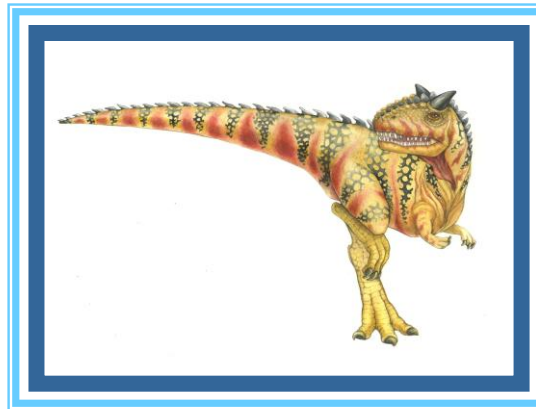# Chapter 14: File System Implementation

Chapter 14: File System Implementation

- File-System Structure
- File-System Operations
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery
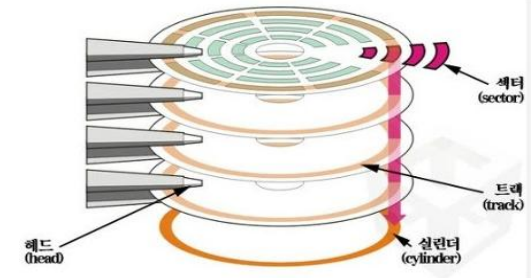- Example: WAFL File System

Objectives

- Describe the details of implementing local file systems and directory structures

- Discuss block allocation and free-block algorithms and trade-offs

- Explore file system efficiency and performance issues

- Look at recovery from file system failures
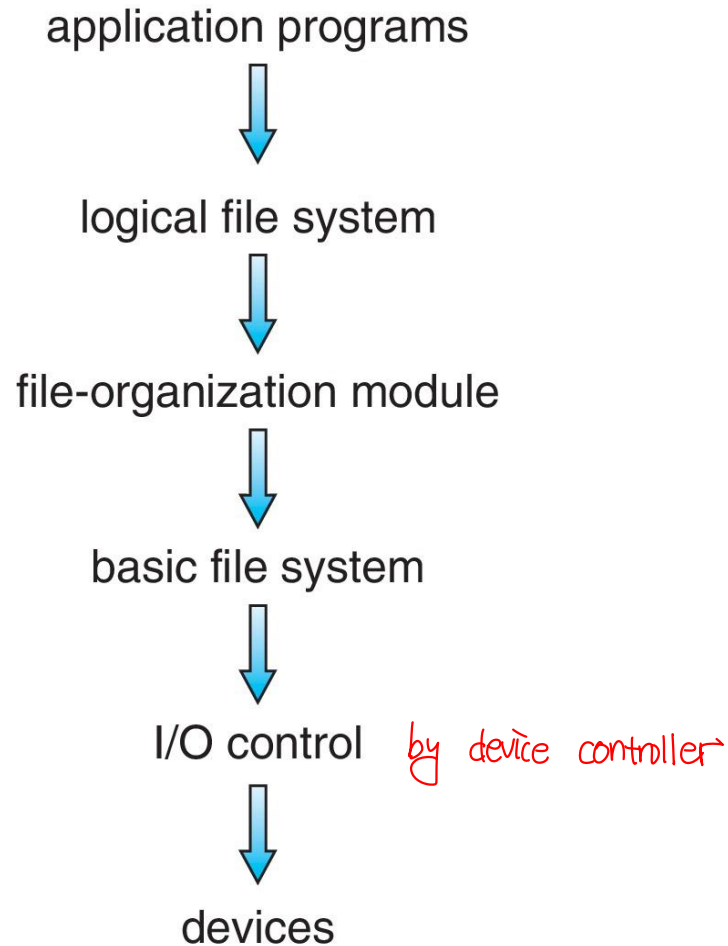
- Describe the WAFL file system as a concrete example

File-System Structure

- File structure
  - **Logical** storage unit
  - **Collection of related information**
- **File system** resides **on secondary storage** (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
- Disk provides **in-place rewrite and random access**
  - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- **File control block** (**FCB**) – **storage structure consisting of information about a file**
- **Device driver** **controls the physical device**
- File system **organized into layers**

Layered File System

application programs

⬇

logical file system

⬇

file-organization module

⬇

basic file system

⬇

I/O control   *by device controller*

⬇

devices

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
- **Basic file system** given command like "retrieve block 123" translates to device driver *block 단위 읽기/쓰기 요청 수행.*
  - 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
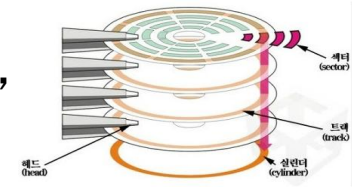
File System Layers (Cont.)

- **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
    - 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 130 types, with **extended file system** ext3 and ext4 leading; plus distributed file systems, etc.)

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

File-System Operations

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

on-disk

File-System Implementation (Cont.)

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

  on-disk

  - typically inode number, permissions, size, dates

  - NTFS stores into 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

- **system-wide open-file table** contains a copy of the FCB of each file and other info 시스템 전체에서 열린 파일들의 FCB 사본 보관.

- **per-process open-file table** contains pointers to appropriate entries in system-wide open-file table as well as other info 프로세스 별 열린 파일 목록.

