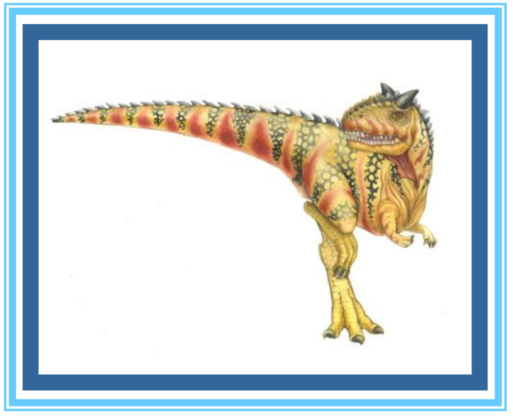
# Chapter 14:  File System Implementation



# Outline

- 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)
  - Provides 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

↓

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 (Ch.12)
- **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
- Logical layers can be implemented by any coding method according to OS designer

# File System Layers (Cont.)

- Many file systems, sometimes many are supported within an OS
  - 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
- On-disk structures
  - Boot control block (per volume)
  - Volume control block (per volume)
  - Directory structure (per file system)
  - Per-file File-control block (FCB)

# On-disk Structures

- **Boot control block**: info needed to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
  - UNIX boot block, NTFS partition boot sector
- **Volume control block** (**superblock, master file table**) contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
  - UNIX superblock, NTFS master file table
- Directory structure organizes the files
  - UNIX filenames and inode numbers, NTFS master file table
- Per-file **File Control Block** (**FCB**) contains many details about the file
  - typically inode number, permissions, size, dates
  - NTFS stores info in master file table using relational DB structures

