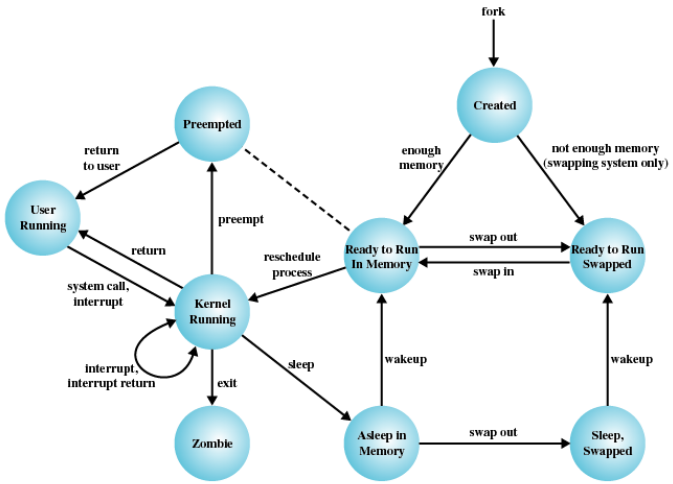
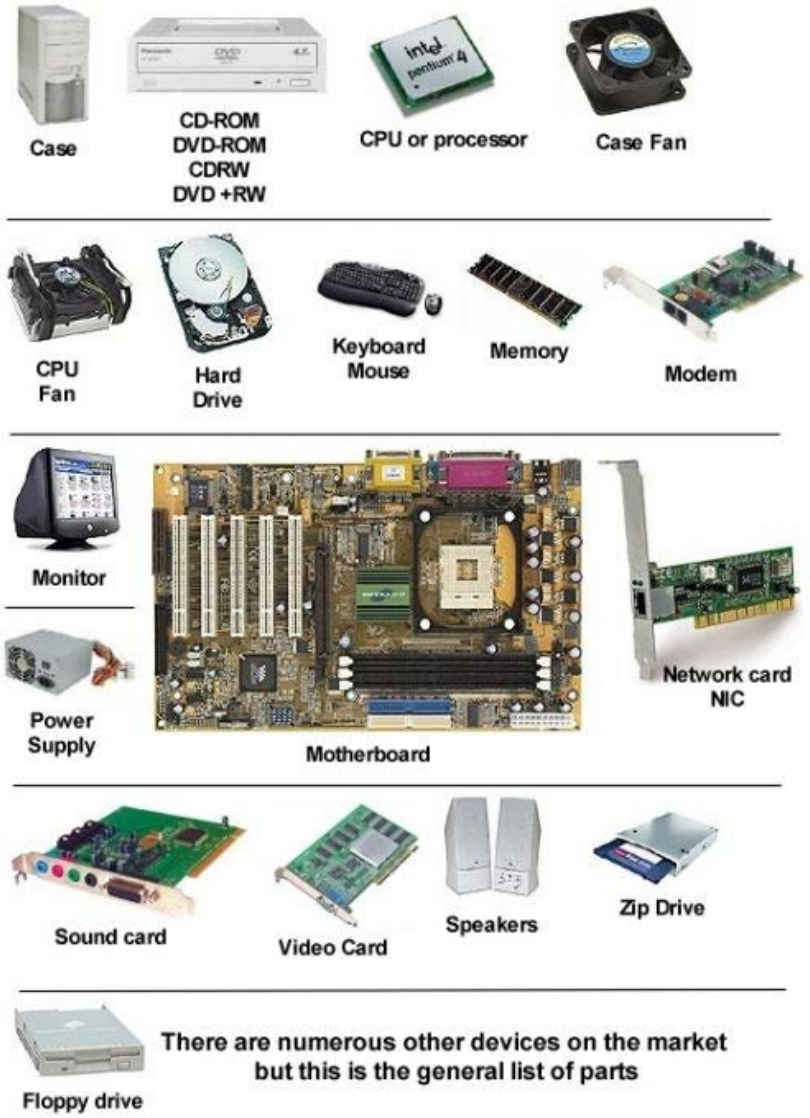
Operating Systems & Computer Networks

I/O and File System



Content

Operating System Design and I/O

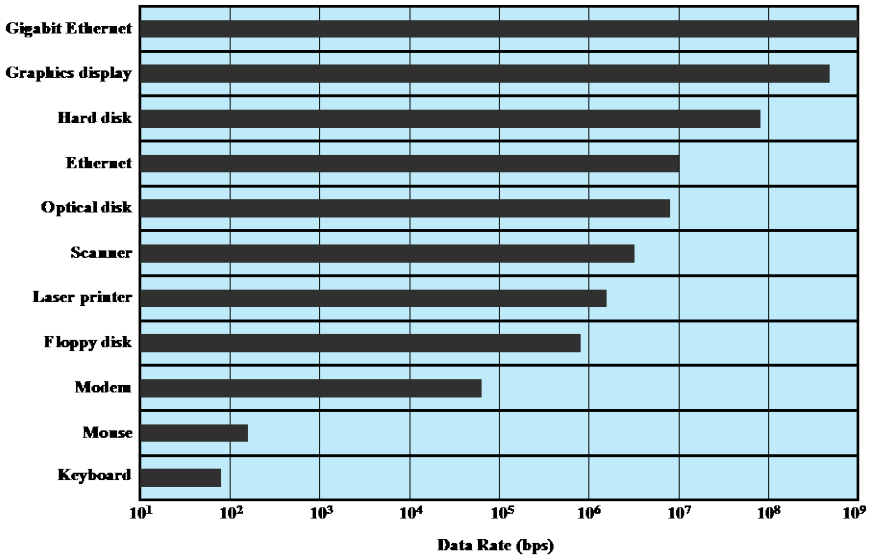


Operating System Design and I/O

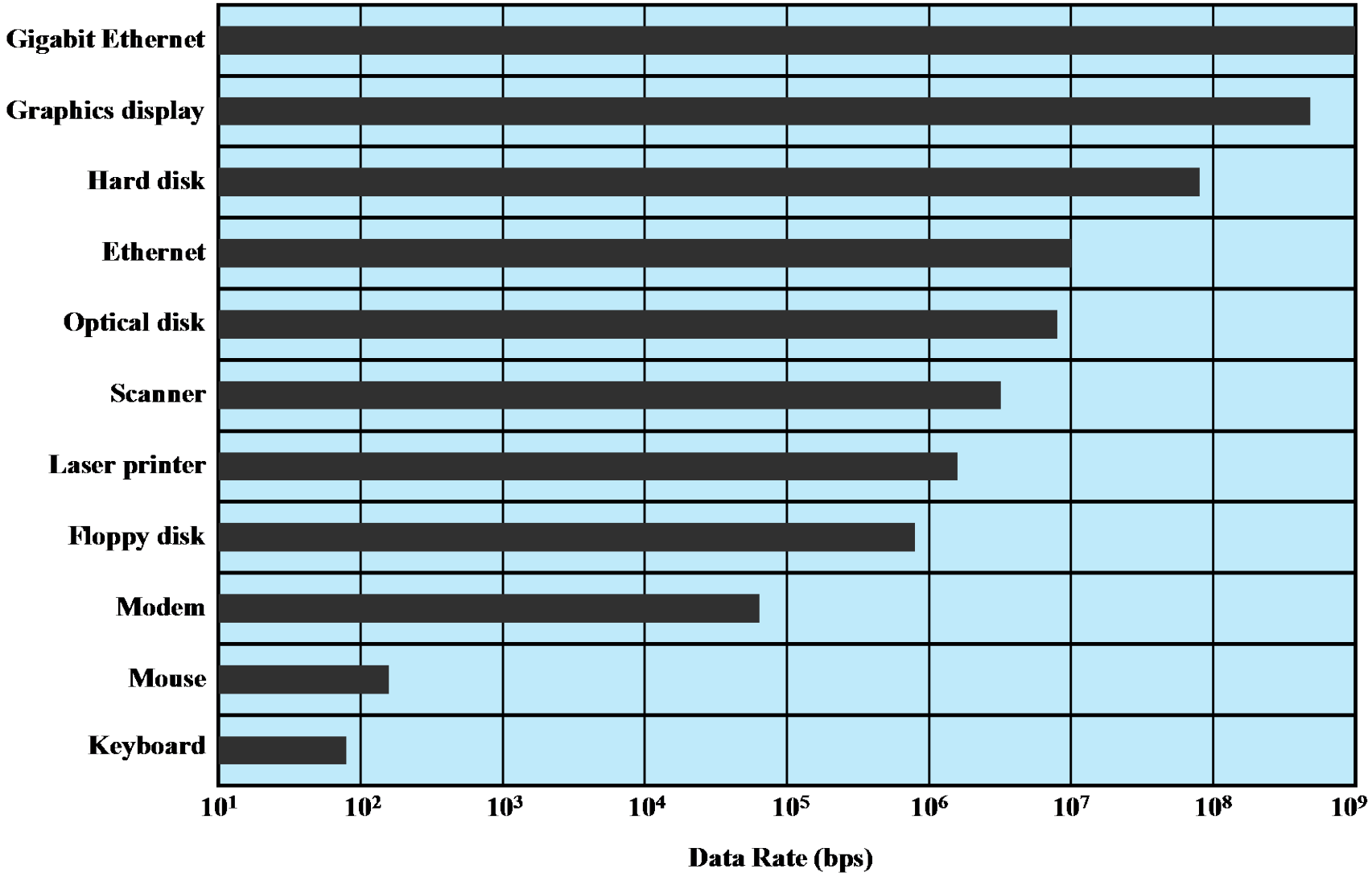
- Efficiency Problems
  - I/O (usually) cannot keep up with processor speed
  - Use of multiprogramming allows for some processes to be waiting on I/O while another process executes
  - Most I/O devices extremely slow compared to main memory
  - Swapping is used to bring in additional Ready processes (requires I/O operations)
- Generality
  - Desirable to handle all I/O devices in a uniform manner, i.e., provide good abstraction to application programmer
  - Hide most of details of device I/O in lower-level routines
    - Processes and upper levels see devices in general terms, e.g., read, write, open, close, lock, unlock
- Conflicting goals motivate focus on API design

Types of I/O Devices

- Wide variety of I/O devices
  - Human readable, e.g., display, keyboard, mouse
  - Machine readable, e.g., disk and tape drives, sensors, controllers, actuators
  - Communication, e.g., digital line drivers, modems
- Data rate
  - Application (software support, priority)
  - Complexity of control
  - Unit of transfer (stream, blocks, characters)
  - Data representation
  - Encoding schemes
  - Error conditions

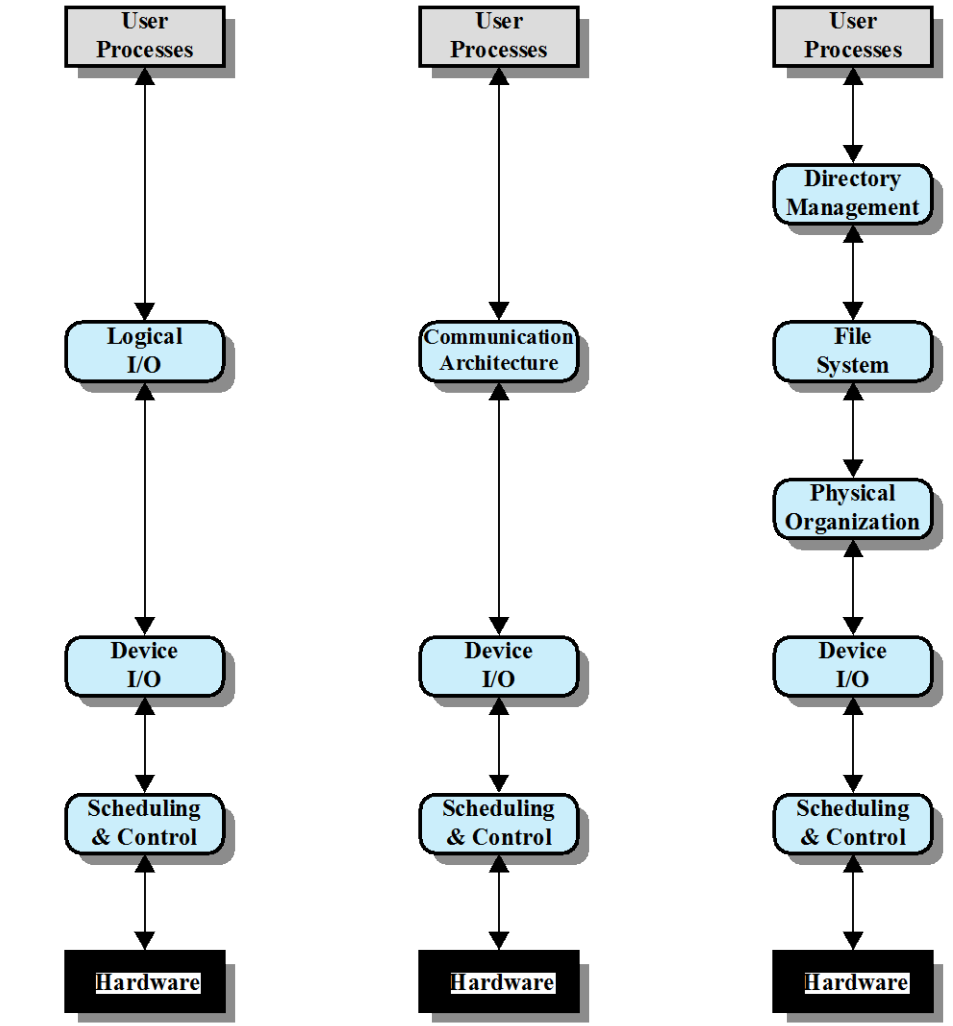


Types of I/O Devices



Alternatives for I/O Organization

- Device abstraction
  - Character-based I/O
    - E.g. input devices like keyboard or mouse
  - Block-based I/O
    - E.g. data storage
  - ➢ Not necessarily related to implementation, e.g. USB
- Communication endpoint (socket) abstraction
  - Used for networking
  - ➢ Second part of this lecture
- File abstraction
  - Structured, persistent storage
  - Sometimes with additional semantics, e.g., locking, transaction support, etc.



(a) Local peripheral device (b) Communications port (c) File system

Evolution of the I/O Function

1. Processor directly controls a peripheral device
2. A controller or I/O module is added
3. Same configuration as step 2, but now interrupts are employed
4. The I/O module is given direct control of memory via DMA
5. The I/O module is enhanced to become a separate processor, with a specialized instruction set tailored for I/O
6. The I/O module has a local memory of its own and is, in fact, a computer in its own right

I/O Related Programming Techniques

- Programmed I/O
  - Process is busy-waiting for the operation to complete

```
while (*IO_STATUS_ADDR != IO_DONE){}
```

- Interrupt-driven I/O
  - I/O command is issued
  - Processor continues executing instructions
  - I/O module sends an interrupt when done

- Direct Memory Access (DMA)
  - DMA module controls exchange of data between main memory and the I/O device
  - Processor interrupted only after entire block has been transferred

Comparison of I/O Techniques

- Programmed I/O
  - Only when there's no alternative, e.g., timing with very high accuracy
- Interrupt-driven I/O
  - Event-based programming, e.g., user input
- Direct memory access
  - Data transfer, e.g. disk I/O, graphics operations, network packet processing

	No Interrupts	**Use of Interrupts**
I/O-to-memory transfer through processor	Programmed I/O	Interrupt-driven I/O
Direct I/O-to-memory transfer		Direct memory access (DMA)

DMA and Buffering

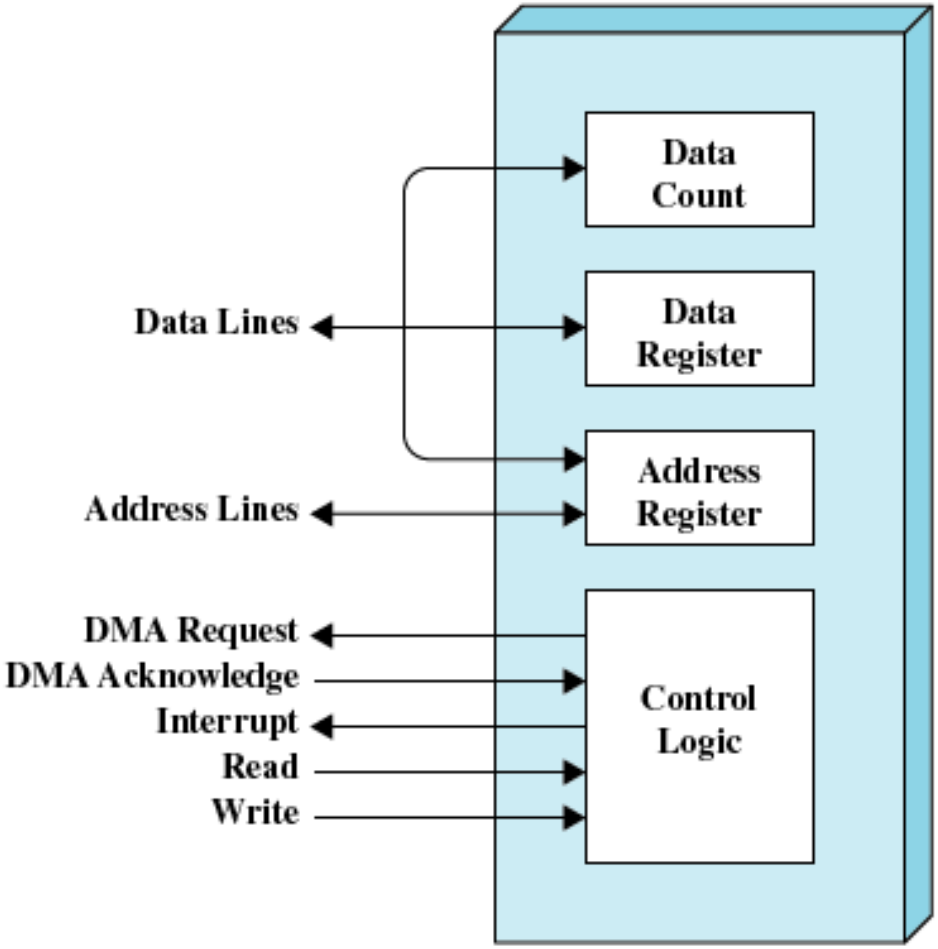
Direct Memory Access

- Moving data between main memory and peripherals is a simple operation, but keeps CPU busy

➢ Delegates I/O operation to extra hardware: DMA module

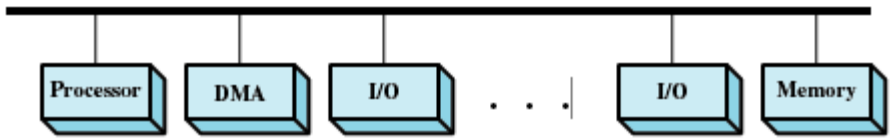
- DMA module transfers data directly to or from memory
  - "For-loop in hardware"
  - ➢ Continuous memory regions

- When complete, DMA module sends interrupt signal to CPU



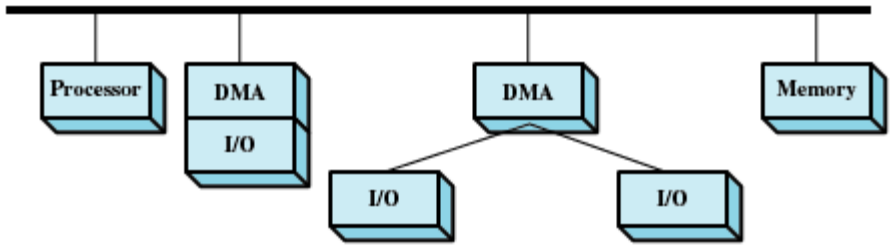
DMA Configurations

a) Single-bus, detached DMA
- Simple, but inefficient
- Requires multiple I/O requests to device



