File Systems

Profs. Bracy and Van Renesse

based on slides by Prof. Sirer

Storing Information

- Applications could store information in the process address space
- Why is this a bad idea?
  - Size is limited to size of virtual address space
  - The data is lost when the application terminates
    - Even when computer doesn’t crash!
  - Multiple process might want to access the same data

File Systems

- 3 criteria for long-term information storage:
  1. Able to store very large amount of information
  2. Information must survive the processes using it
  3. Provide concurrent access to multiple processes
- Solution:
  – Store information on disks in units called **files**
  – Files are persistent, only owner can 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
    - Windows names not case sensitive, UNIX family is

File Extensions

- Name divided into 2 parts: Name+Extension
- On UNIX, extensions are not enforced by OS
  - Some applications might insist upon them
    - *Think:* .c, .h, .o, .s, *etc.* for C compiler
- Windows attaches meaning to extensions
  - Tries to associate applications to file extensions

File Access

- Sequential access
  - read all bytes/records from the beginning
  - particularly convenient for 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 (#) 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

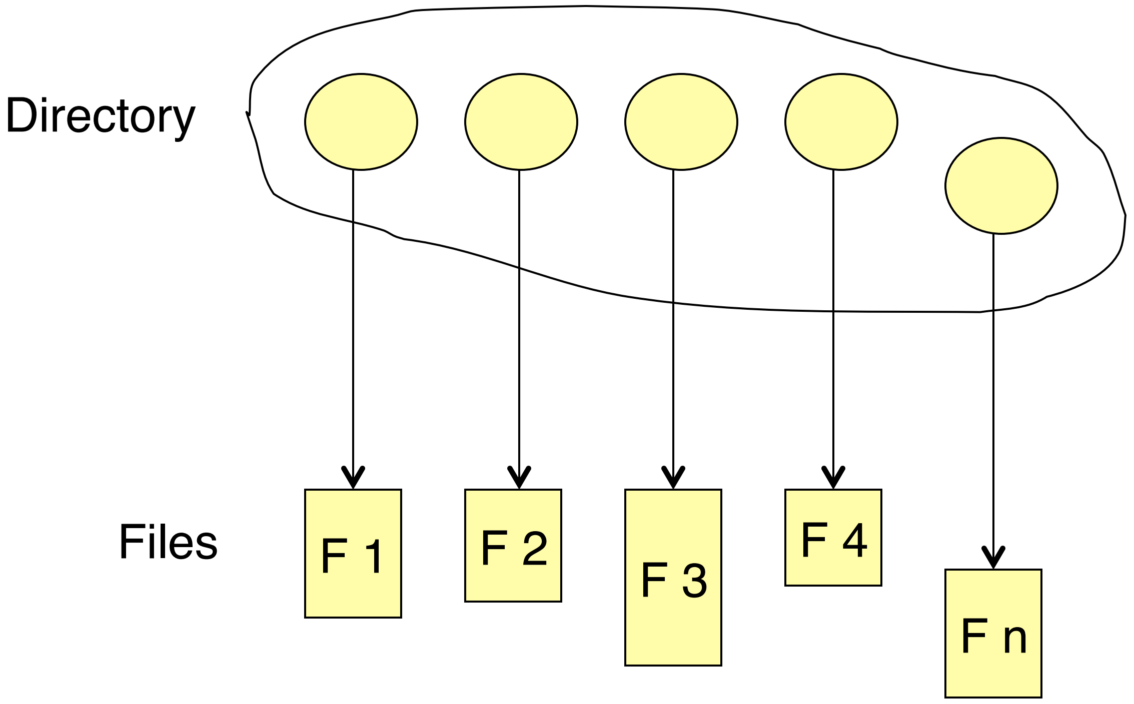
- Create a file
- Write to a file
- Read from a file
- Seek to somewhere in a file
- Delete a file
- Truncate a file

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 / paging
- Disk divided into *partitions*
  - Chunk of storage that holds a FS is called a *volume*

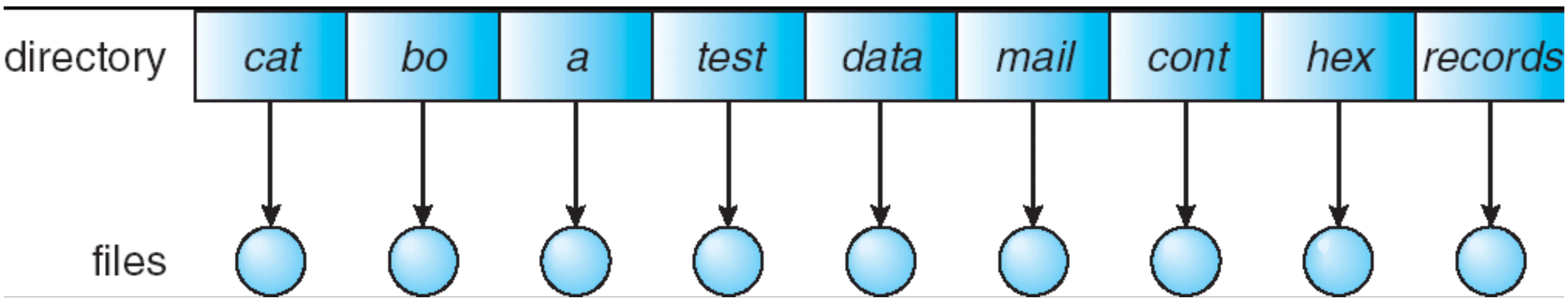
Directory

- Directory keeps track of files
  – Is a symbol table that translates file names to directory entries
  – Usually are themselves files
- How to structure directory to optimize all of:
  – Search a file
  – Create a file
  – Delete a file
  – List directory
  – Rename a file
  – Traversing the FS



