File Systems

Logical view and implementation

Chapters 11+12

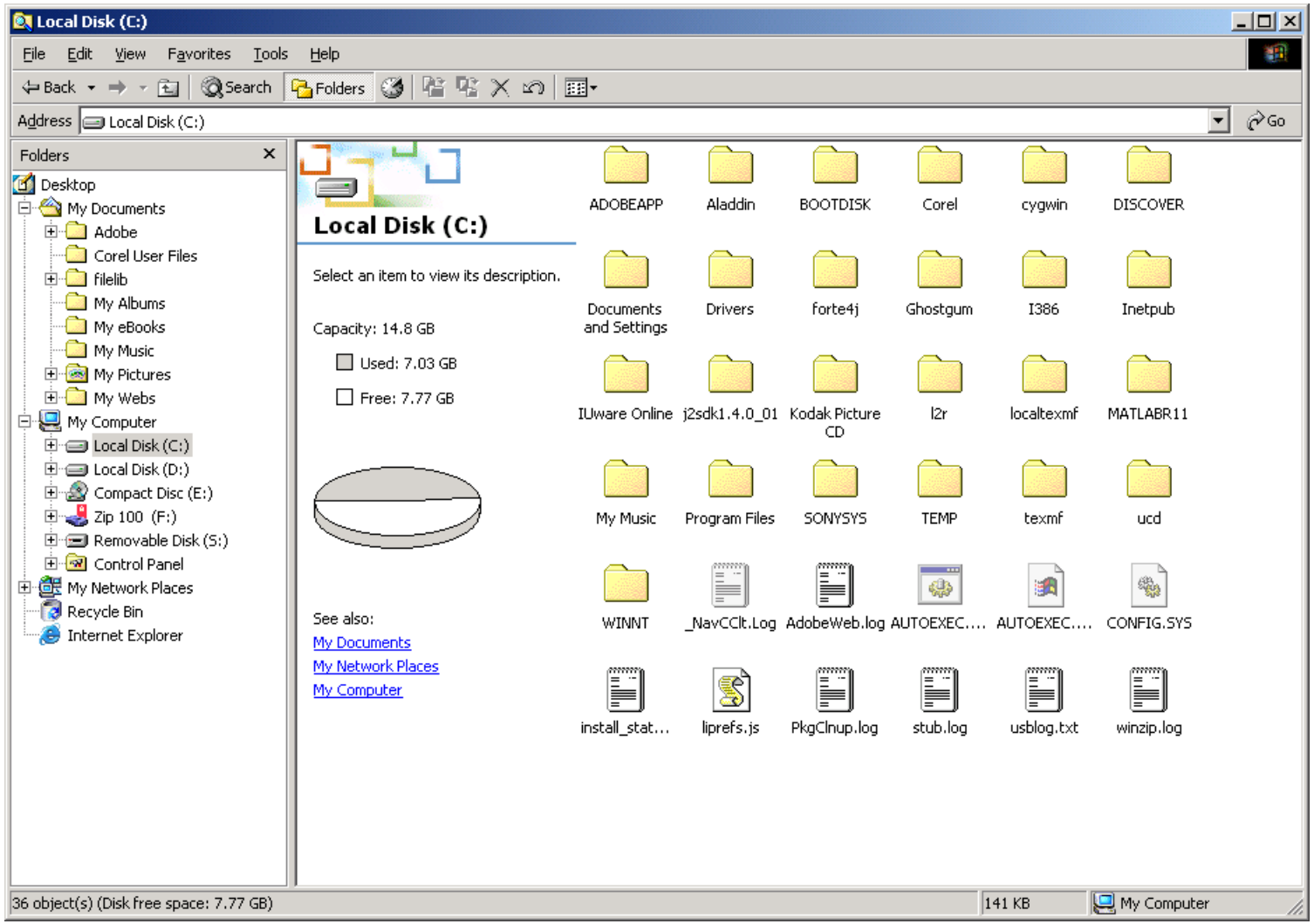
Outline

- Two parts:
  1. File system concepts/structures (Chapter 11)
  2. File system implementation details (Chapter 12)

Long-term Information Storage

1. Must store large amounts of data
2. Information stored must survive the termination of the process using it
3. Multiple processes must be able to access the information concurrently

File Systems



File system view on Windows

File Systems

```
phoenix% ls -l
total 86
-rw-------   1 tuceryan staff         76 Jan 15 14:37 ex1.c
-rw-------   1 tuceryan staff        105 Jan 15 15:29 ex2.c
-rw-------   1 tuceryan staff         77 Jan 15 15:30 ex3.c
-rw-------   1 tuceryan staff        382 Jan 30 13:44 forkex.c
-rw-------   1 tuceryan staff        187 Jan 29 22:17 myval.c
-rwx--x--x   1 tuceryan staff       6452 Feb  3 18:11 pthrex*
-rw-------   1 tuceryan staff        393 Feb  3 13:35 pthrex.c
-rwx--x--x   1 tuceryan staff      29108 Feb 16 12:04 q3*
-rw-------   1 tuceryan staff        486 Feb 16 12:04 q3.c
phoenix%
```

More detailed view of a file system view on Unix/linux

Why Files?

- Physical reality
  - Block oriented
  - Physical sector #'s
  - No protection among users of the system
  - Data might be corrupted if machine crashes
- File system model
  - Byte oriented
  - Named files
  - Users protected from each other
  - Robust to machine failures

File System Requirements

- Users must be able to:
  - create, modify, and delete files at will.
  - read, write, and modify file contents with a minimum of fuss about blocking, buffering, etc.
  - share each other's files with proper authorization
  - transfer information between files.
  - refer to files by symbolic names.
  - retrieve backup copies of files lost through accident or malicious destruction.
  - see a logical view of their files without concern for how they are stored.

File and Disk Concepts

File systems reside on disk partitions

- Partitions
  - A chunk of disk space that keeps the file system
  - Helpful for easy backups, archival, and porting of files
    - e.g., a flash drive can be removed from one machine and mounted on another. The flash drive would be a disk storage with one partition with a file system on it.
  - Different access privileges can be set for different partitions.
  - Different OS's can be installed in different partitions (e.g., dual boot machines: linux in one partition and windows in another).
  - Analogous to segmentation for virtual memory

Roadmap

- First look at logical view of files.
  - What do the users see as files?
  - How do they access and manipulate files and information in them?
- Then look at the implementation of files
  - Various methods of implementing the logical views on physical magnetic disk devices.

File Concepts – Logical view

- File concept: OS *abstracts* from the physical properties of its storage device to define a *logical* storage unit, called a *file*.
  - Files are mapped by the OS onto physical storage devices

File Attributes

- Files have attributes:
  - At least a name.
  - Files are accessed by users via their “name” attribute.
  - Other attributes include
    1. location (address on the device)
    2. Size
    3. access control information
    4. time and date information
    5. etc.

File Attributes

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Other possible file attributes:

- No real system implements all of this, but each one is implemented by some real system.

File Operations

1. Create/delete
2. Open/close
3. Read/write
4. Append
5. Seek
6. Get/set attributes
7. Rename

Could be done

- at the command interpreter, or
- via system call interface in programs

An Example Program Using File System Calls (1/2)

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                  /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* ANSI prototype */

#define BUF_SIZE 4096                   /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                /* protection bits for output file */

int main(int argc, char *argv[])
{
      int in_fd, out_fd, rd_count, wt_count;
      char buffer[BUF_SIZE];

      if (argc != 3) exit(1);           /* syntax error if argc is not 3 */
```

An Example Program Using File System Calls (2/2)

System calls Related to file systems

```
/* Open the input file and create the output file */
in_fd = open(argv[1], O_RDONLY);     /* open the source file */
if (in_fd < 0) exit(2);              /* if it cannot be opened, exit */
out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
if (out_fd < 0) exit(3);             /* if it cannot be created, exit */

/* Copy loop */
while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
if (rd_count <= 0) break;            /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4);  /* wt_count <= 0 is an error */
}

