

Chapter 12: File-System Implementation



Chapter 12: Outline

- File-System Structure
- File-System Operations
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery
- Example: WAFL File System (Write-Anywhere File Layout)



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 (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

■ *Disk* provides in-place rewrite and random access

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

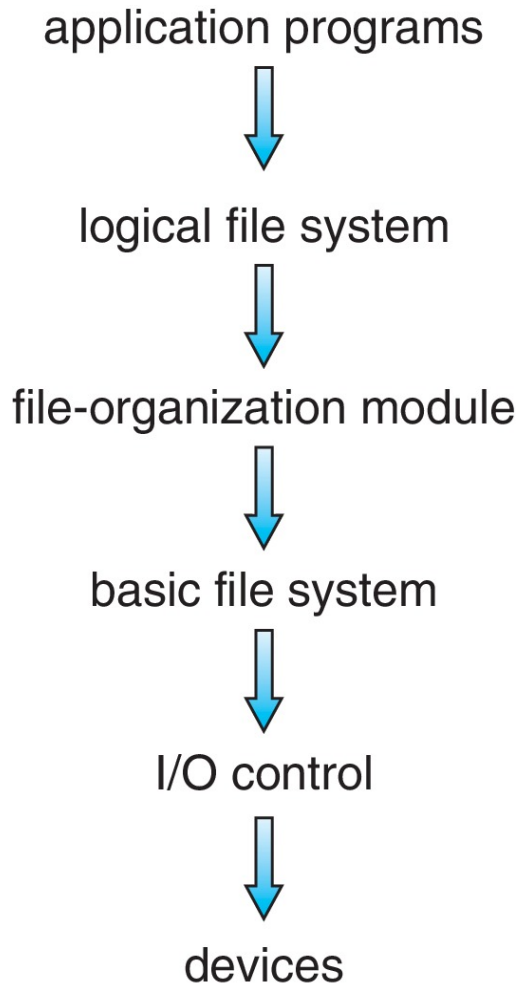
■ *File control block (FCB)* – storage structure consisting of information about a file

■ *Device driver* controls the physical device

■ File system is organized into *layers*



Layered File System



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

File System Layers

- *Device drivers* manage I/O devices at the I/O control layer
 - E.g., 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



File System Layers (Cont.)

- *Logical file system* manages metadata information
 - Translates file name into file number, file handle, location by maintaining file control blocks (i.e., *inodes* in UNIX)
 - Directory management
 - Protection
 - Logical layers can be implemented by any coding method according to OS designer
- Many file systems exist 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 NFS, etc.)
 - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE, Lustre

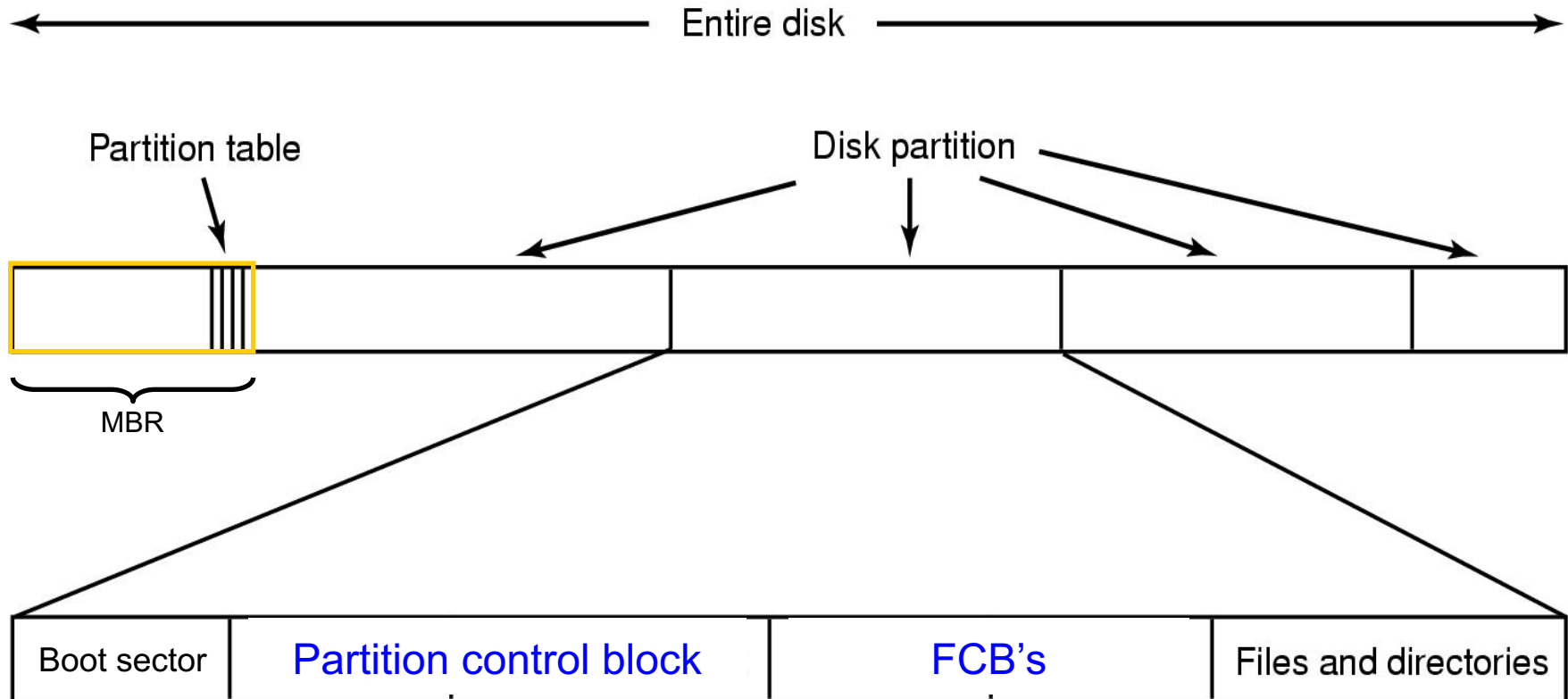


File-System Implementation

- 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* (e.g., 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.)



File-System Implementation (Cont.)

