Files and File Systems

CS 416: Operating Systems Design

Department of Computer Science
Rutgers University

http://www.cs.rutgers.edu/~vinodg/teaching/416/

File Concept

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

File Structure

- None - sequence of words, bytes
- Simple record structure
  - Lines
  - Fixed length
  - Variable length
- Complex Structures
  - Formatted document
  - Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters
- Who decides:
  - Operating system
  - Program

File Attributes

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

File Operations

- File is an **abstract data type**
- **Create**
- **Write**
- **Read**
- **Reposition within file**
- **Delete**
- **Truncate**
- *Open*(F_i) – search the directory structure on disk for entry F_i, and move the content of entry to memory
- *Close* (F_i) – move the content of entry F_i in memory to directory structure on disk

Open Files

- Several pieces of data are needed to manage open files:
  - File pointer: pointer to last read/write location, per process that has the file open
  - File-open count: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
  - Disk location of the file: cache of data access information
  - Access rights: per-process access mode information

Open File Locking

- Provided by some operating systems and file systems
- Mediates access to a file
- Mandatory or advisory:
  - **Mandatory** – access is denied depending on locks held and requested
  - **Advisory** – processes can find status of locks and decide what to do

File Locking Example – Java API

```
import java.io.*;
import java.nio.channels.*;
public class LockingExample {
    public static final boolean EXCLUSIVE = false;
    public static final boolean SHARED = true;
    public static void main(String arsg[]) throws IOException {
        FileLock sharedLock = null;
        FileLock exclusiveLock = null;
        try {
            RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");
            // get the channel for the file
            FileChannel ch = raf.getChannel();
            // this locks the first half of the file - exclusive
            exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);
            /** Now modify the data . . . */
            // release the lock
            exclusiveLock.release();
```

File Locking Example – Java API (cont)

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

File Types – Name, Extension

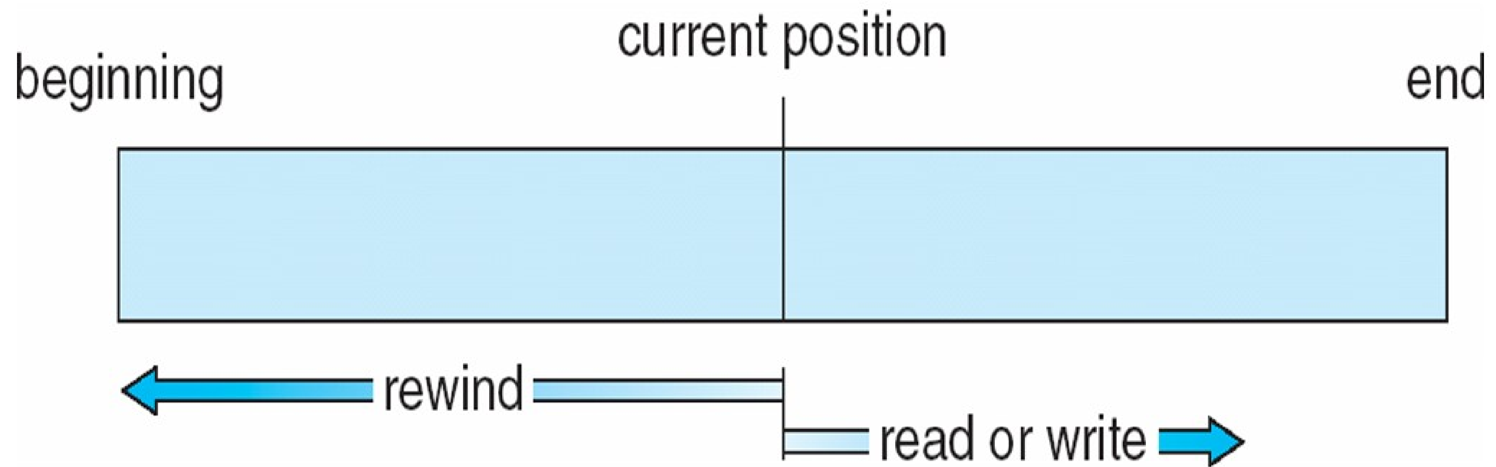
file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual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

Access Methods

- **Sequential Access**
  - read next
  - write next
  - reset
  - no read after last write
    - (rewrite)
- **Direct Access**
  - read n
  - write n
  - position to n
    - read next
    - write next
  - rewrite n

n = relative block number

Sequential-access File

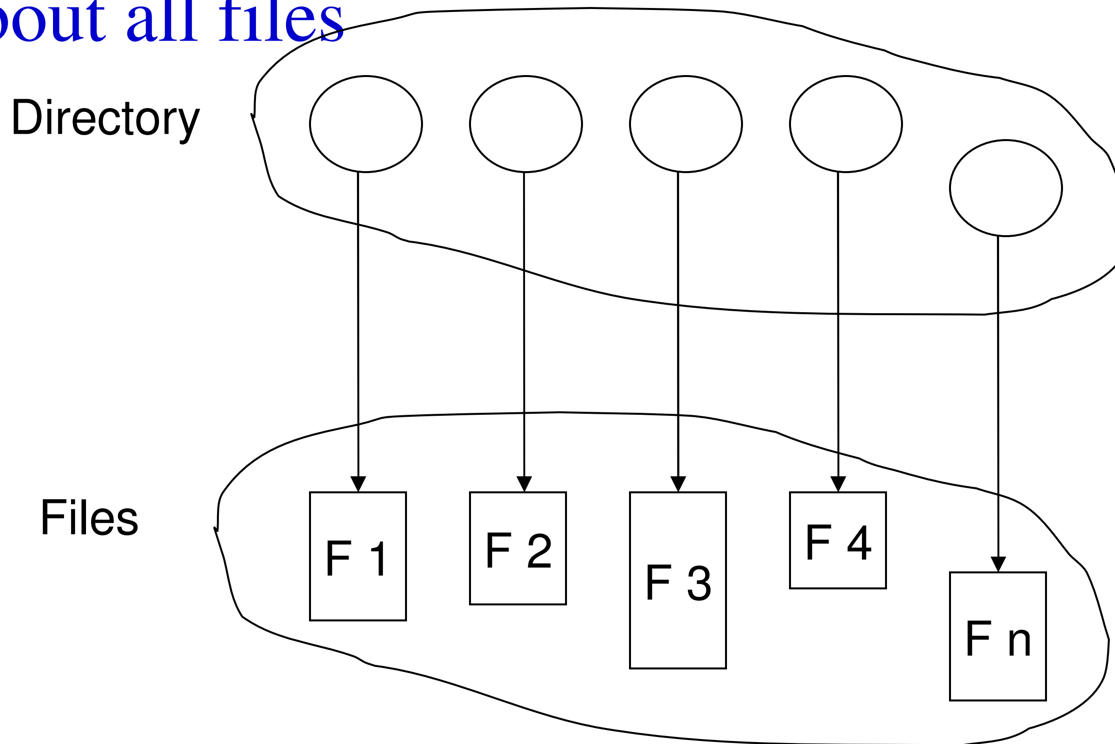


Simulation of Sequential Access on Direct-access File

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

Directory Structure

- A collection of nodes containing information about all files

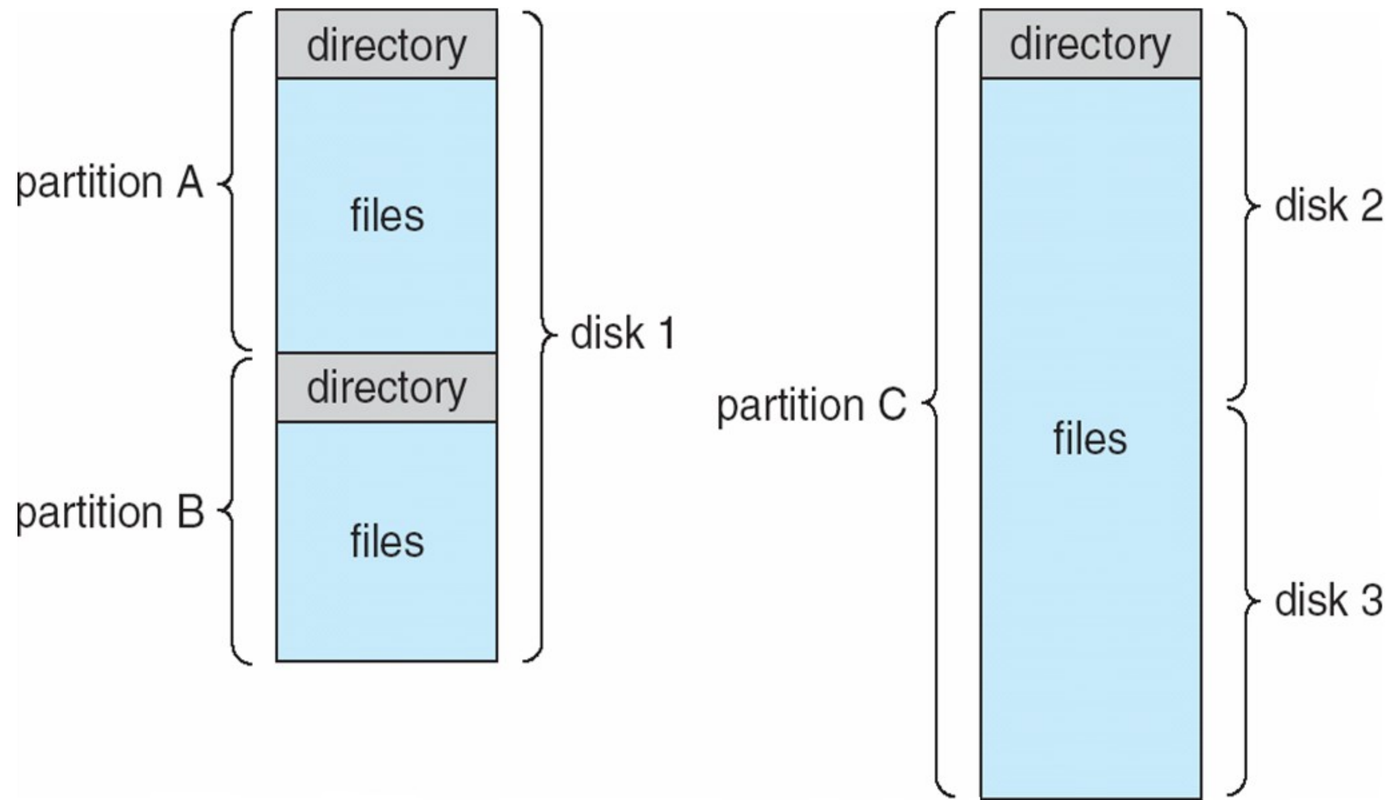


Both the directory structure and the files reside on disk
Backups of these two structures are kept on tapes

Disk Structure

- Disk can be subdivided into partitions
- Disks or partitions can be RAID protected against failure
- Disk or partition can be used raw – without a file system, or formatted with a file system
- Partitions also known as minidisks, slices
- Entity containing file system known as a volume
- Each volume containing file system also tracks that file system's info in device directory or volume table of contents
- As well as general-purpose file systems there are many special-purpose file systems, frequently all within the

A Typical File-system Organization



Operations Performed on Directory

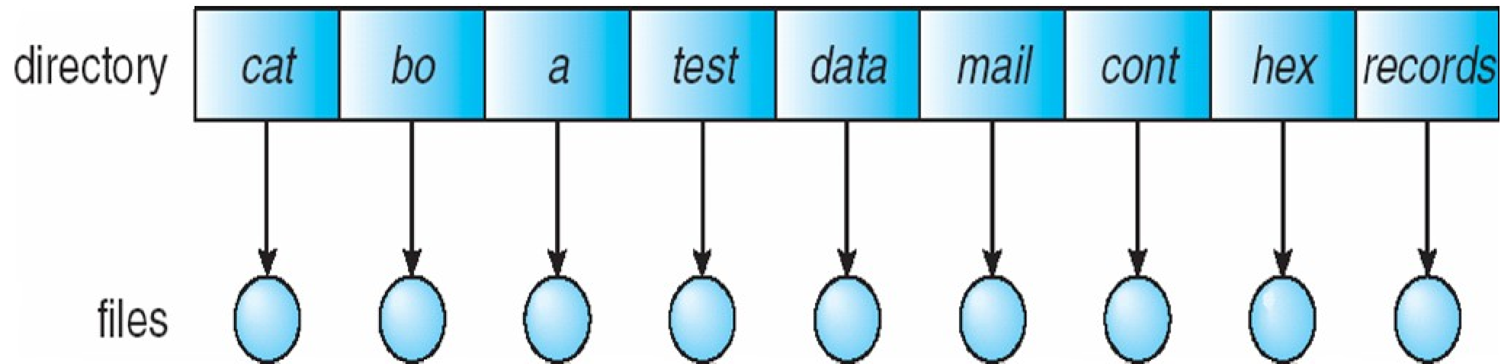
- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