/* Close the files */
close(in_fd);
close(out_fd);
if (rd_count == 0)                   /* no error on last read */
        exit(0);
else
        exit(5);                     /* error on last read */
}
```

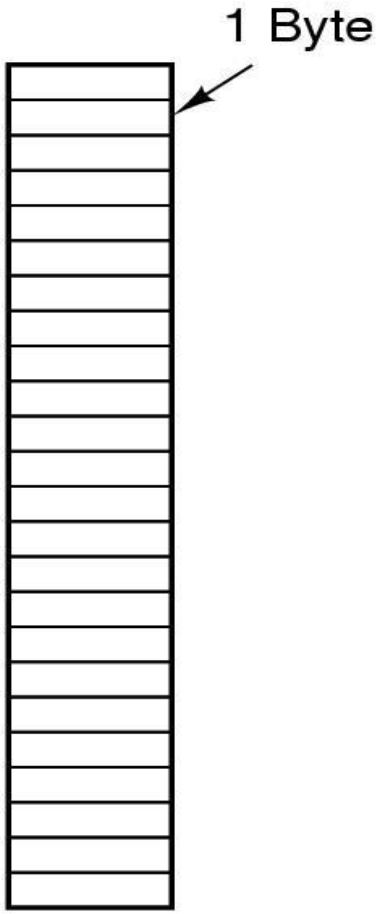
File structures (logical view)

Files are structured in various ways:

- None - sequence of words or bytes
- Simple record structure
  - Lines
  - Fixed length (e.g., 80 bytes)
  - Variable length
- Complex Structures
  - Formatted document (word files)
  - Relocatable load file
  - Can simulate last two with first method by inserting appropriate control characters.

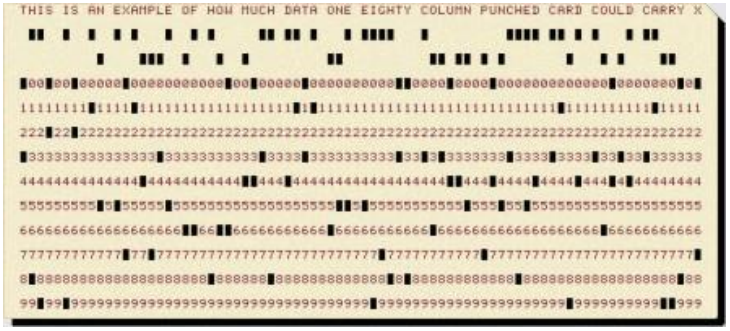
File structure: none

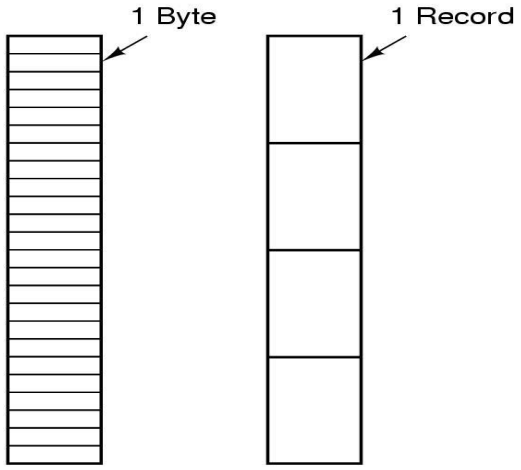
- Files consist of a sequence of bytes
  - Some mainframes and older systems might have the unit be a word instead of a byte.



File structure: records

- Files consist of a sequence of fixed length records
  - Originally on mainframes that mapped the record size to a punched card on input files (1 record = 80 columns) or to a printer line (1 record = 132 columns).

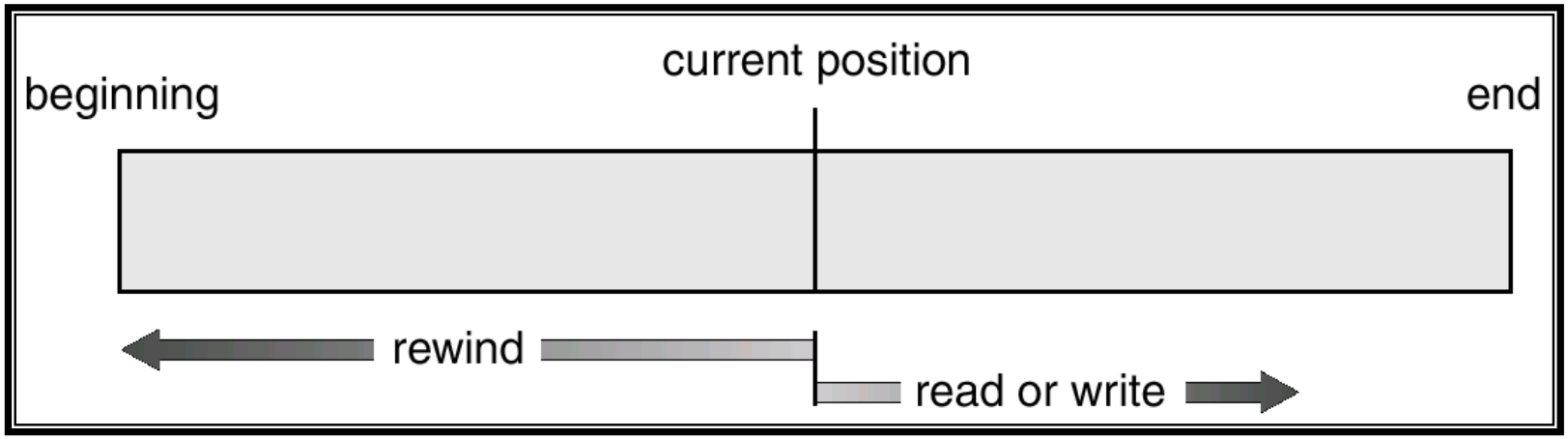




File Access types (logical view)

- Sequential access
  - read all bytes/records from the beginning
  - cannot jump around, could rewind or back up
  - convenient when medium was mag tape
- Random access
  - bytes/records read in any order
  - essential for data base systems
  - read can be …
    - move file marker (seek), then read or …
    - read and then move file marker

Sequential-access File

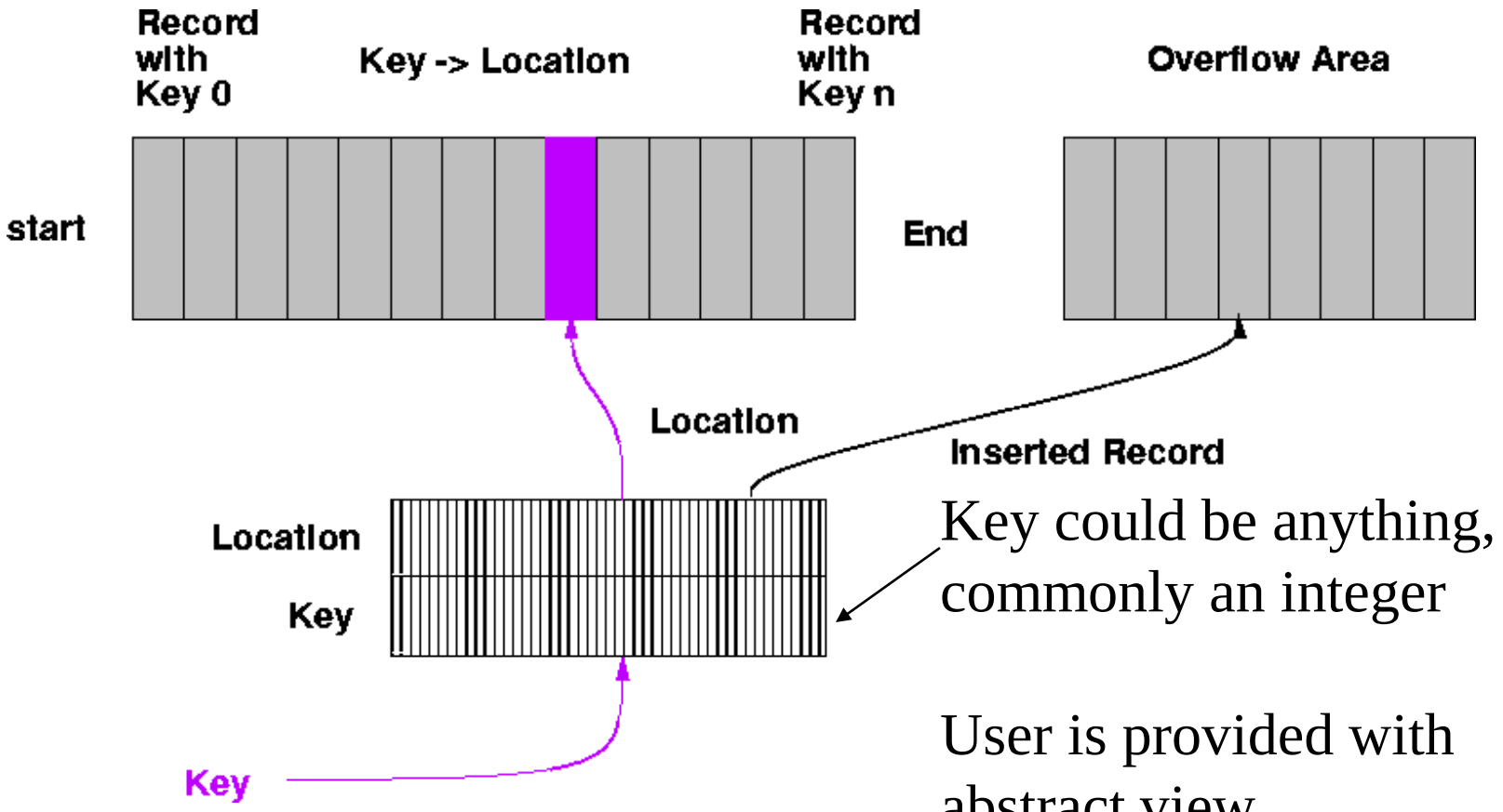


Cannot write in the middle!

Indexed Sequential or Indexed File Organization

- Each record has an identifying key
- System maintains an index of keys
- Records are stored in logical order on track
- Inserted records stored in overflow area
- Access sequential or random
- When random access, must be on direct access device [e.g., disk--cannot be tape]

Indexed Sequential or Indexed File Organization



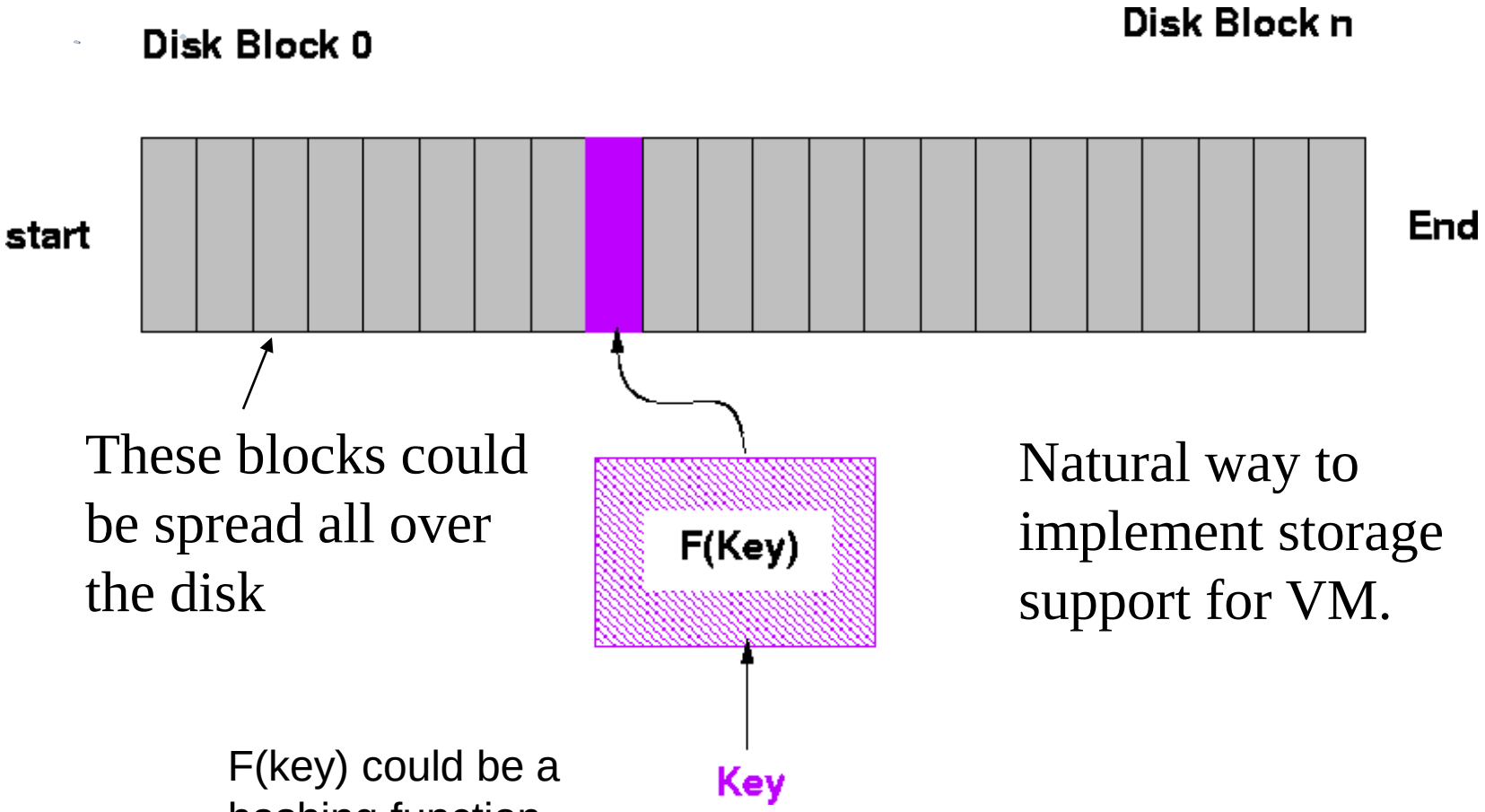
Key could be anything, commonly an integer

User is provided with abstract view

Direct Access File Organization

- User maps key into disk address
- Access in physical order or random
- No notion of next record [because there is no index table]
- Good for VM implementations
- Good for database implementations

Direct Access File Organization



Directory Structure Organization

- Maps symbolic names into logical file names
- Directories are used to organize and access files in a file system.

Directory Operations

1. Create/delete
2. Open/close
3. Read
4. Rename
5. List
6. Link/Unlink
7. Traverse (change directory)
8. Search

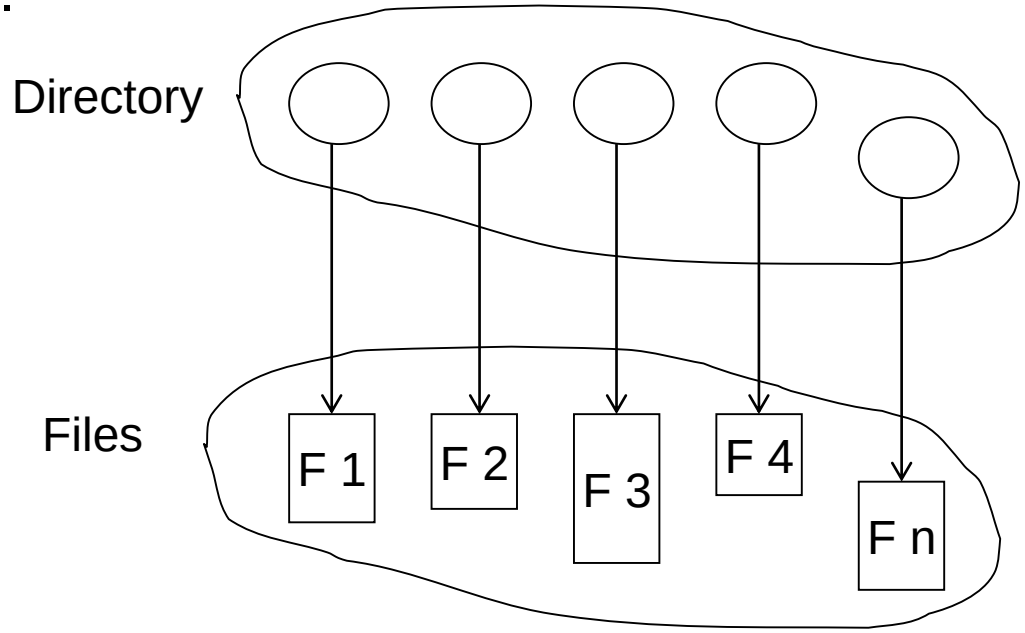
- Once again, system call interface provided so most of these ops can be done from within programs.

Directory Contents

- File name -- symbolic name
- File type indicates format of file
- Size
- Protection
- Creation, access, and modification date
- Owner identification
- The following items may be stored on a per file, process basis
  - Current read, write position
  - Usage count
- Some of these can be stored
  - In the directory entry for the file
  - Or in the file itself.
  - Different OS's implement them differently (we'll see).

Directory Structure

- A collection of nodes containing information about all files.



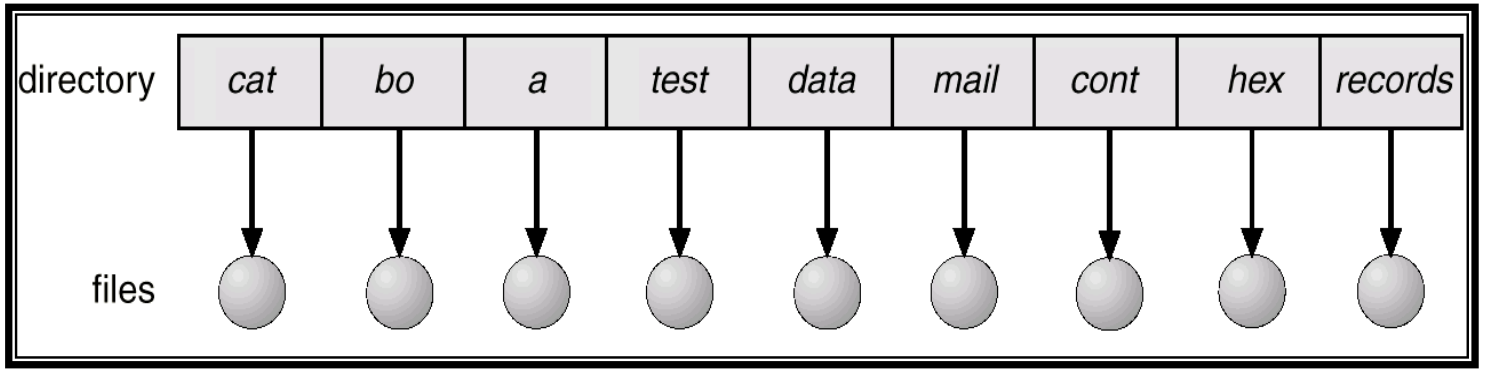
- Both the directory structure and the files reside on disk.
- Backups of these two structures can be kept on tapes.

Organize the Directory (Logically) to Obtain

- 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 Systems

- A single directory for all users (very old systems).



Problems With Single Level Directory

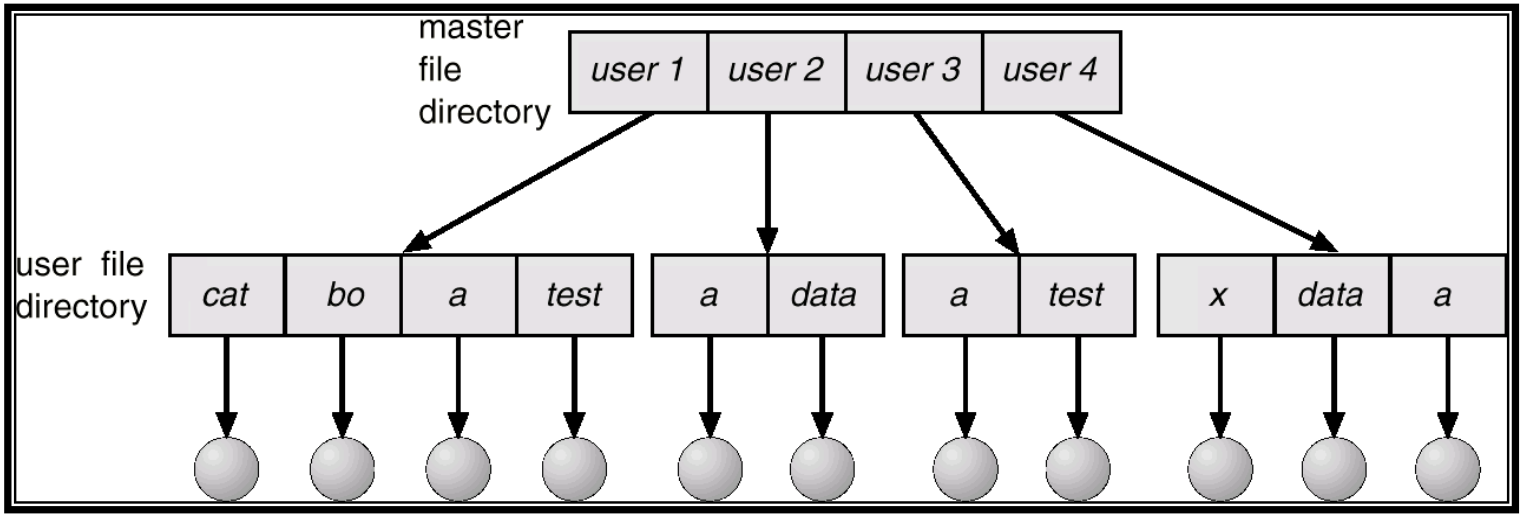
- Difficulty when there is more than one user
- Name clashes
- Lack of modularity
- Problems with large file systems
- Moving files from one system to another

Two-level Directory

- Introduced to remove naming problems between users
- First level contains list of user directories
- Second level contains user files
