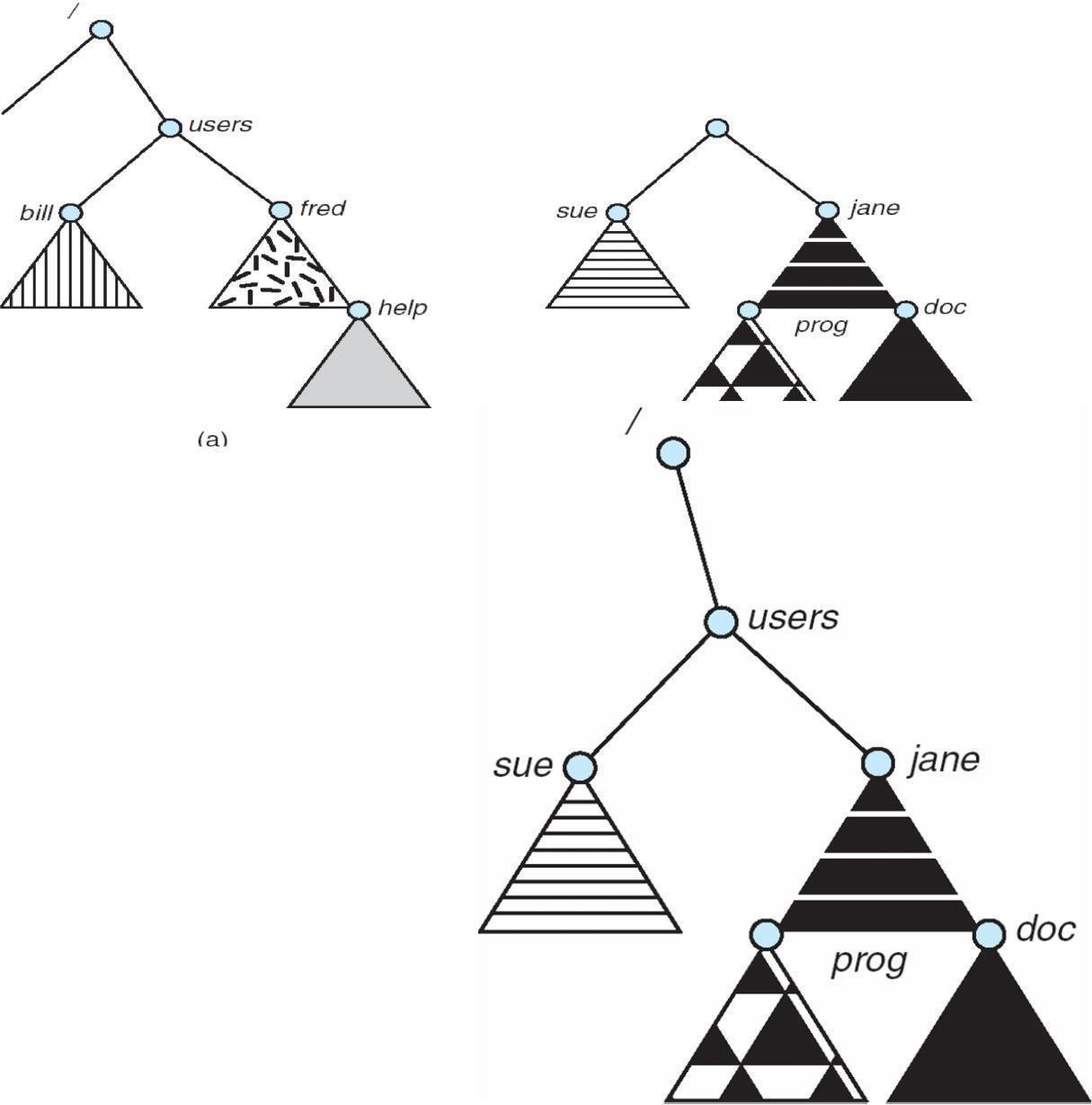


* How do we guarantee no cycles?
* Allow only links to file not subdirectories
* Garbage collection
* Every time a new link is added use a cycle detection algorithm to determine whether it is OK

# 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
* **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 whomTypes of access

# Read

* **Access Lists and Groups**
* Mode of access: read, write, execute
* Three classes of users
* RWX

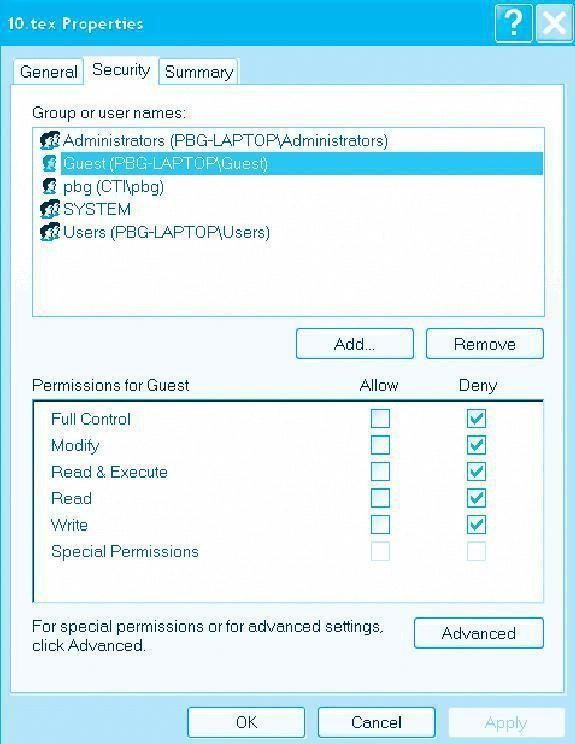
|  |  |  |  |
| --- | --- | --- | --- |
| a) **owner access** | 7 | Þ | 1 1 1 |
| RWX |  |  |  |
| b) **group access** | 6 | Þ | 1 1 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| RWX |  |  |  |
| c) **public access** | 1 | Þ | 0 0 1 |

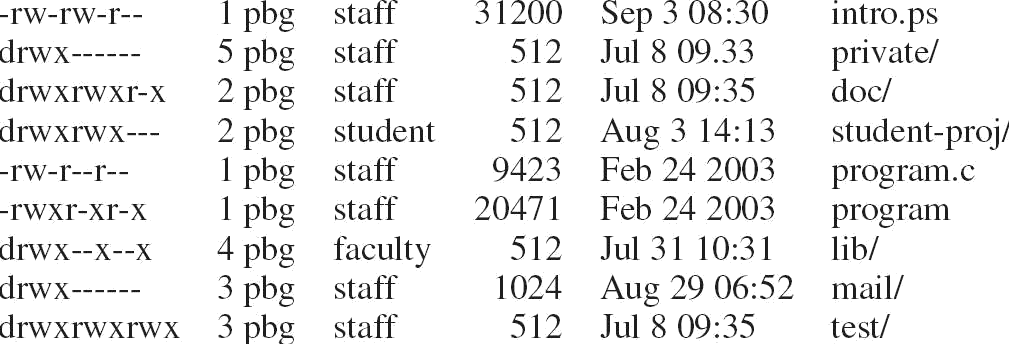
* Ask manager to create a group (unique name), say G, and add some users to the group. For a particular file (say *game*) or subdirectory, define an appropriate access.

|  |  |  |
| --- | --- | --- |
| owne | grou | publi |
| chmo | 76 | gam |
| Attach a group to a file | chgrp | G game |

# Windows XP Access-control List Management



**A Sample UNIX Directory Listing**



# Mass-Storage Systems

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

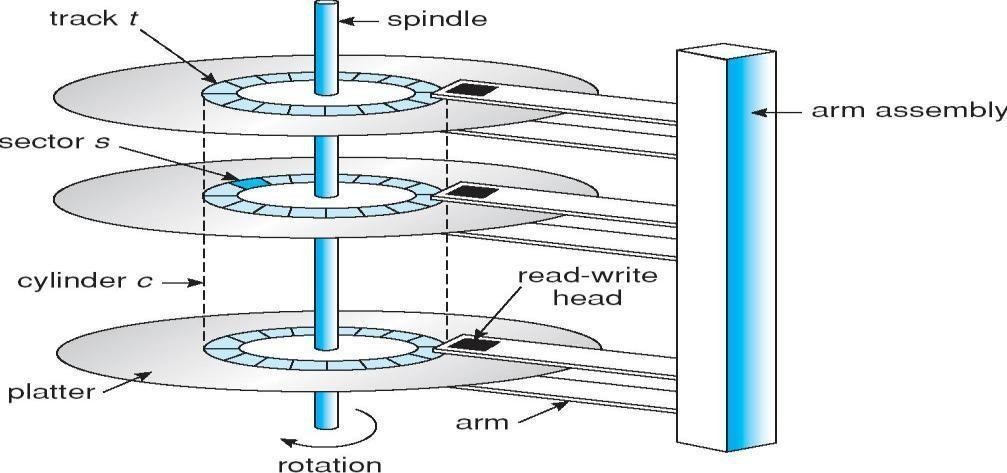
Disks can be removable

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



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

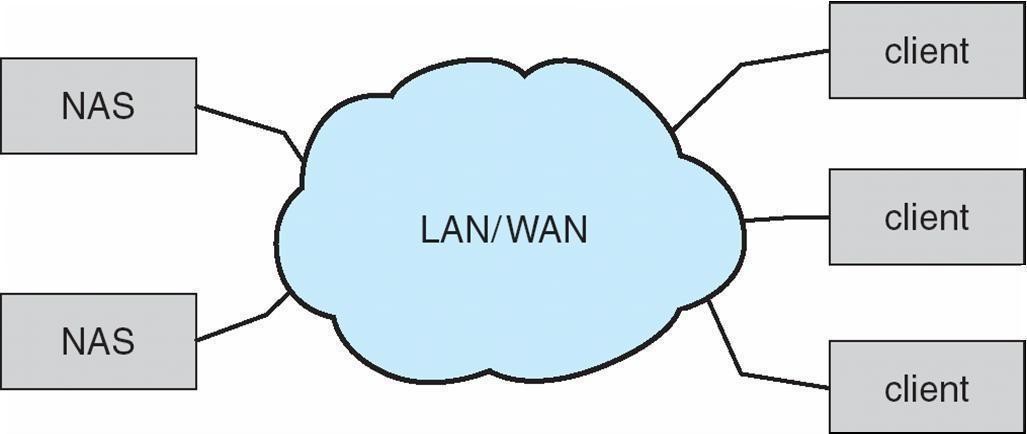
Can be arbitrated loop (FC-AL) of 126 devices

# Network-Attached Storage

Network-attached storage (NAS) is storage made available over a network rather than over a local connection (such as a bus)

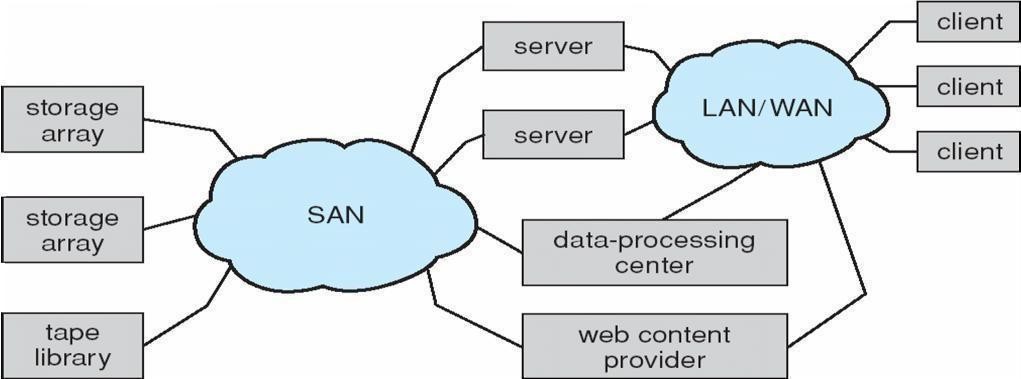
NFS and CIFS are common protocols

Implemented via remote procedure calls (RPCs) between host and storage New SCSI 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 » 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

