File Systems

CS 4410
Operating Systems

Cornell University
Cornell CIS
COMPUTING AND INFORMATION SCIENCE

Storing Information

- Applications can store it in the process address space
- Why is it a bad idea?
  - Size is limited to size of virtual address space
    - May not be sufficient for airline reservations, banking, etc.
  - The data is lost when the application terminates
    - Even when computer doesn't crash!
  - Multiple process might want to access the same data
    - Imagine a telephone directory part of one process

File Systems

- 3 criteria for long-term information storage:
  - Should be able to store very large amount of information
  - Information must survive the processes using it
  - Should provide concurrent access to multiple processes
- Solution:
  - Store information on disks in units called **files**
  - Files are persistent, and only owner can explicitly delete it
  - Files are managed by the OS

- File Systems: How the OS manages files!

File Naming

- Motivation: Files abstract information stored on disk
  - You do not need to remember block, sector, …
  - We have human readable names
- How does it work?
  - Process creates a file, and gives it a name
    - Other processes can access the file by that name
  - Naming conventions are OS dependent
    - Usually names as long as 255 characters is allowed
    - Digits and special characters are sometimes allowed
    - MS-DOS and Windows are not case sensitive, UNIX family is

File Extensions

- Name divided into 2 parts, second part is the extension
- On UNIX, extensions are not enforced by OS
  - However C compiler might insist on its extensions
    - These extensions are very useful for C
- Windows attaches meaning to extensions
  - Tries to associate applications to file extensions

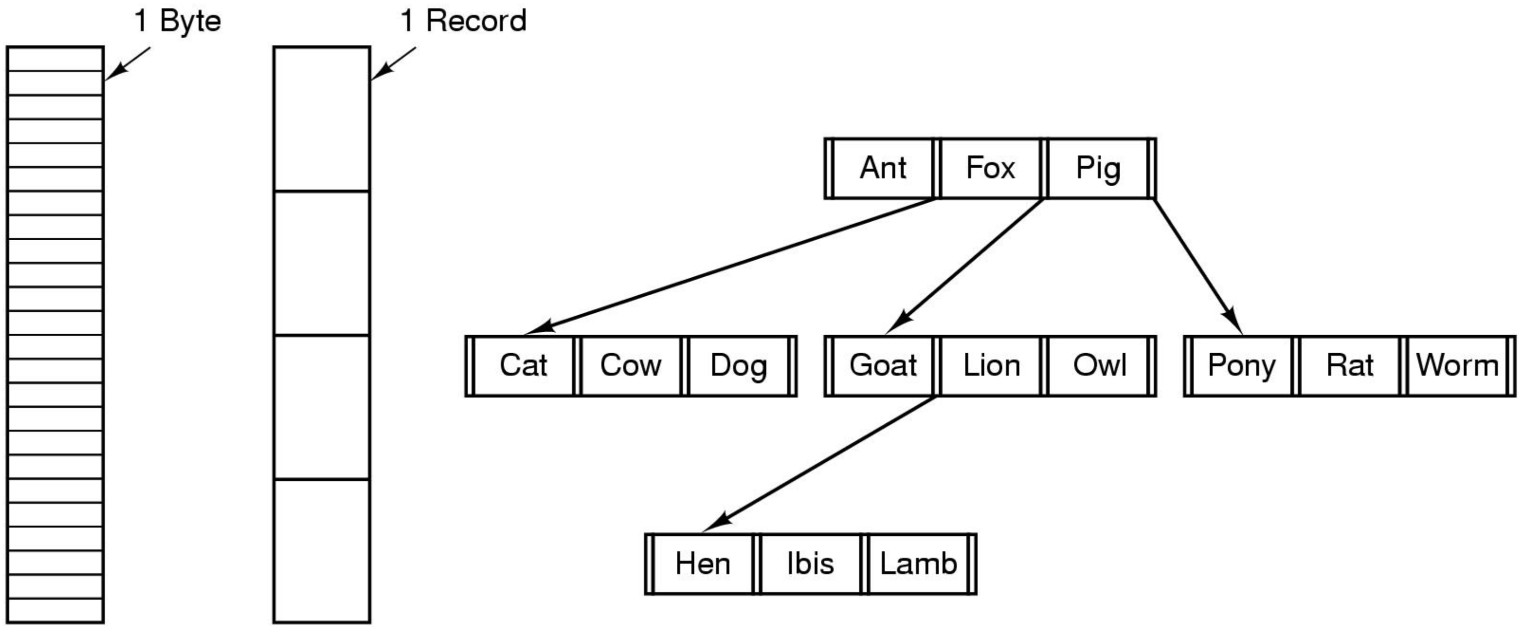
Internal File Structure

(a) Byte Sequence: unstructured

(b) Record sequence: r/w in records, relates to sector sizes

(c) Complex structures, e.g. tree

- Data stored in variable length records; OS specific meaning of each file



(a) (b) (c)

File Access

- Sequential access
  - read all bytes/records from the beginning
  - cannot jump around, could rewind or forward
  - convenient when medium was magnetic tape
- Random access
  - bytes/records read in any order
  - essential for database systems

File Attributes

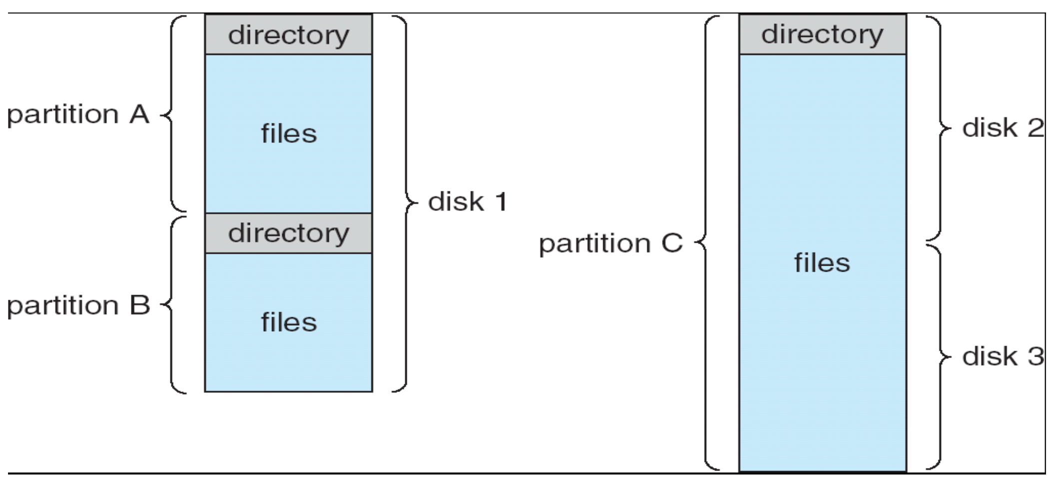
- File-specific info maintained by the OS
  - File size, modification date, creation time, etc.
  - Varies a lot across different OSes
- Some examples:
  - Name – only information kept in human-readable form
  - Identifier – unique tag (number) identifies file within file system
  - Type – needed for 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

Basic File System Operations

- Create a file
- Write to a file
- Read from a file
- Seek to somewhere in a file
- Delete a file
- Truncate a file

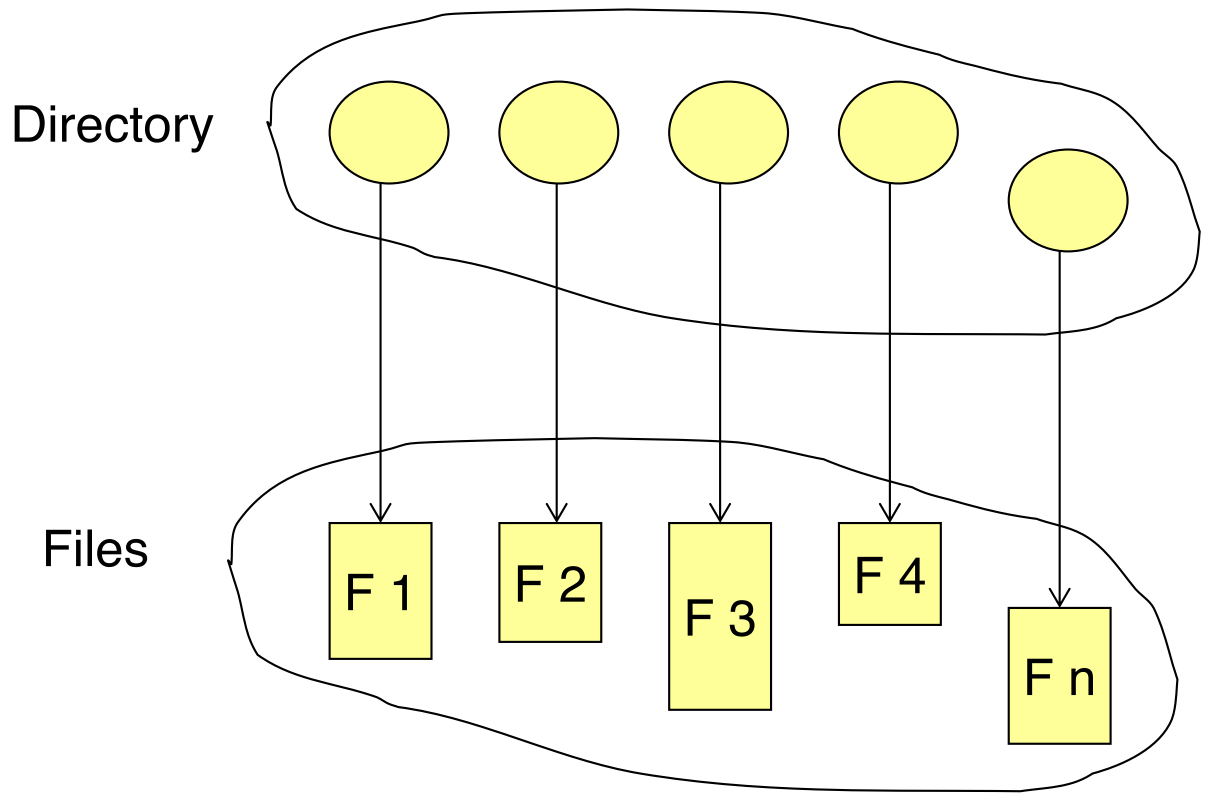
FS on disk

- Could use entire disk space for a FS, but
  - A system could have multiple FSes
  - Want to use some disk space for swap space
- Disk divided into partitions, slices or minidisks
  - Chunk of storage that holds a FS is a volume
  - Directory structure maintains info of all files in the volume
    - Name, location, size, type, …



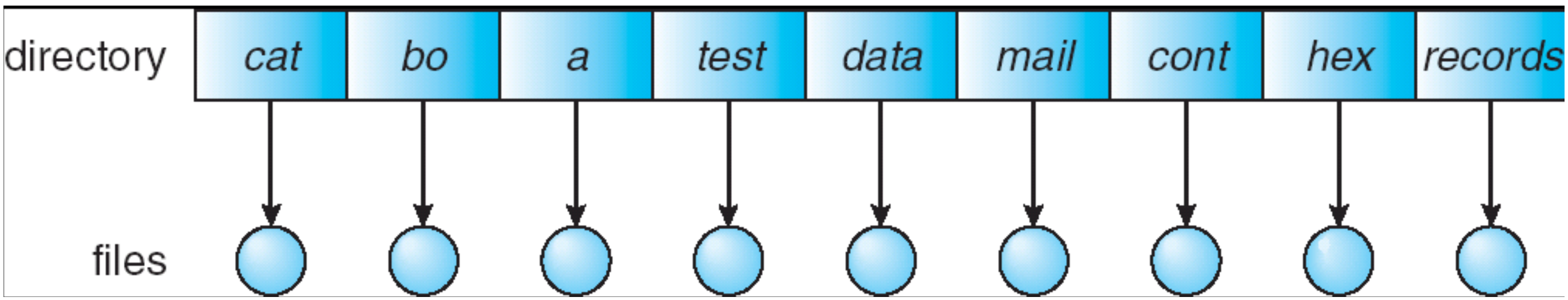
Directories

- Directories/folders keep track of files
  - Is a symbol table that translates file names to directory entries
  - Usually are themselves files
- How to structure the directory to optimize all of the following:
  - Search a file
  - Create a file
  - Delete a file
  - List directory
  - Rename a file
  - Traversing the FS



