TecnoCampus

u*pf.* **Universitat Pompeu Fabra** *Barcelona*

Operating Systems
Laboratory 1

Imanol Rojas, Dr. Pere Tuset-Peiró

{`irojas, ptuset`}@tecnocampus.cat

Grau en Enginyeria Informàtica de Gestió i Sistemes d'Informació

Escola Politècnica Superior TecnoCampus

Universitat Pompeu Fabra

Curs 2023-2024

Laboratory Overview

This first laboratory provides a comprehensive exploration of basic Linux commands, the structure of the Linux file system, programming in C, and the use of compilation and 'make' utilities. Key concepts include navigating and manipulating the Linux file system, understanding and utilizing essential Linux commands for file and directory operations and pipelines/background operations, and mastering the C programming language for creating a functional application in several progressive steps. Additionally, the laboratory explains the basics of the compilation process, emphasizing the creation and usage of Makefiles for efficient project management and software development in a Linux environment. This hands-on approach integrates theoretical knowledge with practical skills, essential for the next laboratory sessions of the Operating Systems course.

Installation of the GNU/Linux virtual machine

As previously introduced, in this course you will use the GNU/Linux operating system (Debian 12 Bookworm) as a working tool. Thus, the first step is to install it on your machine. To do this, you will use a virtualization environment, such as VMWare or VirtualBox, as described below[1].

The first step is to download the virtualization environment, which can be either VMWare or VirtualBox. In the case of VMWare, you need to go to the website and download the latest version (17.5 latest) of VMware Player. In the case of VirtualBox, go to the website and download the latest version of the program. In both cases, the installation process is very intuitive and should not present any complication, so the process is not detailed in this guide.

Next, it is necessary to use a virtual machine created specifically for this course. For this purpose, you will use a pre-configured virtual machine with the Debian 12 Bookworm distribution. This virtual machine can be downloaded from the following link. It is also available on the online campus of the course.

Once the virtualization environment is installed and the operating system image has been downloaded, you can proceed with its installation. The recommended environment for carrying out this process is VMWare Player, so the rest of the document describes the process of creating the virtual machine for this environment. For VirtualBox the procedure is similar, and you should not have any trouble completing it. If you find any problem during the process, please refer to the laboratory professor for help.

The first step is to open VMWare Player and create a new virtual machine. For this, choose the "*Open a Virtual Machine*" option, as shown in Figure 1.1a. Then select the virtual machine to open on the directory you extracted the VM zip file to (Figure 1.1b). Now you should have what you can see in Figure 1.1c.

Now, if you enter the VM and input the user/password combination user=`esupt` and password=`esupt`, you should be able to log in and use the machine.
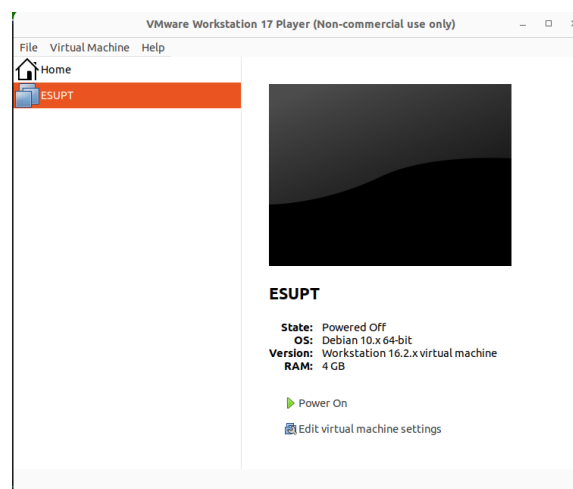
---

[1]You can **skip this step if you already have a Linux machine**.
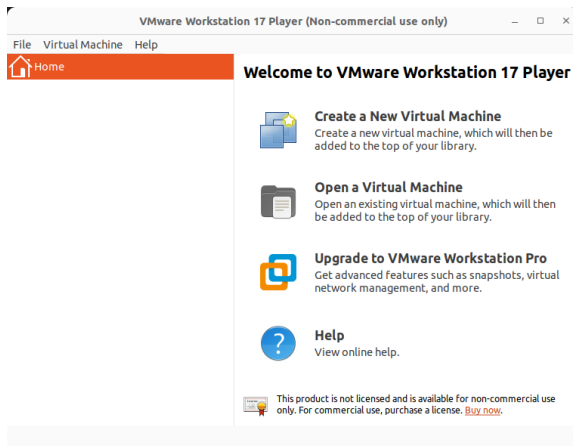
(a)

(b)

(c)

Figure 1.1: Opening the downloaded VM in VMware Player

Introduction to the GNU/Linux operating system

GNU/Linux is an open-source Unix-like operating system that combines the GNU tools and the Linux kernel, which was first released by Linus Torvalds on September 17, 1991. Unlike proprietary operating systems, like Windows or MacOS, GNU/Linux is released under the GNU General Public License (GPL), meaning it is freely distributable and modifiable by the users. Its key differentiating features include:

- **Open source**: Anyone can study, modify, and distribute the software.

- **Security**: Linux is known for its robust security, making it a popular choice for servers and sensitive environments.

- **Customizability**: High degree of customization possible due to its open-source nature.

- **Support**: Strong community resources and support, allowing users to learn and fix issues.

- **Free to use**: Most distributions of GNU/Linux can be downloaded, installed, and used freely.

GNU/Linux distributions, often known as "distros", are various flavors of the GNU/Linux operating system tailored to meet different needs. Each distribution has its unique package management system and community support, catering to different user preferences and use cases. Some popular distributions are:

- **Debian**: One of the first GNU/Linux distributions, praised for its stability and a large repository of packages.

- **Fedora**: Features cutting-edge technology and is used often by software developers.

- **CentOS**: A community-supported derivative of Red Hat Enterprise Linux, known for its reliability in server environments.

- **Arch Linux**: Designed for experienced users who prefer a do-it-yourself approach and a rolling release model.

- **Ubuntu**: A derivative of the Debian distribution, known for its user-friendliness and popularity among desktop users.

This introduction to the GNU/Linux operating system is structured to cover three fundamental aspects:

1. **File Path References**: Understanding how Linux handles file paths, including absolute and relative paths, and special notations like ~ and ..

2. **File System Structure**: Exploring the hierarchical file system of Linux, including key directories like /home, /etc, and /bin.

3. **Basic Commands**: An overview of basic Linux commands for file manipulation, directory navigation, and system monitoring.

Understanding file path references

On GNU/Linux operating systems, file paths can be specified in several ways, each serving different purposes:

- **Absolute Path**: Starts from the root directory (denoted by a leading slash '/') and specifies the complete path to a file or directory. For example, '/home/user/documents'.

