Design Report

Introduction

This is the design report for CS611 Mp7 of file system implementation. In this assignment, I finished the file system design and implementation, file class design and implementation and also the bonus thread-safe design & implementation. The files modified in this assignment are:

```
file_system.C
file_system.H
file.C
file.H
```

File System Design

The file system manages all the files in the disk such as file creation, file deletion, file status check, memory allocation, deallocation and etc. In this design, I used the superblock to store file information such as number of files in the file system, file size of all files as well as start inode index, then I use bitmap to manage the allocation and deallocation of memory in the disk. As for files, the inodes are used to store the indexes of all data blocks, and the data blocks store the actual data for each file. The memory distribution in the disk is illustrated below:

A more detailed description of the data structure design for each region is shown below.

```
struct SuperBlock {
    unsigned int fsize;
    unsigned int inode_index;
    unsigned int files_no[63];
    unsigned int files_size[63];
    unsigned int inode_no[63];
    unsigned int inode_size[63];
};
struct I_Node {
    unsigned int file_id;
        unsigned int next_block;
    unsigned int curr_size;
    unsigned int blocks[125];
};
```

Here, the size of SuperBlock is 4 + 4 + 63*4*4 = 1016 occupying exactly two blocks, the size of one I_Node is 4 + 4 + 4 + 125*4=512 occupying one block. In the superblock, fsize is the number of files created in this file system, inode_index is the current potential free block index, files_no is the array of file number for each file, files_size is the array of file size, inode_no is the inode start number of each file, inode_size is the number of inodes for each file. Now the bitmap can be used to manage the allocation and deallocation of blocks in the disk. Here, I used 6 blocks for bitmap and the number of blocks can be managed is 3072*8=24576, the total size that can be managed is 24576*512 = 12 MB. In the I_Node struct, the I_Node holds the data block information such as index of data blocks used, and the maximum number of data blocks managed by each inode is 125. Here, the variable next_block is for next inode block if the number of data blocks required is more than 125, so one more inode block is required. The next is the file system class design and file class design.

```
bitmap[3072]
```

1. File System Class Design

   ```
   class FileSystem {
       SimpleDisk * disk;
       unsigned int size;
       unsigned char bitmap[BITMAP_SIZE];
       File* files;
       unsigned int file_size;
       unsigned int inode_index;
      }
   ```

   The file system has private variable disk, size, bitmap, files array to monitor all files creation, file_size for the current file size array.

2. File System Format
   In the file system format, the file system will clear all the data in the disk and reset the blocks in disk occupied for super blocks, bitmap, inodes and data to zero, and then update all the information in disk as well as set all data in disk to zero.

```
//super block data
memset(super_block_data, 0, SUPER_BLOCK_SIZE);
...
//bitmap data
int num_bitmap = BITMAP_SIZE/512;
for (int i = 0; i < num_bitmap; i++) {
    memset(buff, 0, 512);
    _disk->write(num+i, buff);
}
//data block
memset(buff, 0, 512);
for (int i = 0; i < num_of_blocks; i++)
_disk->write(i+start_block_no, buff);
```

3. File System Mount
   The file system mount method will mount all the previously defined file system information to the current file system. So it will read the super block data from disk and bitmap information, inode information, data block information from disk, and give this information to the current file system.

```
....
  for (int i = 0; i < num; i++) {
    disk->read(i, buff);
    memcpy(&super_block_data[i*512], buff, 512);
    }
SuperBlock* superblock = (SuperBlock*) super_block_data;
....
```

   The files array is also created and bitmap is updated.

```
    if (file_size != 0) {
    files = (File*) new File[file_size];
    }
    ....
    int num_bitmap = BITMAP_SIZE/512;
    for (int i = 0; i < num_bitmap; i++) {
    disk->read(num+i, buff);
    memcpy(&bitmap[i*512],buff, BLOCK_SIZE);
    }
```

4. Lookup File
   The look up file method simply iterate the all the files created in the file system and return the address of the file object created. Otherwise, return NULL.

```
    for (int i = 0; i < file_size; i++) {
    if (files[i].file_id  == _file_id) {
        return &files[i];
    }
    }
```

5. File Creation
   In the file creation, the file system first check the existence of a file through file lookup. If the file does not exist, it will create one. In order to create a file, the file system will first find a free block in the disk as the first inode for that file, then mark that block as used in bitmap. In the inode, it will set all the information as zero.

```
File* new_file = new File(this, disk);
new_file->file_id = _file_id;
new_file->inode_no = getFreeBlock();
updateBitmap();
//update inode block
disk->read(new_file->inode_no, buff);
I_Node* i_node = (I_Node*) buff;
......
disk->write(new_file->inode_no,buff);
//increae the total number of files by 1
File* new_files = (File*) new File[file_size + 1];
.....
file_size++;
updateDisk();
return true;
```

