Operating System

Chapter 11: File System

Why file system is important?

- For most users, File System (FS) is the most visible aspect of an OS
- Provides mechanism to access data/programs on storage
- Any FS consists of two distinct parts
  – A collection of Files
  – A directory structure that organizes and provides information about all files in the system

What is a file?

- File: A contiguous logical address space, logical storage unit
- Types
  - Data
    - Numeric, text, data, photo, music, etc
  - Program
- Contents defined by file's creator, many types are
  - Text file
    - A sequence of characters
  - Source file
    - A sequence of functions
  - Executable file
    - A series of code sections that loader can bring into memory and execute

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 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

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 $\boldsymbol{F_i}$, and move the content of entry to memory
- ***Close (F_i)***
  - Move the content of entry $\boldsymbol{F_i}$ in memory to directory structure on disk

Open files

- Open (Fi): move the content of a file to memory
- Search the directory structure on disk for the file
- To avoid constant searching: open() system should be called before a file is first used
  - Open-file table: tracks all open files
    - Per-process table
    - System-wide table
- When the file is no longer being actively used, it is closed by the process, and OS removes its entry from open-file table using *Open Count*

Other information for an open file

- File pointer
  - Pointer to last read/write location, per process that has the file open
- File-open count
  - Counter of the number of times a file is open to allow removal of data from open-file table when last process closes it
- Disk location of the file
  - Moving data access information to memory
- Access rights
  - Per-process access mode information

Locking in open file

- File locks allows one process to lock a file, prevent other process from gaining access to it

- Similar to reader-writer locks
  - Shared lock similar to reader lock
    - Several processes can acquire concurrently

  - Exclusive lock similar to writer lock
    - Only on process can acquire it

Other locking mechanisms

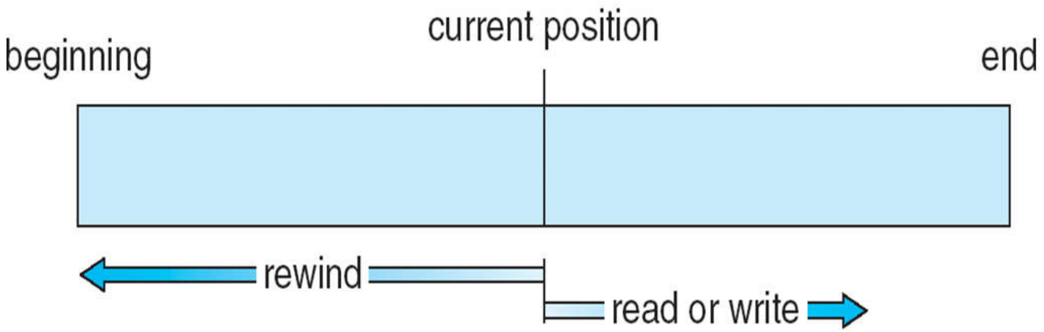
- Mandatory
  - OS will prevent access until the exclusive lock is released
    - Windows®
- Advisory
  - OS will not prevent applications from acquiring access to a file, and the application must be developed to manually acquire the lock before accessing the file
    - UNIX®

File structure

- Files must conform to structures that are understood by OS
  - OS requires an executable file has a specific structure; it can determine where in memory to load the file, what the location of first instruction is
- Support of multiple file structures?
  - Size of OS could be big; it needs to contain codes to support these file structures
  - Severe problems may result if OS does not support some file structures

File access methods: 1) Sequential access

- Simplest and most common
- Based on tape model of a file
- Processing information in a file is in order: one record after the other
- A read operation, read_next()
  - Reads the next portion of the file and automatically advances a file pointer
- A write operation, write_next()
  - Appends to the end of the file and advances to the end of the newly written information



File access methods: 2) Direct access

- Based on a disk model of a file
- A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order
- Immediate access to large amount of information
  - Databases are often of this type
- read(*n*) rather than read_next()
  - *n* is block number
- write(*n*) rather than write_next()

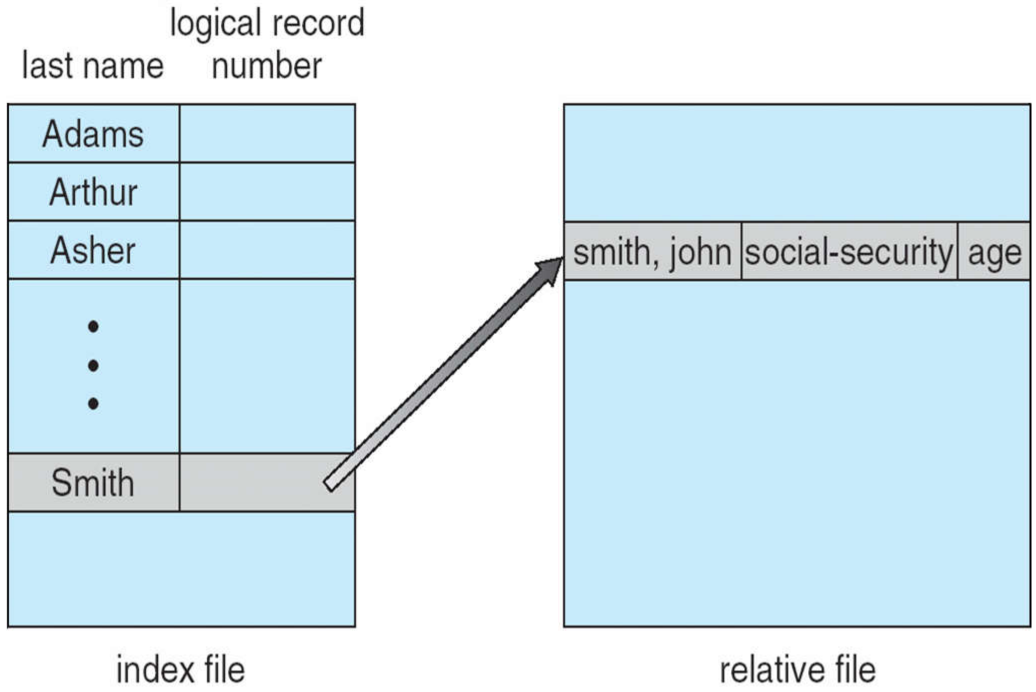
sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Other access methods

- Can be built on top of base methods

- Creation of an index for a file
  - Having pointers to various blocks

- Keep index in memory for fast determination of location of data



IBM's indexed sequential access method (ISAM)

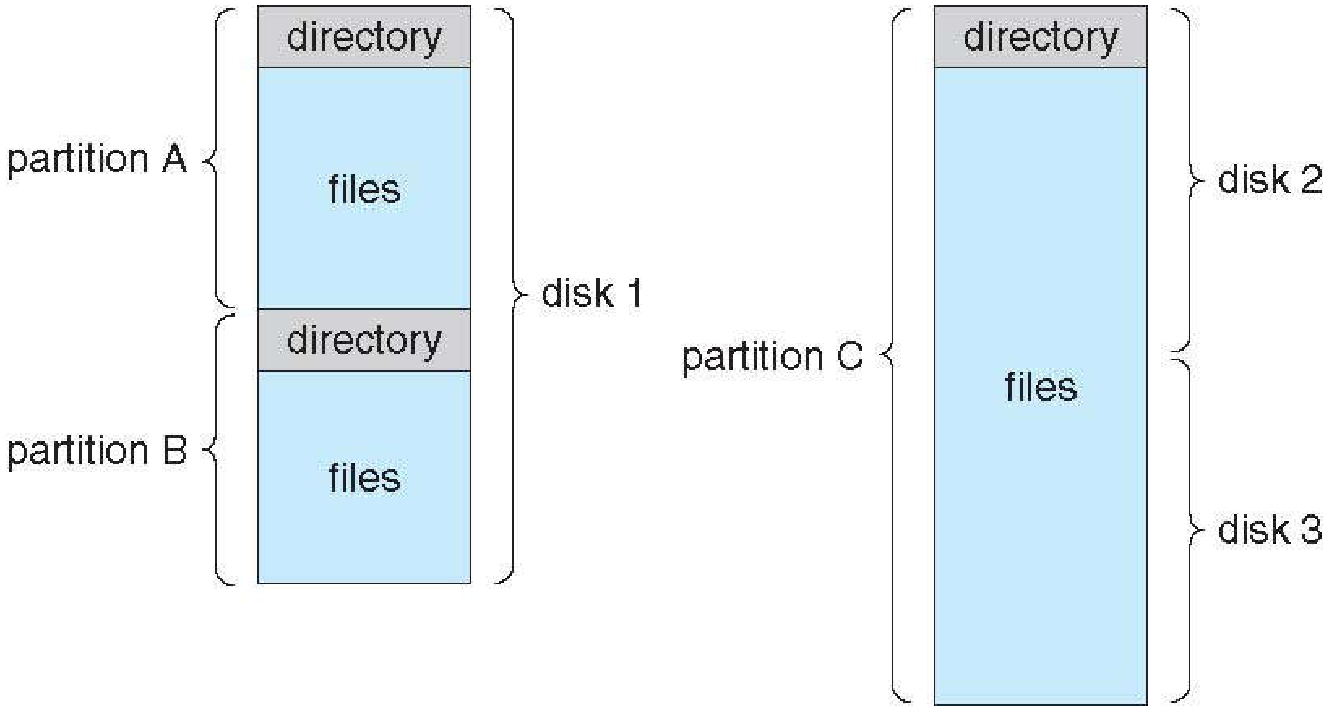
Directory & Disk Structure

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**
- Partitions also known as minidisks, slices
- Entity containing file system known as a **volume**
- Each volume contains information about the files in the system
  - This information is kept in entries in a device directory or volume table of contents
- The device directory, (known as the directory), records information such as name, location, size, and type for all files on that volume.

A typical file-system organization

- A file system can be created on each of these parts of the disk. Any entity containing a file system is generally known as a **volume**.
- Each volume that contains a file system must also contain information about the files in the system.
  - ✓ **Device directory** or **volume table of contents**.



Types of file systems

- Systems frequently have many 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

Directory

- Directory
  - Can be viewed as a symbol table that translates file names into their directory entries

- Both the directory structure and the files reside on disk

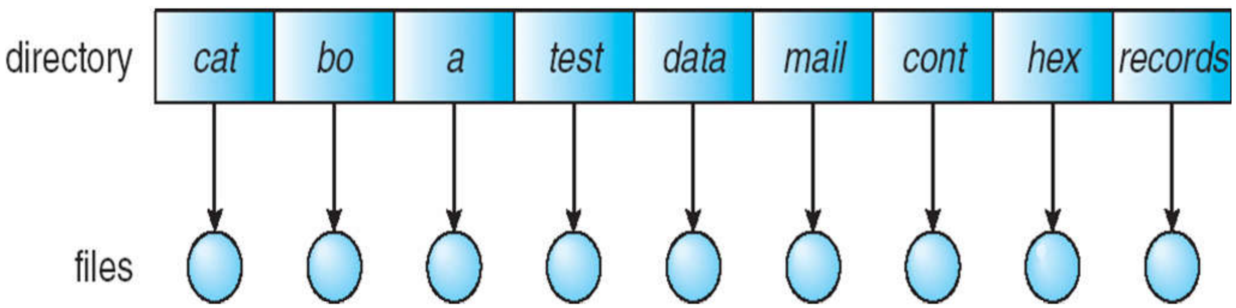
- Operations on directories
  - Search for a file
  - Create a file
  - Delete a file
  - List a directory
  - Rename a file
  - Traverse the file system

Directory organization

- 1) Single-level directories
- 2) Two-level directories
- 3) Tree-structure directories
- 4) Acyclic-graph directories
- 5) General graph directories

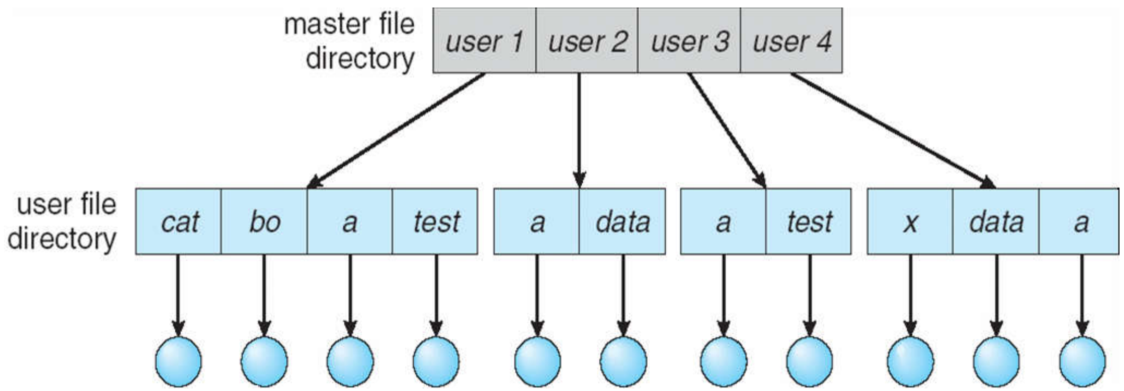
1) Single-level directory

- A single directory for all users
- Naming problem: they must have unique names
- Grouping problem