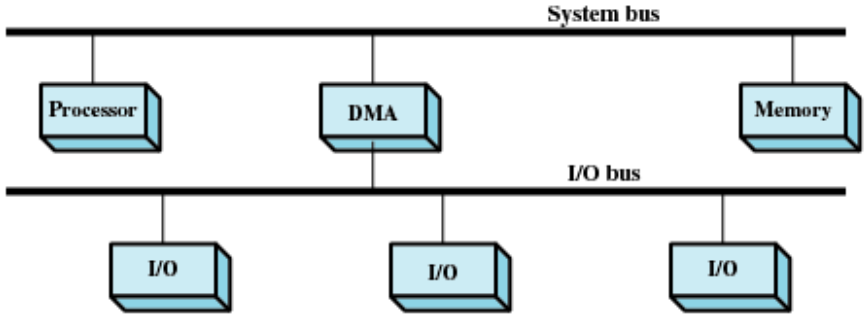
(a) Single-bus, detached DMA

b) Single-bus, integrated DMA-I/O
- Efficient, but expensive
- One controller per device (group)



(b) Single-bus, Integrated DMA-I/O

c) I/O bus
- Efficient and less expensive
- Separate bus, one controller



(c) I/O bus

I/O Buffering

- Main memory used to temporarily store data
  - Mitigates differences in data processing speeds
    - Processes must wait for I/O to complete before proceeding
  - Manage pages that must remain in main memory during I/O
    - Buffer must be accessible to low-level drivers and hardware

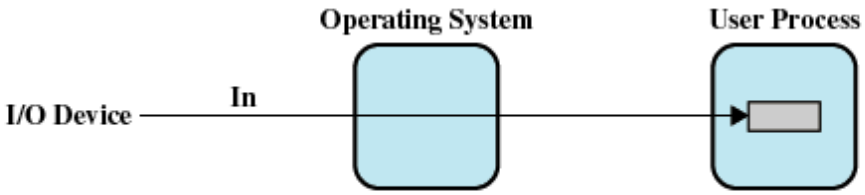
Approaches (with different buffering strategies)

- Block-oriented
  - Information is stored in fixed sized blocks
  - Transfers are made one block at a time
  - ➢ Used for disks and tapes

- Stream-oriented (stream of characters)
  - Transfer information as a stream of bytes
  - ➢ Used for terminals, printers, communication ports, mouse and other pointing devices, and most other devices that are not secondary storage

I/O Buffering Implementations

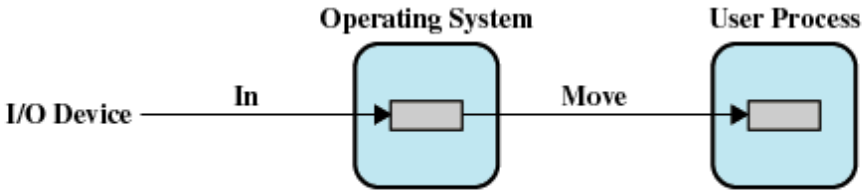
- No buffering



(a) No buffering

I/O Buffering Implementations

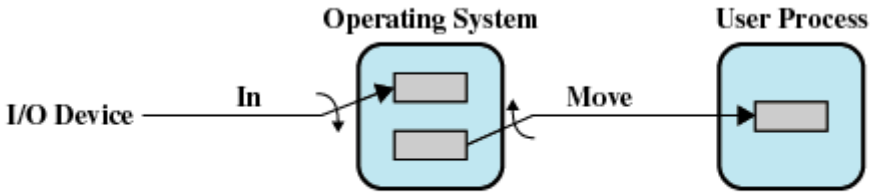
- Single buffering
  - Block-oriented: User process can process one *fixed-sized* block of data while next block is read in
  - Stream-oriented: Process one *variable-sized and delimited* line at time



(b) Single buffering

I/O Buffering Implementations

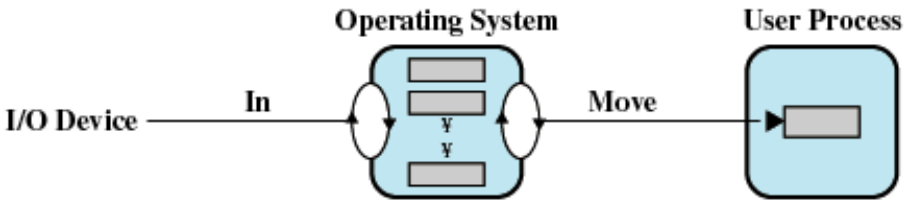
- Double buffering
  - Process can transfer data to or from one buffer while OS empties or fills other buffer



(c) Double buffering

I/O Buffering Implementations

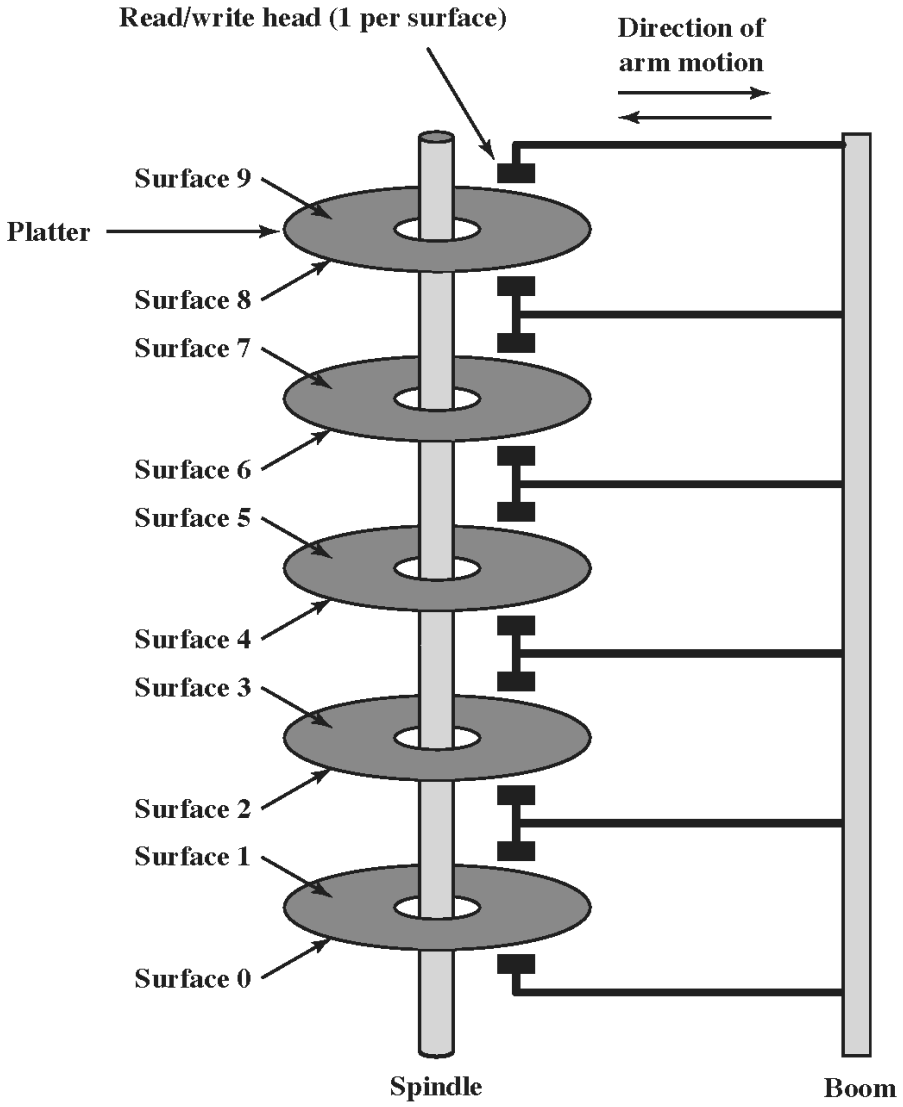
- Circular/ring buffering
  - Each individual buffer is one unit in circular buffer
  - ➢ Used when I/O operation must keep up with process



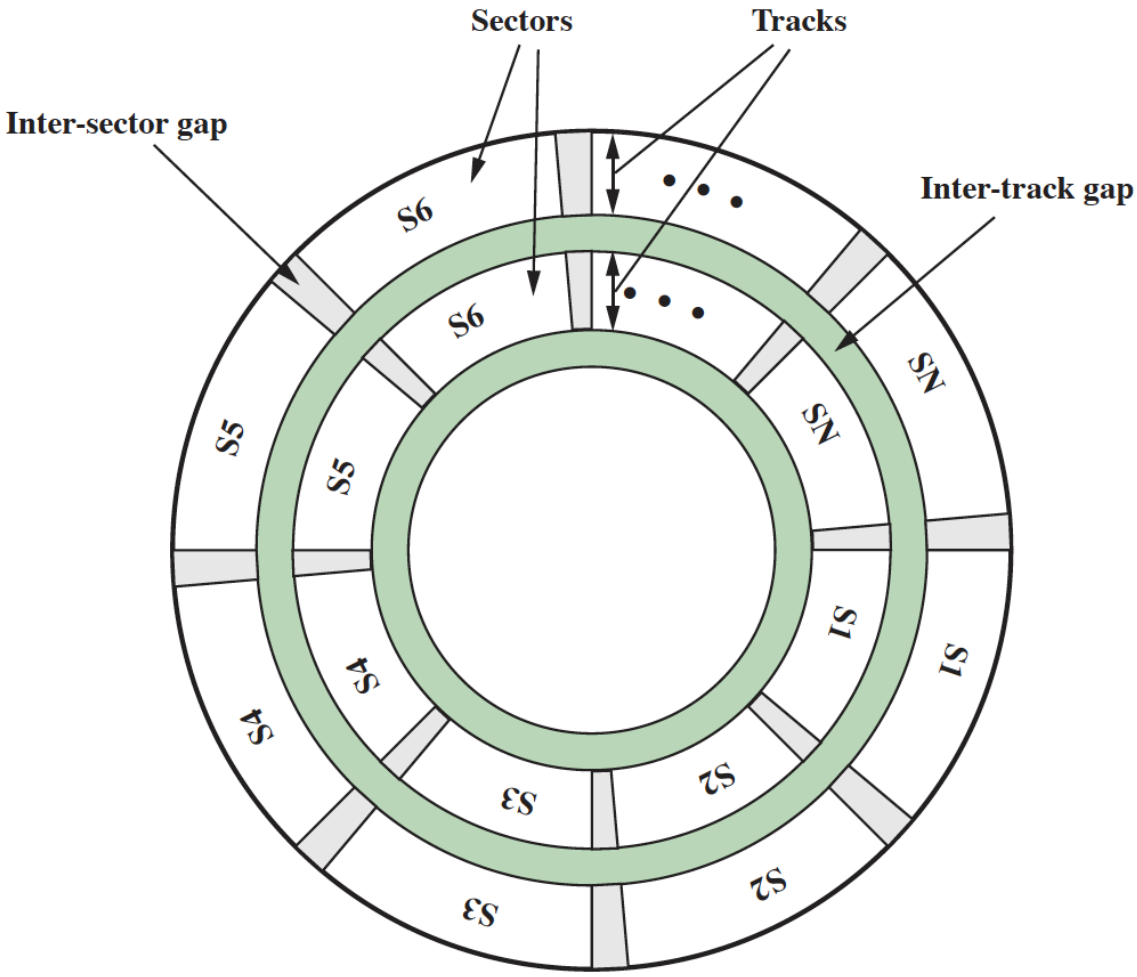
(d) Circular buffering

Disk Drive IO

Disk Drive as Mass Storage



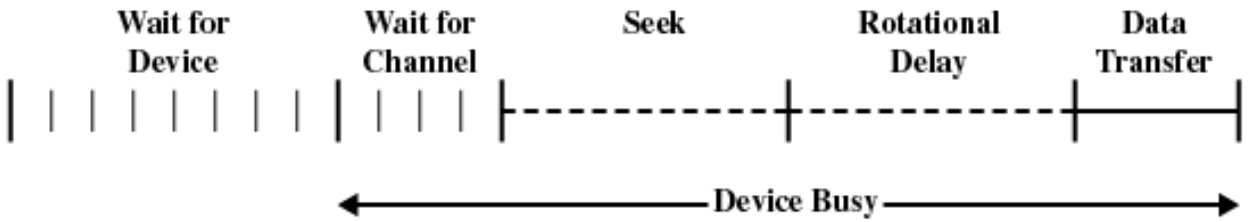
Disk Drive as Mass Storage



I/O Scheduling

- For a single resource there will be a number of I/O requests
  - From *one or several* processes
  - Some devices keep internal state, so ordering of I/O requests matters

- Example: Disk access



  - Access time
    - Sum of seek time and rotational delay
      - Time it takes to get in position to read or write
    - ➢ Seek time is the reason for differences in performance
  - Data transfer occurs as the sector moves under the head
  - ➢ Reorder I/O requests according to current state of disk

Positioning the Read/Write Heads

- When the disk drive is operating, the disk is rotating at constant speed
- To read or write the head must be positioned at the desired track and at the beginning of the desired sector on that track
- Track selection involves moving the head in a movable-head system or electronically selecting one head on a fixed-head system
- On a movable-head system the time it takes to position the head at the track is known as seek time
- The time it takes for the beginning of the sector to reach the head is known as rotational delay
- The sum of the seek time and the rotational delay equals the access time

Disk Scheduling Algorithms

Name	Description	Remarks
Selection according to requestor		
RSS	Random scheduling	For analysis and simulation
FIFO	First in first out	Fairest of them all
PRI	Priority by process	Control outside of disk queue management
LIFO	Last in first out	Maximize locality and resource utilization
Selection according to requested item		
SSTF	Shortest service time first	High utilization, small queues
SCAN	Back and forth over disk	Better service distribution
C-SCAN	One way with fast return	Lower service variability
N-step-SCAN	SCAN of N records at a time	Service guarantee
FSCAN	N-step-SCAN with N = queue size at beginning of SCAN cycle	Load sensitive

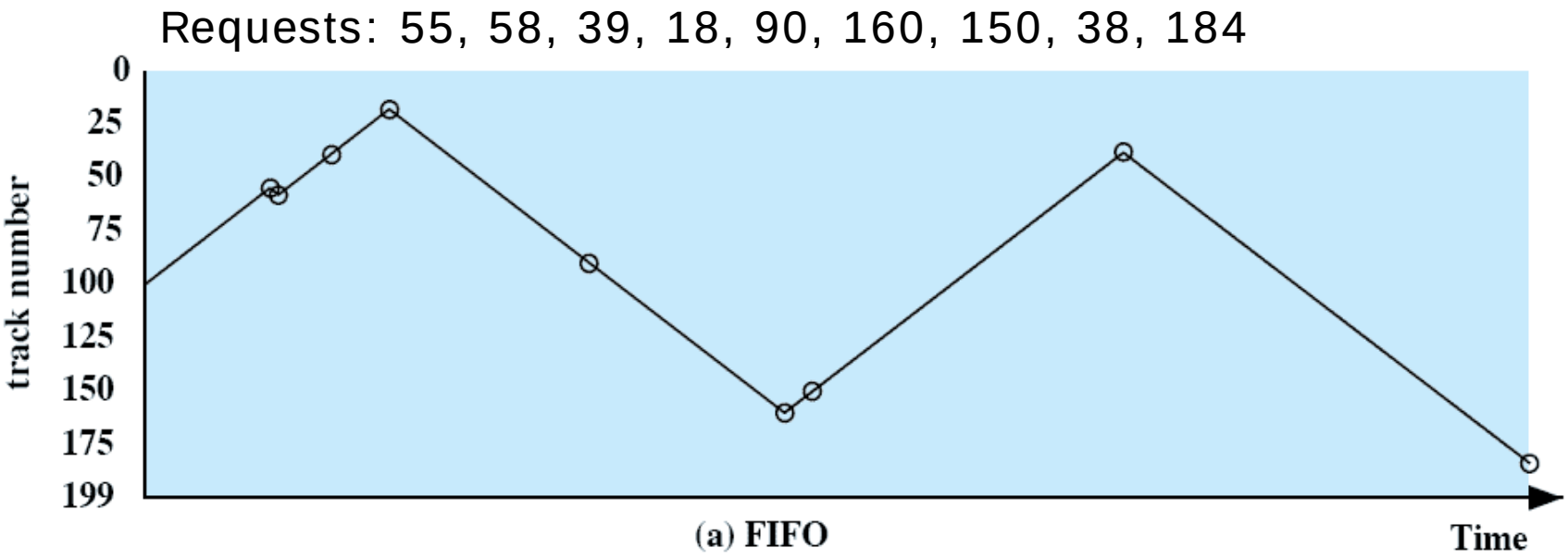
Disk I/O Scheduling Policies

- Example
  - Disk with 200 tracks
  - Disk request queue has random requests
  - Order of requests

55, 58, 39, 18, 90, 160, 150, 38, 184

First-In, First-Out (FIFO)

- Processes in sequential order
- Fair to all processes
- Approximates random scheduling in performance if there are many processes competing for the disk



Priority (PRI)

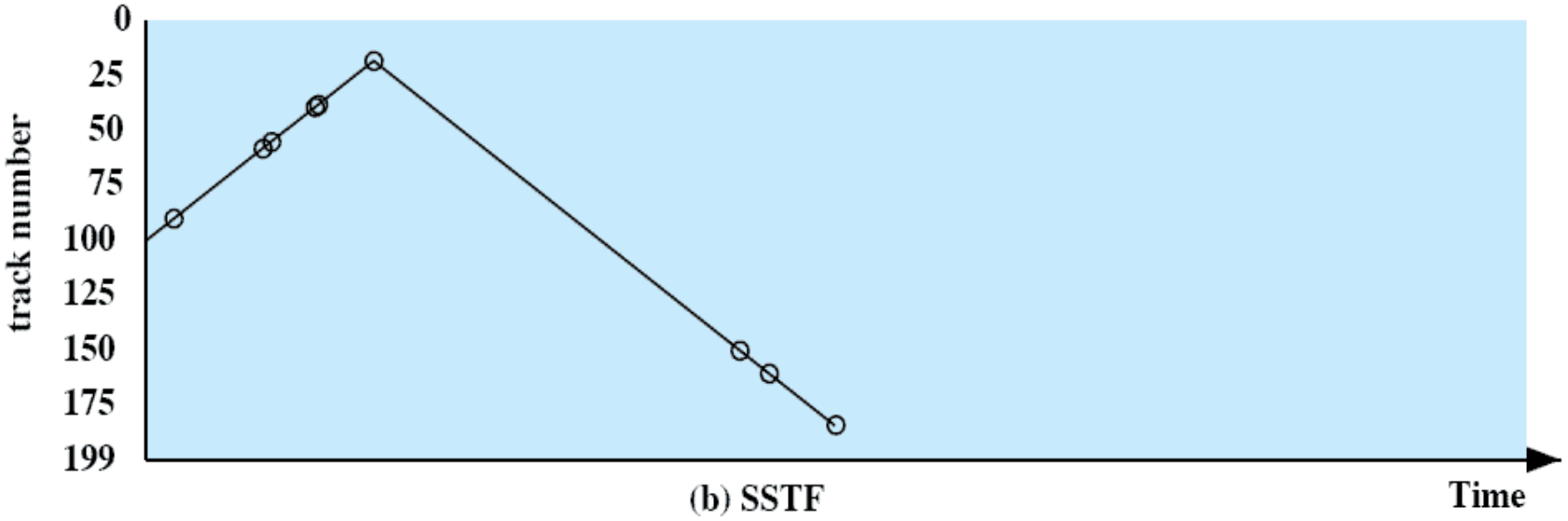
- Control of the scheduling is outside the control of disk management software
- Goal is not to optimize disk utilization but to meet other objectives
- Short batch jobs and interactive jobs are given higher priority
- Provides good interactive response time
- Longer jobs may have to wait an excessively long time
- A poor policy for database systems

Shortest Service Time First (SSTF)

- Select the disk I/O request that requires the least movement of the disk arm from its current position
- Always choose the minimum seek time

