TP n°2: Process management (2)

1. Objectives

At the end of this practical course based on the Linux system, the student should be able to understand the concept of synchronization between processes.

2. Synchronization between processes (parent and child)

In this phase, we will use the following functions:

a) exit(i):

This function Terminates a process, *i* is a byte (so possible values: 0 to 255) returned in a variable of type *int* to the parent process.

b) wait(&Status):

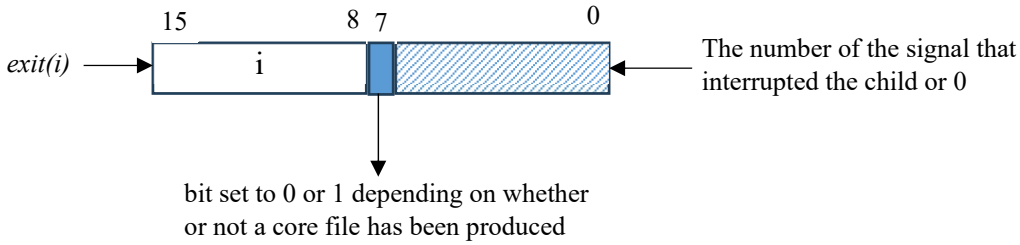
Puts the process on hold until one of its child processes has finished. When a process terminates, the SIGCHILD signal is sent to its parent. Receiving this signal switches the parent process from the blocked state to the ready state. The parent process therefore exits the wait function. The return value of wait is the number of the child process that has just terminated.

When there is no child process to wait for, the function "*wait*" returns -1.

Each time a child terminates, the parent process exits wait, and can consult "***Status***" to obtain information about the child that has just terminated.

"State" is a pointer to a two-byte word:

- The high byte contains the value returned by the child (i of the function *exit(i)*),
- The low byte contains 0 in the general case. However, if the child process terminates abnormally, this low byte contains the value of the signal received by the child. In addition, this value is increased by 80 in hexadecimal (128 decimal), if this signal has caused the process memory image to be saved in a core file. The following diagram summarizes the contents of the word pointed to by State at the output of wait.



c) **waitpid**()

The *waitpid()* system call suspends execution of the calling process until a child specified by the pid argument changes state. By default, *waitpid()* only waits for terminated children, but this behavior can be modified by the options argument, as described below.

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

The *pid* value can be one of the following:

- " <-1 " : meaning wait for any child process whose process group ID is equal to the absolute value of *pid*.
- "-1" : meaning wait for any child process. In this case, *waitpid(-1, &status, 0)* is equivalent to *wait(&status)*.
- *"0":* meaning wait for any child process whose process group ID is equal to that of the calling process.
- ">0": meaning wait for the child whose process ID is equal to the value of *pid*.

The value of *options* is an OR of zero or more of the following constants:

- **WNOHANG:** return immediately if no child has exited.
- **WUNTRACED:** Receive information about blocked children blocked if you have not received it yet. The status of tracked children is provided even without this option.
- **WCONTINUED:** Also return whether a stopped child process has been restarted by the SIGCONT signal.

If *status* is not NULL, **wait**() and **waitpid**() store status information in the *int* to which it points. This integer can be inspected with the following macros:

- **WIFEXITED(status)** : returns true if the child process was terminated by exit(). It is used to check whether the child has terminated with exit().
- **WEXITSTATUS(status)**: to extract the return code. This macro should only be employed if *WIFEXITED* returned true.
- **WIFSIGNALED(status)** : returns true if the child process was terminated by a signal. It is used to check whether the child has been terminated by a signal.
- **WTERMSIG(status)** : returns the number of the signal that caused the child process to terminate. This macro should only be employed if *WIFSIGNALED* returned true.
- **WCOREDUMP(***status***)** : returns true if the child produced a core dump. This macro should only be employed if *WIFSIGNALED* returned true.

d) Return value

- **wait()**: on success, the PID of the terminated child process is returned; on error, the return value is -1.
- **waitpid()**: if successful, the call returns the PID of the child process whose state has changed; if WNOHANG is used and one (or more) children specified by PID exists, but has still not changed state, the return value is 0. In the event of an error, -1 is returned.

3. Running an external program

A process can run an executable program located in a file on disk using one of the *exec()* system calls. The code of the program on disk replaces the code of the current process and so a call to *exec()* never returns, except in the event of an error where it returns -1 and sets the *errno* system variable.

There are many versions of exec functions (execl, execv, execle, execve, execlp, execvp) but here we only give details of the simplest to use, for the others go to: man 2 exec.

Example : execlp()

```
int execlp(char *path, char *arg0, char *arg1, ..., char *argn, NULL)
```

In this version, "*path*" is a character string giving the name of the file containing the program to be executed (searched for in the "PATH" paths, ***arg0*** by convention contains the name of the file (with no directory) and the other ***args*** are the parameters of the program to be executed. The list of parameters must end with a null pointer (NULL).

Examples :

```
execlp("ps", "ps", "-l", NULL);
```

```
execlp("/bin/ls", "ls", "-l", NULL);
```

Exercises

Exercise n°1

1) Write a program that creates N child processes in parallel, then waits for them to terminate. Each process created must display its pid, the pid of its parent and its creation order number.
2) In this program, each of the children must return its creation number to the termination. The main process must display the output value of each of the children and the pid of the child whose termination it has just detected.

Exercise n°2

Write a program using the functions *fork()*, *exit()* and *wait()* to transmit an ASCII character from the child process to the parent process. The child process reads the character using the function *getchar()*. The parent process displays the ASCII code of this character and transforms it into uppercase using the function *toupper(char c)*.

Exercise n°3

Use the compiler "c" under Linux "gcc" to compile the following program.

#gcc -c filename.c

#gcc -o exe-name filename.o

#./exe-name

```
#include<stdio.h>
#include <unistd.h>

int main(void){
int x,pid,s;

pid =fork();
if (pid){
          wait(&x);
          s = x>>8;
          s = s*s;
          printf("The parent process calculates the square of x :%d\n",s);
          exit(0);
          }
else {
       printf("entrer la valeur de x : \n");
       scanf("%d",&x);
       exit(x);
      }
return 0;
}
```

1) Indicate the result displayed by this program.
2) Indicate the relation between *exit()* and *wait()*.

Exercise n° 4

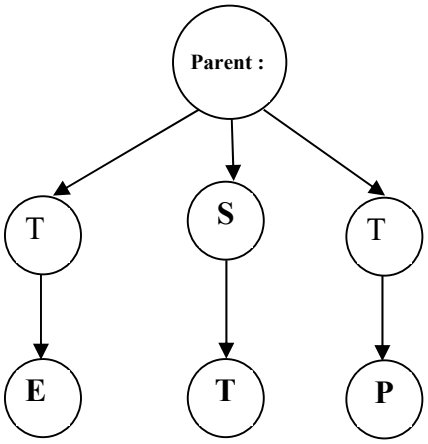
Write a program whose parent, after creating three children (f1, f2, f3), waits for these three children to return before performing the calculation $3 \times 10 + 5$.

The data:

- The child f1 returns the value 5;
- The child f2 returns the value 10;
- The child f3 returns the value 3.

Exercise n° 5

1) Write a program in C language to create the following tree structure:



2) Synchronize the different processes to display: Parent: TEST TP using the wait(0) and exit(0) functions.