- 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
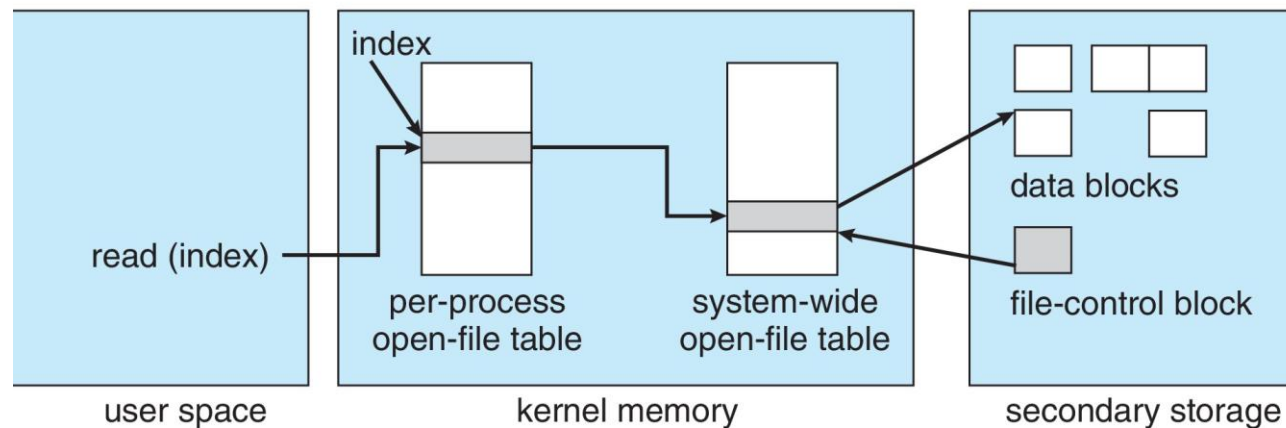
In-Memory File System Structures
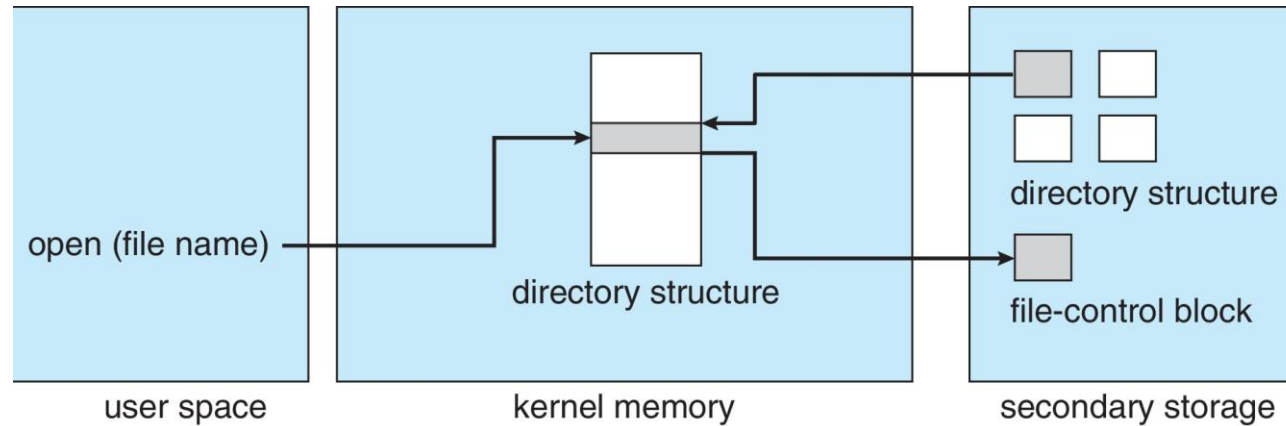
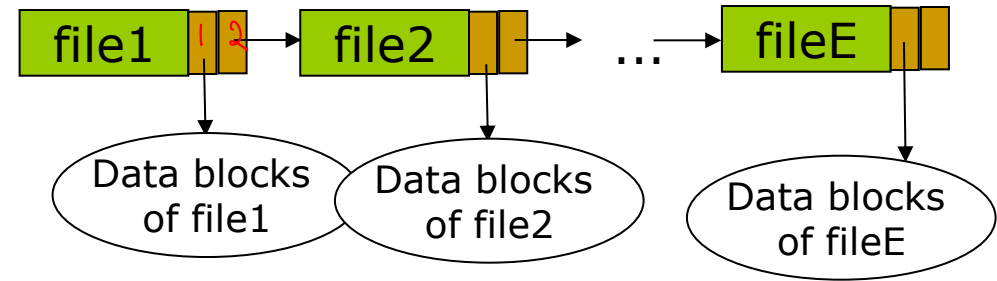(a) Opening a file

(b) Reading a file
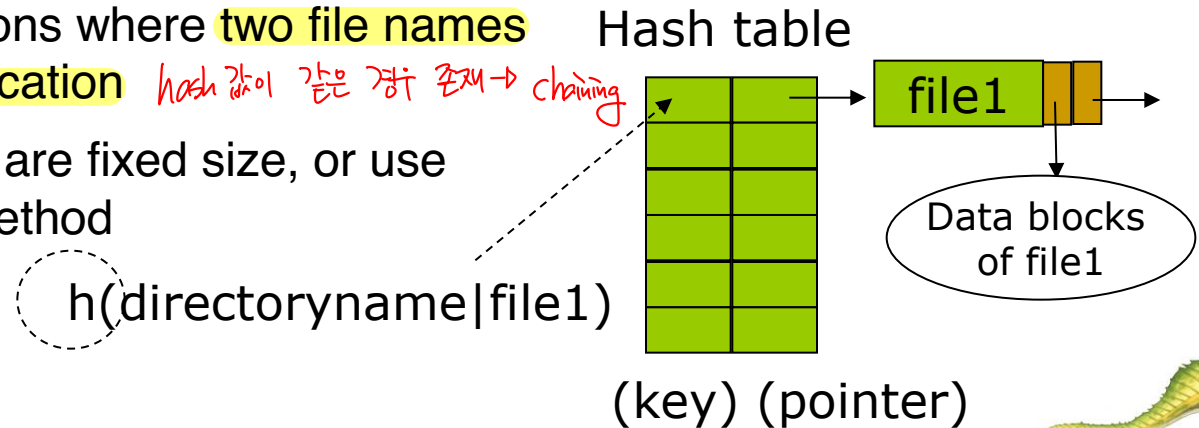
Directory Implementation

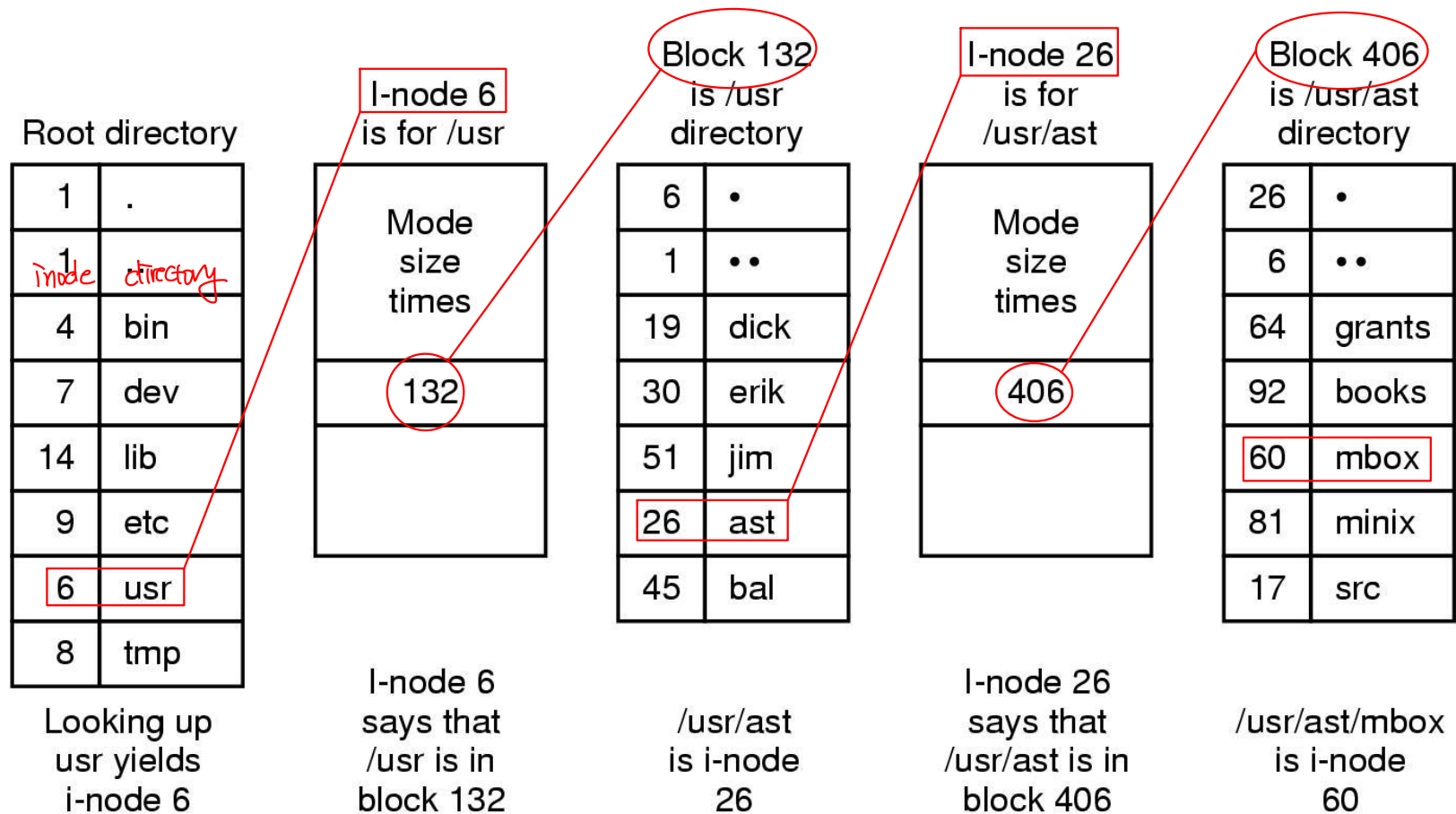- **Linear list** of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - ▸ Linear search time
    - ▸ Could keep ordered alphabetically via linked list

