CSCE611: Operating System
MP7: Vanilla File System

Name: Rahul Ravi Kadam
UIN: 834005554

Assigned Tasks

Main: Completed.

Bonus Option 1: Attempted (At the end of this design document)

Bonus Option 2: Did not attempt.

System Design

The goal of this machine problem is to implement a simple file system. Files support sequential access only.

The file system consists of files with a maximum size of one block i.e., 512 bytes. Block 0 used for storing information related to the inode, also called as Inodes block. Block 1 is used for storing a bitmap of free blocks, also called as FreeList block.

The free-block list is implemented as an array of characters. The inode list is implemented as an array of Inode class objects. Inode class object contains information such as the File ID, block number where the file is stored and the size of file.

When a file is created, a free block is located and assigned to the file. The block number assigned to the file is stored in the inode. The block number is the entire allocation information for the file since all files are at most one block long.

Functions were implemented under the FileSystem class to read and write inode list and free-block list from and to disk. The functions are listed below: (Function descriptions provided later in the document)

- read_inode_block_from_disk()
- write_inode_block_to_disk()
- read_freelist_block_from_disk()
- write_freelist_block_to_disk()

List of files modified

The following files were modified:

1. file_system.H
2. file_system.C
3. file.H
4. file.C

Code Description

1. file_system.H : class Inode:

In class Inode, data structures and functions for managing the inodes are declared. The data structures added are given below:

- unsigned long block_no - Variable to track block number where file is stored
- unsigned long size - Variable to track file size

```
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"

  /* You will need additional information in the inode, such as allocation
     information. */
  unsigned long block_no;
  unsigned long size;

  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. */
};
```

2. file_system.H : class FileSystem:

In class FileSystem, data structures and functions for managing the filesystem are declared. The following changes were made in this class:

- Uncommented functions ‘GetFreeInode()’ and ‘GetFreeBlock()’ - used to return the index of a free block or free inode if available

```
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. */
```

- Added functions to read / write freelist and inode blocks from the disk. Added functions to read data from disk into given buffer and write buffered data into disk.

```
    void read_inode_block_from_disk();

    void write_inode_block_to_disk();

    void read_freelist_block_from_disk();

    void write_freelist_block_to_disk();

    void write_block_to_disk( unsigned long block_number, unsigned char * buffer );

    void read_block_from_disk( unsigned long block_number, unsigned char * buffer );
```

3. file_system.C : FileSystem():

This method is the constructor for class FileSystem. The ‘inodes’ and ‘free_blocks’ data structures are initialized. Memory for the inode list and free block list are allocated.

```
FileSystem::FileSystem()
{
    Console::puts("In file system constructor.\n");
    inodes = new Inode [DISK_BLOCK_SIZE];
    free_blocks = new unsigned char [DISK_BLOCK_SIZE];
}
```

4. file_system.C : ~FileSystem():

This method is the destructor for class FileSystem. The inodelist and freelist blocks are written to disk. The memory for 'inodes' and 'free_block' data structure are deallocated using 'delete' operation.

```
FileSystem::~FileSystem()
{
    Console::puts("unmounting file system\n");
    /* Make sure that the inode list and the free list are saved. */

    write_inode_block_to_disk();
    write_freelist_block_to_disk();

    delete []inodes;
    delete []free_blocks;
}
```

5. file_system.C : GetFreeBlock ():

This method is used to iterate through the list of free blocks; check and return the index of the free block, if it is available. Otherwise, it returns the 'END_INDICATOR', i.e. -1.

```
int FileSystem::GetFreeBlock()
{
    unsigned int index = 0;
    for(index = 0; index < DISK_BLOCK_SIZE; index++)
    {
        if( free_blocks[index] == 0 )
        {
            // Return free block number
            return index;
        }
    }

    // Free block not available, return -1
    return END_INDICATOR;
}
```

6. file_system.C : GetFreeInode ():

This method is used to iterate through the list of inode blocks; check and return the index of an unused inode block, if it is available. Otherwise, it returns the 'END_INDICATOR', i.e. -1.

```
short FileSystem::GetFreeInode()
{
    unsigned int index = 0;
    for(index = 0; index < MAX_INODES; index++)
    {
        if( inodes[index].id == END_INDICATOR )
        {
            // Return Free Inode
            return index;
        }
    }

    // Free Inode not available, return -1
    return END_INDICATOR;
}
```

