

Operating System

PR1

Name	Daniel
Surnames	Maestre Sánchez
UOC Username	dmaestresan

Content

Introduction	2
Module 2	3
Module 3: Memory.....	13
Module 4: In/Out.....	25
Bibliography	29

Introduction

I think a good practice before starting the practice is to know what system I'm working on to know all the possibilities and how to approach the practice. Apart from being able to see it from virtualbox, what I have done has been to install a tool called inxi, that takes out the most important information of your operating system, in the image you can see the most important data which are information of the system, machine and CPU.

```
daniel-uoc@daniel-uoc-VirtualBox: ~/Desktop
daniel-uoc@daniel-uoc-VirtualBox:~/Desktop$ inxi -Fxz
System:
  Kernel: 6.8.0-48-generic arch: x86_64 bits: 64 compiler: gcc v: 13.2.0
  Desktop: GNOME v: 46.0 Distro: Ubuntu 24.04.1 LTS (Noble Numbat)
Machine:
  Type: Virtualbox System: innotek GmbH product: VirtualBox v: 1.2
  serial: <superuser required>
  Mobo: Oracle model: VirtualBox v: 1.2 serial: <superuser required>
  BIOS: innotek GmbH v: VirtualBox date: 12/01/2006
Battery:
  ID-1: BAT0 charge: 40.5 Wh (81.0%) condition: 50.0/50.0 Wh (100.0%)
  volts: 10.0 min: 10.0 model: innotek 1 status: discharging
CPU:
  Info: quad core model: AMD Ryzen 7 4800H with Radeon Graphics bits: 64
  type: MCP arch: Zen 2 rev: 1 cache: L1: 256 KiB L2: 2 MiB L3: 8 MiB
  Speed (MHz): avg: 2895 min/max: N/A cores: 1: 2895 2: 2895 3: 2895 4: 2895
  bogomips: 23156
  Flags: ht lm nx pae sse sse2 sse3 sse4_1 sse4_2 sse4a ssse3
Graphics:
  Device-1: VMware SVGA II Adapter driver: vmwgfx v: 2.20.0.0 bus-ID: 00:02.0
  Display: wayland server: X.Org v: 23.2.6 with: Xwayland v: 23.2.6
  compositor: gnome-shell driver: dri: swrast gpu: vmwgfx
  resolution: 1920x969~60Hz
  API: EGL v: 1.5 drivers: kms_swrast,swrast platforms:
```

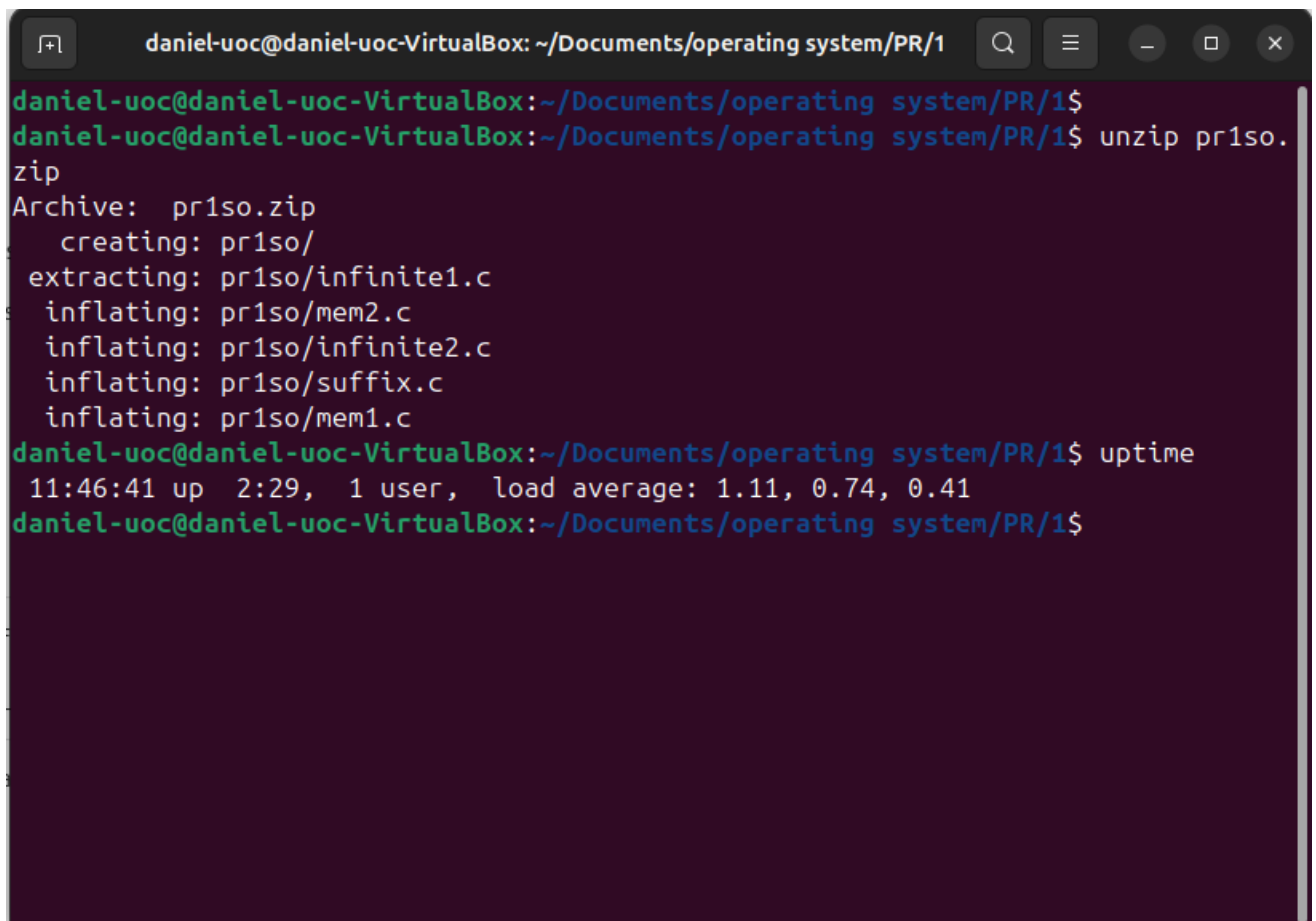
Module 2

We provide the programs count1.c, count2.c and the shellscript launch.sh (you do not need to analyse how they are implemented).

1.1 Run the uptime command on a Unix system. Attach a screenshot of the result obtained. Interpret the meaning of the information displayed, attempting to relate it to the concepts studied in this course.

I have installed mate-system-monitor -view to be able to know how my system works, this is just informative to compare data. It is like a Windows task manager, just to compare the values and to get better knowledge.

In the following image we can see how I run the unzip command on the pr1so.zip folder to unzip the files it has. Creating the folder pr1so and where the documents infinite1.c, mem2.c, infinite2.c, suffix.c and mem1.c are located.



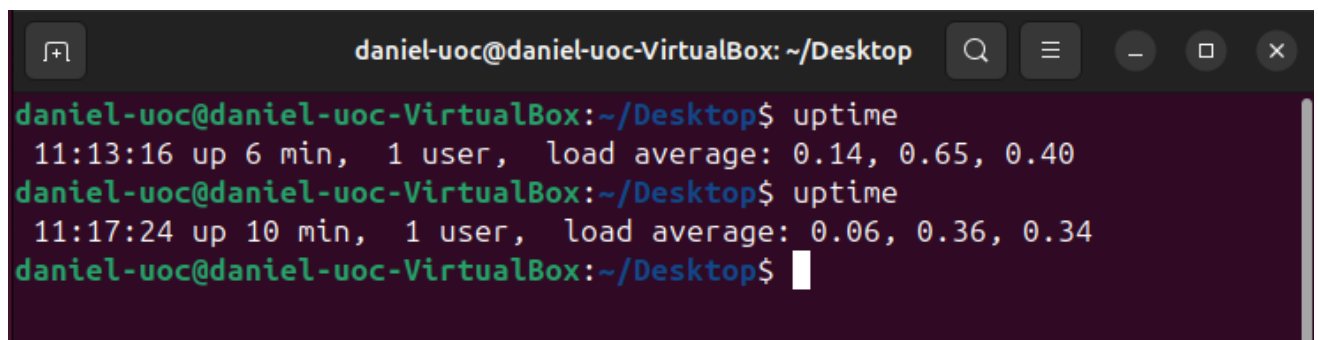
```
daniel-uoc@daniel-uoc-VirtualBox: ~/Documents/operating system/PR/1
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1$ unzip pr1so.zip
Archive:  pr1so.zip
  creating:  pr1so/
 extracting:  pr1so/infinite1.c
 inflating:  pr1so/mem2.c
 inflating:  pr1so/infinite2.c
 inflating:  pr1so/suffix.c
 inflating:  pr1so/mem1.c
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1$ uptime
 11:46:41 up  2:29,  1 user,  load average: 1.11, 0.74, 0.41
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1$
```

Next, as the objective of this exercise, I run the uptime command at 11:46:41, where the system has been running for 2 hours and 29 minutes uninterruptedly (up 2:29), only has one user and where the average system load in the last 1, 5 and 15 minutes respectively is 1.11, 0.74 and 0.41. We can see that there has been an overload on the system in the last minute with respect to the previous minutes. Where it has used more CPU resources.