# File-System Implementation (Cont.)

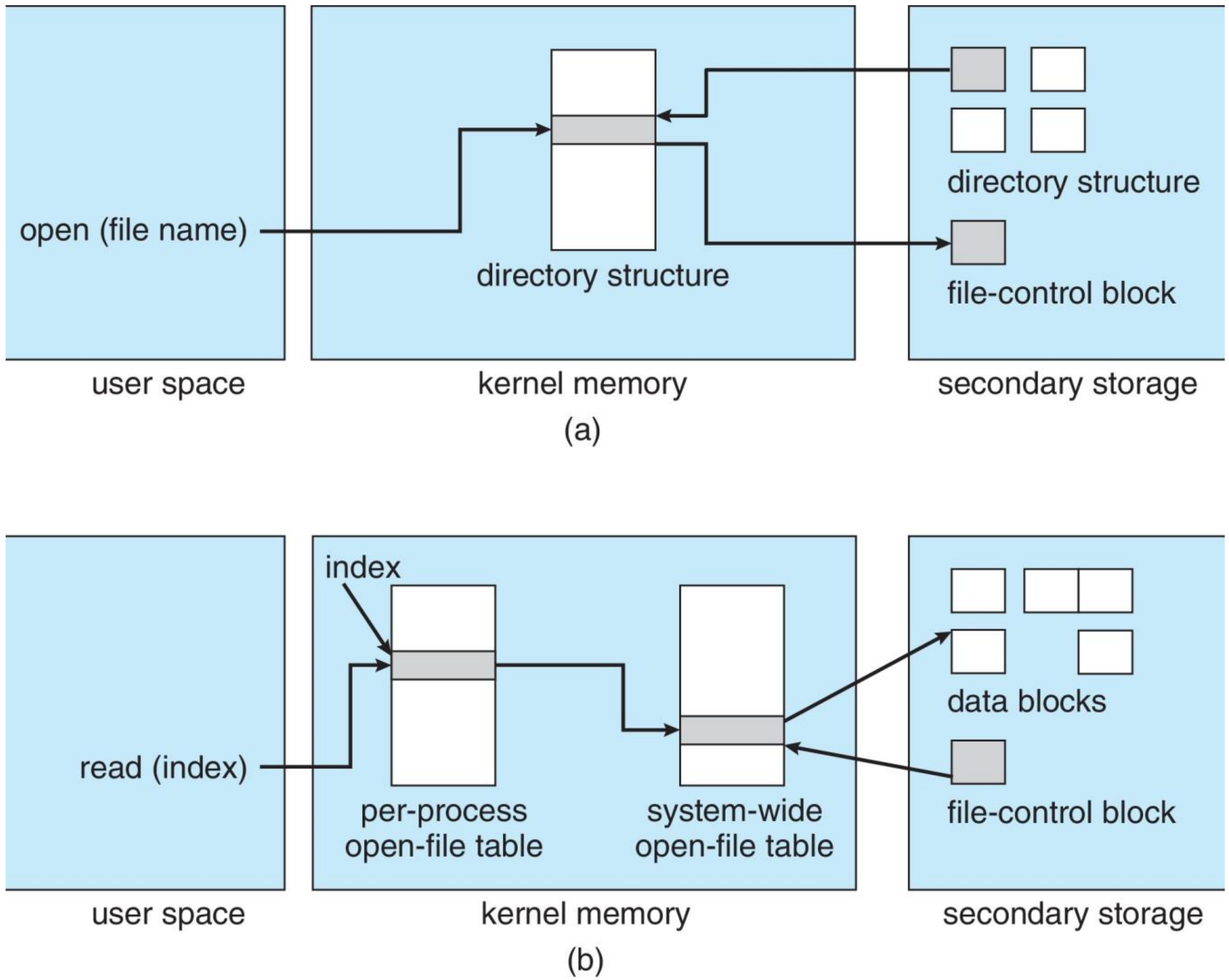
- A typical file-control block

| |
|---|
| 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
- **per-process open-file table** contains pointers to appropriate entries in system-wide open-file table as well as other info
- Directory-structure cache holds directory information of recently accessed directories
- Plus buffers hold data blocks from secondary storage

# In-Memory File System Structures



# 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

# 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
  - 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

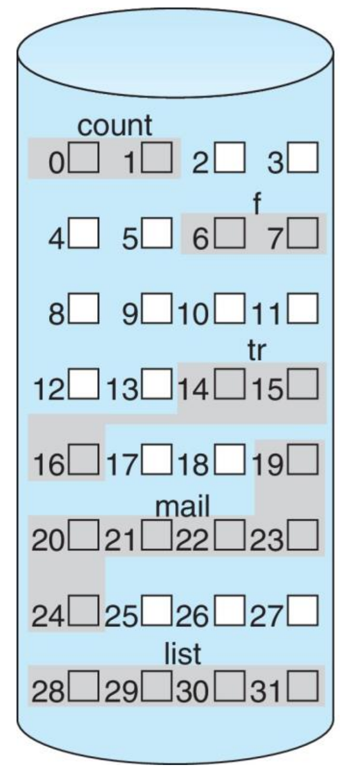
- Mapping from logical to physical

LA/512

Q

R

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

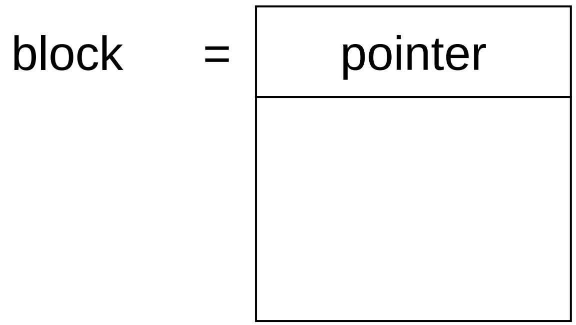
- An **extent** is contiguous blocks on disk
  - Extents are allocated for file allocation
  - A file consists of one or more extents

# Allocation Methods - Linked

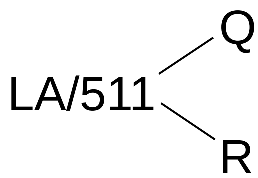
- **Linked allocation** – each file a linked list of blocks
  - File ends at nil pointer
  - Each block contains pointer to next block
  - Free space management system called when new block needed
- Pros and cons
  - No compaction, external fragmentation
  - Reliability can be a problem
  - Locating a block can take many I/Os and disk seeks
    - Improve efficiency by clustering blocks into groups but increases internal fragmentation

