People's Democratic Republic of Algeria
Ministry of higher education and scientific research
University of Mohamed Boudiaf - M'sila

Faculty of Mathematics and Computer Science
Department of Computer Science

2nd Year License (2L)

Operating Systems 1 (OS 1)

Semester : 04

2025/2026

Directed by

Dr. DABBA ALI

Contact us

alidabba@gmail.com

ali.dabba@univ-msila.dz

- In case of problems or difficulties, please contact me or your TD / TP teacher.
- We are at your disposal to help you

CHAPTER 3.1 (Lab)

Unix / Linux Processes

Plan

I. Introduction

II. Process Review

III. Types of Processes

IV. Signals

V. Pipes (Tubes)

I. Introduction

- A running instance of a program is called a **process**.
- It should be understood that a program is part of the file system that resides on a non-volatile media (such as disk), and a process is an entity that is being executed (with at least some portion, i.e. segment/page) in RAM. As one author cleverly stated "...the file system has 'places' and that processes have 'life'."
- On the other hand, Unix being a **multiuser** and a **multi-tasking** system, there could be several programs belonging to different users or the same user running at the same time. **All these programs share the same CPU**.

I. Introduction

- The kernel generates or spawns processes for every program under execution and allocates definite and equal CPU time slots (Unix is a time-sharing system) to these various programs.
- Processes are the building blocks of a Unix system. **Why**?
- Because any code that is executed happens inside a process. When you execute a program on your Unix system, the system creates a special *environment* for that program.
- This environment contains everything needed for the system to run the program as if no other program were running on the system.

I. Introduction

- This chapter explains the fundamental concept of processes in Linux, explores different types of processes, and discusses two inter-process communication mechanisms: **signals** and **pipes**.

II. Process Review

- Processes have the concept of a lifetime associated with them. Processes come to life, or are said to be **born** as a program is loaded and begins execution.
- Processes remain alive while the program continues execution (processes, however can have different states of life).
- Processes cease to exist and are said to **die** as the running program terminates (either normally or abnormally).
- A process is a program in progress. Thus, it is a dynamic object with a specific life cycle.

II. Process Review

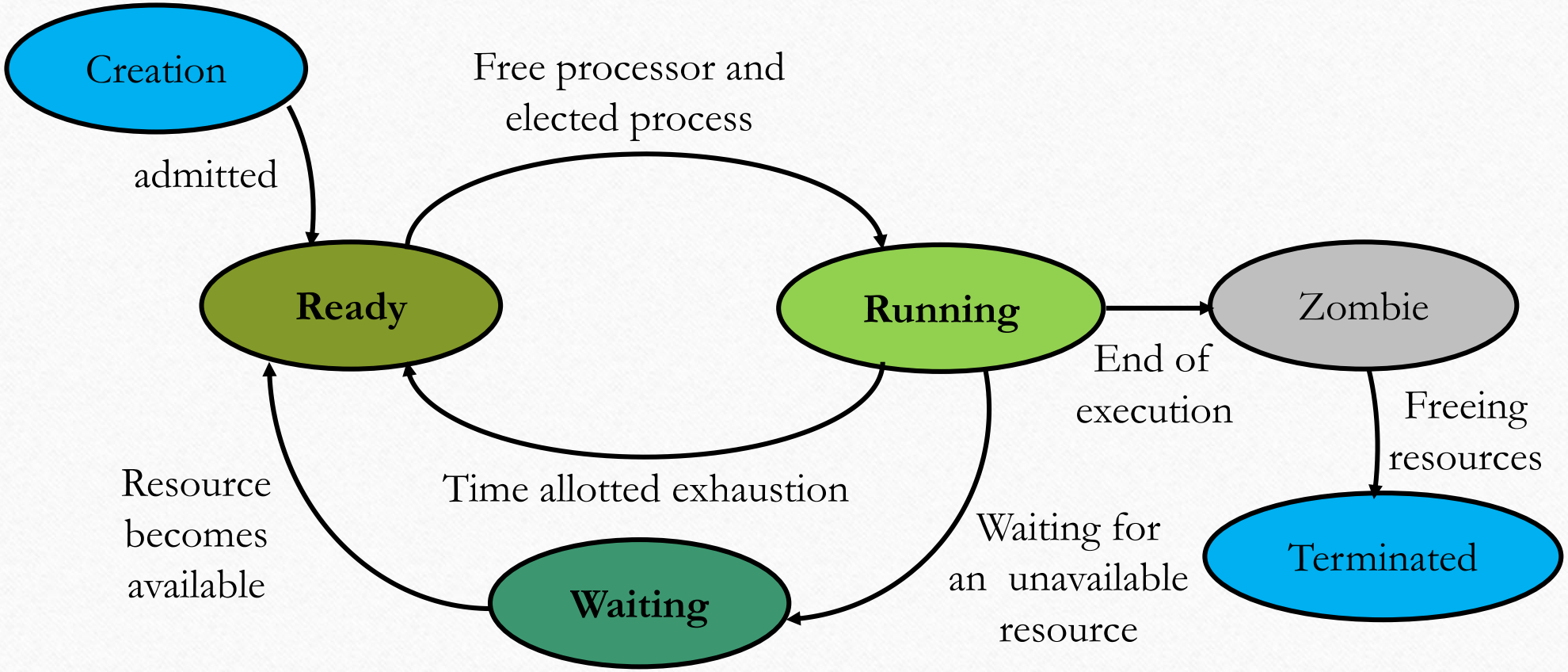


Figure 3.1 : Process Life Cycle

II. Process Review

1) Looking at Processes

Processes are running at all times. Each running program uses one or more processes.

- A process is an instance of running a program.
- OS tracks processes through a five-digit ID number known as the process-ID (**PID**).
- Each process has a unique **PID**. **PID** is an integer of the type **pid_t**, declared as a synonym of the type **int** or **unsigned long int**.
- A process has a priority and an execution mode.
- We distinguish between two modes: the low-priority user (or master) mode and the higher-priority kernel (or slave) mode.

II. Process Review

1) Looking at Processes

- Usually, a process belongs to the person who wrote the program that is running, but this is not always the case.
- When you run the "**ls**" command, you run the "**/bin/ls**" program, which belongs to **root**.
- Take a look at the result of the "**ls -l**" command.

```
$ ls -l /bin/ls
-rwxr-xr-x 1 root    root      29980 Apr 24 1998    /bin/ls
```

