Assignment 2: Structs, Files and Dynamic Memory

Introduction

The goal of this assignment is to practice system calls and dynamic memory allocation. You'll be writing a program that explores the `/proc` virtual file system to generate (and display) a tree of the processes (the executing programs) on the system. To do so, you will need to dynamically allocate and manipulate nodes in a tree structure. The end result will be output that describes the children of a given process, much like the `pstree` command does as show in this example:

```
$ pstree -p 1123
sshd(1123)---sshd(25914)---sshd(25987)---bash(25988)---pstree(1243)
           |-sshd(27395)---sshd(27468)---bash(27469)
           |-sshd(29462)---sshd(29498)---bash(29531)
           |-sshd(29499)---sshd(29529)---sftp-server(29530)
           |-sshd(29874)---sshd(29912)---bash(29913)
           |-sshd(30932)---sshd(30962)---sftp-server(30963)
           |                           |-sftp-server(30964)
           |                           |-sftp-server(30965)
           |-sshd(31511)---sshd(31547)---bash(31548)
```

Getting Started

Pull your git repository. There should be a new folder named `a2`. All of your code for this assignment should be located in this folder and includes a `Makefile`; `print_ptree.c`, which contains the *main* function and will need some additional argument handling code; `test_ptree.c`, which contains a *main* function for testing a single function; `ptree.h`, which contains function prototypes and a struct for a tree node; and `ptree.c`, which contains the signatures for the two functions to be written. There is also a directory for testing your solution with simulated data.

Background

A *process* is an executing program. Every time that you run a program on the command line, you are creating a process. The processes on a Unix system are naturally organized in a tree structure, since each process has a parent. When you execute a progam in a shell (on the command line), the shell process is the parent. The shell process also has a parent (often the `login` that created the shell). Each process knows the identify of its parent as well as some information about itself.

The operating system must track processes to make sure that they are getting the resources they need. For many flavors of linux, the information about processes is stored in the directory `/proc`. (Mac users: note that OS X does not have a `/proc` directory since it's based on `bsd`, which uses a different interface.) `/proc` is very special in that it is a *virtual filesystem*. It doesn't contain "real" files; rather, it contains runtime system information that can be accessed as if it were stored in files. Many system utilities are simply programs that access data from the "files" in this directory. Your task is also to access the data in `/proc` to get information about the processes running on the system.

`/proc` Structure

Every directory in `/proc` that we care about is a number. (There are quite a few non-numeric directories. Those refer to mounted devices, memory information, and other system data. We'll ignore them for this assignment.) That number is a *process ID (PID)* -- a unique identifier for a process on the system. Every process has a PID,

and every process running on the system has an entry in the `/proc` directory. In this document, we'll refer to the entry for a single PID as the directory `/proc/PID`.

You will need to access three paths:

- `/proc/PID/exe` is the link to check to make sure the entry is for a running program. If it exists, then PID is a valid running program.
- `/proc/PID/cmdline` contains information about how the program was called. In particular, you'll need it to get the name of the executable.
- `/proc/PID/task/PID/children` contains the PIDs of child processes. The PIDs are separated by a single space.

Take a few minutes to explore the structure of `/proc`. Start by looking at the `a2/tests` directory in your repository. This directory is a mock version of the `/proc` directory that has been simplified to only include the files that you need for testing. Look through it using `cd`, `cat` and `grep`. Once you've figured out the structure, log onto teach.cs and browse through its `/proc`. To find PIDs for entries that correspond to running programs, you can use the commands `pstree -p` or `ps -u`.

Useful C Functions

To access the data in `/proc`, you'll need to use a number of system and library calls. In addition to functions that we have seen in class and the videos (such as *fopen*, *fscanf*, *fgets*, *malloc*, *strchr* and possibly others), you may need two calls that you may not have seen yet: *lstat* and *sprintf*. You will want *lstat* to determine that a link exists without opening it. You may find *sprintf* helpful for creating filenames. Read the man pages for these functions to see how they work. (We will discuss *stat* and *lstat* in class in week 5 but you should learn about *sprintf* on your own.)

You may **not** use the `system` function or functions like it that allow you to execute commands on the shell. The intent of the assignment is to have you working with system calls -- not to use shell programs that will invoke those system calls for you.

Requirements

You must write two functions, `generate_ptree` and `print_ptree`, that are located in `ptree.c`. You must also write code in `print_ptree.c` to handle the expected command-line arguments and to emit appropriate errors.

`print_ptree.c` Requirements

Your program should emit an appropriate error message (which we are not specifying) to `stderr` and **return 1** if an error is detected in the command line arguments used. The arguments may be incorrect if insufficient or too many arguments are provided or if an option is provided that is not supported. (The only option we will support is "-d".) Your program should **return 0** if it is able to run without error. If there is some sort of error that is not a command line error, **return 2** (for example, if a library call for `generate_ptree` were to fail).

The expected command line usage is already provided in the starter code. The program takes one optional flag, "-d", which requires an argument. This sets the maximum depth of the process ID tree to be displayed. If the depth is not provided, it is assumed to be 0 (which means "print the entire tree"). The program requires one argument, a process ID, which must be an integer. If an integer is provided, but it is not a process ID with an entry in `/proc`, your program should not print anything and should **return 2**. You may assume that any command-line arguments provided for the PID or maximum depth will be non-negative integers.

`print_ptree`

You should complete the `print_ptree` function before the other required function, since you can test it by manually creating small trees for it to print. (See the starter code file `test_ptree.c` for an example of how to test the print function separately.) Then, you can test the other function, `generate_ptree`, by printing the trees it generates.

