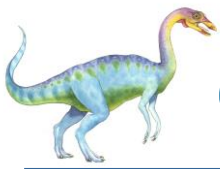


# Chapter 14: File System Implementation

---





# Chapter 14: File System Implementation

---

1. File-System Structure
2. File-System Implementation (Operations)
3. Directory Implementation
4. Allocation Methods
  - 4.1 contiguous
  - 4.2 linked
  - 4.3 indexed
5. Free-Space Management
6. Efficiency and Performance
7. Recovery
- A. Example: WAFL File System





# Objectives

---

To describe the details of **implementing local file systems (not logical) and directory structures**

To describe the implementation of remote file systems (제외)

To discuss **block allocation and free-block algorithms** and trade-offs





# 1. File-System Structure

---

File structure

Logical storage unit (13장 remind)

Collection of related information

**File system** resides on secondary storage (disks) : 아래 매우 중요

Provided **user interface to storage**, mapping logical to physical

Provides **efficient and convenient access to disk** by allowing data to be stored, located retrieved easily

Disk provides in-place rewrite and random access

I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)

**File control block (FCB, PCB remind)** – storage structure consisting of information about a file

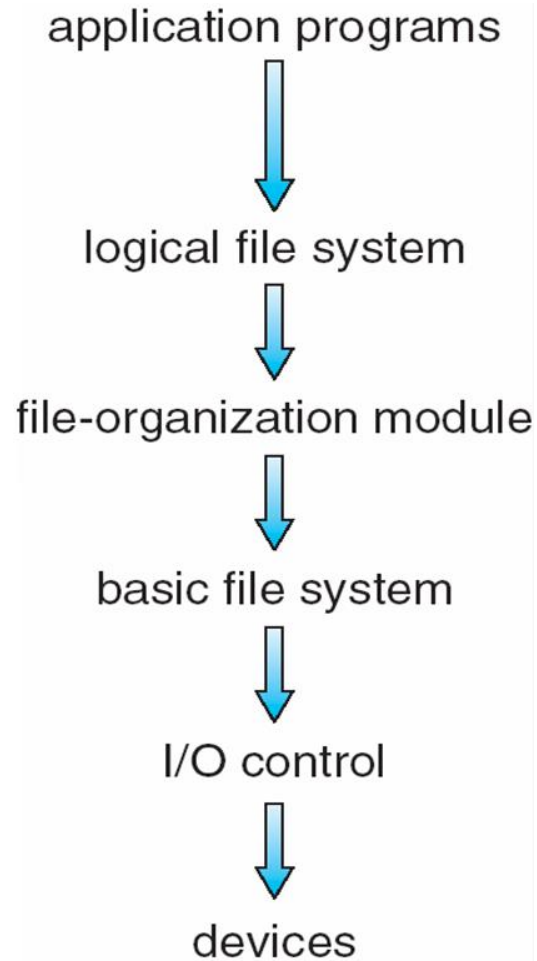
**Device driver** controls the **physical device** (중요한 S/W & 제조사)

File system organized into **layers** (mapping, mapping & mapping !!!)





# Layered File System (꼭 이해해야함)





# File System Layers

---

**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 (**I/O 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





# File System Layers (Cont.)

---

**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

**Layering** useful for reducing complexity and redundancy, but adds overhead and can decrease performance (**why???**)

**Logical layers** can be implemented by any coding method according to OS designer





# File System Layers (Cont.)

---

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** ext2 and ext3 leading; plus distributed file systems, etc.)

New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE







## 2. File-System Implementation

---

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

Both On-disk and in-memory structures

**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





# File-System Implementation (Cont.)

Per-file **File Control Block (FCB)** contains many details about the file

inode number, permissions, size, dates (Unix)

NFTS stores into in master file table using relational DB structures

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

---

Mount table storing file system mounts, mount points, file system types

The following figure illustrates the necessary file system structures provided by the operating systems

Figure 12-3(a) refers to opening a file

Figure 12-3(b) refers to reading a file

Plus buffers hold data blocks from secondary storage

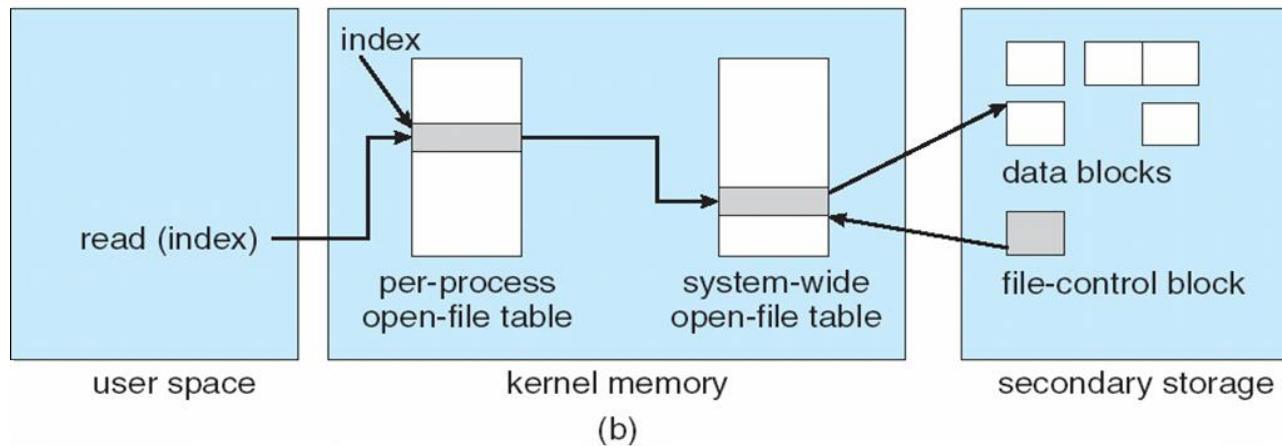
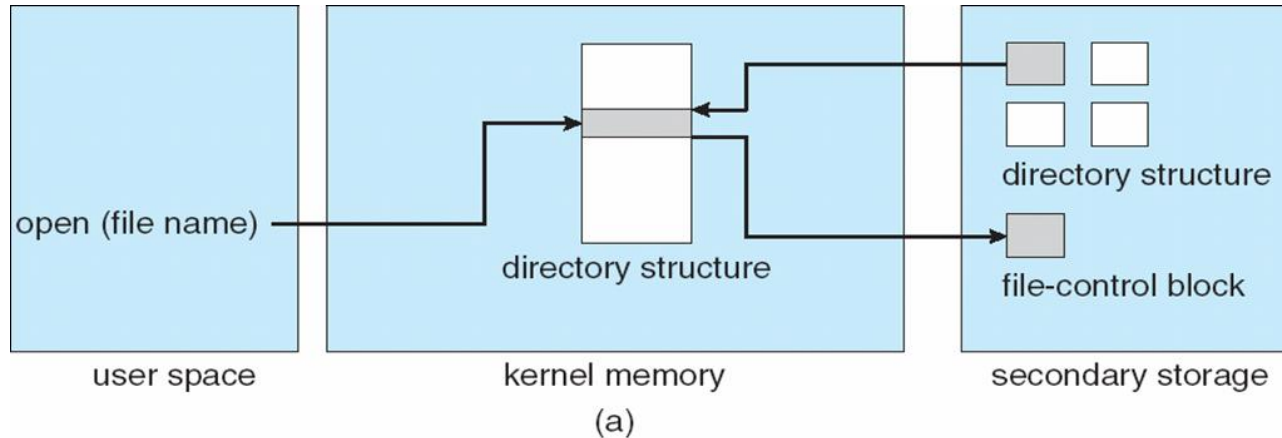
Open returns a file handle for subsequent use

