Chapter 14: File System Implementation

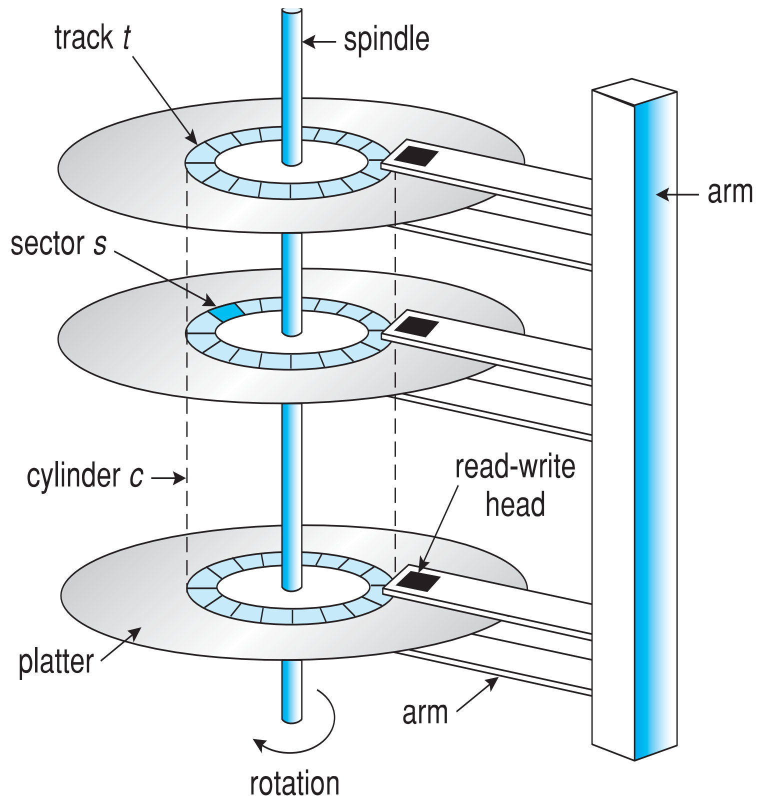
Outline

- File-System Structure
- File-System Operations
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- 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
- Describe the WAFL file system as a concrete example

Moving-head Disk Mechanism



- One sector is one block
- Disk systems typically have a well-defined block size (usually 512 bytes) determined by the size of a sector
- Disk provides in-place rewrite and random access
- All disk I/O is performed in units of one block (physical record) of the same size, indexed by block no. in a linear array
  - E.g., I/O sequence 20, 35, 11
- It is unlikely that the physical record size will exactly match the length of the desired logical record
  - Logical records may even vary in length
  - Packing a number of logical records into physical blocks is a common solution to this problem

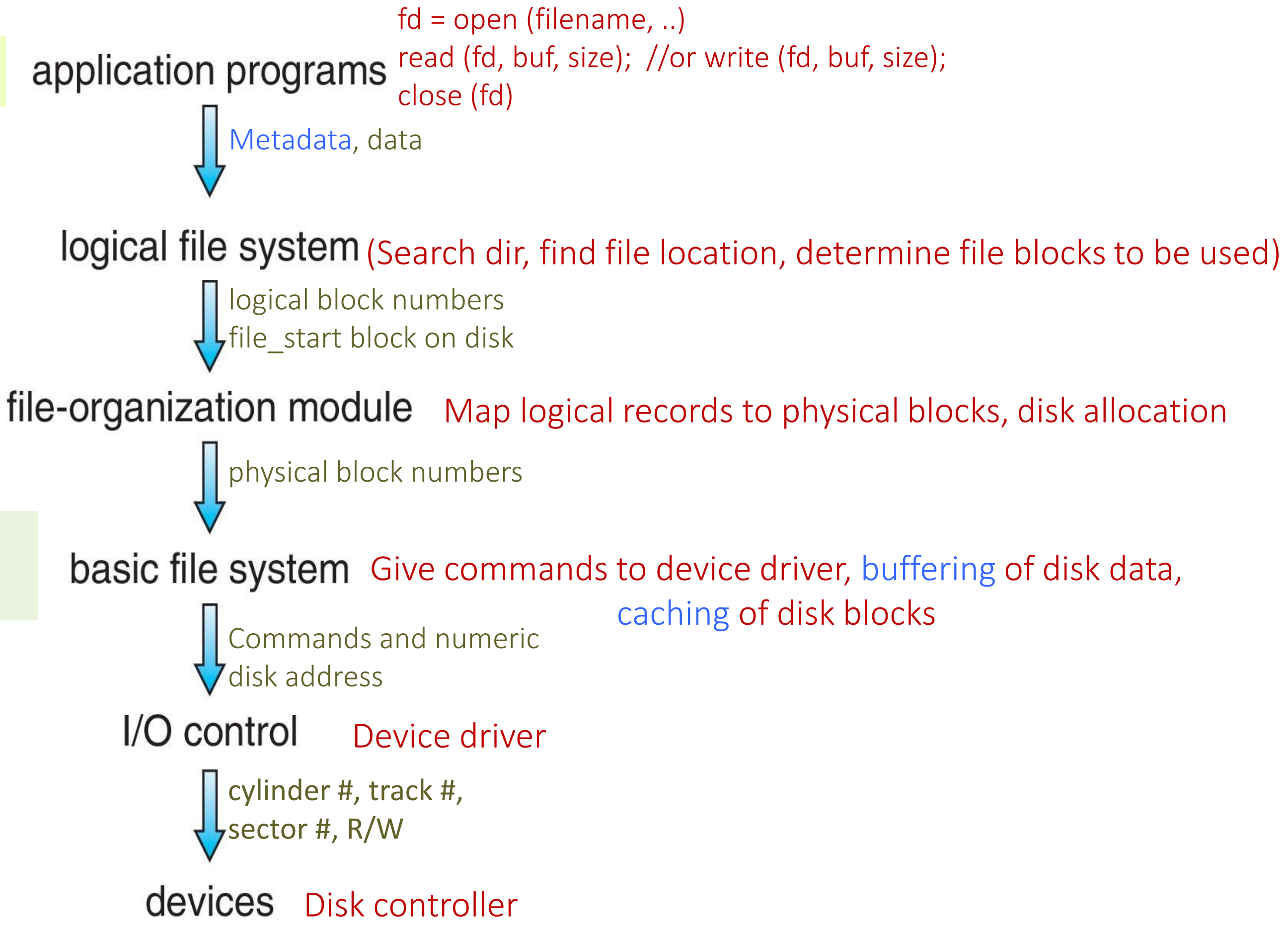
Layered File System

- File system organized into layers
- Each layer in the design uses the features of lower layers to create new features.

Process

Metadata: all details of a file except the file's contents

- Buffers hold data in transit
- Caches hold frequently used data



fd = open (filename, ..)
read (fd, buf, size); //or write (fd, buf, size);
close (fd)

(Search dir, find file location, determine file blocks to be used)

Map logical records to physical blocks, disk allocation

Give commands to device driver, buffering of disk data, caching of disk blocks

Device driver

Disk controller

A layered design: minimize the duplication of code

OS and File System

- One operating system can support one or more file systems
- Many operating systems allow administrators or users to add file systems
- Each file system has its own format
  - disk file-system formats
  - CD-ROM and DVD file-system formats (ISO 9660)
- Unix supports
  - Unix File System (UFS), which is based on Fast File System (FFS)
- Linux supports over 130 files systems
  - Extended File System (Standard Linux file system)
    - Versions: ext2, ext3, ext4
- Windows supports
  - Disk file formats
    - 16 bit File Allocation Table (FAT),
    - 32 bit File Allocation Table (FAT32),
    - New Technology File System (NTFS)
  - CD, DVD Blu-ray file system formats

File-System Implementation

- Structures used in file system implementation
  - Two categories of structures
    - on-disk (on-storage) structures contain information about
      - how to boot an operating system stored in disk,
      - the total number of blocks,
      - the number and location of free blocks,
      - the directory structure, and
      - individual files
    - in-memory structures contain information used for
      - file-system management
      - performance improvement via caching

