Machine Problem 7: Vanilla File System

Gaurangi Sinha
UIN: 734004353
CPSC611: Operating System

Assigned Task

Main Task: Completed.
Bonus Option 1: Completed.
Bonus Option 2: Not Completed.

System Design

1. **Main Task: File System and File**

   The file system is designed:
   (a) Inode List and Free Block Management: The file system will manage an inode list, which keeps track of file metadata, including identifiers and block locations. It will also maintain a structure to track free disk blocks, ensuring efficient space utilization.
   (b) Disk Formatting Capability: The system will be equipped with a function to format the disk, setting up a clean slate for the file system to be installed, which includes initializing the inode list and free block management structures.
   (c) Mount Functionality: The file system will include a mount function that attaches the file system to a specified disk, enabling the disk to be used for file operations such as read and write.
   (d) File Creation: The file system will support operations to create new files on the disk, allocating necessary resources and updating the inode list accordingly.
   (e) Read and Write Operations: The File class will facilitate reading from and writing to files.

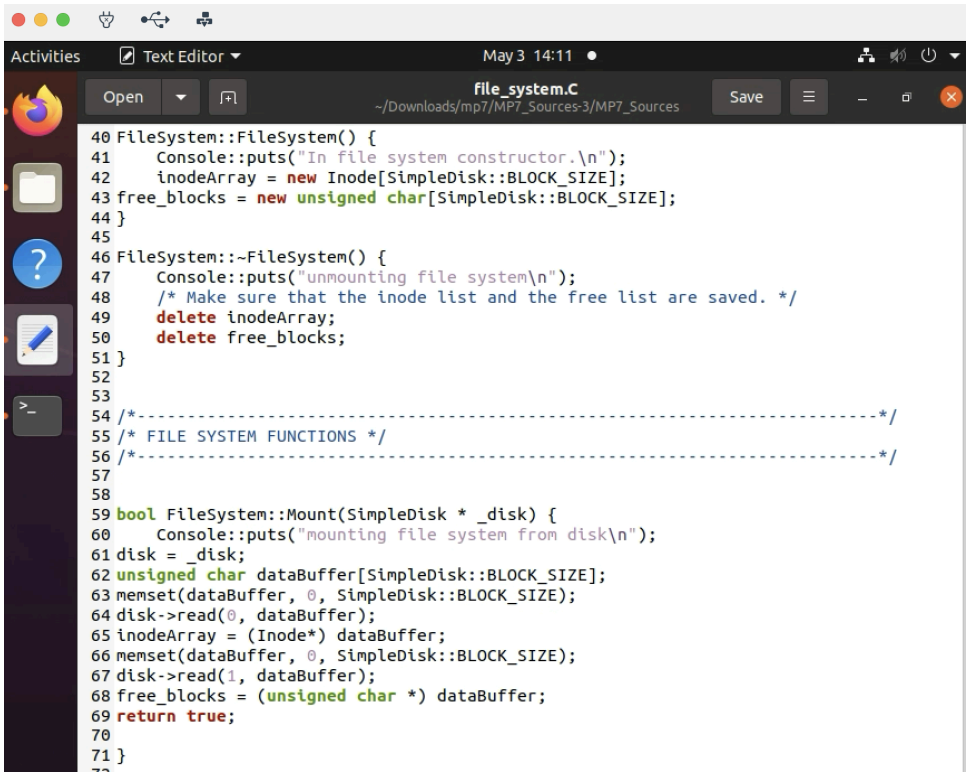
2. **BONUS Option 1: DESIGN of an extension to the basic file system to allow for files that are up to 64kB long.**

   a) Files are stored in a continuous block sequence, and since the inode already contains information about the first block and file size, no modifications are necessary for the inode structure.
   b) The Write method in the File class requires alterations to accommodate files that exceed the size of a single block. When the file size surpasses the block size, the system must allocate an additional block to store the excess data.

c) If the sum of the current position and the number of characters to be written exceeds the block size (position + n > BLOCK SIZE), the system needs to fetch a new block for reading operations.
d) To maintain file contiguity, any new block allocated must directly follow the last block used by the file.
e) When a contiguous free block is located, it should be marked as occupied in the free blocks array and the file's inode should be updated to reflect the new size.
f) If no contiguous free blocks are available, the system must:
i. Identify a new set of contiguous free blocks that can accommodate the current number of blocks plus one additional block.
ii. Transfer the data from the old blocks to this new contiguous set and update the free block tracker to reflect their usage.
iii. Free up the previously occupied blocks by marking them as available in the free block array.
iv. Update the inode to point to the new first block and adjust the file size metadata.
v. Continue the write operation from where it left off in the buffer.
vi. Ensure that the file does not exceed the 64KB size limit during write operations; if it does, the write operation should be terminated.
g) If it proves impossible to locate a sufficient number of contiguous free blocks that meet the new requirements, the write operation should be halted due to lack of disk space.
h) Modifications to the File class's read function are required to allow it to access data extending across multiple blocks. Since the blocks are contiguous, reading can seamlessly continue from one block to the next until the end of the file is reached (from the current block to inode.block + size/BLOCK Size).

