**5132 – OPERATING SYSTEM**

**MODULE 4**

**CO4 Explain file organization and disk scheduling algorithms**

File system - Concept of file and directory - Various file operations - File organization concepts – sequential and indexed. Different directory structures – single level, two-level, and tree structured directories. - Different allocation methods – contiguous, linked and indexed allocations. Various disk scheduling algorithms-FCFS, SSTF, Scan, C-Scan, Look & C-Look.

**FILE SYSTEM**

**Concept Of File And Directory**

*A* **file** is a named collection of related information that is recorded on secondary storage. Commonly, files represent programs and data. Data files may be numeric, alphabetic, alphanumeric, or binary. 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 and is referred to by its name. *A* name is usually a string of characters, such as *example.c.* Some systems differentiate between uppercase and lowercase characters in names, whereas other systems do not.

*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.

The information about all files is kept in the **directory structure**, which also resides on secondary storage. 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. Because directories, like files, must be nonvolatile, they must be stored on the device and brought into memory piecemeal, as needed.

**VARIOUS FILE OPERATIONS**

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. 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. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process current-file-position-pointer. Both the read and write operations use this same pointer, saving space and reducing system complexity.

**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. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged –except for file length-but lets the file be reset to length zero and its file space released.

These six basic operations comprise the minimal set of required file operations. 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.

**FILE ORGANIZATION CONCEPTS**

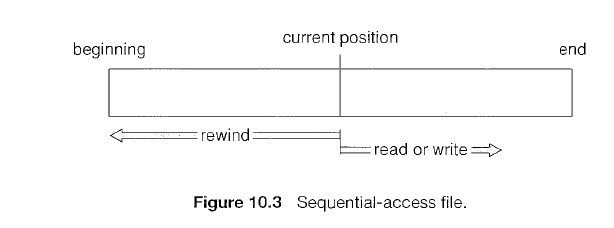
Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems provide only one access method for files. Other systems, such as those of IBM, support many access methods, and choosing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

**SEQUENTIAL AND INDEXED**

**Sequential Access**

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

A read operation –***read next****-* readsthe 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 (the new end of file).

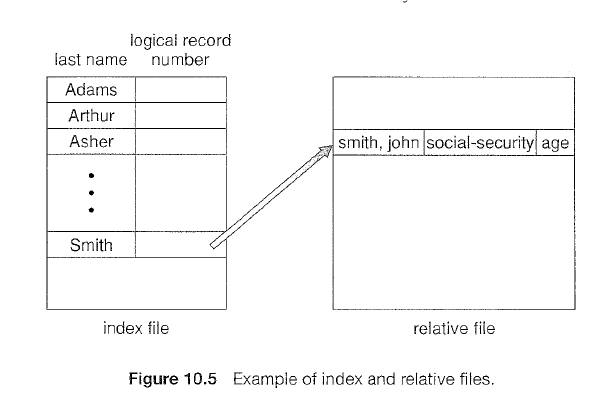


**Indexed Access**

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.

From this search, we learn exactly which block contains the desired record and access that block. This structure allows us to search a large file doing little I/0. 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.

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. The file is kept sorted on a defined key. To find a particular item, we first make a binary search of the master index, which provides the block number of the secondary index. This block is read in, and again a binary search is used to find the block containing the desired record. Finally, this block is searched sequentially. In this way, any record can be located from its key by at most two direct-access reads.



**DIFFERENT DIRECTORY STRUCTURES**

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.

**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.

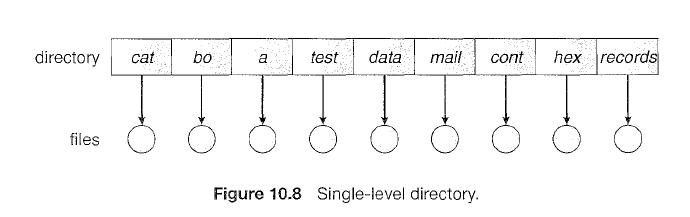
**Traverse the file system**. We may wish to access every directory and every file within a directory structure.