Data from read eventually copied to specified user process memory address





# In-Memory File System Structures





# 3. Directory Implementation

---

**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





## 4. Allocation Methods

---

An allocation method refers to how disk blocks are allocated for files:

**Contiguous**

**Linked**

**Indexed**





## 4..1 Contiguous

---

**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

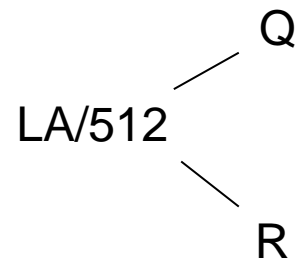
Problems include finding space for file, knowing file size, external fragmentation, need for **compaction off-line** (**downtime**) or **on-line**





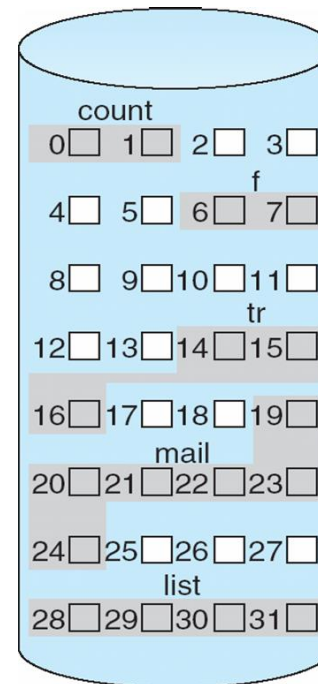
# Contiguous Allocation

Mapping from logical to physical



Block to be accessed = Q +  
starting address

Displacement into block = R



directory

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







# Extent-Based Systems

---

Many newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme

Extent-based file systems allocate disk blocks in extents

An **extent** is a contiguous block of disks

Extents are allocated for file allocation

A file consists of one or more extents





## 4.2 Linked

---

**Linked allocation** – each file a linked list of blocks

- File ends at nil pointer

- No external fragmentation

- Each block contains pointer to next block

- No compaction, external fragmentation

- Free space management system called when new block needed

- Improve efficiency by clustering blocks into groups but increases internal fragmentation

