# File-System Interface

## 1 File Concept

A file is a named collection of related information that is recorded on secondary storage.
Files represent programs and data.

### 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 humanreadable form,.

• **Identifier**. This is 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.

The information about all files is kept in the directory structure. Typically, a directory entry consists of the file's name and its unique identifier. The identifier in turn locates the other file attributes. It may take more than a kilobyte to record this information for each. file. In a system with many files, the size of the directory itself may be megabytes.

### File Operations

A file is an **abstract data type.** 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 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.

• 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 bv 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.
This function allows all attributes to remain unchanged—except for file length—be reset to 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 operating system keeps a small table, called the **open-file table,** containing information about all open files. When a file operation is requested, the file is specified via an index into this table, so no searching is required.

When the file is no longer being actively used, it is *closed* by the process, and the operating system removes its entry from the open-file table, create and delete are system calls that work with closed rather than open files.

### Fiie 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.

| 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, pas, asm, a | source code in various languages |
| batch | bat, sh | commands to the command interpreter |
| text | txt, doc | textual data, documents |
| word processor | wp, tex, rtf, doc | various word-processor formats |
| library | lib, a, so, dll | libraries of routines for programmers |
| print or view | ps, pdf, jpg | ASCII or binary file in a format for printing or viewing |
| archive | arc, zip, tar | related files grouped into one file, sometimes compressed, for archiving or storage |
| multimedia | mpeg, mov, rm, mp3, avi | binary file containing audio or A/V information |

**Internal File Structure**

Disk systems have a well-defined block size determined by the size of a sector. All disk I/O is performed in units of one block (physical record), and all blocks are 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.

For example, the UNIX operating system defines all files to be simply streams of bytes. In this case, the logical record size is 1 byte. The file system automatically packs and unpacks bytes into physical disk blocks—say, 512 bytes per block—as necessary.

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.

Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted. If each block were 512 bytes, for example, then a file of 1,949 bytes would be allocated four blocks (2,048 bytes); the last 99 bytes would be wasted. The waste incurred is internal fragmentation. All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

**2. Access Methods**

Files store information. The information in the file can be accessed in several ways.

**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 used editors and compilers.

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. Such a file can be reset to the beginning

**Direct Access**

Another method is **direct access** (or **relative** access). A file is made up of fixed length **logical records. It** allow programs to read and write records rapidly in no particular order.

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. A*nd to add an operation *position file to n,* where *n* is the block number.

The block number provided by the user to the operating system is normally a **relative block number.** A relative block number is an index relative to the beginning of the file.

Not all operating systems support both sequential and direct access for files. Some systems allow only sequential file access; others allow only direct access. Some systems require that a file be defined as sequential or direct when it is created; such a file can be accessed only in a manner consistent with its declaration.

**Other Access(Indexed) 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. The index, contains pointers to the various blocks. 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, IBM's indexed sequential-access method (ISAM) uses a small master index that points to disk blocks of a secondary index. The secondary index blocks point to the actual file blocks.

**3.Directory Structure**

The file systems store millions of files. To manage all these files, we need directories to organize them. In this section, we explore the topic of directory structure. First, we explain some basic features of storage structure.

**Storage Structure**

Each volume contains information about the files in the system. This information is kept in entries in a **device directory.** The device directory records information—such as name, location, size, and type—for all files on that volume.

**Directory Overview**

The directory can be viewed as a symbol table of 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.

• 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.

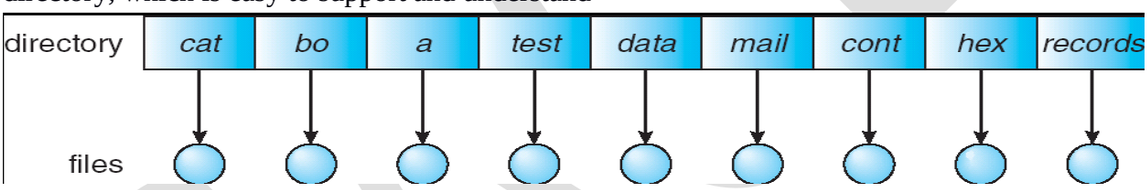
• Rename a file. we must be able to change the name when the contents or use of the file changes.

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

In the following sections, we describe the most common schemes for defining the 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



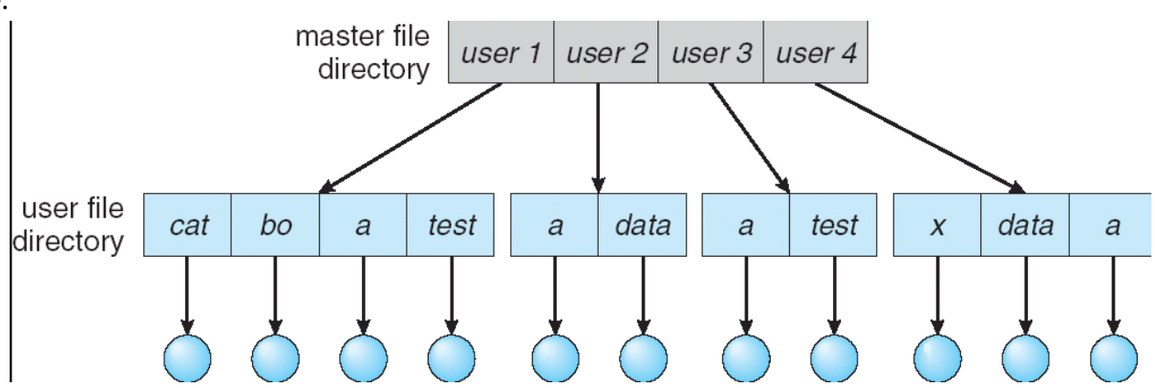

A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names.

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.

**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.

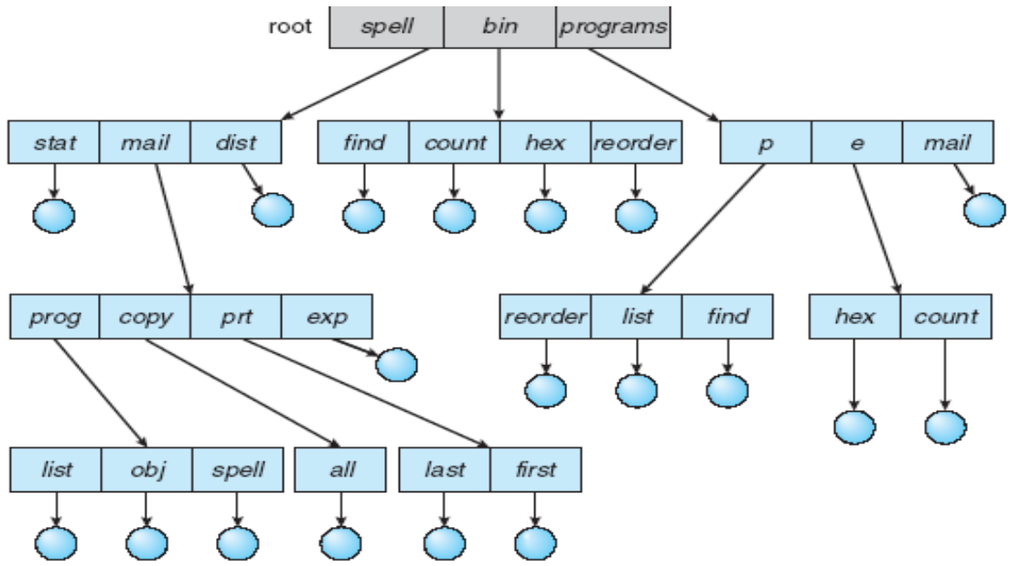
In the two-level directory structure, each user has his own user file directory (UFD). The UFDs have similar structures, but each lists only the files of a single user. When a user job starts or a user logs in, the system's master file directory (MFD) is searched. The MFD is indexed by user name or account 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.



Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. Isolation do not allow local user files to be accessed by other users.

**Tree-Structured Directories**

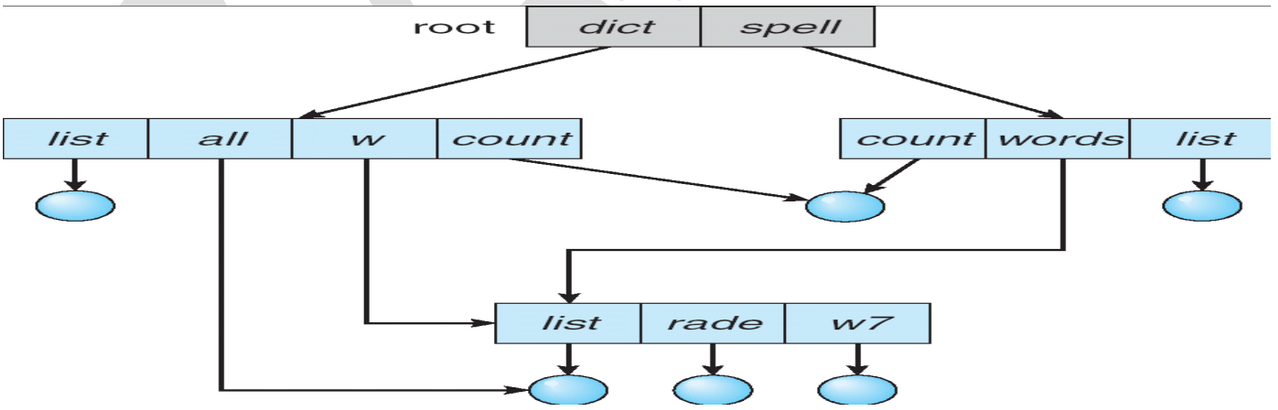
It is generalization of two-level directory, to extend the directory structure to a tree of arbitrary height. This generalization allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. A directory (or subdirectory) contains 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.




With a tree-structured directory system, users can be allowed to access, the files of other users by specifying its path names.

**Acyclic-Graph Directories**

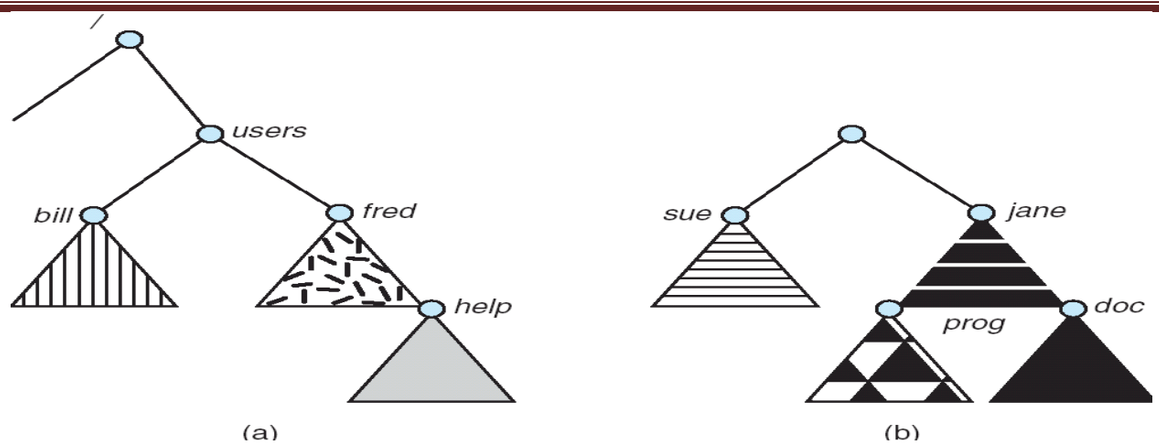
Consider two programmers who are working on a joint project. Both want the subdirectory to be *shared. A* shared directory *or* file will exist in the file system in two (or more) places at once. A tree structure prohibits the sharing of files or directories. An **acyclic graph** —that is, a graph with no cycles—allows directories to share subdirectories and files. The *same* file or subdirectory may be in two different directories.




The acyclic graph is a natural generalization of the tree-structured directory scheme. With a shared file, only *one* actual file exists, so any changes made by one person are immediately visible to the other. Sharing is particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.

**4.File-System Mounting**

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. Next, the operating system verifies that the device contains a valid file system. 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 Figure. Here the triangles represent subtrees of directories that are of interest. Figure (a) shows an existing file system, while Figure (b) shows an unmounted volume residing on */device'/dsk.* At this point, only the files on the existing file system can be accessed.

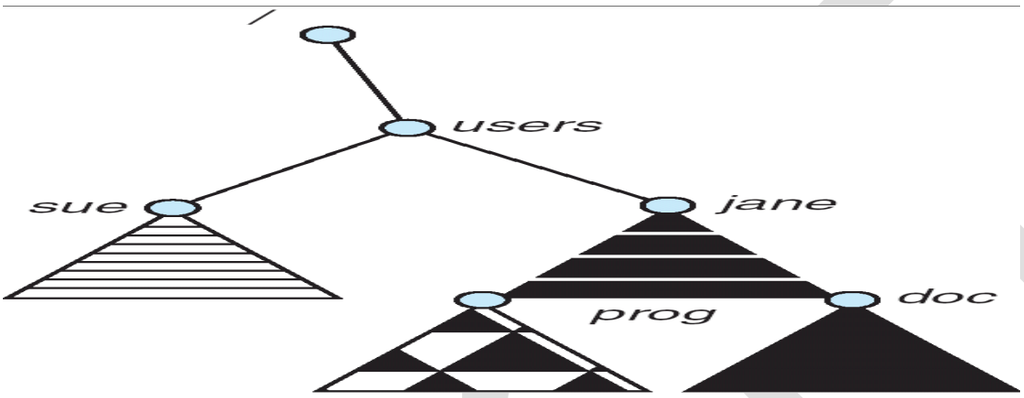



Figure shows the effects of mounting the volume residing on */device/disk* over */users.* If the volume is unmounted, the file system is restored to the previous situation depicted in Figure.

