

Assignment 2

Due Feb 12 by 4pm **Points** 10

Assignment A2: Structs and Dynamic Memory

Introduction

The goal of this assignment is to practice linked lists, strings, pointers, and dynamic memory allocation in C. In this assignment, you'll write the code to perform a traversal over a file tree.

Background: Files and Directories

Persistent data on computers is stored in files, and to make it easier to find what we need, we organize files by placing them into directories. Directories can contain other directories, so the directory and file structure on a computer can be modelled by a tree. (We don't need a full graph -- i.e., a tree is sufficient -- since a directory can only have one "parent" -- one directory that contains it.) For this assignment, you'll write code to traverse such a "file tree" and to collect data about the files and directories that are encountered.

Let's use the command line to explore a few directories. We'll use `ls` to list the files in the solution directory for assignment 2:

```
$ ls -lai
total 32
43268828 drwx----- 3 campbell instrs 4096 Jan 22 10:14 .
43259711 drwx----- 22 campbell instrs 4096 Jan 14 16:22 ..
34362726 -rw-r----- 2 campbell instrs 4113 Jan 22 10:12 ftree.c
34362730 -rw-r----- 1 campbell instrs 1014 Jan 22 10:12 ftree.h
34362664 -rw-r----- 1 campbell instrs 254 Jan 22 10:12 Makefile
34362738 -rw-r--r-- 1 campbell instrs 309 Jan 22 10:12 print_ftree.c
43268830 drwxr-x--- 2 campbell instrs 4096 Jan 22 10:22 temp
$
```

`ls` shows us that the current directory contains seven entries. Each file has a lot of information associated with it. From left to right, `ls` shows us the inode number (the OS's unique identifier for a file), type and permissions (what type of file it is and who can access it), the number of links to the file (the number of entries in directories which reference that file), who owns it (campbell), what group the file is in, the size of the file, the modification date, and the filename. The OS actually stores a few more pieces of data -- like the last file access time -- but the options we specified for `ls` didn't request those pieces of information.

Much of this information won't be too important to you for this assignment, but you *do* need to know if a file is a directory, a link, or a regular file. In this case, three of the entries above are directories (they have a "d" at the front of the permissions, which is the second group of characters from the left), and four are regular files. There are no links. Let's check the `temp` directory to see if we can find one:

```
$ ls -lai
total 20
43268830 drwxr-x--- 2 campbell instrs 4096 Jan 22 10:22 .
43268828 drwx----- 3 campbell instrs 4096 Jan 22 10:14 ..
34362726 -rw-r----- 2 campbell instrs 4113 Jan 22 10:12 ftree_linked.c
43268832 -rw-r----- 1 campbell instrs   14 Jan 22 10:19 hello.txt
43268838 lrwxrwxrwx 1 campbell instrs    1 Jul 29 15:56 temp_link -> .
$
```

This example shows what a link looks like. `temp_link` is an alias for `../temp` -- for the directory we did an ls on! Does that break our ability to model file trees as trees? (Have we just found a cycle in our file tree?) Fortunately, no. The link isn't a directory: links like these (*symbolic links*) are "aliases". They are files that contain a *path* (a sequence of file names that specifies a location in a filesystem).

There's actually another link in that picture. Did you notice that the inode (the first number on the left) for `ftree_linked.c` is the same as the inode for `ftree.c` in our first example? That means that both filenames refer to the *same data on disk*. This is more than a symbolic link: this is a *hard link*. Directories cannot have hard links: this preserves the requirement that each directory have exactly one parent.

Did you notice the directories `."` and `.."` in both directories we looked at? These are special names that will show up in every directory. `."` is the current directory. `.."` is the parent directory. These names allow us to navigate through the file system. For example, when we execute `./a.out`, the `.` means that we're looking for `a.out` in our current directory. We've also navigated from one directory to another using `cd ..`, and this command changes our current directory to the parent of the directory we started from.

So, what have we seen? In the first example, we used `ls` on my current directory. That directory has a parent (we can see that it has a `..`), so it is not the *root* (the base) of the entire file structure on my disk. However, it is the *root of its own file tree*. It contains seven entries, two of which are itself and its parent directory. One of the entries it contains is a child directory, and when we look inside that directory, we found examples of links. So, *the file tree we examined in these examples has depth three*. The top level contains `.` and nothing else. The next level includes the files accessible from `.`, excluding itself and its parent. (There are 5 nodes on the second level.) One of these nodes, `temp`, is a directory, and the third level of the tree includes the files in `temp`.

Try drawing the tree represented by our examples. Then, take a look at some of your own directories. Can you draw the directory trees?

