

File System Implementation

Operating Systems
Wenbo Shen

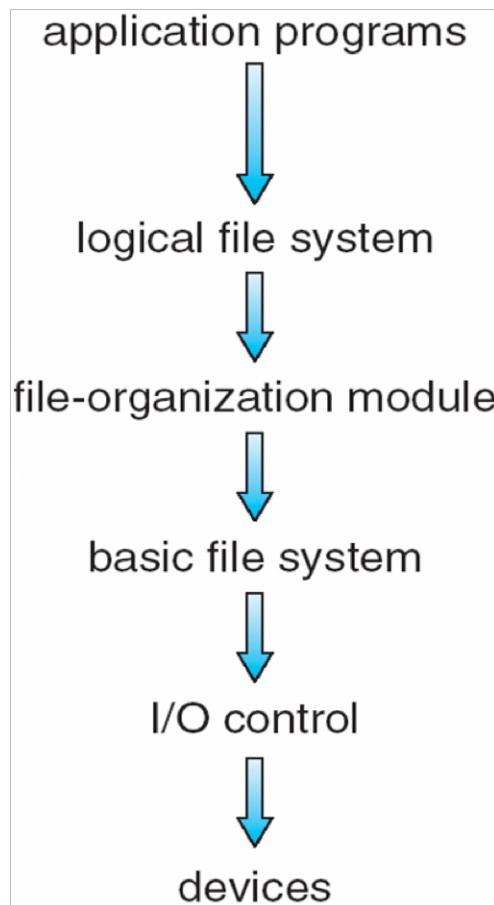
Review

- File system
- File operations
 - Create, open, read/write, close
- Directory structure
 - Single level, two-level, tree, acyclic-graph, general graph

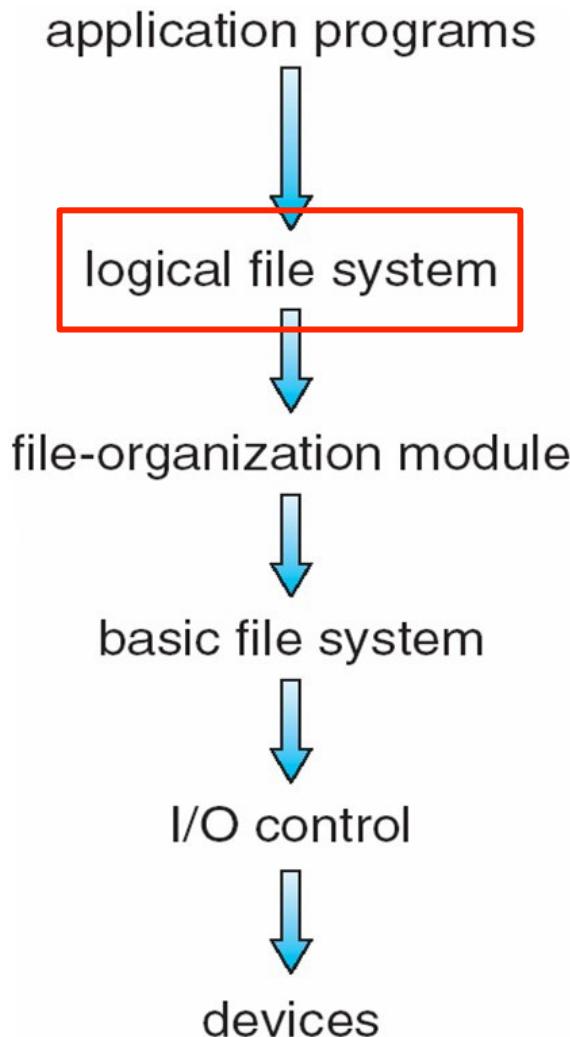
File-System Structure

- File is a logical storage unit for a collection of related information
- There are many file systems; OS may support several **simultaneously**
 - Linux has Ext2/3/4, Reiser FS/4, Btrfs...
 - Windows has FAT, FAT32, NTFS...
 - new ones still arriving - ZFS, GoogleFS, Oracle ASM, FUSE
- File system resides on **secondary storage** (disks)
 - disk driver provides interfaces to read/write disk blocks
 - **fs** provides user/program interface to storage, mapping logical to physical
 - file control block - storage structure consisting of information about a file
- File system is usually implemented and organized into **layers**

