

# A2: Write Once?

Himanshu Rathi (hr393)

Shashwenth Muralidharan (sm2785)

## 1 Introduction

The File System is an important structure used by the operating system to manipulate and modify the data stored in the system. This document is a comprehensive report of our implementation of a user level file system that allows the user to create, read, write and perform various operations on the file. In this implementation we will use a 4MB address space acting as our main memory. The main structure of our main memory consists of a superblock that stores the data about the file system I.e metadata about the file system. For each file we create, a corresponding inode will be created to store information about the file. The following diagram shows the memory representation of our file system.

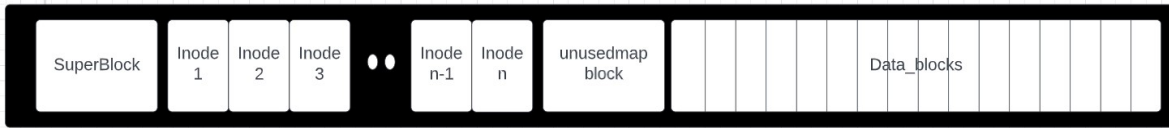


Figure 1: Structure Of Our File System

In this report we will extensively detail the methods and API implemented, the data structures used and the performance comparison in our library.

## 2 Methods

This section will provide all the necessary information regarding the functions we implemented in this library.

For simplicity, we have split this section into the following subsections based on their usage.

### 2.1 File System Operation

This section elucidates all the functions implemented for the purpose of file system operation, this includes:

1. `wo_mount()`
2. `wo_unmount()`
3. `wo_open()`
4. `wo_read()`
5. `wo_write()`
6. `wo_close()`

#### 2.1.1 `wo_mount()`

Before accessing any file systems we are required to mount them. Mounting them makes them available to the system and thus allows us to use them. Our `wo_mount()` function takes in two parameters namely filename and memory address. The 'filename' represents the disk I.e the 4MB address space that represents the file in the operating system. Also, during the mount process we will read the whole

file system from the disk into a buffer that acts as our main memory.

The first step in the mount function is to open the file and create a corresponding file descriptor to the file. There are possible conditions when we try to mount a file. The two cases are:

1. The first case is when the file system is empty that is this is the first mount() call. Now if the file is empty or uninitialized we will be creating the required structures for performing various operations in the file. These include, initializing a superblock, initializing I-nodes, data blocks and other structures that are deemed necessary for efficient performance.

2. The second case is when the necessary data structures are already initialized i.e this is not the first function call to mount() function. In this case, we directly move to the mounting operation.

The superblock is an important structure that is used to maintain important information about the file system. This includes the total number of blocks, the number of inodes, data blocks and also information about unused data blocks.

The inodes are used to record information about individual files. The metadata stored in the inodes are used to access the data blocks and gather the necessary information about them in order to access the files and perform actions on them.

In our implementation we have also used additional 4 KB space in the memory to store information about the current status of the data block. We will use '1' to represent that the corresponding data block is in use and '0' to represent the inverse.

Finally, after all the initialization procedures are completed the file system is read into the buffer and could be used for various purposes required by the user.

The following flow chart illustrates the working of mount operation using our implementation.

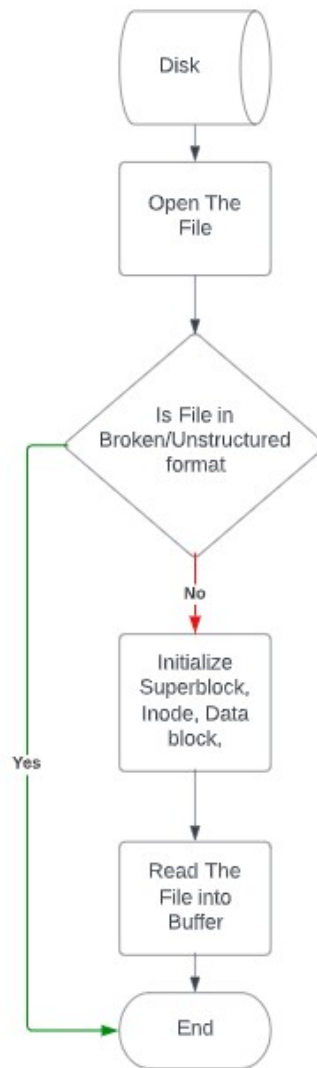


Figure 2: Mount operation Flow Chart Representation

### 2.1.2 wo\_unmount()

Once all the required operations are carried out to the file in the buffer we need to detach it from the memory and update them back into the disk. The unmount function is used to perform that operation. This function takes in the buffer's memory address as an parameter. This address represents the location from which the data should be read and written back into the disk. Once the unmount operation is done, the file descriptor is closed.