Requests: 55, 58, 39, 18, 90, 160, 150, 38, 184

Service order: 90, 58, 55, 39, 38, 18, 150, 160, 184



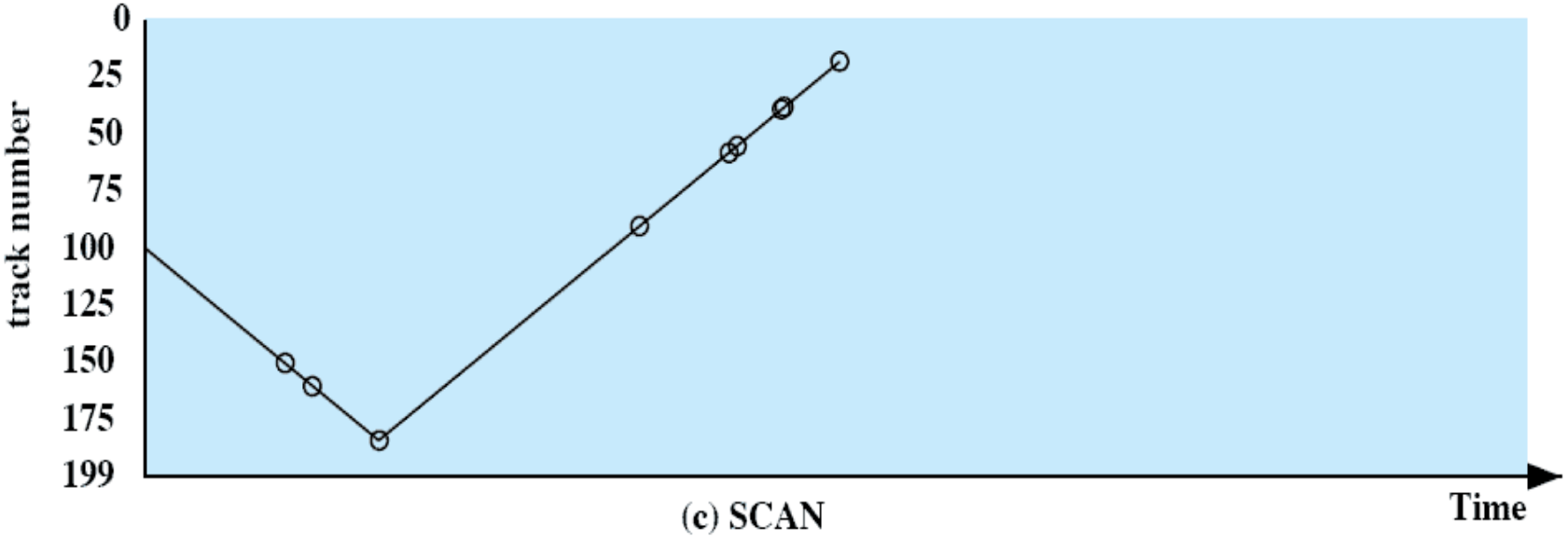
(b) SSTF

SCAN

- Also known as the elevator algorithm
- Arm moves in one direction only
  - satisfies all outstanding requests until it reaches the last track in that direction then the direction is reversed
- Favors jobs whose requests are for tracks nearest to both innermost and outermost tracks

Requests: 55, 58, 39, 18, 90, 160, 150, 38, 184

Service order: 150, 160, 184, 90, 58, 55, 39, 38, 18



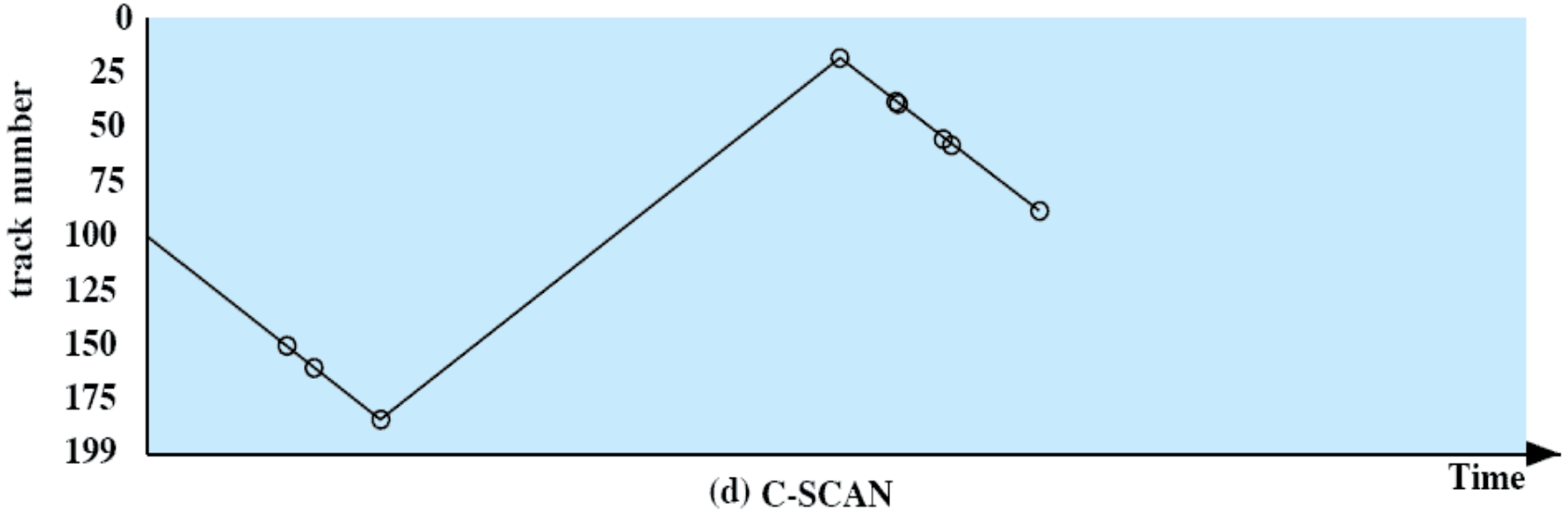
(c) SCAN

C-SCAN (Circular SCAN)

- Restricts scanning to one direction only
- When the last track has been visited in one direction, the arm is returned to the opposite end of the disk and the scan begins again

Requests: 55, 58, 39, 18, 90, 160, 150, 38, 184

Service order: 150, 160, 184, 18, 38, 39, 55, 58



(d) C-SCAN

N-Step-SCAN

- Segments the disk request queue into subqueues of length N
- Subqueues are processed one at a time, using SCAN
- While a queue is being processed new requests must be added to some other queue
- If fewer than N requests are available at the end of a scan, all of them are processed with the next scan

FSCAN

- Uses two subqueues
- When a scan begins, all of the requests are in one of the queues, with the other empty
- During scan, all new requests are put into the other queue
- Service of new requests is deferred until all of the old requests have been processed

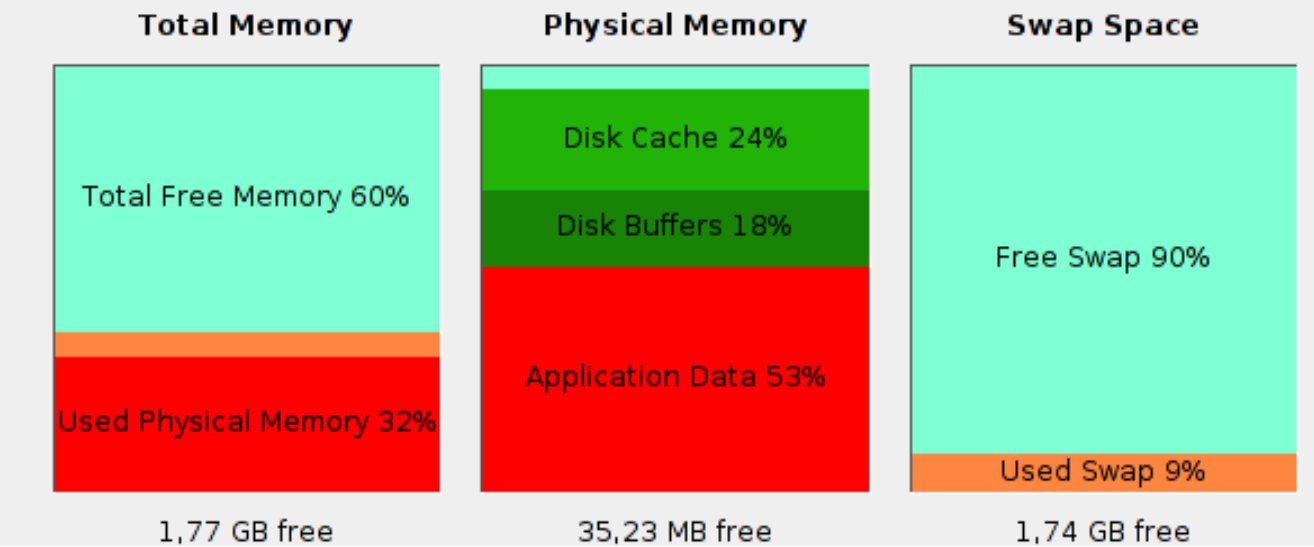
Comparison of Disk Scheduling Algorithms

Freie Universität Berlin

(a) FIFO (starting at track 100)		(b) SSTF (starting at track 100)		(c) SCAN (starting at track 100, in the direction of increasing track number)		(d) C-SCAN (starting at track 100, in the direction of increasing track number)	
Next track accessed	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Average seek length	55.3	**Average seek length**	27.5	**Average seek length**	27.8	**Average seek length**	35.8

Disk Cache

- Main memory buffer for disk sectors
  - Contains copy of subset of sectors on disk
  - ➢ Speeds up I/O requests to these sectors



- Policies:
  - Least Recently Used
    - Block longest in cache with no reference to it is replaced
  - Least Frequently Used
    - Block with fewest references is replaced
    - ➢ Reference count is misleading for bursty access patterns

RAID

RAID

- Redundant Array of Independent Disks
- Set of physical disk drives viewed by the operating system as a single logical drive
- Data are distributed across the physical drives of an array
- Redundant disk capacity is used to store parity information

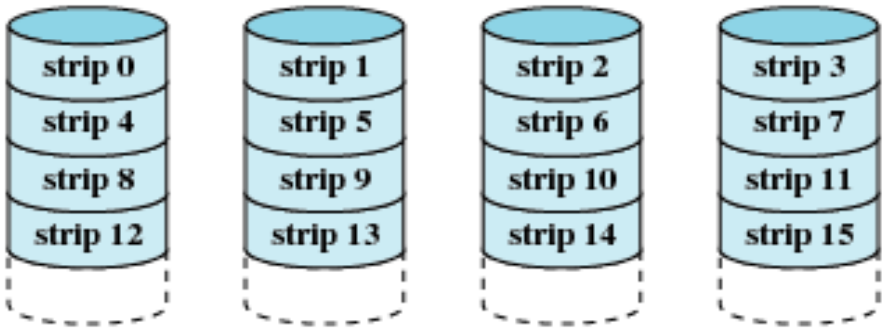
RAID Levels

Category	Level	Description	Disks required	Data availability	Large I/O data transfer capacity	Small I/O request rate
Striping	0	Nonredundant	N	Lower than single disk	Very high	Very high for both read and write
Mirroring	1	Mirrored	$2N$	Higher than RAID 2, 3, 4, or 5; lower than RAID 6	Higher than single disk for read; similar to single disk for write	Up to twice that of a single disk for read; similar to single disk for write
Parallel access	2	Redundant via Hamming code	$N + m$	Much higher than single disk; comparable to RAID 3, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
	3	Bit-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
Independent access	4	Block-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 5	Similar to RAID 0 for read; significantly lower than single disk for write	Similar to RAID 0 for read; significantly lower than single disk for write
	5	Block-interleaved distributed parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 4	Similar to RAID 0 for read; lower than single disk for write	Similar to RAID 0 for read; generally lower than single disk for write
	6	Block-interleaved dual distributed parity	$N + 2$	Highest of all listed alternatives	Similar to RAID 0 for read; lower than RAID 5 for write	Similar to RAID 0 for read; significantly lower than RAID 5 for write

N = number of data disks; m proportional to $\log N$

RAID Level 0

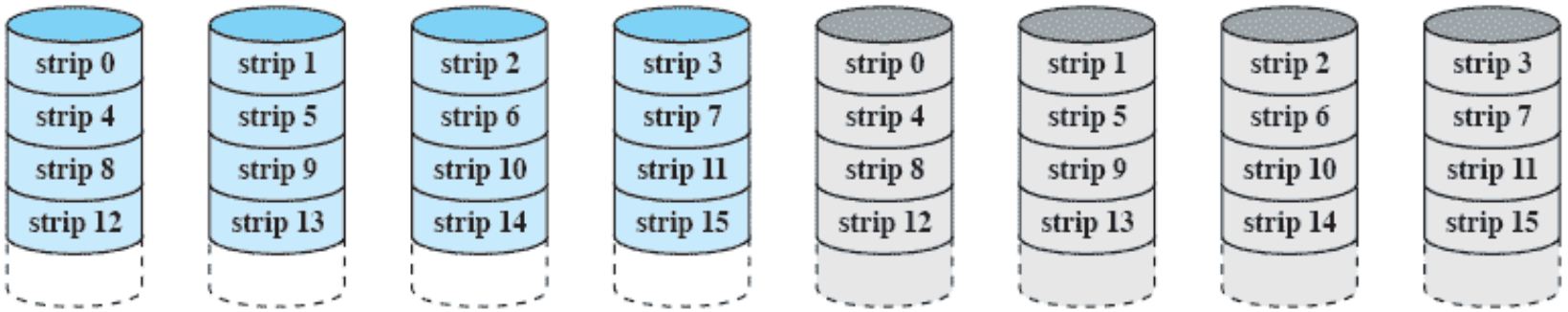
- Not a true RAID because it does not include redundancy to improve performance or provide data protection
- User and system data are distributed across all of the disks in the array
- Logical disk is divided into strips



(a) RAID 0 (non-redundant)

RAID Level 1

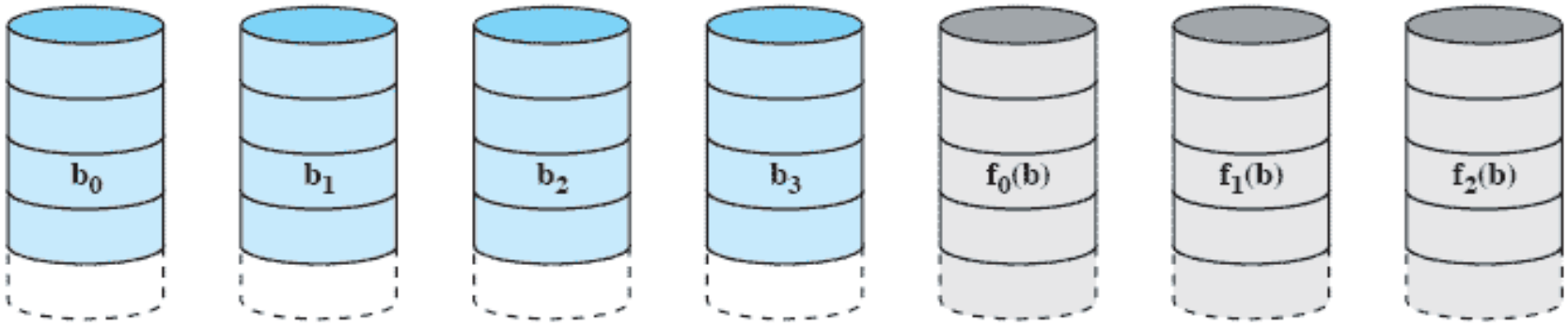
- Redundancy is achieved by the simple expedient of duplicating all the data
- There is no “write penalty”
- When a drive fails the data may still be accessed from the second drive
- Principal disadvantage is the cost



(b) RAID 1 (mirrored)

RAID Level 2

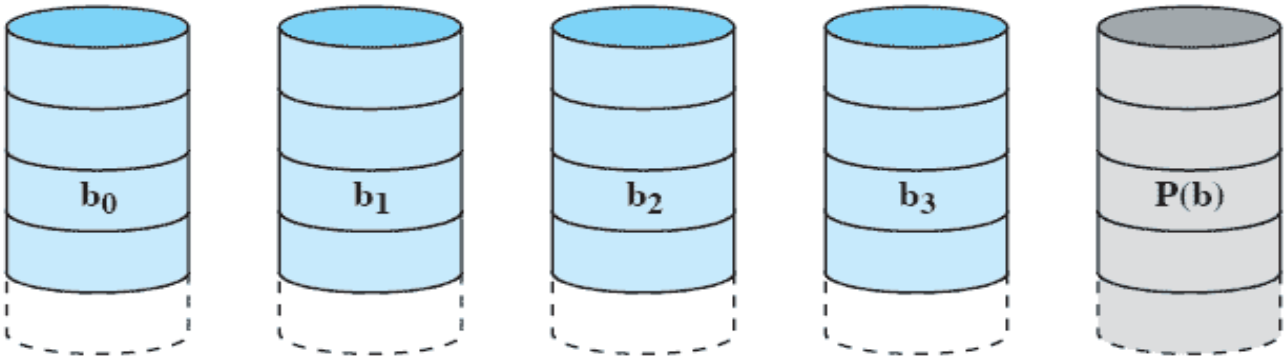
- Makes use of a parallel access technique
- Data striping is used
- Typically a Hamming code is used
- Effective choice in an environment in which many disk errors occur



(c) RAID 2 (redundancy through Hamming code)

RAID Level 3

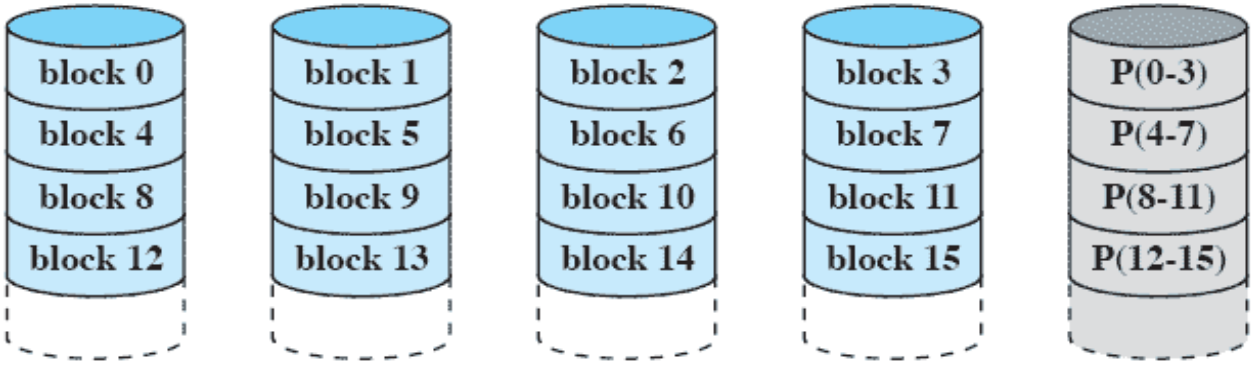
- Requires only a single redundant disk, no matter how large the disk array
- Employs parallel access, with data distributed in small strips
- Can achieve very high data transfer rates



(d) RAID 3 (bit-interleaved parity)

RAID Level 4

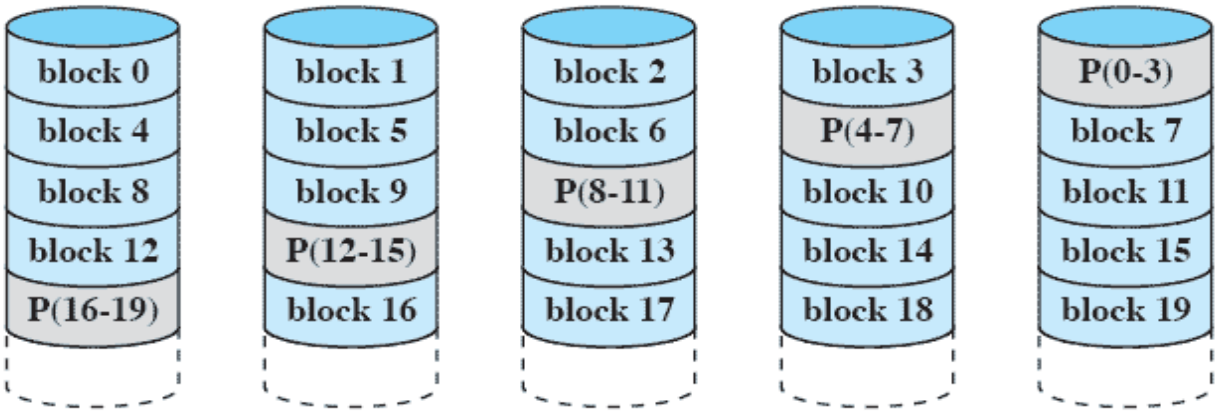
- Makes use of an independent access technique
- A bit-by-bit parity strip is calculated across corresponding strips on each data disk, and the parity bits are stored in the corresponding strip on the parity disk
- Involves a write penalty when an I/O write request of small size is performed