Layered File System

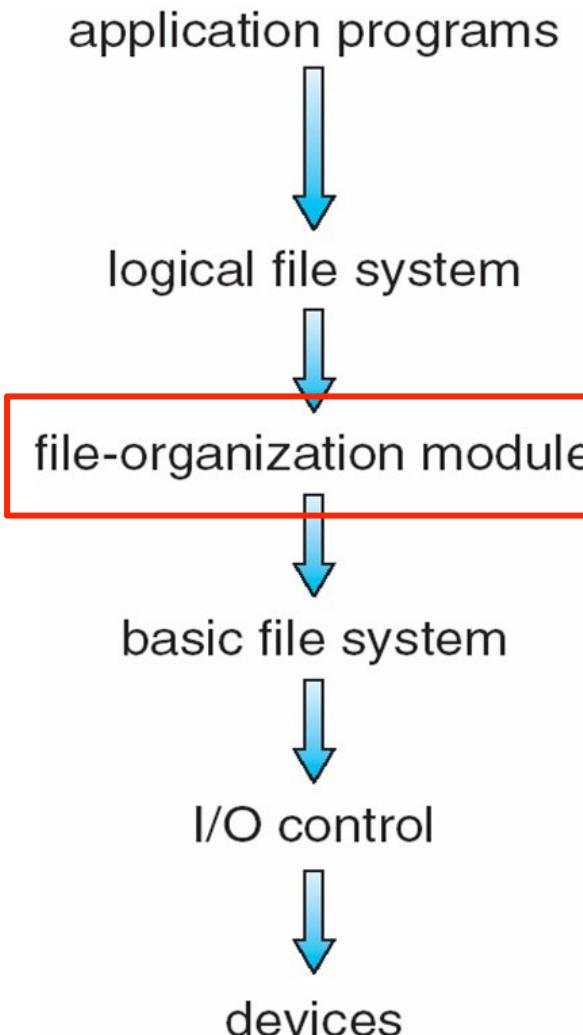


File System Implementation



- Logical file system
- Keep all the meta-data necessary for the file system
 - i.e., everything but file content
- It stores the directory structure
- It stores a data structure that stores the file description (**File Control Block - FCB**)
 - Name, ownership, permissions
 - Reference count, time stamps, pointers to other FBCs
 - Pointers to data blocks on disk
- Input from above:
 - Open/Read/Write filepath
- Output to below:
 - Read/Write logical blocks

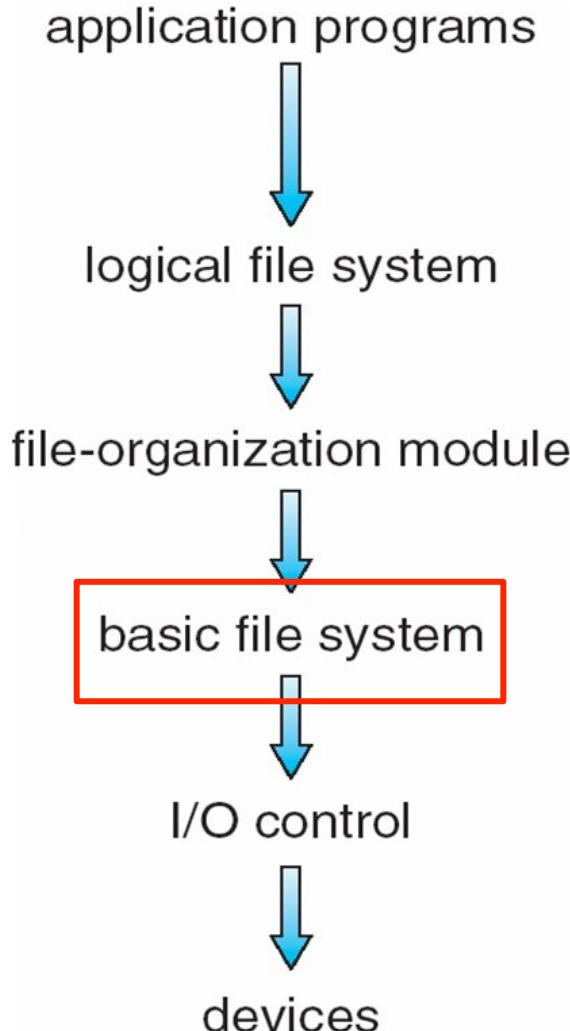
File System Implementation



- File-organization module
- Knows about **logical** file blocks (from 0 to N) and corresponding **physical** file blocks: it performs **translation**
- It also manages **free space**
- Input from above:
 - Read **logical** block 3
 - Write **logical** block 17
- Output below:
 - Read **physical** block 43
 - Write **physical** block 421

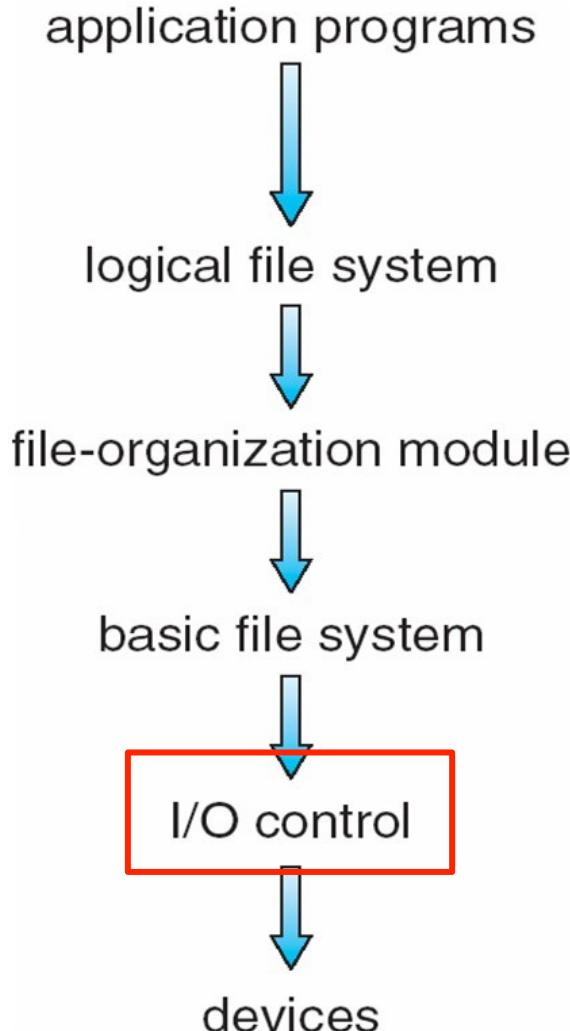
□

File System Implementation



- Basic file system
- Allocates/maintains various buffers that contain file-system, directory, and data blocks
- These buffers are caches and are used for enhancing performance
- Input from above:
 - Read physical block #43
 - Write physical block #421
- Output below:
 - Read physical block #43
 - Write physical block #421