Single-level Directory

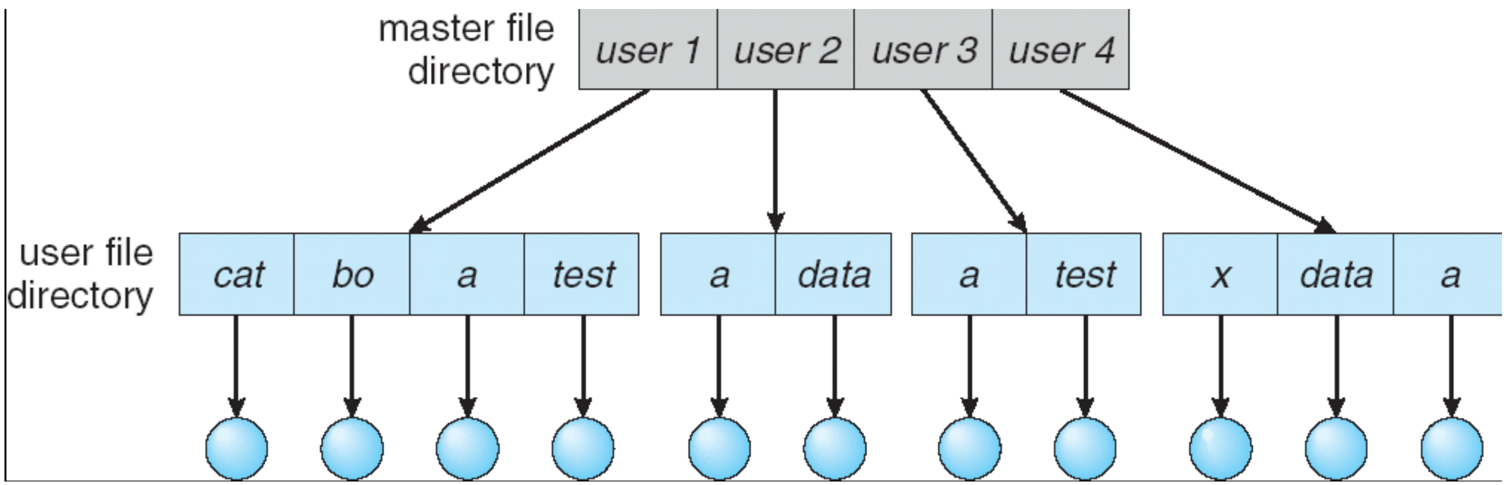
- One directory for all files in the volume
  - Called root directory



  - Used in early PCs, even the first supercomputer CDC 6600
- Pros: simplicity, ability to quickly locate files
- Cons: inconvenient naming (uniqueness, remembering all)

Two-level directory

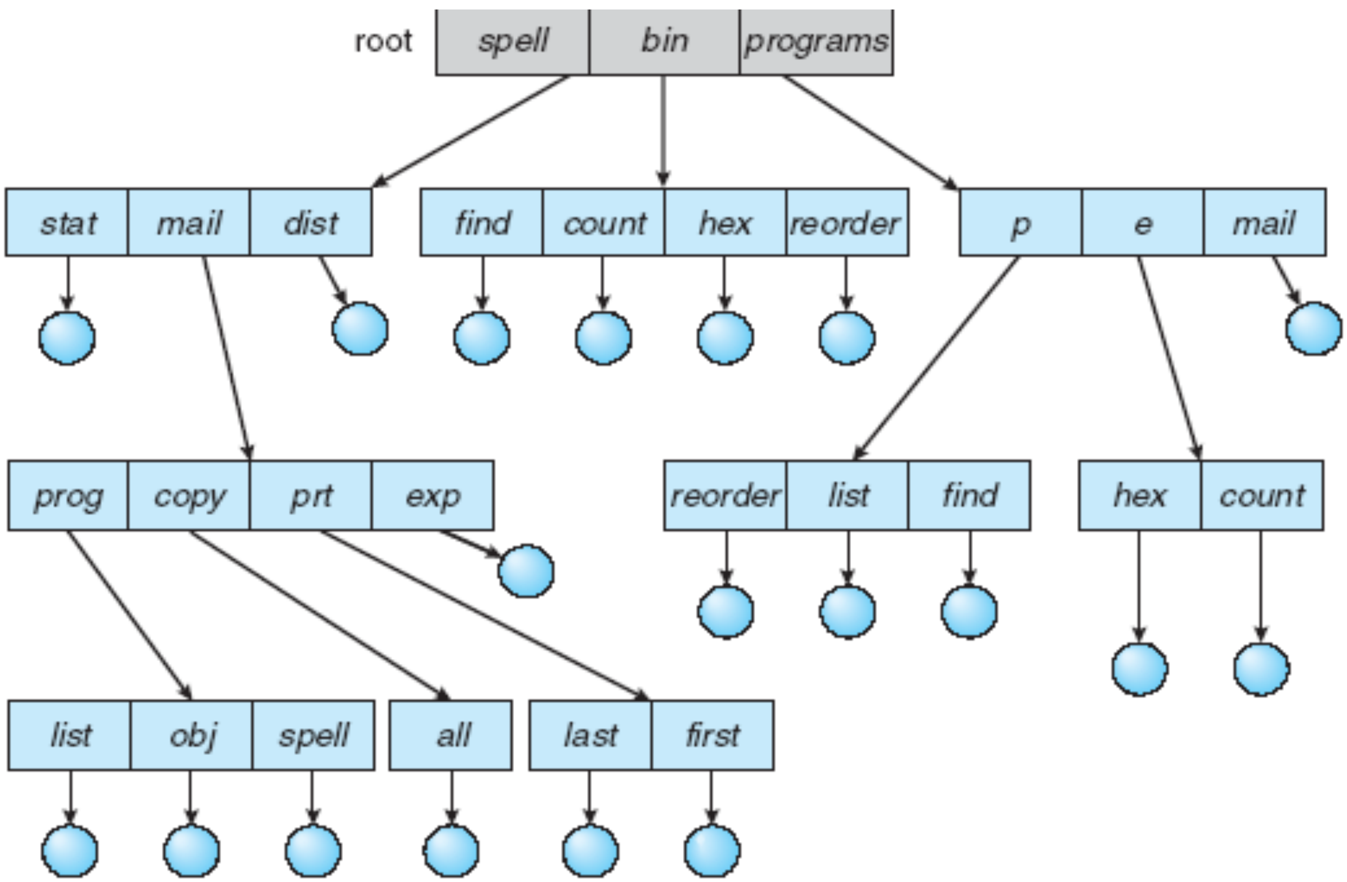
- Each user has a separate directory



- Solves name collision, but what if user has lots of files
- May not allow a user to access other users' files

Tree-structured Directory

- Directory is now a tree of arbitrary height
  - Directory contains files and subdirectories
  - A bit in directory entry differentiates files from subdirectories

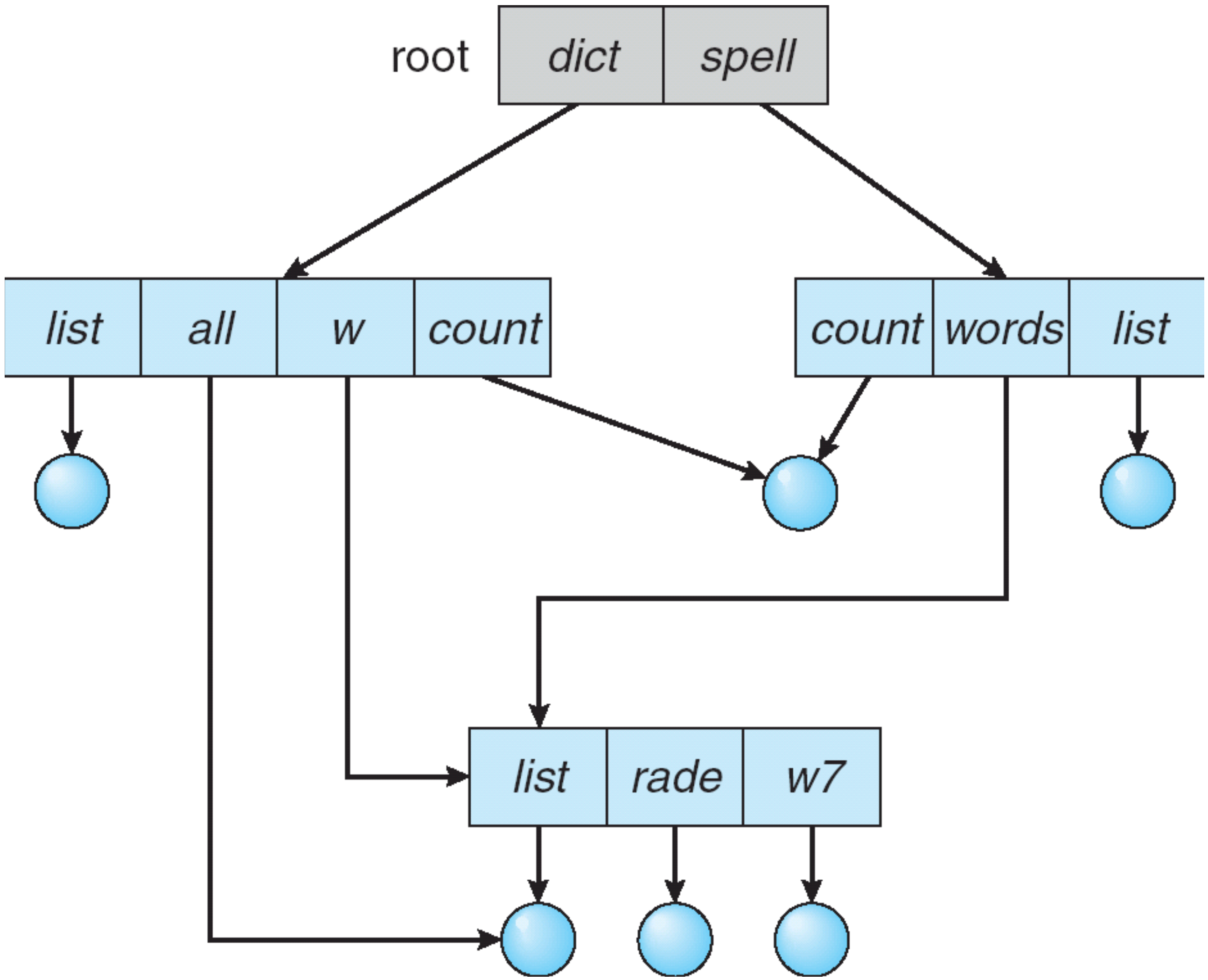


Path Names

- To access a file, the user should either:
  - Go to the directory where file resides, or
  - Specify the **path** where the file is
- Path names are either absolute or relative
  - Absolute: path of file from the root directory
  - Relative: path from the current working directory
- Most OSes have two special entries in each directory:
  - "." for current directory and ".." for parent