On-disk (On-storage) Structures

Volume control block

•Volume details

⬥Total # of blocks, free blocks,

⬥block size,

⬥free-block count and free-block pointers,

⬥free-FCB count and FCB pointers

•Ext (Linux): called superblock

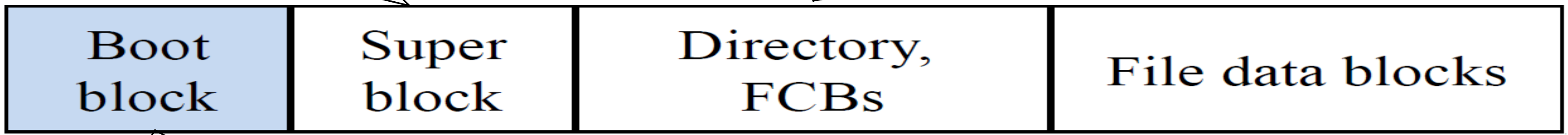
•NTFS (Windows): called master file table

Directory structure

•Organize the files

•UFS (Unix): includes file names and associated inode (same as File Control Blocks) numbers

•NTFS(Windows): File Control Blocks (FCB), called master file table



A volume: logical disk drive, perhaps a partition

Boot control block

•First block of volume

•Information for booting OS if OS is in this volume; otherwise empty

•UFS(Unix): called boot block

•NTFS(Windows): called partition boot sector

Each file has a File Control Block or inode

- Unix: Indexed using inode number, includes permissions, size, dates
- Windows: FCBs are stored in master file table using relational DB structures

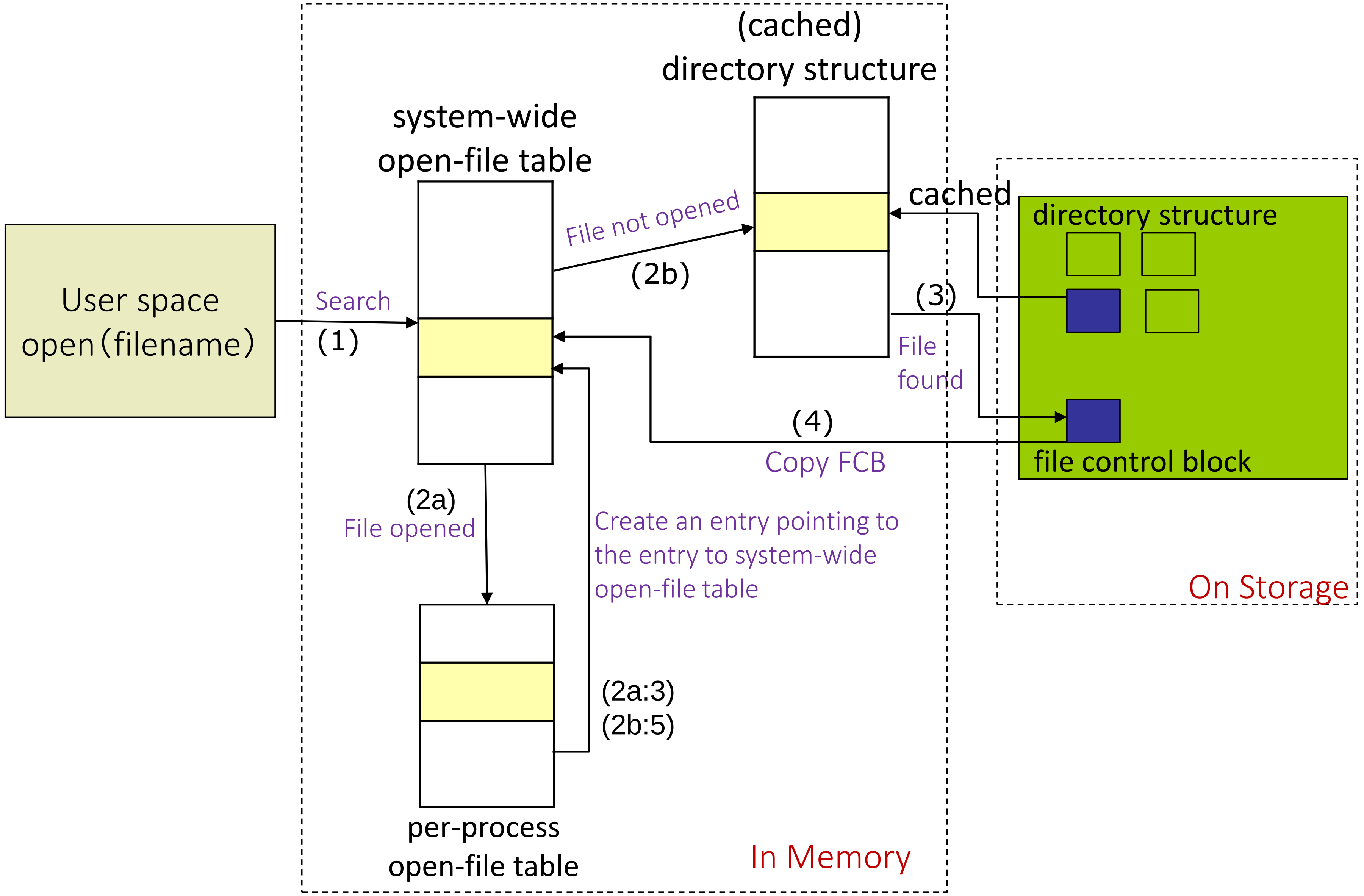
In-Memory Structures

- Mount table
  - stores file system mounts, mount points, file system types
- Directory-structure cache
  - holds the directory information of recently accessed directories
- system-wide open-file table
  - for all the processes
  - contains a copy of the FCB of each file
  - tracks all open files
  - keeps the counter of processes that have opened each file
  - contains process-independent information
    - E.g., the location of the file on disk, access dates, and file size
- per-process open-file table
  - contains pointers to appropriate entries in system-wide open-file table
  - tracks all the files that a process has opened
  - keeps a file pointer that indicates the last read-write position of the file for each opened file
  - keeps file access rights

File Open and Close

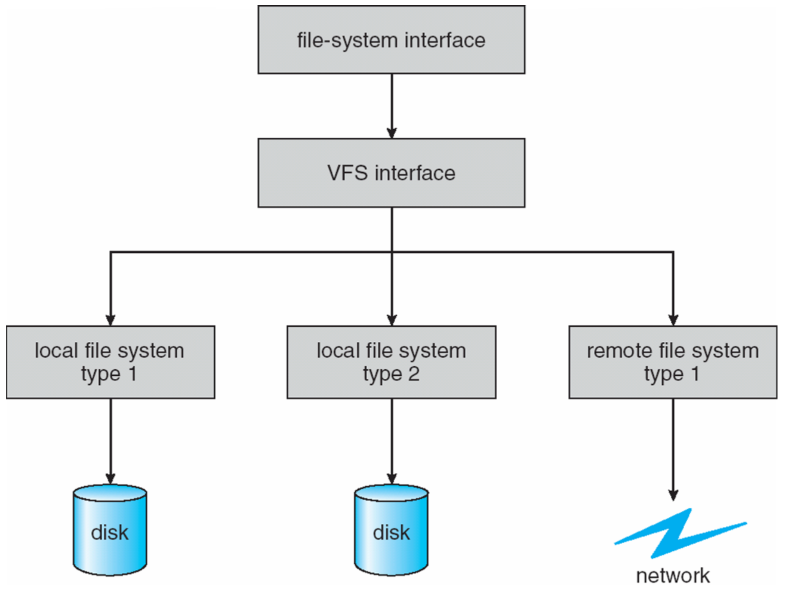
- Open a file
  - Search the directory structure on disk for the file and copy the content of entry（metadata）to system-wide open file table if the file is opened for the first time
  - Update the per-process open-file table by adding a pointer to system-wide open file table
- Close a file
  - Remove the file entry in system-wide open file table if no process to use it any more
  - Update the per-process table by removing a pointer to system-wide open file table

Open A File


(cached)
directory structure
system-wide
open-file table
User space
open(filename)
Search
(1)
File not opened
(2b)
cached
directory structure
(3)
File
found
(4)
Copy FCB
file control block
(2a)
File opened
Create an entry pointing to
the entry to system-wide
open-file table
On Storage
(2a:3)
(2b:5)
per-process
open-file table
In Memory

Virtual File Systems

- Virtual File Systems (VFS) on Unix provide an object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - Separates file-system generic operations from implementation details
  - Implementation can be one of many file systems types, or network file system
- The API is to the VFS interface, rather than any specific type of file system
- OO programming: polymorphism

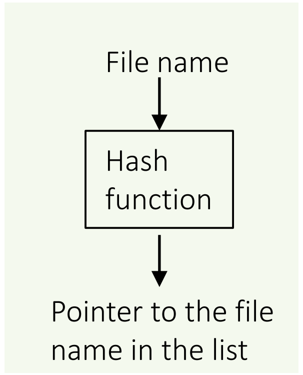


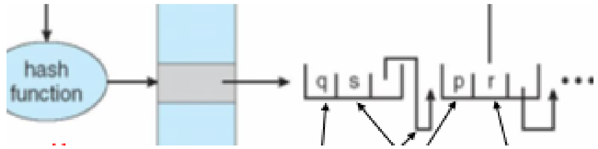
Virtual interface instead of specific File System interface

- For example, Linux has four object types:
  1. inode object: represent an individual file;
  2. file object: represent an open file;
  3. superblock object: represent an entire file system;
  4. dentry object: represent an individual directory entry.

