CSE3320
Operating Systems
File Systems and Implementations

Jia Rao

Department of Computer Science and Engineering

http://ranger.uta.edu/~jrao

Recap of Previous Classes

- Processes and threads
  - Abstraction of processor execution
- Memory management
  - Abstraction of physical memory
- File systems
  - Abstraction of persistent data storage on disks

Long-term Information Storage

Three essential requirements for long-term information storage

- Must store a large amount of data
- Information stored must survive the termination of processes using it
- Multiple processes must be able to access the information concurrently

A file is an abstraction of the long-term (persistent) data storage

The part of an OS dealing with files is the file system

What are users' concerns of the file system?

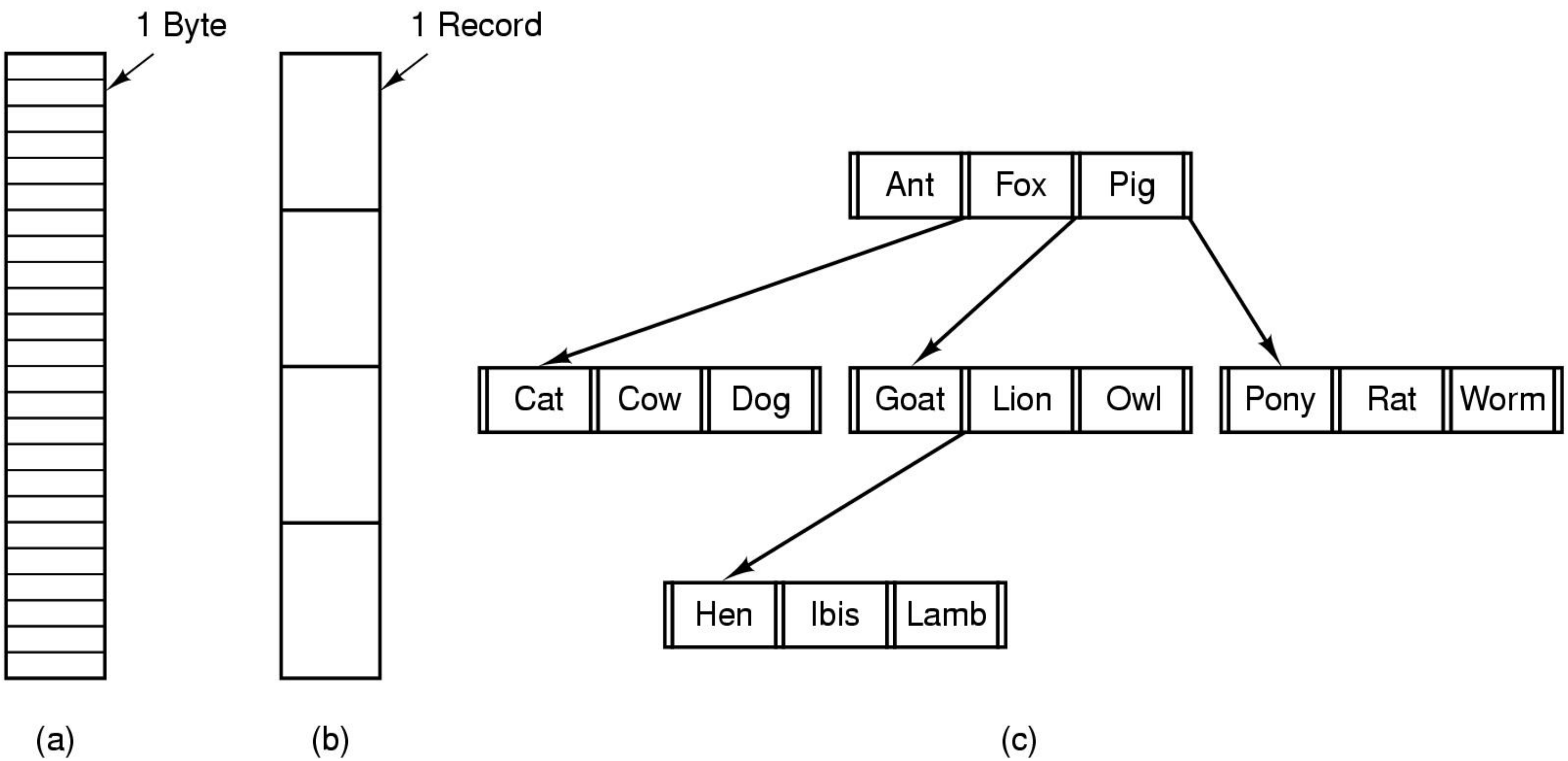
What are implementers' concerns of the file system?

File Naming

- Files are an *abstraction* mechanism
  - two-part file names

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

File Structures



- Three kinds of file structures

Maximum flexibility

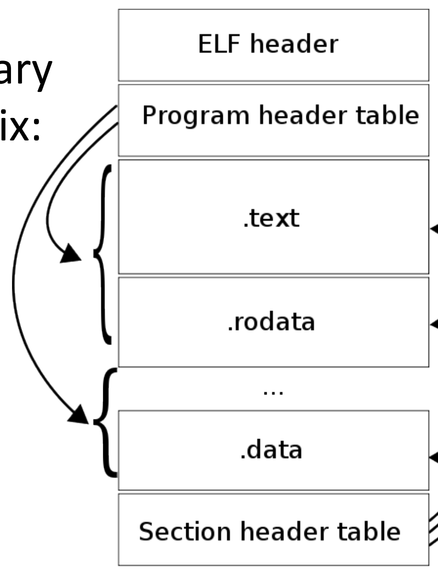
  - Unstructured byte sequence (Unix and WinOS view)
  - Record sequence (early machines' view)
  - Tree (mainframe view)

