

The goal of this computer problem is to construct a straightforward file system with files having a maximum size of 1 block, or 512 bytes. Simpler versions of the inode and free blocks array are used to implement the file system, which simply provides sequential access within files. The following classes are used to implement the file system:

1) FILE SYSTEM: It contains variables:

- a. MAX_INODES: maximum number of inodes we can use in the file system
- b. Free_block_count: maximum number of free blocks in the file system
- c. Free_blocks: pointer to a bitmap of free blocks. Uses the identifier 'f' for denoting a free block and 'u' for denoting the occupied block
- d. Disk: pointer to the disk
- e. Inodes: pointer to an array of inodes

The free block bitmap is stored in the first block on the disk (block index number 0) and the inodes array is stored on the second block of the disk (block index number 1).

The free block bitmap is stored in the first block on the disk (block index number 0) and the inodes array is stored on the second block of the disk (block index number 1).

Here's the list of functions:

- a. FileSystem(): initializes the data structures on class file system like inodes array etc.
- b. ~FileSystem(): saves the file system class data structures to disk.
- c. Mount(SimpleDisk *_disk): loads the data structures like inode array and free blocks bitmap from disk to file system class. And associates this disk with our file system class.
- d. LookupFile(int _file id) gives back the inode linked to the specified file id. If the inode for the specified file id cannot be identified, an assertion error is thrown.
- e. Mount(SimpleDisk *_disk) loads data structures from the disk to the file system class, including inode arrays and free blocks bitmaps. and connects our file system class to this disk.
- f. CreateFile(int _file_id): creates a new file with the given file id. Basically initializes a new inode for the file, maps it to the file, finds a new free block, allocates it to the file, and marks the block as used in the free blocks bitmap.
- g. DeleteFile(int _file_id): deletes everything related to the file. Frees up the inode associated as well as the block(marks the block in free blocks bitmap as free)

2) FILE

All of the variables and functions related to a single file are contained in this class. like opening, reading, or updating a file, etc. The following variables are present:

- a. `block_cache[SimpleDisk::BLOCK_SIZE]`; : buffer handler for file . basically a cached copy of the block we are reading and writing to. max size is 512 bytes.
- b. `FileSystem *fs`: file system associated with the file
- c. `int file_id`; id of the file
- d. `unsigned int current_position`; indicates where the next character will be read from or written to
- e. `unsigned int inode_index`; index of an inode in inode array that is assigned to the file
- f. `unsigned int file_size`; the size of the file
- g. `unsigned int block_no`; block number allocated the file

Functions present:

- a. `File(FileSystem * _fs, int _id)`; constructor of the file handle. Initializes and updates the variables of a particular file into the class data structures from the data of the fs
- b. `~File()`; close the file. Updates the inode associated accordingly and saves it to the disk.
- c. `int Read(unsigned int _n, char * _buf)`; reads from the file from current position. Reads `_n` characters; if `_n` characters are not there, then to the end of the file.
- d. `void Reset()`; sets the current position to the beginning of the file
- e. `bool EoF()`; checks if the current position is at the end of the file and returns true in that case.
- f. `int Write(unsigned int _n, const char * _buf)`; writes `_n` characters starting with current position. Increments file size to one block to 1 block if not enough. But never beyond that. Returns the number of characters written

3) INODE

This is a class containing all the variables associated with the inode like file number, allocated block number, file system, file size, and a flag to see whether the inode is free or not.

OPTION 1: DESIGN OF FILE SYSTEM FOR FILES THAT ARE UPTO 64KB LONG.

In this situation, several blocks must be assigned to a single file. In this scenario, all we need to do is keep a pointer in the inode for the sequence of blocks linked to the file.