File System Implementation



- **I/O Control**
- Device drivers and interrupt handlers
- Input from above:
 - Read physical block #43
 - Write physical block #124
- Output below:
 - Writes into device controller's memory to enact disk reads and writes
 - React to relevant interrupts

Layered File System

- Layering useful for **reducing complexity** and redundancy, but adds overhead and can **decrease performance**
 - Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
 - Logical layers can be implemented by any coding method according to OS designer

File Systems

- Most OSes support many file systems
 - e.g., the ISO 9660 file system standard for CD-ROM
- UNIX:
 - UFS (UNIX FS), based on BFFS (Berkeley Fast FS)
- Windows:
 - FAT, FAT32, and NTFS
- Basic Linux supports 40+ file systems
 - Standard: ext2 and ext3 (Extended FS)
- An active area of research and development
 - Distributed File Systems
 - Not new, but still a lot of activity
 - High-Performance File Systems
 - The Google File System

File System Data Structures

- The file system comprises data structures
- **On-disk structures:**
 - An optional **boot control block**
 - First block of a volume that stores an OS
 - boot block in UFS, partition boot sector in NTFS
 - A **volume control block**
 - Contains the number of blocks in the volume, block size, free-block count, free-block pointers, free-FCB count, FCB-pointers
 - superblock in UFS, master file table in NTFS
 - A **directory**
 - File names associated with an ID, FCB pointers
 - A **per-file File Control Block (FCB)**
 - In NTFS, the FCB is a row in a relational database
- **In-memory structures:**
 - A **mount table** with one entry per mounted volume
 - A **directory cache** for fast path translation (performance)
 - A **global open-file table**
 - A **per-process open-file table**
 - Various **buffers** holding disk blocks "in transit" (performance)

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

File Creation

- application process requests the creation of a new file
- logical file system allocates a new FCB, i.e., inode structure
- appropriate directory is updated with the new file name and FCB, i.e., inode

Directory

- Unix - directories are treated as files containing special data
- Windows - directories differently from files;
 - they require a separate set of systems calls to create, manipulate, etc

Operations - open()

- search **System-Wide Open-File Table** to see if file is currently in use
 - if it is, create a **Per-Process Open-File** table entry pointing to the existing **System-Wide Open-File Table**
 - if it is not, search the directory for the file name; once found, place the **FCB** in the **System-Wide Open-File Table**
- make an entry, i.e., Unix file descriptor, Windows file handle in the **Per-Process Open-File Table**, with pointers to the entry in the **System-Wide Open-File Table** and other fields which include a pointer to the current location in the file and the access mode in which the file is open

Operations - open()

- increment the open cont in the System-Wide Open-File Table
- returns a pointer to the appropriate entry in the Per-Process Open-File Table
- all subsequent operations are performed with **this pointer**
- process closes file -> Per-Process Open-File Table entry is removed; **open count decremented**
- all processes close file -> copy in-memory directory information to disk and System-Wide Open-File Table is removed from memory

Unix (UFS)

- System-Wide Open-File Table holds inodes (index node) for files, directories, devices, and network connections
- inode numbering system is only unique within a given file system