Organize the Directory (Logically) to Obtain

- Efficiency – locating a file quickly
- Naming – convenient to users
  - Two users can have same name for different files
  - The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, …)

Single-Level Directory

- A single directory for all users

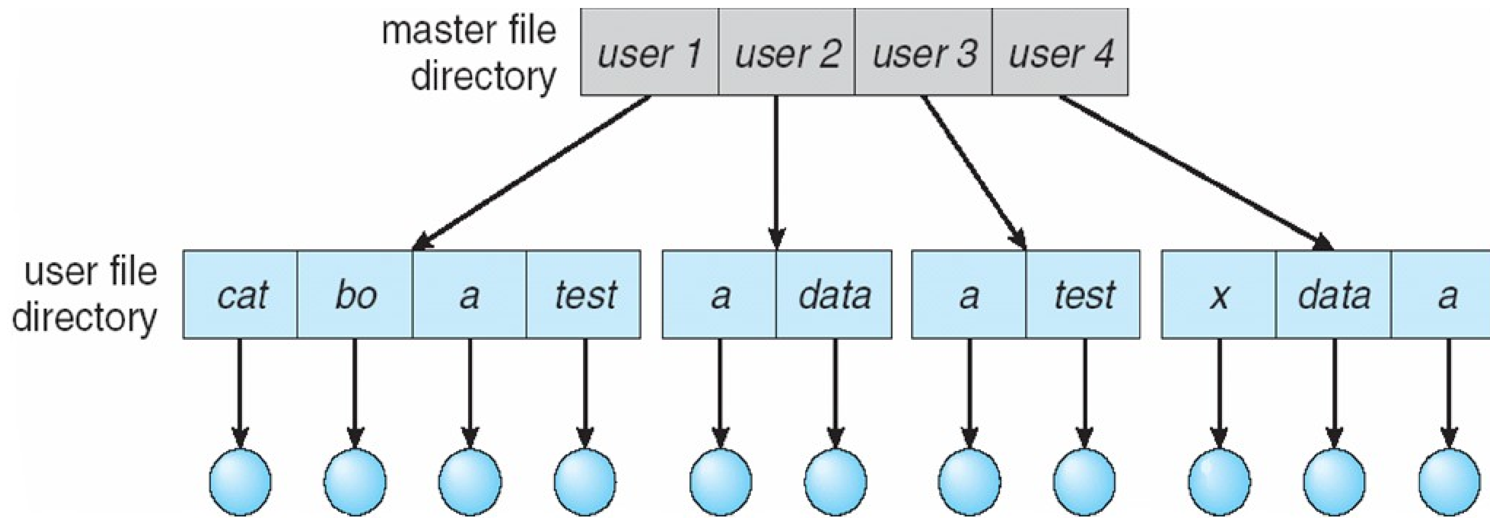


Naming problem

Grouping problem

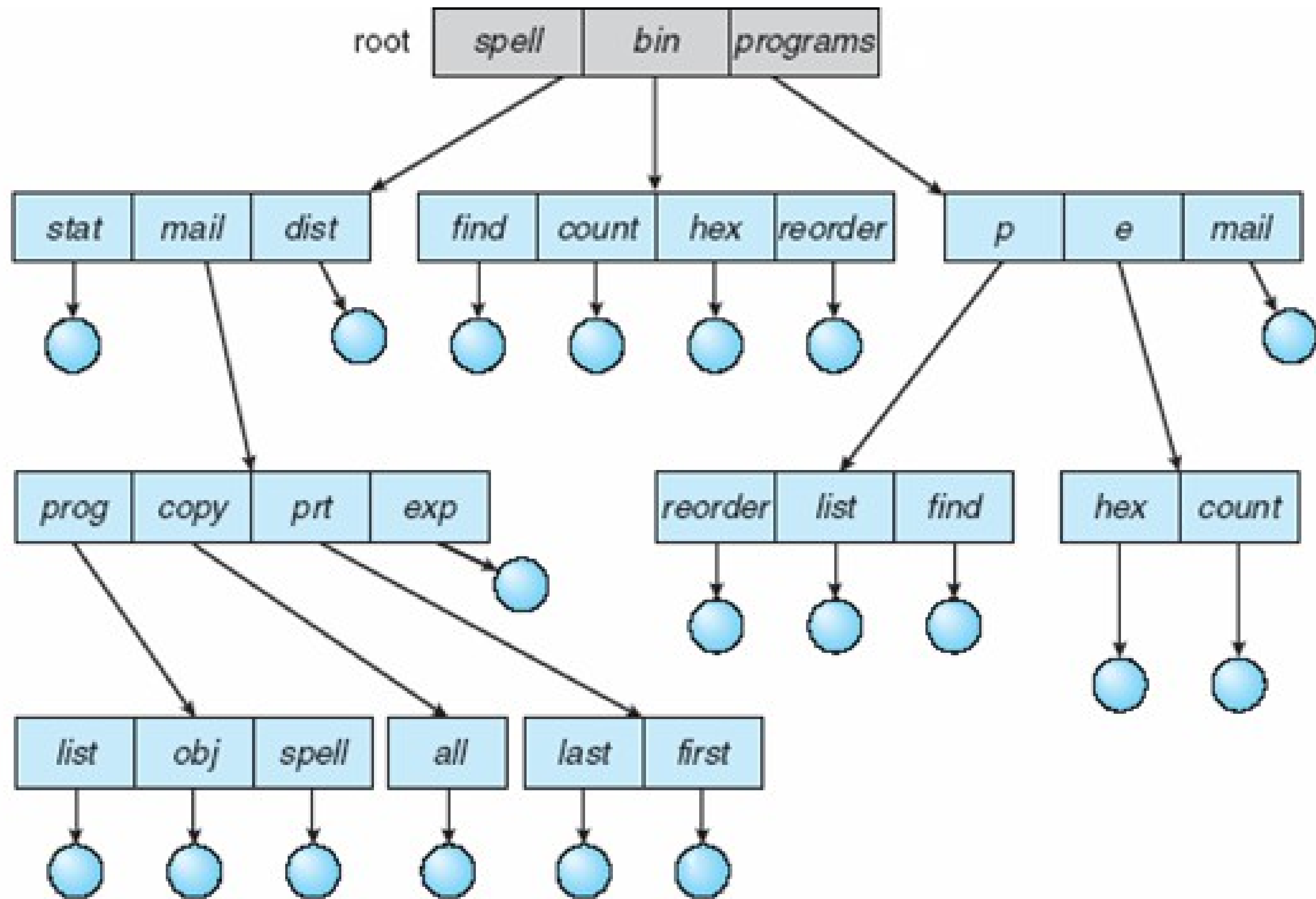
Two-Level Directory

- Separate directory for each user



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

Tree-Structured Directories



Tree-Structured Directories (Cont)

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

Tree-Structured Directories (Cont)

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

  rm <file-name>

- Creating a new subdirectory is done in current directory

  mkdir <dir-name>

  Example: if in c[urrent] directory /mail

  mkdir count

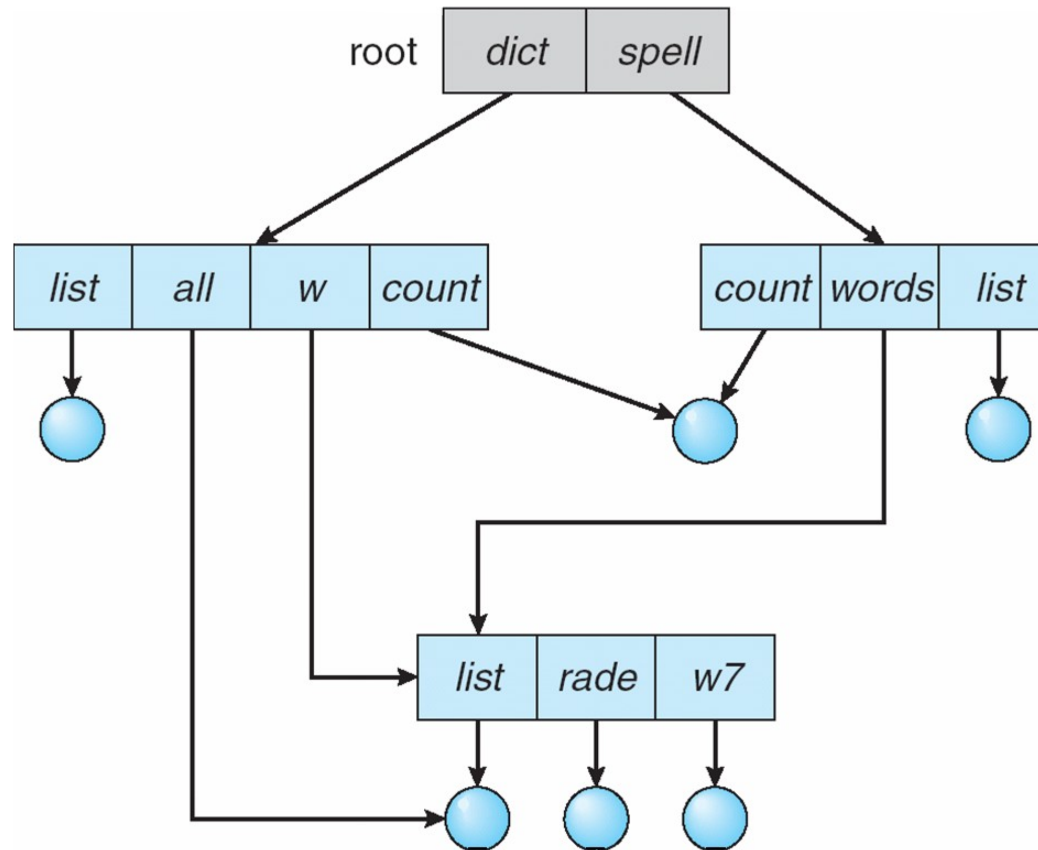
mail

prog | copy | prt | exp | count

Deleting "mail" ⇒ deleting the entire subtree rooted by "mail"

Acyclic-Graph Directories

- Have shared subdirectories and files



Acyclic-Graph Directories (Cont.)

- Two different names (aliasing)

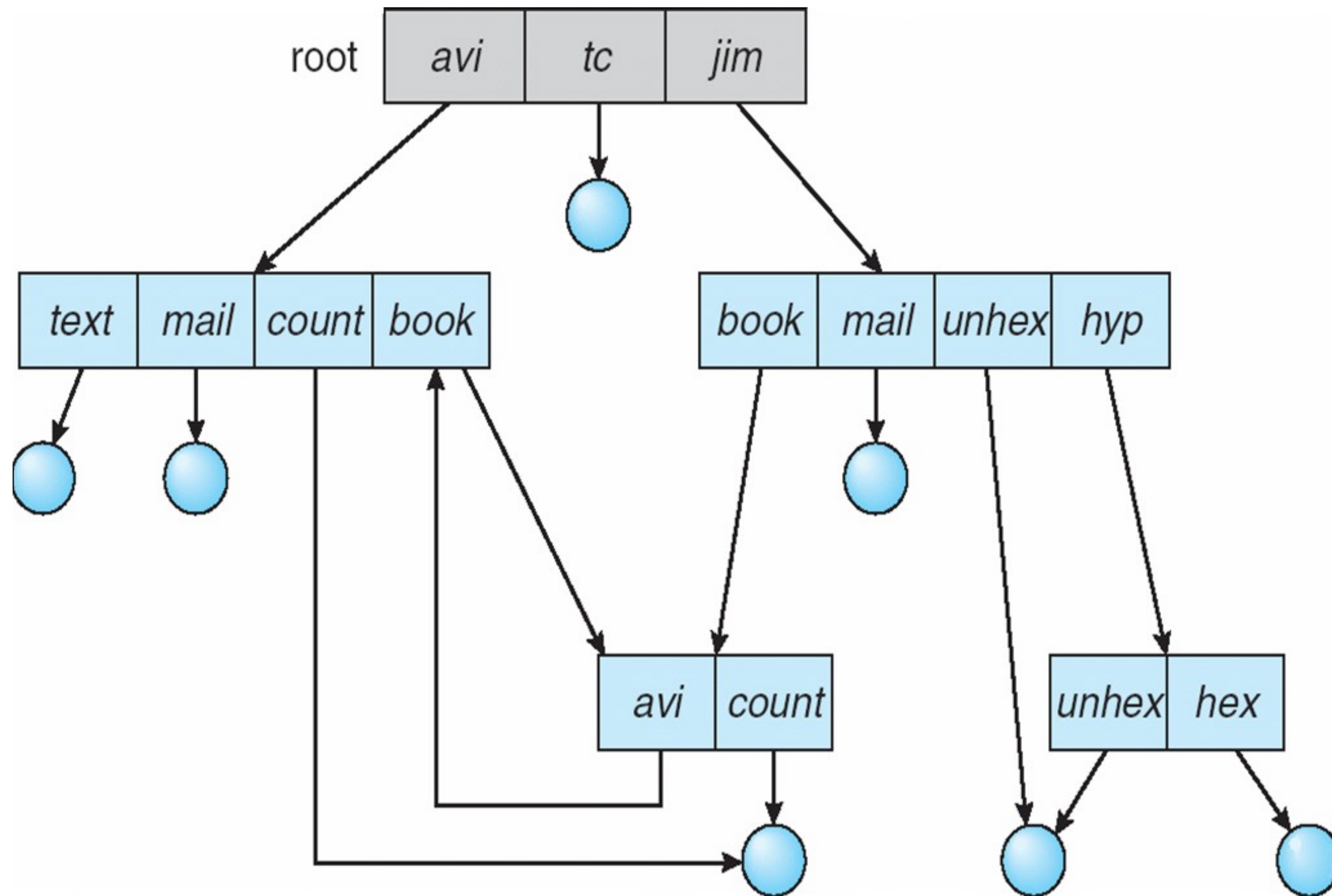
- If *dict* deletes *list* ⇒ dangling pointer