- Per-file *File Control Block* (**FCB**) contains many details about the file
 - typically inode number, permissions, size, dates
 - NTFS stores into in *master file table* using relational DB structures

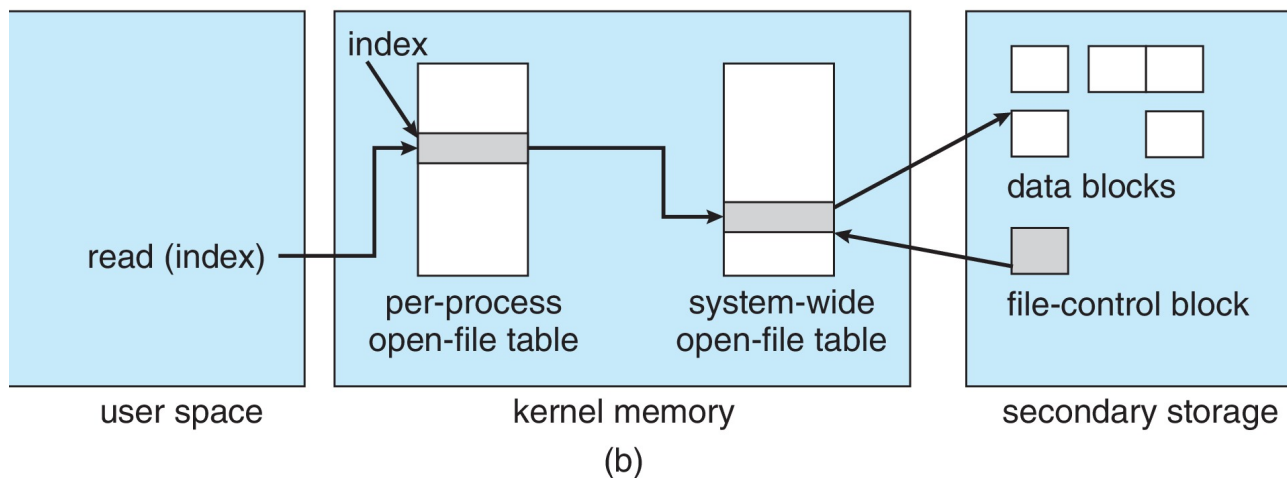
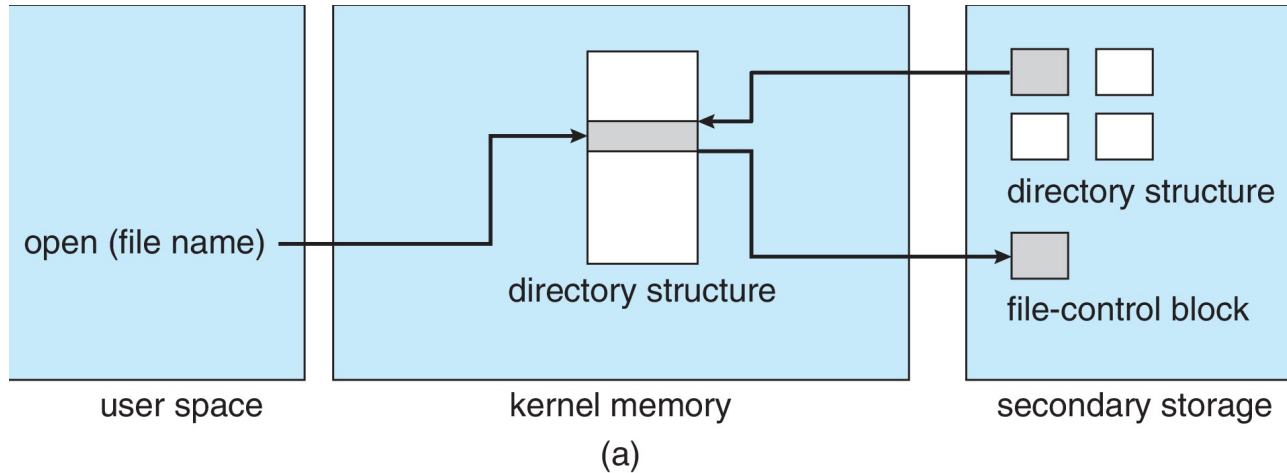
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

In-Memory File System Structures

- *Mount table* storing file system mounts, mount points, file system types
- *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
- The following figure illustrates the necessary file system structures provided by the operating systems
 - Figure (a) refers to opening a file
 - Figure (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 copied to specified user process memory address



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

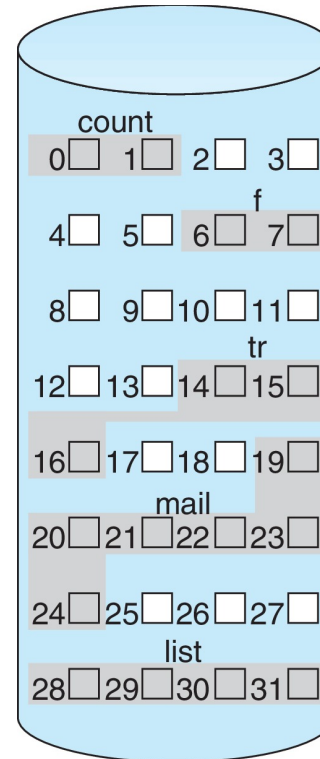
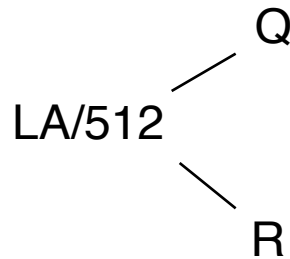
Allocation Methods – Contiguous

- 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



directory

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

- Block to be accessed = $Q + \text{starting address}$
- Displacement into block = R

Extent-Based Systems

- Many newer file systems (e.g., *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

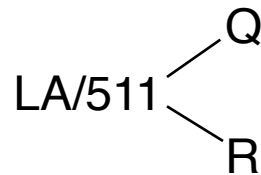
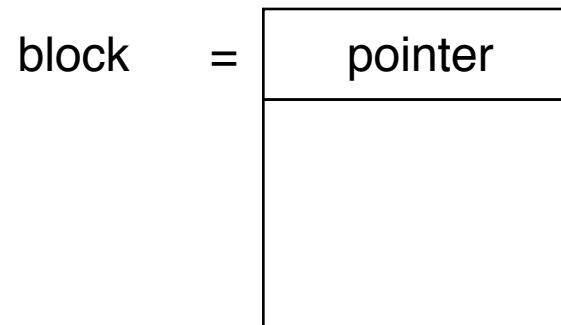
Allocation Methods - Linked