II. Process Review

2) Process IDs

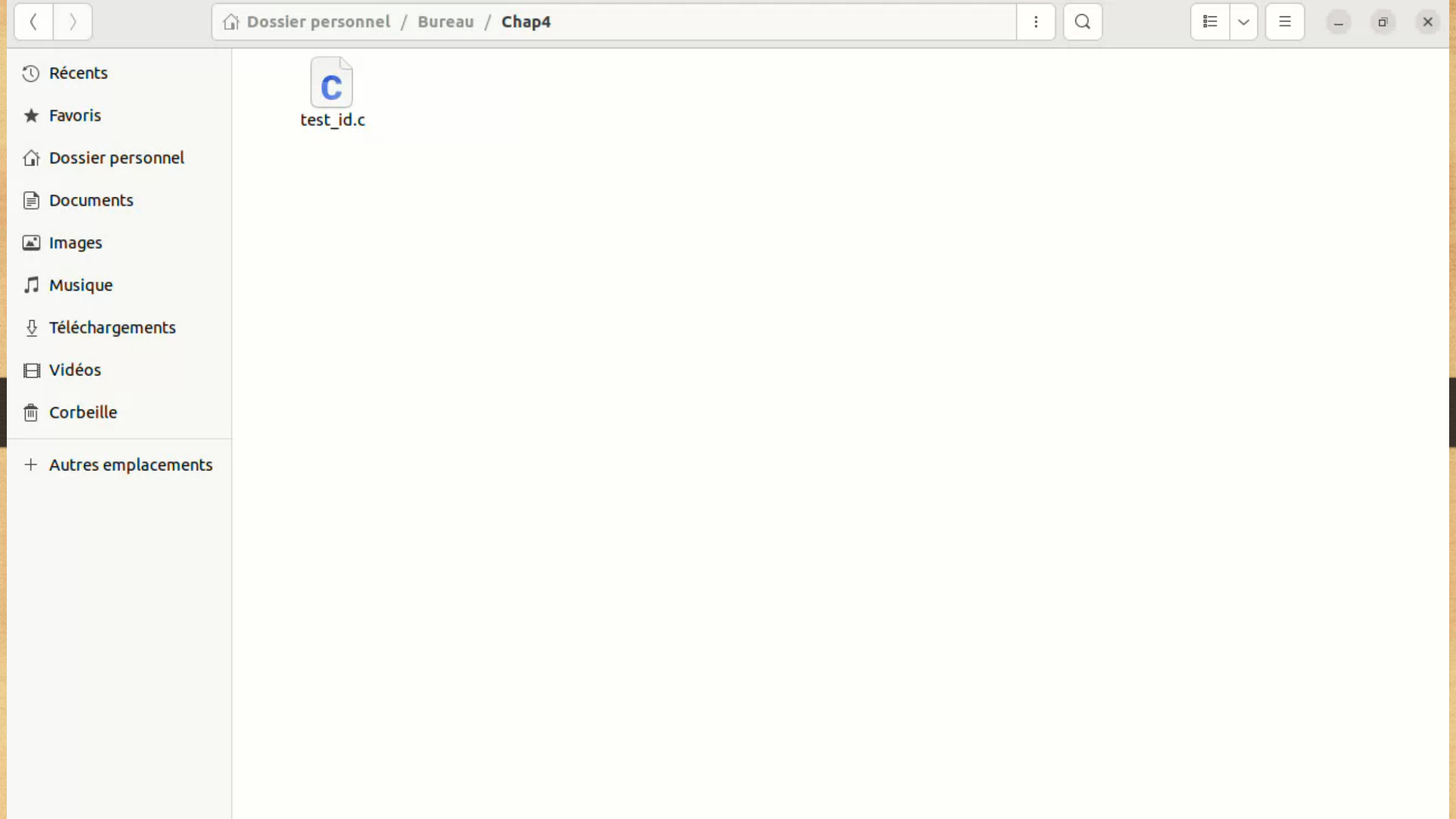
- Each process in a Linux system is identified by its unique process ID (**PID**). Process IDs are **16-bit** numbers that are assigned sequentially by Linux as new processes are created.
- Every process also **has a parent process** (except the special "**init**" process). Thus, we can think of the processes on a Linux system as **arranged in a tree**, with the "**init**" process at its root.
- The **parent process ID**, or **ppid**, is simply the process ID of the process's parent.

II. Process Review

2) Process IDs

- Let's take a look at the various primitives that can be used to identify these different identifiers:

pid_t getpid()	/* returns the process identifier */
pid_t getpgrp()	/* returns the process group identifier */
pid_t getppid()	/* return process parent identifier */
pid_t setpgrp()	/* sets the process group identifier to the value of its pid: creates a new process group. Value returned: new process group identifier.*/


Dossier personnel / Bureau / Chap4
Récents
Favoris
Dossier personnel
Documents
Images
Musique
Téléchargements
Vidéos
Corbeille
Autres emplacements
test_id.c

II. Process Review

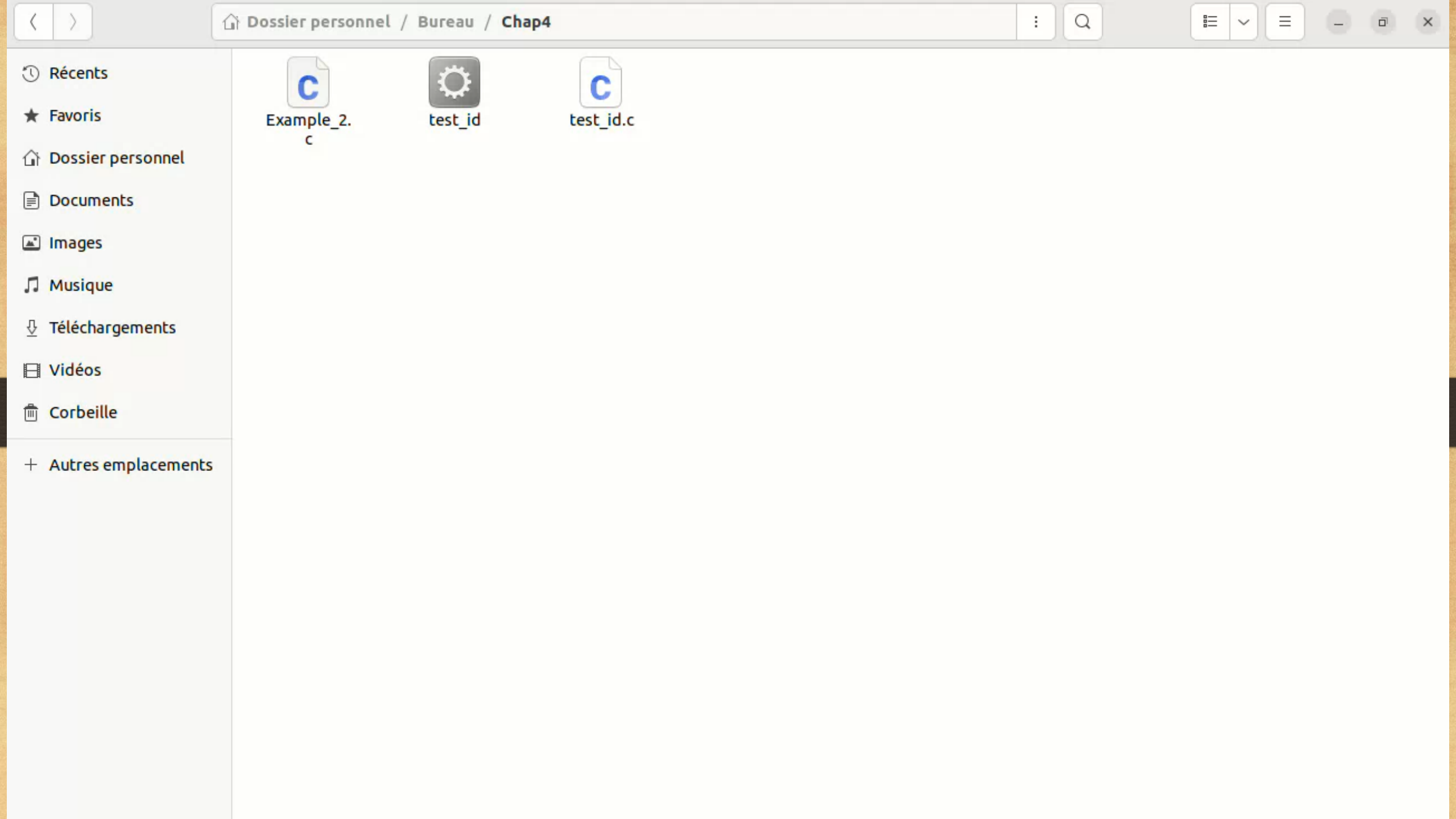
3) Creating Processes (Using "system“)

- The system function in the standard C library provides an easy way to execute a command from within a program, much as if the command had been typed into a **shell**.
- In fact, system creates a subprocess running the standard Bourne shell (**/bin/sh**) and hands the command to that shell for execution.

