Lecture 15: File Systems

Professor G. Sandoval

Some slides derived from : Tanenbaum/Bo, John Regehr, and Brendan Dolan-Gavitt
Thanks !!

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.

File Systems (2)

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

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

File Systems (3)

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 System Layers

User's viewpoint:

- Objects: Files, directories, bytes
- Operations: Create, read, write, delete, rename, move, seek, set attributes

Physical viewpoint:

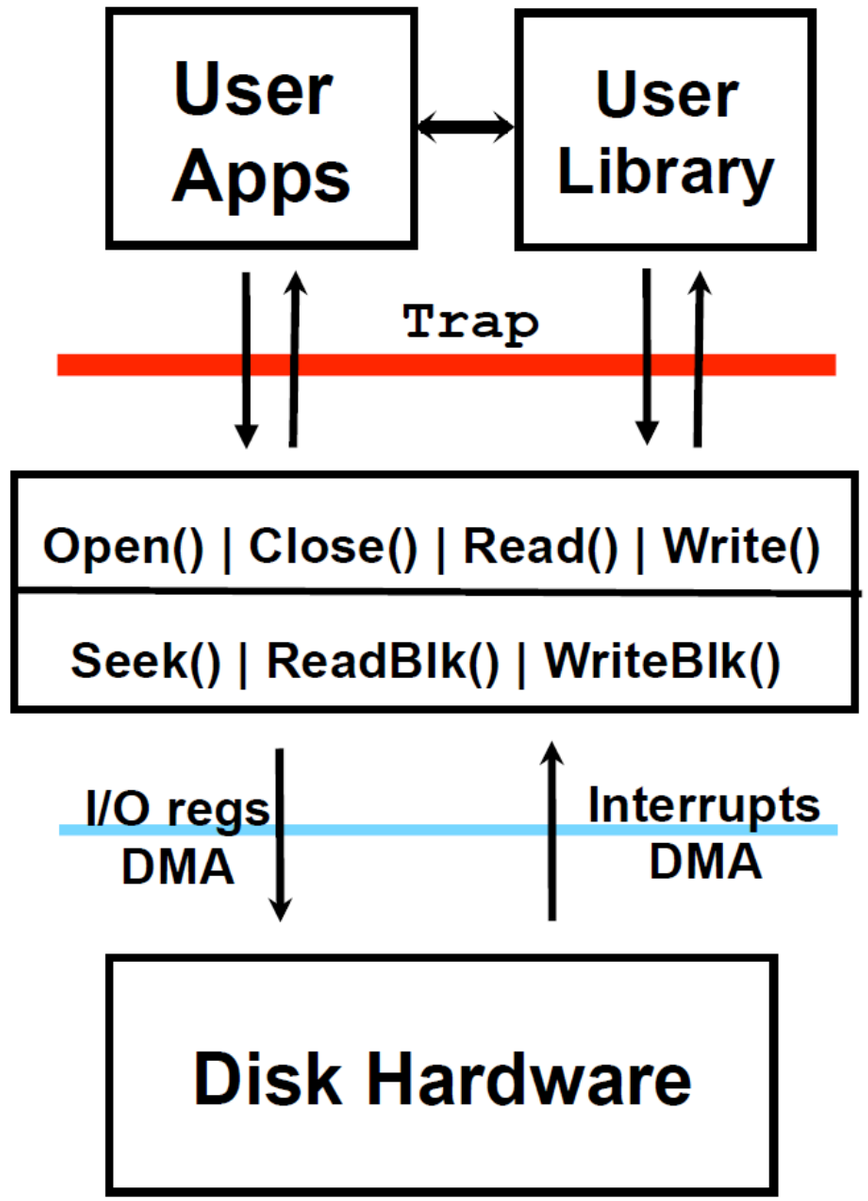
- Objects: Sectors, tracks, disks
- Operations: Seek, read block, write block

User ←→ OS layer

- User library hides many details
- OS can directly read/write user data

OS ←→ Hardware layer

- IO registers
- Interrupts
- DMA



Today

- Files
- Directories
- File System Implementation
  - Implementing Files
  - Implementing Directories
  - Shared Files
  - Free Space

Files

- Given that providing direct access to disk blocks is unacceptable, we need some abstraction
- A *file* is that abstraction – a logical unit of data that is backed by multiple underlying blocks on disk
  - In some ways, this is like having an address space for memory, except the underlying resource is a disk

Naming

- We need some way of referring to the data in a file, i.e. a name
- File names typically have some set of allowed characters that varies depending on the OS and filesystem
  - For example, * is allowed in Linux filenames, but not in Windows
  - OS X allows emoji 👍💯🔥
- Other differences: name length limits (255 on most Linux filesystems, 8.3 in DOS), case sensitivity

Filesystem

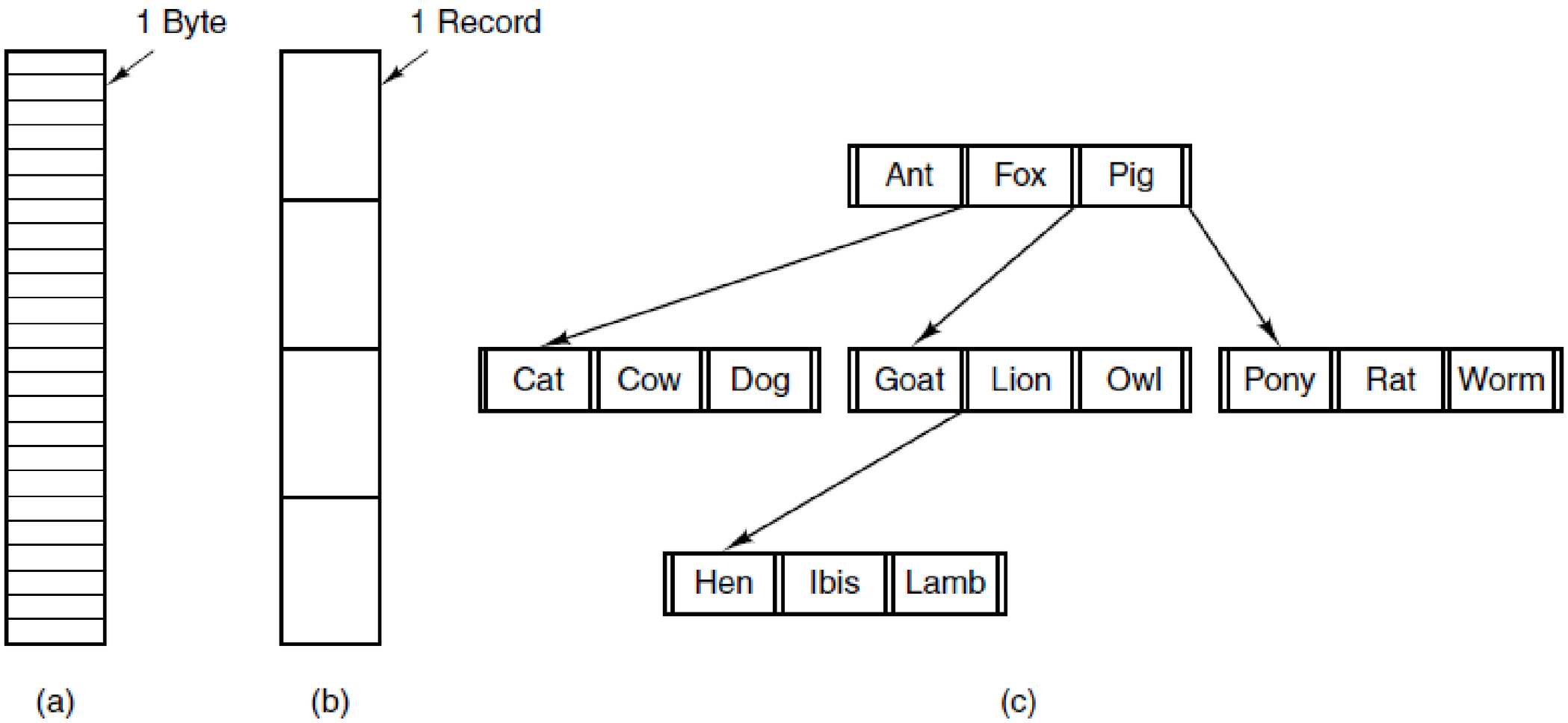
- Typically when we say *filesystem* we refer to the on-disk layout and data structures used to represent files

- Many filesystems invented over the years:
  - FAT16/FAT32 – used by DOS/Windows
  - NTFS – used by Windows NT and later
  - HFS+ – used by OS X
  - ext2/3/4, reiserfs, btrfs, ... – Linux
  - ISO9660 – used for CDs/DVDs

Structure of Files

- In most OSes, the *content* of a file on disk is stored as just an unstructured bunch of bytes
  - Any structure is specific to the application
- However, some operating systems have structured files:
  - Fixed size blocks (records)
  - Key-value storage

File Structure



Three kinds of files. (a) Byte sequence.
(b) Record sequence. (c) Tree.