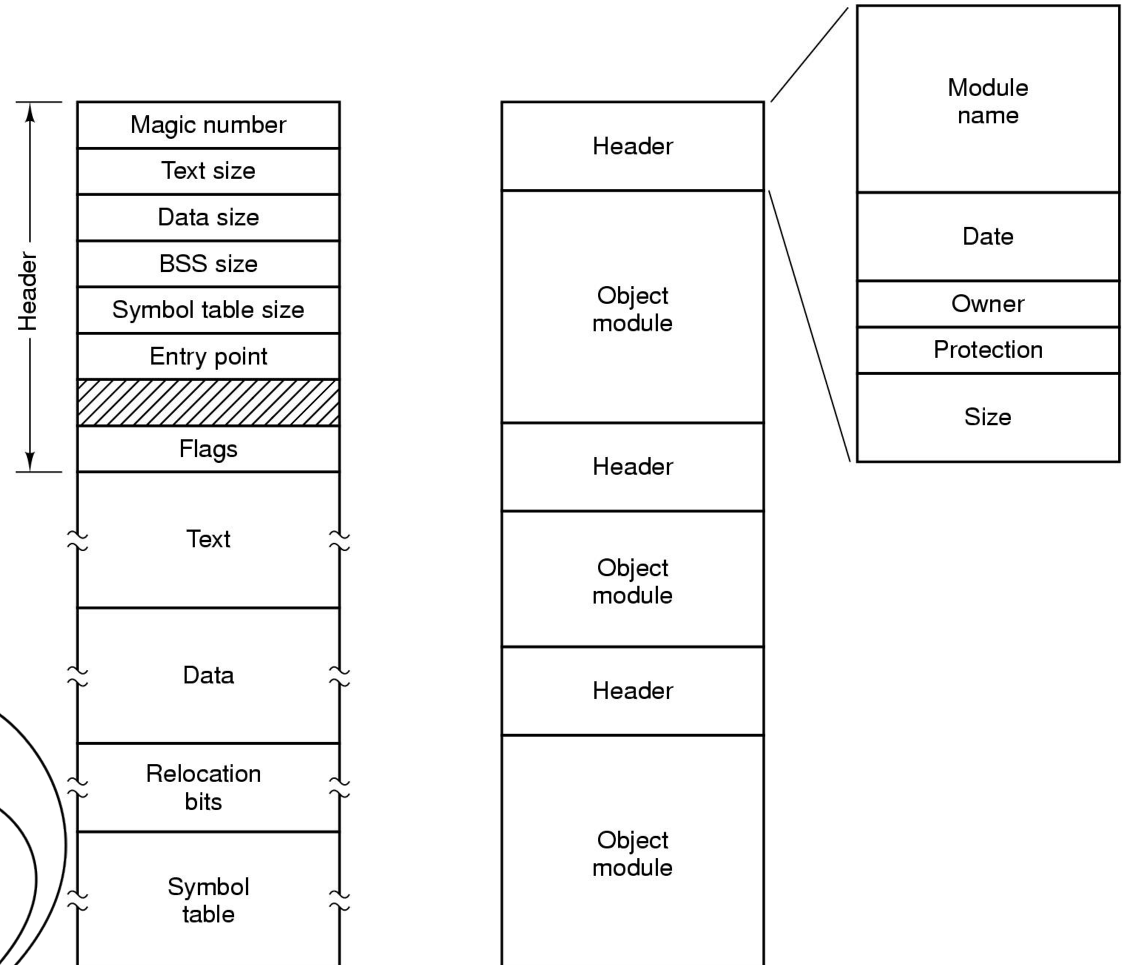
File Types

- Regular files
  - ASCII files or binary files
- Directories
- Character special files
- Block special files

Standard binary Format in Unix: ELF

`readelf`





(a) (b)

File Access

- Sequential access
  - read all bytes/records from the beginning
  - cannot jump around, could rewind or back up
  - convenient when medium was mag tape
- Random access
  - bytes/records read in any order
  - essential for database systems
  - read can be …
    - move file marker (seek), then read or …
    - read and then move file marker

File Attributes

In Linux, us `stat`
to check 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 has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

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

An Example Program Using File System Calls

```
/* 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 */
```

An Example Program Using File System Calls (cont.)

