Operating Systems

0107451

Chapter 4 File Systems

Dr. Naeem Odat

Tafial Technical University
Department of Computer and Communications Engineering

Introduction

Introduction

- Three essential requirements for long-term information storage:
    1. It must be possible to store a very large amount of information. Address space of a process is small and limited.
    2. The information must survive the termination of the process using it. While info stored in RAM within address space of a process, vanishes when a process is terminated.
    3. Multiple processes must be able to access the information at once.

Introduction

Introduction

- Magnetic disks, solid-state drives, tapes and optical disks have been used for long-term storage.
- A disk can be thought of as a linear sequence of fixed-size blocks and supporting two operations:
  - Read block k
  - Write block k.
- However, other things should be thought of: How do you find information?, how do you keep one user from reading another user's data? and how do you know which blocks are free?

Introduction

File abstraction

Just as we saw how the operating system abstracted away the concept of the processor to create the abstraction of a process and how it abstracted away the concept of physical memory to offer processes (virtual) address spaces, we can solve this problem with a new abstraction: the file.
Together, the abstractions of processes (and threads), address spaces, and files are the most important concepts relating to operating systems. If you really understand these three concepts from beginning to end, you are well on your way to becoming an operating systems expert.

The part of the operating system dealing with files is known as the **file system**.

4.1 Files

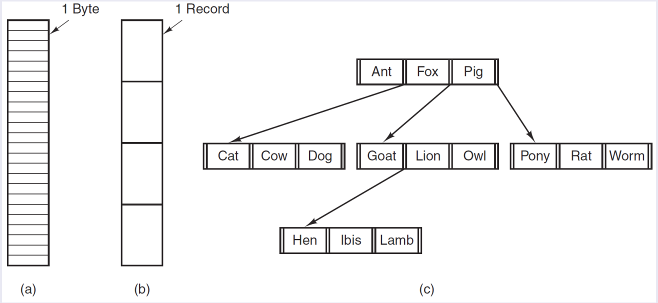
User's point of view for files.

File Naming

- All current operating systems allow strings of one to eight letters as legal file names, e.g., andrea, bruce, and cathy.
- Frequently digits and special characters are also permitted, e.g., 2, urgent!, and Fig.2-14.
- Some file systems distinguish between upper- and lowercase letters, e.g., maria, Maria, and MARIA.
- Examples of file systems:
  - FAT-16 (MS-DOS, Windows 95, Windows 98)
  - FAT-32 (Windows 98, ...)
  - NTFS (Windows NT, ...)
  - ReFS or Resilient File System (Windows 8 server version)
  - exFAT (Microsoft extension of FAT to flash drives and large file systems)
- Some file names have extension that represent a file type, e.g., myfile.c (c file).

4.1 Files

File Structure



UNIX and Windows use (a), unstructured approach. Then it is up to user to put what ever he wants in his file.
Mainframe operating systems base their file systems on (b).
In (c) a file consists of a tree of records, not necessarily all the same length, each containing a key field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key.

4.1 Files

File Types

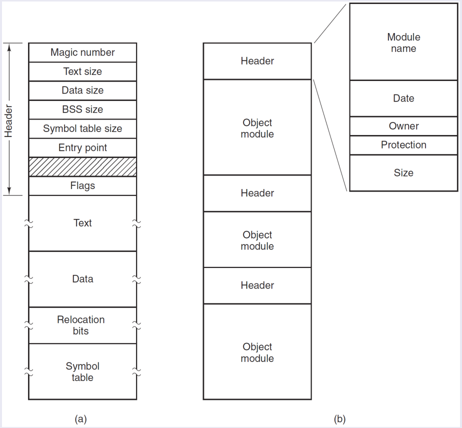
UNIX (including OS X) and Windows, for example, have regular files and directories. UNIX also has character and block special files.
Regular files are the ones that contain user information.
Directories are system files for maintaining the structure of the file system.
Regular files are generally either:

1. ASCII files: Can be edited and each line is terminated by CR/LF.
2. or binary files: not ASCII and some can be executed by the system.

4.1 Files

File Types - binary files in UNIX

- (a) Executable has several sections.
- (b) Archive has certain format for the libraries stored in.



