RV College of Engineering®
Mysore Road, RV Vidyaniketan Post,
Bengaluru - 560059, Karnataka, India

Go, change the world®

Experiential Learning Report
on

"Custom File System for Linux"

Submitted by

S Sahana (1RV22CS166)

Prajwal M (1RV22CS140)

Submitted in
partial fulfillment for the award of degree
of
BACHELOR OF ENGINEERING
in
Computer Science and Engineering,

Course: Operating Systems
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

2023-24

Problem Statement:
Design and implement a file system for Linux operating systems that provides efficient storage, organization, and management of files and directories on disk, ensuring data integrity and reliability.

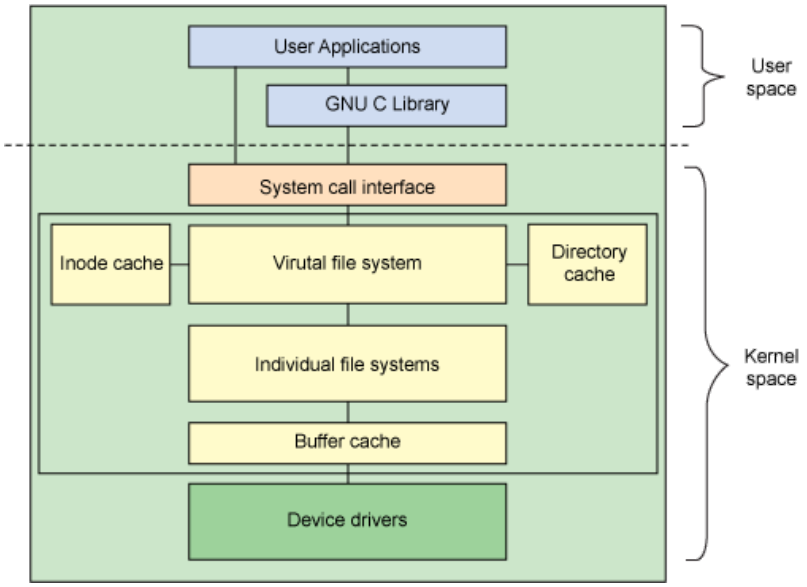
Introduction to Problem:
A file system is a method an operating system uses to store, organize, and manage files and directories on a storage device. Eg: FAT (File Allocation Table), NTFS (New Technology File System), ext (Extended File System)
File system implementation in an operating system provides several advantages, including efficient data storage, data security, data recovery, improved performance, scalability, flexibility, and cross-platform compatibility. File system implementation in an operating system refers to how the file system manages the storage and retrieval of data on a physical storage device such as a hard drive, solid-state drive, or flash drive.

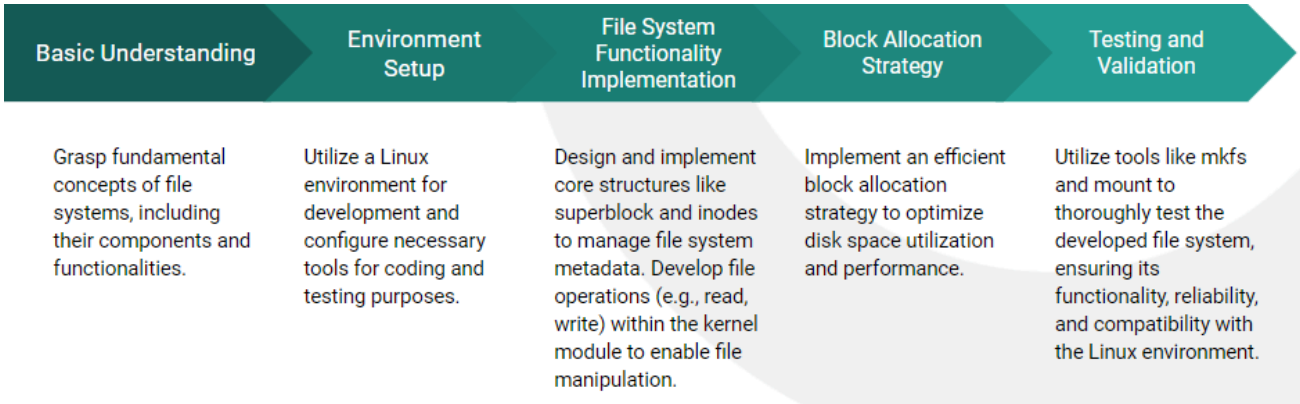
- File System Structure
- File Allocation
- Data Retrieval
- Security and Permissions