file1 | 1 2 → file2 → ... → fileE

Data blocks of file1   Data blocks of file2   Data blocks of fileE

- **Hash Table** – linear list with hash data structure
  - Decreases directory search time
  - **Collisions** – situations where two file names hash to the same location *hash값이 같은 경우 존재 → chaining*
  - Only good if entries are fixed size, or use chained-overflow method

h(directoryname|file1)

Hash table

file1

Data blocks of file1

(key) (pointer)

Root directory	
1	.
1	..
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

(handwritten: inode directory)

Looking up
usr yields
i-node 6

I-node 6
is for /usr

Mode
size
times

132

I-node 6
says that
/usr is in
block 132

Block 132
is /usr
directory

6	.
1	..
19	dick
30	erik
51	jim
26	ast
45	bal

/usr/ast
is i-node
26

I-node 26
is for
/usr/ast

Mode
size
times

406

I-node 26
says that
/usr/ast is in
block 406

Block 406
is /usr/ast
directory

26	.
6	..
64	grants
92	books
60	mbox
81	minix
17	src

/usr/ast/mbox
is i-node
60

* The steps in looking up /usr/ast/mbox.

Allocation Methods - Contiguous

- 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
        - Random access is possible
    - 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

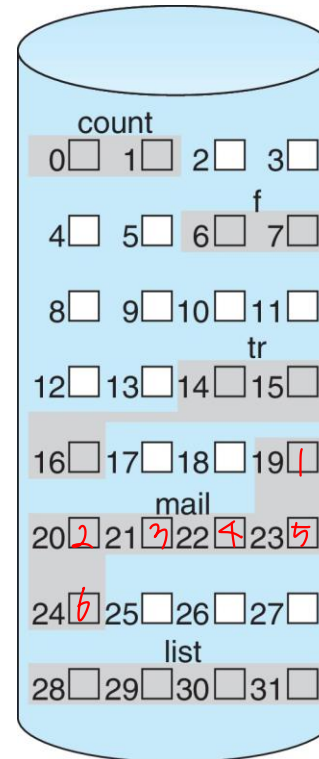- Mapping from logical to physical

Logical #:
3rd block of mail

⬇

Find the
starting block
of 'mail' → 19

⬇

Add displacement (=2)
➔ 19 + 2 = 21 (Physical block #)

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

Initial allocation (N contiguous block)(1st extent)
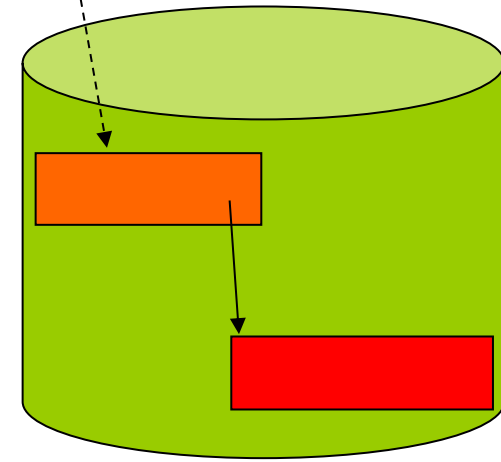
Next allocation (2nd extent)

Allocation Methods - Linked

- **Linked allocation** – each file a <mark>linked list of blocks</mark>
    - File ends at nil pointer
    - <mark>No external fragmentation</mark>
    - Each block contains <mark>pointer to next block</mark>
    - <mark>No compaction, external fragmentation</mark>
    - <mark>Free space management system called</mark> when new block needed
    - <mark>No random access</mark>
    - <mark>Reliability can be a problem</mark>
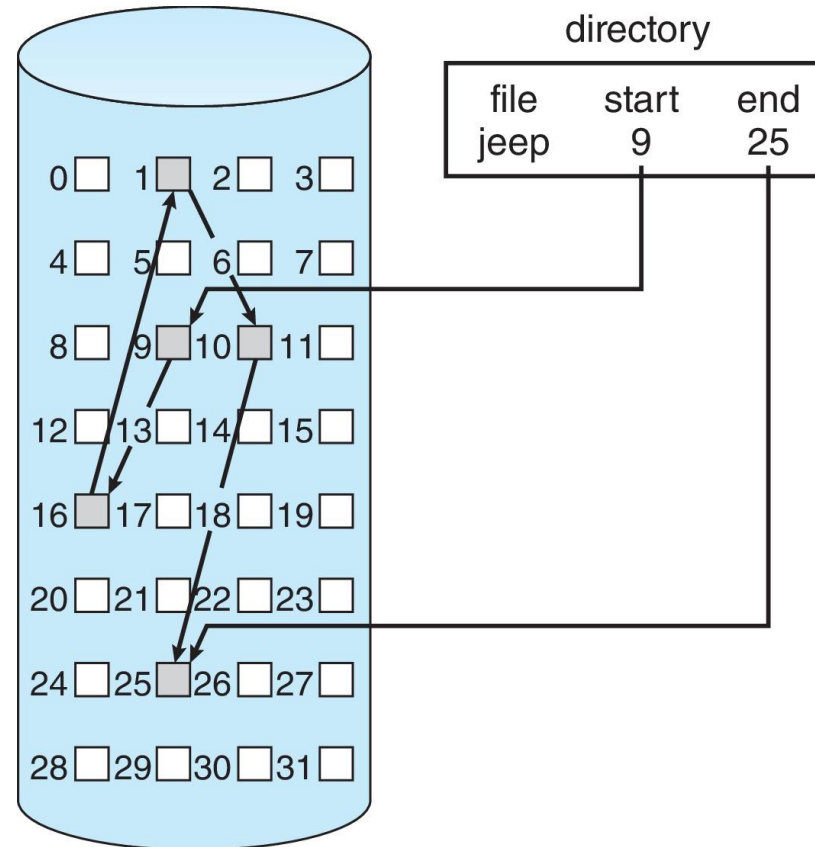   - Locating a block can take <mark>many I/Os and disk seeks</mark>

