Module 1: INTRODUCTION

Definition of file structures:

A file structure is the organization of data in files and files in secondary storage and operations for accessing the data. A file structure allows applications to read, write and modify data. It also supports in finding the data that matches some search criteria.

The Heart of File Structure Design.

Disks are slow compared to RAM or ROM present in the system. It takes more time to access data from disk when compared with RAM or ROM. On the other hand they provide enormous capacity to hold data at much lesser cost. Disks are non-volatile and preserve the information even when the system is turned off.

The main driving force behind the file structure design is the relatively slow access time of disk and its enormous, non-volatile capacity.

Good file structure design will give us access to all the capacity of disk, without making our application spend a lot of time waiting for the disk.

A file structure is the representation of organizing the data on the secondary memory in a particular fashion, so that retrieving data becomes easy.

An improvement in file structure design may make an application hundreds of times faster.

A Short History of file structure design

General goals in design of file structures-

- Ideally, we would like to get the information we need with one access to the disk.
- If it is impossible to get the data in one access, we want structures that allow us to find the target information with few access as possible.
- The file structures should be grouped, so that large amount of required data can be fetched by a single access.

Earlier files were stored on tapes. Access to these tapes was sequential. The cost to access the tape grew in direct proportion to the size of the file.

As files grew and as storage devices such as disk drives became available, indexes were added to files. Index consist of a list of keys and corresponding pointers to the files. This type of accessing files could search the files quickly.

As the index file grew, they too became difficult to manage. In early 1960's, the idea of applying tree structures emerged. But this also resulted in long searches due to uneven growth of trees, as a result of addition & deletion of records.

In 1963 researchers developed, an elegant, self adjusting binary tree structure called the AVL tree, for storing data in memory. The problem with this binary tree structure was that dozens of access were required to find a record.

The solution to this problem emerged in the form of B-tree. B-tree grows from the bottom-up, as records are inserted. B-tree provided excellent access performance, except that the files couldn't be accessed sequentially. This problem was solved by introducing B+ trees.

But all the above methods needed more than one disk access.
Hashing approach promised to give required data in one disk access, but performed badly for large amount of data.

Extendible Dynamic hashing approach is proposed to improve the performance of hashing algorithm.

Fundamental File Processing Operations:

- Opening files.
- Closing files.
- Reading
- Writing.
- Seeking.

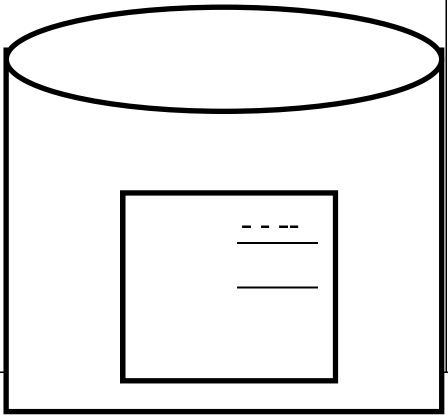
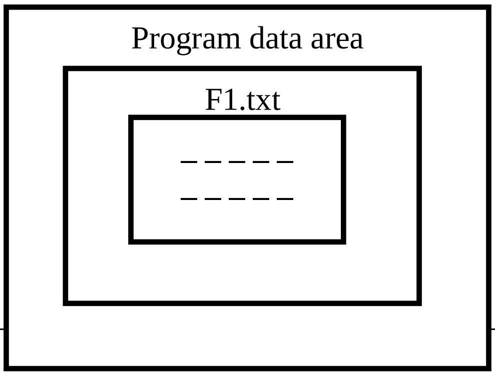
Physical files and logical files:

A file is a particular collection of stored bytes. The file residing in secondary devices and managed by operating system is referred as Physical files.

In order to manage (edit or use) the file, it has to be loaded on the primary memory and must be accessed by a particular application to handle the contents of the file. Such an alternate view of a file on the primary memory is called the Logical file.

Logical file resembles the physical file in its content, but is visible only to the application which handles it. The changes are done in the logical file and finally when the file is saved, the changes are updated to the physical file by the operating system.

Difference between physical File and Logical File

Physical File	Logical File
1) Resides in secondary memo 2) Change are not done dir by the application 3) Change are updated when file is saved 4) Can be accessed by any nur of applicants 5) Sec. Memory 	1) Resides in primary memory 2) Change are done directly to logical file by the application 3) Change are updated when application changes the values 4) Can be accessed by only application which handle it 5) Primary memory 

Opening files:

Opening a file can be done in 2 ways-

1. Opening a existing file.
2. Create a new file.

When a file is opened, the read/write pointers are positioned at the beginning of the file and are ready to read/write. The file contents are not distributed by the open statement. Creating a new file also opens the file, as it doesn't have any contents, it is always created in write or read write mode.

The "open" function is used to open an existing file or to create a new file. This function is present in the header "fcntl.h" in C language.

The syntax of open function is-

```
fd=open(filename,flags[,pmode]);
```

where,

->fd - is the file descriptor, which is of type int. if there is an error in the attempt to open the file, this value is negative.

->filename - this is of type char*. This string contains the physical filename.

->flags - this argument controls the operation of the open function. It determines whether to open an existing file for reading or writing. It also determines whether to open an existing file or create a new file.

A bitwise OR of the following values can also be performed. The different flags are-

- O_APPEND : append every write operation to the end of the file.
- O_CREAT : create and open a file for writing.
- O_EXCL : returns an error if O_CREAT is specified & file already exists.
- O_RDONLY : open a file for reading only.
- O_TRUNC : if a file exists, truncate it to a length of zero, ie. delete its contains.
- O_RDWR : open a file for reading and writing.
- O_WRONLY : open a file for writing only.

->pmode- this is of type int. it is used only if O_CREAT is useed. pmode is protection mode which specifies the read, write and execute permission for owner, group and others on the created file.
It is a 3digit octal number. The first digit indicate how the file can be used by owner, 2nd indicate the permission on file by group and 3rd by others.

```
                   r w e     r w e    r w e
Pmode = 0751 ->    1  1  1   1  0  1  0  0  1
                   Owner     group    others
```

*owner has all read,write and execute permission.
*group has both read and execute permission.
*others have only read permission.

Example of opening an existing file for reading and writing or create a new one if necessary (doesn't exist)
fd=open(filename,O_RDWR|O_CREAT,0751);

Closing files:

Files are usually closed automatically by the operating system when a program terminates normally. The execution of close statement within a program is needed to protect it against data loss, in the event tat the program is interrupted and to free up logical filenames for reuse.
"close()" function is used to close the file.

Reading and Writing:

Reading and writing are the fundamental file processing operations, a read function call requires three pieces of information.

The syntax of Read function is-
Read(source_file,Destination_addr,size) – read from file.

-> source_file	:the filename from where the data has to be read.
-> destination_addr	:specifies the address where the read data has to be stored.
-> size	:the maximum number of bytes that can be read from the file.

The syntax of write function is-

Write(destination_file,source_addr,size) – write to file.

->destination_file : the logical filename that is used for writing the data.

-> source_addr : address, where the information is found. This information is written to the destination file.

-> size :the maximum number of bytes that can be written to the file.

File handling with C streams and C++ stream classes:

2 ways of manipulating files in c++:

1) Using standard C functions defined in header file stdio.h – C streams.

2) using stream classes defined in iostream.h and fstream.h _ C++ streams.

In C streams uses ‘fopen’ function is used to open the file.

File=fopen(filename,type);

->file : a pointer to the file descriptor.

->filename : name of the file to be opened.

->type : This argument controls the operation of the open function.

The values can be:

“r”	: open an existing file for input(read).
“w”	: create a new file, or truncate an existing one for output.
“a”	: create a new file, or append to an existing one for output.
“r+”	: open an existing file for input and output.
“w+”	: create a new file or truncate an existing one, for input and output.
“a+”	: create a new file,or append to an existing one for input and output.

Read and write operations are supported by fread, fget, fwrite, fput, fscanf and fprintf.

The C++ streams uses equivalent operations, but the syntax is different. The function to open the file is ‘open’.

int open(char*filename, int mode);

filename - is the name of the file to be opened.

mode – controls the way the file is opened, the options are:

->ios::in - open file for input(reading).

->ios::out – open file for output(writing).

->ios::nocreate – open file if it exists, no creation of file.
->ios::noreplace – open a new file, no replacement ie, error if file already exists.
->ios::binary – specifies that the file is binary.

The methods to read and write are-
int read(unsigned char*, dest_addrs,int size);
A line of data can be read to a buffer using the extraction(>>) operator.
Syntax is –
fp(file stream object) >> buffer

To write to the file -
int write(unsigned char*,source_addr,int size);
A line of data can be written to the file from buffer, using the insertion(<<) operator.
Syntax is –
fp(file stream object) << buffer

A C++ program that opens a file for input and reads it, character by character, sending each character to the screen after it's read from the file.(Implementation of **cat** command)

```
#include<fstream>
#include<iostream>

using namespace std;
main()
{
	char ch;
	fstream file;
	char filename[20];
	cout<<"enter the name of the file:"<<flush;  // force output
	cin>>filename;
	file.open(filename,ios::in);
	while(1)
	{
		file>>ch;
		if(file.fail())
			break();
```

```
        cout<<ch;
    }
    file.close();
}
```

In the program, the file is opened in input(read)mode.

Here a character is read from the file and written to the standard output.

The function fail() returns true, if the previous operation on the stream failed. In the above program the stream reads a character from file. If the reading fails(), as there is no more content ie end of file, then the fail() function returns true and the whole loop is exited(due to break statement).

In C, the same program can be written as –

```
#include<stdio.h>
#include<fcntl.h>
main()
{
    char ch;
    file *fp;
    char filename[20];
    printf("Enter the name of the file");
    gets(filename);
    fp =  fopen(filename,"r");
    while(fread(&ch,1,1,fp));
    {
        fwrite(&ch,1,1,stdout);
    }
    fclose(fp);
}
```

2) Implement the UNIX command ‘tail –n filename’, where n is the number of lines from the end of the file to be copied to the stdout(screen).

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
#define REC_SIZE 20

void main()
{
        int n,i;
        char buf[100];
        fstream fp;
        clrscr();
        cout<<"Enter the no. of lines to be displayed\n";
        cin>>n;
        fp.open("in.txt",ios::in);
        if(fp.fail())
                cout<<"couldnot open the file\n";
        fp.seekg( (REC_SIZE*n),ios::end);
        while(fp)
        {
                fp>>buf;
                cout<<buf<<"\n";
        }
        getch();
}
```

3) Implement the UNIX command ‘cp file1 file2’, where ‘file1’ contents are copied to ‘file2’.

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>

void main()
{
        int i;
        char buf[100];
        fstream fp1,fp2;
        clrscr();
        fp1.open("in.txt",ios::in);  //’read’ mode
        fp2.open(“out.txt”,ios::out);   //’write’ mode
        if(fp1.fail() || fp2.fail())
                cout<<"couldnot open the file\n";
        while(fp1)
        {
                fp1>>buf;
                fp2<<buf<<"\n";
        }
        getch();
}
```

Seeking:

The action of moving directly to a certain position in a file is called seeking. Sometimes in the program, we may need to jump to a byte which is ten thousand bytes away or to the end of the file to read or write some contents, then seek() function is used.

seek() function requires atleast two pieces of information –

seek(source-file,offset)

source-file	: is the logical filename in which seek operation will occur.
offset	: number of positions to be moved from the start of the file.

Eg - seek(data, 370);

The read/write pointer is moved to the 370th byte in the file 'data'.

Seeking using C stream -

In C fseek() is the function used to move the pointer to any byte in a file.

pos=fseek(file,byte_offset,origin);

where: pos	- moved position value of read/write pointer after fseek() function.
file	- file descriptor of the file, to which fseek() is to be applied.
byte_offset	- the no of bytes to be moved from some origin in file.
origin	- a value that specifies the starting position from which the byte_offset must be taken,it can have 3 values SEEK_SET(0),SEEK_CUR(1),SEEK_END(2).

0 - fseek from beginning of file.
1 - fseek from the current position.
2 – fseek from the end of the file.

Eg: pos=fseek(data,370L,0)

The pointer is moved to 370th position from start of the file 'data'.

Seeking using C++ Stream:

In C++, the fstream has 2 pointers. A **get** pointer for reading the data and a **put** pointer for writing the data.

During seeking, seekg() method moves the get pointer and seekp() method moves the put pointer.

The syntax of seek operations are –

```
file.seekg(byte_offset,origin);
file.seekp(byte_offset,origin);
```

byte_offset – is the no of bytes to be moved from origin.

Origin takes values like

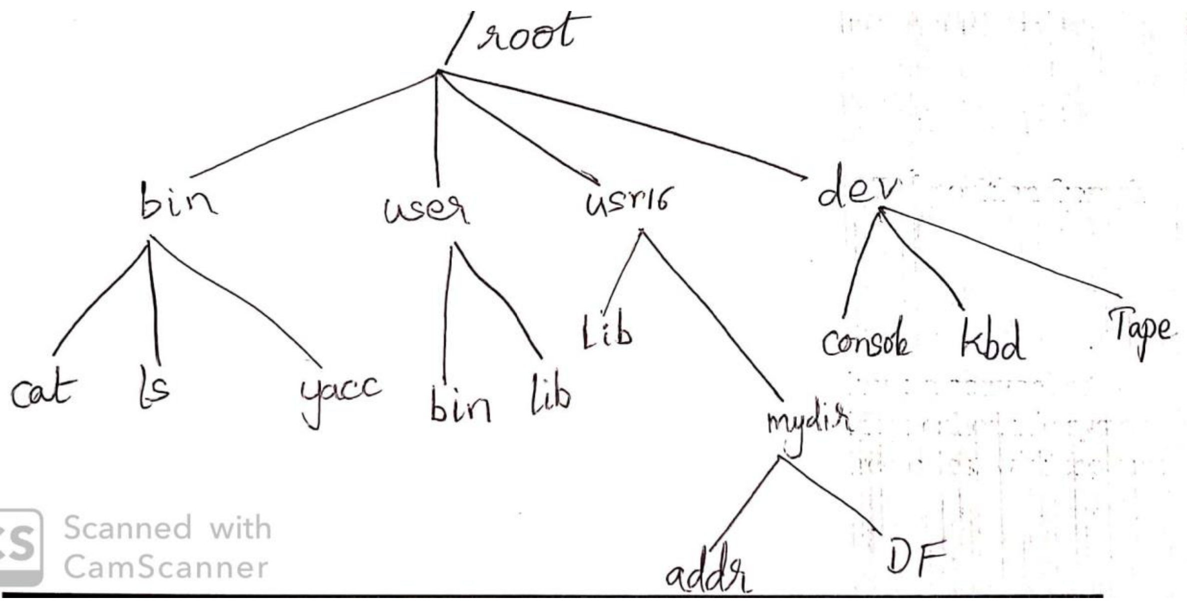
ios::beg – move from beginning of the file.
ios::cur – move from current position.
ios::end – move from the end of the file.