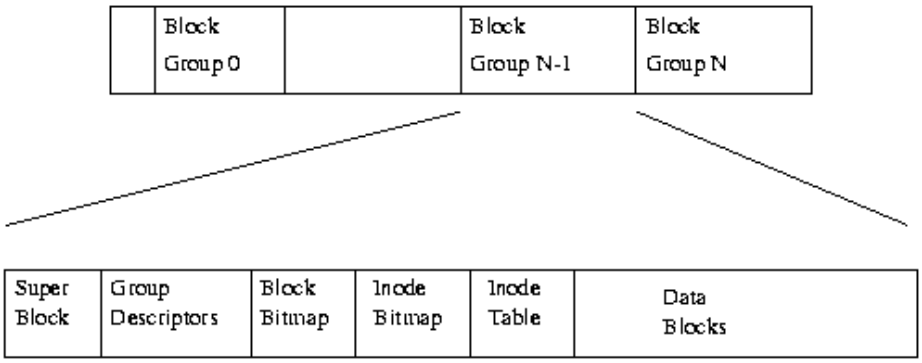
Relevant APIs and System Calls:

- **Dynamic memory allocation functions** like malloc() can be used to allocate memory for the file system structures dynamically.
- Data structures **bitmaps** are used to represent file system structures such as inodes.
- **System calls** are required when a file system wants to create or delete files. System calls are also used for the creation and management of new processes and access to hardware devices like keyboards.
    - malloc() and free(): System calls for dynamic memory allocation and deallocation.
    - memcpy(): Used to copy memory blocks, potentially for copying file data to data blocks.
    - memset(): Used to set memory blocks to a specified value, potentially for initializing file system structures.
- **APIs** provide a layer of abstraction that allows developers to interact with the operating system and other software components in a more standardized and consistent manner.
- printf() and scanf(): Standard library functions for formatted input/output to the console.

Design of the problem:







- Super Block: Contains information about the file system, such as the total number of inodes, total number of data blocks, and counts of free inodes and free data blocks.
- Data Blocks: Actual data storage for files. These blocks are allocated as needed and contain the file content.
- Data Block Bitmap: Represents the allocation status of data blocks in the file system.
- Inode Bitmap: Represents the allocation status of inodes in the file system.
- Inode Table: Contains inode structures, each representing a file in the file system. Each inode structure includes information such as inode number, file size, number of data blocks allocated, block indexes of allocated data blocks, and file name.

Super Block: aligned to 1024 byte boundary.

```
struct super_block {
  u_int total_inodes;
  u_int total_datablocks;
  u_int free_inodes;
  u_int free_datablocks;
};
```

Data block bit map:

-> 1024 bits are vaild, representing 1024 blocks of the block group and rest of the bits are invalid.

-> A bit is set when the block is allocated to a particular inode.

-> A bit is cleared when the block is deallocated or removed from the inode

Inode bit map:

-> 1024 bit are valid representing an inode and the rest of the bits are invalid

-> A bit position is an index into the inode table which contains 1024 inode structures.

-> A bit is set when the corresponding inode is allocated.

-> A bit is cleared, when the corresponding inode is deallocated.

Table of 1024 inode structure: Each inode structure has a corresponding bit representation in the inode bit map i.e., there is a one-to-one mapping between bit position in the inode bitmap and index into the inode table.

Data Blocks:

-> Free data blocks are allocated when needed.

-> First 25 blocks of the block group are occupied by super block, inode block bitmap, data block bitmap, inode table and the remaining 999 blocks are available for allocation for files.

-> There is one-to-one mapping between bit position in the data block bitmap and the data block.