- System files kept in separate directory or level 1
- Sharing accomplished by naming other users files

Two-Level Directory

- Separate directory for each user.

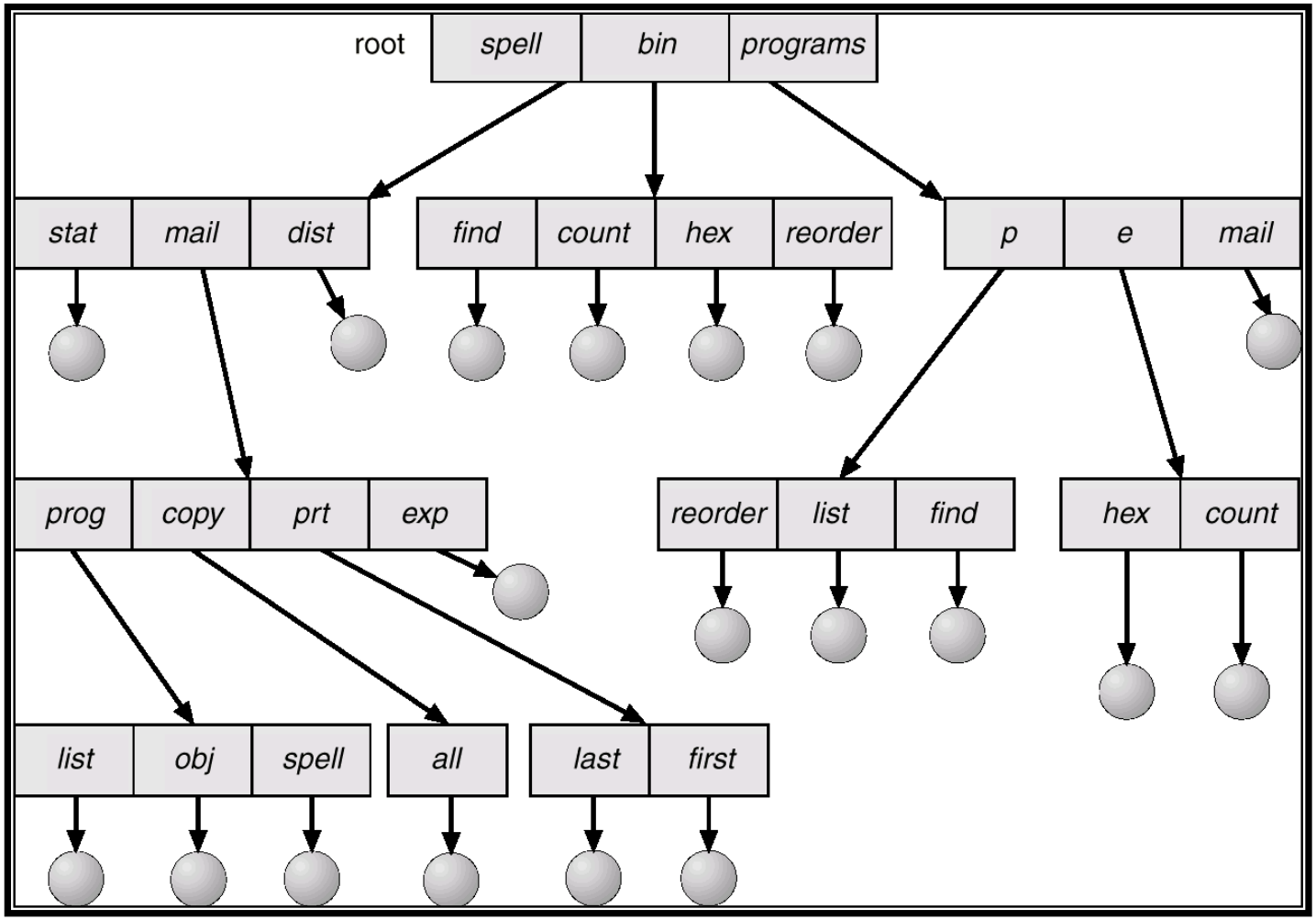


- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability for one user

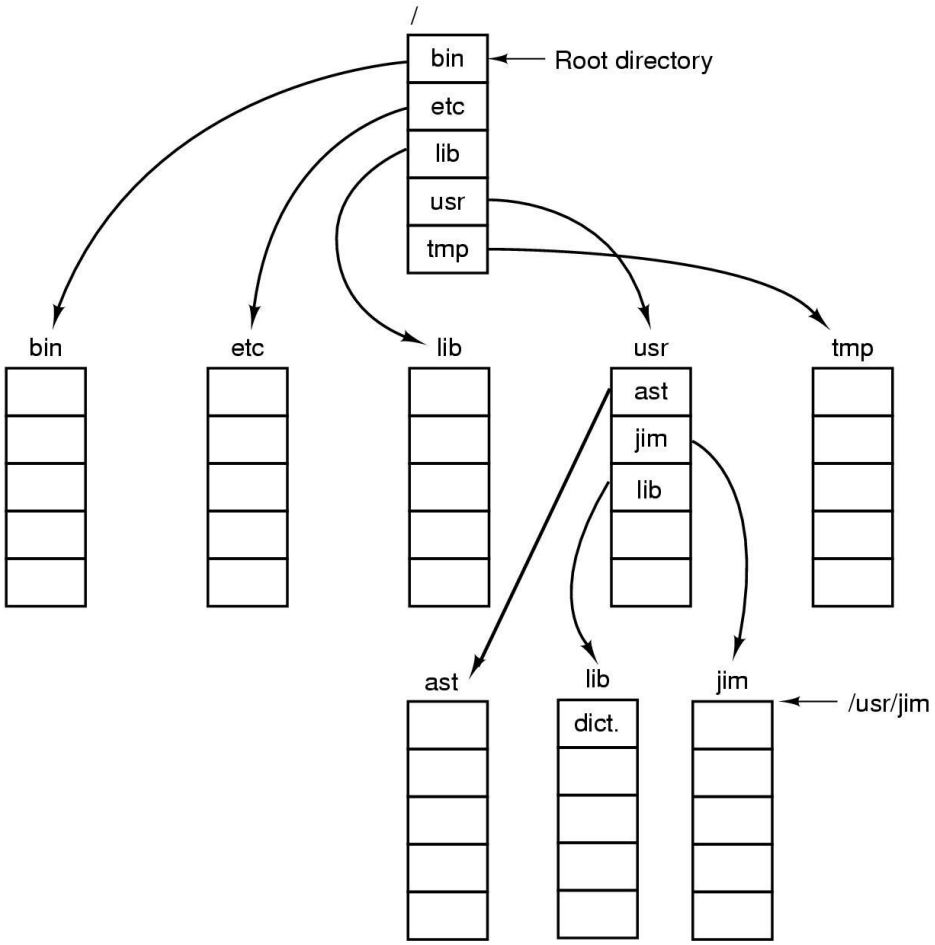
Tree Structured Directories

- Arbitrary depth of directories
- Leaf nodes are files
- Interior nodes are directories
- Path name lists nodes in tree to traverse to find node
- Use absolute paths from root
- Use relative paths from current working directory pointer

Tree-Structured Directories



Path Names



A UNIX directory tree

Tree-Structured Directories

- Efficient searching
- Grouping Capability
- Current directory (working directory)
  - **cd** /spell/mail/prog
  - **type** list

Tree-Structured Directories

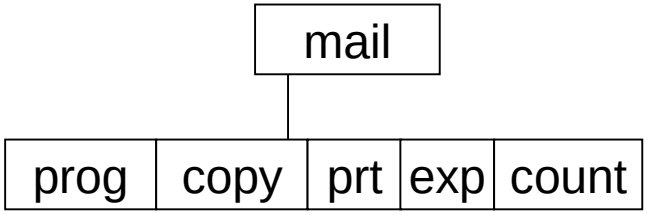
- **Absolute** or **relative** path name
- Creating a new file is done in current directory.
- Delete a file

  rm <file-name>
- Creating a new subdirectory is done in current directory.

  mkdir <dir-name>

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

  mkdir count



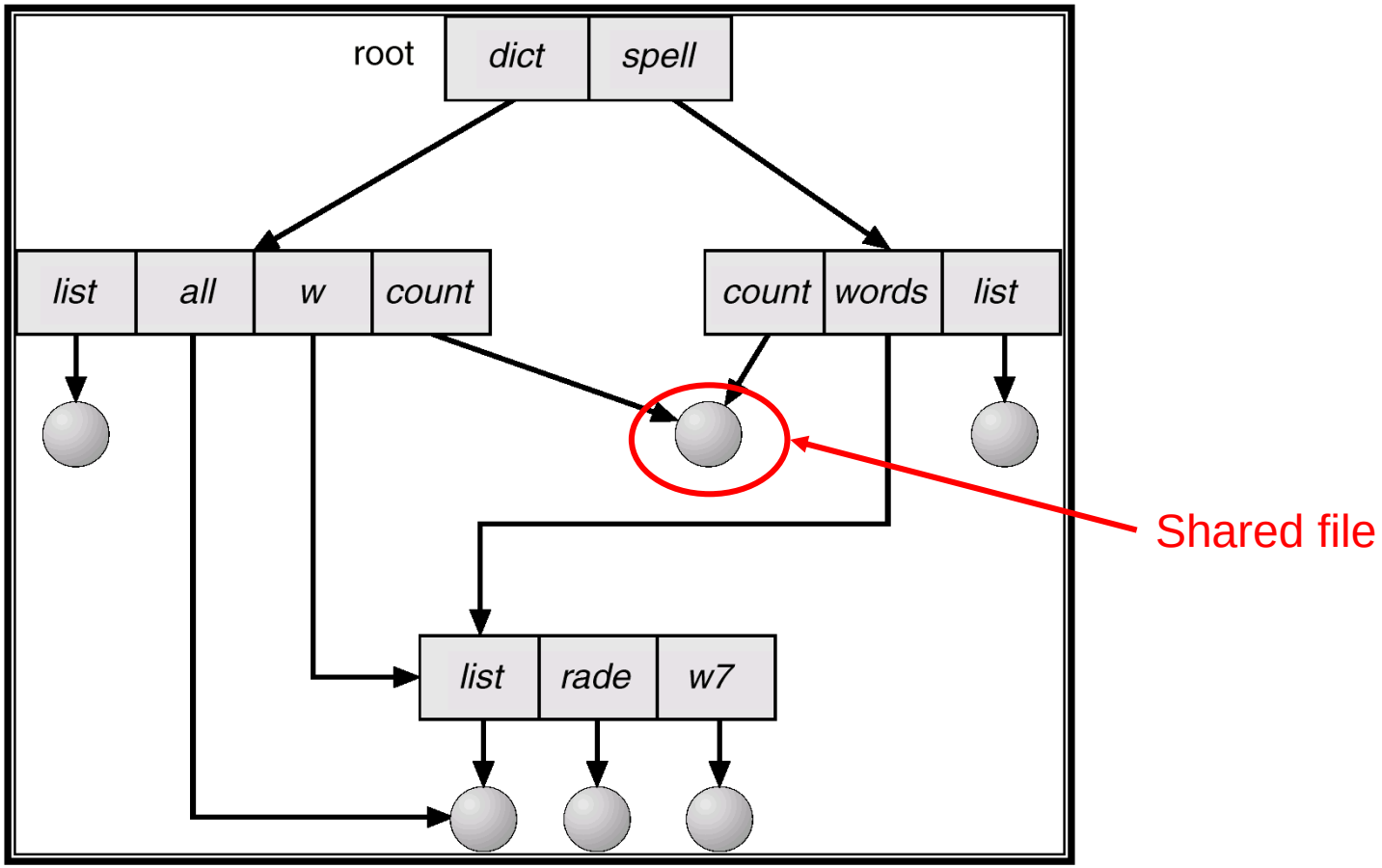
Deleting “mail” ⇒ deleting the entire subtree rooted by “mail”.

Directed Acyclic Graph (DAG) Structured Directories

- Property of a tree?
- Acyclic graphs allow sharing
- Two users can name same file
- Implementation by links - use logical names of files (file system and file)
- Implementation by symbolic links map pathname into a new pathname
- Duplicate paths complicates backup copies
- Need reference counts for hard links (as opposed to soft links or symbolic links)

DAG Directories

- Have shared subdirectories and files.



DAG Directories

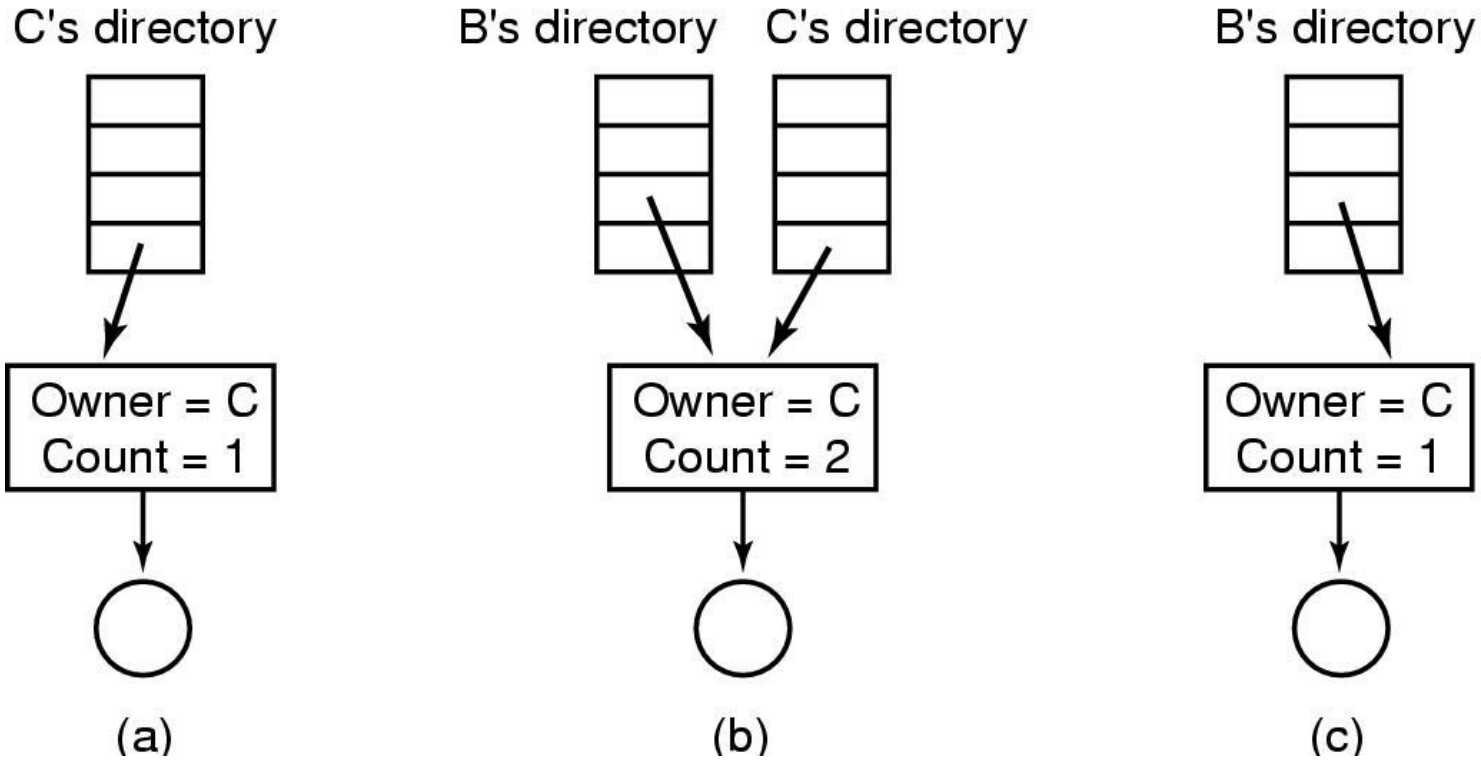
- Two different names (aliasing)

- If *dict* deletes *list* $\Rightarrow$ dangling pointer.

  Solutions:

  - Backpointers, so we can delete all pointers. Variable size records a problem.
  - Reference count mechanism (when refcount = 0 delete file).

Sharing example with reference counts



(a) Situation prior to linking

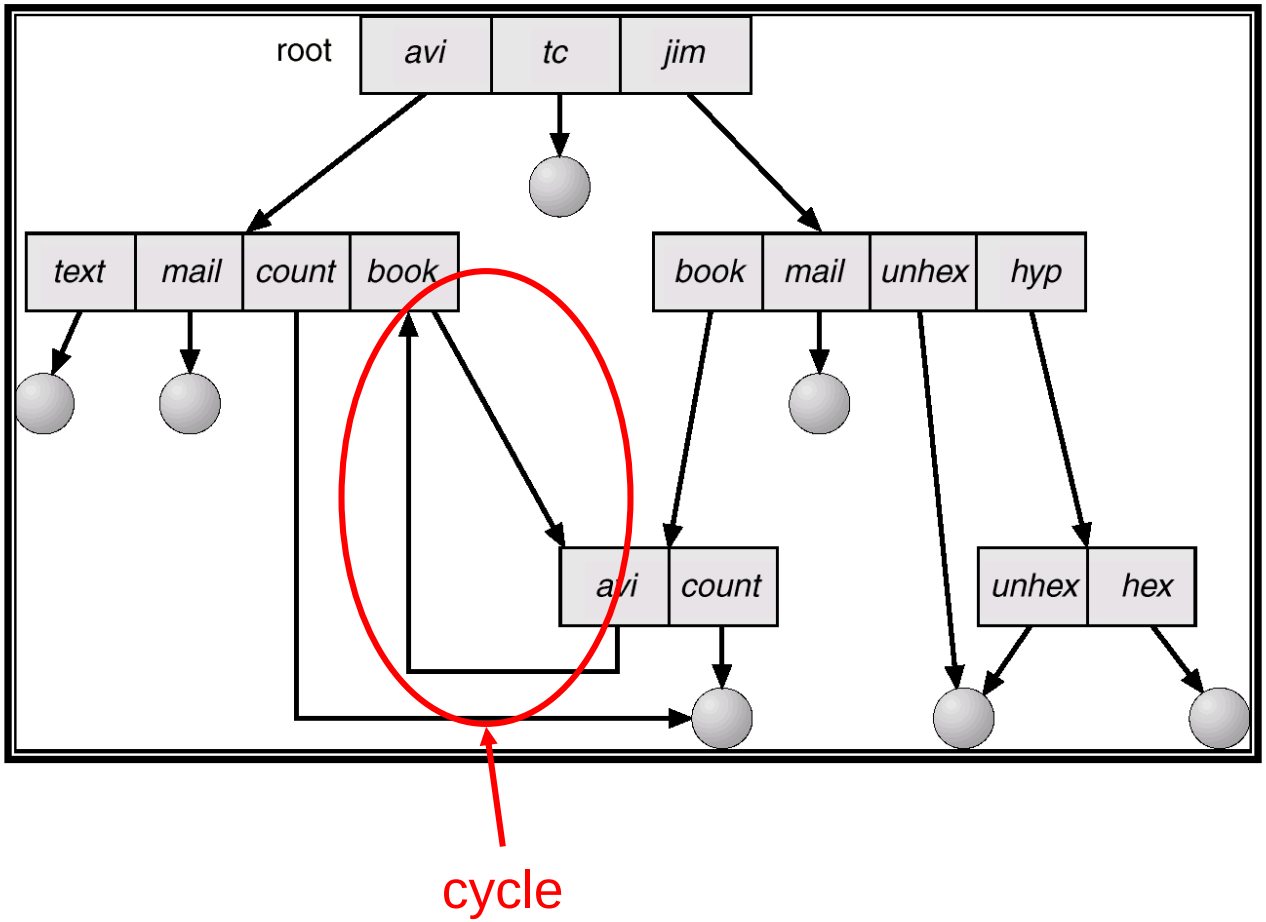
(b) After the link is created

(c) After the original owner removes the file

General Graph Structured Directories

- Cycles possible
- More flexible
- More costly
  - implementation costly
  - one can get lost in the traversal
- Need garbage collection (unused circular structures take up space)
- Must prevent infinite searches

General Graph Directory



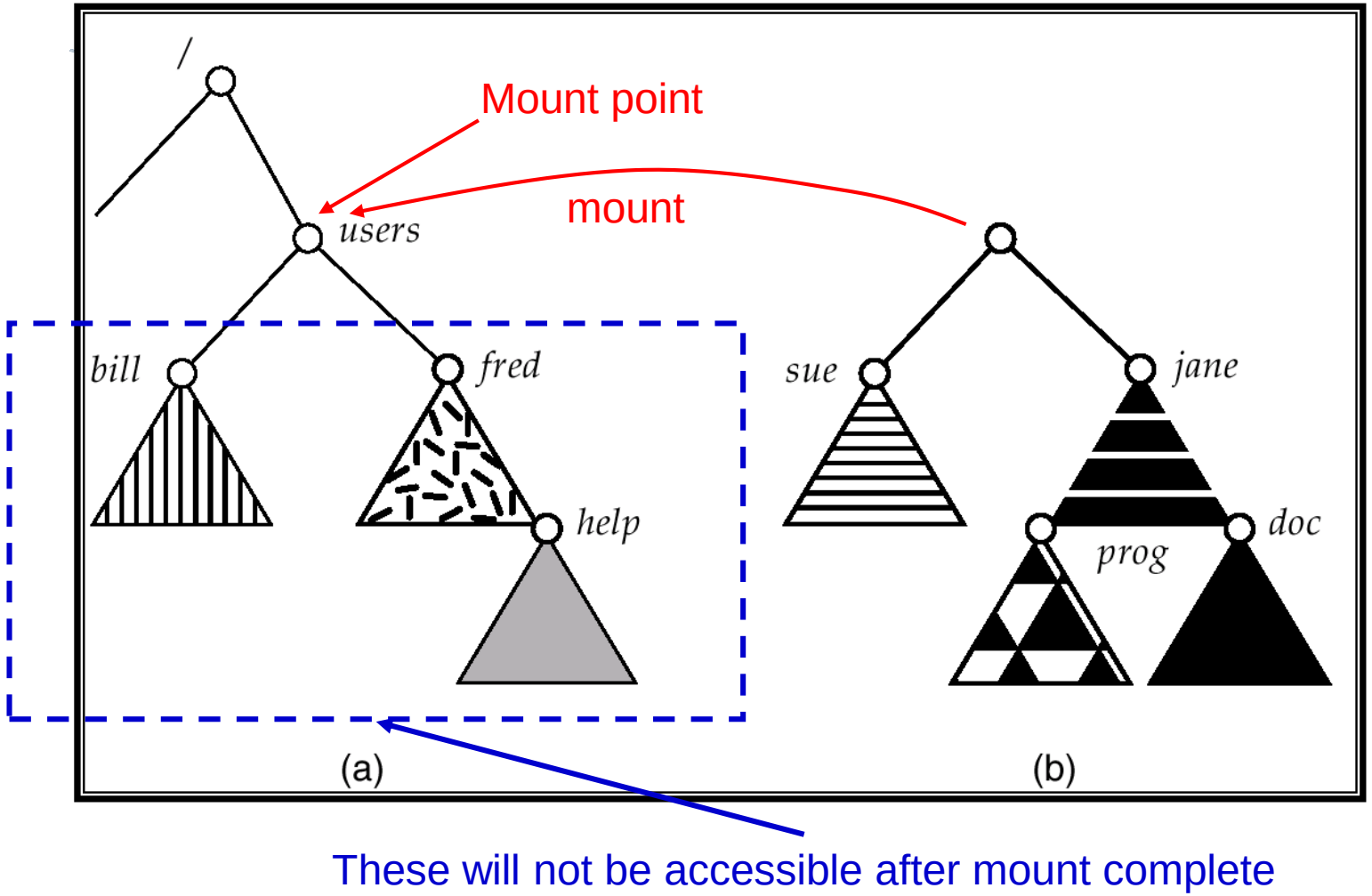
General Graph Directory

- If we don't want cycles, how do we guarantee no cycles?
  - Allow links only to files, not to subdirectories.
  - Garbage collection.
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK. (expensive)

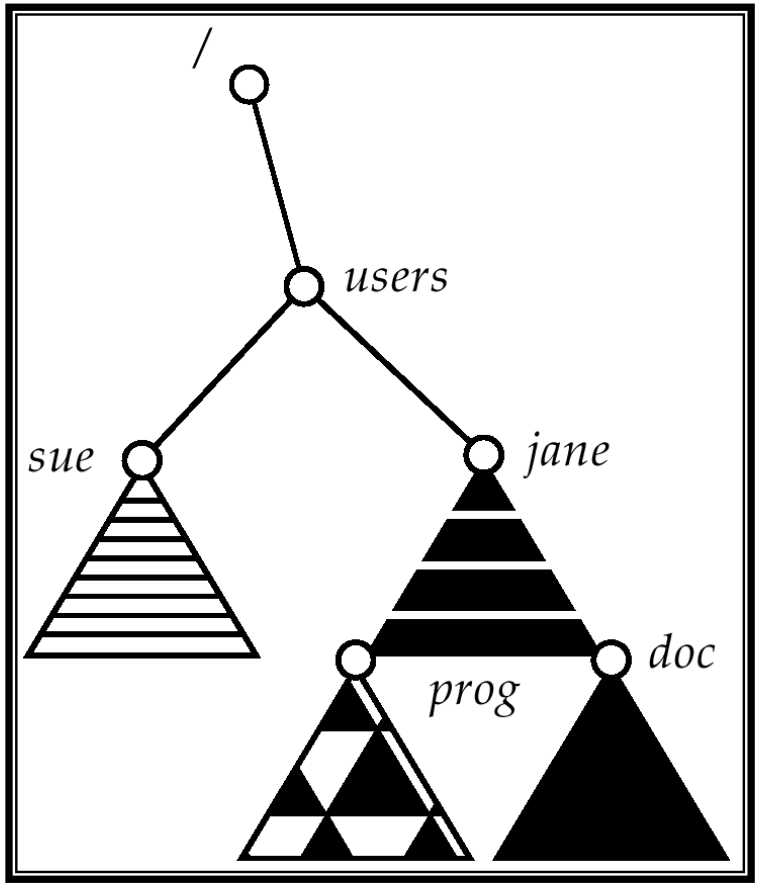
File System Mounting

- A file system must be **mounted** before it can be accessed.
- A unmounted file system (i.e., (b) in Figure on next slide) is mounted at a **mount point**.

(a) Existing. (b) Unmounted Partition



After mount

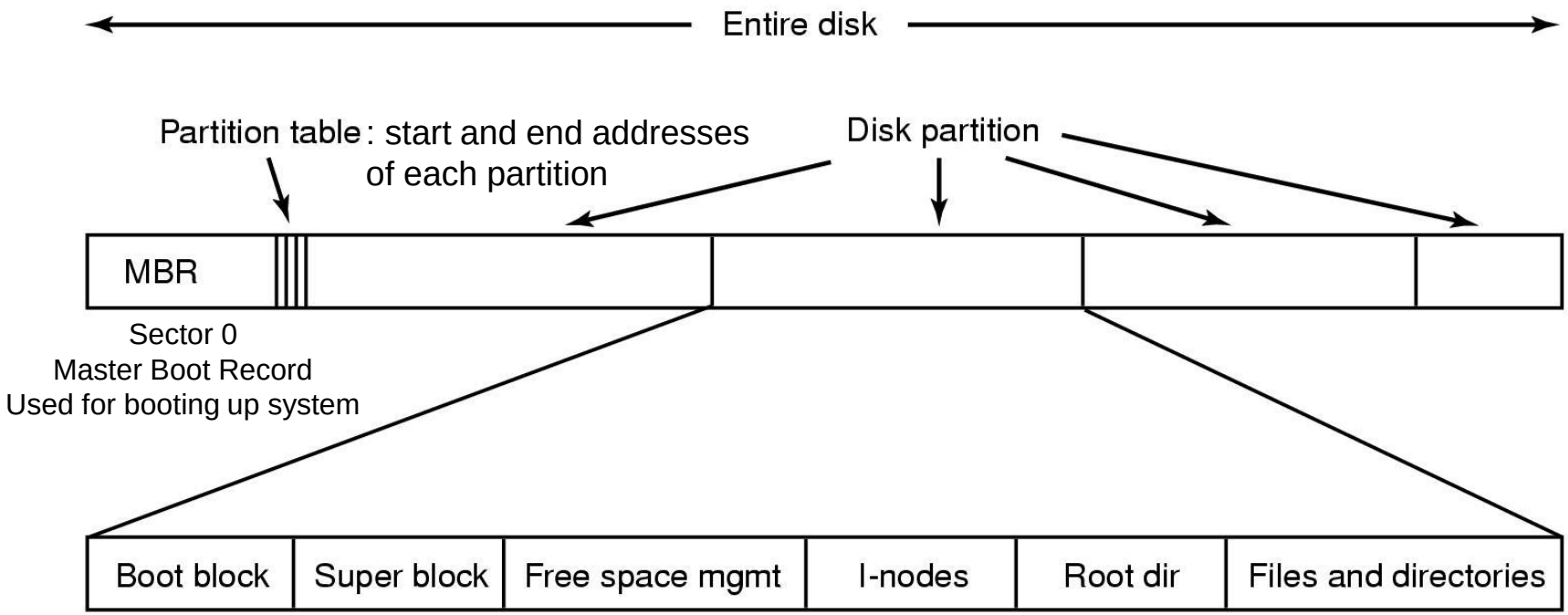


File System Implementation

File system implementation

- Organization of physical disk device
- Allocation strategies
- Data structures for various allocation strategies

A possible file system layout



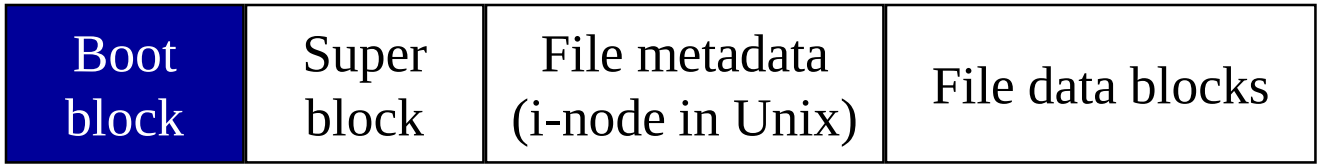
MBR

- MBR (Master Boot Record) is used for system boot.
- BIOS reads and executes MBR
  - BIOS in non-volatile memory:
    - in the past ROM
    - these days flash memory
- Program in MBR locates active partition, reads in its first block and executes.
- This initiates the boot up process: The program in the boot block loads the operating system in that partition.
- For uniformity, every partition starts with a boot block (even if it doesn't contain an OS; the boot partition is reserved).

Boot block	Super block	File metadata (i-node in Unix)	File data blocks

Superblock

- Superblock contains all the key parameters about the file system (defines a file system)
  - size of the file system
  - size of the file descriptor area