Directory Implementation

- Methods for directory implementation
  - Linear list of file names with pointer to the data blocks
    - Advantage
      - Simple to program
    - Disadvantage
      - Time-consuming in finding a file
        - Linear search time
      - Improvement
        - keep ordered alphabetically via linked list or use B+ tree
  - Hash Table – linear list with hash data structure
    - Decreases directory search time
    - Problem: two file names might hash to the same location
    - Solution: use chained-overflow method
      - Each hash entry can be a linked list instead of an individual value.





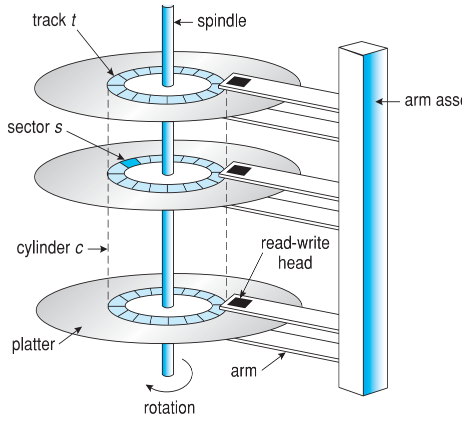
Similar to hashed page table

Allocation Methods

- How disk blocks are allocated for files
  - Contiguous allocation(not common now)
  - Linked allocation (e.g. FAT32 in Windows)
  - Indexed allocation (e.g. ex3 in Unix)

Allocation Methods - Contiguous

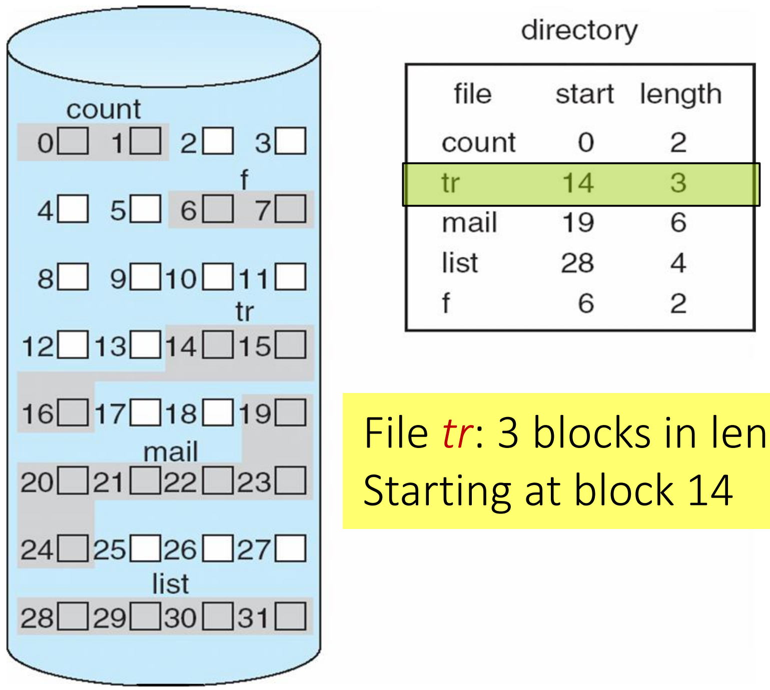
- Contiguous allocation
  - Each file occupies set of contiguous blocks
  - Each file can be located by start block # and length (total number of blocks)
    - E.g., A file that occupies n blocks: b, b+1, b+2, b+3, ..., b+n-1
  - Advantages
    - Simple
    - Each to access
    - Minimal disk head movement
      - Normally on same track, or at most move to next track.
  - Problems
    - Not suitable for a file with varied size
      - file size must be determined at the beginning (size-declaration)
    - External fragmentation
      - Solution: compacts all free space into one contiguous space
      - Compaction can be done off-line (down time) or on-line
    - Internal fragmentation



Contiguous Allocation: An Example

- Mapping from logical to physical
- Assumption
  - LA: Logical Address, starting from 0
  - Block size: 512 bytes
- Physical address
  - (Q + File physical start block) * 512 + R
    - Q: block no. relative to start
    - R: displacement into the block

$$\begin{tikzcd} & \overset{\text{Quotient}}{Q} \\ LA/512 \arrow[ur, dash] \arrow[dr, dash] & \\ & \underset{\text{Remainder}}{R} \end{tikzcd}$$



directory

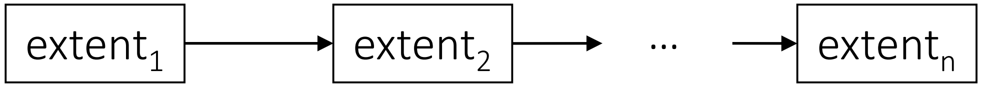
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

File *tr*: 3 blocks in length
Starting at block 14

Thinking:
For file ***tr***, if a byte's LA is 1025, what is its physical address?

Allocation Methods – Modified Contiguous Allocation

- Extent-Based File Systems
  - A modified contiguous allocation scheme
  - Allocate disk blocks in extents
    - An extent is a contiguous blocks (i.e., several contiguous blocks)

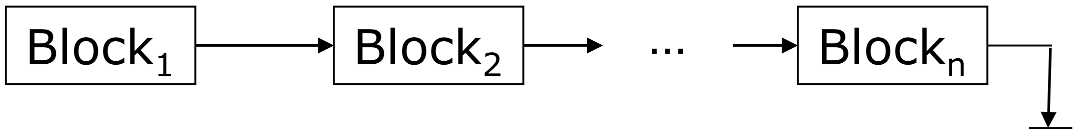


    - A file consists of one or more extents
    - Implementation
      - A contiguous chunk of space is allocated initially
      - If that amount proves not to be large enough, another extent is added
      - The location of a file's blocks is decided by
        - the location of the first extent and a block count
        - a link to the first block of the next extent
    - Cannot solve internal and external fragmentation

Thinking:
What is the advantage of this method over the previous contiguous method?

Allocation Methods – Linked Allocation

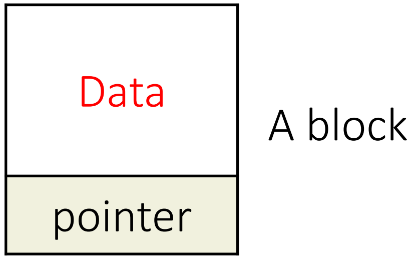
- Linked allocation
  - Each file is a linked list of blocks, ends at NULL pointer
  - Each block contains pointer to next block
  - Advantages
    - Solved the external-fragmentation problem and size-declaration problems of contiguous allocation
    - No need to find contiguous blocks



Thinking:
What is the disadvantage of this method over the previous contiguous method?