  Solutions:

  - Backpointers, so we can delete all pointers
    Variable size records a problem
  - Backpointers using a daisy chain organization
  - Entry-hold-count solution

- New directory entry type
  - **Link** – another name (pointer) to an existing file
  - **Resolve the link** – follow pointer to locate the file

General Graph Directory



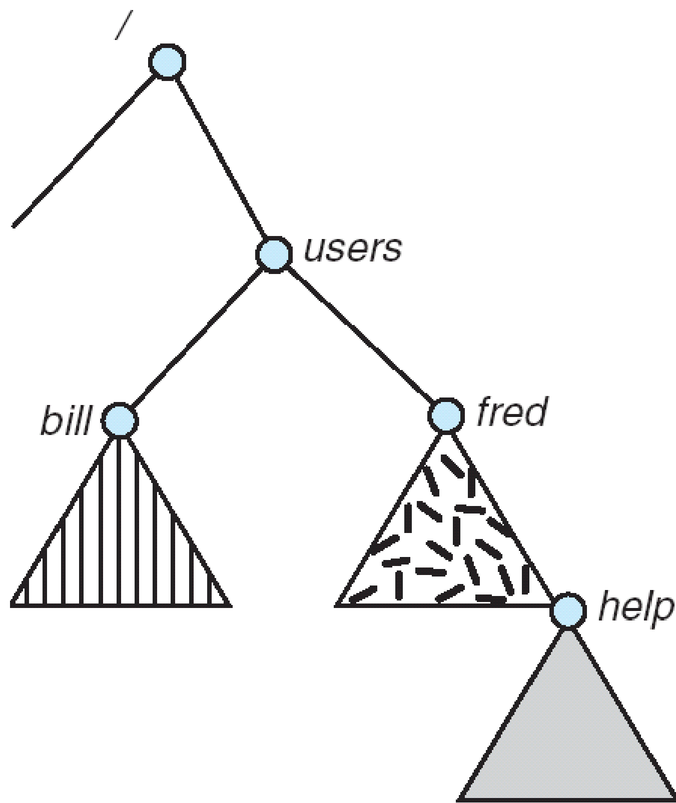
General Graph Directory (Cont.)

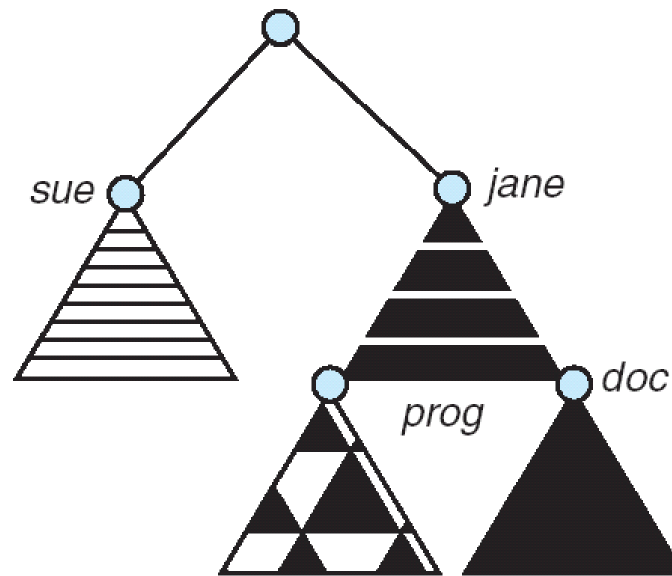
- How do we guarantee no cycles?
  - Allow only links to file not subdirectories
  - Garbage collection
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK

File System Mounting

- A file system must be **mounted** before it can be accessed
- A unmounted file system is mounted at a **mount point**

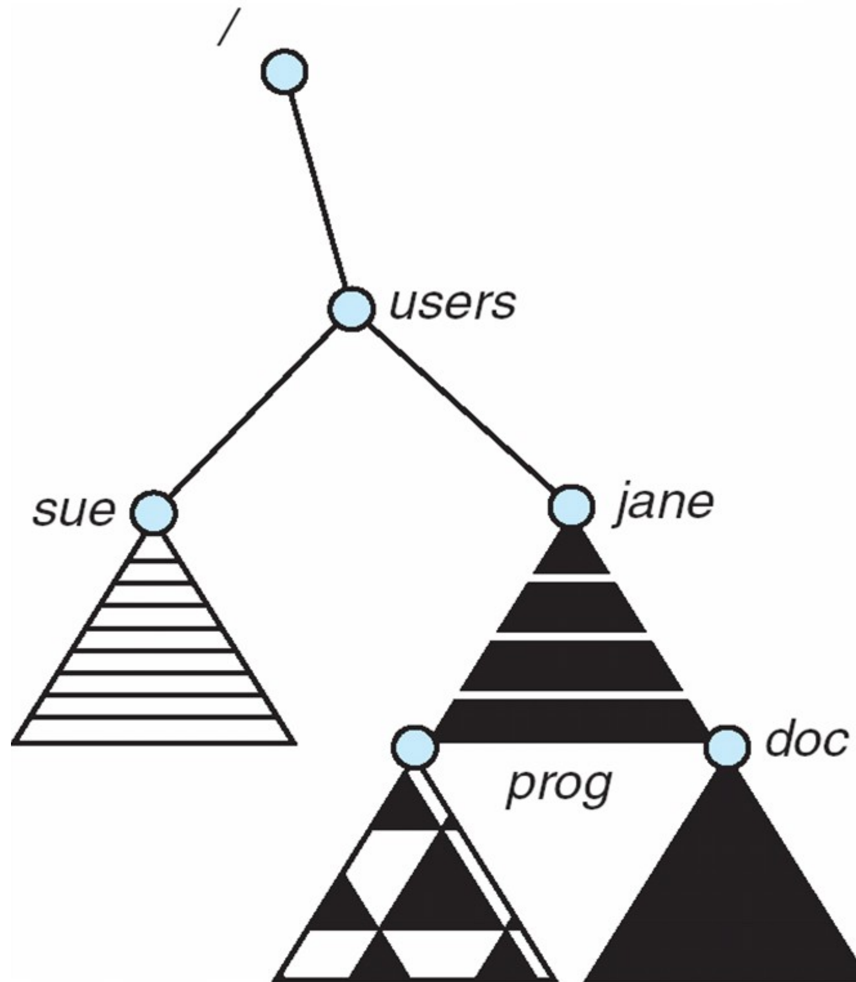
(a) Existing. (b) Unmounted Partition





(a) (b)

Mount Point



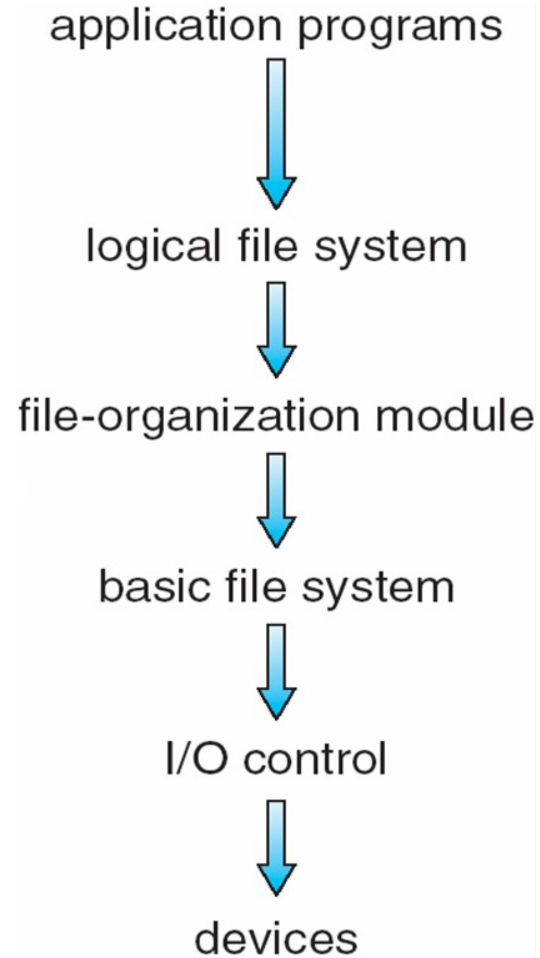
File System

- File system is an abstraction of the disk
  - File ➜ Tracks/sectors
  - File Control Block stores mapping info (+ protection, timestamps, size, etc)
  - To a user process
    - A file looks like a contiguous block of bytes (Unix)
    - A file system provides a coherent view of a group of files
    - A file system provides protection
- API: create, open, delete, read, write files
- Performance: throughput vs. response time
- Reliability: minimize the potential for lost or destroyed data
  - E.g., RAID could be implemented in the OS (disk device driver)

File API

- To read or write, need to open
  - open() returns a handle to the opened file
  - OS associates a (per-process) data structure with the handle
  - This data structure maintains current "cursor" position in the stream of bytes in the file
    - Read and write takes place from the current position
    - Can specify a different location explicitly
- When done, should close the file

