Project Report: File System Management

COP-5614 Operating System

Group:

Md Shahadat Iqbal - 4887079
Md Abdullah Al Mamun - 6144422
Vitalii Stebliankin - 6102408

Professor: Liting Hu
Submission date: December 2, 2019

1 Introduction

The project focus on developing a file management system. The main objectives of this project are:

- Gain a details understanding on how does file system work, specifically the directory hierarchy and storage management.
- Gain an understanding on some of the performance issues file systems must deal with.

In this project, we have built the user-level library, *libFS*, that can simulate some of the functions of a file management system.

1.1 Contribution

We divided the implementation into three parts and each group member is assigned to each part as follows.

- Bitmap and miscellaneous by Md Abdullah Al Mamun
- Read/Write of file or directory by Md Shahadat Iqbal
- Remove file/directory/inode by Vitalii Stebliankin

Assigned member has full contribution for designing and developing related functions. Code developing, checking possible test cases, writing a report, and preparing the presentation were performed under the active collaboration and productive discussions. Even though each of us was responsible for its part, we helped each other with the architecture and code design.

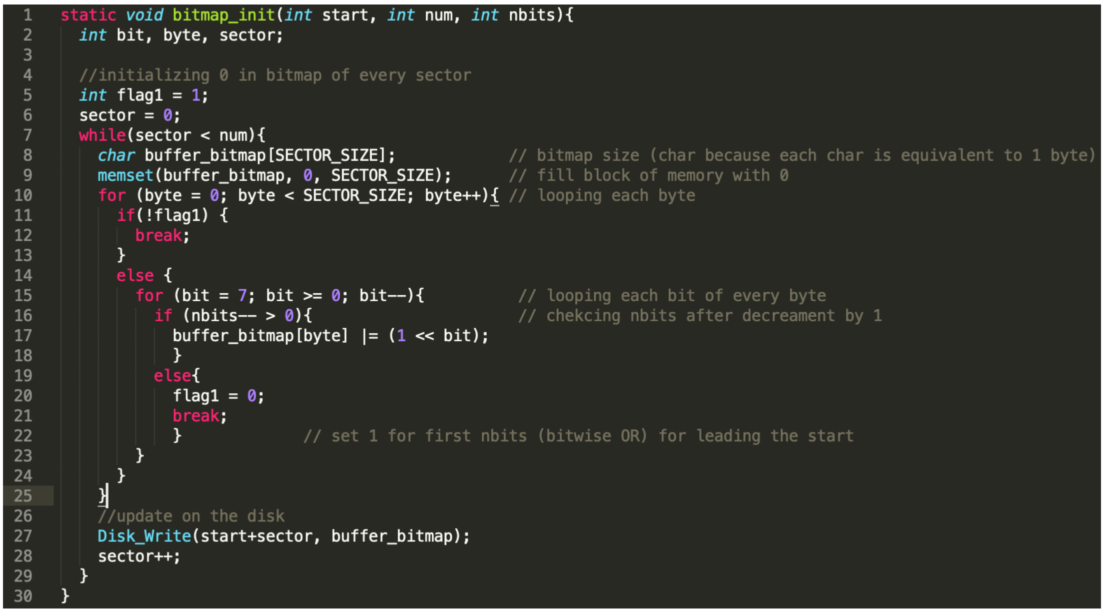
2 Implementation

2.1 Bitmap

Memory space in the disk is limited. Thus, managing free space is crucial part of any file system. File management includes keeping track of unused disk blocks, allocating space for newly created files and reuse the released space when a file is removed. For handling such operation, Bitmap or bit vector is used that contains the index of memory space. The disk is divided into units for bitmap. These units are range from a few bytes to several kilobytes. Each memory unit is associated with a bit in the bitmap. The value of a bit could be either 0 or 1 where 0 indicates the unit is free and 1 indicates that it is occupied. In this file system, two bitmaps has been created for inode and files. In this section, implementation design of bitmap initialization, finding the first unused bit and reset a bit has been described.

2.1.1 Initialization

bitmap_init() function has three parameters, *start*, *num*, and *nbits*. This function is initializing zero value in the bit vector with *num* of sectors starting from *start* sector except that the first *nbits* number of bits. The number of first *nbits* are set to one. The code snippet of this function is shown in Fig 1. A bitmap buffer of character type is created because each character size is 8 bit which is equal to 1 byte. First each bit of the every sector are set to 0 then we changed first *nbits* from 0 to 1 in line# 16-18. A single 1 is shifted to the left shifted and then a bitwise OR operation is performed to make them 1 for number of first *nbits*. *flag1* is used to avoid unnecessary loop execution means if nbits are already set to 1, then no need to check remaining sector/bytes. This improves the efficiency.

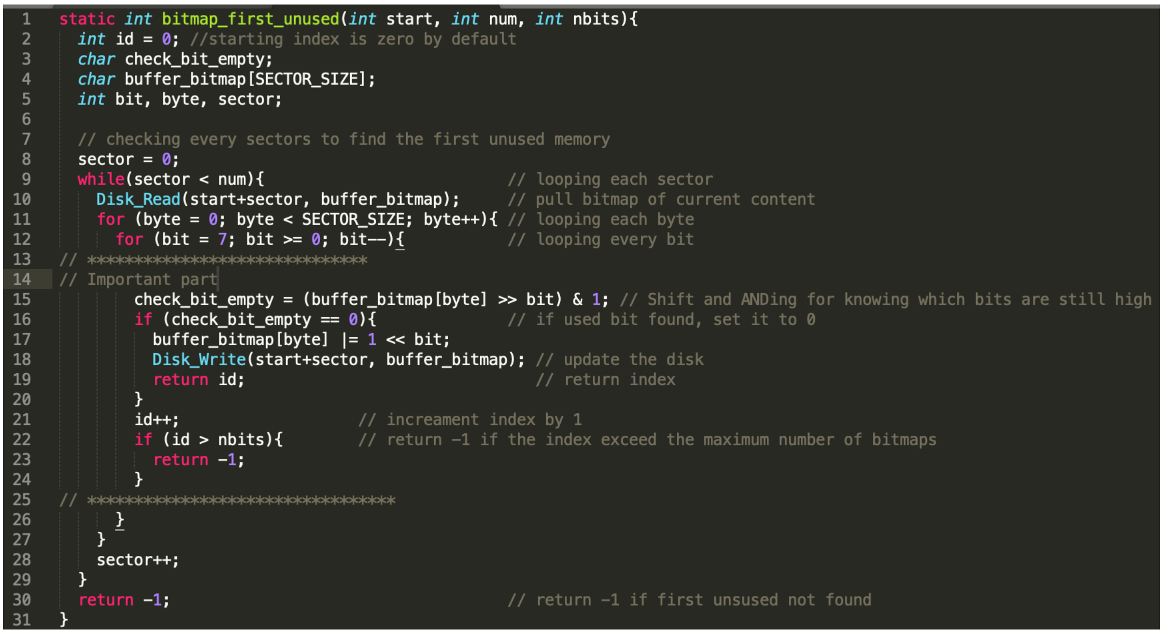


```
1  static void bitmap_init(int start, int num, int nbits){
2    int bit, byte, sector;
3
4    //initializing 0 in bitmap of every sector
5    int flag1 = 1;
6    sector = 0;
7    while(sector < num){
8      char buffer_bitmap[SECTOR_SIZE];          // bitmap size (char because each char is equivalent to 1 byte)
9      memset(buffer_bitmap, 0, SECTOR_SIZE);    // fill block of memory with 0
10     for (byte = 0; byte < SECTOR_SIZE; byte++){ // looping each byte
11       if(!flag1) {
12         break;
13       }
14       else {
15         for (bit = 7; bit >= 0; bit--){          // looping each bit of every byte
16           if (nbits-- > 0){                      // chekcing nbits after decreament by 1
17             buffer_bitmap[byte] |= (1 << bit);
18             }
19           else{
20             flag1 = 0;
21             break;
22             }           // set 1 for first nbits (bitwise OR) for leading the start
23         }
24       }
25     }
26     //update on the disk
27     Disk_Write(start+sector, buffer_bitmap);
28     sector++;
29   }
30 }
```

Figure 1: Implementation of bitmap initialization

2.1.2 Finding First Unused Bit

To allocate memory space, we need to find the index of first unused bit in the bitmap. The implementation of this function is illustrated in Fig. 2. After pulling the current bitmap from the disk, it is checking for first 0 bit through each bit of every byte. Each bit of the current bitmap is shifted to the right and performing a bitwise AND with a 1 to check whether the examining bit is unused(0) or not. Since it is an AND operation, if the reference bit is 0, the result will be also 0 that means we found the first unused bit and the index of that bit is returned after flipping that bit from 0 to 1 using left shift and bitwise OR operation (line#15-19). Otherwise return -1 if the bitmap is already full indicates no more zeros. We improve the efficiency of the code by checking the index(id) in line#22-24. More precisely, if id has already exceed nbits, we are not required to check any further which reduces the execution cost of loop.

