OPERATING SYSTEMS

Unit Objectives:

1. Explain the function of file systems.
2. Understand access methods, file sharing, file locking and directory structures.
3. Explore file-system protection.
4. Implement local file-system, remote file-system and directory structure.

Unit -6

File System: File concept; Access methods; Directory structure; File system mounting; File sharing; Protection. Implementing File System: File system structure; File system implementation; Directory implementation; Allocation methods; Free space management.

FILE SYSTEM, IMPLEMENTATION OF FILE SYSTEM

File Concept

- Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks.
- Operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file.
- Many different types of information may be stored in a file - source programs, executable programs, numeric data, text, graphic images, sound recordings, and so on.

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

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. Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.
- 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 location 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. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process current-file-position-pointer.
- Repositioning within a file. The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. 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.
- Most of the file operations mentioned involve searching the directory for the entry associated with the named file. To avoid this constant searching, many systems require that an open() system call be made before a file is first used actively.
- The open() operation takes a file name and searches the directory, copying the directory entry into the open-file table. When a file operation is requested, the file is specified via an index into this table, so no searching is required.
- The operating system uses two levels of internal tables: a per-process table and a system-wide table. The per- process table tracks all files that a process has opened. Stored in this table is information regarding the use of the file by the process.
- Each entry in the per-process table in turn points to a system-wide open-file table. The system-wide table contains process-independent information. Once a file has been opened by one process, the system-wide table includes an entry for the file.

several pieces of information are associated with an open file.

- File pointer. On systems that do not include a file offset as part of the read() and write() system calls, the system must track the last read-write location as a current-file-position pointer. This pointer is unique to each process operating on the file.
- File-open count. As files are closed, the operating system must reuse its open-file table entries, or it could run out of space in the table. Because multiple processes may have opened a file, the system must wait for the last file to close before removing the open-file table entry.
- Disk location of the file. Most file operations require the system to modify data within the file. The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.
- Access rights. Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/O requests.

File Types

- A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts-a name and an extension, usually separated by a period character.
- In this way, the user and the operating system can tell from the name alone what the type of a file is. The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.

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 compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

File Structure

- File types also can be used to indicate the internal structure of the file. Source and object files have structures that match the expectations of the programs that read them.
- For example, the operating system requires that an executable files have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is.
- This point brings us to one of the disadvantages of having the operating system support multiple file structures. If the operating system defines five different file structures, it needs to contain the code to support these file structures.
- When new applications require information structured in ways not supported by the operating system, severe problems may result.

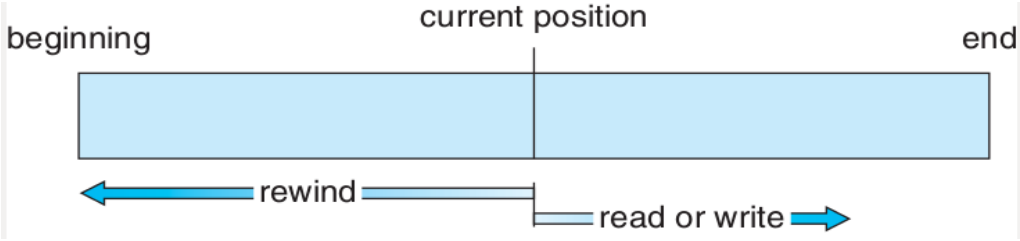
Internal File Structure

- Disk systems typically have a well-defined block size. All disk I/O is performed in units of one block(physical record), and all blocks are of the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record.
- Logical records may even vary in length. Packing a number of logical records into physical blocks is a common solution to this problem.
- The logical record size, physical block size, and packing technique determine how many logical records are in each physical block. The packing can be done either by the user's application program or by the operating system.

Access Methods

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.
- A read operation-*read next*-reads the next portion of the file and automatically advances a file pointer. Similarly, the write operation-*write next*-appends to the end of the file and advances to the end of the newly written material.



Direct Access

- A file is made up of fixed length logical records that allow programs to read and write records rapidly in no particular order.
- 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.
- For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have *read n*, where n is the block number, rather than *read next*, and *write n* rather than *write next*.

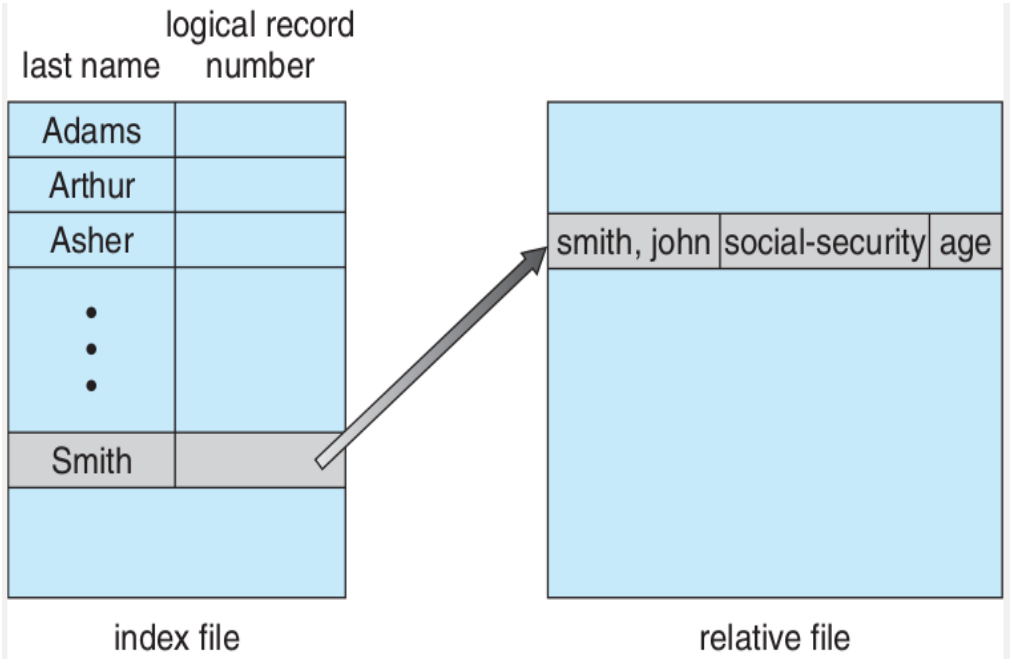
sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp ; cp = cp + 1;
write_next	write cp; cp = cp + 1;

- Assuming we have a logical record of length L, the request for record N is turned into an I/O request for L bytes starting at location L * (N) within the file (assuming the first record is N = 0).

