mood-book

UNIT - 5

File- System Interface

1) File Concept: A file is a named Collection of related information that is recorded on secondary Storage. From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file.

Commonly, files represent programs(both source and Object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free from, such as text files, or may be formatted rigidly.

A file is a sequence of bits, bytes, lines or records. Many different types of information may be stored in a file - source or executable programs, numeric or text data, photos, music, video and so on. A file has a certain defined structure, which depends on its type.

A text file is a sequence of characters organized into lines (and possibly pages). A source file is a sequence of functions, each of which is further organized as declarations followed by executable statements. An executable file is a series of code sections that the loader can bring into memory and execute.

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

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

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

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

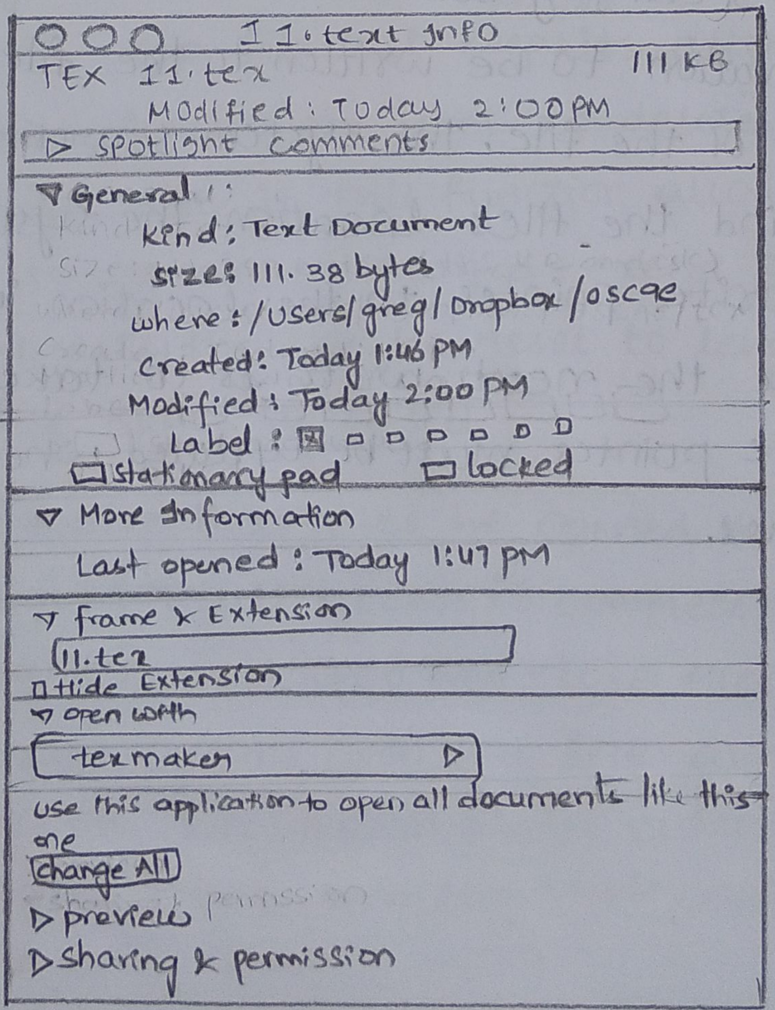
Location: This information is a pointer to a device and to the location of the file on that device.

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

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

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

The following fig shows A file info window on Mac OS X :



○○○ 11.text info

TEX 11.tex 111 KB

Modified: Today 2:00 PM

▷ Spotlight comments

▽ General :

Kind: Text Document

size: 111.38 bytes

where: /users/greg/Dropbox/osc9e

Created: Today 1:46 PM

Modified: Today 2:00 PM

Label: ☒ □ □ □ □ □ □

□ stationary pad □ locked

▽ More Information

Last opened: Today 1:47 PM

▽ frame & Extension

11.tex

□ Hide Extension

▽ open with

texmaker ▷

use this application to open all documents like this one

Change All

▷ preview

▷ Sharing & permission

1.2 File Operations: A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files.

→ Creating a file: Two steps are necessary to create a file. First, space in the file system must be found for the file.

→ writing a file: To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a write pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.

→ Repositioning a file: The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as file seek.

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

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

→ Most OS require that files be opened before access and closed after all access is complete. Normally the programmer must open and close files explicitly, but some rare systems open the file automatically at first access. Information about currently open files is stored in an open file table, containing for example:

* File pointer. On systems that do not include a file offset as part of the read() and write() system calls, the system must track the last read-write location as a current-file-position pointer. This pointer is unique to each process operating on the file and therefore must be kept separate from the on-disk file attributes.

* File-open count. As files are closed, the operating system must reuse its open-file table entries,. Or it could run out of space in the table. Multiple processes may have opened a file, and the system must wait for the last file to close before removing the open-file table entry. The file-open count tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry.

* Disk location of the file: Most file operations require the system to modify data within the file. The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.

* Access rights. Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/O requests.

→ Some systems provide support for file locking.

* A shared lock is for reading only.
* A exclusive lock is for writing as well as reading.
* A mandatory lock is enforced. If a lock is mandatory, then once a process acquires an exclusive lock, the operating system will prevent any other process from accessing the locked file.

Ex: assume a process acquires an exclusive lock on the file system. log. If we attempt to open system. log from another process - for example, a text editor - the operating system will prevent access until the exclusive lock is released.

* An advisory lock is informational only, and not enforced. if the lock is advisory, then the operating system will not prevent the text editor from acquiring access to system. log. Rather, the text editor must be written so that it manually acquires the lock before accessing the file. In other words, if the locking scheme is mandatory, the operating system ensures locking integrity.

* UNIX used advisory locks, and windows uses mandatory locks.

1.3 File Types: A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts - a name and an extension, usually seperated by a period as shown in fig below: In this way, the user and the operating system can tell from the name alone what the type of a file is. Most operating systems allow users to specify a file name as a sequence of characters followed by a period and terminated by an extension made up of additional characters. Examples include resume.docx, server.c, and ReaderThread.cpp.

file type	usual extension	function
executable	exe, com, bin, or none	ready-to-run machine - language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats

file type	usual extension	function
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format. for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

→ Macintosh stores a creator attribute for each file, according to the program that first created it with the create() system call.

→ UNIX stores magic numbers at the beginning of certain files. (Experiment with the "file" command, especially in directories such as /bin and /dev)

1.4 File Structure:

* Some files contain an internal structure, which may or may not be known to the Os.

* For the Os to support particular file formats increases the size and complexity of the Os.

* UNIX treats all files as sequence of bytes, with no further consideration of the internal structure. (with the exception of executable binary programs, which it must know how to load and find the first executable statement, etc.)

* Macintosh files have two forks - a resource fork, and a data fork. The resource fork contains information relating to the UI, such as icons and button images, and can be modified inpendently of the data fork, which contains the code or data as appropriate.

1.5 Internal File structure:

* Disk files are accessed in units of physical blocks, typically 512 bytes or some power-of-two multiple thereof. (Larger physical disks use larger block sizes, to keep the range of block numbers within the range of a 32-bit integer).

* Internally files are organized in units of logical units, which may be as small as a single byte, or may be a larger size corresponding to some data record or structure size.

* The number of logical units which fit into one physical block determines its packing, and has an impact on the amount of internal fragmentation (wasted space) that occurs.

* As a general rule, half a physical block is wasted for each file, and the larger the block sizes the more space is lost to internal fragmentation.

2) Access Methods

2.1 Sequential Access:

* The simplest access method is sequential access.

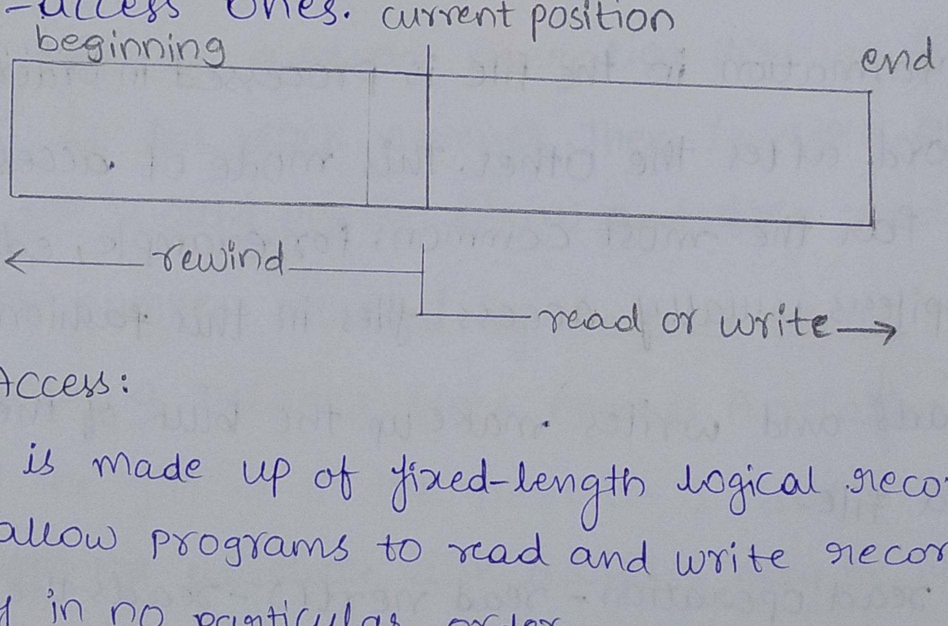
* Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.

* Reads and writes make up the bulk of the operations on a file.

* A read operation - read next() - reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location.

* Similarly, the write operation - write next() - appends to the end of the file and advances to the end of the newly written material (the new end of file). such a file can be reset o the beginning, and on some systems, a program may be able to skip forward or backward n records for some integer n-perhaps only for n=1.

* Sequential access, which is depicted in Figure below, is based on a tape model of a file and works as well on sequential-access devices as it does on random-access ones.



