File Management

Contents

- File Overview: File Naming, File Structure, File Types, File Access, File Attributes, File Operations, Single Level, two Level and Hierarchical Directory Systems, File System Layout.
- Implementing Files: Contiguous allocation, Linked List Allocation, Linked List Allocation using Table in Memory, I-nodes.
- Directory Operations, Path Names, Directory Implementation, Shared Files
- Free Space Management: Bitmaps, Linked List

Introduction

- A file is a collection of correlated information which is recorded on secondary or non-volatile storage like magnetic disks, optical disks, and tapes.
- It is a method of data collection that is used as a medium for giving input and receiving output from that program.
- Three essential requirements for long-term information storage:
  1. It must be possible to store a very large amount of information.
  2. The information must survive the termination of the process using it.
  3. Multiple processes must be able to access the information at once.
- **Files** are logical units of information created by processes. A disk will usually contain thousands or even millions of them, each one independent of the others.
- A file system is a method of organizing files on physical media, such as hard disks, CD's, and flash drives.
- Information stored in files must be **persistent**, that is, not be affected by process creation and termination.

File Naming

Extension	Meaning
.bak	Backup file
.c	C source program
.gif	Compuserve Graphical Interchange Format image
.hlp	Help file
.html	World Wide Web HyperText Markup Language document
.jpg	Still picture encoded with the JPEG standard
.mp3	Music encoded in MPEG layer 3 audio format
.mpg	Movie encoded with the MPEG standard
.o	Object file (compiler output, not yet linked)
.pdf	Portable Document Format file
.ps	PostScript file
.tex	Input for the TEX formatting program
.txt	General text file
.zip	Compressed archive

Figure 5-1. Some typical file extensions.

File Structure

- Files can be structured in any of several ways. Three common possibilities are depicted in Fig. 5-2.

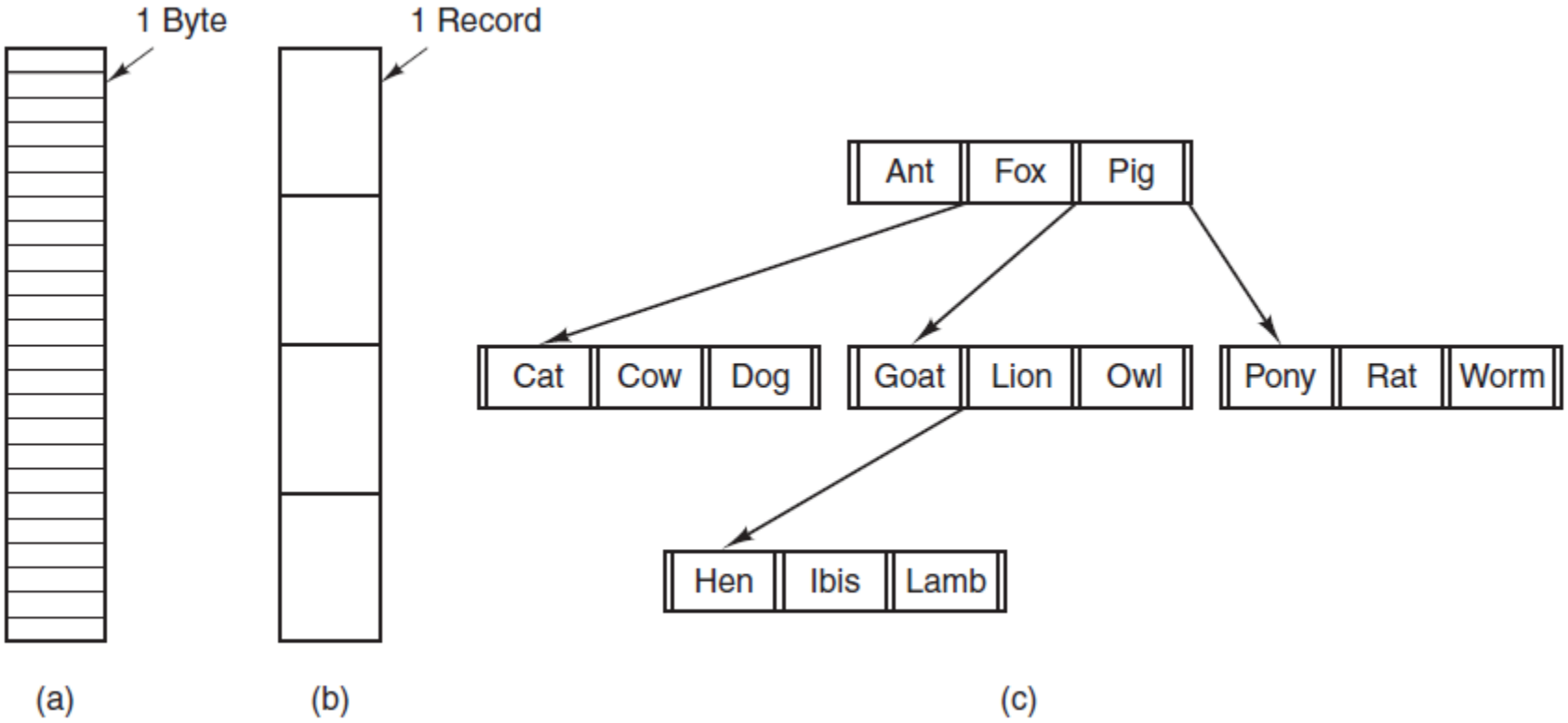


Figure 5-2. Three kinds of files. (a) Byte sequence. (b) Record sequence.(c) Tree.

- The file in Fig. 5-2(a) is an unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes.
- Both UNIX and Windows use this approach.
- Fig. 5-2(b), a file is a sequence of fixed-length records, each with some internal structure.
- No current general-purpose system uses this model as its primary file system any more, but back in the days of 80-column punched cards and 132-character line printer paper this was a common model on mainframe computers.
- Fig. 5-2(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.
- It is used on some large mainframe computers for commercial data processing.

File Types

- Many operating systems support several types of files. UNIX and Windows, for example, have regular files and directories.
- **Regular files** are the ones that contain user information. Application programs are responsible for understanding the structure and content of any specific regular file. All the files of Fig. 5-2 are regular files. Eg. word file, excel file etc.
- Regular files are generally either ASCII files or binary files(eg. executable files, compiled programs, spreadsheets, graphics(image) files etc.). ASCII files consist of lines of text. eg. c/c++, perl, HTML
- **Directories** are system files for maintaining the structure of the file system.
- **Character special files** are related to input/output and used to model serial I/O devices, such as terminals, printers, and networks.
- **Block special files** it is type of device file 1 block at a time(1 block=512 bytes to 32 kb) ,are used to model disks.

- For example, in Fig. 5-3(a) we see a simple executable binary file taken from an early version of UNIX. Although technically the file is just a sequence of bytes, the operating system will execute a file only if it has the proper format.

