Basic Concepts of File System

→ Volatile nature of primary memory leads to use secondary memory.

→ File System controls how data is stored and retrieved. Without a file system, data placed in a storage medium would be one large body of data with no way to tell where one piece of data stops and next begins.

→ By seperating the data into pieces and giving each piece a name, the data is easily indentified & isolated

→ Each group of data is called a "file".

→ The structure & logic rules used to manage the groups of data and their names is called "file system."

⇒ File system is the spn which empowers users and applications to organize and manage their files.

What is file?

→ A program
→ Web Image
→ Executable
→ Music in digital format
→ Irrespective of the content any organize information is a file

⇒ A file is a named collection of related information that is recorded on secondary storage device, such as magnetic disks, magnetic tapes & optical disks.

File Attributes :-

Name – only information kept in human readable form.

(2) Identifier → Identifies the file in the file system.

(3) Type → needed for systems that support different types.

(4) Location → pointer to file location on device.

(5) Size → Current file size

(6) Protection : controls who can reading, writing, executing

(7) Time, date & user identification - data for protection, security & usage monitoring.

→ Information about files are kept in directory structure which is maintained on the disk

File Operations

i) Create → Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in directory.

ii) Writing a file → To write a file, we make a system call specifying both the name of a file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a write pointer to the location in the file where the next write be

is to take place. The write pointer must be updated whenever a write occurs.

(3) Reading a file → To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block should be put. Again, the directory is searched for the associated entry and the system needs a record pointer to the location in the file where the next read is to take place.

(4) Repositioning within a file → The directory is searched for the appropriate entry, and the current file-position pointer is repositioned to a given value. This file operation is also known as a file seek.

v) Deleting a file :-

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

vi) Truncating a file :-

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

Other common operation includes appending new information to the end of an existing file and renaming an existing file.

vii) open() & close() :-

Some systems implicitly open a file when the first reference to its made. The file is automatically closed when the job or program that opened the file terminates. Most systems, however, require that the programmer open a file explicitly with the open() system call before the file can be used.

→ The open() operation takes a file name and searches the

directory copying the directory entry into the open-file table.

→ The open() call typically returns a pointer to the entry in the open-file table.

→ The implementation of open() & close() operations is more complicated in an environment where several processes may open the file simultaneously. This may occur in a system where several different applications open the same file at the same time.

→ Typically OS uses two level of internal table - a per process table and a system-wide table.

Several pieces of data are needed to manage open files —

i) File pointer :- pointer to last read/write location, per process that has the file open.

ii) File-open count — Counter of no. of times a file is open — to allow removal of data from open-file table when last processes closes it.

iii) Disk location of the file → cache of data access information

iv) Access rights → per process access mode information.

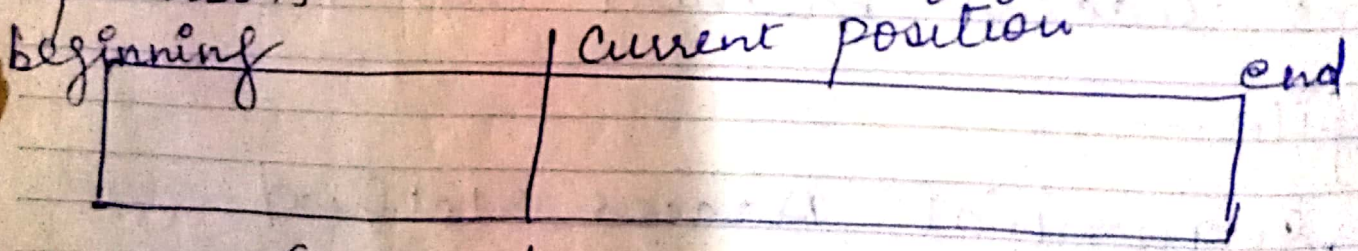
Access Methods of File :-

The file contains the information but when it required to used this information can be access by the access methods & reads into the computer memory — Some Systems provides only one access method and some provide more than one access method to access the file —

i) Sequential Access Method
ii) Direct or random Access Method
iii) Index Access Method
iv) Index sequential Access Method

i) Sequential Access →

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



Sequential access file

Read Operation → read next - reads the next portion of the file and automatically advances a file pointer.