Acyclic Graph Directories

- Share subdirectories or files



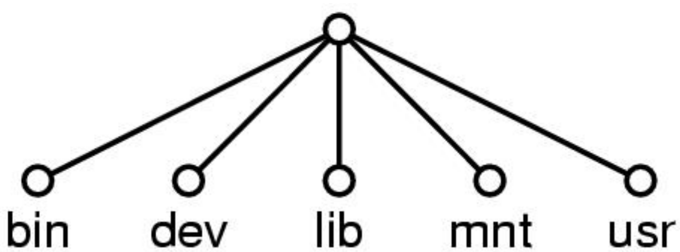
Acyclic Graph Directories

- How to implement shared files and subdirectories:
  - Why not copy the file?
  - New directory entry, called Link (used in UNIX)
    - Link is a pointer to another file or subdirectory
    - Links are ignored when traversing FS
    - *ln* in UNIX, *fsutil* in Windows for hard links
    - *ln –s* in UNIX, shortcuts in Windows for soft links
- Issues?
  - Two different names (aliasing)
  - If *dict* deletes *count* ⇒ dangling pointer
    - Keep backpointers of links for each file
    - Leave the link, and delete only when accessed later
    - Keep reference count of each file

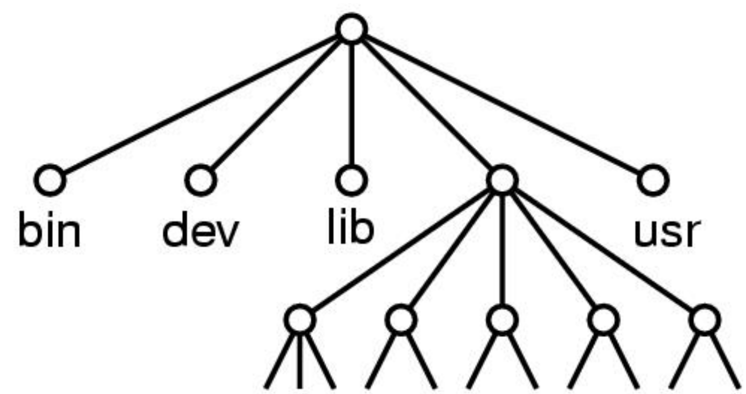
File System Mounting

- Mount allows two FSes to be merged into one
  - For example you insert your floppy into the root FS

mount("/dev/fd0", "/mnt", 0)



(a)



(b)

Remote file system mounting

- Same idea, but file system is actually on some other machine
- Implementation uses remote procedure call
  - Package up the user's file system operation
  - Send it to the remote machine where it gets executed like a local request
  - Send back the answer
- Very common in modern systems

File Protection

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

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

Categories of Users

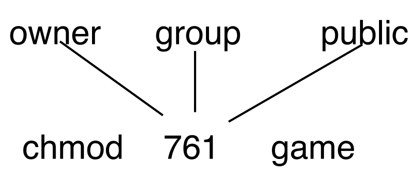
- Individual user
  - Log in establishes a user-id
  - Might be just local on the computer or could be through interaction with a network service
- Groups to which the user belongs
  - For example, “einar” is in “facres”
  - Again could just be automatic or could involve talking to a service that might assign, say, a temporary cryptographic key

Linux Access Rights

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

			RWX
a) **owner access**	7	⇒	1 1 1
			RWX
b) **group access**	6	⇒	1 1 0
			RWX
c) **public access**	1	⇒	0 0 1

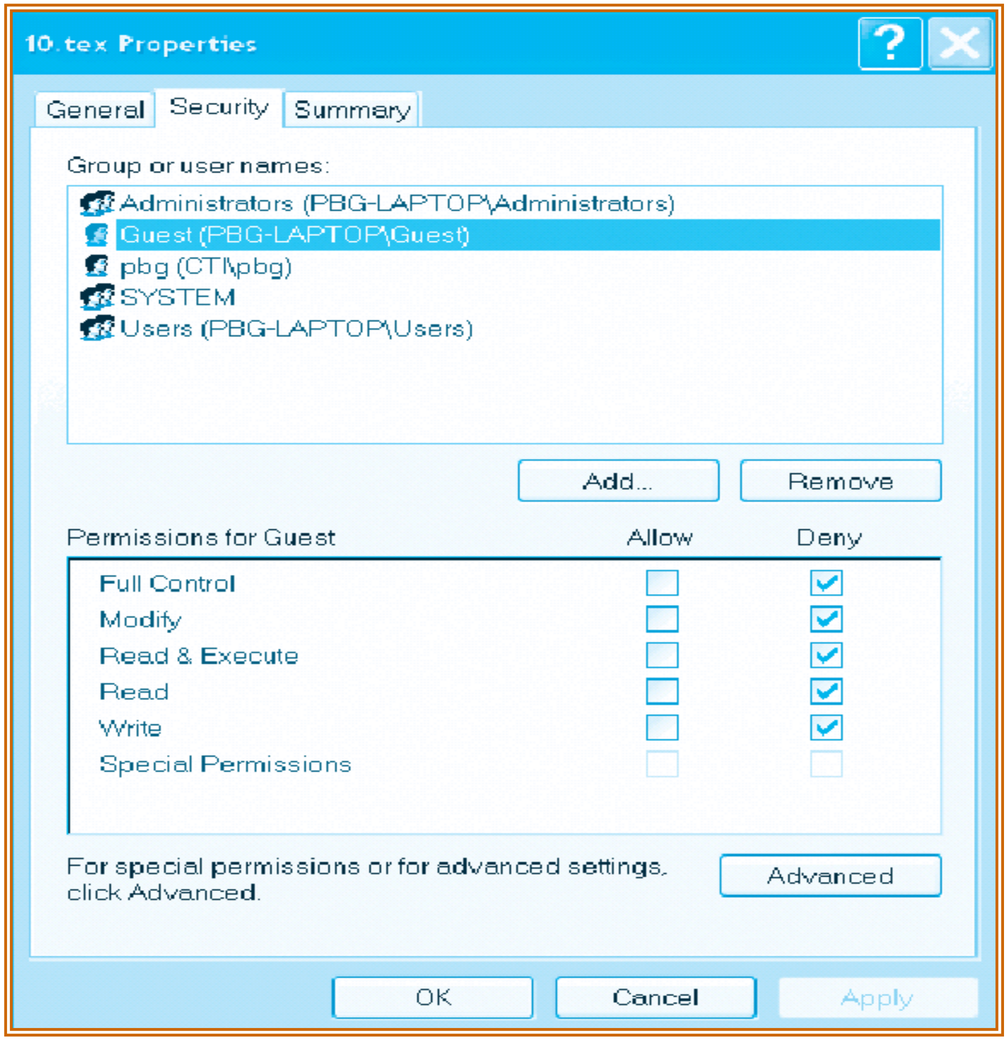
- For a particular file (say *game*) or subdirectory, define an appropriate access.



Issues with Linux

- Just a single owner, a single group and the public
  - Pro: Compact enough to fit in just a few bytes
  - Con: Not very expressive
- *Access Control List:* This is a per-file list that tells who can access that file
  - Pro: Highly expressive
  - Con: Harder to represent in a compact way

XP ACLs


10.tex Properties
General
Security
Summary
Group or user names:
Administrators (PBG-LAPTOP\Administrators)
Guest (PBG-LAPTOP\Guest)
pbg (CTI\pbg)
SYSTEM
Users (PBG-LAPTOP\Users)
Add...
Remove
Permissions for Guest
Allow
Deny
Full Control
Modify
Read & Execute
Read
Write
Special Permissions
For special permissions or for advanced settings, click Advanced.
Advanced
OK
Cancel
Apply

Security and Remote File Systems

- Recall that we can “mount” a file system
  - Local: File systems on multiple disks/volumes
  - Remote: A means of accessing a file system on some other machine
    - Local stub translates file system operations into messages, which it sends to a remote machine
    - Over there, a service receives the message and does the operation, sends back the result
    - Makes a remote file system look “local”

Unix Remote File System Security

- Since early days of Unix, NFS has had two modes
  - Secure mode: user, group-id's authenticated each time you boot from a network service that hands out temporary keys
  - Insecure mode: trusts your computer to be truthful about user and group ids
- Most NFS systems run in *insecure* mode!
  - Because of US restrictions on exporting cryptographic code…

Spoofing

- Question: what stops you from "spoofing" by building NFS packets of your own that lie about id?
- Answer?
  - In insecure mode… nothing!
  - In fact people have written this kind of code
  - Many NFS systems are wide open to this form of attack, often only the firewall protects them

File System Implementation

- How exactly are file systems implemented?
  - Comes down to: how do we represent
    - Volumes/partitions
    - Directories (link file names to file “structure”)
    - The list of blocks containing the data
    - Other information such as access control list or permissions, owner, time of access, etc?
  - And, can we be smart about layout?

Implementing File Operations

- Create a file:
  - Find space in the file system, add directory entry.
- Writing in a file:
  - System call specifying name & information to be written. Given name, system searches directory structure to find file. System keeps ***write pointer*** to the location where next write occurs, updating as writes are performed
- Reading a file:
  - System call specifying name of file & where in memory to stick contents. Name is used to find file, and a ***read pointer*** is kept to point to next read position. (can combine write & read to ***current file position pointer***)
- Repositioning within a file:
  - Directory searched for appropriate entry & current file position pointer is updated (also called a file ***seek***)

Implementing File Operations

- Deleting a file:
  - Search directory entry for named file, release associated file space and erase directory entry
- Truncating a file:
  - Keep attributes the same, but reset file size to 0, and reclaim file space.

Other file operations

- Most FS require an open() system call before using a file.
- OS keeps an in-memory table of open files, so when reading a writing is requested, they refer to entries in this table.
- On finishing with a file, a close() system call is necessary. (creating & deleting files typically works on closed files)
- What happens when multiple files can open the file at the same time?

Multiple users of a file

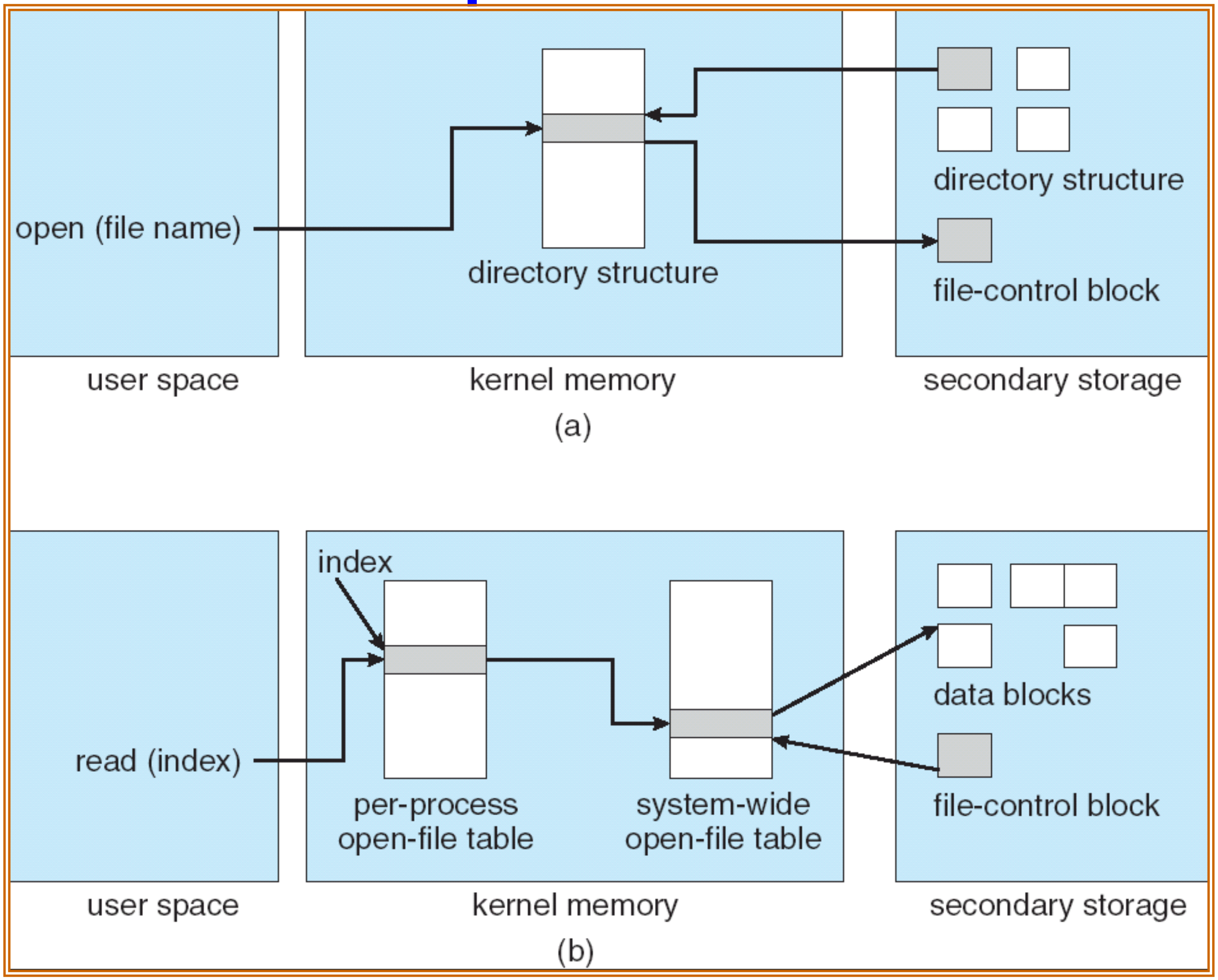
- OS typically keeps two levels of internal tables:
- Per-process table
  - Information about the use of the file by the user (e.g. current file position pointer)
- System wide table
  - Gets created by first process which opens the file
  - Location of file on disk
  - Access dates
  - File size
  - Count of how many processes have the file open (used for deletion)