- **Relative Path**: Specifies a path relative to the current directory. It does not begin with a '/'. For example, if you are in '/home/user', and you want to refer to a file in '/home/user/documents', you can just use 'documents/filename'.

- **Tilde (~)**: Represents the current user's home directory. For instance, ' /documents' refers to the 'documents' directory in the current user's home folder.

- **Dot (.)**: Represents the current directory. For example, './script.sh' would run 'script.sh' that's in the current directory.

- **Double Dot (..)**: Refers to the parent directory. For example, if you are in '/home/user/documents' and use '../pictures', it refers to '/home/user/pictures'.

GNU/Linux file-system structure

The GNU/Linux operating system (and UNIX operating systems in general) has a file system with a well-defined structure, starting with the root (/) and where each directory serves a specific purpose. This structure may slightly vary between distributions, but the core directories are common as seen in Figure 2.1:

- **/bin**: Essential user command binaries are stored here. Commands used in single-user mode and required for system functioning are placed in this directory.

- **/boot**: Contains bootloader files, including kernels, initrd images, and configuration files required during the boot process.
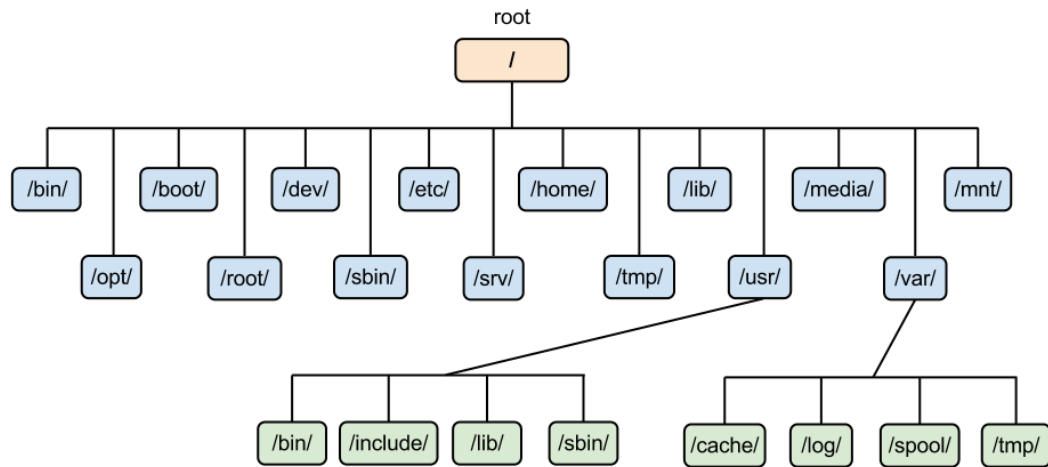
Figure 2.1: Basic file-system of GNU/Linux

- **/dev**: Device files, representing hardware components or other physical and virtual devices, are located here.

- **/etc**: System-wide configuration files and scripts are stored here. It contains configuration files that are local to the machine.

- **/home**: Personal directories for users are found here. Each user has a directory within /home for personal files, settings, etc.

- **/lib**: Essential shared libraries and kernel modules. Libraries needed by the binaries in /bin and /sbin are located here.

- **/media**: Removable media such as USB drives are typically mounted here.

- **/mnt**: Temporary mount points for mounting file systems temporarily.

- **/opt**: Optional or third-party software that does not conform to the standard file system hierarchy can be found here.

- **/root**: The home directory for the root user.

- **/sbin**: System administration binaries that are not necessarily needed by ordinary users, are placed here.

- **/srv**: Contains data served by the system, such as web pages and FTP files.

- **/tmp**: Temporary files (deleted at reboot) are placed in this directory.

- **/usr**:

    - /usr/bin: Contains the majority of user-executable binaries.

– /usr/sbin: Similar to /sbin, but for non-essential system binaries.

– /usr/lib: Libraries for the binaries in /usr/bin and /usr/sbin.

– /usr/local: Local hierarchy for manually installed software.

– /usr/share: Architecture-independent (shared) data.

- /**var**:

  – /var/log: Log files and directories.

  – /var/spool: Task spooler directories.

  – /var/lib: Variable state information.

  – /var/mail: User mailboxes.

  – /var/run: Information about the running system since last boot.

- /**proc**: A virtual file system providing process and kernel information as files.

- /**sys**: Another virtual file system, providing information about devices, drivers, and some kernel features.

Essential GNU/Linux commands

This section provides detailed explanations of commonly used Linux commands, highlighting their functionalities and differences.

Please notice that for every command, you can use the −−help to obtain a detailed list of arguments that can be used to modify the program behavior. For example, you can run mkdir −−help to obtain the help menu of the mkdir command. In addition, you can also use the man command to obtain the manual of the operating system commands and libraries. For example, you can use the man chmod to obtain the reference of the chmod command.

Directory Operations

mkdir : Creates a new directory.

mv : Moves or renames directories.

cp : Copies directories or files.

rmdir : Removes empty directories.

ls : Lists contents of a directory.

pwd : Prints the current working directory.

Figure 2.2: Using the −−**help** argument to obtain the list of arguments that a program supports.

Figure 2.3: Using the **man** command to obtain the manual for a given program.

File Operations

touch : Creates a new file or updates the timestamp of an existing file.

mv : Moves or renames files.

cp : Copies files.

rm : Removes files or directories.

Displaying File Contents

`cat` : Displays the entire content of a file.

`more` : Paginates file content for easier reading.

`tail` : Displays the last part of a file.

`head` : Displays the first part of a file.

File Editing

`vi` : A powerful text editor.

`nano` : A simpler, easier-to-use text editor.

Process Management

`ps` : Displays information about active processes.

`top` : Displays real-time information about running processes.

`kill` : Terminates a process.

File Compression and Decompression

`zip` : Compresses files into a zip archive.

`tar` : Used for compressing and archiving files.

System Help

`man` : Displays the manual page for commands.

Ownership and Permissions

`chown` : Changes the owner of a file or directory.

`chmod` : Changes file permissions.

Pipes and Background Execution

`|` : The pipe symbol, used to pass the output of one command as input to another.

`&` : Runs a command in the background, allowing the user to continue other tasks without waiting.

`\` : In command-line inputs, the backslash ($\backslash$) character is used as a line continuation character (or an escape character). It allows long commands to be broken into multiple lines for improved readability.

`bg` : Resumes suspended jobs by running them in the background.

`fg` : Brings a job to the foreground, allowing the user to interact with it.

User and Group Management

    `who` : Lists users currently logged in.