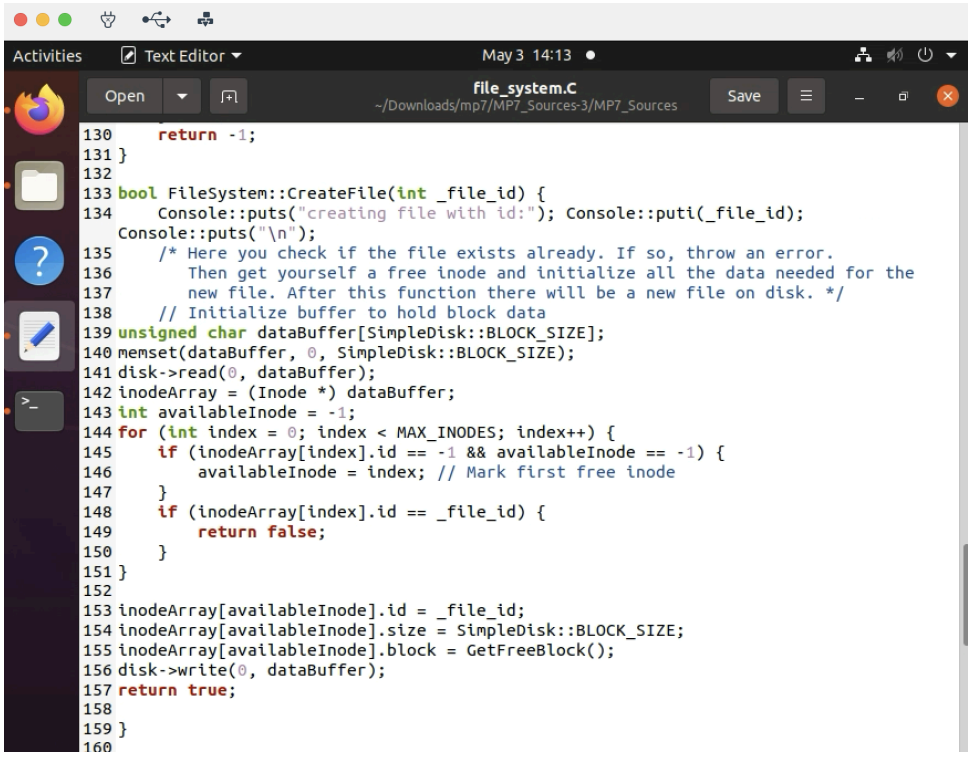
Code Description

I mainly made several changes in the files "file_system.C", "file_system.H", "file.C", and “file.H”.



