DE VINCI INNOVATION CENTER

iiM LÉONARD DE VINCI | INSTITUT DE L'INTERNET ET DU MULTIMEDIA PARIS-LA DÉFENSE

ESILV LÉONARD DE VINCI | ÉCOLE D'INGÉNIEURS PARIS-LA DÉFENSE

Operating System And Programming

Workshop Sheet

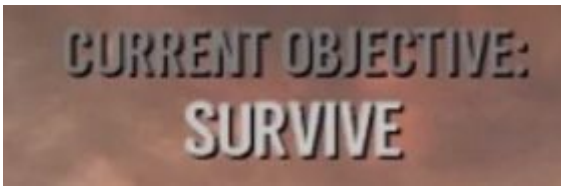
Directions

The Workshops are designed to start slowly and to quickly ramp up. The first steps are detailed, and you will have to look up more and more things as you progress. Try to go as far as you can.

1. GNU / Linux

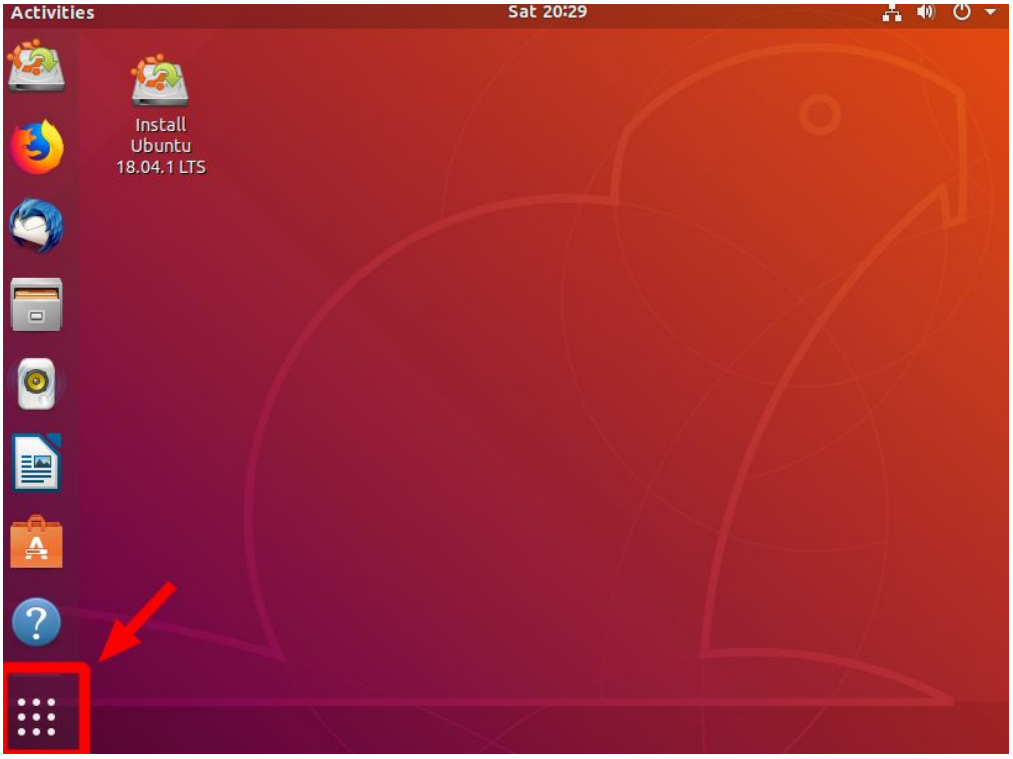
a. Exploring The Terminal

i. Lv1: “Objective: Survive”

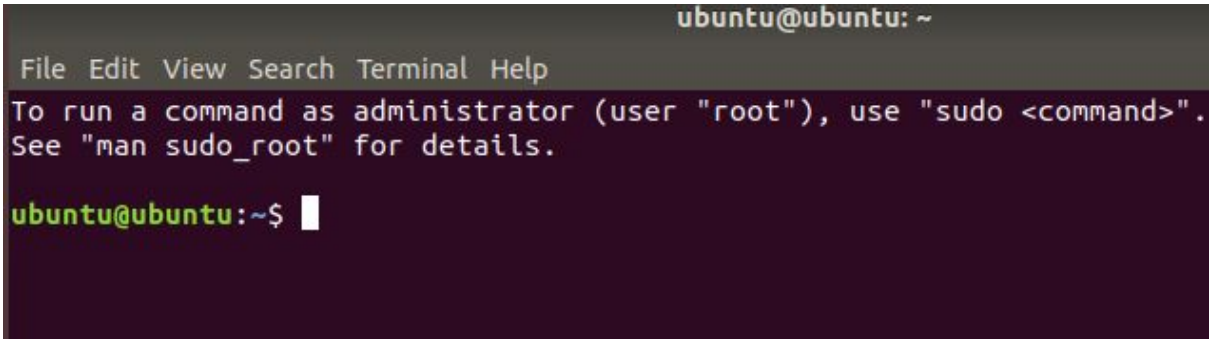


In this workshop, you will mainly work with the command line. The terminal can be intimidating or seen as counter-productive but it is, still today, the most reliable, and most efficient way to tell a machine (especially a linux one) what you want to do.