`whoami` : Displays the current user's name.

Symbolic Links

    `ln` : Creates hard or symbolic links.

Searching in Text Files

`grep` : Searches for patterns within files.

Counting Lines, Words, and Characters

    `wc` : Counts lines, words, and characters in a file.

Downloading Files from the Internet

`wget` : Downloads files from the web.

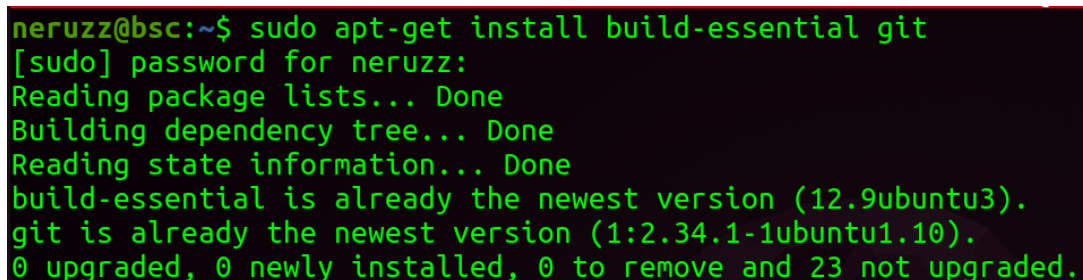# Installation and verification of the working environment

Once the GNU/Linux operating system is working in your computer, you can install `gcc` and `git`. The former is a compiler responsible for generating the binary file that runs on the processor from the source code. The latter is a widely used version control system that allows us to work with the source code.

Installation of `git` and `gcc`

To ensure that both `gcc` and `git` are installed on your system, open a terminal and execute the following command:

```
1  sudo apt-get install build-essential git
```

After entering the password, the installation will proceed, or it will indicate that either package is already installed on the system.

Figure 3.1: Installation of `Git`, a version control system.

Once both packages are installed, you can see which version is available on our system by opening a terminal and executing the following command:

```
1  gcc --version
```
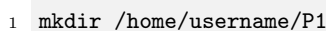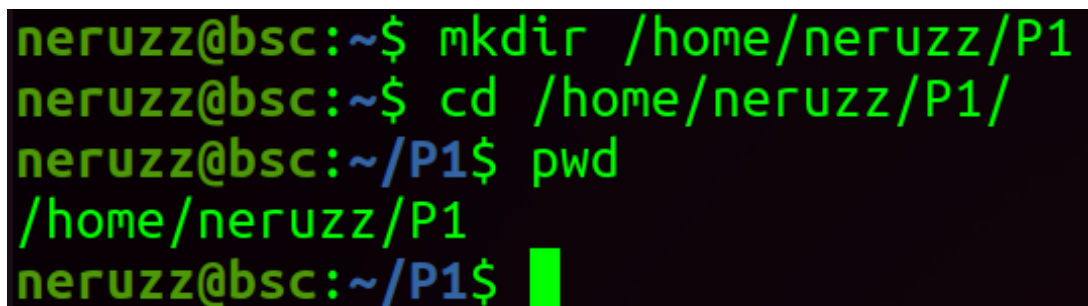
The response will depend on the installed version.

Figure 3.2: Version of `gcc`

Obtaining the source code
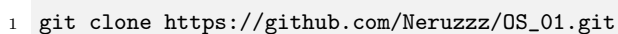
After installing the packages, you can proceed to obtain the repository with the source code of examples of the course, which is located at `https://www.github.com`.

First, you need to create a folder named `P1`, where you will keep the different working files. To do this, open a terminal and execute the following command[1]:

```
1  mkdir /home/username/P1
```


Figure 3.3: Creating reference directory and checking it exists.

Next, we will clone the `GitHub` repository of this lab with the command

```
1  git clone https://github.com/Neruzzz/OS_01.git
```

As it is a public repository, we will not be asked for access credentials, but if it were a private repository, you would be requested to enter the username and password.

Once the repository is cloned, you can enter the directory and see its contents with the following commands:

---

[1]Use your own username on the command

```
1  cd os_01
2  ls -la
```

The folder contains different files that we will comment on later. Finally, you can print the file permissions and its name (without any additional information) using the following command:

```
1  ls -laF | awk '{print $1,$9}'
```

Building the *Hello world!* program

Once we have the source code, we can start using `gcc` with the `Hello World!` example, which will be recorded in a text file named `hello1.c`. As seen in Listing 1, the program prints on screen the message `Hello World 1!` and ends its execution in an orderly manner (returning `0`).

```
1   /*
2    * Filename: hello1.c
3    */
4
5   #include <stdio.h>
6
7   int main(int argc, char* argv[])
8   {
9       printf("Hello world 1!\n");
10
11      return 0;
12  }
```

Llista 1: Source code of the file `hello1.c`

To compile the file, we can use the command

```
1  gcc hello1.c
```

and if the result is successful (there are no problems in the compilation process) we will obtain an executable file named `a.out`. It is important to note that in GNU/Linux operating systems the file extension (in this case `.out`) is irrelevant to determine whether it is a program or another type of file. In fact, the ability to execute a file is based on the properties of the file, so most programs in a GNU/Linux operating system do not have an extension. On the other hand, in Windows operating systems, most programs have an extension (`.exe`).

We also remember that the input parameters of the program are defined with the `argv` and `argc` variables of the `main()` function, as shown in line 7. On the one hand, `argc` will be an integer value with the number of arguments passed to the program execution. On the other hand, `argv` will be a vector of pointers to the character strings (*string*) that represent the different parameters passed to the program execution. In this way, the parameter `argv[0]` will always be the name of the program as we have introduced it in the command line, while from `argv[1]` to `argv[argc-1]` we will have the rest of the parameters.

Once successfully compiled, we can execute the program by writing the command

```
1  ./a.out
```

As seen in Figure 3.4, the output of the program is `Hello World 1!`. It is important to note that the prefix `./` has been included before the name of the program to indicate that it is in the same directory. To avoid this, the current directory could be included in the `PATH` variable of the console, but this could lead to problems in searching for system programs and is not recommended. Therefore, it is a good habit to always indicate that we are executing the file from the current directory using the prefix `./` before the name of the program.
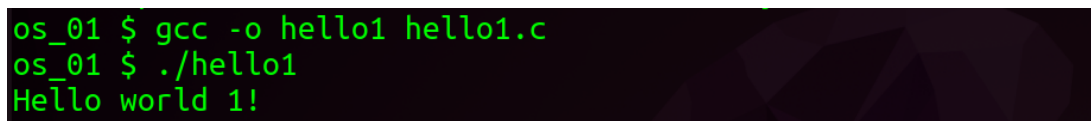
Figure 3.4: Execution of the example program `a.out`.

