File Access Methods in Operating System

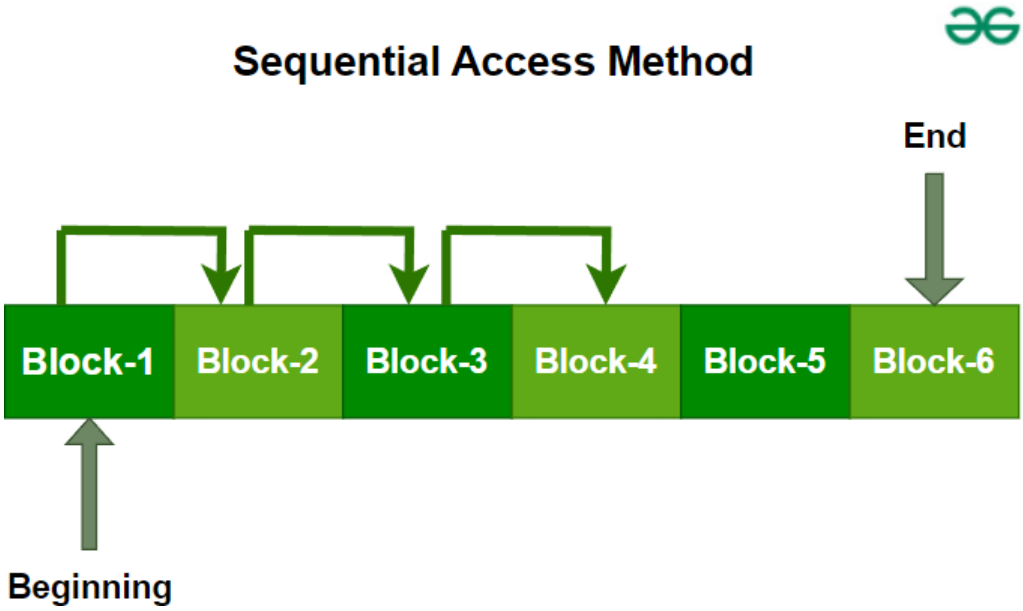
File access methods are techniques used by an OS to read and write data in files. They define how information is organized, retrieved, and modified. Choosing the right method is important for performance and data management.

There are three ways to access a file in a computer system:

- Sequential-Access
- Direct Access
- Index sequential Method

Sequential Access

A file access method where data is read or written **in order, one record after another**, starting from the beginning. The file pointer moves forward automatically after each operation.



Sequential Access Method

Key Points related to Sequential Access

- Data is accessed from one record right after another record in an order.
- When we use the read command, it moves ahead pointer by one.
- When we use the write command, it will allocate memory and move the pointer to

the end of the file.

- Such a method is reasonable for tape.

Advantages of Sequential Access Method

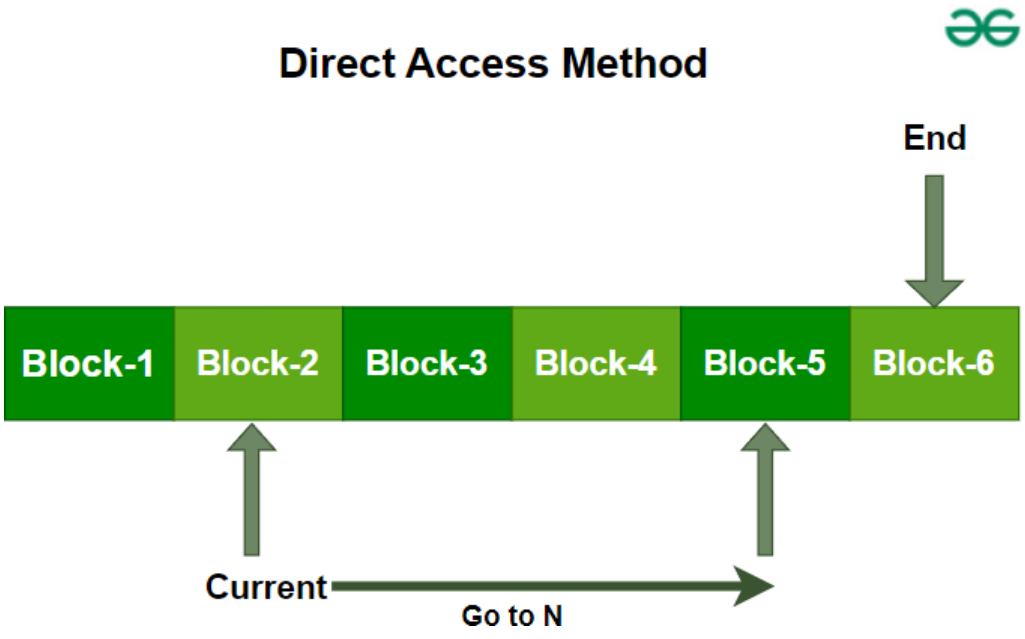
- It is simple to implement this file access mechanism.
- Data is accessed sequentially in the order it is stored.
- It is less prone to data corruption as the data is written sequentially and not randomly.

Disadvantages of Sequential Access Method

- Slow for searching specific records.
- Inserting/updating in the middle is difficult.
- Can waste storage if records have varying lengths.

Direct Access Method

A file access method that allows data to be **read or written directly at any block or record**, using its address (block number). It supports **random access** without scanning previous records.



Direct Access Method

Advantages of Direct Access Method

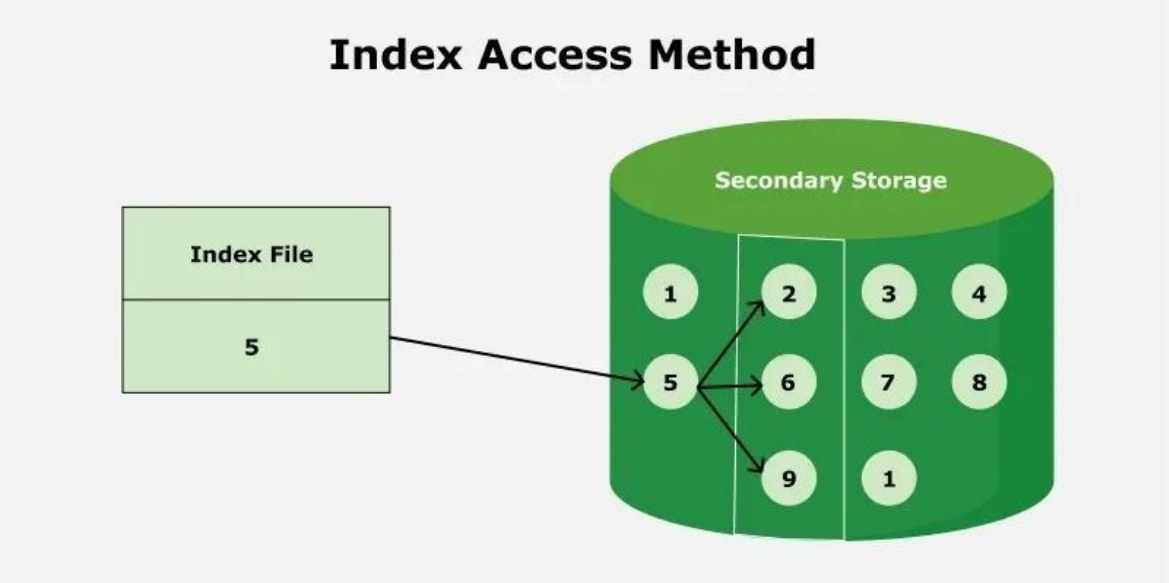
- The files can be immediately accessed decreasing the average access time.
- In the direct access method, in order to access a block, there is no need of traversing all the blocks present before it.

Disadvantages of Direct Access Method

- **Complex Implementation** : Implementing direct access can be complex, requiring sophisticated algorithms and data structures to manage and locate records efficiently.
- **Higher Storage Overhead** : Direct access methods often require additional storage for maintaining data location information (such as pointers or address tables), which can increase the overall storage requirements.

Index Sequential method

It is the other method of accessing a file that is built on the top of the sequential access method. These methods construct an index for the file. The index, like an index in the back of a book, contains the pointer to the various blocks. To find a record in the file, we first search the index, and then by the help of pointer we access the file directly.



Index Access Method

Key Points Related to Index Sequential Method

- It is built on top of Sequential access.

- It control the pointer by using index.

Advantages of Index Sequential Method

- **Fast Searching**: Index enables quick lookups.
- **Flexible:** Supports both sequential and random access.
- **Reduced Access Time:** Quickly locates data in large files.

Disadvantages of Index Sequential Method

- **Complexity:** Harder to implement and maintain than sequential access.
- **Extra Storage:** Requires additional space for indexes.
- **Slower Updates:** Both data and index must be updated during insertions, deletions, or modifications.
- **Maintenance Overhead:** Indexes need frequent updates in dynamic environments.

