File Systems Internals

CS 3113

12248

File-System Structure

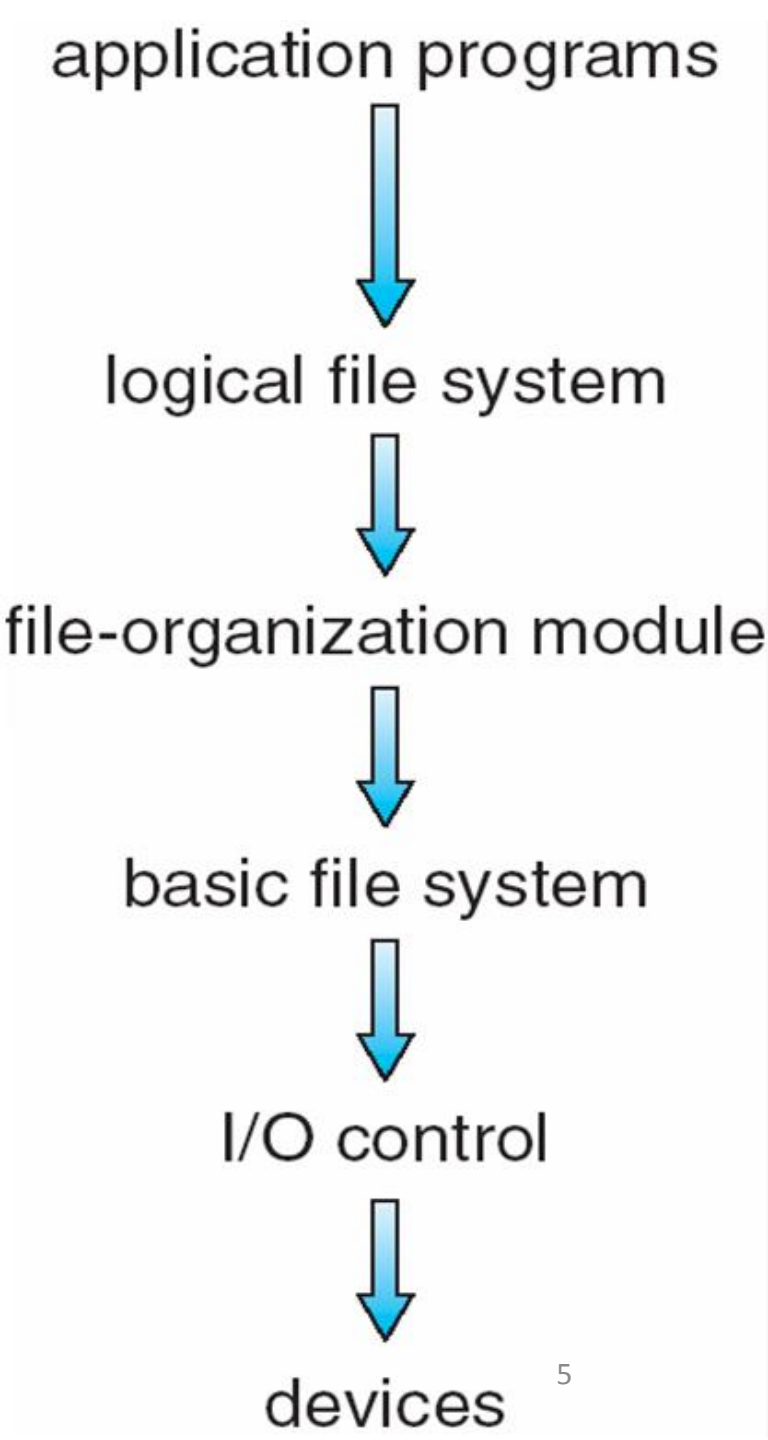
- File structure
  - Logical storage unit
  - Collection of related information
- **File system** resides on secondary storage (e.g., disks)
  - Provided user interface to storage, mapping logical to physical
  - Provides efficient and convenient access to disk by allowing data to be stored, located, and retrieved easily

File-System Structure

- Disk provides in-place rewrite and random access
  - I/O transfers performed in **blocks** of **sectors** (usually 512-4K bytes)
- **File control block** – storage structure consisting of information about a file (or directory)
- Storage for the content of a file (or directory)
- **Device driver** controls the physical device

Layered File System

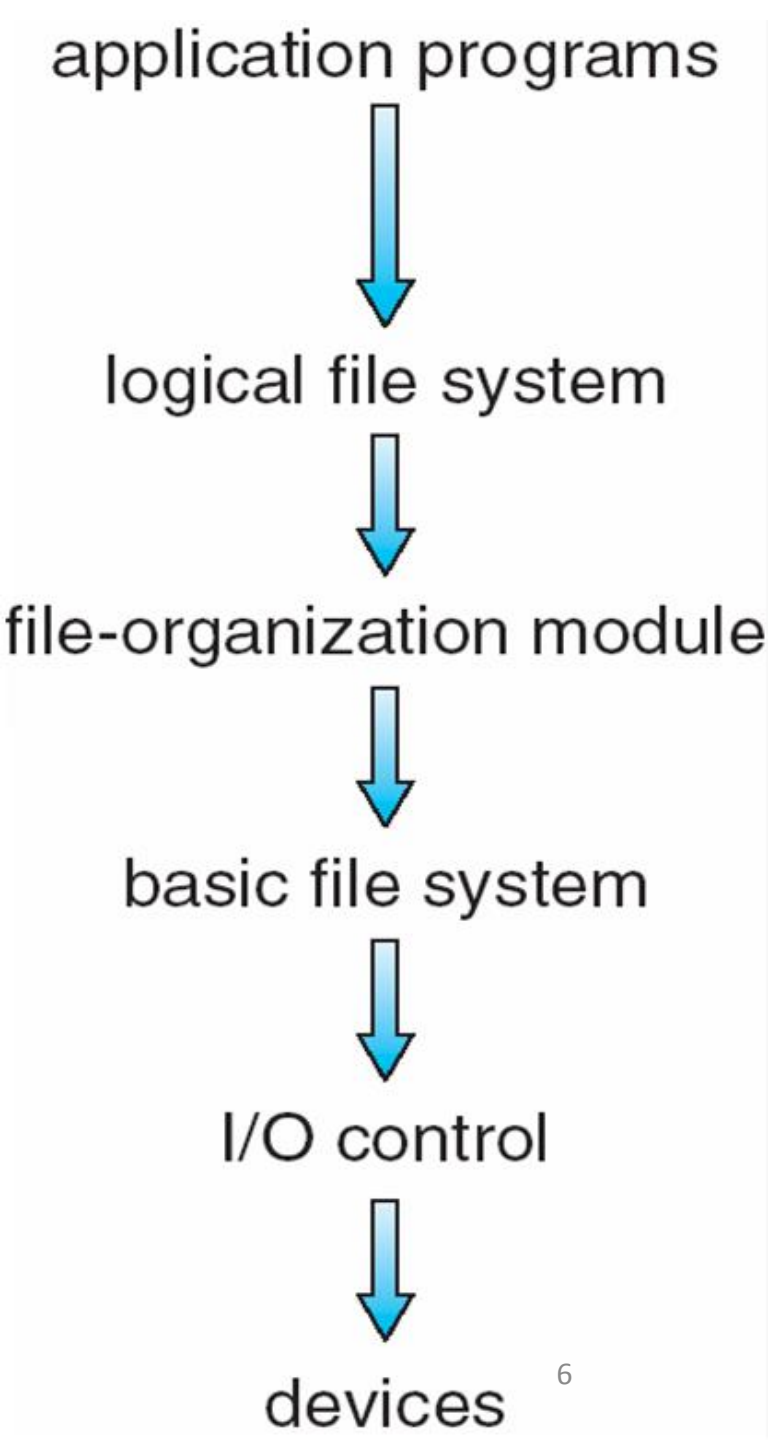
Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance



Device Drivers

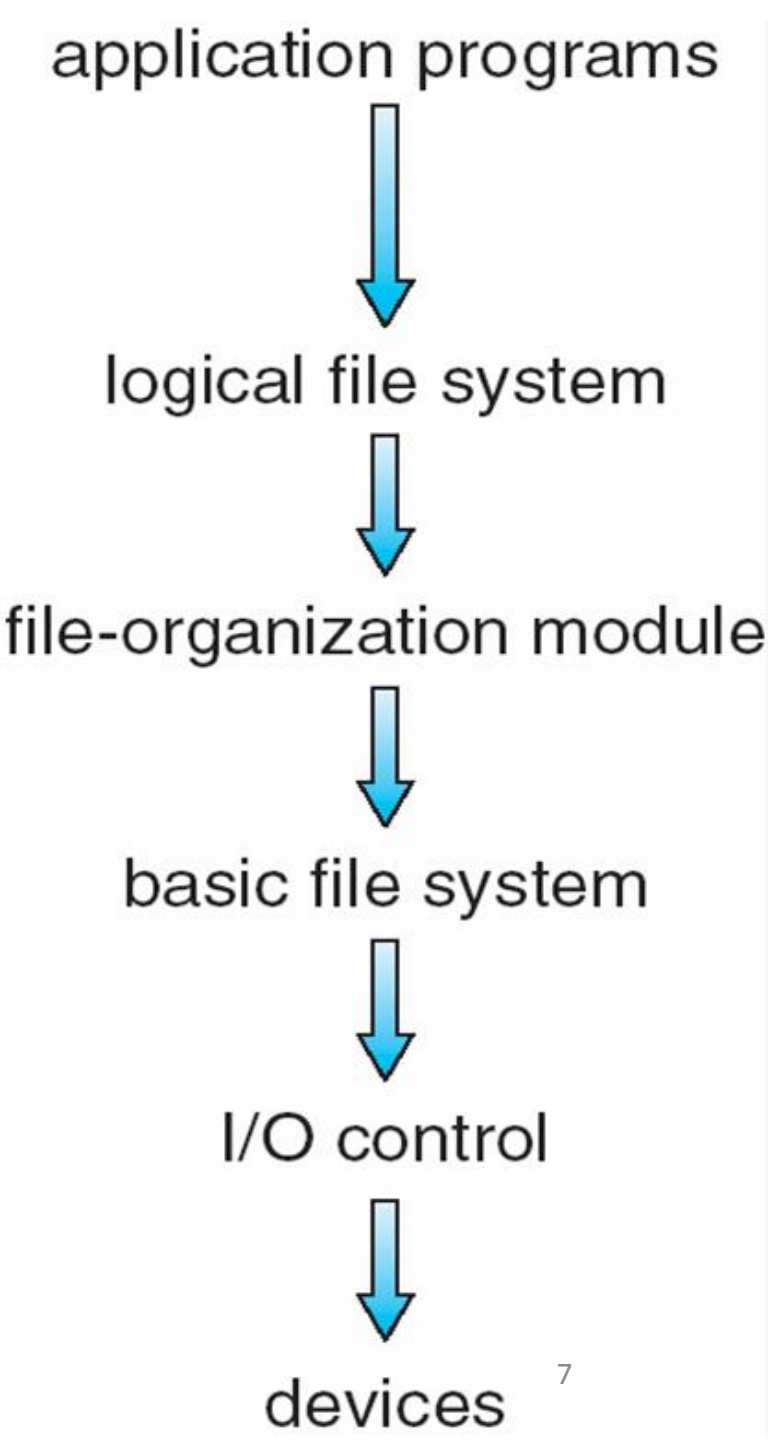
Manage I/O devices at the I/O control layer

- Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level hardware specific commands to hardware controller



Basic File System

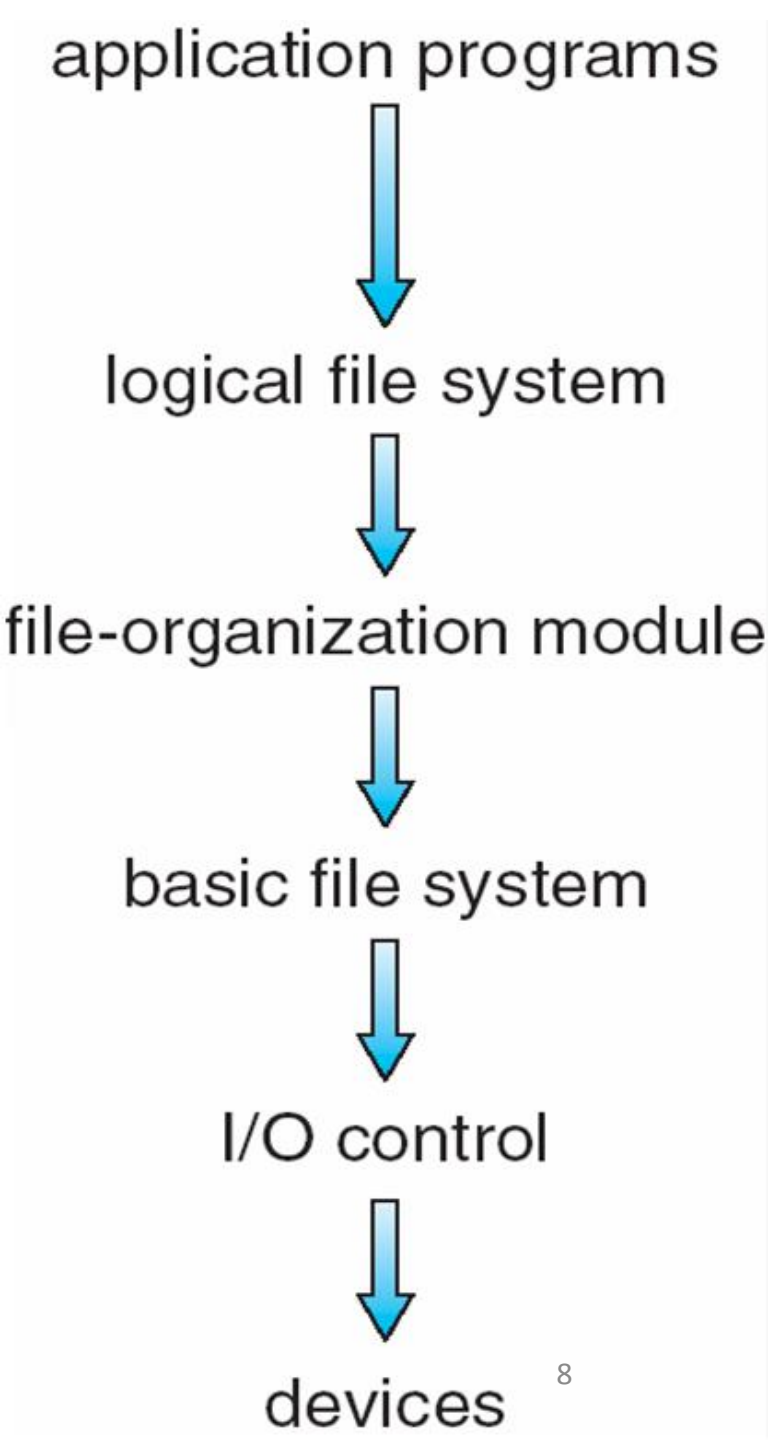
- Given command like “retrieve block 123” translates to device driver
- Also manages memory buffers and caches (allocation, freeing, replacement)
  - Buffers hold data in transit
  - Caches hold frequently used data



File Organization Module

Understands files, logical address, and physical blocks

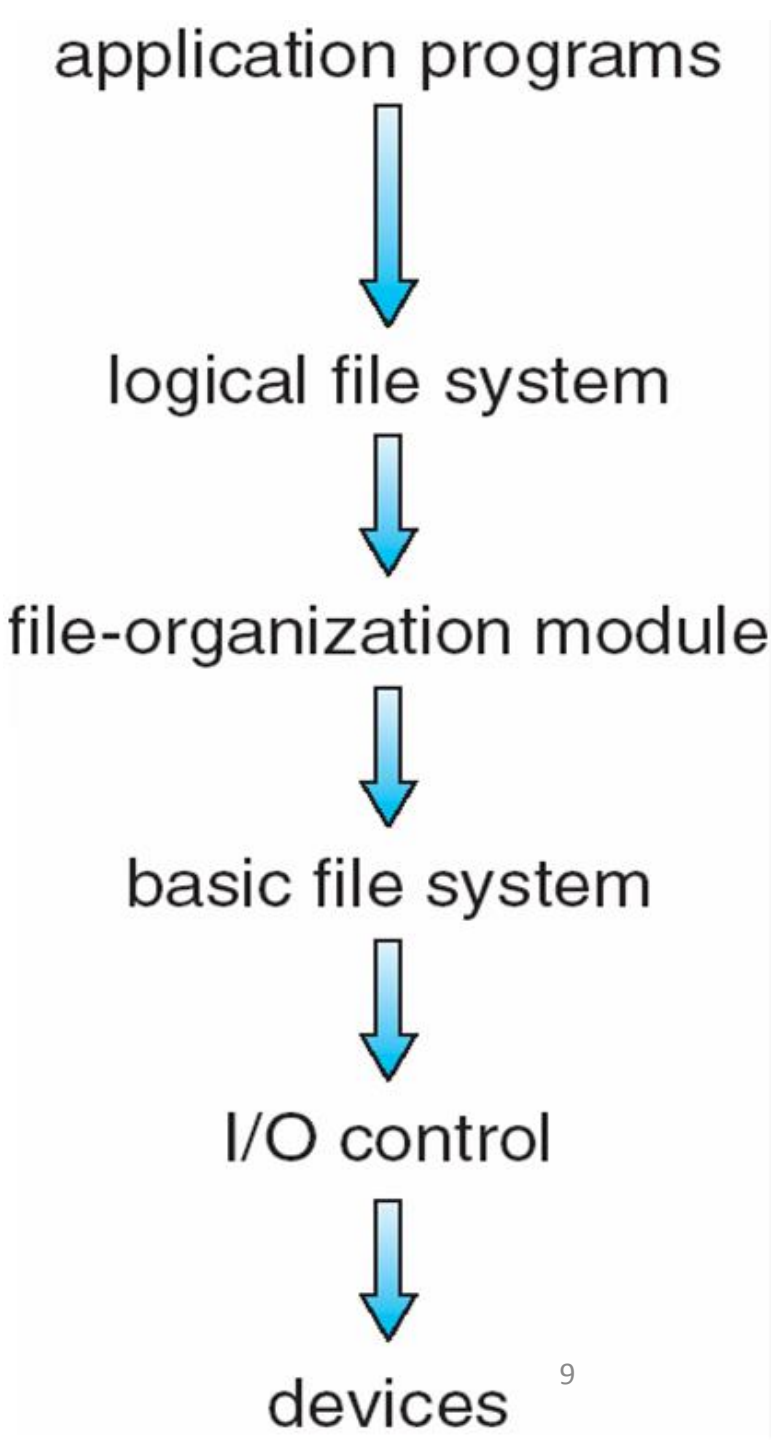
- Translates logical block # to physical block #
- Manages free space, disk allocation