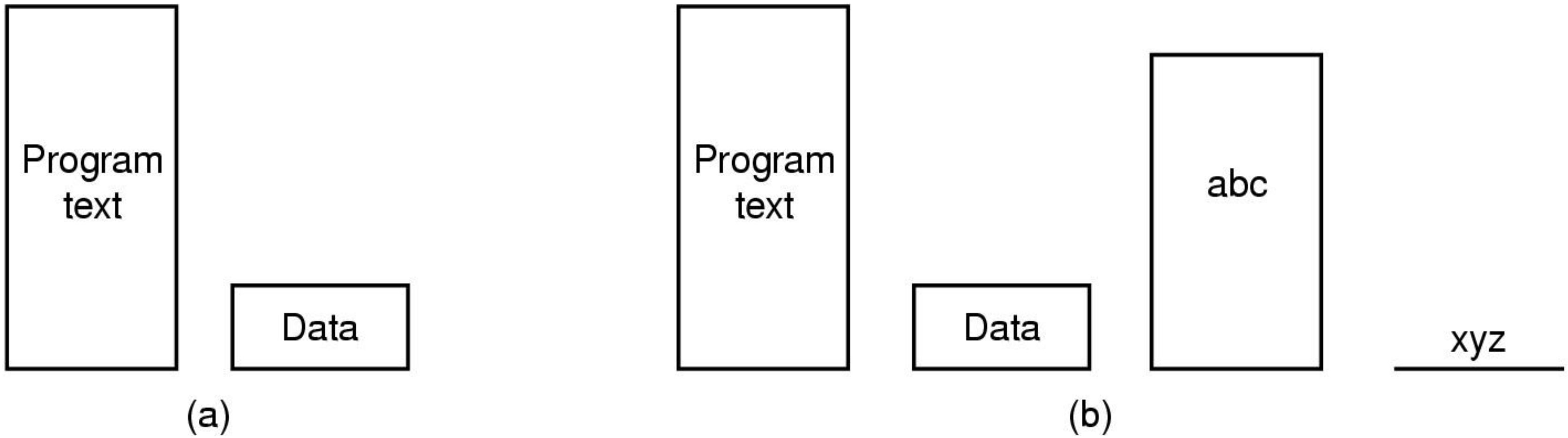
```
/* Open the input file and create the output file */
in_fd = open(argv[1], O_RDONLY);     /* open the source file */
if (in_fd < 0) exit(2);              /* if it cannot be opened, exit */
out_fd = creat(argv[2], OUTPUT_MODE); /* create the destination file */
if (out_fd < 0) exit(3);           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

Memory-Mapped Files

- OS provide a way to map files into the address space of a running process; *map()* and *unmap()*
  - No read or write system calls are needed thereafter



(a) Segmented process before mapping files into its address space

(b) Process after mapping

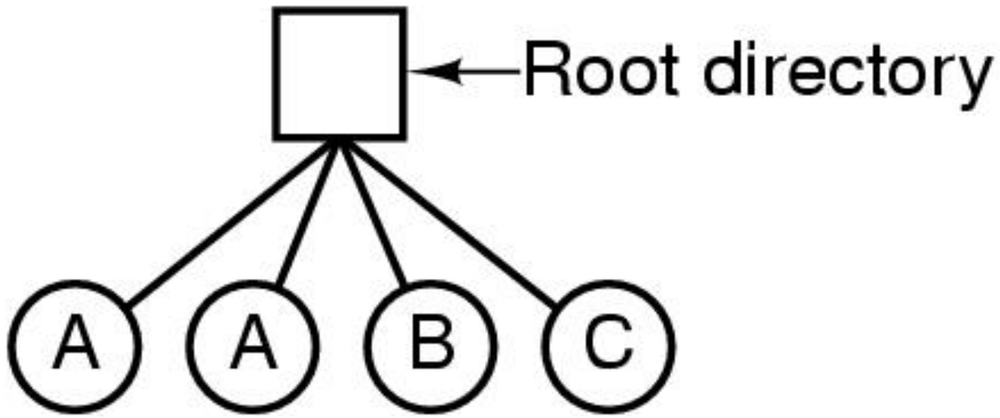
existing file *abc* into one segment

creating new segment for *xyz*

Increased performance

1. Accessing local virtual address is faster
2. Avoiding data copy from kernel to user

Directories: Single-Level Directory Systems

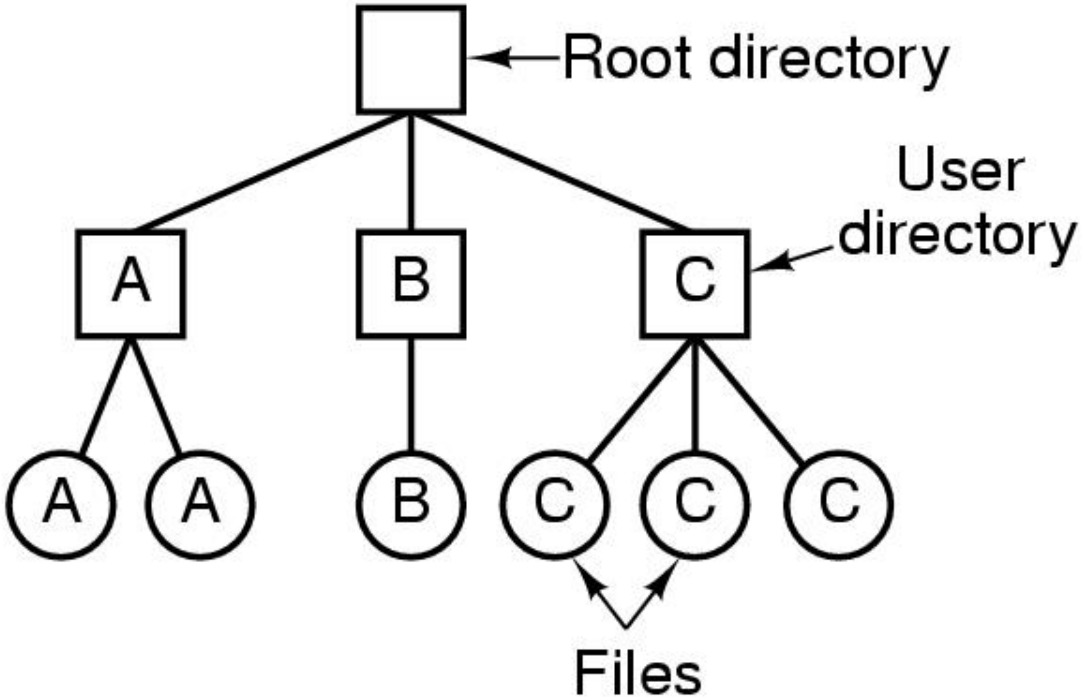