  - free list pointer, or pointer to bitmap
  - location of the file descriptor of the root directory
  - other meta-data such as permission and various times
- Superblock is read into memory at boot time.
- For reliability, superblock is replicated
  - If superblock is lost or damaged, very difficult to recover files system.



File implementation

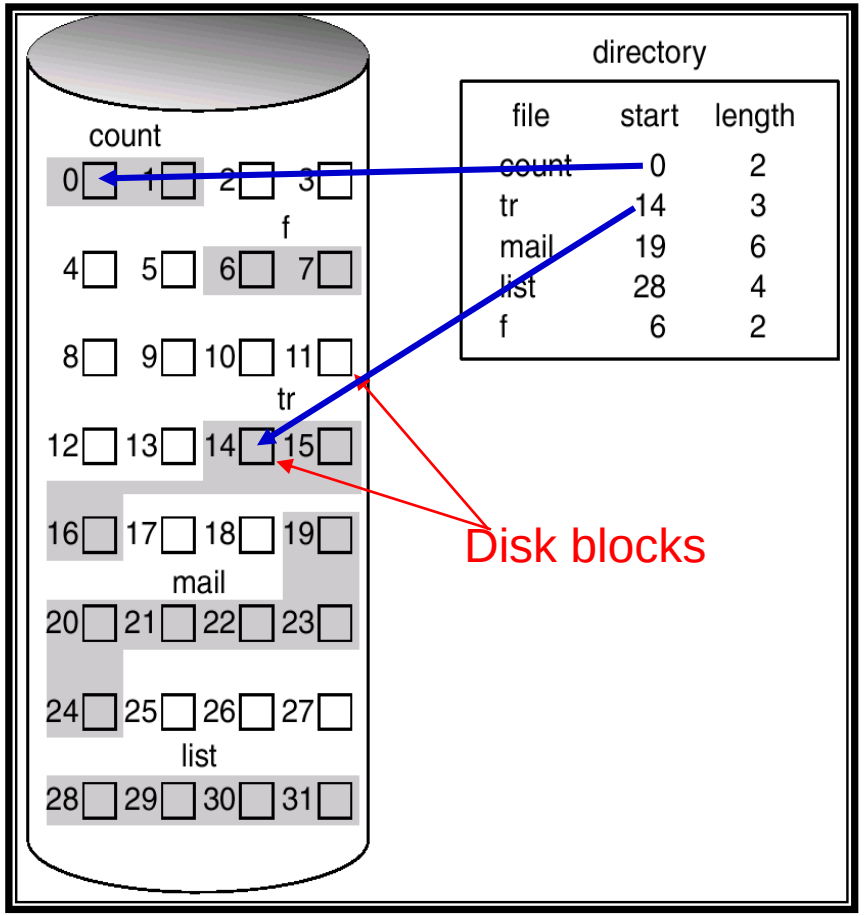
- Files consist of data blocks
- Data blocks are organized and allocated in a particular way.
  - Allocation strategies.

Allocation Methods

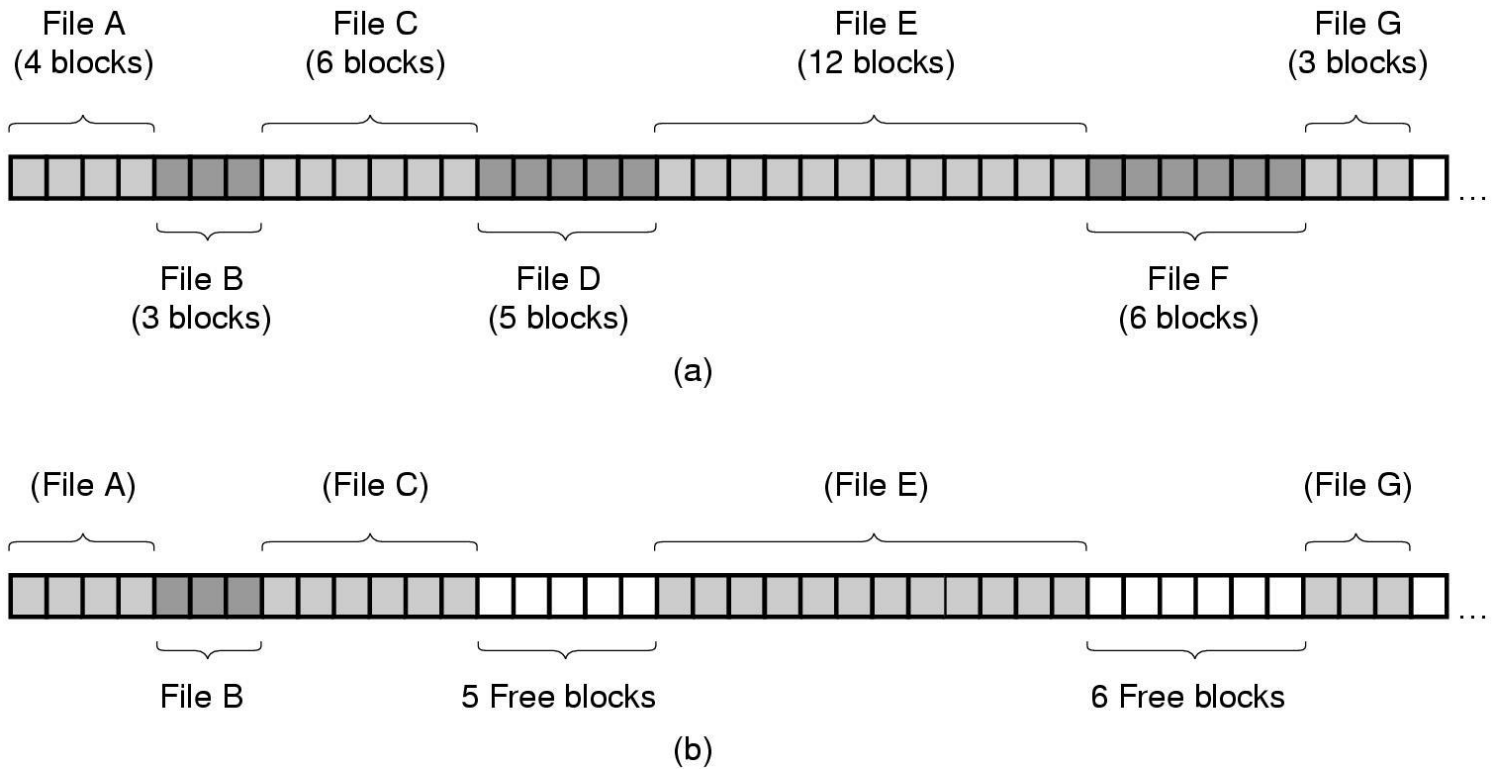
- An allocation method refers to how disk blocks are allocated for files:
  - Contiguous allocation
  - Linked allocation
  - Indexed allocation or block allocation (have a block that indexes other blocks)
- Low level access methods depend upon the disk allocation scheme used to store file data

Contiguous Allocation of Disk Space

- Each file occupies *a set of contiguous blocks* on the disk.



Contiguous Allocation Example



(a) Contiguous allocation of disk space for 7 files

(b) State of the disk after files *D* and *F* have been removed

Contiguous Allocation

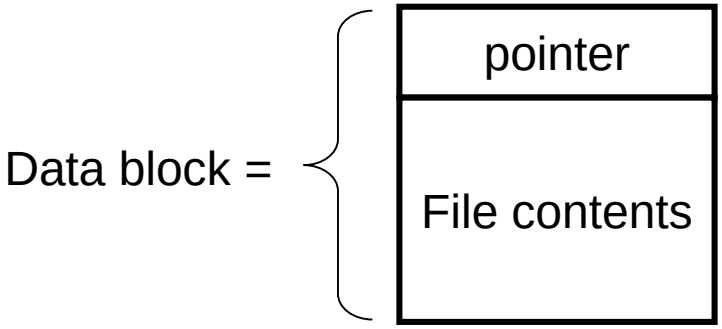
- Advantages
  - *Simple* – only starting location (block #) and length (number of blocks) are required.
  - Random access easy. Any block in the file can be accessed by adding the offset to starting location.
  - Easier to recover if directory entry is corrupted.
- Disadvantages
  - *External fragmentation* and the need to know the final file size at time of creation.
  - Users tend to overestimate space ➔ internal fragmentation
  - Expanding the file requires copying.
  - Dynamic storage-allocation - first fit, best fit
    - how to choose where to place a file?
- Older systems (mainframes) used this method because it was simple and had good performance, but was dropped later because of unfriendly user interface.
- Now it's used in WORM media such as CD-ROM's and DVD's.
  - All the file sizes are known at the time CD-ROM's are created, and they will not change again. Fragmentation does not become an issue.
  - Example of old techniques becoming more relevant because of new technology.

Contiguous Allocation Issues

- Access method suits sequential and direct access
- Directory table maps files into starting physical address and length (real simple)
- Easy to recover in event of system crash
  - in the file descriptor have head and tail addresses
  - when crash occurs, look for head and tail. Once either head or tail is found, the file is easily recovered, because it is contiguous.
- Fast, often requires no head movement and when it does, head only moves one track or switch platter

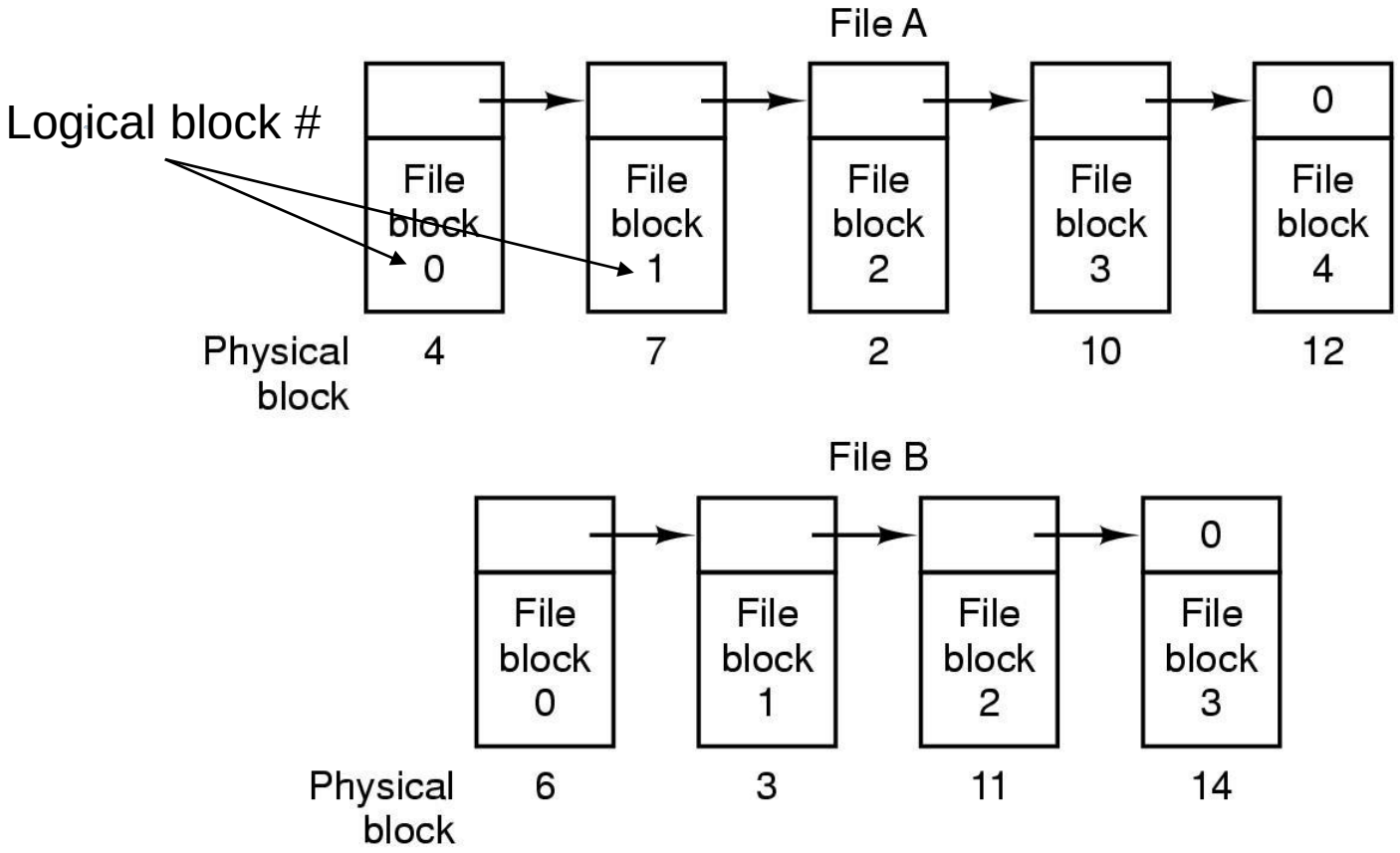
Linked Allocation

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



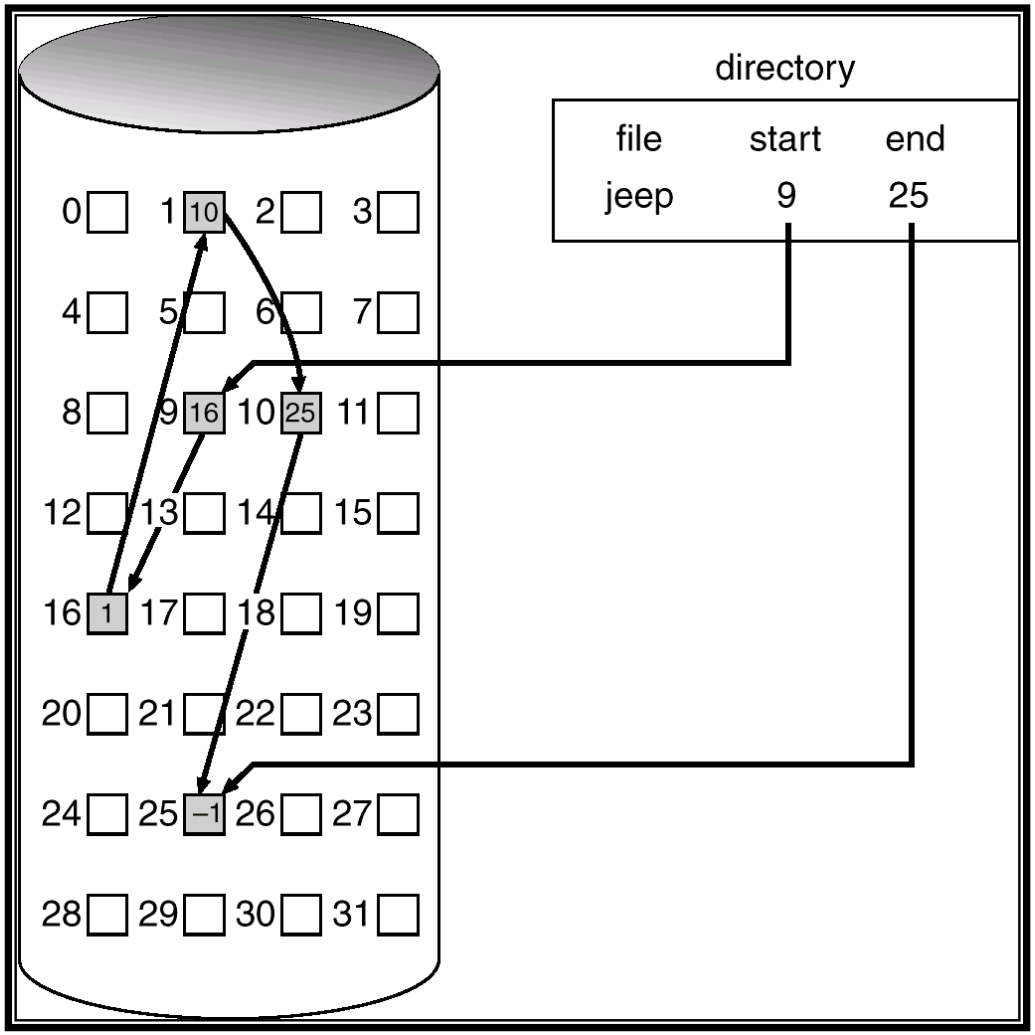
- Pointers in list are not accessible to user
- Directory table maps files into head of list for a file

Linked allocation example



Storing a file as a linked list of disk blocks

Linked Allocation



Linked List Allocation Issues

- Advantages
  - Easy to use – no estimation of size necessary ahead of time
  - Can grow dynamically in middle and at ends
  - Space efficient
    - no external fragmentation
    - internal fragmentation only on last block
    - only one pointer overhead per block
- Disadvantages
  - Slow – defies the principle of locality (physical blocks scattered). Need to read through linked list nodes sequentially to find record of interest blocks
  - Suited for sequential access but not direct access
  - User sees strange sized blocks – disk space must be used to store pointers (if disk block is 512 bytes, and disk address requires 4 bytes, then the user sees data blocks of 508 bytes)

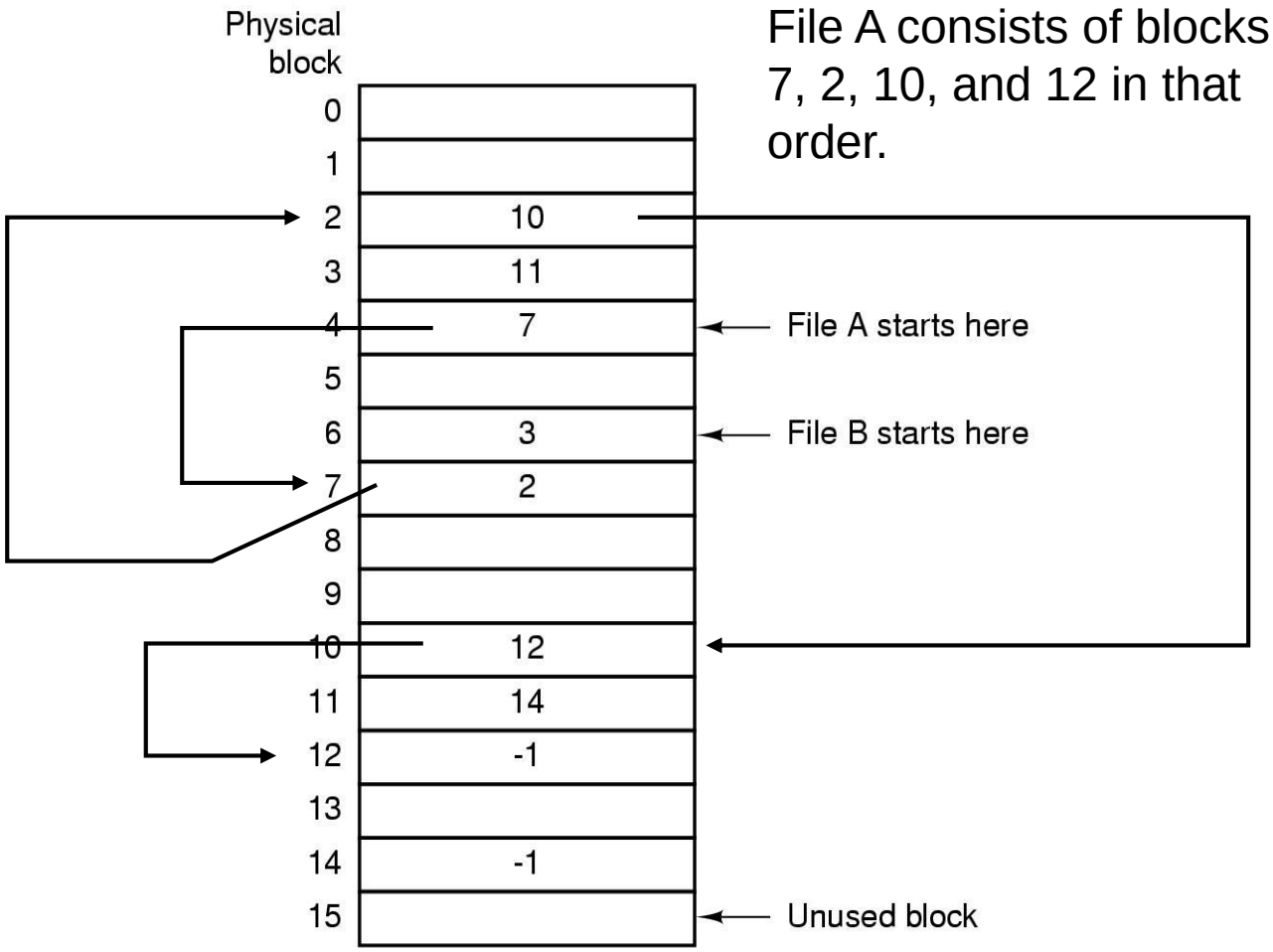
Linked List Allocation Issues

- Disadvantages (continued)
  - Not very reliable. System crashes can scramble files being updated
    - if system crashes during an update, files and information can be lost.
