METU - EE 442 - Operating Systems

# File Systems

Chapter 4

Cüneyt F. Bazlamaçcı (METU-EEE)

The slides include material from Tanenbaum & Bos, Modern Operating Systems, 4th ed., Pearson

# File Systems (1)

Essential requirements for long-term information storage:

1. It must be possible to store a very large amount of information.
2. Information must survive termination of process using it.
3. Multiple processes must be able to access information concurrently.

- Disks are used to store files
- Information is stored in blocks on the disks
- Can read and write blocks

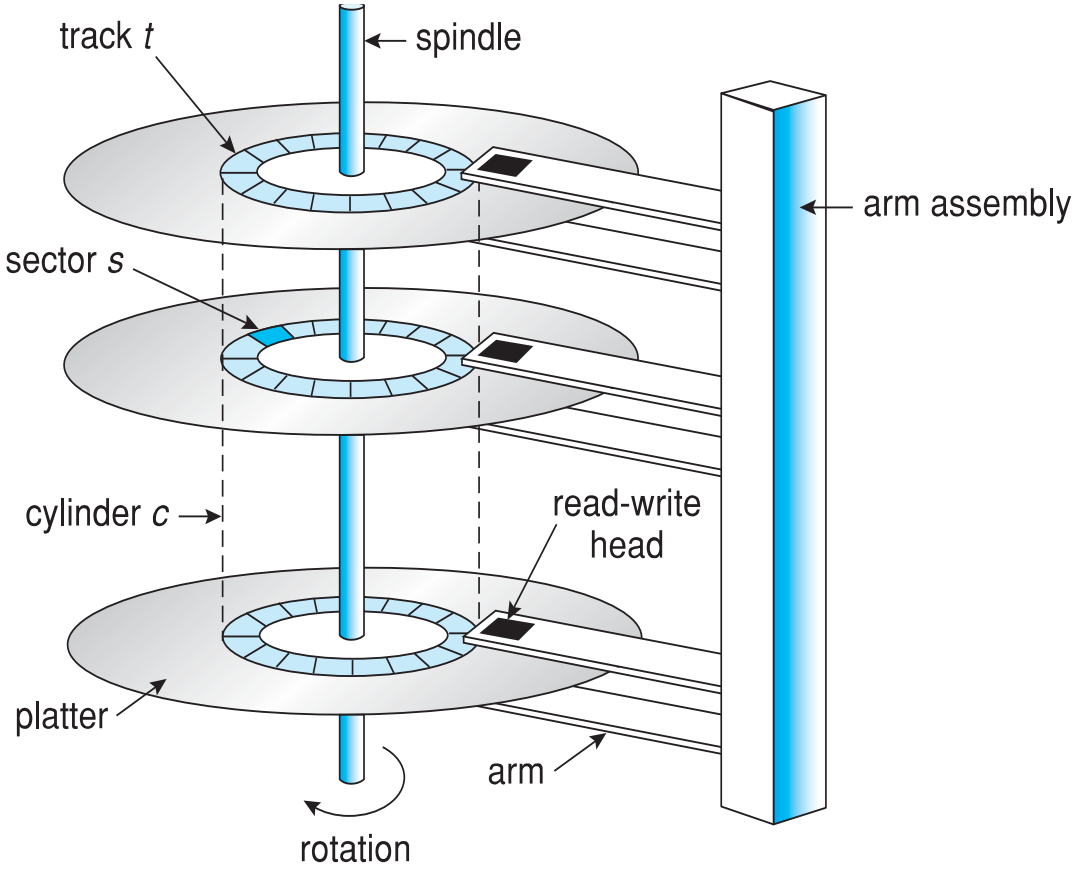
# Overview of Mass Storage Structure

- **Magnetic disks** provide bulk of secondary storage of modern computers
  - Drives rotate at 60 to 250 times per second
  - **Transfer rate** is the rate at which data flow between drive and computer
  - **Positioning time** (**random-access time**) is the time to move disk arm to desired cylinder (**seek time**) and the time for desired sector to rotate under the disk head (**rotational latency**)
  - **Head crash** results from disk head making contact with the disk surface -- bad
- Disks can be removable.

# Overview of Mass Storage Structure

- 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

Moving-head disk mechanism



# Hard 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) = 60 / RPM
  - Average latency = ½ latency

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

(From Wikipedia)

# Hard 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 + 0.1ms + transfer time =
  - Transfer time = 4KB / 1Gb/s * 8b / B * 1G / (1024K * 1024K = 32 / ($1024^2$) = 0.031 ms
  - Average I/O time for 4KB block = 9.27ms + .031ms = 9.301ms

# 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

# Solid-State Disks

- Nonvolatile memory used like a hard drive
  - Many technology variations
- Can be more reliable than HDDs
- More expensive per MB
- May have shorter life span
- Less capacity
- But much faster
- Busses can be too slow -> connect directly to PCI for example
- No moving parts, so no seek time or rotational latency

# 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

# Disk Structure

- Disk drives are addressed as large 1-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer
  - Low-level formatting creates **logical blocks** on physical media

- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially
  - Sector 0 is the first sector of the first track on the outermost cylinder
  - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost
  - Logical to physical address tranlation should be easy
    - Except for bad sectors
    - Non-constant # of sectors per track via constant angular velocity

# File Systems

Think of a disk as a linear sequence of fixed-size blocks and supporting two operations:

1. Read block *k*
2. Write block *k*

Questions that quickly arise:

1. How do you find information?
2. How do you keep one user from reading another user's data?
3. How do you know which blocks are free?

# File Systems

- Use file system as an abstraction to deal with accessing the information kept in blocks on a disk
- Files are created by a process and thousands on a disk
- Managed by the OS which
  - structures them,
  - names them,
  - protects them
- Two ways of looking at file system
  - User-how do we name a file, protect it, organize the files
  - Implementation-how are they organized on a disk
- Start with user, then go to implementer

# File Systems – User Point of View

- Files
  - Naming
  - Structure
  - Types
  - Access
  - Attributes
  - Operations
- Directories
  - Single-level
  - Hierarchical
  - Path names
  - Operations

# File Concept

- Contiguous logical address space
- Types:
  - Data
    - numeric
    - character
    - binary
  - Program
- Contents defined by file's creator
  - Many types
    - Consider text file, source file, executable file

# File Naming

- One to 8 letters plus a 3 letter extension in all previous OS's
- In Unix size of the extension is upto the user
- Fat (16 and 32) were used in first Windows systems
- Latest Window systems use Native File System (NTFS)
- All OS's use suffix as part of name
- Unix does not always enforce a meaning for the suffixes
- DOS does enforce a meaning

# File Naming

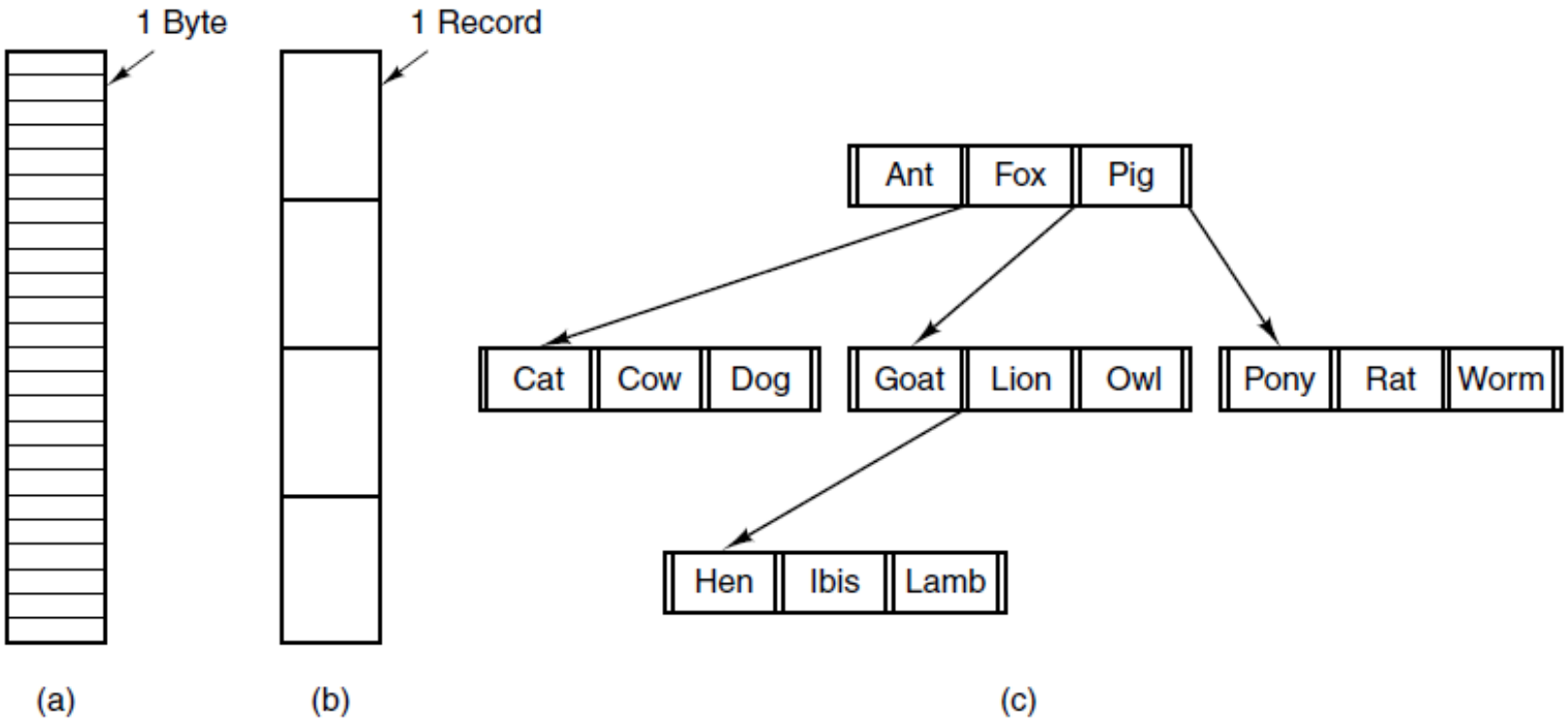
| Extension | Meaning |
|---|---|
| .bak | Backup file |
| .c | C source program |
| .gif | Compuserve Graphical Interchange Format image |
| .hlp | Help file |
| .html | World Wide Web HyperText Markup Language document |
| .jpg | Still picture encoded with the JPEG standard |
| .mp3 | Music encoded in MPEG layer 3 audio format |
| .mpg | Movie encoded with the MPEG standard |
| .o | Object file (compiler output, not yet linked) |
| .pdf | Portable Document Format file |
| .ps | PostScript file |
| .tex | Input for the TEX formatting program |
| .txt | General text file |
| .zip | Compressed archive |

Some typical file extensions.

# File Structure

- Byte sequences
  - Maximum flexibility - can put anything in
  - Unix and Windows use this approach
- Fixed length records (card images in the old days)
- Tree of records- uses key field to find records in the tree

# File Structure




Three kinds of files.

(a) Byte sequence (b) Record sequence (c) Tree

# File Types

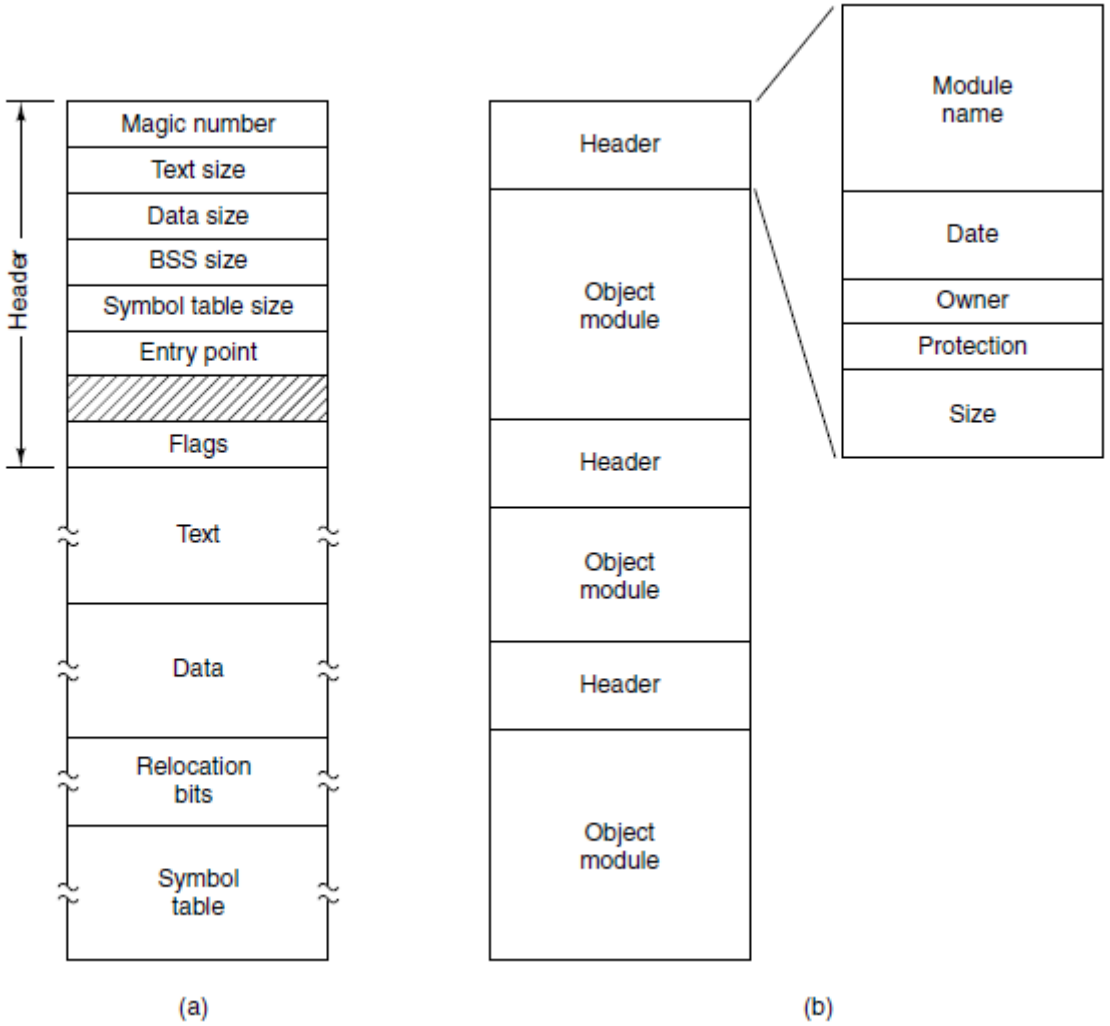
- Some UNIX files
  - Regular- contains user information
  - Directories
  - Character special files - model serial (e.g. printers) I/O devices
  - Block special files -model disks