II. Process Review

3) Creating Processes (Using "fork" and "exec“)

- Linux provides one function, **fork**, that makes **a child process that is an exact copy of its parent process.** Linux provides another set of functions, the **exec** family, that causes a particular process to cease being an instance of one program and to instead become an instance of another program.
  - To spawn a new process, we first use **fork** to make a copy of the current process.
  - Then, we use **exec** to transform one of these processes into an instance of the program we want to spawn.


Dossier personnel / Bureau / Chap4
Récents
Favoris
Dossier personnel
Documents
Images
Musique
Téléchargements
Vidéos
Corbeille
Autres emplacements
Example_2.c
test_id
test_id.c

II. Process Review

3) Creating Processes (Calling "fork“)

- When a program calls **fork**, a duplicate process, called the **child process**, is created.
- The parent process continues executing the program from the point that **fork** was called.
- The child process, too, executes the same program from the same place.
- The parent and child each have their own, private memory images. If the parent subsequently changes any of its variables, the changes are not visible to the child, and vice versa.

II. Process Review

3) Creating Processes (Calling "fork")

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

OR

```
pid_t fork();
```

II. Process Review

3) Creating Processes (Calling "fork“)

So how do the two processes differ?

- First, the child process is a new process and therefore has a new process ID, distinct from its parent’s process ID.
- Parent and Child differ only in the result returned by **fork**. For the parent, this function returns the number of the child (or **-1** if creation is impossible) and for the child, it returns **0**.

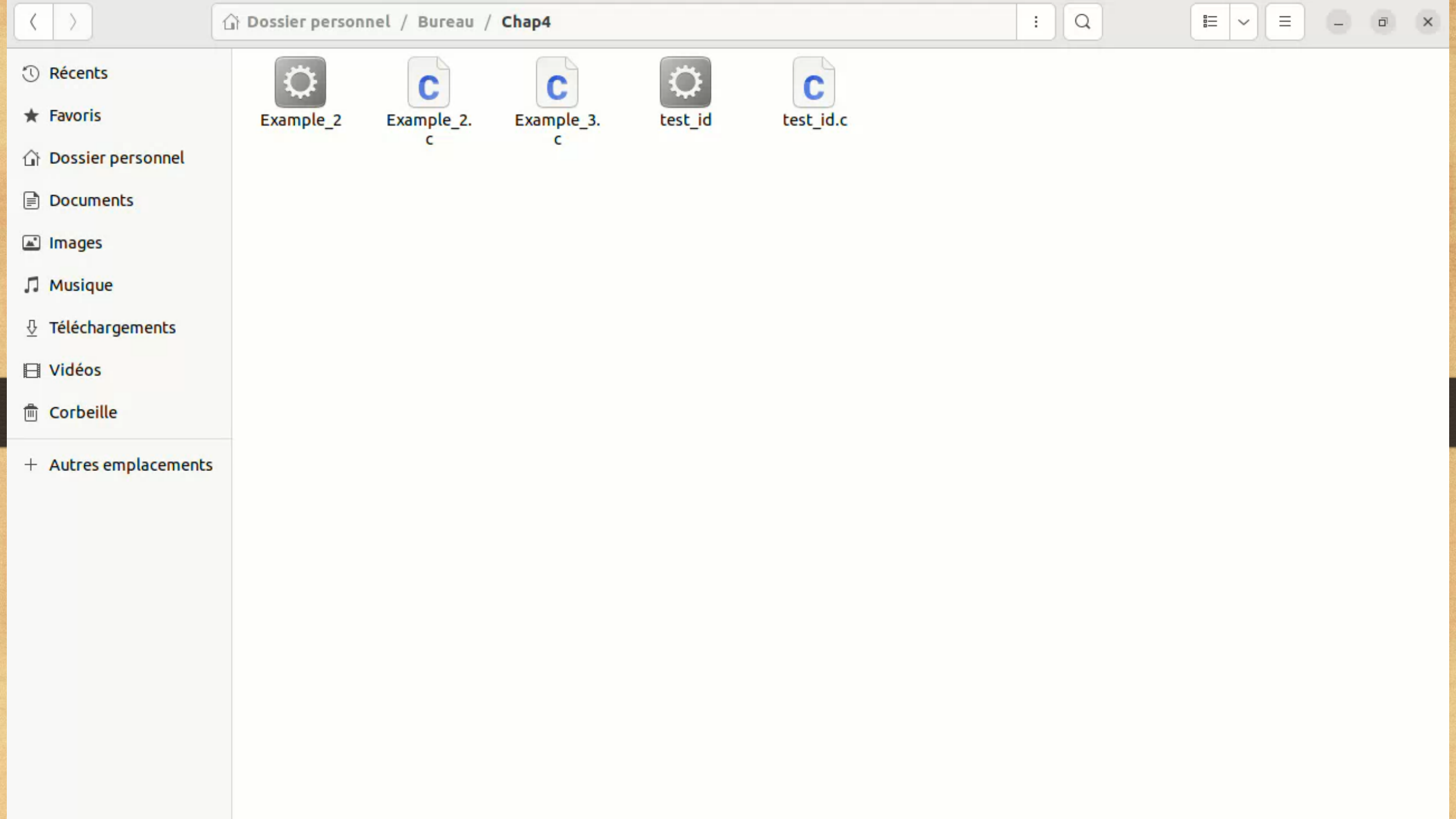
II. Process Review

Parent	pid=3345

```
int main () {
    pid_t pid =    fork ();
    if (pid == 0)
        ClidPocess();
    else
        ParentPocess();
}
```

Child	pid=0

```
int main () {
    pid_t pid = fork ();
    if (pid == 0)
        ClidPocess();
    else
        ParentPocess();
}
```


Dossier personnel / Bureau / Chap4
Récents
Favoris
Dossier personnel
Documents
Images
Musique
Téléchargements
Vidéos
Corbeille
Autres emplacements
Example_2
Example_2.c
Example_3.c
test_id
test_id.c

II. Process Review

3) Creating Processes (Using The "exec" Family)

- The **exec** functions replace the program running in a process with another program.
- When a program calls an **exec** function, that process immediately ceases executing that program and begins executing a new program from the beginning, assuming that the **exec** call doesn't encounter an error.

II. Process Review

3) Creating Processes (Using The "exec" Family)

Within the **exec** family, there are functions that vary slightly in their capabilities and how they are called.

- Functions that contain the letter ***p*** in their names (**execvp** and **execlp**) accept a program name and search for a program by that name in the current execution path; functions that don't contain the ***p*** must be given the full path of the program to be executed.
- Functions that contain the letter ***v*** in their names (**execv**, **execvp**, and **execve**) accept the argument list for the new program as a **NULL**-terminated array of pointers to strings.
- Functions that contain the letter ***l*** (**execl**, **execlp**, and **execle**) accept the argument list using the **C** language's varargs mechanism.

II. Process Review

3) Creating Processes (Using The "exec" Family)

Within the **exec** family, there are functions that vary slightly in their capabilities and how they are called.

- Functions that contain the letter ***e*** in their names (**execve** and **execle**) accept an additional argument, an array of environment variables. The argument should be a **NULL**-terminated array of pointers to character strings. Each character string should be of the form **"VARIABLE=value"**.
- Because **exec** replaces the calling program with another one, it never returns unless an error occurs.