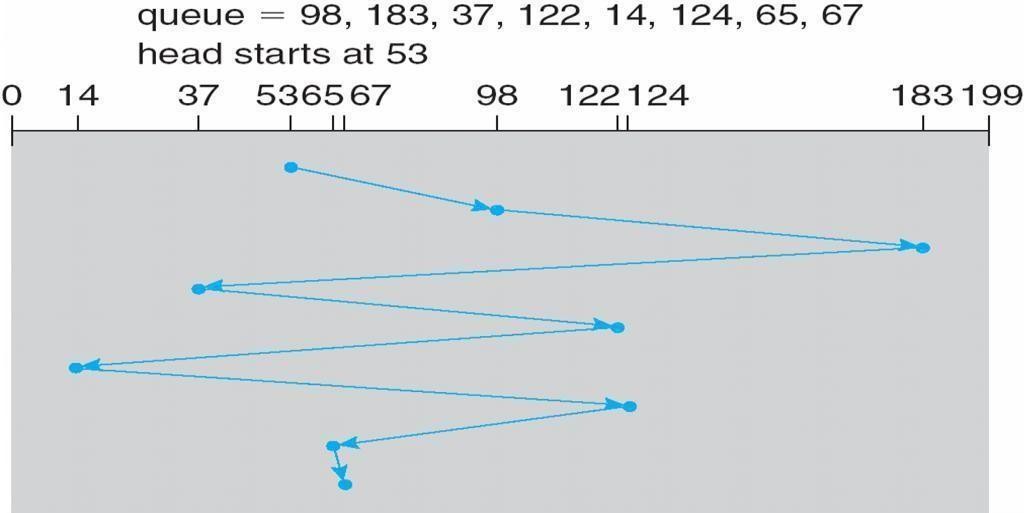
Several algorithms exist to schedule the servicing of disk I/O requests .We illustrate them with a request queue (0-199)

# FCFS

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

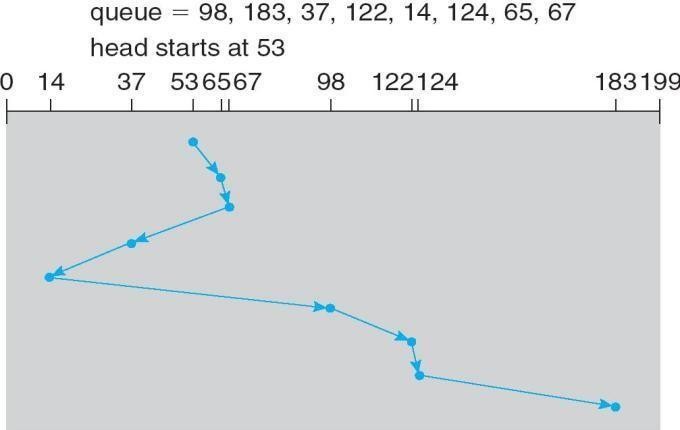
Head pointer 53

Illustration shows total head movement of 640 cylinders



# SSTF

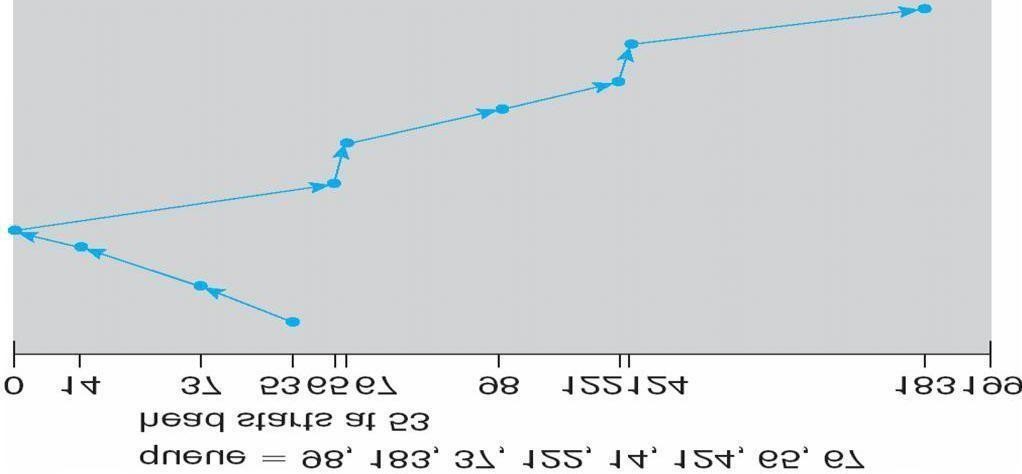
Selects the request with the minimum seek time from the current head position SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests Illustration shows total head movement of 236 cylinders



# SCAN

The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues. SCAN algorithm Sometimes called the elevator algorithm

Illustration shows total head movement of 208 cylinders



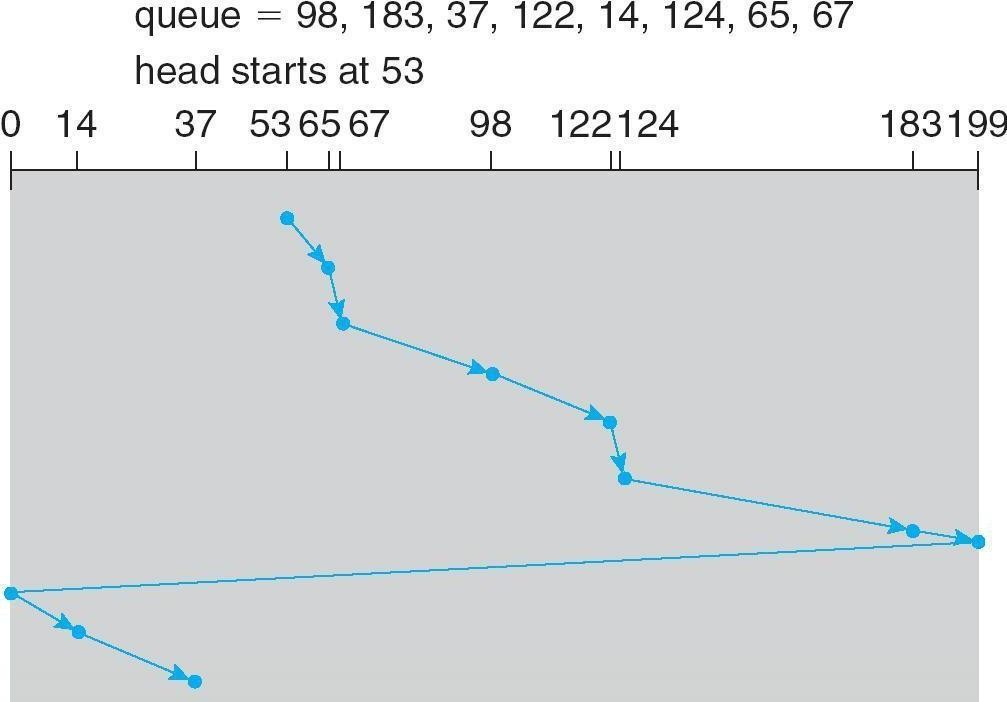
# C-SCAN

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

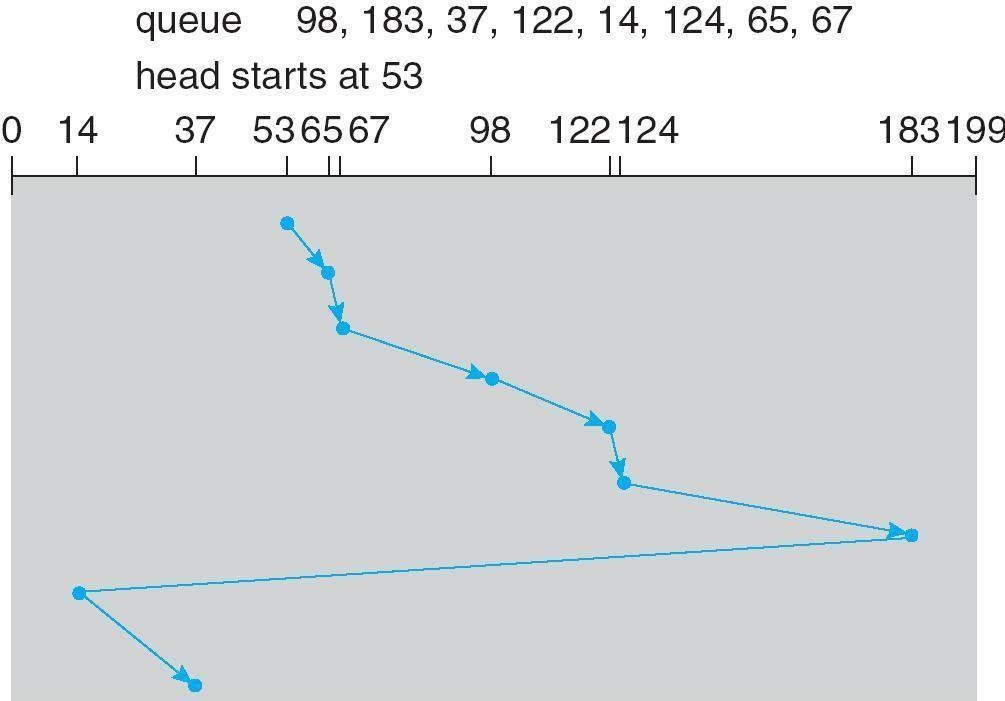
Treats the cylinders as a circular list that wraps around from the last cylinder to the first one



# C-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



# 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‖

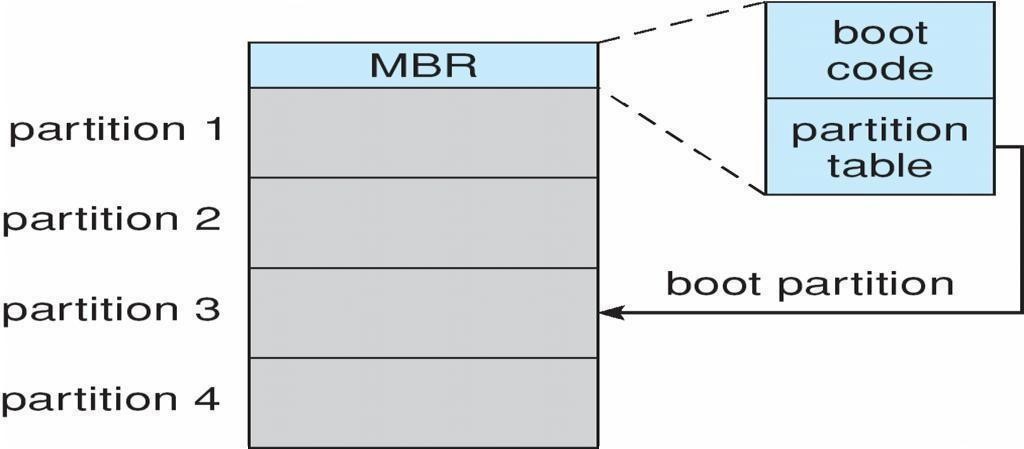
To increase efficiency most file systems group blocks into clusters Disk I/O done in blocks

File I/O done in clusters 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**