Eg: file.seekg(370,ios::beg);
file.seekp(370,ios::beg); - moves both pointers to the 370th position from the beginning of the file.

The UNIX Directory Structure:

There are hundreds or thousands of files in a computer. To have a convenient access to these files, they are organized in a simple way. In UNIX it is called file system.

The UNIX file system is a tree-structures organization of directories, with the root of the tree signified by the character '/'.



Any directory in UNIX can contain only 2 kinds of files:

1. Regular file.
2. Directory, containing files or directories.

Any file in UNIX file system can be uniquely identified by the absolute pathname, that begins with the root directory.
For eg: absolute path name of file ‘addr’ is ‘/usr6/mydir/addr’
All the pathnames that begin with current directory are called as Relative pathname. The special file names “.” Stands for current directory and “..” stands for parent directory.
<u>Physical devices and logical files:</u>
In UNIX, physical devices like keyboard, monitor, hard disk etc directories and logical files everything are considered as files.

A keyboard is considered as a file, which produces a sequence of bytes that are sent to the computer when keys are pressed. The console (monitor) accepts a sequence of bytes and displays their corresponding symbols on screen. These devices have their filename as ‘/dev/kbd’ and ‘ /dev/console’ respectively.
In UNIX, a file is represented logically by an integer file descriptor. A keyboard, a diskfile and a magnetic tape are all represented by integers. Once the integer that describes a file is identified, a program can access that file using the integer.

The below statements show, how data can be read from a file using the file descriptor and then displayed on the console.

```
file=fopen(“newfile.c”, “r”);
while(fread(&ch,1,1,file)!=0)
        fwrite(&ch,1,1stdout);
```

The logical file of “newfile.c” is represented by the value returned by fopen() method. Similarly the console is represented by the value ‘stdout’ defined in ‘stdio.h’.
Similarly the keyboard is represented by ‘stdin’ and the error file is represented by ‘stderr’(standard error) in stdio.h.

The statement, fread(&ch,1,1,stdin) reads a single character from the keyboard. ‘stderr’ is a error file which represents the console. Errors can be passed to the console by using this file also.

<u>I/O Redirection pipes:</u>

The output of a program can be sent to a regular file rather than to stdout or the output can be sent as the input to other program, by using some short symbols . Such short symbols are called <u>I/O redirection and pipes</u>.

I/O redirection allows to specify alternative files for input or output at the execution time. The notations for input and output redirection on the command line in UNIX are

< file (redirect stdin to 'file')
> file (redirect stdout to 'file')

Eg: If the output of a program should be redirected to a file instead of stdout(console), the below syntax is used:

List.exe>myfile //output of 'list.exe' is put in a file called "myfile"

If the output of a program is to be fed as input to another program, then the pipe symbol('|') is used.

Syntax: program1| program2

The result of program1 is sent as input to program2.

Eg: In UNIX, 'ls' cmd is used to list all the files and subdirectory in the current directory.

The output of this can be sent as input to 'wc' command.

The wc command displays the no of lines, no of words and no of characters in the input file or input.

$ ls|wc lists the no of files, no of words and no of characters in the output of ls.

<u>File-Related header files:</u>

The special values like end-of-file(EOF), access permissions while opening a file etc are defined in particular header files. In UNIX headers files are present in /usr/include file.

Header files requires for file handling problems are- iostream.h, fstream.h, fcntl.h, file.h.

The C++ streams are in iostream.h and fstream.h. Many UNIX operation are in fcntl.h and file.h.

The flags O_RDONLY, O_WRONLY and O_RDWR are usually found in file.h.

<u>UNIX File system commands:</u>

UNIX provides many commands for manipulating files. Some of them are listed below-

• cat filename	– print the contents of the named file.
• tail filename	– print the last ten lines of the text file.
• cp f1 f2	– copy the contents of f1 to f2.
• mv f1 f2	– move (rename) f1 to f2.
• rm filename	– remove(delete) file.
• chmod mode f1	– change the protection mode on the named file.
• ls	– list the contents of the directory.
• mkdir name	– create a new directory with the name given.
• rmdir	– remove the named directory.

Secondary storage and system software:

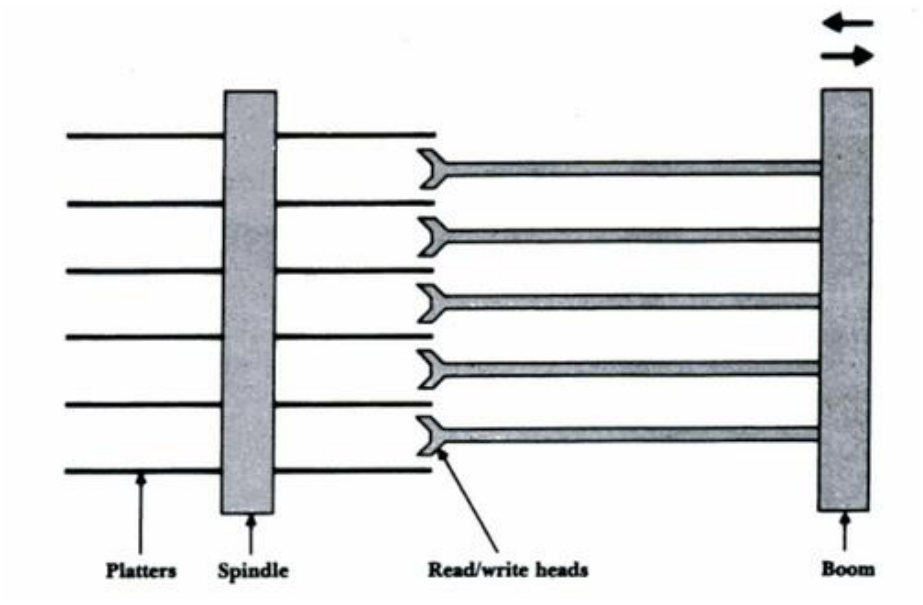
Organization of disks-

Magnetic disks come in many forms, like hard disk, floppy disk, removable disk.

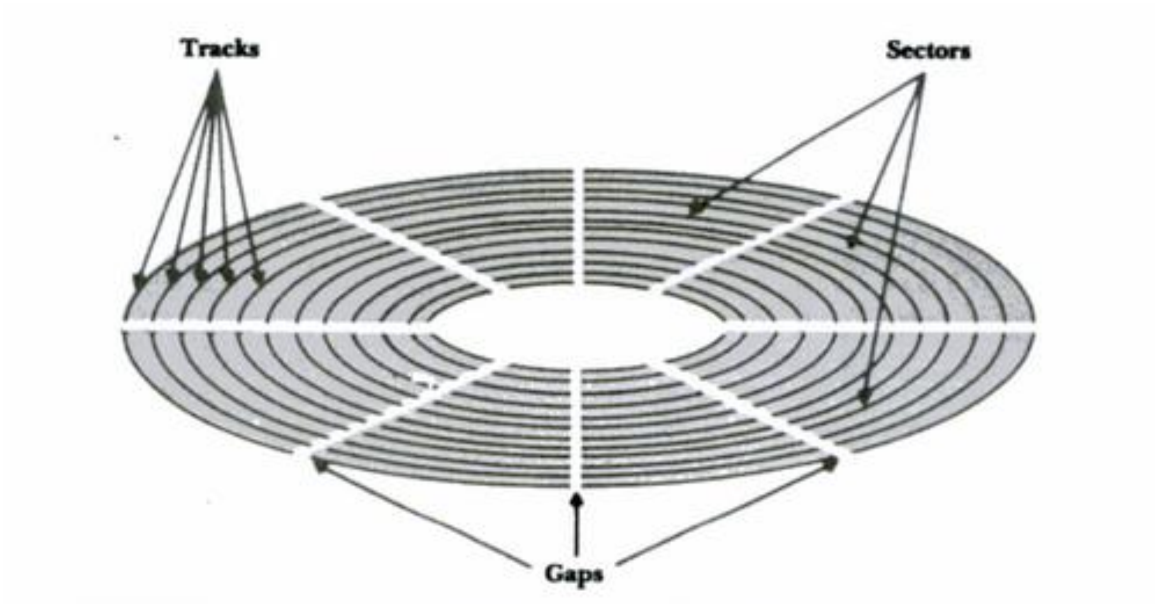
Hard disks offers high capacity and low cost per bit. Hence, it is commonly used in everyday file processing.

The information stored on a disk is stored on the surface of one or more platters. The information is stored in successive tracks on the surface of the disk. The read/write head assembly is used to read and write the data from/to the disk.

Schematic illustration of disk dirve



Surface of disk:

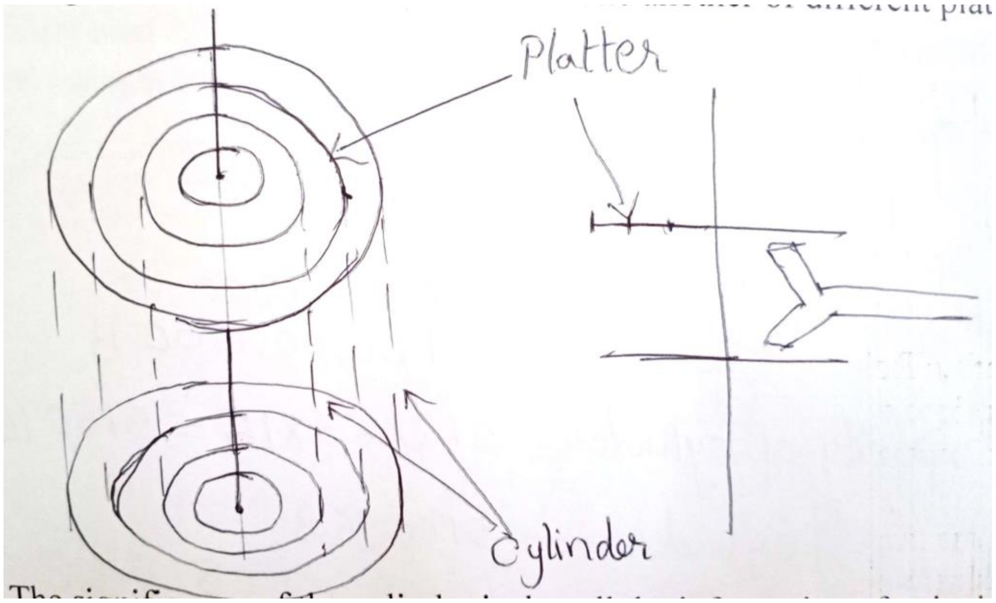


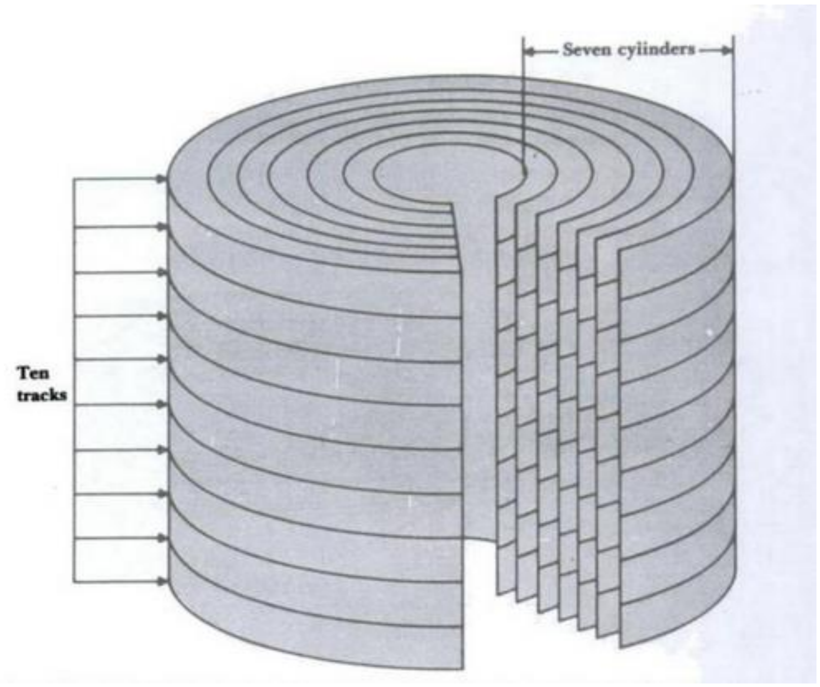
Each track is often divided into a number of sectors.

A sector is the smallest addressable portion of the disk.

When a read statement calls for a particular byte from a disk file, the operating system finds the correct surface, track and sector. Operating system copies the entire content of the sector into a special area in memory called the buffer and finds the requested byte with that buffer.

As shown in the above figure, a disk drive has a number of platters. The corresponding tracks that are below and above one another of different platters form a cylinder.





The significance of the cylinder is that all the information of a single cylinder can be accessed without moving the read/write head. Moving the arm holding the read/write head is called seeking. This is the slowest activity in reading the information from the disk.

In a typical disk, each platter has two surfaces, so the number of tracks per cylinder is twice the number of platters.

The number of cylinders is same as the no of tracks on the disk surface and each track has the same capacity. Hence the capacity of the disk is a function of the number of cylinders, number of tracks per cylinder and the capacity of the track.

A cylinder consists of group of tracks.
A track consists of group of sectors.
A sector consists of group of bytes.

So, Track capacity=no of sectors per track*bytes per sector.
Cylinder capacity=no of tracks per cylinder* track capacity.
Drive capacity=No of cylinders*cylinder capacity.