### 2.1.3 wo\_open()

This function opens the file specified by the user and initializes the corresponding access permission to the file. This function takes in filename and flags ( This value is used to represent the access conditions) as parameters. The various file access permissions used are 'read only', 'write only' and 'read and write' values. These access parameters are represented as integer values.

Initially we will check for possible errors including:

1. If the file exists or not. If it doesn't we will print the error no 2.
2. If the user has entered a valid permission flag value to access the file etc. If the permission is denied we will print error number 13.

3. If an invalid argument is passed we will print error number 22.

If either of these conditions fail to meet then '-1' will be returned and the error message will be printed.

Now, we will traverse through the inodes with the filename for our open procedure. This is done by the FindFreeInode(). This function returns the address of the inode in the buffer. Once the inode of the corresponding file is found, the inode is updated and the file descriptor is returned to the user.

The various values updated in the inode are:

1. Access permission for the file based on the user requirement.
2. Status of the file is set to 'O' representing open.
3. If the file is not already in use the value of inuse is set.

Once all the values are updated the file descriptor value is returned back to the user. The user can use the file descriptor to perform various operations in the file.

The following flow chart illustrates the working of unmount operation using our implementation.

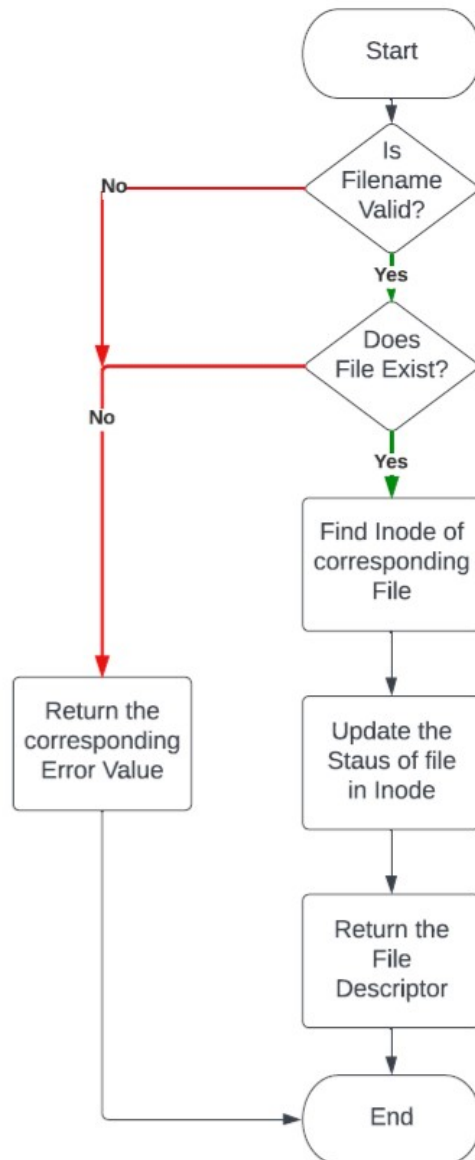


Figure 3: Open Operation flowchart representation

#### 2.1.4 wo\_create()

The wo\_create() is an special version of wo\_open() that allows the user to open files that are not created yet. In other words, if a file doesn't exist in the disk or if a file is not created yet, then the user can call the wo\_create() function which creates a new file with the specified name and parameters. Furthermore, once the file is created we will be opening the file.

Furthermore, we need to check for certain conditions to make sure that the create operation is valid. These conditions include:

- 1.If the file exists or not. If it does exist already we will print the error no 2.
2. If an invalid argument is passed we will print error number 22.
3. If the inode is full and we are unable to create a new file then we will output error value 23.

The procedure for creating and opening the file is detailed below.