Source code:

```
/* command  action
 * -------  ------
 *  root    initialize root directory
 *  print   print current working directory and all descendants
 *  chdir   change current working directory (.. refers to parent directory)
 *  mkdir   sub-directory create
 *  rmdir                delete
 *  mvdir                rename
 *  mkfil   file create
 *  rmfil        delete
 *  mvfil        rename
 *  szfil        resize
 *  exit        quit the program
 */

int debug = 1;

int do_root (char *name, char *size);
int do_print(char *name, char *size);
int do_chdir(char *name, char *size);
int do_mkdir(char *name, char *size);
int do_rmdir(char *name, char *size);
int do_mvdir(char *name, char *size);
int do_mkfil(char *name, char *size);
int do_rmfil(char *name, char *size);
int do_mvfil(char *name, char *size);
int do_szfil(char *name, char *size);
int do_exit (char *name, char *size);
```

Root initialization:

```
int do_root(char *name, char *size)
{
    (void)*name;
    (void)*size;
    if ( disk_allocated == true )
        return 0;


    disk = (char*)malloc ( DISK_PARTITION );
        if ( debug ) printf("\t[%s] Allocating [%d] Bytes of memory to the disk\n", __func__, DISK_PARTITION );


    add_descriptor("descriptor");
        if ( debug ) printf("\t[%s] Creating Descriptor Block\n", __func__ );
    add_directory("root");
        if ( debug ) printf("\t[%s] Creating Root Directory\n", __func__ );


    strcpy(current.directory, "root");
    current.directory_index = 3;
    strcpy(current.parent, "" );
    current.parent_index = -1;
        if ( debug ) printf("\t[%s] Set Current Directory to [%s], with Parent Directory [%s]\n", __func__, "root", "" );

    if ( debug ) printf("\t[%s] Disk Successfully Allocated\n", __func__ );
    disk_allocated = true;

    return 0;
}
```

Change current working directory:

```
int do_chdir(char *name, char *size)
{
    (void)*size;
    if ( disk_allocated == false ) {
        printf("Error: Disk not allocated\n");
        return 0;
    }


    if ( strcmp(name, ".." ) == 0 ) {


        if ( strcmp(current.directory, "root") == 0 )
            return 0;


        strcpy ( current.directory, current.parent );
        strcpy (current.parent, get_directory_top_level( current.parent) );
            if ( debug ) printf ("\t[%s] Current Directory is now [%s], Parent Directory is [%s]\n", __func__, current.directory, curren
        return 0;
    }
    else
    {
        char tmp[20];

        if ( (strcmp(get_directory_subitem(current.directory, -1, name), "-1") == 0) && strcmp( current.parent, name ) != 0 ) {
            if ( debug ) printf( "\t\t\t[%s] Cannot Change to Directory [%s]\n", __func__, name );
            if (!debug ) printf( "%s: %s: No such file or directory\n", "chdir", name );
            return 0;
        }

        strcpy( tmp, get_directory_name(name));
        if ( strcmp(tmp, "") == 0 )
            return 0;

        if ( strcmp( tmp, name ) == 0 ) {
            strcpy ( current.directory, tmp);
```

Create subdirectory:

```
int do_mkdir(char *name, char *size)
{
    (void)*size;
    if ( disk_allocated == false ) {
        printf("Error: Disk not allocated\n");
        return 0;
    }


    if ( get_directory_subitem(current.directory, -1, name) == 0  ) {
            if ( debug ) printf( "\t\t\t[%s] Cannot Make Directory [%s]\n", __func__, name );
            if (!debug ) printf( "%s: cannot create directory '%s': Folder exists\n", "mkdir", name );
            return 0;
        }


    if ( debug ) printf("\t[%s] Creating Directory: [%s]\n", __func__, name );
    if ( add_directory( name ) != 0 ) {
        if (!debug ) printf("%s: missing operand\n", "mkdir");
        return 0;
    }


    edit_directory( current.directory, name, NULL, false, true );
        if ( debug ) printf("\t[%s] Updating Parents Subitem content\n", __func__ );

    if ( debug ) printf("\t[%s] Directory Created Successfully\n", __func__ );
    if( debug ) print_directory(name);

    return 0;
}
```

Rename directory:

```
int do_mvdir(char *name, char *size) //"size" is actually the new name
{
    if ( disk_allocated == false ) {
        printf("Error: Disk not allocated\n");
        return 0;
    }

    //Rename the directory
    if ( debug ) printf("\t[%s] Renaming Directory: [%s]\n", __func__, name );
    //if the directory "name" is not found, return -1
    if( edit_directory( name, "", size, true, true ) == -1 ) {
        if (!debug ) printf( "%s: cannot rename file or directory '%s'\n", "mvdir", name );
        return 0;
    }

    //else the directory is renamed
    if (debug) printf( "\t[%s] Directory Renamed Successfully: [%s]\n", __func__, size );
    if (debug) print_directory(size);
    return 0;
}
```