4.1 Files

File Access

There are two kinds of file accesses:

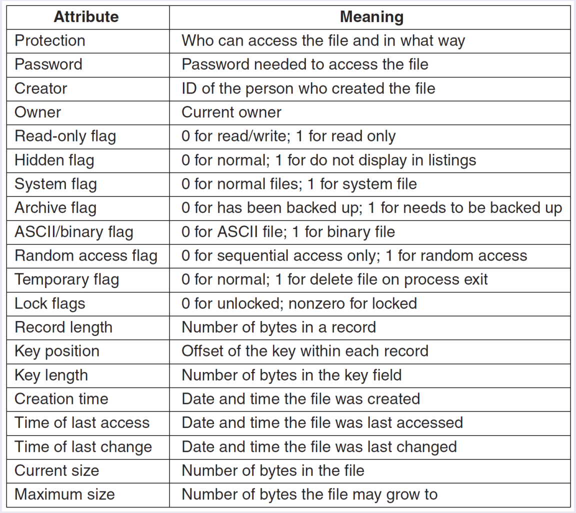
1. Sequential access, done on tapes by early operating systems.
2. Random access, where reading starts from any point within a file.

Two methods can be used for specifying where to start reading:

1. Every *read* operation gives the position in the file to start reading at.
2. A special operation, *seek*, is provided to set the current position, where reading starts, as in UNIX and Windows.

4.1 Files

File Attributes - name, data, and other info (metadata or 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 was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

4.1 Files

File Operations

1. Create.
2. Delete.
3. Open.
4. Close.
5. Read.
6. Write.
7. Append.
8. Seek.
9. Get attributes.
10. Set attributes.
11. Rename.

4.1 Files

An Example Program Using File-System Calls

```
#include <sys/types.h> /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#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 */
   /* 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 */
   while (TRUE) {  /* Copy loop */
       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(in_fd);
   close(out_fd);
   if (rd_count == 0) exit(0); /* no error on last read */
   else exit(5); /* error on last read */
}
```

4.1 Files

System Calls - *lseek*

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence)
```

- sets the file pointer for fd:
  - if whence is **SEEK_SET**, the pointer is set to offset bytes.
  - if whence is **SEEK_CUR**, the pointer is set to its current value plus offset bytes
  - if whence is **SEEK_END**, the pointer is set to the size of the file plus offset bytes
- it returns the (possibly) updated value of the file pointer relative to the beginning of the file.
- Thus, $n =$ *lseek(fd, (off_t)0, SEEK_CUR);*
  returns the current value of the file pointer for **fd**

4.1 Files

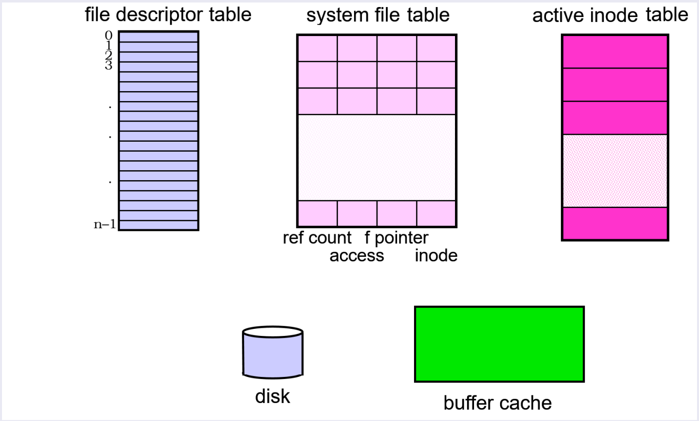
Standard File Descriptors

```
main(){
    char buf[100];
    int n;
    const char* note = "Write failed\n";
    while ((n = read(0, buf, sizeof(buf))) > 0)
        if (write(1, buf, n) != n) {
            (void)write(2, note, strlen(note));
            exit(EXIT_FAILURE);
        }
    return(EXIT_SUCCESS);
}
```

- 0: standard input
- 1: standard output
- 2: standard error output

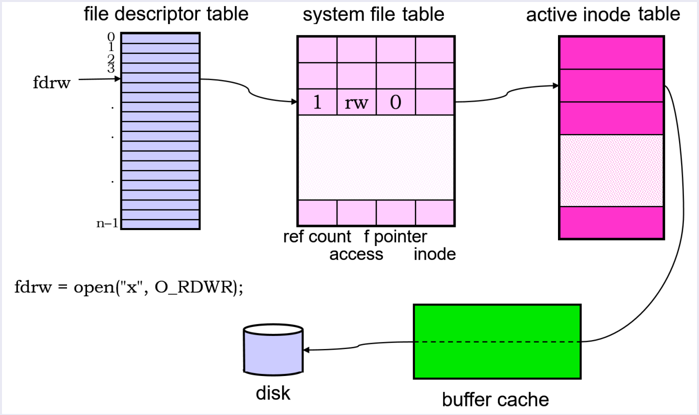
4.1 Files

Representing Open File



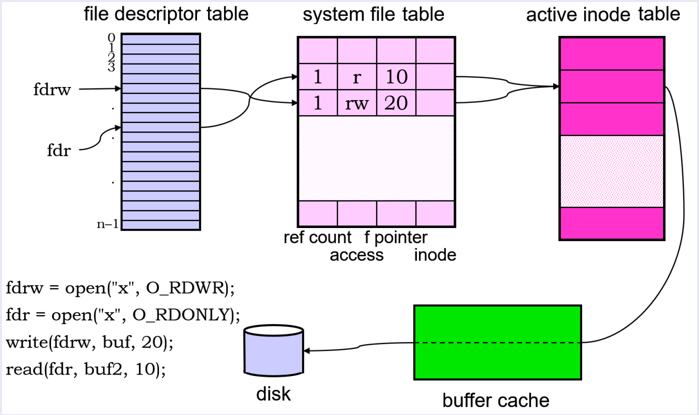
4.1 Files

Representing Open File



4.1 Files

Representing Open File