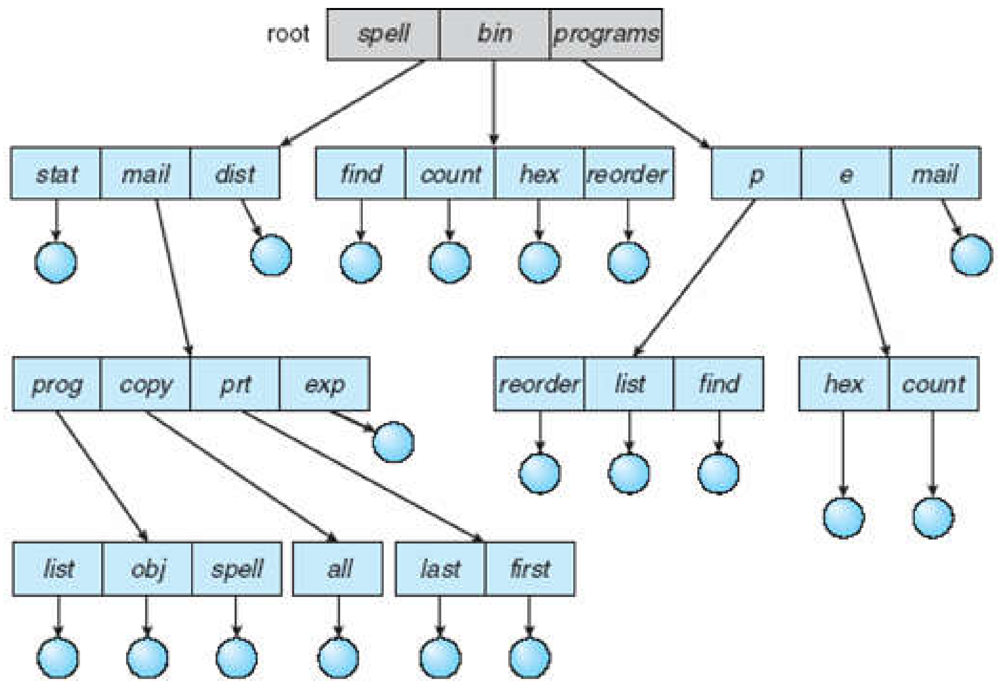
2) Two-level directory

- Separate directory for each user
- Can have the same file name for different users
- Efficient searching
- Path name: two level path, e.g., /userN/file.txt
- No grouping capability
- Sharing problem



3) Tree-structured directories

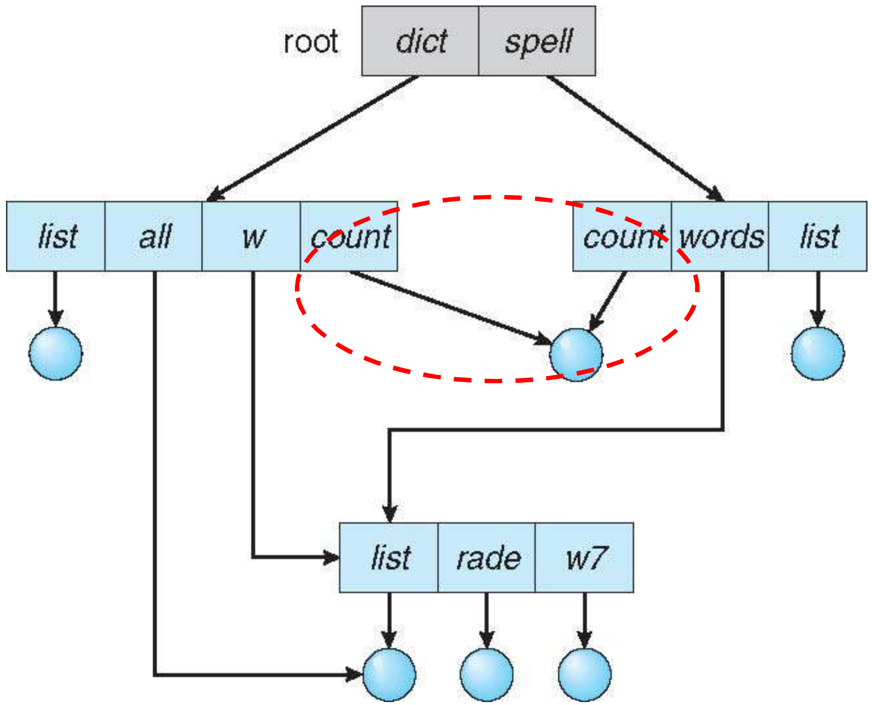
- Efficient searching
- Grouping capability
- Two types of path names
  - Absolute path name
    - Begins at the root and follows a path down to the specified file
  - Relative path name
    - A path from the current directory
- Deleting a directory
  - 1. Not allowed if is not empty
  - 2. Have an option of delete internal nodes
- Sharing problem



4) Acyclic-graph directories

- Have shared subdirectories and files
  - Only one actual file exists, so any changes made by one person are immediately visible to the other

- Methods of shared files implementation
  - 1) Link
    - Another name (pointer) to an existing file
    - Resolve the link: follow pointer to locate the file
  - 2) Duplicate all information about the file
    - Both entries are identical and equal
    - Consistency problem (why?)



➢ Deletion & Traversing problems (?)

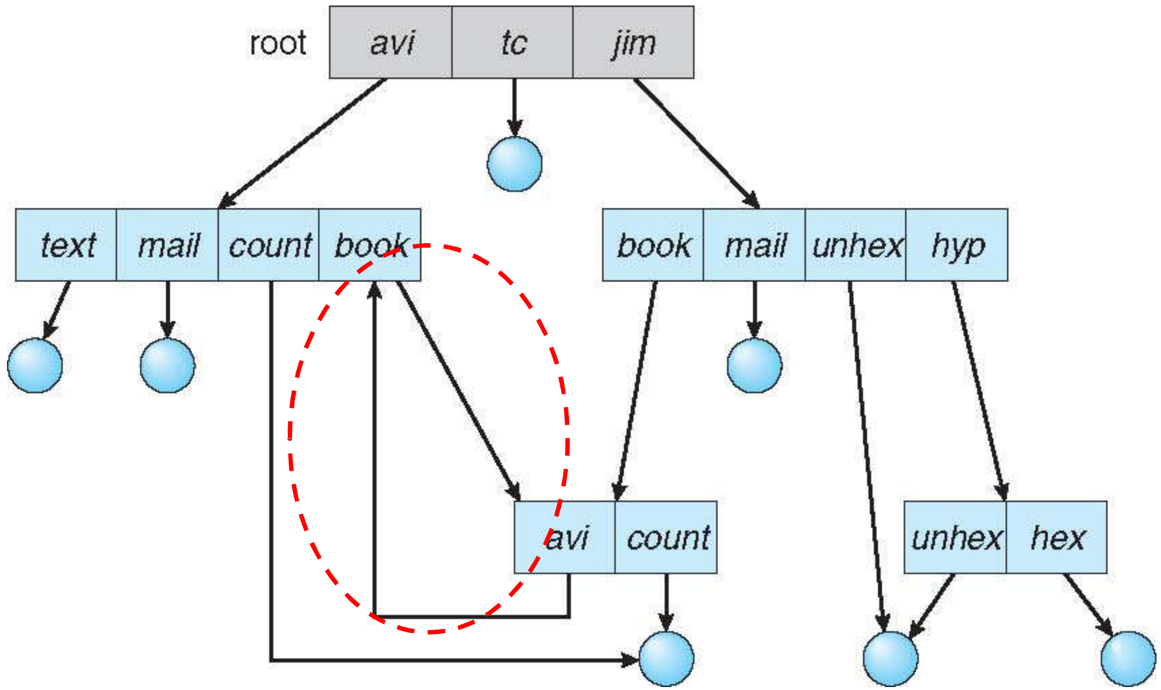
Deletion possibilities?

- 1) Remove the file content whenever anyone deletes it
  - Dangling pointers: pointing to the nonexistent file
  - What if the remaining file pointers contain actual disk addresses?
  - Easy with soft-links (symbolic links)
- 2) Preserve the file until all references to it are deleted
  - Hard links
  - Counting number of references

4) General graph directories

- Remove problem of no cycles

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



File System Mounting

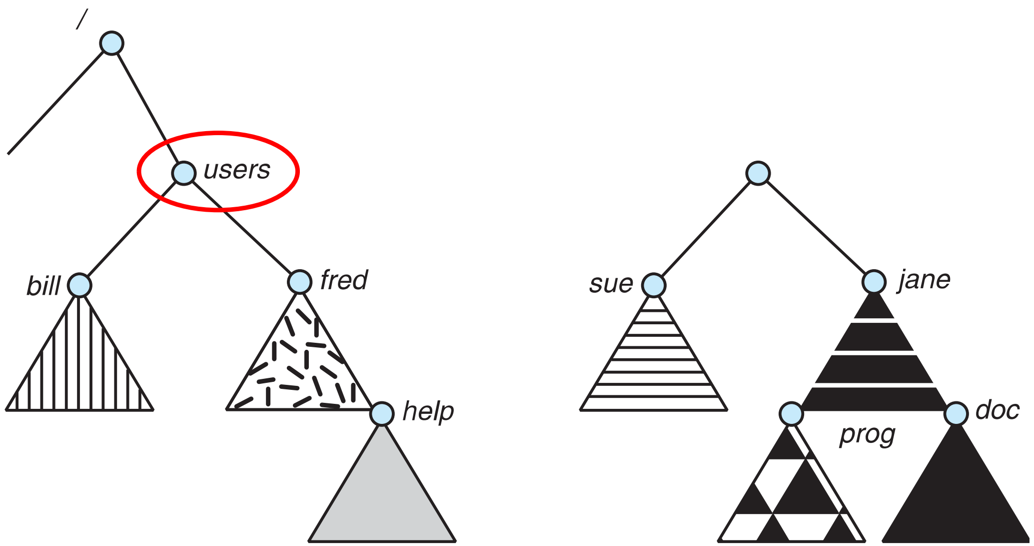
File system mounting

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

- A unmounted file system is mounted at a mount point

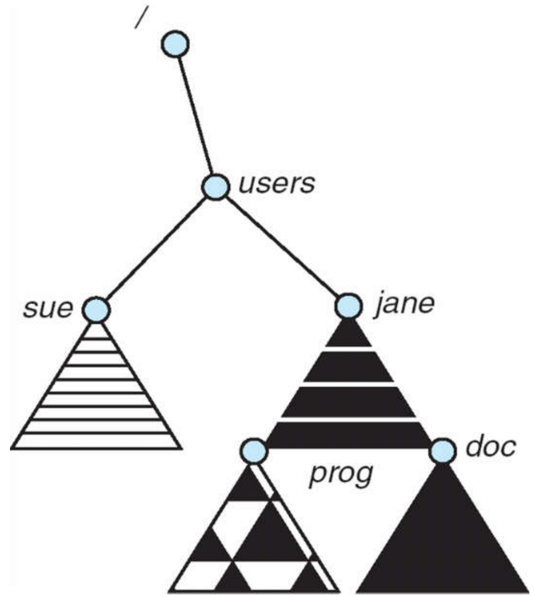
- Mount point
  - The location within the file structure where the file system is to be attached

File system mounting & mount point



(a) (b)

(a) Existing FS (b) unmonted FS



The mounted FS

```
mount -t type device dir
```

mount –t ext4 /device/dsk /home/users

File Sharing & Protection

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 system

- 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 (Common Internet FS)** is standard Windows protocol
  - Standard OS 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 **(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 semantics

- 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)
    - 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

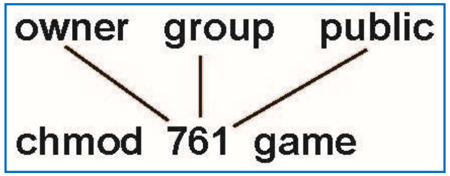
- File owner/creator should be able to control
  - What can be done
  - By whom
- Types of access
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List