Single-level Directory

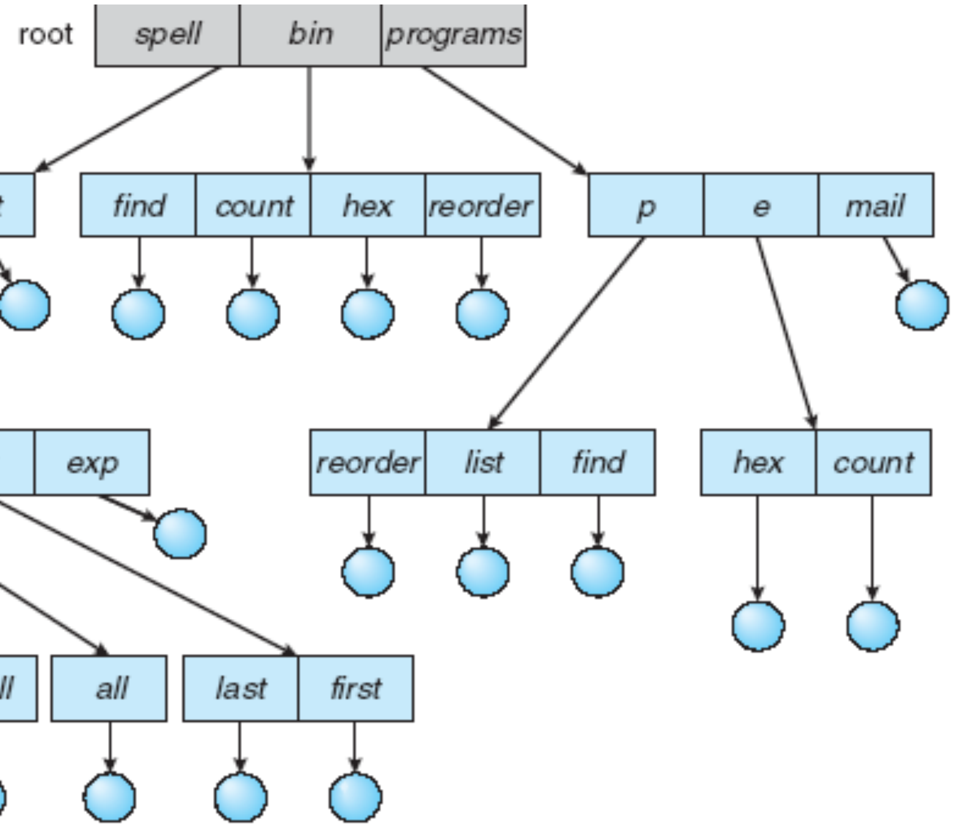
- One directory for all files in the volume
  - Called root directory



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

Tree-structured Directory

- Directory is now a tree of folders
  - Each folder contains files and sub-folders



Terminology Warning

- Term “folder” as we are using it is often referred to as a “directory”

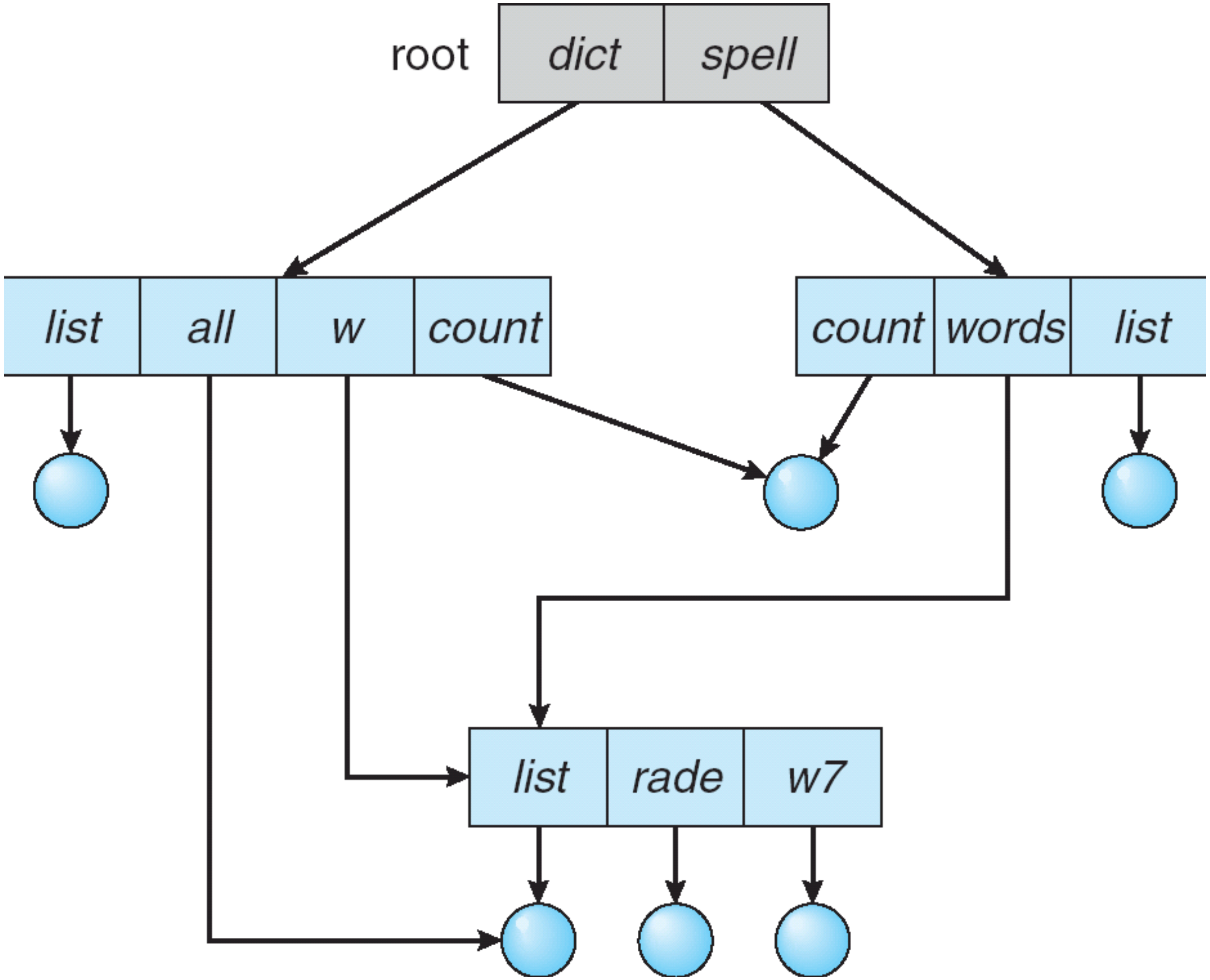
And vice versa!

