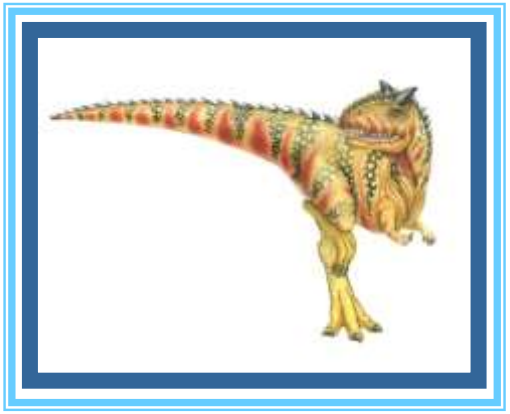
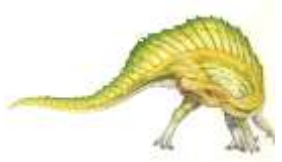
Chapter 10: File-System Interface



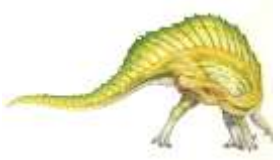
Chapter 10: File-System Interface

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



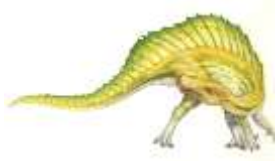
Objectives

- 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



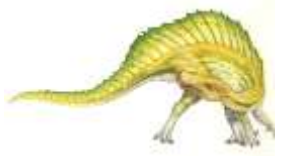
File Concept

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



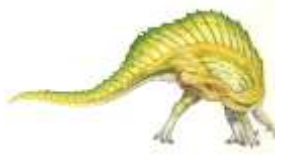
File Structure

- 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



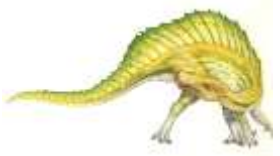
File Attributes

- **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



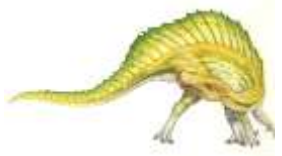
File Operations

- 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



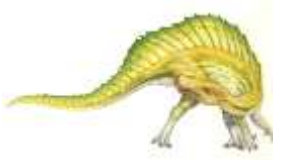
Open Files

- 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



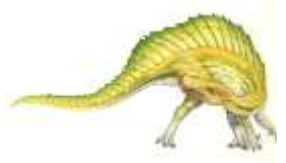
Open File Locking

- 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



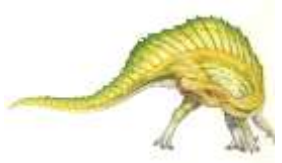
File Locking Example – Java API

```
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 arsg[]) throws IOException {
        FileLock sharedLock = null;
        FileLock exclusiveLock = null;
        try {
            RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");
            // get the channel for the file
            FileChannel ch = raf.getChannel();
            // this locks the first half of the file - exclusive
            exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);
            /** Now modify the data . . . */
            // release the lock
            exclusiveLock.release();
```



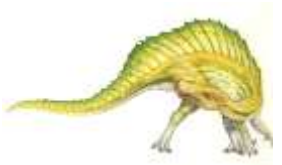
File Locking Example – Java API (Cont.)

```
                    // this locks the second half of the file - shared
                    sharedLock = ch.lock(raf.length()/2+1, raf.length(),
                                         SHARED);
                    /** Now read the data . . . */
                    // release the lock
                    sharedLock.release();
                } catch (java.io.IOException ioe) {
                    System.err.println(ioe);
                }finally {
                    if (exclusiveLock != null)
                    exclusiveLock.release();
                    if (sharedLock != null)
                    sharedLock.release();
                }
            }
        }
```



File Types – Name, Extension

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information



Access Methods

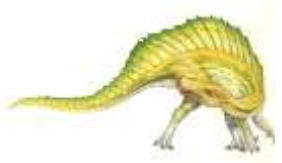
- **Sequential Access**

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

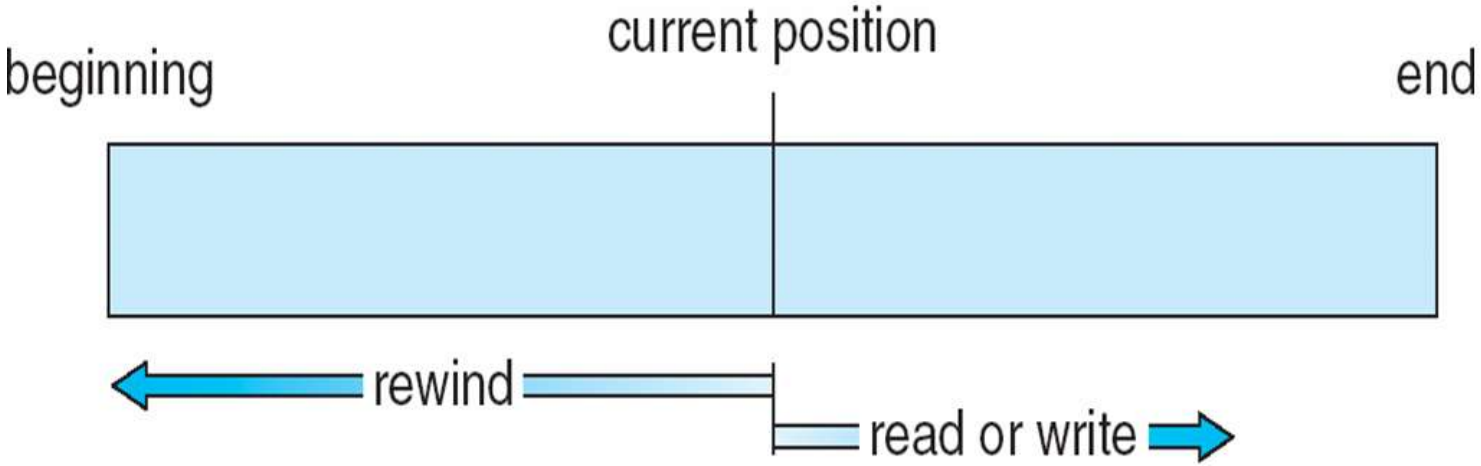
- **Direct Access**

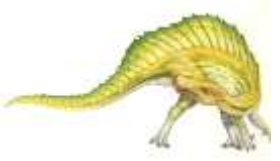
  read *n*
  write *n*
  position to *n*
  read next
  write next
  rewrite *n*

  n = relative block number



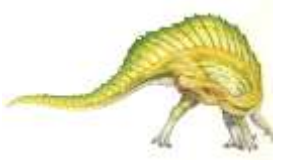
Sequential-access File



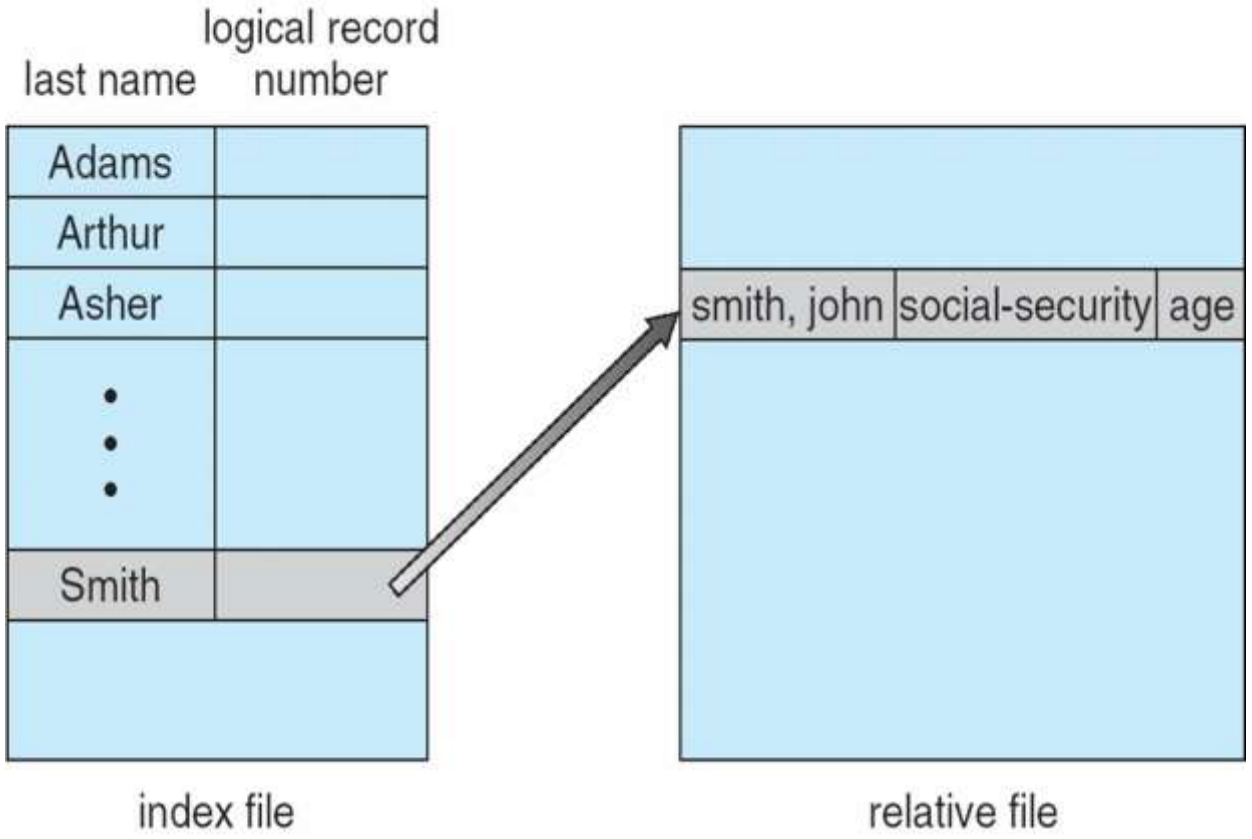


Simulation of Sequential Access on Direct-access File

sequential access	implementation for direct access
reset	$cp = 0;$
read next	*read cp*; $cp = cp + 1;$
write next	*write cp*; $cp = cp + 1;$

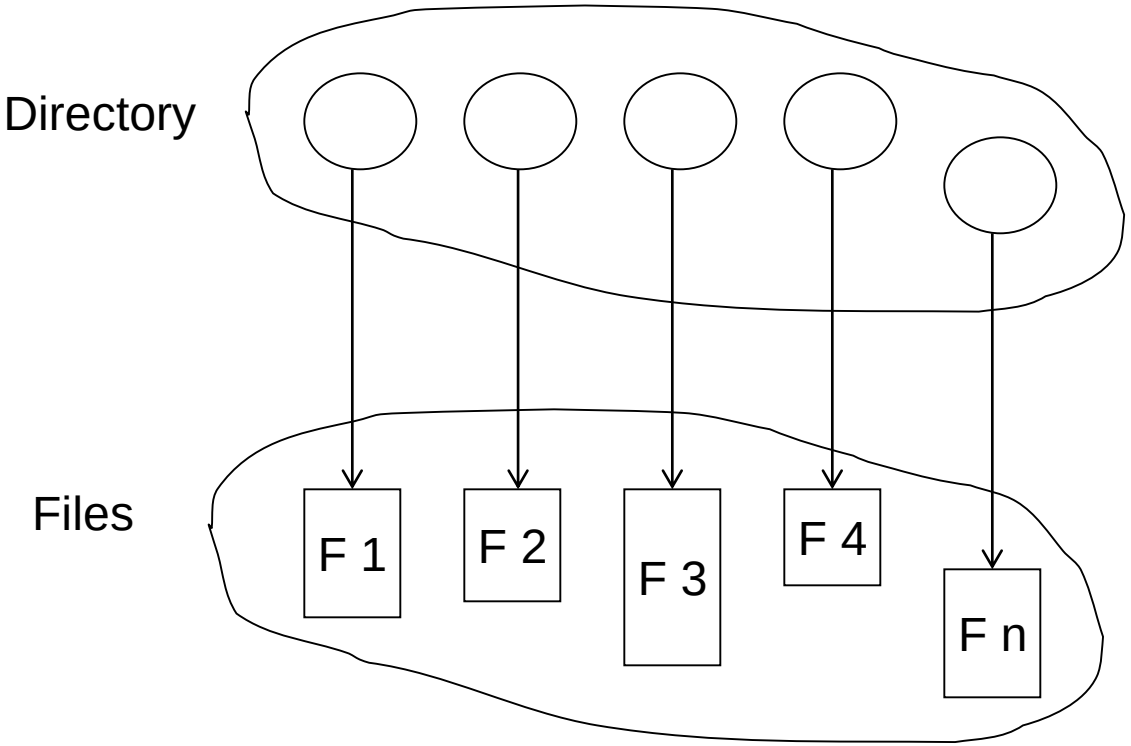


Example of Index and Relative Files

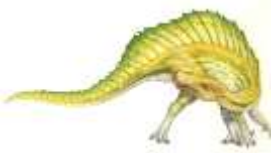


Directory Structure

- A collection of nodes containing information about all files

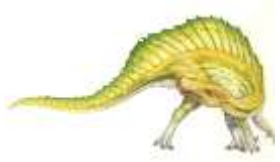


Both the directory structure and the files reside on disk
Backups of these two structures are kept on tapes

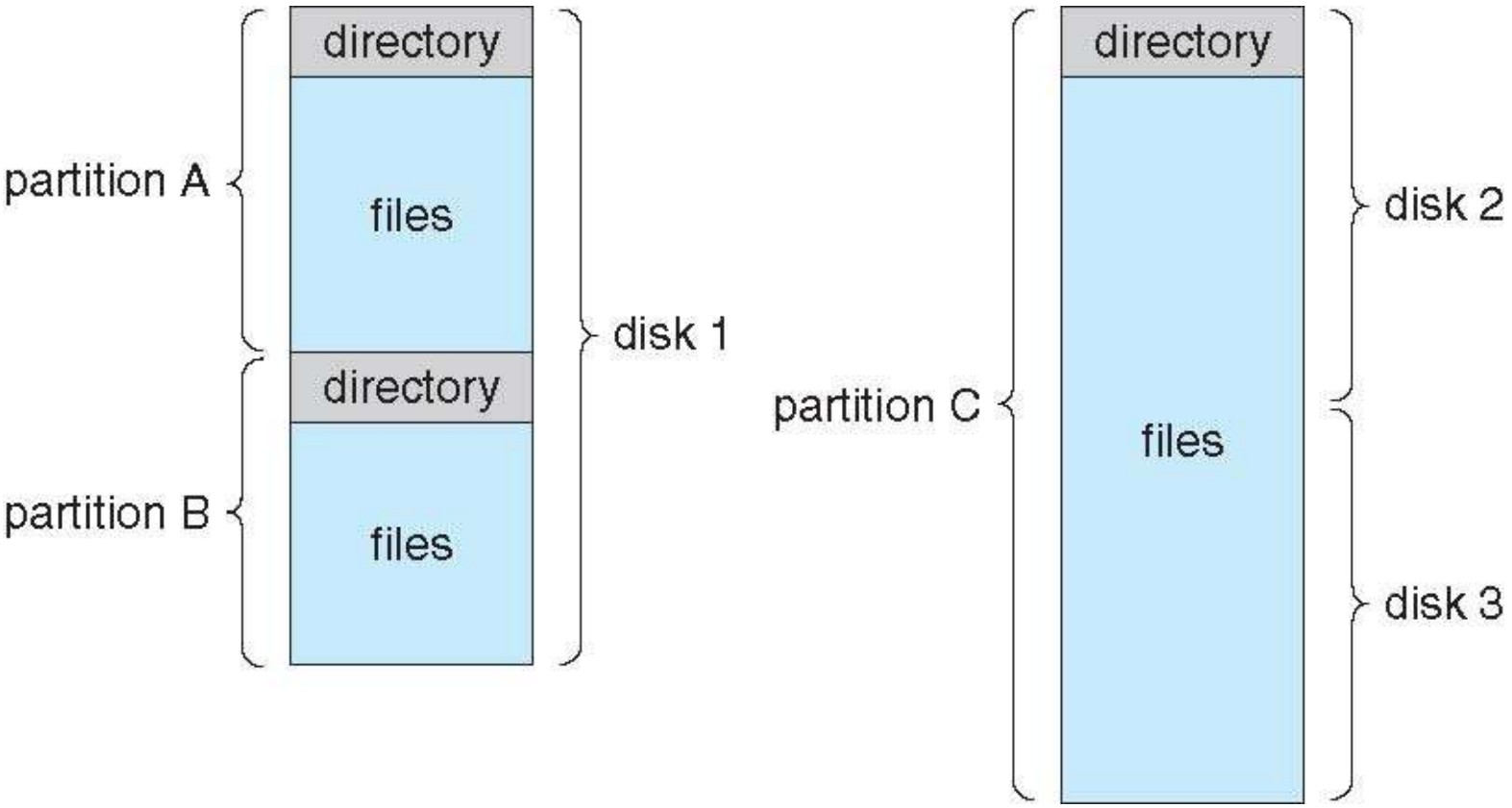


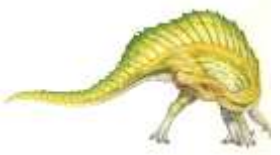
Disk Structure

- 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



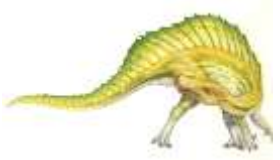
A Typical File-system Organization





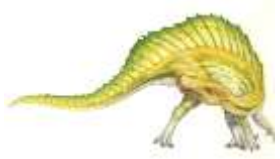
Operations Performed on Directory

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



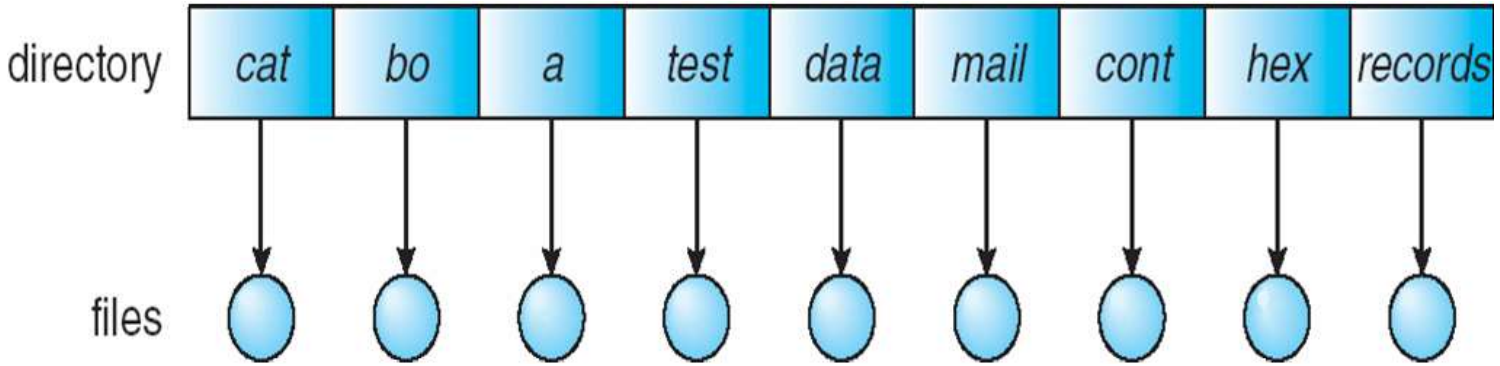
Organize the Directory (Logically) to Obtain

- 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, …)



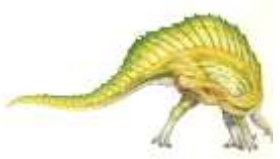
Single-Level Directory

- A single directory for all users



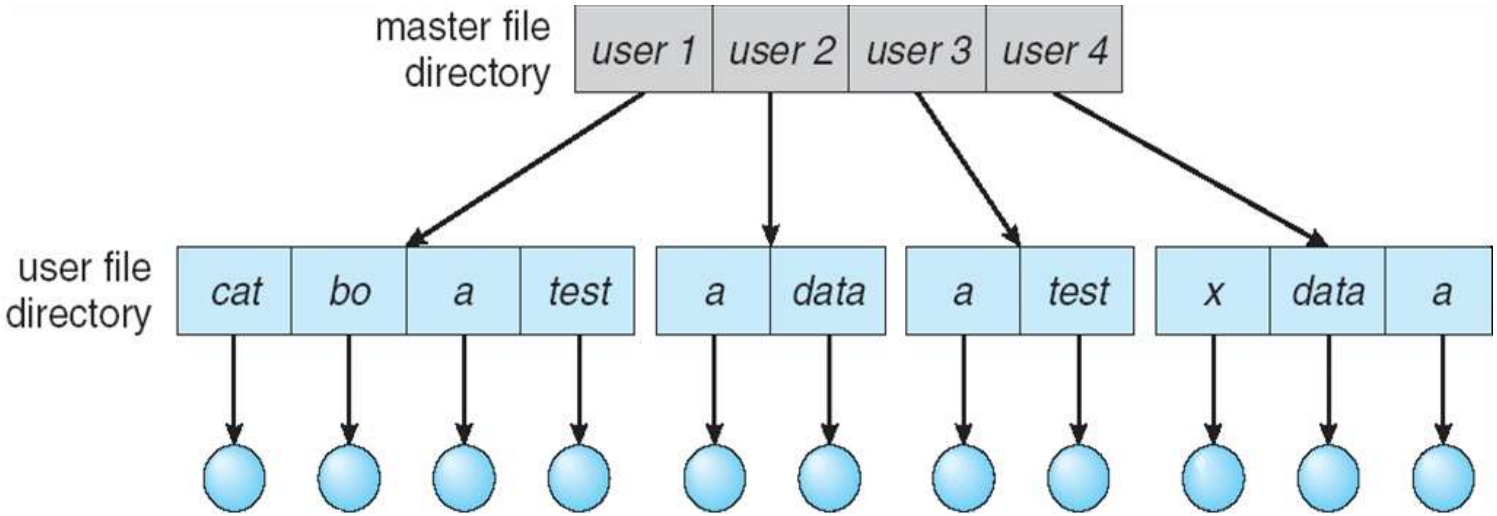
Naming problem

Grouping problem

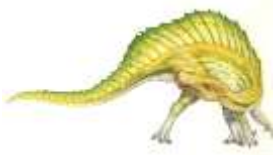


Two-Level Directory

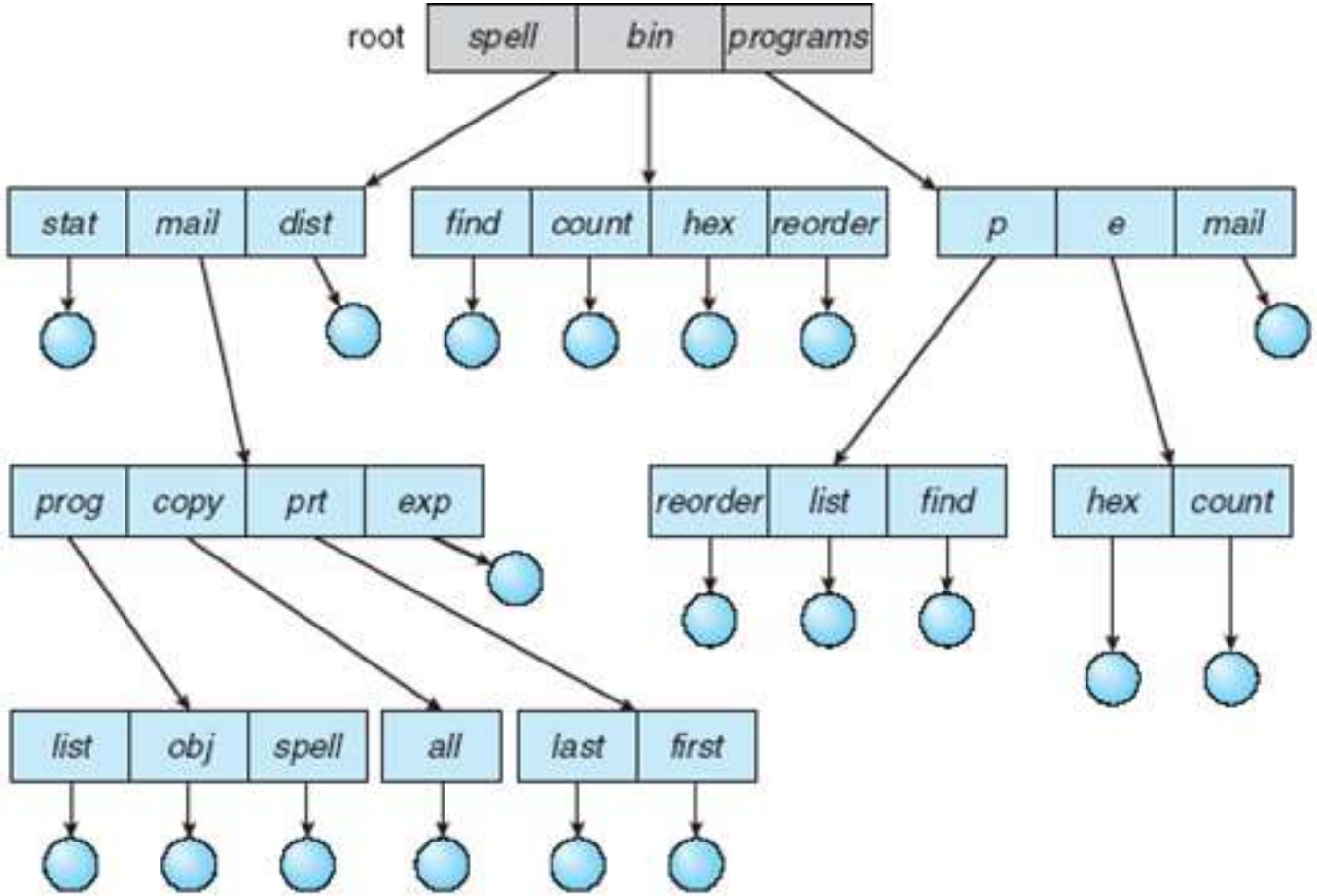
- Separate directory for each user

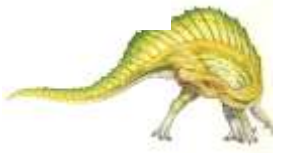


- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability



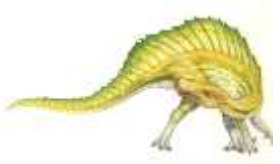
Tree-Structured Directories





Tree-Structured Directories (Cont.)

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



Tree-Structured Directories (Cont)

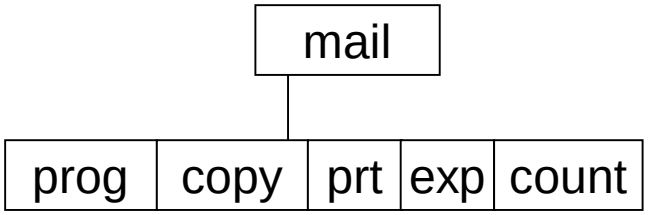
- **Absolute** or **relative** path name
- Creating a new file is done in current directory
- Delete a file

  rm <file-name>
- Creating a new subdirectory is done in current directory

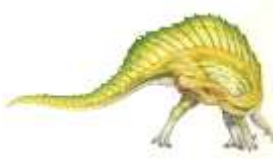
  mkdir <dir-name>

  Example: if in current directory **/mail**

  mkdir count

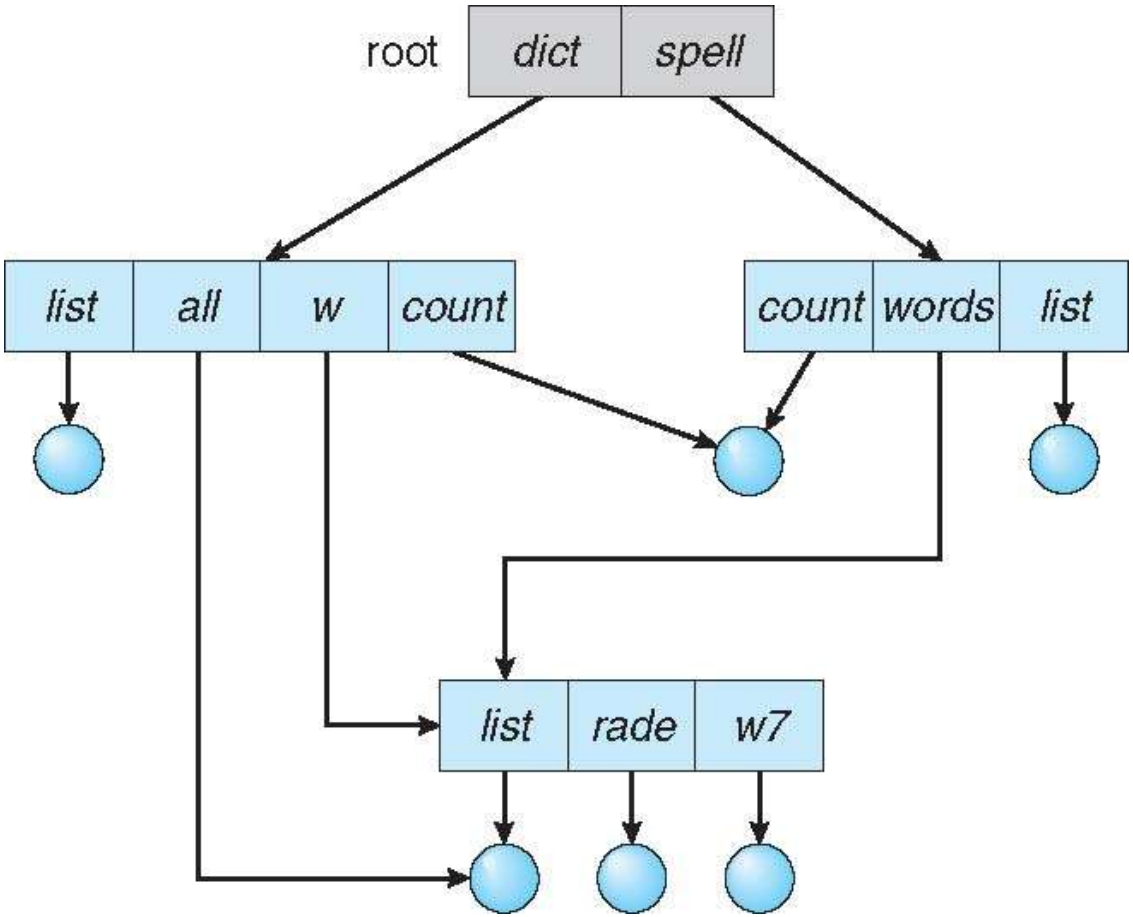


Deleting “mail” ⇒ deleting the entire subtree rooted by “mail”



Acyclic-Graph Directories

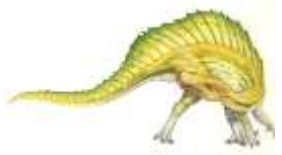
- Have shared subdirectories and files



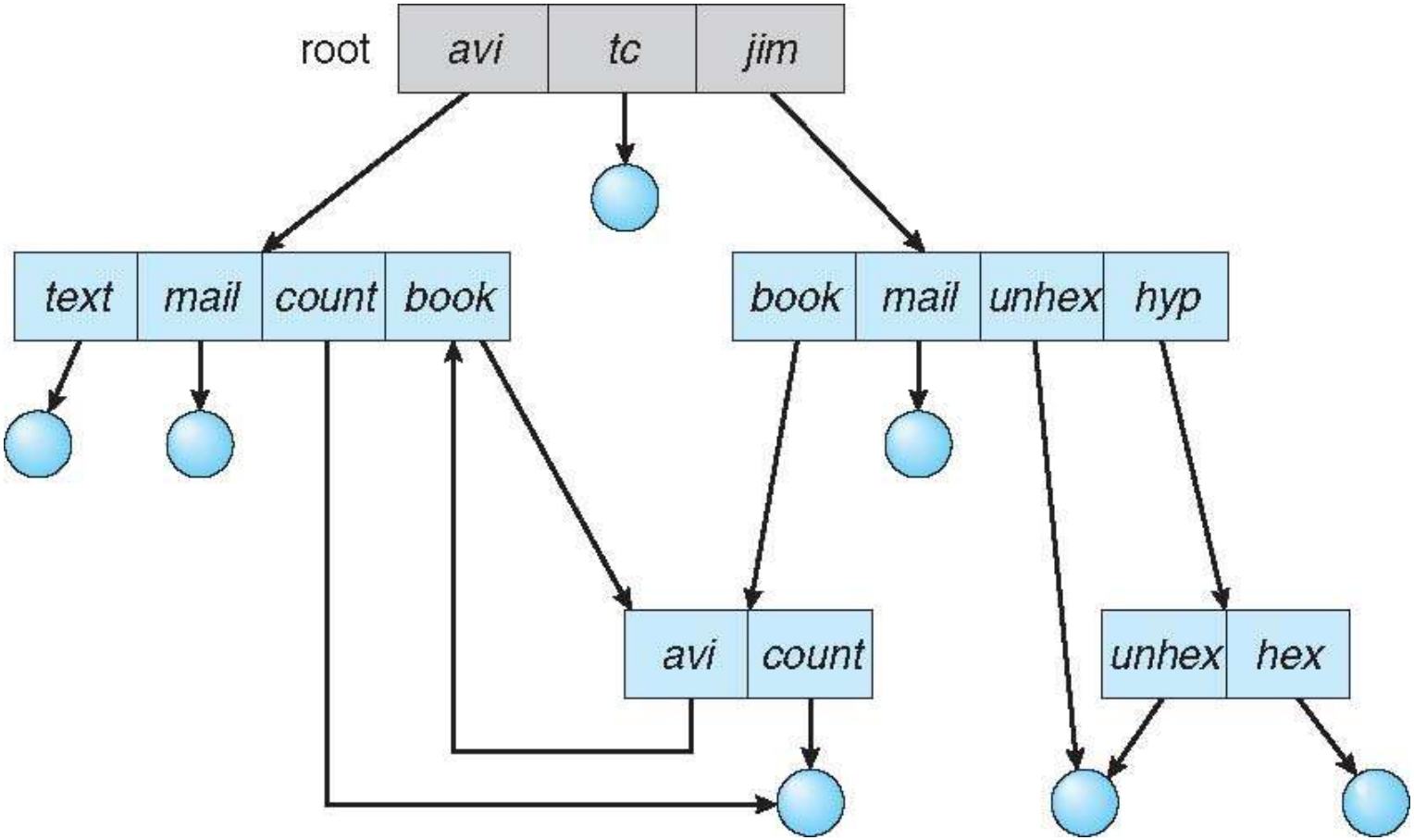
Acyclic-Graph Directories (Cont.)