Other Access Methods

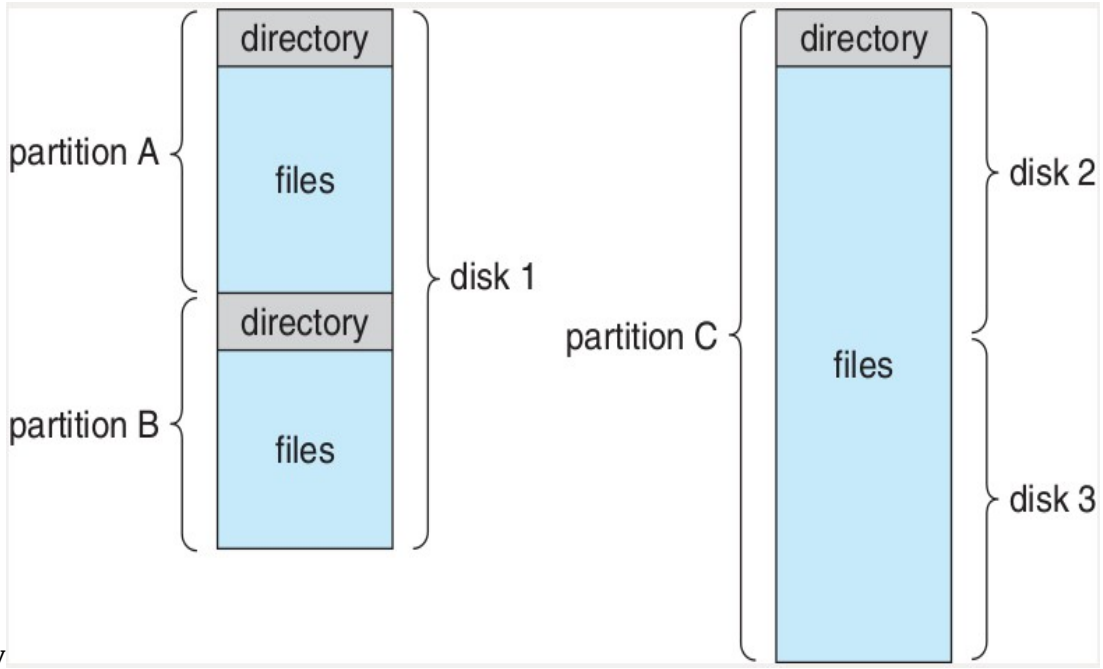
- Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file.
- To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.
- For example, a retail-price file might list the universal product codes (UPCs) for items, with the associated prices. Each record consists a 10-digit UPC and a 6-digit price, for a 16-byte record.
- If our disk has 1,024 bytes per block we can store 64 records per block. A file of 1,28,000 records would occupy about 2,000 blocks.
- By keeping the file sorted by UPC, we can define an index consisting of the first UPC in each

block. This index would have entries of 10 digits each, or 20,000 bytes, and thus could be kept in memory.



Directory and Disk Structure

- Partitioning is useful for limiting the sizes of individual file systems, putting multiple file-system types on the same device, or leaving part of the device available for other uses.
- A file system can be created on each of these parts of the disk. Any entity containing a file system is generally known as a volume.
- The volume may be a subset of a device, a whole device, or multiple devices linked together. Each volume can be thought of as a virtual disk.



Directory Overview

- The directory can be viewed as a symbol table that translates file names into their directory entries.

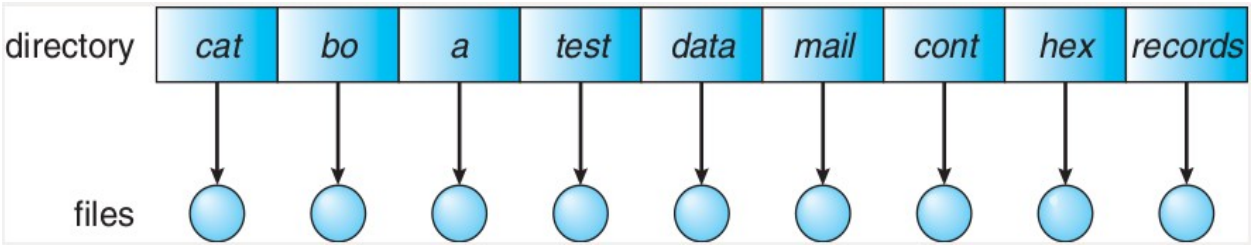
- We want to be able to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory.

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.
- 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.
- Traverse the file system. We may wish to access every directory and every file within a directory structure.

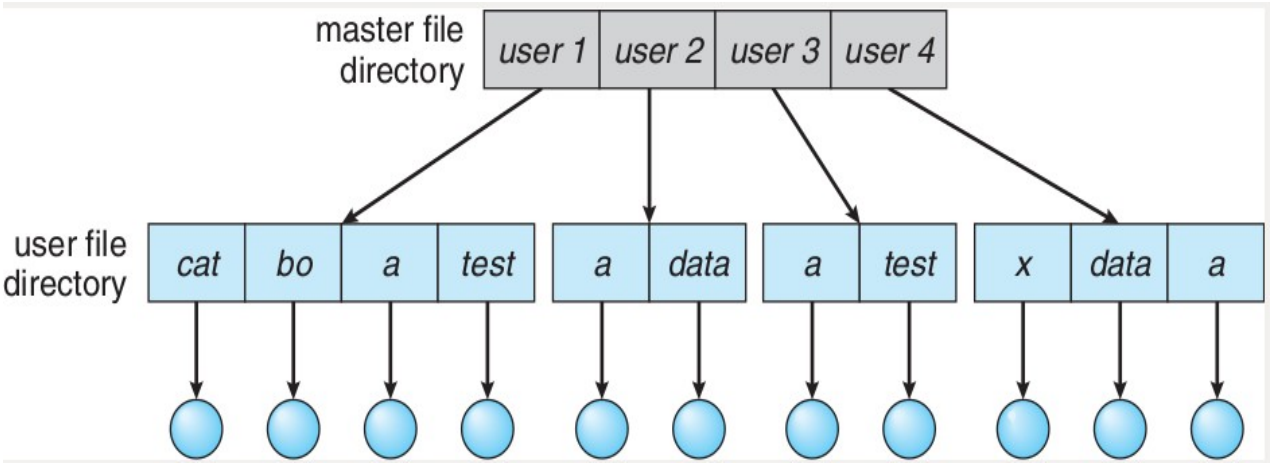
Single-level Directory

- The simplest directory structure is the single-level directory. All files are contained in the same directory.
- 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.



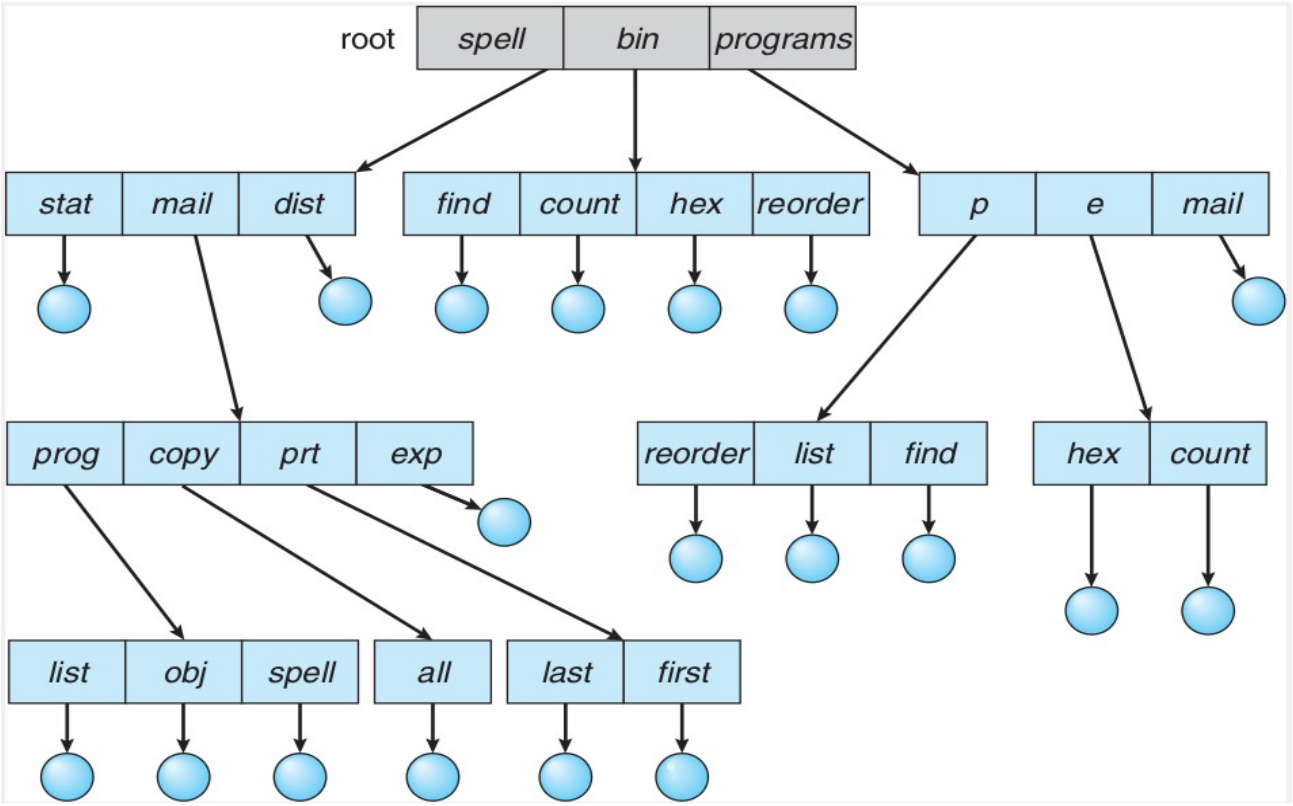
Two-Level Directory

- 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.
- Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. To name a particular file uniquely in a two-level directory, we must give both the user name and the file name.