Accessing File Information



A *filesystem* is an operating system module that encapsulates the data structures used to track files on a

disk and the functions used to manipulate those data structures. Fortunately for us, we don't need to know much about those structures to use files. Instead, we simply need to understand the *system calls* that the operating system exposes for users to access the file system and the *macros* that C provides to make using those system calls easier. (If you're curious about how filesystems are implemented, you'll learn a great deal more about them in CSC369.)

The system call you'll need for this assignment is `stat`, and you'll call it using the [C library function `lstat`](https://linux.die.net/man/2/stat) (<https://linux.die.net/man/2/lstat>). (Take a moment to read the man page that was just linked. Keep it open as a reference while you code, too.) `lstat` fills in a struct `stat` with information about the file you ask about. Note that you'll need to include several header files to get everything you need for `lstat`. (The man page lists `sys/types.h`, `sys/stat.h`, and `unistd.h`.)

The `st_mode` field of the struct `stat` is particularly interesting, since it can tell you whether a file is a directory or link. A series of macros (and constants) are provided (check the man page!) to help you extract data from the field: `S_ISREG`, `S_ISDIR`, and `S_ISLNK` will be useful in this assignment. It's also useful to know that the *permissions* on the file can be extracted by applying a *bitwise and* (`&`) to the `st_mode` field and the octal value `0777`.

In addition to extracting data about individual files, you'll need to be able to get all of the files within a directory. [The `opendir`](https://linux.die.net/man/3/opendir) (<https://linux.die.net/man/3/opendir>) and [the `readdir`](https://linux.die.net/man/3/readdir) (<https://linux.die.net/man/3/readdir>) functions will let you iterate through the files in a directory.

Here is some demo code using `lstat`, `opendir`, and `readdir`: [lstat demo.c](#)  and [readdir demo.c](#) . We'll do a similar demo during the Week 5 lectures.

Tasks

This is a tree assignment. You will need to write a function that constructs a tree (an `FTree`) representing a file tree rooted at the file specified by the user. You will also write a function to walk an `FTree` and to print some of the file information stored in it. For example, here is the output of our solution run on the example directory from the previous section (see below for information on the order that items appear):

```
$ ../print_ftree .
===== . (d700) =====
  ftree.c (-640)
  Makefile (-640)
===== temp (d750) =====
  hello.txt (-640)
  temp_link (l777)

  ftree_linked.c (-640)
  ftree.h (-644)
  print_ftree.c (-644)
$
```

Here is an example run on a single directory (that contains links):

```
$ ../print_ftree temp
===== temp (d750) =====
hello.txt (-640)
temp_link (l777)
ftree_linked.c (-640)
$
```

Here is an example run on a regular file (a link would be similar, but with **l** (lowercase L) instead of **d**):

```
$ ../print_ftree temp/ftree_linked.c
temp/ftree_linked.c (-640)
$
```

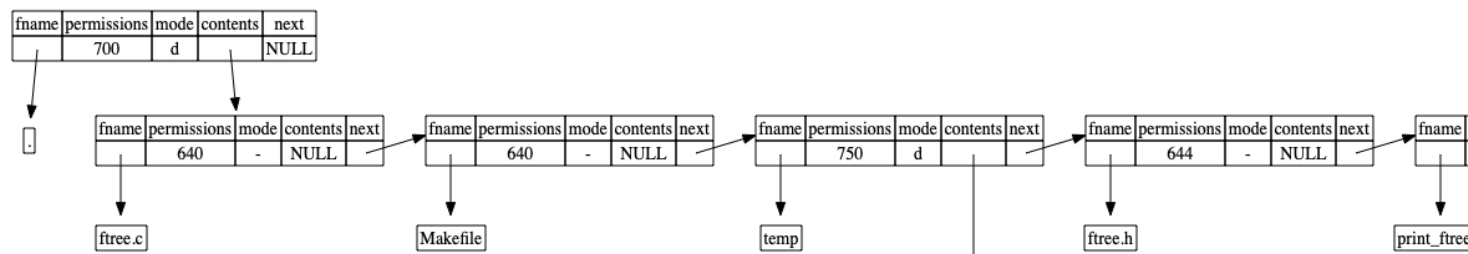
And finally, here is an example run on a file that doesn't exist (use the `fprintf` provided in the starter code to output following message to `stderr`):

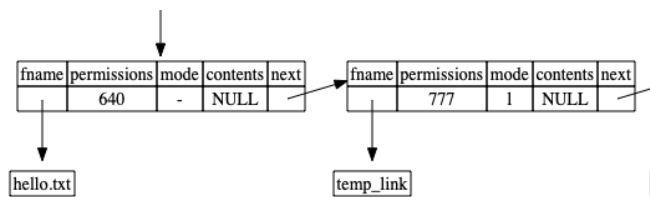
```
$ ../print_ftree nofile
The path (nofile) does not point to an existing entry!
$
```