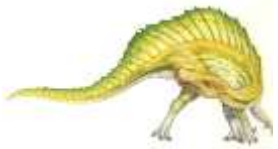
- Two different names (aliasing)
- If *dict* deletes *list* ⇒ dangling pointer

  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



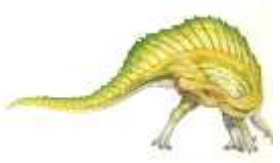
General Graph Directory





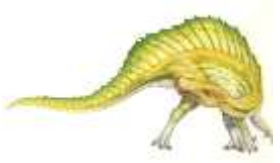
General Graph Directory (Cont.)

- 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

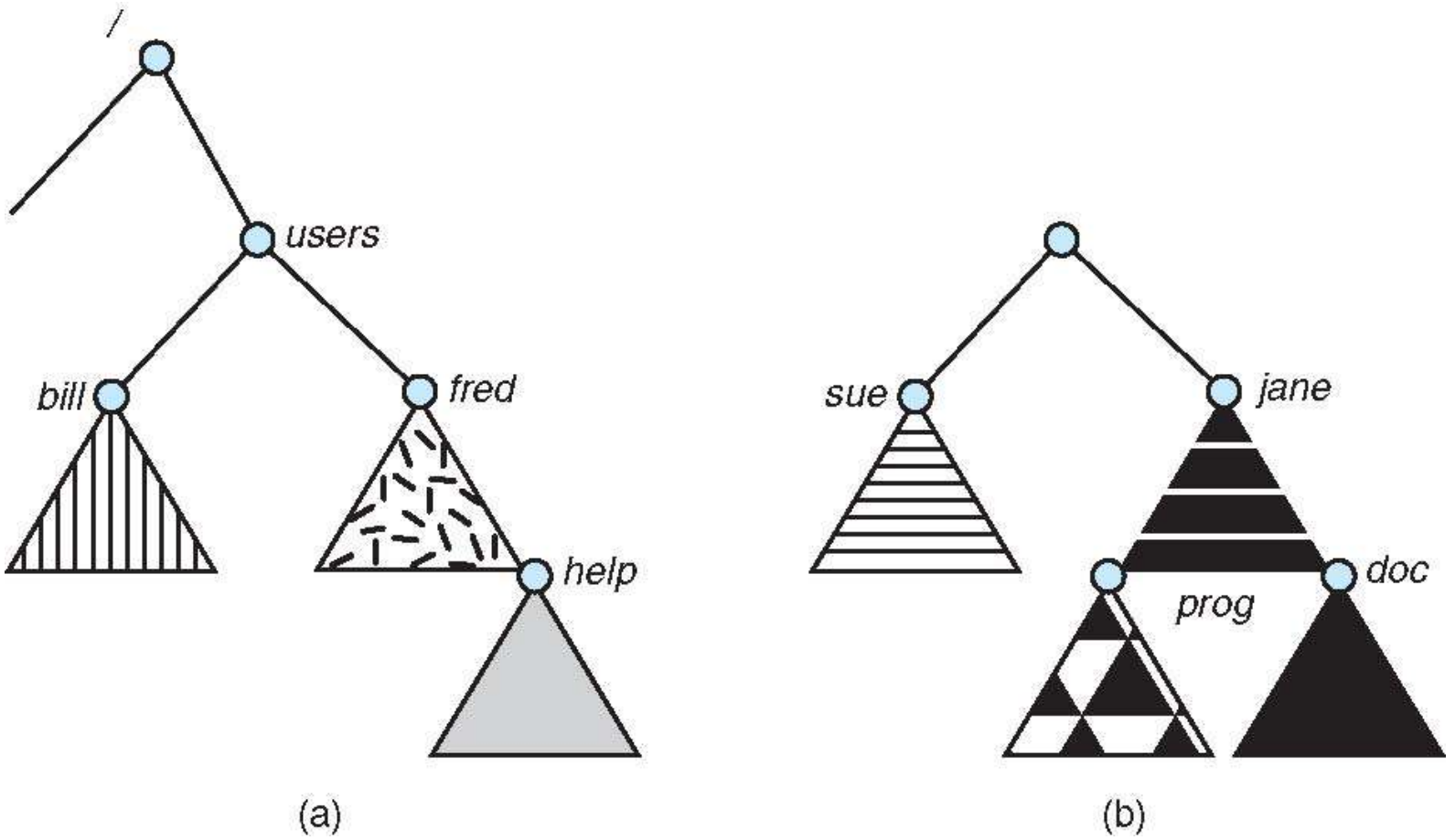


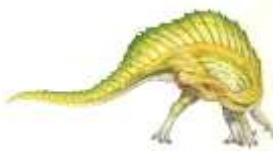
File System Mounting

- 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**

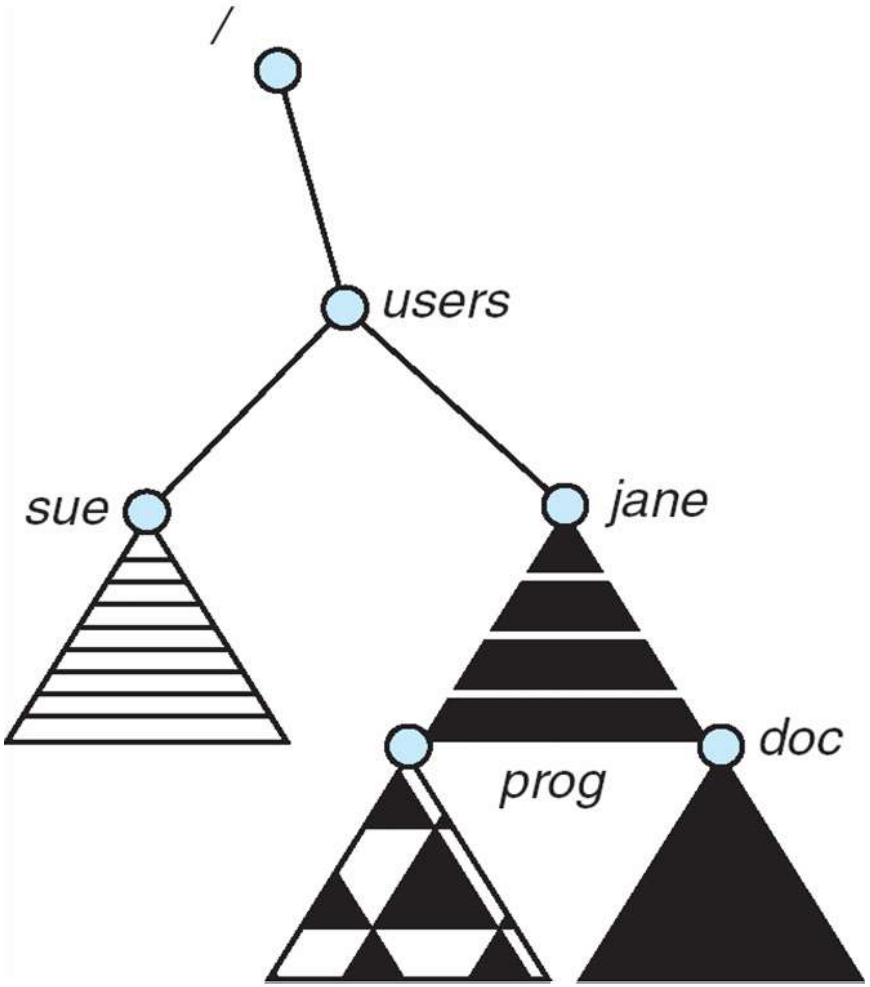


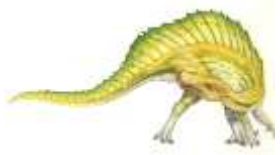
(a) Existing (b) Unmounted Partition





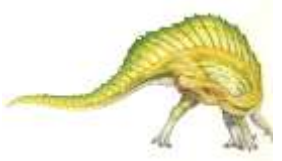
Mount Point





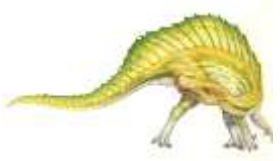
File Sharing

- 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



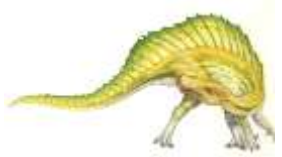
File Sharing – Multiple Users

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



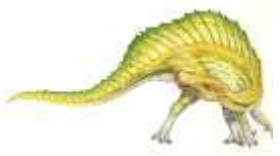
File Sharing – Remote File Systems

- 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



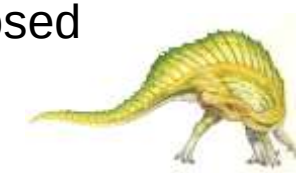
File Sharing – Failure Modes

- 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



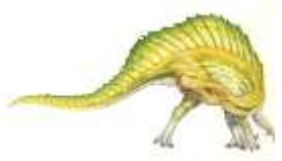
File Sharing – Consistency Semantics

- **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**

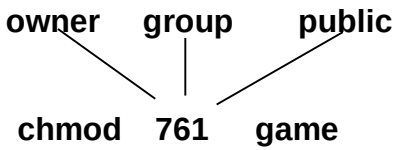


Access Lists and Groups

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

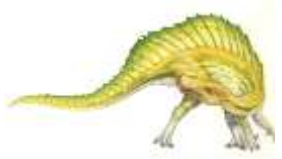
			RWX
a) **owner access**	7	⇒	1 1 1
b) **group access**	6	⇒	RWX 1 1 0
c) **public access**	1	⇒	RWX 0 0 1

- 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.

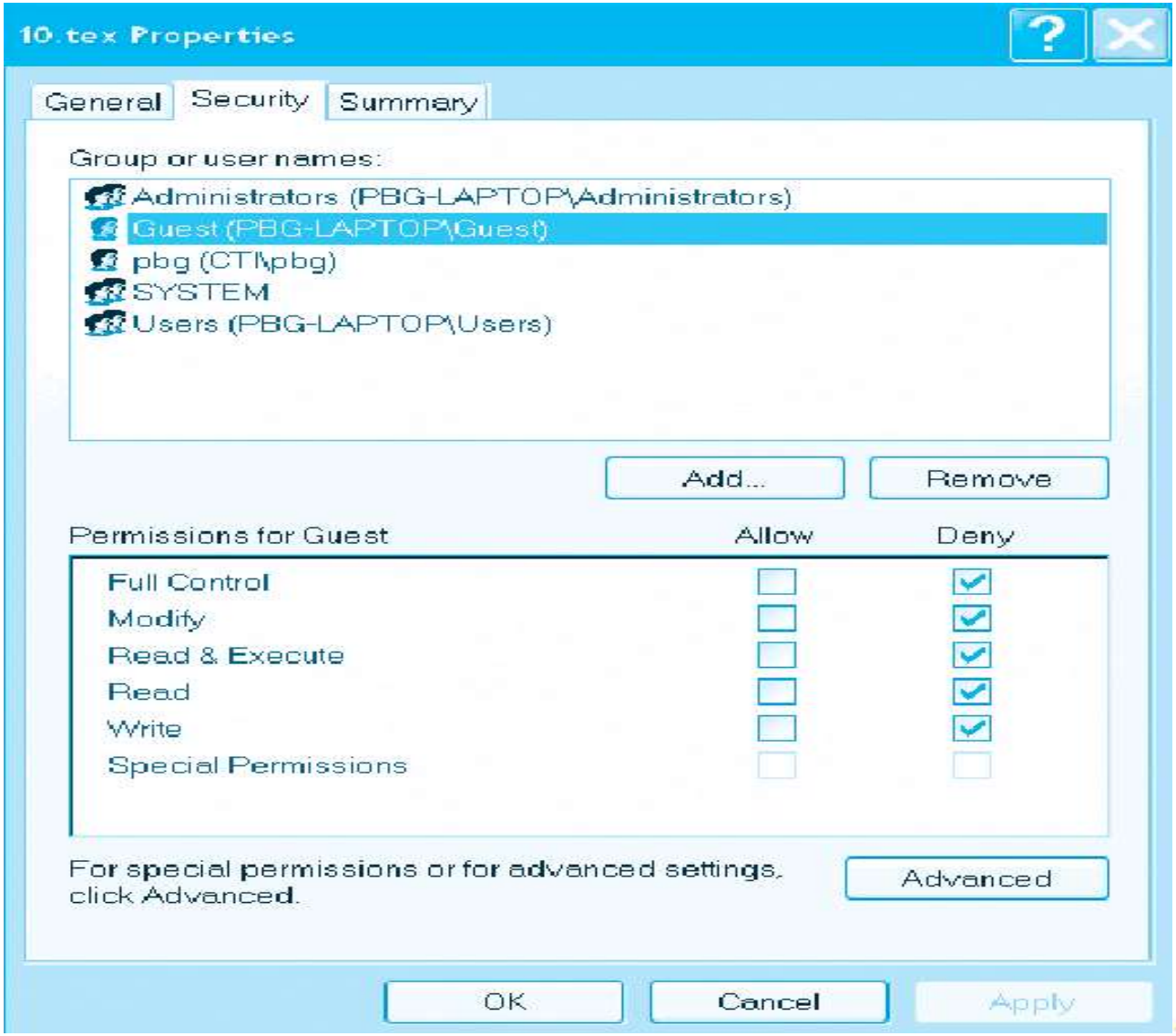


Attach a group to a file

chgrp G game



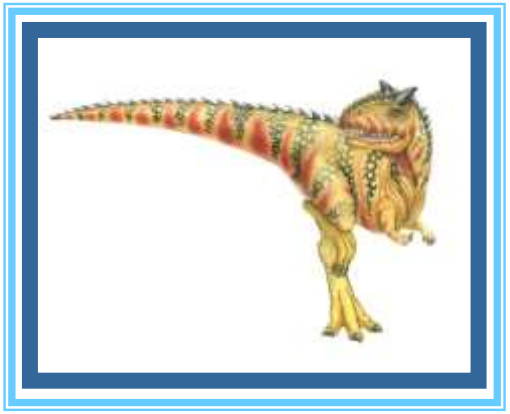
Windows XP Access-Control List Management



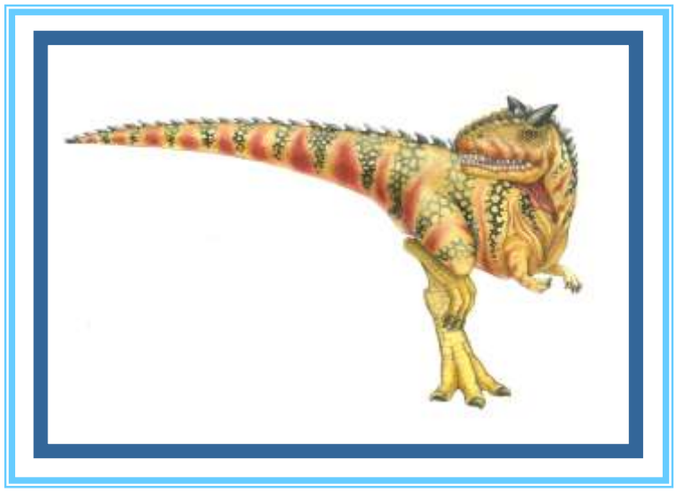
A Sample UNIX Directory Listing

-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 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
- Stable-Storage Implementation
- Tertiary Storage Devices

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 250 times per second
  - **Transfer rate** is rate at which data flow between drive and computer
  - **Positioning time** (**random-access time**) is time to move disk arm to desired cylinder (**seek time**) and time for desired sector to rotate under the disk head (**rotational latency**)
  - **Head crash** results from disk head making contact with the disk surface
    - That's bad
- Disks can be removable
- Drive attached to computer via **I/O bus**
  - Busses vary, including **EIDE**, **ATA**, **SATA**, **USB**, **Fibre Channel**, **SCSI, SAS, Firewire**
  - **Host controller** in computer uses bus to talk to **disk controller** built into drive or storage array

Magnetic Disks

- Platters range from .85” to 14” (historically)
  - Commonly 3.5”, 2.5”, and 1.8”
- Range from 30GB to 3TB per drive
- Performance
  - Transfer Rate – theoretical – 6 Gb/sec
  - Effective Transfer Rate – real – 1Gb/sec
  - Seek time from 3ms to 12ms – 9ms common for desktop drives
  - Average seek time measured or calculated based on 1/3 of tracks
  - Latency based on spindle speed
    - 1/(RPM * 60)
  - Average latency = ½ latency

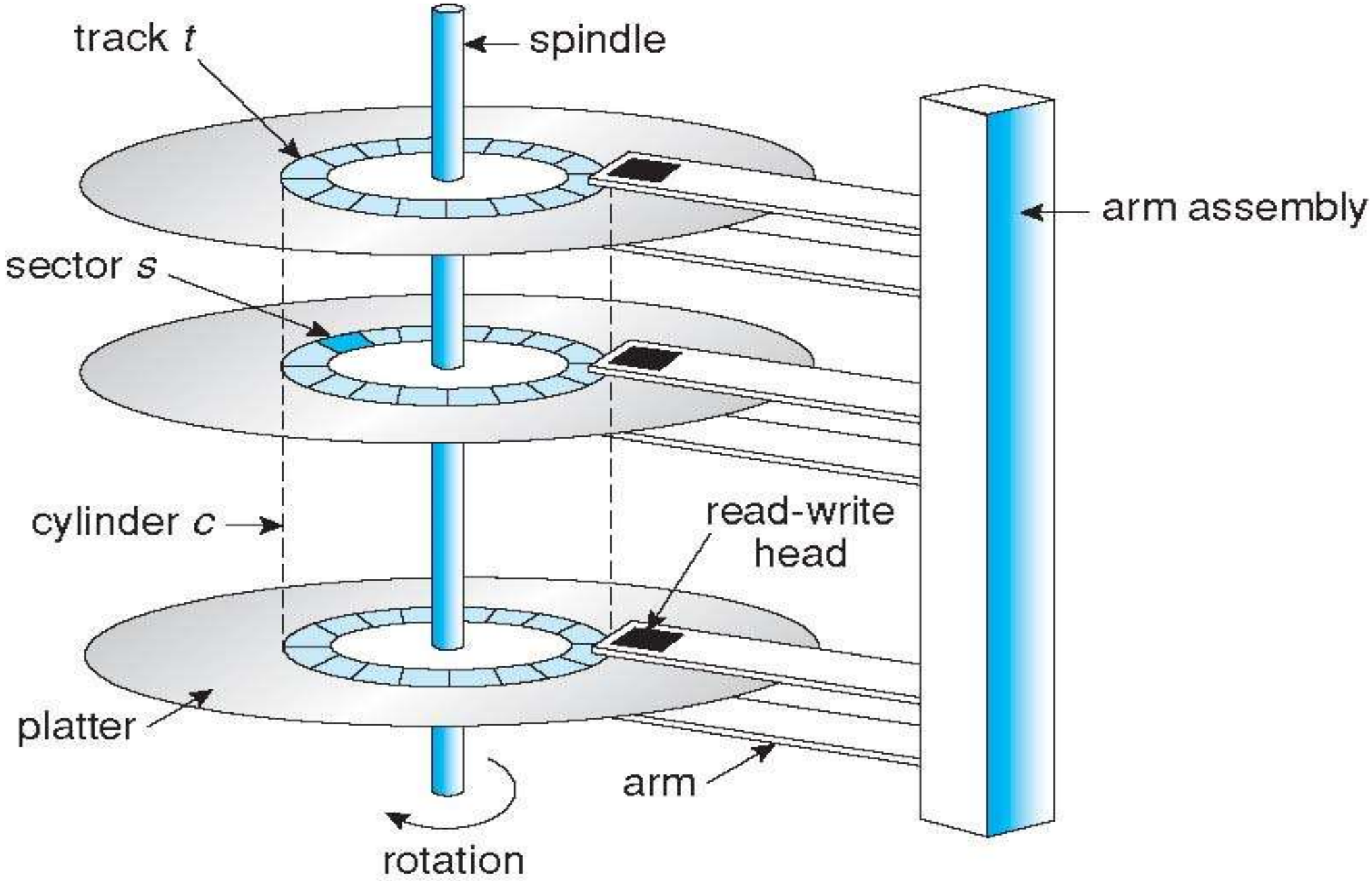
Spindle [rpm]	Average latency [ms]
4200	7.14
5400	5.56
7200	4.17
10000	3
15000	2

(From Wikipedia)

Magnetic Disk Performance

- **Access Latency** = **Average access time** = average seek time + average latency
  - For fastest disk 3ms + 2ms = 5ms
  - For slow disk 9ms + 5.56ms = 14.56ms
- Average I/O time = average access time + (amount to transfer / transfer rate) + controller overhead
- For example to transfer a 4KB block on a 7200 RPM disk with a 5ms average seek time, 1Gb/sec transfer rate with a .1ms controller overhead =
  - 5ms + 4.17ms + 4KB / 1Gb/sec + 0.1ms =
  - 9.27ms + 4 / 131072 sec =
  - 9.27ms + .12ms = 9.39ms

Moving-head Disk Mechanism



The First Commercial Disk Drive

1956
IBM RAMDAC computer included the IBM Model 350 disk storage system

5M (7 bit) characters
50 x 24” platters
Access time = < 1 second

Magnetic Tape

- Was early secondary-storage medium
  - Evolved from open spools to cartridges
- 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
  - 140MB/sec and greater
- 200GB to 1.5TB typical storage
- Common technologies are LTO-{3,4,5} and T10000

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
  - Logical to physical address should be easy
    - Except for bad sectors
    - Non-constant # of sectors per track via constant angular velocity

Disk Attachment

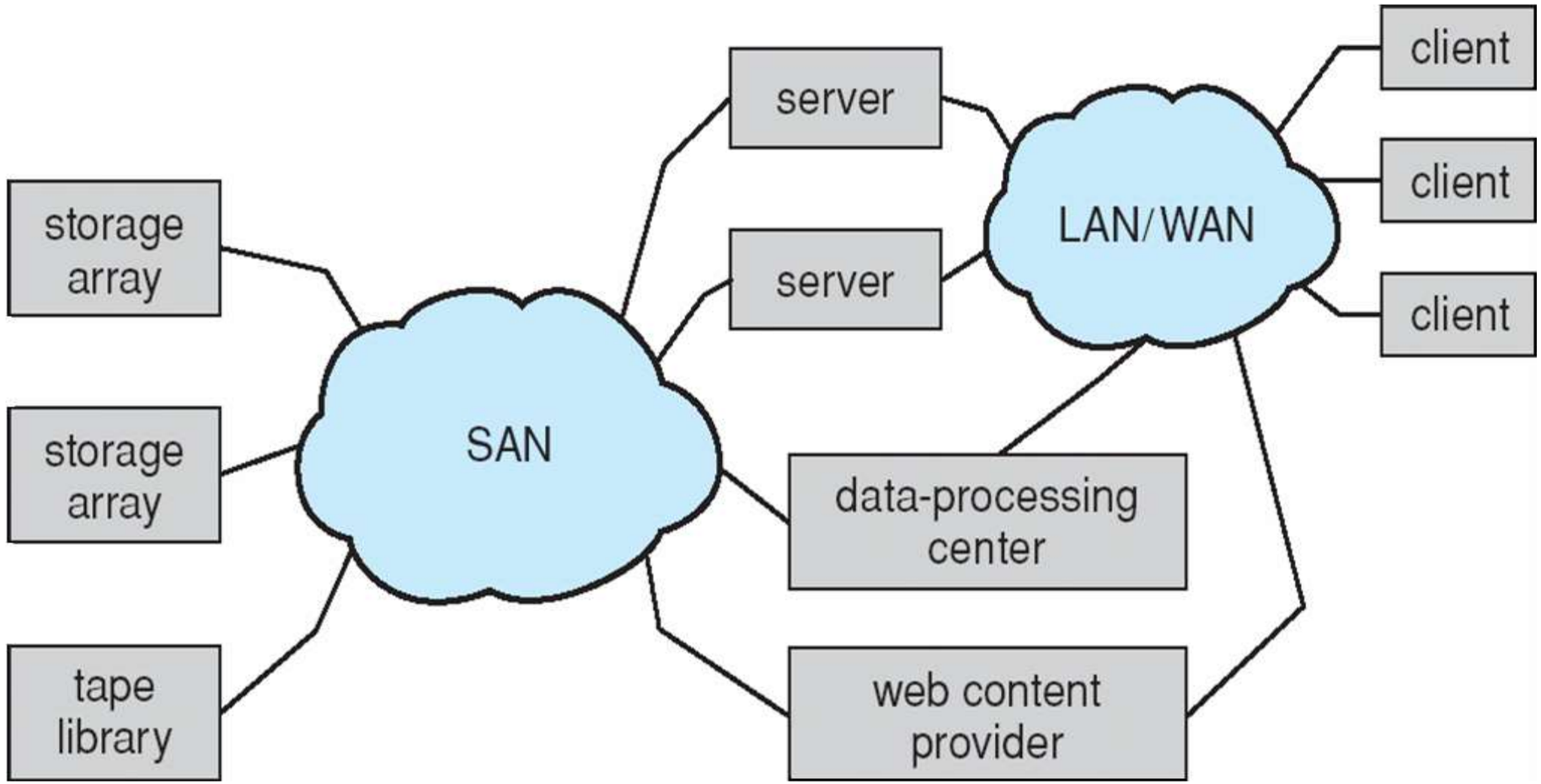
- Host-attached storage accessed through I/O ports talking to I/O busses
- SCSI itself is a bus, up to 16 devices on one cable, **SCSI initiator** requests operation and **SCSI targets** perform tasks
  - Each target can have up to 8 **logical units** (disks attached to device controller)
- FC is high-speed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

Storage Array

- Can just attach disks, or arrays of disks
- Storage Array has controller(s), provides features to attached host(s)
  - Ports to connect hosts to array
  - Memory, controlling software (sometimes NVRAM, etc)
  - A few to thousands of disks
  - RAID, hot spares, hot swap (discussed later)
  - Shared storage -> more efficiency
  - Features found in some file systems
    - Snaphots, clones, thin provisioning, replication, deduplication, etc

Storage Area Network

- Common in large storage environments
- Multiple hosts attached to multiple storage arrays - flexible

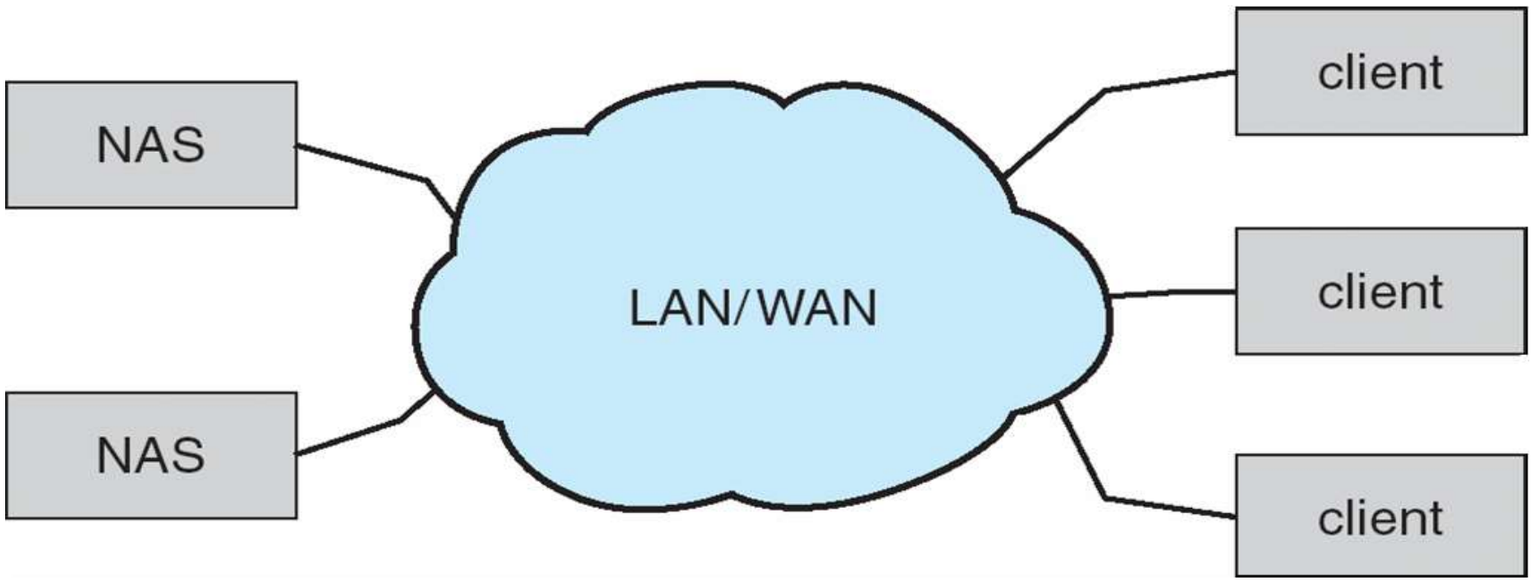


Storage Area Network (Cont.)