(e) RAID 4 (block-level parity)

RAID Level 5

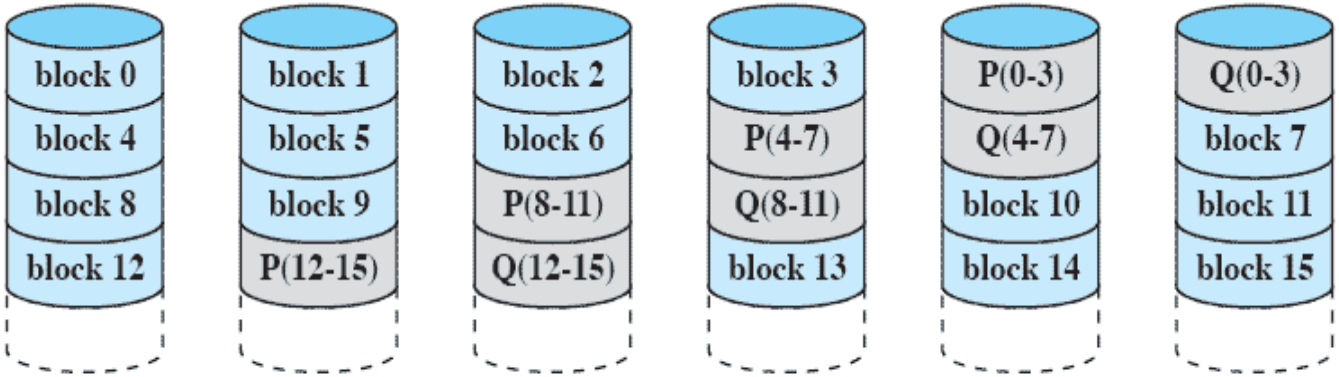
- Similar to RAID-4 but distributes the parity bits across all disks
- Typical allocation is a round-robin scheme
- Has the characteristic that the loss of any one disk does not result in data loss



(f) RAID 5 (block-level distributed parity)

RAID Level 6

- Two different parity calculations are carried out and stored in separate blocks on different disks
- Provides extremely high data availability
- Incurs a substantial write penalty because each write affects two parity blocks

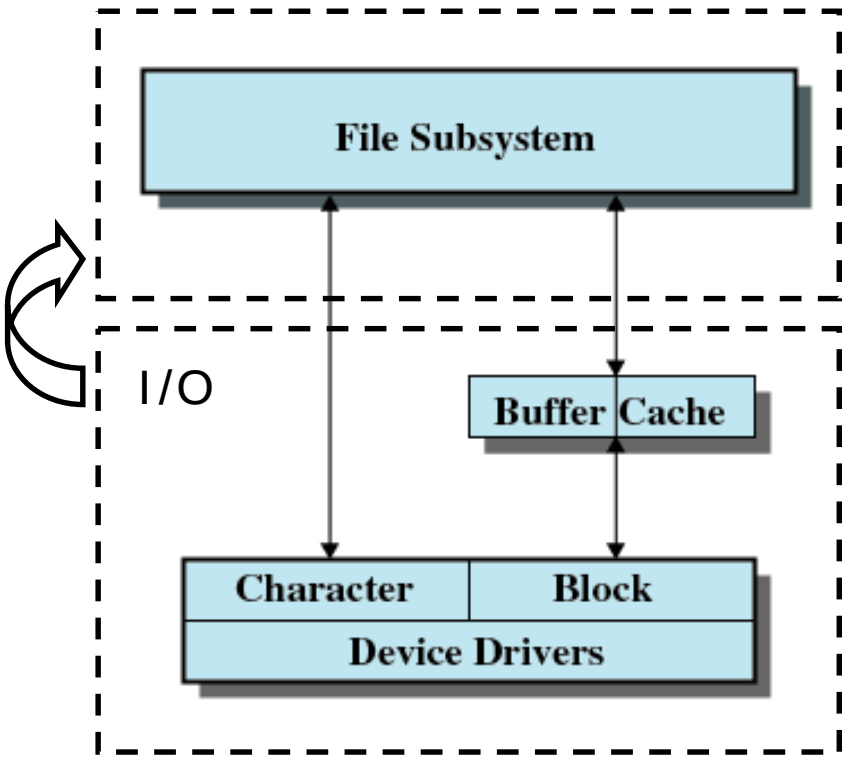


(g) RAID 6 (dual redundancy)

Example

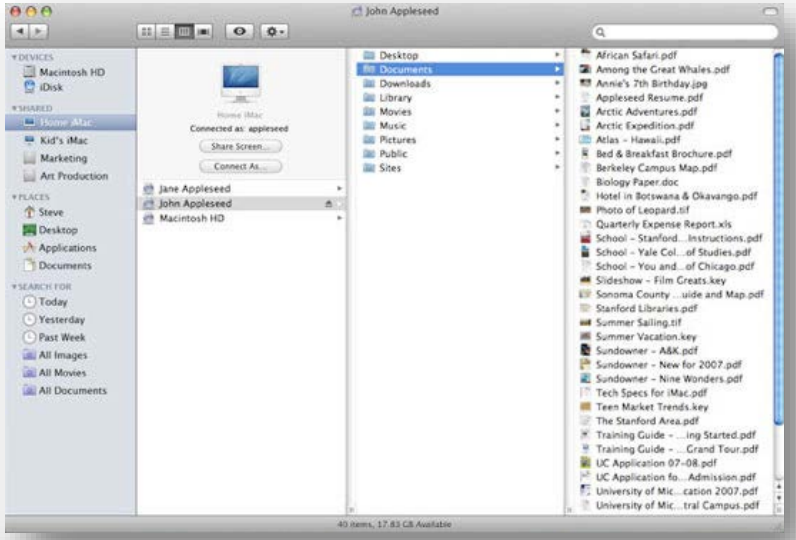
UNIX SVR4 I/O

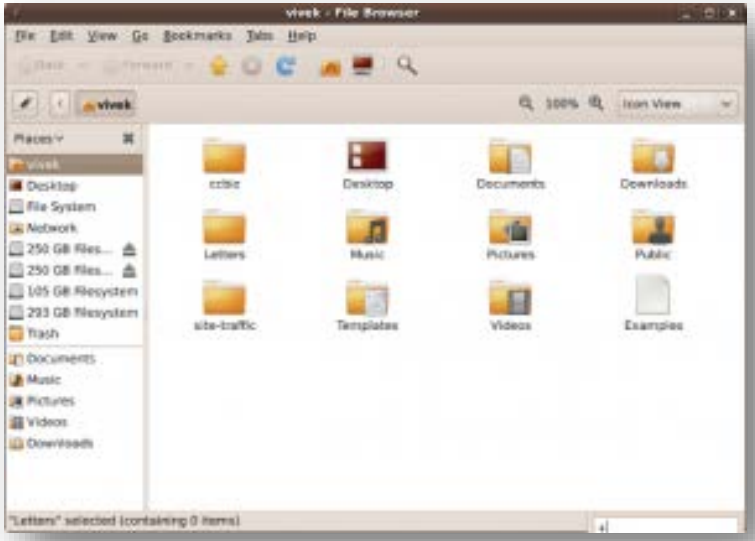
- Each individual device is associated with a special file:
  - /dev/dsp0: First sound card
  - /dev/hda: IDE, primary master
  - /dev/hda1: First partition on /dev/hda
  - /dev/tty: Controlling terminal
  - ...
- Two types of I/O:
  - Buffered (default)
  - Unbuffered (raw)

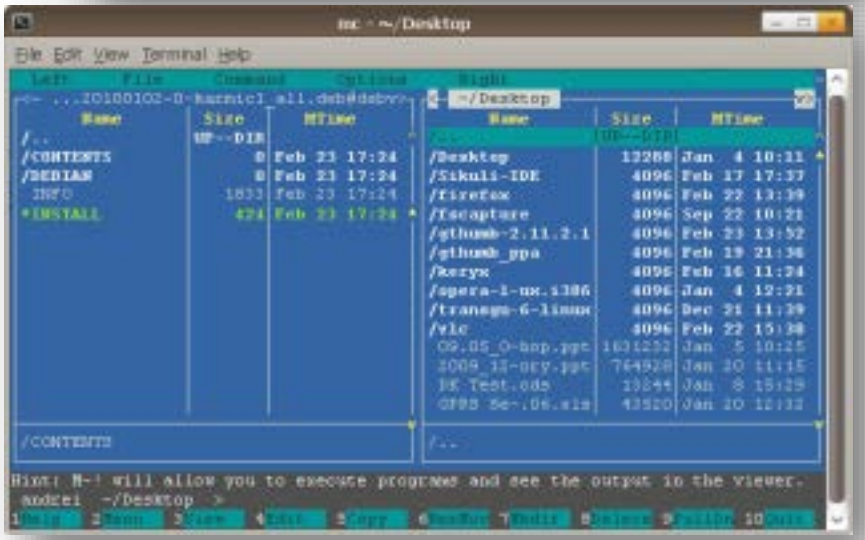


Content

File System Overview







Goals

- Meet data management needs and requirements of user
- Guarantee that data in file is valid (over time)
- Optimize performance
- Provide I/O support for variety of storage device types
- Minimize or eliminate the potential for lost or destroyed data (redundancy)
- Provide a standardized set of I/O interface routines
- Provide I/O support for multiple users
  - Concurrency, access control, etc.

Types of File Systems

- Disk File Systems
  - Windows: FAT, FAT16, FAT32, NTFS
  - Linux: ext, ext2, ext3
  - UNIX: UFS, ...
  - MAC OS X: HFS, HFS+
- Distributed File Systems
  - NFS, AFS, SMB
- Special Purpose File Systems

File System Overview

- Properties
  - Long-term existence
  - Sharable between processes
  - Structure (internal /organizational)

- Typical File Operations
  - Create Create new file
  - Delete Delete existing file
  - Open Open new/existing file
  - Close Close open file
  - Read Read data from open file
  - Write Write data to open file

Minimal User Requirements

- Each user …

1. should be able to create, delete, read, write and modify files
2. may have controlled access to other users' files
3. may control what type of accesses are allowed to the files
4. should be able to restructure the files in a form appropriate to the problem
5. should be able to move data between files
6. should be able to back up and recover files in case of damage
7. should be able to access his or her files by name rather than by numeric identifier

File System Architecture

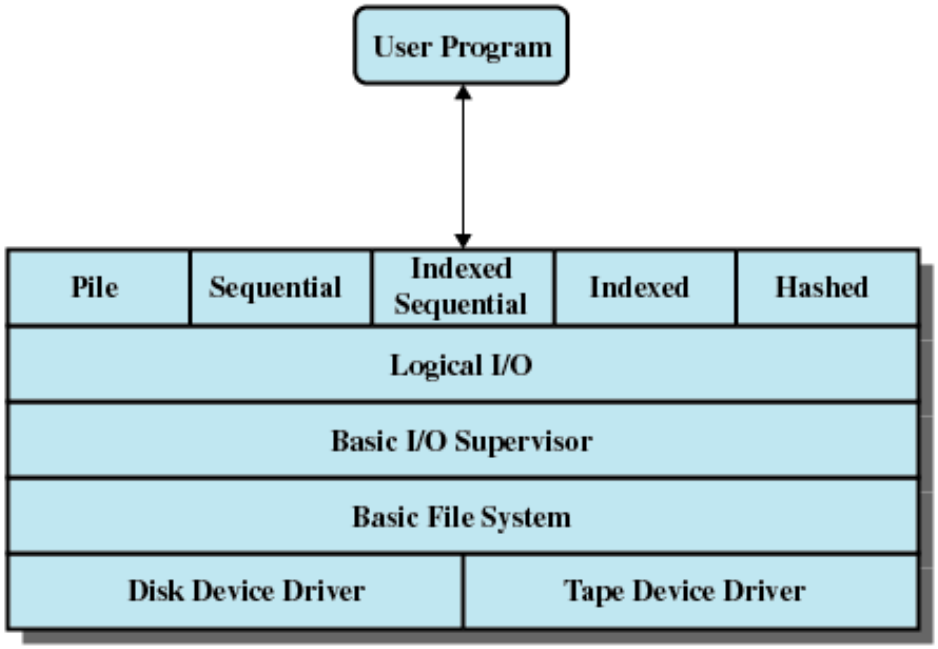


Figure 12.1 File System Software Architecture

- Access Method (API)
  - Reflect different file structures
  - Different ways to access data
- Logical I/O
  - Enables users to access records
  - General-purpose record I/O capability
  - Maintains basic data about file
- Basic I/O Supervisor
  - I/O initiation and termination
  - Selection of the I/O device
  - Scheduling to optimize performance
- Basic File System
  - Physical I/O
  - Placement of blocks
  - Buffers blocks in main memory
- Device Drivers
  - Communicate with peripheral devices
  - Responsible for starting I/O operations on a device
  - Processes completion of I/O request

Elements of File Management

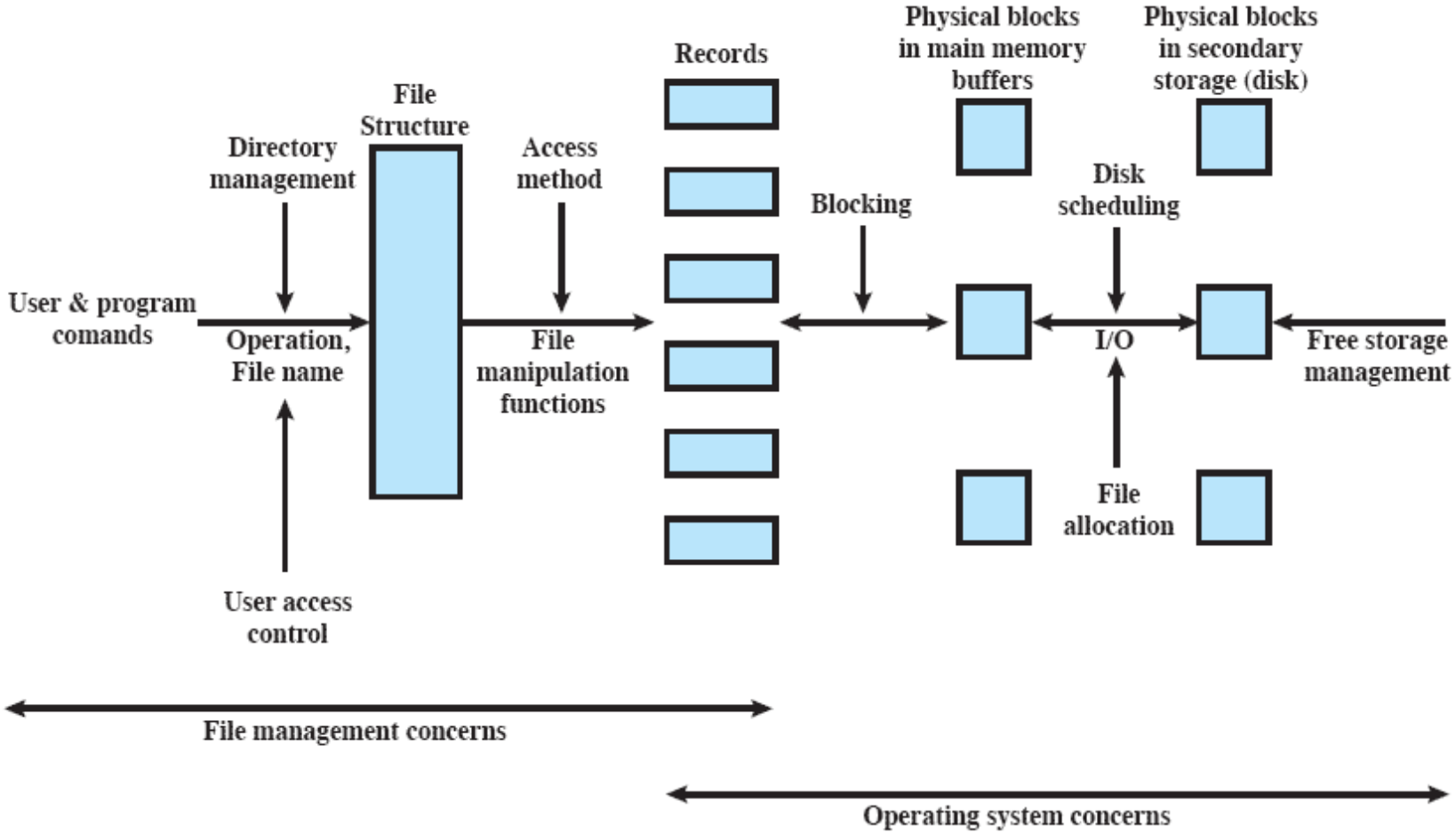
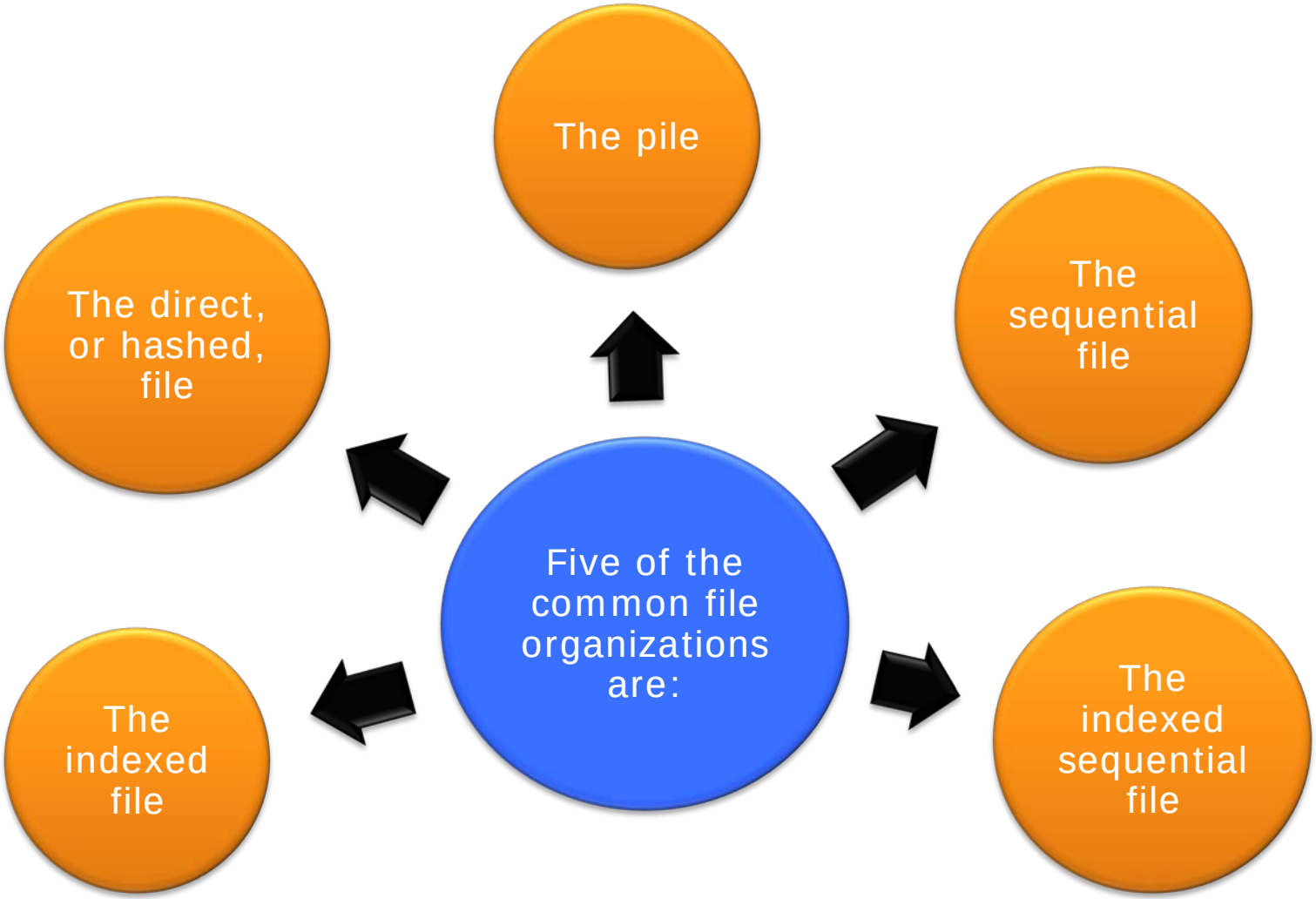