Write Operation - write next - appends to the end of the file and advances to the end of newly written material
→ Such a file can be reset to the beginning and on some systems, a program may be able to skip forward and backward n records for some integer n - perhaps only for n = 1

ii) Direct access Method ↓

Another method is direct access or (relative access). A file is made up of a fixed length logical ~~address~~ records that allow programs to read and write records rapidly in no particular order.

→ The direct access method is based on a disk model of a file, since disks allow random access to any file block. 1

→ For direct access, the file is viewed as a numbered sequence of blocks or records. Thus, we may read block 14, then read block 53 and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

→ For the direct-access method, the file operations must be modified to include the block number as a parameter.

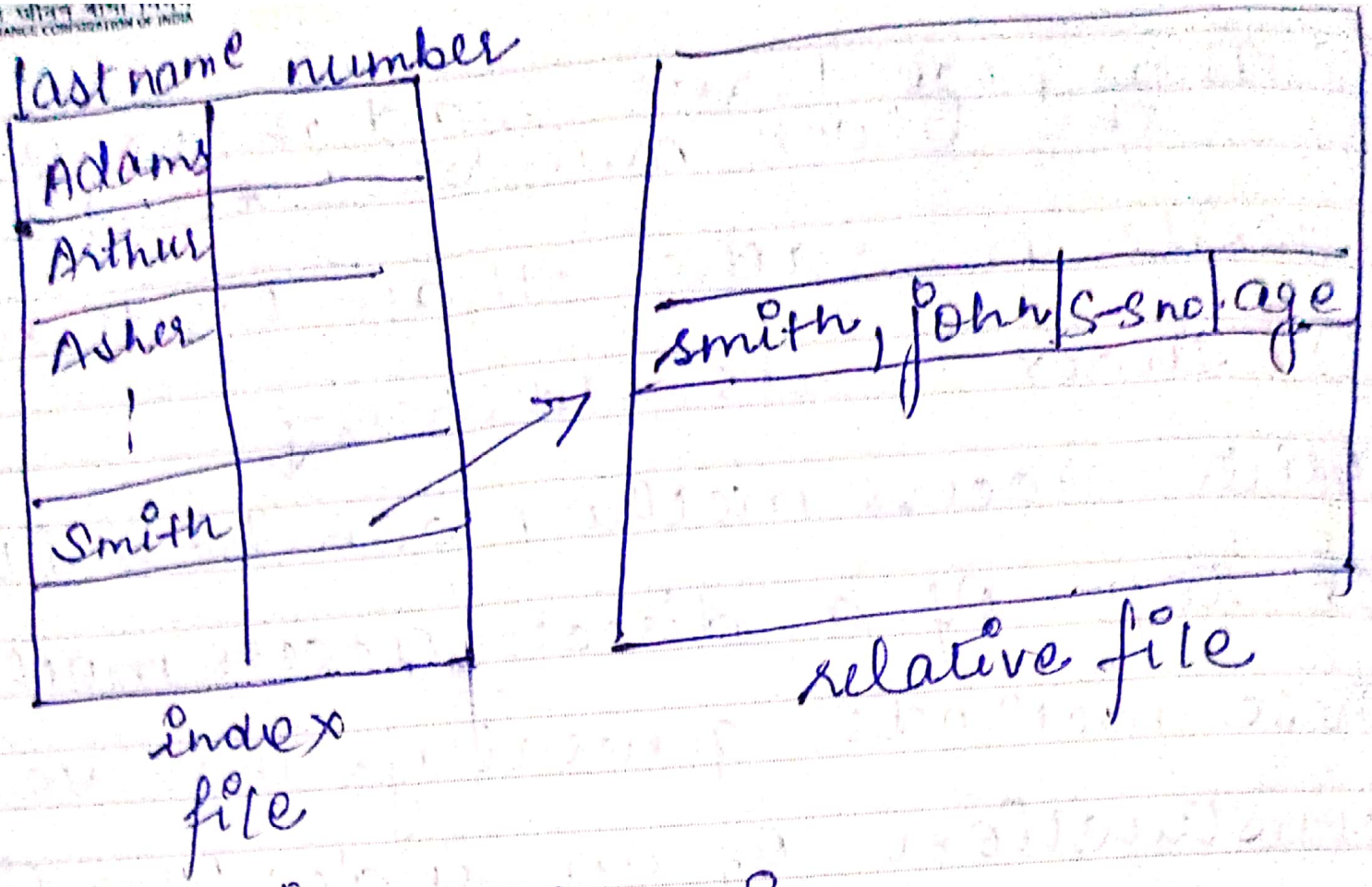
→ Thus, we have read n, where n is the block number. ~~as a parameter~~ & write n rather than write next.

