

CSCE 611

OPERATING SYSTEMS

MP – 7

FILE SYSTEM

STUDENT – RAM SANKAR

UIN: 433003775

The objective of this machine programming is to implement a simple file system. I have implemented the functions defined in the file\_system.H/C and file.H/C.

The file system is made of free blocks and inodes

### **Inode attributes**

- File\_id
- Block\_id
- File size
- Is\_free

A free block list and inode list are used for managing the lifecycle of a file .

The free block list is implemented as char array and the state of the block is marked as 'u' – used or 'f' -free depending on the operation performed by the filesystem.

Inode list is stored in 0<sup>th</sup> free block and free block list in the 1<sup>st</sup> free block

### **File system functions**

#### **FileSystem()**

Here the inode list and the free block list declared in the header file are created by memory allocation

#### **~FileSystem()**

The inode and the free list are written to the 0<sup>th</sup> and 1<sup>st</sup> block of the disk before unmounting the filesystem

#### **Mount()**

The disk object passed in the parameter is assigned to the disk of filesystem. The inode list is read from 0<sup>th</sup> block of the disk and the free block from 1<sup>st</sup> block of disk

#### **Format()**

Here all the inodes are marked free and other attributes such as file\_id , block\_id and file size are also set appropriately

#### **LookUpFile (file\_id)**

This is a linear search function which is used for finding the file id in the inode list. If found it will return the corresponding inode else it returns NULL. The LookUpFile function is used for creating and deleting the file

### CreateFile(file\_id)

This function calls the LookUpFile to see if there exists any file with same file\_id. If not it creates a new file by calling get\_free\_block() and get\_free\_inode() which returns index of the free block and the inode.

The inode is set as used and the attributes of the inode are set appropriately

### get\_free\_block()

Linear search on free block list which returns the index of the first found free block

### get\_free\_inode()

Linear search on inode list which returns the index of the first free inode

### DeleteFile(file\_id)

This function calls the LookUpFile to verify whether the given file id exists. If it exists, it returns the corresponding Inode. Using the attributes of the inode, we free the corresponding memory block and the set the file\_id of the inode to NULL. Then we clear the data present on the disk by writing NULL to the block no. of the inode

## **File functions**

File object needs the following attributes apart from the file system

- inode\_index
- current position
- file size
- block no

### File(file system, file\_id)

The file system finds the inode using the file id, then we find the block belonging to the file. We then copy the data of the block from the disk to block cache. This would come handy for reading and writing data to and from the disk

### ~ File()

The updated block cache and the inode list is written back to the disk.

### Read(n, buffer)

In this function we read n characters from the block cache starting from the current position and populate the data to the disk. But this function has a catch, if we reach the end of file we need to stop the reading. The following logic is used to check the condition

```
while( !EoF() || buf_pos < _n )
{
    //read from block cache and populate to buffer
}
```

#### Write(n, buffer)

Here we start writing data from the current pointer and we increase the size of the file if we reached eof by incrementing it as and when the cur position exceeds file size

#### Reset()

The current position of the file is set to 0

#### EoF()

This is a helper function which is used for checking whether we reached end of file while reading. We use file size attribute for comparison

### **OPTION – 1**

#### **file system to handle large files (64 KB)**

In the vanilla file system, each file was restricted to 1 block. We would require multiple blocks of memory for larger file system.

This can be done by storing a list of blocks allocated for the file in the inode object.

Instead of the block\_id attribute in the inode , we can have an linked list which would store the blocks of the file in sequential order

Struct Node

```
{
    Int block_id;
    Struct Node* next_block;
}
```

Following changes need to be made to the File system for handling larger files

#### createFile( file\_id)

This would be mostly similar to the simple file system but the initial free block would be stored as the head of the block list belonging to the file

### deleteFile(file id)

Instead of just freeing one block we would need to mark multiple blocks as free using the linked list of blocks of the file

Also, the disk write operation should be called multiple times ( = no. of blocks belonging to file) to clear the data present on the disk

Following changes would be made to the File

Here we would need additional attributes such as current block along with the current position to perform read and write operations

### Read(n, buffer)

The read operation should be altered as follows:

```
temp = head // block list head
```

```
While ( temp != NULL)
{
    cur_block = head;
    read(cur_block); // similar to vanilla file system;
    cur_block = cur_block -> next_block;
}
```

### Write(n, buffer)

The write operation is hard to implement. Here if we exceed the size of block, we need to do assign a new block to the block\_list of the file and write the data to it. The helper function get\_free\_block() would be used in this

