

MP7: Vanilla File System

Sathvik Kote Revanasiddappa

UIN: 634005706

CSCE611: Operating System

Assigned Tasks

Main: Completed.

Bonus Option 1: Did not attempt.

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. If a character corresponding to a block is set 0, then the block is empty. If the character is set to 1, then the block is in use.

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.

File Operations

File Create

- Store the data structures required for handling file. These are reference to file system, file id. In the file system, find the inode of a given file.
- Set the current position to 0.
- Read the file from the block and maintain the block in the cache.

File Close

- While closing the file, write the cache back to the disk.
- Write the inode block to the disk if there are any changes to the file metadata.

File read

- Read n characters from the current position from the cached block.

File write

- Check whether the new write will exceed the limit of the file size.
- Copy the characters from buffer to block and increment the current position.

File System

FS Mount

- Copy the reference of disk into your variable.
- Read the first block (Inode block) into inode cache and second block (Free Block list) into free list cache.
- Check whether the first two blocks are marked as used in the free list.
- If not return false.

FS Format

- Clear all the information on the first two blocks (Inode Block and free block list).
- Mark the first elements of the free block list with value 1, to denote the first two blocks are used to store inode and free block information.

File Creation

- Check if there is any file with the same id in the inode list. If yes, return false.
- Find a free block for the new file.
- Find a new Inode to store the meta data about the file.
- Update both the Inode and files block and write it to disk.

File Delete

- LookUp for the file.
- Update the free block list and inode information.
- Write inode and free block list back to disk.

Implementation Details

Mac Support

I have modified the below files to support the functioning of MP3 problem on Mac M2.

- copykernel.sh
- bochsrc.bxrc
- file_system.C
- file.C

```

#define INODE_BLOCK 0
#define FILES_BLOCK 1
#define BLOCK_SIZE 512
#define NO_META_BLOCKS 2

FileSystem::FileSystem()
{
    Console::puts("In file system constructor.\n");

    inodes = new Inode[MAX_INODES];

    free_blocks = new unsigned char[BLOCK_SIZE];

    no_files = 0;

    no_nodes = 0;

    disk = nullptr;

    size = 0;
}

```

Figure 1: File System Constructor

```

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

    delete[] inodes;
    delete[] free_blocks;
}

```

Figure 2: File System Destructor

fel_system.H/C

1. `FileSystem::FileSystem()` : The file system constructor is used to initialize all the variables.
2. `FileSystem::FileSystem()` : All the updates are written back to the disk before the file system is destructed.
3. `bool FileSystem::Mount(SimpleDisk *_disk)` : Mounts the disk onto the file system. Performs few checks before mounting the disk.
4. `bool FileSystem::Format(SimpleDisk *_disk, unsigned int _size)`: Clear all the information on the first two blocks. And Use the first two blocks for storing the inode and free blocks.
5. `Inode *FileSystem::LookupFile(int _file_id)`: Check if the file with the given file id exists in the file system by searching the inode list.
6. `unsigned long FileSystem::find_free_block` : Check for free block in the file system and return the block number.

```

// Copy the reference of disk into your variable
// Read the first block into inode cache and second block into free list cache
// Check whether the first two blocks are marked as used in the free list
bool FileSystem::Mount(SimpleDisk *_disk)
{
    Console::puts("mounting file system from disk\n");

    disk = _disk;

    disk->read(INODE_BLOCK, (unsigned char *)inodes);

    disk->read(FILE_BLOCK, free_blocks);

    if (free_blocks[INODE_BLOCK] == 1 && free_blocks[FILE_BLOCK] == 1)
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

Figure 3: File System Mount

```

bool Console::format(SimpleDisk *_disk, unsigned int _size)
{
    Console
    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 i = 0;
    unsigned char buffer[BLOCK_SIZE];
    Inode *buf_inodes = new Inode[MAX_INODES];

    // Initialize inode block to -1
    for (i = 0; i < MAX_INODES; i++)
    {
        buf_inodes[i].id = -1;
    }