7. file_system.C : Mount ():

This method is used to mount the filesystem from the disk. The inode and freelist blocks are read from disk and stored in the free-block list. Next, check and return boolean 'true' if the first two blocks of the free-block list are used (which are to be occupied by inode block and freelist block). Otherwise, return boolean 'false'.

```
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;

    // Load Inode Block
    read_inode_block_from_disk();

    // Load FreeList Block
    read_freelist_block_from_disk();

    // Check if first 2 blocks in disk are used
    if( (free_blocks[0] == 1) && (free_blocks[1] == 1) )
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

8. file_system.C : Format ():

This method is used to format the entire filesystem. Next, initialize an empty inode list and empty free list. The first two blocks used to store the inode list and free block list. The first two blocks are marked as used to prevent them from getting overwritten.

```
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 index = 0;
    unsigned char buffer[DISK_BLOCK_SIZE];

    // Initialize inode block to be empty
    for( index = 0; index < DISK_BLOCK_SIZE; index++ )
    {
        buffer[index] = END_INDICATOR;
    }

    _disk->write(INODE_BLOCK_NO, buffer);

    // Initialize free list to be empty
    for( index = 0; index < DISK_BLOCK_SIZE; index++ )
    {
        buffer[index] = 0x00;
    }

    // Set Inode block as used
    buffer[INODE_BLOCK_NO] = 1;

    // Set Free List block as used
    buffer[FREELIST_BLOCK_NO] = 1;

    _disk->write(FREELIST_BLOCK_NO, buffer);

    return true;
}
```

9. file_system.C : LookupFile ():

This method is used to find a file with a given 'file_id' in the filesystem. First, iterate through the inode list and compare the 'file_id'. If the file is found, return a pointer to the inode. Otherwise, NULL is returned.

```
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 index = 0;
    for(index = 0; index < MAX_INODES; index++)
    {
        if( inodes[index].id == _file_id )
        {
            return &inodes[index];
        }
    }

    Console::puts("LookupFile: file does not exist \n");
    return NULL;
}
```

10. file_system.C : CreateFile ():

This method is used to create a file with the given ID in the filesystem. Using the 'lookup' function, check if the file already exists in the filesystem, in which case a 'false' is returned. If the file does not exist already, obtain a free block. Next, attempt to obtain an unused inode. If free block and unused inode are obtained, then mark the free block as used and populate the fields in the inode with details of the file (file_id, free block number, pointer to filesystem object). Next, the inode block and free list block are written back to disk.

```
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. */

    short free_block_no = 0;
    int free_inode_idx = 0;

    if( LookupFile(_file_id) != NULL )
    {
        Console::puts("CreateFile: file exists already, cannot create file \n");
        return false;
    }

    free_block_no = GetFreeBlock();
    if( free_block_no == END_INDICATOR )
    {
        Console::puts("CreateFile: free blocks not available \n");
        return false;
    }

    free_inode_idx = GetFreeInode();
    if( free_inode_idx == END_INDICATOR )
    {
        Console::puts("CreateFile: free inodes not available \n");
        return false;
    }

    free_blocks[free_block_no] = 1;

    inodes[free_inode_idx].size = 0;
    inodes[free_inode_idx].id = _file_id;
    inodes[free_inode_idx].block_no = free_block_no;
    inodes[free_inode_idx].fs = this;

    write_inode_block_to_disk();
    write_freelist_block_to_disk();

    Console::puts("CreateFile: created file having id: ");
    Console::puti(_file_id);
    Console::puts("\n");

    return true;
}
```

11. file_system.C : DeleteFile ():

This method is used to delete a file with a given ID in the file system. First, check if the file exists using the 'lookup' function. If the file exists, then proceed to mark the block as free in the free-block

list. Also, the information stored in the inode list is also erased. The inode list and free-block list are written back to disk.

