

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

Outline

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

File System History

- **Early File Systems (1950s-1970s)**
 - Computers were used for scientific research and large businesses
 - Early file systems were quite basic, often custom-designed for specific operating systems or even specific computer models
 - For instance, IBM's early mainframes had their own file systems



IBM 350 disk storage unit

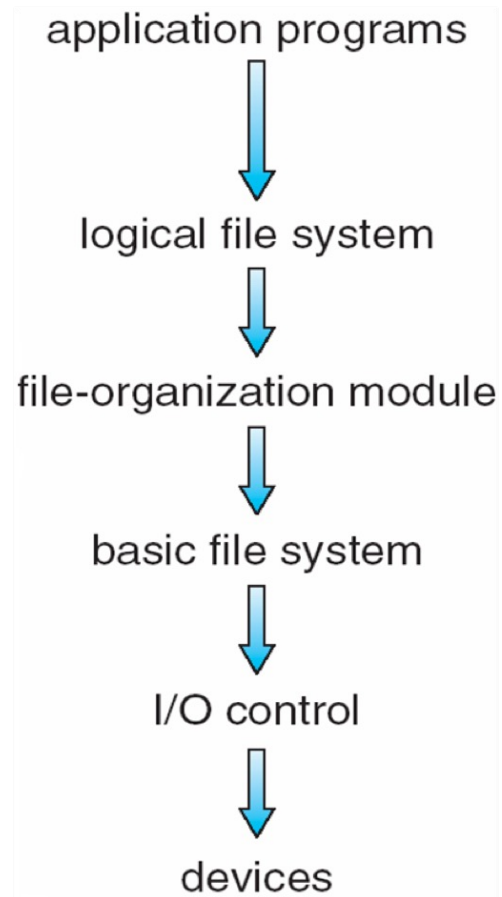
File System History

- **Hierarchical File Systems (1970s-1980s):**
 - Allowed files to be organized in directories
 - Examples include UNIX's file system and Microsoft's FAT (File Allocation Table) system
- **Network File Systems (1980s-1990s):**
 - File systems that supported access over networks emerged
 - such as NFS (Network File System).
- **Journaling File Systems (1990s-2000s):**
 - Journaling file systems (like ext3, ext4, NTFS) appeared
 - Log changes before operations, allowing recovery after system crashes, improving performance and reliability
- **Modern File Systems (2000s-Present):**
 - File systems like ZFS and Btrfs introduced advanced features
 - Snapshots, dynamic volume management, and data integrity checks.
 - Distributed file systems (like Google's GFS and Amazon's S3) became more prevalent, designed for large-scale data storage and high reliability

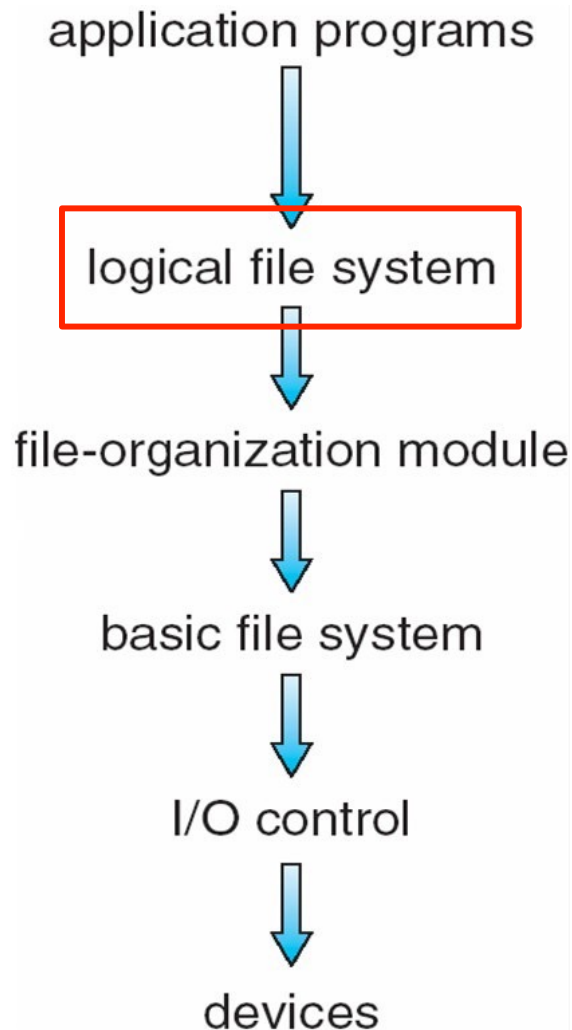
File-System Structure

- 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

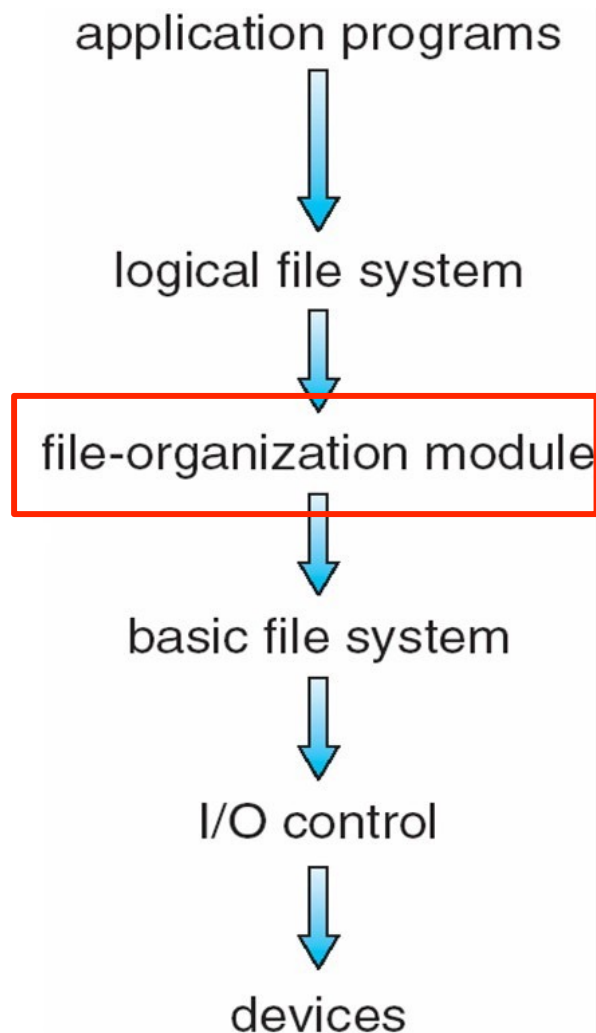


File System Implementation



- 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 FCBs
 - Pointers to data blocks on disk
- Input from above:
 - Open/Read/Write filepath
- Output to below:
 - Read/Write logical blocks