block =

pointer
data

Linked Allocation

directory

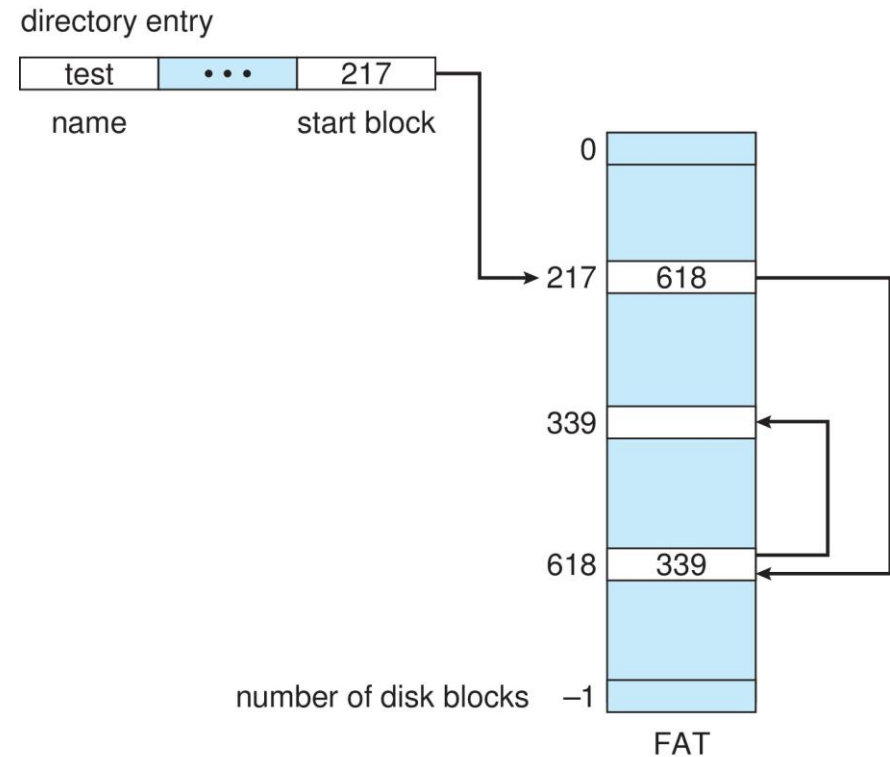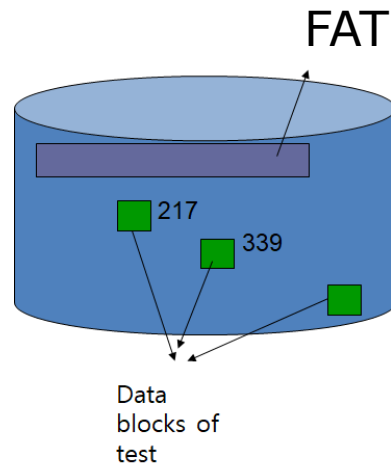| file	start	end
jeep	9	25

link lost
⇒ cannot retrieve next link

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
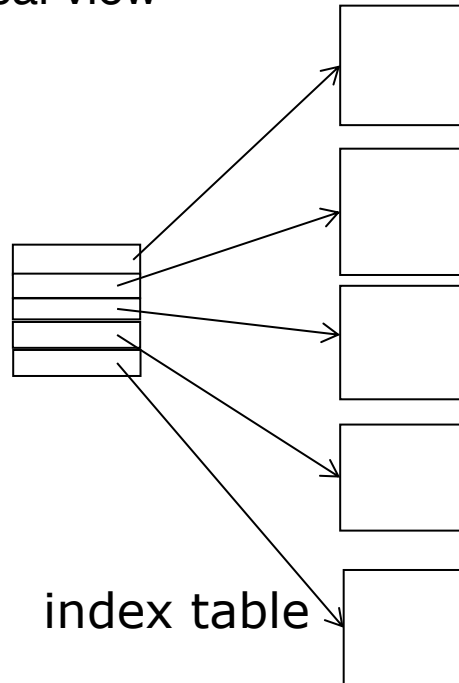 - New block allocation simple

directory entry

test	• • •	217
name		start block

FAT
217
339

Data blocks of test

0
217 | 618
339
618 | 339
number of disk blocks −1

FAT

□ **Indexed allocation**

multi-level indexing due to limitation of # index in a single block
⇒ cover more range

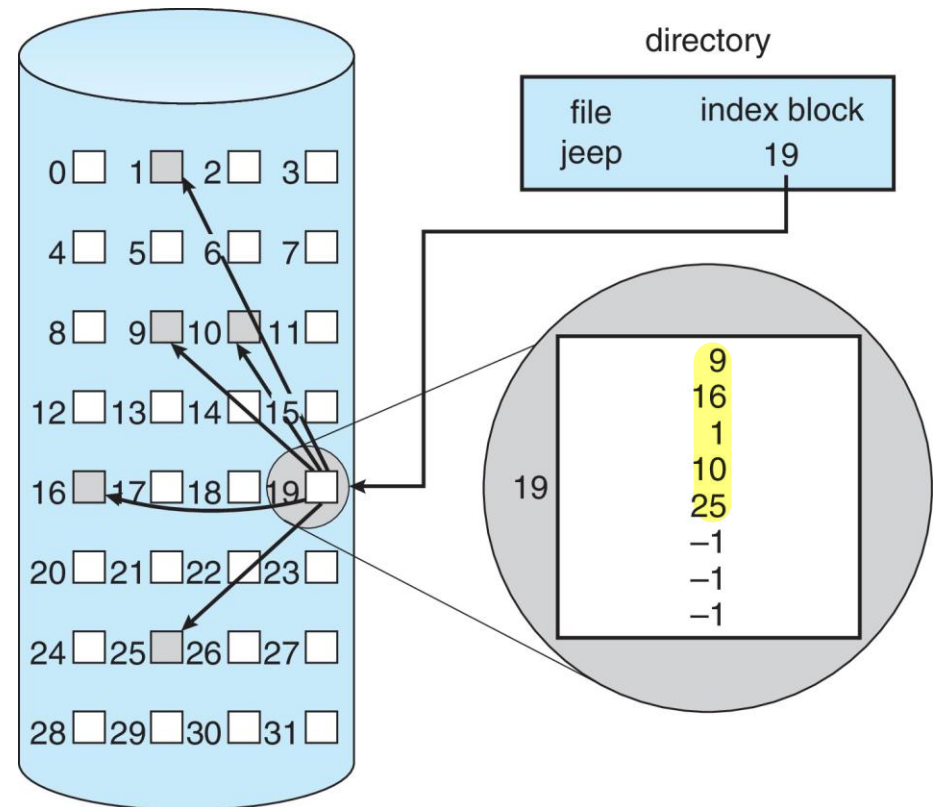   □ Each file has its own **index block**(s) of <mark>pointers to its data blocks</mark>