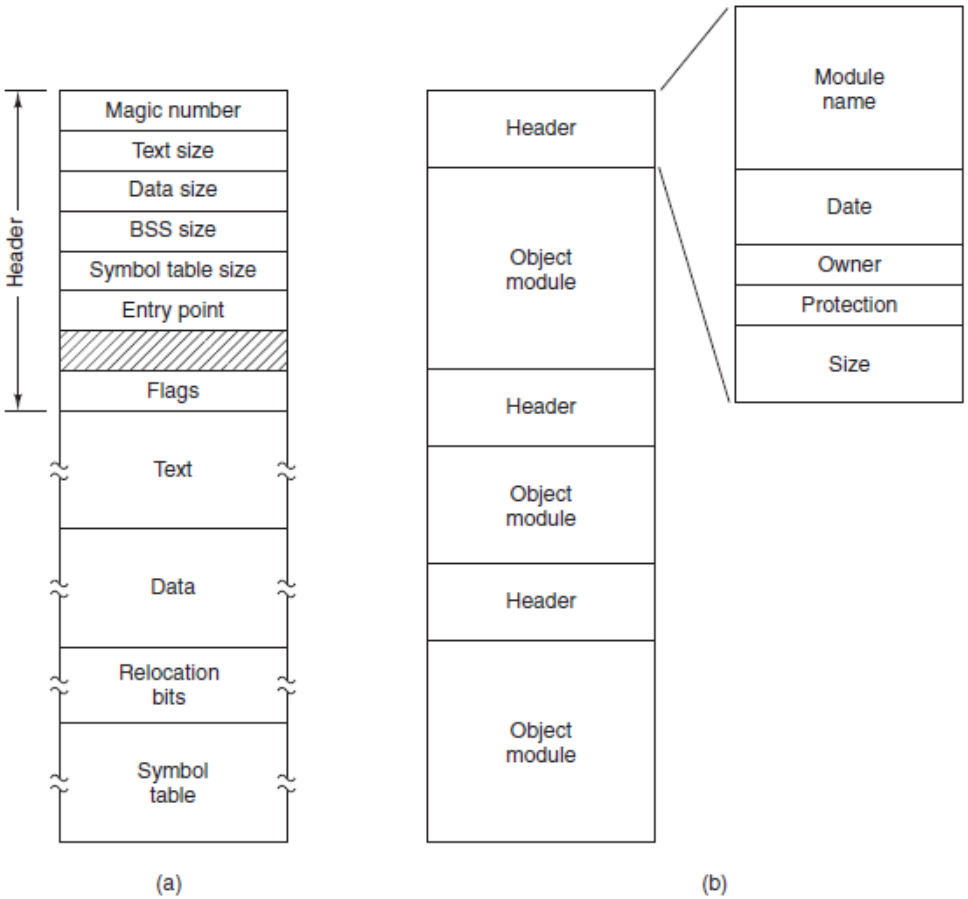


Figure 5-3. (a) An executable file. (b) An archive.

- It has five sections: header, text, data, relocation bits, and symbol table.
- The header starts with a so-called **magic number**, identifying the file as an executable file. Then come the sizes of the various pieces of the file, the address at which execution starts, and some flag bits.
- Following the header are the text and data of the program itself. These are loaded into memory and relocated using the relocation bits. The symbol table is used for debugging.
- Second example of a binary file is an archive, also from unix.
- It consists of a collection of library procedures (modules) compiled but not linked. Each one is prefaced by a header telling its name, creation date, owner, protection code, and size. Just as with the executable file, the module headers are full of binary numbers.

File Access

- File access is a process that determines the way that files are accessed and read into memory.
- **Sequential Access**, records are accessed in a certain pre-defined sequence, information stored in the file is also processed one by one . eg. magnetic tape. operations: read next, write next, rest/rewind
- **Random Access** is also called **direct random access**. This method allow accessing the record directly. Each record has its own address on which can be directly accessed for reading and writing. User now says "read n" rather than "read next".
- **Indexed Sequential Access** It is built on top of Sequential access. It uses an Index to control the pointer while accessing files.
- Random access files are essential for many applications, for example, database systems.
- If an airline customer calls up and wants to reserve a seat on a particular flight, the reservation program must be able to access the record for that flight without having to read the records for thousands of other flights first.

File Attributes

- A file has a name and data. Moreover, it also stores meta information like file creation date and time, location, current size, last modified date, etc. All this information is called the attributes of a file system.

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

Figure 5-4. Some possible file attributes.

File Operations

- The most common system calls relating to files:

1. Create: The file is created with no data.

2. Delete: When the file is no longer needed, it has to be deleted to free up disk space.

3. Open: Before using a file, a process must open it.

4. Close: When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space.

5. Read: Data are read from file.

6. Write : Data are written to the file again, usually at the current position.

7. Append :This call is a restricted form of write. It can add data only to the end of the file.

8. Seek. For random-access files, a method is needed to specify from where to take the data.

9. Get attributes: Processes often need to read file attributes to do their work.

10. Set attributes : Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible.

11. Rename : It frequently happens that a user needs to change the name of an existing file.

An Example Program Using File-System Calls

```
#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 */

    /* 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 */
}
```

Figure 4-5. A simple program to copy a file.

File-system implementation

- From now, we have only view the file system from the users point of view.
- Now it's time to view the file system from the implementor point of view.
- File system implementors are interested in how files and directories are stored, how disk space for file system is managed, and how to make everything work efficiently and reliably.

File-System Layout

- File systems are stored on disks. 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.
- The **partition table** gives the starting and ending addresses of each partition of the disk. This table gives the starting and ending addresses of each partition.

- Whenever the computer system is booted up, the bios reads in and executes the master boot record(MBR).
- The very first thing that the master boot record program does is, locate the active partition, read in its first block, that is called as the **boot block** and execute it.
- The **superblock** contains all the key parameters about the file system and is read into memory when the computer is booted or the file system is first touched.

