**Operating System Project Report**

**[Project Title: Unix File System]**

Muhammad Ahmed (18K-0256) – 2A

**Project Description:**

UNIX File system includes all the necessary functions needed in file system of any operating system.

int make\_fs();

int open\_file(char\* froute,int mode);

int lseek\_file(int fd, int offset, int whence);

int read\_file(int fd, char\* buff, int count);

int write\_file(int fd, char\* buff, int count);

int close\_file(int fd);

int cd\_Dir(char\* froute);

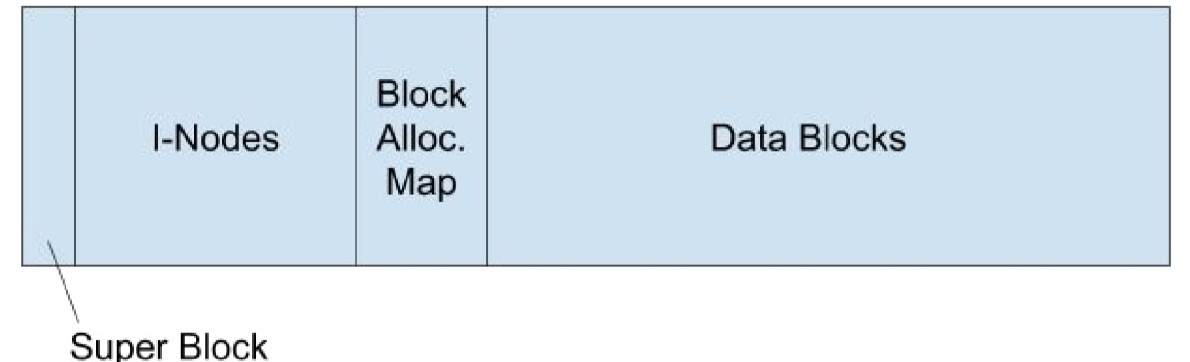
int make\_Dir(char\* dirname);

int rm\_Dir(char\* dirname);

We have used a regular file to simulate as our disk. Our file system is implemented with 4KB block size, bit vector for tracking free blocks, and inode structure with 12 direct and a single level indirect block.

**Overall File System Layout**

The file system I implemented is divided into four logical sections. In order, these are the super block, i­nodes, block allocation map, and data blocks. These sections are contiguous on disk and each serve a particular function. The details of each section are outlined below.

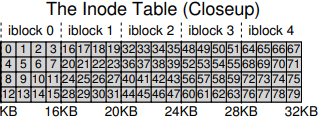


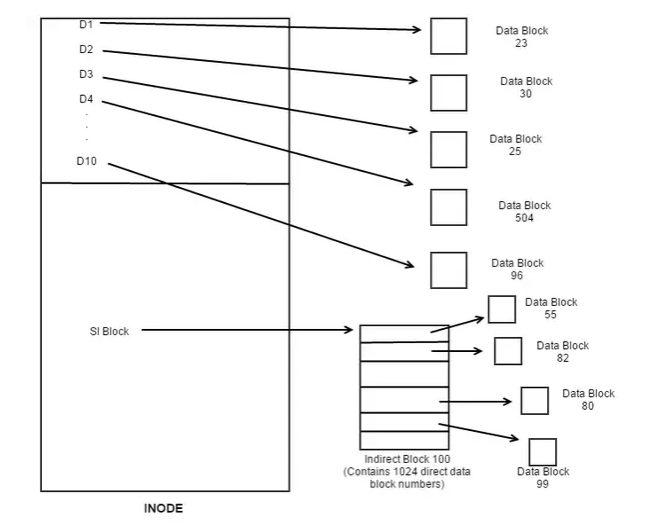
**Super Block**

The super block contains various persistent information about the file system, such as the overall file system size, the starting block of each section, and the number of blocks each section contains, all of which are set when the system is initialized. The super block also contains a “magic number” at a particular address, which indicates to processes attempting to access the file system that the disk has been formatted for use. As the name suggests, the super block takes up exactly one block of disk space, and it is located at block index 0. In my file system implementation, the super block struct also remains in memory in order for persistent system info to be accessible to file system functions without needing to make many disk I/O operations.