Direct Access:

- A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order.
- The direct-access method is based on a disk model of a file, since disks allow random access to any file block

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

- For direct access method, the file operations must be modified to incude the block number as a parameter. Thus, we have read(n), where n is the block number, rather than read next(), and write(n) rather than write next(). An alternative approach is to retain read next() and write next(), as with sequential access, and to add an operation position file(n) where n is the block number. Then, to effect a read(n), we would position file(n) and ~~then~~ read next().
- The block number provided by the user to the OS is normally a relative block number. A relative block of the file is 0, the next is 1, and so on, even though the absolute disk address may be 14703 for the first block and 3192 for the second. The use of relative block numbers allows the operating system to decide where the file should be placed and helps to prevent the user from accessing portions of the file system that may not be part of her file. Some systems start their relative block numbers at 0; Others start at 1.

The following fig shows simulation of sequential access on a direct-access file.

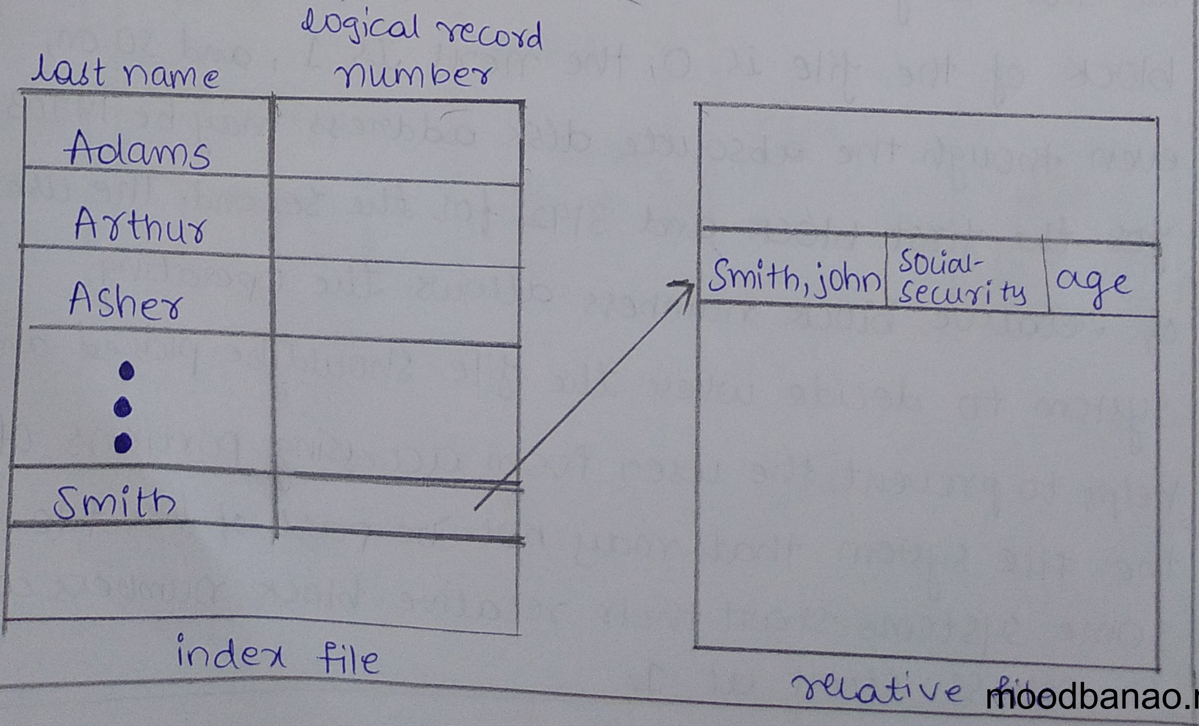
Sequential access	implementation for direct access
reset	cp = 0;

read_next	read cp;
	cp = cp + 1;

write_next	write cp;
	cp = cp + 1;

2.3 Other Access Methods:

- An indexed access scheme can be easily built on top of a direct access system. very large files may require a multi-tiered indexing scheme i.e. indexes of indexes.



3) Directory and disk structure

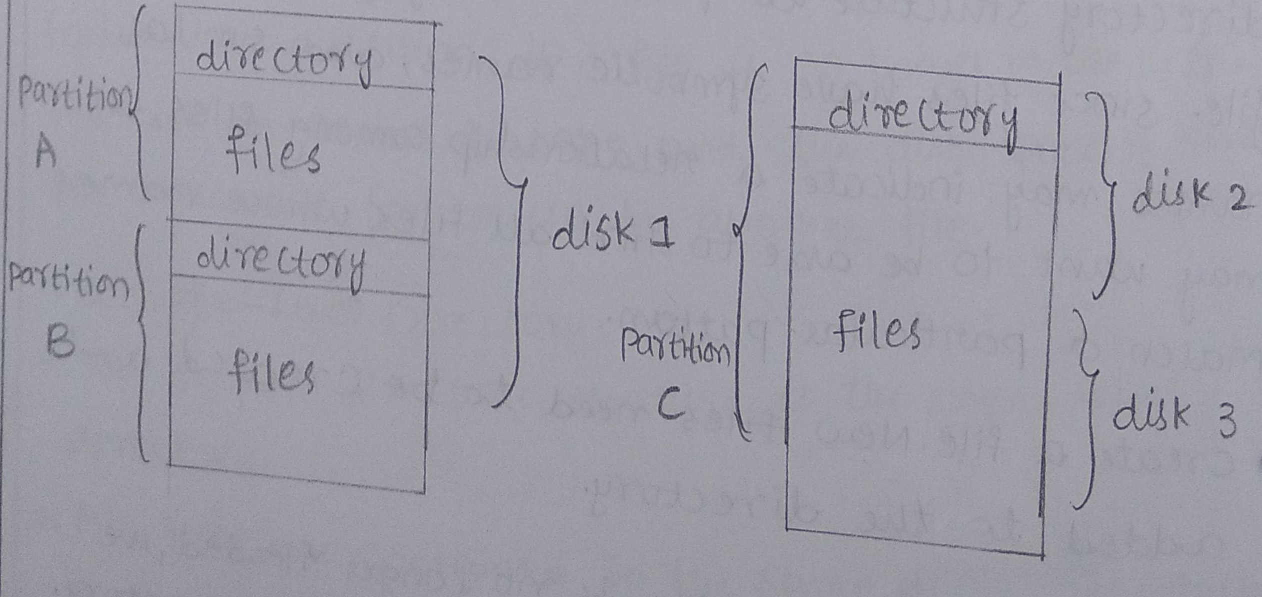
3.1 Storage structure

* A disk can be used in its entirety for a file System

* Alternatively a physical disk can be broken up into multiple partitions, slices, or mini-disks, each of which become a virtual disk and can have its own file system. (or be used for raw storage, swap space etc)

* Or, multiple physical disks can be combined into one Volume, i.e. a larger virtual disk, with its own file system spanning the physical disks.

The following fig shows A typical file-system organization.



3.2 Directory Overview: The directory can be viewed as a symbol table that translates file names into their directory entries. If we take such a view, we see that the directory itself can be organized in many ways. The organization must allow us to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory. In this section, we examine several schemes for defining the logical structure of the directory system. when considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

- Search for a file. we need to be able to search a directory structure to find the entry for a particular file. since files have symbolic names, and similar names may indicate a relationship among files, we may want to be able to find all files whose names match a particular pattern.
- Create a file. New files need to be created and added to the directory.
- Delete a file. when a file is no longer needed, we want to be able to remove it from the directory.

- List a directory. we need to be able to list the files in a directory and the contents of the directory entry for each file in the list.

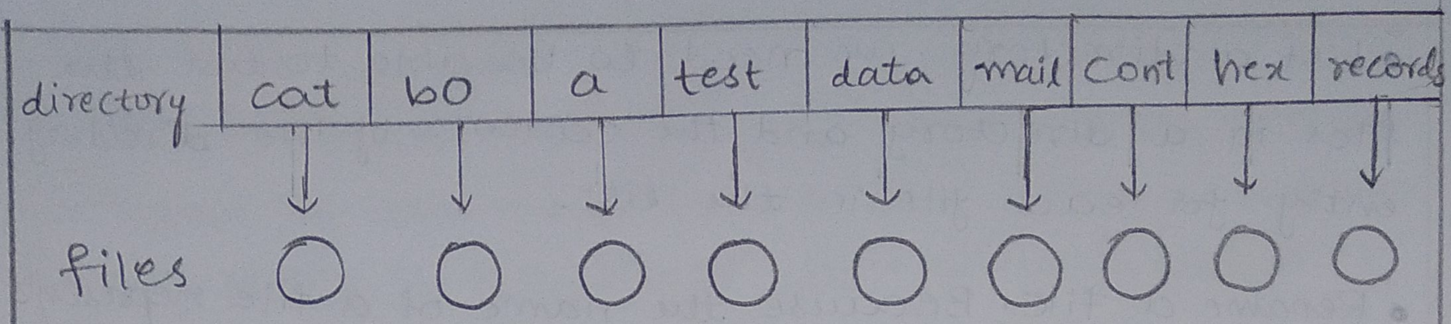
- Rename a file. Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.

- Traverse the file system. we may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, we do this by copying all files to magnetic type. This technique provides a backup copy in case of system failure. In addition, if a file is no longer in use, the file can be copied to tape and the disk space of that file released for reuse by another file.

3.3 Single-Level Directory:

* The simplest directory structure is the single-level directory.

* All files are contained in the same directory, which is easy to support and understand as shown in fig below:



* A single-level directory has significant limitations, however, when the number Of files increases or when the system has more than One user. since all files are in the same directory, they must have unique names.

*If two users call their data file test.txt, then the unique-name rule is violated.