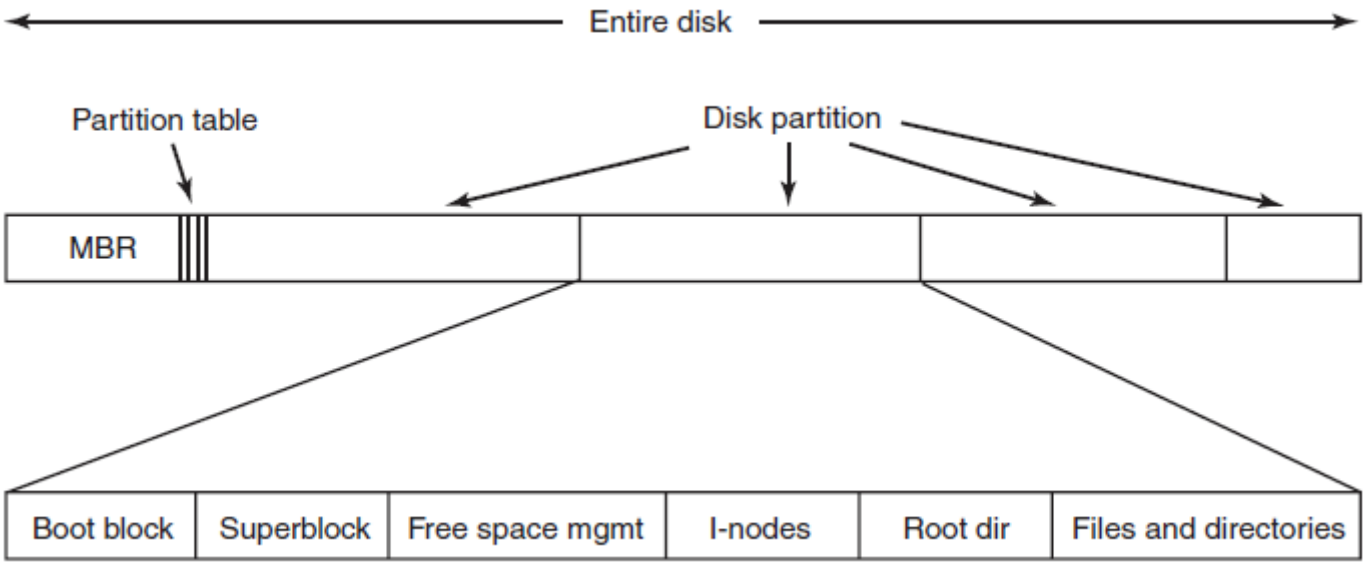


Figure 4-9. A possible file-system layout.

Implementing Files

- The most important issue in implementing file storage is keeping track of which disk blocks go with which file.
- There are various types of file allocations method few of them are:

Contiguous allocation

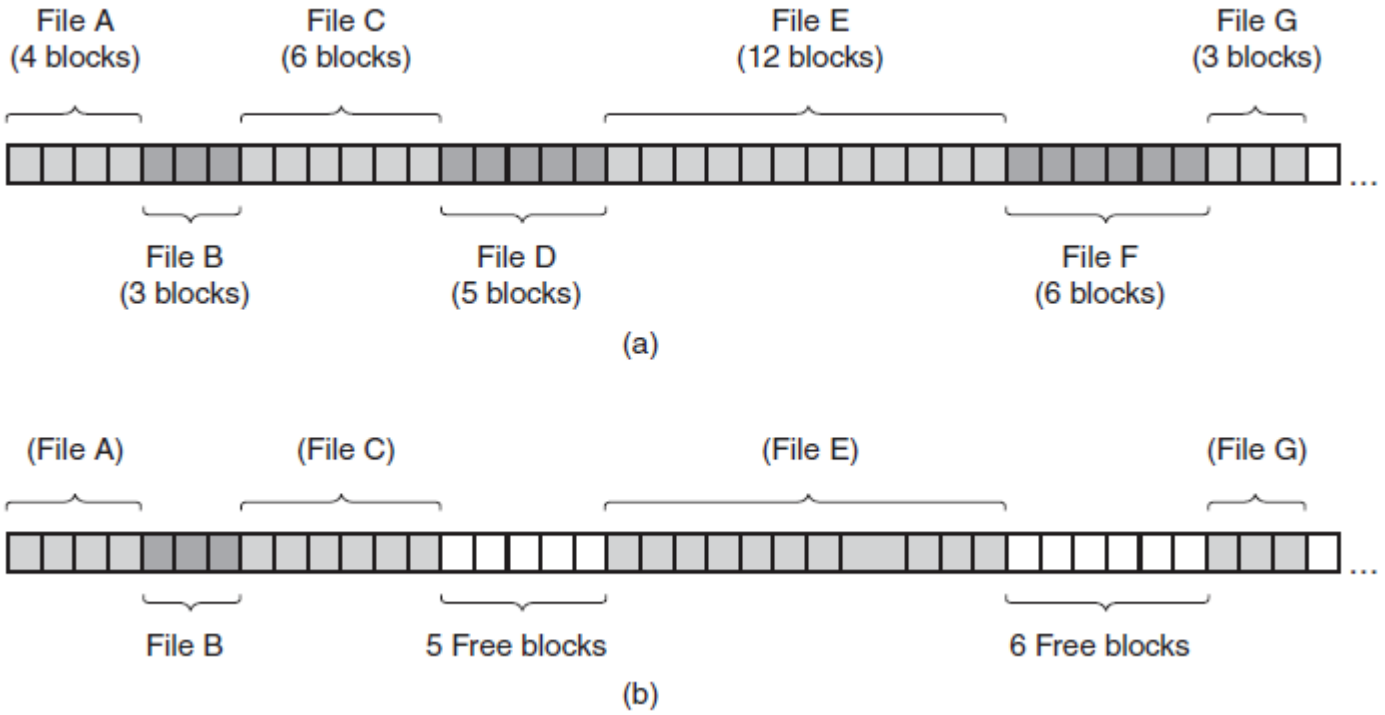


Figure 4-10. (a) Contiguous allocation of disk space for 7 files. (b) The state of the disk after files D and F have been removed.

- Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk. The word contiguous means continuous.

Advantages:

- In the contiguous allocation, sequential and direct access both are supported.
- This is very fast and the number of seeks is minimal in the contiguous allocation method.

Disadvantages:

- Contiguous allocation method suffers internal as well as external fragmentation.
- In terms of memory utilization, this method is inefficient.
- It is difficult to increase the file size because it depends on the availability of contiguous memory.

Linked-List Allocation

- Linked List allocation solves all problems of contiguous allocation. In linked list allocation, each file is considered as the linked list of disk blocks.
- Each disk block allocated to a file contains a pointer which points to the next disk block allocated to the same file.