    - Since blocks can be scattered across disk and the only information keeping them together are the pointers to next block,
      ➔ if the pointers are messed up during crashes, recovery can be impossible.

File Allocation Table (FAT)

- Important variation on linked allocation method: `File Allocation Table' (FAT) - OS/2 and MS-DOS
- A section of the disk is allocated ahead of time to contain the table.
- Table contains one entry per disk block
- Directory entry contains the start of the block
- then each entry in the table contains the index of the next block

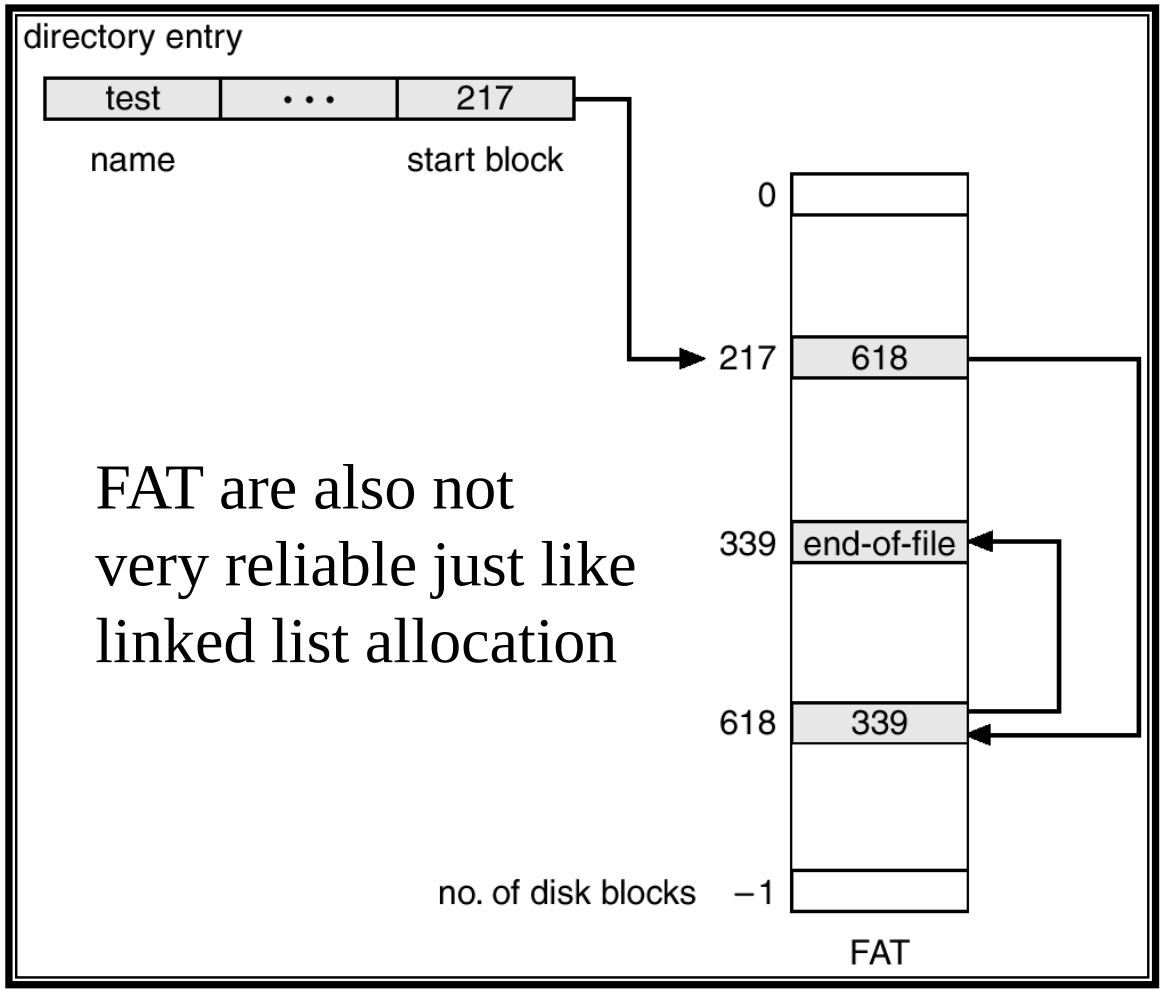
File Allocation Table



File A consists of blocks 4, 7, 2, 10, and 12 in that order.

Linked list allocation using a file allocation table

File-Allocation Table



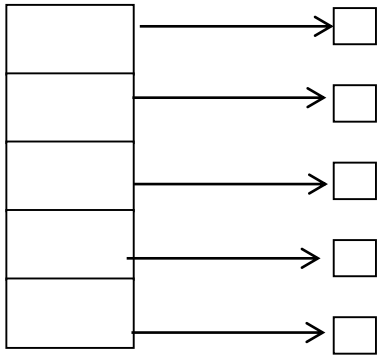
FAT are also not very reliable just like linked list allocation

Linked List Allocation Issues

- *Summary*: linked allocation solves the external fragmentation and size-declaration problems of contiguous allocation,
- However, it can't support efficient direct access

Indexed Allocation

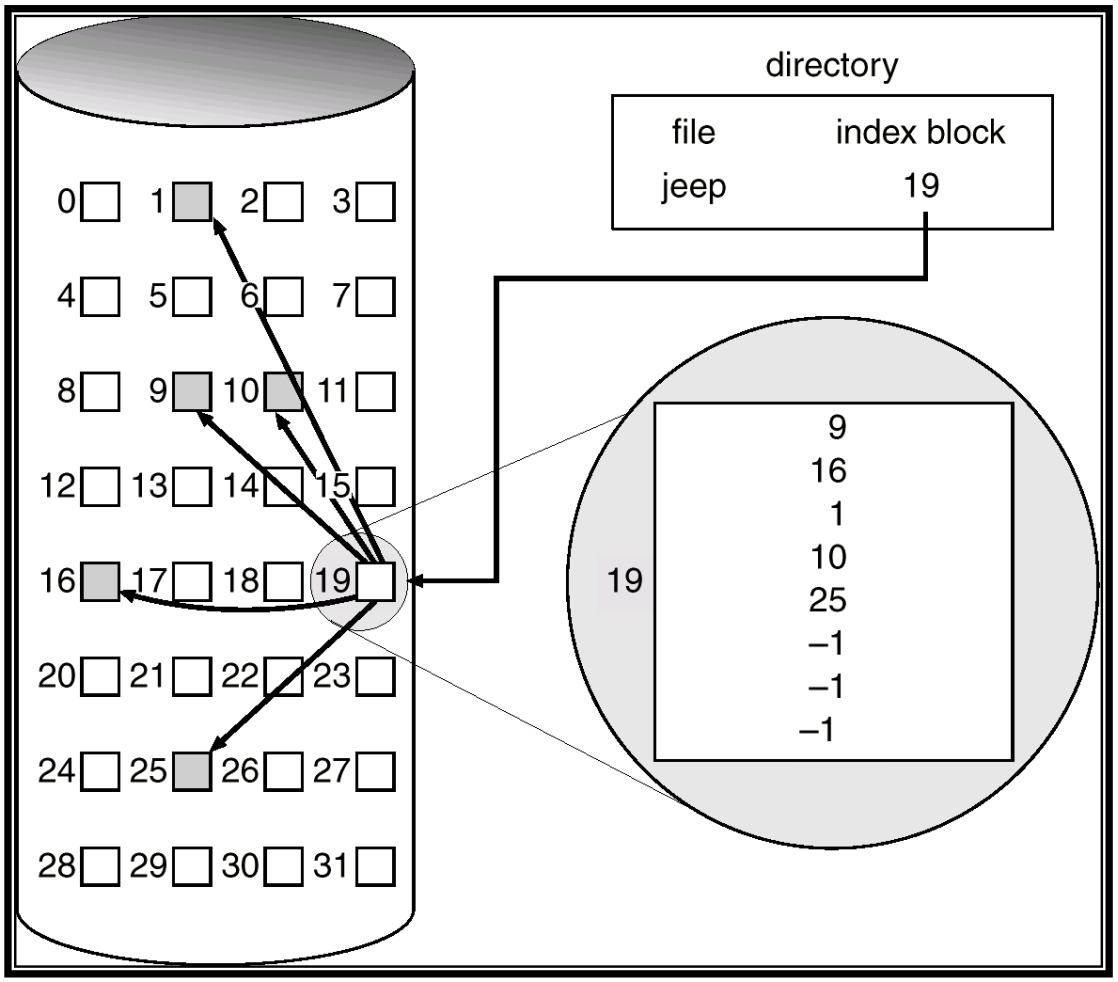
- Brings all pointers together into the *index block*.
- Logical view.



index table

Example of Indexed Allocation

•Accessing middle of the file is easier (no need to follow chain of pointers)
•if you want to keep redundant info, that's possible

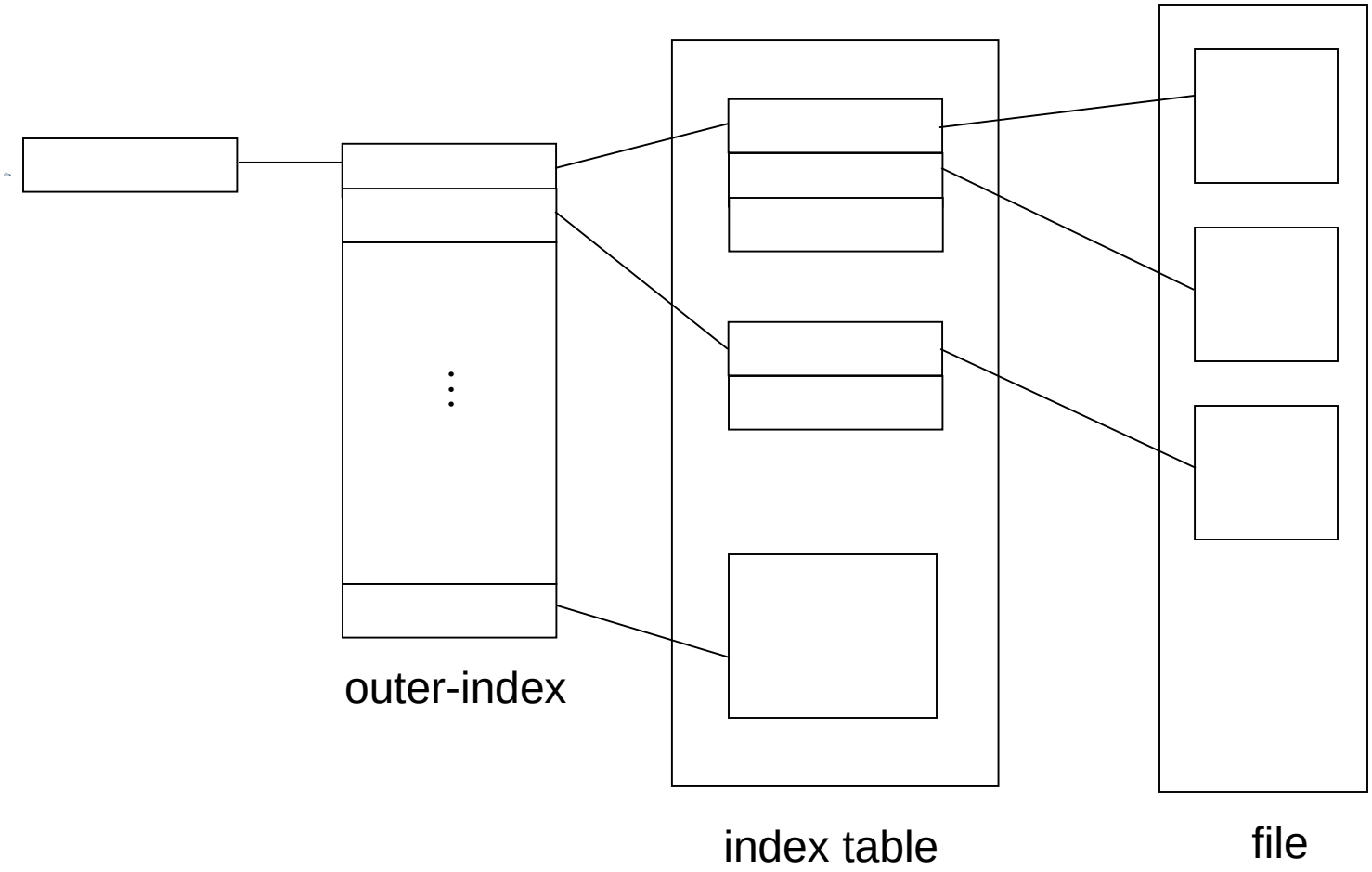


•Split everything into fixed sized blocks
•First thing directory points to is an index block
•key is used to search index block

Indexed Allocation

- Overhead – Requires extra space for index block, possible wasted space
- Random access easier
- Solves external fragmentation
- Dynamic allocation possible
  - File can be extended by rewriting a few new blocks and updating/rewriting the index block
- Supports sequential, direct and indexed access
  - Unix is implemented with indexed blocks even though files are sequential character oriented (sequential access is emulated).
- Access requires at most one access to index block first.
  - The index table can be cached in main memory

Two-level Indexed Allocation

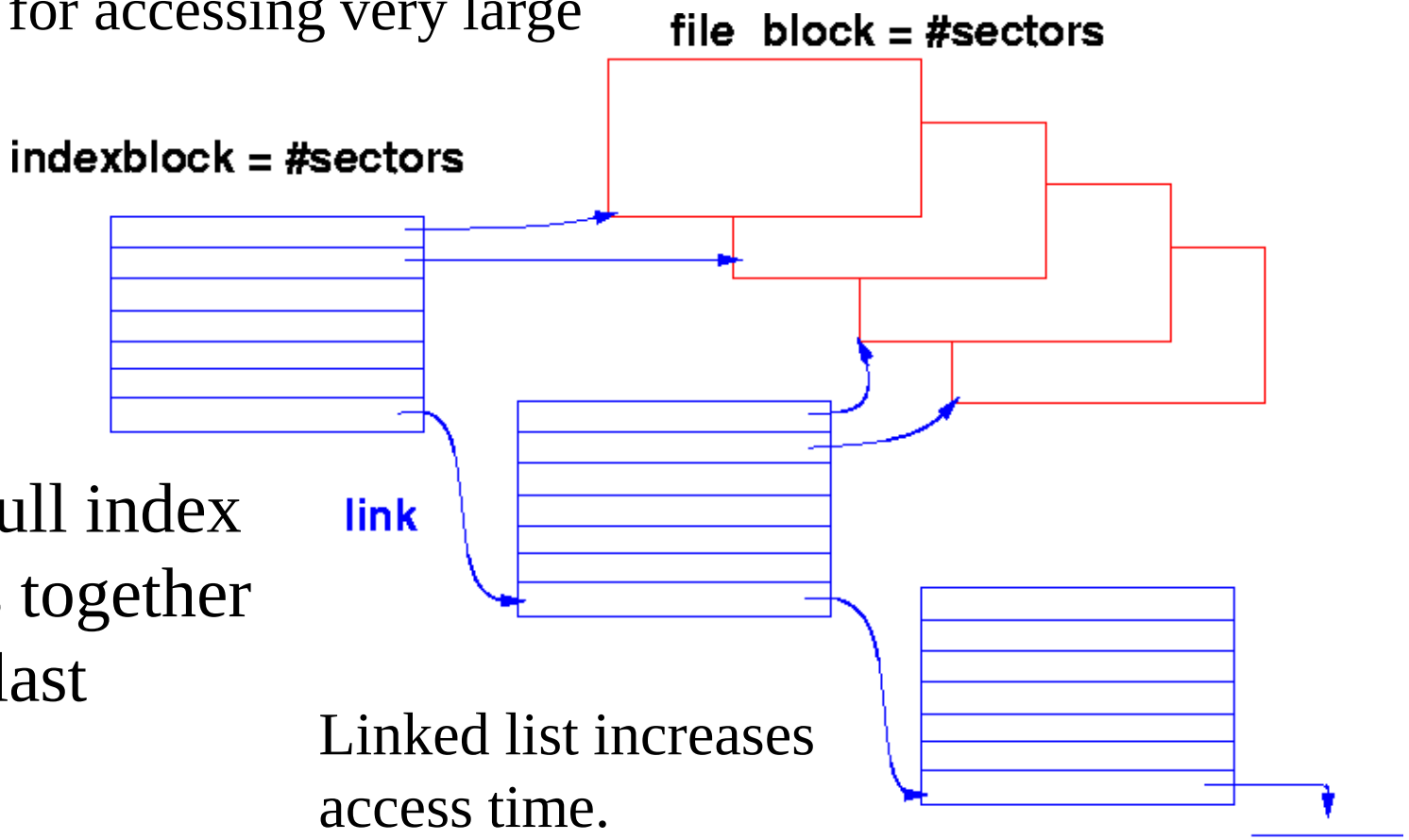


Uniform access time to get to any part of the disk

disadvantage: accessing directories take two lookups➔ too slow

Other Forms of Indexed File Linked

Useful for accessing very large files

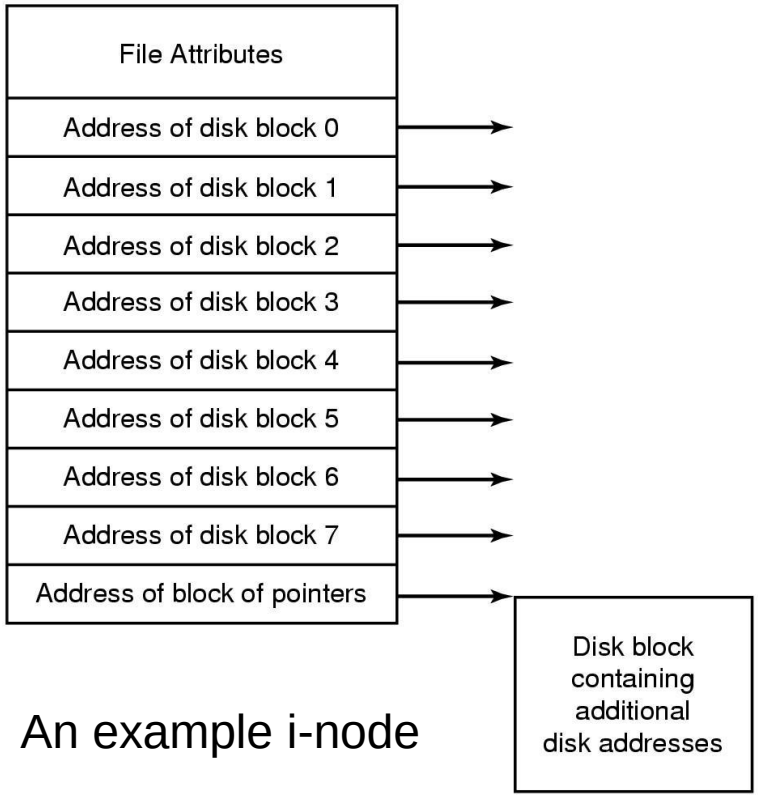


Link full index blocks together using last entry.

Linked list increases access time.

i-nodes

- i-node (index-node) contains file attributes, addresses of disk blocks making up the file and last entry a pointer to other blocks of disk addresses.
- Unix uses a variant of i-nodes



An example i-node

Performance Issues

- Allocation methods vary in their storage efficiency and data-block access times
- Difficulty in comparing the performance of various systems
- Contiguous allocation is fine for WORM types of applications: (write once read many)
- Linked allocation is fine for sequential access;
- Indexed allocation is more complex but more useful in real systems.

Free Space Management

- How does one allocate from and manage free space?
- A number of methods exist
  - Bit vector
  - Linked list of free blocks
  - Linked list of contiguous blocks
  - Others…

Free Space Management

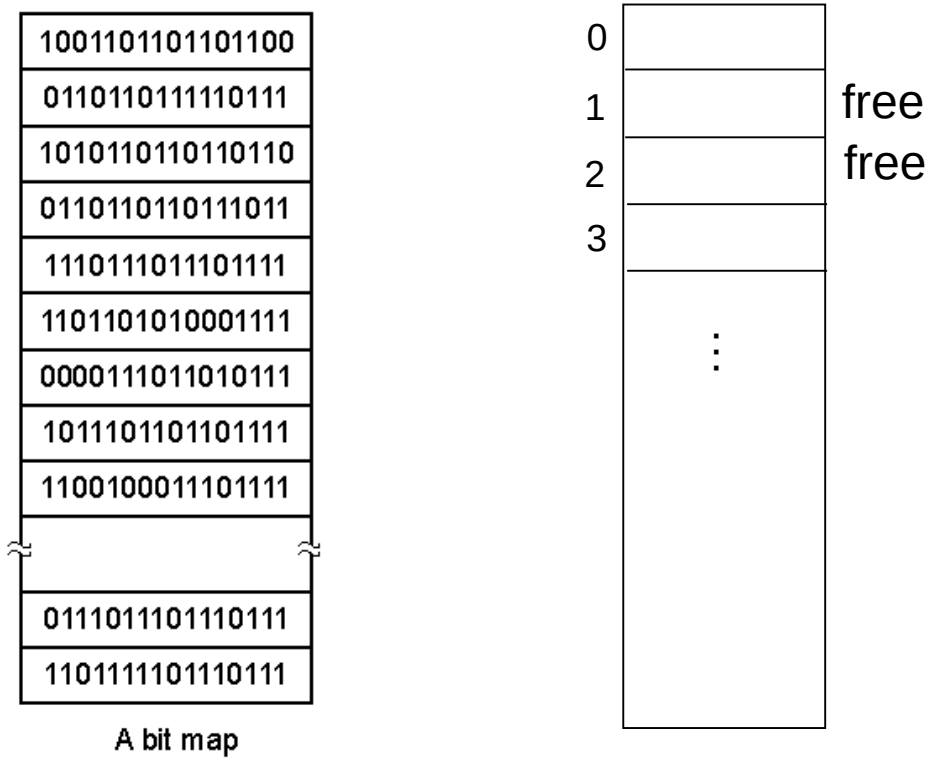
- Bit vector
  - A bit map is kept of free blocks (similar to paging schemes)
  - Each bit in a vector represents one block
  - If the block is free, the bit is zero
  - Simple to find n consecutive free blocks (look for n 0's: 0000…)
  - Overhead is bit map (bit map requires extra space). Example:

    block size = 2^{12} bytes

    disk size = 2^{30} bytes (1 gigabyte)

    $n = 2^{30}/2^{12} = 2^{18}$ bits (or 32K bytes)
  - Example BSD file system (bitmap gets stored on disk)

Free Space Management



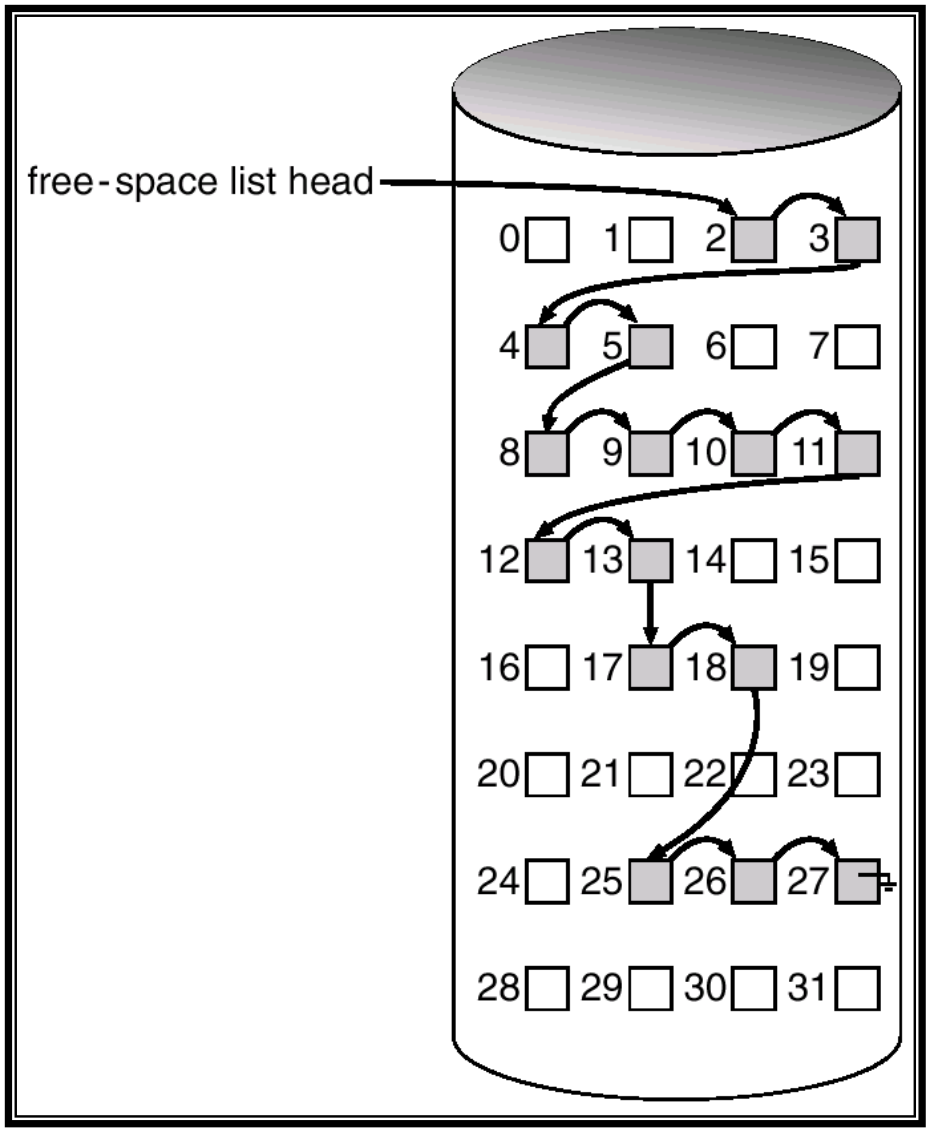
1 bit per block. Bit = 0 means block is free.

Bit = 1 means block is used.

Free Space Management

- Linked list (free list)
  - Keep a linked list of free blocks
  - Not very efficient because linked list needs traversal
