# MP7: Vanilla File System

Vaibhav Pundir
UIN: 734004197
CSCE611: Operating System

## Assigned Tasks

**Main:** Completed.
**Bonus Option 1:** Did not attempt.
**Bonus Option 2:** Did not attempt.

## System Design

In this machine problem, the aim is to implement a basic file system that only supports sequential access. The file system supports files with a maximum size of 512B (size of one block). Block 0 is used for storing information about inodes (Inodes block) and block 1 stores a bitmap of free blocks (FreeList block).

The list of free blocks is represented as a char array. The list of inodes is represented as an array of Inode class objects. An Inode object includes details like the File ID, the block number where the file is stored, and the file's size.

During file creation, we search for a free block and assign it to the file. The block number of the block given to the file is saved in the inode. Since all files are 512B (one block size) long at most, the block number is the only allocation information needed for the file.

The FileSystem class includes functions for reading and writing the inode list and free block list to and from the disk. These functions are listed below:
- ReadInodeFromDisk(): Read inode block from disk.
- WriteInodeToDisk(): Write inode block to disk.
- ReadBlock(): Read block with _blk_no from disk to _buffer.
- WriteBlock() Write block with _blk_no to disk from _buffer.

The implementation is further explained below:

## Code Description

For this machine problem, I've made changes to the following code files:

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

## file_system.H

In the Inode class, data structures and methods for managing inodes are defined.
- blk_no: keeps track of block no. where the file is stored.
- size: keeps track of the size of the file.
- ReadInodeFromDisk(): Read inode block from disk.
- WriteInodeToDisk(): Write inode block to disk.

```
class Inode
{
  ...
  unsigned long blk_no;      // block no where file is stored
  unsigned long size;        // file size
  ...
  void ReadInodeFromDisk();
  void WriteInodeToDisk();
};

class FileSystem
{
  ...
  short GetFreeInode();
  int GetFreeBlock();
  ...
  void ReadBlock(unsigned long _blk_no, unsigned char *_buffer);
  /* Read _blk_bo from disk and load into _buffer. */

  void WriteBlock(unsigned long _blk_no, unsigned char *_buffer);
  /* write _buffer to _blk_no of the disk. */
};
```

Listing 1: C++ code snippet

In the FileSystem class, data structures and methods for managing the file system are defined.
- The functions GetFreeInode() & GetFreeBlock() were uncommented since they are used to return the index of an available free block & free inode respectively.
- Added methods ReadBlock() & WriteBlock() to read/ write data to/from the disk.

```
class FileSystem
{
  ...
  short GetFreeInode();
  int GetFreeBlock();
  ...
  void ReadBlock(unsigned long _blk_no, unsigned char *_buffer);
  /* Read _blk_bo from disk and load into _buffer. */

  void WriteBlock(unsigned long _blk_no, unsigned char *_buffer);
  /* write _buffer to _blk_no of the disk. */
};
```

Listing 2: C++ code snippet

## file_system.C

**Inode::ReadInodeFromDisk()** : Helper function to read inode block from disk.

```
void Inode::ReadInodeFromDisk() {
    fs->ReadBlock(INODES_BLOCK_NO, (unsigned char *)fs->inodes);
}
```

Listing 3: C++ code snippet

**Inode::WriteInodeToDisk()** : Helper function to write inode block to disk.

```
void Inode::WriteInodeToDisk() {
    fs->WriteBlock(INODES_BLOCK_NO, (unsigned char *)fs->inodes);
}
```

Listing 4: C++ code snippet

**FileSystem::FileSystem()** : Constructor of FileSystem class. We initialize inodes and free_blocks variables here.

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

Listing 5: C++ code snippet

**FileSystem:: FileSystem()** : Destructor method. Write inode list and free list to disk and free memory used by inodes and free_blocks variable.