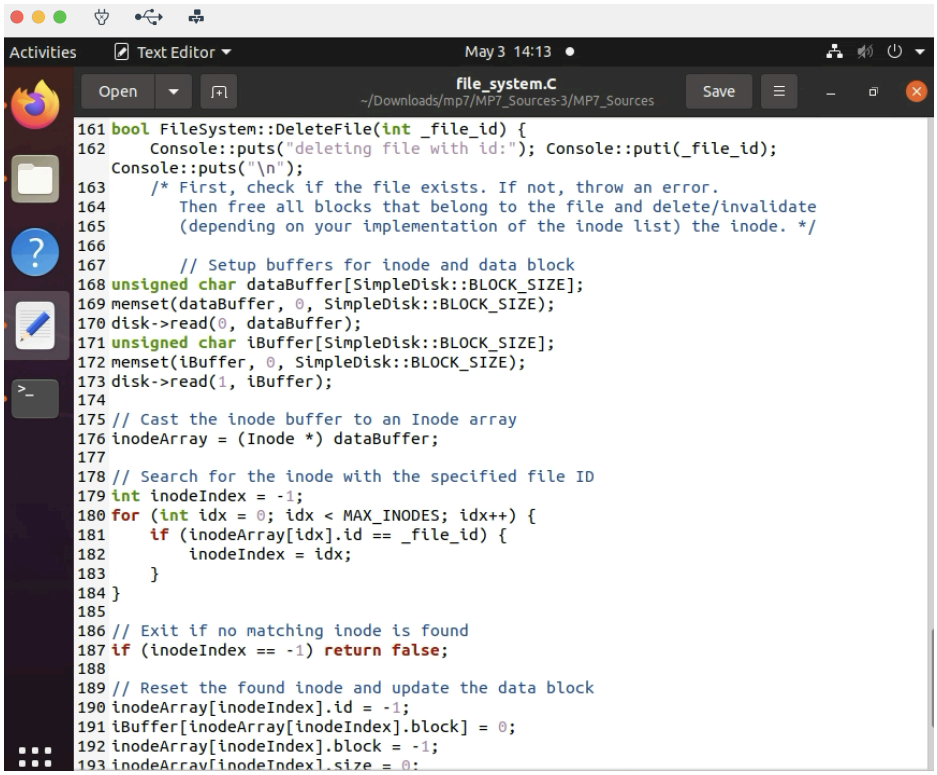

```
40 FileSystem::FileSystem() {
41     Console::puts("In file system constructor.\n");
42     inodeArray = new Inode[SimpleDisk::BLOCK_SIZE];
43 free_blocks = new unsigned char[SimpleDisk::BLOCK_SIZE];
44 }
45
46 FileSystem::~FileSystem() {
47     Console::puts("unmounting file system\n");
48     /* Make sure that the inode list and the free list are saved. */
49     delete inodeArray;
50     delete free_blocks;
51 }
52
53
54 /*--------------------------------------------------------------------------*/
55 /* FILE SYSTEM FUNCTIONS */
56 /*--------------------------------------------------------------------------*/
57
58
59 bool FileSystem::Mount(SimpleDisk * _disk) {
60     Console::puts("mounting file system from disk\n");
61 disk = _disk;
62 unsigned char dataBuffer[SimpleDisk::BLOCK_SIZE];
63 memset(dataBuffer, 0, SimpleDisk::BLOCK_SIZE);
64 disk->read(0, dataBuffer);
65 inodeArray = (Inode*) dataBuffer;
66 memset(dataBuffer, 0, SimpleDisk::BLOCK_SIZE);
67 disk->read(1, dataBuffer);
68 free_blocks = (unsigned char *) dataBuffer;
69 return true;
70
71 }
```

Figure: FileSystem Class Mount. To mount a disk by loading essential file system structures into memory, such as the inode array and free block list. It prepares the file system for operations like file creation, deletion, and data manipulation.



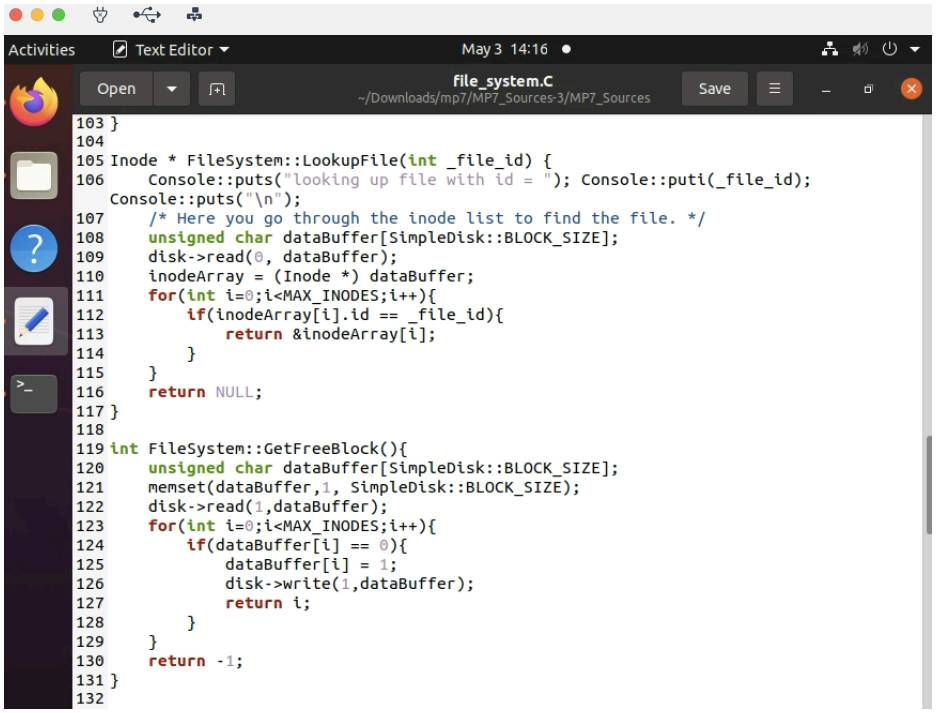

```
130     return -1;
131 }
132
133 bool FileSystem::CreateFile(int _file_id) {
134     Console::puts("creating file with id:"); Console::puti(_file_id);
    Console::puts("\n");
135     /* Here you check if the file exists already. If so, throw an error.
136        Then get yourself a free inode and initialize all the data needed for the
137        new file. After this function there will be a new file on disk. */
138     // Initialize buffer to hold block data
139 unsigned char dataBuffer[SimpleDisk::BLOCK_SIZE];
140 memset(dataBuffer, 0, SimpleDisk::BLOCK_SIZE);
141 disk->read(0, dataBuffer);
142 inodeArray = (Inode *) dataBuffer;
143 int availableInode = -1;
144 for (int index = 0; index < MAX_INODES; index++) {
145     if (inodeArray[index].id == -1 && availableInode == -1) {
146         availableInode = index; // Mark first free inode
147     }
148     if (inodeArray[index].id == _file_id) {
149         return false;
150     }
151 }
152
153 inodeArray[availableInode].id = _file_id;
154 inodeArray[availableInode].size = SimpleDisk::BLOCK_SIZE;
155 inodeArray[availableInode].block = GetFreeBlock();
156 disk->write(0, dataBuffer);
157 return true;
158
159 }
160
```