- SAN is one or more storage arrays
  - Connected to one or more Fibre Channel switches
- Hosts also attach to the switches
- Storage made available via **LUN Masking** from specific arrays to specific servers
- Easy to add or remove storage, add new host and allocate it storage
  - Over low-latency Fibre Channel fabric
- Why have separate storage networks and communications networks?
  - Consider iSCSI, FCOE

Network-Attached Storage

- Network-attached storage (**NAS**) is storage made available over a network rather than over a local connection (such as a bus)
  - Remotely attaching to file systems
- NFS and CIFS are common protocols
- Implemented via remote procedure calls (RPCs) between host and storage over typically TCP or UDP on IP network
- **iSCSI** protocol uses IP network to carry the SCSI protocol
  - Remotely attaching to devices (blocks)



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
- 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

Disk Scheduling (Cont.)

- There are many sources of disk I/O request
  - OS
  - System processes
  - Users processes
- I/O request includes input or output mode, disk address, memory address, number of sectors to transfer
- OS maintains queue of requests, per disk or device
- Idle disk can immediately work on I/O request, busy disk means work must queue
  - Optimization algorithms only make sense when a queue exists
- Note that drive controllers have small buffers and can manage a queue of I/O requests (of varying “depth”)

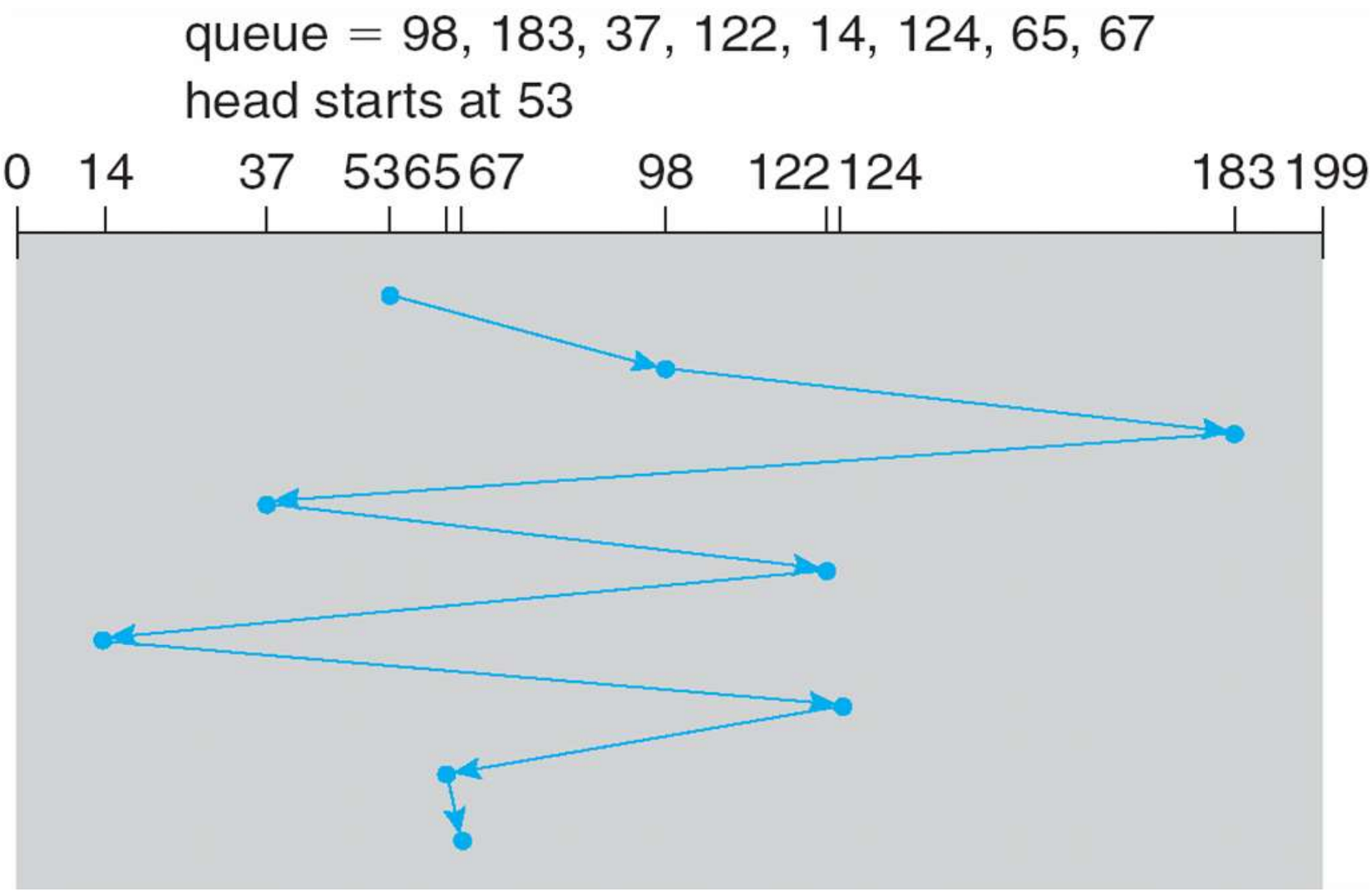
- Several algorithms exist to schedule the servicing of disk I/O requests
- The analysis is true for one or many platters
- We illustrate scheduling algorithms with a request queue (0-199)

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

Head pointer 53

FCFS

Illustration shows total head movement of 640 cylinders



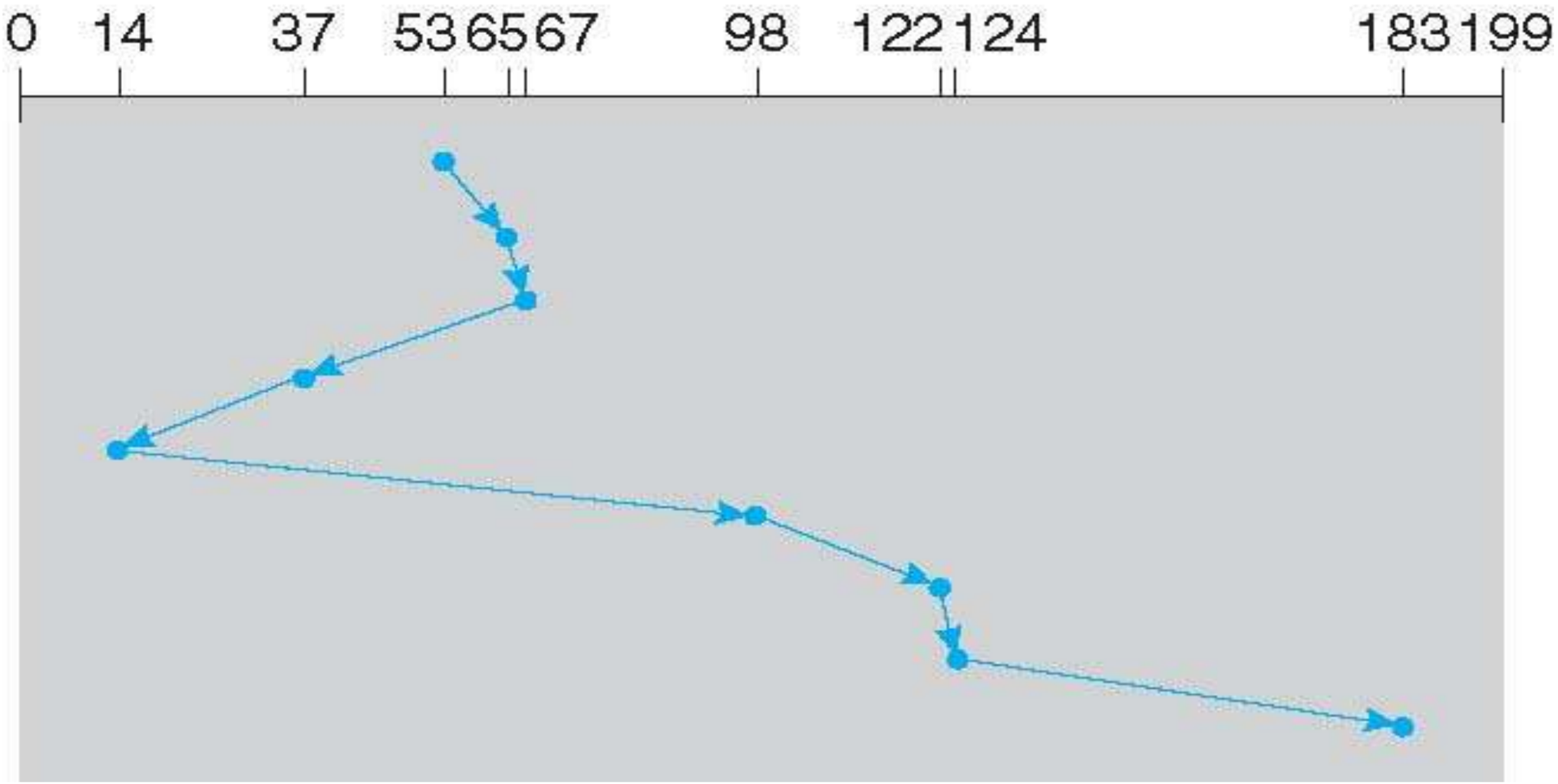
SSTF

- Shortest Seek Time First 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



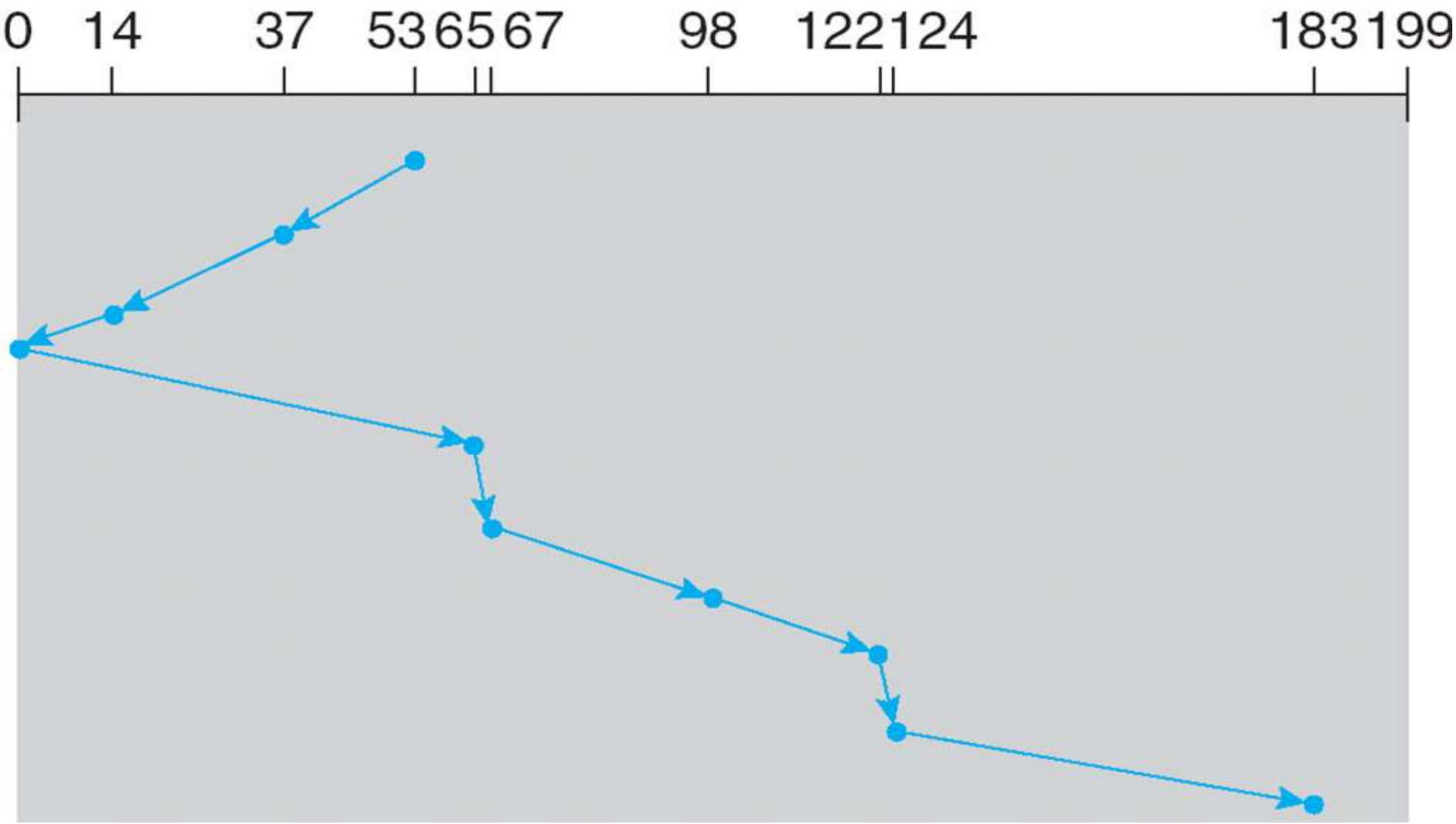
SCAN

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

SCAN (Cont.)

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

head starts at 53



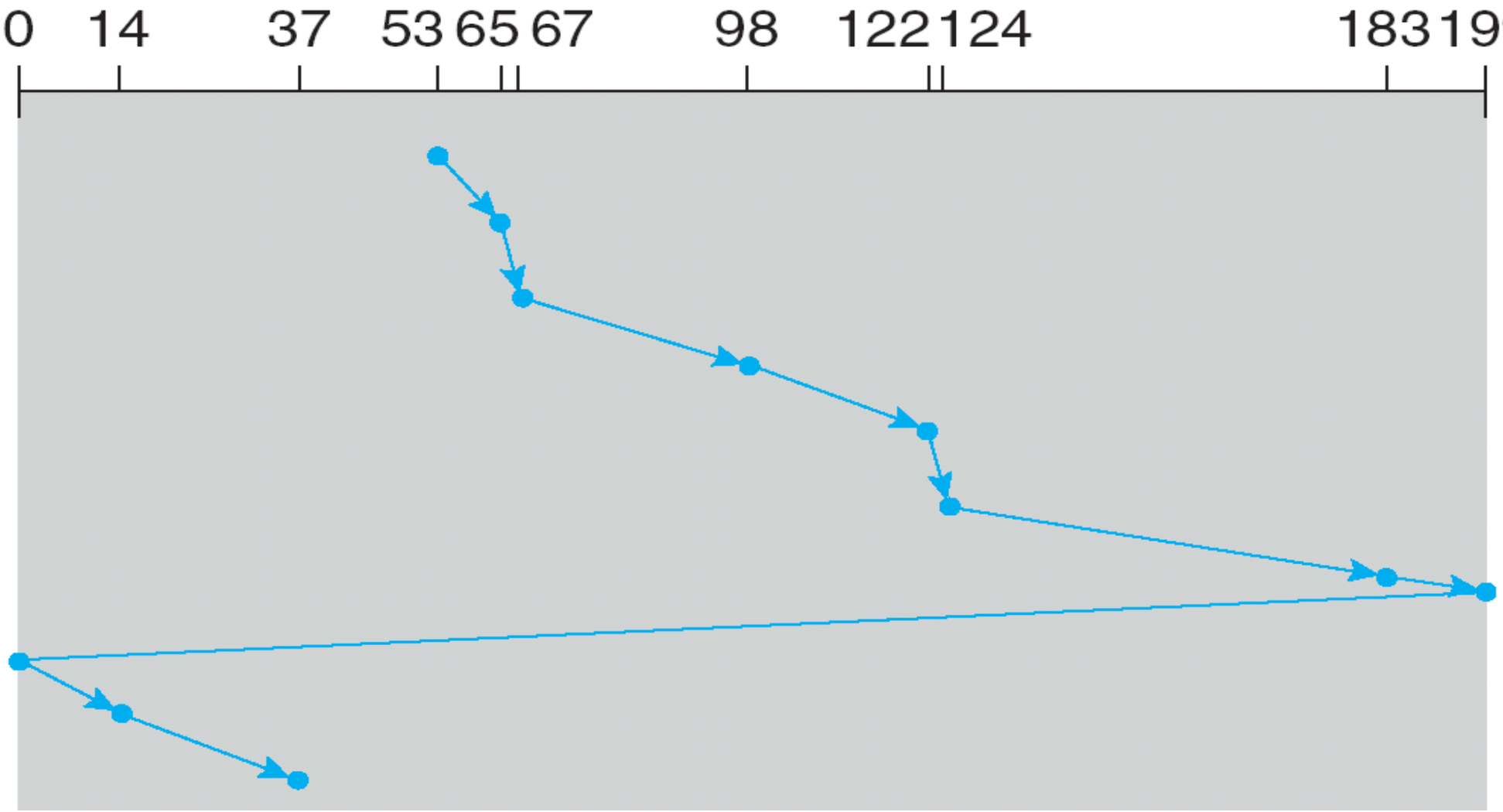
C-SCAN

- Provides a more uniform wait time than SCAN
- The head moves from one end of the disk to the other, servicing requests as it goes
  - When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one
- Total number of cylinders?

C-SCAN (Cont.)

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

head starts at 53



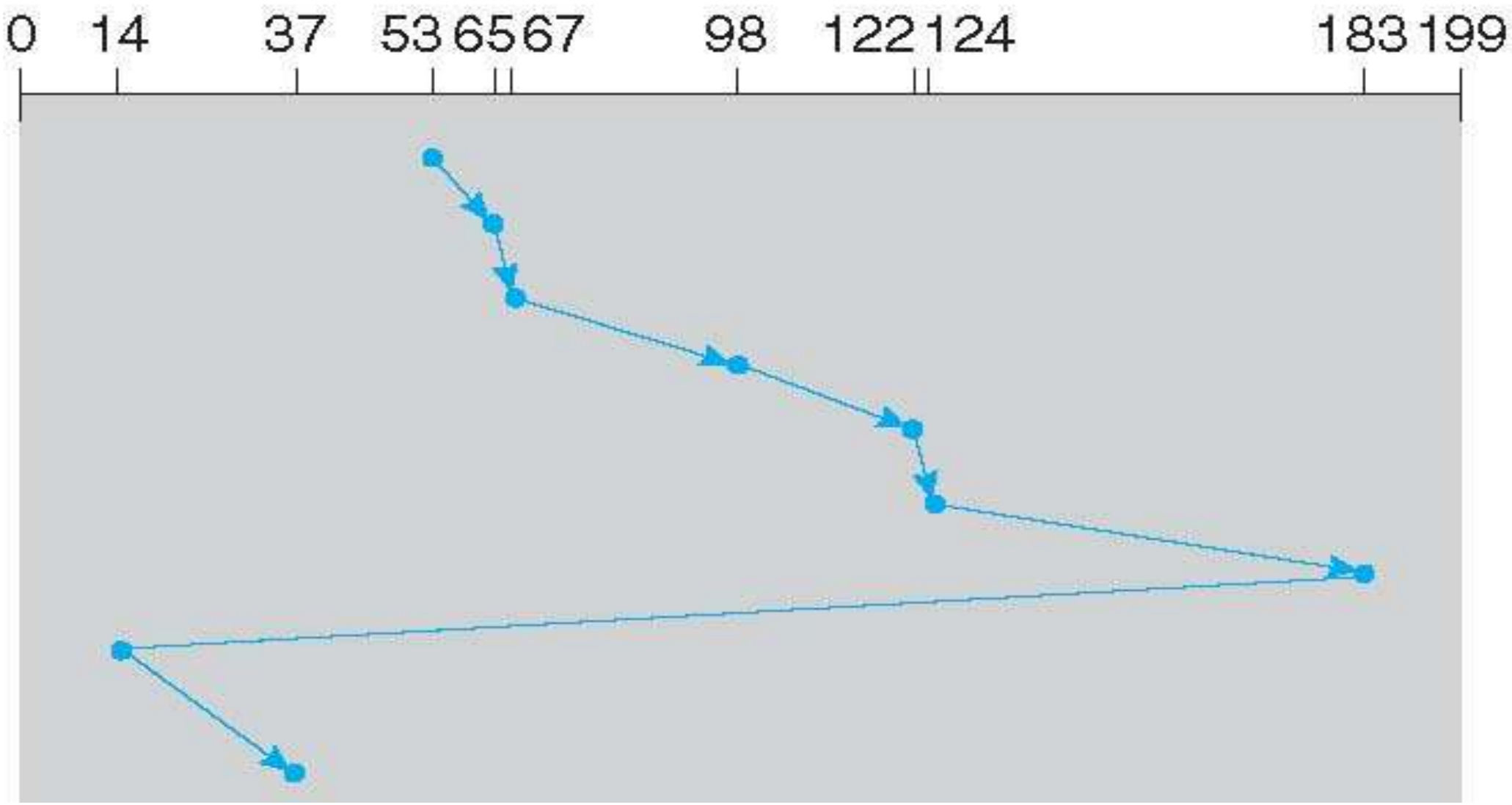
C-LOOK

- LOOK a version of SCAN, C-LOOK a 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
- Total number of cylinders?

C-LOOK (Cont.)

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

head starts at 53



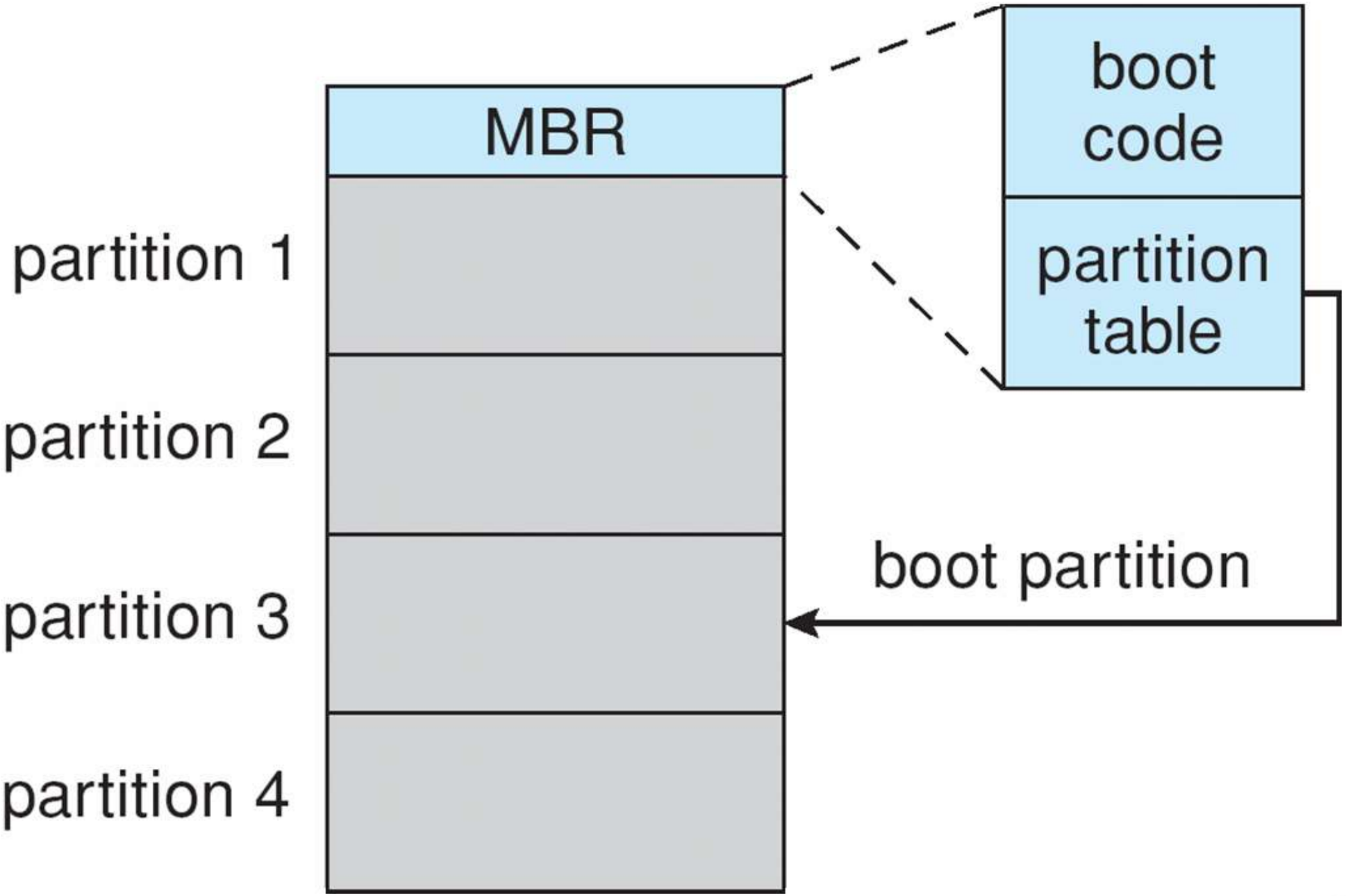
Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk
  - Less starvation
- Performance depends on the number and types of requests
- Requests for disk service can be influenced by the file-allocation method
  - And metadata layout
- 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
- What about rotational latency?
  - Difficult for OS to calculate
- How does disk-based queuing effect OS queue ordering efforts?

Disk Management

- **Low-level formatting**, or **physical formatting** — Dividing a disk into sectors that the disk controller can read and write
  - Each sector can hold header information, plus data, plus error correction code (**ECC**)
  - Usually 512 bytes of data but can be selectable
- To use a disk to hold files, the operating system still needs to record its own data structures on the disk
  - **Partition** the disk into one or more groups of cylinders, each treated as a logical disk
  - **Logical formatting** or “making a file system”
  - To increase efficiency most file systems group blocks into **clusters**
    - Disk I/O done in blocks
    - File I/O done in clusters
- Boot block initializes system
  - The bootstrap is stored in ROM
  - **Bootstrap loader** program stored in boot blocks of boot partition
- Methods such as **sector sparing** used to handle bad blocks

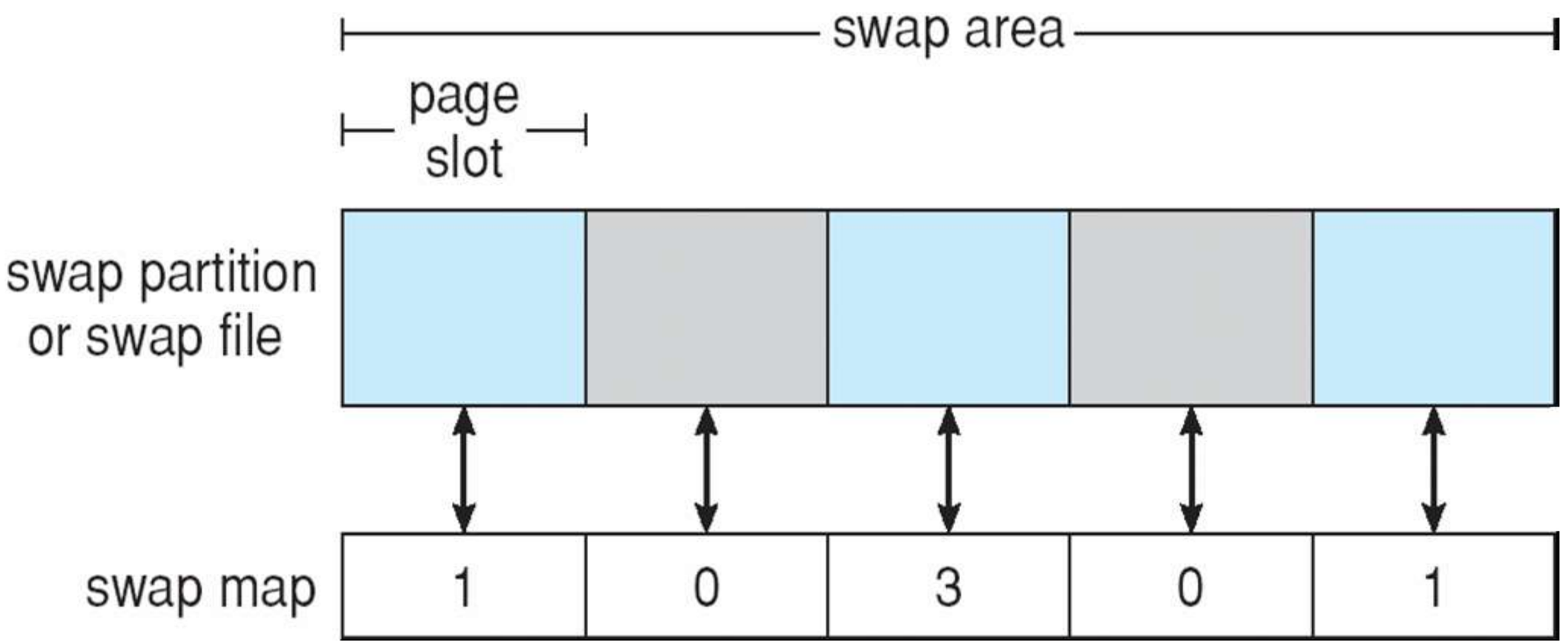
Booting from a Disk in Windows 2000



Swap-Space Management

- Swap-space — Virtual memory uses disk space as an extension of main memory
  - Less common now due to memory capacity increases
- Swap-space can be carved out of the normal file system, or, more commonly, it can be in a separate disk partition (raw)
- 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 dirty page is forced out of physical memory, not when the virtual memory page is first created
    - File data written to swap space until write to file system requested
    - Other dirty pages go to swap space due to no other home
    - Text segment pages thrown out and reread from the file system as needed
- What if a system runs out of swap space?
- Some systems allow multiple swap spaces

Data Structures for Swapping on Linux Systems



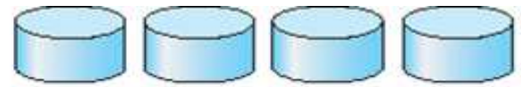
RAID Structure

- 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

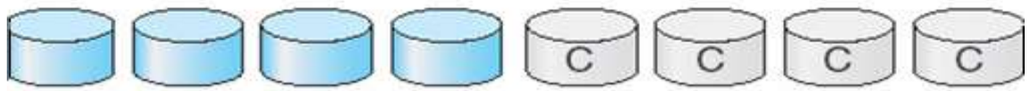
RAID (Cont.)

- 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** (**RAID 1**) keeps duplicate of each disk
  - Striped mirrors (**RAID 1+0**) or mirrored stripes (**RAID 0+1**) provides high performance and high reliability
  - **Block interleaved parity** (**RAID 4, 5, 6**) uses much less redundancy
- RAID within a storage array can still fail if the array fails, so automatic **replication** of the data between arrays is common
- Frequently, a small number of **hot-spare** disks are left unallocated, automatically replacing a failed disk and having data rebuilt onto them

RAID Levels



(a) RAID 0: non-redundant striping.



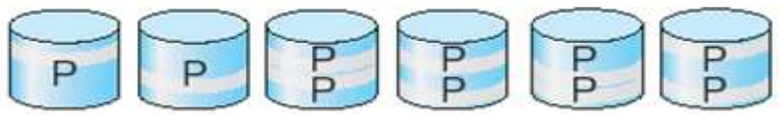
(b) RAID 1: mirrored disks.

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

(d) RAID 3: bit-interleaved parity.