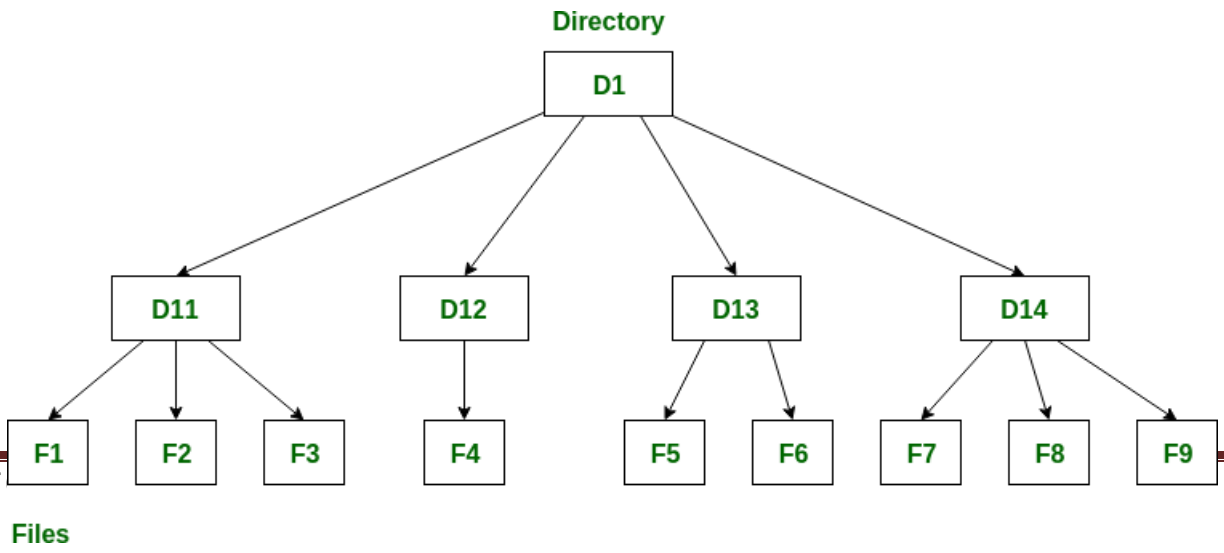
DIRECTORY STRUCTURE:

Structures of Directory in Operating System

The operating system uses directories to track where files are stored, just like using folders to organise papers.

- Different directory structures can be used to suit various organisational needs.
- Understanding directory structures helps in organising and accessing files more easily.

By using these structures, it becomes simpler to manage and navigate files on your computer.



Different Types of Directories in OS

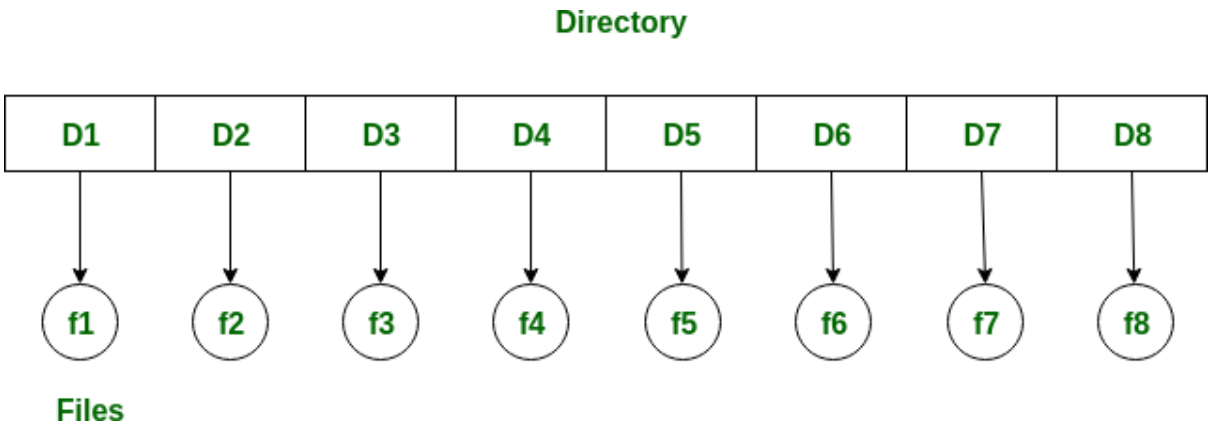
In an operating system, there are different types of directory structures that help organise and manage files efficiently. Each type of directory has its own way of arranging files and directories, offering unique benefits and features. These are

- Single-Level Directory
- Two-Level Directory
- Tree Structure/ Hierarchical Structure
- Acyclic Graph Structure
- General-Graph Directory Structure

1) Single-Level Directory

The single-level directory is the **simplest directory structure**. In it, all files are contained in the same directory which makes it easy to support and understand.

A single level directory has a significant limitation, however, when the number of files increases or when the system has more than one user. Since all the files are in the same directory, they must have a **unique name**. If two users call their dataset test, then the unique name rule violated.



- Since it is a single directory, so its implementation is very easy.
- If the files are smaller in size, searching will become faster.
- The operations like file creation, searching, deletion, updating are very easy in such a directory structure.

Advantages

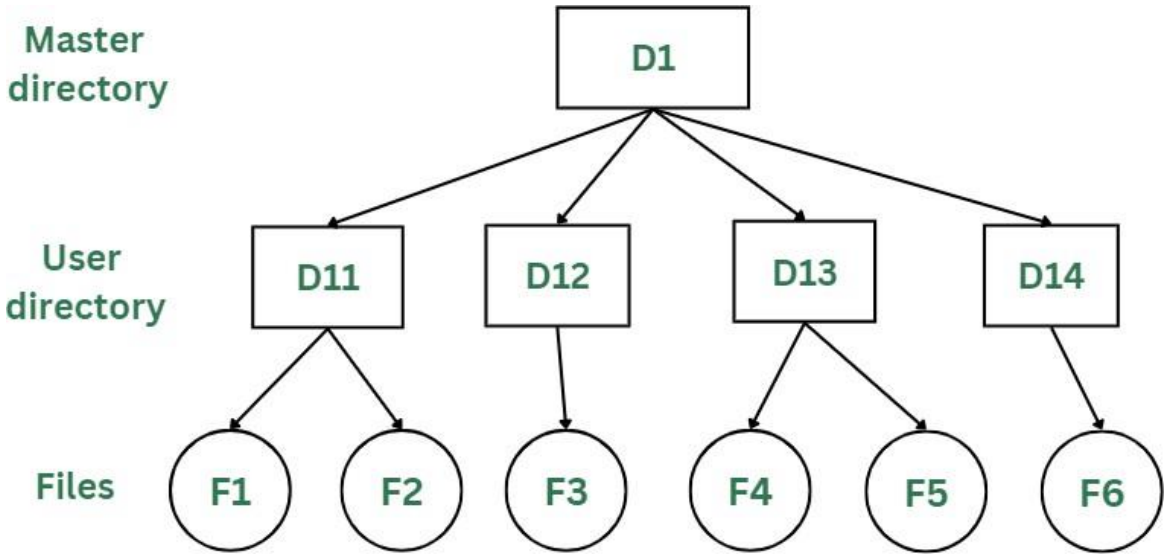
- **Logical Organization:** .Simple to implement and suitable for small systems.
- **Efficiency:** Faster file searching and access.
- **Security:** Restrict access at directory level to protect data.
- **Backup & Recovery:** Simplifies locating and restoring files.
- **Scalability:** Easily supports growth with new files and directories

Disadvantages

- There may chance of name collision because two files can have the same name.
- Searching will become time taking if the directory is large.
- This can not group the same type of files together.

2) Two-Level Directory

In a two-level directory structure, each user has a separate User File Directory (UFD) containing only their files. A Master File Directory (MFD) stores entries for all users and points to their respective UFDs, preventing filename conflicts between users.



Two-Levels Directory Structure

Advantages

- Different users can have files with the same name.

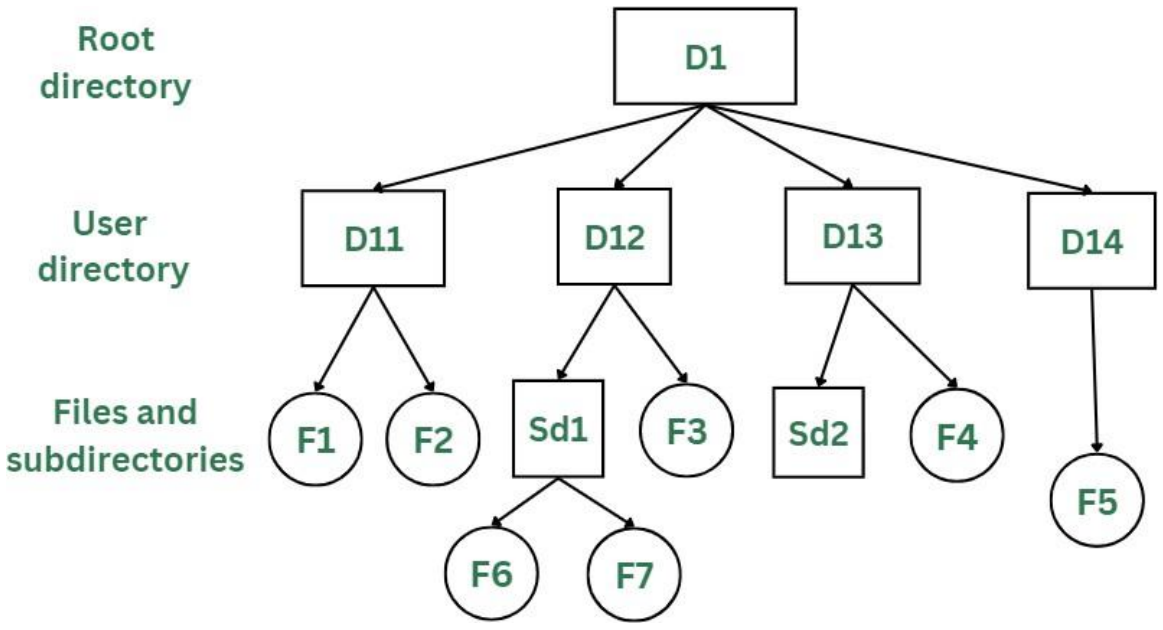
- A security would be there which would prevent user to access other user's files.
- Searching of the files becomes very easy in this directory structure.

Disadvantages

- File sharing is limited and less convenient between users.
- Unlike the advantage users can create their own files, users don't have the ability to create subdirectories.
- Scalability is not possible because one user can't group the same types of files together.

3) Tree Structure/ Hierarchical Structure

The tree directory structure is the most common in personal computers. It resembles an upside-down tree, with the root directory at the top containing all user directories. Each user can create files and subdirectories within their own directory but cannot access or modify the root or other users' directories.