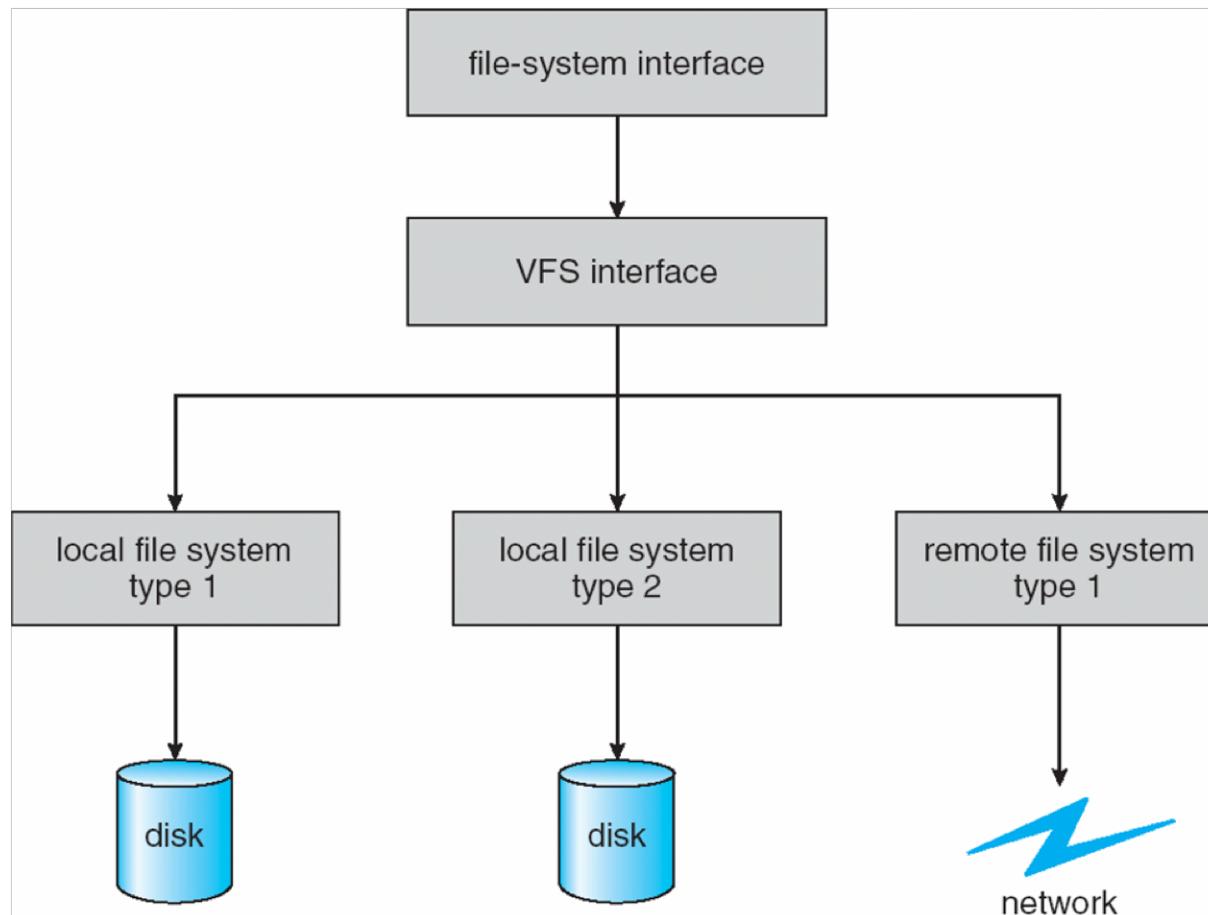
Mounting File Systems

- Boot Block - series of sequential blocks containing a memory image of a program, call the boot loader, that locates and mounts the root partition; the partition contains the kernel; the boot loader locates, loads, and starts the kernel executing
- In-memory mount table - external file systems must be mounted on devices, the mount table records the mount points, types of file systems mounted, and an access path to the desired file system
- Unix - the in-memory mount table contains a pointer to the **superblock** of the file system on that device

Virtual File Systems

- **VFS** provides an **object-oriented** way of implementing file systems
 - OS defines a **common interface** for FS, all FSes implement them
 - *system call is implemented based on this common interface*
 - it allows the same syscall API to be used for different types of FS
- VFS separates FS generic operations from implementation details
 - implementation can be one of many FS types, or network file system
 - OS can dispatches syscalls to appropriate FS implementation routines

Virtual File System



Virtual File System Example

- Linux defines four **VFS object types**:
 - **superblock**: defines the file system type, size, status, and other metadata
 - **inode**: contains metadata about a **file** (location, access mode, owners...)
 - **dentry**: associates names to inodes, and the directory layout
 - **file**: actual data of the file
- VFS defines set of operations on the objects that must be implemented
 - the set of operations is saved in a function table