  - when space is freed up, inserting into free space list is easy
  - Example System V, R1
  - Disadvantage: Cannot get contiguous space easily
  - Advantage: No waste of space

Linked Free Space List on Disk



Free Space Management

- Linked list of contiguous blocks that are free
  - The free list node consists of a pointer and the number of free blocks starting from that address
  - Blocks are joined together into larger blocks as necessary

Example Real file systems

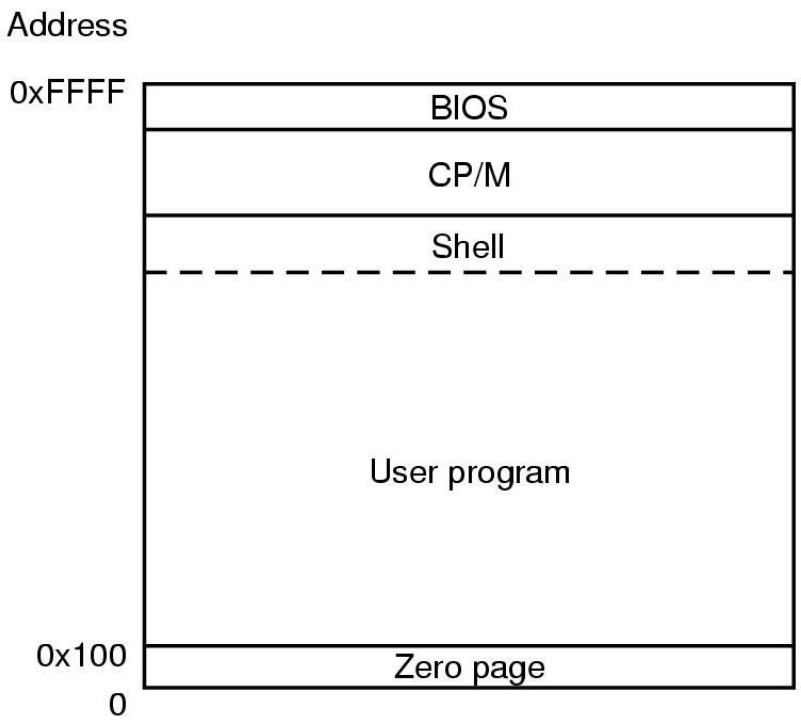
- CP/M file system (really old; precursor to MS-DOS)
- MS-DOS
- NTFS
- Unix V7 (PDP-11)

The CP/M File System

- CP/M was a simple operating system on early microcomputers in early 1980s (before the time of PC's)
- CP/M = Control Program for Microcomputers
- Some stats:
  - 4KB RAM (later versions went up to 64KB)
  - 8-inch floppy of 180KB as mass storage (later versions went up to 720KB)
  - CP/M v 2.2: the OS itself occupied 3584 bytes (< 4KB) of memory
  - The shell occupied another 2KB.
- CP/M is interesting to study because it can be relevant to small, embedded systems.

The CP/M File System

- Memory layout of CP/M



BIOS implements 17 I/O calls to handle keyboard, screen, etc

CP/M Operating system (< 4KB)

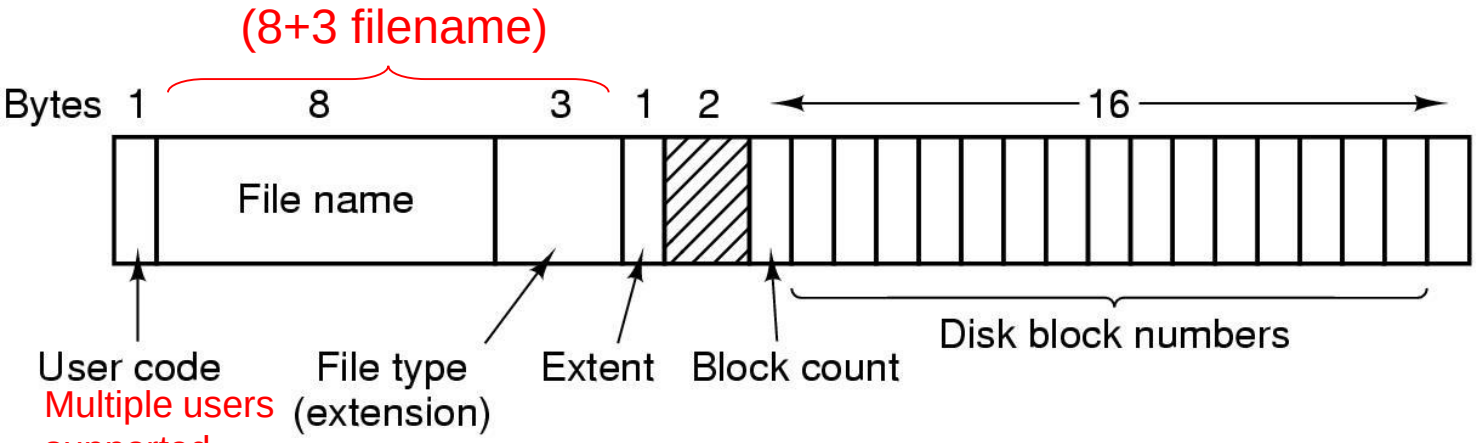
Shell (2KB)

User programs (the rest of memory)

Reserved for interrupt vectors and I/O buffer (256 bytes)

The CP/M File System

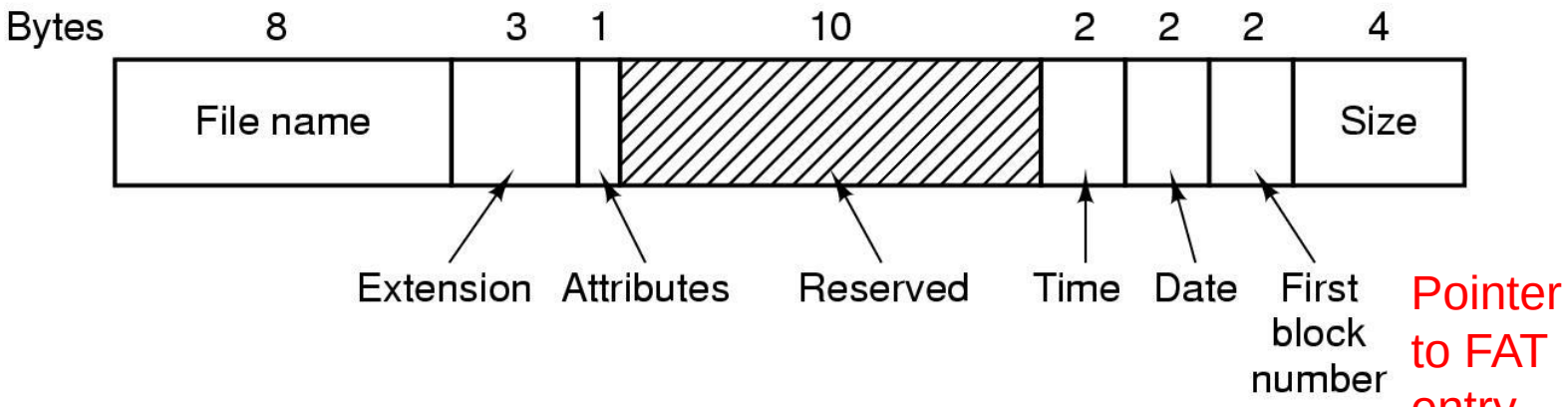
- There is only one directory for the entire file system
- After an open file call, the directory entry for the file is located and all the blocks in the file are known.
- If the file size is larger than 16KB (last 16 entries of 1KB blocks each in the directory entry), then multiple directory entries are used to have more disk block information. Extent field keeps track of the order of the blocks.



The CP/M directory entry format (total 32 bytes)

The MS-DOS File System

- Very similar to CP/M, except
  - It implements a hierarchical file system (after v 2.0)
  - Keeps time information
  - Loses the user information – whoever is using the machine has access to all the files.
  - Block numbers are kept in the FAT



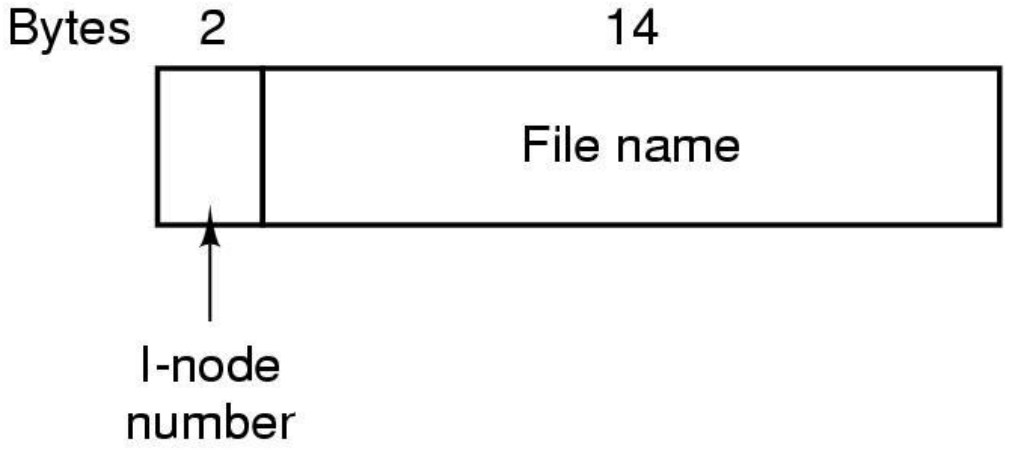
The MS-DOS directory entry (total 32 bytes)

The MS-DOS File System

Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

- Three versions of FAT: FAT-12, FAT-16, and FAT-32 (numbers are the number of bits used for a disk address)
- Disk block sizes are multiples of 512 bytes (0.5KB)
  - These combinations define maximum partition sizes for different block sizes for each FAT type.
  - Example: FAT-12 with 0.5K block size = $2^{12} \times 2^{9} = 2^{21}$ = 2MB
  - The empty boxes represent forbidden combinations
- But, some OS's have smaller max partition sizes by design.

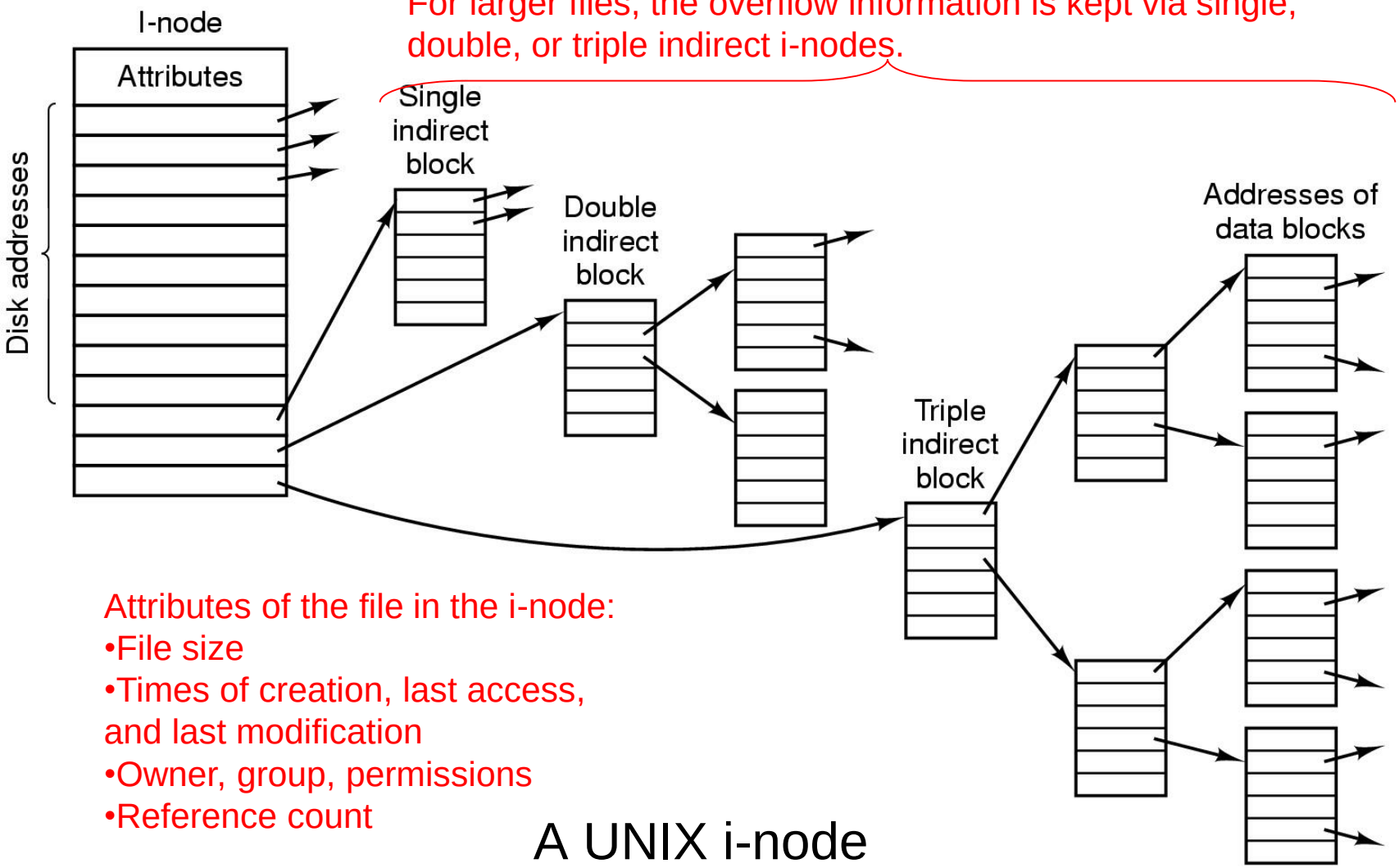
The UNIX V7 File System (used in PDP-11)



- A UNIX V7 directory entry
- Directory entry does not contain block information. This is in the i-node entry in Unix.
- i-node number (i-number) is 2 bytes.
  - Number of files per file system = 2^{16} = 64K

The UNIX V7 File System

For larger files, the overflow information is kept via single, double, or triple indirect i-nodes.



Attributes of the file in the i-node:

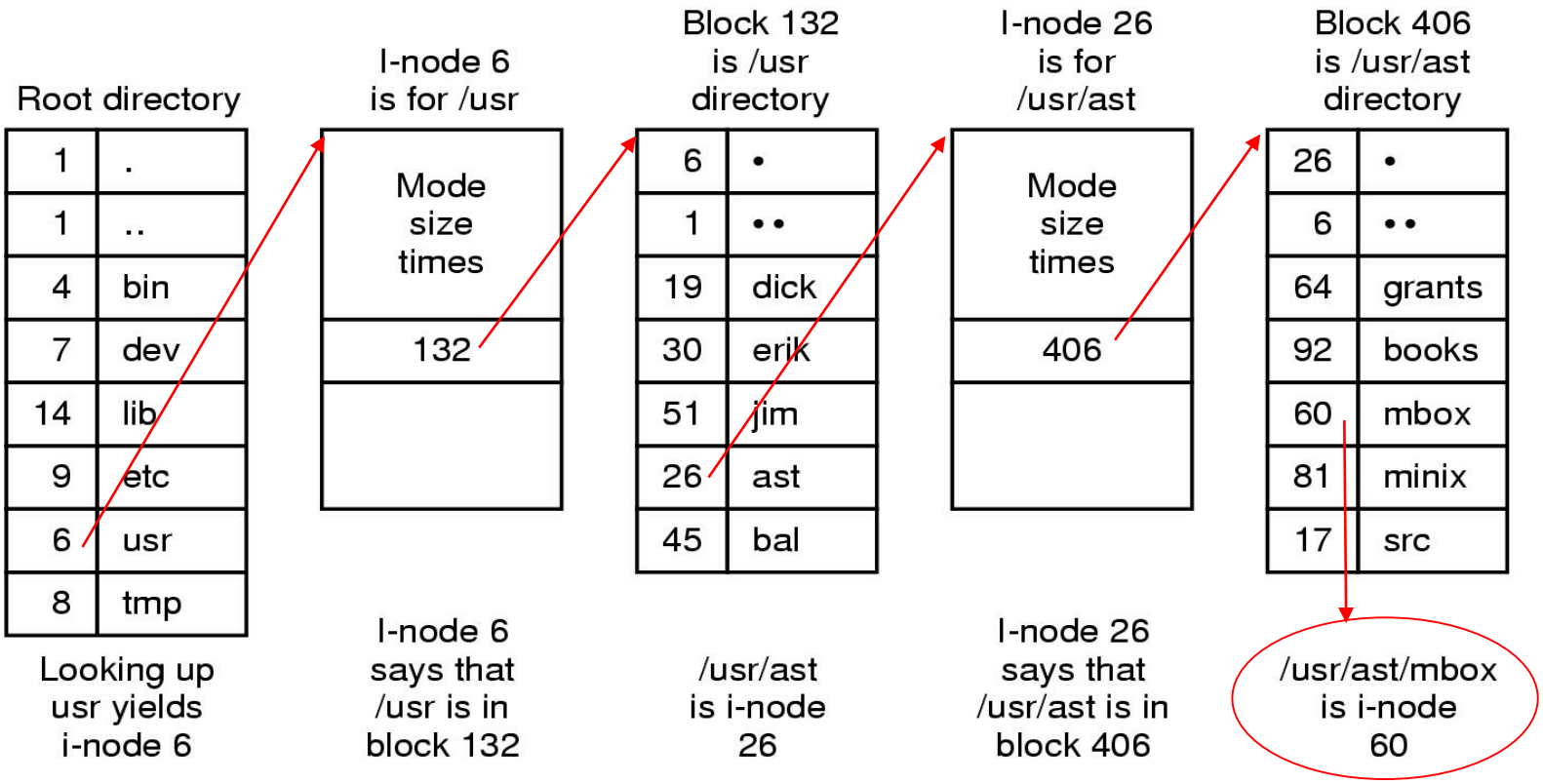
- File size
- Times of creation, last access, and last modification
- Owner, group, permissions
- Reference count

A UNIX i-node

Largest file

- Suppose i-node has 10 block address entries and a block size of 1024 bytes and a block address specified by 4 bytes. What is the largest file that can be represented with such a structure?
- Answer:
  - i-node itself contains 10 block addresses.
  - Single indirect block will contain 256 (1024 byte/4 bytes) addresses.
  - Double indirect blocks will represent 256^2 addresses
  - Triple indirect blocks will represent 256^3 addresses.
  - Total number of data blocks that can be represented $=10+256+256^2+256^3 =16,843,018$.
  - Each block is 1KB, so the largest file size that can be represented ≈ 16MB x 1KB = 16 GB