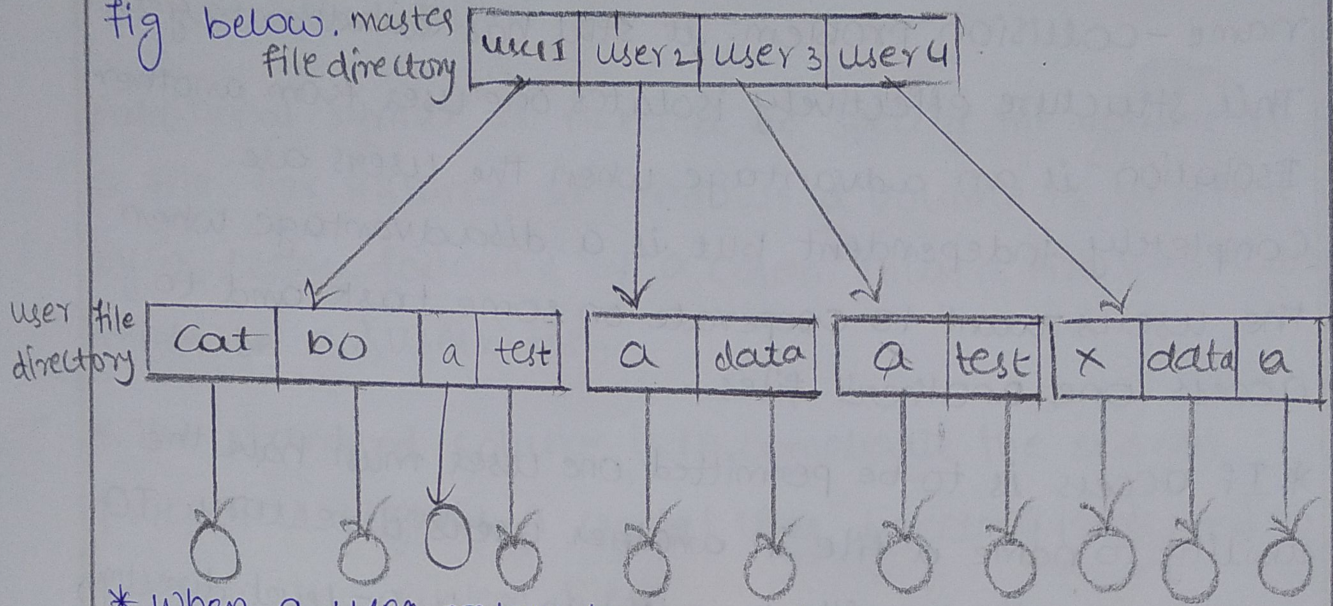
* Even a single user on a single level directory may find it difficult to remember the names Of all the files as the number Of files increases.

3.4 Two-Level Directory:

* A single-level directory often leads to confusion of file names among different users. The standard solution is to create a separate directory for each user.

* In the two level directory structure, each user has his own user file directory (UFD). The UFD's have similar structures, but each lists only the files of a single user.

* When a user job starts or a user logs in, the system's master file directory (MFD) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that users as shown in fig below.



* when a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique.

* To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists.

* To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

* The user directories themselves must be created and deleted as necessary. A special system program is run with the appropriate user name and account information.

* The program creates a new UFD and adds an entry for it to the MFD. The execution of this program might be restricted to system administrators.

*Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. Isolation is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another's files.

*If access is to be permitted, one user must have the ability to name a file in another user's directory. TO name a particular file uniquely in a two-level directory, we must give both the user name and the file-name.

* A two-level directory can be thought of as a tree, or an inverted tree, of height 2. The root of the tree is the MFD. Its direct descendants are the UFD's.

* The descendants of the UFD's are the files themselves. The files are the leaves of the tree. Specifying a user name and a file name defines a path in the tree from the root (the MFD) to a leaf (the specified file). Thus, a user name and a file name define a path name.

*Every file in the system has a path name. To name a file uniquely, a user must know the path name of the file desired.

Ex: if user A wishes to access her own test file named test.txt, she can simply refer to test.txt. To access the file named test.txt of user B (with directory-entry name userb), however, she might have to refer to userb/test.txt.

Additional syntax is needed to specify the volume of a file. For instance, in windows a volume is specified by a letter followed by a colon. Thus, a file specification might be c:\userb\test.

* The standard solution is to complicate the search procedure slightly. A special user directory is defined to contain the system files.

* Whenever a file name is given to be loaded, the OS first searches the local UFD. If the file is found, it is used. If it is not found, the system automatically searches the special user directory that contains the system files. The sequence of directories searched when a file is named is called the search path. The search path can be extended to ~~certain~~ contain an unlimited list of directories to search when a command name is given.

3.5 Tree-structured Directories:

* Two - level tree, natural generalization is to extend the directory structure to a tree of arbitary height (shown in fig below)

* This generalization allows users to create their own sub directories and to organize their files accordingly. A tree is the most common directory structure.

* The tree has a root directory, and every file in the system has a unique path name.

* A directory (or subdirectory) contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and delete directories.

* In normal use, each process has a current directory. The current directory should contain most of the files that are of current interest to the process. When reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file.

* The initial current directory of a user's login shell is designated when the user job starts or the users logs in. The operating system searches the accounting file to find an entry for this user.

* If the accounting file is a pointer to (or the name of) the user's initial directory. This pointer is copied to a local variable for this user that specifies the user's initial current directory.

From that shell, other processes can be spawned.

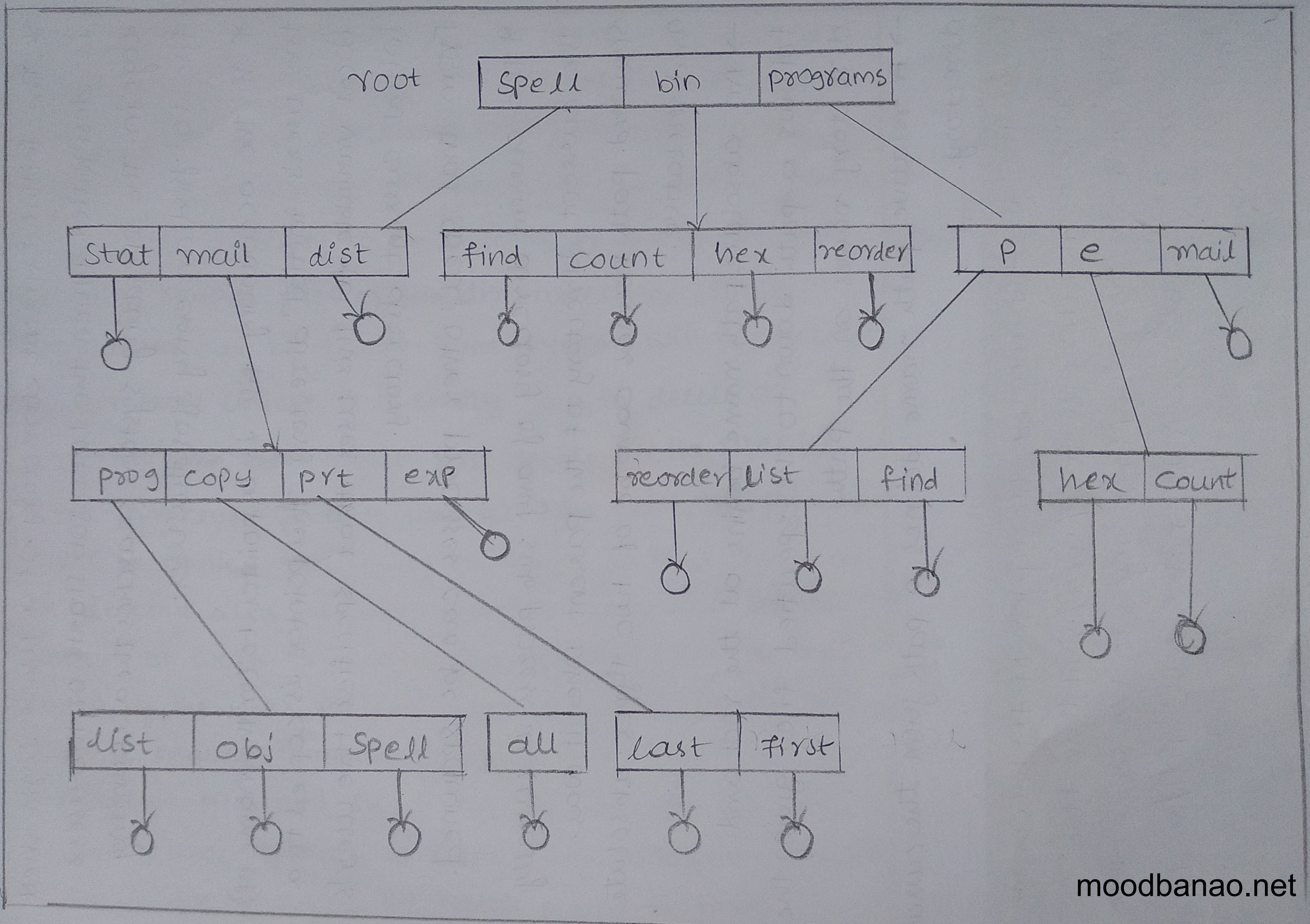
* The current directory of any sub process is usually the current directory of the parent when it was. spawned. path names can be of two types: absolute and relative.

→ An absolute path name begins at the root and follows a path down to the specified file, giving the directory names on the path.

→ A relative path name defines a path from the current directory.

Ex: in the tree-structured file system of Fig below, it the current directory is root/spell/mail, then the relative path name prt/first refers to the same file as does the absolute path name root/spell/mail/prt/first.

25


root
spell
bin
programs
stat
mail
dist
find
count
hex
reorder
p
e
mail
prog
copy
prt
exp
reorder
list
find
hex
count
list
obj
spell
all
last
first

An interesting policy decision in a tree-structured directory concerns how to handle the deletion of a directory. If a directory is empty, its entry in the directory that contains it can simply be deleted. However, suppose the directory to be deleted is not empty but contains several files or sub directories. One of two approaches can be taken. Some systems will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory. If any subdirectories exist, this procedure must be applied recursively to them, so that they can be deleted also. This approach can result in a substantial amount of work. An alternative approach, such as that taken by the UNIX rm command, is to provide an option: when a request is made to delete a directory, all that directory's files and subdirectories are also to be deleted. Either approach is fairly easy to implement; the choice is one of policy. The latter policy is more convenient, but it is also more dangerous, because an entire directory structure can be removed with one command. If that command is issued in error, a large number of files and directories will need to be restored.