Figure 12.2 Elements of File Management

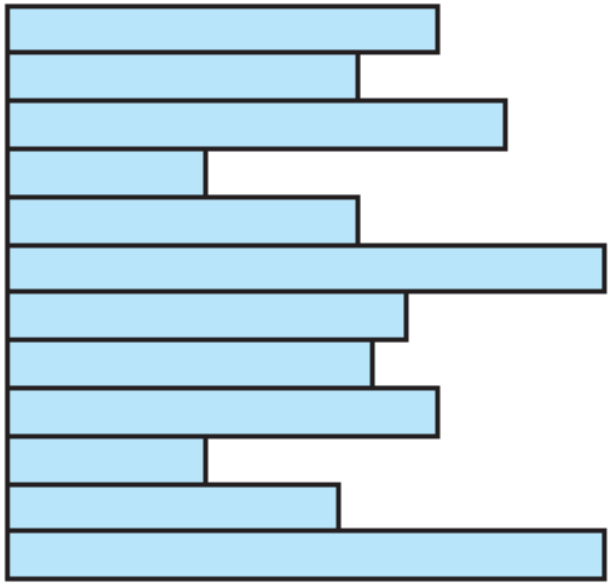
File Organization

File Organization Types



The Pile

- Least complicated form of file organization
- Data are collected in the order they arrive
- Each record consists of one burst of data
- Purpose is simply to accumulate the mass of data and save it
- Record access is by exhaustive search

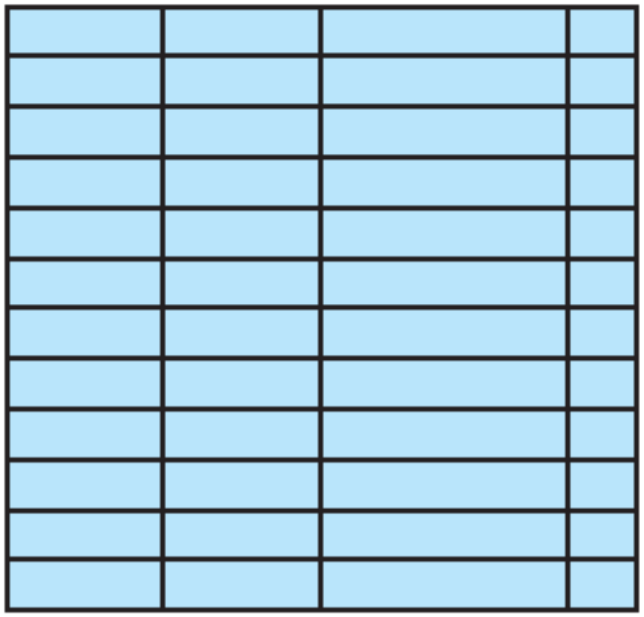


Variable-length records
Variable set of fields
Chronological order

(a) Pile File

The Sequential File

- Most common form of file structure
- A fixed format is used for records
- Key field uniquely identifies the record
- Typically used in batch applications
- Only organization that is easily stored on tape as well as disk

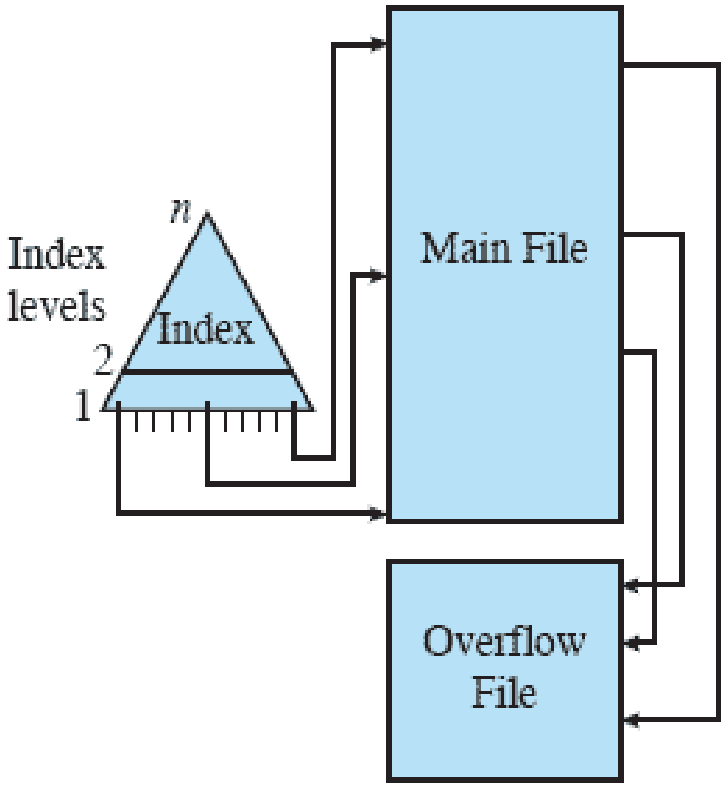


Fixed-length records
Fixed set of fields in fixed order
Sequential order based on key field

(b) Sequential File

Indexed Sequential File

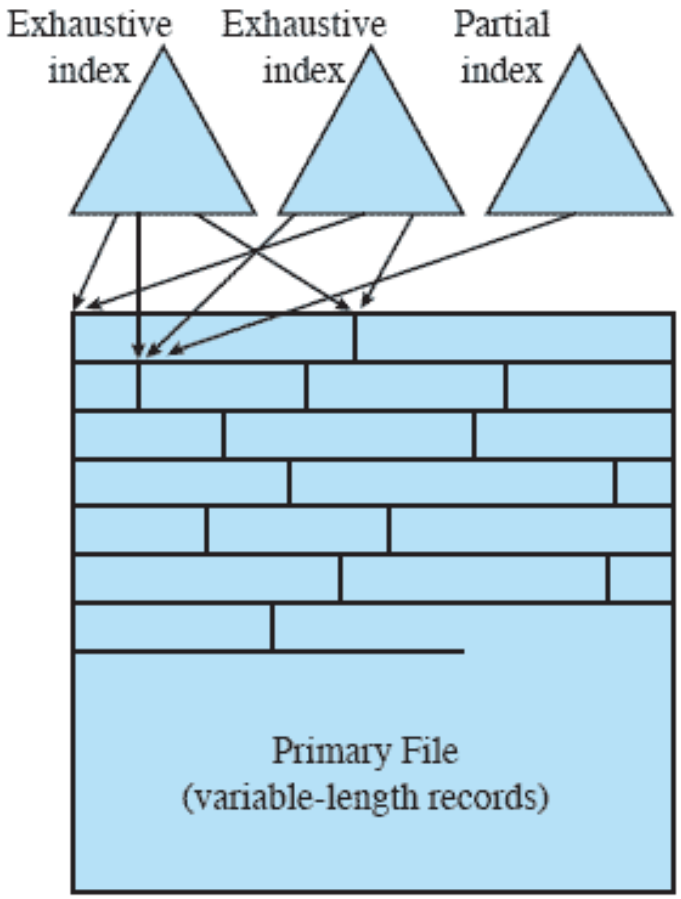
- Adds an index to the file to support random access
- Adds an overflow file
- Greatly reduces the time required to access a single record
- Multiple levels of indexing can be used to provide greater efficiency in access



(c) Indexed Sequential File

Indexed File

- Records are accessed only through their indexes
- Variable-length records can be employed
- Exhaustive index contains one entry for every record in the main file
- Partial index contains entries to records where the field of interest exists
- Used mostly in applications where timeliness of information is critical
- Examples would be airline reservation systems and inventory control systems



(d) Indexed File

Direct or Hashed File

- Access directly any block of a known address
- Makes use of hashing on the key value
- Often used where:
  - very rapid access is required
  - fixed-length records are used
  - records are always accessed one at a time

Examples are:

- directories
- pricing tables
- schedules
- name lists

Grades of Performance

Table 12.1 Grades of Performance for Five Basic File Organizations [WIED87]

	Space		Update		Retrieval		
	Attributes		Record Size				
File Method	Variable	Fixed	Equal	Greater	Single record	Subset	Exhaustive
Pile	A	B	A	E	E	D	B
Sequential	F	A	D	F	F	D	A
Indexed sequential	F	B	B	D	B	D	B
Indexed	B	C	C	C	A	B	D
Hashed	F	B	B	F	B	F	E

A	=	Excellent, well suited to this purpose	$\approx O(r)$
B	=	Good	$\approx O(o \times r)$
C	=	Adequate	$\approx O(r \log n)$
D	=	Requires some extra effort	$\approx O(n)$
E	=	Possible with extreme effort	$\approx O(r \times n)$
F	=	Not reasonable for this purpose	$\approx O(n^{>1})$

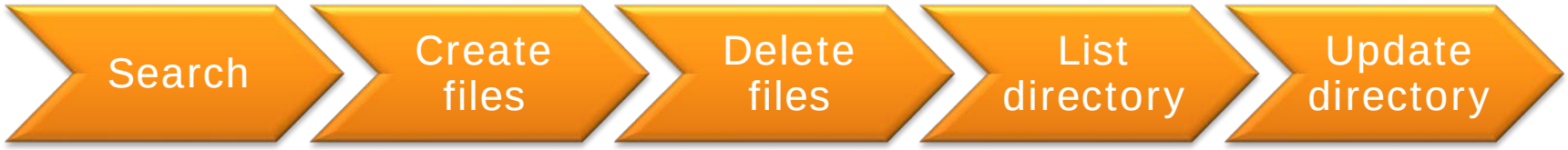
where

r = size of the result
o = number of records that overflow
n = number of records in file

Directories

Operations Performed on a Directory

- To understand the requirements for a file structure, it is helpful to consider the types of operations that may be performed on the directory:

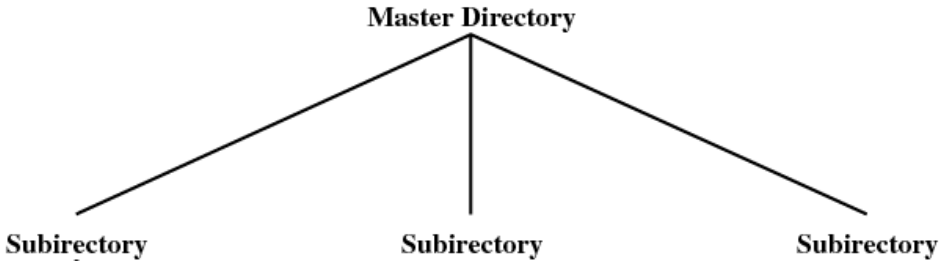


File Directories

- Contains information about files
  - Attributes, e.g., read/write/executable bits, access time
  - Ownership, e.g., user/group or Access Control List (ACL)
  - Location with regard to logical structure of medium
- Directory itself may be implemented as file owned by operating system
- Provides mapping between file names and files themselves
  - “inodes” in Unix
  - ➢ One file can have multiple names (“hard links”)
- Structure
  - List of entries, one for each file
  - Sequential file with name of file serving as key
  - Initially no support for organizing files (except for naming)
    - Forces user to be careful not to use the same name for two different files

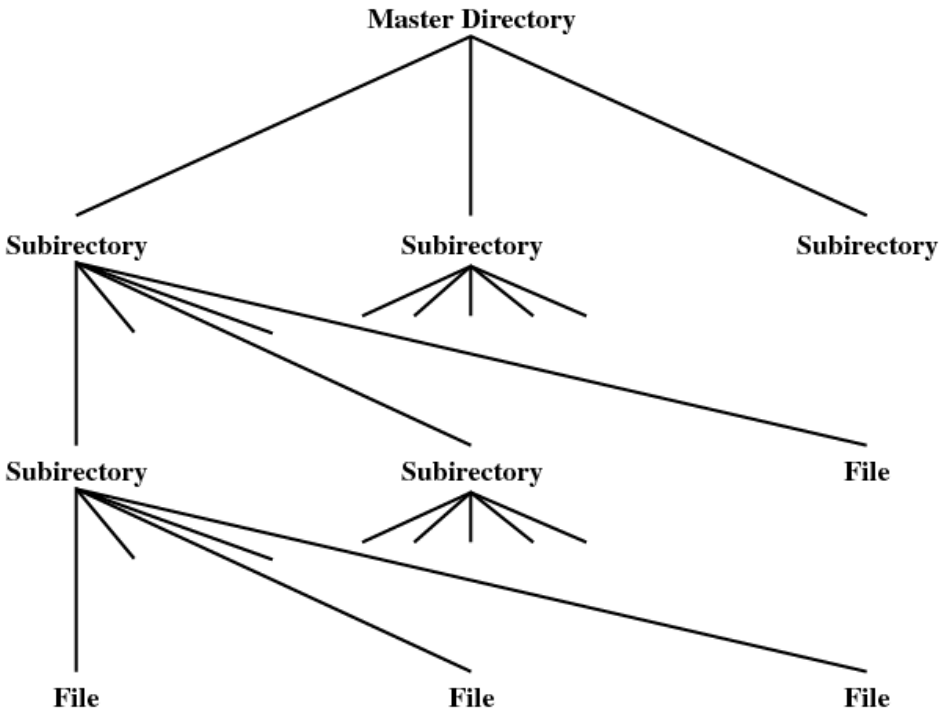
Multi-Level Directories

- Two-level Scheme (hist.)
  - One directory for each user and a master directory
  - Master directory contains entry for each user
  - Each user directory is a simple list of files for that user
  - ➢ Provides no help in structuring collections of files



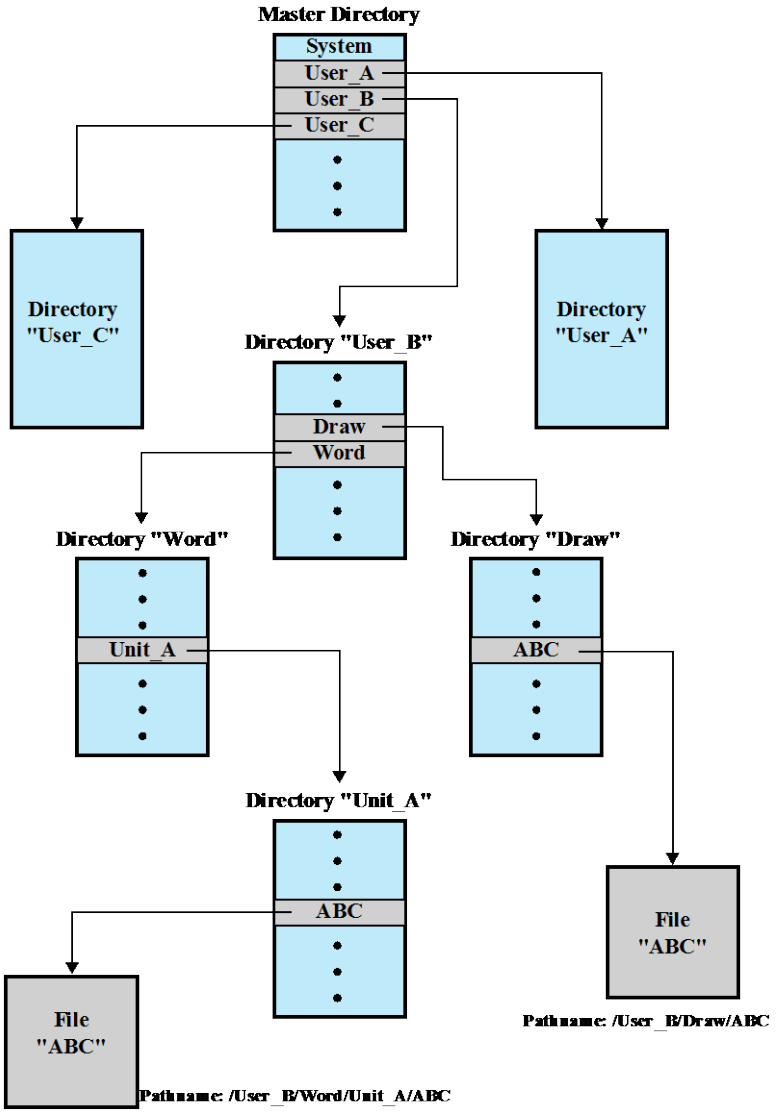
Multi-Level Directories

- Hierarchical / Tree-Structure
  - Master directory with user directories underneath it
  - Each user directory may have subdirectories and files as entries
  - Some OSes use multiple trees with own identifiers, e.g., drive letters (A:, C:)



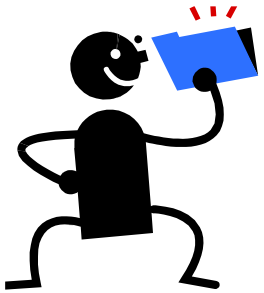
Hierarchical/Tree-Structured Directory

- Files are located by following path from root (master) directory down various branches
  - *Pathname* of file
- Supports several files with same file name as long as path names differ
- Per-process current directory is working directory
- Files are referenced relative to current working directory (CWD)



File Sharing

File Sharing



Two issues arise when allowing files to be shared among a number of users

- Access rights
- Management of simultaneous access

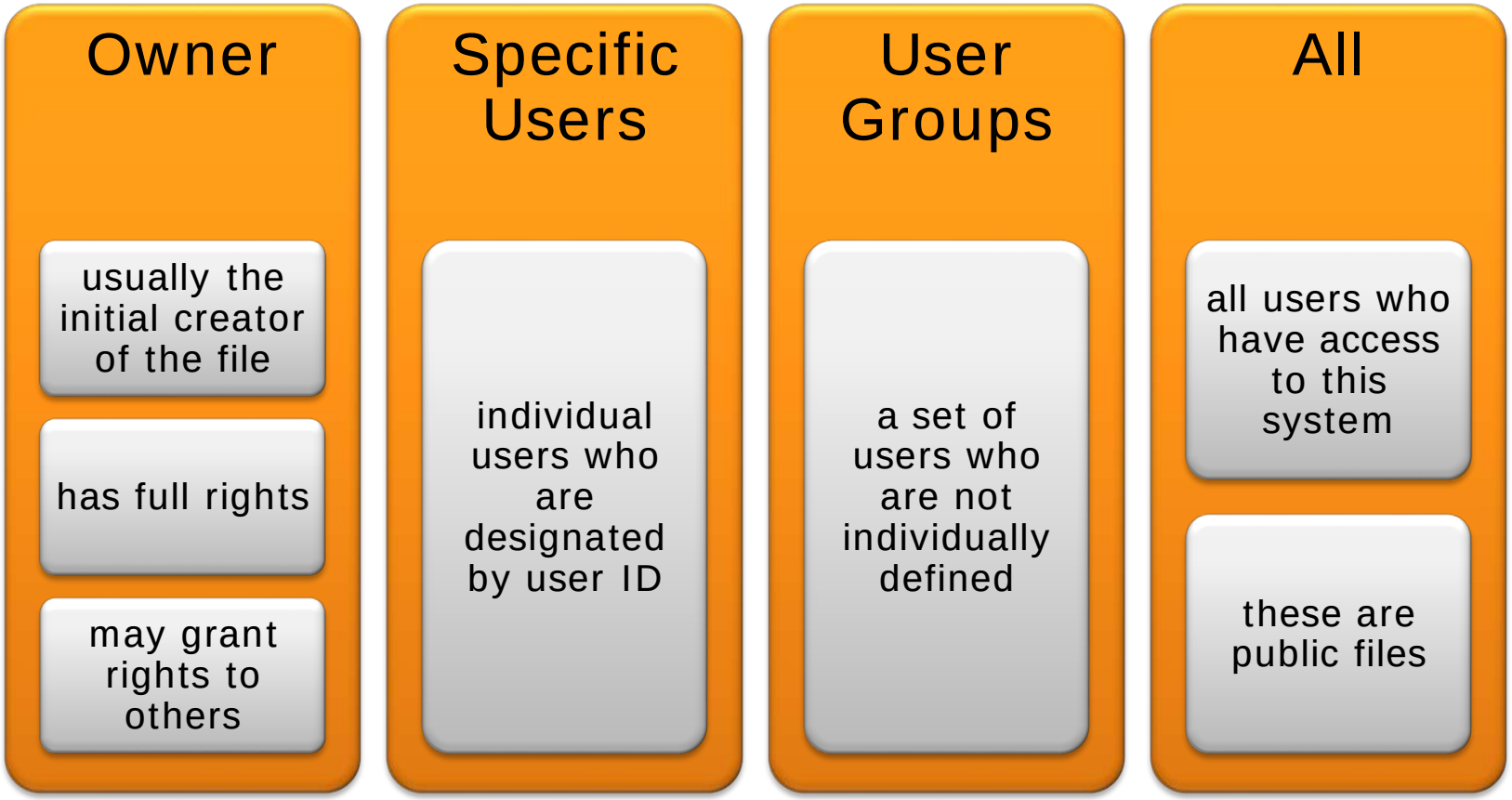
Access Rights

- None
  - User may not know of existence of file
  - User is not allowed to read user directory that includes file
- Knowledge
  - User can only determine that file exists and who its owner is
- Execution
  - User can load and execute program but cannot copy it
- Reading
  - User can read file for any purpose, including copying and execution
- Appending
  - User can add data to file but cannot modify or delete any of its contents
- Updating
  - User can modify, delete and add to file's data
  - Includes creating file, rewriting it and removing all or part of its data
- Changing protection
  - User can change access rights granted to other users
- Deletion
  - User can delete a file
- Owner
  - All rights previously listed
  - Grant rights to others using classes of users:
    - Specific user
    - User groups
    - Everybody

- Complex access policies implemented with Access Control Lists (ACLs)

➢ Watch out for semantic differences between files and directories!