6. File Deletion

   In the file deletion method, this method first check the existence of the file, it not exits return false. Otherwise, the method needs to read the inode information of the files, then delete all the data blocks associated with the inode and delete inode, and then update the files array to remove files and decrease the number of files. Later on, the bitmap and disk needs to be updated.

```
    // if file not exists, then return false;
    //now start deleting, find the file number i
    unsigned int inode_no   =  files[i].inode_no;
    unsigned int inode_size =  files[i].inode_size;
    for (int j = 0; j < inode_size; j++) {
    disk->read(inode_no, buff);
    I_Node* i_node = (I_Node*) buff;
    //remove all the data in blocks according to inode
    //update bitmap for data block
    memset(buff, 0, 512);
    disk->write(inode_no, buff);
    releaseBitmap(inode_no);
    //update files array
        ......
    updateDisk();
```

File Implementation

1. File Reset
   In the file reset method, it simply set the current data block index, current inode size, current read position to the beginning of the file.

```
void File::Reset() {
    //Console::puts("reset current position in file\n");
    cur_block = 0;
    cur_size = 0;
    cur_position = 0;
}
```

2. File Rewrite
   In the file rewrite method, it will set all data in the file to zero.

```
    for (int j = 0; j < inode_size; j++) {
    disk->read(start_inode, buff);
    I_Node* i_node = (I_Node*) buff;
    memset(temp,0,512);
    // set all the data blocks in the inode to zero
    for (int k = 0; k < i_node->curr_size; k++) {
        disk->write(i_node->blocks[k], temp);
    }
    start_inode = i_node->next_block;
}
```

3. File Read
   In the file read method, it will first locate the current block and current position pointer through inode of this file. After locating the position, it will also check the EoF of this file to see if it has reached the end of the file.

```
    for (int j = 0; j < inode_size; j++) {
        disk->read(start_inode, temp1);
        .......
        //This is to find the current read block and current read position
    }
    if (EoF()) return 0;
```

   Now it starts data read from disk, and the count set to be the total number of chars to read, here it checks if the end of file has reached. Also, if the current block finishes reading and it needs to turn to another unread block, if the current inode finishes reading all the blocks because each inode only manages 125 data blocks, it will go to the next inode if it has. In the whole process, the end of file description needs to be checked all the time to make sure valid data read.

```
k = 0
disk->read(i_node->blocks[block_index], buff);
//cur_size += 1;
for (; k < count; k++, cur_position++) {
    if (EoF())  break;
    if (cur_position >= 512) {
        block_index++;
        if (block_index == 125) {
            start_inode = i_node->next_block;
            disk->read(start_inode, temp1);
            i_node = (I_Node*) temp1;
            block_index = 0;
        }
        disk->read(i_node->blocks[block_index], buff);
        cur_size += 1;
        cur_position = 0;
    }
    _buf[k] = buff[cur_position];
}
return k;
```

4. File Write
   In the file write method, it will first find the location of the end of file and its corresponding block number and position in the disk. If the end of file is the last character in the block, then new block needs to be allocated to store new data information. Here, I used a helper method **check_inode_full** to manage inode creation if current inode is full and new inode required.

```
void File::check_inode_full(unsigned int* curr_inode, I_Node* i_node) {
   //increase the inode size
    inode_size += 1;
    //get a new free block to save inode information
    i_node->next_block = file_system->getFreeBlock();
    //save the current inode because its job done
    disk->write(*curr_inode, temp1);
    //get the next inode block and write some basic information
    *curr_inode = i_node->next_block;
    disk->read(*curr_inode, temp1);
    i_node = (I_Node*) temp1;
    i_node->file_id = file_id;
    i_node->next_block = 0;
    i_node->curr_size = 0;
    memset(i_node->blocks, 0, 500);
    disk->write(*curr_inode, temp1);
}
```

The next is to check how many characters capacity left for the current write block `remaining = 512 - section_pos`. If the number of characters to write is less than the number of characters capacity of block, then it is fine because the data can be written to the disk directly without any problem.

```
if (_n <= remaining) {
    disk->read(i_node->blocks[i_node->curr_size-1], buff);
    memcpy(&buff[section_pos], _buf,_n);
    disk->write(i_node->blocks[i_node->curr_size-1], buff);
}
```

