CPSC 3500 Computing Systems
Winter 2018

Programming Assignment 3: File System
Due: Monday, March 5 at 2:05pm

Description

In this assignment, you will be implementing a simple file system over a simulated disk.

Copy the source code files from `/home/fac/elarson/cpsc3500/pa3` to get started. A `Makefile` is also provided.

Since we do not have a raw disk for each of you, a single (large) Unix file to emulate a disk. The file is to be thought of as an array of blocks. In this assignment, there are 1,024 blocks (numbered from 0 to 1023) and each block is 128 bytes.

The provided code implements a layered architecture as follows:

Shell (`Shell.cpp` and `Shell.h`)	Processes commands from the command line.
File System (`FileSys.cpp` and `FileSys.h`)	Provides an interface for file system commands.
Basic File System (`BasicFileSys.cpp` and `BasicFileSys.h`)	A low-level interface that interacts with the disk.
Disk (`Disk.cpp` and `Disk.h`)	Represents a "virtual" disk that is contained within a file.

Each of the four layers is implemented using a class. The class contains ("has-a") a single instance of the lower layer. For instance, the file system class has an instance of the basic file system.

Your task is to implement the following file system commands. It is recommended to implement these functions in the order they appear:

`mkdir <directory>`	Creates an empty subdirectory in the current directory.
`ls`	List the contents of the current directory. The precise format is described later in the document.
`cd <directory>`	Change to specified directory. The directory must be a subdirectory in the current directory. No paths or ".." are allowed.
`home`	Switch to the home directory.
`rmdir <directory>`	Removes a subdirectory. The subdirectory must be empty.
`create <filename>`	Creates an empty file of the filename in the current directory. An empty file consists of an inode and no data blocks.
`append <filename> <data>`	Appends the data to the file. Data should be appended in a manner to first fill the last data block as much as possible and then allocating new block(s) ONLY if more space is needed. More information about the format of data files is described later.
`stat <name>`	Displays stats for the given file or directory. The precise format is described later in the document.
`cat <filename>`	Display the contents of the file to the screen. Print a newline when completed.
`rm <filename>`	Remove a file from the directory, reclaim all of its blocks including its inode. Cannot remove directories.

The above list shows the shell commands. You will implementing these commands in the file system (`FileSys.cpp`). Each command has a corresponding function with the same name. Commands that require a file name or directory name have a name parameter. The append function also has a second parameter for the data.

The only files that requires modification are `FileSys.cpp` and `FileSys.h`. In `FileSys.h`, you are only allowed to modify the private section. You can add extra data members and private member functions.

Basic File System Interface Routines

To implement these commands, you will need to utilize routines provided by the basic file system. The file system class contains a basic file system interface, specifically private data member `bfs`. Here is a description of the provided routines you will need to use:

```
// Gets a free block from the disk.
short get_free_block();

// Reclaims block making it available for future use.
void reclaim_block(short block_num);

// Reads block from disk. Output parameter block points to new block.
void read_block(short block_num, void *block);

// Writes block to disk. Input block points to block to write.
void write_block(short block_num, void *block);
```

Note that basic file system also provides code for mounting (initializing) and unmounting (cleaning up) the basic file system. The basic file system is already mounted and unmounted in the provided code so there is no need to use the `mount` and `unmount` functions in your code.

File System Blocks

There are two types of files: data files (that store a sequence of characters) and directories. Data files consist of an inode and zero or more data blocks. Directories consist of a single directory block that stores the contents of the directory.

There are four types of blocks used in the file system:

- *Superblock:* There is only one superblock on the disk and that is always block 0. It contains a bitmap on what disk blocks are free.

- *Directories:* Represents a directory. The first field is a magic number which is used to distinguish between directories and inodes. The second field stores the number of files located in the directory. The remaining space is used to store the file entries. Each entry consists of a name and a block number (the directory block for directories and the inode block for data files). Unused entries are indicated by having a block number of 0. Block 1 always contains the directory for the "home" directory.

- *Inodes:* Represents an index block for a data file. In this assignment, only direct index pointers are used. The first field is a magic number which is used to distinguish between directories and inodes. The second field is the size of the file (in bytes). The remaining space consists of an array of indices to data blocks of the file. Use 0 to represent unused pointer entries (note that files cannot access the superblock).

- *Data blocks:* Blocks currently used to store data in files.

