Unit V - Storage Management

Since main memory is usually too small to accommodate all the data and programs permanently, the computer system must provide secondary storage to back up main memory. Modern computer systems use disks as the primary on-line storage medium for information (both programs and data). The file system provides the mechanism for on-line storage of and access to both data and programs residing on the disks. A file is a collection of related information defined by its creator. The files are mapped by the operating system onto physical devices. Files are normally organized into directories for ease of use.

File System

For most users, the file system is the most visible aspect of an operating system. It provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system. The file system consists of two distinct parts: a collection of *files,* each storing related data, and a *directory structure,* which organizes and provides information about all the files in the system. File systems reside permanently on secondary storage devices.

File Concept

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

A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user.

Many different types of information may be stored in a file - source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on. *A* file has a certain defined *structure,* which depends on its type. *A text* file is a sequence of characters organized into lines (and possibly pages). *A source* file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements. An *object* file is a sequence of bytes organized into blocks understandable by the system's linker. An *executable* file is a series of code sections that the loader can bring into memory and execute.

- **File Attributes**

A file is named, for the convenience of its human users, and is referred to by its name. *A* name is usually a string of characters. Some systems differentiate between uppercase and lowercase characters in names, whereas other systems do not. When a file is named, it becomes independent of the process, the user, and even the system that created it. The file's owner might write the file to a pen drive, send it in an e-mail, or copy it across a network.

The information about all files is kept in the directory structure, which also resides on secondary storage. A file's attributes vary from one operating system to another but typically consist of these:

- ✓ **Name**. The symbolic file name is the only information kept in human readable form.
- ✓ **Identifier**. This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- ✓ **Type.** This information is needed for systems that support different types of files.
- ✓ **Location.** This information is a pointer to a device and to the location of the file on that device.
- ✓ **Size.** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- ✓ **Protection.** Access-control information determines who can do reading, writing, executing, and so on.
- ✓ **Time, date, and user identification**. This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

• File Operations

A file is an *abstract data type.* To define a file properly, it has to consider the operations that can be performed on files. The operating system can provide system calls to create the following six basic file operations.

Creating a file. Two steps are necessary to create a file. First, space in the file system must be found for the file, and 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. The write pointer must be updated whenever a write occurs.

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. Once the read has taken place, the read pointer is updated.

Repositioning within a file. The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/0. 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.

Other common operations include *appending* new information to the end of an existing file and *renaming* an existing file. These primitive operations can then be combined to perform other file operations. For instance, we can create a *copy* of a file, or copy the file to another I/O device, such as a printer or a display, by creating a new file and then reading from the old and writing to the new.

Most of the file operations 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. Several pieces of information are associated with an open file is specified below.

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.

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. The file-open counter tracks the number of opens and closes and reaches zero on the last close. The system can then remove the 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/0 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. For example, most operating systems allow users to specify a file name as a sequence of characters followed by a period and terminated by an extension of additional characters. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways. File name examples include *resume.doc, Server.java,* and *ReaderThread.*c.

• File Structure

File types also can be used to indicate the internal structure of the file. Certain files must conform to a required structure that is understood by the operating system. For example, the operating system requires that an executable file 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. All operating systems must support at least one structure-that of an executable file-so that the system is able to load and run programs.

Access Methods

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

- **Sequential Access**

The simplest access method is sequential access. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.

Reads and writes make up the bulk of the operations on a file. A read operation read *next-reads* the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, the write *operation-write* next-appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning; and on some systems, a program may be able to skip forward or backward *n* records Sequential access is based on a tape model of a file and works as well on sequential-access devices as it does on random-access ones.

- **Direct Access**

A file is made up of fixed-length that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type. When a query concerning a particular subject arrives, we compute which block contains the answer and then read that block directly to provide the desired information.

- **Other Access Methods**

These methods generally involve the construction of an index for the file. The *index* like an index in the back of a book 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.

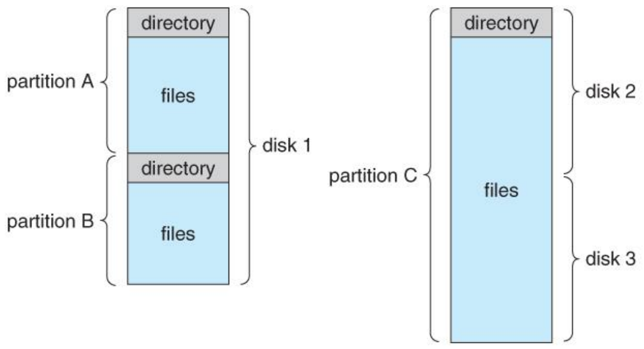
With large files, the index file itself may become too large to be kept in memory. One solution is to create an index for the index file. The primary index file would contain pointers to secondary index files, which would point to the actual data items.

