File Management

Objectives

• To explain the function of file systems.
• To describe the interfaces to file systems.
• To discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures.
• To explore file-system protection.

File Concept

- Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks.
- So that the computer system will be convenient to use, the operating system provides a uniform logical view of stored information.
- The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**.
- Files are mapped by the operating system onto physical devices.
- These storage devices are usually nonvolatile, so the contents are persistent between system reboots.

File

- In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user.
- The information in a file is defined by its creator. Many different types of information maybe stored in a file—source or executable programs, numeric or text data, photos, music, video, and so on. A file has a certain defined structure, which depends on its type.
- A **text file** is a sequence of characters organizedinto lines (and possibly pages).
- A **source file** is a sequence of functions, each of which is further organized as declarations followed by executable statements.
- An **executable file** is a series of code sections that the loader can bring into memory and execute.

File Attributes

- A file's attributes vary from one operating system to another but typically consist of these:
- **Name**. The symbolic file name is the only information kept in human readable form.
- **Identifier**. This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type**. This information is needed for systems that support different types of files.
- **Location**. This information is a pointer to a device and to the location of the file on that device.
- **Size**. The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection**. Access-control information determines who can do reading, writing, executing, and so on.
- **Time, date, and user identification**. This information may be kept for creation, last modification, and last use.

These data can be useful for protection, security, and usage monitoring.

File Operations

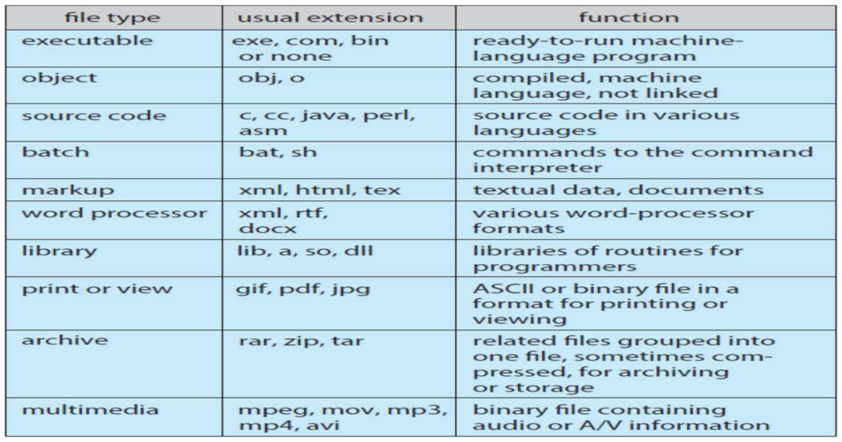
- A file is an abstract data type.
- To define a file properly, we need to consider the operations that can be performed on files.
- The operating system can provide system calls to create, write, read, reposition, delete, and truncate files.
- **Creating a file**. create a file.
- **Writing a file**. To write a file, we make a system call specifying both the name of the file and the information to be written to the file.
- Given the name of the file, the system searches the directory to find the file's location.
- The system must keep a **write pointer** to the location in the file where the next write is to take place.
- **Reading a file**. To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put.
- Again, the directory is searched for the associated entry, and the system needs to keep a **read pointer** to the location in the file where the next read is to take place.

File Operations contd…….

- **Repositioning within a file**. The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value.
- Repositioning within a file need not involve any actual I/O.
- This file operation is also known as a file **seek**.
- **Deleting a file**. To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.
- **Truncating a file**. The user may want to erase the contents of a file but keep its attributes.
- Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length—but lets the file be reset to length zero and its file space released.

File Types

- When we design a file system—indeed, an entire operating system—we always consider whether the operating system should recognize and support file types.
- If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways.



file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes com-pressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Figure 11.3 Common file types.

Access Methods

- Files store information.
- When it is used, this information must be accessed and read into computer memory.
- The information in the file can be accessed in several ways.
- Some systems provide only one access method for files.
- while others support many access methods, and choosing the right one for a particular application is a major design problem.

Sequential Access

- The simplest access method is **sequential access**. Information in the file is processed in order, one record after the other.
- This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.
- Reads and writes make up the bulk of the operations on a file.
- **A read operation—read next()—**reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location.
- Similarly, the **write operation—write next()**—appends to the end of the file and advances to the end of the newly written material (the new end of file).
- Such a file can be reset to the beginning, and on some systems, a program may be able to skip forward or backward *n* records for some integer *n*—perhaps only for *n* = 1. Sequential access, which is depicted in fig.1 is based on a tape model of a file and works as well on sequential-access devices as it does on random-access ones.

