CSCE 611: Operating Systems
MP7: Design Document

Shweta Sharma
UIN: 433003780

Design Idea:

Vanilla File System – MP7:

I have implemented the following:

- Vanilla File System
- Option 1/2: Design and implementation of an extended file system for files up to 64kB long

Vanilla File System:

A very simple file system is implemented with the following assumptions as mentioned in the handout:

- The file system can only manage single-level directory.
- Length of any file is at most one block. (For the compulsory portion)
- File operations are separate from the File System functions.

The various components of the file system are described as follows:

1. **Inode**:

```
class Inode
{
  friend class FileSystem; // The inode is in an uncomfortable position between
  friend class File;       // File System and File. We give both full access
                           // to the Inode.

private:
  long id; // File "name"
  unsigned int blk_number;
  bool inode_is_free;
  int fle_size;


  /* You will need additional information in the inode, such as allocation
     information. */

  FileSystem *fs; // It may be handy to have a pointer to the File system.
                  // For example when you need a new block or when you want
                  // to load or save the inode list. (Depends on your
                  // implementation.)

  /* You may need a few additional functions to help read and store the
     inodes from and to disk. */
```

- id - points to the identifier of the file it holds information about

- blk_number - block in the file system where the file pointed by the inode is stored
- inode_is_free - checks if the current inode available or holding any file
- fle_size - size of file pointed by the inode
- fs - pointer to the file system

2. **File:**

Variables:

- fle_system – pointer to the file system
- fle_identifier – file identifier
- blk_number – block in the file system where the file is stored
- inode_indx – position in inode array assigned to file
- fle_size – size of file
- curr_pos – current position pointed to in file (for read and write operations)
- block_cache – buffer for file, acting as a cache, of size 512 bytes

```
class File {

    friend class FileSystem;

private:
    /* -- your file data structures here ... */

      FileSystem* fle_system;
      int fle_identifier;
      unsigned int blk_number;
      unsigned int inode_indx;
      unsigned int fle_size;
      unsigned int curr_pos;

    /* You will need a reference to the inode, maybe even a reference to the
       file system.
       You may also want a current position, which indicates which position in
       the file you will read or write next. */

    unsigned char block_cache[SimpleDisk::BLOCK_SIZE];
    /* It will be helpful to have a cached copy of the block that you are reading
       from and writing to. In the base submission, files have only one block, which
       you can cache here. You read the data from disk to cache whenever you open the
       file, and you write the data to disk whenever you close the file.
    */
```

Functions:

- File(_fs, _id) - Opens the file with file id _id using the file system _fs. Sets the ‘curr_pos’ to the beginning of the file. It loops through the list of inodes to retrieve the inode for the file. Once it has found the inode, it sets other variables node_indx, blk_number, file_size etc. for the particular file created and then breaks the loop.

```
File::File(FileSystem *_fs, int _id) {
    Console::puts("Opening file with id");Console::puti(_id);Console::puts(".\n");
    fle_system = _fs;
    fle_identifier = _id;
    curr_pos = 0;

    bool fle_blk_found = false;
    unsigned int i=0;
    while(i<fle_system->MAX_INODES){
        if(fle_system->inodes[i].id==fle_identifier){
            inode_indx = i;
            blk_number = fle_system->inodes[i].blk_number;
            fle_size = fle_system->inodes[i].fle_size;
            fle_blk_found = true;
            break;
        }
        i++;
    }

    assert(fle_blk_found);
}
```

- ~File() - Closes the currently open file and deletes data structures associated with it. It writes all the caches data onto the disk and updates the inode in the list of inodes.

```
File::~File() {
    Console::puts("Closing file.\n");
    /* Make sure that you write any cached data to disk. */
    /* Also make sure that the inode in the inode list is updated. */

    fle_system->disk->write(blk_number, block_cache);
    fle_system->inodes[inode_indx].fle_size = fle_size;

    Inode* tmp_inodes_ref = fle_system->inodes;
    unsigned char* tmp_inode_ref = (unsigned char*) tmp_inodes_ref;
    fle_system->disk->write(1,tmp_inode_ref);
}
```

- Read(unsigned int _n, char * _buf) – Reads _n characters from the file/block_cache into the buffer _buf, unless EOF is reached and returns actual characters read from file.

```
int File::Read(unsigned int _n, char *_buf) {
    Console::puts("reading from file\n");
    int char_cnt = 0;
    for(int indx = curr_pos; indx<fle_size;indx++){
        if(char_cnt==_n)
            break;
        _buf[char_cnt++] = block_cache[indx];
    }

    Console::puts("reading from file complete.\n");
    return char_cnt;
}
```