Tree/Hierarchical Directory Structure

Advantages

- This directory structure allows subdirectories inside a directory.
- The searching is easier.

- It helps in organizing files by importance, making important files easier to locate.
- This directory is more scalable than the other two directory structures explained.

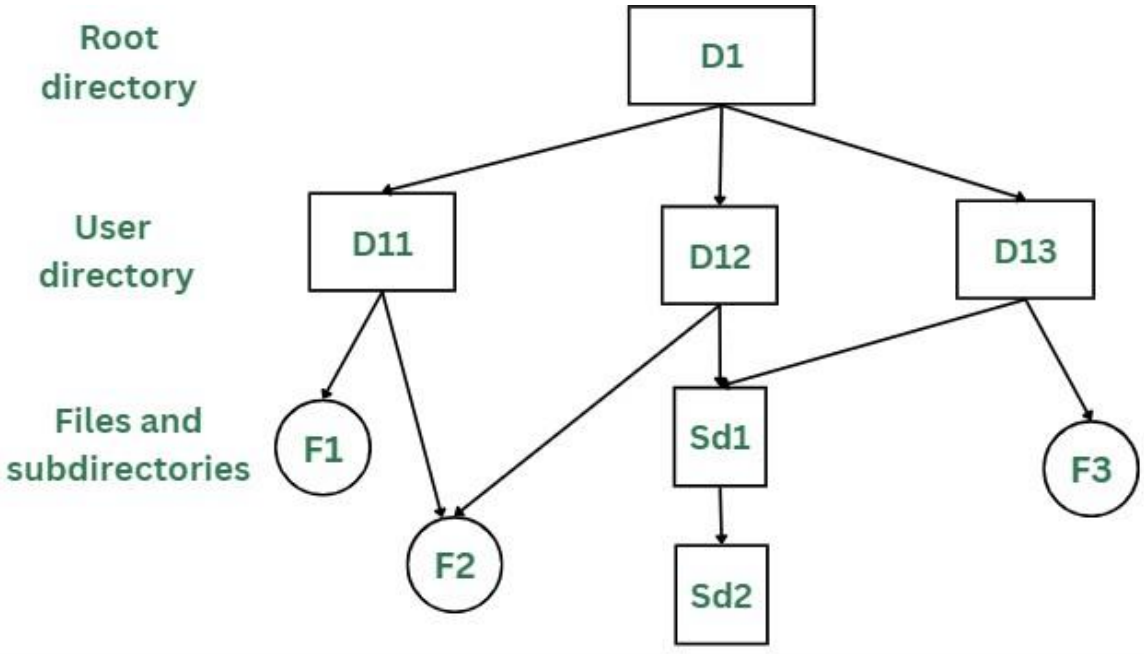
Disadvantages

- As the user isn't allowed to access other user's directory, this prevents the file sharing among users.
- As the user has the capability to make subdirectories, if the number of subdirectories increase the searching may become complicated.
- Users cannot modify the root directory data.
- Searching may become complex as the directory depth increases.

4) Acyclic Graph Structure

In the earlier directory structures, a file could only be accessed from the directory it was stored in. The acyclic graph directory structure solves this by allowing a file or subdirectory to be shared across multiple directories using links. Changes made by one user are visible to all users sharing that file.

In the below figure, this explanation can be nicely observed, where a file is shared between multiple users. If any user makes a change, it would be reflected to both the users.



Acyclic Graph Structure

Advantages

- Sharing of files and directories is allowed between multiple users.
- Searching becomes too easy.
- Flexibility is increased as file sharing and editing access is there for multiple users.

Disadvantages

- Because of the complex structure it has, it is difficult to implement this directory structure.
- The user must be very cautious to edit or even deletion of file as the file is accessed by multiple users.
- If we need to delete the file, then we need to delete all the references of the file inorder to delete it permanently.

5) General-Graph Directory Structure: Unlike the acyclic-graph directory, which avoids loops, the general-graph directory can have cycles, meaning a directory can contain paths that loop back to the starting point. This can make navigating and managing files more complex.

General Graph Directory Structure

In the above image, you can see that a cycle is formed in the User 2 directory. While this structure offers more flexibility, it is also more complicated to implement.

Advantages of General-Graph Directory

- More flexible than other directory structures.
- Allows cycles, meaning directories can loop back to each other.

Disadvantages of General-Graph Directory

- More expensive to implement compared to other solutions.
- Requires garbage collection to manage and clean up unused files and directories.

Protection in File System

In computer systems, alot of user's information is stored, the objective of the

operating system is to keep safe the data of the user from the improper access to the system. Protection can be provided in number of ways. For a single laptop system, we might provide protection by locking the computer in a desk drawer or file cabinet. For multi-user systems, different mechanisms are used for the protection.

Types of Access :

The files which have direct access of the any user have the need of protection. The files which are not accessible to other users doesn't require any kind of protection. The mechanism of the protection provide the facility of the controlled access by just limiting the types of access to the file. Access can be given or not given to any user depends on several factors, one of which is the type of access required. Several different types of operations can be controlled:

- **Read -** Reading from a file.
- **Write -** Writing or rewriting the file.
- **Execute -** Loading the file and after loading the execution process starts.
- **Append -** Writing the new information to the already existing file, editing must be end at the end of the existing file.
- **Delete -** Deleting the file which is of no use and using its space for the another data.
- **List -** List the name and attributes of the file.

Operations like renaming, editing the existing file, copying; these can also be controlled. There are many protection mechanism. each of them mechanism have different advantages and disadvantages and must be appropriate for the intended application.

Access Control :

There are different methods used by different users to access any file. The general way of protection is to associate *identity-dependent access* with all the files and directories an list called access-control list (ACL) which specify the names of the users and the types of access associate with each of the user. The main problem with the access list is their length. If we want to allow everyone to read a file, we must list all the users with the read access. This technique has two undesirable consequences:

Constructing such a list may be tedious and unrewarding task, especially if we do not

know in advance the list of the users in the system.

Previously, the entry of the any directory is of the fixed size but now it changes to the variable size which results in the complicates 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 classification of users in connection with each file:

- **Owner -** Owner is the user who has created the file.
- **Group -** A group is a set of members who has similar needs and they are sharing the same file.
- **Universe -** In the system, all other users are under the category called universe.

The most common recent approach is to combine access-control lists with the normal general owner, group, and universe access control scheme. For example: Solaris uses the three categories of access by default but allows access-control lists to be added to specific files and directories when more fine-grained access control is desired.

Other Protection Approaches:

The access to any system is also controlled by the password. If the use of password is random and it is changed often, this may be result in limit the effective access to a file.

The use of passwords has a few disadvantages:

- The number of passwords are very large so it is difficult to remember the large passwords.
- If one password is used for all the files, then once it is discovered, all files are accessible; protection is on all-or-none basis.

OS File System Architecture
An operating system (OS) is software that manages computer hardware and software resources and provides related services to computers. It acts as an intermediary between applications and computer hardware.

What is a Document System?

A file system is a method that helps in organizing and managing data on different storage devices, such as a hard drive or pen drive. File systems also provide efficient

access to users where they can enable disks for easy data storage, location, and recovery. Most operating systems use a layer for each task, including file systems. Each layer of the archive system is responsible for specific functions.

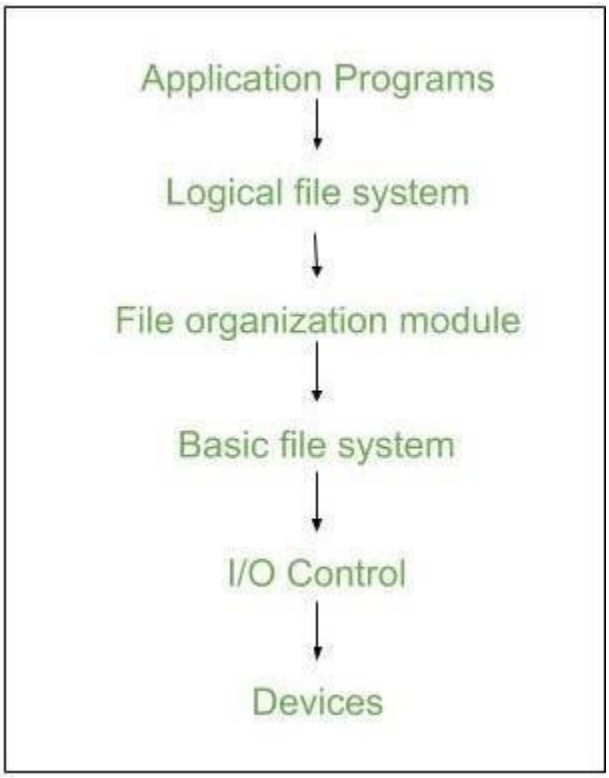
File-System Structure

- File Structure
  - Logical storage unit
  - Collection of related Information
- File System in the storage unit
- The file system is organized into layers

The File System Structure refers to how the files and directories are organized and stored on the physical storage device. This includes the layout of file system data structure such as Directory structure, **file allocation table**, and **inodes**. To Provide efficient and convenient access to the files on the disk, OS imposes one or more file Systems to allow the data to be stored, located, and retrieved easily.

Layered File System

The image describes how the file system is divided into different layers and also the functionality of each layer. When an application requests information, the first request is sent to the information source, which contains metadata for the data and structure. The hard drive is divided into many parts and sectors, and the installation file module determines which physical block the application needs. The overall data is divided into several logical blocks. Data is stored on and retrieved from the hard drive. The hard drive is divided into many parts and sectors and is also responsible for managing free space.