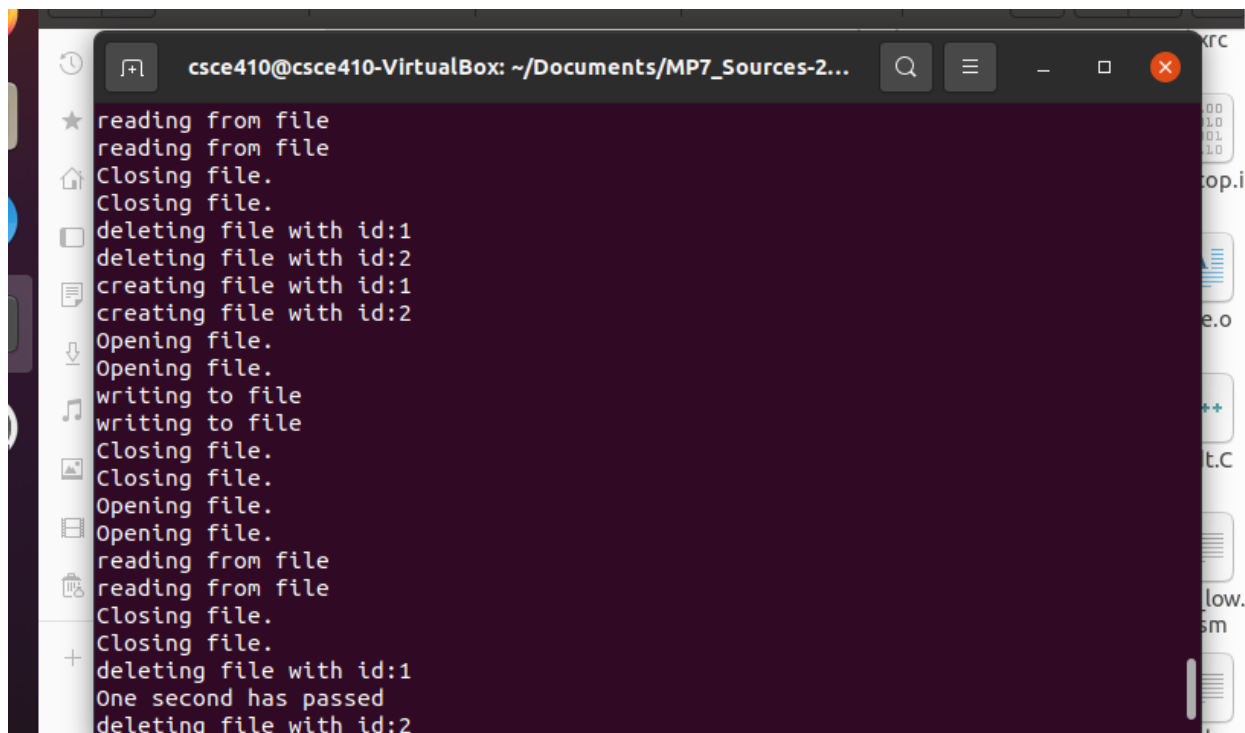
Hence inode class has an extra array `Long blocks_list[BLOCK_SIZE]`

For example: if the size of the file is up to five blocks we can maintain a sequence of blocks like this: `blocks_list = [2,3,4,5,7]`

When creating a new file in the file system, we assign a block list rather than a single block. And then indicate the free blocks. When we delete a file, we also designate all of the array's blocks as free and delete the array in the associated inode.

OPTION 2: IMPLEMENTATION of the extensions proposed in Option 1.

Sequential reading and writing will need to be somewhat altered for the file class. Here, we are reading from a series of blocks rather than a single block, thus the code must be adjusted accordingly. In the same way, writing to a file. When the current position reaches the final block, the file is considered to be out of the file. By grabbing a free block from the file system and adding it to the block list, we can also build a new function that, if the file size is insufficient, can expand the file size.



The image shows a terminal window titled "csce410@csce410-VirtualBox: ~/Documents/MP7_Sources-2...". The terminal output consists of the following lines:

```
★ reading from file
reading from file
Closing file.
Closing file.
deleting file with id:1
deleting file with id:2
creating file with id:1
creating file with id:2
Opening file.
Opening file.
writing to file
writing to file
Closing file.
Closing file.
Opening file.
Opening file.
reading from file
reading from file
Closing file.
Closing file.
deleting file with id:1
One second has passed
deleting file with id:2
```