Directory and Disk Structure

A storage device can be used in its entirety for a file system. It can also be subdivided for finer-grained control. For example, a disk can be *partitioned* into quarters, and each quarter can hold a file system. Storage devices can also be collected together into RAID (Redundant Array of Independent Disks) sets that provide protection from the failure of a single disk. Sometimes, disks are subdivided and also collected into RAID sets.

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, such as swap space or unformatted (raw) disk space. Partitions are also known as *slices*. 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 into a RAID set. Each volume can be thought of as a virtual disk. Volumes can also store multiple operating systems, allowing a system to boot and run more than one operating system.

Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a *device directory* or *volume table of contents*. The device directory (more commonly known simply as that *directory*) records information - such as name, location, size, and type - for all files on that volume. The following figure shows a typical file-system organization.



(A typical file-system organization)

- ## Storage Structure

A general-purpose computer system has multiple storage devices, and those devices can be sliced up into volumes that hold file systems. Computer systems may have zero or more file systems, and the file systems may be of varying types.

- ## Directory Overview

The directory can be viewed as a symbol table that translates file names into their directory entries. The directory itself can be organized in many ways. It is able to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory. When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

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

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

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

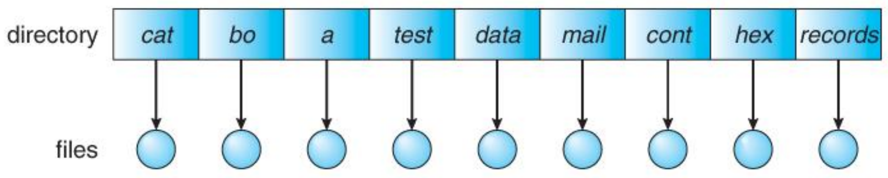
List a directory. We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.

Rename a file. Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.

Traverse the file system. We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, we do this by copying all files *to* magnetic tape. This technique provides a backup copy in case of system failure. In addition, if a file is no longer in use, the file can be copied *to* tape and the disk space of that file released for reuse by another file.

- ## 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.

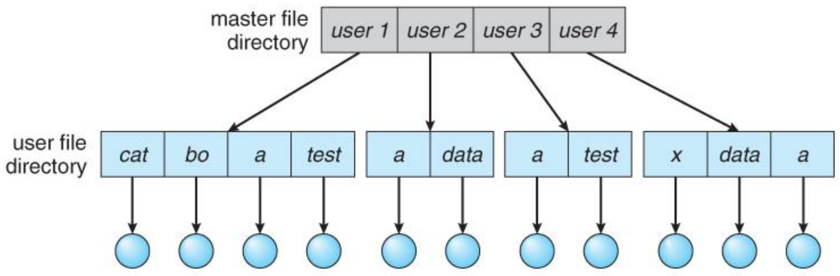


(Single-level directory)

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. If two users call their data file *test*, then the unique-name rule is violated. The MS-DOS operating system allows only 11-character file names; UNIX, in contrast, allows 255 characters

- ## Two-Level Directory

In the two-level directory structure, each user has the own *user file directory* (UDF). 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.