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	⇒	0 0 1

- 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



Attach a group to a file

```
chgrp     G     game
```

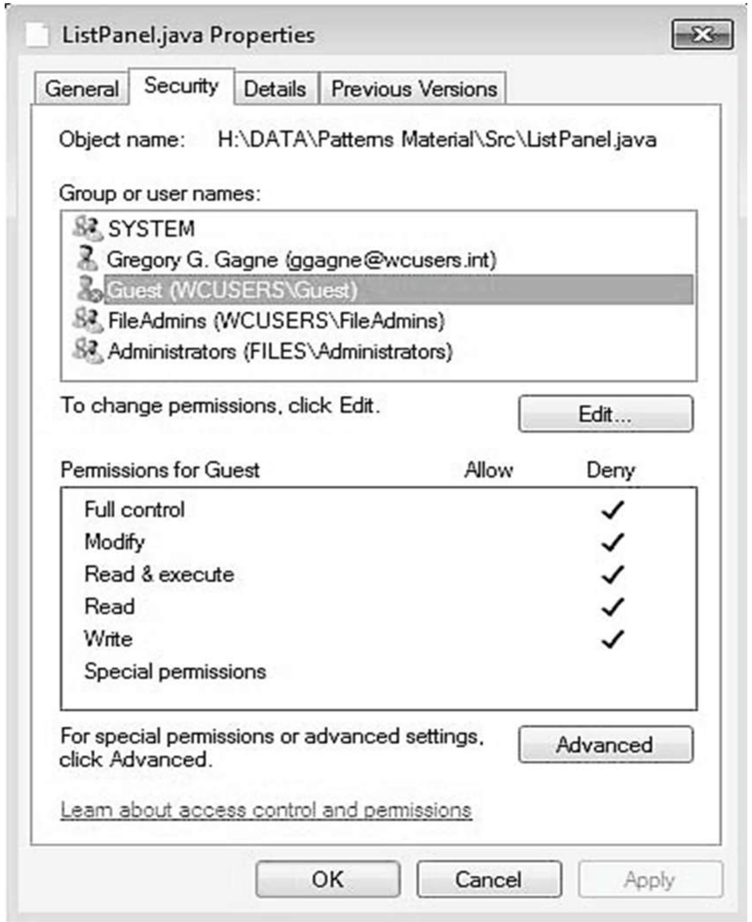
A sample UNIX directory listing

Associated with each subdirectory are three fields—owner, group, and universe—each consisting of the three bits rwx.

-rw-rw-r--	1	pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5	pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2	pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2	pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1	pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1	pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4	pbg	faculty	512	Jul 31 10:31	lib/
drwx------	3	pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3	pbg	staff	512	Jul 8 09:35	test/

Windows 7 access control list management

Windows users typically manage access-control lists via the GUI.



implementation

I/O & Storage Layers

Application / Service

Operations, Entities and Interface

High Level I/O — *streams*

Low Level I/O — *handles*

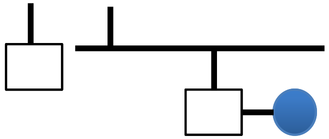
Syscall — *registers*

`file_open, file_read, … on struct file * & void *`

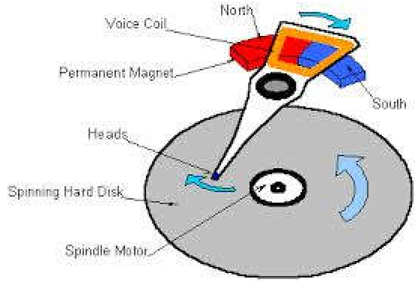
File System — *descriptors*

we are here …

I/O Driver — *Commands and Data Transfers*



Disks, Flash, Controllers, DMA



C Low level I/O

- Operations on File Descriptors – as OS object representing the state of a file
  - User has a “handle” on the descriptor

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

int open (const char *filename, int flags [, mode_t mode])
int create (const char *filename, mode_t mode)
int close (int filedes)
```

Bit vector of:
- Access modes (Rd, Wr, …)
- Open Flags (Create, …)
- Operating modes (Appends, …)

Bit vector of Permission Bits:
- User|Group|Other X R|W|X

http://www.gnu.org/software/libc/manual/html_node/Opening-and-Closing-Files.html

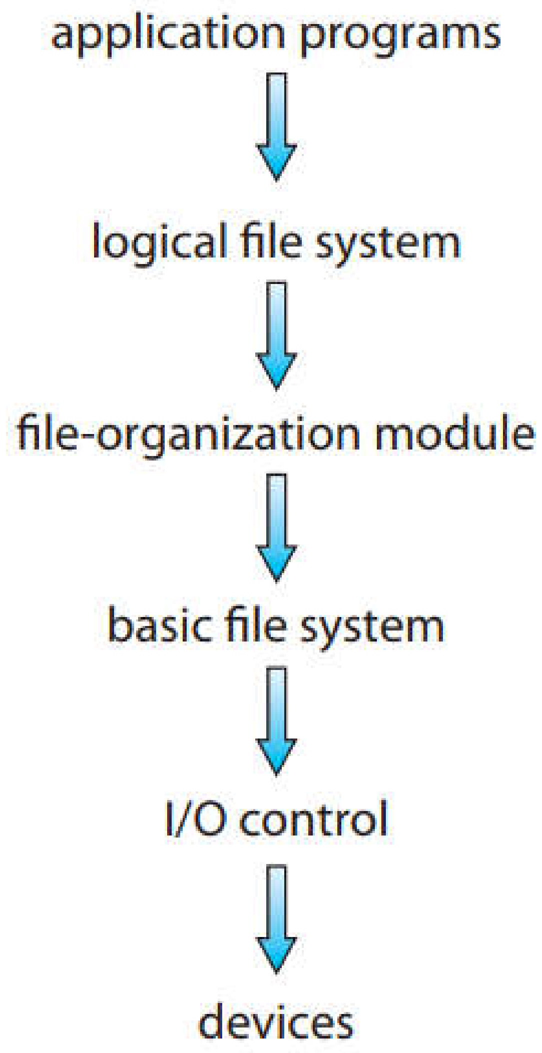
C Low Level Operations