With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users. For example, user B can access a file of user A by specifying its path names. User B can specify either an absolute or a relative path name. Alternatively, user B can change her current directory to be user A's directory and access the file by its file names.

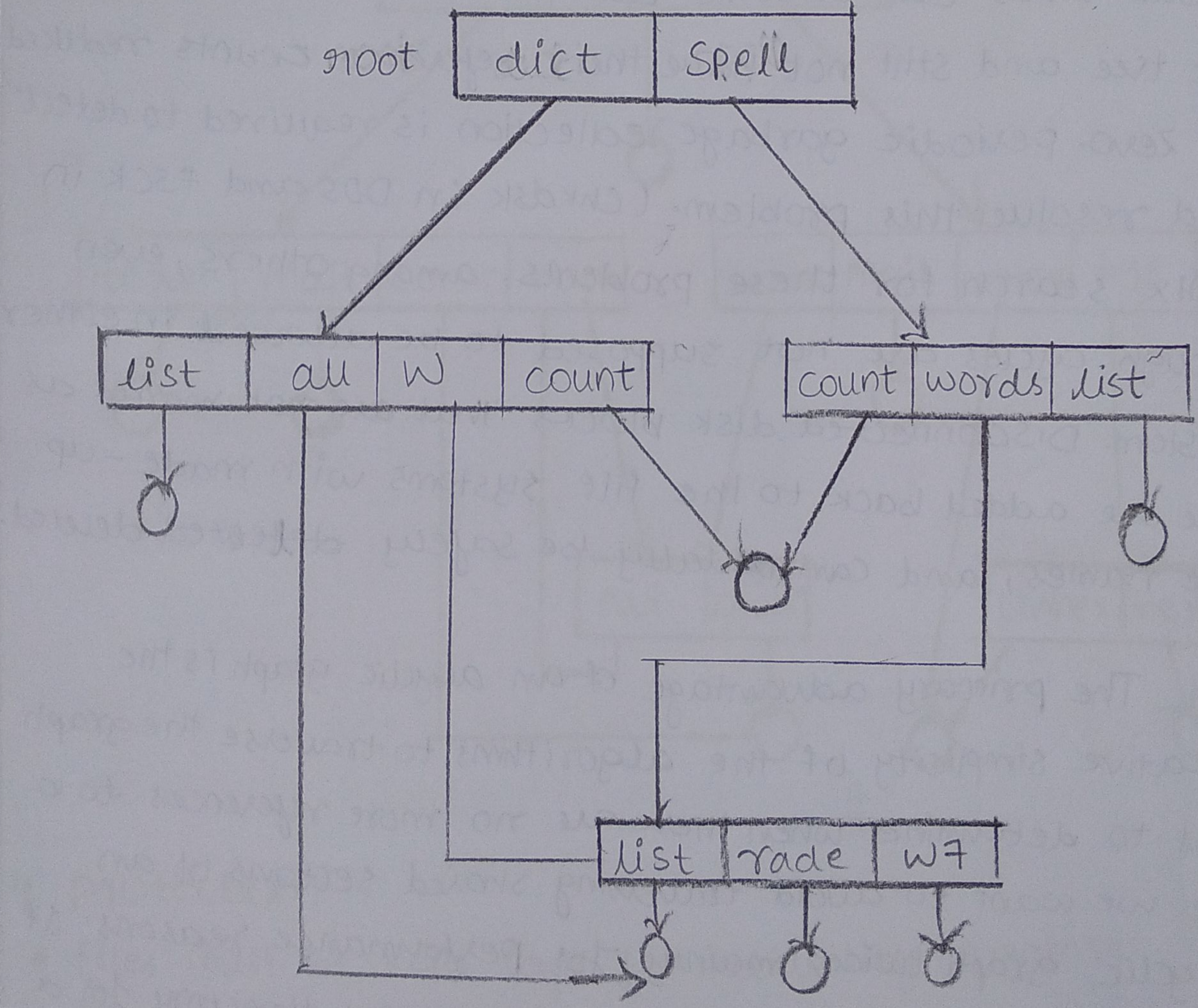
3.6 Acyclic - Graph Directories:

- When the same files need to be accessed in more than one place in the directory structure (e.g. because they are being shared by more than one user/process), it can be useful to provide an acyclic-graph structure. (Note the directed arcs from parent to child).
- UNIX provides two types of links for implementing the acyclic-graph structure. (see "man ln" for more details).

→ A hard link (usually just called a link) involves multiple directory entries that both refer to the same file. Hard links are only valid for ordinary files in the same file system.

→ A symbolic link, that involves a special file, containing information about where to find the linked file. Symbolic links may be used to link directories and/or files in other file systems, as well as ordinary files in the current file system.

- Windows only supports symbolic links, termed shortcuts
- Hard links require a reference count, or link count for each file, keeping track of how many directory entries are currently referring to this file. whenever one of the references. is removed the link count is reduced, and when it reaches zero, the disk space can be reclaimed.



3.7 General Graph Directory:

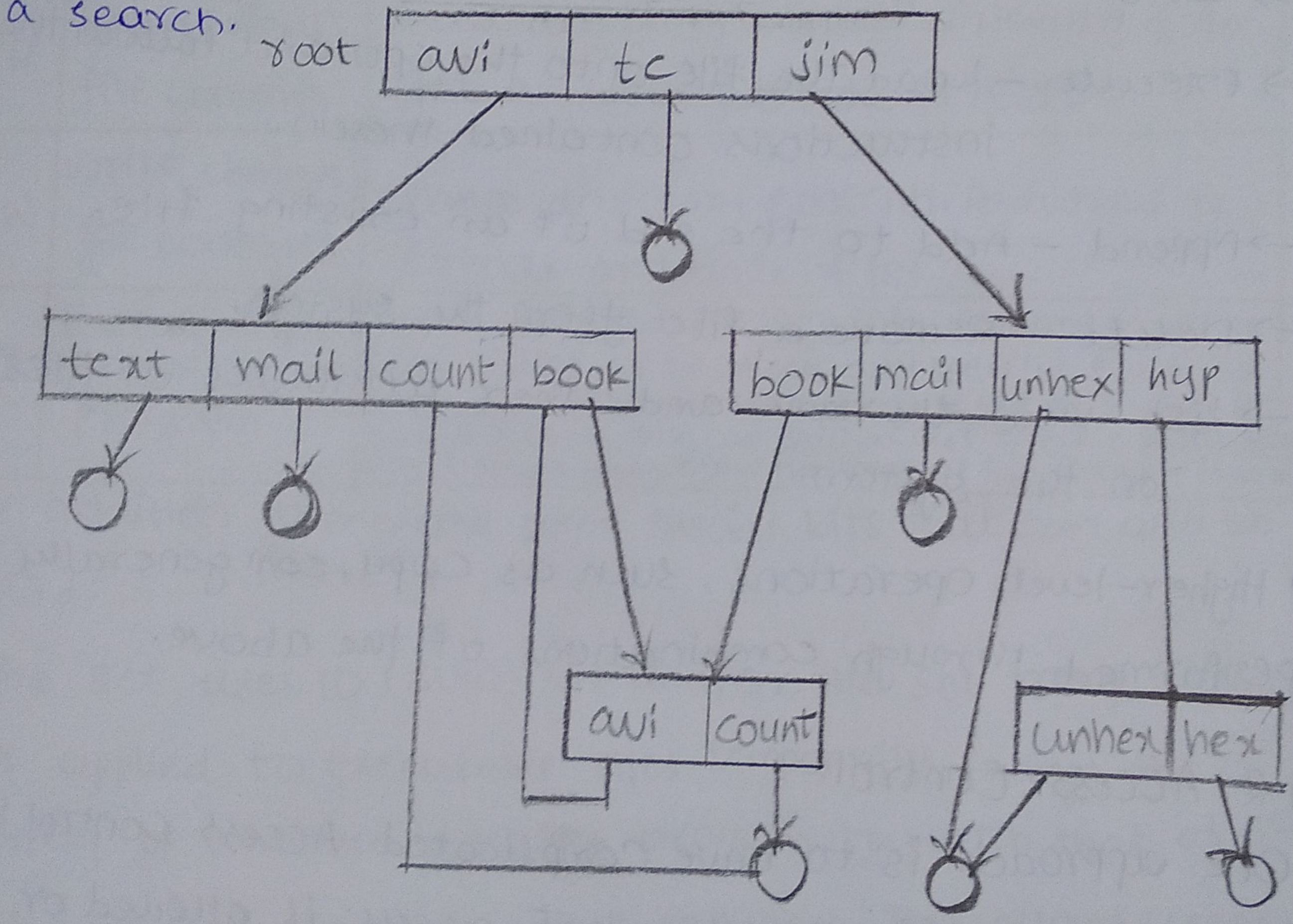
- If cycles are allowed in the graphs, then several problems can arise:

→ Search algorithms can go into infinite loops. One solution is ~~no~~to not follow links in search algorithms. (or not to follow symbolic links, and to only allow symbolic links to refer to directories).

→ Sub trees can become disconnected from the rest of the tree and still not have their reference counts reduced to zero. periodic garbage collection is required to detect and resolve this problem. (chkdsk in DOS and fsck in UNIX search for these problems, among others, even though cycles are not supposed to be allowed in either System. Disconnected disk blocks that are not marked as free are added back to the file systems with made-up file names, and can usually be safely ~~deleted~~ .deleted.

The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse the graph and to determine when there are no more references to a file. we want to avoid traversing shared sections of an acyclic graph twice, mainly for performance reasons. If we have just searched a major shared subdirectory for a particular file without finding it, we want to avoid searching that subdirectory again; the second search would be a waste of time.

If cycles are allowed to exists in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance. A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never teeminating. One solution is to limit arbitrarilly the number of directories that will be accessed during a search.



