CHAPTER 3.1

Unix / Linux Processes (Lab)

IV.1. 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. 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.

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.

IV.2. 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.

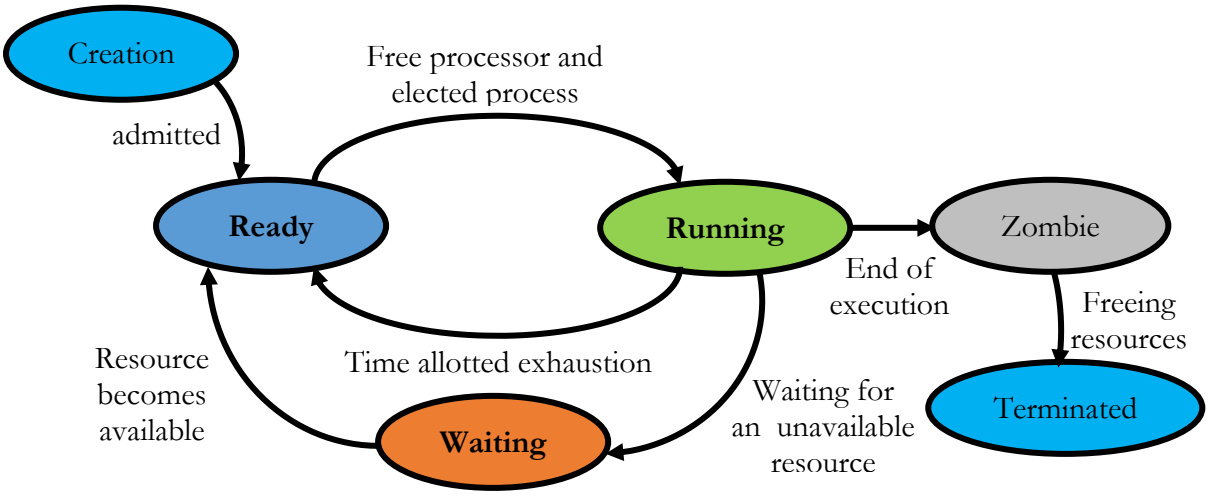

Figure 4.1 : Process Life Cycle

IV.2.1. 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.

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
```

IV.2.2. 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.

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.*/

Example 1: (test_id.c file)

```
#include <stdlib.h>
void main() {
    printf("I am the process %d of father %d and group %d\n",
                getpid(), getppid(), getpgrp()) ;
}
```

Result of execution

```
$ ps
PID TTY TIME COMMAND
6658 ttyp5 0:04 csh
$ test_id
I am the process 8262  of father 6658  and group 8262
```

Note that the parent of the process running **`test_id`** is **`csh`**.

IV.2.3. Creating Processes

Two common techniques are used for creating a new process. The first is relatively simple but should be used sparingly because it is inefficient and has considerably security risks. The second technique is more complex but provides greater flexibility, speed, and security.

IV.2.3.1. 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.

Example 2:

This program invokes the "**ls**" command to display the contents of the root directory, as if we typed "**ls -l /**" into a shell.

```
#include <stdlib.h>
int main (){
    int return_value;
    return_value = system ("ls -l /");
    return return_value;
}
```

The system function returns the exit status of the shell command.

- If the shell itself cannot be run, system returns **127**;
- If another error occurs, system returns **–1**.

Because the system function uses a shell to invoke our command, it is subject to the features, limitations, and security flaws of the system's shell. Therefore, it's preferable to use the **fork** and **exec** method for creating processes.

IV.2.3.2. 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.

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.

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

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**.

```
pid_t pid=fork();
switch(pid){
      case -1:                    /* Creation error */
        ...
      case 0:                     /* Child program */
        ...
      default:                    /* Parent program */
        ...
}
```

Example 3:

Note that the first block of the **"if"** statement is executed only in the parent process, while the **"else"** clause is executed in the child process.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main () {
  pid_t child_pid;
  printf ("the main program process ID is %d\n", (int) getpid ());
  child_pid = fork ();
  if (child_pid != 0) {
   printf ("this is the parent process, with id %d\n", (int) getpid ());
   printf ("the child's process ID is %d\n", (int) child_pid);
  }
  else
   printf ("this is the child process, with id %d\n", (int) getpid ());
  return 0;
}
```

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.

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.
- 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.

IV.2.3.3. 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.

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.