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

    WriteBlock(INODES_BLOCK_NO, (unsigned char *)inodes);
    delete []inodes;

    WriteBlock(FREELIST_BLOCK_NO, free_blocks);
    delete []free_blocks;
}
```

Listing 6: C++ code snippet

**FileSystem::GetFreeInode()** : Method used to find a free inode from inode list and return its index.

```cpp
short FileSystem::GetFreeInode() {
    for(unsigned int idx=0; idx<MAX_INODES; ++idx) {
        // check and return the index of free inode
        if(inodes[idx].id == 0xFFFFFFFF)
            return idx;
    }
    // no free inode available
    return -1;
}
```

Listing 7: C++ code snippet

**FileSystem::GetFreeBlock()** : Method used to find a free block from free list and return its index.

```cpp
int FileSystem::GetFreeBlock() {
    for(unsigned int idx=0; idx<DISK_BLOCK_SIZE; ++idx) {
        // check and return the index for free block
        if(free_blocks[idx] == 0)
            return idx;
    }
    // no free block available
    return -1;
}
```

Listing 8: C++ code snippet

**FileSystem::Mount()** : Method to read the inode and free list block from disk. Checks if the first two blocks are in-use (for inodes and free list).

```cpp
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;

    ReadBlock(INODES_BLOCK_NO, (unsigned char *)inodes);
    ReadBlock(FREELIST_BLOCK_NO, free_blocks);

    // check if first two blocks are in use for inodes and free list
    return (free_blocks[0] == 1 && free_blocks[1] == 1);
}
```

Listing 9: C++ code snippet

**FileSystem::Format()** : Method to format the filesystem. All the inodes and free blocks are marked as unused except for the first two blocks which are used to store inode and free list.

```
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 char buffer[DISK_BLOCK_SIZE];

    // mark all the inodes as unused
    for(unsigned int idx=0; idx<DISK_BLOCK_SIZE; ++idx)
        buffer[idx] = 0xFF;

    _disk->write(INODES_BLOCK_NO, buffer);

    // mark all the free blocks as unused
    for(unsigned int idx=0; idx<DISK_BLOCK_SIZE; ++idx)
        buffer[idx] = 0x00;

    // first two blocks are in use as inode block and free list block
    buffer[0] = buffer[1] = 0x01;
    _disk->write(FREELIST_BLOCK_NO, buffer);

    return true;
}
```

Listing 10: C++ code snippet

**FileSystem::LookupFile()** : Method to return the pointer to the inode which contains the file information for _file_id. If file does not exist, return nullptr.

```
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. */

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

    Console::puts("file with id = "); Console::puti(_file_id); Console::puts(" does not exist!\n");
    return nullptr;
}
```

Listing 11: C++ code snippet

**FileSystem::CreateFile()** : Checks if file already exists. If not, find an inode and a free block to be used for the file creation. Mark the free block as used and update the inode with file information.

```
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. */

```

```cpp
    // check if file exists
    if(LookupFile(_file_id)) {
        Console::puts("file already exists!\n");
        return false;
    }

    // get a free block
    int free_blk_no = GetFreeBlock();
    if(free_blk_no == -1) {
        Console::puts("free blocks not available!\n");
        return false;
    }

    // get a free inode
    short free_inode_idx = GetFreeInode();
    if(free_inode_idx == -1) {
        Console::puts("free inodes not available!\n");
        return false;
    }

    // mark block in-use
    free_blocks[free_blk_no] = 1;

    // update inode
    inodes[free_inode_idx].id = _file_id;
    inodes[free_inode_idx].blk_no = free_blk_no;
    inodes[free_inode_idx].size = 0;
    inodes[free_inode_idx].fs = this;

    WriteBlock(INODES_BLOCK_NO, (unsigned char *)inodes);
    WriteBlock(FREELIST_BLOCK_NO, free_blocks);

    return true;
}
```

Listing 12: C++ code snippet

**FileSystem::DeleteFile()** : If the file exists, mark the block and inode as unused.