- First thing to do is to start a terminal. Click on the “Show Appllications” button and look for “terminal”



The terminal greets you with a command prompt:



- Input the commands **id groups pwd ls**

```
File  Edit  View  Search  Terminal  Help
ubuntu@ubuntu:~$ id
uid=999(ubuntu) gid=999(ubuntu) groups=999(ubuntu),4(adm),
gdev),116(lpadmin),126(sambashare)
ubuntu@ubuntu:~$ groups
ubuntu adm cdrom sudo dip plugdev lpadmin sambashare
ubuntu@ubuntu:~$ pwd
/home/ubuntu
ubuntu@ubuntu:~$ ls
Desktop    Downloads  Pictures  Templates  examples.deskto
Documents  Music      Public    Videos
ubuntu@ubuntu:~$
```

- Look up what these commands are used for or use the **man** command (use 'q' to exit the man command)
- Input the command **uname -a**

```
ubuntu@ubuntu:~$ uname -a
Linux ubuntu 4.15.0-29-generic #31-Ubuntu SMP Tue Jul 17 15:39:52 UTC 2018 x86_64 x86_64 x8
6_64 GNU/Linux
ubuntu@ubuntu:~$
```

  Here the **-a** partis called an **argument**. Arguments are everything that comes after a command and is given to the command.
  - Look up what other arguments the command **uname** accepts.
  - Explain the output of the commands
    - uname -r
    - uname -m
  - Try out the **--help** argument on all the commands you have seen so far
    NB: Notice that 1 letter argument only have 1 '-'
    You can combine 1 letter arguments, for example:
    Try the command: **uname -r -m**
    Try the command: **uname -rm**
- Try out the **clear** command. Try to press **Ctrl + L** while focused on the terminal.
- Try to press **Ctrl + D**. Open the terminal again. Look up what this combination actually does.

ii. "WHERE ARE WE???"

Next step is to be able to navigate across folders.

- You are currently in a directory, you can know which one using the **pwd** command.
- Use the **ls** command to list the files in this directory, use **ls -ls** or **ll** to have a bit more details. Explain what each column means.
- Input **cd /** to change directory to the **/** directory. This is the root of the file system. list the files in it.

```
ubuntu@ubuntu:/$ ll
total 2
drwxr-xr-x   1 root root  260 Aug 22  2020 ./
drwxr-xr-x   1 root root  260 Aug 22  2020 ../
drwxr-xr-x   2 root root 3372 Jul 25  2018 bin/
drwxr-xr-x   1 root root   60 Jul 25  2018 boot/
dr-xr-xr-x   1 root root 2048 Jul 25  2018 cdrom/
drwxr-xr-x  19 root root 3860 Aug 22  2020 dev/
drwxr-xr-x   1 root root  600 Aug 22 20:29 etc/
drwxr-xr-x   1 root root   60 Aug 22  2020 home/
lrwxrwxrwx   1 root root   33 Jul 25  2018 initrd.img -> boot/initrd.img-4.15.0-29-generic
lrwxrwxrwx   1 root root   33 Jul 25  2018 initrd.img.old -> boot/initrd.img-4.15.0-29-gene
ric
drwxr-xr-x   1 root root   60 Jul 25  2018 lib/
drwxr-xr-x   2 root root   43 Jul 25  2018 lib64/
drwxr-xr-x   1 root root   60 Aug 22  2020 media/
drwxr-xr-x   2 root root    3 Jul 25  2018 mnt/
drwxr-xr-x   2 root root    3 Jul 25  2018 opt/
dr-xr-xr-x 162 root root    0 Aug 22  2020 proc/
drwxr-xr-x  22 root root  379 Jul 25  2018 rofs/
drwx------   3 root root   60 Jul 25  2018 root/
drwxr-xr-x  29 root root  880 Aug 22  2020 run/
drwxr-xr-x   2 root root 4127 Jul 25  2018 sbin/
drwxr-xr-x   1 root root  220 Aug 22 20:28 snap/
drwxr-xr-x   2 root root    3 Jul 25  2018 srv/
dr-xr-xr-x  13 root root    0 Aug 22  2020 sys/
drwxrwxrwt  13 root root  280 Aug 22 20:30 tmp/
drwxr-xr-x   1 root root  100 Jul 25  2018 usr/
drwxr-xr-x   1 root root  180 Jul 25  2018 var/
lrwxrwxrwx   1 root root   30 Jul 25  2018 vmlinuz -> boot/vmlinuz-4.15.0-29-generic
lrwxrwxrwx   1 root root   30 Jul 25  2018 vmlinuz.old -> boot/vmlinuz-4.15.0-29-generic
ubuntu@ubuntu:/$
```

- Each folder has a particular signification and is essential to the OS function.
- When you use the **ls -ls** command, notice the ".." folder at the top. Try to use **cd ..** in different folders. What happens ?

iii. Execution, environment and piping

- When you run any command, the terminal has to find its executable before… executing it. This process uses the PATH environment variable.
- Run the **env** command to list the currently set environment variables.
- Identify the PATH variable. Or use a pipe to the **grep** command to search for the "PATH" text:

```
ubuntu@ubuntu:/usr/lib/firefox$ env | grep PATH
WINDOWPATH=1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/gam
es:/snap/bin
ubuntu@ubuntu:/usr/lib/firefox$
```

(Don’t hesitate to run **grep --help**)
The PATH env variable is made of path separated by a :. List the files in some of these paths. What do you notice ?

- The **|** (pipe) character is used to take the output of the left command and use it as the input of the right command
    - Use the **ls** and grep command with a pipe to find a file in /etc that contains the word “sysctl”
- Use the command **cat** to display the content of the /etc/sysctl.conf file.
- Use the **more** command on the same file. Use the **less** command. What are the differences ?

iv. “Am i admin yet ?”

It is time to become an Administrator.

- Try to go in the /root directory. What’s happening? Why ?
- Use the command **sudo -s** then the **id** command.
  Your password won’t show, this is normal, you are not writing nothing.
  If the result is **uid=0(root)** Congratulations, you are now in administrator mode.
- **Warning: In admin mode, the system rarely asks for confirmation. Be. Very. Careful. Try to spend as less time as possible in this mode. Enter exit or Ctrl+D to exit the mode.**
- You can prepend the **sudo** command to any command to execute it as administrator.

v. I want new stuff

Ubuntu is shipped with the apt package manager. Most of the programs you will need or want to install are available with apt-install.

- Run **apt-get update** to get a fresh list of packages. (Don’t forget the sudo)
- Run the **htop** command, install it if necessary
    - Try to display the /etc/shadow file. Try again with **sudo**. What is this file used for ?

vi. Access Control

About the access control.

