SESSIÓ 2: LINUX ENVIRONMENT (RAQUEL)

Obtenir el contingut de la variable d'entorn HOME
echo $HOME

To change from 'tcsh' mode of the Shell to 'bash':
/bin/bash

All the shells have two types of commands:

- external commands: any program installed in the machine
- internal commands: functions implemented by the command-line.

GETTING HELP		
1 'man'	**2. 'info',**	**3.bash 'help'**

All the options of the man with 'man man'.
Basics of man command:git c

1. Press space to reed in more than one screen
2. letter 'b' : to go to a previous screen
3. '/word interested': to search a word in the man pages. To go to the other occurrence, press n
4. 'q' to exit.

Open directories	Close directories
'cd'	'cd ..'

Creating folders	Delete folders	Open the editor
'mkdir name_folder'	'rmdir name_foler'	'gedit test &'

To show all the content of the directory 'ls'
To show information about the test we have created in the terminal 'more name file', 'cat name file' or 'less name file'.

Copy paste	Delete file	Move to another directory
'cp file_to_copy name_of_new file'	'rm name_file'	'mv name_file new directory'

IMPORTANT! Si volem copiar un fitxer en una carpeta different en la que estem hem d'escriure aquesta comanda.

cp input.dat /home2/users/alumnes/1289188/dades/COM-Labs/provaLab

En aquest cas teníem l'arxiu 'input.dat' a la carpeta de descàrregues i des d'allà l'hem copiat a la carpeta de provaLab.

PERMISSIONS: 'ls - la'

1. Owner of the file (u)
2. Users in the same group (g)
3. Rest (o)

OPERATIONS: read(r), write(w) and execution (x)

Interpretation:
-rw-r--r-

1. Firsts caracter: 'd' -> directory or '-' means data file
2. Three characters for the owner (rwx)
3. Three character for the member of the owner's group
4. Three for the rest

CHMOD COMMAND

Activation of the execution of f1 for all users	Remove permissions
‘chmod ugo+x f1 ’	#chmod o-x f1’

Change of permissions
“#chmod ug=rwx f1’

SPECIAL CHARACTER FOR THE SHELL

- ***** : The Shell substitutes the ‘*’ for any group of characters, matching a file name in the directory where the shell is executing
- **>**: Redirect the output to another file. For example, “ls > output_ls”, stores the output of the ls in the file output_ls
- **>>** : Redirect the output to another file but it does not remove what it was in the file previously, it adds it to the end.
- **|**: known as pipe: communicate the output of a given command execution (left side) to the input of another command execution (right side).
- **^**: to specify that the following character is at the very beginning of a line:

‘grep’ -> allows the search of a text in one or more files
For example, in one of the files: test1 test2 test3 test4 there is the word ‘hello’.

INPUT→ grep hello test1 test2 test3 test4
OUTPUT→ test:hello

Filter the list of the folder contents to show only the ones starting with d
ls -l | grep '^d'

To stop the execution of a program (w/o Ctrl + C)
kill -9

FILESYSTEMS:

The **mount** command lists the set of filesystems available in your environment

- /proc -> to obtain info about the system and to change certain kernel parameters
- /dev -> the location of special or device files
- /sys -> is the filesystem containing system information
- //pax/dades -> is mounted over the network

CHANGE NAME OF A COMMAND: ‘alias ls = ‘ls -la’

To see all the variables defined in the current environment and their value ‘env’

See the value of a specific environment variable
‘echo $USERNAME’

modify an environment variable
‘export VARIABLE_NAME=value’

“file” command, that informs the user about the estimated content of the file provided:-> file *.c
observe the exact content of files in octal format: **‘od -t cd1x1 program.c** ‘ or using **“xxd** to show the content in hexadecimal format. ‘xxd -p file.txt’

SESSIÓ 3: C & GIT (MARTINA)

GIT ENVIRONMENT

- ➢ Initial commands

Creating a git repository	**Adding a file to the repository**
$ git init	$ git **add** hello.c ‘hello.c’ is any file

- ➢ User identification commands (needed to document the development)

1st line: Name **|** **2nd line:** Email
$ git config --global user.name “MyName” $ git config --global user.email “UPC e-mail address”

- ➢ Registering modifications

Commits (small changes)
$ git commit -m “This is the first file”

Re-uploading a file after changing something in it
$ git add hello.c $ git commit -m “This is a modification”

Checking repository modifications
$ git log --abbrev-commit
Here, the message we added to the commit (ex: “This is the first file”) will let us identify the exact upload
$ git reflog   # for complementary data (alphanumeric identification)

- ➢ Checking other versions of a file

Going to a concrete version
$ git checkout XXX   # ‘XXX’ comes from the output of ‘$ git reflog’
If you check the file now, it will have the content of version ‘XXX’
$ git checkout YYY   # ‘YYY’ identifies a new version you want to check

- ➢ Managing repository branches

Showing the branches of the current repository
$ git branch

Creating a new branch
$ git checkout -b ZZZ   # ‘ZZZ’ is the name of the new branch
This command automatically **switches location to the new branch**

Switching branches
$ git switch ZZZ
This command **doesn’t create ‘ZZZ’**, it just switches location if it already exists

Merging branches
$ git merge AAA BBB   # ‘BBB’ is the current branch, the content of ‘AAA’ is added to ‘BBB’
This command **doesn’t create ‘ZZZ’**, it just switches location if it already exists

Deleting branches
```
$ git branch -d name      # 'name' is the name of the branch we want to delete
```

Showing the graph of current branches
```
$ git --graph --branches log
```

C ENVIRONMENT

➢ Initial concepts

Terminal related variables

- ★ argc: Counts how many arguments we write in the command line
- ★ argv[i]: Refers to a concrete argument

example: ./fibonacci 10 --> **argc** = 2, always >= 1 ('./fibonacci' and '10')
--> **argv[0]** = ./fibonacci, **argv[1]** = 10