```
fdrw = open("x", O_RDWR);
fdr = open("x", O_RDONLY);
write(fdrw, buf, 20);
read(fdr, buf2, 10);
```

4.1 Files

Multiple Descriptors and One File

- It is guarantee that the file descriptor is the smallest free number in the file descriptors table.
- To have a file with multiple descriptors:
  - Using *dup* system call.
    $int\ dup(int\ fd);$
  - Using *dup*2 system call.
    $int\ dup2(int\ oldfd,\ int\ newfd);$

4.1 Files

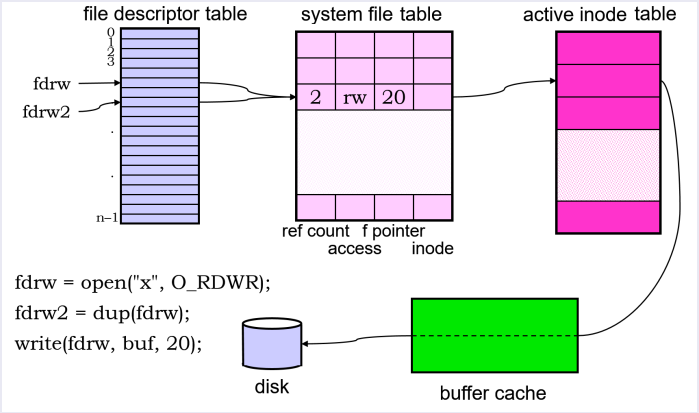
dup Example

```
/* redirect stdout and stderr to same file */
/* assumes file descriptor 0 is in use */
close(1);
open("file", O_WRONLY|O_CREAT, 0666);
close(2);
dup(1);

/* alternatively, replace last two lines with: */
dup2(1, 2);
```

4.1 Files

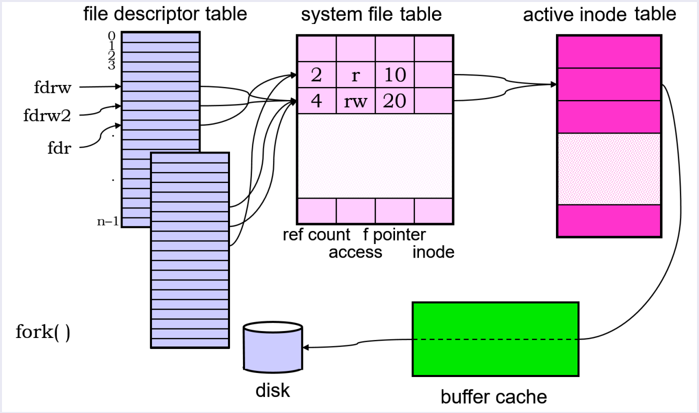
Representing Open File



fdrw = open("x", O_RDWR);
fdrw2 = dup(fdrw);
write(fdrw, buf, 20);

4.1 Files

Representing Open File



4.1 Files

I/O Redirection

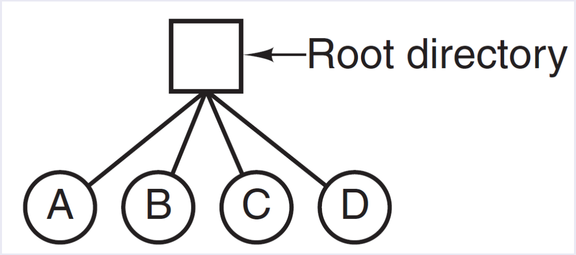
```
$ who > file &