```
ssize_t read (int filedes, void *buffer, size_t maxsize)
 - returns bytes read, 0 => EOF, -1 => error
ssize_t write (int filedes, const void *buffer, size_t size)
 - returns bytes written
off_t lseek (int filedes, off_t offset, int whence)
 - set the file offset
   * if whence == SEEK_SET: set file offset to “offset”
   * if whence == SEEK_CRT: set file offset to crt location + “offset”
   * if whence == SEEK_END: set file offset to file size + “offset”
int fsync (int fildes)
 – wait for i/o of filedes to finish and commit to disk
void sync (void) – wait for ALL to finish and commit to disk
```

- When write returns, data is on its way to disk and can be read, but it may not actually be permanent!

File-System Structure

- The **I/O control** level consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system.
- The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.
- The **file-organization module** knows about files and their logical blocks, as well as physical blocks.
- The **logical file system** manages metadata information. Metadata includes all of the file-system structure except the actual data .



Building a File System

- File System: Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.

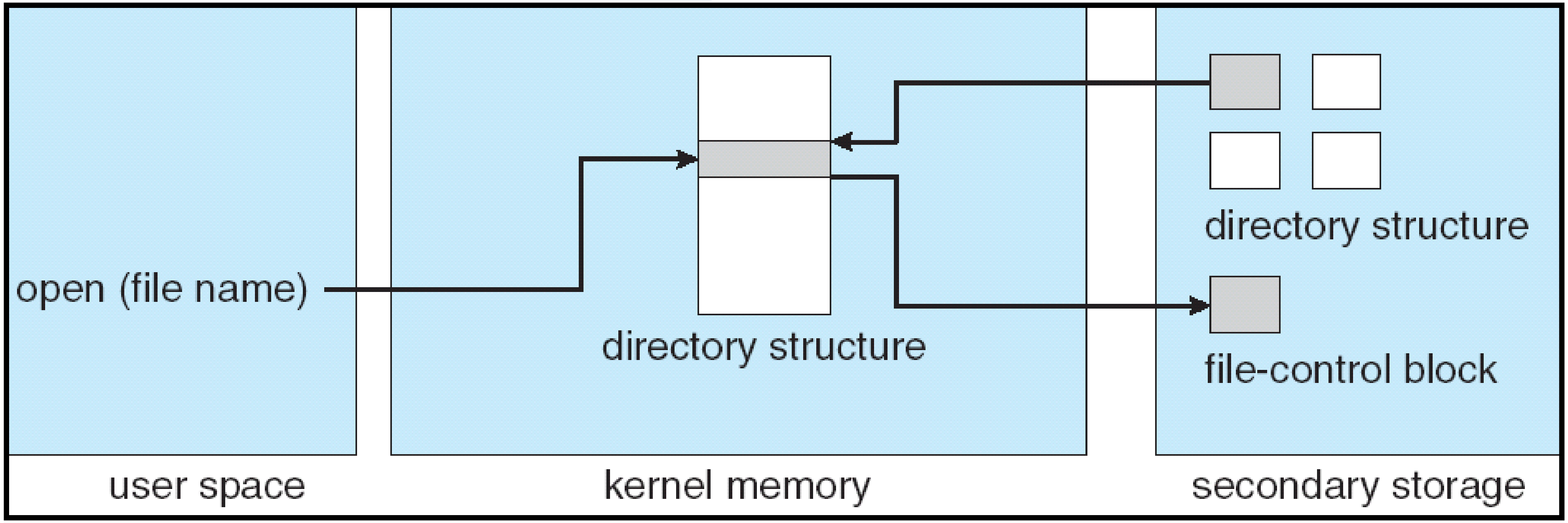
- File System Components
  - Naming: Interface to find files by name, not by blocks
  - Disk Management: collecting disk blocks into files
  - Protection: Layers to keep data secure
  - Reliability/Durability: Keeping of files durable despite crashes, media failures, attacks, etc.

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

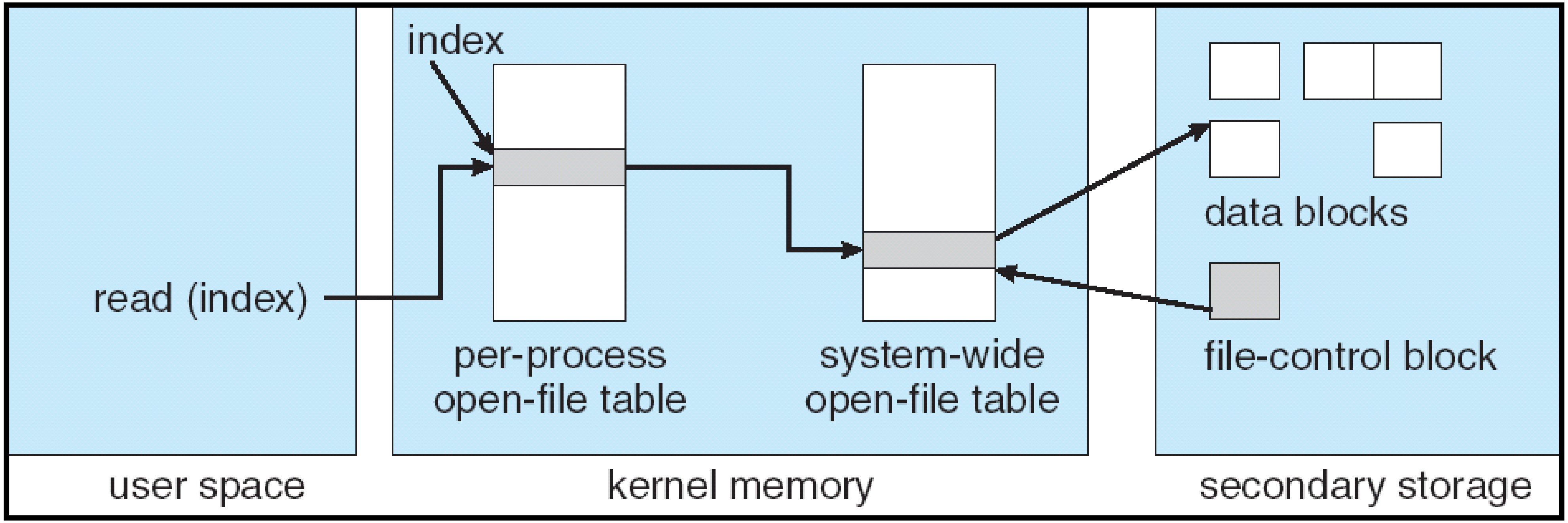
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

In-Memory File System Structures



- Open system call:
  - Resolves file name, finds file control block (inode)
  - Makes entries in per-process and system-wide tables
  - Returns index (called “file handle”) in open-file table

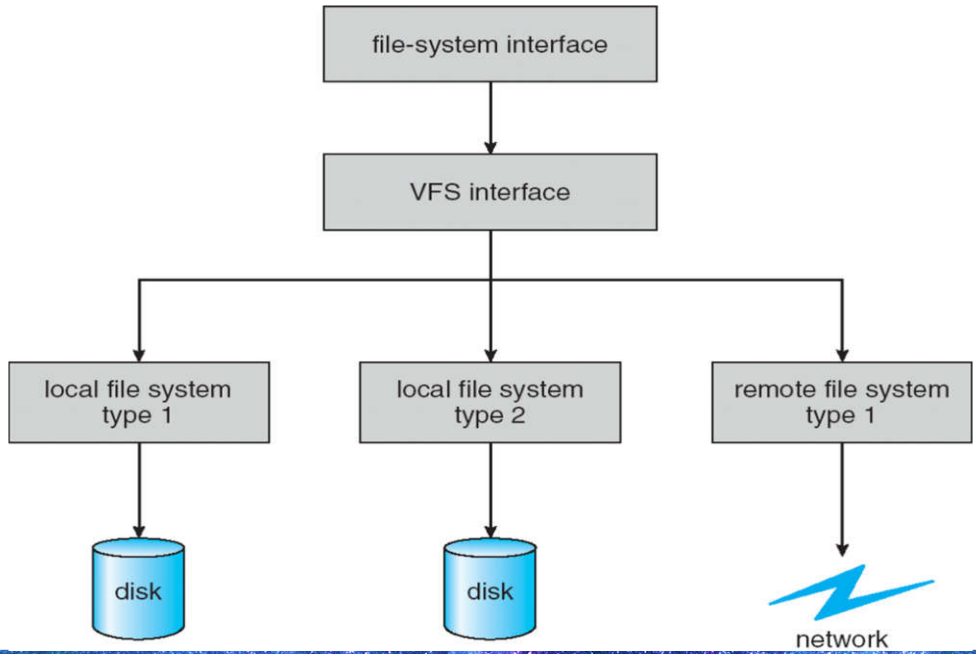
In-Memory File System Structures