While the default name of a program compiled with `gcc` is `a.out`, we could define its name at the time of compilation (it is usual) using the command

```
1  gcc -o hello1 hello1.c
```

where the `-o` parameter indicates the name of the output file. As seen in Figure 3.5, the result is that our program is now called `hello1`, and we can execute it with the command

```
1  ./hello1
```

Figure 3.5: Execution of the example program `hello1.c` compiled with the name `hello1`.

Building C programs

The process of developing a program using the C programming language is complex and full of errors inherent to the human condition. For example, misspelling the name of a variable or forgetting a ; at the end of a line. In light of this, it is a good habit to compile the program frequently (to detect possible errors incrementally) and also to activate the compiler's warning options. This can be done by activating the `-Wall` or `-Werror` parameters. The first activates all compilation warning messages, thereby facilitating the detection of errors made by the programmer. The second turns compilation warning messages into errors, ensuring that the programmer corrects them before generating the program. Although it can be bothersome, using `-Werror` maximizes the likelihood that the program will function correctly once it is running, so its use is recommended.

To test the functionality of these parameters, we will add the declaration of an unused integer variable in our program, as shown in Listing 2. As we can see in Figure 4.1, with the `-Wall` option activated, the compiler warns us that the variable `i` is declared on line 9 but is not used throughout the program.

```
1  /*
2   * Filename: hello2.c
3   */
4
5  #include <stdio.h>
6
7  int main(int argc, char* argv[])
8  {
9      int i;
10
11     printf("Hello world 2!\n");
12
13     return 0;
14 }
```

Llista 2: Source code of the file `hello2.c`

Now we know how to compile a program written in the C programming language using the `gcc` compiler in a GNU/Linux system. However, most programs are not composed of a single file,

Figure 4.1: Compilation of the example program `hello2.c` with `-Wall`.

but are formed by different modules (compilation units) that are compiled independently and then linked together to form the final executable program. In addition, most C programs have inclusions (directives `include`) and definitions (directives `define`) that must be resolved before the compilation process of each compilation unit. Thus, the process from when we have our source code until we can execute it could be decomposed into the phases shown in Figure 4.2 and are described below.
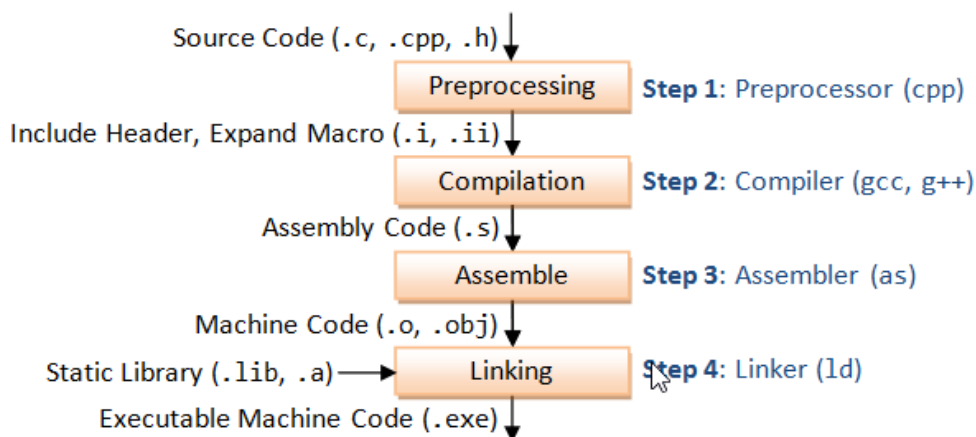
Figure 4.2: Compilation process of a C program.

Step 1: The Preprocessor

The preprocessor is responsible for incorporating header files through the `include` directive (files with `.h` extension), expanding possible macros in our code through the `define` directive (for example, constant values), and removing user comments (as they are not interpreted by the compiler). We can generate the intermediate file produced by the preprocessor with the following command, where `hello2.i` will be the output file.
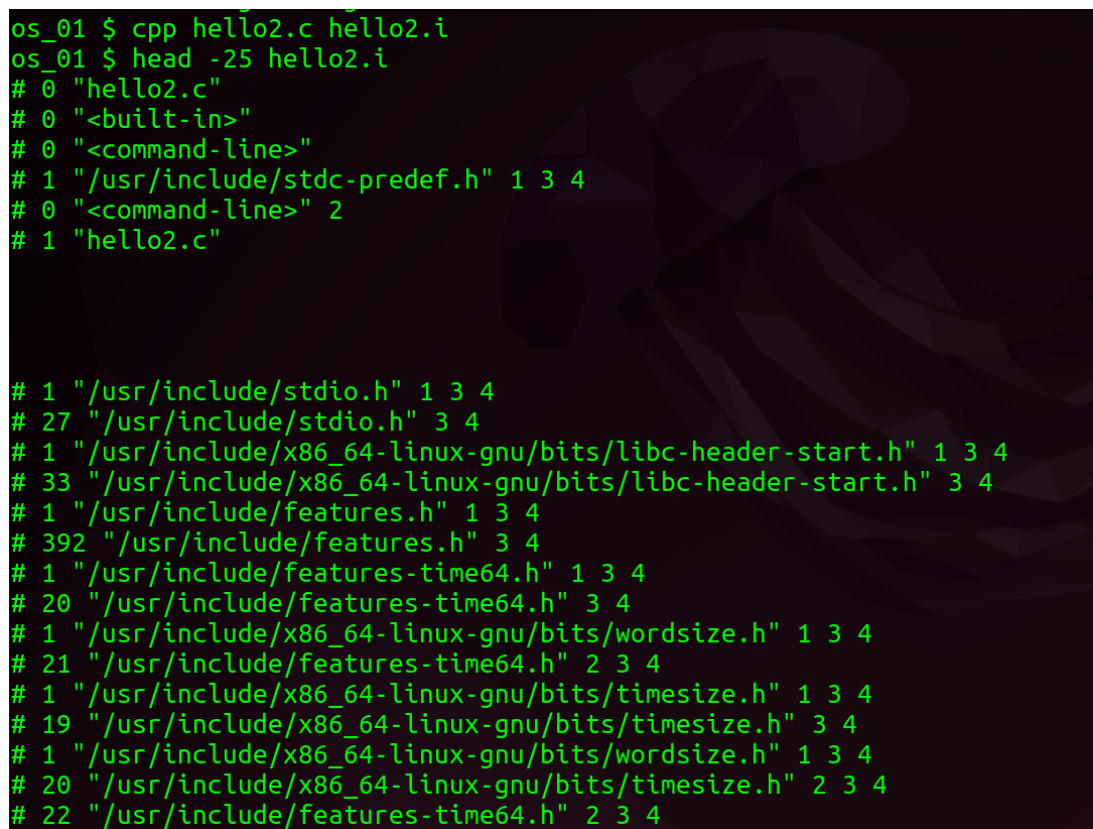
```
1  cpp hello2.c hello2.i
```

You can view the content of the expanded file `hello2.i` with the following command:

```
1  cat hello2.i
```

To display only the first 25 lines of the output (for easier reading) you can use the following command:

```
1  head -25 hello2.i
```


Figure 4.3: First 25 lines of the preprocessed file result.