(iii) Index access method,

Other access method can be built on top of a direct-access method. These methods generally involve the construction of an index of a file.

→ The index, like an index in the back of a book, is created for each file which contains pointers to the various blocks.

→ To ~~access the file~~ find a record in a file, we must search the index and then use the pointer to access the file directly and to find the desired record.



Protection & Security

Protection mechanisms control access to a system by limiting the types of file access permitted to users.

→ In addition, protection must ensure that only processes that have gained proper authorization from the OS can operate on memory segments, the CPU and other resources.

→ Security ensures the authentication of system users to protect the integrity of the information stored in the system (both data & code), as well as the physical resources of the computer system.

Protection and Security Design Principles

i) least privilege → Every object should work within a minimal set of privileges access rights should be obtained by explicit request and the default level of access should be "none".

ii) Economy of mechanisms →

Security mechanisms should be small and simple as possible, aiding their verification. They

imply that that they should be integeral to an O.S's design and not an afterthought.

iii) Acceptability →
Security mechanisms should be robust.

iv) Complete → Mechanisms must be pervasive and access control checked during all operations — including the tasks of backup and mainteance.

v) Open design →
An OS security should not remain secret, nor be provided by stealth. Open mechanisms are subject to scrutiny, review and continued refinement.

Security & Protection : Policies & Mechanism

Security Policy Specify whether a person can become a user of the system. This function is performed by the system administrator

Mechanism
i) Add or delete users
ii) Verify whether a person is an authorized user.

Protection Policy Specify whether a user can access a specific file. The owner of a file performs this function while creating it

Mechanisms
i) Set or change protection info for a file
ii) Check whether a file can be accessed by a user

file permission and access modes in Unix. File ownership is an important component of Unix that provides a secure method for storing files. Every file in Unix has the following attributes −

- **Owner permissions** − The owner's permissions determine what actions the owner of the file can perform on the file.
- **Group permissions** − The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.
- **Other (world) permissions** − The permissions for others indicate what action all other users can perform on the file.

The Permission Indicators

While using **ls -l** command, it displays various information related to file permission as follows −

```
$ls -l /home/amrood
-rwxr-xr--  1 amrood   users 1024  Nov 2 00:10  myfile
drwxr-xr--- 1 amrood   users 1024  Nov 2 00:10  mydir
```

Here, the first column represents different access modes, i.e., the permission associated with a file or a directory.

The permissions are broken into groups of threes, and each position in the group denotes a specific permission, in this order: read (r), write (w), execute (x) −

- The first three characters (2-4) represent the permissions for the file's owner. For example, **-rwxr-xr--** represents that the owner has read (r), write (w) and execute (x) permission.
- The second group of three characters (5-7) consists of the permissions for the group to which the file belongs. For example, **-rwxr-xr--** represents that the group has read (r) and execute (x) permission, but no write permission.
- The last group of three characters (8-10) represents the permissions for everyone else. For example, **-rwxr-xr--** represents that there is **read (r)** only permission.

File Access Modes

The permissions of a file are the first line of defense in the security of a Unix system. The basic building blocks of Unix permissions are the **read**, **write**, and **execute** permissions, which have been described below −

Read

Grants the capability to read, i.e., view the contents of the file.

Write

Grants the capability to modify, or remove the content of the file.

Execute

User with execute permissions can run a file as a program.