- For example, if user A wishes to access her own test file named test, she can simply refer to test. To access the file named test of user B (with directory-entry name userb), however, she might have to refer to /userb/test.
- Many command interpreters simply treat commands as the name of a file to load and execute. With the directory system defined presently, this file name would be searched for in the current UFD.
- One solution would be to copy the system files into each UFD. However, copying all the system files would waste an enormous amount of space.
- A special user directory is defined to contain the system files. Whenever a file name is given to be loaded, the operating system first searches the local UFD.
- If the file is found, it is used. If it is not found, the system automatically searches the special user directory that contains the system files.

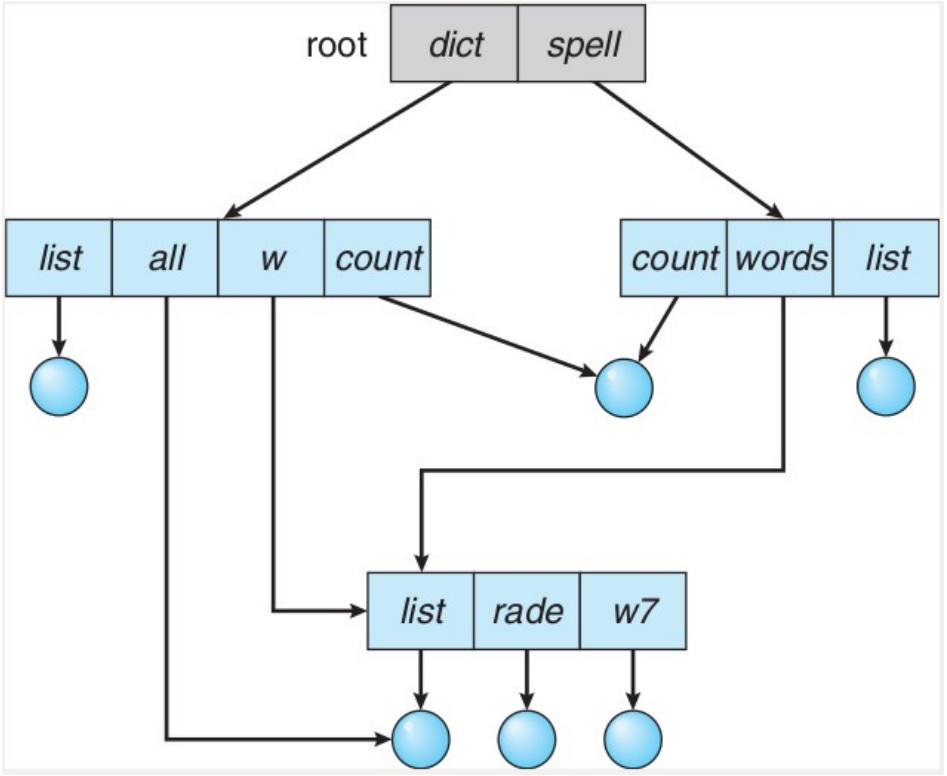
Tree-Structured Directories



- 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).
- In normal use, each process has a current directory. The current directory should contain most of the files that are of current interest to the process.
- When reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file.
- Path names can be of two types: absolute and relative. An absolute path name begins at the root and follows a path down to the specified file, giving directory names on the path.
- A relative path name defines a path from the current directory. For example,in the above Drawing, if the current directory is root/spell/mail, then the relative path name prt/first refers to the same file as does the absolute path name root/spell/mail/prt/first.
- An interesting policy decision in a tree-structured directory concerns how to handle the deletion of a directory. If a directory is empty, its entry in the directory that contains it can simply be deleted.
- However, suppose the directory to be deleted is not empty but contains several files or subdirectories. The approach taken by the UNIX rm command, is to provide an option: when a request is made to delete a directory, directory's files and subdirectories are also to be deleted.

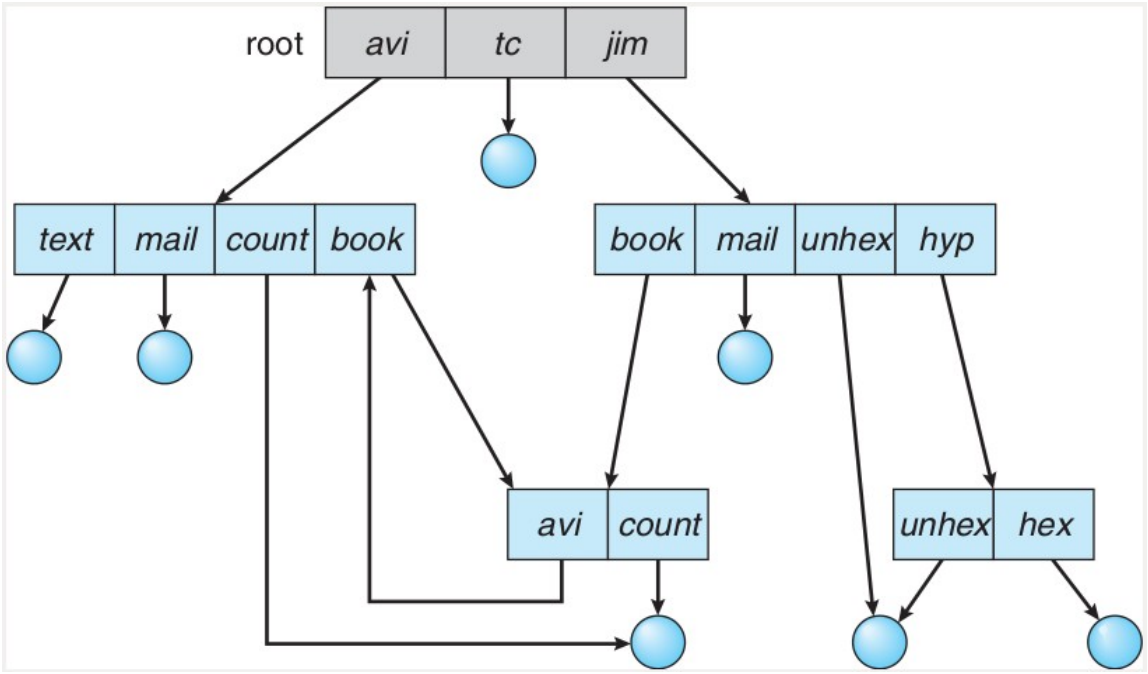
Acyclic-Graph Directories

- A tree structure prohibits the sharing of files or directories. An acyclic graph that is, a graph with no cycles-allows directories to share subdirectories and files. The same file or subdirectory may be in two different directories.
- With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other. A new file created by one person will automatically appear in all the shared subdirectories.
- Shared files and subdirectories can be implemented in several ways. A link is effectively a pointer to another file or subdirectory.
- For example, a link may be implemented as an absolute or a relative path name. When a reference to a file is made, we search the directory. If the directory entry is marked as a link, we resolve the link by using that path name to locate the real file.
- In a system where sharing is implemented by symbolic links, the deletion of a link need not affect the original file; only the link is removed.
- If the file entry itself is deleted, the space for the file is deallocated, leaving the links dangling. We can leave the links until an attempt is made to use them. At that time, we can determine that the file with name given by the link does not exist and delete the link.
- Another approach to deletion is to preserve the file until all references to it are deleted. We could keep a list of references to a file.
- When a link is established, a new entry is added to the file-reference list. When a link, we remove its entry on the list. The file is deleted when its file-reference list is empty.