Sequential Access

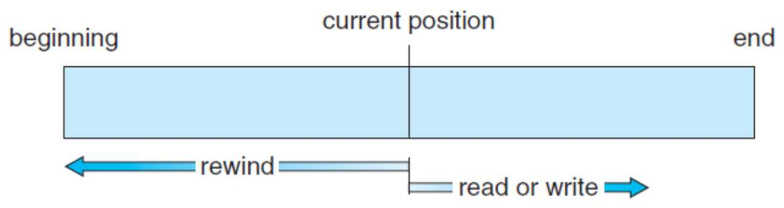


Figure 11.4 Sequential-access file.

Direct access or relative access

- Another method is **direct access** (or) **relative access**
- Here, a file is made up of fixed-length **logical records** that allow programs to read and write records rapidly in no particular order.
- The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records.
- Thus, we may read block 14, then read block 53, and then write block 7.
- There are no restrictions on the order of reading or writing for a direct-access file.
- Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type.
- When a query concerning a particular subject arrives, we compute which block contains the answer and then read that block directly to provide the desired information.

Directory and Disk Structure

- Next, we consider how to store files. Certainly, no general-purpose computer stores just one file.
- There are typically thousands, millions, even billions of files within a computer.
- Files are stored on random-access storage devices, including hard disks, optical disks, and solid-state (memory-based) disks.

Directory and Disk Structure

- a general-purpose computer system has multiple storage devices, and those devices can be sliced up into volumes that hold file systems.
- Each volume that contains a file system must also contain information about the files in the system.
- This information is kept in entries in a **device directory** or **volume table of contents**.
- The device directory (mor commonly known simply as the **directory**) records information—such as name, location, size, and type—for all files on that volume.
- Figure shows a typical file-system organization.

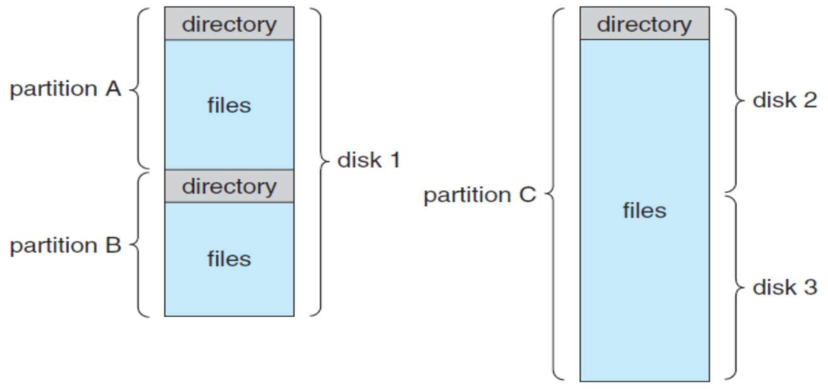


Figure 11.7 A typical file-system organization.

Directory Overview

- The directory can be viewed as a symbol table that translates file names into their directory entries.
- If we take such a view, we see that the directory itself can be organized in many ways. The organization must allow us to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory.
- In this section, we examine several schemes for defining the logical structure of the directory system.

Directory Overview

- When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

• **Search for a file.** We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names, and similar names may indicate a relationship among files, we may want to be able to find all files whose names match a particular pattern.

• **Create a file**. New files need to be created and added to the directory.

• **Delete a file**. When a file is no longer needed, we want to be able to remove it from the directory.

- • **List a directory.** We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
- • **Rename a file.** Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.

• **Traverse the file system**. We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals.

Logical structure of a directory-Single Level Directory

- The simplest directory structure is the single-level directory.
- All files are contained in the same directory, which is easy to support and understand
- A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user.
- Since all files are in the same directory, they must have unique names.
- Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases.
- It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system.
- Keeping track of so many files is a daunting task.