Logical File System

Manages metadata information

- Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
- Directory management
- Protection



Many File System Types

Many file systems, sometimes many within an operating system

- Each with its own format
  - CD-ROM is ISO 9660
  - Unix has **UFS**, FFS;
  - Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray,
  - Linux has more than 40 types, with **extended file system** (common to see ext3 and ext4)
  - Distributed file systems: Network File System (NFS), Andrew
- New ones still arriving: ZFS, GoogleFS, Oracle ASM, FUSE

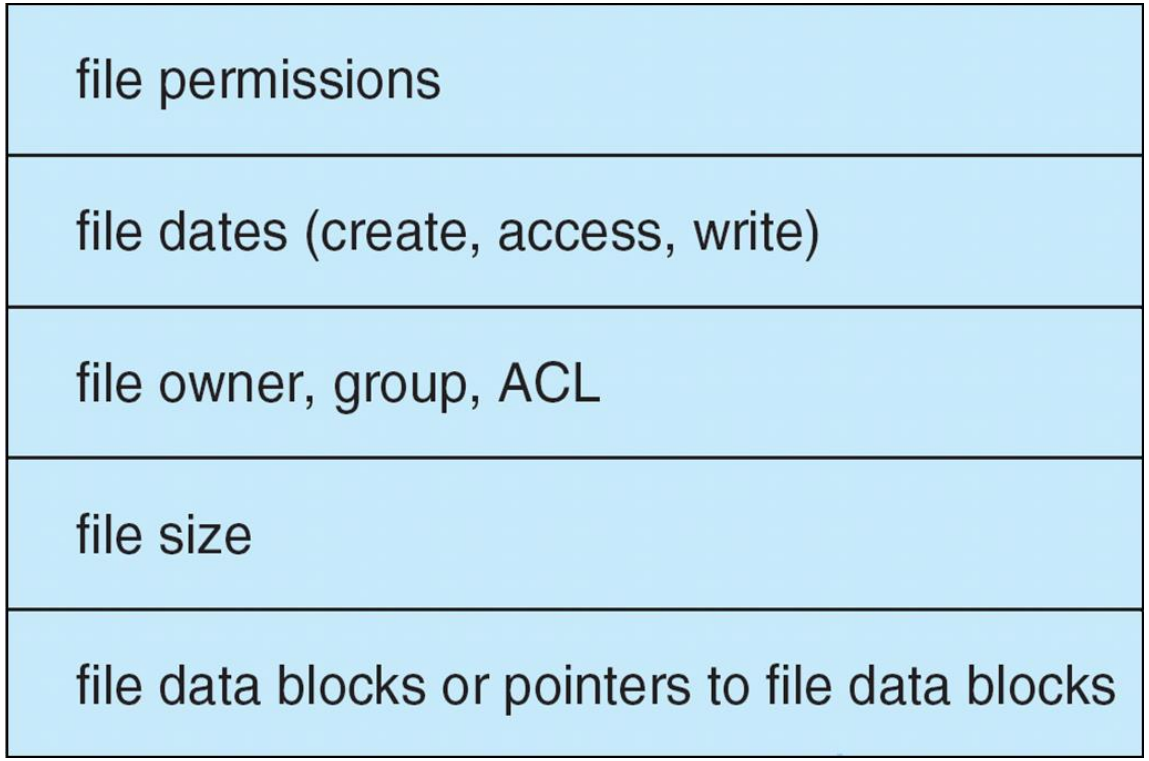
File System Implementation

We are familiar with the system calls at the API level, but how do we implement their functions?

- **Boot control block** contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- **Volume control block (superblock, master file table)** contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure organizes the files
  - Names and inode numbers, master file table

Representing Individual Entities (Files/Directories)

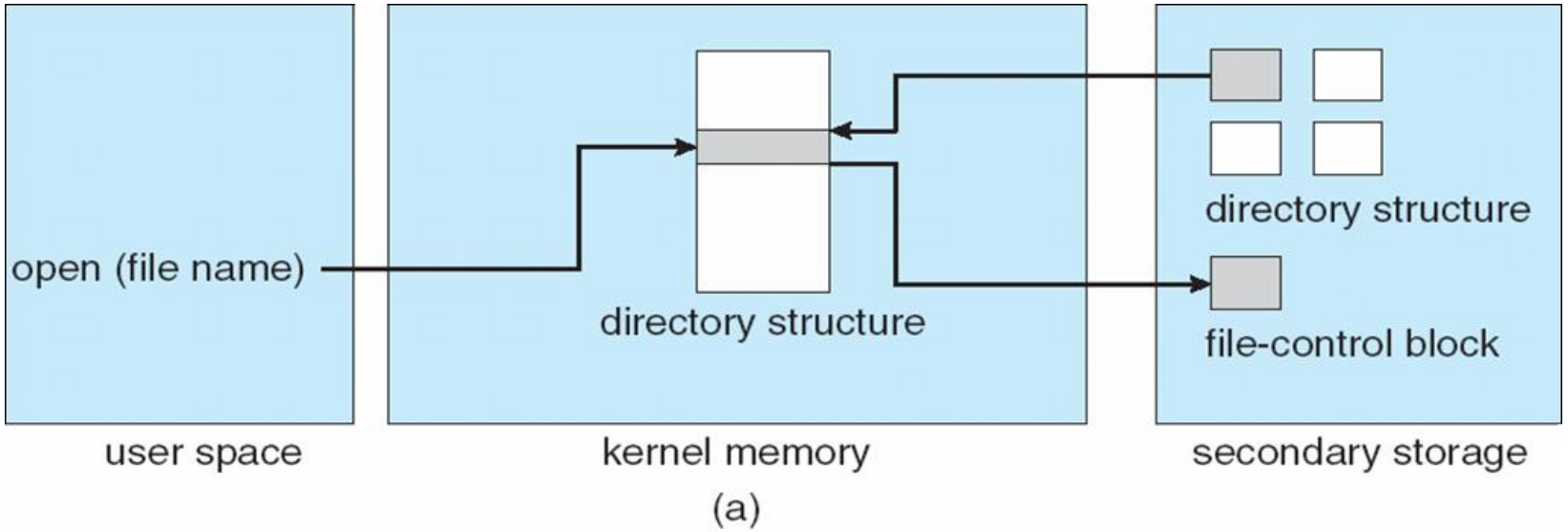
File Control Block (FCB) contains meta-level details about the file. Also called an **INODE**



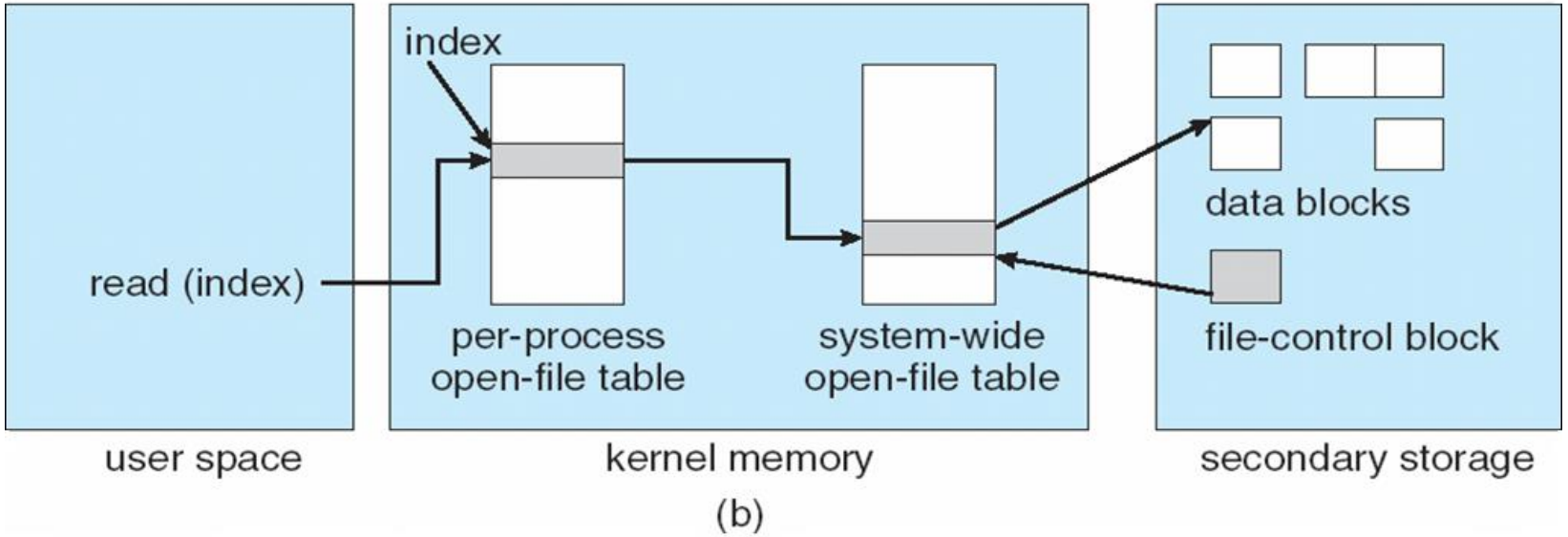
Representation of Files/Directories

- Secondary storage:
  - Persistent
  - All data needed to represent the files/directories
- In-memory storage:
  - Ephemeral
  - Represents the data needed to manipulate the file (e.g., the offset)
  - Can also cache a subset of the file/directory data
    - Fast access
    - Modifications to the data are done here first

