UNIT - V

File System Interface and Operations

Definition of a File:

A file is a named collection of related information that is recorded on secondary storage.

(or)A file is the smallest allotment of logical secondary storage.

(or)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.

File Attributes

File Attributes gives the Operating System information about the file and how it is intended to use.

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.

Some newer file systems also support **extended file attributes**, including character encoding of the file and security features such as a file checksum.

File Types

When we design a file 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. Examples include resume.docx, server.c, and ReaderThread.cpp.

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Common file types.

File Structure

File types also can be used to indicate the internal structure of the file. Some operating systems extend this idea by supporting their own file structures. But it has the following disadvantages

1. If operating system support multiple file structures: the resulting size of the operating system is large.
2. Some applications may require information structured in a way that is not supported by the OS some operating systems impose (and support) a minimal number of file structures. This approach has been adopted in UNIX, Windows, and others.

Internal File Structure

Block Structure

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

Record Structure

Files contain a sequence of fixed length records. Physical records may or may not get exact match with the logical record. Logical records even vary in length.

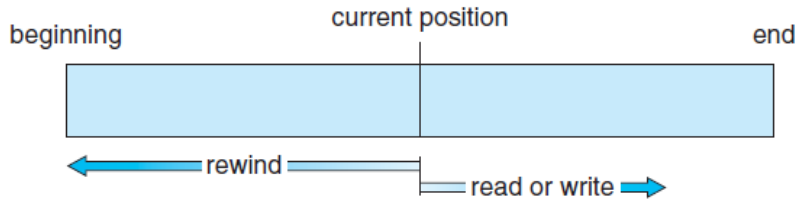
Access methods

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 the following ways,

1. Sequential Access

- The simplest access method is **sequential access**. Information in the file is processed in order, one record after the other. It is based on a tape model of a file and works as well on sequential-access devices.

- **Example**: Editors and Compilers usually access files in this fashion.
- **Operations**
  - 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). On some systems, a program may be able to skip forward or backward *n* records for some integer *n*—perhaps only for *n* = 1.



Sequential-access file.

2. **Direct Access** (or **Relative Access**)
   - Another method is **direct access** (or **relative access**). Here, a file is made up of fixed-length **logical records** 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.
   - **Examples**:
     - 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.
     - On an airline-reservation system, we might store all the information about a particular flight (for example, flight 713) in the block identified by the flight number. Thus, the number of available seats for flight 713 is stored in block 713 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.
   - **Operations**
     - For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have read (n), where *n* is the block number, rather than read next (), and write (n) rather than write next ().
     - An alternative approach is to retain read next () and write next (), as with sequential access, and to add an operation position file (n) where *n* is the block number. Then, to affect a read (n), we would position file (n) and then read next ().

3. **Indexed Access**
   - It involves 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.
   - **Example**:
     A retail-price file might list the universal product codes (UPCs) for items, with the associated prices. Each record consists of a 10-digit UPC and a 6-digit price, for a 16-byte record. If our disk has 1,024 bytes per block, we can store 64 records per block. A file of 120,000 records would occupy about 2,000 blocks (2 million bytes). By keeping the file sorted by UPC, we can define an index consisting of the first UPC in each block. This index would have 2,000 entries of 10 digits each, or 20,000 bytes, and thus could be kept in memory. To find the price of a particular item, we can make a binary search of the index. 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/O.

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 contains pointers to secondary index files, which 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.

Directory Overview

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**. 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 (or **directory**) records information—such as name, location, size, and type—for all files on that volume.

The directory can be viewed as a symbol table that translates file names into their directory entries. The following are the operations that are to be performed on a directory:

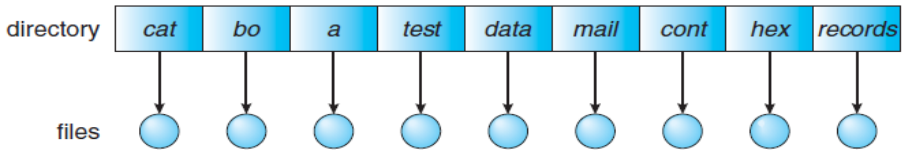
- **Search for a file**.
- **Create a file**
- **Delete a file**.
- **List a directory**.
- **Rename a file**
- **Traverse the file system**

Directory Structure

The most common schemes for defining the logical structure of a directory are the following,

1. **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.