- *Linked allocation* – each file is 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
- *File Allocation Table (FAT)* 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

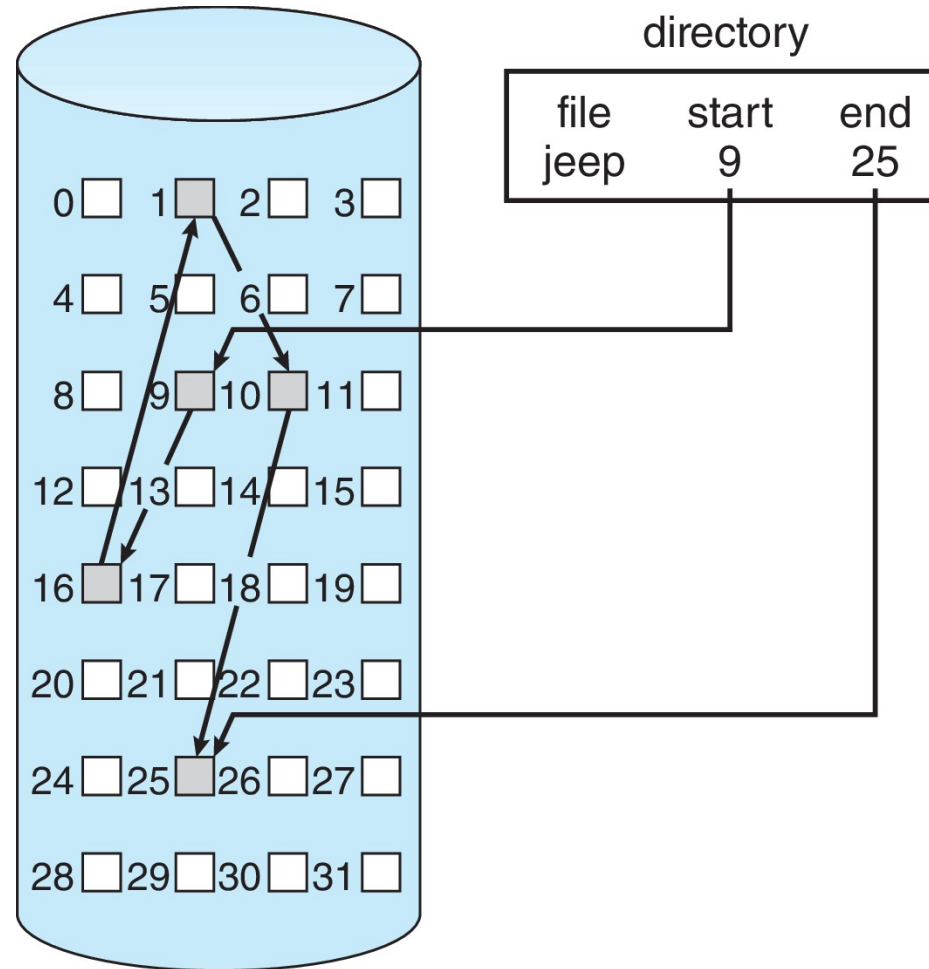
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk



- Mapping

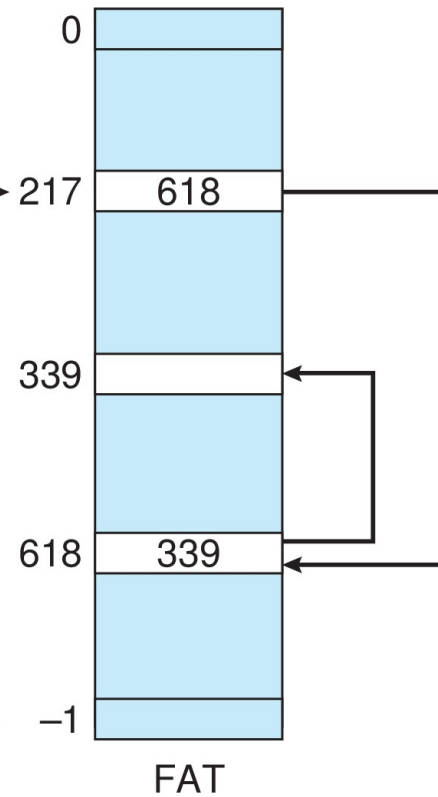
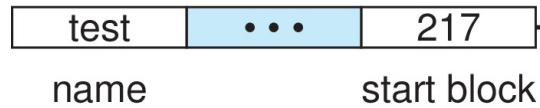
- Block to be accessed is the Q^{th} block in the linked chain of blocks representing the file
- Displacement into block = $R + 1$

Linked Allocation



File-Allocation Table (FAT)