- Regular files
  - ASCII or binary
  - ASCII (consists of lines of text)
    - Printable
    - Can use pipes to connect programs if they produce/consume ASCII

# Binary File Types

- Two Unix examples
  - Executable (magic field identifies file as being executable)
  - Archive - compiled, not linked library procedures
- Every OS must recognize its own executable

# Binary File Types (Early version of Unix)




(a) An executable file (magic # identifies it as an executable file)

(b) An archive - library procedures compiled but not linked

# File Access

- Sequential access - read from the beginning, can’t skip around
  - Corresponds to magnetic tape
- Random access - start where you want to start
  - Came into play with disks
  - Necessary for many applications, e.g. airline reservation system

# File Attributes (Metadata)

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk
- Many variations, including extended file attributes such as file checksum
- Information kept in the directory structure

# File Attributes

| Attribute | Meaning |
|---|---|
| Protection | Who can access the file and in what way |
| Password | Password needed to access the file |
| Creator | ID of the person who created the file |
| Owner | Current owner |
| Read-only flag | 0 for read/write; 1 for read only |
| Hidden flag | 0 for normal; 1 for do not display in listings |
| System flag | 0 for normal files; 1 for system file |
| Archive flag | 0 for has been backed up; 1 for needs to be backed up |
| ASCII/binary flag | 0 for ASCII file; 1 for binary file |
| Random access flag | 0 for sequential access only; 1 for random access |
| Temporary flag | 0 for normal; 1 for delete file on process exit |
| Lock flags | 0 for unlocked; nonzero for locked |
| Record length | Number of bytes in a record |
| Key position | Offset of the key within each record |
| Key length | Number of bytes in the key field |
| Creation time | Date and time the file was created |
| Time of last access | Date and time the file was last accessed |
| Time of last change | Date and time the file was last changed |
| Current size | Number of bytes in the file |
| Maximum size | Number of bytes the file may grow to |

Some possible file attributes.

# File Operations

1. Create
2. Delete
3. Open
4. Close
5. Read
6. Write
7. Append
8. Seek
9. Get attributes
10. Set attributes
11. Rename

# System Calls for Files

- Create - with no data, sets some attributes
- Delete - to free disk space
- Open - after create, gets attributes and disk addresses into main memory
- Close - frees table space used by attributes and addresses
- Read - usually from current pointer  position. Need to specify buffer into which data is placed
- Write - usually to current position

# System Calls for Files

- Append - at the end of the file
- Seek - puts file pointer at specific place in file. Read or write from that position on
- Get Attributes - e.g. make needs most recent modification times to arrange for group compilation
- Set Attributes - e.g. protection attributes
- Rename

# How can system calls be used? An example - *copyfile abc xyz*

Copies file abc to xyz

- If xyz exists it is over-written
- If it does not exist, it is created
- Uses system calls (read, write)
- Reads and writes in 4K chunks
- Read (system call) into a buffer
- Write (system call) from buffer to output file

# Example Program Using File System Calls (1)

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                  /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* ANSI prototype */

#define BUF_SIZE 4096                   /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                /* protection bits for output file */

int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit(1);             /* syntax error if argc is not 3 */

    /* Open the input file and create the output file */
```

A simple program to copy a file.

# Example Program Using File System Calls (2)

```
if (argc != 3) exit(1);                          /* syntax error if argc is not 3 */

/* Open the input file and create the output file */
in_fd = open(argv[1], O_RDONLY);                 /* open the source file */
if (in_fd < 0) exit(2);                          /* if it cannot be opened, exit */
out_fd = creat(argv[2], OUTPUT_MODE);            /* create the destination file */
if (out_fd < 0) exit(3);                         /* if it cannot be created, exit */

/* Copy loop */
while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
if (rd_count <= 0) break;                        /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
```

A simple program to copy a file.

# Example Program Using File System Calls (3)

```
    /* Copy loop */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
    if (rd_count <= 0) break;                     /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4);               /* wt_count <= 0 is an error */
    }