```
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. */

    // Check if file exists
    Inode * inode = LookupFile( _file_id );
    if( inode != NULL )
    {
        free_blocks[inode->block_no] = 0;

        inode->id = END_INDICATOR;
        inode->size = END_INDICATOR;
        inode->block_no = END_INDICATOR;

        write_inode_block_to_disk();
        write_freelist_block_to_disk();

        return true;
    }
    else
    {
        return false;
    }
}
```

12. file_system.C : Helper functions to read/write blocks to disk

The below mentioned functions perform the following operations:

- read_inode_block_from_disk() - Function to read data in disk into the inode block
- write_inode_block_to_disk() - Function to write data in inode block into the disk
- read_freelist_block_from_disk() - Function to read data in disk into the freelist block
- write_freelist_block_to_disk() - Function to write data in freelist block into the disk
- write_block_to_disk() - Function to write data stored in given buffer into disk
- read_block_from_disk() - Function to read data from disk into given buffer

```
void FileSystem::read_inode_block_from_disk()
{
    disk->read( INODE_BLOCK_NO, (unsigned char *)inodes );
}


void FileSystem::write_inode_block_to_disk()
{
    disk->write( INODE_BLOCK_NO, (unsigned char *)inodes );
}


void FileSystem::read_freelist_block_from_disk()
{
    disk->read( FREELIST_BLOCK_NO, free_blocks );
}


void FileSystem::write_freelist_block_to_disk()
{
    disk->write( FREELIST_BLOCK_NO, free_blocks );
}


void FileSystem::write_block_to_disk( unsigned long block_number, unsigned char * buffer )
{
    disk->write( block_number, buffer );
}


void FileSystem::read_block_from_disk( unsigned long block_number, unsigned char * buffer )
{
    disk->read( block_number, buffer );
}
```

13. file.H : class File:

In class File, data structures and functions for managing files and performing file operations are declared. The data structures added are given below:

- FileSystem * fs - Pointer to the file system object
- Inode * inode - Pointer to the inode object
- unsigned long current_position - Variable to track position within a file where read/write operations will be performed

```
class File  {

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

    /* 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. */
       FileSystem * fs;
       Inode * inode;
       unsigned long current_position;

    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:
```

14. file.C : File():

This method is the constructor for class File. The current position is set to the beginning of the file. The inode is obtained using the ID and 'lookup' function. Using the block number obtained, data is read from the correct block within the disk and loaded into the buffer.

```
File::File(FileSystem *_fs, int _id)
{
    Console::puts("Opening file.\n");
    fs = _fs;
    inode = fs->LookupFile(_id);
    current_position = 0;
    fs->read_block_from_disk( inode->block_no, block_cache );
}
```

15. file.C : ~File():

This method is the destructor for class File. The block cache buffer is written into the disk. The inode list is also written into the disk.

```
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. */
    fs->write_block_to_disk( inode->block_no, block_cache );
    fs->write_inode_block_to_disk();
}
```

16. file.C : Read():

This method is used to read 'n' characters (or till end of file is reached) from the file starting at the current position and copy it into the buffer. Data is read from the block cache and the buffer is filled. The number of characters read is returned by this method.

```
int File::Read(unsigned int _n, char *_buf)
{
    Console::puts("reading from file\n");

    unsigned long read_count = 0;

    while( (read_count < _n) && (EoF() == false) )
    {
        _buf[read_count] = block_cache[current_position];
        read_count = read_count + 1;
        current_position = current_position + 1;
    }

    return read_count;
}
```

17. file.C : Write():

This method is used to write 'n' characters to the file starting at the current position. If the write extends over the end of the file, the length of the file is extended until all the data is written or maximum size (one block) of the file is reached. The number of characters written is returned by this method.

```
int File::Write(unsigned int _n, const char *_buf)
{
    Console::puts("writing to file\n");

    unsigned int write_count = 0;
    unsigned int updated_inode_size = current_position + _n;

    // Updating inode size if required
    if( updated_inode_size > inode->size )
    {
        inode->size = updated_inode_size;
    }

    // Since disk block size = 512 bytes
    if( inode->size > DISK_BLOCK_SIZE )
    {
        inode->size = DISK_BLOCK_SIZE;
    }

    while( !EoF() )
    {
        block_cache[current_position] = _buf[write_count];
        write_count = write_count + 1;
        current_position = current_position + 1;
    }

    return write_count;
}
```

18. file.C : Reset():

This method is used to reset the current position to the beginning of the file.

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

19. file.C : EoF():

This method is used to check if the current position pointer has reached the end of the file. The operation is performed by checking if the current position is equal to the size of the file. A boolean 'true' is returned if the end of the file was reached. Otherwise, a boolean 'false' is returned.