Explanation of values:

- 1 minute (1.11): This value indicates an average load slightly above 1, suggesting that there has been a heavy load in the last minute. A value of 1 means that the system is using all of its available resources, while values above 1 may suggest that processes are competing for resources (especially if there is only one CPU core).
- 5 minutes (0.74): The average load over the last 5 minutes is lower, suggesting that the system has been less loaded in this interval. This could reflect a decrease in the number of processes or CPU demand in the last 5 minutes
- 15 minutes (0.41): Finally, the average load for the last 15 minutes is quite low, indicating that the system has been working with few resources in this interval.

I wanted to point out that this exercise was altered because the unzip process was done when the virtual machine was started after being hibernated. That's why you see an excessive value of 1.11 in the average CPU load value. The next day I waited 6 minutes to make sure that the virtual machine had no activity after the boot and that's why the values are different as you can see in the following image.



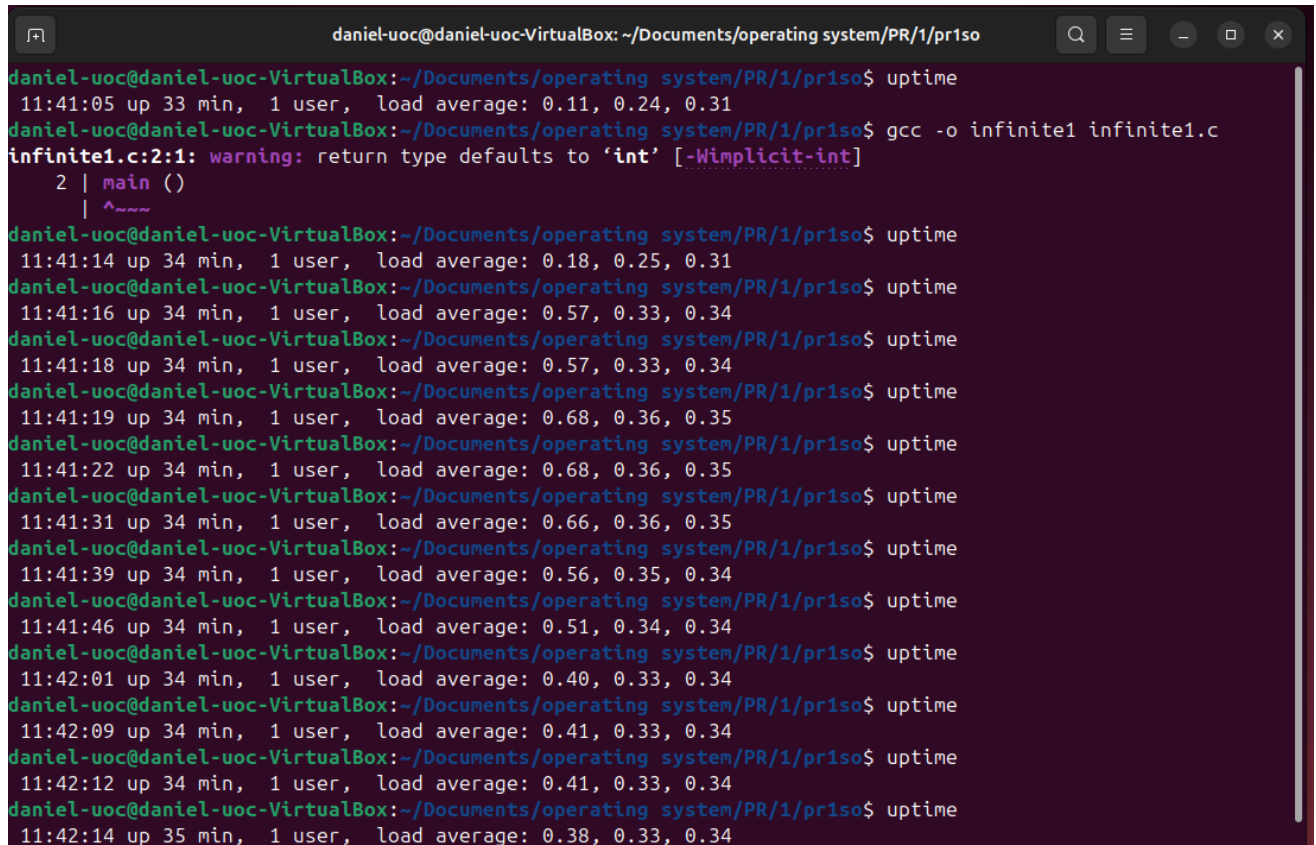
```

daniel-uoc@daniel-uoc-VirtualBox: ~/Desktop
daniel-uoc@daniel-uoc-VirtualBox:~/Desktop$ uptime
11:13:16 up 6 min,  1 user,  load average: 0.14, 0.65, 0.40
daniel-uoc@daniel-uoc-VirtualBox:~/Desktop$ uptime
11:17:24 up 10 min,  1 user,  load average: 0.06, 0.36, 0.34
daniel-uoc@daniel-uoc-VirtualBox:~/Desktop$

```

Wait 4 minutes and 8 seconds as you can see in the picture above, the average CPU load uptime is much lower. So, the comparison values I would use with the rest of the exercise would be.

1.2 We provide you with the program `infinite1.c`; this program runs an infinite loop. Compile it and run it from another window. From the original window, run the `uptime` command multiple times (about 5 times in one minute). Attach the screenshots of the results shown by the `uptime` command.



```
daniel-uoc@daniel-uoc-VirtualBox: ~/Documents/operating system/PR/1/priso
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
11:41:05 up 33 min,  1 user,  load average: 0.11, 0.24, 0.31
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ gcc -o infinite1 infinite1.c
infinite1.c:2:1: warning: return type defaults to 'int' [-Wimplicit-int]
  2 | main ()
    | ^~~~~
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
11:41:14 up 34 min,  1 user,  load average: 0.18, 0.25, 0.31
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
11:41:16 up 34 min,  1 user,  load average: 0.57, 0.33, 0.34
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
11:41:18 up 34 min,  1 user,  load average: 0.57, 0.33, 0.34
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
11:41:19 up 34 min,  1 user,  load average: 0.68, 0.36, 0.35
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
11:41:22 up 34 min,  1 user,  load average: 0.68, 0.36, 0.35
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
11:41:31 up 34 min,  1 user,  load average: 0.66, 0.36, 0.35
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
11:41:39 up 34 min,  1 user,  load average: 0.56, 0.35, 0.34
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
11:41:46 up 34 min,  1 user,  load average: 0.51, 0.34, 0.34
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
11:42:01 up 34 min,  1 user,  load average: 0.40, 0.33, 0.34
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
11:42:09 up 34 min,  1 user,  load average: 0.41, 0.33, 0.34
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
11:42:12 up 34 min,  1 user,  load average: 0.41, 0.33, 0.34
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
11:42:14 up 35 min,  1 user,  load average: 0.38, 0.33, 0.34
```

What differences do you observe compared to the screenshot from the first section?

Average CPU load in the last 1.5 and 15 minutes of exercise 1.1: 0.06, 0.36, and 0.34, respectively.

As we can see these values in the previous section are low, so the system is not heavily loaded as the CPU resources are being used lightly. This is normal because no CPU-intensive process is running.

After compiling and running the `infinite1.c` program, the average load values change dramatically over a 1 minute interval. Starting from 0.18 until you reach peak values of 0.57, 0.68, 0.66. Indicating a considerable increase in the average CPU load. I also did an `uptime` before running so you can see what the virtual machine was like at idle.