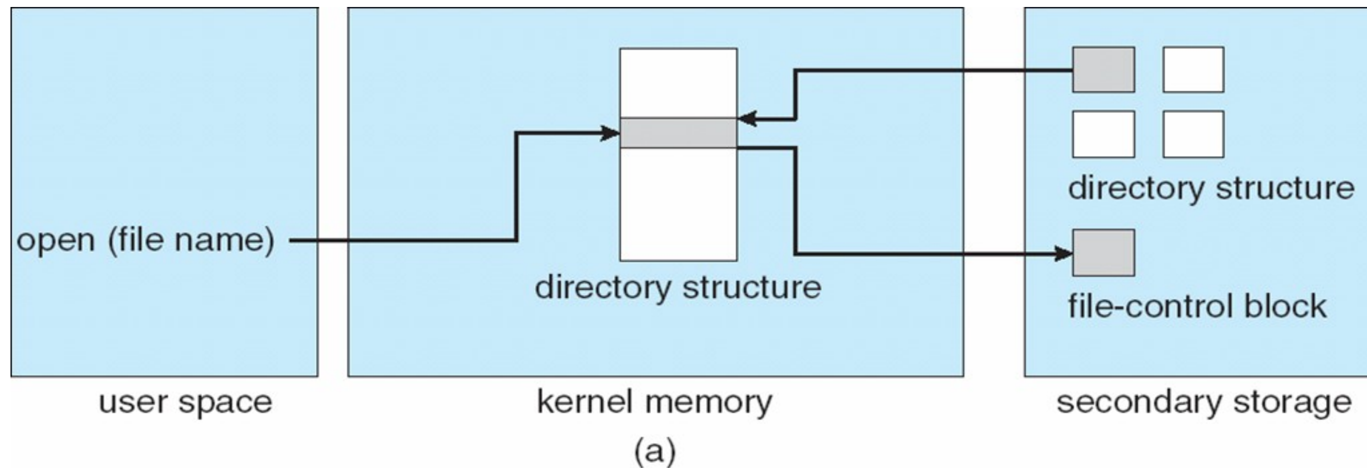
Layered File System

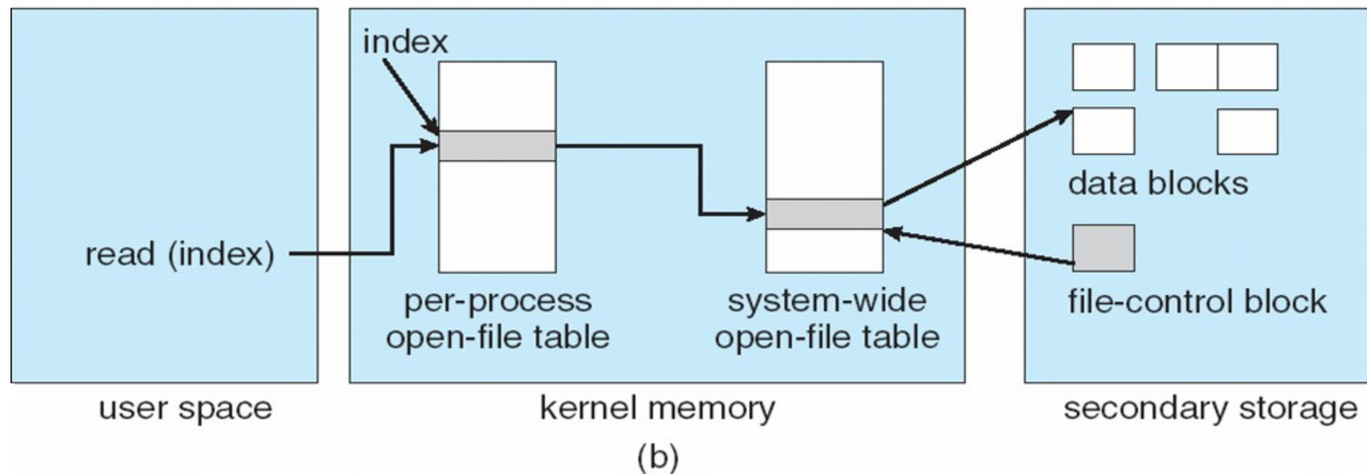


A Typical File Control Block

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

In-Memory File System Structures

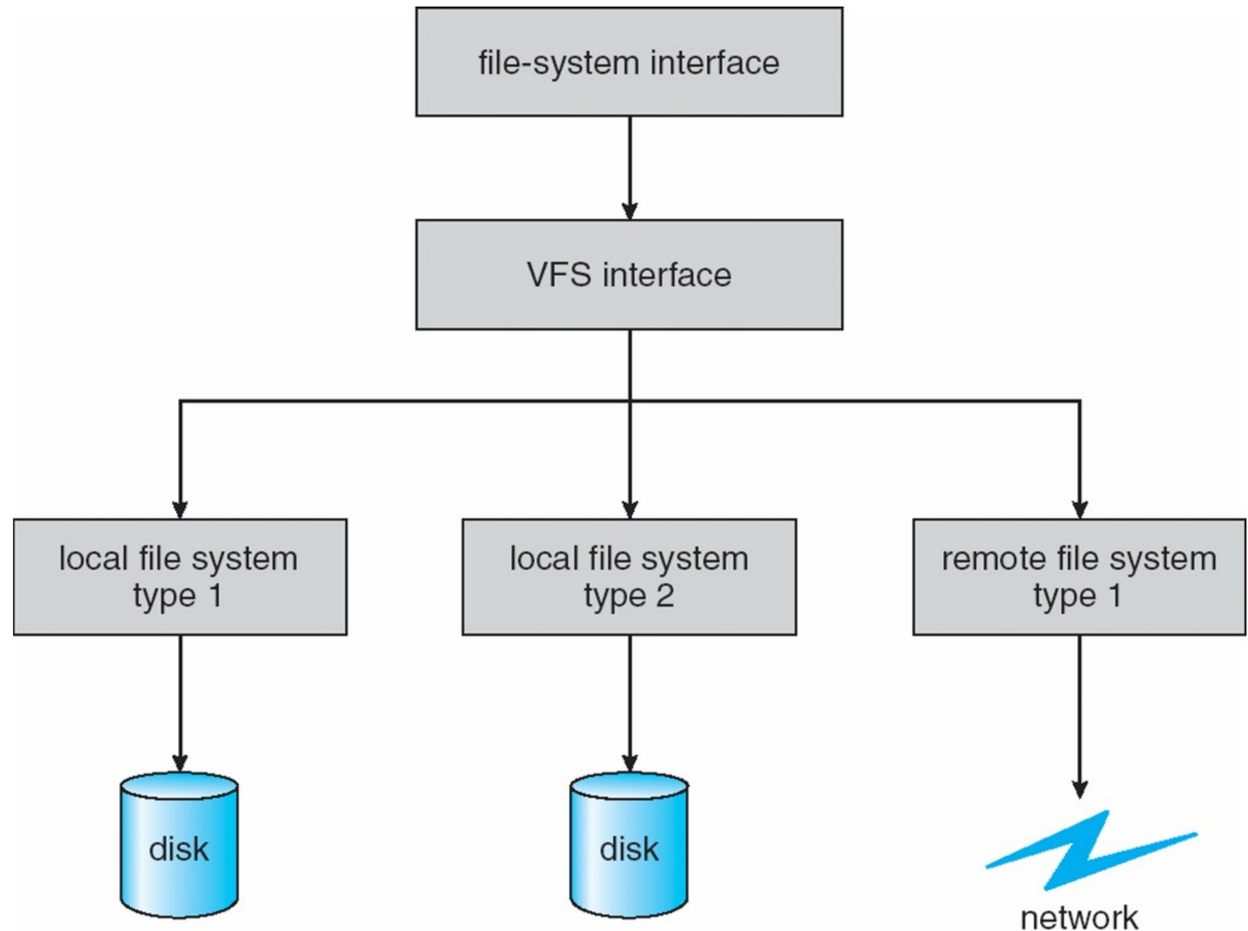




Source: SGG

Virtual File Systems

- Virtual file systems allow the same API to be used by different types of file systems
- The API is to the VFS, rather than any specific type of FS



Source: SGG

VFS details

Data structures used:

- struct inode: represents an individual file
- struct file: represents an open file
- struct superblock: entire file system
- struct dentry: individual directory entry

Implements top-level file system functions

Int open(…)

Ssize_t read(…)

Ssize_t write(…)

Int mmap(…)

Each of these invokes low-level functions within specific file system implementations (e.g., ext2, ext3, Windows FAT, …)

See example code from Linux VFS

Directory Implementation

- **Linear list** of file names with pointer to the data blocks.
  - simple to program
  - time-consuming to execute

- **Hash Table** – linear list with hash data structure.
  - decreases directory search time
  - **collisions** – situations where two file names hash to the same location
  - fixed size

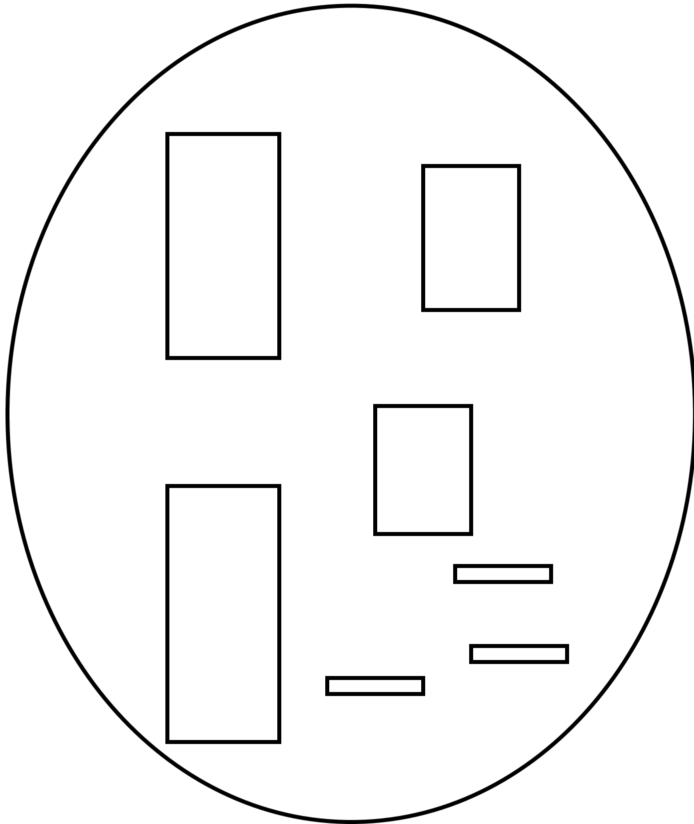
Allocation Methods

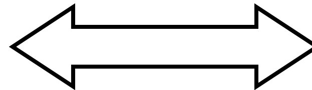
- An allocation method refers to how disk blocks are allocated for files:
- **Contiguous allocation**
- **Linked allocation**
- **Indexed allocation**