Representation of Files/Directories



(a)



(b)

Implementing Directories

- **Linear list** of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree
- **Hash Table** – linear list with hash data structure
  - Decreases directory search time
  - **Collisions** – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method

Allocating Space on the Disk

- An allocation method refers to how disk blocks are allocated for files:
- **Contiguous allocation** – each file occupies set of contiguous blocks
  - Best performance in most cases
  - Simple – only starting location (block #) and length (number of blocks) are required

Contiguous Allocation

Suppose we want to access byte 2021 of file “mail”. How do we find it on the disk?

- Assume block size is 512

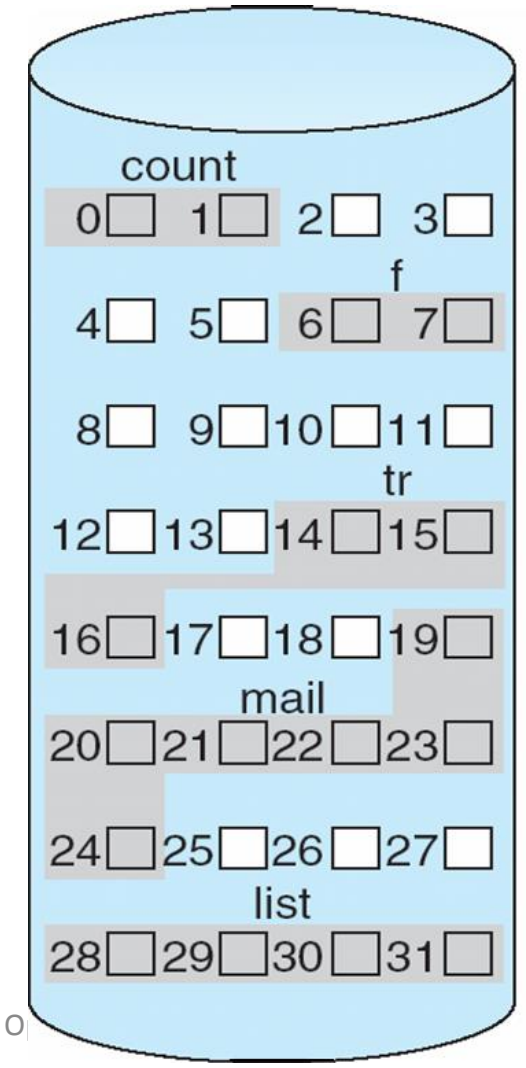
Load block: start + byte / BSIZE

19 + 2021 / 512 = 22

- Which byte?

byte % BSIZE

2021 % 512 = 485



directory

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

Contiguous Allocation

What are the problems with this approach?

Contiguous Allocation

Problems include:

- Finding space for a file
- Knowing file size ahead of time
- External fragmentation
  - Spaces between files that are not used because only small files will fit inside of them
  - Solution: **compaction off-line** (**downtime**) or **on-line**

Extent-Based Systems

- Many newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme
- Allocate disk blocks in ***extents***: a fixed-size contiguous set of blocks
  - A file consists of one or more extents
  - We must maintain a table of these extents and their starting locations

- Exam
- Fs impl
- Project 3

Extent-Based Systems

Example: want to access byte 4131 of a file

- Assume:
  - Extent size: 4 blocks
  - Block size: 512
- Which extent?

  ex = 4131 / (4*512) = 2
- Which block?

  block = BSTART[ex] + 4131 / 512 – ex * 4 = 44 + 0 = 44
- Which byte?

  byte = 4131 % (512) = 35

Extent #	Block Start
0	32
1	8
2	44

Extent-Based Systems

Advantages

- Contiguous allocation allows for quick access to groups of blocks on the disk
- We don’t have to know file size ahead of time
- No external fragmentation
  - A space between files can be used for any other file

Extent-Based Systems

What are the problems?

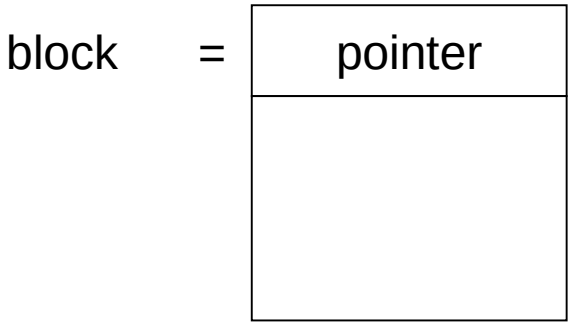
Extent-Based Systems

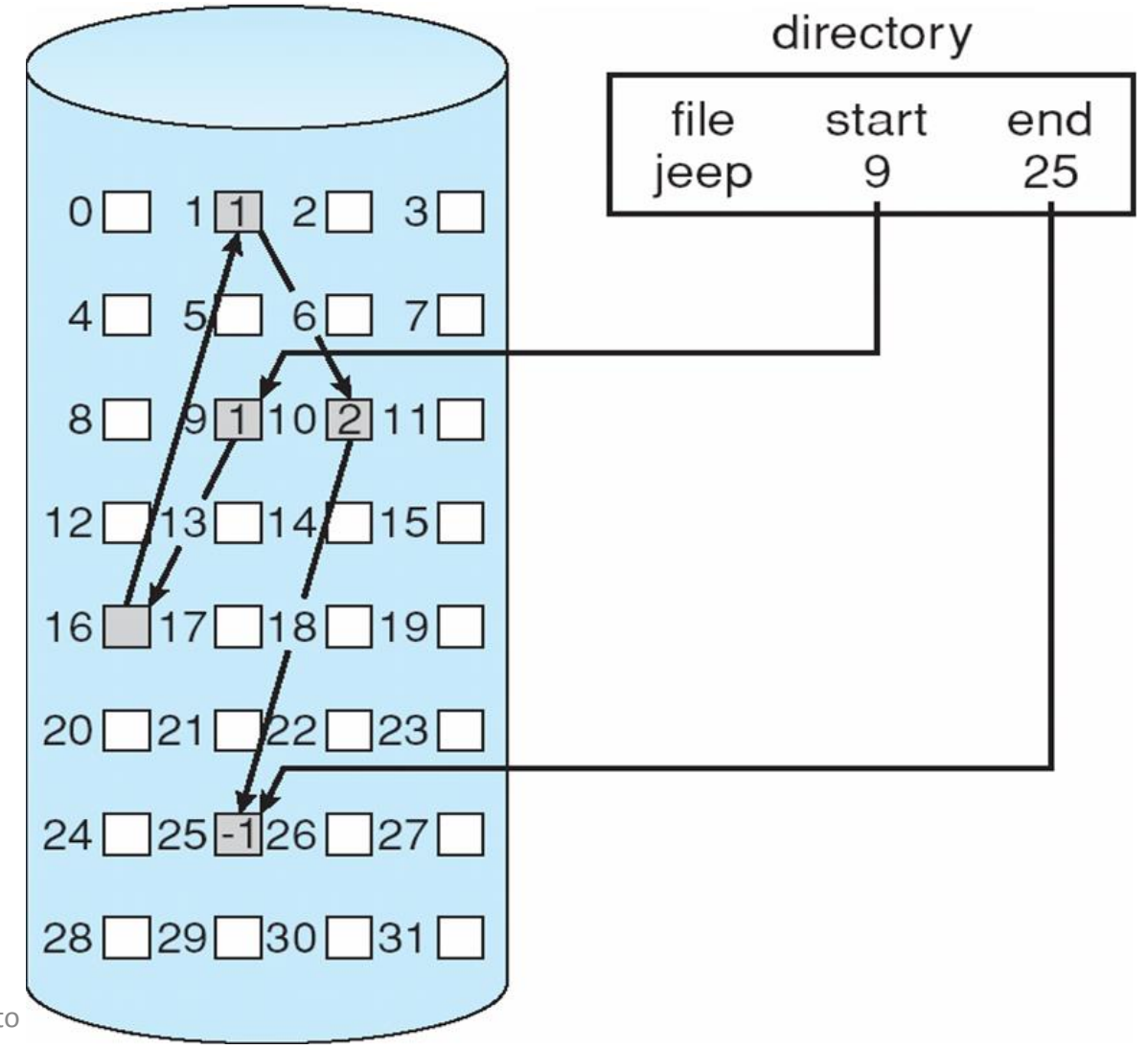
What are the problems?