    /* Close the files */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0)                            /* no error on last read */
        exit(0);
    else
        exit(5);                                  /* error on last read */
}
```

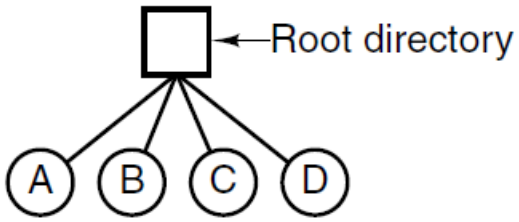
A simple program to copy a file.

# Directories

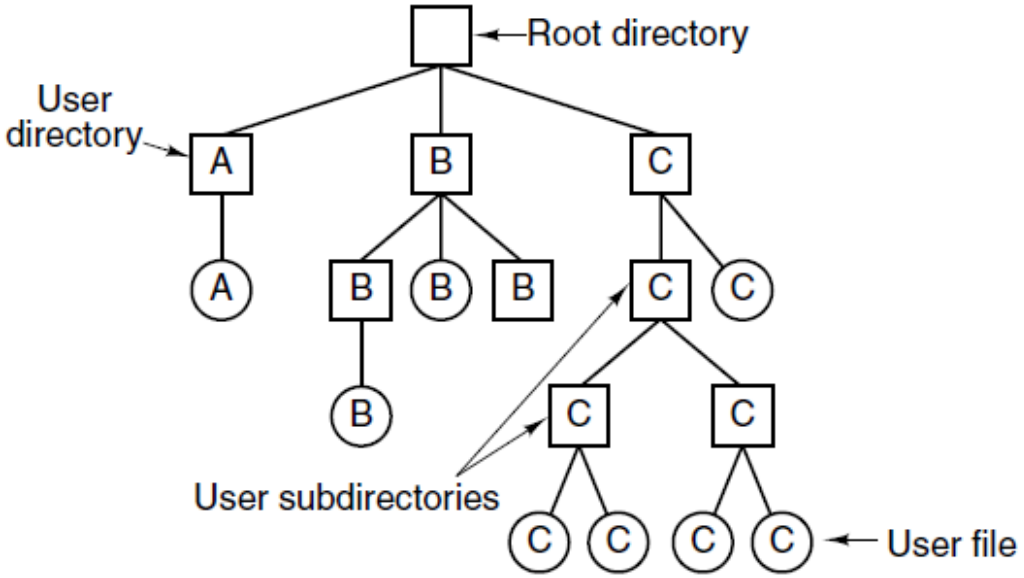
- Files, which are used to organize a collection of files
- Also called folders in some OS's
- Two types
  - Single-Level Directory Systems
  - Hierarchical Directory Systems

.

# Directory Systems

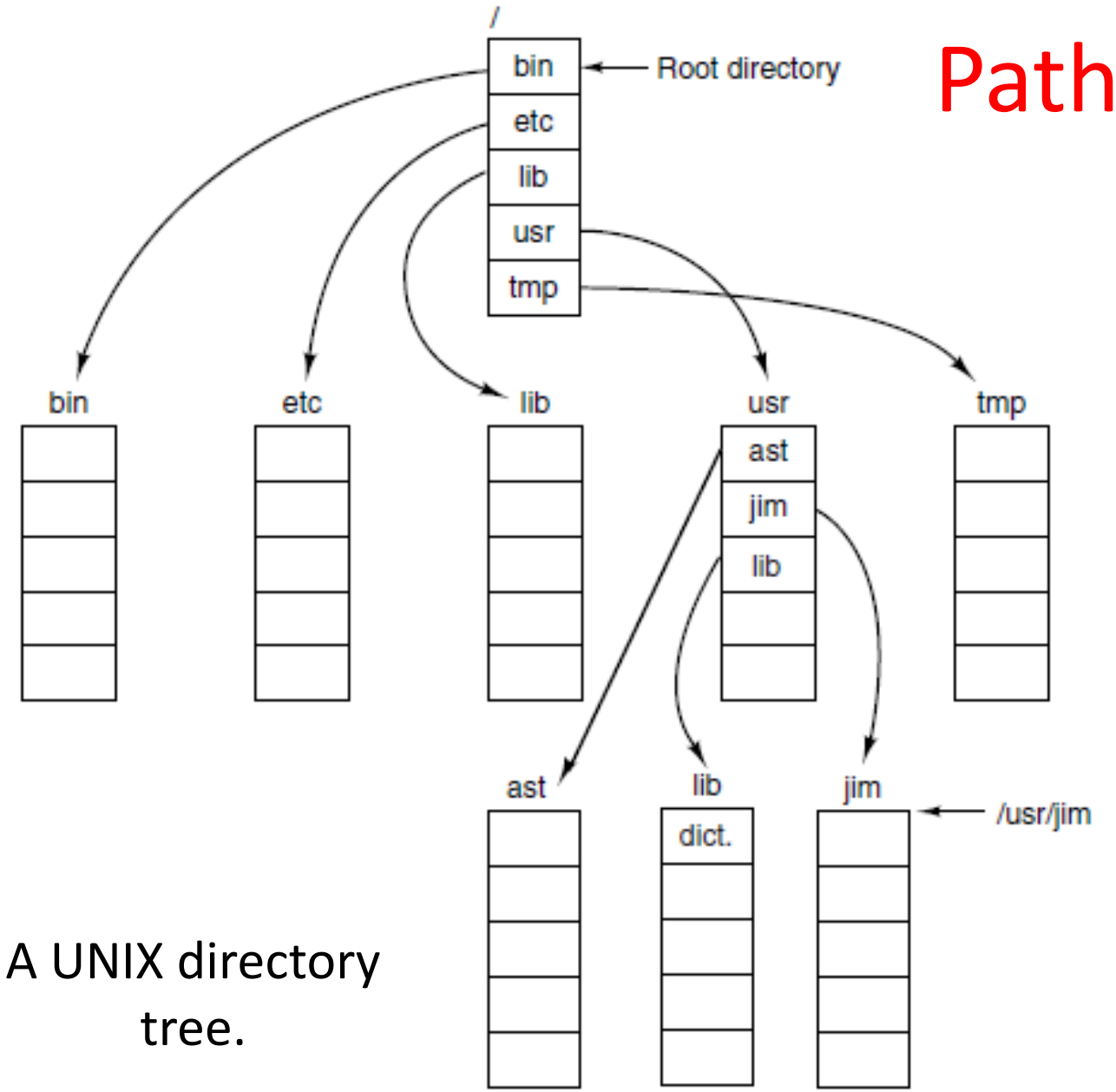



A single-level directory system containing four files.




A hierarchical directory system.

# Path Names




A UNIX directory tree.

# Path names

- Absolute

  /usr/cb/ee442/midterm/answers
- Relative

  ee442/midterm/answers
- . refers to current (working) directory
- .. refers to parent of current directory

# Unix cp commands involving dots

cp ../lib/dictionary .

- .. says go to parent (usr)
- . says that target of the copy is current directory
- cp /usr/lib/dictionary dictionary works
- cp /usr/lib/dictionary /usr/ast/dictionary also works

# Directory Operations

1. Create
2. Delete
3. Opendir
4. Closedir
5. Readdir
6. Rename
7. Link
8. Unlink

# Directory Operations

- Create - creates directory
- Delete - directory has to be empty to be deleted
- Opendir - should be done before any operations on directory
- Closedir
- Readdir - returns next entry in open directory
- Rename
- Link - links file to another directory
- Unlink - gets rid of directory entry

# File Implementation

- Users are concerned with
  - how files are named,
  - what operations are allowed on them,
  - what the directory tree looks like,
  - similar interface issues, etc.

- Implementors are interested in
  - how files and directories are stored,
  - how disk space is managed,
  - how to make everything work efficiently and reliably.

- Various trade-offs need be considered in implementation.

# File Implementation

- Files stored on disks.
  - Disks broken up into one or more partitions, with separate fs on each partition
- Sector 0 of disk is the Master Boot Record (MBR)
  - Used to boot the computer
  - End of MBR has partition table.
  - Has starting and ending addresses of each partition.
- One of the partitions is marked active in the master boot table.

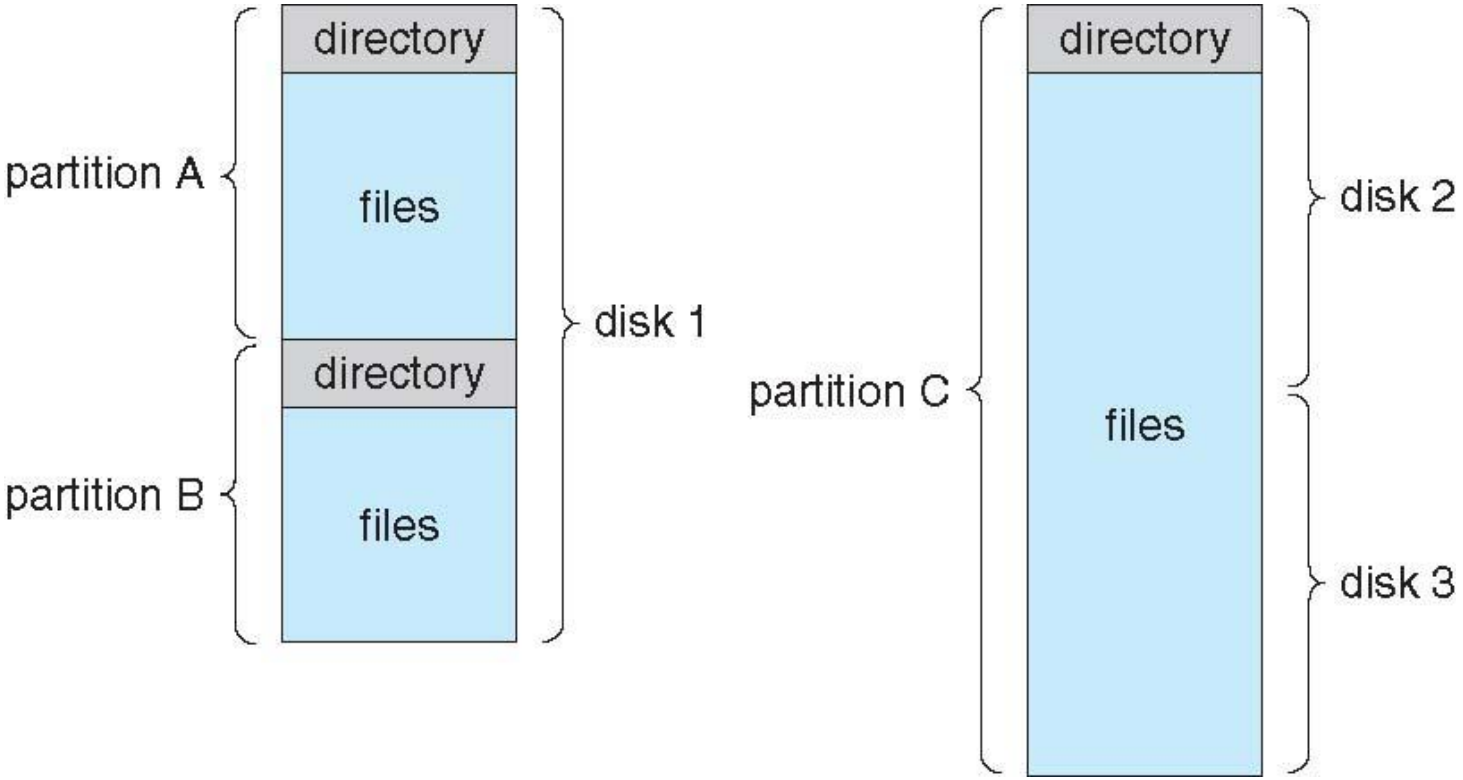
# File Implementation

- Boot computer => BIOS reads/executes MBR
- MBR finds active partition and reads in first block (boot block)
- Program in boot block locates the OS for that partition and reads it in
- All partitions start with a boot block

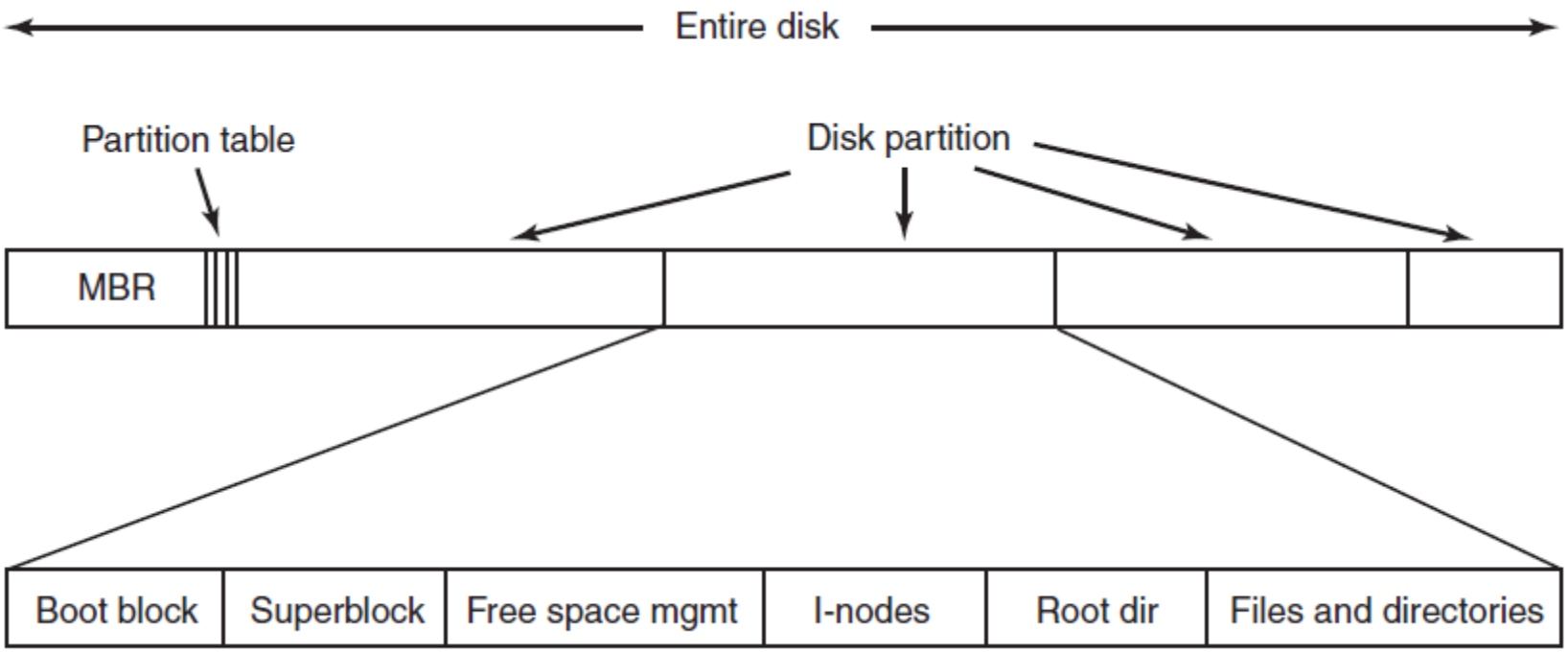
# Disk Structure

- 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



# File System Layout




A possible file system layout.

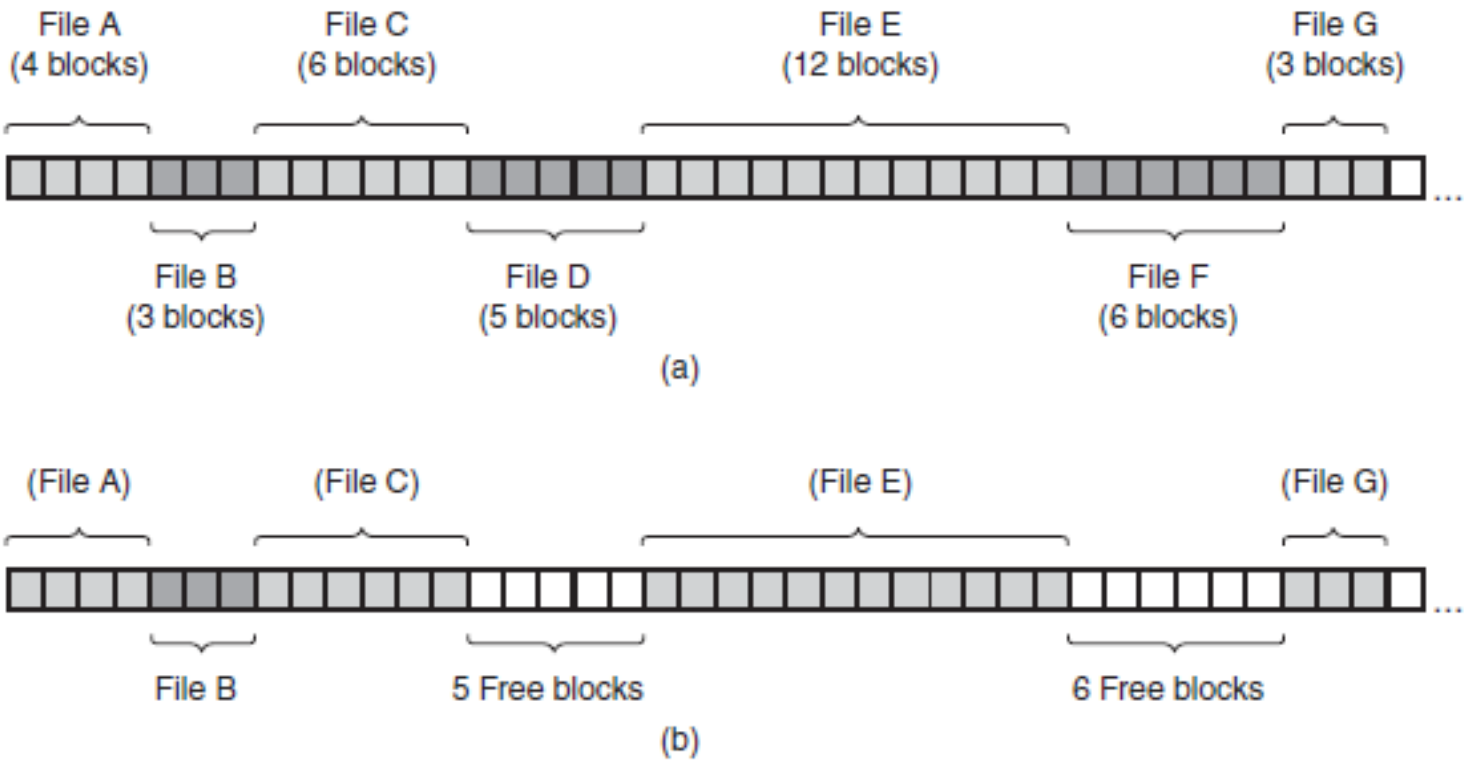
# File System Layout

- Superblock contains info about the fs (e.g. type of fs, number of blocks, …)
- I-nodes contain info about files

# Allocating Blocks to files

- Most important implementation issue
- Methods
  - Contiguous allocation
  - Linked list allocation
  - Linked list using table
  - I-nodes

# Implementing Files - Contiguous Layout




(a) Contiguous allocation of disk space for seven files.

(b) The state of the disk after files *D* and *F* have been removed.

# Contiguous Allocation

## The good

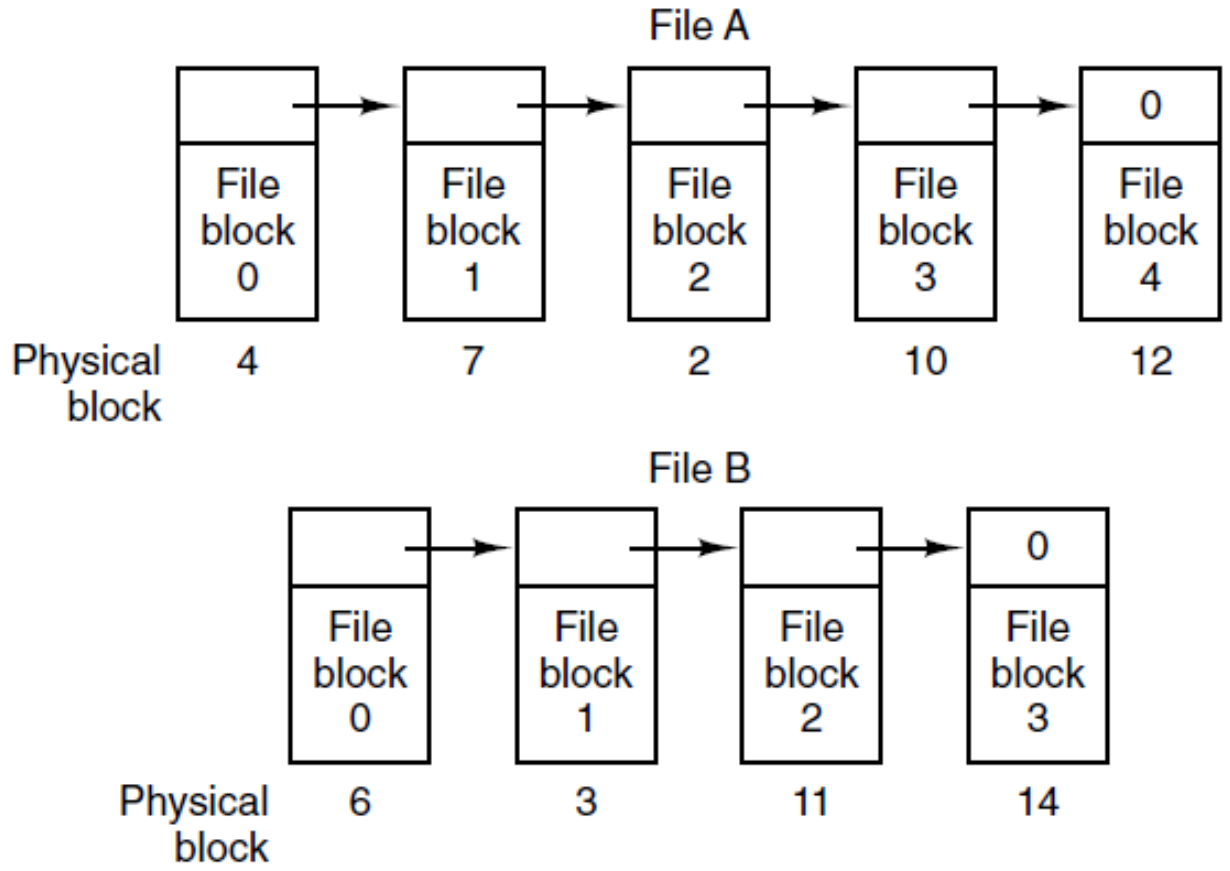
- Easy to implement
- Read performance is great. Only need one seek to locate the first block in the file. The rest is easy.

## The bad

- disk becomes fragmented over time

- CD-ROM's use contiguous allocation because the fs size is known in advance
- DVD's are stored in a few consecutive 1 GB files because standard for DVD only allows a 1 GB file max

# Implementing Files - Linked List Allocation




Storing a file as a linked list of disk blocks.

# Linked Lists

## The good

- Gets rid of fragmentation

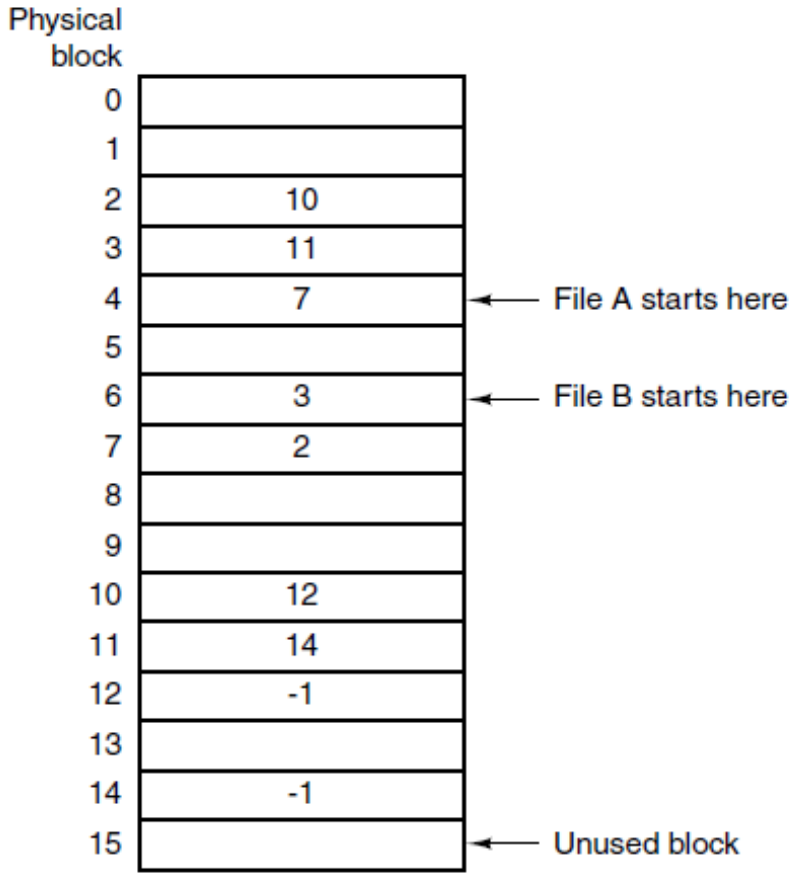
## The bad

- Random access is slow. Need to chase pointers to get to a block

# Linked lists using a table in memory

- Put pointers in a table in memory
- File Allocation Table (FAT)
- Used in Windows