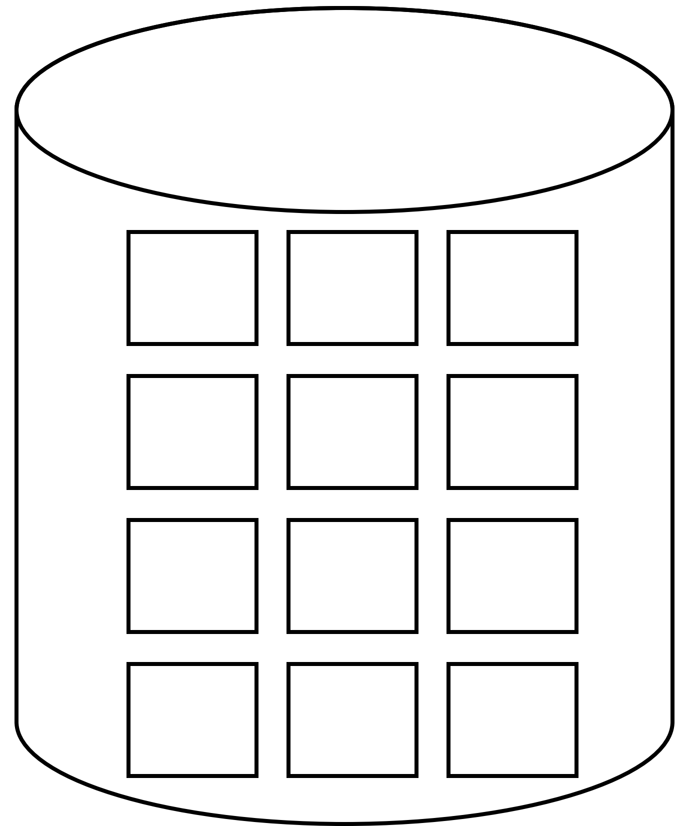
Files vs. Disk: Allocation Methods

Files

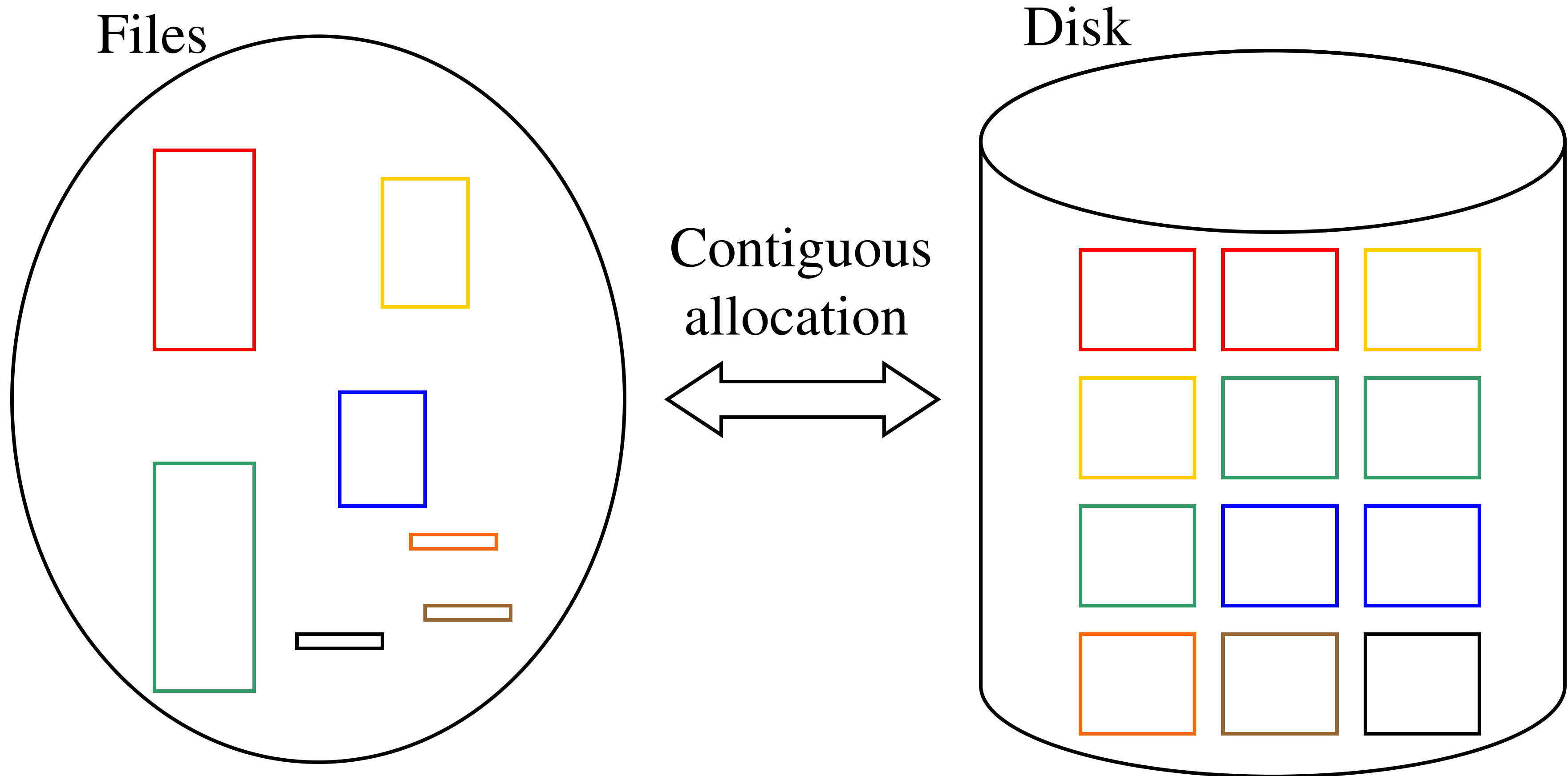
Disk





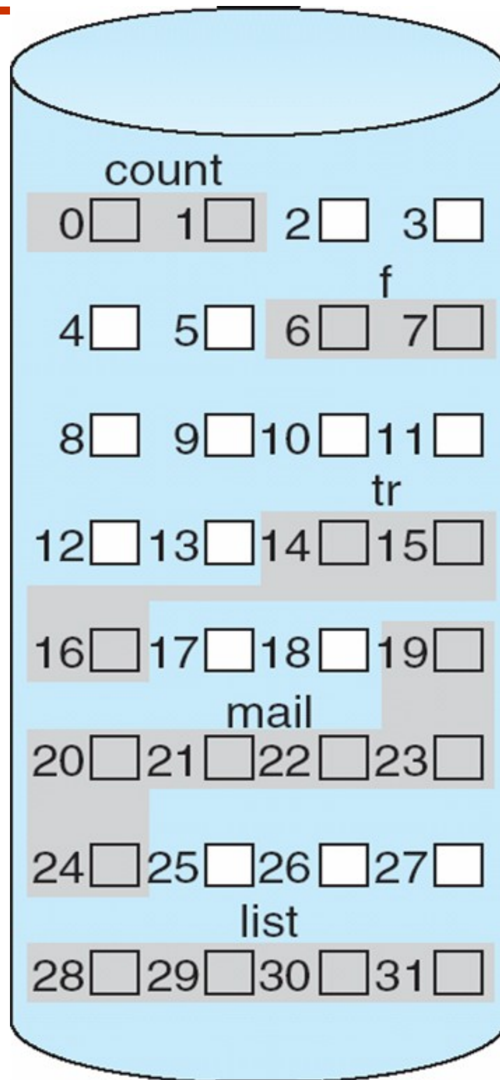


Files vs. Disk: Contiguous



What's the problem with this mapping function?
What's the potential benefit of this mapping function?

Contiguous Allocation of Disk Space



directory

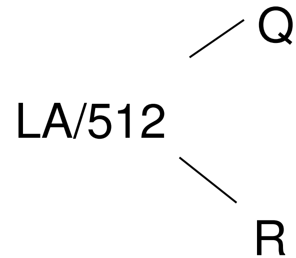
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk
- Simple – only starting location (block #) and length (number of blocks) are required
- Random access
- Wasteful of space (dynamic storage-allocation problem)
- Files cannot grow

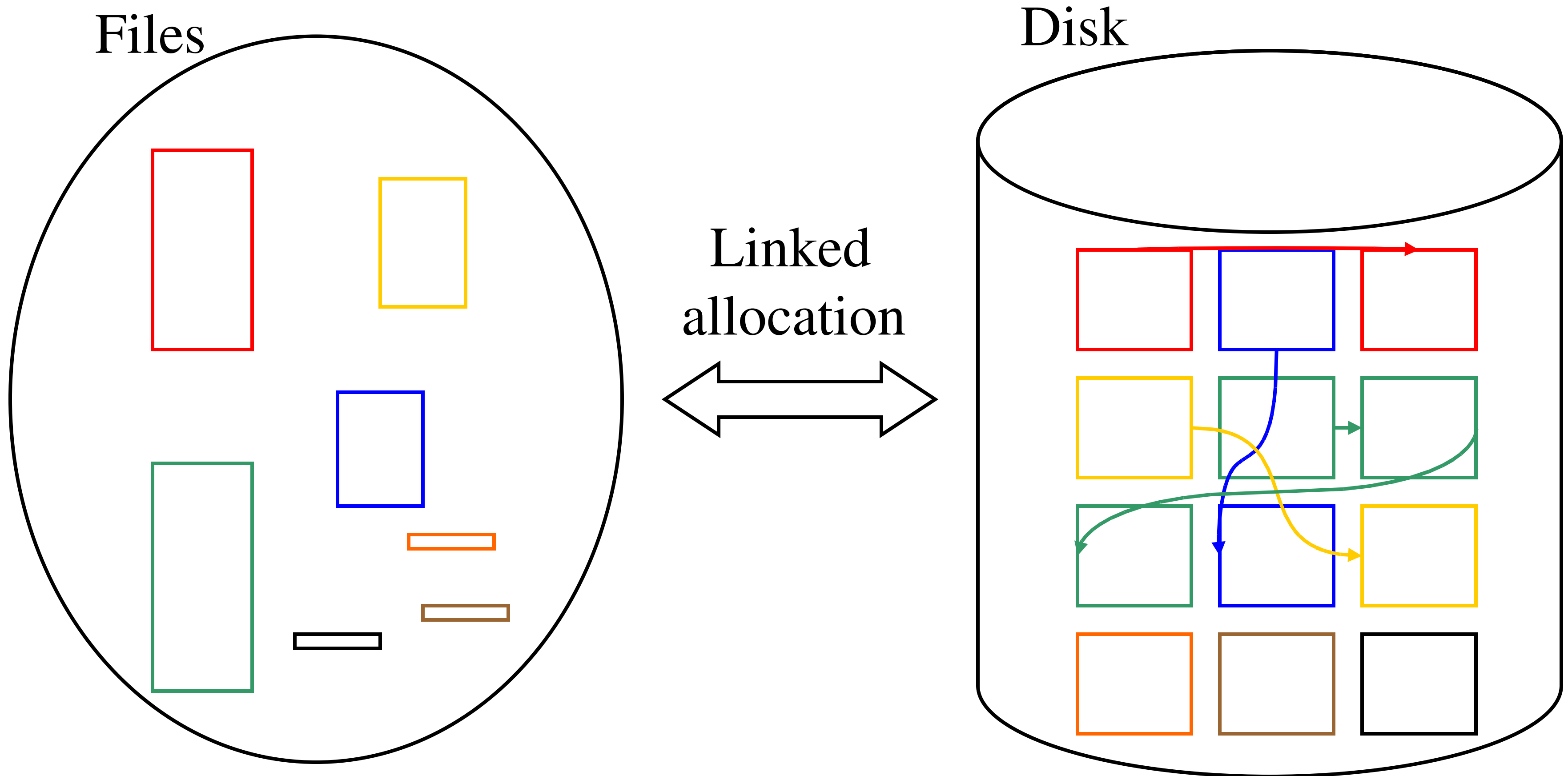
Contiguous Allocation

- Mapping from logical to physical



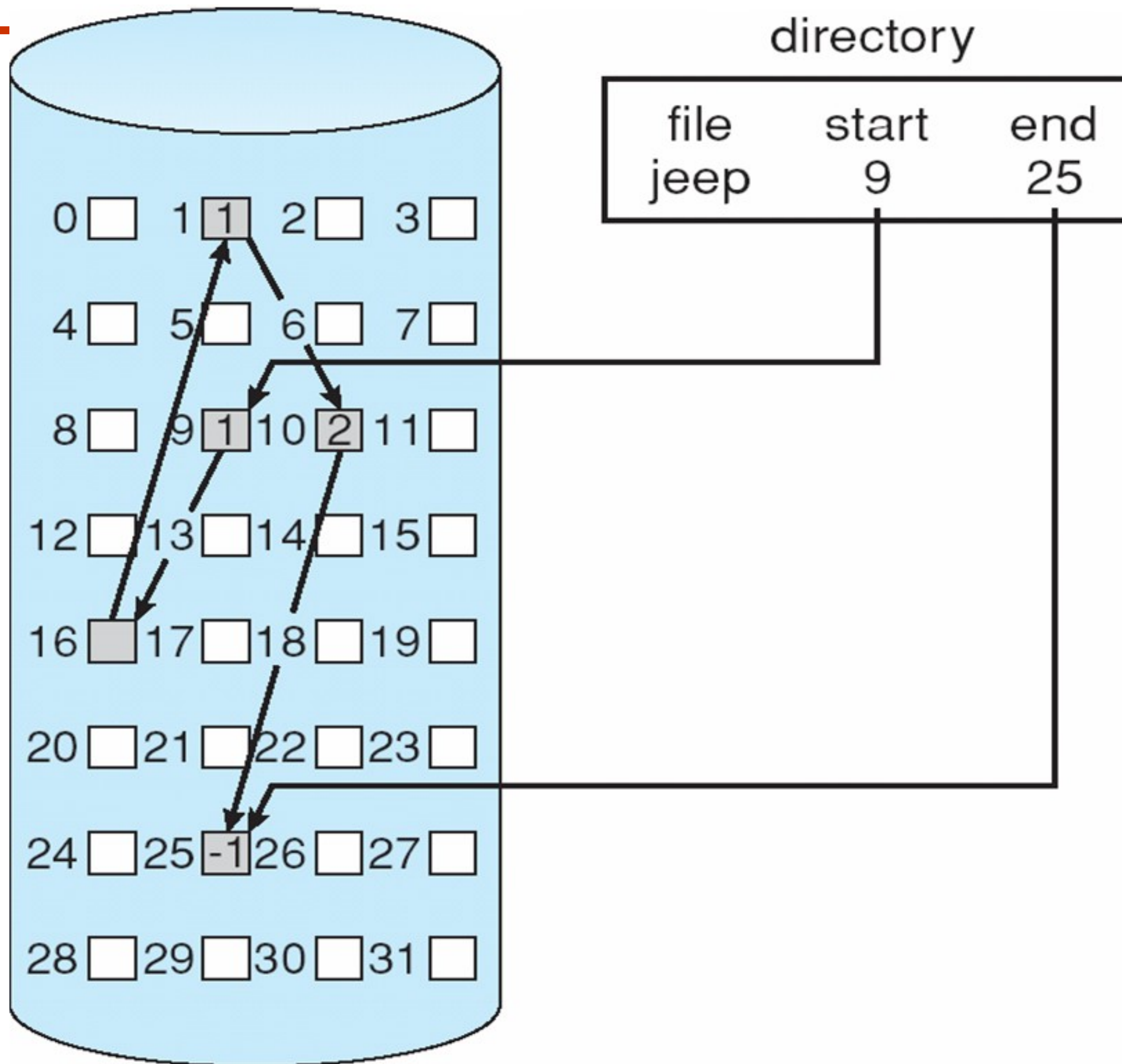
Block to be accessed = ! + starting address
Displacement into block = R

Files vs. Disk



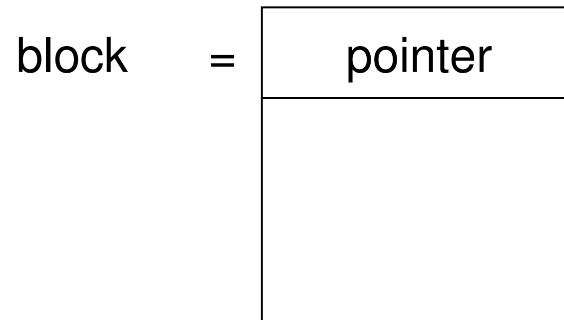
What's the problem with this mapping function?