if (fork( ) == 0) {
    char *args[ ] = {"who", 0};
    close(1);
    open("file", O_WRONLY|O_TRUNC, 0666);
    execv("who", args);
    printf(\cannot print it\n");
    exit(1);
}
```

4.2 Directories

Single-Level Directory Systems

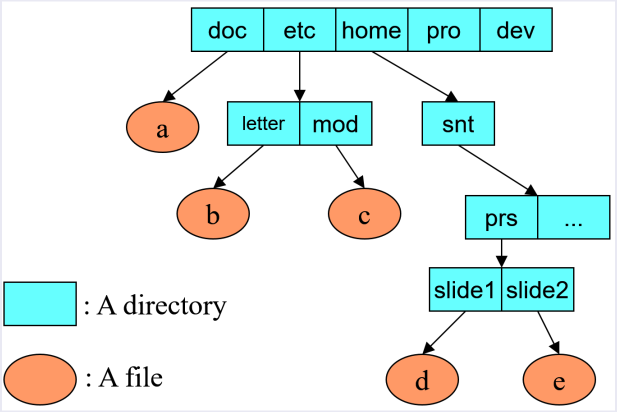
- To keep track of files operating systems usually have directories or folders.
- The simplest form of directory system is having one directory (root directory) containing all the files.



4.2 Directories

Hierarchical Directory Systems

- It is hard to find files in single level structure.
- Therefore a hierarchy is needed for that purpose and for organizing data in a better way.

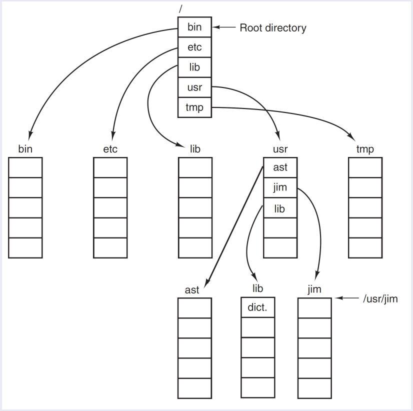


4.2 Directories

Path Names

Two method are there to specify file names:

1. Absolute path name, e.g., */usr/local/share/a.txt*
2. Relative path name, e.g., ./d.c, ../share/g.txt



4.2 Directories

Directory Operations

1. Create.
2. Delete.
3. Opendir.
4. Closedir.
5. Readdir.
6. Rename.
7. Link.
8. Unlink.

4.3 File-System Implementation

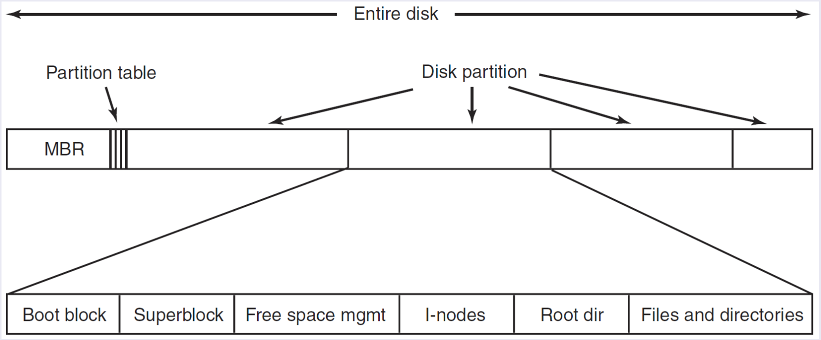
File-System Layout

- Most disks can be divided up into one or more partitions, with independent file systems on each partition.
- Sector 0 of the disk is called the **MBR** (Master Boot Record) and is used to boot the computer.
- MBR has a partition table which contains the start and end addresses of each partition. One of the partition is active.
- MBR reads and execute the first block (boot block) of the active partition.
- Boot block loads and starts the OS in that partition.

4.3 File-System Implementation

File-System Layout

Often the file system will contain some of the items shown:



Superblock: read into memory upon booting. Contains, typically, a magic number to identify the file-system type, the number of blocks in the file system, and other key administrative information.
i-nodes: an array of data structures, one per file, telling all about the file.

4.3 File-System Implementation

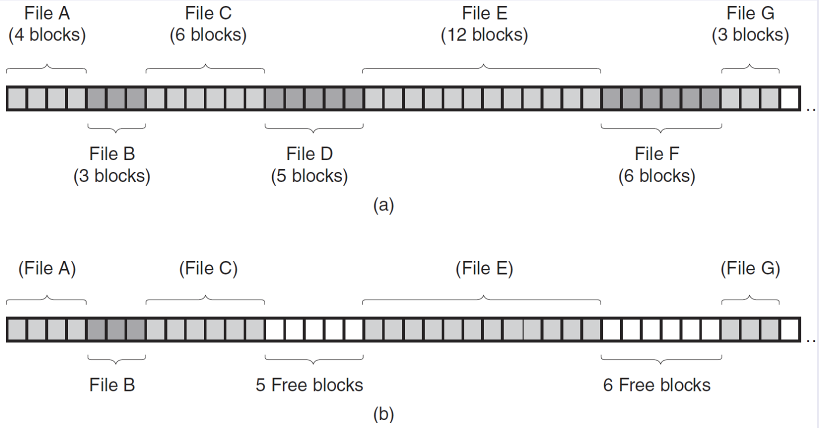
Implementing Files

- Contiguous allocation
- Linked-list allocation
- Linked-list allocation using table in memory
- I-node (index node)

4.3 File-System Implementation

Implementing Files

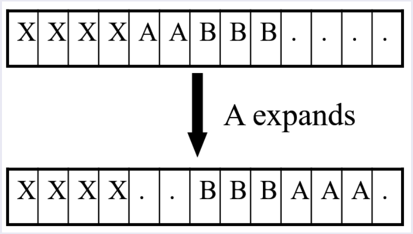
Contiguous Allocation:



Used in CD-ROMs, DVDs, Blu-rays, ...

4.3 File-System Implementation

Implementing Files

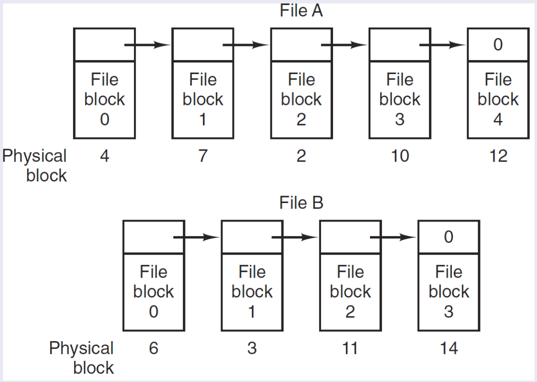


- All disk blocks of a file allocated sequentially
- Advantages:
  - (very) Fast read
  - Useful for read-only file systems (CD-ROM)
  - Keeping track of blocks of a file is easy
- Problems:
  - Fragmentation with deletes
  - File growth might be expensive

4.3 File-System Implementation

Implementing Files

- **Linked-List Allocation:**

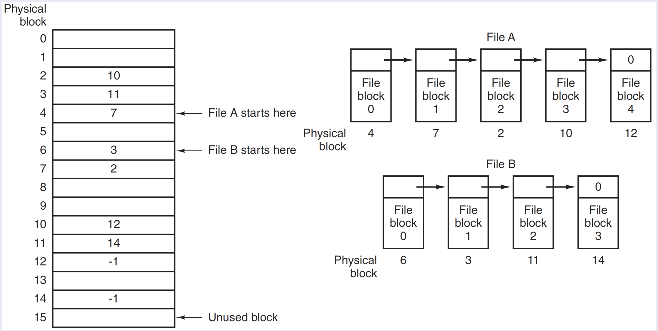


- Drawbacks:
  - Random access is painful.
  - Reading a block of a file requires 2 disk blocks.

4.3 File-System Implementation

Implementing Files

- **Linked-List Allocation Using a Table in Memory:** Such a table in main memory is called a FAT (File Allocation Table).

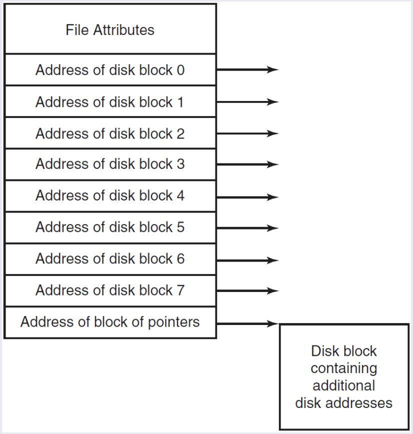


- **Disadvantage:** For large disks it takes huge space of the main memory.

4.3 File-System Implementation

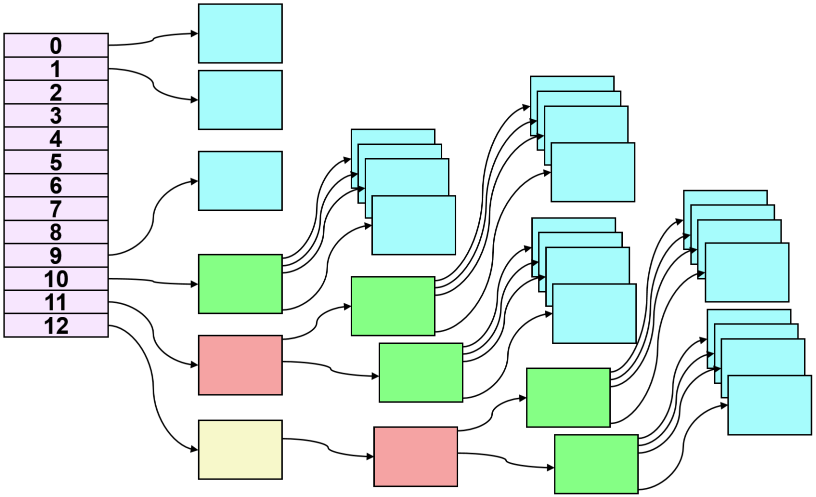
Implementing Files - I-nodes (index-nodes)

- I-node lists the attributes and disk addresses of the file's blocks.
- I-node is in memory only when the corresponding file is open.



4.3 File-System Implementation

Implementing Files - I-nodes in Linux



4.3 File-System Implementation