- Create a new folder, not as root, create new files with **touch** and add data in them with a text editor
- With **chmod** and **chown** :
    - Change the owner of a file to root, make the file readable by you but not the other users, test this behaviour
    - Make a file executable but not readable, test this behaviour
    - Using **addgroup** add the *test* group.
    - Make a file writable by the group *test* but not by you. Test this behaviour
    - Add yourself in the *test* group with **gpasswd.** Test the behaviour again
    - Delete the file owned by root. Did you have to use sudo ?

- Make a folder, add files in it and make root the owner of that folder. Can you delete these files now ?
- Make the folder not-executable for everyone and the group. List the files in that folder.
- What permissions are necessary on the folder to delete the file owned by root ?
- What happens if you only give the read access to a directory ?

b. Exploring The Filesystem

i. “Exploring the maze”

Let’s dive into the way the filesystem works.

- Go to the /boot directory, find the vmlinuz-XXX file
- Use the command **file** with the vmlinuz-XXX file path as argument. (Use tab to auto-complete the command)

```
ubuntu@ubuntu:/$ file /boot/vmlinuz-4.15.0-29-generic
/boot/vmlinuz-4.15.0-29-generic: Linux kernel x86 boot executable bzImage, version 4.15.0-2
9-generic (buildd@lgw01-amd64-057) #31-Ubuntu SMP Tue Jul 17 15:39:52 UTC 2018, RO-rootFS,
swap_dev 0x7, Normal VGA
ubuntu@ubuntu:/$
```

This is indeed the file containing the Linux kernel you are currently using.

- The /usr/lib contains the system libraries. Run the **file** command against some .so files.
- Use the **du** command to figure out the size of the /usr/lib folder.

Directories can be used as **mount points** meaning a certain filesystem can be attached to a directory and be displayed in it.

- Run the **mount** command to display the currently mounted filesystems on your machine.

```
ubuntu@ubuntu:~$ mount
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
udev on /dev type devtmpfs (rw,nosuid,relatime,size=1990612k,nr_inodes=497653,mode=755
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=000)
tmpfs on /run type tmpfs (rw,nosuid,noexec,relatime,size=404028k,mode=755)
/dev/sda1 on / type ext4 (rw,relatime,errors=remount-ro,data=ordered)
securityfs on /sys/kernel/security type securityfs (rw,nosuid,nodev,noexec,relatime)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)
```

Notice that:

- The /sys /proc /dev folders are mount points.
- The / folder is also a mount point ! You can look up the linux boot sequence to learn more about this.

- List the files in the /dev folder
- Display the contents of the /dev/urandom file. Use **Ctrl + C** to make it stop.

- What size is the /dev directory ?
- Check that you have the **hexdump** command installed. Display the contents of the /dev/sda file, pipe it in the **hexdump -C** command, pipe that in the **more** command. This is the actual content of the hard disk where (most likely) Ubuntu is installed ! This file is readable but also **writable** and directly tied to your hard drive. **It means that writing to that file will immediately write it to your hard disk**… Which will most likely induce data (and time) loss. Especially if your Windows is installed on /dev/sda, this could be a disaster. Other files in /dev such as /dev/sdX or /dev/mmblckX or /dev/nvmeX are also tied to storage devices. Be careful

The /dev directory is actually a “pseudo filesystem”, it is not written on a disk but is generated by the system when accessed.

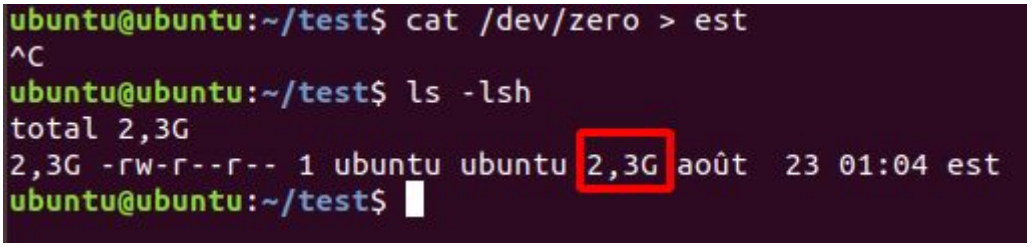
- Create a new directory in your home folder. Use the **mount** command to mount a dev filesystem on this new folder. Use the **umount** command to unmount it.
- Create a file in this directory and mount the filesystem again. What happened to the file. Unmount the filesystem. What about the file you created ?

ii. Writing to files

- Input the **echo Hello World !** command. What is the echo command used for ?
- Go to your home directory (**cd ~**)
- Input the **echo Hello World ! > test** command. What happened ? List your files.
- Using **rm**, remove the test file you just created after displaying its contents.

The > symbol is used to redirect the standard output of a command to a file.

- Write something to a test file. Using **mv** rename this file to test2
- Create a new folder with **mkdir**. Move your test2 file there.
- Output the contents of test2 in the new directory.
- Display the contents of the /dev/zero file. Redirect it to a new file in your directory. /!\ The command will fill up your disk quickly, if the command runs for more than 2 seconds use **Ctrl + C** to stop it !



- What size is this new file ? Display its contents. Try using **hexdump.** Remove it. Lookup what the /dev/zero file does.
- Try to write something to the /dev/full file.
- Try to write something to the /dev/null file. Display its contents.
- Use the **dd** command to write 1024 bytes of /dev/random to a test file.
- How could you use **dd** to make a bootable USB flash drive ? (Try it if you have a usb you wish to wipe out. Call a teacher before running the command)

c. Exploring The Processes

i. The /proc filesystem

All filesystems are not necessarily written on a disk, an interesting example of such a file system is the procfs or /proc filesystem.
The /proc filesystem is a pseudo filesystem to view different elements about the currently running processes. It can also configure a lot of kernel parameters. The complete guide of /proc is available here: https://man7.org/linux/man-pages/man5/proc.5.html

- Go to the /proc/self directory and list the files. Wants of t does this folder represent ?
- Display the contents of the “maps” file.
- Display the contents of /proc/meminfo multiple times. What do you notice ?
- What do /proc/uptime /proc/interrupts contain ?
- You can mount a filesystem like /proc anywhere. Mount it somewhere in your home folder
- What is the /proc/sys/net/ipv6/conf/all/disable_ipv6 file used for ?

ii. Seeing the syscalls using strace

A very interesting program that can help you troubleshoot is **strace**. Strace is used to see the system calls and what they return.