```cpp
bool File::EoF()
{
    Console::puts("checking for EoF\n");
    if( current_position == inode->size )
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

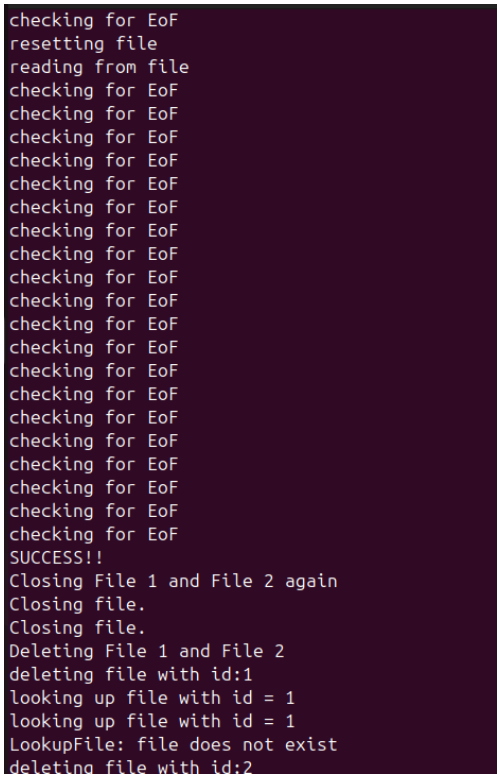
Testcase Output:

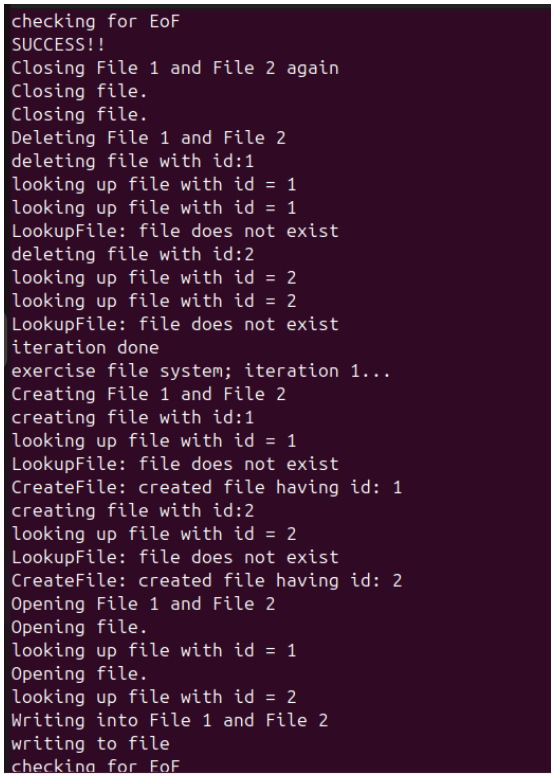
Testcase: Default testcase provided in kernel.C: Using the exercise_file_system() function to test all operations of the filesystem.

