File System

Definition:

Computer can store the information on different storage media such as magnetic disk, tapes, etc. and for convenience to use the operating system provides the uniform logical view of the information storage. The operating system abstracts the physical properties of the storage devices to define the logical storage unit called ***file.***

File is a sequence of bits, bytes, lines or records whose meaning is defined by the user or the file creator. Files may be data file, executable file, or any other type. Files are mapped on the physical device by the operating system.

File management:

- File management is the process of facilitating access to, and management and maintenance of, files, folders, and their directory structure.
- Read, write, open, and close functions and permissions facilitate access; create, change, overwrite, and delete functions, facilitate management; and, back-up, file protection, and smooth interaction with the operating system facilitate maintenance.
- In its most basic sense, file management is the process of designing new folders and assigning files to those folders.
- The main goal in file management is to have a system that enables you to find your files quickly and easily, both every day and when you are ready to move your data to a new location.
- A well-designed file management structure can:
- Make it easy to locate important files quickly.
- Keep related documents or files together.
- Make it easier to move groups of data, rather than having to search for individual files, when you move to a new machine.
- Facilitate sharing of appropriate files and protection of non-public files.
- Help you back up important data quickly.

File attributes:

- **Name:** the symbolic file name, in human readable form.
- **Type:** this info need for those systems that support different types.
- **Location:** the pointer that refers to the exact location of the file on that device.
- **Size:** the size of the file and possible the maximum allowed size.
- **Protection:** information about who can read, write or executes the file.
- **Time, date and user identification:** this info is for creation, last modification, and last use.

File operations:

A file is an abstract data type. The operating system provides the system calls to create, write, read, reposition, delete and truncate the files. Following are some operation performed on the files:

- **Create:** two steps are necessary for the file creation, first the space in the file system must be found for the file and second the new file must be made in the directory.
- **Write:** for write we have to specify the name of the file and the info to be written on the file. For writing system must keep the write pointer to the location in the file wherever the next write to take place.
- **Read:** to read from a file we have to specify the name of the file and location. The system needs to keep the read pointer in to the location in the file where the next read is to take place.
- **Repositioning within the file:** the directory is searched for the appropriate entry and the current file position to the given value. This is also known as file *seek.*
- **Deleting the file:** to delete a file we can search for the specified name and after founding that we release all the file space and erase the directory entry.
- **Truncating the file:** sometimes the user wants to keep the attributes of the file as it is and just change the file content, so rather then forcing the user to delete the file and recreate this function will allow to reset the file size and keep all attribute unchanged.

File Access:

There are two ways for accessing information within a file: sequential and direct.

- In sequential access, information in the file must be accessed in the order it is stored in the file. Operation to read or write the file need not specify the logical location within the file. The system maintains a file pointer that determines the location of the next access.
- With direct access, any logical location within the file may be accessed at any time. (a) By specifying the logical location to be accessed as a parameter to read or write operation; (b) by specifying the location in a seek operation to be called before the read or write.

Protection:

User may need to protect the information stored from the unauthorized access as well as from the physical damage. This results in some access control to the file of the user. Access is permitted or denied depends upon many factors. Several different types of operations may be controlled. This may include:

- **Read**: read information contained in the file.
- **Write**: write new information into a file or overwrite existing information in a file.

- **Append:** write new information at the end of a file.
- **Delete**: delete a file and release its storage space.
- **List**: read the names contained in a directory.
- **Execute**: load the contents of a file into main memory and create a process to execute it.
- **Change access**: change some user's access rights for some controlled operation.

Directory structure:

Sometimes the system stores thousands of the files on hundreds of gigabytes of disk. To manage all these data, we need to organize them. This is done in two parts.

- First, the file system is broken into partitions, known as volumes or minidisks. This is the low level structure of file in which files and directories reside.
- Second, each partition contains information about files within it. This information is kept in entries in a device directory or volume table of contents. The device directory records information such as name, size and type for all files on that partition.

There are also some operations which can be performed on a directory.

- **Search for the file**: we need to search the directory structure for the particular file.
- **Create a file**: new files created needs to be added in the directory.
- **Delete a file**: when a file is no longer needed we want to remove it from the directory.
- **List a directory**: we need to be able to list the file in the directory and also contents of the directory entry for each file in the list.
- **Rename a file**: the file name must be changeable when the contents or user of the file changes. Renaming the file may allow its position in the directory to be changed.
- **Traverse the file system**: it is useful to access every file and the directory within the structure. For reliability it is a good idea to save the contents and the structure of the entire file system at regular interval. This provides the backup copy in case of the file system failure.