Identifying File Types

- An OS may need some way of identifying the *type* of a file so that it can be opened by the correct application
- One way to do this is by giving different types of files different *extensions* – .mp3, .jpg, etc.
- Another way is to look at the first few bytes and guess based on a *magic number*
- Yet another way is to store information in some *filesystem metadata*
- A final way (used on Linux) is to not try to guess and make the user explicitly specify the program

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.

Guessing with Magic

- Simple idea in principle – file types can be identified by looking at the first few bytes
  - JPEG files start with 0xFF 0xD8
  - DOS/Windows executables start with "MZ" (0x4D 0x5A)
  - Boot sector ends with 0x55 0xAA
- Linux & OS X come with a utility called "file" that uses a database of such signatures to identify file types
- But not all file types can be identified this way, and the identification is not always unambiguous

"Openoffice Can't Print on Tuesdays"

- In 2008, some people started reporting that the word processor OpenOffice would randomly fail to print
  - "When I click print I get nothing." -Tuesday, August 5, 2008
  - "I downloaded those updates and Open Office Still prints." -Friday, August 8, 2008
  - "Open Office stopped printing today." -Tuesday, August 12, 2008
  - "I just updated and still print." -Monday, August 18, 2008
  - "I stand corrected, after a boot cycle Open Office failed to print." -Tuesday, August 19, 2008

What Happened?

- Printing in OpenOffice converts the file to PostScript and includes the date:

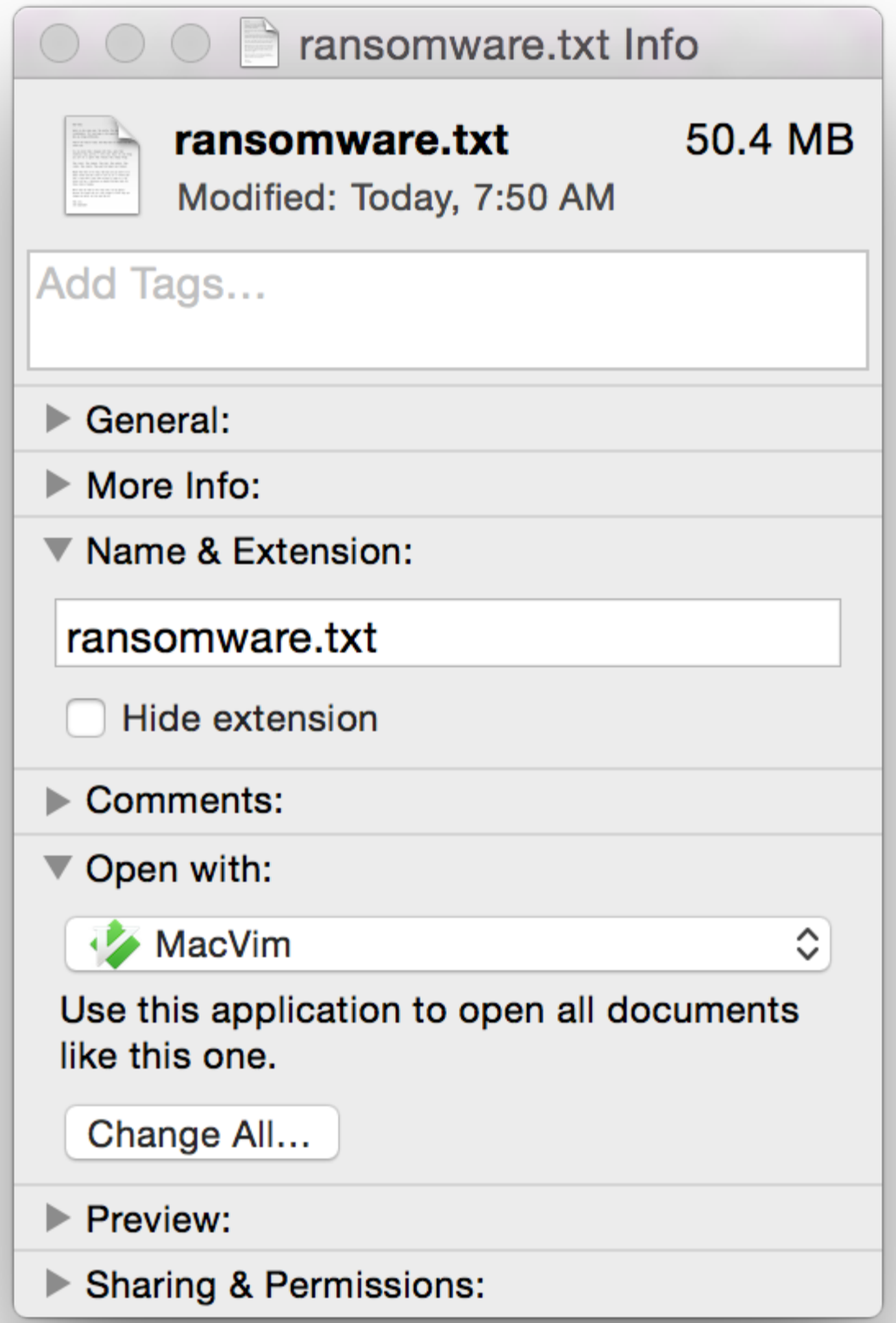
```
%!PS-Adobe-3.0
%%Creator: (OpenOffice.org 2.4)
%%For: (steve)
%%CreationDate: (Tue Mar  3 19:47:42 2009)
```

- Then the printing daemon reads the file, checks to make sure it's a valid PostScript file using the **file** command, and prints it

- **file** had a bug – if "Tue" appeared at the right position in the file, it would be misidentified as a particular database.

File Associations as Metadata

- This strategy is used by OS X
- HFS+ has a notion of *resource forks* – a hidden set of metadata stored in the filesystem
- You can set a specific file to be opened by a particular application, and that information will be saved in the resource fork
- If no resource fork exists, OS X falls back to using the file extension


ransomware.txt Info
ransomware.txt
50.4 MB
Modified: Today, 7:50 AM
Add Tags…
General:
More Info:
Name & Extension:
ransomware.txt
Hide extension
Comments:
Open with:
MacVim
Use this application to open all documents like this one.
Change All…
Preview:
Sharing & Permissions:

File Attributes

- Aside from associations, filesystems can keep many other kinds of metadata about a file
  - Permissions
  - Creator/owner
  - Hidden
  - Locked
  - Timestamps
  - File size

Side Note: Timestamps

- Timestamps on files are stored as filesystem metadata

- Even regular users can modify the timestamps on files they create!