Code navigation

- ★ printf: To print a value, string…
  - \n: Indicates a line return
  - %: Followed by a letter, represents the type of another variable
  - argv[i]: We can also print something we have put in the terminal
- ★ ;: (punt i coma) Has to go at the end of every line!
- ★ int main(){}: The main function is always initialized with **int**
  - return 0: We have to put it at the end so it doesn't return an error

example:
```
int main(int argc, char **argv) {
    printf("Hello! %d \n", 10);   // %d: The value after ',' is an 'int'
    return 0;
    }
```

➢ Compilation commands

Simple compilation (without creating an executable)
```
$ gcc -c hello.c              # 'hello.c' is the file we want to compile
```

Simple compilation (creating an executable)
```
$ gcc -o hello.exe hello.o    # 'hello.exe' is the executable ('.exe' is not
                                needed)
```

IMPORTANT!: 'hello.o' is not compiled yet, to compile it completely we need to execute the command line in the previous box

Executing the compiled program
```
$ ./hello.exe
```

➢ Makefile

Creating the Makefile from the terminal
```
$ gedit Makefile &            # opens a blank Makefile document to fill
```

Filling out the Makefile (example)
```
all: hello                    # includes all the programs to be compiled here

hello: hello.o
       gcc -o hello hello.o   # creates 'hello.o' from the file 'hello'
hello.o: hello.c
```

```
        gcc -c hello.c

clean:
        rm -f hello hello.o       # to remove any additional generated files
```

SESSIÓ 4: LIBRARIES AND COMPILATION (RAQUEL)

COMPILER COMMAND LINE OPTIONS

Getting help:
‘ **--help**’ # simple help   ‘**--help --verbose**’ # help of the compiler driver and sub-processes

Common options
‘-S’   #generate assembly only
‘c’   #generates objects files only
-o <name>   #name the output file

Code generation options
‘-fpic’   #generate position independent code, used in shared libraries (small mode)
‘fPIC’   #generate positions independent code (large mode)

Optimization options
- ➔ -O equivalent to -O1
- ➔ -O0 no optimizations
- ➔ -O1 basic optimizations that do not take compilation time
- ➔ -O2 more expensive optimizations
- ➔ -O3: all optimizations

Debug support options
‘-g’   #generates debug info

Linking options
-L<path>   # adds path to the list of directories where to find libraries for linking
-l<name>   # adds lib<name>.so for shared linking, and/or lib<name>.a for static linking
-shared   # generates a shared library, instead of a binary executable
-static   # generates a statically linked binary executable

Present the symbol table of a compiled files
‘nm’
The symbol tables of the executables contain more symbols compared to the object table because they include symbols of the libraries used in the program.

To import a function from another file:
For example, if we have a function in a file called fib.h, we will import it in the main file called fibonacci as a header:
#include “fib.h”

The makefile will be:

```
all: fibonacci fibonacci fibonacci.o assembly

fibonacci: fibonacci.c fib.c
	gcc -o fibonacci fibonacci.c fib.c

fibonacci.o: fibonacci.c
	gcc -o fibonacci.c
assembly:
	gcc -S fibonacci.c fib.c

fib.o: fib.c
	gcc -o fib.c

clean:
	rm fibonacci
```

STATIC LIBRARY:

Include an static library
‘>ar -csr libCOMstatic.a fib.o’
‘gcc -o fib-liba.exe fibonacci.c -L -lCOMstatic’
‘gcc -o fib-liba.exe fibonacci.c libCOMstatic.a’
gcc -o libCOMstatic.so -static fib.o

Per saber si un fitxer està enllaçat dinàmicament o estaticament:
file filename → *Inclourà informació sobre si s’ha compilat estàticament o dinàmicament*
OPCIO 2: nm -D file

- Si surten llibreries → dinàmicament
- Si surten symbols → staticament

This command line will create the static library library libCOMstatic.a. The flags indicate: (c) create the library; (s) create an index of the files inside the library; and (r) to add files to the library.
The ‘ar ‘ command can be used to extract object files from a given library file. Options with ‘-x’, ‘-p’.

SHARED LIBRARY:Using the extension .so

1. We need to add the **‘-fpic’** flag to the object program.
2. Execute **‘gcc -o libCOMdyn.so -shared fib.o’**
3. Compile the new version using the newly created library. Introduce the flag **‘-L’ and ‘lCOMdyn’.**

STEPS
gcc -c -fpic fib.c -o fib.o
gcc -shared -o libCOMdyn.so fib.o
gcc -o fib-libso.exe -L -lCOMdyn fibonacci.c

Shared objects dependencies
‘ldd’ flags of ‘ldd’:
‘-v’ #info about the libraries dependencies
‘-d’ #process data relocation
‘-r’ #process data and function relocations

PROBLEMS WE CAN FIND: If we try to execute with a library and it is not found we need to check the environment variables of ‘LD_LIBRARY_PATH’. ‘**echo $LD_LIBRARY_PATH’** We need to add ‘.’ path -> export LD_LIBRARY_PATH = $LD_LIBRARY_PATH.