Figure: FileSystem class Create: It first checks if a file with the given ID already exists. If it does, the function returns `false`, indicating failure to create the file due to a duplicate ID. Once a free inode is found, it sets the inode's ID to the provided file ID, initializes the size, and assigns a free block from the disk.



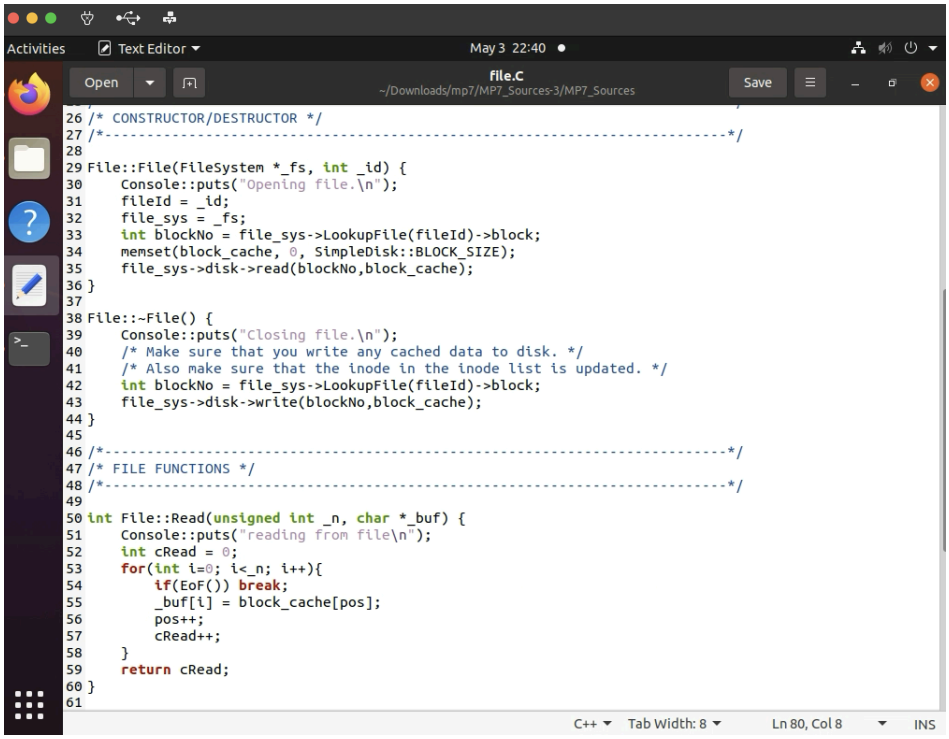

```
161 bool FileSystem::DeleteFile(int _file_id) {
162     Console::puts("deleting file with id:"); Console::puti(_file_id);
    Console::puts("\n");
163     /* First, check if the file exists. If not, throw an error.
164        Then free all blocks that belong to the file and delete/invalidate
165        (depending on your implementation of the inode list) the inode. */
166
167        // Setup buffers for inode and data block
168 unsigned char dataBuffer[SimpleDisk::BLOCK_SIZE];
169 memset(dataBuffer, 0, SimpleDisk::BLOCK_SIZE);
170 disk->read(0, dataBuffer);
171 unsigned char iBuffer[SimpleDisk::BLOCK_SIZE];
172 memset(iBuffer, 0, SimpleDisk::BLOCK_SIZE);
173 disk->read(1, iBuffer);
174
175 // Cast the inode buffer to an Inode array
176 inodeArray = (Inode *) dataBuffer;
177
178 // Search for the inode with the specified file ID
179 int inodeIndex = -1;
180 for (int idx = 0; idx < MAX_INODES; idx++) {
181     if (inodeArray[idx].id == _file_id) {
182         inodeIndex = idx;
183     }
184 }
185
186 // Exit if no matching inode is found
187 if (inodeIndex == -1) return false;
188
189 // Reset the found inode and update the data block
190 inodeArray[inodeIndex].id = -1;
191 iBuffer[inodeArray[inodeIndex].block] = 0;
192 inodeArray[inodeIndex].block = -1;
193 inodeArray[inodeIndex].size = 0;
```

