**MCA Part-II**

**Paper-XIII: Operating System**

**Topic: FILE MANAGEMENT**

**PREPARED BY: DR. KIRAN PANDEY**

**(School of Computer Science)**

**Email-id: kiranpandey.nou@gmail.com**

## IINTRODUCTION

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 nonvolatile, 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. Files may be free form, such as text files, or may be formatted rigidly. 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.

### *File Attributes*

*A* file's attributes vary from one operating system to another but typically consist of these:

**Name:** The symbolic file name is the only information kept in human readable form.

**Identifier**: This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.

**Type**: This information is needed for systems that support different types of files.

**Location:** This information is a pointer to a device and to the location of the file on that device.

**Size:** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.

**Protection:** Access-control information determines who can do reading, writing, executing, and so on.

**Time, date, and user identification:** This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

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.

### *File Operations*

To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. Let's examine what the operating system must do to perform each of these 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. 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. 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/O. This file operation is also known as a file *seek.*

**Deleting a file:** To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.

**Truncating a file:** The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged –except for file length-but lets the file be reset to length zero and its file space released.

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.

### *File Type*

When we design a file system-indeed, an entire operating system-we always consider whether the operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways. 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 (Table 1). 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 |

**Table 1: Common types of files**

***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. Some operating systems extend this idea into a set of system-supported file structures, with sets of special operations for manipulating files with those structures. For instance, DEC's VMS operating system has a file system that supports three defined file structures.

This point brings us to one of the disadvantages of having the operating system support multiple file structures: the resulting size of the operating system is cumbersome. If the operating system defines five different file structures, it needs to contain the code to support these file structures. In addition, it may be necessary to define every file as one of the file types supported by the operating system. When new applications require information structured in ways not supported by the operating system, severe problems may result.

Some operating systems impose (and support) a minimal number of file structures. This approach has been adopted in UNIX, MS-DOS, and others. UN1X considers each file to be a sequence of 8-bit bytes; no interpretation of these bits is made by the operating system. This scheme provides maximum flexibility but little support. Each application program must include its own code to interpret an input file as to the appropriate structure. However, 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.

It is useful for an operating system to support structures that will be used frequently and that will save the programmer substantial effort. Too few structures make programming inconvenient, whereas too many cause operating-system bloat and programmer confusion.

### *File Access Mechanisms*

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. There are several ways to access files –

**Sequential access**: A sequential access is that in which the records are accessed in some sequence, i.e., the information in the file is processed in order, one record after the other. This access method is the most primitive one. Example: Compilers usually access files in this fashion.

**Direct/Random 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.

As a simple example, on an airline-reservation system, we might store all the information about a particular flight (for example, flight 615) in the block identified by the flight number. Thus, the number of available seats for flight 615 is stored in block 615 of the reservation file. To store information about a larger set such as people, we might compute a hash function on the people's names or search a small in-memory index to determine a block to read and search.

**Indexed Sequential Access Method (ISAM):** It 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.

### *File Related System Services*

In this section we briefly describe the system services, which relate to file management. We can broadly classify these under categories:

i) Online services

ii) Programming services.

**Online-services:** Most operating systems provide interactive facilities to enable the on-line users to work with files. Few of these facilities are built-in commands of the system while others are provided by separate utility programs. Many such services provided by the operating system related to directory operations are listed below:

- Create a file
- Delete a file
- Copy a file
- Rename a file
- Display a file
- Create a directory
- Remove an empty directory
- List the contents of a directory
- Search for a file
- Traverse the file system.

**Programming services:** The complexity of the file services offered by the operating system vary from one operating system to another but the basic set of operations like: open (make the file ready for processing), close (make a file unavailable for

processing), read (input data from the file), write (output data to the file), seek (select a position in file for data transfer).

In addition to file functions described above the operating system must provide directory operation support also like:

- Create or remove directory
- Change directory
- Read a directory entry
- Change a directory entry etc.

These are not always implemented in a high level language but language can be supplied with these procedure libraries. For example, UNIX uses C language as system programming language, so that all system calls requests are implemented as C procedures.

### *Synchronous and Asynchronous I/O*

Synchronous I/O is based on blocking concept while asynchronous is interrupt -driven transfer. If a user program is doing several things simultaneously and request for I/O operation, two possibilities arise. The simplest one is that the I/O is started, then after its completion, control is transferred back to the user process. This is known as synchronous I/O where you make an I/O request and you have to wait for it to finish. This could be a problem where you would like to do some background processing and wait for a key press. Asynchronous I/O solves this problem, which is the second possibility. In this, control is returned back to the user program without waiting for the I/O completion. The I/O then continues while other system operations occur. The CPU starts the transfer and goes off to do something else until the interrupt arrives.