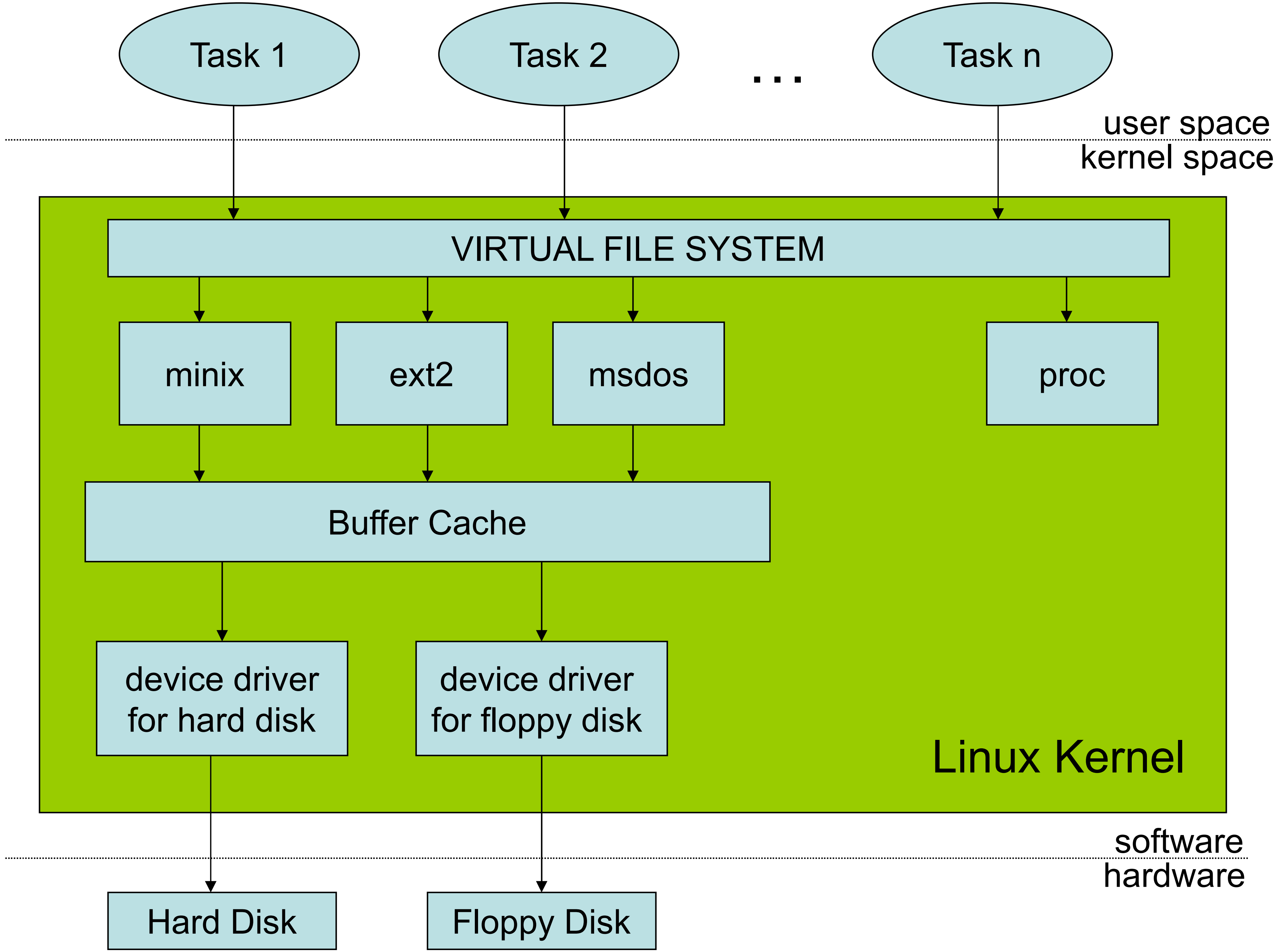


- Read/write system calls:
  - Use file handle to locate inode
  - Perform appropriate reads or writes

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





Robbed from http://www.cs.usfca.edu/~cruse/cs326/lesson22.ppt

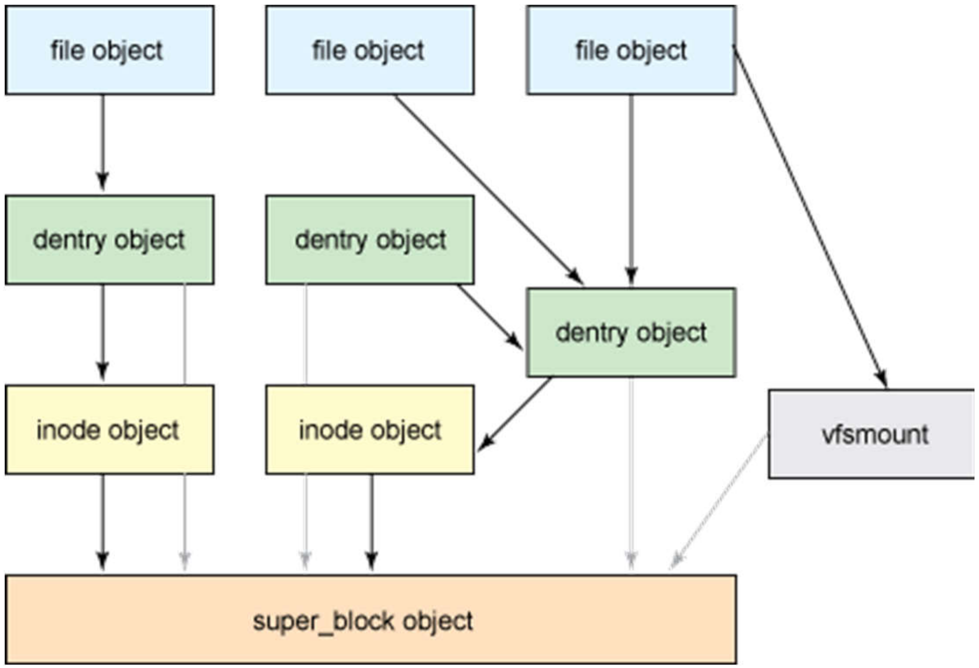
Virtual File System Example

- 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

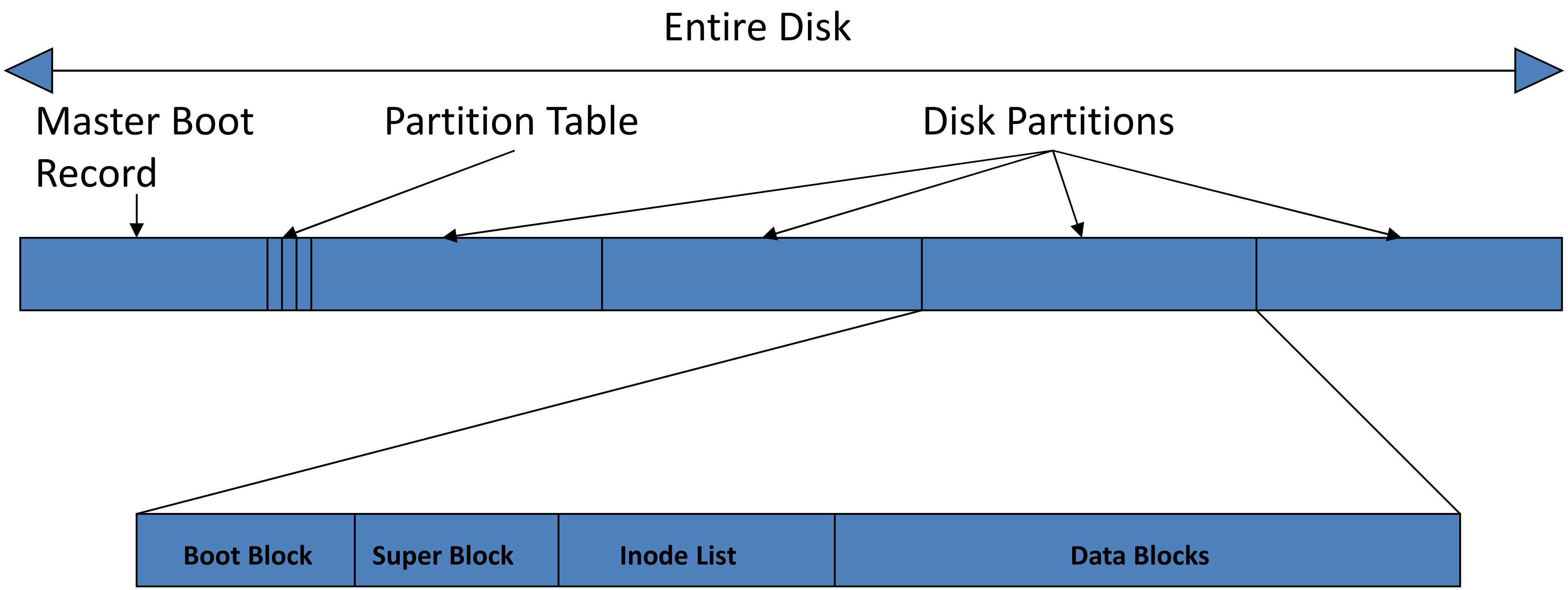
```
struct file_operations {
        int (*lseek) (struct inode *, struct file *, off_t, int);
        int (*read) (struct inode *, struct file *, char *, int);
        int (*write) (struct inode *, struct file *, const char *, int);
        int (*readdir) (struct inode *, struct file *, void *, filldir_t);
        int (*select) (struct inode *, struct file *, int, select_table *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        int (*mmap) (struct inode *, struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        void (*release) (struct inode *, struct file *);
        int (*fsync) (struct inode *, struct file *);
        int (*fasync) (struct inode *, struct file *, int);
        int (*check_media_change) (kdev_t dev);
        int (*revalidate) (kdev_t dev);
};
```

Relationships of major objects in the VFS

- An **Inode** is a data structure on a Unix / Linux file system. An inode stores meta data about a regular file, directory, or other file system object.
- **Dentry** uses to keep track of the hierarchy of files in directories. Each dentry maps an inode number to a file name and a parent directory.
- The **superblock** is the container for high-level metadata about a file system.



file system Layout



A Possible File System Instance Layout

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

Disk Block Allocation

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
  - …

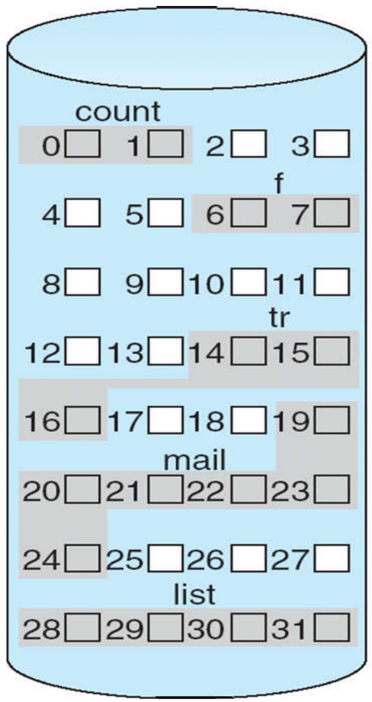
Contiguous Allocation

- Contiguous allocation: each file occupies set of contiguous blocks