II. Process Review

3) Creating Processes (Process Hierarchy)

- In the operating system, a process can create a child process.
- Hence, all processes come from a **single root** in which a creating process is the parent process and the created process is the child of that process.
- The child process can itself create more processes, thereby forming a process hierarchy.

II. Process Review

3) Creating Processes (Process Hierarchy)

- For example, in UNIX: Once the initial boot process is done and the operating system is loaded, a special process called **init**, present in the boot image starts running. It reads a file telling how many terminals are there. Then it forks off a new process per terminal. These processes wait for someone to log in. If a login is successful, the login process executes a shell to accept commands. These commands may start up more processes and so forth. Thus, all the processes in the whole system belong to a single tree, with **init** at the root.

II. Process Review

3) Creating Processes (Process Hierarchy)

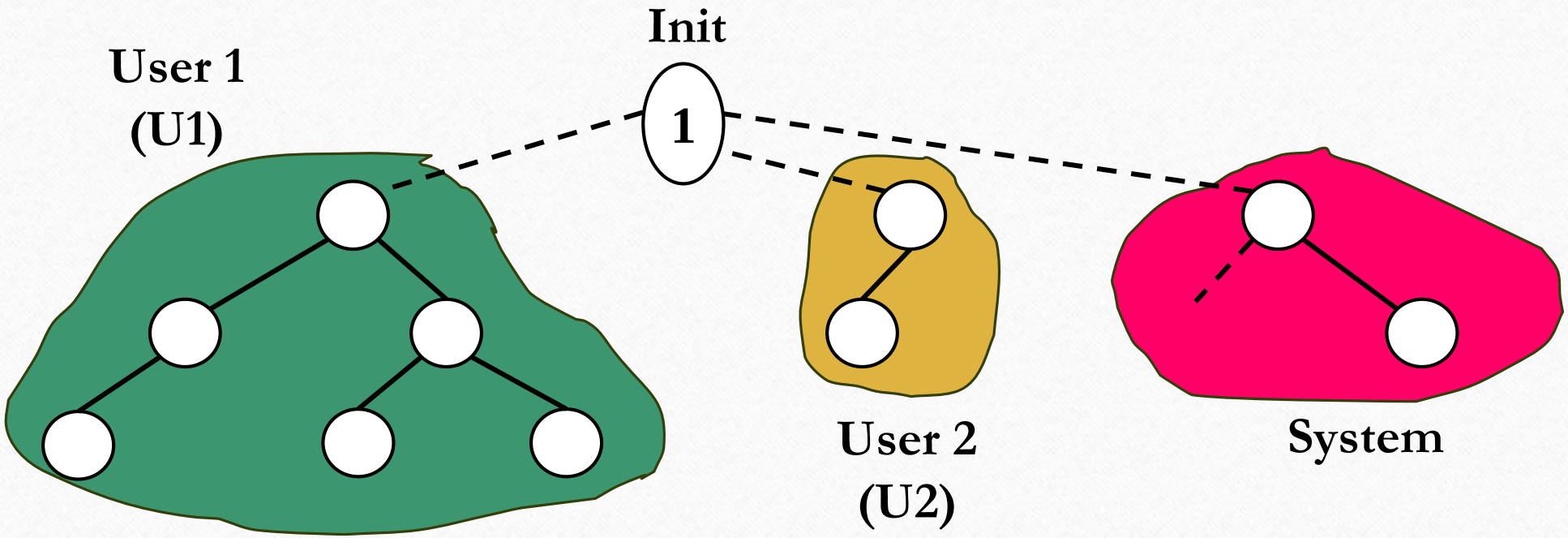


Figure 4.2 : Process Hierarchy

II. Process Review

3) Creating Processes (Process Hierarchy)

- In UNIX-like environment, we can view the current process tree by:

```
$ ps -e
```

Then, you can find that we have many processes and the root for all of them is the process with **pid=1** and name **init** (for UNIX-based systems).

- To find the process related only to the current terminal use:

```
$ ps -a
```


ali@Ali-PC: ~
ali@Ali-PC:~$

II. Process Review

3) Creating Processes (Process Termination)

- A process terminates when it finishes executing its last statement.
- Its resources are returned to the system, it is purged from any system lists or tables, and its process control block (PCB) is erased i.e., the PCB's memory space is returned to a free memory pool.
- Under normal conditions, a process can be terminated in one of two ways:
  - Either the program's **`main`** function terminates, or
  - the program calls the **`exit`** function.
- As it can be explicitly eliminated by another process, by sending a **`kill`** signal (see below).

II. Process Review

3) Creating Processes (Process "exit" Code)

- Each process has an **`exit` code**: a number that the process sends back to its parent. The **`exit` code** is the argument passed to the **exit** function or the value returned from **main**.
- By convention, the **`exit` code** is used to indicate whether the program has executed correctly:
  - An **`exit` code** of **zero** indicates **correct execution**,
  - A **non-zero code** indicates that an **error has occurred**.

II. Process Review

3) Creating Processes (Synchronization of Parent/Child Processes)

- When a child process terminates, it becomes a **zombie** process.
- A **zombie process** is a process that has terminated but has not been cleaned up yet.(its entry in the process table is not removed).
- It will remain in this state until its parent process has recovered its exit code. It is the responsibility of the parent process to clean up its zombie children.

II. Process Review

3) Creating Processes (Synchronization of Parent/Child Processes)

- The **wait** and **waitpid** system calls allow the parent process to retrieve this code. So, it's not necessary to track whether your child process is still executing before waiting for it.
- Suppose, for instance, that a program forks a child process, performs some other computations, and then calls **wait**.

II. Process Review

3) Creating Processes (Synchronization of Parent/Child Processes)

- If the child process has not terminated at that point, the parent process will block in the **wait** call until the child process finishes.
- If the child process finishes before the parent process calls **wait**, the child process becomes a **zombie**.
- When the parent process calls **wait**, the zombie child's termination status is extracted, the child process is deleted, and the **wait** call returns immediately.

II. Process Review

3) Creating Processes (The "wait()" System Call)