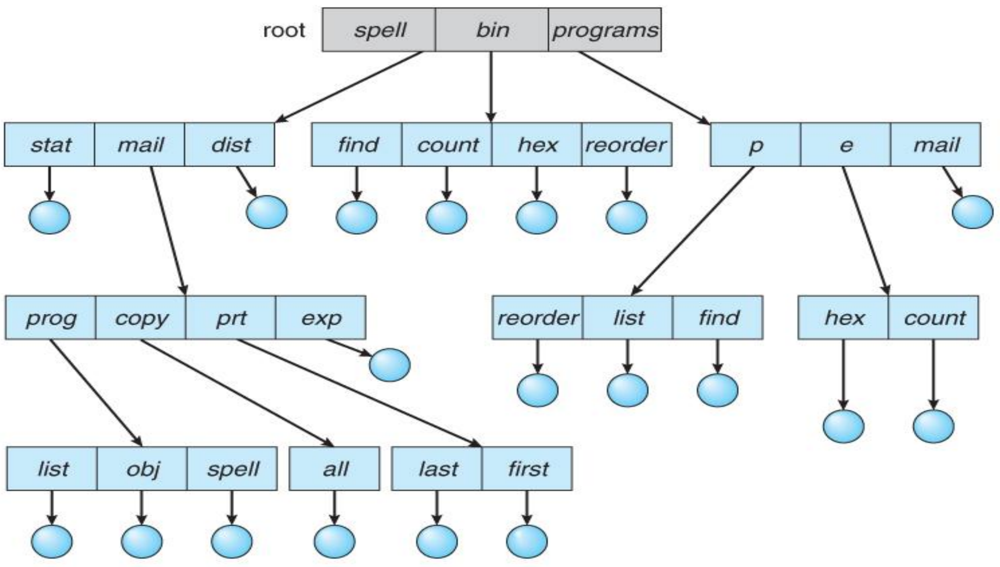
(Two-level directory structure)

When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique. To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

A two-level directory can be thought of as a tree, or an inverted tree, of height 2. The root of the tree is the MFD. Its direct descendants are the UFDs. The descendants of the UFDs are the files themselves. The files are the leaves of the tree. Specifying a user name and a file name defines a path in the tree from the root (the MFD) to a leaf (the specified file). Thus, a user name and a file name define a *path name*. Every file in the system has a path name. To name a file uniquely, a user must know the path name of the file desired.

- ## Tree-Structured Directories

The natural generalization is to extend the directory structure to a tree of arbitrary height. This generalization allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.



(Tree-structured 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.

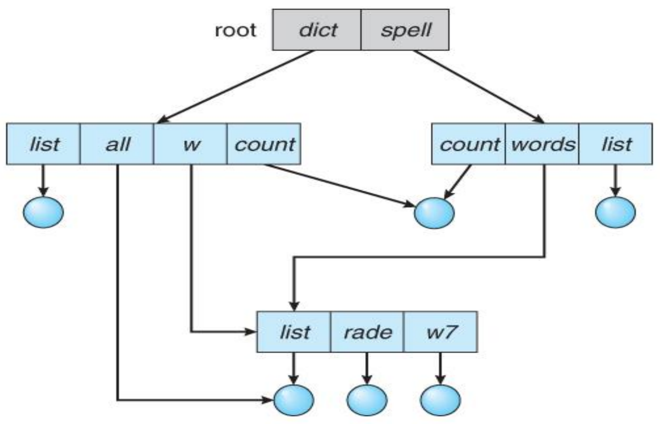
In normal use, each process has a current directory. The *current directory* should contain most of the files that are of current interest to the process. When reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file. To change directories, a system call is provided that takes a directory name as a parameter and uses it to redefine the current directory.

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 the directory names on the path. A *relative path* defines a path from the current directory. 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.*

• Acyclic-Graph Directories

Consider two programmers who are working on a joint project. The files associated with that project can be stored in a subdirectory, separating them from other projects and files of the two programmers. But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. The common subdirectory should 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 (figure below). The *same* file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.



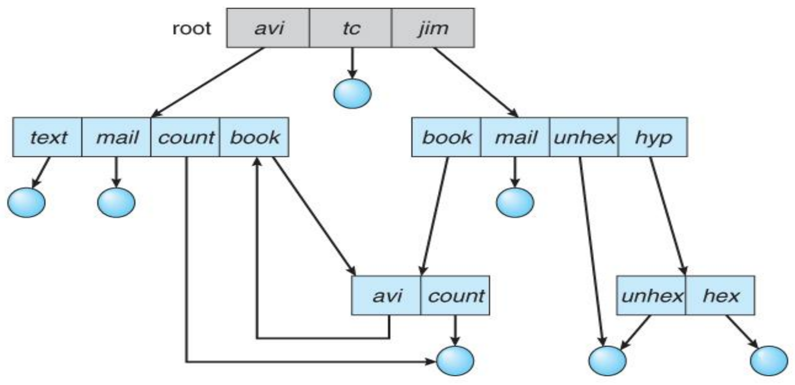
(Acyclic-graph directory structure)

It is important to note that a shared file (or directory) is not the same as two copies of the file. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy. 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.