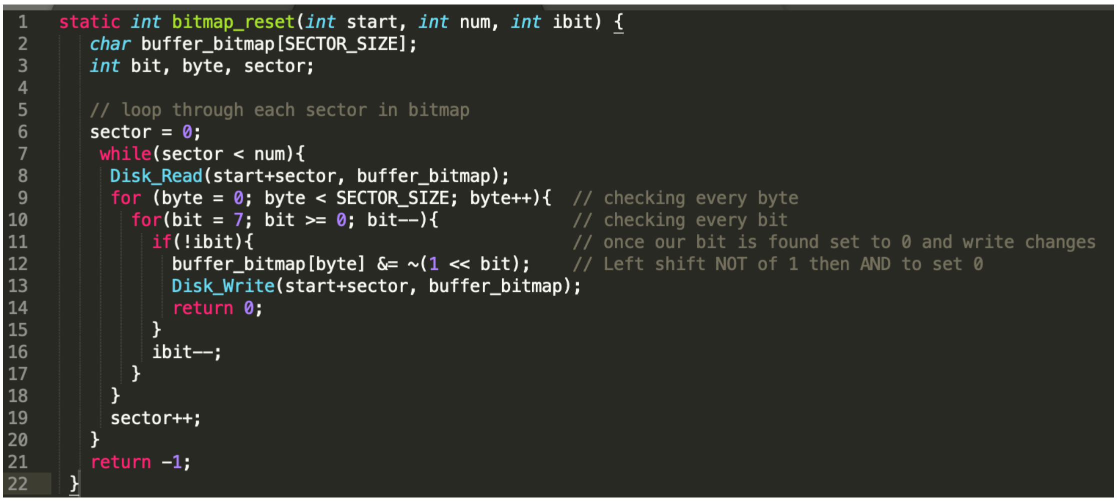


```
1   static int bitmap_first_unused(int start, int num, int nbits){
2     int id = 0; //starting index is zero by default
3     char check_bit_empty;
4     char buffer_bitmap[SECTOR_SIZE];
5     int bit, byte, sector;
6
7     // checking every sectors to find the first unused memory
8     sector = 0;
9     while(sector < num){                          // looping each sector
10      Disk_Read(start+sector, buffer_bitmap);     // pull bitmap of current content
11      for (byte = 0; byte < SECTOR_SIZE; byte++){ // looping each byte
12        for (bit = 7; bit >= 0; bit--){           // looping every bit
13  // ******************************
14  // Important part
15          check_bit_empty = (buffer_bitmap[byte] >> bit) & 1; // Shift and ANDing for knowing which bits are still high
16          if (check_bit_empty == 0){              // if used bit found, set it to 0
17            buffer_bitmap[byte] |= 1 << bit;
18            Disk_Write(start+sector, buffer_bitmap); // update the disk
19            return id;                               // return index
20          }
21          id++;                   // increament index by 1
22          if (id > nbits){        // return -1 if the index exceed the maximum number of bitmaps
23            return -1;
24          }
25  // *********************************
26        }
27      }
28      sector++;
29    }
30    return -1;                                    // return -1 if first unsused not found
31  }
```

Figure 2: Implementation of *bitmap_first_unused()*

2.1.3 Reset

When a file/inode is removed, we need to manage released free space. *bitmap_reset()* function is finding the i-th bit location and set it to 0. A single high(1) bit is left shifted then bitwise NOT is applied to perform an AND operation that reset the ibit. See the implementation snapshop in Fig 3 line#11-16.

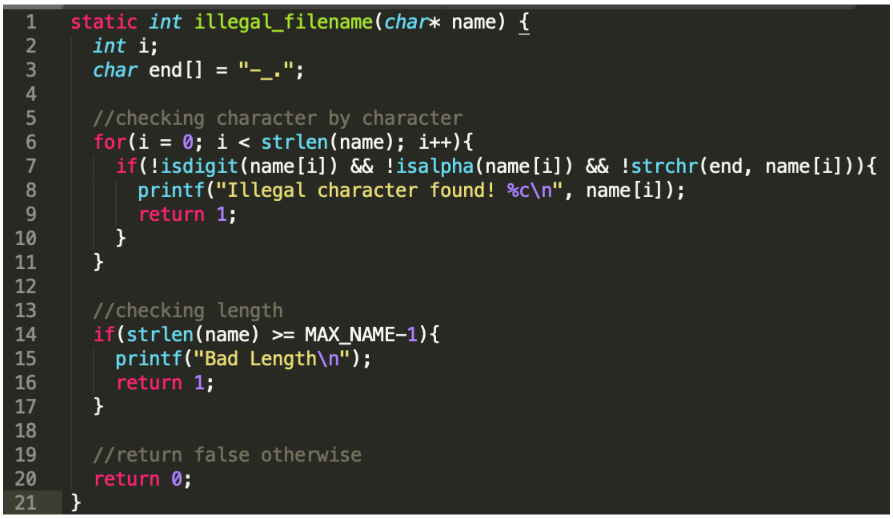


```
1   static int bitmap_reset(int start, int num, int ibit) {
2      char buffer_bitmap[SECTOR_SIZE];
3      int bit, byte, sector;
4
5      // loop through each sector in bitmap
6      sector = 0;
7       while(sector < num){
8        Disk_Read(start+sector, buffer_bitmap);
9        for (byte = 0; byte < SECTOR_SIZE; byte++){  // checking every byte
10         for(bit = 7; bit >= 0; bit--){               // checking every bit
11           if(!ibit){                                 // once our bit is found set to 0 and write changes
12             buffer_bitmap[byte] &= ~(1 << bit);      // Left shift NOT of 1 then AND to set 0
13             Disk_Write(start+sector, buffer_bitmap);
14             return 0;
15           }
16           ibit--;
17         }
18       }
19       sector++;
20     }
21     return -1;
22  }
```

Figure 3: Implementation of *bitmap_reset()*

2.1.4 Checking File Name

At the time of creating new file, given file name should be suitable for OS file system. The implementation of the function *illegal_file_name()* is shown in Fig. 4. Legal characters for a file name include letters (case sensitive), numbers, dots, dashes, and underscores. There are some build-in string function is available those are used for this purpose. *isdigit()* checks given character is digit or not, *isalpha()* checks given character is a alphabet or not. Another part of the code from line#14-17 is checking the length of the file. We have used built-in function *strlen()* to check the length of the filename where file name should not be more than MAX_NAME-1. Finally, it returns 1 if file name/length is incorrect otherwise return 0.



```
1   static int illegal_filename(char* name) {
2     int i;
3     char end[] = "-_.";
4
5     //checking character by character
6     for(i = 0; i < strlen(name); i++){
7       if(!isdigit(name[i]) && !isalpha(name[i]) && !strchr(end, name[i])){
8         printf("Illegal character found! %c\n", name[i]);
9         return 1;
10      }
11    }
12
13    //checking length
14    if(strlen(name) >= MAX_NAME-1){
15      printf("Bad Length\n");
16      return 1;
17    }
18
19    //return false otherwise
20    return 0;
21  }
```

Figure 4: Implementation of *illegal_file_name()*

2.2 Read/Write

2.2.1 File Read

File read function (File_Read()) reads specific bytes from a open file which is referenced by a file descriptor (fd). The data is read into a buffer (function given parameter).

Every open file has a pointer which indicated the current location of the file read/write. This function start reading the file from that position and updated the pointer position at the end of reading.

The code checks if the file is open or not. If it is not open then the code returns -1 and set osErrno to E_BAD_FD. If the file is open then the read operation performed.

The function perform the read for a specified size (function parameter) if there is enough data on the file. If the available data is less the the specified size then only the available portion of the data is returned.

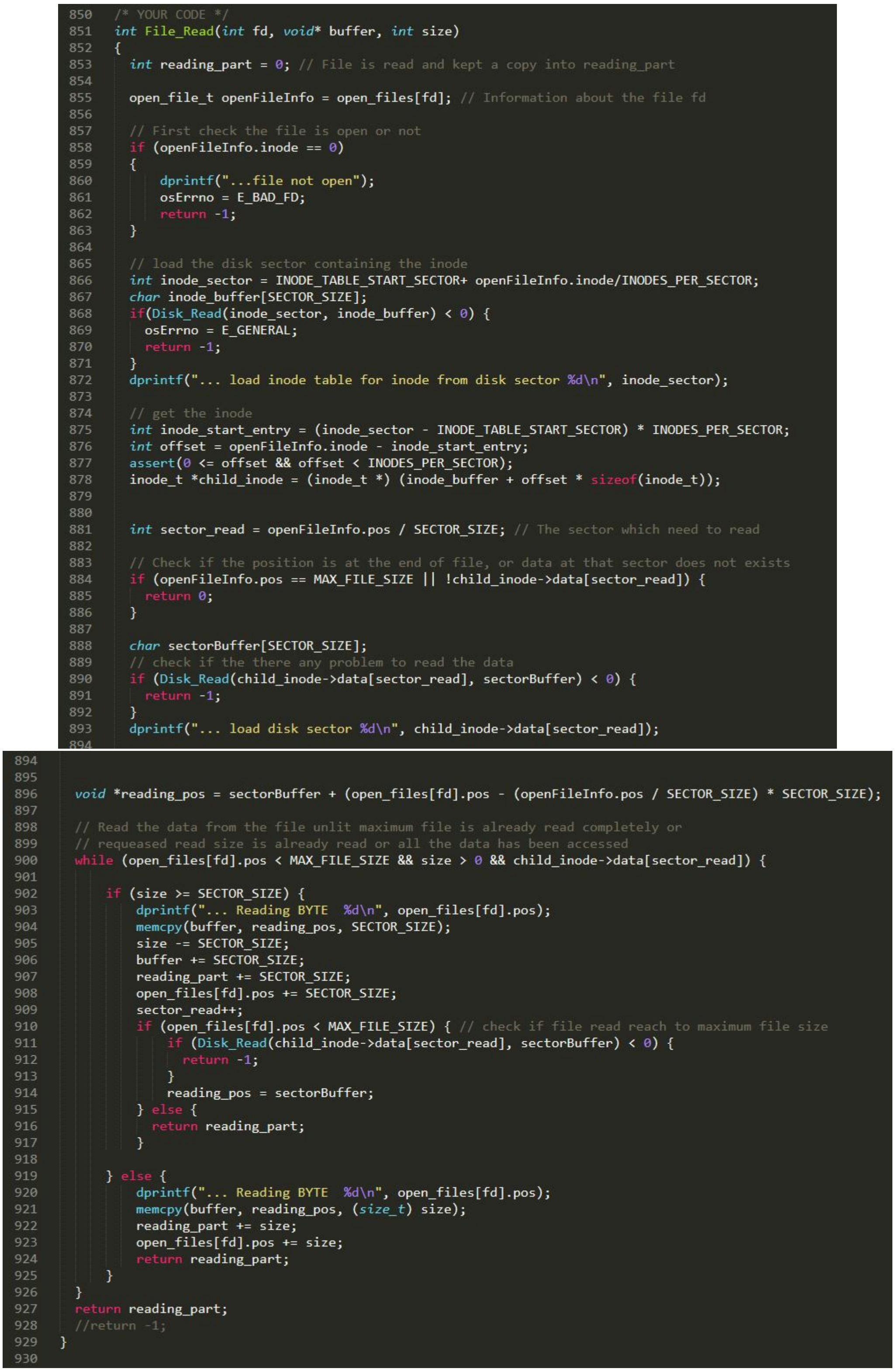
The function also check if the file pointer is already at the end of the file. In such case no data is returned. [Figure 5].

2.2.2 File Write

File write function (File_Write()) write specific bytes from buffer to a open file which is referenced by fd.