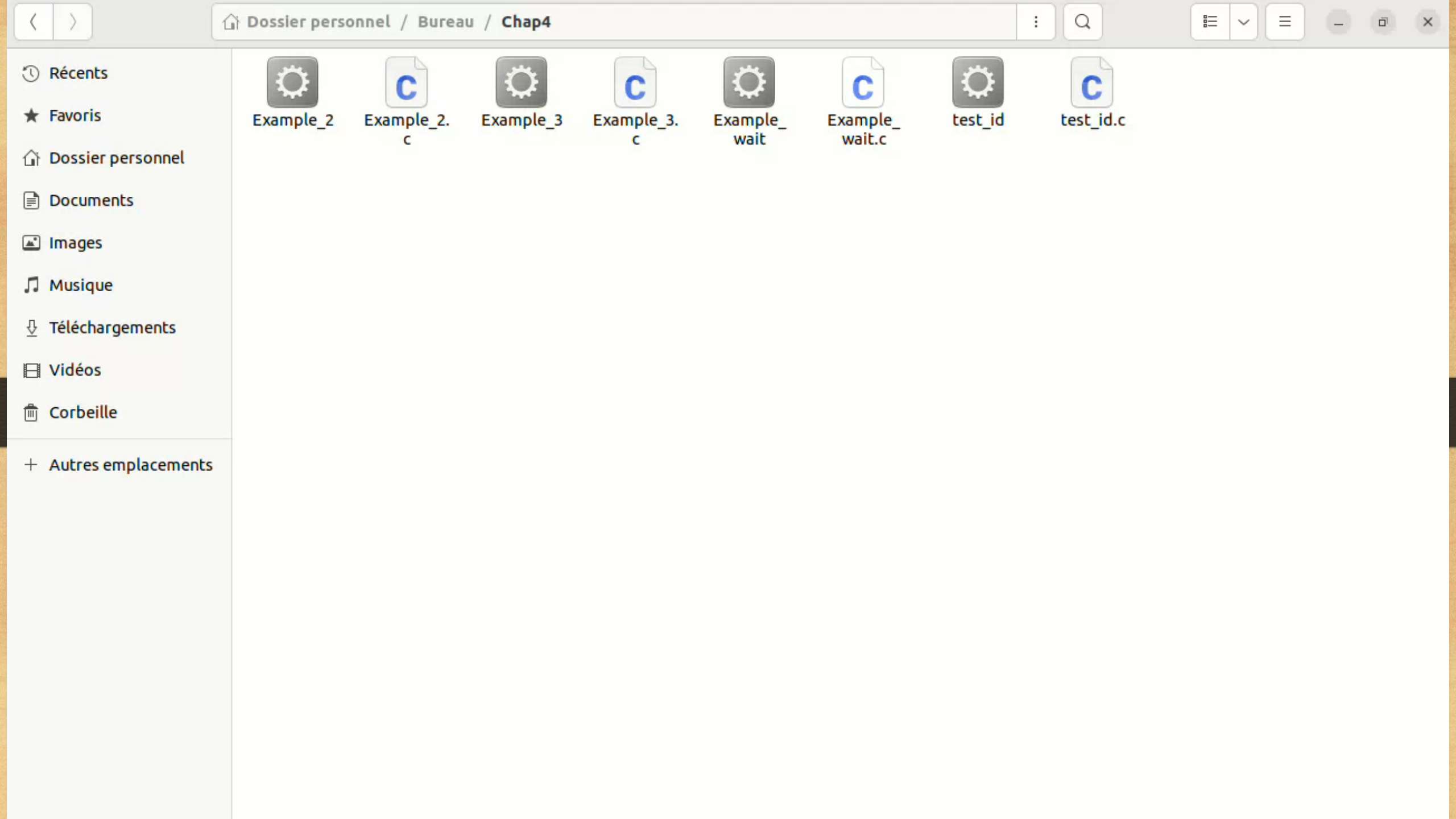
- The simplest such function is called simply **wait**. It blocks the calling process until one of its child processes exits (or an error occurs).

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait (int* status);
```

II. Process Review

3) Creating Processes (The "wait()" System Call)

- The **wait** system call suspends execution of the calling process until one of its children terminates.
  - If a child has already terminated, **wait** returns the child's PID immediately without blocking. It returns the PID of the child process and its termination **status** in status (if status is not **NULL**).
  - On the other hand, if the calling process has no children, **wait** returns **-1**.
- We know if more than one child processes are terminated, then **wait()** reaps any arbitrarily child process but if we want to reap any specific child process, we use **waitpid()** function.


Dossier personnel / Bureau / Chap4
Récents
Favoris
Dossier personnel
Documents
Images
Musique
Téléchargements
Vidéos
Corbeille
Autres emplacements
Example_2
Example_2.c
Example_3
Example_3.c
Example_wait
Example_wait.c
test_id
test_id.c

II. Process Review

3) Creating Processes (The "waitpid()" System Call)

- The **waitpid()** system call allows a parent process to wait for a particular child of identity **pid**, either blocking (**options=0**) or non-blocking (**options=WNOHANG**).
  - If the call **succeeds**, it **returns PID of the child process** and its termination status in status (if status is not **NULL**).
  - In case of **error**, **-1 is returned**.
  - If the **WNOHANG** option is used and no child has changed state, the **return value is 0**.
- The **waitpid(-1, &status, 0)** call is equivalent to the **wait(&status)** call.

III. Types of Processes

Processes with in Unix are classified into three general categories as

- **Interactive (Foreground) Processes**
- **Non-interactive (Background) Processes**
- **Daemons.**

III. Types of Processes

1) Interactive Processes (Foreground Processes)

- All the user processes, which are created by users with the shell, act upon the directions of the users and are normally attached to the terminal are called **interactive processes.** These types of processes are also called **foreground processes.**
- When a command is given to the terminal, the shell parses, rebuilds and then hands it over to the kernel for execution.

III. Types of Processes

1) Interactive Processes (Foreground Processes)

- The shell then keeps on waiting for the kernel to complete the execution.
- During this shell-waiting period the user cannot issue any other command because the terminal is held up with the command under execution.
- As already mentioned, commands that hold up the terminal during their execution are called foreground processes.

III. Types of Processes

1) Interactive Processes (Foreground Processes)

- The **advantage** of foreground processing is that, no further commands can be given from the terminal as long as the older one is running.
- The **disadvantage** is when a currently running process is big and takes a lot of time for processing

III. Types of Processes

2) Non-interactive Processes (Background Processes)

- Certain processes can be made **to run independent of terminals.** Such processes that run without any attachment to a terminal are called **non-interactive processes.** These types of processes are also called **background processes.**
- It is possible to make processes to run without using the terminal. Such processes take their input from some file and process it without holding up the terminal (non-interactively), and write their output on to another file are called background processes.
- Typical jobs that could be run in background are sorting of a large database file or locating a file in a big file system by using the find command and so on.

III. Types of Processes

2) Non-interactive Processes (Background Processes)

- **Running a Command in the Background:** A command is made to run in the background by terminating the command line with an ampersand **(&)** character as shown in the following example.

