UNIX 101

OR "HOW TO FEEL LIKE A TRUE HACK3R"

myps

Tuesday 17th May, 2022

Contents

1 Foreword

1.1 Objective

Welcome to myps! The goal of this subject is to dig into Linux concepts by rewriting some functionalities of the popular command line tool `ps`.

1.2 Notions required

Before digging into the subject, here are the pre-requisite. Make sure you can do the following things:

- Use a terminal, run commands on a UNIX environment
- Code simple programs
- Navigate in a UNIX filesystem
- Create commits and push code to a distant git repository
- Code simple CLI applications

2 Setup

You need a Linux machine, a text editor and a working development environment.

Any language can work; if you need guidance on eithr Python or C, check out the following sections. They briefly go over the installation of the tools you need to start working.

2.1 Python

If python3.5 (or more) is not installed, run `sudo apt install python3-pip`. That should be enough to install an interpretor.

2.2 C

To make sure you have everything set up, run `sudo apt install gcc binutils make`. That should be enough.

3 Evaluation

3.1 Instructions

You are asked to constituate a team of 2 persons. You will be graded as a team on the quality of your code, the number of level passed and the quality of your explanation regarding your work.

You have to deliver your code before Wednesday 15^{th} December, 2021, 7:00 AM.

3.2 Expected output

The expected output is a git repository, hosted on a public platform like Github. The repository should host code that either compiles to a `myps` binary or that can be interpreted through a `./myps.<extension>` entrypoint, where `<extension>` is th extension of the programming language (e.g. `py`).

3.3 A word on cheating

You have to remember that you should be studying for your own good. Cheating will not bring you any good in the long term; it is fine not to be able to finish every exercise of the subject, your main goal is to train and learn things.

Any form of cheating will immediately bring your grade down to 0. Additionally, your main teacher will be taken notice of that.

Note: Changing the name of some variables will of course not trick the anti-cheat engine :)

4 Subject

4.1 Objective

The goal of this subject is to rewrite some functionalities of `ps(1)`. Recoding elementary UNIX utilities is an incredibly effective way to learn more about UNIX systems inner workings.

This subject will make you learn in-depth notions regarding Linux processes: pid, parenting, processes statuses, etc. Also, you will learn about the /proc/ filesystem, an API to the Linux kernel.

4.2 Walkthrough

The subject is delimited in levels. The first levels are pre-requisite so you can build `myps` core features. The mid-levels are the main features of `myps`. The last levels describe more difficult features. Sections labelled **Bonus** are optional.

Levels should be completed one after another. Commit often, push often, ask for help whenever you are blocked.

4.3 How does ps work?

The goal of this subject is to understand how ps work by rewriting it. However, we will cover the fundamentals in this section so you know where to begin and where to look for information.

`ps(1)` is a program written in C. Its goal is to display to the user information regarding the currently running processes on the system. To keep things simple, we can consider that a process is an instance of a program. The information that `ps(1)` can display include the likes memory usage, process identifiers (PIDs), CPU time used since process launch, etc. It is a widely used tool in the UNIX world and you may have already used it.

`ps(1)` runs without specific permissions; yet, it can grab so much information! It looks complicated, but in reality, that program is merely an interface. In fact, its only made of around 10 000 lines of code! Processes information is made available by the kernel and the sole job of `ps(1)` is to gather them and format them nicely for the user.

There are a few APIs that the kernel exposes. The one that `ps(1)` uses (and that you are going to use to) is the `/proc/` filesystem. It is a pseudo-filesystem which provides an interface to kernel data structures. To keep things simple, think about it as some kind of data store where useful kernel information is readable. This information is made available through files that you can read in `/proc/*`. Try it!

```
$ls /proc/
ls /proc/
1     1606  21    25    299  4    439  623  80          bus
  execdomains  key-users   misc         self          tty
10    17    2126  2532  30   414  468  659  81          cgroups    fb
             keys         modules      slabinfo      uptime
1017  18    2128  2551  308  417  471  668  82          cmdline
  filesystems  kmsg         mounts       softirqs      version
11    19    2129  2552  309  422  476  675  822         consoles   fs
             kpagecgroup  mtrr         stat
  version_signature
116   2     22    26    32   423  525  7    83          cpuinfo
  interrupts   kpagecount   net          swaps         vmallocinfo
12    20    2203  27    34   424  6    716  89          crypto
  iomem        kpageflags   pagetypeinfo  sys          vmstat
13    203   2204  274   35   425  618  78   9           devices
  ioports      loadavg      partitions   sysrq-trigger  zoneinfo
14    204   23    28    356  426  619  79   98          diskstats  irq
             locks        sched_debug   sysvipc
15    205   2378  29    357  437  620  798  acpi        dma
  kallsyms     mdstat       schedstat    thread-self
16    208   24    298   36   438  622  8    buddyinfo   driver
  kcore        meminfo      scsi         timer_list
```

