

AMSA

Activity 2: Creating our own htop!

Ferran Aran Domingo

Oriol Agost Batalla

Pablo Fraile Alonso

Table of contents

1. Introduction	2
2. Delivery	3
3. Outline	3
4. Resources	4
amsatop library	4
Repository template	4
5. Tasks	5
PRAC-2.1: Getting all processes	5
Prac-2.2: Adding priorities	7
Prac-2.3: Filtering where SIGHUP is ignored	7
6. Setting the environment up!	11
Installing libfuse	11
Cloning the repository and set it up:	11
Running tests:	12
[Optional] Running tests with coverage:	12
7. Evaluation	13
8. Rules	13
9. Resources	13
10. Doubts	14

! Very very important

Please read everything carefully! The only part you may skip is Section 5, which explains the PRAC-2.1, PRAC-2.2, and PRAC-2.3 tasks (so, if you're working on PRAC-2.1, you can skip the instructions for PRAC-2.2 and PRAC-2.3). However, don't miss the important information that comes after Section 5!

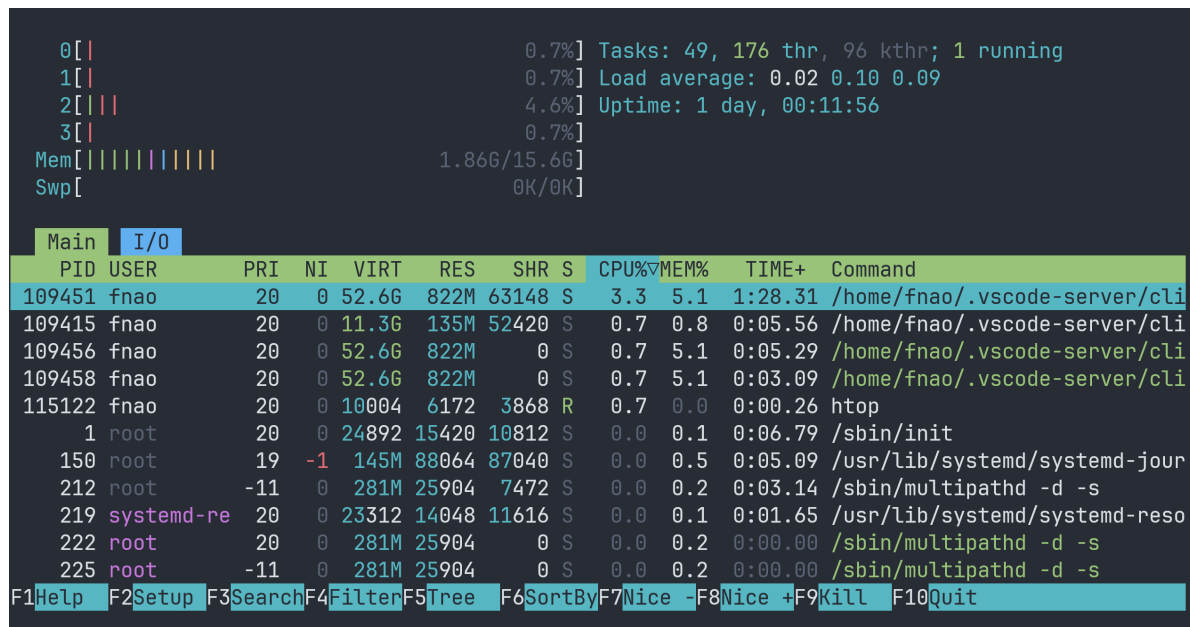
1. Introduction

You may have heard about the linux program [htop](#), it is an **interactive process viewer** for Unix-like systems.

With `htop` you can:

- See a **real-time overview** of system processes, CPU cores, memory, and swap usage.
- **Scroll and search** through processes easily.
- Interactively **sort, filter, and kill processes** with simple key commands.
- Get a quick sense of **system health** thanks to its bars and colors.