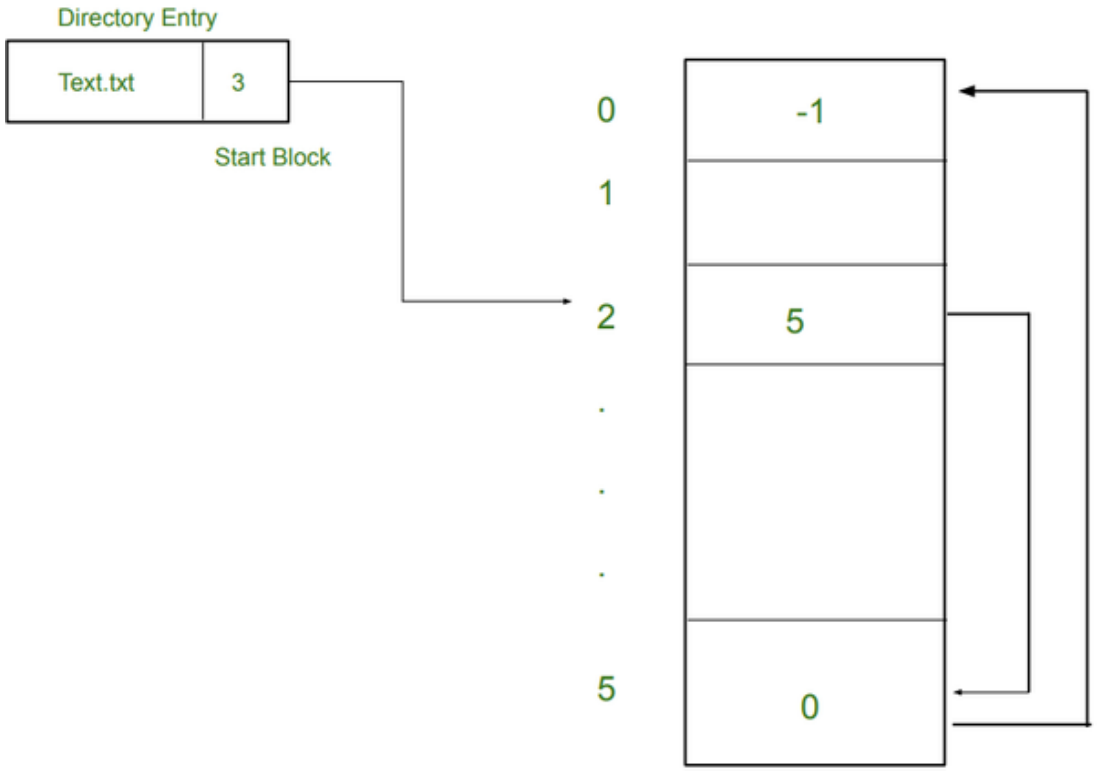
When the installation file module determines which physical block the application needs, it passes this information to the underlying file. The main file is responsible for issuing commands to the I/O controller to retrieve these blocks. The I/O control has access rights to the hard disk. These numbers are called drivers. I/O management is also responsible for interrupts. Unix uses the Unix File System (UFS). Windows series supports FAT, FAT32 and N-TFS.

FAT (File Allocation Table)

File Allocation Table is a file created by Microsoft to support small disks and simple formats. The file system, called a partition table, uses a table to keep track of the volume's heaps. FAT is used to overcome the shortcomings of mounting partition names and has slightly faster access.

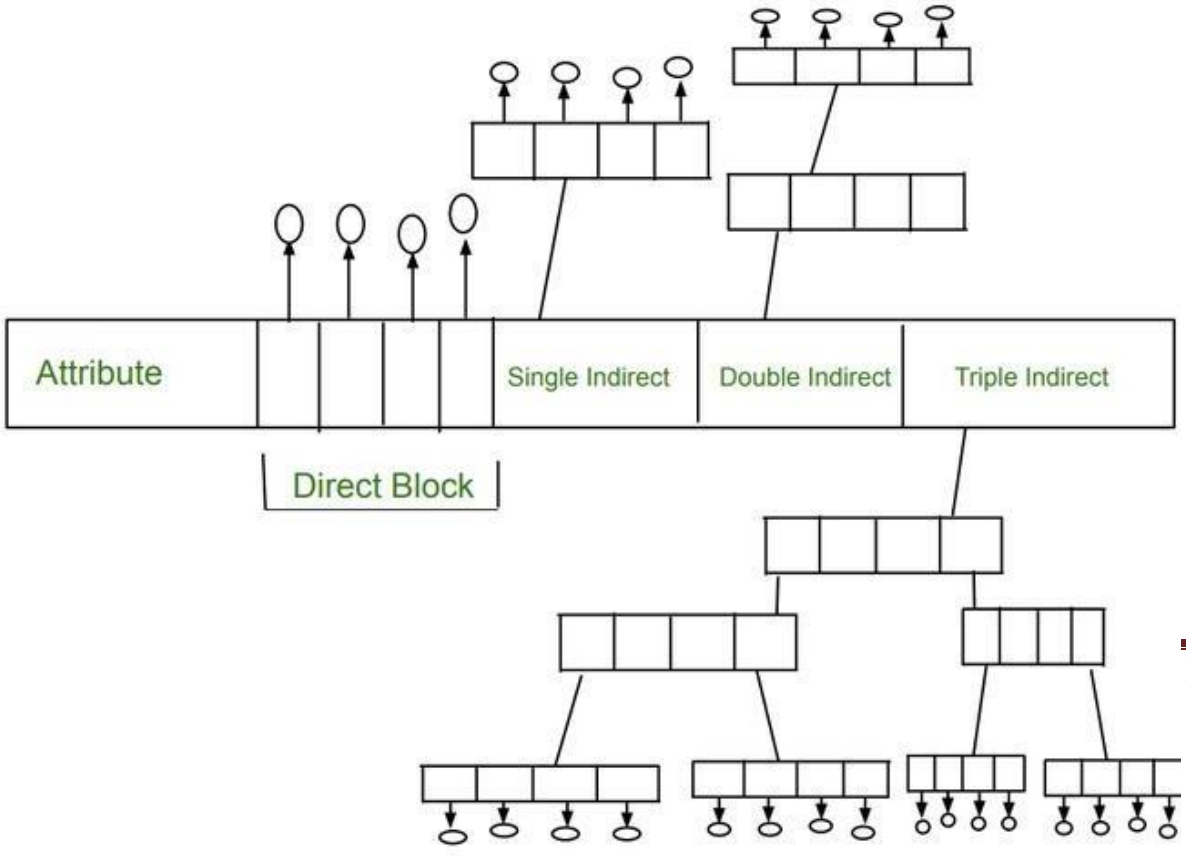
FAT The system creates an index table for data stored on the device or system.

- The index table contains the entry data (data stored in the local area) for each data.
- The operating system searches for the group number of each extension of the file until it reaches the end.
- It supports a maximum volume of 4 GB.



Inode

Inode is a file structure in the UNIX operating system that contains important information about the files in the file. When a file system is created in UNIX, a certain number of indos are also created. Usually about 1% of the file system's disk space is a



File Allocation Methods

The allocation methods define how the files are stored in the disk blocks. There are three main disk space or file allocation methods.

- Contiguous Allocation
- Linked Allocation
- Indexed Allocation

The main idea behind these methods is to provide Efficient disk space utilization & Fast access to the file blocks.

Contiguous Allocation

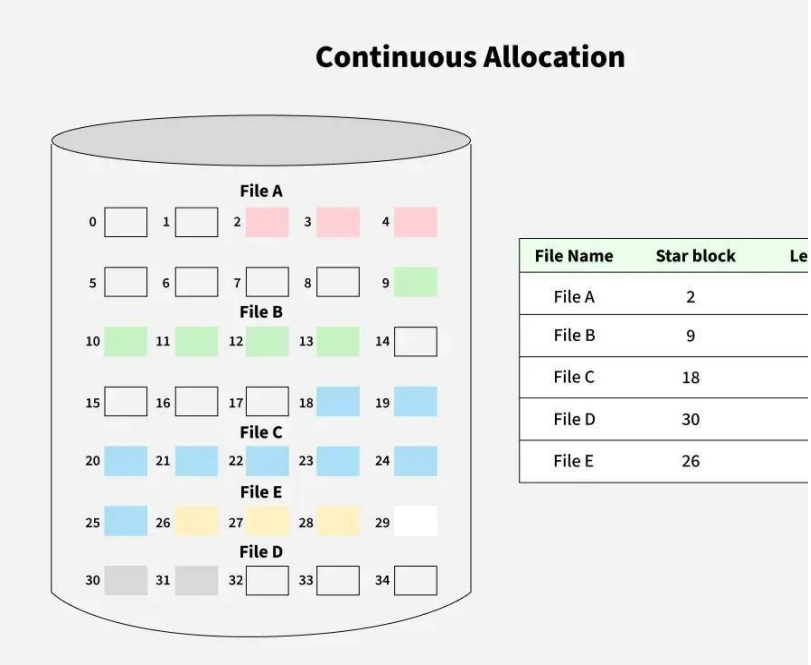
In this scheme, each file occupies a contiguous set of blocks on the disk. This means that given the starting block address and the length of the file (in terms of blocks required), we can determine the blocks occupied by the file. The directory entry for a file with contiguous allocation contains:

- Address of starting block
- Length of the allocated portion.

- **Example:** If a file requires n blocks and is given a block b as the starting location, then the blocks assigned to the file will be:

  b, b+1, b+2,..... b+n-1.