To summarize: The root of the FTree being created is the file specified on the command line. The program recursively walks the file system from that starting point and creates a FTree that models the file system structure. Each node of the FTree is a `TreeNode` and represents a file (regular, directory, or link) encountered in the walk. Specific pieces of information about the file must be stored in each `TreeNode`:

- The name of the file.
- The permissions on the file.
- If the file is a directory, the linked list of `TreeNode`s that represent the files in the directory. No filename that starts with a `.` should be included. **The order of files in the list must be the same as the order used by `readdir`.**
- The type of the file (`d`, `-`, or `l`) (i.e., a lowercase D, a hyphen, or a lowercase L)

Here is a visual representation of the FTree for directory `.` shown above (again, the order of `TreeNode`s corresponds to the order returned by `readdir`):





While the FTree must track all of the information, your program will not print all of it. The program must provide exactly the output specified (as described in the examples above). In particular:

- Each directory should be identified using equals signs. (There are five in the front and five in the back.)
- The contents of a directory should be indented two spaces more than the parent directory.
- Every line should contain the name of the file/directory followed by the permissions (in parentheses, as three octal digits).
- The order should be the same as produced by `readdir`.

Starter Code

Most of the structure of the assignment has been provided in the starter files. Note that we have provided both header (.h) files, which declare the existence of functions and global variables, and source (.c) files, where functions and variables are actually defined. The only starter file you may *modify* is `ftree.c`.

In addition to source and header files, the starter code also includes a Makefile. This file contains commands for building your program. To build the program, just run `make` on the command line. To erase all of the intermediate files and the executable, run `make clean`.

Tips

man pages are the best source of information for the system and library calls you need. Start by reading them carefully. Then, construct short programs to try out the syscalls you need, to develop an understanding of how they work. For example, you may want to start by writing a program that calls `lstat` on a single file and prints all of the information that you need to store in a `TreeNode`. Later, you may want to write a program that checks if a file is a directory and, if it is, uses `opendir` and `readdir` to print the names of all the files in the directory. After you've constructed these test programs, try to incorporate what you've learned into the program you're writing for this assignment.

While only three functions are required, you may wish to create additional recursive helper functions, and you're welcome to do so. However, remember that none of the files except for `ftree.c` should be modified.

Build up functionality one small piece at a time and commit to git often.

Check for errors on system calls (see marking section for more details). In the case of a failure, print a message to `stderr` (not to `stdout`) and exit immediately with a return code of 1.

Be very careful to initialize pointers to NULL and to make sure that you copy any strings that you use.

Marking

We will use testing scripts to evaluate correctness. As a result, it's very important that your output matches the expected output precisely. Unlike A1, which was marked only for correctness, this assignment will be marked not only for correctness, but also for style and design. For this and future assignments, your code may be evaluated on:

- **Style:** at this point in your programming career you should be able to make good choices about code structure, use of helper functions, variable names, comments, formatting, etc.
- **Memory management:** your programs should not exhibit any memory leaks. This means freeing any memory allocated with malloc. This is a very important marking criterion. Use valgrind to check your code.
- **Error-checking:** library and system call functions (even malloc!) can fail. Be sure to check the return values of such functions, and terminate your program if anything bad happens.
 - For A2, you are required to error check library and system calls (e.g., malloc, opendir, `closedir`, etc.)
 - However, for A2, you are **not** required to use errno to check system calls that depend on errno.
- **Warnings:** your programs should not cause any warnings to be generated by gcc -Wall -std=gnu99.

MarkUs Checker

We have provided a checker program for Assignment 2 on MarkUs. There are three checks for each of the functions generate_ftree and print_ftree. You can run the checks yourself up to once per hour on MarkUs (if server load becomes an issue, we may need to increase the time between runs). After the deadline, we will run a larger set of tests on your submission for marking, so you should test your code thoroughly and not rely solely on the checker. Note that these checks are only for correctness and do not check for issues related to the other marking criteria.

Submission

Please commit to your repository often. We will look at ftree.c, so that file must be pushed for us to successfully test your code. You must *NOT* change any of the other .c or .h files, or the provided Makefile. We will be testing your code with the original versions of those files. We will run a series of

additional tests on your full program and also on your individual functions (in ftree.c) so it is important that they have the required signatures. Do not add executables or test files to your repo.

Remember to also test your code on teach.cs before your final submission. Your program must compile cleanly (without any warning or error messages) on teach.cs *using our original Makefile* and may not crash (seg fault, bus error, abort trap, etc.) on our tests. Programs that do not compile or which crash will

be assigned a 0.