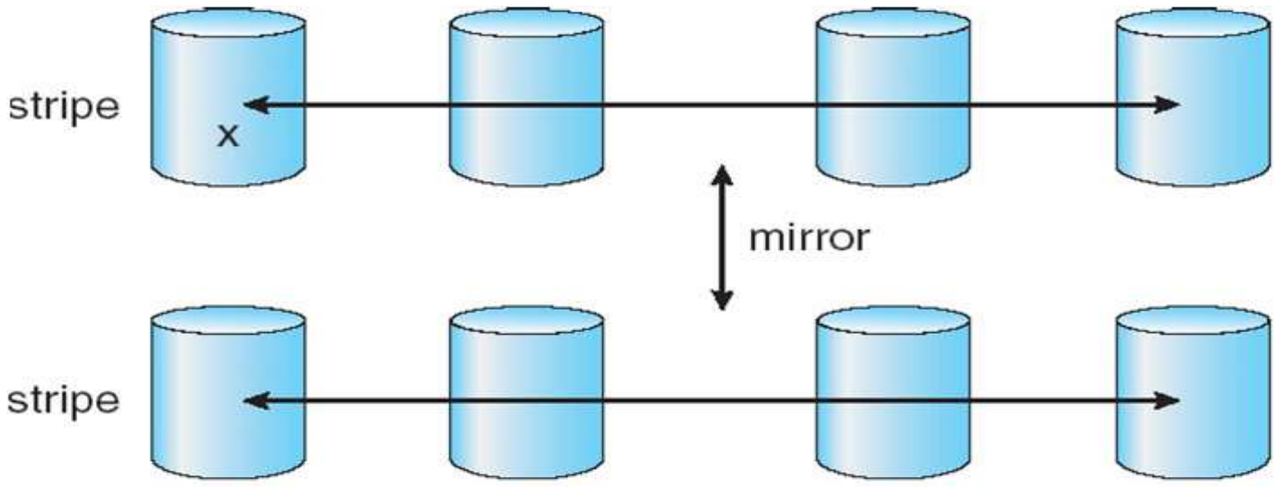
(e) RAID 4: block-interleaved parity.

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

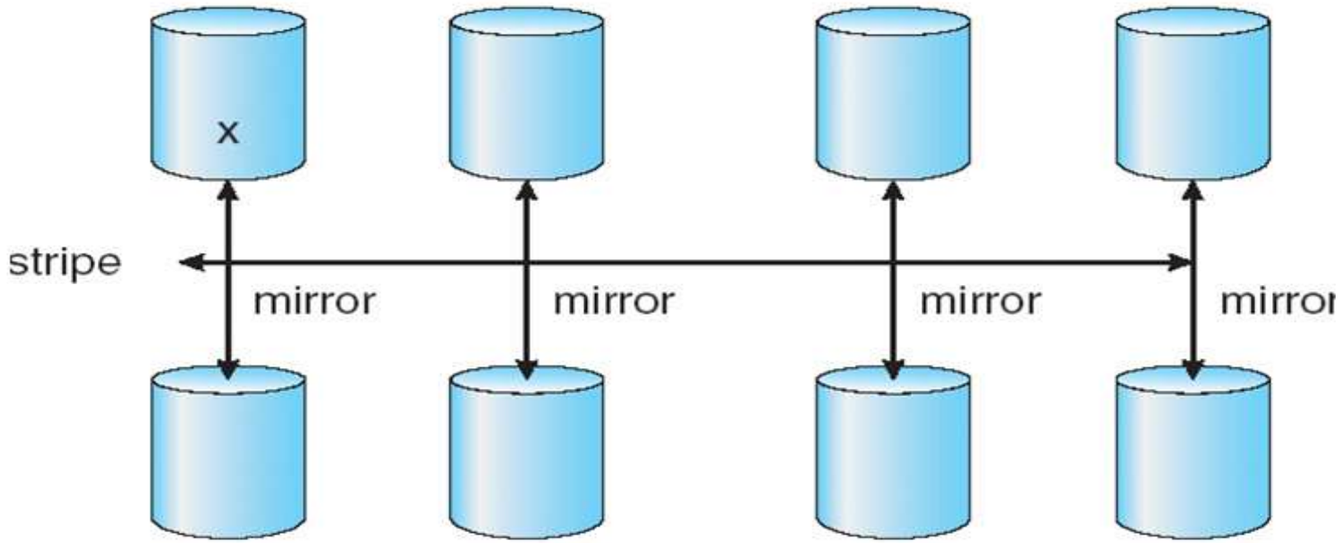


(g) RAID 6: P + Q redundancy.

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



a) RAID 0 + 1 with a single disk failure.

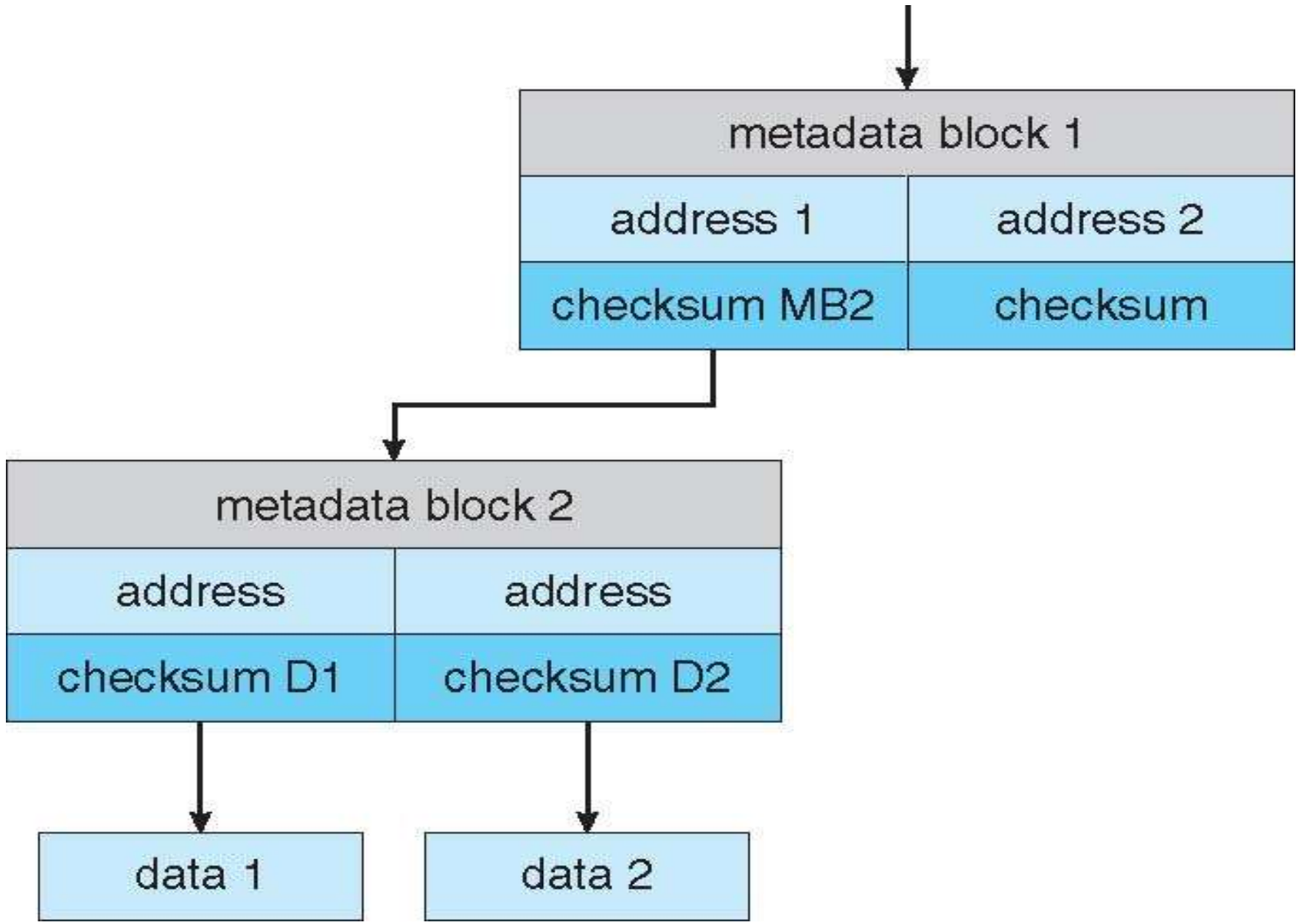


b) RAID 1 + 0 with a single disk failure.

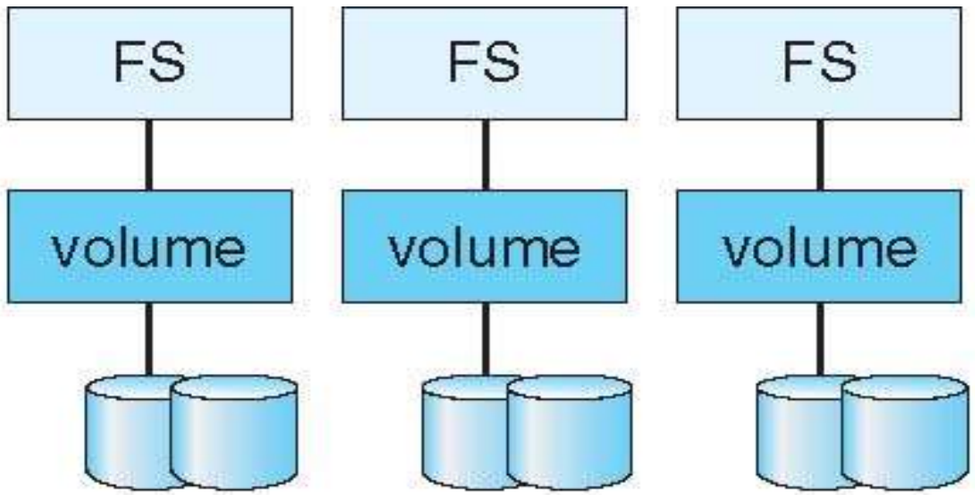
Extensions

- 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, partititions
  - Disks allocated in **pools**
  - Filesystems with a pool share that pool, use and release space like “malloc” and “free” memory allocate / release calls

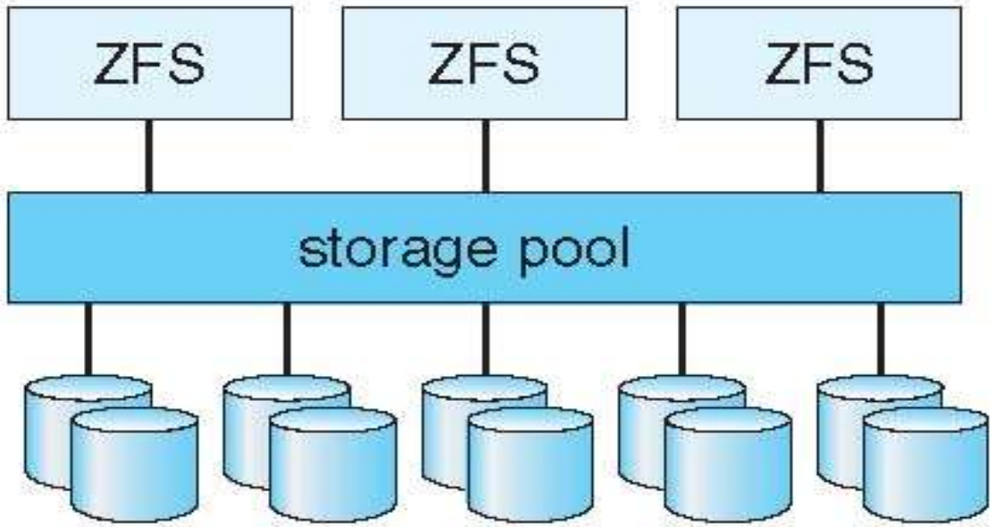
ZFS Checksums All Metadata and Data



Traditional and Pooled Storage



(a) Traditional volumes and file systems.



(b) ZFS and pooled storage.

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 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

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

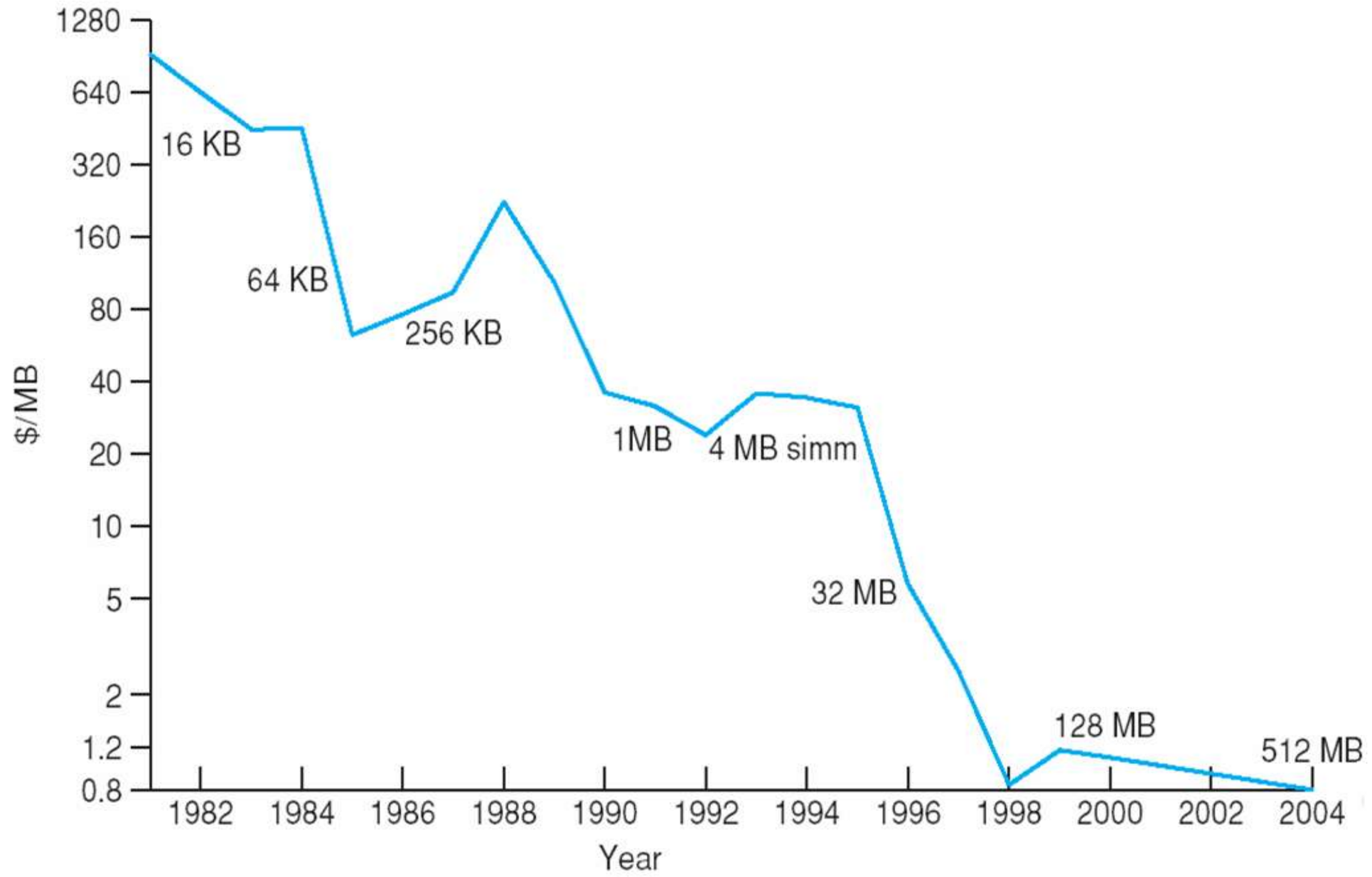
Reliability

- 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

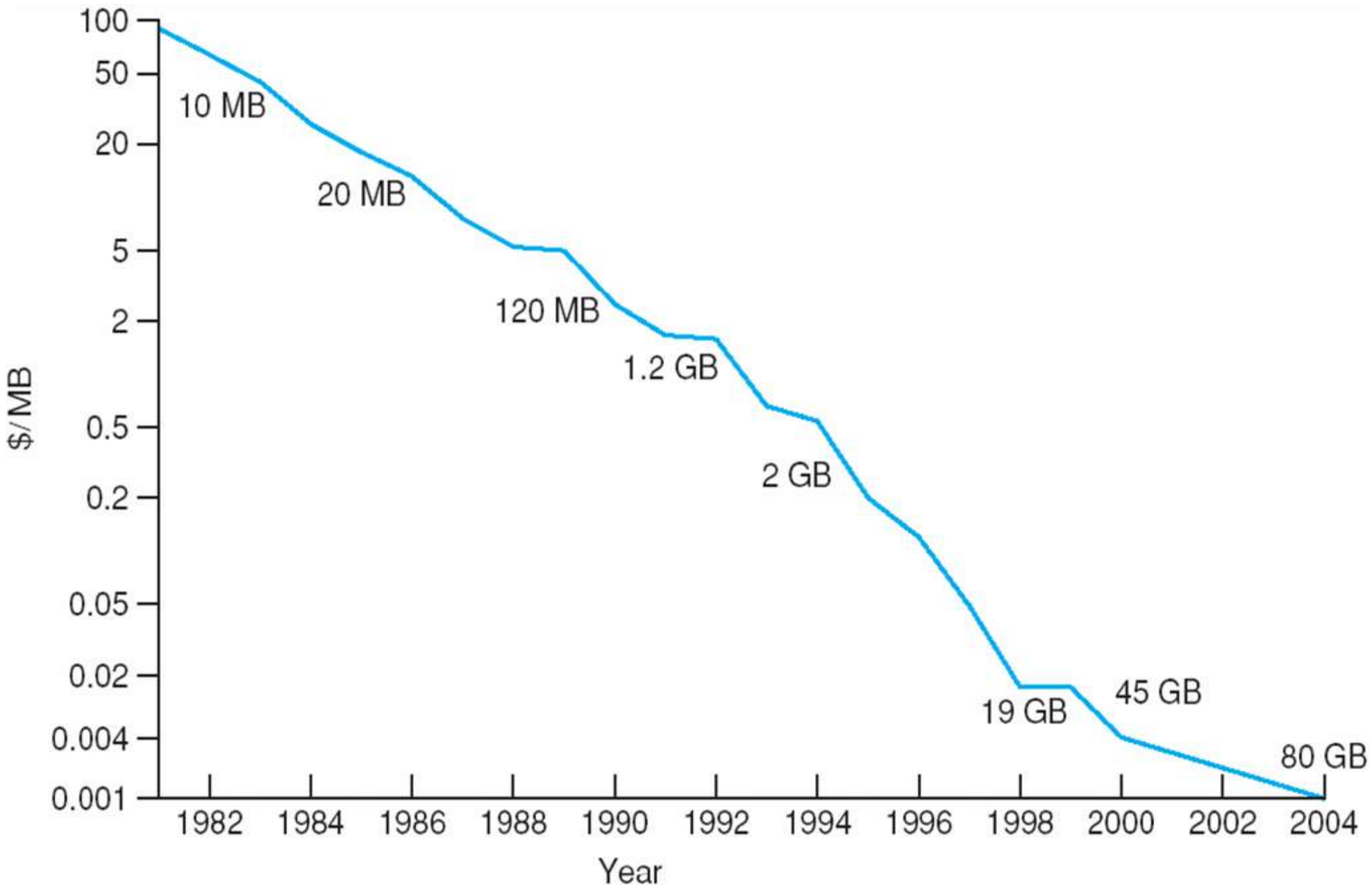
Cost

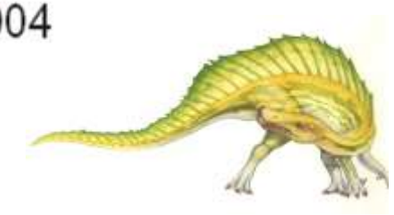
- Main memory is much more expensive than disk storage
- The cost per megabyte of hard disk storage is competitive with magnetic tape if only one tape is used per drive
- The cheapest tape drives and the cheapest disk drives have had about the same storage capacity over the years
- Tertiary storage gives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

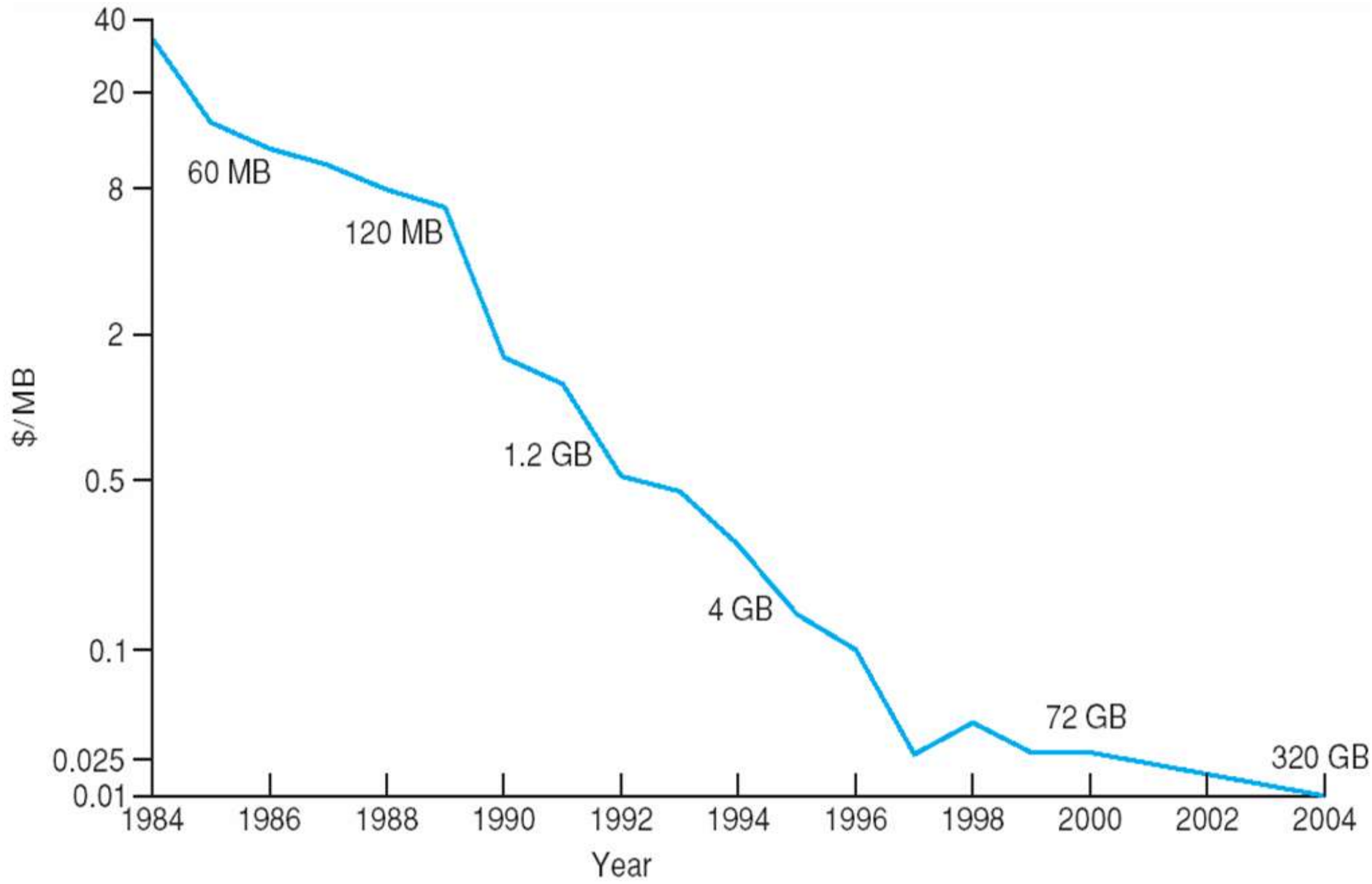


Price per Megabyte of Magnetic Hard Disk From 1981 to 2004

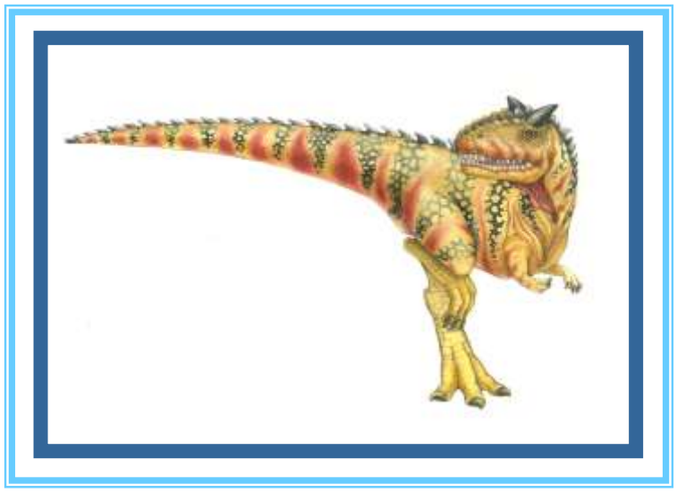




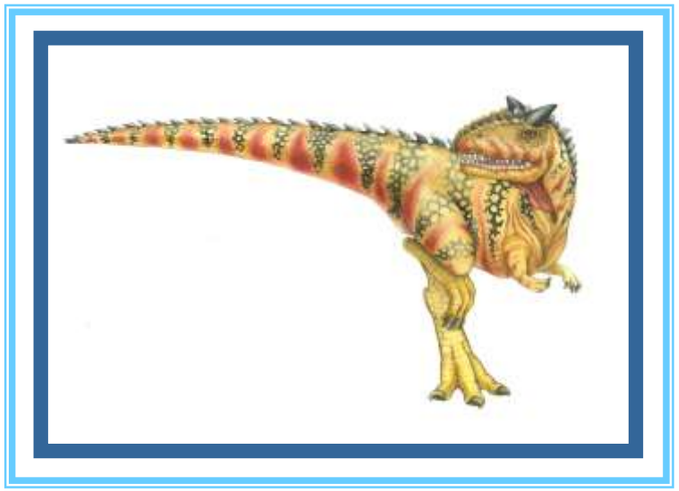
Price per Megabyte of a Tape Drive From 1984-2000



End of Chapter 12



Chapter 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
- NFS
- Example: WAFL File System

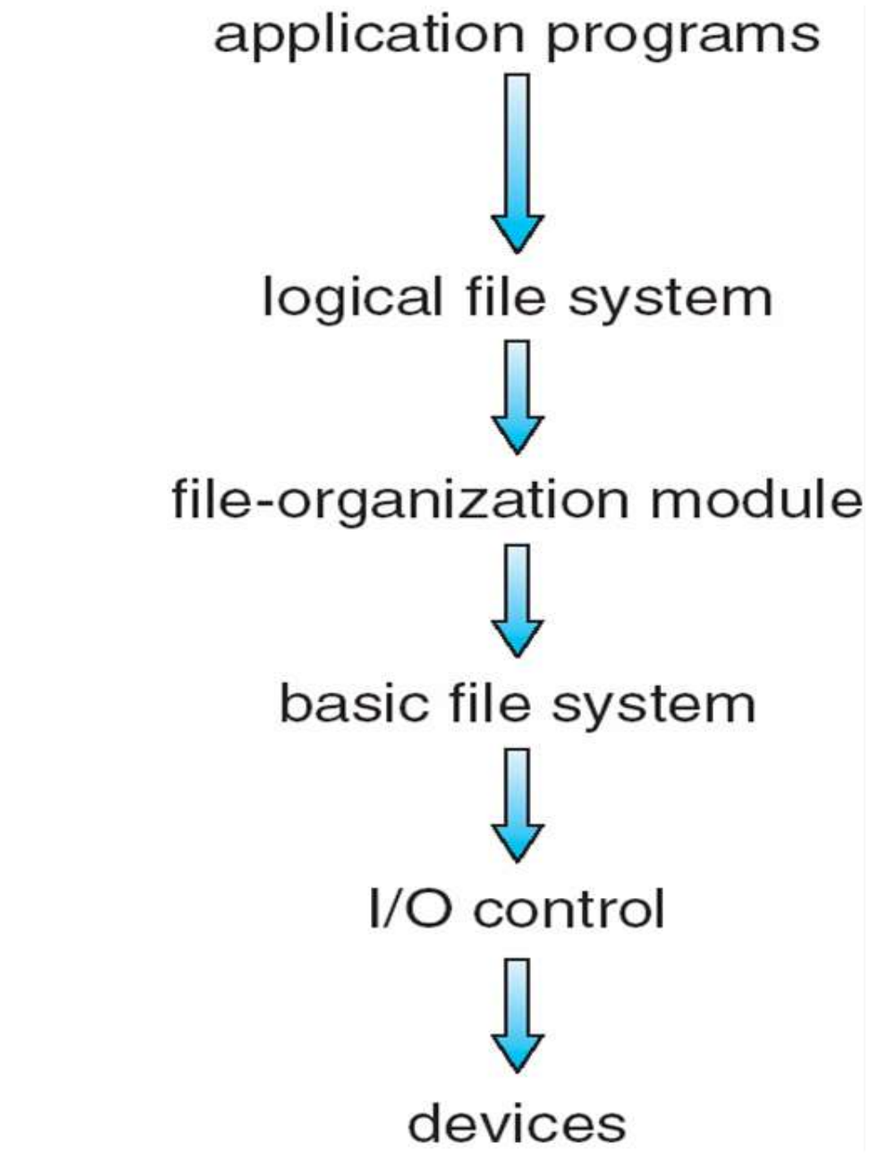
Objectives

- 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)
  - Provided user interface to storage, mapping logical to physical
  - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily
- Disk provides in-place rewrite and random access
  - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- **File control block** – storage structure consisting of information about a file
- **Device driver** controls the physical device
- File system organized into layers

Layered File System



File System Layers

- **Device drivers** manage I/O devices at the I/O control layer
  - Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level hardware specific commands to hardware controller
  - **Basic file system** given command like “retrieve block 123” translates to device driver
- Also manages memory buffers and caches (allocation, freeing, replacement)
  - Buffers hold data in transit
  - Caches hold frequently used data
  - **File organization module** understands files, logical address, and physical blocks
- Translates logical block # to physical block #
- Manages free space, disk allocation

File System Layers (Cont.)