Even though the size of each cylinder is the different, the capacity of each cylinder is same.

Solve:

1) Suppose we want to store a file of 50,000 fixed length data records on a typical small computer with the following characteristics:
   No of bytes per sector=512 bytes.
   No of sectors per track = 63.
   No of tracks per cylinder = 16.
   No of cylinders= 4092.
   How many cylinders does the file require if each data record requires 200 bytes? What is the total capacity of the disk?

Solution:

Size of the file – 50,000 X 200 bytes
= 1,00,00,000 B

Capacity of cylinder = 512 x 63 x 16 = 516096 B

1 cylinder ---------- 516096 B
? ------------1,00,00,000 B

No. of cylinders required = 1,00,00,000 / 516096 = 19.38
~ 20 cylinders.

Total capacity of disk = no.of cylinders x cylinder capacity
= 4092 x 516096 B

= 2,111 MB

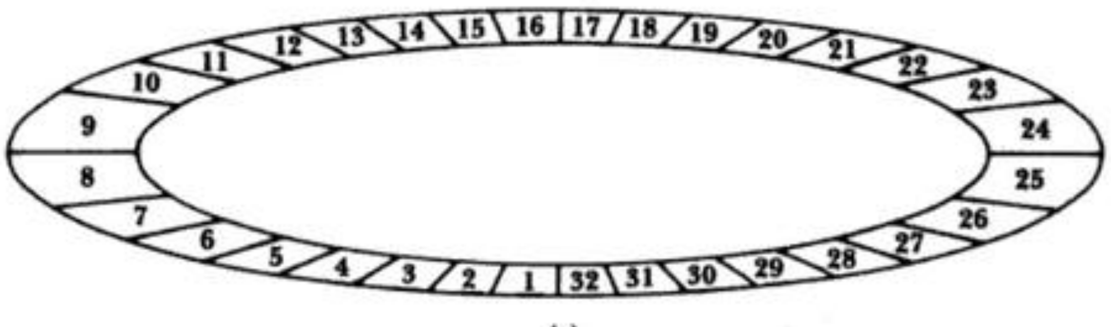
<u>Organizing tracks:</u>

There are 2 ways to organize data on a disk:
1. By sector.
2. By user defined block.

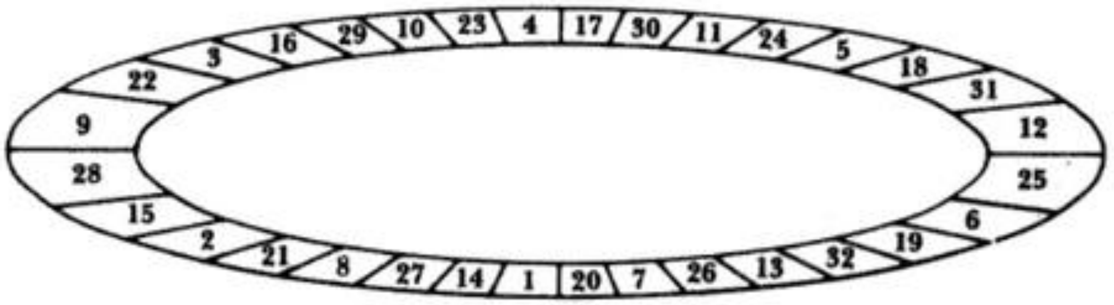
<u>The physical placement of sector:</u>

The sectors can be organized on a track in several ways. The simplest among them is that sectors are adjacent, fixed-sized segments of a track that hold a file. This is a perfect way to view a file, but not good way to store sectors physically.

When we want to read the data of a file continuously, which are stored in adjacent sectors one after the other, reading cannot be done easily. This is because after reading the data, the disk controller takes a certain amount of time to process the received information, before it accepts the next data. If the file is stored on adjacent sectors, then we would miss the start of the following sector, as the disk controller is busy processing the previously read data. To read the next sector, the disk must be revolved once. Hence only one sector can be read per revolution of the disk.



(a)



(b)

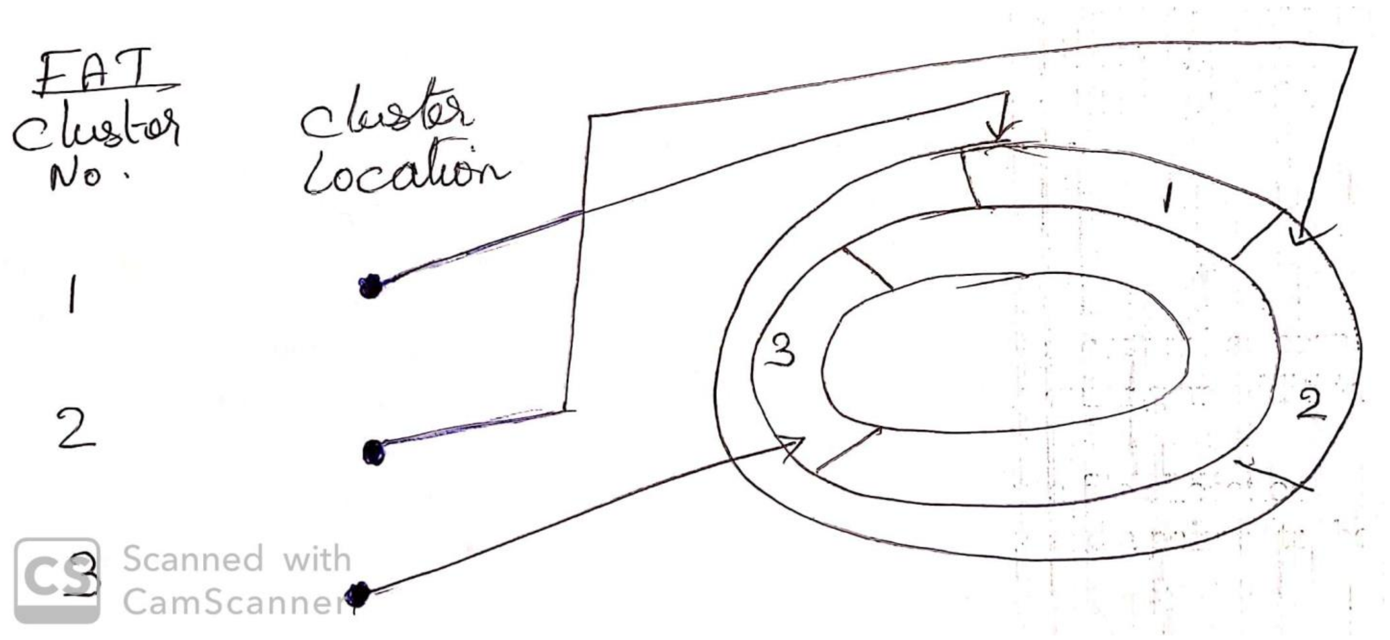
a) Adjacently arranged sectors.
b) Arranged by interleaving factor of 5.

This problem can be solved by interleaving few sectors. Several sectors are interleaved between 2 logically adjacent sectors. In the figure shown there is an interval of 5 physical sectors between the logically adjacent sectors. In the disk having interleaving factor 5, only five revolutions are required to read all the 32 sectors of a track.

<u>Clusters:</u>

A cluster is a fixed number of contiguous sectors. Once a given cluster has been found on a disk, all sectors in that cluster can be accessed without requiring an additional seek.

A part of the operating system called the file manager creates a file allocation table(FAT). The FAT contains a list of all clusters of a file, in order and a pointer to the physical location of the cluster in the disk.



Extents:

Extents of a file are the parts of a file which are stored in contiguous clusters. So that the number of seek to access a file reduces.

It is preferable to store the entire file in one extent. But this may not be possible due to non-availability of contiguous space, errors in allocated space etc. then the file is divided into two or more extents.

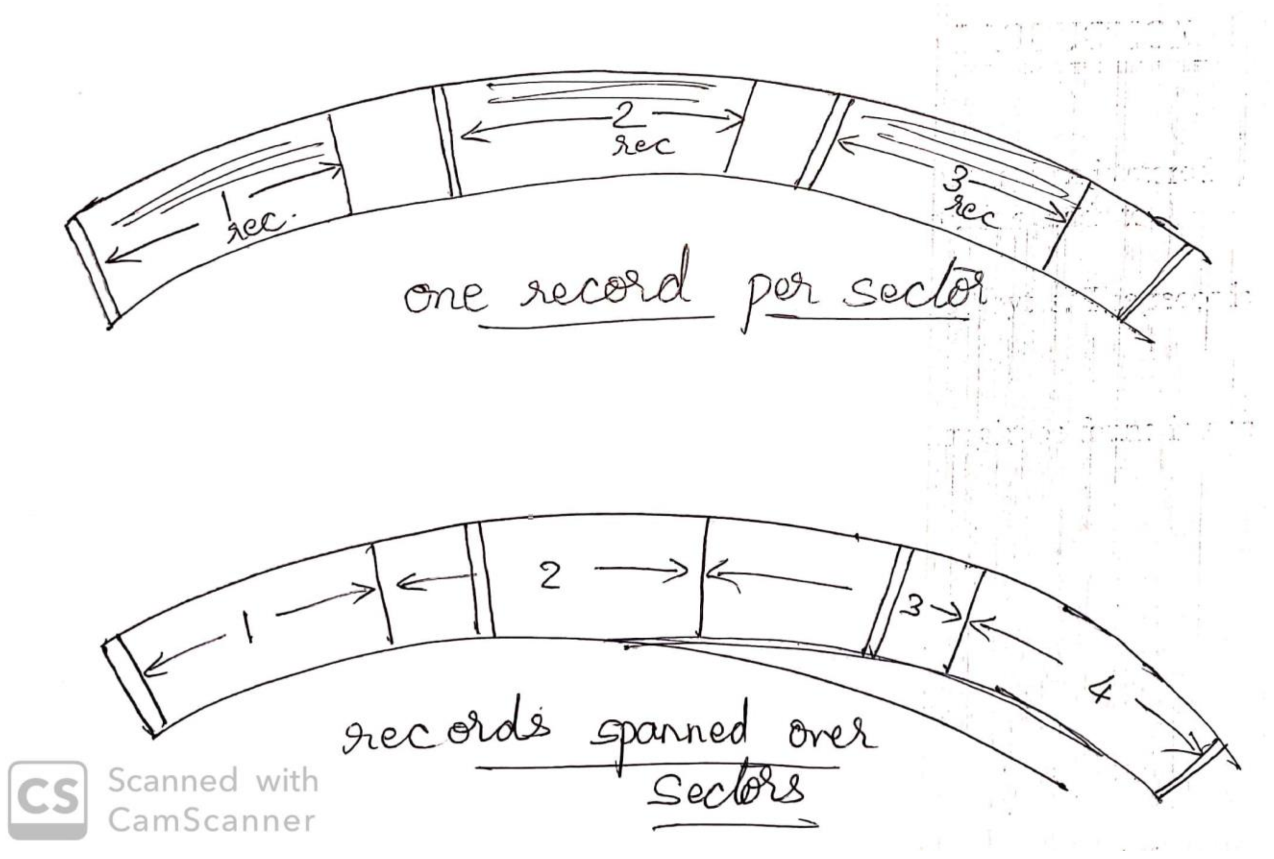
As the number of extents in a file increase, the file becomes more spread out on the disk and the amount of seeking required to process the file increases.

Fragmentation:

Since the smallest organizational unit of a disk is one sector, the data is stored interms of a multiple sectors and new file is started in a new sector, this leads to unused(waste) disk space resulting in fragmentation.

Suppose the sector size is 512 bytes and the record size is 300 bytes. The records can be stored in two ways:

- Store only one record per sector.
- Allow records to span sectors, so the beginning of a record might be found in one sector and the end of it in another.



<u>One record per sector:</u>
Advantage - records can be retrieved by retrieving just one sector.
Disadvantage - it leaves enormous amount of unused space within each sector. This loss of space is called internal fragmentation.

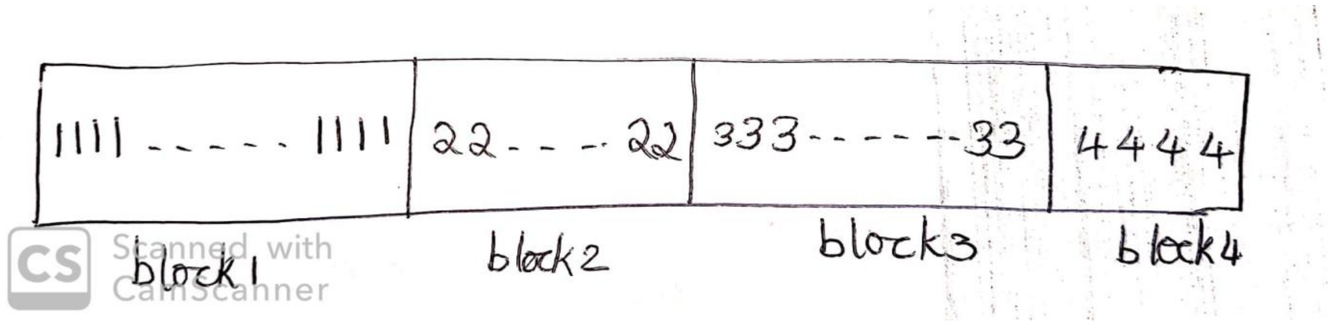
<u>Records spanned over sectors:</u>
Advantage - no loss of space due to internal fragmentation.
Disadvantage - some records can be accessed only by accessing two sectors.

Organizing tracks by block

Sometimes disk tracks are divided into integral no of user-defined blocks, where size can vary.

As the blocks are of varying size,(to fit the data) there is no sector spanning and fragmentation problems.

The term blocking factor is used to indicate the number of records that can be stored in each block.



If the file is with record size 300 bytes, then the block size will be a multiple of 300 bytes.