Directory Access Modes

Directory access modes are listed and organized in the same manner as any other file. There are a few differences that need to be mentioned −

Read

Access to a directory means that the user can read the contents. The user can look at the **filenames** inside the directory.

Write

Access means that the user can add or delete files from the directory.

Execute

Executing a directory doesn't really make sense, so think of this as a traverse permission.

A user must have **execute** access to the **bin** directory in order to execute the **ls** or the **cd** command.

Changing Permissions

To change the file or the directory permissions, you use the **chmod** (change mode) command. There are two ways to use chmod — the symbolic mode and the absolute mode.

Using chmod in Symbolic Mode

The easiest way for a beginner to modify file or directory permissions is to use the symbolic mode. With symbolic permissions you can add, delete, or specify the permission set you want by using the operators in the following table.

Sr.No.	Chmod operator & Description
1	**+** Adds the designated permission(s) to a file or directory.
2	**-** Removes the designated permission(s) from a file or directory.
3	**=** Sets the designated permission(s).

Here's an example using **testfile**. Running **ls -1** on the testfile shows that the file's permissions are as follows −

```
$ls -l testfile
-rwxrwxr--  1 amrood   users 1024  Nov 2 00:10  testfile
```

Then each example **chmod** command from the preceding table is run on the testfile, followed by **ls –l**, so you can see the permission changes −

```
$chmod o+wx testfile
```

```
$ls -l testfile
-rwxrwxrwx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod u-x testfile
$ls -l testfile
-rw-rwxrwx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod g = rx testfile
$ls -l testfile
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Here's how you can combine these commands on a single line −

```
$chmod o+wx,u-x,g = rx testfile
$ls -l testfile
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Using chmod with Absolute Permissions

The second way to modify permissions with the chmod command is to use a number to specify each set of permissions for the file.

Each permission is assigned a value, as the following table shows, and the total of each set of permissions provides a number for that set.

Number	Octal Permission Representation	Ref
0	No permission	---
1	Execute permission	--x
2	Write permission	-w-
3	Execute and write permission: 1 (execute) + 2 (write) = 3	-wx
4	Read permission	r--
5	Read and execute permission: 4 (read) + 1 (execute) = 5	r-x
6	Read and write permission: 4 (read) + 2 (write) = 6	rw-
7	All permissions: 4 (read) + 2 (write) + 1 (execute) = 7	rwx

Here's an example using the testfile. Running **ls -1** on the testfile shows that the file's permissions are as follows −

```
$ls -l testfile
-rwxrwxr--  1 amrood   users 1024  Nov 2 00:10  testfile
```

Then each example **chmod** command from the preceding table is run on the testfile, followed by **ls –l**, so you can see the permission changes −

```
$ chmod 755 testfile
$ls -l testfile
-rwxr-xr-x  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod 743 testfile
$ls -l testfile
-rwxr---wx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod 043 testfile
$ls -l testfile
----r---wx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Changing Owners and Groups

While creating an account on Unix, it assigns a **owner ID** and a **group ID** to each user. All the permissions mentioned above are also assigned based on the Owner and the Groups.

Two commands are available to change the owner and the group of files −

- **chown** − The **chown** command stands for **"change owner"** and is used to change the owner of a file.
- **chgrp** − The **chgrp** command stands for **"change group"** and is used to change the group of a file.

Changing Ownership

The **chown** command changes the ownership of a file. The basic syntax is as follows −

```
$ chown user filelist
```

The value of the user can be either the **name of a user** on the system or the **user id (uid)** of a user on the system.

The following example will help you understand the concept −

```
$ chown amrood testfile
$
```

Changes the owner of the given file to the user **amrood**.

NOTE − The super user, root, has the unrestricted capability to change the ownership of any file but normal users can change the ownership of only those files that they own.

Changing Group Ownership

The **chgrp** command changes the group ownership of a file. The basic syntax is as follows −

```
$ chgrp group filelist
```

The value of group can be the **name of a group** on the system or **the group ID (GID)** of a group on the system.

Following example helps you understand the concept −

```
$ chgrp special testfile
$
```

Changes the group of the given file to **special** group