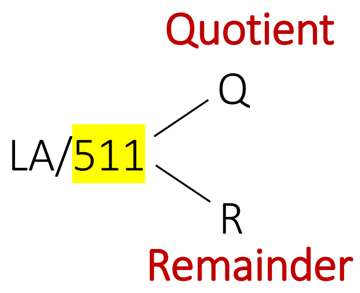
Linked Allocation: An Example

- Each file is a linked list of disk blocks
  - Blocks may be scattered anywhere on the disk
  - Physical address
    - Rth byte in the Qth block in the list (index starts from 0)
      - Q: block index in the list relative to the start
      - R: displacement into the block

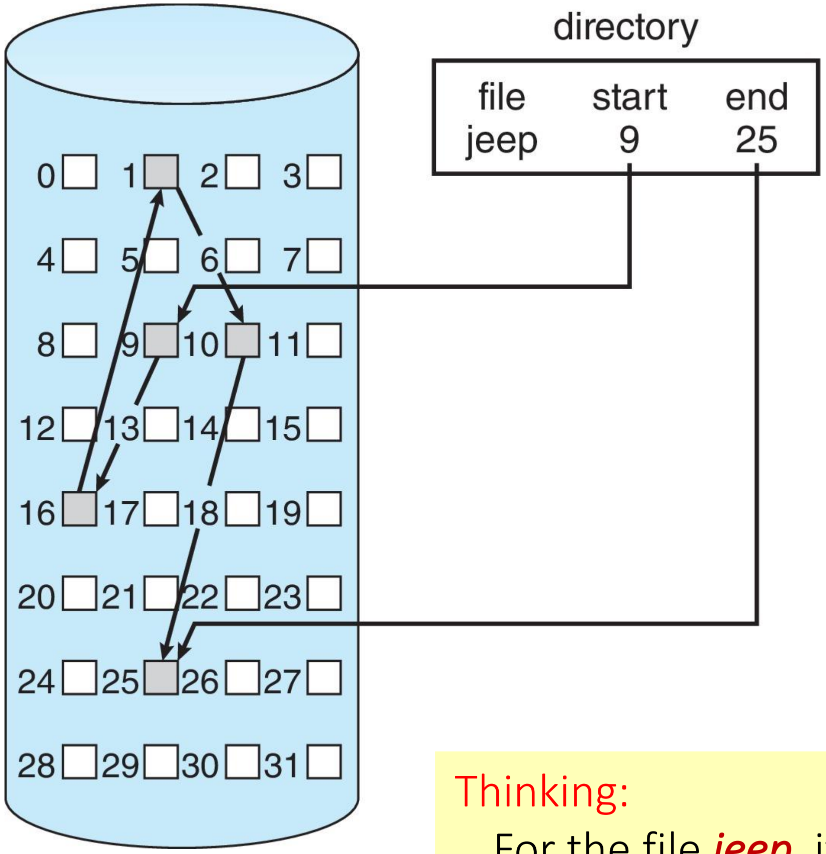


Assumption
Pointer: 1 byte
Block size: 512 bytes

Thinking:
Why divided by 511?



Linked Allocation

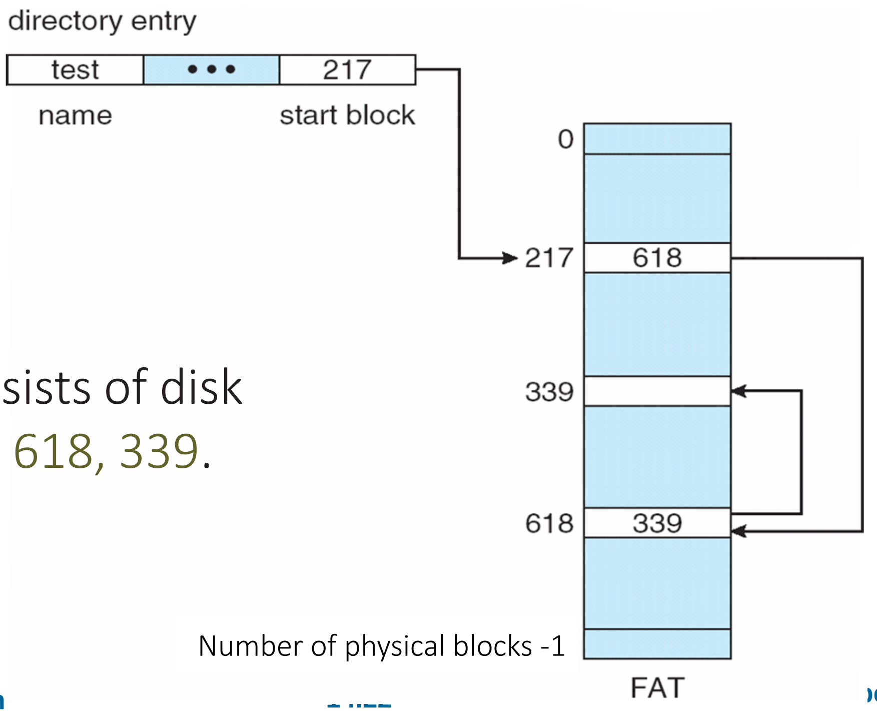


Thinking:

For the file ***jeep***, if LA is 512, what is its physical address?

Allocation Methods – Linked Allocation Variation

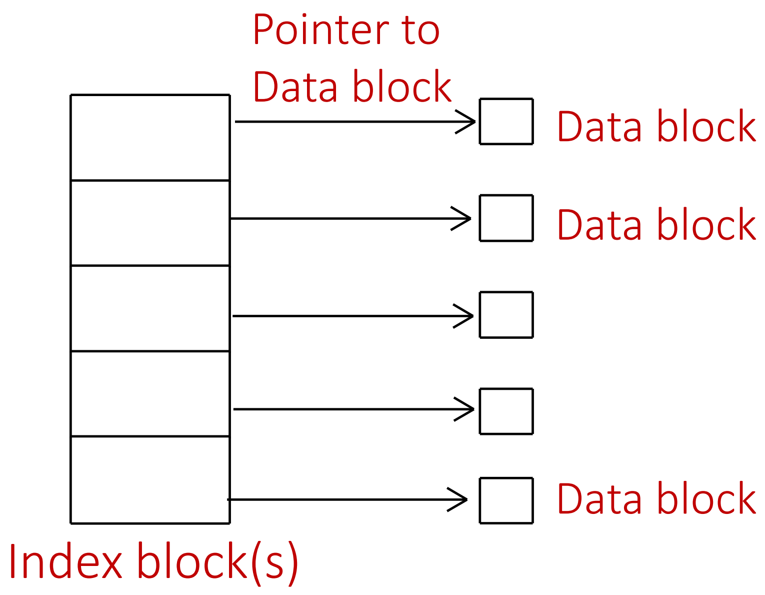
- File Allocation Table (FAT)
  - An important variation on linked allocation used in MS-DOS
  - Indexed in physical block no.
  - Each entry stores the no. of next block for this file
  - Like a linked list, but faster on disk and cacheable
  - Simple and efficient



File *test* consists of disk blocks: 217, 618, 339.

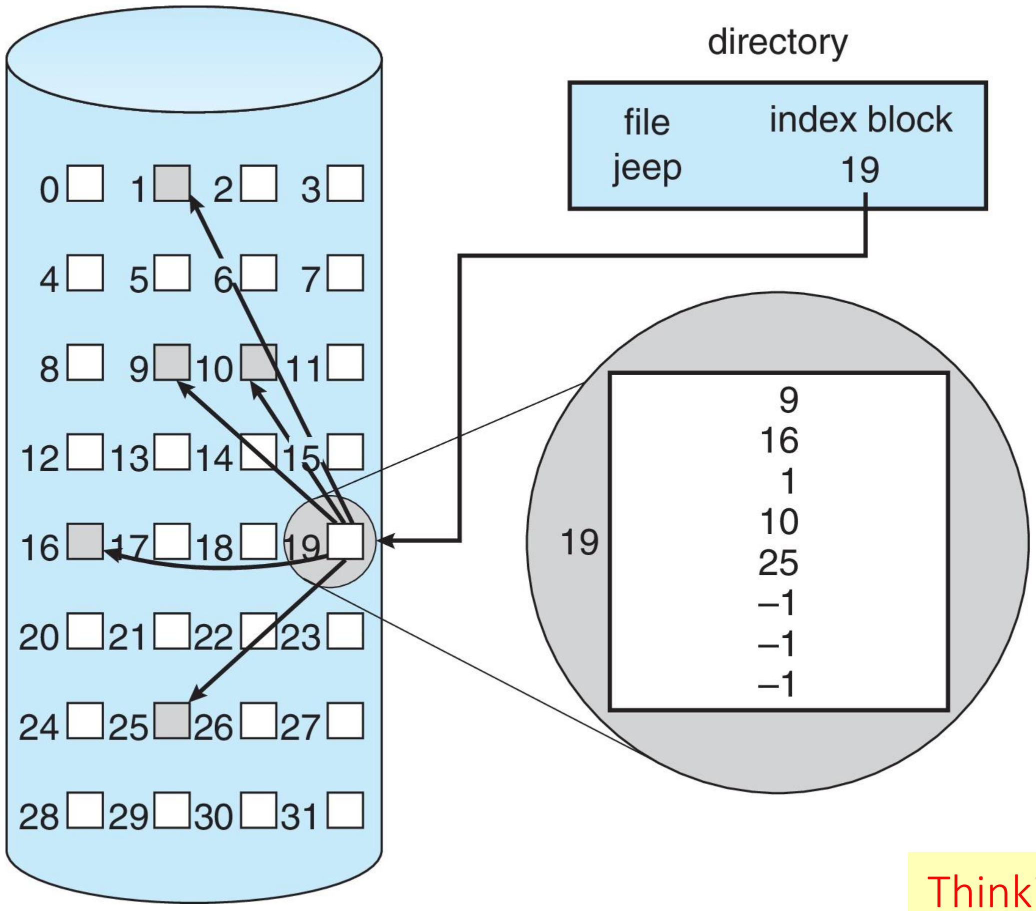
Allocation Methods – Indexed Allocation

- Indexed allocation
  - Bring all the pointers together into the index block(s)
  - Each entry points to a block in a file
  - Each file has its own index block(s) of pointers to its data blocks
  - Advantages
    - Random access
    - No external fragmentation
  - Disadvantages
    - Extra space for index table, more waste space than linked allocation



An index table

Example of Indexed Allocation



Indexed allocation of disk space

Assumption

Pointer: 1 byte

Block size: 512 bytes

Thinking:

1. For the file ***jeep***, if LA is 513, what is its physical address?
2. For a file has size 256K, how many blocks are used as index block?

