UNIT V FILE SYSTEMS

Implementing File-system: File-System Structure, File-System Implementation, Directory Implementation, Allocation methods, Free-space management.

Case Study- Real Time operating system and Mobile operating system.

File-System - File Concept

- Contiguous logical address space
- Types:
  - Data
    - numeric
    - character
    - binary
  - Program
- Contents defined by file's creator
  - Many types
    - Consider **text file, source file, executable file**

File Attributes

- **Name –** only information kept in human-readable form
- **Identifier –** unique tag (number) identifies file within file system
- **Type –** needed for systems that support different types
- **Location –** pointer to file location on device
- **Size –** current file size
- **Protection –** controls who can do reading, writing, executing
- **Time, date, and user identification –** data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk
- Many variations, including extended file attributes such as file checksum
- Information kept in the directory structure

File Operations

- File is an **abstract data type**
- **Create**
- **Write –** at **write pointer** location
- **Read –** at **read pointer** location
- **Reposition within file - seek**
- **Delete**
- **Truncate**
- ***Open(F_i)*** **–** search the directory structure on disk for entry F_i, and move the content of entry to memory
- ***Close (F_i)*** **–** move the content of entry F_i in memory to directory structure on disk

Open Files

- Several pieces of data are needed to manage open files:
  - **Open-file table**: tracks open files
  - File pointer: pointer to last read/write location, per process that has the file open
  - **File-open count**: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it

- Disk location of the file: cache of data access information
- Access rights: per-process access mode information

Open File Locking

- Provided by some operating systems and file systems
  - Similar to reader-writer locks
  - **Shared lock** similar to reader lock – several processes can acquire concurrently
  - **Exclusive lock** similar to writer lock
- Mediates access to a file
- Mandatory or advisory:
  - **Mandatory** – access is denied depending on locks held and requested
  - **Advisory** – processes can find status of locks and decide what to do

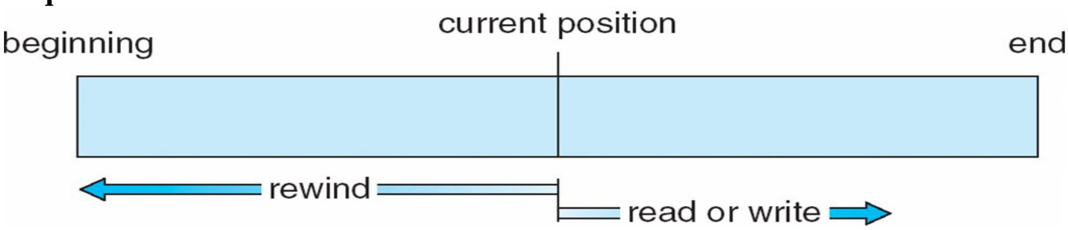
File Types – Name, Extension

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

File Structure

- None - sequence of words, bytes
- Simple record structure
  - Lines
  - Fixed length
  - Variable length
- Complex Structures
  - Formatted document
  - Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters
- Who decides:
  - Operating system
  - Program

Sequential-access File



Access Methods

- **Sequential Access**
  - **read next**
  - **write next**
  - **reset**
  - no read after last write
    - (rewrite)
- **Direct Access –** file is fixed length logical records
  - **read *n***
  - **write *n***
  - **position to *n***
    - **read next**
    - **write next**
  - **rewrite *n***

n = relative block number

- Relative block numbers allow OS to decide where file should be placed
  - o See allocation problem in Ch 12

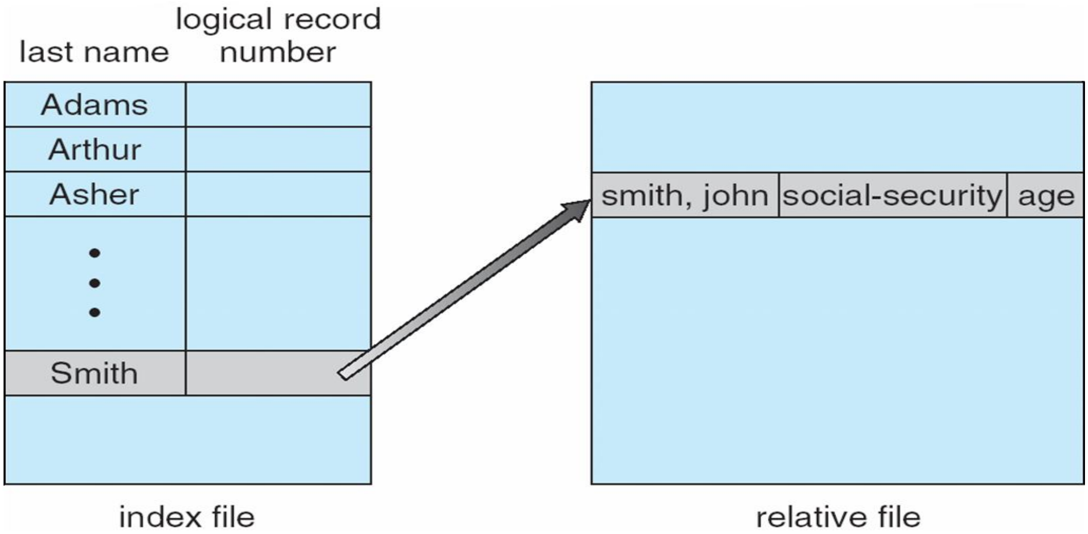
Simulation of Sequential Access on Direct-access File

sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Other Access Methods

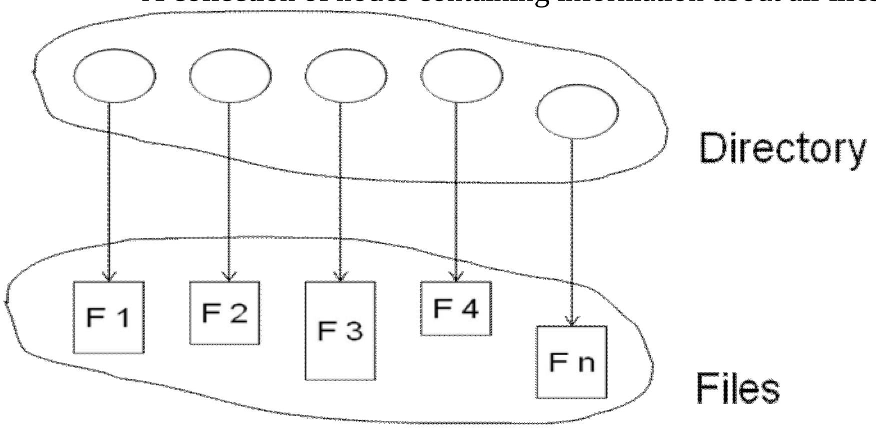
- Can be built on top of base methods
- General involve creation of an index for the file
- Keep index in memory for fast determination of location of data to be operated on (consider UPC code plus record of data about that item)
- If too large, index (in memory) of the index (on disk)
- IBM indexed sequential-access method (ISAM)
  - o Small master index, points to disk blocks of secondary index
  - o File kept sorted on a defined key
  - o All done by the OS
- VMS operating system provides index and relative files as another example (see next slide)

Example of Index and Relative Files



Directory Structure

- A collection of nodes containing information about all files

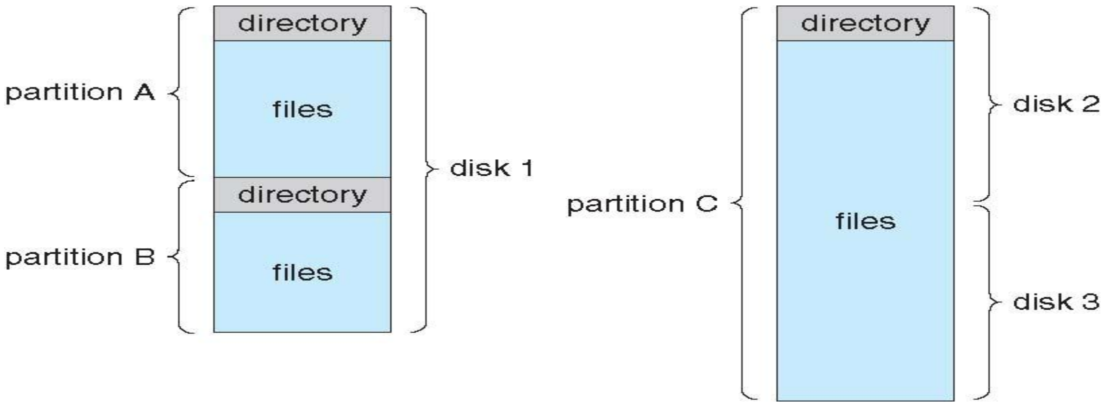


Both the directory structure and the files reside on disk

Disk Structure

- Disk can be subdivided into **partitions**
- Disks or partitions can be **RAID** protected against failure
- Disk or partition can be used **raw –** without a file system, or **formatted** with a file system
- Partitions also known as minidisks, slices
- Entity containing file system known as a **volume**
- Each volume containing file system also tracks that file system's info in **device directory** or **volume table of contents**
- As well as **general-purpose file systems** there are many **special-purpose file systems**, frequently all within the same operating system or computer

A Typical File-system Organization



Types of File Systems

- We mostly talk of general-purpose file systems
- But systems frequently have may file systems, some general- and some special- purpose
- Consider Solaris has
  - tmpfs – memory-based volatile FS for fast, temporary I/O
  - objfs – interface into kernel memory to get kernel symbols for debugging
  - ctfs – contract file system for managing daemons
  - lofs – loopback file system allows one FS to be accessed in place of another
  - procfs – kernel interface to process structures
  - ufs, zfs – general purpose file systems

Operations Performed on Directory

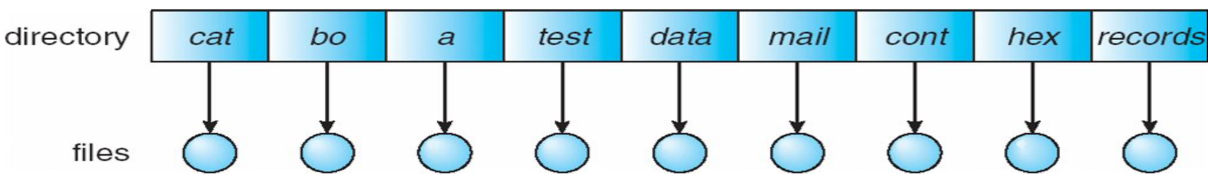
- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

Directory Organization

- Efficiency – locating a file quickly
- Naming – convenient to users
  - Two users can have same name for different files
  - The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, …)