SESSIÓ 5: DEBUGGING (MARIA)

DEBUGGING PYTHON PROGRAMS

Execute python file with the Python debugger (pdb) - Exemple amb Fibonacci de 4
python3 -m pdb ./fib.py 4

The fibonacci program is automatically started, and the debugger shows a little bit of context (file and line number, and next line to execute), and waits for commands:
> /home/alumne/Documents/COM_GCED/Lab/S5/fib.py(2)()
-> import sys
(Pdb)

next
Continue execution until the next line in the current function is reached or it returns.
The difference between next and step is that **step stops inside a called function**, while next executes called functions only stopping at the next line in the current function.

step
Execute the current line, stop at the first possible occasion

list
List source code for the current file. It also indicated with ‘>>’ if an exception is being debugged (the exception is responsible for stopping the execution)

ll (longlist)
list all source code for the current function or frame

where
used to see in which recursion level we are in

up
Move the current frame one level up

down
Move the current frame one level down

display<name_variable>
Is a debugger command that keeps a list of symbols to be displayed when they change their value.

print <name_variable>
Is used to print the value of a variable or expression at a specific point in the code

exit
To exit the pdb

The python debugger manages breakpoints automatically. Additionally you can **add breakpoints with the 'break' command**

DEBUGGING C PROGRAMS

REMEMBER TO FIRST COMPILE THE FILE BEFORE STARTING DEBUGGING!
If there are several files involved, do a Makefile:

```
all: fib

fib: fib.c fibonacci.c
	gcc -o fib -g fib.c fibonacci.c
clean:
	rm fib
```

Start the debugging in C - Example with file called 'fib'
gdb ./fib

gdb run
To run the program from inside de debugger area

DIFFERENCES BETWEEN DEBUGGING IN C AND IN PYTHON
Unlike the Python case, in C the program:

- Is not automatically started until the user issues the 'run' command with its arguments
- Is run till the end, as no breakpoints are set implicitly by the debugger (in python breakpoints are automatic)
- Is stopped only in case that a problem occurs

list main (and then 'list')

Will show the source code around the main() function in the current executable file being debugged. This is useful for examining the code around where the program started running and **for setting breakpoints using the line where you want to set the breakpoint**

list fibonacci

This command will show the source code of the function called 'fibonacci' in the file being debugged. The same applies to any name of a function you have in a program.

WAYS OF SETTING BREAKPOINTS

break <line - number>

Set a breakpoint in a C source line in the current file

break <filename>:<line-number>

Set a breakpoint in a C source line on a specific file

break <function-name>

<function-name> refers to the name of the function in your program, where you want the execution to stop. **Used to set a breakpoint at a function entry-point (és a dir, a l'inici de la funció).**

break <number -line>

To set the breakpoint we have to first call 'list main' and then 'list'. Once we have identified the line in which we want to set the breakpoint we can use the ***break <number-line>***

delete <number-of-breakpoint>

Used to delete a breakpoint. We will use the **'info break'** command to know the number that corresponds to the breakpoint we want to delete.

info break

To see the breakpoints we have. For example:

```
2       breakpoint     keep y   0x00000000004006c5 in fibonacci at fibonacci.c:5
5       breakpoint     keep y   0x0000000000400666 in main at fib.c:18
```

Additional GBD commands

continue or **cont** or **c**

to continue with the execution till the next breakpoint, or **the end of the program in case there are no more breakpoints**

next

to execute the next program sentence **without stopping inside function calls** (és a dir quan vegi que en una línia donada s'està cridant a una funció, no hi entrarà a dins sinó que seguirà llegint el codi sense entrar dins la funció)

step

to execute to the next program sentence, **stopping inside function calls** (en aquest cas sí que entrarà dins el codi de la funció i la recorrerà)

finish
Aquesta comanda pot ser útil quan vols arribar al final d'una funció mentre depures el codi d'aquesta funció.
NOTA: si s'ha cridat la comanda 'finish' mentre es troba en el programa principal (és a dir, si no estàs dins una funció), aquesta no tindrà cap efecte. Això es deu al fet que la comanda està dissenyada per executar el programa fins que hi ha un ‘return’.
La comanda ‘finish’ retorna el valor de la funció que s’ha acabat d’executar

set argc=<value you want>
To change the value of the ‘argc’ variable. It is necessary to set a breakpoint in the line where the ‘argc’ variable is and then use the command ‘**set argc=1**’. In this case we change the value to 1.

Comparing the cost of executing compiled versus interpreted code

COMPILE USING DIFFERENT OPTIMIZATIONS WITH THE MAKEFILE

```
all: fib

fib: fib.c fibonacci.c
	gcc fib.c fibonacci.c -o fib0 -O0
	gcc fib.c fibonacci.c -o fib1 -O1
	gcc fib.c fibonacci.c -o fib2 -O2
	gcc fib.c fibonacci.c -o fib3 -O3

clean:
	rm fib0 fib1 fib2 fib3
```

CALCULATE THE EXECUTION TIME
/usr/bin/time myprogram

MEASURE THE NUMBER OF INSTRUCTIONS EXECUTED
valgrind -tool=callgrind myprogram

If we want to see the value of a variable we will call: display numse

---

SESSIÓ 6: DATA REPRESENTATION (MARTINA)

- File management through the terminal

Redirecting the output to a file

```
$ ./writeint 10 > out.dat        # ‘./writeint 10’ executes the program
                                 # ‘out.dat’ is the file we redirect the output to

$ ./writeint 10 >> out.dat       # adds more content, without erasing the previous
```

Checking hexadecimal (hxd) format

```
$ xxd out.dat                    # Shows the contents of out.dat in hxd format
```

Redirecting the input to a file

```
$ ./readchar < out.dat           # ‘out.dat’ is the file we want to read from
```

Here, we don’t need to put a number in the terminal, because the data is extracted directly from the file ‘out.dat’.