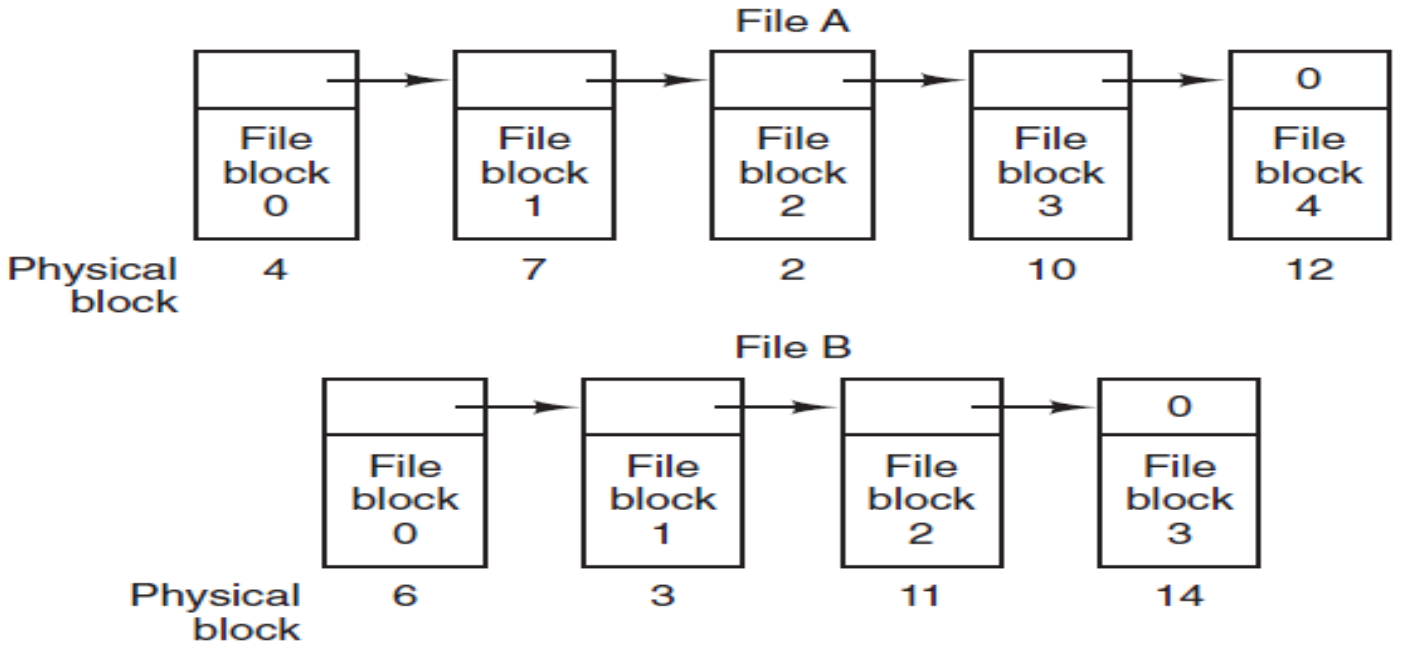


Figure 4-11. Storing a file as a linked list of disk blocks

Advantages:

- In terms of the file size, this scheme is very flexible.
- We can easily increase or decrease the file size
- There is no external fragmentation with linked allocation.

Disadvantages:

- Random Access is not provided.
- Linked allocation is comparatively slower than contiguous allocation.
- Pointers require some space in the disk blocks.
- The pointer is extra overhead on the system due to the linked list.

Linked-List Allocation Using a Table in Memory

- File Allocation Table overcomes this drawback of linked list allocation. In this scheme, a file allocation table is maintained, which gathers all the disk block links. The table has one entry for each disk block and is indexed by block number.
- It simply accesses the file allocation table, read the desired block entry from there and access that block.
- This is the way by which the random access is accomplished by using FAT(File Allocation Table). It is used by MS-DOS and pre-NT Windows versions.

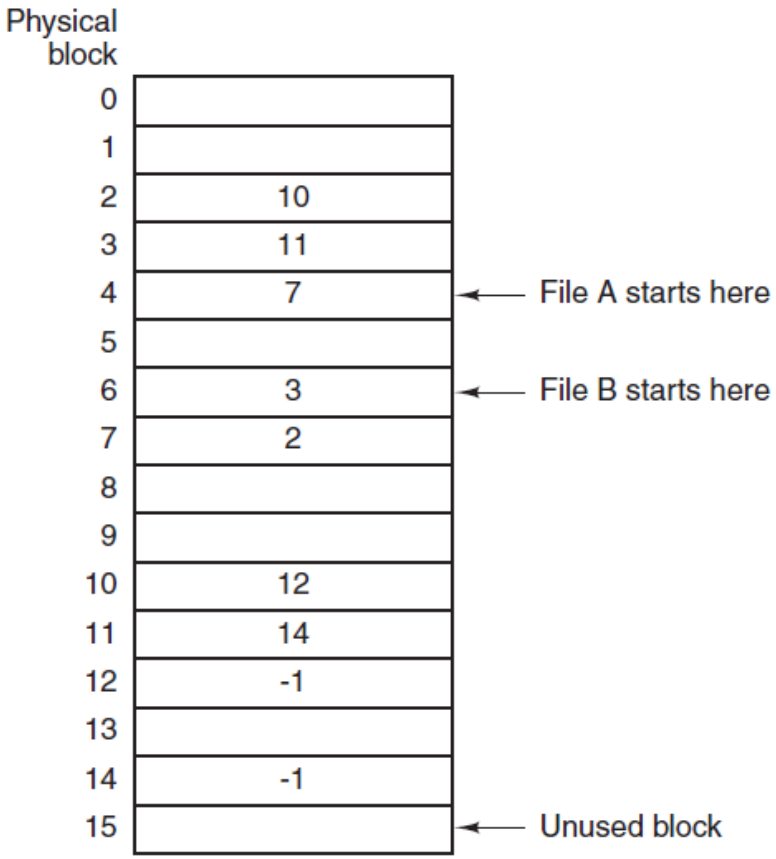


Figure 4-12. Linked-list allocation using a file-allocation table in main memory.

Advantages:

- Uses the whole disk block for data.
- A bad disk block doesn't cause all successive blocks lost.
- Random access is provided although its not too fast.
- Only FAT needs to be traversed in each file operation.

Disadvantages:

- Each Disk block needs a FAT entry.
- FAT size may be very big depending upon the number of FAT entries.
- Number of FAT entries can be reduced by increasing the block size but it will also increase Internal Fragmentation.

I-nodes (index-node)

- To keeping track of which blocks belong to which file is to associate with each file a data structure called an **i-node** (**index-node**), which lists the attributes and disk addresses of the file's blocks.
- I-node is a data structure which is used to identify which block belongs to which file.
- It contains the attributes and disk addresses of the file's blocks. Unlike the in-memory table the i-node need to be in memory only when the corresponding file is open.
- Extra disk blocks may be reserved to store the block addresses of a large file.

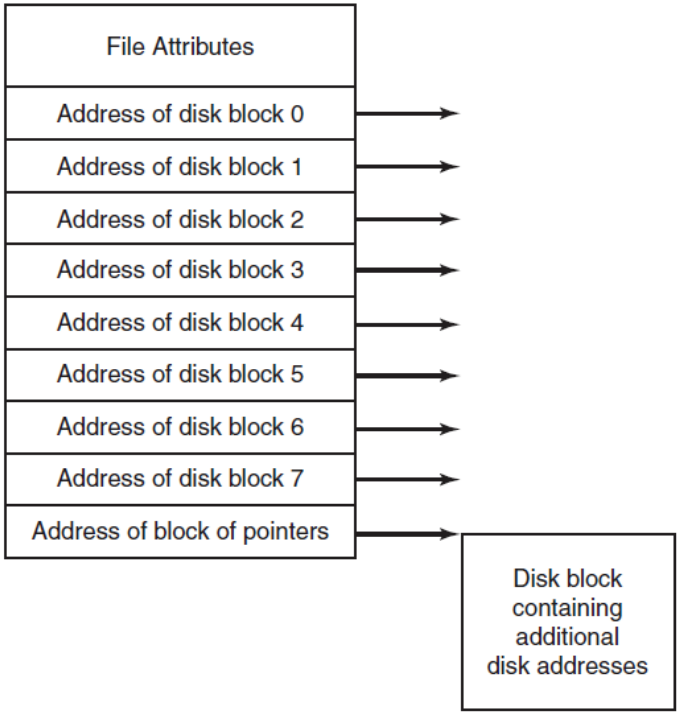


Figure 4-13. An example i-node.

Directories

- To keep track of files, file systems normally have **directories** or **folders**, which are themselves files.
- There are several logical structures of a directory, these are given below.