Logical structure of a directory-Single Level Directory

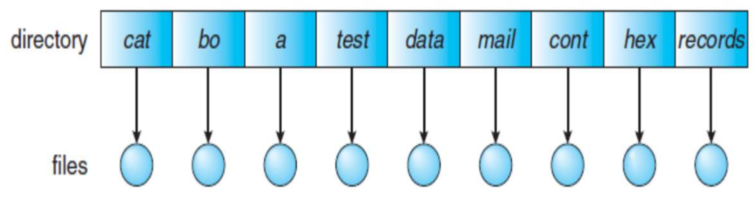


Figure 11.9 Single-level directory.

Two-Level Directory

- As we have seen, a single-level directory often leads to confusion of file names among different users.
- The standard solution is to create a separate directory for each user.
- In the two-level directory structure, each user has his own **user file directory (UFD)**.
- The UFDs have similar structures, but each lists only the files of a single user.
- When a user job starts or a user logs in, the system's **master file directory (MFD)** is searched.
- The MFD is indexed by user name or account number, and each entry points to the UFD for that user
- When a user refers to a particular file, only his own UFD is searched.
- Thus, different users may have files with the same name, as long as all the file names within each UFD are unique.
- To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists.
- To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

Two-Level Directory

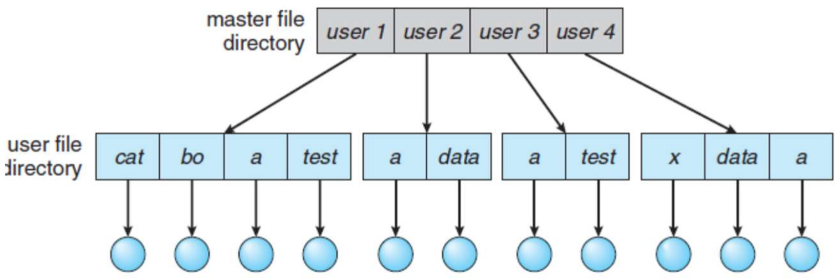


Figure 11.10 Two-level directory structure.

Two-Level Directory

- The user directories themselves must be created and deleted as necessary.
- A special system program is run with the appropriate user name and account information.
- The program creates a new UFD and adds an entry for it to the MFD.
- The execution of this program might be restricted to system administrators.

Tree-Structured Directories

- Once we have seen how to view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height
- This generalization allows users to create their own subdirectories and to organize their files accordingly.
- A tree is the most common directory structure.
- The tree has a root directory, and every file in the system has a unique path name.
- A directory (or subdirectory) contains a set of files or subdirectories.
- A directory is simply another file, but it is treated in a special way.
- All directories have the same internal format.
- One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and delete directories.

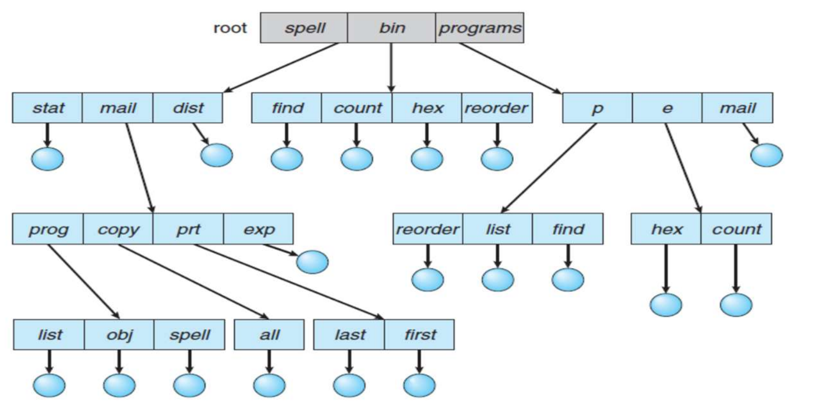


Figure 11.11 Tree-structured directory structure.

File-System Mounting

- The File must be opened before it is used, a file system must be mounted before it can be available to processes on the system.
- More specifically, the directory structure may be built out of multiple volumes, which must be mounted to make them available within the file-system name space.
- The mount procedure is straightforward.
- The operating system is given the name of the device and the mount point—the location within the file structure where the file system is to be attached.
- Some operating systems require that a file system type be provided, while others inspect the structures of the device and determine the type of file system
- Typically, a mount point is an empty directory
- For instance, on a UNIX system, a file system containing a user’s home directories might be mounted as /home; then, to access the directory structure within that file system, we could precede the directory names with /home, as in /home/jane.
- Mounting that file system under /users would result in the path name /users/jane, whichwe could use to reach the same directory.