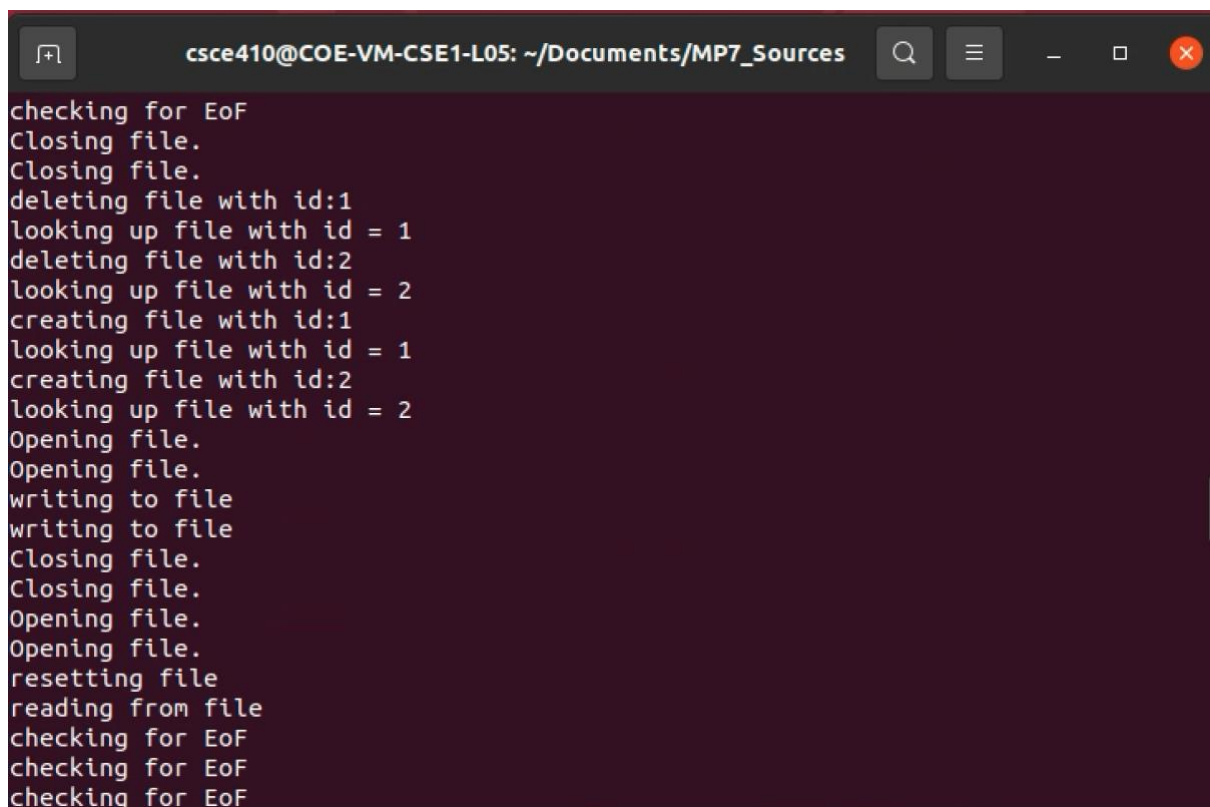
### EoF()

In the EoF we need to calculate global position of the file pointer. This can be done by multiplying the position of current position with number of blocks we have traversed. If the global position reaches the file size, then we say it is EOF

## OPTION -2

I have tried implementing the described detailed design. I was able to make changes successfully to the file system. However, I had trouble in making changes to the file operations, specifically write operation. So, I have reverted to simple file system

## RESULTS

A terminal window with a dark purple background and light green text. The window title bar shows the user 'csce410@COE-VM-CSE1-L05' and the directory '~/Documents/MP7\_Sources'. The terminal output consists of a series of file system operations: 'checking for EoF', 'Closing file.', 'deleting file with id:1', 'looking up file with id = 1', 'deleting file with id:2', 'looking up file with id = 2', 'creating file with id:1', 'looking up file with id = 1', 'creating file with id:2', 'looking up file with id = 2', 'Opening file.', 'writing to file', 'Closing file.', 'Opening file.', 'resetting file', 'reading from file', and 'checking for EoF'.

```
csce410@COE-VM-CSE1-L05: ~/Documents/MP7_Sources
checking for EoF
Closing file.
deleting file with id:1
looking up file with id = 1
deleting file with id:2
looking up file with id = 2
creating file with id:1
looking up file with id = 1
creating file with id:2
looking up file with id = 2
Opening file.
writing to file
Closing file.
Opening file.
resetting file
reading from file
checking for EoF
checking for EoF
checking for EoF
```

This screenshot displays the infinite process of creating, reading, writing and deleting the file