A Comparison Table

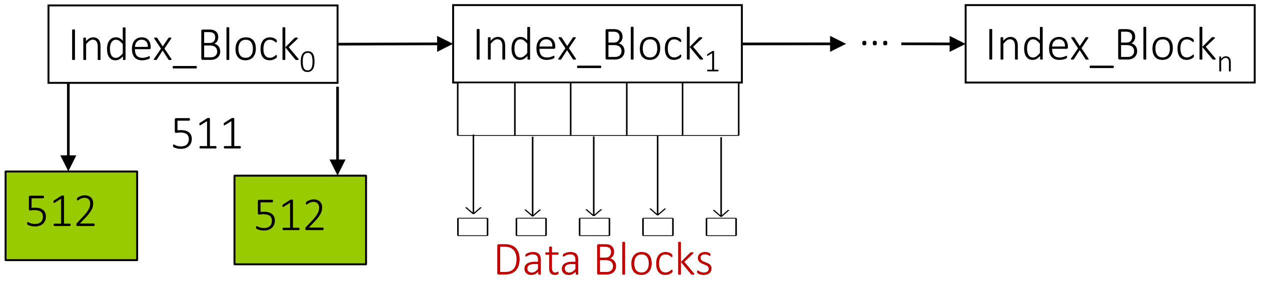
	Contiguous allocation	Linked allocation	Linked Allocation(FAT)	Indexed allocation
allocation	contiguous	scattered	scattered	scattered
Directory hold information	For each file: file name, start block, number of blocks	For each file: file name, start block, end block. Linked through pointer	• For each file: file name, start block • One volume table indexed by block no. including next block no. Linked through block no.	• For each file: Index block no • Index block: Pointers to data blocks
Direct access support	Yes	No	Yes (Partial)	Yes
fragmentation	External, internal	Internal	Internal	Internal
Overhead	No	Pointer	FAT	Index block
File size declaration	Necessary	Not necessary	Not necessary	Not necessary

Indexed Allocation – Schemes in Handling Large Files

- One index block is not enough for large files
- Three schemes to solve this issue
  1. Linked scheme
  2. Multilevel index
  3. Combined scheme

Indexed Allocation – Linked Scheme

- Linked scheme
  - Link blocks of index table
  - File has no limit on size (due to disk size)



Assumption
Pointer: 1 byte
Block size: 512 bytes

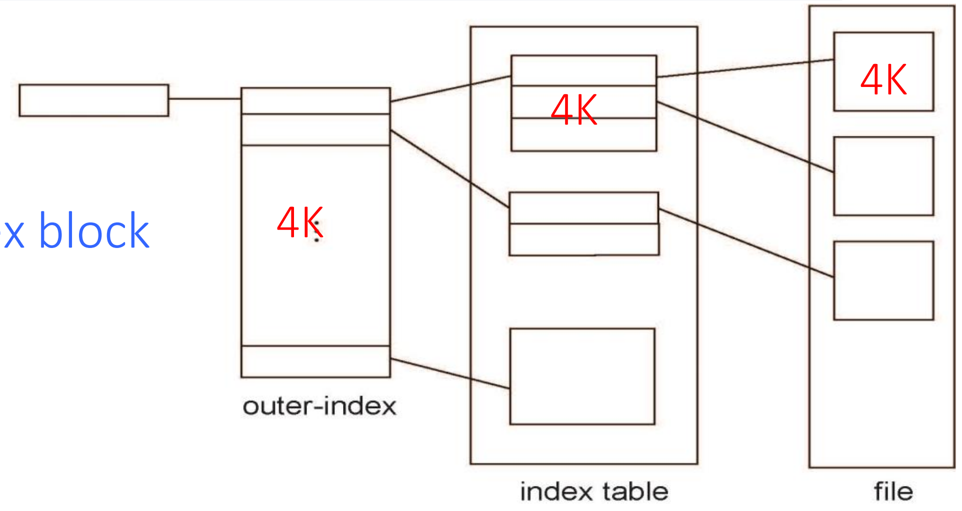
LA / (512 x 511) → Q_1 Index_Block_{Q_1} ; R_1 For further location

R_1 / 512 → Q_2 Q_2th Data_Block in Index_Block_{Q_1} ; R_2 Displacement into Data_Block_{Q_2}

Thinking:
If LA = 512 x 511+512, which data block is this byte in? (refer to above figure)?

Indexed Allocation – Multilevel Scheme

- Two-level index
  - One outer index block
    - Each entry points to an inner index block
  - Inner index blocks
    - Each entry points to a data block
  - Maximum size of a file
    - 1024 x 1024 x 1024 x 4 = 4G



Assumption

Pointer: **4** bytes

Block size: 4K bytes

(1024 x 4 =4096)

LA / (1024 x 4096) → Q_1 Displacement into outer-index block; R_1 For further location

R_1 / 4096 → Q_2 Displacement into inner index block (index table); R_2 Displacement into data block

Indexed Allocation - Combined Scheme

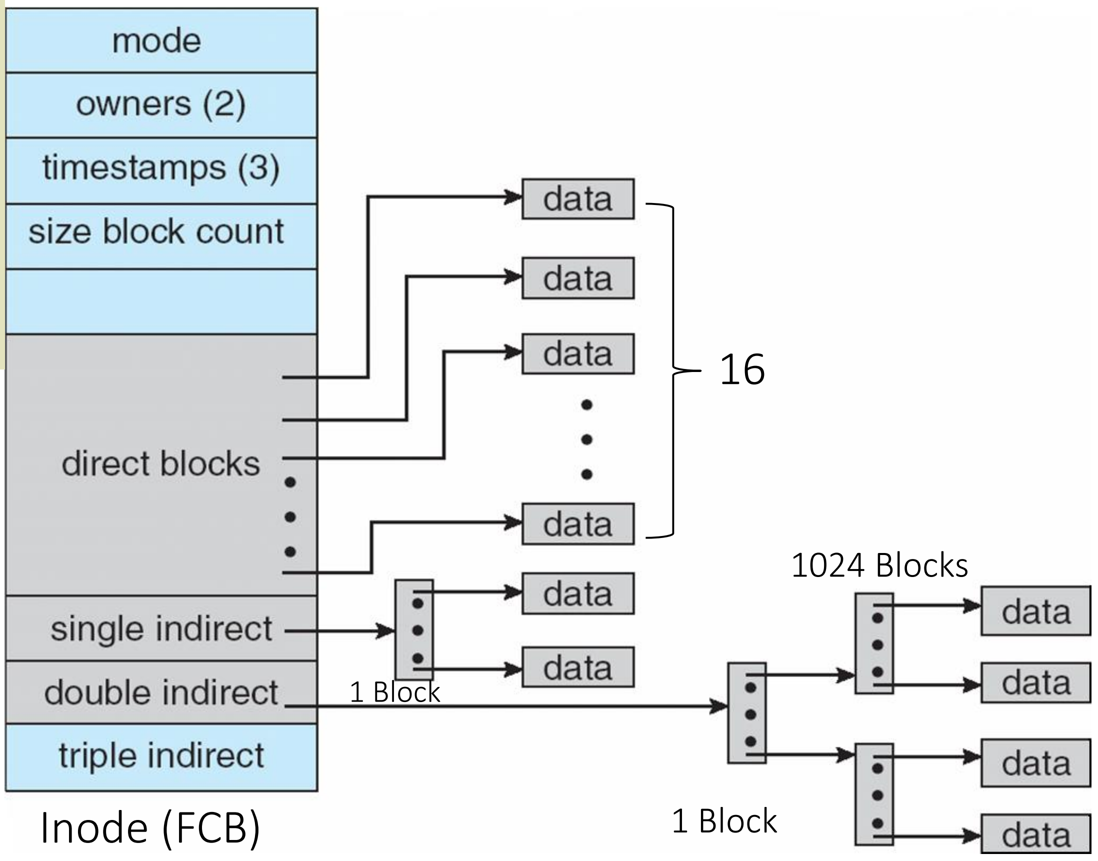
- Combined scheme
  - Used in Unix UFS
  - Small file stored in the inode itself
  - inode size is normally 128 bytes