Single-Level Directory

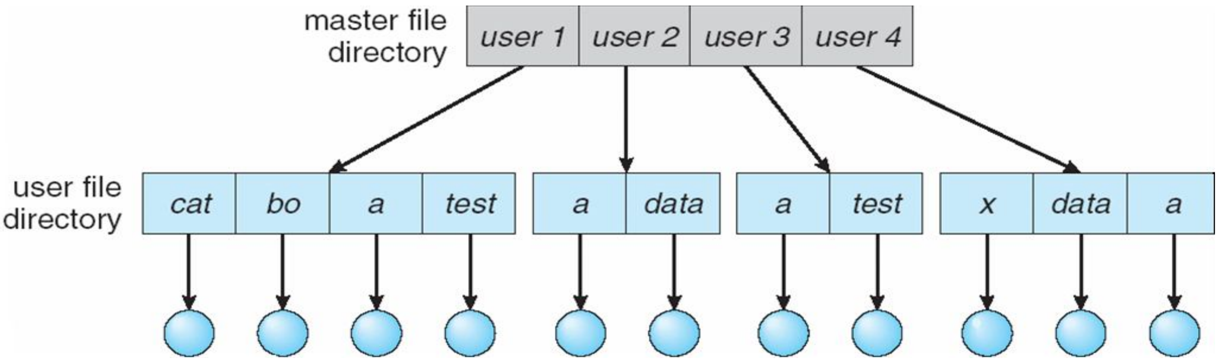
- A single directory for all users



- Naming problem
- Grouping problem

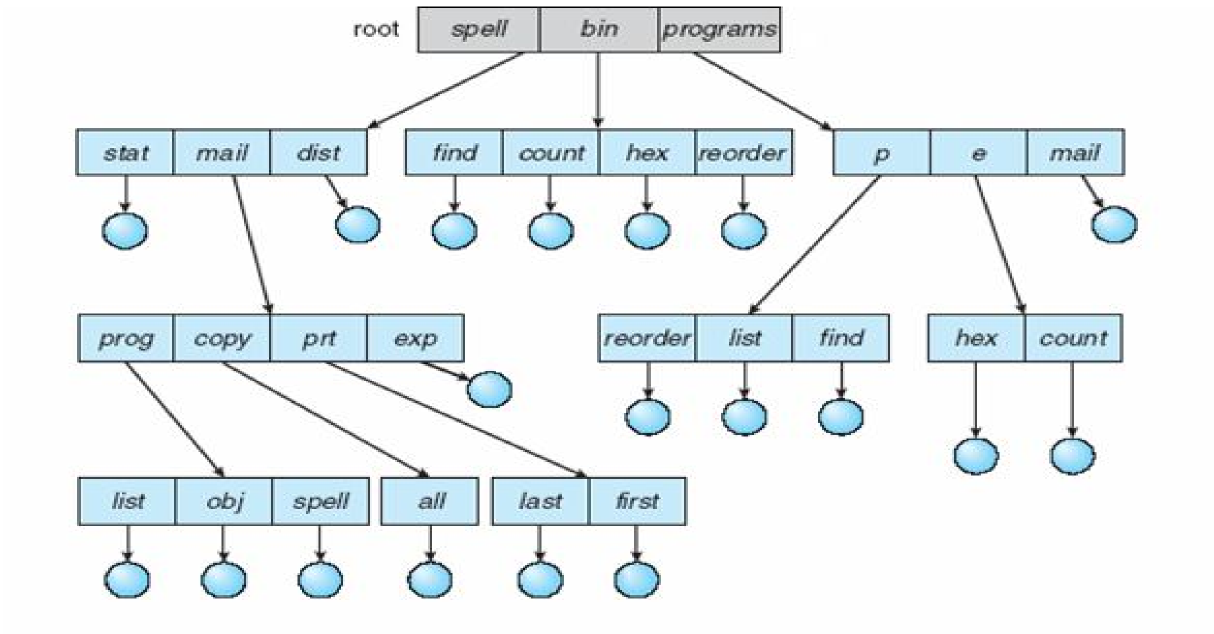
Two-Level Directory

- Separate directory for each user



- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability

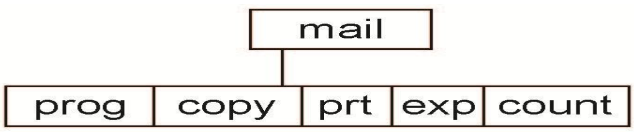
Tree-Structured Directories



- Efficient searching
- Grouping Capability
- Current directory (working directory)
    - **cd /spell/mail/prog**
    - **type list**
- **Absolute** or **relative** path name
- Creating a new file is done in current directory
- Delete a file
    - **rm <file-name>**
- Creating a new subdirectory is done in current directory
    - **mkdir <dir-name>**

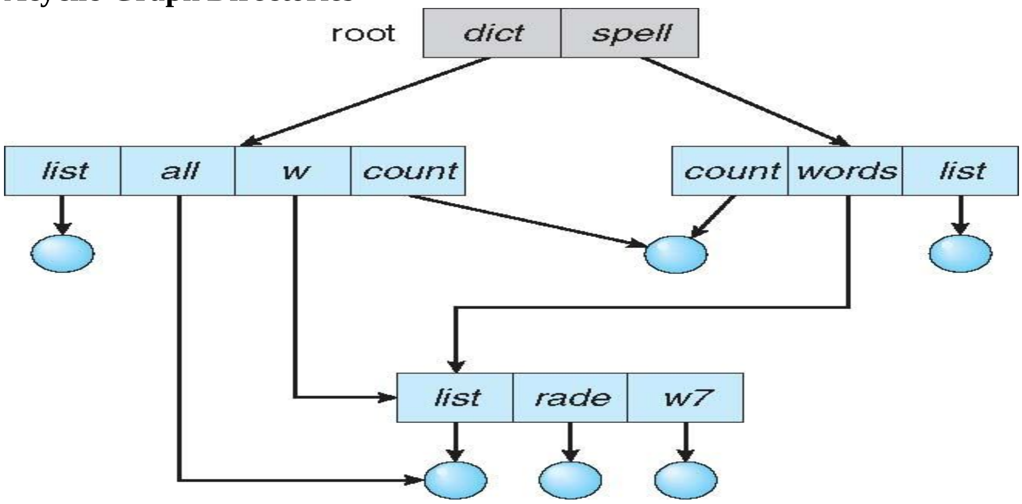
Example: if in current directory **/mail**