1. Single-Level Directory Systems

- The simplest form of directory syster

is having one directory containing all the files. Sometimes it is called the **root directory**

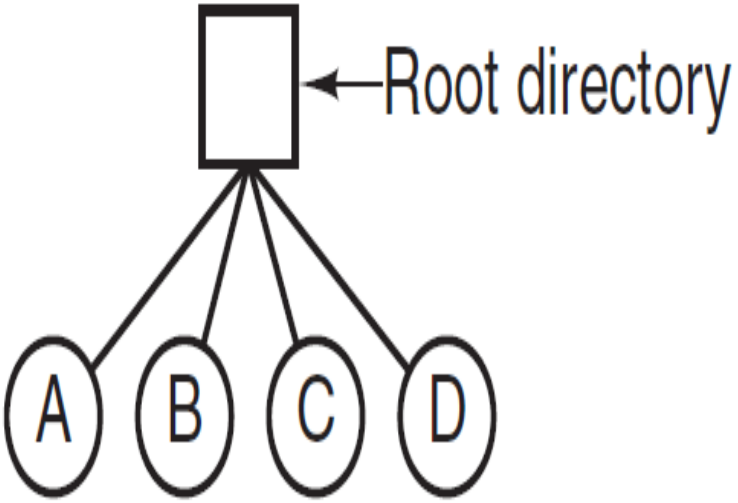


Figure 4-6. A single-level directory system containing four files.

2. Two-level directory systems

- In this two-level directory system, names chosen by one user don't interfere with names chosen by a different user and there is no any problem that is caused by the same name occurring in two or more than two directories.
- The two-level directory system even on a single-user PC, is inconvenient.
- Since it is common for the users, want to group their files together in logical ways.

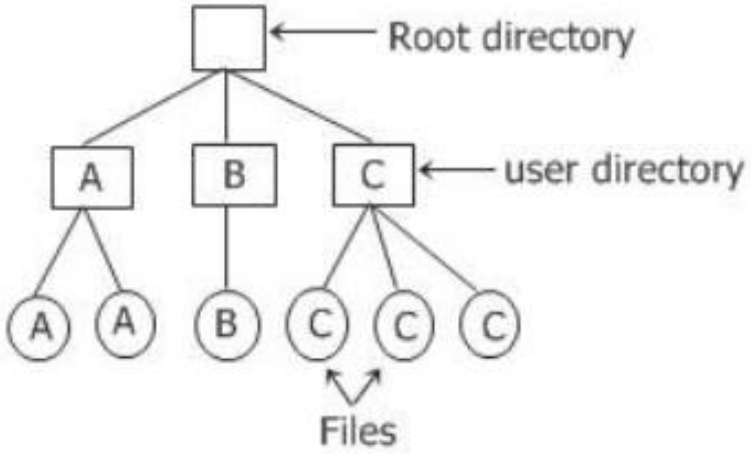


Figure 4-6. A single-level directory system containing four files.

3. Hierarchical Directory Systems

- With hierarchical approach, each user on the computer system can have as many directories as are needed, so that files can be grouped together in natural ways.
- Here, the directories *A*, *B*, and *C* contained in the root directory each belong to a different user, two of whom have created subdirectories for projects they are working on.

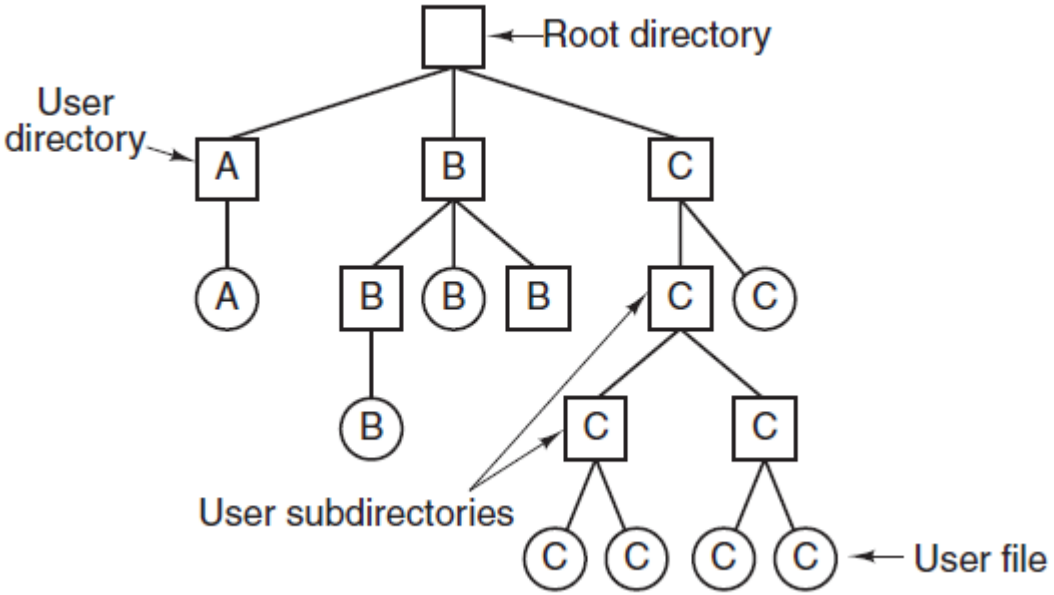


Figure 4-7. A hierarchical directory system.

Directory Operations

Create: A directory is created. It is empty except for dot and dotdot, which are put there automatically by the system.

Delete: A directory is delete. Here, only those directory can be deleted which are empty.

Opendir: Directories can be read. But before reading any directory, it must be opened first.

Therefore to list all the files present in a directory, a listing program opens that required directory to read out the name of all files that this directory contains.

Closedir: Directory should be closed just to free up the internal table space when it has been read.

Readdir: This call returns the next entry in an open directory.

Rename: Directory can also be renamed just like the files.

Link: Linking is a technique that allows a file to appear in more than one directory.

Unlink: A directory entry is removed.

Path Names

- When the file system is organized as a directory tree, some way is needed for specifying file names.
- Two different methods are commonly used.
    1. Absolute path name
    2. Relative path name
- An **absolute path name** consisting of the path from the root directory to the file. As an example, the path /usr/ast/mailbox means that the root directory contains a subdirectory usr, which in turn contains a subdirectory ast, which contains the file mailbox.
- Absolute path names always start at the root directory and are unique.

Windows: \usr\ast\mailbox

UNIX : /usr/ast/mailbox

MULTICS : >usr>ast>mailbox

- The **relative path name** is used in conjunction with the concept of the working directory (also called the current directory).
- A user can designate one directory as the current working directory in which case all the path names not beginning at the root directory are taken relative to the working directory.
- For example, if the current working directory is */usr/ast*, then the file whose absolute path is */usr/ast/mailbox* can be referenced simply as *mailbox*.