We can also see less significant increases in the 5 and 15 minute environment. In the 5 minute average starting at 0.25 going up to 0.36 and in the 15 minute we see a variation of 0.31 to 0.35 environment.

The clearest difference is that there has been a considerable increase in load. In the first section we can see that the values are 0.06, 0.36, and 0.34. In the second section due to the `infinite1.c` program we have seen a peak in the average CPU load of 0.68, 0.36 0.35, where we have seen a considerable increase as I discussed earlier.

What are the reasons for these differences?

The execution of `infinite1.c` generates an infinite loop that consumes CPU resources without performing useful operations. This causes an increase in the average load in the last minute due to constant CPU occupancy. Being in an infinite loop, `infinite1.c` does not relinquish control, which immediately impacts the 1-minute average.

1.3 We provide you with the program `infinite2.c`, which also runs an infinite loop.

Analyse its source code: how does it differ from `infinite1.c`? Compile it and run it from another window. From the original window, run the `uptime` command multiple times (about 5 times in one minute). Attach the screenshots of the results shown by the `uptime` command.

The difference between `infinite2.c` and `infinite1.c` is simply that it has one more line of code which is `sleep(1)`, this line gives 1 second of pause to the program in each iteration of the loop. So, it gives rest to the CPU during that time. That is to say, `infinite2.c` does not overload the CPU as much as `infinite1.c` does. This is because `infinite1.c` consumes CPU without interruptions.

As we can see in the following image there are two terminals, a new one located on the left where the `infinite2` program is executed and the start and end time is observed, with the `uptime` commands, this is simply to have a time reference. And then on the right the original window with all the uptimes. We can also see two white boxes with the different programs `infinite1` and `infinite2`.


```

daniel-uoc@daniel-uoc-VirtualBox: ~/Documents/operating system/PR/1/priso
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ gcc -o infinite2 infinite2.c
infinite2.c:2:1: warning: return type defaults to 'int' [-Wimplicit-int]
  2 | main ()
    | ^~~~~
infinite2.c: In function 'main':
infinite2.c:5:5: warning: implicit declaration of function 'sleep' [-Wimplicit-function-declaration]
  5 |     sleep(1);
    |     ^~~~~
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
12:00:53 up 53 min,  1 user,  load average: 0.91, 1.12, 0.64
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
12:02:05 up 54 min,  1 user,  load average: 0.53, 0.93, 0.61
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$

```

```

daniel-uoc@daniel-uoc-VirtualBox: ~/Documents/operating system/PR/1/priso
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
11:42:12 up 34 min,  1 user,  load average: 0.41, 0.33, 0.34
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
11:42:14 up 35 min,  1 user,  load average: 0.38, 0.33, 0.34
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
12:00:48 up 53 min,  1 user,  load average: 0.99, 1.14, 0.64
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
12:00:56 up 53 min,  1 user,  load average: 0.83, 1.10, 0.64
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
12:00:57 up 53 min,  1 user,  load average: 0.83, 1.10, 0.64
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
12:00:59 up 53 min,  1 user,  load average: 0.83, 1.10, 0.64
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
12:01:00 up 53 min,  1 user,  load average: 0.77, 1.08, 0.63
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
12:01:03 up 53 min,  1 user,  load average: 0.77, 1.08, 0.63
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
12:01:04 up 53 min,  1 user,  load average: 0.71, 1.06, 0.63
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
12:01:19 up 54 min,  1 user,  load average: 0.62, 1.03, 0.62
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
12:01:31 up 54 min,  1 user,  load average: 0.53, 0.99, 0.62
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
12:01:38 up 54 min,  1 user,  load average: 0.48, 0.98, 0.61
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
12:01:44 up 54 min,  1 user,  load average: 0.41, 0.94, 0.61
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ uptime
12:01:50 up 54 min,  1 user,  load average: 0.38, 0.93, 0.60

```

```

main ()
{
    while (1);
}

```

```

main ()
{
    while (1)
        sleep(1);
}

```

What differences do you observe compared to the screenshot from the second section?

In the previous section (1.2) with the `infinite1.c` program, the average CPU load increased because the program was constantly using the CPU. However, when running `infinite2.c`, which includes `sleep(1);`, the average load decreases, as the process stops using the CPU for 1 second in each cycle of the loop.

What this means is that the `infinite2.c` program has a lower average load because of the rest it gives the CPU, as seen in the average CPU load values for the 1 minute average. I do notice the initial peak of 0.99, which drops rapidly as the minute passes.

Unlike the 5 and 15 minute indicators, they do not change as fast because they reflect a cumulative average; however, overall, the load is lower compared to `infinite1.c`.

What are the reasons for these differences?

The reason for these differences is that `infinite2.c` includes `sleep(1);`, which means it allows the process to pause for 1 second at each iteration. This gives the CPU a rest during that time, significantly reducing the average CPU load, especially at the 1 minute interval, which is where you see the biggest difference. In contrast, `infinite1.c` runs a loop without pauses, which increases the CPU load is completely different.

1.4 Using the ps command (properly parameterized) and filters such as grep, wc, sort, cut, uniq, ..., answer the following questions: (Indicate which command you would execute to answer each question).

Before I start answering these questions, I have been researching the ps (properly parameterized) command, I have seen that depending on the system you are using one type of command is more efficient than the other, between syntax standards or BSD syntax. As I have the system information and I know that the kernel is using version 6.8.0-48-generic, this confirms that I am on a Linux based system and not on a BSD or macOS system.

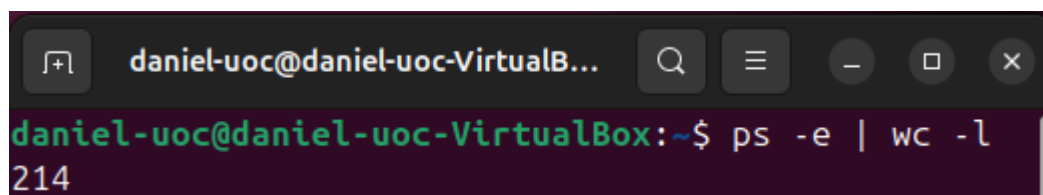
This does not mean that they are equally useful, I just think, after reading manuals, that it is more convenient to use those commands specifically designed for the standard system.

Once I have confirmed which system I am using, I can orient myself towards which types of commands are more efficient, which are the ones I have used to solve this exercise.

a) How many processes are running on the machine?

To answer this question we will use the command **ps -e | wc -l**, it tells us in a whole number how many processes are currently running.

In this case it would be 214, as we can see in the image.



```
daniel-uoc@daniel-uoc-VirtualB...
daniel-uoc@daniel-uoc-VirtualBox:~$ ps -e | wc -l
214
```

