Project 2

A custom `top` using `/proc`

Intermediate Deliverable Due: February 27th, 2018 @ 11:59pm
Final Deliverable Due: March 6th, 2018 @ 11:59pm

The goal of this project is to implement a `top(1)`-like command for gleaning information about a Linux system. For a general idea of the type of program you'll implement, try running the real `top` executable on an Alamode machine:

```
jthomas@bb136-01 > top
21:12:35 up 12 days, 11:39,  4 users,  load average: 0.00, 0.01, 0.05
Tasks: 165 total,   1 running, 164 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.4%us,  0.2%sy, 11.2%ni, 88.2%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   8133452k total,  3009824k used,  5123628k free,   498784k buffers
Swap: 16385272k total,    42648k used, 16342624k free,  1854228k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
    1 root      20   0 26812 1816  736 S    0  0.0   0:01.60 init
    2 root      20   0     0    0    0 S    0  0.0   0:00.04 kthreadd
    3 root      20   0     0    0    0 S    0  0.0   1:27.20 ksoftirqd/0
    6 root      RT   0     0    0    0 S    0  0.0   0:43.42 migration/0
    7 root      RT   0     0    0    0 S    0  0.0   0:02.36 watchdog/0
    8 root      RT   0     0    0    0 S    0  0.0   0:32.60 migration/1
    9 root      20   0     0    0    0 S    0  0.0   0:00.00 kworker/1:0
   10 root      20   0     0    0    0 S    0  0.0   0:02.25 ksoftirqd/1
```

Also, try reading the `man` page. This is always a good idea when you want to know more about what a program does and how it works.

```
man top
```

Your version of `top` will be implemented in much the same way the actual program is, meaning it will get its data from the `/proc` pseudo-filesystem and use the `ncurses` library for displaying the information.

The `/proc` Filesystem

The `proc(5)` filesystem is a pseudo-filesystem which provides an interface to kernel data structures, commonly mounted at `/proc`.

In a normal filesystem, reading a file results in some sort of an I/O operation. For example, the file systems with which most of us are familiar are those that allow us to store and retrieve data from our hard drives.

However, `/proc` is a *pseudo*-filesystem. When reading a file in `/proc`, the operating system is really only accessing its internal data structures that it keeps in memory. Treating these data structures like files allows them to be accessed using a mechanism almost all developers know how to do. Simply read the files like you would any other file.

An immense amount of information is available via the `/proc` file system. A complete and detailed description can be found by reading the `man` page for `proc`. The `man` page also includes useful information about the variable types you should use when reading in various values.

```
man proc
```

Here are some particularly interesting files and directories (not all of which are entirely relevant to this project):

```
/proc/[pid]
       There is a numerical subdirectory for each running process;
       the subdirectory is named by the process ID.  Each such
       subdirectory contains numerous pseudo-files and
       directories with information about the process.

/proc/cpuinfo
       This is a collection of CPU and system architecture dependent
       items, for each supported architecture a different list.  Two
       common entries are processor which gives CPU number and
       bogomips; a system constant that is calculated during kernel
       initialization.  SMP machines have information for each CPU.
       The lscpu(1) command gathers its information from this file.

/proc/loadavg
       The first three fields in this file are load average figures
       giving the number of jobs in the run queue (state R) or
       waiting for disk I/O (state D) averaged over 1, 5, and 15
       minutes.  They are the same as the load average numbers given
       by uptime(1) and other programs.  The fourth field consists of
       two numbers separated by a slash (/).  The first of these is
       the number of currently runnable kernel scheduling entities
       (processes, threads).  The value after the slash is the number
       of kernel scheduling entities that currently exist on the
       system.  The fifth field is the PID of the process that was
       most recently created on the system.

/proc/meminfo
       This file reports statistics about memory usage on the system.
```

```
       It is used by free(1) to report the amount of free and used
       memory (both physical and swap) on the system as well as the
       shared memory and buffers used by the kernel.  Each line of
       the file consists of a parameter name, followed by a colon,
       the value of the parameter, and an option unit of measurement
       (e.g., "kB").

/proc/stat
       kernel/system statistics.  Varies with architecture.

/proc/uptime
       This file contains two numbers: the uptime of the system
       (seconds), and the amount of time spent in idle process
       (seconds).

/proc/version
       This string identifies the kernel version that is currently
       running.  It includes the contents of /proc/sys/kernel/ostype,
       /proc/sys/kernel/osrelease and /proc/sys/kernel/version.  For
       example:
     Linux version 1.0.9 (quinlan@phaze) #1 Sat May 14 01:51:54 EDT 1994
```

You will need to parse data from most of these files. The `man` page contains helpful information about the For example, it describes the `MemTotal` field found in `/proc/meminfo` as follows:

```
MemTotal %lu
       Total usable RAM (i.e., physical RAM minus a few
       reserved bits and the kernel binary code).
```

The `%lu` in the above example is the format specifier to use with a function like `fscanf`, indicating that the value is represented as an unsigned long integer. If you wanted to read this field, you would need a corresponding variable declared in your program:

```
unsigned long mem_total;
```

You can use any mechanism you would normally use for reading files. However, I would strongly recommend either using a function like `fscanf` or using an `ifstream`. In both cases, make sure you're reading into variables of the correct type.