Assumption
  Pointer: **4** bytes
  Block size: 4K bytes
    (1024 x 4 =4096)

Direct: (16 x 4 KB) =64KB
Single indirect: (1024 x 4 KB) =4MB
Double indirect:
 (1024 x 1024 x 4 KB) = 4GB
Triple indirect:
 (1024 x 1024 x 1024 x 4 KB) = 4TB
total: ≈ 4 terabytes

Medium: direct pointers to data 64B

large: indirect pointer 4B
larger: double indirect pointer 4B
huge: triple indirect pointer 4B



Performance

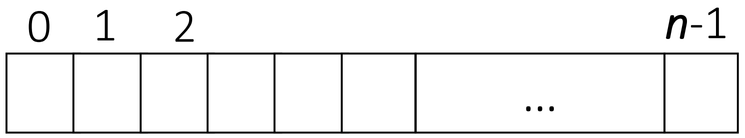
- Best method depends on file access type
  - Sequential access or random access
- Two or more level index increase access time (read index block)
- Adding instructions to the execution path to save one disk I/O is reasonable
  - One I/O time can execute many instructions
    - Intel Core i7 Extreme 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 MIPS / 250 = 630 million instructions during one disk I/O
    - Fast SSD drives provide 60,000 IOPS
      - 159,000 MIPS / 60,000 = 2.65 millions instructions during one disk I/O

Free-Space Management

- File system need to track available (free) blocks
- Approaches
    1. Bit vector (map)
    2. Linked list
    3. Grouping
    4. Counting

Free-Space Management – Bit Map

- Bit vector (map)
  - Block number calculation for first free block
    - (number of bits per word) x (number of 0-value words) + offset of first 1 bit in the first non-zero word



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

```
00000000
00000000
00111110
..
```

Example: 16 x 1 + 2

Assumption
Word: 2 bytes

- Advantages
  - Easy to get contiguous files if desired
  - Relative simple and efficient in finding the first free block or *n* consecutive free blocks on the disk
- Disadvantage
  - Bit map requires extra space
  - Example:

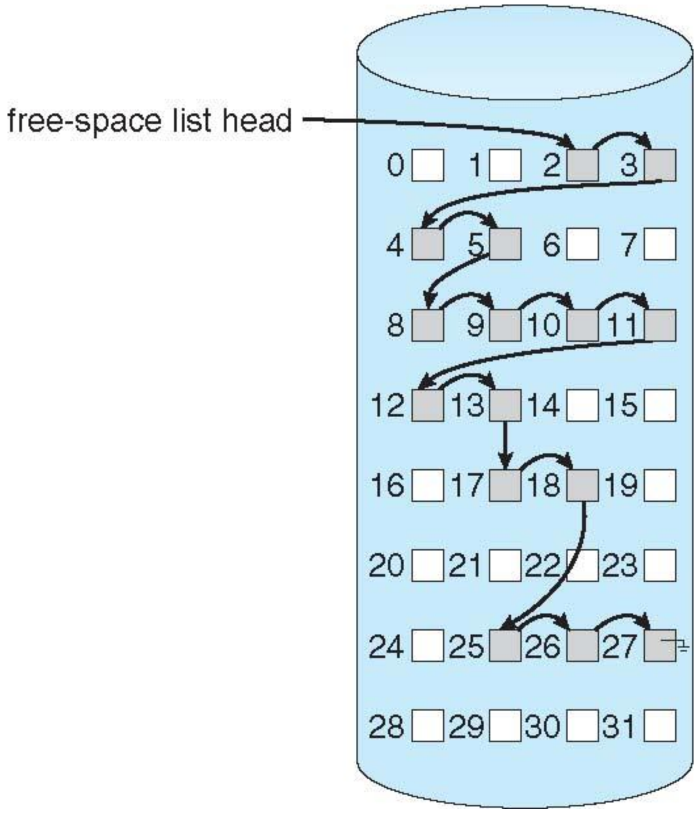
    block size = 4KB = 2^{12} bytes

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

    $n = 2^{40}/2^{12} = 2^{28}$ bits = 32MB for map

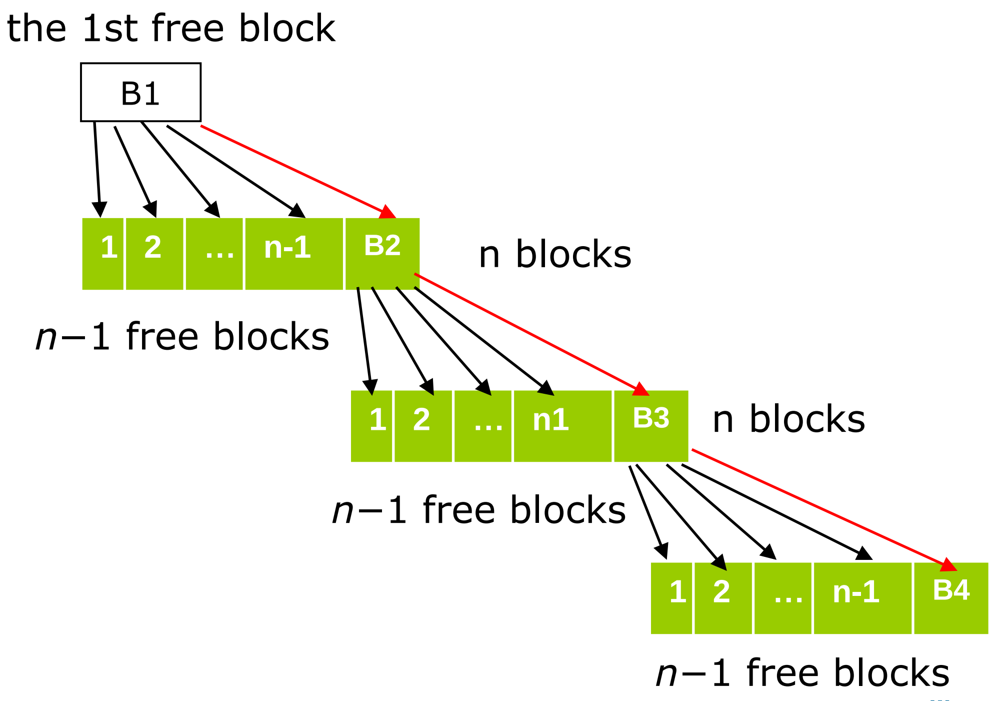
Free-Space Management – Linked List

- Linked list
  - List of free blocks
  - Superblock can hold pointer to head of linked list
  - Advantages
    - No waste of space
    - No need to traverse the entire list
  - Disadvantage
    - Cannot get contiguous space easily



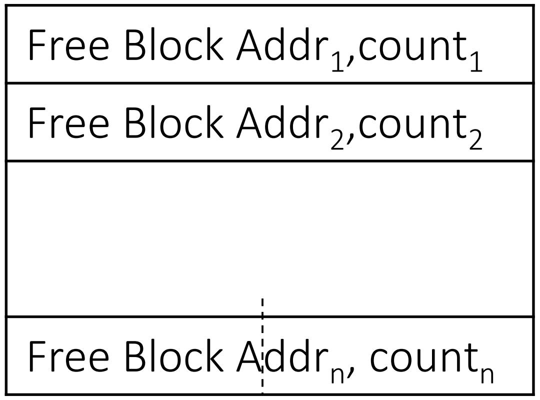
Free-Space Management – Grouping

- Grouping
  - A modification of the free-list approach
  - Stores the addresses of *n* free blocks in the first free block.
    - The first *n*−1 of these blocks are actually free.
    - The last block contains the addresses of another *n* blocks
  - Easy to find contiguous free blocks