- A single-level directory system is simple for implementation
  - contains 4 files
  - owned by 3 different people, A, B, and C

What is the key problem with the single-level directory systems?

Different users may use the same names for their files

Two-level Directory Systems

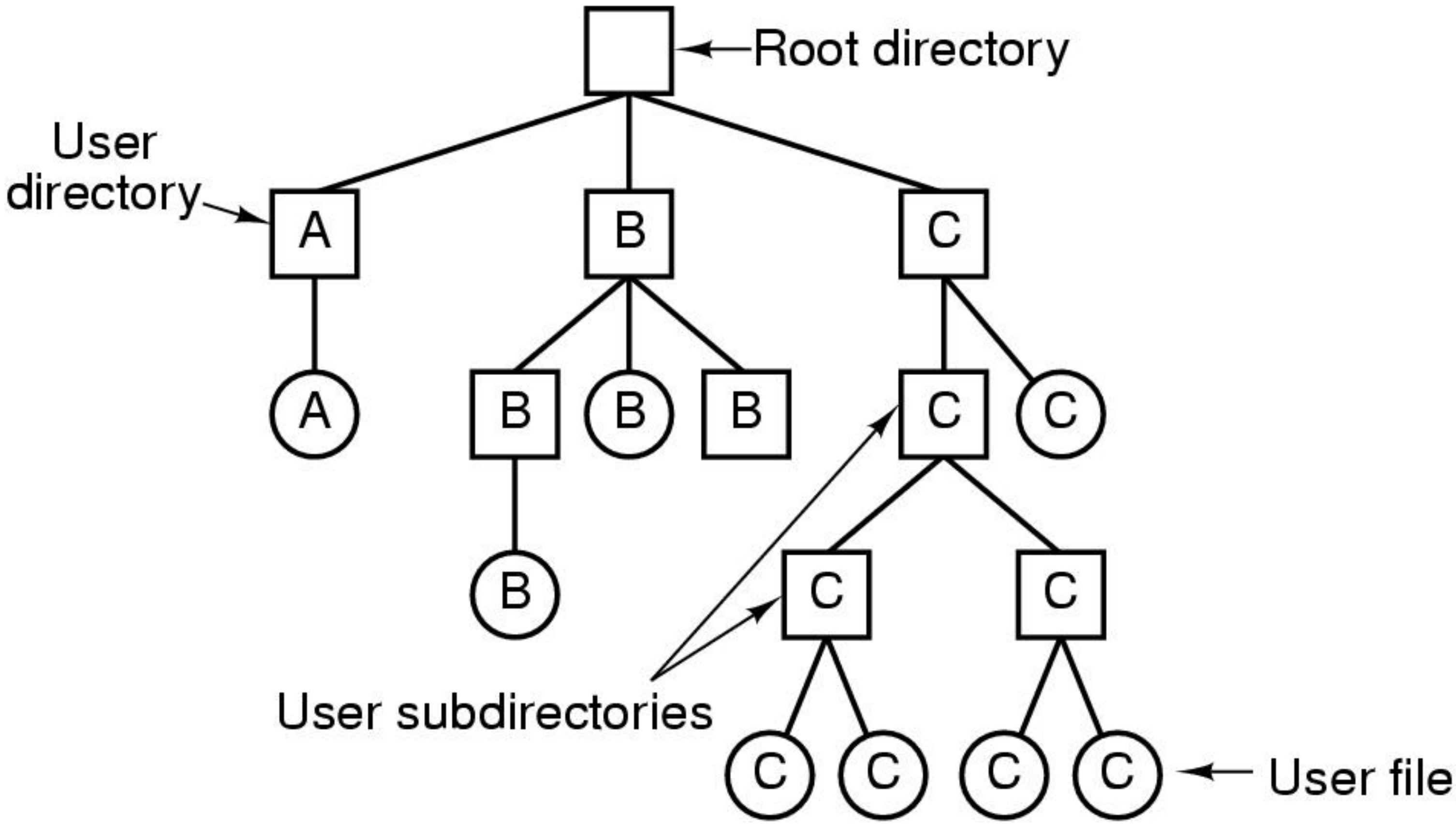


What additional operation required, compared with single-level directory systems?

Login procedure

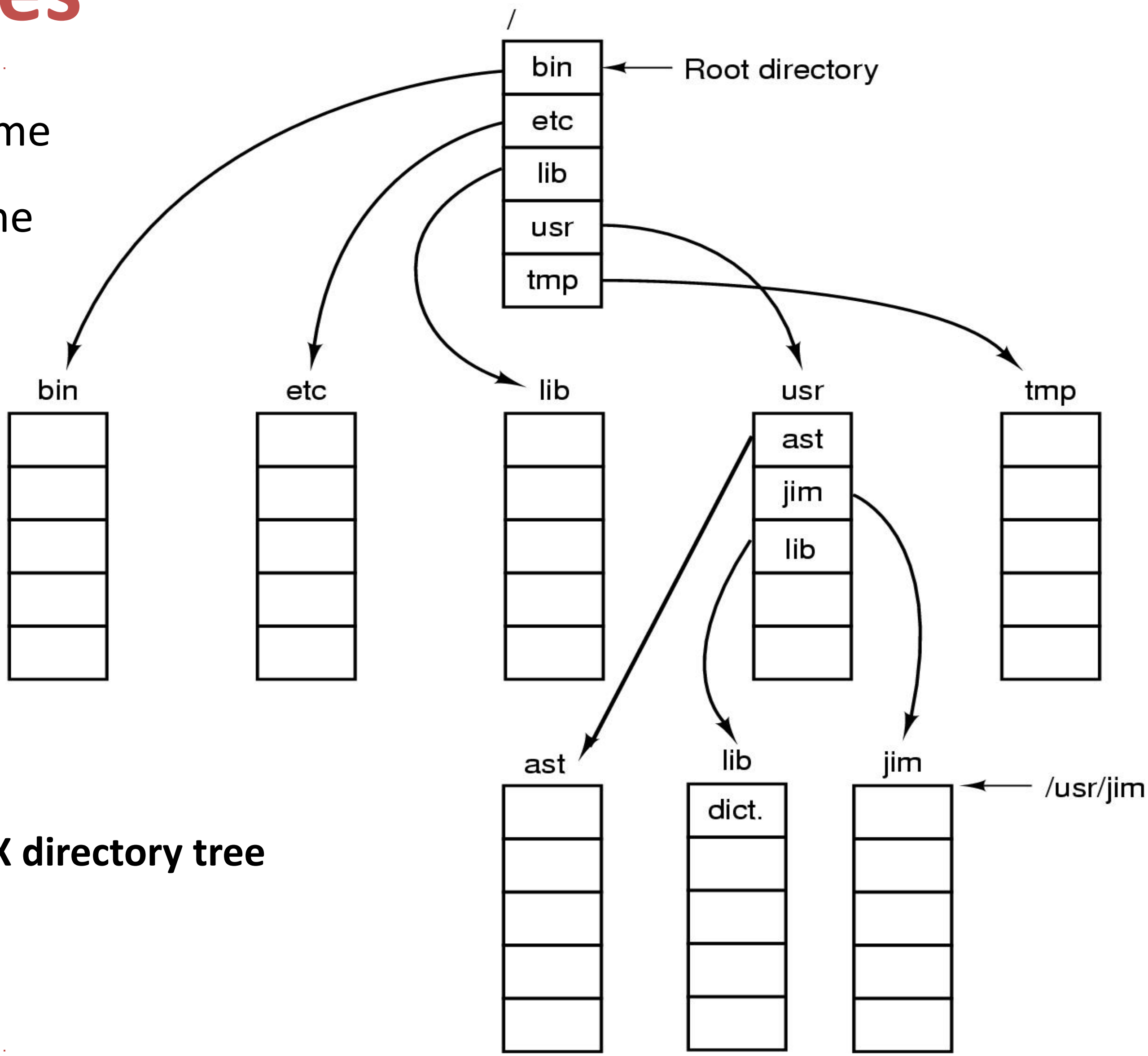
What if a user has many files and wants to group them in a logical way?

Hierarchical Directory Systems



Path Names

- Absolute path name
- Relative path name



A UNIX directory tree

Directory Operations

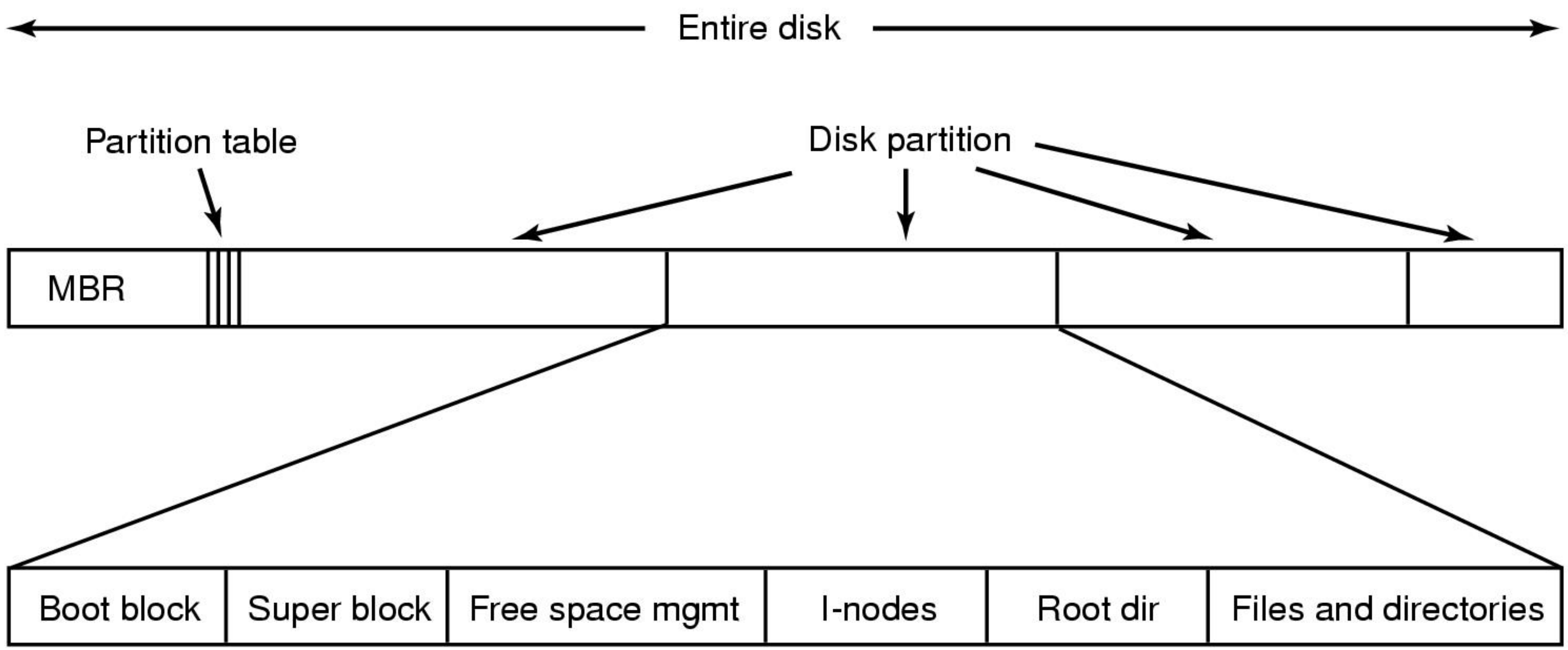
1. Create
2. Delete
3. Opendir
4. Closedir
5. Readdir
6. Rename
7. Link
8. Unlink

What are file system implementers' concerns?

How files & directories stored?
How disk space is managed?
How to make everything work efficiently and reliably?

