Laboratory Documentation

2021-2022 Q1

This document contains the sessions that have to be done in the laboratory lectures. The sessions include a previous work and the work that has to be done during the session.

SO Professors-AC Department

Index of sessions

Session 1: The command-line interpreter: shell

Previous work

1. Objectives

The objective of this session is learning to get along with the laboratory work environment. We will see some operations that can be done either with the command-line or using the window manager. We will center in the practice of some basic commands and the usage of the online manual (man) that you will find in all the Linux machines.

2. Skills

- Be able to use man pages.
- Be able to use the basic system commands to modify/navigate around the file system: cd, ls, mkdir, cp, rm, rmdir, mv.
- Know the special directories ".." and "..".
- Be able to use the basic commands and programs of the system to access files: less, cat, grep, gedit (or another editor).
- Be able to modify the access permissions of a file.
- Be able to consult/modify/define an environment variable.
- Be able to use some special characters of the shell (command-line interpreter):
    - & to execute a program in the background.
    - > to store the output of a program (redirecting the output).

3. Previous knowledge

This session doesn't require previous knowledge.

4. Access to the system

In the labs we have an Ubuntu 20.04 LTS 64 bits installed. We have several users created in order to make some testings that involve several users. The usernames of the users are:"alumne", "so1", "so2", "so3", "so4" and "so5". The password is "sistemes" for all of them.

To start, we will execute what we call a *Shell* or a command-line interpreter. A *Shell* is a program that the OS offers us to work in an interactive text mode. This environment could seem less intuitive than a graphical environment, but it is really simple and powerful.

There are several command-line interpreters in the market, in the lab you will use Bash (GNU-Bourne Shell), but in general we will refer to it as Shell. Most of the things that we will

explain in this session can be consulted through the Bash manual (executing the command **man bash**).

To execute a Shell we only need to execute the "Terminal" application. With this application, it opens a new window (similar to the one in the image) where a new Shell is executed.
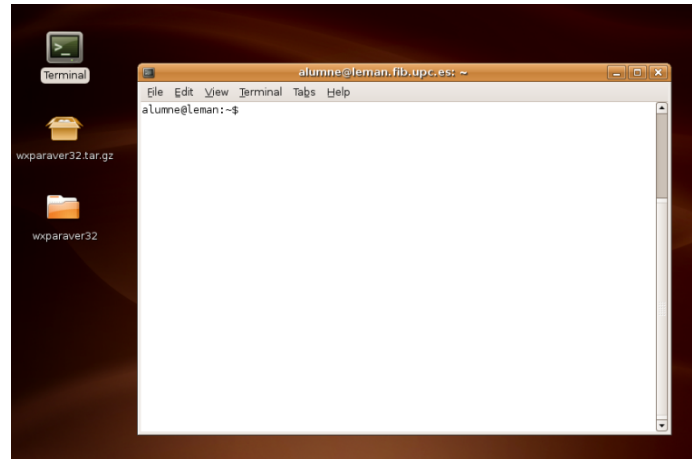

Figure 1 Shell window

The text that appears next to the blinking cursor is what is known as prompt and it is used to indicate that the Shell is ready to receive new orders or commands. Note: in the laboratory documentation we will use the # character to represent the prompt and indicate that what comes next is a command line (to try the line you DO NOT NEED TO WRITE #, only the command that appears next to it).

The Shell code could be summarized as the following:

```
while(1){
    command=read_command();
    execute_command(command);
}
```

There are two types of commands: internal commands and external commands. The **external commands** are any program installed in the machine and the **internal commands** are functions implemented by the command-line interpreter (each interpreter implements its own, some of them common and some of them particular to the interpreter).

4.1. Commands to obtain help

In Linux, there are two commands that we can execute locally in the machine to obtain interactive help: the **man** command, which offers help about external commands (as part of the installation, the manual pages that we can consult using the **man** command are also installed), and the **help** command, which offers help about internal commands.

- **Read the guide about "How to use the Linux man"** that you have at the end of this section ("Using the manual"). Afterwards, **query the man** (man command_name) of the following commands. Specifically, for each command you have to read and understand perfectly: the SYNOPSIS, the DESCRIPTION and the options that appear under the "Options" column of the table.

To read with man	Basic description	Options
man	Accesses the on-line manuals	
ls	Shows the contents of a directory	-l, -a
alias	Defines an alternative name for a command	
mkdir	Creates a directory	
rmdir	Removes an empty directory	
mv	Changes the name of a file or moves it to another directory	-i
cp	Copies files and directories	-i
rm	Deletes files or directories	-i
echo	Visualize a text (can be an environment variable)	
less	Shows files in a format suitable for the terminal	
cat	Concatenates files and shows them through the standard output	
grep	Searches text (or patterns of text) in files	
gedit	Text editor for GNOME	
env	Executes a command in a modified environment, if no command is specified, it shows the environment	
chmod	Modifies the access permissions of a file	
which	Finds a command	

- **Use the help command** to get help about the following internal commands:

To consult with help	Basic description	Options
help	Offers information about the internal commands of the Shell	
export	Defines an environment variable	
cd	Changes the current directory	
alias	Defines an alternative name for a command	

- **Access the man page for bash (executing the "man bash" command)** and search the meaning of the environment variables PATH, HOME and PWD (note: the "/" character is used to search patterns in man pages. Use it to find the descriptions of these variables).

Using the manual

Knowing how to use the manual is basic since, although some commands will be explicitly explained, you will have to search for the rest yourself in the manual. The manual is self-contained, you can see all its options executing:

man man

The manual information is organized in sections. Section 2, for example, is called system calls. The sections that we can find are:

1. commands
2. system calls
3. calls to user or language libraries
4. etc.

The information provided when executing man is what is called a "man page". A "man page" is usually the name of a command, system call or function call. All the pages of the manual follow a similar format, organized in several parts. In figure 2 you have an example of the output of the man for the command ls (we have deleted some lines to be able to see all the principal parts). In the first part you can find the name of the command, its description and a schema (SYNOPSIS) of its usage. In this part you can observe if the command accepts options, or if it needs some fixed or optional parameters, etc.

```
LS(1)                 User Commands                 LS(1)
NAME
     ls - list directory contents
SYNOPSIS
     ls [OPTION]... [FILE]...
DESCRIPTION
     List  information  about  the FILEs (the current directory by default).
     Sort entries alphabetically if none of -cftuSUX nor --sort.
     Mandatory arguments to long options are  mandatory  for  short  options
     too.
     -a, --all
          do not ignore entries starting with .
SEE ALSO
     The  full  documentation  for ls is maintained as a Texinfo manual.  If
     the info and ls programs are properly installed at your site, the  command
                    info ls
     should give you access to the complete manual.
ls 5.93               November 2005                 LS(1)
```

Figure 2 man ls (simplified)

The next part is the DESCRIPTION of the command. This part includes a more detailed description of its usage and the list of options it supports. Depending on the installation of the man pages you can find also here the EXIT STATUS of the command. Finally there's usually several parts that include the authors of the manual, the way to report errors, examples, and related commands (SEE ALSO section).

In figure 3 you have the result of executing "man 2 write", which corresponds with the system call write. The number that we put before the page is the section where we want to search and that we include in case that more than one page with the name write exist in other sections. In this case the SYNOPSIS includes the files that need to be included in the C program to be able to use the concrete system call (in this case unistd.h). If it would be necessary to link your program to some concrete library, different from the by default libraries that the C compiler uses, it would be listed here as well. In addition to the DESCRIPTION, in the function calls in general (be it a system call or a language library call) we can find a RETURN VALUE section (with the values that returns the function) and a special section, ERRORS, with the list of possible errors. Finally we can also find some more additional sections, NOTES (clarifications) and SEE ALSO (related calls).

```
WRITE(2)                    Linux Programmers Manual                    WRITE(2)
NAME
       write - write to a file descriptor
SYNOPSIS
       #include <unistd.h>
       ssize_t write(int fd, const void *buf, size_t count);
DESCRIPTION
       write()  writes  up  to  count bytes to the file referenced by the file
       descriptor fd from the buffer starting at buf.  POSIX requires  that  a
       read()  which  can  be  proved  to  occur  after a write() has returned
       returns the new data.  Note that not all file systems  are  POSIX  con-
       forming.
RETURN VALUE
       On    success,   the   number of bytes written are returned (zero
indicates
       nothing was written).  On error, -1  is  returned,  and  errno  is
set
       appropriately.   If  count  is zero and the file descriptor refers
to a
       regular file, 0 will be returned without causing any other effect.
For
       a special file, the results are not portable.
ERRORS
       EAGAIN Non-blocking  I/O  has  been  selected  using O_NONBLOCK and the
              write would block.
       EBADF  fd is not a valid file descriptor or is not open for writing.
       ….

       Other errors may occur, depending on the object connected to fd.
NOTES
       A successful return from write() does not make any guarantee that  data
       has been committed to disk.  In fact, on some buggy implementations, it
       does not even guarantee that space has successfully been  reserved  for
       the  data.   The  only way to be sure is to call fsync(2) after you are
       done writing all your data.
```

Figure 3 man 2 write (simplified)

The man is simply a system tool that interprets some markings in a file text and displays them following the instructions of these markings. The four basic things that you need to know are:

- Usually a man page occupies several screens, to advance you just need to press the space bar.
- To go to a previous screen you can press the letter **b** (back).
- To search a text and directly go to it you can use the character "/" followed by the text. For example "/SEE ALSO" will send you to the first appearance of the text "SEE ALSO". To go to the next appearance of the same text simply use the letter **n** (next).
- To exit the man page use the letter **q** (quit).

5. Bibliography

The documentation we give you in this booklet is usually enough to do the sessions, but in each session we will give you some extra references.

- BASH shell guide:
    - From the ASO course(in Catalan):
      http://docencia.ac.upc.edu/FIB/grau/ASO/files/lab/aso-lab-bash-guide.pdf

    - In English: http://tldp.org/LDP/abs/html/index.html
    - Usage tutorial of the Shell:
      http://www.ant.org.ar/cursos/curso_intro/c920.html

Exercises to do in the laboratory

- ➢ The practises will be done in a Ubuntu 20.04 LTS system.
- ➢ You have at your disposal a system image equal to the one used in the laboratories to be able to prepare the sessions at home. The image is for VMPlayer:
    - https://my.vmware.com/web/vmware/free#desktop_end_user_computing/vmware_player/6_0
    - Image: https://softdocencia.fib.upc.edu/software/Ubuntu-2021.zip
- ➢ Answer in a text file named to_deliver.txt all the questions that appear in this document, indicating for each question its number and answer. This document must be delivered through Racó (the FIB's intranet). The questions are highlighted in bold and marked with the following symbol:

- ➢ The lines of the instructions that start with the character "#" indicate commands that you have to try. You do NOT have to write the character #.
- ➢ **To deliver:: file sesion01.tar.gz**

    ```
    #tar zcfv sesion01.tar.gz to_deliver.txt
    ```

Navigate through directories (folders in graphical environments)

You can observe that the vast majority of the basic Linux commands are 2 or 3 letters that synthesize the operation to realise. For example, to change the directory you have to use the command cd. To see the contents of a directory (list directory) you have to use the ls command, etc.

In Unix based systems, the directories are organized in a hierarchical way. The base directory is the root (represented with the character /) and from there hang the rest of the directories of the system, where the directories and files common to all the users are situated. Moreover, inside this hierarchy, each user has a directory assigned (home directory), though to act as a base for the rest of his directories and files. When a user initiates a terminal, its working directory is its home directory. To change the working directory you can use the command **cd**, which lets you navigate through the file system hierarchy.
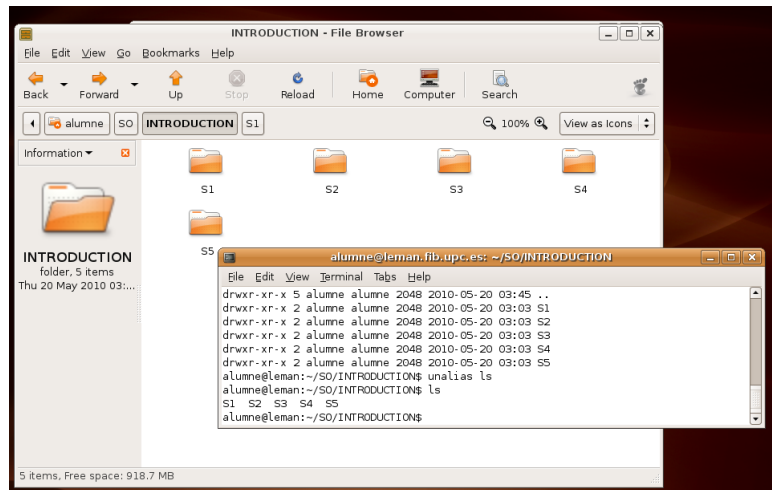
Next, do the following exercises using the commands that you think are more appropriate:

1. Create the directories for the first five sessions of the course with names S1, S2, S3, S4 and S5.

2. If you open the File Browser of Ubuntu, and go to the same "folder" where you are in the Shell, you should see something similar to this image:

3. Change the current work directory to directory S1.
4. List the contents of the directory. Apparently there's nothing there. However, there are two "**hidden files**". All the Unix files that start with the character "." Are **hidden files**, and they are usually special. Look up what options do you need to add to the command to see all the files. The files that you can see now are:
   o Directory type file ".": It references the same directory where you are at the moment. If you execute (`cd .`) you will see that you are still in the same directory. We will see its utility later.
   o Directory type file "..": It references the directory of the immediate upper level from where we are. If you execute (`cd ..`) you will see that you change to the previous directory.
   o Note that these hidden files don't appear in the graphical environment: if you access the S1 folder it appears empty (in the graphical environment, it also exists a configuration option for folders that allows the display of hidden files).

QUESTION 2. **What commands do you use to list the contents of a directory? What option do you have to add to display the hidden files?**

5. The options of the commands can usually be accumulated. Look up in the manual which option do you have to use to see extended information about the files and try it out.

**QUESTION 3.** **What option do you have to add to ls to see the extended information of the files?**

6. When we use really often a specific configuration of a command, it is usual to use what is known as "alias". It consists in defining a pseudo-command that the Shell can

recognize. For example, if we see that we always execute the ls command with the options "–la", we can define "ls" as an alias in the following way:

```
#alias ls='ls –la'
```

Execute this alias command and then execute ls without options. Check that the output is the output of the ls command with the options -la.

