MP7: Vanilla File System

Geetesh Challur
UIN: 834006606
CSCE611: Operating System

Assigned Tasks

Main: Completed.
Bonus Option 1 DESIGN: Completed.
Bonus Option 2 IMPLEMENTATION: Completed.

System Design

- Two separate classes, FileSystem and File, are defined to implement and manage vanilla file system operations.
- An inode structure is defined by using an Inode class. The first block in the file is used for free list and second block for inode list.
- Functions like mounting the file on the disk, creating a new file, deleting a file etc are written in FileSystem class.
- Functions like reading a file, writing a file etc are written in the File class.

Code Description

file_system.H: : Variables are defined in inode class to keep track of the status of inode list. Similar data structures are defined in file system class to manage the file system. All the variables are declared in private mode since each classes are friend classes to others, giving full access to private data types.

```
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"
  bool is_inode_free;
  unsigned int block_number;
  int file_system_size;


//   int blocks[]; // for storing large files we need a array of blocks
// int *blocks;


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

  FileSystem *file_system; // 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. */
public:
   Inode();

};
```

```
class FileSystem
{

  friend class Inode;
  friend class File;

private:
  /* -- DEFINE YOUR FILE SYSTEM DATA STRUCTURES HERE. */

  SimpleDisk *disk;
  unsigned int size;

  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 */

  unsigned int free_count;
  unsigned int inode_count;
  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. */
```

file_system.C: Inode, constructor, destructor : In Inode constructor and filesystem constructor, all the class variables are initialized. In filesystem destructor the filesystem unmounted from the disk.

```
Inode::Inode()
{
  file_system = NULL;
  is_inode_free = true;
  file_system_size= 0;
  id = -1;
}

/*--------------------------------------------------------------------------*/

/* You may need to add a few functions, for example to help read and store
   inodes from and to disk. */

/*--------------------------------------------------------------------------*/
/* CLASS FileSystem */
/*--------------------------------------------------------------------------*/

/*--------------------------------------------------------------------------*/
/* CONSTRUCTOR */
/*--------------------------------------------------------------------------*/

FileSystem::FileSystem() {
    Console::puts("File System Constructor: Initializing the data structures\n");
    disk = NULL;
    size = 0;
    free_count = SimpleDisk::BLOCK_SIZE;
    free_blocks = new unsigned char[free_count];
    inode_count= 0;
    inodes = new Inode[MAX_INODES];
}

FileSystem::~FileSystem() {
    Console::puts("File System Destructor: Unmounting the file system\n");
    disk->write(0,free_blocks);
    disk->write(1,(unsigned char*) inodes);
}
```

file_system.C: Mount : The following are done in this function step by step

- Reading the first two blocks of disk which contains free blocks and inodes
- Mounting the inodes and setting the inode count.

file_system.C: Format : The following are done in this function step by step

- Resetting the status of the blocks to format the disk
- Creating a new inode array and writing it to the disk

```
bool FileSystem::Mount(SimpleDisk * _disk) {
    Console::puts("Mounting file system onto the disk\n");
    inode_count= 0;
    unsigned char* temp;
    // Reading the first two blocks of disk which contains free blocks and inodes
    disk = _disk;
    _disk->read(0,free_blocks);
    _disk->read(1,temp);
    // Mounting the inodes and setting the inode count
    inodes = (Inode *) temp;
    int i = 0;
    while(i<MAX_INODES&&!inodes[i].is_inode_free){
        i++;
    }
    inode_count = i;

    return true;
}

bool FileSystem::Format(SimpleDisk * _disk, unsigned int _size) { // static!
    Console::puts("Formatting disk\n");
    unsigned char* array = new unsigned char[(_size/SimpleDisk::BLOCK_SIZE)];

    //Resetting the status of the blocks to format the disk
    array[0] = '0';
    array[1] = '0';
    int i =2;
    while(i<(_size/SimpleDisk::BLOCK_SIZE)){
      array[i]='1';
      i++;
    }
    _disk->write(0,array);

    //Creating a new inode array and writing it to the disk
    Inode* temp = new Inode[MAX_INODES];
    unsigned char* temp1 = (unsigned char*) temp;
    _disk->write(1,temp1);
    return true;
}
```

file_system.C: LookupFile : The following are done in this function step by step

- Looking for the file in the inode list
- If the file is not present, assert false

file_system.C: CreateFile : The following are done in this function step by step

- Finding the first free block and inode
- Setting the values of the inode and block according to new created file

```
Inode * FileSystem::LookupFile(int _file_id) {
    Console::puts("Looking up file "); Console::puti(_file_id); Console::puts("\n");

    // Looking for the file in the inode list
    for (int i = 0;i++;i<inode_count){
      if (inodes[i].id==_file_id){
        return &inodes[i];
      }
    }

    //If the file is not present
    assert(false);

}

bool FileSystem::CreateFile(int _file_id) {
    Console::puts("Creating file "); Console::puti(_file_id); Console::puts("\n");



    int free_inode_idx;
    int free_block_idx;

    //Finding the first free block and inode
    int i = 0;
    while(i<free_count){
        if (free_blocks[i]=='1'){
            free_block_idx = i;
            break;
        }
        i++;
    }
    i = 0;
    while(i<MAX_INODES){
        if (inodes[i].is_inode_free){
            free_inode_idx = i;
            break;
        }
        i++;
    }


    //Setting the values of the inode and block according to new created file
    inodes[free_inode_idx].file_system = this;
    inodes[free_inode_idx].id = _file_id;
    inodes[free_inode_idx].is_inode_free = false;
    free_blocks[free_block_idx] = '0';
    inodes[free_inode_idx].block_number = free_block_idx;
```