directory entry



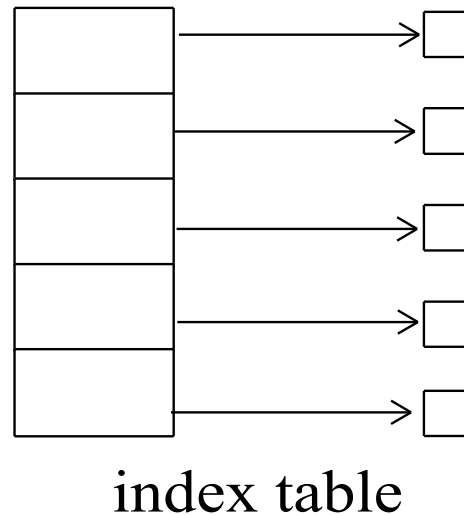
number of disk blocks

Allocation Methods – Indexed

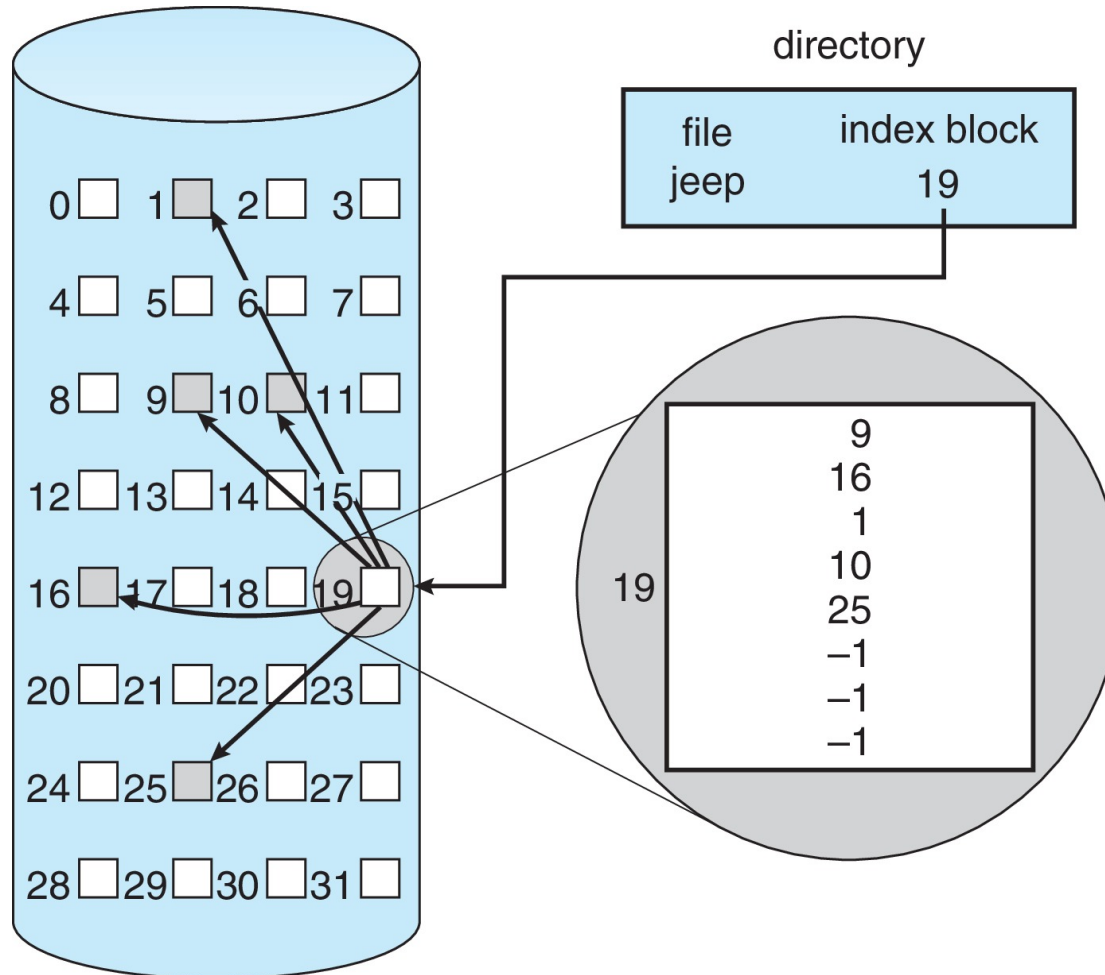
■ Indexed allocation

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

■ Logical view

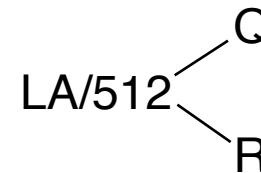


Example of Indexed Allocation



Indexed Allocation (Cont.)

- Need *index table*
- *Random access*
- *Dynamic access* without external fragmentation, but have overhead of index block
 - E.g., 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.)

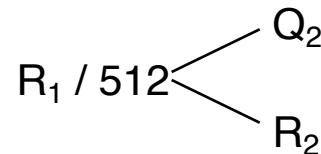
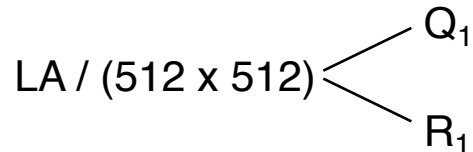
- Mapping from logical to physical in a file of unbounded length (block size of 512 words)
- Linked scheme – Link blocks of index table (no limit on size)
 - Q_1 = block of index table
 - R_1 is used as follows:
 - ▶ Q_2 = displacement into block of index table
 - ▶ R_2 = displacement into block of file:

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

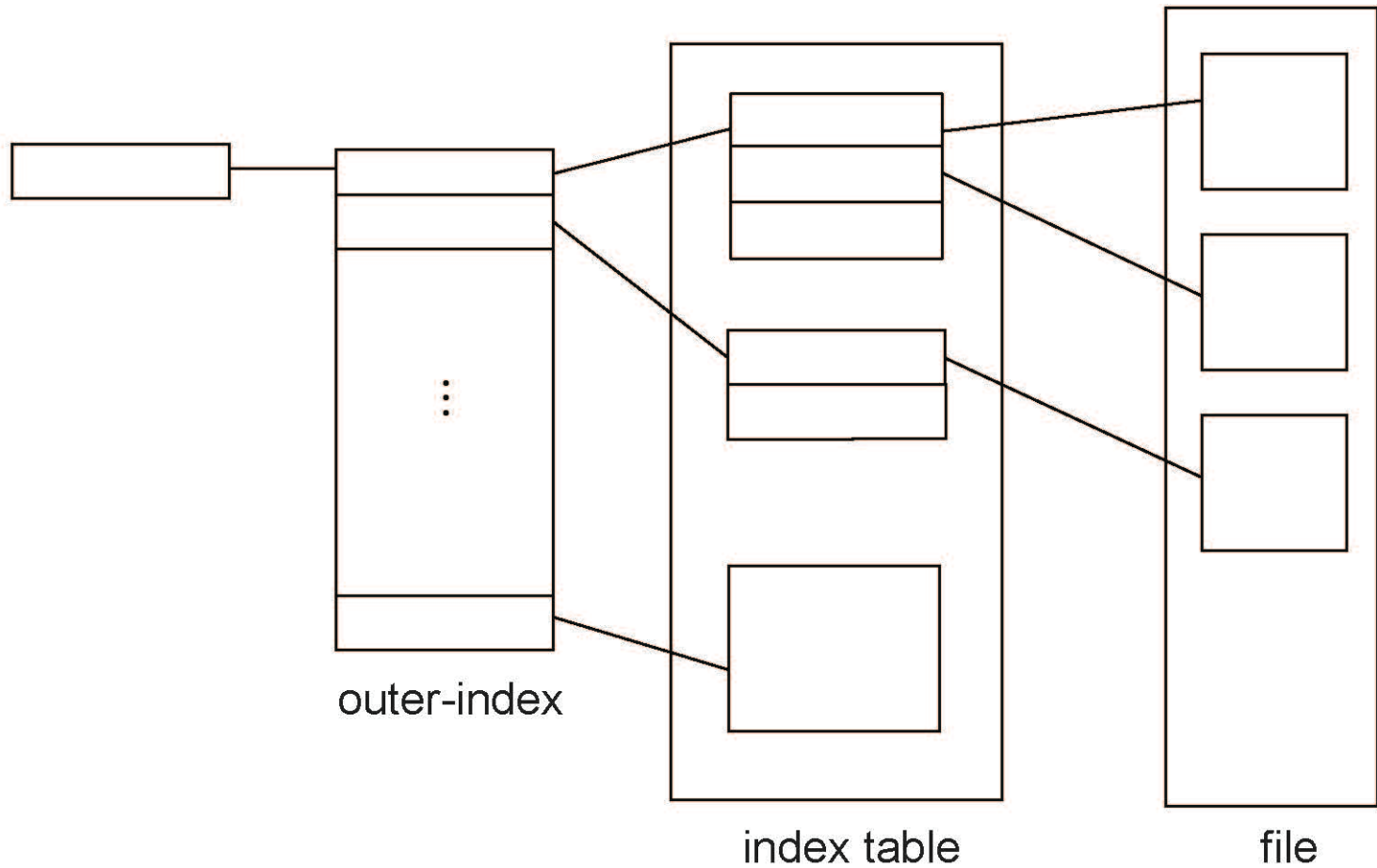
$$R_1 / 512 \begin{cases} Q_2 \\ R_2 \end{cases}$$