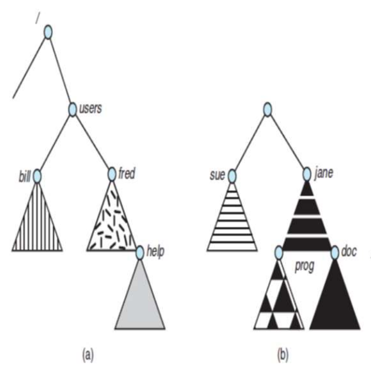


Figure 11.14 File system. (a) Existing system. (b) Unmounted volume.

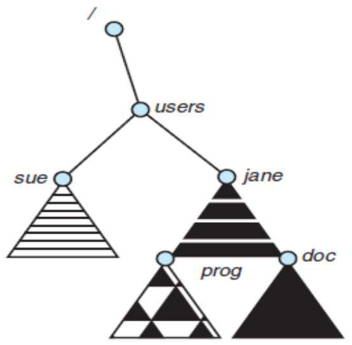


Figure 11.15 Mount point.

File Sharing

- A **network operating system** is an operating system that provides features such as file sharing across the network, along with a communication scheme that allows different processes on different computers to exchange messages.
- A computer running a network operating system acts autonomously from all other computers on the network, although it is aware of the network and is able to communicate with other networked computers.

Aspects of File Sharing

- To implement sharing and protection, the system must maintain more file and directory attributes than are needed on a single-user system.
- Although many approaches have been taken to meet this requirement, most systems have evolved to use the concepts of file (or directory) **owner** (or **user**) and **group**.
- The owner is the user who can change attributes and grant access and who has th most control over the file.
- The group attribute defines a subset of users who can share access to the file.
- For example, the owner of a file on a UNIX system can issue all operations on a file, while members of the file's group can execute one subset of those operations, and all other users can execute another subset of operations.
- Exactly which operations can be executed by group members and other users is definable by the file's owner.

Remote File Systems

- With the advent of networks communication among remote computers became possible.
- Networking allows the sharing of resources spread across a campus or even around the world
- Through the evolution of network and file technology, remote file-sharing methods have changed.
- The first implemented method involves manually transferring files between machines via programs like ftp.
- The second major method uses a **distributed file system (DFS)** in which remote directories are visible from a local machine.
- In some ways, the third method, the **World Wide Web**, is a reversion to the first.
- A browser is needed to gain access to the remote files, and separate operations (essentially a wrapper for ftp) are used to transfer files.

Remote File Systems-The Client–Server Model

- Remote file systems allow a computer to mount one or more file systems from one or more remote machines.
- In this case, the machine containing the files is the **server**, and the machine seeking access to the files is the **client**.
- The client–server relationship is common with networked machines.
- Generally, the server declares that a resource is available to clients and specifies exactly which resource (in this case, which files) and exactly which clients.
- A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client–server facility.

Remote File Systems-The Distributed Information Systems

- To make client–server systems easier to manage, **distributed information systems**, also known as **distributed naming services**, provide unified access to the information needed for remote computing.
- The **domain name system (DNS)** provides host-name-to-network-address translations for the entire Internet.
- Before DNS became widespread, files containing the same information were sent via e-mail or ftp between all networked hosts.

Protection

- When information is stored in a computer system, we want to keep it safe from physical damage (the issue of reliability) and improper access (the issue of protection).

Protection-Types of Access

- The need to protect files is a direct result of the ability to access files. Systems that do not permit access to the files of other users do not need protection. Thus, we could provide complete protection by prohibiting access.
- Alternatively, we could provide free access with no protection. Both approaches are too extreme for general use. What is needed is controlled access.
- Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:
- • **Read**. Read from the file.
- • **Write**. Write or rewrite the file.
- • **Execute**. Load the file into memory and execute it.
- • **Append**. Write new information at the end of the file.
- • **Delete**. Delete the file and free its space for possible reuse.
- • **List**. List the name and attributes of the file.

Protection-Access Control

- The most common approach to the protection problem is to make access dependent on the identity of the user.
- Different users may need different types of access to a file or directory.
- The most general scheme to implement identity dependent access is to associate with each file and directory an **access-control list (ACL)** specifying user names and the types of access allowed for each user.
- When a user requests access to a particular file, the operating system checks the access list associated with that file.
- If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

Protection-Access Control

- To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:

• **Owner**. The user who created the file is the owner.