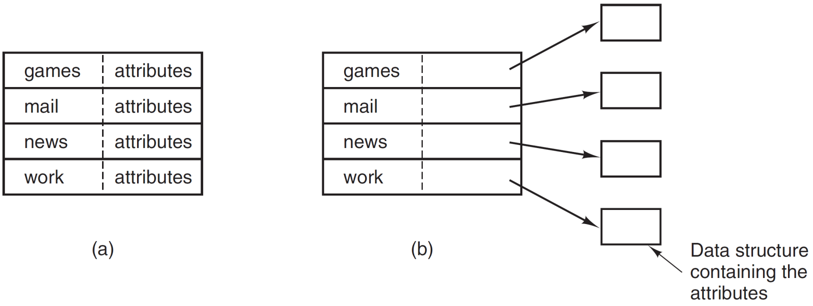
Implementing Directories

- The main function of the directory system is to map the ASCII name of the file onto the information needed to locate the data.
- Where the attributes should be stored?:
    - Store them directly in the directory entry.
    - Store them in the i-nodes, rather than in the directory entries.

4.3 File-System Implementation

Implementing Directories

Files attributes storage:



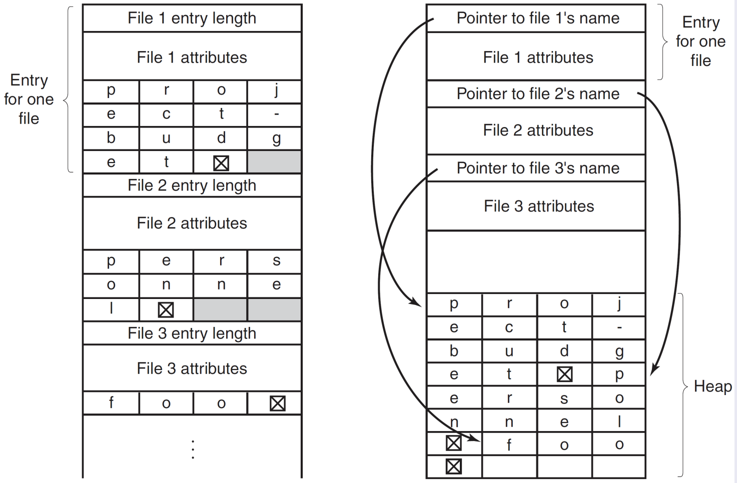
4.3 File-System Implementation

Implementing Directories - File names

- In MS-DOS, 1-8 character long name and an optional extension of 1 - 3 characters.
- In UNIX Version 7, 1 - 14 characters, including any extensions.
- Nearly all modern operating systems has variable-length file names.
- Reserve 255 characters for a name (in previous slide designs):
  - Directory header is fixed and have a variable name length terminated by a special character (a) (-ve gaps when deleted).
  - Keep file names on heap and pointing to them from the header (b) (page faults when loaded).

4.3 File-System Implementation

Implementing Directories - File names



(a) (b)

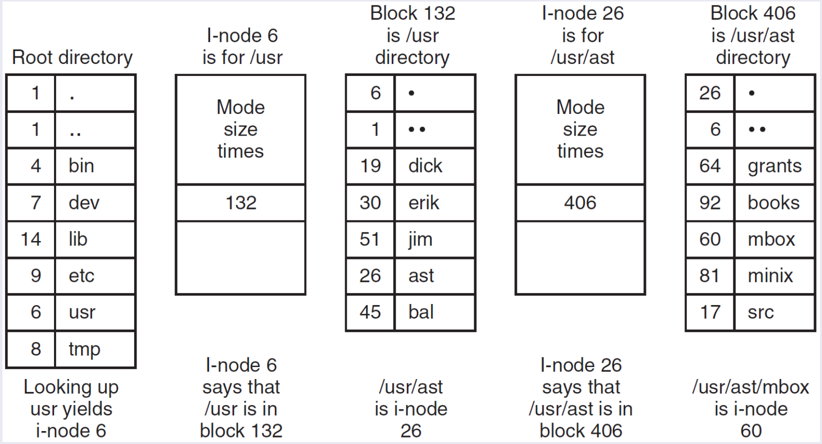
4.3 File-System Implementation

Implementing Directories - Searching a directory for a file name

- Directories are searched linearly from beginning to end when a file name has to be looked up.
- For extremely long directories, linear searching can be slow. To speed up the search:
    - a **hash table** is used in each directory.
    - Another way is to **cache** the results of searches.
- To add a file in a hashed directory:
    - the table (hash) entry corresponding to the hash code is inspected.
    - If it is unused, a pointer is placed there to the file entry.