User Access Rights



Access Matrix

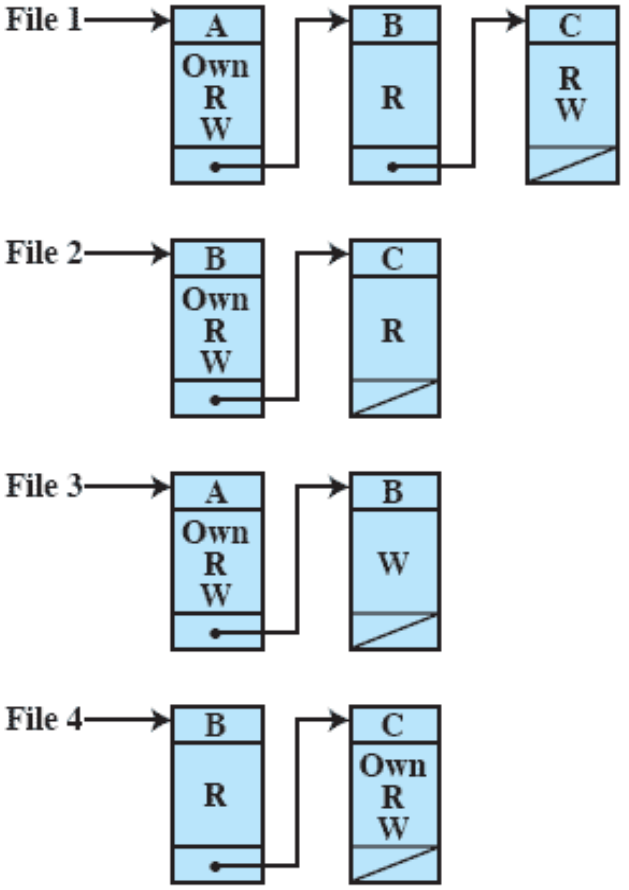
- The basic elements are
  - subject: an entity capable of accessing objects
  - object: anything to which access is controlled
  - access right: the way in which an object is accessed by a subject

	File 1	File 2	File 3	File 4	Account 1	Account 2
User A	Own R W		Own R W		Inquiry Credit	
User B	R	Own R W	W	R	Inquiry Debit	Inquiry Credit
User C	R W	R		Own R W		Inquiry Debit

(a) Access matrix

Access Control Lists

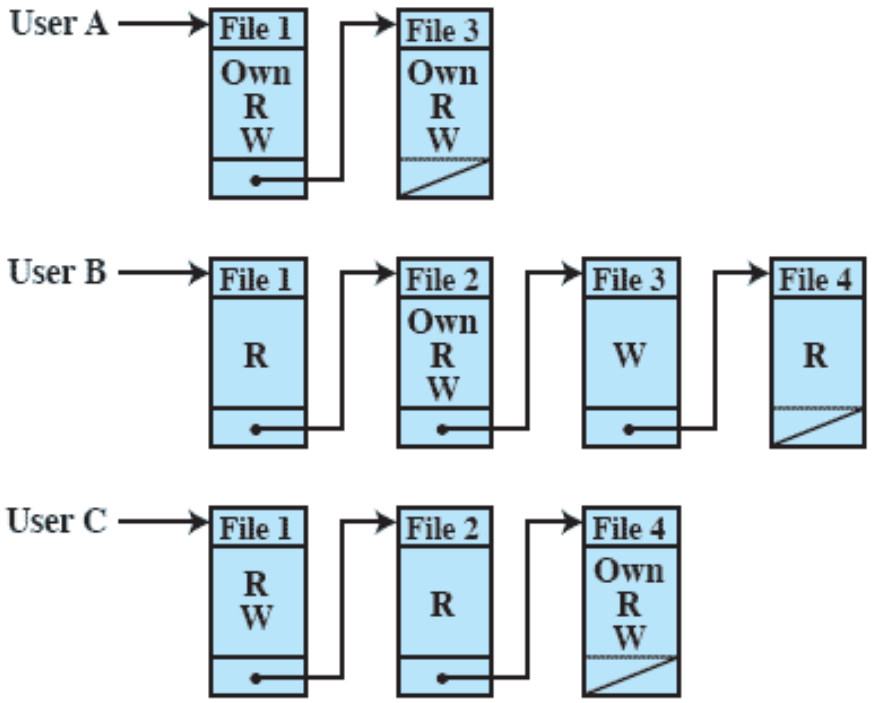
- A matrix may be decomposed by columns, yielding access control lists
- The access control list lists users and their permitted access rights



(b) Access control lists for files of part (a)

Capability Lists

- Decomposition by rows yields capability tickets
- A capability ticket specifies authorized objects and operations for a user

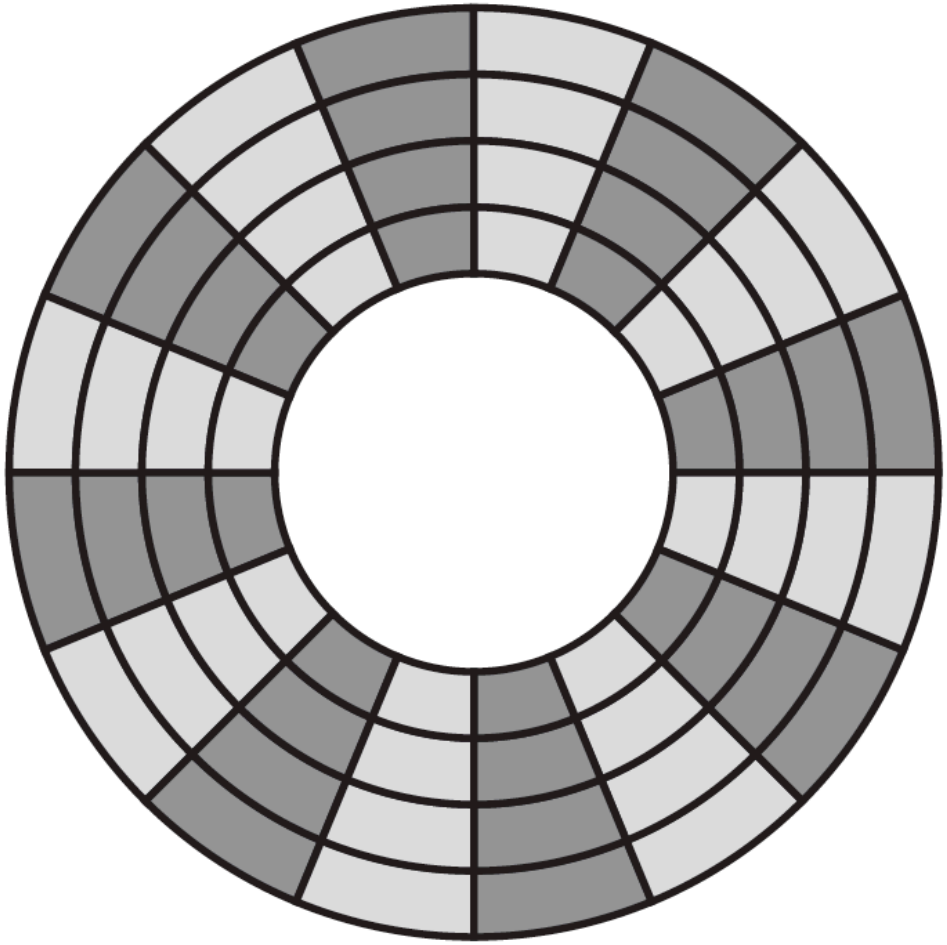


(c) Capability lists for files of part (a)

Secondary Storage Management

Secondary Storage Management

- Secondary storage space must be allocated to files
- Must keep track of space available for allocation



File Allocation

- On secondary storage, a file consists of a collection of blocks
- The operating system or file management system is responsible for allocating blocks to files
- The approach taken for file allocation may influence the approach taken for free space management
- Space is allocated to a file as one or more portions (contiguous set of allocated blocks)
- File allocation table (FAT)
  - data structure used to keep track of the portions assigned to a file

Preallocation vs Dynamic Allocation

- A preallocation policy requires that the maximum size of a file be declared at the time of the file creation request

- For many applications it is difficult to estimate reliably the maximum potential size of the file
  - Tends to be wasteful because users and application programmers tend to overestimate size

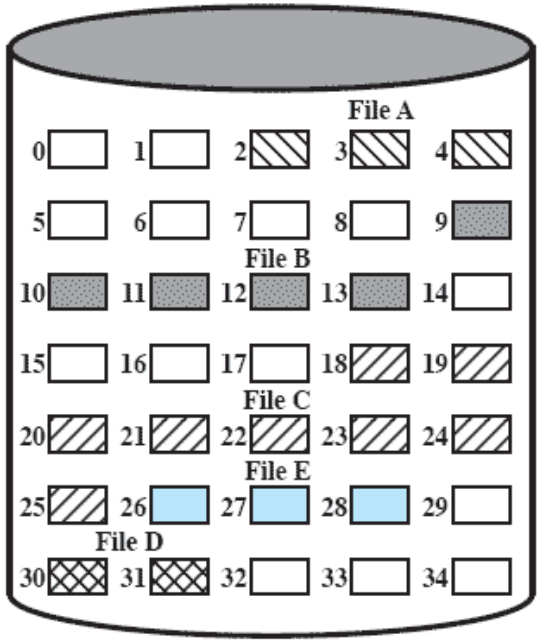
- Dynamic allocation allocates space to a file in portions as needed

File Allocation Methods

- Contiguous allocation
  - Single set of blocks is allocated to a file at time of creation
  - Single entry in file allocation table (starting block, length of file)
  - ➢ Incurs fragmentation; changing size of a file is expensive
- Chained allocation
  - Allocation on basis of individual block
  - Each block contains a pointer to next block in chain
  - Single entry in file allocation table (starting block, length of file)
  - ➢ Seeking within file (random access) is expensive
- Indexed allocation
  - File allocation table contains a separate one-level index for each file
  - The index has one entry for each portion allocated to file
  - The file allocation table contains block number for index
  - ➢ Avoids problems mentioned above, incurs some storage overhead

Methods of File Allocation

- Contiguous File Allocation
  - A single contiguous set of blocks is allocated to a file at the time of file creation
  - Preallocation strategy using variable-size portions
  - Is the best from the point of view of the individual sequential file

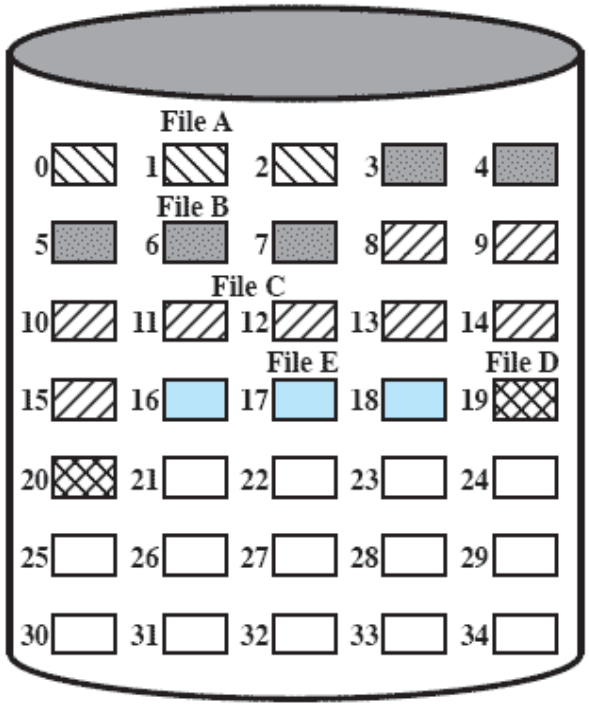


File Allocation Table

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

➢ External fragmentation on disk

Methods of File Allocation



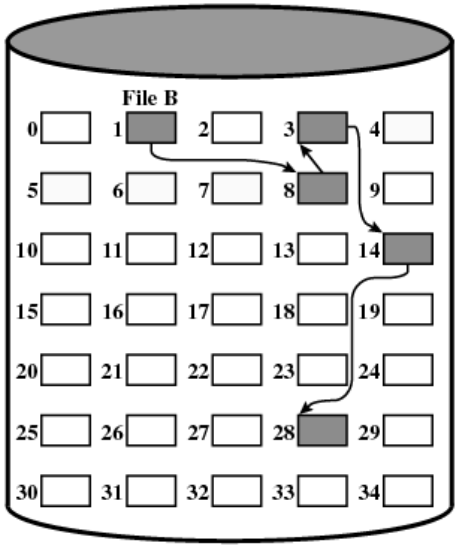
File Allocation Table

File Name	Start Block	Length
File A	0	3
File B	3	5
File C	8	8
File D	19	2
File E	16	3

Contiguous File Allocation (After Compaction)

Methods of File Allocation

- Chained Allocation
  - Allocation is on an individual block basis
  - Each block contains a pointer to the next block in the chain
  - The file allocation table needs just a single entry for each file
  - No external fragmentation to worry about
  - Best for sequential files

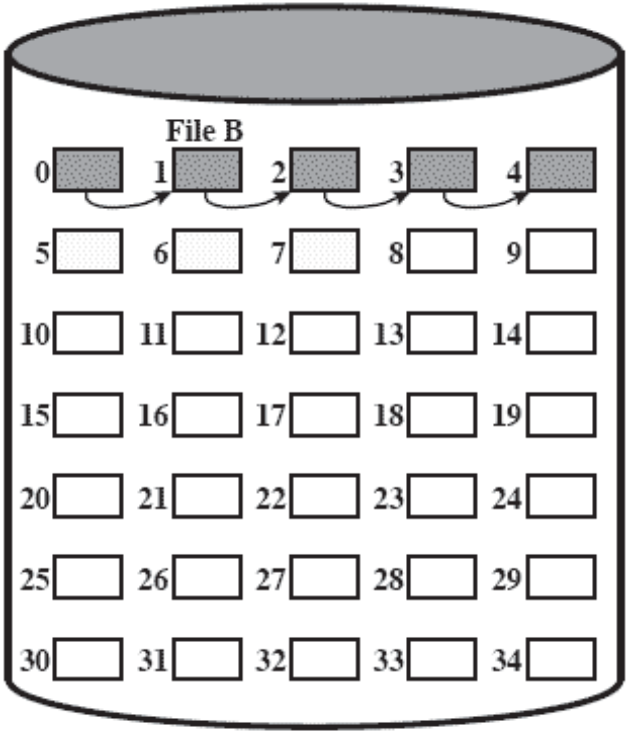


File Allocation Table

File Name	Start Block	Length
...	...	...
File B	1	5
...	...	...

➢ Low random access performance

Methods of File Allocation



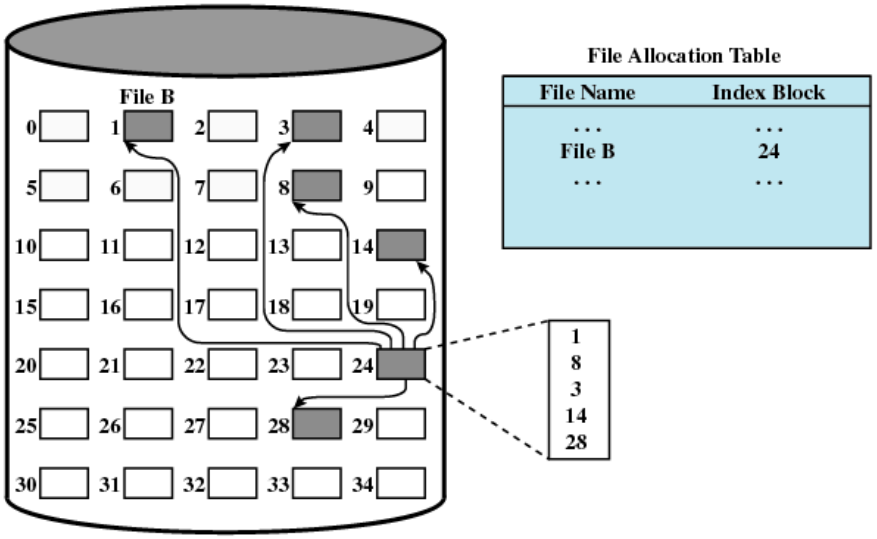
File Allocation Table

File Name	Start Block	Length
• • •	• • •	• • •
File B	0	5
• • •	• • •	• • •

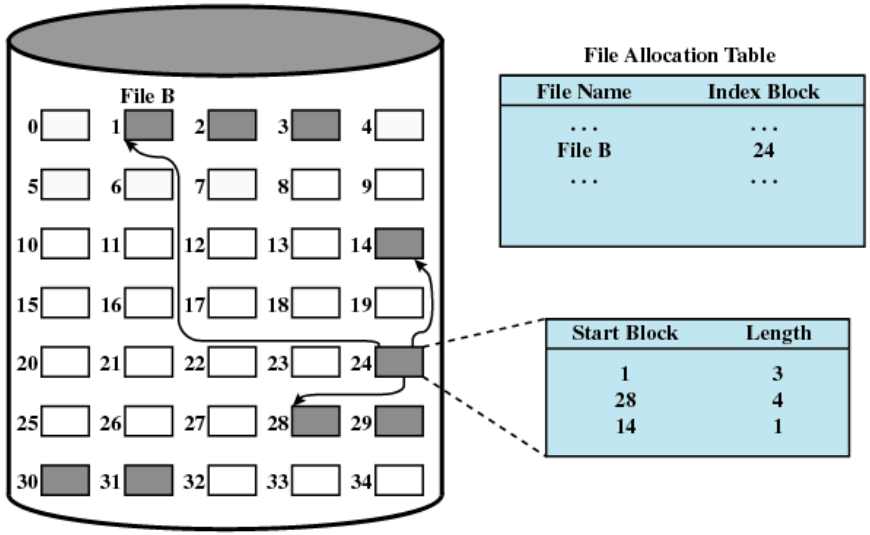
Chained Allocation After Consolidation

Methods of File Allocation

- Indexed Allocation with Block Portions



- Index Allocation with Variable-Length Portions



- Indexing overhead
- Good compromise

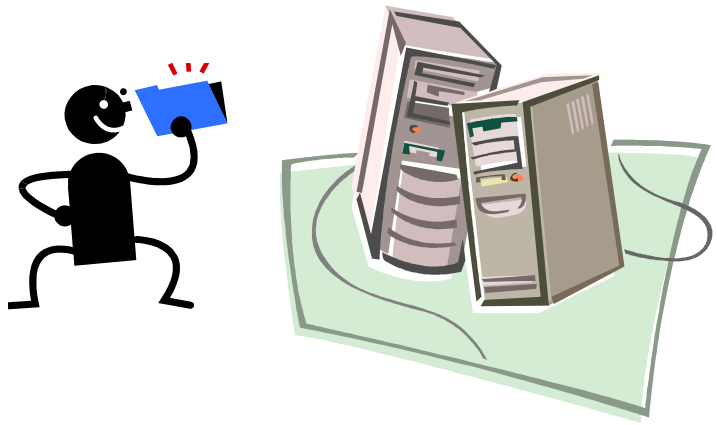
File Allocation Methods

	Contiguous	Chained	Indexed	
Preallocation?	Necessary	Possible	Possible	
Fixed or variable size portions?	Variable	Fixed blocks	Fixed blocks	Variable
Portion size	`Large`	Small	Small	Medium
Allocation frequency	Once	Low to high	High	Low
Time to allocate	Medium	Long	Short	Medium
File allocation table size	One entry	One entry	Large	Medium