# Implementing Files
# Linked List – Table in Memory




Linked list allocation using a file allocation table in main memory.

# Linked lists using a table in memory

## The bad

- table must be in memory all the time
- table becomes really big
- e.g.  1 TB disk with 1 KB blocks needs 1 billion entries, 1 entry for each of the 1 billion (1G) disk blocks. Each entry of 4 bytes implies that the table occupies 4 GB of main memory.
- growth of the table size is linear with the growth of the disk size

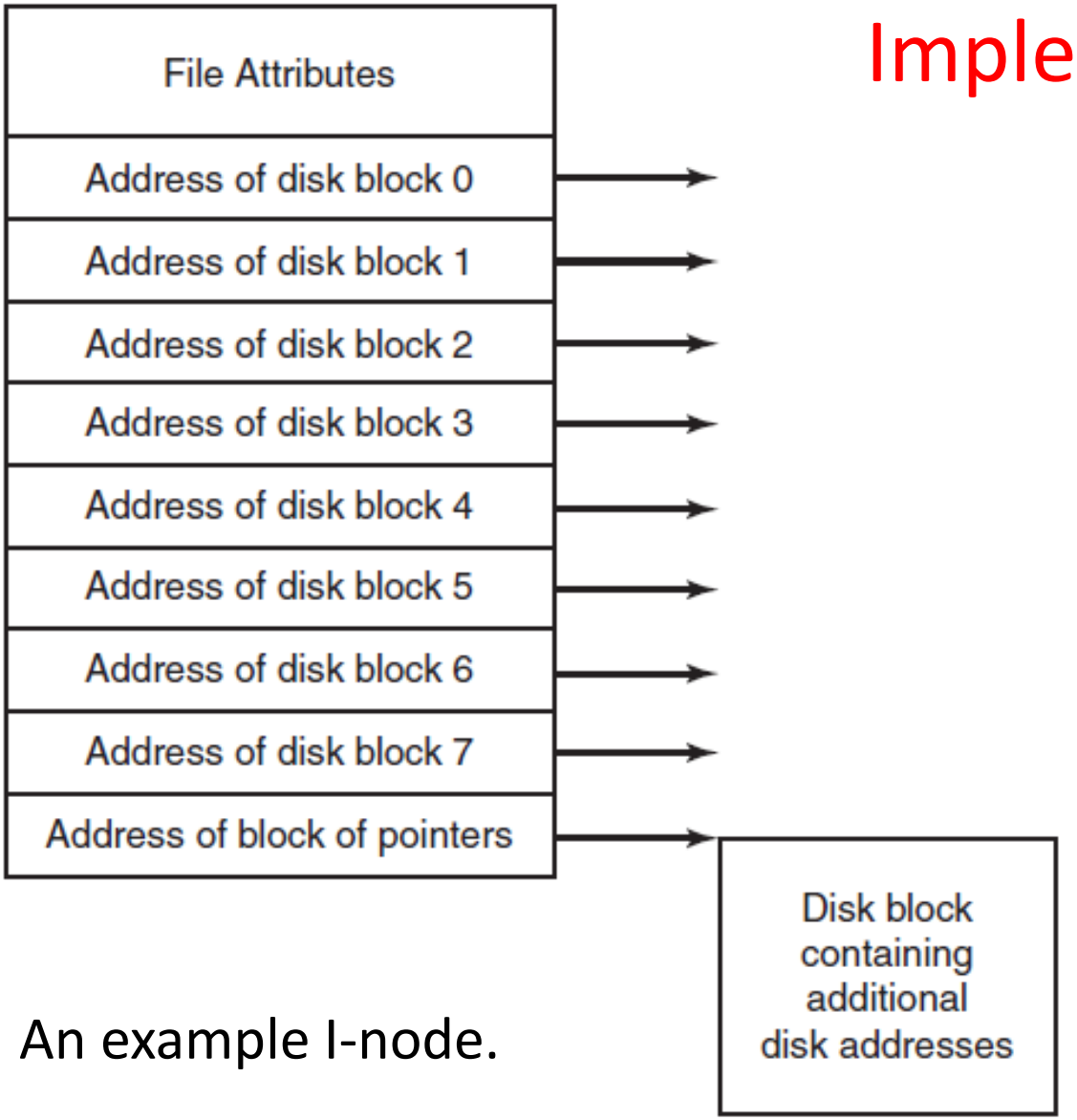
- 1 KB = 2^10
- 1 MB = 2^20
- 1 GB = 2^30
- 1 TB = 2^40

# I-nodes

- Keep data structure in memory only for active files
  - Data structure lists disk addresses of the blocks and attributes of the files
- *k* active files, *n* blocks per file => *k***n* blocks max!!
- Solves the growth problem

- How big is *n*?
- Solution: Last entry in table points to disk block, which contains pointers to other disk blocks

# Implementing Files - I-nodes

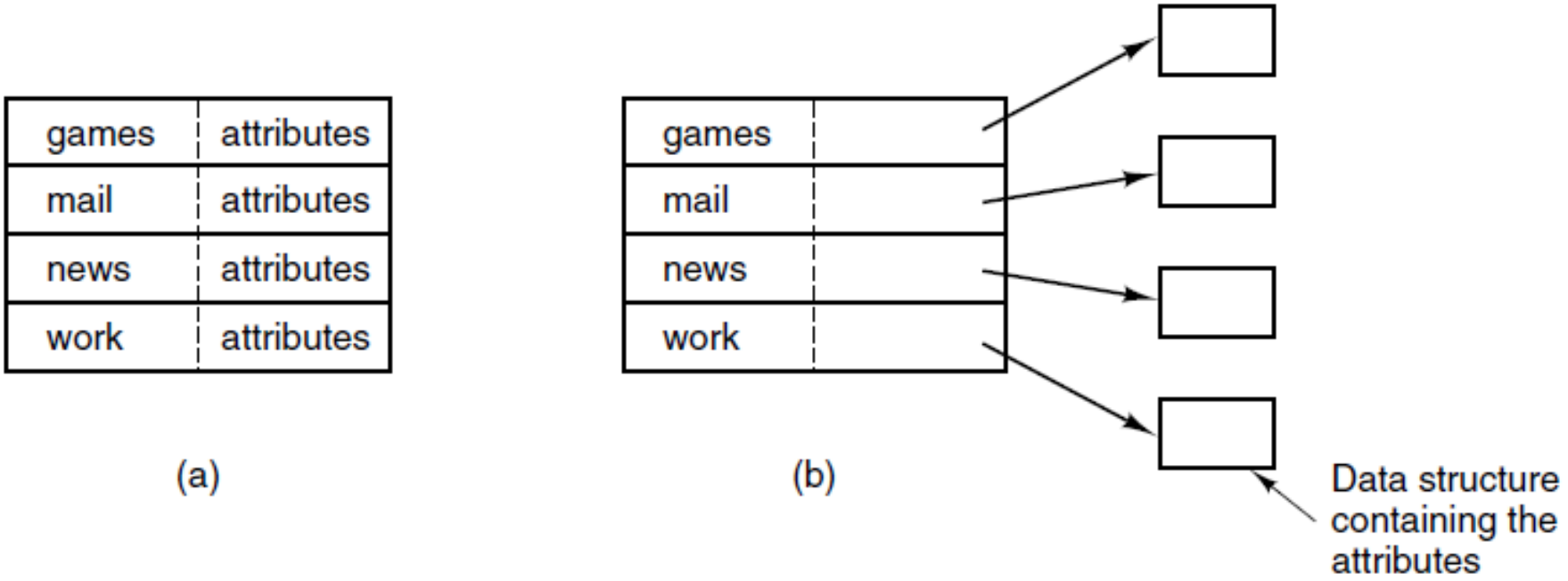



An example I-node.

# Implementing Directories

- Before a file can be read, it must be opened.
- Path name used to locate directory.
- Directory specifies block addresses by providing, either of
  - Disk address of the file (contiguous)
  - Number of first block (linked list)
  - Number of the i-node

# Implementing Directories (1)



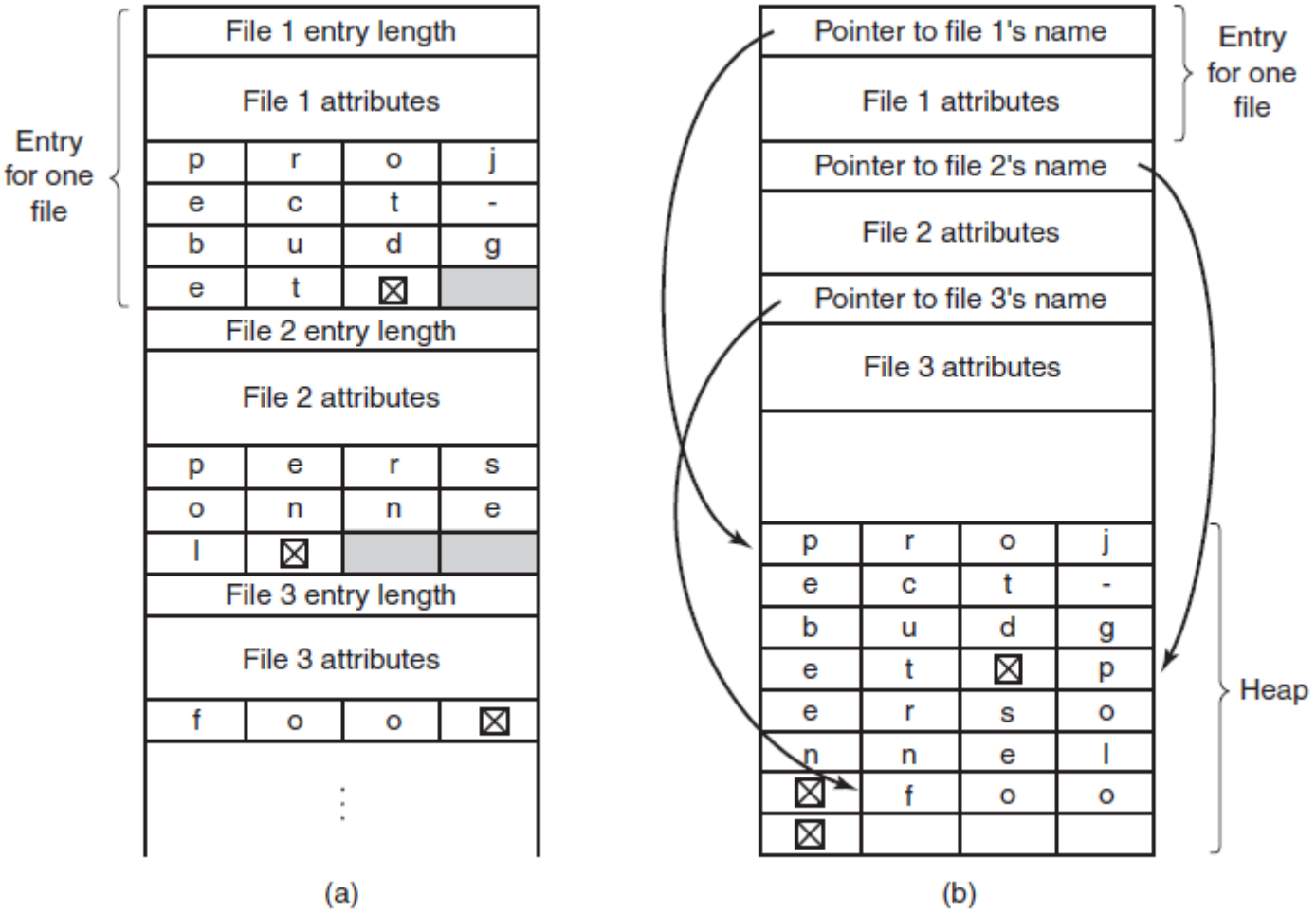

(a) A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry.

(b) A directory in which each entry just refers to an i-node.

# Implementing Directories

- How do we deal with variable length names?
- Problem is that names have gotten very long
- You may still use fixed headers (inefficient since few files have long names)
- Two other possible approaches
  - Fixed header followed by variable length names
  - Heap-pointer points to names

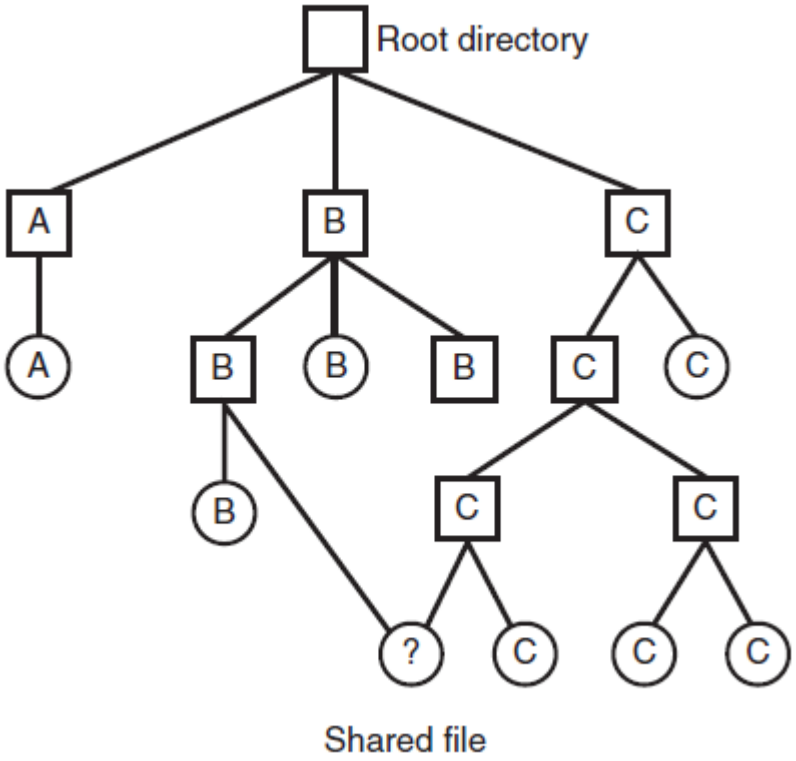
# Implementing Directories (2)




Two ways of handling long file names in a directory.

(a) In-line. (b) In a heap.

# Shared Files (1)




The file system itself is now a **Directed Acyclic Graph**, or **DAG**, rather than a tree.

Having the file system be a DAG complicates maintenance.

File system containing a shared file.

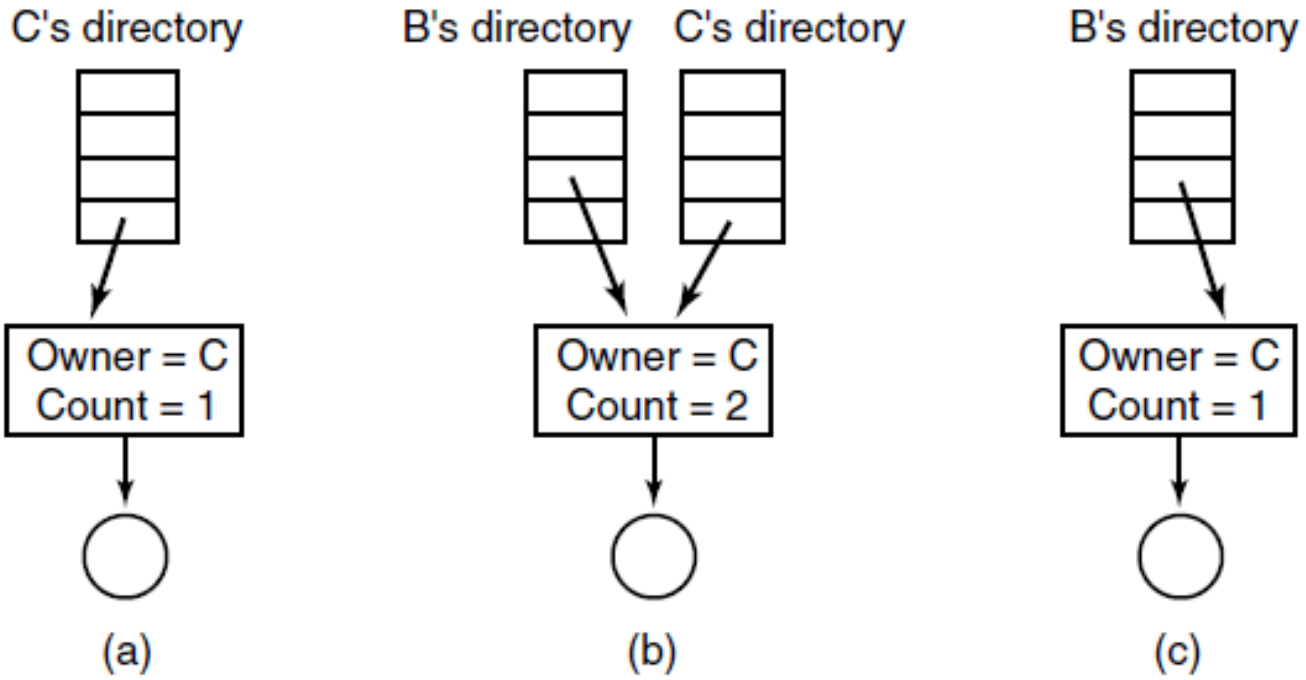
# Shared files

- If B or C adds new blocks, how does other owner find out?
  - Use special i-node for shared files,
    - which indicates that file is shared
  - Use symbolic link
    - a special file put in B's directory if C is the owner. Contains the path name of the file to which it is linked

# I-node problem

- If C removes file, B's directory still points to i-node for shared file
- If i-node is re-used for another file, B's entry points to wrong i-node
- Solution is to leave i-node and reduce number of owners

# Shared Files (2)




(a) Situation prior to linking.

(b) After the link is created.

(c) After the original owner removes the file.

# Symbolic links

- Symbolic link solves problem:

- When the owner removes the file, it is destroyed.
- Subsequent attempts to use the file via a symbolic link will fail when the system is unable to locate the file.

- Can have too many symbolic links and they take time to follow

- Big advantage - can point to files on other machines

# Log Structured File System (LSF)

CPU's are faster, disks and memories are bigger (much) but disk seek time has not decreased

- Caches bigger - can do reads from cache
- Want to optimize writes because disk needs to be updated
- In most file systems, writes are done in very small chunks.
- Small writes are highly inefficient, since a 50-μ sec disk write is often preceded by a 10-msec seek and a 4-msec rotational delay.
- With these parameters, disk efficiency drops to a fraction of 1%.

# Log Structured File System (LSF)

- Structure entire disk as a big log - collect writes in a memory buffer and periodically send them to a segment in the disk. Writes tend to be very small.
- Segment has summary of contents (i-nodes, directories, blocks, etc.).
- i-nodes are now scattered all around the disk (needs to be located)
- Keep i-node map on disk and cache it in memory to locate i-nodes

# Log Structured File System

- ***Cleaner*** thread compacts log. Scans segment for current i-nodes, discarding ones not in use and sending current ones to memory.