    - If that slot is already in use, a linked list is constructed, headed at the table entry and threading through all entries with the same hash value.

4.3 File-System Implementation

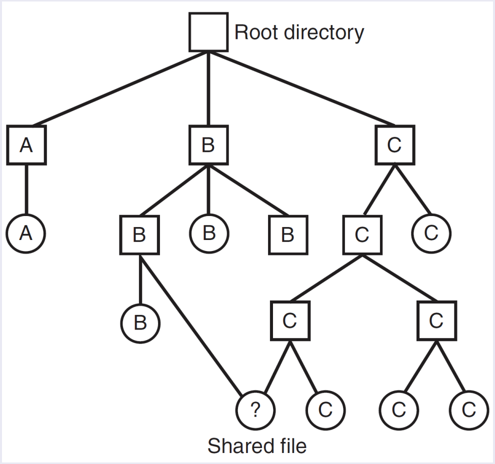
Finding a File in Linux - */usr/ast/mbox*



Root directory
1 .
1 ..
4 bin
7 dev
14 lib
9 etc
6 usr
8 tmp
Looking up usr yields i-node 6
I-node 6 is for /usr
Mode size times
132
I-node 6 says that /usr is in block 132
Block 132 is /usr directory
6 •
1 ••
19 dick
30 erik
51 jim
26 ast
45 bal
/usr/ast is i-node 26
I-node 26 is for /usr/ast
Mode size times
406
I-node 26 says that /usr/ast is in block 406
Block 406 is /usr/ast directory
26 •
6 ••
64 grants
92 books
60 mbox
81 minix
17 src
/usr/ast/mbox is i-node 60

Shared Files

The connection between B's directory and the shared file is called a link.



4.3 File-System Implementation

Shared Files

Two kind of links:

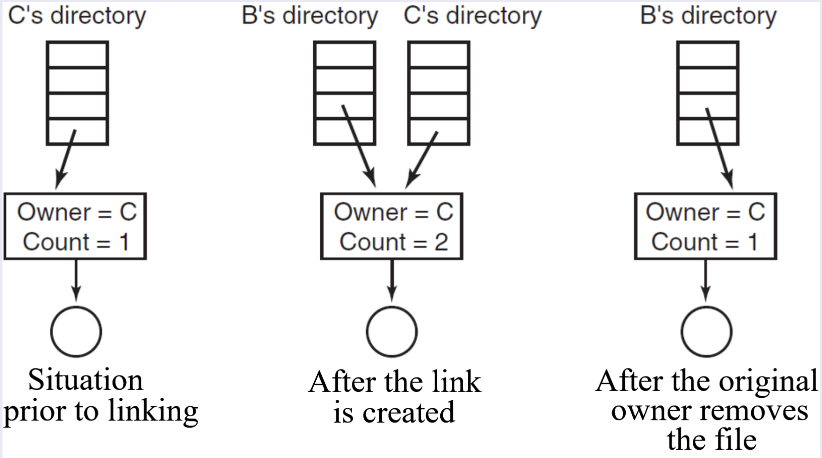
1. **Hard linking:** The directories point to the file i-node.
2. **Symbolic linking:** B links to one of C's files by having the system create a new file, of type **LINK**, and entering that file in B's directory. The new file contains just the path name of the file to which it is linked.

Drawbacks:

- For hard linking, the owner of the files cannot delete the file if there are shared users. Owner cannot know about the users.
- For symbolic linking, extra overhead for locating the shared file.
- For both if copied to a tape, there will be two copies instead of one.

4.3 File-System Implementation

Hard Link

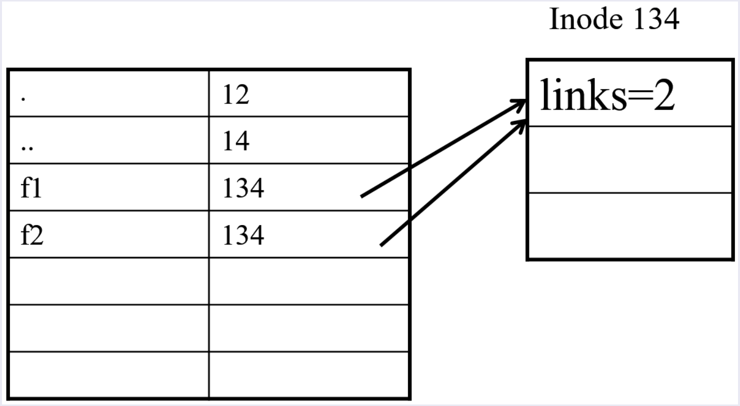


4.3 File-System Implementation

Hard Link

Both files point to the same inode

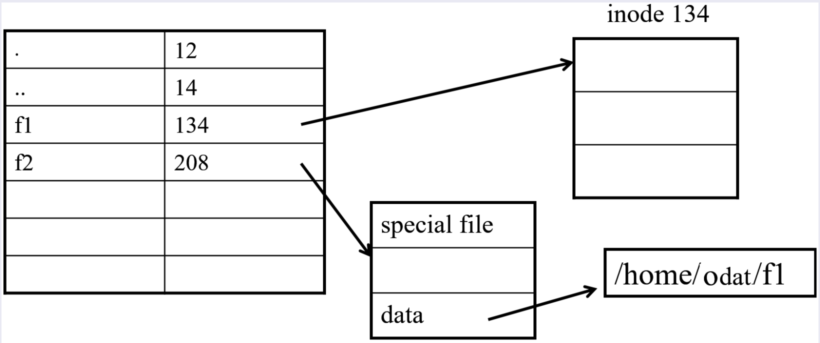
```
ln /home/odat/f1 /home/odat/f2
```



Soft Link

Files point to different inodes