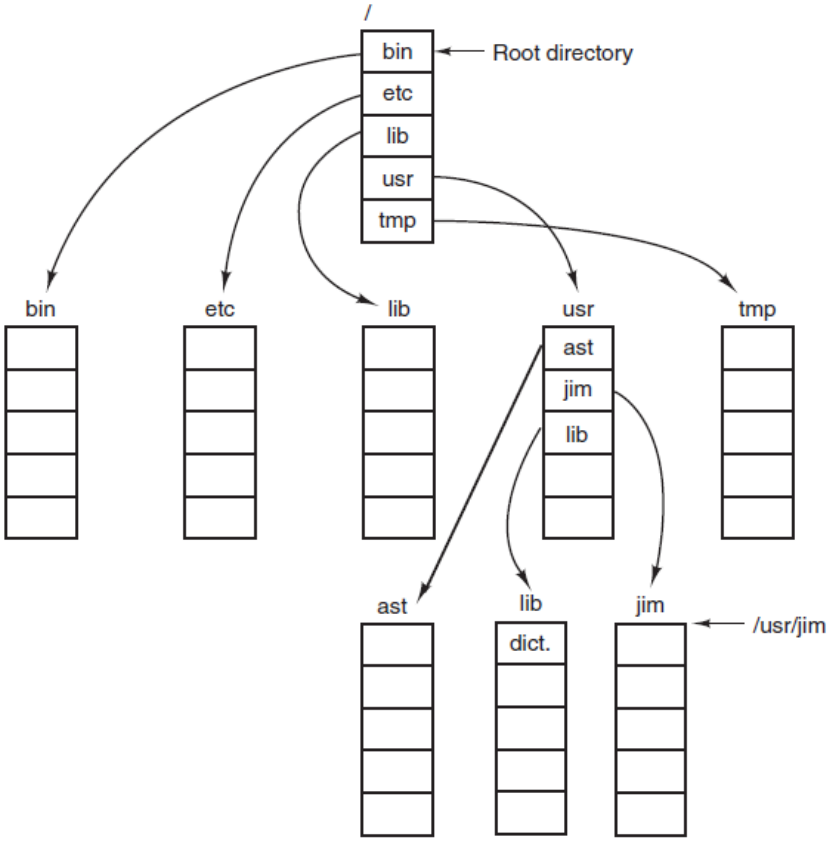


Figure 4-8. A UNIX directory tree.

Implementing Directories

(a) A simple directory fixed size entries disk addresses and attributes in directory entry

(b) Directory in which each entry just refers to an i-node

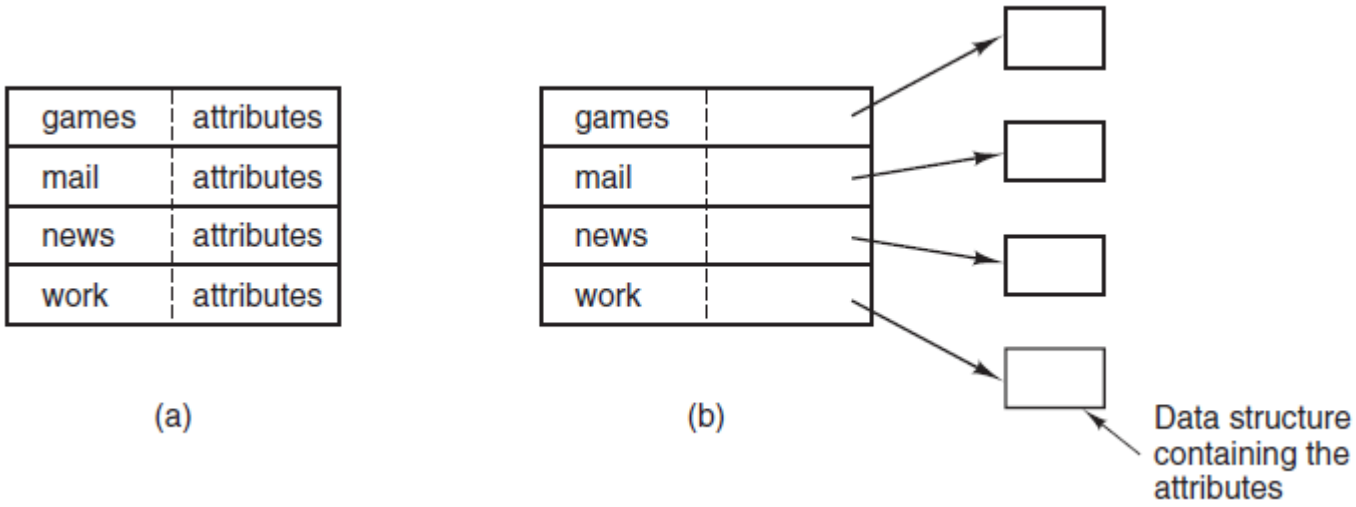


Figure 4-14. (a) A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry. (b) A directory in which each entry just refers to an i-node.