  - best performance in most cases
  - simple to implement: only starting location and length are required
- Contiguous allocation is not flexible
  - how to increase/decrease file size?
    - need to know file size at the file creation?
  - external fragmentation
    - how to compact files offline or online to reduce external fragmentation
  - need for compaction off-line (downtime) or on-line
- appropriate for sequential disks like tape
- Some file systems use extent-based contiguous allocation
  - extent is a set of contiguous blocks
  - a file consists of extents, extents are not necessarily adjacent to each other

Contiguous Allocation

- Mapping from logical to physical

Block to be accessed = Q + starting address
Displacement into block = R



directory

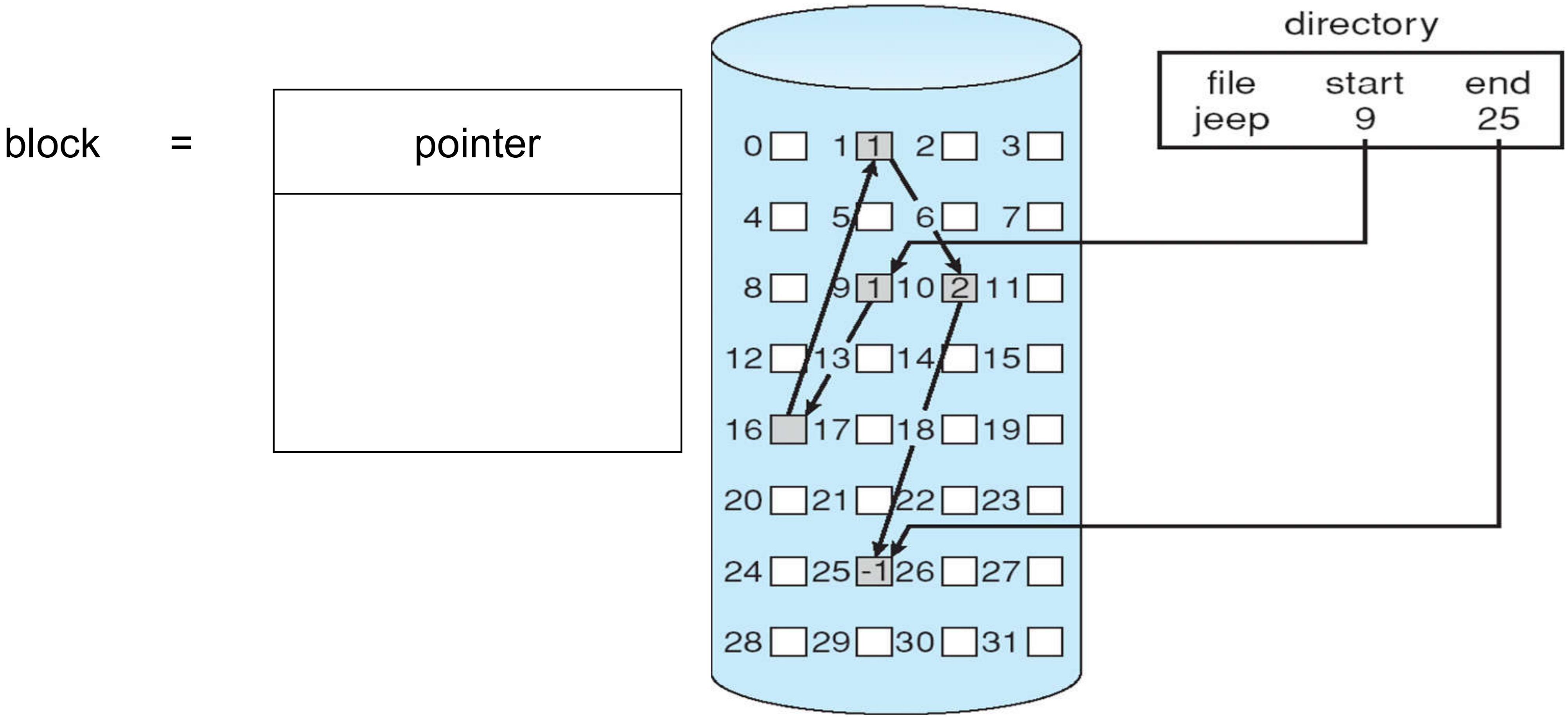
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Linked Allocation

- Linked allocation: each file is a linked list of disk blocks
  - each block contains pointer to next block, file ends at null 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

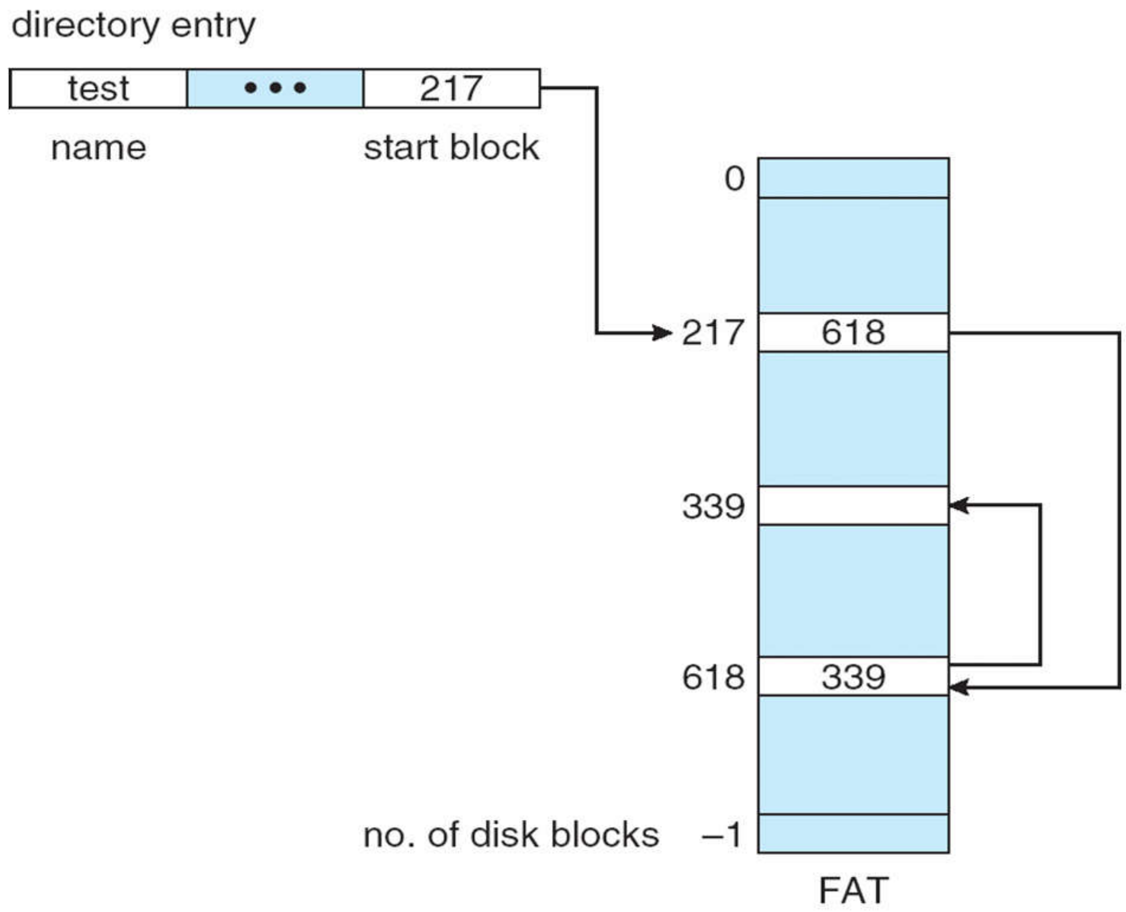
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk



File-Allocation Table (FAT): MS-DOS

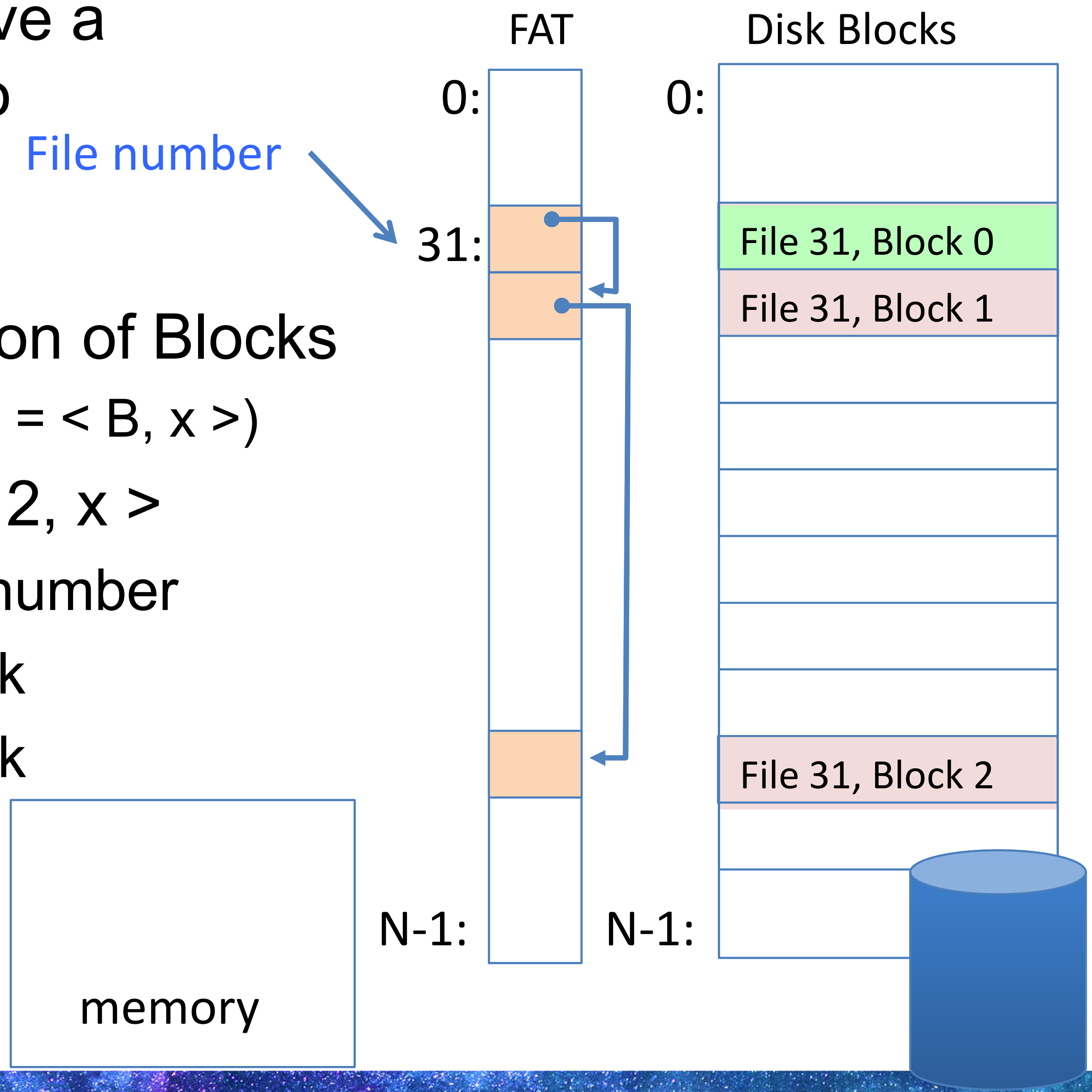
- FAT (File Allocation Table) uses linked allocation

- A section of disk at the beginning of each volume is set aside to contain the table.
- The table has one entry for each disk block and is indexed by block number.



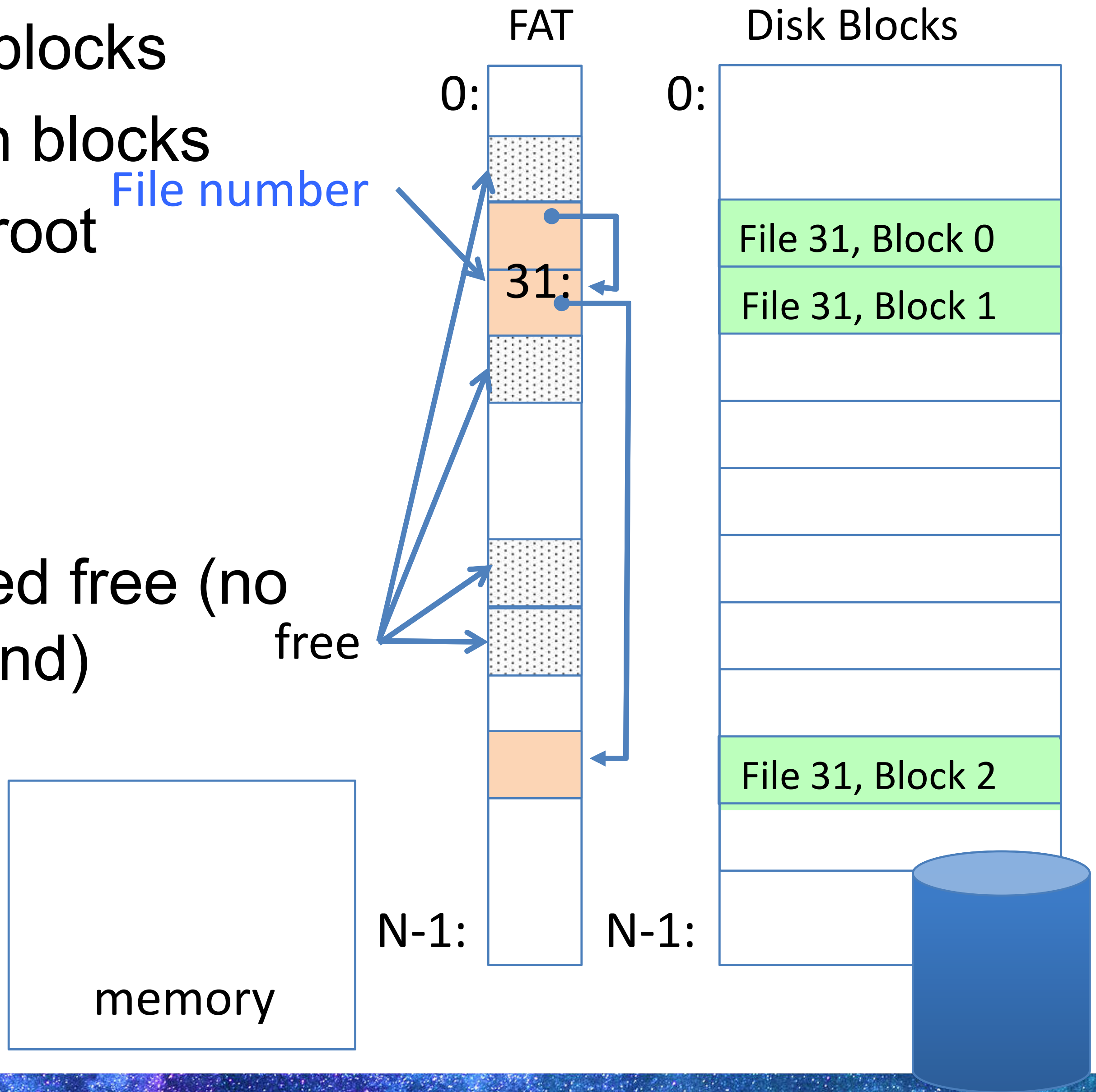
FAT (File Allocation Table)

- Assume (for now) we have a way to translate a path to a "file number"
  - i.e., a directory structure
- Disk Storage is a collection of Blocks
  - Just hold file data (offset o = < B, x >)
- Example: file_read 31, < 2, x >
  - Index into FAT with file number
  - Follow linked list to block
  - Read the block from disk into memory



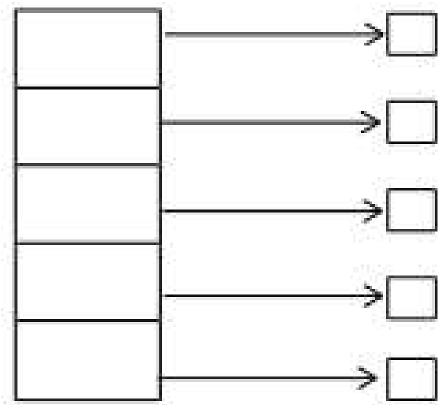
FAT Properties

- File is collection of disk blocks
- FAT is linked list 1-1 with blocks