In. the following sections, we describe the most common schemes for defining the logical structure of a directory.

**SINGLE LEVEL DIRECTORY STRUCTURE**

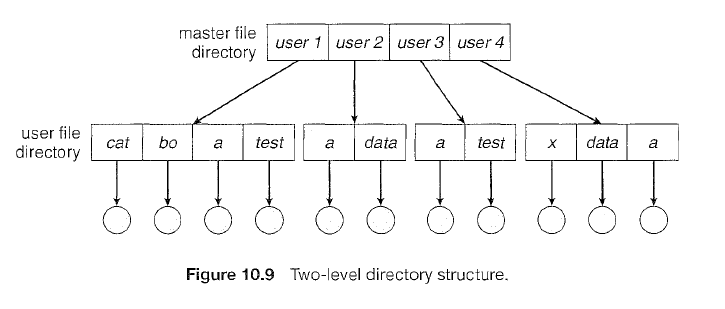
The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand (Figure 10.8).

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.



**TWO-LEVEL DIRECTORY STRUCTURE**

In the two-level directory structure, a *separate* directory is there for each user. Each user has his own ***User File Directory***. 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 (Figure 10.9).



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.

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

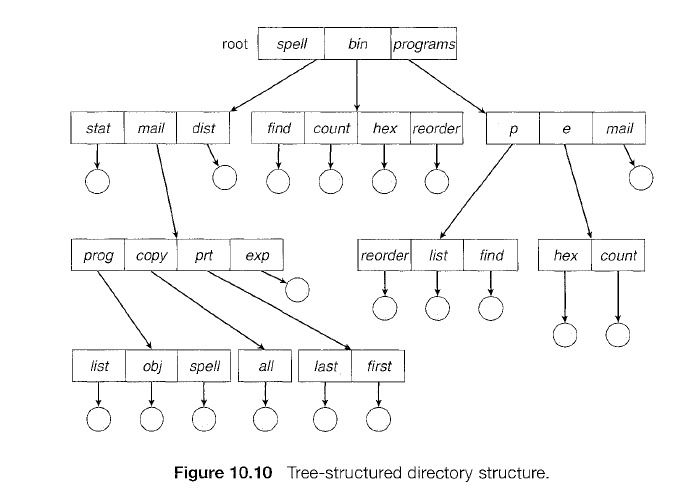
Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. Isolation is an advantage when the users are completely independent but is a disadvantage when the users *want* to cooperate on some task and to access one another's files.

A two-level directory can be thought of as a 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.

For example, if user A wishes to access her own test file named *test,* she can simply refer to *test.* To access the file named *test* of user B (with directory-entry name *userb),* however, she might have to refer to */userb/test.* Every system has its own syntax for naming files in directories other than the user's own.

**TREE-STRUCTURED DIRECTORIES**

We can extend the two-level directory structure to a tree of arbitrary height (Figure 10.10). The tree has a root directory, and every file in the system has a unique path name.



A directory (or subdirectory) contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and delete directories.

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. Thus, the user can change his current directory whenever he desires.

The current directory of any subprocess is usually the current directory of the parent when it was spawned.

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 name defines a path from the current directory. For example, in the tree-structured file system of Figure 10.10, if the current directory is *root/spell/mail,* then the relative path name *prt/first* refers to the same file as does the absolute path name *root/spell/mail/prt/first.*

An interesting policy decision in a tree-structured directory concerns how to handle the deletion of a directory. If a directory is empty, its entry in the directory that contains it can simply be deleted. However, suppose the directory to be deleted is not empty but contains several files or subdirectories. One of two approaches can be taken. Some systems, such as MS-DOS, will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory. If any subdirectories exist this procedure must be applied recursively to them, so that they can be deleted also. This approach can result in a substantial amount of work. An alternative approach, such as that taken by the UNIX rm command, is to provide an option: when a request is made to delete a directory, all that directory's files and subdirectories are also to be deleted. Here an entire directory structure can be removed with one command. If that command is issued in error, a large number of files and directories will need to be restored (assuming a backup exists).

