ELE 305 C Project

Submit online on Brightspace and notify me by email (to sendag@uri.edu) as "ELE 305 proj1 submitted". Due 4/17/2025.

This is an individual assignment; you must work alone. Submit your solution, a file named `directory.c`, as `proj1`. Successful completion of this project will add 5% extra to your overall grade.

Goals

This project is intended to give you substantial practice with C pointers and strings, linked structures, and use of `malloc` and `free`.

Background

Most of you have had experience working with the UNIX operating system. UNIX provides a tree-structured file system and operations for navigating around it. In this project, you will work with a command interpreter, implementing commands that create and delete nodes in a tree that loosely *simulates* a UNIX directory. (An actual UNIX file system is organized somewhat differently; see *The UNIX Programming Environment*, by Brian Kernighan and Rob Pike (Prentice-Hall, 1984), for further information.)

The nodes in the tree represent *text files* and *directory files*. A directory file contains zero or more text files and zero or more directory files. A text file contains characters, and corresponds to a leaf in the tree. (An empty directory also corresponds to a leaf.) Each file has a name, a sequence of alphanumeric espace characters. The root of the tree represents the *root directory* in the file system. The *working directory* is also a directory in the file system; the `cd` command (for "change directory") lets the user move the working directory up and down in the tree.

The commands to be supported by the interpreter are described in the table below. They are simpler than their counterparts in the UNIX shell: as noted above, there are only two kinds of files; no command options (normally specified using "–") are allowed; all arguments refer to files in the working directory except as specified in the table; and there is no special handling of "/" within file names.

command	*number of arguments*	*comments*
`pwd`	0	"Print Working Directory" — Prints the full path name of the working directory.
`cd`	1	"Change Directory" — Changes the working directory to that specified by the argument, which must be one of the following: • the name of a directory in the working directory; • the string "..", meaning the parent directory (the root is its own parent); or • the string "/", meaning the root directory.
`ls`	0 or 1	"LiSt files" — If given without arguments, prints the names of files in the working directory in alphabetical order; if given with the name of a text file in the working directory, prints that name; if given with the name of a directory in the working directory, prints the names of files in that directory in alphabetical order by name; otherwise prints nothing.

`cat`	≥ 1	"conCATenate" — Prints the contents of the text files named by the arguments.
`mkdir`	1	"MaKe DIRectory" — Creates a directory with the given name. The working directory must not already contain a file with that name.
`rmdir`	1	"ReMove DIRectory" — Removes the directory with the given name. The working directory must contain the named directory, and the named directory must be empty.
`create`	1	Creates a "text file" with the given name, then reads the contents of the file from standard input. The working directory must not already contain a file with the given name.
`rm`	1	"ReMove" — Removes the named text file from the working directory. The working directory must contain the named file.
`cp`	2	"CoPy" — Creates a *copy* of the first argument in the working directory; the second argument specifies the name of the copied file. If the first argument names a directory, the copy will contain copies of the directory's files and subdirectories. (I.e. it is a *deep copy*.) The working directory must contain the file named by the first argument, and must not already contain a text file whose name is the second argument.
`mv`	2	"MoVe" — If the second argument names a directory, moves the first argument into the directory; the directory must not already contain a file whose name is the first argument. If the working directory does not contain a file whose name is the second argument, changes the name of the file named by the first argument to be the second argument. The working directory must not contain a file whose name is the second argument.

Table of commands to be supported by the project 1 command interpreter

The compressed file cproject.zip contains three files: `dirmain.c`, `directory.h`, and `directory.c`. The main program in `dirmain.c` repeatedly reads a command from standard input, checks that it is syntactically correct, and if so calls one of the functions in `directory.c` to execute the command. The file `directory.h` contains declarations for functions that access and modify the directory tree. The file `directory.c` contains the actual definitions of these functions, as well as a `struct entryNode` declaration that represents a file (directory or text file). It also provides complete implementations for commands that don't change the directory structure, namely `cd`, `pwd`, `ls`, and `cat`. Don't change these functions or the `struct entryNode` declaration. *You are to submit only your `directory.c` file.* Changes to other files will not be accepted.

Each `entryNode` stores the name of the corresponding file (that is, a pointer to the file name's first character), a pointer to the file in the same directory whose name is next in alphabetical order, an indicator of whether the file is a text file or a directory file, a pointer to the parent node, and a pointer to the contents of the file (text for a text file, a list of files, ordered alphabetically by file name, for a directory file). Here is the `entryNode` declaration.