4 protection:

- Files must be kept safe for reliability (against accidental damage), and protection (against deliberate malicious access). The former is usually managed with backup copies. This section discusses the latter.

- One simple protection scheme is to remove all access to a file. However this makes the file unusable, so some sort of controlled access must be arranged.

4.1 Types of Access:

- The following low-level operations are often controlled: → Read - View the contents of the file

→ Write - Change the contents of file

→ Execute - Load the file onto the CPU and follow the instructions contained therein.

→ Append - Add to the end of an existing file.

→ Delete - Remove a file from the system.

→ List - View the name and other attributes of files on the system.

- Higher-level operations, such as copy, can generally be performed through combinations of the above.

4.2 Access Control:

One approach is to have complicated Access Control Lists, ACL, which specify exactly what access is allowed or denied for specific users or groups.

The AFS uses this system for distributed access. Control is very finely adjustable, but may be complicated, particularly when the specific users involved are unknown (AFS allows some wild cards, so for example all users on a certain remote system may be trusted, or a given username may be trusted when accessing from any remote system).

UNIX uses a set of 9 access control bits, in three groups of three. These correspond to R, W, and X permissions for each of the owner, Group and Others. (See "man chmod" for full details). The RWX bits control the following privileges for ordinary files and directories:

bit	Files	Directories
R	Read (view) file contents.	Read directory contents. Required to get a listing of the directory.
W	write (changes) file contents.	Change directory contents. Required to Create or delete files.
X	Execute file contents as a program.	Access detailed directory info. Required to get a long listing, or to access any specific file in the directory. Note that if a user has x but no R permissions on a directory, they can still access specific files, but only if they already know the name of file they try to access.

In addition there are some special bits that can also be applied:

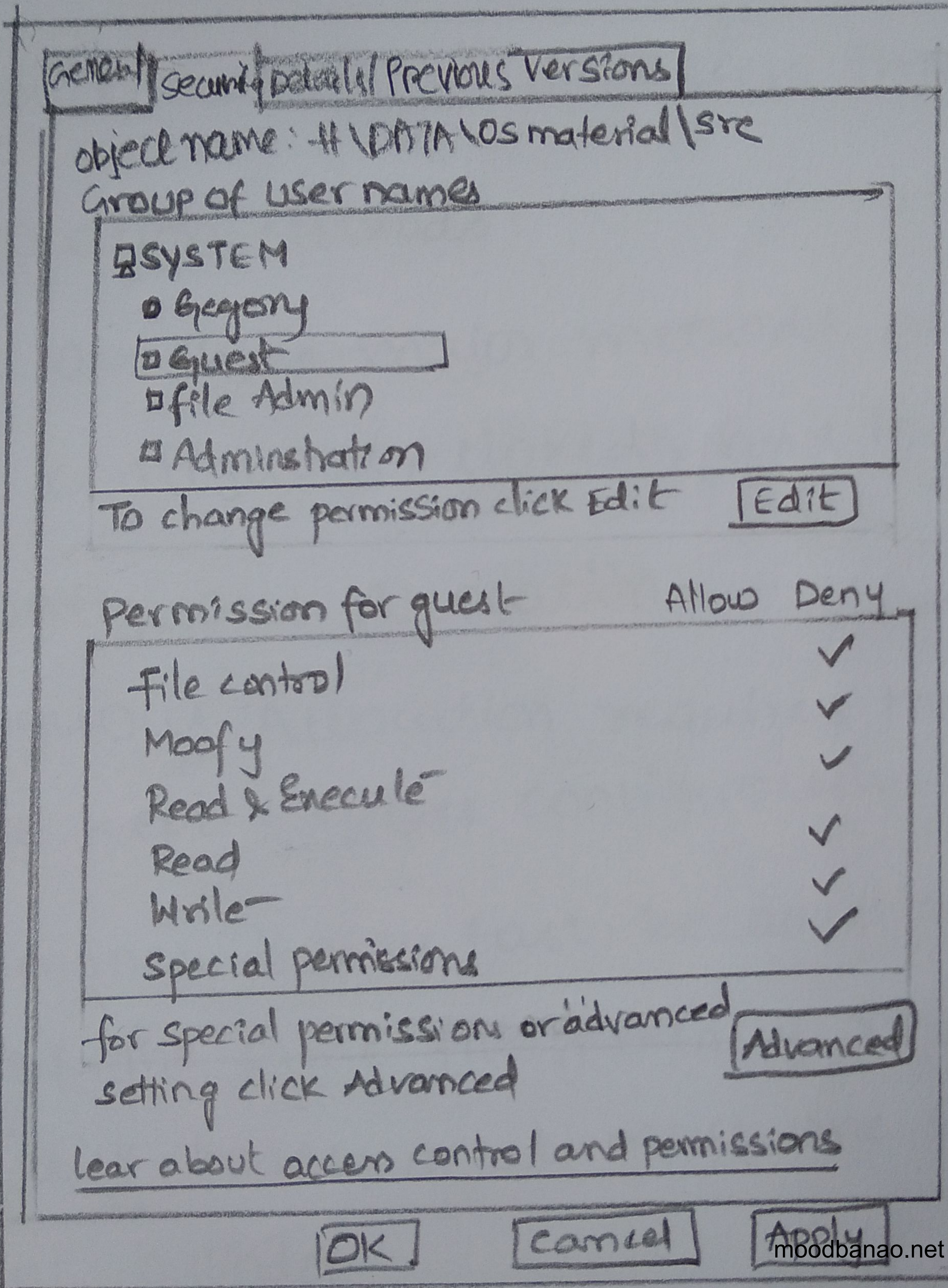
→ The set user ID (SUID) bit and/or the set group ID (SGID) bits applied to executable files temporarily change the identity of whoever runs the program to match that of the owner/group of the executable program. This allows users running specific programs to have access to files (while running that program) to which they would normally be unable to access. Setting of these two bits is usually restricted to root, and must be done with caution, as it introduces a potential security leak.

→ The sticky bit on a directory modifies write permission, allowing users to only delete files for which they are the owner. This allows everyone to create files in /tmp, for example, but to only delete files which they have created, and not anyone else's.

→ The SUID, SGID, and sticky bits are indicated with an S, S and T in the positions for execute permission for the user, group, and others, respectively. If the letter is lower case (s, s, t) then the corresponding execute permission is not also given. If it is uppercase (S, S, T) then the corresponding execute permission is given.

→ The numeric form of chmod is needed to set these advanced bits.

```
-rw-rw-r--    1 pbg  staff   31200  Sep 3 08:30    intro.ps
drwx------    5 pbg  staff   512    Jul 8 09:33    private/
drwxrwxr-x    2 pbg  staff   512    Jul 8 09:35    doc/
drwxrwx---    2 jwg  student 512    Aug 3 14:13    student-proj/
-rw-r--r--    1 pbg  staff   9423   Feb 24 2012    program.c
-rwxr-xr-x    1 pbg  staff   20471  Feb 24 2012    program
drwx--x--x    4 tag  faculty 512    Jul 31 10:31   lib/
drwx------    3 pbg  staff   1024   Aug 29 06:52   mail/
drwxrwxrwx.   3 pbg  Staff   512    Jul 8 09:35    test/
```



object name : H\DATA\OS material\src

Group of User names

☐ SYSTEM
● Gegory
☐ Guest
☐ file Admin
☐ Adminstration

To change permission click Edit [Edit]

Permission for guest	Allow	Deny
File control		✓
Moofy		✓
Read & Enecute		✓
Read		✓
Write		✓
Special permissions		✓

for special permissions or advanced setting click Advanced [Advanced]

lear about accen control and permissions

Sample peemissions in a UNIX system.

- windows adiusts files access through a simple GUI:

5 Allocation Methods:

- There are three major methods of storing files on disks: contiguous, linked, and indexed.

5.1 Contiguous Allocation:

- Contiguous Allocation requires that all blocks of a file be kept together contiguously.

- performance is very fast, because reading successive blocks of the same file generally requires no movement of the disk heads, or at most one small step to the next adjacent cylinder.

- Storage allocation involves the same issues discussed earlier for the allocation of contiguous blocks of memory (first fit, best fit, fragmentation problems, etc). The distinction is that the high time penalty required for moving the disk heads from spot to spot may now justify the benefits of keeping files continguously when possible.

- (Even file systems that do not by default store files contiguosly can benefit from certain utilities that compact the disk and make all files contiguous in the process).

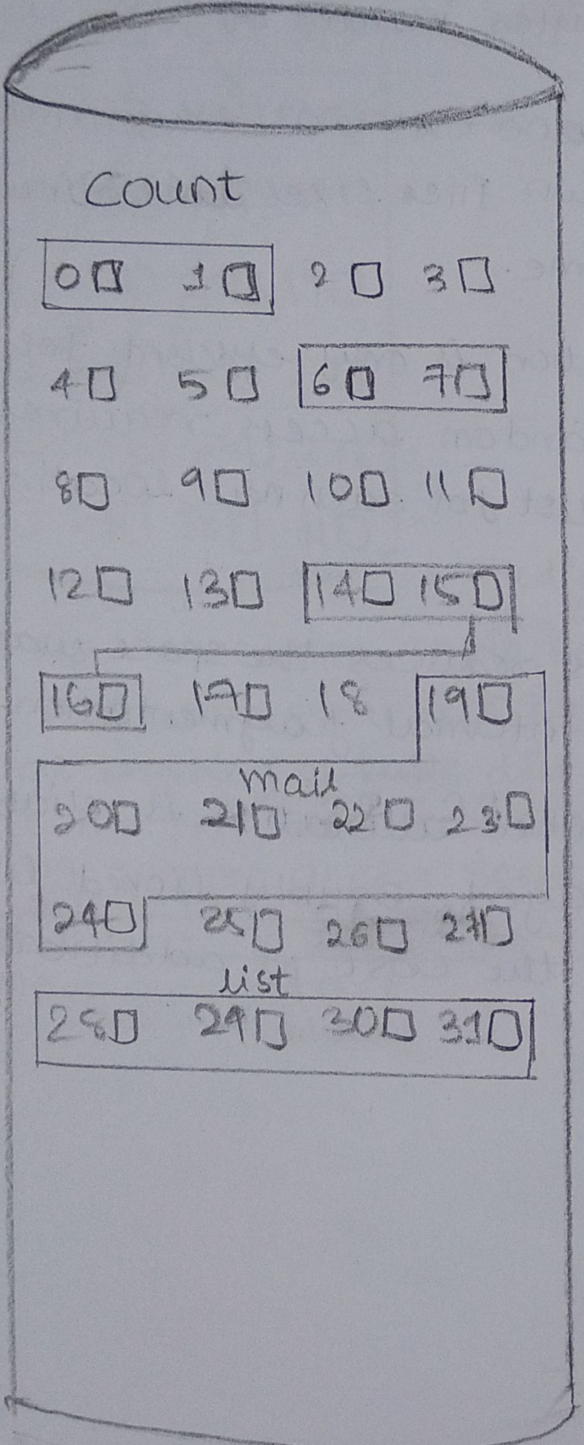
- problems can arise when files grow, or if the exact size of a file is unknown at creation time:

→ over-estimation of the files final size increases external fragmentation and wastes disk space.