Representing UTF-8 symbols

```
$ echo -e '\U1f600'          # '\U1f600' is the UTF-8 reference for the symbol
```

SCALAR DATA TYPES

- ➢ Printing integers without converting them from strings

Writing an integer value

- ★ write(1, &num, sizeof(int));
  - 1: Default value for the function 'write'
  - &num: Content stored in the variable (of type *int*) we want to write
  - sizeof(int): Amount of Bytes that will be written (in this case, 4)

Casts --> Changing variable types in the code

example: Expressing the division of two integers

- ★ Case 1: We want the result as an *int*
  - result = num1/num2;   # C defaults **integer rounding** for int division
- ★ Case 2: We want the result as a *float* --> result = (float) num1/num2 **[NO!]**
  - n1 = (float) num1;
  - n2 = (float) num2;
  - result = n1/n2;   # It's a float division, **no integer rounding**

COMPOUND DATA TYPES

- ➢ Structures vs Unions

Structure initialization (example)

- ★ **struct** person_data { char name[256]; int age; };
  - char name[256] | int age: Variables of different types
  - };: We always have to put it at the end of a struct
- ★ **union** person_data { char name[256]; int age; }; --> Only the name changes

Assignation of structs/unions

- ★ **struct** person_data **person**;   # '**person**' is of type 'struct person_data'
- ★ **union** person_data **person**;   # '**person**' is of type 'union person_data'

Assignation of attributes

- ★ **strcpy(**person.name, argv[i]**);**   # the attribute 'name' is equal to argv[i]

We use 'strcpy', to convert the value of argv[i] to a variable of type *string*

- ★ person.age = **atoi(**argv[i+1]**);**   # the attribute 'name' is equal to argv[i]

We use 'atoi', to convert the value of argv[i+1] to a variable of type *char/int*

SESSIÓ 7: COMPUTER ELEMENTS (MARTINA)

- ➢ The processor and components layout

To get information about the CPU

```
$ cat /proc/cpuinfo       # we can use 'cat'/'more' because it is a text file
```

```
$ lscpu                    # shows information about the CPU arrangement

$ lstopo                   # shows the output of 'lscpu' in a graphical display
```

VERY IMPORTANT!: *launch.sh* is a file we will have to create if not given to us!!

```
#!/bin/bash --> Write this command in the terminal before executing
argc=$#
argv=($@)

for((i=0; i<argc; i=i+2)); do
      echo running program ${argv[i+1]} at CPUid ${argv[i]}
      /usr/bin/time taskset -c ${argv[i]} ${argv[i+1]}&
done
```

Program distribution in the CPU

```
$ ./launch.sh 0 ./integers 1 ./floats        # ./integers is executed in HW 0
                                             # ./floats is executed in HW 1
```

IMPORTANT!: Both programs are executed at the same time. To see graphically how the program enters and leaves a HW, we need to have a **second terminal open**, and there execute the command --> **htop**

```
$ ./launch.sh 0 ./integers      # to run a single instance of ./integers in HW 0
```

The output will be the elapsed time (how long it takes to execute completely) and the % of CPU taken.

Checking execution specifics

```
$ valgrind --tool=cachegrind ./PROGRAM        # 'PROGRAM' is the name of the file
```

The output is a series of categories, introduced by different cap letters.

- ★ D1: Displays the amount of misses in the L1 level of cache
- ★ I: Displays the amount of instructions executed
- ★ Other parameters: LLD references, D...

SESSIÓ 8: PROCESS MANAGEMENT (MARIA)

- ➢ Processes Management Commands

ps (process status)

Shows summarized information about a selected group of processes that exist in the system. By default 'ps' shows information of processes launched by the current user in the current terminal.

YOU CAN ACCESS TO THE 'PID' USING THIS COMMAND

FLAGS

- **ps -a** : Displays information about all processes with terminals
- **ps -u** : Displays the following information → USER, PID, %CPU, %MEM, SZ, RSS, TTY, STAT, STIME, TIME and COMMAND fields

- **ps -u <username> :** Displays information about the processes owned by the specified user,
- **ps -e :** Displays the environment as well as the parameters to the command regardless of the user or terminal associated with them.
- **ps -f :** It lists processes in a detailed format, providing more information about each process compared to the default output
- **ps -fL :** It will list processes along with information about their threads.

ps -o

Allows a full configuration of the values displayed. The values that we can use are:

```
PID USER    PR    NI    S    %CPU    %MEM    TIME+    COMMAND
```

For example, if we put: > ps -o pid -o user -o pri -o ni we see:

```
PID USER       PRI  NI
4165 maria.s+  19   0
5001 maria.s+  19   0
5501 maria.s+  19   0
```

pstree

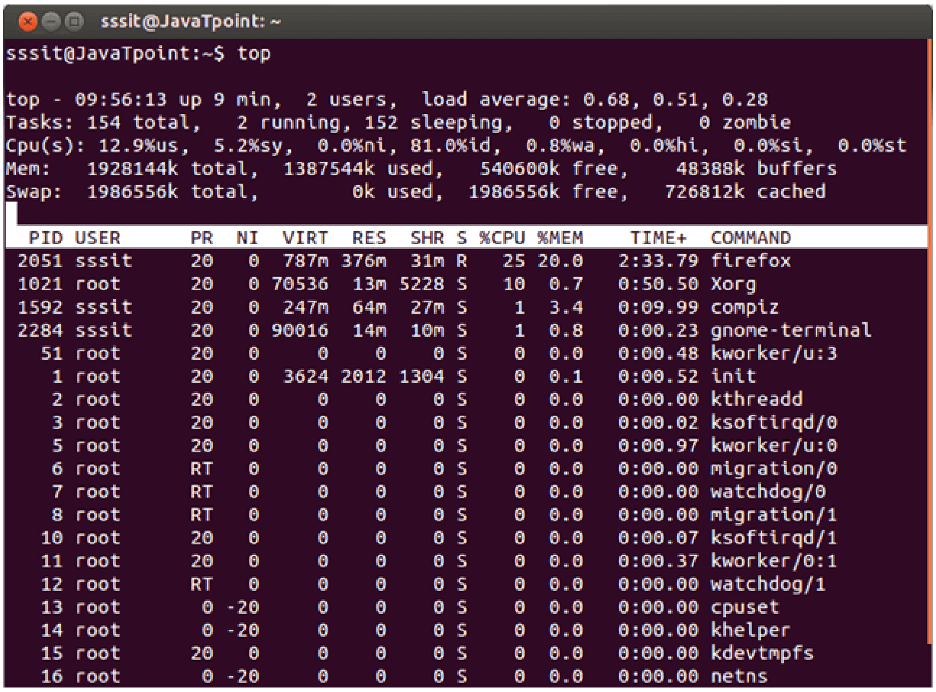
This command shows the process hierarchy of all processes in the system and how they are related

htop

Is similar to ‘top’ but more detailed information, and an interactive interface. This command updates the information every 3 seconds, by default. You can configure the delay using the flag ‘-d <time>’ where time is the tenth of seconds to update the output. You can quit pressing ‘q’.