The File Control Block (FCB)

- FCB has all the information about the file
  - Linux systems call these *inode* structures

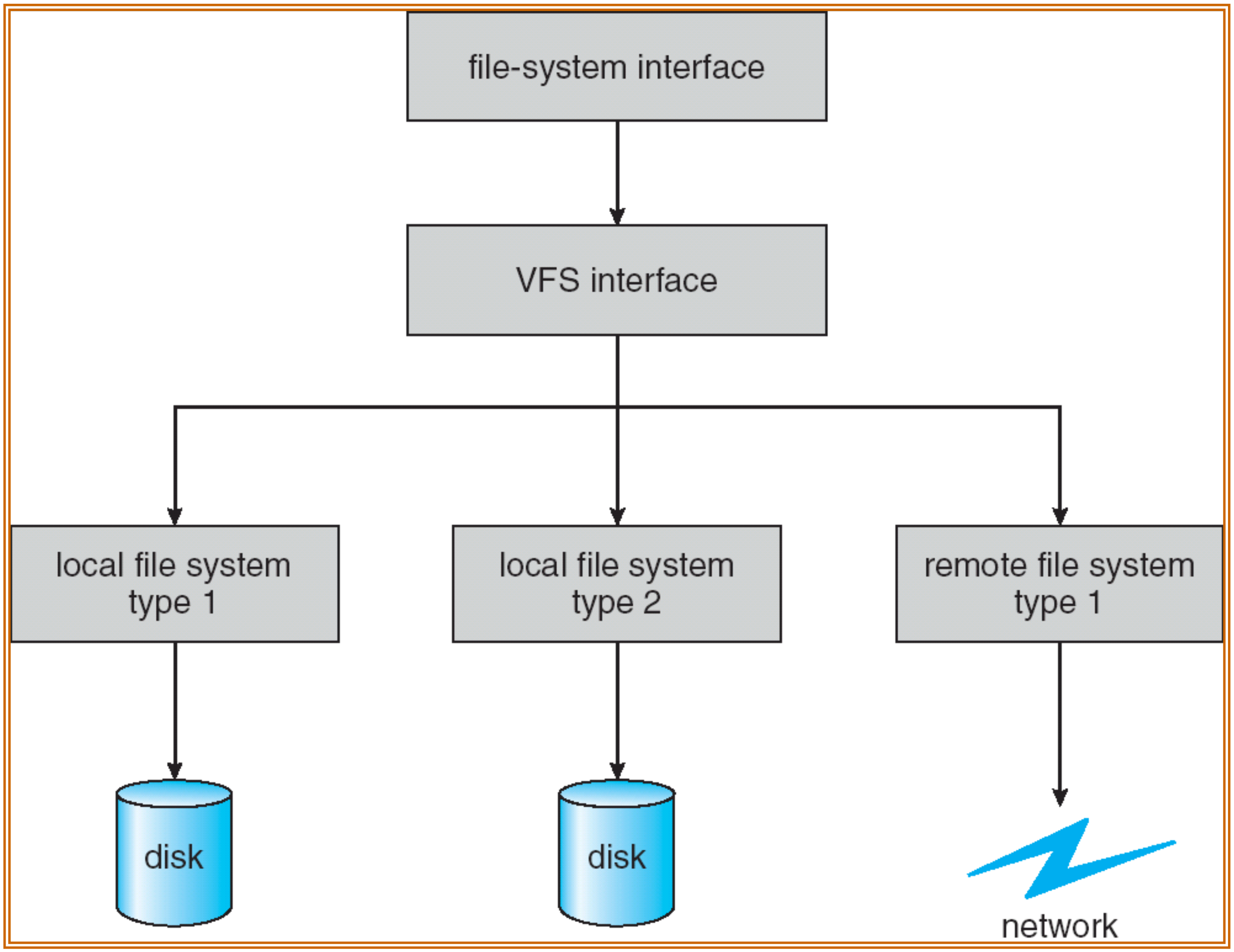
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Files Open and Read



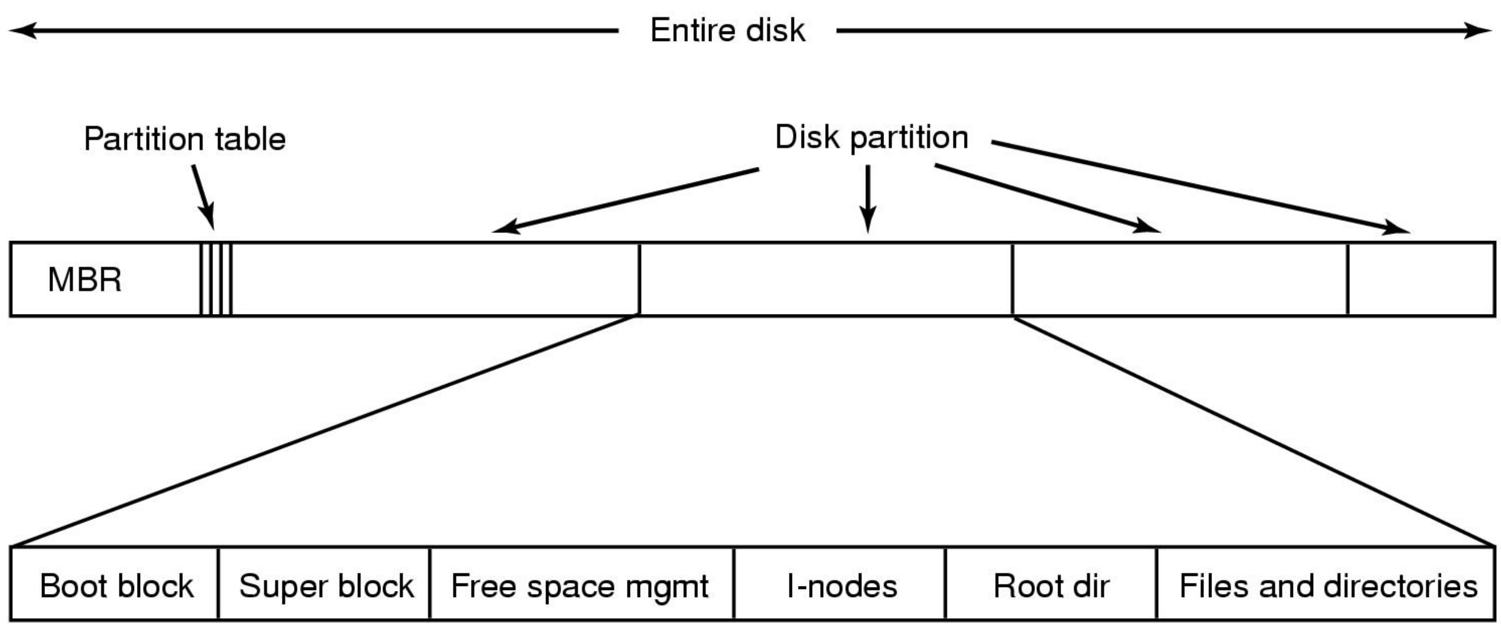
Virtual File Systems

- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.


file-system interface
VFS interface
local file system
type 1
local file system
type 2
remote file system
type 1
disk
disk
network

File System Layout

- File System is stored on disks
  - Disk is divided into 1 or more partitions
  - Sector 0 of disk called Master Boot Record
  - End of MBR has partition table (start & end address of partitions)
- First block of each partition has boot block
  - Loaded by MBR and executed on boot

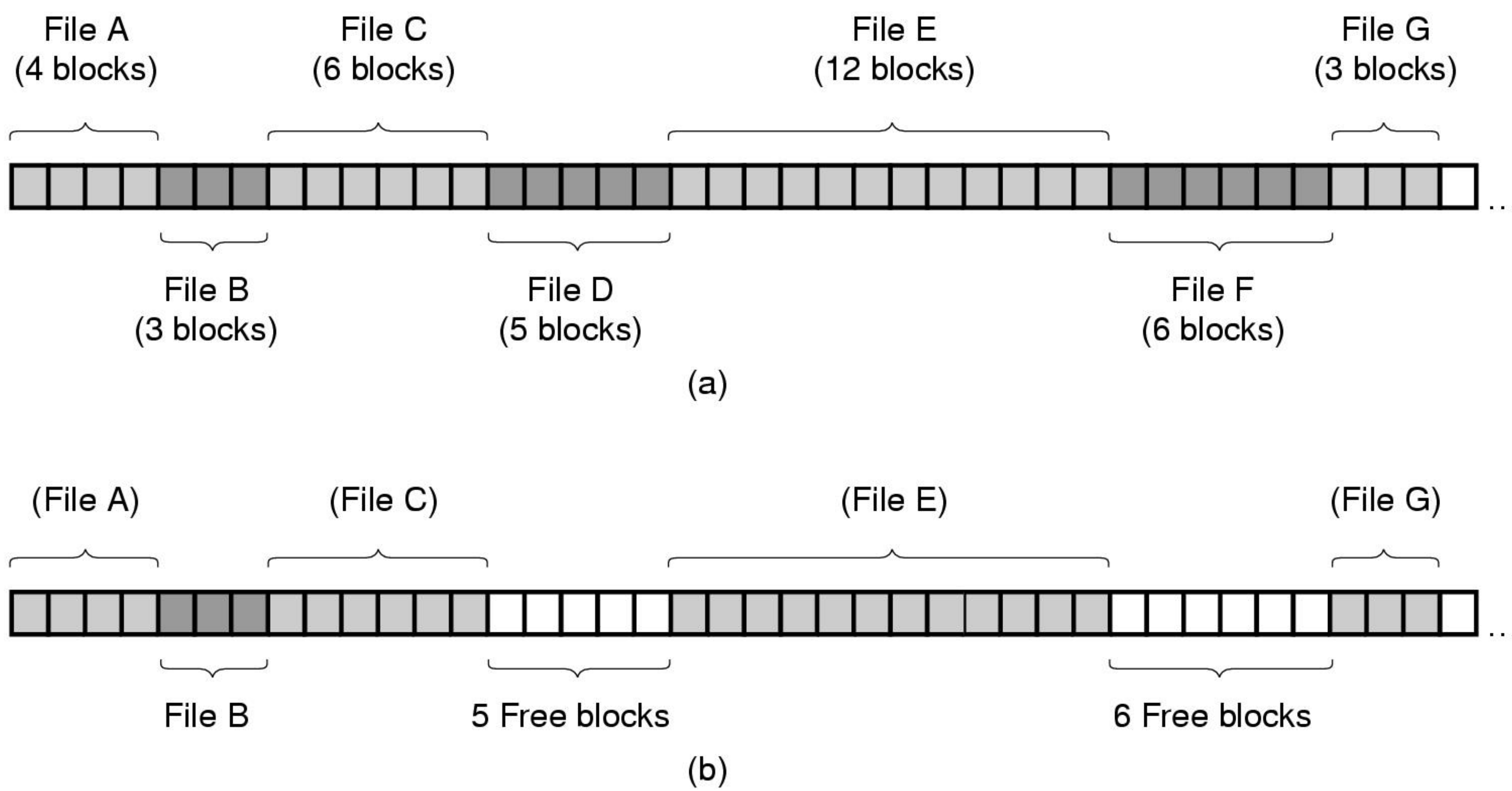


Storing Files

- Files can be allocated in different ways:
  - Contiguous allocation
    - All bytes together, in order
  - Linked Structure
    - Each block points to the next block
  - Indexed Structure
    - An index block contains pointer to many other blocks
  - Rhetorical Questions -- which is best?
    - For sequential access? Random access?
    - Large files? Small files? Mixed?

