CS 4420/5420

Fall 2019/2020

Programming Assignment 1

Parts I and II

Exploring the Proc Filesystem

Max. Points for Part I and II: 25 (see below)

INTRODUCTION

In assignment 1, you will develop a set of tools to explore the Linux proc filesystem

Each of those tools are based on the Unix proc filesystem. The proc filesystem is located under "/proc" on every Unix system. It is really a pseudo-filesystem: the contents of proc are not actually stored anywhere on any mass storage device like in a real filesystem, but its contents (directories, files) are only created ("on the fly") when someone accesses them.

At its root (i.e., /proc), the proc filesystem (on Linux) contains a number of directories and files. The directories either have integers as names (in which case they represent Unix process ID's (PIDs), and contain various information the kernel maintains about a process that is currently running in the system), or they have other names. For example, the directory "/proc/1234" contains various information (i.e., files and other directories) about the process with the PID 1234.

This is an example of a /proc directory on a Linux computer:

```
drews@tesla1:~$ ls /proc
1      1170   1327   14409  1565  1867   241  38   579  957          iomem         sched_debug
10     1174   1337   14423  1567  1870   242  40   58   988          ioports       schedstat
1009   1175   1340   14425  16    188    243  41   59   990          irq           scsi
101    1176   1342   14430  1663  189    245  416  63   998          kallsyms      self
1014   1188   1345   14431  169   190    246  42   64   acpi         kcore         slabinfo
1020   1192   1347   14442  17    191    25   43   65   asound       keys          softirqs
1043   1193   1348   14461  170   197    26   430  66   buddyinfo    key-users     stat
1044   12     1349   14462  171   199    27   45   671  bus          kmsg          swaps
1046   1201   1353   14476  1713  2      28   46   672  cgroups      kpagecgroup   sys
1047   1206   13740  14478  1716  20     280  47   7    cmdline      kpagecount    sysrq-trigger
1050   1207   13741  14479  172   204    294  48   8    consoles     kpageflags    sysvipc
1060   1209   13789  14482  1722  206    3    49   82   cpuinfo      loadavg       thread-self
1073   1214   13791  14561  173   208    30   5    83   crypto       locks         timer_list
1095   1216   13841  14562  174   209    308  50   84   devices      mdstat        timer_stats
11     1222   13909  14563  1744  21     31   51   85   diskstats    meminfo       tty
1120   1227   14001  14584  175   21052  32   52   86   dma          misc          uptime
1127   1237   1408   14617  176   2134   325  53   87   driver       modules       version
1134   1244   1410   14631  18    216    33   54   88   execdomains  mounts        version_signature
115    12784  14168  14634  180   22     343  55   9    fb           mtrr          vmallocinfo
1150   13     14191  15     181   224    35   56   90   filesystems  net           vmstat
116    1321   14248  1517   182   225    36   57   91   fs           pagetypeinfo  zoneinfo
1169   1325   1426   1550   183   23     37   575  92   interrupts   partitions
drews@tesla1:~$
```

In this example, you an see the directories with numbers as directory names on the left hand side (they correspond to processes; in this case in the range between 1 and 14634). In addition, you can see various other directories and files. They are used by the operating system to report various system information to users and applications. Most of the files are ASCII files that can be read by simply opening them in a text editor. For example, the file "/proc/cpuinfo" contains a detailed description of the properties of the CPU.

You can obtain information about a process simply by opening the right file in the directory corresponding to the process you're interested in. For example, if you need to obtain some basic profiling statistics for a given process (example, the process with a process ID (PID) of 14584), you need to access the file "/proc/14584/stat". It contains a single line of tokens (ASCII characters) separated by spaces.