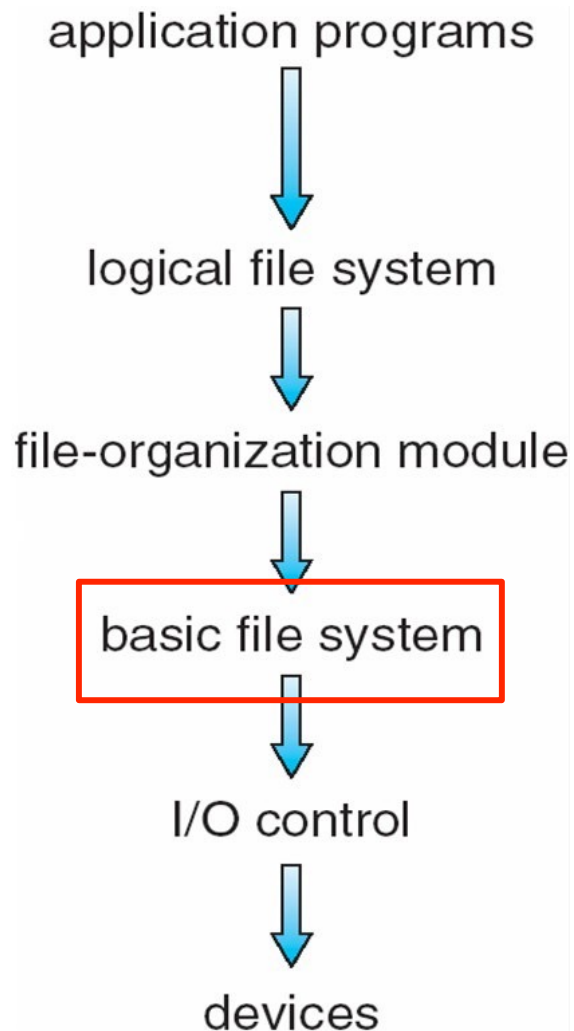
File System Implementation



- 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

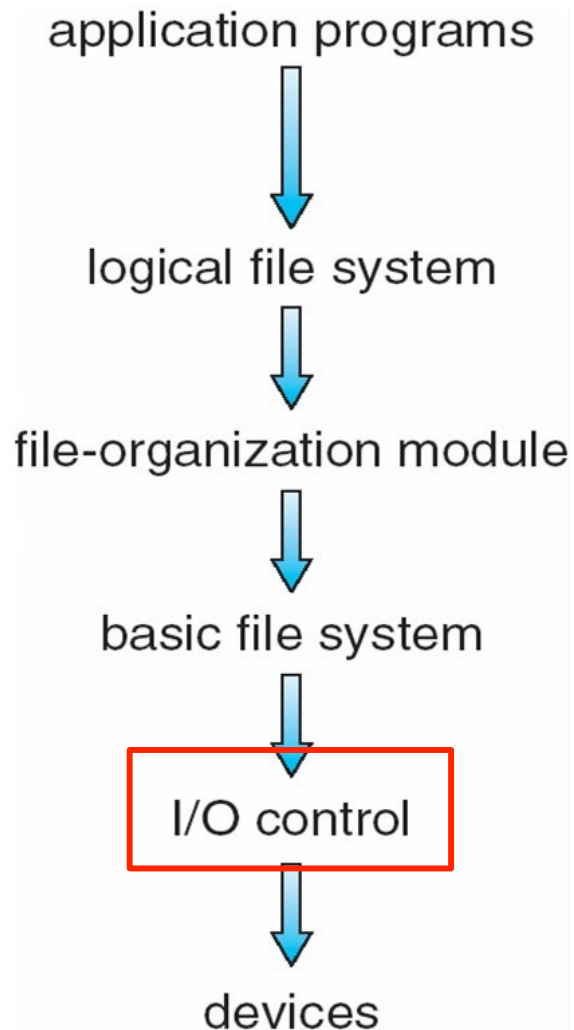
□

File System Implementation



- Basic file system
- “block I/O subsystem” in Linux
- 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