Shared files and subdirectories can be implemented in several ways. A common way, exemplified by many of the UNIX systems, is to create a new directory entry called a link. A link is effectively a pointer to another file or subdirectory. If the directory entry is marked as a link, then the name of the real file is included in the link information. We resolve the link by using that path name to locate the real file. Links are easily identified by their format in the directory entry (or by having a special type on systems that support types) and are effectively indirect pointers. The operating system ignores these links when traversing directory trees to preserve the acyclic structure of the system.

• General Graph Directory

A serious problem with using an acyclic-graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature. However when we add links, the tree structure is destroyed, resulting in a simple graph structure (figure below).



(General graph directory)

If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance. A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating. One solution is to limit arbitrarily the number of directories that will be accessed during a search.

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 (or a cycle) in the directory structure. In this case, we generally need to use a garbage-collection scheme to determine when the last reference has been deleted and the disk space can be reallocated.

Garbage collection involves traversing the entire file system, marking everything that can be accessed. Garbage collection is necessary only because of possible cycles in the graph. Thus, an acyclic-graph structure is much easier to work with. The difficulty is to avoid cycles as new links are added to the structure. There are algorithms to detect cycles in graphs; however, they are computationally expensive, especially when the graph is on disk storage.

Implementing File Systems

The file system provides the mechanism for on-line storage and access to file contents, including data and programs. The file system resides permanently on *secondary storage*, which is designed to hold a large amount of data permanently. While considering the most common secondary-storage medium, the disk, it explores the ways to structure file use, to allocate disk space, to recover freed space, to track the locations of data, and to interface other parts of the operating system to secondary storage.

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:

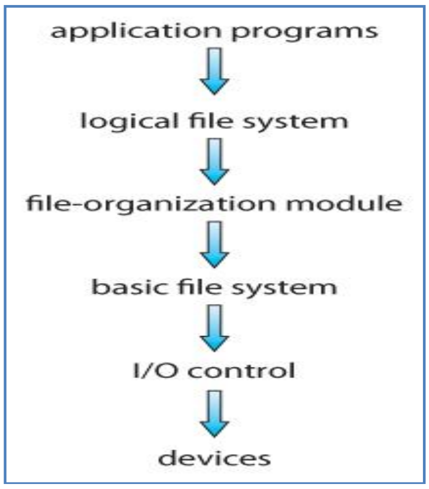
- A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same place.

- A disk can access directly any block of information it contains. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another requires only moving the read-write heads and waiting for the disk to rotate.

To improve I/O efficiency, I/O transfers between memory and disk are performed in units of *blocks*. Each block has one or more sectors. Depending on the disk drive, sector size varies from 32 bytes to 4,096 bytes; the usual size is 512 bytes.

File Systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily. A file system poses two quite different design problems. The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files. The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

The file system itself is generally composed of many different levels. The structure shown in the following figure is an example of a layered design. Each level in the design uses the features of lower levels to create new features for use by higher levels.



(Layered file system)

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, which interfaces the I/0 device to the rest of the system. The device driver usually writes specific bit patterns to special locations in the I/0 controller's memory to tell the controller which device location to act on and what actions to take.

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, sector 10). This layer also manages the memory buffers and caches that hold various file-system, directory, and data blocks. A block in the buffer is allocated before the transfer of a disk block can occur. When the buffer is full, the buffer manager must find more buffer memory or free up buffer space to allow a requested I/O to complete.

The ***file organization module*** knows about files and their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer. 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.

The ***logical file system*** manages metadata information. Metadata includes all of the file-system structure except the actual *data* (or contents of the files). The logical file system manages the directory structure to provide the file organization module with the information the latter needs, given a symbolic file name. It maintains file structure via file-control blocks. A *file control block* contains information about the file, including ownership, permissions, and location of the file contents.

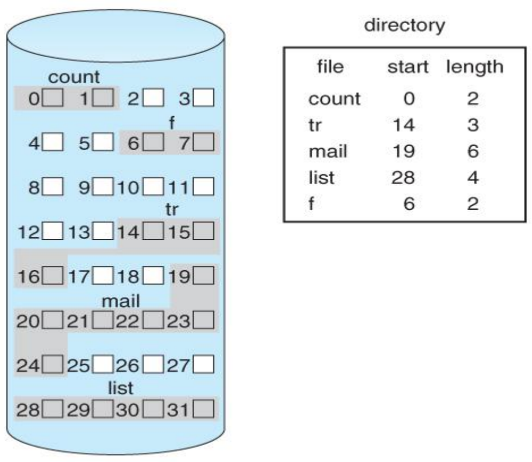
Allocation Methods

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

• Contiguous Allocation

Contiguous Allocation requires that each file occupy a set of contiguous blocks on disk. Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block $b + 1$ after block b normally requires no head movement. When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), 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, as is seek time when a seek is finally needed.

Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block. If the file is n blocks long and starts at location b, then it occupies blocks $b, b + 1, b + 2, \dots, b + n - 1$. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file (Figure below).



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

(Contiguous 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, 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. The system chosen to manage free space determines how this task is accomplished. The contiguous-allocation problem can be seen as a particular application of the general *dynamic storage allocation* problem which involves 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.

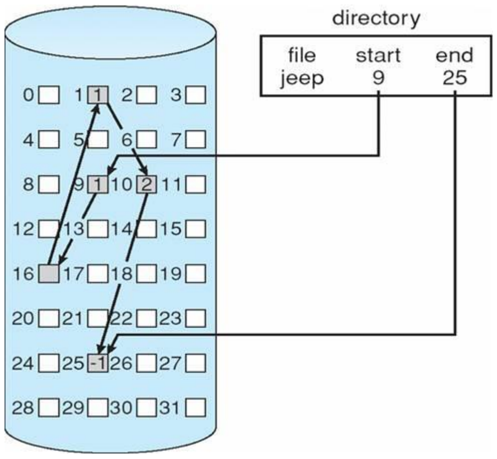
Problems with contiguous allocation is determining how much space is needed for a file, when the file is created, the total amount of space it will need must be found and allocated and identify the size of the file to be created. In some cases, this determination may be fairly simple (copying an existing file, for example); in general, however, the size of an output file may be difficult to estimate. Even if the total amount of space needed for a file is known in advance, pre-allocation may be inefficient. 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

To minimize these drawbacks, some operating systems use a modified contiguous-allocation scheme. Here, a contiguous chunk of space is allocated initially; 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. On some systems, the owner of the file can set the extent size, but this setting results in inefficiencies if the owner is incorrect.

• 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 (figure 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.



(Linked allocation of disk space)

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* (the end-of-list pointer value) 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. The size of a file need not be declared when that file is created.
A file can continue to grow as long as free blocks are available. Consequently, it is never necessary to compact disk space.

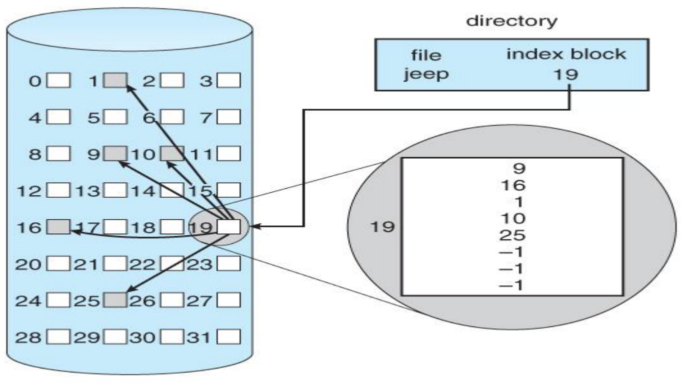
Linked allocation does have disadvantages, however. The major problem is that it can be used effectively only for sequential-access files. To find the *i*th block of a file, we must start at the beginning of that file and follow the pointers until we get to the *i*th block. Each access to a pointer requires a disk read, and some require a disk seek. Consequently, it is inefficient to support a direct-access capability for linked-allocation 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. Each file requires slightly more space than it would otherwise.

- **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, 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 bringil1.g 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 (Figure below). To find and read the i*th* block, we use the pointer in the i*th* index-block entry.



(Indexed allocation of disk space)

Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space. Indexed allocation does suffer from wasted space, however. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation. Consider a common case in which we have a file of only one or two blocks. With linked allocation, we lose the space of only one pointer per block. With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-nil.

Free-Space Management

Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. 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. When a file is deleted, its disk space is added to 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

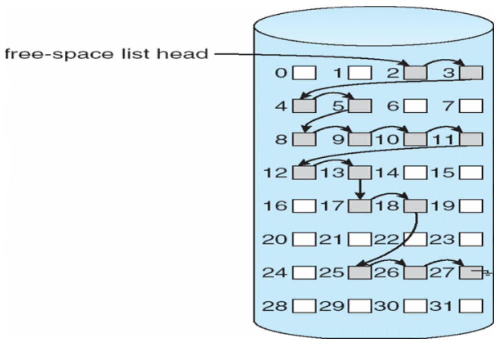
001111001111110001100000011100000 ...