→ under-estimation may require that a file be moved or a process aborted if the file grows beyond its originally allocated space.

→ If a file grows slowly over a long time period and the total final space must be allocated initially, then a lot of spaces becomes unusable before the file fills the space.

- A variation is to allocate file space in large contiguous chunks, called extends. when a file outgrows its original extent, then an additional one is allocated. (For example an extent may be the size of a complete track or even cylinder, aligned on an appropriate track or cylinder boundary). The high-performance files system Vertias uses extents to optimize perfomance.



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation of disk space

5.2 Linked Allocation:

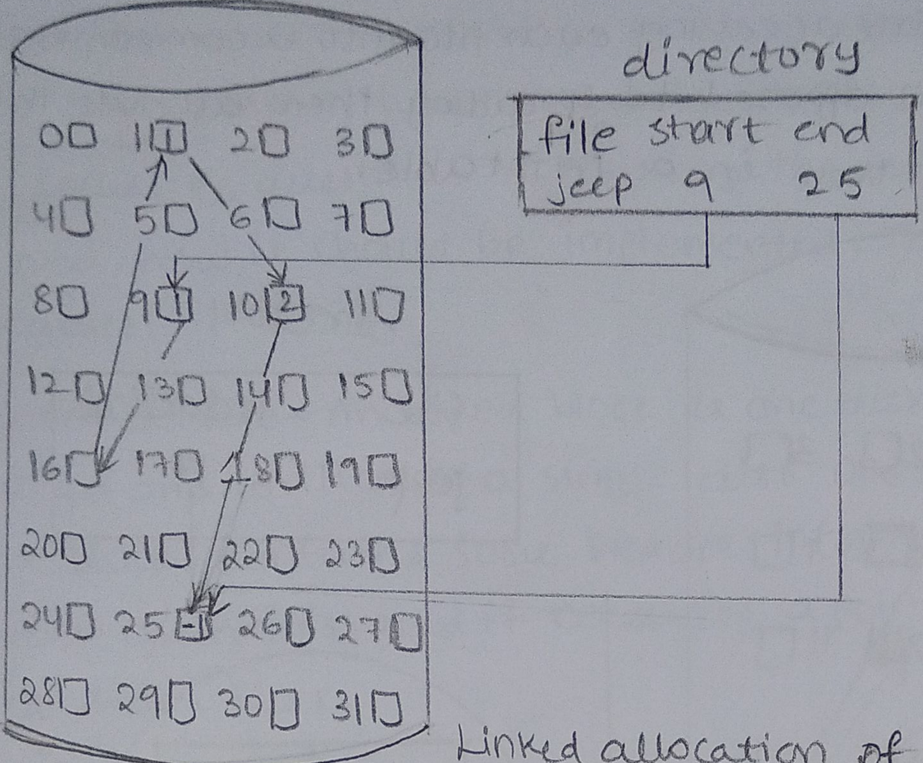
→ Disk files can be stored as linked lists, with the expense of the storage space consumed by each link.
(Eg: a block may be 508 bytes instead of 512).

→ Linked allocation involves no external fragmentation, does not require. pre-known files .sizes, and allows files to grow dynamically at any time.

→ Unfortunately linked allocation is only efficient for sequential access files, as random access requires starting at the beginning of list for each new location access.

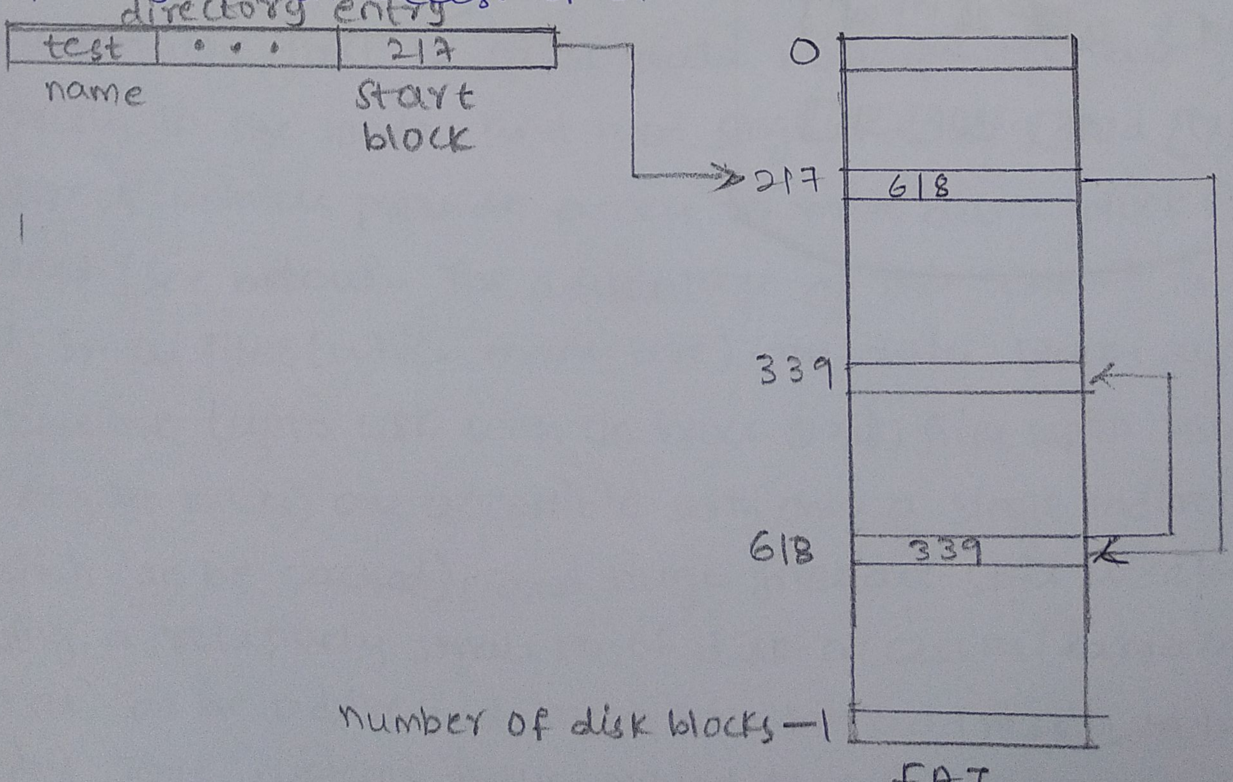
→ Allocating clusters of blocks reduces the space wasted by pointers, at the cost of internal fragmentation.

→ Another big problem with linked allocation is reliability if a pointer is lost or damaged. Doubly linked lists provide some protection, at the cost of additional overhead and wasted space.