The `ncurses` Library

`ncurses` is a library that allows developers to write powerful text-based user interfaces that run in terminals. It intelligently optimizes screen changes, such that only text that has changed from one render to the

next are updated.

It is used by many programs, including `top`. You're probably familiar with the general behavior: you run a binary, and it takes over your entire terminal until you exit.

You can find an introduction to the library here:

```
http://tldp.org/HOWTO/NCURSES-Programming-HOWTO/
```

1 The Starter Code

As with the first project, you'll be provided with some initial code to get you started. To create your private Git repository for this project, follow this link:

`https://classroom.github.com/a/BsiQ1zMp`

From your repository?s GitHub page, you can get the relevant URL by clicking on the big green ?Clone or download? button and selecting either the Git or HTTPS version. Replace the placeholder URL below and run the following command:

```
git clone git@github.com:csm-csci442/project-2-GITHUB USERNAME.git
```

The starter code already has various `struct`s declared that accurately reflect the data you'll be reading. They are located in the following files, under the `info/` subdirectory:

```
src/info/cpu_info.h            // Represents CPU time spent in various modes
src/info/load_average_info.h   // Structure containing various load averages
src/info/memory_info.h         // Contains information about system memory
src/info/process_info.h        // Represents a process or thread
src/info/system_info.h         // Aggregates all information for the system
```

These also have some starting method stubs for the various data retrieval operations you'll need to do. These are found in the associated `.cpp` files.

Each of these functions has unit tests already written for them, and you will be required to implement the function stubs such that all the tests pass. You will not be allowed to change the function signatures of these methods, nor should you modify the tests themselves.

To run the tests (though almost all of them should fail initially), run the following from within your repository:

```
make test
```

The starter code also includes a static snapshot of the `/proc` filesystem that I took on an Alamode machine. The tests run against this data, allowing the results to be consistent and predictable. You should

avoid making changes to any of this fake `/proc` directory.

Finally, you are also provided with a file that contains a simple `main()` function, called `top.cpp`. It implements a very simple counter using `ncurses`. The entire screen is cleared and redrawn each iteration of the loop (though `ncurses` is smart enough to only re-draw things that have changed).

To build this program, type:

```
make
```

It should compile without errors on the Alamode machines, producing a file called `mytop`. To run it, simply type:

```
./mytop
```

Just like the real `top`, you can exit by typing `q`.

For additional information on the starter code, read its README file.

2 Reading from `/proc`

The real Linux `top` program reads its data from the `/proc` filesystem, and so will you. If you want to get an idea about what all `top` is doing on each iteration / refresh, try running the following command on imagine.mines.edu:

```
strace top -b -n 1  > /dev/null
```

`strace` will show you all the system calls a given command or process is executing (if you want to know more, read its `man` page). The `-b` flag tells `top` to run in batch mode (rather than in an `ncurses` terminal environment), and the `-n 1` flag instructs `top` to stop after a single iteration.

All of that is happening in a *single iteration* of `top`! By default, it refreshes its data every couple seconds, or more frequently if you request it. Luckily, most of that is simply iterating over each each active process, so your task is less daunting than the `strace` output might suggest.

Your first task will be to implement the function stubs in `src/info/.{h,cpp}` so that the all provided unit tests pass. The header files contain extensive comments about what each function should do, and the tests themselves should also help you understand what is expected.

The unit tests are written using googletest, which is Google's C++ testing framework. They run against a static copy of a `/proc` filesystem, which is included with the starter code, allowing the tests to be completely deterministic (like all good tests should be). You can run the test suite by typing the following command:

```
make test
```

Once all the unit tests pass, you will have confidence that your code is retrieving all the correct data and populating the structs appropriately. Even better, you can modify your code and still be confident that you haven't broken anything (assuming the tests still pass). I encourage you to write additional unit tests as you progress further, though this isn't required.

Now that you have all the data you need, you can move on to the really gratifying[1] part and start displaying it!

3 Displaying the Data

Your implementation of `top` is required to display the following general system information:

- System uptime: how long the computer has been on, with appropriate labeling. For example: `XX days, HH:MM:ss`
- Load average over the last 1, 5, and 15 minutes
- Percent of time 1) all processors and 2) each individual processor spent:
  - in user mode
  - in kernel mode
  - idling
- Total number of processes that currently exist on the system
- Total number of processes that are currently running
- Total number of threads (both user-land and kernel-level)
- The amount of memory:
  - that is installed on the machine
  - that is currently in use
  - that is currently available

Additionally, you must display a process table with the following (labeled) columns:

- Process ID (PID)
- Resident memory size
- Current state (single-letter abbreviation)
- % of CPU currently being used

[1] This word may not mean what I think it does...