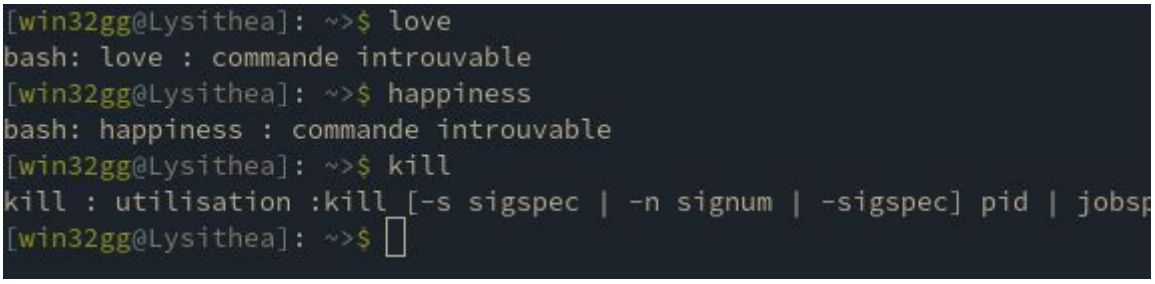
- Install **strace**
- Use **strace pwd**
  Each line is a system call. Notice that even for a very simple program such as pwd you have a lot of syscalls fired by the loader. Look for some of the syscalls here, namely execve getcwd brk and mmap
- Strace works with any program, you can see all calls to the kernel for all the programs you use. Try it with mkdir, cat, etc…
- Try it with cat /proc/meminfo Notice that the syscall is exactly the same with a regular file.
- Without being root, try to **strace cat /etc/shadow** Look for the failing syscall.
- The last line of strace is *** exited with X ***. This number is called an exit code. A 0 exit code usually means that everything went well, 1 means some kind of error happened. Verify this with the last commands you ran;

iii. Using htop to get insight of the processes

An essential monitoring tool for a linux system is htop. It displays the cpu state and usage, the processes and lots of information about them.

- Install **htop**
- Run htop, in the options, enable the “detailed cpu time”. What is CPU time and how is it split ? What is the red bar in the cpu usage in htop ?
- What process niceness ?
- What is the difference between the virt, shr and res columns ?
- What process uses the most cpu right now ?
- Attach **strace** to a web browser and find the syscalls used to access the internet.

iv. kill & signals



Signals are used to send basic information to processes. The kill command is used to transmit a signal to a process.

- Use **kill -l** to have a list of available signals
- What are SIGHUP, SIGINT and SIGSEGV used for ?
- What is the difference between SIGKILL and all the other signals ?
- In a new terminal, enter **sleep 10000**
- Use the **ps** command with **grep** to find the PID of the sleep process that you just launched. Use the **kill** command to stop it.
- Using **strace**, what syscall is used to transmit signals ?
- **BONUS**: A bit harder, find a way to find the PID and kill it in a single line.

d. Using Bash

Bash (Bourn Again Shell) is what you have been using so far to input your commands. It is also a scripting language that can be used to execute commands, parse inputs etc… The syntax is a bit unusual because the bash language revolves around executing programs.

- Create a .sh file where you will be writing your bash scripts. The first line of a bash script is called a *shebang* and is typically **#!/bin/bash** Look up what a shebang is used for.
- Create a new folder, inside that folder create a hundred new folders. called 1 to 100
- Write a bash script that says Hello World ! Make you bash script executable using **chmod** and execute it.
- Write a bash script that asks for an input and outputs it to the console.
- Write a bash script that asks for 2 numbers and output their sum.
- Write a bash script that says “Careful what you type” if it is run as root.

- Write a bash script that outputs the current date and user
- Write a bash script that reads and outputs the contents of the /etc/issue file
- (hard) Write a bash script that outputs the available memory ram in green or red if less than 20% is available (Use: https://misc.flogisoft.com/bash/tip_colors_and_formatting the **expr** command and the **awk** command)

e. Some Assembly Required

To give you an idea of what a computer is doing and to make you have a deeper understanding of the next lessons, we will do a little x86 assembly.

- Install **nasm**

Use this file as a start file and complete the indicated section to make the program print the “Hello World” message in the *msg* mnemonic

```
section    .text ; declare text segment, where the actual program
code is.
   global _start  ;must be declared for linker (ld)

_start:     ;tells linker entry point, next lines are the
instructions to execute the program.
      <COMPLETE HERE>

section     .data ; the data section where the constants of our
program are stored. Here the constants are the string we want to
display and its length.
msg db 'Hello, world!', 0xa  ;string to be printed. The msg is
mnemonic equivalent to a variable name, db means that we declare
bytes of data, initiated to ‘Hello, world!’ and the 0xa byte which
is the new line character ‘\n’
len equ $ - msg   ;length of the string, len is the mnemonic, equ $
- msg is the difference between the current address ‘$’ and the
adress of msg
```

To help you, use:

- The x86 syscall wikibook: https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux#syscall
- A linux syscall table: https://filippo.io/linux-syscall-table/
- The write(2) syscall documentation: https://man7.org/linux/man-pages/man2/write.2.html
- The x86 **mov** instruction: mov <destination>,<source>
- You will have to use **nasm** to produce an elf64 object file. Then, use **ld** to produce the executable.