Path Names

- To access a file, the user should either:
  - Go to the folder 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
    - e.g., /home/pat/projects/test.c
  - Relative: path from the current *working directory*
    - projects/test.c (when executing in directory /home/pat)
    - current working directory stored in PCB of a process
- Unix has two special entries in each directory:
  - "." for current directory and ".." for parent

Acyclic Graph Directories

- Share subdirectories or files



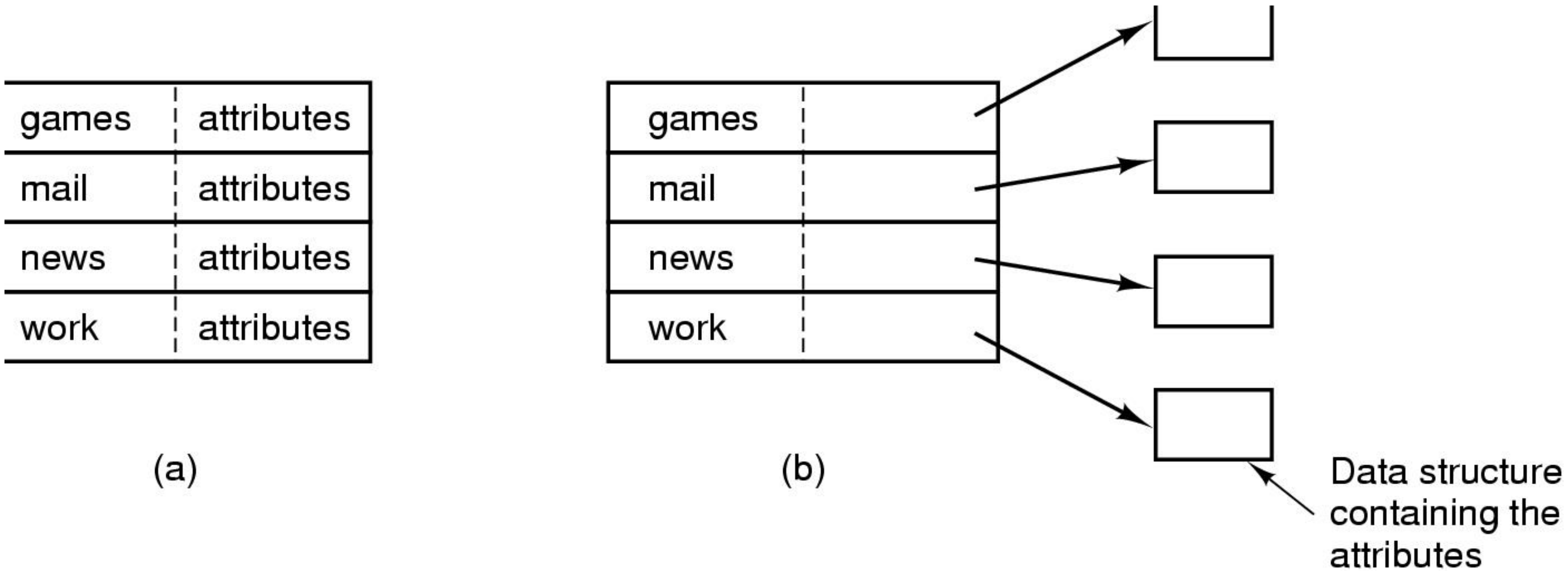
Acyclic Graph Directories

How to implement shared files and subdirectories:

- Why not copy the file?
- Multiple directory entries may “link” to the same file
  - *ln* in UNIX, *fsutil* in Windows for hard links
    - File has to maintain a “reference count” to prevent dangling links
  - “soft link:” special file w/ the name of another file in it
    - *ln –s* in UNIX, shortcuts in Windows
    - dangling soft links hard to prevent

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



File System Mounting

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

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

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
  - Network File System (NFS)
  - Server Message Block (SMB)

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 location where next write occurs, updating as writes 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 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

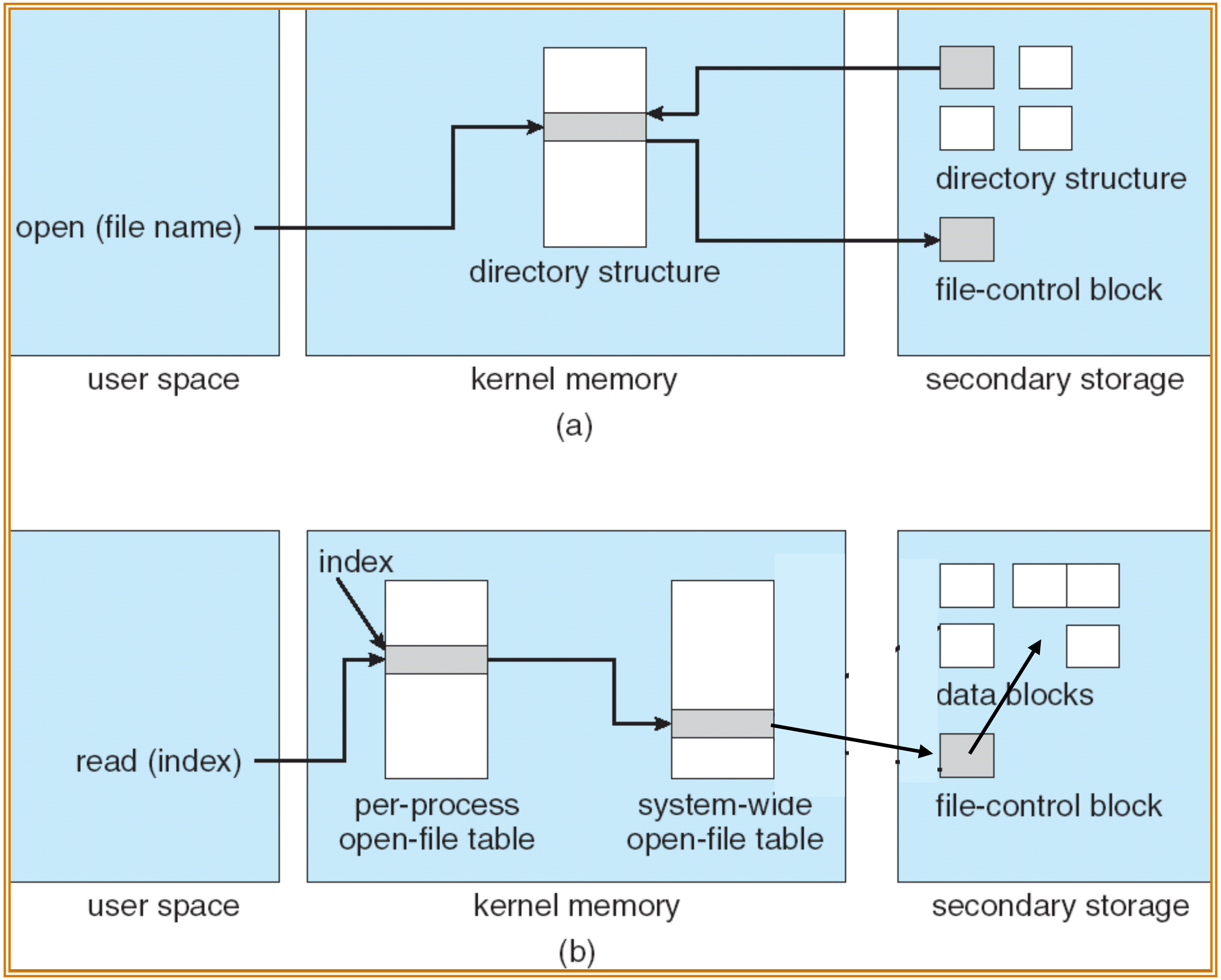
- OS typically keeps two levels of internal tables:
- Per-process table
  - Information about the use of the file by the user (e.g. current file position pointer)
- System wide table
  - Gets created by first process which opens the file
  - Location of file on disk
  - Access dates
  - File size
  -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