- They can't be relied on to prove a file was created at a particular time (**demo**)

Operations on Files

- There are many high-level operations that can be provided for working with files in an OS:

- Create
- Delete
- Open
- Close

- Read
- Write
- Append
- Seek

- Get Attributes
- Set Attributes
- Rename

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 */
```

Figure 4-5. 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 */
```

Figure 4-5. 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 */
}
```

Figure 4-5. A simple program to copy a file.

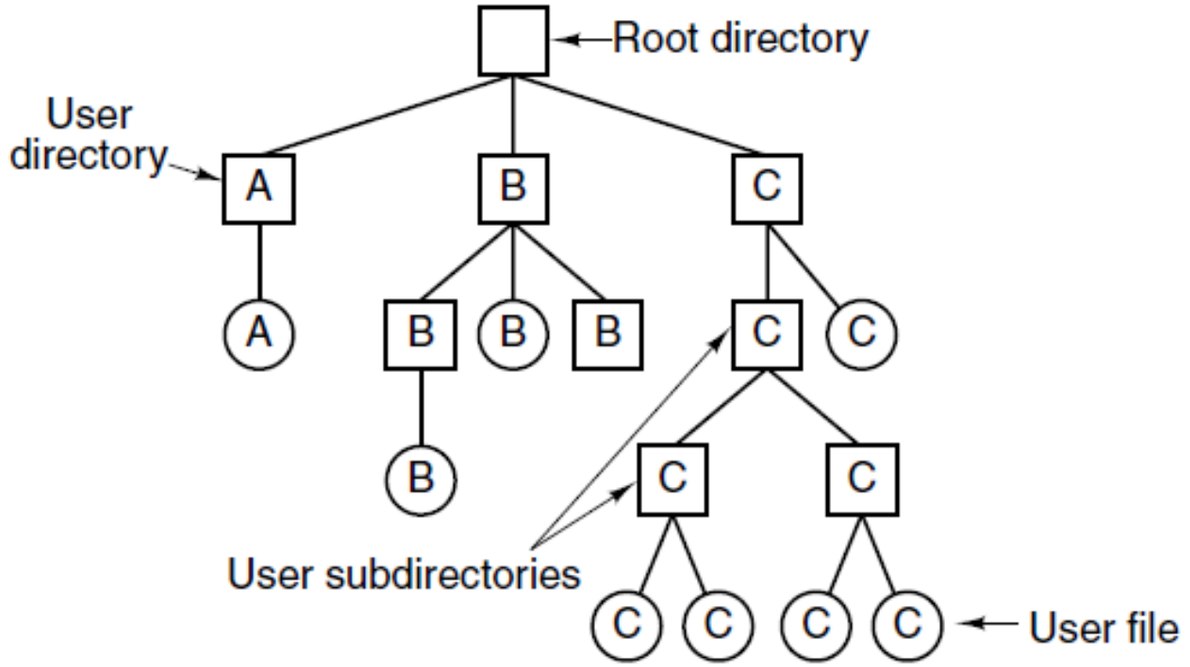
Today

- Files
- Directories
- File System Implementation
  - Implementing Files
  - Implementing Directories
  - Shared Files
  - Free Space

Directories

- Directories are filesystem entries that store lists of files
- Simplest filesystems have no hierarchy – all files are kept in one top-level directory
- Most filesystems now are *hierarchical* – form a tree structure
- Note: a system might have multiple filesystem trees
  - E.g., Windows has one tree for each drive (C:, D:, etc.)
  - (Why does it start at C?)

Hierarchical Directory Systems



A hierarchical directory system.

Paths

- To specify a file, you specify a *path* through the filesystem tree
- Each directory in a path is delimited by a *path separator* which varies between OSes
  - C:\Users\Gustavo\Desktop\homework.txt
  - /Users/Gustavo/Desktop/homework.txt
  - Users>Gustavo>Desktop>homework.txt

Relative Paths

- We saw that one of the pieces of information stored in a process data structure is the *current working directory*
- This allows us to specify files without giving their full path, e.g. if working directory is /home/gustavo:
  - cp a.txt b.txt
  - cp /home/gustavo/a.txt /home/gustavo/b.txt
- Most OSes also support two special path components: "." and ".." for "current directory" and "parent directory"
  - /home/gustavo/../a.txt = /home/a.txt

Working Directory

- What happens if a library function needs to change the working directory?
  - For example, perhaps it needs to call another command that assumes files are in the current directory
- If it doesn't change the working directory back before returning, the rest of the program may write files into the wrong place
- Library functions that modify the working directory are also not *thread safe*, because the working directory is shared by all threads in a process

Directory Operations

- As with files, directories support a typical set of operations:
  - Create
  - Delete
  - Opendir
  - Readdir
  - Closedir
  - Rename
  - Link
  - Unlink

Special Files

- Some OSes have *special files*, which do not correspond to data on disk
- We have already seen *device files*, which let you treat hardware devices as simple files
- These need to be identified specially somehow
  - In UNIX, filesystem metadata is used to distinguish
  - DOS just had special names – AUX, CON, LPT1

DOS Devices

Name	Function
CON	Keyboard and display
PRN	System list device, usually a parallel port
AUX	Auxiliary device, usually a serial port
CLOCK$	System real-time clock
NUL	Bit-bucket device
A:-Z:	Drive letters
COM1	First serial communications port
LPT1	First parallel printer port
LPT2	Second parallel printer port
LPT3	Third parallel printer port
COM2	Second serial communications port
COM3	Third serial communications port
COM4	Fourth serial communications port

Mounting

- On systems with a unified directory hierarchy (UNIX-like OSes), we may want to connect multiple filesystems together
- This is done by *mounting* a filesystem
- Attaches the new filesystem hierarchy at an existing (empty) directory
- In Linux/OS X, this is done with the *mount* command

Questions

- How is this all implemented?!
- How do we make it fast? (Buffering, Caching)
- How do we make it reliable?
- How does xv6's filesystem work?

Today

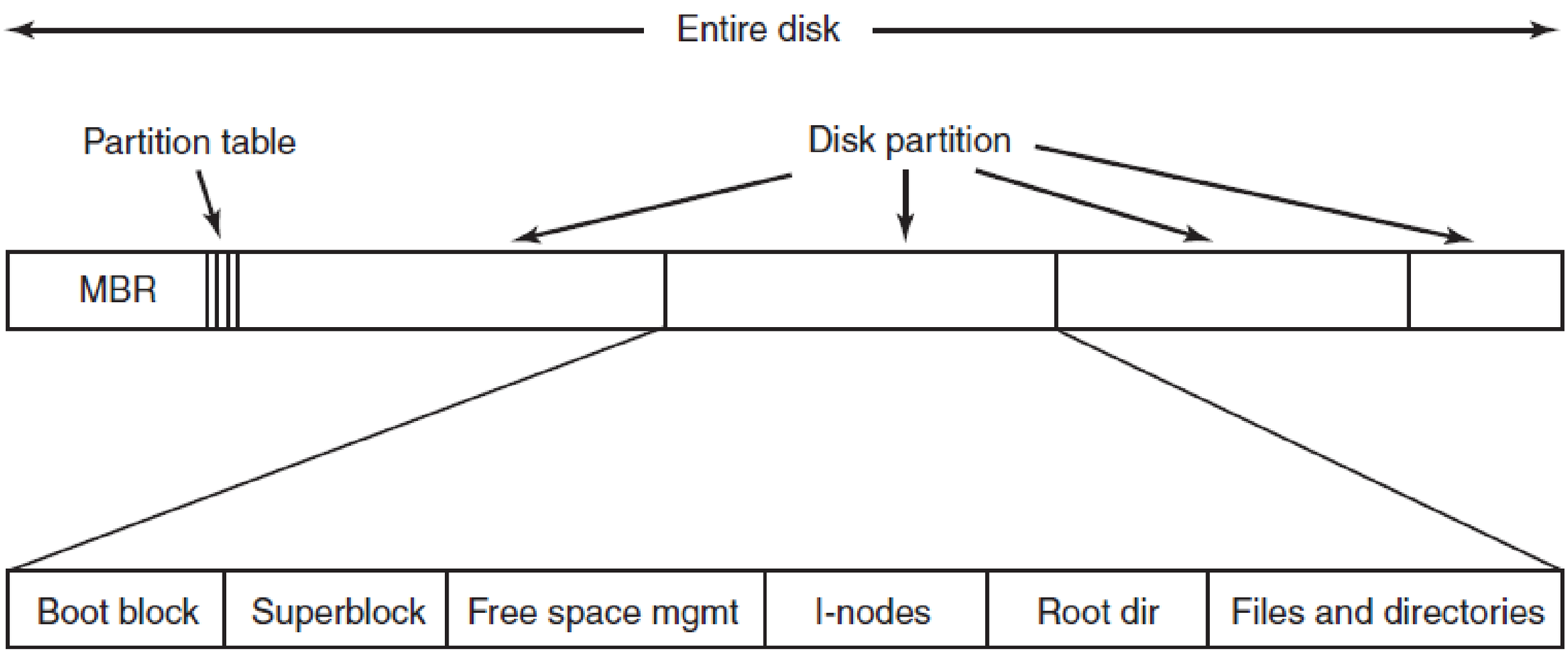
- Files
- Directories
- File System Implementation
  - Implementing Files
  - Implementing Directories
  - Shared Files
  - Free Space

Filesystem Layout on Disk

- A filesystem is defined by the **layout of data** on disk and the **algorithms** used to store and retrieve that data

- A typical disk layout consists of
  - A ***master boot record*** that contains code for booting the system
  - Multiple partitions, each with their own *boot block*
  - Within a partition, the actual filesystem data, usually beginning with a *superblock* that has key filesystem parameters

One Filesystem Layout



Example Superblock (xv6)