Indexed Allocation – Mapping (Cont.)

- E.g., Two-level index (4K blocks could store 1,024 four-byte pointers in outer index \rightarrow 1,048,567 data blocks and file size of up to 4GB)
 - Q_1 = displacement into outer-index
 - R_1 is used as follows:
 - ▶ Q_2 = displacement into block of index table
 - ▶ R_2 displacement into block of file:



Indexed Allocation – Mapping (Cont.)



Combined Scheme: UNIX UFS



Suppose that on a computer, the OS uses i-nodes

and each disk block is 4KB.

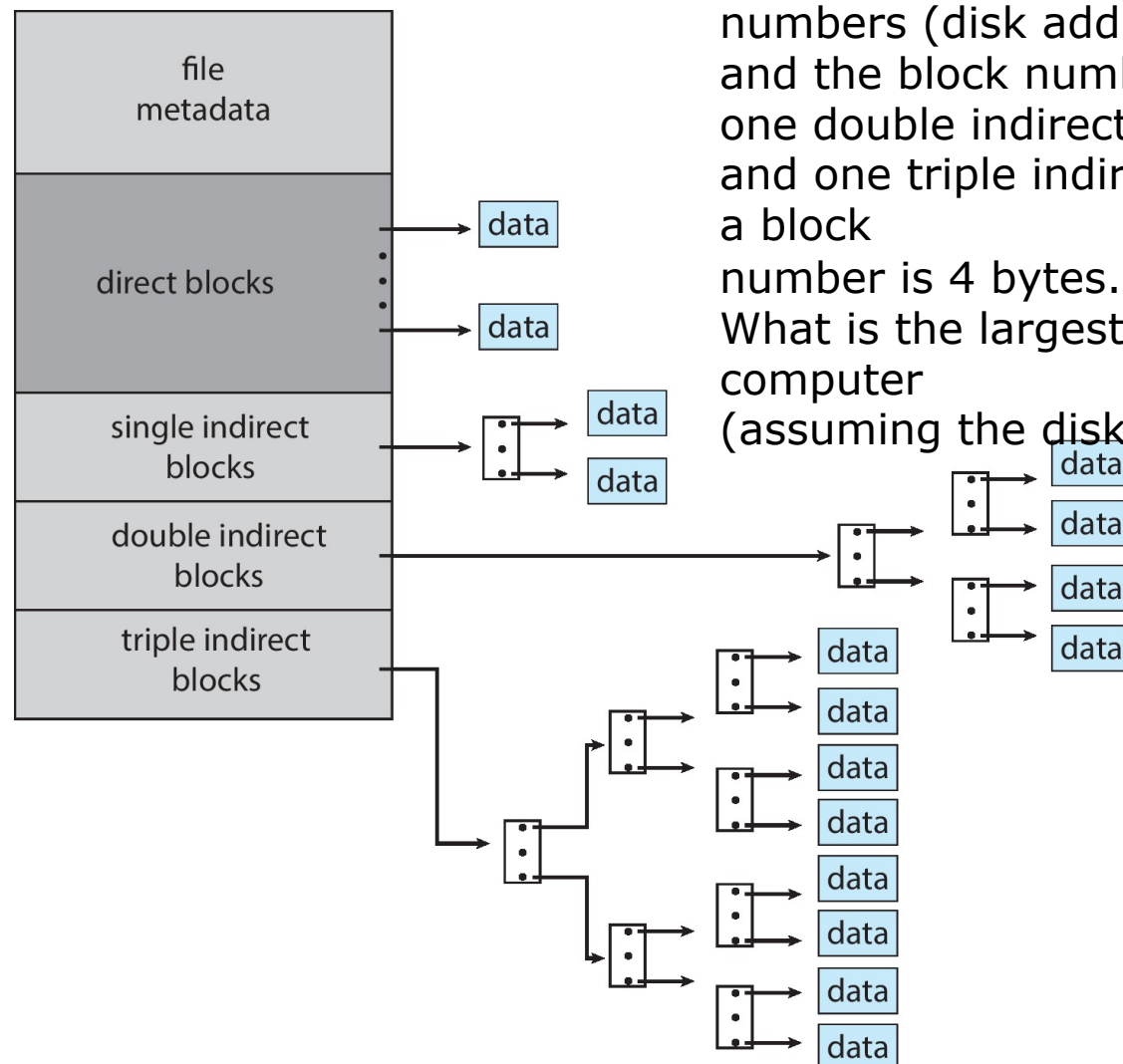
Assume that an i-node contains 12 direct block numbers (disk addresses)

and the block numbers for one indirect block, one double indirect block, and one triple indirect block. Assume also that a block

number is 4 bytes.