You can also display the last 25 lines of the same file with the following command:

```
1  tail -25 hello2.i
```

As you can see, the directive `#include <stdio.h>` has incorporated a large number of lines, including library paths and function definitions.

```
os_01 $ tail -25 hello2.i
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));


extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;


extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
# 885 "/usr/include/stdio.h" 3 4
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);
# 902 "/usr/include/stdio.h" 3 4

# 6 "hello2.c" 2


# 7 "hello2.c"
int main(int argc, char* argv[])
{
    int i;

    printf("Hello world 2!\n");

    return 0;
}
```

Figure 4.4: Last 25 lines of the preprocessed file result.

Step 2: The Compiler

From the output of the preprocessor, the compiler generates assembler code for the architecture of the processor that your computer uses (i.e., x86, x64 or ARM). To generate the assembler code from the preprocessor output, you can use the following command:

```
1  gcc -S hello2.i
```

As the output of the preprocessor is not very useful for the user, you can perform the compilation process directly from the original file using the following command:

```
1  gcc -S hello2.c
```

In this case, the compiler will call the preprocessor transparently.

```
os_01 $ gcc -S hello2.i
os_01 $ gcc -S hello2.c
```

Figure 4.5: Assembler code result of our example.

The result of the compilation process is assembler code, as shown in Listing 3. As you can

```
1    .file "hello2.c"
2    .text
3    .section .rodata
4   .LC0:
5    .string "Hello world 2!"
6    .text
7    .globl main
8    .type main, @function
9   main:
10  .LFB0:
11   .cfi_startproc
12   pushq %rbp
13   .cfi_def_cfa_offset 16
14   .cfi_offset 6, -16
15   movq %rsp, %rbp
16   .cfi_def_cfa_register 6
17   subq $16, %rsp
18   movl %edi, -4(%rbp)
19   movq %rsi, -16(%rbp)
20   movl $.LC0, %edi
21   call puts
22   movl $0, %eax
23   leave
24   .cfi_def_cfa 7, 8
25   ret
26   .cfi_endproc
27  .LFE0:
28   .size main, .-main
29   .ident "GCC: (GNU) 9.1.0"
30   .section .note.GNU-stack,"",@progbits
```

Llista 3: Assembler code of the file `hello2.s`.

see, the high-level code has been converted into a set of assembler instructions for a generic architecture.

Step 3: The Assembler

The assembler is responsible for converting assembler code into the machine language of our processor's architecture. That is, it converts the sequential set of instructions (and their respective parameters) into instructions that can be executed by the processor.
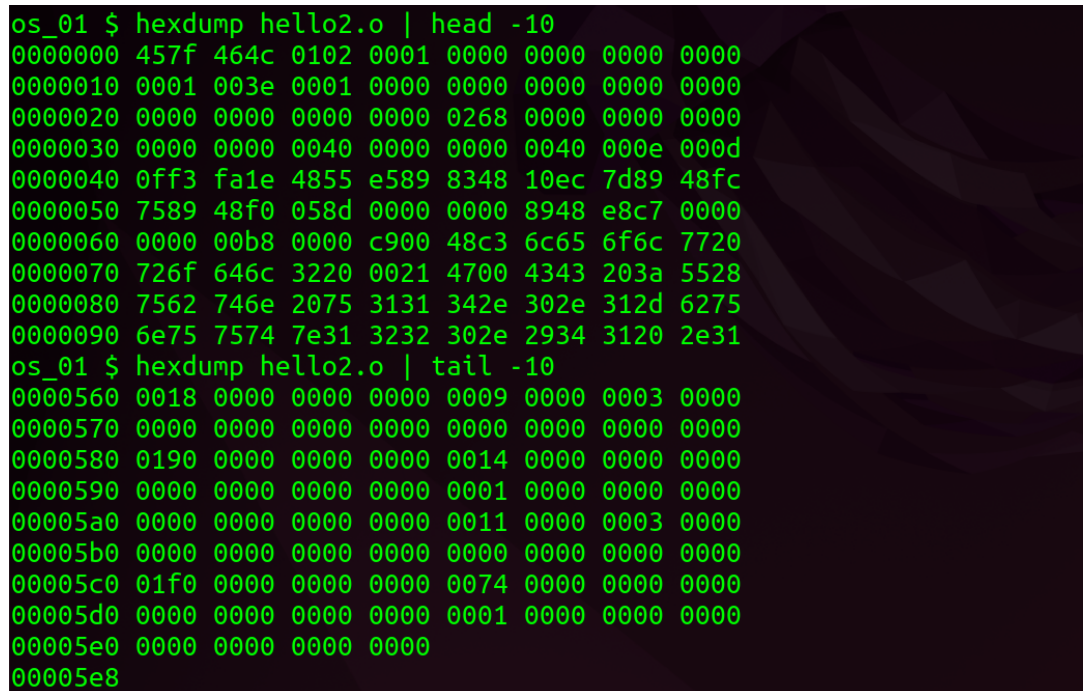
Just like in the case of the preprocessor, the intermediate output is not very useful for people, so we can also use the assembling tool. For this, you use the following command:

```
1   as hello2.s -o hello2.o
```

The output of the assembler is a file that is no longer editable with a normal text editor, as

it may contain unprintable screen information (ASCII codes below `0x20`). Therefore, the file `hello2.o` must be viewed with an hexadecimal editor like `hexdump`, as shown in Figure 4.6. For this, you use the following command:

```
hexdump hello2.o
```


Figure 4.6: Result of the assembly process of our example. First and last 25 lines.

If the command is not found on the system, it can be installed using the following command:

```
apt-get install bsdmainutils
```

Step 4: The Linker

Finally, the linker takes the machine language code along with the libraries to produce the executable file for the system. For this, you can use the following command:

```
ld -o hello2 hello2.o -lc --entry main
```

To avoid errors that prevent us from generating the output file, you must link the program with the standard C programming language library for the operating system used (parameter `-Lc`)

and indicate the entry point in program execution (parameter `-entry main`). Otherwise, the linker will indicate that it cannot find references to the functions used or that it cannot find the execution entry point, as shown in Figure 4.7.