Every open file has a pointer which indicated the current location of the file read/write. This function start writing the file from that position and updated the pointer position at the end of writing.

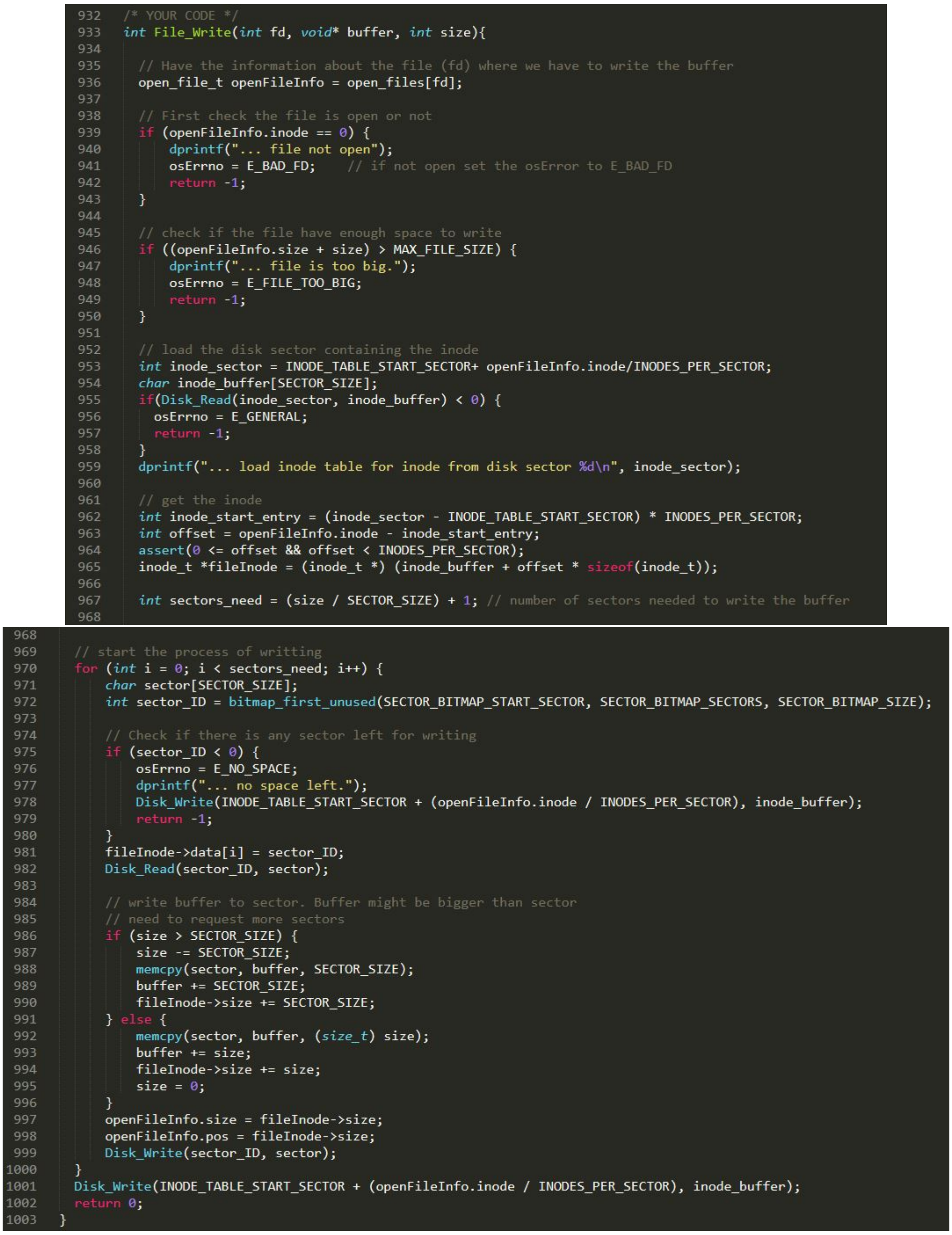
The code checks if the file is open or not. If it is not open then the code returns -1 and set osErrno to E_BAD_FD. If the file is open then the write operation performed and value of written file size if return.



```c
/* YOUR CODE */
int File_Read(int fd, void* buffer, int size)
{
  int reading_part = 0; // File is read and kept a copy into reading_part

  open_file_t openFileInfo = open_files[fd]; // Information about the file fd

  // First check the file is open or not
  if (openFileInfo.inode == 0)
  {
      dprintf("...file not open");
      osErrno = E_BAD_FD;
      return -1;
  }

  // load the disk sector containing the inode
  int inode_sector = INODE_TABLE_START_SECTOR+ openFileInfo.inode/INODES_PER_SECTOR;
  char inode_buffer[SECTOR_SIZE];
  if(Disk_Read(inode_sector, inode_buffer) < 0) {
    osErrno = E_GENERAL;
    return -1;
  }
  dprintf("... load inode table for inode from disk sector %d\n", inode_sector);

  // get the inode
  int inode_start_entry = (inode_sector - INODE_TABLE_START_SECTOR) * INODES_PER_SECTOR;
  int offset = openFileInfo.inode - inode_start_entry;
  assert(0 <= offset && offset < INODES_PER_SECTOR);
  inode_t *child_inode = (inode_t *) (inode_buffer + offset * sizeof(inode_t));


  int sector_read = openFileInfo.pos / SECTOR_SIZE; // The sector which need to read

  // Check if the position is at the end of file, or data at that sector does not exists
  if (openFileInfo.pos == MAX_FILE_SIZE || !child_inode->data[sector_read]) {
    return 0;
  }

  char sectorBuffer[SECTOR_SIZE];
  // check if the there any problem to read the data
  if (Disk_Read(child_inode->data[sector_read], sectorBuffer) < 0) {
    return -1;
  }
  dprintf("... load disk sector %d\n", child_inode->data[sector_read]);


  void *reading_pos = sectorBuffer + (open_files[fd].pos - (openFileInfo.pos / SECTOR_SIZE) * SECTOR_SIZE);

  // Read the data from the file unlit maximum file is already read completely or
  // requeased read size is already read or all the data has been accessed
  while (open_files[fd].pos < MAX_FILE_SIZE && size > 0 && child_inode->data[sector_read]) {

      if (size >= SECTOR_SIZE) {
          dprintf("... Reading BYTE  %d\n", open_files[fd].pos);
          memcpy(buffer, reading_pos, SECTOR_SIZE);
          size -= SECTOR_SIZE;
          buffer += SECTOR_SIZE;
          reading_part += SECTOR_SIZE;
          open_files[fd].pos += SECTOR_SIZE;
          sector_read++;
          if (open_files[fd].pos < MAX_FILE_SIZE) { // check if file read reach to maximum file size
              if (Disk_Read(child_inode->data[sector_read], sectorBuffer) < 0) {
                return -1;
              }
              reading_pos = sectorBuffer;
          } else {
            return reading_part;
          }

      } else {
          dprintf("... Reading BYTE  %d\n", open_files[fd].pos);
          memcpy(buffer, reading_pos, (size_t) size);
          reading_part += size;
          open_files[fd].pos += size;
          return reading_part;
      }
  }
  return reading_part;
  //return -1;
}
```

Figure 5: Implementation of *File_Read()*

The code also checks if the write can not be performed because of lack of space or file exceed maximum file size. In case of lack of space, it returns -1 and set osErrno to E_NO_SPACE. if the file exceeds the maximum file size, it also returns -1 and set osErrno to E_FILE_TOO_BIG.[Figure 6].

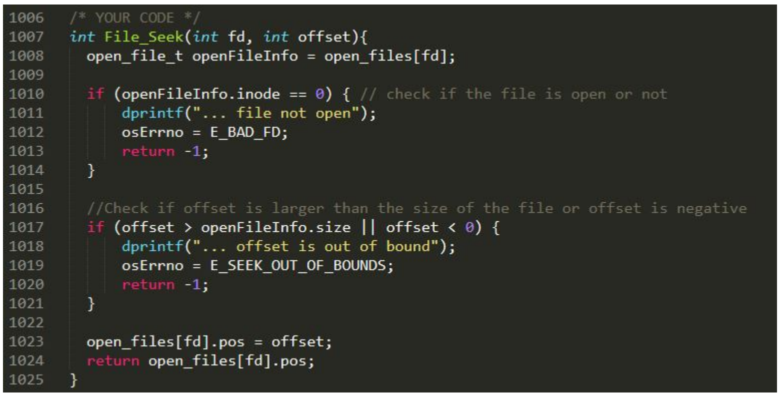


```c
/* YOUR CODE */
int File_Write(int fd, void* buffer, int size){

    // Have the information about the file (fd) where we have to write the buffer
    open_file_t openFileInfo = open_files[fd];

    // First check the file is open or not
    if (openFileInfo.inode == 0) {
        dprintf("... file not open");
        osErrno = E_BAD_FD;    // if not open set the osError to E_BAD_FD
        return -1;
    }

    // check if the file have enough space to write
    if ((openFileInfo.size + size) > MAX_FILE_SIZE) {
        dprintf("... file is too big.");
        osErrno = E_FILE_TOO_BIG;
        return -1;
    }

    // load the disk sector containing the inode
    int inode_sector = INODE_TABLE_START_SECTOR+ openFileInfo.inode/INODES_PER_SECTOR;
    char inode_buffer[SECTOR_SIZE];
    if(Disk_Read(inode_sector, inode_buffer) < 0) {
      osErrno = E_GENERAL;
      return -1;
    }
    dprintf("... load inode table for inode from disk sector %d\n", inode_sector);

    // get the inode
    int inode_start_entry = (inode_sector - INODE_TABLE_START_SECTOR) * INODES_PER_SECTOR;
    int offset = openFileInfo.inode - inode_start_entry;
    assert(0 <= offset && offset < INODES_PER_SECTOR);
    inode_t *fileInode = (inode_t *) (inode_buffer + offset * sizeof(inode_t));

    int sectors_need = (size / SECTOR_SIZE) + 1; // number of sectors needed to write the buffer

    // start the process of writting
    for (int i = 0; i < sectors_need; i++) {
        char sector[SECTOR_SIZE];
        int sector_ID = bitmap_first_unused(SECTOR_BITMAP_START_SECTOR, SECTOR_BITMAP_SECTORS, SECTOR_BITMAP_SIZE);

        // Check if there is any sector left for writing
        if (sector_ID < 0) {
            osErrno = E_NO_SPACE;
            dprintf("... no space left.");
            Disk_Write(INODE_TABLE_START_SECTOR + (openFileInfo.inode / INODES_PER_SECTOR), inode_buffer);
            return -1;
        }
        fileInode->data[i] = sector_ID;
        Disk_Read(sector_ID, sector);

        // write buffer to sector. Buffer might be bigger than sector
        // need to request more sectors
        if (size > SECTOR_SIZE) {
            size -= SECTOR_SIZE;
            memcpy(sector, buffer, SECTOR_SIZE);
            buffer += SECTOR_SIZE;
            fileInode->size += SECTOR_SIZE;
        } else {
            memcpy(sector, buffer, (size_t) size);
            buffer += size;
            fileInode->size += size;
            size = 0;
        }
        openFileInfo.size = fileInode->size;
        openFileInfo.pos = fileInode->size;
        Disk_Write(sector_ID, sector);
    }
    Disk_Write(INODE_TABLE_START_SECTOR + (openFileInfo.inode / INODES_PER_SECTOR), inode_buffer);
    return 0;
}
```