# Linked Allocation

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



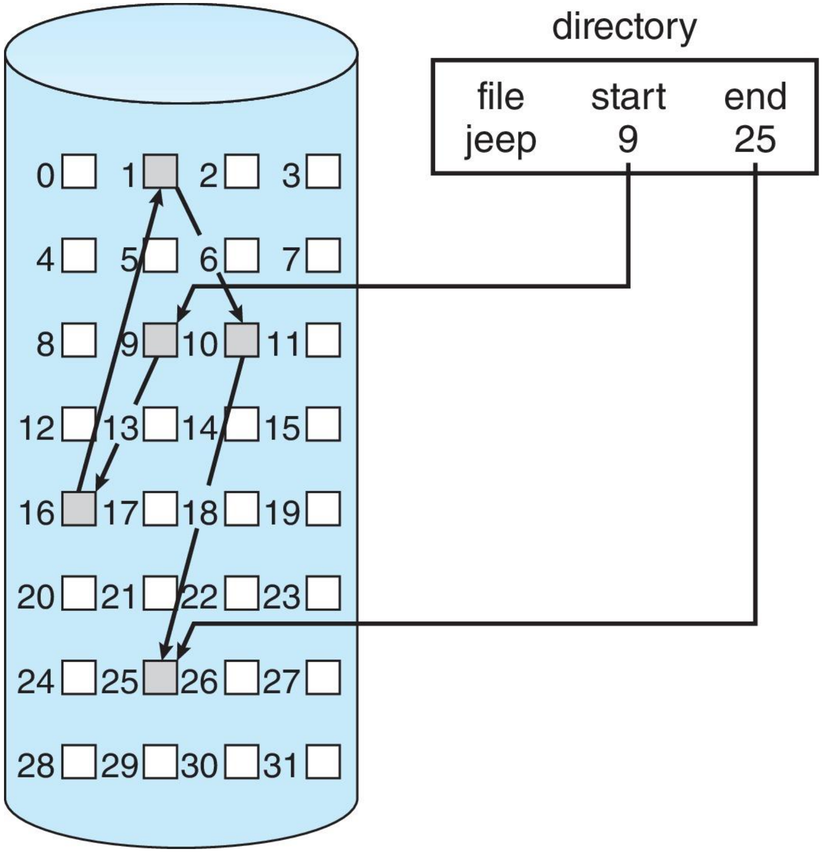

- Mapping



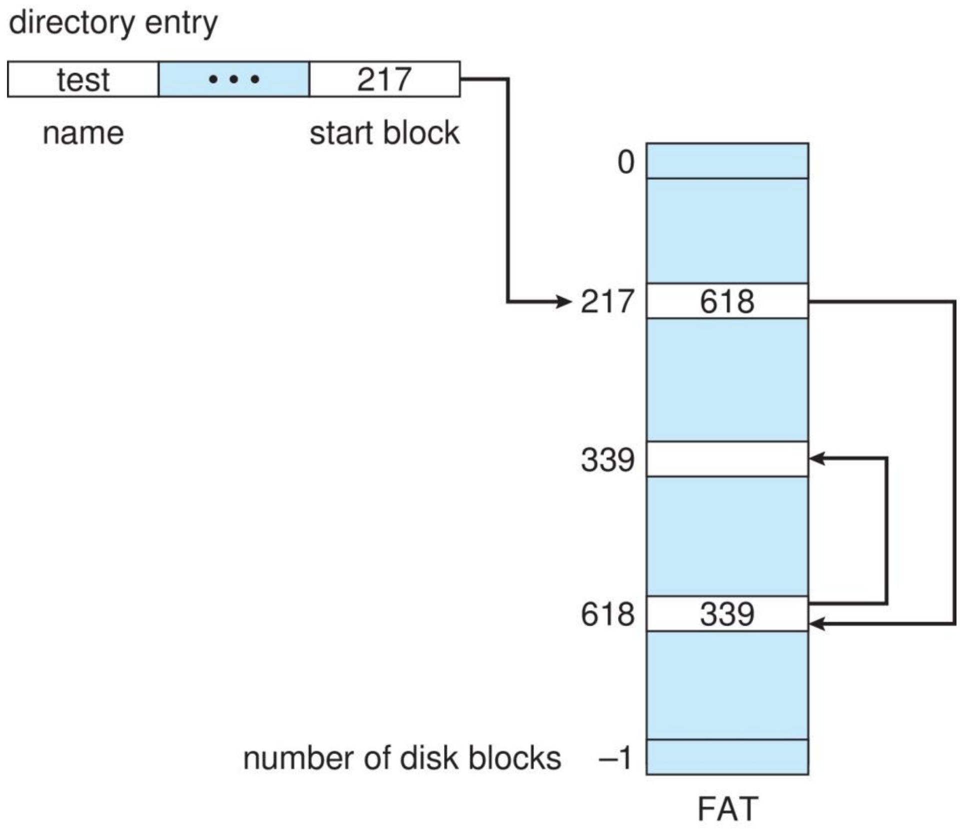

Block to be accessed is the Qth block in the linked chain of blocks representing the file