- ***Writer*** thread writes current ones out into new segment.
- Works well in Unix. Not compatible with most file systems
- Not a heavily used approach.

# Journaling File Systems

Want to guard against lost files when there are crashes.

Consider what happens when a file has to be removed (in Unix).

- Remove the file from its directory.
- Release the i-node to the pool of free i-nodes.
- Return all the disk blocks to the pool of free disk blocks

- If there is a crash somewhere in this process, you have a mess.

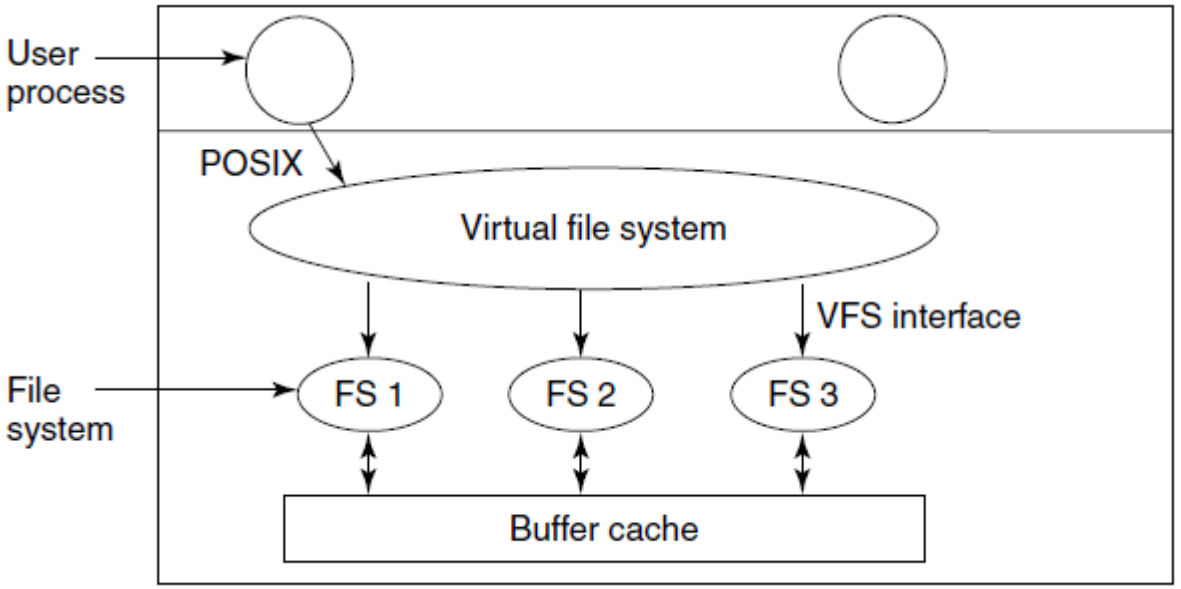
# Journaling File Systems

- Keep a journal (i,.e. list) of actions before you take them, write journal to disk, then perform actions. Can recover from a crash!
- Need to make operations idempotent (which means they can be repeated as often as necessary without harm). Must arrange data structures to do so.
  - Mark block *n* as free is an idempotent operation.
  - Adding freed blocks to the end of a list is not idempotent
  - NTFS (Windows) and Linux use journaling

# Virtual File Systems

- Have multiple FSs on the same machine
- Windows specifies FS (drives)
- Unix integrates into VFS
  - VFS calls from user
  - Lower level calls to actual FS
- Supports Network File System - file can be on a remote machine

# Virtual File Systems (1)

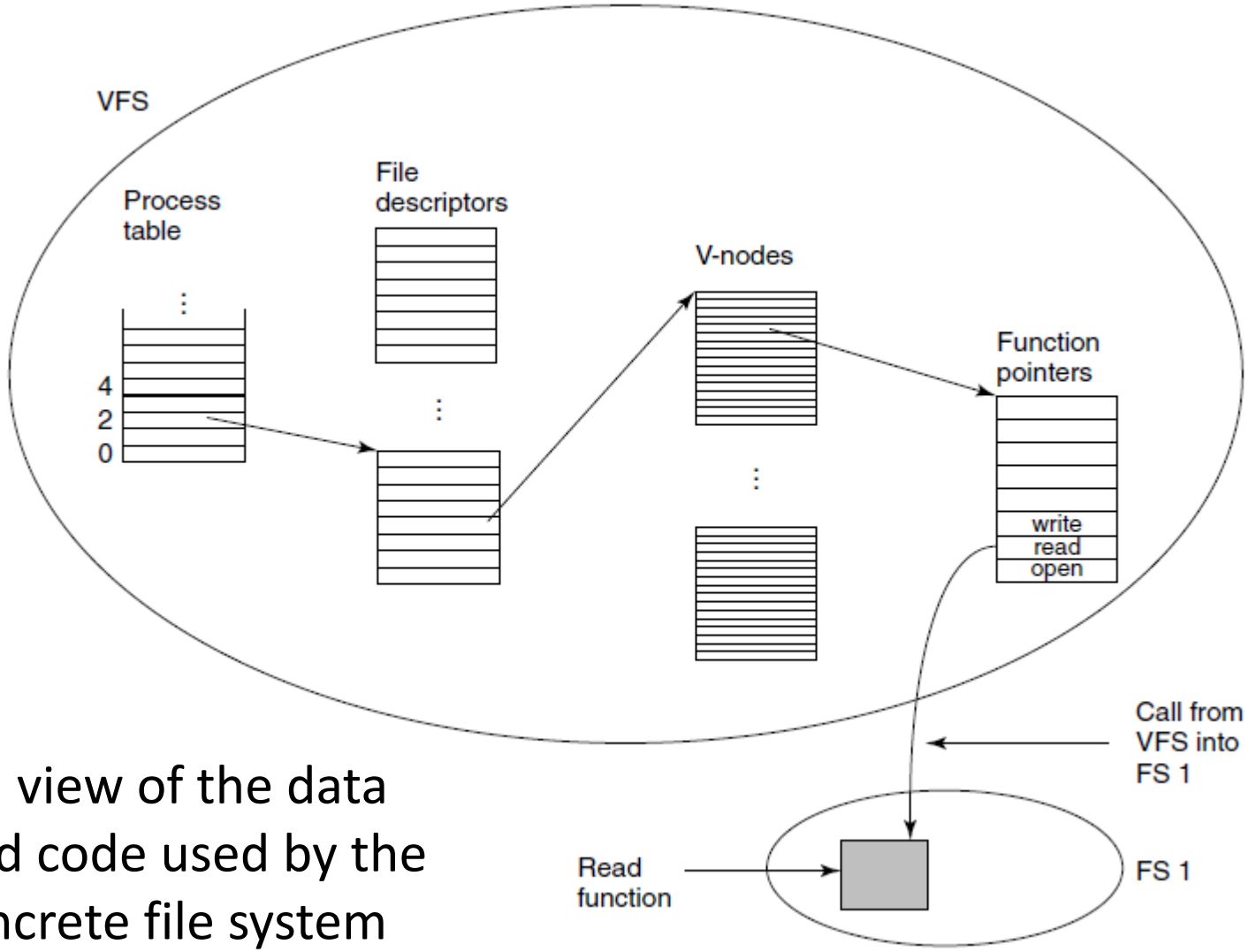



Position of the virtual file system.

# VFS-how it works

- File system registers with VFS (e.g. at boot time)
- At registration time, FS provides list of addresses of function calls the VFS wants
- VFS gets info from the new FS i-node and puts it in a v-node
- Makes entry in fd table for process
- When process issues a call (e.g. read), function pointers point to concrete function calls

# Virtual File Systems (2)




A simplified view of the data structures and code used by the VFS and concrete file system to do a *read*.

# File System Management and Optimization

- Disk Space Management
  - Block size
  - Keeping track of free blocks
  - Disk quotas
- File-System Backups
- File-System Consistency
- File-System Performance
  - Caching
  - Block read ahead
  - Reducing disk-arm motion
- Defragmenting Disks

# Disk Space Management

- Disk space management - use fixed size blocks which don’t have to be adjacent
  - If stored as consecutive bytes and file grows it will have to be moved
- What is optimum (good) block size? Need information on file size distribution.
  - Don’t have it when building fs.

# Disk Space Management (1)

| Length | VU 1984 | VU 2005 | Web |
|---|---|---|---|
| 1 | 1.79 | 1.38 | 6.67 |
| 2 | 1.88 | 1.53 | 7.67 |
| 4 | 2.01 | 1.65 | 8.33 |
| 8 | 2.31 | 1.80 | 11.30 |
| 16 | 3.32 | 2.15 | 11.46 |
| 32 | 5.13 | 3.15 | 12.33 |
| 64 | 8.71 | 4.98 | 26.10 |
| 128 | 14.73 | 8.03 | 28.49 |
| 256 | 23.09 | 13.29 | 32.10 |
| 512 | 34.44 | 20.62 | 39.94 |
| 1 KB | 48.05 | 30.91 | 47.82 |
| 2 KB | 60.87 | 46.09 | 59.44 |
| 4 KB | 75.31 | 59.13 | 70.64 |
| 8 KB | 84.97 | 69.96 | 79.69 |

| Length | VU 1984 | VU 2005 | Web |
|---|---|---|---|
| 16 KB | 92.53 | 78.92 | 86.79 |
| 32 KB | 97.21 | 85.87 | 91.65 |
| 64 KB | 99.18 | 90.84 | 94.80 |
| 128 KB | 99.84 | 93.73 | 96.93 |
| 256 KB | 99.96 | 96.12 | 98.48 |
| 512 KB | 100.00 | 97.73 | 98.99 |
| 1 MB | 100.00 | 98.87 | 99.62 |
| 2 MB | 100.00 | 99.44 | 99.80 |
| 4 MB | 100.00 | 99.71 | 99.87 |
| 8 MB | 100.00 | 99.86 | 99.94 |
| 16 MB | 100.00 | 99.94 | 99.97 |
| 32 MB | 100.00 | 99.97 | 99.99 |
| 64 MB | 100.00 | 99.99 | 99.99 |
| 128 MB | 100.00 | 99.99 | 100.00 |

Percentage of files smaller than a given size (in bytes).

# Disk Space Management

- For example, in 2005, 59.13% of all files at the VU were 4 KB or smaller and 90.84% of all files were 64 KB or smaller.
- The median file size was 2475 bytes (maybe surprising).

- Some conclusions from these data?
- with a block size of 1 KB, only about 30–50% of all files fit in a single block, whereas with a 4-KB block, the percentage of files that fit in one block goes up to the 60–70% range.
- Other data in the paper show that with a 4-KB block, 93% of the disk blocks are used by the 10% largest files.