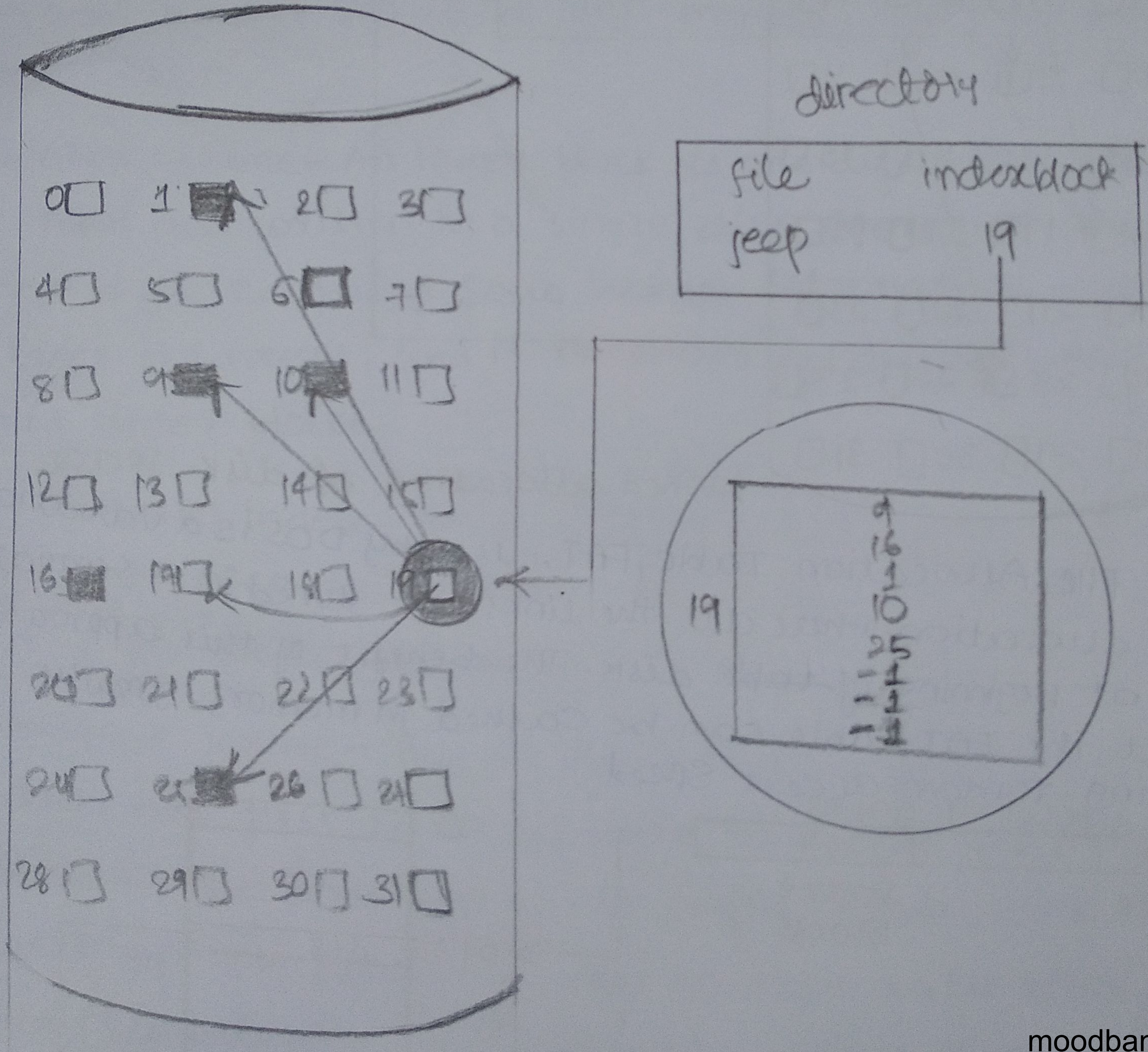
Linked allocation of disk space

→ The File Allocation Table, FAT, used by DOS is a variation of linked allocation, where all the links are stored in a separate table at beginning of ~~table~~ disk. The benefit of this approach is that the FAT table can be cached in memory. greatly improving random access Speed



File allocation table

5.3 Indexed Allocation: Indexed Allocation combines all of the indexes for accessing each file into a common block (for that file), as opposed to spreading them all over the disk or storing them in a FAT table.

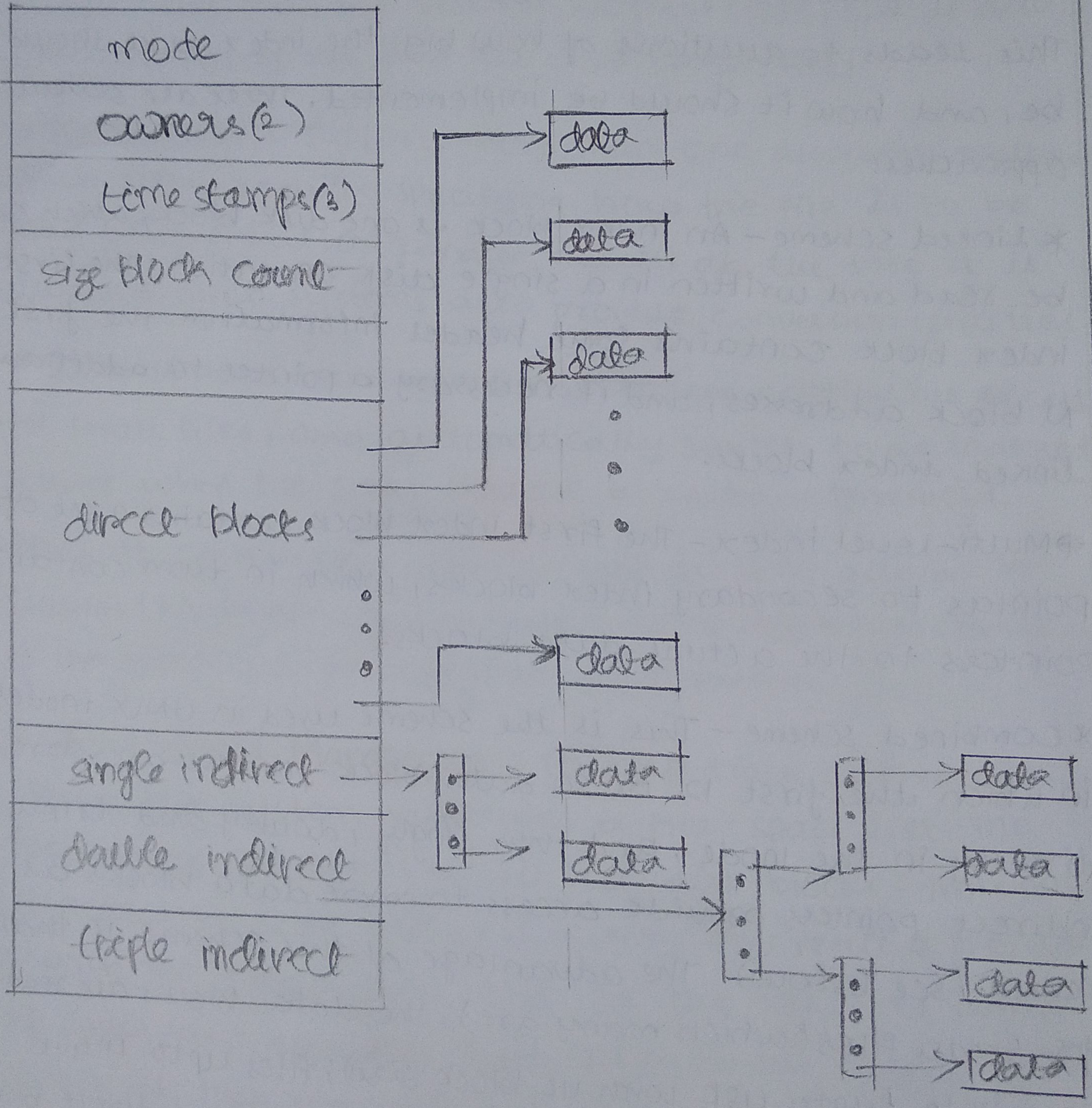


→ Some disk space is wasted (relative to linked lists or FAT table) because an entire index block must be allocated for each file, regardless of how many data blocks the file contains. This leads to questions of how big the index block should be, and how it should be implemented. There are several approaches:

* Linked scheme – An index block is one disk block, which can be read and written in a single disk operation. The first index block contains some header information, the first N block addresses, and if necessary a pointer to additional linked index blocks.

* Multi-Level index – The first index block contains a set of pointers to secondary index blocks, which in turn contain pointers to the actual data blocks.

* Combined scheme – This is the scheme used in UNIX inodes, in which the first 12 or so data block pointers are stored directly in the inode, and then singly, doubly and triply indirect pointers provide access to more data blocks as needed. (see below). The advantage of this scheme is that for small files (which many are), the data blocks are readily accessible (upto 48K with 4K block sizes); files upto 4144K (using 4K blocks) are accessible with only a single indirect block (which can be cached), and huge files are still accessible using a relatively small no. of disk accesses (larger in theory that can be addressed by a 32-bit address, which is why some systems have moved to 64-bit file pointers)


mode
owners(2)
time stamps(3)
size block count
direct blocks
single indirect
double indirect
triple indirect
data
data
data
data
data
data
data
data
data
data

5.4 performance:

- The Optimal allocation method is different for sequential access files than for random access files, and is also diffeent for small files than for large files.

→ Some systems support more than one allocation method, which may reauire specifying how the file is to be used (sequential or random access) at the time it is allocated. such systems also provide conversion utilities.

→ Some systems have been known to use contiguous access for small files, and automatically switch to an indexed scheme when file sizes surpass a certain threshold.

→ And of course some systems adjust their allocation schemes (eg: block sizes) to best match the characteristics of the hardware for optimum performance.

6. Free-Space Management: To keep track of free disk space, the system maintains a free-space list. The free-space list records all free disk blocks - those not allocated to some file or directory. To create a file, we search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from free-space list. when a file is deleted, its disk space is added to the free-space list. Another important aspect of disk management is keeping track of and allocating free space.

6.1 Bit vector:

The free-space list is implemented as a bit map or bit vector. Each block is represented by 1 bit. If the block is free, the bit 1; if the block is allocated, the bit is 0.

For example, consider a disk where blocks 2,3,4,5, 8,9,10,11,12,13,17,18, 25,26 and 27 are free and the rest of the blocks are allocated. The free-space bit map would be.

001111001111110001100000011100000...

The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or n consecutive free blocks on the disk. Indeed, many computers supply bit-manipulation instructions that can be used effectively for that purpose. One techniaue for finding the first free block on a system that uses a bit-vector to allocate disk space is to sequentially check each word in the bit map to see whether that value is not 0, since a 0-valued word contains only 0 bits and represents a set of allocated blocks. The first non-0 word is scanned for the first 1 bit, which is the location of the first free block. The calculation of the block number is

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

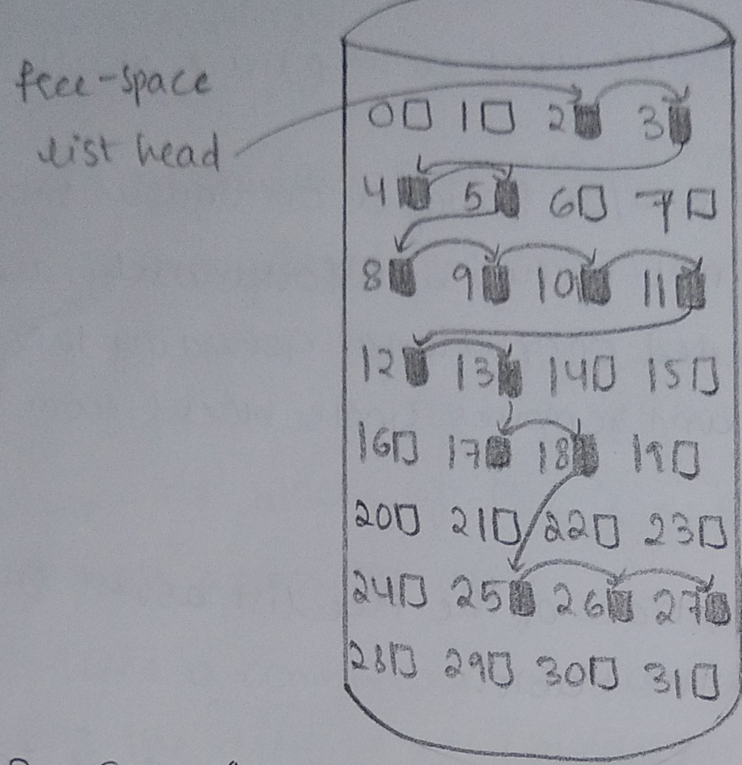
6.2 Linked List:

→ A linked list can also be used to keep track of all free blocks.