Create file:

```
int do_mkfil(char *name, char *size)
{
    if ( disk_allocated == false ) {
        printf("Error: Disk not allocated\n");
        return 0;
    }

    if ( debug ) printf("\t[%s] Creating File: [%s], with Size: [%s]\n", __func__, name, size );

    //If it returns 0, there is a subitem with that name already
    if ( get_directory_subitem(current.directory, -1, name) == 0  ) {
            if ( debug ) printf( "\t\t\t[%s] Cannot make file [%s], a file or directory [%s] already exists\n", __func__, name, name );
            if (!debug ) printf( "%s: cannot create file '%s': File exists\n", "mkfil", name );
            return 0;
        }

    if ( add_file ( name, atoi(size)) != 0 )
        return 0;

    //Edit the current directory to add our new file to the current directory's "subdirectory" member.
    edit_directory( current.directory, name, NULL, false, false);
        if ( debug ) printf("\t[%s] Updating Parents Subitem content\n", __func__ );

    if ( debug ) print_file(name);
    return 0;
}
```

Remove file:

```
// Remove a file
int do_rmfil(char *name, char *size)
{
    if ( disk_allocated == false ) {
        printf("Error: Disk not allocated\n");
        return 0;
    }

    (void)*size;
    if ( debug ) printf("\t[%s] Removing File: [%s]\n", __func__, name);

        //If the file to be removed actually exists in current directory, remove it
    if ( get_directory_subitem(current.directory, -1, name) == 0  ) {
            remove_file(name);
            return 0;
        }
    else{ // If it doesn't exist, print error and return 0
        if ( debug ) printf( "\t\t\t[%s] Cannot remove file [%s], it does not exist in this directory\n", __func__, name );
        if (!debug ) printf( "%s: %s: No such file or directory\n", "rmfil", name );
        return 0;
    }
}
```

Print all directories and files:

```
int do_print(char *name, char *size)
{
    (void)*name;
    (void)*size;
    if ( disk_allocated == false ) {
        printf("Error: Disk not allocated\n");
        return 0;
    }

    printing("root");

    if (debug) if ( debug ) printf("\n\t[%s] Finished printing\n", __func__);
    return 0;
}
```

Rename a File:

```
// Rename a file
int do_mvfil(char *name, char *size)
{
    if ( disk_allocated == false ) {
        printf("Error: Disk not allocated\n");
        return 0;
    }

    if ( debug ) printf("\t[%s] Renaming File: [%s], to: [%s]\n", __func__, name, size );


    if ( get_directory_subitem(current.directory, -1, size) == 0  ) {
            if ( debug ) printf( "\t\t\t[%s] Cannot rename file [%s], a file or directory [%s] already exists\n", __func__, name, size );
            if (!debug ) printf( "%s: cannot rename file or directory '%s'\n", "mvfil", name );
            return 0;
        }

    int er = edit_file( name, 0, size);

    if (er == -1) return -1;
    if (debug) print_file(size);

    return 0;
}
```

Resize a file:

```
int do_szfil(char *name, char *size)
{
    if ( disk_allocated == false ) {
        printf("Error: Disk not allocated\n");
        return 0;
    }

    if ( debug ) printf("\t[%s] Resizing File: [%s], to: [%s]\n", __func__, name, size );

    if (remove_file(name) != -1)  do_mkfil(name, size);

    else {
        if ( debug ) printf("\t[%s] File: [%s] does not exist. Cannot resize.\n", __func__, name);
        if (!debug ) printf( "%s: cannot resize '%s': No such file or directory\n", "szfil", name );
    }

    return 0;
}
```