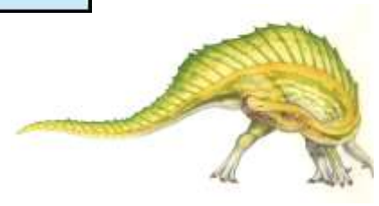
- **Logical file system** manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in Unix)
  - Directory management
  - Protection
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
  - Logical layers can be implemented by any coding method according to OS designer
- Many file systems, sometimes many within an operating system
  - Each with its own format (CD-ROM is ISO 9660; Unix has **UFS**, FFS; Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray, Linux has more than 40 types, with **extended file system** ext2 and ext3 leading; plus distributed file systems, etc)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE

File-System Implementation

- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures
- **Boot control block** contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- **Volume control block (superblock, master file table)** contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure organizes the files
  - Names and inode numbers, master file table
- Per-file **File Control Block (FCB)** contains many details about the file
  - Inode number, permissions, size, dates
  - NFTS stores into in master file table using relational DB structures

A Typical File Control Block

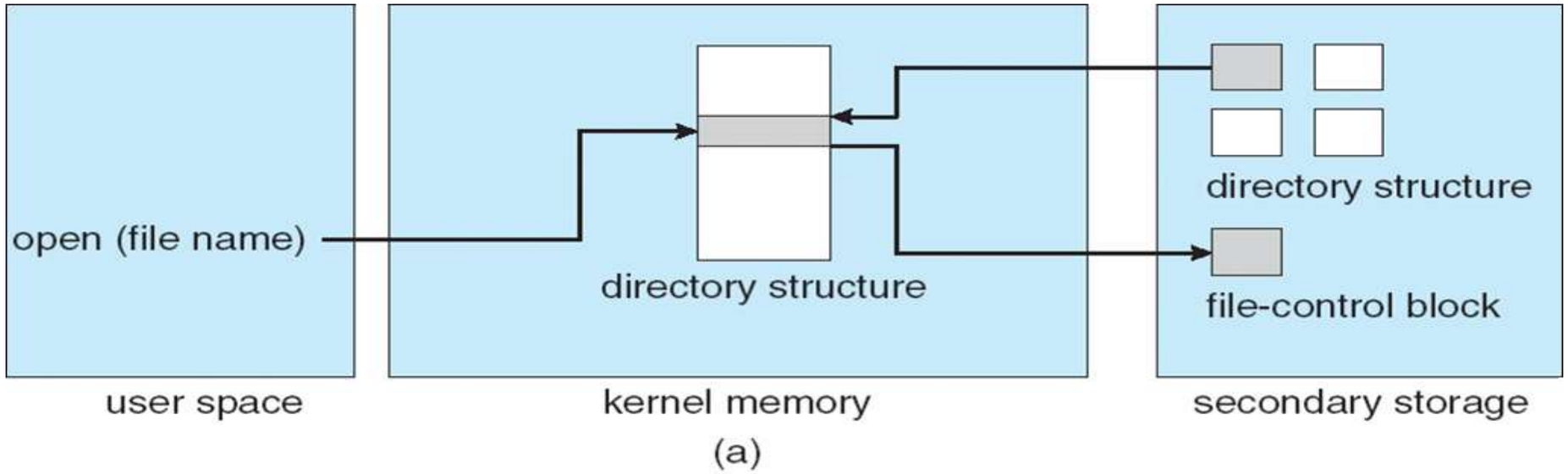
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks



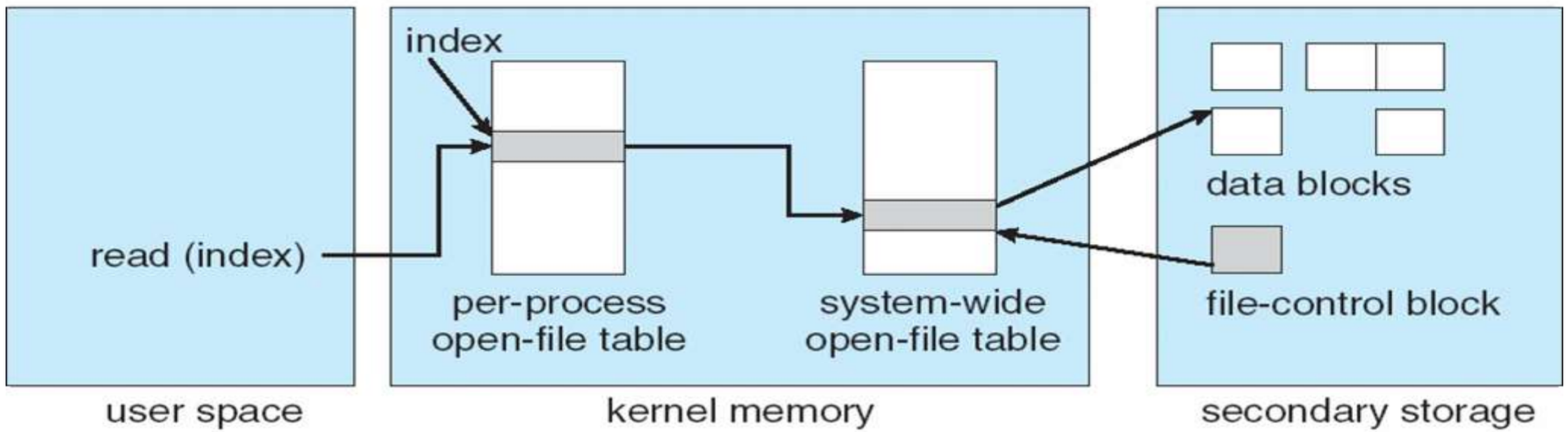
In-Memory File System Structures

- Mount table storing file system mounts, mount points, file system types
- The following figure illustrates the necessary file system structures provided by the operating systems
- Figure 12-3(a) refers to opening a file
- Figure 12-3(b) refers to reading a file
- Plus buffers hold data blocks from secondary storage
- Open returns a file handle for subsequent use
- Data from read eventually copied to specified user process memory address

In-Memory File System Structures



(a)



(b)

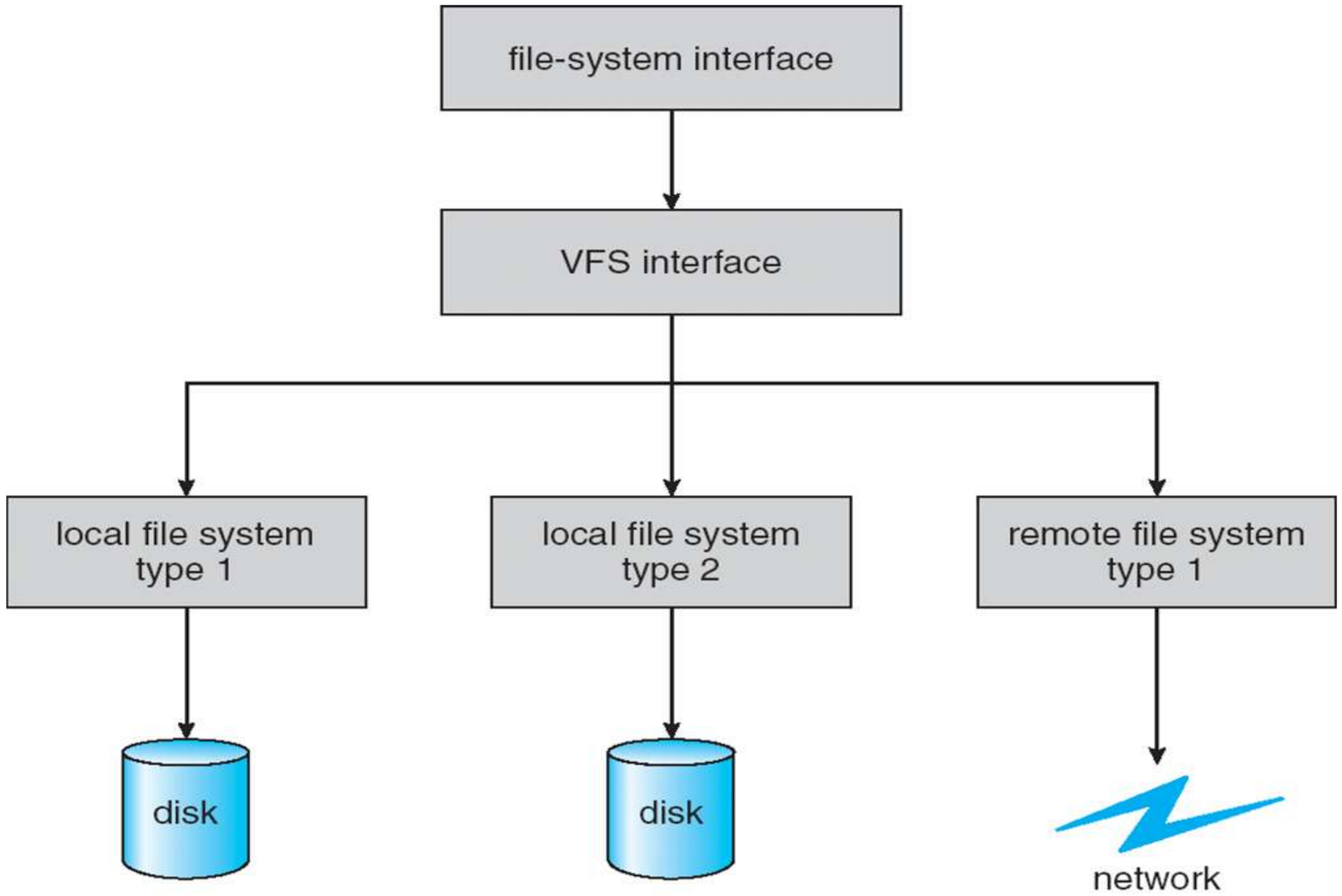
Partitions and Mounting

- Partition can be a volume containing a file system (“cooked”) or **raw** – just a sequence of blocks with no file system
- Boot block can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system
  - Or a boot management program for multi-os booting
- **Root partition** contains the OS, other partitions can hold other Oses, other file systems, or be raw
  - Mounted at boot time
  - Other partitions can mount automatically or manually
- At mount time, file system consistency checked
  - Is all metadata correct?
    - If not, fix it, try again
    - If yes, add to mount table, allow access

Virtual File Systems

- Virtual File Systems (VFS) on Unix provide an object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - Separates file-system generic operations from implementation details
  - Implementation can be one of many file systems types, or network file system
    - Implements vnodes which hold inodes or network file details
  - Then dispatches operation to appropriate file system implementation routines
- The API is to the VFS interface, rather than any specific type of file system

Schematic View of Virtual File System



Virtual File System Implementation

- For example, Linux has four object types:
  - inode, file, superblock, dentry
- VFS defines set of operations on the objects that must be implemented
  - Every object has a pointer to a function table
    - Function table has addresses of routines to implement that function on that object

Directory Implementation

- **Linear list** of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree
- **Hash Table** – linear list with hash data structure
  - Decreases directory search time
  - **Collisions** – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method

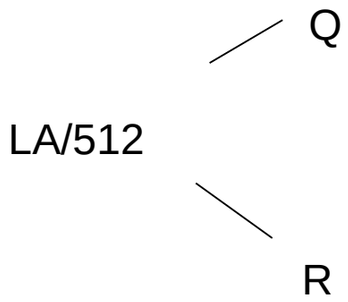
Allocation Methods - Contiguous

- An allocation method refers to how disk blocks are allocated for files:

- **Contiguous allocation** – each file occupies set of contiguous blocks
  - Best performance in most cases
  - Simple – only starting location (block #) and length (number of blocks) are required
  - Problems include finding space for file, knowing file size, external fragmentation, need for **compaction off-line** (**downtime**) or **on-line**

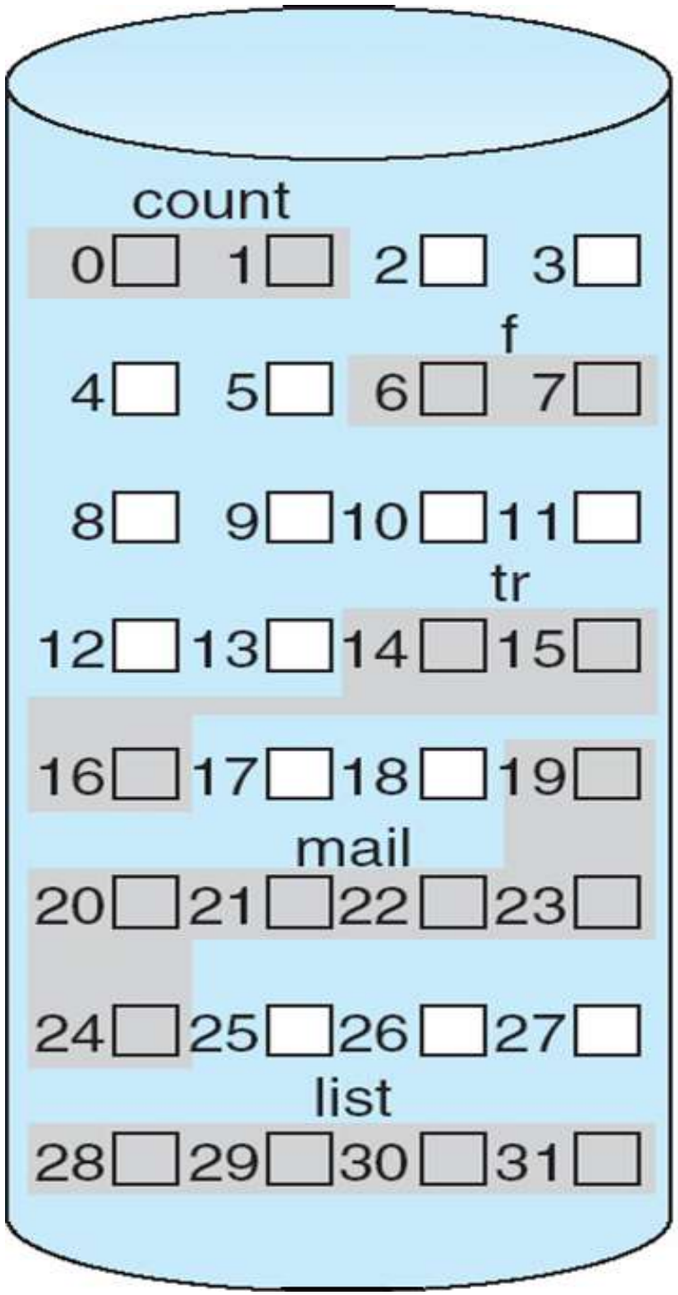
Contiguous Allocation

- Mapping from logical to physical



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

Contiguous Allocation of Disk Space



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Extent-Based Systems

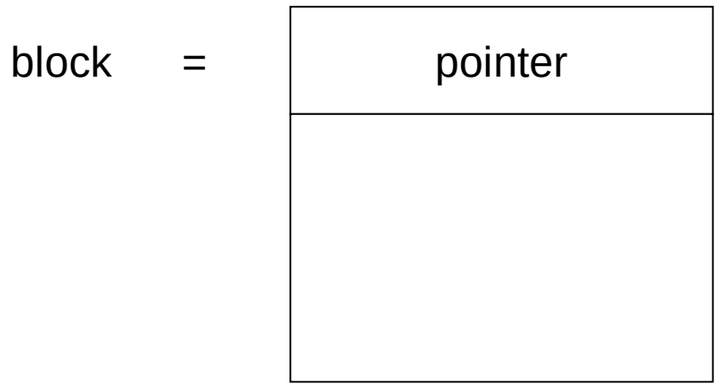
- Many newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in extents
- An **extent** is a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents

Allocation Methods - Linked

- **Linked allocation** – each file a linked list of blocks
  - File ends at nil pointer
  - No external fragmentation
  - Each block contains pointer to next block
  - No compaction, external fragmentation
  - Free space management system called when new block needed
  - Improve efficiency by clustering blocks into groups but increases internal fragmentation
  - Reliability can be a problem
  - Locating a block can take many I/Os and disk seeks
- FAT (File Allocation Table) variation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple

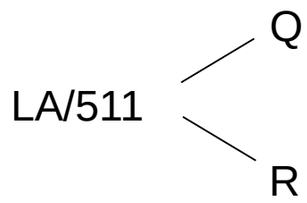
Linked Allocation

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



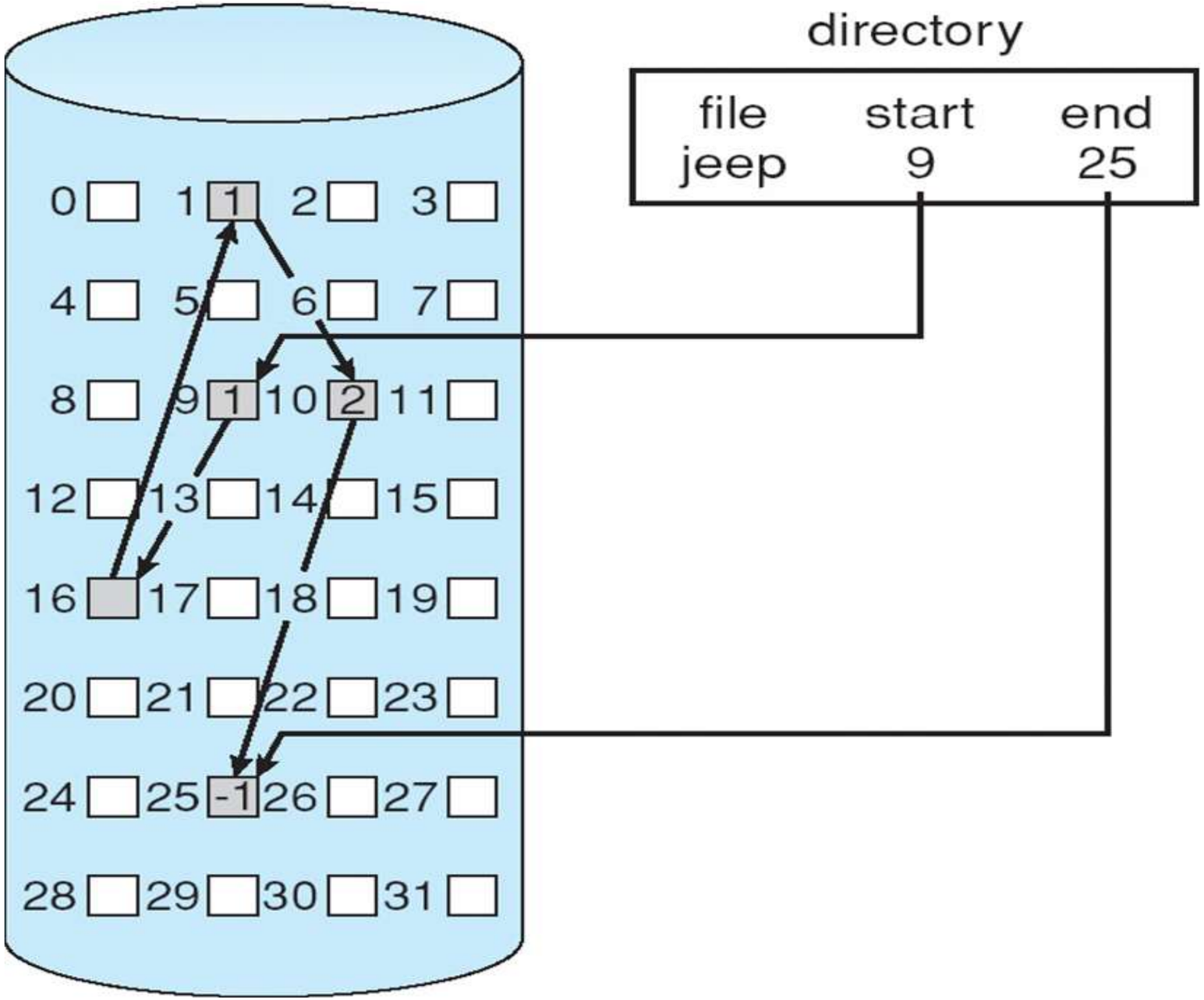
Linked Allocation

- Mapping

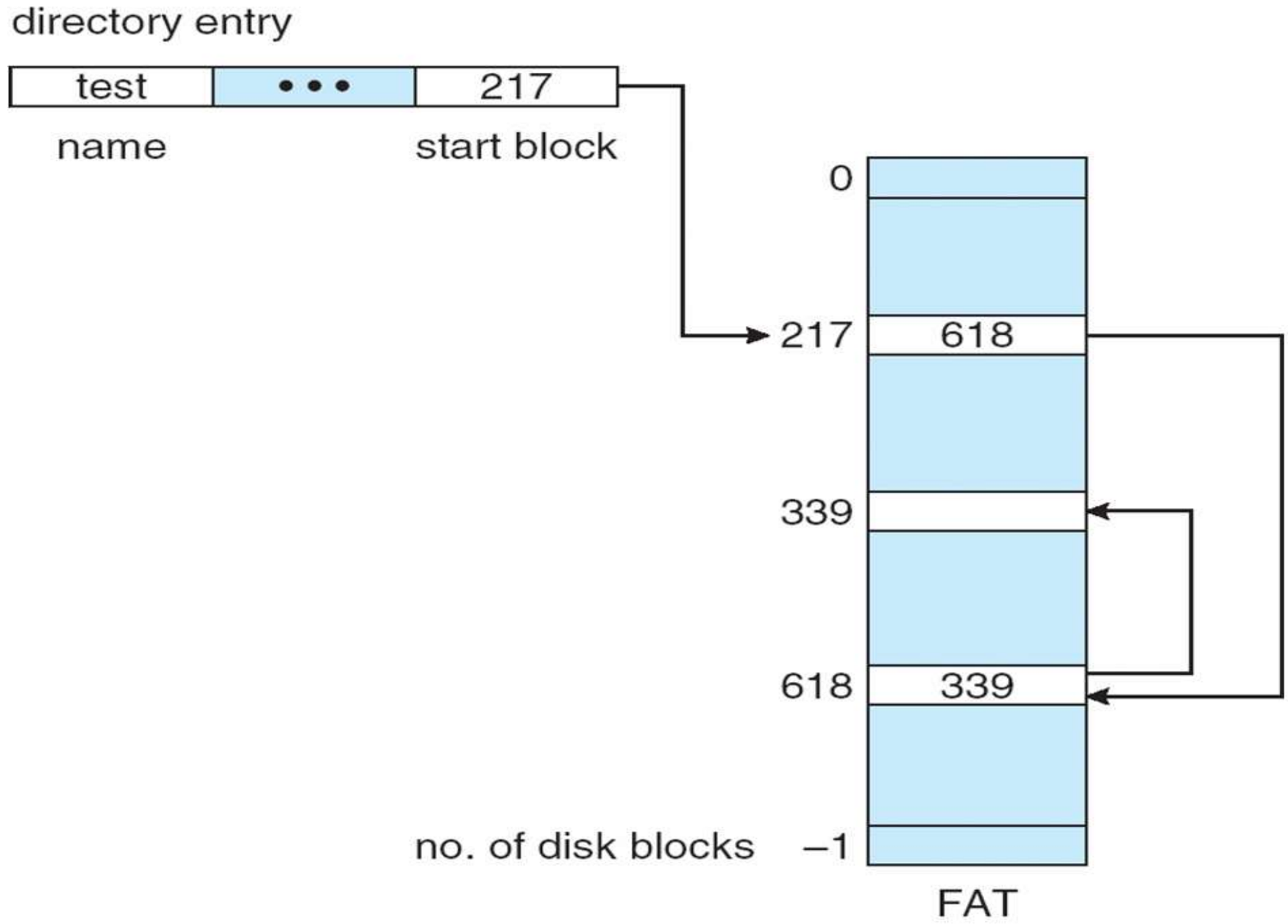


Block to be accessed is the Qth block in the linked chain of blocks representing the file.
Displacement into block = R + 1

Linked Allocation



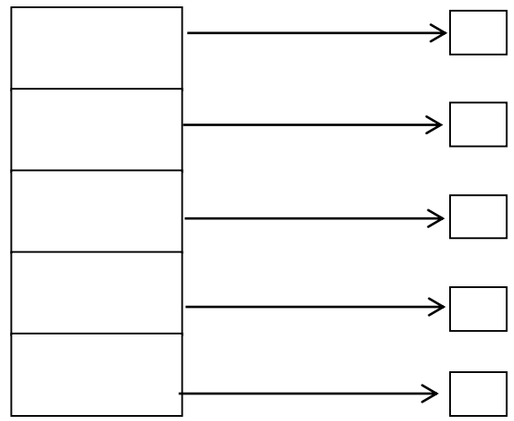
File-Allocation Table



Allocation Methods - Indexed

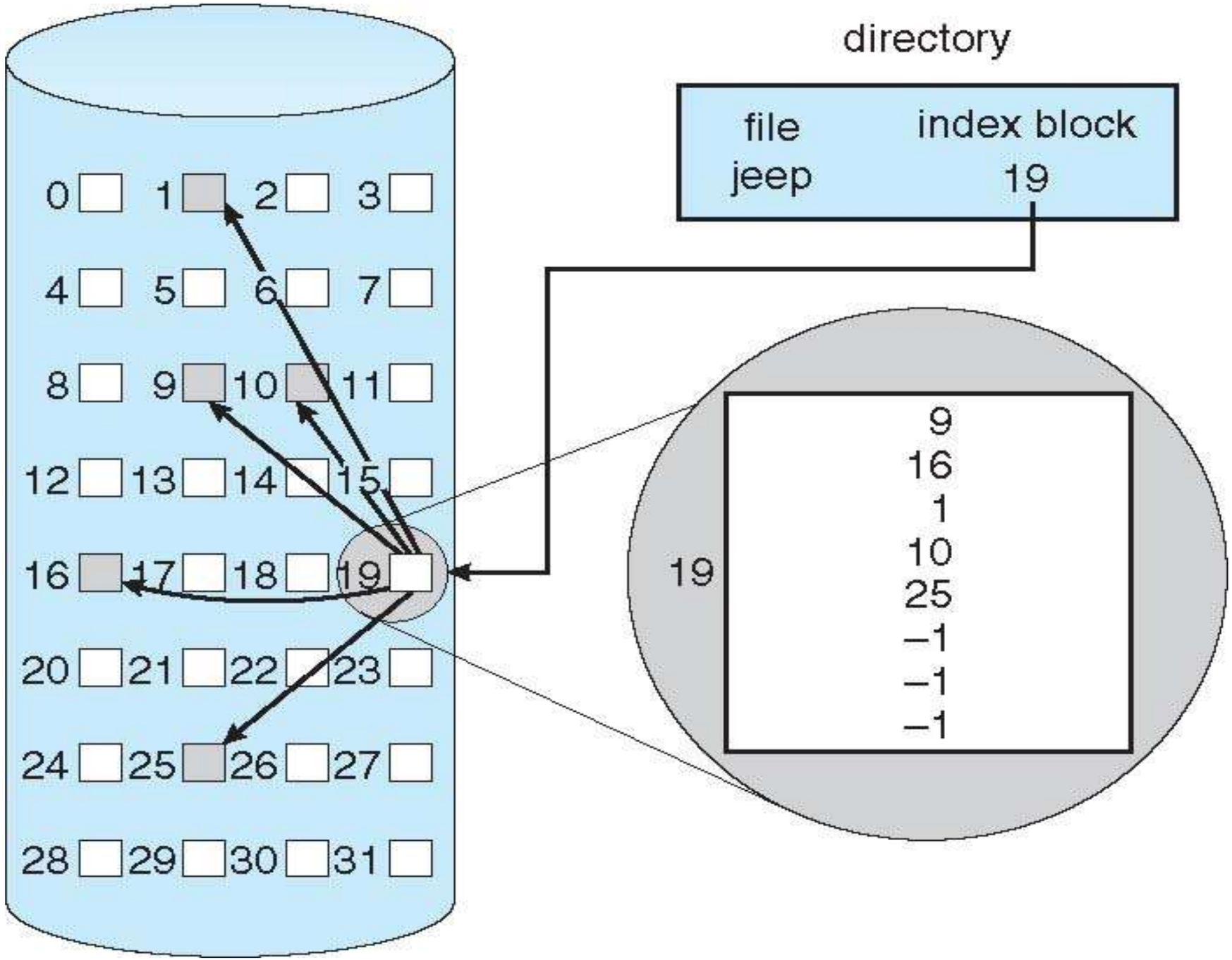
- **Indexed allocation**
  - Each file has its own **index block**(s) of pointers to its data blocks

- Logical view



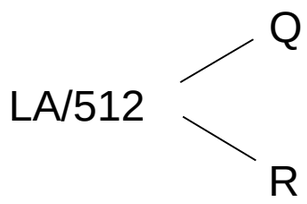
index table

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 bytes and block size of 512 bytes. 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)

$$LA / (512 \times 511) \begin{cases} Q_1 \\ R_1 \end{cases}$$

Q_1 = block of index table
R_1 is used as follows:

$$R_1 / 512 \begin{cases} Q_2 \\ R_2 \end{cases}$$

Q_2 = displacement into block of index table
R_2 displacement into block of file:

Indexed Allocation – Mapping (Cont.)

- Two-level index (4K blocks could store 1,024 four-byte pointers in outer index -> 1,048,567 data blocks and file size of up to 4GB)

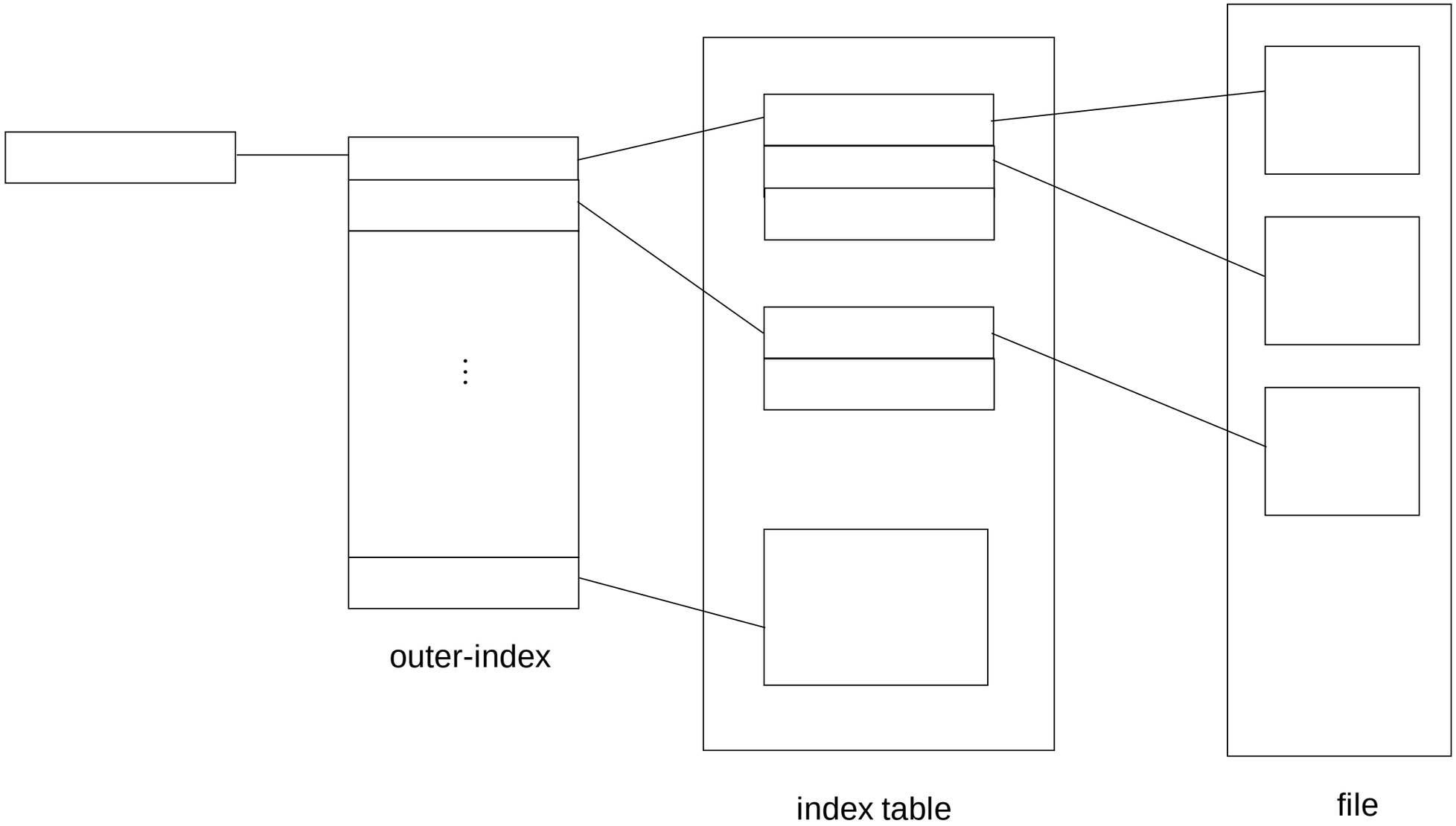
$$LA / (512 \times 512) \begin{cases} Q_1 \\ R_1 \end{cases}$$

Q_1 = displacement into outer-index
R_1 is used as follows:

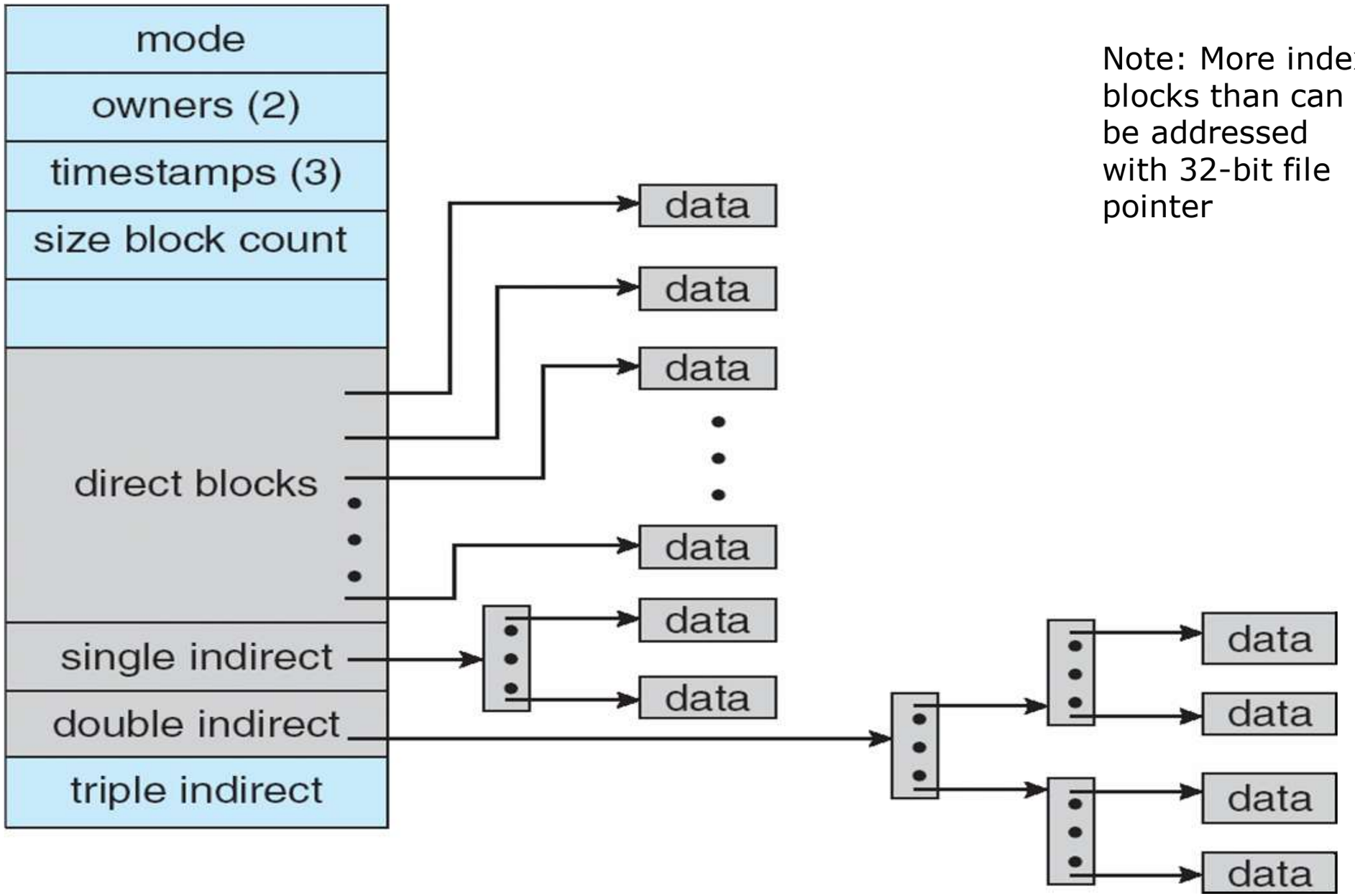
$$R_1 / 512 \begin{cases} Q_2 \\ R_2 \end{cases}$$

Q_2 = displacement into block of index table
R_2 displacement into block of file:

Indexed Allocation – Mapping (Cont.)