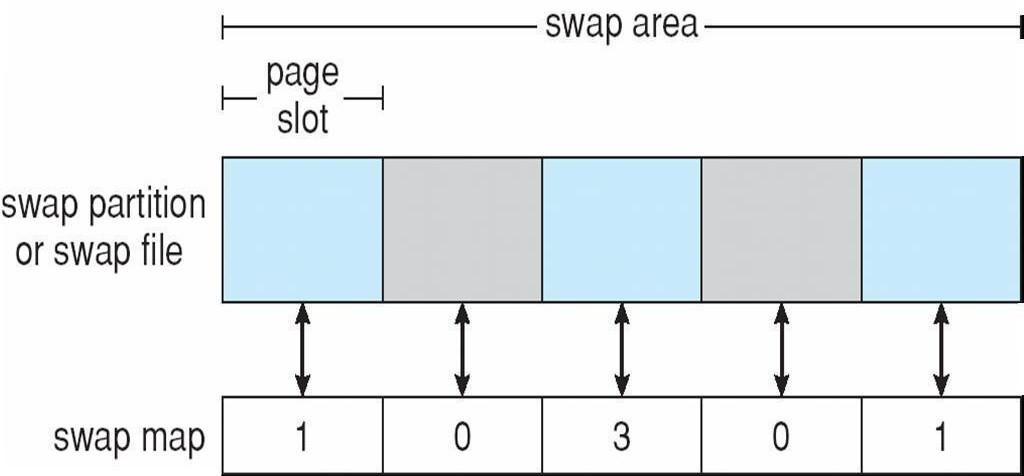
Swap-space — Virtual memory uses disk space as an extension of main memory

Swap-space can be carved out of the normal file system, or, more commonly, it can be in 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 Increases the mean time to failure Frequently combined with NVRAM to improve write performance

RAID is arranged into six different levels

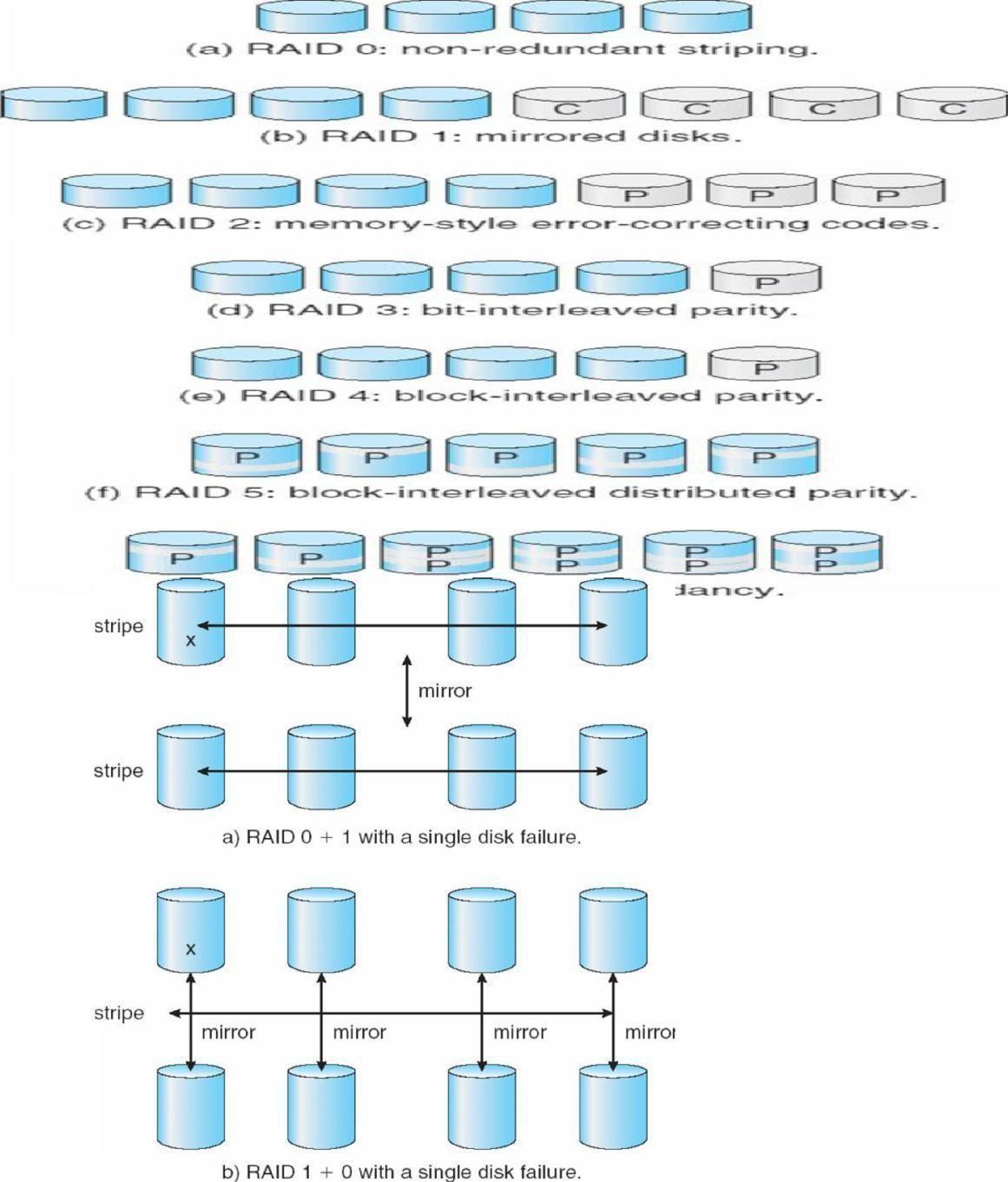
Several improvements in disk-use techniques involve the use of multiple disks working cooperatively Disk striping uses a group of disks as one storage unit RAID schemes improve performance and improve the reliability of the storage system by storing redundant data Mirroring or shadowing (RAID 1) keeps duplicate of each disk

Striped mirrors (RAID 1+0) or mirrored stripes (RAID 0+1) provides high performance and high reliability Block interleaved parity (RAID 4, 5, 6) uses much less redundancy

RAID within a storage array can still fail if the array fails, so automatic replication of the data between arrays is

common

Frequently, a small number of hot-spare disks are left unallocated, automatically replacing a failed disk and having data rebuilt onto them



**+ 1) and (1 + 0)**

# RAID (0

**Extensions**

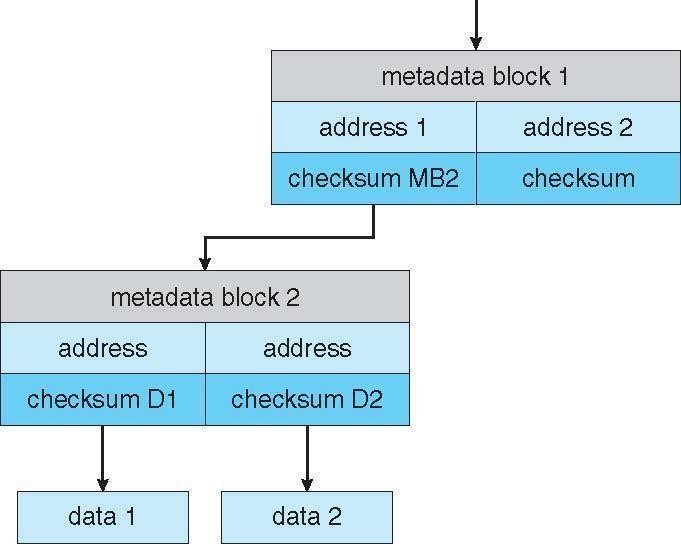
RAID alone does not prevent or detect data corruption or other errors, just disk failures Solaris ZFS adds checksums of all data and metadata

Checksums kept with pointer to object, to detect if object is the right one and whether it changed Can detect and correct data and metadata corruption

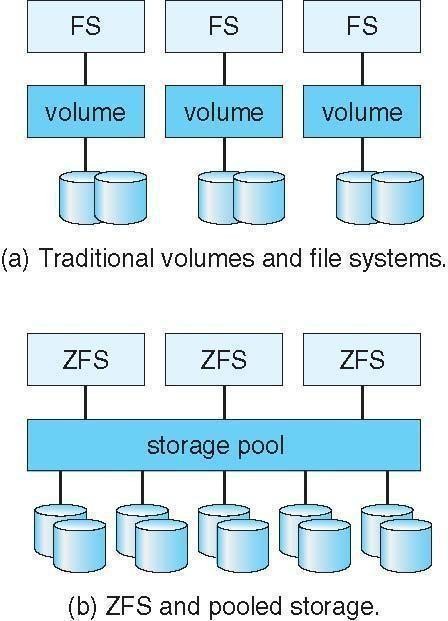
ZFS also removes volumes, partitions. Disks allocated in pools

File systems with a pool share that pool, use and release space like ―malloc‖ and ―free‖ memory allocate / release calls

# ZFS Checksums All Metadata and Data



**Traditional and Pooled Storage**



# 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

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 Support

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 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

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

# UNIT – V DEADLOCKS

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 disk drives

*P*1 and *P*2 each hold one disk drive and each needs another one Example

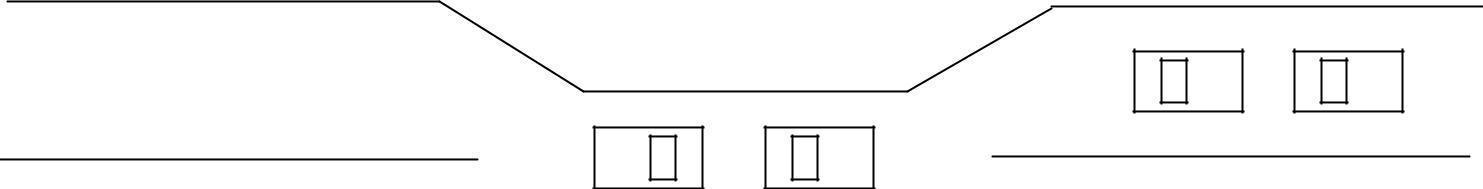
semaphores *A* and *B*, initialized to 1

*P*0 *P1*

wait (A); wait(B)

wait (B); wait(A)

# 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
* Note – Most OSes do not prevent or deal with deadlocks

# System Model

* Resource types *R*1, *R*2, . . ., *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, …, *P*0} 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, …, *Pn*–1 is waiting for a resource that is held by

*P*n, and *P*0 is waiting for a resource that is held by *P*0.