Asynchronous I/O operations run in the background and do not block user applications. This improves performance, because I/O operations and applications processing can run simultaneously. Many applications, such as databases and file servers, take advantage of the ability to overlap processing and I/O. To manage asynchronous I/O, each asynchronous I/O request has a corresponding control block in the application's address space. This control block contains the control and status information for the request. It can be used again when the I/O operation is completed. Most physical I/O is asynchronous.

After issuing an asynchronous I/O request, the user application can determine when and how the I/O operation is completed. This information is provided in any of three ways:

- The application can poll the status of the I/O operation.

- The system can asynchronously notify the application when the I/O operation is done.
- The application can block until the I/O operation is complete.

Each I/O is handled by using a device-status table. This table holds entry for each I/O device indicating device's type, address, and its current status like bust or idle. When any I/O device needs service, it interrupts the operating system. After determining the device, the Operating System checks its status and modifies table entry reflecting the interrupt occurrence. Control is then returned back to the user process.

## DIRECTORIES

The directory can be viewed as a symbol table that translates file names into their directory entries. If we take such a view, we see that the directory itself can be organized in many ways. 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. When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory. The operations performed on a directory or file system are:

1) Create, delete and modify files.
2) Search for a file.
7) Mechanisms for sharing files should provide controlled access like read, write, execute or various combinations.
4) List the files in a directory and also contents of the directory entry.
5) Renaming a file when its contents or uses change or file position needs to be changed.
6) Backup and recovery capabilities must be provided to prevent accidental loss or malicious destruction of information.
7) Traverse the file system.

The most common schemes for describing logical directory structure are:

### (i) Single-level directory

In Single Level Directory all files are in the same directory.

#### *Limitations of Single Level Directory*

a) since all files are in the same directory, they must have unique name.

b) If two user call their data free test, then the unique name rule is violated.

c) Files are limited in length.

d) Even a single user may find it difficult to remember the names of all files as the number of file increases.

e) Keeping track of so many file is daunting task.

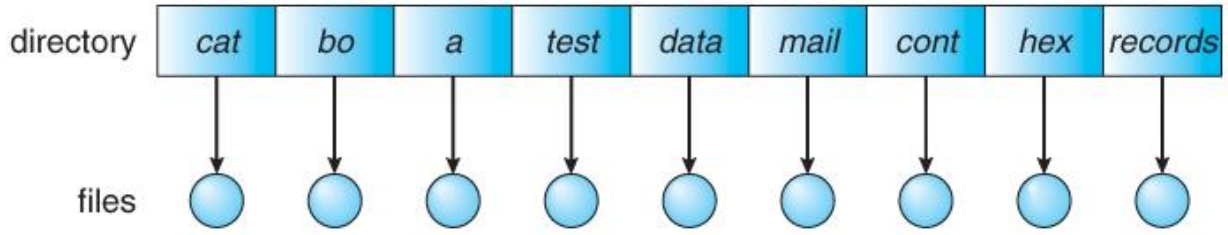



**Figure 1: Single-level directory**

### (ii) Two-level directory

Each user has its own User File Directory (UFD). When the user job start or user log in, the system Master File Directory (MFD) is searched. MFD is indexed by user name or Account Number. When user refers to a particular file, only his own UFD is searched. Thus different users may have files with same name. To have a particular file uniquely, in a two level directory, we must give both the user name and file name.

*A two level directory can be a tree or an inverted tree of height 2*

The root of a tree is Master File Directory (MFD). Its direct descendants are User File Directory (UFD). The descendants of UFD's are file themselves. The files are the leaves of the tree.

### *Limitations of Two Level Directory*

The structure effectively isolates one user from another.