□ Logical view

index table

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
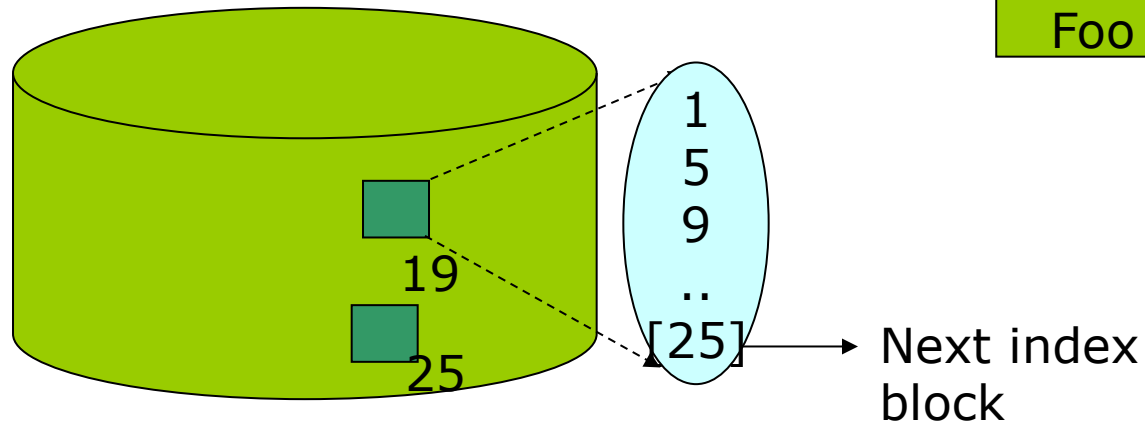
Example of Indexed Allocation

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)
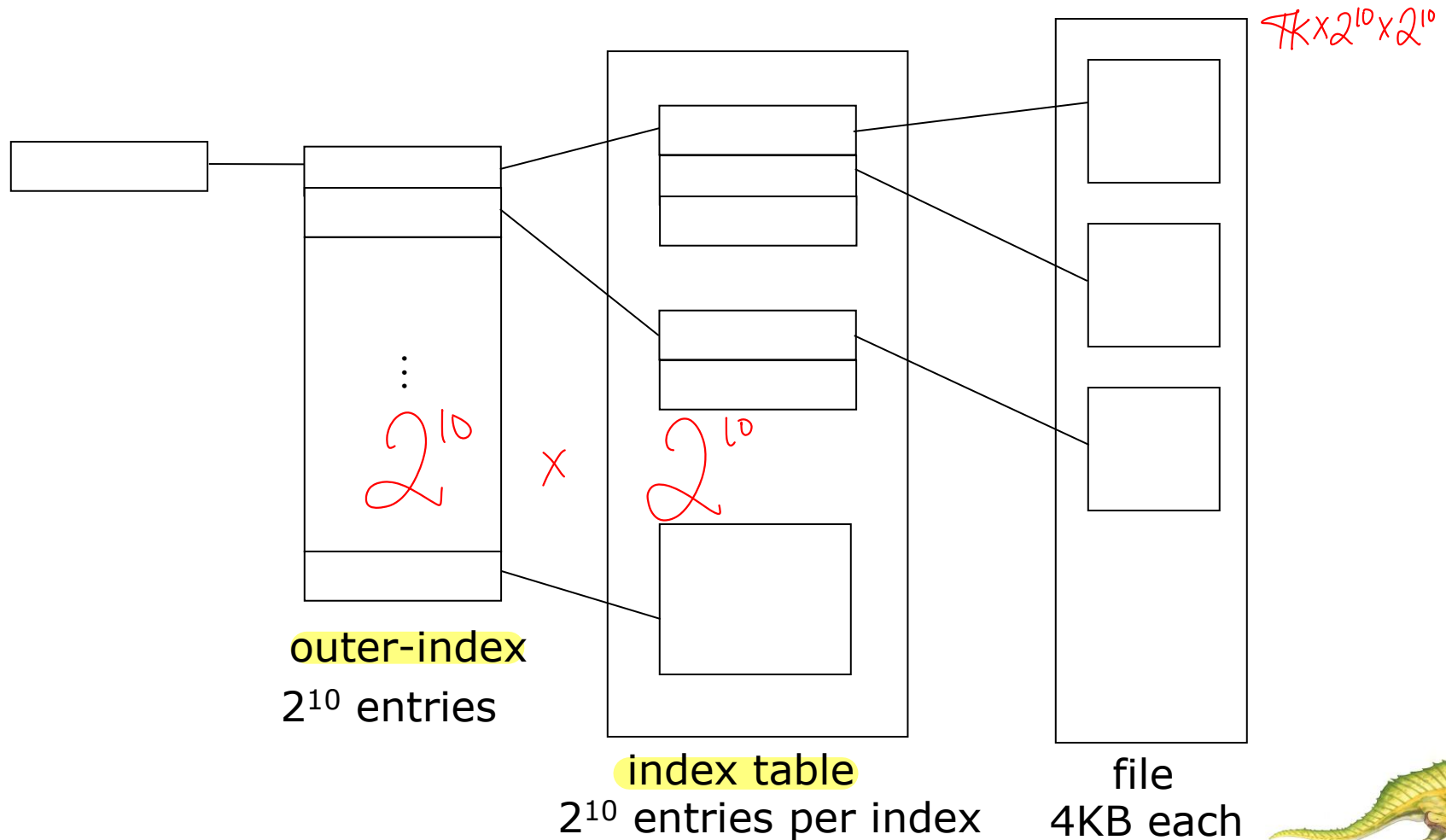
■ : index block