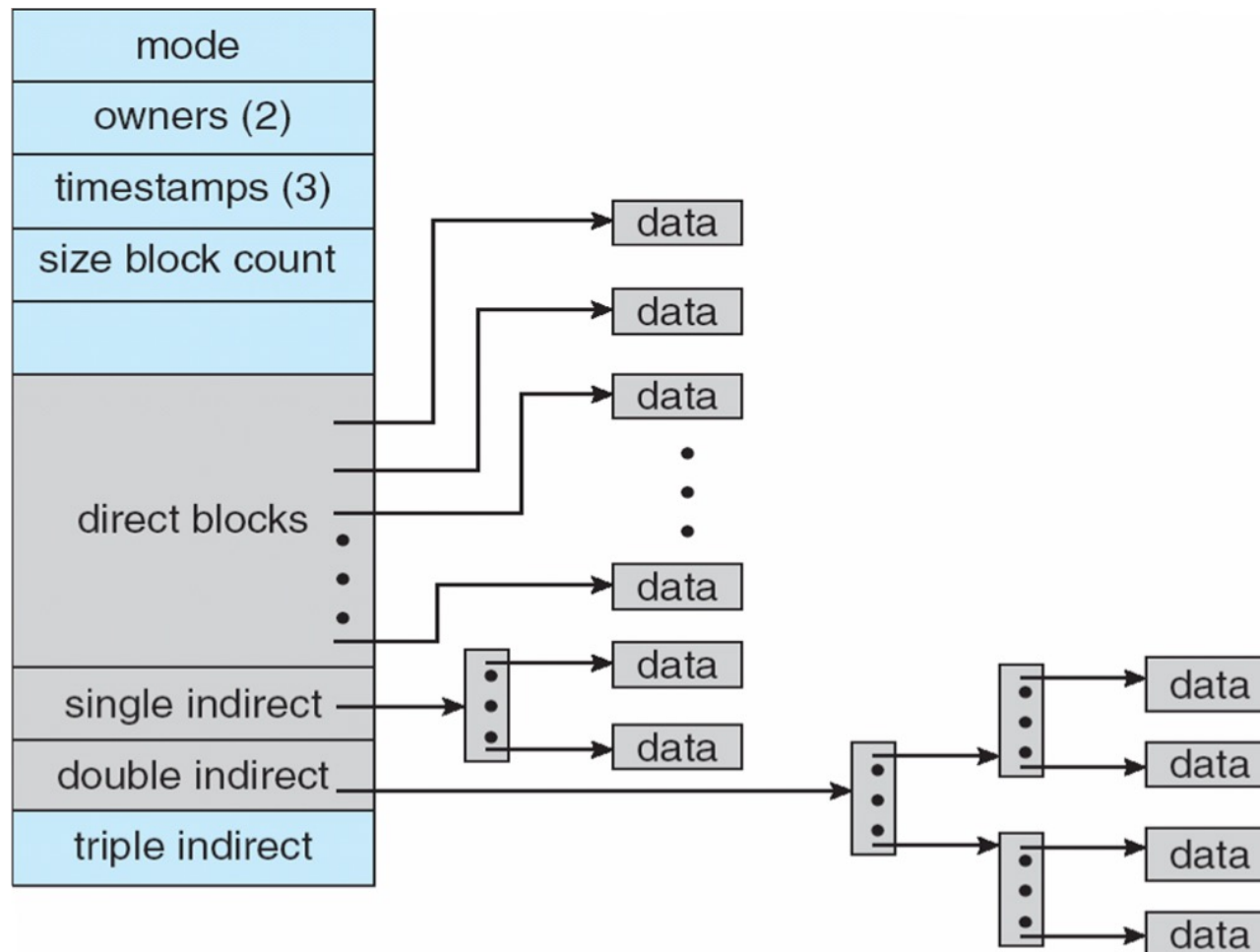
- The file system comprises data structures
- On-disk structures:
 - An optional **boot control block**
 - First block of a volume that stores an OS
 - boot block in UFS, partition boot sector in NTFS
 - A **volume control block**
 - Contains the number of blocks in the volume, block size, free-block 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)

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 Control Block

- ext2_inode

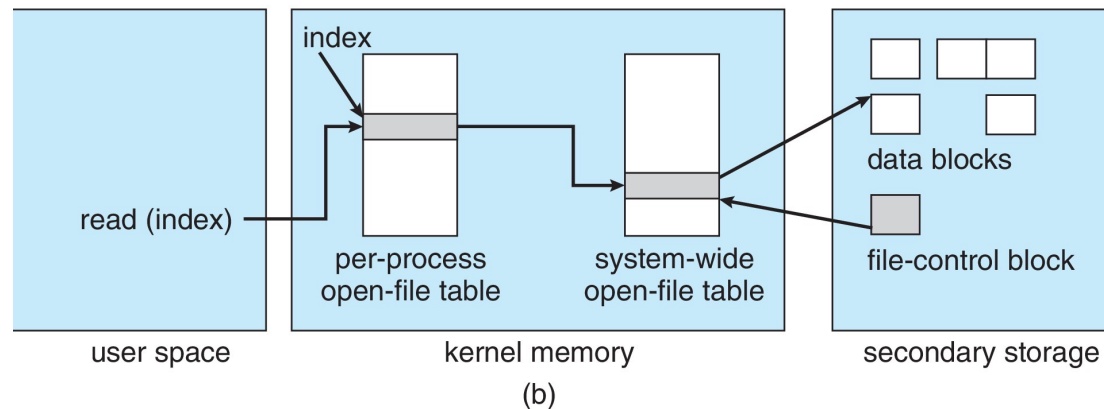
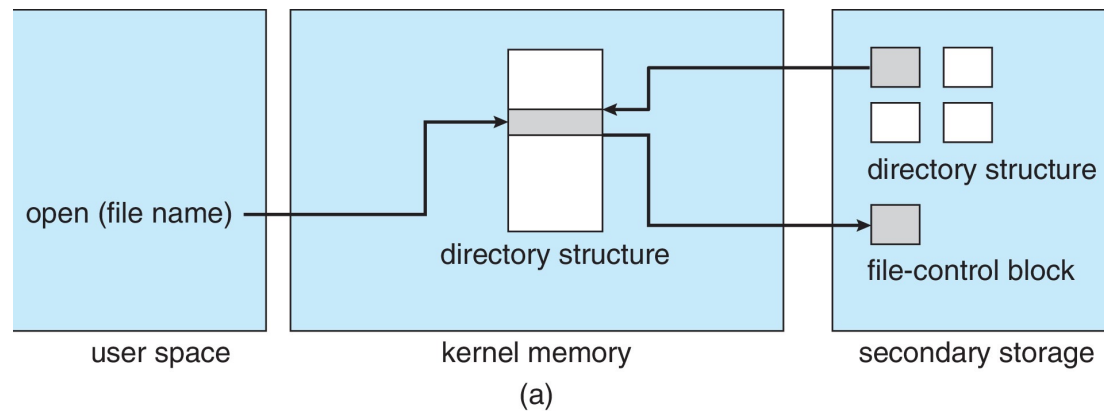


File Creation

- 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

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, load the **FCB** from disk to memory and place it in the system-wide open-file table



Operations - open()

- Make an entry 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
- Increment the open count in the system-wide open-file table
- Returns a pointer to the 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