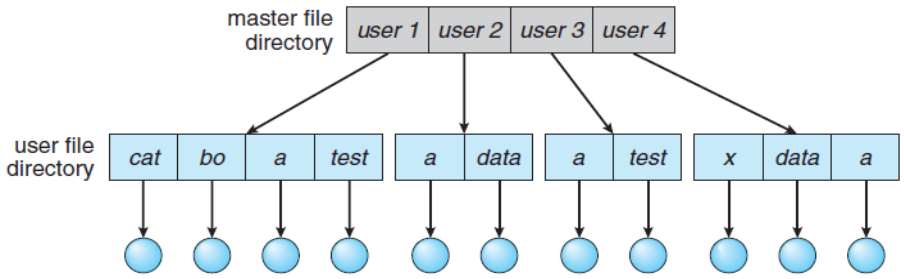
- **Limitations**
  - All files are in the same directory, they must have unique names. If two users call their data file test.txt, then the unique-name rule is violated.
  - Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases. Keeping track of so many files is a problem.



Single-level directory.

2. Two-Level Directory

- The standard solution to eliminate confusion of file names among different users is to create a separate directory for each user.
- So the two level directory structure contains 2 directories
  - Master File Directory (MFD) at the top level.
  - User File Directory (UFD) at the second level and
  - Actual files are at the third level.
- Each user has his own **user file directory (UFD).** When a user job starts or a user logs in, the system's **master file directory (MFD)** is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user
- When a user refers to a particular file, only his own UFD is searched.
- 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.



Two-level directory structure.

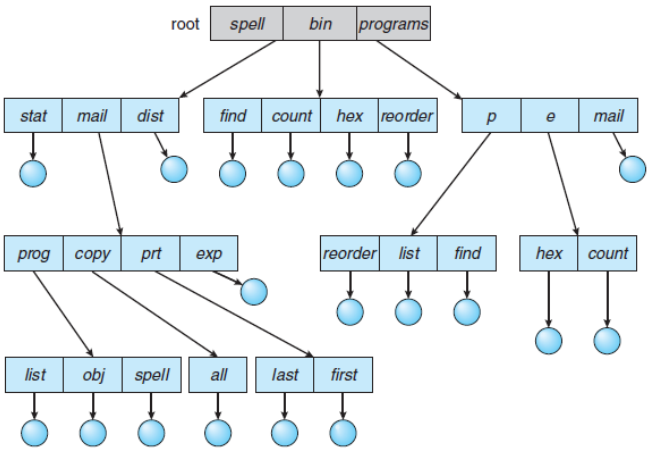
- Although the two-level directory structure solves the name-collision problem, it still has disadvantages.
- This structure effectively isolates one user from another. Isolation is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another's files. Some systems simply do not allow local user files to be accessed by other users.
- If access is permitted one user must have the ability to name a file in another user's

directory. To name a file uniquely, the user must give both user name and file name as Path Name.

- If user A wishes to access her own test file named test.txt, she can simply refer to test.txt. To access the file named test.txt of user B (with directory-entry name userb), however, she might have to refer to /userb/test.txt(windows os) and /u/pbg/test(Unix, Linux).
- A special situation occurs with the system files. If a user wants them, they are searched in USD if found ok if not found we should copy the system files into each UFD but copying all the system files would waste an enormous amount of space.
- The standard solution is to use special user directory. Whenever a file name is given to be loaded, the operating system first searches the local UFD. If the file is found, it is used. If it is not found, the system automatically searches the special user directory that contains the system files.
- The sequence of directories searched when a file is named is called the **search path**.

3. Tree-Structured Directories

- 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.
- 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.
- **Current Directory**
  - 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 (using change directory () system call) to be the directory holding that file.



Tree-structured directory structure.

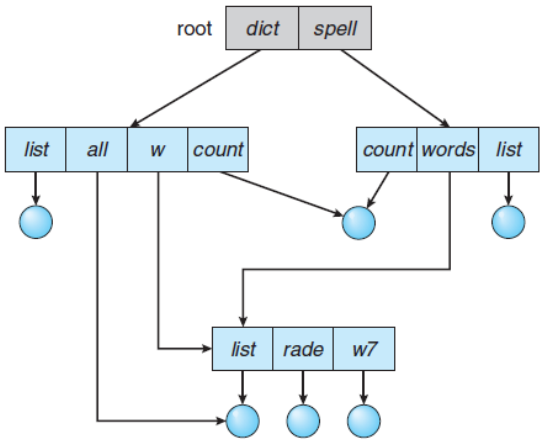
- **Path Names**
  - It describes the path the OS must take to get to some point.
  - Path names can be of two types: absolute and relative.