File System Implementation

- File system layout
  - Most disks can be divided into one or more partitions
  - BIOS →MBR (Master Boot Record)

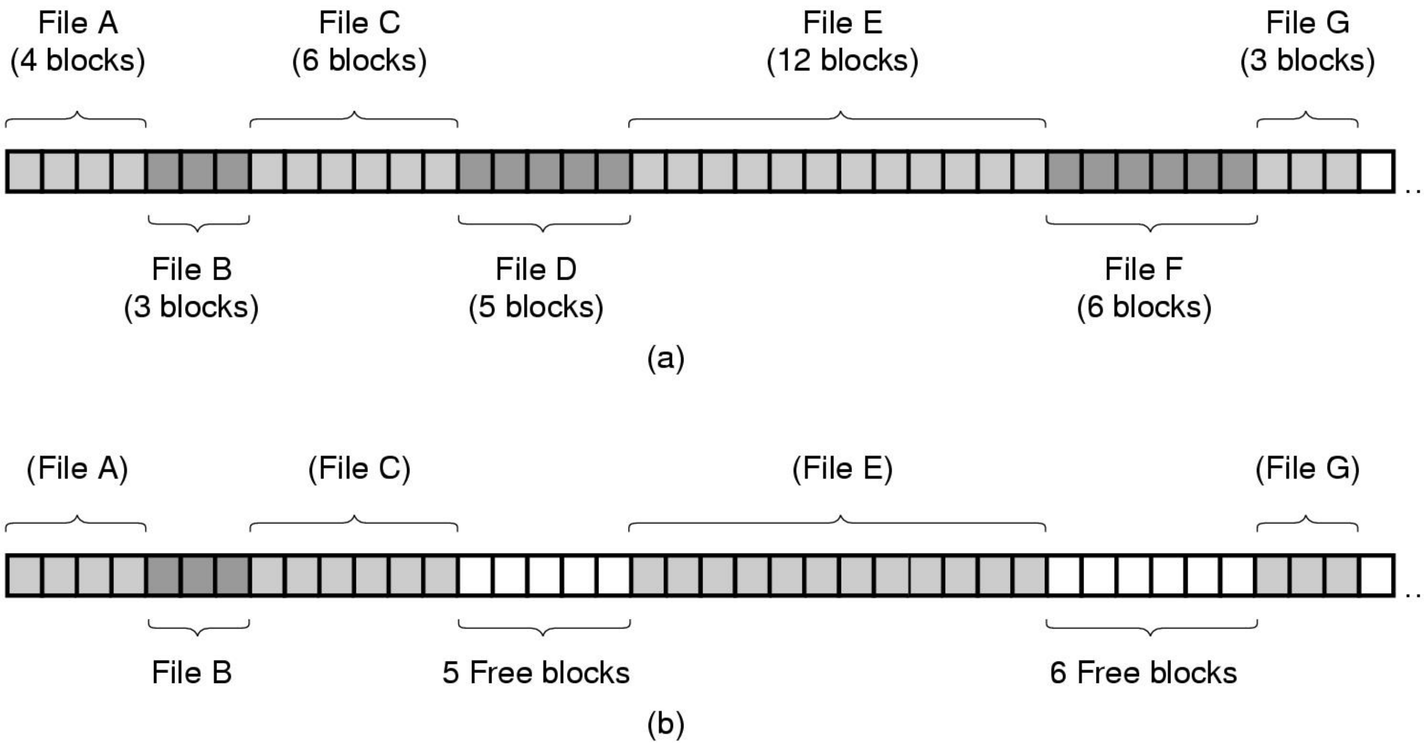


A possible file system layout

How to keep track of which disk blocks go with which file?

Implementing Files (1) – Contiguous Allocation

- Pros: simple addressing and one-seek only reading
- Cons: disk fragmentation (like Swapping)

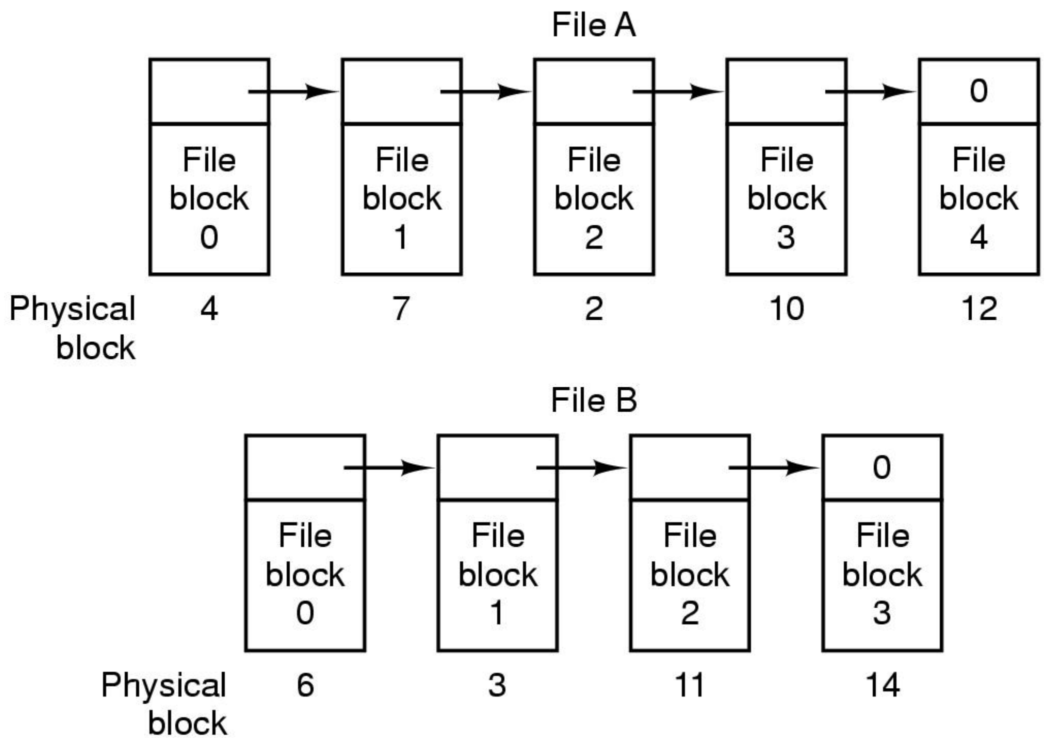


(a) Contiguous *block allocation* of disk space for 7 files

(b) State of the disk after files *D* and *F* have been dynamically removed

Implementing Files (2) – Linked List Allocation

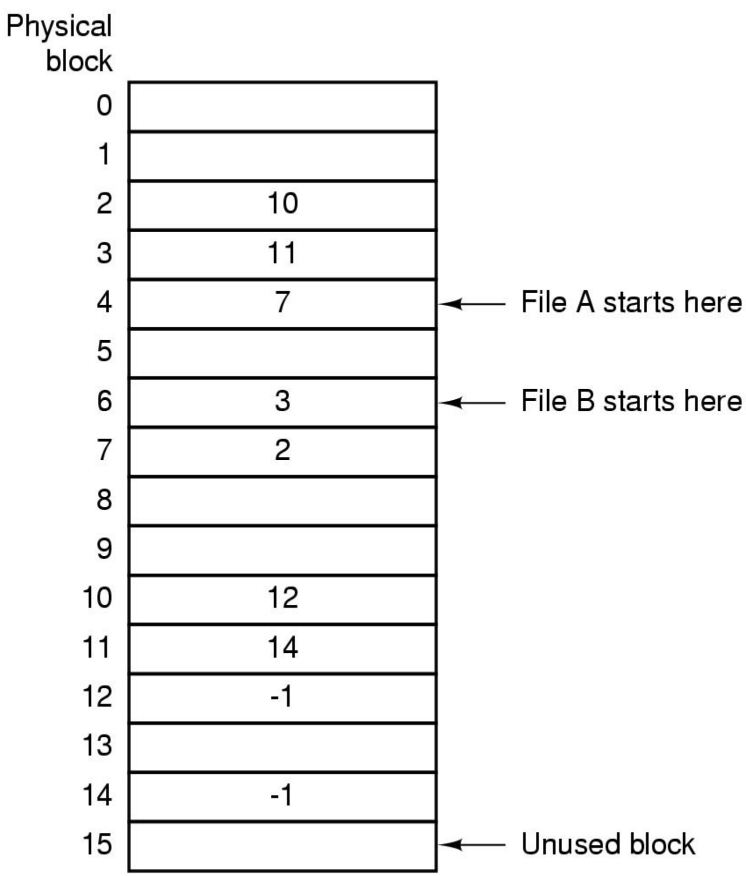
- Keep each file as a linked list of disk blocks
  - Pros: no space is lost due to disk fragmentation
  - Cons: how about random access?



Storing a file as a linked list of disk blocks

Implementing Files (3) – FAT (File Allocation Table)

- FAT: a table in memory with the pointer word of each disk block