A system may allow the same file system to be mounted repeatedly, at different mount points; or it may only allow one mount per file system.

**5. File Sharing**

We begin by discussing general issues that arise when multiple users share files.

**Multiple Users**

To implement sharing among multiple users, the system must maintain more file and directory attributes than are needed on a single-user system. Most systems have use the concepts of file *owner* (or *user)* and *group.* The owner is the user who can change attributes and grant access over the file. The group attribute defines a subset of users who can share access to the file.

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**

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.

The server usually specifies the available files on a volume or directory level. Client identification is by using secure authentication of the client via encrypted keys.

**Consistency Semantics**

**Consistency semantics** represent an important criterion for file sharing. These semantics specify how multiple users of a system are to access a shared file simultaneously. These semantics are typically implemented as code with the file system.

**UNIX Semantics**

The UNIX file system uses the following consistency semantics:

• Writes to an open file by a user are visible immediately to other users that 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. Here, a file has a single image that interleaves all accesses, Regardless of their origin.

**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 (possibly different) 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. Almost no constraints are enforced on scheduling accesses.

**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. The implementation of these semantics in a distributed system is simple.

**6. Protection**

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

Reliability is generally provided by duplicate copies of files. Protection can be provided in many ways in a multiuser system.

**Types of Access**

Protection provide by the types of file access that can be made. Access is permitted or denied depending on 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 tree its space for possible reuse.

• List. List the name and attributes of the file.

**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 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 length. If we want to allow everyone to read a file, we must list all users with read access. This problems can be resolved by use of a condensed version of the access list.

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 (and easier to implement) owner, group, and universe access control scheme just described.

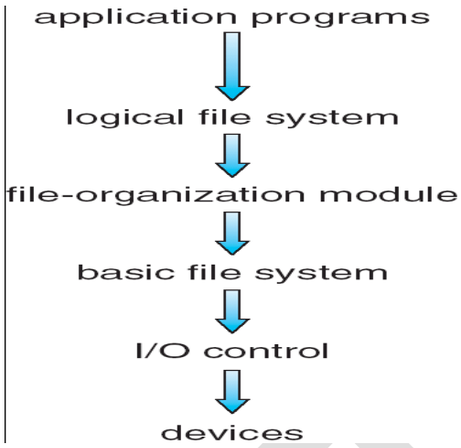
**Other Protection Approaches**

Another approach to the protection problem is to associate a password with each file. 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. Some systems allow a user to associate a password with a subdirectory, rather than with an individual file, to deal with this problem. The IBMVM/CMS operating system allows three passwords for a minidisk—one each for read, write, and multi write access.

# FILE SYSTEM IMPLEMENTATION

## 1. File-System Structure

File are stored on disk. To provide efficient access to the disk, the operating system imposes file systems to allow the data to be stored, and retrieved easily. The file system is generally composed of many different levels. The structure shown in Figure. Each level in the design uses the features of lower levels to create new features for use by higher levels.




**Layered file system.**

The lowest level, the *I/O control*, consists of **device drivers** 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 lowlevel, 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. Each physical block is identified by its numeric disk address (for example, drive 1, cylinder 73, track 2, and sector 10).

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.

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). It maintains file structure via file-control blocks. A **file-control block** (FCB) contains information about the file, including ownership, permissions,
and location of the file contents.

## 2. File-System implementation

In this section, we discuss the structures and operations used to implement file system operations.
Several on-disk and in-memory structures are used to implement a file system.

On disk, structures are

• **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 called as boot block;

• A **volume control block** (per volume) contains volume (or partition) details, such as the number of blocks in the partition, size of the blocks, free block count and free-block pointers, and free FCB count and FCB pointers. this is called as **superblock.**

• A **directory structure** per file system is used to organize the files. This includes file names and associated **inode** numbers.

• A **per-file FCB** contains many details about the file, including file permissions, ownership, size, and location of the data blocks.

The in-memory structures may include the ones described below:

• 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.

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.

**3. Directory implementation**

The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system. In this section, we discuss various algorithms.

**3.1 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, 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.

**3.2 Hash Table**

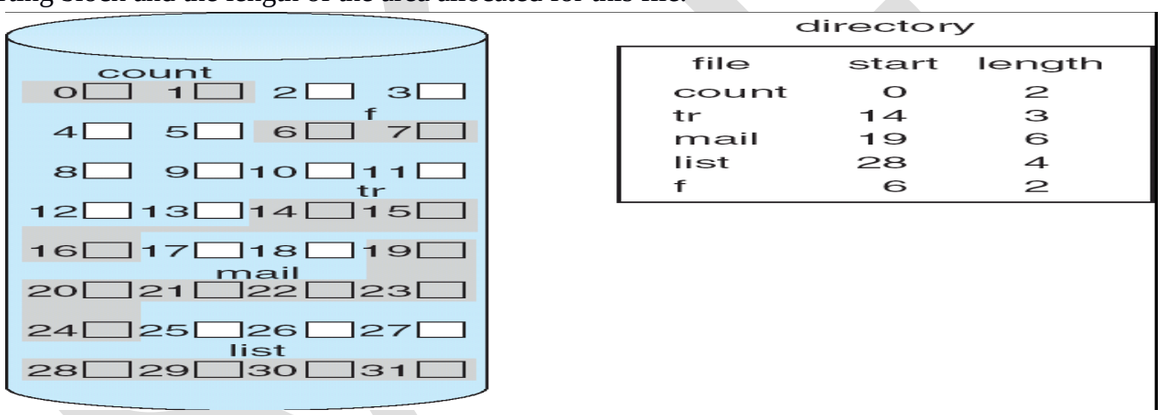
Another data structure used for a directory is a **hash table.** With this method, a hash data structure is used. The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. Therefore, it can greatly decrease the directory search time. Insertion 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. a chained-overflow hash table can be used. Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list.

**4. Allocation Methods**

In this section we mainly discuss about how to allocate space to the files. Three major methods of allocating disk space are in use: contiguous, linked, and indexed.

**4.1 Contiguous Allocation**

Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. With this ordering, accessing block *b* + 1 after block *b* normally requires no head movement. When head movement is needed, the head need only move from one track to the next. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal. The IBM VM/CMS operating system uses contiguous allocation. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.