- 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.
- 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.

- **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 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.

4. Acyclic-Graph Directories

- A tree structure prohibits the sharing of files or directories. An **acyclic graph** i.e**.,** a graph with no cycles which allows directories to share subdirectories and files.
- The same file or subdirectory may be in two different directories. An acyclic-graph directory structure is more flexible than a simple tree structure, but it is also more complex.



Acyclic-graph directory structure.

- **Implementation**
  - **a. Link:** A common way is to create a new directory entry called a link. A **link** is effectively a pointer to another file or subdirectory. A link may be implemented as an absolute or a relative path name. When a reference to a file is made, we search the directory. If the directory entry is marked as a link, 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.

b. Duplication: Shared files duplicate all information about them in both sharing directories. Thus, both entries are identical and equal. A major problem with duplicate directory entries is maintaining consistency when a file is modified.

- **Problems**
  - A file may now have multiple absolute path names creating problem in traversing.
  - **Deletion**: When can the space allocated to a shared file be deallocated and reused?
    - One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now-nonexistent file.
    - Another possibility occurs when symbolic links are used. The deletion of a link need not affect the original file; only the link is removed. If the file entry itself is deleted, the space for the file is deallocated, leaving the links dangling. We can search for these links and remove them as well, but unless a list of the associated links is kept with each file, this search can be expensive. Alternatively, we can leave the links until an attempt is made to use them. At that time, we can determine that the file of the name given by the link does not exist and can fail to resolve the link name; the access is treated just as with any other illegal file name.
    - Another approach to deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted. We could keep a list of all references to a file (directory entries or symbolic links). When a link or a copy of the directory entry is established, a new entry is added to the file-reference list. When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty.

Protection

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

Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed.

File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism, accidental deletion , Bugs in the file-system software etc.,

Protection can be provided in many ways. For a single-user laptop system, we might provide protection by locking the computer in a desk drawer or file cabinet. In a larger multiuser system, however, other mechanisms are needed.

1. **Types of Access**

Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:

- **Read**. Read from the file.
- **Write**. Write or rewrite the file.
- **Execute**. Load the file into memory and execute it.
- **Append**. Write new information at the end of the file.
- **Delete**. Delete the file and free its space for possible reuse.
- **List**. List the name and attributes of the file.

Other operations, such as renaming, copying, and editing the file, may also be controlled.

2. Access Control

The most common approach to the protection problem is to make access dependent on the identity of the user. Different users may need different types of access to a file or directory. The most general scheme to implement identity dependent access is to associate with each file and directory an **access-control list (ACL)** specifying user names and the types of access allowed for each user.

When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

This approach has the advantage of enabling complex access methodologies. The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access. This technique has two undesirable consequences:

- Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.
- The directory entry, previously of fixed size, now must be of variable size, resulting in more complicated space management.

These problems can be resolved by use of a condensed version of the access list. To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:

- **Owner**. The user who created the file is the owner.
- **Group**. A set of users who are sharing the file and need similar access is a group, or work group.
- **Universe**. All other users in the system constitute the universe.

The most common recent approach is to combine access-control lists with the more general (and easier to implement) owner, group, and universe access control scheme.

For this scheme to work properly, permissions and access lists must be controlled tightly. This control can be accomplished in several ways. For example, in the UNIX system, groups can be created and modified only by the manager of the facility (or by any superuser). Thus, control is achieved through human interaction.

With the more limited protection classification, only three fields are needed to define protection. Often, each field is a collection of bits, and each bit either allows or prevents the access associated with it. For example, the UNIX system defines three fields of 3 bits each—

rwx, where r controls read access, w controls write access, and x controls execution. A separate field is kept for the file owner, for the file's group, and for all other users. In this scheme, 9 bits per file are needed to record protection information.

Example:

19 -rw-r--r--+ 1 jim staff 130 May 25 22:13 file1

3. Other Protection Approaches

Another approach to the protection problem is to associate a password with each file. Just as access to the computer system is often controlled by a password, access to each file can be controlled in the same way. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file. The use of passwords has a few disadvantages, however.

First, the number of passwords that a user needs to remember may become large, making the scheme impractical.

Second, if only one password is used for all the files, then once it is discovered, all files are accessible; protection is on an all-or-none basis. Some systems allow a user to associate a password with a subdirectory, rather than with an individual file, to address this problem.

File System Structure

Disks provide most of the secondary storage on which file systems are maintained. Two characteristics make them convenient for this purpose are,

1. A disk can be rewritten.
2. A disk can access directly any block of information it contains.

File systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily.