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. Indeed, many computers supply bit-manipulation instructions that can be used effectively for that purpose. 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,

(number of bits per word) x (number of 0-value words) +offset of first 1 bit.

- **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 the 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 (figure below). This scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time.



(Linked free-space list on disk.)

- **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. Although each entry requires more space than would a simple disk address, the overall list is shorter, as long as the count is generally greater than 1. Note that this method of tracking free space is similar to the extent method of allocating blocks. These entries can be stored in a B-tree, rather than a linked list for efficient lookup, insertion, and deletion.

- **Space Maps**

Sun's ZFS (Zettabyte F**ile System**) was designed to encompass huge numbers of files, directories, and even file systems (in ZFS, we can create file-system hierarchies). The resulting data structures could have been large and inefficient if they had not been designed and implemented properly. On these scales, metadata I/0 can have a large performance impact. Consider for example, that if the free space list is implemented as a bit map, bit maps must be modified both when blocks are allocated and when they are freed. Freeing 1GB of data on a 1-TB disk could cause thousands of blocks of bit maps to be updated, because those data blocks could be scattered over the entire disk.

Disk Scheduling

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having 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.

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:

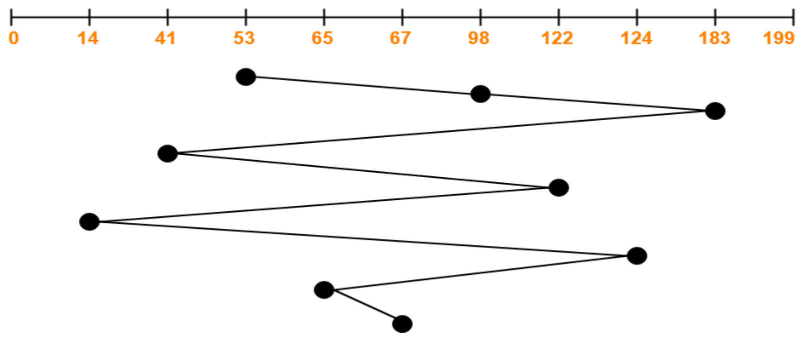
- Whether this operation is input or output
- What the disk address for the transfer is
- What the memory address for the transfer is
- What the number of sectors to be transferred is

• FCFS Scheduling

The simplest form of disk scheduling is, of course, 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.



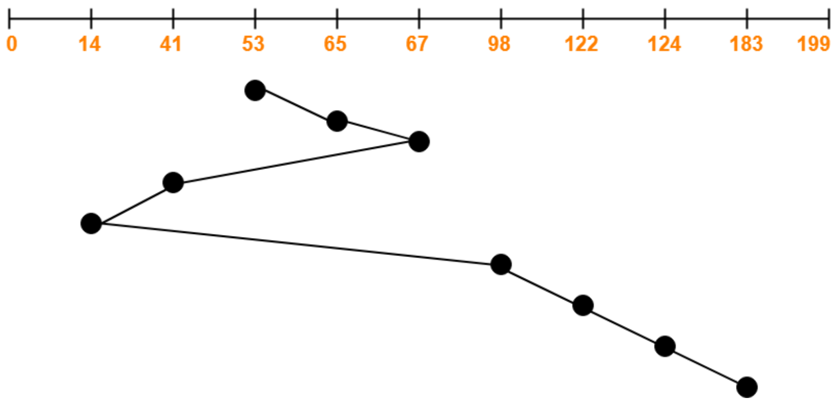
(FCFS disk scheduling)

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 for 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

• SSTF Scheduling

It seems reasonable to service all the requests close to the current head position before moving the head far to service other requests. This assumption is the basis for the *shortest-seek-time-first (SSTF) algorithm*. The SSTF algorithm selects the request with the least 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 (Figure below). This scheduling method results in a total head movement of only 236 cylinders - little more than one-third of the distance needed for FCFS scheduling of this request queue. SSTF scheduling is essentially a form of shortest-job-first (SJF) scheduling. Clearly, this algorithm gives a substantial improvement in performance.