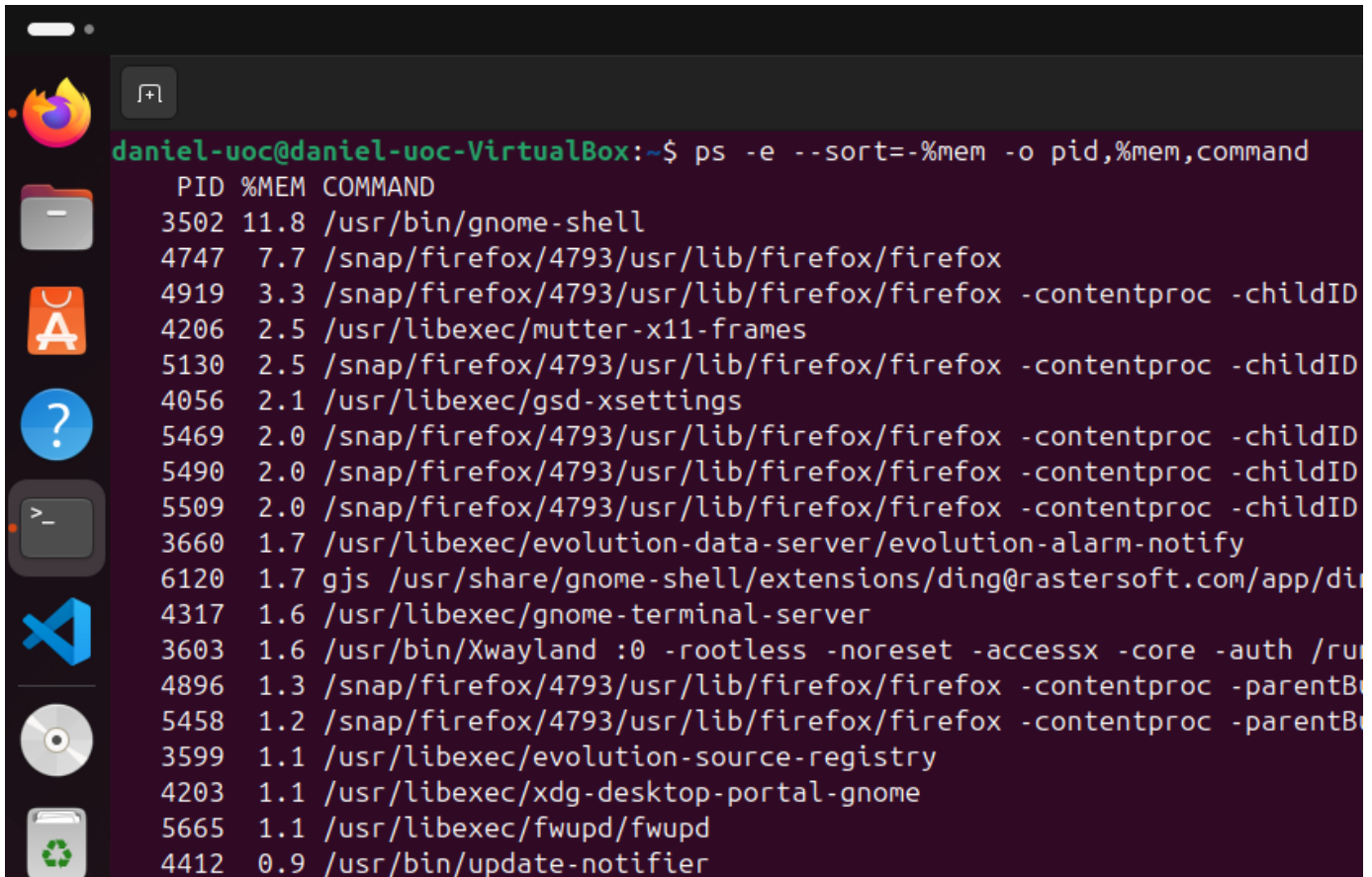
Explanation of the command part by part:

- ps, is the command to use.
- -e, is used to see all the processes that are running, line by line.
- | It is used to use the previous functionality and structure it with a new command. In order to sort the output of ps -e to your liking.
- Wc (word coun), is used to know the number of lines, words and characters the previous output has.
- -l is used to find out the exact number of lines, in order to answer the question in the best way.

b) Which process is using the most physical memory?

To answer this question you will use the command **ps -e --sort=-%mem -o pid,user,%mem,command**, it tells you which process occupies more physical memory, it represents it in percentage of memory occupation.

In this case it is gnome-shell, 'system user interface for launching and switching applications, viewing notifications and system status. Tightly integrated with Mutter, the GNOME window manager', information obtained from the official gnome website.



```
daniel-uoc@daniel-uoc-VirtualBox:~$ ps -e --sort=-%mem -o pid,%mem,command
PID %MEM COMMAND
3502 11.8 /usr/bin/gnome-shell
4747 7.7 /snap/firefox/4793/usr/lib/firefox/firefox
4919 3.3 /snap/firefox/4793/usr/lib/firefox/firefox -contentproc -childID
4206 2.5 /usr/libexec/mutter-x11-frames
5130 2.5 /snap/firefox/4793/usr/lib/firefox/firefox -contentproc -childID
4056 2.1 /usr/libexec/gsd-xsettings
5469 2.0 /snap/firefox/4793/usr/lib/firefox/firefox -contentproc -childID
5490 2.0 /snap/firefox/4793/usr/lib/firefox/firefox -contentproc -childID
5509 2.0 /snap/firefox/4793/usr/lib/firefox/firefox -contentproc -childID
3660 1.7 /usr/libexec/evolution-data-server/evolution-alarm-notify
6120 1.7 gjs /usr/share/gnome-shell/extensions/ding@rastersoft.com/app/di
4317 1.6 /usr/libexec/gnome-terminal-server
3603 1.6 /usr/bin/Xwayland :0 -rootless -noreset -accessx -core -auth /run
4896 1.3 /snap/firefox/4793/usr/lib/firefox/firefox -contentproc -parentB
5458 1.2 /snap/firefox/4793/usr/lib/firefox/firefox -contentproc -parentB
3599 1.1 /usr/libexec/evolution-source-registry
4203 1.1 /usr/libexec/xdg-desktop-portal-gnome
5665 1.1 /usr/libexec/fwupd/fwupd
4412 0.9 /usr/bin/update-notifier
```

Explanation of the command part by part:

This command is divided into three parts, where we get the information from (ps -e), how we sort it(--sort= - - %mem) and how we want to see it (-o pid ,%mem,command).

- Ps, is the command to use.
- -e, is used to see all the processes that are running, line by line.
- --sort is a ps option that allows you to sort the output by a specific column.
- - %mem indicates that we want to sort the processes by physical memory usage in the %MEM column. It is sorted in descending order, from highest to lowest, by the - character placed before the %MEM.

- -o is used to specify the output format, in my case I want to see the pid(Process ID), the percentage of memory used and the path to the file.
 - pid: Displays the Process ID, which is a unique number assigned to each running process.
 - %mem: Displays the percentage of physical memory used by the process.
 - command: Displays the complete command that started the process, including its path if available.

c) Which process has the largest virtual memory?

Para responder a esta pregunta utilizaremos el comando `ps aux --sort= -vsz`, con esto sabremos cuales son los proceso que ocupa más a memoria.

`ps -e --sort=-vsize -o pid ,vsize,command`

```
daniel-uoc@daniel-uoc-VirtualBox: ~/Desktop
daniel-uoc@daniel-uoc-VirtualBox:~/Desktop$ ps aux --sort -vsz
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
daniel-+  6033  0.1  4.9 1189282580 174548 ?        Sl   15:09   0:18 /usr/share/code/code /home/daniel-uoc/Documents/
daniel-+  6106  0.3  5.9 1189101556 212280 ?        Sl   15:09   0:31 /usr/share/code/code --type=renderer --crashpad-
daniel-+  6141  0.2  5.2 1189062892 187824 ?        Sl   15:09   0:28 /usr/share/code/code --type=utility --utility-su
daniel-+  6152  0.1  2.6 1189061680 94204 ?         Sl   15:09   0:12 /usr/share/code/code --type=utility --utility-su
daniel-+  6151  0.1  3.1 1189061128 113620 ?        Sl   15:09   0:13 /usr/share/code/code --type=utility --utility-su
daniel-+  6077  0.1  3.4 34113848 123636 ?         Sl   15:09   0:11 /usr/share/code/code --type=gpu-process --crashp
daniel-+  6081  0.0  1.9 33845436 68000 ?         Sl   15:09   0:03 /usr/share/code/code --type=utility --utility-su
daniel-+  6040  0.0  0.3 33795272 12292 ?          S    15:09   0:00 /usr/share/code/code --type=zygote
daniel-+  6037  0.0  1.3 33795256 47488 ?          S    15:09   0:00 /usr/share/code/code --type=zygote --no-zygote-s
daniel-+  6038  0.0  1.3 33795248 47616 ?          S    15:09   0:00 /usr/share/code/code --type=zygote
daniel-+  6060  0.0  0.0 33575880 3328 ?          Sl   15:09   0:00 /usr/share/code/chrome_crashpad_handler --monito
daniel-+  3522  5.0 12.1 5064332 431864 ?        Rsl  13:58  11:46 /usr/bin/gnome-shell
daniel-+  7742  0.2  2.0 3029888 72468 ?          Sl   17:41   0:01 gjs /usr/share/gnome-shell/extensions/ding@raste
daniel-+  4126  0.0  0.7 2663096 27052 ?          Sl   13:58   0:00 /usr/bin/gjs -m /usr/share/gnome-shell/org.gnome
daniel-+  3627  0.0  0.7 2663048 26820 ?          Sl   13:58   0:00 /usr/bin/gjs -m /usr/share/gnome-shell/org.gnome
root      772  0.0  0.9 1917644 32416 ?          Ssl  13:55   0:05 /usr/lib/snapd/snapd
daniel-+  4261  0.1  3.2 1564672 114836 ?        Sl   13:58   0:15 /usr/libexec/mutter-x11-frames
daniel-+  3617  0.0  1.2 1271864 43008 ?          Ssl  13:58   0:00 /usr/libexec/evolution-source-registry
daniel-+  3859  0.0  0.6 1178084 24320 ?          Ssl  13:58   0:00 /usr/libexec/evolution-calendar-factory
daniel-+  4209  0.0  1.1 852376 41216 ?          Ssl  13:58   0:00 /usr/libexec/xdg-desktop-portal-gnome
daniel-+  3683  0.0  1.7 823288 62980 ?          Sl   13:58   0:00 /usr/libexec/evolution-data-server/evolution-ala
daniel-+  3929  0.0  0.8 760376 29824 ?          Ssl  13:58   0:00 /usr/libexec/evolution-addressbook-factory
```

Explicacion del commando paso por paso:

- Ps, is the command to use.
- a, is used to know all the system processes of both users and terminals (TTY), services and programs in the background.
- u, with this we get the information that we will see in columns such as the user name (USER), the percentage of CPU usage (%CPU), the percentage of memory usage (%MEM), the size of virtual memory (VSZ), the size of resident memory (RSS), the status (STAT), the execution time (TIME), and the command that started the process (COMMAND).
- x This option includes processes that are not associated with a controlling terminal.
- - - sort, this is used to sort the processes by -vsz.