Displacement into block = R + 1

# Linked Allocation



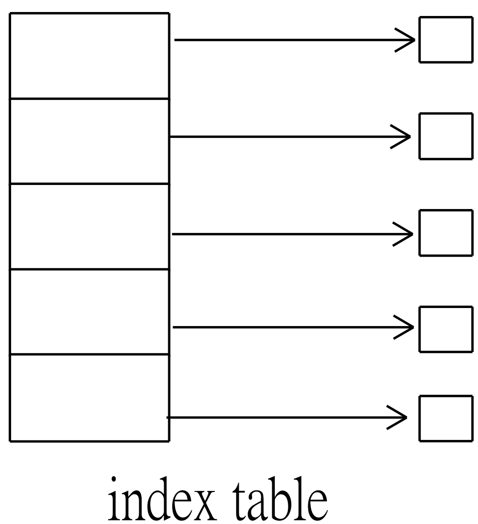
# File-Allocation Table



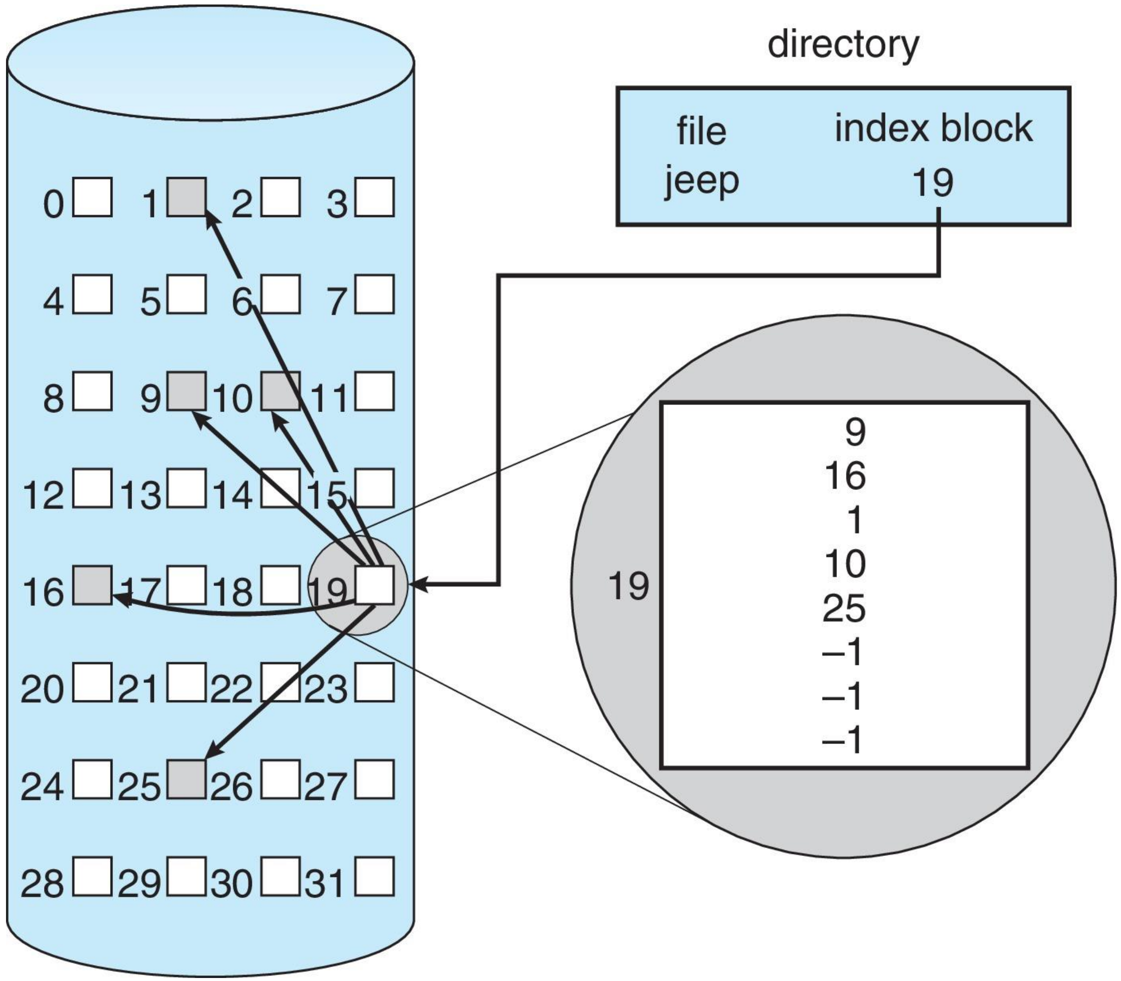
# Allocation Methods - Indexed

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

- Logical view

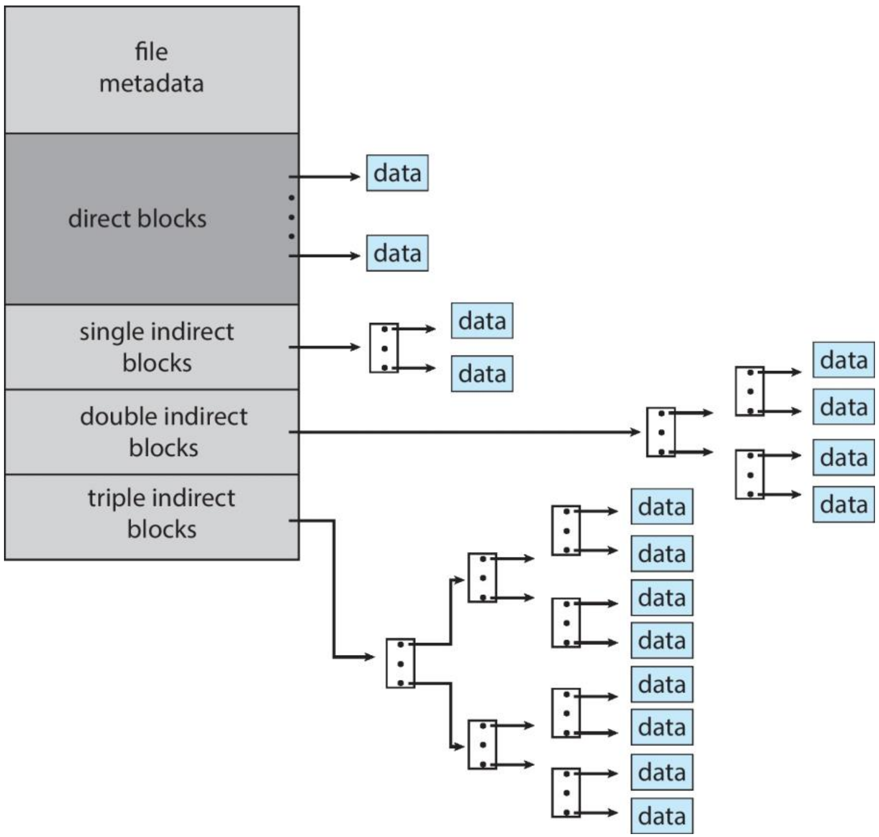


# Example of Indexed Allocation



# Combined Scheme: UNIX UFS

4K bytes per block, 32-bit addresses



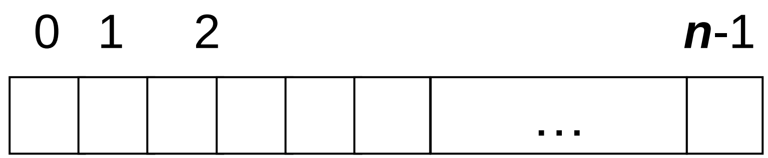

More index blocks than can be addressed with 32-bit file pointer