  Contiguous Allocation



Advantages

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

Disadvantages

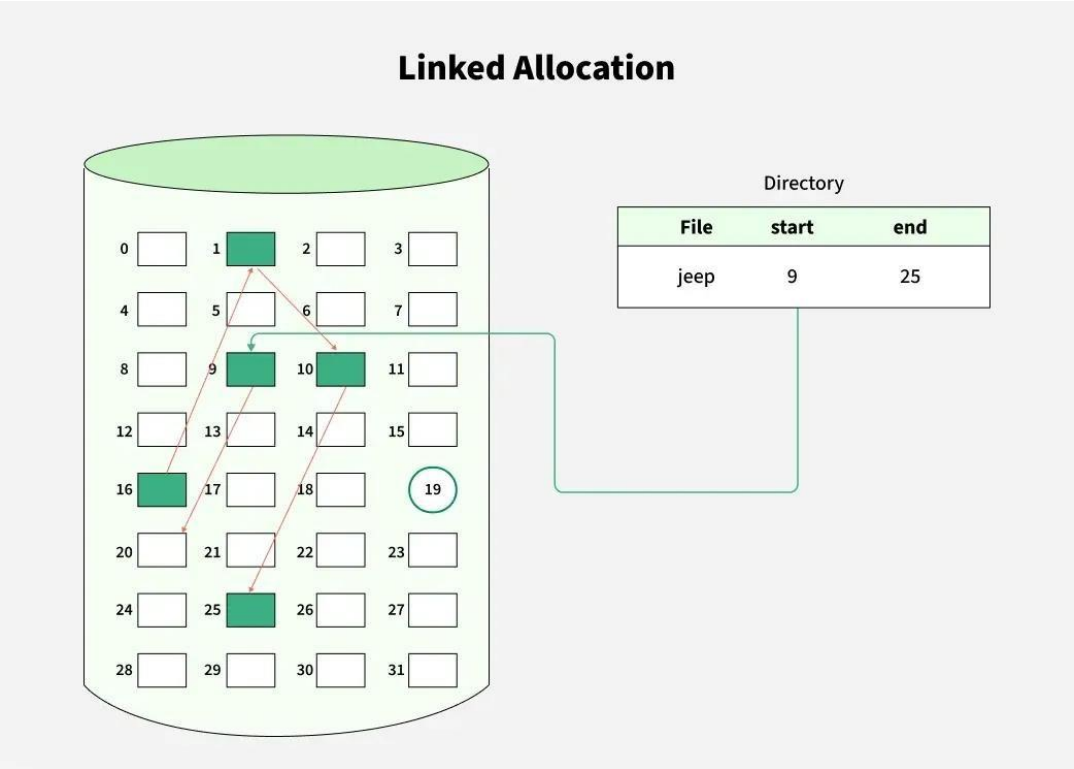
- This method suffers from both internal and external fragmentation. This makes it inefficient in terms of memory utilization.
- Increasing file size is difficult because it depends on the availability of contiguous memory at a particular instance.

Linked Allocation

In this scheme, each file is a linked list of disk blocks which need not be contiguous. Here,

- The disk blocks can be scattered anywhere on the disk. The directory entry contains a pointer to the starting and the ending file block.
- Each block contains a pointer to the next block occupied by the file.
- **Example:** The file 'jeep' in following image shows how the blocks are randomly

distributed. The last block (25) contains -1 indicating a null pointer and does not point to any other block.



-
- Linked Allocation

Advantages

- This is very flexible in terms of file size. File size can be increased easily since the system does not have to look for a contiguous chunk of memory.
- This method does not suffer from external fragmentation. This makes it relatively better in terms of memory utilization.

Disadvantages

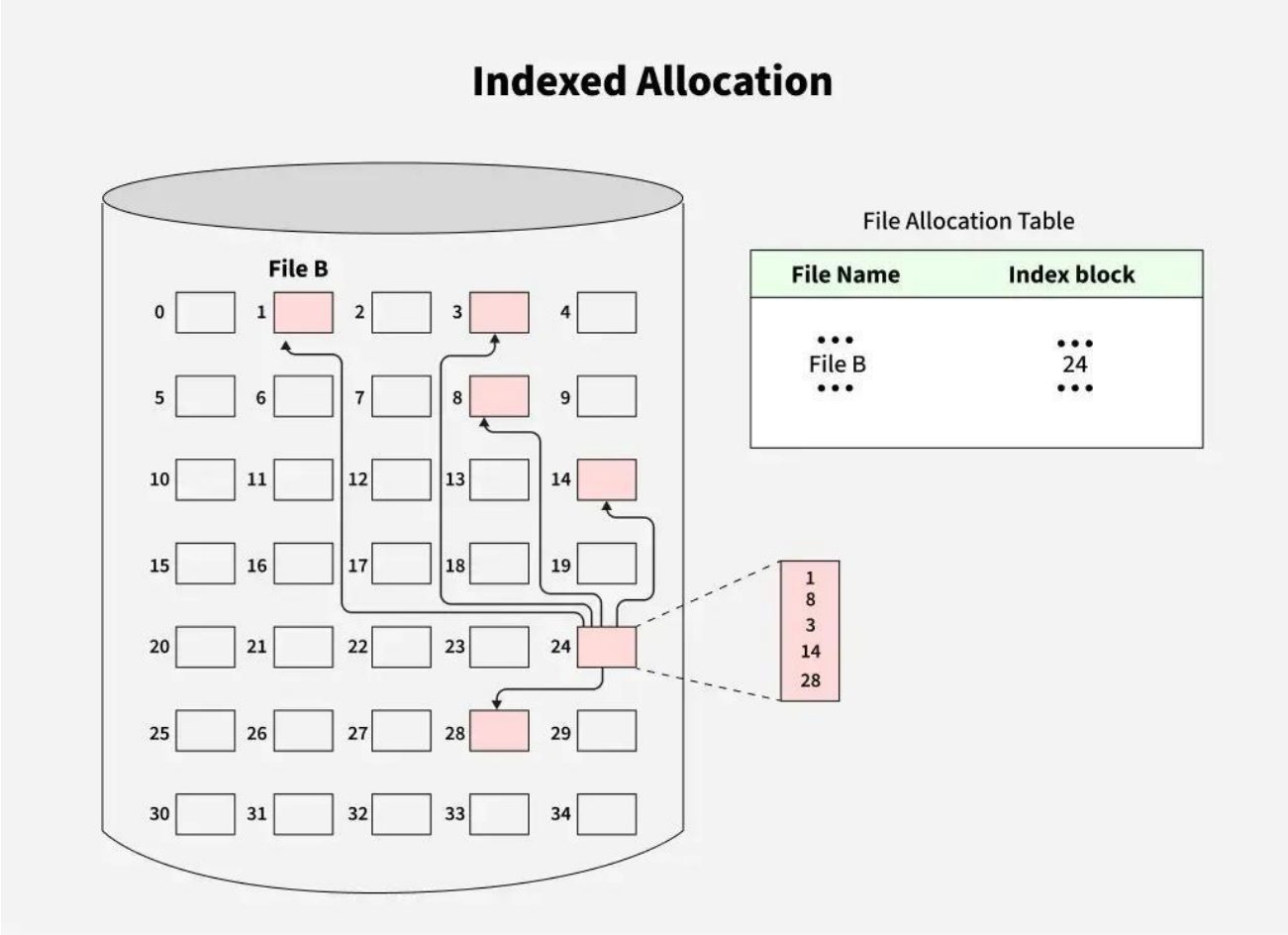
- Because the file blocks are distributed randomly on the disk, a large number of seeks are needed to access every block individually. This makes linked allocation slower.
- It does not support random or direct access. We can not directly access the blocks of a file. A block k of a file can be accessed by traversing k blocks sequentially (sequential access) from the starting block of the file via block pointers.

- Pointers required in the linked allocation incur some extra overhead.

Indexed Allocation

In this scheme, a special block known as the Index block contains the pointers to all the blocks occupied by a file. Here,

- Each file has its own index block.
- The i^{th} entry in the index block contains the disk address of the i^{th} file block.
- The directory entry contains the address of the index block as shown in the image.



Indexed Allocation

Advantages

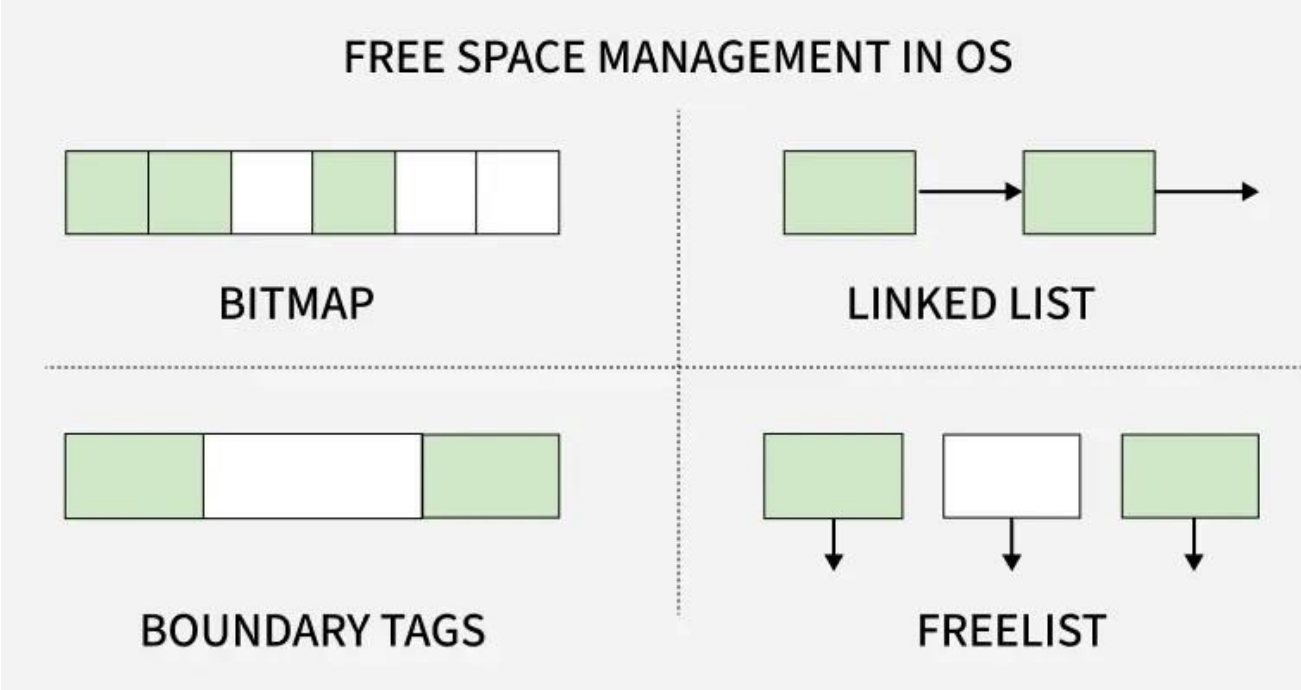
- This supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks.
- It overcomes the problem of external fragmentation.