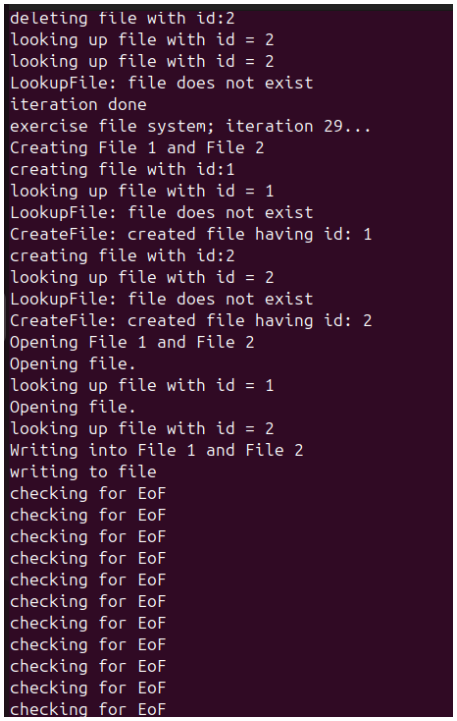
```
rahul-611@rahul-611-VirtualBox:~/Desktop/MP7_Sources/MP7_SourcesV2$ make run
Linux
qemu-system-x86_64 -kernel kernel.bin -serial stdio \
-device piix3-ide,id=ide -drive id=disk,file=c.img,format=raw,if=none -device ide-hd,drive=disk,bus=ide.0
Installed exception handler at ISR <0>
Allocating Memory Pool... done
Installed interrupt handler at IRQ <0>
Installed interrupt handler at IRQ <14>
In file system constructor.
Hello World!
before formatting...formatting disk
formatting completed
before mounting...mounting file system from disk
mounting completed
exercise file system; iteration 0...
Creating File 1 and File 2
creating file with id:1
looking up file with id = 1
LookupFile: file does not exist
CreateFile: created file having id: 1
creating file with id:2
looking up file with id = 2
LookupFile: file does not exist
CreateFile: created file having id: 2
Opening File 1 and File 2
Opening file.
looking up file with id = 1
Opening file.
looking up file with id = 2
Writing into File 1 and File 2
writing to file
checking for EoF
checking for EoF
```

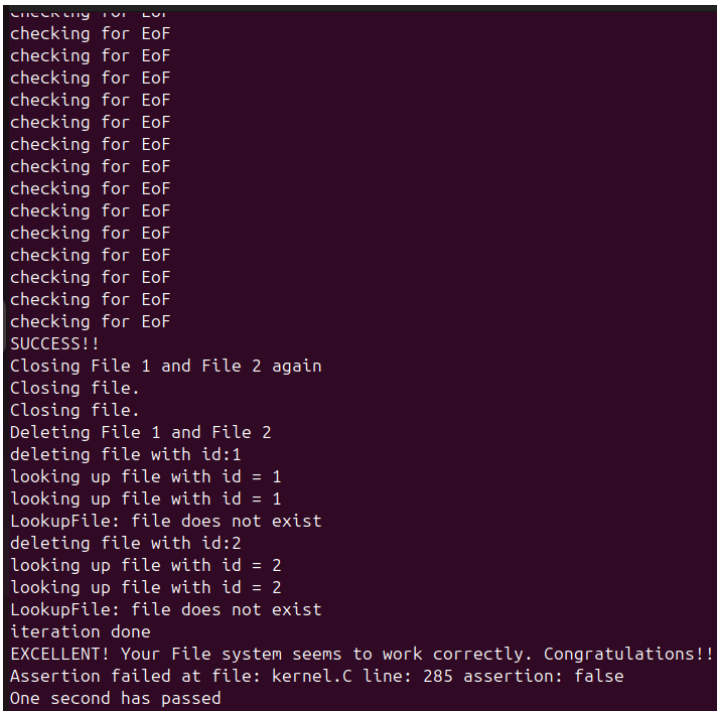
```
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
writing to file
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
```

```
checking for EoF
checking for EoF
Closing File 1 and File 2
Closing file.
Closing file.
Opening File 1 and File 2 again
Opening file.
looking up file with id = 1
Opening file.
looking up file with id = 2
Checking content of File 1 and File 2
resetting file
reading from file
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
checking for EoF
```









Filesystem was created. Two files were created and opened. Two strings of data were written into files and closed. The two files were opened again and data was read out of the files and the results were compared. The data written into files and read out of files were equivalent. No assertion errors were observed.

Testcase successful!

Bonus Option 1:
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 'block_no' (block 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 22, 28, 57. Then the array 'blocks' contains [22,28,57].

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

1. **Inode**:
    - Replace 'block_no' with 'blocks' to store all the blocks where the file is stored orderly.

2. **File**:
    - Replace 'block_no' 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 the 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 the '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 'block_no', the newly assigned free block is added to the 'blocks' list.
    - **DeleteFile** – Apart from removing the data structures associated with the file, all the 'blocks' allocated to the file are also freed.