```
struct superblock {
  uint size;         // Size of file system image (blocks)
  uint nblocks;      // Number of data blocks
  uint ninodes;      // Number of inodes.
  uint nlog;         // Number of log blocks
  uint logstart;     // Block number of first log block
  uint inodestart;   // Block number of first inode block
  uint bmapstart;    // Block number of first free map block
};
```

Today

- Files
- Directories
- File System Implementation
  - Implementing Files
  - Implementing Directories
  - Shared Files
  - Free Space

Implementing Files

- One of the most important things to decide in a filesystem is how to map between files and disk blocks

- Possible implementations:
  - Contiguous allocation
  - Linked list (on-disk or in-memory)
  - I-nodes

Contiguous Allocation

- Conceptually trivial: just put all the blocks for each file one after another on disk
- Has many of the same problems of non-virtual memory management:
  - As files are created and deleted, the layout becomes *fragmented*, wasting space

Contiguous Allocation

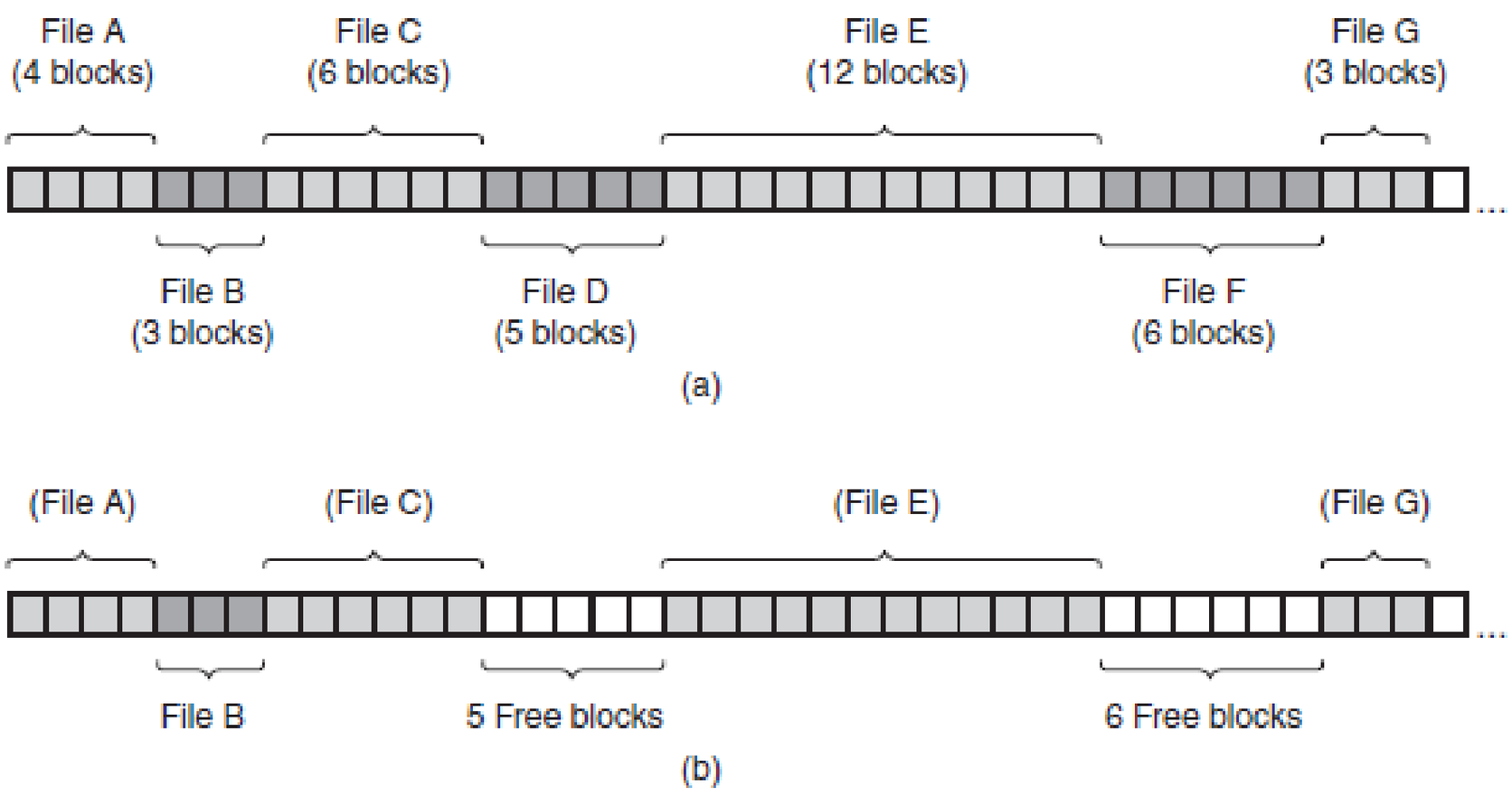
Advantages:

- Simple to implement
- Excellent read performance

Disadvantages:

- Fragmentation

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

- Despite these disadvantages, contiguous allocation is still used in CDs and DVDs (ISO9660 and UDF)

- Since these are written all at once and don't get updated over time, we avoid fragmentation