Figure 6: Implementation of *File_Write()*

2.2.3 File Seek

File seek function (File_Seek()) updates the current location of the file pointer which is provided as an offset from the beginning of the file.
The function first checks if the file is open or not. If it is not open then the code returns -1 and set osErrno to E_BAD_FD. If the file is open then the file seek operation is performed. This function also checks if the offset is larger than the file size or negative then it returns -1 and set osErrno to E_SEEK_OUT_OF_BOUNDS. [Figure 7].



```c
/* YOUR CODE */
int File_Seek(int fd, int offset){
  open_file_t openFileInfo = open_files[fd];

  if (openFileInfo.inode == 0) { // check if the file is open or not
      dprintf("... file not open");
      osErrno = E_BAD_FD;
      return -1;
  }

  //Check if offset is larger than the size of the file or offset is negative
  if (offset > openFileInfo.size || offset < 0) {
      dprintf("... offset is out of bound");
      osErrno = E_SEEK_OUT_OF_BOUNDS;
      return -1;
  }

  open_files[fd].pos = offset;
  return open_files[fd].pos;
}
```

Figure 7: Implementation of *File_Seek()*

2.2.4 Directory Read

Directory read function (Dir_Read()) reads the contents of a directory and returns the set of directory entries into the buffer.
The function check if the buffer size is large enough to hold all the entries of the directory. If not then the function returns -1 and set osErrno to E_BUFFER_TOO_SMALL.
If the function successfully read the data into the buffer then it returns the number of directory entries that are in the directory.[Figure 8].

2.2.5 Directory Size

Directory size function (Dir_Size()) returns the number of bytes in the directory referred to by path.
This function first check if the directory exists or not. If exists then it also check if it is a directory or file. If it is a directory then the size of the directory is returned. [Figure 9].

2.3 Remove

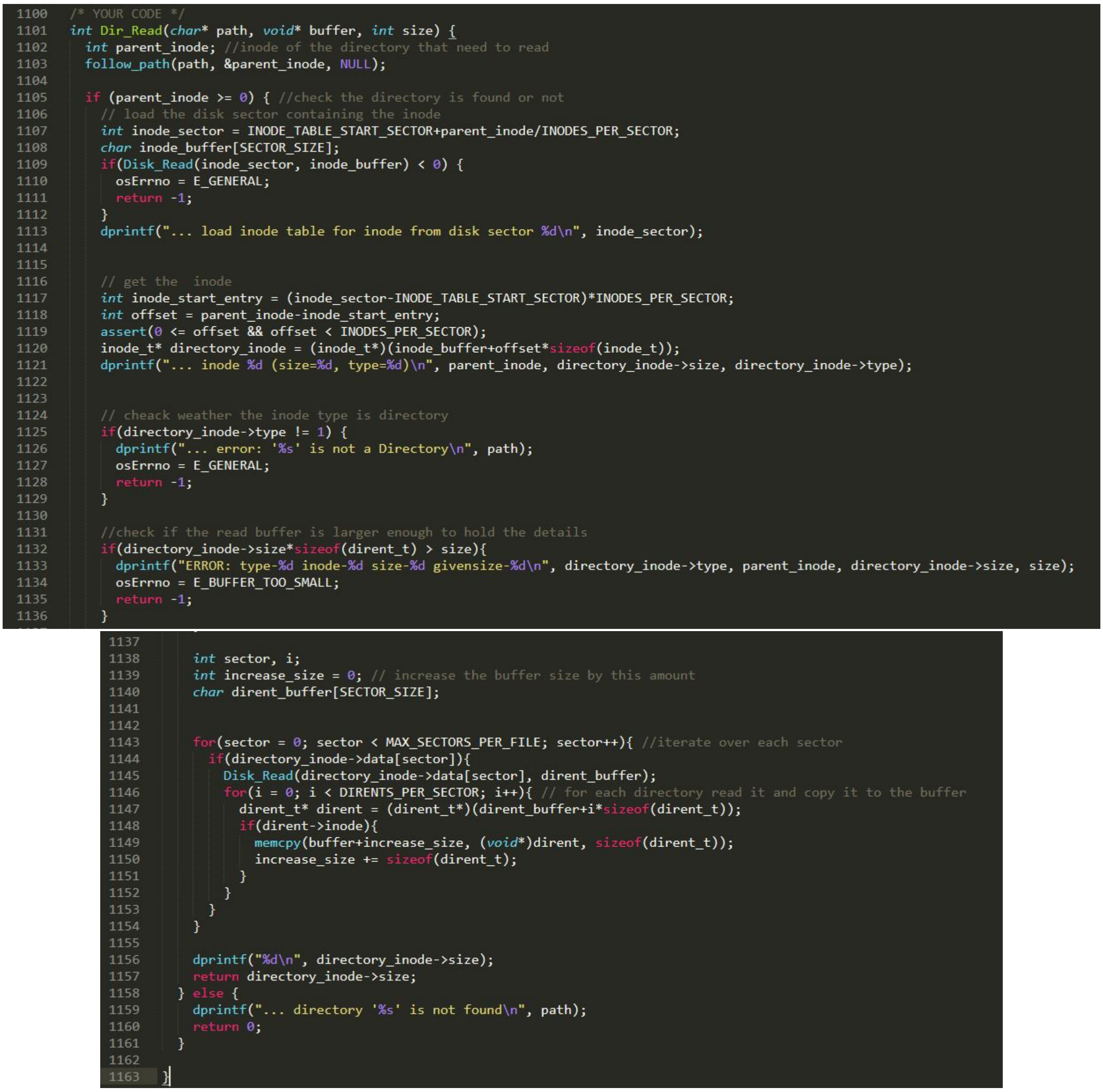
In order to remove the file/directory, we need to follow similar steps as create file/directory. In particular, we need to remove corresponding iNode, update the iNodes Bitmap, remove the content from the disk, and update the Sector Bitmap.
The following functions are responsible for the object removing:

- *File_Unlink(char* file)* - to remove the file with path *file*
- *Dir_Unlink(char* path)* - to remove the directory with path *path*

Both functions *File_Unlink(char* file)* and *Dir_Unlink(char* path)* (Figure 10) use the helper function *remove_file_or_directory(int type, char* pathname)*, which has similar structure with *create_file_or_directory(int type, char* pathname)*. Variable *type* corresponds to the type of object (0 for files, and 1 for directory). Attribute *pathname* is the absolute path of the file/directory that we want to remove.

remove_file_or_directory(int type, char* pathname)

On Figure 11 you can observe the structure of the function *remove_file_or_directory(int type, char* pathname)*.



```c
/* YOUR CODE */
int Dir_Read(char* path, void* buffer, int size) {
  int parent_inode; //inode of the directory that need to read
  follow_path(path, &parent_inode, NULL);

  if (parent_inode >= 0) { //check the directory is found or not
    // load the disk sector containing the inode
    int inode_sector = INODE_TABLE_START_SECTOR+parent_inode/INODES_PER_SECTOR;
    char inode_buffer[SECTOR_SIZE];
    if(Disk_Read(inode_sector, inode_buffer) < 0) {
      osErrno = E_GENERAL;
      return -1;
    }
    dprintf("... load inode table for inode from disk sector %d\n", inode_sector);


    // get the  inode
    int inode_start_entry = (inode_sector-INODE_TABLE_START_SECTOR)*INODES_PER_SECTOR;
    int offset = parent_inode-inode_start_entry;
    assert(0 <= offset && offset < INODES_PER_SECTOR);
    inode_t* directory_inode = (inode_t*)(inode_buffer+offset*sizeof(inode_t));
    dprintf("... inode %d (size=%d, type=%d)\n", parent_inode, directory_inode->size, directory_inode->type);


    // cheack weather the inode type is directory
    if(directory_inode->type != 1) {
      dprintf("... error: '%s' is not a Directory\n", path);
      osErrno = E_GENERAL;
      return -1;
    }

    //check if the read buffer is larger enough to hold the details
    if(directory_inode->size*sizeof(dirent_t) > size){
      dprintf("ERROR: type-%d inode-%d size-%d givensize-%d\n", directory_inode->type, parent_inode, directory_inode->size, size);
      osErrno = E_BUFFER_TOO_SMALL;
      return -1;
    }

    int sector, i;
    int increase_size = 0; // increase the buffer size by this amount
    char dirent_buffer[SECTOR_SIZE];


    for(sector = 0; sector < MAX_SECTORS_PER_FILE; sector++){ //iterate over each sector
      if(directory_inode->data[sector]){
        Disk_Read(directory_inode->data[sector], dirent_buffer);
        for(i = 0; i < DIRENTS_PER_SECTOR; i++){ // for each directory read it and copy it to the buffer
          dirent_t* dirent = (dirent_t*)(dirent_buffer+i*sizeof(dirent_t));
          if(dirent->inode){
            memcpy(buffer+increase_size, (void*)dirent, sizeof(dirent_t));
            increase_size += sizeof(dirent_t);
          }
        }
      }
    }

    dprintf("%d\n", directory_inode->size);
    return directory_inode->size;
  } else {
    dprintf("... directory '%s' is not found\n", path);
    return 0;
  }

}
```