What is the largest possible file on that computer

(assuming the disk is large enough).



■ 4K bytes per block, 32-bit addresses

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



- *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
 - An old algorithm uses many CPU cycles for trying to avoid non-existent head movement
 - With NVM goal is to reduce CPU cycles and overall path needed for I/O

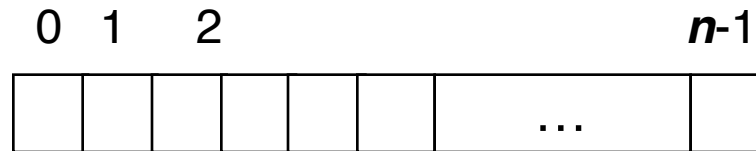
Performance (Cont.)

- 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
 - Typical disk drive at 250 I/Os per second
 - ▶ $159,000 \text{ MIPS} / 250 = 630 \text{ million instructions during one disk I/O}$
 - Fast SSD drives provide 60,000 IOPS
 - ▶ $159,000 \text{ MIPS} / 60,000 = 2.65 \text{ millions instructions during one disk I/O}$



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)



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

- 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 = 212 bytes

disk size = 240 bytes (1 terabyte)

$n = 240/212 = 228$ bits (or 32MB)

if clusters of 4 blocks -> 8MB of memory

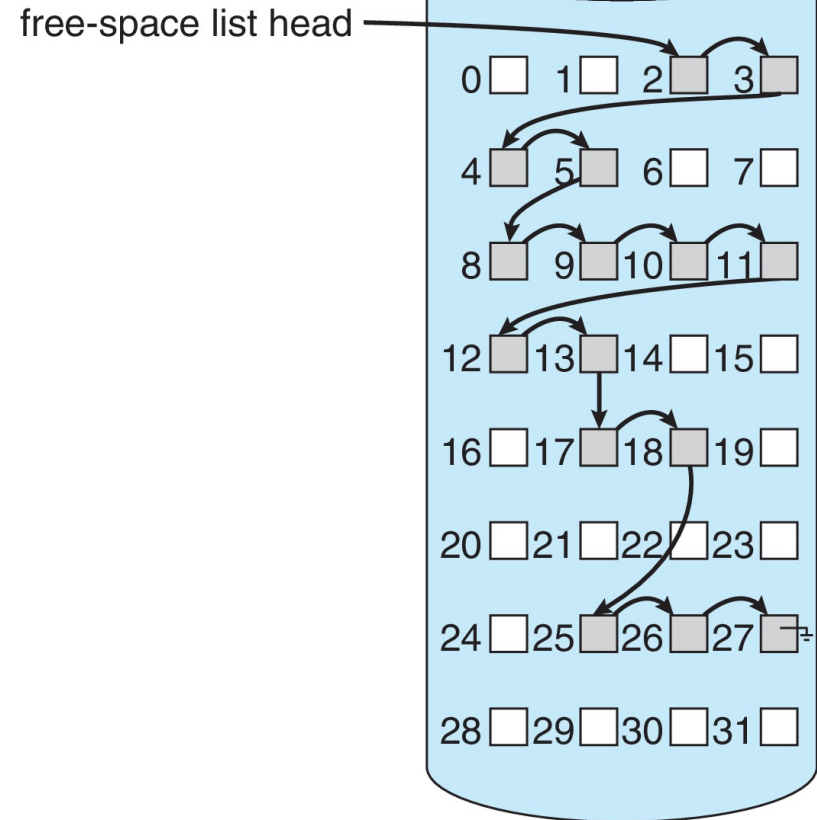
- Easy to get contiguous files



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



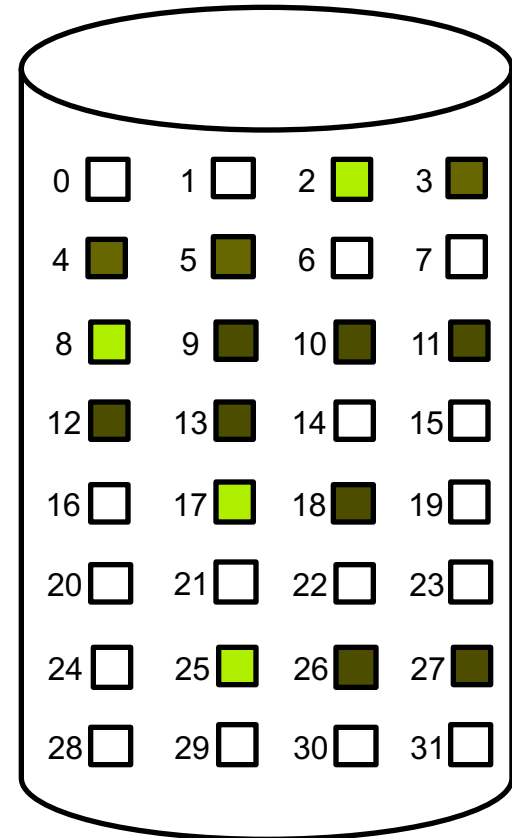
Free-Space Management (Cont.)

■ 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)

Grouping, $n = 3$

Block 2 stores 3, 4, 5
 Block 5 stores 8, 9, 10
 Block 10 stores 11, 12, 13
 Block 13 stores 17, 28, 25
 Block 25 stores 26, 27



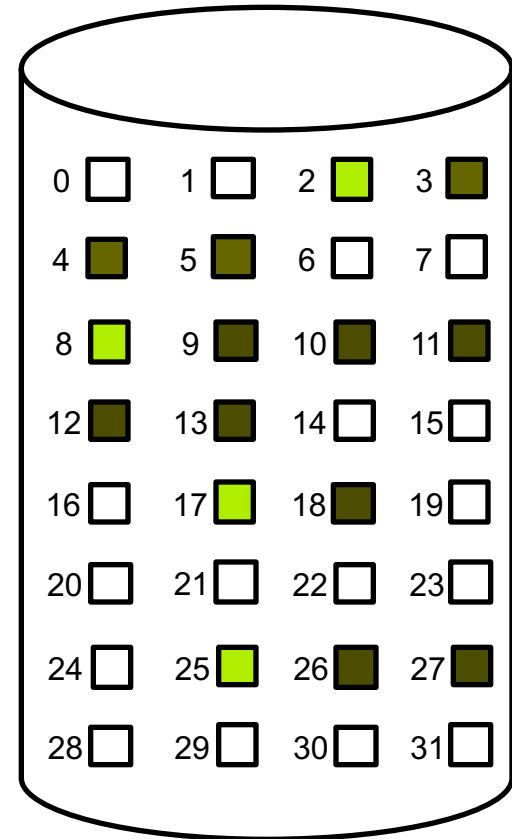
Free-Space Management (Cont.)