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t);
    ssize_t (*write) (struct file *, const char __user *, size_t,
                     size_t);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iopoll)(struct kiocb *kiocb, bool spin);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    __poll_t (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned int);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    unsigned long mmap_supported_flags;
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
```

Directory Implementation

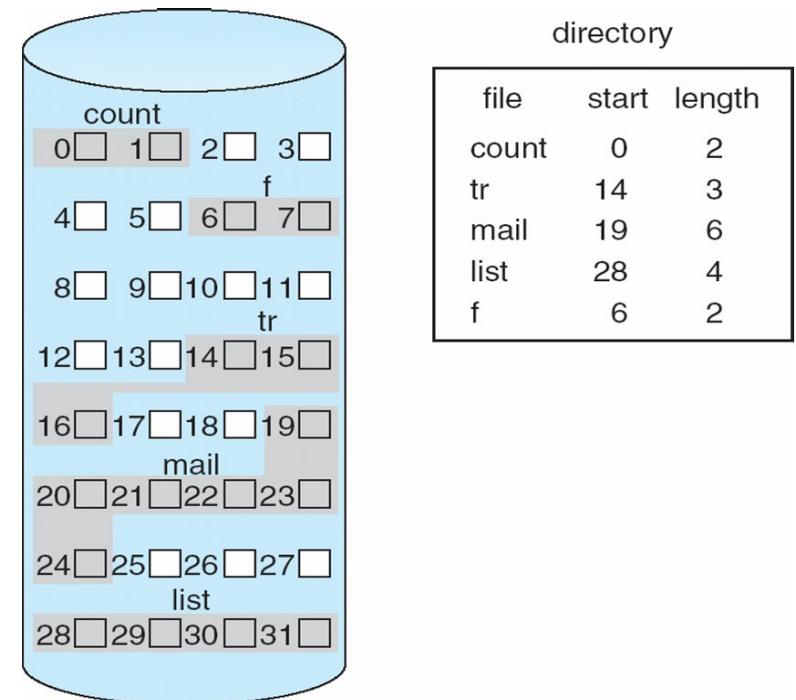
- **Linear list of file names** with pointer to the file metadata
 - simple to program, but **time-consuming to search** (e.g., linear search)
 - could keep files ordered alphabetically via linked list or use B+ tree
- **Hash table:** linear list with hash data structure to reduce search time
 - collisions are possible: two or more file names hash to the same location

Disk Block Allocation

- Files need to be allocated with disk blocks to store data
 - different allocation strategies have different complexity and performance
- Many allocation strategies:
 - contiguous
 - linked
 - indexed
 - ...

Contiguous Allocation

- Each file is in a set of contiguous blocks
 - Good because sequential access causes little disk head movement, and thus short seek times
- The directory keeps track of each file as the address of its first block and of its length in blocks



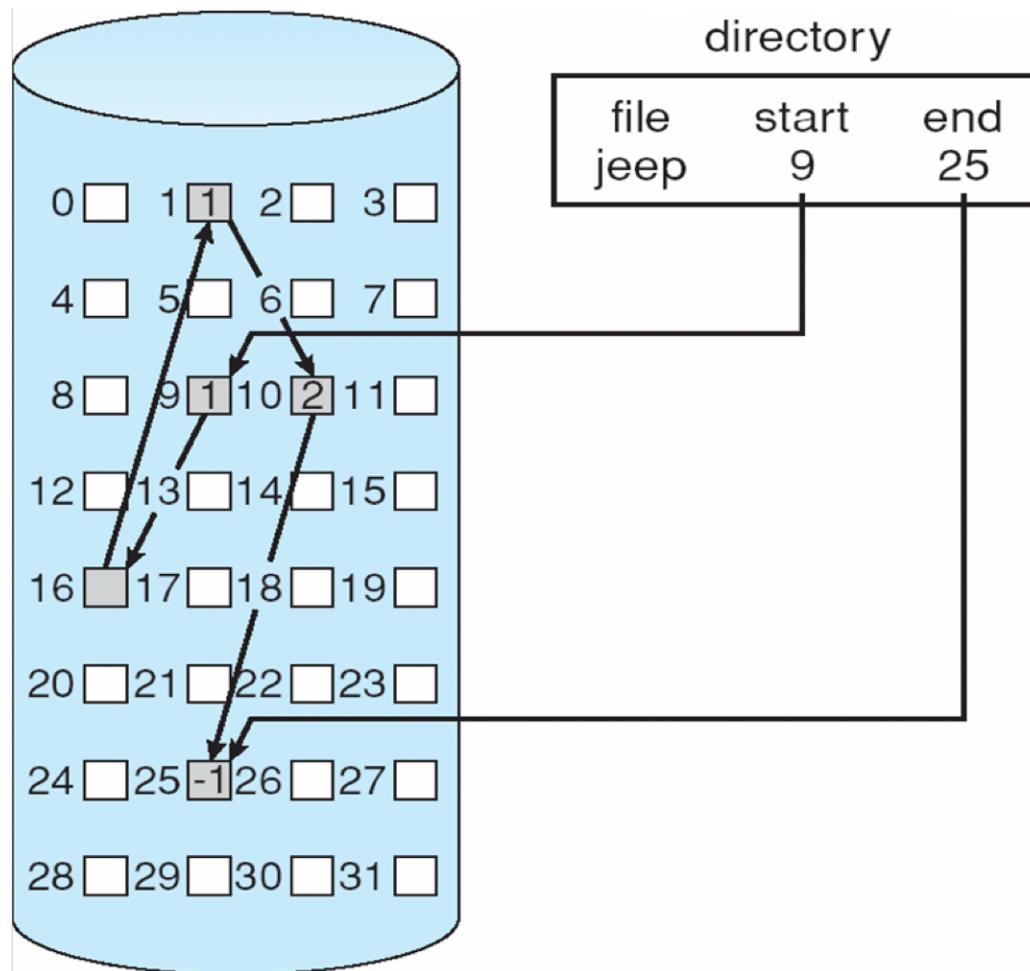
Contiguous Allocation

- Can be difficult to find free space
 - Best Fit, First Fit, etc.
- External fragmentation
 - With a big disk, perhaps we don't care
 - Compaction/defrag
 - Expensive but doable
- Difficult to have files grow
 - Copies to bigger holes under the hood?
 - High overhead
 - Ask users to specify maximum file sizes?
 - Inconvenient
 - High internal fragmentation
 - Create a linked list of file chunks
 - Called an extent

Linked Allocation

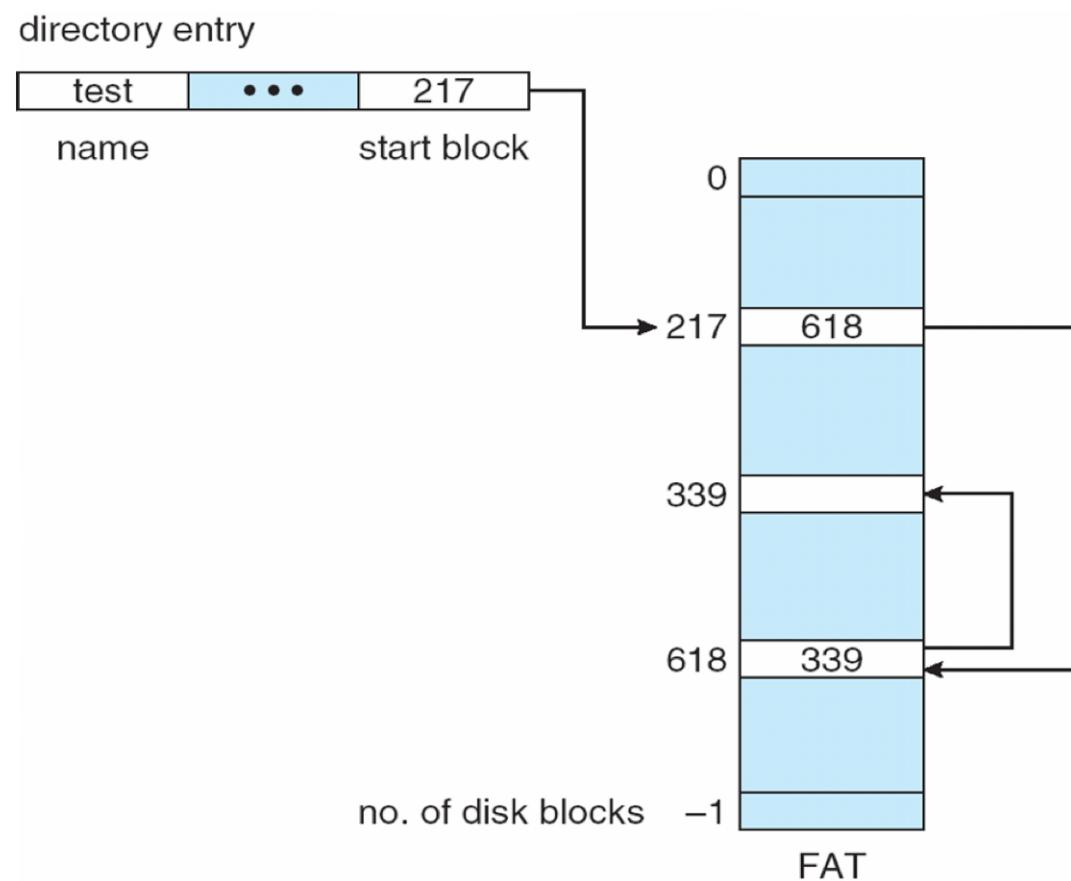
- Linked allocation: each file is a **linked list of disk blocks**