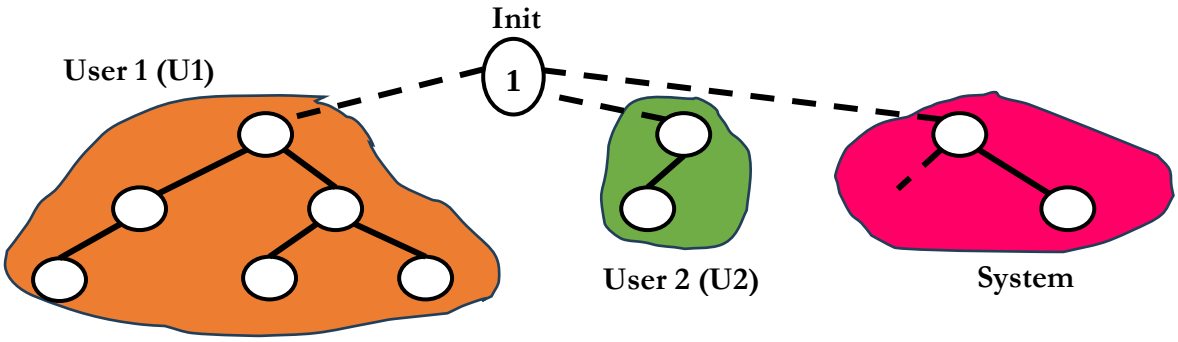

Figure 4.2 : Process Hierarchy

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
```

IV.2.3.4. 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.

```
#include <stdlib.h>
void exit (int status);
```

As it can be explicitly eliminated by another process, by sending a **kill** signal (see below).

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**.

```
pid_t pid=fork();
switch(pid){
     case -1:          /* Creation error */
       ...
       exit(1);
     case 0:           /* Child program */
       ...
       exit(0);
     default:          /* Parent program */
       ...
       exit(0);
}
```

IV.2.3.5. 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.

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**. 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.

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);
```

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**.

```
pid_t pid=fork();
switch(pid){
    case -1:            /* Creation error */
       ...
    case 0:             /* Child program */
       printf("Child process %d\n", getpid());
       exit(10);
    default:            /* Parent program */
       printf("Parent process %d\n", getpid());
       pid_t id = wait (&status);
       printf("End Child process %d\n", id);
       exit(0);
}
```

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.

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.

```
pid_t id;
pid_t pid=fork();
switch(pid){
     case -1:                          /* Creation error */
        ...
     case 0:                           /* Child program */
        ...
     exit(10);
     default:                          /* Parent program */
        ...
        while((id = waitpid(pid, &status, WNOHANG))==0)
           printf("Child process not yet completed \n");
        printf("End Child process %d\n", id) ;
        ...
}
```

Output Code Returned by wait() and waitpid()

To extract the return code of status's child process, we first need to check that the process has terminated normally.

- To do this, we use the **WIFEXITED(status)** macro, which returns **true** if the child process has terminated normally.
- Then, to obtain its return code, we use the **WEXITSTATUS(status)** macro.

```
pid_t id;
pid_t pid=fork();
switch(pid){
    case -1:                          /* Creation error */
         ...
    case 0:                           /* Child program */
         ...;
       exit(10);
    default:                          /* Parent program */
         ...
       while((id = waitpid(pid, &status, WNOHANG))==0)
         printf("Child process not yet completed \n");
       if(WIFEXITED(status))
         printf("child %d terminated by exit(%d)\n", id,
         WEXITSTATUS(status));
         ...
}
```

IV.3. Types of Processes

Processes with in Unix are classified into three general categories—as interactive processes, noninteractive processes and daemons.

IV.3.1. 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. 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.

- 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

IV.3.2. 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.

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. 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.

Example 4:

Suppose we want to execute a command but do not want to wait for its completion, i.e. we want to be able to issue other commands in the meantime. We can do this by specifying that the command be executed in the background.

There are two ways to do this.

The first is to specify that it be a background process when we submit it, which we can do by appending an ampersand ('**&**') to the end of the command. Suppose we have a very large program, which will take a long time to compile. We could give the command :

```
$ gcc bigprog.c &
```

The C-shell will let us know what the **pid** is for this background process (so that we can later track it using **ps**, or **kill** it), but will also give us your regular prompt, inviting me to submit new commands while the other one is running.

But what about the compiler error messages?

We hope we don't have any : but if we do have some, we don't want them to be interspersed with the output of other commands we are running while the compile is executing. To avoid this, we redirect the error messages:

```
$ gcc bigprog.c >& errorlist &
```

All error messages will now be sent to the file '**errorlist**', which we can view later

Another good example is when we start a window during a X session. We would like to start the window from an existing window, but we still want to be able to use the original window. We execute the command

```
$ xterm &
```

This will start a new window, and allow us to keep using the current window.

The other way to put a job in the background is to stop it, using C-z as described earlier, and then use another command, **bg**, to move the process to the background. Suppose we started our long-running compile,

```
$ gcc bigprog.c
```

but we forget to append the ampersand. We can type control-z to suspend/stop the job, and then type '**bg**' to resume the job in the background, allowing us to submit other commands while the compilation takes place. Unix will tell us when the background job has completed, with a statement like :

```
[1]    Done    gcc bigprog.c
```

By the way, if you log out, whatever background processes you have running at the time will not be killed; they will continue to run.

IV.3.3. 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.

IV.4. 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.

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.

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**.

IV.4.1. 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>`**.).

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. The following table lists some of the main 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

IV.4.2. 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)

IV.4.3. 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()`**.
- **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.

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**. 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.

IV.4.4. 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.
- **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.

IV.4.5. 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.