```
$ sort –o student_file &
655
```

- The shell immediately returns the process identification (PID) number as well as the shell prompt **$.** In the above example, **655** is the **PID** of the just-submitted background job. As the shell prompt (**$**) re-appears immediately.

III. Types of Processes

2) Non-interactive Processes (Background Processes)

Some of these **problems** could be due to any one of the following.

- The success or failure of the background processes are not reported. The user has to find it out. For this purpose, the identification number is used.
- The output has to be redirected to a file as otherwise the display on the monitor gets mixed up.
- Too many processes running in the background degrades the overall efficiency of the system.
- There is a danger of the user logging out when some processes are still running in the background.

III. Types of Processes

3) Daemons

- All processes that keep running always without holding up any terminals and keep waiting for certain instructions either from the system or the user and then immediately get into action are called daemons. **swapper**, **init**, **cron**, **bdflush**, and **vhandle** are some examples of daemons.
- These daemons come into existence as soon as the system is booted and will be alive till the system is shut down. One cannot kill these processes prematurely.

III. Signals

- Signals are **mechanisms for communicating** with and manipulating processes in Linux.
- A signal **is a special message sent to a process**.
- Each signal has a specific meaning linked to the source that produced the event. Signals are the **simplest form of inter-process communication.**
- A signal can be **generated by hardware, by software, or by the user himself.**
- Signals are asynchronous; when a process receives a signal, it processes the signal immediately, without finishing the current function or even the current line of code.

III. Signals

- For example, a division by zero in the central processing unit triggers a hardware interrupt, signaled by **SIGFPE** (floating point exception).
- Simultaneous pressing of the **Ctrl + C** keys generates the SIGINT interrupt signal.
- When a child process terminates, Linux sends the parent process the **SIGCHLD** signal.

III. Signals

Definition 4.1 (signal)

- A signal is an asynchronous **software interruption** of a process.
- The signal is a special message sent to a process.
- It can be sent by a **process** or by the **kernel**.

III. Signals

1) Signals Identification

- Signals are identified by its signal (integer) number, but in programs, we usually refer to a signal by its name.
- In Linux, these are defined in **"/usr/include/bits/signum.h"** (We shouldn't include this header file directly in our programs; instead, use **<signal.h>**.).

III. Signals

1) Signals Identification

- The number of signals available in a system varies from **0** to **31**, from **0** to **63**, etc., depending on the system version. The values may change from one implementation to another; the symbolic names remain the same.
- The **kill -l** command provides a list of system signals.

Signal Name	Default Action	Comment	POSIX
SIGTERM	Terminate	Process termination	Yes
SIGINT	Terminate	Interrupt from keyboard, Ctrl-C	Yes
SIGKILL	Terminate	Forced-process termination	Yes
SIGQUIT	Core	Quit from keyboard, Ctrl-\	Yes
SIGTSTP	Stop	Stop process issued from tty	Yes
SIGSTOP	Stop	Stop process execution, Ctrl-Z	Yes
SIGCONT	Continue	Resume execution, if stopped	Yes
SIGCHLD	Ignore	Child process stopped or terminated	Yes
SIGABRT	Core	Abnormal termination	Yes
SIGALRM	Terminate	Real-timer clock	Yes
SIGUSR1	Terminate	Available to processes	Yes
SIGUSR2	Terminate	Available to processes	Yes
SIGSEGV	Core	Invalid memory reference	Yes
SIGFPE	Core	Floating-point exception	Yes
SIGILL	Core	Illegal instruction	Yes
SIGTRAP	Core	Breakpoint for debugging	No
SIGPIPE	Terminate	Write to pipe with no readers	Yes
SIGCHLD	Ignore	Child stopped or terminated	Yes

III. Signals

2) Signal Origins

The origins of Unix/Linux signals vary, and they can be :

- **Transmitted by the kernel** (division by zero, overflow, forbidden instruction, etc.),
- **Sent by the user from the keyboard** (**Ctrl - z** , **Ctrl - c**, ...etc.), sent by the kill command from the shell,
- By another process: **kill()**, and **sigqueue()** primitives (issued by the kill command from the shell or by the kill function from a program written in C)

III. Signals

3) Possible Signal Behaviors

Upon delivery of a signal, a process carries out one of the following default actions, depending on the signal:

- **The signal is ignored;** that is, it is discarded by the kernel and has no effect on the process. (The process never even knows that it occurred.)
- **The process is terminated (killed)**. This is sometimes referred to as abnormal process termination, as opposed to the normal process termination that occurs when a process terminates using **exit()**.

III. Signals

3) Possible Signal Behaviors

- **A core dump file is generated, and the process is terminated**. A core dump file contains an image of the virtual memory of the process, which can be loaded into a debugger in order to inspect the state of the process at the time that it terminated.
- **The process is stopped** : execution of the process is suspended.
- **Execution of the process is resumed** after previously being stopped.

III. Signals

3) Possible Signal Behaviors

- Instead of accepting the default for a particular signal, a program can change the action that occurs when the signal is delivered.
- This is known as setting the **disposition of the signal**.

III. Signals

3) Possible Signal Behaviors

A program can set one of the following dispositions for a signal:

- **The default action should occur.** This is useful to undo an earlier change of the disposition of the signal to something other than its default.
- **The signal is ignored.** This is useful for a signal whose default action would be to terminate the process.
- A **signal handler** is executed

A signal handler is a function, written by the programmer, that performs appropriate tasks in response to the delivery of a signal.

III. Signals

4) Signal Response

There are three ways to respond to a signal:

- **Execute the default action for the signal.** Five default treatments are possible:
  - **Terminate** : Default action is to terminate the process.
  - **Ignore** : Default action is to ignore the signal.
  - **Core** : Default action is to terminate the process and dump core.
  - **Stop** : Default action is to stop the process.
  - **Continue** : Default action is to continue the process if it is currently stopped.

III. Signals

4) Signal Response

There are three ways to respond to a signal:

- **Ignore the signal.** A number of signals can be ignored (except, for example, SIGKILL, SIGSTOP);
- Intercept and invoke a function in response to the event. The signal can be caught by the receiving process, which causes a diversion and the launch of a specific processing routine (handler). After processing, the process resumes

III. Signals

5) Changing Signal Dispositions

- UNIX systems provide two ways of changing the disposition of a signal:
  - **signal()** and **sigaction()** or
  - by using the **trap** command from the shell.

III. Signals

5) Changing Signal Dispositions (The "signal()" System Call)

- The **signal()** system call, which is described in this section, was the original API (Application Programming Interface) for setting the disposition of a signal, and it provides a simpler interface than **sigaction()**.

```
#include <signal.h>

void (* signal(int signum, void (* handler)(int))) (int);

// Returns previous signal disposition on success, or SIG_ERR on error
```

III. Signals

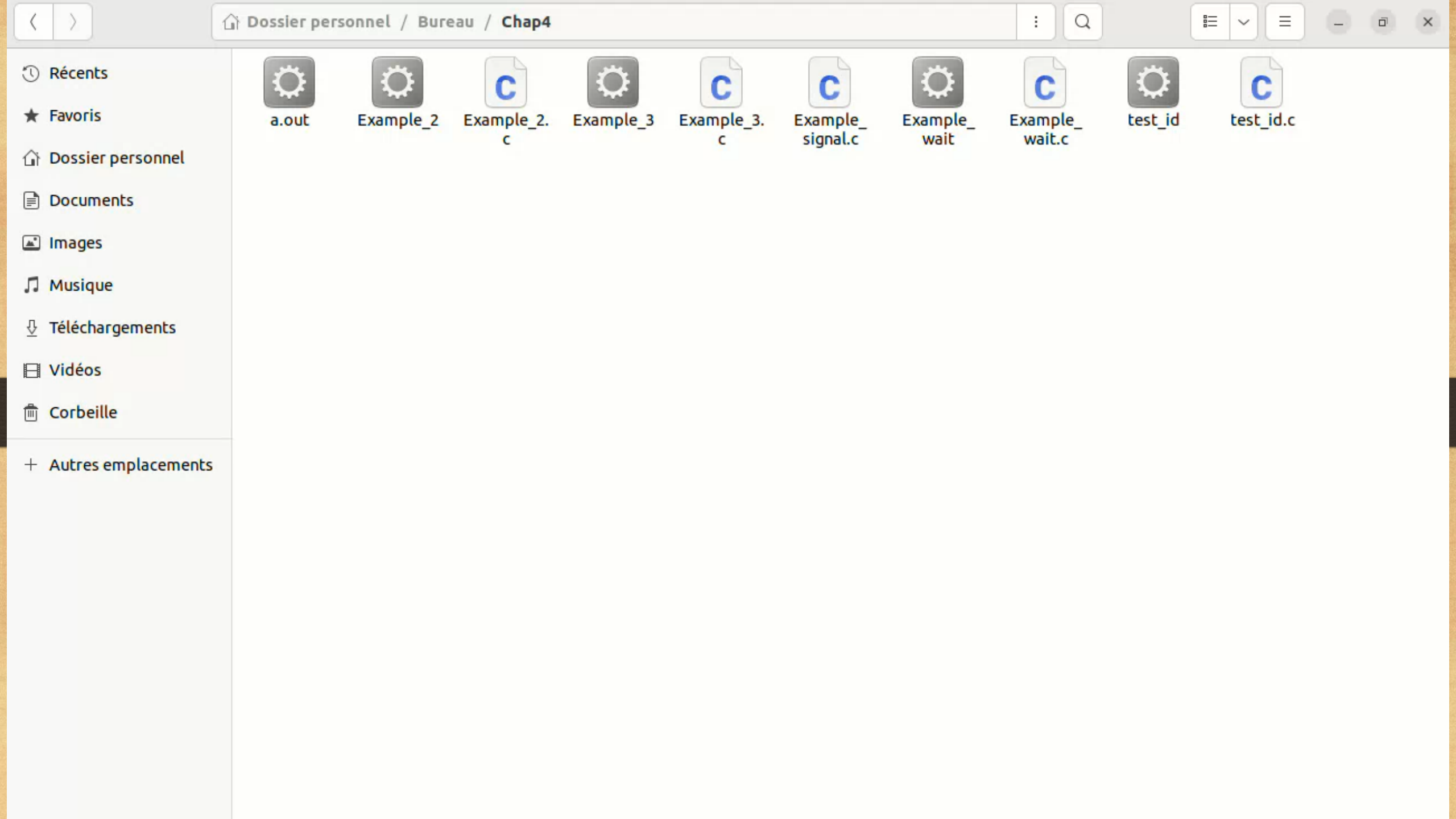
5) Changing Signal Dispositions (The "signal()" System Call)

- The function prototype for **signal()** requires some decoding.
- The first argument, **signum**, identifies the signal whose disposition we wish to change.
- The second argument, **handler**, is the address of the function that should be called when this signal is delivered.
- This function returns nothing (**void**) and takes one integer argument.
- The function returns previous signal disposition on success, or **SIG_ERR** on error.

III. Signals

5) Changing Signal Dispositions (The "signal()" System Call) Remark