**Contagious Allocation of Disk Space**

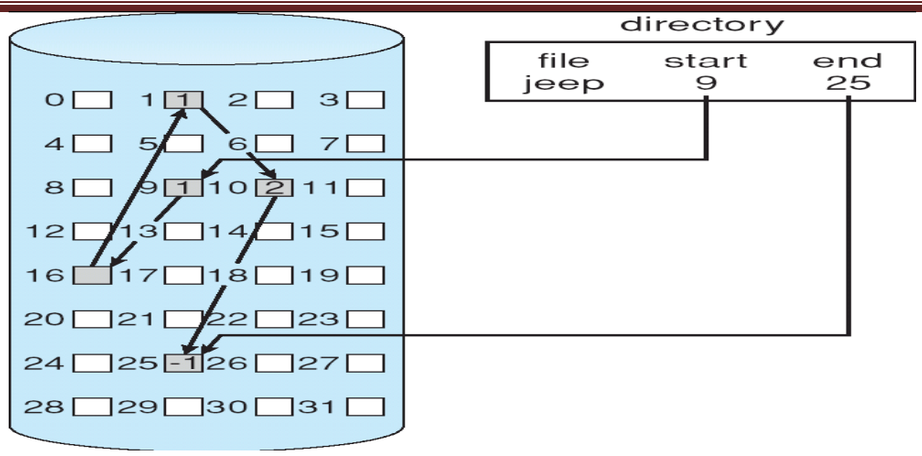
Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, reads the next block. For direct access to block i of a file that starts at block *b*, we can immediately access block *b + i.* Thus, both sequential and direct access can be supported by contiguous allocation.

Contiguous allocation has some problems. One difficulty is finding space for a new file. Second suffer from the problem of **external fragmentation.** Another problem is determining how much space is needed for a file, When the file is created.

To minimize these drawbacks, some operating systems use a modified contiguous-allocation scheme. Here, a contiguous chunk of space is allocated initially; and then, if that amount proves not to be large enough, another chunk of contiguous space, known as an **extent,** is added. The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent.

**4.2 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. There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. The size of a file need not be declared when that file is created.




**Linked Allocation**

The major problem is that it can be used effectively only for sequential-access files. To find the $i^{th}$ block of a file, we must start at the beginning of that file and follow the pointers until we get to the $i^{th}$ block .

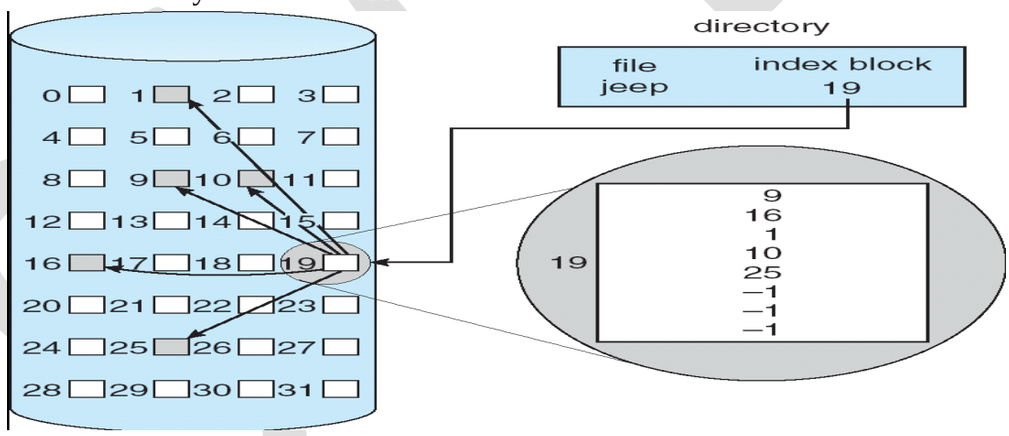
Another disadvantage is the space required for the pointers. If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information.

The usual solution to this problem is to collect blocks into multiples, called clusters, and to allocate clusters rather than blocks. This approach is increases internal fragmentation, because more space is wasted when a cluster is partially full than when a block is partially full.

Another problem of linked allocation is reliability, because files are linked together by pointers scattered all over the disk, and pointers are lost or damaged.

## 4.3 Indexed Allocation

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order. **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.




**Indexed Allocation**

Indexed allocation supports direct access, without suffering from external fragmentation. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation. If the index block is too small, 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 the linked representation is to use 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.** It keeps the first, 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 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.**

**5. Free-Space Management**

Since disk space is limited, we need to reuse the space from deleted files for new Files. So, To keep track of free disk space, the system maintains a **free-space list.** The free-space list records *all free* disk blocks.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. When a file is deleted, its disk space is added to the free-space list. The free-space list, be implemented as we discussed below

**5.1 Bit Vector**

In this 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

Unfortunately, bit vectors are inefficient because the entire vector is kept in main memory, is not possible for larger disks.

**5.2 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 memory. This first block contains a pointer to the next free disk block, and so on.
However; this scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time.

**5.3 Grouping**

A modification of the free-list approach is to store 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 other n free blocks, and so on. The addresses of a large number of free blocks can now be found quickly.

**5.4 Counting**

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. Although each entry requires more space than, but the overall list will be shorter, as long as the count is generally greater than 1.

# Mass-Storage Structure

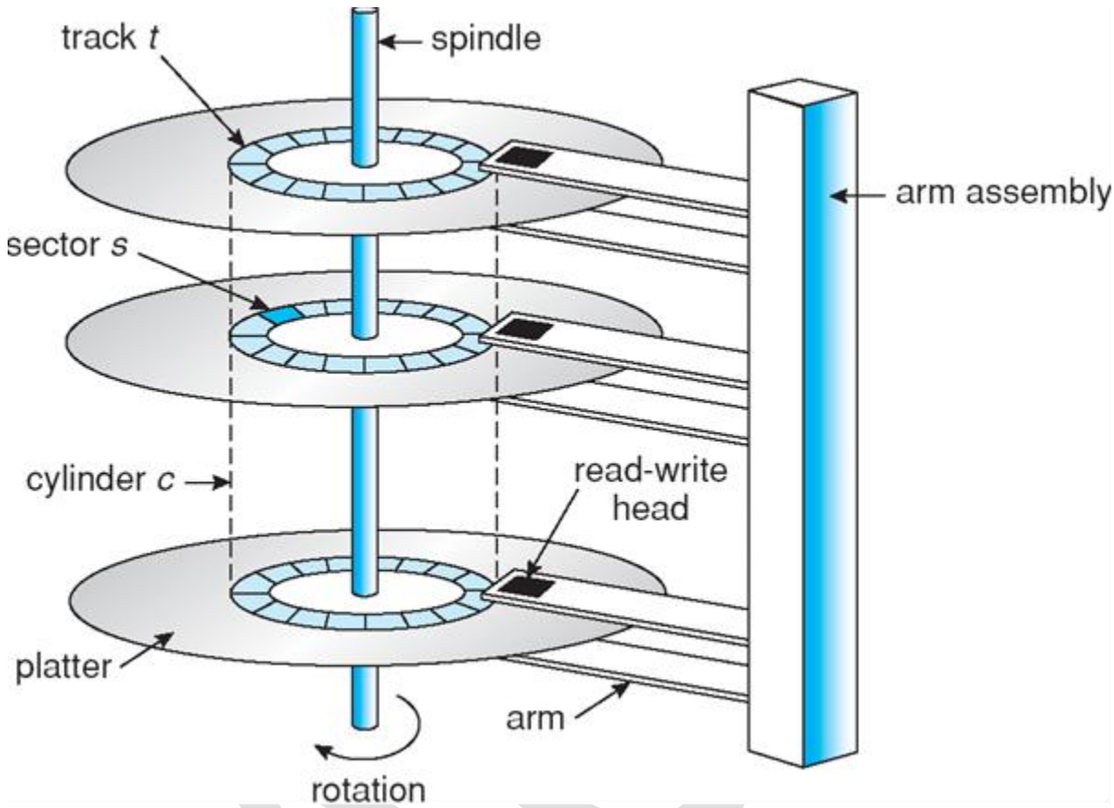
## 1. Overview of Mass-Storage Structure