- File Number is index of root of block list for the file
- File offset (o = < B, x >)
- Follow list to get block #
- Unused blocks ⬄ Marked free (no ordering, must scan to find)



Indexed Allocation

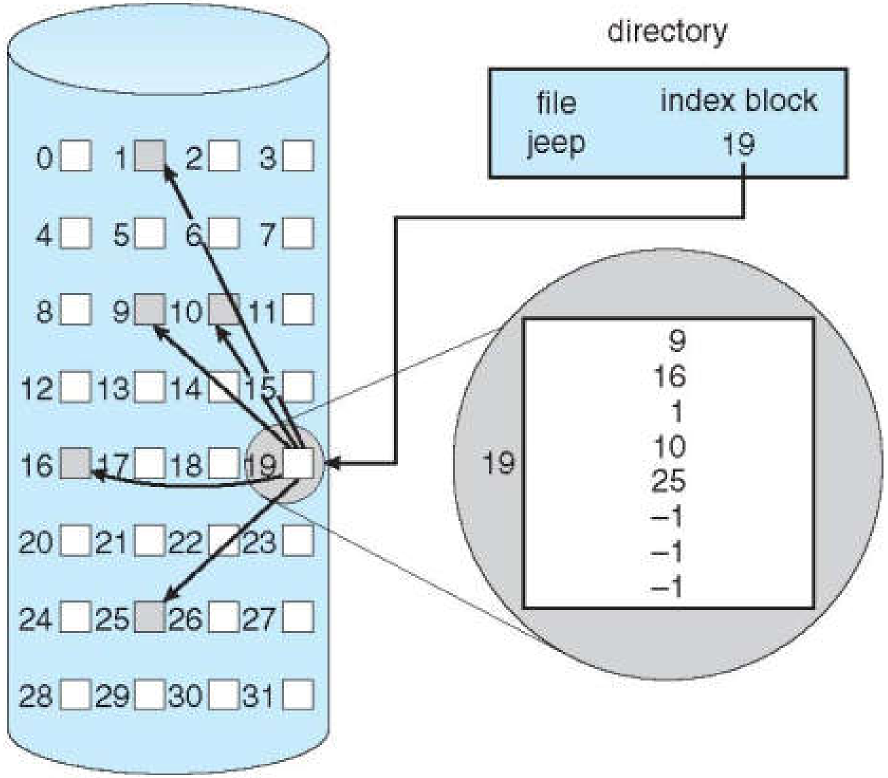
- Indexed allocation: each file has its own index blocks of pointers to its data blocks
  - index table provides random access to file data blocks
  - no external fragmentation, but overhead of index blocks
  - allows holes in the file
  - Index block needs space - waste for small files



index table

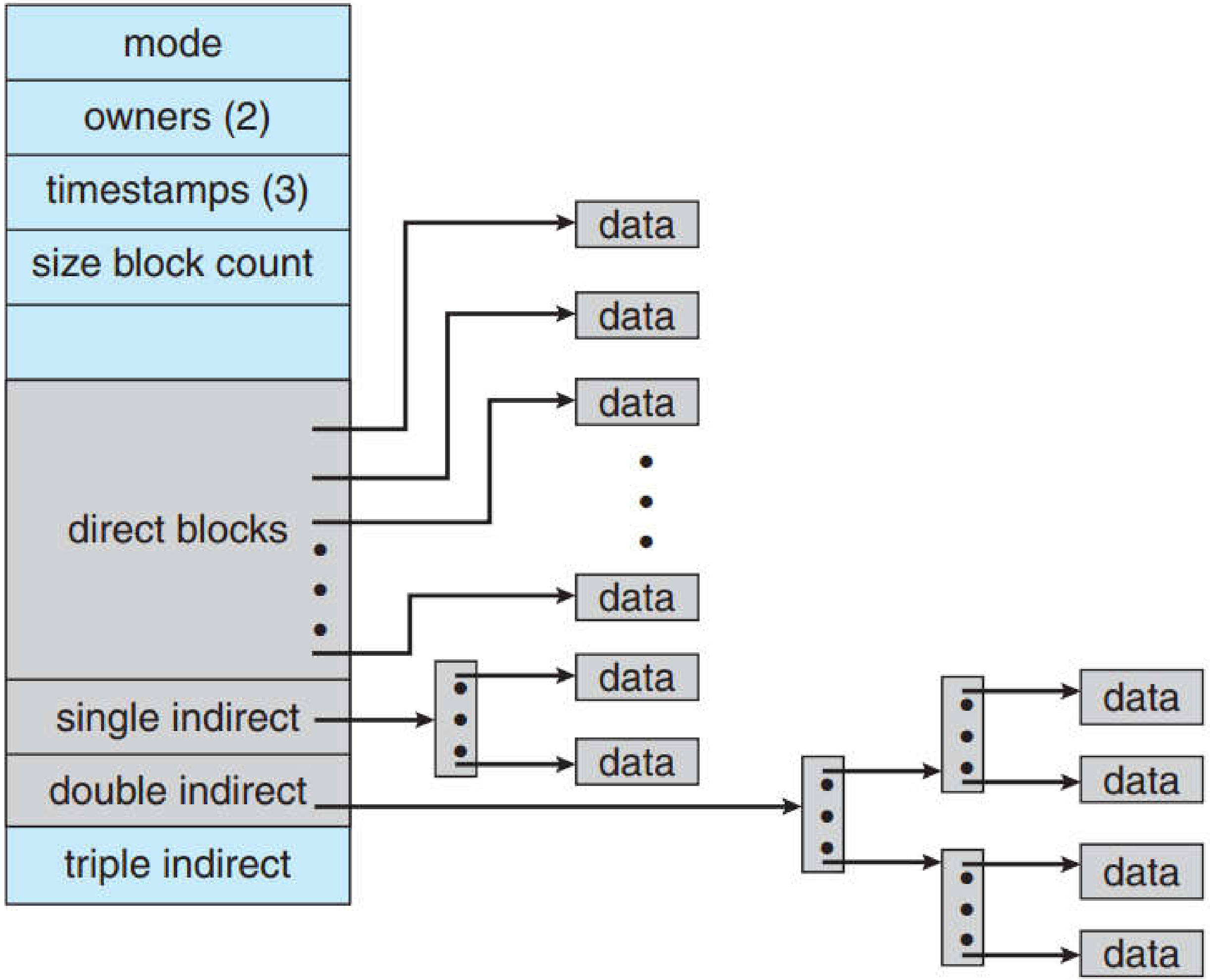
Indexed Allocation

- 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)
  - ✓ combined scheme
    - First 15 pointers are in inode
    - Direct block: first 12 pointers
    - Indirect block: next 3 pointers



The UNIX inode

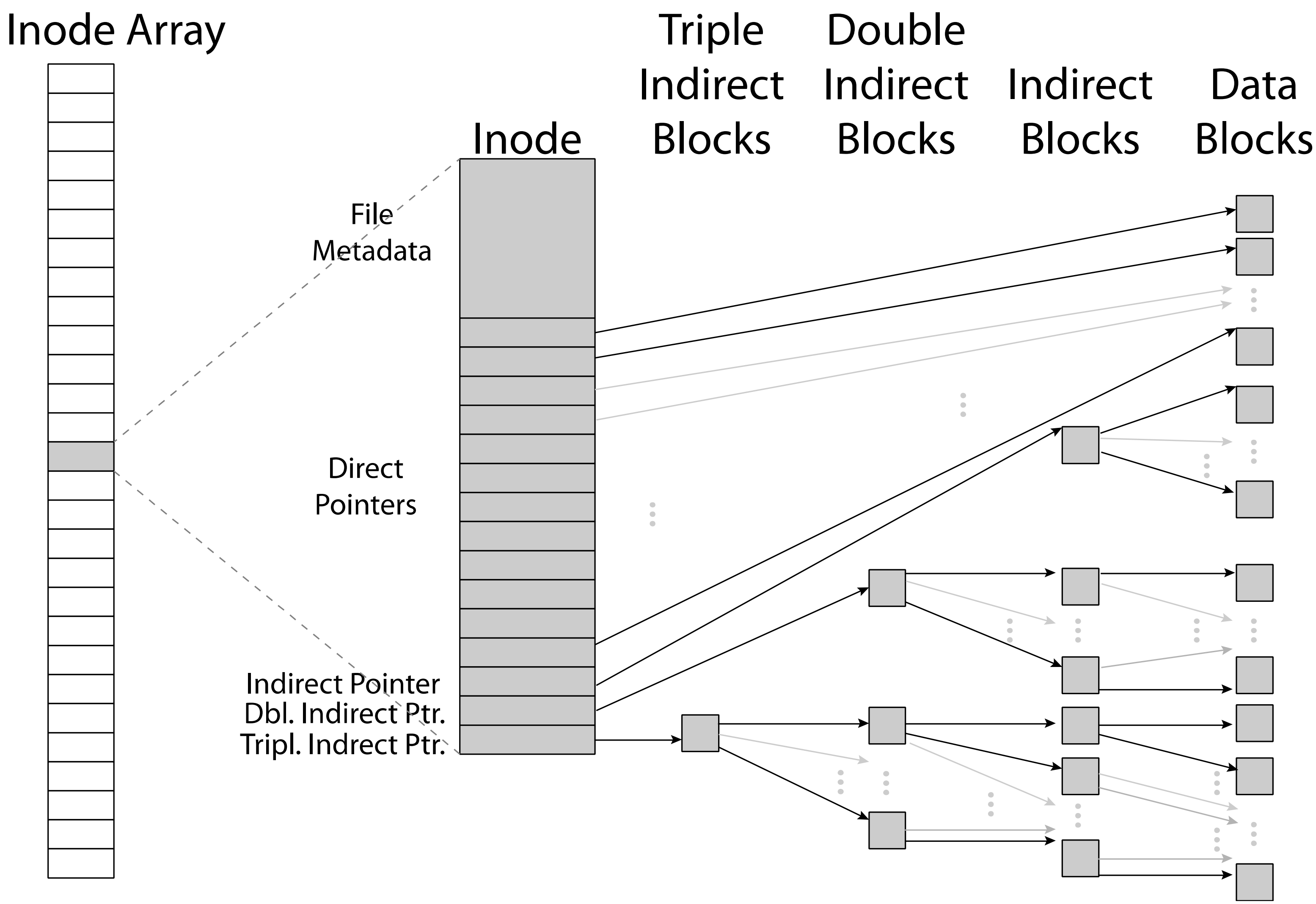
- The first 12 of these pointers point to **direct blocks**
- The next three pointers point to **indirect blocks**.
  - **a single indirect block**
  - **double indirect block**
  - **triple indirect block**



The UNIX inode

Inode Structure

- inode metadata



Data Storage

- Small files: 12 pointers direct to data blocks

Direct pointers

4kB blocks ⇒ sufficient for files up to 48KB

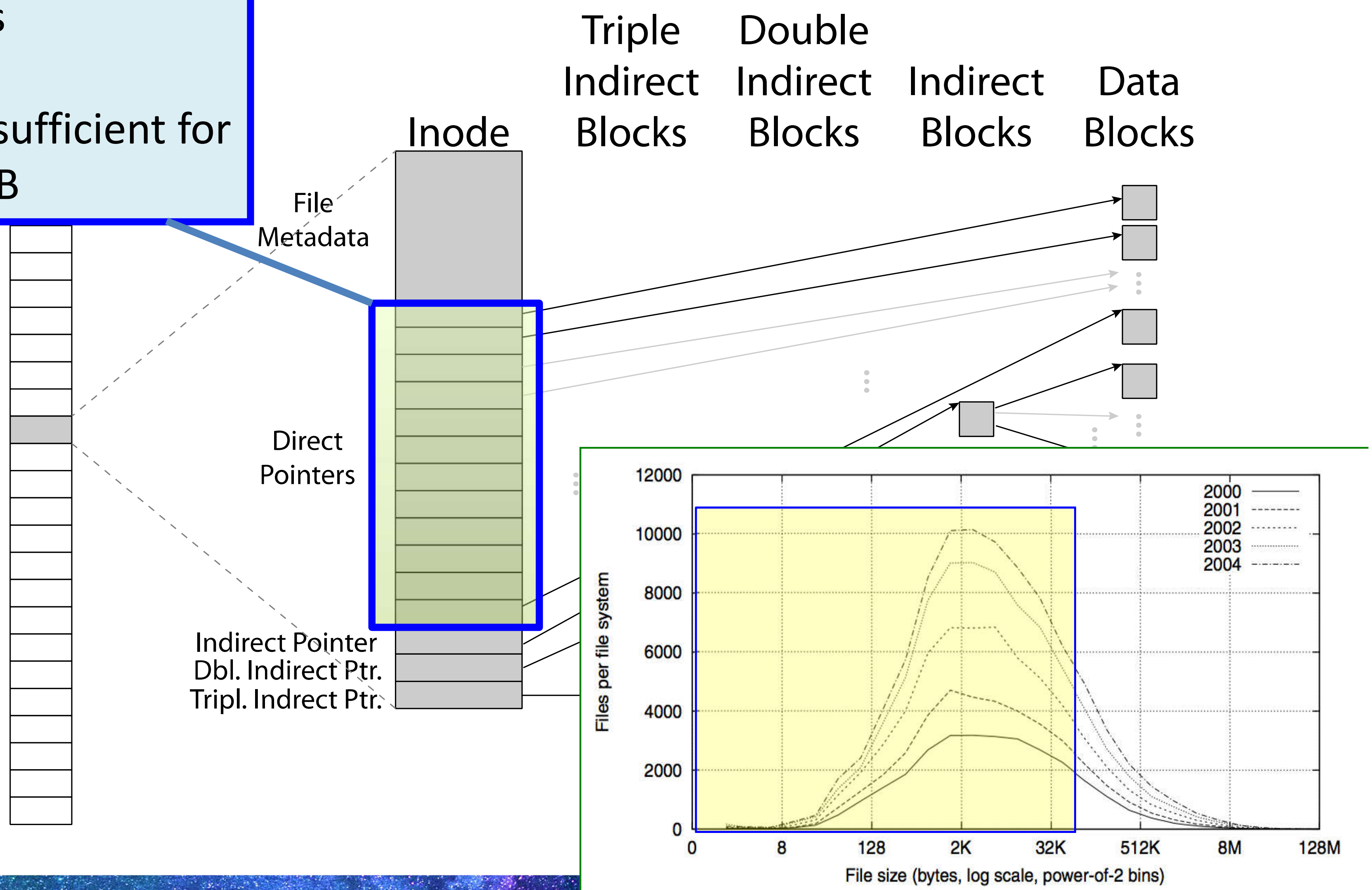
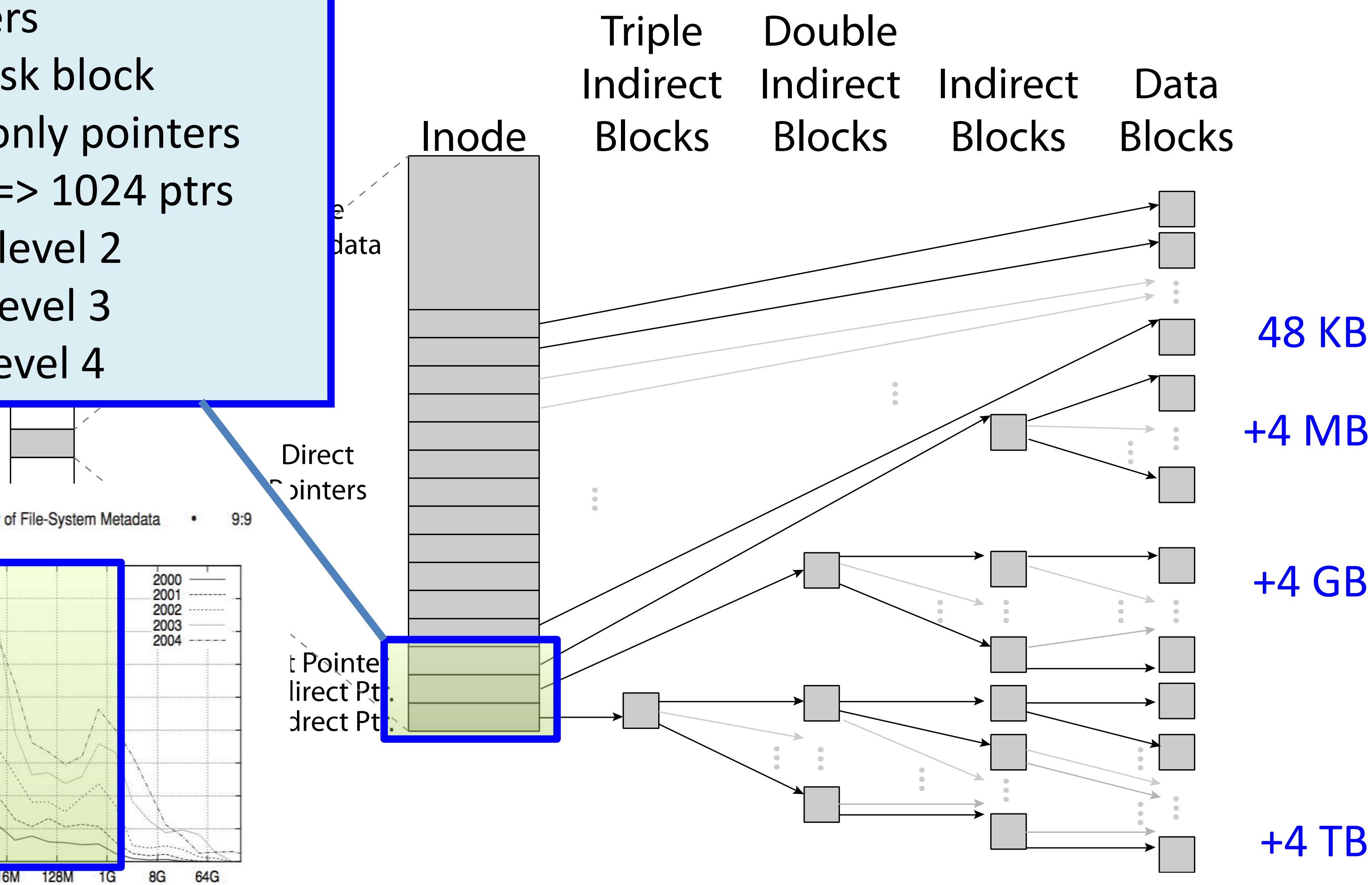


Fig. 2. Histograms of files by size.

Data Storage

- Large files: 1,2,3 level indirect pointers

Indirect pointers
- point to a disk block containing only pointers
- 4 kB blocks => 1024 ptrs
  => 4 MB @ level 2
  => 4 GB @ level 3
  => 4 TB @ level 4



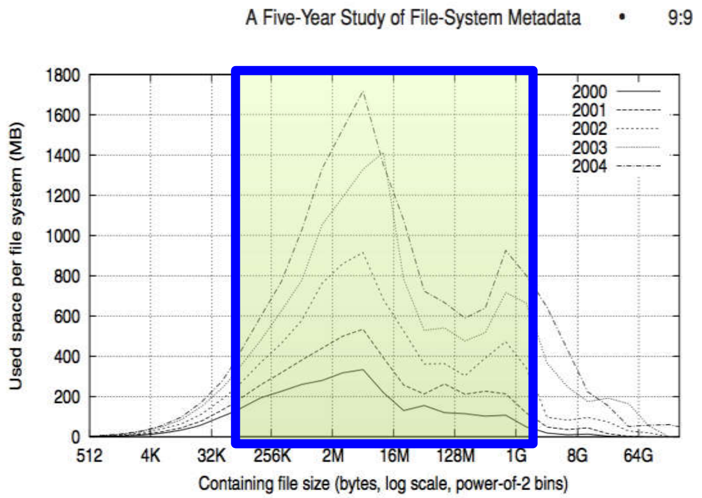


Fig. 4. Histograms of bytes by containing file size.

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
    - ✓ Disk I/O is slow, reduce as many disk I/Os as possible

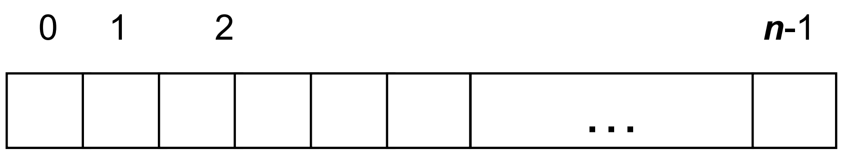
Free-Space Management

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
  - …

Free-Space Management

- File system maintains **free-space list** to track available blocks/clusters