mkdir count



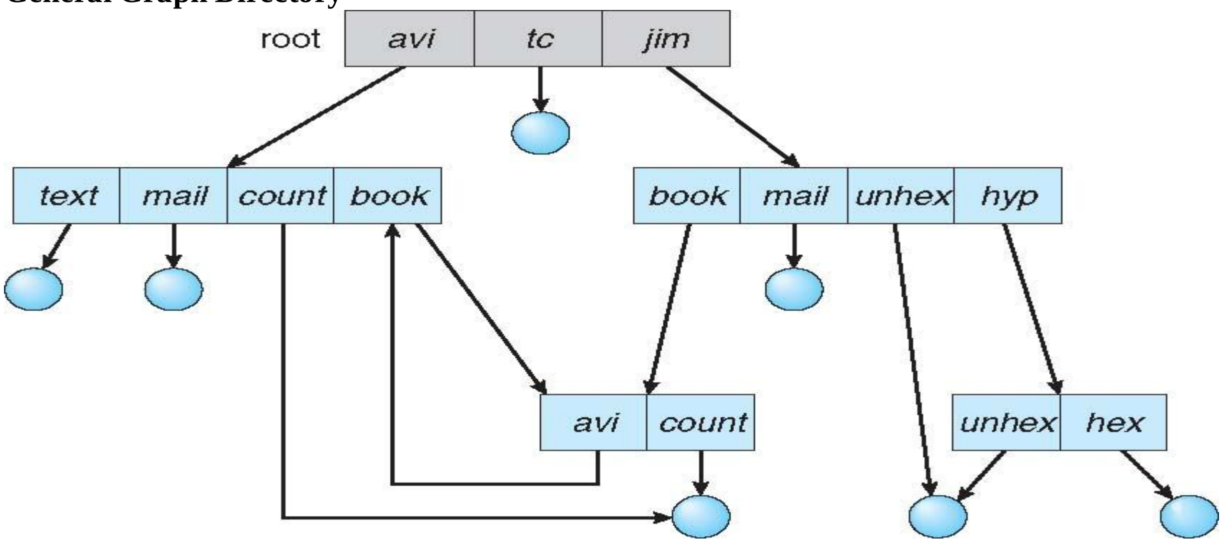
Deleting “mail” ⇒ deleting the entire subtree rooted by “mail”

Acyclic-Graph Directories



- Two different names (aliasing)
- If ***dict*** deletes ***list*** ⇒ dangling pointer
  - Solutions:
  - Backpointers, so we can delete all pointers
    Variable size records a problem
  - Backpointers using a daisy chain organization
  - Entry-hold-count solution
- New directory entry type
  - **Link** – another name (pointer) to an existing file
  - **Resolve the link** – follow pointer to locate the file

General Graph Directory

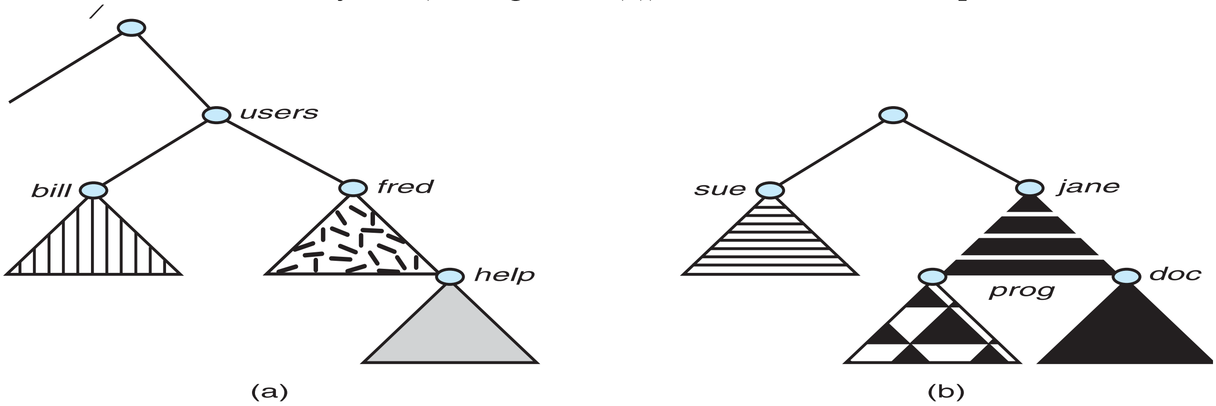


- How do we guarantee no cycles?
  - Allow only links to file not subdirectories
  - **Garbage collection**
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK

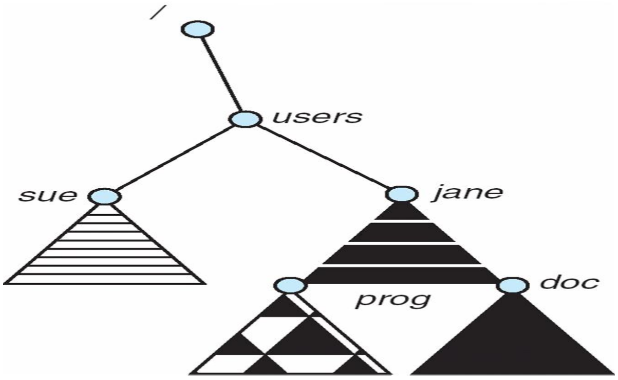
<u>File System Mounting</u>

- A file system must be **mounted** before it can be accessed

- A unmounted file system (i.e., Fig. 11-11(b)) is mounted at a **mount point**



Mount Point



File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a **protection** scheme
- On distributed systems, files may be shared across a network
- Network File System (NFS) is a common distributed file-sharing method
- If multi-user system
    - **User IDs** identify users, allowing permissions and protections to be per-user
      Group IDs allow users to be in groups, permitting group access rights
    - Owner of a file / directory
    - Group of a file / directory