 - each block contains pointer to **next block**, file ends at nil pointer
 - blocks may be scattered anywhere on the disk (**no external fragmentation, no compaction**)
- *Disadvantages*
 - *locating a file block can take many I/Os and disk seeks*
 - *Pointer size: 4 of 512 bytes are used for pointer - 0.78% space is wasted*
 - *Reliability: what about the pointer has corrupted!*
- *Improvements: cluster the blocks - like 4 blocks*
 - *however, has internal fragmentation*

Linked Allocation



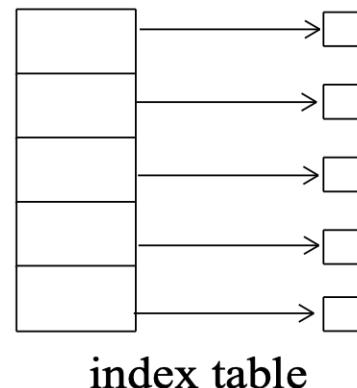
File-Allocation Table (FAT): MS-DOS

- FAT (File Allocation Table) uses linked allocation

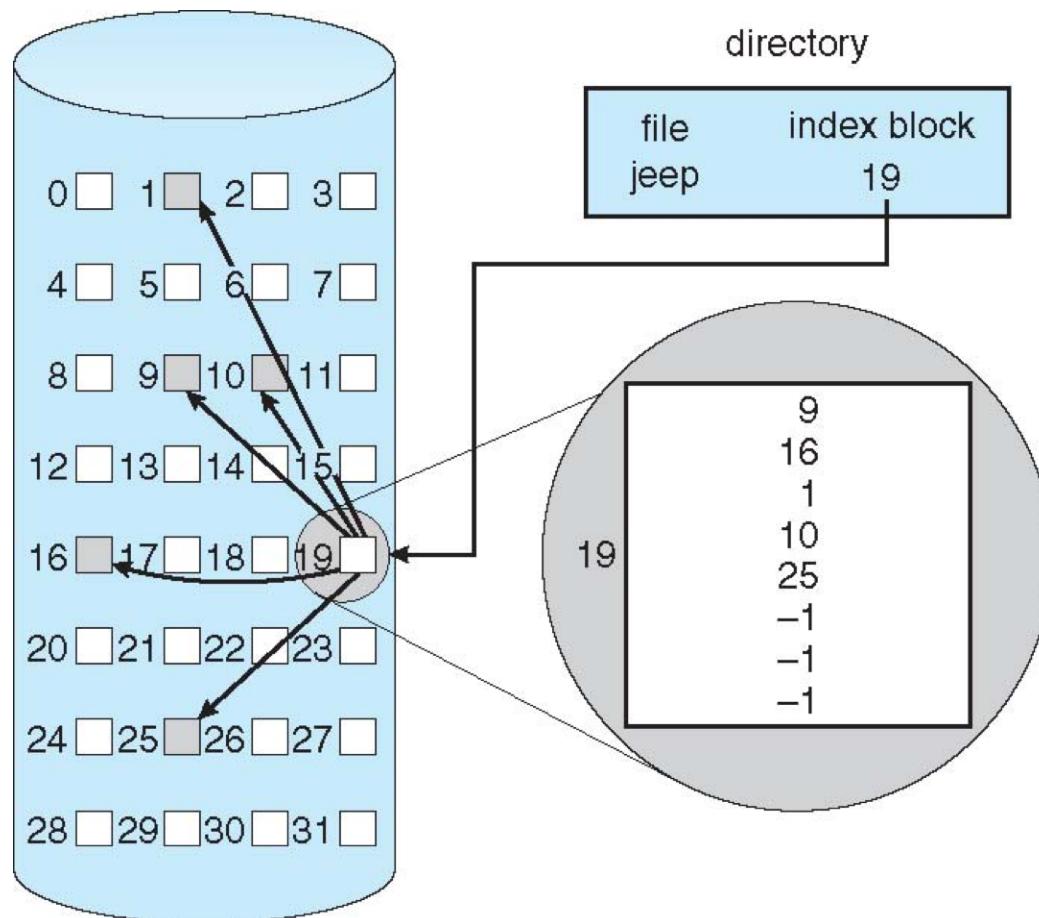


Indexed Allocation

- Indexed allocation: each file has its own **index blocks of pointers to its data blocks**
 - index table provides **random access** to file data blocks
 - no **external fragmentation**, but overhead of index blocks
 - allows **holes** in the file
 - Index block needs space - waste for small files



Indexed Allocation



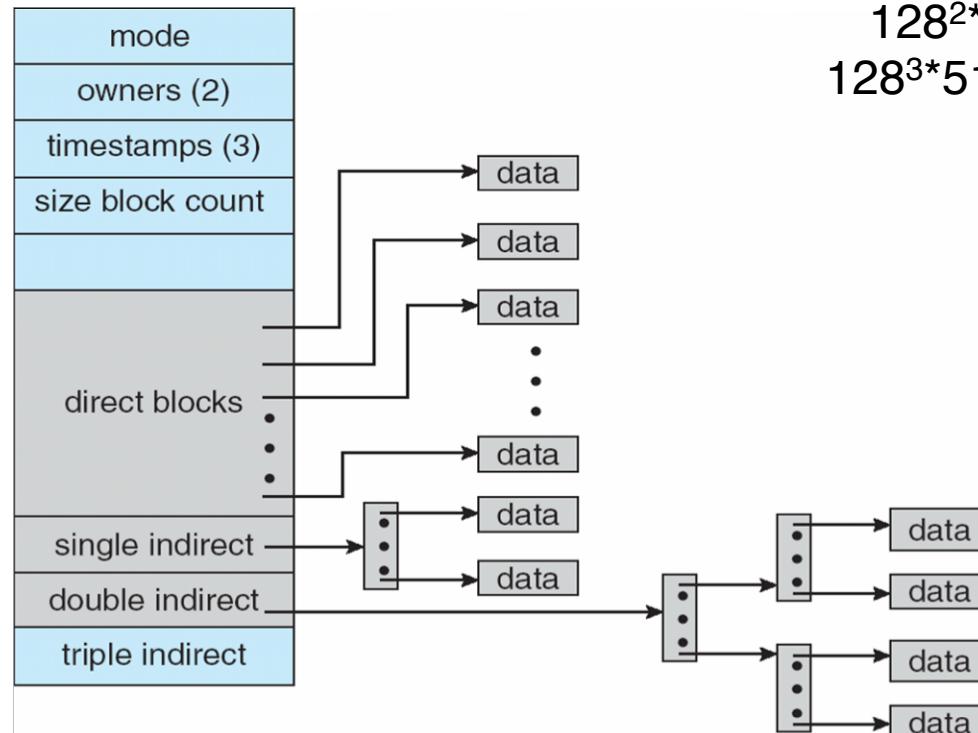
Indexed Allocation

- Need a method to allocate index blocks - cannot too big or too small
 - linked index blocks: link index blocks to support huge file
 - multiple-level index blocks (e.g., 2-level)

Indexed Allocation

- Combined scheme
- The UNIX FCB: the **inode** (**index node**)
 - First 15 pointers are in inode
 - Direct block: first 12 pointers
 - Indirect block: next 3 pointers

Block size = 512 bytes
Pointer = 4 bytes
Max file size =
 $12 \times 512 +$
 $128 \times 512 +$
 $128^2 \times 512 +$
 $128^3 \times 512$ bytes



Allocation Methods

- Best allocation method depends on file access type
 - **contiguous** is great for **sequential** and **random**
 - **linked** is good for **sequential**, not random
 - **indexed** (combined) is more complex
 - single block access may require 2 index block reads then data block read
 - clustering can help improve throughput, reduce CPU overhead
 - cluster is a set of contiguous blocks