Free-Space Management – Counting

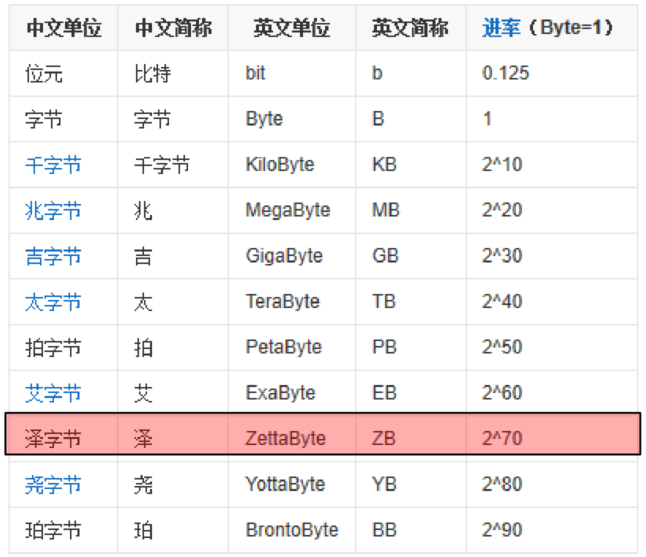
- Counting
  - Observation
    - space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering
  - This scheme keeps
    - the address of the first free block and
    - the number (*n*) of free contiguous blocks that follow the first block
  - Each entry in the free-space list consists of a disk address and a count.


Free Block Addr_1,count_1
Free Block Addr_2,count_2
Free Block Addr_n, count_n

the free-space list: A vector or a linked list

Free-Space Management - Others

- Space Maps
  - Used in Oracle's ZFS (Zettabyte File System)
- ZFS
  - encompass huge numbers of files, directories, and even file systems
  - divides device space into metaslabs
    - allocates or frees space from a metaslab
    - each metaslab has associated space map
    - the space map is a log of all block activity in counting format
    - loads the space map into memory in balanced-tree when allocate or free from metaslab
  - Uses a combination of schemes
    - Counting
    - Map



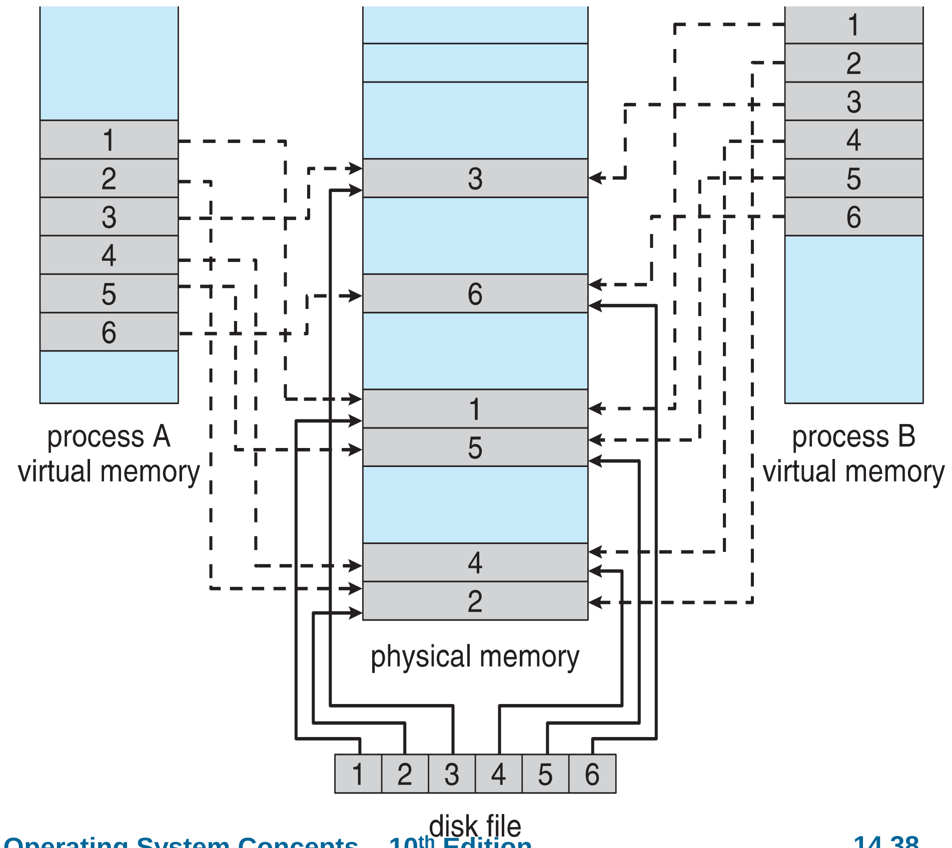
中文单位	中文简称	英文单位	英文简称	进率（Byte=1）
位元	比特	bit	b	0.125
字节	字节	Byte	B	1
千字节	千字节	KiloByte	KB	2^10
兆字节	兆	MegaByte	MB	2^20
吉字节	吉	GigaByte	GB	2^30
太字节	太	TeraByte	TB	2^40
拍字节	拍	PetaByte	PB	2^50
艾字节	艾	ExaByte	EB	2^60
泽字节	泽	ZettaByte	ZB	2^70
尧字节	尧	YottaByte	YB	2^80
珀字节	珀	BrontoByte	BB	2^90

Efficiency and Performance

- The efficient use of storage device space depends on
  - disk allocation method
    - Contiguous, Linked, and Indexed
  - directory structure / algorithms
  - types of data kept in file' s directory entry
  - allocation of metadata structures
  - fixed-size or varying-size data structures
    - e.g. the open file tables
- Ways to improve efficiency and performance
  - Keep data and metadata close together (for hard disk, but not SSD)
  - Use buffer cache for quick access
  - Use asynchronous writes
    - No need to write to disk immediately
  - Use Trim mechanism
    - Keep the unused blocks for writing
    - No need to do garbage collection and erase steps before the device is nearly full

Memory Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping disk blocks to pages in memory
- Application of virtual memory techniques
- Allows a part of the virtual address space to be logically associated with the file

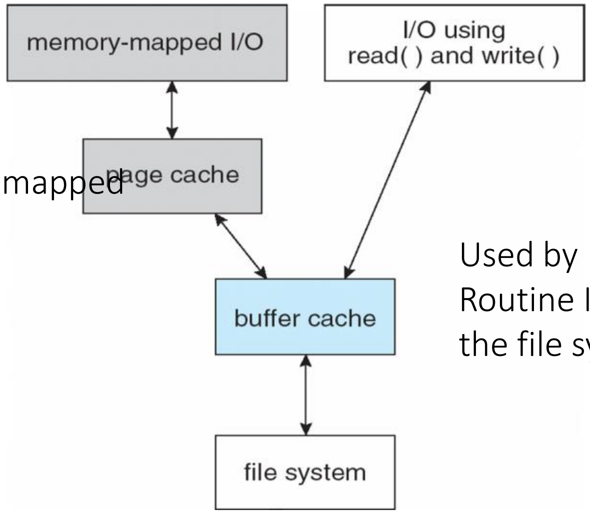


- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system into a physical page/frame
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than read() and write() system calls
- File in memory can be shared by multiple processes
- Copy-on Write can be applied

Unified Buffer Cache

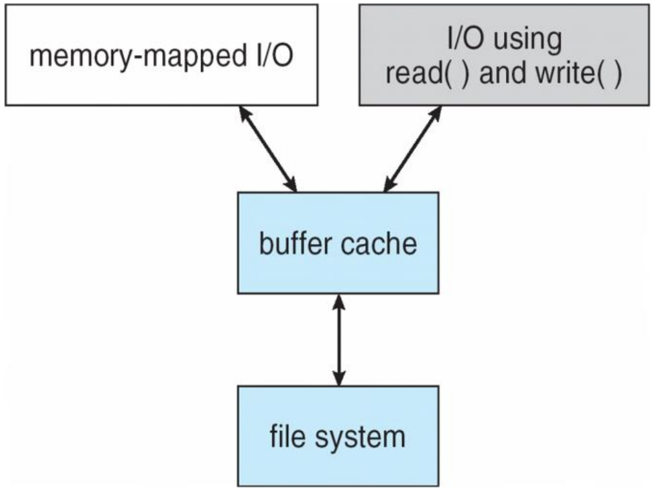
- A page cache caches pages rather than disk blocks using virtual memory techniques and addresses
  - Caching file data using virtual address is more efficient than caching through physical disk blocks
- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to
  - avoid double caching, and
  - allows the virtual memory system to manage file-system data more efficiently
- Some systems use unified buffer cache, such as Linux, Windows, etc.