File Sharing – Remote File Systems

- Uses networking to allow file system access between systems
    - Manually via programs like FTP
    - Automatically, seamlessly using **distributed file systems**
    - Semi automatically via the **world wide web**
- **Client-server** model allows clients to mount remote file systems from servers
    - Server can serve multiple clients
    - Client and user-on-client identification is insecure or complicated
    - **NFS** is standard UNIX client-server file sharing protocol
    - **CIFS** is standard Windows protocol
    - Standard operating system file calls are translated into remote calls
- Distributed Information Systems **(distributed naming services)** such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing

File Sharing – Failure Modes

- All file systems have failure modes
    - For example corruption of directory structures or other non-user data, called **metadata**
- Remote file systems add new failure modes, due to network failure, server failure
- Recovery from failure can involve **state information** about status of each remote request
- **Stateless** protocols such as NFS v3 include all information in each request, allowing easy recovery but less security

File Sharing – Consistency Semantic

- Specify how multiple users are to access a shared file simultaneously
    - Similar to process synchronization algorithms
        - Tend to be less complex due to disk I/O and network latency (for remote file systems
    - Andrew File System (AFS) implemented complex remote file sharing semantics
    - Unix file system (UFS) implements:
        - Writes to an open file visible immediately to other users of the same open file
        - Sharing file pointer to allow multiple users to read and write concurrently
    - AFS has session semantics
        - Writes only visible to sessions starting after the file is closed

Protection

- File owner/creator should be able to control:
    - what can be done
    - by whom
- Types of access
    - **Read**
    - **Write**
    - **Execute**
    - **Append**
    - **Delete**
    - **List**

Access Lists and Groups

- **Mode of access: read, write, execute**
- **Three classes of users on Unix / Linux**

			RWX
a) owner access	**7**	⇒	**1 1 1**
			RWX
b) group access	**6**	⇒	**1 1 0**
			RWX
c) public access	**1**	⇒	**1 0 0**

- **Ask manager to create a group (unique name), say G, and add some users to the group.**
- **For a particular file (say *game*) or subdirectory, define an appropriate access.**

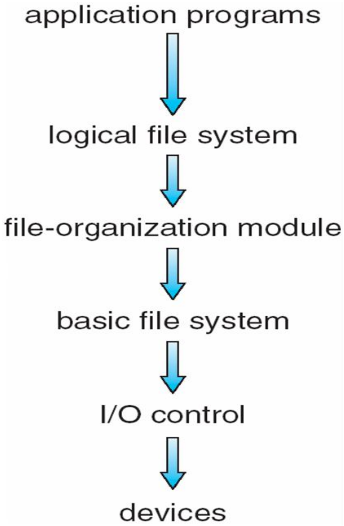
<u>File System Implementation</u>

File-System Structure

- File structure
    - Logical storage unit
    - Collection of related information
- **File system** resides on secondary storage (disks)

- Provided user interface to storage, mapping logical to physical
    - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily
- Disk provides in-place rewrite and random access
    - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- **File control block –** storage structure consisting of information about a file
- **Device driver** controls the physical device
- File system organized into layers

Layered File System



File System Layers

- **Device drivers** manage I/O devices at the I/O control layer
    - Given commands like "read drive1, cylinder 72, track 2, sector 10, into memory location 1060" outputs low-level hardware specific commands to hardware controller
- **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
- **Logical file system** manages metadata information
    - Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
    - Directory management
    - Protection
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performanceTranslates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
- Many file systems, sometimes many 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 40 types, with **extended file system** ext2 and ext3 leading; plus distributed file systems, etc.)

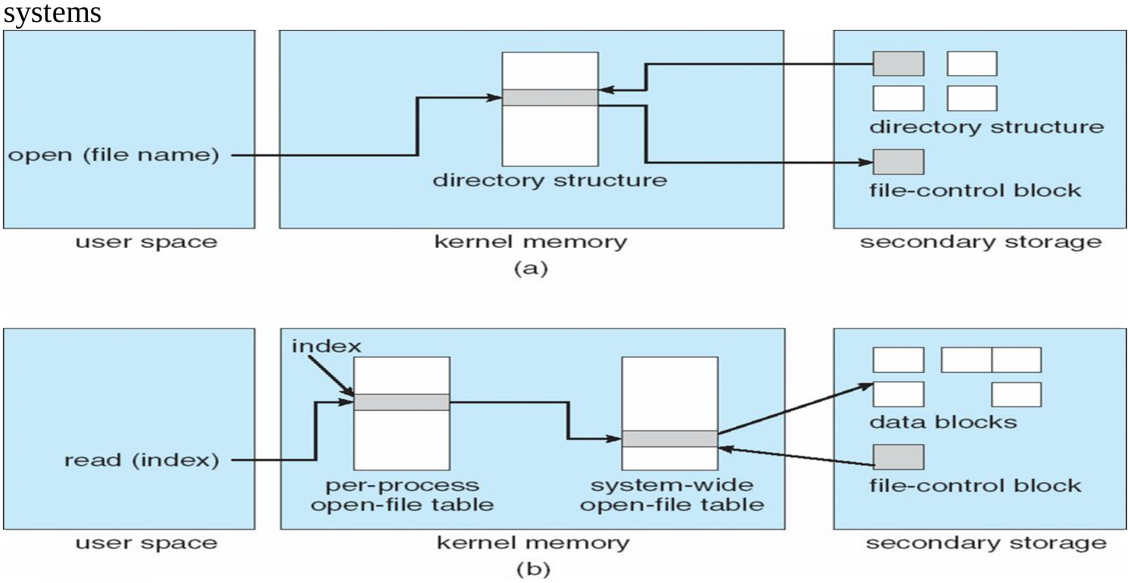
File-System Implementation

- 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 (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
- Per-file **File Control Block (FCB)** contains many details about the file
    - inode number, permissions, size, dates
    - NFTS 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
- The following figure illustrates the necessary file system structures provided by the operating systems