  - High utilization + easy random access, but too *“FAT”* maybe?



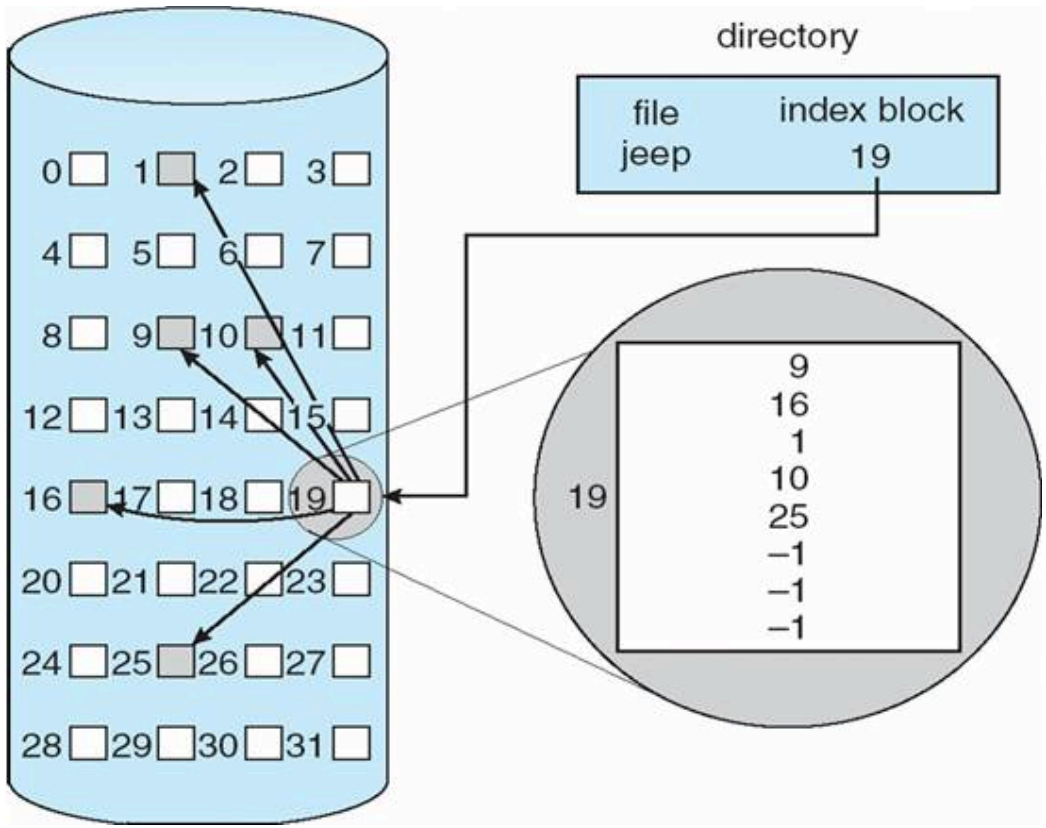
Consider:

A 20 GB disk
1 KB block size
Each entry 3 B

How much space for a FAT?
How about paging it?

Implementing Files (4) – Indexed Allocation

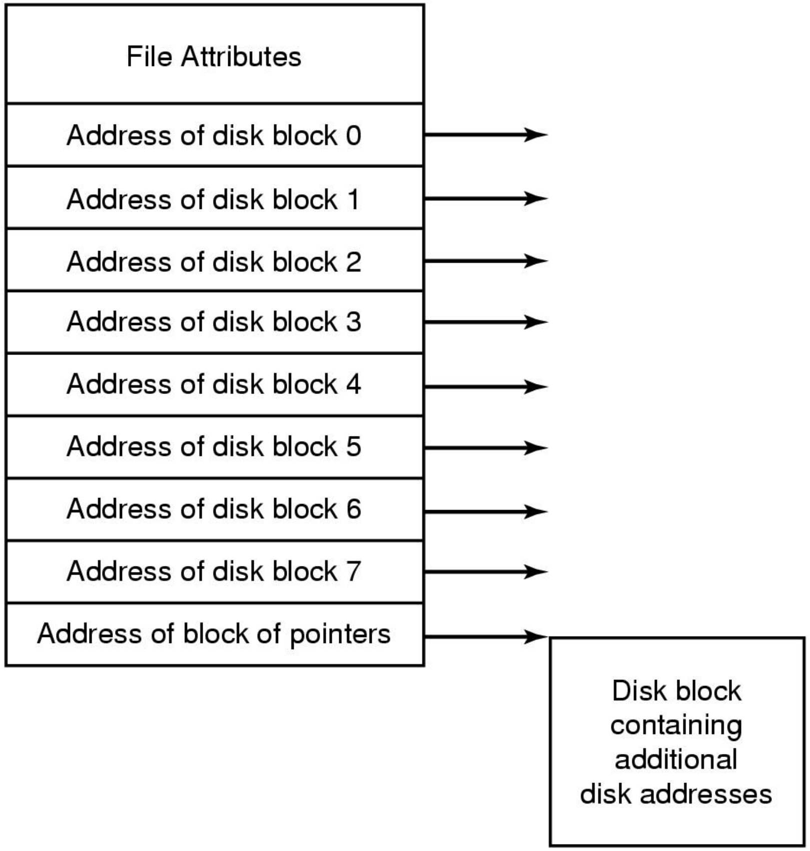
- Grouping all pointers together in a index block
- Each file has its own index block on disk
- it contains an array of disk addrs



Implementing Files (4) – Indexed Allocation Example: I-node

- i-node: a data structure listing the attributes and disk addresses of the file's blocks; in memory when the corresponding file is open

Why i-node scheme requires much less space than FAT?



Implementing Files (5) – Summary

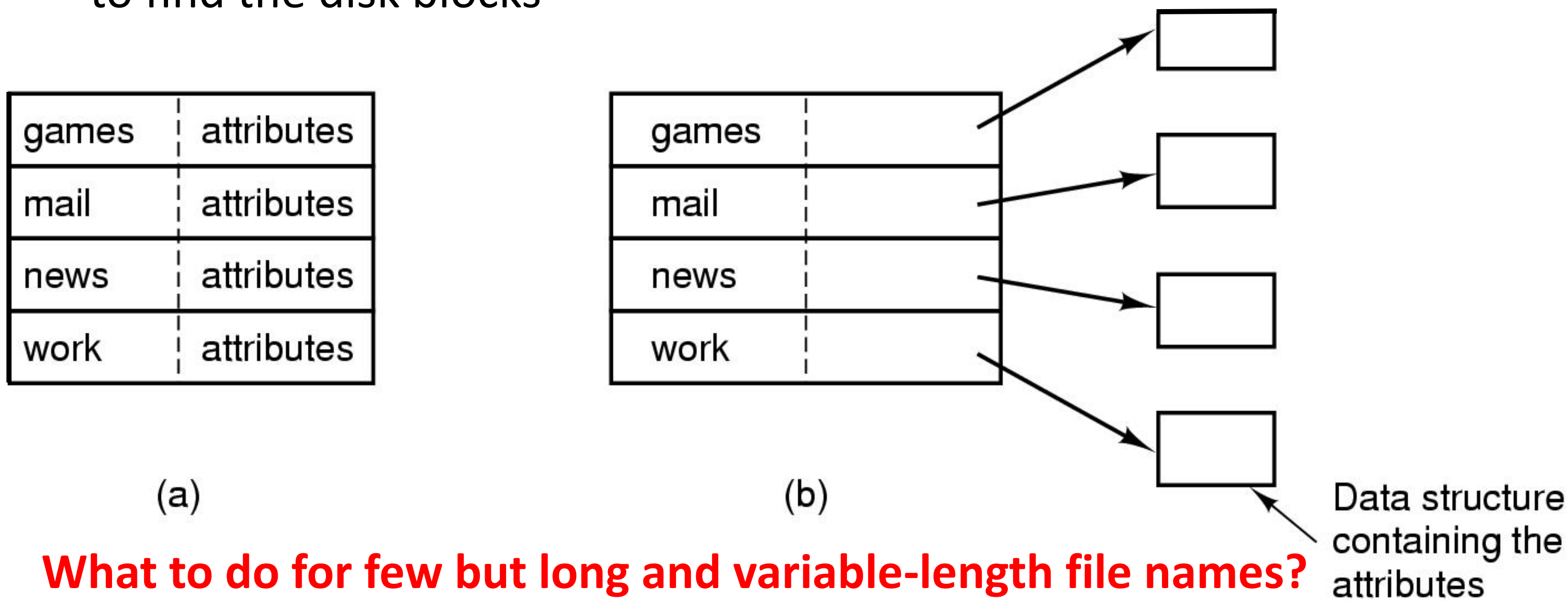
- How to find the disk blocks of a file?
  - Contiguous allocation: the disk address of the entire file
  - Linked list & FAT: the number of the first block
  - i-node: the number of the i-node