Used by Memory-mapped I/O



Used by Routine I/O through the file system

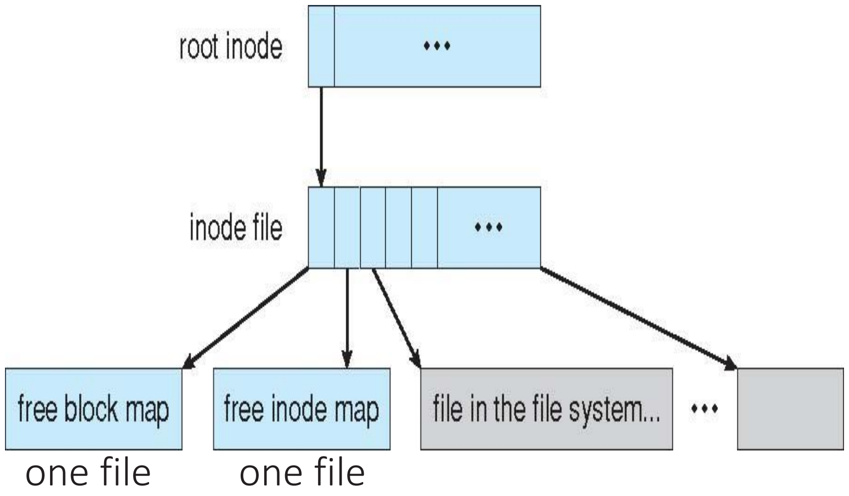
double caching



Unified buffer cache

Example: WAFL File System

- WAFL
  - Write-Anywhere File Layout
  - Used on distributed file system appliances (filers)
  - Serves NFS, CIFS, http, ftp

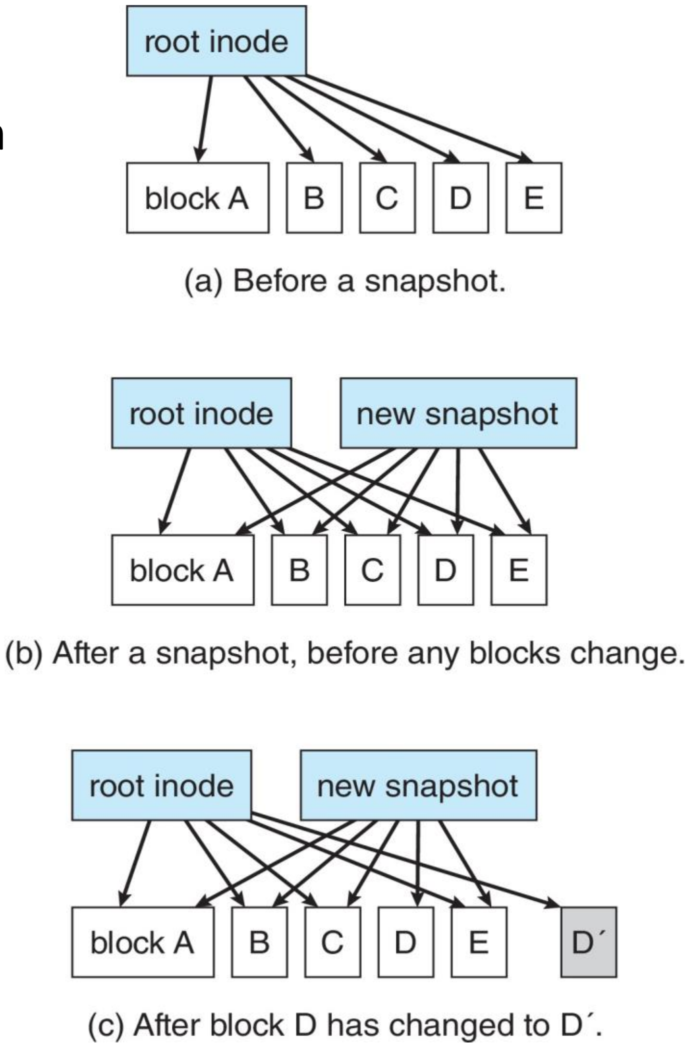


The WAFL File Layout

- Layout
  - It is block-based and uses inodes to describe files.
  - Each file system has a root inode. All of the metadata lives in files
  - All inodes are in one file
  - Each inode contains 16 pointers to blocks (or indirect blocks) belonging to the file described by the inode
  - Each free block map is in a file

Snapshots in WAFL

- Snapshot is
  - a view of the file system at a specific point in time before any updates
  - a method used in consistency checking and recovery
  - useful for backups, testing, versioning, and so on
- WAFL's snapshot facility is very efficient.

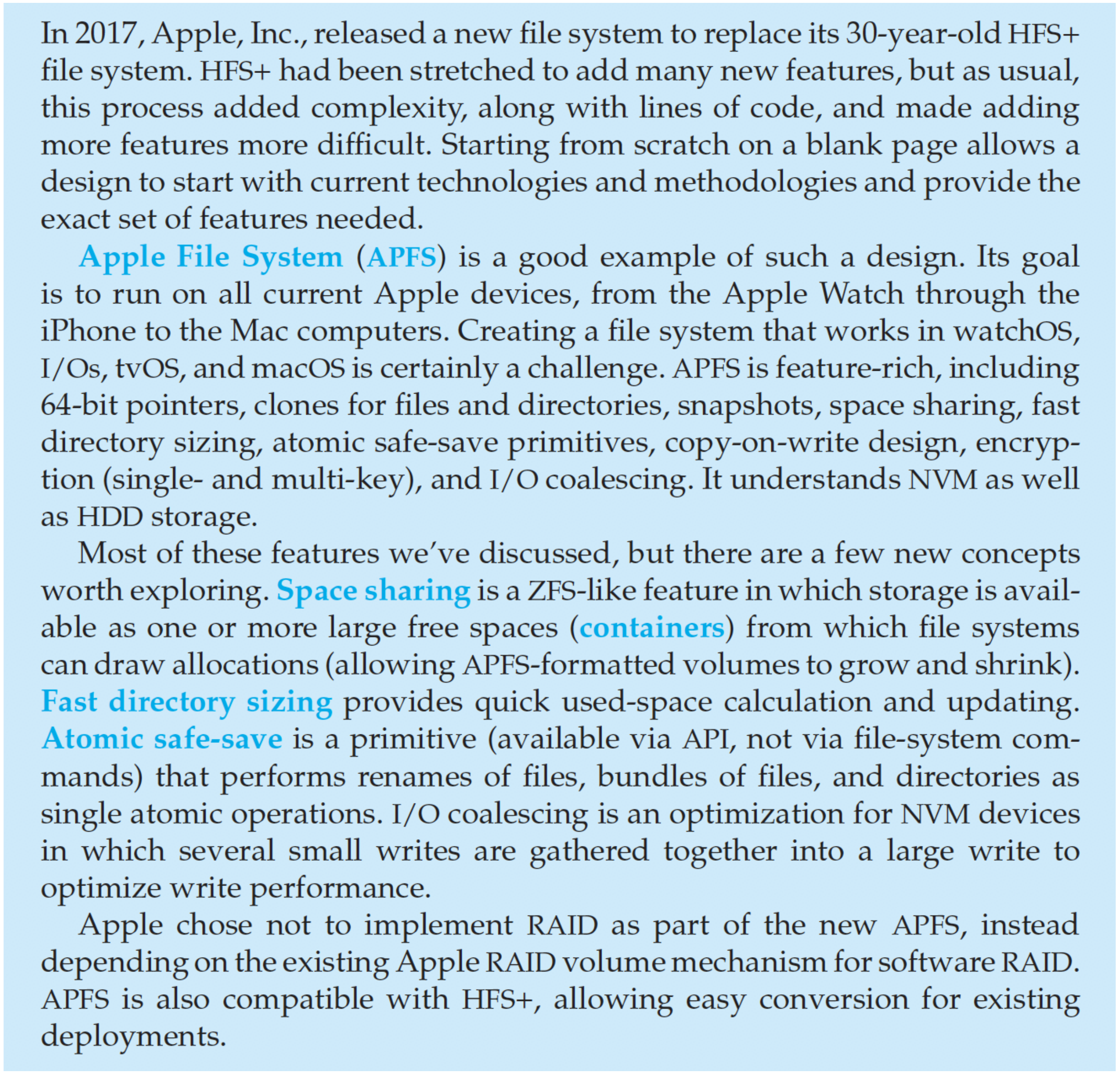


(a) Before a snapshot.

(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