Output:

```
PS D:\git_new1\os_el> cd "d:\git_new1\os_el\show\" ; if ($?) { gcc fs.c -o fs } ; if ($?) { .\fs }
Welcome to your file system
root
:root:::
        [do_root] Allocating [4000000] Bytes of memory to the disk
                [add_descriptor] Allocating Space for Descriptor Block
                [add_descriptor] Allocating Space for Descriptor's Name Member
                [add_descriptor] Initializing Descriptor to Have All of Memory Available
                [add_descriptor] Updating Descriptor to Show that first [2] Memory Blocks Are Taken
        [do_root] Creating Descriptor Block
                [add_directory] Allocating Space for New Folder
                        [allocate_block] Finding Free Memory Block in the Descriptor
                        [allocate_block] Allocated [root] at Memory Block [1]
                [add_directory] Assigning New Folder to Memory Block [1]
                [add_directory] Folder [root] Successfully Added
        [do_root] Creating Root Directory
        [do_root] Set Current Directory to [root], with Parent Directory []
        [do_root] Disk Successfully Allocated
```

```
mkdir docs 4
:mkdir:docs:4:
                        [find_block] Searching Descriptor for [root], which is a [Folder]
                        [find_block] Found [root] at Memory Block [1]
                        [get_directory_subitem] Did Not Find [docs] as a Subitem of Directory [root]
        [do_mkdir] Creating Directory: [docs]
                [add_directory] Allocating Space for New Folder
                        [allocate_block] Finding Free Memory Block in the Descriptor
                        [allocate_block] Allocated [docs] at Memory Block [2]
                [add_directory] Assigning New Folder to Memory Block [2]
                [add_directory] Folder [docs] Successfully Added
                        [find_block] Searching Descriptor for [root], which is a [Folder]
                        [find_block] Found [root] at Memory Block [1]
                [edit_directory] Folder [root] Found At Memory Block [1]
                [edit_directory] Added Subitem [docs] at Subitem index [0] to directory [root]
                [edit_directory] Folder [root] Now Has [1] Subitems
        [do_mkdir] Updating Parents Subitem content
        [do_mkdir] Directory Created Successfully
                        [find_block] Searching Descriptor for [docs], which is a [Folder]
                        [find_block] Found [docs] at Memory Block [2]
        ----------------------------
        New Folder Attributes:

        name = docs
        top_level = root
        subitems =
        subitem_count = 0
        ----------------------------
```

```
rmdir hello
:rmdir:hello::
                        [find_block] Searching Descriptor for [root], which is a [Folder]
                        [find_block] Found [root] at Memory Block [1]
                        [get_directory_subitem] Found [hello] as a Subitem of Directory [root]
                        [find_block] Searching Descriptor for [hello], which is a [Folder]
                        [find_block] Found [hello] at Memory Block [2]
                        [find_block] Searching Descriptor for [root], which is a [Folder]
                        [find_block] Found [root] at Memory Block [1]
        [do_rmdir] Removing Directory: [hello]
                        [find_block] Searching Descriptor for [hello], which is a [Folder]
                        [find_block] Found [hello] at Memory Block [2]
                        [unallocate_block] Unallocating Memory Block [2]
        [do_rmdir] Directory Removed Successfully
```

```
chdir ..
:chdir:..::
                        [find_block] Searching Descriptor for [root], which is a [Folder]
                        [find_block] Found [root] at Memory Block [1]
                        [get_directory_top_level] top_level [] found for [root] folder
        [do_chdir] Current Directory is now [root], Parent Directory is []
print
:print:::
                        [find_block] Searching Descriptor for [root], which is a [Folder]
                        [find_block] Found [root] at Memory Block [1]
root:
        docs
                        [find_block] Searching Descriptor for [docs], which is a [Folder]
                        [find_block] Found [docs] at Memory Block [2]
docs:
        hello

        [do_print] Finished printing
exit
:exit:::
        [do_exit] Exiting
PS D:\git_new1\os_el\show>
```

```
mvdir docs dot
:mvdir:docs:dot:
        [do_mvdir] Renaming Directory: [docs]
                        [find_block] Searching Descriptor for [docs], which is a [Folder]
                        [find_block] Found [docs] at Memory Block [2]
                [edit_directory] Folder [docs] Found At Memory Block [2]
                        [find_block] Searching Descriptor for [dot], which is a [Folder]
                        [find_block] Block Not Found: Returning -1
                [edit_directory] Folder [docs] Now Has Name [dot]