```
ln -s /home/odat/f1 /home/odat/f2
```



4.3 File-System Implementation

Log-structured File Systems (LFS)

- CPUs get faster and RAM memories get larger, disk caches are also increasing rapidly.
- The bottleneck is the disk seek time.
- Consequently, it is now possible to satisfy a very substantial fraction of all read requests directly from the file-system cache, with no disk access needed.

4.3 File-System Implementation

Log-structured File Systems (LFS)

- The basic idea is to structure the entire disk as a great big log.
- Periodically all the buffered writes are written to the disk in a single segment, at the end of the log.
- At the start of each segment is a segment summary, telling what can be found in the segment.
- Opening a file now consists of using the map (maps i-node to actual place on disk) to locate the i-node for the file.
- Once the i-node has been located, the addresses of the blocks can be found from it. All of the blocks will themselves be in segments, somewhere in the log.

4.3 File-System Implementation

Log-structured File Systems (LFS)

- Many existing segments may have blocks that are no longer needed.
- For example, if a file is overwritten, its i-node will now point to the new blocks, but the old ones will still be occupying space in previously written segments.
- LFS has a **cleaner** thread that spends its time scanning the log circularly to compact it.
- The disk is a big circular buffer, where the writer thread adding new segments to the front and the cleaner thread removing old ones from the back.

4.3 File-System Implementation

Journaling File Systems

- Keep a log of what the file system is going to do before it does it.
- If the system crashes before it can do its planned work, upon rebooting, the system can look in the log to see what was going on at the time of the crash and finish the job.
- NTFS, ext3 and ReiserFS. OS X offers journaling file systems as an option.

4.3 File-System Implementation

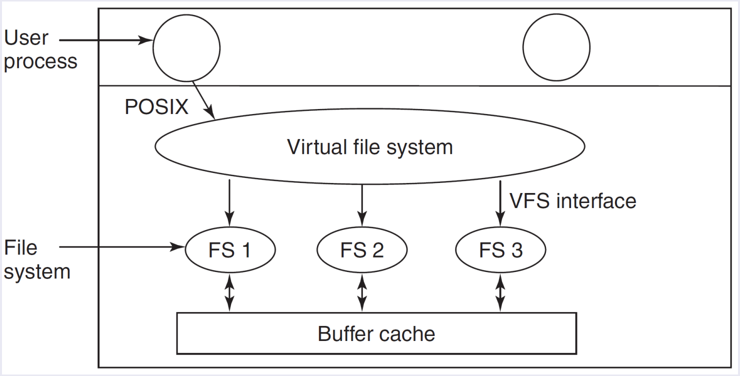
Journaling File Systems - What is the problem?

- To remove a file in Linux three steps are required:
  1. Remove the file from its directory.
  2. Release the i-node to the pool of free i-nodes.
  3. Return all the disk blocks to the pool of free disk blocks.
- If a crash happens at any point before ending the operation then inconsistency happens.
- Journaling first writes a log entry listing the three actions to be completed.
- The log entry is then written to disk.
- Only after the log entry has been written, do the various operations begin.
- After the operations complete successfully, the log entry is erased.

4.3 File-System Implementation

Virtual File Systems

The key idea is to abstract out that part of the file system that is common to all file systems and put that code in a separate layer that calls the underlying concrete file systems to actually manage the data.


User process
POSIX
Virtual file system
VFS interface
File system
FS 1
FS 2
FS 3
Buffer cache