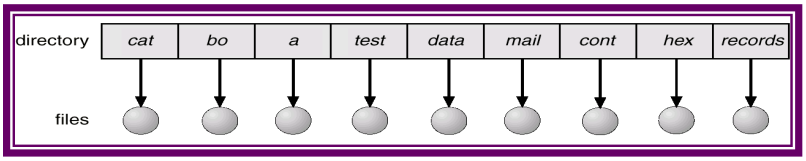
There are some most common schemes for defining the director structures,

- **Single-level Directory**:

  In this, all files are store in the single directory, which is easy to support and understand.

  There are many limitations of this scheme, such as:
  - All files must have unique names.
  - Names are limited to lengths

So it is difficult to remember the hundreds of different names in one directory.

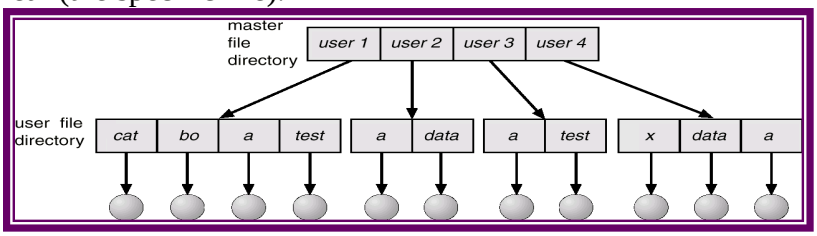


- **Two-level Directory:**

  In this scheme each user has her own user file directory (UFD). Each UFD has a similar structure but lists only files of particular user. When user logs in the system's master file directory (MFD) is searched. The master file directory is indexed by the user name or account number.

  This scheme solves the problem of the naming problem but the disadvantage is if one user wants to cooperate on some task and to access each other's files then the system will not allow them to do so. If the access is permitted the then users will able to do that operation.

  Two level directory structures can be thought of as a tree, where user name and file name defines a path in the tree from root (MFD) to a leaf (the specific file).

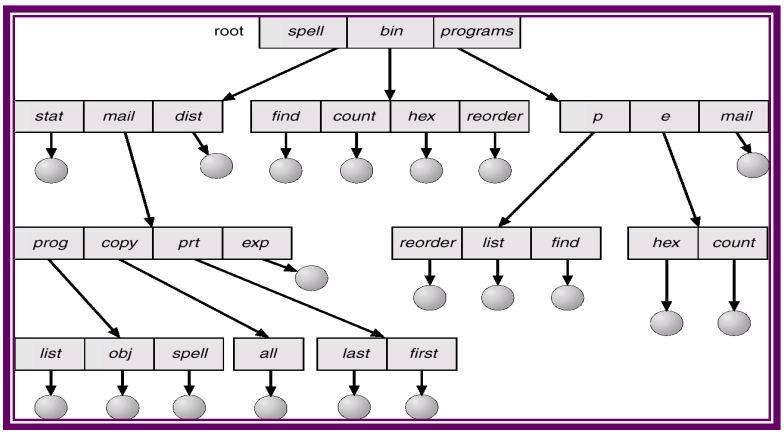


- **Three level Directory:**

  This scheme will allow user to create their own subdirectory and manage their files accordingly. A directory contains a set of files or subdirectories. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1).

  The main question is of deletion of the directory. If a directory is empty then it can be simply deleted. If the directory is not empty then some system will

not allow deleting them until it is empty (MS-DOS). On the other hand some systems will ask the user to delete all the content of that directory (UNIX).



There are some more other features of the file system also, but because of the complexity and our limitations we could not include those.

After having this basic idea about the file system of the operating system we will go to the next section which describes our project and also what it does.

Project Description

In this project we are simulating the UNIX like file system that works properly and shows some functionalities of the file system.

COMMANDS: This section will explain the commands that are used in our project.