Combined Scheme: UNIX UFS (4K bytes per block, 32-bit addresses)



Note: More index blocks than can be addressed with 32-bit file pointer

Performance

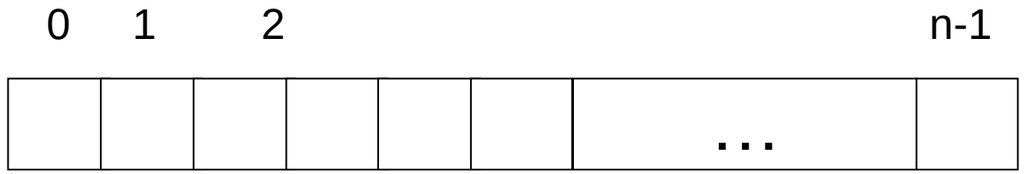
- Best method depends on file access type
  - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation -> select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads then data block read
  - Clustering can help improve throughput, reduce CPU overhead

Performance (Cont.)

- Adding instructions to the execution path to save one disk I/O is reasonable
  - Intel Core i7 Extreme Edition 990x (2011) at 3.46Ghz = 159,000 MIPS
    - http://en.wikipedia.org/wiki/Instructions_per_second
  - Typical disk drive at 250 I/Os per second
    - 159,000 MIPS / 250 = 630 million instructions during one disk I/O
  - Fast SSD drives provide 60,000 IOPS
    - 159,000 MIPS / 60,000 = 2.65 millions instructions during one disk I/O

Free-Space Management

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



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

Block number calculation

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

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

Free-Space Management (Cont.)

- Bit map requires extra space
  - Example:

    block size = 4KB = 2^{12} bytes

    disk size = 2^{40} bytes (1 terabyte)

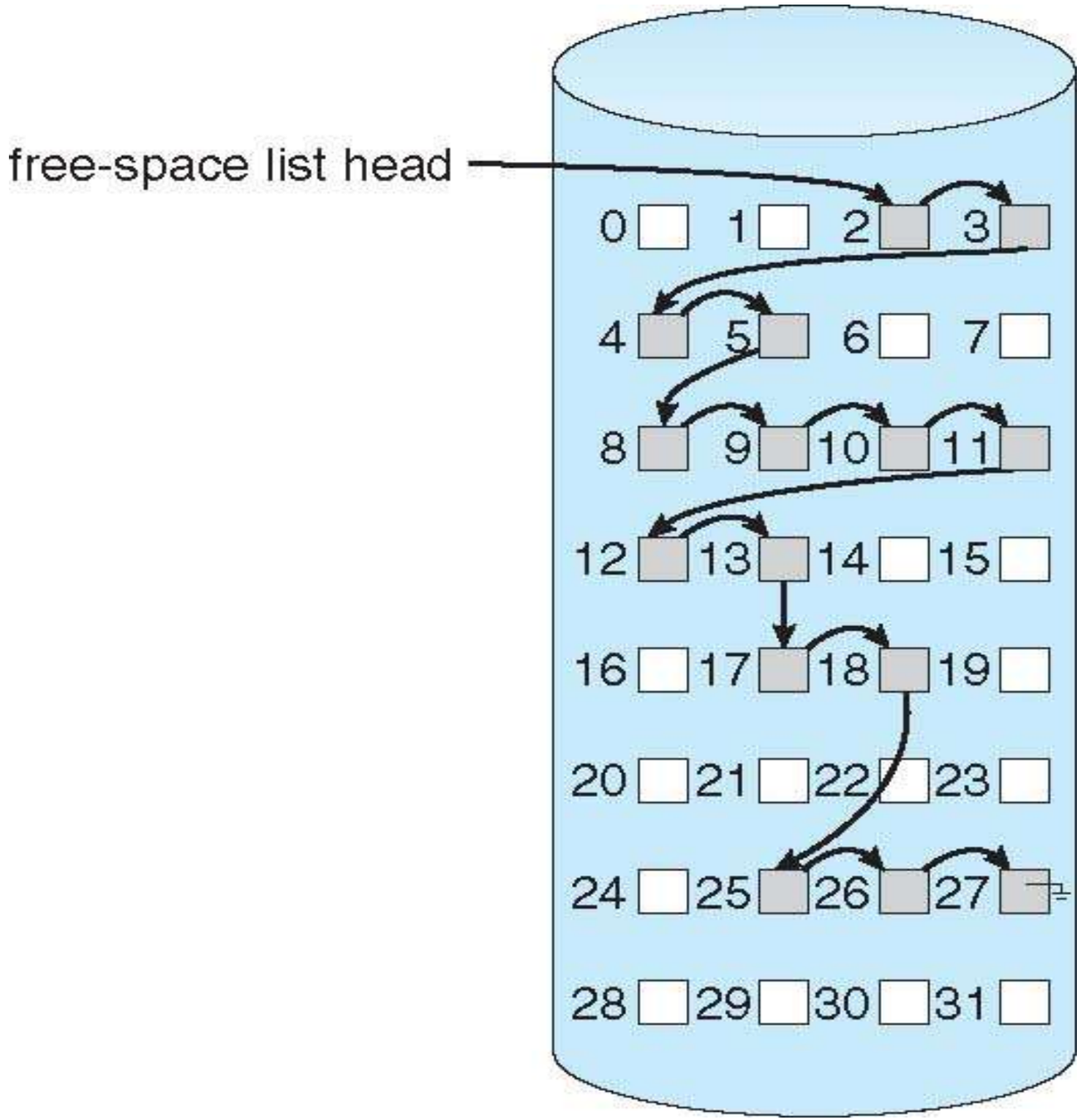
    $n = 2^{40}/2^{12} = 2^{28}$ bits (or 256 MB)

    if clusters of 4 blocks -> 64MB of memory

- Easy to get contiguous files

- Linked list (free list)
  - Cannot get contiguous space easily
  - No waste of space
  - No need to traverse the entire list (if # free blocks recorded)

Linked Free Space List on Disk



Free-Space Management (Cont.)

- Grouping
  - Modify linked list to store address of next *n-1* free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this one)
- Counting
  - Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering
    - Keep address of first free block and count of following free blocks
    - Free space list then has entries containing addresses and counts

Free-Space Management (Cont.)

- Space Maps
  - Used in ZFS
  - Consider meta-data I/O on very large file systems
    - Full data structures like bit maps couldn't fit in memory -> thousands of I/Os
  - Divides device space into **metaslab** units and manages metaslabs
    - Given volume can contain hundreds of metaslabs
  - Each metaslab has associated space map
    - Uses counting algorithm
  - But records to log file rather than file system
    - Log of all block activity, in time order, in counting format
  - Metaslab activity -> load space map into memory in balanced-tree structure, indexed by offset
    - Replay log into that structure
    - Combine contiguous free blocks into single entry

Efficiency and Performance

- Efficiency dependent on:
  - Disk allocation and directory algorithms
  - Types of data kept in file's directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures

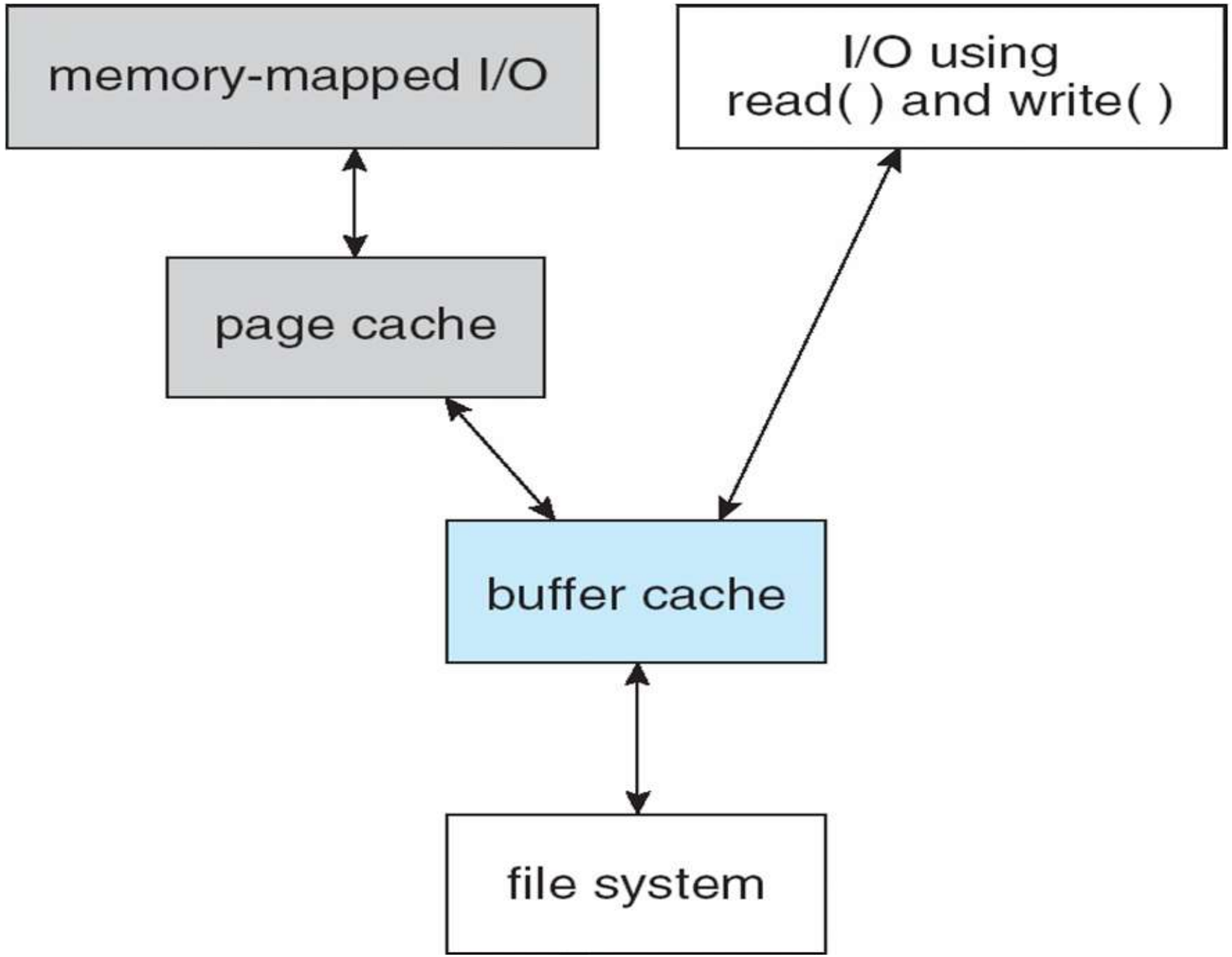
Efficiency and Performance (Cont.)

- Performance
  - Keeping data and metadata close together
  - **Buffer cache** – separate section of main memory for frequently used blocks
  - **Synchronous** writes sometimes requested by apps or needed by OS
    - No buffering / caching – writes must hit disk before acknowledgement
    - **Asynchronous** writes more common, buffer-able, faster
  - **Free-behind** and **read-ahead** – techniques to optimize sequential access
  - Reads frequently slower than writes

Page Cache

- A **page cache** caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

I/O Without a Unified Buffer Cache

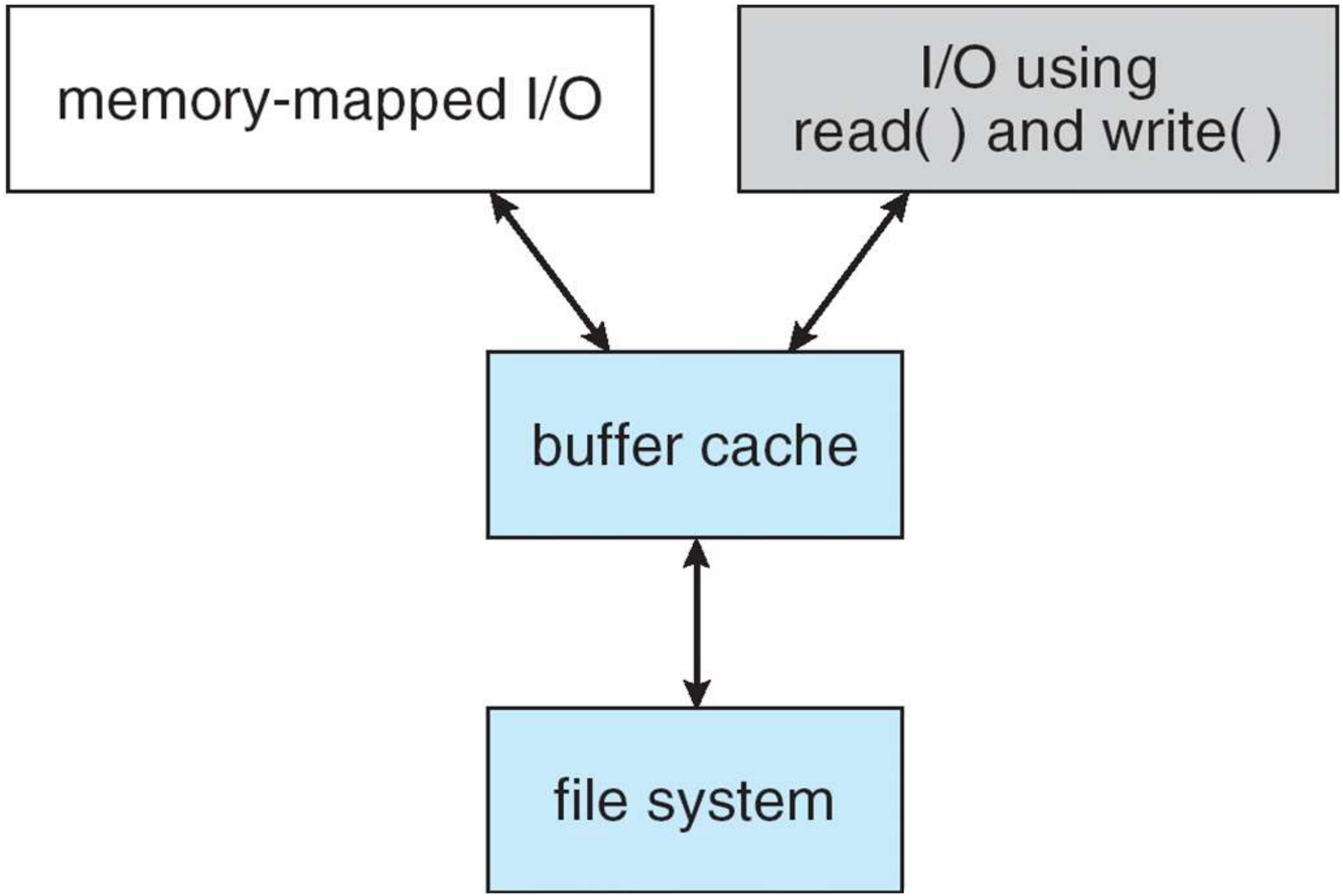


Unified Buffer Cache

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

- But which caches get priority, and what replacement algorithms to use?

I/O Using a Unified Buffer Cache



Recovery

- **Consistency checking** – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- Use system programs to **back up** data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by **restoring** data from backup

Log Structured File Systems

- **Log structured** (or **journaling**) file systems record each metadata update to the file system as a **transaction**
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

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

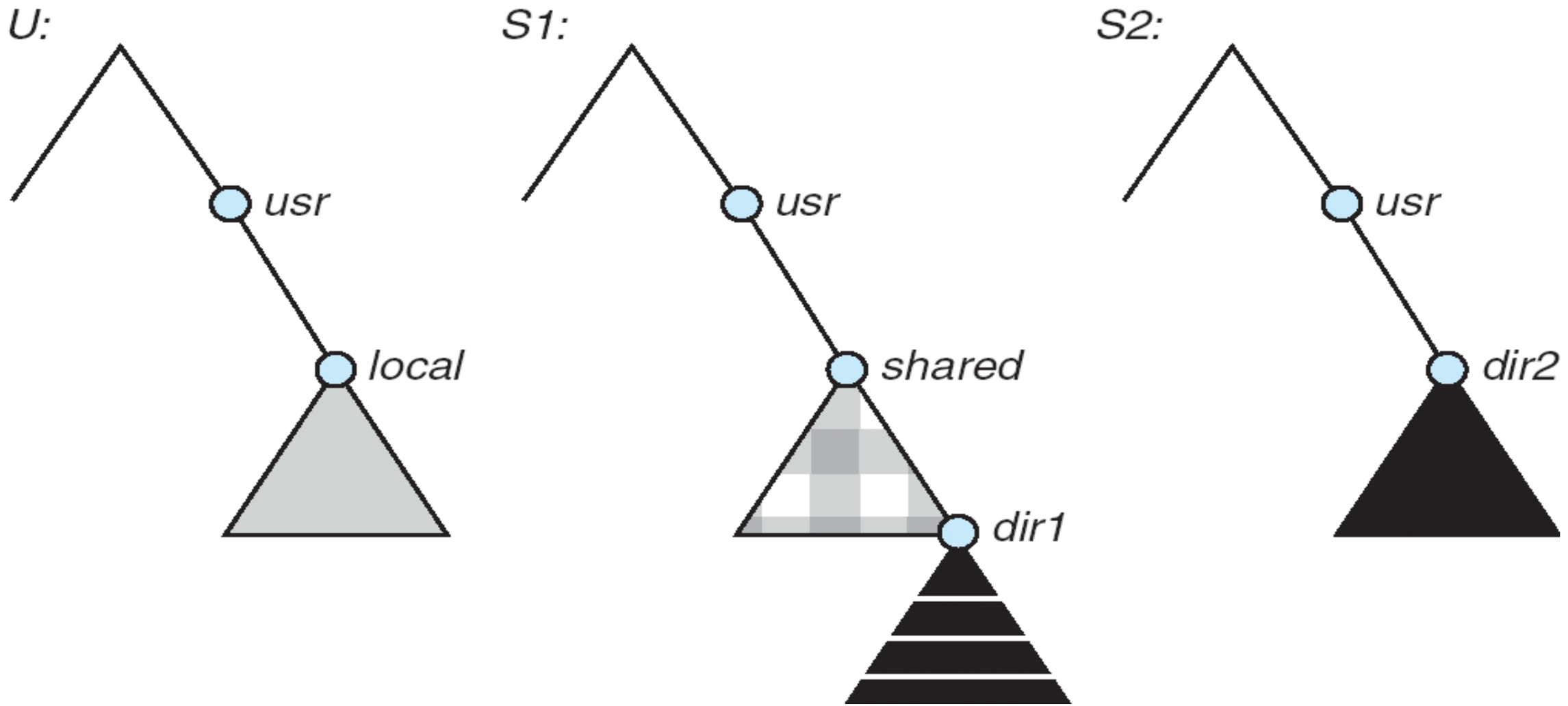
NFS (Cont.)

- 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

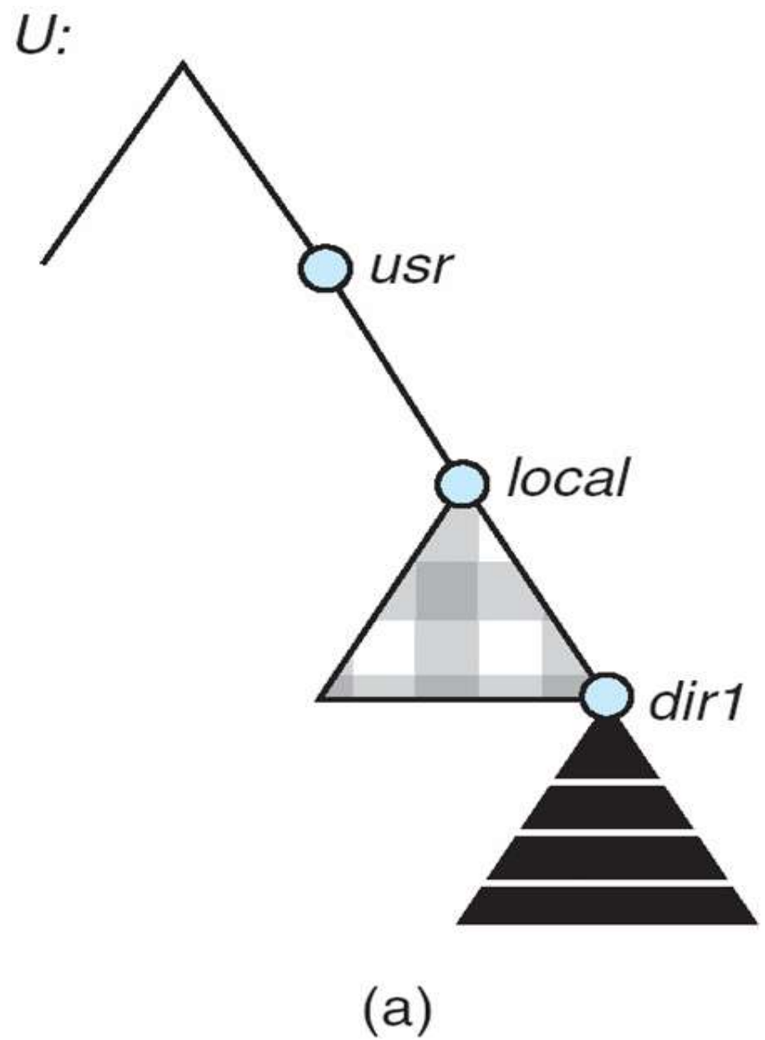
NFS (Cont.)

- 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

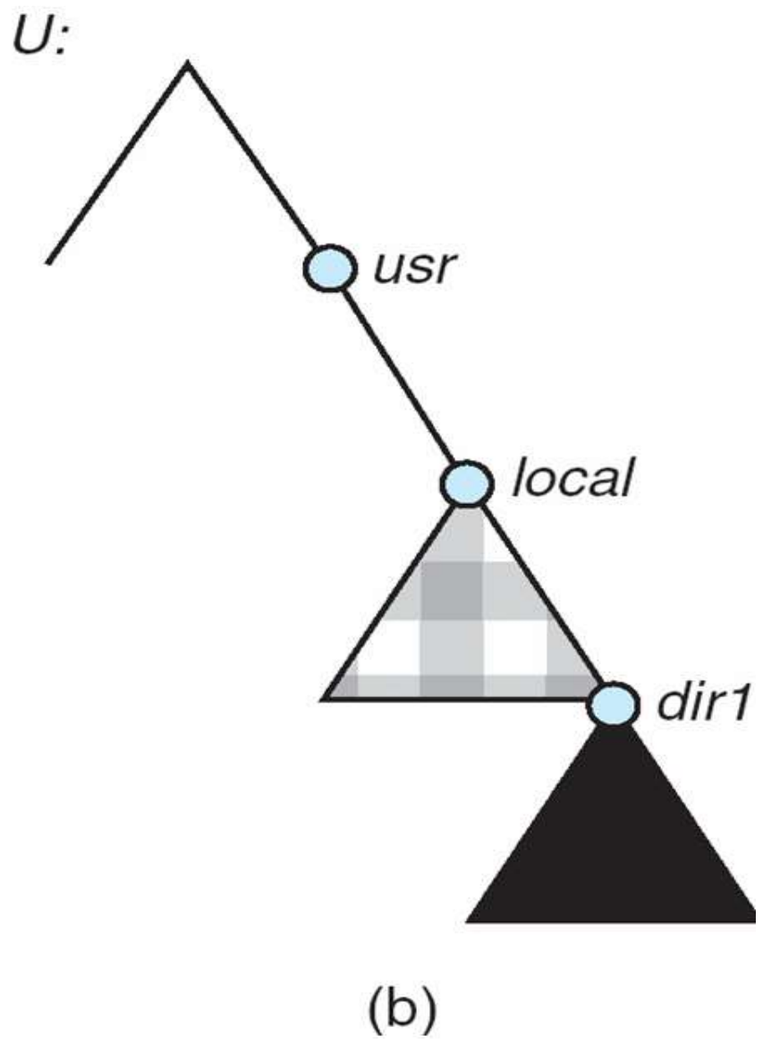


Mounting in NFS



(a)

Mounts



(b)

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

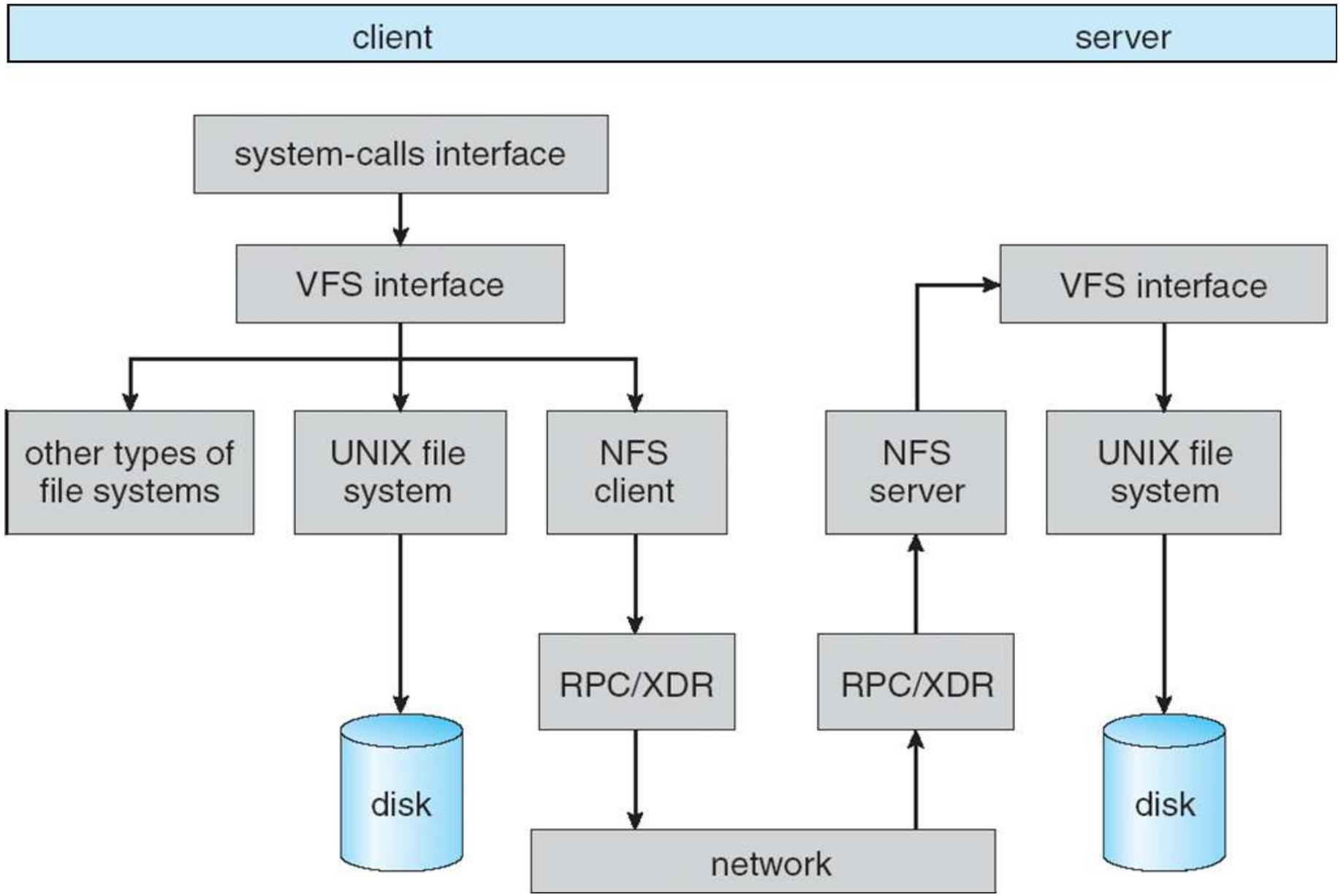
NFS Protocol

- 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

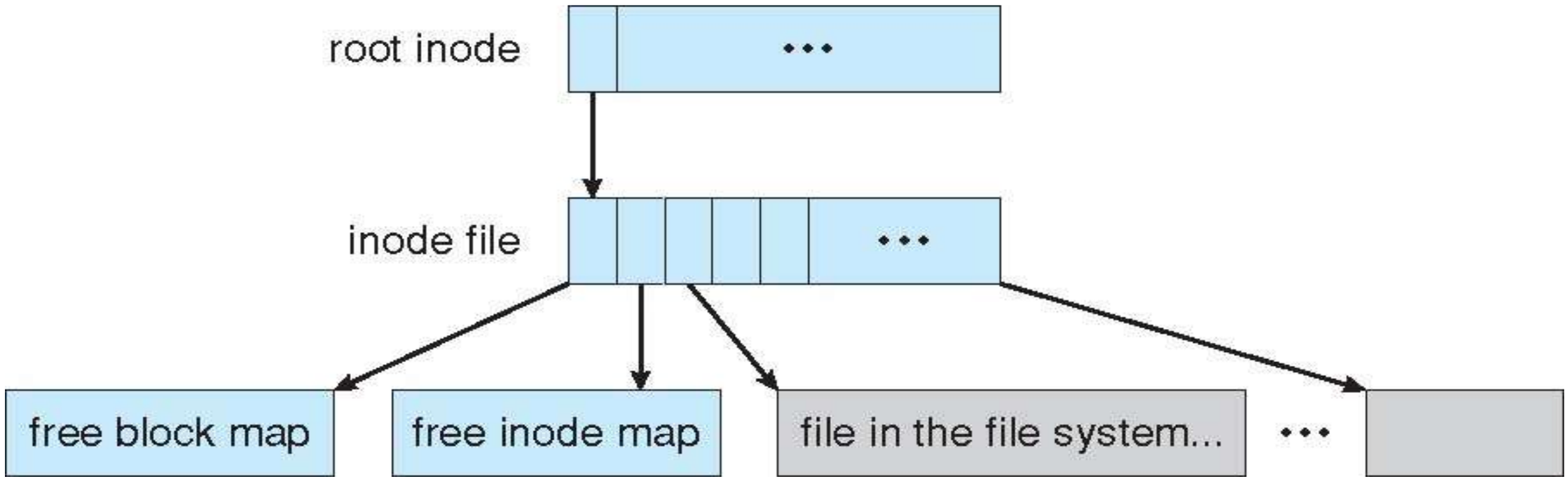
NFS Remote Operations

- 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

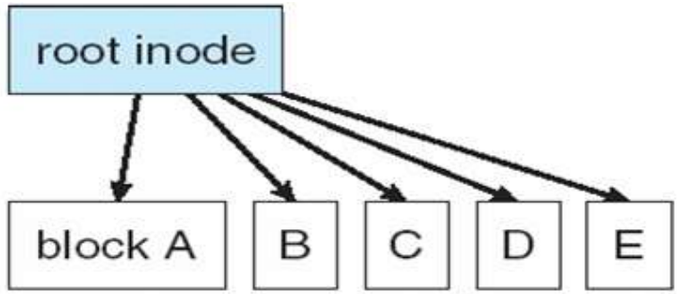
Example: WAFL File System

- 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

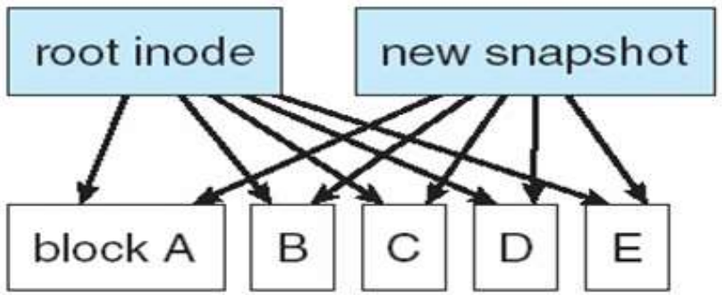
The WAFL File Layout



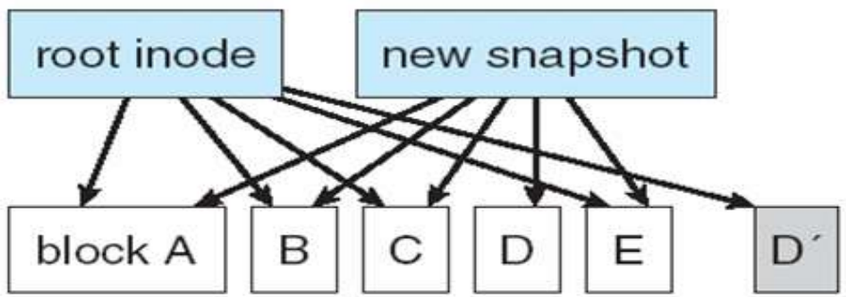
Snapshots in WAFL



(a) Before a snapshot.