- A bit harder to compute the block #
- Internal fragmentation: we don’t use some of the blocks in an allocated extent

Linked Allocation

- Linked list of blocks
- Each block contains a "pointer" to the next block
  - This is a block number (not a memory pointer)





Linked Allocation

Accessing byte 2345:

- Which block (logical)?

  2345 / 512 = 4

- How to find this block?
  - Start with the first block
  - Follow the linked list for 4 more hops
- Which byte in the block?

  2345 % 512 = 297

Linked Allocation

Advantages

- No external fragmentation
- Easy to add new bytes onto the end of a file

Linked Allocation

Disadvantages

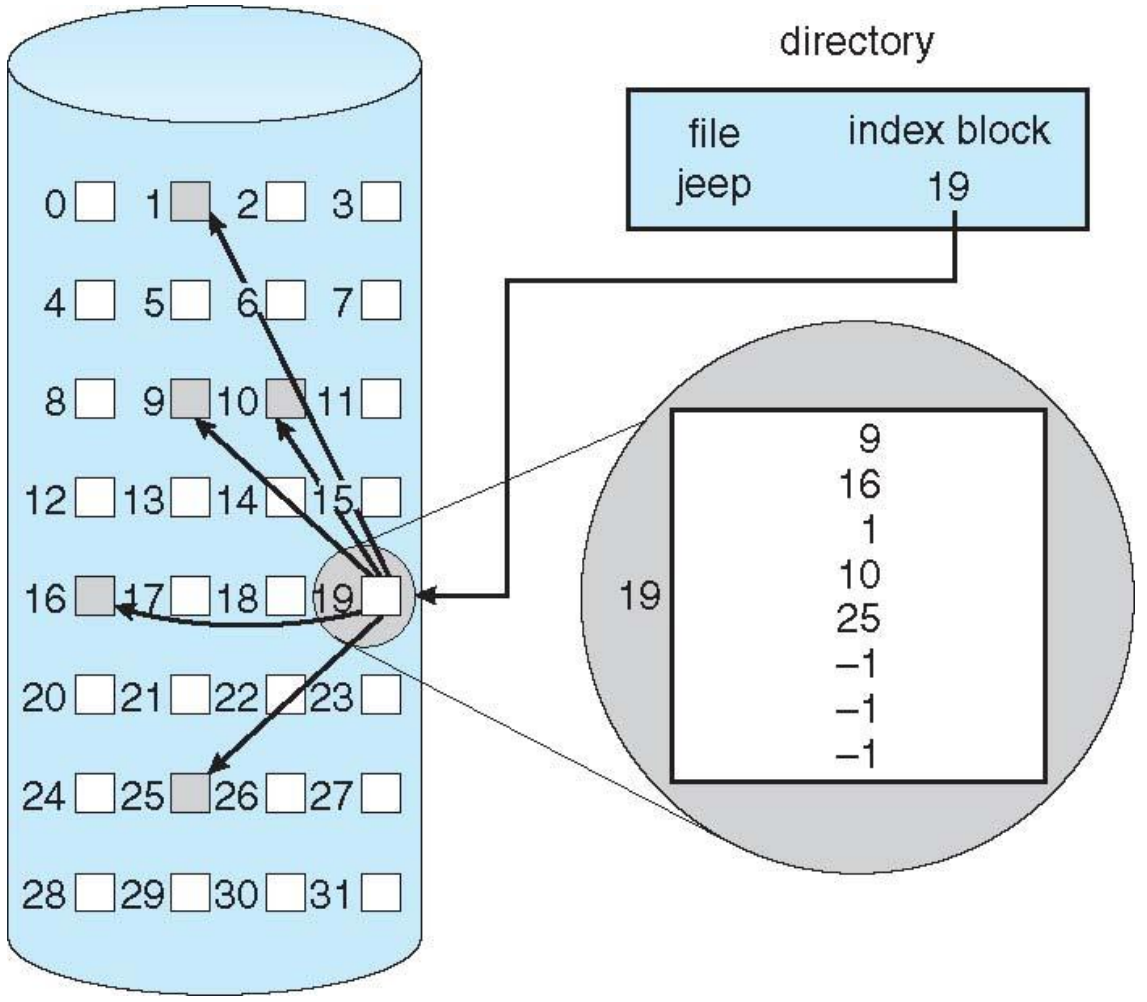
Linked Allocation

Disadvantages

- Reliability can be a challenge
  - Data about the location of the file contents are scattered all over the disk
- Finding the right block to read can be a challenge

Indexed Allocation

A block is dedicated to maintaining an ordered list of the blocks that contains the file contents



Indexed Allocation

Where to find byte 1281?

- Which block (logical)?

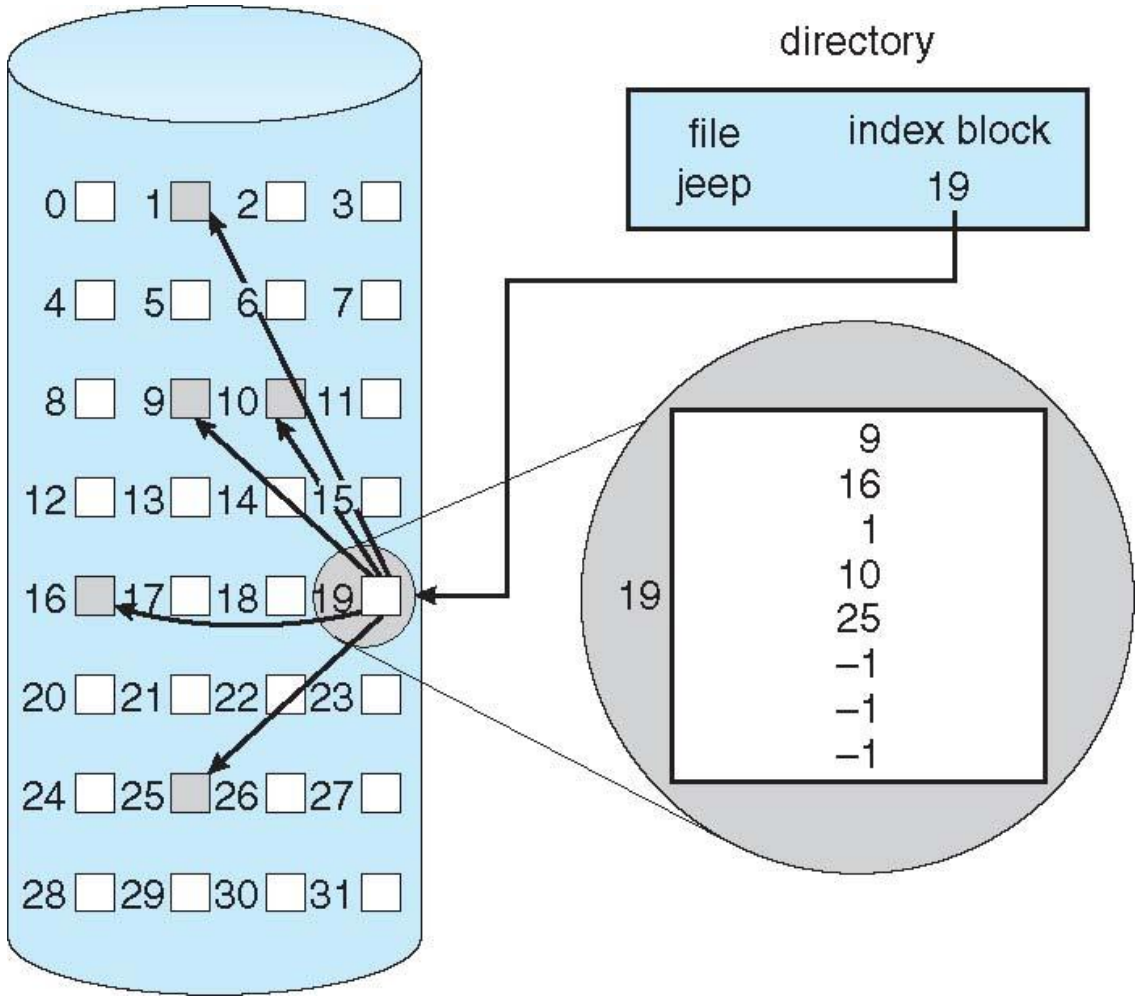
  1281/ 512 = 2

- Which block (actual)?

  table[2] = 1

- Which byte?

  1281 % 512 = 257



Indexed Allocation

Advantages

- No external fragmentation

Indexed Allocation

Disadvantages?

Indexed Allocation

Disadvantages?

- Must allocate another block for the index table
- Extra overhead to translate from logical to physical blocks
- Random access to physical blocks could mean more access time

Indexed Allocation

How can we deal with large files?