■ 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

■ start count

2	4
8	6
17	2
25	3



■ 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 (e.g., **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

■ *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

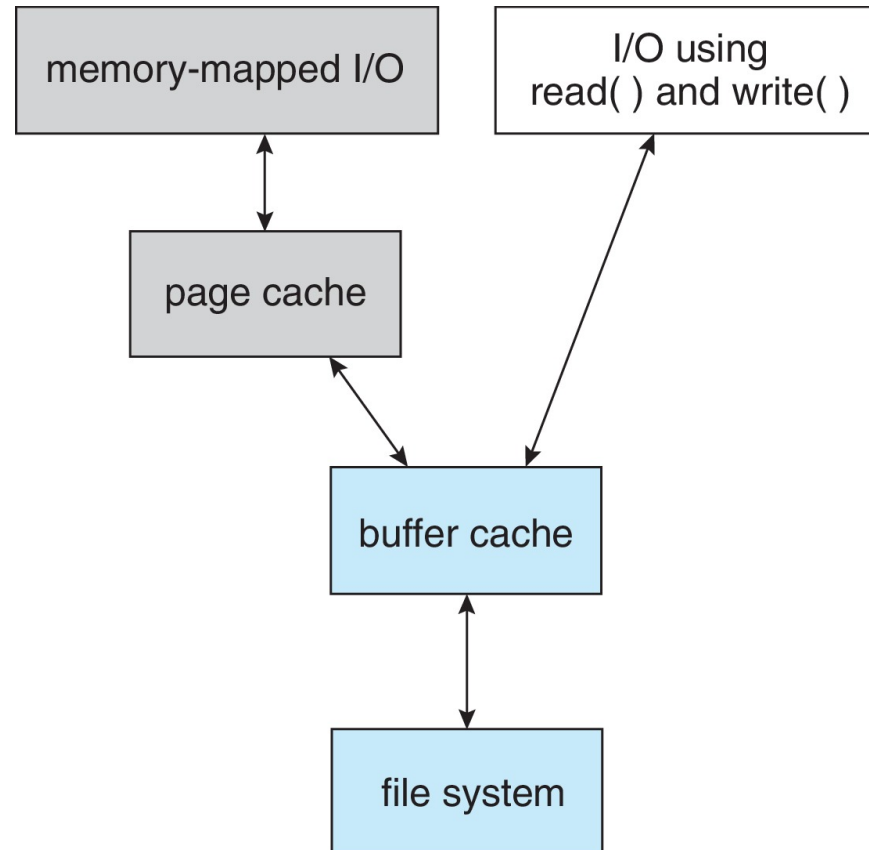


Page Cache

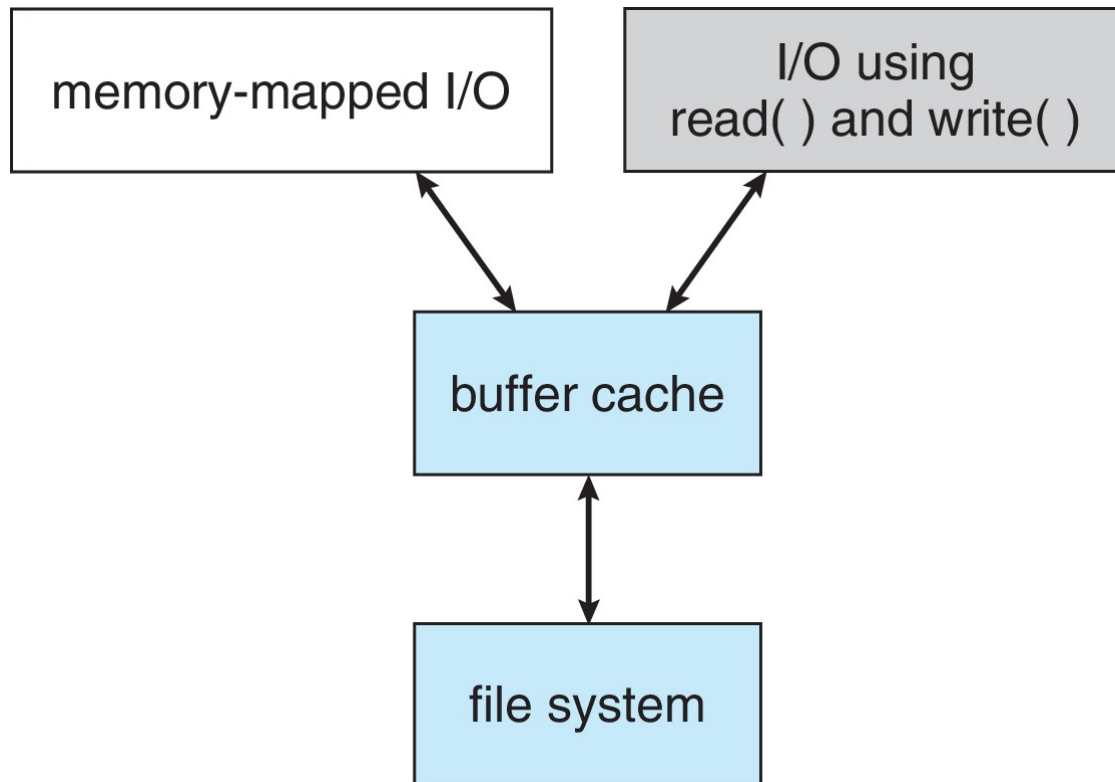
- A page cache caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure



I/O Without a Unified Buffer Cache



I/O Using a 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?