- -vsz, This is the column representing the size of virtual memory used by each process in kilobytes. Put minus(-) to display the processes from most to least according to virtual memory usage.

d) Which users have processes running?

To answer this question, we will use the command **ps -e -o user= | sort | uniq**, with this we will know which are the unique users that are currently running.

In the image we can see which are those users.



```
daniel-uoc@daniel-uoc-VirtualBox: ~
daniel-uoc@daniel-uoc-VirtualBox:~$ ps -e -o user= | sort | uniq
avahi
colord
cups-browsed
daniel-uoc
gnome-remote-desktop
kernoops
lp
messagebus
polkitd
root
rtkit
syslog
systemd-oom
systemd-resolve
```

Explanation of the command part by part:

- Ps, is the command to use.
- -eo user, is used to extract only the users column of all running processes.
- | is used to use the above functionality and structure it with a new command. In order to sort the output of ps -eo user to your liking.
- Sort: is a ps option that allows you to sort the output according to a specific column.
- | : It is used to use the above functionality and structure it with a new command. In order to sort the output of ps -eo user plus sort as desired.
- Uniq: removes repetitions, leaving only one entry per user with running processes.

Module 3: Memory

2.1 In this activity, you will analyze the behavior of the program mem1.c.

Study its source code. The program reads a password from the keyboard and compares it with a specific string ("uoc"). If they are the same, it displays a message indicating that the password is correct and terminates execution. If they are different, it prompts for a new password and repeats the process. We attach the result of two executions.

```
[enricm@willy dev]$ ./mem1
Passwd? jk
Passwd? sdfsd
Passwd? uoc
Passwd OK
[enricm@willy dev]$ ./mem1
Passwd? kkk
Passwd? 01234567abc
Passwd? uoc
Passwd? kkk
Passwd? abc
Passwd OK
[enricm@willy dev]$ █
```

Compile and run the program. Verify that it behaves as in the example. Answer the following questions:

- Why, in the second execution, is the password "abc" considered correct and "uoc" not considered correct?

Before answering this question, what we will do is to analyse the programme and how it is structured.

Mem1.c

```
#include <stdio.h>
#include <string.h>

struct {
    char input[8];
    char test[8];
} data;

int main(int argc, char *argv[])
{

    sprintf(data.test, "uoc");

    do
    {
```

```

    printf("Passwd? ");
    scanf("%s", data.input);
}
while (strcmp(data.input, data.test) != 0);

printf("Passwd OK\n");
return(0);
}

```

As we can see in the program we have two libraries:

- stdio.h, which gives the input and output function (printf and scanf).
- String.h with which you have the ability to manipulate characters (strcmp).

D We define a data structure with the struct function named data. It contains two arrays of characters (input and test), each with a length of 8 characters. It allows us to store the following variables:

- input: the password entered by the user.
- test: the correct password to verify.

As usual we have the main function, which when called the first data passed is the string 'uoc' which is copied into data.test using sprintf. This sets 'uoc' as the correct password.

Then we find a do-while loop, where it has the function of prompting for the password until the user enters the correct one. We will get an on-screen prompt for 'Passwd?' by `printf("Passwd?");` to prompt the user for the password.

The `scanf("%s", data.input);` line reads the user's input and stores it in data.input.

In each while loop, it checks `strcmp(data.input, data.test)`, which compares data.input with data.test. If they are different, strcmp returns a value other than 0, which causes the loop to continue.

When the user enters the correct password, the do-while loop stops and prints 'Passwd OK' on the screen and the program ends with the value of return(0).

To answer the question, why does the password change from 'uoc' to 'abc', as seen in the example. Quick answer is that there is a buffer overflow in the variable inside the struct data.input. What this is, having a maximum of 8 characters when you enter a string longer than 8 characters scanf does not restrict the input, and the data is written outside the bounds of data.input, so it ends up impacting directly to the data.test variable that overwrites it, as we can see in the result you see in the image.


```
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/pr1so$ ./mem1
Passwd? uoc
Passwd OK
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/pr1so$ ./mem1
Passwd? kkk
Passwd? 01234567abc
Passwd? uoc
Passwd? kkk
Passwd? abc
Passwd OK
```

How a struct works, it functions as a 'box' containing several consecutive variables or data. The box has a total storage size that is the sum of the sizes that occupy the memory space of the individual data. Each field in the struct is accessible by name and is stored in a fixed location within the structure. This is important to know in order to know the real problem.

- **What would need to be done to correct the error in this program?**

I can think of three ways to solve it, I want to clarify that it is not the same to solve the programme as to make it robust. The option to use is solution 3. But, even so, answering the question I will put them. These three solutions are organised from least to most robust.

Solution 1:

The first idea I had was to change the order of the variables inside the struct, like this:

```
struct {
    char test[8];
    char input[8];
} data;
```

This way, as the only variable that can vary by a factor external to the program is the input variable, any overflow will not affect the following variables. This program is functional, but I see it as a limitation in case you want to add more variables that can overflow in the same way.

```
daniel-uoc@daniel-uoc-VirtualBox: ~/Documents/operating system/PR/1/pr1so
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/pr1so$ gcc -o mem1 mem1.c
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/pr1so$ ./mem1
Passwd? kkk
Passwd? 01234567abc
Passwd? uoc
Passwd OK
```

Solution 2:

Then I thought that I could put the internal variables inside the main, so that they don't have any conflict between them. Changing the program in this way:

```
#include <stdio.h>
#include <string.h>

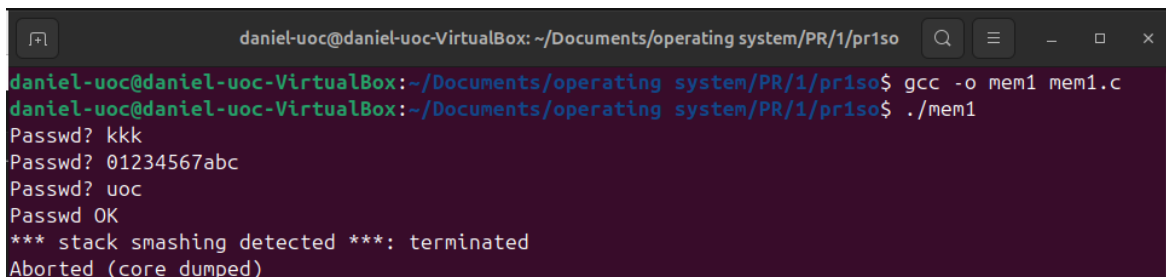
// struct {
//     char input[8];
//     char test[8];
// } data;

int main(int argc, char *argv[])
{
    char test[8];
    char input[8];
    sprintf(test, "uoc");

    do
    {
        printf("Passwd? ");
        scanf("%s", input);
    }
    while (strcmp(input, test) != 0);

    printf("Passwd OK\n");
    return(0);
}
```