- Write(unsigned int _n, char * _buf) – Writes _n characters from the file/block_cache into the buffer _buf, unless EOF is reached and returns actual characters written to file.

```
int File::Write(unsigned int _n, const char *_buf) {
    Console::puts("writing to open file.\n");
    int char_cnt = 0;
    int end_indx = curr_pos+_n;
    while(curr_pos<end_indx){
        if(curr_pos==SimpleDisk::BLOCK_SIZE){
            Console::puts("EOF reached while writing.\n");
            break;
        }
        block_cache[curr_pos++]=_buf[char_cnt++];
        fle_size++;
    }

    Console::puts("writing to file complete.\n");
    return char_cnt;
}
```

- Reset() - Sets the 'curr_pos' to the beginning of the file.

```
void File::Reset() {
    Console::puts("resetting file\n");
    curr_pos=0;
}
```

- EoF() – Checks if the end of file is reached by 'curr_pos'.

```
bool File::EoF() {
    Console::puts("checking for EoF\n");
    return (curr_pos<fle_size);
}
```

3. File System:

Variables:

- free_blk_cnt – count of free blocks in the file system
- inode_cntr – count of free inodes in the file system
- free_blocks –pointer to bitmap of free blocks in the file system. 'F' represents a free block and 'U' represents a used block
- disk – pointer to the SimpleDisk ecosystem
- inodes – pointer to array of inodes in the file system

```
class FileSystem
{

  friend class Inode;

private:
  /* -- DEFINE YOUR FILE SYSTEM DATA STRUCTURES HERE. */
  unsigned int size;
  unsigned int free_blk_cnt;
  unsigned int inode_cntr;

  unsigned char *free_blocks;
  /* The free-block list. You may want to implement the "list" as a bitmap.
     Feel free to use an unsigned char to represent whether a block is free or not;
     no need to go to bits if you don't want to.
     If you reserve one block to store the "free list", you can handle a file system up to
     256kB. (Large enough as a proof of concept.) */

  // short GetFreeInode();
  // int GetFreeBlock();
  /* It may be helpful to two functions to hand out free inodes in the inode list and free
     blocks. These functions also come useful to class Inode and File. */

public:

  SimpleDisk *disk;
  static constexpr unsigned int MAX_INODES = SimpleDisk::BLOCK_SIZE / sizeof(Inode);
  /* Just as an example, you can store MAX_INODES in a single INODES block */

  Inode *inodes; // the inode list
  /* The inode list */
```

Functions:

- FileSystem() – Initialized the File System local data structures and inode.

```
FileSystem::FileSystem() {
    Console::puts("In file system constructor.\n");
    disk = NULL;
    size = 0;
    free_blk_cnt = SimpleDisk::BLOCK_SIZE / sizeof(unsigned char);
    inode_cntr = 0;

    free_blocks = new unsigned char[free_blk_cnt];
    for(unsigned int i=0;i<free_blk_cnt;i++){
        free_blocks[i] = FREE_BLK_REP;
    }
    inodes = new Inode[MAX_INODES];

    Console::puts("FileSystem constructor initialized.\n");
}
```

- ~FileSystem() – Writes the local data structures and inode to the disk and unmounts the FileSystem. The first two blocks are reserved for storing ‘free_blocks’ and ‘inodes’ respectively.

```
FileSystem::~FileSystem() {
    Console::puts("unmounting file system\n");
    /* Make sure that the inode list and the free list are saved. */
       disk->write(BITMAP_BLK_NUMBER, free_blocks);
    unsigned char* tmp_inode_ref = (unsigned char*) inodes;
    disk->read(INODE_BLK_NUMBER, tmp_inode_ref);

    Console::puts("unmounted file system.\n");
}
```