Contiguous Allocation

- Allocate files contiguously on disk

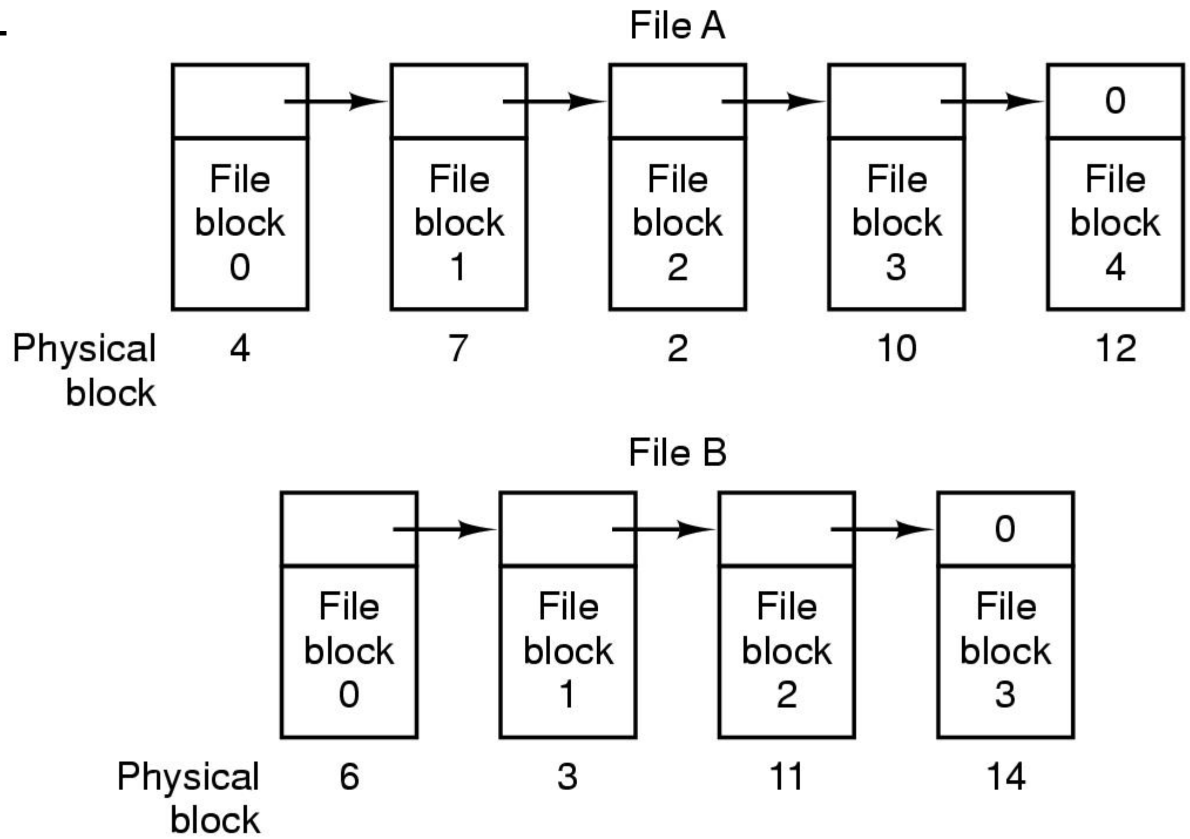


Contiguous Allocation

- Pros:
  - Simple: state required per file is start block and size
  - Performance: entire file can be read with one seek
- Cons:
  - Fragmentation: external is bigger problem
  - Usability: user needs to know size of file
- Used in CDROMs, DVDs

Linked List Allocation

- Each file is stored as linked list of blocks
  - First word of each block points to next block
  -

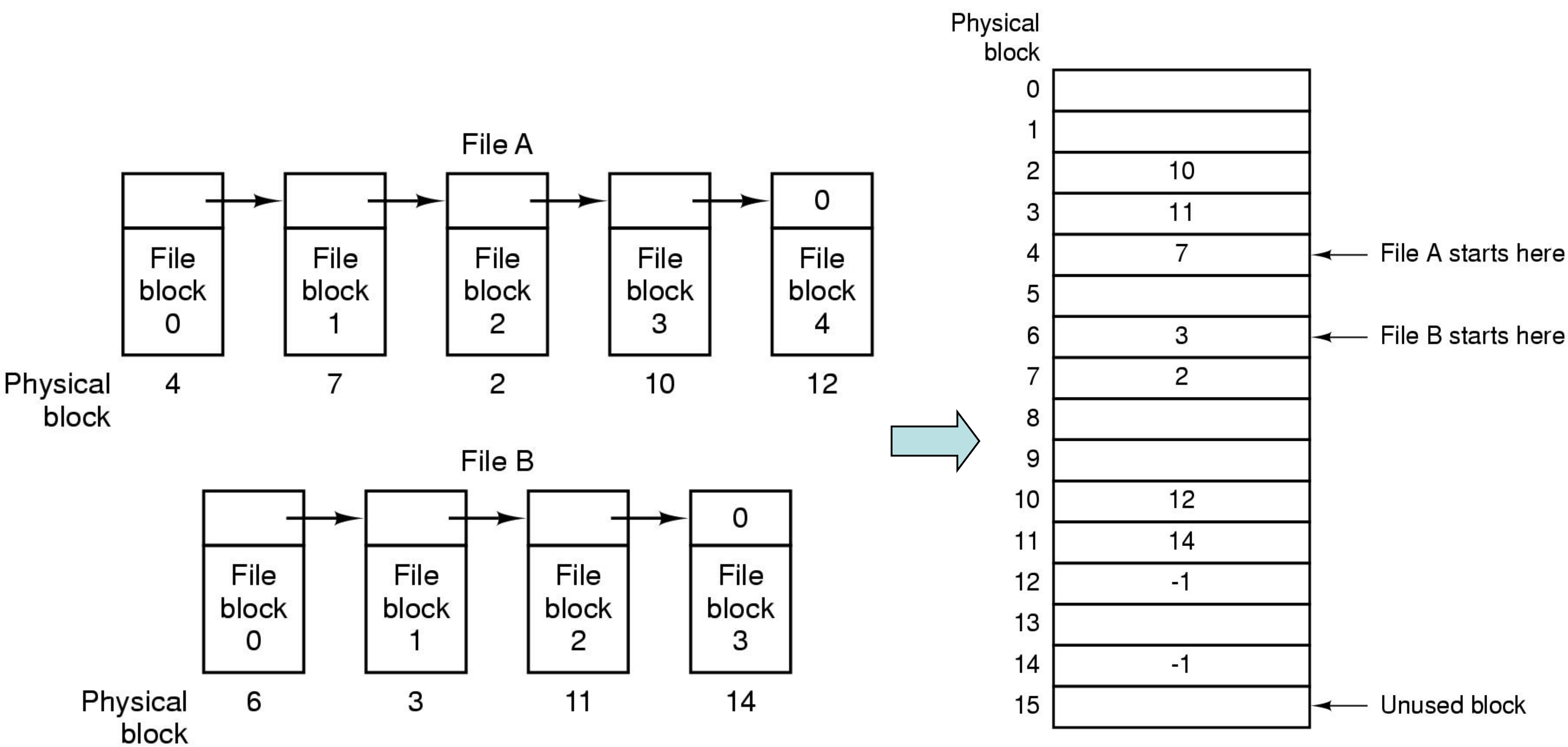


Linked List Allocation

- Pros:
  - No space lost to external fragmentation
  - Disk only needs to maintain first block of each file
- Cons:
  - Random access is costly
  - Overheads of pointers.

MS-DOS file system

- Implement a linked list allocation using a table
  - Called File Allocation Table (FAT)
  - Take pointer away from blocks, store in this table

