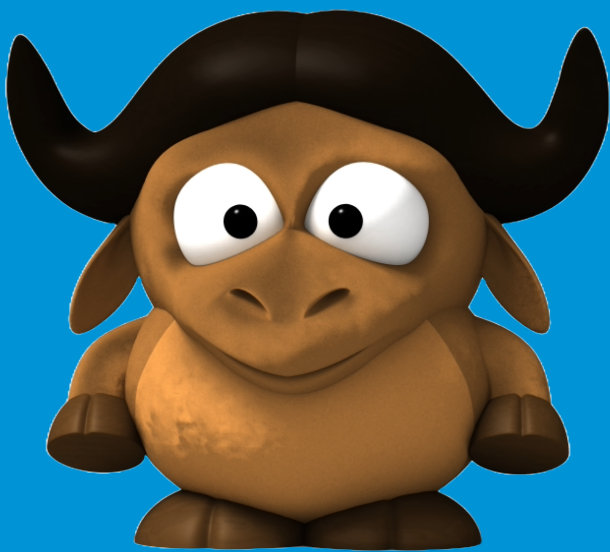
{EPITECH}

MY_TOP

LET'S TAKE A LOOK AT UNIX PROCESSES !



MY_TOP

binary name: my_top
language: C
compilation: via Makefile, including re, clean and fclean rules
Forbidden functions: system, exec*, popen, getloadavg, getrusage, getrlimit, getutent, setutent or any other function that retrieves process or system information for you.

- ✓ The totality of your source files, except all useless files (binary, temp files, objfiles,. ..), must be included in your delivery.
- ✓ All the bonus files (including a potential specific Makefile) should be in a directory named bonus.
- ✓ Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

Lore

> **My_Ls: Resolution**
>
> Finally, after hours of effort, Aki executes My_Ls. The terminal crackles, hesitates, and then suddenly, the files appear on the screen. A stream of names scrolls by—directories with cryptic names that seemed lost forever. It's as if she has opened a door to a forgotten world, a world where this wreck was once alive and functional.
>
> Some files are mundane—traces of daily activity, maintenance logs, navigation records. Others, however, bear mysterious and unsettling names, hinting at unknown functions and obsolete systems. Each file is a piece of the puzzle, a clue about what this wreck once was and what it could perhaps become again.
>
> Aki realizes that My_Ls is not just a tool to explore the ship; it's the first key to understanding the secrets hidden within. Through this command, she begins mapping the ship's digital architecture, revealing inaccessible areas, locked systems, and laying the groundwork for her future exploration.
>
> But she knows this is only the beginning. My_Ls has brought her to the surface of the ship's mysteries, yet there are still so many hidden systems, protected by layers of security and oblivion. To delve deeper, she'll have to continue applying all her skills, recreating forgotten tools, and awakening the machine that slumbers beneath her feet.

My_Top: Diving into the Depths of Forgotten Systems

After successfully listing the files and directories with My_Ls, Aki feels a new surge of excitement—a sense of mastering something ancient and powerful. But she knows this is just the surface. The wreck is still alive, with mysterious processes stirring in the machine's depths, invisible yet perceptible. Occasionally, low rumbles and vibrations echo through the walls, as if parts of the ship are attempting to awaken.

She recalls another tale from the Book of UNIX: a story about a way to "see inside," to monitor the operations of these ancient machines in real time, to observe their rhythms as a doctor might listen to a heartbeat.

The command top comes to mind, an ancient method of tracking a system's load and activity. Aki realizes that if she wants to understand what's happening within the belly of the wreck, she must recreate this command—forge a new version for this timeworn, fragmented machine. However, the ship's internal systems are incomplete and scattered, and the original top code has likely been corrupted by centuries of inactivity. She embarks on a new challenge: rebuilding this internal vision.

Her fingers fly across the keyboard, trying to coax the dormant processes back to life. But the screen remains silent, displaying cryptic errors. The ship, though partially alive, seems to resist her intrusion. It feels almost as if the wreck harbors a primitive consciousness, intent on guarding its secrets.

That's when Aki realizes the ship doesn't function as a unified system. Different modules are isolated from one another, autonomous processes lingering as remnants of a once-mighty integrated network. Combining her knowledge with her intuition, she painstakingly rewrites a command tailored to the fractured structure of the wreck. My_Top won't be just a simple process monitor—it will be a tool to observe the multiple, scattered hearts of the ship, barely synchronized and barely beating.

The project

Objectives

You must recode the **top** command.

It is a tool to monitor your system and processes.

Your project will need to use the Terminal User Interface library **ncurses**.