- Mount(disk) – Mounts the file system from the disk and loads the data structures

and inode. It then iterates through the inodes list and finds out the total number of inodes that are free.

```
bool FileSystem::Mount(SimpleDisk * _disk) {
    Console::puts("mounting file system from disk\n");

    /* Here you read the inode list and the free list into memory */

    disk = _disk;
    unsigned char* tmp_inode_ref;

    disk->read(BITMAP_BLK_NUMBER, free_blocks);
    disk->read(INODE_BLK_NUMBER, tmp_inode_ref);

    inodes = (Inode *) tmp_inode_ref;

    // finding a free inode
    inode_cntr = 0;
    unsigned int i = 0;
    while(i<MAX_INODES){
        if(!inodes[i].inode_is_free){
            inode_cntr++;
        }
        i++;
    }

    free_blk_cnt = SimpleDisk::BLOCK_SIZE / sizeof(unsigned char);

    Console::puts("Mounted file system from disk.\n");

    return true;
}
```

- Format(disk, _size) – Formats the entire disk (of _size) and re-initiates the local data structures and inode.

```
bool FileSystem::Format(SimpleDisk * _disk, unsigned int _size) { // static!
    Console::puts("formatting disk\n");
    /* Here you populate the disk with an initialized (probably empty) inode list
       and a free list. Make sure that blocks used for the inodes and for the free list
       are marked as used, otherwise they may get overwritten. */
    unsigned int n_free_blks  = _size/SimpleDisk::BLOCK_SIZE;
    unsigned char* free_blks_arr = new unsigned char[n_free_blks];

    // reserved blocks
    free_blks_arr[0] = free_blks_arr[1] = USED_BLK_REP;

    //freeing blocks
    unsigned int i = 2;
    while(i<n_free_blks){
        free_blks_arr[i] = FREE_BLK_REP;
        i++;
    }

    _disk->write(BITMAP_BLK_NUMBER, free_blks_arr);

    Inode* tmp_inodes_ref = new Inode[MAX_INODES];
    unsigned char* tmp_inode_ref = (unsigned char*) tmp_inodes_ref;
    _disk->write(INODE_BLK_NUMBER,tmp_inode_ref);

    Console::puts("formatted disk.\n");
    return true;
}
```

- LookupFile(_file_id) – If the file with identifier _file_id is found in the file system, returns its associated inode otherwise throws an exception.

```
Inode * FileSystem::LookupFile(int _file_id) {
    Console::puts("looking up file with id = "); Console::puti(_file_id); Console::puts("\n");
    /* Here you go through the inode list to find the file. */
    unsigned int i;
    for(i=0;i<inode_cntr;i++){
        if(inodes[i].id== _file_id){
            return &inodes[i];
        }
    }
    Console::puts("File with id = "); Console::puti(_file_id); Console::puts(" not found.\n");
    return NULL;
}
```

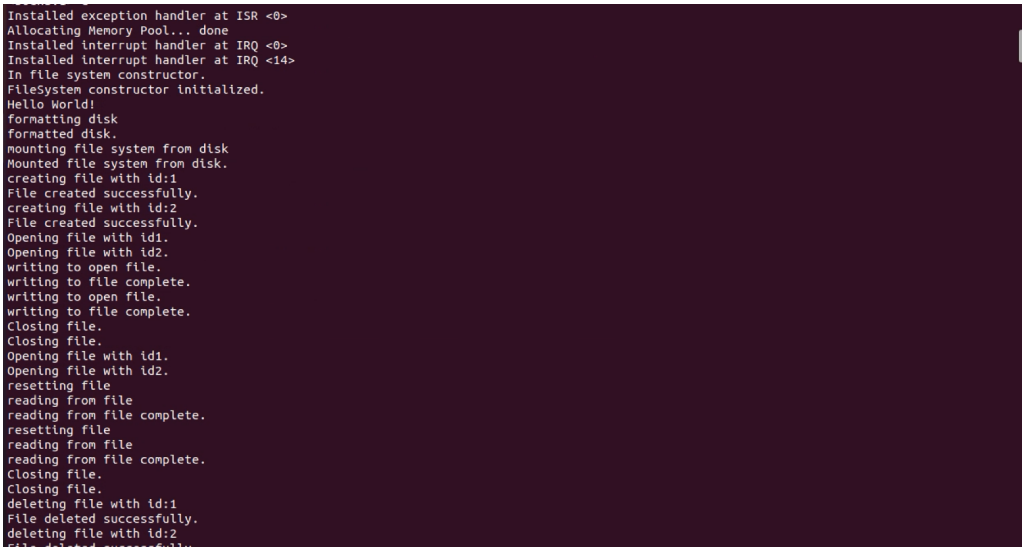
- CreateFile(_file_id) – Creates a new file in the system with identifier _file_id by initializing a new inode and assigning a free block.

```
bool FileSystem::CreateFile(int _file_id) {
    Console::puts("creating file with id:"); Console::puti(_file_id); Console::puts("\n");
    /* Here you check if the file exists already. If so, throw an error.
       Then get yourself a free inode and initialize all the data needed for the
       new file. After this function there will be a new file on disk. */
    unsigned int i;
    for(i=0;i<inode_cntr;i++){
        if(inodes[i].id== _file_id){
            Console::puts("Error: File with id: ");Console::puti(_file_id); Console::puts(" already exists.\n");
            assert(false);
        }
    }


    unsigned int free_inode_indx = -1;
    for(i=0;i<MAX_INODES;i++){
        // Console::puti(inodes[i].inode_is_free);Console::puts(", ");
        if(inodes[i].inode_is_free){
            free_inode_indx = i;
            // Console::puts("\n free_inode_indx: ");Console::puti(free_inode_indx);Console::puts("\n");
            break;
        }
    }
    // Console::puts("Prininting free_blocks: ");
    unsigned int free_blk_indx = -1;
    for(i=0;i<free_blk_cnt;i++){
        if(free_blocks[i] == FREE_BLK_REP){
            // Console::puts("\n free_blocks: ");Console::putch(free_blocks[i]);Console::puts(",");
            free_blk_indx = i;
            break;
        }
    }
    // Console::puts("\n");

    assert((free_inode_indx!=-1) && (free_blk_indx!=-1));
    inodes[free_inode_indx].inode_is_free = false;
    inodes[free_inode_indx].fs = this;
    inodes[free_inode_indx].id = _file_id;
    inodes[free_inode_indx].blk_number = free_blk_indx;
    free_blocks[free_inode_indx] = USED_BLK_REP;

    Console::puts("File created successfully.\n");

    return true;
}
```