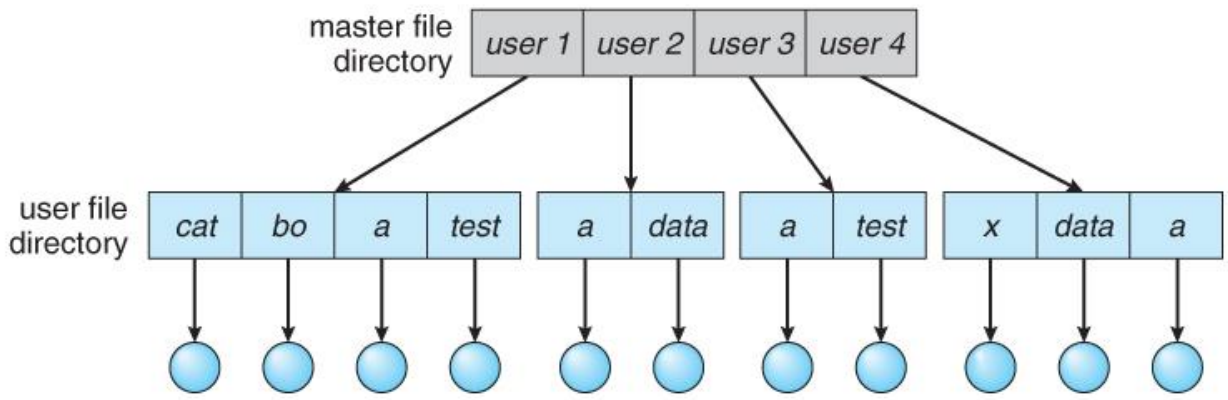



**Figure 2: Two-level directory**

### (iii) Tree-structured directory

A directory (or Sub directory) contains a set of files or sub directories. All directories has the same internal format. One bit in each directory entry defines the entry. Special calls are used to create and delete directories. Each process has a current directory. Current directory should contain most of the files that are of current interest to the process. When a reference is made to a file, the current directory is searched. The user can change his current directory whenever he desires. If a file is not needed in the current directory then the user usually must either specify a path name or change the current directory. **Paths** can be of two types :-

a) *Absolute Path*
Begins at root and follows a path down to the specified file.

b) *Relative Path*
Defines a path from current directory.

Deletions if directory is empty, its entry in the directory that contains it can simply deleted. If it is not empty: One of the Two approaches can be taken :-
a) User must delete all the files in the directory.

b) If any sub directories exist, same procedure must be applied. The **UNIX rm command** is used. **MS dos** will not delete a directory unless it is empty.

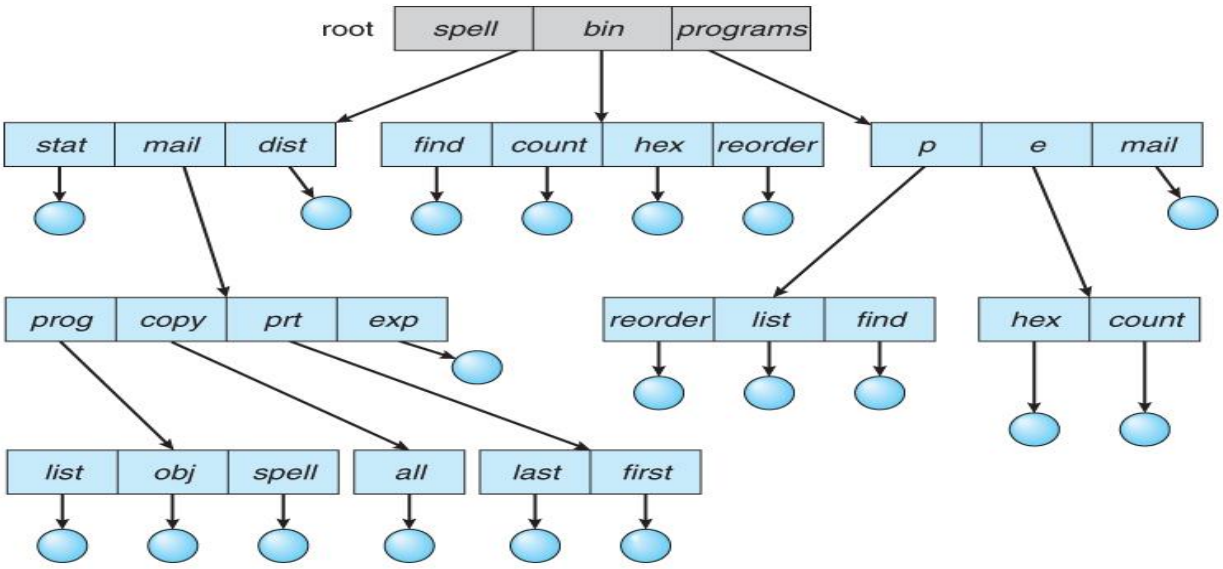



**Figure 3: Tree-structured directory**

### (iv) Acyclic-graph directory:

Acyclic Graph is the graph with no cycles. It allows directories to share sub directories and files. With a shared file, only one actual file exists, so any changes made by one person are immediately visible to another.