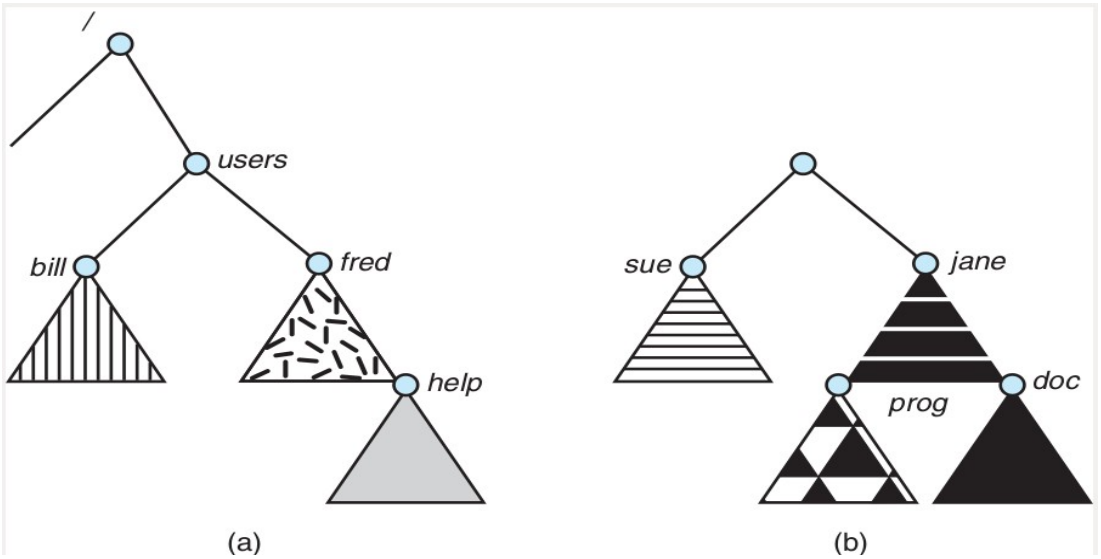
General Graph Directory

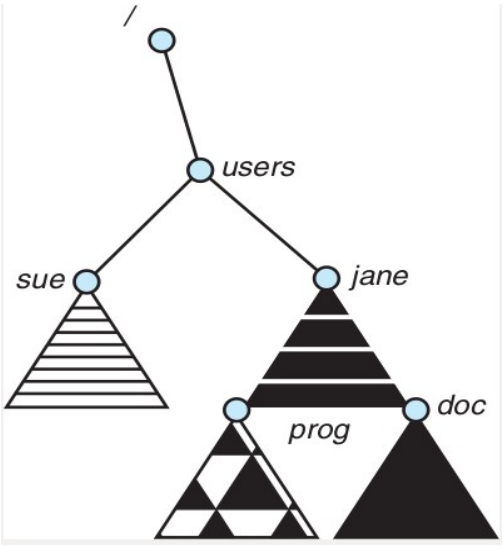
- A serious problem with using an acyclic-graph structure is ensuring that there are no cycles. The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse the graph and to determine when there are no more references to a file.
- With acyclic-graph directory structures, a value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted.
- However, when cycles exist, the reference count may not be 0 even when it is no longer possible to refer to a directory or file. This anomaly results from the possibility of self-referencing.



File System Mounting

- Just as a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system.
- 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.
- 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.
- Next, the operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format.
- Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point.
- To illustrate file mounting, consider the file system depicted in Drawing below, where the triangles represent subtrees. Drawing 'a' shows an existing file system, while Drawing 'b' shows an unmounted volume residing on /device/dsk.
- At this point, only the files on the existing file system can be accessed. Drawing 'c' shows the effects of mounting the volume residing on /device/dsk over /users.





(c)

File Sharing

Multiple Users

- When an operating system accommodates multiple users, the issues of file sharing, file naming, and file protection become preeminent.
- To implement sharing and protection, the system must maintain more file and directory attributes than are needed on a single-user system.
- 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 the most control over the file.
- Exactly which operations can be executed by group members and other users is defined by the file's owner. The owner and group IDs of a given file (or directory) are stored with the other file attributes.
- When a user requests an operation on a file, the user ID can be compared with the owner attribute to determine if the requesting user is the owner of the file. Likewise, the group IDs can be compared.
- The result indicates which permissions are applicable. The system then applies those permissions to the requested operation and allows or denies it.

Remote File Systems

- 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.

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.

- A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client-server facility.
- Once the remote file system is mounted, file operation requests are sent on behalf of the user across the network to the server via the DFS protocol.
- Typically, a file-open request is sent along with the ID of the requesting user. The server then applies the standard access checks to determine if the user has credentials to access the file in the mode requested.
- The request is either allowed or denied. If it is allowed, a file handle is returned to the client application, and the application then can perform read, write, and other operations on the file.

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.
- The industry is moving toward use of the lightweight directory access protocol(LDAP) as a secure distributed naming mechanism.
- Sun Microsystems includes LDAP with the operating system and allows it to be employed for user authentication as well as system-wide retrieval of information, such as availability of printers.
- Conceivably, one distributed LDAP directory could be used by an organization to store all user and resource information for all the organization's computers.
- The result would be secure single sign in for users, who would enter their authentication information once for access to all computers within the organization.

Failure Modes

- Local file systems can fail for a variety of reasons, including failure of the disk containing the file system, corruption of the directory structure, disk-controller failure, cable failure.
- Remote file systems have even more failure modes. Because of the complexity of network systems and the required interactions between remote machines, many more problems can interfere with the proper operation of remote file systems.
- In the case of networks, the network can be interrupted between two hosts. Such interruptions can result from hardware failure, poor hardware configuration, or networking implementation issues.
- Consider a client in the midst of using a remote file system. It has files open from the remote host; among other activities, it may be performing directory lookups to open files, reading or writing data to files, and closing files.
- Now consider a partitioning of the network, a crash of the server, or even a scheduled shutdown of the server. Suddenly, the remote file system is no longer reachable.
- System can either terminate all operations to the lost server or delay operations until the server is again reachable. Termination of all operations can result in users' losing data-and patience.
- Thus, most DFS protocols either enforce or allow delaying of file-system operations to remote hosts, with the hope that the remote host will become available again.

Consistency Semantics

- Consistency Semantics represent an important criterion for evaluating any file system that supports file sharing. These semantics specify how multiple users of a system are to access a shared file simultaneously.

UNIX Semantics

The UNIX file system uses the following consistency semantics:

- Writes to an open file by an user are visible immediately to other users who have this file open.
- One mode of sharing allows users to share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users.
- In the UNIX semantics, a file is associated with a single physical image that is accessed as an exclusive resource. Contention for this single image causes delays in user processes.

Session Semantics

The Andrew file system (AFS) uses the following consistency semantics:

- Writes to an open file by a user are not visible immediately to other users that have the same file open.
- Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes.
- According to these semantics, a file may be associated temporarily with several images at the same time. Consequently, multiple users are allowed to perform both read and write accesses concurrently on their images of the file, without delay.

Immutable-Shared-Files Semantics

- A unique approach is that of immutable shared files. Once a file is declared as shared by its creator, it cannot be modified.
- An immutable file has two key properties: its name may not be reused, and its contents may not be altered. Thus, the name of an immutable file signifies that the contents of the file are fixed.

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).
- Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically copy disk files to tape at regular intervals to maintain a copy if a file system is accidentally destroyed.

Types of Access

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.

Other operations, such as renaming, copying, and editing the file, may also be controlled. For many systems, however, these higher-level functions may be implemented by a system program that uses the lower-level system calls.