The UNIX V7 File System



The steps in looking up */usr/ast/mbox*

- Start with root: its i-node at a fixed location. The follow the chain.

Log-Structured File Systems (LFS)

Motivation

- CPUs becoming faster, but disk seek times are not getting smaller.
- With CPUs getting faster, and memory getting larger and cheaper
  - disk caches are also increasing
  - Therefore, increasing number of read requests can come from cache instead of read operations directly from disk
  - thus, in the future, most disk accesses will be writes and the pay-off from read-ahead would be diminished.
  - In most file systems, writes are done in small chunks which are very costly
    - example: creating a file on Unix involves the following: (a) write i-node for the directory entry, (b) write the directory block, (c) write the i-node for the file, and (d) write the file itself.
    - In particular, the i-node writes need to be done immediately to increase reliability. No chance to group the writes for these blocks.

Recall from Ch 10

To utilize bandwidth fully, we need to transfer larger chunks of data.

Block Size (Kbytes)	% of Disk Transfer Bandwidth
1Kbytes	0.5%
8Kbytes	3.7%
256Kbytes	55%
1Mbytes	83%
2Mbytes	90%

Log-Structured File Systems (LFS)

Solution

- Log-structured or journaling file systems developed at Berkeley.
- Goal: try to achieve the full bandwidth of the disk.
- Implementation: LFS Strategy structures entire disk as a log
  - have all writes initially buffered in memory
  - periodically write these to the end of the disk log (called segments)
  - when file opened, locate i-node (involves search), then find blocks

Log-Structured File Systems (LFS)

Details

- The writes are collected into a single big segment to be written to the disk. If segments are selected to be ≈ 1MB, the full bandwidth of the disk would be utilized. (recall chapter 10 material)
- They are written onto disk as a single segment.
- Segments contain many i-nodes, directory blocks, and data blocks all mixed together.
  - The i-nodes are no longer at fixed locations on the disk.
  - So, when a file is opened, the i-node for that file needs to be located. Implies search.
  - An i-node map is maintained, indexed by the i-number.
    - The i-map is kept on disk, but is also cached, so heavily used parts will be in memory most of the time.
  - When an i-node is located, locating the blocks is done in the usual way.
- At the head of each segment, there is a segment summary that tells what is in that segment.

Log-Structured File Systems (LFS)

Details

- If disks were infinite, we'd keep appending new segments to the end of the disk and we would be done.
- But disks are finite and after a while they will fill up. But previous segments (for files that have been modified), would contain outdated information that can be freed.
- Implemented by a background thread called the cleaner.
  - Disk is like a big circular buffer. Cleaner is like a garbage collector.
  - Cleaner scans the log (one segment at a time) and tries to identify i-nodes that are not up to date (by comparing them to current i-node map). If they are not up to date, it claims the space.
  - The i-nodes and blocks that are up to date for that segment are put in memory scheduled to be written into the new log. The examined segment is then reclaimed.

Observations

- Most files that get accessed are either very small
  - e.g., in Unix directories get accessed a lot which are small files or shell scripts are accessed a lot which are small files.
- Or very large
  - binary files, video files, etc.
- So file system should be optimized to access very small files or very large files fast

Observations

- Files get read more often than get written
  - so optimize the files system for reads more than writes
- Many files created do not last very long
  - e.g., on Unix, average lifetime of a file is 2 seconds.
  - This is because a lot of temporary files are created and then discarded by utility programs (editors, etc)
  - So optimize the file system so that short-lived files are accessed fast.