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
```

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.

Example 5:

The **`handler`** function displays the number and symbolic name of the signal being diverted. If the signal is **`SIGINT`**, the program is stopped.

The **`strsignal`** function returns the symbolic name of signal number **`sig`**.

Code :

```
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <stdlib.h>
//Routing function
```

```
void handler (int sig) {
   printf("Well Received %d %s\n", sig, strsignal(sig));
   if (sig == SIGINT) {
      printf ("Voluntary End \n");
      exit (1);
   }
}
void main () {
   signal (SIGINT, handler);       //Ctrl-c : signal 2
   signal (SIGQUIT, handler);      //Ctrl-\ : signal 3
   signal (SIGTSTP, handler);      //Ctrl-z : signal 20
   //SIGKILL is not routable
   if (signal (SIGKILL, handler) == SIG_ERR)
       perror("SIGKILL");

   for (;;);
}
```

Remark

The processing function can be replaced by one of the following constants:

- **SIG_IGN**, which indicates that the signal should be ignored.
- **SIG_DFL**, to restore the default action for the signal.

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

The "sigaction ()" System Call

The **sigaction()** primitive associates a signal with a diversion context, represented by a **sigaction** structure.

```
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*/
}
```

The **sa_handler** field is the only mandatory one; it's a pointer to the function that will act as handler.

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

Example 6:

```
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <stdlib.h>
void handler (int sig) {
   printf ("Well received %d %s\n", sig, strsignal(sig));
   if (sig == SIGINT) {
      printf ("Fin volontaire\n");
      exit (1);
   }
}
void main () {
   struct sigaction act;
   act.sa_handler = handler;
   sigaction (SIGINT, &act, NULL);     //Ctrl-c : signal 2
   sigaction (SIGQUIT, &act, NULL);    //Ctrl-\ : signal 3
   sigaction (SIGTSTP, &act, NULL);    //Ctrl-z : signal 20
   //SIGKILL est non déroutable
   if (sigaction(SIGKILL, &act, NULL) == -1)
      perror ("SIGKILL");
   for (;;);
}
```

The "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 ...
```

Example 7:

Command:

```
$ trap 'rm /tmp/* ; exit' 2 3
```

executes the **rm /tmp/* ; exit** command when **SIGINT** or **SIGQUIT** signals are received.

To disable one or more signals, simply pass the empty string as an argument to the trap command.

Command:

```
$ trap '' 2 3 15
```

hides the **SIGINT**, **SIGQUIT** and **SIGTERM** signals.

To restore the default action for one or more signals, simply pass the signal numbers as the only argument to the **trap** command.

Command:

```
$ trap 2 3 15
```

restores the default actions associated with **SIGINT**, **SIGQUIT** and **SIGTERM** signals.

IV.5. 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.

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

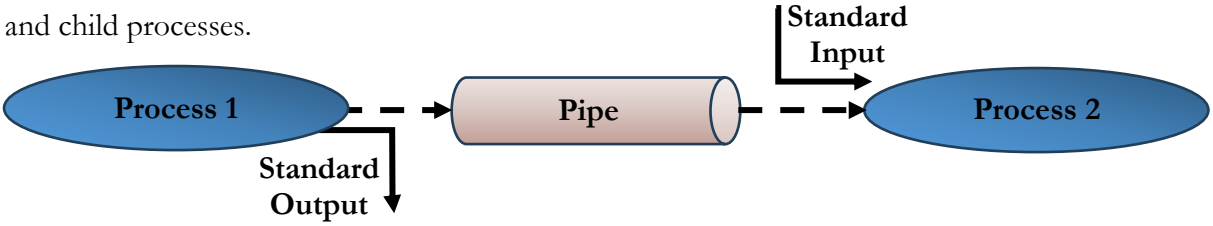

Figure 4.3: Operating Principle of an Inter-Process Communication Pipe

IV.5.1. 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).
- 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.5.2. 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.).

Example 8:

```
int p[2];
if ( pipe(p) == -1 )
   fprintf(stderr, "Impossible to open pipe \n");
```

As already mentioned, a pipe must be created before the processes that will use it are launched. This is the case, for example, in Parent/Child or Child/Child cooperation, when the parent process creates the pipe before **`fork()`**.

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.5.3. 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()`**.

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.5.4. 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);
```

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.

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.

Example 9:

The parent sends the child a message in a pipe. Message sent as a block and read character by character, then printed.

```
#include <stdio.h>
char message[25] = "It comes from a tube";
main()
{
    /*  communication PARENT --> CHILD by pipe  */
    int p[2];
    int pipe(int[2]);
    if (pipe(p) == -1) {
       fprintf(stderr, "pipe opening error \n");
       exit(1);
    }
    if (fork() == 0) {          /* CHILD */
        char c;
        close(p[1]);
        while (read(p[0], &c, 1) != 0)
             printf("%c", c);
        close(p[0]);
        exit(0);
    }
    else {                      /* PARENT */
        close(p[0]);
        write(p[1], message, 24);
        close(p[1]);
        exit(0);
    }
}
```

Remark :

- Stop **`read()`** on zero return value (performed by **`close(p[1])`**; in the parent process).

IV.5.5. 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.