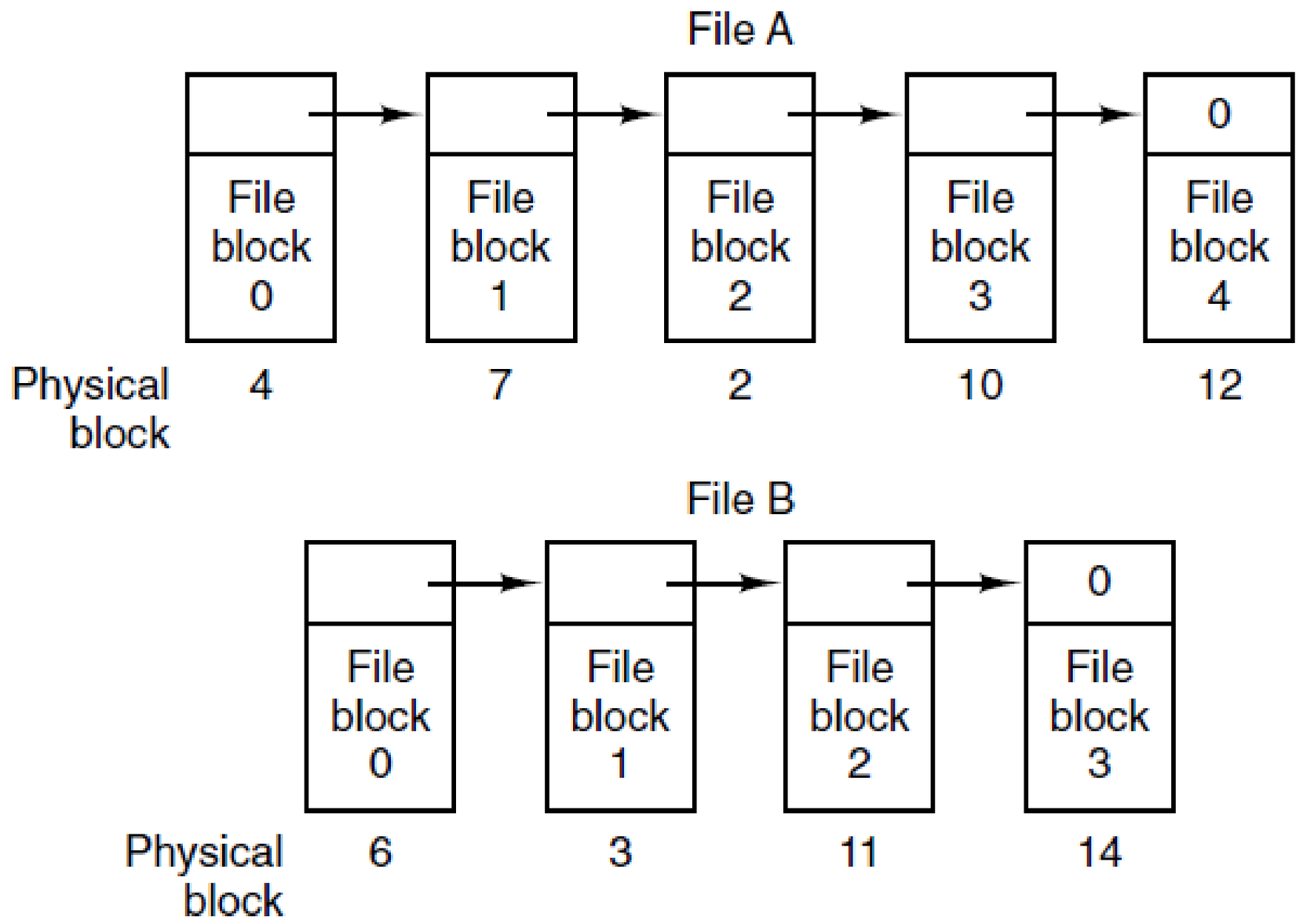
- Size of every file is known ahead of time

Linked List Allocation

- Inside each file block, keep a pointer to the next block in the file

- No issues with fragmentation & wasted space

Linked List Allocation



Linked List Allocation

- Storing a linked list on disk has performance issues though:
  - To read disk block *n*, we need to read *n-1* blocks before it
  - Entire block can no longer be used for data (we need space for the next pointer), so reads of file block must span two physical blocks

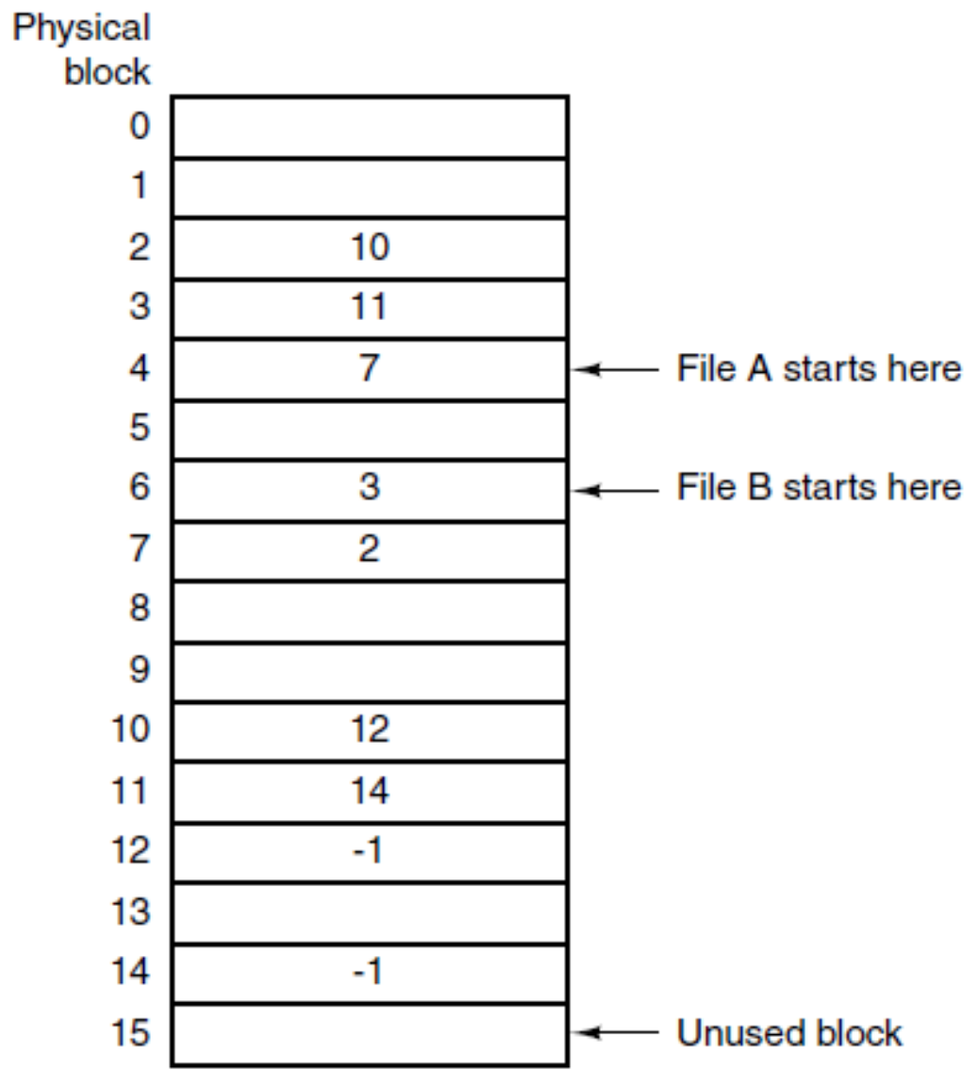
Linked List Allocation (In-Memory)

- Improvement: keep a table in memory with just the pointer information

- Each entry corresponds to a physical block and contains the pointer to the next block

- This is called a *File Allocation Table (FAT)* and is how the FAT16/FAT32 filesystem works

File Allocation Table



Disadvantages of FAT

- FAT is, well, fat – if you have a large disk, it takes up a lot of space in memory
- Suppose we have a 1TB disk, 1KB blocks:
  - (1 TB / 1 KB) entries * (4 bytes / entry) = 4GB for FAT
- For 32GB disk with FAT32 (4KB blocks):
  - (32GB / 4KB) entries * (4 bytes / entry) = 32MB

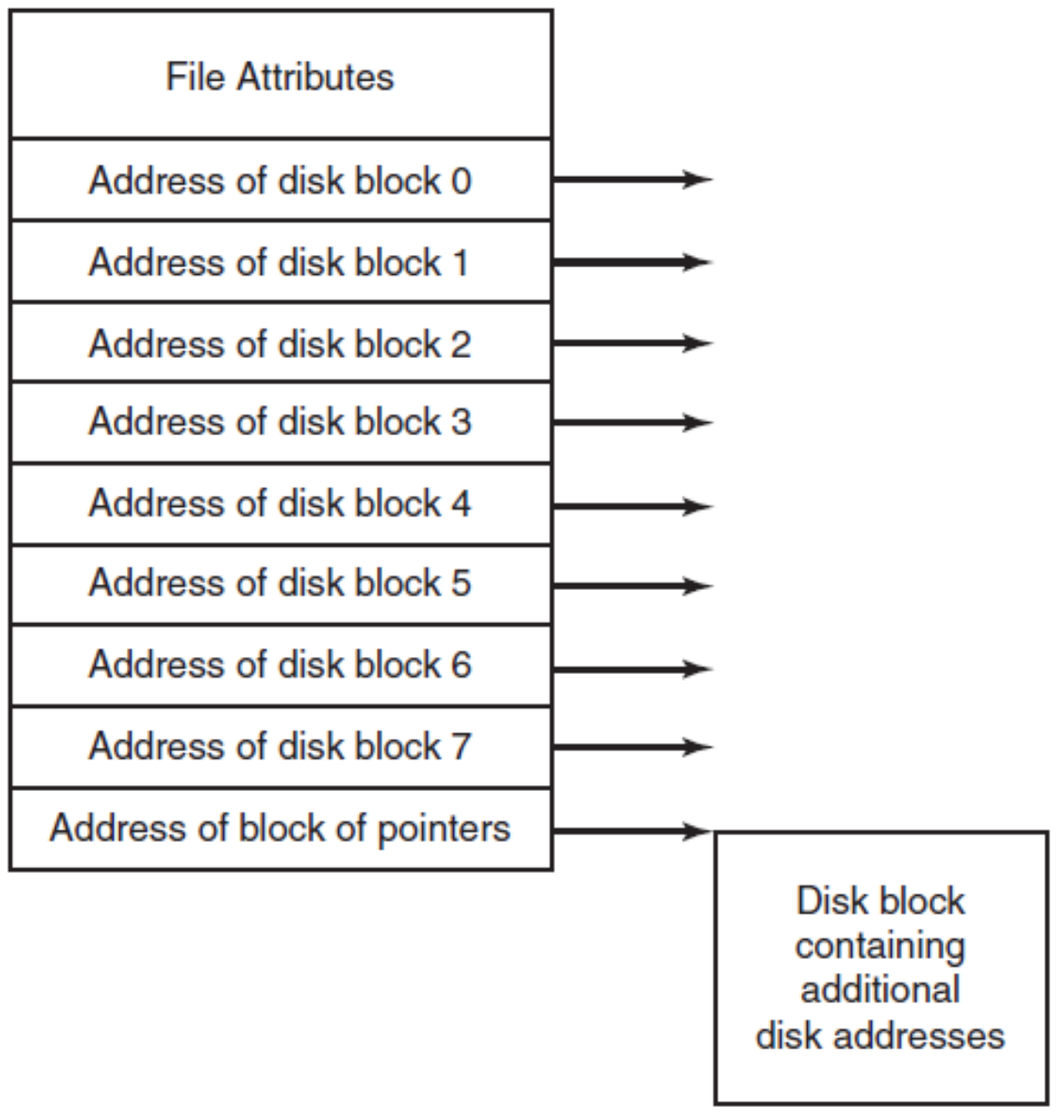
I-Nodes

- A more compact way is to give each file a small data structure that lists its blocks, called an *i-node* (*index node*)