The different blocks are defined using these structures defined in `Blocks.h`. These structures are all `BLOCK_SIZE` (128) bytes.

```
// Superblock - keeps track of which blocks are used in the filesystem.
// Block 0 is the only super block in the system.
struct superblock_t {
  unsigned char bitmap[BLOCK_SIZE]; // bitmap of free blocks
};

// Directory block - represents a directory
struct dirblock_t {
  unsigned int magic;           // magic number, must be DIR_MAGIC_NUM
  unsigned int num_entries;     // number of files in directory
  struct {
    char name[MAX_FNAME_SIZE + 1]; // file name (extra space for null)
    short block_num;               // block number of file (0 - unused)
  } dir_entries[MAX_DIR_ENTRIES];  // list of directory entries
};

// Inode - index node for a data file
struct inode_t {
  unsigned int magic;            // magic number, must be INODE_MAGIC_NUM
  unsigned int size;             // file size in bytes
  short blocks[MAX_DATA_BLOCKS]; // array of direct indices to data blocks
};

// Data block - stores data for a data file
struct datablock_t {
  char data[BLOCK_SIZE];        // data (BLOCK_SIZE bytes)
};
```

You will use the basic file system interface routines to read and write these blocks. For instance, to read the home directory (block 1):

```
struct dirblock_t dirblock;
bfs.read_block(1, (void *) &dirblock);
```

In some cases, you will not know whether the block is a directory or an inode. To address this issue, both the directory and inode contain a magic number located as the first field (same spot in memory). For an unknown block, read the block as a directory block (or an inode block – it doesn't matter). Then read the magic number. If the magic number is DIR_MAGIC_NUM, then it is a directory. If the magic number is INODE_MAGIC_NUM, then it is an inode.

TIP: Create a private member function `is_directory` than returns true if the block corresponds to a directory and false otherwise.

Since the blocks are a fixed size, there are limits on the size of files, size of file names, and the number of entries in a directory. There constants, along with other file system parameters, are also defined in `Blocks.h`.

Data File Format

Here are the rules concerning data files:

- Data files consists of a single index block and zero or more data blocks.
- The command create creates an empty file. This creates an inode but no data blocks as the file is empty.
- The data string passed into `append` is null terminated, you can use `strlen` to determine its size.
- When appending data using `append`, *do not add a null termination character*. If appending "ABC" to the file, exactly three characters are appended. Use the size data member to determine the end of the file.
- When appending data, you need to add characters where you left off. If there is room in the last block, that block needs to be filled before adding a new block. If the data to append does not completely fit in the last block: completely fill in the last block, then create a new data block for the remainder.
- There is no limit* on the size of the data to append so it may be necessary to create two or more data blocks with a single call to append. *You will have to check for situations where the append would exceed the maximum file size.
- Only create a new block when it is absolutely necessary to create one. For instance, a file consisting of exactly 128 bytes should have only one (completely full) data block.
- Since the data is not null terminated, it is recommended that `append` copies characters one at a time and that `cat` displays characters one at a time. You may be tempted to use C string functions (such as `strcpy`) or using `<<` on the entire block but they rely on a null termination character being present. C++ strings aren't appropriate or necessary either.
- Due to the nature of the shell and the limited commands available, it is impossible to append special characters to a file including '`\0`', '`\n`', and a space.

ls Format

The contents of the directory must be printed out such that each item (file or directory) in the directory is printed out on its own line. Here is an example:

```
dir1/
file1
file3
dir2/
file2
```

Additional notes:

- Directories should have a '`/`' suffix such as '`myDir/`'.
- Files do not have a suffix.
- The items can appear in any order.
- If the directory is empty, print out nothing.

stat Format

For directories, print out the directory number.

```
Directory block: 7
```

For data files, print out the inode block, number of bytes stored in the file, and the number of blocks the file consumes (including the inode).

```
Inode block: 5
Bytes in file: 170
Number of blocks: 3
```

Empty files store 0 bytes and take up 1 block (inode). Non-empty files take up at least 2 blocks (one inode block and at least one data block).

Running the Program

There are two ways to run the program: interactive mode and script mode. In interactive mode, you enter commands just like a UNIX shell. To run in interactive mode, simply type:

```
./filesys
```

The program will run indefinitely until the user enters '`quit`' for a command.