top

Displays real-time information about processes, system resource usage, and overall system performance



```
sssit@JavaTpoint: ~
sssit@JavaTpoint:~$ top

top - 09:56:13 up 9 min,  2 users,  load average: 0.68, 0.51, 0.28
Tasks: 154 total,   2 running, 152 sleeping,   0 stopped,   0 zombie
Cpu(s): 12.9%us,  5.2%sy,  0.0%ni, 81.0%id,  0.8%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1928144k total,  1387544k used,   540600k free,    48388k buffers
Swap:  1986556k total,        0k used,  1986556k free,   726812k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2051 sssit     20   0  787m 376m  31m R   25 20.0   2:33.79 firefox
 1021 root      20   0 70536  13m 5228 S   10  0.7   0:50.50 Xorg
 1592 sssit     20   0  247m  64m  27m S    1  3.4   0:09.99 compiz
 2284 sssit     20   0 90016  14m  10m S    1  0.8   0:00.23 gnome-terminal
   51 root      20   0     0    0    0 S    0  0.0   0:00.48 kworker/u:3
    1 root      20   0  3624 2012 1304 S    0  0.1   0:00.52 init
    2 root      20   0     0    0    0 S    0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S    0  0.0   0:00.02 ksoftirqd/0
    5 root      20   0     0    0    0 S    0  0.0   0:00.97 kworker/u:0
    6 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/0
    7 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/0
    8 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/1
   10 root      20   0     0    0    0 S    0  0.0   0:00.07 ksoftirqd/1
   11 root      20   0     0    0    0 S    0  0.0   0:00.37 kworker/0:1
   12 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/1
   13 root       0 -20     0    0    0 S    0  0.0   0:00.00 cpuset
   14 root       0 -20     0    0    0 S    0  0.0   0:00.00 khelper
   15 root      20   0     0    0    0 S    0  0.0   0:00.00 kdevtmpfs
   16 root       0 -20     0    0    0 S    0  0.0   0:00.00 netns
```

- **PR**: Priority of the process. It represents the priority level at which the process is scheduled to run.. **It is an integer value (ranged from 0 to 39, which default value of 20 for standard processes). Higher integer values mean lower priority. Lower decimal values mean higher priority. “rt” mean the highest priority (i.e ‘real-time’).**
    - **PROCESS WITH HIGHER PRIORITY USE MORE CPU THAN THE PROCESS WITH LOWER PRIORITY.**
- **NI: The latter, “NI” stands for “Niceness”. It is an integer value added to the priority base. Users only can reduce their own processes priorities by increasing the nice value. For example, if you launch a process with “NI=10”, it means that “PR= 20+10” and therefore ‘PR=30’.**
    - To change the niceness value of any process : renice -n <niceness> <pid>
- **VIRT**: Virtual memory usage. It shows the total amount of virtual memory used by the process, including both physical memory (RAM) and swap space.
- **RES**: Resident memory usage. It represents the amount of physical memory (RAM) used by the process.
- **SHR:** Shared memory usage. It indicates the amount of memory shared with other processes.
- **S:** Process status. It shows the current state of the process, such as "R" for running, "S" for sleeping, "D" for uninterruptible sleep, and so on.
- **%CPU:** Percentage of CPU usage. It represents the portion of CPU resources being utilized by the process.
- **%MEM:** Percentage of memory usage. It indicates the proportion of physical memory (RAM) being utilized by the process.
- **TIME+:** Cumulative CPU time. It shows the total amount of CPU time consumed by the process since it started.
- **COMMAND**: The name of the command or program associated with the process.

➢ Processes Management Commands

/proc

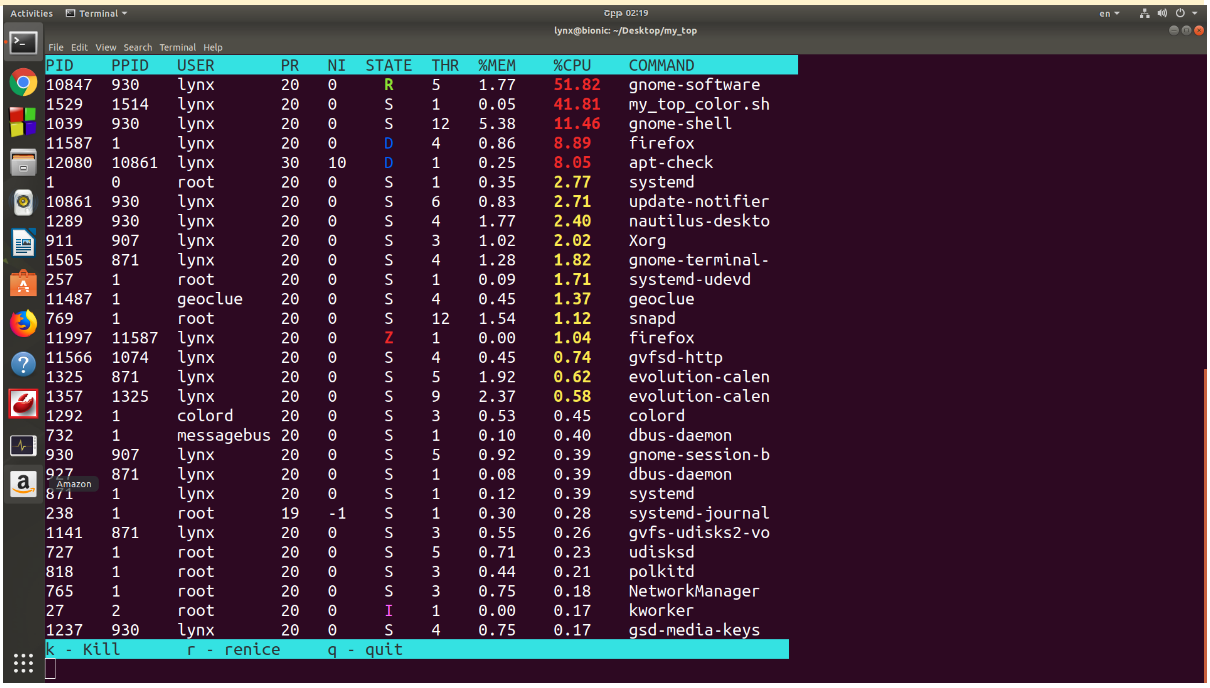
Is a special folder that keeps dynamically updated, that holds data about the status of the system. In the /proc folder there are several entries that provide information dynamically updated about the current status of the system.

