Programming Assignment 2

Due Oct 14 by 10pm **Points** 100

Overview

Our regular use of computers (in various forms) includes interactions (often indirectly) with a file system or a file system-like structure. This course includes exploration of the systems programmer's interactions with the file system through a variety of system calls. For this assignment, however, you are asked to emulate a simplified file system using actual files (to reduce complexity). More specifically, your program will support directories and "regular" files, allow a user to navigate the file system (change between directories), and allow the user to modify the file system (e.g., add/move directories and files).

This assignment should serve to further hone your C programming skills and to provide a new perspective on files and directories.

Course Learning Objectives

This assignment addresses the following course learning objectives.

Direct

- Read and write complex programs in the C programming language.
- Transition to upper-division coursework where programming is a tool used in the exploration of course-specific content.
- Transition from following stepwise, prescribed decompositions to independent problem solving.

Indirect

- Use standard UNIX user-level commands and the UNIX software development environment.
- Discuss the architecture of the UNIX operating system from a system-programmer’s perspective and be able to write programs that use operating system services (system calls) directly.

Setup

Clone the repository.

- Accept the **GitHub Classroom Assignment 2 invitation (https://classroom.github.com/a/Qv4cQpY_)**.

There are two provided example file systems. One (empty) contains only the root directory. The other (fs) contains a few files and directories as demonstrated by the following commands. I **strongly** recommend that you copy one of these and test with the copy while developing to avoid corrupting the provided files (I speak from experience); of course, you can restore the file system from the git repository as well.

```
> ls
0 .
0 ..
1 357
> cd 357
> ls
1 .
0 ..
2 assignment2
3 final
6 assignment3.tex
> cd final
> ls
3 .
1 ..
4 final.tex
5 code.c
>
```

Requirements

There is, in effect, only a single task, so the following outlines the requirements for your program.

Functional Overview

One approach to a file system is to store information (metadata) that pertains to a "file" (e.g., a regular file or a directory) in an **inode** **(https://en.wikipedia.org/wiki/Inode)**. This metadata includes information such as the physical location of the file's contents on the storage medium (e.g., hard disk), the user that owns the file, the access rights to the file, the file's timestamps, etc. We will ***not*** emulate all of this.

Of note, the metadata for an inode does not contain the name of the file. Instead, a directory maps names to inodes. But a directory is also a "file". This is what we will emulate.

Some Specifics

Your program will store a fixed number of "inodes" (which is not unreasonable when emulating a physical device). The metadata for these "inodes" will contain only a number (an index) and a "type" to indicate a directory ('d') or a file ('f'). The contents of the "file" associated with an "inode" are to be stored in an actual file with name matching the index number (i.e., if your program is meant to access the contents of the "file" associated with "inode" 3721, then it will open the actual file named "3721"; this is a simplification of tracking blocks on a physical medium).

Limit: Your program must support a minimum of 1024 inodes (ranging from 0 to 1023). This is an entirely artificial limit and we will work to remove such limits as the quarter progresses, but given where we are at this point in the quarter, a limit is ok.

A Directory

The contents of a directory will be a sequence of file entries stored as a 32-bit integer (use the `uint32_t` type defined in `stdint.h` to represent an unsigned 32-bit integer) and a sequence of up to 32 characters (in this order). Each file entry will specify, in this order, the inode for the entry and the name of the entry. This format places a limit on the length of entry names (i.e., file names). Shorter names will include a null character within the first 32 characters (but a full 32 charactes must be stored in the file to keep each entry the same size); it is valid to use the full 32 characters for the file name, so your program must account for the lack of a null character in that case. The entries within a directory are stored as the binary representations of each piece of data, so you will want to use `read`/`fread` to extract the data from the file.

The following is an example (shown in plain text, not the exact format stored by your program) of a directory containing the files `.`, `..`, and `example.c` (based on the inodes in this file, can you determine the inode for the directory itself?).

```
13 .
14 ..
27 example.c
```

Initial Directory

Your program will begin in the (assumed) directory at inode 0 (this is meant to be the root of the file system, typically referenced as /). Your program will then take commands from the user to traverse and manipulate the file system (see below).