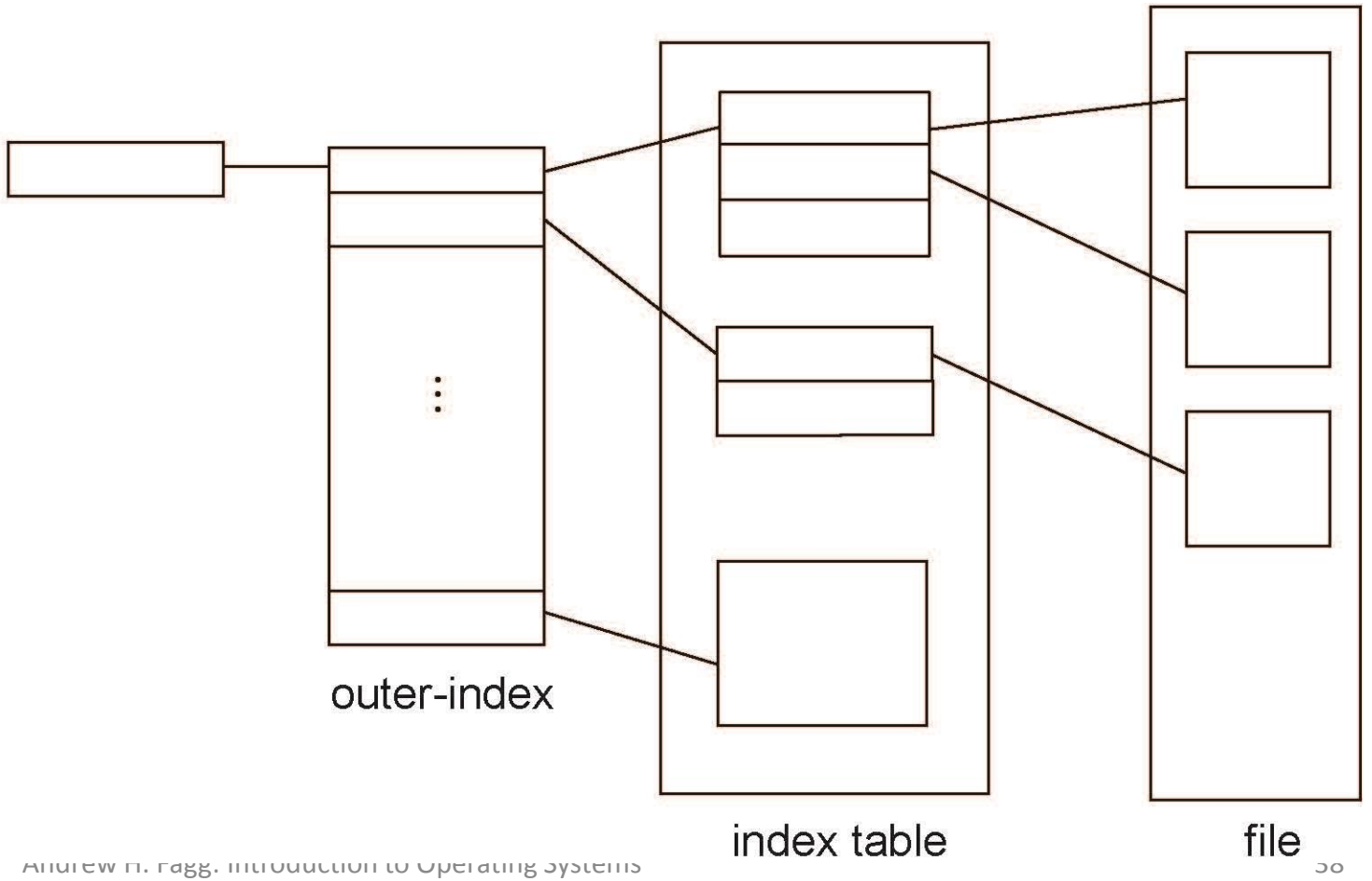
Indexed Allocation

How can we deal with large files?

- Linked list of index tables
- Hierarchical tables

Indexed Allocation

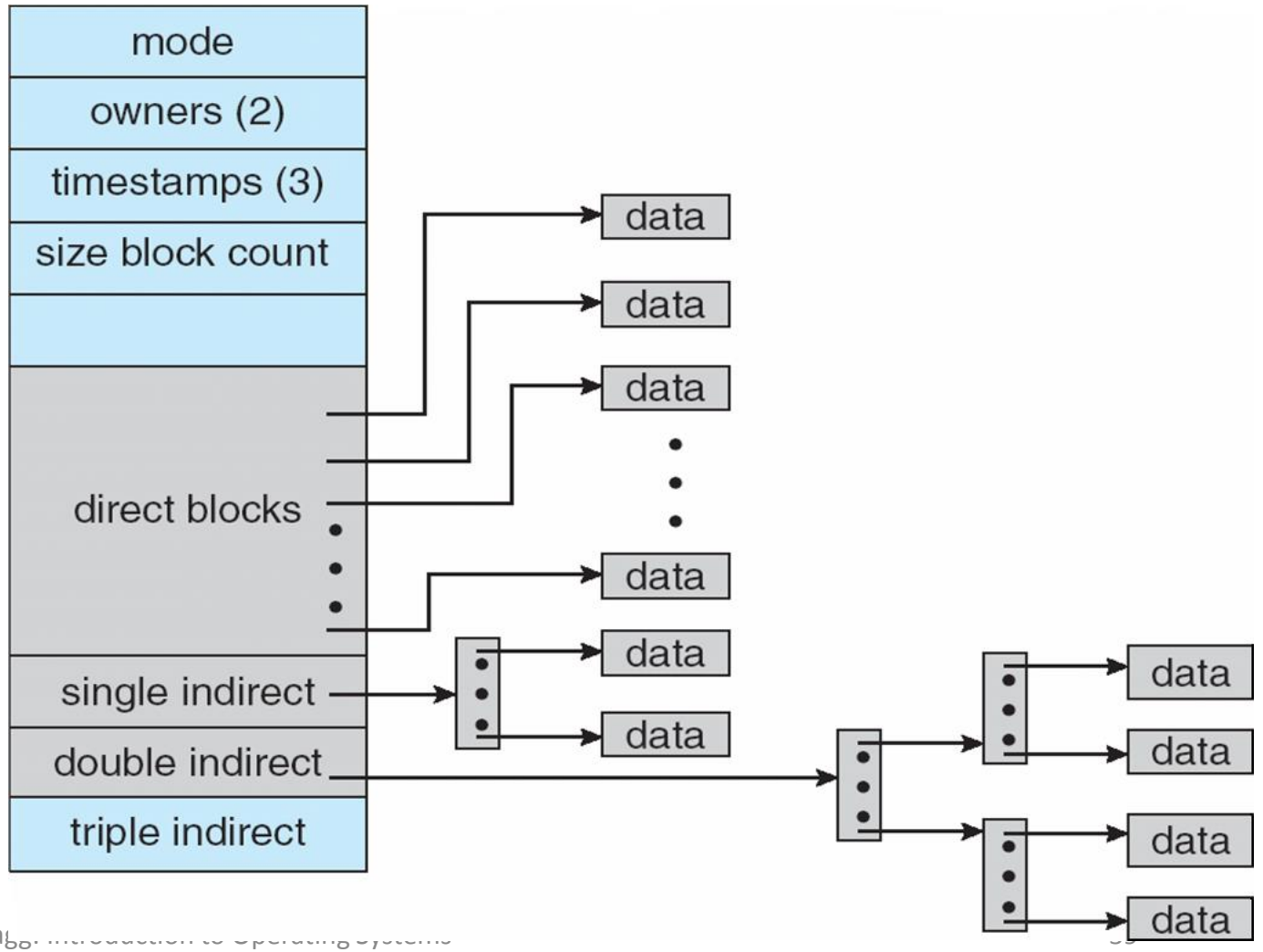
Hierarchical tables



Hybrid Indexed Scheme (Unix)

Unix INODE

- Direct block references: point to data blocks
- Indirect block references: point to blocks that contain references
- Very large files possible
- Efficient allocation when we have small files



Read Performance Considerations

Best method depends on file access type

- Contiguous great for sequential and random
- Linked lists good for sequential, but not random
- Indexed more complex
  - Single block access could require 2 or more index block reads, then the data block read
- Clustering of blocks can help improve throughput and reduce CPU overhead
  - Extent-based approaches

Performance, cont.

Adding data structures that save disk I/O operations is appropriate

- Intel Core i7 Extreme Edition 990x (2011) at 3.46Ghz = 159,000 MIPS
- Typical disk drive: 250 I/O operations per second
  - 159,000 MIPS / 250 = 630 million instructions during one disk I/O operation
- Fast SSD drives provide 60,000 IOPS
  - 159,000 MIPS / 60,000 = 2.65 millions instructions during one disk I/O

Performance, cont.