Initially, a new file will be created using the FindFreeInode() function. Initially, this function will search for a free Inode from the list of inodes. If it is not found then the error value will be thrown. Now, once the inode is found we will create the file and assign the necessary parameters including file descriptor, status etc to the new file.

Now, for the opening process the inode is updated with certain parameters. The various values updated in the inode are:

1. Access permission for the file based on the user requirement.
2. Status of the file is set to 'O' representing open.
3. The value of inuse parameter is set.

Once all the values are updated the file descriptor value is returned back to the user. The user can use the file descriptor to perform various operations in the file.

The following flow chart illustrates the working of unmount operation using our implementation.

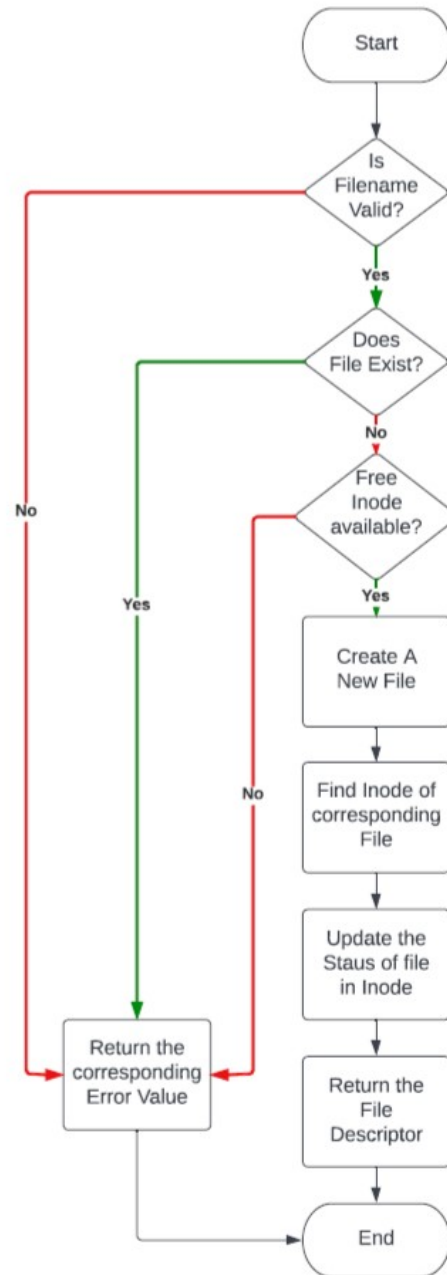


Figure 4: Create Operation flowchart representation

### 2.1.5 wo\_close()

The purpose of the close operation is to let the system know that the file is not being currently used or it's no longer has usage by the user. In our implementation we will update the value of status to 'C' representing that the file is closed in the corresponding inode of the specified file descriptor. However, if the file descriptor value is invalid or the file is already is closed then the function throws an error specifying the same.

### 2.1.6 wo\_write()

This function is used to write data into the file specified by the user. This is done by copying the bytes from the buffer into the data blocks. During the write operation we need to check if the write should be performed on a new file or not. This check gives us with the following cases:

1. If the condition is true i.e we need to write in a new file then we find a new data block and start our writing process.
2. If we are required to write in a file that already has data in it then we traverse through the file until EOF and write the bytes.

Furthermore, we need to check for certain conditions to make sure that the write operation is valid. These conditions include:

1. If the file exists or not. If it doesn't we will print the error no 2.
2. User has the permission to write into the file, if not an access violation error 13 will be thrown to represent read only file.
3. If the file descriptor specified by the user is valid I.e it is within the specified range. If not then error number 9 will be printed.
4. If the file is opened or not, because write operation is invalid in a mounted but closed file if not we will return 1.
5. If an invalid argument is passed we will print error number 22.

The write operation procedure is specified below. Firstly, we need to find an empty data block to write into. If there are no empty datablocks then we will return an error value. Now, we will access the inode of the specified file to check if the start data block value is '-1'. If this is true then the data block we write into will be the start data block and we are not required to traverse because this is the first write operation on the file. Here will will copy the bytes from the buffer into the data block.