```cpp
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. */

    Inode *inode = LookupFile(_file_id);
    // check if file exists
    if(inode == nullptr) {
        Console::puts("file does not exist!\n");
        return false;
    }

    // mark block as not in-use
    free_blocks[inode->blk_no] = 0;
    // update inode
    inode->id = 0xFFFFFFFF;
    inode->blk_no = 0xFFFFFFFF;
    inode->size = 0xFFFFFFFF;

    WriteBlock(INODES_BLOCK_NO, (unsigned char *)inodes);
    WriteBlock(FREELIST_BLOCK_NO, free_blocks);

    return true;
}
```

Listing 13: C++ code snippet

**FileSystem::ReadBlock()** : Method to read data in _blk_no from disk into _buffer.

```cpp
void FileSystem::ReadBlock(unsigned long _blk_no, unsigned char *_buffer) {
    disk->read(_blk_no, _buffer);
}
```

Listing 14: C++ code snippet

**FileSystem::WriteBlock()** : Method to write data to _blk_no in disk from _buffer.

```cpp
void FileSystem::WriteBlock(unsigned long _blk_no, unsigned char *_buffer) {
    disk->write(_blk_no, _buffer);
}
```

Listing 15: C++ code snippet

## file.H

In the File class, data structures and methods for managing files and handling file operations are defined.
- inode: pointer to the inode.
- fs: pointer to the file system.
- pos: keeps track of position within a file for read and write operations.

```cpp
class File  {
    ...
    Inode           *   inode;  // pointer to the inode
    FileSystem      *   fs;     // pointer to the file system
    unsigned long       pos;    // keeps track of position within a file for read and
    write operations.
    ...
};
```

Listing 16: C++ code snippet

## file.C

**File::File()** : Constructor method to initialize fs, inode, and pos. Using _id, the respective inode point is retrieved and the position is initialized to 0 initially. From the inode's block no., the entire block is loaded from disk to block_cache.

```cpp
File::File(FileSystem *_fs, int _id) {
    Console::puts("Opening file.\n");

    fs = _fs;
    inode = fs->LookupFile(_id);
    pos = 0;
    fs->ReadBlock(inode->blk_no, block_cache);
}
```

Listing 17: C++ code snippet

**File:: File()** : Destructor method. Writes block_cache and inode to disk.

```cpp
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->WriteBlock(inode->blk_no, block_cache);
    inode->WriteInodeToDisk();
}
```

Listing 18: C++ code snippet

**File::Read()** : Method to read data from the file and return the no. of bytes read. Read until _n bytes are read or EoF is reached.

```cpp
int File::Read(unsigned int _n, char *_buf) {
    Console::puts("reading from file\n");
    unsigned int len = 0;
    // iterate until EoF is reached or _n bytes have been read
    while(!EoF() && len < _n) {
        _buf[len] = block_cache[pos];
        ++len;
        ++pos;
    }

    // return no. of total bytes read
    return len;
}
```

Listing 19: C++ code snippet

**File::Write()** : Method to write the data to the file. First, the updated size of the file is determined (capped at 512B). Then, _n bytes are written to the block_cache from the _buf.

```cpp
int File::Write(unsigned int _n, const char *_buf) {
    Console::puts("writing to file\n");
    // adjust file size if write would extend beyond current size
    if(pos + _n > inode->size)
        inode->size = pos + _n;

    // ensure file size does not exceed 512B
    if(inode->size > DISK_BLOCK_SIZE)
        inode->size = DISK_BLOCK_SIZE;

    unsigned int len = 0;
    while(!EoF() && len < _n) {
        block_cache[pos] = _buf[len];
        ++len;
        ++pos;
    }