Below is a screenshot of how it looks:



```
0[|
1[|
2[|||
3[|
Mem[|||||]
Swp[

0.7%] Tasks: 49, 176 thr, 96 kthr; 1 running
0.7%] Load average: 0.02 0.10 0.09
4.6%] Uptime: 1 day, 00:11:56
0.7%]
1.86G/15.6G
0K/0K

Main I/O
PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command
109451 fnao 20 0 52.6G 822M 63148 S 3.3 5.1 1:28.31 /home/fnao/.vscode-server/cli
109415 fnao 20 0 11.3G 135M 52420 S 0.7 0.8 0:05.56 /home/fnao/.vscode-server/cli
109456 fnao 20 0 52.6G 822M 0 S 0.7 5.1 0:05.29 /home/fnao/.vscode-server/cli
109458 fnao 20 0 52.6G 822M 0 S 0.7 5.1 0:03.09 /home/fnao/.vscode-server/cli
115122 fnao 20 0 10004 6172 3868 R 0.7 0.0 0:00.26 htop
1 root 20 0 24892 15420 10812 S 0.0 0.1 0:06.79 /sbin/init
150 root 19 -1 145M 88064 87040 S 0.0 0.5 0:05.09 /usr/lib/systemd/systemd-jour
212 root -11 0 281M 25904 7472 S 0.0 0.2 0:03.14 /sbin/multipathd -d -s
219 systemd-re 20 0 23312 14048 11616 S 0.0 0.1 0:01.65 /usr/lib/systemd/systemd-reso
222 root 20 0 281M 25904 0 S 0.0 0.2 0:00.00 /sbin/multipathd -d -s
225 root -11 0 281M 25904 0 S 0.0 0.2 0:00.00 /sbin/multipathd -d -s
F1Help F2Setup F3Search F4Filter F5Tree F6SortBy F7Nice F8Nice +F9Kill F10Quit
```

On this activity you are going to build your own simplified version of `htop` by writing python code that analyzes the contents of `/proc`. Once finished, it will look like this:

O	PID	Command	Type	Priority	HtopTUI
1	1	systemd	task	20	
2	2	kthreadd	kthread	20	
3	3	pool_workqueue_release	kthread	20	
4	4	kworker/R-kvfree_rcu_reclaim	kthread	0	
5	5	kworker/R-rcu_gp	kthread	0	
6	6	kworker/R-sync_wq	kthread	0	
7	7	kworker/R-slub_flushwq	kthread	0	
8	8	kworker/R-netns	kthread	0	
10	10	kworker/0:0H-events_highpri	kthread	0	
13	13	kworker/R-mm_percpu_wq	kthread	0	
15	15	rcu_tasks_kthread	kthread	20	
16	16	rcu_tasks_rude_kthread	kthread	20	
17	17	rcu_tasks_trace_kthread	kthread	20	
18	18	ksoftirqd/0	kthread	20	
19	19	rcu_preempt	kthread	20	
20	20	rcu_exp_par_gp_kthread_worker/0	kthread	20	
21	21	rcu_exp_gp_kthread_worker/0	kthread	20	

2. Delivery

Fork the template of the assignment in Github, following the [link](#)

To complete this activity, you must do the following for each sub-delivery (PRAC-2.1, 2.2 and 2.3):

1. Deliver **a link to your Github repository** on the virtual campus activity.
2. Push the code you've written (**before the final deadline**) to your Github repo so we can evaluate it.

Tip

Remember that we're only going to evaluate your code after the final deadline, but following the recommended tempos and pushing each part of the activity accordingly can grant you extra points.

3. Outline

This activity is split onto 3 incremental parts, which are going to be related to the contents explained during the different classes.

1. On the first week, once we have explained **syscalls, processes and ProcFS**, your job will be to implement a function that retrieves a list of all the system processes.

2. Next, after having explained **process priorities**, we'll want you to get the priority of each of the processes.
3. Finally, since you'll understand the **SIGHUP signal**, the idea is that you also find out which processes are ignoring this signal or not.

4. Resources

Building an entire htop is too big of a task and out of the scope of the subject, even if it is a simplified version. For that reason, the teachers have prepared some resources that will hopefully make the activity shorter and more enjoyable.

amsatop library

A python library available on [PyPI](#) has been built with all of our love so that it already implements the UI and some abstractions. Make sure to check out the [docs](#)!