Disadvantages

- The pointer overhead for indexed allocation is greater than linked allocation.
- For very small files, say files that expand only 2-3 blocks, the indexed allocation would keep one entire block (index block) for the pointers which is inefficient in terms of memory utilization. However, in linked allocation we lose the space of only 1 pointer per block.

Free Space Management in Operating System

Free space management involves managing the available storage space on the hard disk or other secondary storage devices. To reuse the space released from deleting the files, a free space list is maintained. The free space list can be implemented mainlFree Space Management



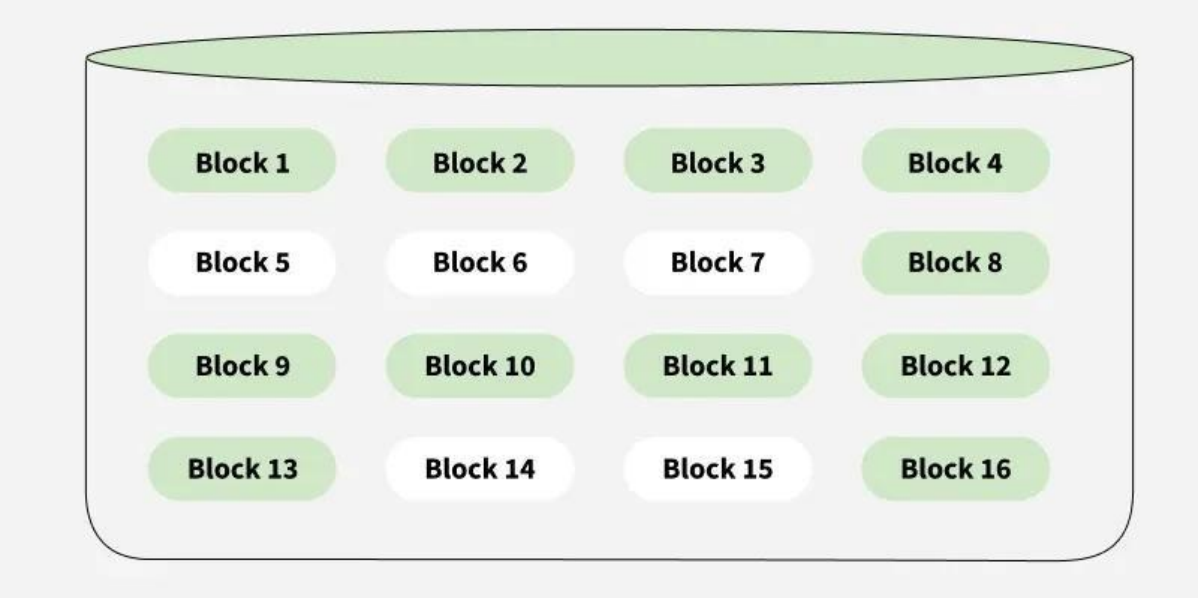
- Bitmap or Bit vector
- Linked List

- Boundary Tags
- Free List

Bitmap or Bit vector

In this approach, A Bitmap or Bit Vector is series or collection of bits where each bit corresponds to a disk block. The bit can take two values 0 and 1:

- 0 indicates that the block is free and 1 indicates an allocated block.
- The given instance of disk blocks on the disk in Figure 1 can be represented by a bitmap of 16 bits as: 1111000111111001.



- Bitmap

Advantages:

- Simple to understand.
- Finding the first free block is efficient. It requires scanning the words (a group of 8 bits) in a bitmap for a non-zero word. (A 0-valued word has all bits 0). The first free block is then found by scanning for the first 1 bit in the non-zero word.

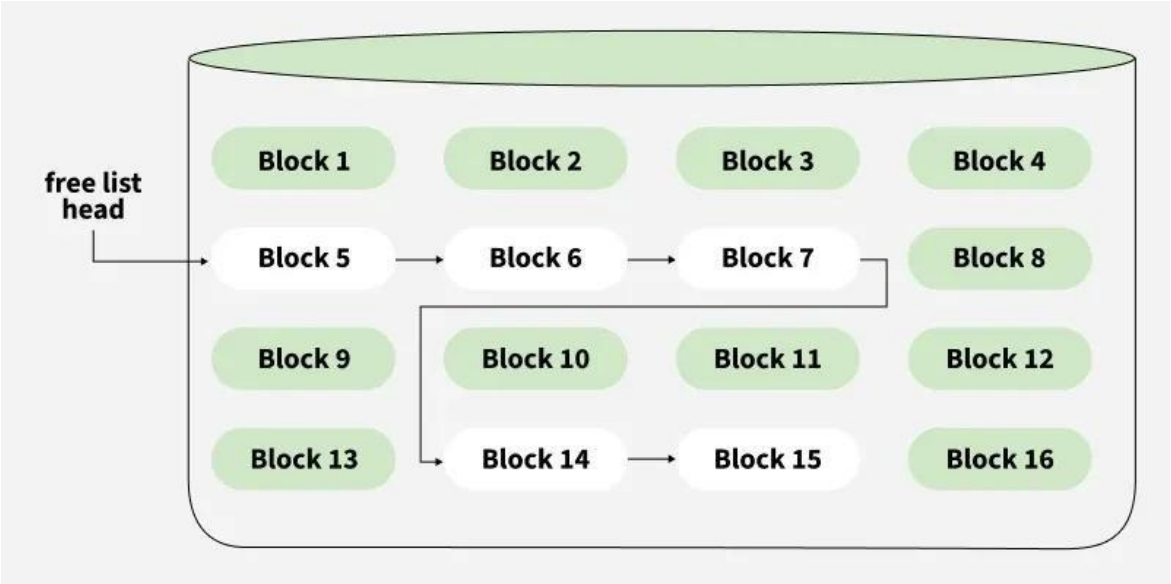
Disadvantages:

- For finding a free block, Operating System needs to iterate all the blocks which is time consuming.

- The efficiency of this method reduces as the disk size increases.

Linked List

In this approach, Free blocks are linked together in a list. Each free block stores the address of the next free block. The list is maintained dynamically as blocks are allocated and freed.



Linked List

Note: *The free space list head points to Block 5 which points to Block 6, the next free block and so on. The last free block would contain a null pointer indicating the end of free list.*

Advantages:

- The total available space is used efficiently using this method.
- Dynamic allocation in Linked List is easy, thus can add the space as per the requirement dynamically.

Disadvantages:

- When the size of Linked List increases, the headache of maintaining pointers is also increases.
- This method is not efficient during iteration of each block of memory.
- I/O is required for free space list traversal.

Boundary Tags

In this approach, Each block contains a boundary tag indicating its size and whether it is free or occupied. Adjacent free blocks are merged during deallocation to reduce fragmentation.

Advantages:

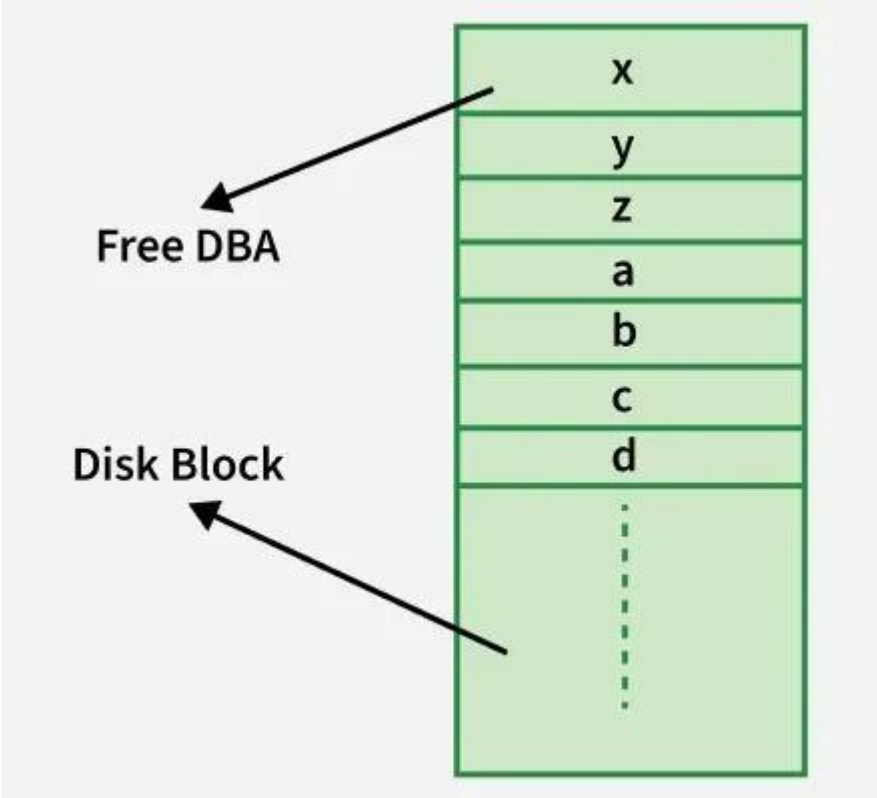
- Useful in memory management systems.
- Simplifies coalescing of adjacent free blocks.

Disadvantages:

- Slight overhead in storing tag information.
- More complex than bitmap or linked list.