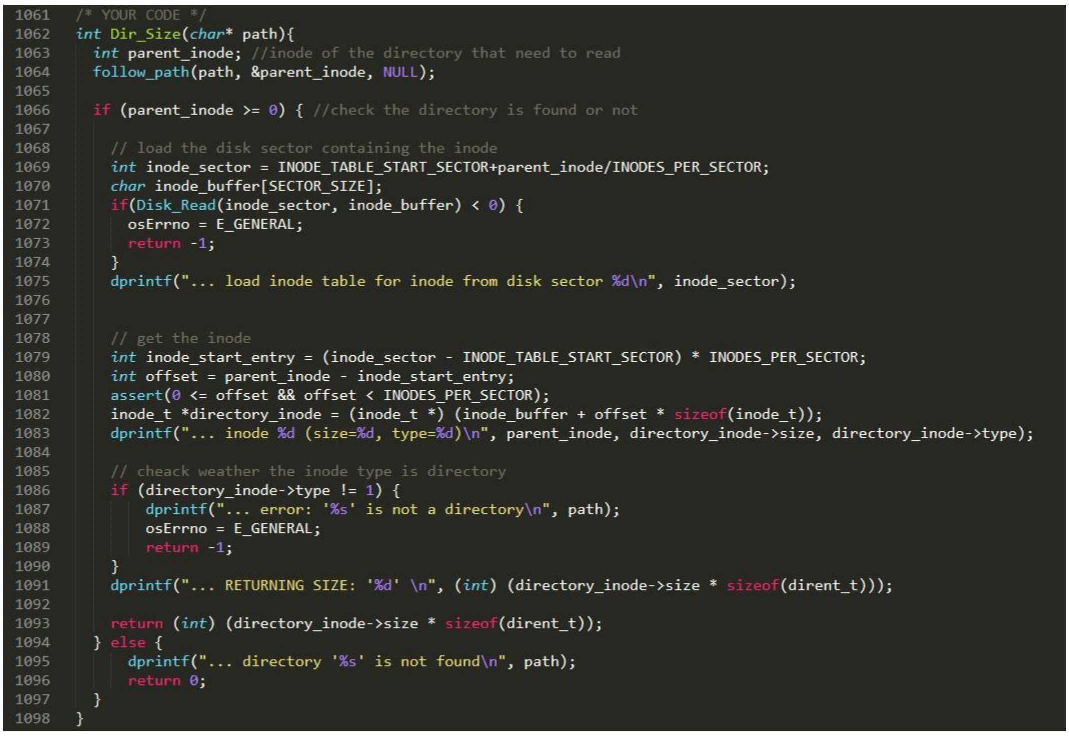
Figure 8: Implementation of *Dir_Read()*

On line 9 we are finding iNode numbers of the file and it's parent iNode using function *follow_path(char* path, int* last_inode, char* last_fname)*. This function divides the path into tokens by splitting it with "/" character. Then starting from the root iNode, by iterating through each iNode child and comparing the corresponding filename entry in the *dirent* table with the next level in the path, we are finding the iNode number that corresponds to the last file in the path.

If the iNode for a file (lines 14-21) or parent (lines 48-53) can't be found, we are returning "No such file/directory" error.

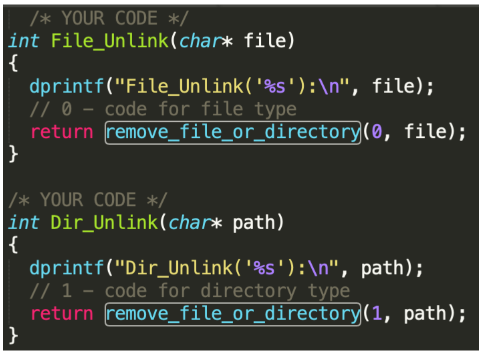
On lines 26-28, we are checking if the file is currently open. If so, we can't remove it, since other application might overwrite our changes.

If a child and parent iNode is found, and the file is not currently open, we are attempting to remove the iNode (line 32), by calling the function *remove_inode(int type, int parent_inode, int child_inode)*.



```c
/* YOUR CODE */
int Dir_Size(char* path){
  int parent_inode; //inode of the directory that need to read
  follow_path(path, &parent_inode, NULL);

  if (parent_inode >= 0) { //check the directory is found or not

    // load the disk sector containing the inode
    int inode_sector = INODE_TABLE_START_SECTOR+parent_inode/INODES_PER_SECTOR;
    char inode_buffer[SECTOR_SIZE];
    if(Disk_Read(inode_sector, inode_buffer) < 0) {
      osErrno = E_GENERAL;
      return -1;
    }
    dprintf("... load inode table for inode from disk sector %d\n", inode_sector);


    // get the inode
    int inode_start_entry = (inode_sector - INODE_TABLE_START_SECTOR) * INODES_PER_SECTOR;
    int offset = parent_inode - inode_start_entry;
    assert(0 <= offset && offset < INODES_PER_SECTOR);
    inode_t *directory_inode = (inode_t *) (inode_buffer + offset * sizeof(inode_t));
    dprintf("... inode %d (size=%d, type=%d)\n", parent_inode, directory_inode->size, directory_inode->type);

    // cheack weather the inode type is directory
    if (directory_inode->type != 1) {
      dprintf("... error: '%s' is not a directory\n", path);
      osErrno = E_GENERAL;
      return -1;
    }
    dprintf("... RETURNING SIZE: '%d' \n", (int) (directory_inode->size * sizeof(dirent_t)));

    return (int) (directory_inode->size * sizeof(dirent_t));
  } else {
    dprintf("... directory '%s' is not found\n", path);
    return 0;
  }
}
```

Figure 9: Implementation of *Dir_Size()*

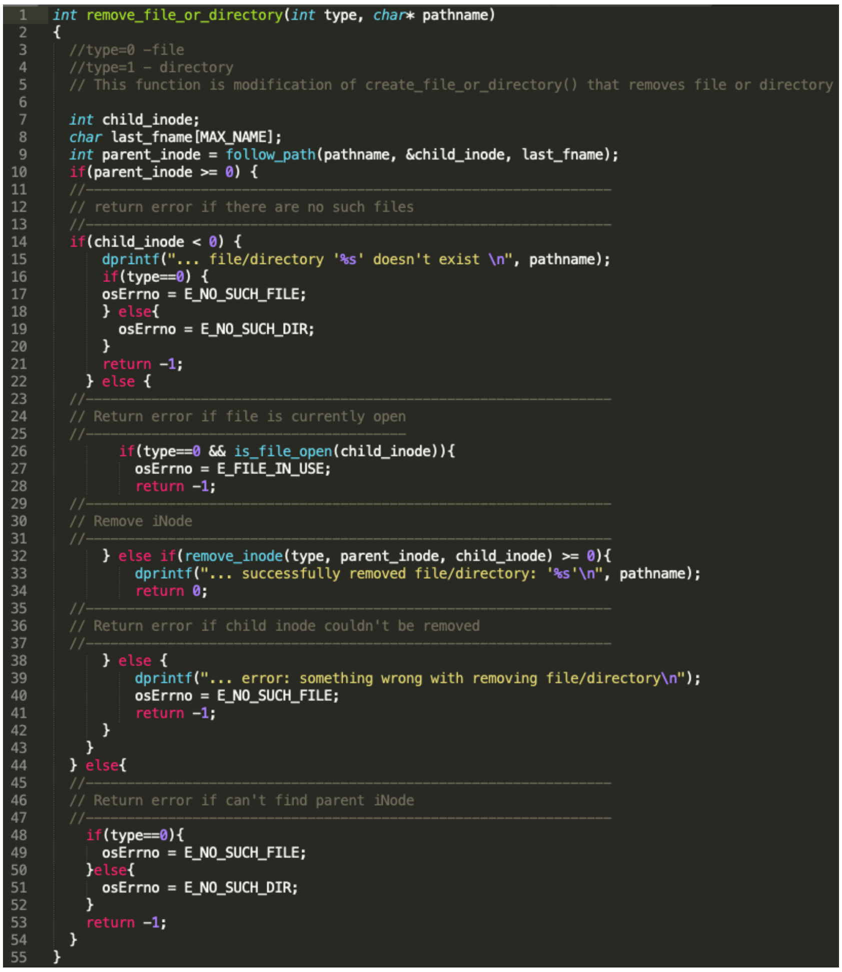


```c
 /* YOUR CODE */
int File_Unlink(char* file)
{
  dprintf("File_Unlink('%s'):\n", file);
  // 0 - code for file type
  return remove_file_or_directory(0, file);
}

/* YOUR CODE */
int Dir_Unlink(char* path)
{
  dprintf("Dir_Unlink('%s'):\n", path);
  // 1 - code for directory type
  return remove_file_or_directory(1, path);
}
```

Figure 10: General structure of functions to remove File or Directory

remove_inode(int type, int parent_inode, int child_inode)

This function takes care of removing the child iNode, removing the link from the parent iNode to child iNode, removing the content of the file, removing iNode from the disk, and updating sector bitmap and iNode bitmap. The screenshot of this function is at Figure 12.
First, we are loading the child iNode from the disk (lines 9-22). Since on previous step we obtained the child iNode number, we know its exact location on a disk. Therefore, we just load the iNode content to the buffer and transform it into iNode type.
Second, we check for errors (lines 28-38): if the type in the argument doesn't matche the iNode type or if we are trying to remove the directory that is not empty.
Third, if we are removing the file, we need to remove all it's content from the disk (lines 43-60). Since the child iNode contains the addresses of each data block of the file to the disk sector where it's belongs to, we just scan through each such sector and set all the bits there to zero. Along with updating the disk, we also set the bitmap entry that corresponds to each sector of the file to zero.
Forth, we are removing the child iNode from the disk (lines 65-67). Those, we are setting all



```
int remove_file_or_directory(int type, char* pathname)
{
  //type=0 -file
  //type=1 - directory
  // This function is modification of create_file_or_directory() that removes file or directory

  int child_inode;
  char last_fname[MAX_NAME];
  int parent_inode = follow_path(pathname, &child_inode, last_fname);
  if(parent_inode >= 0) {
  //-------------------------------------------------------------
  // return error if there are no such files
  //-------------------------------------------------------------
  if(child_inode < 0) {
      dprintf("... file/directory '%s' doesn't exist \n", pathname);
      if(type==0) {
      osErrno = E_NO_SUCH_FILE;
      } else{
        osErrno = E_NO_SUCH_DIR;
      }
      return -1;
    } else {
  //-------------------------------------------------------------
  // Return error if file is currently open
  //--------------------------------------
        if(type==0 && is_file_open(child_inode)){
          osErrno = E_FILE_IN_USE;
          return -1;
  //-------------------------------------------------------------
  // Remove iNode
  //-------------------------------------------------------------
      } else if(remove_inode(type, parent_inode, child_inode) >= 0){
          dprintf("... successfully removed file/directory: '%s'\n", pathname);
          return 0;
  //-------------------------------------------------------------
  // Return error if child inode couldn't be removed
  //-------------------------------------------------------------
      } else {
          dprintf("... error: something wrong with removing file/directory\n");
          osErrno = E_NO_SUCH_FILE;
          return -1;
      }
    }
  } else{
  //-------------------------------------------------------------
  // Return error if can't find parent iNode
  //-------------------------------------------------------------
    if(type==0){
      osErrno = E_NO_SUCH_FILE;
    }else{
      osErrno = E_NO_SUCH_DIR;
    }
    return -1;
  }
}
```