• **Group**. A set of users who are sharing the file and need similar access is a group, or work group.

• **Universe**. All other users in the system constitute the universe.

Example Protection-Access Control

- To illustrate, consider a person, Sara, who is writing a new book. She has
- hired three graduate students (Jim, Dawn, and Jill) to help with the project.
- The text of the book is kept in a file named book.tex. The protection associated with this file is as follows:
- Sara should be able to invoke all operations on the file.
- • Jim, Dawn, and Jill should be able only to read and write the file; they should not be allowed to delete the file.
- • All other users should be able to read, but not write, the file. (Sara is interested in letting as many people as possible read the text so that she can obtain feedback.)

File systems Structure

- **File systems** provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily.
- A file system poses two quite different design problems.
- The first problem is defining how the file system should look to the user.
- This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files.
- The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.
- The file system itself is generally composed of many different levels.
- The structure shown in Figure is an example of a layered design. Each level in the design uses the features of lower levels to create new features for use by higher levels.

File systems Structure

- The **I/O control** level consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system. A device driver can be thought of as a translator.
- The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.
- Each physical block is identified by its numeric disk address
- The **file-organization module** knows about files and their logical blocks, as well as physical blocks.
- By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer.
- Finally, the **logical file system** manages metadata information.
- Metadata includes all of the file-system structure except the actual data (or contents of the files).
- The logical file system manages the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name.

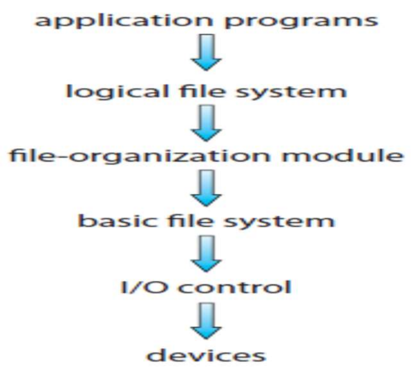


Figure 12.1 Layered file system.

File-System Implementation

- operating systems implement open()and close() systems calls for processes to request access to file contents.
- In this section, we delve into the structures and operations used to implement file-system operations.

File-System Implementation

- To create a new file, an application program calls the logical file system.
- The logical file system knows the format of the directory structures.
- To create anew file, it allocates a new FCB. (Alternatively, if the file-system implementation creates all FCBs at file-system creation time, an FCB is allocated from the set of free FCBs.)
- The system then reads the appropriate directory into memory, updates it with the new file name and FCB, and writes it back to the disk. A typical FCB is shown in Figure 12.2

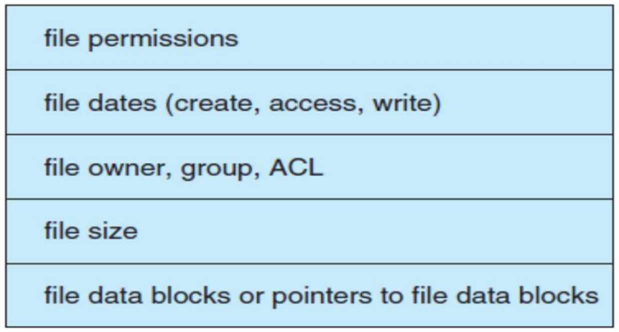

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Figure 12.2 A typical file-control block.

File-System Implementation

- Now that a file has been created, it can be used for I/O.
- First, though, it must be opened. The open() call passes a file name to the logical file system.
- The open() system call first searches the system-wide open-file table to see if the file is already in use by another process.
- If it is, a per-process open-file table entry is created pointing to the existing system-wide open-file table.
- This algorithm can save substantial overhead. If the file is not already open, the directory structure is searched for the given file name.
- Parts of the directory structure are usually cached in memory to speed directory operations. Once the file is found, the FCB is copied into a system-wide open-file table in memory.
- This table not only stores the FCB but also tracks the number of processes that have the file open.

File-System Implementation

- Next, an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table and some other fields.
- These other fields may include a pointer to the current location in the file (for the next read() or write() operation) and the access mode in which the file is open.
- The open() call returns a pointer to the appropriate entry in the per-process file-system table.
- When a process closes the file, the per-process table entry is removed, and the system-wide entry's open count is decremented.
- When all users that have opened the file close it, any updated metadata is copied back to the disk-based directory structure, and the system-wide open-file table entry is removed.