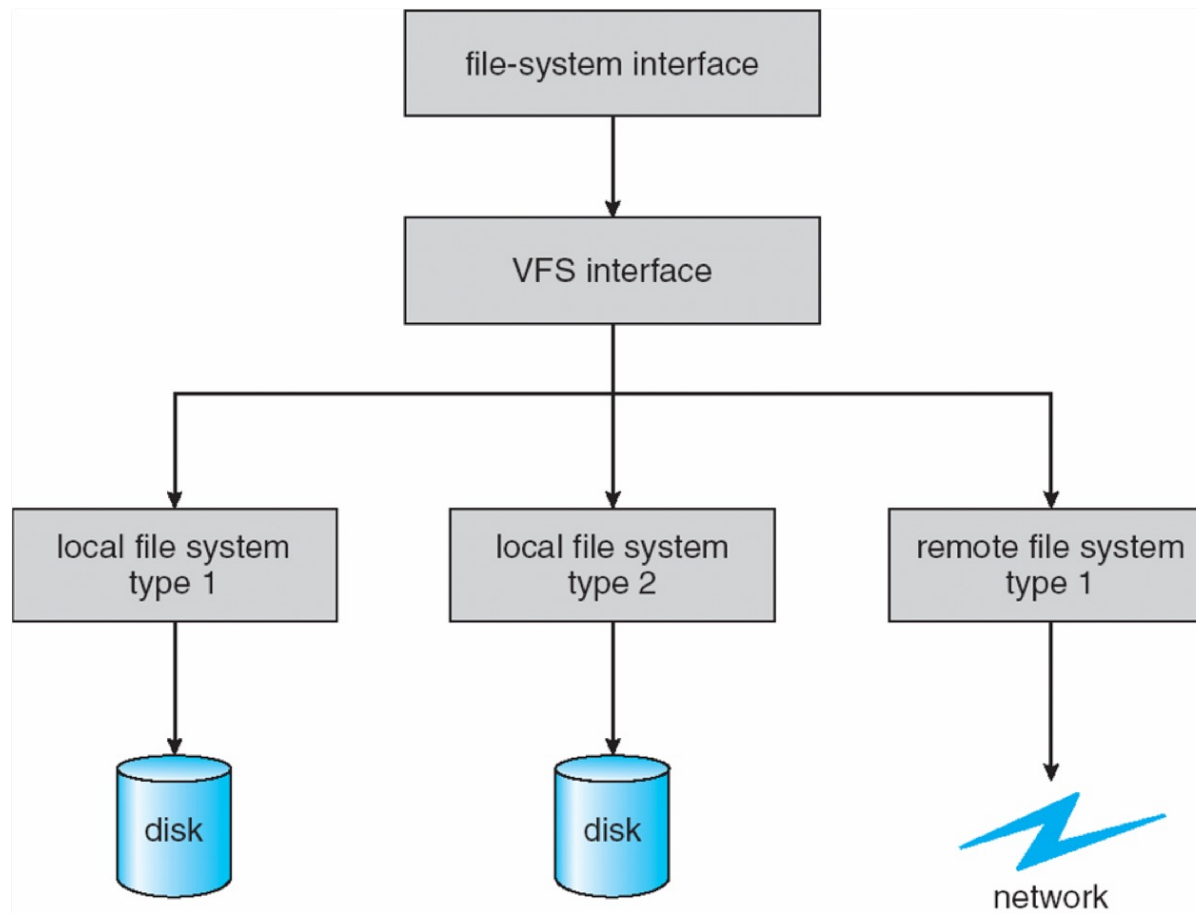
Virtual File Systems

- Questions: how does OS handle different file systems?
- “All problems in computer science can be solved by another level of indirection,
 - By David Wheeler

Virtual File Systems

- Questions: how does OS handle different file systems?
- “All problems in computer science can be solved by another level of indirection, except for the problem of too many layers of indirection.”
 - By David Wheeler

Virtual File System



Virtual File System

- 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 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, loff_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 long);  
    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 *);  
};
```

VFS Implementation

- Write syscall → vfs_write → indirect call → ext4_file_write_iter

```
ssize_t vfs_write(struct file *file, const char __user *buf, si:
{
    ssize_t ret;

    if (!(file->f_mode & FMODE_WRITE))
        return -EBADF;
    if (!(file->f_mode & FMODE_CAN_WRITE))
        return -EINVAL;
    if (unlikely(!access_ok(buf, count)))
        return -EFAULT;

    ret = rw_verify_area(WRITE, file, pos, count);
    if (ret)
        return ret;
    if (count > MAX_RW_COUNT)
        count = MAX_RW_COUNT;
    file_start_write(file);
    if (file->f_op->write)
        ret = file->f_op->write(file, buf, count, pos);
    else if (file->f_op->write_iter)
        ret = new_sync_write(file, buf, count, pos);
    else
        ret = -EINVAL;
    if (ret > 0) {
        fsnotify_modify(file);
        add_wchar(current, ret);
    }
    inc_syscw(current);
    file_end_write(file);
    return ret;
}
```