    return len;
}
```

Listing 20: C++ code snippet

**File::Reset()** : Method to set the position to 0.

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

Listing 21: C++ code snippet

**File::EoF()** : Method to find if end-of-file is reached or not.

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

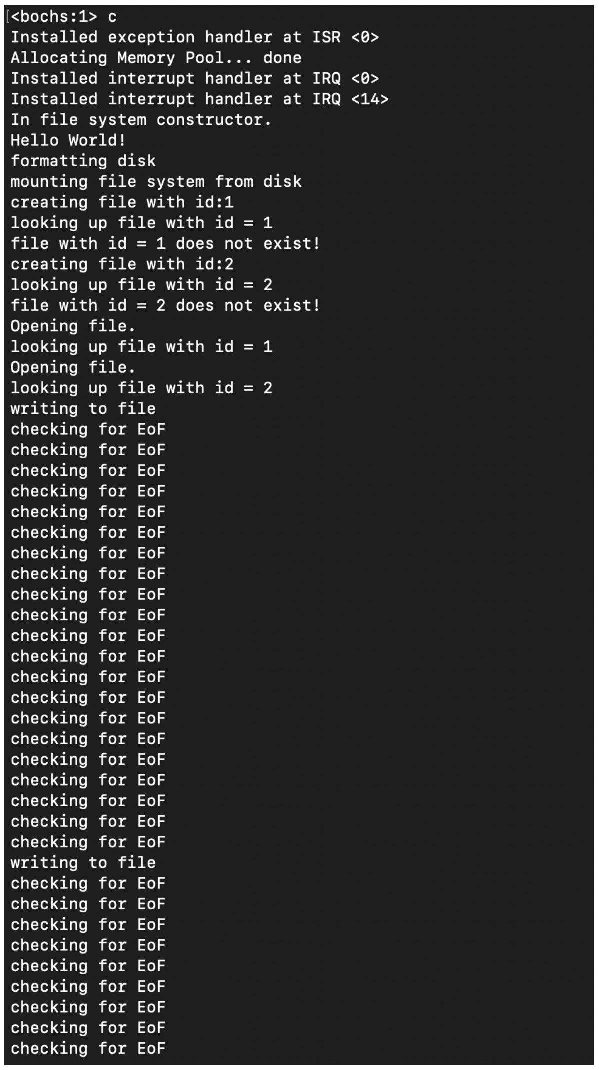
Listing 22: C++ code snippet

## Testing

To test the implementation, I used the default test provided in the kernel.C.

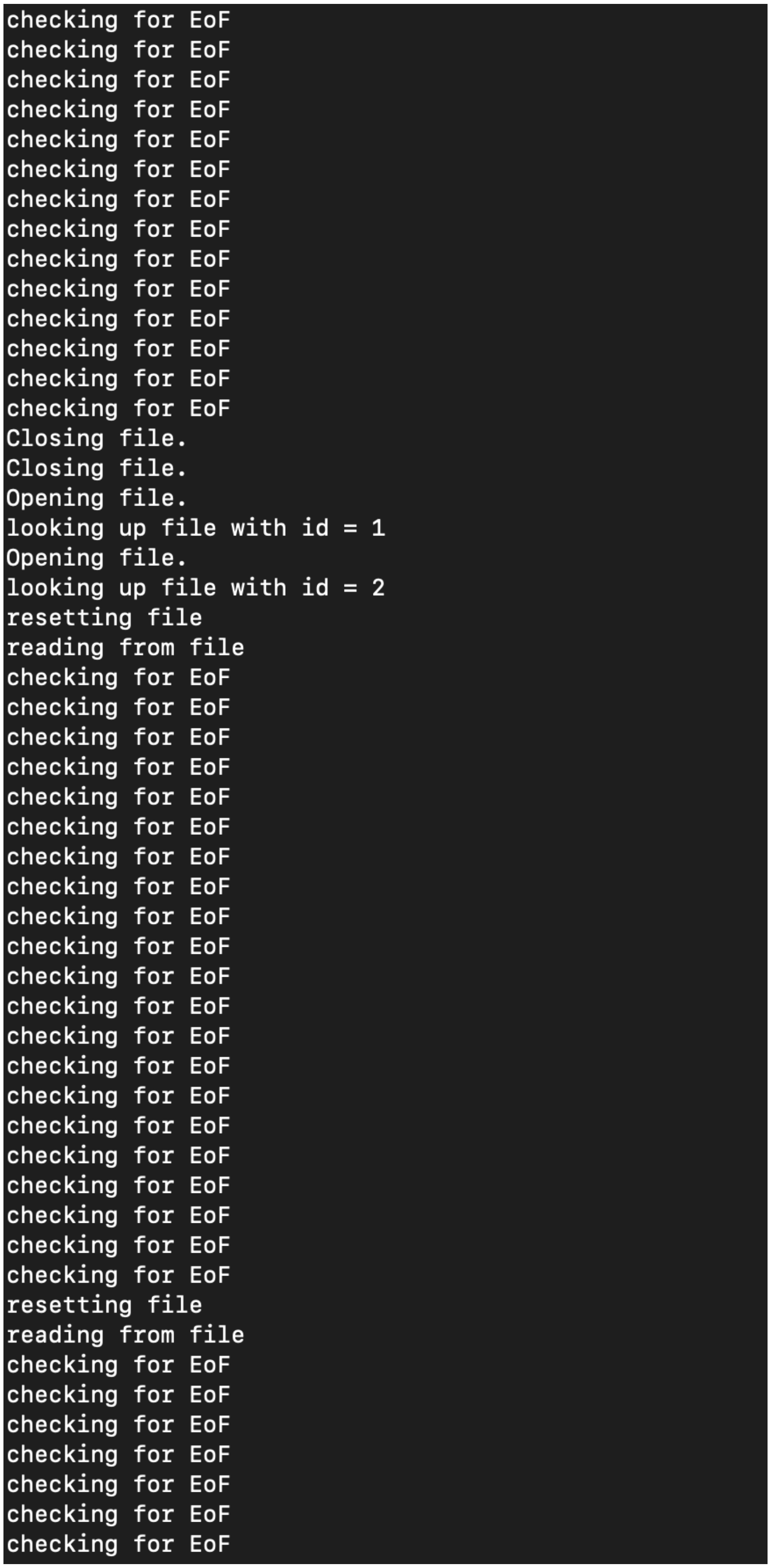
### TEST

To test the operations of the file system, the exercise_file_system() method was invoked.
**Expected execution:**
- The file system is set up and two files are created and opened.
- Two strings are written to the files and then closed.
- The files are reopened and the data is read back and compared.
- The data written to the files and read from them should match exactly and no assertion errors should occur.
**Actual output (as seen from the screenshots):**
- The file system is set up and two files are created and opened.