Figure 11: Function to remove file or directory

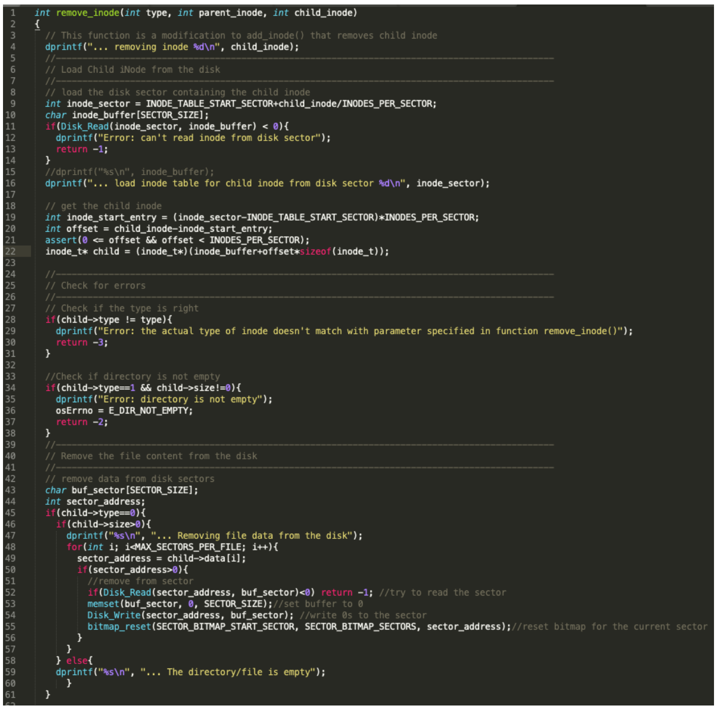
bits of the iNode's sector to zero. Since we freed the disk space, we need to update the iNode bitmap entry that corresponds to newly free sector (line 72)

Finally, we need to remove the pointer from the parent iNode to the child iNode. For that, we load the parent iNode from the disk (lines 78-89) in the similar maner that we did for the child iNode. Then, we iterate through each data pointer of the parent iNode, and check if it points to the child iNode. If it is, than we remove this pointer from the parent iNode, and we also remove the corresponding entry in a dirent table that maps the child iNode number with it's file name. (lines 100-125).

3 Testing

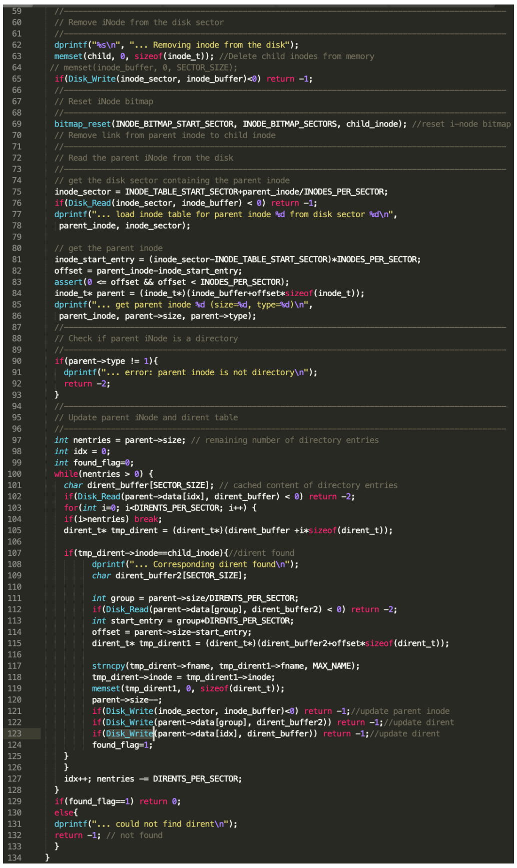
We have created some custom test cases to show the basic functionalities of the implemented file system. The rusult of make command is shown in Fig. 13. Before running custom test cases, we run simple-test.exe to make sure that our code is working fine. Results of simple test is shown in Fig. 14 that ensures that the following functions are working fine.

- bitmap initialization, finding first unused bit, and reset
- illegal file name
- Creating file/directory



```c
int remove_inode(int type, int parent_inode, int child_inode)
{
  // This function is a modification to add_inode() that removes child inode
  dprintf("... removing inode %d\n", child_inode);
  //----------------------------------------------------------------------------
  // Load Child iNode from the disk
  //----------------------------------------------------------------------------
  // load the disk sector containing the child inode
  int inode_sector = INODE_TABLE_START_SECTOR+child_inode/INODES_PER_SECTOR;
  char inode_buffer[SECTOR_SIZE];
  if(Disk_Read(inode_sector, inode_buffer) < 0){
    dprintf("Error: can't read inode from disk sector");
    return -1;
  }
  //dprintf("%s\n", inode_buffer);
  dprintf("... load inode table for child inode from disk sector %d\n", inode_sector);

  // get the child inode
  int inode_start_entry = (inode_sector-INODE_TABLE_START_SECTOR)*INODES_PER_SECTOR;
  int offset = child_inode-inode_start_entry;
  assert(0 <= offset && offset < INODES_PER_SECTOR);
  inode_t* child = (inode_t*)(inode_buffer+offset*sizeof(inode_t));

  //----------------------------------------------------------------------------
  // Check for errors
  //----------------------------------------------------------------------------
  // Check if the type is right
  if(child->type != type){
    dprintf("Error: the actual type of inode doesn't match with parameter specified in function remove_inode()");
    return -3;
  }

  //Check if directory is not empty
  if(child->type==1 && child->size!=0){
    dprintf("Error: directory is not empty");
    osErrno = E_DIR_NOT_EMPTY;
    return -2;
  }
  //----------------------------------------------------------------------------
  // Remove the file content from the disk
  //----------------------------------------------------------------------------
  // remove data from disk sectors
  char buf_sector[SECTOR_SIZE];
  int sector_address;
  if(child->type==0){
    if(child->size>0){
      dprintf("%s\n", "... Removing file data from the disk");
      for(int i; i<MAX_SECTORS_PER_FILE; i++){
        sector_address = child->data[i];
        if(sector_address>0){
          //remove from sector
          if(Disk_Read(sector_address, buf_sector)<0) return -1; //try to read the sector
          memset(buf_sector, 0, SECTOR_SIZE);//set buffer to 0
          Disk_Write(sector_address, buf_sector); //write 0s to the sector
          bitmap_reset(SECTOR_BITMAP_START_SECTOR, SECTOR_BITMAP_SECTORS, sector_address);//reset bitmap for the current sector
        }
      }
    } else{
    dprintf("%s\n", "... The directory/file is empty");
      }
  }
```