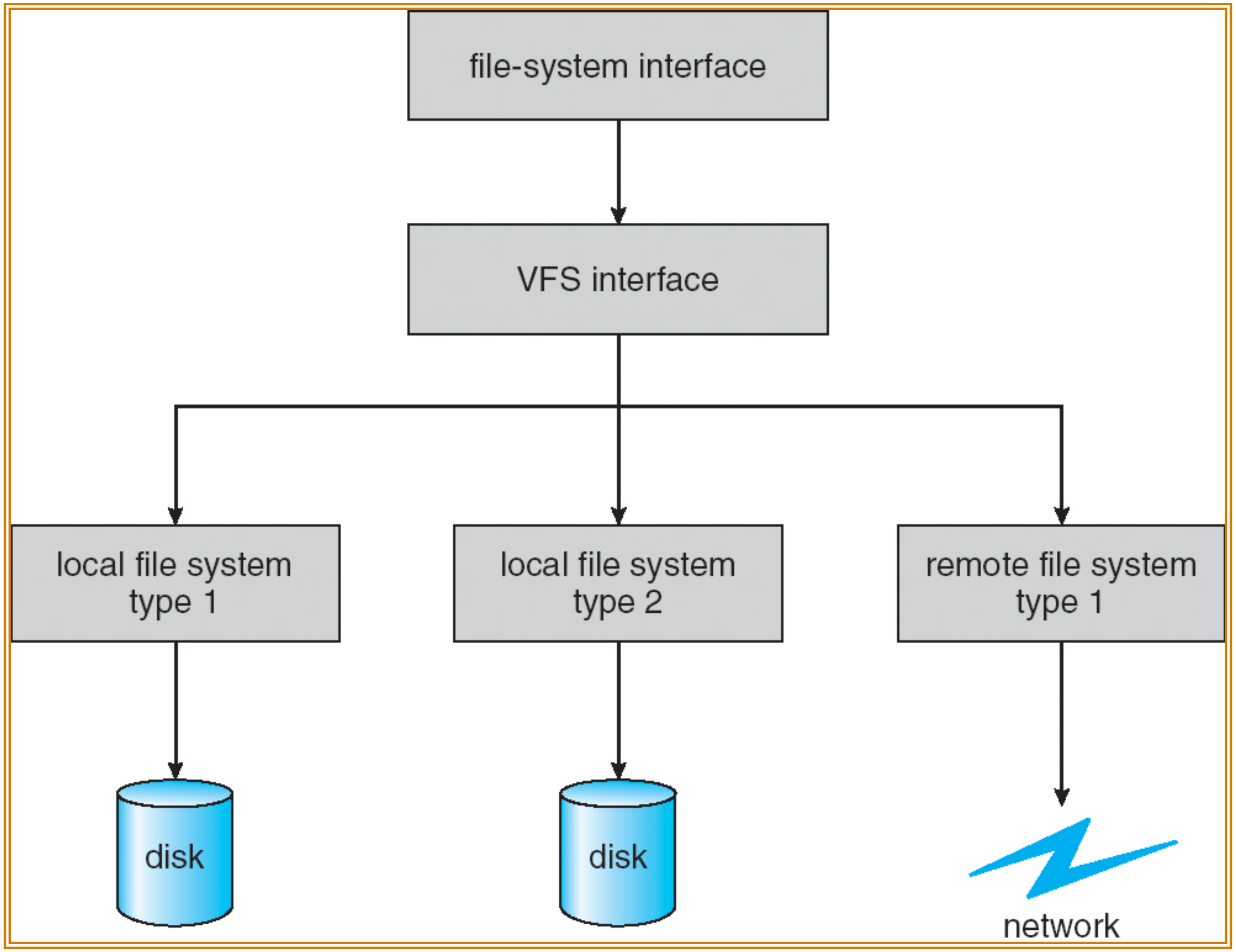
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Files Open and Read