- Total amount of time spent being executed by a processor (HH:MM:SS)
- The name of the binary (or the cmdline) that was executed

Your process table should be sorted, as described in the next section. You are required to show at least the top 10 processes for the chosen category.

This probably sounds like a lot, but don't despair! Implement one small detail at a time. Do one specific thing at a time and then cross it off the list before moving to the next. If the unit tests are passing, you should already have all the data that you need; all you have to do is display it.

Your implementation must be able to run in a reasonably sized terminal window. Please document the ideal size in your README so that your grader can view your UI in the intended manner.

The `printw` method in the `ncurses` library allows you to output text to the window. You should not use other output mechanisms, like `cout`; the results will not be what you expect.

4 Command-Line Flags

Your implementation of `top` must support the following command-line flags:

```
-d --delay=DELAY
  Delay between updates, in tenths of seconds

-s --sort-key COLUMN
  Sort by this column; one of: PID, CPU, MEM, or TIME

-h --help
  Display a help message about these flags and exit
```

You must use the `getopt_long(3)` method for parsing these flags. An example will be posted to Piazza. As always, check the `man` page for additional information.

The user of your program must be able to pass a flag in the format of:

```
-d DELAY
--delay DELAY
```

If he does, your program must refresh its data every `DELAY` tenths of a second. If no such flag is passed, your program must default to refreshing every second.

Additionally, your program must support a flag in the format of:

`-s COLUMN`
`--sort-key COLUMN`

If this flag is passed, your process table must be sorted in order of the given column. Valid values are:

- PID (for sorting by process ID, in ascending order)
- CPU (for sorting by CPU utilization, in descending order)
- MEM (for sorting by memory utilization, in descending order)
- TIME (for sorting by total CPU time executed, in descending order)

Finally, the `--help` (or `-h`) flag must cause your program to print out instructions for how to run your program and about the flags it accepts and then immediately exit.

5 Requirements and Reference

- Always, always, always check the return values of system calls. Always. The return value is the way for the kernel to communicate to user-level processes that an error has occurred. Usually errors are denoted by negative return values, but check the man page for each system call. If you check the return value of a system call and detect an error, use `errno(3)` and `perror(3)` to print an informative message, then exit.
- Use good design. Do not code monolithic functions. In your design, think about what you learned in Software Engineering. You should avoid coding practices that make for fragile, rigid and redundant code.
- Use good formatting skills. A well formatted project will not only be easier to work in and debug, but it will also make for a happier grader.
- You can develop your project anywhere you want, but it must execute correctly on the machines in the Alamode lab (BB136).
- Your final submission must contain a `README` file with the following information:
  - Your name.
  - A list of all the files in your submission and what each does.
  - Any unusual/interesting features in your programs.
  - Approximate number of hours you spent on the project.
  - The ideal terminal size at which to run your `top` implementation (if it's not a reasonable size, then don't expect a perfect grade)
- To compile your code, I should be able to clone your repository, `cd` into it, and simply type `make`.

6 Deliverables

For each deliverable, your grader must be able to clone your repository by 11:59 on the due date. There are no exceptions for intermediate deliverables. For final deliverables, you may use slip days to extend your due date without penalty; for each slip day you use, you have one extra day to submit your work. Any requirements not mentioned in an intermediate deliverable are due as part of the final deliverable.

6.1 Intermediate Deliverable: Due February 27th, 2018 @ 11:59pm

You must push a version of your repository where:

- All of the provided function stubs under `src/info/` are fully implemented
- All existing unit tests pass without any modifications

Tag and push this deliverable as follows:

```
git tag deliverable-1
git push --tags
```

6.2 Final Deliverable: Due March 6th, 2018 @ 11:59pm

You must push a completed version of your program where:

- All required information is displayed and updated after the appropriate delay
- Data is read from the real `/proc` filesystem
- All required flags are implemented

Tag and push this deliverable as follows:

```
git tag final
git push --tags
```

You may also earn extra credit (as much as 10 points, completely at the discretion of your grader) by implementing noteworthy additional features in your `top` program. Any such extra credit will not increase your grade beyond 100%.

Examples include outputting the users associated with each process, CPU meter bars, colorized output, batch mode, interactive functionality, etc. The bottom line is that going above and beyond will be rewarded. It's a great way to learn more about interacting with the operating system. You must explicitly document any functionality that you think qualifies for extra credit in your README file; otherwise, your grader will not know what to look for and cannot be faulted for overlooking your super-awesome feature.