During copy, the number of bytes is determined by calculating the minimum of either the bytes to be written or the space available in the data block. This ensures that we do not exceed the size of data block (Overflow Error) during write and later cause an segmentation fault for an invalid access. For example, if the number of bytes to be written is 100bytes and the remaining space in the data block is 50bytes, we write the min I.e 50bytes in the first iteration and the remaining in the upcoming iterations. Also we will continue the process of finding a free data block, writing into the block and updating the inode until all the data is written. However, if there are no free blocks the data written in the previous iterations will be retained and an error will be thrown.

Now, if the file already has data written in it, we will traverse through the data blocks until we find the last block. Now, we will do the same steps explained above to write the bytes into the file until either the input buffer is empty or the memory is full.

The following flow chart illustrates the working of write operation using our implementation.

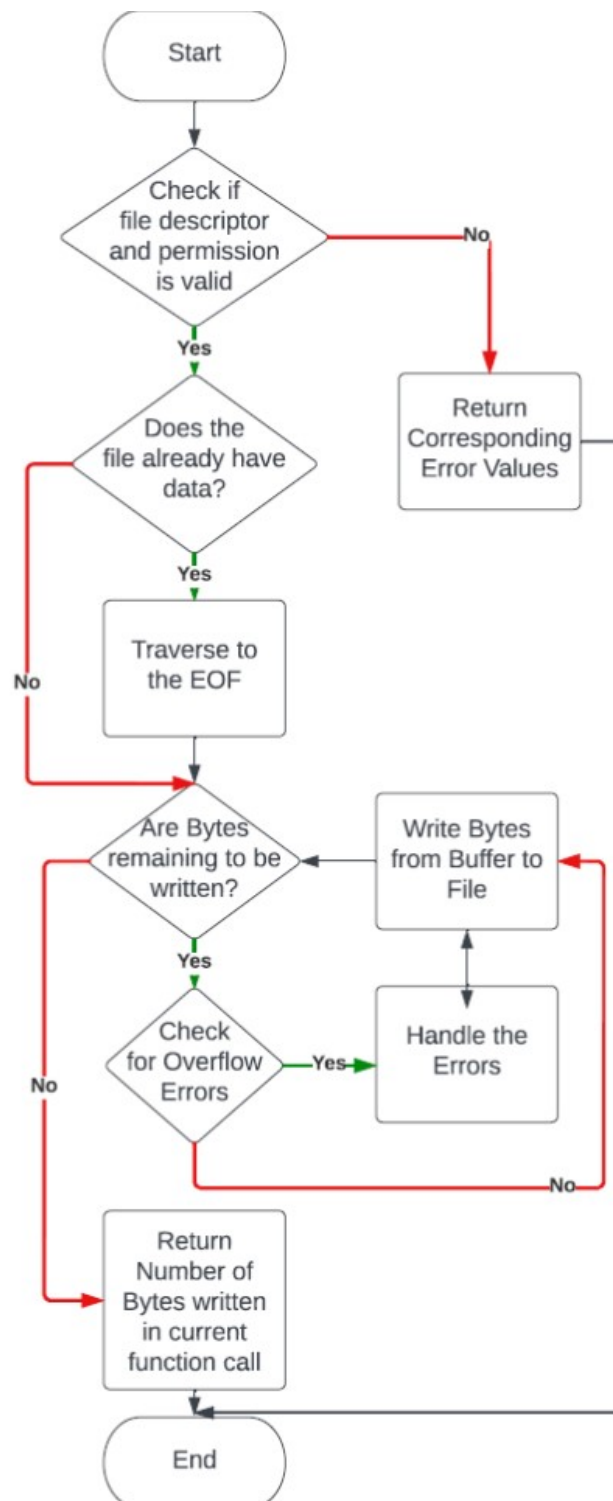


Figure 5: Write operation Flow Chart Representation

### 2.1.7 wo\_read()

The purpose of this function is to read from the user specified file that we have mounted already.

This function takes in the file descriptor, number of bytes to be read and the buffer memory space address as parameters. The primary operation here is to read the bytes from the file into the buffer



memory. If the read operation is successful this function returns '0' and if any error occurs it will return the corresponding error value.

Before reading from a file there are certain conditions that should be satisfied. They include:

1. The file descriptor not be invalid I.e the file descriptor value should be within the valid range if it is invalid error no: 9 will be printed.
2. The user must open the file with either read only or read and write permission. If the file has write only permission then the read operation becomes invalid. Then we will return error number 13.
3. The file must be open I.e the status value must be 'O'. Else error val 2 will be printed representing that a write was tried on a closed file.
4. The size of the file must be greater than 0 because we cannot read from an empty file.
5. If an invalid argument is passed we will print error number 22.