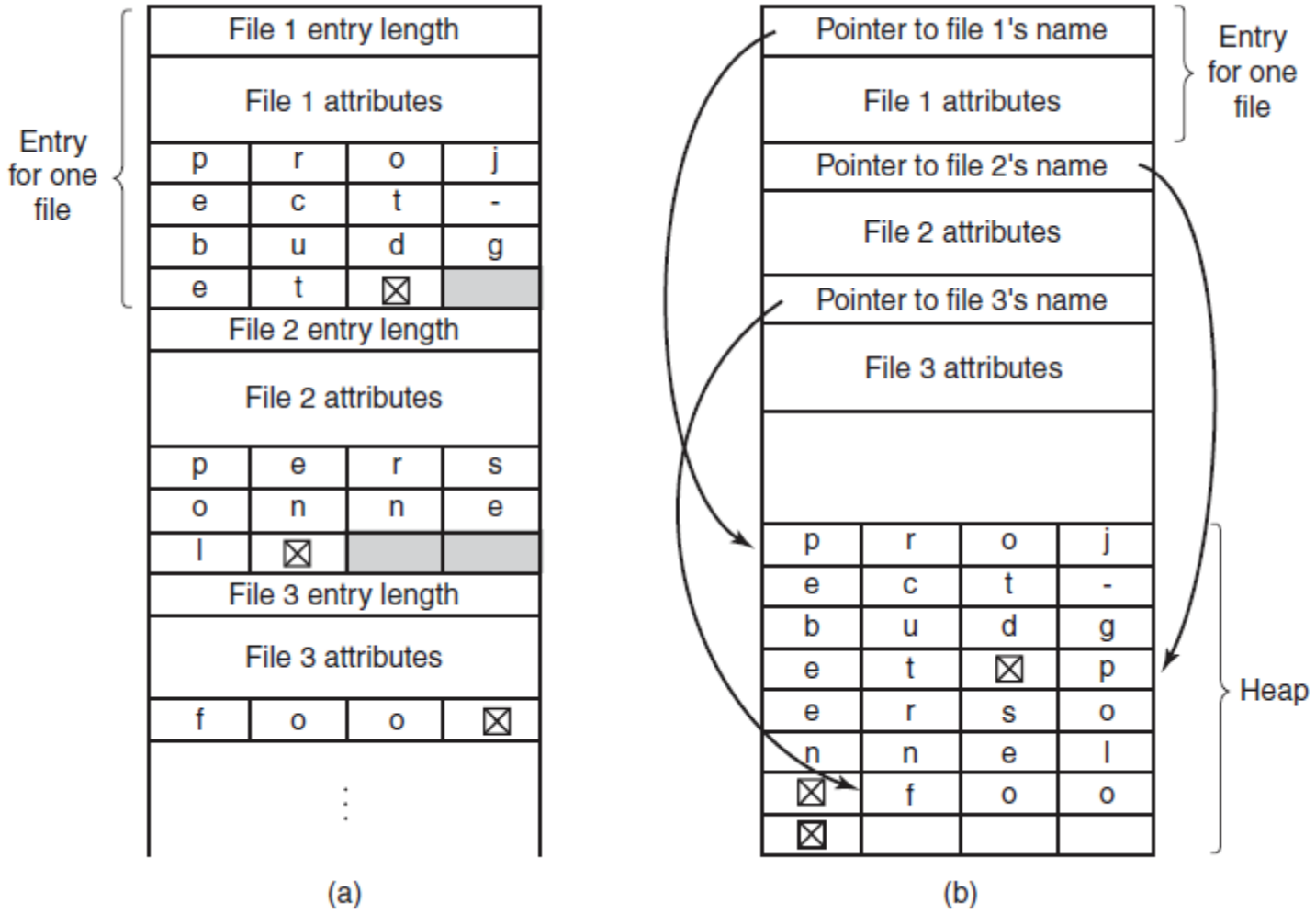


Figure 4-15. Two ways of handling long file names in a directory. (a) In-line.(b) In a heap.

- One alternative is to give up the idea that all directory entries are the same size.
- **In-line method,** each directory entry contains a fixed portion, typically starting with the length of the entry, and then followed by data with a fixed format, usually including the owner, creation time, protection information, and other attributes.
- Each file name is terminated by a special character:
  - Usually 0;
  - Represented in the figure by a box with a cross in it ☒
- A disadvantage of this method is that when a file is removed, a variable-sized gap is introduced into the directory into which the next file to be entered may not fit.

- Another way to handle variable-length names is to make the directory entries themselves all fixed length and keep the file names together in a heap at the end of the directory,
- This method has the advantage that when an entry is removed, the next file entered will always fit there.
- the heap must be managed and page faults can still occur while processing file names.

Shared Files

- When several users are working together on a project, they often need to share files.
- As a result, it is often convenient for a shared file to appear simultaneously in different directories belonging to different users.
- Sharing files is convenient but also introduces problems like if the original file or the shared file are appended with new features then it will not be updated in both the copies of the file, it is updated only at its original location.

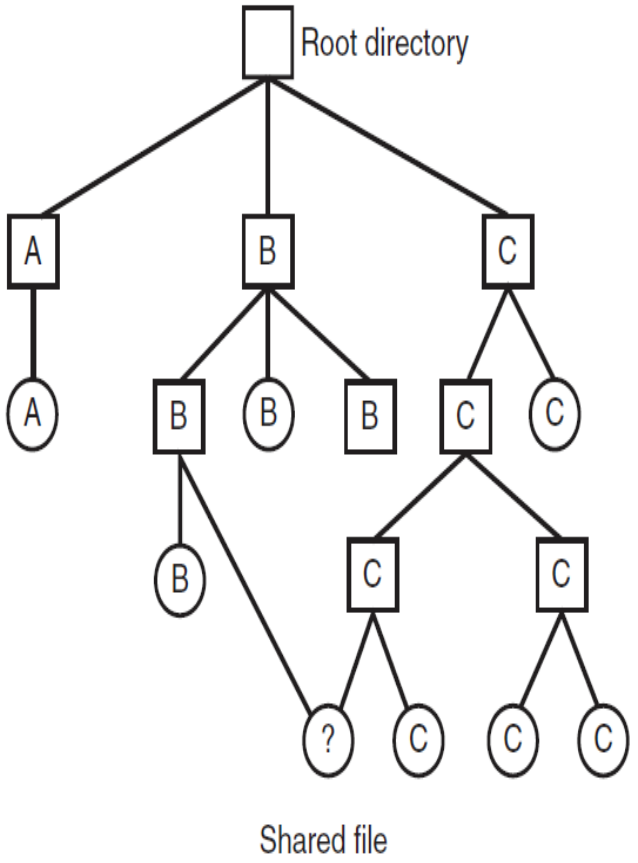


Figure 4-16. File system containing a shared file.

- Sharing files is convenient, but it also introduces some problems.

This problem can be solved by two ways:

- i-node
- Symbolic linking

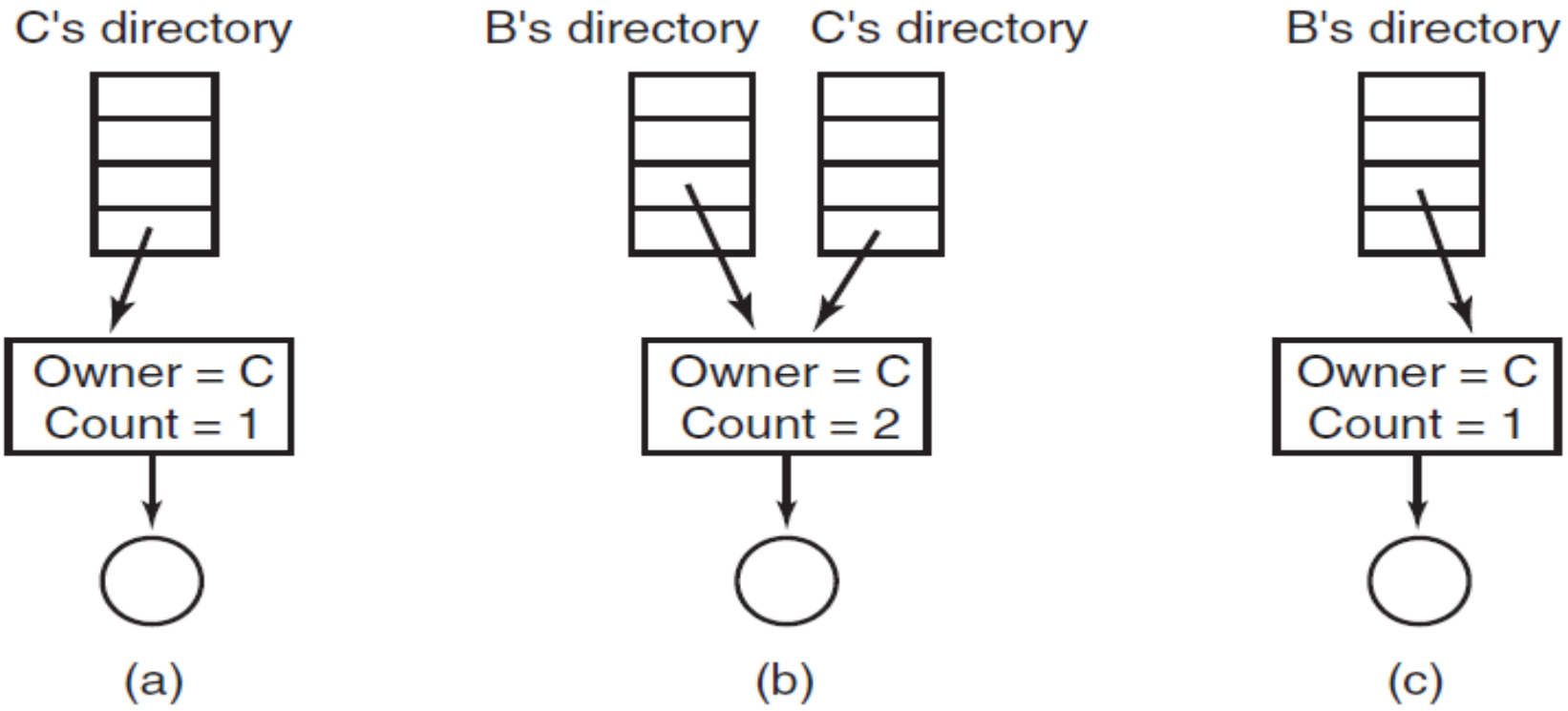


Figure 4-17. (a) Situation prior to linking. (b) After the link is created. (c) After the original owner removes the file.