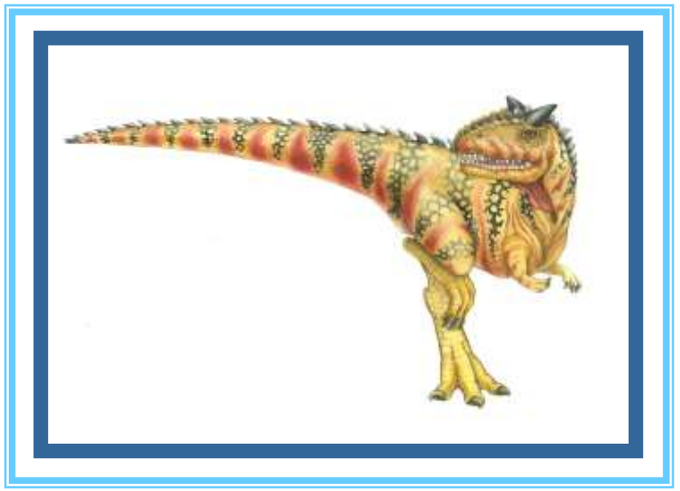


(b) After a snapshot, before any blocks change.



(c) After block D has changed to D´.

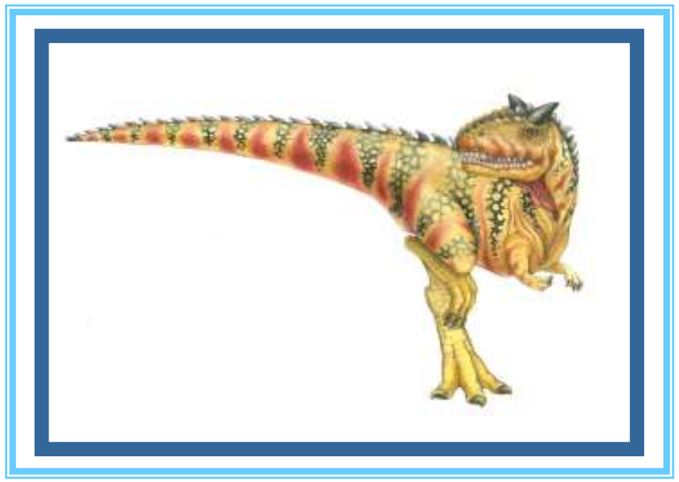
End of Chapter 10



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 block[*i*] to have a situation where bit[*i*] = 1 in memory and bit[*i*] = 0 on disk
  - Solution:
    - Set bit[*i*] = 1 in disk
    - Allocate block[*i*]
    - Set bit[*i*] = 1 in memory

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

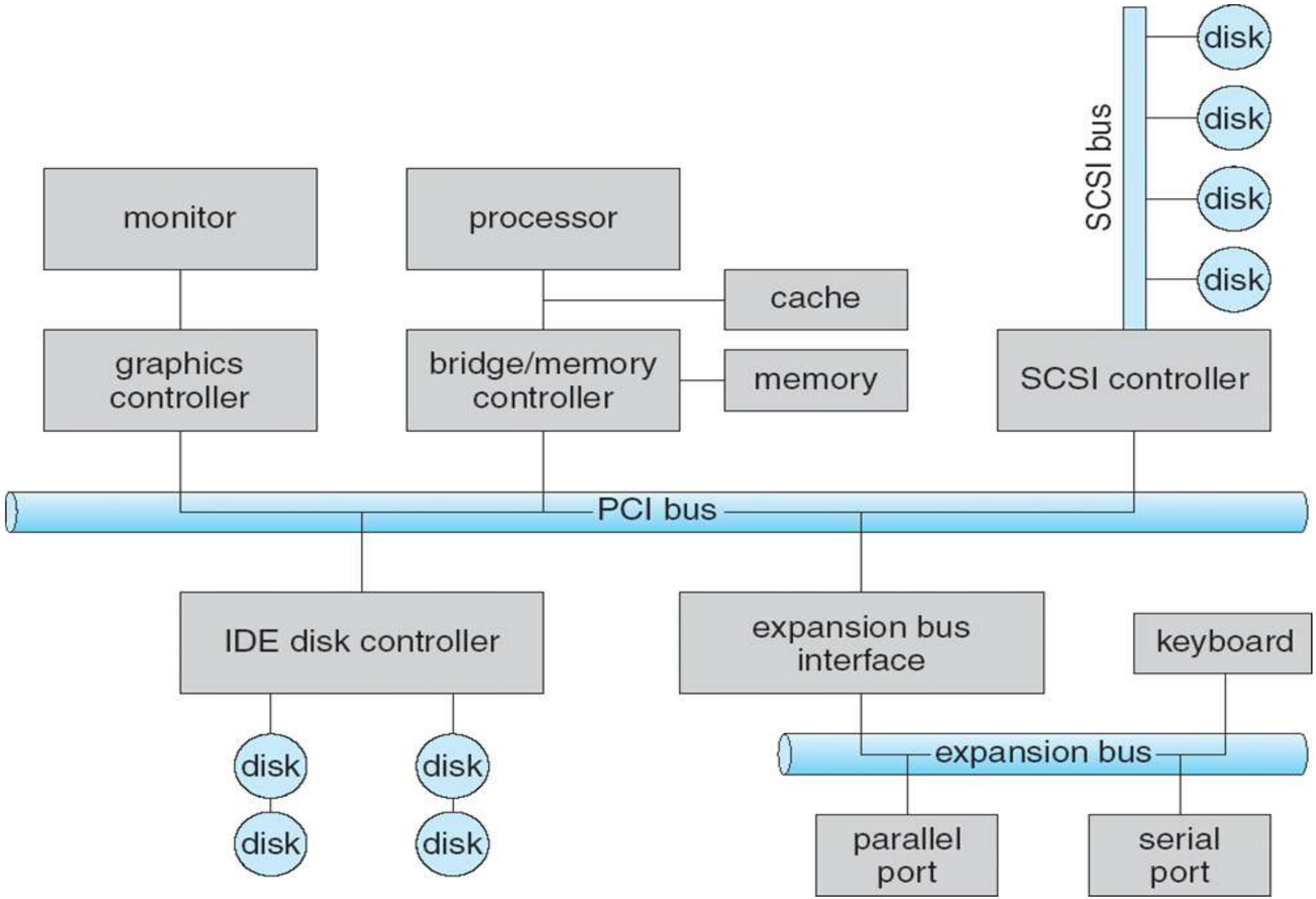
Overview

- I/O management is a major component of operating system design and operation
  - Important aspect of computer operation
  - I/O devices vary greatly
  - Various methods to control them
  - Performance management
  - New types of devices frequent
- Ports, busses, device controllers connect to various devices
- **Device drivers** encapsulate device details
  - Present uniform device-access interface to I/O subsystem

I/O Hardware

- Incredible variety of I/O devices
  - Storage
  - Transmission
  - Human-interface
- Common concepts – signals from I/O devices interface with computer
  - **Port** – connection point for device
  - **Bus** - **daisy chain** or shared direct access
  - **Controller** (**host adapter**) – electronics that operate port, bus, device
    - Sometimes integrated
    - Sometimes separate circuit board (host adapter)
    - Contains processor, microcode, private memory, bus controller, etc
      - Some talk to per-device controller with bus controller, microcode, memory, etc

A Typical PC Bus Structure



I/O Hardware (Cont.)

- I/O instructions control devices
- Devices usually have registers where device driver places commands, addresses, and data to write, or read data from registers after command execution
  - Data-in register, data-out register, status register, control register
  - Typically 1-4 bytes, or FIFO buffer
- Devices have addresses, used by
  - Direct I/O instructions
  - **Memory-mapped I/O**
    - Device data and command registers mapped to processor address space
    - Especially for large address spaces (graphics)

Device I/O Port Locations on PCs (partial)

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Polling

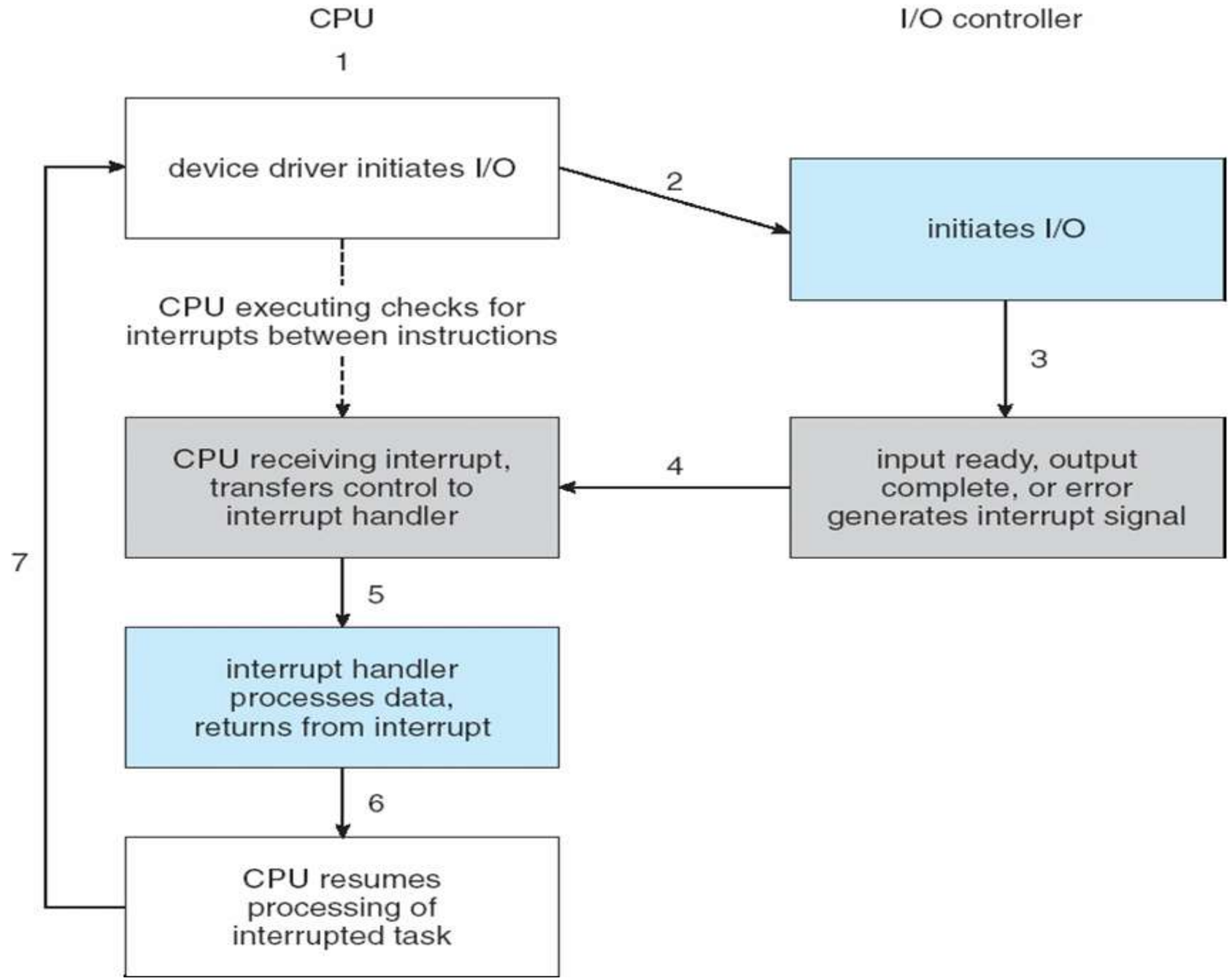
- For each byte of I/O
  1. Read busy bit from status register until 0
  2. Host sets read or write bit and if write copies data into data-out register
  3. Host sets command-ready bit
  4. Controller sets busy bit, executes transfer
  5. Controller clears busy bit, error bit, command-ready bit when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

- Step 1 is **busy-wait** cycle to wait for I/O from device
  - Reasonable if device is fast
  - But inefficient if device slow
  - CPU switches to other tasks?
    - But if miss a cycle data overwritten / lost

Interrupts

- Polling can happen in 3 instruction cycles
  - Read status, logical-and to extract status bit, branch if not zero
  - How to be more efficient if non-zero infrequently?
- CPU **Interrupt-request line** triggered by I/O device
  - Checked by processor after each instruction
- **Interrupt handler** receives interrupts
  - **Maskable** to ignore or delay some interrupts
- Interrupt vector to dispatch interrupt to correct handler
  - Context switch at start and end
  - Based on priority
  - Some **nonmaskable**
  - Interrupt chaining if more than one device at same interrupt number

Interrupt-Driven I/O Cycle



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

Interrupts (Cont.)

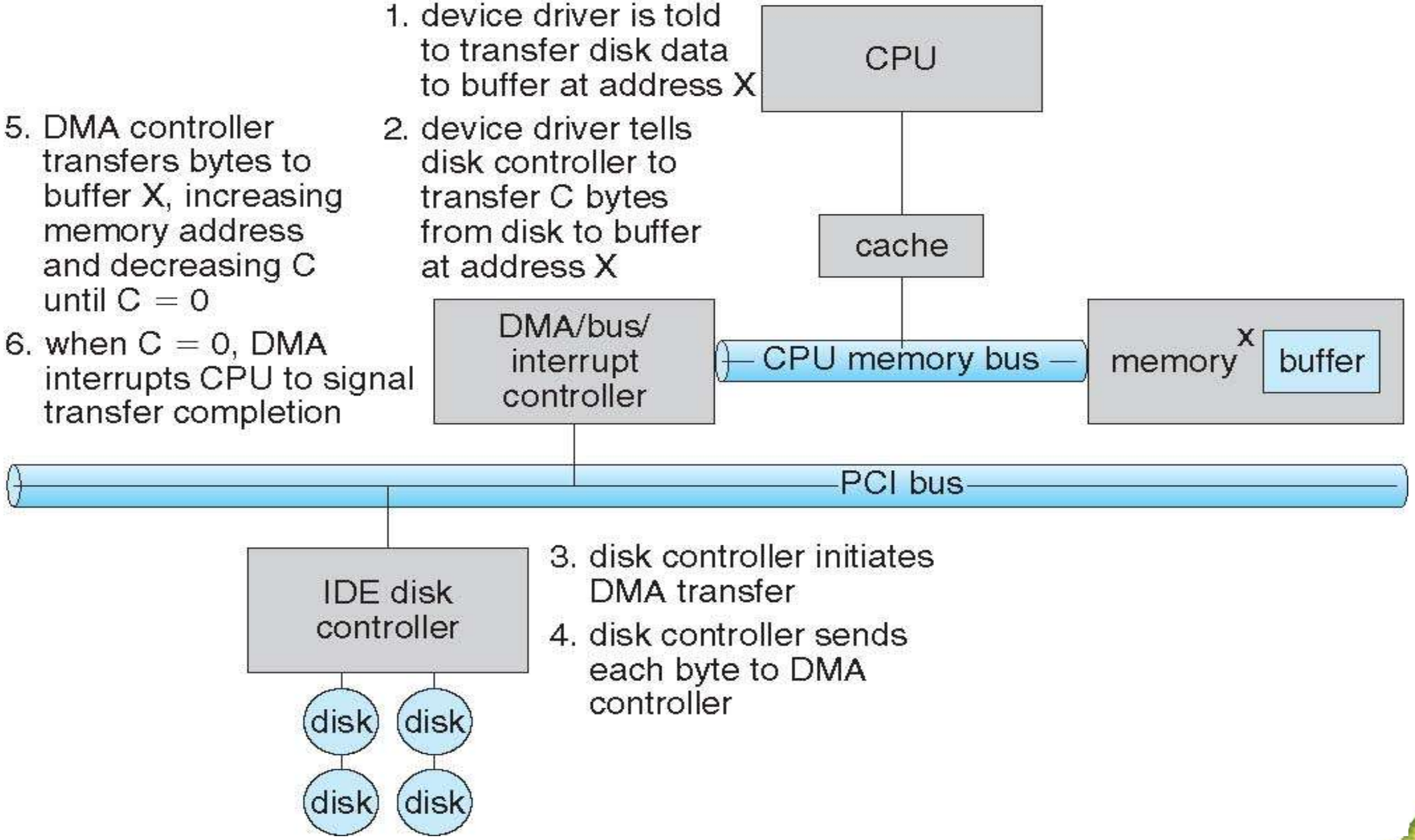
- Interrupt mechanism also used for exceptions
  - Terminate process, crash system due to hardware error
- Page fault executes when memory access error
- System call executes via trap to trigger kernel to execute request
- Multi-CPU systems can process interrupts concurrently
  - If operating system designed to handle it
- Used for time-sensitive processing, frequent, must be fast

Direct Memory Access

- Used to avoid **programmed I/O** (one byte at a time) for large data movement
- Requires **DMA** controller
- Bypasses CPU to transfer data directly between I/O device and memory
- OS writes DMA command block into memory
  - Source and destination addresses
  - Read or write mode
  - Count of bytes
  - Writes location of command block to DMA controller
  - Bus mastering of DMA controller – grabs bus from CPU
  - When done, interrupts to signal completion

Six Step Process to Perform DMA Transfer

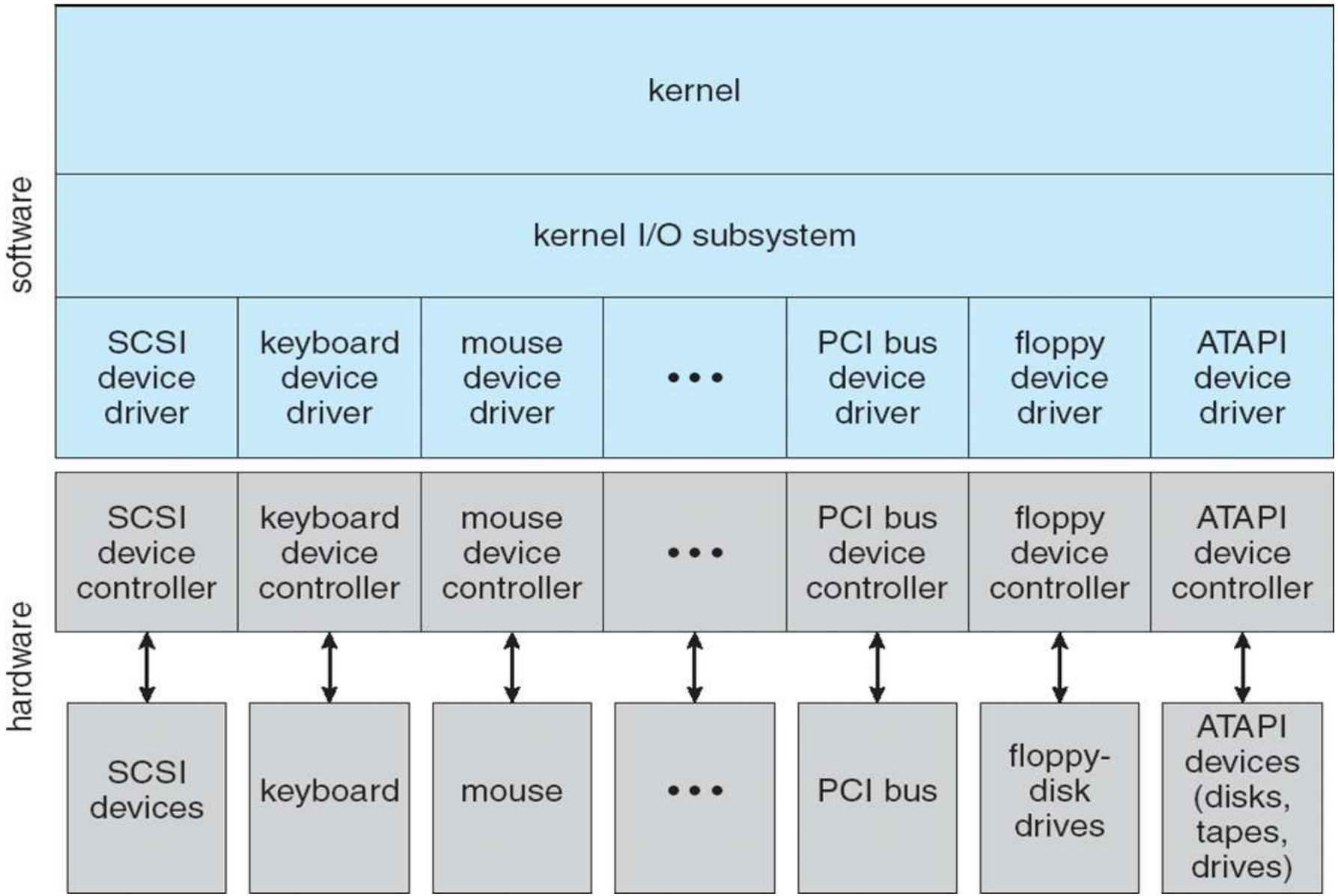
1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion



Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- New devices talking already-implemented protocols need no extra work
- Each OS has its own I/O subsystem structures and device driver frameworks
- Devices vary in many dimensions
  - **Character-stream** or **block**
  - **Sequential** or **random-access**
  - **Synchronous** or **asynchronous** (or both)
  - **Sharable** or **dedicated**
  - **Speed of operation**
  - **read-write**, **read only**, or **write only**

A Kernel I/O Structure



Characteristics of I/O Devices

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

Characteristics of I/O Devices (Cont.)

- Subtleties of devices handled by device drivers
- Broadly I/O devices can be grouped by the OS into
  - Block I/O
  - Character I/O (Stream)
  - Memory-mapped file access
  - Network sockets
- For direct manipulation of I/O device specific characteristics, usually an escape / back door
  - Unix `ioctl()` call to send arbitrary bits to a device control register and data to device data register

Block and Character Devices

- Block devices include disk drives
  - Commands include read, write, seek
  - **Raw I/O**, **direct I/O**, or file-system access
  - Memory-mapped file access possible
    - File mapped to virtual memory and clusters brought via demand paging
  - DMA
- Character devices include keyboards, mice, serial ports
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing

Network Devices

- Varying enough from block and character to have own interface
- 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)

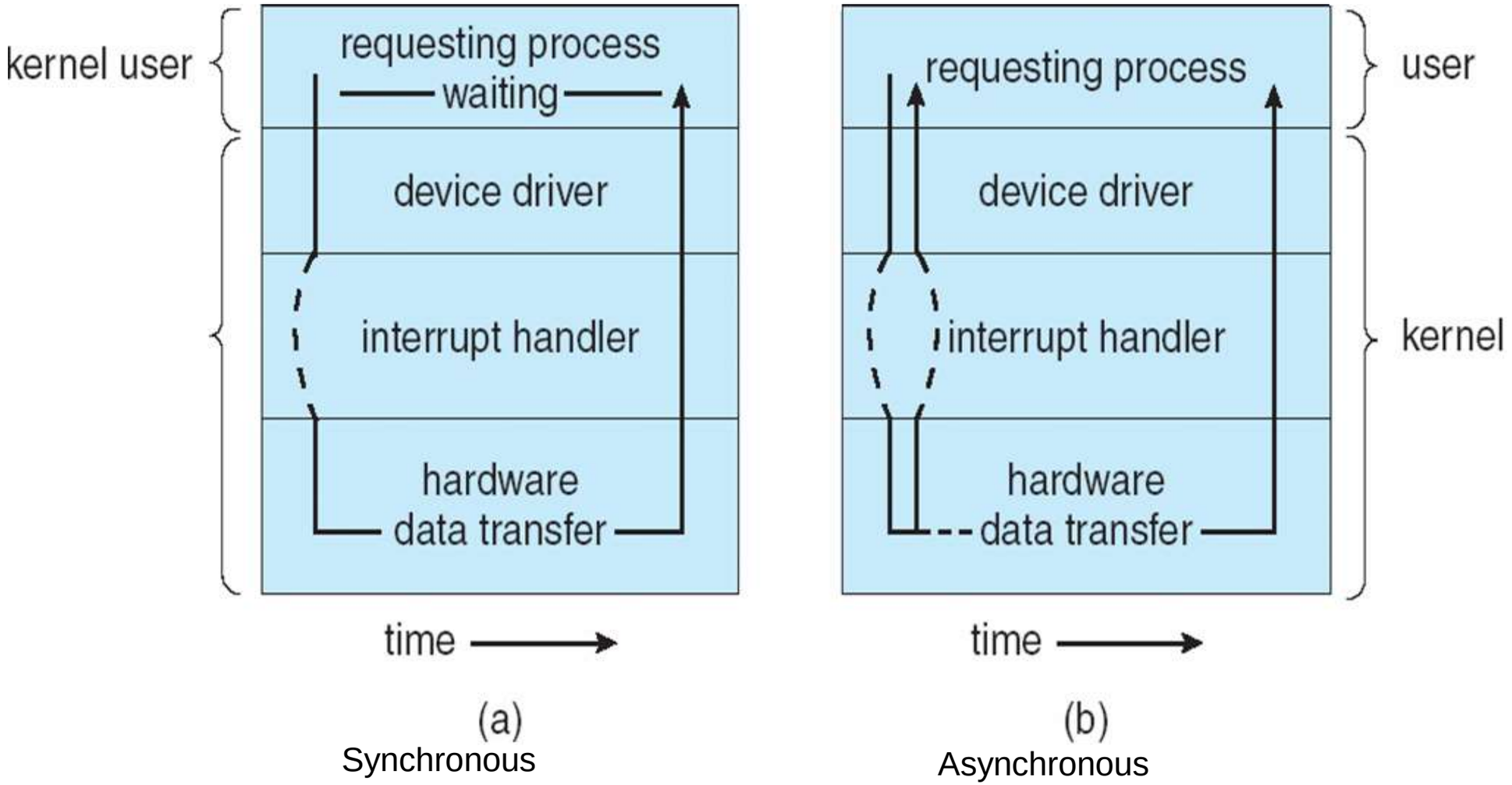
Clocks and Timers

- Provide current time, elapsed time, timer
- Normal resolution about 1/60 second
- Some systems provide higher-resolution timers
- **Programmable interval timer** used for timings, periodic interrupts
- `ioctl()` (on UNIX) covers odd aspects of I/O such as clocks and timers

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
  - `select()` to find if data ready then `read()` or `write()` to transfer
- **Asynchronous** - process runs while I/O executes
  - Difficult to use
  - I/O subsystem signals process when I/O completed