Linked Allocation



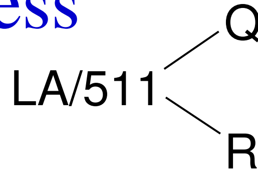
Linked Allocation

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



Linked Allocation (Cont.)

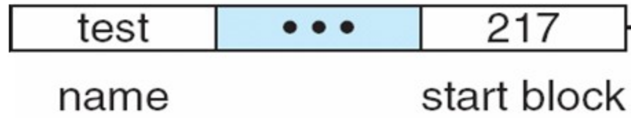
- Simple – need only starting address
- Free-space management system – no waste of space
- No random access
- Mapping

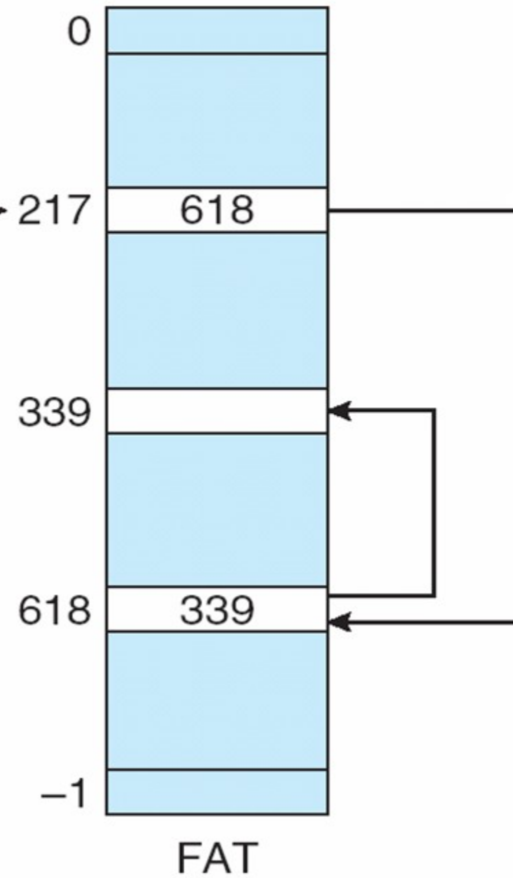


Block to be accessed is the Qth block in the linked chain of blocks representing the file.
Displacement into block = R + 1

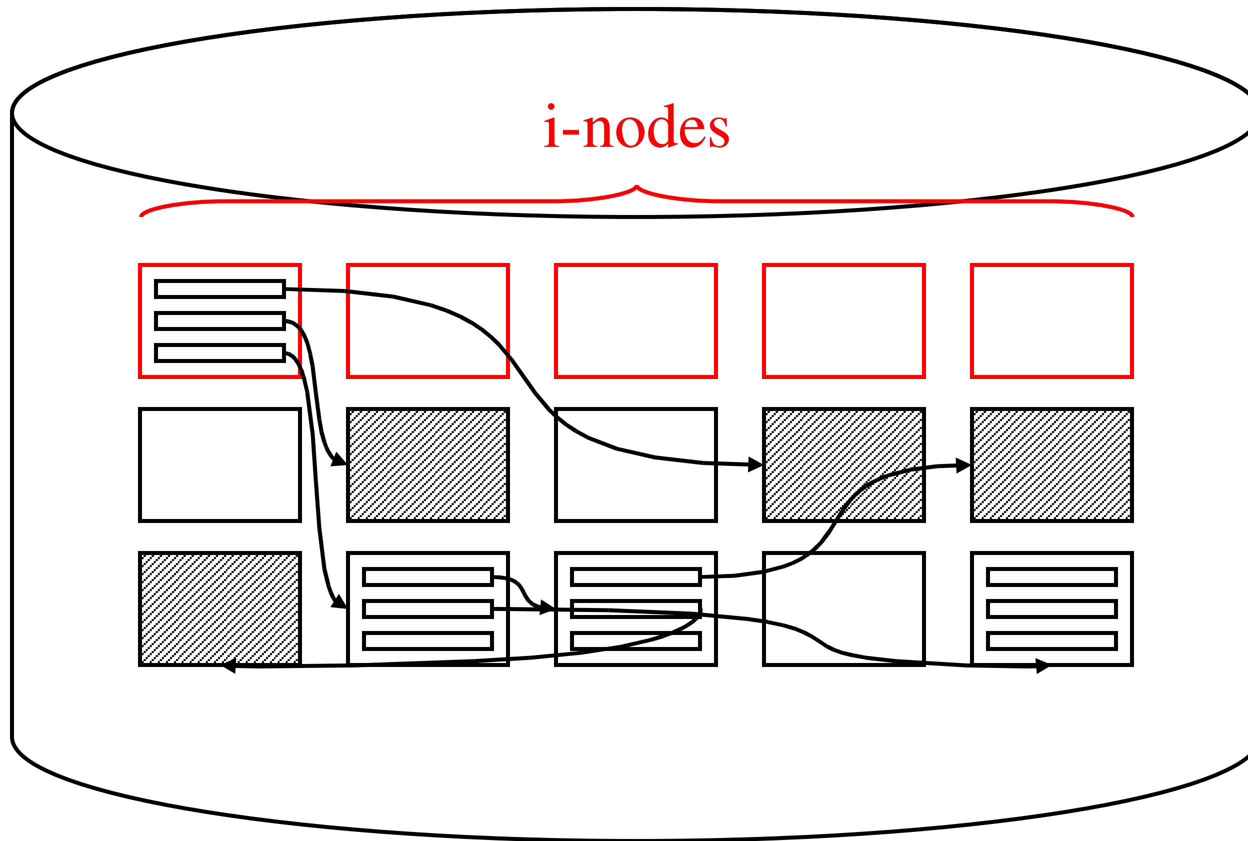
File-allocation table (FAT) – disk-space allocation used by MS-DOS and OS/2.

File-Allocation Table

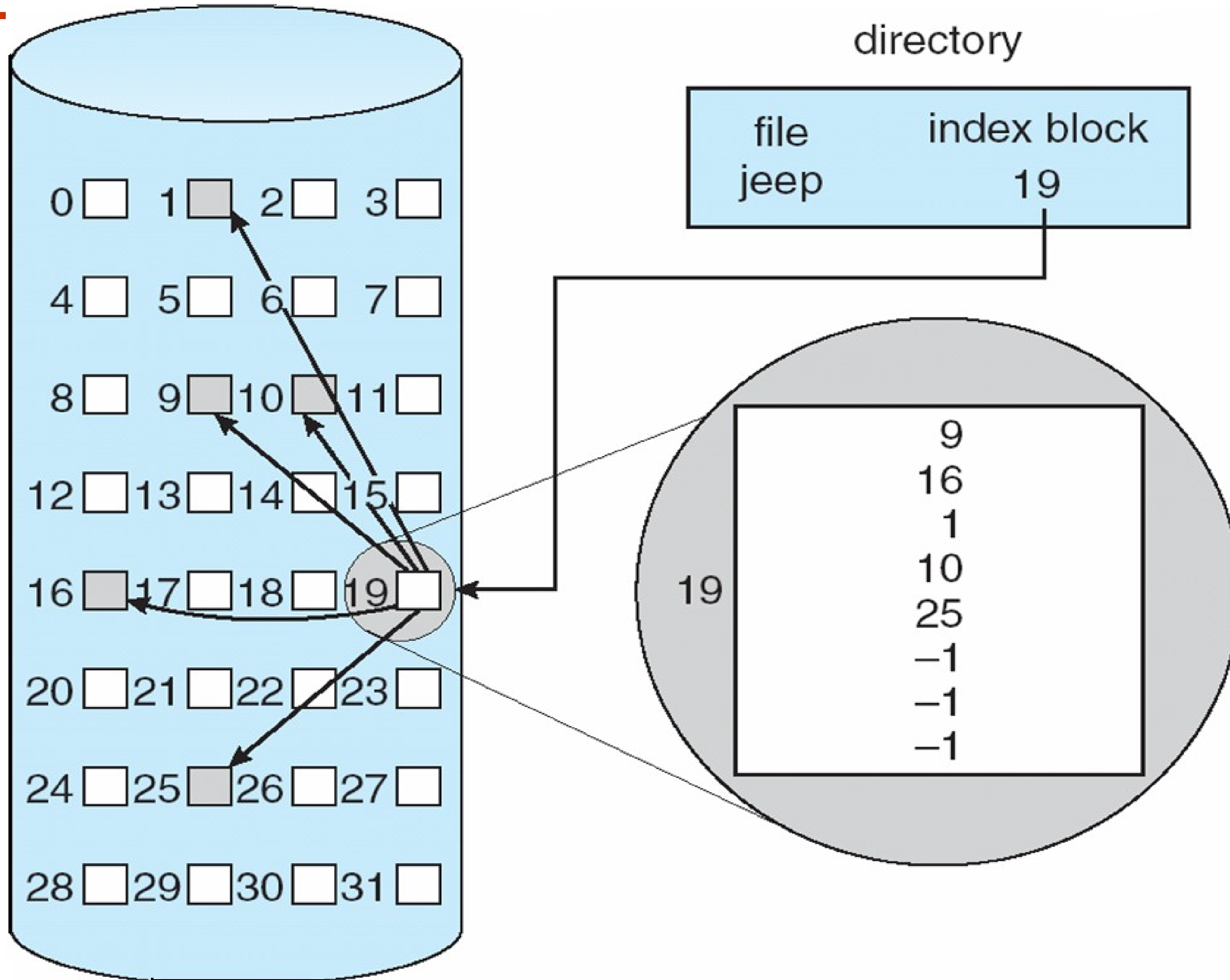




Indexed Allocation: UNIX File

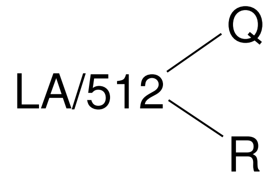


Example of Indexed Allocation



Indexed Allocation (Cont.)

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block.
- Mapping from logical to physical in a file of maximum size of 256K words and block size of 512 words. We need only 1 block for index table.



Q = displacement into index table
R = displacement into block

Indexed Allocation – Mapping (Cont.)

- Mapping from logical to physical in a file of unbounded length (block size of 512 words).
- Linked scheme – Link blocks of index table (no limit on size).

Indexed Allocation – Mapping (Cont.)

- Two-level index (maximum file size is 512^3)

$$LA / (512 \times 512) \rightarrow Q_1, R_1$$

Q_1 = displacement into outer-index

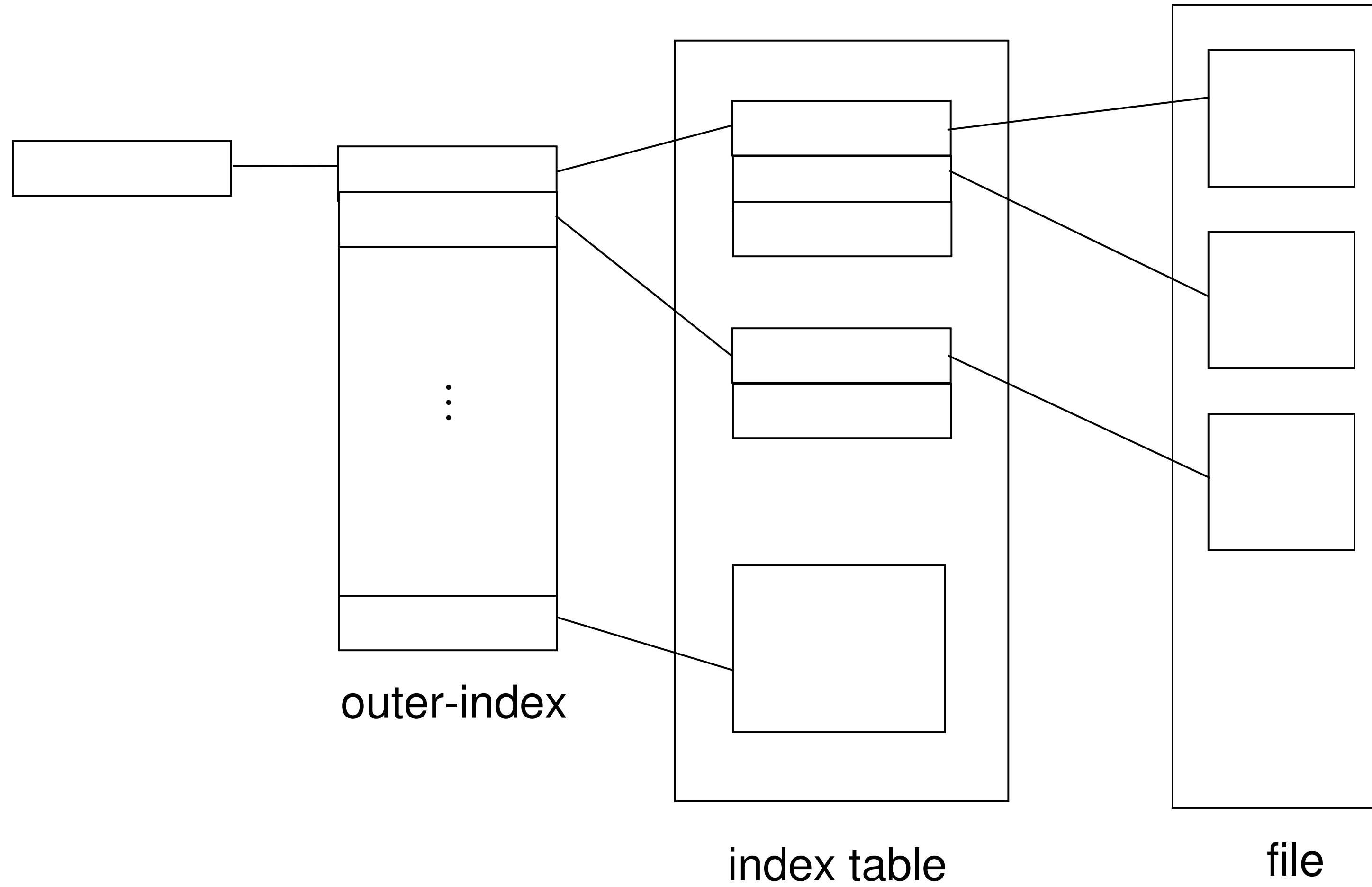
R_1 is used as follows:

$$R_1 / 512 \rightarrow Q_2, R_2$$

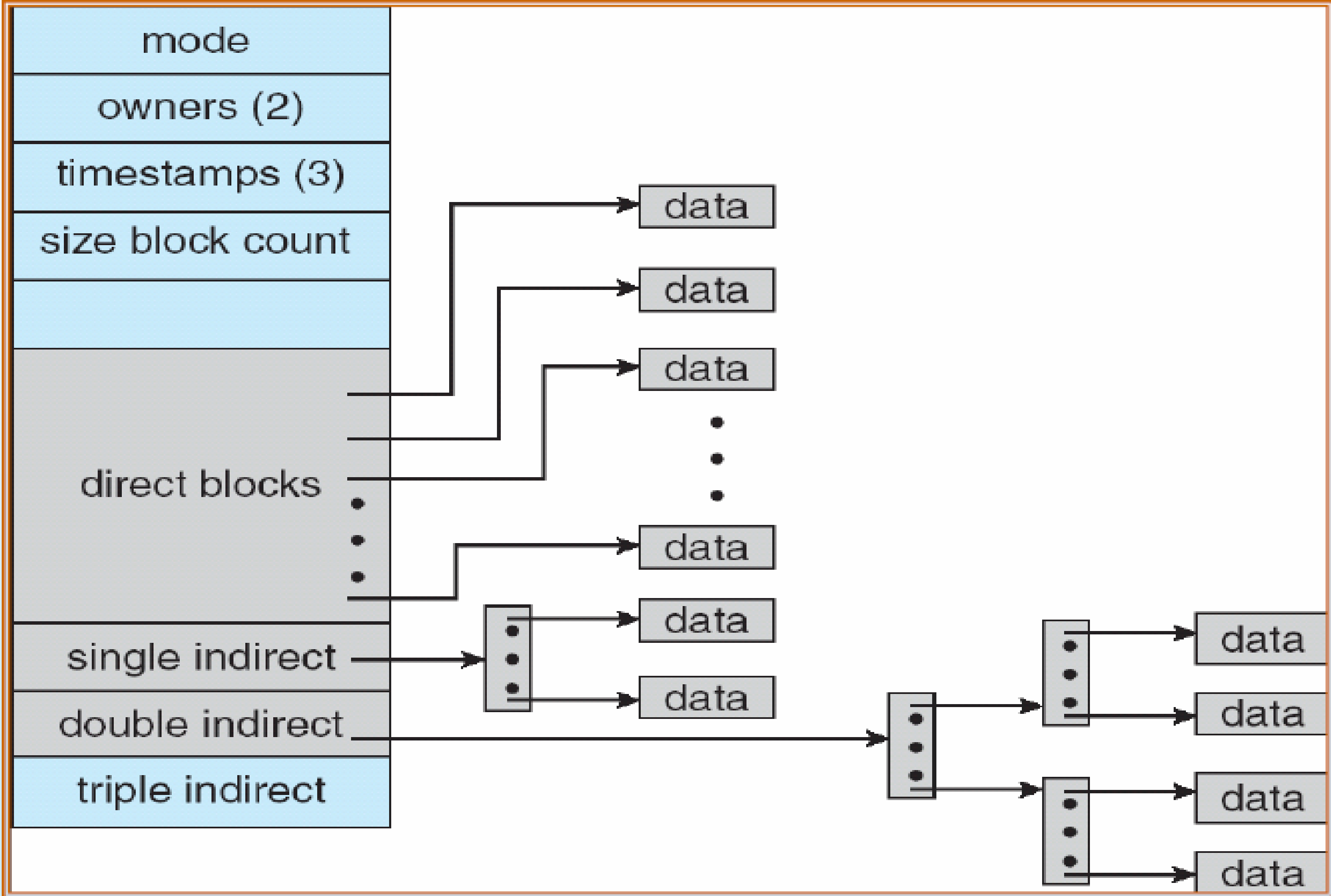
Q_2 = displacement into block of index table

R_2 displacement into block of file:

Indexed Allocation – Mapping (Cont.)



Indexed Allocation: UNIX File



De-fragmentation

Want index-based organization of disk blocks of a file for efficient random access and no fragmentation

Want sequential layout of disk blocks for efficient sequential access

How to reconcile?

De-fragmentation (cont'd)

Base structure is index-based

Optimize for sequential access

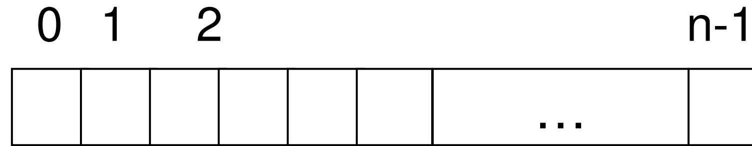
- De-fragmentation: move the blocks around to simulate actual sequential layout of files
- Group allocation of blocks: group tracks together (cylinders). Try to allocate all blocks of a file from a single cylinder group so that they are close together. This style of grouped allocation was first proposed for the BSD Fast File System and later incorporated in ext2 (Linux).
- Extents: on each write that extends a file, allocate a chunk of consecutive blocks. Some modern systems use extents, e.g. VERITAS (supported in many systems like Linux and Solaris), the first commercial journaling file system. Ext4 can use them also (extents are not the default option, though).

Free Space Management

- No policy issues here – just mechanism
- Bitmap: one bit for each block on the disk
  - Good to find a contiguous group of free blocks
    - Files are often accessed sequentially

Free-Space Management

- Bit vector (*n* blocks)



bit[*i*] =
- 0 ⇒ block[*i*] occupied
- 1 ⇒ block[*i*] free

Block number calculation

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

Free-Space Management (Cont.)

- Bit map requires extra space
  - Example:

    block size = 2^{12} bytes

    disk size = 2^{30} bytes (1 gigabyte)

    $n = 2^{30}/2^{12} = 2^{18}$ bits (or 32K bytes)
- Easy to get contiguous files
- Linked list (free list)
  - Cannot get contiguous space easily
  - No waste of space
- Grouping
- Counting

Free-Space Management (Cont.)

- Need to protect:
  - Pointer to free list
  - Bit map
    - Must be kept on disk
    - Copy in memory and disk may differ
    - Cannot allow for block[*i*] to have a situation where bit[*i*] = 1 in memory and bit[*i*] = 0 on disk
  - Solution:
    - Set bit[*i*] = 1 in disk
    - Allocate block[*i*]
    - Set bit[*i*] = 1 in memory

File System

OK, we have files

How can we name them?

How can we organize them?

File Naming

- Each file has an associated human-readable name
  - E.g., usr, bin, mid-term.pdf, design.pdf
- File name must be globally unique
  - Otherwise how would the system know which file we are referring to?
- OS must maintain a mapping between a file name and the set of blocks belonging to the file
  - In Unix, this is a mapping between names and i-nodes
- Mappings are kept in directories

Unix File System

Ordinary files (uninterpreted)

Directories

- Directory is differentiated from ordinary file by bit in i-node
- File of files: consists of records (directory entries), each of which contains info about a file and a pointer to its i-node
- Organized as a rooted tree
- Pathnames (relative and absolute)
- Contains links to parent, itself
- Multiple links to files can exist: hard (points to the actual file data) or symbolic (symbolic path to a hard link). Both types of links can be created with the ln utility. Removing a symbolic link does not affect the file data, whereas removing the last hard link to a file will remove the data.

Storage Organization

BB : Boot Block
SB : Super Block
IL : inode List
DB: Data Blocks

Info stored on the SB: size of the file system, number of free blocks, list of free blocks, index to the next free block, size of the I-node list, number of free I-nodes, list of free I-nodes, index to the next free I-node, locks for free block and free I-node lists, and flag to indicate a modification to the SB

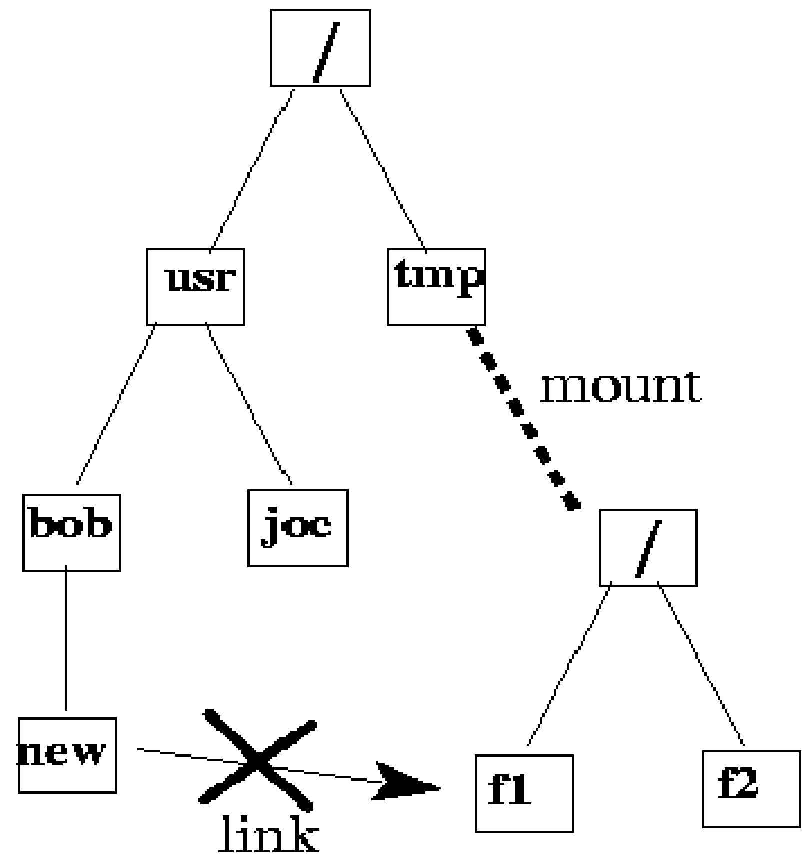
I-node contains: owner, type (directory, file, device), last modified time, last accessed time, last I-node modified time, access permissions, number of links to the file, size, and block pointers

Unix File Systems (Cont'd)

Tree-structured file hierarchies

Mounted on existing space by using mount

No hard links between different file systems



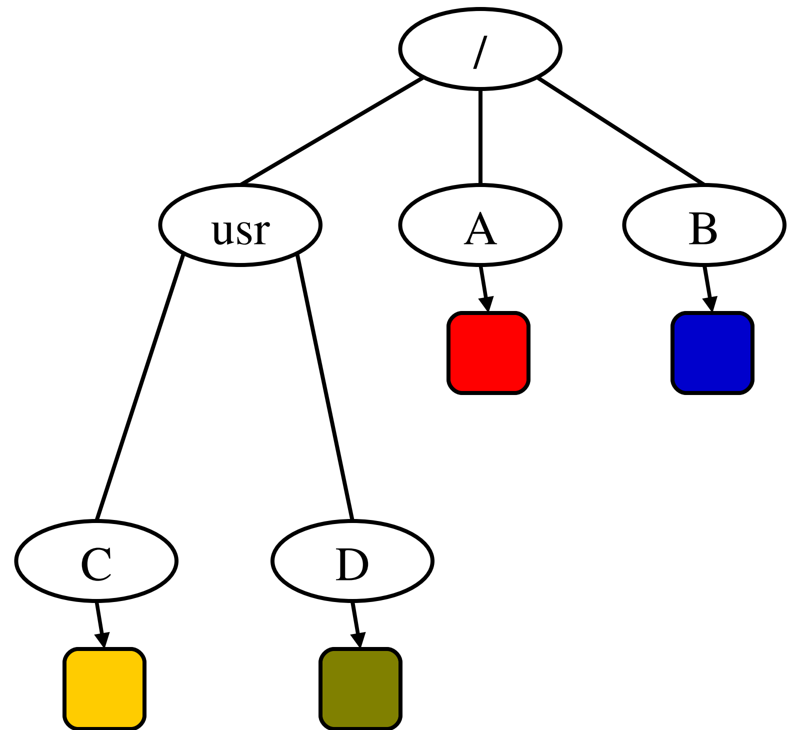
Name Space

In UNIX, “devices are files”