Free Space Management

Free Space Management

- Just as allocated space must be managed, so must the unallocated space
- To perform file allocation, it is necessary to know which blocks are available
- A **disk allocation table** is needed in addition to a file allocation table



Bit Tables

- This method uses a vector containing one bit for each block on the disk
- Each entry of a 0 corresponds to a free block, and each 1 corresponds to a block in use
- Advantages
  - works well with any file allocation method
  - it is as small as possible

Chained Free Portions

- The free portions may be chained together by using a pointer and length value in each free portion
- Negligible space overhead because there is no need for a disk allocation table
- Suited to all file allocation methods

- Disadvantages
  - leads to fragmentation
  - every time you allocate a block you need to read the block first to recover the pointer to the new first free block before writing data to that block

Indexing

- Treats free space as a file and uses an index table as it would for file allocation
- For efficiency, the index should be on the basis of variable-size portions rather than blocks
- This approach provides efficient support for all of the file allocation methods

Free Block List

Each block is assigned a number sequentially

- the list of the numbers of all free blocks is maintained in a reserved portion of the disk

Depending on the size of the disk, either 24 or 32 bits will be needed to store a single block number

- the size of the free block list is 24 or 32 times the size of the corresponding bit table and must be stored on disk

There are two effective techniques for storing a small part of the free block list in main memory:

- the list can be treated as a push-down stack with the first few thousand elements of the stack kept in main memory
- the list can be treated as a FIFO queue, with a few thousand entries from both the head and the tail of the queue in main memory

UNIX File Management

UNIX File Management

- In the UNIX file system, six types of files are distinguished:

Regular, or ordinary
- contains arbitrary data in zero or more data blocks

Directory
- contains a list of file names plus pointers to associated inodes

Special
- contains no data but provides a mechanism to map physical devices to file names

Named pipes
- an interprocess communications facility

Links
- an alternative file name for an existing file

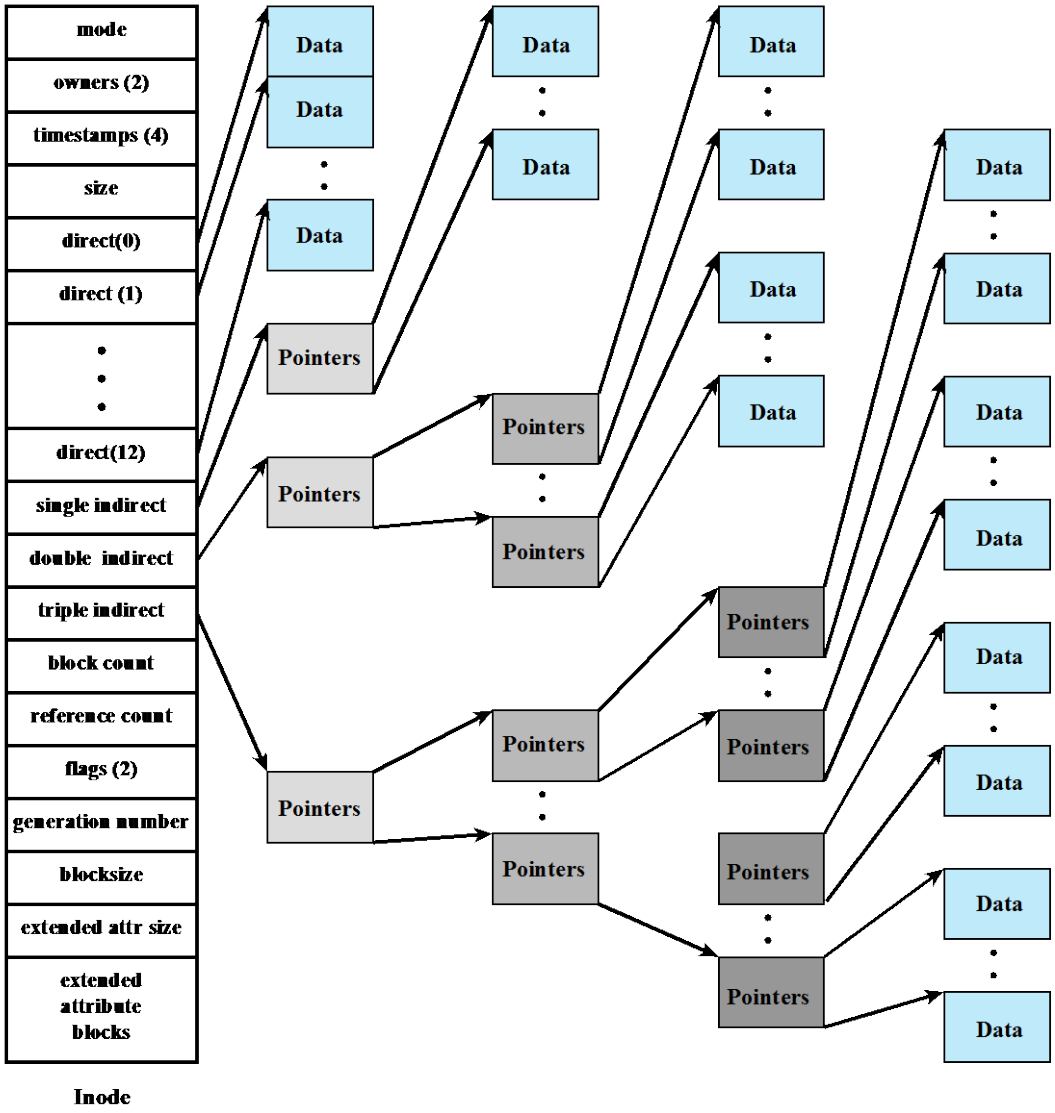
Symbolic links
- a data file that contains the name of the file it is linked to

Inodes

- All types of UNIX files are administered by the OS by means of **inodes**
- An inode (index node) is a control structure that contains the key information needed by the operating system for a particular file
- Several file names may be associated with a single inode
  - an active inode is associated with exactly one file
  - each file is controlled by exactly one inode

File Mode	16-bit flag that stores access and execution permissions associated with the file. 12-14 File type (regular, directory, character or block special, FIFO pipe 9-11 Execution flags 8 Owner read permission 7 Owner write permission 6 Owner execute permission 5 Group read permission 4 Group write permission 3 Group execute permission 2 Other read permission 1 Other write permission 0 Other execute permission
Link Count	Number of directory references to this inode
Owner ID	Individual owner of file
Group ID	Group owner associated with this file
File Size	Number of bytes in file
File Addresses	39 bytes of address information
Last Accessed	Time of last file access
Last Modified	Time of last file modification
Inode Modified	Time of last inode modification

FreeBSD Inode and File Structure



File Allocation

- File allocation is done on a block basis
- Allocation is dynamic, as needed, rather than using preallocation
- An indexed method is used to keep track of each file, with part of the index stored in the inode for the file
- In all UNIX implementations the inode includes a number of direct pointers and three indirect pointers (single, double, triple)

UNIX Directories and Inodes

- Directories are structured in a hierarchical tree
- Each directory can contain files and/or other directories
- A directory that is inside another directory is referred to as a subdirectory

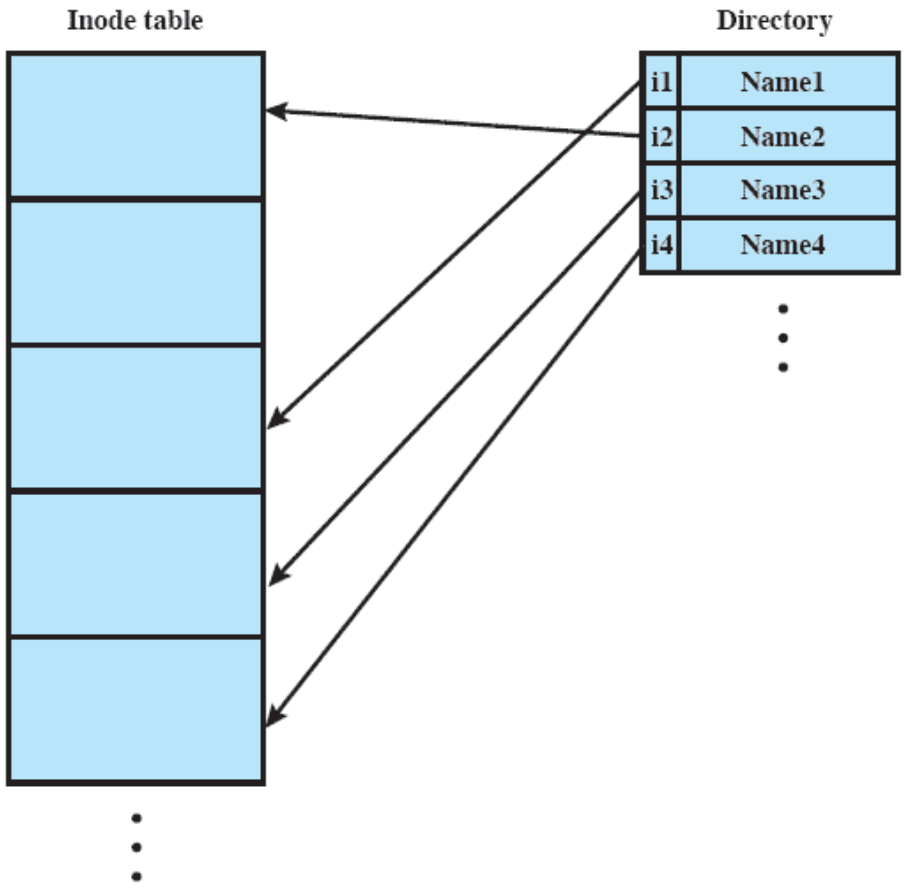
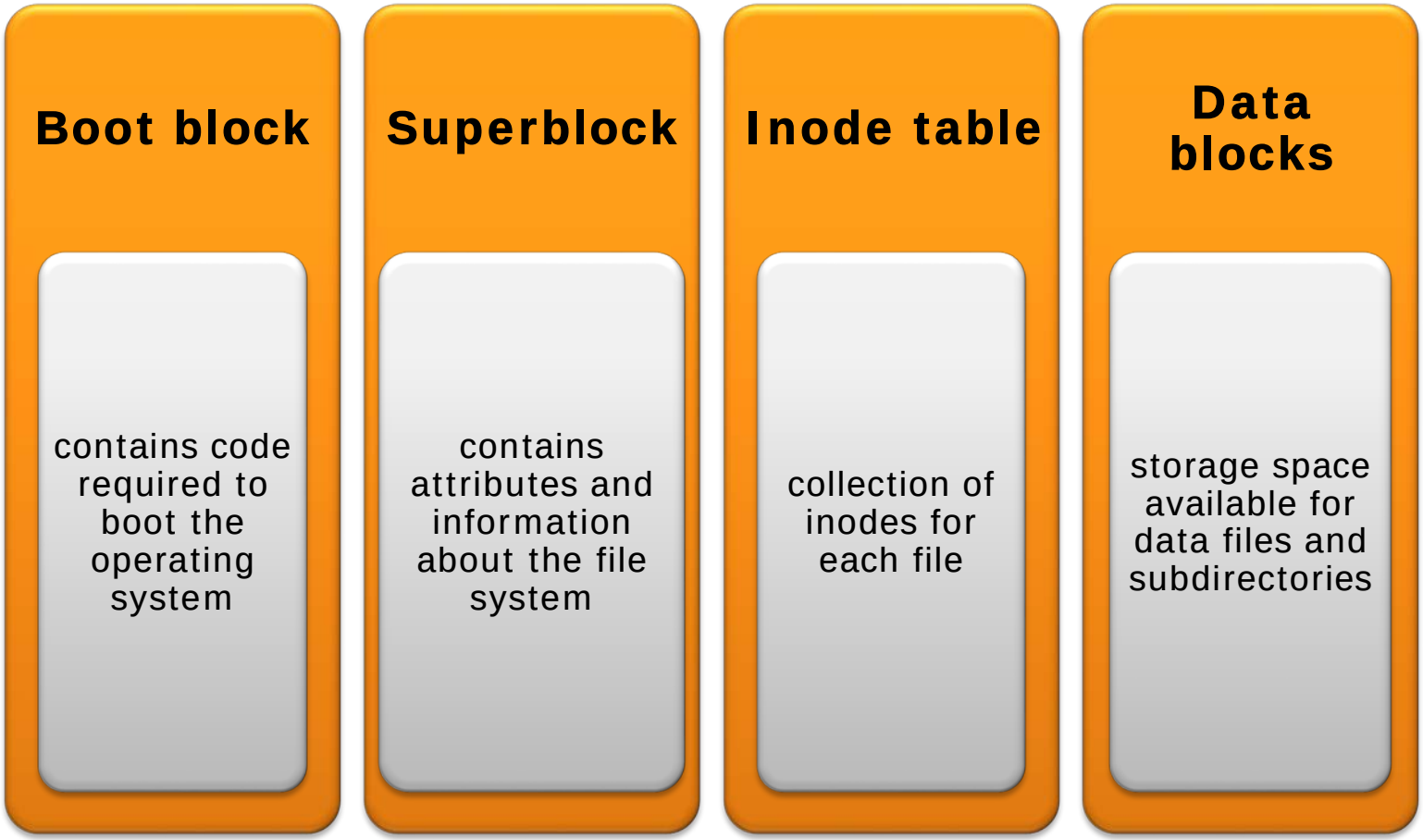


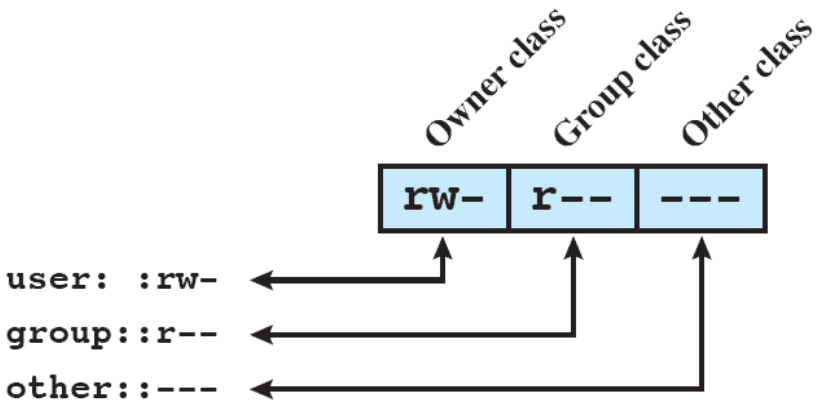
Figure 12.15 UNIX Directories and Inodes

Volume Structure

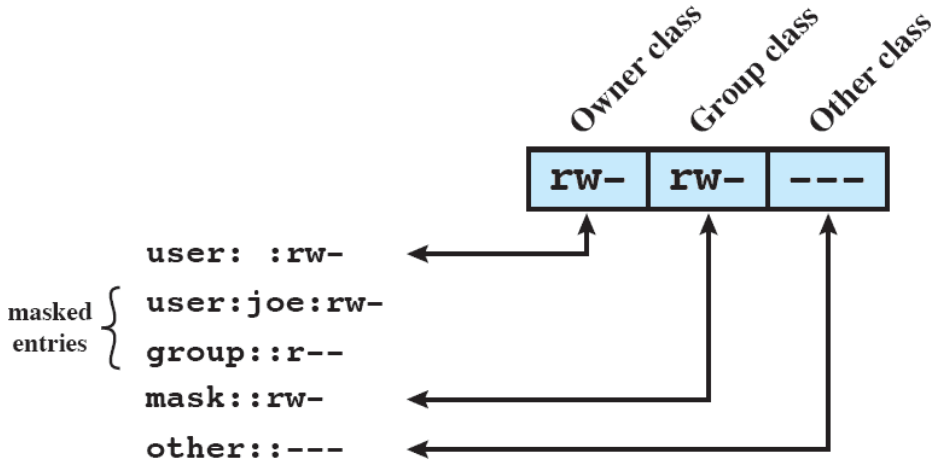
- A UNIX file system resides on a single logical disk or disk partition and is laid out with the following elements:



UNIX File Access Control

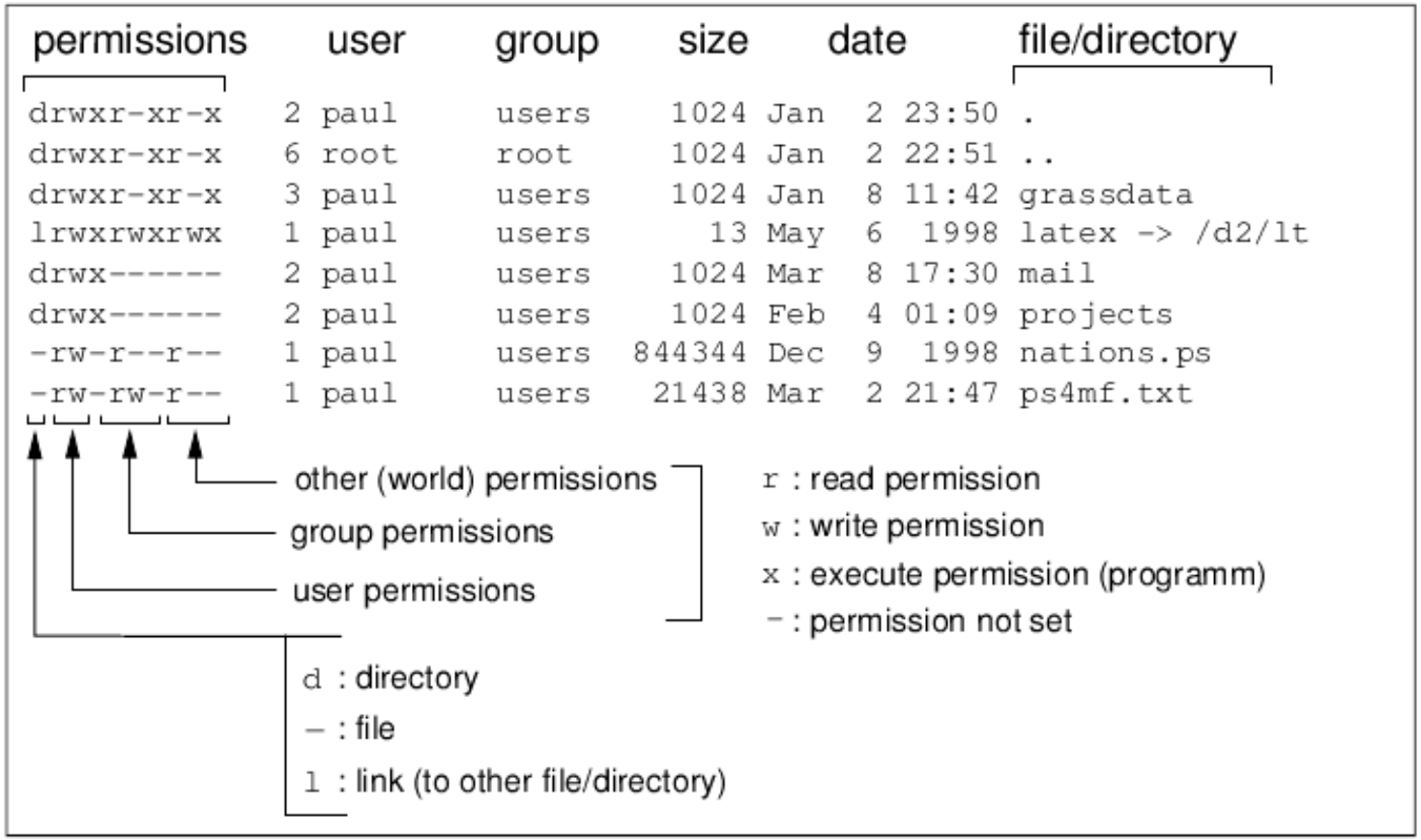


(a) Traditional UNIX approach (minimal access control list)