7. We can see information similar to the ls –la command in the graphical environment. Look up how to modify the file browser to achieve it. You should see something similar to the next figure:
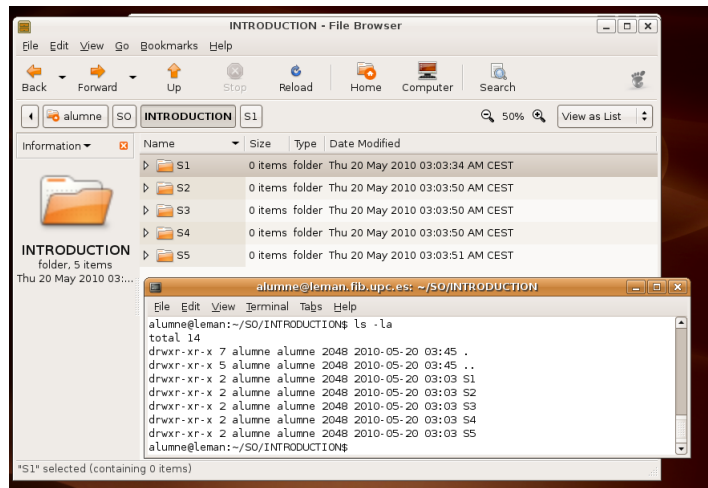
The columns shown here are the ones selected by default but you can change them with the option View.

QUESTION 4. **Which options of the menu have you activated to extend the information displayed by the File Browser?**

8. From the Shell, delete some of the directories that you have created, check that they don't appear and create them again. Look up how to do that in the graphical environment as well.

QUESTION 5. **Which command sequence have you executed to delete a directory, check that it's not there and create it again?**

Basic file system commands to access files

1. Create a file. To create a file that contains any text we have several options, for example opening the editor and writing anything:
   o #gedit test
2. To be able to execute any other command you will see that you need to open another Shell because the one you had is blocked by the editor (this only happens when opening the first file, not if you had the editor already opened). This is so because the Shell, by default, waits until the current command finishes before showing the prompt and processing the next command. To avoid needing an opened Shell for each command

that we want to execute simultaneously, we can ask the Shell to execute the commands in the **background.** When we use this method, the Shell executes the command and immediately shows the prompt and starts to wait for the next command (without waiting for the previous one to finish). To execute a command in the background you have to add at the end of the line the special character "&". For example:

- o #gedit test &

3. To see in a quick way the contents of the file, without opening again the editor, we have several commands. We will mention two here: cat and less. Add to the test file 3 or 4 **pages** of text (anything). Try the commands cat and less.

- o Note: if the file you have created is not big enough you won't see any difference.

QUESTION 6.   What difference exists between cat and less?

4. Copy the test file several times (adding a different number at the end of the name of each destination file, e.g. "test2"). What would happen if the source and destination files had the same name? Look in the man the option "-I" of the command cp. What does it do? Create an alias of the command cp (call it cp) that includes the option –i.

QUESTION 7.   What's the function of the –i option in the cp command? Which is the command to do an alias for the cp command that includes the –i option?

5. Try to delete some files that you have just created and change the name of others. Make an alias for the –i option of the rm command (call it rmi). Check also the –i option of the mv command.

QUESTION 8.   What does the –i option of the rm command? And the –i option of mv? Write the command to do an alias of the rm command that includes the –i option.

6. Another really useful command is the grep command. The grep command allows the search of a text (explicit or through a pattern) in one or more files. Add a word in one of the files that you have copied and try the grep command to search that word. For example, add the word "hello" to one of the files and we do this test:

```
#grep hello test test1 test2 test3 test4
```

7. The ls –l command also allows the visualisation of the permissions of a file. In UNIX, the permissions are applied in three levels: the owner of the file (u), the users in the same group (g), and the rest of users (o). And they reference three operations or access modes: read (r), write (w) and execution (x). For example, if in the current directory there's only file f1, and this file has read and write permissions for the owner of the file, read only for the members of the group and read only for the rest of the users of the machine, the execution of the command would give the following output: # ls –la

```
drwxr-xr-x  26      alumne      alumne      884   2011-09-15 14:31 .
drwxr-xr-x  3       alumne      alumne      102   2011-09-15 12:10 ..
-rw-r--r--  1       alumne      alumne      300   2011-09-15 12:20 f1
```

The first column of the output indicates the type of file and its access permissions. The first character encodes the type of file (the character 'd' means directory and the

character '-' means data file). Next, the first group of 3 characters represent, in order, if the owner has read permission (using the 'r' character) or doesn't have it (then the character '-' shows up), if the owner has write permission (character 'w') or he cannot write (character '-') and if he has or not permission to execute it (character 'x' or character '-'). The second group of three characters are the permissions that the members of the owner's group have and the last group of 3 characters are the permissions of the rest of users of the machine.

These permissions can be modified using the chmod command. The chmod command offers several ways to specify the access permissions, a very simple way consists in indicating first the users that are going to be affected by the permission change, how do we want to change these permissions (adding, removing or directly assigning) and the affected operation. For example the command:

```
#chmod ugo+x f1
```
Would modify the permissions of f1, activating the execution permission over f1 for all the users of the machine.

The command:

```
#chmod o-x f1
```
Would modify the permissions of f1 removing the execution permission for users that are not the owner of the file nor belong to its group.

And the command:

```
#chmod ug=rwx f1
```
Would change the f1 permissions to the given ones: read, write and execute for the owner and its group members.

Modify the permissions of the test file to have only write permission for the owner, its group and the rest of users and try to do a cat. Modify again the permissions of test leaving only read permissions for the owner of the file, its group and the rest of users and try to delete it.

QUESTION 9. **What options of the chmod have you used to leave only write permissions? What was the result of cat when trying to see the test file? What options of chmod have you used to leave only the write permissions? Have you succeeded in deleting it?**

Environment variables

The programs are executed in a given environment or context: they belong to a user, a group, from a concrete directory, with a system configuration regarding limitation, etc. More details about the context or environment of a program will be explained in the processes unit.

In this session we will introduce the **environment variables**. The environment variables are similar to the constants that can be defined in a program, but they are defined before starting the program and they usually reference system aspects (e.g. default directories) and they mark

some important aspects of its execution, since some of them are used by the Shell to define its functioning. They are usually defined in capital letters, but it's not mandatory. These variables can be consulted during the execution of a program through functions of the C library. To consult the meaning of the variables defined by the Shell, you can consult the manual of the Shell you are using, in this case bash (man bash, Shell Variables part).

8. Execute the "env" command to see the list of the variables defined in the current environment and their values.

To indicate the Shell that we want to consult an environment variable we have to use the $ character before the name of the variable, with the purpose of not confusing it with some text chain. To see the value of a concrete environment variable we use the command echo:

```
#echo $USERNAME
#echo $PWD
```

9. Some variables are dynamically updated by the Shell, for example, change the directory and consult again the value of PWD. What do you think is the meaning of this variable?
10. Check the values of the variables PATH and HOME.

> **QUESTION 10. What's the meaning of the environment variables PATH, HOME and PWD?**
> **QUESTION 11. The PATH variable is a list of directories, what characters acts as separator between directories?**

We can also define or modify an environment variable using the following command (to modify it the $ character is not used):

```
export VARIABLE_NAME=value (without spaces).
```

11. Define two new environment variables with the value you want and check their value.

> **QUESTION 12. Which command have you used to define and check the values of the new variables you have defined?**

12. Download the file S1.tar.gz and copy it to the S1 folder. Unzip it using the tar xvfz S1.tar.gz command to obtain the "ls" program we will use next.
13. Place yourself in the S1 folder and execute the following commands:

```
#ls
#./ls
```

Note that in the first option, the system command is executed instead of the ls command that resides in your directory. However, with the second option you have executed the ls program you just downloaded instead of using the ls system command.

14. Add the "." Directory at the beginning of the PATH variable using the command: (note the directory separator character):

```
#export PATH=.:$PATH
```

Show the new value of the PATH variable and check that, apart from the "." Directory, it still contains the directories that had originally (we don't want to lose them).

Execute now the command:

```
#ls
```

15. Modify the PATH variable to eliminate the "." directory. You cannot partially modify a variable thus you will have to define it again. Show the current content of PATH and redefine it using all the directories that had except for the ".".

Execute now the command:

```
#ls
```

16. Add the "." directory at the end of the PATH variable (note the directory separator character):

```
#export PATH=$PATH:.
```

Check that, besides the "." directory, the PATH variable still contains the directories that originally had (we don't want to lose them).

Execute now the command:

```
#ls
```

Keeping the changes: .bashrc file

The changes that we have done during this session (except the ones that reference the file system) will be lost when the session ends (alias definition, PATH changes, etc). To not lose them, we have to insert these changes in the session configuration file that the Shell uses. The name of the file depends on the Shell that we are using. In the case of Bash, the file is $HOME/.bashrc. Each time we initiate a session, the Shell is configured executing the commands that finds in that file.

17. Edit the $HOME/.bashrc file and add the PATH modification that we have asked in the last section. Also add the definition of an alias to make the execution of the ls command using always the –m option. To change that you have correctly modified the .bashrc execute the following command:

```
#source $HOME/.bashrc
#ls
```

And check that the output of the ls corresponds with the –m option. The source command causes the execution of all the commands provided as a parameter (it's a way for not having to reboot the session to make those changes effective).

- **Note:** in the work environment in the lab classes, the system boots using REMBO. Meaning, it loads a new system image and therefore all your files are lost and the configuration files are reinitialized with a remote copy. This means that if you reinitiate the session you will start working with the original .bashrc file and your changes will not be maintained.

Some useful special characters of the Shell

In previous sections we have introduced the & character, that is used to execute a command in the background. Other useful characters that we will introduce in this session are:

- *: The Shell substitutes it for any group of characters (except the "."). For example, if we execute (#grep test t*) we will see that the Shell substitutes the t* pattern for the list of all the files that start with the string "t". The special characters of the Shell can be used with any command.
- >: The data output by default of the commands is associated with the screen. If we want to change this association, and make the output redirect to a file, we can do this with the ">" character. This action is called "output redirection". For example, `ls > output_ls`, stores the output of the ls in the file output_ls. Try to execute the previous command. Next, try it with another command but with the same output file and check the content of the file output_ls.
- >>: Redirection of the data output of a command to a file but instead of removing the content of the file, the output is appended at the end of the file. Repeat the previous example with ">>" instead of ">" to check the difference.

QUESTION 17. What's the difference between using > and >>?

Make a backup for the next session

Given that a new image is booted each time we start a laboratory computer, it is necessary to do a backup of the changes made if we want to preserve them for the next session. To save the changes you can use the tar command. For example, if you want to generate a file that contains all the files in the S1 directory, in addition to the .bashrc file, you can use the following command from your HOME directory:

```
#tar zcvf folderS1.tar.gz S1/* .bashrc
```

Finally you have to save this file in a safe place, like a pendrive or your own email account.

Session 2: The C language

Previous work

Objectives

In this session we will practise everything related to the creation of executables. In this course we will use the C language. We will practise the error correction both in makefiles and in C files and the generation of executables from scratch: source files, makefiles and libraries.

Skills

- Be able to create executables given a C code using simple makefiles (created from scratch).
- Be able to create C programs from scratch:
    - Definition of basic types, tables, functions, conditionals and loops.
    - Usage of pointers.
    - Screen formatting of the program results.
    - Well-structured, clear, robust and well documented programs.
    - Creation and modification of simple makefiles: add new rules and add dependencies.
    - Indentation application in C source code.

Previous knowledge

- Basic programming: basic data types, tables, sprintf.
- Intermediate programming: C pointers, access to command line parameters. Stages of the development of a C program: Processor/Compiler/Linker
- Use of simple makefiles

Previous work guide

- Read the man pages for the following commands. These commands have multiple options, read with special interest the ones noted in the "Options" column of the following table. It is NOT necessary to read the complete man pages of every command, only their basic functioning that you can broaden when needed.

To read in the man	Basic description	Options
make	Tool to automate the compilation/linking process of a program or group of programs	
gcc	GNU C compiler	-c,-o, −I (uppercase i), −L,-l (lowercase l)
sprint	Format conversion storing it on a buffer	
atoi	Converts a string to an integer number	

indent	Source code indentation

- Consult the programming in C overview available in the course webpage: http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/ejemplos/EsquemaProgramaC.pdf
- Consult the overview of the comparison between C and C++ available in the course webpage: http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/Laboratorios/C++vsC.ppsx
- Create the $HOME/S2 folder and place yourself on it to do the exercises.
- Download the S2.tar.gz file, decompress it with "tar zxfv S2.tar.gz" to obtain the files of this session.

Throughout the course, we will directly use system calls to write on the screen. What we did in C++ with a cout, here we do it with a combination of **sprintf** (C library function) + **write** (system call to write).

The **sprintf** function of the standard C library is used to format a text and store the result in a buffer. The first parameter is the output buffer, of type char*, the second parameter is a character string that specifies the text to be stored as well as the format and type of all the variables or constants that you want to include in the string, and from there, these variables and constants. (Look the examples that we give you in the C overview). Everything we write in the screen has to be ascii codes, that's why we first need to format it using the sprint function (except for the case where it is already ascii). The % special character specifies the variable data type to be substituted at that position. We will use %d for integers, %s for "strings" and %c for characters.

Afterwards, we have to use the **write** system call to write this buffer to a device. During these first sessions we will write in the "standard output", which by default is the screen. In UNIX/Linux, the devices are identified with a number, which is usually called channel. The **standard output** is the channel 1 and the **standard error output** is the channel 2. The **channel number** is the first parameter in the write system call. The rest of parameters are the **address** where the data starts (the buffer we generated with sprintf) and the number of bytes that we want to write from this address. To know the length of the text we will use the **strlen** (C library function).

A usage example of sprintf could be (use the man to consult how to specify different formats):

```
char buffer[256];
sprintf(buffer, "One number %d, one char %c and a text %s\n", 1,'a',"hola");
write(1, buffer, strlen(buffer));
```

Which shows through standard output: "One number 1, one char a and a text hola".

Solving problems with the makefile

1. Modify the makefile to make it work

- The makefile is the file used by default by the make tool. If you execute "make" without parameters, the default file is makefile
- For a makefile to be correct, it has to follow the following formatting (TAB means that it needs a tabulator)

```
Target: dependency1 dependency2.. dependencyN
[TAB] how to generate target from the dependencies
```

For example, if P1 executable depends on P1.c source code:

```
P1: P1.c
        gcc –o P1 P1.c
```

2. Modify the makefile for the rule listaParametros to have its dependencies well defined. The executable files depend on source files from which they are generated. (or object files if we separate the executable in compilation and linking)
3. Modify the makefile for the output file to be listaParametros instead of a.out
4. Create a copy of the makefile, calling it makefile_1, to deliver it.