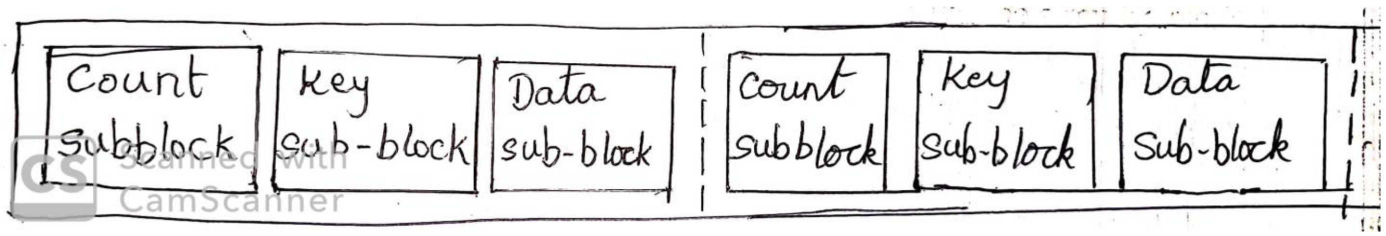
Advantages of block organization:

- No space is lost for internal fragmentation.
- No need to load 2 blocks to retrieve one record.

In this organization of tracks, each block contains one or more sub-blocks containing extra information about the data block.

The subblock may be-

- count sub-block : contains number of bytes in the data block.
- Key sub-block : key for the last record in the data block.



Non-data overhead:

I both blocks and sector organization of track, certain amount of space is taken up from the disk to store some information about storage data, this is called non-data overhead.

On sector addressable disks, non-data overhead contains information such as sector address, block address, sector condition, synchronization marks between fields of information etc. This non-data overhead is of no concern to the programmer.

On a block-addressable disk, some non-data overhead is considered by the programmer. Since sub-block and inter-block gaps have to be provided in every block, there is more non-data overhead with blocks than with sector organization.

Note: If blocking factor is large, more no of records can be stored. But larger blocking factor may lead to fragmentation within a track.

Problem 2: Suppose a block-addressable disk drive is 20,000 bytes per track and the amount of space taken up by sub-blocks and inter-block gaps is equivalent to 300 bytes per block. If a file containing 100 bytes record is to be stored. How many records can be stored per track if the blocking factor is 10? If it is 60?

Solution:

If blocking factor is 10, there are 10 records per block.
Sub block & inter block gap = 300 bytes
Record size – 100 B

So block size = (100 x 10) + 300
= 1300 bytes (1 block size)

So a track of 20,000 bytes can have, 20000 / 1300 = 15.38 blocks
~ 15 blocks

So no. of records in a track = 15 x 10 = 150 records.

b)

If blocking factor is 60, there are 60 records per block.
So block size = (100 x 60) + 300
= 6300 bytes (1 block size)

So a track of 20,000 bytes can have, 20000 / 6300 = 3 blocks

So no. of records in a track = 3 x 60 = 180 records.

<u>The cost of disk acces-</u>

To access a file from the disk, the total amount of time needed can be divided based on three operations. The time needed id seek time+ rotational delay + transfer time.

a) <u>Seek time:</u>

The time taken to move the read/write head to the required cylinder, is called seek time.

The amount of seek time spent depends on how far the read/write head has to move.

If we are accessing a file sequentially and the file is packed into several consecutive cylinders, then the seek time required for consecutive access of data is less.

Since the seek time required for each file operation various, usually the average seek time is determined.

Most hard disks available today have an average seek time of less than 10 milliseconds.

b) <u>Rotational delay:</u>

The time taken for a disk to rotate and bring the required sector under read/write head is called rotational delay.

On average, this is considered to be half of the time taken for one rotation. If a hard disk rotates at 5000rpm, then it takes 12msec to complete one rotation, so rotational delay is 6msec.

c) <u>Transfer time:</u>

The time required to transfer one byte of data from track to read/write head or vice versa, is called Transfer time.

The transfer time is given by:

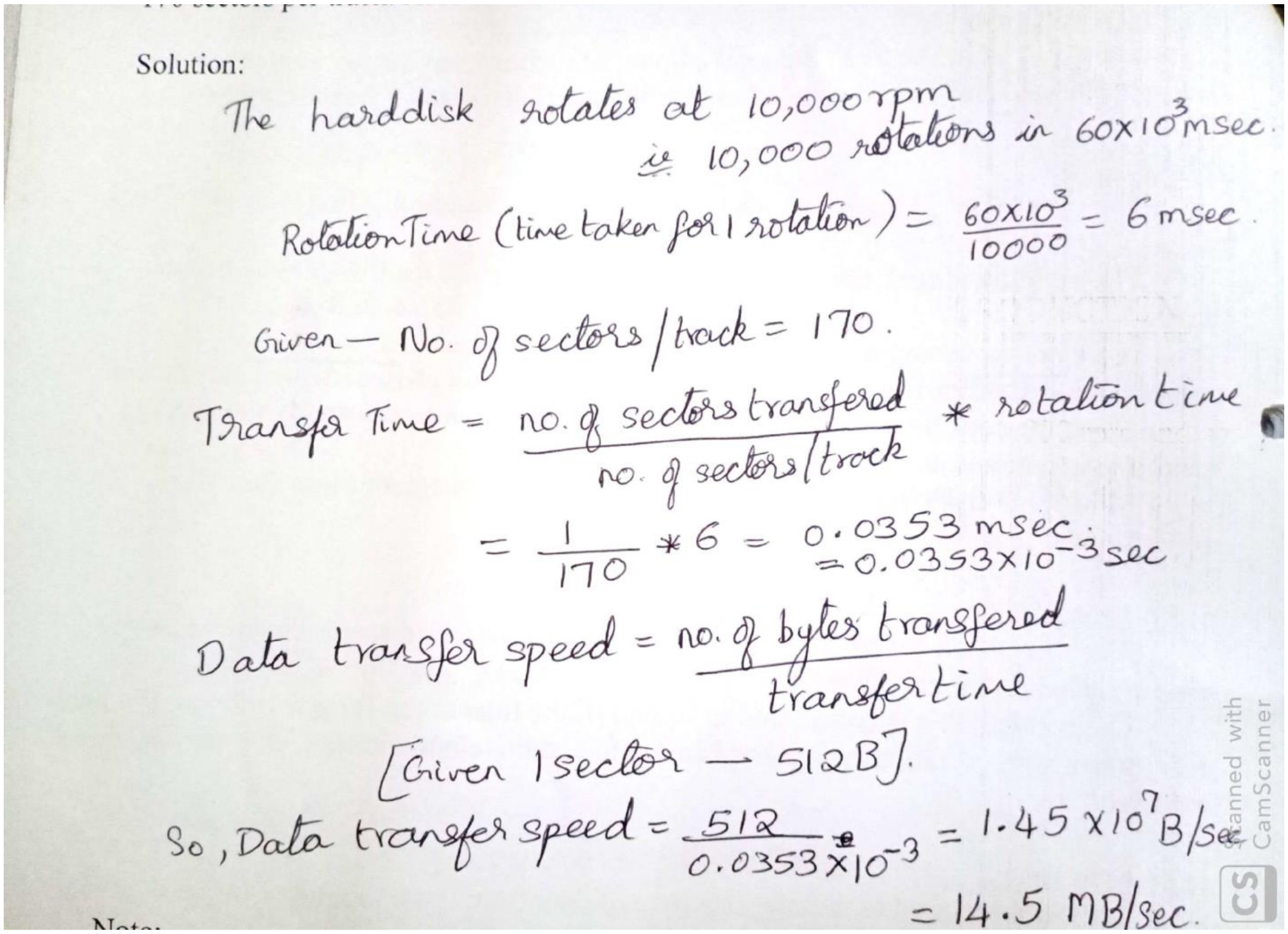
Transfer time = (no of bytes transferred/ no of bytes on a track)* rotation time.
where, rotation time is time taken for one rotation

Data transfer speed = no of bytes transferred/ transfer time.

$$\text{Transfer time (in sectors)} = \frac{\text{no. of sectors transferred}}{\text{No. of sectors/ track}} \times \text{rotation time}$$

Note:
1) Sequential access of file takes less time to transfer data when compared to random access.
2) Seek time is more in random access.

Problem 3:
Calculate the data transfer speed of the hard disk if it has a speed of 10,000 rpm and has 170 sectors per track. Assume that a sector can store 512B of data.



Solution:

The harddisk rotates at 10,000 rpm
i.e 10,000 rotations in 60×10^3 msec.

Rotation Time (time taken for 1 rotation) $= \frac{60 \times 10^3}{10000} = 6$ msec.

Given — No. of sectors/track = 170.

Transfer Time $= \frac{\text{no. of sectors transfered}}{\text{no. of sectors/track}} * \text{rotation time}$

$= \frac{1}{170} * 6 = 0.0353$ msec.
$= 0.0353 \times 10^{-3}$ sec

Data transfer speed $= \frac{\text{no. of bytes transfered}}{\text{transfer time}}$

[Given 1 sector — 512B].

So, Data transfer speed $= \frac{512}{0.0353 \times 10^{-3}} = 1.45 \times 10^7$ B/sec
$= 14.5$ MB/sec.

Disk as Bottleneck

The disk transfer time is much slower when compared to the network and computer CPU. So the network or CPU has to wait for long time for the disk to transmit data.

A number of techniques are used to solve this problem one among those is multiprogramming, in which the CPU works on other jobs while waiting for the data to arrive from the disk.

Another technique is striping, Disk striping involves splitting the parts of a file on several different drives, then letting the separate driver to deliver the part of the file to network simultaneously. This improves the throughput of the disk.

Another approach to solve the disk bottleneck is to avoid accessing the disk. As the cost of memory is steadily decreasing, more programmers are using memory to hold data. Two ways of using memory instead of secondary storage are memory disks and disk caches.

A RAM disk is a large part of memory configured to simulate the behavior of mechanical disk, other than speed and validity.

The data in RAM disk can be accessed much faster than disks i.e. without a seek or relational delay. But here the memory is volatile, the content of RAM disk are lost when the computer turn off.

A Disk cache is a large block of memory configured to contain pages of data from a disk when data is requested by a program, the file manager first looks into the disk cache, to check if it contains the page with the requested data. If it contains data, it is processed immediately. Otherwise, the file manager reads the page from the disks, replacing the page in disk cache.

RAM disks and cache memory are example of buffering

Magnetic Tape

Magnetic tape is a device that provides no direct (random) accessing facility, but can provide very rapid sequential access to data.

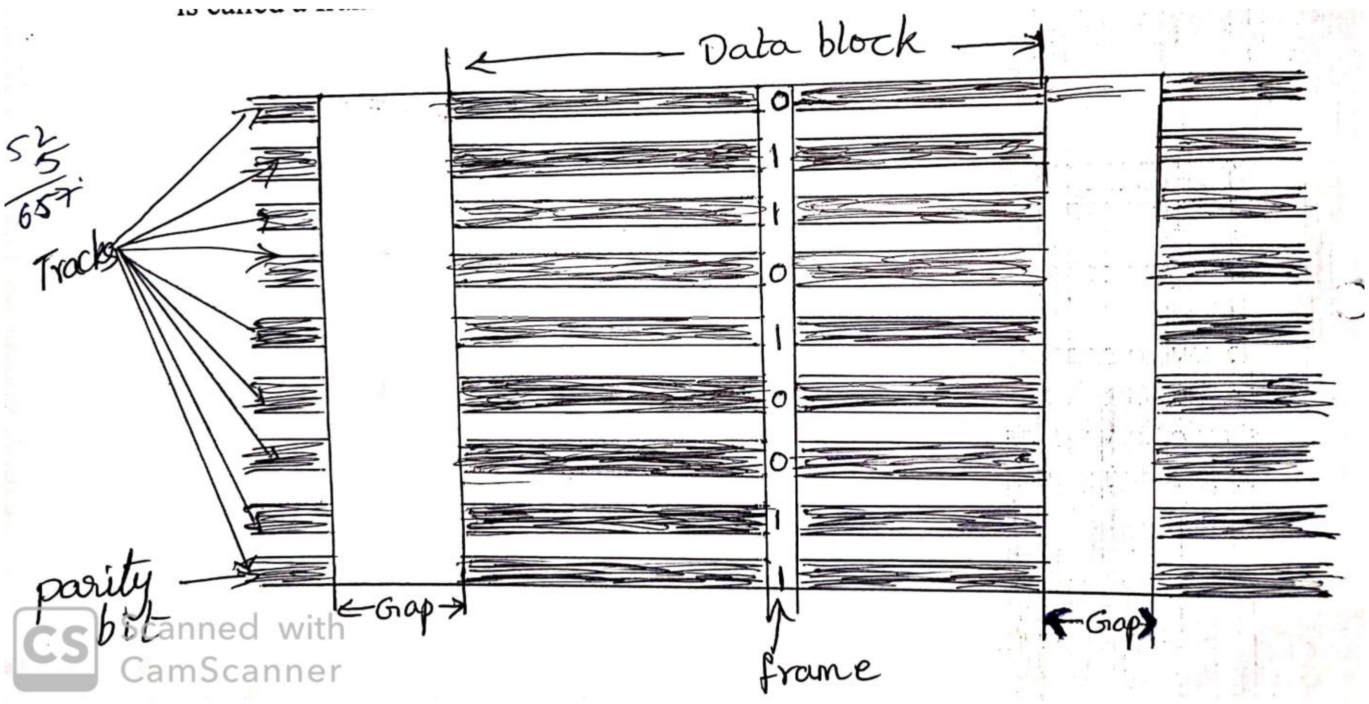
Tapes are compact, works in different environmental conditions, are easy to store and transport and are less expensive than disks. Tapes are commonly used as a backup device.

Organization of Data on Nine-Track Tapes

On tapes, there is no need of addresses to identify the location of data, as the data can be accessed sequentially only.

On tapes, the logical position of a byte with in a file corresponds directly to its physical position from start of the file.

The surface of a typical tape is a set of parallel tracks, each of which is a sequence of bits. If there are nine tracks, the nine bits, at the corresponding position in the nine tracks are considered as 1 byte, plus a parity bit. So a byte of data in tape is one-bit wide slice of tape. Such a slice is called a frame.



The parity bit is not part of the data, but it is used to check the validity of the data. If odd parity is set for tape, this bit is set to make the number of 1 bit in the frame as odd.