- Reliability can be a problem

- Locating a block can take many I/Os and disk seeks





# Allocation Methods – Linked (Cont.)

---

FAT (File Allocation Table) variation

Beginning of volume has table, indexed by block number

Much like a linked list, but faster on disk and cacheable

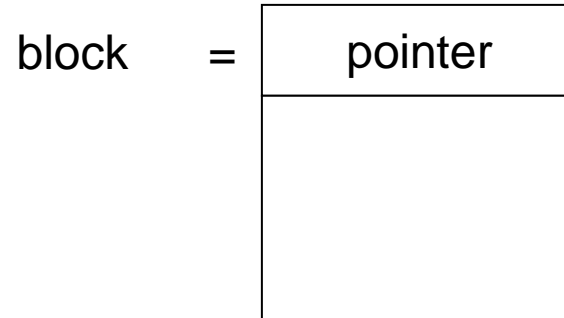
New block allocation simple



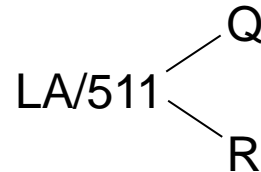


# Linked Allocation

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



Mapping



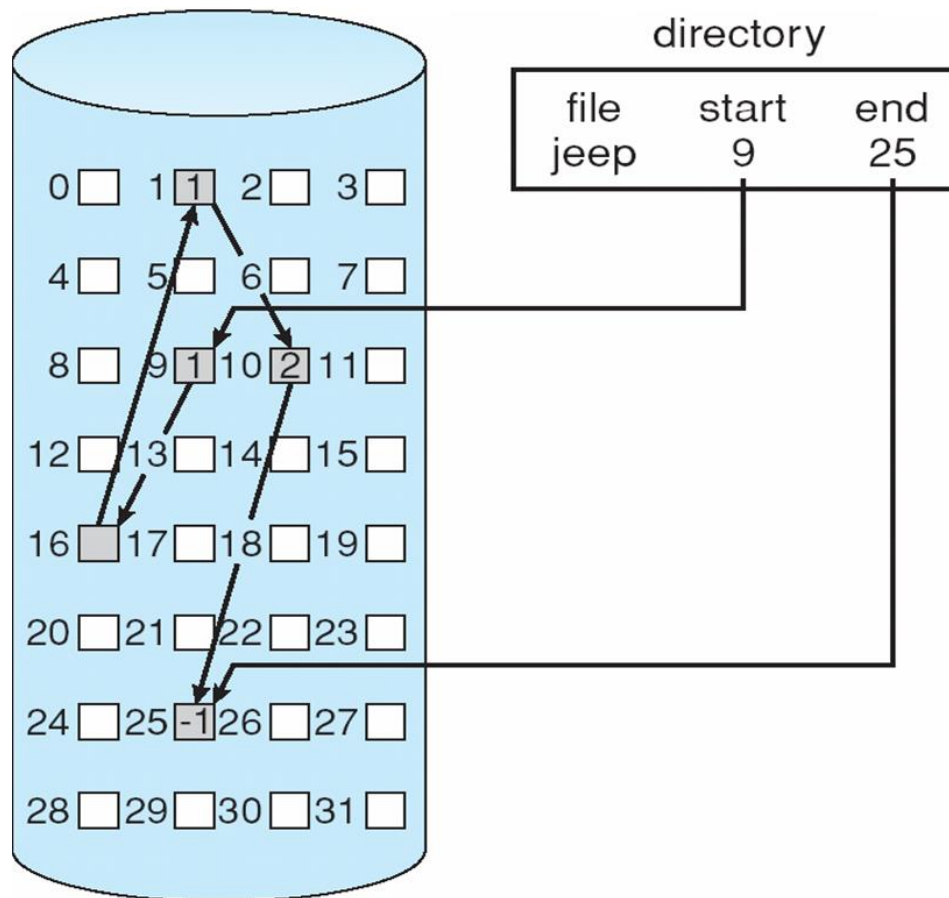
Block to be accessed is the Qth block in the linked chain of blocks representing the file.

