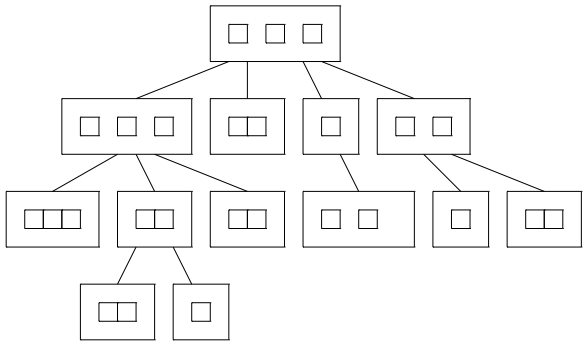
csci-e28
Assignment 2: pseudofind

Introduction

For this assignment you will write a program that implements part of the Unix **find** command. In doing so, you will have a chance to work with the Unix directory structure, recursion, and the stat() system call.

Directory Trees: The Big Idea

Every modern operating system organizes storage media (disks, flash drives, CDROMs, DVDs, etc) into a tree of directories. And every modern system provides tools for searching those trees. Here is a picture of a directory tree:



The top directory in the diagram is the root of the tree, and the boxes below that are subdirectories. The little rectangles inside the directories represent files. A directory tree can contain many, many subdirectories and many, many files. On a Unix/Linux system, a single directory tree provides access to all disks and all input/output devices.

Directory Trees: Operations

What sort of operations can one perform on a directory tree and its contents? Unix provides a wide variety of tools to analyze and process directory trees and their contents. In fact, many regular file processing tools can be told to process an entire tree of files. Some examples are:

COMMAND	ACTION
`ls dirname`	list contents of dirname
`ls -R dirname`	list contents of tree with root dirname
`rm filename`	remove file called filename
`rm -r dirname`	remove entire tree with root dirname
`chmod o-r filename`	make filename unreadable to other users
`chmod -R o-r dirname`	make all things in tree unreadable to others
`cp f1 f2`	copy the single file f1
`cp -r dir1 dir2`	copy the entire tree with root dir1

These four commands support a **-r** or **-R** option (stands for recursive) that performs the operation for all items in and below the specified directory.

Learning to use recursive versions of file and directory tools is an essential part of understanding and using Unix.

☞ PART ONE: Name Three More Recursive Tools

The first part of this assignment involves no programming, just some simple research. Name and describe three more Unix programs that operate on directory trees. Do not include **find** or the four commands mentioned in the previous section. For each of the three you mention, state the name of the program, write a sample command line, and write a one sentence description of what the program does.

Similarities and Differences

Most directory processing programs have the same basic idea. The programs visit each directory in the tree, and they perform some action on each directory and/or file they encounter. Thus, there are two parts to the code: the tree traversal and the specific action. The tree traversal logic is pretty much the same from tool to tool; the specific action is the part that varies. Therefore, once you learn to write one tree-processing program, you have done half the work for several other programs.

Explanation of find

Imagine you have set up a complex, multi-level directory structure to organize your files. Directories contain subdirectories which in turn contain subdirectories of their own. Everything is in perfect order, but suddenly you realize that you mislaid a file called wtmplib.c .

With the find command, you could type:

```
$ find . -name wtmplib.c
```

The Unix **find** command searches directories and all their subdirectories for files that fit certain specifications. In the example just mentioned, find is used to search by name. Find also allows you to search for files by modification time, by type, by size, by owner, by number of links, *etc*, as well as combinations of attributes. Read the manual page on find for more info.

☞ PART 2: Writing Simplified find

The actual find command is not too tricky to write once you get the general structure in place. Rather than focus on the details of all the different search criteria, your version will support only two of the search criteria: name and file type. The syntax of your version of find, (pfind for pseudo-find), is:

```
$ pfind starting_dir [-name filename-or-pattern] [-type {f|d|b|c|p|l|s}]
```

Here are three examples:

```
$ pfind /home/s/smith -name core
$ pfind . -name '*.c' -type f
$ pfind cscie28/hw
```

The first command searches /home/s/smith and all its subdirectories and prints the full pathnames of all items called core. This is an example of calling find with one criterion -- a filename.

The second command shows calling pfind with two criteria -- a filename pattern and a file type. This command searches the current directory and all its subdirectories and prints the paths to things that have names ending with **.c** and are files.

The third command is an example of calling find with no criteria -- all items match the search rules. Therefore, this command prints the paths to all files and directories in and below the directory called cscie28/hw . **pfind**, like the real **find**, does not stop until it has searched all subdirectories.

NOTES

- The order of the criteria does not matter
- It is a syntax error if either criterion appears more than once
- Exactly one starting directory must be specified
- Your solution may not use the ftw library function or its relatives

Getting Started

1. Read the manual page on the find command. Try it out in your own directory or in the lib215 directory. For example, you might try:

```
$ find ~lib215 -name Makefile
```

2. cd to ~lib215/hw/pfind and explore the directory trees under pft.d and pft2.d . Search for things named *cookie* under pft.d, and search for things called *d8* under pft2.d . Use the find command to perform these two searches. Your program will need to perform the same search and produce the same results.

3. Explore the manual to find a function to help you match filename patterns.

4. Once you see how real find works and have worked through the procedure manually, think about what your find has to do. It has to look at each entry in the starting directory. If the entry matches **all** criteria specified on the command line, print the directory name with the file name appended. If the entry is a directory your program will have to do the same operations on that subdirectory. Therefore the search and print function will be recursive.

5. You need to use lstat(2) to determine the type of a file. Read the text, online documentation, or header files. This step is an essential part of your program.

6. Write a function called searchdir(char *dirname, char *findme , char type). This function will open the directory, read it entry by entry, print out any names of items that match *all* search criteria. If `findme´ is not null, then the function will only list files that match that pattern. If `type´ is not '\0', then the function will only include items that are of the type specified.

In addition, if any entry in the directory is a directory, then searchdir() will have to create a new string to hold the new directory name and pass that string to itself. Use malloc() to create the string; a fixed size buffer is liable to run out when you least expect it.

Building Your Program

Write your own file or files of code. Be sure to write a Makefile and an explanatory document.

Testing Your Program

You can make up your own test system data, but you must test it sometime with ~lib215/hw/pfind/test.pfind . This shell script will exercise your program on a preset directory tree. To run it, type:

```
$ ~lib215/hw/pfind/test.pfind
```

☞ PART 3: The Open File Limit

There is no limit on the depth of a directory tree; any directory can contain subdirectories. The burrowing shell script example from class shows how to create extremely deep structures.

Each time you open a directory in the recursive calls, your progam has another open file. Unix usually sets a limit on the number of open files a program can have. Once that limit was 20, then 60, now it is much greater. Nonetheless, each open directory requires some system resources. It is a good practice to program with use and limitations of system resources in mind. In particular, plan your programs so they do not crash if they run out of resources (here, too many open directories) and also so they do not use more system resources than they need. In this example, if your program has too many directories open, the `opendir()` call returns NULL.

Describe a solution to this limit. Devise a method that will allow your program to process directory trees that are deeper than the maximum number of open files.

What to Turn In

Hand in

1. A ReadMe text file with
   a) A summary of what you are submitting
   b) the list of three tools that process directories
   c) a short answer to the *Open File Limit* question,

2. your source code

3. a typescript of its output from the test.pfind test script and any other samples you wish to include

4. a Makefile

5. a design outline

Prepare these files in a directory then type

```
~lib215/handin pfind
```