Here is an example of a /proc/[pid]/stat file:

```
14584 (bash) S 14562 14584 14562 34824 14598 4194393 1061 1325 0 1 5 0 0 0 20 0 1 0 33837297 …
```

Each token is separated by a space character. The meaning of the individual tokens is described in the Linux manpage for the proc filesystem. The man page entry for "proc" gives you a detailed description of the format of this file. You can display the man page entry by using the "shell> man proc" command (**Note: you may need to specify the man page section in order to get the correct man page entry. Try, e.g., "man -s5 proc")**. The information for the "stat" file can be found under "/proc/[pid]/stat" on the manpage.

In the above example, the first token is "14584" which is the process ID of the process. The second token, "(bash)" is the command/program name of the process (in parenthesis). In our example it happens to be the bash shell program. The next token, "S", denotes the current state of the process (which, according to the information in the manpage, in this case is "S" = sleeping). The following token "14562" is the process ID (PPID) of the parent process of our bash process, etc.

Part I: The "4420_proctool_1" Tool

Overview:

The "4420_proc1" tool is a very simple tool that prints some basic information for all the processes running on a Linux system. The tool does not require any additional command arguments. You should be able to run the tool simply by using the following command:

Shell> 4420_proctool_1

It will generate the following information (columns) for every process:

Process ID (PID)	Command (name of the program)	Parent Process ID (PPID)	Process State	Virtual Memory Size (in bytes)

In order to create this output, your tool will need to access all of the the "/proc/<pid>/stat" files (i.e, you will need to enter every directory corresponding to a PID). All the information required can be directly extracted from the corresponding "/proc/<pid>/stat" files.

The following table shows some sample output:

```
drews@pu1:~/proc_project$ ./a.out
       PID      Command               PPID   State    VM Size (bytes)
-------------------------------------------------------------------
         1      systemd                  0    S     185796
         2      kthreadd                 0    S          0
         3      ksoftirqd/0              2    S          0
         7      rcu_sched                2    S          0
...  // some processes removed
       248      jbd2/sdb5-8              2    S          0
       249      ext4-rsv-conver          2    S          0
       300      rpciod                   2    S          0
       308      systemd-journal          1    S      43892
       311      kauditd                  2    S          0
       312      lvmetad                  1    S      94772
       323      systemd-udevd            1    S      39040
       385      kvm-irqfd-clean          2    S          0
...   // some processes removed
```

```
  875      ext4-rsv-conver          2    S          0
 1072      rpcbind                  1    S      47624
 1091      ModemManager             1    S     413440
 1095      cron                     1    S      22680
 1099      atd                      1    S      19716
 1105      rsyslogd                 1    S     320012
 1111      bluetoothd               1    S      31956
 1115      dbus-daemon              1    S     109008
 1210      avahi-daemon             1    S     110916
 1213      acpid                    1    S       4396
 1220      systemd-logind           1    S      20572
 1233      snapd                    1    S     537232
 1241      avahi-daemon          1210    S     110556
 1308      polkitd                  1    S     338460
... // some processes removed
```

Specific Requirements:

- Implement the above described tool "4420_proctool_1". The tool has no required command argument and should display the above described columns for each Linux process.
- You have to implement the tool either in C or C++. You are only allowed to obtain the information via the proc filesystem (as explained above) for this project. You should only use functions to open and read files and directories. Do not use any other functions (example: system(), getrusage(), etc.).
- The screenshot above shows a sample output of my reference tool.
- You will need to submit the source file(s), header files (if necessary), and you WILL NEED TO SUBMIT A **MAKEFILE**. I will grade the project on "pu1.cs.ohio.edu". I will type "make" to build the tool. If "make" fails (due to an error), the GRADE FOR THIS PART OF ASSIGNMENT 1 WILL BE AN "F"!
- You will need to submit the program from "**p2.cs.ohio.edu**" (NOTE: THIS IS IMPORTANT. THE SUBMISSION WILL NOT WORK FROM ANY OTHER (LAB)MACHINE OR SERVER!). The submission is simple: "cd" into your project folder and type the following command (exactly as is shown here): "**bash> /home/drews/bin/442submit 1**". Should you need to re-submit your project (due to changes), you will need to add an override switch "-f" to the submission: "**bash> /home/drews/bin/442submit -f 1**".