### *Implementation of Shared files and directories*

i) *To create a link*

. A link is effectively a pointer to another file or sub directory.

. Duplicate all the information about them in both sharing directories.

ii) *Deleting a link*

. Deletion of the link will not affect the original file, only link is removed.

. To preserve the file until all references to it are deleted.

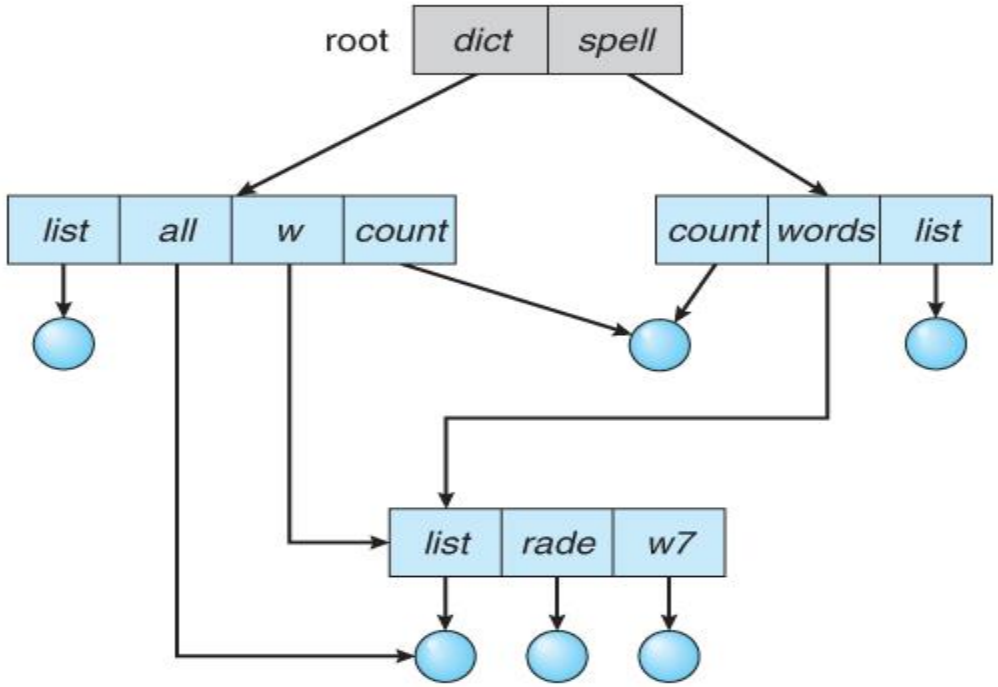



**Figure 4: Acyclic-graph directory**

The limitations of this approach are the difficulties in traversing an entire file system because of multiple absolute path names. Another issue is the presence of dangling pointers to the files that are already deleted, though we can overcome this by preserving the file until all references to it are deleted. For this, every time a link or a copy of directory is established, a new entry is added to the file-reference list. But in reality as the list is too lengthy, only a count of the number of references is kept. This count is then incremented or decremented when the reference to the file is added or it is deleted respectively.

### (v) General graph Directory:

Acyclic-graph does not allow cycles in it. However, when cycles exist, the reference count may be non-zero, even when the directory or file is not referenced anymore. In such situation garbage collection is useful. This scheme requires the traversal of the whole file system and marking accessible entries only. The second pass then collects everything that is unmarked on a free-space list. This is depicted in Figure .