# 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
- For NVM, no disk head so different algorithms and optimizations needed
  - Using old algorithm takes many CPU cycles trying to avoid non-existent head movement
  - With NVM goal is to reduce CPU cycles and overall path needed for I/O

# 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)




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

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)

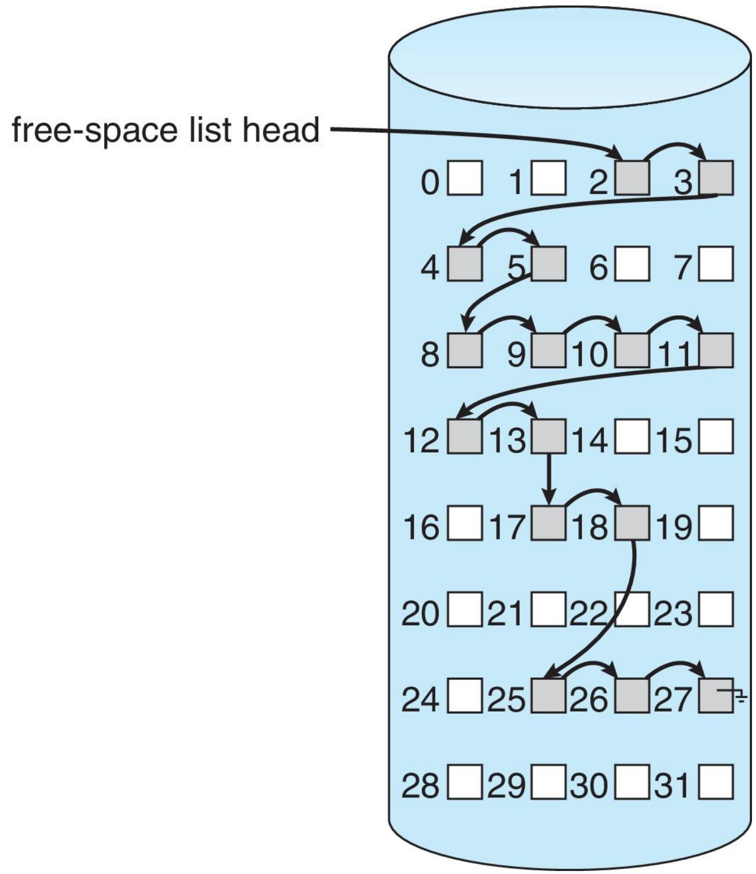
    $\boldsymbol{n} = 2^{40}/2^{12} = 2^{28}$ bits (or 32MB)