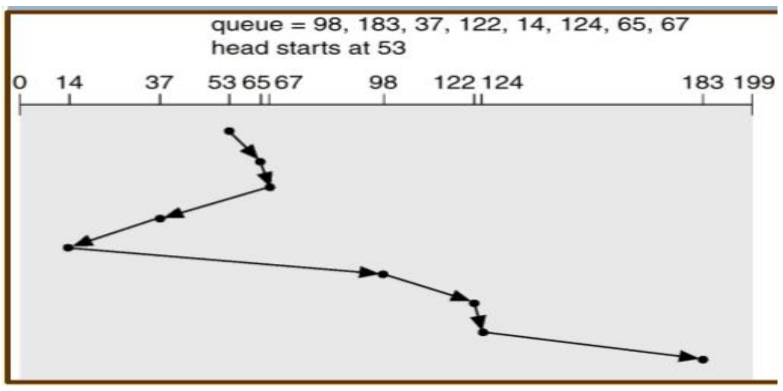


(SSTF disk scheduling)

- ## 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.

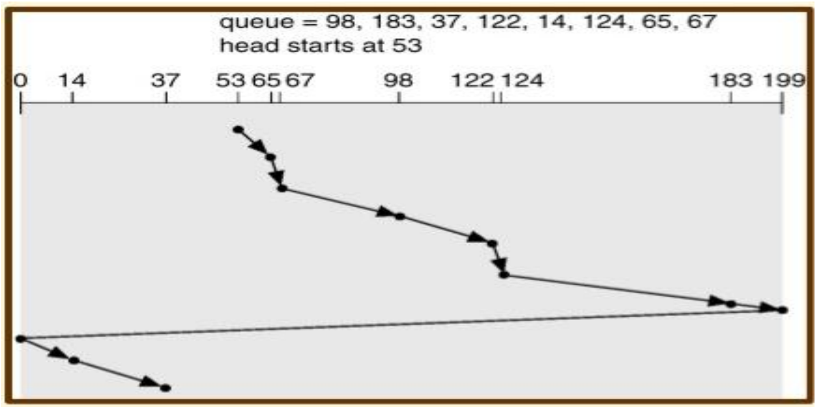
Assuming that the disk arm is moving toward 0 and that the initial head position is again 53, the head will next service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183 (Figure below). If a request 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.



(SCAN disk scheduling)

- ## C-SCAN Scheduling

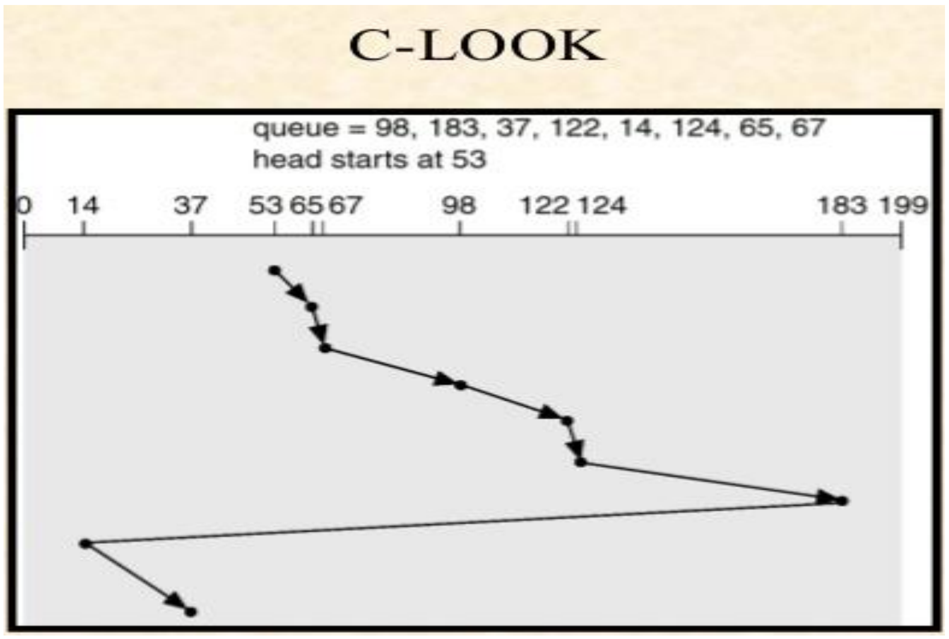
Circular SCAN(C-SCAN) scheduling is variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk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



(C-SCAN disk scheduling)

• LOOK Scheduling

Both SCAN and C-SCAN 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 (figure below).



****