- Now we only need to store in memory an amount of data proportional to the number of open files

- Amount of storage needed to store the mapping grows with the number of files, *not* the total size of the disk

I-Nodes


File Attributes
Address of disk block 0
Address of disk block 1
Address of disk block 2
Address of disk block 3
Address of disk block 4
Address of disk block 5
Address of disk block 6
Address of disk block 7
Address of block of pointers
Disk block containing additional disk addresses

Today

- Files
- Directories
- File System Implementation
  - Implementing Files
  - Implementing Directories
  - Shared Files
  - Free Space

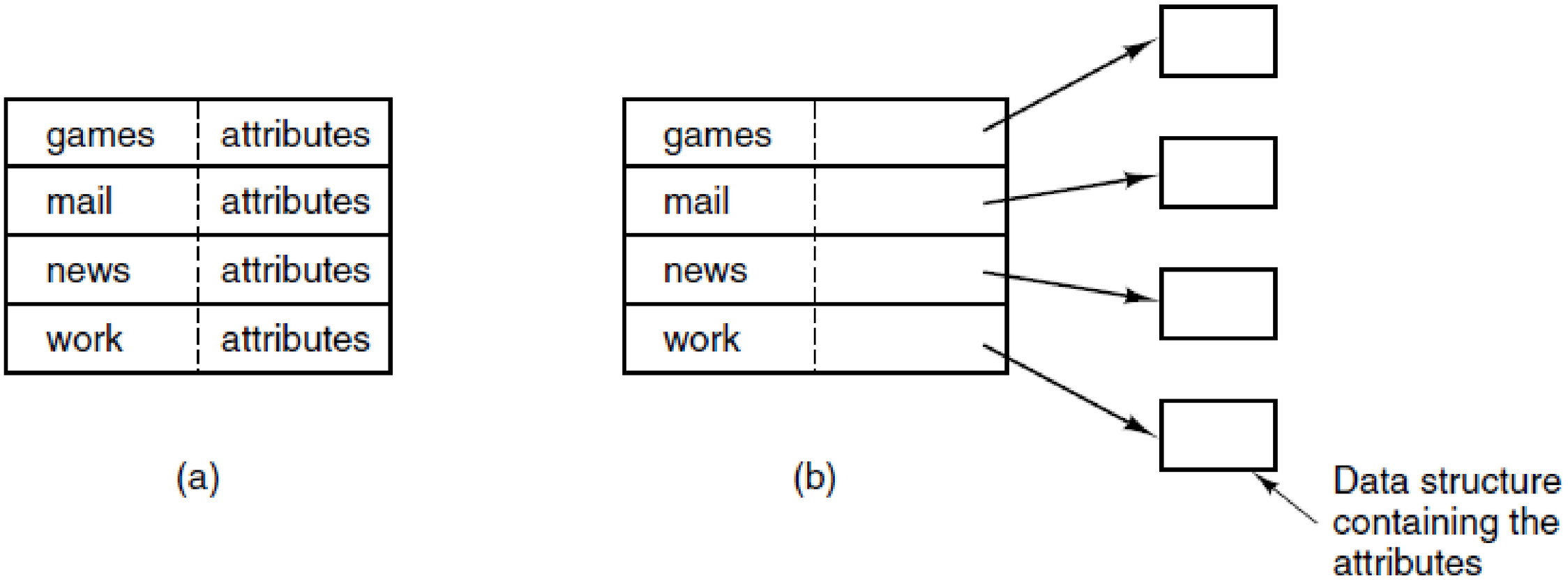
Implementing Directories

- To actually open a file, we supply a path, which gives a list of directories and a filename
- The directory entry provides the info to find the disk blocks.
- A directory must have some representation that stores the list of filenames, attributes, and pointer to the file data
- Basically needs to implement the map filename => file data

Directory Design Decisions

- How is the file data located using the directory entry?
  - Usually determined by how files are represented
    - E.g., with i-nodes you just store a pointer to the i-node
- How should we store the file attributes?
- How should we store filenames?

Implementing Directories

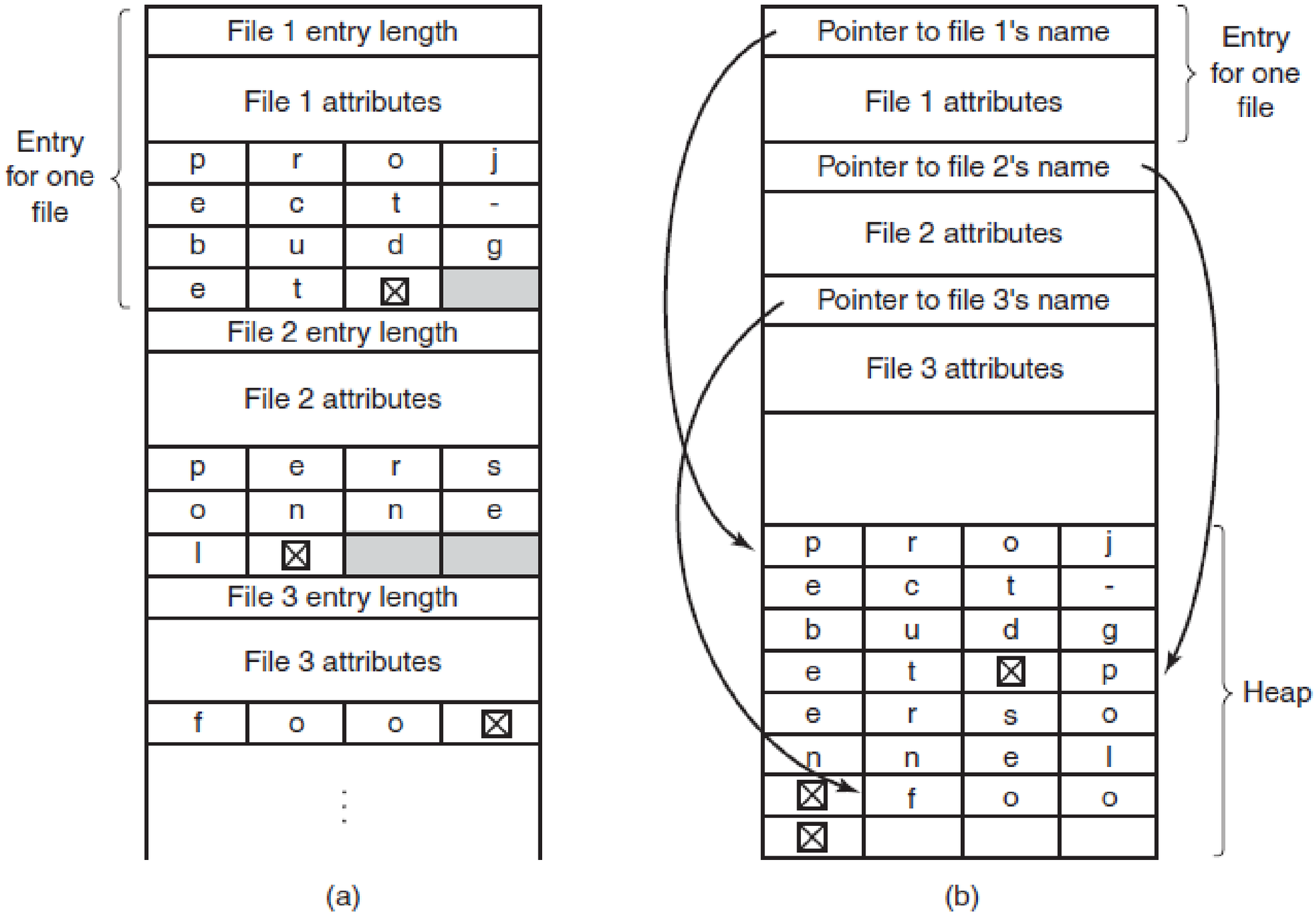