- DeleteFile(_file_id) – Deletes the file identified by _file_id and its data structures. Also, removes the associated inode and free the block it was stored at.

```
bool FileSystem::DeleteFile(int _file_id) {
    Console::puts("deleting file with id:"); Console::puti(_file_id); Console::puts("\n");
    /* First, check if the file exists. If not, throw an error.
       Then free all blocks that belong to the file and delete/invalidate
       (depending on your implementation of the inode list) the inode. */

    bool file_exists = false;
    unsigned int indx=0;

    while(indx<MAX_INODES){
        if(inodes[indx].id ==_file_id){
            file_exists = true;
            break;
        }
        indx++;
    }

    assert(file_exists);

    int blk_number = inodes[indx].blk_number;
    free_blocks[blk_number] = FREE_BLK_REP;

    inodes[indx].inode_is_free = true;
    inodes[indx].fle_size = 0;

    Console::puts("File deleted successfully.\n");
    return true;
}
```

Test Screenshots / Output:



Option 1/2: Design and implementation of an extended file system for files up to 64kB long

To accommodate longer files, 1 block per file is not enough. For files 64kB long, we need 64kB/512b = 12 blocks. So instead of just storing the 'blk_number', we now maintain a blocks list in the inode and file, that stores all the blocks where the file is stored orderly.

*unsigned int *blocks*

Example: To store files of size 1.5kB, we need 3 blocks. Suppose they are stored at block numbers 23, 29, 58. Then the array 'blocks' contain [23,29,58].

To cope with multiple blocks, following changes need to be done:

1. Inode:
    - Replace 'blk_number' with 'blocks' to store all the blocks where the file is stored orderly.
2. File:
    - Replace 'blk_number' with 'blocks' to store all the blocks where the file is stored orderly.
    - Read – If the end of the current block is reached when reading from the file, go to the next block in the 'blocks' list and continue reading unless the last block is reached.
    - Write - If the end of the current block is reached when writing to the file, assign a new free block, if file size is under 64kB, and add it to the 'blocks' list. Then, continue writing to the file.
    - EOF- return true if the end of the last block in 'blocks' list is reached, otherwise false.
3. FileSystem:
    - Format – When formatting the disk, all the 'blocks' allocated to the file are freed.
    - CreateFile – Instead of storing the 'blk_number', the newly assigned free block is added to the 'block' list.
    - DeleteFile – Apart from removing the data structures associated with the file, all the 'blocks' allocated to the file are also freed.

Documents changed:

I have changed the following files:

- **file.C**
- **file.H**
- **file_system.C**
- **file_system.H**