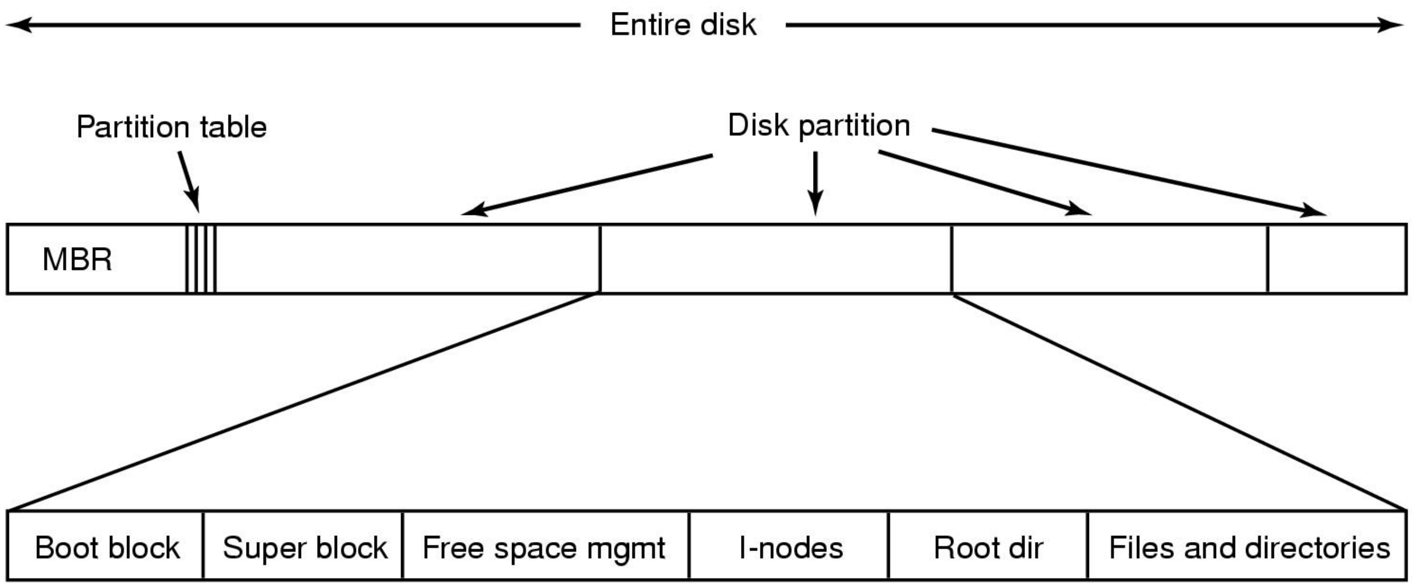
Virtual File Systems

- 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



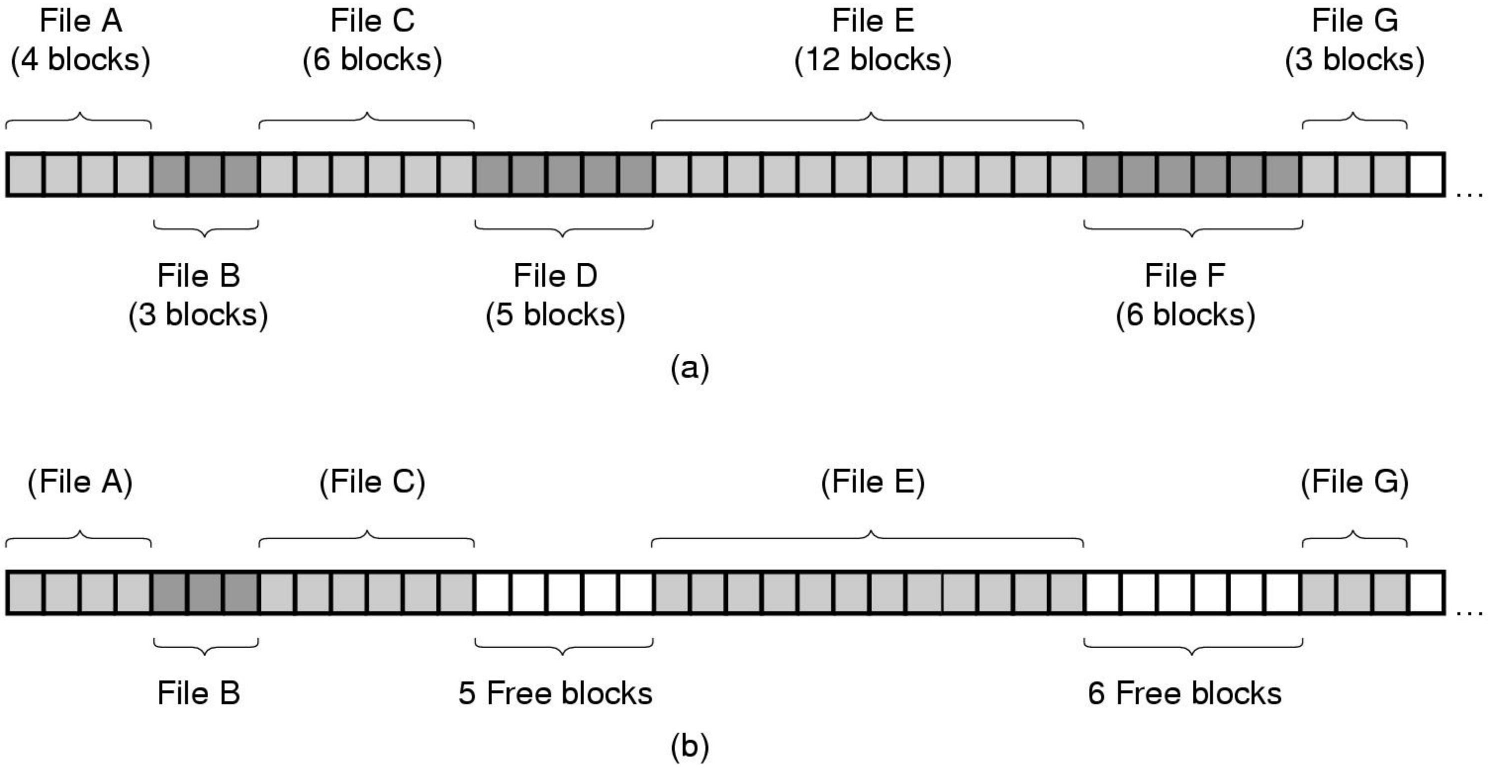
Storing Files

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

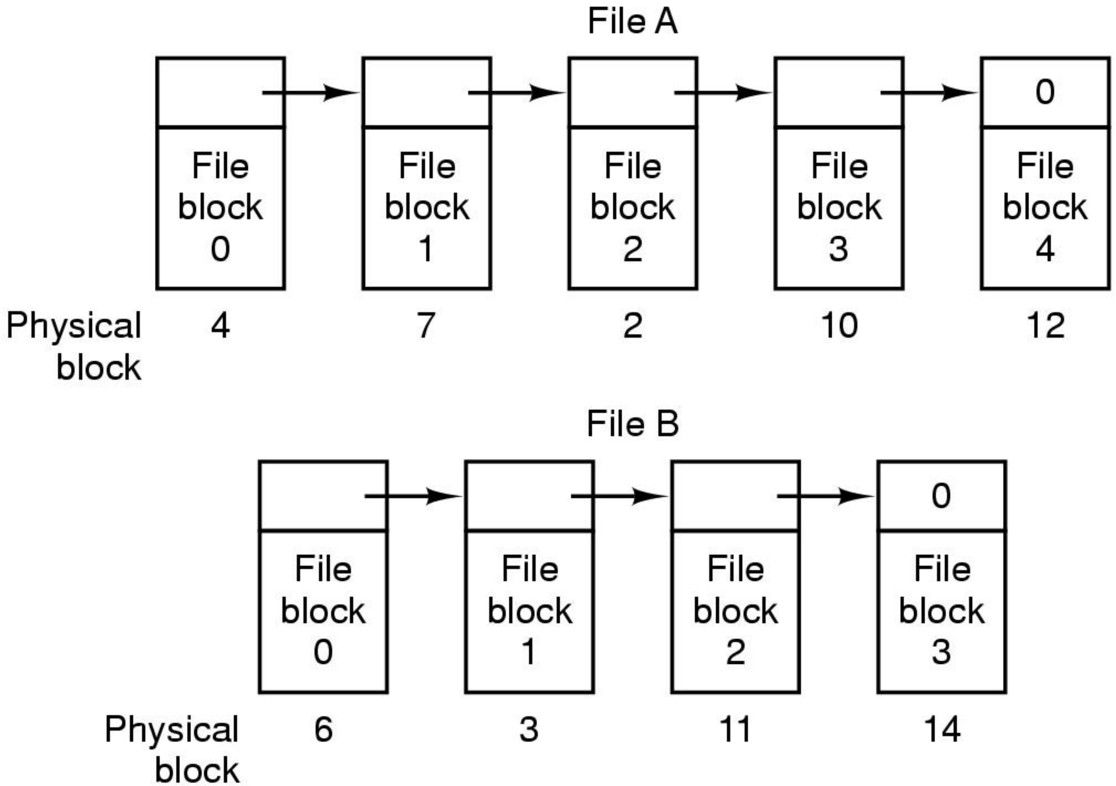


Contiguous Allocation

- 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
  - Rest of disk block is file data