- E.g., /dev/cdrom, /dev/tape
- User process access devices by accessing corresponding file

A name space

- Name $\leftrightarrow$ object
- Objects may support same API (as in Unix) or different APIs (object-oriented systems)



File System Buffer Cache

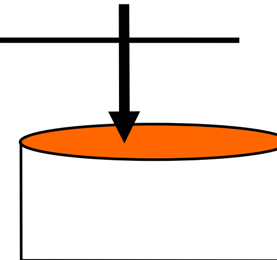
application: *read/write files*

OS: *translate file to disk blocks*

maintains

controls disk accesses: read/write blocks

hardware:

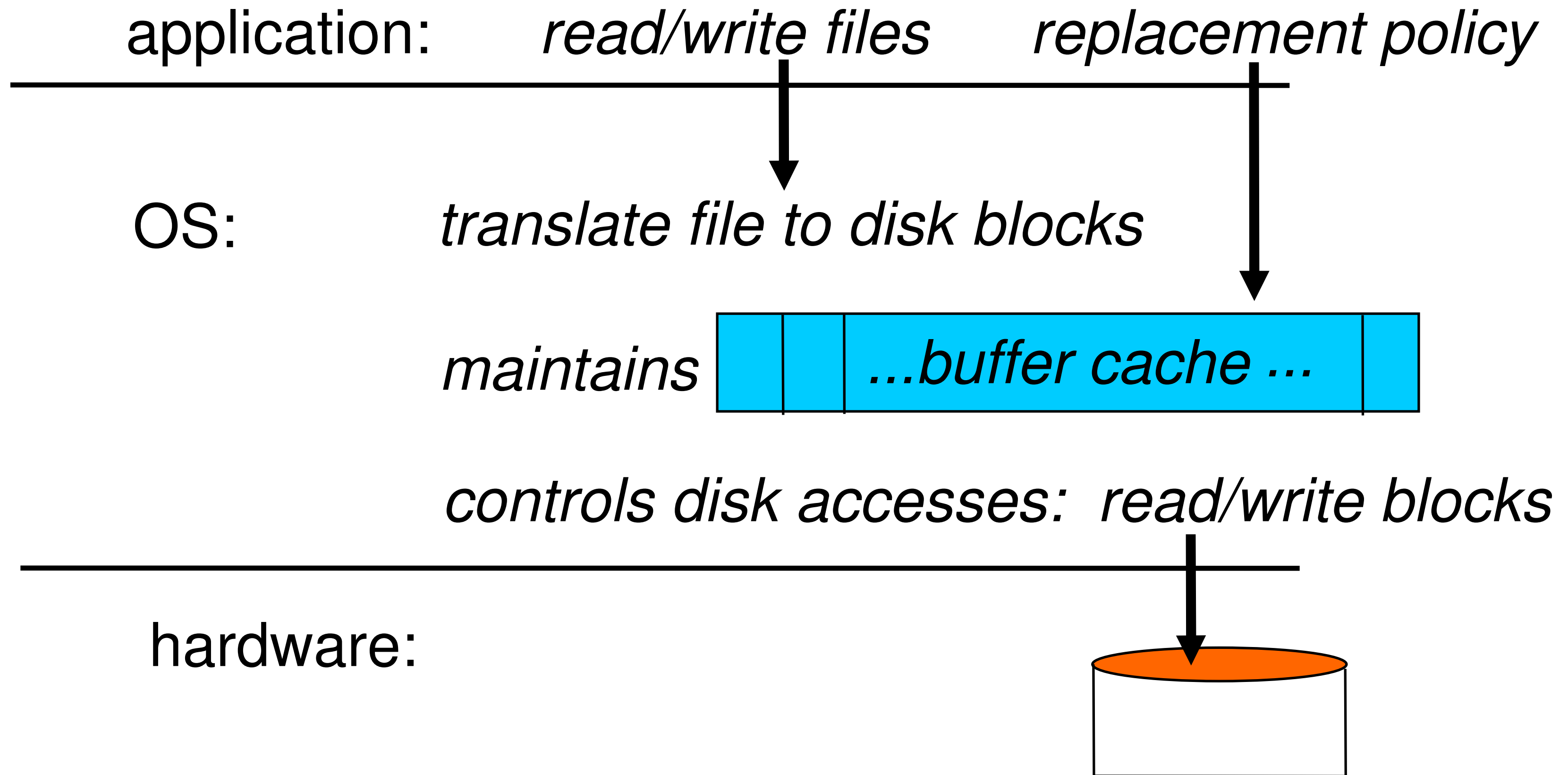


Any problems?

File System Buffer Cache

- Disks are “stable” while memory is volatile
  - What happens if you buffer a write and the machine crashes before the write has been saved to disk?
  - Can use write-through but write performance will suffer
  - In UNIX
    - Use unbuffered I/O when writing i-nodes or pointer blocks
    - Use buffered I/O for other writes and force sync every 30 seconds
  - Will talk more about this in a few slides
- What about replacement?
- How can we further improve performance?

Application-Controlled Caching



File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a **protection** scheme
- On distributed systems, files may be shared across a network
- Network File System (NFS) is a common distributed file-sharing method

File Sharing – Multiple Users

- **User IDs** identify users, allowing permissions and protections to be per-user

- **Group IDs** allow users to be in groups, permitting group access rights

File Sharing – Remote File Systems

- Uses networking to allow file system access between systems
  - Manually via programs like FTP
  - Automatically, seamlessly using **distributed file systems**
  - Semi automatically via the **world wide web**
- **Client-server** model allows clients to mount remote file systems from servers
  - Server can serve multiple clients
  - Client and user-on-client identification is insecure or complicated
  - **NFS** is standard UNIX client-server file sharing protocol

File Sharing – Failure Modes

- Remote file systems add new failure modes, due to network failure, server failure
- Recovery from failure can involve state information about status of each remote request
- Stateless protocols such as NFS include all information in each request, allowing easy recovery but less security

File Sharing and Consistency

Can multiple processes open the same file at the same time?

What happens if two or more processes write to the same file?

What happens if two or more processes try to create the same file at the same time?

What happens if a process deletes a file when another has it opened?

File Sharing – Consistency Semantics

- **Consistency semantics** specify how multiple users are to access a shared file simultaneously
  - Similar to Ch 7 process synchronization algorithms
    - Tend to be less complex due to disk I/O and network latency (for remote file systems

File Sharing and Consistency (Cont'd)

- Several possibilities for file sharing semantics
- Unix semantics: file associated with single physical image
  - Writes by one user are seen immediately by others who also have the file open.
  - One sharing mode allows file pointer to be shared.
- Session semantics (AFS): file may be associated temporarily with several images at the same time
  - Writes by one user are not immediately seen by others who also have the file open.
  - Once a file is closed, the changes made to it are visible only in sessions starting later.
- Immutable-file semantics: file declared as shared cannot be written.

File System Consistency on Crashes

File system almost always uses a buffer/disk cache for performance reasons

Two copies of a disk block (buffer cache, disk) ➜ consistency problem if the system crashes before all the modified blocks are written back to disk

This problem is critical especially for the blocks that contain control information: i-node, free-list, directory blocks

Utility programs for checking block and directory consistency

Write back critical blocks from the buffer cache to disk immediately

Data blocks are also written back periodically: sync

More on File System Consistency

To maintain file system consistency the ordering of updates from buffer cache to disk is critical

Writing to a file may involve updating several pieces of metadata. For example, extending a file requires updates to the directory entry (file size, last access), to the i-node (extra block), and to the free list (one fewer block free).

Example problem: if the directory block is written back before the i-node and the system crashes, the directory structure will be inconsistent

Similar case when i-node and free list are updated

A more elaborate solution: use dependencies between blocks containing control data in the buffer cache to specify the ordering of updates

More on File System Consistency

Even with a pre-specified ordering of metadata updates, it might be impossible to re-establish consistency after a crash.

Hence, another solution: Journaling (e.g., ext3 for Linux)

How does it work? OS writes metadata updates synchronously to a log and returns control to user process. In the background, the log is replayed with transaction semantics. When a set of related operations (i.e., a transaction) is performed across the actual file system, they are deleted from the log. On a crash, any incomplete transactions are rolled back.

Side advantage: metadata updates perform quickly because log is sequential.

Protection

- File owner/creator should be able to control:
  - what can be done
  - by whom

- Types of access
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List**

A Sample UNIX Directory Listing

```
-rw-rw-r--   1 pbg  staff    31200  Sep 3 08:30   intro.ps
drwx------   5 pbg  staff      512  Jul 8 09.33   private/
drwxrwxr-x   2 pbg  staff      512  Jul 8 09:35   doc/
drwxrwx---   2 pbg  student    512  Aug 3 14:13   student-proj/
-rw-r--r--   1 pbg  staff     9423  Feb 24 2003   program.c
-rwxr-xr-x   1 pbg  staff    20471  Feb 24 2003   program
drwx--x--x   4 pbg  faculty    512  Jul 31 10:31  lib/
drwx------   3 pbg  staff     1024  Aug 29 06:52  mail/
drwxrwxrwx   3 pbg  staff      512  Jul 8 09:35   test/
```

Protection Mechanisms

Files are OS objects (like other resources, such as a printer): they have unique names and a finite set of operations that processes can perform on them

Protection domain defines a set of {object,rights} where right is the permission to perform one of the operations on the object

At every instant, each process runs in some protection domain

In Unix, a protection domain is {uid, gid}

Protection domain in Unix is switched when running a program with SETUID/SETGID set or when the process enters the kernel mode by issuing a system call

How to store the info about all the protection domains?

Protection Mechanisms (cont'd)

Access Control List (ACL): associate with each object a list of all the protection domains that may access the object and how.

- In Unix, an ACL defines three protection domains: owner, group and others

Capability List (C-list): associate with each process a list of objects that may be accessed along with the operations

- C-list implementation issues: where/how to store them (hardware, kernel, encrypted in user space) and how to revoke them

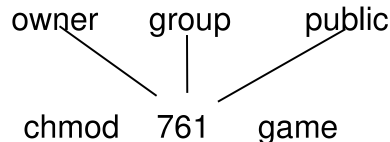
Most systems use a combination of ACLs and C-Lists. Example: In Unix, an ACL is checked when first opening a file. After that, system relies on kernel information (per-process file table) that is established during the open call. This obviates the need for further protection checks.

Access Lists and Groups

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

		RWX	
a) **owner access**	7	$\Rightarrow$	1 1 1
b) **group access**	6	$\Rightarrow$	1 1 0
c) **public access**	1	$\Rightarrow$	0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.



Attach a group to a file

chgrp G game

Research in FSs: An Energy-Aware File System

Large file/storage servers are pretty common these days.

For these servers (and the data centers where they reside), power and energy are serious problems. Power affects installation and cooling investments. Energy affects electricity costs.

Given the replication of resources in these servers, can conserve energy by turning resources off, just like in laptops or other battery-operated devices.

Can you think of how to do this at the file- or storage-system level?

Leveraging Redundancy

Idea 1: segregate “original” and “redundant” files/blocks onto different sets of disks.

At each server, can turn off disks that store redundant data under light and moderate loads. Extrapolate to entire servers, so that whole nodes can be turned off.

Somehow keep logs of writes so that redundant disks can be updated when they are turned back on.

Problems: keeping write logs, deciding when to turn on the redundant disks, etc.

Leveraging Popularity

Idea 2: segregate popular and unpopular files/blocks, accepting some performance degradation for the latter.

At each server, can turn off disks that contain unpopular files/blocks. Extrapolate to all servers, so that whole nodes can be turned off.

Implement file/block migration and/or replication to adjust to variations in popularity.

Problems: provisioning, determining popularity, policies for migration and replication, etc.