Displacement into block =  $R + 1$



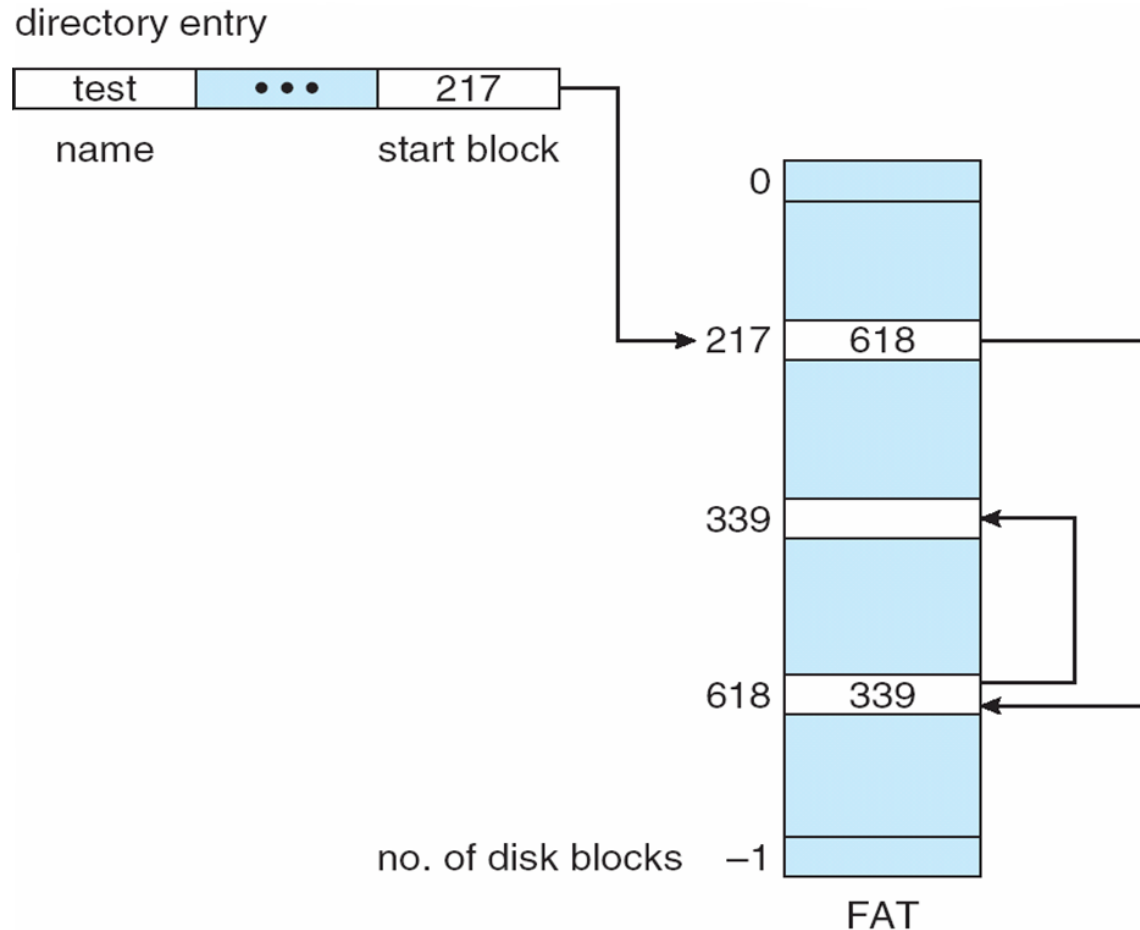


# Linked Allocation





# File-Allocation Table



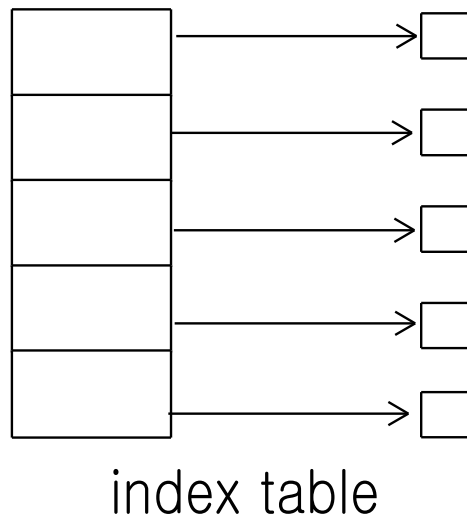


## 4.3 Indexed

### Indexed allocation

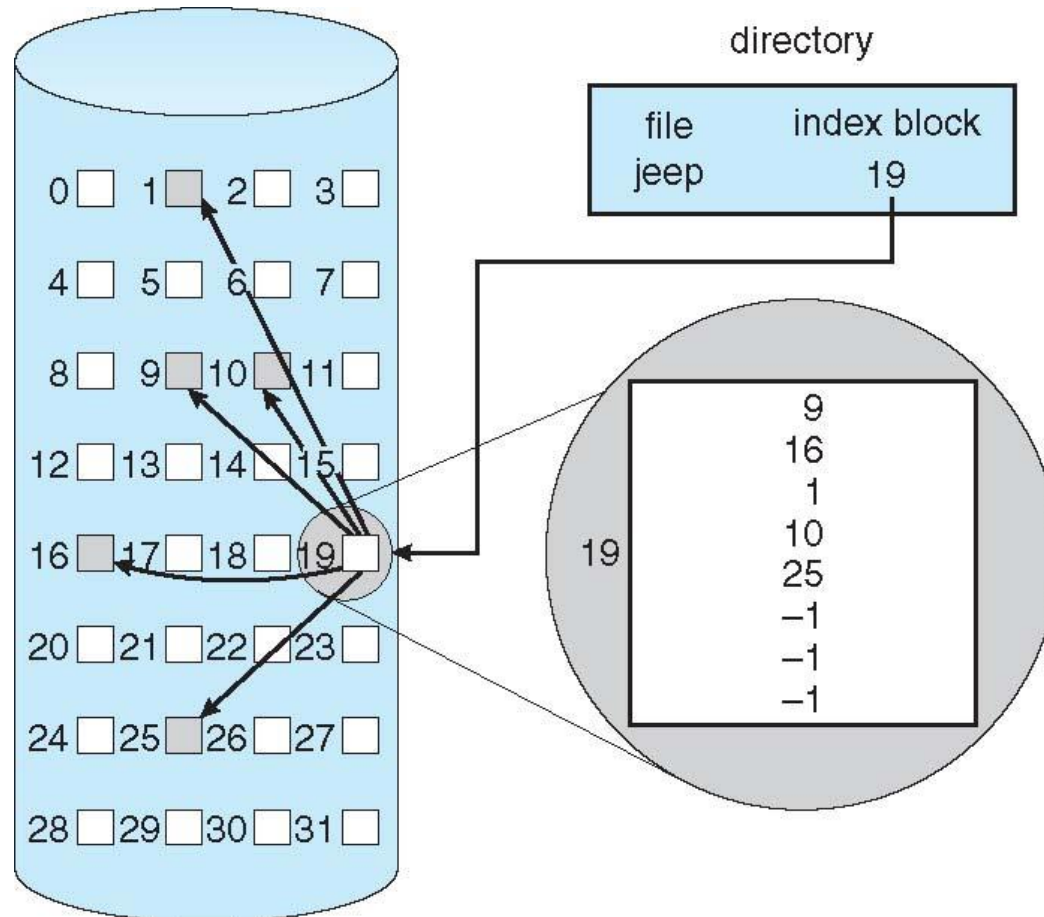
Each file has its own **index block**(s) of pointers to its data blocks

Logical view





# Example of Indexed Allocation







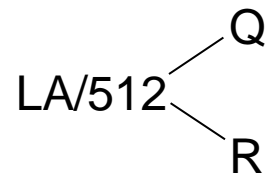
# Indexed Allocation (Cont.)

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 bytes and block size of 512 bytes. 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)

$$LA / (512 \times 511) \begin{cases} Q_1 \\ R_1 \end{cases}$$

$Q_1$  = block of index table

$R_1$  is used as follows:

$$R_1 / 512 \begin{cases} Q_2 \\ R_2 \end{cases}$$

$Q_2$  = displacement into block of index table

$R_2$  displacement into block of file:





# Indexed Allocation – Mapping (Cont.)

Two-level index (4K blocks could store 1,024 four-byte pointers in outer index -> 1,048,567 data blocks and file size of up to 4GB)

$$LA / (512 \times 512) \begin{cases} Q_1 \\ R_1 \end{cases}$$

$Q_1$  = displacement into outer-index

$R_1$  is used as follows:

$$R_1 / 512 \begin{cases} Q_2 \\ R_2 \end{cases}$$

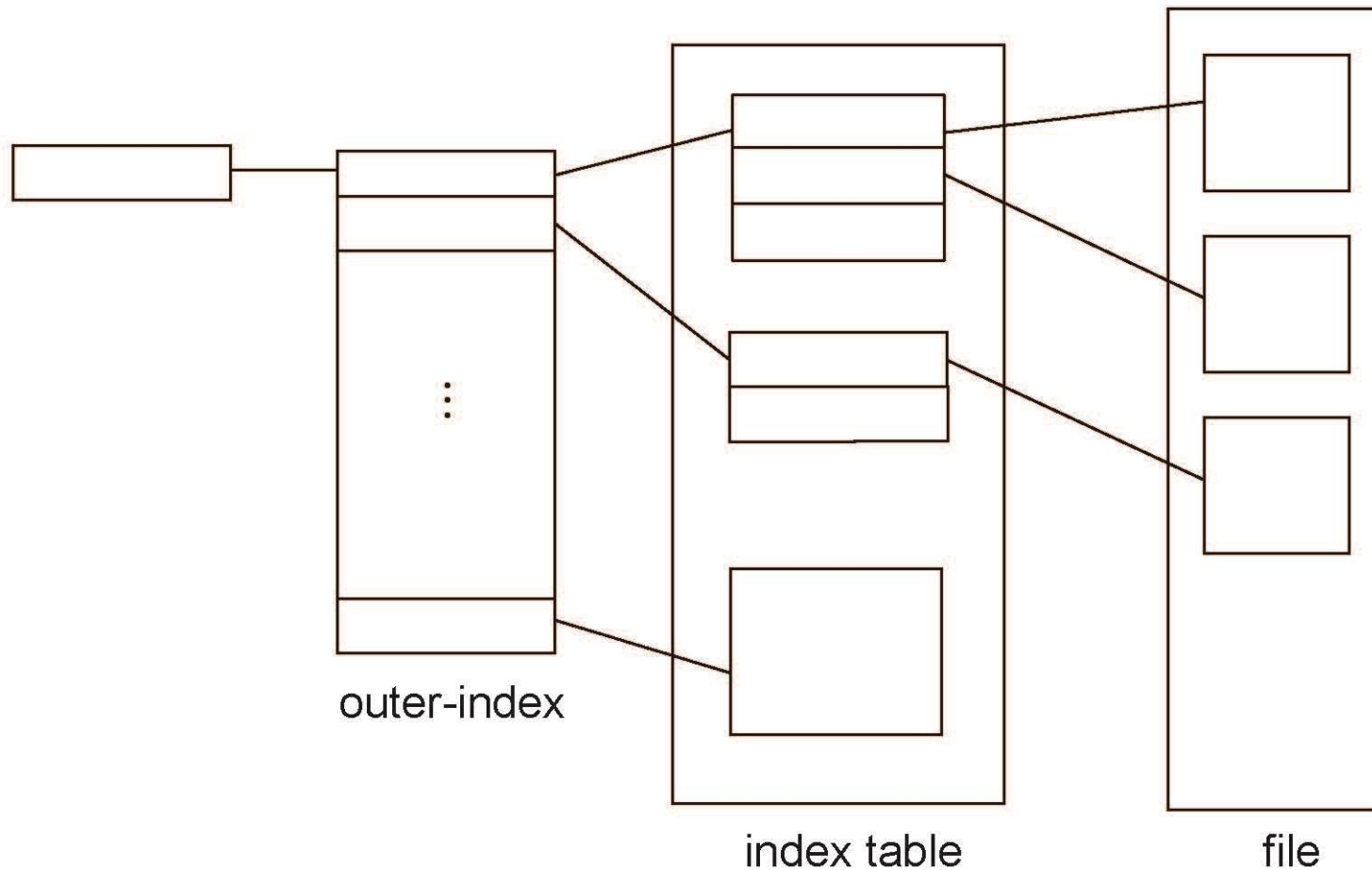
$Q_2$  = displacement into block of index table

$R_2$  displacement into block of file:





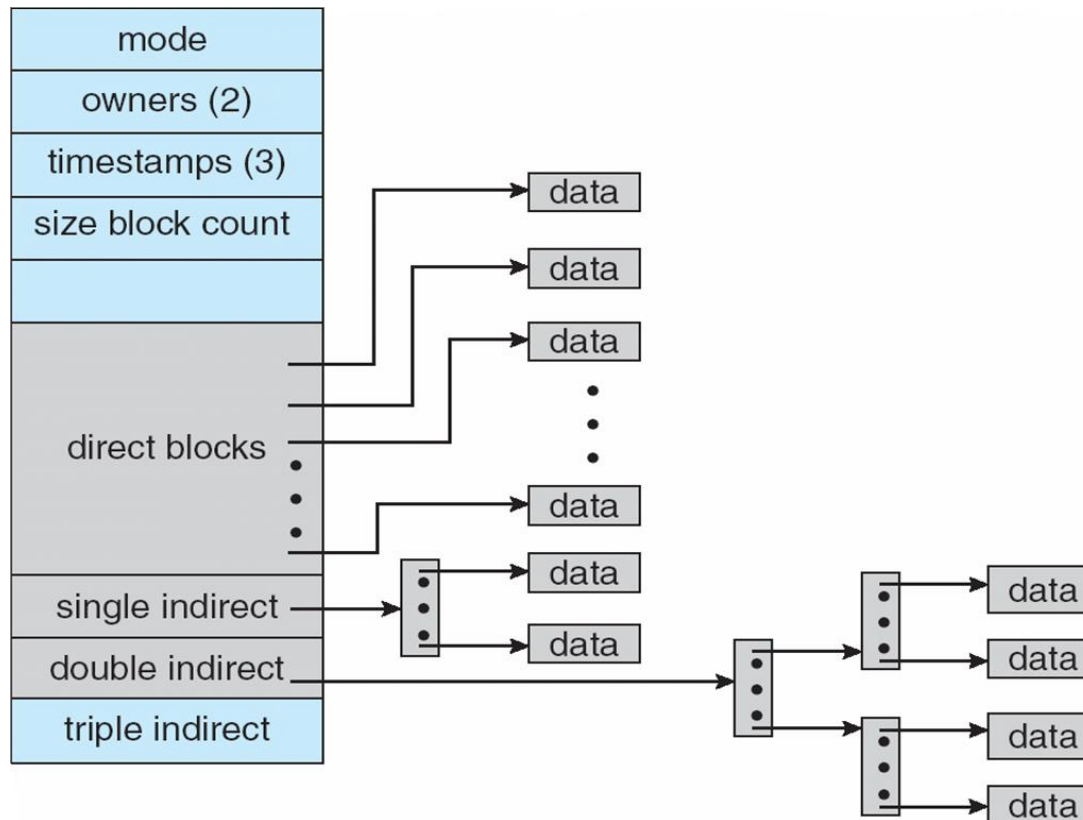
# Indexed Allocation – Mapping (Cont.)