1. **mkfs:** this command will take 5 arguments and actually creates the file system according to the user input and correctly scale the internal data structure.
2. **mount:** this will copy the super block in to the separate memory block that is apart from the superblock within the file system.
3. **unmount:** this will unmount the file system and also frees the space used by the file system.
4. **mkdir:** this works just like mkdir in UNIX. It will create a directory according to the user input. The length of the name of the directory is only limited to 8 characters.
5. **ls:** this also works like ls command in UNIX. It will list all the directories and files within the file system or within specific directory.
6. **write:** It will write the string entered by the user into the specific file. If the file is not present then it will create a file and then writes and if the file is already there it will overwrite it.
7. **read:** this will read the content of the specific file or directory. If anything written in the file, it will print it otherwise works as the ls command.
8. **symlink:** this command will establishes the soft link between the two parameters entered by the user. Make sure that the first argument is the file to which we want to point and the second is the pointer. That means if we do symlink x y and then if we read the y then it will read the contents of x.
9. **part:** this will print the current status of the super block contained on the partition, not the mounted super block. This is to make sure that no data is written on the partition super block.
10. **live:** this will print the current status of the live superblock and prints the change that has been made during the creation of the directories and the files.
11. **help:** this will list the commands of this program.
12. **quit:** this will unmount the file system and quit the program.

Abstract data types:

In this program superblock s completely scalable except the file system will not fit in the defined block size. In this program the disk is a global variable and also a block pointer. The amount of the memory allocated to the global_disk can be changed by disk_create. The block contains for data types,

- Superblock: this is a struct of the 5 integers and 2 pointers and struct of 8 bit fields. This means each bitmap is the array of the “ibits”, where ibits are the union of the char and 8 bits.
- Data block: it is a struct containing the character pointer. It is used for the commands such as read, write, symlink.

- Directory block: it is a struct of pointer that has 7 bytes and 8 bit inode number field. Used for listing and making directories.
- Inode block: pointer to inode struct, to which memory is allocated after mkfs runs. They are static in size, 4 byte for block_number, 3 bytes for block_size, and 1 byte for block_type.

Here is the list of all the functions and their short description that are used in our program.

HEADER FILE:

1.) Relevant structures and unions are defined for:

- **Directory_Name_(struct)** - includes character array of seven characters and unsigned char inode number.
- **Bitmap (union)** - includes unsigned char data and instance of ibit.
- **Inode Block (struct)** - contains inode pointer to nodes.
- **Superblock_(struct)** - include arguments ri, is, bs, fs, and ib, as explained below, and bitmap pointers to inode_block and block_bitmap.
- **Data block (struct)** - contains char pointer to data.
- **Directory block (struct)** - cointains dir_entry pointer to dirs.
- **Block (union)** - includes instances of dirblock, inode_block, data_block, and superblock.

The union "block" is defined by instances from: Directory block, Inode block; Data block; and Superblock. This is used for creating a new file system in C source file **parse.c** by pointing to it.

2.) Function Prototyping:

- **block * disk_create** - Takes 5 arguments (ri = root inode; is = inode size; bs = block size; fs = file system size; and ib = number of blocks devoted to inodes). This is used to point to new file system instance of the superblock struct, which is represented as 'a'. a global superblock is then created from instance 'a', which invokes the qualities of the superblock created from the superblock_create function. Dynamic sizes for each of the block types in the "block" union are then created, making sure that the size of the blocks are constant and all portions of the blocks are accessible by any reference in the "block" union. Root inode is then mounted and the global superblock is partitioned, which is an initialization record of the file system.
- **superblock_superblock_create** - Creates and returns a fully functional superblock used by the disk function to create the global disk's superblock.
- **superblock_print** - Prints out the contents of a given superblock, given the size of the file system. The file system size is needed in order to print out the bitmaps in the right size. Bitmaps are used to identify available blocks for the file system's use.
- **print_bmp_entry** - Prints a bitmap. Used for error checking.

- **superblock_get** - Initializes the global modify-able superblock in memory to the value of the partition's superblock. Used as a mount command.
- **superblock_put** - Initializes the partition's superblock to the value of the global superblock's. Used as the unmount command.
- **free_bmp** - Takes a bmp_entry pointer to instance 'a' and parameter "size", which is the size of the given bitmap, and finds free space.
- **inode_free** - Takes parameter int inode_number and frees the parameter in the live version of the inode bitmap.
- **inode_get** - Takes parameter int inode_number and returns its inode data structure.
- **inode_put** - Takes parameter int inode_number and the depth of the inode. Corresponds to the two dimensional array positions of the inode in the partition blocks.
- **print_inode** - Prints a given inode by its block number and inode number.
- **block_free** - Frees an associated block number's position in the block bitmap in the active superblock. Takes int block_number as parameter.
- **block_put** - Takes instance of block union, "block a", and int inode_number and puts the block instance in the associating block number. (inode num = data_num).
- **block_get** - Takes int block_number and returns a block data type given the data_num value of inode.
- **block_print** - Prints block given int block_number.