- Disk I/O is slow, reduce as many disk I/Os as possible
 - Intel Core i7 extreme edition 990x (2011) at 3.46Ghz = 159,000 MIPS
 - 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
 - The space of deleted files should be reclaimed
- Many allocation methods:
 - bit vector or bit map
 - linked free space
 - ...

Bitmap Free-Space Management

- Use one bit for each block, track its allocation status
 - relatively easy to find contiguous blocks
 - 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 256 MB)

if clusters of 4 blocks \rightarrow 64MB of memory

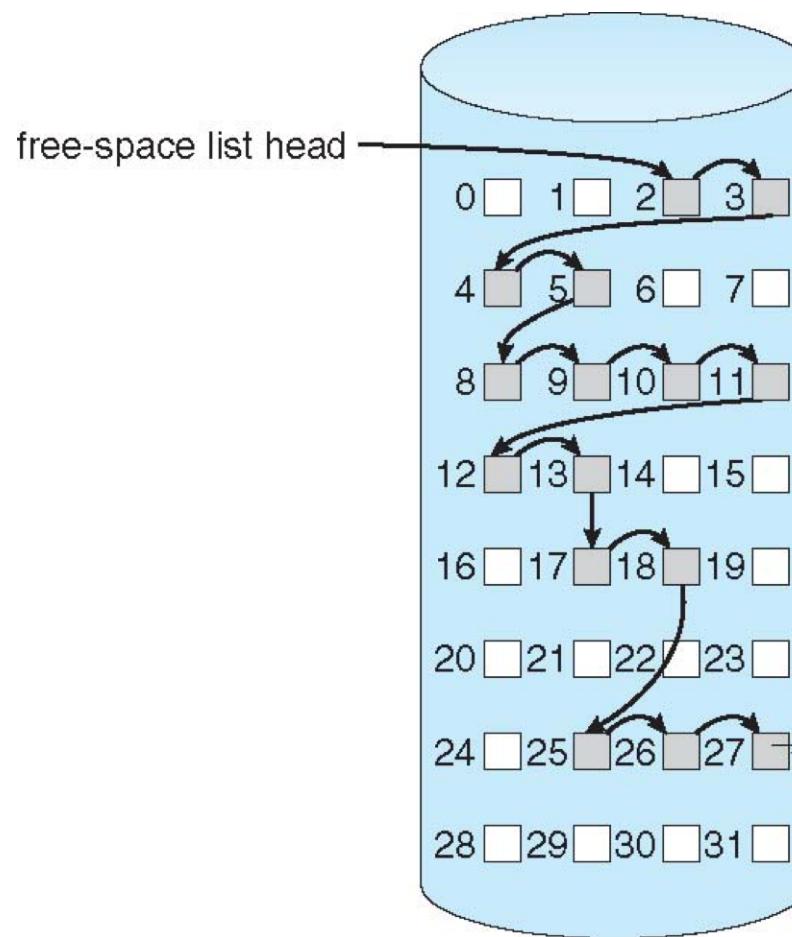


$$\text{bit}[i] = \begin{cases} 1 & \rightarrow \text{block}[i] \text{ free} \\ 0 & \rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

Linked Free Space

- Keep free blocks in linked list
 - no waste of space, just use the memory in the free block for pointers
 - cannot get contiguous space easily
 - Usually no need to traverse the entire list: return the first one

Linked Free Space



Grouping and Counting

- Simple linked list of free-space is inefficient
 - one extra disk I/O to allocate one free block (disk I/O is extremely slow)
 - allocating **multiple free blocks** require traverse the list
 - difficult to allocate contiguous free blocks
- **Grouping:** use indexes to group free blocks
 - store address of $n-1$ free blocks in the **first free block**, plus a pointer to the next **index block**
 - allocating **multiple free blocks does not need to traverse the list**
- **Counting:** a link of clusters (starting block + # of contiguous blocks)
 - space is frequently contiguously used and freed
 - in link node, keep address of first free block and # of following free blocks

File System Performance

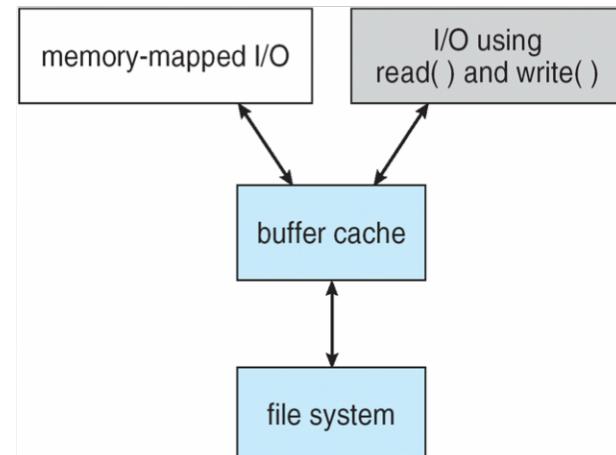
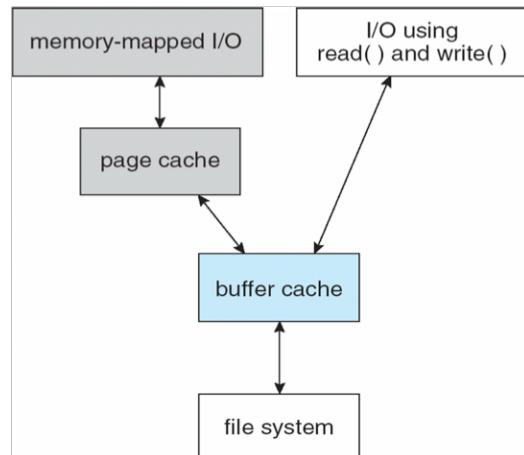
- File system efficiency and performance dependent on:
 - disk allocation and directory algorithms
 - types of data kept in file's directory entry
 - pre-allocation or as-needed allocation of metadata structures
 - fixed-size or varying-size data structures