- Figure 12-3(a) refers to opening a file   Figure 12-3(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 eventually copied to specified user process memory address

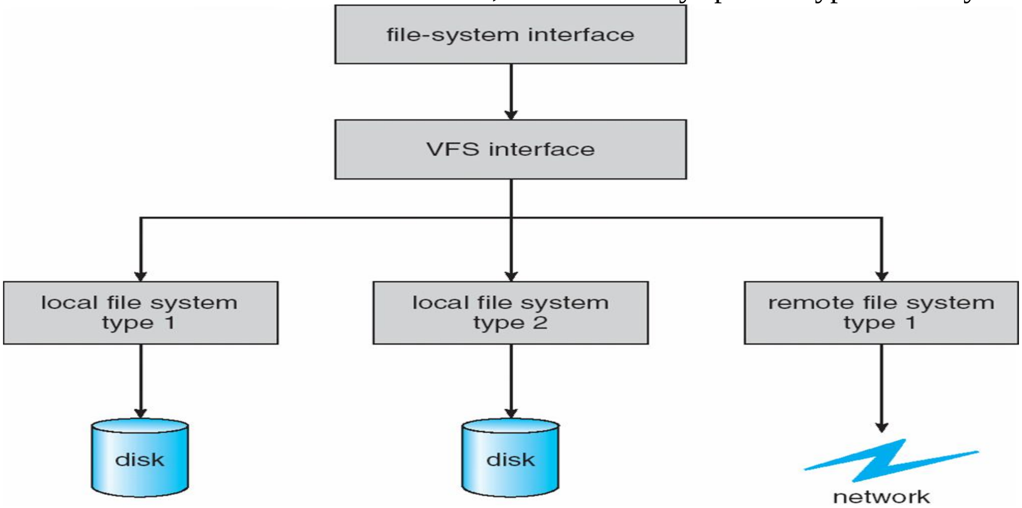
Partitions and Mounting

- Partition can be a volume containing a file system ("cooked") or **raw** – just a sequence of blocks with no file system

- Boot block can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system
    - Or a boot management program for multi-os booting
- **Root partition** contains the OS, other partitions can hold other Oses, other file systems, or be raw
    - Mounted at boot time
    - Other partitions can mount automatically or manually
- At mount time, file system consistency checked
    - Is all metadata correct?
        - If not, fix it, try again
        - If yes, add to mount table, allow access

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
        - Implements **vnodes** which hold inodes or network file details
    - Then dispatches operation to appropriate file system implementation routines
- The API is to the VFS interface, rather than any specific type of file system



Virtual File System Implementation

- For example, Linux has four object types:
    - inode, file, superblock, dentry
- VFS defines set of operations on the objects that must be implemented
    - Every object has a pointer to a function table
        - Function table has addresses of routines to implement that function on that object
        - For example:
            - **int open(. . .)**—Open a file
            - **int close(. . .)**—Close an already-open file
            - **ssize t read(. . .)**—Read from a file
            - **ssize t write(. . .)**—Write to a file
            - **int mmap(. . .)**—Memory-map a file

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

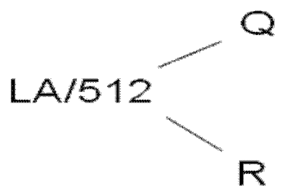
- **Hash Table –** linear list with hash data structure
    - Decreases directory search time
    - **Collisions** – situations where two file names hash to the same location
    - Only good if entries are fixed size, or use chained-overflow method

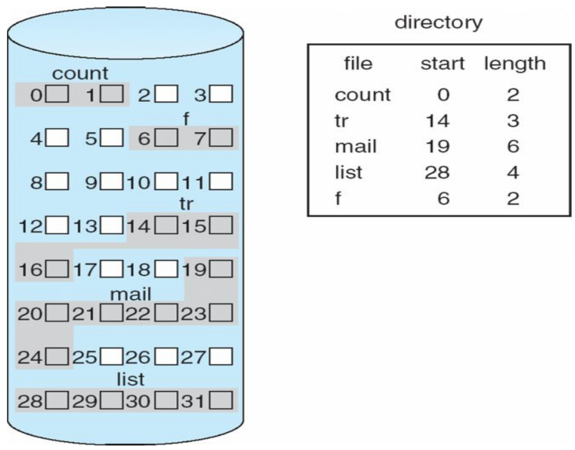
Allocation Methods – Contiguous

- 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**

1. **Contiguous Allocation**

n Mapping from logical to physical





n Block to be accessed = Q + starting address

Displacement into block = R

2. **Extent-Based Systems**

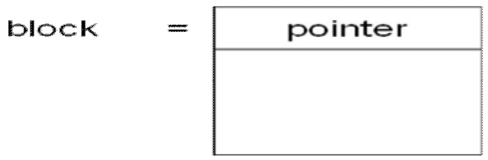
- Many newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in extents
- An **extent** is a contiguous block of disks
    - Extents are allocated for file allocation
    - A file consists of one or more extents
    -

3.Allocation Methods – Linked

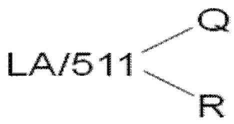
- **Linked allocation –** each file a linked list of blocks
    - File ends at nil pointer
    - No external fragmentation
    - Each block contains pointer to next block
    - No compaction, external fragmentation
    - Free space management system called when new block needed
    - Improve efficiency 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
- FAT (File Allocation Table) 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