Deadline:

The project is **due by Fridy, 9/27/2019 (11.59pm)**. I will **not accept any late submissions**.

Grading:

Total possible points for Part I of the assignment: **10 points**

Breakdown:

- Each of the above columns yields 2 points * 5 columns = 10 points

Additional Help and Resources:

This project is fairly straightforward. You can use any file access functions available in C/C++. I would recommend you use basic Linux system calls: open(), read(), close() (see "man -s2 open", "man -s2 read", man -s close", etc.). But you could also use C standard I/O: fopen(), fread(), fclose(), or C++ standard I/O.

The only thing that may be a little unusual is that you will have to determine (and later enter) all the directories corresponding to process IDs under "/proc" (i.e., all the "/proc/<pid>" directories). The following sample code shows how to accomplish this in C:

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
#include <stdlib.h>


int pid;
struct dirent *ep;
DIR* dp;

dp = opendir ("/proc");

  if (dp != NULL)
    {
      while (ep = readdir (dp))
      {
        pid = strtol(ep->d_name, NULL, 10);
        if( ( ep->d_type == DT_DIR ) && ( pid > 0) )
        {
          puts (ep->d_name);
        }
      }
      closedir(dp);
    }
  else
    perror ("Couldn't open the directory");
```

Part II: The "4420_proctool_2" Tool

Overview:

In part II of this assignment, you will start with the solution to part I and create a tool that allows drawing a process tree consisting of all processes in your Linux system. The nodes of this tree represent processes and each (directed) edge between two processes represents a parent/child relationship.

In Unix, every process that is created, is created by a so-called "parent process". A parent process is a process that created one (or more) child processes. Each process in Unix has a unique parent process. The only process that does not have a parent process is the root of the tree. Some UNIX processes introduce a process with PID 0 (e.g., SOLARIS sched process), but this is not a real process like all the other processes. Specifically, there is no process with PID 0 in Linux (hence, there is also no directory "/proc/0"). In Linux, there are processes, however, that list a process with the parent process ID (PPID) of 0 (example: PID 1 and PID 2).

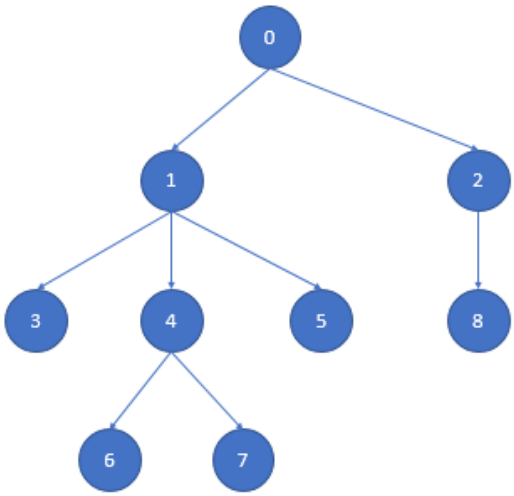
For the sake of this project, we assume that this "artificial" or "dummy" process with the PID 0 exists and represents the root of the Linux process tree.

The process three that your program should draw should be created using standard I/O print functions (such as "printf()" in C or "cout << …" in C++.

Consider the following simple example process tree, consisting of 8 processes + 1 dummy root process:

Process Name	PID	PPID
Dummy	0	N/A
A	1	0
B	2	0
C	3	1
D	4	1
E	5	1
F	6	4
G	7	4
H	8	2

The above parent/child relationships give rise to the following process tree graph (drawn in traditional fashion):



Note that in your solution to part I of this assignment, you already extracted all the information corresponding to the above table or proesses (namely, the PID, command name, PPID, etc.) for all process in Linux system.

Your tool should create a graph using ASCII characters. Edges are visualized using "-" and "|" characters. Processes are represented by their PID followed by the command name (in parenthesis).

The output for the above example should look as follows:

```
---->     0(dummy)              ---->     1(A)   ---->     3(C)
                |                 |
                |                 ---->     4(D)   ---->     6(F)
                |                 |                  |
                |                 |                  ---->     7(G)
                |                 |
                |                 |
                |                 ---->     5(E)
                |
                ---->     2(B)    ---->     8(H)
```

A complete output file for a real Linux system (pu1.cs.ohio.edu) will be provided on blackboard.

Specific Requirements:

- Implement the above described tool "4420_proctool_2". The tool has no required command argument and should display the above described process tree.
- You have to implement the tool either in C or C++. You are only allowed to obtain the process information via the proc filesystem (as explained in part 1 above) for this part of the assignment. To do that, you should only use functions to open and read files and directories. Do not use any other functions (example: system(), getrusage(), etc.). Of course, this restriction is only for extracting process information.
- You will need to submit the source file(s), header files (if necessary), and you WILL NEED TO SUBMIT A **MAKEFILE**. I will grade the project on "pu1.cs.ohio.edu". I will type "make" to build the tool. If "make" fails (due to an error), the GRADE FOR THIS PART OF ASSIGNMENT 1 WILL BE AN "F"!
- You will need to submit the program from "**p2.cs.ohio.edu**" (NOTE: THIS IS IMPORTANT. THE SUBMISSION WILL NOT WORK FROM ANY OTHER (LAB)MACHINE OR SERVER!). The submission is simple: "cd" into your project folder and type the following command (exactly as is shown here): "**`bash> /home/drews/442submit 2`**". Should you need to re-submit your project (due to changes), you will need to add an override switch "-f" to the submission: "**`bash> /home/drews/442submit -f 2`**".
- **Do not accidentally overwrite your submission for part 1 of this assignment.**

Deadline:

The project is **due by Wednesday, 10/2/2019 (11.59pm)**. I will **not accept any late submissions**.

Grading:

Total possible points for Part I of the assignment: **15 points**

Breakdown:

- 12 points for (largely) correctly drawing the process tree. This includes adding the pseudo process PID 0 as a root of the process tree.
- 3 points if the graph does not contain any missing (or not correctly displayed) vertical "|" or horizontal "-" characters corresponding to the edges of the graph and if the output looks nice.

Additional Help and Resources:

- As stated above, start with the code for part 1 of the assignment
- Create some sort of array (or other suitable data structure) to store the information about all processes (including command name, PID, PPID).
- I would suggest you manually add a dummy process with PID 0 to this data structure

- I recommend you implement the function to display the process tree as a **recursive function**. **Note: this function is not really that difficult but may take a little bit of trying to get done right. Hint: I implemented this function using around 15-20 lines of C code.**
    - I recommend using the C function "printf()" to created formatted output
    - I also suggest you make use of the tab character '\t' to format the output of the tree
    - One weird thing about the tab character is that it is interpreted by your terminal, and thus depends on your terminal configuration. You may end up having to change the tab width manually for this project. This should be done "outside" of your program by using the Unix command "tabs" (see man tabs for details). For example typing: "tabs 20" sets the tab width to 20 characters.
    - You can change the tabs programmatically (i.e., from within your program, as opposed to by a separate shell command) by adding the following function call somewhere right at the beginning of main():

      ```
      system("tabs 25");
      ```

      If you "mess up" your tab width and wishto reset it to default settings, use the call

      ```
      system("tabs -8");
      ```
    - One weird thing about tabs is that each tab is simply a '\t' ASCII code that is sent to your terminal. The terminal then interprets it based on the terminal configuration. This also means that when you redirect the output of your program to a file (e.g., "./4420_proctool_2 > outfile"), and later open the redirected file, the format of the output may vary. Note that this is normal and expected.
- On blackboard, I will provide you with a test directory of (pseudo) proc files that correspond to the above example (the one with processes "A" through "H"). You can use this directory to test your program.