File System Performance

- To improve file system performance:
 - **keeping data and metadata close together**
 - use cache: separate section of main memory for frequently used blocks
 - use **asynchronous writes**, it can be buffered/cached, thus faster
 - cannot cache synchronous write, writes must hit disk before return
 - synchronous writes sometimes requested by apps or needed by OS
 - **free-behind and read-ahead**: techniques to optimize sequential access - remove the previous page from the buffer, read multiple pages ahead
 - **Reads frequently slower than write**: really?

Page Cache

- OS has different levels of cache:
 - a **page cache** caches pages for MMIO, such as memory mapped files
 - file systems uses **buffer (disk) cache** for disk I/O
 - memory mapped I/O may be cached twice in the system
- A unified buffer cache uses the same page cache to cache both memory-mapped pages and disk I/O to avoid double caching



Recovery

- File system needs consistency checking to ensure consistency
 - compares data in directory with some metadata on disk for consistency
 - fs recovery can be slow and sometimes fails
- File system recovery methods
 - backup
 - log-structured file system

Log Structured File Systems

- In LSFS, metadata for updates sequentially written to a **circular log**
 - once changes written to the log, it is committed, and syscall can return
 - log can be located on the other disk/partition
 - meanwhile, log entries are replayed on the file system to actually update it
 - when a transaction is replayed, it is removed from the log
 - a log is circular, but un-committed entries will not be overwritten
 - garbage collection can reclaim/compact log entries
 - upon system crash, only need to replay transactions existing in the log

Takeaway

- File system layers
- File system implementation
 - On-disk structure, in-memory structure
 - inode
- File creation(), open()
- VFS
- Directory Implementation
- Allocation Methods
 - Contiguous, linked, indexed
- Free-Space Management