# Resource-Allocation Graph

A set of vertices *V* and a set of edges

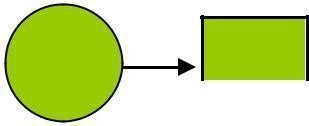
*E* V is partitioned into two types:

*P* = {*P*1, *P*2, …, *Pn*}, the set consisting of all the processes in the system *R* = {*R*1, *R*2, …, *Rm*}, the set consisting of all resource types in the system request edge – directed edge *P*1 ® *Rj* assignment edge – directed edge *Rj* ® *Pi*

Process

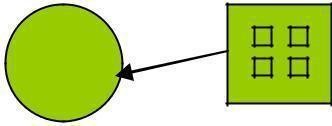
Resource Type with 4 instances

*Pi* requests instance of *Rj*n



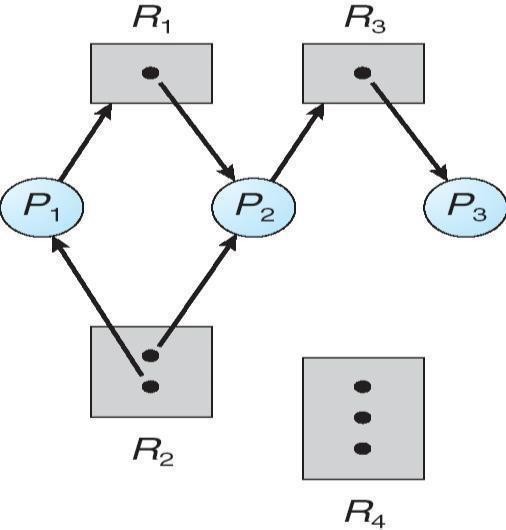
*P*

*i*

*Rj*

|  |  |
| --- | --- |
| *Pi* is holding an instance of *Rj* | *Pi* |

# Example of a Resource Allocation Graph



*Rj*

**Resource Allocation Graph With A Deadlock**

# Graph With A Cycle But No Deadlock

**Basic Facts**

If graph contains no cycles no deadlock If graph contains a cycle if only one instance per resource type, then deadlock

if several instances per resource type, possibility of deadlock

# Methods for Handling Deadlocks

Ensure that the system will *never* enter a deadlock state 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 non sharable resources

**Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources

Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none

Low resource utilization; starvation possible

# No Preemption –

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

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

# Deadlock Avoidance

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

Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need

The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition

Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

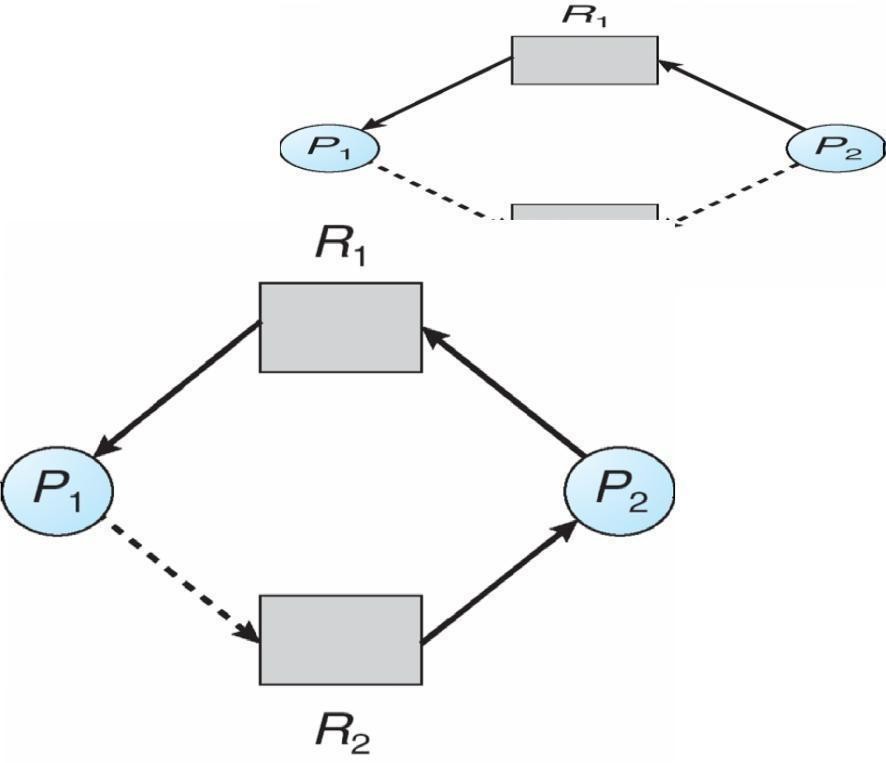
# Safe State

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state

System is in safe state if there exists a sequence <*P1, P2, …, Pn*> of ALL the processes is the systems such that for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the *Pj*, with *j* < *i*n That is:

If Pi resource needs are not immediately available, then *Pi* can wait until all *Pj* have finished When *Pj* is finished, *Pi* can obtain needed resources, execute, return allocated resources, and terminate When *Pi* terminates, *Pi* +1 can obtain its needed resources, and so on

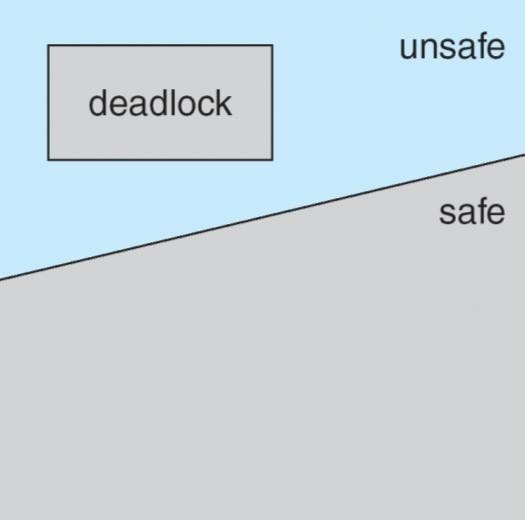
# Basic Facts



**Unsafe State In Resource-Allocation Graph**

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

# Safe, Unsafe , Deadlock State



**Avoidance algorithms**

* Single instance of a resource type
* Use a resource-allocation graph
* Multiple instances of a resource type
* Use the banker’s algorithm

# Resource-Allocation Graph Scheme

Claim edge *Pi* ® *Rj* indicated that process *Pj* may request resource *Rj*; represented by a dashed linen Claim edge converts to request edge when a process requests a resource Request edge converted to an assignment edge when the resource is allocated to the process

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

# Resource-Allocation Graph

**Resource-Allocation Graph Algorithm**

Suppose that process *Pi* requests a resource *Rj*

The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

# Banker’s Algorithm

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

# Data Structures for the Banker’s Algorithm

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

**Available***:* Vector of length *m*. If available [*j*] = *k*, there are *k* instances of resource type *Rj* available

**Max***: n x m* matrix. If *Max* [*i,j*] = *k*, then process *Pi* may request at most *k* instances of resource type *Rj* **Allocation***: n* x *m* matrix. If Allocation[*i,j*] = *k* then *Pi* is currently allocated *k* instances of *Rj***Need***: n* x *m* matrix. If *Need*[*i,j*] = *k*, then *Pi* may need *k* more instances of *Rj* to complete its task

*Need* [*i,j]* = *Max*[*i,j*] – *Allocation* [*i,j*]

# Safety Algorithm

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

*Work* = *Available*

*Finish* [*i*] = *false* for *i* = 0, 1, …, *n-* 1

1. Find and *i* such that both:
   1. *Finish* [*i*] = *false*(b) *Needi* £ *Work*

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

1. *Work* = *Work* + *Allocationi Finish*[*i*] = *true*

go to step 2

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

# Resource-Request Algorithm for Process *Pi*

1. *Request* = request vector for process *Pi*. If *Requesti* [*j*] = *k* then process *Pi* wants *k* instances of resource type

*Rj*1. If *Requesti* £ *Needi* go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

1. If *Requesti* £ *Available*, go to step 3. Otherwise *Pi* must wait, since resources are not available
2. Pretend to allocate requested resources to *Pi* by modifying the state as follows:

*Available* = *Available* – *Request; Allocationi* = *Allocationi* + *Requesti*; *Needi* = *Needi* – *Requesti;*

*If safe Þ the resources are allocated to Pi*

*If unsafe Þ Pi must wait, and the old resource-allocation state is restored*

# Example of Banker’s Algorithm

n5 processes *P*0 through *P*4;

3 resource types:

*C*

2

*A* (10 instances), *B* (5instances), and *C* (7 instances) Snapshot at time *T*0:

*Allocation Max Available A B C A B C AB P*0 0 1 0 7 5 3 *P*1 2 0 0 3 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

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 < *P*1, *P*3, *P*4, *P*2, *P*0> satisfies safety criteria

# Example: *P*1 Request (1,0,2)

Check that Request £ Available (that is, (1,0,2) £ (3,3,2) Þ true

|  |  |  |  |
| --- | --- | --- | --- |
|  | *Allocation* | *Need* | *Available* |
|  | *ABC* | *ABC* | *ABC* |
| *P0* | *010* | *743* | *230* |
| *P1* | *302* | *020* |  |
| *P2* | *301* | *600* |  |
| *P3* | *211* | *011* |  |
| *P4* | *002* | *431* |  |

Executing safety algorithm shows that sequence < *P*1, *P*3, *P*4, *P*0, *P*2> satisfies safety requirement Can request for (3,3,0) by *P*4 be granted? Can request for (0,2,0) by *P*0 be granted?

# Deadlock Detection

Allow system to enter deadlock state Detection algorithm Recovery scheme

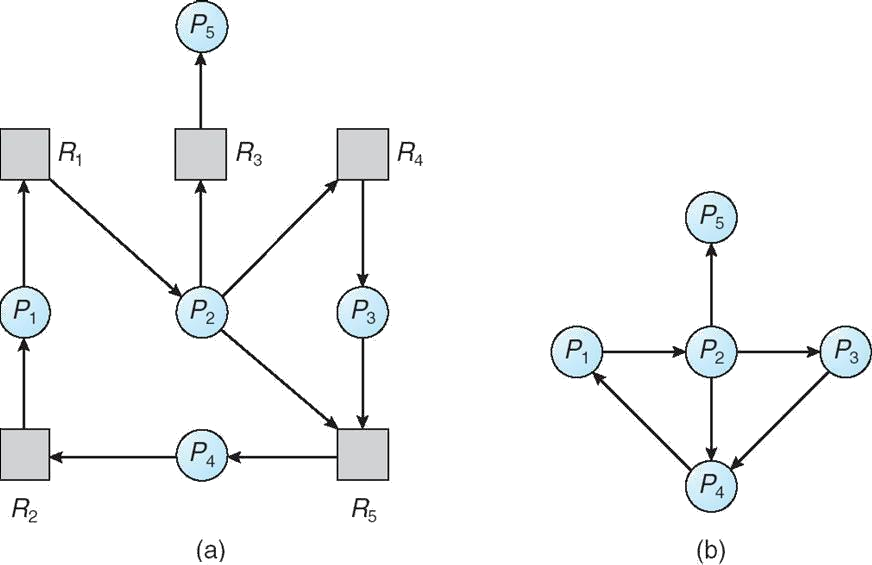
# Single Instance of Each Resource Type

Maintain *wait-for* graph Nodes are processes

*Pi* ® *Pj* if *Pi* is waiting for *Pj* Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

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

# Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

# Several Instances of a Resource Type

**Available***:* A vector of length *m* indicates the number of available resources of each type. **Allocation***:* An *n* x *m* matrix defines the number of resources of each type currently allocated to each process. **Request***:* An *n* x *m* matrix indicates the current request of each process. If *Request* [*ij*] = *k*, then process *Pi* is requesting *k* more instances of resource type. *Rj*.