    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

# 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

# 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

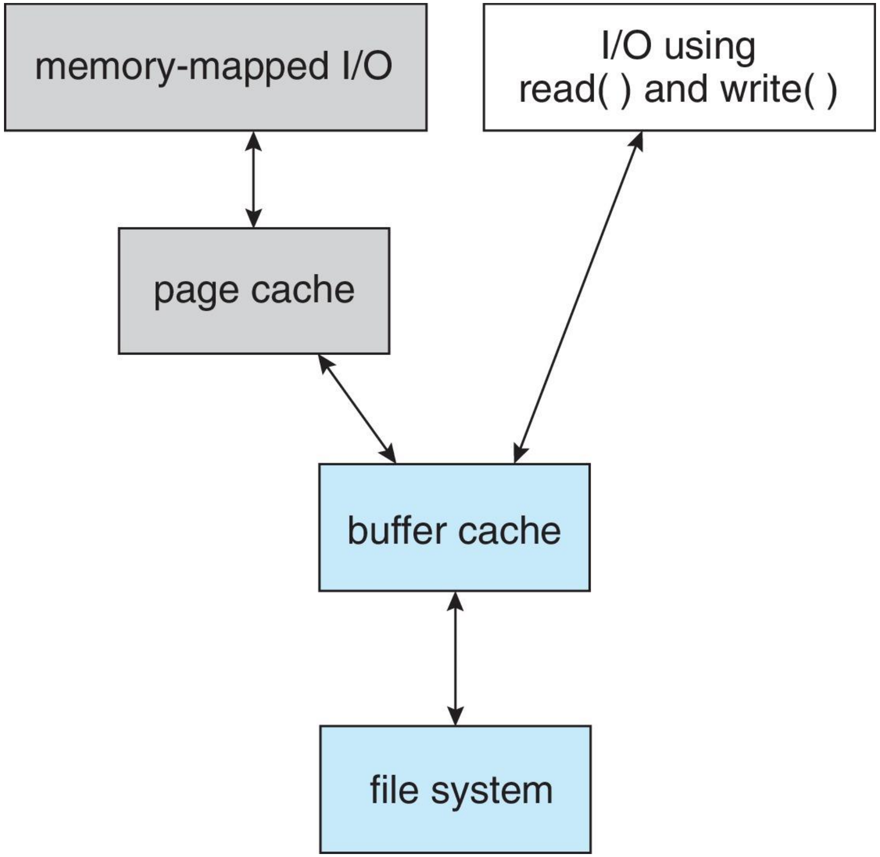
# Efficiency and Performance (Cont.)