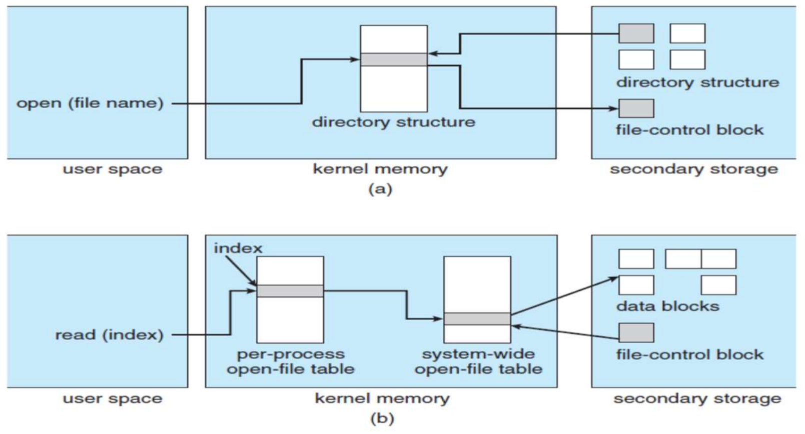


Figure 12.3 In-memory file-system structures. (a) File open. (b) File read.

Directory Implementation

- The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system.

Linear List

- The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks.
- This method is simple to program but time-consuming to execute.
- To create a new file, we must first search the directory to be sure that no existing file has the same name.
- Then, we add a new entry at the end of the directory.
- To delete a file, we search the directory for the named file and then release the space allocated to it.
- To reuse the directory entry, we can do one of several things.
- We can mark the entry as unused (by assigning it a special name, such as an all-blank name, or by including a used– unused bit in each entry), or we can attach it to a list of free directory entries.
- A third alternative is to copy the last entry in the directory into the freed location and to decrease the length of the directory.
- A linked list can also be used to decrease the time required to delete a file

Hash Table

- Another data structure used for a file directory is a hash table.
- Here, a linear list stores the directory entries, but a hash data structure is also used.
- The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list.
- Therefore, it can greatly decrease the directory search time.
- Insertion and deletion are also fairly straightforward, although some provision must be made for collisions—situations in which two file names hash to the same location.

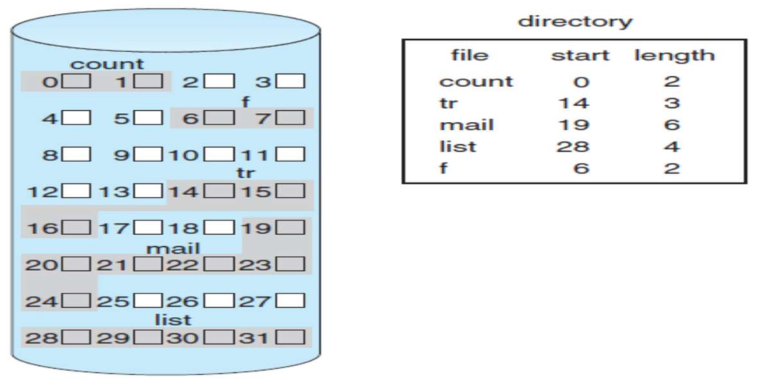
Allocation Methods

- The direct-access nature of disks gives us flexibility in the implementation of
- files. In almost every case, many files are stored on the same disk.
- The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly.
- Three major methods of allocating disk space are in wide use: contiguous, linked, and indexed.

. Contiguous Allocation

- In this scheme, each file occupies a contiguous set of blocks on the disk. For example, if a file requires n blocks and is given a block b as the starting location, then the blocks assigned to the file will be: *b, b+1, b+2,……b+n-1.* This means that given the starting block address and the length of the file (in terms of blocks required), we can determine the blocks occupied by the file.
  The directory entry for a file with contiguous allocation contains
- Address of starting block
- Length of the allocated portion.
- The *file 'mail'* in the following figure starts from the block 19 with length = 6 blocks. Therefore, it occupies *19, 20, 21, 22, 23, 24* blocks.