# Detection Algorithm

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

(a) *Work* = *Available*(b) For *i* = 1,2, …, *n*, if *Allocationi* ¹ 0, then

*Finish*[i] = false; otherwise, *Finish*[i] = *true*2. Find an index *i* such that both:

(a) *Finish*[*i*] == *false*(b) *Requesti* £ *Work* If no such *i* exists, go to step 4

3. *Work* = *Work* + *Allocationi Finish*[*i*] = *true*

go to step 24. If *Finish*[*i*] == false, for some *i*, 1 £ *i* £ *n*, then the system is in deadlock state. Moreover, if

*Finish*[*i*] == *false*, then *Pi* is deadlocked

# Algorithm requires an order of O(*m* x *n*2) operations to detect whether the system is in deadlocked state Example of Detection Algorithm

nFive processes *P*0 through *P*4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances) nSnapshot at time *T*0:

|  |  |  |  |
| --- | --- | --- | --- |
|  | *Allocation* | *Need* | *Available* |
|  | *ABC* | *ABC* | *ABC* |
| *P0* | *010* | *000* | *000* |
| *P1* | *200* | *202* |  |
| *P2* | *303* | *000* |  |
| *P3* | *211* | *100* |  |
| *P4* | *002* | *002* |  |

Sequence <*P*0, *P*2, *P*3, *P*1, *P*4> will result in *Finish*[*i*] = true for all *i*

*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

# I/O Systems

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

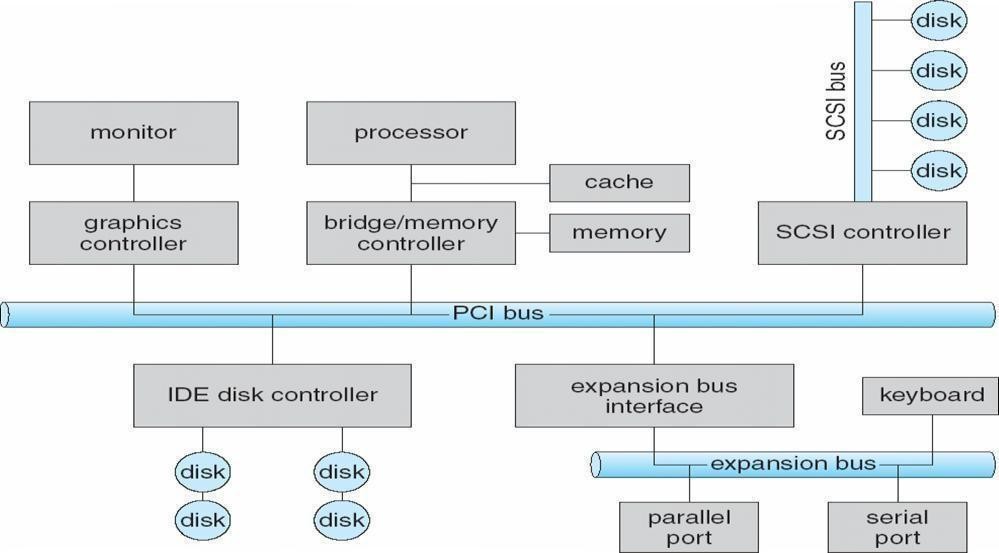
Bus (daisy chain or shared direct access) Controller (host adapter)

I/O instructions control devices

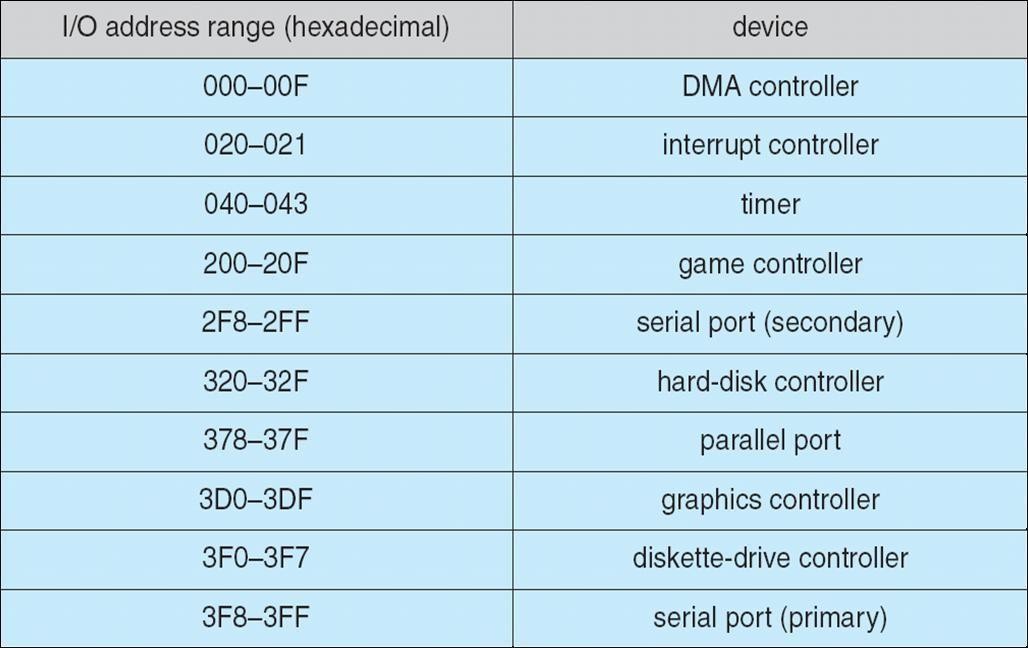
Devices have addresses, used by

Direct I/O instructions Memory-mapped I/O

# A Typical PC Bus Structure



**Device I/O Port Locations on PCs (partial)**



# Polling

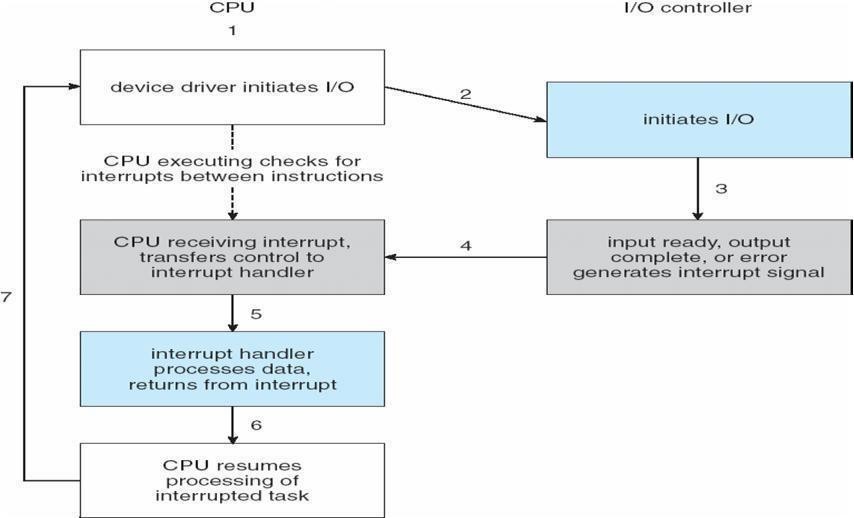
Determines state of device command-ready busy

Error Busy-wait cycle to wait for I/O from device

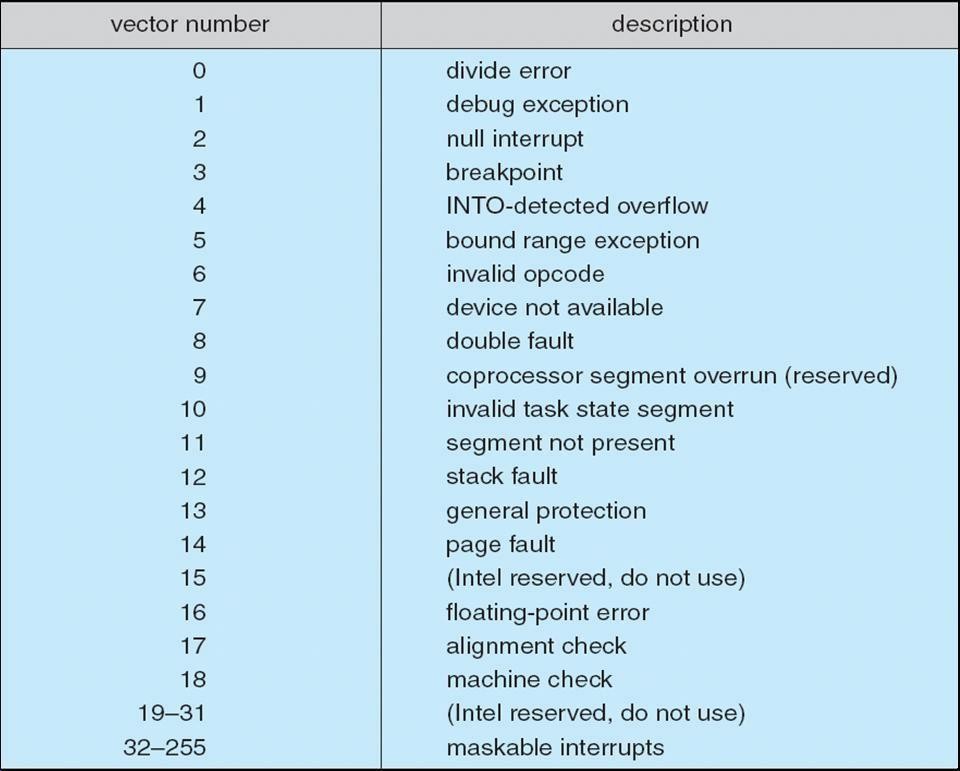
# Interrupts

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

# Interrupt-Driven I/O Cycle



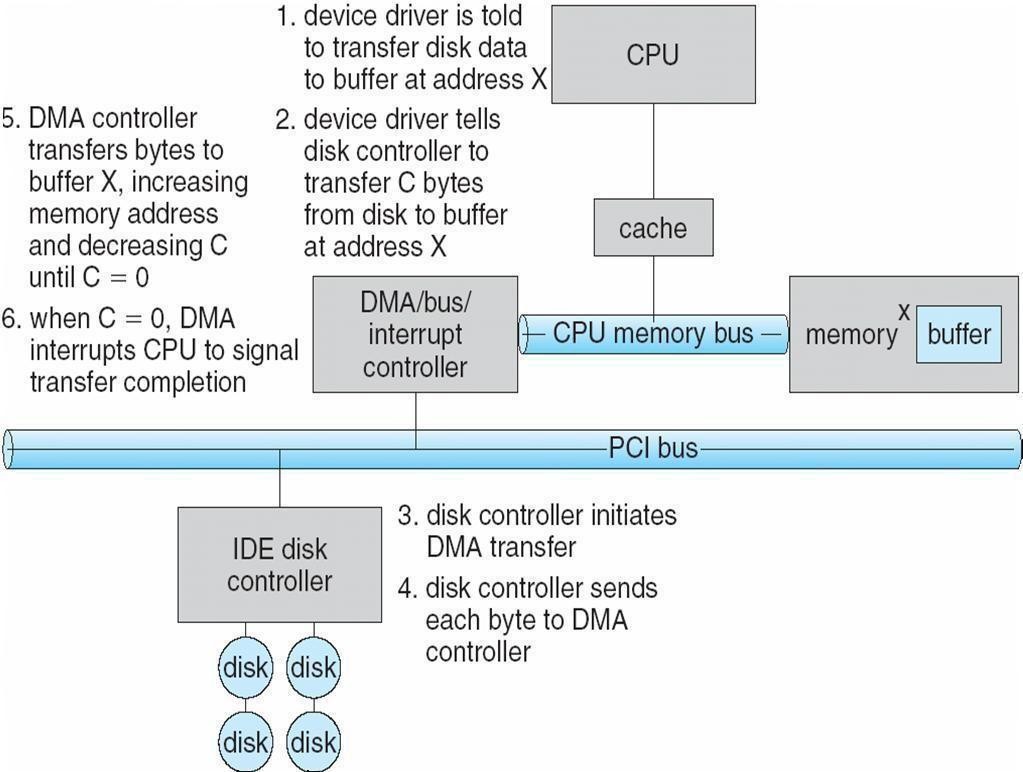
**Intel Pentium Processor Event-Vector Table**