Protection is provided at only the lower level. For instance, copying a file may be implemented simply by a sequence of read requests. In this case, a user with read access can also cause the file to be copied, printed, and so on.

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.
- The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access.

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.

The most common recent approach is to combine access-control lists with the more general owner, group, and universe access- control scheme just described.

For example, the UNIX system defines three fields of 3 bits each -rwx, where r controls read access, w controls write access, and x controls execution. A separate field is kept for the file owner, for the file's group, and for all other users.

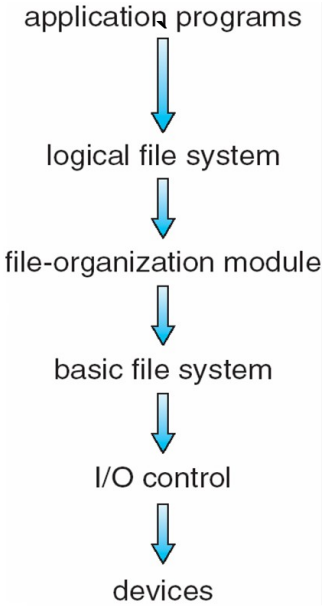
Other Protection Approaches

- Another approach to the protection problem is to associate a password with each file. Just as access to the computer system is often controlled by a password, access to each file can be controlled in the same way.
- If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file. The use of passwords has a few disadvantages, however.
- First, the number of passwords that a user needs to remember may become large, making the scheme impractical. Second, if only one password is used for all the files, then once it is discovered, all files are accessible; protection is on an all-or-none basis.
- In a multilevel directory structure, we need to protect not only individual files but also collections of files in subdirectories; that is, we need to provide a mechanism for directory protection.

File System Structure

- Disks provide the bulk of secondary storage on which a file system is maintained. They have two characteristics that make them a convenient medium for storing multiple files:
  - A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same place.
  - A disk can access directly any block of information it contains. Thus, it is simple to access any file either sequentially or randomly.

- 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 Drawing below. The lowest level, the I/O control, 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. Its input consists of high-level commands such as "retrieve block 123." Its output consists of low- level, hardware-specific instructions that are used by the hardware controller.
- The basic file system needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.
- The file-organization module knows about files and their logical blocks, as well as physical blocks. 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.



- The file-organization module also includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.
- Finally, the logical file system manages metadata information. Metadata includes all of the file-system structure except the actual data.
- The logical file system maintains file structure via file-control blocks. A file-control block (an inode in most UNIX file systems) contains information about the file, including ownership, permissions, and location of the file contents.

File System Implementation

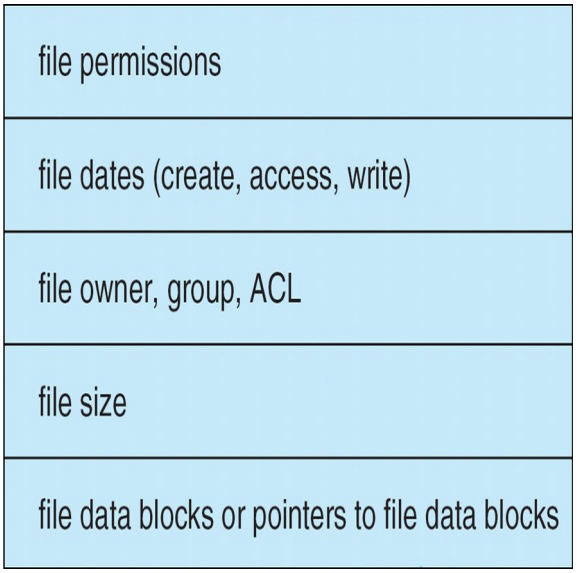
Overview

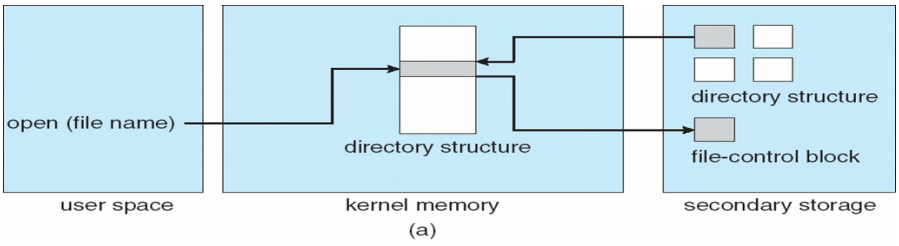
- Several on-disk and in-memory structures are used to implement a file system.
  - On disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.
  - A boot control block (per volume) can contain information needed by the system to boot an operating system from that volume. If the disk does not contain an operating system, this

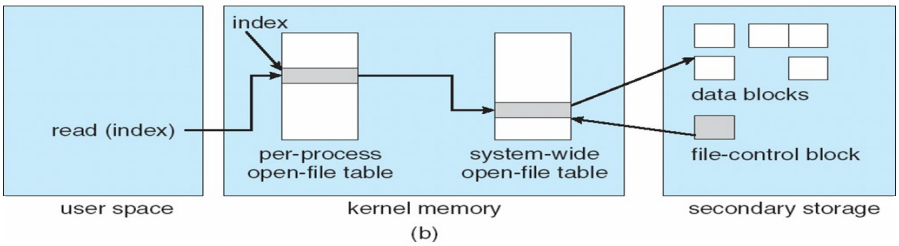
block can be empty. It is typically the first block of a volume.

- A volume control block (per volume) contains volume (or partition) details, such as the number of blocks in the partition, the size of the blocks, a free-block count and free-block pointers.
- A directory structure (per file system) is used to organize the files. In UFS, this includes file names and associated inode numbers.
- A per-file FCB contains many details about the file. It has a unique identifier number to allow association with a directory entry.

- The in-memory information is used for both file-system management and performance improvement via caching. The data are loaded at mount time, updated during file-system operations, and discarded at dismount. Several types of structures may be included.
  - An in-memory mount table contains information about each mounted volume.
  - An in-memory directory-structure cache holds the directory information of recently accessed directories.
  - The system-wide open-file table contains a copy of the FCB of each open file, as well as other information.
  - The per-process open-file table contains a pointer to the appropriate entry in the system-wide open-file table, as well as other information.
  - Buffers hold file-system blocks when they are being read from disk or written to disk.







- 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 a new file, it allocates a new FCB.
- 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 Drawing above.

- 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.
- If the file is not already open, the directory structure is searched for the given file name. Once the file is found, the FCB is copied into a system-wide open-file table in memory.
- 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.
- The open() call returns a pointer to the appropriate entry in the per-process file-system table. All file operations are then performed via this pointer.
- 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.