→ Traversing the list and/or finding a contiguous block of a given size are not easy, but fortunately are not frequently needed operations. Generally the system just adds and removes single blocks from the beginning of the list.

→ The FAT table keeps track of the free list as just one more linked list on the table.

→ Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block, and so on. Recall our earlier example (section 12.5.1) in which blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 & 27. were free and the rest of the blocks were allocated. In this situation, we would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on. This scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time.



Linked free-space list on disk

6.3 Grouping: A Variation on linked list free lists is to use links of blocks of indices of free blocks. If a block holds up to N addresses, then the first block in the linked list contains upto N-1 addresses of free blocks and a pointer to the next block of free addresses.

6.4 Counting: When there are multiple contiguous blocks of free space then the system can keep track of the starting address of the group and the number of contiguous free blocks. As long as the average length of a contiguous group of free blocks is greater than 2 this offers a savings in space needed for the free list. (similar to compression techniques used for graphics images when a group of pixels all the same color is encountered).

6.5 Space Maps:

→ Sun's ZFS file system was designed for HUGE numbers and sizes of files, directories, and even file systems

→ The resulting data structures could be VERY inefficient if not implemented carefully. For example, freeing up a 1 GB file on a 1 TB file system could involve updating thousands of blocks of free list bit maps if the file was spread across the disk.

→ ZFS uses a combination of techniques, starting with dividing the disk up into (hundreds of) metaslabs of a manageable size, each having their own space map.

→ Free blocks are managed using the counting technique, but rather than write the information to a table, it is recorded in a log-structured transaction record. Adjacent free blocks are also coalesced into a larger single free block.

→ An in-memory space map is constructed using a balanced tree data structure, constructed from the log data.

→ The combination of the in-memory tree and the on-disk log provide for very fast and efficient management of these very large files and free blocks.

File- system structure :

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

1. A disk can be rewritten in place ; it is possible to read a block from the disk, modify the block & write it back into the same place.

2. A disk can access directly any block of information it contains. Thus, it is simple to access any file either sequentially (or) randomly & switching from one file to another requires only moving the read-write heads & waiting for the disk to rotate.

To improve I/o efficiency, I/o transfers between memory & disk are performed in units of blocks. Each block has one or more sectors. Depending on the disk drive, sector size varies from 32 bytes to 4096 bytes ; the usual size is 512 bytes.

File Systems provide efficient & convenient access to the disk by allowing data to be stored, located & retrieved easily. A file system poses two quite different design problems. The first problem is defining how the file system should look to the user. This task involves defining a file & its attributes, the operations allowed on a file, & the directory structure for organizing files. The second problem is creating algorithms & data structures to map the logical file system onto the physical secondary-storage devices.

The file system itself is generally composed of many different levels. The structure shown in below figure is an Example of a layered design. Each level in the design uses the features of lower levels to create a new features for use by higher levels.

Application programs

↓

logical file system

↓

file organization module

↓

Basic file system

↓

I/o Control

↓

devices

fig. Layered file system

The I/o control level consists of device drivers & interrupt handlers to transfer information between the main memory & the disk system. A device driver can be thought of as a translator. Its input consists of high level commands (Eg:- read 123) & output consists of low-level, hardware specific instructions that are used by the hardware controller, which interfaces the I/o device to the rest of the system. The device driver usually writes specific bit patterns to special locations in the I/o controller's memory to tell controller which device location to act on & what actions to take.

→ The basic file system needs only to issue generic commands to the appropriate device driver to read & write physical blocks on the disk. Each physical block is identified by its numeric disk address. This layer also manages the memory buffers & caches that hold various file-system, directory & data blocks.

→ The file-organization module knows about files & their logical blocks, as well as physical blocks. By knowing the type of file allocation used the location of the file, the file-organization module can translate logical block addresses to physical block addresses for the basic file system to tranfer. The file organization module also includes the free-space manager which keeps tracks of unallocated blocks & provides these blocks to the file-organization module when requested.

→ Finally, the logical file system manages meta data information. Metadata information includes all of the file-system structure Except the actual data. The logical file system manages the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name. It maintains file structure via file-control blocks.

File-system Mounting:

A file-system must be mounted before it can be available to processes on the system. The mount procedure is straightforward. The operating system is given the name of the device & the mount point - the location within the file structure where the file system is to be attached.

→ Typically, a mount point is an empty directory

For eg:- on an unix system, a file system containing a user's home directories might be mounted as /home ; then to access the directory structure within that file system, we could precede the directory names with /home as in /home/jane. Mounting that file system under /users would result in the pathname /users/jane, which we could use to reach the same directory.

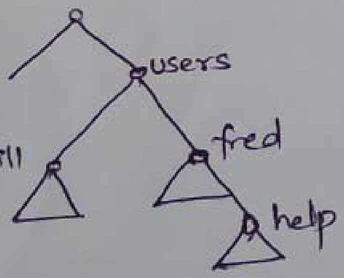


fig.(a) Existing file system

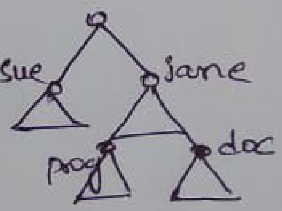


fig.(b) Unmounted Volume

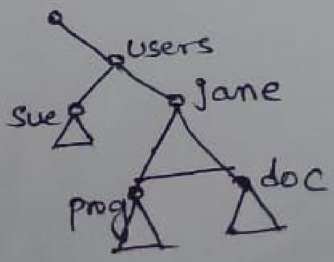
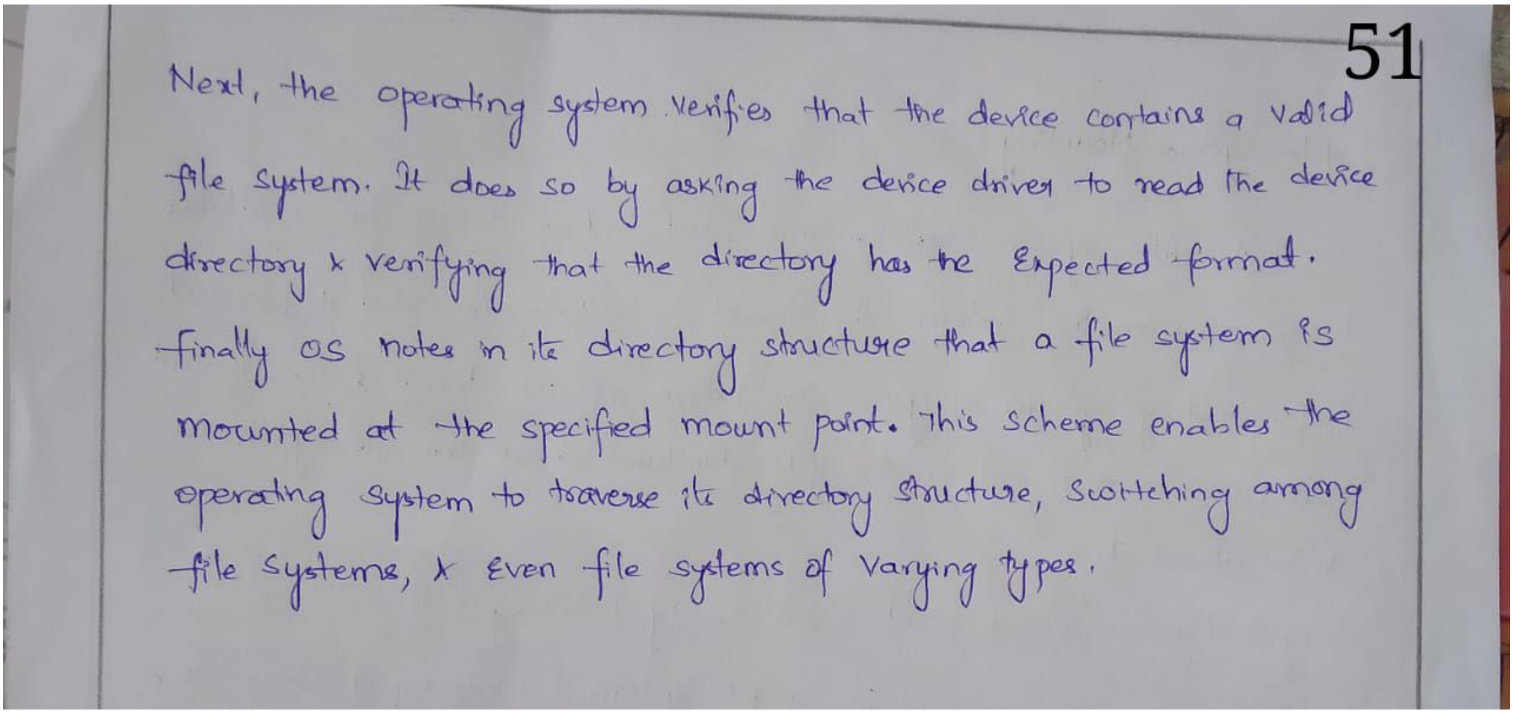


fig. Mount point



Next, the operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory & verifying that the directory has the Expected format. finally OS notes in its directory structure that a file system is mounted at the specified mount point. This scheme enables the operating system to traverse its directory structure, switching among file systems, & Even file systems of Varying types.