# 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



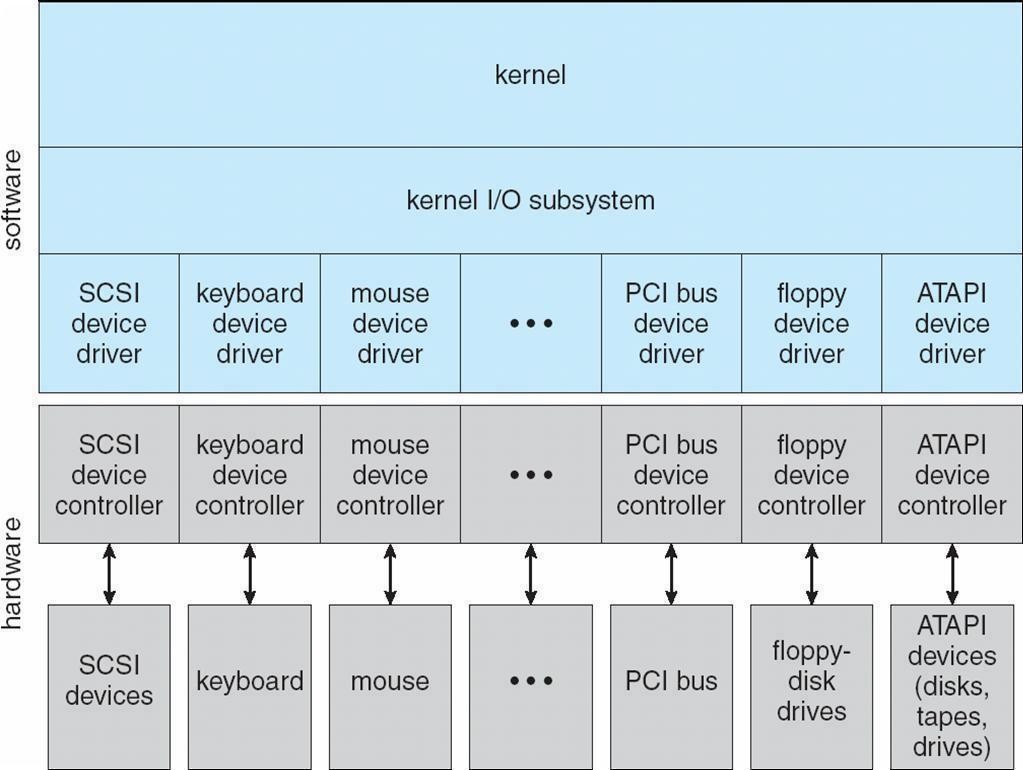
**Application I/O Interface**

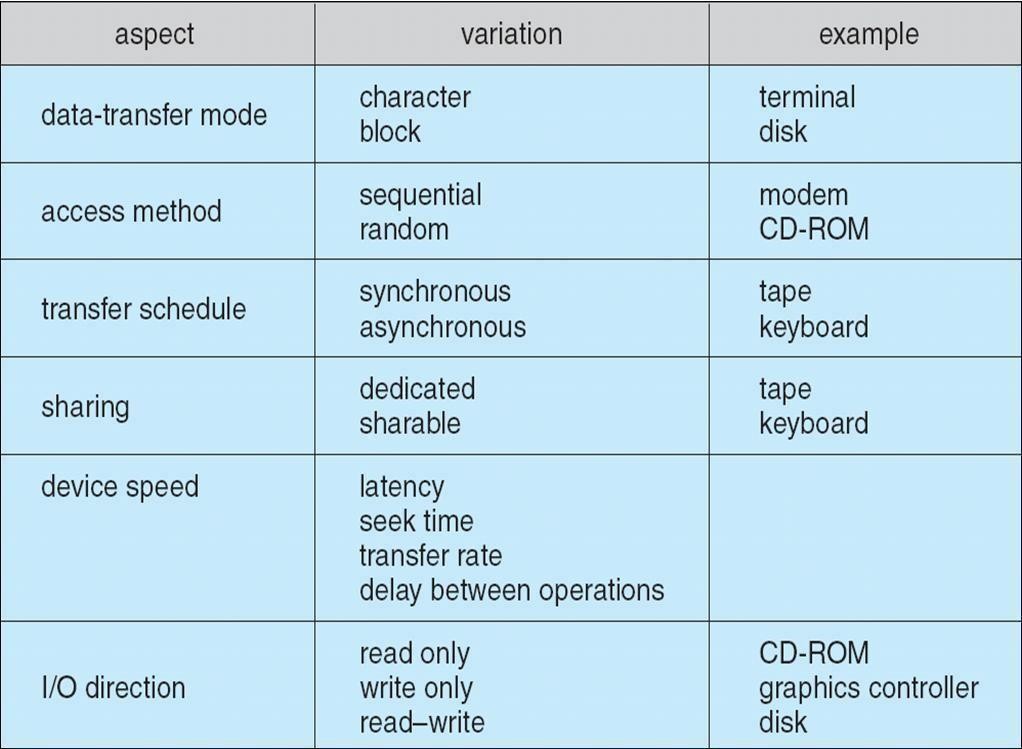
I/O system calls encapsulate device behaviors in generic classes Device-driver layer hides differences among I/O controllers from kernel Devices vary in many dimensions

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**

**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/9*x*/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 interruptsnioctl() (on UNIX) covers odd aspects of I/O such as clocks and timers

# Blocking and Non blocking I/O

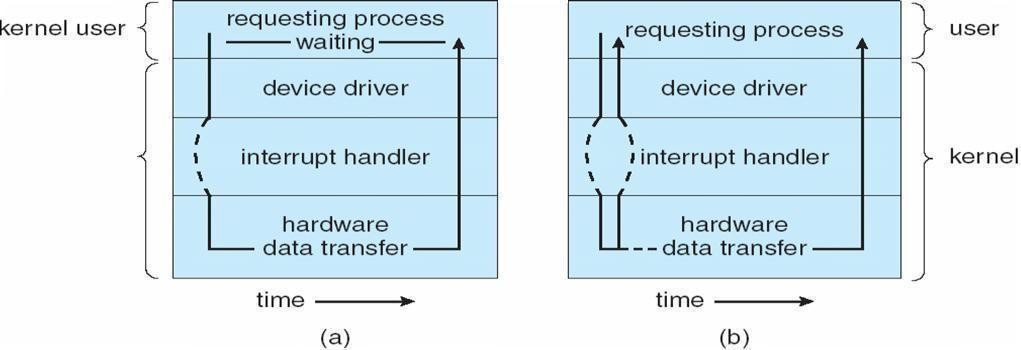
Blocking - process suspended until I/O completed Easy to use and understand

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



Synchronous Asynchronous

# Kernel I/O Subsystem

Scheduling

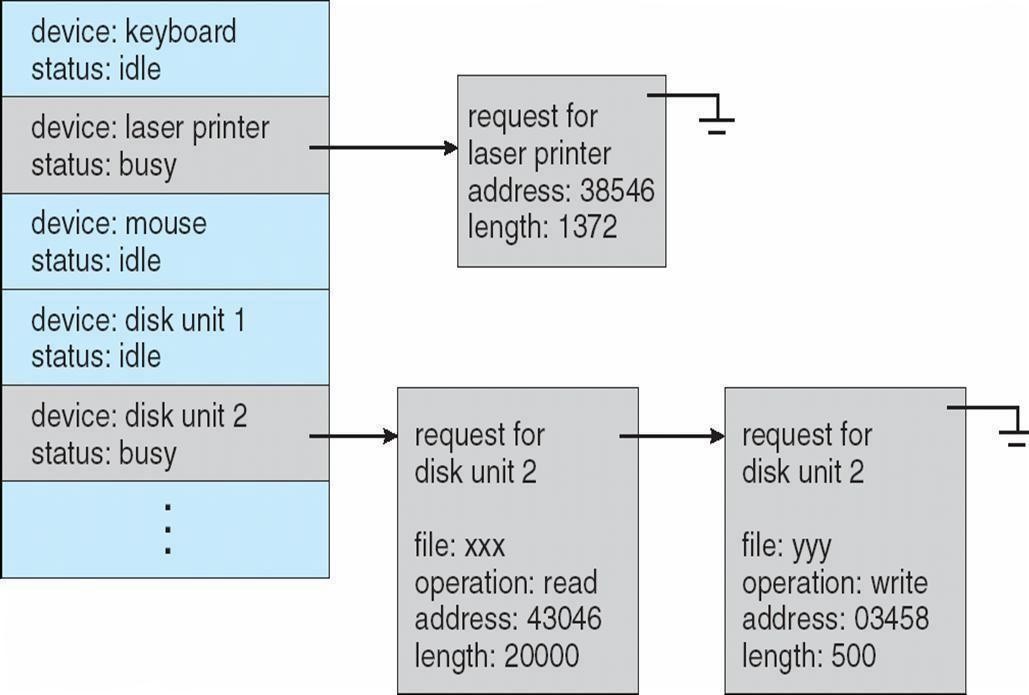
Some I/O request ordering via per-device queue

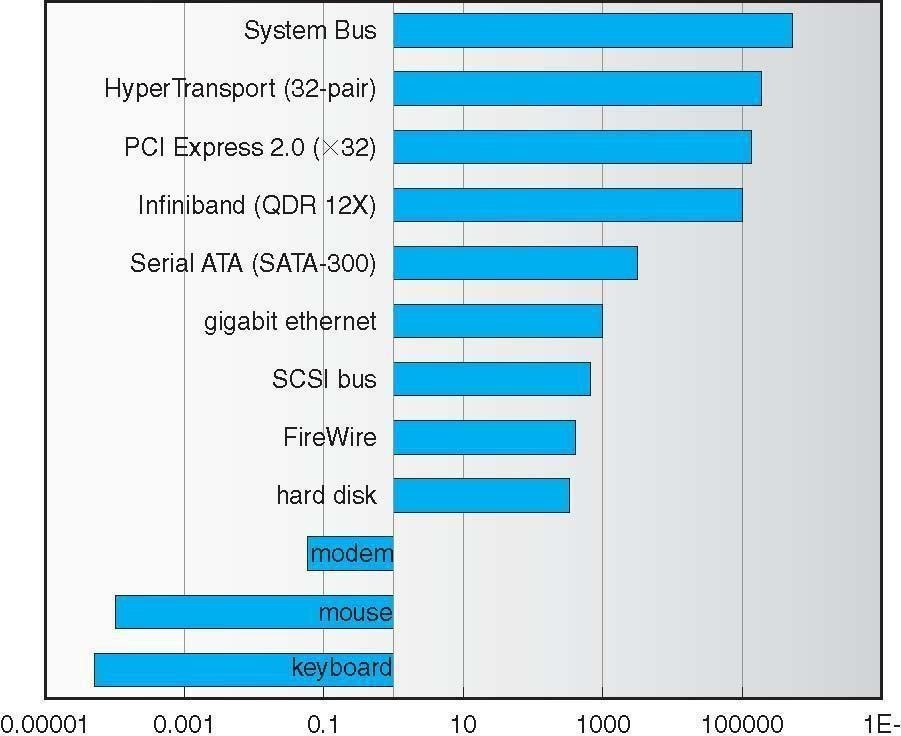
Some OSs try fairness Buffering - store data in memory while transferring between