/proc/<PID>

Has a list of files and sub-folders that show information of the processes ‘PID’.

/proc/<PID>/status

Shows detailed summary information of the current status of the process.



PID	PPID	USER	PR	NI	STATE	THR	%MEM	%CPU	COMMAND
10847	930	lynx	20	0	R	5	1.77	51.82	gnome-software
1529	1514	lynx	20	0	S	1	0.05	41.81	my_top_color.sh
1039	930	lynx	20	0	S	12	5.38	11.46	gnome-shell
11587	1	lynx	20	0	D	4	0.86	8.89	firefox
12080	10861	lynx	30	10	D	1	0.25	8.05	apt-check
1	0	root	20	0	S	1	0.35	2.77	systemd
10861	930	lynx	20	0	S	6	0.83	2.71	update-notifier
1289	930	lynx	20	0	S	4	1.77	2.40	nautilus-deskto
911	907	lynx	20	0	S	3	1.02	2.02	Xorg
1505	871	lynx	20	0	S	4	1.28	1.82	gnome-terminal-
257	1	root	20	0	S	1	0.09	1.71	systemd-udevd
11487	1	geoclue	20	0	S	4	0.45	1.37	geoclue
769	1	root	20	0	S	12	1.54	1.12	snapd
11997	11587	lynx	20	0	Z	1	0.00	1.04	firefox
11566	1074	lynx	20	0	S	4	0.45	0.74	gvfsd-http
1325	871	lynx	20	0	S	5	1.92	0.62	evolution-calen
1357	1325	lynx	20	0	S	9	2.37	0.58	evolution-calen
1292	1	colord	20	0	S	3	0.53	0.45	colord
732	1	messagebus	20	0	S	1	0.10	0.40	dbus-daemon
930	907	lynx	20	0	S	5	0.92	0.39	gnome-session-b
927	871	lynx	20	0	S	1	0.08	0.39	dbus-daemon
871	1	lynx	20	0	S	1	0.12	0.39	systemd
238	1	root	19	-1	S	1	0.30	0.28	systemd-journal
1141	871	lynx	20	0	S	3	0.55	0.26	gvfs-udisks2-vo
727	1	root	20	0	S	5	0.71	0.23	udisksd
818	1	root	20	0	S	3	0.44	0.21	polkitd
765	1	root	20	0	S	3	0.75	0.18	NetworkManager
27	2	root	20	0	I	1	0.00	0.17	kworker
1237	930	lynx	20	0	S	4	0.75	0.17	gsd-media-keys

- **PID:** The Process ID of the process.
- **State:** The state of the process:
    - **"R" for Running**
    - **"S" for Sleeping,**
    - **"D" for Disk Sleep,**
    - **"Z" for Zombie,**
    - **"T" for Stopped**
- Context Switches: The number of context switches performed by the process. Context switches represent the number of times the process has transitioned to or from the "Running" state. This includes both voluntary context switches (when the process voluntarily yields the CPU, such as when making a blocking system call) and non-voluntary context switches (when the operating system forcibly preempts the process due to time slicing or other reasons).

lscpu

To know how many hardware threads you have in the processor

Maybe you’ll have to change the permissions using chmod +x launch_fib.sh

a) Launch a single instance
`./launch_fib.sh 1`
b) Launch as many instances as hardware threads you have in the processor
`./launch_fib.sh <number_of_hardware_threads>`
c) Launch as many instances as you have in the system, plus two additional instances

```
./launch_fib.sh <number_of_instances_in_system + 2>
```

When we launch a single instance the time execution is very low because there is only one program being executed and it has all the access to the CPU only for it. When we launch as many instances as hardware threads we have in the processor the time increases a little but not too much. This is because every period of time that is obtained relates to one Hw thread and does not affect. However we can see that when we launch as many instances as we have in the system, plus two additional instances the time increases quite a lot compared to the previous executions. This is because there are not as many hardware resources to execute the program quickly.