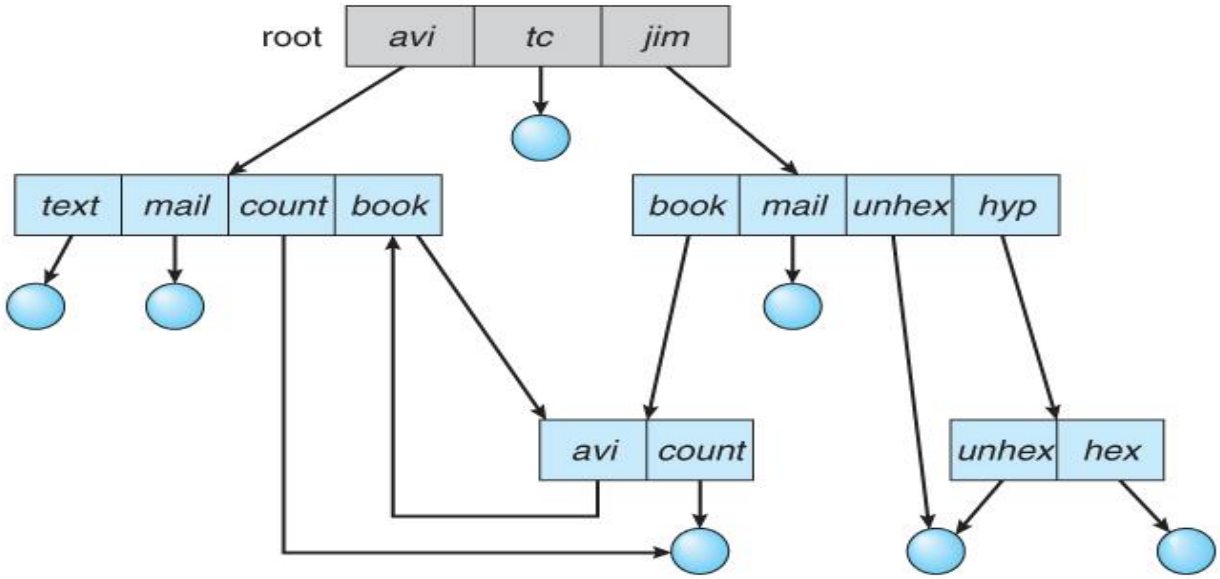



**Figure 5: General-graph directory**

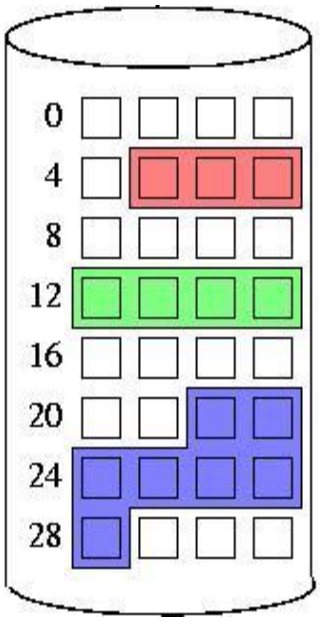
## DISK SPACE MANAGEMENT

Files are allocated disk spaces by operating system. Operating systems deploy following three main ways to allocate disk space to files.

- Contiguous Allocation
- Linked Allocation
- Indexed Allocation

### *Contiguous Allocation*

- Each file occupies a contiguous address space on disk.
- Assigned disk address is in linear order.
- Easy to implement.
- External fragmentation is a major issue with this type of allocation technique.
- A repacking routine called compaction can solve this problem.




Directory:

| file | start | length |
|---|---|---|
| moo | 5 | 3 |
| snow | 22 | 7 |
| fall | 12 | 4 |

**Figure 6: Contiguous Allocation on the Disk**

In compaction, an entire file system is copied on to tape or another disk and the original disk is then freed completely. Then from the copied disk, files are again stored back using contiguous space on the original disk. But this approach can be very expensive in terms of time. Also, size-declaration in advance is a problem because each time, the size of file is not predictable. But it supports both sequential and direct accessing. For sequential access, almost no seeks are required. Even direct access with seek and read is fast. Also, calculation of blocks holding data is quick and easy as we need just offset from the start of the file.

### *Linked Allocation or chaining*

All files are stored in fixed size blocks and adjacent blocks are linked together like a linked list. The disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last block of the file. Also each block contains pointers to the next block, which are not made available to the user. There is no external fragmentation in this as any free block can be utilised for storage. So, compaction and relocation is not required. But the disadvantage here is that it is potentially inefficient for direct-accessible files since blocks are scattered over the disk and have to follow pointers from one disk block to the next. It can be used effectively for sequential access only but there also it may generate long seeks between blocks. Another issue is the extra storage space required for pointers. Yet the reliability problem is also there due to loss/damage of any pointer. The use of doubly linked lists could be a solution to this problem but it would add more overheads for each file. A doubly linked list also facilitates searching as blocks are threaded both forward and backward. The Figure 14

depicts linked /chained allocation where each block contains the information about the next block (i.e., pointer to next block). We can summaries as:

- Each file carries a list of links to disk blocks.
- Directory contains link / pointer to first block of a file.
- No external fragmentation
- Effectively used in sequential access file.
- Inefficient in case of direct access file.