Free List

In this approach, The free blocks are maintained in a list (array or linked list). Each entry in the free list points directly to a free block on disk.



Free List

Advantages:

- Fast allocation as free blocks are known upfront.
- Easy to traverse and maintain.

Disadvantages:

- Extra memory needed to store the list.
- May suffer from fragmentation over time.

Input-output system calls

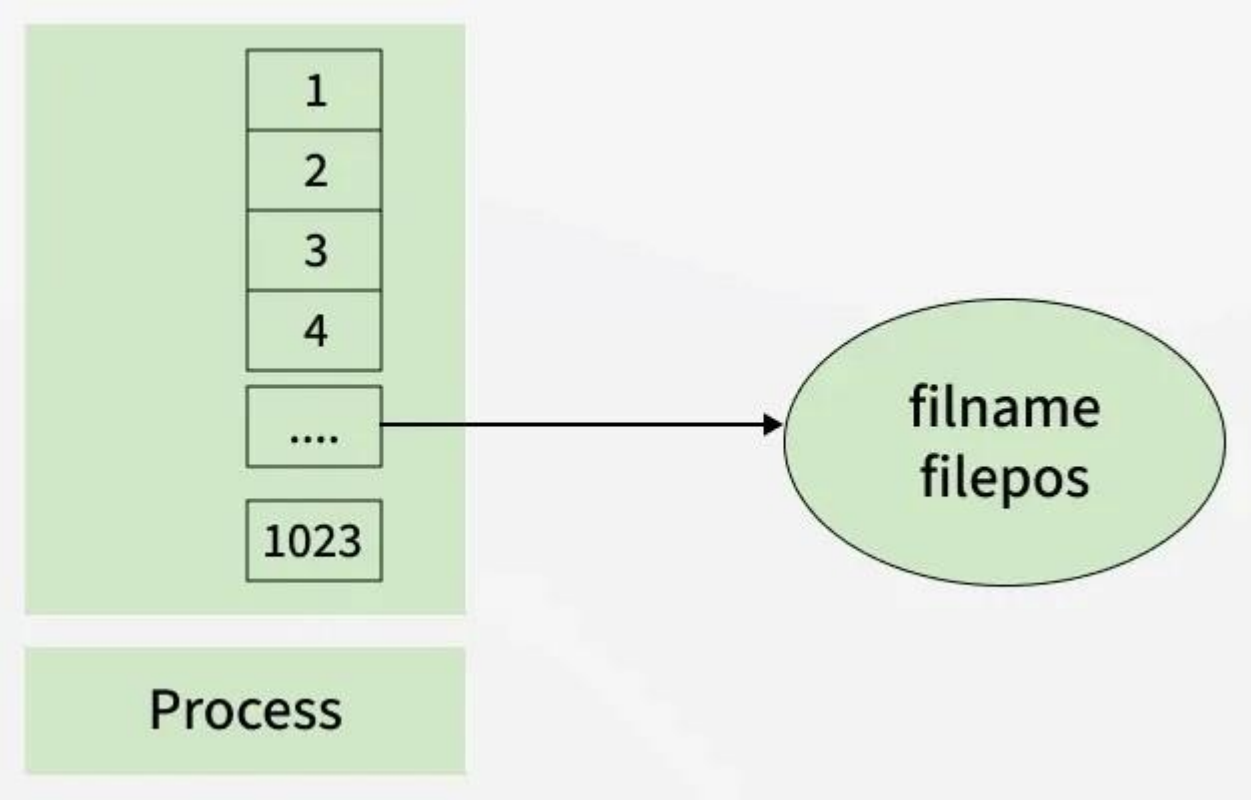
System calls are the interface through which a program requests services from the operating system kernel, especially for operations that require privileged access. In C, many input and output operations ultimately rely on system calls, where the program interacts with the kernel to access devices such as the keyboard, monitor and other I/O resources.

Before we move on to the I/O System Calls, we need to know about the file descriptor.

File Descriptor

A file descriptor is an integer that uniquely identifies an opened file of the process. It is stored in the file descriptor table in which elements are pointers to file table entries. One unique file descriptor table is provided in the operating system for each process.

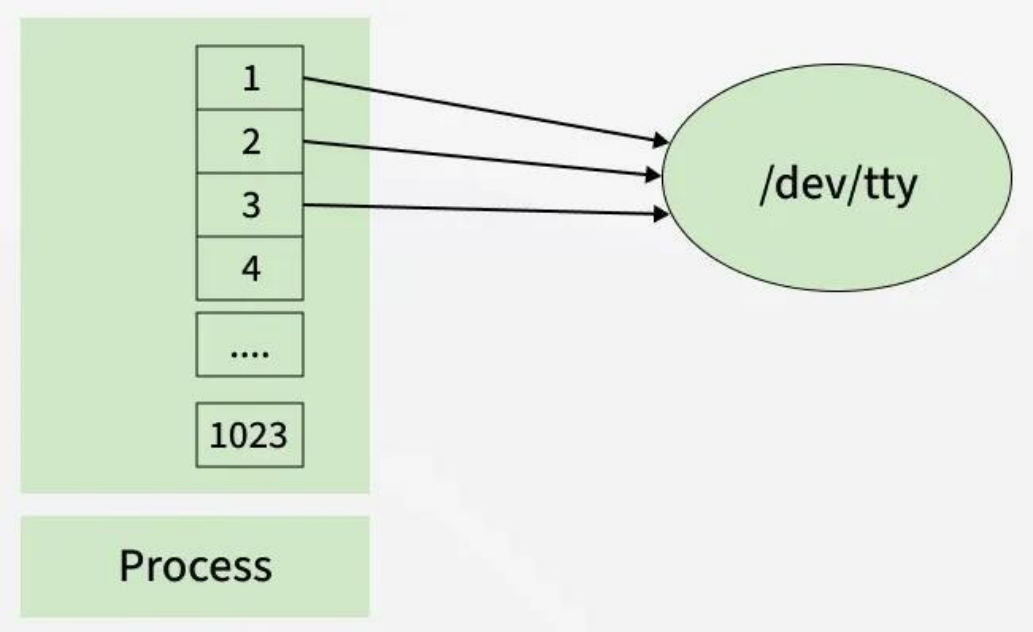
File table entries are a structure in-memory surrogate for an open file, which is created when processing a request to open the file and these entries maintain file position.



Standard File Descriptors

When any process starts, then that process file descriptors table's fd(file descriptor) 0, 1, 2 open automatically, (By default) each of these 3 fd references file table entry for a file named **/dev/tty**

- **stdin (file descriptor 0):** Standard input stream, whenever we write any character from the keyboard, it reads from stdin through fd 0 and saves to a file named /dev/tty.
- **stdout (file descriptor 1)**: Standard output stream, any output to the video screen, it's from the file named /dev/tty and written to stdout in screen through fd 1.
- **stderr (file descriptor 2)**: We see any error to the video screen, it is also from that file write to stderr in screen through fd 2.



Input-Output System Calls in C

The I/O system calls offer direct access to the operating systems' input and output functionalities and provide complete control over file operations, memory management and device communications. There are 5 input and output system calls in C:

1. Creat

The creat system call is used to create a new empty file in C and is provided as the **creat()** function. It allows specifying the file name and permissions of the file. It is declared in the **<fcntl.h>** header file.

Syntax

creat(filename, mode);

Parameters

- **filename:** Name of the file which you want to create
- **mode:** Indicates permissions of the new file.

Return Value

- Returns the first unused file descriptor (generally 3, since 0, 1, and 2 are reserved)

- Return -1 when an error occurs.

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main() {

   // Create a new file or open it
   //if it exists (write-only access)
   // File permissions: rw-r--r--
   int fd = creat("newfile.txt", 0644);

   if (fd == -1) {
      perror("Error creating file");
      return 1;
   }

   // File was successfully created
   printf("File 'newfile.txt' created successfully \n"
         "File descriptor: %d\n", fd);

   // Close the file descriptor
   close(fd);

   return 0;
```

```
}
```

Output

File 'newfile.txt' created successfully

File descriptor: 3

How C create() works in OS?

The create() system call works as follows:

- Create a new empty file on the disk.
- Create file table entry.
- Set the first unused file descriptor to point to the file table entry.
- Return file descriptor used, -1 upon failure.

2. Open

The **open()** function provides the open system call in C. It is used to open the file for reading, writing, or both. It is also capable of creating the file if it does not exist. It is defined inside **<unistd.h>** header file and the flags that are passed as arguments are defined inside **<fcntl.h>** header file.

Syntax

open (path, flags);

Parameters:

Path: Path to the file which we want to open.

- Use the **absolute path** beginning with "/" when you are **not working in the same directory** as the C source file.
- Use **relative path** which is only the file name with extension, when you are **working in the same directory** as the C source file.

flags: It is used to specify how you want to open the file. We can use the following flags:

Flags	Description
O_RDONLY	Opens the file in read-only mode.
O_WRONLY	Opens the file in write-only mode.

Flags	Description
O_RDWR	Opens the file in read and write mode.
O_CREAT	Create a file if it doesn't exist.
O_EXCL	Prevent creation if it already exists.
O_APPEND	Opens the file and places the cursor at the end of the contents.
O_ASYNC	Enable input and output control by signal.
O_CLOEXEC	Enable close-on-exec mode on the open file.
O_NONBLOCK	Disables blocking of the file opened.
O_TMPFILE	Create an unnamed temporary file at the specified path.

Example: Below example demonstrates how to use the open() system call to open an existing file or create it if it does not exist.