    _disk->write(INODE_BLOCK, (unsigned char *)buf_inodes);

    // Initialize File List to 0
    for (i = 0; i < BLOCK_SIZE; i++)
    {
        buffer[i] = 0;
    }

    // Mark the first two blocks as used
    buffer[INODE_BLOCK] = 1;
    buffer[FILE_BLOCK] = 1;

    // Write the updated file block to disk
    _disk->write(FILE_BLOCK, buffer);

    delete[] buf_inodes;
    delete[] buffer;

    return true;
}

```

Figure 4: File System Format

7. unsigned long FileSystem::find_free_inode : Check for free inode in the list and return the index of it.
8. bool FileSystem::CreateFile(int _file_id): To create a file in the file system, check whether the file exists and then find a free block and free inode to store all the information for the file. Once found, update both the inode list and free block list and write it to disk.
9. bool FileSystem::DeleteFile(int _file_id): Deleting a file requires updating the inode list and free block list. Once updated, write back the updated blocks back to the disk.

file.H/C

1. File::File(FileSystem *_fs, int _id): Initialise all the variables and read the file block into cache.
2. File::File(): Update disk and inode information before closing the closing the file descriptor.

```

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 (int i = 0; i < no_nodes; i++)
    {
        if (inodes[i].id == _file_id)
        {
            return &inodes[i];
        }
    }
    return nullptr;
}

```

Figure 5: File System LookUp

```

unsigned long FileSystem::find_free_block()
{
    for (unsigned long i = 0; i < no_files; i++)
    {
        if (free_blocks[i] == 0)
        {
            return i;
        }
    }

    return -1;
}

```

Figure 6: File System FindFreeBlock

```

unsigned long FileSystem::find_free_inode()
{
    for (unsigned long i = 0; i < no_nodes; i++)
    {
        if (inodes[i].id == -1)
        {
            return i;
        }
    }

    return -1;
}

```

Figure 7: File System FindFreeInode

3. int File::Read(unsigned int _n, char *_buf): Read n characters from the current position.
4. int File::Write(unsigned int _n, const char *_buf): Write n characters from the current position with limit of block size.
5. void File::Reset(): Resets the current position to the beginning of the file.
6. bool File::Eof(): Check if the current position reached the end of file.

Testing

You can find logs in the output.txt.

- Default testcase provided in kernel.C: Using the exercise_file_system() function to test all operations of the filesystem. Filesystem is created. Two files are created and opened.

All tests are passing.

```

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. */
    if (LookupFile(_file_id) != nullptr)
    {
        return false;
    }

    unsigned long free_block_no = find_free_block();

    if (free_block_no == -1)
    {
        if (no_files == BLOCK_SIZE - 2)
        {
            Console::puts("CreateFile: free blocks not available \n");
            return false;
        }
        free_block_no = no_files;
        no_files++;
    }

    // Can improve
    unsigned long free_inode_no = find_free_inode();
    if (free_inode_no == -1)
    {
        if (no_nodes == MAX_INODES)
        {
            Console::puts("CreateFile: iNodes not available \n");
            return false;
        }
        free_inode_no = no_nodes;
        no_nodes++;
    }

    free_blocks[free_block_no] = 1;
    inodes[free_inode_no].size = 0;
    inodes[free_inode_no].id = _file_id;
    inodes[free_inode_no].block_no = free_block_no;

    disk->write(INODE_BLOCK, (unsigned char *)inodes);
    disk->write(FILE_BLOCK, (unsigned char *)free_blocks);

    Console::puts("CreateFile: File is created.");

    return true;
}

```

Figure 8: File System CreateFile

```

bool FileSystem::DeleteFile(int _file_id)
{
    Console::puts("deleting file with id:");
    Console::puti(_file_id);
    Console::puts("\n");

    Inode *file_inode = LookupFile(_file_id);
    if (file_inode == nullptr)
    {
        Console::puts("DeleteFile: File not found.");
        return false;
    }

    free_blocks[file_inode->block_no] = 0;
    file_inode->id = -1;

    disk->write(INODE_BLOCK, (unsigned char *)inodes);
    disk->write(FILE_BLOCK, (unsigned char *)free_blocks);