Frames are grouped into <u>data blocks</u> whose size can vary from a few bytes to many kilobytes, depending on the needs of the user. Blocks are separated <u>inter block gaps</u>, which does not contain any information.

The performance of tape driver is measured in terms of 3 quantities –

1) Tape density – in bits per inch(bpi) per track , commonly 800,1600,6250 or 30000 bpi
2) Tape Speed – in inches per second (ips), commonly 30 to 200 ips.
3) Size of inter block gap – in inches commonly between 0.3-0.75 inches.

The space required to store a file in tape is calculator using the formula,

S=n x (b + g)

Where, s – space required

n – Number of data blocks (1 or more record)

b – Physical length of the data block

g – Length of an inter block gap

$$b = \frac{\text{data block size}}{\text{Tape density}}$$

Problem –

Suppose a back-up copy of large mailing-list file is to be stored. It has one record. If the file is stored on a 6250 bpi tape that has an inter block gap of 0.3 inches and blocking factor is 50, how much tape is needed?

Solution: blocking factor-50 i.e. 50 record in 1 block

So block size=50 x 100=5000bytes.

Tape density=6250 dpi

Inter block gap (g) =0.3 inches

Number of data block (n) =1000000/50=20000

$$b = \frac{\text{data block size}}{\text{Tape density}} = \frac{5000 \text{ x } 8}{6250} = 6.4 \text{ inches.}$$

S=n x (b+g)

= 20000 x (6.4 +0.3)

= 1, 34,000 inches [12 inches = 1 feet]

= 11,166.67 feet

So a tape of 11166.67 feet is used to store 1 million records.

Estimating Data Transmission Times

Transmission rate is affected by tape density and tape speed. Transmission rate can be calculated using,

transmission rate=tape density (bpi) x tape speed (ips).

$$\text{Effective recording density of tape} = \frac{\text{No. of bytes/block}}{\text{No. of inches req. to store a block (b+g)}}$$

<u>Disk Vs Tape</u>

In past, both disk and tapes were used for secondary storage. Disks were proffered for random access and tape for sequential access.

Now, disks have over much of secondary storage because of decreased cost of disks and memory storage tapes are used as tertiary storage.

<u>Introduction to CD-ROM</u>

CD-ROM is an acronym for compact disk Read Only Memory .A single disk can hold more than 600 MB of data.CD-ROM is read only i.e. it is publishing medium rather than a data storage & retrieval like magnetic disks.

CD- ROM Strengths – high storage capacity, inexpensive price, durability

CD-ROM weakness – extremely slow performance, this makes intelligent file structure difficult.

<u>History of CD-ROM</u>

CD-ROM is used to store any kind of digital information (text, audio and Video).

CD-ROM is the offspring of videodisc technology develop in late 1960's. Different companies developed a number of methods to store video signals, the outcome was the laser vision format of string data in analog form. The laser vision format support recording in both a constant linear velocity (CLV) format – that maximizes storage capacity and a constant angular velocity (CAV) format – that enables fast seek performance.

The Philips and Sony worked on a way to store music on optical disc in digital data format rather than the analog format used earlier.

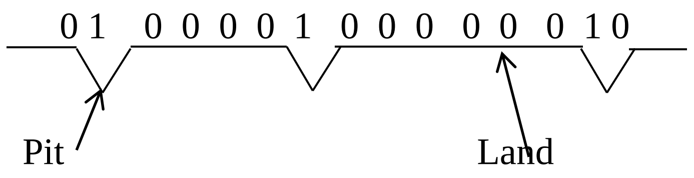
Thus the CD audio appeared in 1984. CD-ROM was built using the same technology of CD audio. The CD-ROM drive appeared in 1985. It was a read only device. But with the introduction of CD-RW (compact disk re-writable) in 1997, the use of CD has been widespread.

<u>Physical Organization of CD-ROM</u>

CD-ROM is descendent of CD-Audio, in which data was accessed sequentially.

Reading Pits & Lands- (It is made up of polycarbonate plastic a thin layer of aluminum to make reflection surface)

- CD-ROM's are stamped from a glass master disk which has coating that is changed by the laser beam. When the coating is developed, the areas hit by the laser beam turn into pits and the smooth, unchanged areas between the pits are called lands.
- When the stamped copy of the disk is read, we focus a beam of laser light on the track. The pits scatters the light, but the land reflects back most of the light. This alternating pattern of high and low intensity the original digital information.
- 1's are represented by the transition from land to pit and back again. 0's are represented by the amount of time between transitions. The longer between transitions, the more 0's we have.



The encoding scheme used is such that, it is not possible to have two adjacent 1's.1's are always separated by 2 or more 0's. The data read from track has to be translated to 8-bit pattern of 1's and 0's to get back the original data

The encoding scheme called EFM encoding (Eight to Fourteen Modulation), which is done through a look up table turn the original 8-bit of data into 14 expanded bits, that is represented as pits and lands on the disk. The reading

Decimal Value	Original bits	Translated bits
0	0000 0000	0100100 0100000
1	0000 0001	1000010 0000000
2	0000 0010	1001000 0100000
3	0000 0011	1000100 0100000
4 and so on…..	0000 0100	0100010 0000000

process from the disk, reverses the translation

The below look up table shows a pattern of EFM encoding

- Since 0’s are represented by the length of time between transitions, the disk must be rotated at a precise and constant speed. This affects the CD-ROM driver’s ability.

<u>CLV instead of CAV</u>

- Data on a CD-ROM is stored on spiral tracks. Each track is divided into sectors and data is read/write sector wise.
- There are basically two ways of storing information on disk, namely
  1) Constant Angular Velocity (CAV)
  2) Constant Linear Velocity (CLV)

- In constant angular velocity, the tracks are concentric and sectors are pie-shaped. It writes data less densely in the outer tracks than in the center tracks, as there is equal amount of data in all the sectors. This leads to wasting of storage capacity in outer tracks but have the advantage of being able to spin the disk at the same speed for all positions of read/write head.

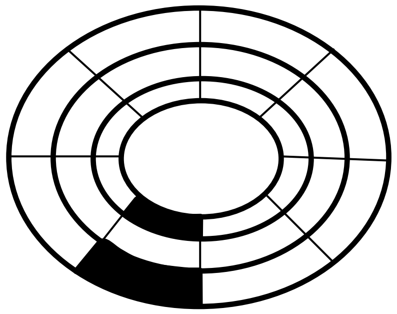
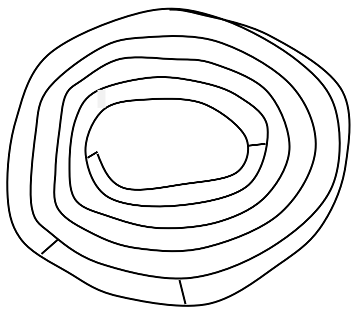
- In constant linear velocity (CLV), sector towards the center of the disk takes the same amount of space as a sector towards the outer edge of desk. Hence data can be stored in maximum density in all the sectors.

  Since reading the date requires that it has to pass under the optical pick up device at constant rate, the disk has to spin more slowly when we are reading at the outer edges than when we are reading towards the center of the disk. Hence the disk moves at varying speed.

CLV is used in CD-ROM instead of CAV because CLV gives much better storage capacity even through varying speed movements of read/write head may cause some difficulties.

Differentiate between CAV and CLV

CAV (content Angular Velocity)	CLV (Constant Linear velocity)

• Disk is divided into pie shaped sector • Data is densely packed at center & loosely packed at the ends • Wastage of disk space at outer edge • Disk spins at a content speed • Lesser storage space than architecture • 	• Disk is divided into sectors of equal physical size • Data is densely packed all over the disk • No wastage of space • Disk spins more slowly when reading at outer edge • Much better storage space 

CD-ROM Strength and weaknesses

1) Seek Performance

The chief weakness of CD-ROM is the random-access performance. Current magnetic disk has an average random access time of about 30msec (Combing seek time & rotational delay). On CD-ROM this access takes 500msec.

2) Data Transfer Rate – A CD-ROM drive reads seventy sectors or 1540 KB of data per second. It is about five times faster than the transfer rate of floppy disks. The transfer rate is faster when compared to the CD-ROM's seek performance.

3) Storage capacity – A CD-ROM holds more than 600MB of data. This is a huge amount of memory for storage. Many typical text databases and document collections stored on CD-ROM use only a fraction memory. With such large capacity it enables us to build indexes and other structures that can help us to overcome the disadvantages of seek performance of CD-ROM.

4) **<u>Read-Only Access</u>**- CD-ROM is publishing medium, a storage device that cannot be changed after manufacturer develops it. The advantage of this is that, the user need not worry about the updating. This simplifies some of the file structures.

5) **<u>Asymmetric writing and Reading</u>** – with CD-ROM, we create files are placed on the disk once and access the file content thousands or millions of times. With intelligent and carefully designed file structure built once, the user can enjoy the benefit of this investment again and again.

<u>Storage as a Hierarchy</u>

There are different types of storage devices of different speed, capacity and cost. The users can select the device depending on their need.

The different storage devices can be summarized as follows-

Types of Memory	Devices & media	Access times (sec)	Capacity (bytes)	Cost (cost/bits)
(primary) -Registers -memory -ram disk -disk cache	Semiconductors	10^{-9}-10^{-5}	10^{0}-10^{9}	10^{0}-10^{-3}
(secondary) -Direct Access	Magnetic disk	10^{-3}-10^{-1}	10^{4}-10^{9}	10^{-2}-10^{-5}
-serial	Tape and mass storag	10^{1}-10^{2}	10^{0}-10^{11}	10^{-5}-10^{-7}
(offline) Archive And Back	Removable Magnetic disks, Optical disks, & tapes	10^{0}-10^{2}	10^{4}-10^{12}	10^{-5}-10^{-7}

A journey of a Byte

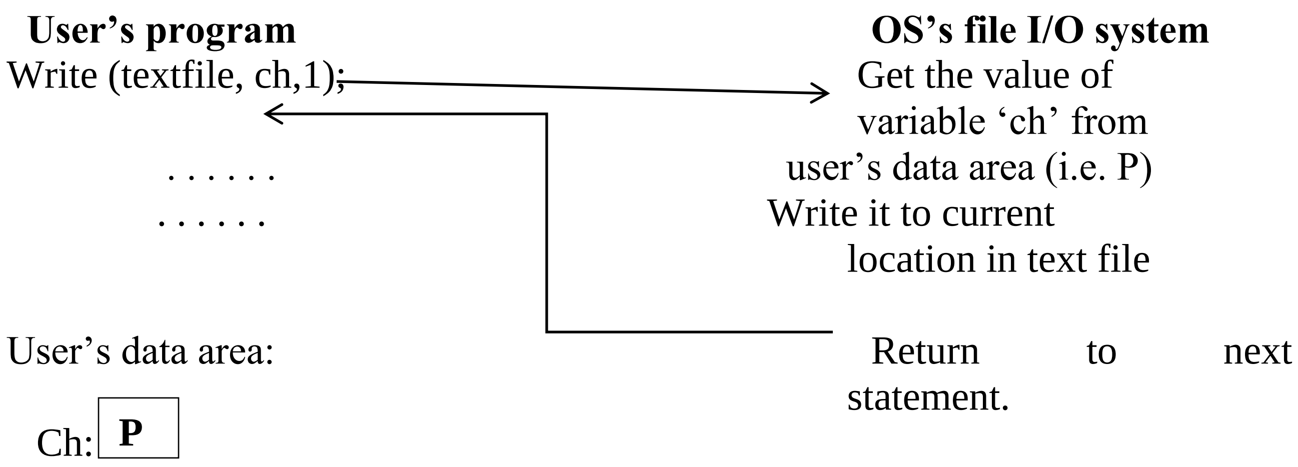
How a byte is stored from a program,

write (textfile,ch,1); //write value of ch to hard disk

calls the operating system. The operating system invokes the file manager, an OS program which deals with the file – related matter and I/O devices.

The file manager does the following tasks when write operation is requested

- Checks whether the operation requested write is permitted.
- Locates the physical location where the byte has to be stored (i.e. locate drive, cylinder, track & sector)
- Finds out whether the sector to store the character ('ch') is already in memory, if not call I/O Buffer.
- Puts 'ch' in the buffer
- Keeps the sector in memory to see if more operations are to be done in the same sector.


User's program
OS's file I/O system
Write (textfile, ch,1);
. . . . . .
. . . . . .
User's data area:
Ch: P
Get the value of variable 'ch' from user's data area (i.e. P)
Write it to current location in text file
Return to next statement.

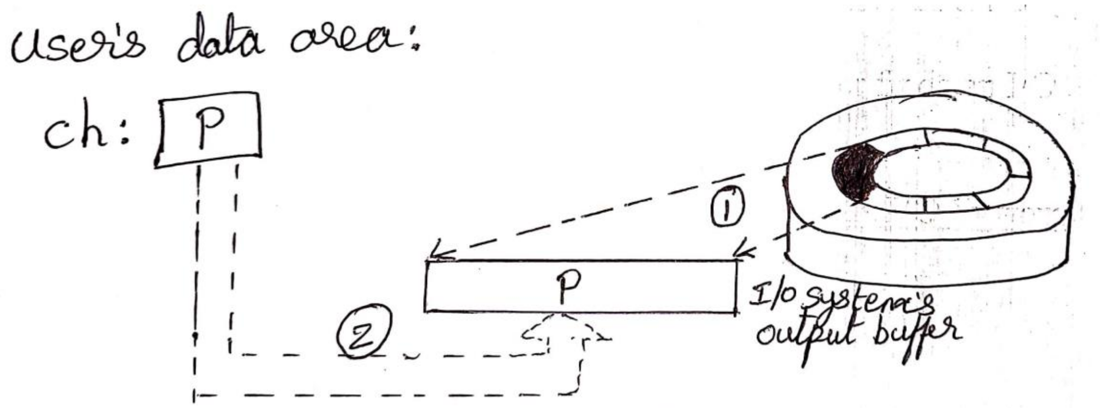
The I/O Buffer

To read/write data to a text file, stored in secondary storage, the sector where the data has to be stored must be in the buffer in primary memory.

If such sector is not found in the buffer, the file manager finds an I/O Buffer space and reads the sector from the disk.