```
/*Make Ctrl-C (SIGINT) inoperative*/
/*Stop this program with kill -9 pid*/
#include <signal.h>
void main () {
 signal (SIGINT, SIG_IGN);
 for (;;);
}
```


Dossier personnel / Bureau / Chap4
Récents
Favoris
Dossier personnel
Documents
Images
Musique
Téléchargements
Vidéos
Corbeille
Autres emplacements
a.out
Example_2
Example_2.c
Example_3
Example_3.c
Example_signal.c
Example_wait
Example_wait.c
test_id
test_id.c

III. Signals

5) Changing Signal Dispositions (The "sigaction ()" System Call)

```
struct sigaction{
  void (*sa_handler)();    /*SIG_DFL or SIG_IGN or ptr on handler*/
  sigset_t sa_mask;        /*Additional signals to block*/

  int sa_flags;            /*Optional indicators*/
}
```

III. Signals

5) Changing Signal Dispositions (The "sigaction ()" System Call)

```
#include <signal.h>
int sigaction (int signum, const struct sigaction* act, struct sigaction* oldact)
```

III. Signals

5) Changing Signal Dispositions ("trap" Command)

- The trap command allows the user to associate a treatment with one or more signals directly from the shell.

```
$ trap 'list of commands' sig1 sig2 ...
```

IV. Pipes (Tubes)

- Pipes **allow one group of processes to send data to another group of processes.**
- This data is **sent directly into memory without being temporarily stored** on disk, which is very fast.
- A pipe is a **communication device** that permits unidirectional communication.
- Data written to the "**write end**" of the pipe is read back from the "**read end**."
- Pipes are serial devices; the data is **always read from the pipe in the same order it was written**.

IV. Pipes (Tubes)

- Typically, a pipe is used to communicate between two **threads in a single process** or **between parent and child** processes.

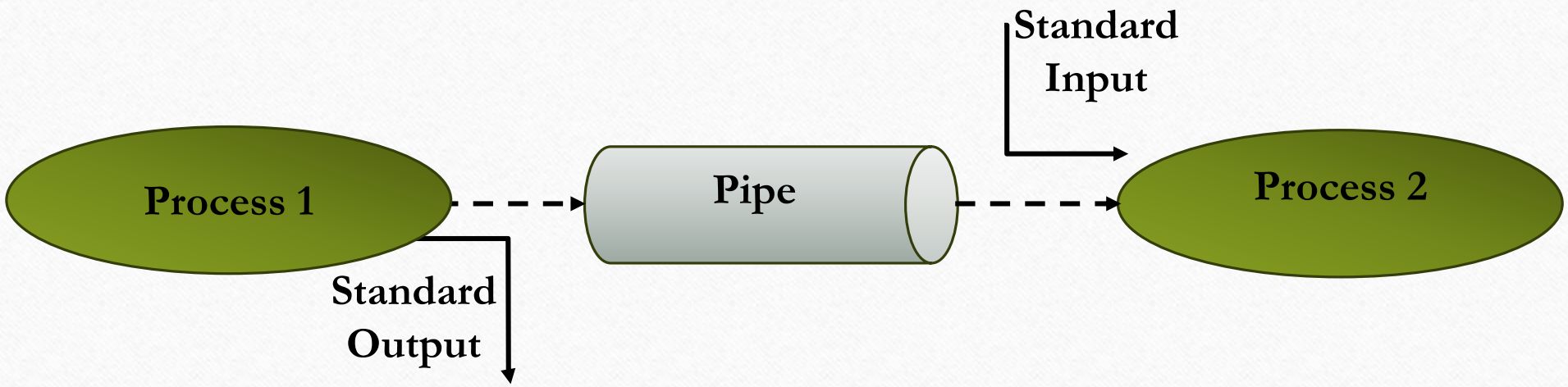


Figure 4.3: Operating Principle of an Inter-Process Communication Pipe

IV. Pipes (Tubes)

1) Pipe Characteristics

- A pipe is a means of transmitting data from one process to another (Figure 4.3). As it is implemented per file, it will be designated by descriptors and manipulated by the **read()** and **write()** primitives. But with the following special features:
  - **Communication is unidirectional**: you write at one end and read at the other (hence the name pipe). This means that at least two descriptors are needed to manipulate a pipe.
  - **Communication takes place in FIFO (first in first out) mode**. First written, first read. Commands such as **Lseek()** or other direct accesses make no sense for a pipe (tube).

IV. Pipes (Tubes)

1) Pipe Characteristics

- Whatever is read leaves the pipe permanently and cannot be reread. Similarly, what is written is permanently written.
- Transmission takes place in **continuous byte stream mode**. Consecutive transmission of the two sequences is similar and can be read as a whole or in parts (“abcd” and “efg” to “abcdefg”).
- To function, a pipe must have at least one reader and one writer. There may be several.
- A pipe has a finite capacity, and there is a producer/consumer type of synchronization between readers and writers: a reader can sometimes wait for something to be written before reading, and a writer can wait until there is space in the pipe before writing.

IV. Pipes (Tubes)

2) Creating Pipes

- The primitive creates a pipe and associates it with two descriptors rendered in the array of two integers **p**.

```
int pipe (int p[2])
```

- By definition, **p[0]** is the descriptor for reading (pipe output) and **p[1]** is the descriptor for writing (pipe input). The return value of **pipe()** is **0** on success, **-1** otherwise (too many active descriptors, too many open files, etc.).

IV. Pipes (Tubes)

2) Creating Pipes

Remark:

- There is no notion of opening a pipe (**open()**). Once created, a pipe can be used directly. On the other hand, **close()** applies to a pipe.

IV. Pipes (Tubes)

3) Reading from a Pipe: Primitive read()

A pipe can be read using the classic **read()** primitive. We use the **p[0]** descriptor rendered by **pipe()**. In the following example :

```
char buf[100];
int p[2];
                    ...
read(p[0], buf, 20);
```

- We have a request to read **20** characters from pipe **p**. The characters read are made available in the **buf** zone. Their number is the return value of **read()**.

IV. Pipes (Tubes)

3) Reading from a Pipe: Primitive read()

As a precaution, a process should systematically close descriptors it doesn't need: in this case, we need to read from pipe **p**. We must close descriptor **p[1]**.