(b) Extended access control list

UNIX File Access Control



Access Control Lists in UNIX

- FreeBSD allows the administrator to assign a list of UNIX user IDs and groups to a file
- Any number of users and groups can be associated with a file, each with three protection bits (read, write, execute)
- A file may be protected solely by the traditional UNIX file access mechanism
- FreeBSD files include an additional protection bit that indicates whether the file has an extended ACL

Linux Virtual File System (VFS)

- Presents a single, uniform file system interface to user processes
- Defines a common file model that is capable of representing any conceivable file system's general feature and behavior
- Assumes files are objects that share basic properties regardless of the target file system or the underlying processor hardware

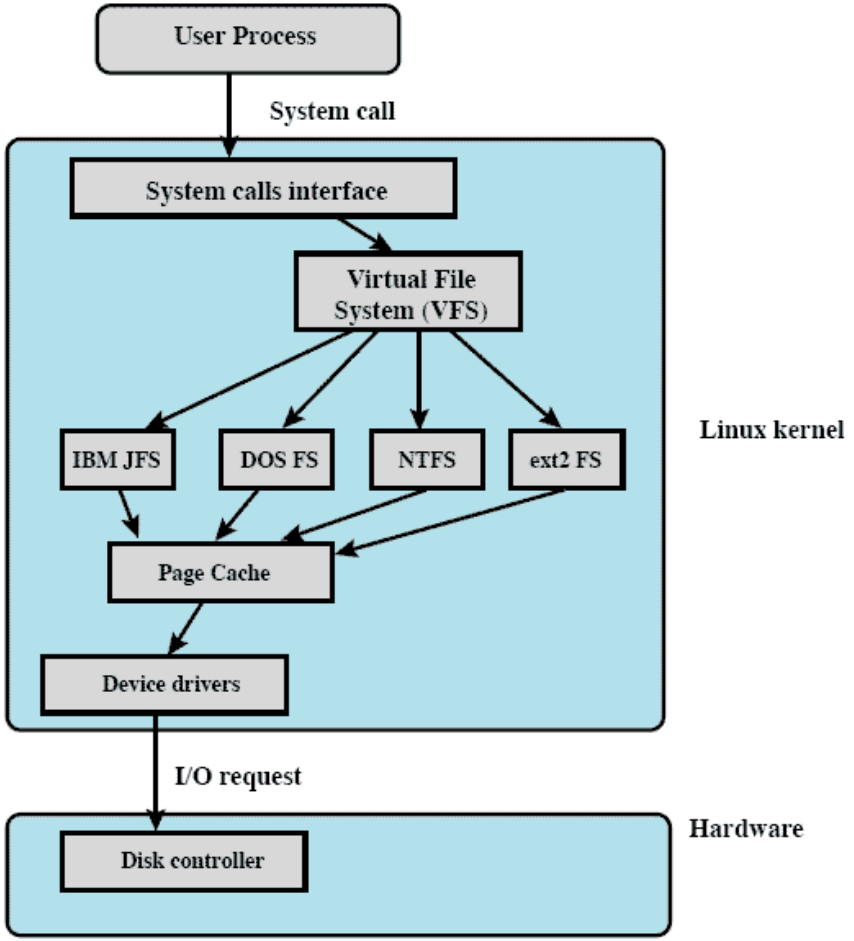
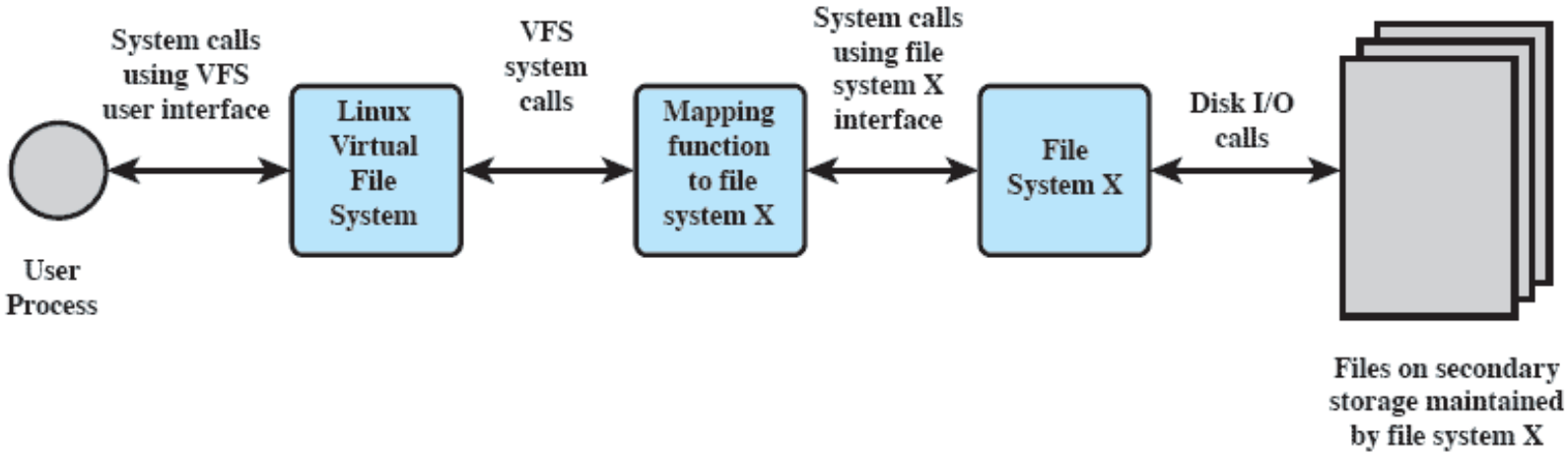


Figure 12.17 Linux Virtual File System Context

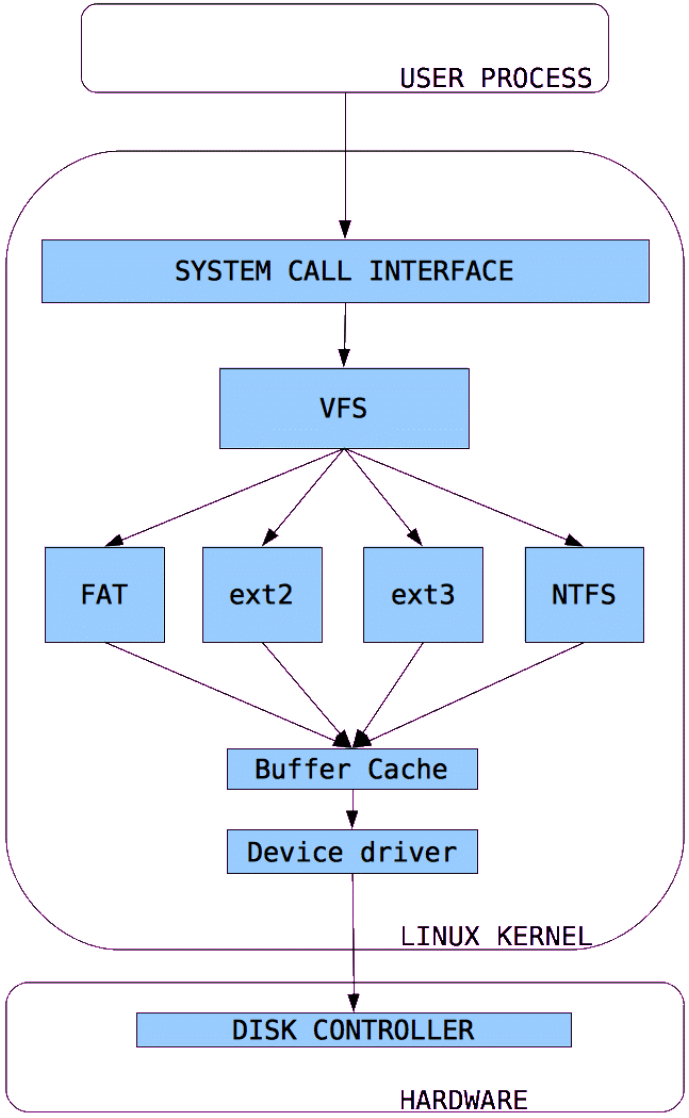
The Role of VFS within the Kernel



Example Linux File System

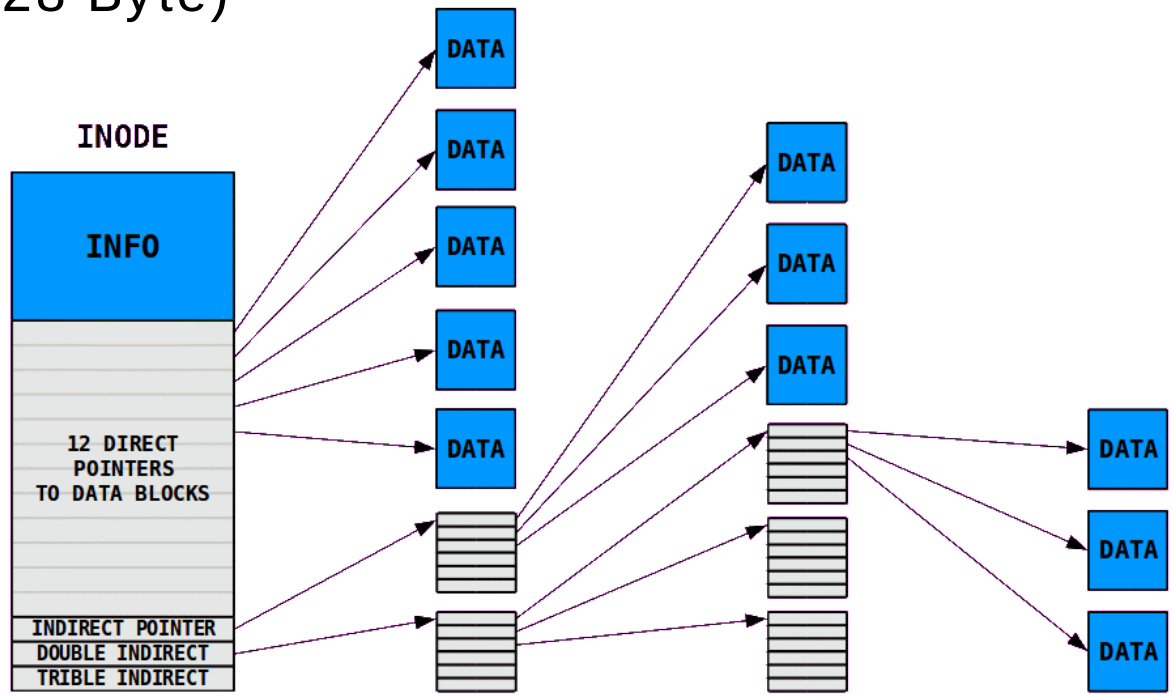
Example: Linux VFS/Ext2

- The virtual file system is a layer between the kernel and the file system code
- Manages all the different file systems that are mounted
- The real file systems are either built into the kernel itself or are built as loadable modules



Ext2: Inodes

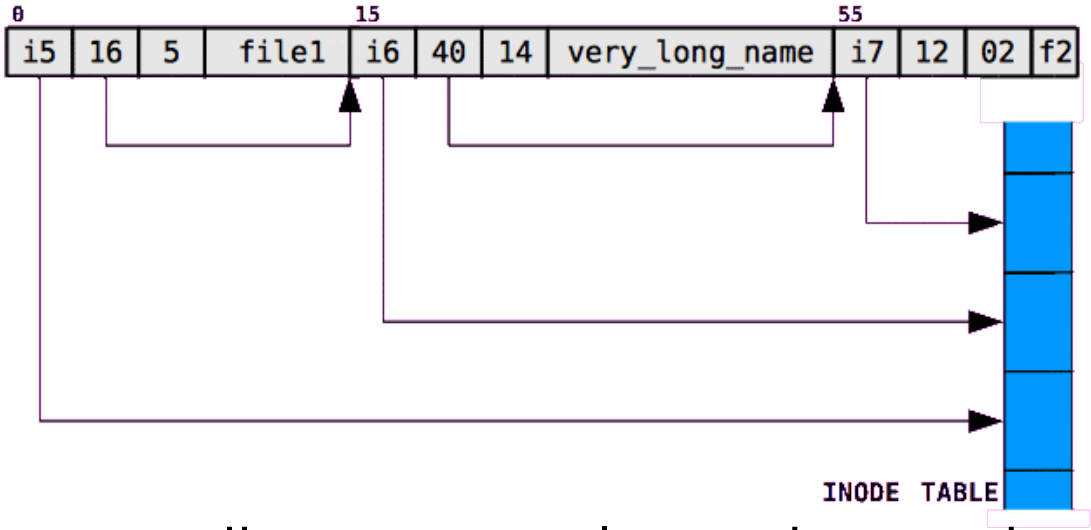
- Basic concept of the Ext2 system (and of all Unix file systems) is the structure called **inode** (index node)
- A file is represented by one inode
- The length of files is variable but all inodes are of the same length (128 Byte)



- Smaller files are more quickly accessed than larger files

Ext2: Directories

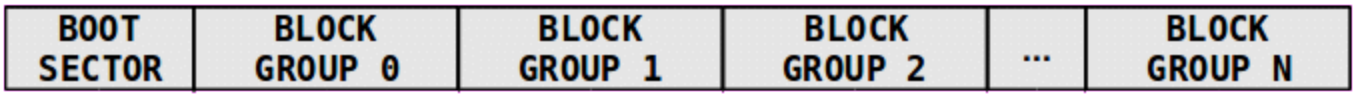
- A directory is a file which is formatted with a special format - a list of directory entries
  - Also a directory has an inode
- Directory entries are of variable length
  - File name of varying length are supported
- An entry consists of
  - Inode number
  - Entry length
  - Name length
  - File name



- The first two entries for every directory are always the standard “.” and “..” (“this directory” and “the parent directory”)
- The inode number of the root directory is stored in the super block so the system can access it directly at any time

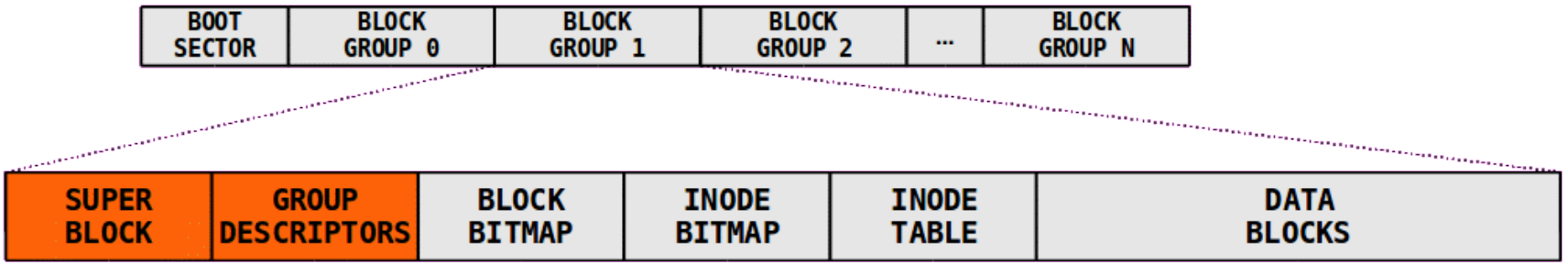
Ext2: Blocks and Block Groups

- A block is the smallest unit that can be allocated on an Ext2 partition
- The blocks are grouped into block groups of the same size



- If possible, data blocks for a file are allocated in the same group as its inode
  - Related data is kept physically close, seek time is reduced
- Each group stores a copy of critical administrative information
  - Data security is increased
- All block groups have the same size and are stored sequentially
  - The location of a block group can be derived from its index

Ext2: Block Groups



Super block

Description of basic size and shape of the file system

- information that allows to use and maintain the system

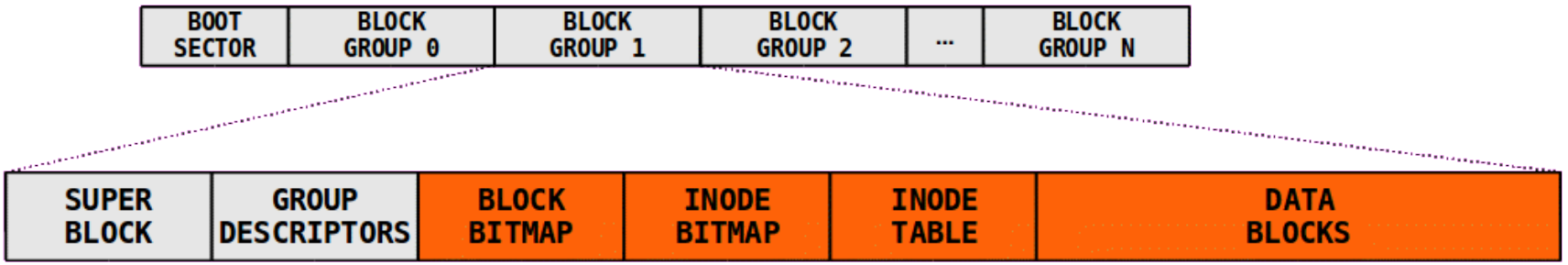
Group descriptors

Position of block bitmap, inode bitmap and inode table, number of free data blocks, inodes and directories

- information used when new data blocks are allocated

- Copies in all block groups
  - Only super block and group descriptors in block group 0 are actually used, the remaining copies are only used in case of file system corruption

Ext2: Block Groups



Block bitmap/ inode bitmap

One bit per block/inode, indicates whether the block/inode is used or free

- to keep track of allocated blocks and inodes

Inode table

A predefined number of inodes

Data blocks

Blocks storing the actual data

Related System Calls

Related System Calls (Linux)

- **`int open(const char *pathname, int flags)`**
  - Open file at **`pathname`** with options **`flags`** and return file descriptor
- **`int close(int fd)`**
  - Close file desciptor **`fd`**

- **`ssize_t read(int fd, void *buf, size_t count)`**
- **`ssize_t write(int fd, const void *buf, size_t count)`**
  - Read/write data at **`buf`** with **`count`** bytes from/to file descriptor **`fd`**
- **`off_t lseek(int fd, off_t offset, int whence)`**
  - Seek, i.e. change current “cursor position”, in file descriptor **`fd`** by **`offset`** bytes in relation to **`whence`** (**`SEEK_SET`**, **`SEEK_CUR`**, **`SEEK_END`**)

- **`int fcntl(int fd, int cmd)`**
- **`int fcntl(int fd, int cmd, long arg)`**
- **`int fcntl(int fd, int cmd, struct flock *lock)`**
  - Performs operation **`cmd`** on file descriptor **`fd`**, e.g. locking to protects against concurrent access, signaling on I/O, ...
    - Is this a well-designed interface?

Content