In this section we present a general overview of the physical structure of secondary and tertiary storage devices.

### Magnetic Disks

Magnetic disks provide the bulk of secondary storage for modern computer systems. Each disk

platter has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 5.25 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters.




**Moving-head Disk Mechanism**

A read-write head "flies" just above each surface of every platter. The heads are attached to a **disk arm** that moves all the heads as a unit. The surface of a platter is logically divided into circular **tracks,** which are subdivided into **sectors.** The set of tracks that are at one arm position makes up a **cylinder.**

When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to 200 times per second. Disk speed has two parts. The **transfer rate** is the rate at which data flow between the drive and the computer. The **positioning time,** sometimes called the **random-access time,** consists of the time to move the disk arm to the desired cylinder, called the **seek time,** and the time for the desired sector to rotate to the disk head, called the **rotational latency.** Typical disks can transfer several megabytes of data per second, and they have seek times and rotational latencies of several milliseconds.

Because the disk head flies on an extremely thin cushion of air , there is a danger that the head will make contact with the disk surface. Although the disk platters are coated with a thin protective layer, sometimes the head will damage the magnetic surface. This accident is called a **head crash.** A head crash normally cannot be repaired; the entire disk must be replaced.

A disk can be **removable,** allowing different disks to be mounted as needed. Removable magnetic disks generally consist of one platter, held in a plastic case to prevent damage while not in the disk drive. **Floppy disks** are inexpensive removable magnetic disks that have a soft plastic case containing a flexible platter.

A disk drive is attached to a computer by a set of wires called an I/O **bus.** Several kinds of buses are available, including **enhanced integrated drive electronics (EIDE), advanced technology attachment (ATA), serial ATA** (SATA), **universal serial bus (USB), fiber channel (FC),** and **SCSI** buses.

The data transfers on a bus are carried out by special electronic processors called **controllers.** The **host controller** is the controller at the computer end of the bus. A **disk controller** is built into each disk drive. To perform a disk I/O operation, the computer places a command into the host controller. The host-

**controller then sends the command via messages to the disk controller, and the disk controller operates the disk-drive hardware to carry out the command.**

**Magnetic Tapes**

Magnetic tape was used as an early secondary-storage medium. Although it is relatively permanent and can hold large quantities of data, its access time is slow compared with that of main memory and magnetic disk. In addition, random access to magnetic tape is about a thousand times slower than random access to magnetic disk, so tapes are not very useful for secondary storage.

Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another. A tape is kept in a spool and is wound or rewound past a read-write head.

Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can write data at speeds comparable to disk drives. Tape capacities vary greatly, they store from 20 GB to 200 GB.

**2. Disk Structure**

Modern disk drives are addressed as large one-dimensional arrays of **logical blocks,** where the logical block is the smallest unit of transfer. The size of a logical block is usually 512 bytes. The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

By using this mapping, we can convert a logical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder, and a sector number within that track. In practice, it is difficult to perform, this translation, because the number of sectors per track is not a constant on some drives.

On media that use **constant linear velocity** (CLV), the density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head. This method is used in CD-ROM and DVD-ROM drives.

Alternatively, the disk rotation speed can stay constant, and the density of bits decreases from inner tracks to outer tracks to keep the data rate constant. This method is used in hard disks and is known as **constant** angular **velocity (CAV).**

**3. Disk Attachment**

Computers access disk storage in two ways. One way is via I/O ports (or host-attached storage); this is common on small systems. The other way is via a remote host in a distributed file system; this is referred to as network-attached storage.

**Host-Attached Storage**

Host-attached storage is storage accessed through local I/O ports. These ports use several technologies.

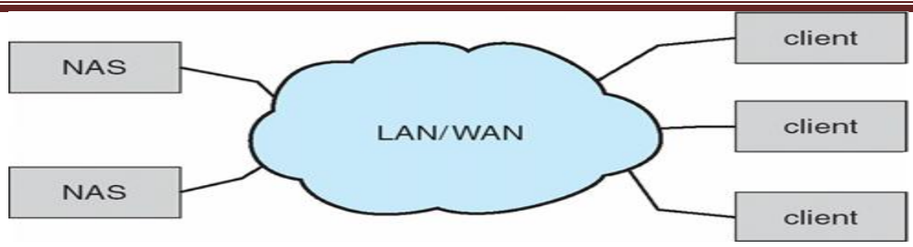
The typical desktop PC uses an I/O bus architecture called IDE or ATA. This architecture supports a maximum of two drives per I/O bus. A newer, similar protocol that has simplified cabling is SATA. High-end workstations and servers generally use more sophisticated I/O architectures, such as SCSI and fiber channel (FC).

SCSI is a bus architecture. Its physical medium is usually a ribbon cable. The SCSI protocol supports a maximum of 16 devices on the bus. Generally, the devices include one controller card in the host (the **SCSI initiator)** and up to 15 storage devices (the **SCSI targets).**

FC is a high-speed serial architecture that can operate over optical fiber or over a four-conductor copper cable. It is the basis of **storage-area networks** (SANs),

**Network-Attached Storage**

A network-attached storage (NAS) device is a special-purpose storage system that is accessed remotely over a data network. Clients access network-attached storage via a remote-procedure-call interface. The remote procedure calls (RPCs) are carried via TCP or UDP over an IP network—-usually the same local-area network (LAN) that carries all data traffic to the clients.



