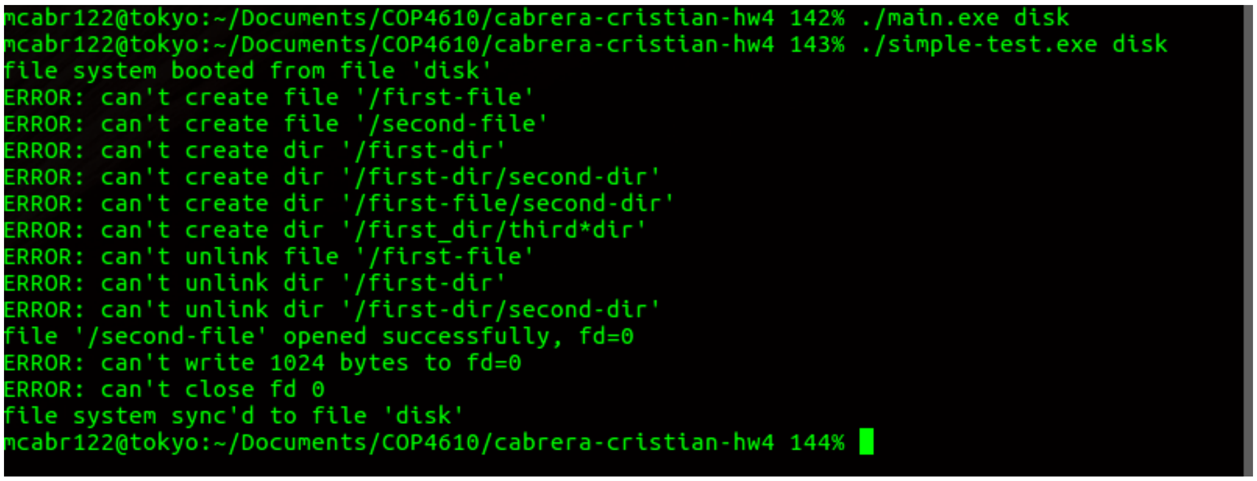
REPORT

Team Members

Marcial Cabrera – mcabr122@fiu.edu
Cristian Cepeda – ccepe014@fiu.edu

Under the Section Notes of this file is where we show the notes we were taking as we attempted to do our homework.

The picture below is the output of what we got currently when executing the files.



```
mcabr122@tokyo:~/Documents/COP4610/cabrera-cristian-hw4 142% ./main.exe disk
mcabr122@tokyo:~/Documents/COP4610/cabrera-cristian-hw4 143% ./simple-test.exe disk
file system booted from file 'disk'
ERROR: can't create file '/first-file'
ERROR: can't create file '/second-file'
ERROR: can't create dir '/first-dir'
ERROR: can't create dir '/first-dir/second-dir'
ERROR: can't create dir '/first-file/second-dir'
ERROR: can't create dir '/first_dir/third*dir'
ERROR: can't unlink file '/first-file'
ERROR: can't unlink dir '/first-dir'
ERROR: can't unlink dir '/first-dir/second-dir'
file '/second-file' opened successfully, fd=0
ERROR: can't write 1024 bytes to fd=0
ERROR: can't close fd 0
file system sync'd to file 'disk'
mcabr122@tokyo:~/Documents/COP4610/cabrera-cristian-hw4 144% 
```

NOTES

Execution of main.c
FS_Boot is first called inside of main.c
 -> and the disk is intitalize
 -> then its filename is copied to a variable and the file is then opened and set to read
 -> buffer is created and sector size is cleared and set to 0
 -> if its a new file then open and then this happens
    -> buf is then set to OS_Magic number
    -> Disk_Wirte is called and buffer is written to sector by calling memcpy
    -> use bitmap_init to
 -> else file was not null and so we need to check a few things
    -> check the size of file equals
    -> see if the magic number is there

FS_Sync() is the next thing called in main.c
 -> If any of the methods within Disk_Save are true true then it should fail to save disk to file.