Figure: FileSystem class delete

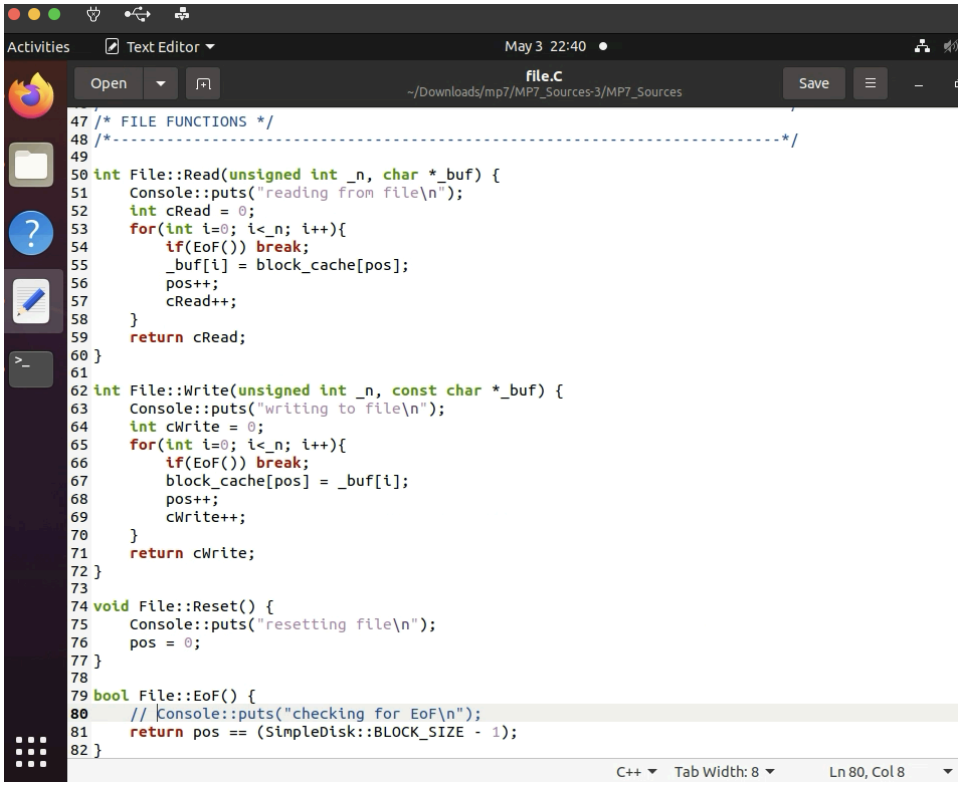



```
103 }
104
105 Inode * FileSystem::LookupFile(int _file_id) {
106     Console::puts("looking up file with id = "); Console::puti(_file_id);
    Console::puts("\n");
107     /* Here you go through the inode list to find the file. */
108     unsigned char dataBuffer[SimpleDisk::BLOCK_SIZE];
109     disk->read(0, dataBuffer);
110     inodeArray = (Inode *) dataBuffer;
111     for(int i=0;i<MAX_INODES;i++){
112         if(inodeArray[i].id == _file_id){
113             return &inodeArray[i];
114         }
115     }
116     return NULL;
117 }
118
119 int FileSystem::GetFreeBlock(){
120     unsigned char dataBuffer[SimpleDisk::BLOCK_SIZE];
121     memset(dataBuffer,1, SimpleDisk::BLOCK_SIZE);
122     disk->read(1,dataBuffer);
123     for(int i=0;i<MAX_INODES;i++){
124         if(dataBuffer[i] == 0){
125             dataBuffer[i] = 1;
126             disk->write(1,dataBuffer);
127             return i;
128         }
129     }
130     return -1;
131 }
132
```