Partitions and Mounting

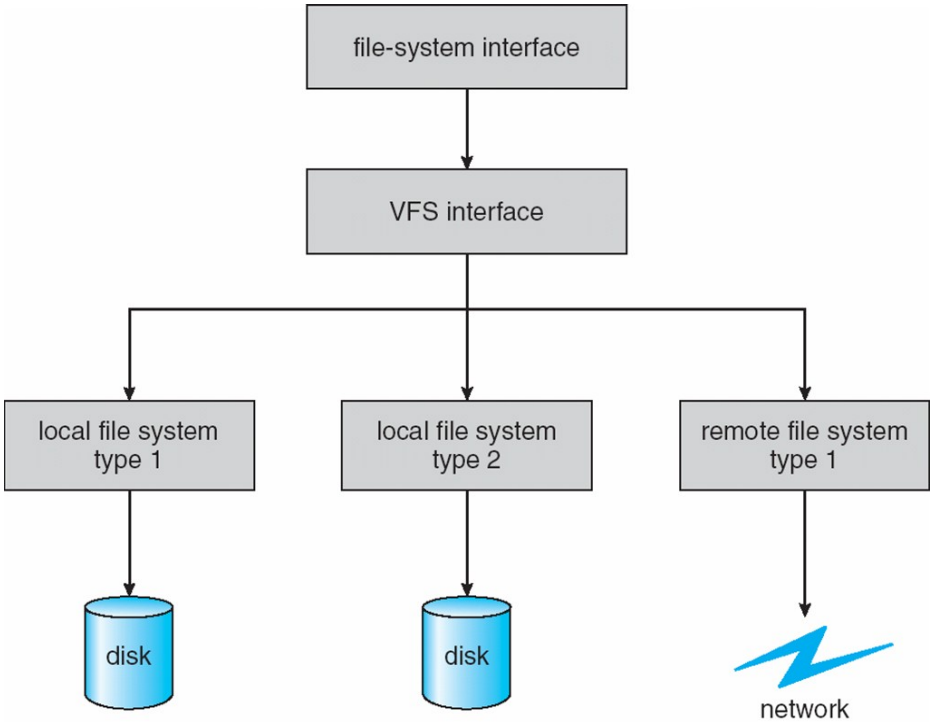
- A disk can be sliced into multiple partitions, or a volume can span multiple partitions on multiple disks. Each partition can be either "raw," containing no file system, or "cooked," containing a file system.
- Boot information can be stored in a separate partition. Again, it has its own format, because at boot time the system does not have the file-system code loaded and therefore cannot interpret the file-system format.
- Rather, boot information is usually a sequential series of blocks, loaded as an image into memory. Execution of the image starts at a predefined location, such as the first byte.
- This boot loader in turn knows enough about the file-system structure to be able to find and load the kernel and start it executing.
- The root partition which contains the operating-system kernel and some- times other system files, is mounted at boot time.
- Other volumes can be automatically mounted at boot or manually mounted later, depending on the operating system.
- As part of a successful mount operation, the operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format.

Virtual File Systems

- An obvious but suboptimal method of implementing multiple types of file systems is to write directory and file routines for each type.
- Instead, however, most operating systems, including UNIX, use object-oriented techniques to simplify, organize, and modularize the implementation.
- The use of these methods allows very dissimilar file-system types to be implemented within the same structure, including network file systems, such as NFS.
- The file-system implementation consists of three major layers, as depicted schematically in

Drawing below. The first layer is the file-system interface, based on the open(), read(), write(), and close() calls and on file descriptors.

- The second layer is called the virtual file system(VFS) layer. VFS layer serves two important functions:
  - It separates file-system-generic operations from their implementation by defining a clean VFS interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally.
  - It provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a vnode that contains a numerical designator for a network-wide unique file.
- The VFS activates file-system-specific operations to handle local requests according to their file-system types and calls the NFS protocol procedures for remote requests.
- The layer implementing the file-system type or the remote-file-system protocol is the third layer of the architecture.



Directory Implementation

Linear List

- The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks.
- 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.
- The real disadvantage of a linear list of directory entries is that finding a file requires a linear search. Directory information is used frequently, and users will notice if access to it is slow.
- A sorted list allows a binary search and decreases the average search time. However, the requirement that the list be kept sorted may complicate creating and deleting files.

Hash Table

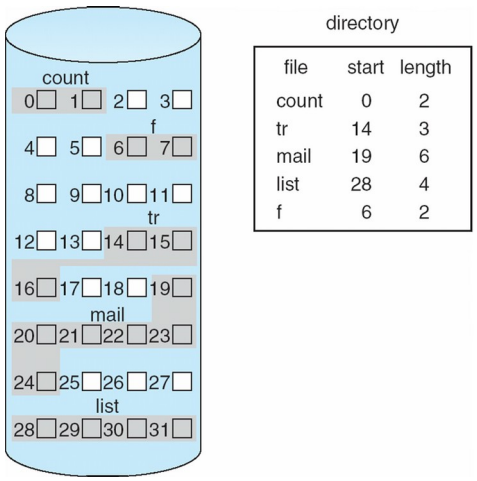
- Another data structure used for a file directory is a hash table. With this method, a linear list stores the directory entries, but a hash data structure is also used.
- The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. Therefore, it can greatly decrease the directory search time.
- The major difficulties with a hash table are its fixed size and the dependence of the hash function on that size.
- For example, assume that we make a hash table that holds 64 entries. The hash function converts file names into integers from 0 to 63, for instance, by using the remainder of a division by 64.
- If we later try to create a 65th file, we must enlarge the hash table-say, to 128 entries. As a result, we need a new hash function that must map file names to the range 0 to 127, and we must reorganize the existing directory entries to reflect their new hash-function values.

Allocation Methods

- The direct-access nature of disks allows us flexibility in the implementation of files. 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 Allocation

- Contiguous Allocation requires that each file occupy a set of contiguous blocks on the disk. Contiguous allocation of a file is defined by the disk address and length of the first block.
- If the file is n blocks long and starts at location b, then it occupies blocks b, b + 1, b + 2, ... , b + n - 1. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.
- Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block.
- For direct access to block i of a file that starts at block b, we can immediately access block b + i. Thus, both sequential and direct access can be supported by contiguous allocation.
- Contiguous allocation has some problems, however. One difficulty is finding space for a new file. Which involves how to satisfy a request of size n from a list of free holes.
- First fit and best fit are the most common strategies used to select a free hole from the set of available holes. All these algorithms suffer from the problem of external fragmentation.
- One strategy for preventing loss of significant amounts of disk space to external fragmentation is to copy an entire file system onto another disk or tape. The original disk is then freed completely, creating one large contiguous free space. We then copy the files back onto the original disk by allocating contiguous space from this one large hole. This scheme effectively compacts all free space into one contiguous space, solving the fragmentation problem. However, the cost of this compaction is time and it can be particularly severe for large hard

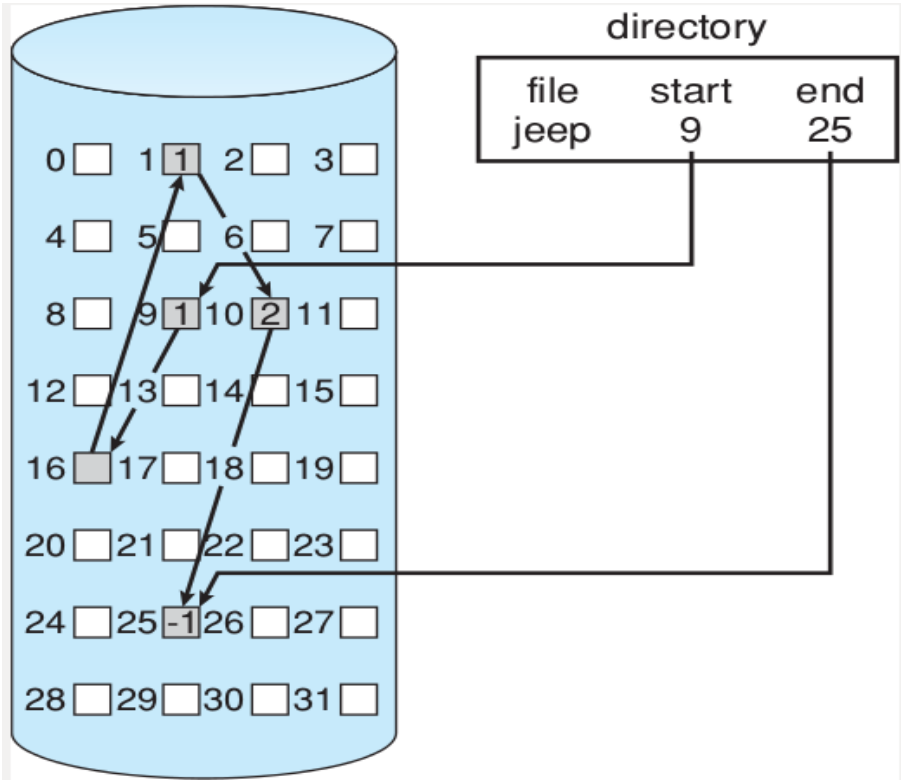


disks that use contiguous allocation.