**I­Nodes**

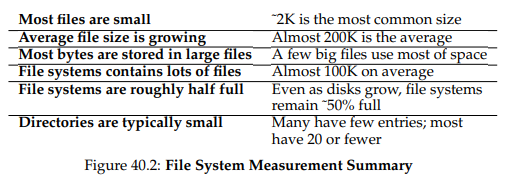
Each distinct file or directory in the file system has a corresponding i­node that defines various metadata about that file. In particular, a file’s i­node tracks its size, type (file or directory), the number of extant links and open file descriptors to it, and addresses of the data blocks on disk where data for the file are stored.



An i­node and the data blocks associated with it will not be freed until there are no longer any links or file descriptors to that inode. ****

In my file system implementation, an i­node can contain up to 12 direct links to data blocks where file data are stored, for a maximum file size of 4096 bytes. A single level Indirect Block is

also used to support large files. Thus, with a small number of direct pointers (12 is a typical number), an inode can directly point to 48 KB of data, needing one (or more) indirect blocks for larger files.



**Bit Vector:**

The block allocation map consists of a set of boolean flags that indicate whether each data block is in use by some i­node. Each data block is represented in the map by a single 0 or 1 bit at the corresponding index. A data block is allocated and set aside for an i­node simply by setting its usage bit to 1 in the block allocation map and adding its index to the i­node’s list of data blocks. Freeing a block works similarly, by setting the usage bit to 0 in the block allocation and removing it from the i­node struct to which it was assigned. When all bits in the block allocation map are set to 1, there are no free blocks remaining in the file system.

**For example:**

*void mark\_Block\_Used(int dBlock){*

*int byte = dBlock / 8;*

*int bit = dBlock % 8;*

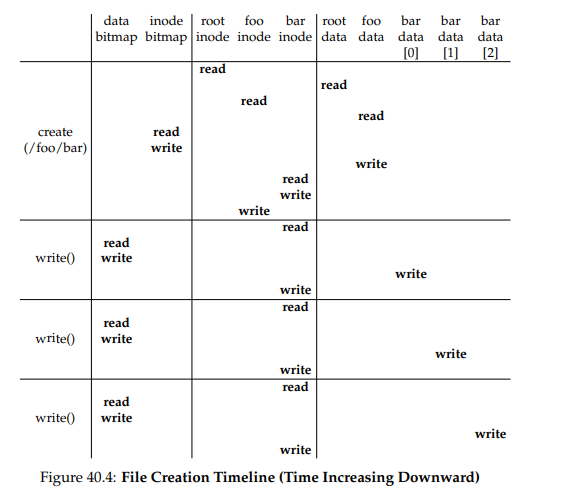
*bitmap[byte] = bitmapBuffer[byte] | power\_2(bit);*

*}*

**Reading and Writing:**

To read and write, a file must be opened first. Opening a file is a time consuming process and increases as the path becomes longer since more directories has to be searched. For larger directory, it gets even time consuming since more blocks have to be read. Finding the file inode by traversing the directories for every read and write call is a bad idea, so to save ourselves, we open a file and store its inode number in a **file descriptor table**.

Of course, it would still take a lot of reads and writes to perform a single open, read, or write call. To remedy this, Caching and Buffering is used. Our simple Unix file system doesn’t cover it. For example, the first open may generate a lot of I/O traffic to read in directory inode and data, but subsequent file opens of that same file (or files in the same directory) will mostly hit in the cache and thus no I/O is needed.



**Other Helper Functions**

Void parsefilename(char\* filepath)

void set\_Inode(Inode\* inode, short type);

void init\_superBlock(Superblock\* superBlock);

int make\_fs();

int find\_Free\_Inode(short type);

int add\_Dir\_Entry(int dir,int newInode, char\* fname);

int find\_Entry(char filepath[][30],int f\_len,int\* lastEntryParent, int\* lastEntry);

int rm\_Dir\_Entry(int dir,int entryPos);

int find\_Free\_Block();

int create\_Bitmap(int bsize);

int read\_Bitmap\_Block(int idx);

void mark\_Block\_free(int dBlock);

void mark\_Block\_Used(int dBlock);

int writeBitmap();

int reserve\_Block();

int fd\_open(int inode,int mode);

void fd\_close(int fd);

int initDisk(char\* name);

int openDisk(char\* name);

int closeDisk(char\* name);

int isBlockValid(int block);

int writeBlock(int blockNumber, char\* buffer);

int readBlock(int blockNumber,char\* buffer);

How to run:

* Compile if not already with gcc disk.c fs.c –o main and other header files
* Run with ./main
* Enter r to make disk or reset it for the first time.