- 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

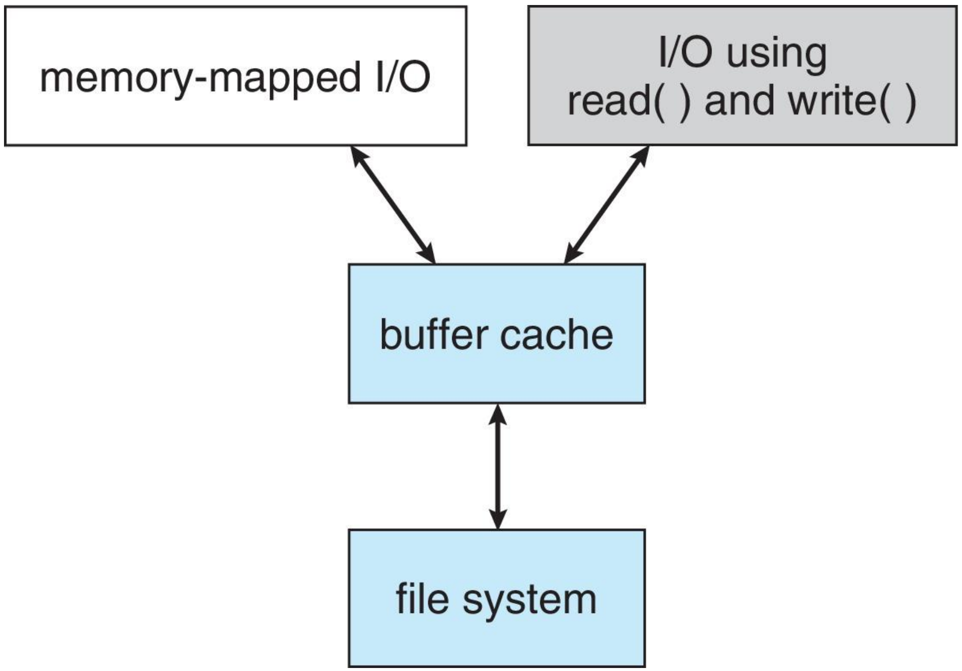
# I/O Without a Unified Buffer Cache



# 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



# 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

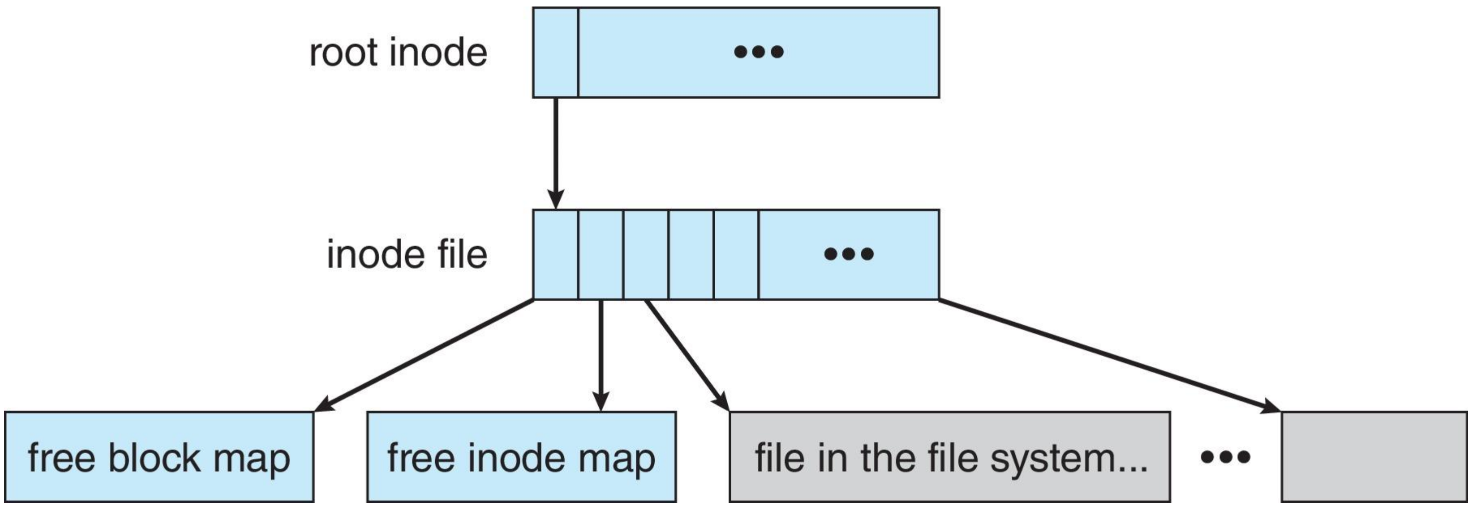
# Log Structured File Systems

- **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

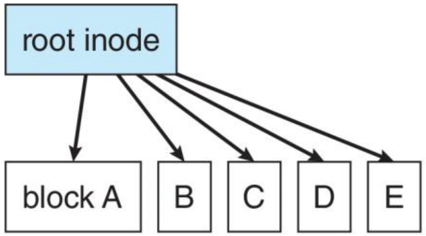
# Example: WAFL File System

- Used on Network Appliance (NetApp) "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

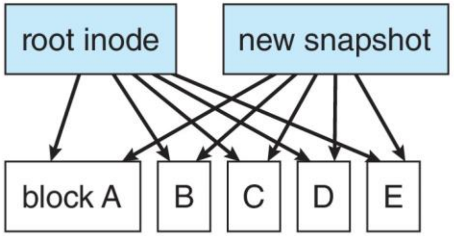
# The WAFL File Layout



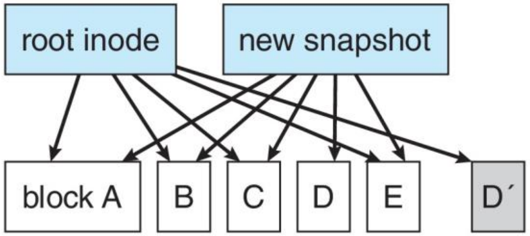
# Snapshots in WAFL




(a) Before a snapshot.




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




(c) After block D has changed to D´.

# The Apple File System

- Apple released a new file system in 2017 called APFS to replace its 30-year-old HFS+
- The goal is to run on all current Apple devices
  - From Apple Watch through the iphone to the Mac computers
  - watchOS, iOS, tvOS, macOS
- Features include
  - 64-bit pointers, clones for files and directories, snapshots, copy-on-write design, encryption
  - Space sharing: storage is available as one or more large free spaces (containers) from which file systems can draw allocations
  - Fast directory sizing: provides quick used space calculation and updating
  - Atomic safe-save primitives: perform renames of files, bundles of files, and directories as single atomic operations

# End of Chapter 14