Here is an example of what the `top` command output looks like:

```
top - 10:19:23 up  3:06,  0 user,  load average: 2.01, 1.49, 1.19
Tasks:   6 total,   2 running,   4 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.4 us,  7.3 sy,  2.7 ni, 89.5 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  11950.5 total,   8126.7 free,   2742.2 used,   1316.7 buff/cache
MiB Swap:   2048.0 total,   2048.0 free,      0.0 used.   9208.4 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   20 clery     23   3  372764   4932   2412 R  99.7   0.0   1:42.90 yes
    1 root      20   0  375128   7608   4168 S   0.0   0.1   0:00.07 bash
   16 root      20   0  376336   7548   4388 S   0.0   0.1   0:00.05 su
   17 clery     20   0  369924   4888   2360 S   0.0   0.0   0:00.02 sh
   18 clery     20   0  375692   9988   3976 S   0.0   0.1   0:00.11 bash
   29 clery     20   0  379284   9084   3956 R   0.0   0.1   0:00.03 top
```

Parameters

You must handle the following options, **in any order**:

* `-U` <`username`>: allows to filter the processes shown by username.
* `-d` <`seconds[.cents]`>: modifies the delay between refreshes (in seconds. Default: 3.0).
* `-n` <`frames`>: Defines how many frames must show before the program exits (default: unlimited).

These options should behave in the same way that they do with **top**

This should show 10 frames of processes owned by user `clery` with 1.5 second interval between frames.

Terminal

```
~/B-PSU-100> ./my_top -U clery -n 10 -d 1.5
```

This should just show processes owned by user `clery`, until the program is manually stopped, with 3 second interval.

Terminal

```
~/B-PSU-100> ./my_top -U clery
```

```
top - 10:39:58 up  3:27,  0 user,  load average: 0.95, 1.01, 1.17
Tasks:   5 total,   1 running,   4 sleeping,   0 stopped,   0 zombie
%Cpu(s): 11.8 us,  5.9 sy,  0.0 ni, 82.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  11950.5 total,   8180.0 free,   2685.3 used,   1320.4 buff/cache
MiB Swap:   2048.0 total,   2048.0 free,      0.0 used.   9265.2 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   26 clery     20   0  369924   4884   2360 S   0.0   0.0   0:00.02 sh
   27 clery     20   0  375116   7544   4104 S   0.0   0.1   0:00.07 bash
   29 clery     20   0  379272   8952   3952 R   0.0   0.1   0:00.04 top
```

Features

Your program must be able to retrieve system information, and process statistics.

In the upper section of your ncurses window, you must display the system informations.
Below that, you must display an array of individual process statistics.

In the upper section you must display:

- ✓ Time of day
- ✓ How long the laptop has been up (powered on)
- ✓ How many users are currently logged in
- ✓ Load average
- ✓ The amount of Tasks total, running, sleeping, stopped or zombie
- ✓ CPU usage (*SHOULD*)
- ✓ Memory usage
- ✓ Swap usage

In the lower section, you must display informations about processes, including:

- ✓ **PID** of the process
- ✓ **USER** owning that process
- ✓ **PR**iority
- ✓ **NI**ce Value
- ✓ **VIRT**ual Memory Size
- ✓ **RES**ident Memory Size
- ✓ **SH**a**R**ed Memory Size
- ✓ Process **S**tatus
- ✓ **CPU** percent usage (*SHOULD*)
- ✓ **MEM**ory percent usage (*SHOULD*)
- ✓ **TIME** since the process has started
- ✓ The **COMMAND** name

```
top - 09:04:31 up 3 days, 21:03,  0 users,  load average: 1.29, 1.00, 0.95
Tasks:   2 total,   1 running,   1 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.5 us,  0.2 sy,  0.0 ni, 99.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  11952.6 total,    608.4 free,   1884.6 used,   9459.6 buff/cache
MiB Swap:   2048.0 total,   2048.0 free,      0.0 used.   9821.5 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   25 root      20   0  376216   6824   3456 R   0.7   0.1   0:00.04 top
    1 root      20   0  375108  10140   4352 S   0.0   0.1   0:00.05 bash
```

The image above describes how difficult the different sections are.

This can be due to multiple factors : the research needed to find out how to calculate the results, how to handle data to make it change over time, or any other reason that would make something difficult.

The uptime value can be tricky:

- ✓ it must print days only if up for at least 24 hours.
- ✓ it must print hours only if there is hours is more than 0.
- ✓ if no hours are printed, you should print `30 min`, otherwise, 2:30. `So, if system has been up for 48 hours and 30 min, you should print`up 2 days, 30 min‘.