- Run the program with strace, what do you notice ?
- Make the program exit with code 0, using the **exit** syscall

f. Going further

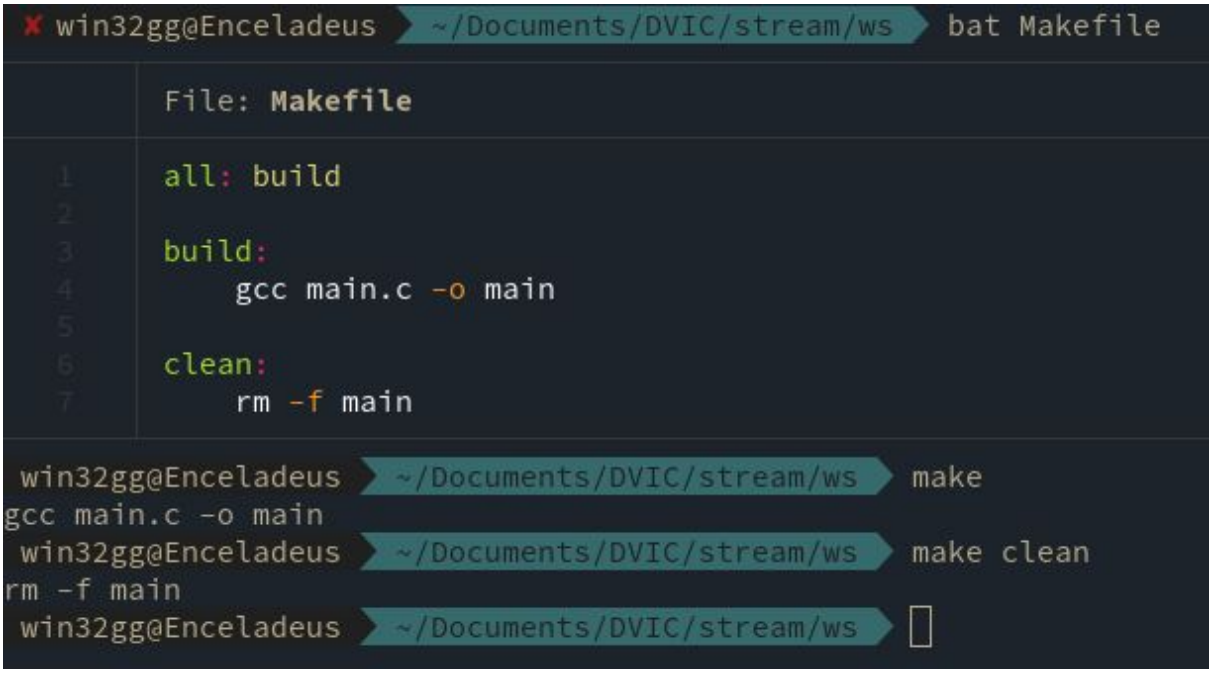
- FIle types
  - Using **ln** create a symbolic link in your folder to the / directory
  - Create a file similar to /dev/null in your home directory using **mknode**
  - Create a FIFO file named myfifo in your directory, show its contents using **cat** In another terminal, while **cat** is waiting, write something to the fifo.
- Ergonomy & Comfort
  - Install **zsh**, **git** and **curl**
  - Install Oh-My-ZSH: https://github.com/ohmyzsh/ohmyzsh#basic-installation
  - Choose a theme https://github.com/ohmyzsh/ohmyzsh/wiki/Themes
  - If zsh is to your liking, change your default shell using **chsh**

- Sharing your work
    - You will have to use git https://git-scm.com/ to share your code and submit your work. You can have a look at it now.

2. C Programming

a. Setting up your tools

Often, C lessons use all integrated IDE for windows with the MINGW emulator. In this workshop you will use a combination of a text editor, the GNU compiler suite, and the **make** command.

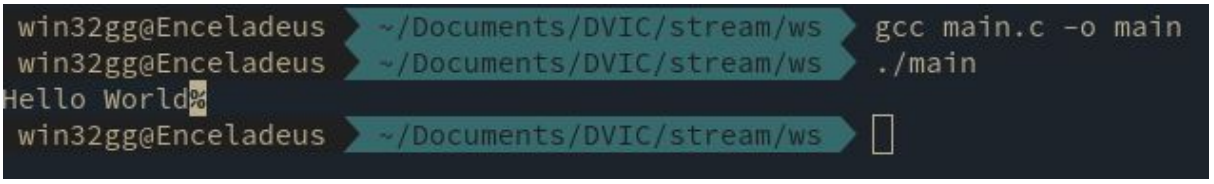


i. Text editor

First thing you will need is a text editor. There are a **LOT** available, the most versatile and famous are VsCode and Atom. You can also use sublime text, gedit, or even vim if you feel fancy. Depending on what text editor you'll choose, the installation procedure will be different. The important thing is that you can edit text, have syntax highlighting and debugging.

ii. Compiler

The gcc compiler is automatically installed on your machine. You can use it by invoking the **gcc** command. The straightforward way to compile is just **gcc file.c -o program** which will create an executable. Check that the **gcc** command is available.



iii. Additional tools

A common tool used for compilation is the **make** tool. Inside a Makefile you can specify whan commands to run to compile your program or to clean your workspace. You can input the command by hand each time of course, but using Makefile can be helpful. You can see an example of a Make file here:
https://riptutorial.com/makefile/example/16459/basic-makefile

b. Hello world

i. Hello

A good first program in a language is the hello world. In C this takes place in 3 parts:

- #include<stdio.h> To import the printf function
- int main() Which is the entry point of the program
- printf("Hello World\n"); Where we say hello :)

Write your hello world program, compile it and make it run.

ii. Include files & prototypes

A quick tangent to talk about the "#include" directive you just used.
In C any line that starts with # is a preprocessor instruction. This means that line is interpreted by the compiler and never makes it to the executable. the #include directive imports the declarations from the referenced file. The .h file stands for Header file, and contains "prototypes" of the functions. It defines the name of the function, the return type and the arguments. The header files a bridge to go from our application to a library code: they act as API.