Efficiency and Performance

- Efficiency dependent on:
  - disk allocation and directory algorithms
  - types of data kept in file's directory entry
- Performance
  - disk cache – separate section of main memory for frequently used blocks
  - free-behind and read-ahead – techniques to optimize sequential access (later)
  - improve PC performance by dedicating section of memory as virtual disk, or RAM disk.

Efficiency and Performance

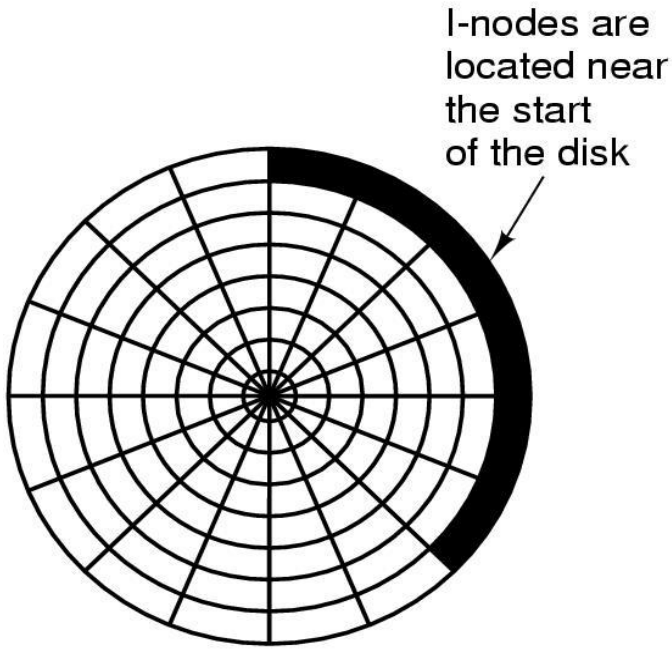
- **Free-behind** - removes a block from the buffer as soon as the next block is requested
  - The previous blocks are not likely to be used again and waste buffer space
- **Read-ahead** - a requested block and several subsequent blocks are read and cached

Efficiency and Performance

- Performance is improved by having disk cache in the main memory
  - Can have dirty bits just as in VM. So a dirtied file in cache needs to be written out.
  - Typically blocks are modified in cache, then a number of blocks are put together and written to disk in one large chunk.
  - Consistency problems arise and need to be solved
  - The existence of caches is why we cannot simply turn the power off on most computers.
    - The cache modifications need to be written to disk before power can be turned off.
  - File systems can be made more reliable (by writing everything, etc), but that would slow down IO operations too much.

Efficiency & Performance

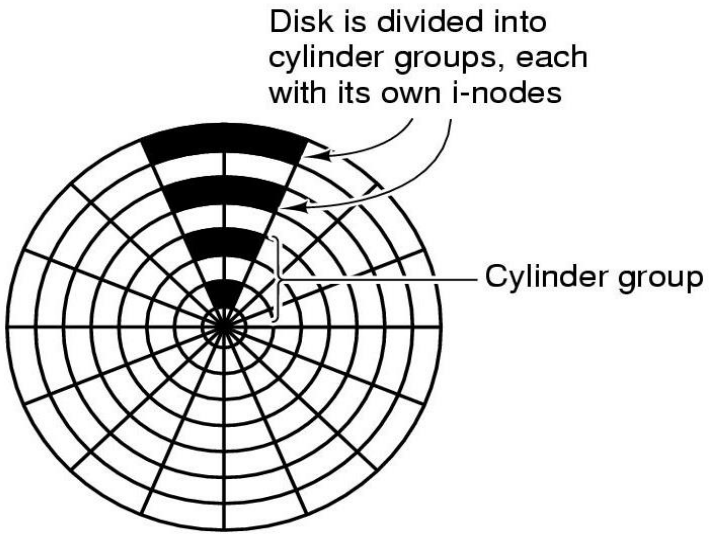
- In systems that implement i-nodes, reading a file requires two disk accesses: one for the i-node and a second one to access the blocks.
- Typically, i-nodes are placed at the start of the disk
- This makes average seek time to be ½ the number of cylinders



(a)

Efficiency & Performance

- The seek time can be improved by placing the i-node close to where the data blocks are.
- Disk divided into cylinder groups
  - each with its own blocks and i-nodes
- Ideally, try to put the data blocks in the same track as the i-node
- If free blocks on the same track cannot be found, put the data blocks on nearby tracks.



(b)

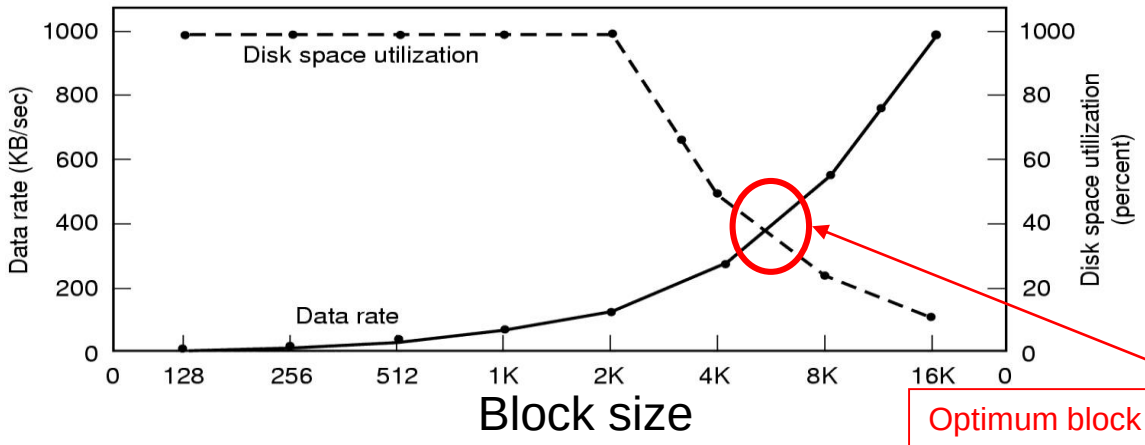
Efficiency and Performance

- Performance is improved by having a section of memory set aside and treated as a virtual disk or RAM disk
  - average lifetime of a file on Unix is about 2 seconds
  - most files are temporary; created by programs and discarded
  - Unix has elaborate caching schemes & usually these temporary files are never written to disk

Block size

- To improve I/O efficiency, I/O transfers between memory and disk are performed in blocks
- What should the block size be in a file system?
- Block size: from 32 bytes to 4096 bytes
  - lower limit depends on cache line size
  - upper limit depends on sector size. One wants to transfer 1 or more sectors at a time to minimize access time.
- It also depends on the average file size on a system.
  - Empirical data in one particular Unix system with 1000 users and 1 million Unix disk files:
    - Median file size = 1680 bytes
    - i.e., half the files are below and half are above this size.

Block size



Data rate and space utilization are inherently in conflict.

Optimum block size somewhere around 4K for this example (power of 2).
But unix has 1K block sizes for historical reasons.

- All files 2KB (approximation to 1680)
- Dark line (left hand scale) gives data rate of a disk for various block sizes
  - Data access time dominated by seek time: the larger the block, the better the data rate becomes once the block is located on disk.
- Dotted line (right hand scale) gives disk space efficiency for various block sizes.
  - The larger the block size, the bigger portion of the block will be wasted.

Summary

- File system interface issues
  - Logical view: attributes
  - File operations
  - Directory operations
  - File access types (sequential, indexed, direct, etc)
- File system implementation issues
  - Allocation schemes
  - Free space management methods
  - Efficiency and performance issues
- Some example real file systems
  - CP/M
  - Dos
  - unix