You get an expected output with this program, but not what we want. The problem you have is that when you test with this code, you get a message that the program detected a stack corruption due to a buffer overflow, so it knows that more characters than allowed have been inserted in the input variable as we can see in the following image.



```
daniel-uoc@daniel-uoc-VirtualBox: ~/Documents/operating system/PR/1/priso
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ gcc -o mem1 mem1.c
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ ./mem1
Passwd? kkk
Passwd? 01234567abc
Passwd? uoc
Passwd OK
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

Solució 3:

For me, some of the important points that this solution must have are:

- There must be no buffer overflow.
- Globalise the size of the data.input / data.test, and input variables.

The most optimal solution is to cap the possibility of input characters. I have also put a central variable, BUFFER_SIZE, which defines the size of the variables data.test, data.input. On the other hand, I have created a function clear_input_buffer that has the objective to clear the input buffer if the user enters more characters than the program expects.

This code will be in the delivery folder with the name of mem1_modified.c.

Code:

```
#include <stdio.h>
#include <string.h>

#define BUFFER_SIZE 8

struct {
    char input[BUFFER_SIZE];
    char test[BUFFER_SIZE];
} data;

void clear_input_buffer()
{
    int ch;
    while ((ch = getchar()) != '\n' && ch != EOF);
}

int main(int argc, char *argv[])
{
    // Initialise the correct password
    snprintf(data.test, BUFFER_SIZE, "uoc");

    do
    {
        printf("Passwd? ");

        // We use fgets to read the user's complete input.
        if (fgets(data.input, BUFFER_SIZE, stdin) != NULL)
        {
            // If the entry was longer than BUFFER_SIZE - 1, clear
            the rest of the buffer.
        }
    } while (1);
}
```

```

        if (data.input[strlen(data.input) - 1] != '\n') {
            clear_input_buffer();
        }

        // Remove line break at the end if present
        data.input[strcspn(data.input, "\n")] = '\0';
    }
}
while (strcmp(data.input, data.test) != 0);

printf("Passwd OK\n");
return 0;
}

```

Explanation of changes:

- **Funtion clear_input_buffer:** This function is used to clear the input buffer if the user enters more characters than allowed. When fgets is used with a length of BUFFER_SIZE, any excess characters remain in the input buffer. The **clear input buffer** function makes sure to empty that excess, preventing it from affecting the next input by removing any residue that might interfere.
How it works:
 - Each additional character is read until a linefeed (\n) or the end of file (EOF) is encountered.
EOF: EOF (End Of File) is a Boolean parameter useful to facilitate closing data extraction loops from file.
- **BUFFER_SIZE:** Define a global size of BUFFER_SIZE that allows us to change the buffer length centrally without modifying multiple lines of code. This value also ensures that both data.input and data.test are limited to this length while maintaining consistency in the program. Not an apparent improvement, but just in case you need to extend the size of the variables.
- **Change from scanf to fgets:** Unlike scanf, which does not limit the length of the characters it reads, fgets allows you to specify the maximum input length (BUFFER_SIZE), being one of my goals. This automatically avoids the buffer overflow problem, as fgets reads up to BUFFER_SIZE - 1 characters and leaves space for the null character (\0) at the end of the string.
- **Line break removal (\n):** At the end of the input in data.input, strcspn is used to find and remove the line break (\n), if present. This ensures that only the string entered by the user is stored, with no additional characters. I have thought of this, because since the input is done by the terminal when you enter a possible

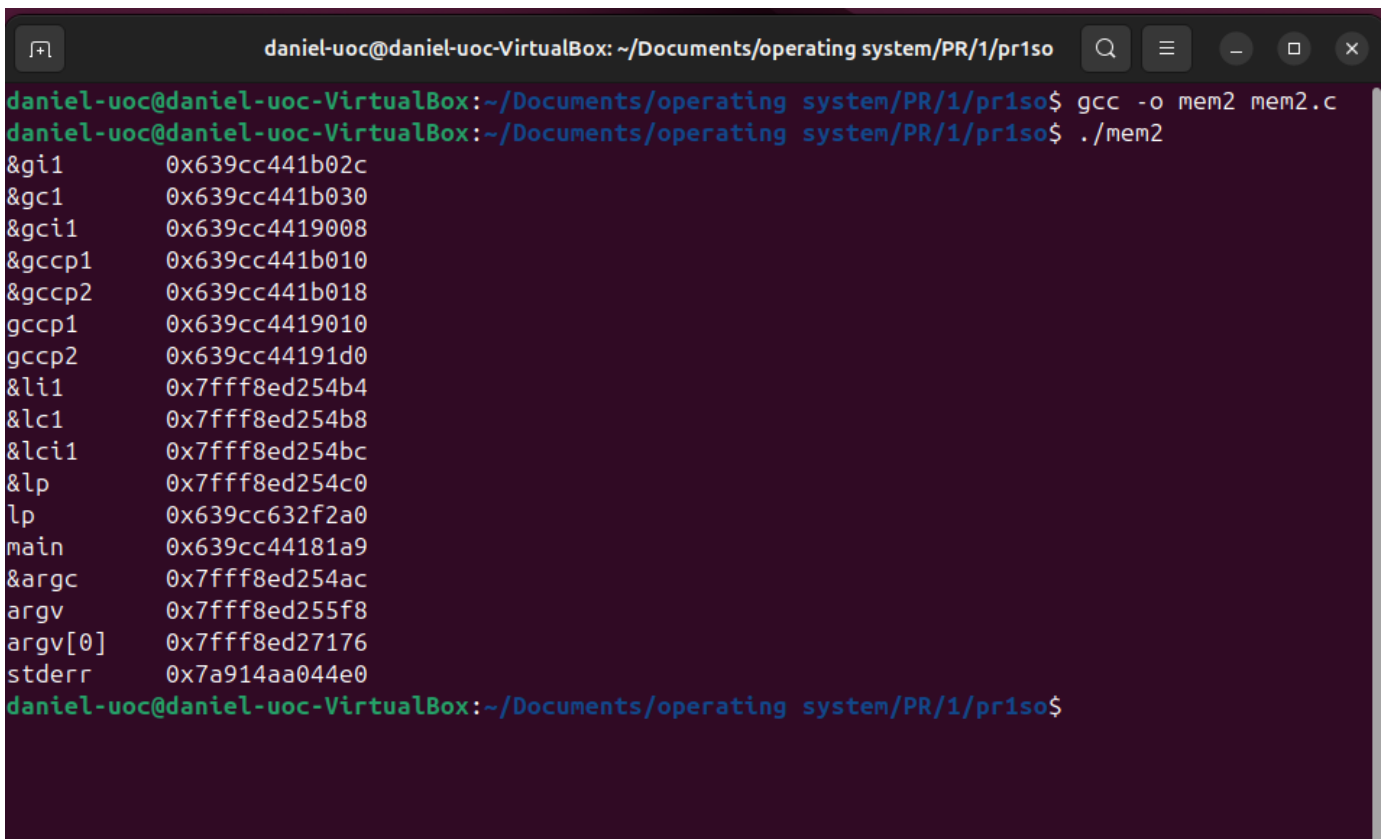
password, the user presses enter thus generating a line break to tell the program. It is very important this line in the program because the strcmp function makes an exact comparison, so 'uoc' and 'uoc' are not considered equal.

2.2 In this activity, you will analyze the behavior of the program mem2.c.

Study its source code. The program declares a series of objects: global and local variables, as well as constants. Finally, it prints the memory locations where these objects are stored. Compile and run the program.

- **Attach a screenshot of the execution result.**
- **Why are some objects grouped in nearby memory locations? Deduce in which region of the process's logical space each object is located.**

In the following screenshot is a terminal output of the mem2.c program. In this screenshot we can see 4 clear groups of how the memory is organised. They are as follows:



```
daniel-uoc@daniel-uoc-VirtualBox: ~/Documents/operating system/PR/1/pr1so
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/pr1so$ gcc -o mem2 mem2.c
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/pr1so$ ./mem2
&gi1      0x639cc441b02c
&gc1      0x639cc441b030
&gci1     0x639cc4419008
&gccp1    0x639cc441b010
&gccp2    0x639cc441b018
gccp1     0x639cc4419010
gccp2     0x639cc44191d0
&li1      0x7fff8ed254b4
&lc1      0x7fff8ed254b8
&lci1     0x7fff8ed254bc
&lp       0x7fff8ed254c0
lp        0x639cc632f2a0
main      0x639cc44181a9
&argc     0x7fff8ed254ac
argv      0x7fff8ed255f8
argv[0]   0x7fff8ed27176
stderr    0x7a914aa044e0
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/pr1so$
```

Group 1:

&gi1: 0x639cc441b02c : Int type, 4 bytes.
 &gc1: 0x639cc441b030 : Type char, 1 byte.
 &gci1: 0x639cc4419008 : Type const int, 4 bytes.
 &gccp1: 0x639cc441b010 : Type const char * (pointer to string), 8 bytes on 64-bit systems.
 &gccp2: 0x639cc441b018 : const char * type (pointer to string), 8 bytes.
 gccp1 (pointed string): 0x639cc4419010 : Constant string in RODATA, approximate size: ~445 bytes (the length of the string 'Lorem ipsum dolor sit amet...').
 gccp2 (pointed string): 0x639cc44191d0 : RODATA constant string, approximate size: ~479 bytes (the length of the string 'Viral non adipisicing messenger bag...').
 main 0x639cc44181a9 - FILE TEXT, which in total occupies 3663 bytes, as we'll see below

Group 2:

lp 0x639cc632f2a0 : A char * pointer to a dynamically allocated block of memory, 8 bytes (64 bits).
 malloc(1024) allocates 1024 bytes on the Heap to store the data pointed to by lp.

Group 3:

&li1: 0x7fff8ed254b4: Int type, 4 bytes.
 &lc1: 0x7fff8ed254b8: Int type, 4 bytes.
 &lci1: 0x7fff8ed254bc: Int const type, 4 bytes.
 &lp: 0x7fff8ed254c0: Pointer of type char *, 8 bytes.
 &argc: 0x7fff8ed254ac :Int type, 4 bytes.
 argv: 0x7fff8ed255f8: Pointer to an array of pointers of type char *, 8 bytes.
 argv[0]: 0x7fff8ed27176 :First element of argv, a char * pointer pointing to a text string (8 bytes).

Group 4:

stderr 0x7a914aa044e0 : FILE type *, 8 byte

Once the groups are defined, we will analyse what kind of variables are stored together and why. In **Group 1**, we find both **uninitialized global variables** (gi1 and gc1) and **global constants** (gci1, gccp1, and gccp2).

The uninitialized variables (gi1 and gc1) are placed in the **BSS** (Block Started by Symbol) segment, reserved for this type of data. The location in BSS allows the system to automatically initialise these variables to zero, optimising the use of memory.

On the other hand, the constants (gci1, gccp1, and gccp2) are divided into two types:

- gci1 is an integer constant.
- gccp1 and gccp2 are constant pointers pointing to text strings.

These constants are stored in the **RODATA** (read-only data segment), which is intended for read-only data. The gccp1 and gccp2 pointers contain the addresses of the strings they point to, which are also stored in the read-only section as string literals.

In addition, we have the main function, which is located in **the code segment (TEXT)**, where the executable code of the program is stored.

This aggrupation in **Group 1** allows to optimise the use of resources, since the data are organised in specific segments according to their type. This arrangement facilitates the application of protection properties, such as read-only for RODATA, allows segment sharing between processes, and helps to improve the performance and security of the operating system.

Let's move on to **Group 2**, which is formed by a single variable lp, which is in charge of storing the address of a dynamically allocated memory block (with malloc), located in the heap.

In **Group 3**, you can see a change in the memory location. These are local variables within the main, they are located on the stack. This area is often used for params and temporary data. In unix systems it is usual to position the memory address at 0x7fff...As can be seen, Stack follows a lifo(Last in, First out) form, with the variable argv[0] 0x7fff8ed27176 being at a higher position and consecutively going down to &li1 0x7fff8ed254b4 which is where the lowest position is.

Finally, the **Group 4**, this group is located outside the process where the addresses we have seen above are located as this is managed by the operating system itself because it is an area reserved to help **the communication of the program with the operating system environment**, which allows the system to redirect or control error output efficiently. This is an internal program function that is automatically created and initialised when the program starts.

To see more information, what I have done is to see the size of the mem2 file. With this command size mem2 it gives us this output.

```
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/priso$ size mem2
text    data    bss     dec     hex filename
3663    640     24    4327    10e7 mem2
```

We can see that the executable file mem2 occupies 4327 bytes in memory, with 3663 bytes for the code (text), 640 bytes for initialized data and constants (data), and 24 bytes for uninitialized variables (bss).

The size of BSS is (24 bytes), it can be seen that the sum of gi1 and gc1 is done, being a 64-bit architecture, the compiler aligns the variables in multiples of 8 bytes to optimize the access in memory, which can result in a larger BSS than expected.

2.3 Given a string, its i -th suffix ($0 < i < \text{string length}$) is the string formed by the characters $i, i + 1, \dots, \text{string length} - 1$ of the string. For example, the suffixes of the string "paleta" are: paleta, aleta, leta, eta, ta, a.

Complete the code of `suffix.c` so that it fills the suffix array with all the suffixes of the string received via the argument vector (`argv`). Once filled, the code will display the content of the array on the screen. An example of the desired result is attached:

Observations:

- In the program, you must use dynamic memory management routines to allocate memory both for the pointer array and to copy each suffix.
- The solution described for this problem is not the most efficient one, but it is the solution we are asking you to implement.
- Your code must be placed between the lines in `suffix.c` marked as `/* Your code starts here */` and `/* Your code ends here */`.
- You are not allowed to modify the rest of the provided program.
- You must submit the source code of the program and, if it works, a screenshot showing it.

Program:

```
/* Your code starts here */

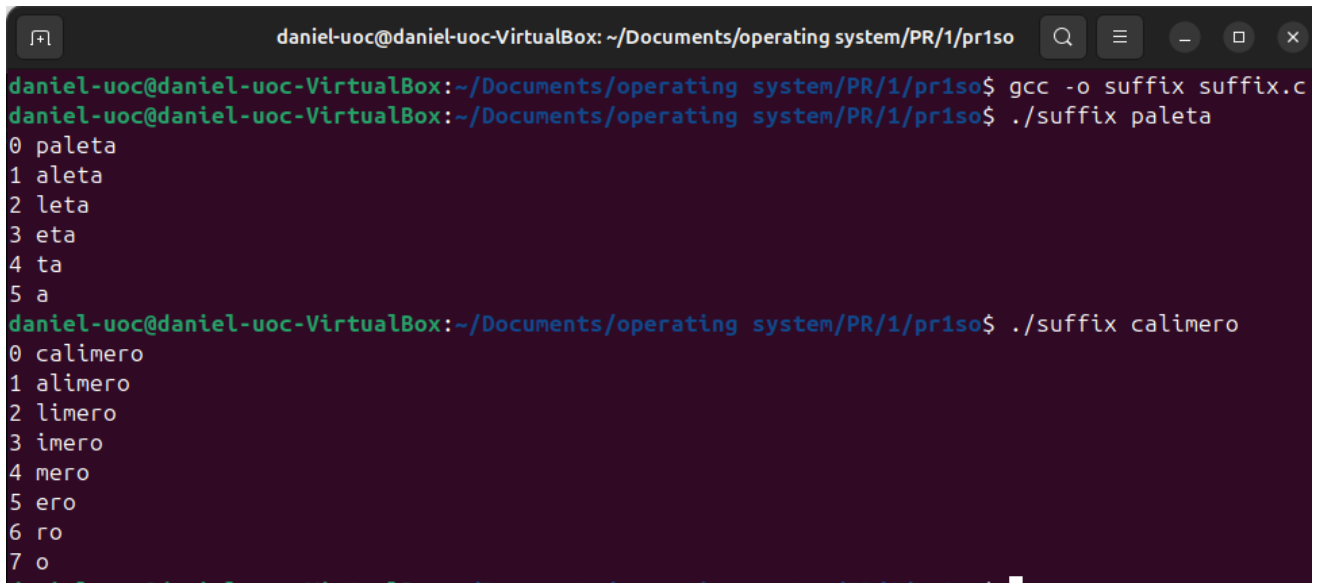
char *input_string = argv[1];
n = strlen(input_string);

// Allocate memory for the pointer array
suffix = malloc(n * sizeof(char *));
if (suffix == NULL) {
    panic("Memory allocation failed for suffix array");
}

// Create and allocate memory for each suffix
for (i = 0; i < n; i++) {
    suffix[i] = malloc((n - i + 1) * sizeof(char)); // +1 for null
    terminator
    if (suffix[i] == NULL) {
        panic("Memory allocation failed for suffix string");
    }
    strcpy(suffix[i], input_string + i); // Copy the suffix from location i
}

/* Your code ends here */
```

Terminal:



```
daniel-uoc@daniel-uoc-VirtualBox: ~/Documents/operating system/PR/1/pr1so
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/pr1so$ gcc -o suffix suffix.c
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/pr1so$ ./suffix paleta
0 paleta
1 aleta
2 leta
3 eta
4 ta
5 a
daniel-uoc@daniel-uoc-VirtualBox:~/Documents/operating system/PR/1/pr1so$ ./suffix calimero
0 calimero
1 alimero
2 limero
3 imero
4 mero
5 ero
6 ro
7 o
```

The modified program is in the attached folder in the suffix_modified.c file.

Module 4: In/Out

I had problems with the command `ps u > filename` displayed by `tty`, as I was not using a BSD syntax, I had to change the filename to `displayed_tty`. In the following image you can see the execution of the command and the error.