Solving compilation problems

- Solve all the compilation errors that appear. It's always advisable to solve the errors in their apparition order because they are propagated sequentially.

The compiler always gives messages indicating where the error has been produced, in which line of code, and what error is it. In this concrete case we have the following code:

```
1.   void main(int argc,char *argv[])
2.   {
3.           char buf[80];
4.           for (i=0;i<argc;i++){
5.                   sprintf(buf,"The argument number %d is %s\n",i,argv[i]);
6.                   write(1,buf,strlen(buf));
7.           return 0;
8.   }
```

And errors are (they are really typical errors, that's why we have put them here):

```
listaParametros.c: In function "main":
listaParametros.c:4: error: "i" undeclared (first use in this
function)
listaParametros.c:4: error: (Each undeclared identifier is
reported only once
listaParametros.c:4: error: for each function it appears in.)
listaParametros.c:5: warning: incompatible implicit declaration
of built-in function "sprintf"
listaParametros.c:6: warning: incompatible implicit declaration
of built-in function "strlen"
listaParametros.c:7: warning: "return" with a value, in
function returning void
listaParametros.c:8: error: syntax error at end of input
```

The first one indicates that there's a variable (`i` in this case) without being declared in **line 4.** Line 4 is where it is used. If you look closely, we use `i` in the loop but it's not declared. In **lines 5 and 6** we use functions (`sprintf` and `strlen`) that we haven't declared. The compiler doesn't find them and cannot know if they are correct or not. These functions are defined in the C library that is added when generating the executable file, but the compiler needs the headers to see if they are used correctly. Check the man to see which headers files (.h) are needed. **Line 7** indicates that we have a function (the main) that ends with a return 0, when it was declared as a void. Common practise is defining the main as a function that returns an int. The last error usually appears when there's an error that has been propagated. In this case, we needed to close a brace (the one that closes the for loop).

- Make sure that the "." directory appears in the PATH variable in such a way that the executables that are in the current directory can be found.

Analyse and understand the listaParametros.c file

If you are unable to program in C reed the recommended documents before doing these exercises.

- The first function executed in a C program is the `main` function.
- The main function has two parameters called `argc` and `argv`. The `argc` parameter is an integer that contains the number of elements of the array `argv` (0..(number_elements-1)). And `argv` is a string array that contains the input parameters that the program receives when being executed. There exists the convention of treating the name of the executable as the first parameter. Therefore, is this convention is followed, `argc` will always be greater or equal to one and the first element of `argv` will be the name of that executable (the Shell follows this convention for all the programs that executes).
- Compile listaParametros.c to obtain the listaParametros executable, and execute it with different parameters (type and number), for example:

```
#./listaParametros
#./listaParametros a
#./listaParametros a b
#./listaParametros a b c
#./listaParametros 1 2 3 4 5 6 7
```
What are the values of argc and argv in every one of the examples? Note that, even though we pass numbers as parameters, the program ALWAYS receives a string of characters.

The shell is the responsible of filling both parameters (`argc` and `argv`) using the data introduced by the user (we will see it in theory Unit 2).

Analyse carefully and understand the punteros.c file.

The objective is that you assimilate the meaning of declaring a variable of type pointer. A pointer is used to store a memory address. And in that memory address there's a value of the type

indicated in the declaration of the pointer. Here you have an example with the concrete case of pointers to integers.

```
char buffer[128];
int A;       // This is an integer
int *PA;     // This is an integer pointer (it is still not initialized
             // it can't be used)
PA=&A;       // We initialize PA as the address of A
*PA=4;       // We place a 4 in the memory address pointed by PA
if (*PA==A) {
      sprint(buffer,"This message should be always displayed!\n");
}else{
      sprint(buffer,"This message should NEVER be displayed!\n");
}
write(1,buffer,strlen(buffer));
```

- Create a numbers.c file with a function that checks that a string that receives as a parameter only contains ASCII between the '0' and the '9' (in addition to the '\0' at the end and a potential '-' at the beginning for negatives). In C the string data type doesn't exist. It's defined as a "char *" or as a character vector. To be considered as a string, it must finish with the '\0' special character. The function returns 1 if the string represents a number and has at much 8 digits (define a constant that we will call MAX_SIZE and use it), or 0 otherwise. The function must have the following prototype:
    - int isNumber(char *str);
- To check the isNumber function, add the function to the main of the numbers.c file and make it to execute the function isNumber passing all the parameters that the program receives (argv[1], argv[2], etc). Write a message for each one indicating if it's a number or not.
- Create a Makefile for this program
- Use indent to indent the numbers.c file (#indent numbers.c).
- **To DELIVER: prevS2.tar.gz**
    #tar zcfv prevS2.tar.gz numbers.c Makefile

Bibliography

- C programming tutorial: http://www.elrincondelc.com/cursoc/cursoc.html
- Overview about C programming: http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/ejemplos/EsquemaProgramaC.pdf
- Comparison between C and C++: http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/Laboratorios/C++vsC.ppsx

Complementary bibliography

- Programming in C:
    - Programming Language. Kernighan, Brian W.; Ritchie, Dennis M. Prentice Hall
    - Interactive C course: http://labsopa.dis.ulpgc.es/cpp/intro_c.
- Makefiles:

- o http://es.wikipedia.org/wiki/Make
- o http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/
- GNU Coding Standards
  - o http://www.gnu.org/prep/standards/standards.html, specially the point: http://www.gnu.org/prep/standards/standards.html#Writing-C

Exercises to do in the laboratory

➢ For all the exercises, we assume that the makefile will be modified when necessary and all the exercises will be tested.

➢ Answer in a file called to_deliver.txt all the questions that appear in this document, indicating for each question its number and its answer. This document has to be delivered through the Racó. The questions are highlighted in bold and marked with this symbol:

➢ **To deliver: sesion02.tar.gz**

#tar zcfv sesion02.tar.gz to_deliver.txt makefile_1 listaParametros.c makefile_4 my_functions.h my_functions.c suma.c punteros.c makefile_5 words.c makefile makefile_5

Checking the number of parameters

From now on, we will do all our programs robust and usable checking that the number of parameters received is the correct one. Otherwise, the programs (1) will print a message describing how to use them, (2) a line describing its functionality, and (3) will end the execution. That's what is known to be as adding a usage() function.

1. Implement **a function** that handles the showing of a usage descriptive message of the program listaParametros and that ends the execution of the program (call this function Usage()). **Modify the program listaParametros** to check that at least it receives 1 parameter and otherwise it calls the Usage() function. Remember that the name of the executable is considered as an extra parameter and this is reflected in the variables argc and argv.

   Example of the expected behaviour:

   ```
   #listaParametros  a b          ⟵————————  Example of correct use
   The argument number 0 is listaParametros
   The argument number 1 is a
   The argument number 2 is b
   #listaParametros               ⟵————————  Example of incorrect use
   Usage:listaParametros arg1 [arg2..argn]
   This program writes in the standard output the arguments it
   receives
   ```

Parameter processing
2. Create a copy of numbers.c (previous work) named suma.c.

3. Add another function to suma.c that converts a character to a number (1 digit). The function assumes that the character corresponds with the character of a number.

```
unsigned int char2int(char c);
```

4. Modify suma.c adding a mi_atoi function that receives as a parameter a string and returns an integer corresponding to the string converted to a number. This function assumes that the string is not a pointer to NULL, but can represent a negative number. If the string doesn't contain the correct number, the function returns the conversion until the first incorrect digit. The behaviour of this function is the same as the *atoi* of the C library. Use the function char2int of the previous section.

```
int mi_atoi(char *s);
```

5. Modify suma.c to make it behaves in the following way: After checking that all parameters are numbers, it converts them to integer, it adds them and writes the result. Modify also the Makefile to create the executable file *suma*. The next figure shows an example of the functioning of the program:

```
#suma 100 2 3 4 100
The addition is 209
#suma -1 1
The addition is 0
#suma 100 a
Error: the parameter "a" is not a number
```

6. Create a copy of the makefile, calling it makefile_4, to deliver it.

Using the C pre-processor: Divide the code (#include)

We want to separate the auxiliary functions that we create in the main programs, in such a way that we can reuse them when needed. In C, when we want to encapsulate a function or a set of functions, the usual way is creating two types of files:

- **Header** files ("*include*" files). They are text files with the extension ".h" that contain the **prototypes of the functions** (headers), definitions of the **data types** and definitions of **constants.** These files are included by the pre-processor using the #include <file.h> directive in the exact place where the directive appears, meaning that the position is important. Including a ".h" file **is equivalent to copying and pasting the file** in the place where the #include directive is written and its sole objective is to facilitate the readability and modularity of the code. The correct way is putting in the include files those function prototypes, data types and constants that we want to use in more than one file.

- Auxiliary files, **object files or libraries.** These files contain the **global variable definitions** that are needed in the auxiliary functions (if they need any) and the implementation of those functions. We can offer the ".c" file directly, the object file once compiled (if we don't want to offer the source code) or, in the case of having more than one object file, grouping all of them in a library (archive with the command ar).

7. Separate the functions done in the previous exercises (except for the main) in a separate file (my_functions.c) and create a headers file (my_functions.h), where you define the

headers of the functions you offer, and include them in the program suma.c. Add a brief description of each function together with the prototype (add it as a comment). Modify the makefile adding a new rule to create the object file and modify the rule of the *suma* program to use this new object. Add the dependencies that you believe are necessary.

8. Modify the Usage function in a way that the *suma* program receives at least 2 parameters.

9. Create a copy of the makefile, calling it makefile_5, to deliver.

> **QUESTION 18. Which options did you need to add to the gcc to generate the object file? Which option did you need to add to the gcc for the compiler to find the my_functions.h file?**

Working with pointers in C

10. Look at the code of the program punteros.c. Modify the makefile to compile it and execute it. Analyse the code to understand what it does.

11. Create a program called words.c that accepts a unique parameter. This program counts the number of words present in the string passed as a parameter. Consider that a new word starts after: a space, a dot, a comma and a line break ('\n'). The rest of punctuation signs are not taken into account. To include more than one word in a single parameter, put them between double quotes. An example of its functioning would be:

```
#words hello
1 words
#words "This is a sentence."
4 words
#words "This parameter is treated" "This parameter is not
treated"
4 words
```

12. Modify the makefile to compile and mount words.c

Session 3: Processes

Previous work

Objectives
The objectives of this session are practicing with the basic system calls to manage processes, and the basic commands and mechanisms to monitor kernel information associated to the active processes of the system.

Abilities
- BASIC user level:
  - Be able to do a concurrent program using process management system calls: fork, execlp, getpid, exit, waitpid.
  - Understand the process inheritance and the parent/child relation.
- BASIC admin level
  - Be able to see the process list of a user and some details of his/her status using commands (ps, top).
  - Begin to get information through the /proc pseudo-filesystem.

Previous work guide
- Consult the video about process creation that you have in the course's web page: http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/ejemplos/EjemploCreacionProcesosVideo.wmv
- Read the manual pages of the getpid/fork/exit/waitpid/execlp system calls. Understand the parameters, return values and basic functionality associated to the theory explained at the lectures. Notice the necessary includes, error cases and return values. Consult the description and indicated options of the command ps and the /proc pseudo-filesystem.

To read in the man	Basic description	Principal Parameters/arguments that we will practice
`getpid`	Return the PID of the executing process	
`fork`	Creates a new process, child of the executing process	
`exit`	Ends the executing process that calls it	
`waitpid`	Waits until the finalization of a child process	
`execlp`	Executes a program in the context of the same process	
`perror`	Writes the message of the last error produced	
`ps`	Return information about the processes	-a, -u, -o

proc	Pseudo-filesystem that offers information about the kernel data	cmdline, cwd, environ exe, status

- Create the directory for the development environment for this session ($HOME/S3).
- Download the file S3.tar.gz and unzip it in the directory you just created to obtain the files of this session (tar zxvf S3.tar.gz).
- Create a text file called previo.txt and write in it the answer to all the questions.
- **Analyse the code** of the example programs and the Makefile.ejemplos file
  - The Makefile.ejemplos file is prepared to compile all the programs except for ejemplo_fork7.c
- **Compile all the programs,** except for ejemplo_fork7, **using the file Makefile.ejemplos** (see file README_S3).
- **Execute ejemplo_fork1**
  - Write in the file previo.txt the messages that appear on screen and explain which process shows each of them (parent or child) and why.
- **Execute ejemplo_fork2**
  - Write in the file previo.txt the messages that appear on screen and explain which process shows each of them (parent or child) and why.
- **Execute ejemplo_fork3**
  - Write in the file previo.txt the messages that appear on screen and explain which process shows each of them (parent or child) and why.
- **Execute ejemplo_fork4**
  - In which order do the messages appear? Which process ends its execution first?
  - **Modify the code of this program** in order to make that the parent doesn't write the last message of its code until his child has ended execution.
- **Execute ejemplo_fork5**
  - Write in the file previo.txt the messages that appear on screen and explain which process shows each of them (parent or child) and why.
  - **Modify the code of this program**, to make the child process modify the values of the variables variable_local and variable_global before printing their values. Check that the parent still sees the same value the variables had before doing the fork.
- **Execute ejemplo_fork6,** redirecting its standard output in a file
  - Describe the contents of the output file
  - Can we ensure that if we execute this program multiple times the contents of the output file will always be the same? Why?
- **Modify the Makefile.ejemplos file** to add the compilation of ejemplo_fork7.c and use it to compile it now.
  - Why does the program ejemplo_fork7.c NOT compile? Does it have something to do with the fact of creating processes? How can the code be modified to make it write the value of "variable_local"?
- **Execute ejemplo_exec1**

o Describe the behaviour of this program. What do you see on the screen? How many times does the process pid appear on screen? Why is that?
- **Execute ejemplo_exec2**
  o Describe the behaviour of this code. What messages appear on screen? How many processes are executed?

- **Consult in the manual** to which section belong the man pages that you have consulted. Moreover, write here if you have consulted some manual page different to the ones that have been explicitly asked.

- **TO DELIVER: previo03.tar.gz**
  #tar zcfv previo03.tar.gz Makefile.ejemplos ejemplo_fork4.c ejemplo_fork5.c ejemplo_fork7.c previo.txt

Exercises to do in the laboratory

➤ For all the exercises, it is assumed that you do all the involved steps.
➤ Answer in a text file named entrega.txt all the questions that appear in this document, indicating for each question its number and answer. This document must be delivered through Racó. The questions are highlighted in bold and marked with the following symbol:
➤ **To deliver: sesion03.tar.gz**

#tar zcfv sesion03.tar.gz entrega.txt makefile myPS_v0.c myPS.c myPS2.c myPS3.c parsExec.c.