(a) A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry. (b) A directory in which each entry just refers to an i-node.

Filename Storage

- Simplest: just pick a maximum filename length and always store
- But if we want to allow long names, this is really wasteful
- Some other options:
  - Make each file entry structure in a directory variable-length
  - Put all filenames at the end of the directory entry and store pointers to them in each file entry

Implementing Directories

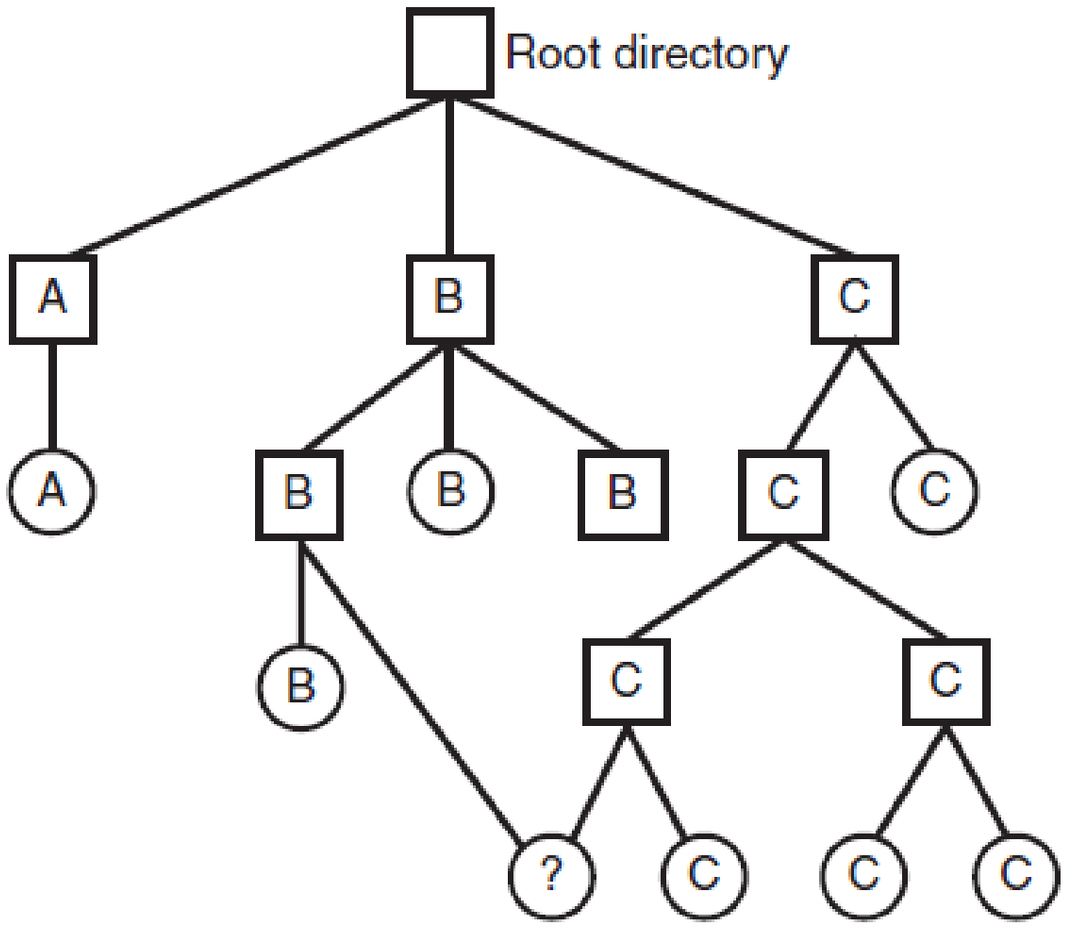


Two ways of handling long file names in a directory. (a) In-line. (b) In a heap.

Today

- Files
- Directories
- File System Implementation
  - Implementing Files
  - Implementing Directories
  - Shared Files
  - Free Space

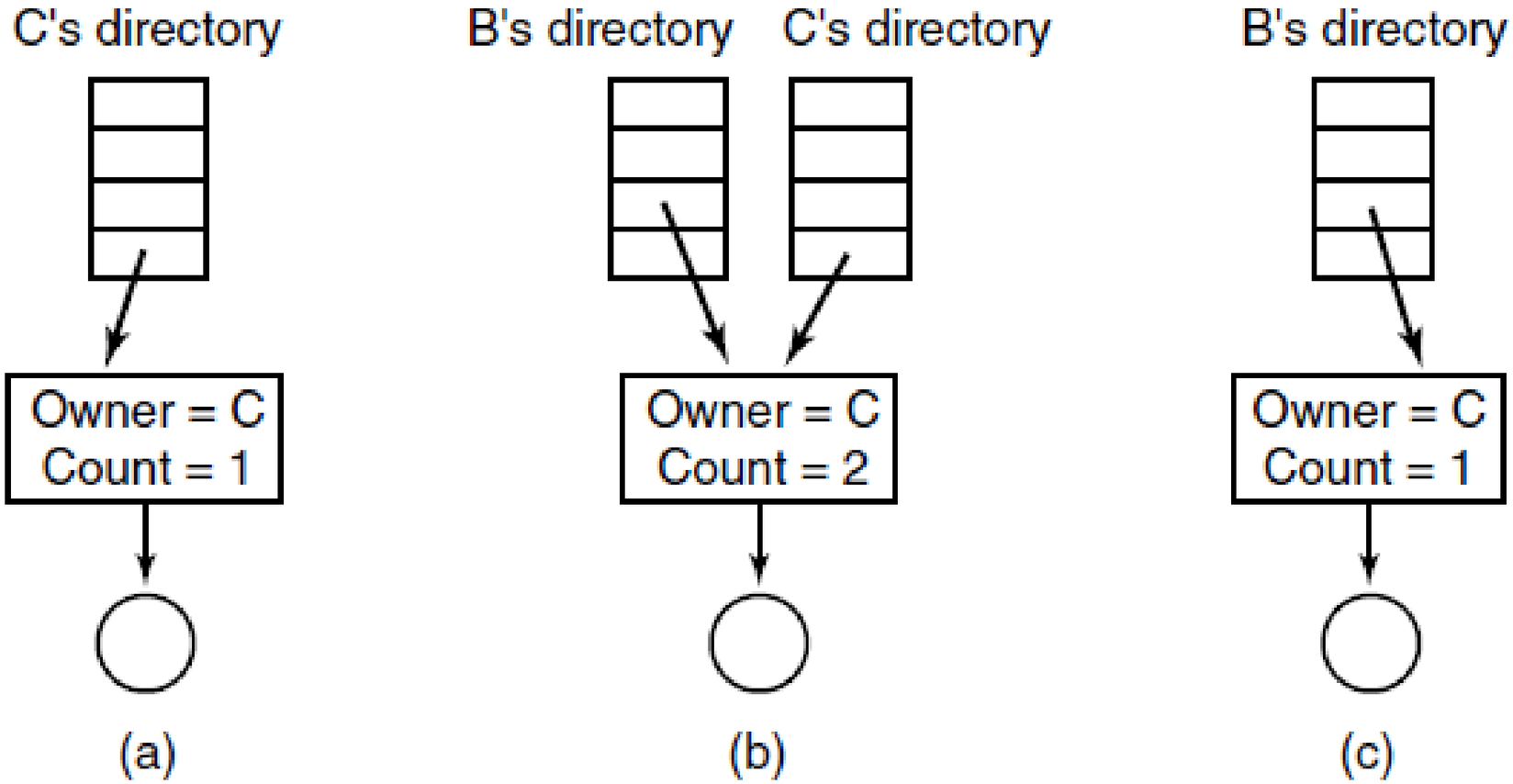
Shared Files (1)



Shared file

File system containing a shared file.

Shared Files (2)



(a) Situation prior to linking. (b) After the link is created. (c) After the original owner removes the file.

Hard and Symbolic Links

- Sometimes it's convenient to have multiple names for the same file
  - E.g., have a file shared by making it accessible from multiple users' home directories
- How can we do this in terms of directory entries?

Hard and Symbolic Links

- **Hard link**: just have each directory store a pointer to the same file information; e.g., same i-node

- **Symbolic link (symlink)**: create a new directory entry type, which stores the path to the link target