- This means that wasting some space at the end of each small file hardly matters because the disk is filled up by a small number of large files (videos) and the total amount of space taken up by the small files hardly matters at all.
- Even doubling the space the smallest 90% of the files take up would be barely noticeable.

# Disk Space Management

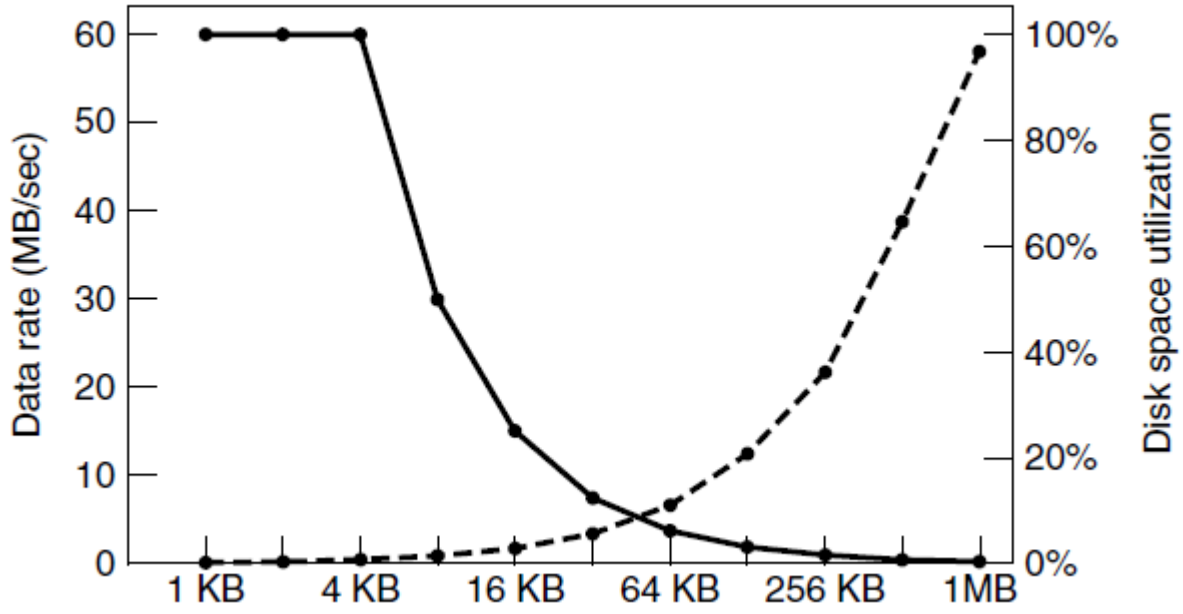
- For example, in 2005, 59.13% of all files at the VU were 4 KB or smaller and 90.84% of all files were 64 KB or smaller.
- The median file size was 2475 bytes (maybe surprising).
- Some conclusions from these data?
- With a block size of 1 KB, only about 30–50% of all files fit in a single block, whereas with a 4-KB block, the percentage of files that fit in one block goes up to the 60–70% range.
- Other data in the paper show that with a 4-KB block, 93% of the disk blocks are used by the 10% largest files.
- This means that wasting some space at the end of each small file hardly matters because the disk is filled up by a small number of large files (videos) and the total amount of space taken up by the small files hardly matters at all.
- Even doubling the space the smallest 90% of the files take up would be barely noticeable.

# Disk Space Management

- On the other hand, using a small block means that each file will consist of many blocks.
- Reading each block normally requires a seek and a rotational delay (except on a solid-state disk), so reading a file consisting of many small blocks will be slow.
- Consider a disk with 1 MB per track, a rotation time of 8.33 msec, and an average seek time of 5 msec.
- The time in milliseconds to read a block of k bytes is then the sum of the seek, rotational delay, and transfer times:

$$5 + 4.165 + (k/1000000) \times 8.33$$

# Disk Space Management (2)

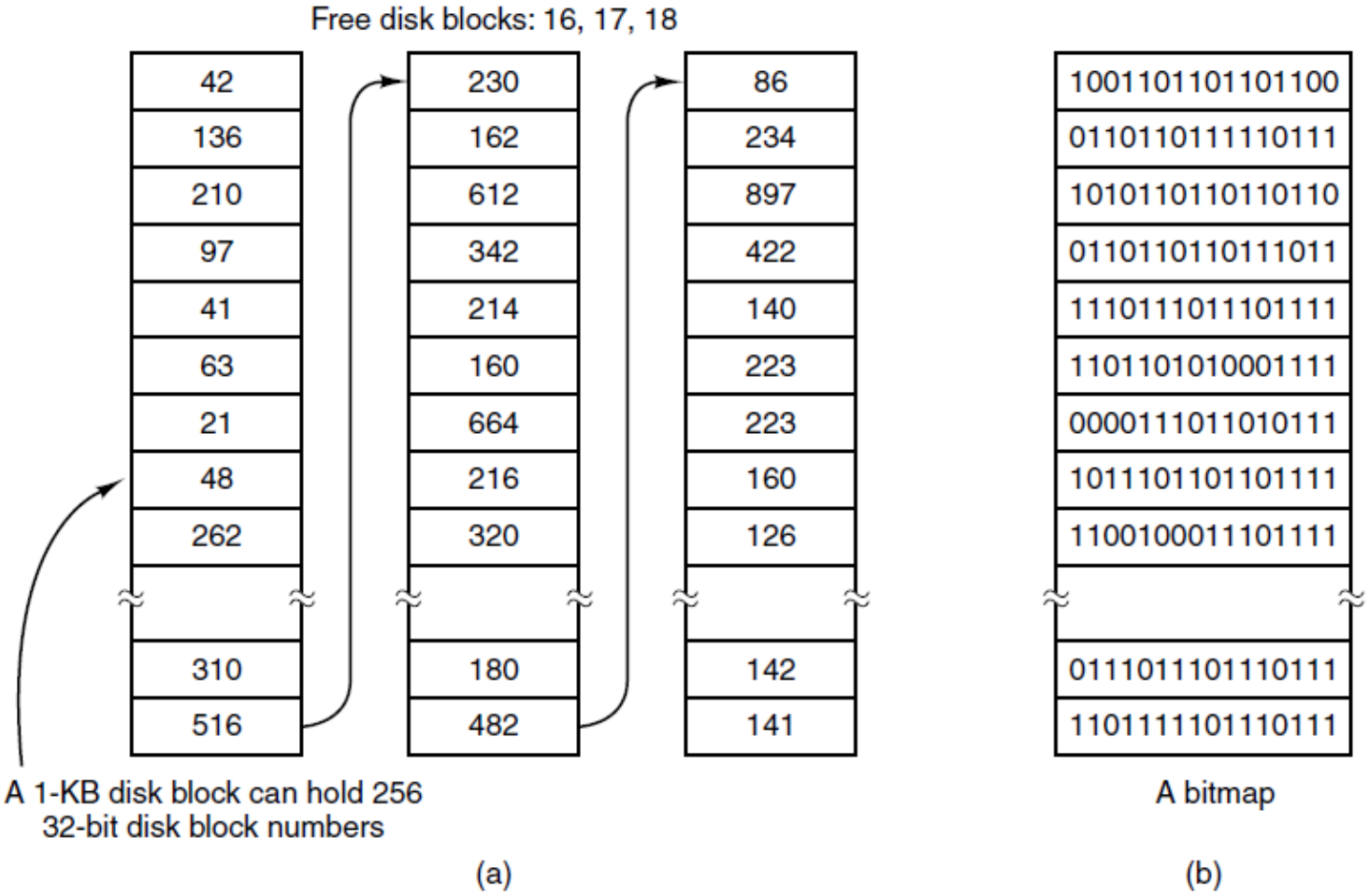



The dashed curve (left-hand scale) gives the data rate of a disk. The solid curve (right-hand scale) gives the disk space efficiency. All files are 4 KB.

# Disk Space Management

- Bigger block size results in better transfer utilization but worse space utilization,
- Trade-off  is space vs data rate
- There is no good answer
- Disk space is hardly in short supply any more.
- Historically, file systems have chosen sizes in the 1-KB to 4-KB range, but with disks now exceeding 1 TB, we might as well use large (64 KB) block size as disks are getting much bigger
- Hence stop thinking about it and accept the wasted disk space.

# Keeping Track of Free Blocks (1)




(a) Storing the free list on a linked list. (b) A bitmap.

# How to keep track of free blocks

- With a 1-KB block and a 32-bit disk block number, each block on the free list holds the numbers of 255 free blocks. (One slot is required for the pointer to the next block.)
- Consider a 1-TB disk, which has about 1 billion disk blocks.
- To store all these addresses at 255 per block requires about 4 million blocks.

# How to keep track of free blocks

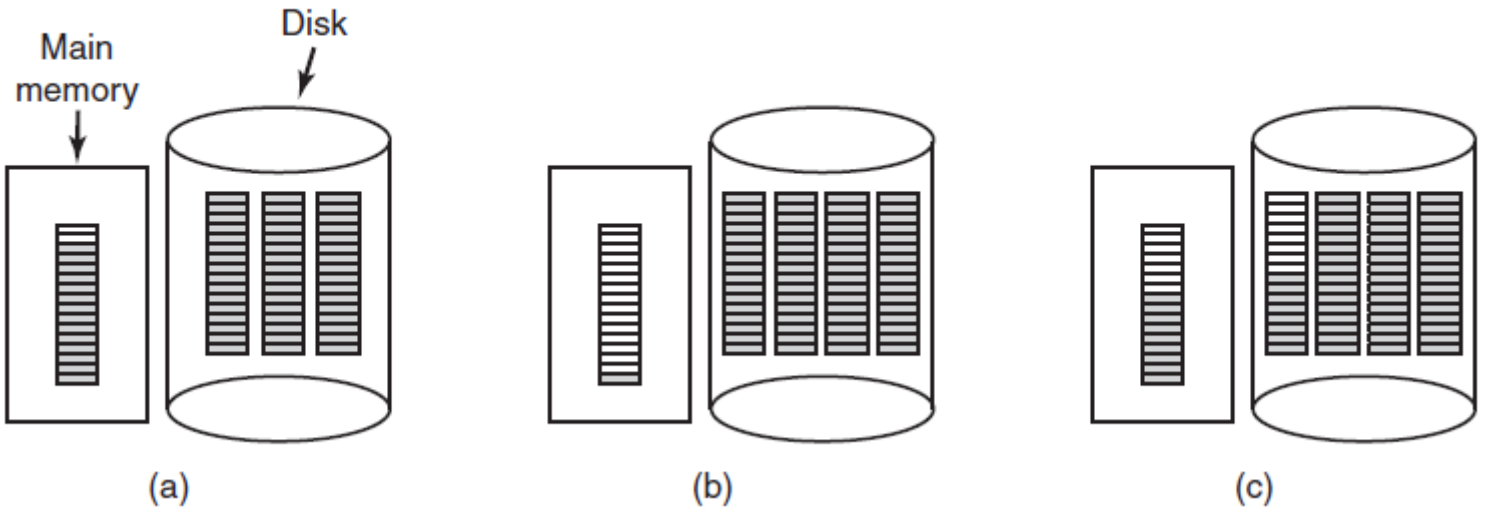
- A disk with n blocks requires a bitmap with n bits.
- Free blocks are represented by 1s in the map, allocated blocks by 0s (or vice versa).
- For our example 1-TB disk, we need 1 billion bits for the map, which requires around 130,000 1-KB blocks to store.
- It is not surprising that the bitmap requires less space, since it uses 1 bit per block, vs. 32 bits in the linked-list model.
- Only if the disk is nearly full (i.e., has few free blocks) will the linked-list scheme require fewer blocks than the bitmap.

# How to keep track of free blocks

- If free blocks come in runs, could keep track of beginning and block and run length. Need nature to cooperate for this to work.

- Only need one block of pointers in main memory at a time. Fills up => go get another

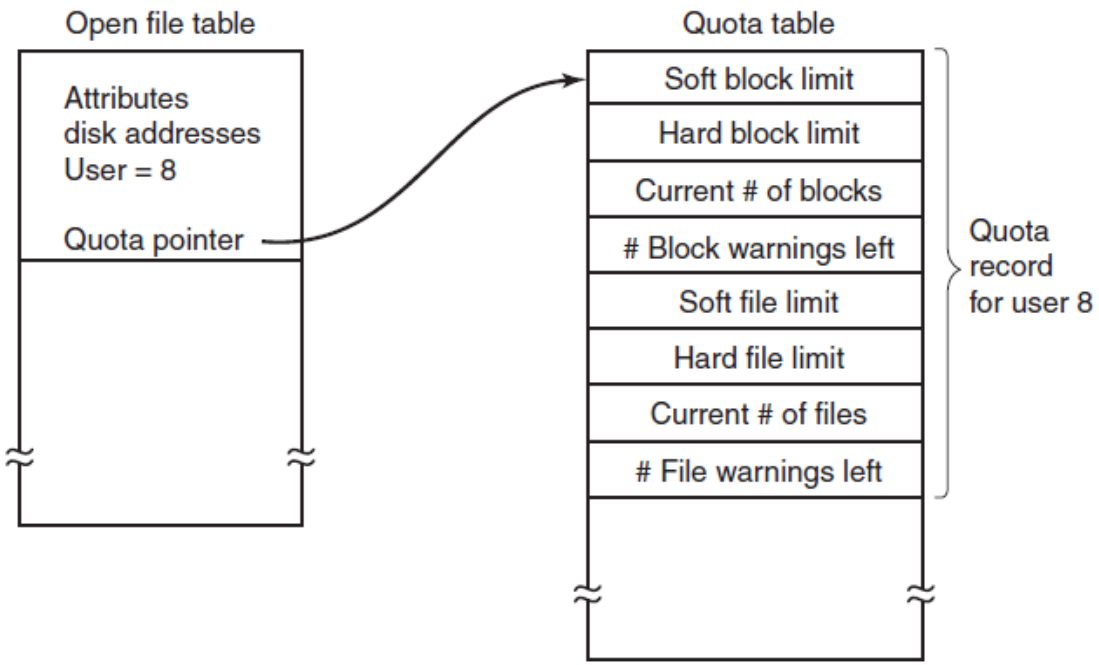
# Keeping Track of Free Blocks (2)




(a) An almost-full block of pointers to free disk blocks in memory and three blocks of pointers on disk. (b) Result of freeing a three-block file. (c) An alternative strategy for handling the three free blocks. The shaded entries represent pointers to free disk blocks.

# Disk Quotas - you can't have it all

System administrator assigns each user a maximum allotment of files and blocks, and OS makes sure that the users do not exceed their quotas.