Initially, we will gather the necessary information from the inode of the file including the number of data blocks, size of the file, starting address of the data block etc. The procedure involves iteratively copying bytes from the file to the buffer until the total number of bytes specified by the user is copied. Also, in the process if the current data block is read and we still have bytes remaining to be read, we traverse through the data blocks to the next data block. The above process is continued until the end of file or until the number of bytes to be read is exhausted.

Furthermore, the number of bytes we read is equal to the minimum value between the remaining bytes to be read and data block size. This ensures that we do not underflow (Read less bytes than available) or overflow (Read more bytes than available) the read operation from the file and hence causing any segmentation fault.

Finally, once all the bytes are read without any errors this function returns '0' representing that the read operation was successful.

The following flow chart illustrates the working of read operation using our implementation.

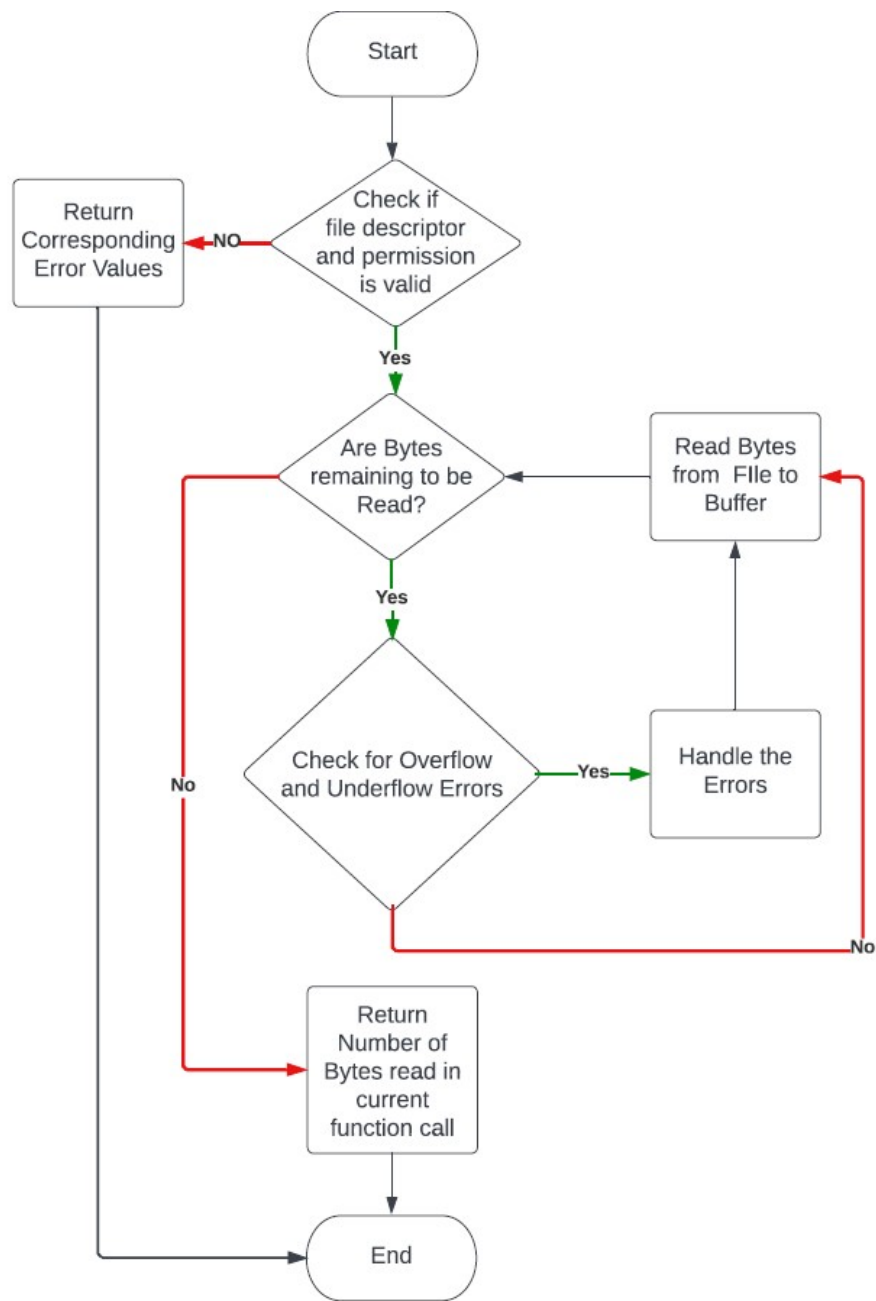


Figure 6: Read Operation Flow Chart Representation

## 2.2 Additional API

This section elucidates all the additional functions implemented for the execution of our file system, this includes:

1. min()
2. Charconcat()
3. FindFreeInode()
4. FindFreeDataBlock()
5. CheckIfFileExists()

### **2.2.1 min()**

The min function is used to find the minimum of two integers and return the value. This function is mainly used in the read() and write() operations in order to compute the number of bytes to be read from and written into a file correspondingly.

### **2.2.2 Charconcat()**

This function is used during the read operation to copy the bytes from the file into the buffer memory. This function iteratively assigns the current value of the source to destination until an EOF or end is found. Also, it appends and EOF to the destination file to specify that the copy operation is completed.

### **2.2.3 FindFreeInode()**

As the name suggests, the FindFreeInode() is used to find a free inode from the list of inodes and return the starting address of this inode to the user. This function is used whenever the user wants to create a new file during write or open a file in creator mode. The process involved in this function is to find a free inode iteratively by traversing through the inodes and finding a free block. This is done by checking the 'inuse' value in the inode structure which stores the information about the inodes usage status. Also, if the inode already exists for the file, we will return the starting address of the corresponding inode to the user. If no free inodes are found then we will return NULL specifying the same.

### **2.2.4 FindFreeDataBlock()**

The purpose of this function is to find a free data block on request from the buffer memory space. This data block will be used to write files into. This function makes use of a special address space called the UNUSEDMAP\_BLOCKS. This 4KB region is used to store the status of the corresponding data blocks I.e the Kth value in this region will represent the status of the Kth data block in the memory.