iii. Something about char*

Because C is a rather low level language, it is very common to give as arguments the start of a memory structure (pointer, structure etc…) and its length. Because strings are so widely used, it was decided that the character '\0' would mark the end of a string, this way you don't have to supply its length when you want to read it.
But remember to append the \0 character when necessary. (Usually specified in the doc)

c. Reading from stdin

To do interesting programs, we need to be able to read user input. This is actually more complicated than in a high level language. Use the **fgets** function to read a line from stdin or use **scanf** to directly read an integer or float, even if you can convert it later.
Be mindful that fgets accepts a **FILE***

- Make a program that accepts a string on stdin and rewrites it on stdout.
- Make a program that reads a string on stdin and outputs it on stdout only if an integer was inputted. (Use the **atoi** function)

- Make a function that asks for an integer, reads it from stdin and keep asking while the integer is not in a certain range supplied to the function

d. Pointers

Memory manipulation is the very base of the C programming language and computer science more generally. A fundamental tool for this purpose are pointers.
Pointers are variables that contain a memory address. You can make a pointer to any kind of data. Remember however that when using a pointer you have to think about where is your pointer variable, and where is the data that you point to.
Usually the data you point to is in the *heap*. You can reserve memory in the heap using the **malloc** function. When you don't need the memory anymore, use the **free** function on the pointer to free the memory. Malloc is often used with the **sizeof** function to allocate a precize chunk of memory.

- Write a program that outputs the size of an int variable, the size of an int* variable.
- Using the & operator, write a program that outputs the address of an int variable in the stack, and the address of one in the heap. (Use %p in printf) compare it to a map layout of a process in /proc/[pid]/maps
- Using the **sbrk(2)** syscall find and print where the heap actually ends.
- Write a program that allocates an integer in the heap using malloc
- Write a program that allocates 40 integers in the heap and sets them all to 0.
    - Foreshadowing bonus: Do the same using the **memset** function from the <string.h> import
- Write a program that asks for numbers (integers), stores them in the heap and when no number is supplied, prints the total sum of what was inputted.
- Double Pointers: Write a program that reads user input, and returns the sum of the string lengths that were inputted. The program computes the length when an empty string is supplied. To store the strings you may have to use double pointers (pointers to pointer).

You can also create a pointer to a function.

- Create a function with the signature "void my_func(int a)" that displays the number a
- Create a pointer to this function, call this function by dereferencing the pointer you created (use a static argument)

e. Structs

Structs are a way to layout contiguous memory. Structures can contain normal variables, arrays, pointers etc..

- Declare a Point structure that can hold 2 float.
- Initialize 2 of Point structures in the stack. Then in the heap.
- Write a program that asks the user how many points he is going to enter, then read each x and y of these points, then compute and output the centroid of this set.

- Write a program to manage a library: From a main menu you can choose to list the books, add a new one made of an id (int) and a title or to delete one. You can't have two books with the same id.
- Write a program to multiply matrices. The user should be asked the dimensions of the matrices, to input each element and the matrice be printed.

f. File manipulation

An important part of a language is how it interacts with the filesystem. With C you will mainly use either the libc FILE* abstraction (or stream) with the fopen, fread & fwrite function or the syscalls with the file descriptor directly.

- Syscalls
    - Use the **open(2)** syscall to create and open a file. Use **write(2)** to write Hello inside of it.
    - Open a file and use **read(2)** to read the first 50 bytes. Display them on screen.
    - Don't forget to close your files with **close**
- FILE*
    - Use **fopen** and **fwrite** to write Hello stream in a file.

- Write a program that displays the content of a file whose name is given as an argument.
- Use the library exercise from before and add options to write the library to a file and read it.
- Write a program that outputs the data given in stdin to the file path given in argument.
- Using the **stat** syscall write a program that outputs the size of a file given in arguments in bytes

g. Threads

Threads are useful to make multiple parallel tasks in the same process.

- Use the pthread library
    - Add the -lpthread to your compilation line
    - Use #include <pthread.h>

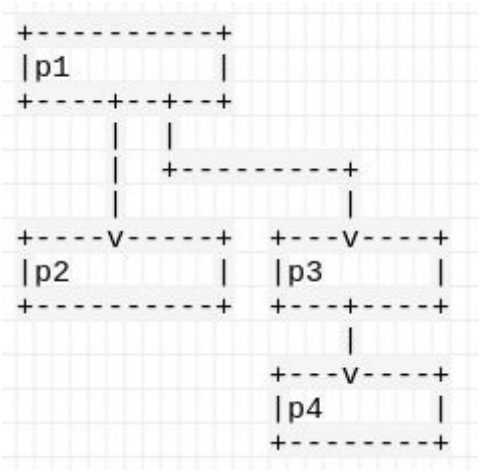
- Use pthread_create to create a new thread that will wait 5 seconds before returning
- Use pthread_create to create two threads that increment the same counter until it reaches 100. Print the counter each step. What happens ?
- Use **pthread_mutex_lock** to create a critical section where only one thread has access and increments the counter. Don't forget to unlock
- Create a program that initializes an array of 100 integers from 1 to 100 and distributes the tasks of calculating the inverse of each of these numbers in a new

memory area (of doubles). Make the numbers of integers and the numbers of threads adjustable and print out the time each thread spent calculating as well as the total time of the program. Try different values.

h. Forking

Forking is used to create new processes. The **fork(2)** syscall has tricky behaviour: When called, the process is cloned and fork() returns in the 2 processes: it returns 0 in the child process and returns the PID of the child process in the parent process.

- Fork a process and print the child process and parent process pid using the fork return and **getpid.**
- Write a program that forks to match the following diagram, where each box is a process and each arrow is a fork.