Requirement 1

Your program must take one command-line argument (a string) specifying the name of a directory that stores the emulated file system (referenced as the file system directory below). As such, your program may be run as follows (assuming that `fs` is such a directory).

```
fs_emulator fs
```

Your program must verify that the specified directory actually exists.

Requirement 2

Your program must load, from the file system directory (consider using `chdir` once for this), the `inodes_list` file. This file includes an entry for each inode currently in use. Each such entry will include the inode number (of type `uint32_t`, as within a directory) and an indicator (a single character, d or f) of the file type. These entries are stored as the binary representations of each piece of data, so you will want to use `read`/`fread` to extract the data from the file. Verify that these indicators are valid. Any inode numbers outside of the range from 0 to the maximum (exclusive) or with invalid indicators should be reported as invalid and ignored.

Requirement 3

With the emulation initialized, the user will start in the directory at inode 0. If this inode is not a directory (or is not in use), then the program should report an error and terminate. The program will need to track the current working directory (initially 0).

Requirement 4

Your program must take user commands as input from `stdin`. The commands and their behavior are listed below. If the user attempts to run any other command, or if a command is provided with the wrong number of arguments, then report an appropriate error and wait for the next command.

Limit: The use of file names in the commands discussed below must be restricted in length to properly interact with the represented directories. In particular, if your program allows a user to enter an arbitrarily long name (e.g., when executing `mkdir`), then it will need to truncate the name to the maximum length of 32 (specified earlier).

- **exit** - (or EOF; ^D in Unix). Save the state of the inode list (see the next requirement) and exit the program.
- **cd <name>** - Change to the specified directory (by name) from the current directory. The name must be given. Your program does not need to support paths (i.e., `cd lab2` is valid, but `cd lab2/task1` is not). If the specified name is not a directory (does not exist or is a regular file), then report the error and do not change the current working directory.
- **ls** - List the contents (names and inodes) of the current working directory.

- **mkdir <name>** - Create a new directory (in the current directory) with the given name. Doing so will require that there is an available inode and that the contents of this directory are updated. Moreover, the file corresponding to this new inode will need to be created with the default entries of `.` and `..`. Report an error and continue if there is already an entry in the current directory with this name.
- **touch <name>** - Create a regular file (in the current directory) with the given name. Doing so will require that there is an available inode and that the contents of this directory are updated. The real file corresponding to this new inode will need to be created; for emulation/debugging purposes the *name* of the file should be written to the real file (corresponding to the inode number). If there is already an entry with this name, then this command will do nothing.
- ~~**rm <name>**~~ - *You know what? I think the above is illustrative enough. We will not use this file system for anything beyond an improved understanding of the concepts and further development of C programming skills, so a full set of features is unnecessary. As such, you do not need to implement rm. That said ... I encourage you to consider how this might be done.*

Requirement 5

On exit, your program must update the `inodes_list` file in the file system directory to reflect any modifications to the inodes.

Utility Function

It is likely that your program will need to convert an integer into a string. There are a few ways to do this (especially given that the maximum number of digits that will be printed for this program is known to be small), but the following is a pretty common approach.

```
char *uint32_to_str(uint32_t i)
{
   int length = snprintf(NULL, 0, "%lu", (unsigned long)i);       // pretend to print to a string to get leng
th
   char* str = checked_malloc(length + 1);                        // allocate space for the actual string
   snprintf(str, length + 1, "%lu", (unsigned long)i);            // print to string

   return str;
}
```

Deliverables

- Source Code - Push all relevant source code and an appropriate Makefile to your repository.
- Code Review Monitoring - This is not really a deliverable, but be sure to monitor the github messages for comments on your code and then address them as appropriate (certainly meeting with me if anything is unclear or in dispute).

Programming Assignment 2

Criteria	Ratings		Pts
Functionality	**75 pts** **Full Marks**	**0 pts** **No Marks**	75 pts
Code Review	**15 pts** **Full Marks**	**0 pts** **No Marks**	15 pts
Valgrind Check	**10 pts** **Full Marks**	**0 pts** **No Marks**	10 pts
			Total Points: 100