- *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

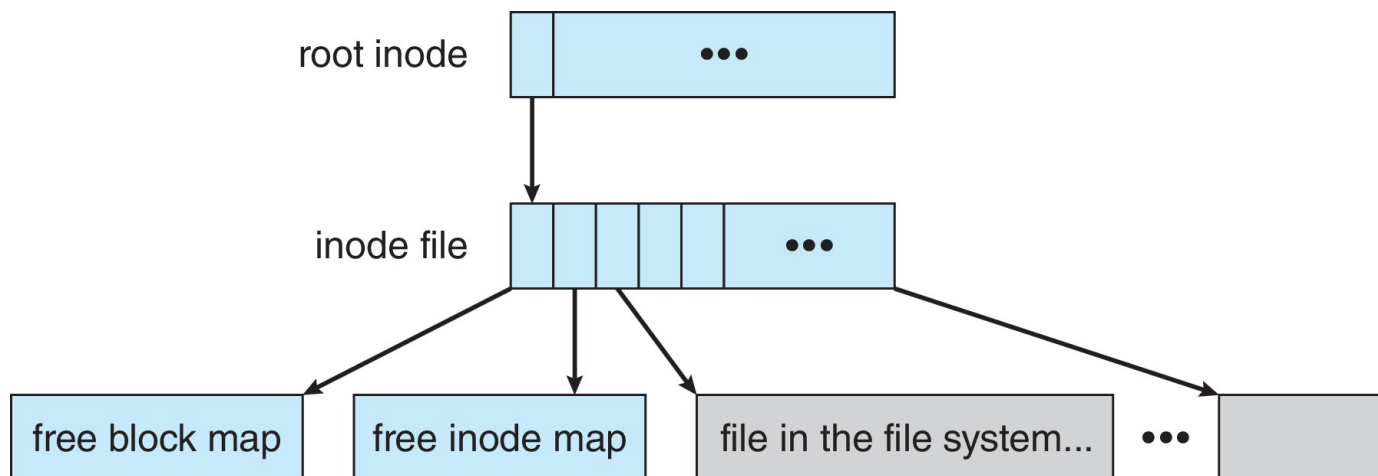
- *Log structured (or journaling) file systems* record each metadata update to the file system as a transaction
 - All transactions are written to a log
 - ▶ A transaction is considered committed once it is written to the log (sequentially)
 - ▶ Sometimes to a separate device or section of disk
 - ▶ 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



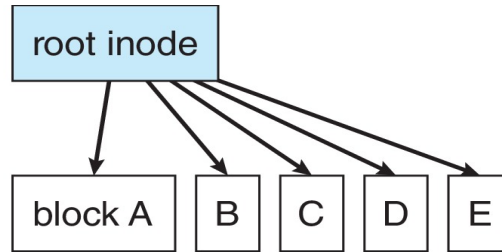
Example: WAFL File System

■ *Write-Anywhere File Layout* (WAFL)

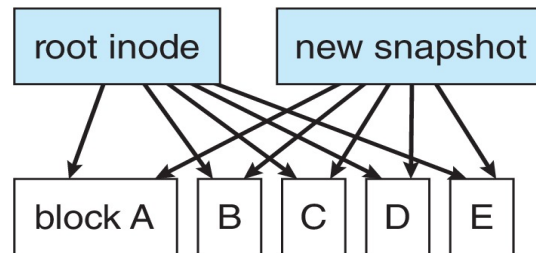
- Used on Network Appliance “Filers” – distributed file system appliances
- Serves up NFS, CIFS, http, ftp
- Random I/O optimized, write optimized
 - ▶ NVRAM for write caching
- Similar to Berkeley Fast File System, with extensive modifications



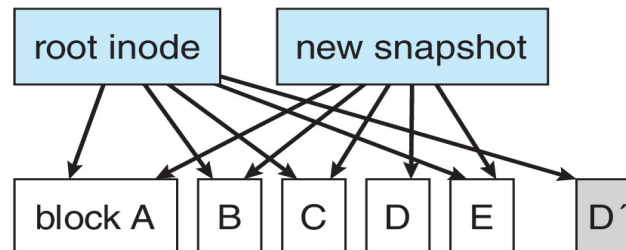
Snapshots in WAFL



(a) Before a snapshot.



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



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

The Apple File System

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.



Summary

- Most file systems reside on *secondary storage*, which is designed to hold a large amount of data permanently. The most common secondary-storage medium is the *disk*, but the use of *NVM devices* is increasing.
- Storage devices are segmented into *partitions* to control media use and to allow *multiple, possibly varying, file systems on a single device*. These file systems are mounted onto a logical file system architecture to make them available for use.
- *File systems* are often implemented in a *layered or modular structure*. The lower levels deal with the physical properties of storage devices and communicating with them. Upper levels deal with symbolic file names and logical properties of files.



Summary (Cont.)

- The various files within a file system can be allocated space on the storage device in three ways: through *contiguous*, *linked*, or *indexed allocation*. Contiguous allocation can suffer from external fragmentation. Direct access is very inefficient with linked allocation. Indexed allocation may require substantial overhead for its index block.

These algorithms can be optimized in many ways. Contiguous space can be enlarged through extents to increase flexibility and to decrease external fragmentation. Indexed allocation can be done in clusters of multiple blocks to increase throughput and to reduce the number of index entries needed. Indexing in large clusters is similar to contiguous allocation with extents.



Summary (Cont.)

- *Free-space allocation methods* also influence the efficiency of disk-space use, the performance of the file system, and the reliability of secondary storage. The methods used include bit vectors and linked lists. Optimizations include grouping, counting, and the FAT, which places the linked list in one contiguous area.
- *Directory-management routines* must consider efficiency, performance, and reliability. A hash table is a commonly used method, as it is fast and efficient. Unfortunately, damage to the table or a system crash can result in inconsistency between the directory information and the disk's contents.
- A *consistency checker* can be used to repair damaged file-system structures. Operating-system backup tools allow data to be copied to magnetic tape or other storage devices, enabling the user to recover from data loss or even entire device loss due to hardware failure, operating system bug, or user error.



Summary (Cont.)

- Due to the fundamental role that file systems play in system operation, their *performance* and *reliability* are crucial. Techniques such as log structures and caching help improve performance, while log structures and RAID improve reliability. The WAFL file system is an example of optimization of performance to match a specific I/O load.



End of Chapter 12