Figure 4.7: Linking of our example.

Once the program is linked, you can see the libraries used to link our executable using the following command:

```
1  ldd hello2
```


Figure 4.8: List of libraries of our example.

As you can see in Figure 4.8, in this case the program has been linked with the library `linux-vdso.so.1`, `libc.so.6`, and `ld64.so.1`. The first is a (dynamic) library that takes care of automatically mapping the address space of the program to increase the performance of system calls. The second is the standard library of the C programming language for the operating system used, in our case, GNU/Linux. The third is the library that takes care of performing the dynamic linking required by the program at the time of its execution, loading it into memory, and executing.

Of course, you can perform all four steps (preprocessing, compilation, assembly, and linking) in a chained and transparent way for the user using the following command:

```
1  gcc -o hello2 hello2.c
```

As in the previous case, this will generate an executable file named `hello2`.

Once the program is compiled, you can use the operating system tools to analyze the type of the different files generated, as you can see in Figure 4.9. By executing the `file` command for each file, we can see that `hello2.i` is a text (ASCII) file containing C source code, and that

`hello2.s` is a text (ASCII) file containing assembler source code. On the other hand, `hello2.o` is an object file pending linking and `hello2` is an executable file for the `x86-64` architecture.

Figure 4.9: Information about the different files in the compilation process of our example.

Furthermore, you can also use the `nm` utility to have the table of symbols referenced in our program. In the case of the `hello2.o` file, as shown in Figure 4.10, you can see that the only symbols are `main` and `puts`.

Figure 4.10: Symbols referenced in `hello2.o`.

In contrast, for the executable file `hello2`, as you can see in Figure 4.11, the list of symbols and functions increases due to the linking process.

Figure 4.11: Symbols referenced in `hello2`.

Automation of the Compilation Process with `make`

The process of compiling a program in C language is quite tedious and prone to human errors due to all the steps that must be performed for each file containing source code. To solve this problem, there are numerous tools dedicated to automating the compilation process and generating the resulting executable file. For example, we have tools like `CMake` and `SCons` that are cross-platform. However, for small projects on the GNU/Linux operating system, the most common is to use the `make` tool and its configuration files `Makefile`, as described below.

In Listing 4, you can see the `Makefile` configuration file for the `hello2` program, which we have renamed as `hello2.Makefile`. As you can see, the file consists of rules (*targets*), which have
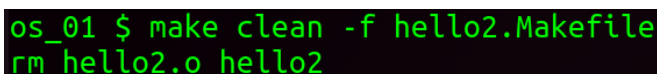
dependencies and a set of orders to execute. As expected, for a specific rule, all the rules to meet the dependencies are first executed, and then the orders associated with the same rule are executed. For example, the `all` rule depends on the `hello2` file, so this dependency is attempted to be resolved first. The `hello2` rule depends on the `hello2.o` file, so the `hello2.o` rule is executed, which depends on the `hello2.c` file. Since the `hello2.c` file already exists, the command associated with the rule is executed, in this case, `gcc -c hello2.c`. This command generates the `hello2.o` file, so the `hello2` rule is satisfied, and the command associated with the rule is executed, in this case, `gcc -o hello2 hello2.o`.

```
1  all: hello2
2
3  hello2: hello2.o
4   gcc -o hello2 hello2.o
5
6  hello2.o: hello2.c
7   gcc -c hello2.c
8
9  clean:
10   rm hello2.o hello2
```

Llista 4: Content of the `hello2.Makefile` file.

A common *target* of a `Makefile` is `clean`, which is responsible for removing all intermediate files generated by the compilation process. As you can see in Figure 4.12, the *target* `clean` has no dependencies and executes the command `rm hello2.o hello2`, thereby eliminating the files `hello2.o` and `hello2` that were generated during the compilation process of the program.
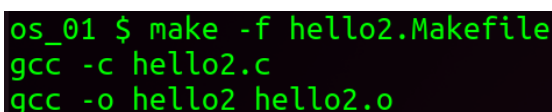
Figure 4.12: Removal of intermediate files in our example, to regenerate them.

If you now run the `make` command again, it will execute (or re-execute) all the orders described in the `hello2.Makefile` file in the correct order to generate the `hello2` program.

As you can see in Figure 4.13, this process is executed recursively until all dependencies are met and all orders are executed, ensuring that the program is updated.
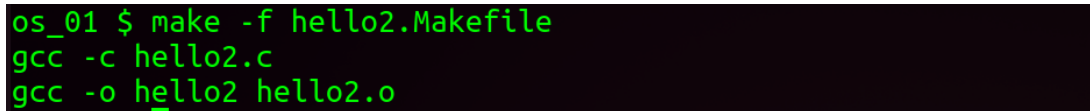
Figure 4.13: Generation of the output file of our example.

It is important to remember that if a *target* is not specified, the `make` program runs with the

all target by default. If you have compiled earlier and have not updated any file, it indicates the message that everything is updated, as we see in Figure 4.14.
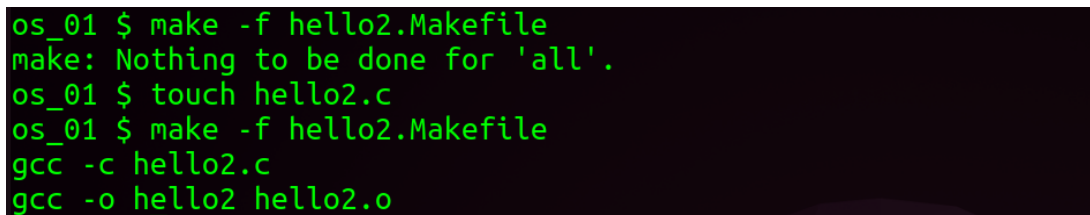
Figure 4.14: Generation of the output file of our example, without updating any component.

It is important to keep in mind that to decide whether a file has been modified and needs to be recompiled, the `make` utility is based on the modification time of the file itself. Thus, if we pretend to update the file using the following command:

```
1  touch hello2.c
```

and then run the `make` command again, we see how it recompiles and regenerates the `hello2` program, as we see in Figure 4.15.

Figure 4.15: Updating the file, to recompile.

Now that you are familiar with the process of compiling a C program and its automation using the `make` utility, let's take a look at a slightly more complex example. For this purpose, we will use the previous example, but we will remove the declared variable again, as seen in Listing 5.