File Class



The code snippet contains methods related to managing file operations in a filesystem. The constructor `File::File(FileSystem *_fs, int _id)` opens a file by fetching its data block from the disk based on the file ID and reads it into a cache. The destructor `File::~File()` handles the closing of a file by ensuring that any cached data is written back to the disk to maintain data consistency and updates the corresponding inode. The `File::Read(unsigned int _n, char *buf)` function reads data from the file into a buffer up to `_n` bytes or until the end of the file is reached, managing the read position within the cache and updating it accordingly. These functions collectively ensure that files are opened, data is read, and changes are saved back efficiently, maintaining the integrity and consistency of the filesystem.




```
/* FILE FUNCTIONS */
/*-----------------------------------------------------------------------*/

int File::Read(unsigned int _n, char *_buf) {
    Console::puts("reading from file\n");
    int cRead = 0;
    for(int i=0; i<_n; i++){
        if(EoF()) break;
        _buf[i] = block_cache[pos];
        pos++;
        cRead++;
    }
    return cRead;
}

int File::Write(unsigned int _n, const char *_buf) {
    Console::puts("writing to file\n");
    int cWrite = 0;
    for(int i=0; i<_n; i++){
        if(EoF()) break;
        block_cache[pos] = _buf[i];
        pos++;
        cWrite++;
    }
    return cWrite;
}

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

bool File::EoF() {
    // Console::puts("checking for EoF\n");
    return pos == (SimpleDisk::BLOCK_SIZE - 1);
}
```

Compilation

To compile the code, we run the following commands:

make clean

make

A simple script to copy the kernel onto the floppy image. The script mounts the floppy image, copies the kernel image onto it, and then unmounts the floppy image again. So we use the below command.