You may have already used the command `uptime(1)`. It gives you the information on how long the system has been up and running since last reboot.

```
$uptime
 00:13:09 up 14:59,  1 user,  load average: 0.00, 0.00, 0.00
```

Now, notice how there is a file named `/proc/uptime`. If we read its content, we obtain this:

```
$cat /proc/uptime
54049.70 53908.20
```

By reading the man page of `procfs` (`man procfs`), the documentation indicates that *this file contains two numbers (values in seconds): the uptime of the system (including time spent in suspend) and the amount of time spent in the idle process.*

There is a pretty good chance that we can rewrite the program `uptime` by reading that file.

This is just an example of what kind of information `/proc/` exposes. It contains directories named after numbers. Those numbers reference processes identifiers (pid). Inside each directory, there are files containing information related to the given pid. Those are read by `ps(1)` to report information. This is the magic behind `ps(1)`.

4.4 Reference documentation

- **procfs documentation**: Your main entry point to understand what files reference what in the `/proc` filesystem
- **ps man page**: Your main entry point regarding `ps(1)` options
- **ps source code**: I do not recommend using it unless for very specific use cases. It is difficult to read and understand, so only take a look there if you are desperate

4.5 Ready?

Your goal is to write a program that works like `ps(1)`. We will start small and add features level by level. At all times, you should be able to compare your program with the official one, for the features you recoded. They should behave the exact same way. Ensuring that `myps` behaves exactly like `ps(1)` will allow you to know if you are on the right track, debug some features and brag about your skills!

4.6 Level 1: Columns and padding

4.6.1 Objective

The goal of this level is to code the building blocks of your program. Your program may correctly retrieve processes information, if it cannot display it correctly, it is all for nothing.

Your goal is to implement the `-o` option of `ps(1)`. `ps -o` allows one to select which columns should be displayed. By default, `ps` displays the columns PID, TTY, TIME and CMD:

```
$ps
  PID TTY          TIME CMD
 2204 pts/0    00:00:00 bash
 2566 pts/0    00:00:00 ps
```

Explicitely setting the `-o` option selects which columns are going to be printed out:

```
$ps -o pid,time
  PID     TIME
 2204 00:00:00
 2568 00:00:00
```

To pass this level, you should write a program that accepts the `-o` option. It authorized column names are, for now, PID, PPID and CMD.

As our program does not yet retrieve processes information, no data should be displayed under column names.

4.6.2 Example usage

```
$./myps -o pid
  PID
$./myps -o pid,ppid
  PID  PPID
$./myps -o pid,ppid,command
  PID  PPID COMMAND
$./myps -o pid,command,ppid
  PID COMMAND                      PPID
```

4.6.3 Help

Notice the padding? Each column has its own padding rules.

The reference can be found in the source code, **here**. The *width* column indicates how many characters long the column should be. The *RIGHT* mention indicates that the column should be padded to the right. A *LEFT* mention or no mention at all seems to indicate that the column should be padded to the left. The rule of thumb seems to be "pad left for text" and "pad right for numbers".

In the example, PPID is has a width of 5 and is padded to the right. On the other hand, COMMAND has a width of 27 and is padded to the left.

4.6.4 Bonus

If you play with `ps(1)` a bit, you can notice that a column's padding behave differently if it the last column displayed. Find out what the rule is and implement it.

4.7 Level 2: Display a specific process information

Now that we can output some columns, we will implement the `-p` option. `-p` gives information on a given process, identified by its pid. You are not asked to handle the case where multiple pids are given to `-p` (e.g. `ps -p 1,10`)

```
$ ps -p 1
  PID TTY          TIME CMD
    1 ?        00:00:01 systemd
```

Since we did not recode the TTY, TIME and CMD information retrieval, you are only asked to print relevant information for the PID column. Typically, your program should handle this kind of invocations: `ps -p <PID> -o pid`. If the given pid does not relate to a running process, only print the column names.