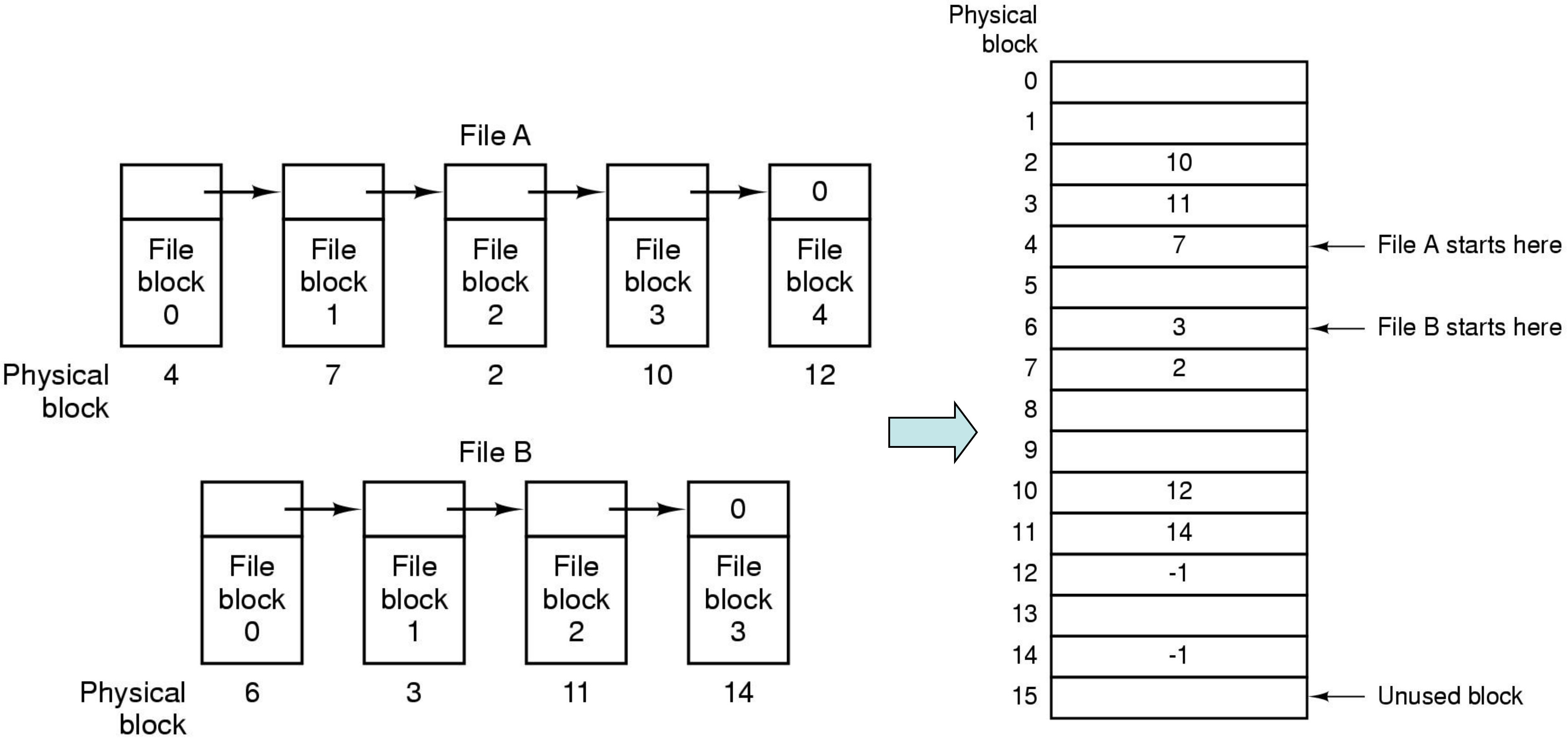
Linked List Allocation

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

FAT 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 Usage

- Initially the file system for MS-DOS
- Still used in CD-ROMs, Flash Drives

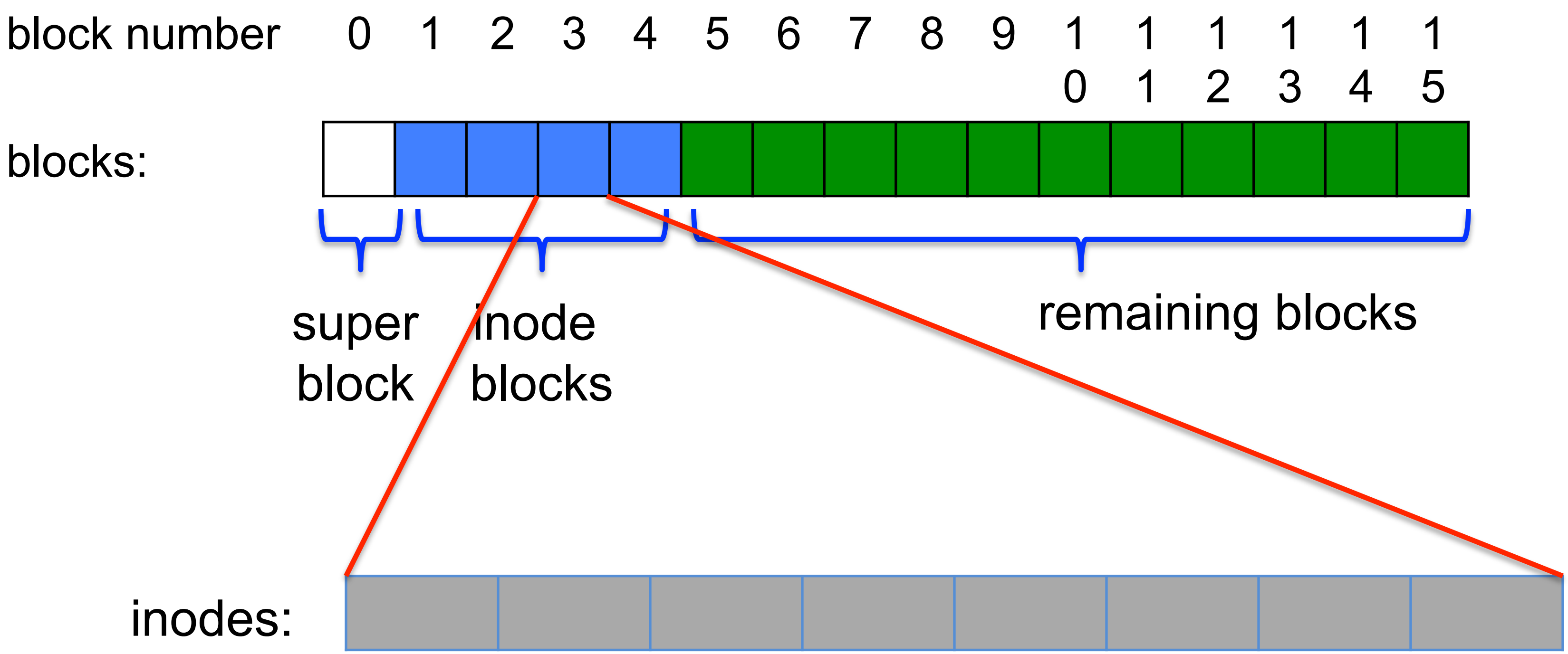
FAT Discussion

- 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 1TB (2^{40} bytes) disk, 4KB (2^{12}) block size, FAT has 256 million (2^{28}) entries. If 4 bytes used per entry $\Rightarrow$ 1GB (2^{30}) of main memory required for FS, which is a sizeable overhead