- In the first solution, disk blocks are not listed in directories, but in a little data structure associated with the file itself. The directories would then point just to the little data structure. This is the approach used in UNIX (where the little data structure is the i-node).
- In the second solution, *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. When *B* reads from the linked file, the operating system sees that the file being read from is of type LINK, looks up the name of the file, and reads that file.(**symbolic linking**, to contrast it with traditional (hard) linking).

Disk-Space Management

- All the files are normally stored on disk one of the main concerns of file system is management of disk space.

Block Size

- The main question that arises while storing files in a fixed-size blocks is the size of the block.
- If the block is too large space gets wasted and if the block is too small time gets waste. So, to choose a correct block size some information about the file-size distribution is required.

Keeping track of free blocks

- After a block size has been finalized the next issue that needs to be catered is how to keep track of the free blocks. In order to keep track there are two methods that are widely used:
- **Using a linked list:** Using a linked list of disk blocks with each block holding as many free disk block numbers as will fit.

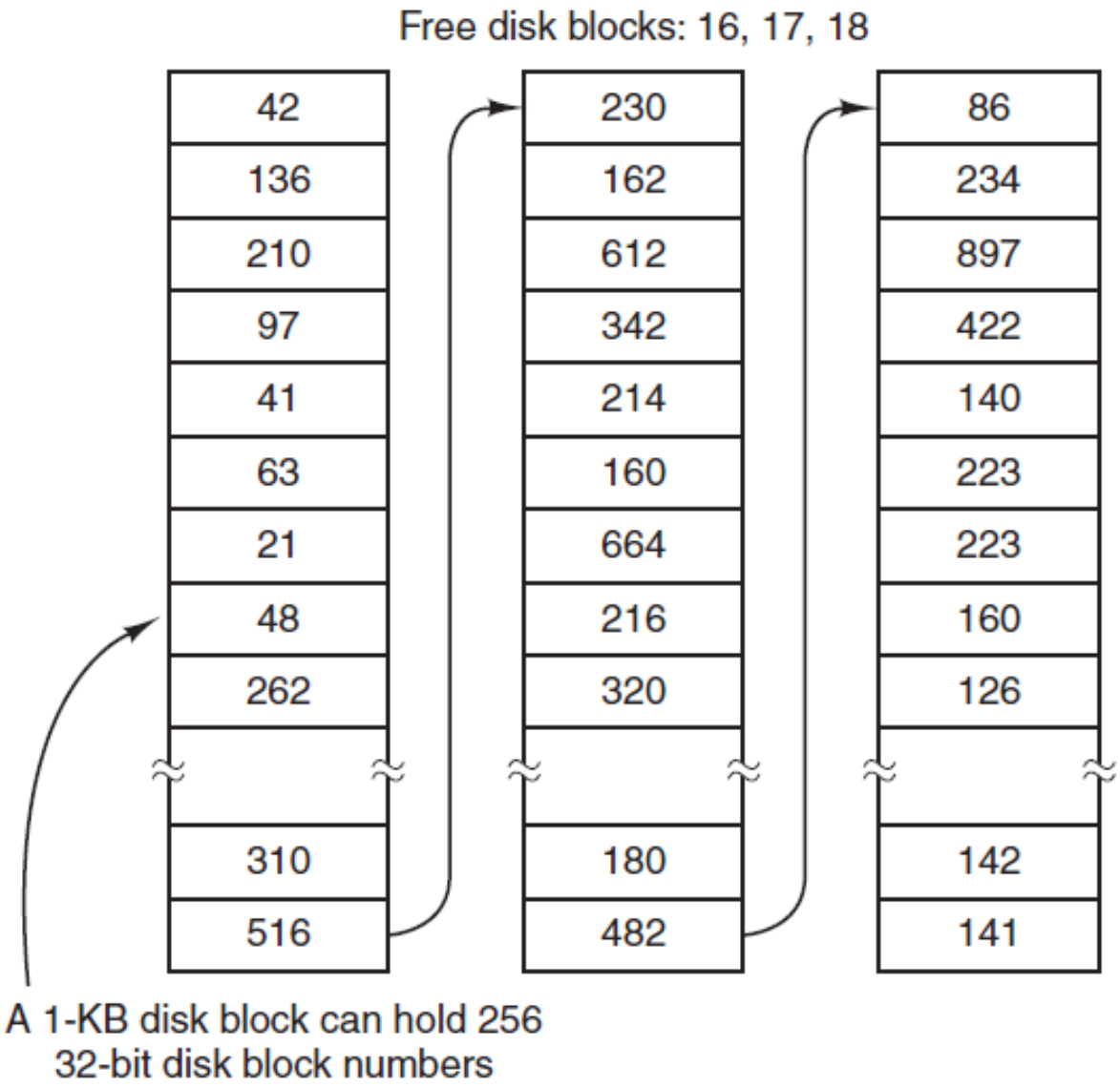
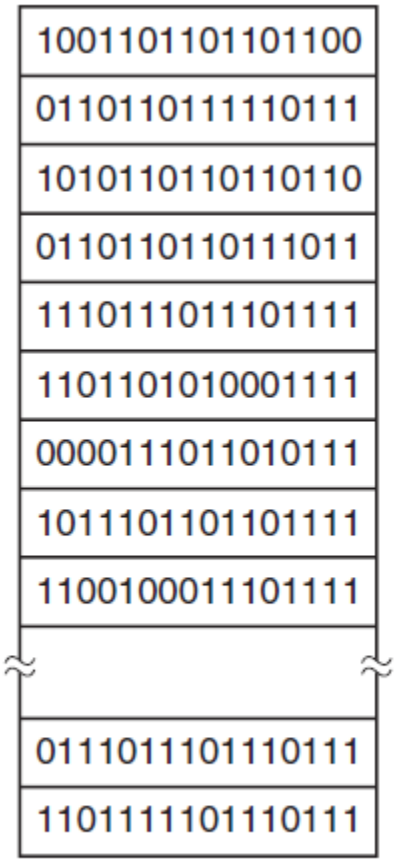


Figure. Storing the free list on a linked list.

- Using a Bitmap: A disk with n blocks has a bitmap with n bits. Free blocks are represented using 1's and allocated blocks as 0's



A bitmap