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

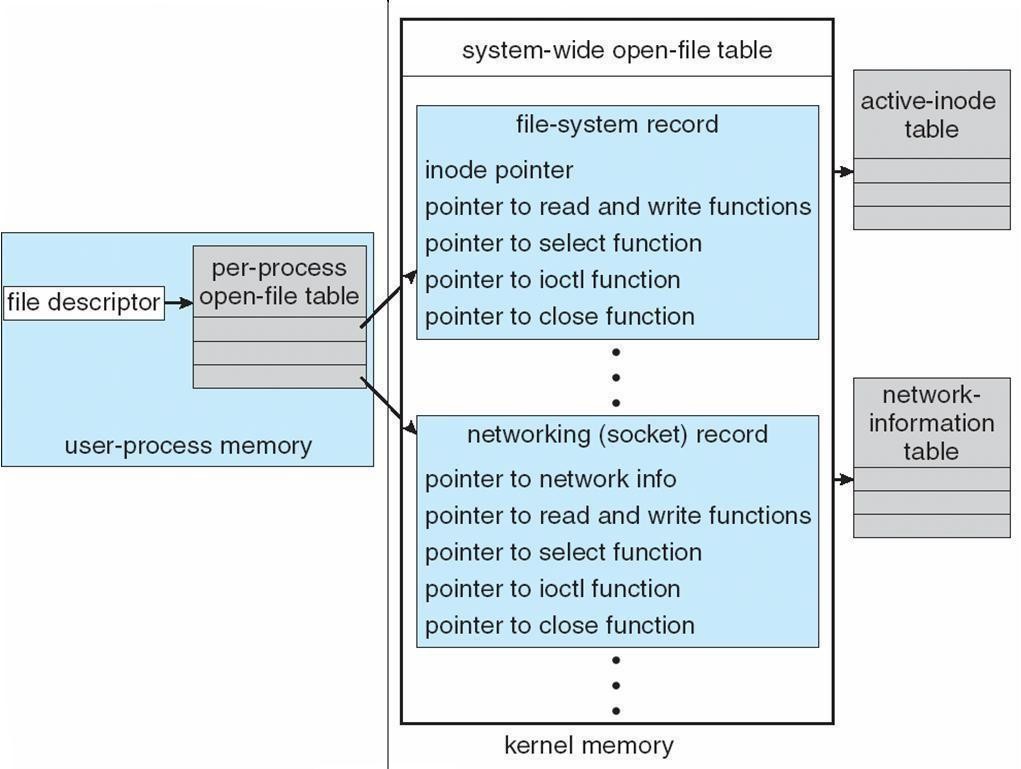
I/O must be performed via system calls Memory-mapped and I/O port memory locations must be protected too

# 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



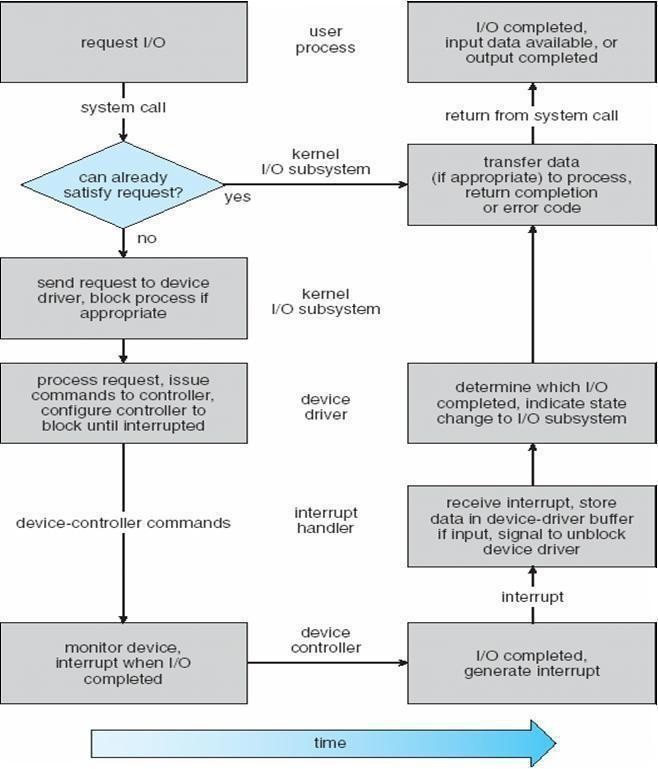
**I/O Requests to Hardware Operations**

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**

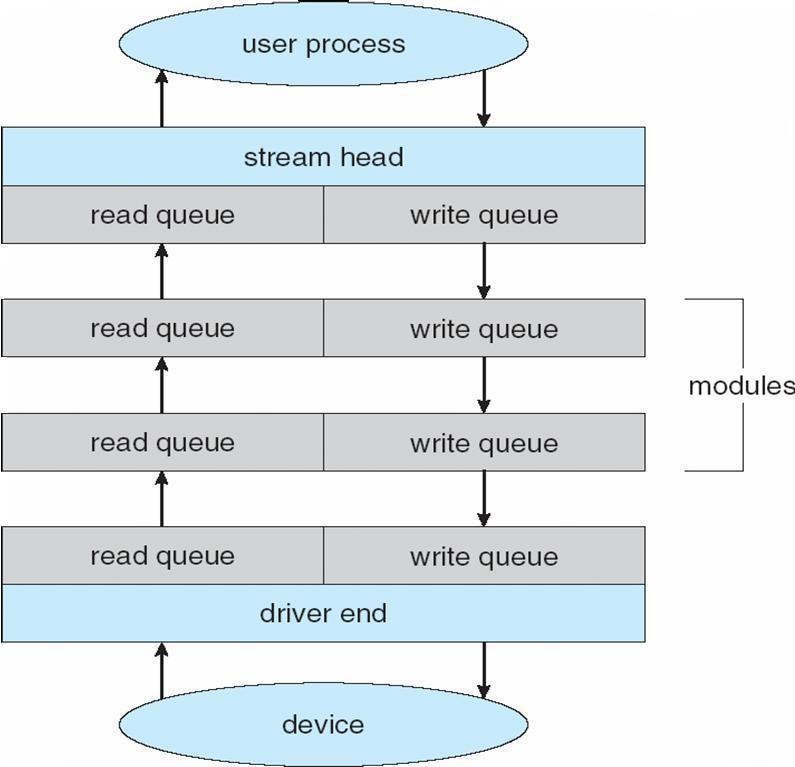
STREAM – a full-duplex communication channel between a user-level process and a device in Unix System V and beyond

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



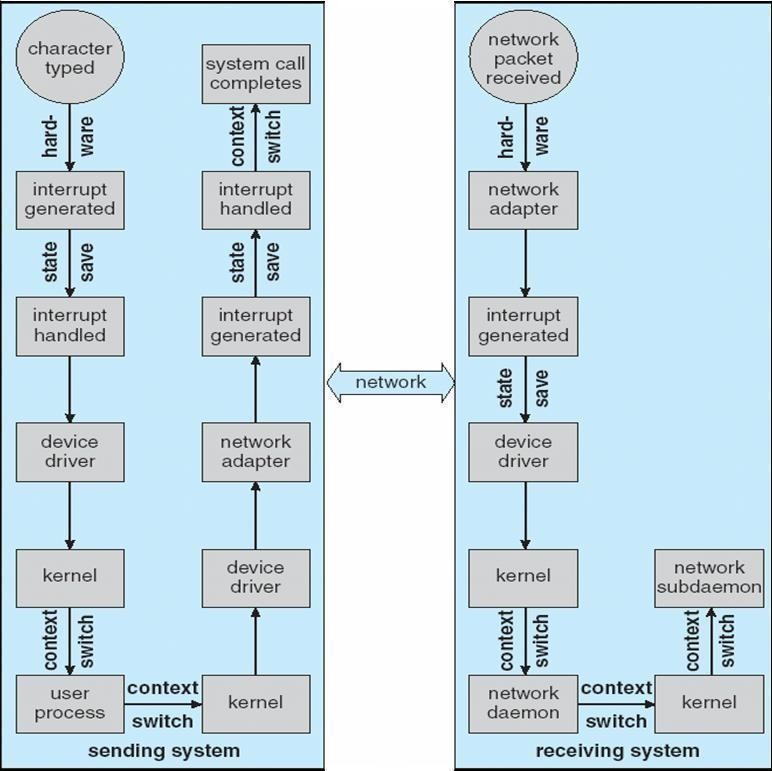
**Performance**

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



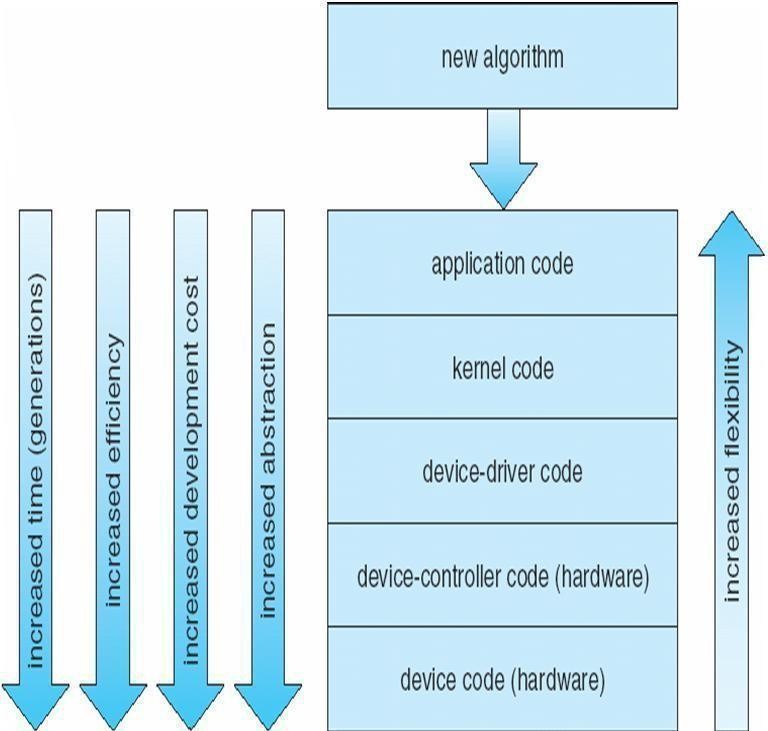
**Improving Performance**

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



**P R O T E C T I O N**

# 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 son

# Principles of Protection

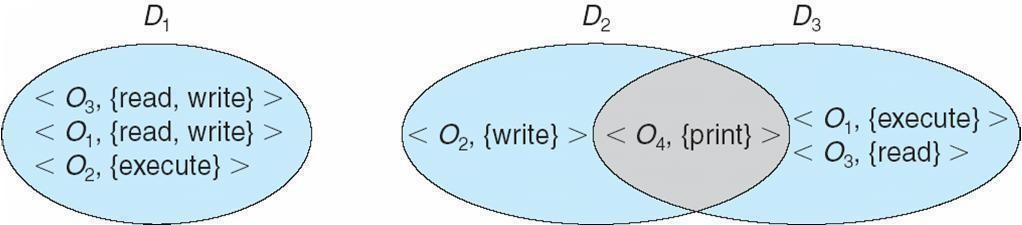
Guiding principle – principle of least privilege

Programs, users and systems should be given just enough privileges to perform their tasks

# Domain Structure

Access-right = <*object-name*, *rights-set*>

where *rights-set* is a subset of all valid operations that can be performed on the object. Domain = set of access-rights



System consists of 2 domains:

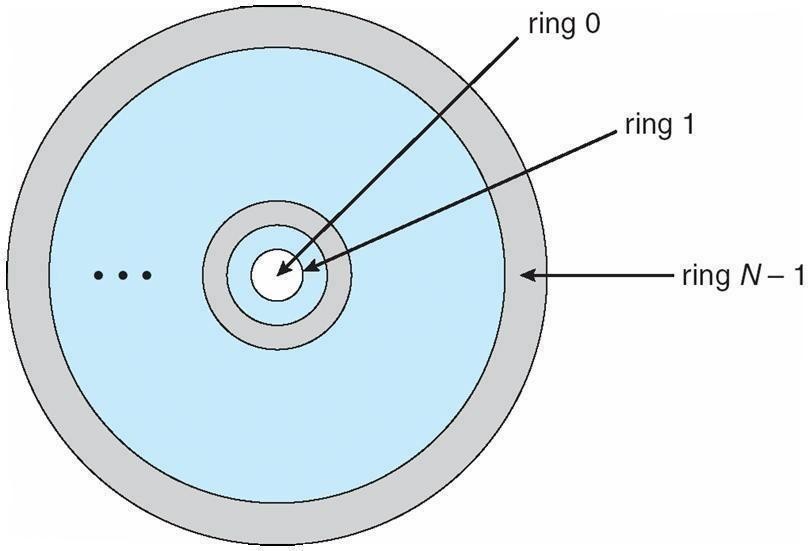
* User
* Supervisor UNIX

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

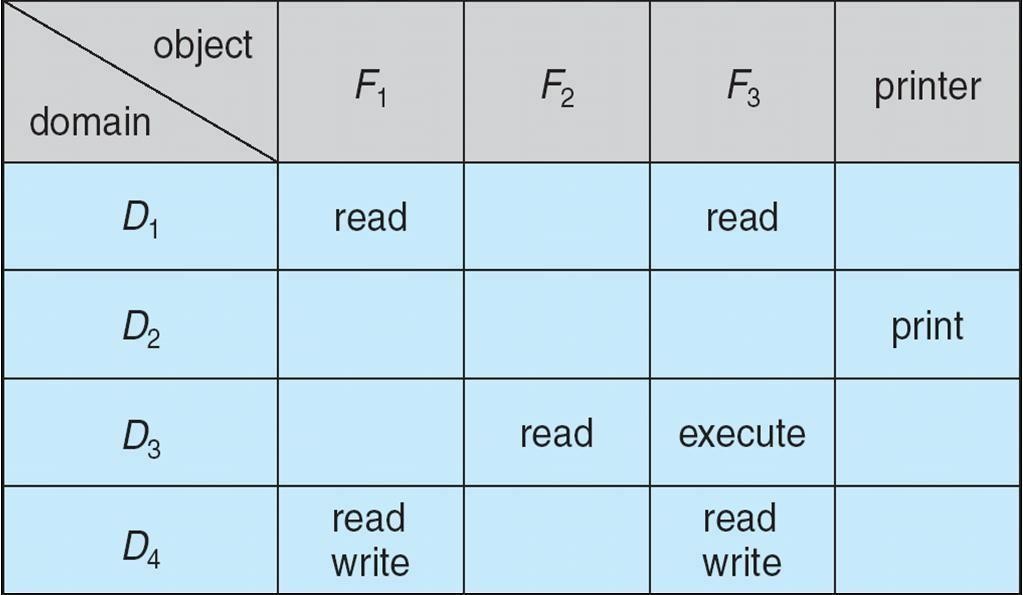
**Domain Implementation (MULTICS)** Let *Di* and *Dj* be any two domain rings If *j* < *I* Þ *Di* Í *Dj*

# Access Matrix

View protection as a matrix (*access matrix*) Rows represent domains

Columns represent objects

*Access(i, j)* is the set of operations that a process executing in Domaini can invoke on Objectj



# Use of Access Matrix

If a process in Domain *Di* tries to do ―op‖ on object *Oj*, 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 Oi*
  + *Copy op from Oi to Oj*
  + *Control – Di can modify Dj access rights*
  + *Transfer – switch from domain Di to Dj*

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.

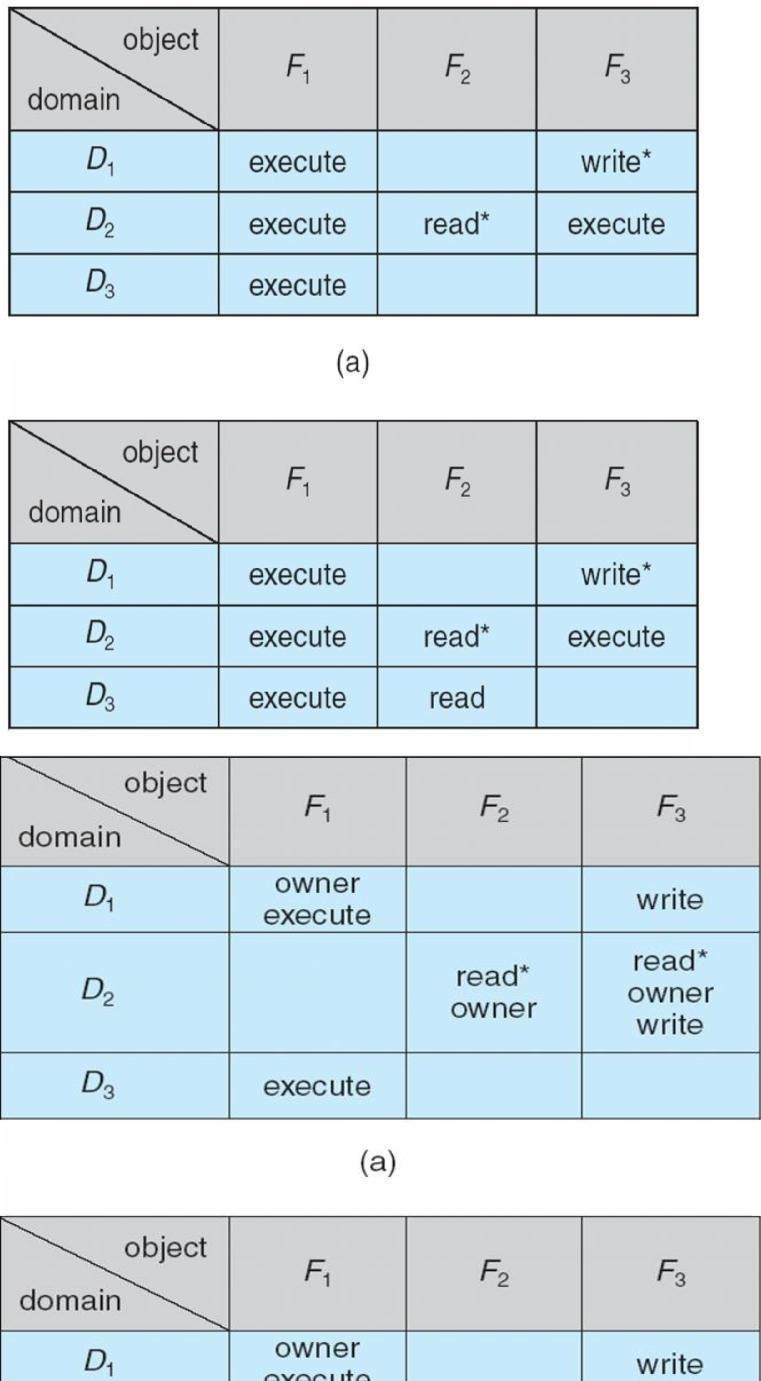
Domain 1 = Read, Write Domain 2 = Read Domain 3 = Read

M Each Row = Capability List (like a key)

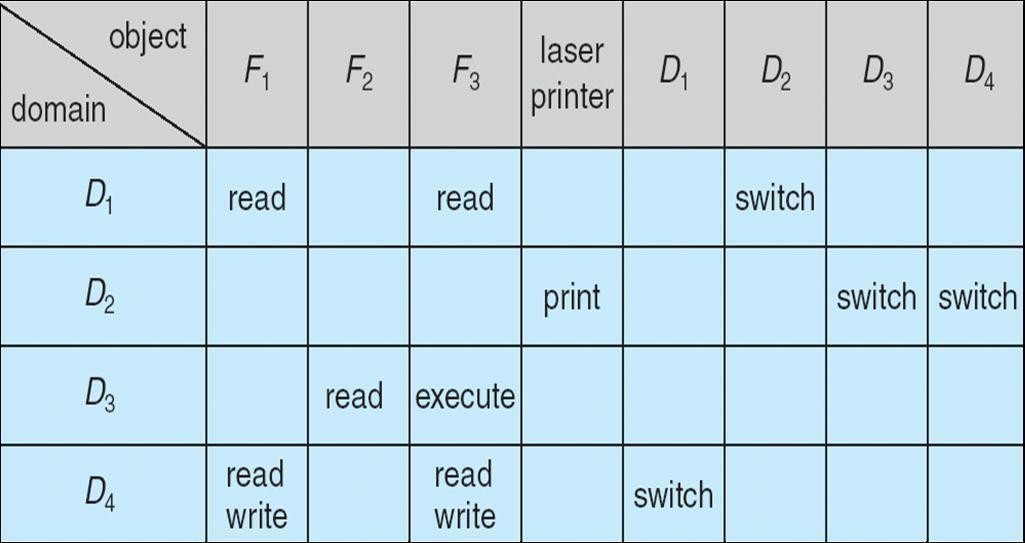
Fore each domain, what operations allowed on what objects. Object 1 – Read

Object 4 – Read, Write, Execute

Object 5 – Read, Write, Delete, Copy



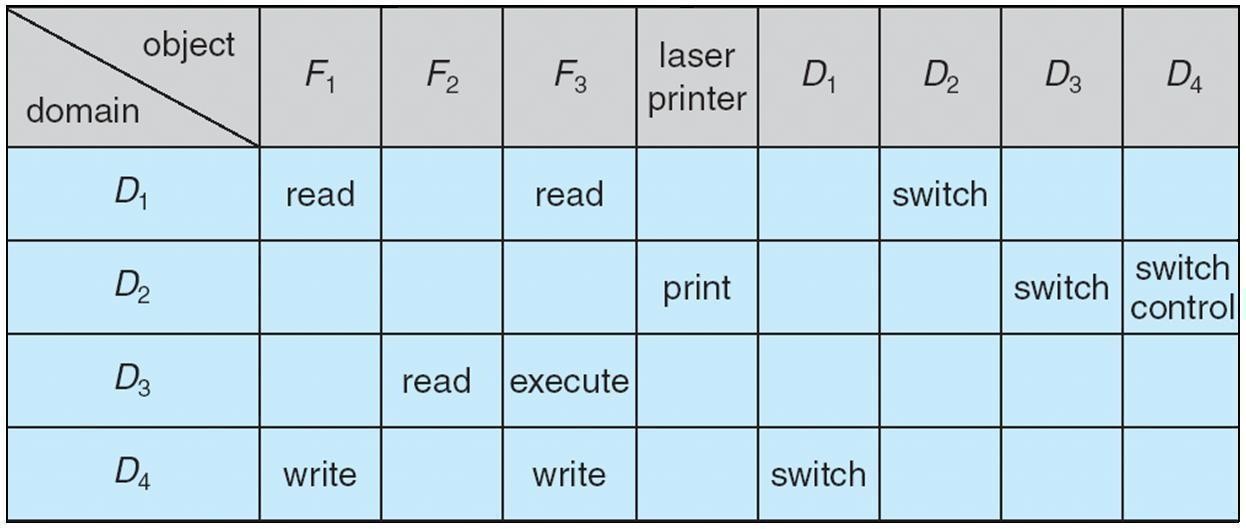
# Access Matrix of Figure A With Domains as Objects



**Access Matrix with *Copy* Rights**

# Access Matrix With *Owner* Rights

**Modified Access Matrix of Figure B**



# Access Control

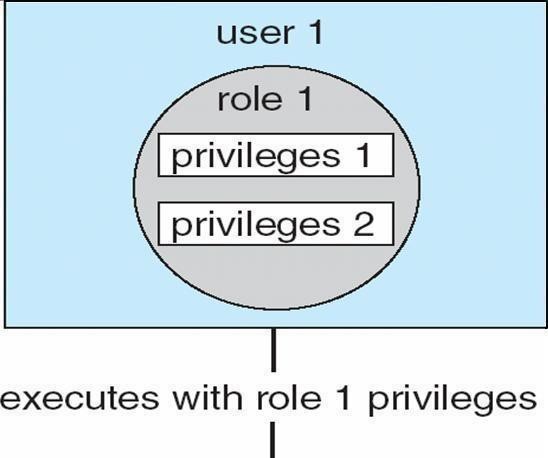
Protection can be applied to non-file resources

Solaris 10 provides role-based access control **(**RBAC**)** 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

# 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

nProtection is handled by the Java Virtual Machine (JVM)nA class is assigned a protection domain when it is loaded by the JVMnThe protection domain indicates what operations the class can (and cannot) performnIf 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