SESSIÓ 9: MEMORY AND STORAGE MANAGEMENT (MARTINA)

CHECKING AND DEFINING MEMORY SPECIFICATIONS IN THE CODE

- Modifications regarding the code ***mem-stack-orig.c*** (as an example)
  - All modifications **in green** are done inside the **Recursivity** function
  - All modifications **in blue** are done inside the **main** function

Code overview (Recursivity)

```
void Recursivity(int max, int num){
	int len;
	int localVar = num;
	char buf[256];

	localVar++;
```

First modification (in green)

```
len = sprintf(buf, "The local variable localVar is located at %p when executing
              recursivity at level %d, holding the value %d\n", &localVar,
              localVar, localVar);
write(1, buf, len);
read(1, buf, 1);
```

- ★ **&localVar:** To reference the address of localVar
- ★ **localVar:** To reference the exact content of localVar

Second modification (in green)

```
len = sprintf(buf, "Returning from the function call. Thus, the local variable
              localVar located at %p in recursivity level %d still holds the
              value %d\n", &localVar, localVar, localVar);
write(1, buf, len);
read(1, buf, 1);
```

Code overview (main)

```
int main(int argc, char **argv){
	int len;
	char buf[256];
	int *ptr = NULL;

	localVar++;
```

Third modification (in blue)

```
len = sprintf(buf, "Checkpoint 1:\n The variable pointer ptr is located at %p and
              now it holds the address %p\n, &ptr, ptr);
write(1, buf, len);
read(0, buf, 1);
```

Fourth modification (in blue)

```
ptr = &globalVar;
*ptr = 123;

len = sprintf(buf, "Checkpoint 2:\n The ptr now points to %p, that is the same
              address than globalVar %p\n Thus, both have the same integer value
              %d vs %d\n, ptr, &globalVar, 123, 123);
write(1, buf, len);
read(0, buf, 1);
```

- ★ ptr: A pointer points to an address --> That's why **ptr = &globalVar**
- ★ ***ptr = 123:** Assignates content (123) to the pointer variable

➢ Memory distribution through different parts of a code

Memory distribution through recursivity

- ★ When a recursive function is called, the variable has to enter through all the necessary recursion levels **and then come out of them!**
- ★ At every level the value of the variable changes --> **Address changes**
  [The local variable localVar is **located at 0x7ffec78cdca8** when executing recursivity at level 10, **holding the value 10**]
- ★ When coming back from recursion --> The address of a value in a level already visited when going down **doesn't change**
  [Returning from the function call. Thus, the local variable localVar **located at 0x7ffec78cdca8** in recursivity level 10 **still holds the value 10**]

Allocating heap memory --> Code overview *(mem-heap-orig.c)*

```
startHeap = TopHeap();
endHeap = TopHeap();
len = sprintf(buf, "Checkpoint 1:\n Before allocating heap memory to the ptr, the
              heap memory region has a size of %d, (int)(endHeap - startHeap));
write(1, buf, len);
read(0, buf, 1);
```

- ★ Checkpoint 1: The heap memory region will have a size of 0 Bytes, we haven't initialized any global variable, so it's empty and not used

```
ptr = (int*) malloc(numItems*sizeof(int));     # malloc to allocate Heap Memory
endHeap = TopHeap();

len = sprintf(buf, "Checkpoint 2:\n After allocating heap memory ptr now points to
              %p and Heap size is %d Bytes\n Ptr holds an array of %lld integers
              (%lld Bytes)\n First item has address %p\n Last item has address %p
              \n, ptr, (int)(endHeap-startHeap), numItems, (numItems*sizeof(int)),
              &ptr[0], &ptr[numItems-1]);
write(1, buf, len);
read(0, buf, 1);
```

★ <u>Checkpoint 2</u>: We have allocated a *numItems* amount of integer items initialized as global variables in ptr, so to the heap memory region

- Heap size = (int)(endHeap-startHeap)
- ptr: Array of *numItems* integers --> Size of 4*numItems Bytes
- Addresses have changed and now

```
free(ptr);                        # ptr is no longer in the Heap section
endHeap = TopHeap();
len = sprintf("Checkpoint 3:\n After releasing the memory allocated to ptr, the
              heap region has a size of %d Bytes\n, (int)(endHeap - startHeap));
write(1, buf, len);
read(0, buf, 1);
```

★ <u>Checkpoint 3</u>: The heap region has a size of its maximum capacity range, just in case there's more memory to be allocated

<u>ANALYZING MEMORY PARTICULARITIES IN THE TERMINAL</u>

➢ Checking Heap/Stack Region size

Getting the PID of a process

```
$ ps -a
```

Showing information about the memory sections

```
$ cat /proc/<PID>/maps        # We know <PID> from executing the previous command
                              # We can also use 'more' --> It's a text file!
```

IMPORTANT!: Every line in the output corresponds to a different memory region, for the most important ones, **the name is indicated far right** ([heap], [stack], and also dynamic libraries)

example: [./mem-heap-orig 1000]
/dades/martina.massana/COM-Labs/S9/S9_files/mem-heap-orig
00ff9000-0101a000 rw-p 00000000 00:00 0 [heap]

★ <u>First column</u>: Start memory address |
★ <u>Second column</u>: End memory address | **2nd Column - 1st Column = Region size**
★ <u>Third column</u>: Permissions

Multi-Terminal checking (at the same time)

★ <u>Terminal 1</u>: Launch the program
$ ./launch.sh 0 ./program
★ <u>Terminal 2</u>: Reduced output of /maps (**ALERT!:** Heap section not specified)
$ ps -a
$ pmap <PID>
★ <u>Terminal 3</u>: Contents of the memory sections
$ ps -a
$ /proc/<PID>/maps
★ <u>Terminal 4</u>: CPU taken by the program at each step of its execution
$ top -p <PID>   # VIRT column (amount of Bytes in the virtual memory)

SESSIÓ 10: I/O AND FILE SYSTEM (LAURA)