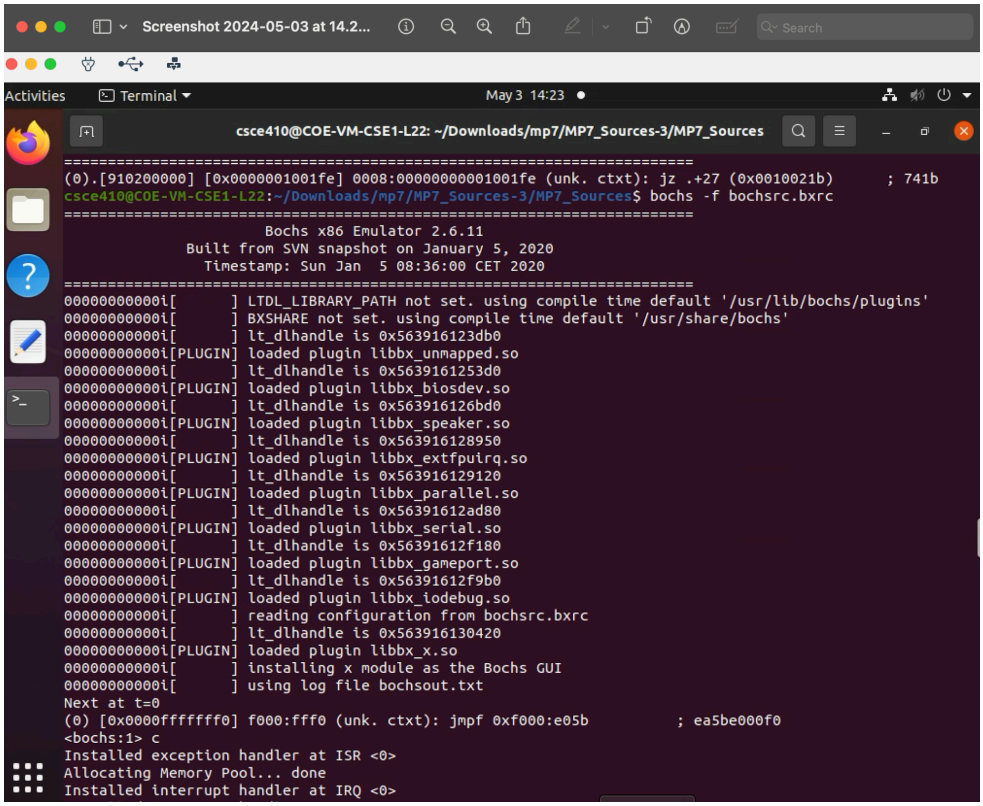
./copykernel.sh

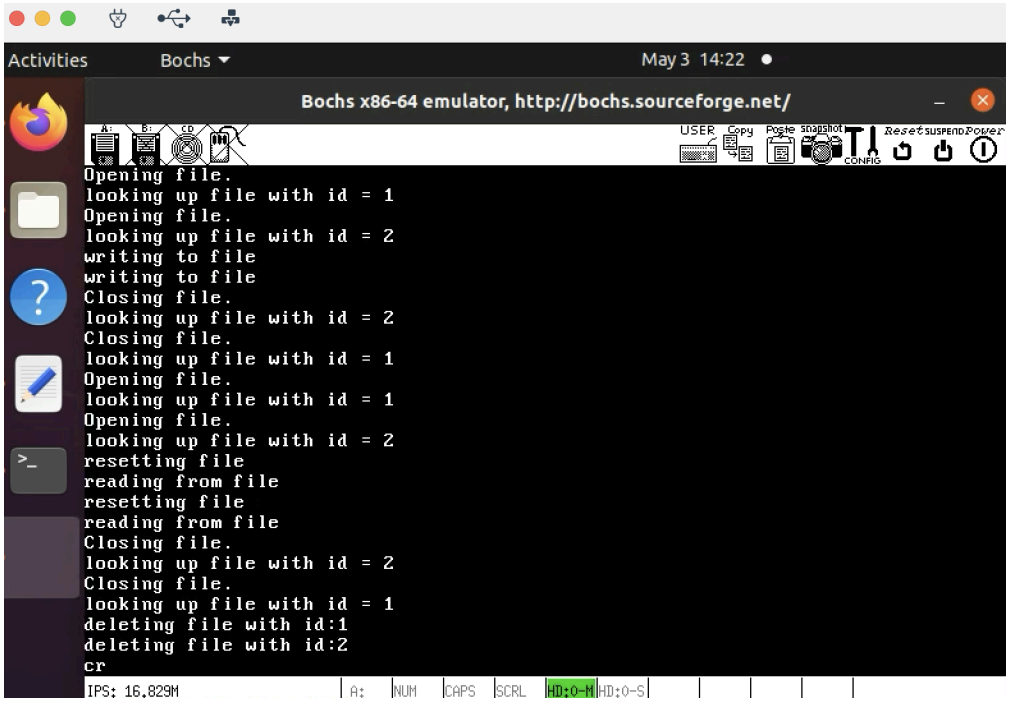
and then to start the Bochs simulator we run the following command:

bochs –f bochsrc.bxrc

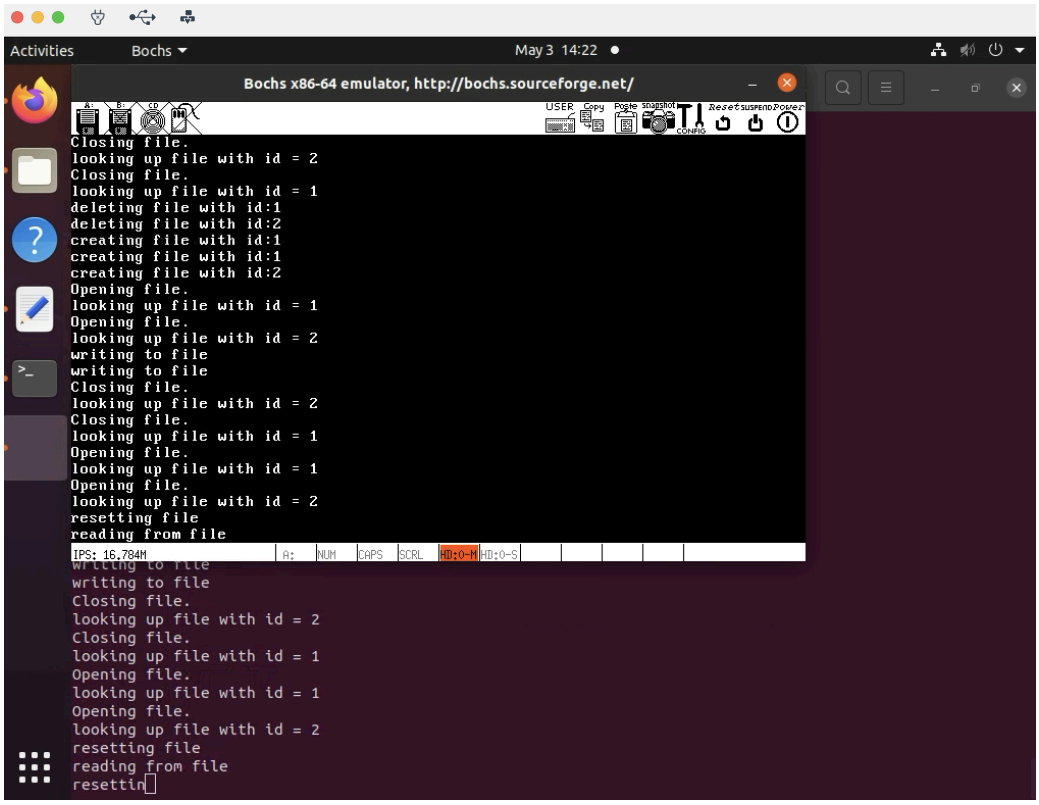
Testing

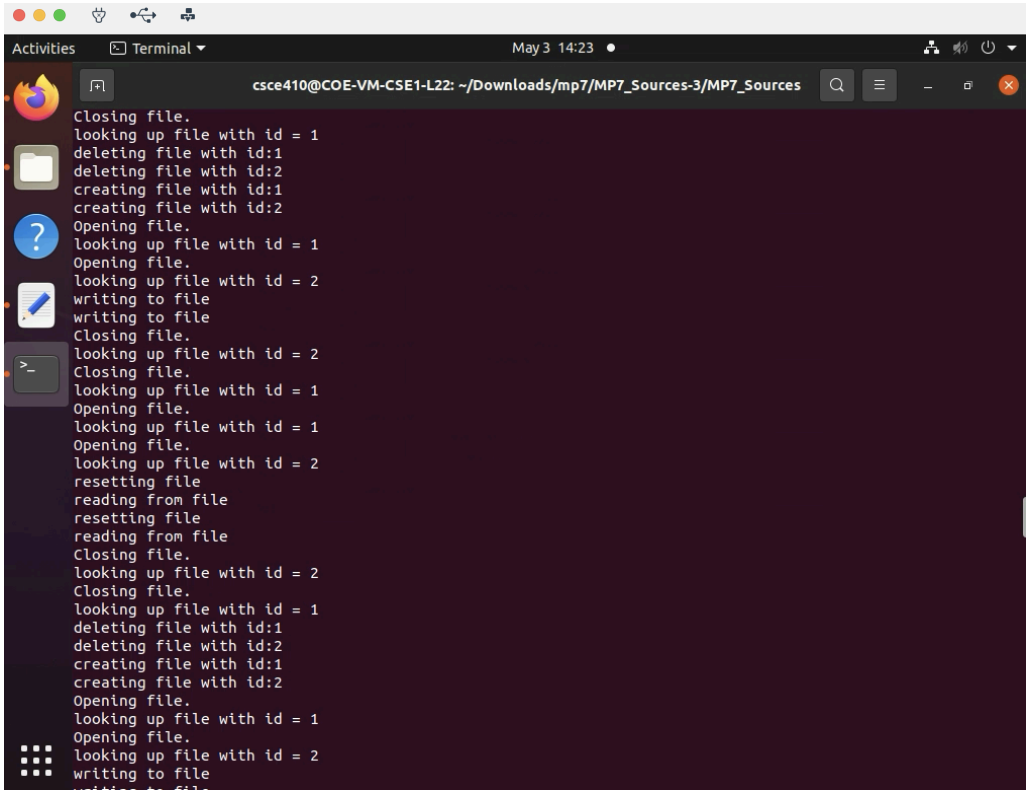
For testing, the machine problem, I ran the above command and, I used the given test function from kernel.C. I ran the normal testing given in kernel. All the tests are passing, for given and additional scenarios.





Testing the creation, deletion, read and write two files with id 1 and 2.


Activities
Bochs
May 3 14:22
Bochs x86-64 emulator, http://bochs.sourceforge.net/
Closing file.
looking up file with id = 2
Closing file.
looking up file with id = 1
deleting file with id:1
deleting file with id:2
creating file with id:1
creating file with id:1
creating file with id:2
Opening file.
looking up file with id = 1
Opening file.
looking up file with id = 2
writing to file
writing to file
Closing file.
looking up file with id = 2
Closing file.
looking up file with id = 1
Opening file.
looking up file with id = 1
Opening file.
looking up file with id = 2
resetting file
reading from file
IPS: 16.784M
writing to file
Closing file.
looking up file with id = 2
Closing file.
looking up file with id = 1
Opening file.
looking up file with id = 1
Opening file.
looking up file with id = 2
resetting file
reading from file
resettin


Activities
Terminal
May 3 14:23
csce410@COE-VM-CSE1-L22: ~/Downloads/mp7/MP7_Sources-3/MP7_Sources
Closing file.
looking up file with id = 1
deleting file with id:1
deleting file with id:2
creating file with id:1
creating file with id:2
Opening file.
looking up file with id = 1
Opening file.
looking up file with id = 2
writing to file
writing to file
Closing file.
looking up file with id = 2
Closing file.
looking up file with id = 1
Opening file.
looking up file with id = 1
Opening file.
looking up file with id = 2
resetting file
reading from file
resetting file
reading from file
Closing file.
looking up file with id = 2
Closing file.
looking up file with id = 1
deleting file with id:1
deleting file with id:2
creating file with id:1
creating file with id:2
Opening file.
looking up file with id = 1
Opening file.
looking up file with id = 2
writing to file