Error checking in the system calls

1. From now on you will always include, **for ALL system calls,** the error checking. Use the perror call (man 3 perror) to write a message that describes the reason why the error was produced. Moreover, in the case of the error being critical, for example a fork or an execlp failing, the execution of the program must end. The error management of the system calls can be done in a way similar to the following code:

```
int main (int argc,char *argv[])
{
…
        if ((pid=fork())<0) error_and_exit("Error in fork",1);
…
}
void error_and_exit(char *msg,int exit_status)
{
        perror(msg);
        exit(exit_status);
}
```

Creating and mutating processes

The objective of this section is to practise with the process creation and mutation system calls. For it you will create two different codes: myPS.c y myPs_v0.c. These codes will serve as a base for the exercises of the following sections.

2. Create a program called myPS.c that receives a parameter (which will be a username: root, alumne, etc). Add the function usage that we saw in session 2 to make sure that the user enters at least one parameter. Here you have an example:

> #myPS [↵ Enter]
>
> usage: myPS username
>
> This program executes a ps of the processes of the user it receives as a parameter

3. The program will create a child process. The parent process will write a message indicating its PID. The child process will write a message with its PID and the parameter that the program has received. After writing the message, both processes will execute a while(1) loop; with the aim of not ending the program (we add this loop because we will use this code in the next section about consulting the information of processes, and in that section we are interested in the processes not ending their execution).

4. Create a makefile that includes the rules all and clean, to compile and link the myPS.c program.

⚠️ **QUESTION 19. How can a process know the pid of its children? What system calls can the processes use to consult their own PID?**

5. Copy the code of myPS.c in a version myPS_v0.c. Modify the Makefile to compile myPS_v0.c

6. Modify myPS.c in order to make the child process, after writing the message, execute the function mute_to_PS. This function will mutate to the ps program. Add the code of the mute_to_PS function:

```
/* Execute the command ps –u username using the system call execlp */
/* Returns: the error code in case it couldn't be mutated */
void mute_to_PS(char *username)
{
  execlp("ps", "ps", "-u", username, (char*)NULL);
  error_and_exit("The ps mutation has failed", 1);
}
```

⚠️ **QUESTION 20. In which cases will any code that appears after the execlp system call be executed (In any case /In case execlp is executed correctly / In case execlp fails)?**

Consult the information of the executing processes: myPS.c

The objective of this session is learning how to use the /proc pseudo-filesystem to consult some information about the execution of processes. For it we will use the codes myPS.c (mutates) and myPS_v0.c (doesn't mutate) that you have developed in the previous section.

7. For this exercise we will use two Shell terminals. In one of the terminals execute myPS_v0 with only one username as a parameter. In the second terminal go to the directory /proc and check that several directories appear there that match with the PID of the processes. Access the parent's and the child's directory and look the extended information (permissions, owner, etc) of the files of the directory.

> **QUESTION 21.** Which directories are inside the /proc/PID_PARENT? Which ls option have you used?
>
> **QUESTION 22.** For the parent process, write down the content of the files status and cmdline. Compare the content of the file environ with the result of the command env (for example the variables PATH and PWD). What's the relation between them? Search in the contents of the status file the state in which the process is and write it down in the answers file. Also write the CPU time that it has consumed in user mode (look it in the stat file of the process).
>
> **QUESTION 23.** In the case of the child process, which files do the files cwd and exe point to? What do you think is the meaning of cwd and exe?
>
> **QUESTION 24.** In the case of the child process, can you show the contents of the environ, status and cmdline files of the child process? What is its status?

8. Repeat the experiment using the program myPS.c instead of myPS_v0.c and answer again the questions for the child process. Observe the differences between both code versions. Remember that in v0 the child process didn't mutate. What's the status of the child process now? (consult it in the status file)

> **QUESTION 25.** In the case of the child process, which files do the files cwd and exe point to? What do you think is the meaning of cwd and exe? What are the differences with the previous version? What is the reason of these differences?
>
> **QUESTION 26.** In the case of the child process, can you show the contents of the environ, status and cmdline files of the child process? What is its status? What are the differences with the previous version? What is the reason of these differences?

Sequential execution of the children:myPS2.c

The objective of this session is to practise with the waitpid system call and understand how it influences the concurrency of the created processes. In particular you are going to use it to create processes that execute in a sequential way.

This call is used to make the parent process wait until their children end, to check their finalization status and to make the kernel free the data structures that represented them internally (PCBs). The place where this wait is executed is crucial to generate a sequential (all the children processes are created and executed one after one) or concurrent (all the children processes are created and executed in a potentially parallel way, depending on the architecture

where they are executed) code. In this section we want to do a **sequential code.** For it we will use the waitpid system call between a process creation and the next, in a way that we ensure that we won't have two children processes executing at the same time.

```
//Example sequential schema
for (i=0;i<num_children;i++){
    pid=fork();
    if (pid==0) {
        // children code
        exit(0);// Only if the child doesn't mutate and we want it to end
    }
    // We wait until it ends before creating the next one
    waitpid(-1,NULL,0);
}
```

9. Create a copy of myPS.c, called myPS2.c, with which you will work in this exercise. Modify, also, the makefile to be able to compile and link myPS2.c.

10. Modify myPS2.c in order to, instead of receiving a parameter (username), make it able to receive N. Make the main program create a process for each username (each parameter) and execute them in a sequential way. You can eliminate the infinite loop at the end of the parent's execution.

Concurrent execution of the children: myPS3.c

In this section we will continue working with the system call waitpid. Now you will use it to create a concurrent execution schema.

```
//Example concurrent schema
for (i=0;i<num_children;i++){
    pid=fork();
    if (pid==0) {
        // children code
        exit(0);// Only if the child doesn't mutate and we want it to end
    }
}
// We wait for all the processes
while (waitpid(-1,NULL,0)>0);
```

We will also check the possible effects that concurrency can have over the result of the execution.

11. Create a copy of myPS2.c, name it myPS3.c, with which you will work in this exercise. Also modify the makefile to be able to compile and mount myPS3.

12. Modify the program myPS3.c in order to create the children in a concurrent way. To be able to consult the information of the processes, make the parent wait for a keystroke after executing the waitpid loop. To wait for a keystroke you can use the read system call (`read(0,&c,sizeof(char))`, where c is a char).

13. Execute myPS3 with several usernames and leave the parent blocked after the waitpid loop. In another window check that none of the children processes has a directory in /proc. Check the status in which the parent is.

QUESTION 27. Check the status file in /proc/PID_PARENT. In which status is the parent process?

14. To check the effects of a concurrent execution, and to see that the system scheduling generates different results, execute several times the command myPS3 with the same parameters and save the outputs in different files. Check if the order in which the ps's are executed is always the same. Think that several results can be identical.

⚠ **QUESTION 28. What have you done to save the output of the executions of myPS3?**

Parameters passing to the executables through execlp

The aim of this section is to understand the relation between the parameters received by the main of an executable and the parameters passed to an execlp system call. Remember: the main routine receives two parameters: argc, of type integer, and argv of type string array. The operating system fills the argc parameter with the number of elements that argv has, and fills argv with the parameters that the user indicates in the execlp system call (which has to be of type string). To follow the same convention that the Shell uses, the first parameter has to be the name of the executable to which it's going to mutate.

15. The listaParametros program is a valid version of the code inside listaParametros.c that we used in session 2. It simply shows on screen the parameters that it receives. Write a program parsExec.c that creates 4 children processes. Once created, the parent process only has to wait until all children have finished. Each children process only has to mutate to the listaParametros executable, each of them passing different parameters, in a way that, when executing parsExec, on screen you have to see the following messages (the order of the messages doesn't matter and can be different for every execution). Each group of messages corresponds to the output of a mutated process:

```
The argument number 0 is listaParametros
The argument number 1 is a
The argument number 2 is b

Usage:listaParametros arg1 [arg2..argn]
This program writes in its output the arguments it receives

The argument number 0 is listaParametros
The argument number 1 is 25
The argument number 2 is 4


The argument number 0 is listaParametros
The argument number 1 is 1024
The argument number 2 is hola
The argument number 3 is adios
```

16. Modify the makefile to be able to compile and link parsExec.c.

Session 4: Process communication

Previous work

1. Objectives

During this session we will introduce the management of events between processes as a mechanism of communication and synchronization between processes. We will also work some aspects related to the concurrency of processes.

2. Abilities

- Be able to reprogram/wait/send events using the Unix interface between processes. We will practise with: sigaction/sigsuspend/alarm/kill.
- Be able to send events to processes using the command kill.

3. Previous knowledge

The signals or events can be sent by other processes or by the system itself automatically, for example when a child processes ends (SIGCHLD) or ends an alarm timer (SIGALRM).

Each process has a table within its PCB where it's written, for each signal, which action has to be realized, which can be: **Ignore the event** (not all events can be ignored), **do the by defect action** that the system has programmed for this event, or **execute a function that the process has defined** explicitly using the sigaction system call. The signal treatment functions must have the following header:

```
void function_name( int number_of_the_signal_received );
```

When the process receives a signal, the system executes the associated treatment to that signal for that process. In the case the treatment is a user-defined function, the function receives as a parameter the signal number that has caused its execution. This allows us to associate a same function to different types of signals and do a differentiated treatment inside this function.

4. Previous work guide

With some examples we will see, in a simple way, how to program a new event, how to send it, what happens with the signal programing table when doing a fork, etc. For this session you have to review the concepts explained in the theory lectures about processes and signals. Read the pages of the manual related with the topic that are indicated in the following table.

To read in the man	Basic description	Options

`sigaction`	Reprogram the action associated to a particular event	
`kill (system call)`	Send an event to a particular process	
`sigsuspend`	Blocks the process that executes it until it receives a signal (the signals which treatment is to be ignored don't unblock the process)	
`sigprocmask`	Modifies the mask of blocked signals of the calling process	
`alarm`	Program the sending of a SIGALRM signal in N seconds	
`sleep`	C library function that blocks the process during the time passed as a parameter	
`/bin/kill (command)`	Send an event to a process	-L
`ps`	Shows information about the processes in the system	-o pid,s,cmd,time
`waitpid`	Waits for the finalization of a process	WNOHANG

Download the s4.tar.gz file and unzip it (tar zxvf S4.tar.gz). Compile the files and execute them. In the README file you will find a brief description of what the files do and how to compile them. Try to understand them and how the system calls that we will practise before the laboratory are used. The files are commented in a way that you can understand what is being done.

The signals that we will basically use are SIGALRM (alarm, timer), SIGCHLD (end of a child process), or SIGUSR1/SIGUSR2 (without predefined meaning, the programmer can use them with the meaning that suits him/her). Take a look at the by default actions of these signals.

Make the following tests before going to the laboratory. Create a file named entrega.txt and write in it the answers to the following questions (indicating their number).

About alarm1: reception of different types of signals and sending events from the terminal

1. Execute alarm1 in a terminal and observe its behaviour. What happens after 5 seconds?

2. Execute again alarm1 and before the timer ends send it a SIGKILL signal. To do it, in another terminal execute the ps command to obtain the pid of the process and next use the kill command with the option –KILL to send the SIGKILL signal to the process. Is the behaviour the same as if you wait for the arrival of SIGALRM? Do you receive a different message in the terminal?

Remember that the Shell is in charge of creating the process that executes the commands that you introduce and of recovering the finalization status of that process. The Shell pseudo-code to execute a foreground command is the following:

```
while (1) {
        command = read_command();
        pid_h = fork();
        if (pid_h ==  0)
                execlp(command,….);
        }
        waitpid (pid_h, &status, 0);
}
```

3. Which process is in charge of showing the messages that appear on screen when the alarm1 process dies? With which system call the shell recovers the finalization status of the process that executes alarm1 when it ends?

4. Is it necessary the exit at the end of the code of alarm1? Is it ever executed?

About alarm2: any signal can be sent through shell commands.

1. Execute alarm2 in a terminal. Open another one, find out the pid of the alarm2 process and use the kill command to send the **–ALRM** signal several times. What behaviour can you see? The time control behaved as the code pretended?

2. Can the associated treatment of any signal be modified?

3. Search in the man (man alarm) the return value of the alarm system call and think how we could fix the code to detect when SIGALRM reaches us without being related to any timer.

About alarm3: the signals programming table is inherited.

1. Who receives the SIGALRM: the parent, the child, or both? How did you check it? Modify the "funcion_alarma" function to make that the message also writes the pid of the process, in a way that we can easily see which process receives the signals.

About alarm4: the SIGALRM are only received by the process that generates them

1. How many SIGALRMS programs each process? Who receives the alarm: the parent, the child, or both? What happens to the parent? How did you check? Modify the code in such a way that the first alarm is programmed by the parent before the fork (and not the child), and observe how the child is kept waiting in the sigsuspend call.

About ejemplo_waitpid: sending signals using system calls, managing the finalization status of the children and deactivation of a timer.

1. Analyse the code of this program and execute it. Observe that inside the code of the children processes a timer is used to fix their execution time to a second.

2. Modify the code in order to make the parent process to show a message describing the cause of death of the child through its standard output when exiting the waitpid (successfully finished or dead because it received a signal).

3. Complete the program to limit the max waiting time of the waitpid at each iteration of the loop: if no child has finished its execution after 2 seconds, the parent has to send a SIGKILL to each of its alive children. In case some child finishes on time, the parent will deactivate the timer and will show a message indicating how much time was left before the sending of

the SIGALRM. To deactivate the timer you can use the system call alarm passing as a parameter a 0. NOTE: have in mind that in order to do that you need to store all the pids of your created children and register them as they die.

TO DELIVER: previo04.tar.gz

#tar czfv previo04.tar.gz entrega.txt

5. Bibliography

- Slides of Topic 2 (Processes) of SO-grau.
- Chapter 21 (Linux) de A. Silberschatz, P. Galvin y G. Gagne. Operating System Concepts, 8th ed, John Wiley &Sons, Inc. 2009.

Exercises to do in the laboratory

➢ For all the exercises, it is assumed that all the exercises that are asked will be tested and that the makefile will be modified in order to be able to compile and mount the executables.