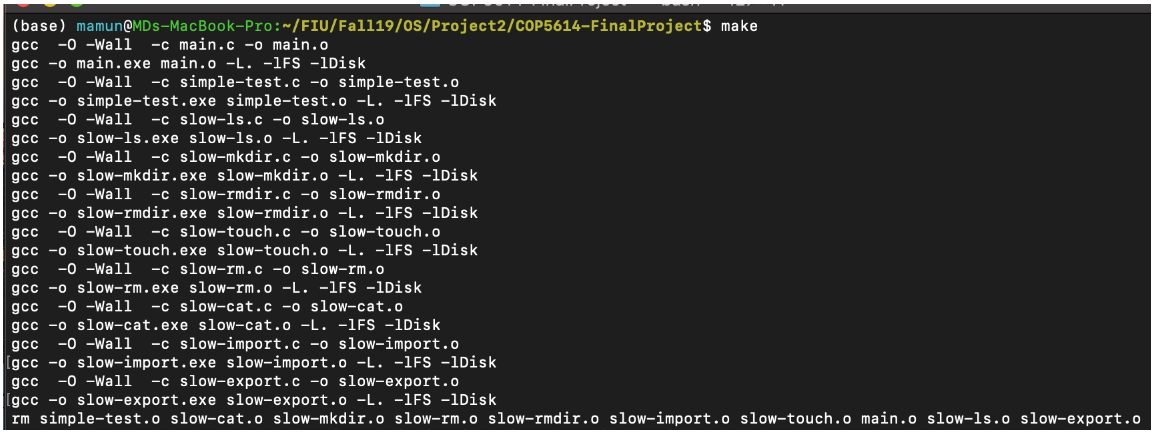
(a) Loading child iNode and removing the content from the disk



```c
    //----------------------------------------------------------------------------
    // Remove iNode from the disk sector
    //----------------------------------------------------------------------------
    dprintf("%s\n", "... Removing inode from the disk");
    memset(child, 0, sizeof(inode_t)); //Delete child inodes from memory
  // memset(inode_buffer, 0, SECTOR_SIZE);
    if(Disk_Write(inode_sector, inode_buffer)<0) return -1;
    //----------------------------------------------------------------------------
    // Reset iNode bitmap
    //----------------------------------------------------------------------------
    bitmap_reset(INODE_BITMAP_START_SECTOR, INODE_BITMAP_SECTORS, child_inode); //reset i-node bitmap
    // Remove link from parent inode to child inode
    //----------------------------------------------------------------------------
    // Read the parent iNode from the disk
    //----------------------------------------------------------------------------
    // get the disk sector containing the parent inode
    inode_sector = INODE_TABLE_START_SECTOR+parent_inode/INODES_PER_SECTOR;
    if(Disk_Read(inode_sector, inode_buffer) < 0) return -1;
    dprintf("... load inode table for parent inode %d from disk sector %d\n",
     parent_inode, inode_sector);

    // get the parent inode
    inode_start_entry = (inode_sector-INODE_TABLE_START_SECTOR)*INODES_PER_SECTOR;
    offset = parent_inode-inode_start_entry;
    assert(0 <= offset && offset < INODES_PER_SECTOR);
    inode_t* parent = (inode_t*)(inode_buffer+offset*sizeof(inode_t));
    dprintf("... get parent inode %d (size=%d, type=%d)\n",
     parent_inode, parent->size, parent->type);
    //----------------------------------------------------------------------------
    // Check if parent iNode is a directory
    //----------------------------------------------------------------------------
    if(parent->type != 1){
      dprintf("... error: parent inode is not directory\n");
      return -2;
    }
    //----------------------------------------------------------------------------
    // Update parent iNode and dirent table
    //----------------------------------------------------------------------------
    int nentries = parent->size; // remaining number of directory entries
    int idx = 0;
    int found_flag=0;
    while(nentries > 0) {
      char dirent_buffer[SECTOR_SIZE]; // cached content of directory entries
      if(Disk_Read(parent->data[idx], dirent_buffer) < 0) return -2;
      for(int i=0; i<DIRENTS_PER_SECTOR; i++) {
      if(i>nentries) break;
      dirent_t* tmp_dirent = (dirent_t*)(dirent_buffer +i*sizeof(dirent_t));

      if(tmp_dirent->inode==child_inode){//dirent found
            dprintf("... Corresponding dirent found\n");
            char dirent_buffer2[SECTOR_SIZE];

            int group = parent->size/DIRENTS_PER_SECTOR;
            if(Disk_Read(parent->data[group], dirent_buffer2) < 0) return -2;
            int start_entry = group*DIRENTS_PER_SECTOR;
            offset = parent->size-start_entry;
            dirent_t* tmp_dirent1 = (dirent_t*)(dirent_buffer2+offset*sizeof(dirent_t));

            strncpy(tmp_dirent->fname, tmp_dirent1->fname, MAX_NAME);
            tmp_dirent->inode = tmp_dirent1->inode;
            memset(tmp_dirent1, 0, sizeof(dirent_t));
            parent->size--;
            if(Disk_Write(inode_sector, inode_buffer)<0) return -1;//update parent inode
            if(Disk_Write(parent->data[group], dirent_buffer2)) return -1;//update dirent
            if(Disk_Write(parent->data[idx], dirent_buffer)) return -1;//update dirent
            found_flag=1;
      }
      }
      idx++; nentries -= DIRENTS_PER_SECTOR;
    }
    if(found_flag==1) return 0;
    else{
    dprintf("... could not find dirent\n");
    return -1; // not found
    }
  }
```

(b) Resetting bitmaps, removing child iNode from the disk, updating parent iNode and dirent table

Figure 12: Function to remove iNode



```
(base) mamun@MDs-MacBook-Pro:~/FIU/Fall19/OS/Project2/COP5614-FinalProject$ make
gcc  -O -Wall  -c main.c -o main.o
gcc -o main.exe main.o -L. -lFS -lDisk
gcc  -O -Wall  -c simple-test.c -o simple-test.o
gcc -o simple-test.exe simple-test.o -L. -lFS -lDisk
gcc  -O -Wall  -c slow-ls.c -o slow-ls.o
gcc -o slow-ls.exe slow-ls.o -L. -lFS -lDisk
gcc  -O -Wall  -c slow-mkdir.c -o slow-mkdir.o
gcc -o slow-mkdir.exe slow-mkdir.o -L. -lFS -lDisk
gcc  -O -Wall  -c slow-rmdir.c -o slow-rmdir.o
gcc -o slow-rmdir.exe slow-rmdir.o -L. -lFS -lDisk
gcc  -O -Wall  -c slow-touch.c -o slow-touch.o
gcc -o slow-touch.exe slow-touch.o -L. -lFS -lDisk
gcc  -O -Wall  -c slow-rm.c -o slow-rm.o
gcc -o slow-rm.exe slow-rm.o -L. -lFS -lDisk
gcc  -O -Wall  -c slow-cat.c -o slow-cat.o
gcc -o slow-cat.exe slow-cat.o -L. -lFS -lDisk
gcc  -O -Wall  -c slow-import.c -o slow-import.o
[gcc -o slow-import.exe slow-import.o -L. -lFS -lDisk
gcc  -O -Wall  -c slow-export.c -o slow-export.o
[gcc -o slow-export.exe slow-export.o -L. -lFS -lDisk
rm simple-test.o slow-cat.o slow-mkdir.o slow-rm.o slow-rmdir.o slow-import.o slow-touch.o main.o slow-ls.o slow-export.o
```

Figure 13: Output of 'make'

- Removing file/directory/inode

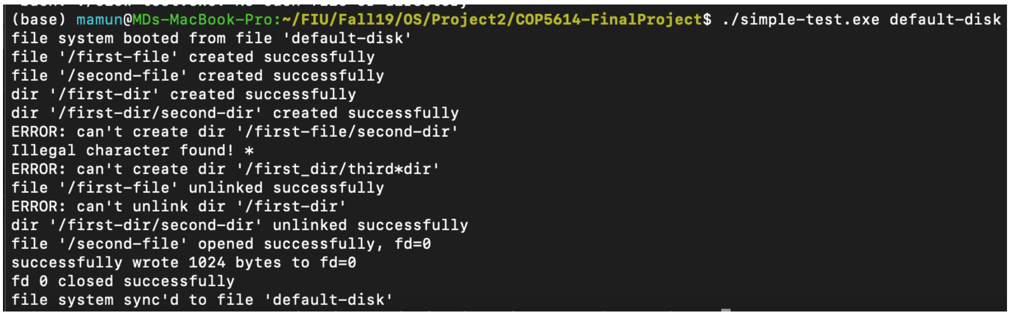
We design a custom test cases as similar as Fig. 15 where three directories and two files will be created under root '/'. Also, dir2 will have two sub directories such as dir21, and dir22, finally dir21 will have two files file21a and file21b. We run the following command

- ./slow-mkdir.exe /dir1
- ./slow-touch.exe /file1
- ./slow-mkdir.exe /dir2
- ./slow-touch.exe /file2
- ./slow-mkdir.exe /dir3
- ./slow-mkdir.exe /dir2/dir21
- ./slow-mkdir.exe /dir2/dir22
- ./slow-touch.exe /dir2/dir21/file21a
- ./slow-touch.exe /dir2/dir21/file21b

After running the above lines of command, directories and files has been created and the current inode structure is showing in Fig 15. Now, we will delete '/file1' and add '/dir0' to see inode. Fig. 16 confirms that file1 has been deleted and fig. 17 confirms that unused inode-2 is now using by dir0.
The slow-cat.exe, slow-export.exe, and slow-import.exe test scenarios are also implemented with our code and found that those are working perfectly. [Figure 18].

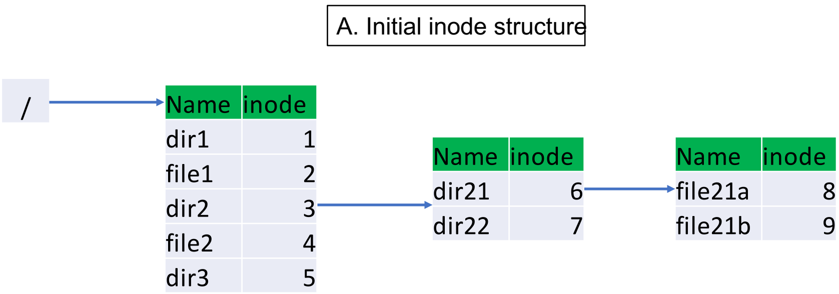
4 Project challenges

The main challenge for this project was to implement remove_inode() correctly. We encounter a problem that after removing the file or directory, the newly created object was getting assigned to the end of the dirent table of the parent directory, creating the segmentation. As a result, the inode->size does not represent the actual size of the directory list. The remove inode function is modified to overcome the segmentation. Instead of deleting the entries in the middle, the function copied the last entry into the blank position.[Figure 19].



```
(base) mamun@MDs-MacBook-Pro:~/FIU/Fall19/OS/Project2/COP5614-FinalProject$ ./simple-test.exe default-disk
file system booted from file 'default-disk'
file '/first-file' created successfully
file '/second-file' created successfully
dir '/first-dir' created successfully
dir '/first-dir/second-dir' created successfully
ERROR: can't create dir '/first-file/second-dir'
Illegal character found! *
ERROR: can't create dir '/first_dir/third*dir'
file '/first-file' unlinked successfully
ERROR: can't unlink dir '/first-dir'
dir '/first-dir/second-dir' unlinked successfully
file '/second-file' opened successfully, fd=0
successfully wrote 1024 bytes to fd=0
fd 0 closed successfully
file system sync'd to file 'default-disk'
```