Who provides the information above?

The directory entry (based on the path name)

Implementing Directories (1)

- The *directory entry*, based on the path name, provides the information to find the disk blocks



What to do for few but long and variable-length file names?

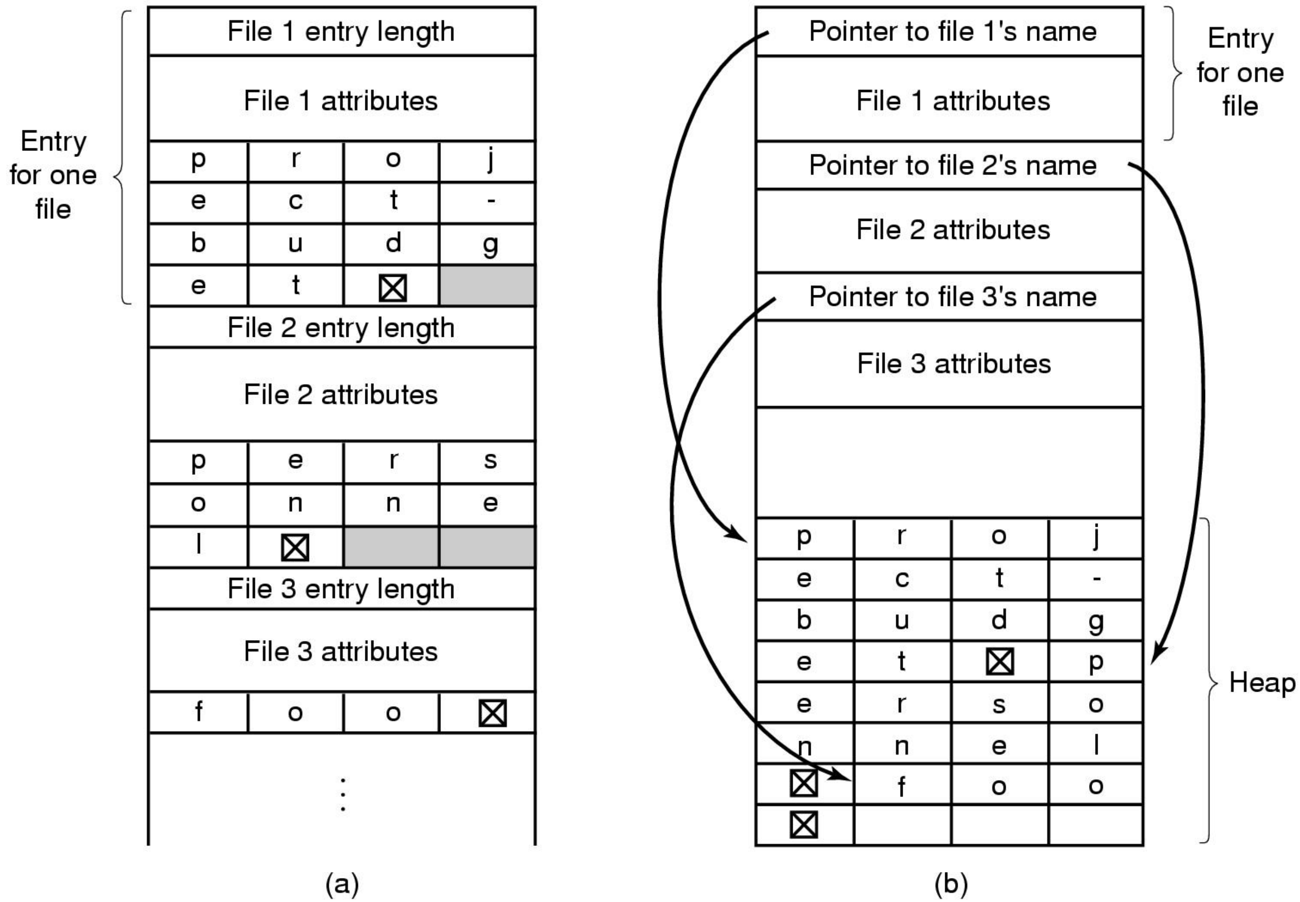
(a) A simple directory (MS-DOS/Windows)

Fixed-size entries

File names, attributes, and disk addresses in directory entry

(b) Directory (UNIX); each entry just refers to an i-node, *i-number* returned

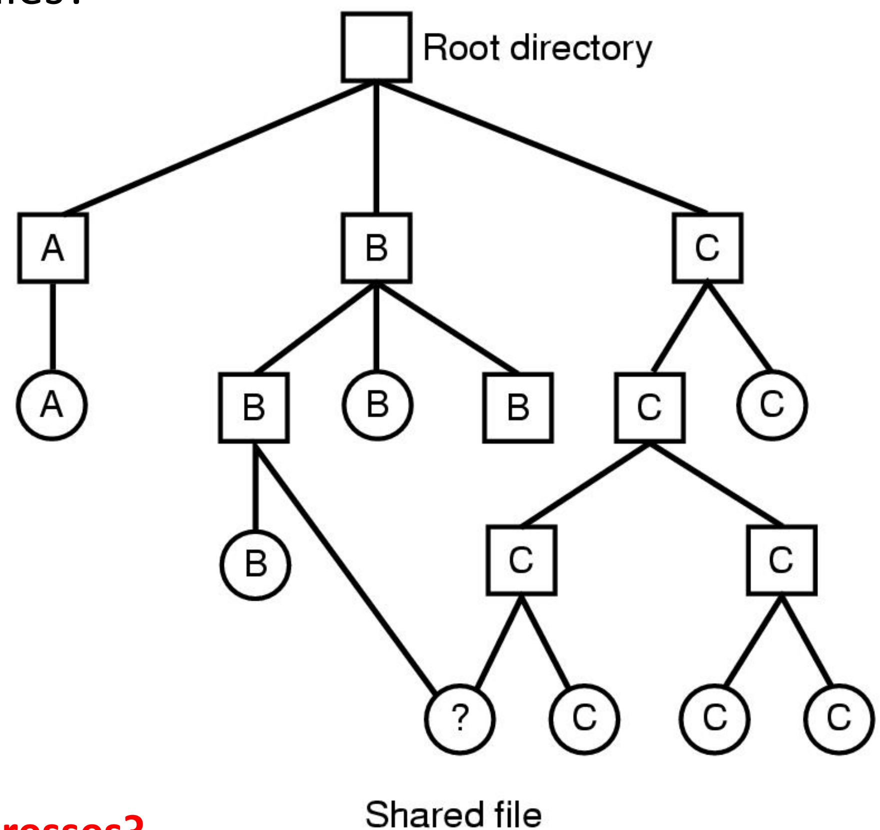
Implementing Directories (2)



- Two ways of handling long and variable-length file names in directory
  (a) In-line: compaction and page fault. (b) In a heap: page fault

Shared Files

- How to let multiple users share files?

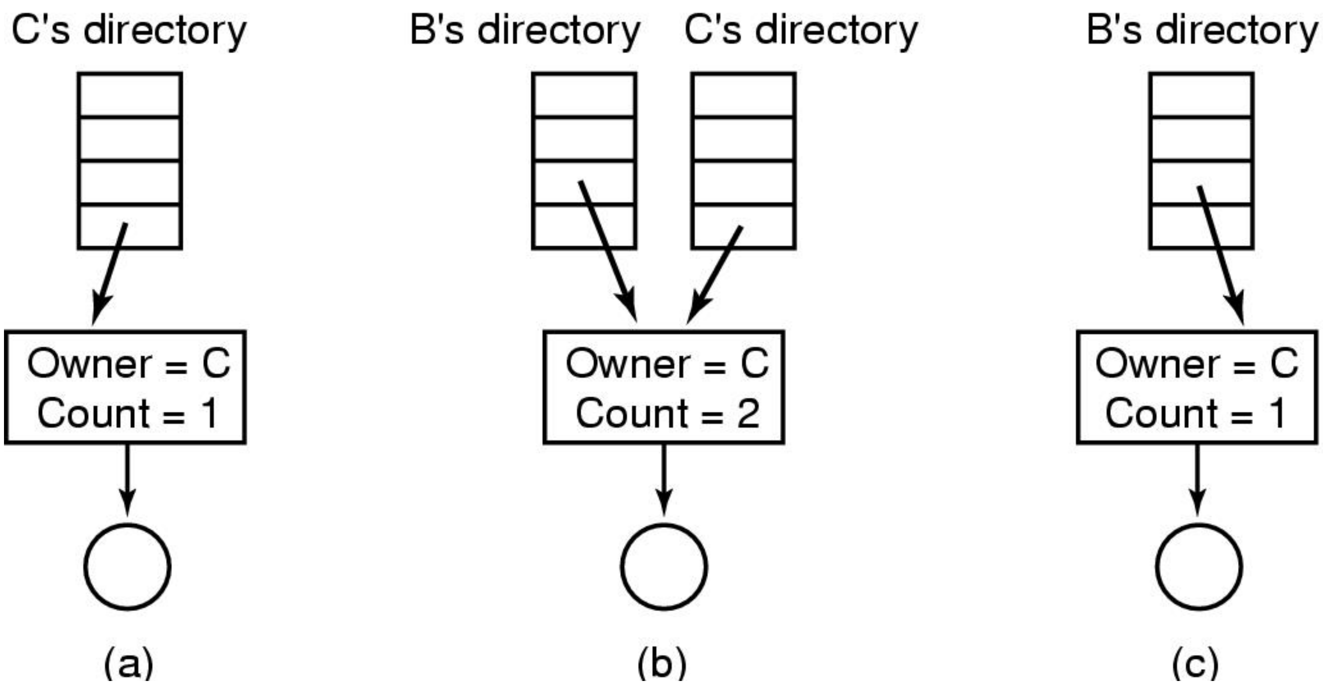


What if directories contain the disk addresses?

File system containing a shared file

Shared Files in UNIX

- UNIX utilizes i-node' data structure