With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users. For example, user B can access a file of user A by specifying its path names. User B can specify either an absolute or a relative path name. Alternatively, user B can change her current directory to be user *A's* directory and access the file by its file names.

**DIFFERENT ALLOCATION METHODS**

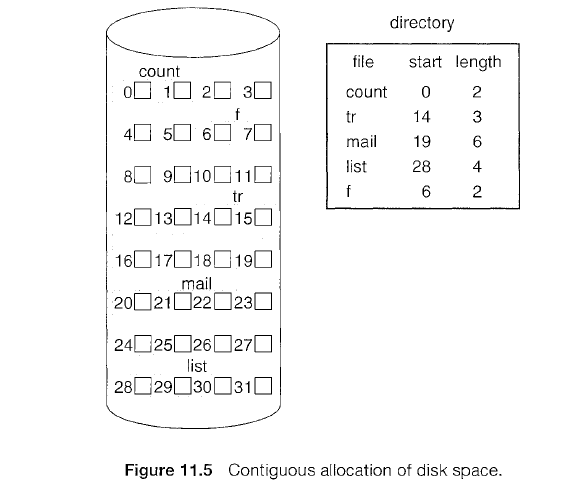
File allocation methods describe how to allocate space in the disk so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are in wide use: contiguous, linked, and indexed. Each method has advantages and disadvantages. Some systems (such as Data General's RDOS for its Nova line of computers) support all three. More commonly, a system uses one method for all files within a file-system type.

**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. The IBM VM/CMS operating system uses contiguous allocation because it provides such good performance.

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, ... , *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 11.5).

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.



**Advantages:**

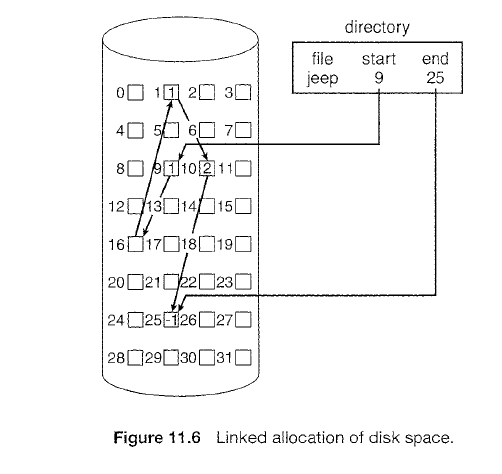
* Both the Sequential and Direct Accesses are supported by this. For direct access, the address of the ith block of the file which starts at block b can easily be obtained as (b+i).
* This is extremely fast since the number of seeks are minimal because of contiguous allocation of file blocks.

**Disadvantages:**

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

**LINKED 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 11.6). Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes in size, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.



To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialized to *nil* (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 filed 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 filed the ith block of a file, we must start at the beginning of that file and follow the pointers until we get to the ith 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.

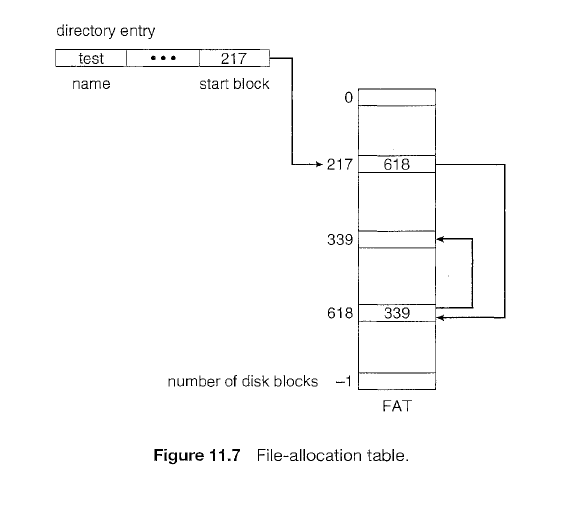
The usual solution to this problem is to collect blocks into multiples, called clusters and to allocate clusters rather than blocks. For instance, the file system may define a cluster as four blocks and operate on the disk only in cluster units. Pointers then use a much smaller percentage of the file's disk space.