Building your solution on top of a library makes it easier to give you useful tools such as tests and [helper functions](#).

Repository template

When you join the Github Classroom activity with the link provided on Step 2, a remote repository is automatically created for you to work on this assignment. This repository will already contain some code, below are the details of what comes with the repo:

1. It already has `amsatop` library added.
2. A class `AmsaTop` with a function to be implemented for each sub-delivery located on `amsatop_solution.py`. **Here is where we expect you to write your code.**
3. Unit tests we've made for you to have some guidance (which are not exactly the same tests we're going to use to evaluate your project but they will be similar). Passing the provided tests on the [Ubuntu VM](#) is a very good indicator that you're doing it right.
4. A `README.md` file where you'll have instructions on how to run the provided tests and linter, and how to set up the python environment.

5. Tasks

Below is a detailed explanation of what you have to do on each task, but before reading through we want to make something clear:

On each of the following tasks you'll be obtaining a list of processes and then looking for further information on these processes.

Notice that if I get a list of all the processes currently active on my system, and then I go fetch information on them, it may have happened that **one of these processes has ended and its information is no longer available**. Your code has to be prepared for this scenario and be prepared to not find any information about a given process because it is no longer running and thus there is nothing for that process inside `/proc`.

In that case just exclude the process from the returned list. Be aware that some tests already check that, so don't bother too much.

PRAC-2.1: Getting all processes

Implement [the `get\_processes` method](#) so it returns a list where each element is an instance of `Process`. There has to be an element on the list for each **process, thread and kthread** running on the system.

This part has to be done exclusively by reading the raw contents of `/proc`. It is **strictly forbidden to use any external libraries or commands**.

When creating instances of `Process`, notice that the field `priority` can be `None`. As stated on the docs:

Can be `None` if unavailable or you're doing Prac-2.1.

When working on Prac-2.1 just set that field to `None`.

Tip

Make sure to read through [the docs](#) of the helper classes and functions, we promise using them will make your life easier !

Very very important

It is **very important** that you make use of the property `proc_folder` of the `Htop` class when accessing files on `"/proc"`. **Do not use a hardcoded string `"/proc"`**, instead use `self.proc_folder` (which is a string that defaults to `"/proc"`). This is essential for the tests to work.

Here is an example of **what NOT to do**:

```
1 class Amsatop(Htop):
2     def __init__(self):
3         super().__init__()
4
5     def get_processes(self) -> List[Process]:
6         for entry in os.listdir("/proc"): # Do NOT do it like this
7             if ...
```

Here is an example of **what you should do**:

```
1 class Amsatop(Htop):
2     def __init__(self):
3         super().__init__()
4
5     def get_processes(self) -> List[Process]:
6         for entry in os.listdir(self.proc_folder): # Do it like this
7             if ...
```

Hints about how to get the different processes/threads/kthreads from /proc.

As stated on the [week-2 slides](#), there are different ways to get information about /proc. We recommend you to use the following ones:

- To detect processes or kernel threads, the algorithm is “trivial”, the idea is:
 - 1) List the contents of the proc directory, and filter the ones that are digits (pids/tgid).
 - 2) Now you have to filter if they’re a process or kernel thread. We recommend you to [apply the mask](#) that the [htop code applies to the flags content](#) to see if they are a kernel thread ¹.
 - If it’s a kernel thread, add it to the list and stop analyzing.

¹Remember, that in c, applying a mask would be

```
1 /* Not exposed yet. Defined at include/linux/sched.h */
2 #ifndef PF_KTHREAD
3 #define PF_KTHREAD 0x00200000
4 #endif
5 // More code here...
6 if (lp->flags & PF_KTHREAD) {
7     proc->isKernelThread = true;
8 }
```

In python, the **exact** same code would look like this:

- If it's a process, add it, but be aware that can have threads spawned (next point will explain this).
- To detect threads, it's a little bit more complicated, you should:
 - For each detected process, inspect the tasks folder. Be aware that:
 - * This folder will have, at least, one entry, which should be equal to the pid of the current process.
 - * If it has more than one entry, it means it has some thread spawned.