  - What if a file is removed by the owner?



(a) Situation prior to linking; (b) After the link is created

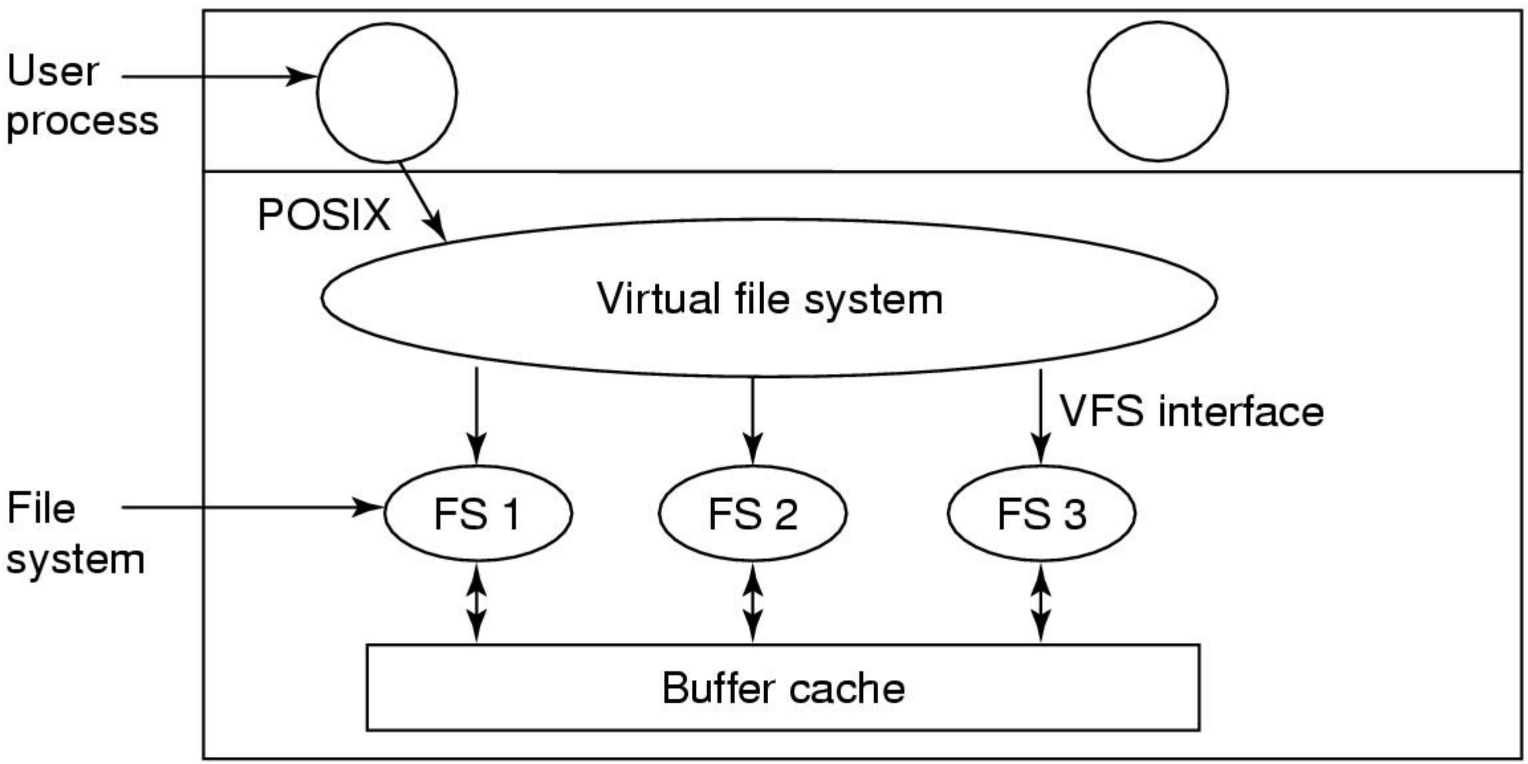
(c) After the original owner removes the file

Shared Files – Symbolic Linking

- A new file, created with type LINK, enters B's directory
  - The file contains just the path name of the linked file
  - Con: extra overhead with each file access, parsing
  - Pro 1: Only when the owner removes the file, it is destroyed
    - Removing a symbolic link does not affect the file
  - Pro 2: networked file systems

Virtual File System (VFS)

- A virtual file system is an abstract layer with common functionalities supported by all the underlying concrete file systems



Summary

- Implementing files
  - Contiguous allocation
  - Linked allocation
  - FAT
  - Indexed allocation (I-node)
- Implementing directories
- Sharing files
- File system management and optimizations
  - Free blocks, consistency, performance …