Otherwise, if the number of characters to write is larger than the remaining characters in the current disk block, then more blocks required to store data, so block allocation is required to hold more data.

```
else if (_n > remaining){
    if (remaining > 0) {
        // fill the remaining characters to disk block
        // then it will move a new empty block
    }
    count = _n - remaining; // check how many character left
    //now it will start writing to disk
    while (count > 0) {
            i_node->curr_size += 1;
            new_block_no = file_system->getFreeBlock();
            i_node->blocks[i_node->curr_size-1] = new_block_no;
            if (count < 512) {
                //work is done, set count = 0
            } else {
                //write all data to one block and set count-512
            }
            //now check if the current inode is full or not
            if (i_node->curr_size == 125) {
                if (count > 0) // more data to read
                    check_inode_full();
                     //get new free inode block and data block
                else
                    disk->write(cur_inode, temp1)
                    //if no data to read, then finish
            }
    }
    // add number _n to the file size variable
    file_size += _n;
```

Bonus

In the thread-safe file system, we can assume there are multiple threads accessing file systems and files. Here, thread might access the files to read, write, delete files. In order to make it thread-safe, I used the lock array to control read, write, delete for all files, and the thread access uses the busy-waiting method, namely, the current thread will wait until the lock is released prior to any operation. In the current interface, file lookup operation needs to be performed before any read and write or delete operation. I used a boolean array lock in both File System class and file class because file system managed file creation and deletion while file class manages file read and write, and usually file creation has no problem but file deletion might cause problem when one thread try to read or write but the other thread want to delete files. My design is simple because I am not able to use more sophisticated mutex lock from library which are much more powerful, so I disabled the interrupt for all the thread. In this case, the file operation will be atomically performed without any interference from outside interrupt.

a. File System access
In the file system, I used the lock array to assign locks to every file created in this file system, so I added one boolean array variable

`bool *file_locks`

for lock of every file in the file system. Now, we will have methods to update locks, check lock status, lock, unlock and even delete locks for particular file. The locks are associated with the status of file, so if no file exists, then no locks required for that file.

```
//thread-safe design for specific file
bool lock(int file_id); // lock file with file id file_id
bool unlock(int file_id); //unlock file with file id file_id
bool is_locked(int index); // check the file array index lock status
lock with index i
void update_add_locks();//add lock to the lock array
void update_delete_locks(int i);//delete specific lock
```

With this helper method, now the lock and unlock file can be performed in the file system for creation and deletion. More specifically,

```
bool FileSystem::CreateFile(int _file_id) {
    ......
    //add locks
    update_add_locks();
    file_size++;
    .....
}
```

The same operation can be performed in file deletion. Here, we want to make deletion thread safe, namely, we do not want to delete files when some thread try to read or write files. So I used the simple busy-waiting approach, it will first check the lock and wait for the lock release if lock is locked.

```
bool FileSystem::DeleteFile(int _file_id) {
    .......
   while(is_locked(_file_id));
   lock(_file_id);
    .......
    //file operations
   //delete locks
    update_delete_locks(i);
    file_size--;
    ........
}
```

In the `update_add_locks()` and `update_delete_locks()` method, new lock for new file will be created and old lock for old file will be deleted and the size of locks array will be changed dynamically also.

b. File access
In the file access, locks has to be checked before any read or write operation, so in any file access, busy-waiting method is employed for read or write, and it release the lock after completion of file operation.

```
int File::Read(unsigned int _n, char * _buf) {
    //lock first
    while(file_system->is_locked(file_id));
    ......//perform read operation
    file_system->unlock(file_id);
}
```

In order not to conflict with other thread for the **reset()** method in case one thread reset the current block or current position while other thread is reading data, thus the lock is also applied to the reset method.

```
void File::Reset() {
    //lock first
    while(file_system->is_locked(file_id));
    file_system->lock(file_id);
    cur_block = 0;
    cur_size = 0;
    cur_position = 0;
    file_system->unlock(file_id);
}
```

In the file write method, we have the same strategy to lock first.

```
void File::Write(unsigned int _n, const char * _buf) {
    //lock first
    while(file_system->is_locked(file_id));
    file_system->lock(file_id);
    .....//perform write operation
    file_system->unlock(file_id);
}
```

The above lock based file system design is simple and use only one lock to control read, write and delete operation to make sure these operation can not be performed simultaneously. The lock status needs to be checked before any operation, and if the current lock is locked, it needs to wait until lock is released.