Quotas are kept track of on a per-user basis in a quota table.

It is an extract from a quota file on disk for the users whose files are currently open.

When all the files are closed, the record is written back to the quota file.

Warnings and errors can be generated when soft and hard limits are exceeded.

# File System Backups (1)

While the file system cannot offer any protection against physical destruction of the equipment and media, it can help protect the information.

MAKE BACKUPS

(not quite as trivial as it sounds)

Backups to tape* are generally made to handle one of two potential problems:

1. Recover from disaster.
2. Recover from stupidity.

*Tapes hold hundreds of gigabytes and are very cheap

# File System Backups (2)

Making a backup takes a long time and occupies a large amount of space, so do it efficiently:

1. First, should the entire file system be backed up or only part of it?
   – it is usually desirable to back up only specific directories and everything in them rather than the entire file system.
2. Second, it is wasteful to back up files that have not changed since the previous backup (incremental dump - > complicates recovery)

# File System Backups (3)

3. Third, since immense amounts of data are typically dumped, it may be desirable to compress the data before writing them to tape (decreases reliability)
4. Fourth, it is difficult to perform a backup on an active file system (generates inconsistency)
5. Fifth and last, making backups introduces many nontechnical problems into an organization (off-site storage, security issues, etc.)

# File System Backups (4)

- Two strategies for dumping a disk to a backup disk: ***physical*** dump or ***logical*** dump.

A ***physical dump*** starts at block 0 of the disk, writes all the disk blocks onto the output disk in order, and stops when it has copied the last one.

  - Unused blocks and bad blocks should be handled properly
  - Adv: simplicity, speed
  - Disadv: inability to skip selected directories, make incremental dumps, and restore individual files upon request

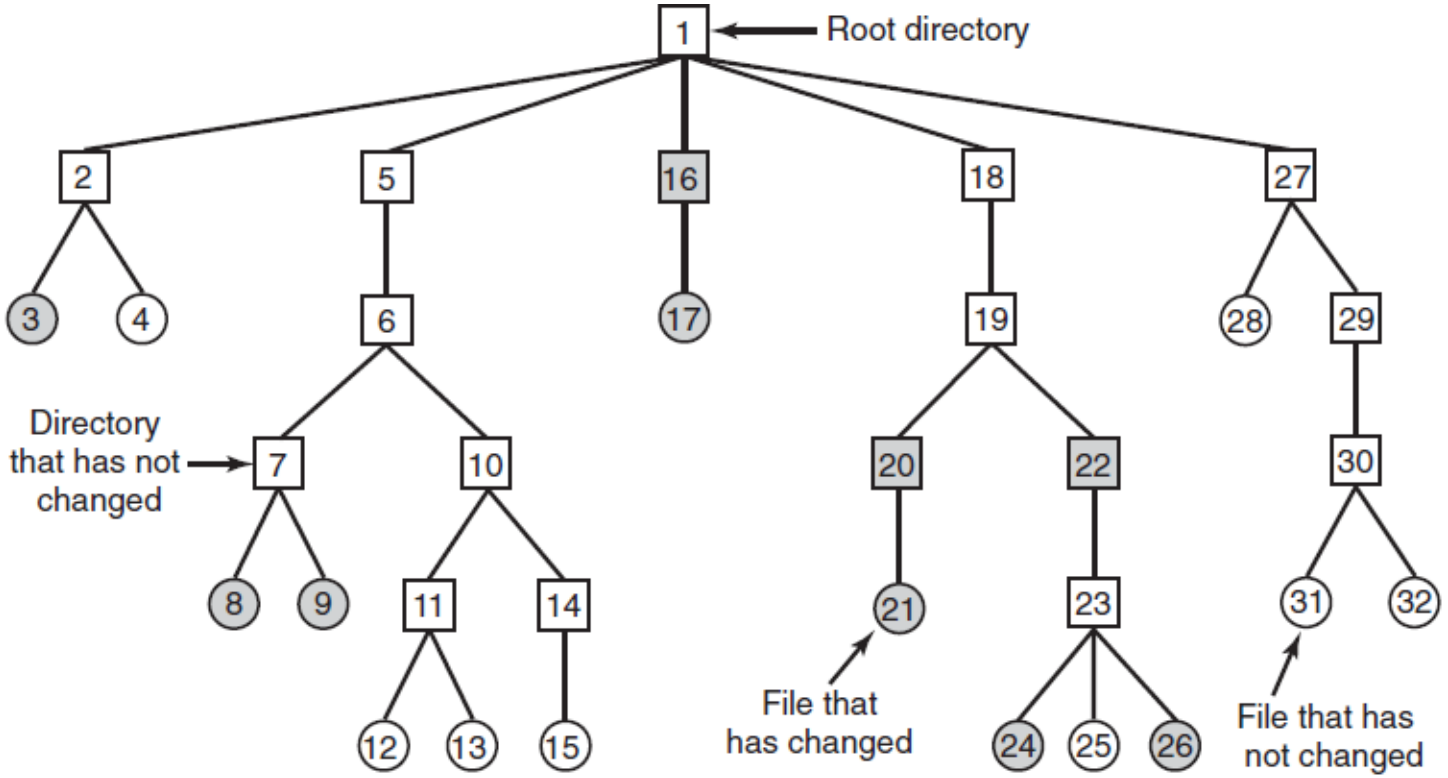
# File System Backups (5)

A ***logical dump*** starts at one or more specified directories and recursively dumps all files and directories found there that have changed since some given base date.

Example algorithm dumps all directories (even unmodified ones) that lie on the path to a modified file or directory for two reasons.

- to make it possible to restore the dumped files and directories to a fresh file system on a different computer.
- to make it possible to incrementally restore a single file (possibly to handle recovery from stupidity).

# File System Backups (6)

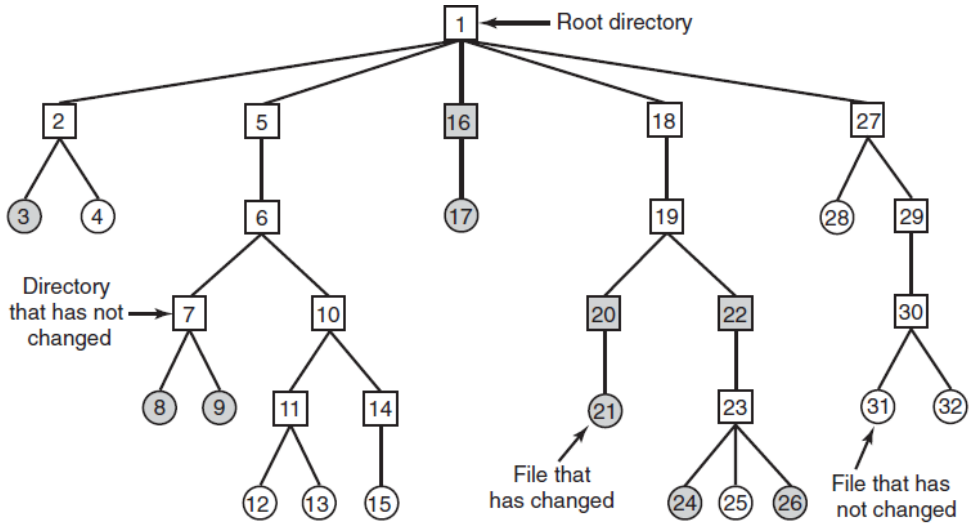



A file system to be dumped. The squares are directories and the circles are files. The shaded items have been modified since the last dump. Each directory and file is labeled by its i-node number.

# File System Backups (7)

- Phase 1: mark each directory and modified file
- Phase 2: unmark directories that have no modified files or directories in them or under them (10, 11, 14, 27, 29, and 30 in the example)
- Phase 3: scan i-nodes in numerical order and dump all the directories that are marked for dumping prefixed by their attributes.
- Phase 4: dump the files marked prefixed by their attributes.

# File System Backups (8)




(a) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

(b) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

(c) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

(d) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

Bitmaps used by the logical dumping algorithm.

# File System Backups (9)

Issues in restoring a ***logical dump:***

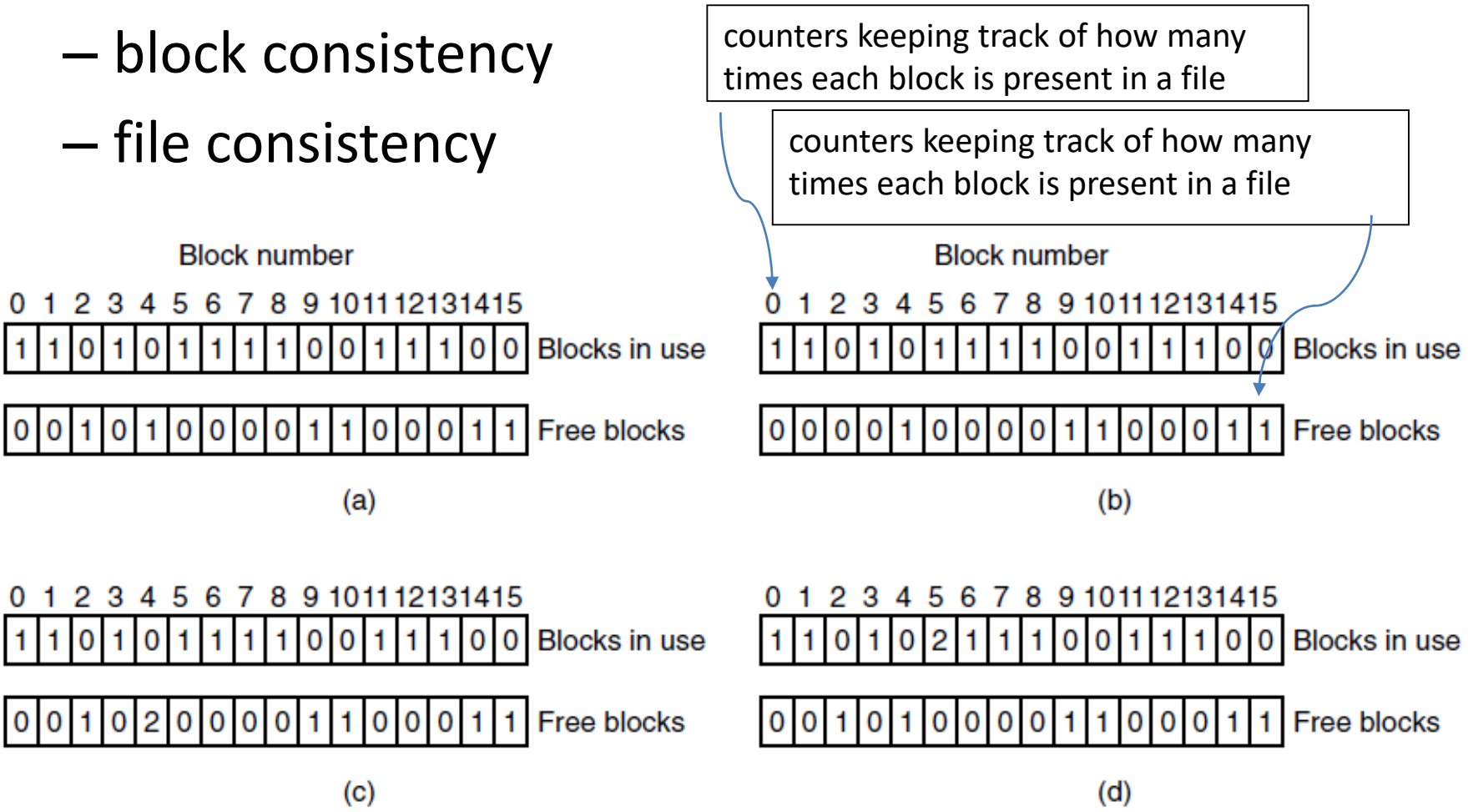
- restore free blocks list
- restore linked files only once
- do not restore holes in unix files
- do not dump and restore special files such as pipes

# File System Consistency

- Many file systems
  - read blocks, modify them, and write them out later.
- If the system crashes before all the modified blocks have been written out, the file system can be left in an ***inconsistent*** state.
- This problem is especially critical if some of the blocks that have not been written out are i-node blocks, directory blocks, or blocks containing the free list.
