# CS518 - A2: Write Once

Group Members:

Jay Patil (jsp255)

Aishwarya Harpale (ach149)

## OVERVIEW

A **file system** is a way in which files are named and where they are placed logically for storage and retrieval. Without a file system, stored information wouldn't be isolated into individual files and would be difficult to identify and retrieve. In this assignment, we have implemented a logical Write Once File system, wherein it is possible to write in the data region only once when the disk is unmounted. A file system consists of various data structures that are used to maintain the metadata of a file system, such as Superblock, Inodes, etc. **Superblock** can be looked at as a higher level of metadata maintenance of a file system. It generally stores information such as the number of blocks, number of inodes per block, file system type, etc. An **Inode** exists in a file system that stores the metadata about the files in a file system, such as ownership information, access mode of the file, file type, etc.

Note: This code was tested on ilab machine Butter in the Meltdown Lab hosted on butter.cs.rutgers.edu.

## API

1. wo_mount(char* filename, void* address):

   This function attempts to read the entire disk. It returns 0 on success after reading and returns a negative number if there is an error in reading the file. This method also verifies if the disk is correctly formatted. It is also responsible for building initial structures if the disk is blank, and if the initialized format of the disk is corrupted, then it removes the corrupted structure and empties the disk. In our implementation, when this method is invoked for the first time, the superblock is initialized with 50 inodes, and the number of data blocks to 4045. The inodes are also initialized to their default values. Otherwise, the file is read from the desired location passed as the parameter 'address' to the function.

2. int wo_unmount( void* address):

   This function, when invoked, attempts to write out the whole disk. Similar to the wo_mount() method, it returns 0 on success and a negative number on the error, the number representing the type of error. In our implementation, this method logically unmounts the disk by closing all the file descriptors and also marking all the inodes as close.

3. int wo_open(char *filename, int flags, ...)

   This method is responsible for creating a file in the file system and hence also creates a file entry for the requested file. The flags that are passed to this method are access requirement flags, viz. ReadOnly, WriteOnly, and ReadAndWrite.  Another dynamic parameter this method accepts is 'mode', which represents if the user is in file create mode. If the user is in create mode and if the requested file exists, this method throws an error along with an error number. The method creates the file and returns its file descriptor if the request is valid.

4. int wo_read( int fd,  void* buffer, int bytes)

   This function is responsible for reading from the file. When it is invoked, it checks if the file descriptor is valid. If it is not, it sets the error number and returns the error. Otherwise,  it starts reading from the location given for the file. If the file is empty or does not have read permissions, or the location provided is beyond the end of the file, appropriate errors are raised.

5. int wo_write( int fd,  void* buffer, int bytes)

   On invocation, this method checks if the file descriptor is valid. If not, it sets the error number and returns an error. If the file descriptor is valid, the file is written from the location provided as input. The method also checks whether the file has write permission or if a free data block is available or not. If not, it returns the error and sets the error number appropriately.

6. int wo_close( int fd)

   On invocation, this method checks if the passed file descriptor is valid; if valid, it marks the file as closed. If the file descriptor is not valid, it returns an error and sets the error number.

7. short getFreeDataBlockIndex()

   This is an auxiliary method used to find the next free data block index in the data block region.

8. short getFreeInodeIndex()

   This is an auxiliary function used to get the next free inode.

9. char* str_concat(char *dest, char *src, size_t n)

   This method is an auxiliary method that helps in the concatenation of two strings and appends a break delimiter toward the end of the concatenated string.

## STRUCTURES

### Disk Structure

| Superblock (20 bytes) | Inode Bitmap (70 bytes) | Inodes (3150 bytes) | Data Bitmap (4096 bytes) | Data (4088 kilobytes) |
|---|---|---|---|---|

```
1. typedef struct

{

  int data_start;

  int num_inodes;

  int num_data_block;

  int magic_num;

} super_block;
```

The above structure represents the superblock of the file system of our implementation.

‘data_start’ holds the start of the data region of the file system.

‘no_inodes’ represents the number of inodes that are available in the file system.

‘no_data_block’ tells us the number of data blocks the file system has.

‘magic_number’ is used to represent a unique number to identify the correctness of the disk.

```
2. typedef struct

{

    char filename[20];
    int id;
    int file_size;
    int permission;
    int read_seek;
    short data_block_start;
    bool is_open;

} inode;
```

The above structure represents the inode data structure of the file system in our implementation.

‘filename’ holds the name of the file with which the inode is associated. Filename is limited to 20 characters.

‘id’ is the inode ID to identify an inode uniquely.

‘file_size’ tells the size of the file that is associated with the current inode.

‘permission’ represents what permission the file has.

‘read_seek’ holds the read pointer from when the last read was done.

‘data_block_start’ holds the value of the start of the data region for that inode. ‘is_open’ is a flag that tells whether the file is open or closed.

```
3. typedef struct

{

    char data[1020];
    short next;

} data_block;
```

The above structure helps to access the data from the data blocks. ‘Data’ represents the actual data or memory space in the file system. ‘next’ is the pointer to the next data block in the file system.

## OTHER DATA STRUCTURES

1. Data Bitmap:

   In order to track the allocation of data blocks, we have used a data bitmap. The value at the particular index tells if that data block is free or not. If the value is ‘n,’ the corresponding data block is not free.

2. Inode Bitmap:

   In order to track the allocation of inodes, we have used a inode bitmap. The value at the particular index tells if that inode block is free or not. If the value is ‘n’ the corresponding inode is not free.

## HOW TO RUN

To make the library, run “make all”..

To clean all the created files, run “make clean”.

## ANALYSIS

The implementation that we have done as a part of this assignment helped us understand the basic functioning of a file system. Even though this implementation is logical, it gives a pretty fair idea of how the basic workflow of an actual file system is in our Operating System. During our implementation, we faced challenges of effective validation of the mount, unmount, read, write, open and close operations. Each case is meticulously handled so as to give the user intuition of interacting with a real-time file system.  The importance of data structures used in the file system, such as superblock, inode, data bitmap, and an inode bitmap, was realized.