```
const struct file_operations ext4_file_operations = {
    .llseek      = ext4_llseek,
    .read_iter   = ext4_file_read_iter,
    .write_iter  = ext4_file_write_iter,
    .iopoll      = iomap_dio_iopoll,
    .unlocked_ioctl = ext4_ioctl,
#ifdef CONFIG_COMPAT
    .compat_ioctl = ext4_compat_ioctl,
#endif
    .mmap        = ext4_file_mmap,
    .mmap_supported_flags = MAP_SYNC,
    .open        = ext4_file_open,
    .release     = ext4_release_file,
```

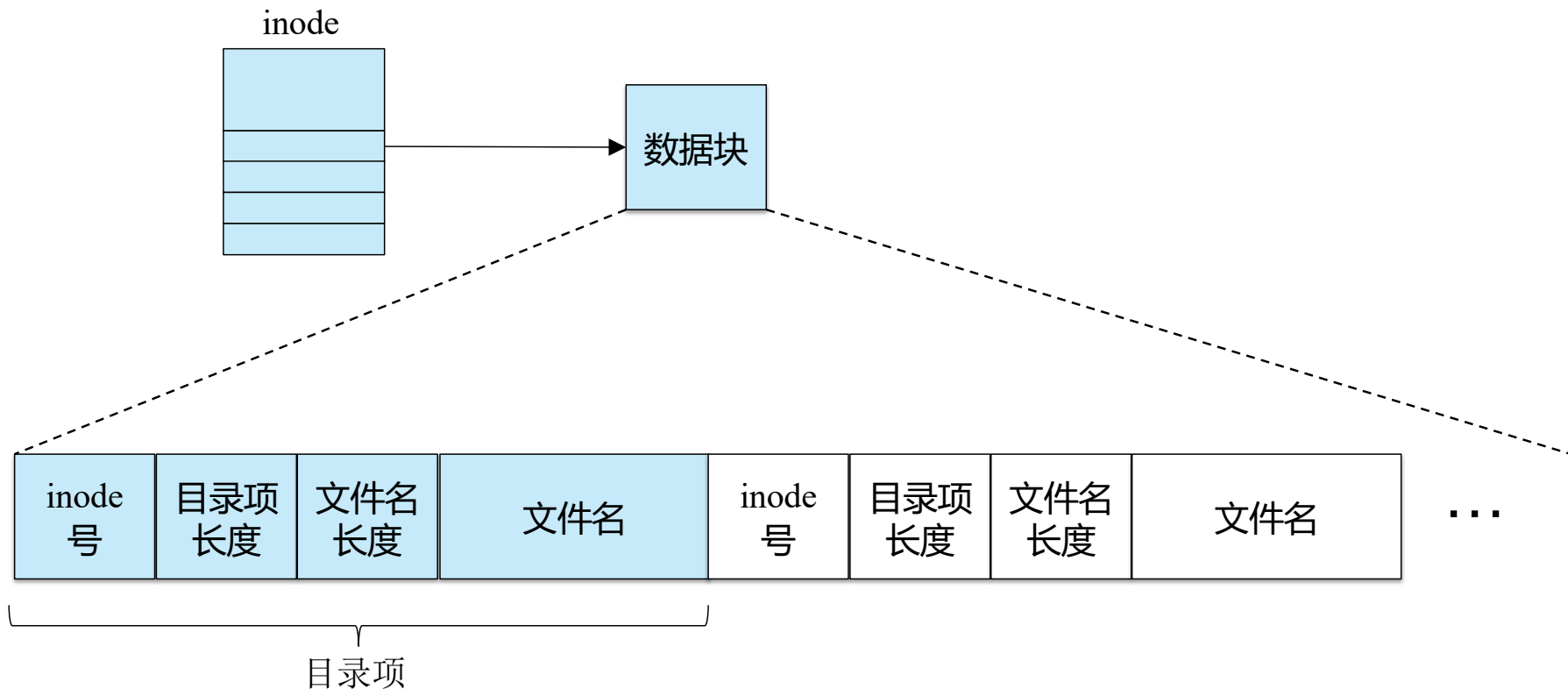
- What's the call path when writing a nfs file?
- When is file->f_op setted?

Outline

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

Directory Implementation

- Directory is a special file
 - Store the mapping from file name to inode



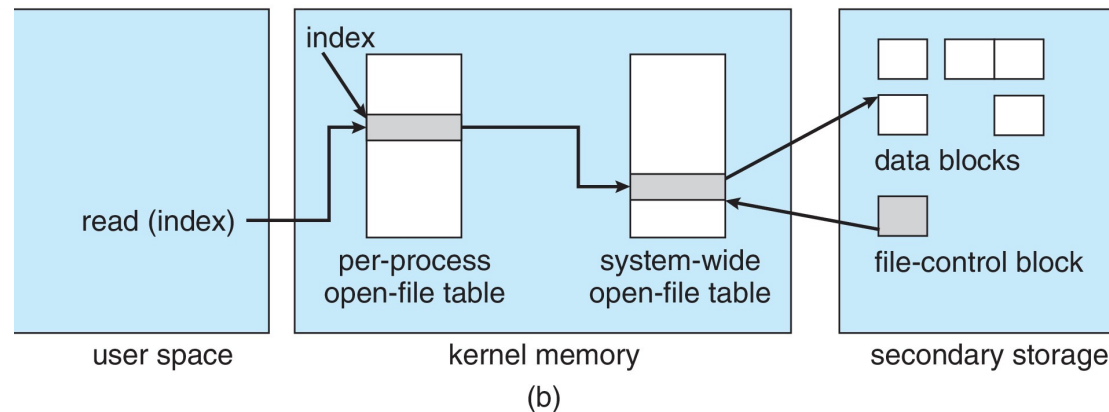
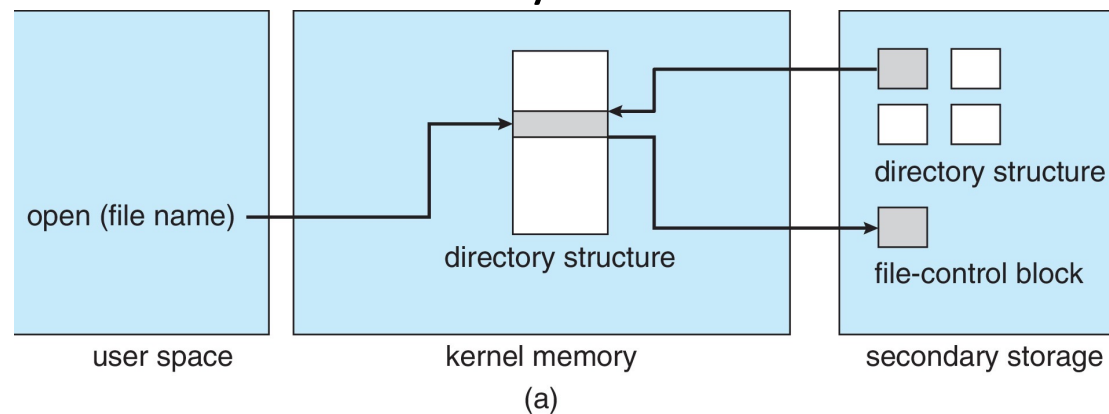
```
struct ext2_dir_entry {  
    __le32  inode;  
    __le16  rec_len;  
    __le16  name_len;  
    char    name[];  
};  
  