- Print the result of **sbrk(0)** for a parent and a child process. What do you remark, why is that ?

i. Inter Process Communication

By default, processes are supposed to be separated and offer a way to separate user's activities from one another. Sometimes it can be useful to make process communicate.
The IPC can take 2 forms: memory sharing and message passing.
The memory sharing can be useful to make multiple processes share a large chunk of data asynchronously, while message passing can be used to trigger events in the recipient process. You can see how many IPC objects are preset on your system with the **ipcs** command.

i. Message passing

You have multiple ways to pass messages across processes. A common way is to create a Unix Socket, you can also use a local network connection, or you can use **msgget**

Msgget works in 2 steps:

- Get the key for the IPC communication channel using **ftok** (this step is optional, the key can be arbitrary. ftok is used to help generate a key)
- Use **msgrcv** or **msgsnd** to receive or send messages to the queue.

- Write 2 programs, one for receiving and one for sending a message on an arbitrary IPC key that you will have chosen.
- Make the user input the message before sending it.
- What happens if two processes are waiting for a message ?

ii. Memory Sharing

First, some fun with **mmap**, unrelated to IPC

- Use **mmap** to map a file of your choosing to memory, preferably a small file you have created yourself for this test. You will notice mmap asks for the size of the mapping I.e the size of the file, you can use **stat** to find this out.
  You can supply a NULL address, mmap will attribute a location for your mapping based on its size. Use PROT_READ for prot and MAP_PRIVATE for flags.
- Make your program print its pid using **getpid** and make it wait for user input.
- While your program is running, display the contents of /proc/[pid]/maps. What do you see ?

And now,

- Using *PROT_READ*|*PROT_WRITE* and *MAP_SHARED*|*MAP_ANONYMOUS* make a shared memory space between two process using fork (make the call to mmap, then to fork) (mac users may have a hard time with MAP_ANONYMOUS, which is not implemented on OSX)

j. Signal handling

Except for SIGKILL, processes can set up specific functions to be called when they receive a signal using **sigaction(2).** The sigaction api makes use of function pointers, have a look at the example section in the documentation.

- Write a program that exits after 3 Ctrl + C instead of 1.
- What is the **nohup** command used for ?

k. Problems

Some problems that can take you some time, ordered from easiest to hardest

- Write a program to convert a base 10 number to any base between 2 and 16, the base should be provided as an argument.

- With the **sbrk** syscall, explain, then implement how to make your own **malloc(3)** and **free(3)** functions. How would you do it with the **mmap** syscall?

- Write your own **printf(3)** function using **write(2)** and **Variable Arguments**. Handle %i and %s at first. Use **itoa** or sprintf if it is not available.

3. Python Programming

The Python programming language is a modern language widely used for scripting, machine learning, system management etc… The application of this language has become extremely wide over the years and Python even passed Java on the StackOverFlow language popularity survey in 2019 and is the language that is growing the fastest.
Python offers a great flexibility, Oriented Programming, low level interfaces, automatic typing and garbage collection.

a. Setting up the tools

We will first go briefly over the tools that you will have to use to make the best of your Python classes and later.

i. Python itself

Python comes in 2 big versions: Python2 and Python3. Python2 has now officially expired and you only use Python3. The python executable is the entry point of the python language, it will read your python file, interpret it, parse it and execute it. You should use it like so:
python3 script.py
(The **python3** command should be available. You can also use the **python** command directly but it can be a link to python2 or not even exist.) You can see the source code of python here: https://github.com/python/cpython

- Check your python version

ii. pip

Pip is the Python package manager. You can use it to install new Python packages.

- Install the numpy package using **pip**.
- List the currently installed package
- What is a requirements.txt file ?

iii. IDE

You can use any text editor to code in Python. It is recommended to at least have a syntax highlighting editor. Some editors provide debugging information which can be useful. We encourage you to try out multiple IDE: Atom, VSCode works well, but you also have Python oriented IDE like PyCharm, Spider etc…

- Pick your weapon, choose an IDE and install it.

- If you are using VsCode or Atom, install the Python extension

b. Python basics

Let's go over the basics of the Python language.

i. Automatic typing & Garbage collection & Interpreted

Python is auto-typed, meaning that variables don't declare a type. The same variable can go from an int to a float. You don't have to **free** the variables, Python does that for you, so you can just focus on writing code. Because it is interpreted you don't have to worry about compiling either.

To make you familiar with the Python variable and flow control, start with these exercises:

- Make a python script that outputs Hello World
- Make a python script that prints integers from 0 to 100
- Make a python script that reads a value from the user using **input** and prints it.
- Make a python script that reads a value from the user and outputs Hello Python as many times as the user entered.
- Make a python script that reads a value from the user and outputs its log value, if the number given is <= 0, the script outputs nothing.

ii. Modules

Python modules are groups of files defining functions, variables and access points for a programmer. Each module has a specific purpose and can require several other modules to work correctly. Modules can be built-in the Python language, or added to your system using the pip package manager.
In a Python project, the files can call each other. It is important to remember 2 things:

- An import is relative to where the executable is launched.
- A __init__.py is required in all the folders where modules are present.

- Create a python project (just.. a new folder, an empty requirements.txt and a README)
- Create a *test* module, which is a folder and a *test_module.py* in it.
- In the root of your project create a main.py file which import your test_module
- Import the math module, the numpy module, make an alias for this import as "np"
- What happens if you put a hello world in the __init__.py file ?

iii. Lists & Dictionaries

List and dictionaries are the most useful, most used Python data structures. They are respectively used to store data contiguously and in an ordered way and to organize data with a key/value association.

- Write a program that reads numbers from the console until an empty line is entered, then outputs the numbers backward.
- Write a program that reads numbers from the console until an empty line is entered and outputs the number of times each number has been entered.
- Write a program that reads numbers from the console until an empty line is entered and outputs the numbers from the lowest to the highest value.
- Write a program that reads a sentence and outputs the number of occurrences of each character.
- Write a program that checks of two lists have at least 1 element in common
- Write a program that converts a list of single characters to a string
- Write a program to insert a string at the beginning of every item in a list
- Write a program to merge two dictionaries.
- Write a program that removes even numbers from a list of integers

iv. Functions

Python functions are defined with the keyword **def**, functions are also not typed and can return different types of value. You can return multiple values using python's tuple and unpacking. Using **kargs** and **kwargs** you can pass an arbitrary amount of arguments to a function. (The C equivalent would be **Variable Arguments** if you're interested.)