FAT Folder Structure

- A folder is a file filled with 32-byte entries
- Each entry contains:
  - 8 byte name + 3 byte extension (ASCII)
  - creation date and time
  - last modification date and time
  - first block in the file (index into FAT)
  - size of the file
- Long and Unicode file names take up multiple entries.

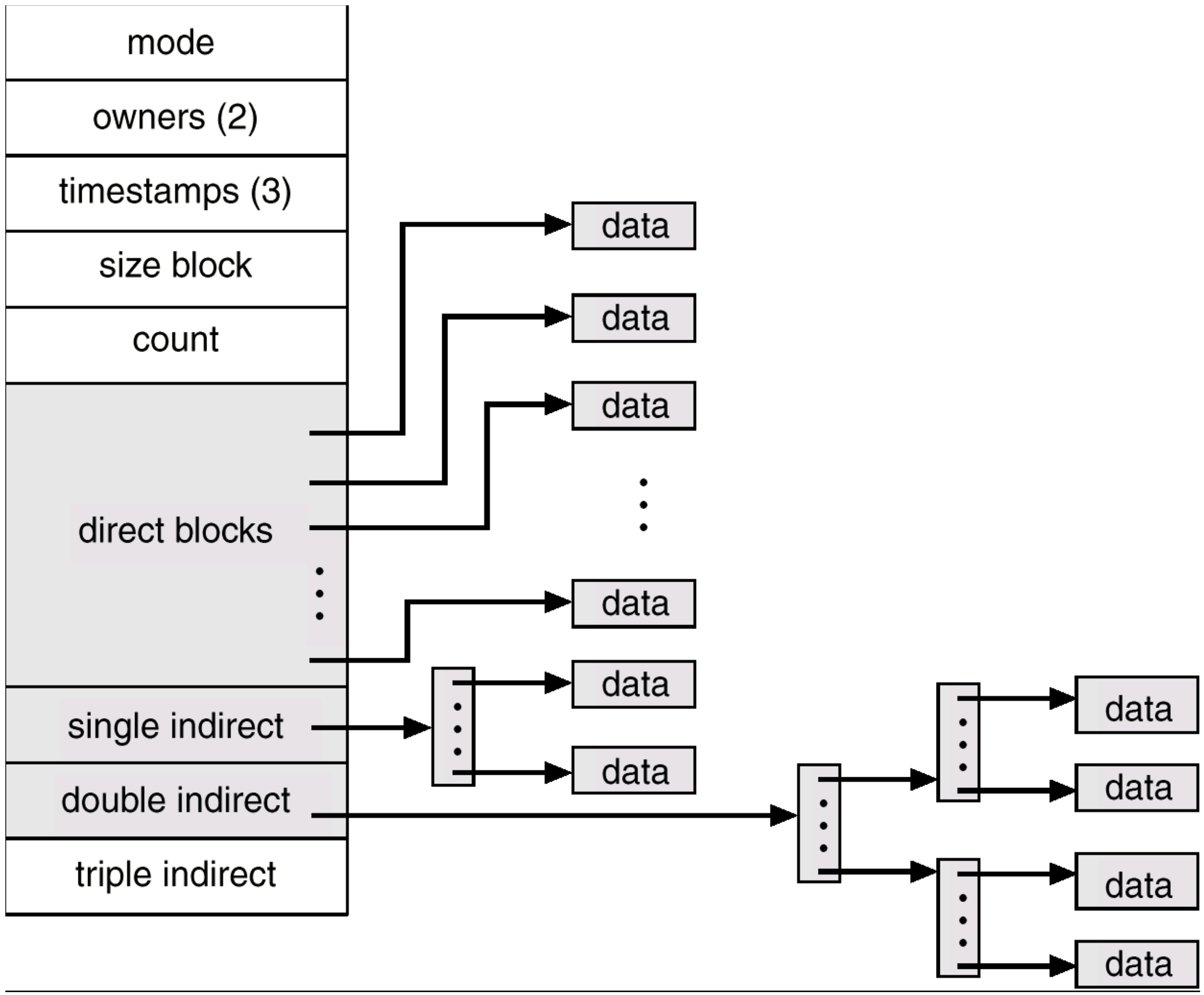
UFS - Unix File System: Layout



UFS Superblock

- Contains info about volume such as
  - #blocks with inodes
  - first block on the free list