- Two strings are written to the files and then closed.
- The files are reopened and the data is read back and compared.
- The data written to the files and read from them match exactly and no assertion errors occurred.



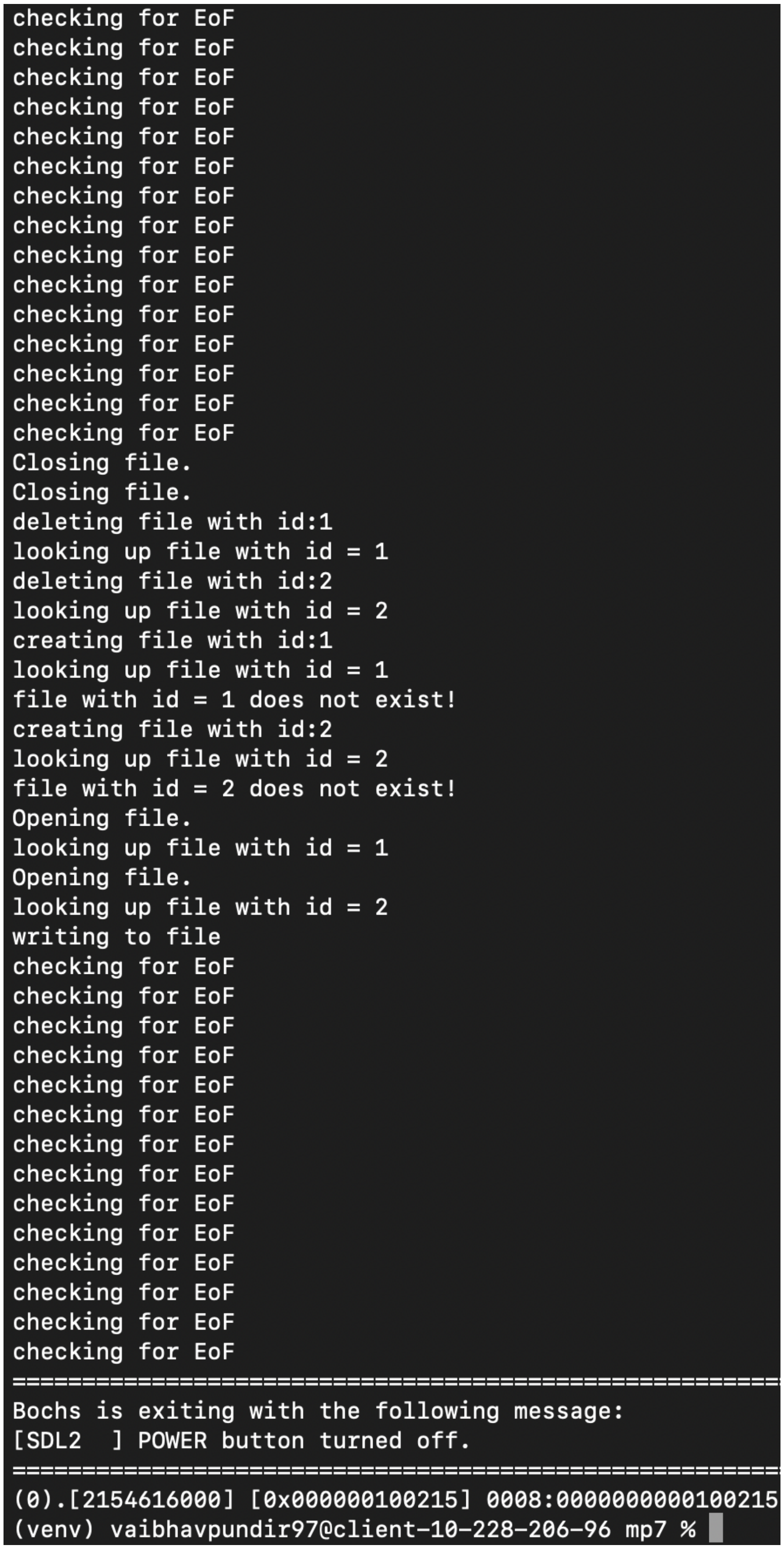
```
[<bochs:1> c
Installed exception handler at ISR <0>
Allocating Memory Pool... done
Installed interrupt handler at IRQ <0>
Installed interrupt handler at IRQ <14>
In file system constructor.
Hello World!
formatting disk
mounting file system from disk
creating file with id:1
looking up file with id = 1
file with id = 1 does not exist!
creating file with id:2
looking up file with id = 2
file with id = 2 does not exist!
Opening file.
looking up file with id = 1
Opening file.
looking up file with id = 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
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
```

Figure 1: Test



```
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.
Closing file.
Opening file.
looking up file with id = 1
Opening file.
looking up file with id = 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
checking for EoF
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
```

Figure 2: Test



```
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.
Closing file.
deleting file with id:1
looking up file with id = 1
deleting file with id:2
looking up file with id = 2
creating file with id:1
looking up file with id = 1
file with id = 1 does not exist!
creating file with id:2
looking up file with id = 2
file with id = 2 does not exist!
Opening file.
looking up file with id = 1
Opening file.
looking up file with id = 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
=========================================================
Bochs is exiting with the following message:
[SDL2  ] POWER button turned off.
=========================================================
(0).[2154616000] [0x000000100215] 0008:0000000000100215
(venv) vaibhavpundir97@client-10-228-206-96 mp7 %
```

Figure 3: Test