User's Program	**File I/O System**

Write (textFile,ch,1 …………. …………..	If necessary, load the sector from text f the system output buffer. Move ‘ P’ into the buffer.



Here, The filemanager moves P from the program's data area to the system output buffer

The I/O Processor

There is large difference in transmission speed & data path width between the CPU and external devices. The process disassembling and assembling groups of byte for transmission to and from external devices, is done by a special purpose device called I/O processor.

Disk Controller

The job of controlling the operation of the disk is done by disk controller.

- The I/O processor asks the disk controller if the disk drive is available for waiting
- Disk controller instructs the disk drive to move its read/write head to the right track and right sector.
- Disk spins to right location and byte is written

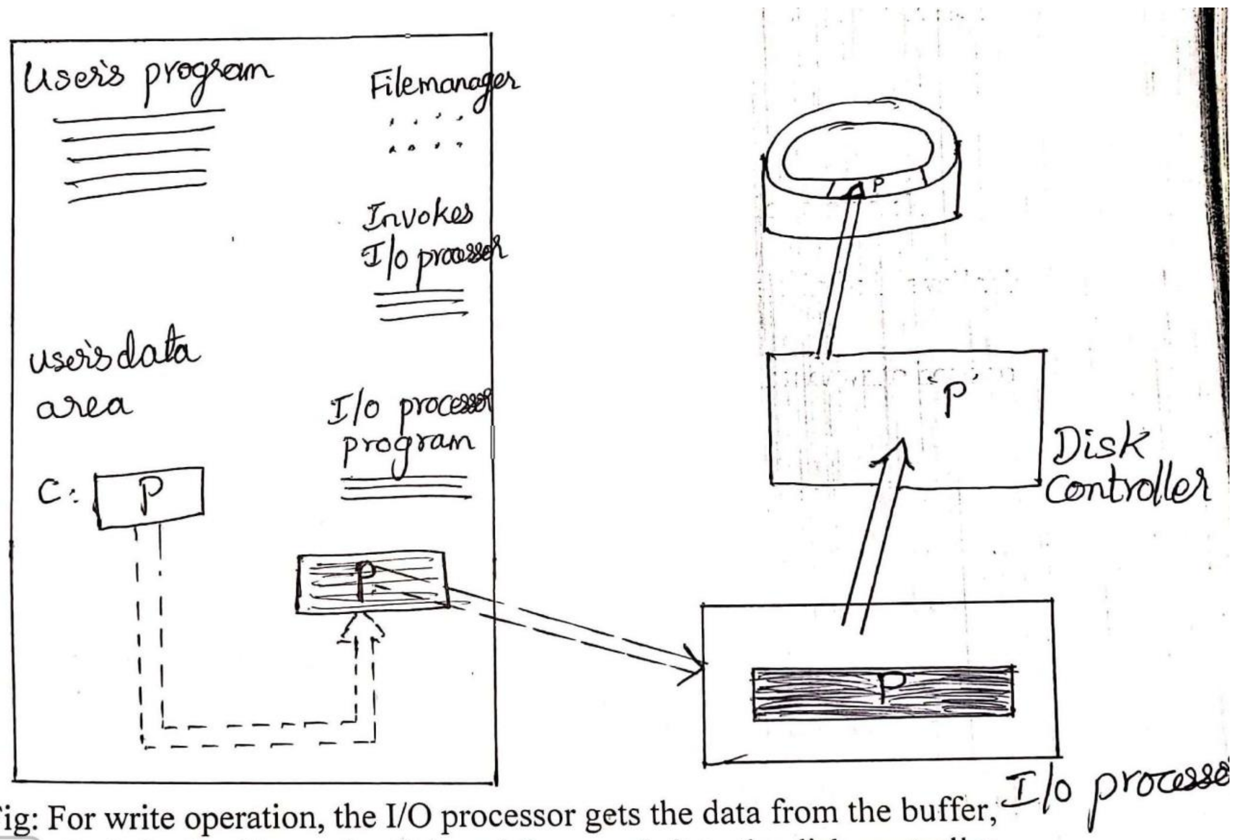


Fig: For write operation, the I/O processor gets the data from the buffer, prepares it for storing on the disk and then sends it to the disk controller, which deposits it on the surface of the disk.

Buffer Management

Use of Buffer – Buffering involves working with large chunks of data in memory, so that the number of access to secondary storage can be reduced.

Buffer Bottleneck –

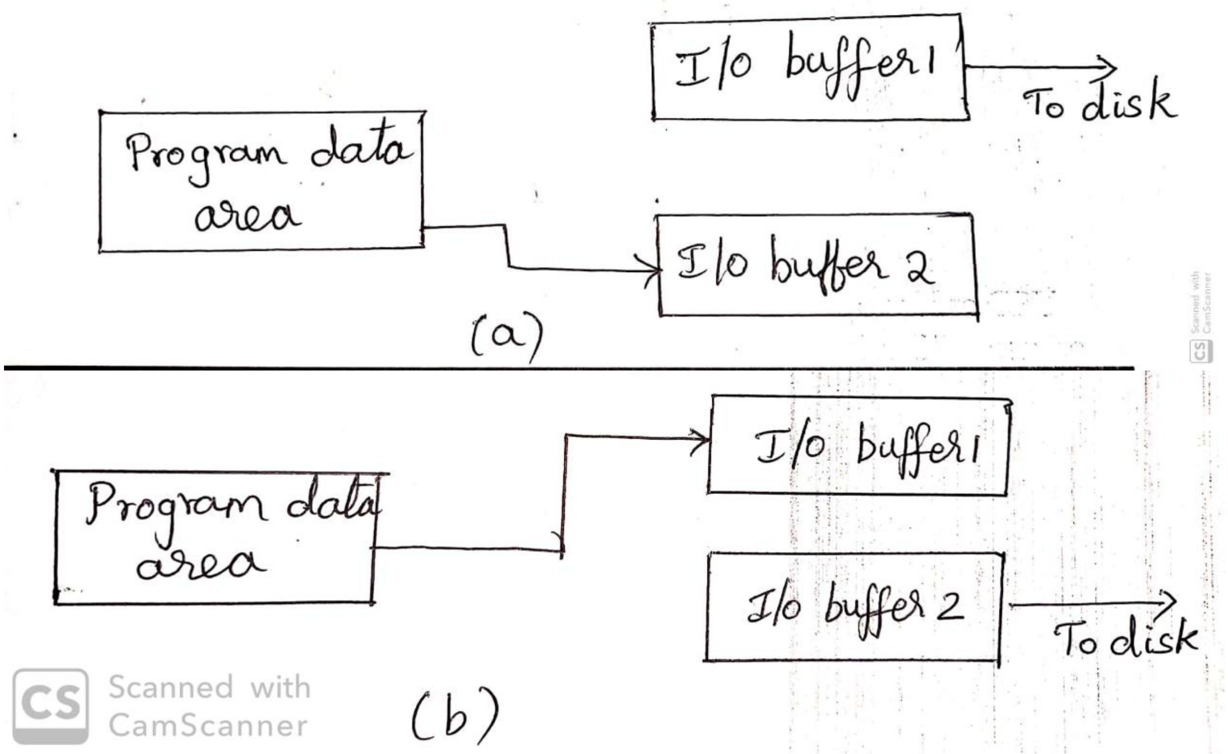
- Assume that the system has a single buffer and is performing input and output on one character at a time alternatively.
- In this case, the sector containing the character to be read is constantly overwritten by the sector containing spot where the character has to be written and vice versa.
- In such a case, the system needs more than one buffer
- Moving data to and from disk is very slow and programs may become I/O bound. Therefore we need to find better strategies to avoid this problem.

Some **Buffering Strategies** –

- Multiple Buffering
    - Double Buffering
    - Buffer pooling
- Move mode and locate mode
- Scatter/Gather I/O

Multiple Buffering –

Reading or writing from a disk is time consuming. To utilize CPU efficiently buffers are maintained. If two buffers are used, the CPU can be filling one buffer while the contents of the other are being transmitted to disk. When both the tasks are finished, the roles of the buffers can be exchanged. This method of swapping the roles of 2 buffers after each operation is called double buffering.



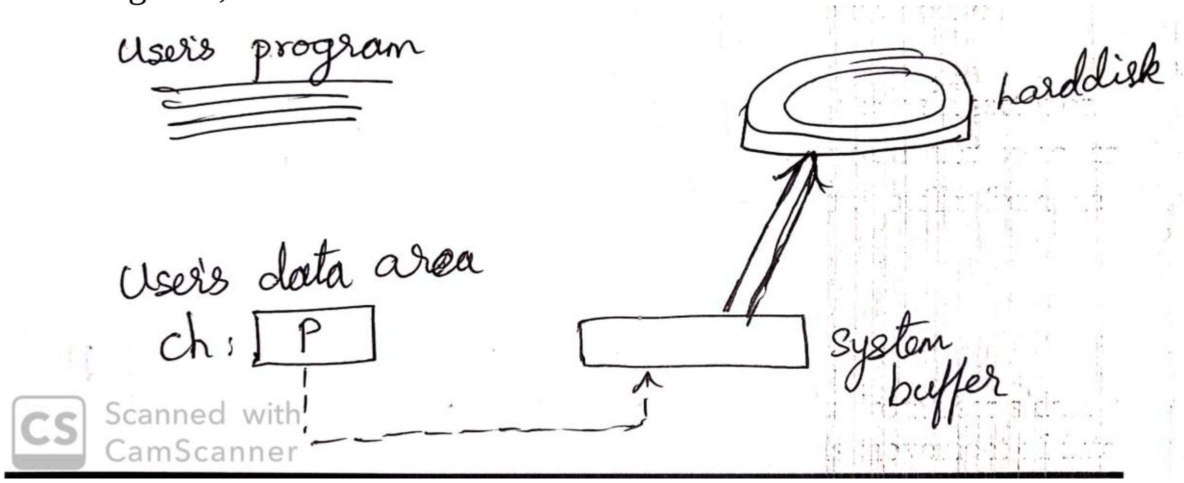
Double buffering allows the OS to operate on one buffer is being loaded or emptied. In (a) the contents of system I/O buffer1 are sent to disk while I/O buffer 2 is being filled and (b) the contents of buffer 2 are sent to disk while I/O buffer 1 is being filled.

Buffer pooling

- When system buffer is needed, it is taken from a pool of available buffers and used.
- When the system receives a request to read a certain sector or block, it search to find if any buffer in the block contains that sector or block. If no buffer contains it, the system finds a free buffer from the pool and loads the sector or block into it.

Move Mode and Locate Mode

For a program to read/write contents a file, the data from user's data area has to move to the system buffer (or vice versa). This movement of data takes some time. This method of moving chunks of data from one place to another in memory, before accessing data, is called move mode.



The move mode can be avoided in 2 ways, called locate mode.

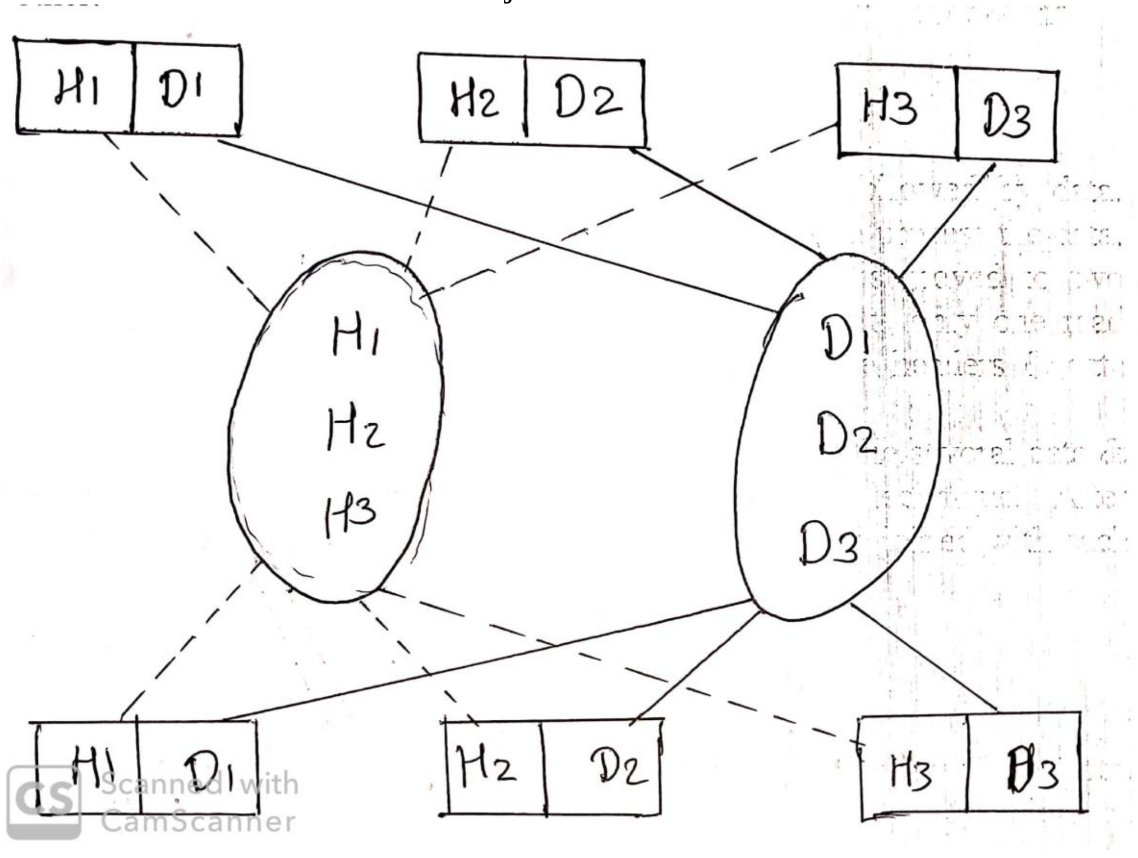
1) The file manager can directly perform I/O operations between secondary storage and user's data area.
2) The file manager can locate the data from system buffer using pointers

In locate mode, there is no transfer of data b/n the system buffer & the data area.