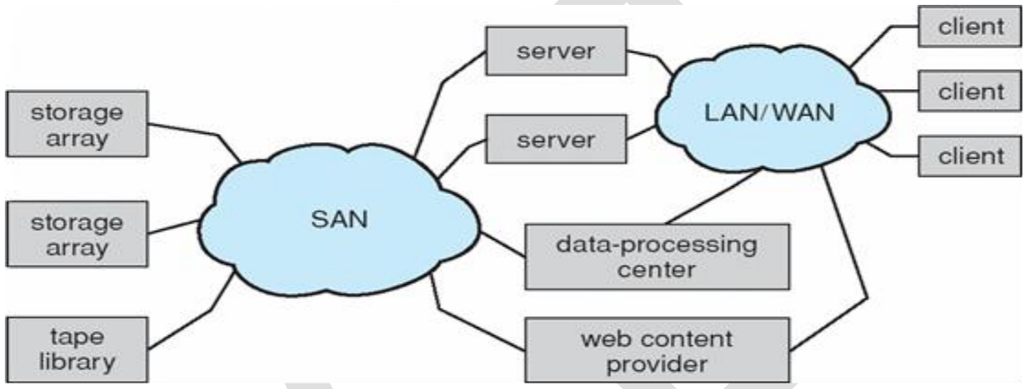

**Network-Attached Storage**

Network-attached storage provides a convenient way for all the computers on a LAN to share a pool of storage. However, it tends to be less efficient and have lower performance than some direct-attached storage options.

ISCSI is the latest network-attached storage protocol. In essence, it uses the IP network protocol to carry the SCSI protocol. Thus, networks rather than SCSI cables can be used as the interconnects between hosts and their storage. As a result, hosts can treat their storage as if it were directly attached, but the storage can be distant from the host.

**Storage-Area Network**

One drawback of network-attached storage systems is that the storage I/O operations consume bandwidth on the data network, thereby increasing the latency of network communication. It overcome by SAN.




**Storage Area Network**

A storage-area network (SAN) is a private network (using storage protocols rather than networking protocols) connecting servers and storage units, as shown in Figure. The power of a SAN lies in its flexibility. Multiple hosts and multiple storage arrays can attach to the same SAN. A SAN switch allows or prohibits access between the hosts and the storage.

An emerging alternative is a special-purpose bus architecture named InfiniBand, which provides hardware and software support for high-speed interconnection networks for servers and storage units.

**4. Disk Scheduling**

To use the hardware efficiently, the disk drives have fast access time and large disk bandwidth. The access time has two major components. The **seek time** is the time for the disk arm to move the heads to the cylinder containing the desired sector. The **rotational latency** is the additional time for the disk to rotate the desired sector to the disk head. The disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer. We can improve both the access time and the bandwidth by scheduling the servicing of disk I/O requests in a good order.

Whenever a process needs I/O to or from the disk, it issues a system call to the operating system. The request specifies several pieces of information:

- Whether this operation is input or output
- What the disk address for the transfer is
- What the memory address for the transfer is
- What the number of sectors to be transferred is

If the desired disk drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed in the queue of pending requests for that drive.

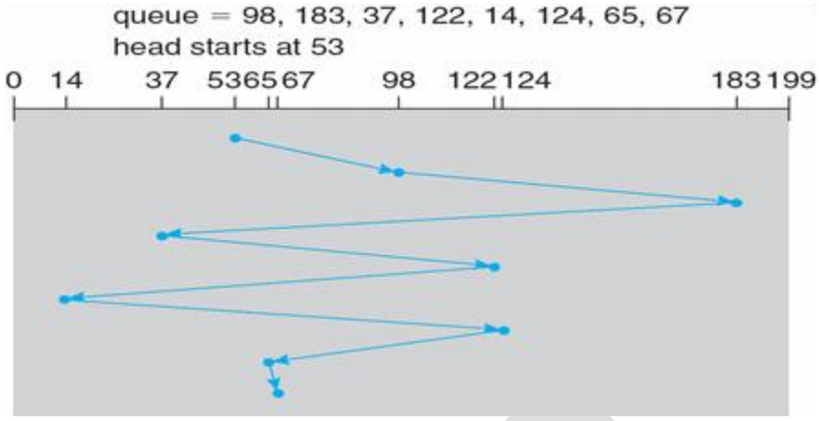
For a multiprogramming system with many processes, the disk queue may often have several pending requests. Thus, when one request is completed,, the operating system chooses which pending request to service next. Several disk-scheduling algorithms can be used, and we discuss them next.

**FCFS Scheduling**

The simplest form of disk scheduling is the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. Consider, for example, a disk queue with requests for I/O to blocks on cylinders

98, 183, 37,122, 14, 124, 65, 67,

in that order. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124/65, and finally to 67, for a total head movement of 640 cylinders.

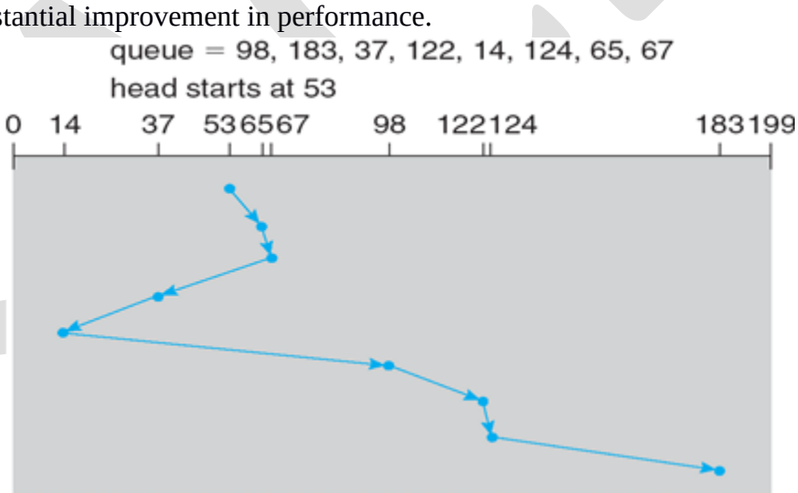



The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders 37 and 14 could be serviced together, before or after the requests at 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

**SSTF Scheduling**

The shortest-seek-time-first (SSTF) algorithm selects the request with the minimum seek time from the current head position. Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position.

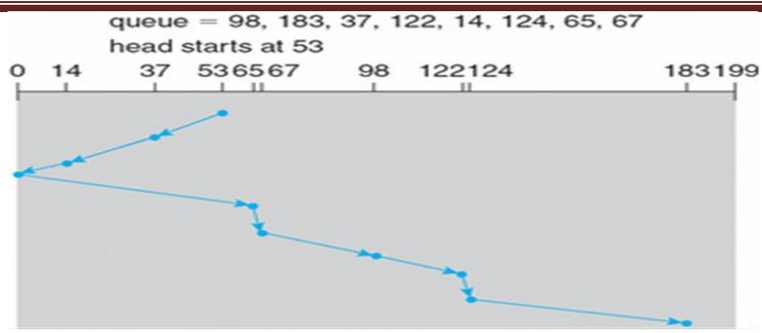
For our example request queue, the closest request to the initial head position (53) is at cylinder 65. Once we are at cylinder 65, the next closest request is at cylinder 67. From there, the request at cylinder 37 is closer than the one at 98, so 37 is served next. Continuing, we service the request at cylinder 14, then 98,122, 124, and finally 183. This scheduling method results in a total head movement of only 236 cylinders. This algorithm gives a substantial improvement in performance.