```
close(p[1])
read(p[0], buf, 20)
```

- This avoids errors that sometimes lead to deadlock situations: processes communicate, but each wait for the other to start.

IV. Pipes (Tubes)

4) Writing to a Pipe: Primitive write()

We can write to a pipe using the classic **write()** primitive. Writing is done using the **p[1]** descriptor this time. The following sequence: is a request to write **20** characters to the open descriptor pipe **p[1]**.

```
char buf[100];
int p[2];
...
close p[0];
buf = " text to write";
write(p[1], buf, 20);
```

IV. Pipes (Tubes)

4) Writing to a Pipe: Primitive write()

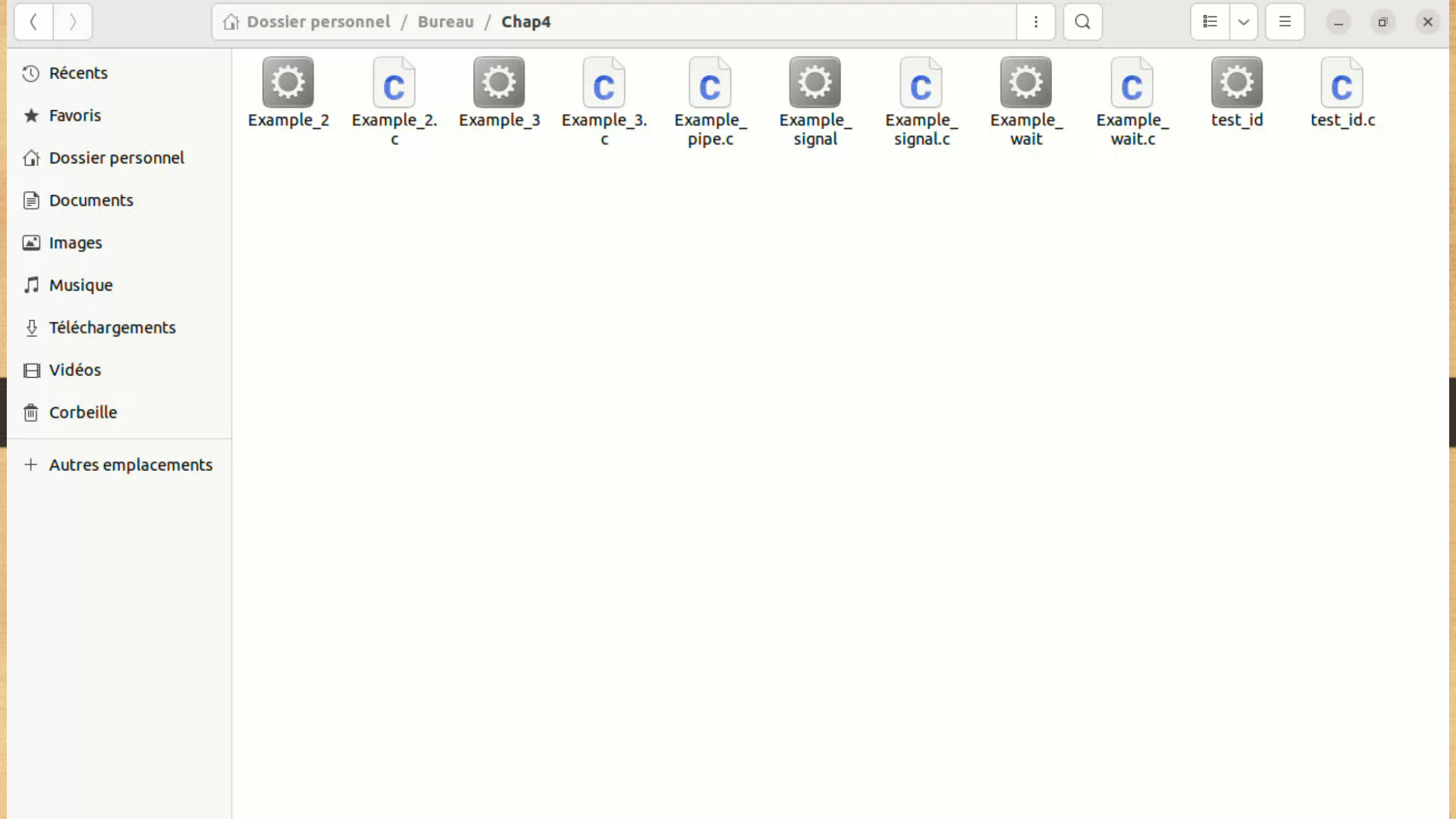
- The sequence to be written is taken from the **`buf`** zone.
- The return value of **`write()`** is the number of bytes written.
- Here too, the **`p[0]`** read descriptor has been closed.
- Writing to a pipe is atomic (everything is written or nothing is written) and does not interfere with other possible writers.
- The **20** characters written here will be consecutive in the pipe.

IV. Pipes (Tubes)

4) Writing to a Pipe: Primitive write()

Remark:

- The size of a pipe is of course limited.
- Its value is usually **4096** (constant **`PIPE_BUF`** in **`<limits.h>`**).
- If the number of characters to be written exceeds this limit, the message can be written but broken down by the system into several batches. Atomicity would then be lost.


Dossier personnel / Bureau / Chap4
Récents
Favoris
Dossier personnel
Documents
Images
Musique
Téléchargements
Vidéos
Corbeille
Autres emplacements
Example_2
Example_2.c
Example_3
Example_3.c
Example_pipe.c
Example_signal
Example_signal.c
Example_wait
Example_wait.c
test_id
test_id.c

IV. Pipes (Tubes)

5) Pipes on the Command Line

In a shell, the symbol | creates a pipe.

```
$ commande1 | commande2 | ... | commandeN
```

- For example, this shell command causes the shell to produce two child processes, one for **ls** and one for **less**:

```
$ ls | less
```

- The shell also creates a pipe connecting the standard output of the **`ls`** subprocess with the standard input of the **`less`** process. The filenames listed by **`ls`** are sent to **`less`** in exactly the same order as if they were sent directly to the terminal.

Questions ?

Exercise IV.1: (Understanding Questions)

1. **What is the difference between the `system` and `exec` functions?**

   `system` : requires a working shell, creates a new process, combines fork, wait, and exec

   `exec` : does not require a shell, replaces the calling process (does not create a new one)

Exercise IV.1: (Understanding Questions)

2.In Unix, Which system call creates the new process?

In UNIX, a new process is created by **`fork()`** system call.

Exercise IV.1: (Understanding Questions)

3. What is a zombie process?

A zombie process is a process that has finished executing but has not completely terminated, usually because its exit status has not been collected, so its resources (at a minimum, its process control block) have not been deallocated.

Exercise IV.1: (Understanding Questions)

4. **Why all processes are technically zombie processes for at least a brief time. (Hint: think about which of the five process states zombie processes must exist in.)**

Any process in the terminated state is technically a zombie, since it has finished executing but its resources have not deallocated.

Exercise IV.1: (Understanding Questions)

5. What is an orphan process?

An orphan process is one for which the parent terminate first without collecting the child's exit status, making the child an orphan.

Exercise IV.1: (Understanding Questions)

6. **Why UNIX systems reassign processes to a new "parent" rather than simply terminating a process as soon as it's orphaned.**

   An orphaned process might still have useful work to do, but something must collect its exit status once it does terminate to ensure the orphan completed without errors.