  The only red box is because this specific value is tricky, and might require some research.

You should probably start with the easiest columns. CPU and MEM usage are not easy ones !

Formatting exactly like `top` is not mandatory.

You should also implement a number of commands, including the following:

- ✓ Typing **E** should cycle through different units of memory for processes
  Shift+E should cycle through different units of memory for the system section
  - KiB
  - MiB
  - GiB
  - TiB
  - PiB
  - EiB (only for the system section)
- ✓ Using **up and down arrows** must allow you to scroll through your process list
- ✓ Typing **K** should open a prompt to send a signal to a process
  - By default, it should offer to send the signal to the **highest process in your list**
  - By default, it should offer to send the `SIGTERM` signal
  - You are not required to provide line edition

man top
man procfs
top -> hit H on your keyboard to open the documentation

Scrolling down or up changes the **highest** process in your list. So the default PID to send a signal to with the K command **should change** !

Sending a signal should not be your highest priority. You probably want to implement that later on.

But, really, I mean it. Read the manuals for **top** and **procfs**. I swear they're useful.

Warning

For the moulinette to work, you must respect a few things:

You **MUST** implement the PID column before any other process column, without it, your project will not be evaluated.

For processes, the “title bar” must match the values underneath. So for each name in the title bar, a value must be found in the process infos. Eg:

All data is aligned with its header name, so it’s ok

```
PID USER PR

1 root 20
27 root 20
33 root 20
34 root 20
123 root 20
218 root 20
231 root 20
241 root 20
321 root 20
343 root 20
357 root 20
408 root 20
10127 root 20
10880 root 20
21875 root 20
67201 root 20
67225 root 20
69249 root 20
69963 root 20
70348 root 20
72524 root 20
72574 root 20
72642 root 20
```

PR is the 3rd header name, but 2nd value, so it’s not ok

```
PID USER PR

1       20
27       20
33       20
34       20
123       20
218       20
231       20
241       20
321       20
343       20
357       20
408       20
10127       20
10880       20
21875       20
67201       20
67225       20
69249       20
69963       20
70348       20
72524       20
72574       20
72586       20
```

On the system statistics side, make sure that every word that is not a value is the same as in `top`.

> Once again, remember: software development is an exact science. It’s important to avoid typos, and other kinds of issues that might cause trouble with the moulinette’s tests.

Bonus

There's so many possible bonuses it would be longer to enumerate them than writing the rest of the subject... But here are a few ideas:

- ✓ Sort rows by column value
- ✓ Using **<** and **>** could change which column the array is sorted by (default PID)
- ✓ Typing **Shift+R** could change the sorting direction
  - You could sort from highest to lowest value by default
- ✓ More columns ? **PPID, GROUP, UID, GID**. There's so many, just read the man you'll find some easy ones
- ✓ Scrolling horizontally ? Try it out in `top`, it's probably not that hard.
- ✓ Did you try the **Z** command ? Some colors would make that window look better.
- ✓ Forest mode ? **Shift+V**
- ✓ Threads ? **Shift+H**
- ✓ Show the entire command line ? **C**
- ✓ ...
- ✓ **htop** ?...

Unit tests

Testing a project which is using external libraries and system calls can be tedious, but it is possible to organize a project in such a way that external calls are isolated, and the core functionalities are easier to test. Also, architecturing your project to make it easily testable from the beginning usually leads to cleaner code.

Here is an example. Let's say we have a `my_top.h` file:

```
#ifndef MY_TOP_H_
    #define MY_TOP_H_

extern const char *LOADAVG_PATH;

typedef struct system_s {
    float loadavg[3];
    // Some fields
} system_t;

#endif
```

This header defines a `LOADAVG_PATH` variable to be used within our my_top to use this path as the `loadavg` file to read.

We can override this variable within our tests to specify a fixture file.

```
#include <criterion/criterion.h>
#include "my_top.h"

Test(my_top, check_loadavg)
{
    int err;
    system_t *system;

    LOADAVG_PATH = "./fixtures/loadavg";
    system = fetch_system_data(); // This function uses LOADAVG_PATH internally
    cr_assert_neq(system, NULL, "An error occured, fetch_system_data returned NULL");
    cr_assert_eq(system->loadavg[0], 2);
    cr_assert_eq(system->loadavg[1], 3);
    cr_assert_eq(system->loadavg[2], 4);
}
```

This way, we can test that the data parsing part of our my_top works the way it should, and we can be alerted by our unit tests if any other change in our code would break this functionality.

We could test other parts, such as formatting of said data, or refreshing this same data, and ensure that those work as well, without ever testing ncurses code.

v 1.0.0