4.7.1 Example usage

```
$ps -p 1 -o pid
  PID
    1
$ps -o pid -p 10
  PID
   10
$ps -p 1111111 -o pid
  PID
```

You are asked to print relevant information only for the **PID** column.

4.8 Level 3: Parent pid

To validate this level, the `-o` option of your program should handle the `ppid` argument. Typically, your program should handle this kind of invocations: `ps -p <PID> -o pid,ppid`.

When in doubt with what the implementation should be, refer to `ps(1)`. The information you look for should live somewhere in `/proc/<pid>/`. Refer to `procfs(5)`.

4.8.1 Example usage

```
$ps -o pid,ppid -p 10
  PID  PPID
   10     2
$ps -o pid,ppid -p 1
  PID  PPID
    1     0
$ps -o ppid,pid -p 1
 PPID   PID
    0     1
$ps -o pid,ppid -p 1
  PID  PPID
    1     0
$ps -o ppid -p 10
 PPID
    2
```

4.9 Level 4: Command column

Similarly to last level, you are asked to handle the `command` argument for the `-o` option. Beware, `-o command` is different from `-o cmd`! In case of doubt, once again, refer to `ps(1)`'s behavior.

4.9.1 Example usage

```
$ps -o pid,ppid,command -p 3809
  PID  PPID COMMAND
 3809  3705 sshd: vagrant@pts/0
$ps -o pid,ppid,command -p 10
  PID  PPID COMMAND
```

```
   10      2 [migration/0]
$ps -o pid,ppid,command -p 1
  PID  PPID COMMAND
    1     0 /sbin/init
$ps -o pid,command,ppid -p 5
  PID COMMAND                       PPID
$ps -o pid,command,ppid -p 6
  PID COMMAND                       PPID
    6 [mm_percpu_wq]                   2
$ps -o command,ppid,pid -p 3809
COMMAND                       PPID   PID
sshd: vagrant@pts/0           3705  3809
$ps -o command,ppid,pid -p 3915
COMMAND                       PPID   PID
bash -c while true ;do slee   3810  3915
```

4.9.2 Help

- Notice how the command is sometimes displayed with [&] while sometimes it is not.
- Notice how the value of *command* is truncated in the last example.

4.10 Level 5: Display multiple processes

To pass this level, you are asked to fully implement the `-p` flag so it can be given multiple pids.

4.10.1 Example usage

```
$ps -o pid,ppid,command -p 1,10
  PID  PPID COMMAND
    1     0 /sbin/init
   10     2 [migration/0]
$ps -o pid,ppid,command -p 1,10,88888
  PID  PPID COMMAND
    1     0 /sbin/init
   10     2 [migration/0]
```

4.11 Level 6: Display all processes!

To pass this level, you are asked to implement the `-e` flag. This flag makes `ps(1)` display information on **all** currently running processes.

4.11.1 Example usage

```
$ps -e -o pid,ppid,command | tail -n 10
 1102     1 /lib/systemd/systemd --user
 1103  1102 (sd-pam)
 1598   696 sshd: vagrant [priv]
 1686  1598 sshd: vagrant@pts/1
 1687  1686 -bash
 1811     2 [kworker/u2:0]
 1886     2 [kworker/u2:1]
 1985     2 [kworker/u2:2]
 1986  1687 ps -e -o pid,ppid,command
 1987  1687 tail -n 10
```

4.12 Level 7: Display processes related to the current terminal

To pass this level, you are asked to implement the behavior when neither `-e` is given nor `-p`. In this case, `ps(1)` displays process information related to the processes related to the current terminal

4.12.1 Example usage

```
$ps -o pid,command
  PID COMMAND
 1687 -bash
 2039 ps -o pid,command
$while true ; do sleep 1 ; done &
[1] 2040
$ping 8.8.8.8  > /dev/null &
[2] 2068
$ps -o pid,command
  PID COMMAND
 1687 -bash
 2040 -bash
 2068 ping 8.8.8.8
 2075 sleep 1
```

```
2076 ps -o pid,command
```

4.13 Level 8 and beyond (bonus)

With all the work you have done, you pretty much have a minimal `ps(1)`. In fact, most of the times, when we use `ps(1)`, we are looking for the output of the command `ps -e -o pid,command`, which we implemented in level 6.

From now on, you can add features to your program! Each `-o` column or additional flag implemented is a way to learn new things on the Linux kernel!