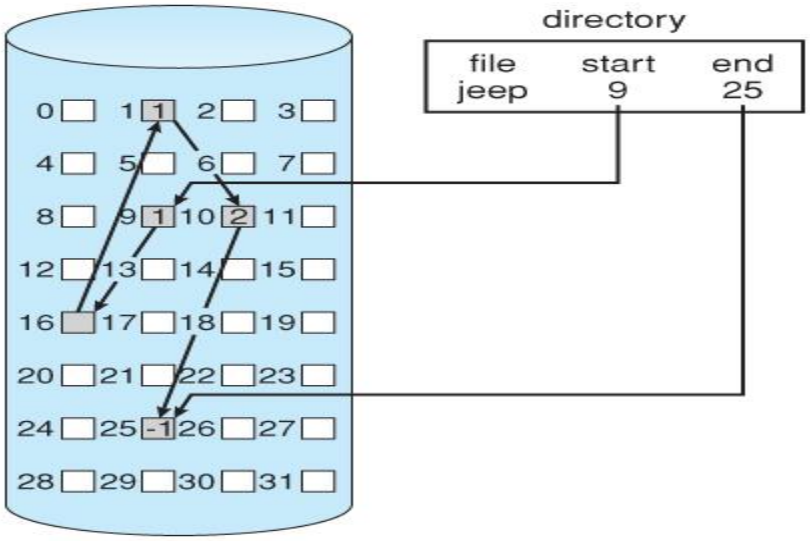



**Figure 7: Linked Allocation on the Disk**

### *Indexed Allocation*

In this each file has its own index block. Each entry of the index points to the disk blocks containing actual file data i.e., the index keeps an array of block pointers for each file. So, index block is an array of disk block addresses. The ith entry in the index block points to the ith block of the file. Also, the main directory contains the address where the index block is on the disk. Initially, all the pointers in index block are set to NIL. The advantage of this scheme is that it supports both sequential and random access. The searching may take place in index blocks themselves. The index blocks may be kept close together in secondary storage to minimize seek time. Also space is wasted only on the index which is not very large and there's no external fragmentation. But a few limitations of the previous scheme still exists in this, like, we still need to set maximum file length and we can have overflow scheme of the file larger than the

predicted value. Insertions can require complete reconstruction of index blocks also**.** We can summaries as :

- Provides solutions to problems of contiguous and linked allocation.
- A index block is created having all pointers to files.
- Each file has its own index block which stores the addresses of disk space occupied by the file.
- Directory contains the addresses of index blocks of files.

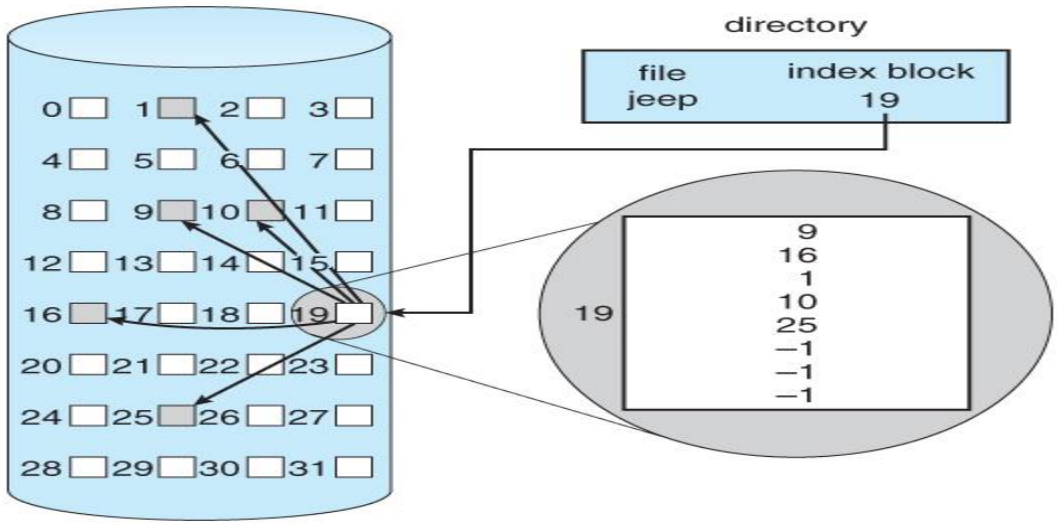



**Figure 8: Indexed Allocation on the Disk**