UFS Inode Structure



Unix inodes

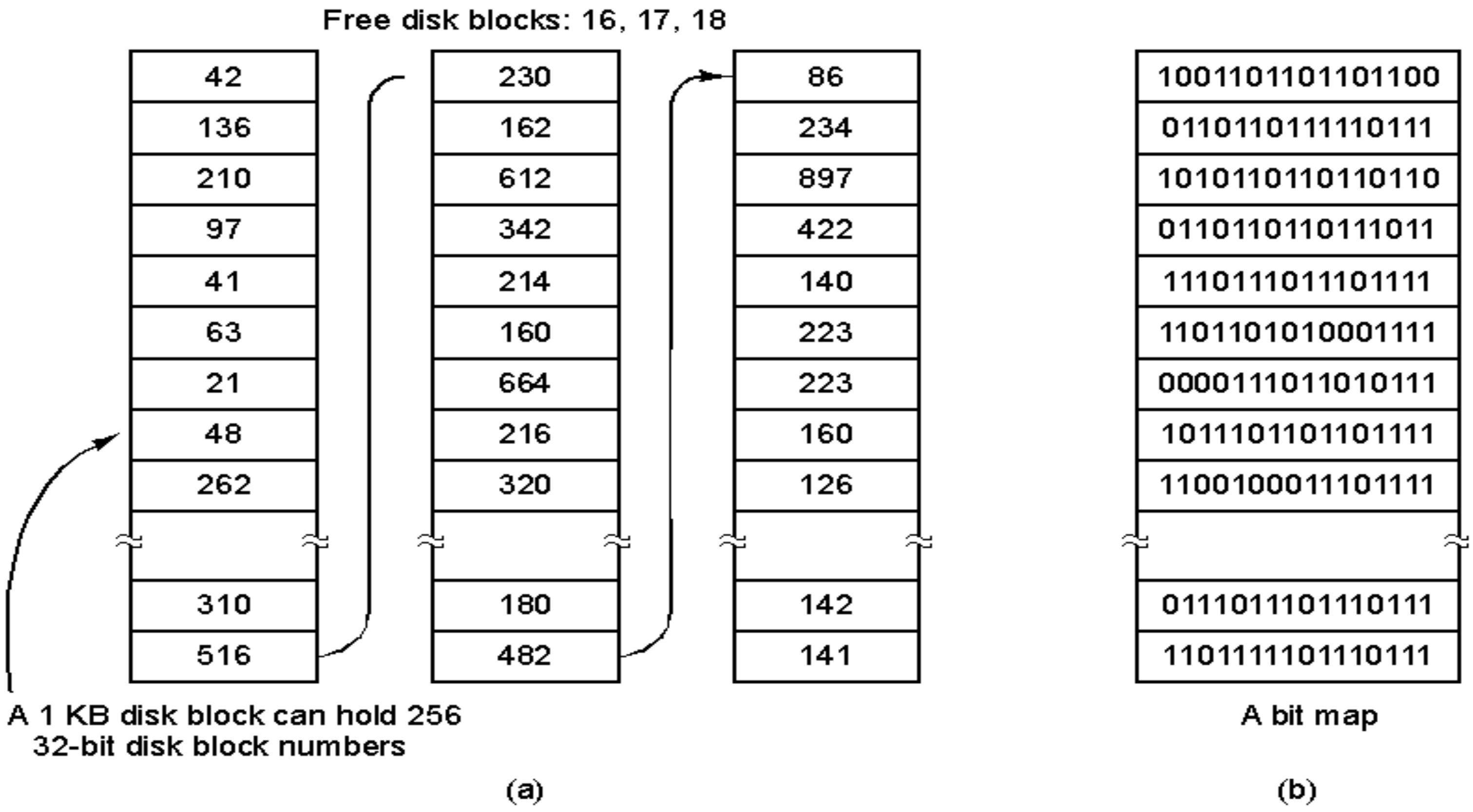
- If blocks are 4K and block references are 4 bytes…
  - First 48K reachable from the inode
  - Next 4MB available from single-indirect
  - Next 4GB available from double-indirect
  - Next 4TB available through the triple-indirect block
- Any block can be found with at most 4 disk accesses
  - not counting the superblock and inode…
  - not counting the directory access either...

Other info in i-node

- Type
  - ordinary file, directory, symbolic link, special device, …
- Size of the file (in #bytes)
- #links to the i-node
- Owner (user id and group id)
- Protection bits
- Times
  - creation, last accessed, last modified

Managing Free Disk Space

- 2 approaches to keep track of free disk blocks



UFS directory structure

- Array of (originally) 16 byte entries
  - 14 byte file name
  - 2 byte i-node number
- In modern implementations, directories are usually linked lists. An entry contains:
  - 4-byte inode number
  - Length of name
  - Name (UTF8 or some other Unicode encoding)
- First entry is “.”, points to self
- Second entry is “..”, points to parent inode

File System Consistency

- System crash before modified files written back
  - Leads to inconsistency in FS
  - fsck (UNIX) & scandisk (Windows) check FS consistency
- Algorithm:
  - Build table with info about each block
    - initially each block is unknown except superblock
  - Scan through the inodes and the freelist
    - Keep track in the table
    - If block already in table, note error
  - Finally, see if all blocks have been visited

A changing problem

- Consistency used to be very hard
  - One problem was that driver implemented C-SCAN and this could reorder operations
  - But cache can also re-order operations for efficiency
  - 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

(c)

	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

(d)

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**
  - Transaction is considered **committed** once it is written to the log
  - However, the file system may not yet be updated

Approach 1: "Write-Ahead Log" (WAL) or "Journaling File System"

- Inspired by database systems
- 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..

Approach 2: “moving blocks”

- When a block is updated, it is added to the log, rather than updated in place.
- The old block is now free to be re-used.
- Note, superblock and inodes also move, so it’s a little trickier to keep track of where they are.
- Periodically, the disk is “cleaned”
  - Essentially defragmentation
- E.g. LFS. While interesting, the approach is not in much use today.

LFS: why?

- Operations on multiple blocks can be made “atomic”
  - Much simplifies consistency management
- Avoids disk arm movements for improved performance
  - Less of an issue today
- Reduces wear on SSD/Flash drives
  - *Automatic wear leveling*