The status is stored as a binary representation where '0' represents that the corresponding data block is empty and vice versa. In order to find a free data block we will traverse this unusedmap\_block and find the blocks with status '0' and return the value of K. If no free data blocks are available '-1' will be returned representing the same.

### **2.2.5 CheckIfFileExists()**

This function is used to check if a file already exists in the main memory I.e buffer. This function is used in the open() function, especially in creator mode, to check if the file already exists. This function works by traversing all the inodes and checking if it is inuse. If the inuse value is set, then we will check if the file exists by comparing the filename.

This function returns '0' if the file exists and it is in open status. If it exists but the status is closed then the return value is '1'. Finally, it will return '-1' if the file does not exist.

## 3 Data Structures

This section explains the important data structures involved in our implementation

### 3.1 Super Block

```
typedef struct superblock
{
    char uid[10]; // Unique identifier for the file system
    int total_blocks; // Number of blocks in the entire file system
    int total_inodes; // Number of inodes in the file system
    int total_datablocks; // Number of data blocks in the file system
    int total_files; // Total files in the file system
} superblock;
```

### 3.2 Inode

```
typedef struct inode
{
    char inuse; // Represents if file is inuse or not
    char status; // Represents if file is open or closed
    int total_data_blocks; // Total data blocks used by the file
    short start_data_block; // Address of the starting data block
    char filename[MAX_FILE_NAME_SIZE]; // Name of the file with max 50
    unsigned short fd; // The File descriptor of the file
    int file_size; //Size of the file in bytes
    unsigned short curr_offset; // Number of bytes already read from the file
    short permission; // Access permission of the file
} inode;
```

### 3.3 Data Block

```
typedef struct datablock
{
    short nextblock; // Address of the next data block
    char data[1020]; // Array storing the data for the file
} datablock;
```

### 3.4 Memory Buffer

char \*Buffer // Buffer memory used to store the mounted file

FS\_SIZE 4\*1024\*1024 // This value represents the size of the file system

### 3.5 Data Block

BLOCK\_DATA\_SIZE 1020 // Size of each data block in the Buffer

### 3.6 Unused Map Block

UNUSEDMAP\_BLOCKS 4 // Additional 4KB used to store status of data block

## 4 Benchmarks

We have tested our implementation for two benchmarks, they are:

1. Maximum Files Allocated: This benchmark is used to test the maximum number of files that can be allocated in our file system. These files have their individual Inodes that contains their respective metadata. In our implementation, we successfully allocated 60 files in the memory. This shows that, our implementation has successfully crossed the threshold of fifty files to be created in the file system.