Prac-2.2: Adding priorities

Implement `get_priorities` method so it returns a list of `Process`. It has to be the same list we obtained on **Prac-2.1** but now we want you to set the value of the `priority` field if available.

Tip

There are no hints here.... It should be pretty easy to implement if you already have Prac-2.1 solved.

Play with the `nice` command so you can see how the priorities change on real time!

Don't worry too much!

It's normal to see some "strange" priorities on kernel threads!!

Prac-2.3: Filtering where SIGHUP is ignored

Implement `the get_hup` method so it returns a list of `Process`. It has to be the same list we obtained on **Prac-2.2 (with the priorities)** but now we want to add a filter so only those processes that are ignoring the SIGHUP signal are returned, those that are not ignoring it are expected to **not be present on the returned list**. Be aware that **ignoring a signal is not the same as handling a signal**.

```

1 ...
2 PF_KTHREAD = 0x00200000
3 if (lp.flags & PF_KTHREAD) != 0:
4     proc.isKernelThread = True

```

Hints about detecting signals

As we've seen in class, processes can **ignore signals**. But how can we find out which signals a process is ignoring?

Lukily, Linux gives us this information!

One of the files inside the process `/proc/{pid}` has a `SigIgn` (signal ignore) field which shows the set of signals the process is currently ignoring. Retrieving this information works in much the same way as checking whether a PID/TID corresponds to a kernel thread (by examining and interpreting bitmasks). This field could be for example:

```
SigIgn: 0000000000000005
```

This might look like a strange number, but it's actually quite simple!

Step 1: Understanding SigIgn

`SigIgn` is a **bitmap represented in hexadecimal**.

A bitmap is just a binary number where **each bit represents something different**.

In our case, each bit represents one signal:

- 1 means the signal is being ignored.
- 0 means the signal is NOT being ignored.

The bits correspond to signal numbers, **counting from right to left**:

```
Signal:  8 7 6 5 4 3 2 1
          | | | | | | |
Binary:  0 0 0 0 0 1 0 1
```

In this example:

- Signal 1 (`SIGHUP`) is ignored.
- Signal 2 (`SIGINT`) is NOT ignored.
- Signal 3 (`SIGQUIT`) is ignored.

Note

Remember that `SigIgn` is written in **hexadecimal**, not binary!
For example, hexadecimal 5 is 101 in binary. The zeros on the left don't change the number.

Step 2: How do we check a specific signal?

Imagine that we want to know whether **signal 3** is ignored.

We could convert **SigIgn** into binary and look at the third bit from the right.

But there's an easier way to do this in Python: **using a mask!**

A mask is simply another binary number that has a 1 in the position we want to check, and 0 everywhere else.

For example:

```
Signal:  8 7 6 5 4 3 2 1
         | | | | | | |
Mask:    0 0 0 0 0 1 0 0
```

Since we're interested in signal 3, our mask is 00000100, which is **4 in decimal**.

How can we calculate the mask for any signal?

Notice this pattern:

Signal number	Mask in binary	Mask in decimal
1	00000001	1
2	00000010	2
3	00000100	4
4	00001000	8
5	00010000	16

Every time we move one position to the left, the value doubles. Therefore, we can calculate the mask for signal number N with:

$$\text{mask} = 2^{N-1}$$

Step 3: Using the mask to check the signal

Now that we have our mask, we are going to use the **bitwise AND operator (&)**.

This operator compares two binary numbers, bit by bit. The result has a 1 only in positions where **both numbers have a 1**.

Let's try it with our example:

```

SigIgn:  0 0 0 0 0 1 0 1    (5)
Mask:    0 0 0 0 0 1 0 0    (4)
-----
Result:  0 0 0 0 0 1 0 0    (4)

```

The result is **4**, which means that the bit corresponding to signal 3 was set to 1, so **signal 3 is ignored**.

What if we check signal 2 instead?

```

SigIgn:  0 0 0 0 0 1 0 1    (5)
Mask:    0 0 0 0 0 0 1 0    (2)
-----
Result:  0 0 0 0 0 0 0 0    (0)

```

The result is **0**, meaning signal 2 is NOT ignored.