Design Problems in a File System

1. How the file system should look to the user i.e. file, and the directory structure for organizing files.
2. Creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

Layered design of a File systems

Each level in the design uses the features of lower levels to create new features for use by higher levels.

Application Programs

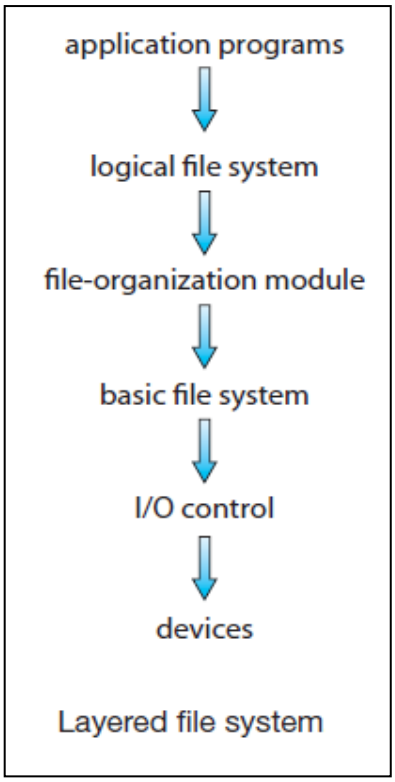
It contains user code that is making a request.

Logical File System

The **logical file system** manages metadata information. Metadata includes all of the file-system structure except the actual data. The logical file system manages the directory structure to provide the file-organization module with this information.

File-Organization Module

The **file-organization module** knows about files and their logical blocks and 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.



Basic File System

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.

I/O control

The **I/O control** level consists of device drivers and interrupts handlers to transfer information between the main memory and the disk system. It acts like a translator, inputting high-level commands such as "retrieve block 123." And outputting low-level, hardware -specific instructions that are used by the hardware controller

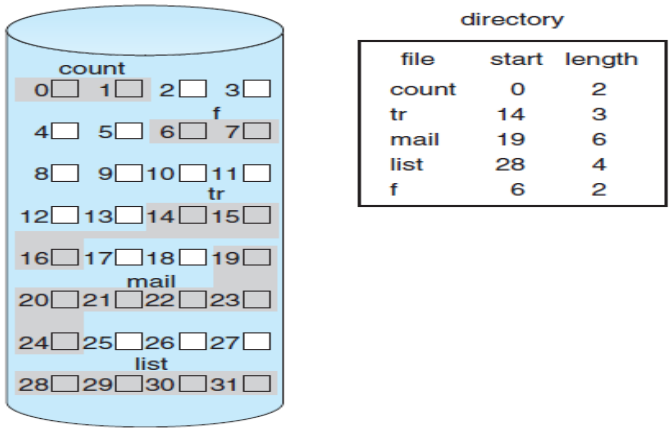
Devices

These are the actual hardware devices like disk.

Allocation methods

Many files can be 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. The following are the three major methods of allocating disk space that are in wide use:

1. **Contiguous Allocation**
   - **Contiguous allocation** requires that each file occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk.
   - 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.
   - **Accessing a file :**
     Accessing a file that has been allocated contiguously is easy. It supports both sequential and random access. 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.*



Contiguous allocation of disk space.

- **Drawbacks**
  - Finding space for a new file. The system chosen to manage free space determine show this task is accomplished. First fit and best fit are the most common strategies used to select a free hole from the set of available holes.
  - **External Fragmentation**

    As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists whenever free space is broken into chunks. It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, none of which is large enough to store the data.
  - **Solution to external fragmentation**

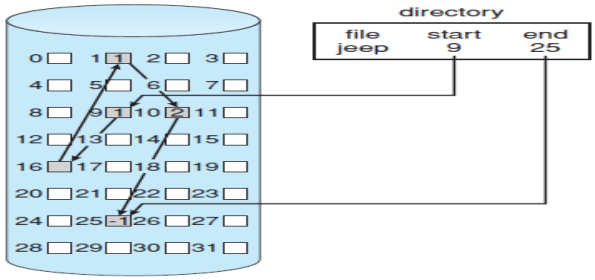
    Copy an entire file system onto another disk. The original disk is then freed completely, creating one large contiguous free space. We then copy the files back onto the original disk by allocating contiguous space from this one large hole. This scheme effectively **compacts** all free space into one contiguous space, solving the fragmentation problem.
  - 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. If we allocate too little space to a file, we may find that the file cannot be extended.