  - (Using term “block” for simplicity)
- **Bit vector** or **bit map** (***n*** blocks)



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

CPUs have instructions to return offset within word of first “1” bit

- 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 32MB)
    if clusters of 4 blocks -> 8MB of memory
- Easy to get contiguous files

bit map example:

Block number calculation

(number of bits per word) * (number of 0-value words) + offset of first 1 bit

a bitmap of 16 bits as: **0000111000000110**

- Finding the first free block is efficient.
- It requires scanning the words (a group of 8 bits) in a bitmap for a non-zero word. (A 0-valued word has all bits 0).
- The first free block is then found by scanning for the first 1 bit in the non-zero word.

the first free block number = 8*0+5 = 5

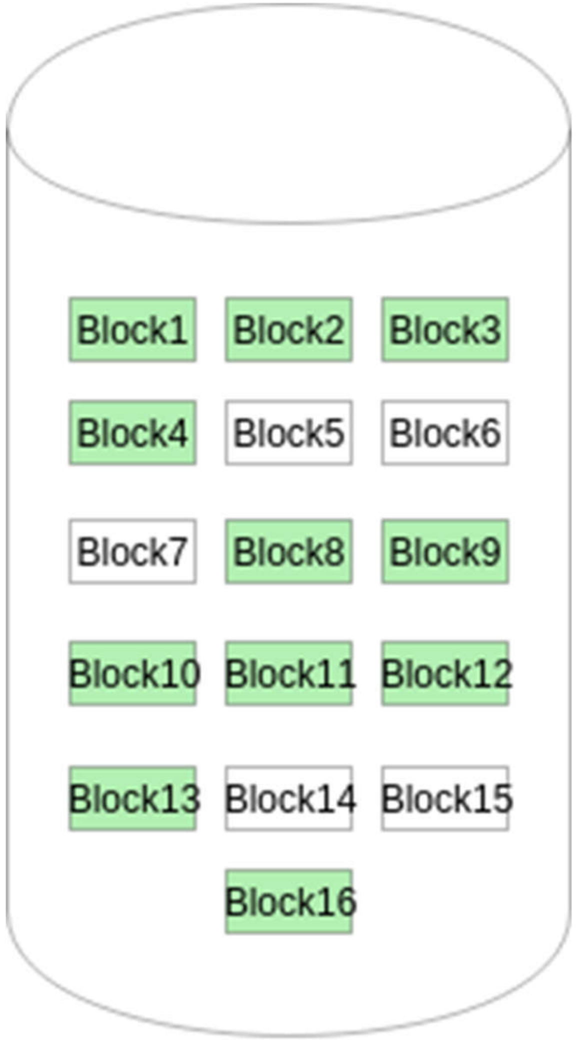
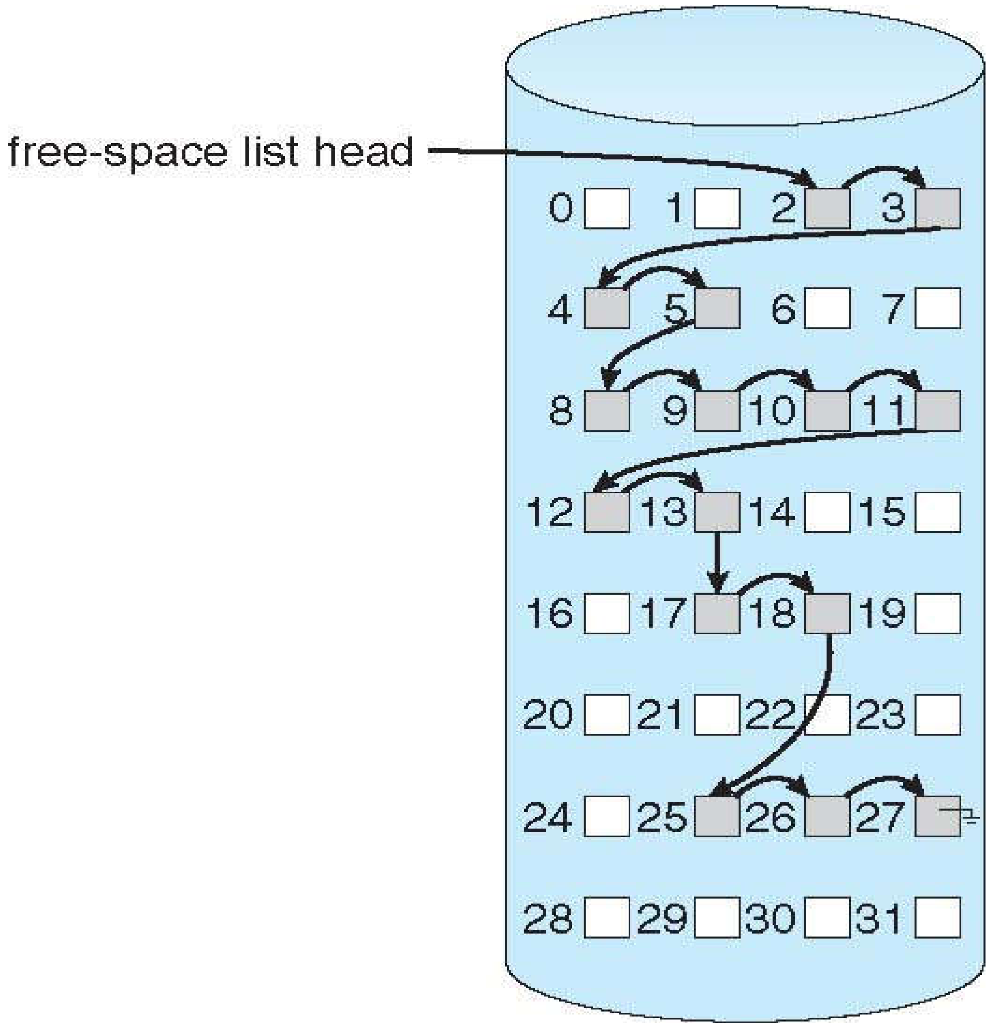


Figure - 1

Linked Free Space List on Disk

Linked list (free list)

- Cannot get contiguous space easily
- No waste of space
- No need to traverse the entire list (if # free blocks recorded)



Free-Space Management (Cont.)

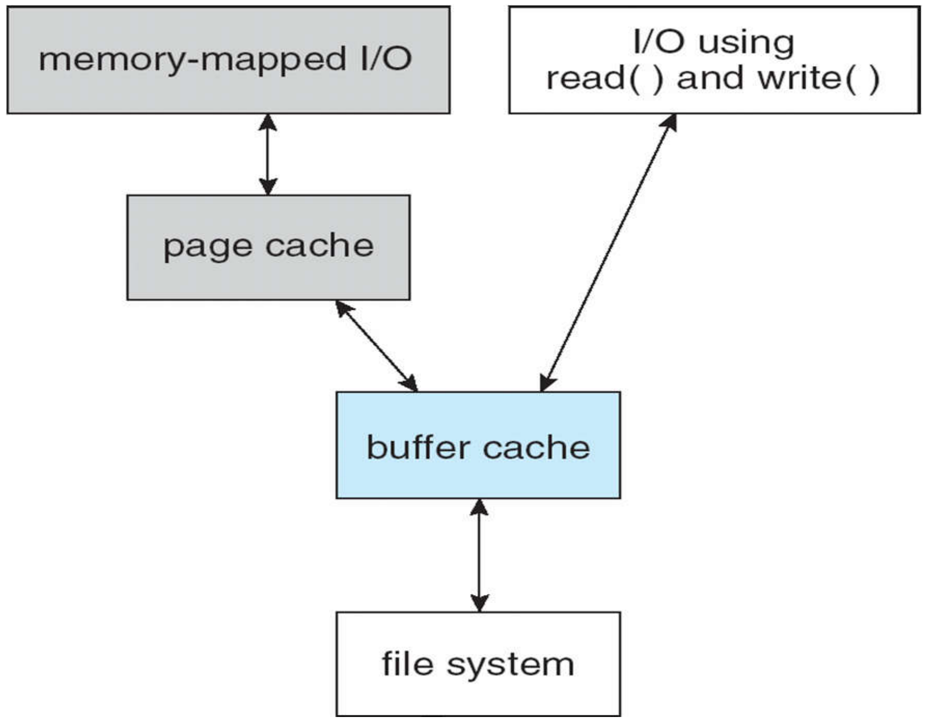
- Grouping
  - Modify linked list to store address of next *n-1* free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this one)
- Counting
  - Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering
    - Keep address of first free block and count of following free blocks
    - Free space list then has entries containing addresses and counts

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

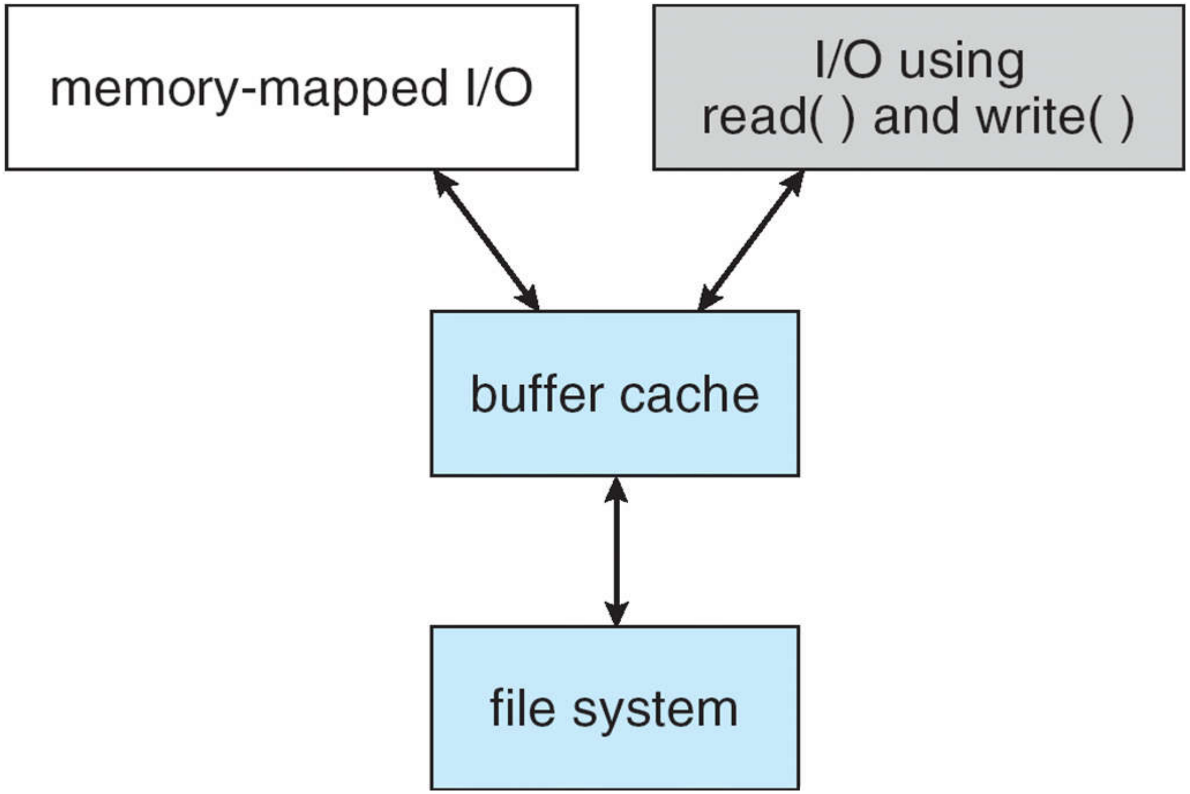
Page Cache

- 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



Unified Buffer Cache

- A **unified buffer cache** uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid **double caching**
- But which caches get priority, and what replacement algorithms to use?



Recovery

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

Questions?