Tip

The important thing to remember:

- If SigIgn & mask is **zero**, the signal is NOT ignored.
- If SigIgn & mask is **not zero**, the signal IS ignored.

Example: Using a bitmask in Python

```

number = 11 # Value of the bitmap
mask   = 4  # We want to check the second bit

result = number & mask

if result != 0:
    print("The bit is set to 1!")
else:
    print("The bit is set to 0!")

```

The & operator compares each bit:

```

  1011    (number)
& 0010    (mask)
-----
  0010    (result)

```

If the result is **not zero**, the bit we're checking is set to 1.

Step 4: Applying this to PRAC-2.3

Your task is to use this idea to filter the processes obtained in PRAC-2.2, returning **only the processes that ignore SIGHUP**.

! Important

Ignoring a signal is NOT the same as handling it!

- `SigIgn` tells us which signals are **ignored**.
- `SigCgt` tells us which signals have a **custom handler**.

For PRAC-2.3, we only care about `SigIgn`.

Keep in mind that [each signal corresponds to a number...](#)

! Don't worry too much!

It's normal to see some `kthreads` ignoring SIGHUP!! Don't worry too much about it!

6. Setting the environment up!

Installing libfuse

We use a Linux feature called FUSE in our tests. This lets us emulate the special capabilities of the `/proc` filesystem without bothering you (the student) with the technical details of our test setup.

For this reason, in the virtual machine's command line, **you must run the following command for the tests to work:**

```
sudo apt update && sudo apt install libfuse2t64
```

Cloning the repository and set it up:

Remember that, once you forked the Github repository on [section 2](#), you will have to clone your repository with ²:

²If you don't have any idea of what we're talking about, revisit the [setting up the VM and brief git summary](#)

```
git clone <ssh_address_of_your_git_repository>
```

Then, enter inside the folder of the repository and setup a virtual environment for the python project, this will be done with:

```
uv sync
```

And then, you can enable the virtualenv on your current shell (zsh or bash) with:

```
source .venv/bin/activate
```

Running tests:

You can run all the tests we give to you with the following command (from the project root path):

```
uv run pytest
```

You can also run specific test-suites for each prac, for example:

```
# For prac 2.1:
uv run pytest test/prac_2_1

# For prac 2.2:
uv run pytest test/prac_2_2

# For prac 2.3:
uv run pytest test/prac_2_3
```

[Optional] Running tests with coverage:

If you want to see if all the lines of your code are covered by the tests (this is usually a good indicator), you can run a test coverage, this will display information about the relationship between your code and our tests! Run:

```
uv run coverage run -m pytest -v -s
uv run coverage report
```



Tip

Learn more about [pytest options here](#)

7. Evaluation

Your final score will come from various parts:

- The tests we give you are passing on the Ubuntu VM → 50%.
- The linter (ruff) runs without errors → 5%.
- Our tests (you won't have access to them) are passing on the Ubuntu VM → 25%.
- Best practices are used → 20%.



Tip

Keep in mind that the first two items (tests and linter) can be automatically verified using [Github Actions on your repository](#)

8. Rules

1. **No external python libraries** can be used, only `amsatop` and built-in libraries are allowed (e.g `os`, `typing`).
2. It is **forbidden to make python execute a Linux command** that will provide the information about processes.
3. The process information needed can **only be obtained by reading the contents of `/proc` and parsing it out** (remember you have [helpers](#) for that).
4. Make sure the tests are passing on the [Ubuntu VM](#), we'll pass them on the VM too.
5. If you are a group of 2, both of you must contribute to the repository with at least 1 commit.

9. Resources

- [Python for beginners](#)
- [Pytest for beginners](#)
- [Executing Github Actions on your repository for knowing your mark](#)

10. Doubts

Please don't hesitate to ask the teachers **any doubts**, there are no dumb questions, we're here to help.

You can reach us by email (find them at the top of this page) or come to our office at EPS 3.07 (we're here mostly during mornings).