file_system.C: DeleteFile : The following are done in this function step by step

- Checking if the file is present in the inode list
- setting found value to true if file_id is found
- If the file is not present, assert false
- Deleting the file by setting the inode and block values to free

```
bool FileSystem::DeleteFile(int _file_id) {
    Console::puts("Deleting file "); Console::puti(_file_id); Console::puts("\n");
    int d_inode_index=-1;
    int i = 0;
    //Checking if the file is present in the inode list
    while(i<MAX_INODES)
    {
      if (inodes[i].id==_file_id){
        d_inode_index = i;
        //setting found value to true if file_id is found
        break;
      }
      i++;
    }
    //If the file is not present
    assert(d_inode_index!=-1);

    //Deleting the file by setting the inode and block values to free
    int block_no = inodes[d_inode_index].block_number;
    free_blocks[block_no] = '1';
    inodes[d_inode_index].file_system_size = 0;
    inodes[d_inode_index].is_inode_free = true;

    //int blocks = inodes[free_inode_idx].file_system;
    //int i =0;
    //while(blocks)
    //{
    //    inodes[free_inode_idx].blocks[0] = free_block_idx;
    //    blocks = blocks/64;
    //    i++;
    //}

    return true;
}
```

file.H: : The following variables in the picture are defined in the private to keep track and manage the files.

```
private:
    /* -- your file data structures here ... */
      unsigned int block_number;
      unsigned int inode_idx;
      int file_id;
      unsigned int file_system_size;
      unsigned int current_position;
      FileSystem *filesystem;

    /* 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.
    */

public:
```

file.C: : file.C is modified.

- Constructor initializes the file variables. Finds the inode for the file.
- Destructor closes the file.
- Read function reads from the file and return the number of bytes read.
- Write function writes to the file and return the number of bytes written.
- Reset function resetts the current_position variable to 0.
- EoF checks if the current position is at the end of the file.

```cpp
int File::Read(unsigned int _n, char *_buf) {
    Console::puts("Reading from file\n");
    // Read from the file and return the number of bytes read
    int count = 0;
    int i = current_position;
    while(i < this->file_system_size){
        if (count==_n){
            break;
        }
        _buf[count]=block_cache[i];
        count++;
        i++;
    }
    return count;
}

int File::Write(unsigned int _n, const char *_buf) {
    Console::puts("writing to file\n");
    // Write to the file and return the number of bytes written
    int count = 0;
    int i = current_position;
    while(i - current_position < _n){
        if (i == SimpleDisk::BLOCK_SIZE){
            break;
        }

        block_cache[current_position] = _buf[count];
        count++;
        current_position++;
        this->file_system_size++;
        i++;
    }
    return count;
}

void File::Reset() {
    // Reset the current position to the beginning of the file
    current_position = 0;
}

bool File::EoF() {
    Console::puts("checking for EoF\n");
    // Check if the current position is at the end of the file
    if(current_position >= this->file_system_size)
    {
        return true;
    }
    return false;
}
```

Testing

In the first picture, we can see that the formatting and the mounting of the disc are happening properly.

```
Allocating Memory Pool... done
Installed interrupt handler at IRQ <0>
Installed interrupt handler at IRQ <14>
File System Constructor: Initializing the data structures
Hello World!
Formatting disk
Mounting file system onto the disk
```

Later when the exercise_file_system function is implemented repeatedly in an infinite loop, file 1 and file 2 are created, data is written to both files, both files are closed, data is read from both files checked if it's correct, and the both files are delete. We can see that all these operations are executed seamless in the figure below

```
Deleting file 2
Creating file 1
Creating file 2
File Constructor: Opening the file.
File Constructor: Opening the file.
writing to file
writing to file
File Destructor: Closing the file.
File Destructor: Closing the file.
File Constructor: Opening the file.
File Constructor: Opening the file.
Reading from file
Reading from file
File Destructor: Closing the file.
File Destructor: Closing the file.
Deleting file 1
Deleting file 2
Creating file 1
Creating file 2
File Constructor: Opening the file.
File Constructor: Opening the file.
writing to file
writing to file
File Destructor: Closing the file.
File Destructor: Closing the file.
File Constructor: Opening the file.
File Constructor: Opening the file.
Reading from file
Reading from file
File Destructor: Closing the file.
File Destructor: Closing the file.
Deleting file 1
Deleting file 2
```

Bonus Option 1: DESIGN

- To handle larger file sizes that are up to 64 KB long, we have to use multiple blocks to store the state of the file system instead if single block.
- To implement this, we can a define a new array **int* blocks** for each block with an **unsigned int block_number** to store the filled index of the array.
- While creating a new file or deleting a new file, all these blocks have initialized or reset according to the file size. This can be done in the **CreateFile** and **DeleteFile** functions in the FileSystem class.
- While reading and writing through a file, the entire block list has to be iterated to do this operation. End of the file should be checked in the last block of the blocks array.

Bonus Option 2: IMPLEMENTATION

‘

- To test the code for this bonus option uncomment the define _BONUS_OPTION in the kernel.C file.
- As shown below, the output is as expected even when this bonus option is handled.