# Combined Scheme: UNIX UFS

4K bytes per block, 32-bit addresses



More index blocks than can be addressed with 32-bit file pointer





# Performance

---

Best method depends on file access type

**Contiguous great for sequential and random**

Linked good for sequential, not random

Declare access type at creation -> select either contiguous or linked

Indexed more complex

Single block access could require 2 index block reads then data block read

Clustering can help improve throughput, reduce CPU overhead





# Performance (Cont.)

---

Adding instructions to the execution path to save one disk I/O is reasonable

Intel Core i7 Extreme Edition 990x (2011) at 3.46Ghz = 159,000 MIPS

- ▶ [http://en.wikipedia.org/wiki/Instructions\\_per\\_second](http://en.wikipedia.org/wiki/Instructions_per_second)

Typical disk drive at 250 I/Os per second

- ▶  $159,000 \text{ MIPS} / 250 = 630$  million instructions during one disk I/O

Fast SSD drives provide 60,000 IOPS

- ▶  $159,000 \text{ MIPS} / 60,000 = 2.65$  millions instructions during one disk I/O

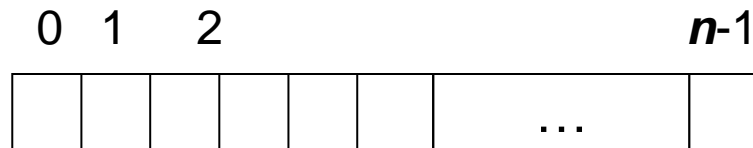




## 5. Free-Space Management

File system maintains **free-space list** to track available blocks/clusters  
(Using term “block” for simplicity)

**Bit vector** or **bit map** ( $n$  blocks)



$$\text{bit}[i] = \begin{cases} 1 \Rightarrow \text{block}[i] \text{ free} \\ 0 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

Block number calculation

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

CPUs have instructions to return offset within word of first “1” bit







# Free-Space Management (Cont.)

---

Bit map requires extra space

Example:

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

Easy to get contiguous files





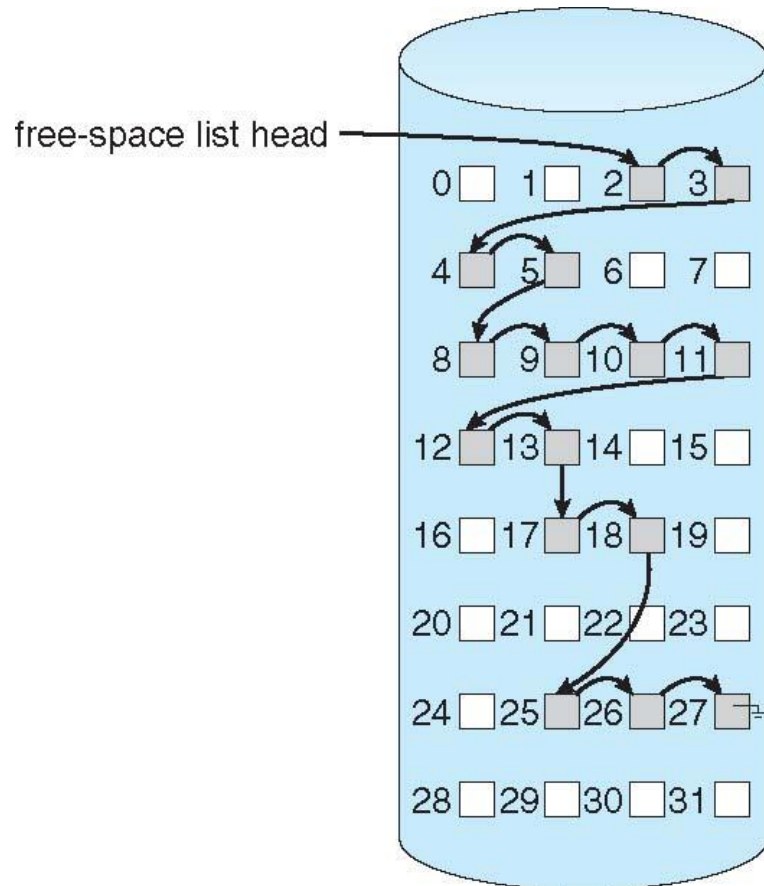
# Linked Free Space List on Disk

Linked list (free list)

Cannot get contiguous space easily

No waste of space

No need to traverse the entire list (if # free blocks recorded)





# Free-Space Management (Cont.)

---

## Grouping

Modify linked list to store address of next  $n-1$  free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this one)

## Counting

Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering

- ▶ Keep address of first free block and count of following free blocks
- ▶ Free space list then has entries containing addresses and counts





# Free-Space Management (Cont.)

---

## Space Maps

Used in **ZFS**

Consider meta-data I/O on very large file systems

- ▶ Full data structures like bit maps couldn't fit in memory -> thousands of I/Os

Divides device space into **metaslab** units and manages metaslabs

- ▶ Given volume can contain hundreds of metaslabs

Each metaslab has associated space map

- ▶ Uses counting algorithm

But records to log file rather than file system

- ▶ Log of all block activity, in time order, in counting format

Metaslab activity -> load space map into memory in balanced-tree structure, indexed by offset

- ▶ Replay log into that structure
- ▶ Combine contiguous free blocks into single entry





## 6. Efficiency and Performance

---

Efficiency dependent on:

- Disk allocation and directory algorithms

- Types of data kept in file's directory entry

- Pre-allocation or as-needed allocation of metadata structures

- Fixed-size or varying-size data structures





# Efficiency and Performance (Cont.)

---

## Performance

Keeping data and metadata close together

**Buffer cache** – separate section of main memory for frequently used blocks

**Synchronous** writes sometimes requested by apps or needed by OS

- ▶ No buffering / caching – writes must hit disk before acknowledgement
- ▶ **Asynchronous** writes more common, buffer-able, faster