The cost of this approach is an increase in internal fragmentation, because more space is wasted when a cluster is partially full than when a block is partially full.

Yet another problem of linked allocation is reliability. Recall that the files are linked together by pointers scattered all over the disk, and consider what would happen if a pointer were lost or damaged. A bug in the operating-system software or a disk hardware failure might result in picking up the wrong pointer. This error could in turn result in linking into the free-space list or into another file. One partial solution is to use doubly linked lists, and another is to store the file name and relative block number in each block; however, these schemes require even more overhead for each file.

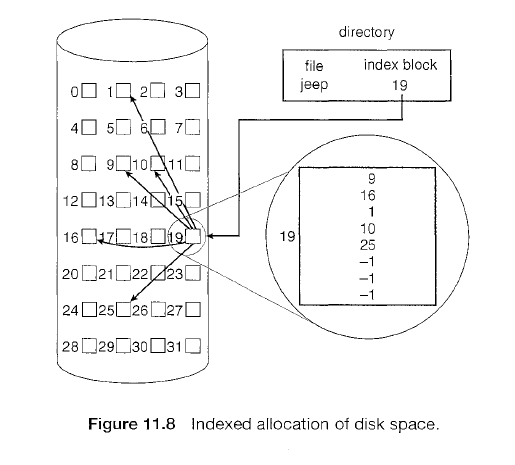
An important variation on linked allocation is the use of a **File Allocation Table (FAT).** This simple but efficient method of disk-space allocation is used by the MS-DOS and OS/2 operating systems. A section of disk at the beginning of each volume is set aside to contain the table. The table has one entry for each disk block and is indexed by block number. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number contains the block number of the next block in the file. This chain continues until it reaches the last block, which has a special end-of-file value as the table entry.

An unused block is indicated by a table value of 0. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-of-file value. An illustrative example is the FAT structure shown in Figure 11.7 for a file consisting of disk blocks 217, 618, and 339.



**INDEXED ALLOCATION**

Indexed allocation brings all the pointers together into one location: **the index block,** instead of FAT. 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 11.8). To find and read the *i th* block, we use the pointer in the *i th* index-block entry.



When the file is created, all pointers in the index block are set to *nil.* When the ith block is first written, a block is obtained from the free-space manager and its address is put in the ith index-block entry.

Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space. 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.

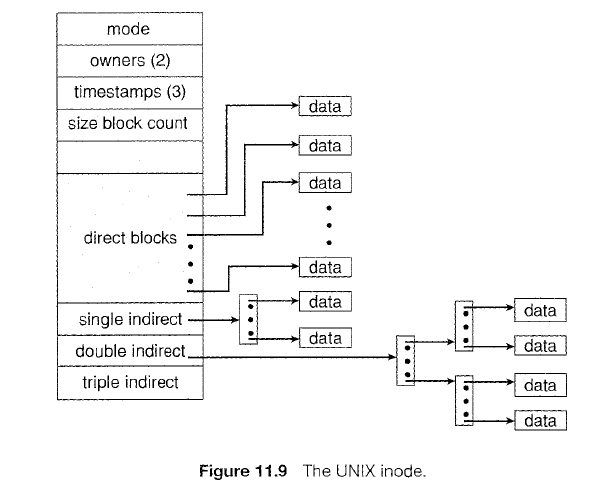
This point raises the question of how large the index block should be. Every file must have an index block, so we want the index block to be as small as possible. If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue. 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. For example, an index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses. The next address (the last word in the index block) is *nil* (for a small file) or is a pointer to another index block (for a large file).

• **Multilevel index.** A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size.