    Two possibilities then exist. First, the user program can be terminated, with an appropriate error message. The user must then allocate more space and run the program again. These repeated runs may be costly. To prevent them, the user will normally overestimate the amount of space needed, resulting in considerable wasted space. The other possibility is to find a larger hole, copy the contents of the file to the new space, and release the previous space. This series of actions can be repeated as long as space exists, although it can be time consuming. The user need never be informed explicitly about what is happening, however; the system continues despite the problem, although more and more slowly. Even if the total amount of space needed for a file is known in advance, preallocation may be inefficient. A file that will grow slowly over a long period.
- **Modified Contiguous-Allocation**
  - 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.

2. Linked Allocation

- **Linked allocation** solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. 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.
- 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 null (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.



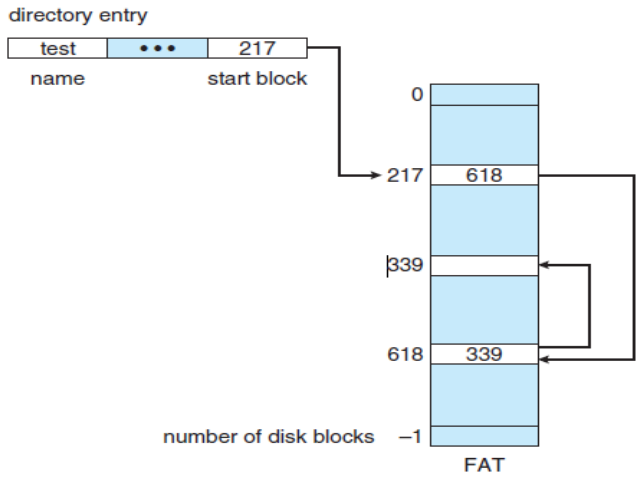
Linked allocation of disk space.

- **Advantages**
  - 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 the 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.
- **Disadvantages**
  - 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.
  - 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.
  - Another problem of linked allocation is reliability the files are linked together by pointers scattered all over the disk, and consider what would happen if a pointer were lost or damaged.

- 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.

Variation on Linked Allocation

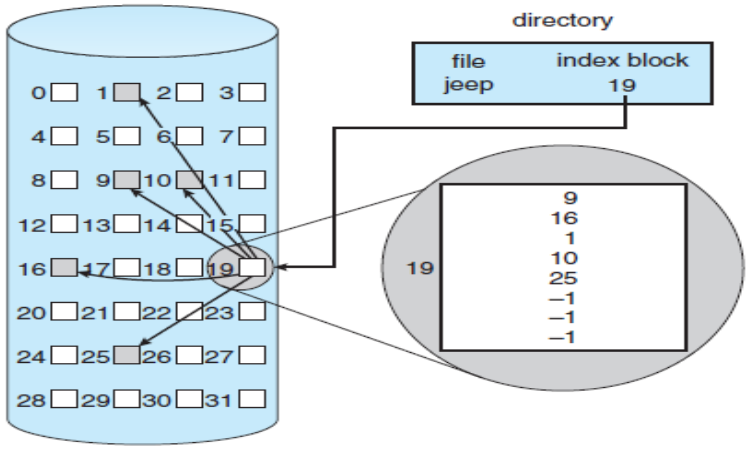
- An important variation on linked allocation is the use of a **file-allocation table (FAT)**. This simple but efficient method of disk-space allocation was used by the MS-DOS operating system.
- A section of disk at the beginning of each volume is set aside to contain the table. The table has one entry for each disk block and is indexed by block number.
- The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number contains the block number of the next block in the file. This chain continues until it reaches the last block, which has a special end-of-file value as the table entry.
- An unused block is indicated by a table value of 0. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-of-file value.



3. 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 bringing all the pointers together into one location: the **index block**.
- Each file has its own index block, which is an array of disk-block addresses. The i^{th} entry in the index block points to the i^{th} block of the file. The directory contains the address of the index block.
- To find and read the i^{th} block, we use the pointer in the i^{th} index-block entry.
- When the file is created, all pointers in the index block are set to null. When the i^{th} block is first written, a block is obtained from the free-space manager, and its address is put in the i^{th} index-block entry.