➢ Answer in a text file named entrega.txt all the questions that appear in this document, indicating for each question its number and answer. This document must be delivered through Racó. The questions are highlighted in bold and marked with the following symbol:

➢ **To deliver: sesion04.tar.gz**

#tar zcfv sesion04.tar.gz entrega.txt makefile ejemplo_alarm2.c ejemplo_alarm3.c ejemplo_signal.c ejemplo_signal2.c eventos.c eventos2.c signal_perdido2.c

About alarm2: Detecting which signal has arrived

1. Reprogram the SIGUSR1 signal and associate it to the same function as the alarm. Modify the function "funcion_alarma" in such a way that it updates the seconds in the case a SIGALRM signal arrives and writes a message in case a SIGUSR1 arrives. You should also modify the signal mask that is passed to the sigsuspend call. Check that the function works sending SIGUSR1 signals through the terminal using the command kill.

   NOTE: Remember that the function that attends the signal receives as a parameter the number of the received signal. In the previous work you have more details.

About alarm3: Signals and creation/mutation of processes

The aim of these exercises is that you see what we have studied in the theory lectures related to the management of signals and the creation/mutation of processes: when creating a process the child INHERITS the signal table of its parent. When mutating a process, its signal table is replaced by the default signal table.

2. Modify the code in such a way that the reprogramming of the SIGALRM signal (signal system call) is only done by the child. **OBSERVE** how before this modification, both the parent and the child (through inheritance) had the SIGALRM captured. After the change, the modification of the child is private, thus the parent has the by default action associated to the SIGALRM.

3. Modify the code in such a way that after programming the timer the child process mutates to execute another program. This program has to last more than 10 seconds in order to receive the SIGALRM signal before finishing (for example, you can implement a program that only executes an infinite loop). **OBSERVE** how when mutating the SIGALRM associated action is placed by default, since the signal table is reset.

About ejemplo_waitpid: We wait for the children to finish. Impact of the implementation of signals

The waitpid system call is commonly used with its blocking form (by default behaviour of waitpid). The process that executes it doesn't continue until some child process ends. However, there are some situations where it could interest us to continue with the execution of the parent process and at the same time control the end of the execution of its children. This is possible if we capture the signal SIGCHLD, which the system sends to a process when some of its children ends (it is ignored by default). In this exercise, as it is a bit unnatural, we will have to make an active wait in the parent process, given that otherwise it would end its execution before its children processes finish.

4. Create a program with the name ejemplo_signal.c. Modify this program in such a way that the parent process reprograms the treatment of the SIGCHLD. The treatment of SIGCHLD must execute the waitpid function without blocking the parent process. In this manner, the parent can control if any child ends and continue its normal execution. Recall that, in this case, you will have to add an active wait in the parent process so that it does not finish its execution. However, the most typical situation would be to use this option when the parent does have some other work to do. You can use a global variable to know if there still are child processes alive.

   CAREFUL!: While the treatment of a signal is being executed, the system will block the notification of new signals of the same type, and the process will receive them when exiting the routine. But, the operating system is only capable of remembering a pending signal of each type. Meaning, if while the treatment of a signal is being executed the process receives more signals, when ending the routine it will only have one notification

of each signal type and the rest will be lost. For this reason, the treatment routine of SIGCHLD before exiting must ensure that all the children that have died during the signal treatment have been waited for without blocking (remember that inside a signal treatment routine you must not ever block). Consult in the man the meaning of the option WNOHANG of the waitpid system call (man waitpid) and use it so the waitpid does not block.

5. Create a copy of ejemplo_signal.c called ejemplo_signal2.c. Modify this program so that the parent process, after all children processes have been created, sends to all of them a SIGUSR1 signal (whoes default action is to finish the process) and then continues incrementing the counter variable. In addition, after calling the waitpid system call, the parent will show the PID of the child together with the reason why this process has finished (WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG macros). What exit code is showed by each child process?

   - HINT: Consider that the sigsuspend() system call blocks the process until the reception of a signal that the process has captured, but if the signal arrives before that the process executes the sigsuspend(), it may block indefinitively. You should modify the signal mask of sigsuspend so that it captures the SIGUSR1.
   - Create a function that, using the macros mentioned above, writes the PID of the process that has finished and a message telling if the process has finished due to an exit call or a signal, and in each case the exit_status or signal_code. There is an example of how to use this macros in the slides of T2.

PROTECTION BETWEEN PROCESSES

The processes cannot send signals to processes that belong to other users. In the system there are several users created (so1, so2, so3, so4 and so5) with the password "sistemes" for all of them.

6. Execute the program alarm2 in a terminal and check its PID. Open a new session and change to user so1 (executing #su so1). Try to send signals to the process that is executing alarm2 from the session initiated with the user so1.

QUESTION 32. Can the user so1 send signals to the process launched by the user alumne? What error is displayed?

SIGNAL MANAGEMENT

The aim of this section is that you become capable of implementing from scratch a program that manages several types of signals using a treatment different from the by default one.

7. Make a program, called eventos.c, that executes an infinite loop and that has a global variable to store the time that a program has been executing (in seconds). This variable is managed through signals in the following way:
   - Each second it is incremented
   - If the process receives a SIGUSR1 the variable is set to 0

- If the process receives a SIGUSR2 it will write on screen the current value of the counter

All the signals of the program have to be attended by the same function. Send the signals from the terminal and check that it works correctly.

BY DEFAULT BEHAVIOUR

The reprogramming of a signal in Linux is maintained through all the lifespan of a process. For this reason, sometimes, it's necessary to force the by default behaviour of the signals in the case that we are not interested in processing more events.

8. Create a copy of eventos.c with the name eventos2.c. Modify the code in eventos2.c in such a way that the second time it receives a signal it executes the by default behaviour of that signal. HINT: consult in the sigaction manual page the constant SA_RESETHAND.

⚠️ **QUESTION 33.   Which message shows the Shell when the same signal is sent for second time?**

CONCURRENCY RISKS

When you program applications with various concurrent processes you cannot assume anything about the execution order of the instructions in the different processes. This also applies to the sending and reception of signals: we cannot assume that a process will receive a signal in a determined moment. This has to be taken into account when using the sigsuspend system call.

The program signal_perdido.c shows this problematic in an artificial way. This program creates a process that pretends to wait in a sigsuspend system call the reception of a signal. The parent process is the one in charge of sending the signal. The moment when the signal is sent depends on the parameter of the program: it's made immediately (parameter value 0) or it is delayed some time (parameter value 1).

9. Execute the program signal_perdido passing 1 as a parameter. Write down in the entrega.txt file the result that you got. Next, execute again the program with the parameter 0. Write down again the result that you got.

⚠️ **QUESTION 34.   Explain what causes the result in the executions of signal_perdido with 1 as a parameter and 0 as a parameter**

10. Modify the signal_perdido program (call it signal_perdido2.c) in such a way that, before the wait, provisionally blocks the reception of the signal usisng the sigprocmask system call, so that the signal cannot be lost. Recall unblocking the signal after the wait.

Session 5: Memory management

Previous work

Objectives
- Understand the relation between the code generated by the user, the logical space of the process that will execute this program and the physical memory space occupied by the process.
- Understand the difference between linking an executable to static or dynamic libraries.
- Understand how some basic commands, which allow the analysis of the memory usage of processes, work.
- Understand the functioning of the C library functions and simple system calls that allow the modification of the logical address space of the processes in runtime (dynamic memory).

Abilities
- Be able to link parts of a binary with its logical memory address space.
- Be able to differentiate the different regions of the memory and in which region each element of a process is located.
- Understand the effect of malloc/free/sbrk on the address space, in particular on the heap.
- Know how to allocate and free dynamic memory and its implications on the system.
- Be able to detect and correct errors in the memory use in a code.

Previous knowledge
- Understand the basic format of an executable.
- C programming: use of pointers.
- Be able to interpret and consult the available information about the system and the processes in the /proc directory.

Previous work guide
- Before the session, consult the man (`man command_name`) of the following commands. Concretely, for each command you have to read and perfectly understand: the SYNOPSIS, the DESCRIPTION and the options commented on the "Options" column of the table.

To read in the manual	Basic description	Options
gcc	C compiler	-static
nm	Command that shows the symbol table of the program (global variables and functions)	
objdump	Command that shows information about the object file	-d

/proc	Contains information about the system and the running processes	/proc/[pid]/maps
malloc	C library function that validates a logic memory region	
free	C library function that frees a logic memory region	
sbrk	System call that modifies the size of the data section	

- In the website of the course (http://docencia.ac.upc.edu/FIB/grau/SO) you have the S5.tar.gz file that contains all the source files that you will use in this session. Create a directory in your machine, copy in it the S5.tar.gz file and unzip it (tar zxfv S5.tar.gz).
- Create a text file called previo.txt and answer in it the following questions.
- Practise the usage of nm and objdump with the following exercises applied to the following code (which you can find in the file mem1_previo.c):

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
char buff[100];
void suma (int op1, int op2, int *res){
      *res = op1 + op2;
}