**Free-behind** and **read-ahead** – techniques to optimize sequential access

Reads frequently slower than writes





# Page Cache

---

A **page cache** caches pages rather than disk blocks using virtual memory techniques and addresses

Memory-mapped I/O uses a page cache

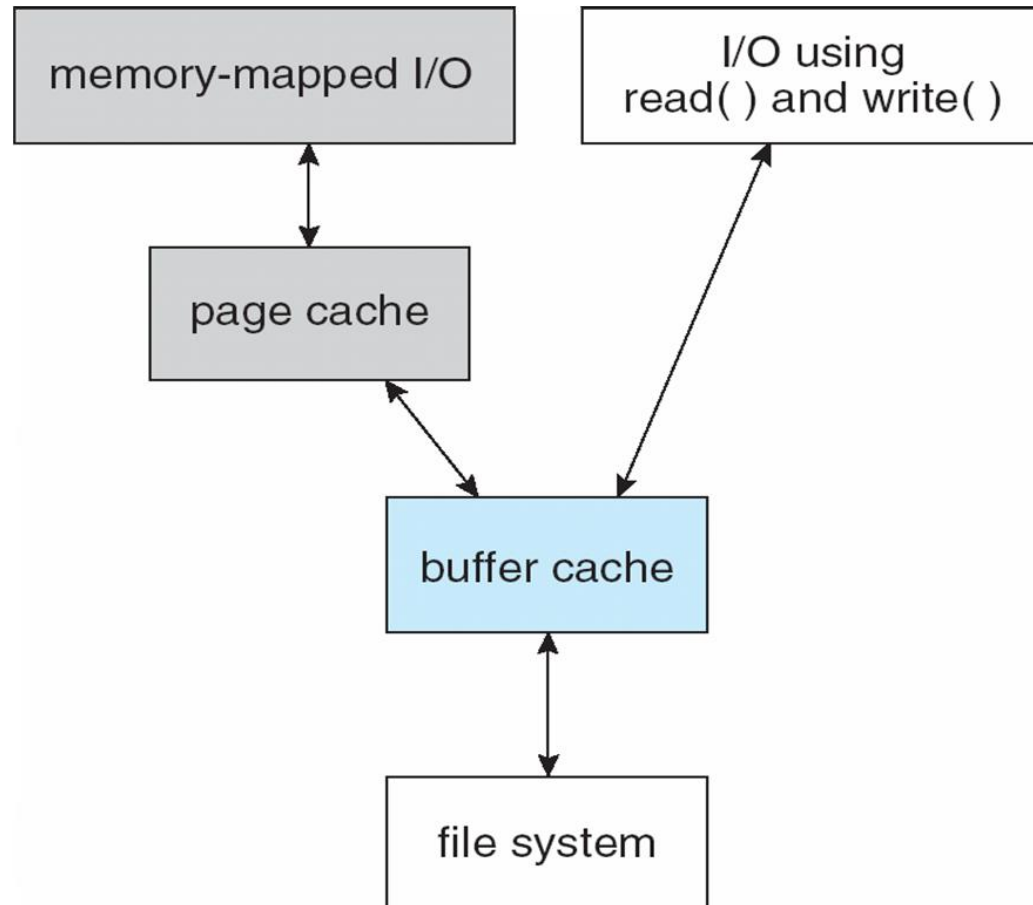
Routine I/O through the file system uses the buffer (disk) cache

This leads to the following figure





# I/O Without a Unified Buffer Cache







# Unified Buffer Cache

---

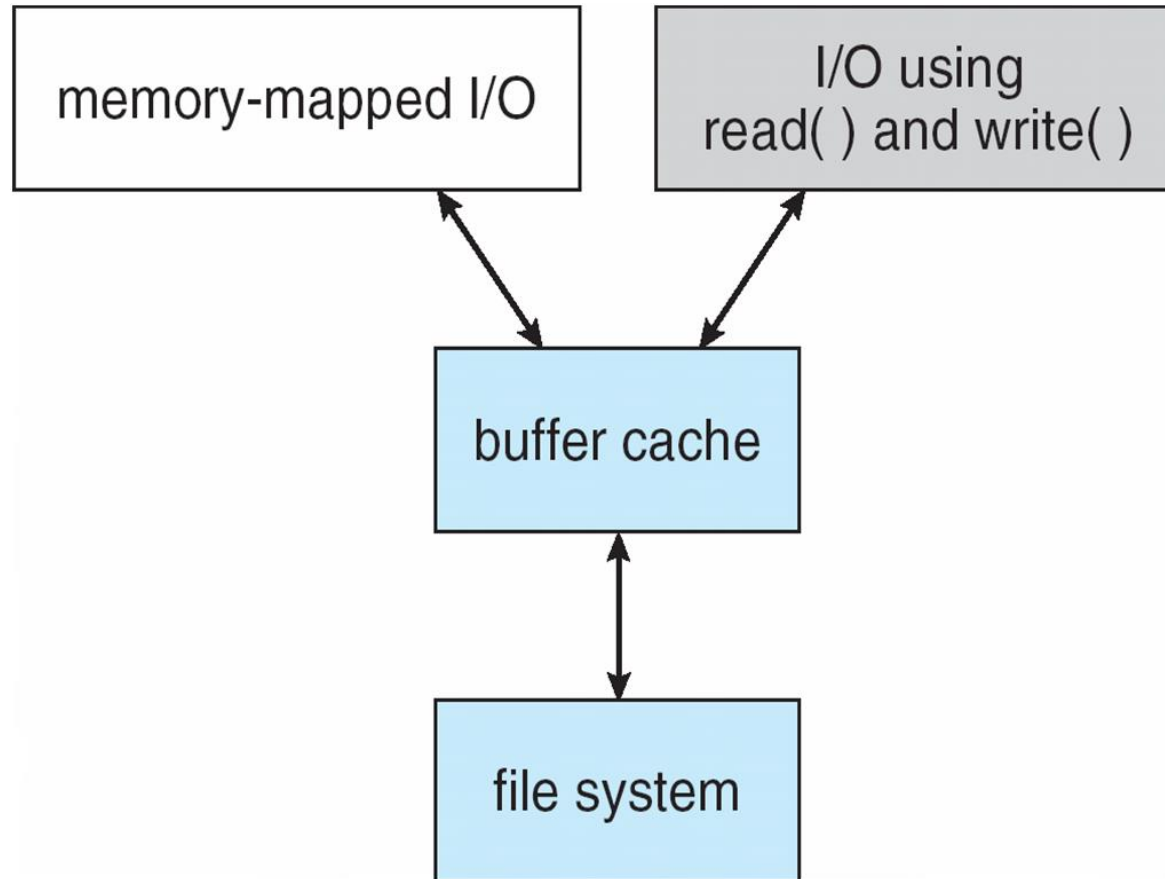
A **unified buffer cache** uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid **double caching**