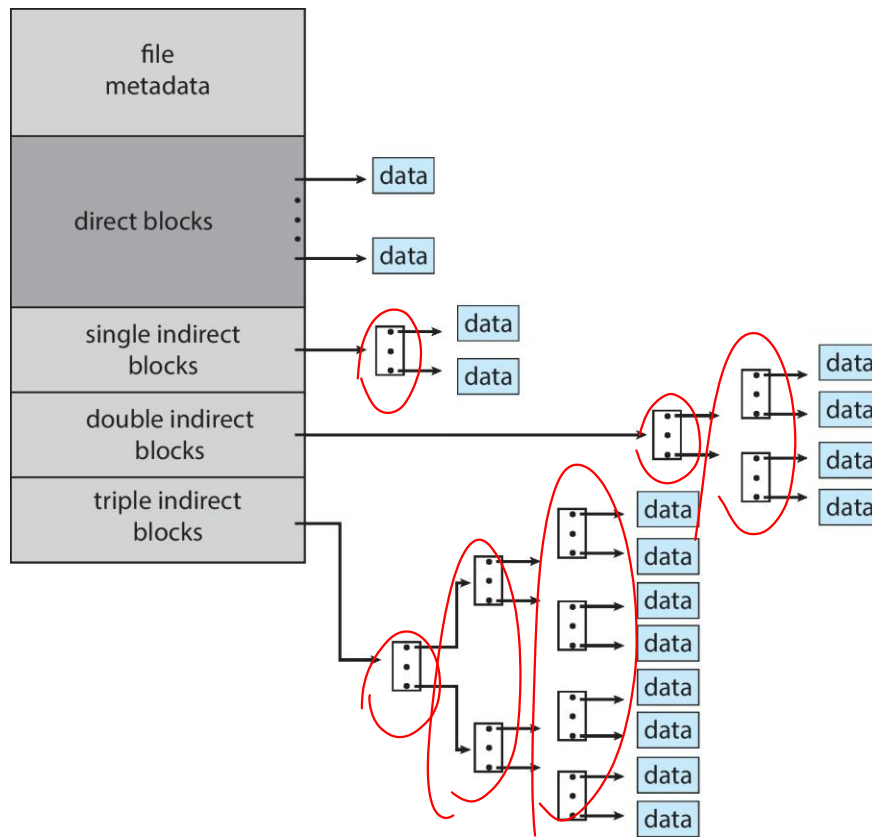| File	Index block
Foo	19

1
5
9
..
[25] → Next index block

19

25

Indexed Allocation – Mapping (Cont.)

- Two-level index (4K blocks could store 1,024 four-byte pointers in outer index → 1,048,567 (=1024*1024) data blocks and file size of up to 4GB)

$4K \times 2^{10} \times 2^{10}$

$2^{10} \times 2^{10}$

outer-index
2^{10} entries

index table
2^{10} entries per index

file
4KB each

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

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)

0   1   2   *n*-1

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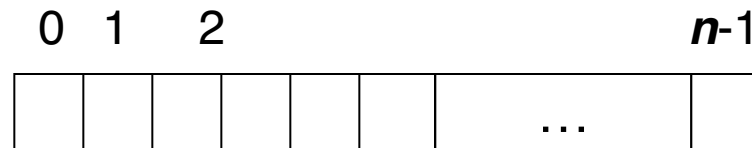
$$\text{bit}[i] = \begin{cases} 1 \Rightarrow \text{block}[i] \text{ free} \\ 0 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

Block number calculation

000…   000..   ….   00….1

(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

- 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)

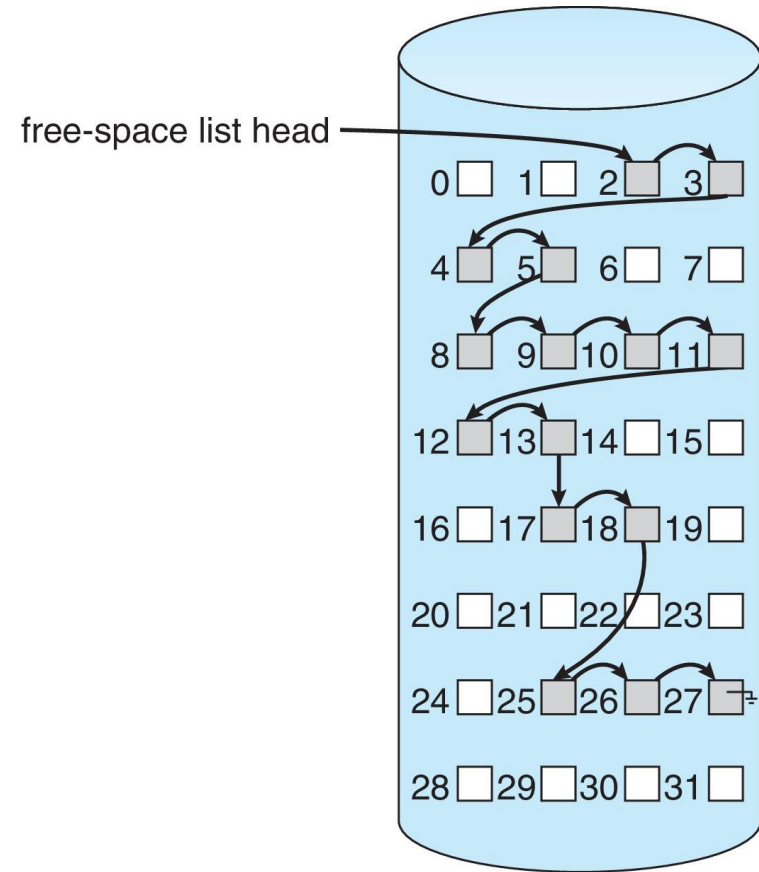   if clusters of 4 blocks -> 8MB of memory

- 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 list head

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, as contiguous-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

Efficiency and Performance

- Efficiency dependent on:

  - Disk allocation and directory algorithms

    ▸ E.g.) inode preallocation

  - Types of data kept in file's directory entry

    ▸ E.g.) Last access time/Last write time

  - Fixed-size or varying-size data structures

    ▸ C.f.) MS FAT file system ⇔ Solaris's ZFS

Efficiency and Performance (Cont.)