- Another problem with contiguous allocation is determining how much space is needed for a file. If we allocate too little space to a file, we may find that the file cannot be extended.
- Especially with a best-fit allocation strategy, the space on both sides of the file may be in use. Hence, we cannot make the file larger in place.
- A file that will grow slowly over a long period (months or years) must be allocated enough space for its final size, even though much of that space will be unused for a long time. The file therefore has a large amount of internal fragmentation.

Linked Allocation

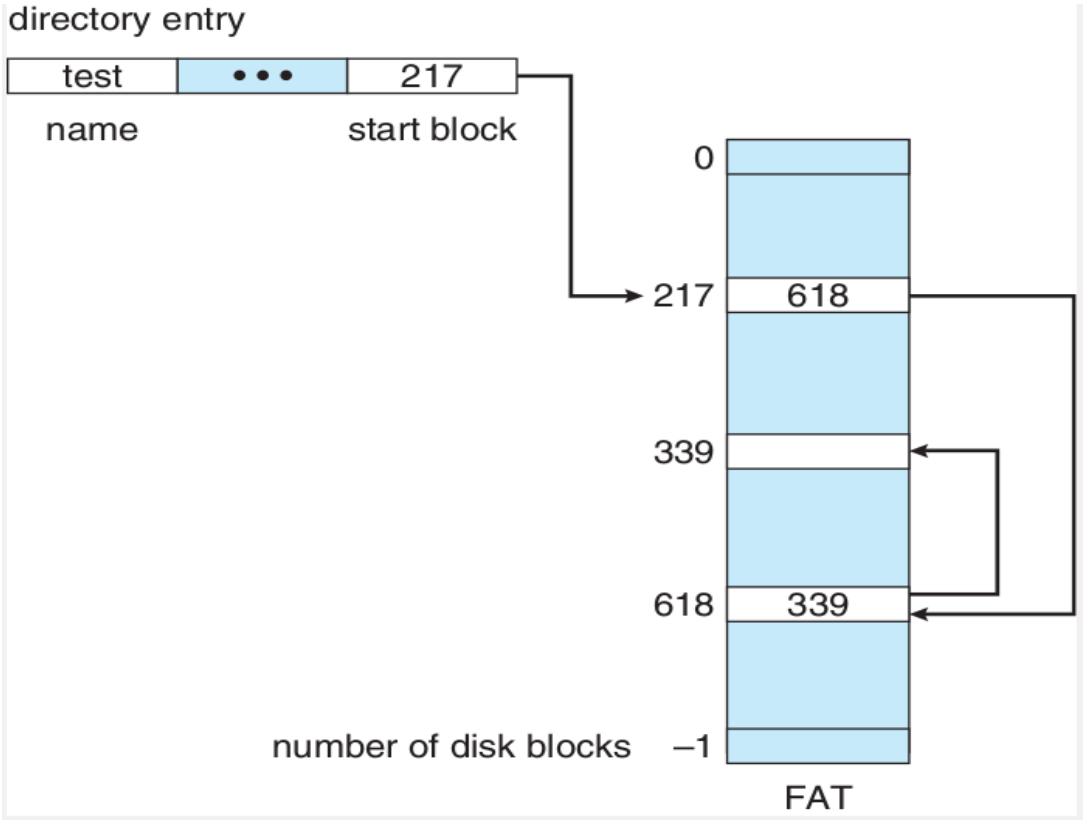
- Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk.
- The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25 (Drawing below).
- Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes in size, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.
- To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialized to nil to signify an empty file. The size field is also set to 0.
- A write to the file causes the free-space management system to find a free block, and this new block is written to and is linked to the end of the file.
- To read a file, we simply read blocks by following the pointers from block to block. There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. A file can continue to grow as long as free blocks are available.



- Linked allocation does have disadvantages, however. The major problem is that it can be used effectively only for sequential-access files.
- Another disadvantage is the space required for the pointers. If a pointer requires 4 bytes out of a

512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information.

- The usual solution to this problem is to collect blocks into multiples, called clusters and to allocate clusters rather than blocks. Pointers then use a much smaller percentage of the file's disk space.
- The cost of this approach is an increase in internal fragmentation, because more space is wasted when a cluster is partially full than when a block is partially full.
- An important variation on linked allocation is the use of a file allocation table(FAT). A section of disk at the beginning of each volume is set aside to contain the table. The table has one entry for each disk block and is indexed by block number.
- The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number contains the block number of the next block in the file.
- This chain continues until it reaches the last block, which has a special end-of-file value as the table entry. An unused block is indicated by a table value of 0.
- Allocating a new block to a file is a simple matter of finding the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-of-file value.