```
daniel-uoc@daniel-uoc-VirtualBox:~/Desktop$ ps u > filename displayed by tty
error: unsupported option (BSD syntax)

Usage:
ps [options]

Try 'ps --help <simple|list|output|threads|misc|all>'
or 'ps --help <s|l|o|t|m|a>'
for additional help text.

For more details see ps(1).
```

As it says in the statement, two different terminals must be created. In the image on the next page you can see it. Where the `tty` command is shown on one side, and the rest of the commands on the other. Also at the bottom of the image you can see the contents of the Desktop folder, once all the commands are done and the `xxx` file showing its contents, it strikes me that when I run the command `ps > u`, it shows less content in the terminal than there is in the `xxx` file.

Ubuntu [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Nov 9 08:24

daniel-uoc@daniel-uoc-VirtualBox: ~/Desktop

```
daniel-uoc@daniel-uoc-VirtualBox:~/Desktop$ tty
/dev/pts/0
daniel-uoc@daniel-uoc-VirtualBox:~/Desktop$
```

daniel-uoc@daniel-uoc-VirtualBox: ~/Desktop

```
daniel-uoc@daniel-uoc-VirtualBox:~/Desktop$ ps u
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
daniel-+      3255  0.0  0.1 244336 6016 tty2     Ssl+  07:57   0:00 /usr/libexec/
daniel-+      3263  0.0  0.4 306876 16512 tty2     Sl+   07:57   0:00 /usr/libexec/
daniel-+      4810  0.0  0.1 19692  4992 pts/0     Ss+   08:12   0:00 bash
daniel-+      4824  0.2  0.1 19692  4992 pts/1     Ss    08:12   0:00 bash
daniel-+      4832  250  0.1 22284  4608 pts/1     R+    08:13   0:00 ps u

daniel-uoc@daniel-uoc-VirtualBox:~/Desktop$ ps u > xxx
daniel-uoc@daniel-uoc-VirtualBox:~/Desktop$ ps u > displayed_tty
daniel-uoc@daniel-uoc-VirtualBox:~/Desktop$ ps u > /dev/null
daniel-uoc@daniel-uoc-VirtualBox:~/Desktop$
```

Files

Home / Desktop

Recent

Starred

Home

Documents

Downloads

Music

Pictures

Videos

Trash

displayed_tty

xxx

Open

xxx

```
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
daniel-+      3255  0.0  0.1 244336 6016 tty2     Ssl+  07:57   0:00 /usr/libexec/gdm-wayland-session env
GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu
daniel-+      3263  0.0  0.4 306876 16512 tty2     Sl+   07:57   0:00 /usr/libexec/gnome-session-binary --
session=ubuntu
daniel-+      4810  0.0  0.1 19692  4992 pts/0     Ss+   08:12   0:00 bash
daniel-+      4824  0.1  0.1 19692  4992 pts/1     Ss    08:12   0:00 bash
daniel-+      4833  0.0  0.1 22284  4608 pts/1     R+    08:13   0:00 ps u
```

CTRL DERECHA

3.1 What does each of the four commands do?

- **Ps u** : This command displays a list of the processes running for the current user in a detailed format, including information such as the process ID (PID), the percentage of CPU usage (%CPU), the percentage of memory usage (%MEM), the size of virtual memory in kilobytes (VSZ), the amount of physical memory in kilobytes which is loaded into RAM (RSS), the terminal flag which the process is associated with, teletype (TTY), the current status of the process in characters (STAT), explanation below. It also shows the time the process was started (START), the total time the CPU has been consumed in minutes:seconds (TIME) and finally it shows the complete command to start the process.
- **Ps u > xxx**: Generates a file and redirects the output of the ps u command to the file ensi which in this case is called xxx..
- **Ps u > displayed_tty** (filename displayed by tty)
- **Ps u > /dev/null**: this is a special file that discards all content redirected to it. More detailed explanation below.

STAT works with the character representation indicating the current status of the process, these are the meaning of each character that can be displayed on the screen.

- **R**: Running, the process is active.
- **S**: Sleeping, the process is waiting for a resource.
- **D**: Uninterruptible sleep, waiting for hardware access.
- **T**: Stopped, the process has been stopped by a signal or debugging.
- **X**: Dead, not often seen.
- **Z**: Zombie, the process has terminated but has not yet been cleaned up by its parent process.

Additional characters:

- **+**: Foreground process.
- **<**: High priority.
- **N**: Low priority.
- **L**: has locked pages in memory (for real-time and custom IO).
- **s**: the process is a session leader.
- **l**: is multi-threaded.

On the other hand, `dev/null/` is a special file that acts as a 'data sink': anything sent there is automatically discarded. This is useful when you want to run a command but don't need to see its output.

3.2 How does allowing the `ps` command to be executed in these four ways affect the command's programmer?

Throughout this work, I have seen the importance and versatility of the `ps` command. Honestly, I have been amazed at what can be achieved with a single command. The ability to redirect process information between files or different outputs is really useful and impressive.

For example, the `ps u > xxx` command allows you to easily save all the information about the processes the user is using in a file. This is especially valuable from a programming and system administration perspective, as it makes it easy to record the complete state of the machine at a specific point in time. This can help to avoid errors, or to understand where you were working in case you need to check it later.

On the other hand, the possibility to execute a command without displaying the output, by redirecting it to `/dev/null`, seems very convenient to me. This is useful when commands generate very long outputs or when we just need to run the process without seeing the result on screen, keeping the terminal cleaner and the system tidier.

Bibliography

- **DigitalOcean.** (2023). Understanding /dev/null: How to Use this Special File in Linux. Recuperado de <https://www.digitalocean.com/community/tutorials/dev-null-in-linux>
- **Die.net.** (2023). PS(1) - Linux manual page. Recuperado de <https://linux.die.net/man/1/ps>
- **FPGenRed.** (s.f.). Guía de comandos en Linux: Comando uptime. Recuperado de <https://www.fpgenred.es/GuiaComandosLinux/uptime.html>
- **GNU Project.** (2023). Using GDB: The GNU Debugger. Recuperado de <https://www.gnu.org/software/gdb/>
- **GNOME.** (s.f.). GNOME Technologies Overview. Recuperado de <https://www.gnome.org/technologies/>
- **Aprender a Programar.** (s. f.). *Final de archivo EOF y feof() con C. End of File. Leer datos de un fichero hasta el final. Ejemplo código CU00543F.* Recuperado de https://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=942&catid=82&Itemid=210.
- **Linux Foundation.** (2023). Linux Memory Management. En Linux Kernel Documentation. Recuperado de <https://www.kernel.org/doc/html/latest/mm/index.html>
- **Linux Programmer's Manual.** (2023). Scanf and Fgets: Input and Buffer Management. En man pages. Recuperado de <https://man7.org/linux/man-pages/man3/scanf.3.html>
- **LPI (Linux Professional Institute).** (2022). Introduction to System Monitoring in Linux: Using ps and top commands. En LPIC-1 Exam Guide. Recuperado de <https://learning.lpi.org/en/learning-materials/>
- **Softzone.** (2023). Ver procesos en ejecución en Linux y sus consumos de CPU y RAM. Recuperado de <https://www.softzone.es/linux/tutoriales/ver-procesos-ram-cpu-linux/>
- **Tutorials Point.** (2023). C - Memory Management. En C Programming Tutorial. Recuperado de https://www.tutorialspoint.com/cprogramming/c_memory_management.htm
- **Ubuntu Manpages.** (s. f.). *PS(1) - Manual de usuario de Ubuntu en español.* Recuperado de <https://manpages.ubuntu.com/manpages/trusty/es/man1/ps.1.html>.

- **Silberschatz, A., Galvin, P. B., & Gagne, G.** (2008). Operating System Concepts (8a ed.). John Wiley & Sons.
- **Tanenbaum, A. S.** (2009). Modern Operating Systems. Prentice-Hall.