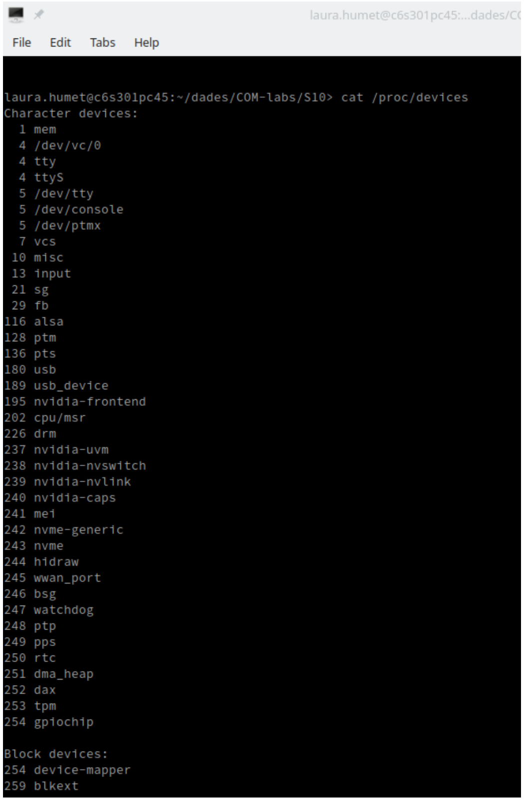
cat /proc/devices

It shows the different device types recognizable in the system, classified by “block” or “character” devices, with the major number and the name of the device.

· Major number: type of device.

· Minor number: instance type

Example: the major number tells you that it is a terminal window and the minor number tells you./ which terminal window is.



ls -l

When executing this command, we have to look into:

- The character of the first column tells you the type of the device file (i.e: “b” for block or “c” for character).
- The numbers before the timestamp column tell you the major and minor number respectively, separated by a comma.

· When executing **ps**, the **“TTY” column represents the terminal the process is bound to** --> usually pts/0 (0 meaning the first terminal opened).

./program<> /dev/pts/XXX (XXX = nombre de la terminal)

To redirect the output of the program to a new terminal XXX.

The <> are the permissions you give to the device:

- “<”: only read.
- “>”: only write.
- “<>”: read and write.

/proc/PID/fd to see the permissions of a process ‘fd’ id ‘d’ do not change it.

strace -o output.dat -e read,write ./program 0< exempleText.txt 1> sortida.dat
Registers the trace of selected system calls (i.e read and write) in the "output.dat". It reads from the file "exempleText.txt" and writes the output in the file "sortida.dat".

Buffer: region of a memory used to temporarily store data while it is being moved from one place to another. Is useful to decrease the system calls in a program (i.e: if you initialize a char buf[100] it will read 100 characters at the same time instead of one).

To open the file "Fitxer.dat" with permission only to read. If you want permission only to write --> O_WRONLY. If you want permission to read and write --> O_RDWR.

read(fd, &num, sizeof(int))
It returns how many bytes you read. Fd is the number indicated in the file descriptor (0 stdin, 1 stdout), point to the memory address num and return the size in bytes of an integer (int).

write(fd, &num, sizeof(int))
Same as read, but writing instead of reading.

SESSIÓ 11 (LAURA): PARAL·LELISME

Si volem paral·lelitzar els codis amb OpenMP, a l'hora de compilar s'hauria d'utilitzar la comanda: **g++ -o estadistiques estadistiques.cpp -fopenmp -O3**

Si volem configurar la variable d'entorn que determina el nombre de threads que volem utilitzar enel codi paral·lel: **OMP_NUM_THREADS=4 ./estadistiques**

OPENMP COMMANDS FOR YOUR CODE

· **#pragma omp parallel num_threads(n) {}** : to parallelize part of your code using n threads. You can use this command without the "num_threads(n)" to parallelize your code.
· **omp_get_thread_num()**: returns the number of the thread executing the instruction.
· **#pragma omp for**: to parallelize loops (only the *for* loops!).
· **#pragma omp critical {}**: to protect a shared variable "var". If the variable is private (initialized inside the parallel section) it is not necessary to protect it.
· **#pragma omp parallel private(var) {}**: to make the variable "var" private.
· **#pragma omp parallel shared(var) {}**: to make the variable "var" shared.
· **#pragma omp task {}**: when it finds an available thread, it assigns the instruction to it.
· **#pragma omp taskwait**: like a barrier.

→ You can combine the commands, for example, **#pragma omp parallel for private(i,j,k) shared(A,B,C) num_threads(atoi(argv[1]))** → i,j,k are the variables we want to be private, A,B,C the variables we want to be shared and argv[1] is the number of threads we want to use.

→ When we initialize the variable outside the parallelized code it takes (by default) type shared, which as we've seen in theory lectures, all threads can modify the variables simultaneously. That's why **we need to protect the variables**.

IT'S IMPORTANT TO COMPILE THE PROGRAM USING THE FLAG "-fopenmp" TO LINK THE OPENMP LIBRARY.

DEFINIR VARIABLE GLOBALS

```
#define MAXVECTOR 300000
int vector[MAXVECTOR]
```

SABER LA MIDA D'UN FITXER ls -sh

SABER L'ADREÇA DE LES VARIABLES GLOBALS nm ./nom_executable

MAKE FILE DEFINITIU ESTÀTICAMENT I DINÀMICAMENT:
implementa un Makefile amb regles per: compilar tots els fitxers d'aquesta prova; compilar
individualment cada programa; esborrar aquells fitxers que s'hagin generat durant la compilació.

```
all: matrix-row matrix-column matrix-row-dyn matrix-row-sta

matrix-row-dyn: matrix-row.o
	gcc -o matrix-row-dyn matrix-row.o

matrix-row-sta: matrix-row.o
	gcc -o matrix-row-sta matrix-row.o -static

matrix-column: matrix-column.c
	gcc -o matrix-column matrix-column.c

matrix-row: matrix-row.o
	gcc -o matrix-row matrix-row.o

matrix-row.o: matrix-row.c
	gcc -c matrix-row.c

clean:
	rm -f *.o
	rm -f matrix-row matrix-column matrix-row-dyn matrix-row-sta
```