```
#include <errno.h>

#include <fcntl.h>

#include <stdio.h>

#include <unistd.h>


extern int errno;
```

```
int main() {

   // If file does not have in directory
   // then file foo.txt is created.
   int fd = open("foo.txt", O_RDONLY | O_CREAT);
   printf("fd = %d\n", fd);

   if (fd == -1) {

      // Print which type of error the code have
      printf("Error Number % d\n", errno);

      // print program detail "Success or failure"
      perror("Program");
   }
   return 0;
}
```

Output

fd = 3

How C open() works in OS?

The open() system call works as follows:

- Find the existing file on the disk.
- Create file table entry.
- Set the first unused file descriptor to point to the file table entry.
- Return file descriptor used, -1 upon failure.

3. Close

The **close system call** is provided as **close()** function in C that tells the operating system that you are done with a file descriptor and closes the file pointed by the file descriptor. It is defined inside **<unistd.h>** header file.

Syntax

close(fd);

Parameter: fd: File descriptor of the file that you want to close.

Return Value

- **0** on success.
- **-1** on error.

Example 1

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main() {
   int fd1 = open("foo.txt", O_RDONLY);
   if (fd1 < 0) {
      perror("c1");
      exit(1);
   }
   printf("opened the fd = % d\n", fd1);

   // Using close system Call
   if (close(fd1) < 0) {
```

```
        perror("c1");
        exit(1);
    }
    printf("closed the fd.\n");
}
```

Output

```
opened the fd = 3
closed the fd.
```

Example 2:

```
#include<stdio.h>
#include<fcntl.h>
int main() {

    // Assume that foo.txt is already created
    int fd1 = open("foo.txt", O_RDONLY, 0);
    close(fd1);

    // assume that baz.tzt is already created
    int fd2 = open("baz.txt", O_RDONLY, 0);

    printf("fd2 = % d\n", fd2);
    exit(0);
}
```

Output

```
fd2 = 3
```

Here, In this code first open() returns **3** because when the main process is created, then fd **0, 1, 2** are already taken by **stdin**, **stdout,** and **stderr**. So the first unused file descriptor is **3** in the file descriptor table. After that in close() system call is free it these **3** file descriptors and then set **3** file descriptors as **null**. So when we called the second open(), then the first unused fd is also **3**. So, the output of this program is **3**.

How C close() works in the OS?

The close() system call works as follows:

- Destroy file table entry referenced by element fd of the file descriptor table
  - As long as no other process is pointing to it!
- Set element fd of file descriptor table to **NULL**

4. Read

The **read system call**, implemented as the **read()** function reads the specified amount of bytes **cnt** of input into the memory area indicated by **buf** from the file indicated by the file descriptor **fd**. A successful **read()** updates the access time for the file.
The **read()** function is also defined inside the **<unistd.h>** header file.

Syntax

read (fd, buf, cnt);

Parameters

- **fd:** file descriptor of the file from which data is to be read.
- **buf:** buffer to read data from.
- **cnt:** length of the buffer.

Return Value

- Return number of bytes read on success.
- Return 0 on reaching the end of file.
- Return -1 on error.
- Return -1 on signal interrupt.

buf needs to point to a valid memory location with a length not smaller than the specified size because of overflow. **fd** should be a valid file descriptor returned from

open() to perform the read operation because if fd is NULL then the read should generate an error. **cnt** is the requested number of bytes read, while the return value is the actual number of bytes read. Also, sometimes read system call should read fewer bytes than cnt.

Example

```
#include <fcntl.h>

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>


int main() {

   int fd, sz;

   char* c = (char*)calloc(100, sizeof(char));

   fd = open("foo.txt", O_RDONLY);

   if (fd < 0) {

      perror("r1");

      exit(1);

   }


   sz = read(fd, c, 10);

   printf("called read(% d, c, 10). returned that"

         " %d bytes were read.\n",

         fd, sz);

   c[sz] = '\0';

   printf("Those bytes are as follows: % s\n", c);
```

```
    return 0;
}
```

Output

called read(3, c, 10). returned that 10 bytes were read.
Those bytes are as follows: 0 0 0 foo.

Suppose that **sample.txt** consists of the 6 ASCII characters **"foobar".** Then what is the output of the following program?

```
#include    <fcntl.h>
#include    <stdio.h>
#include   <stdlib.h>
#include <unistd.h>

int main() {
    char c;
    int fd1 = open("sample.txt", O_RDONLY, 0);
    int fd2 = open("sample.txt", O_RDONLY, 0);
    read(fd1, &c, 1);
    read(fd2, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```

Output

c = f

The descriptors ***fd1*** and ***fd2*** each have their own open file table entry, so each descriptor has its own file position for ***foobar.txt***. Thus, the read from ***fd2*** reads the

first byte of ***foobar.txt***, and the output is c = f, not c = o.

5. Write

The write system call is provided as **write()** function. It writes **cnt** bytes from **buf** to the file or socket associated with **fd**. **cnt** should not be greater than **INT_MAX** (defined in the limits.h header file). If **cnt** is zero, **write()** simply returns 0 without attempting any other action.

The write() is also defined inside **<unistd.h>** header file.

Syntax

write (fd, buf, cnt);

Parameters

- **fd:** file descriptor
- **buf:** buffer to write data from.
- **cnt:** length of the buffer.

Return Value

- Returns the number of bytes written on success.
- Return 0 on reaching the End of File.
- Return -1 on error.
- Return -1 on signal interrupts.

The file needs to be opened for write operations**.** buf needs to be at least as long as specified by cnt because if buf size is less than the cnt then buf will lead to the overflow condition. cnt is the requested number of bytes to write, while the return value is the actual number of bytes written. This happens when fd has a less number of bytes to write than cnt.

If write() is interrupted by a signal, the effect is one of the following:

- If write() has not written any data yet, it returns -1 and sets errno to EINTR.
- If write() has successfully written some data, it returns the number of bytes it wrote before it was interrupted.

Example 1

```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>

main() {
   int sz;

   int fd = open("foo.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
   if (fd < 0) {
      perror("r1");
      exit(1);
   }

   sz = write(fd, "hello geeks\n", strlen("hello geeks\n"));

   printf("called write(% d, \"hello geeks\\n\", %d)."
         " It returned %d\n", fd, strlen("hello geeks\n"), sz);

   close(fd);
}
```

Output

called write(3, "hello geeks\n", 12). it returned 11

Here, when you see in the file foo.txt after running the code, you get a "*hello geeks*". If foo.txt file already has some content in it then the write a system calls overwrite the content and all previous content is ***deleted*** and only "*hello geeks*" content will have in the file.

Example 2

```
#include <fcntl.h>

#include <stdio.h>

#include <string.h>

#include <unistd.h>


int main(void) {

   int fd[2];

   char buf1[12] = "hello world";

   char buf2[12];


   // assume foobar.txt is already created

   fd[0] = open("foobar.txt", O_RDWR);

   fd[1] = open("foobar.txt", O_RDWR);


   write(fd[0], buf1, strlen(buf1));

   write(1, buf2, read(fd[1], buf2, 12));


   close(fd[0]);

   close(fd[1]);


   return 0;
```

```
}
```

Output

hello world

lseek( ) System call:

To change the position of a file pointer in a file can be done by calling the lseek system call.

The syntax for lseek is:

```
#include <sys/types.h>

#include <unistd.h>

off_t lseek(int fd, off_t offset, int ref);
```

The first argument "fd" refers file descriptor of an opened file. The second argument "offset"

refers number of positions to be moved. The third argument "ref" gives the position from where

the displacement of file pointer to be done.

- If "ref" is set to SEEK_SET the positioning is done from the beginning of the file.
- If "ref" is set to SEEK_CUR the positioning is done from the current position.
- If "ref" is set to SEEK_END then the positioning is done from the end of the file.

The function returns the new current position after displacement from the given file or -1 in case of an error.

vii. stat( ) System calls:

The system call stat is used to read the attributes of a file. The syntax is:

```
#include <sys/types.h>

#include <sys/stat.h>

int stat(const char* path, struct stat* buf);
```

The first argument "path" gives the file name. The second argument "buf" is used to

store the file attributes read from the given i-node of a file. The file attributes can be file access types, owner, file size, last access time, last modified time, etc. On success, the

functions return zero, and on error, −1 is returned. The structure struct stat is described in the sys/stat.h header and has the following fields:

```
struct stat {
mode_t st_mode;  /* file access types and rights */
ino_t st_ino;  /* i-node */
dev_t st_dev;  /* identifier of device containing file */
nlink_t st_nlink;  /* nr of links */
uid_t st_uid;  /* owner ID */
gid_t st_gid;  /* group ID */
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char* path, struct stat* buf);
 off_t st_size;  /* ordinary file size */
time_t st_atime;  /* last time it was accessed */
time_t st_mtime;  /* last time it was modified */
time_t st_ctime;  /* last time settings were changed */
long st_blksize;  /* optimal size of the I/O block */
long st_blocks;  /* nr of 512 byte blocks allocated */
};
```

viii. ioctl system call:

IOCTL is referred as Input and Output Control. The system call ioctl( ) is used to interact with

device driver files. The major use of this is to handle some specific operations of a device for

which the kernel does not have a system call by default. It manipulates the underlying

device

parameters of device driver files. Some real time applications of ioctl( ) are:

- Ejecting the media from a "cd" drive
- to change the Baud Rate of Serial port
- Adjust the Volume
- Reading or Writing device registers, etc.

The syntax is:

```
#include <sys/ioctl.h>
int ioctl(int fd , int request, <Arguments> );
```

The first argument "fd" is a file descriptor of an opened file. The ioctl command needs to be executed on this opened file, which would generally be device files. The second argument

"request" is a device-dependent request code. The request code varies from device to device.

The ioctl command implements the task associated with request code to achieve the desired functionality. The third argument is an untyped pointer to memory. It's traditionally char

*argp. An ioctl( ) request has encoded in it whether the argument is an in parameter or out parameter, and the size of the argument argp in bytes. Macros and defines used in specifying

an ioctl( ) request are located in the file <sys/ioctl.h>. Usually, on success zero or positive value is returned. On error, -1 is returned