Saving on disk I/O operations: cache blocks in main memory

- 1st read operation: go to the disk, but cache the block
- Subsequent operations: don't have to wait for the I/O operation (these operations can be reads or writes)
- Must remember to push any updated blocks back to the disk before clearing them from the cache

Free Space Management

File system must also keep track of which blocks are available on the disk for allocation to new directories or files

- Tracking can be done at the block level or at the level of a cluster of blocks (extents of blocks)

Free Space Management: Bit Vectors

One bit allocated per block on the disk

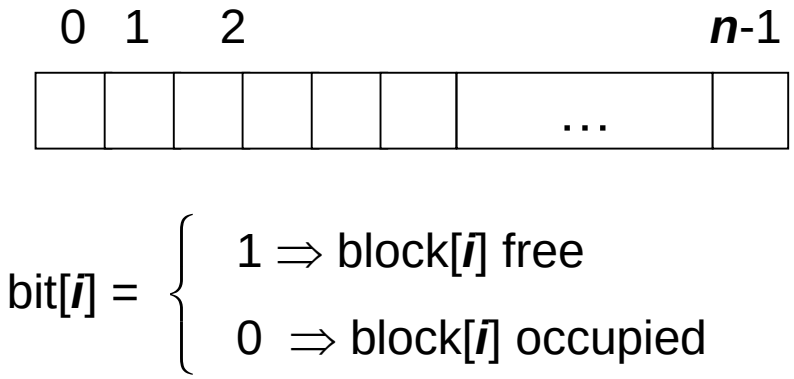
- Requires extra space:

  block size = 4KB = 2^{12} bytes

  disk size = 2^{40} bytes (1 terabyte)

  n = $2^{40}/2^{12} = 2^{28}$ bits (or 32MB)

  if clusters of 4 blocks -> 8MB of memory



Free Space Management: Bit Vectors

One implementation for the last example:

- 32MB table: `unsigned char table[32 x 2^20]`
- Block 0: table[0], bit 0
- Block 1: table[0], bit 1
- Block 2: table[0], bit 2
- Block k: table[k >> 3], bit k%8

Free Space Management: Bit Vectors

- Bit interpretation: 0 -> used; 1 -> free
- Assume table[3] = 0xA4, which blocks do we know are free?
- 0xA4 = 1010 0100
- Blocks: 3*8+2, 3*8+5, 3*8+7 (26, 29, 31)

Bit Vector

Assuming that the table is in memory:

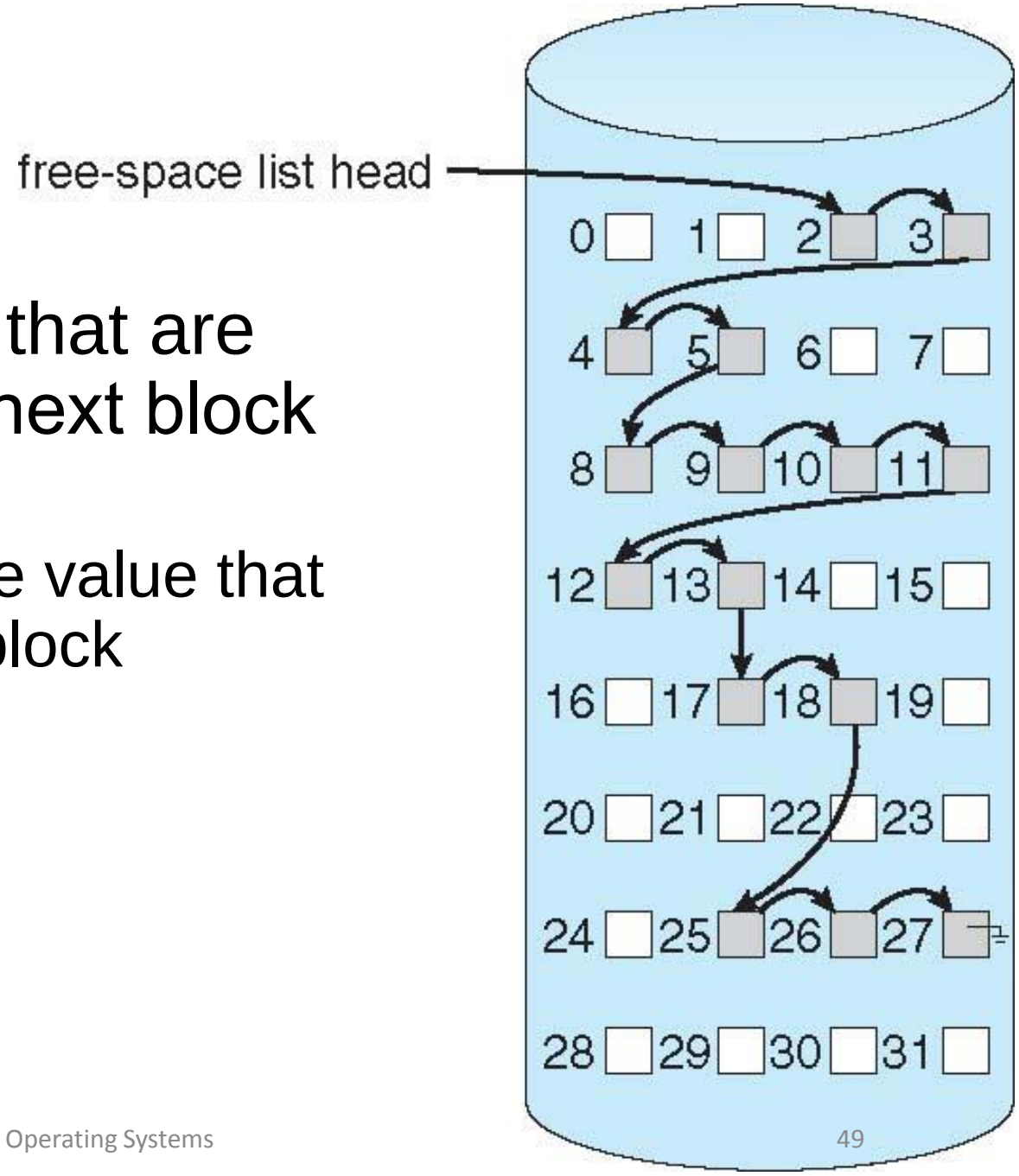
- Can quickly find an available block
  - What is the algorithm?
  - Scan table until we find a byte value that is non-zero
  - Then scan the bits in the byte, looking for a 1
- Can also quickly find a contiguous set of blocks
  - Scan table until we find a byte value of 0xff
  - This is a contiguous set of 8 blocks!

Bit Vectors in Project 3

- Bit vectors are used to represent the allocation table for the individual blocks
- Note that there are some differences in implementation from what we have done here:
  - One bit per block
  - 1 = used; 0 = free

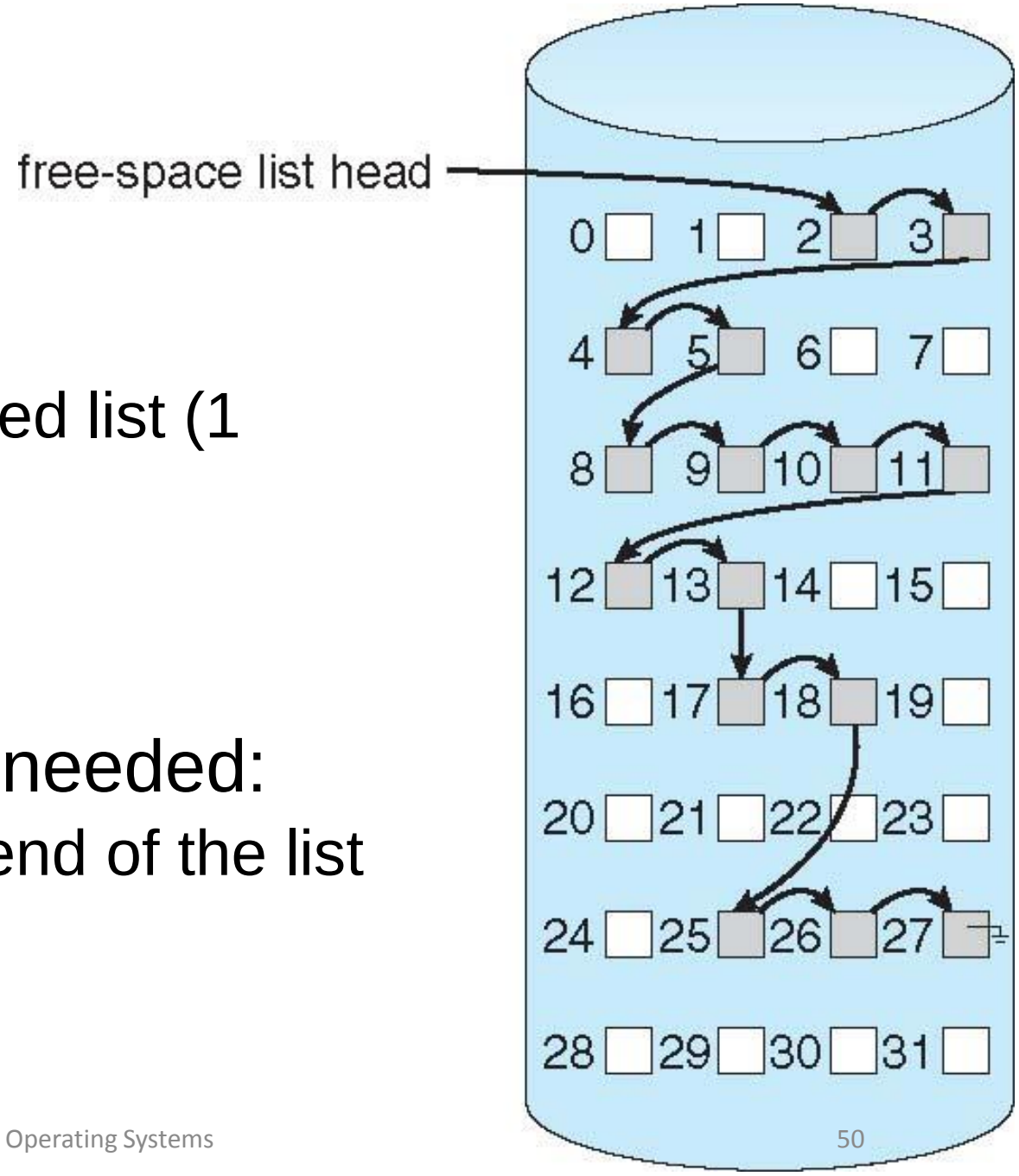
Free Space with Linked Lists

- Each block has several bytes that are dedicated to “pointing to” the next block in the list
  - Last block: contains a reference value that indicates that there is no next block