• **Combined scheme.** Another alternative is to keep the first, say, 15 pointers of the index block in the file's inode (information node). The first 12 of these pointers point to direct blocks; that is, they contain addresses of blocks that contain data of the file. Thus, the data for small files (of no more than 12 blocks) do not need a separate index block. If the block size is 4 KB, then up to 48 KB of data can be accessed directly. The next three pointers point to indirect blocks. The first points to a single indirect block, which is an index block containing not data but the addresses of blocks that do contain data. The second points to a double indirect block, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer contains the address of a triple indirect block.

A UNIX inode is shown in Figure 11.9.



**VARIOUS DISK SCHEDULING ALGORITHMS**

**Seek Time:** is the time for the disk arm to move the heads to the cylinder containing the desired sector.

**Rotational Latency:** is the additional time for the disk to rotate the desired sector to the disk head.

**Access Time:** is the sum of seek time and rotational latency.

**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 managing the order in which disk I/O requests are serviced. Whenever a process needs I/0 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?

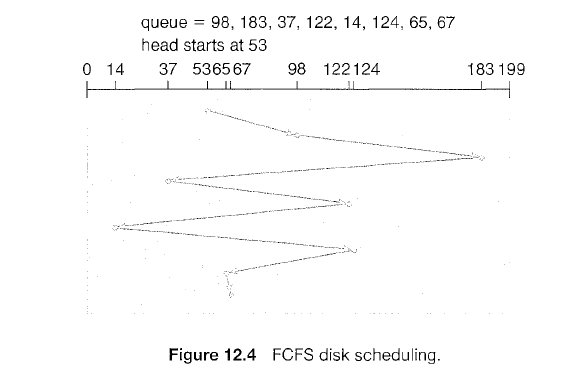
What the disk address for the transfer is?

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. When one request is completed, the operating system chooses which pending request to service next. How does the operating system make this choice? Any one of several disk-scheduling algorithms can be used to resolve such issues.

**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/0 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. This schedule is diagrammed in Figure 12.4.

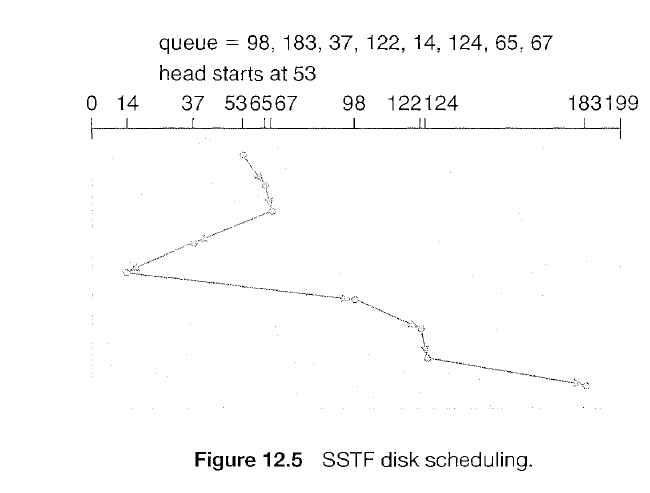
The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders 37 and 14 could be serviced together, before or after the requests for 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

**SSTF SCHEDULING**

It seems reasonable to service all the requests close to the current head position before moving the head far away 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.

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 12.5). 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. Clearly, 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 the request from 14 is being serviced, a new request near 14 arrives. This new request will be serviced next, making the request at 186 wait. In theory, a continual stream of requests near one another could cause the request for cylinder 186 to wait indefinitely.