. Contiguous Allocation



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

Figure 12.5 Contiguous allocation of disk space.

. Contiguous Allocation

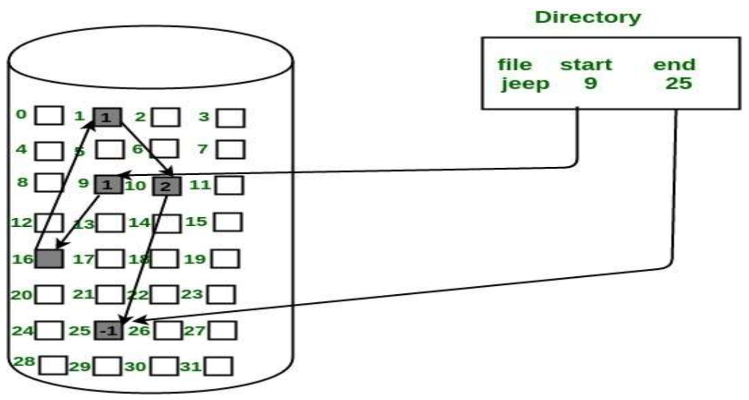
Advantages:
•Both the Sequential and Direct Accesses are supported by this.
• For direct access, the address of the kth block of the file which starts at block b can easily be obtained as (b+k).
•This is extremely fast since the number of seeks are minimal because of contiguous allocation of file blocks.

Disadvantages:
•This method suffers from both internal and external fragmentation.
•This makes it inefficient in terms of memory utilization.
•Increasing file size is difficult because it depends on the availability of contiguous memory at a particular instance.

Linked List Allocation

- In this scheme, each file is a linked list of disk blocks which **need not be** contiguous.
- The disk blocks can be scattered anywhere on the disk.
- The directory entry contains a pointer to the starting and the ending file block.
- Each block contains a pointer to the next block occupied by the file.
- *The file 'jeep' in following image shows how the blocks are randomly distributed. The last block (25) contains -1 indicating a null pointer and does not point to any other block.*

Linked List Allocation



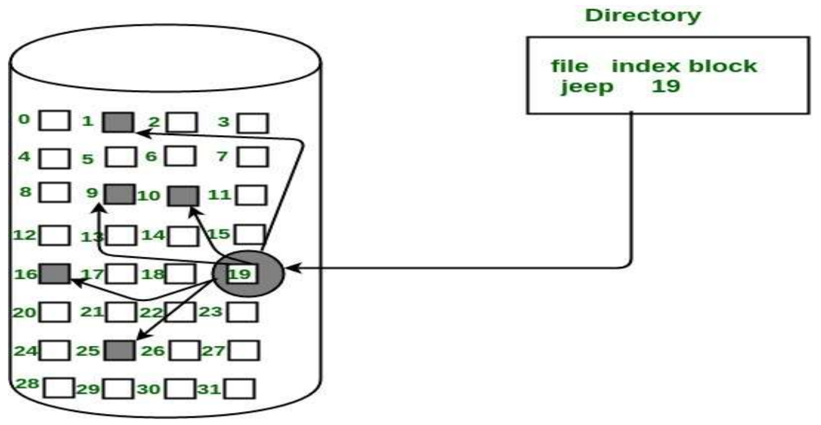
Linked List Allocation

- **Advantages:**
- This is very flexible in terms of file size. File size can be increased easily since the system does not have to look for a contiguous chunk of memory.
- This method does not suffer from external fragmentation. This makes it relatively better in terms of memory utilization.
- **Disadvantages:**
- Because the file blocks are distributed randomly on the disk, a large number of seeks are needed to access every block individually. This makes linked allocation slower.
- It does not support random or direct access. We can not directly access the blocks of a file. A block k of a file can be accessed by traversing k blocks sequentially (sequential access) from the starting block of the file via block pointers.
- Pointers required in the linked allocation incur some extra overhead.

Indexed Allocation

- **Indexed Allocation**
- In this scheme, a special block known as the **Index block** contains the pointers to all the blocks occupied by a file.
- Each file has its own index block.
- The ith entry in the index block contains the disk address of the ith file block.
- The directory entry contains the address of the index block as shown in the image:

Indexed Allocation


Directory
file index block
jeep 19
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
16 17 18 19
20 21 22 23
24 25 26 27
28 29 30 31