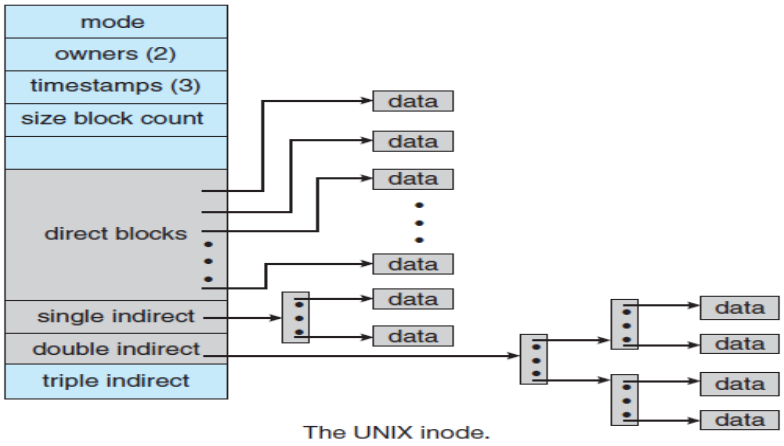
- Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space.



Indexed allocation of disk space.

- **Disadvantages**
  - Indexed allocation does suffer from wasted space.
  - The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.
- **Mechanisms for implementing Index Block**
  - **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 null (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. With 4,096-byte blocks, we could store 1,024 four-byte pointers in an index block. Two levels of indexes allow 1,048,576 data blocks and a file size of up to 4 GB.
  - **Combined scheme**. Another alternative, used in UNIX-based file systems, is to keep the first, say, 15 pointers of the index block in the file's inode. The first 12 of these pointers point to **direct blocks**; that is, they contain addresses of blocks that contain data of the file. 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**.



The UNIX inode.

Free-space Management

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. The following are implementations of free space list.

1. **Bit Vector**
   - Free-space list is frequently 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
     0011110011111100011000000111000000...
   - **Advantage**

     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 a0-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

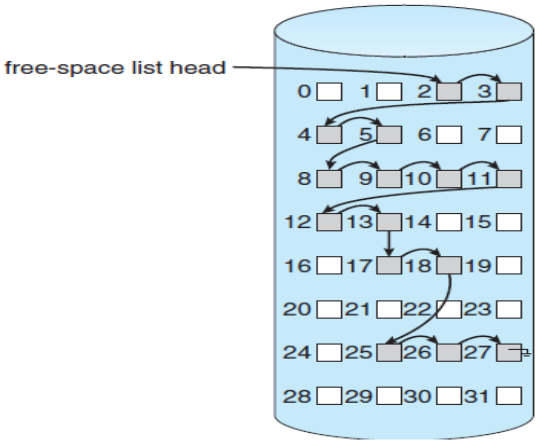
     (number of bits per word) × (number of 0-value words) + offset of first 1 bit.
   - **Disadvantage**

     Bit vectors are inefficient unless the entire vector is kept in main memory (and is written to disk occasionally for recovery needs). Keeping it in main memory is possible for smaller disks but not necessarily for larger ones.

2. **Linked List**
   - Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in 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.

- For example, 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 tom block 8, and so on.
- **Disadvantages**
  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.

3. **Grouping**
   - A modification of the free-list approach stores the addresses of *n* free blocks in the first free block. The first *n*−1 of these blocks are actually free. The last, block contains the addresses of another *n* free block, and so on. The addresses of a large number of free blocks can now be found quickly.

4. **Counting**
   - 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.

5. **Space Maps**
   - Oracle's **ZFS** file system was designed to encompass huge numbers of files, directories, and even file systems.
   - In its management of free space, ZFS creates **metaslabs** to divide the space on the device into chunks of manageable size. Each metaslab has an associated space map.
   - The space map is a log of all block activity (allocating and freeing), in time order, in counting format. When ZFS decides to allocate or free space from a metaslab, it loads the associated space map into memory in a balanced-tree structure (for very efficient operation), indexed by offset, and replays the login to that structure.

- The in-memory spacemap is then an accurate representation of the allocated and free space in the metaslab.

System calls for file operations - open (), read (), write (), close (), seek (), unlink () (File Operations)

create ()

This is used to create 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.

open ()

Many systems require that an open () system call be made before a file is first used. When a file has been opened its entry is added in the open file table. It also contains open count associated with each file to indicate how many processes have the file open.

read ()

To read from a file, we use a system call that specifies the name of the file and **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.

write ()

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.

close ()

This closes a file. Each close () decrements the open count and when the count reaches zero, the file is no longer in use so it can be closed.

delete ()

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.

truncate ()

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.

seek ()

It is also called as Reposition. 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.

unlink ()

Deletes a name from the file system. If that name was the last link to a file and no processes have the file open the file is deleted and the space it was using is made available for reuse.