/* Inode number */  
/* Directory entry length */  
/* Name length */  
/* File name, up to EXT2_NAME_LEN */
```

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
- 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

Directory - file open/read revisit

- `open()` syscall searches file name in the system-wide open-file table
 - If found, a per-process open-file table entry is created
 - pointing to the existing system-wide open-file table.
 - If not found, the directory structure is searched for given file name
 - Once the file is found, the FCB is copied into a system-wide open-file table in memory.

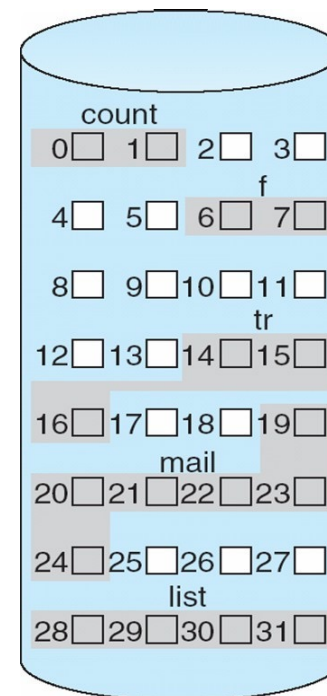


Disk Block Allocation

- **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 shorten seek times
- The directory keeps track of each file as the address of its first block and of its length in blocks



directory

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

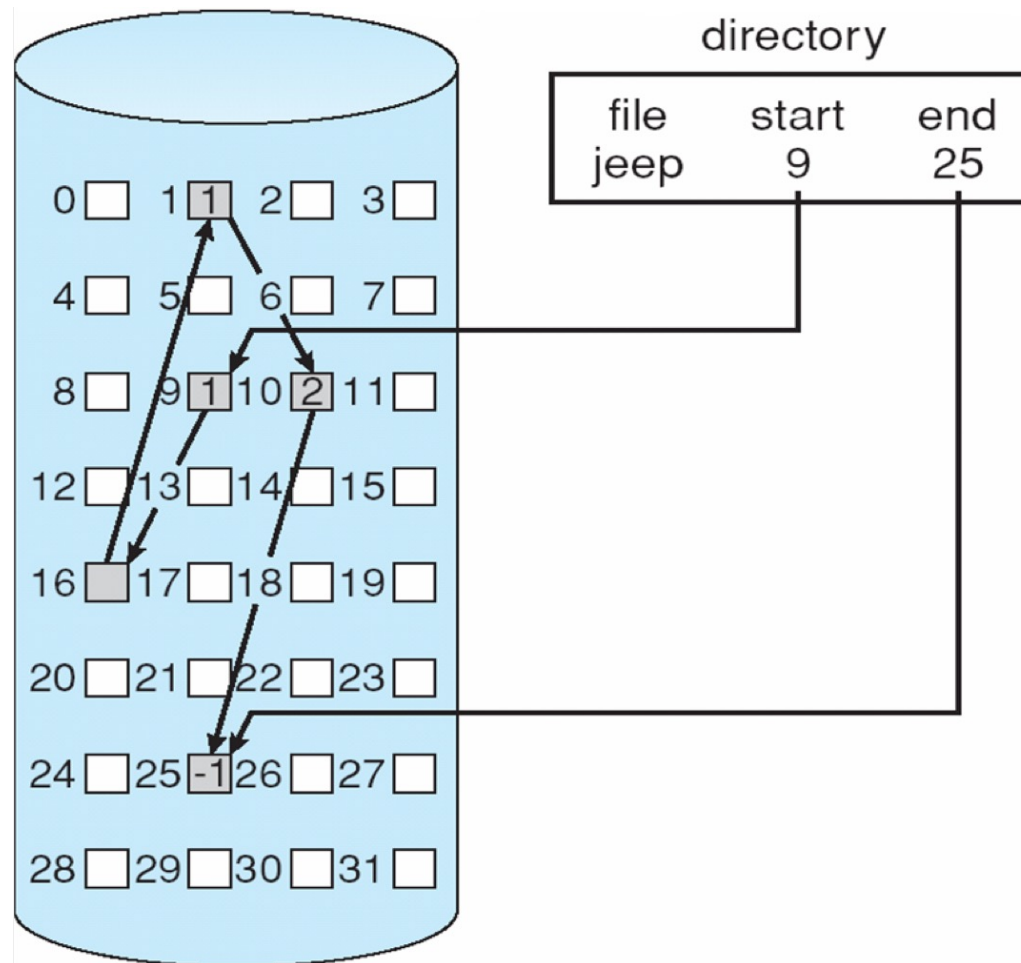
Contiguous Allocation

- 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

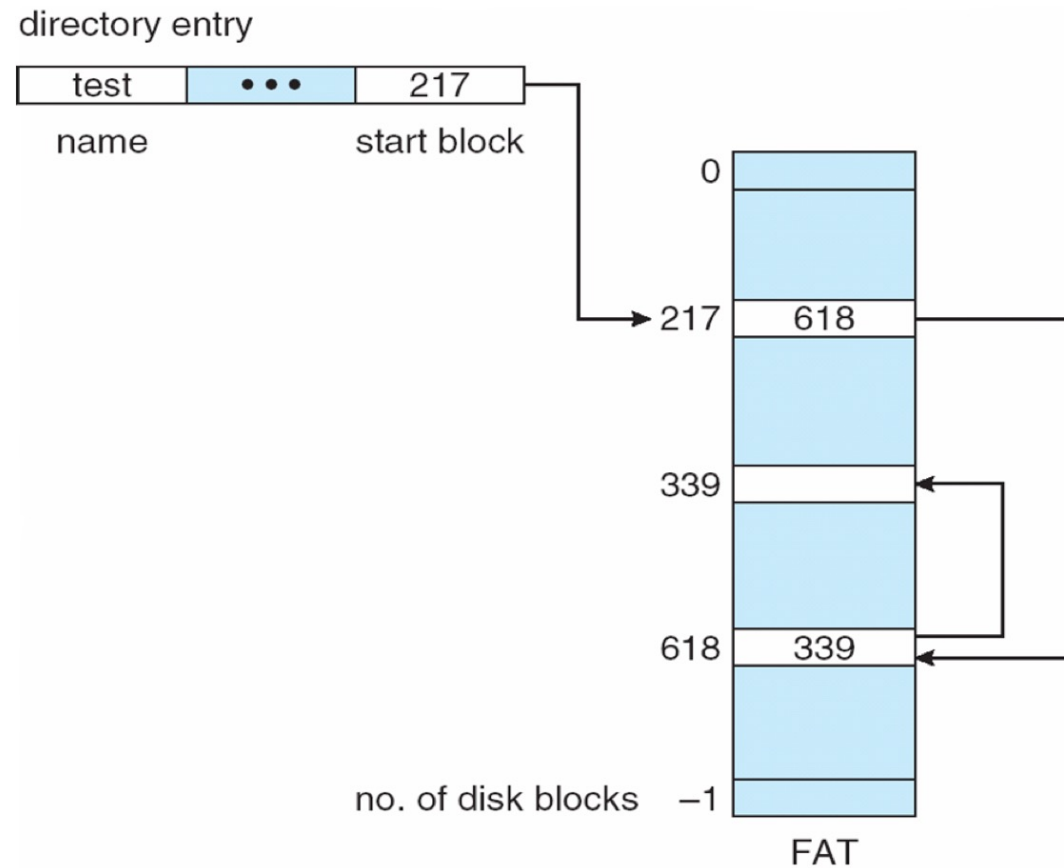
- Linked allocation: each file is a linked list of disk blocks