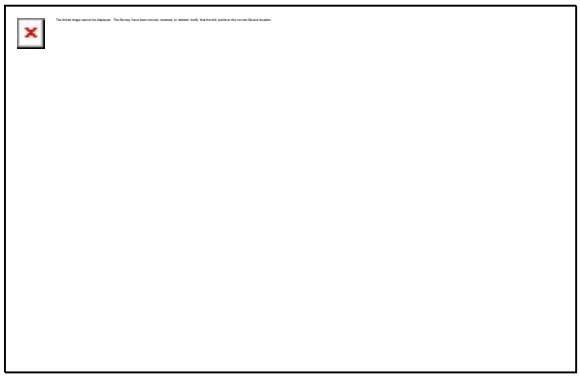
```
struct entryNode {
    char * name;
    struct entryNode * next;       /* sibling */
    int isDirectory;
    struct entryNode * parent;

    union {
        char * contents;           /* for a text file */
        struct entryNode * entryList;     /* for a directory */
    } entry;
```

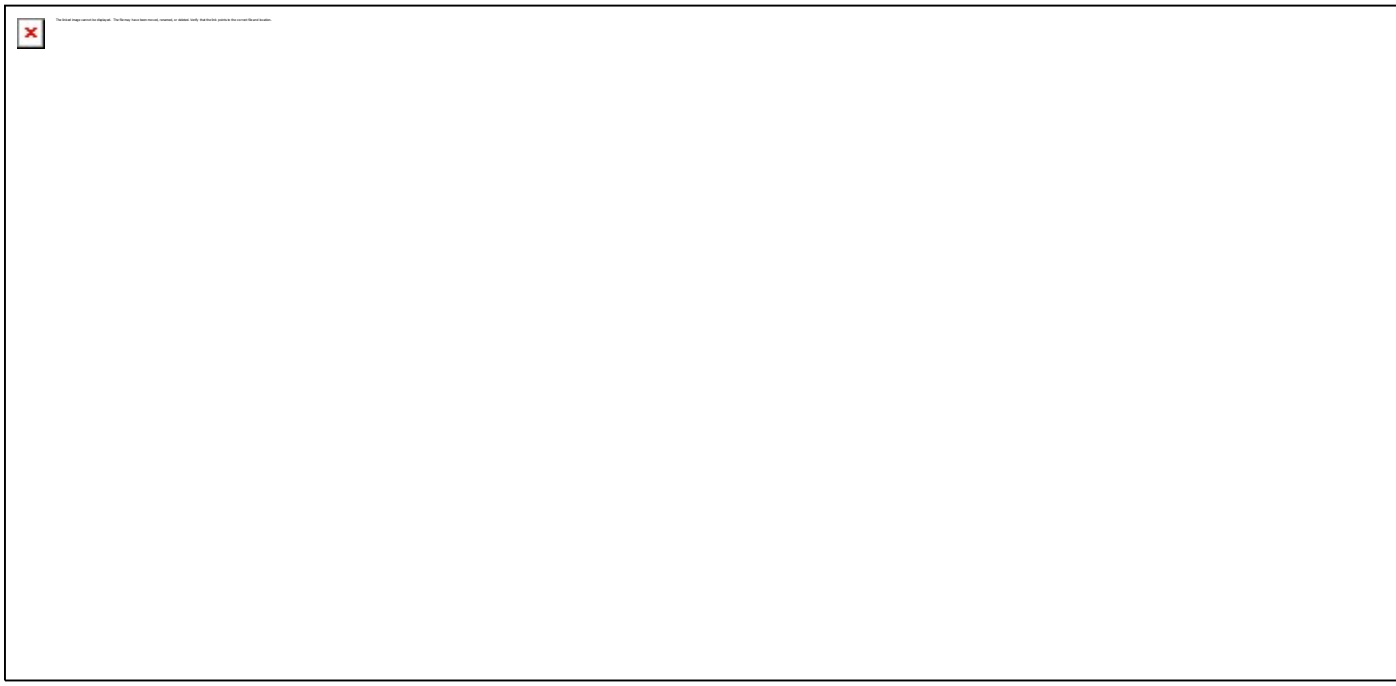
```
};
```

Figure 1 contains a sample directory and its representation in `entryNodes`.

A sample directory:



Its implementation:



Project

Implementation of the `pwd`, `cd`, `ls`, and `cat` commands have been provided in the `directory.c` file for you. You are to complete the program, providing code to initialize the file system and implement the `mkdir`, `create`, `rmdir`, `cp`, `mv`, and `rm` commands. You may provide auxiliary functions. Detailed information about the behavior of these commands follows.

file system initialization	returns a pointer to a directory `entryNode` with name = "/", no siblings, parent = itself, and no children.