- Note: in UNIX systems, hard links can be created with `ln`, symlinks can be created with `ln -s`

Hard/Symlink Comparison

- Hard links can only point to files on the same filesystem (why?), but deleting renaming or moving the original file will not affect the hard link as it links to the underlying inode

- With hard links, we must keep a count of how many links to the file exist – otherwise we won't know when we can delete a file

- Symbolic links have extra overhead – we have to store a full path to the target file

Linking Pitfalls

- Since you now can have multiple names for the same file, you have to be careful not to duplicate data when copying
- With symbolic links, you can also create links to directories
  - This means you can create loops in the filesystem!
  - Most standard utilities detect this, but some applications might not

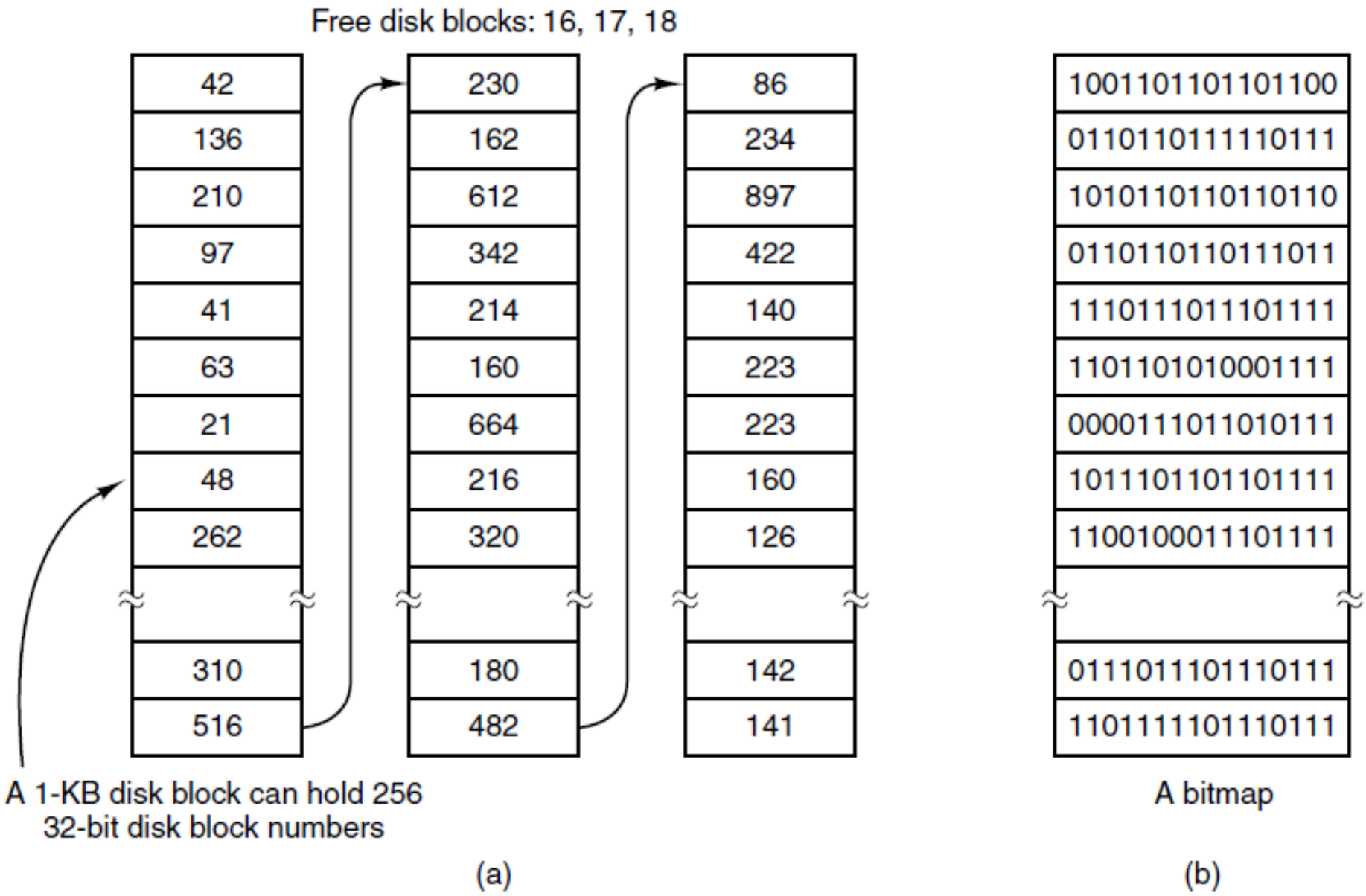
Today

- Files
- Directories
- File System Implementation
  - Implementing Files
  - Implementing Directories
  - Shared Files
  - Free Space

Keeping Track of Free Space

- To write new data to a disk, we need to find some place to put it
- You may recall from memory management two strategies from tracking free space:
  - Store a list somewhere (possibly in the free space)
  - Keep a bitmap of free blocks

Keeping Track of Free Blocks



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