The first step is to generalize the compilation *targets*, as shown in Listing 6, so they do not depend on the file name. As we see, the compilation orders take values from the label (`$@`) and the compilation requirements (`$<`).

Thus, if you execute the following command:

```
1  make -f hello3.Makefile
```
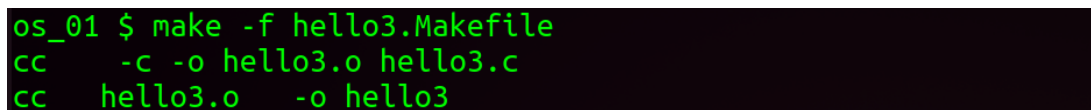
you can see in Figure 4.16 how the different files are compiled, and the program is linked successfully.

```
1  /*
2   * Filename: hello3.c
3   */
4
5  #include <stdio.h>
6
7  int main(int argc, char* argv[])
8  {
9      printf("Hello world 3!\n");
10
11     return 0;
12 }
```

Llista 5: Source code of the file `hello3.c`.

```
1  all: hello3
2
3  hello2: hello3.o
4   gcc -o $@ $<
5
6  hello2.o: hello3.c
7   gcc -c $<
8
9  clean:
10  rm hello3.o hello3
```

Llista 6: File `hello3.Makefile`.

Figure 4.16: Generation of the file `hello3`.

But you can still go a step further to make our `Makefile` configuration file more generic. For this, you can define in the Code 7 some `make` variables that represent the name of the program (`PROGRAM_NAME`) and the object files (`hello4.o`) necessary to generate it. These variables are substituted before starting the compilation process, so that the same result is achieved as in the previous case.

As you can see in Figure 4.17, if you execute the following command:
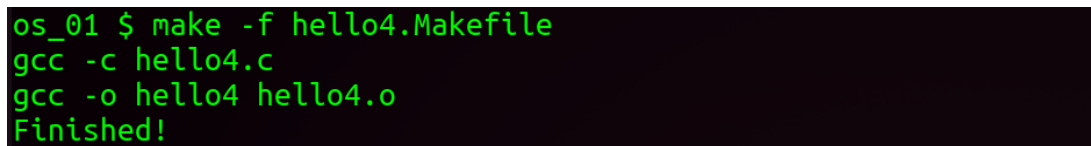
```
1  make -f hello4.Makefile
```

you can observe how the different files are compiled, and the program is linked successfully, just

```
1  PROGRAM_NAME = hello4
2  PROGRAM_OBJS = hello4.o
3
4  REBUIDABLES = $(PROGRAM_OBJS) $(PROGRAM_NAME)
5
6  all: $(PROGRAM_NAME)
7   @echo "Finished!"
8
9  $(PROGRAM_NAME): $(PROGRAM_OBJS)
10  gcc -o $@ $<
11
12 %.o: %.c
13  gcc -c $<
14
15 clean:
16  rm -f $(REBUIDABLES)
17  @echo "Clean done"
```

Llista 7: File `hello4.Makefile`.

like in the previous case.

Figure 4.17: Generation of the file `hello4`.

Finally, you can separate the source file `.c` into two files named `hello5a.c` and `hello5b.c`. As seen in Code 8, the first file contains the `main` function, while in Code 9 the file contains the `printer` function. In addition, the header files named `hello5a.h` and `hello5b.h` have also been separated, as seen in Code 10 and Code 11, respectively.

```
1  /*
2   * Filename: hello5a.c
3   */
4
5  #include "hello5a.h"
6  #include "hello5b.h"
7
8  int main(int argc, char* argv[])
9  {
10     return printer();
11 }
```

Llista 8: File `hello5a.c`.

```
1  /*
2   * Filename: hello5b.c
3   */
4
5  #include "hello5b.h"
6
7  int printer(void)
8  {
9      printf("Hello world 5!\n");
10     return 0;
11 }
```

Llista 9: File `hello5b.c`.

```
1  /*
2   * Filename: hello5a.h
3   */
4
5  #include <stdio.h>
6  #include <stdlib.h>
```

Llista 10: File `hello5a.h`.

```
1  /*
2   * Filename: hello5b.h
3   */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7
8  int printer(void);
```

Llista 11: File `hello5b.h`.

As shown in Code 12, the variable `$(PROGRAM_NAME)` represents the name of the program to be built and the variable `$(PROGRAM_OBJS)` represents the different objects that need to be generated in order to link the program `hello5`. From there, the different *targets* of compilation are defined. The *target* `all` depends on the program, but does not perform any action, as this is handled by the *targets* `%.o` and `$(PROGRAM_NAME)` respectively. Thus, the first takes care of compiling all the `.c` files into `.o` objects, while the second takes care of linking all the `.o` objects indicated by the `$(PROGRAM_OBJS)` variable into a program with the name defined by the `$(PROGRAM_NAME)` variable. In this way, the program `hello5` is generated from the compilation and linking of the different modules that compose it.

```
1  PROGRAM_NAME = hello5
2  PROGRAM_OBJS = hello5b.o hello5a.o
3
4  REBUIDABLES = $(PROGRAM_OBJS) $(PROGRAM_NAME)
5
6  all: $(PROGRAM_NAME)
7   @echo "Finished!"
8
9  $(PROGRAM_NAME): $(PROGRAM_OBJS)
10  gcc -Wall -Werror -o $@ $^ -I ./
11
12 %.o: %.c
13  gcc -Wall -Werror -c $< -I ./
14
15 clean:
16  rm -f $(REBUIDABLES)
17  @echo "Clean done"
```
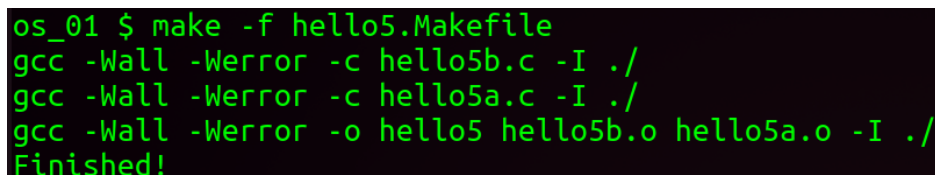
Llista 12: File `hello5.Makefile`.

It is also important to note that the parameter `-I ./` is used to indicate the directories where the compiler will look for header files (`.h`) to resolve the `include` directives during pre-processing. In this case, the header files are in the same directory, but if not, it would need to be explicitly indicated. This is usually done using relative paths (i.e., relative to the root directory of the program) to ensure that the compilation process works even if the files are in another directory.