- Write a python function that returns the square value of the passed argument if it is >0 or the exponential of the argument otherwise
- Modify the function to return both values in a tuple and unpack the returned value in the calling script.
- Write a function that takes an arbitrary amount of arguments and returns the sum of these arguments.
- Modify the function to return the sum, mean and standard deviation of the passed argument list.
- Write a function that returns the min and max value of an arbitrary list of arguments.

v. Files

Python has interfaces to manipulate files on the machine. It mainly used the **open** function to open a file and get its file descriptor.

- Write a program that creates a file and writes what the user inputs in the console to it, until an empty string is entered.
- Write a program that reads a file where each lien is a number and outputs the sum of the read numbers.
- Using **json.dumps** from the json module, write a program that reads a string from the console and writes to a file the number of occurence of each character in a JSON format.
- Write a program that creates a folder, reads a number n from the console, then creates files 1 to n in the new folder, each file containing the "This was created by python" string.

- Write a program that reads the /proc/self/maps file and outputs a list of information about each mapping, including the memory range, protection and name.
- Using **os.stat** and **os.isdir** or **os.isfile** and a recursive function, write a program that scans a given directory and outputs the path of the 5 biggest files, and the total size of the scanned directory.

vi. Traceback

Tracebacks are Python's way of handling Errors. A Traceback is a list of the functions that were called before the program encountered an error. You can intercept an exception with a **try except** block and have your own code take care of the exception. You can, for example, change the return value of a function if it encountered an error.

- Write a function that writes a list to a file, with each element on a different line the function prints the error if it fails to write instead of terminating the program
- Write a function to get an element from a dictionary, if the Key is not in the dictionary, make the function return None. Write this function with and without a **try except**
- Write a program that exits with error code 3 if it can't open a file.

c. Classes and OOP

Object Oriented Programming is programmation paradigm based on classes and instances. Classes represent an idea or an element of the Program. If you do a library program, your classes can be Book and Pages. Instances are memory regions structured by the class. Each instance has its own memory region.
Python allows us to do object oriented programming. The keyword to define a class is simply **class.**

i. General OOP

Remember 2 things for any OOP program you design:

1. Regroup data in Classes, regroup common data in Parent classes. Avoid repeating data.
2. Always make a quick draft before coding. Split the program in multiple parts if necessary.

In python classes, the variable **self** refers to the current instance you are manipulating.

- You are tasked with the realisation of a Hotel Management system. Your program has to keep track of customers, rooms and their occupation planning.
    - Draft the different classes required for this system and their interactions.
    - Write your classes in Python
    - Add a menu to add a new room, a new customer and when they will be occupying the room.
    - Make the program read and write the current state of the hotel to a file, likewise, add a function to load the state from a file.

ii. Python classes and their secrets

Special functions can be used to make your programs more elegant and simpler to understand. You have already seen the special __init__ function which is the class constructor. You have other functions of this kind.

- Using the **__getitem__** and **__setitem__** functions, make a class similar to a dictionary but that returns None instead of throwing an error when the key is not found.
- Using the **__getitem__** function, write a class that returns the file contents of the given argument: a["file.txt"] returns the contents of file.txt or None if file.txt does not exist.
- Add a **__str__** function to a class and pass the class in print directly.
- Add a **__del__** method that prints Goodbye and delete all references to the class.
- Create a Matrix class, using **__getitem__ __setitem__** implement a way to access or set matrix elements. Using **__add__ __mul__** implement matrix multiplication and addition, throw an error if the matrices size do not match.
- What are the **__enter__** and **__exit__** functions used for ?

d. The Pythonic ways

Pythons has some specific syntax to help you solve complex problems and make your code cleaner.

i. Generators

Generators are functions that hold their internal state and can return multiple times using the keyword **yield**
To go to the next value, use the **next** built-in function or give it directly to a for loop.

- Write a generator that returns each character of a given string.
- Write a Fibonnaci number generator.
- Write a generator that returns each file in a given folder

ii. Ternary operator

The ternary operator will save you 3 lines in a classy way.

- Using the ternary operator, make the absolute value function.
- Using the ternary operator, make a function that return the log value of the argument or return None if the argument is <= 0

iii. List comprehension

List comprehension is a syntax to create a list without using append.
The syntax is **`[<expr> for <var> in <iter> (condition)]`**

- Write a list of squared numbers from 1 to 100
- Given a string, make a list of the length of each word in the string.
- Given a list of floats, make a list of integers from this list which contains only the positive numbers.

e. Numpy & Matplotlib, and other packages

Numpy is a very powerful library to do calculations:

- Use the **numpy.linspace** to create a numpy array of 100 numbers from 0 to 100 spaced by the same increment.
- Using the Matplotlib **plot** and **show** function, show the graphs of the square function, the inverse and log functions.
- Using the **multiprocessing.Pool** class, distribute the calculation of the nth prime number. You can use a function to determine if a given number is prime or not.
- Using the **imageio** module, read an example image and show it with matplotlib
- Print the shape of the numpy array representing the image, nullify the red channel and print the image again.
- Use numpy to generate a 440Hz sine wave, use the **simpleaudio** module to play this note. You will have to convert the sine wave to **int16** by first, scaling your [-1.0, 1.0] sine wave to [-32 768, 32 767] and then convert your data using **np.astype.** Then use simpleaudio.play_buffer. (Decide a sample rate, which is the number of points per second, before creating your sine wave a sample rate of 44100, 44kHz works well)
- Likewise you can use **sounddevice** module to record your microphone and use matplotlib to show what your voice sounds like.