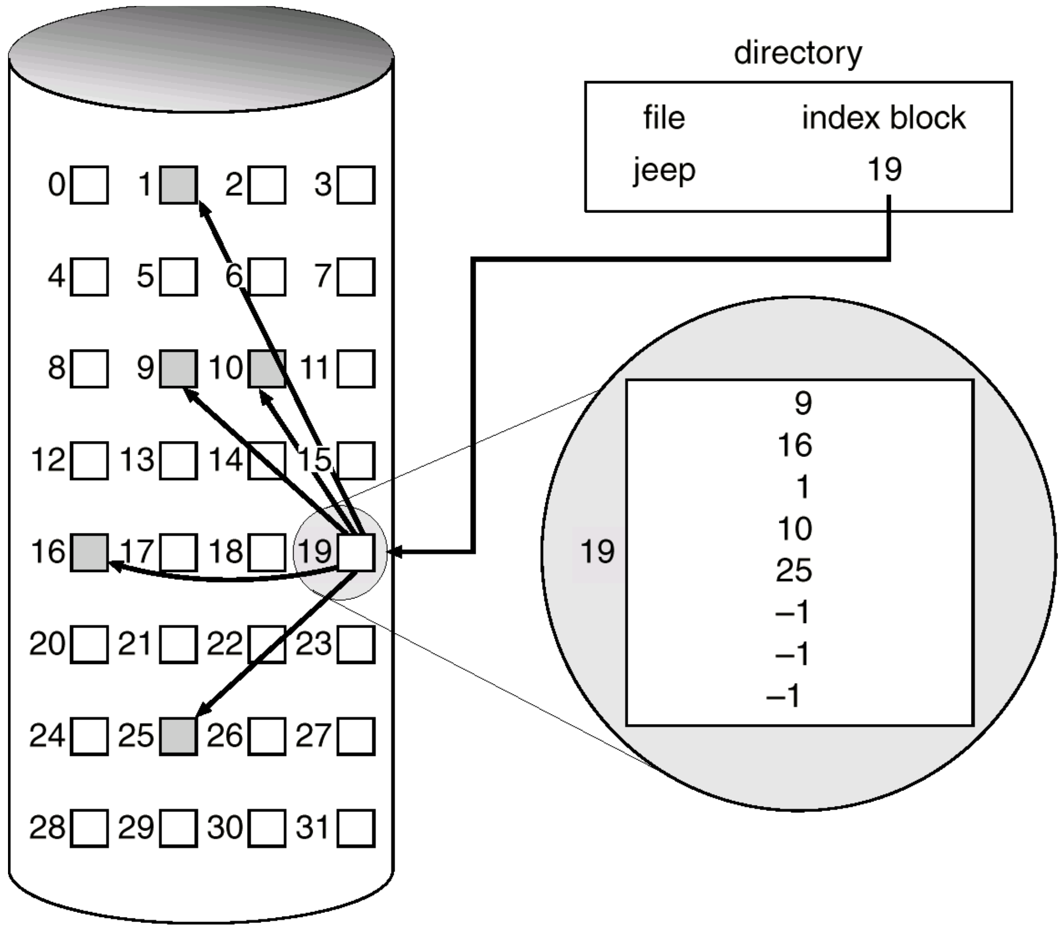


FAT Discussion

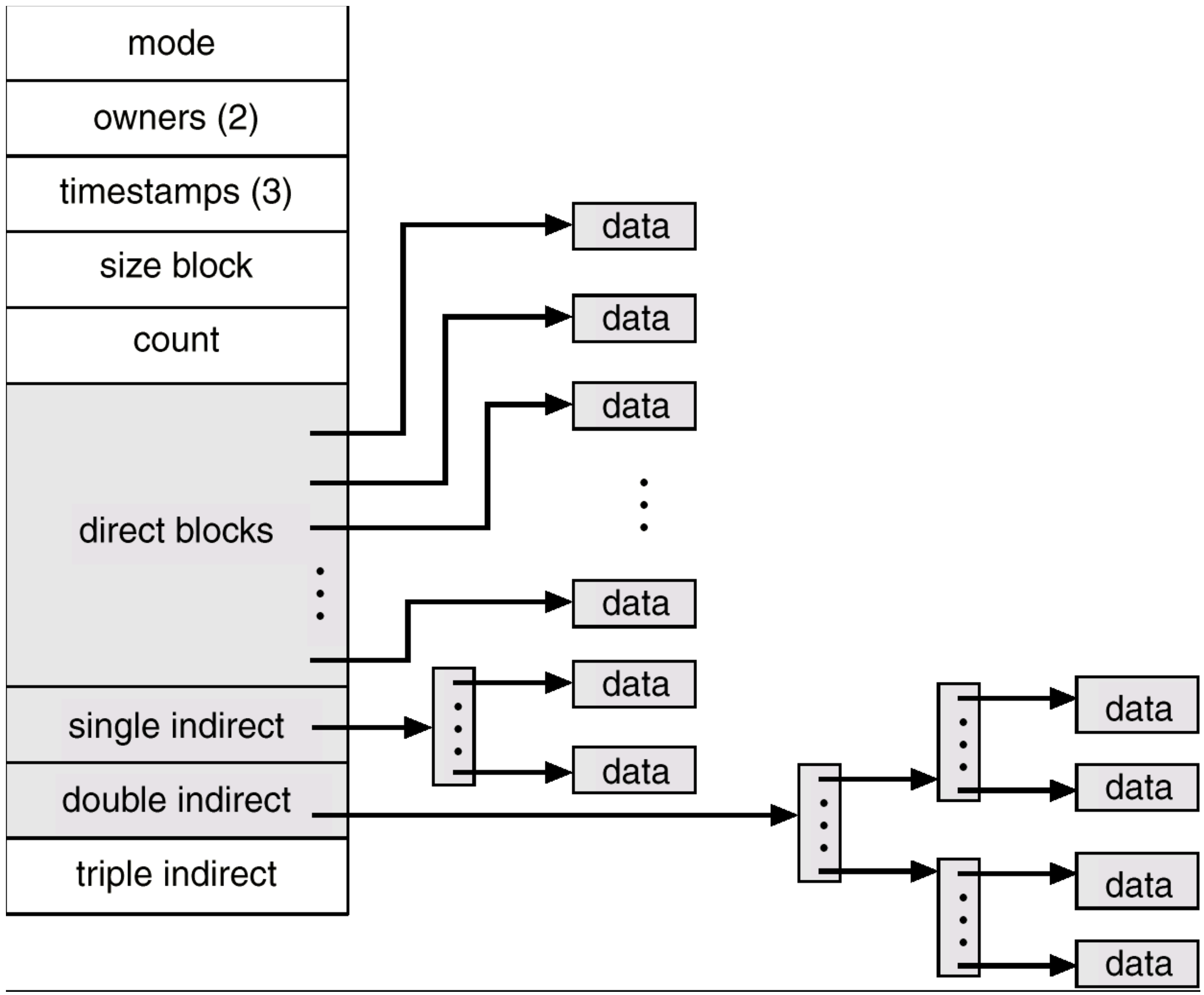
- Pros:
  - Entire block is available for data
  - Random access is faster than linked list.
- Cons:
  - Many file seeks unless entire FAT is in memory
    - For 20 GB disk, 1 KB block size, FAT has 20 million entries
    - If 4 bytes used per entry $\Rightarrow$ 80 MB of main memory required for FS

Indexed Allocation

- Index block contains pointers to each data block
- Pros?
- Cons?



UFS - Unix File System

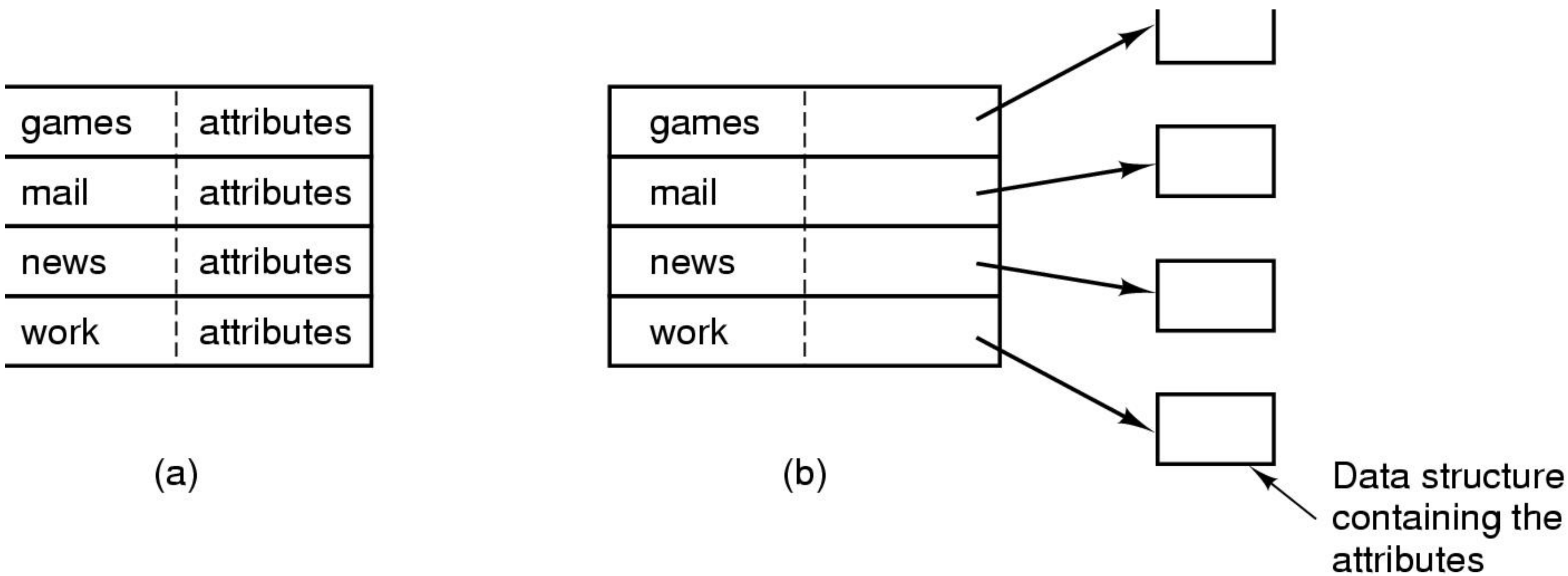


Unix inodes

- If data blocks are 4K …
  - First 48K reachable from the inode
  - Next 4MB available from single-indirect
  - Next 4GB available from double-indirect
  - Next 4TB available through the triple-indirect block
- Any block can be found with at most 3 disk accesses

Implementing Directories

- When a file is opened, OS uses path name to find dir
  - Directory has information about the file's disk blocks
    - Whole file (contiguous), first block (linked-list) or I-node
  - Directory also has attributes of each file
- Directory: map ASCII file name to file attributes & location
- 2 options: entries have all attributes, or point to file I-node

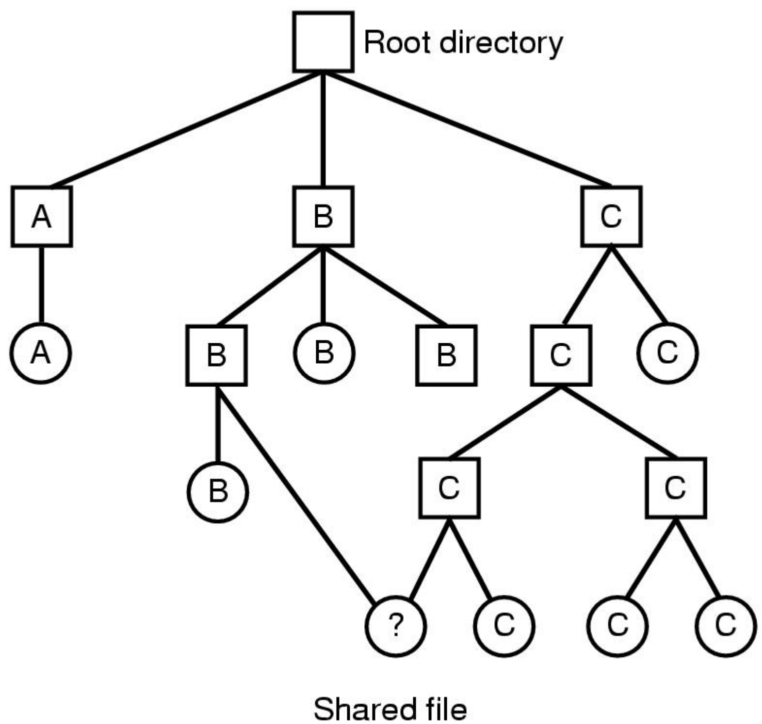


Directory Search

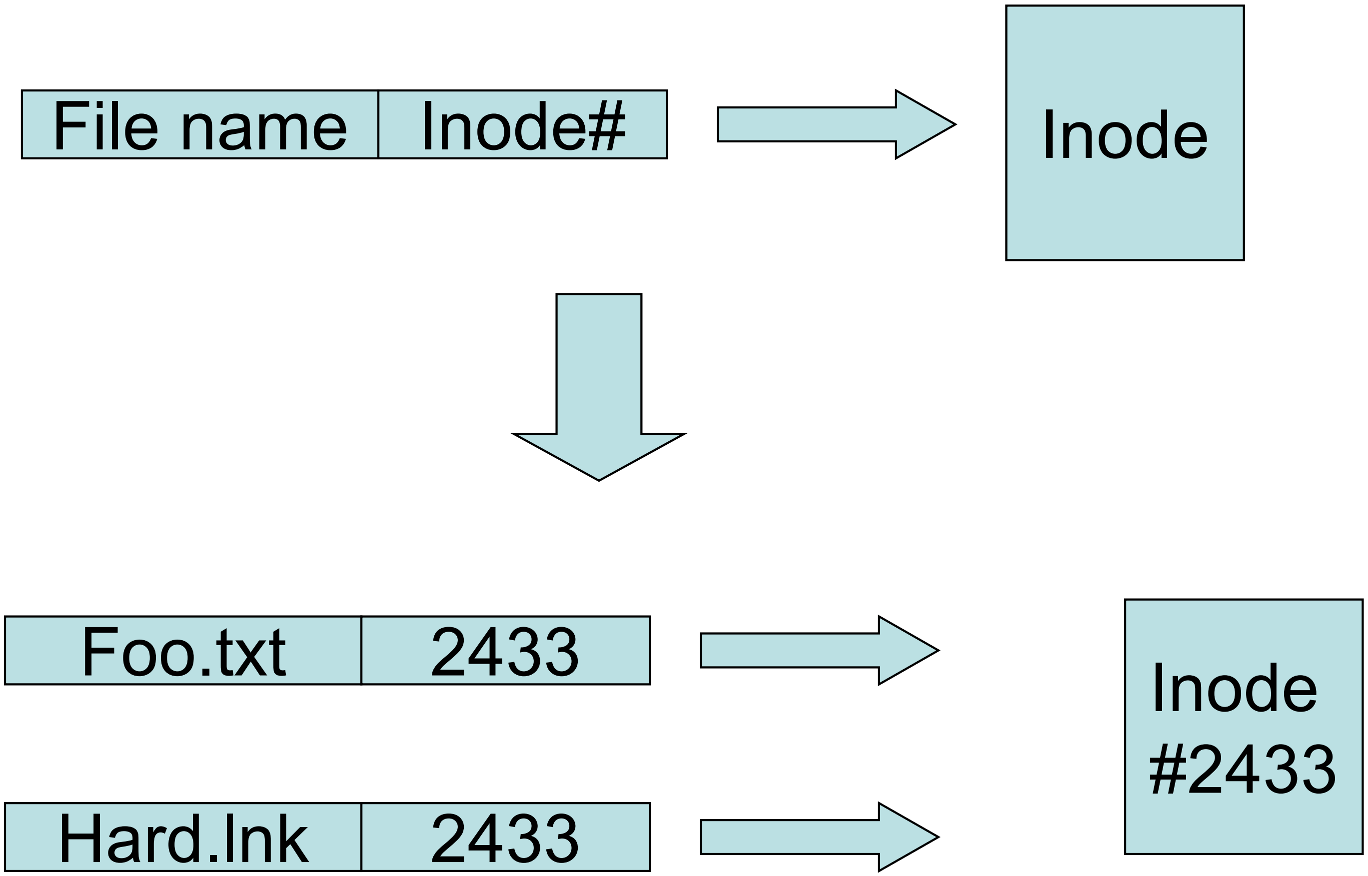
- Simple Linear search can be slow
- Alternatives:
  - Use a per-directory hash table
    - Could use hash of file name to store entry for file
    - Pros: faster lookup
    - Cons: More complex management
  - Caching: cache the most recent searches
    - Look in cache before searching FS