int j;
main(int argc, char *argv[]){
      int i;
      int s;
      i=atoi(argv[1]);
      j=atoi(argv[2]);
      suma(i,j,&s);
      sprintf(buff,"suma de %d y %d es %d\n",i,j,s);
      write(1,buff, strlen(buff));
}
```

- Use the nm command on the executable and write down in the "previo.txt" file the address assigned to each of the variables of the program. Search in the man (and write down in previo.txt) the symbol types that nm shows. For example, the symbols tagged with a "D" are in the data section (global variables). **Is it possible to know the addresses of the variables "i" and "j" with the nm command? Why? In which memory zone are these variables allocated? Search also the address assigned to the "suma" function.**

- **Modify the previous program (call it mem1_previo_v2.c)** in order to make the variable "s" defined in the main program as a pointer to int (int *s) and assign it a valid address before calling the function "suma". For it, use the malloc function to allocate enough space to store an int and assign the address of that region to the variable. Adapt the rest of the code to make it work after this change.

- Use the gcc command to compile the file mem1_previo.c linking it with static libraries. **Indicate in the file "previo.txt" which commands have you used.**

- The program mem2_previo reads from the keyboard the number of elements of an int vector, initializes this vector with this number of elements and it sums them. Modify this program (call it mem2_previo_v2.c) in order to, instead of using a static vector, use dynamic memory. For it, declare the variable "vector" as a pointer to int. After reading from the keyboard the number of elements that the vector needs to have, the program has to use the sbrk system call to allocate a memory region where those element can be placed and assign the address of this region to the vector variable.

- Compile statically both mem2_previo and mem2_previo_v2. For both programs do the following: execute the program and before pressing the Return to finish it, from another terminal, access the /proc/PID_of_the_process directory that contains the information about the process and observe the maps file. This file contains a line for each allocated memory region. The first column indicates the initial and final address of the region (in hexadecimal). The difference between both values gives us the size of the region. Search in the man page of *proc* the output format of the maps file and the meaning of the rest of fields. Write down in the previo.txt file the total size of the heap region and data for the following number of elements of the vector: 10 and 40000. **The heap region is tagged as [heap] but the data region is tagged with the name of the executable. You will have to deduce which it (there's more than one) is for the permissions of the region.** Is there any difference between the different values of the executions in both programs?

- The file mem3_previo.c contains some code that has an error in the use of pointers. Execute the program and check the error that appears. Modify the code (in the file mem3_previo_v2.c) in order to, when the program generates the SIGSEGV signal type (segmentation fault), make the signal handler show an error message on screen and ends the execution of the program.

- **TO DELIVER: previo05.tar.gz**
  - **#tar zcfv previo05.tar.gz previo.txt mem1_previo_v2.c mem2_previo_v2.c mem3_previo_v2.c**

Bibliography
- Chapter 8 (Main Memory) de A. Silberschatz, P. Galvin y G. Gagne. Operating System Concepts, 8th ed, John Wiley &Sons, Inc. 2009.

Exercises to do in the laboratory

➢ For each new code file created you need to modify the Makefile to compile it and mount the executable.

➢ Answer in a text file named entrega.txt all the questions that appear in this document, indicating for each question its number and answer. This document must be delivered through Racó. The questions are highlighted in bold and marked with the following symbol:

➢ **TO DELIVER: sesion05.tar.gz**

o **#tar zcfv sesion05.tar.gz entrega.txt Makefile mem1_v2.c mem1_v3.c mem2_v2.c**

Address space of a process and dynamic and static compilation

The objective of this section is double: on one hand, understand how the data and program code is organized in the address space of the process; on the other hand, understand how the type of compilation (static or dynamic) influences in both the executable and the address space. For it, we use commands that allow us to analyse the executables and observe the information about the address space store in /proc.

The file mem1.c contains a program that allocates memory during execution. Analyse its content before answering the following questions.

1. Compile the program linking it with the dynamic libraries of the system (compilation by default) and save the executable with the name mem1_dynamic. Execute the command nm on the executable.

⚠️ **QUESTION 35. Which variables appear on the output of nm on mem1_dynamic? Which direction occupies each of them? Indicate to which region each variable belongs according to the output of nm and the type of variable (local or global).**

2. Compile now the program, linking it with the static libraries of the system, and save the executable with the name mem1_static. Compare now the executables mem1_dynamic and mem1_static on the following way:

   a. Using the command nm to see the symbols defined inside each executable.
   b. Using the command objdump with the option –d to see the translated code.
   c. Comparing the sizes of the resulting executables.

⚠️ **QUESTION 36. For both executables, static and dynamic compiling, observe their size and the output of the commands nm and objdump. What are the differences between the files?**

3. Execute in background the two versions of the executable and compare the contents of the maps file of the /proc of each of the processes.

⚠️ **QUESTION 37. Observe the contents of the maps file of the /proc of each process and write down for each region the initial address, the final address and the associated tag. What's the difference between the maps of each process?**

⚠️ **QUESTION 38. To which region of the ones described in the maps belongs each variable and each zone allocated with malloc? Write down the initial address, the final address and the name of the region.**

Dynamic memory

The objective of this section is to understand the effects, in the address space of a process, of the dynamic memory allocated in run time, depending on the used interface: sbrk or malloc. Remember that sbrk is a system call and it simply increases or decreases the size of the heap in

X bytes (what the user asks). Malloc/free are C library functions, which offer a much more sophisticated management of the dynamic memory. The C library reserves a big zone in the heap and uses it to manage the petitions of the user without having to modify the size of the heap. For it, it uses data structures (also stored in the heap) which are used to record the occupied and free zones.

4. Execute in the background the program mem1 passing as parameters 3, 5 and 100 (three different executions). Observe the maps of the /procs to compare the size of the memory zones in function of the number of allocated regions with malloc. Analyse the results in function of the theory explained in the lectures about the functioning of the malloc.

QUESTION 39. For each execution write down the number of mallocs done and the initial and final address of the heap shown by the maps file of the /proc. Does the size change depending on the input parameter? Why?

5. Create a copy of the file mem1.c called mem1_v2.c.
6. Add a free at the end of each iteration of the region allocation loop in mem1_v2.c and execute it again in the background (with the parameter 100), observing the maps file of the /proc and the size of the zone that hosts the regions reserved by the malloc.

QUESTION 40. What is the size of the heap in this case? Explain the results.

7. Create a copy of mem1.c called mem1_v3.c. Modify the code of mem1_v3.c substituting the function call malloc for the system call sbrk and execute it on the background, using the same parameters of section 4 (3, 5 and 100). Observe the maps file of the /proc and the size of the heap that hosts the regions reserved by the sbrk.

QUESTION 41. For each execution write down the number of mallocs done and the initial and final address of the heap shown in the maps of the /proc. Does the size of the heap change regarding the observations of question 6? Why?

8. Modify the previous code to implement (besides the reservation) the release of the memory using sbrk (system call) instead of malloc (C library).

Incorrect accesses

The mem2.c file contains a code that has an error in the use of a pointer (different from the error present in the mem3_previo.c seen in the previous work). Execute the program and check that it really doesn't work.

9. Modify the code (call it mem2_v2.c) so that when the process receives a SIGSEGV type signal a message is printed through the standard output indicating:
   a. The address of variable p.
   b. The value of the variable (address pointed by the pointer).
   c. The address where the heap of the process ends (last valid address).

   NOTE: Print only the 3 values that we ask, if you copy directly the write of the code you can be repeating the same error that causes the signal.

QUESTION 42. What is the error of the program? Why the program doesn't fail in the first iterations? Propose an alternative to this code that avoids the generation of the exception, detecting, before it happens, that we are going to make an access outside the address space.

Session 6: Performance Analysis

Previous work

1. Objectives

The objectives of this session are: 1) understand the implications of the process and memory management in the performance of a multiuser system, and 2) measure the execution time of a program regarding to the system load and the priority change of the processes.

2. Abilities

- Understand how the different planning and priority policies influence in the user's programs.
- Understand how the system load influences the execution times of the programs.
- Relate the performance of a system with the memory use of the process.
- Be able to obtain information about the running processes through the /proc.
- Be able to see the list of processes of all the users to detect possible problems in the system.

3. Previous work guide

Before the session, consult the man (`man  command_name`) of the following commands. Concretely, for each command you have to read and perfectly understand: the SYNOPSIS, the DESCRIPTION and the options commented on the "Options" column of the table.

To read in the man	Basic description	Options
nice	Execute a program modifying the planning priority	
Uptime	Shows how much time has been the system on and the average load	
W	Shows who is connected and what is s/he doing	
/proc	Pseudo-file system that offers information about the kernel's data	cpuinfo, /proc/[pid]/stat
time	Execute a program and measure its execution time	
vmstat	Shows statistics about the usage of memory	delay
top	Shows information about the system and the running processes	Interactive command: f

The process scheduling policy used by default in current Linux releases is called Completely Fair Scheduling and it is an evolution of Round Robin with priorities. Processes at Ready status (candidates to use the CPU) are sorted by the CPU spent time. Thus, the next to occupy the CPU is the process with least spent time. CPU usage is granted either till the process releases the CPU (either it blocks or it ends) or till it reaches a variable period of time, unlike Round Robin using a static quantum. This amount of time is calculated based on the period of time that would be

granted in an ideal scheduling (elapsed time divided by the number of running processes) and its priority. The users can **lower** the priority of their processes executing them with the command **nice** (only root can raise the priority of any process). The priority is defined using a number. Lower numerical values of the priority mean that the process has more priority.

The **top, uptime and w** commands show us information about the system. These commands are useful for doing a simple control of the system load.

- The **top** command shows us all the processes of the system and details such as the %CPU consumed by them, the priority they have, the user, etc. Moreover, it shows us global information of the system such as the average load. This value is not a one-time value, but an average of the last 1, 5 and 15 minutes. Htop is a slightly different command that shows a more visual and complete information about what is going on in every single processing unit of the system (this command is not installed by default in the OS image, but it can be easily installed executing the command line "**sudo apt-get install htop**")
- The **w** command shows us really summarized information about who is connected.
- The **uptime** command offers statistical information of the system.

Figure 4 Output example of the top command

The previous image shows us an example of the output of the top command. In the upper part there's an overview of the system and then the list of tasks. In this case we have highlighted the load of the system and the two processes that we were executing. The **PR** column shows the priority of the processes. In Linux a high value in priority indicates less priority. We can see how the process with PR=25 receives more %CPU (66.9) than the process with PR=35 (32.0).

When interpreting the information that top and uptime give us is necessary to have in mind the number of processing units (CPU's, cores, etc) that we have in the machine, since this determines the number of programs that we can have running simultaneously. For example, if we have 2 cores could happen that we have 2 processes at the same time consuming an 80% of the CPU. The file **/proc/cpuingo** contains information about the CPU's that we have in our machine. For every single CPU, among others, it shows the available number of cores as well as siblings, also known as the number of processing units. If there are more siblings than cores (it should be a multiplier), it means the CPU supports *hyperthreading technology* (Intel's terminology for multithreading). This involves it is possible to exploit a given level of parallelism in a single core. The command lstopo shows a more visual summary of this information. This command is not installed in the OS image, but it can be easily installed running the command line "**sudo apt-get install hwloc**".

1. Consult the man of the *proc* (man proc) and indicate, in the "previo.txt" file, in which file of the /proc, and in which field of the file, does the number of total page faults of the process (minor page faults + major page faults in Linux).

   a. **NOTE:** In Linux, it is possible that a page fault doesn't imply a disk access thanks to some memory optimizations. In this case it is considered as a "soft" or minor page fault, since it's less costly than those that imply going to the disk.

2. Execute the command vmstat in such a way that the output is shown each second. What option have you used? In which columns is the swap-in and swap-out amounts displayed? Answer the questions in the file "previo.txt".
   a. **NOTE:** Swap-in and swap-out is the amount of memory that has been brought/sent from/to disk.

3. Consult the manual page of the **time** command (man time). The command **/usr/bin/time** is used to execute a program measuring its execution time. This program shows 3 time values by default: time used in user mode (*user),* time used in system mode (*system*) and total time (*elapsed*). During the session we will use the total time of execution.

   a. **NOTE:** the Bash command-line interpreter has an internal command that is also called time and it is the one that will be executed if we don't put the complete path of our command.

4. **Familiarize with the environment you will use in the session:** download the S6.tar.gz file and unzip it (tar zxvf S6.tar.gz). In this session we will provide you with several **scripts** to facilitate the execution of the programs. A script is nothing more than a text file that contains a set of commands that can be interpreted by a Shell. The first line of a script has to indicate the Shell that you want to use to interpret the contents of the file. This is important because besides commands, in the scripts you can use control structures (conditionals, loops, ...) and each Shell defines its own syntax for these structures. Scripts can be executed like any other program, but since they are text files and these files are created without execution permission, it is necessary to check if that permission is activated and if it isn't we have to activate it using the chmod command (see session2).

5. In the S6.tar.gz package you have the Fibonacci.c file that contains a calculus program and we will use it to see the impact of the system load and priorities in the execution of the program. Use the makefile to compile the program and execute it with different parameters to see how it works.

6. **Use the /usr/bin/time command** to measure the execution time of Fibonacci if the parameters 10, 20, 30, 40 and 50 are passed and write down those times in the previo.txt file.

7. In the package you also have two scripts to automate the execution of several Fibonacci instances. The FIB script receives as a parameter how many Fibonacci programs we want to launch and executes them **concurrently** (in the background), measuring the time it takes for each of them to finish. The script BAJA_PRIO_FIB also executes concurrently the number of Fibonacci programs that are passed as a parameter and measures their execution time, but in this case it uses the **nice** command to execute them with less priority. For example, if we execute the command:

./FIB 2

Is equivalent to executing the following commands consecutively:

/usr/bin/time ./fib 45&
/usr/bin/time ./fib 45&

Write down in the previo.txt file with which nice value is the Fibonacci executed from the script BAJA_PRIO_FIB. To find out, consult the manual page of the behaviour of nice.

8. You also have a script that allows execute an instance of Fibonacci in a particular processing unit: FIB_affinity

Finally, we also give you a file s6.ods which is an OpenOffice spreadsheet that you will fill through the session.

- **TO DELIVER: previo06.tar.gz**
    #tar zcfv previo06.tar.gz previo.txt

4. Bibliography
- Slides of Topics 2 and 3 (Processes and Memory) of SO-grau
- Chapters 5, 8 and 9 (Scheduling, Main Memory y Virtual Memory) de A. Silberschatz, P. Galvin y G. Gagne. Operating System Concepts, 8th ed, John Wiley &Sons, Inc. 2009.

Exercises to do in the laboratory

- ➢ Answer in a text file named entrega.txt all the questions that appear in this document, indicating for each question its number and answer. This document must be delivered through Racó. The questions are highlighted in bold and marked with the following symbol:
- ➢ **To deliver: sesion06.tar.gz**

    #tar zcfv sesion06.tar.gz entrega.txt s6.ods

PERFORMANCE OF THE PROCESS MANAGEMENT

Session setup

For this session, we are going to use a virtual machine that emulates the execution environment of the SO laboratories. You can download the virtual machine image from https://softdocencia.fib.upc.edu/software/Ubuntu-2021.zip.

Although all the exercises in this session could be performed on a physical machine with a native Linux installed, using this virtual machine we get all the students working on a similar environment, which is independent of any hardware upgrade on the laboratories.

There are two useful commands for this session and they are not installed in the current OS image: lstopo and htop. Firstly, we have to install them using the following command lines:

#>sudo apt-get install hwloc

#>sudo apt-get install htop

How many CPU's or cores do I have?

Execute the command **lstopo** to find out how many CPUs, cores and hardware threads (also known as "processing unit" or "PU") are in the host. This information will help you to interpret the results about CPU consumption of the processes. In the remaining documentation of this lab session we use the variable NTHREADS as the number of hardware threads of the host.

> **QUESTION 43. Write down in the entrega.txt file the number of processing units (hardware threads, cores and CPUS) that your machine has.**

- Access to the file /proc/cpuinfo and check lstopo shows the same information.

Execution of an "alone" application. Measuring the time.

- For this exercise we will use 2 terminals. In one of them execute the top command and in the other execute 5 times sequentially the following command (when one finishes, launch the next one). Remember that since the processes are executed in the background, you have to wait until the messages are displayed, it is not enough to wait for the prompt:
    - # ./FIB 1
- Check in the output of the htop command that each execution receives approximately all the CPU. Calculate the average, maximum and minimum time (real time) of the 5 executions.

> **QUESTION 44. Write down the average, maximum and minimum time for an instance in the spreadsheet "impacto concurrencia" from the attached file s6.ods.**

Multiprocess environment impact. Influence of affinity, workload, execution time and priorities.

- We are going to use again two terminals: one to execute the command htop; and the other to launch concurrent instances of fibonacci. We are going to do two experiments with different workloads. Monitor the htop output, especially how the workload is distributed in the CPU. Observe how it is attempted to perform a "fair" distribution and how the execution time of all processes is incremented at the same rate, To properly interpret these results take into account the number of PUs of your host.
  - In the first experiment we launch less instances than the total number of hardware threads. Thus, we run the command line "./FIB ninstances", where ninstances = (NTHREADS/2)+1. Run this experiment only once.
  - In the second experiment, we are going to launch more instances than the total number of threads. Thus, we execute again the command line "./FIB ninstances", but now ninstances = NTHREADS/2. This test is run only once.

- Check that the priorities of the processes are all the same.

> **QUESTION 45. Calculate the average, maximum and minimum time of the executed Fibonacci in every experiment. Write down the results in the spreadsheet "impacto concurrencia" of the s6.ods file.**

- Linux allows launching processes bound to a particular PU to prevent contention or interferences with other processes. We are going to use the command taskset, that allows binding processes to PUs, to measure such impact of potential interferences. In this case, we are going to do two experiments running the script FIB_affinity, that uses taskset to bind the execution of Fibonacci to a particular PU. In fact, it receives the PU identifier as input parameter and then launches one instance in such PU (the command lstopo shows what are the ids of the PUs and how are spread out in the CPU). The experiments are the following:
  - We are going to launch two instances of Fibonacci at a time, but in two different cores. Thus, execute, from two different terminals, the command line "./FIB_affinity" with the id of PUs from different cores as input parameter each.
  - We are going to run two instances in the same core. Launch in two different terminals the command line "./FIB_affinity" with the id of PUs of the same core each.

> **QUESTION 46. Write down the average time for a single instance in the spreadsheet "impacto afinidad" of the s6.ods file. Also introduce in the same spreadsheet the time required by a single instance, taken from the previous exercise.**

Multiprocess and multiuser environment impact

The scheduling policy of Linux allows to group processes according to different criteria and tries to distribute the CPU across groups. The following experiments will help to find out the default criteria used in the lab hosts.

Grouped by user?

For this experiment we will use 3 terminals. Keep in execution the htop command in a terminal.

- Executions by other users in other terminals:
  - In another terminal use the command "su" to change into another user (e.g. "so1") and then since that user launches Fibonacci instances to trigger significant workload:

    #su so1

    #./FIB (NTHREADS+2)
  - At the same time, in another terminal and with the initial user ("alumne"), execute 5 times sequentially the command ./FIB 1.
- Executions of the same user in different terminals. Now we repeat the executions with the same user (e.g. alumne): in one terminal "./FIB (NTHREADS+2)" and in the other 5 times, in a sequentially manner, the command line "./FIB 1".

Grouped by process hierarchy?

For this experiment we only need two windows: one to launch the command htop and the other to launch Fibonacci. Actually, run in the same terminal processes of other users. To do this, the initial user has to execute the following command line:

> # su -c './FIB (NTHREADS+2)' so1

This command launches the script FIB as it was the user "so1".

Then, execute 5 times, in a sequential manner, the command line "./FIB 1".

> ⚠️ **QUESTION 47. How is the execution time of the process affected? Write down the average time of the sequential executions of the command line "./FIB 1" in the different rows of the spreadsheet "detección grupos".**

> ⚠️ **QUESTION 48. How much %CPU has been assigned to each process in every case?**

> ⚠️ **QUESTION 49. The allocation has been by process hierarchy or by user?**

Priority impact on the execution time

The nice command allows lessening the priority to the processes of a user in such a way that more CPU is allocated to the higher priority processes. Let's check how the priority influences in the CPU distribution.

- Keep the execution of the htop command in a terminal and execute simultaneously, with the same user, the commands ./FIB 1 and ./BAJA_PRIO_FIB 1. Measure the execution times of each Fibonacci. Repeat the experiment 5 times and write down in

the spreadsheet "impacto prioridad" of the file s6.ods the average execution time for FIB and BAJA_PRIO_FIB in the row "1 y 1".

- Check, using the htop command, how the different priorities have been assigned to each one and how a %CPU distribution similar to the one in the previous work is observed.
- Repeat the previous experiment, launching simultaneously ./FIB 1 and ./BAJA_PRIO_FIB NTHREADS+2. Write down the execution time in the row "1 y NTHREADS+2", for these experiments.

⚠️ **QUESTION 50. How is the FIB execution time affected with respect to the number of instances of BAJA_PRIO_FIB?**

⚠️ **QUESTION 51. Which %CPU has been assigned to each process with 2 and NTHREADS+2 instances of BAJA_PRIO_FIB?**

However, the priorities only affect the CPU sharing if the processes that compete for it have different priorities. To check it, execute in a terminal the top command and in another terminal launch the command ./BAJA_PRIO_FIB NTHREADS+2. Measure the average execution time and check that this time is similar between them. Now execute the command ./FIB NTHREADS+2 several times and check that even though they are executed with higher priority than in the previous case, the execution time of each of them is similar to the time they had when executed with low priority.

MEMORY MANAGEMENT PERFORMANCE

The swap area

Analyse the code that contains the file mem1.c and compile it. Do the following exercises:

1. Using the free command, consult the amount of physical memory, **in bytes,** of the machine (assume that the value is F bytes, we will use F to refer to this value in the rest of the document). Consult in the man the necessary option for the free command to get the information in bytes.

⚠️ **QUESTION 52. How much physical memory has the system (F) in bytes?**

2. **Execution with only one process:** Use two different shells. Launch the vmstat command in the first one in order to execute it periodically (for example, each second) and in the other shell launch the program mem1 with the parameters indicated in the following table (the execution have to be sequential, meaning, don't launch the next mem1 before eliminating the previous one):

Execution	Region size	Number Processes	Number Iterations
Execution 1	F/4	1	1
Execution 2	F/4	1	4

For each execution observe the execution time of each access loop and the number of swap-in and swap-out that the vmstat reports. When the loop that counts the traversals of the vector (just before entering the infinite loop) finishes, consult for that process the number of total page faults that has caused its execution (minor page faults + major page faults), accessing to the stat file of its corresponding directory in the /proc. Observe how with only one process the swap mechanism of the system is not enabled.

QUESTION 53. Fill the following table and add it to the s6.ods file. In execution 2, why do you think the access time changes depending on the iteration number?

Execution	Page faults	Minimum access loop time	Maximum access loop time
Execution 1			
Execution 2			

3. **Execution with several processes:** Use again two different shells. In a Shell execute the top command, which you should configure to show at least the following information for each process: pid, command, virtual memory, resident memory, swap memory and number of page faults. To configure it, when you are executing top press the key "h" and a menu will be displayed with the different options of top that will allow you to sort the columns, choose certain columns, etc. In another Shell execute the program mem1 isong the following parameters (the executions have to be sequential, meaning, don't launch the next one until you haven't eliminated the current one):

Execution	Region size	Number Processes	Number Iterations
Execution 3	F/4	4	1
Execution 4	F/4	6	1
Execution 5	F/4	4	4

You don't need to wait until the finalisation of Execution 5. Once you have obtained the message of the first iteration for all the processes you can interrupt the execution.

QUESTION 54. Fill the following table and add it to the s6.ods file(sum all the page faults of all the processes of the same execution):

Execution	Sum of Page Faults
Execution 3	
Execution 4	
Execution 5	

Session 7:Input/Output management

Previous work

Objectives
- Understand how a device driver works
- Understand the link between logic device and specific operations
- Understand the concept of device independence
- Understand the mechanisms offered by the Shell for the redirection and communication of processes

Abilities
- Be able to create and eliminate new logic devices
- Be able to load and unload kernel modules
- Be able to understand the specific implementation of the read and write operations
- Be able to apply the advantages of device independence
- Be able to redirect the input and output of a process from the shell
- Be able to communicate two commands through anonymous pipes from the shell

Previous work guide
- Before the session, consult the man (`man command_name`) of the following commands. Concretely, for each command you have to read and perfectly understand: the SYNOPSIS, the DESCRIPTION and the options commented on the "Options" column of the table.

To read in the manual	Basic description	Options
mknod	Command that creates a special file	c,p
insmod	Command that inserts a kernel module	
rmmod	Command that unloads a kernel module	
lsmod	Command that shows the status of the loaded kernel modules	
sudo	Command that allows the execution of a command as root	
open	Open a file or device	
write	System call to write to a virtual device	
read	System call to read from a virtual device	

grep	Command that searches patterns in a file or in its standard input if no file is passed as a parameter	-c
ps	Command that shows information about the running processes	-e, -o
strace	Lists the system calls done by a process	-e, -c

- In the website of the course (http://docencia.ac.upc.edu/FIB/grau/SO) you have the S7.tar.gz file that contains all the source files that you will use in this session. Create a directory in your machine, copy in it the S7.tar.gz file and unzip it (tar zxfv S7.tar.gz).

- Answer the following questions in the file "previo.txt".

Input/output redirection, usage of the terminal and pipe logical devices

The file es1.c contains a program that reads from the standard input character by character and writes what it has read in the standard output. The process ends when the reading indicates that there are no more characters left to read. Compile the program and, next, execute it in the following ways to see how it behaves in function of the devices associated to the standard channels of the process:

1. Introduce data through the keyboard to see how they are copied on the screen. To indicate that there's no more data press ^D (Control+D), which is equivalent to an end of file in the keyboard reading. **Which value returns the read call after pressing ^D?**
2. Create a file with any text editor and run the program ./es1 associating using the shell its standard input to that file. Remember (see Session 1) that is possible to redirect the standard input (or output) of a command to a file using the shell special character < (or > for the output). **Write down the command used in the "previo.txt" file.**

Linux Shells allow the exchange of data amongst two commands using anonymous pipes (represented by the character '|' in the shell). The command sequence connected through pipes is called pipeline. For example, the execution of the pipeline:

command1 | command2

It makes the Shell create two processes (that execute command1 and command2 respectively) and connects them through an anonymous pipe. This pipeline redirects the standard output of the process executed by command1, associating it with the writing end of the pipe, and redirects the standard input of the process executed by command2, associating it with the reading end of the same pipe. In this way, everything that the command1 process writes in its standard output will be received by the command2 process when it reads it from its standard input.

For example, in the directory where you have unzipped the session files, execute the pipeline:

#ls –l |grep es

What is the result? What does the command 'grep es'?

Linking the two commands using a pipe is similar to doing the following combination:

ls –l > output_ls

grep es < output_ls

rm output_ls

3. Execute a pipeline that shows in the standard output the PID, the user and the name of all the bash processes that are being executed in the system. For it use the ps and grep commands combined with a pipe. **Write down the command in the file "previo.txt".**

Output format

In Linux, the input/output interface is designed for the exchange of bytes **without interpreting the content of the information.**

Meaning, the operative system is limited to the transfer of the indicated number of bytes from the indicated memory address, and it's the programmer's responsibility to interpret correctly those bytes, storing them in the appropriate data structures. When recovering a datum that has been stored in a file, the programmer should have in mind the format in which it was stored.

For example, if we want to read a number from the terminal (which is a device that only accepts ASCII, both in the input and the output), first you have to read the characters and then convert them to a number. Here you have an example assuming the user writes the number and then presses Ctr-D.