    Console::puts("DeleteFile: File is deleted.");
    return true;
}

```

Figure 9: File System DeleteFile

```

// Initiate all the variables.
// Read the file block into the cache.
File::File(FileSystem *_fs, int _id)
{
    Console::puts("Opening file.\n");
    fs = _fs;
    file_id = _id;
    current_position = 0;
    inode = fs->LookupFile(_id);
    fs->read_block_from_disk(inode->block_no, block_cache);
}

```

Figure 10: File Constructor

```

// Write back the block to 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_to_disk();
}

```

Figure 11: File Destructor

```

// Read n characters from the cached block.
int File::Read(unsigned int _n, char *_buf)
{
    Console::puts("reading from file\n");
    unsigned long index = 0;

    while (!Eof() && index < _n)
    {
        _buf[index] = block_cache[current_position];
        current_position++;
        index++;
    }
    return index;
}

```

Figure 12: File Read

```

// write n characters to the cached block.
int File::Write(unsigned int _n, const char *_buf)
{
    Console::puts("writing to file\n");
    unsigned long new_position = current_position + _n;

    if (new_position > inode->size)
    {
        inode->size = new_position;
    }

    if (inode->size > FILE_SIZE)
    {
        inode->size = FILE_SIZE;
    }

    unsigned long index = 0;
    while (!Eof() && index < _n)
    {
        block_cache[current_position] = _buf[index];
        current_position++;
        index++;
    }
    return index;
}

```

Figure 13: File Write

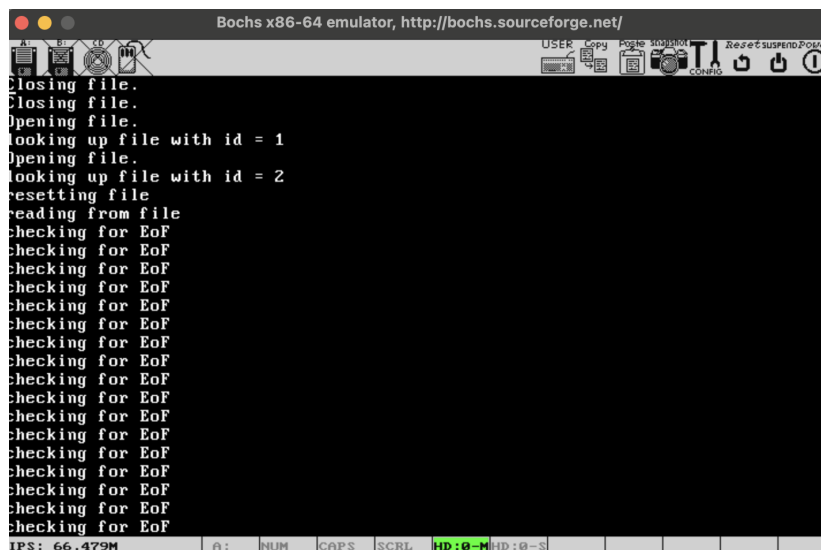
```

// Reset the position from the current position.
void File::Reset()
{
    Console::puts("resetting file\n");
    current_position = 0;
}

// Check if the current position reached the end of file.
bool File::EoF()
{
    Console::puts("checking for EoF\n");
    if (current_position >= inode->size)
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

Figure 14: File Reset and EOF



The screenshot shows a Bochs x86-64 emulator window with the title bar "Bochs x86-64 emulator, http://bochs.sourceforge.net/". The window contains a series of text output lines representing file operations and EOF checks. The output is as follows:

```

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

```

At the bottom of the window, there is a status bar with the following information: "IPS: 66.479M", "A: ", "NUM", "CAPS", "SCRL", "HD:0-M", "HD:0-S", and several icons for user, copy, paste, and other functions.

Figure 15: Test 1