Shared Files

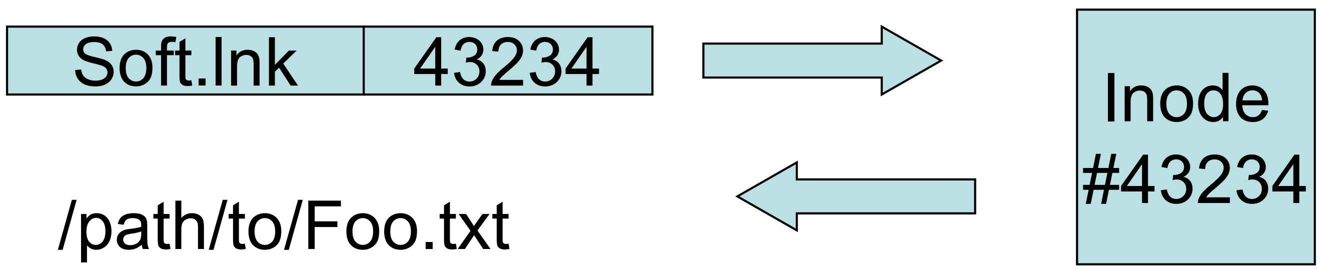
- If B wants to share a file owned by C
  - One Solution: copy disk addresses in B's directory entry
  - Problem: modification by one not reflected in other user's view



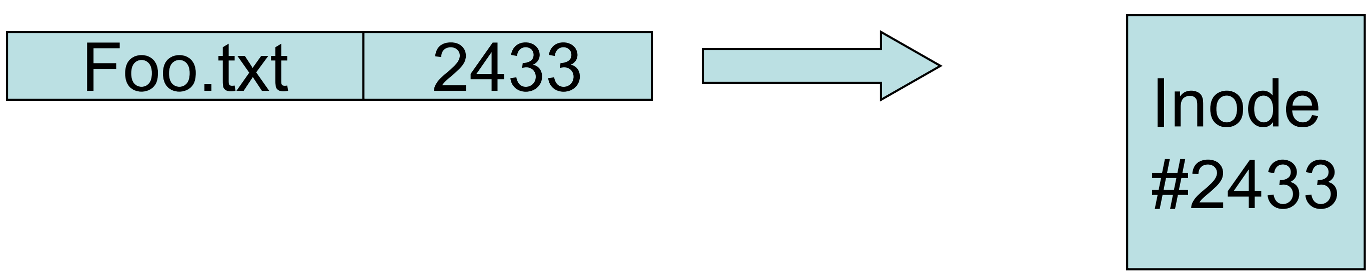
Hard vs Soft Links



Hard vs Soft Links

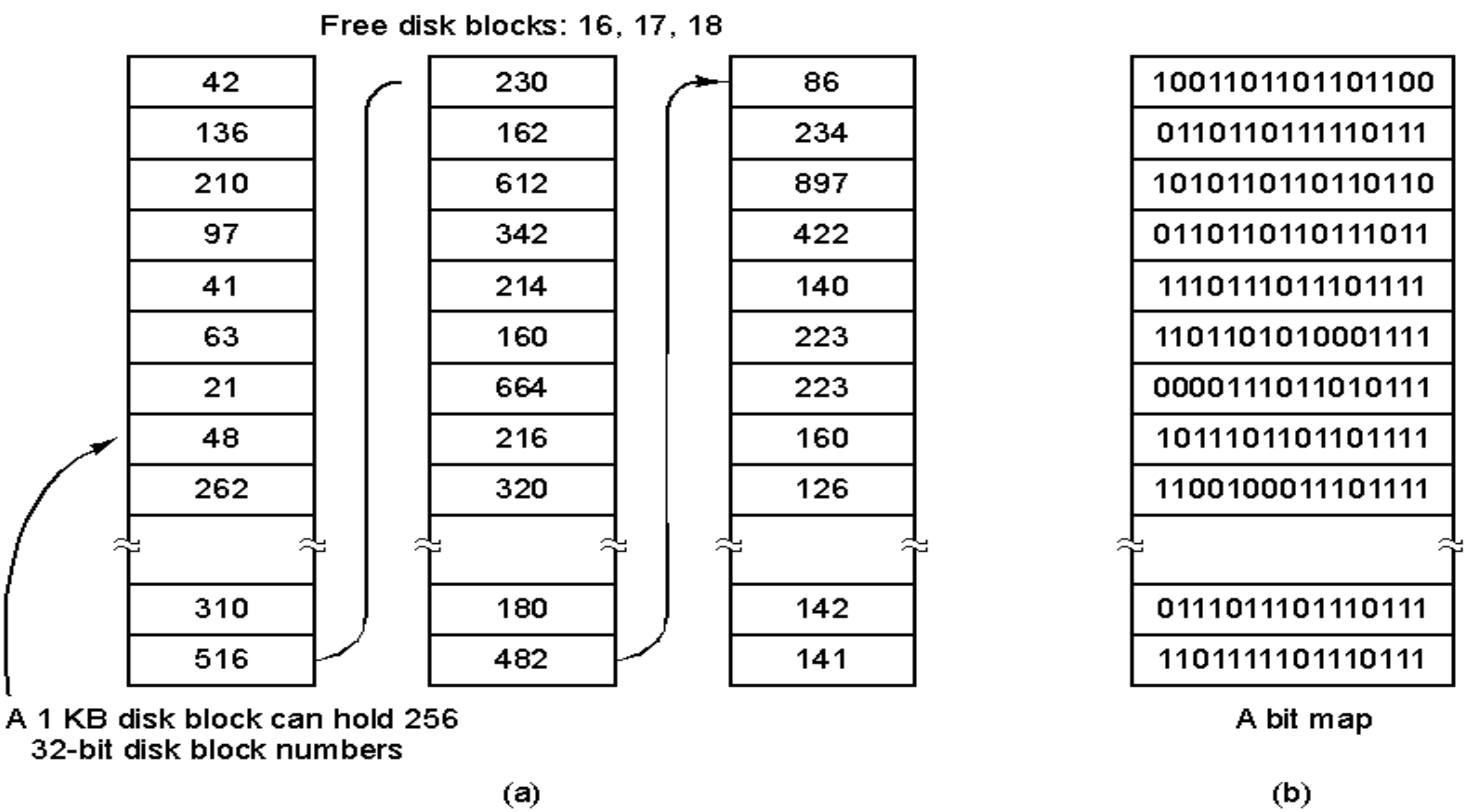


..and then redirects to Inode #2433 at open() time..



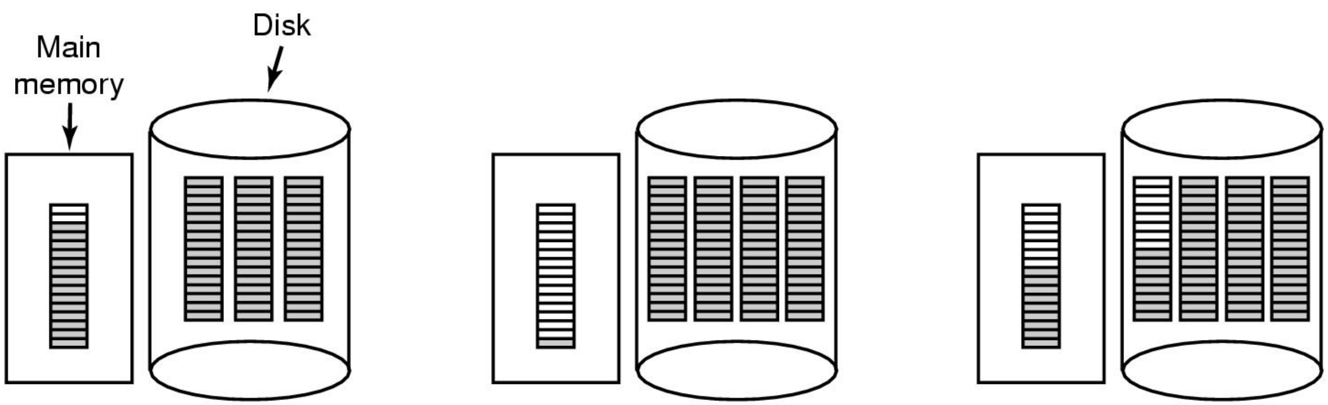
Managing Free Disk Space

- 2 approaches to keep track of free disk blocks



Tracking free space

- Storing free blocks in a Linked List
  - Only one block need to be kept in memory
  - Bad scenario: Solution (c)