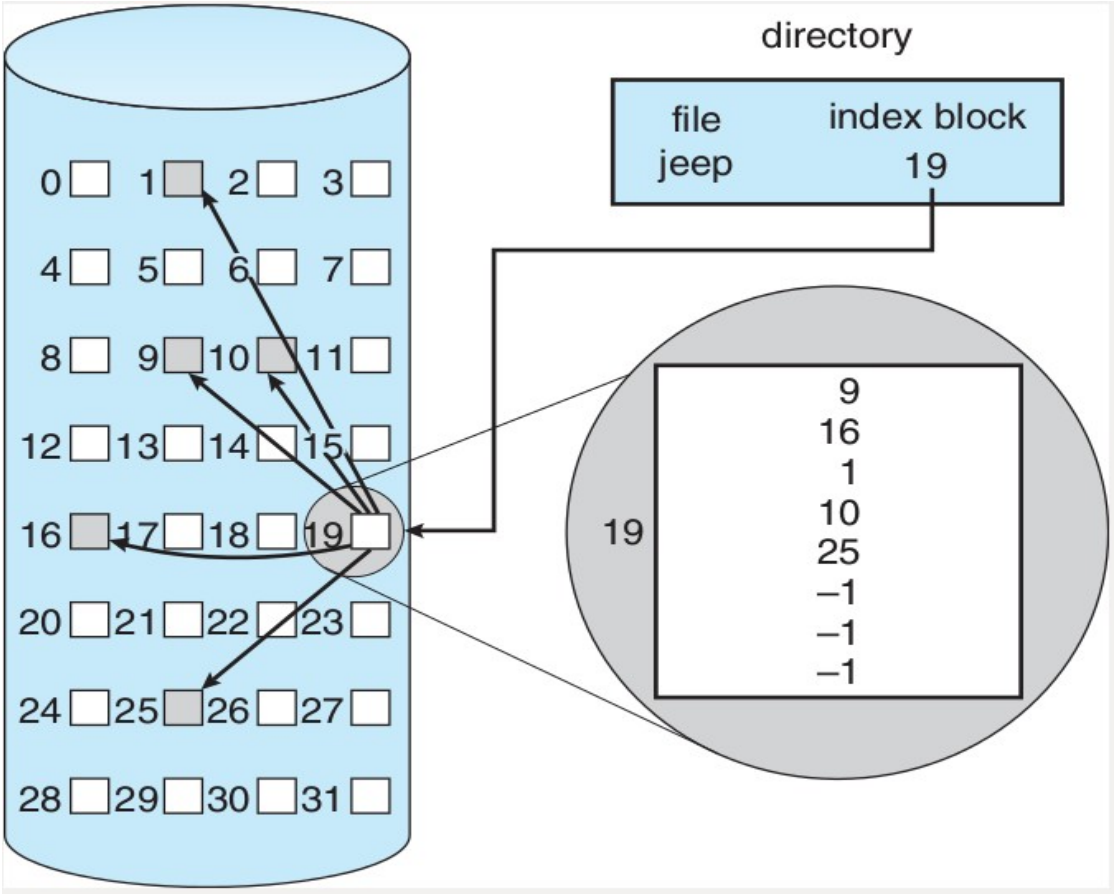


Indexed Allocation

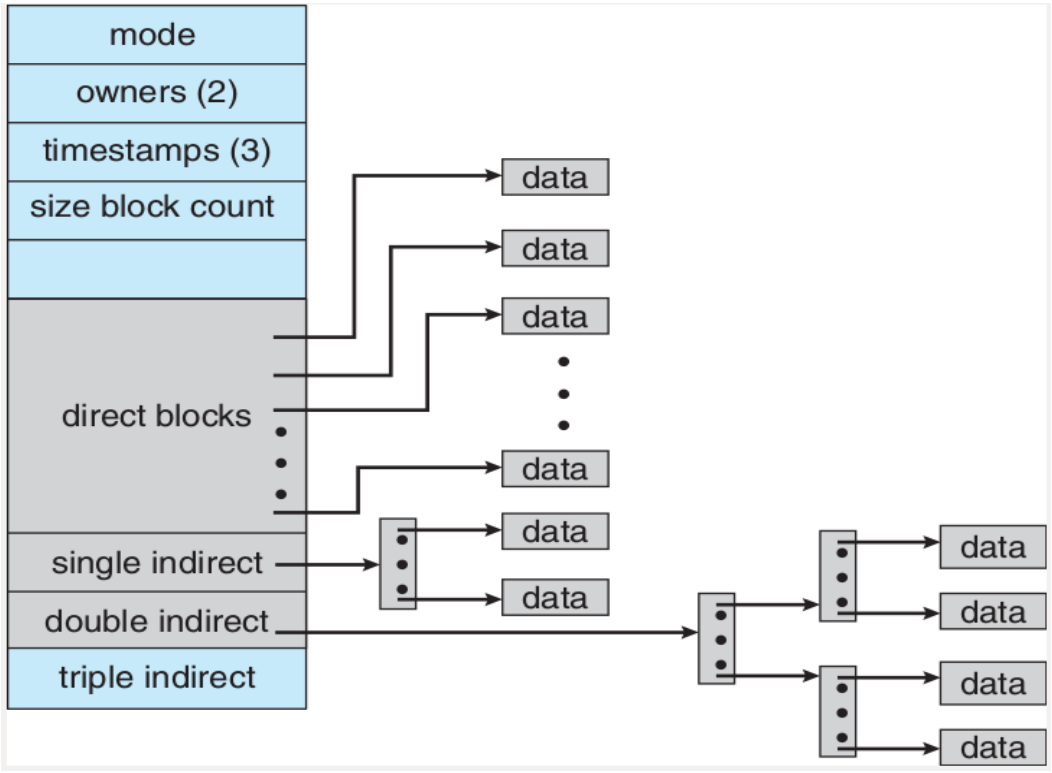
- Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, in the absence of a FAT, linked allocation cannot support efficient direct access.
- Indexed allocation solves this problem by bringing all the pointers together into one location: the index block. Each file has its own index block, which is an array of disk-block addresses. The i^{th} entry in the index block points to the i^{th} block of the file.
- The directory contains the address of the index block. To find and read the i^{th} block, we use the pointer in the i^{th} index-block entry. When the file is created, all pointers in the index block are

set to nil.

- When the i^{th} block is first written, a block is obtained from the free-space management and its address is put in the i^{th} index-block entry.
- Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space.



- Every file must have an index block, so we want the index block to be as small as possible. If the index block is too small, however, it will not be able to hold enough pointers for a large file. Mechanisms for this purpose include the following:
- Linked scheme. An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we can link together several index blocks.
- Multilevel index. A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block.
- Combined scheme. Another alternative is to keep the first, say, 15 pointers of the index block in the file's inode. The first 12 of these pointers point to direct blocks; that is, they contain addresses of blocks that contain data of the file.
  - The next three pointers point to indirect blocks. The first points to a single indirect block, which is an index block containing not data but the addresses of blocks that do contain data.
  - The second points to a double indirect block, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer contains the address of a triple indirect block.



Performance

- The allocation methods that we have discussed vary in their storage efficiency and data-block access times. Both are important criteria in selecting the proper method an operating system to implement.
- A system with mostly sequential access should not use the same method as a system with mostly random access.
- For any type of access, contiguous allocation requires only one access to get a disk block. Since we can easily keep the initial address of the file in memory, we can calculate immediately the disk address of the i^{th} block and read it directly.
- For linked allocation, we also keep the address of the next block in memory. This method is fine for sequential access; for direct access, however, an access to the i^{th} block might require i disk reads.
- Some systems support direct-access files by using contiguous allocation and sequential-access files by using linked allocation.
- A file created for sequential access will be linked and cannot be used for direct access. A file created for direct access will be contiguous and can support both direct access and sequential access, but its maximum length must be declared when it is created.
- Indexed allocation is more complex. If the index block is already in memory, then the access can be made directly. However, keeping the index block in memory requires considerable space.
- Some systems combine contiguous allocation with indexed allocation by using contiguous allocation for small files (up to three or four blocks) and automatically switching to an indexed allocation if the file grows large.

Free Space Management

- Since disk space is limited, we need to reuse the space from deleted files for new files. To keep track of free disk space, the system maintains a free-space list.
- The free-space list records all free disk blocks-those not allocated to some file or directory. To

create a file, we search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from the free-space list.

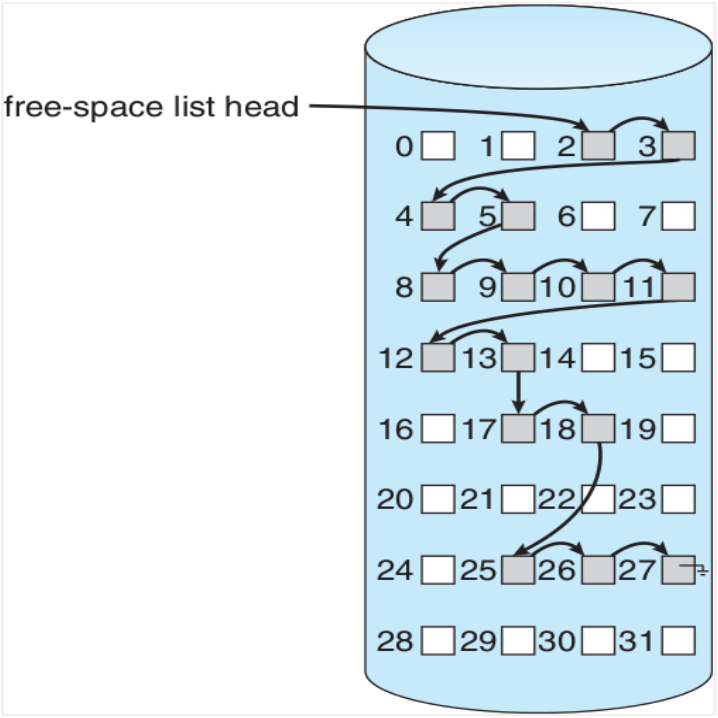
Bit Vector

- Frequently, the free-space list is implemented as a bit map or bit vector. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.
- For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bit map would be

  001111001111110001100000011100000 …
- The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or n consecutive free blocks on the disk.
- One technique for finding the first free block on a system that uses a bit-vector to allocate disk space is to sequentially check each word in the bit map to see whether that value is not 0, since a 0-valued word contains only 0 bits and represents a set of allocated blocks.
- The first non-0 word is scanned for the first 1 bit, which is the location of the first free block. The calculation of the block number is

Linked List

- Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory.
- This first block contains a pointer to the next free disk block, and so on. Recall our earlier example, in which blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 were free and the rest of the blocks were allocated.
- In this situation, we would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on.



Grouping

- A modification of the free-list approach stores the addresses of n free blocks in the first free block. The first n-1 of these blocks are actually free.
- The last block contains the addresses of another n free blocks, and so on. The addresses of a

large number of free blocks can now be found quickly.

Counting

- Another approach takes advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous-allocation algorithm or through clustering.
- Thus, rather than keeping a list of n free disk addresses, we can keep the address of the first free block and the number (n) of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a disk address and a count.