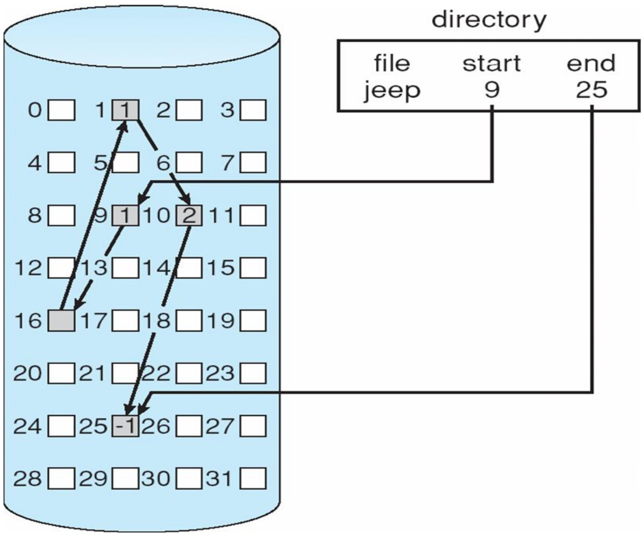


Mapping

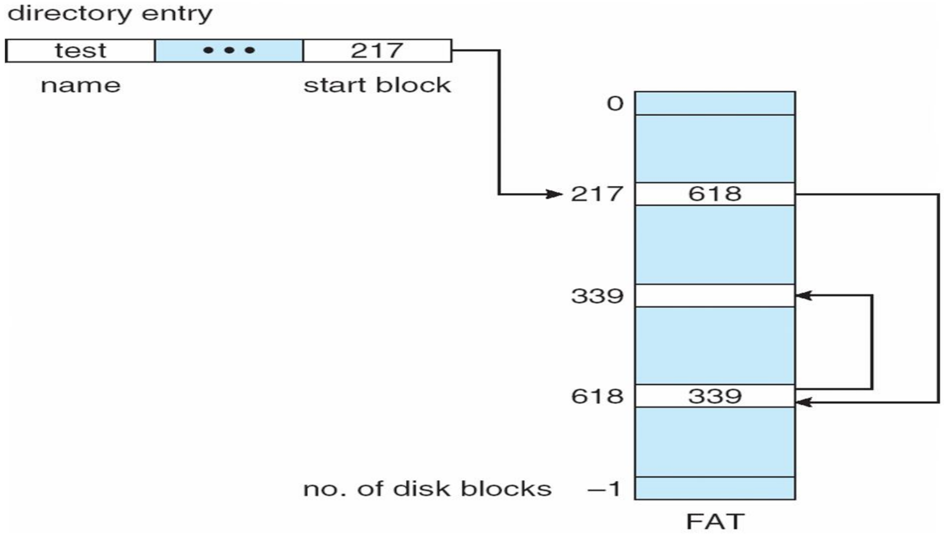


Block to be accessed is the Qth block in the linked chain of blocks representing the file.
Displacement into block = R + 1

Linked Allocation



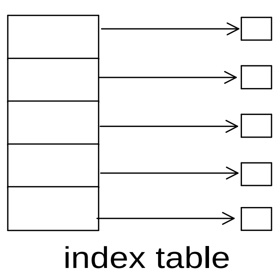
File-Allocation Table



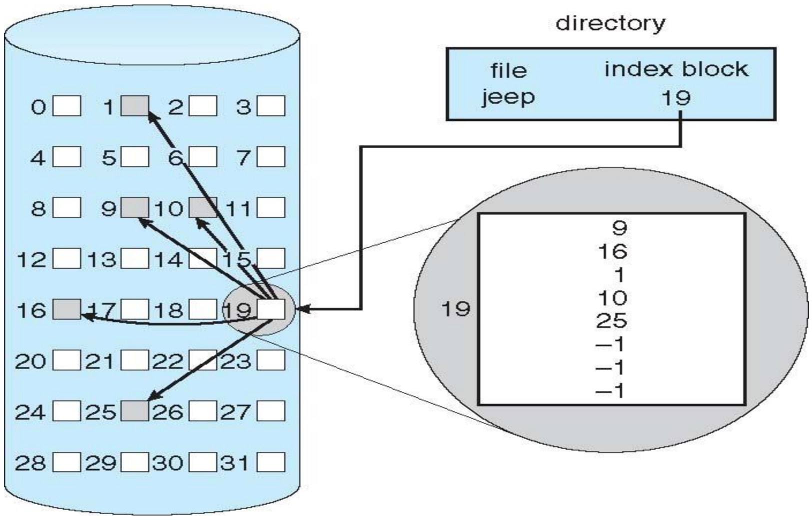
4.Allocation Methods – Indexed

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

- Logical view

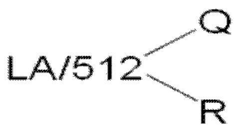


Example of Indexed Allocation

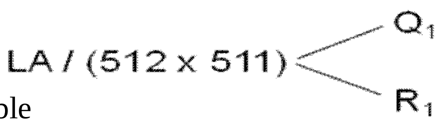


- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index table

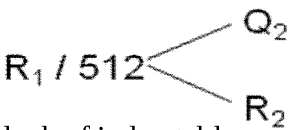
Q = displacement into index table
R = displacement into block



- 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)

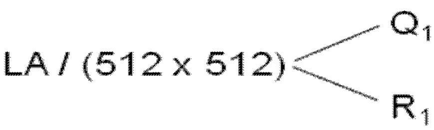


Q1 = block of index table
R1 is used as follows:

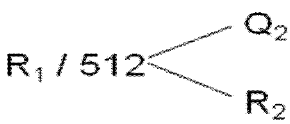


Q2 = displacement into block of index table
R2 displacement into block of file:

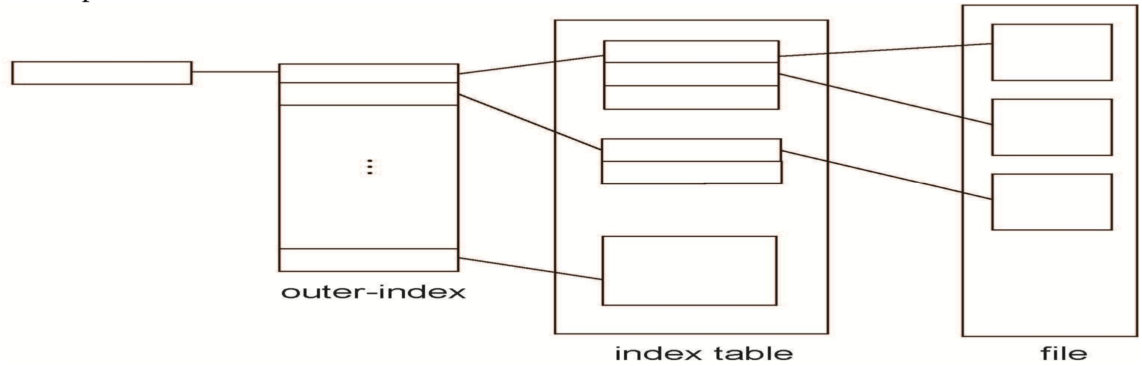
- Two-level index (4K blocks could store 1,024 four-byte pointers in outer index -> 1,048,567 data blocks and file size of up to 4GB)



Q1 = displacement into outer-index
R1 is used as follows:

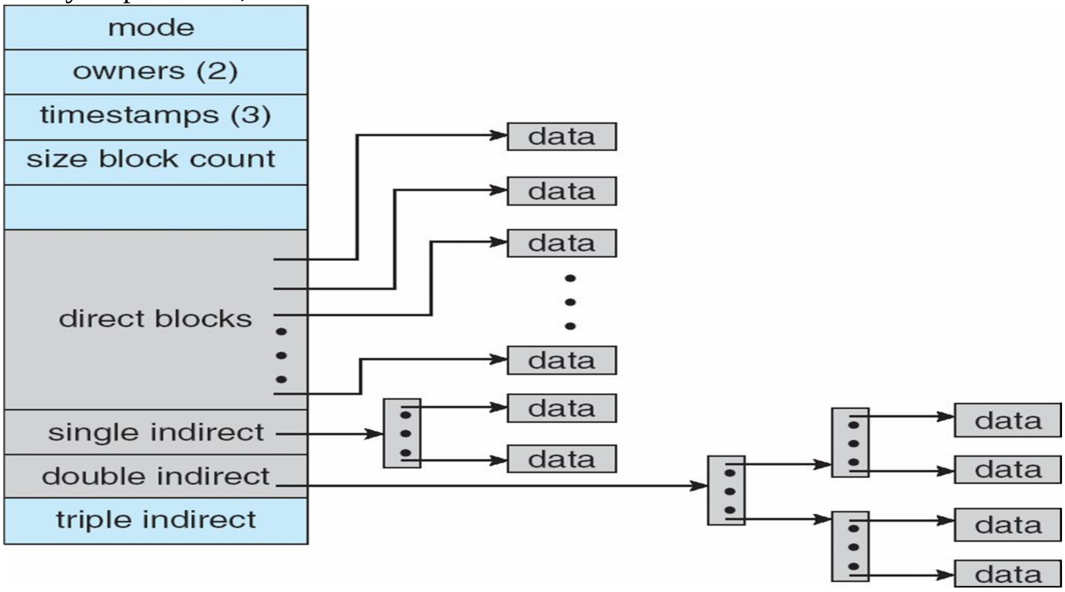


Q2 = displacement into block of index table
R2 displacement into block of file:



<u>Combined Scheme: UNIX UFS</u>

4K bytes per block, 32-bit addresses



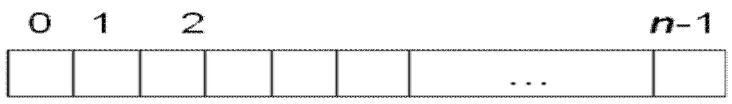
More index blocks than can be addressed with 32-bit file pointer

Performance

- Best method depends on file access type
  - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation -> select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads then data block read
  - Clustering can help improve throughput, reduce CPU overhead
- 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 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 maintains **free-space list** to track available blocks/clusters
  - (Using term "block" for simplicity)
- **Bit vector** or **bit map** (*n* blocks)



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

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

- Bit map requires extra space
  - Example:

    block size = 4KB = 212 bytes

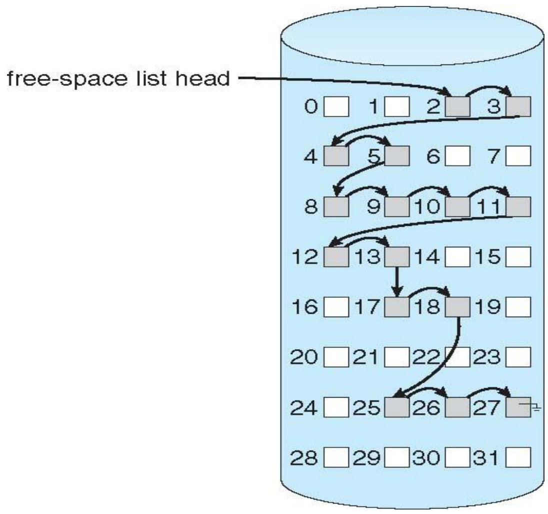
    disk size = 240 bytes (1 terabyte)

    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)



- 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
- 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

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