- Hence run the utility *fsck* (in unix) or *sfc* (in windows) especially after crash.

# File System Consistency

- Consistency check can be done for
  - block consistency
  - file consistency



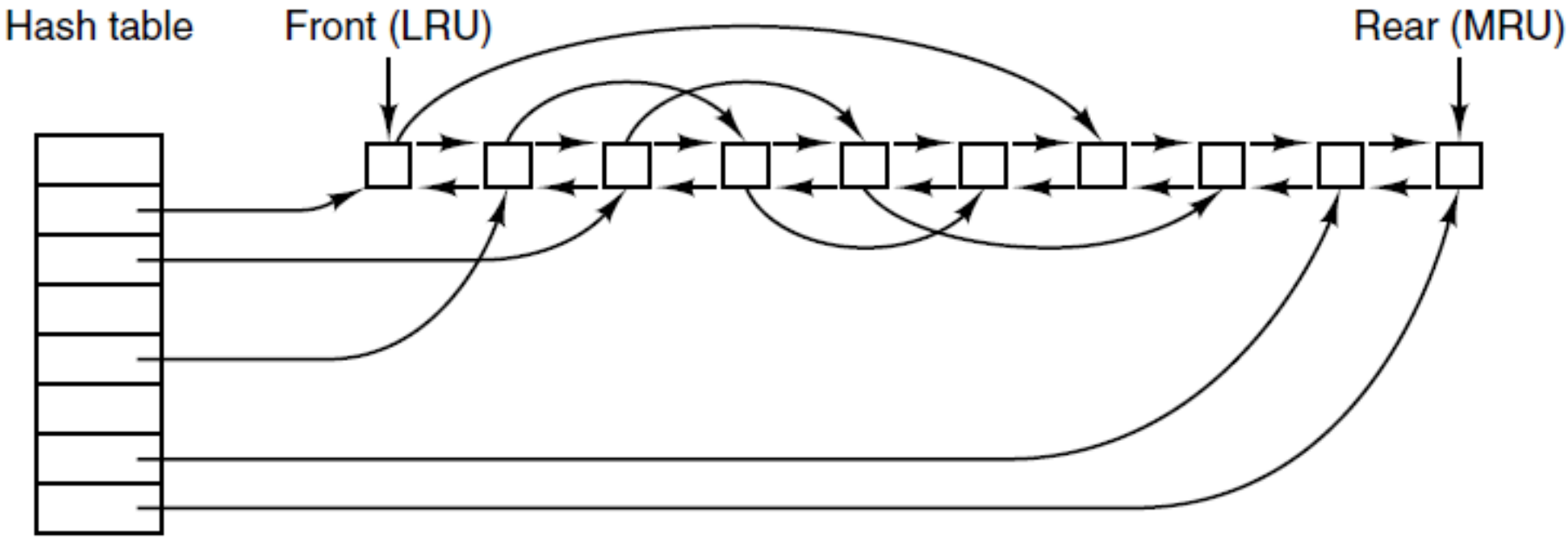

File system states. (a) Consistent. (b) Missing block.
(c) Duplicate block in free list. (d) Duplicate data block.

# File System Performance (1)

- Reading a 32 bit
  - memory word takes ~ 10nsec (@400MB/sec)
  - disk word takes ~ 40nsec (@100MB/sec) but after 5-10 msec seek time
- Hence
  - million times slower when a single word is to be accessed
  - many file systems have been designed with various optimizations to improve the performance.
  - *Caching*, *block-read-ahead* and *disk-arm motion reduction* are possible techniques to be used

# File System Performance - caching

- A buffer cache is a collection of blocks that logically belong on the disk but are being kept in memory for performance reasons.




The buffer cache (block cache) data structures.

# File System Performance - caching

- Many (often thousands of ) blocks in the cache, hence some way is needed to determine quickly if a given block is present.
- Hash the device and disk address and look up the result in a hash table.
- All the blocks with the same hash value are chained together on a linked list so that the collision chain can be followed.
- When cache is full, some block has to be removed. This is very much like paging, and all the usual page-replacement algorithms (FIFO, LRU, etc.) are applicable.

# File System Performance - caching

- Now exact LRU is possible but it may be undesirable (due to increased inconsistency probability)
- Some blocks rarely referenced two times within a short interval.
- Leads to a modified LRU scheme, taking two factors into account:
  1. Is the block likely to be needed again soon?
  2. Is the block essential to the consistency of the file system?

# File System Performance – read ahead

- To increase the hit rate
  - get blocks into the cache before they are needed
- When FS is asked to produce block k in a file, it does that, but when finished, it makes a sneaky check in the cache to see if block k + 1 is already there.
- If it is not, it schedules a read for block k + 1 in the hope that when it is needed, it will have already arrived in the cache.
- At the very least, it will be on the way.
- Of course, this strategy works only for files that are actually being read sequentially.

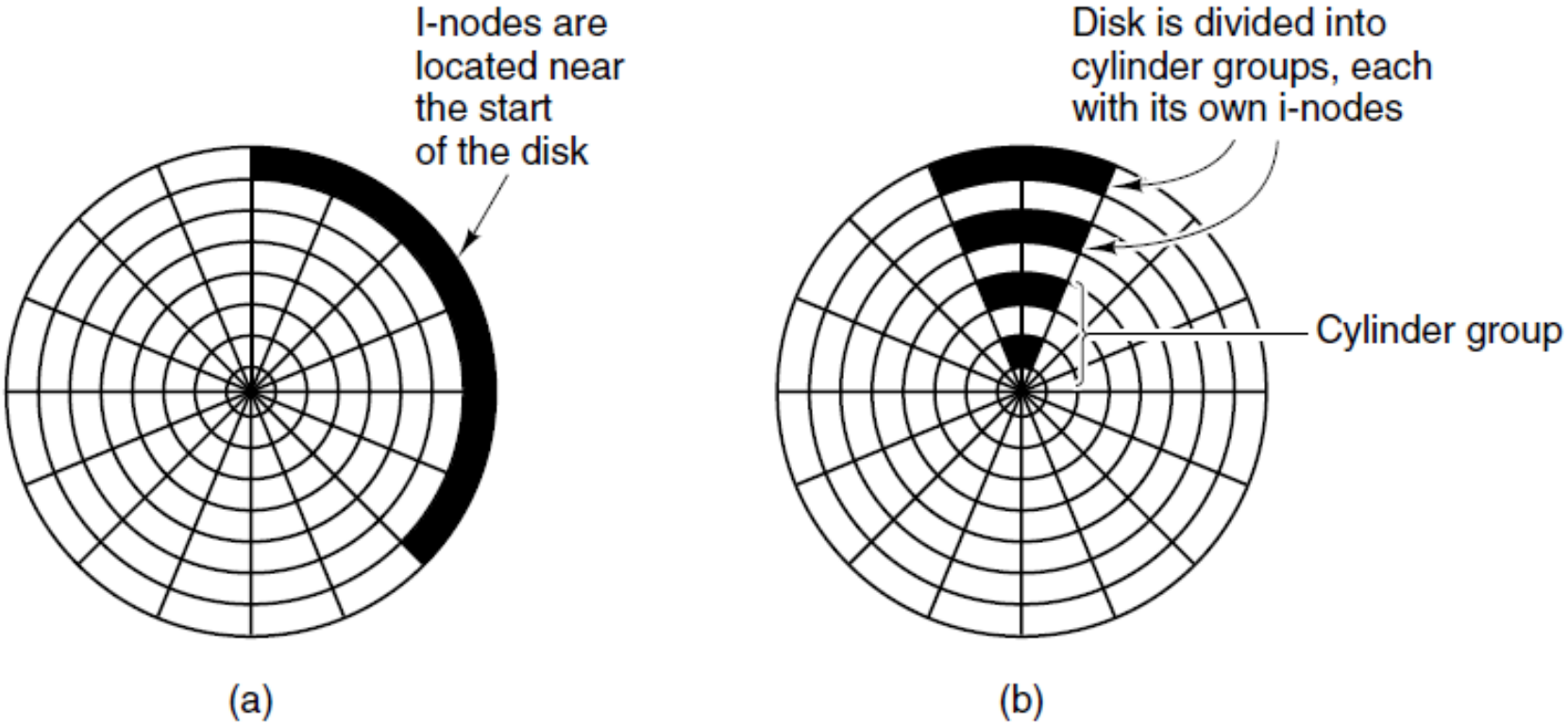
# File System Performance - Reducing Disk Arm Motion

- Reduce the amount of disk-arm motion by putting blocks that are likely to be accessed in sequence close to each other, preferably in the same cylinder
- When an output file is written, the file system has to allocate the blocks one at a time, on demand.
- If the free blocks are recorded in a bitmap, and the whole bitmap is in main memory, it is easy to choose a free block as close as possible to the previous block.
- With a free list, part of which is on disk, it is much harder to allocate blocks close together.

# File System Performance - Reducing Disk Arm Motion

- Another performance bottleneck in systems that use i-nodes or anything like them is that reading even a short file requires two disk accesses:
  - one for the i-node
  - and one for the block.
- In usual i-node placementall the i-nodes are near the start of the disk, so the average distance between an i-node and its blocks is half the number of cylinders, requiring long seeks.

# File System Performance - Reducing Disk Arm Motion




(a) i-nodes placed at the start of the disk. (b) Disk divided into cylinder groups, each with its own blocks and i-nodes.

# File System Performance - Reducing Disk Arm Motion

- Disk-arm movement and rotation time are not issues related to **solid-state disks** (**SSD**), which have no moving parts whatsoever.
- For these disks, built on the same technology as flash cards, random accesses are just as fast as sequential ones
- But in SSDs, each block can be written only a limited number of times, so great care is taken to spread the wear on the disk evenly.

# Example File Systems

- Sec 4.5 Reading Assignment

# End

Chapter 4