- Performance
    - Can increase performance even after algorithms are selected
    - Ways to improve performance
        - Keeping data and metadata close together
        - **Buffer cache** – separate section of main memory for frequently used blocks
        - **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
  - File is accessed as if it is in virtual memory

- Routine I/O through the file system uses the buffer (disk) cache
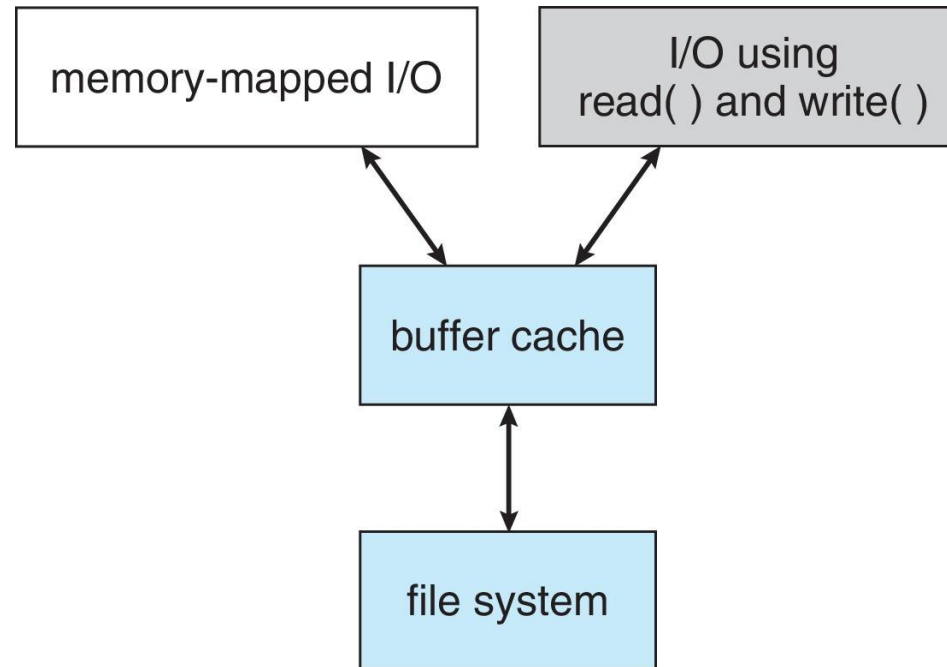
- This leads to the following figure

I/O Without a Unified Buffer Cache

memory-mapped I/O

I/O using
read( ) and write( )

page cache

redundant
cache
⇓
double cache

buffer cache
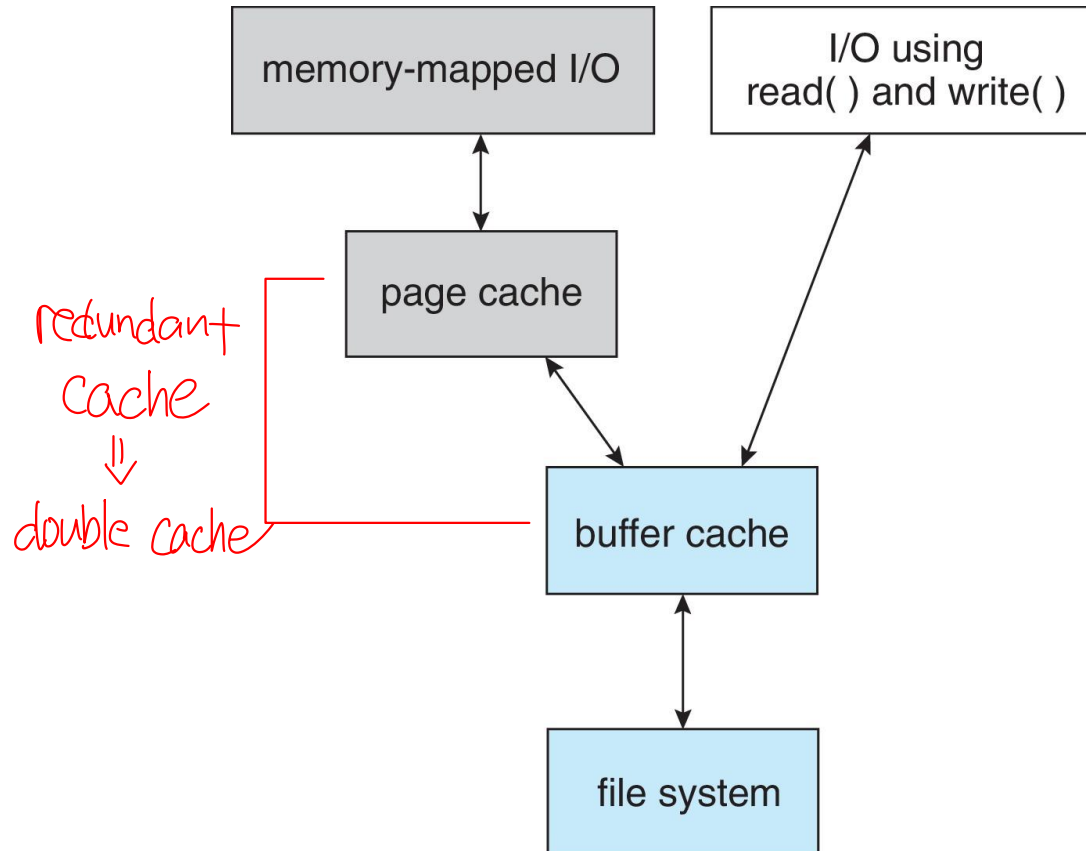
file system

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**

```
┌─────────────────────┐     ┌─────────────────────┐
│  memory-mapped I/O  │     │      I/O using      │
│                     │     │  read( ) and write( )│
└─────────────────────┘     └─────────────────────┘
           ↑                           ↑
            ↘                         ↙
              ┌───────────────────────┐
              │     buffer cache      │
              └───────────────────────┘
                          ↑
                          ↓
              ┌───────────────────────┐
              │     file system       │
              └───────────────────────┘
```

Recovery

- Files and directories are <mark>kept both in main memory and on disk</mark>
    - Care must be taken to ensure a system failure does <mark>not result in loss of data or data inconsistency</mark>

- File system can be crashed for various reasons
    - System crash
    - Bugs in file-system implementation, disk controllers, and user applications
- Deal with methods to recover the file system on crash
    - Consistency checking
   - Log structured file system

Recovery

- **Consistency checking** – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
    - Can be slow and sometimes fails
    - E.g.) Fsck (UNIX)

- 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 transaction (a set of operations)

log

User process → U_1

U_2

U_3

...
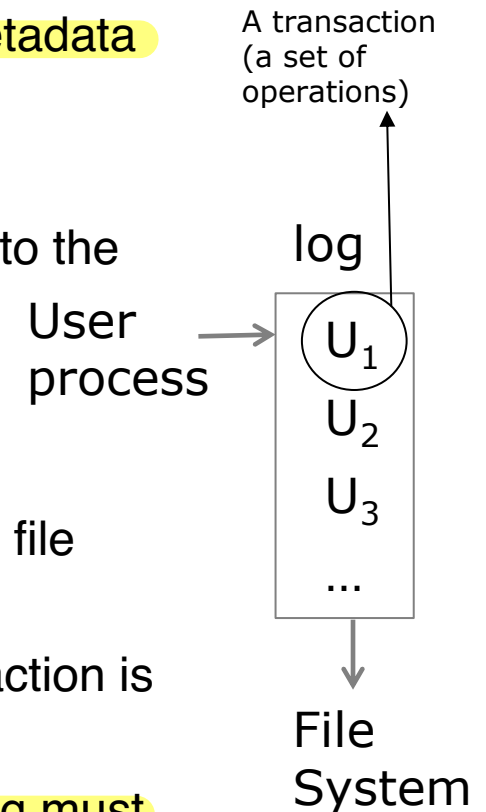
File System

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
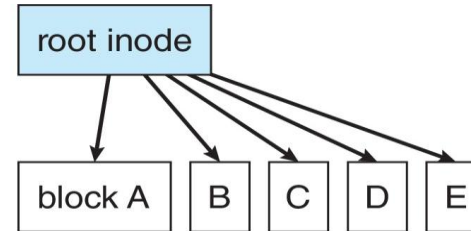
The WAFL File Layout