But which caches get priority, and what replacement algorithms to use?





# I/O Using a Unified Buffer Cache





# 7. Recovery

---

**Consistency checking** – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies

Can be slow and sometimes fails

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

---

**Log structured** (or **journaling**) file systems record each metadata 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 (sequentially)

Sometimes to a separate device or section of disk

However, the file system may not yet be updated

The transactions in the log are asynchronously written to the file system structures

When the file system structures are modified, the transaction is removed from the log

If the file system crashes, all remaining transactions in the log must still be performed

Faster recovery from crash, removes chance of inconsistency of metadata





# A. Example: WAFL File System

---

Used on Network Appliance “Filers” – distributed file system appliances

“Write-anywhere file layout”

Serves up NFS, CIFS, http, ftp

Random I/O optimized, write optimized

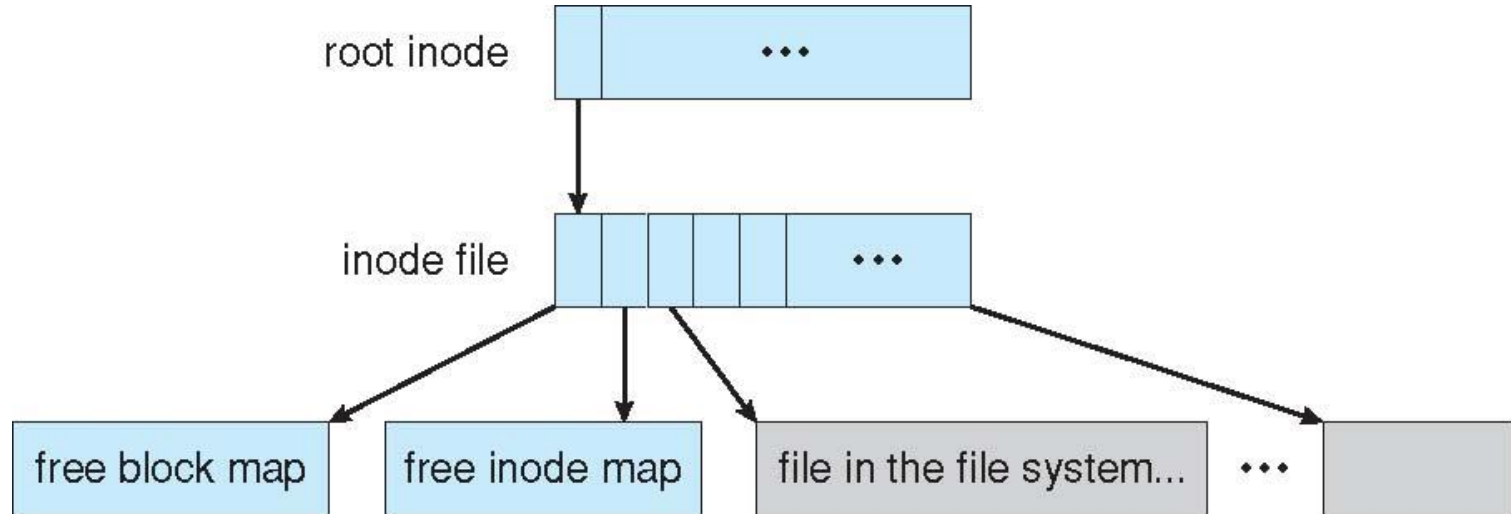
NVRAM for write caching

Similar to Berkeley Fast File System, with extensive modifications



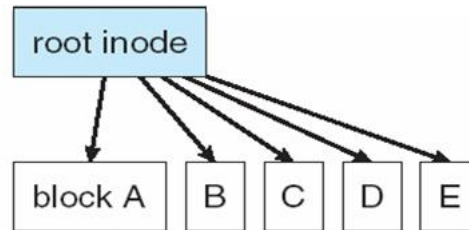


# The WAFL File Layout

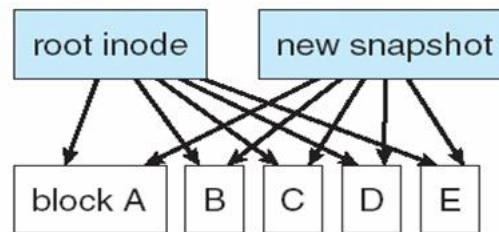




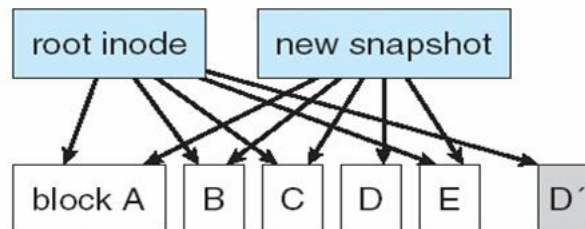
# Snapshots in WAFL



(a) Before a snapshot.



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



(c) After block D has changed to D'.



# End of Chapter 14

---