Figure 14: Output of simple-test



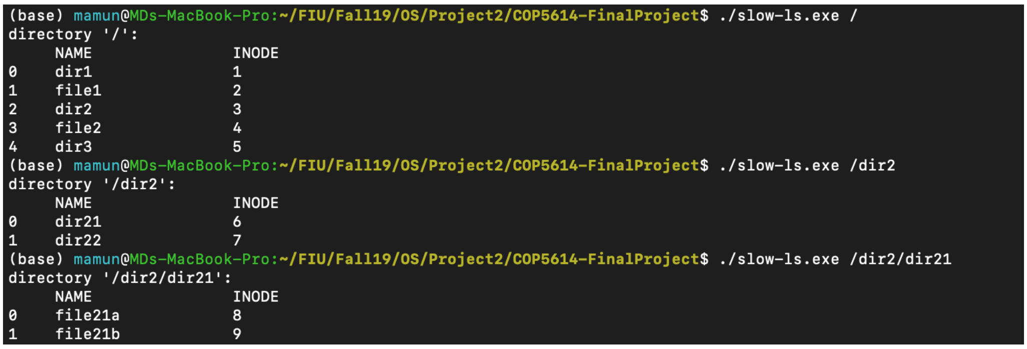
A. Initial inode structure

/

Name	inode
dir1	1
file1	2
dir2	3
file2	4
dir3	5

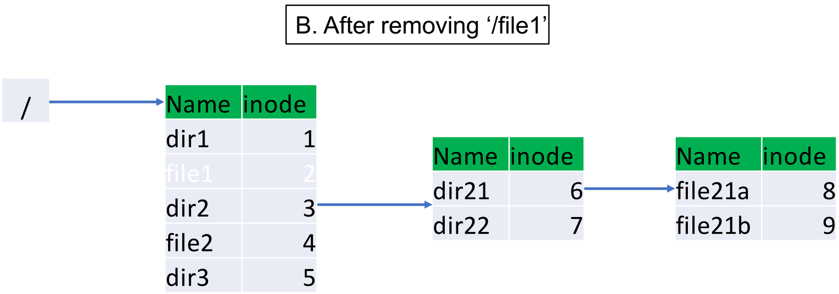
Name	inode
dir21	6
dir22	7

Name	inode
file21a	8
file21b	9



```
(base) mamun@MDs-MacBook-Pro:~/FIU/Fall19/OS/Project2/COP5614-FinalProject$ ./slow-ls.exe /
directory '/':
     NAME                INODE
0    dir1                1
1    file1               2
2    dir2                3
3    file2               4
4    dir3                5
(base) mamun@MDs-MacBook-Pro:~/FIU/Fall19/OS/Project2/COP5614-FinalProject$ ./slow-ls.exe /dir2
directory '/dir2':
     NAME                INODE
0    dir21               6
1    dir22               7
(base) mamun@MDs-MacBook-Pro:~/FIU/Fall19/OS/Project2/COP5614-FinalProject$ ./slow-ls.exe /dir2/dir21
directory '/dir2/dir21':
     NAME                INODE
0    file21a             8
1    file21b             9
```

Figure 15: Initial inode structure



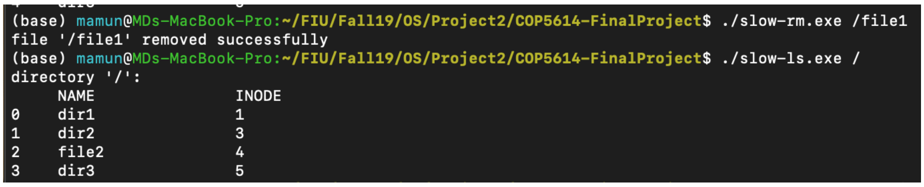
B. After removing '/file1'

/

Name	inode
dir1	1
file1	2
dir2	3
file2	4
dir3	5

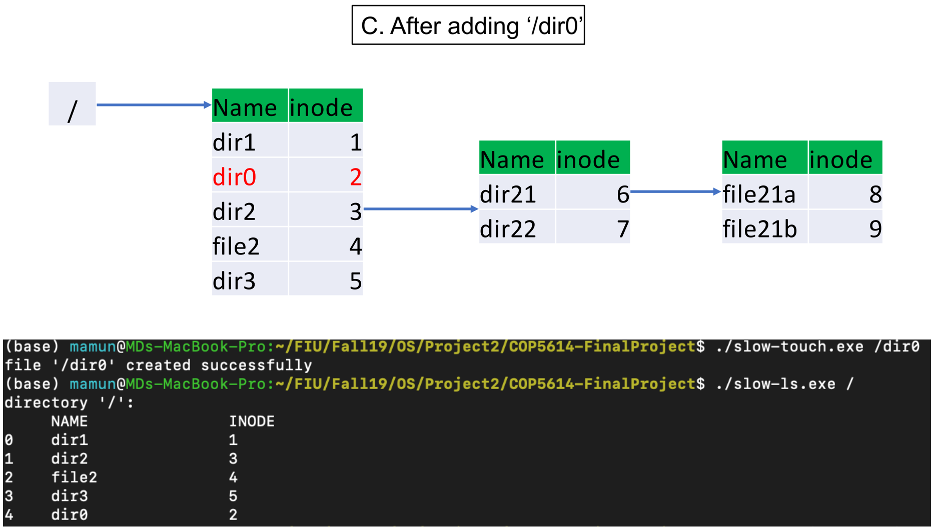
Name	inode
dir21	6
dir22	7

Name	inode
file21a	8
file21b	9



```
(base) mamun@MDs-MacBook-Pro:~/FIU/Fall19/OS/Project2/COP5614-FinalProject$ ./slow-rm.exe /file1
file '/file1' removed successfully
(base) mamun@MDs-MacBook-Pro:~/FIU/Fall19/OS/Project2/COP5614-FinalProject$ ./slow-ls.exe /
directory '/':
     NAME                INODE
0    dir1                1
1    dir2                3
2    file2               4
3    dir3                5
```

Figure 16: inode structure after removing '/file1'



C. After adding ‘/dir0’

/

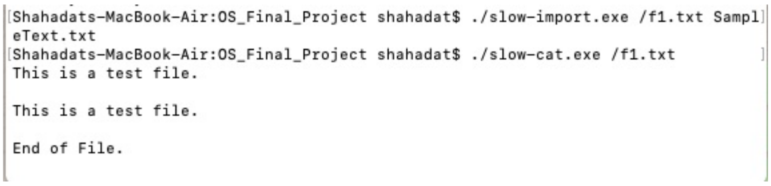
Name	inode
dir1	1
dir0	2
dir2	3
file2	4
dir3	5

Name	inode
dir21	6
dir22	7

Name	inode
file21a	8
file21b	9

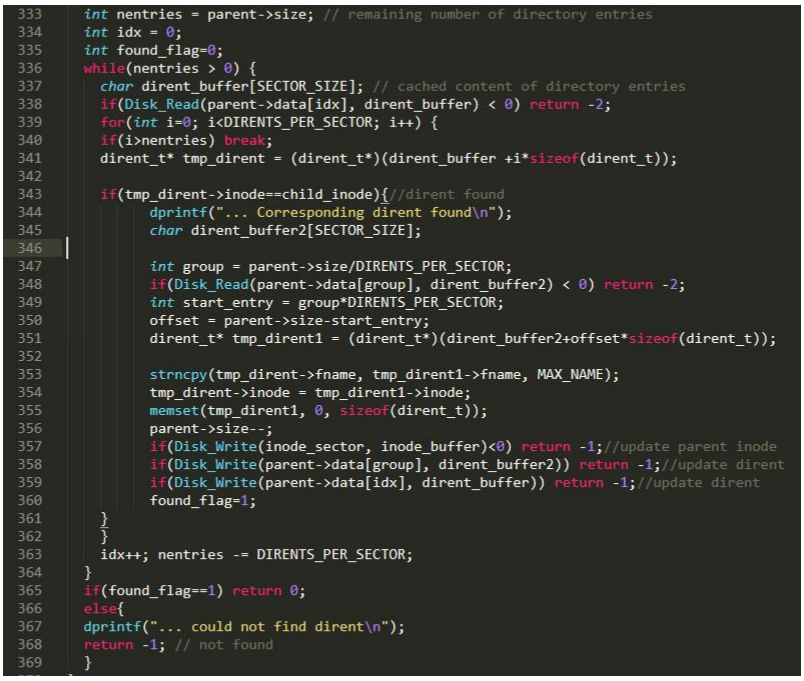
```
(base) mamun@MDs-MacBook-Pro:~/FIU/Fall19/OS/Project2/COP5614-FinalProject$ ./slow-touch.exe /dir0
file '/dir0' created successfully
(base) mamun@MDs-MacBook-Pro:~/FIU/Fall19/OS/Project2/COP5614-FinalProject$ ./slow-ls.exe /
directory '/':
     NAME                  INODE
0    dir1                  1
1    dir2                  3
2    file2                 4
3    dir3                  5
4    dir0                  2
```

Figure 17: inode structure after adding ’/dir0’



```
Shahadats-MacBook-Air:OS_Final_Project shahadat$ ./slow-import.exe /f1.txt Sampl
eText.txt
Shahadats-MacBook-Air:OS_Final_Project shahadat$ ./slow-cat.exe /f1.txt
This is a test file.

This is a test file.

End of File.
```

Figure 18: Output of slow-import and slow-cat



```
int nentries = parent->size; // remaining number of directory entries
int idx = 0;
int found_flag=0;
while(nentries > 0) {
  char dirent_buffer[SECTOR_SIZE]; // cached content of directory entries
  if(Disk_Read(parent->data[idx], dirent_buffer) < 0) return -2;
  for(int i=0; i<DIRENTS_PER_SECTOR; i++) {
  if(i>nentries) break;
  dirent_t* tmp_dirent = (dirent_t*)(dirent_buffer +i*sizeof(dirent_t));

  if(tmp_dirent->inode==child_inode){//dirent found
        dprintf("... Corresponding dirent found\n");
        char dirent_buffer2[SECTOR_SIZE];

        int group = parent->size/DIRENTS_PER_SECTOR;
        if(Disk_Read(parent->data[group], dirent_buffer2) < 0) return -2;
        int start_entry = group*DIRENTS_PER_SECTOR;
        offset = parent->size-start_entry;
        dirent_t* tmp_dirent1 = (dirent_t*)(dirent_buffer2+offset*sizeof(dirent_t));

        strncpy(tmp_dirent->fname, tmp_dirent1->fname, MAX_NAME);
        tmp_dirent->inode = tmp_dirent1->inode;
        memset(tmp_dirent1, 0, sizeof(dirent_t));
        parent->size--;
        if(Disk_Write(inode_sector, inode_buffer)<0) return -1;//update parent inode
        if(Disk_Write(parent->data[group], dirent_buffer2)) return -1;//update dirent
        if(Disk_Write(parent->data[idx], dirent_buffer)) return -1;//update dirent
        found_flag=1;
  }
  }
  idx++; nentries -= DIRENTS_PER_SECTOR;
}
if(found_flag==1) return 0;
else{
dprintf("... could not find dirent\n");
return -1; // not found
}
```

Figure 19: Modification in *Remove_Inode()* function

5 Conclusion

In this project, we have developed a file management system that can perform some actions such as read or write of a file or directory, directory list, create a new file or directory, and delete a file or directory. The whole file system is managed by a bitmap system which is also designed here. The test cases shows that the functions works perfectly.