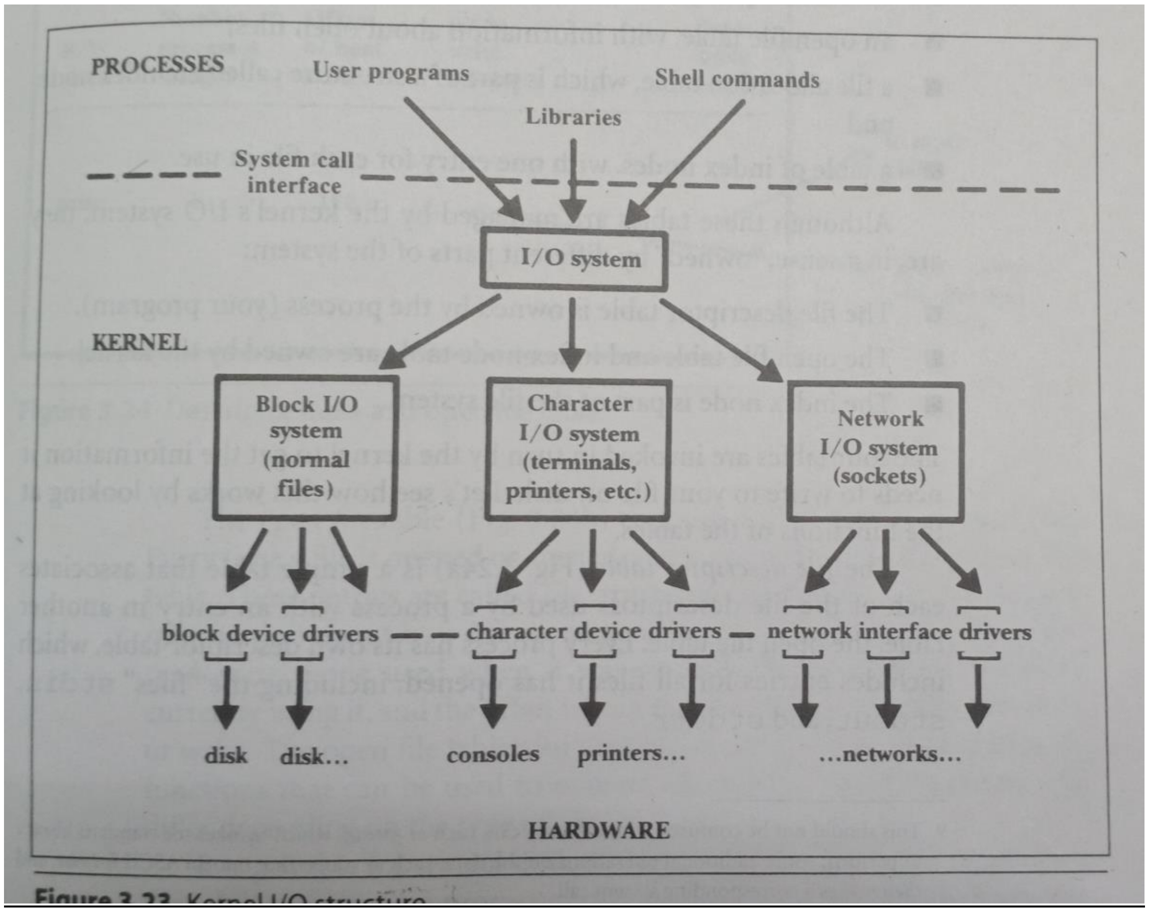
Scatter/Gather I/O

Each block in memory consists of header followed by data. Suppose many blocks of a file read at ones. In order to process the data, the headers of each block and data of each block is moved to two separate buffers. So to read the data of all the blocks only one read operation is required. The technique of separating the headers & data blocks is called 'Scatter'

The reverse of scatter input is gather output. the several data & header block are arranged separately in gather technique. After processing the data, the data and the header blocks are joined with each other.



<u>I/O in Unix</u>



The above diagram shows the process of transmitting data from a program to an external device. The topmost layer deals with data in logical terms. The below layer's carry out the task of turning the logical object into a collection of bits on a physical device. This layer is called the kernel.

The top layer consists of processes, associated with solving some problems using **shell commands**(like cat, tail, ls etc), **user programs** that operate on files, and **library routines** like scanf and so on. Below this layer is the unix kernel, which consists of all the rest of the layers.

In UNIX all the operations below the top layer are independent of applications.

Consider the example of writing a character to disk

```
Write (fd,&ch,1);
```

When this system call is executed kernel is invoked immediately. This system call instructs the kernel to write a character to a file.

The kernel I/O System connects the file descriptor (fd) in program to some file in the file system. It does this by proceeding through a series of four tables that enables the kernel to find the file in the disk.

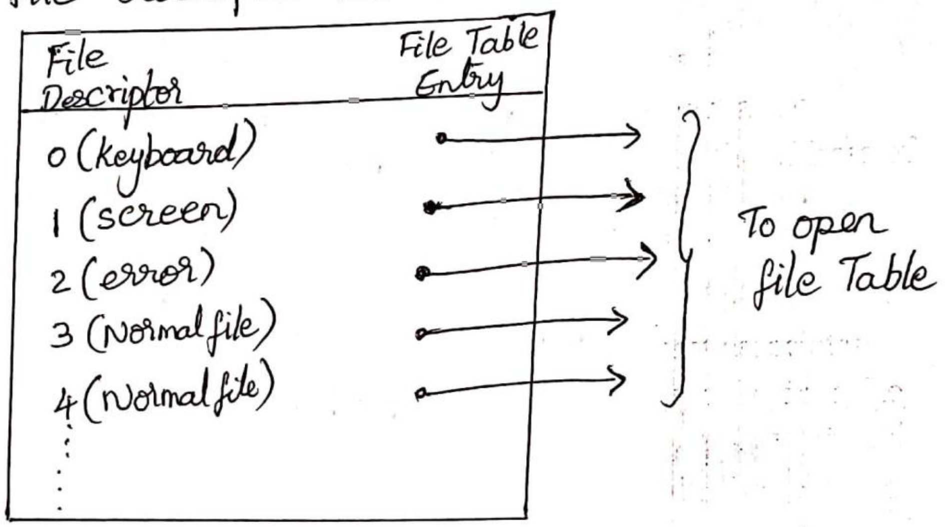
The four tables are –

- A file descriptor table
- An open file table, write information about open files
- A table of index nodes, with detailed information about a file
- A file allocation table, this is part of I node & contains the address of each cluster

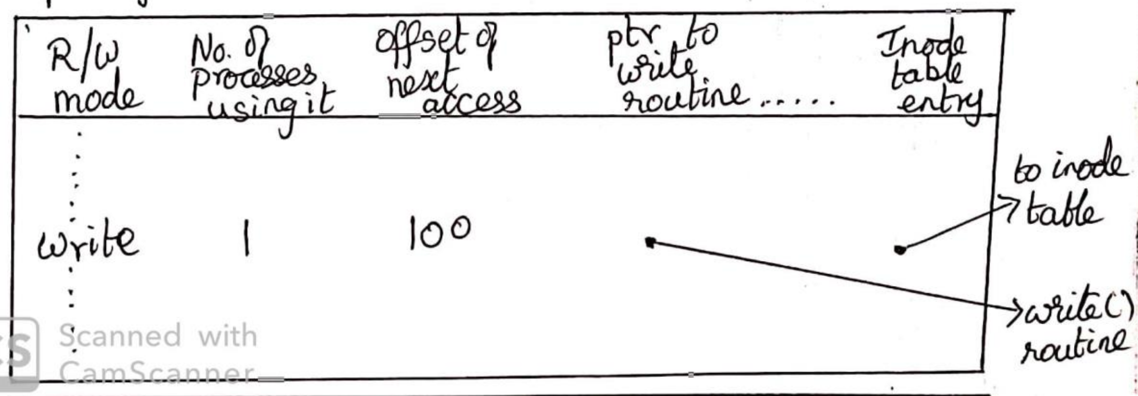
<u>Table involved by kernel to use files-</u>

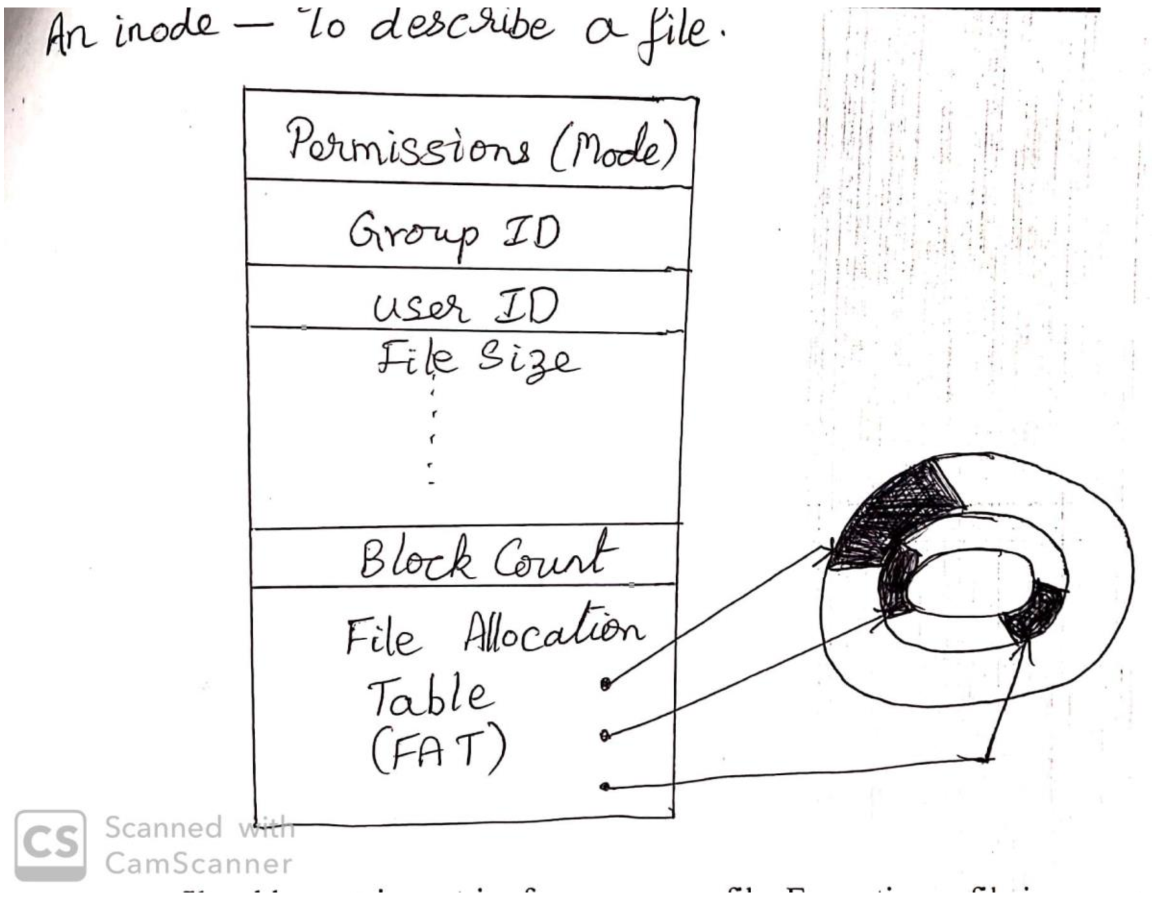
The file descriptor table is a simple table that associates each of file descriptor used by a process with an entry in the open file table. Each process has its own descriptor table.

a) File Descriptor Table



b) Open file table





The open file table contains entries for every file open in the system. Every time a file is opened or created, a new entry is added to this table. It contains information about the file, such as –

- mode in which file is opened
- number of processes currently accessing that file
- the offset within the file where the pointer is pointing for the next read or write operation
- array of pointers to generic functions (generally used functions)
- pointer to file's inode table

More information about the file is present in inode table. The files inode table will be kept on the disk with the file. It contains information like the permission given to the file while creations, owner's id, file size, no. of blocks used by the file, and the File Allocation Table.

File Allocation table (FAT) is within the inode table it associates the clusters of the file with pointer locating that address of the clusters.

Linking Filenames to Files

All file path starts with a directory. A directory is just a small file that contains many files where each file name is associated with pointer to files inode on disk. This pointer from a directory to the inode of a file is called hard link. It provides direct information about the file.

It is possible for a file to be saved in different names, in such case, all such filenames point to the same inode and there are many hard links to the same file. A field in the inode tells how many hard links are there to the inode. When a file name is deleted and there are other file names for the same file, the file itself is not deleted; it's inode's hardlink count is decremented.

A soft link or symbolic link, links a filename to another filename or path rather than pointing to inode. Hence when the original file is deleted, the inode is also deleted and symbolic link becomes dangling.

Types of Files

There are types of file –
Normal files – are normal text or program files
Special files – are files that drive some device, such as line printer or graphics device…(device drivers).
Socket – are abstractions that serve as end points for inter process communication.

Device Drivers

For every peripheral device, there is a separate set of routines called device driver. It performs the I/O operations between I/O buffer and the device.

Fundamental File Structure Concepts

A field is the smallest logically meaningful unit of information in a file.

A stream file.
Operator << is an overloaded function to write the fields to a file as stream of bytes.

```
Eg:     fstream fp1;
fp1.open(filename,ios::out);
fp1<<name<<usn;
```

Field Structures (Organizing fields in a record):

Different ways of adding structure to files to maintain the identity of fields:

1. Fix the length of the field.
2. Begin each field with length indicator.
3. Place a delimiter at the end of each field to separate it from the next field.
4. Use a “keyword=value” expression to identify each field and its contents.

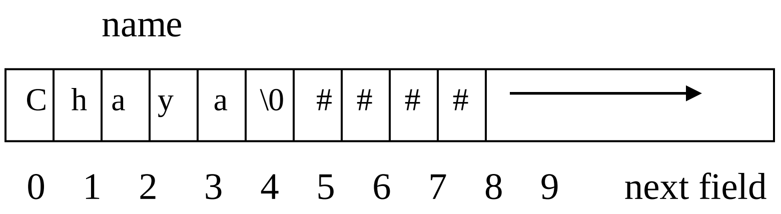
Method 1: Fix the length of the fields.

Each variable is stored as an array of characters of fixed length.