                [edit_descriptor] Descriptor Name Member now shows Memory Block [2] has Name [dot]
                [edit_directory] Updated Descriptor's Name Member
                        [find_block] Searching Descriptor for [dot], which is a [Folder]
                        [find_block] Found [dot] at Memory Block [2]
        ----------------------------
        New Folder Attributes:

        name = dot
        top_level = root
        subitems =
        subitem_count = 0
        ----------------------------
                        [find_block] Searching Descriptor for [root], which is a [Folder]
                        [find_block] Found [root] at Memory Block [1]
                [edit_directory] Folder [root] Found At Memory Block [1]
                [edit_directory] Edited Subitem [docs] to [dot] at Subitem index [0] for directory [root]
                [edit_directory] Updated Parents Subitem Name
        [do_mvdir] Directory Renamed Successfully: [dot]
                        [find_block] Searching Descriptor for [dot], which is a [Folder]
                        [find_block] Found [dot] at Memory Block [2]
        ----------------------------
        New Folder Attributes:

        name = dot
        top_level = root
        subitems =
        subitem_count = 0
        ----------------------------
```

```
mkfil hello.c
:mkfil:hello.c::
        [do_mkfil] Creating File: [hello.c], with Size: []
                        [find_block] Searching Descriptor for [root], which is a [Folder]
                        [find_block] Found [root] at Memory Block [1]
                        [get_directory_subitem] Did Not Find [hello.c] as a Subitem of Directory [root]
                [add_file] Allocating Space for New File
                [add_file] Initializing File Members
                        [allocate_block] Finding Free Memory Block in the Descriptor
                        [allocate_block] Allocated [hello.c] at Memory Block [3]
                [add_file] Allocating [0] Data Blocks in Memory for File Data
                        [allocate_block] Finding Free Memory Block in the Descriptor
                        [allocate_block] Allocated [hello.c->0] at Memory Block [4]
                [add_file] File [hello.c] Successfully Added
                        [find_block] Searching Descriptor for [root], which is a [Folder]
                        [find_block] Found [root] at Memory Block [1]
                [edit_directory] Folder [root] Found At Memory Block [1]
                [edit_directory] Added Subitem [hello.c] at Subitem index [1] to directory [root]
                [edit_directory] Folder [root] Now Has [2] Subitems
        [do_mkfil] Updating Parents Subitem content
                        [find_block] Searching Descriptor for [hello.c], which is a [File]
                        [find_block] Found [hello.c] at Memory Block [3]
        ----------------------------
        New File Attributes:

        name = hello.c
        top_level = root
        file size = 0
        block count = 1
        ----------------------------
```

```
mvfil hello.c dot
:mvfil:hello.c:dot:
        [do_mvfil] Renaming File: [hello.c], to: [dot]
                        [find_block] Searching Descriptor for [root], which is a [Folder]
                        [find_block] Found [root] at Memory Block [1]
                        [get_directory_subitem] Did Not Find [dot] as a Subitem of Directory [root]
                        [find_block] Searching Descriptor for [hello.c], which is a [File]
                        [find_block] Found [hello.c] at Memory Block [3]
                [edit_file] File [hello.c] Found At Memory Block [3]
                        [find_block] Searching Descriptor for [hello.c], which is a [File]
                        [find_block] Found [hello.c] at Memory Block [3]
                        [get_file_top_level] top_level [root] found for [hello.c] file
                        [find_block] Searching Descriptor for [root], which is a [Folder]
                        [find_block] Found [root] at Memory Block [1]
                        [edit_directory_subitem] Edited subitem in root from hello.c to dot
                        [edit_file] File [hello.c] Now Has Name [dot]
                        [find_block] Searching Descriptor for [dot], which is a [File]
                        [find_block] Found [dot] at Memory Block [3]
        ----------------------------
        New File Attributes:

        name = dot
        top_level = root
        file size = 0
        block count = 1
        ----------------------------
```

Conclusion:

In conclusion, our custom file system, featuring an indicated block allocation strategy, optimizes storage efficiency by dynamically allocating space, mitigating fragmentation concerns. Unlike partitioned secondary storage systems, ours offers flexibility and scalability. Through innovative allocation mechanisms, we've crafted a robust solution for efficient data management without the constraints of traditional partitioning.