SSTF scheduling is essentially a form of shortest-job-first (SJF) scheduling; and like SJF scheduling, it may cause starvation of some requests. Remember that requests may arrive at any time. Suppose that we have two requests in the queue, for cylinders 14 and 186, and while servicing the request from 14, a new request near 14 arrives. This new request will be serviced next, making the request at 186 wait. While this request is being serviced, another request close to 14 could arrive. In theory, a continual stream of requests near one another could arrive, causing the request for cylinder 186 to wait indefinitely.

**SCAN Scheduling**

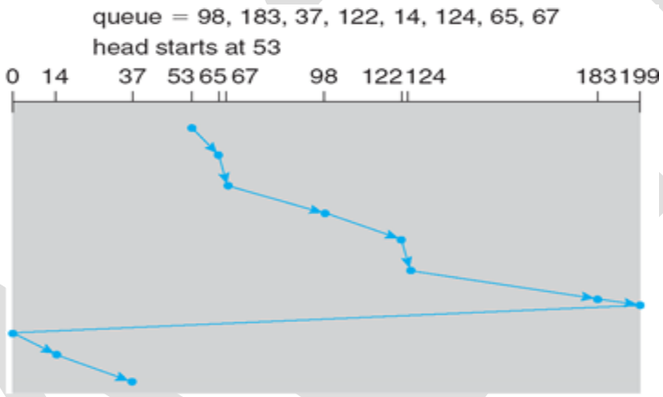
**In the SCAN algorithm, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. The SCAN algorithm is sometimes called the elevator algorithm, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.**




Let's return to our example to illustrate. Before applying SCAN to schedule the requests on cylinders 98,183, 37,122,14, 124, 65, and 67, we need to know the direction of head movement in addition to the head's current position (53). If the disk arm is moving toward 0, the head will service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183 (Figure 12.6). If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.

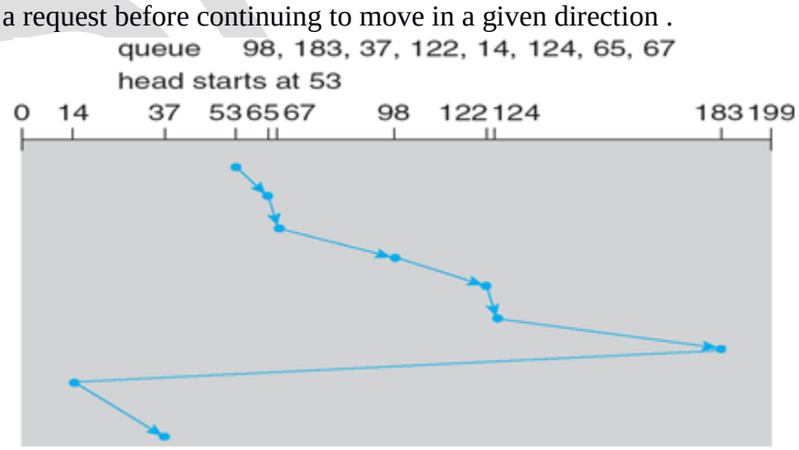
**C-SCAN Scheduling**

Circular SCAN (C-SCAN) **scheduling** is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip . The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.




**LOOK Scheduling**

As we described, both SCAN and C-SCAK move the disk arm across the full width of the disk. In practice, neither algorithm is often implemented this way. More commonly, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk. Versions of SCAN and C-SCAN that follow this pattern are called **LOOK** and **C-LOOK scheduling,** because they *look* for a request before continuing to move in a given direction .




**Selection of a Disk-Scheduling Algorithm**

**SSTF is common and has a natural appeal because it increases performance over FCFS. SCAM and C-SCAN perform better for systems that place a heavy load on the disk, because they are less likely to cause a starvation problem. With any scheduling algorithm, however, performance depends heavily on the number and types of requests. For instance, suppose that the queue usually has just one outstanding request. Then, allscheduling algorithms behave the same, because they have only one choice for where to move the disk head: They all behave like FCFS scheduling.**

Requests for disk service can be greatly influenced by the file-allocation method. A program reading a contiguously allocated file will generate several requests that are close together on the disk, resulting in limited head movement. A linked or indexed file, in contrast, may include blocks that are widely scattered on the disk, resulting in greater head movement.

**6. Swap-Space Management**

Swap-space management is another low-level task of the operating system. Virtual memory uses disk space as an extension of main memory. Since disk access is much slower than memory access, using swap space significantly decreases system performance. The main goal for the design, and implementation of swap space is to provide the best throughput for the virtual memory system. In this section, we discuss how swap space is used, where swap space is located on disk, and how swap space is managed.

**Swap-Space Use**

Swap space is used in various ways by different operating systems, depending on the memory-management algorithms in use. For instance, systems that implement swapping may use swap space to hold an entire process image, including the code and data segments. Paging systems may simply store pages that have been pushed out of main memory.

The amount of swap space needed on a system can therefore vary depending on the amount of physical memory, the amount of virtual memory it is backing, and the way in which the virtual memory is used. It can range from a few megabytes of disk space to gigabytes.Note that it may be safer to overestimate than to underestimate the amount of swap space required, because if a system runs out of swap space it may be forced to abort processes or may crash entirely. Overestimation wastes disk space that could otherwise be used for files, but it does no other harm.

Solaris, for example, suggests setting swap space equal to the amount of physical memory. Historically, Linux suggests setting swap space to double the amount of physical memory.

**Swap-Space Location**

A swap space can reside in one of two places: It can be carved out of the normal file system, or it can be in a separate disk partition. If the swap space is simply a large file within the file system, normal file-system routines can be used to create it, name it, and allocate its space. This approach, though easy to implement, is inefficient. Navigating the directory structure and the disk-allocation data structures takes time and (potentially) extra disk accesses.

Alternatively, swap space can be created in a separate raw partition, as no file system or directory structure is placed in this space. Rather, a separate swap-space storage manager is used to allocate and deallocate the blocks from the raw partition. This manager uses algorithms optimized for speed rather than for storage efficiency, because swap space is accessed much more frequently than file systems.