 - Each block contains pointer to next block, file ends at nil pointer
 - Blocks may be scattered anywhere on the disk (no external fragmentation, no compaction)
 - Disadvantages
 - Locating a file block can take many I/Os and disk seeks
 - 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



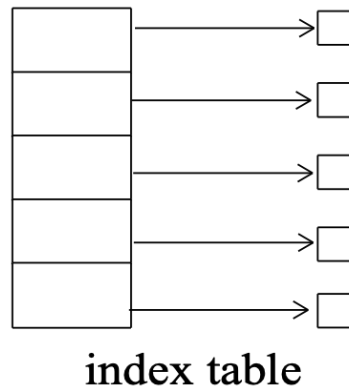
File-Allocation Table (FAT): MS-DOS

- FAT (File Allocation Table) uses linked allocation

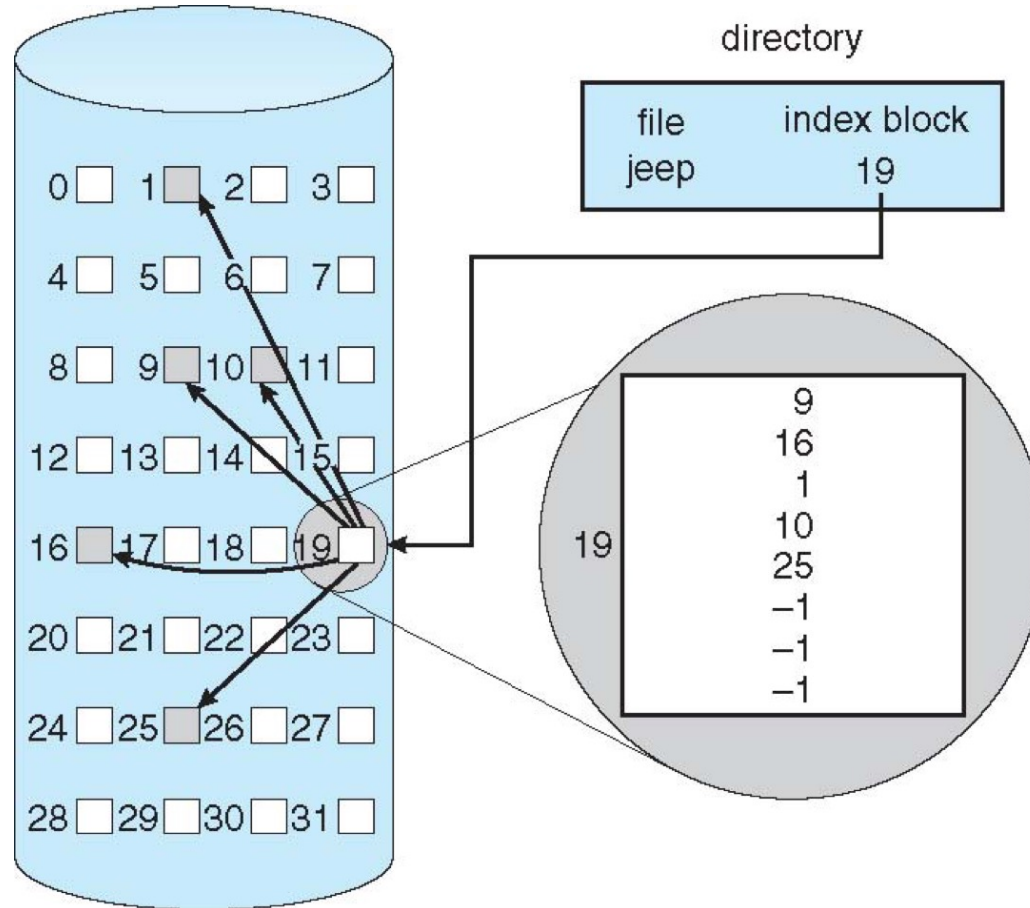


Indexed Allocation

- Indexed allocation: each file has its own **index blocks of pointers to its data blocks**
 - Index table provides **random/direct 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

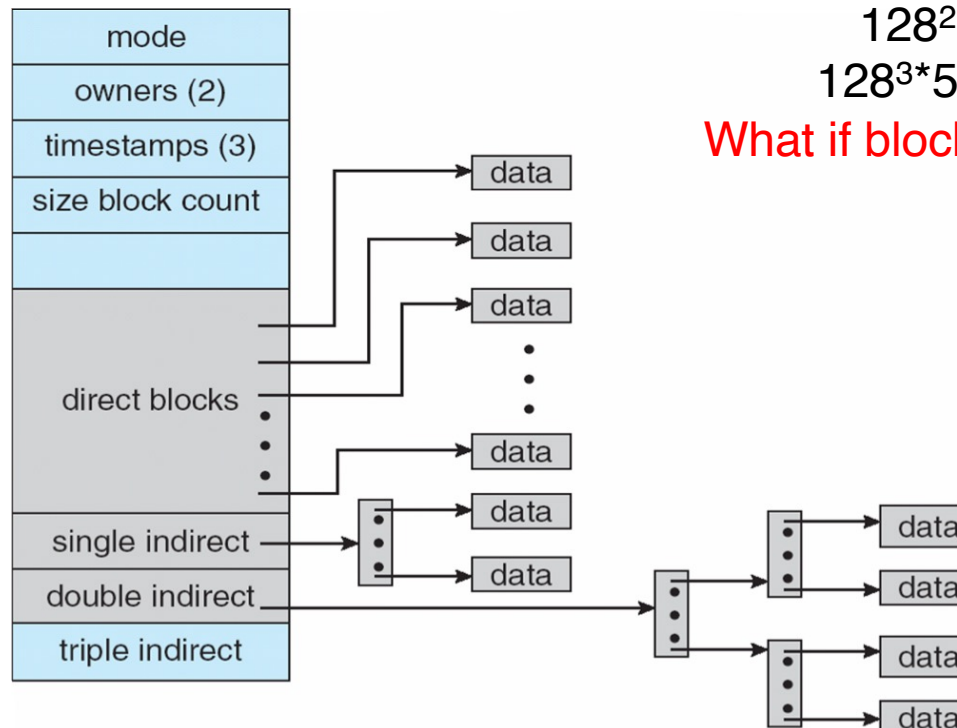


Indexed Allocation

- 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
- Check ext2_inode



Block size = 512 bytes

Pointer = 4 bytes

Max file size =

$$12 \cdot 512 + 128 \cdot 512 + 128^2 \cdot 512 + 128^3 \cdot 512 \text{ bytes}$$

What if block size is 4KB?

Allocation Methods

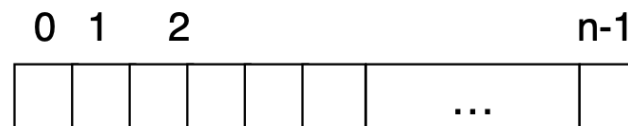
- Best allocation method depends on file access type
 - **Contiguous** is great for **sequential** and **random**
 - **Linked** is good for **sequential**, not random