* Each file system has root inode
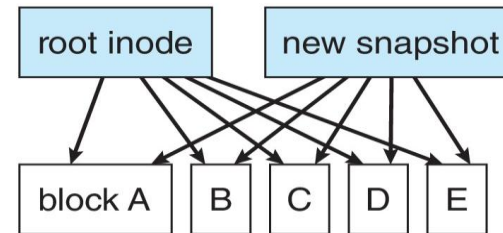* All of metadata lives in files
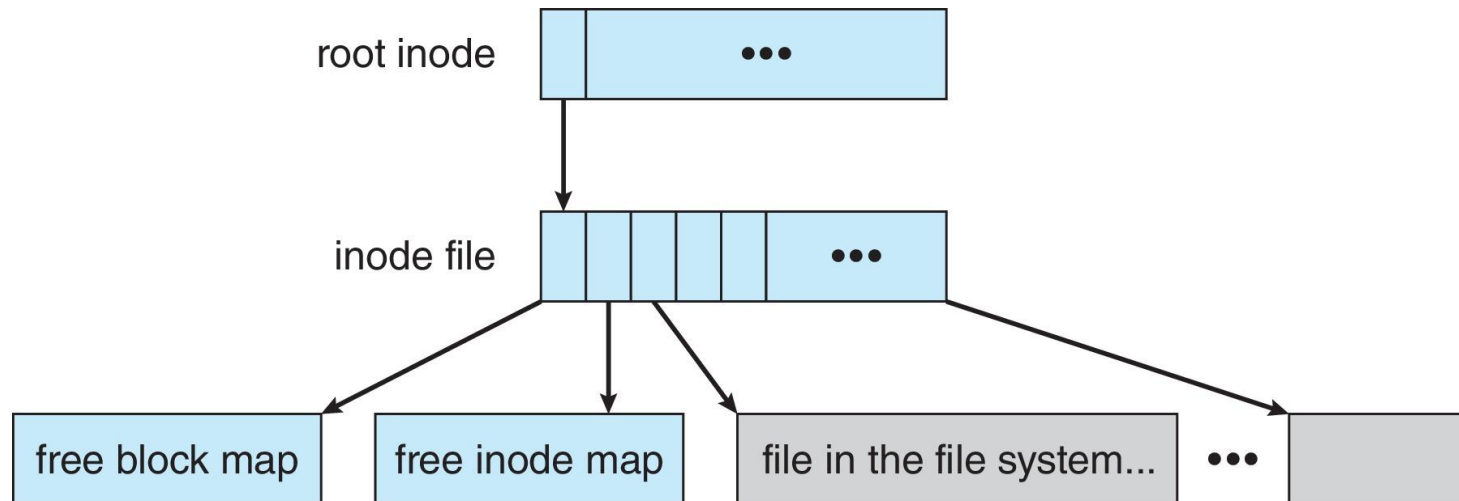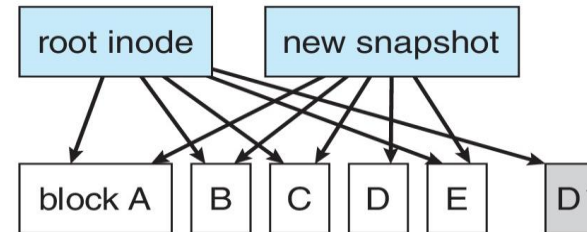
Snapshots in WAFL

Snapshots: useful backups, versioning, testing..

(a) Before a snapshot.

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

(c) After block D has changed to D´.

The Apple File System

In 2017, Apple, Inc., released a new file system to replace its 30-year-old HFS+ file system. HFS+ had been stretched to add many new features, but as usual, this process added complexity, along with lines of code, and made adding more features more difficult. Starting from scratch on a blank page allows a design to start with current technologies and methodologies and provide the exact set of features needed.

Apple File System (APFS) is a good example of such a design. Its goal is to run on all current Apple devices, from the Apple Watch through the iPhone to the Mac computers. Creating a file system that works in watchOS, I/Os, tvOS, and macOS is certainly a challenge. APFS is feature-rich, including 64-bit pointers, clones for files and directories, snapshots, space sharing, fast directory sizing, atomic safe-save primitives, copy-on-write design, encryption (single- and multi-key), and I/O coalescing. It understands NVM as well as HDD storage.

Most of these features we've discussed, but there are a few new concepts worth exploring. Space sharing is a ZFS-like feature in which storage is available as one or more large free spaces (containers) from which file systems can draw allocations (allowing APFS-formatted volumes to grow and shrink). Fast directory sizing provides quick used-space calculation and updating. Atomic safe-save is a primitive (available via API, not via file-system commands) that performs renames of files, bundles of files, and directories as single atomic operations. I/O coalescing is an optimization for NVM devices in which several small writes are gathered together into a large write to optimize write performance.

Apple chose not to implement RAID as part of the new APFS, instead depending on the existing Apple RAID volume mechanism for software RAID. APFS is also compatible with HFS+, allowing easy conversion for existing deployments.

End of Chapter 14
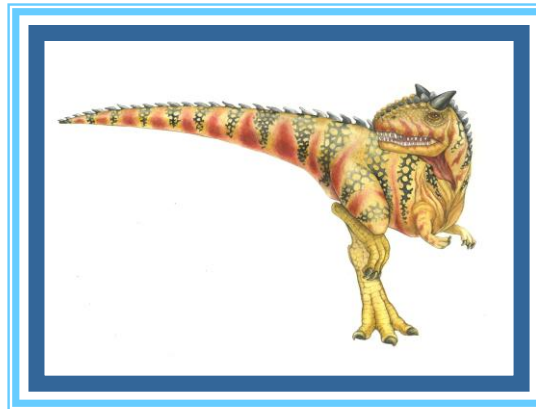
Free-Space Management (Cont.)

- 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

TRIMing Unused Blocks

- HDDS overwrite in place so need only free list

- Blocks not treated specially when freed
  - Keeps its data but without any file pointers to it, until overwritten

- Storage devices not allowing overwrite (like NVM) suffer badly with same algorithm
  - Must be erased before written, erases made in large chunks (blocks, composed of pages) and are slow
  - TRIM is a newer mechanism for the file system to inform the NVM storage device that a page is free
    - Can be garbage collected or if block is free, now block can be erased