-> else then the file bs_filename was successfully saved to the disk
----
Execution of simple-test.c
Same as FS_Boot inside of Main.c
-> File_Create() is then called and based on the type passed to create_file_or_directory in the return
   value then we will get an error or sucess.
-> create_file_or_directory 0 is for a file and 1 is for a directory

----
// Understanding LibDisk.c

```
   Disk_Init()
   -> disk is of type sector_t and on in line 31 they are allocating the
   memory necessary.
   -> return -1 if someting went wrong and return 0 if memory allocation went
   right.

   Disk_Save()
        fwrite(disk, sizeof(sector_t), TOTAL_SECTORS, diskFile)
   size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
      -> ptr − This is the pointer to the array of elements to be written.
      -> size − This is the size in bytes of each element to be written.
      -> nmemb − This is the number of elements, each one with a size of size bytes.
      -> stream − This is the pointer to a FILE object that specifies an output stream.
   -> disk is the global variable of our memory
```

// Functions To Define Inside of LibFS.c:

```
   // initialize a bitmap with 'num' sectors starting from 'start'
   // sector; all bits should be set to zero except that the first
   // 'nbits' number of bits are set to one
   static void bitmap_init(int start, int num, int nbits)

   // set the first unused bit from a bitmap of 'nbits' bits (flip the
   // first zero appeared in the bitmap to one) and return its location;
   // return -1 if the bitmap is already full (no more zeros)
   static int bitmap_first_unused(int start, int num, int nbits)

   // reset the i-th bit of a bitmap with 'num' sectors starting from
   // 'start' sector; return 0 if successful, -1 otherwise
   static int bitmap_reset(int start, int num, int ibit)

   // return 1 if the file name is illegal; otherwise, return 0; legal
   // characters for a file name include letters (case sensitive),
   // numbers, dots, dashes, and underscores; and a legal file name
   // should not be more than MAX_NAME-1 in length
   static int illegal_filename(char* name)

   // remove the child from parent; the function is called by both
   // File_Unlink() and Dir_Unlink(); the function returns 0 if success,
   // -1 if general error, -2 if directory not empty, -3 if wrong type
   int remove_inode(int type, int parent_inode, int child_inode)

   int File_Unlink(char* file)

   int File_Read(int fd, void* buffer, int size)

   int File_Write(int fd, void* buffer, int size)

   int File_Seek(int fd, int offset)

   int Dir_Unlink(char* path)

   int Dir_Size(char* path)

   int Dir_Read(char* path, void* buffer, int size)
```

```
|--------------------------------------------------------------------------|
  // Notes from PDF Provided
|--------------------------------------------------------------------------|
 ON-DISK DATA STRUCTURES
  // First part
  Record some generic information inside the disk
  -> Make this generic information be the very first block(SECTOR)
  --> SUPERBLOCK
     -> generic information about file system
     -> This generic information will be a (MAGIC NUMBER)
  --> Write this MAGIC NUMBER inside the SUPERBLOCK
  --> If we boot up the same file system check to see if we have our same
     MAGIC NUMBER
  --> Assume file system is corrupt if the MAGIC NUMBER is not found.
|--------------------------------------------------------------------------|
  // Second part
  Keep track of all files and directories on the disk
  -> Each file or directory points/corresponds to an inode
  -> Inode is a data structure (file size, type, etc)
  --> Inodes are stored consecutively; Can be refered to them by an index
  --> Using a bitmap to track which inodes have been allocated
```

-> Maximum of 1000 files/directories we will only need 1000 bits for the
  bitmap

|---|

// Third Part

Content of a file/directory is stored in a data block

-> Each data block is assumed be be same size as disk sector

--> bitmap needs to be used to track which sectors of disk have been
  allocated

|---|

// Fourth Part

|---|

// Fifth part

|---|

BOOTING UP

DISK PERSISTENCE

DIRECTORIES AND FILES

OPEN FILE TABLE

CURRENT READ/WRITE LOCATION OF AN OPEN FILE

MISCELLANEOUS NOTES