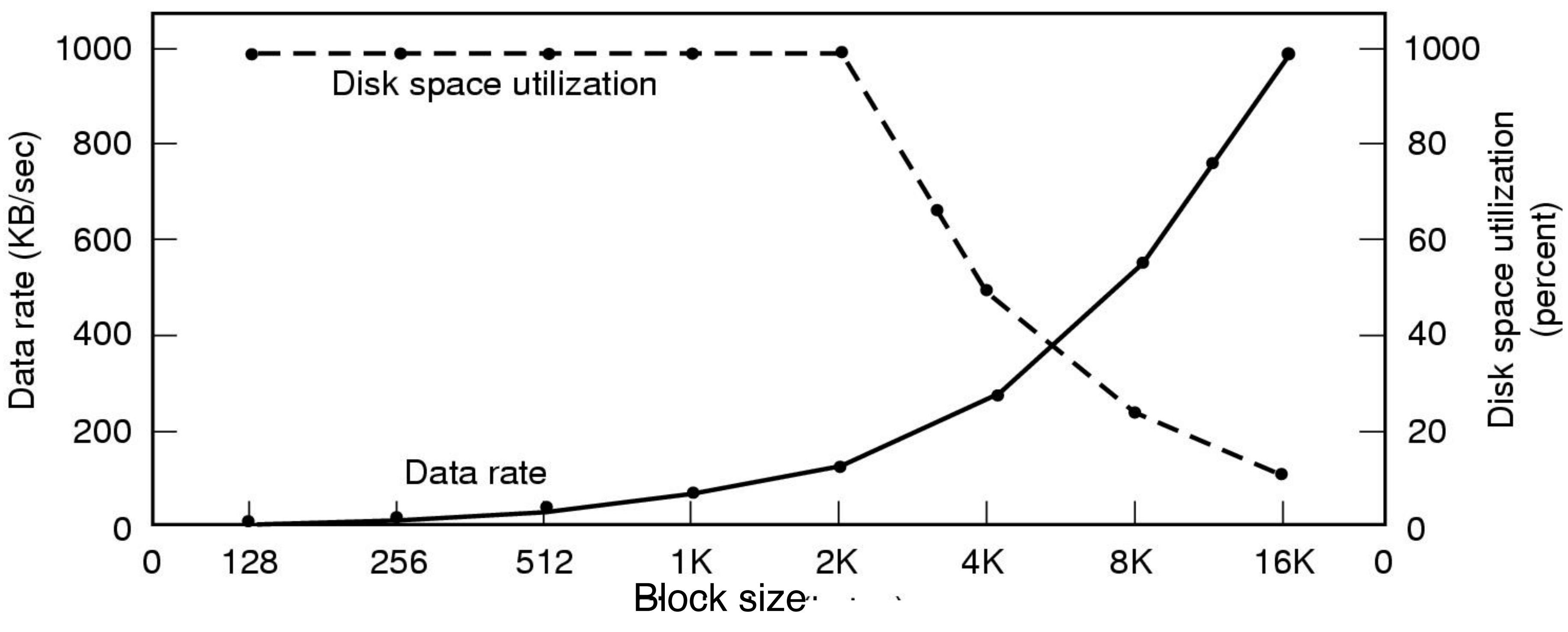


(a) (b) (c)

- Storing bitmaps
  - Lesser storage in most cases
  - Allocated disk blocks are closer to each other

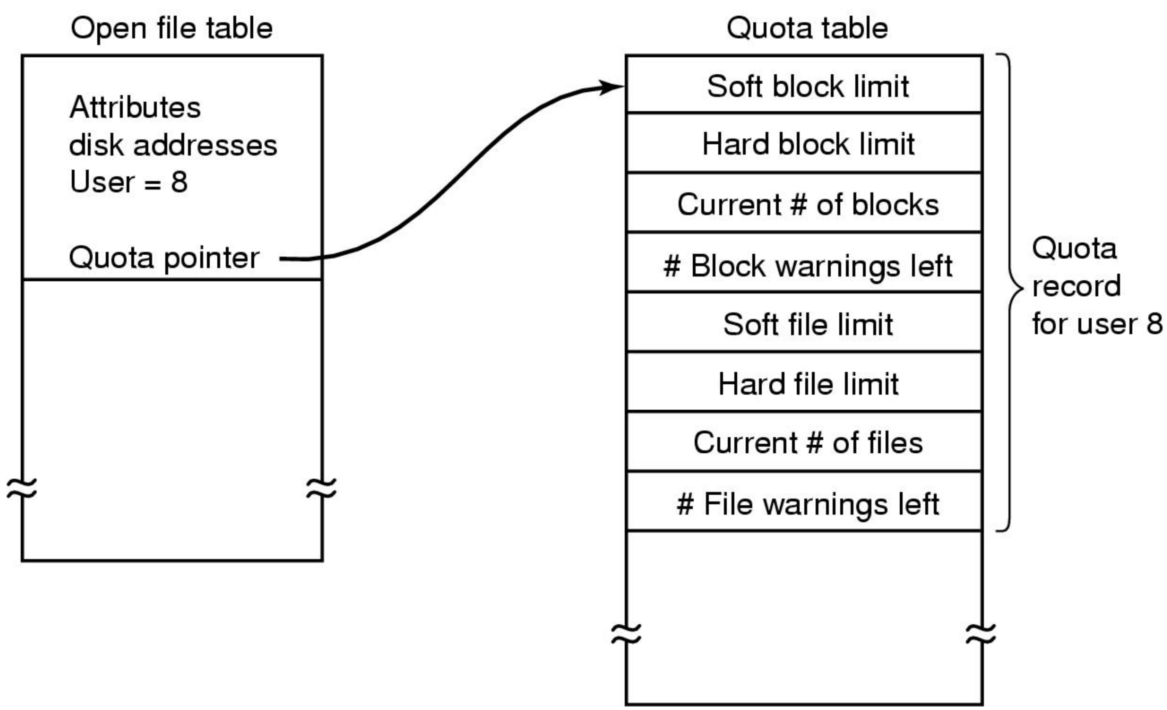
Disk Space Management

- Files stored as fixed-size blocks
- What is a good block size? (sector, track, cylinder?)
  - If 131,072 bytes/track, rotation time 8.33 ms, seek time 10 ms
  - To read k bytes block: 10+ 4.165 + (k/131072)*8.33 ms
  - Median file size: 2 KB



Managing Disk Quotas

- Sys admin gives each user max space
  - Open file table has entry to Quota table
  - Soft limit violations result in warnings
  - Hard limit violations result in errors
  - Check limits on login



Efficiency and Performance

- Efficiency dependent on:
  - disk allocation and directory algorithms
  - types of data kept in file's directory entry
- Performance
  - disk cache – separate section of main memory for frequently used blocks
  - free-behind and read-ahead – techniques to optimize sequential access
  - improve PC performance by dedicating section of memory as virtual disk, or RAM disk

File System Consistency

- System crash before modified files written back
  - Leads to inconsistency in FS
  - fsck (UNIX) & scandisk (Windows) check FS consistency
- Algorithm:
  - Build 2 tables, each containing counter for all blocks (init to 0)
    - 1st table checks how many times a block is in a file
    - 2nd table records how often block is present in the free list
      - >1 not possible if using a bitmap
  - Read all i-nodes, and modify table 1
  - Read free-list and modify table 2
  - Consistent state if block is either in table 1 or 2, but not both

A changing problem

- Consistency used to be very hard
  - Problem was that driver implemented C-SCAN and this could reorder operations
  - For example
    - Delete file X in inode Y containing blocks A, B, C
    - Now create file Z re-using inode Y and block C
  - Problem is that if I/O is out of order and a crash occurs we could see a scramble
    - E.g. C in both X and Z… or directory entry for X is still there but points to inode now in use for file Z

Inconsistent FS examples

(a) Consistent

(b) missing block 2: add it to free list

(c) Duplicate block 4 in free list: rebuild free list

(d) Duplicate block 5 in data list: copy block and add it to one file

Block number

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

(a)

Block number

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1

(b)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	0	2	0	0	0	0	1	1	0	0	0	1	1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	2	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

Check Directory System

- Use a per-file table instead of per-block
- Parse entire directory structure, starting at the root
  - Increment the counter for each file you encounter
  - This value can be >1 due to hard links
  - Symbolic links are ignored
- Compare counts in table with link counts in the i-node
  - If i-node count > our directory count (wastes space)
  - If i-node count < our directory count (catastrophic)

Log Structured File Systems

- **Log structured** (or journaling) file systems record each update to the file system as a **transaction**
- All transactions are written to a **log**
  - A transaction is considered **committed** once it is written to the log
  - However, the file system may not yet be updated

Log Structured File Systems

- The transactions in the log are asynchronously written to the file system
  - When the file system is modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- E.g. ReiserFS, XFS, NTFS, etc..

FS Performance

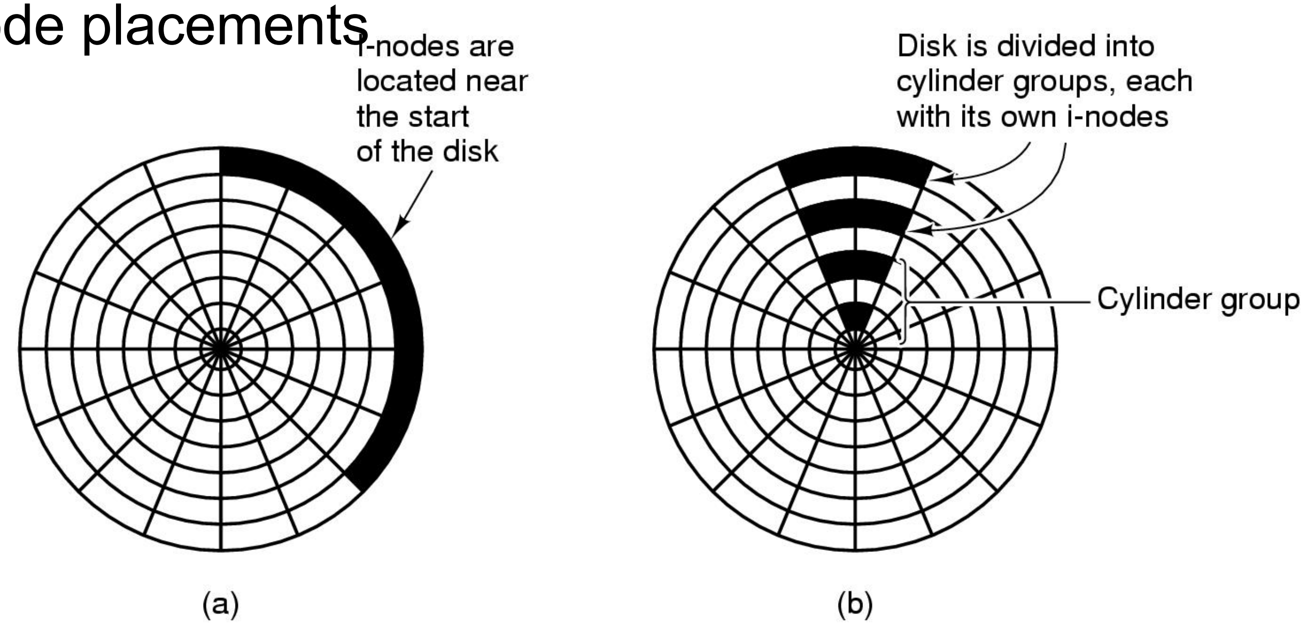
- Access to disk is much slower than access to memory
  - Optimizations needed to get best performance
- 3 possible approaches: caching, prefetching, disk layout
- Block or buffer cache:
  - Read/write from and to the cache.

Block Cache Replacement

- Which cache block to replace?
  - Could use any page replacement algorithm
  - Possible to implement perfect LRU
    - Since much lesser frequency of cache access
    - Move block to front of queue
  - Perfect LRU is undesirable. We should also answer:
    - Is the block essential to consistency of system?
    - Will this block be needed again soon?
- When to write back other blocks?
  - *Update* daemon in UNIX calls *sync* system call every 30 s
  - MS-DOS uses write-through caches

Other Approaches

- Pre-fetching or Block Read Ahead
  - Get a block in cache before it is needed (e.g. next file block)
  - Need to keep track if access is sequential or random
- Reducing disk arm motion
  - Put blocks likely to be accessed together in same cylinder
    - Easy with bitmap, possible with over-provisioning in free lists
  - Modify i-node placements



Storage Area Networks (SANs)

- New generation of architectures for managing storage in massive data centers
  - For example, Google is said to have 50,000-200,000 computers in various centers
  - Amazon is reaching a similar scale
- A SAN system is a collection of file systems with tools to help humans administer the system

Examples of SAN issues

- Where should a file be stored
  - Many of these systems have an indirection mechanism so that a file can move from volume to volume
  - Allows files to migrate, e.g. from a slow server to a fast one or from long term storage onto an active disk system
- Eco-computing: systems that seek to minimize energy in big data centers

Examples of SAN issues

- Disk-to-disk backup
  - Might want to do very fast automated backups
  - Ideally, can support this while the disk is actively in use
- Easiest if two disks are next to each other
- Challenge: back up entire data center in New York at site in Kentucky
  - US Dept of Treasury e-Cavern