Free Space with Linked Lists

- Allocate a new block:
  - Take it from the front of the linked list (1 operation)
- Count number of free blocks:
  - Must traverse the entire list (!)
- Free a block that is no longer needed:
  - Can add it to the beginning or end of the list



Linked Lists

In addition to using the linked list for free blocks, we can also use the “next” pointer to represent the next data block in a file or directory

- This is exactly what we do in project 3!
  - One index node block can link to the next index node block
  - One directory block can link to the next block
- And for project 4!
  - One data block can link to the next data block

Efficiency and Performance

Different design choices can affect efficiency and performance:

- Disk allocation and directory algorithms
- Types of data kept in a file's directory entry
- Pre-allocation or as-needed allocation of metadata structures
- Fixed-size or varying-size data structures

Performance

- Keeping data and metadata close together
- **Buffer / cache:** separate section of main memory for frequently used blocks
- Sequential access optimizations for caches:
  - **Free-behind**: once user process starts accessing the next block in a file, we no longer need to cache the current block
  - **Read-ahead:** proactively fetch the next few blocks from the disk

Synchronous vs Asynchronous Writes

- **Synchronous** writes:
  - No buffering / caching
  - Writes must the hit disk before acknowledgement (i.e., return of the system call that is requesting the write)
  - This way, the process knows for sure that the data have been stored, but it has to wait for completion
- **Asynchronous** writes:
  - Write to a buffer and return immediately
  - At some point, the file system will write the buffered blocks out to the disk
  - More common, faster, and efficient
  - But, if there is a failure, the file changes may never reach the disk

File System Recovery

- Consistency checking: compare data in directory structure with data blocks on disk, and attempt to fix inconsistencies
  - Can be slow and sometimes fails, requiring human input
- Backups:
  - Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)
  - Recover lost file or disk by restoring data from backup

Log Structured File Systems

- Want the file system to be robust to failure at any time
  - In particular, the file system must always be in a consistent state
- Log-structured file systems:
  - Have an existing file structure
  - A write by a user process is recorded as an entry in a log file
  - At a later time, an effective copy of the file structure is made and the log of operations is executed
  - Then, the new state of the file system replaces the old copy in an atomic operation

Log Structured File Systems

If a crash happens somewhere, we either have:

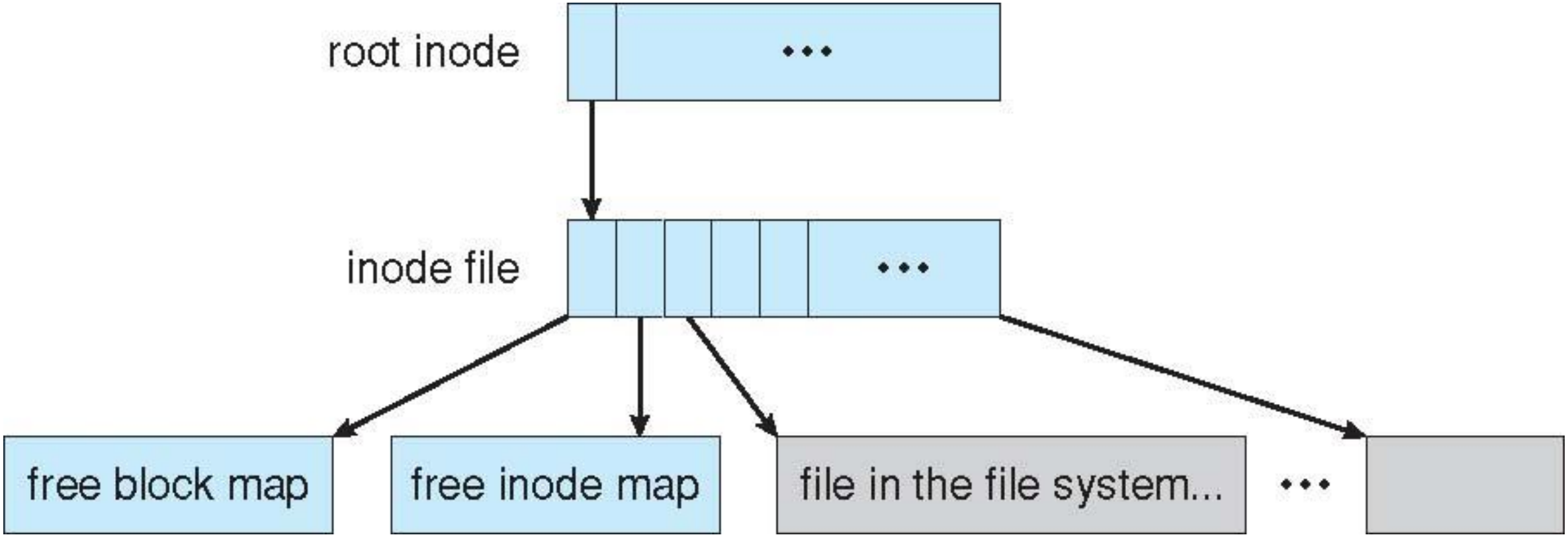
- The original state of the file system + (perhaps) the log of changes
- Or the new state of the file system

- We are never left with a file system that is in the middle of being modified

Example: WAFL File System

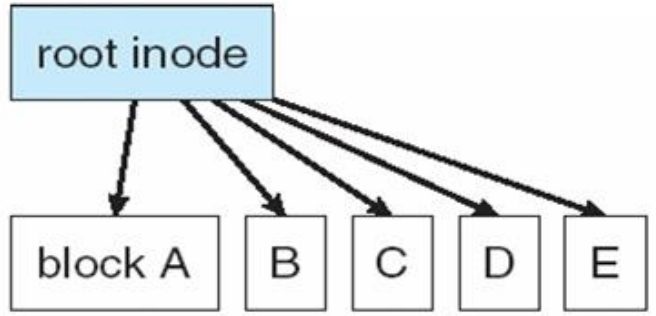
- Similar to Berkeley Fast File System, with extensive modifications
- Used on Network Appliance “Filers”: distributed file system appliances
- Serves up a variety of network protocols: NFS, CIFS, http, ftp
- Random I/O optimized, write optimized
- NVRAM for write caching

The WAFL File Layout

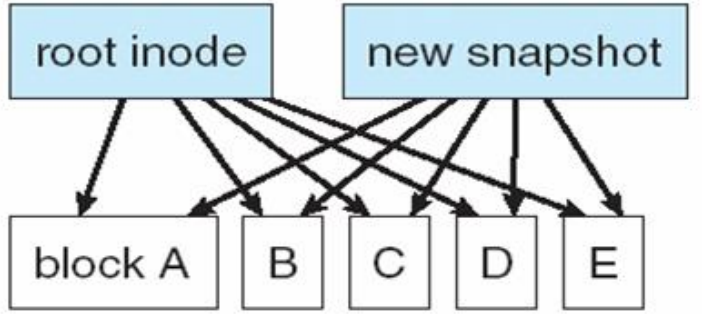


Snapshots in WAFL

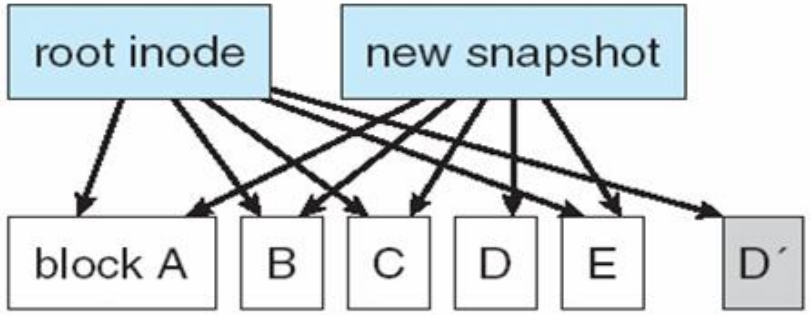
- New snapshot maintains an old copy of the original file system
- Original data are left in place
- Any changes to blocks are made to new copies



(a) Before a snapshot.



(b) After a snapshot, before any blocks change.



(c) After block D has changed to D´.