In script mode, you supply a script that contains a series of commands. The commands will execute in order until the end of script or until the script executes a '`quit`' command. It may be useful in creating small test scripts. These test scripts can be run anytime you make a change to the program without having to manually carrying out a test in interactive mode. To run in script mode:

```
./filesys –s <script_name>
```

In both modes, the disk will be mounted at the beginning of the program, the disk will be mounted. The disk is stored in the filename "DISK". If the disk file exists, it will use those disk contents. If the disk file does not exist, it will create a new disk file and properly initialize block 0 (superblock) and 1 (home directory). The current directory is always the home directory at the start of the program.

The disk is persistent across different runs of the program. In some cases, you may want to start with a fresh disk, simply remove the file DISK. The Makefile will also automatically remove the disk when you recompile.

Implementation Notes

- The size of the block data structures are 128 bytes on cs1 and should be 128 bytes on many other systems. If you are concerned with portability problems, uncomment the sanity check code at the beginning of `main` to double check.
- Unlike data files, the names of files and subdirectories stored in directories are null terminated.
- Neither `get_free_block` nor `reclaim_block` initializes or clears out the corresponding block in any way. Your implementation should not rely on blocks being "empty".
- Be sure that `rmdir` and `rm` actually reclaim blocks that are part of the deleted directory or file. This can be tested for by removing a file, creating a new file, and running `stat` on that new file to see if the block is indeed reused. This test is possible since `get_free_block` deterministically returns the free block with the lowest number.

Error Checking

The program must PRINT the following error messages when appropriate:

- `File is not a directory` (Applies to: `cd`, `rmdir`)
- `File is a directory` (Applies to: `cat`, `append`, `rm`)
- `File exists` (Applies to: `create`, `mkdir`)
- `File does not exist` (Applies to: `cd`, `rmdir`, `cat`, `append`, `rm`, `stat`)
- `File name is too long` (Applies to: `create`, `mkdir`)
- `Disk is full` (Applies to: `create`, `mkdir`, `append`)
- `Directory is full` (Applies to: `create`, `mkdir`)
- `Directory is not empty` (Applies to: `rmdir`)
- `Append exceeds maximum file size` (Applies to: `append`)

After printing the error, return from the function. Do NOT exit the program. In addition, all errors should be detected before any writes are made to the disk.

Grading and Other Requirements

The assignment will be graded with the following rubric. Most tests will be graded all-or-nothing where you receive all the points if the test passes and no points if the test fails.

mkdir — *12 points*
- 3 tests — (4 points each)

ls (proper formatting) — *3 points*
- Does not list "/" after directories or incorrectly lists "/" after files — -2
- Other formatting issues — -1

cd / home — *8 points*
- 2 tests — (4 points each)

rmdir	*12 points*
• 3 tests	(4 points each)
create	*12 points*
• 3 tests	(4 points each)
append / cat	*20 points*
• 4 tests	(5 points each)
stat (proper formatting)	*3 points*
• Does not have proper formatting for directories	-1
• Does not have proper formatting for files	-2
rm	*12 points*
• 3 tests	(4 points each)
error checking	*18 points*
• 9 tests, one for each error	(2 points each)

Additional grading notes:

- Many tests have dependencies on other functions. If those functions fail, then the test will have considered to fail. Here are some dependencies (this is not an exhaustive list):
  - Most directory tests depend on `mkdir`.
  - Most file tests depend on `create`; some `rm` tests also depend on `append`.
  - Many tests use `ls` and/or `stat` for output. Note that the rubric above for `ls` and `stat` is for proper formatting. If they produce incorrect output, many tests will fail.
  - Most error checking tests depend on other functions.
- Up to 10 points may be deducted if programs if they exhibit poor readability or style (including producing extra output). See the programming assignment expectation handout.

Submitting your Program

On cs1, run the following script in the directory with your program:

```
/home/fac/elarson/submit/cpsc3500/pa3_submit
```

This will copy the files `FileSys.cpp` and `FileSys.h` to a directory where that can be accessed by the instructor. Please be sure to keep the same file names or the submission program will not work.

The submission program will attempt to compile your program using the other provided source code files and the provided `Makefile`. Your program must properly compile or the submission program will reject your program. Programs that fail to compile will not be graded. Only the last assignment submitted before the due date and time will be graded. ***Late submissions are not accepted and result in a zero.***