2. Maximum Size of File: This benchmark is used to check the maximum single file that can be allocated in the file system. In our implementation, we were able to successfully allocate a single file of size 3.92 MB. From this we can infer that out of the 4MB memory, only 0.8MB is used to store metadata. This proves the efficiency in storing the file by our implementation.

## 5 Running The File

In order to implement the file, execute the makefile using the following lines of code:

NOTE: Before testing the file write,  
#include "writeonceFS.c"  
to your testing code for executing our file system implementation.

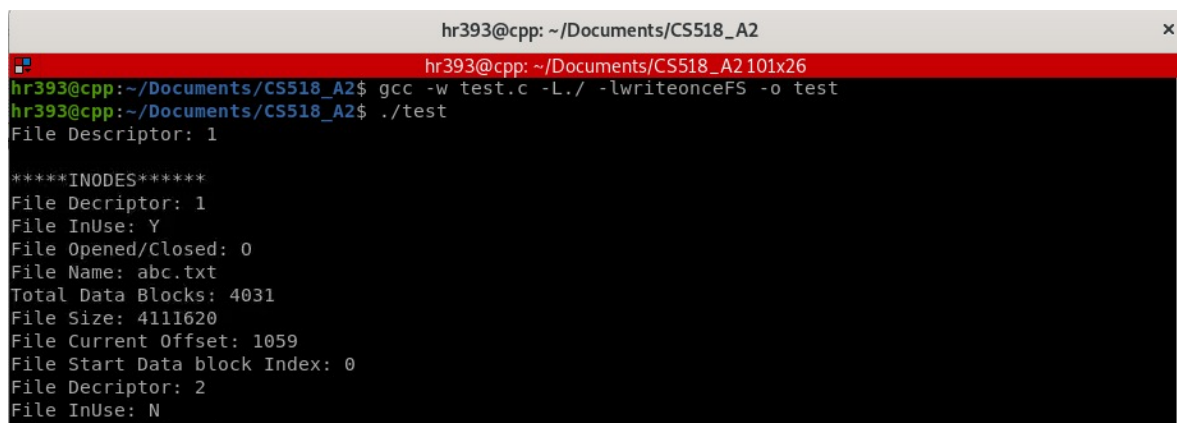
For instance, let the test file name be "test.c". To compile and run the file system implementation follow the following steps:

```
: make clean  
: make all  
: gcc -w test.c -L./ -lwriteonceFS -o test  
: ./test
```

NOTE: Make sure file to be tested is in the same directory as writeonceFS.c and MakeFile

## 6 Sample Output

This section illustrates a sample execution of our file system's implementation. Here we will be creating once file of size 4111620 bytes (approximately 3.92 MB). The output below will show the status of the file in use, the file descriptor, the number of data blocks occupied by the file, size of the file etc.



```
hr393@cpp: ~/Documents/CS518_A2  
hr393@cpp: ~/Documents/CS518_A2 101x26  
hr393@cpp:~/Documents/CS518_A2$ gcc -w test.c -L./ -lwriteonceFS -o test  
hr393@cpp:~/Documents/CS518_A2$ ./test  
File Descriptor: 1  
  
*****INODES*****  
File Descriptor: 1  
File InUse: Y  
File Opened/Closed: 0  
File Name: abc.txt  
Total Data Blocks: 4031  
File Size: 4111620  
File Current Offset: 1059  
File Start Data block Index: 0  
File Descriptor: 2  
File InUse: N
```

Figure 7: Sample Output

## 7 Conclusion

Based on the above benchmarks we can conclude that our file system implementation performs both effectively and efficiently. We were able to allocate a file of maximum size 3.92MB as a single file and we were able to allocate 60 individual files. Since we have included error checking and certain error handling mechanisms our implementation provides a more user accessible library.