`createDir`	creates an `entryNode` for the new directory, copying the argument string into dynamically allocated space for the directory name, then adds the new file to the working directory's `entries` list.
`createFile`	creates an `entryNode` for the new text file, copying the argument string into dynamically allocated space for the file name. It then reads the contents of the simulated text file from standard input, and stores it into suitably allocated dynamic storage. (You should not assume any bound on the amount of text.) Two consecutive carriage returns (that is, '\n' followed immediately by another '\n') indicates the end of this text. The first is stored in the `contents` string, followed by a null character; the second carriage return is not stored. The new file is then added to the working directory's `entries` list.
`removeFile`, `removeDir`	removes the named file from the working directory's `entries` list, and frees all dynamically allocated storage associated with the file.
`moveFile`	removes the named file from the working directory's `entries` list, renames it, and inserts it either back into the working directory's `entries` list or into the specified subdirectory's `entries` list. Checks to ensure that the move does not replace an existing file.
`copyFile`	makes a deep copy of the named file, supplies a new name, and inserts the copy into the working directory's `entries` list. Checks to ensure that the copy does not replace an existing file.
Functions in `directory.c` to be completed	

Tests

Below is a mostly complete list of the tests for project 1. It isn't comprehensive, but it should give you a very complete idea of what we'll be testing for, and it covers mosts of our test cases (we're not trying to deliberately leave anything off the list). You should feel free to write your own automated tests for these cases, and for any potential problem cases. In general, any time you find and fix a bug in your code, you should create a test case which will automatically catch that bug to ensure that no matter what you do afterwards, the bug cannot ever "come back".

- Create a file and then check its contents.
- Create multiple files and then check their contents.
- Create a directory, enter and leave it. Check pwd along the way.
- `cd` to a nonexistant directory.
- `cd` to a file.
- `ls` of a directory, and of a subdirectory.
- Overwriting a file with `create`.
- `mkdir` a directory whose name already exists.
- `rmdir` a non-empty directory.
- `mv` a basic file, and a directory.
- `cp` file, and directory.
- Check for handling of upper and lower case file names, and proper sorting.

How to run an automated test against your project ? Shown below are two commands as an example. The first will send the contents of `tests.in` into your project, which we assume is named `directory` for this example (`gcc ... -o directory`), and directs the output to `tests.out.dirty`. The second command will show you any differences between the `tests.out` and `tests.out.dirty` files. Note

that you might want to have more than one set of tests for your project, in which case you might call them `tests.1.in`, `tests.2.in`, etc, and you may wish to write a script or program to run them all.

```
ephesus> cat tests.in | directory > tests.out.dirty
ephesus> diff -bi tests.out tests.out.dirty
```

Similar commands can be used to test programs which take command line arguments: `cat tests.in | xargs yourprogram` will run your program once for each line of `tests.in`, making that line of `tests.in` the command line arguments for your program. Type `man xargs` on linux machines at the department for more information about `xargs`, a very handy program.

Summary of differences between commands for this project and "real" shell commands

None of the commands to be implemented in this interpreter take any options. Input/output redirection is not allowed, and arguments to all commands except `cd` must be simple file names, each consisting of one or more alphanumeric characters. Other differences are described in the table below.

command	*differences*
`pwd`	No differences.
`cd`	The working directory can be changed only to the root directory, the working directory's parent, or one of the child directories of the working directory.
`ls`	Takes only 0 or 1 argument. Directories are not sorted before files.
`cat`	No differences.
`mkdir`	Takes only one argument.
`rmdir`	Takes only one argument.
`create`	No "real UNIX" counterpart.
`rm`	Takes only one argument.
`cp`	Takes only two arguments; does not allow overwriting an existing file; does recursive copying of directories (enabled by the `-r` option of the real `cp`.
`mv`	Takes only two arguments; does not allow overwriting an existing file.
Differences between commands in this interpreter and "real" UNIX commands	

Example Transcript

Shown below is a transcript of commands and their responses exactly as you would see them from a working project. This is not a comprehensive test suite, but rather an example meant to help clarify this project. The transcript assumes that it is starting in directory `d`, of the above example diagram.

```
> ls
e
f
> cd e
> ls
g
h
i
> cd ..
> create file1
```

```
text in file1

> ls
e
f
file1
> mv file1 e
> ls
e
f
> cd e
> ls
file1
g
h
i
> cat file1
text in file1
>
```