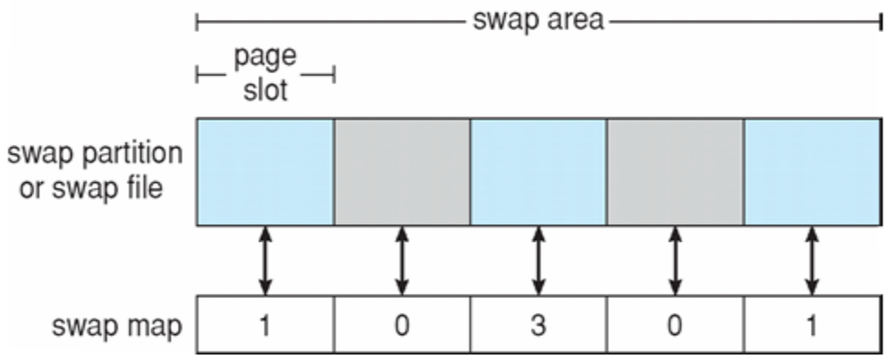
Some operating systems are flexible and can swap both in raw partitions and in file-system space. Linux is an example: The trade-off is between the convenience of allocation and management in the file system and the performance of swapping in raw partitions.

**Swap-Space Management: An Example**

We can illustrate how swap space is used by following the evolution of swapping and paging in various UNIX systems. The traditional UNIX kernel started with an implementation of swapping that copied entire processes between contiguous disk regions and memory. UNIX later evolved to a combination of swapping and paging as paging hardware became available.

In Solaris 1 (SunOS), the designers changed standard UNIX to improve efficiency and reflect technological changes. When a process executes, text-segment pages containing code are brought in from the file system, accessed in main memory, and thrown away if selected for page out. It is more efficient to reread a page from the file system than to write it to swap space and then reread it from there.

More changes were made in later versions of Solaris. The biggest change is that Solaris now allocates swap space only when a page is forced out of physical memory, rather than when the virtual memory page is first created. This scheme gives better performance on modern computers, which have more physical memory than older systems and tend to page less.




**The data structures for swapping on Linux systems**

Linux is similar to Solaris in that swap space is only used for regions of memory shared by several processes. Linux allows one or more swap areas to be established. A swap area may be in either a swap file on a regular file system or a raw swap partition. Each swap area consists of a series of 4-KB page slots, which are used to hold swapped pages. Associated with each swap area is a swap map—an array of integer counters, each corresponding to a page slot in the swap area. If the value of a counter is 0, the corresponding page slot is available. Values greater than 0 indicate that the page slot is occupied by a swapped page. The value of the counter indicates the number of mappings to the swapped page; for example, a value of 3 indicates that the swapped page is mapped to three different processes.

**7. Disk Management**

The operating system is responsible for several other aspects of disk management. Here we discuss disk initialization, booting from disk, and bad-block recovery.

**Disk Formatting**

A new magnetic disk is a platter of a magnetic recording material. Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called low-level formatting, or physical formatting. Low-level formatting fills the disk with a special data structure for each sector. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size), and a trailer. The header and trailer contain information used by the disk controller, such as a sector number
and an error-correcting code (ECC). When the controller writes a sector of data during normal I/O, the ECC is updated with a value calculated from all the bytes in the data area. When the sector is read, the ECC is recalculated and is compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad
Most hard disks are low-level-formatted at the factory as a part of the manufacturing process. This formatting enables the manufacturer to test the disk and to initialize the mapping from logical block numbers to defect-free sectors on the disk. For many hard disks, when the disk controller is instructed to low-level-format the disk, it can also be told how many bytes of data space to leave between the header and trailer of all sectors. It is usually possible to choose among a few sizes, such as 256, 512, and 1,024 bytes. Formatting a disk with a larger sector size means that fewer sectors can fit on each track; but it also means that fewer headers and trailers are written on each track and more space is available for user data.

To use a disk to hold files, the operating system still needs to record its own data structures on the disk. It does so in two steps. The first step is to **partition** the disk into one or more groups of cylinders. The operating system can treat each partition as though it were a separate disk. For instance, one partition can
hold a copy of the operating system's executable code, while another holds user files. After partitioning, the second step is **logical formatting**. In this step, the operating system stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space (a FAT or modes) and an initial empty directory.

**Boot Block**

When a computer is powered up or rebooted—it must have an initial program to run. This initial *bootstrap* program tends to be simple. It initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the operating system. To do its job, the bootstrap program finds the operatingsystem kernel on disk, loads that kernel into memory, and jumps to an initial address to begin the operating-system execution.

For most computers, the bootstrap is stored in read-only memory (ROM). This location is convenient, because ROM is at a fixed location that the processor can start executing when powered up or reset. And, since ROM is read only, it cannot be infected by a computer virus. The problem is that changing this bootstrap code requires changing the ROM, hardware chips.For this reason, most systems **store a tiny bootstrap loader**

program in the boot ROM whose only job is to bring in a full bootstrap program from disk. The full bootstrap program can be changed easily: A new version is simply written onto the disk. The full bootstrap program is stored in "the boot blocks" at a fixed location on the disk. A disk that has a boot partition is called a boot disk or system disk.

**Bad Blocks**

Because disks have moving parts and small tolerances, they are prone to failure. Sometimes the failure is complete; in this case, the disk needs to be replaced and its contents restored from backup media to the new disk. More frequently, one or more sectors become defective. Most disks even come from the factory with bad blocks. Depending on the disk and controller in use, these blocks are handled in a variety of ways.

On simple disks, such as some disks with IDE controllers, bad blocks are handled manually. For instance, the MS-DOS format command performs logical formatting and, as a part of the process, scans the disk to find bad blocks. If format finds a bad block, it writes a special value into the corresponding FAT entry to tell the allocation routines not to use that block. If blocks go bad during normal operation, a special program (such as chkdsk) must be run manually to search for the bad blocks and to lock them away as before. Data that resided on the bad blocks usually are lost.

More sophisticated disks, such as the SCSI disks used in high-end PCs and most workstations and servers, are smarter about bad-block recovery. The controller maintains a list of bad blocks on the disk. The list is initialized during the low-level formatting at the factory and is updated over the life of the disk.

Low-level formatting also sets aside spare sectors not visible to the operating system. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **sector sparing** or **forwarding.**

A typical bad-sector transaction might be as follows:

1. The operating system tries to read logical block 87.
2. The controller calculates the ECC and finds that the sector is bad. It reports this finding to the operating system.
3. The next time the system is rebooted, a special, command is run to tell the
4. SCSI controller to replace the bad sector with a spare.
5. After that, whenever the system requests logical block 87, the request is translated into the replacement sector's address by the controller.

Such a redirection by the controller could invalidate any optimization by the operating system's disk-scheduling algorithm! For this reason, most disks are formatted to provide a few spare sectors in each, cylinder and a spare cylinder as well. When a bad block is remapped, the controller uses a spare sector from the same cylinder, if possible.

As an alternative to sector sparing, some controllers can be instructed to replace a bad block by sector slipping. Here is an example: Suppose that logical block 17 becomes defective and the first available spare follows sector 202. Then, sector slipping remaps all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 is copied into the spare, then sector 201 into 202, and then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18, so sector 17 can be mapped to it.

The replacement of a bad block generally is not totally automatic because the data in the bad block are usually lost.