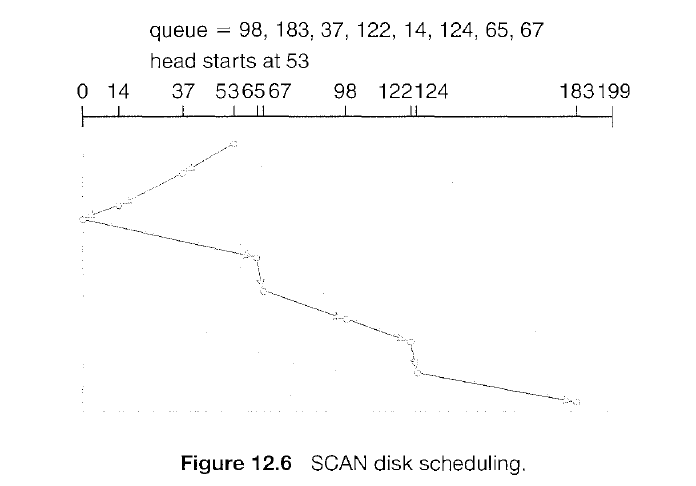
Although the SSTF algorithm is a substantial improvement over the FCFS algorithm, it is not optimal. In the example, we can do better by moving the head from 53 to 37, even though the latter is not closest, and then to 14, before turning around to service 65, 67, 98, 122, 124, and 183. This strategy reduces the total head movement to 208 cylinders.

**SCAN SCHEDULING**

In the SCAN algorithm, the disk arm starts at one end of the disk and moves toward the 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.

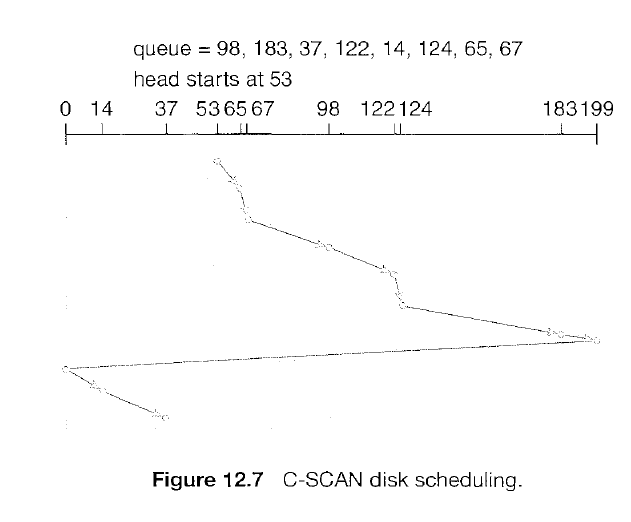
Let's return to our example to illustrate. Before applying SCAN to schedule the requests on cylinders 98, 183,37, 122, 14, 124, 65, and 67, we need to know the direction of head movement in addition to the head's current position.

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 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.



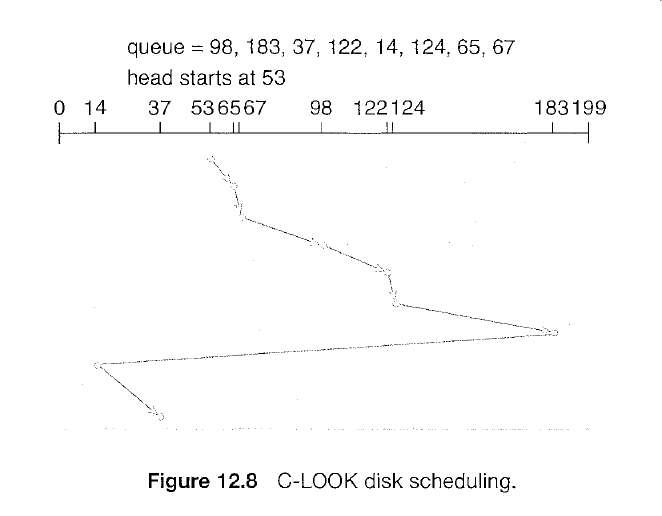
**C-SCAN SCHEDULING**

**Circular Scan (C- SCAN)** disk scheduling is a variant of SCAN. 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 12.7). 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 & C-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 12.8).



**Exercise**

Suppose that a disk drive has 5,000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of pending requests, in FIFO order, is: 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk-scheduling algorithms?

a. FCFS

b. SSTF

c. SCAN

d. LOOK

e. C-SCAN

f. C-LOOK