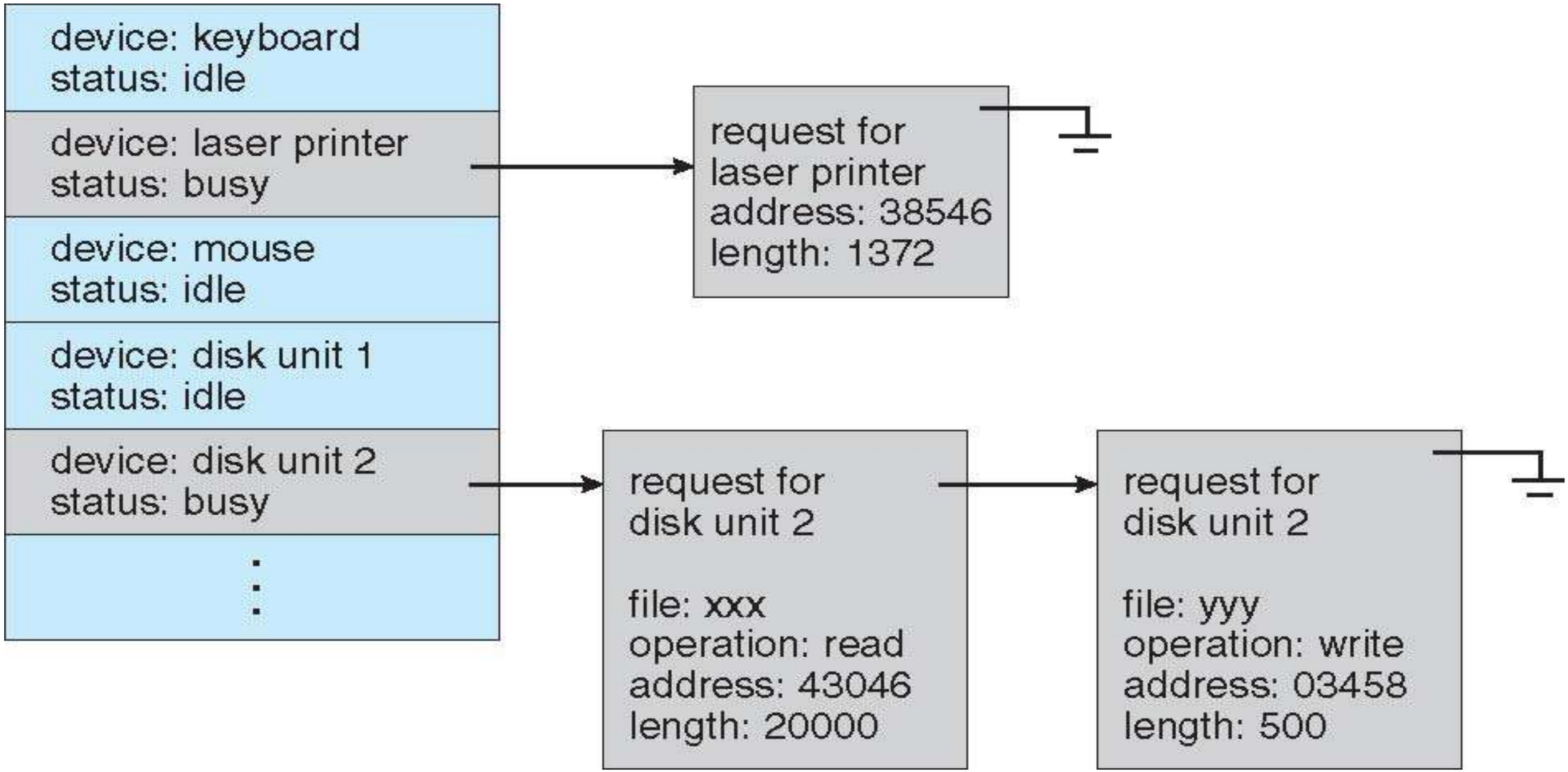
Two I/O Methods



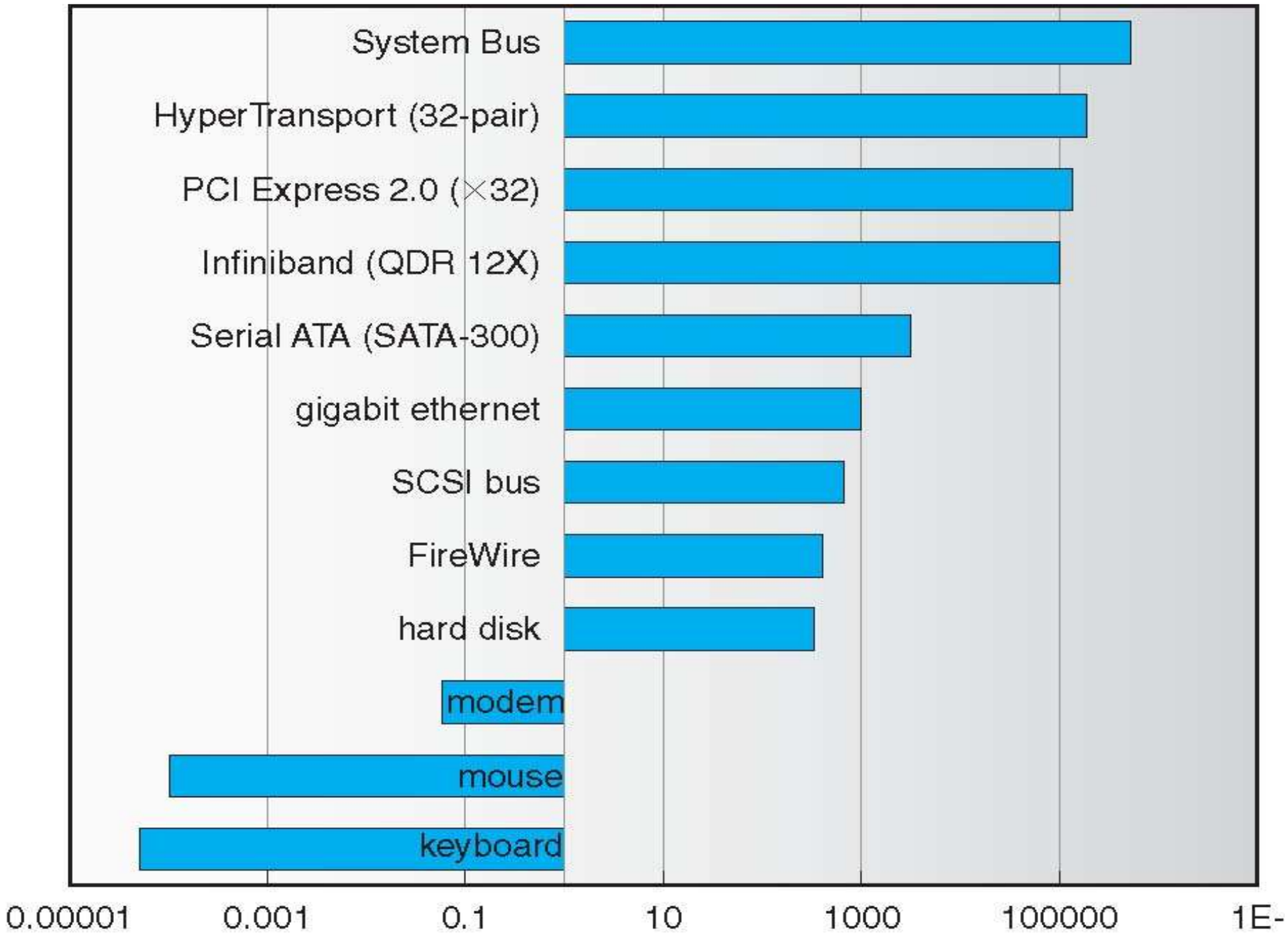
Kernel I/O Subsystem

- Scheduling
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness
  - Some implement Quality Of Service (i.e. IPQOS)
- Buffering - store data in memory while transferring between devices
  - To cope with device speed mismatch
  - To cope with device transfer size mismatch
  - To maintain “copy semantics”
  - Double buffering – two copies of the data
    - Kernel and user
    - Varying sizes
    - Full / being processed and not-full / being used
    - Copy-on-write can be used for efficiency in some cases

Device-status Table



Sun Enterprise 6000 Device-Transfer Rates



Kernel I/O Subsystem

- **Caching** - faster device holding copy of data
  - Always just a copy
  - Key to performance
  - Sometimes combined with buffering
- **Spooling** - hold output for a device
  - If device can serve only one request at a time
  - i.e., Printing
- **Device reservation** - provides exclusive access to a device
  - System calls for allocation and de-allocation
  - Watch out for deadlock

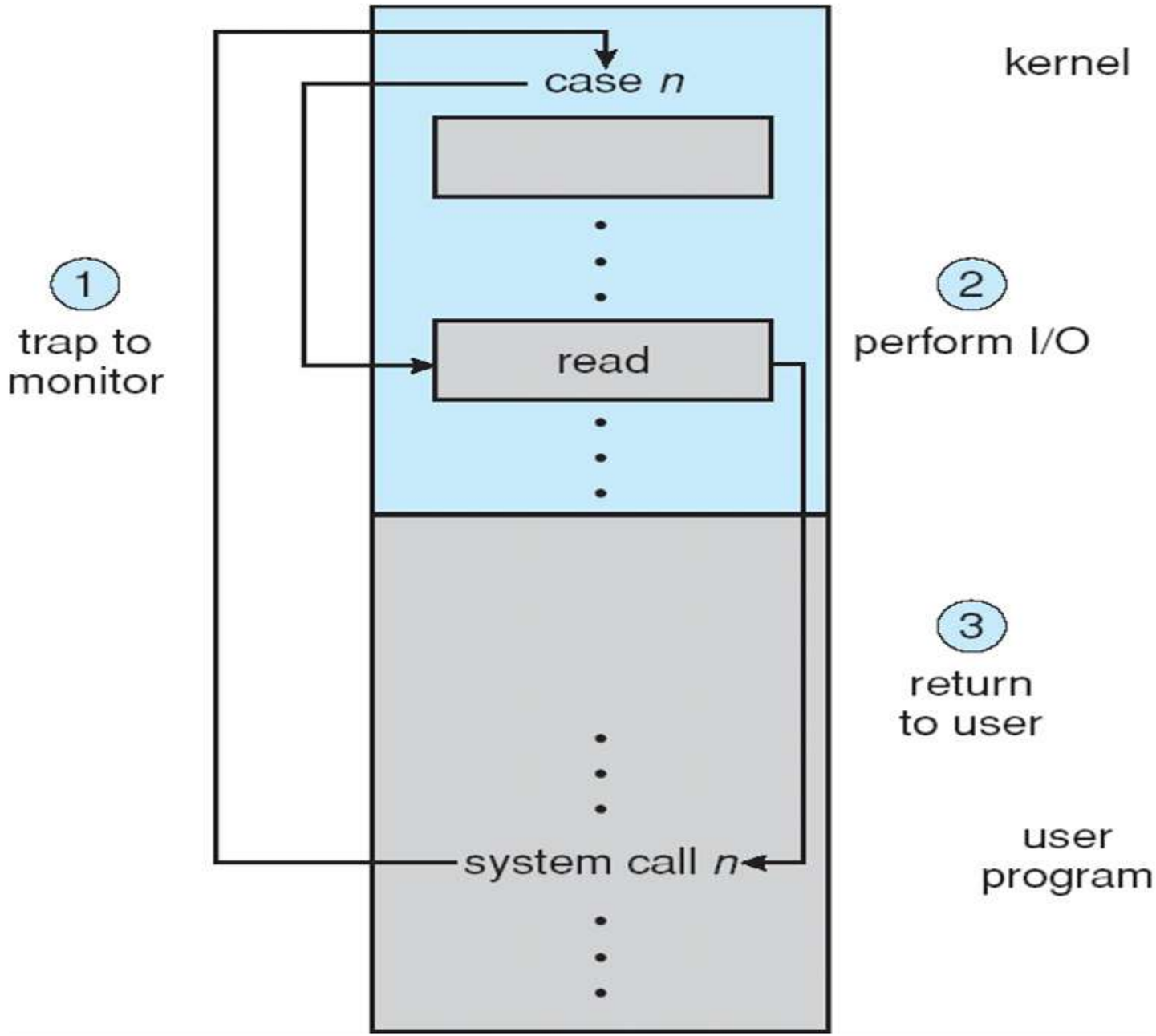
Error Handling

- OS can recover from disk read, device unavailable, transient write failures
  - Retry a read or write, for example
  - Some systems more advanced – Solaris FMA, AIX
    - Track error frequencies, stop using device with increasing frequency of retry-able errors
- Most return an error number or code when I/O request fails
- System error logs hold problem reports

I/O Protection

- User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions
  - All I/O instructions defined to be privileged
  - I/O must be performed via system calls
    - Memory-mapped and I/O port memory locations must be protected too

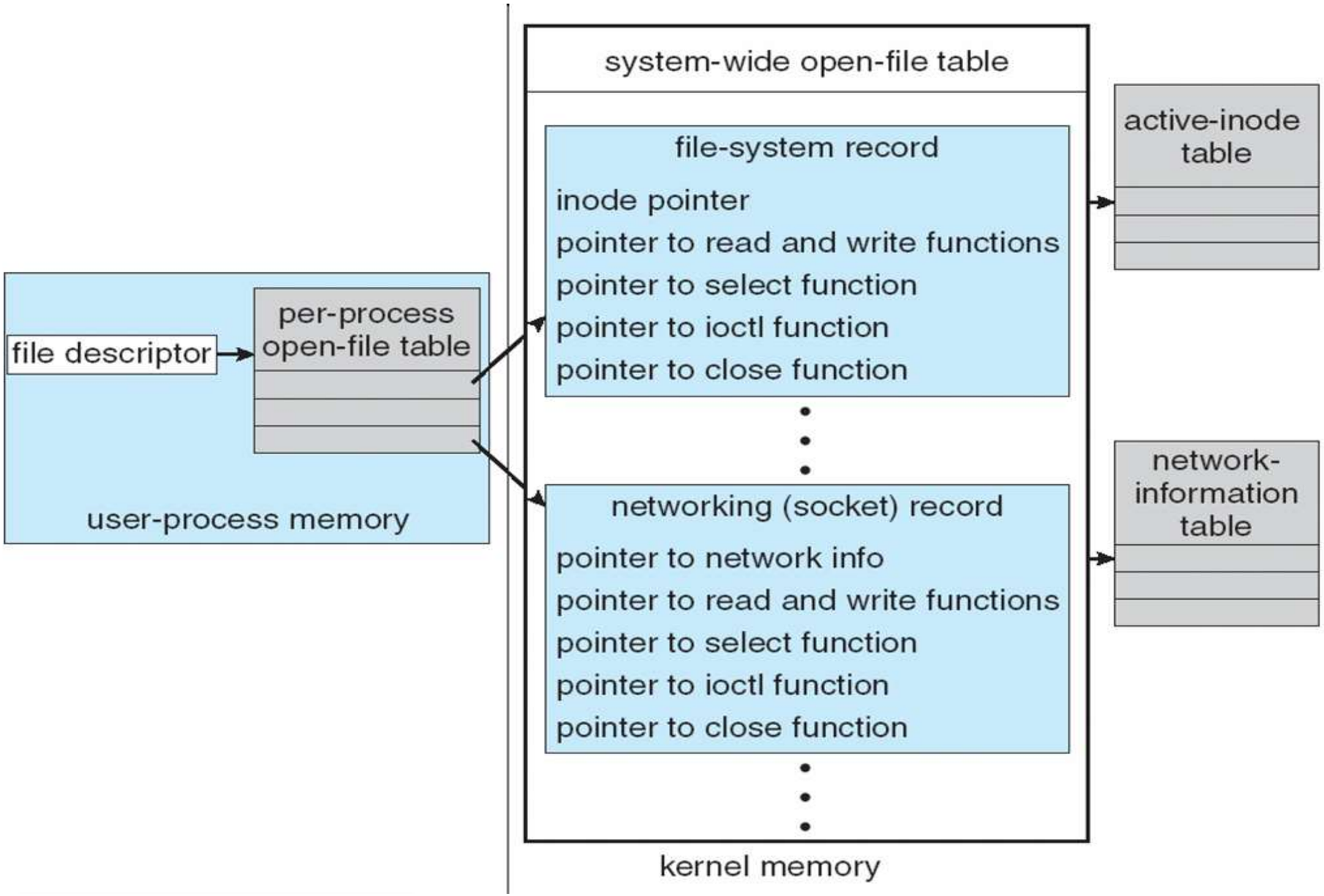
Use of a System Call to Perform I/O



Kernel Data Structures

- Kernel keeps state info for I/O components, including open file tables, network connections, character device state
- Many, many complex data structures to track buffers, memory allocation, “dirty” blocks
- Some use object-oriented methods and message passing to implement I/O
  - Windows uses message passing
    - Message with I/O information passed from user mode into kernel
    - Message modified as it flows through to device driver and back to process
    - Pros / cons?

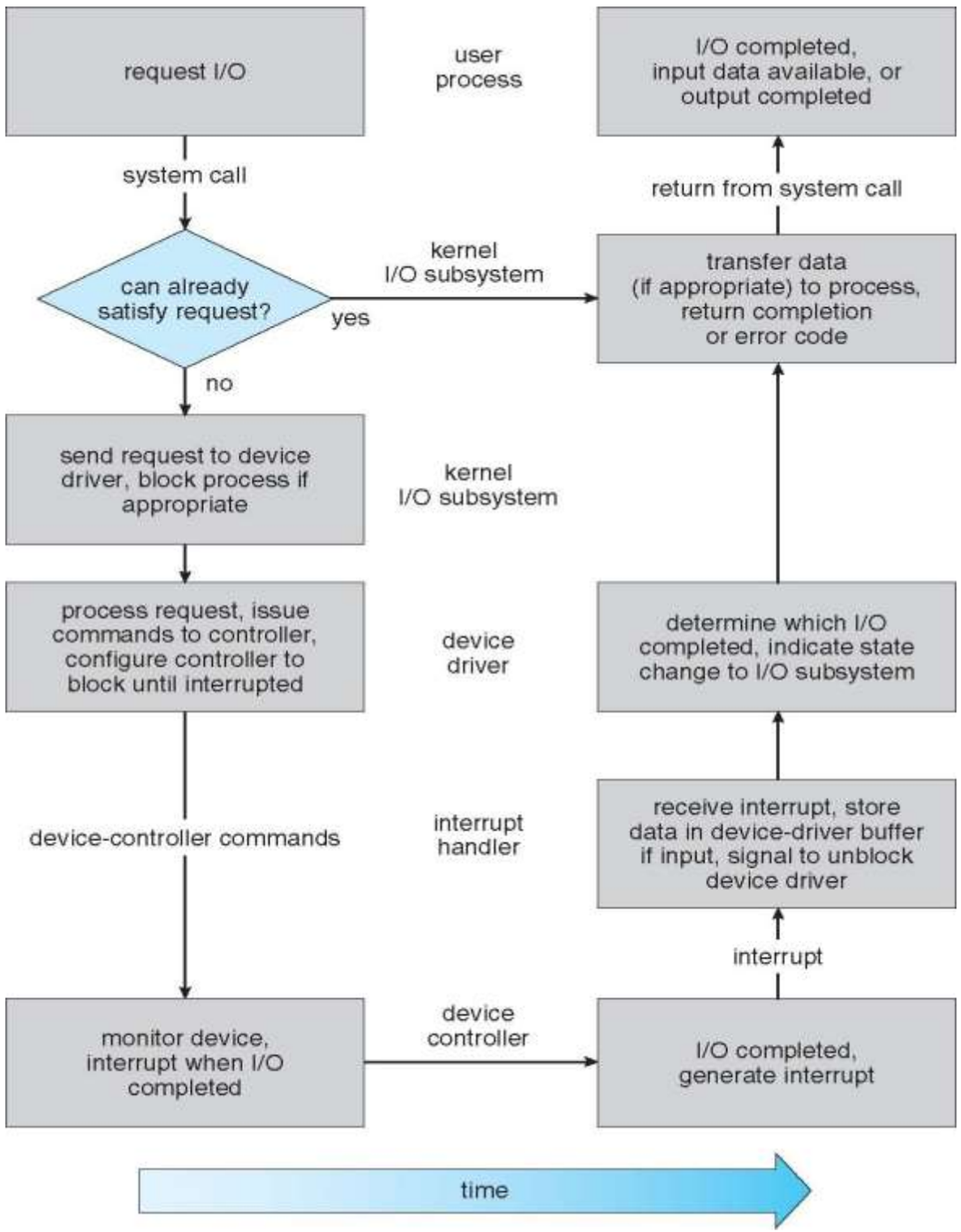
UNIX I/O Kernel Structure



I/O Requests to Hardware Operations

- Consider reading a file from disk for a process:
  - Determine device holding file
  - Translate name to device representation
  - Physically read data from disk into buffer
  - Make data available to requesting process
  - Return control to process

Life Cycle of An I/O Request



STREAMS

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

- A STREAM consists of:

  - STREAM head interfaces with the user process

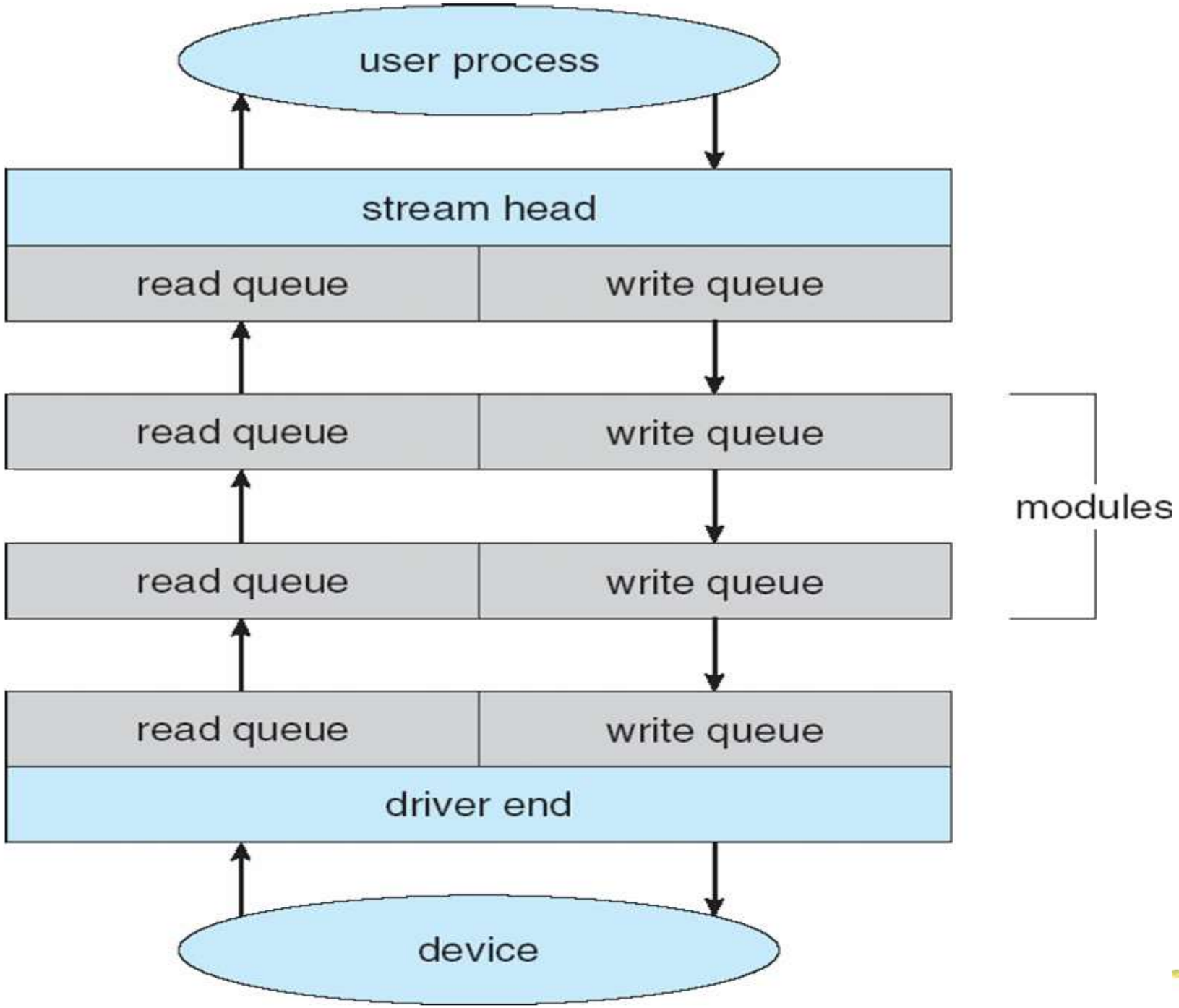
  - driver end interfaces with the device
  - zero or more STREAM modules between them

- Each module contains a **read queue** and a **write queue**

- Message passing is used to communicate between queues
  - **Flow control** option to indicate available or busy

- Asynchronous internally, synchronous where user process communicates with stream head

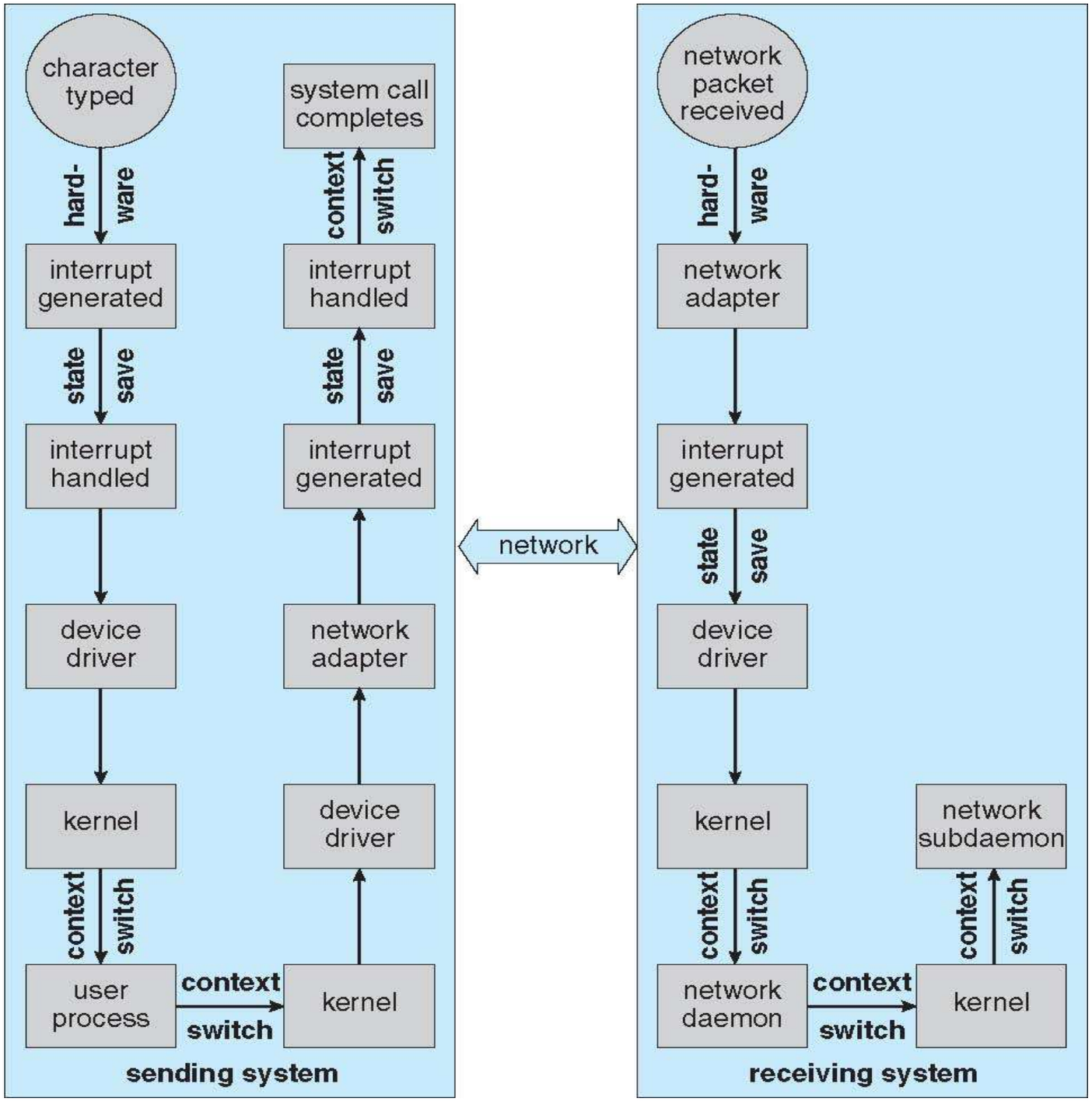
The STREAMS Structure



Performance

- I/O a major factor in system performance:
  - Demands CPU to execute device driver, kernel I/O code
  - Context switches due to interrupts
  - Data copying
  - Network traffic especially stressful

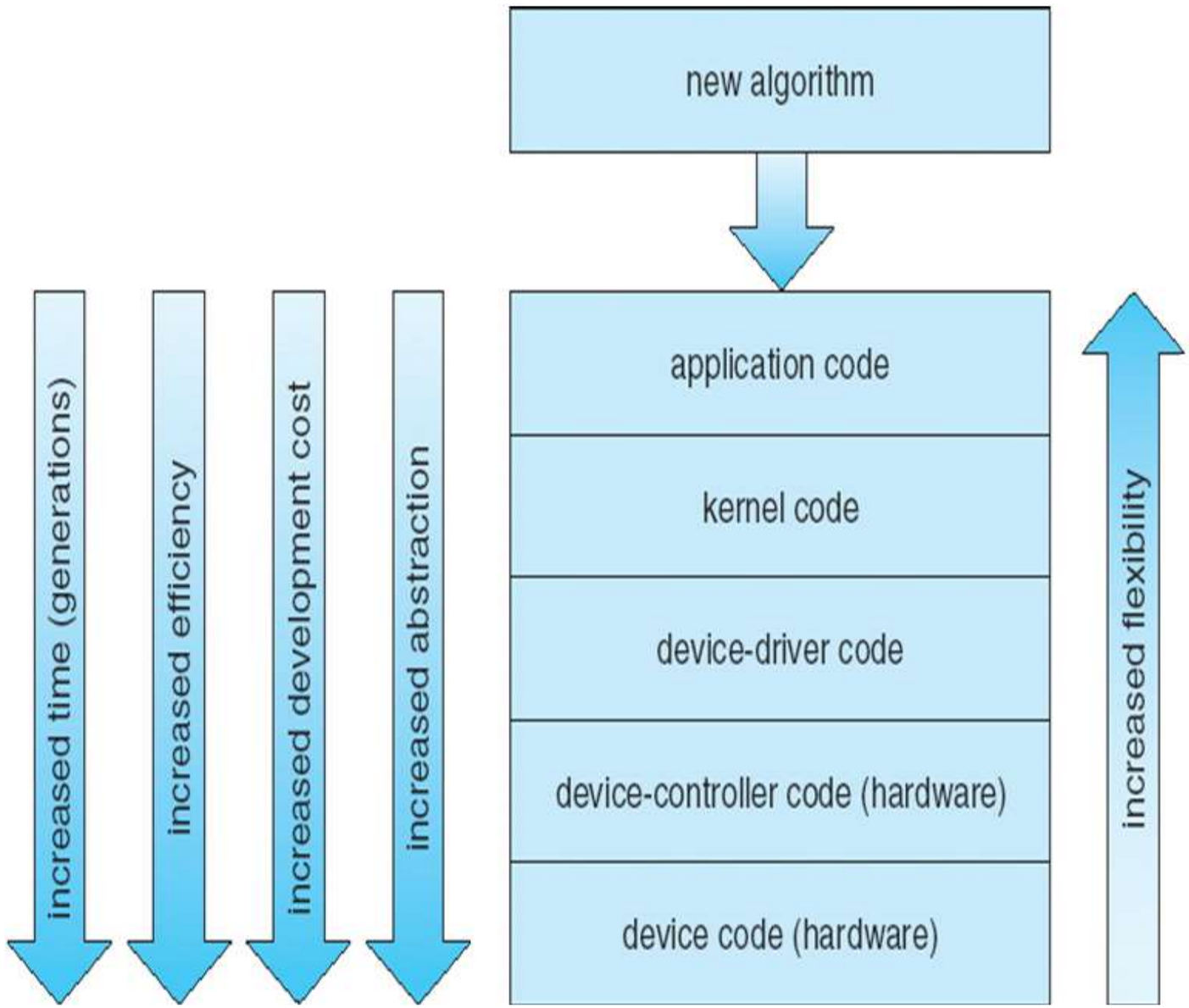
Intercomputer Communications



Improving Performance

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

Device-Functionality Progression



End of Chapter 12