This function should be recursive. It accepts the root of a tree, as described in the section about `generate_ptree` and a maximum-depth argument. The function prints (to `stdout`) the tree to the desired maximum-depth. If the maximum-depth argument is 0, then the entire tree should be printed.

The function will print one process per line. The root of the tree will be printed flush to the left. Each child process will be indented two spaces to the right with respect to its parent. Child processes should be displayed in the order in which they occur in the list of children. Each line will contain the process's ID (its PID), a colon and a space, and the name of the executable. If the executable is not named, then the line will just contain the PID (no colon, no space, no executable name).

Here is the output of your function on a tree encoding the initial example in this handout (the example in the tests directory of the starter code).

Once you have tested that your `print_ptree` function works properly by testing it on small manually-created trees, then you can move on to write the function that will read the `/proc` filesystem and build the correct tree in memory.

`generate_ptree`

This function should be recursive or use a recursive helper function. It takes a PID as one of its arguments and creates a tree that describes that process and all of its children. The root of the tree is communicated back to the calling function by assigning it to the parameter `root`. The return value of the function will be used to communicate success or failure (described later.) The entire tree, including the name strings in each tree node, should be dynamically allocated using `malloc`, and it must be built as described below.

Each node in a tree is a `struct TreeNode`. It contains the PID of the process it is describing, as well as the name of the executable associated with that PID. If there is no executable name (from `/proc/pid/cmdline`), then the `name` pointer should be `NULL`. The node also contains pointers to build a linked data-structure that will represent the subtree rooted at this PID.

In CSC148, you probably built *binary* trees: trees with two child nodes. In those cases, it makes sense to create links to just those two nodes (often named "left" and "right"). However, in this case, each node in your tree can have an arbitrary number of children: it's not binary. Therefore, we have to use a different method for storing children: a list. If the `child` pointer has value `NULL`, then the node has no children. If `child` is non-null, then the first element of the list is the first (left-most, if you prefer) child. The node pointed to by the first child's `sibling` pointer, is the second child. If a node is the last child of its parent, its `sibling` pointer is `NULL`. The children should appear in the list in the same order as they were observed in the `children` file.

In the example we've been using throughout this handout, the node associated with PID 1123 is the root of our tree. Its `sibling` pointer is `NULL`. Its `child` pointer refers to the node for PID 25914. The node for PID 25914 has a `child` pointer that refers to the node for PID 25987. PID 25914's node's `sibling` pointer refers to the node for PID 27395.

From this same example in the handout, consider the subtree rooted at PID 30932, which is shown in the upper half of this image. If we were to call `generate_ptree` for PID 30932, it should generate the structure in memory shown in the lower-half of the same image. Notice that even though process 30932 has siblings in the full tree (30951 for example), its siblings aren't included in the subtree rooted at 30932 and so its sibling pointer is `NULL`.

`generate_ptree` should **return 0** if the tree is built successfully and 1 otherwise. In particular, it should **return 1** if any of the system or library calls fail. Your function will also **return 1** if `/proc/PID` doesn't exist for the

specified PID or if `/proc/PID/exe` doesn't exist, indicating that the process was not executing. In both of these cases, no node should be created for the PID. If your function returns a 1, then the `root` parameter should still contain as much of the tree as could be built. The tree should be as valid as possible: it should contain no dangling or uninitialized pointers and should contain a node for every PID that could be successfully processed.

Note: You will need to create paths to various files in order to view them. The starter code contains an example that uses `sprintf` to generate paths. The example also relies on a constant, `MAX_PATH_LENGTH`, which contains an upper bound on the length of paths we will test with. In general, defining such an upper bound is tricky, but we guarantee that our tests will not exceed this bound, so you can safely use it to define arrays for storing paths. Remember that a full path to a file includes the filename itself. That entire string (including the filename) will be no longer than `MAX_PATH_LENGTH` in our tests.

Testing

Your `Makefile` contains an option to help you with testing. Open it with a text editor. You should see two lines that define `FLAGS`, one of which is commented out:

```
FLAGS = -Wall -g -std=gnu99
# FLAGS = -Wall -g -std=gnu99  -DTEST
```

If you wish to run tests using your `tests/` directory, rather than the actual `/proc` directory for the system, comment out the first line and uncomment the second. Then, when you compile your code, your program will look for process information in the `tests/` directory rather than in `/proc`. You can add additional examples to your `tests/` directory. We have included the processes necessary to reproduce the example at the top of this handout. You may want to create a few other `proc`-like directory trees that you can use for testing.

Important Note: The tests in the `tests/` directory are likely to have small differences from the entries in `/proc`. For example, the `exe` file in our tests is a regular file, rather than a link, and none of the `cmdline` files include anything other than just the executable name. You should also test on teach.cs with `/proc`, but it's useful to have a clean, replicable example to test on (and if you are on a non-linux machine, you don't have a `proc/` at all) which is why `tests/` will be useful.

While creating new tests -- or while looking for real processes in `/proc` to test with -- you'll find the `ps` command useful. In particular, if you type "*ps -u your_username*", you will get a listing of the processes running on the system that are owned by you. You can open up a second terminal and run commands or open a web browser to get additional processes to test.

Marking

Unlike A1, which was marked only for correctness, this assignment will be marked for not only correctness, but also for style and design. 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. We are using testing scripts to evaluate correctness. As a result, it's very important that your output matches the expected output precisely.

Submission

Please commit to your repository often, so that you don't lose your work. We will look at `print_ptree.c` and `ptree.c`, so those files must be pushed for us to successfully test your code. You must *NOT* change `ptree.h`. We will be testing your code with the original versions of `ptree.h` and `Makefile`. Then we will run a series of additional tests on your full program and also on your individual functions (in `ptree.c`) so it is important that they have the required signature. You may add test files to your repository. Do not add executables.

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.