C SOURCE FILE: parse.c

1.) Definitions of functions:

- **int parse()** - Primary function. It uses an array of function pointers to call functions by index. When entered, int parse() will return data based the index of menu option selected. It will return 0 from "quit", CMD -1 if valid, and CMD if invalid command. CMD is the index of the array option menu, which offers options according to file system creation. If invalid option is selected, int parse () tests option string. If string != to '\n' then it prints error. Else it tests of option is 0, in which case the program quits.
- **int quit()** - Prints "Quitting…".

- **int mkfs()** - Creates a new disk with superblock data taken from the user in the parse() function and verified internally. The function first creates instance argument 'a' and checks whether arguments were entered correctly by amount and contradiction. If all arguments meet requirements then the function attempts to create file system. If no file systems exist the disk_create function is invoked. Else, if a file system exists, the functions prompts the user whether to create a new. The function alerts the user that creating a new file system will destroy the current file system. If arguments don't meet function requirements, then function aborts. After a file system has successfully been created the file system level flag is set to 1. This informs the program that a file system exists.

- **int mount()** - The mount function simply creates a new copy of the file system's superblock in memory in order to manipulate with other functions. It utilizes the superblock_get function in the header file.
- **int unmount()** - The unmount function simply copies the contents from the live superblock to the simulated disk partition's superblock. It utilizes the superblock_put function in the header file.
- **int parsePath()** - The parsePath function takes a character pointer array and returns an inode number. If the path passed is NULL, returns the root inode. Else, path is copied to pathCopy and parsed. When parsed, if file entry was previously parsed, return error. Else, for every directory entry, if entry matches token, which is initialized inside the parse for loop by the pathCopy pointer, the function gets the token's inode of matching entry and sets cases for token's inode type, F (set flag) or S (write-over for symbolic links).
- **int mkdir()** - Takes one argument, directory name, and tests for length. If name over seven characters the function returns error. If argument meets requirements and entry limit per directory is not met, instance "inode a" is set by structure parameters.
- **int write()** - Creates a file. Inode parent and file instances are created, as well as char pointers to path, pathToken and filename. Before creating the file the function tests whether the file system is mounted with "if (fs_r >1)". If mounted the function fetches the data to be written, tests for errors; parses path into inode number of file's parent directory; and sets cases for parent's block type, F (set flag) or S (write-over for symbolic links). Once a file is set for creation, for each directory entry if entry matches the token the directory entry inode number is converted into an inode variable called "file" and the data argument is copied into the file variable's datablock's data field. If function specifics aren't met, as in no more room for file to be created in a given directory, an error is returned specifying the problem.
- **int ls()** - Function lists files in a given directory. int ls() first tests whether the file system is mounted. If mounted it fetches path from function parameters and parses path into the inode number of the current directory. The function then tests whether path is a directory. If not it exits, else it tests for matching token. If token matches the items in the directory are printed.

- **int read()** - int read() outputs file from the partition in raw character form. Again, as in other functions above, int read() tests whether file system is mounted. If so, it tests for name argument by character in for loop. If argument is met, it's stored in temp block by function parsePath() and printed on screen.
- **int symlin()** -int symlink() firstly sets a flag so that parsePath() does not active on symbolic link. Refer to int write() for rest of function actions, using case 'S'.
- **int live()** - This function checks if the file system is mounted and displays the global superblock in order to verify that the mounted superblock is being manipulated.

- **int part()** - int part()checks if the file system is mounted and displays the superblock on the partition in order to make sure that the partition's superblock does not get changed when modifying global superblock.

Conclusion: after understanding the file system, we can say that this is the only part of the operating system which can be physically used by the users. File system is actively connected with the memory allocation and also with the memory management.

In our project we could not simulate some functions of file systems such as delete, rename, and also protection features. We have tried to create those functions but then the program was not working properly because of its structure.