Suppose name [10] is declared.

The size of the array must be one larger than the longest string it can hold.

If the name is “Chaya”, this occupies only 5 space, the rest 5 space is occupied by padding with a symbol like ‘#’.



In this method, fields are organized by limiting the maximum size of each field. This is called as **fixed-length field**.

As the fields are of predictable length, we can access them from a file by counting the number of characters or till ‘#’ appears.

The disadvantages of this method:

1. Padding is required to bring the fields up to a fixed length, which makes the file larger.
2. Data might be lost, if it is not able to fit into the allocated space.

Due to the above mentioned problems, the fixed field approach of structuring data fields is inappropriate for data that contains a large amount of variability in the length of fields. This method of structuring is very good solution if the fields are already fixed in length or there is very little variation in field lengths.

```
AMAR        1VACS054    11     12     13
MARY.S      1VA09IS053  15     17     53
```

Method 2: Begin each field with a length indicator.

In this method, it is possible to know the end of the field, as the length of the field is stored at the beginning of each field. This type of fields are called length-based field.

04AMAR**08**1VACS054**02**11**02**12**02**13
06MARY.S**10**1VA09IS053**02**15**02**17**02**53

Method 3: Separate the fields with delimiters:

In this method, the fields are separated by using a special character or sequence of characters. The special character used to separate the fields is called the delimiter.

The selected delimiter to separate the fields should not appear as a field value. A delimiter can be ‘|’, ‘#’, space, newline etc.

```
AMAR|1VACS054|53|32|42
MARY.S|1VA09IS053|54|73|82
```

Method 4: use a “keyword=Value” expression to identify fields.

In this method, the keyword and its values are stored for each record. It is the first structure in which a field provides information about itself. Such self-describing structures are very useful tools for organizing files in many applications.

It is easy to identify the missing fields.

The main disadvantage of this structure is that 50% or more of the files space is occupied by the keywords.

```
Name=AMAR|usn=1VACS054|M1=53|M2=32|M3=42
Name=MARY.S|usn=1VA09IS053|M1=54|M2=73|M3=82
```

Record structure (<u>organization of records in a file):</u>

A record is defined as a set of fields that belong together when the field is viewed in terms of a higher level of organization. A record in a file is represented as a structured data object.

The term object is used to refer to data residing in memory and the term record is used to refer to data residing in a file.

The various methods used to organize the records of a file are:

1. Make records a predictable no of bytes.
2. Make records a predictable no of fields.
3. Begin each record with a length indicator.
4. Use an index to keep track of addresses.
5. Place a delimiter at the end of each record.

<u>Method 1: Make records a predictable no of bytes.</u> (Fixed-length records)

A fixed length record file is one in which each record contains the **same number** of bytes.

The record size can be determined(fixed) by adding the maximum space occupied by each field. This method is also called as **counting bytes structure**.

Here, the size of the entire record is fixed, the fields inside the record can be of varying size or of fixed size.

Fixed length records with the fixed length fields-

```
AMAR########1VACS057####17##### 54####13
MARY.S####### 1VA09IS054###15##### 54####17
```

Fixed length records with variable length fields-

```
AMAR|1VACS057|17|54|13|  ←  unused space          →
MARY.S|1VA09IS054|15|54|17|  ←     unused space    →
```

<u>Method 2: Make records a predictable number of fields.</u>

In this method the number of fields in each record is fixed. This method is also called as fixed field count structure. Assuming that every record has 5 fields, then each record can be displayed by counting the fields modulo five.

```
AMAR|1VACS057|17|54|13| MARY.S|1VA09IS054|15|54|17|…….
```

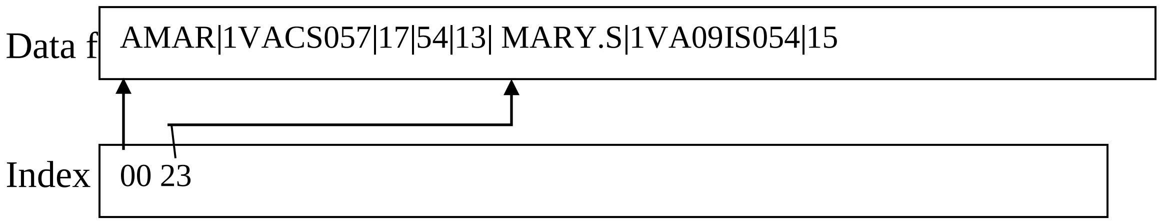
<u>Method 3: Begin each record with a length indicator</u>

In this method, every record would begin with an integer, that indicates how many bytes are there in the rest of the record. This is a commonly used method for handling variable length records.

```
23 AMAR|1VACS057|17|54|13| 27 MARY.S|1VA09IS054|15|….
```

<u>Method 4: Use an index to keep track of addresses:</u>

An index is used to keep a byte offset for each record. The byte offset allows us to find the beginning of each successive record and compute the length of each record. The position of any record can be taken fom the index files then seek to the record in the data file.



<u>Method 5: Place a delimiter at the end of each record.</u>

In this method, a special character other than the field delimiter is placed at the end of each record, which when encountered indicated the end of the record. In the below example, the # character is taken as record delimiter.

```
AMAR|1VACS057|17|54|13|# MARY.S|1VA09IS054|15|#….
```

A record structure that uses a length indicator

A record structure that uses a length indicator at the beginning, has some problems such as-

1. As the length indicator is put at the beginning of every record, the sum of the lengths of the field record must be known before writing the fields to the file.
2. In what form should the record length field be created in the file?

To solve the first problem, all the field values are put to a buffer one-by-one with delimiter separating the fields and finally the length of the buffer is found by using ‘strlen’ function.

```
char buffer[100];
        In a loop,
                buffer[i+1]=field values one-by-one with delimiter.
                The values can be put to buffer using strcpy() and strcat()
//find buffer length.
        int len=strlen(buffer);
//put the record length to file first, but after converting to char.
```

[[To create a buffer’b’ of variables s.usn,s.name,s.sem and s.dept –

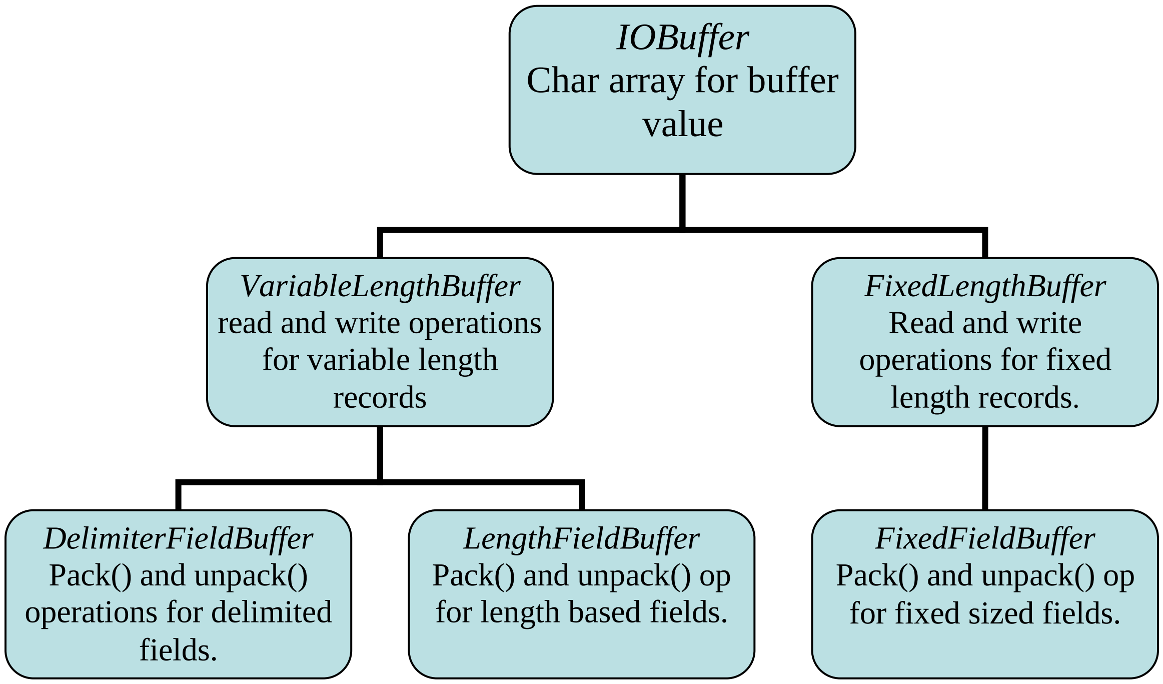
```
strcpy(b,s.usn);
strcat(b,’|’);
strcat(b,s.name);
strcat(b,’|’);
strcat(b,s.sem);
strcat(b,’|’);
strcat(b,s.dept);
strcat(b,’|’);

or use sprintf(b,”%s|%s|%s|%s|”, s.usn,s.name,s.sem,s.dept);   ]]
```

To solve the second problem, the integer value is converted to ASCII characters and then its hexa value is put to the file.

Using Inheritance for record buffer classes

Inheritance allows multiple classes to share members and methods. One or more base classes define members and methods, which are then used by subclasses.


IOBuffer
Char array for buffer value
VariableLengthBuffer
read and write operations for variable length records
FixedLengthBuffer
Read and write operations for fixed length records.
DelimiterFieldBuffer
Pack() and unpack() operations for delimited fields.
LengthFieldBuffer
Pack() and unpack() op for length based fields.
FixedFieldBuffer
Pack() and unpack() op for fixed sized fields.

The members and methods that are common to all the three buffer classes, are included in the base class IO buffer. Other methods are in classes variable length buffer and fixed length buffer, which support the read and write operations for different type of records. Finally the classes LengthFieldBuffer, DelimiterFieldBuffer and FixedFieldBuffer have the pack and unpack methods for the specific field representations.

Record Access:

To view the contents of a specific record with a key, the key must be in canonical form. A standard form of representing a key is called the canonical form.

The unique key, which identifies a single record is the primary key. Ex : USN.

Secondary key is a key that is common to a group of records. Ex : Semester.

Sequential Search:

Sequential search is the technique of searching, where the records are accessed one-by-one and checked till the searching key is found.

The work required to search sequentially for a record in a file with n records is proportional to n.

On average it takes approximately n/2 comparisons.

A sequential search is said to be of the order O(n) because the time taken for this search is proportional to n.

The performance of sequential search can be improved by reading in a block of several records all at once and then processing that blocks of records in memory.

- Although blocking improves the performance, it doesn't change the order of sequential search operation. The order of search is O(n) only.
- Blocking decreases the time of accessing data.
- Blocking doesn't change the number of comparisons made between memory and programs, but sometimes increases the amount of data accessed from secondary memory(as whole block is accessed, even if the first record is the one we wanted.)
- Blocking saves time, as it decreases the amount of seeking.

When is sequential searching good?

Sequential search is too expensive for serious retrieval situations, but have some advantages such as-

- It is extremely easy to program.
- It requires the simplest of file structures.

Sequential search can be used when

- Files have few records only.
- Files in which searching is not done.(ex: tape file)
- ASCII files in which you are searching for some patterns.
- Files in which large number of matches is expected.

UNIX tools for sequential processing:

UNIX is an ASCII file with the new-line character as the record delimiter and the white space as the field delimiter.

A number of tools are present in UNIX which sequentially processes a file, ex:cat, wc, grep.

$cat myfile – displays the contents of the file.

```
AMAR 1VACS057  13      15       17
MARY.S       1VA09IS054         53      17      54
```

➔ wc – reads through an ASCII file sequentially and counts the number of lines, words and characters in a file.

```
$wc myfile
2        10      49
```

➔ grep – A filter, which sequentially searches for a certain word or character string in the specified file.
If the word is found, 'grep' returns the complete line in the file to the standard output.

```
$grep Mary myfile
        Mary.S 1VA09IS053       53      17      54
```

Direct Access:

The alternative to sequential search for a record is the retrieval mechanism known as direct access. There is direct access to a record, when we can seek directly to the beginning of the record and read it.

The problem can be solved if we know the RRN of the required record in fixed-length records. RRN is the Relative Record Number, it gives the position of a record with respect to the beginning of the file. The first record in the file has RRN0, the next has RRN1 and so on.

If the record is of variable length, the records are searched sequentially to get the correct RRN. In case of fixed length records the byte offset from start of the file is-

Byte offset = record size(r) * Required RRN(n)

The byte offset of record of RRN 2 and fixed length record of size 128 bytes is

Byte offset = 128*2 = 256

From 256th byte the record of RRN2 starts.

Header records:

A header record is often placed at the beginning of the file to hold some general information about a file.

The header keeps track of some general information such as no of records in file, size of each record, size of header, date and time of last alteration etc. Header record usually has a different structure than the data records in a file.

Headers are usually defined in a class by the help of 2 methods-

- int readheader(); - reads the header and returns
    - 1 –if header is correct and
    - 0 –if header is wrong.
- int writeheader(); - adds header to the file and returns the number of bytes in the header.

Important Questions

1. What are file structures? Explain briefly the history of file structures design.

2. Explain the different costs of disk access. Define i)seek time ii)rotational d iii)transfer time
3. Explain the functions OPEN,READ and WRITE with parameters.
4. Briefly explain the different basic ways to organize the data on a disk.
5. Briefly explain the organization of data on Nine-Track tapes with a neat diagran

6. With neat sketch, explain Unix directory structure
7. Explain sector based data organization in magnetic disk.
8. Differentiate between constant linear velocity (CLV) and constant angular vel (CAV)
9. Differentiate between physical file and logical file
10. Discuss about the Fundamental File processing operations
11. What are the major strengths and weekness of CD - ROM?
12. What is seeking? How it is supported in 'C' streams and in C++ streams?
13. Explain data organization on disk
14. What is a record? Explain different methods for organizing records of a file. (12

    Or

    Explain different record structures used in the organization of file. (10

Explain the different methods of file organization to maintain the identity of fields. (10)

3. Explain UNIX tools for sequential processing. (5)
4. Design an algorithm for sequential search.
5. What are the different methods of accessing records? Explain
6.Explain the different ways of adding structure to file
7.Explain hierarchy for record buffer objects. (6)
 15.
 16.Problems