As you can see in Figure 4.18, if you execute the following command:

```
1  make -f hello5.Makefile
```

the program `hello5` is generated from the compilation of the different compilation units (files `hello5a.c` and `hello5b.c`) that compose it. The result of the compilation is successful, and the program `hello5` functions as expected. In this new configuration file, the option `$<` has been changed to `$^` to ensure that all files in the list are processed.

Figure 4.18: Generation of the file `hello5`.

As seen in the different examples, the messages resulting from the `echo` command are preceded by a character `@` to prevent them from being printed twice.

Exercises (Part 1)

File path references

1. Create a file named 'testfile.txt' in your home directory. Navigate to another directory and use an absolute path to list the details of 'testfile.txt'.

2. While in your home directory, create a new directory 'myfolder'. Without changing your current directory, create a new file 'myfile.txt' inside 'myfolder' using relative path.

3. Use the tilde (~) to navigate to your home directory from any other location in the file system.

4. Create a script named 'runme.sh' in your current directory. Use the dot (.) notation to execute this script without adding its directory to the PATH.

5. Move to a subdirectory in your home folder. Use the double dot (..) notation to move up two levels in the directory structure and then list the contents of that directory.

Linux basic commands

1. **Create and Navigate**: Create a directory named `my_directory`, enter it, and display your current location with `pwd`.

2. **File Manipulation**: Inside `my_directory`, create an empty file named `file1.txt`, copy this file to `file2.txt`, and then list the files in the directory.

3. **Output Redirection**: Run `ls -l > listing.txt` to save the list of files in `listing.txt`. Then use `cat` to display the content of `listing.txt`.

4. **Using Pipes**: Use `grep` to find lines containing the word "root" in `/etc/passwd`, and use `wc -l` to count how many lines there are.

5. **File Editing**: Open `file1.txt` with `nano` or `vi`, write some text, save it, and then display its content with `cat`.

6. **File Searching**: Use `find` in your home directory to search for files ending in `.txt`.

7. **Compressing and Decompressing**: Compress `my_directory` into a file named `my_directory.tar.gz` and then decompress it into a different directory.

8. **Background Processes**: Run `top`, then press `Ctrl+Z` to stop it, and use `bg` to continue running it in the background.

9. **Permission Management**: Create a file `test_permissions.txt`, change its permissions to read-only, and then try to write to it with `nano`.

10. **Using `head` and `tail`**: Show the first 10 and the last 10 lines of `/etc/passwd`.

11. **Symbolic Links**: Create a symbolic link to `listing.txt` named `link_listing`.

12. **Paged Viewing**: Use `more` or `less` to view the contents of `/var/log/syslog`.

13. **Advanced Use of Pipes**: Use `ps -ef | grep root` to display all processes running as the root user.

14. **Changing Ownership**: Change the owner of `my_directory` to another user on your system (superuser permissions required).

15. **Downloading Files**: Use `wget` to download a file from the Internet directly to your current directory.

Exercises (Part 2)

Below are presented four programming exercises that must be solved using the C programming language and the GNU/Linux operating system.

For each exercise/program, the following must be delivered:

- the source code of the program,

- a file for the `make` utility that allows compiling each of the activities,

- relevant explanations of the code. It is not only requested to incorporate the code into the report, with the comments that the incorporated code may have, but also a written explanation,

- instructions on how to generate the executable file (adding a screenshot of how you have executed it on your system, where the correct generation of the same can be seen),

- instructions on how to execute the program, and screenshots of some executions where the correct functioning of it can be verified, for the use cases indicated in the statement.

Context

In the Galton machine, each ball that enters the machine always faces a binary decision (fall to the left or fall to the right) and each level of the machine has one more possible path than the previous one (on the first level the ball can pass through two different places, on the second level it can pass through three places, etc.). After passing through all the levels of the machine, the ball arrives at one container or another, following a normal or Gaussian bell distribution.

In this activity, we will program a virtual version of a device similar to the Galton machine.

Exercise 1

Write a program `ex1` that generates 1000 random integers following a uniform probability distribution. All the generated numbers must be between 0 and 10 (both included).

The program must count how many times each number is generated and display the results of this count on the screen when it finishes execution.

Figure 6.1: Example of a Galton machine.

Due to the characteristics of the uniform probability distribution, in all executions of the program, similar results should appear, in which all the numbers have come out approximately the same number of times.

Exercise 2

Write a program `ex2` that simulates a single-level Galton machine, in which there are $n = 11$ different containers from which each ball can fall, and in which the probability that each ball falls into each container is conditioned by the point where the ball is introduced into the machine, as follows:

1. Number, starting from 0 and ending at 10, all the containers.

2. To each container i, assign a weight w_i following the following formula:

   $$w_i = \max(0, n - 2 \times (\text{distance between } i \text{ and the entry point into the machine})) \quad (6.1)$$

   Where the distance is the difference between the number assigned to the container and the number assigned to the entry point (for example, if the ball enters at point 5, container 5 will be at distance 0, while containers 3 and 7 will be at distance 2).

3. Then, the probability that the ball falls into container i is equal to the weight w_i assigned to that container divided by the sum of the weights of all the containers:

   $$p_i = \frac{w_i}{\sum_{j=1}^{n} w_j} \quad (6.2)$$

The program must throw 1000 balls, making them all enter at point 5, count how many balls fall into each container, and display the results of this count on the screen.

Given that this probability distribution is different from the previous exercise, the results should

also show a different pattern.

Exercise 3

Modify the previous program to make a new program `ex3` in which there are $m = 10$ levels.

As before, the program will throw 1000 balls that will all enter the first level at point 5, but for each of the following levels, it will be necessary to recalculate the probabilities of where the ball can fall based on the result of the previous level (for example, a ball that falls towards container 3 at the first level will have 3 as the entry point for the second level, and therefore will have a different probability distribution than another ball that has fallen towards container 9, which will have 9 as the entry point).

The program must simulate that the 1000 balls cross the 10 levels, will count how many balls end up in each container, and will display the results.

This program can be approached in two different ways and both are valid:

- Throw all the balls at the first level, calculate where each one falls at this first level, and then iterate through levels, taking into account the results of the previous level.

- Throw a ball at the first level, make it pass through all the levels until it reaches a final container, and then iterate for all the balls.

Exercise 4

Modify the previous program to make a new program `ex4` that receives 4 parameters, so the program call must be `./ex4 NumBalls NumSlots NumLevels FirstSlot`.

To begin with, the program must check that the number of parameters entered is correct and display an error message and end its execution in case it is not.

Then, the program must simulate that it throws `NumBalls` balls, that at each level there are `NumSlots` paths or containers, that there are `NumLevels` levels in the machine, and that the entry point to the machine is `FirstSlot`, make the count of how many balls fall into each container and display the results.