```
char buffer[64];
int num,i=0;
// When the user presses CtrlD the read will return a 0
// Since we don't know how many digits will the number have, we have to read it with a loop
while (read(0,&buffer[i],1)>0) i++;
buffer[i]='\0';
num=atoi(buffer);
```

For example, if we want to write the integer *10562* using its internal representation in the machine, we will be writing the number of bytes that an integer occupies (4 bytes if we use the type *int* in a 32 bits machine).

```
int num=10562;
write(1,&num,sizeof(int)); // If channel 1 is the terminal, we will only see garbage, since it only
                           //accepts ASCII.
```

To recover and interpret correctly its value we have to read this same number of bytes and store it in an int type variable.

```
//Reading example (without error control), assuming that in the datos.txt file there are integers we
//would do …
int fd, num;
fd=open("datos.txt",O_RDONLY);
read(fd,&num,sizeof(int)); // In this case the number of bytes to read is fix
```

If, otherwise, we want to write the same number as a character string (for example, to display it on screen), the first step is to convert it to a character string, in which each character is a digit. This implies that we will use as many bytes as digits the number has (in this example, 5 bytes).

```
char buff[64];
int num=10562;
sprintf(buff,"%d",num);
write(1,buff,strlen(buff));
```

4. The file es7.c contains a program that writes in the standard output an integer using the internal representation of the machine. Compile it and execute it redirecting the standard output to a file:

#./es7 > foo.txt

Write a program es7_lector.c that when executed in the following way:

./es7_lector < foo.txt

is able to read and correctly interpret the contents of that file.

5. In the case of the **terminal** logical device, the **device driver that manages it interprets all the bytes that have to be written as ASCII codes**, showing the corresponding character. The file es8.c contains a program that writes two times a number in its standard output: one using the internal machine's representation and another one converting the number to string. Compile and execute it. **How many bytes are written in each case? What observable differences appear on the screen?**

Logical device and physical device association

6. The subdirectory "deviceDrivers" contains the code of two simple device drivers: myDriver1.c and myDriver2.c. These device drivers only implement their initialization and finalization code and the specific device function of reading. Also, you have a makefile that compiles both device drivers (using the makefile that comes with the Linux distribution) and two scripts that will be in charge of loading and unloading these device drivers.

   Analyse the source files of the two device drivers and answer the following questions:

   a) **Which function is used to implement the specific read of the device managed by each device driver?**
   b) **What are the major and minor? What's their use? Which major and minor use the two device drivers?**

TO DELIVER: previo07.tar.gz
   i. **#tar zcfv previo07.tar.gz es7_lector.c previo.txt**

Bibliography

- Slides of Topic 4 (Input/Output) of SO-grau
- Chapter 13 (I/O Systems) de A. Silberschatz, P. Galvin y G. Gagne. Operating System Concepts, 8th ed, John Wiley &Sons, Inc. 2009.
- A. Rubini y J. Corbet. Linux device drivers, 2nd ed, O'Reilly & Associates, Inc., 2001 (http://www.xml.com/ldd/chapter/book/).

Exercises to do in the laboratory

➢ Answer in a text file named entrega.txt all the questions that appear in this document, indicating for each question its number and answer. This document must be delivered through Racó. The questions are highlighted in bold and marked with the following symbol:

➢ **TO DELIVER: sesion07.tar.gz**
   o **#tar zcfv sesion07.tar.gz entrega.txt es1_v2.c**   **es6_v2.c**

Redirection and buffering

In this first exercise we are going to work with the es1 file seen in the previous work. As we have already commented, this file contains a program that reads from the standard input character by character and writes them in the standard output. Next, do the following exercises:

1. Execute the ps command in a terminal. The column TTY of the output of ps will tell you which file inside the /dev directory represents the terminal associated to the shell that you are running. Open a new terminal and execute again the ps command. Observe now that the file representing the terminal is different.

2. Execute, from the second terminal, the program es1 redirecting its standard output to associate it with the file that represents the first terminal. Observe how what it is written in the second terminal appears on the first.

3. Write a Shell command that launches two processes that execute the es1 program and that are connected through an anonymous pipe. Introduce some characters using the keyboard and press ^D to end the execution of the processes.

4. Create a copy of the es1.c program calling it es1_v2.c. Modify the es1_v2.c program in order to, instead of reading and writing the characters one by one, do it using a buffer (char buffer[256]).

5. The *strace* command executes the program passed as a parameter and shows information about the sequence of system calls that it does. With the *–e* option you can specify to show only specific information about one system call and with the *–o* option you can specify that this information has to be stored in a file.

   We want to compare the number of *read* system calls that the two versions of the program (es1 and es1_v2) execute. For it, execute the following commands:

   # strace –o salida_v2 –e read ./es1_v2 < es2.c

strace –o salida_v1 –e read ./es1 < es2.c

> ⚠️ **QUESTION 55.** Write down in the "entrega.txt" file the commands that you have used in each section. Also hand in the file "es1_v2.c". What differences do you observe in the two executions of strace? How many read system calls each version executes? What influence can they have on the performance of both versions of the code? Why?

Output format

Analyse in detail the code of the files es2.c, es3.c and es4.c and make sure you understand what they do before continuing. Next, compile them using the make command.

1. Execute twice the program es2, first with 0 as a parameter and then with 1 (use the value you want for the second parameter). Redirect the standard output of the process to associate it to two different files. Observe the contents of the generated files.

> ⚠️ **QUESTION 56.** Explain the differences observed in the standard output of the program when the first parameter is 0 or 1. What do you think is the use of this parameter?

2. Execute twice the program es3 redirecting its standard input of each execution with each one of the two files generated in the previous section.

> ⚠️ **QUESTION 57.** Explain the reason of the observed results depending on the format of the input file.

3. Execute twice the program es4 in the same way you have executed the program es3 in the previous section.

> ⚠️ **QUESTION 58.** Explain the observed differences between the output of programs es3 and es4. Why the output is intelligible for one of the files and not for the other?

Life cycle

Analyse the contents of the files es5.c and es6.c and make sure you understand how they work.

1. Compile the two programs and execute each one of them in a different shell. Next, execute the following command:

   # ./showCpuTime.sh ./es5 ./es1

   showCpuTime.sh is a script that shows the CPU consumption time of each one of the programs passed as parameters every so often (every 3 seconds).

2. When the script is over, kill the two processes es5 and es1.

> ⚠️ **QUESTION 59.** Write the values that the showCpuTime.sh script has showed for each one of the processes and comment the differences between them regarding CPU consumption. Why are they caused? Identify the lines of code that mark the difference between them.

Modifying the kernel's signal management: disabling the SA_RESTART flag

In previous sessions we always enabled the SA_RESTART flag in the sigaction system call. This way, when a process is blocked in an input/output and it receives a signal, the process unblocks, manages the signal and, if necessary, the input/output operation continues transparent to the user. However, this is not the standard UNIX behaviour, where the input/output doesn't continue automatically.

Create a copy of the file es6.c naming it es6_v2.c. Modify the program es6_v2.c to reprogram the SIGINT signal management and to show a message through the standard output saying that a signal has been received. Use the sigaction system call without the SA_RESTART flag so that the management is equal to the UNIX management. In this case, the read returns an error when receiving this signal. Modify the main program in order to, after the read, a message is shown through standard output indicating the result of the operation: correct read, read with error (different from signal interruption) or signal interrupted read. Consult in the man the different values of **errno** for these cases.

Make the following sequence of executions to check the proper functioning of your code:

> a) Execute the program and press return to unblock the *read.*
> b) Next, execute the program but while it's waiting in the *read* send a SIGINT signal pressing ^C.

QUESTION 60. Write down in the file entrega.txt the results of both executions. Hand in the code programmed in the file es6_v2.

QUESTION 61. What would happen if we would add the SA_RESTART flag to the sigaction system call? Repeat the previous executions enabling the SA_RESTART flag and write down the result in the file entrega.txt.

Exercise about Device Drivers

The objective of this exercise is to check how Linux maintains the association between the logic device and the physical device. Meaning, how is it capable of translating the generic function of the access interface, in the specific code implemented by the device drivers.

For it we will use the device drivers that you have analysed in the previous work and that are located in the deviceDrivers directory.

1. In the case of the two device drivers, the major and the minor are fixed in the code. Depending on the system could be that this concrete numbers would be already in use and, therefore, the driver wouldn't be able to install. If this happens, substitute the major in the code for one that is not being used. You need to have in mind that both majors have to be different since both drivers will be installed at the same time. To see the list of drivers of the system and the used majors you can see the contents of the /proc/devices file (they are character-type devices).

QUESTION 62.   Was already in use the major specified in the code? If it was, what driver was using it?

2. Execute the following script:

    #./installDrivers.sh

This script compiles and loads in memory the two device drivers (myDriver1 and myDriver2). For it, it uses the make command, obtaining the compiled modules (myDriver1.ko and myDriver2.ko) corresponding to the first and second driver. Next, it uses the insmod command to install myDriver1 and miDriver2 (**note: to load/unload a device you need to be root, that's why the sudo command is used).**

3. Use the lsmod command to check that the modules are loaded correctly.

QUESTION 63. Write down the output line of lsmod corresponding to myDriver1 and myDriver2.

4. Use the mknod command to create a new device, called myDevice, of type *character* (option c) in your current working directory with the major and minor defined by myDriver1. **To create a device you need to be root (see command sudo).**

QUESTION 64. Write down the command line you have used to create the device.

5. Execute the following command:

    #./es1 < myDevice

QUESTION 65. Write down in the file "entrega.txt" the output of the execution and explain the obtained result.

6. Now remove myDevice and create again a device of type character with the same name, but associating the major and minor defined by myDriver2.c. Execute again the command from section 5.

QUESTION 66. Write down the output of the execution. Explain the reason of the differences observed comparing the output of this exercise with the output of section 5.

7. Remove myDevice and execute the following script:

    #./uninstallDrivers.sh

This script uninstalls myDriver1 and myDriver2.

Session 8:Input/Output management

Previous work

Objectives
- Understand the differences between anonymous pipes, named pipes and sockets.
- Understand the workings of the Unix device access interface.

Abilities
- Be able to communicate processes using anonymous pipes.
- Be able to communicate processes using named pipes.
- Be able to communicate processes using local sockets. In this case the objective is to know how to send/receive data with a previously created communication.

Previous knowledge
- Process management system calls

Previous work guide
- Before the session, consult the man (`man command_name`) of the following commands. Concretely, for each command you have to read and perfectly understand: the SYNOPSIS, the DESCRIPTION and the options commented on the "Options" column of the table.

To read in the man	Basic description	Options
mknod	Command that creates a special file	P
mknod (system call)	System call that creates a special file	
pipe	System call to create an anonymous pipe	
open	Opens a file or device	O_NONBLOCK, ENXIO
close	Closes a file descriptor	
dup/dup2	Duplicates a file descriptor	
socket	Creates a socket	AF_UNIX, SOCK_STREAM
bind	Assigns a name or direction to a socket	
listen	Waits connections to a socket	
accept	Accepts a connection in a socket	
connect	Initiates a connection in socket	

- Create a named pipe using the command mknod. Next, launch a process that executes the program 'cat' redirecting its standard output to the pipe you just created. In a different shell launch another process that also executes the program 'cat', but now redirecting its standard input to the pipe you just created. Introduce data with the

keyboard, in the first Shell, and press ^D to indicate the end. Write down in the file "previo.txt" the commands you have executed.

- Is it possible to communicate the two 'cat' commands from two different terminals through an anonymous pipe (for example, using a shell pipeline seen in the previous session)? And from the same terminal? Reason your answer in "previo.txt".

- Write in the "previo.txt" file the fragment of code we should add to any program to redirect its standard input to the writing end of an anonymous pipe using the close and dup system calls. Imagine that the file descriptor associated to the writing end of the pipe is the 4.

- On the webpage of the course (http://docencia.ac.upc.edu/FIB/grau/SO) you have the S8.tar.gz file that contains all the source files that you will use in this session. Create a directory in your machine, copy S8.tar.gz in it and unzip it (tar zxfv S8.tar.gz).

- The "socketMng.c" file contains some basic socket management functions (creation, connection request, connection acceptance and virtual device closing).
  - Analyse in detail the code of the createSocket and serverConnection functions, and search in the man the meaning of the system calls socket, bind, listen and accept.
  - Explain in the "previo.txt" file step by step what those functions do.

- **TO DELIVER: previo8.tar.gz**
  - **#tar zcfv previo8.tar.gz previo.txt**

Bibliography

- Slides of Topic 4 (Input/Output) of SO-grau.
- Chapter 13 (I/O Systems) de A. Silberschatz, P. Galvin y G. Gagne. Operating System Concepts, 8th ed, John Wiley &Sons, Inc. 2009.

Exercises to do in the laboratory

➢ As you do the exercises, modify the Makefile to compile and mount the new programs that are asked.

➢ Answer in a text file named entrega.txt all the questions that appear in this document, indicating for each question its number and answer. This document must be delivered through Racó. The questions are highlighted in bold and marked with the following symbol:

➢ **TO DELIVER: sesion8.tar.gz**
  - **#tar zcfv sesion8.tar.gz entrega.txt sin_nombre.c lector.c escritor.c escritor_v2.c lector_socket.c escritor_socket.c Makefile**

Anonymous pipes

Write a program in the file "sin_nombre.c" that creates an anonymous pipe and a child process whose standard input channel should be associated to the reading end of the pipe. To do the redirection use the system calls close and dup. The child process should mutate its image to proceed with the execution of the 'cat' command seen in the previous work. Meanwhile, the

parent process will send through the pipe the text message "Inicio" to its child, will close the writing end of the pipe and will wait until its child ends. When that happens, the father will show the message "Fin" through its standard output and end its execution.

1. Execute the previous program making the Shell redirect the standard output of the parent to a file.

QUESTION 67.   What's the content of the file after the execution? Does it contain the output of both the parent and child, or only of the parent? How is this content explained?

2. Change the code of the parent to not close the writing end of the pipe after sending the message.

QUESTION 68.   Does the parent program finish? And the child? Why?

Named pipes

Write two programs that communicate through a named pipe. One of them (lector.c) will read from the pipe until the reading indicates it that no more data is left to read and will show in its standard output everything that it reads. The other process (escritor.c) will read from its standard input until the reading indicates that there is no more data and will write in the pipe everything that it reads. When there's no more data to read both program should finish.

QUESTION 69.   If we wanted the reader to be able to send a message to the writer, could we use the same named pipe or should we create another one? Reason your answer.

Write another version of the previous pipe writer, called escritor_v2.c. This program when trying to open the pipe, if there's no reader in the pipe, will show a message in the standard output that indicates that it is waiting for a reader and will block itself in the open of the pipe until a reader opens it to read from the pipe. Consult the ENXIO error in the man of the open (man 2 open) to see how to implement this behaviour.

Sockets

Modify the code of "lector.c" and "escritor.c" done in section 2 in order to, instead of using named pipes, use local sockets (type AF_UNIX) to do the communication. The writer has to take the role of client, while the reader has to take the role of the server. Call this new files "lector_socket.c" and "escritor_socket.c".

To make the code you have to use the functions provided by the file socketMng.c and use the code that you consider appropriate of the files exServerSocket.c and exClientSocket.c. The file exServerSocket.c contains a program that acts as a simple server creating a socket (with the name written as a parameter) and waiting for connection requests from any client. The file exClientSocket.c contains a program that acts as a simple client that requests a connection to the socket it receives as a parameter. It is not necessary to modify the code that establishes the connection, only introduce the data communication.

Session 9: File system

Previous work

Objectives
- During this practise we will realise a series of exercises about the File System topic, with the aim of practising the knowledge acquired in the theory lectures.

Abilities
- Be able to use commands and basic system calls to work with the FS.
- Be able to modify the reading/writing pointer with the lseek call.

Previous knowledge
- Input/Output and file system system calls.
- Process management system calls.

Previous work guide
- Review the theory lecture contents, especially the ones related to the I-node based file system.
- Before the session, consult the man (`man command_name`) of the following commands. Concretely, for each command you have to read and perfectly understand: the SYNOPSIS, the DESCRIPTION and the options commented on the "Options" column of the table.

To read in the man	Basic description	Options
open/creat	Open/create a file or device	O_CREAT,O_TRUNC, "Permissions"
df	Returns information about the file system	-T, –h, –l,-i
ln	Creates links to files	-s
namei	Processes a route of a file until it finds the final dot	
readlink	Reads the contents of a symbolic link	
stat	Shows control information about a file	-Z, -f
lseek	Modifies the reading/writing position of a file	SEEK_SET, SEEK_CUR, SEEK_END

1. **Answer the following questions** in the file "previo.txt":

- How can you know the file systems mounted in your system and their type? Also indicate where they are mounted.

- How can you know the number of free inodes in a file system? Which command have you used and with which options?
- How can you know the free space of a file system? Which command have you used and with which options?

2. **Execute the following commands and answer in the file previo.txt** the following questions:

> # echo "this is a test" > pr.txt
> # ln –s pr.txt sl_pr
> # ln pr.txt hl_pr

- Which is the type of each created link, sl_pr and hl_pr? Execute the command stat on pr.txt, sl_pr and hl_pr. Search in the output of stat the information about the inode, the file type and the number of links and write it down in the file previo.txt. How many links has each file? What's the meaning of this value? Which inode has each file? Does any of the links, sl_pr or hl_pr, have the same inode of pr.txt? If positive, what's the meaning of that?
- Execute the commands **cat**, **namei** and **readlink** on sl_pr and sobre hl_pr:
  - o Write down the results in the file previo.txt.
  - o Do you observe any difference in the results of some of the commands when executed on sl_pr and when executed on hl_pr? If there's any difference, explain the reason.
- Eliminate now the pr.txt file and execute again the commands **stat**, **cat**, **namei** and **readlink** in both sl_pr and hl_pr.
  - o Write down the results in the file previo.txt
  - o Do you observe any difference in the results of some of the commands when executed on sl_pr and when executed on hl_pr? If there's any difference, explain the reason.
  - o Do you observe any difference in the execution of these commands before and after deleting the file pr.txt? If there's any difference, explain the reason.

3. **Write a program "crea_fichero.c"** that using the creat system call creates a file called "salida.txt" with the content "ABCD". If the file already existed it should be overwritten. The created file should have reading and writing permissions for the owner and the rest of user shouldn't be able to perform any operation.
4. **Write a program "insertarx.c"** that inserts in the previous file (salida.txt) the letter X between the two last characters. The result has to be "ABCXD". The program uses the file salida.txt as an example but it has to be generic, independently of the size of the input file it will always write an X "between the two last characters".

- **TO DELIVER: previo9.tar.gz**
  - o **#tar zcfv previo9.tar.gz previo.txt**

Bibliography

- Slides Topic 4 of SO-grau
- Chapters 10 y 11 (File-System Interface & File-System Implementation) de A. Silberschatz, P. Galvin y G. Gagne. Operating System Concepts, 8th ed, John Wiley &Sons, Inc. 2009.

Exercises to do in the laboratory

➢ Answer in a text file named entrega.txt all the questions that appear in this document, indicating for each question its number and answer. This document must be delivered through Racó. The questions are highlighted in bold and marked with the following symbol:

➢ **TO DELIVER: sesion9.tar.gz**

#tar zcfv sesion9.tar.gz entrega.txt append.c invirtiendo_fichero.c insertarx2.c

Hard links and Soft links

Create a directory called "A" inside your home directory ($HOME). Next, place yourself in the "A" directory using the command cd and execute the following commands, that create soft and hard links to different files:

```
$ echo "I'm in directory A" > D
$ ln -s $HOME/A  $HOME/A/E
$ ln -s D $HOME/A/F
$ ln  $HOME/A  $HOME/A/H
$ ln  D $HOME/A/G
```

QUESTION 70. Answer the following questions in the file "entrega.txt"

- Which one/s of the previous commands have given an error when trying to execute it/them? Explain why.
- Explain the result of executing the command "stat" using as a parameter each one of the symbolic names that you have created previously.
- What would be the contents of the files D, E, F and G? Comment the differences that you observe when using the commands "more" or "cat" to see the content of the file and the result of using the command "readlink".
- Write in a paper the accesses that are realized when the files "$HOME/A/F", "$HOME/A/E" and "$HOME/A/G" are accessed. Now compare them with the result obtained when you execute the command "namei" with each one of the previous files. If you execute "readlink $HOME/A/F", are the same accesses done? What is the influence of the fact that one of them is with the absolute path and the other with the relative path?
- Create a soft link of a file to itself (a file that didn't exist previously). Comment the result of looking its content using the command "cat". Observe how the system

controls the existence of cycles in the file system. Execute the command "namei" and comment its result.

File size control

- Create a file named "file" with content "12345".
- Now implement a program, called append.c, that adds to the end of the "file" file the content of the same file. We use the "file" file as an example, but the program has to be generic: for any input file, after executing the append.c program the file should have its content duplicated. **Hint:** if when you try this program it lasts more than some seconds to finish, kill the process and check the size of the file. If this size is more than the double of the original size, then revise in the code the ending condition you have stated for the file reading loop.

> **QUESTION 71.** Deliver the file append.c.

Operations with lseek

Create a program that we will call "invirtiendo_fichero" that will do a copy of the file that receives as a parameter but with the inverse content. The name of the resulting file will be the same as the original file but with the extension ".inv".

> **QUESTION 72.** Deliver the file invirtiendo_fichero.c.

Create a data file whose contents are "123456". Now implement some code (insertarx2.c) that inserts the character "X" between the characters "3" and "4" in such a way that the resulting content of the file is "123X456". Make this test as an example, but the program has to be generic: Given a concrete position of the file, the character "X" has to be inserted there.