 - **Indexed** (combined) is more complex
 - Single block access may require 2 index block reads then data block read
 - Clustering can 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
 - $159,000 \text{ MIPS} / 250 = 630$ million instructions during one disk I/O
 - Fast SSD drives provide 60,000 IOPS (I/Os per second)
 - $159,000 \text{ mips} / 60,000 = 2.65$ 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 M bits, 32 M bytes)
- if clusters of 4 blocks -> 64M bits of memory

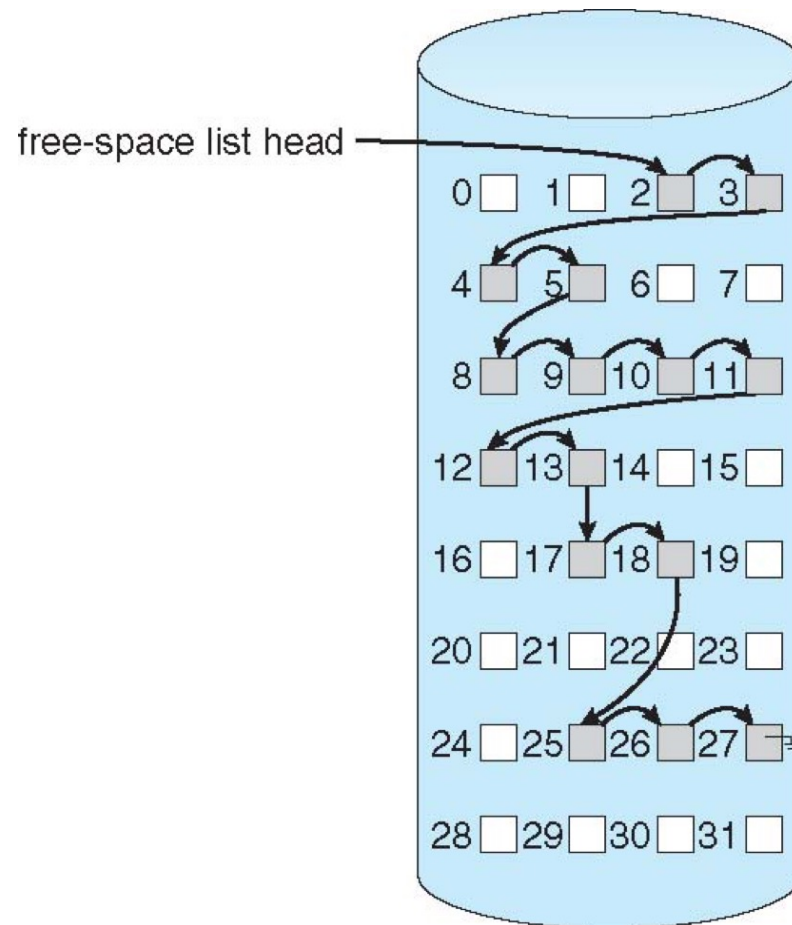


$$\text{bit}[i] = \begin{cases} 1 \rightarrow \text{block}[i] \text{ free} \\ 0 \rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

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

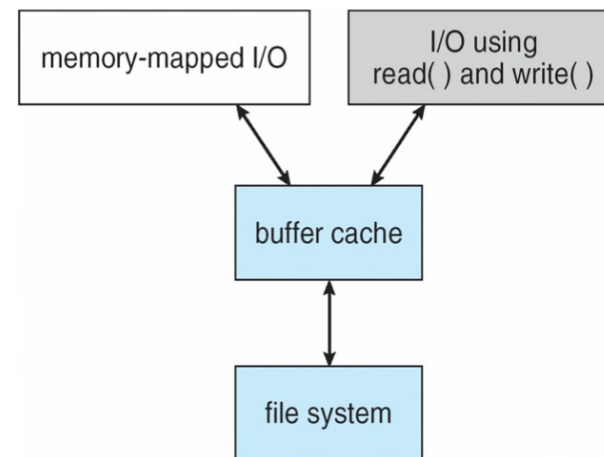
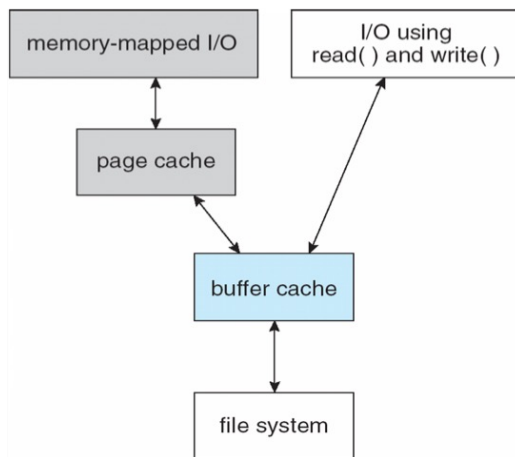
- File system efficiency and performance dependent on:
 - Disk allocation and directory algorithms
 - Types of data kept in file's directory entry
 - Pre-allocation or as-needed allocation of metadata structures
 - Fixed-size or varying-size data structures

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-replayed 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